UNIX PROGRAM DESCRIPTION

Program Generic PG-1C300 Issue 2

Published by the UNIX Support Group

January 1976

A few remarks from April 2022:

This edition of the document has been reconsutructed from a scan found in a TUHS archive. Its origin or copyright status is unknown to me. Please do not distribute without a permission from the copyright holder.

While the original document contains a handful of typoes (preserved here), it is guarranteed to also contain errors not present in the original. A couple of pages in the scanned document were in wrong order and couple more were missing. The missing material is indicated as such in this edition and the ordering has been fixed.

If you know anything about the origin of this document, its author, have the missing parts or spotted an error etc., please reach out to Lubomir Rintel < *lkundrak@v3.sk*>.

The original document was obtained from:
< *https://www.tuhs.org/Archive/Distributions/USDL/unix_program_description_jan_1976.pdf* >

CONTENTS

COMMON SYSTEMS

UNIX OPERATING SYSTEM

OPERATING SYSTEM

This index lists the authorized issues of the sections that form a part of the current issue of this specification.

ISSUE 1 1/30/76

1. GENERAL

This document describes functions contained in pidents from PR-1C301-01 as follows:

ALLOC01	DISK ALLOCATOR
CLOCK01	SYSTEM CLOCK
CONF01	CONFIGURATION TABLE
FIO01	FILE I/O
IGET01	I-NODE INTERFACE
LOW01	LOW CORE CONFIGURATION
MAIN01	MAIN
MCH01	MACHINE INTERFACE CODE
NAMI01	FILE NAME STRING INTERPRETER
PRF01	PRINT FACILITY
RDWRI01	READ/WRITE I-NODE
SIG01	SIGNAL
SLP01	PROCESS SWITCHING
SUBR01	SYSTEM SUBROUTINES
SYS101	SYSTEM CALL INTERFACE 1
SYS201	SYSTEM CALL INTERFACE 2
SYS301	SYSTEM CALL INTERFACE 3
SYS401	SYSTEM CALL INTERFACE 4
SYS501	SYSTEM CALL INTERFACE 5
SYSENT01	TABLE OF SYSTEM ENTRY POINTS
TEXT01	SHARED PROGRAM
TRAP01	TRAP HANDLER

2. PROGRAM CONVENTIONS

A. System calls are made with the first argument in register R0. When the system call is made, the contents of register R0 are moved to the per user control block (user.h) in the variable called u.u_R0. The remaining arguments of a system call are moved into the per user control block array u.u_arg (this means u.u_arg[0] is the second argument).

B. Arguments or results of executing some functions are often left behind in the per user control block. For example, nami.c/namei decodes a pathname into an inode pointer. In the process, a pointer to the inode of the parent directory is left in u.u_pdir. This means it is eady to make a directory entry for a file since the inode for the directory is available. (See the documented header user.h in PR-1C301.)

C. Inodes are always locked during manipulations to prevent simultaneous update by two processv:. The procedure is to always lock and increment the usage count of an inode even if it turns out that a user does not have access to that file. At the end of processing of the inode, the usage count is reduced by 1 if there was an error, and in either success or failure, the inode is unlocked.

D. Error processing that reflects errors back to the user are set in the per user control block error flag (u.u_error). These error conditions can be referenced by the user program through the external variable "errno". (See Section 2 of Programer’s Manual for list of error conditions.)

E. If I/O processing is to be done on a device, the particular driver for that device must be called. Devices are known by major and minor numbers stored in an inode. The system calls the particular device driver indirectly through the major device number. A block switch table and character switch table are defined at system generation time. The major device number is used as a displacement into this table and the appropriate routine is called. For example, the code:

    (*bdevsw[maj].d_close)

    will call the close entry point for the driver associated mith major device "maj".

alloc

CALL

alloc (dev)
int dev;

RETURNS

A block number of allocated block if successful. A NULL is unsuccessful.

SYNOPSIS

Alloc allocates disk blocks from the free list of the associated file system. Companion routine with free.

DESCRIPTION

Alloc allocates disk blocks on device "dev". Alloc.c/getfs is called to get the pointer to the in-core superblock for the file system on device "dev". If the superblock is locked because of replenishing of the free list then alloc will sleep until it becomes unlocked.

The free list of blocks is maintained as a linked list of tables of 100 entries (99 free block pointers and one pointer to the next member of the linked list). The last table contains a zero pointer to indicate the end. S_nfree is a pointer into first table of 100 entries which is kept in memory in the superblock. When s_nfree is zero, alloc replenishes the memory table with the next table in the linked list. If the linked list is depleted an error indicator is set. Alloc returns the block number pointed to by s_nfree.

badblock

CALL

badblock (alp, abn, dev)
int aip,abn,dev;

RETURNS

If no bad blocks return 0, otherwise print BAD BLOCK message on console and return 1.

SYNOPSIS

Checks that a block number is in the range between the ilist and the end of the file system. In other words make sure that the block can be used in block allocation.

DESCRIPTION

If the block number "abn" on the file system pointed to by "afp" is less than isize+2 (isize is the size of the ilist and block 0 is the boot program and block 1 is the superblock) or greater than fsize (the size of the file system) then alloc.c/prdev is called to print the bad block message for device "dev". Otherwise return 0 as indicator of success.

free

CALL

free (dev, brio)
int dev, bno;

RETURNS

None

SYNOPSIS

Free places the block number "bno" back on the free list of the file system on device "dev". Companion routine with alloc.

DESCRIPTION

Alloc.c/getfs is called to get the pointer to the in-core superblock for the file system on device "dev".

The free list of blocks is a linked list of tables of 100 entries (99 free blocks and one pointer to. the next table). When blocks are freed, they are added to this linked list. The first table of 100 entries is kept in memory in the superblock. Block allocation/deallocation takes place in this table at the place pointed to by s_nfree. When the table is full (s_nfree = 100) it is written to disk, added to the linked list, and the incore table is emptied.

getfs

CALL

getfs (dev)
int dev;

RETURNS

A pointer to the in-core superblock.

SYNOPSIS

Getfs maps a device number, "dev", into a pointer to the in-core superblock associated with the file system on that device.

DESCRIPTION

Getfs searches through the Mount Table for a matching device number, "dev". The Mount Table is composed of three word entries:

1. A device pointer
2. A pointer to the buffer containing the superblock
3. A pointer to the i-node entry for the mount point.

If the device is not present in the Mount Table a PANIC "no fs" occurs. (This cannot happen). Otherwise, the pointer to the buffer containing the superblock is returned.

ialloc

CALL

ialloc (dev)
int dev;

RETURNS

A pointer to the incore, allocated i-node.

SYNOPSIS

Allocation of i-numbers and i-nodes for use in file creation. Companion routine with ifree.

DESCRIPTION

Iafloc allocates i-nodes for file creation and refills the free list of i-numbers when it becomes empty.

Alloc.c/getfs is called to get a pointer to the incore superblock of the file system for device "dev".

If the free list of i-numbers (s_ninode) is locked because it is under replenishment then ialloc sleeps until it becomes unlocked.

If the free list is empty, ialloc locks it, then a linear search of the ilist on disk device "dev" is made looking for 100 free i-nodes. the i-number of each free i-node is placed into the free list. If the entire ilist contains allocated i-nodes then a PANIC out of i-nodes occurs. The replenishment is complete when the free list is full or all available ilist i-nodes have been put into the free list. The i-number pointed to by s_ninode, (the pointer to the first available entry in the free list) is used by iget.c/iget to load the associated i-node into the System INODE Table. If the i-number in the free list points to an i-node that is already allocated a message BUSY I (busy i-node) is printed and iget.c/iput is called to release the inode.

When an unallocated i-node is found, its mode and address pointers are zeroed and a pointer to this i-node is returned. A flag (s_fmod) for the incore superblock is also set to indicate that the superblock was modified, so on a subsequent update of the disk (in which all modified superblocks are written to disk) this superblock will be written.

ifree

CALL

ifree (dev, ino)
int dev, ino;

RETURNS

None

SYNOPSIS

Free the specified i-node, and associated inumber, "ino", on the file system on device "dev". Companion routine with ialloc.

DESCRIPTION

A free list of 100 i-numbers are kept in the incore superblock. When an i-node is freed, its inumber is added to the free list (s_inode) at the entry pointed to by s_ninode (the next free slot).

When the free list is full, additional deallocated i-numbers are ignored, since their associated inodes are marked as unallocated and are written back to the ilist where they can be picked up in a subsequent searci of the ilist (see alloc.c/ialloc).

Alloc.c/getfs is called to get a pointer to the incore superblock of the file system on the device "dev". If the free list is locked or full (sninode = 100) then the i-number is abandoned.

Otherwise, the i-number is placed at the entry pointed to by s_ninode and a flag (s_fmod) is set to indicate that the superblock was modified.

unit

CALL

iinit()

RETURNS

If the superblock of the root directory cannot be read a panic results.

SYNOPSIS

Mounts the root file system by building an entry in the Mount Table.

DESCRIPTION

Iinit is called once during UNIX initialization to read the superblock of the root device into memory and place pointers into the Mount Table for this superblock. If an error occurs during the

reading of the root device (rootdev is the entry defined in the configuaration table conf.c) then a PANIC UNIT is issued which causes the system to halt.

prdev

CALL

prdev(str, dev)
int str, dev;

RETURNS

None.

SYNOPSIS

Print out error messages on system console.

DESCRIPTION

Print the string "str" on the system console with the major and minor device numbers for the device pointed to by "dev".

update

CALL

update()

RETURNS

None

SYNOPSIS

Update is the system routine that writes all the changed superbiocks and i-nodes back to disk. In normal system operation this occurs whenever the SYNC command is issued, or by the UPDATE program every 30 seconds.

DESCRIPTION

Because update is initiated by user programs, it maintains a lock to prevent simultaneous updating before one complete pass is done. For each non-zero entry in the Mount Table that has the modified flag (s_fmod) set, is not locked for replenishing the free list (s flock) or ilist (s_ilock) manipulation, and is not read only (s_ronly) the superblock is written to the disk.

Each i-node in the System INODE Table (all those associated with open files) are examined to see if they are locked for modification by some other function. If an i-node is unlocked, it is locked by update to prevent other changes while updating, and iget.c/updat is called to write the i-node to the disk ilist if it was modified. Pipe.c/prele is then called to release the locked i-node.

clock

CALL

clock (device,sp,r1,newps,r0,pc,oldps)

RETURNS

No value is returned.

SYNOPSIS

The clock interrupt handler. Supplies system timing.

DESCRIPTION

Clock.c/clock is the clock interrupt handler. It maintains the system clock and any time dependent services supplied by the system. As there are a number of these software services the interrupt processing for the clock will be described first.

There are currently two clocks available for the PDP-11 series of computers; the KW11-L and KW11-P clocks. The KW11-L is simply a line frequency clock, while the KW11-P is a programmable clock which can count at 10KHz, 100KHz or on the basis of an external trigger in addition to line frequency. UNIX can accomodate either of these clocks, however, only the line frequency option is used. The clock that is present on the system (the clocks have different Unibus addresses) is determined by the main.c/main function when the system is initialized. The Unibus address of the clock is placed in the external variable "lks". If no clock is present on the system or if the clock is malfunctioning when the system is booted, the system panics ("PANIC NO CLOCK"). The main.c/main function is the first to turn on the clock so that interrupts are generated once every sixtieth of a second. The clock interrupt handler thereafter reenables succeeding interrupts so that interrupts occur. (The clock never stops counting so that no delay is encountered by the need to reenable interrupts.) The clock interrupt handler generates an interrupt at bus request level 6, which is higher than that of all hardware controllers on the system. Only traps have a higher priority (7). In line with the fact that interrupt handlers on UNIX are non-reentrant, the clock interrupt is processed at the same priority that the interrupt request was generated. The bits that must be set in the clock status register to reenable clock interrupts are:

1. Bit 6 reenables interrupts. This is the only bit that need be set for the KW11-L.

2. Bit 3 selects repeat interrupt mode (KW11-P only).

3. Bits 2 and 1 select the mode. If bit 2 of the bpair is set, the KW11-P will count at Line frequency.

4. Bit 0 turns the counter in the clock on. (KW11-P only)

The clock interrupt handlers are called is called in the same way that other interrupt handler from the mch.s/call interface. It has arguments (on the stack frame built by mch.s/call) available to it so that it can check the Previous Mode of the processor, the Program Counter, etc.

The following software functions are performed by the clock interrupt handler:

Clock/clock.c updates the system’s notion of the time of day. The time of day is kept in a two word array (a long integer) "time[]" with the least significant bit (in "time[1]") being in units of seconds. As a clock interrupt is genetated once every sixtieth of a second, a count of the number of sixtieths of a second is kept in "lbolt". The "lbolt" is incremented on every interrupt and when it reaches 60, the system time is updated. The time of day must, however, be initialized to the proper yearly value via the date system call.

The clock interrupt handler keeps track of the amount of time a process spends in User mode and Kernel mode. This time is kept in two separate locations ("u_time" and "u_stime") in each process’s U block. The time is kept in sixtieths of a second.

The clock interrupt handler wakes up processes that are delaying execution (i.e., processes that have made the sleep system call). The external variable "tout[]" which has the same form as the time of day, is set to the date that the earliest sleeping process is to be awakened. When the time has elapsed, all processes that have issued the sleep system call (the address of "tout[])" is used as the synchronizing event) are awakened.

The clock interrupt handler maintains an event called the *lightning bolt* which is used by some drivers to achieve long delays. The address of "lbolt" is used as the synchronizing event and all processes that sleep on this event are awawakened when the clock reaches a 4 second interval.

The magnetic tape drivers use the lightening bolt to wait for gap shut down when closing the device.

The age of a process is updated by the clock interrupt handler. The "p_time" entry for each process in the Process Table contains the length of time in seconds that a process has been in memory or on the swap device. Once every second these ages are updated.

Since the Scheduler uses the age of a process as its chief criteria in swapping a process into or out of memory, the clock notifies the Scheduler when these ages change. The Scheduler is notified only if it is waiting ("runin") for available memory to bring a process into memory.

The console display is updated once every clock interrupt by calling mch.s/display.

In line with profiling a process, the clock handler aids the profiler by calling mch.s/incupc when profiling is selected ("u_prof[3]" turns profiling on).

The clock interrupt handler provides for the delayed execution of a function. This is done by maintaining a list, "callout[]", of functions to be executed after a time period. Each entry in "callout[]" contains a pointer to a function ("c_fund"), an argument ("c_arg") to be passed to the function and the relative time "c_time") at which the function is to be executed. The time entry is in sixtieths of a second so that the execution of a function may be delayed up to 32K sixtieths of a second (about 9 minutes). Clock.s/timeout inserts an entries into the "callout[]" list, however, the clock interrupt handler must maintain the list. The time values "c_time" that are inserted in the array are times relative to that of the previous entry. Relative times are used so that the clock interrupt handler does not have to scan the entire array to update each entry. Rather, only the first entry need have it's time decremented. The appropriate function is called by the clock handler when the (relative) time value has reached zero. At this time, the entry must be removed from the "callout[]" list and the list is compressed (all entries are moved down). Since delayed function execution is typically used by character and block device interrupt handlers, some precautions must be exercised in executing them. In particular, since interrupt handlers are not reentrant and since the clock interrupt may cause the stacking of one of these interrupts, it is dangerous to allow a delayed function to be executed when the clock has caused the stacking of interrupts. The delayed function might attempt to access some list whose linkages were only partially established, etc. To avoid these complications and to prevent restrictions being placed on the delayed functions, the clock handler does not execute a delayed function if the clock interrupt occurs while the processor's priority is nonzero. (That is, the processor was already engaged in processing an interrupt, or the processor was in the midst of a critical region of software. The processor's priority is always zero when the processor is in User mode so that delayed functions can always be executed if a user is interrupted.) This means that the function must wait at least until the next clock interrupt before it can be executed. In order to prevent the remaining functions from being delayed because of this, the relative time must be allowed to become negative and the first non-zero time in the list must be decremented to insure that the other functions in the array do not incur the delay. When a clock interrupt finally occurs while the processor's priority is zero (processor in User mode or in Kernel mode but not within a critical region) all the functions that should have been executed previously are executed. After all functions that are to be executed are completed they are popped from the list and the list is compressed.

The clock handler plays an important role in identifying and penalized CPU bound processes. Since the consecutive execution time of a process is not (as yet) kept (only cumulative time "u_utime"), UNIX depends on an averaging type effect to identify CPU bound processes. This scheme examines the Processor Status once every second to determine whether the clock interrupt occurred while the processor was in User mode. If this Is the case, then the process that was interrupter is penalized by having it's priority ("p_pri" in the Process Table) lowered by 1 (the penalty scheme is only allowed to lower the priority as far as 105) and the processor is taken from the process. (Slp.c/swtch is called to select another process.) Since the floating point registers may not have been saved when the interrupt occurred, mch.s/savfp must be called before the process is preempted. Also, because signals are caught by a process only when the process calls on the system for service, a check must be made (by calling sig.c/issig) to see if there are any signals pending for the process. If this were not done, a CPU bound process could not be killed from its controlling teletype via the quit or interrupt keys. Since the operating system is not reentrant, processes cannot be preempted if the clock interrupt occurred in the midst of executing a critical region.

Note:
Critical regions of code are areas where it is necessary to raise the processor's priority to prevent interrupts from occurring or other processes from executing the same code.

timeout

CALL

timeout(function,argument,delay)
int (*function)();
int argument, delay;

RETURNS

No value is returned.

SYNOPSIS

Inserts an entry in the list of functions whose execution is to be delayed.

DESCRIPTION

The "callout[]" array consists of three word entries which specify a function ("c_func"), an argument to be passed to the function ("c_arg") and the amount of time a function is to be delayed ("c_time"). The time entry ("c_time") is the time relative to the previous entry that the function is to be executed, and is in sixtieths of a second. The first time entry is relative to the clock. Inserting an entry in the "callouta" array is done by taking the "delay" and finding the appropriate position in "callouta" to insert the entry. The "delay" is then translated into an appropriate relative time and inserted in "c_time". Since "callout[]" is an array all succeeding entries must be pushed down to make room for the new entry. Also, since the clock handler (clock.c/clock) updates the time in the "callout[]" array and arranges for the deletion of entries, clock interrupts must be locked out and traps prevented (by setting the priority to 7) while the new entry is inserted. The arguments "function" and "argument" are the address of the function to be executed and the value of an argument to be passed to that function.

CALL

none

RETURNS

none

SYNOPSIS

Character Device Switch Table and Block Device Switch Table. Also contains definition of the root device and the swap device.

DESCRIPTION

UNIX divides devices on the system into two classes; character devices and block devices. The distinction is basically between devices that are byte oriented and those that are oriented to transferring larger groups of data. As such, there are two different tables which are used to select the proper device. The Character Device Switch Table ("cdevsw") contains five entries for each device. These are the open ("d_open"), close ("d_close"), read ("d_read"), write ("d_write") and sgtty ("d_sgtty") entries. These entries are invoked when opening, closing, reading, writing or setting the modes for a character device. The Block Device Switch Table has a similar formal Here there is an open ("d_open"), close ("d_close"), strategy ("d_strategy") and device queue ("d_tab") entry. These are the open, and close routines for the device, the strategy routine for reading and writing the device and the device queue entry. Each table has a specific order in which devices may appear. The order is listed in the file and any device not listed may be added to the end of the table. A block device may have entries in both tables. In this case, the entries in the Character Device Table are used for physical I/O. Entries in the tables for which it is an error to reference the entry are filled in with the trap.c/nosys function, while entries which should produce no error are filled in by trap.c/nullsys.

There are four other variables in this file which allow the specification of the root filesystem and the swap device.

"rootdev" - This is the major and minor device number of the root filesystem. The high order byte contains the major device number and the low order byte contains the minor device number.

"swapdev" - This is the major and minor device number of the swap device.

"swplo" - This is the offset in 512 byte disk blocks into the swap device that the swap area begins. This entry is present so that the root filesystem and the swap device may be on the same minor device. This number may not ever be 0.

"nswap" - This is the number of 512 byte disk blocks allocated to the swap area.

access

CALL

access(aip, mode)
int *aip;
int mode;

RETURNS

On success return 0 otherwise return 1 and set appropriate error flag.

SYNOPSIS

Compare the "mode" argument with the mode permissions in the i-node pointed to by "aip". If permission is granted return zero, else one.

DESCRIPTION

The "mode" argument is READ, WRITE, or EXEC.

If the mode is WRITE the read-only status of the file system is checked and if it is read-only set the per user control block error indicator (u.u_error) to EROFS (read-only file system) and return 1. Similarly if the i-node pointi to a text image (read only code) than the file cannot be written upon as long as anyone is using the text image. If the i-node points to a text image set the error indicator to ETXTBSY (Text Busy).

If the user id is that of the super-user (userid 0) then all permissions are granted except for EXEC where at least one of the EXEC bits must be on.

The mode is shifted to match against the permissions for the owner, group, or foreigners depending on the match of the user id against the owner id, and group id in the i-node. If the mode does not match acceptable permissions, the error condition EACCES is set to indicate illegal access and return 1.

closef

CALL

closef (fp)
int *fp;

RETURNS

None.

STNOPSIS

Internal form of close. Decrements usage count (f count) of the System File Table (file.h) entry pointed to by "fp" and completes close when the count is zero.

DESCRIPTION

If the entry in the System File Table pointed to by "fp" is a pipe then it must be treated separately. Otherwise, the usage count is decremented and when it becomes zero, fio.c/closei is called to close the i-node.

closei

CALL

closei (ip, rw)
int *ip;
int rw

RETURNS

None

SYNOPSIS

Close the i-node pointed to by "ip". Closing an i-node implies decrernenting a reference count (done by fio.c/closef) and when it goes to zero rewrite the i-node to disk. Companion routine to openi.

DESCRIPTION

If the reference count is not zero just return. If the i-node pointed to by "ip" in the System INODE Table)inode.h) is a special file (device) then the appropriate device driver is called to close the device. This is done by switching on the major device number which is stored in addr[0] of the i-node.

Otherwise iget.c/iput is called to write the i-node to disk.

falloc

CALL

falloc()

RETURNS

A pointer to the first available System File Table entry or NULL if any errors.

SYNOPSIS

Allocate a user file descriptor from the per user control block (u.u_ofile), and build an entry in the

System File Table (file.h).

DESCRIPTION

Falloc calls fio.c/ufalloc to get the first available file descriptor in the per user control block (u.u_ofile). If no file descriptors are available the falloc returns a NULL as an error indicator.

A search is made for the first available System File Table (file.h) entry by looking for an entry where the count of connected processes (f count) is zero. The System File Table entry is initiated and the file descriptor is set to point to this entry.

If the System File Table is full print an error indication on system console and return NULL.

getf

CALL

getf (f)
int f;

RETURNS

On error set the per user control block error code (u.u_error) and return NULL. On success return a pointer to correct System File Table (file.h) entry.

SYNOPSIS

Checks for valid file descriptor values.

DESCRIPTION

Check to see that the file descriptor "f" is in the valid range (0-NOFILE); currently (0-15). Check that the file descriptor points to an open file and return pointer to the System File Table Structure (file.h). If not a valid file descriptor or the file is not open then set error code (EBADF) in the per user control block I return NULL.

openi

CALL

openi (ip, rw)
int *ip;
int rw;

RETURNS

On error, sets the per user control block error flag u.u_error to ENXIO, no such device or address.

SYNOPSIS

Openi is the routine to open special files (devices) whose associated i-node is pointed to by "ip" with read/write permissions "rw". Companion routine with closei.

DESCRIPTION

Openi is called every time a physical device must be opened. This occurs during the mounting of file systems as well as handling of devices.

Openi calls the open entry in the device driver for the character or block device identified in the i-node pointed to by "ip". Since this is the physical opening of a device, and each one acts differently the particular driver must be referenced for open processing. The only check made in openi is to make sure the major device number is within the range of the number of devices defined in a particular configuaration.

owner

CALL

owner()

RETURNS

On success return a pointer to an i-node. On failure return NULL.

SYNOPSIS

Check a pathname for ownership.

DESCRIPTION

Use nami.c/namei to lookup a pathname in user space and return a pointer to an i-node. If the aid of the i-node matches the uid in the per user control block (u.u_uid) then return the pointer to the i-node.

Call fio.c/suser to check if the effective userid of this user is super-user and if so return the pointer to the i-node. Otherwise, return NULL.

suser

CALL

suser()

RETURNS

Suser returns a flag if successful, otherwise, sets an error condition in the per user control block (u.u_error) indicating wrong owner (EPERM).

SYNOPSIS

Check to see if user is super-user.

DESCRIPTION

The superuser is known internally as userid 0. If the uid in the per user control block (u.u_uid) is zero then return the success flag. Otherwise, set an error indicator.

ufalloc

CALL

ufalloc()

RETURNS

A file descriptor if successful otherwise, return negative.

SYNOPSIS

Allocate a user file descriptor from the per user control block (u.u_ofile).

DESCRIPTION

File descriptors are numbers between 0 and 15 that represent a displacement into a list of pointers to System File Table entries.

Search the list of user file descriptors (current maximum 15) for the first empty slot and return the file descriptor. If all file descriptors have been used then return -1 as an error indicator.

iget

CALL

iget (dev, ino)
int dev;
int ino;

RETURNS

A pointer to a locked, incremented i-node if successful. Otherwise, a panic situation if the System INODE Table (inode.h) is full.

SYNOPSIS

Look up the i-node associated with i-number, "ino", on device, "dev", and return a pointer to the locked, incremented i-node.

DESCRIPTION

I-nodes are locked whenever they are created, updated, or written to the disk. The protocol if an i-node is locked is to set a bit requesting the i-node, then to sleep until it becomes unlocked.

Iget searches through the in-core System INODE Table (inode.h) for a matching i-number, "ino" and device, "dev". If it is present and is a mount point (i.e., some file system has been mounted on it) an indirection takes place to resume the search in the root directory of the mounted file system. The Mount Table contains the new device name for this mount point.

When the i-node is found, the usage count is incremented, it is locked, and a pointer to the inode is returned.

If the i-node is not present in the in-core System INODE Table then the i-node is fetched from the disk ilist of the appropriate device, "dev". The inode is read from the disk at block number B where B = (i-number + 31)/16
since there are 16 i-nodes per block and inumber 1 starts in block 2. The remainder from the above calculation, R, is used to locate the inode within the block. The i-node is moved into the System INODE Table, the count is incremented, and the i-node is locked. The pointer to this i-node is returned.

iput

CALL

iput (p)
struct i-node sp;

RETURNS

None

SYNOPSIS

Deallocate the i-node pointed to by "p".

DESCRIPTION

Iput decrements the usage count (i_count) of the i-node pointed to by "p". If the usage count indicates that there are other processes connnected to this i-node then return.

If this is the last reference (the usage count is zero), then the i-node is freed and may be written back to the disk ilist. This is done by locking the i-node to prevent other processes from connecting to it while it is de-allocated.

If the link count (i_nlink) is zero, the file has been removed by all users so that the i-node can be freed. Iget.c/itrunc is called to release all data blocks and indirect blocks used in this i-node. Alloc.c/ifree is called to put this i-number back on the freelist of i-numbers.

Iget.c/iupdat is called to update the modify date and time and write the i-node back to the ilist if necessary.

In all cases pipe.c/prele is called to unlock the inode.

itrunc

CALL

itrunc (ip)
int *ip;

RETURNS

None

SYNOPSIS

Free all the blocks associated with the i-node pointed to by "ip". (Truncate the file to zero length).

DESCRIPTION

Itrunc frees all the blocks associated with an inode. That means if the file is small it searches through the i-node and frees each block addressed

in the i-node. If the file is large, it reads each indirect block pointed to by the i-node into memory and frees each block in that indirect block. Itrunc continues until all blocks are freed, then the file size is set to zero, the file mode to small, and the update flag is set. If the mode is a special file (device), no blocks are associated so just return.

iupdat

CALL

iupdat (p, tm) int *p; int *tm;

RETURNS: None

SYNOPSIS

Check accessed and update flags on the i-node pointed to by "p" and if either flag is on, update the date and time with the values pointed to by "tm". Write the i-node back to the disk ilist.

DESCRIPTION

Iupdat calls fio.c/getfs to get the pointer to the superbiock of the appropriate file system. If this file system is mounted as read-only then the inode cannot be updated and the routine returns.

The block, B, within the ilist containing this inode is calculated as
B = (i-number + 31)/16
The remainder of the above calculation, R, is used to locate the i-node within the block.

If the access flag is on, the last access time is updated from the user time. If the update flag is on the last modified date and time is updated from the time pointed to by "tm". The i-node is then rewritten to the ilist.

maknode

CALL

maknode (mode)
int mode;

RETURNS

A pointer to the i-node of the file just created.

SYNOPSIS

Build an i-node for a new file with the read/write permissions specified in "mode".

DESCRIPTION

A new file is added to the directory found in the per user control block entry u.u_pdir. Maknode calls alloc.c/ialloc to allocate an i-node from the file system superbiock associated with the directory in the per user control block. It marks the i-node as allocated, set the link count to one, sets the uid (owner) and gid (associated project) from the per user control block (u.u_uid, u.u_gid). Iget.c/wdir is used to write the directory entry for this i-node. The directory entry is composed of an i-number and a 14 character file name.

wdir

CALL

wdir (ip)
int sip;

RETURNS

None

SYNOPSIS

Write a directory entry for the i-node pointed to by "ip".

DESCRIPTION

Directory entries are composed of a two byte inumber and a fourteen character file name. The name is taken from the per user control block where it was left on the previous call to nami.c/namei. The structure u_dent within the per user control block defines the format of directories. Wdir copies the filename from the per user control block temporary area (u_dbuf) to the directory entry area (u_dent). Wdir calls rdwri.c/writei to write the update directory back to the disk. It then calls iget.c/iput to deallocate the directory i-node.

CALL

none

RETURNS

none

SYNOPSIS

Low core vectors including jump table to involk interrupt handlers and the trap handler.

DESCRIPTION

DEC PDP-11 hardware controllers are wired to vector their interrupts to specific locations in low memory. The low.s file contains the new processor status word for each device. The processor status word consists of two words, the new program counter and the new processor status.

The new program counter is the address of a jump table entry (also in low.s) which calls a register save function (mch.s/call) before calling the interrupt handler. The new processor status word contains the priority at which the interrupt is handled and when there are more than one controller of the same type, it contains a number (the minor device number) in the low order 4 bits.

Since the operating system itself is not reentrant, the priority placed in the new processor status is the same as the priority at which the interrupt is handled.

estabur

CALL

estabur(text, data, bss)

RETURNS

No value is returned.

SYNOPSIS

Determines whether a process with a given text, data and stack size can fit within the limits of user virtual address space and loads prototype segmentation registers.

DESCRIPTION

With a 16 bit address, only 64K bytes of virtual address space is available to a user process. (When I and D space is implemented for user processes it will allow 64K bytes of text and 64K bytes of data, bss and stack.) The main.c/estabur function is called wheneyer the size of a process is to be changed (e.g. when an overlay is done, when a process needs more stack space , etc.) to make a test fit and insure that the process can take on tie new size. The first check that is made also insures that the text, data, stack and U block areas, which are loaded contiguously do not exceed the total amount of available user memory "maxmem".

Main.c/estabur insures that the text, data and stack are protected appropriately. The different areas are segregated into different groups of Memory Management registers.

the arguments "text", "data" and "stack" have a different meaning depending on whether the process being tested is reentrant or not. For reentrant processes the arguments represent the size of the text, data and stack areas respectively in memory blocks. For nonreentrant processes the "text" argument are zero and the the text and data is included in the "data" argument. The "data" argument in all cases includes the bss and data areas for a process. Reentrant processes must have their Memory Management Registers write protected and any portion of a register (less than 4K words) that is unused cannot be used to map data. Similarly, since the stack expands downward, it must be segregated from the data registers.

Processes are loaded into memory with the U block, text, data and stack area physically contiguous. The U block is not, however, included in the user’s virtual address space. For reentrant processes, the text may be loaded elsewhere. With this in mind, a check is made to see that the text, data and bss area will not exceed the number of memory management registers available (8).

If any of the checks fail, an error ENOMEN is posted (in "u_error") and a -1 is returned to the caller to indicate that the process should be aborted.

Processes that satisfy the above memory requirements can have their prototype segmentation registers loaded. "U_uisia[]" is the corresponding prototype User Instruction Address Register array (8 words) and "u_uisd[]" is the prototype User Instruction Descriptor Register array. Text registers are set up first (only for reentrant programs). Text registers have the Access Control Field in the prototype descriptor registerS set to read-only to preserve their reentrancy. If text only partially fills the virtual address space mapped by one of the registers, there will be a gap in the virtual address space of the process, as that register cannot be used to map the data portion. (The UNIX loader has forseen how the program will be loaded and relocated the object program appropriately.) The data portion is loaded into the succeeding registers and the access control is marked read/write. A virtual address gap will lie between the data and stack and any unused portion of a Memory Management register in the data area cannot be used to map stack space, because the expansion directions are different even though the access control permissions are the same. The prototype registers for the stack area are loaded starting at the high virtual address end. Each register is marked read/write, but the expansion direction is marked so that the stack grows downward in physical memory.

When the prototype registers have been set up, main.c/sureg is called to load the prototype registers into the User Memory Management registers.

A zero is returned to the caller of main.c/estabur to indicate that the process will fit in user virtual address space.

main

CALL

main()

RETURNS

No value is returned.

SYNOPSIS

Initializes the system.

DESCRIPTION

This function initializes all system buffers, mounts the root file system and creates the Scheduler and INIT processes. Since the Operating System is not loaded with either the C or assembly language library, there is no startoff function (crt0.s) to involk main.c/main. Instead, the mch.s/start function involks main.c/main. An arrangement also exists between main.c/main and mch.s/start for bringing up the INIT process. Mch.s/start sets up the virtual address space for the operating system and creates a stack (which will be owned by the Scheduler). The functions performed by main.c/main are:

1. The version number (release number) is printed on the system console.

2. The amount of memory available for user processes is determined and is cleared. This is done by setting up User Instruction Address Register 0 and User Instruction Descriptor register XXXX map successive memory blocks (64 bytes) XXXX and the address space of the operating system. The mch.s/fubyte function is used to fetch the first byte in a memory block. If a trap occurs as a result of this fetch, the clearing operation is terminated. Mch.s/clearseg is called to zero each memory block and the external variable "maxmem" is used to count the number of memory blocks. As each block is cleared, it is allocated (by calling alloc.c/mfree) to the free memory table ("coremap"). At the end of this operation, the amount of memory available for user processes will be in "maxmem" and the "coremap" array will be initialized to contain all of user memory as its first and only piece of available core.

3. The total amount of available memory may be limited by the system constant "MAXMEM" (see pararn.h). The external variable "maxmem" is set to the minimum value of "MAXMEM" and the amount of memory that was experimentally determined.

4. The amount of swap space ("nswap" in conf.c) and its location specified in "swplo" is entered in the "swapmap" array. The external variable "nswap" contains the number of blocks (512 bytes) available on the swap device. The external variable "swplo" is the offset (in blocks) on the device (specified in "swapdev") that this area begins. This offset cannot be zero as block zero has a special meaning to the system.

5. A determination is made as to which one of the two available clocks (KW11-L or KW11-P) is on the system. This is done in a manner similar to that by which the amount of available memory was determined. That is, a fetch is first attempted on the status register associated with the KW11-L (using mch.s/fuword). If this fails (i.e., a trap occurs) a fetch is attempted on the KW11-P status register. If neither fetch succeeds, then no clock is present and the system panics ("PANIC NO CLOCK"). For the KW11- L clock, it is only necessary to set the interrupt enable bit to start the clock counting at line frequency. For the KW11-P clock, the line frequency rate must be selected and repeat interrupt mode set (see DEC Peripherals Handbook).

6. The Process Table entry for the Scheduler is then set up. The Scheduler is a process which runs entirely in Kernel Address space. It is always process zero in the Process Table and is always locked in memory. (The SLOAD flag in "p_flag" is always set and a special indicator SSYS is also set to mark it as the Scheduler.) The size and location of:the Scheduler do not correspond to the location and size of the function slp.c/sched. The location of the Scheduler is taken to be the start of the U block and its size is taken to be that of the U block. This is done because the INIT process is created by the Operating System forking and as small a core image as possible should be used. The U block created by mch.s/start is allocated to the Scheduler.

7. The block device buffers are initialized by bio.c/binit and the character device buffers are initialized by tty.c/cinit. These routines also determine the number of block and character devices on the system. The root file system is mounted (by calling alloc.c/iinit). The root inode is retrieved from the root file system (via iget.c/iget), and an external variable "rootdev" is loaded with the address in the Inode Table entry of the root inode. The working directory entry "u_cdir" is also set up so that it indicates the root directory. (This is done so that when the INIT process is spawned, it will have the root inode as it’s

working directory.)

8. The INIT process is spawned. This is done by a trick in which a tiny program is hand crafted in memory and executed. This program (a copy of which is in the "icode[]" array") simply requests an overlay of the INIT process. The actual procedure is as follows:

a. The slp.c/newproc function is called to do a fork of the Scheduling process. From 6 above, the size of the Scheduler was set to the size of it's U block, so a fork replicates the U block. The slp.c/newproc function creates a new process within the system which is an identical copy of the original process. Both processes begin executing at the return from slp.c/newproc. The only difference is that since only one process can be executing at a time, one process (the child) will actually return from the slp.c/swtch function, and not slp.c/newproc. The child process (forerunner of INIT) will call for the creation of a one memory block (64 bytes) area for the program (by calling slp.c/expand) and will set up the prototype segmentation registers "u_uisa[]" and "u_uisd[]", so that main.c/sureg can be called to actually load them. Main.c/sureg sets up as a default a 32 word memory block An offset (USIZE) is setup in the prototype segmentation address register "u_uisa[0]", so that the physical area of memory where the program is loaded is directly behind the Scheduler's U block. Once this has been done, the "icode[]" program is copied into the user's address space (by mch.s/copyout). (It should be noted that the child process is essentially creating itself.) The mch.s/start function is set up so that upon a return to it from main.c/main a system call is simulated. This is done by setting up the system stack so that a return from trap (RTT) instruction is executed, which will take the execution into User address space at virtual address 0. The "icode[]" program executes and makes an exec system call to overlay itself with the /etc/init process.

The parent process (in this case the system) only creates the U block for the new process. The child actually loads the "icode[]" program. This is essentially a combination of a fork and exec system call. The parent calls the function slp.c/sched which is the endlessly looping Scheduling process. The Scheduler receives no signals, so that it cannot be killed and thus need never be respawned.

sureg

CALL

sureg()

RETURNS

No value is returned.

SYNOPSIS

Loads User Memory Management registers from their software prototypes.

DESCRIPTION

In order to dispatch any process, that process' virtual address space must be setup. Also, when any growth in the size of a process (stack growth or memory allocation) occurs, the virtual address map in the Memory Mansgement Unit must be changed and reloaded. The main.c/sureg function sets up the user virtual address space by loading the Memory Management registers from the prototype address and descriptor registers ("u_uisa[]" and "u_uisd[]" which contain the virtual address map for the process relative to absolute location 0. In order to load the Memory Management Unit, the address registers ("u_uisa[]") must be relocated to the proper physical address. In particular, the "p_addr" entry in the Process Tabli contains the physical address (in memory blocks) of the process. This value is added to each of the instruction address prototype ("u_uisa[]") registers when they are loaded into the User Memory Management Address Registers. (When the prototype registers were set up by main.c/estabur, allowance was made for the position and size of the U block.) Since reentrant processes may have the text segment loaded elsewhere, a further adjustment of the registers mapping the text area may be necessary. Reentrancy can be checked for by examining the "p_textp" entry in the Process Table. If this is zero, the process is not reentrant. If nonzero, it contains a pointer to a Text Table entry which contains the address of the text ("u_caddr"). A relative correction ("p_addr" - "x_addr") can then be applied to the reentrant text address registers when the prototype descriptor register ("u_uisd[]") are loaded. Registers mapping reentrant text must also have the write only bit set in the Access Control Field of the descriptor register while those for stack and data (including nonreentrant text) are read-write.

aretu

CALL

aretu (save)
int *save;

RETURN

No value is returned.

SYNOPSIS

Restores the stack position or a process. Used as part of a nonlocal goto within the operating system.

DESCRIPTION

This function is similar to mch.s/retu. The only difference is that it does not alter Kernel Instruction Address Register 6 (KISA6) on 11/40's or Kernel Data Address Register 6 (KDSA6) on 11/45's and 11/70's which perform the virtual address mapping for the U block. It is used to transfer execution back several levels of subroutines without returning via the intervening functions. In the case of processes .which are catching their own signals it is used to transfer control to the user process as soon as the presence of a signal is detected even at the expense of aborting a system call. For processes that have performed their own swapping, without interacting with the Scheduler, it is used to transfer control to the area of code within the system that did the swap rather than simply returning from a roadblock (slp.c/sleep).

The nonlocal goto is performed by using mch.s/aretu to restore the system's R5 and SP from values previously saved(in "u_rsav", u_usav" or "u_gsav"). Since the PC for a return from a C function is also on the stack(just above where the stack frame pointer RS is pointing), when a return by a C function(jump cret) is executed. A return is made to the function specified on the stack, instead of returning to the actual caller. This PC is the return address of some higher level function mch.s/savu. The operation appears as if the function which last called mch.s/savu returned to it's caller.

Since changing stack positions is a delicate operation, the processor's priority is raised to 7 to lock out all interrupts while the stack position is changed. The processor's priority is then lowered to zero so that all pending interrupts may be processed.

backup

CALL

backup(framep)
int *framep;

RETURN

A zero is returned to indicate success.

SYNOPSIS

Attempts to back up an instruction. This is always possible on 11/45 and 11/70 processors but not always on 11/40's due to the makeup of the 11/40 hardware.

DESCRIPTION

If a Stack Violation(Segmentation Violation) occurs while executing a user process, it is necessary to undo the portion of the instruction that had been executed before increasing the process's stack size. The Memory Management Unit generates a trap when an illegal virtual memory address occurs and on 11145 and 11/70 processors a register (Memory Management Status Register 1) is loaded as each instruction is executed with information about what register has been modified and by how much. This register is unavailable on the 11/40 processor and since there are autoincrement and autodecrement instructions on PDP-11's not every instruction may be restarted. (The restrictions governing 11/40's will be listed below.) A Stack Violation is distinguished from other possible Segmentation Violations by the trap handler which checks the position of the SP relative to the size of the stack. The trap handler calls mch.s/backup to adjust the stack frame before restarting a process, and if this cannot be done(on 11/40's this is a possibility) an indication of failure is returned so that the process may be aborted. The trap resulting from the Segmentation Violation may have been the result of an illegal memory(stack) reference by either the source or destination fields of an instruction, so that only part of the instruction may have been executed. The ability to determine what part of an instruction aborted is a crucial factor in the determination of whether an instruction can be backed up or not. Another important determination is that of how much and which source and/or destination registers have been incremented or decremented by the use of the autoincrement, deferred autoincrement, autoincrement and autodecrement deferred addressing modes. For

11/45 and 11/70 processors, Memory Management Unit Status Register 1 contains this information. (11/40 processors do not have this register so that the software must make this determination.) Restarting an instruction on 11/45 and 11/70 processors is simply done by using the information in Status Register 1 about how much a source and/or destination register has been changed to adjust the registers to their values before the instruction was executed. Status Register 1 contains two bytes. The high order 5 bits of the first byte contain the amount (positive or negative) that the source register has been changed by incrementing or decrementing. The lower three bits contain which source register was modified. The second byte contains similar informamation about the destination register. When a trap occurs, the user process' registers (context) are saved on the system stack so that to restart an instruction, the registers modified by autoincrement or autodecrement need only be adjusted and the PC moved back to restart the instruction. The argument "stackp" _passed to mch.s/backup is an address on the stack frame where the registers may be found and the array "reloc"(see trap.c/trap) is used as an offset map to find the proper register (see mch.s/call description) on this stack frame.

11/40, 11/45 and 11/70 processors use common features of the mch.s/backup routine to adjust an instruction, however, on 11/40's there is additional software which attempts to simulate the operation of Status Register 1. In most cases the simulation can back up the instruction, however, there are some classes of instructions which cannot be backed up by software.

Rather than give a detailed description of the algorithms used in the 11/40 software to determining which class of instructions the faulting instruction belonged to and how much incrementing or decrementing of registers occurred, a discussion of which instructions can and cannot be backed up will be given. The DEC Processor Handbook may be consulted for instruction formats and the KB11-A Central Processor Maintenance Manual Fig. 6-10 may be consulted for the microprogrammed decoding of the opcode field for those interested in how opcodes are mapped into instruction types. Some of the simplifying assumptions made in the simulation of an instruction will be pointed out.

The key factors in the instruction fetch/execute cycle that must be remembered are,

1. Instructions consist of at most two operands; source and destination. The Stack Violation could only be a result of a bad address constructed for the destination in the case of single operand instructions but may be the result of either the source or destination for double operand instructions.
2. If the addressing modes consist of any incrementing (autoincrement, autoincrement deferred) or decrementing (autodecrement, autodecrement deferred), these are performed up to the point that the instruction aborted. For example, the instruction

   mov (sp)+,(r5)+

   would have the first incrementation (only) performed if the reference using the SP caused the abort. If the reference using R5 caused the abort both incrementations would occur. A similar case occurs if the registers are decremented.

With these two factors in mind , the following conclusions may be drawn.

1. All single operand instructions can be backed up as their only memory reference must have caused the abort.
2. For double operand instructions, a determination of which operand caused the abort must be made. To do this, the mch.s/backup routine must simulate the fetch of both the source and destination. If an error occurs while doing the simulated fetch of the source then the instruction must have aborted at this point and the destination could not have been incremented or decremented. Unfortunately, to do this simulation on instructions which increment or decrement both source and destination, it must be assumed that the registers involved can be unincremented (or undecremented) before simulating the instruction fetch of source or destination. Instructions of the following form

   operator (rA)+,(rA)+

   (any combination of autoincrement and autodecrement is included and rA is an abbreviation for any register) present a problem since it cannot be determined how far the register should be backed up. (That is, one cannot tell whether both incrementations occurred since the same register rA is specified in source and destination fields.)

The characteristics of instructions on the 11/40 that cannot be backed up are as follows. (The

instruction must satisfy *all* of the following conditions.)

a. Double operand.

b. Uses same register for source and destination.

c. Destination involves incrementing or decrementing.

d. Source involves an addressing mode other than mode 0 (register mode addressing). Only a few instructions fall into this category and they are *not* generated by the C compiler. The user must avoid using them in coding assembly language programs that will run on 11/40 processors.

In simulating the fetch cycle for the source field on double operand instructions, an assumption about addressing is made for simplicity. When simulating an autoincrement source fetch, the register can be adjusted to its original value before the simulation, Thus, in the following instruction

cmp (sp)+,(r5)+

the SP is moved back before simulating the source fetch. For the decrementation modes, it is assumed that if the adjustment is not made the same error will occur. Thus, for example, it is assumed that the instruction

cmp -(sp),(r5)+

will give the same violation with or without adjusting the Stack Pointer(SP).

Floating point instructions cannot be executed on the 11/40 (no floating point processor option is available) so they are not backed up. Branches (br, jsr, jmp) are not backed up either as it is assumed that it is an error for the user process to attempt to jump to an address that is not already within his virtual address space.

call

CALL

jsr R0, call; _function

RETURNS

No value is returned.

SYNOPSIS

Builds a stack frame so that a smooth interface to a C language interrupt handler or the trap handler may occur. Also does any context saving including floating point registers.

DESCRIPTION

The assembly language function mch.s/call is complicated as it serves as a common interface to the trap and C language interrupt handlers. There are also differences in its operation based on whether the processor has a floating point unit or not. The basic operation of the routine that is common to 11/40, 11/45, and 11/70 processors will be explained first and differences discussed. afterwards. The common features of the call function are:

1. It builds a stack frame for an interface to interrupt handlers and the trap handler. When an interrupt or trap occurs the PDP-11 hardware causes a new Program Counter (PC) and Processor Status (PS1 to be loaded from the low core vector area (see low.s discussion) and the old PC and old PS to be pushed onto the stack specified by the Current Mode field of the low core PS. (The Kernel mode stack is used for all cases under UNIX.) As part of the operation of loading the new PS from low core, the PDP-11 hardware sets the Previous Mode field of the Processor Status so that it indicates the mode that the processor was in when the interrupt occurred. Upon entering an C language function, the save restore sequence (mch.s/csv, mch.s/cret) saves registers R2-R7 so that only registers R0 and R1 need to be saved by mch.s/call. It should also be mentioned that the stack that is used by the operating system resides in the bottom portion of the U block of the currently executing process. The stack frame that is built in the U block stack area every time an interrupt or trap occurs consists of the following entries in the following order:

   a. old Ps - Processor Status when the interrupt or trap occurred(saved by hardware interrupt mechanism).

   b. old PC - Program Counter when the interrupt or trap occurred(saved by hardware interrupt mechanism).

   c. Register R0 of General Register set 0(UNIX only uses General Register set 0).

   d. new PS - Processor Status that was loaded from low core when the interrupt occurred. The Previous Mode field has, however, been set to the appropriate value.

   e. Register R1 of General Register set 0.

f. SP from Previous Mode - There is one stack pointer for each of the three processor modes; User, Kernel, Supervisor. This stack pointer is saved as a convenience for the trap handler so that Stack Violations may be easily fixed and argument's for system calls may be readily found.

g. The remainder of General Register Set 0 is saved when the mch.s/call function calls the interrupt or trap handler(all C language). The register save sequence is described under mch.s/csv. The order is basically; the PC containing the return address, R5, R4, R3, R2. Register R6 (the Stack Pointer - SP) does not need to be saved as it can be located relative to the contents of R5.

h. Minor device number - The lower 5 bits of the low core PS contains the minor device number of the device initiating the interrupt or the type of trap that occurred. This is placed on the stack as a convenience to both the trap and interrupt handler. Doing this saves the interrupt handler from having to check the attention flags on each device.

Besides building the stack frame, the mch.s/call function is responsible for clearing off the stack frame and executing a return from the trap or interrupt(RTT instruction) when the trap or interrupt handler is finished.

2. The mch.s/call function calls the appropriate trap or interrupt handler. The low core vectors reference what are essentially jump table entries just below the floating vector section of low core. The jump table entries are of the form

jsr R0, call; _handler

The entry _handler is the address of the appropriate interrupt handler. The mch.s/call function calls the appropriate interrupt handler by retrieving this entry.

3. The mch.s/call function determines whether the process that was running when the interrupt or trap occurred may be preempted. Any interrupt that results in a process being awakened indicates that the currently running process should be preempted. Because of reentrancy requirements in the system, preemption can only occur if the interrupt occurred while the processor was in User mode or when a system call completes. Since system calls are made by trapping and the trap interface uses the interface provided by mch.s/call, the logic for making this determination is built into the mch.s/call function. The Previous Mode field of the PS will indicate whether the interrupt occurred out of User or Kernel mode and the external variable "runrun" is used by the slp.c/walreup function to indicate whether the interrupt resulted in another process being awakened. As mentioned previously, interrupts occurring while the processor is handling a system call(or a trap) cannot result in a preemption until the system call is completed. The mch.s/call function calls the process Switcher (slpc/switch) directly to preempt a process when an interrupt is finished or when the system call completes.

4. For processors that have floating point registers (11/45 and 11/70 only), the mch.s/call function calls the mch.s/savfp function to save the floating point registers(in the U block) and restores the registers when the process resumes. The floating point registers only need to be saved if the process is being preempted as the operating system does not use floating point. When restoring the stack frame the processor's priority must temporarily be raised to 7 to prevent another trap or interrupt from destroying the values on the stack frame.

call1

CALL

jsr r0,call1; trap

RETURN

No value is returned.

SYNOPSIS

Saves register R0 for the trap handler.

DESCRIPTION

This is a special entry point to the mch.s/call function used for the trap handler(see fnch.s/trap).

clearseg

CALL

clearseg (addr)
int *addr;

RETURNS

No value is returned.

SYNOPSIS

Zeros a 32 word memory block.

DESCRIPTION

Mch.s/clearseg is very similar in operation to mch.s/copyseg. The argument "add" contains the high order address of a 32 word memory block to be cleared. The argument "addr" can be loaded directly into the address portion of a Memory Management Address Register.

For 11/45 and 11/70 processor's the Supervisor Memory Management registers are available to perform address mapping. Data Address Register 0 of the Supervisor Memory Management registers is used to perform relocation and the Previous Mode of the Processor Status is adjusted so that Supervisory mapping will be used when fetching and storing across address spaces (MFPI,MTPI,etc.).

There is no Supervisory state on 11/40 processors so User Instruction Address Register 0 is saved and restored so that it can be used for mapping. In this case, the processor's priority must be raised to 7 to prevent any interrruption while the User Memory Management Register is modified.

copsu

CALL

jsr pc, copsu

RETURN

No value is returned.

SYNOPSIS

Common subroutine used by mch.s/copyout and mch.s/copyin.

DESCRIPTION

In moving data between the user's address space and the system's buffers, both the mch.s/copyin and mch.s/copyout routine execute a tight loop which move data between address spaces. The mch.s/copsu function sets up registers for both of these functions and has access to the three arguments passed to them. The calling sequence of these two functions are

copyin(vuser, vsys, count)
copyout(vsys, vuser, count)

The "vsys" and "vuser" arguments are the virtual addresses within the system and user address spaces respectively. The "count" is the byte count for the transfer.

Mch.s/copsu sets up registers R0, R1, and R2 to contain these parameters. Registers R0 and R1 are scratch registers when copyin or copyout are called since they are only called from parts of the system that are written in C. Register R2 must, however, be saved on the stack. The registers are setup as follows:

R0 - Argument 1 (either "vuser" or "vsys")

R1 - Argument 2 (either "vsys" or "vuser")

R2 - Argunent 3 byte "count" multiplied by 2 to give the word count, since both mch.s/copyin and mch.s/copyout transfer words across address spaces.

Since both the mch.s/copyin and mch.s/copyout functions move data across address spaces, the possibility of aborting due to bad virtual addresses specified in the call exist. In order to allow both mch.s/copyin and mch.s/copyout to detect these errors, mch.s/copysu sets up the external variable "nofault" (see mch.s/trap discussion) so that if any trap occurs mch.s/copyin or mch.s/copyout will be notified by the trap handler and they can return an error indication (-1) to their caller(rdwri.c/iomove).

copyin

CALL

copyin(vuser, vsys, count)
int *vuser, *vsys;
int count;

RETURNS

Zero on success, -1 on failure.

SYNOPSIS

Copys data form user virtual address spaces to system address space(word by word).

DESCRIPTION

In order to buffer I/O between a user's process and the devices on a system there are a number of system buffers. Moving data from the user's virtual address space to the buffers which are in the

system's virtual address space is done by mch.s/copyin (a higher level routine rdwri.c/iomove does address calculation, buffer allocation, etc.).

The number of bytes specified by "count" are moved, word at a time, starting at the user's virtual address "vuser" to the system's virtual address "vsys". The move is done by using the Move From Previous Instruction Space(MFPI) processor instruction in a tight loop. The routine mch.s/copysu initializes this loop and provides error recovery in case a trap should occur in moving data across address spaces.

copyout

CALL

```
struct block{
   int b[32];
};
copyout(vsys, vusr virtual, count)
struct block *vuser, *vsys;
int     count;
```

RETURNS

Zero on success, -1 on failure.

SYNOPSIS

Copies data from system(Kernel) virtual address space to user address space (word by word).

DESCRIPTION

Comparable function to mch.s/copyin, performing the complimentary operation of transferring data from the system to a user process.

copyseg

CALL

```
struct block(
   int b[32];
);
copyseg(source, destination)
struct block *source, *destination;
```

RETURNS

No value is returned.

SYNOPSIS

Copies a memory block(32 words) from one area of physical memory to another.

DESCRIPTION

Creating a new process, expanding the size of an existing process or correcting a stack violation require copying a process from one area of core to another. Since the transfer of data may be to any place in core (128K word maximum on 11/40, 11/45, and 2M word on 11/70's) the Memory Management Unit must be setup to transfer the data.

The Supervisor Memory Management registers are available for this operation on 11/45 and 11/70's processors so that the transfer may be effected by simply loading two of the Supervisor Memory Management registers so that they address the 32 word "Source" and the 32 word "destination". By setting the Previous Mode field in the Processor Staffs and using a tight loop which first fetches (MFPD - Move From Previous Instruction Space) from the source area, then stores(MTP1 - Move To Previous Instruction Space) into the destination area, the data may be transferred. Two of the Data Space Address Registers of the Supervisor Memory Management Unit are used for mapping the source and destination on 11/45 and 11/70's.

On 11/40's the Supervisory state does not exist so that two of the User Instruction Space Address Registers are used for mapping the transfer. (Their original contents must be saved and restored on the stack.)

cret

CALL

imp cret

RETURNS

No value is returned.

SYNOPSIS

Restores general registers for C functions.

DESCRIPTION

At the end of every compiled C function, mch.s/cret is called to restore the registers that were saved by mch.s/csv and thereby to reset the stack frame to that of the calling function. Register R6(SP) is not saved as part of the register save sequence because it is used in conjunction with register R5 (the stack frame pointer). The registers are restored in the same sequence- that they are saved (see imch.s/csv). Any local storage

allocated on the stack frame (below the register save area) is cleared of by in line instructions generated by the C compiler. Arguments to functions are also passed on the stack frame and in line instructions are generated following the subroutine call to clear them from the stack.

csv

CALL

jsr r5, csv

RETURNS

No value is returned.

SYNOPSIS

Performs register save at the beginning of every C function.

DESCRIPTION

Every function(i.e., subroutine) that is compiled for the C language calls this function to save the general registers before any computation is done. Registers R0 and R1 are regarded as scratch by the C language and therefore need not be saved when a function is called. When any subroutine is called in C, register R5 is setup to point to the current stack location. This is the beginning of a stark frame for the called subrputine. At the beginning of this stack frame the general registers are saved in the order RS(saved by the call to mch.s/csv), R4, R3, R2. The stack pointer(SP) is set up to point to the first free location on the stack below this Register save area. The SP is used as a local stack pointer within the stack frame. Since it can be located in relation to R5, there is no need to save it. Only General Register Set 0 is used by the processor so the mch.s/csv function is the same in the operating system as for compiled user programs. Once a C function has saved the general registers and created a stack frame by calling mch.s/csv, storage for local variables in the stack frame is allocated. (The C compiler generates in line instruction within each C functions to do this.)

display

CALL

display()

RETURNS

No value returned.

SYNOPSIS

Displays the contents of a memory location on the console lights.

DESCRIPTION

Display reads the contents of the console switches and displays the contents of that virtual memory address on the console’s data display register lights. The virtual address is taken from kernel address space if bit zero of the console switches is zero,,,otherwise the address is taken from user addreis space. This function does an immediate rettitn without displaying the contents of a memory rocation on an 11/40 system. On an 11/45, the data display select knob must be set to DISPLAY REGISTER in order to have the memory location’s contents displayed.

dpadd

CALL

dpadd(dbl, sing)
int dbl[2], sing;

RETURNS

No value returned.

SYNOPSIS

Adds a single word integer to a double word integer.

DESCRIPTION

Dpadd adds the single word "sing" to the double word that is pointed to by "dbl".

dpcmp

CALL

dpcmp(f[0], f[1], s[0], s[1])
int f[2], s[2];

RETURNS

The difference between the two double word values if the difference is between -512 and 512. Otherwise, a -512 or 512, depending on whether

the difference is negative or positive.

SYNOPSIS

Subtracts two double word values and guarantees the returned result to be in the range -512 to 512.

DESCRIPTION

Dpcmp is useful for comparing two double word values "f" and "s", which the system does frequently when doing file I/O. The difference between these double words (i.e., f - s) is returned if it is between -512 and 512. If the difference is less than -512 or greater than 512, then -512 or 512, respectively, is returned.

dump

CALL

jmp dump

RETURNS

No value is returned.

SYNOPSIS

A postmortem memory dump to magnetic tape.

DESCRIPTION

This is a utility program for dumping the contents of memory on magnetic tape if the system crashes. Registers R0-R6 and Kernel Instruction Address Register 6 are saved (in that order) in low core starting at absolute location 4. The mch.s/dump function is started after crashing by loading absolute address 044 into the program counter and depressing the start switch. All of core, or all of core until the first magnetic tape error occurs is dumped onto tape unit zero in 512 byte blocks. An end of file is also written on the tape. The mch.s/dump function will not operate unless relocation has been turned off(bit 0 of Memory Management Status Register 0 is 0). A mch.s/dump routine exists for both the TM11 and TU16 magnetic tape units. Before assembling mch.s the proper dump routine is selected by setting one of the conditional assembly flags (.tm). For 11/45 and 11/70 systems, the mch.s/dump routine is loaded in low core and is mapped by D Space Address Registers since it is only used post mortem when relocation is of This is done in order to allow the operating system to have as large a virtual address space(Instruction Space) as possible.

fetch

CALL

mov addr,R0
jsr pc, fetch

RETURNS

The instruction at User virtual address "addr" is returned in register R0. A -1 is returned if the virtual address does not exist.

SYNOPSIS

Simulates the instruction fetch or operand fetch cycle of an instruction. Used in backing up an instruction.

DESCRIPTION

This function is used by mch.s/backup in simulating the instruction fetch and operand fetch cycle of a PDP-11 instruction when backing up an instruction on 11/40 processors. Because of the possibility of improperly accessing across virtual address spaces, mch.s/fetch must be ready to catch a trap that might be generated. It does this by setting the external variable "nofault" to the address of its own internal error handling routine. The trap handler checks "nofault" to see if it is set and transfers to this error routine instead of the trap handler(trap.c/trap) if the fetch aborts.

fubyte

CALL

fubyte (virtual)
char *virtual;

RETURNS

The byte specified by the argument "virtual" isreturned. A -1 is returned if the byte cannot be accessed.

SYNOPSIS

Fetches a byte from a users virtual address space.

DESCRIPTION

This function is used to determine the amount of memory when the system comes up and for moving characters between the I/O subsystem and the user’s virtual address space(subr.c/cpass).

Most of the work in fetching a byte from the user’s virtual address space is performed by the subroutine mch.s/gword. Mch.s/fubyte merely determines which byte is to be returned to the caller.

fuword

CALL

fuword(virtual)
int *virtual;

RETURNS

A -1 on failure.

SYNOPSIS

Fetches a word from user's virtual address space,

DESCRIPTION

Used to fetch arguments to system calls by trap handler and arguments for the exec system call.

Mch.s/fuword uses the common subroutine mch.s/gword for accessing the user's virtual address space and catching any errors in fetching across virtual address spaces.

getc

CALL

getc(cp)
struct dist *cp;

RETURNS

A -1 is returned if there are no characters on the teletype queue "cp".

SYNOPSIS

Used to get a single character off a teletype queue returning character buffer storage area to the free list ("cfreelist") as needed.

DESCRIPTION

Several queues are associated with each character device, an input queue("t_rawq"), a canonical queue("t_canq") and an output queue("t_outq"). The mch.s/getaunction will remove one character from the queue specified by "cp".

Character buffer storage is allocated to a queue as a linked chain of storage. Each increment of buffer ("cblock") contains one pointer word and six characters of buffer space. Each queue header("clist") contains a count of the characters that are allocated to the queue("c_cc"), a pointer to the first character in the first buffer("c_cf") and the next available character posit("c_cl") in the last buffer on the queue.

It should be emphasized that queues maintains pointers to the first and last characters respectively and not the first and last buffer on the queue. This can be done because in allocating and initializing space for the character queues, care was taken to insure that each character buffer began on an eight byte memory address boundary. With this arrangement, the queue pointers reveal not only the (first and last) character but the buffer which they are in. The first character pointer, last character pointer, or the byte count are zero if there are no characters on a queue.

Mch.s/getc removes the first character on the queue (pointed to by sc_cr), and returns it to the caller. If the character removed is the last one in a character buffer, the character buffer is unlinked from the queue and returned to the free list of character queue ("cfreelist"). The free list of buffers is organized as follows:

1. "Cfreelist" is a one word pointer to the beginning of a queue of free buffers. If there is no free storage then "cfreelist" is zero.
2. Each free buffer on the queue uses its pointer entry as a link to the next free buffer. The last buffer on the queue contains a zero in its pointer entry.
3. When a character buffer is returned to the free list, it is placed at the head of the queue.

Since mch.s/getc is called by many character driver interrupt handlers it is necessary to raise the processor's priority to 5 to lock out interrupts from character devices while mch.s/getc is unlinking pointers to a queue.

The character removed from the queue is returned(in R0) or a -1 is returned to indicate that the queue was empty.

gword

CALL

mov addr, rl
jsr pc, gword

RETURNS

A -1 is returned on failure.

SYNOPSIS

Common subroutine used by mch.s/fubyte and mch.s/fuword to fetch data from the virtual address space of a user process.

DESCRIPTION

The Move From Previous Instruction Space (MFPI) instruction on the PDP-11 is for fetching data from another address space. The Previous

Mode field of the Processor Status (PS) indicates the address space from which the data is to be fetched. The operand field of the MFPI instruction specifies the virtual address that is to be used for retrieving the data. Execution of the MFPI instruction results in the desired word being placed on the Current Mode stack(Kernel in this case). These instructions operate only on even addresses and there is no analogous byte oriented instruction.

Whenever data is moved across address spaces, there is a possibility that the desired address does not exist in the target address space. A Segmentation Violation trap will occur if an illegal virtual address is generated so that mch.s/gword must be ready to catch these faults. It sets up the address of it's own error routine in "nofault" so that the trap handler will transfer control to this error routine instead of calling the system trap handler(trap.c/trap).

idle

CALL

idle()

RETURNS

No value is returned.

SYNOPSIS

The processor goes into the WAIT state until the next interrupt occurs.

DESCRIPTION

The process Switcher(slp.c/switch) selects processes to use the CPU. If there are no processes in the system or no processes that are ready to run, the mch.s/idle routine is called to place the processor in the WAIT state. The mch.s/idle routine is called only by the process Switcher and executes the WAIT instruction after it has saved the Processor's Status and lowered the processor's priority to zero(to allow any interrupt to occur). When an interrupt does occur, it is processed and then control is returned to the mch.s/idle function. The Processor Status is returned to its original value and control is returned to the process Switcher. When the processor has no work to do (e.g., overnight), the clock will continue to interrupt once per sixtieth of a second so that the process Switcher will run at this frequency and will idle the processor in between.

incupc

CALL

incupc(pc, prof)
int pc, *prof;

RETURNS

No value returned.

SYNOPSIS

Does the actual profiling of a user process.

DESCRIPTION

Incupc is the function that actually performs profiling, whereas sys4.c/profil only enables and disables the profiling process. When called with a value for the user's program counter ("pc") and the address of the profiling arguments ("prof") found in the. U block (u_prof[]), this function evaluates the expression:

(pc - profile offset) * profile scale

If the result is less than the profiling buffer size, then that word within the profile buffer is incremented by one.

ldiv

CALL

ldiv(divid, divisor)
int divid, divisor;

RETURNS

The quotient of two integers.

SYNOPSIS

Divides two integers.

DESCRIPTION

Ldiv returns the quotient from the division of "divid" by "divisor". The dividend is always regarded as an unsigned number. This permits division of sixteen bit values and assures that the quotient will always be positive. There is no check to ensure that the divisor is not zero. See also mch.s/lrem.

lrem

CALL

lrem(divid, divisor)
int divid, divisor;

RETURNS

The remainder from division of two integers.

SYNOPSIS

Finds the remainder froredivision of two integers.

DESCRIPTION

Lrem returns the remainder from the division of "divid" by "divisor". The dividend is regarded as an unsigned number. This permits division of sixteen bit values and assures that the remainder is always positive. There is no check to ensure that the divisor is not zero. See also mch.s/ldiv.

lshift

CALL

lshift(dbl, bits)
int dbl(2], bits;

RETURNS

The low 16 bits from the shift of a double word.

SYNOPSIS

Shifts a double word to the right or left a designated number of bits.

DESCRIPTION

Lshift shifts the double word "dbl" to the right (i.e., division) or the left (i.e., multiplication) by "bits" number of bits. If "bits" is negative, the shift is to the right, otherwise it is to the left. The low order 16 bits of the result are returned.

putc

CALL

putc(c, cp)
char c;
struct clist *cp;

RETURNS

A zero is returned if a character is successfully placed on a teletype queue; a nonzero value if not.

SYNOPSIS

Places a character on a teletype que,ue allocating character buffer storage as needed.

DESCRIPTION

Mch.s/putc performs the complimentary operation to mch.s/getc. It places the character "c" at the end of the teletpe queue specified by "cp". The queue is organized as described under mch.s/getc with a character count("c_cc") and first and last character pointer("c_cf", "c_cl", respectively). The first character pointer gives the address of the character at the head of the queue, while the last character pointer("c_cl") gives the address of the next available position at the end of the queue. When placing a character on a character queue, the last character pointer is changed to reflect the addition of a new character The character count is also incremented by one. If, however, there is no space available on the queue a new character buffer is allocated from the free list("cfreelist"). Allocating a new buffer to the queue and inserting the character is done as follows:

1. A character buffer is deallocated from the queue of free buffers("cfreelist"). The first buffer on the queue is chosen. The organization of this queue is described under mch.s/getc.
2. If the queue onto which the character is to be placed is empty, the first character pointer("c_cf") must be set in addition to the normal setting of the last character pointer("c_cl"). If there are character buffers already allocated to the teletype queue, the new storage is added to the end of the linked queue. The last buffer on the teletype queue has its pointer field set to zero so that addition is accomplished by merely changing this field to point to the new buffer and zeroing the pointer entry in the new buffer.
3. The last character("c_cl") pointer is adjusted to point to the address of the next free character position in the buffer.
4. The character count("c_cc") is incremented.

If there are no buffers left("cfreelist" empty) in the buffer pool, a nonzero value is returned to the caller of mch.s/putc, a zero is returned for success.

pword

CALL

mov word, r0
mov addr, rl
jsr pc, pword

RETURNS

-1 on failure.

SYNOPSIS

Common subrou4ne used by mch.s/subyte and mch.s/suword to store a word into the user's virtual address space.

DESCRIPTION

Performs the inverse operation to that of mch.s/gword. It uses the Move To Previous Instruction Space(MTPI) instruction to move "word" to the user virtual address specified by "addr". As with the mch.s/gword routine, a trap (Segmentation violation) might occur because of an illegal address ("addr") so that the trap catcher("nofault") is set up.

retu

CALL

retu (addr)
int *addr;

RETURN

No value is returned.

SYNOPSIS

Restores the stack position(U block) of a process and places a user's per process information within the system's virtual address space. Used for restarting a process.

DESCRIPTION

The address "addr" is the location of a process. (It is the contents of the "p_addr" entry of a Process Table entry. The U block is the first 1024 bytes at this address and the data and stack segments of the process follow immediately.). It is in memory block(32 word) granularity so that it may be directly loaded into Memory Management Registers(Kernel Instruction Address Register 6 - KISA6 for 11/40's or Kernel Data Address Register 6 - KDSA6 for 11/45 or 11/70 processors). Mch.s/retu places a user's U block in the system's virtual address space and helps restart the process by restoring athe stack position of the process from the U block. A higher level routine(slp.c/swtch) will issue a C subroutine return(jmp cret - return statement) which will restore the general purpose registers and the PC from this stack frame. Mch.s/retu always restores SP and R5 from the array "u_rsav" in the U block ("u_rsav" is the first entry in the per process information area of the U block.) The stack position(SP and R5) was saved previously by the function mch.s/savu.

Mch.s/retu is not called unless a process is in core as the "p_addr" entry contains the swap address of the process when it is non resident. All interrupts must be disabled to insure that the context can be changed without interruption. To do this, the processor's priority is raised to 7 while the SP, R5 and KISA6 (or KDSA6 on I and D space systems) is being restored. The processor's priority is then lowered to zero to allow processing of all pending interrupts.

savfp

CALL

jsr pc, _savfp
savfp()

RETURNS

No value is returned.

SYNOPSIS

Saves floating point registers. A dummy routine is used for machines that do not have floating point.

DESCRIPTION

If a process is to be preempted as the result of an interrupt waking up another process or of a system call completing, the floating point registers must be saved. (This is true, of course, only for machines that have a floating point units 1/45,11/70.)

The registers are saved in the U block area ("u_fsav") in the following order:

1. Floating Point Status Register
2. FR0 - Floating Register 0
3. FR4
4. FR5
5. FR1
6. FR2
7. FR3

Special precautions must be taken in saving registers FR4 and FR5 as they cannot be referenced using addressing modes other than zero (direct addressing - see DEC Processor Handbook).

The mch.s/savfp function is a dummy routine on machines that do not have floating point hardware. UNIX uses double precision floating point at all times so that each saved register represents 4 words. (The floating point status register is only cite word for a total of 25 words to be saved in "u_fsav").

savu

CALL

savu (save)
int *save[];

RETURN

No value is returned.

SYNOPSIS

Saves the stack position of a process executing within the operating system.

DESCRIPTION

C language functions store the subroutine return address on the stack. The first action performed in the call function is to build a stack frame. Register RS is set to the current position of the stack pointer(SP) as the beginning of this stack frame and the general purpose registers are saved on the stack. Local variables are allocated on the stack frame directly below the register save area. Register R6(SP) is set up just below the local variables and acts as a local stack pointer.

By saving R5 and SP and by establishing a convention about relinquishing the processor, only registers R5 and SP need be saved to save a process's context. The general registers are saved (by mch.s/csv) and restored (by mch.s/cret) on the stack every time an interrupt or trap occurs and every time a C language function is called so that as long as the location of these registers(i.e., R5 is saved) the process can be restarted.

The mch.s/savu, function performs the service of saving the current value of R5 and SP of the caller of mch.s/savu in the array specified by "save". When a process is to be restarted, a converse operation, in which the SP and R5 are restored (mch.s/retu or mch.s/aretu) is performed. It is also necessary to restore the general purpose registers (including the PC) of the restarted user. This can be done by merely executing a C subroutine return (mch.s/cret) once the SP and R5 have been restored.

The process Switcher saves the general registers and context(mch.s/savu) of a process as it is the common routine used to change execution from one process to another. However, by having several arrays("u_rsav","u_qsav","u_ssav") where the stack position(SP and R5) can be saved, a process may resume by returning from a function other than the process Switcher.

In particular, the "u_rsav" area is used to save the stack position of processes that willingly relinquish the processor(slp.c/sleep) or are preempted. The "u_ssav" area is used by processes that do their own swapping(excluding the Scheduler) and is set up so that once the process is brought back into memory, it will be restarted in the code that did the swapping and not by returning from the process Switcher (A flag must be set SSWAP in "p_flags" to inform slp.c/swtch to restore the stack position from "u_ssav" and not "u_rsav") The last area "u_qsav" is a special area which has the stack position of a process saved every time a system call is made. It is useful, not for resuming a process at a non-standard place, but in aborting a system call when a signal is caught. The "u_rsav" area is also used by the slp.c/expand function to restart execution of a process after moving it to a different location in memory.

Because of the critical nature of saving contexts mch.s/savu must raise the processor's priority to 7 to insure that no interrupts modify any registers until the context is changed.

setreg

CALL

mov instr, r0
jsr pc, setreg

RETURNS

Returns in the lower byte of register R2, the register that was modified by the destination field of an instruction and the amount that it was modified by. The high byte of register r2 is untouched.

SYNOPSIS

An assembly language function used by the mch.s/backup function on 11/40's to simulate the operation of Memory Management Status Register 1.

DESCRIPTION

The amount that a register can be incremented or decremented (automatically) in a PDP-11

machine language instruction is dependent on the addressing mode and on the type of instruction(byte or full word). The amount of the modification for the addressing modes is;

a. Autoincrement (+I for byte instruction, +2 for full word instruction).

b. Autoincrernent deferred (+1 for byte and +2 for full word).

c. Autodecrement (-1 for byte, -2 for full word).

d. Autodecrement deferred (-I for byte arid -2 for full word).

This function examines the destination field in the instruction "instr" passed to it and places in the Iow byte of register R2 the number of the register that was modified toy the destination field and the amount that the register was incremented or decrernented. It is put in the same forrn as the Memory Management Status Register I reports it for 11/45 and 11/70 processors. This functions is used only on 11/40's to simulate the operation of Status Register 1. For a full description of instruction backup see mch.s/backup.

spl0

CALL

spl0()

RETURNS

No value returned.

SYNOPSIS

Changes the processor's priority to 0.

DESCRIPTION

Spl0 changes the hardware priority of the processor (found in the processor status word) to 0.

spi1

CALL

spl1()

RETURNS

No value returned.

SYNOPSIS

Changes the processor's priority to 1.

DESCRIPTION

Spl1 changes the hardware priority of the processor (found in the processor status word) to 1.

spl4

CALL

spl4()

RETURNS

No value returned.

SYNOPSIS

Change's the processor's priority to 4.

DESCRIPTION

Spl4 changes the hardware priority of the processor (found in the processor status word) to 4.

spl5

CALL

spl5()

RETURNS

No value returned.

SYNOPSIS

Changes the processor's priority to 5.

DESCRIPTION

Spl5 changes the hardware priority of the processor (found in the processor status word) to 5.

spl6

CALL

spl6()

RETURNS

No value returned.

SYNOPSIS

Changes the processor's priority to 6.

DESCRIPTION

Spl6 changes the hardware priority of the processor (found in the processor status word) to 6.

spI7

CALL

spl7()

RETURNS

No value returned.

SYNOPSIS

Changes the processor's priority to 7.

DESCRIPTION

Spl7 changes the hardware priorifirof the processor (found in the processor status word) to 7.

start

CALL

jmp start

RETURN

No value is returned.

SYNOPSIS

Sets up the virtual address space for the operating system.

DESCRIPTION

The difference between initializing the operating system's virtual address space for 11/40's is sufficiently different from that of 11/45 and 11/70's that a separate discussion of each will be given.

On 11/40 processor's, there is only are one set of General Purpose Registers(set 0) and there are only two sets of Memory Management Registers(User and Kernel). Each set of Memory Management Registers does not have a Data Space Register Set in addition to the normal Instruction Space Register Set. The Memory Management Unit on the 11/40 also does not possess a register(Status Register 1) which aids in backing up instructions(see mch.s/backup discussion). When UNIX is booted into memory by one of the BOOT prograrns(UBOOT, HPBOOT, RKBOOT, TBOOT, MBOOT, etc), it is in the same form as a standard object file on the system. That is, there is an 8 word header followed by the text(instructions) segment. This is followed by any initialized data and any uninitialized data(bss). The size of the uninitialized data(bss) is indicated in the header. The end of the three respective areas are indicated by the following three addresses supplied by the UNIX loader "_etext", "_edata", "_end".

When UNIX is booted into memory, relocation by the Memory Management Unit is off(bit 0 of Status Register 0 turns relocation on). All addresses are mapped into the lower 32K words of memory. Since the operating system is the only software to interact with the device registers, the virtual address map for the Kernel must be set up to access this region. Also, the operating system's stack shifts from area(U block) to area within the full range of memory depending on which process is executing. To perform the address mapping required for these functions Kernel Memory Management Instruction Address register 7(KISA7) is set up to map virtual memory references 28K-32K word in Kernel virtual address space into the high memory Unibus address area (18 bit pysical addresses 124K-128K). Kernel Memory Management Instruction Address register 6 (KISA6) is used for mapping the system stack and the U block area. The U block area is a block of memory 1024 bytes in size which contains per process information. The "u" array is in the low physical address part and the system's stack area is in the high physical address part of the U block. The first six Kernel Instruction Address registers are set up to map the first 24K of physical memory into the corresponding first 24K of virtual address space. The boot programs bring the operating system into this area of memory so no rearrangement of the object file is necessary as on I and D space systerns(see below). The area taken up by the operating system may actually be smaller than the 24K that can be accessed by the memory map ,however, the address map is not adjusted to constrain the operating system. Two other functions must be perfdrmed before the system can be initialized(by main.c). The system allocates the first 1024 by area (on a 64 byte address boundary) as a U block for the Scheduling process. The Page Length Field in all of the Kernel Instruction Address registers is set up to map a full 4K word area and the access control permissions for read/write no abort(ACF = 06 is set). KISA6, which maps the U block, is set up with the same permissions but mapping for only a 1K byte area. Once the Kernel virtual address map has been set up, relocation can be turned on(by setting bit 1 of Status Register 0 in the Memory Management Unit) and the uninitialized data(bss) and Scheduler U block area can be zeroed. The bss area is the area between "_edata" and "_end". Before calling main.c to initialize the system, the Processor Status is set up so that the Current

Mode field indicates Kernel, the Previous Mode indicates User and the processor's priority is lowered to zero. This is done because it is an appropriate start of state for the processor, allowing interrupts and appearing as if a system call from a user process was made.

There is one additional function performed by mch.s/start. This is in conjunction with main.c which spawns the INIT process. A return(RTS) to the mch.s/start routine is executed by main.c on behalf of the INIT process after it has handcrafted a small programricodel in user virtual address space. This program will cause an exec system call to be issued and the INIT process to be brought into memory. The whole operation appears as if a system call was made by the (nonexistent as yet) INIT process. The mch.s/start program simulates the return from the system call by setting up the system's stack so that a return from trap(RTT) instruction can be executed. An appropriate PS(indicating both Previous and Current Mode of the processor to be User) and PC (set to resume execution at virtual address 0 in user virtual address space) is loaded onto the stack before executing the RTT.

For 11/45 and 11/70 processors there is a better Memory Management Unit which allows larger virtual address spaces. In particular, there are three sets of Memory Management Registers; Kernel; User, Supervisor. Each Memory Map is divided into two sections, one for mapping instruction fetches(Instruction space) and one for mapping data fetches (Data space). To take advantage of the expanded virtual address space available in these processors the object file is relocated differently. The major reason for this is that when an interrupt or trap occurs the new PS and PC loaded from the low memory vector area are mapped by the hardware into the low virtual D space area. The low core vectors are however hardwired to low physical core. This means that at least low physical core virtual Data must be in low memory. Also, by arranging that the system buffers have the same physical and virtual addresses, relocation need not be performed when doing buffered I/O. The approach taken for UNIX I and D space systems was to rearrange the object file for the systern(using the SYSFIX command) so that the data precedes the text_Several text portions of the system, notably the C interface area, the post mortem dump (mch.s/dump) and the mch.s/start routine are loaded in Data Space. This is done to save Instruction space or because the routines are used only once. The boot programs are exactly the same as for the 11/40, the only difference being the interchange of the text and data portions of the object file which are transparent to the object programs. The virtual address mapping set up by the mch.s/start routine is also different and the meaning of the symbols "_etext", "_edata" and "_end" supplied by the loader is different. In addition, the operating system must be loaded using a special option on the loader.

When the loader performs relocation on an object file that is to have I and D separation, all of the text is loaded assuming that the first location is virtual address 0 in *Instruction Space*. The data and bss segments are loaded assuming that the first location available for data is also virtual address 0 in *Data Space*. The meaning of "_etext", "_edata" and "_end" at this point is as follows,

"_etext" - This is the last address occupied by instructions in Instruction Space and since the text has been loaded at virtual address 0, it also corresponds to the site of the text.

"edata" - This is similar to the "_etext" symbol since the data is loaded in its own virtual address space starting at virtual 0. It also corresponds to the size of the data segment.

"end" - This is the last virtual location in Data Space. The size of the bss area is "end" - "_edata".

The SYSFIX command must be run on UNIX object files that are to be separated into I and D space. This prtigram interchanges the position of the data and text portions of the object file (text is normally first) and also relocates the text portion so that the first address is at 4K word virtual Instruction Space. This means that "_etext" is increased correspondingly and no longer represents the size of the text.

With the new object file provided by the SYSFIX program, some adjustments of the position of the text(to make room for the bss) in physical memory is still necessary when the system is booted into memory. In order to utilize fully the address space provided by the Kernel Data Space, it is desirable to place the bss segments between the data and text. To do this, the text must be shifted in physical memory to a higher address area which may be beyond the 32K word address space that can be addressed with relocation off. The Memory Management Registers must therefore be setup to accomplish this and relocation

turned on to do the move. (This was not necessary on non I and D space systems since the total text, data ,and bss area was limited to 32K words.)--The Data Space registers are setup so that their 32K virtual address space maps into thel first 32K of physical memory. The last seven of the Instruction Address Space Registers are setup so that they map 28K of virtual address space(beginning at 4K virtual) into the physical area of memory that the text is to reside in (immediately below the data and bss). Instruction Address Register 0 is set up so that the first 4K virtual Space overlaps the 4K physical area already mapped by Kernel Data space Register 0. (This is done because mch.s/start is in the data area as is the C interface area and they must reside in Instruction Space to be executed. Relocation may now be turned on and the text moved to higher physical memory by using the MTPI instruction with the PS setup to indicate that the Current and Previous Modes are Kernel. The bss area is then zeroed. Finally it is necessary to setup Data Space Address Register 6 to point to a U block for the Scheduler(as was done for the 11/40). Space for the U block, is allocated from the first available memory beyond the operating system. It is cleared and a stack is established in the lower portion of the U block for the system. Data Space Address Register 7 must be set up to map the Unibus addresses. (There is a contradiction in terms at this point as one cannot set a data location which is not already within your virtual address space once relocation has been turned on.) This is done by using an addressing mode escape(see DEC Memory Management Maintenance Manual) for the MTPI instruction which allows the I/O area to be accessed and therefore Data Space Address Register, to be set even though no mapping exists for that area. All of the Memory Management Registers are set up to map 4K regions of memory, except for Data Space Register 6 which maps the 1K byte area of the U block. All Memory Management registers are set up for read/write no abort/trap action (Access Control Field - ACF -- 06) and for the expansion direction to be toward high physical addresses(ED = 0). The operation of the mch.s/start routine in conjunction with main.c and the INIT process is the same as for 11/40's.

The 11/70 processor is capable of addressing more memory than either the 11/40 or 11/45 by turning on 22 bit mode addressing in the Memory Management Unit(bit 4 of Status Register 3 is set). Currently only 18 bit addressing(i.e., 11/40, 11/45 mode) is supported on 11/70 processors.

subyte

CALL

```
subyte(virtual, c)
char   c;
char   *virtual;
```

RETURNS

A -1 is returned if the character "c" cannot be placed in the user's virtual address spaces.

DESCRIPTION

This is the analog of mch.s/subyte. Most of the work in storing a byte in the user's address space is done by mch.s/gword(get a word from user virtual address space) and mch.s/pword(place a word in user virtual address space).

Because the only instructions available for moving data between address spaces(MFPI, MTPI, etc.) operate on words, the full word that contains the byte to be overwritten must first be fetched(by mch.s/gword) and the byte carefully inserted before the word is replaced in the user's address space (by # mch.s/p word) . Both mch.s/pword and mch.s/gword are equipped to deal with errors occurring in storing across address spaces and will result in mch.s/subyte returning a -1. They set up the external variable "nofault" so that it contains the address of their own internal error handler so that the trap handler will return control to them instead of processing the error as a trap. This function is used chiefly in moving characters from the I/O subsystem to the user's address space(see subr.c/passc).

suword

CALL

```
suword(virtual, word)
int word;
int *virtual;
```

RETURNS

-1 on failure.

SYNOPSIS

Stores a word in user virtual address space.

DESCRIPTION

Mch.s/suword and mch.s/subyte use the common subroutine mch.c/pword to place a word in the user's virtual address space at the address

"virtual".

trap

CALL

jmp trap

RETURNS

No value is returned.

SYNOPSIS

Trap is an assembly language interface which performs some additional work for interfacing to the trap handler(trap.c/trap).

DESCRIPTION

Additional work is required in handling traps besides the normal stack frame buildup that is done by the mch.c/call routine. First of all, the status registers in the Memory Management unit contain information about memory faults and information that will allow instructions to be backed up and restarted when a segmentation violation occurs. The three registers and the information they represent are, (For exact details see DEC Processor Handbook.)

1. Memory Management Status Register 0 - type of memory violation occurring.
2. Memory Management Status Register 1 - records any autoincrement/autodecrement of general purpose registers. (Necessary for backing instructions up instructions on 11/40's) .
3. Memory Management Status Register 3 - contains the virtual PC when the trap occured.

The trap interface saves the contents of these registers in a three word array "ssr" so that they may be used by the mch.s/backup routine. Once these registers are saved, relocation may be turned on by setting bit 1 in Memory Management Status Register 0.

Since all traps are handled by the same C function (trap.c/trap) there is no jump table entry in low core as there is for the interrupt handlers. These low core jump table entries are, however, responsible for saving register R0. The assembly

language mch.s/trap routine must do this by executing the following sequence,

jsr r0,call1; _trap

This is in the same format as the low core jump table entries and is used both to save register r0 and to interface to the mch.c/call routine. The function mch.s/calll is a special entry point in the mch.c/call routine for the trap handler. It merely adjusts the stack pointer so that the interface to the mch.c/call routine will be smooth and lowers the processors priority to zero so that any interrupts that may be pending can be processed. (Note - The low core vectqfs indicated that the processor's priority should be raised to 7 when handling a trap). This is to prevent another trap from destroying the contents of the Memory Management Status registers before they could be saved and relocation restarted. In moving data between address spaces, (using mch.s/copyseg, mch.c/fuword, mch.s/copyin, etc. i.e.,any routine which uses the MFPI, MFPD, etc. instructions) there is the possibility that the user process has improperly specified to the system the virtue address to or from which data is to be moved. UNIX allows these specialized routines do their own trap catching. These assembly language functions place in a external variable "nofault" the address of their own error handler. If a trap occurs, the mch.s/trap function Will transfer to the address specified by "nofault" rather than calling the trap handler(trap.c/trap).

nami

CALL

nami(func, flag)
int (*func)(), flag;

RETURNS

Pointer to the (locked) Mode table entry for a file or its parent directory; zero if an error occurs. See the following description for specifics regarding return values.

SYNOPSIS

Nami is the basic mechanism for converting a file pathname to an i-number and a system Mode table entry. It is used to either locate an existing file, or, prior to creating or deleting a file, to locate the parent directory.

DESCRIPTION

The file pathname that nami is to manipulate is pointed to by the U block directory pointer variable (u_dirp). The first argument ("func") is the address of the function to be used to obtain the characters of the file's pathname (nami.c/schar if the name is in kernel space, nami.c/uchar if in user space). The second argument ("flag") indicates what operation is to be performed:

0 if the name is to be located

1 if the name is being created

2 if the name is being deleted

The algorithm used to translate the pathname to the file's i-number is as follows:

1. If there are unprocessed filename components remaining in the pathname, then the component just matched must be a directory. This pathname component will be referred to as the "current component". In the first pass through the algorithm, the "current component" is either the current directory (u_cdir) or the root directory, as determined by the pathname. At this point it is verified that the user has execute permission for that directory (ie. the "current component").
2. Get the "next component" of the pathname and place it in a workspace (u_dbuf).
3. Read the "current component" file (which is a directory) block by block, searching for an entry for the "next component".
4. When the "next component" is matched, obtain its i-number and make the next component" the "current component".
5. Repeat the above process until the end of the pathname is reached.

Exits from the preceding algorithm are made at appropriate locations whenever either an error is detected or the function call has been satisfied (as determined by "flag"). It should be noted that the purpose of nami is always to locate files, never to actually create or destroy them, even when called for creation or deletion (flag 1 or 2). When called for these reasons, nami only traces the file's pathname and verifies that the file may be created or destroyed (that is, ensures that the user has write permission in the file's parent directory).

The return values from narni depend upon the reason it was called.

Reason Return Values

Find (flag=0) Zero if the file is not found. If the file is found, a pointer to a (locked) system mode table entry (model]) for the file.

Create (flag=1) Zero if the file does not exist; in this cases, u_pdir points to the system inode table entry of the file's parent directory, u_offset[11 is the offset to an empty directory slot in the parent directory, and u_dbuf is the last component of the filename. If the file already exists, a pointer to the file's (locked) system Mode table entry is returned. Naafi is often called with flap.. 1 wheg a file's existence is in quesOon and it is to be created if it doesn't exist.

Delete (flag=2) Zero if the file is not found. If found, a pointer to a (locked) system Mode table entry for the file's parent directory is returned, u_dent contains the file's directory entry in the parent directory, and u_offsetEl I contains the offset in the parent directory to the entry immediately following that of the file.

If any errors (e.g. access, permission, invalid pathname, etc.) are encountered during nami processing, the appropriate bit(s) will be set in u_error.

schar

CALL

schar()

RETURNS

A character from kernel space.

SYNOPSIS

Schar is used, usually when parsing directory names, to obtain the next character in the name.

DESCRIPTION

Schar returns the character pointed to by the U block's directory name pointer (u_dirp). The address is interpreted as a kernel space address and is then incremented to the next character. This function is typically used by narni.c/nami when parsing a file pathname that is contained in kernel space. Note that it is the caller's responsibility to check the character returned by schar for an end-of-string indicator (which is usually a null character). The nami.c/uchar function performs a similar service for names in user space.

uchar

CALL

uchar()

RETURNS

A character from user space or a -1 on error.

SYNOPSIS

Uchar is used, usually when parsing directory names, to obtain the next character in the name.

DESCRIPTION

Uchar returns the character pointed to by the U block's directory name pointer (udirp). The address is interpreted as a user space address and is then incremented to the next character. This function is typically used by nami.c/nami when parsing a file pathname that is contained in user space (see nami.c/schar for the kernel space version). A return of -1 indicates an error; typically, a memory fault while trying to access the specified address. If this occurs, the error bits (u error) are also set. Note that it is the caller's responsibility to check the character returned by uchar for an end-of-string indicator (which is usually a null character).

panic

CALL

panic(str)
char *str;

RETURNS

No return.

SYNOPSIS

Idles the processor.

DESCRIPTION

Panic is used to gracefully bring down the system when a fatal error (i.e., out of swap space, out of i-nodes etc.) is encountered. After updating any mounted file systems (see alloc.c/update) and printing the message pointed to by "str" on the system console, the processor is placed in an endless idle loop.

printf

CALL

printf(format,a1,...,a9,aa,ab,ac)
char *format;
Arguments al-ac may be one of:
    char *al;
    int al;
    int al; (one char in low byte)

RETURNS

No value returned.

SYNOPSIS

Prints messages on the system console with format control.

DESCRIPTION

The system’s printf function performs exactly like the C library’s printf, except that it always prints on the system console and only a subset of the format control directives are available. In particular, within the format control string "format", only the conversion specifications c, d, l, o, and s are recognized and honored.

printn

CALL

printn(nurn, base)
int num, base;

RETURNS

No value returned.

SYNOPSIS

Prints a number on the system console in any base.

DESCRIPTION

Printn is used to print on the system console the value of "num" in base abases. This is accomplished by having printn recursively call itself to print the quotient of "num" divided by "base". The recursive calling is terminated when this quotient reaches zero. The remainder from each division is then printed as the system crawls back up the chain of printn calls. This has the effect of printing the remainders from the divisions in inverse order, which yields the desired result.

putchar

CALL

putchar(c)
int c;

RETURNS

No value returned.

SYNOPSIS

Prints a single character on the system console.

DESCRIPTION

Bypassing the device driver’s write routines, putchar prints the character "c" on the system console by manipulating the device’s registers directly. The character is not printed if the console switches are set to zero.

iomove

CALL

iomove (bp, o, an, flag)
struct buf *bp;
int o, an, flag;

RETURNS

None

SYNOPSIS

lomove moves an bytes from the byte poSition "o" in the buffer pointed to by "bp" to the users area according to "flag" (READ means from the pool to users area; WRITE means from the users area to the pool).

DESCRIPTION

Iomove uses information in the per user control block. The variables are:

u_base	core address of user’s I/O buffer
uoffset	byte position into the file
u_count	number of bytes to be read/written
u_segflg	flag for I/O; user or kernel space

If the source address; destination address, and count are all even in a copy to user’s space; then mch.s/copyin or mch.s/copyout is used to move words.

If the addresses are not even, then subr.c/cpass or subr.c/passc are used to move a byte at a time. The copying takes place from u_base to the buffer pool address "bp" plus the offset "o" for "an" bytes.

If either copy operation fails, an error indicator EFAULT is set in the per user control block error code (u.u_error).

The base pointer, u_base, is incremented by "an". The count, u_count, is decrernented by "an".

max

CALL

max (a,b)
char *a, *b;

RETURNS

The maximum of "a" or "b".

SYNOPSIS

Return the logical maximum of two arguments "a" and "b".

DESCRIPTION

Compare "a" and "b" and return the maximum.

min

CALL

min(a,b)
char *a, *b;

RETURNS

The minimum of "a" or "b".

SYNOPSIS

Return the logical minimum of two arguments "a" and "b".

DESCRIPTION

Compare "a" and "b" and return the minimum.

readi

CALL

readi (aip)
struct inode *aip;

RETURNS

None

SYNOPSIS

Read the file associated with the i-node pointed to by "aip". Companion routine with writei. Both are the workhorses for file system I/O.

DESCRIPTION

Readi uses information in the per user control block to help read the file associated with the inode pointed to by "aip". The variables are:

u_base	core address of user’s I10 buffer
u_offset	byte position into the file
u_count	number of bytes to be read
u_segfig	flag for I/O; user or kernel space

Because the offset is a 24 bit number, readi calls many auxiliary functions to handle the arithmetic. Readi will read into a buffer from the buffer pool, 512 bytes at a time, starting at u_offset within the file pointed to "aip". U_count bytes are moved from the buffer pool into the user area pointed to by u_base.

UNIX block I/O is read or written in multiples of 512 (one block) bytes. If u_count is less then 512 only one block ’need be read. If u_count is not a multiple of 512 bytes, the correct byte position within a block must be located. If u_count plus u_offset is greater than the size of the file, all data to the end of file must be read.

The i-node is first marked for updating the last-accessed time. If the i-node is a character special file (device) then the appropriate device read routine will be switched to using the major device number found in the first address entry of the i-node.

If the offset is greater than the size of the file then the file is position to the end-of-file and readi returns.

In reading block special files, readi maintains a read ahead protocol that saves the next block number and does a read ahead.

The algorithm is to do repeated reads of blocks and moves to the user’s area until u_count bytes have been read.

Subr.c/bmap is called to get a buffer from the buffer pool. Bio.c/bread is called to read the required block. Rdwri.c/iomove is called to move data (either the entire block or some portion) into the buffer pointed to by u_base. Bio.c/brelse is called to release the buffer back to the pool.

writei

CALL

writei (aip) struct inode *aip;

SYNOPSIS

Write the file associated with the i-node pointed to by "aip". Companion routine to readi. Both are the workhorses for file system I/O.

DESCRIPTION

Writei uses information in the per user control block to help write the file associated with the inode pointed to by "aip". The variables are:

u_base	core address of user’s I/O buffer
u_offset	byte position into the file
mount	number of bytes to be written
u_segflg	flag for I/O; user of kernel space

Writei will move u_count bytes from the user’s program at u_base to buffers in the buffer pool. It will then write the buffers to the file pointed to by "aip" at the position u_offset. Since UNIX I/O is block oriented, data is written in multiples of 512 bytes. If u_offset is less than the size of the file, then part of the file is being overwritten. The block at u offset must then be read, the appropriate bytes replaced and the block rewritten.

If the file is positioned at the end-of-file, new blocks must be allocated and written.

The i-node is first marked to update the last-accessed and last-modified times. If the i-node is a character special file (device) then the appropriate device write routine will be switched to using the major device number found in the first address entry of the i-node.

The block containing the offset is read into a buffer from the buffer pool. The minimum (N) of the number of bytes remaining in the block or the u_count is obtained. If this minimum is exactly 512 bytes a new block must be obtained. In either case, rdwri.c/iomove is called to move N bytes from the user area to the buffer obtained from the buffer pool by calling subr.c/bmap. Bio.c/fwrite is called to write the block. Bio.c/frele is called to release the buffer.

If the size of the .file is less than the offset then the size is set to tie new offset value. This is repeated until u_count bytes have been written.

core

CALL

core()

RETURNS

A one is returned if a core image is successfully produced.

SYNOPSIS

Produces a core image as a result of the standard system action on reception of certain signals.

DESCRIPTION

For program debugging purposes, it is convenient to have a system function which makes a copy of an aborted process in a file. The format of a UNIX core image is simply the 1024 byte U block followed by the data and stack areas. For non-reentrant processes the data area contains the text, data and bss areas however, for reentrant processes, the text is not included. The DB and CDB commands under UNIX can examine these images post mortem as an aid in debugging. The U block will contain the general purpose registers (including virtual PC) so that all information about the state of the program when it aborted is known.

All core images are produced in a file "core" in the working directory of the aborted process. The sig.c/core function essentially simulates a create system call followed by two writes into the file. The steps in the produciion of the core image are as follows,

1. A search is made to see if a file named "core" already exists in the working directory of the process. The nami.c/nami function performs this service. The file name "core" is passed to nami.c/nami by setting up "u_dirp". (Normally the trap handler sets up "u_dirp" for open, create, link, etc. system calls.) The address of a special routine used by nami.c/nami to fetch the string name (nami.c/schar) must be passed as an argument to nami.c/nami. If any errors occur due to conflicts in access permissions on an existing "core" file or if directory permissions are not correct, the core image is aborted.
2. If the file does not already exist, then an inode is allocated for it and given read/write by all access permissions (0666).
3. For an existing file with the name "core", the access permissions for the file must be checked to see if writing is allowed (fio.c/access). There are some serious bugs in the access checking setup for producing core images. For the super user, access privileges are ignored so that any directory named "core" could be overwritten.
4. Once a file has been established and access granted, the file is truncated (iget.c/itrunc).
5. The U block is then written out by setting up parameters for the rdwri.c/writei function. The file offset "u_offset" is set to zero, the address ("u_base") where the transfer is to begin is set to the address of the U block (0140000). "U_count" is set to the size, in bytes, of the U block and finally a flag, "u_segflg" is set to indicate to rdwri.c/writei that the data is within the operating system’s virtual address space. (The core image is actually produced by the terminated process. The mechanism is such that when a process determines that there is a signal pending for which a core image is required, it produces the core image itself before making itself a ZOMBIE).
3. The data and stack area are written into the file "core" by reloading the User Memory Management registers s6 that they are set up as if the entire program was only data. (Main.c/estabur is called to setup the prototype Memory Management registers. It calls main.c/sureg to actually load the hardware registers.) "U_base" is set to the address (0) in the user’s virtual address space where the transfer is to begin, "u_count" is set to the size in bytes to be written and "u_segflg" is set to zero to indicate (to rdwri.c/writei) that the write is to occur on data located in the user’s virtual address space.

When the core image has been produced the inode for the "core" file updated on the filesystern (iget.c/iput). Any errors occurring in the write procedure above will result in only part of the core image being written out. A successful core image is indicated by sig.c/core returning zero (to sig.c/psig).

issig

CALL

issig 0

RETURNS

A 1 is returned if a signal has been sent to a process and some action by the system or the user is required.

SYNOPSIS

Determines whether a signal requiring action is pending for the currently running process.

DESCRIPTION

Processing of signals only occurs when the process that is sent a signal determines that a signal is pending. Sig.c/issig is used to make this check. Signals are posted in the "p_sig" entry of each Process Table entry. If the entry is nonzero, a signal is pending and the value in "p_sig" is the signal number. This is used as an index into the Signal Table ("u_signal[]") of the Per Process information area (U block). There is one entry in this array for each possible signal and the value of the entry determines what action is to be taken. An odd value in the table indicates that the signal is to be ignored. Nonzero even values indicate that some action is to be taken by the user process, while zero entries indicated that the standard system action is to be taken.

psig

CALL

psig()

RETURNS

No value is returned.

SYNOPSIS

Signal processor.

DESCRIPTION

Signals may be ignored or caught by a user process, or the standard system action may be taken. The Signal Table, "u_signal[]" in the user Per Process information area (U block) determines what the action for a particular signal is to be. There is one entry in the "u_signal[]" array for each of the 20 possible signals. A zero in an entry indicates that the standard system action is to be taken, while an odd value indicates that the signal is to be ignored. Any even value indicates that the user process is to handle the signal and is the virtual address of a jump table in the signal library function in the program. This jump table contains the address of the function within the user's program to be executed when a particular signal is received. The transfer is made through an intermediate jump table in the user's library function so that the general purpose registers may be saved and restored on the user's stack. In order to provide for the continued execution of a program once the user's process has handled the signal, there is a mechanism for returning the process to normal execution. This is done by having sig.c/psig adjust the user's stack so that the library functions can execute an RTT instruction. In summary, signal processing is done as follows:

1. The signal is reset (0) so that the standard system action will be taken on the next occurrence of the same signal.
2. The Program Counter and Stack Pointer, saved when the user process made a system call, is placed on the user's stack. This is done so that when the signal library function within the user's program has done it's work, a return can be made to the system call that was aborted (via a RTT executed in the user's signal library interface function). It is unfortunate that the aborted system call is not restarted, so that some arrangement is necessary to check that the system call was completed.
3. The Program Counter saved on the stack frame is set to the value in the appropriate Signal Table ("u_signal") entry. This is the address of a table in the user's library interface to the signal system call and is essentially a jump table for calling the user's signal handler.

The standard system action on reception of the following signaliiis to produce a core image in the working directory of the process receiving the signal.

1. Quit
2. Illegal Instruction
3. Trace Trap
4. IOT Instruction
5. EMT Instruction
6. Floating Point Exception
8. Bus Error
9. Segmentation Violation
10. Bad System Call

The remaining signals, Hangup, Interrupt, Kill, and that are undefined do not produce a core image as a standard system action, but cause, the termination of the process.

Before the process is terminated, the lower 8 bits of the user process register R0 and the type of signal received are saved in the "u_arg[0]" entry of the U block, so that they may be found by the parent of the terminated child. The truncated value of R0 is placed in the high byte and the signal number in the lower byte. A successful core image is indicated by having the high order bit in the signal byte set.

Determining whether a signal is pending is done at three significant points within the system.

1. On every return from a system call, sig.c/issig is called. This checks to see if a signal is pending for active (executable) processes within the system.
2. For processes that will roadblock at a low priority (WAIT priority), a check is made before and after the process is roadblocked to see whether a signal is pending. This is done because the interval that the process is roadblocked may be long, and because some of the system calls (notably wait and sleep) will continuously roadblock a process until the desired event has occurred. By making the check a standard part of this loop the signal can be processed and the system call aborted if needed. This will result in either the standard system action (termination of the process) being taken, or preparation for allowing the user to handle the signal. If the user is to process the signal, the system call is aborted. and a nonlocal goto (mch.s/aretu) to the trap handler is done after the signal catcher in the user's process is set up.
3. The last check is made in the clock interrupt handler (clock.c/clock), when it preempts CPU bound processes. This must be done to insure that CPU bound processes (which never make system calls) can be terminated by a signal (usually quit, interrupt or a kill sent by the kill system call).

psignal

CALL

psignal(p, sig)
struct proc *p;

RETURNS

No value is returned.

SYNOPSIS

Sends the signal "sig" to process up".

DESCRIPTION

Signal sending does not result in any immediate action. Rather, the signal number("sig") is posted in the Process Table entry for that process ("psig") the process. This number is used as an index into a table in the U block for that process ("u_signal[]"), which contains the appropriate action to be taken for each signal.

Processes that are roadblocked at low software priority must be notified that a signal is pending as the event which they are waiting for may never occur. No wakeup is sent to the process, since the location of the process in the Process Table (argument "p") is known ("u_uprocp" in the U block). The process need only be made ready ("p_stat" = SRUN) and the event for which the process was roadblocked ("p_wchan") is zeroed. A situation may exist where there are no ready processes in memory and the process that was awakened is resident on the swap area. The Scheduler is awakened in this case. (The Scheduler would be roadblocked with the external variable "runout" set to a nonzero value, if there were no ready processes in memory.)

signal

CALL

signal(tp, sig)
struct tty *y$tp;

RETURNS

No value is returned.

SYNOPSIS

Sends the signal "sig" to all processes whose controlling teletype is "tp".

DESCRIPTION

Most processes spawned under UNIX are generated as a result of action at a user's teletype. The teletype is made the controlling teletype for

these processes and has the ability to terminate these processes when either the special characters quit or interrupt is sent by the teletype. The sig.c/signal function is used by the teletype I/O subsystem to send the quit and interrupt signals generated by a user at his controlling teletype to all of the processes having that teletype as it's controlling teletype. Since each Process Table entry contains a pointer "p_ttyp" to the controlling teletype, the Process Table need only be scanned and all entries having "tp" (a pointer to the teletype) as the controlling teletype are sent (by calling sig.c/psignal) the signal "sig".

expand

CALL

expand(newsize)

RETURNS

No value is returned.

SYNOPSIS

Enlarges or contracts the size of an existing process.

DESCRIPTION

This function is used to increase or decrease the area occupied by a process. It knows nothing about the contents of this area except that it is contiguous. As such the two functions (trap.c/trap and sys1.c/sbreak) which use it for stack growth and dynamic storage allocation, respecitvely, must do any shifting of parts of the program to adjust virtual addresses.

Since there may not always be enough memory available to grow a process in core, several cases, similar to those of slp.c/newproc occur.

1. If a process's size is to be decreased ("newsize" is the new size of the process including U block and it can be compared to the existing size "p_addr") the extra memory is freed (by calling malloc.c/malloc). Memory is freed from the high (physical address) end of the existing area.

2. If the process is to grow in size, physical memory must be added contiguously to the end of the existing process. Instead of attempting to add the increment of core to the existing area, a request is made (malloc.c/malloc) for a new area of memory having size "newsize" (old plus new). This is appropriate as the memory allocation scheme uses a First Available Fit algorithm to allocate memory and the chances of an incremental portion of core being available directly below a process are slim. Since the memory may not be available, two procedures are used,

a. If memory is available, the process is simply copied into the new area. The old image is abandoned (malloc.c/mfree) and the user's Memory Management registers are reset (mch.s/sureg).

b. If memory is not available, then a scheme similar to that used by slp.c/newproc is used. No tricks need by used in this case, since only one process is involved. The process merely saves it's stack position (in "u_ssavi, swaps itself out (text.c/xswap), marks itself (SSWAP in "p_flag") so that its context will be restored from "u_ssav" and not wr_sav" and calls the process Switcher (slp.c/swtch) to relinquish the processor. It may seem strange that a process can continue execution after having swapped itself, however, the processor is relinquished immediately after the swap I/O is completed. (slp.c/swtch)so no problems occur. In preparing for the swap, text.c/xswap frees the memory that was occupied by the swapped process.

newproc

CALL

newproc()

RETURNS

A zero is returned.

SYNOPSIS

Spawns a new process in the system.

DESCRIPTION

This is a subroutine used by sys1.c/fork to spawn a new process. It provides for the inheritance of essential characteristics of the new process. The new formed process is an image (exact copy) of the parent (spawning) process. The chief features of the newly created process are:

It receives a new unique process ID ("p_pid").

It inherits the controlling teletype ("p_ttyp") of the parent.

It inherits any shared (reentrant) text. The use count "x_count" and memory usage count "x_ccount" are increased for that text entry in the Text Table ("p_textp").

It inherits the User ID ("uuid").

It is marked to indicate the identity of it's parent (the parent's process ID is kept in "p_ppid").

It's age is set to 0, as it is newly created.

It inherits all of the open files belonging to the parent process ("u_ofile[]") and the instance count ("f_count") of each file in the File Table is incremented.

It inherits the context, working directory, signals, etc. of the parent. In short, all information kept in the per process (U block) area is inherited.

A parent process creates the child as a result of executing the sys1.c/fork system call, There may not be enough memory to replicate the process in memory, so a procedure for replicating the

process on the swap area must be available. The steps in actually spawning the image once the Process Table entry has been set up are:

1. The stack position of the parent is saved ("in "ursav"). This is done so that when the child is created as an independent entity, it will resume execution by executing a return from slp.c/newproc. The value returned by the child will actually be a 1 (since even though the stack position is restored by slp.c/swtch,the return that is executed is from slp.c/switch). The significance of this is that the sys1.c/fork function calls slp.c/newproc as a subroutine and uses the value returned to it to identify the child so it can initialize the accumulated execution time of the child. Thus, when a process makes a fork system call (sys1.c/fork), two processes will return from the slp.c/newproc call. One of the processes (the parent) will actually execute the return from slp.c/newproc, returning a zero. The child process will appear to have returned from slp.c/newproc, but this will seem this way because it's stack position was saved by the parent. It will return a 1 since the process Switcher returns a 1.

2. A chunk of contiguous memory equal to the size of the parent process is requested (malloc.c/malloc). There are two cases which may occur:

a. If memory is available, then a new image can be created simply by doing an in core copy of the parent. The copy is done by copying memory blocks (mch.s/copyseg) from the parent.

b. If a piece of contiguous memory is not available, the the image must be made on the swap area. (It might be considered that the forking process could be delayed until memory was available, however, for processes large enough to fill user memory the copy would have to be made on the swap device anyway.) The parent process is placed in an idle state ("p_stat" =- SIDL) to insure that the Scheduler will not swap the process out. The context is saved(mch.s/savu) again (in "u_ssav"). This is done becuase a process is being swapped by a system function other than the Scheduler. A swap out is requested (text.c/xswap) and the parent process is roadblocked while this occurs. (It should be remembered that the parent is doing all of the work associated with creating the child and therefore, incurs any delay in accomplishing this. It should also be remembered that text.c/xswap determines whether a process is reentrant and thereby need only swap out the data and stack portions if the parent is reentrant.) When the swap is completed and the parent is allowed to continue the creation process, it marks the child process ("p_flag" = SSWAP), so that when the child is restarted, it's stack position will be restored from "u_ssav" and not "ursav". The parent is removed from the idle state and made ready ("p_stat" = SRUN). In the process of creating the child a trick is used, so that the child ends up in the proper state when the swap is completed. The child process is made to assume the identity of the parent (by adjusting it's process size "p_size" and address "p_addr" in the Process Table entry to assume those of the parent). Thus the child will is roadblocked while the swap occurs and consequently, the child is awakened ("p_stat" = SRUN) when the swap is complete. After the swap is completed, the identities are restored.

Once the image has been produced, the parent simply returns (a 0) to the caller. The child will eventually be selected by the process Switcher to run and will execute a return (returning a 1) from slp.c/newproc through the process Switcher (slp.c/swtch).

If the Process Table is full when slp.c/newproc is called, the system panics ("PANIC NO PROCS").

The slp.c/newproc function is also used at startup time (ruain.c/main) in handcrafting the INIT process. Essentially, the operating system forks to create a mirror image (of the system's U block) before the "icode[]" program is installed in the child process (forerunner of INIT).

sched

CALL

sched()

RETURNS

No value is returned.

SYNOPSIS

The UNIX Scheduler.

DESCRIPTION

The Scheduler is an endlessly looping process that runs entirely in Kernel Address space. It is always process zero in the system, is always locked in core, and does not receive any signals. The main.c/main function creates and involks the Scheduler. No return to main/main.c ever occurs. The Scheduler has it's own U block, which is borrowed by the process Switcher when it changes execution- from one process to another. The Scheduler's only function is to determine which processes are to be allowed into core and which processes are to be swapped out.

The Scheduler uses the age of processes in determining which processes are to be swapped into or out of memory. The age of a process is kept in "p_time" and is updated by the clock interrupt handler once every second. Since age changes once per second, the Scheduler will run approximately once every second. The Scheduler runs in a piecemeal fashion, selecting a process to swap in or out, roadblocking while the swap I/O occurs, then rerunning. The amount of rearrangement done is limited by the amount of physical memory available and by the Scheduling criteria. The Scheduler is not the only function within the system which does swapping. Processes may swap themselves out or create copies of themselves (via sys1.c/fork) on the swap area however, the Scheduler is the only function that can bring a process into memory.

The Scheduler will roadblock itself at three times

1. While swap I/O occurs, the Scheduler is roadblocked.

2. If there are no processes on the swap area in the ready state, the Scheduler sets the "runout" flags and roadblocks itself. When any process is awakened (by slp.c/wakeup) the Scheduler is awakened.

3. When there is not enough core available to bring a ready process on the swap area into memory, the Scheduler sets the "runin" flag and roadblocks itself. The first process that is willing to give up core (e.g., terminates or goes into the WAIT state) causes the Scheduler to be awakened.

The algorithm used by the Scheduler in rearranging memory is:,

1. The oldest process on_the swap area that is in the ready state ("p_state" SRUN) is found. If there are no ready processes on the swap area, the Scheduler sets the "runout" flag and roadblocks itself. When the Scheduler is awakened, the algorithm is repeated.

2. The total size of the selected process is determined. If the process is reentrant and the text is not already in core (core usage count "x_ccount" is zero) this means that the size of the text ("x_size") must be added to the memory requirements for the process ("p_size").

3. A check is made to see if there is enough available memory to bring the process into memory without removing any process from memory. If memory is available, the swap-in procedure is begun (see below) and the algorithm is repeated (1).

4. If not enough memory is available, some process must be removed from core. The following criteria are applied:

a. All of the processes in the Process Table are examined and the first process that is in the WAIT state ("p_pri" = SWAIT), i.e., roadblocked at low priority), or the first process that is in the stopped state ("p_stat" = SSTOP), this is a new state for processes that are being examinedii_by the interactive C debugger) is selected to be swapped out.

b. If the process selected in 1 as a candidate to be brought into memory has been on the swap device for less than three seconds, the entire procedure is dropped and the Scheduler roadblocks after setting the "runin" flag. This essentially means that the Scheduler waits for core to willingly become available or for the age of processes to change.

c. If the process selected in 1 has been on the swap area for longer than three seconds, then all of the Process Table entries are examined again and the oldest process in memory that is in the ready state ("p_stat" = SRUN) or is in the sleep state ("p_stat" = SSLEEP) is found. No distinction is made between the two. If the process selected by this criteria has been in memory for more than 2 seconds, it is swapped out, otherwise the procedure is dropped and the Scheduler roadblocks ("runin" is set) until core is available or the age of processes change.

Swapping a process out is simply done by marking it as non-resident (SLOAD flag is reset in "p_flags") and arranging for it to be swapped (text.c/xswap).

Swapping a process in may require two swaps if the process is reentrant and the reentrant text is not already in memory. Since reentrant text is maintained on the swap area as a separate quantity, it need never be swapped out. By checking the "p_textp" entry in the Process Table, it can be determined whether reentrant text is associated with a process and it's location and size can also be obtained ("x_caddr", "x_size"). The reentrant text (if it is not already in core), is brought in first and the memory usage count ("u_ccount") for that text is incremented. The remainder of the process (data, bss and stack) is then brought in. When this portion is brought into memory, the space that it occupied on the swap device is freed (alloc.c/mfree). Any reentrant text is not freed as it cannot be modified and leaving it there will save the trouble of swapping it out. When the last process using the reentrant text leaves the system, the text is destroyed (by text.c/xfree).

The Process Table entry for the swapped in process must be updated to reflect the changed status of the process. In particular, the swapped in process must be marked as loaded (SLOAD in "p_flag", its age ("p_time") must be reset to zero and its address in physical memory ("p_addr") must be set.

If any errors occur in attempting to swap a process in or out, the system is halted. The device drivers usually make 10 retries when an error occurs, so that any swap I/O that terminates abnormally for the Scheduler is assumed to be an unrecoverable hardware error.

The Scheduler, having finished swapping a process in or out, cycles back to begin its search (1) for processes to bring into memory. The age requirements placed on processes that can be swapped in or out prevent the Scheduler from thrashing.

In conjunction with the criteria established for swapping a process in or out of memory, the Scheduler is always notified when an in core process goes into the WAIT state . This is done only when the Scheduler cannot find enough memory to bring a process on the swap device into memory. The notification results in the Scheduler being the next process (the Scheduler always has the highest priority) to run and the roadblocked process will probably be swapped out. When searching the Process Table it is necessary to raise the Processor's priority to 6 to prevent the status of the processes being examined from changing.

setrun

CALL

setrun(p)
struct proc *p;

RETURNS

No value is returned.

SYNOPSIS

Awakens one process.

DESCRIPTION

This places the process "p" in the ready state ("p_stat" = SRUN). In addition, it notifies the Scheduler if the process is not in core and the Scheduler is waiting, for processes on the swap area to become ready. The external varialbie "runout" is set when the Scheduler is in that state.

sleep

CALL

sleep(event, priority)

RETURNS

No value is returned.

SYNOPSIS

Roadblocks a process.

DESCRIPTION

This is the complementary routine to slp.c/wakeup. It synchronizes the roadblocking and unblocking of a process. When processes under UNIX relinquish the processor, they post (in "p_wchan") an event whose occurrence they are awaiting. This "event" is some mutually agreed upon value (usually an address within the system) which is used to signify the occurrence of an event. A process also specifies the software priority it is to be assigned when the "event" occurs. Priorities are values (kept in "p_pri") which range from a high of -100 to a low of +127, and are used by the process Switcher in- selecting a process to use the CPU. A process which relinquishes the CPU at high priority (negative priority) is put in the sleep state ("p_stat" = SSLEEP), while processes that relinquish the CPU at low priority (positive priority) are put in the wait state ("p_stat" = SWAIT). Processes which roadblock at high priority (negative) are guaranteed to be awakened in a short time (usually determined by device speed and device

queuing), while those that roadblock at a low priority are not. As such, roadblocking is handled differently for the two cases of priorities.

For processes that roadblock at high priority, no precautions need be taken, so that the process Switcher (slp.c/swtch) is simply called. Before doing this, however, the processor's priority is lowered to zero (the processor's priority is saved first), so that all pending interrupts may be processed. This is done because any pending inter-. rupt may awaken a roadblocked process, thereby producing another candidate for the process Switcher. It is assumed that the process will be blocked for only a short period of time so that signal processing is delayed until the process is awakened and the system call that the awakened process was making is completed. (i.e., the trap handler checks for the presence of a signal).

Processes that roadblock at tow priority may be roadblocked for a long period of time (e.g., wait-. ing for a child process to terminate, etc.), so that it is necessary to check (sig.c/issig) whether any signals are pending for the process before and after it is roadblocked. If a signal is pending and is to be handled by the user process, then a non-local goto (mch.s/aretu) is executed to abort the system call and return control to the trap handler. As with the high priority case, the processor's priority is lowered before the process Switcher is called to process all pending interrupts.

swtch

CALL

swtch()

RETURNS

A one is returned.

SYNOPSIS

The process Switcher.

DESCRIPTION

Selection of a process to use the CPU is a distinct operation from that of selecting a process to be brought into memory (slp.c/sched). The Scheduling process uses the age ("p_time") of a process in determining whether a process should be swapped in or out of memory. The process Switcher uses the priority ("p_pri") of a process as its sole basis for selection. Of course, it can only select from among processes that are already in core and are ready to execute.

Since changing execution from one process to another entails changing U blocks, and thereby changing system stacks, a problem arises in making the transition from one U block to another. The dilemma Occurs because there may occur a situation in which only one process can fit in core. There would then be no place for the system's stack in the interim that when no user process was in memory. Also, if the U block of the current process is retained, even though there may be no other processes to run, all of the system time during the idle period would be accrued to that process unless special arrangements were made. Conceptually, it is also nicer to totally disassociate the system from a process which is relinquishing the CPU. To this end, when the slp.c/swtch function is executed, the Scheduler's U block is grabbed and used until a new process is found. If none are available, the processor is placed in the wait state (by the WAIT instruction) until a process goes into the ready state. In this manner, all processor idle time is accrued to the Scheduler.

The process Switcher must be made unbiased in the sense that it must not give preference to one of several processes with the same priority ("p_pri") due to any positional advantage in the Process Table. To do this, the identity of the process selected by the process Switcher is remembered and the search for a new process to execute is started at this entry in the Process Table. In this way, Round Robin service is given to processes with the same priority.

The steps in the selection of a ready process are:

1. The context of the process relinquishing the CPU is saved (mch.s/savu) in the appropriate place in the U block "u_rsav".

2. The U block belonging to the Scheduler is obtained (mch.s/retu) for use as the system stack. As the Scheduler is simply another process to the process Switcher and as the process Switcher must be executed before the Scheduler runs, the Scheduler's stack is cleared off and is not affected by the Switcher using it.

3. The Process Table is examined to find a process in memory ("p_flag" = SLOAD) with the highest priority that is ready ("p_stat" = SRUN). The search begins at the entry in the Process Table succeeding that of the currently running process and continues (wrapping around the Process Table) until the entry for the currently running process is reached. The highest priority process (lowest value of "p_pri") is selected.

4. If there are no ready processes, then the processor is placed in the wait state (by calling mch.s/idle). The processor will remain in this state until the first interrupt occurs. Since interrupts usually awaken some roadblocked process, there is a good chance that a process will become ready after the interrupt is processed. When a return from the mch.s/idle function occurs, the search is reinitiated and begins with the same Process Table entry as before.

5. Once a process has been selected, it's U block is obtained (mch.s/retu) for the system stack.

6. The User Memory Management registers for the selected process are loaded from the prototype registers ("u_uisan", "u_uisdn").

7. In most cases the process being restarted was roadblocked by calling slp.c/sleep, which caused the process's context to be saved (see I above). To restart these processes, the context is restored from "u_rsav" and a return from slp.c/swtch restarts it. Sonic processes, however, relinquish the processor in order to swap themselves out (slp.c/expand), or to create a new process (sys.c/fork). The functions that do this usually contain two algorithms, one for use when there is enough available memory to perform the operation and the second for use when the operation must be done on the swap device. These processes need to regain control at a point other than that specified by "u_rsav" so that a nonlocal goto (mch.s/aretu) is done if the swap flag (SSWAP in "p_flag") is set. The return executed by slp.c/swtch will not return to the caller of slp.c/swtch, but will return from the function that last saved the context in the array "u_ssav". This is extremely useful,in doing in memory expansions of a process and in spawning and restarting new processes (sys1.c/fork).

A one is returned by the process Switcher to distinguish parent and child processes (see slp.c/newproc)). The sys1.c/fork function will thus have a one returned from the child process and zero returned by the parent (from slp.c/newproc) and will use this in resetting accumulated execution times for the child. The main.c/main function also uses this returned value in the process of spawning the INIT process.

wakeup

CALL

wakeup(event)

RETURNS

No value is returned.

SYNOPSIS

Awakens roadblocked processes.

DESCRIPTION

This is the complementary function to slp.c/sleep. It places processes that are roadblocked waiting for "event" to occur in the ready state ("p_state" SRUN). A simple scan of the Process Table checking the "p_wchan" entry versus "event" shows which processes should be awakened. It is the responsibility of the awakened process to insure that the event has actually occurred. Slp.c/setrun is called to place an individual process in the ready state ("p_pri" = SRUN). If any process is awakened, the "runrun" flag is set. This is a flag checked by the interrupt and trap handlers. Setting this flag allows the currently executing process to be preempted once the interrupt or system call is completed. Since the operating system is not reentrant, this scheme is necessary so that preemption may be done. Preemption as the result of a wakeup (usually a wakeup is sent by an interrupt handler) is only allowed at the end of an interrupt if the interrupt occurred while the processor was in User mode. Preemption is also allowed at the end of a system call.

bcopy

CALL

bcopy(from, to, count)
int *from, *to, count;

RETURNS

No value returned.

SYNOPSIS

Copies a specified number of words from one location to another.

DESCRIPTION

Bcopy copies "count" words from memory location "from" to memory location "to". Both "from" and to must be on word boundaries.

bmap

CALL

bmap(ip, blk)
struct node *ip;
int blk;

RETURNS

Block number on a device. Zero on an error.

SYNOPSIS

Converts a logical block number of an ordinary file or directory into the block number on a device.

DESCRIPTION

Brnap is used to convert, for other than special files, a file's logical block number "blk" into a block number on the logical device containing the file. This function is not performed for special files, since the logical block number is always construed to be the physical block number. The file is identified to bmap by a pointer "ip" to its system inode table entry (inode[]). The manner in which the file logical block number to device logical block number mapping is done depends on whether the file is small (eight blocks or less) or large (between nine and 2048 blocks).

For small files, the logical block number is used as an offset within the)node table entry (from i_addr) to find the device block number.

However, for large files, the logical block number must be divided by 256, which is the number of block pointers in each indirect block of a large file. The quotient of this division is used as an offset within the)node table entry (from i_addr) to find the device block number of an indirect block, which is read. The remainder from the division is then used as an offset within that indirect block to obtain the device block number.

Regardless of whether the file is small or large, if there is no device block allocated for the logical block number in question, a disk block is automatically allocated and added to the file as that logical block number. In the case of small files, this may also entail conversion of the file to a large file.

A return of zero indicates an error, the appropriate error bits are set in u_error. Otherwise, the device block number is returned.

cpass

CALL

cpass()

RETURNS

A single character from an I/O buffer; -1 on error or the completion of a write request.

SYNOPSIS

During processing of a write system call, obtains a single character from the requester's I/O buffer.

DESCRIPTION

During processing of a write system call, cpass is used to obtain a single character from the requester's I/O buffer (which is pointed to by u_base). The method used to extract this character depends on whether the buffer is in system or user space (as determined by u_segflag). After getting the character, the file offset (u_offset) and address of the I/O buffer (u base) are increased by one, and the byte count for the write system call (u count) is decreased by one. The character obtained is then returned to the caller. A -1 return value indicates that the write request byte count has been satisfied (on the previous invocation of cpass) or that an error occurred. In the latter case, an error bit is set in u_error.

nodev

CALL

nodev()

RETURNS

No value returned.

SYNOPSIS

A dummy routine.

DESCRIPTION

Nodev does nothing but return after setting an error bit in u_error. It is most commonly used in the configuration table conf.c as a placeholder for a device driver routine that it is erroneous to call (e.g. the read routine for the line printer).

nulldev

CALL

nulldev()

RETURNS

Always zero.

SYNOPSIS

A dummy routine.

DESCRIPTION

Nulldev does nothing but return. It is most commonly used in the configuration table conf.c as a placeholder for a device driver routine that should be ignored (e.g. an open routine for a device that does not require any special open processing).

passc

CALL

passc (char)
int char;

RETURNS

Zero when the character has been passed; a -1 on error or the completion of a read request.

SYNOPSIS

Places a single character into the requester's I/O buffer during processing of a read system call.

DESCRIPTION

During processing of a read system call, passc is used to move a single character "char" from the system I/O area to the requester's I/O buffer (u_base). In performing this duty, passc must know whether the requester's buffer is in system or user space (as determined by u_segflg). After moving the character, the file offset (u_offset) and address of the I/O buffer (u_base) are increased by one and the number of bytes requested by the read system call (u_count) is decreased by one. A return value of zero indicates that the character was successfully passed. A -1 return indicates that the read system call byte count was satisfied by the character just passed or that an error occurred passing the character. In the latter case, an error bit is set in u_error.

break

CALL

break()

RETURNS

No value is returned.

SYNOPSIS

Sets the program break. Break system call.

DESCRIPTION

The break system call sets the program break which is the highest address in a program. It is used for dynamic storage allocation (alloc and free system calls).

The program break may be set to increase or decrease the existing size of a program. The single argument that is passed (indirectly in "u_arg[0]") is the address of the new break, not an increment to be added or subtracted from current break. Sys1.c/sbreak uses this absolute value to calculate the difference between the old size and new size. New space is added or subtracted from the data area ("u_dsize"). No area can be added to reen trant text ("u_tsize) or the stack ("u_ssize") area. To insure that the program break can be set to the new value, main.c/estabur is called to test fit the new virtual address space and load the User Memory Management registers.

If the new program break decreases the existing size, then the user's stack is moved down in the program's physical area, so that it is in the proper position for the, virtual address map. The slp.c/expand function is then used to release the extra area. For break calls that increase the size of the program, a larger physical space must- be acquired before any internal adjustments are made. Slp.c/expand is used to increase the physical space occupied by a process. It may roadblock the break call if there is not enough memory available. (see slp.c/expand). When the physical area is finally enlarged, sys1.c/sbreak moves the stack downward to the proper position for the virtual address map and the new data area is initialized to zero.

exec

CALL

exec()

RETURNS

No value is returned.

SYNOPSIS

Overlay system call.

DESCRIPTION

This function performs the exec (overlay) system call. Overlaying is a multi-step procedure in the system fraught with difficulties. The chief hazard is that since the exec system call requires two buffers, one to hold system arguments and one to read the program in, a deadly embrace is possible. The risk of this is currently reduced by restricting the number of simultaneous exec's that can be going on at one time.

The exec system call passes the name of the program to be executed along with any arguments to that program. It is the responsibility of sys1.c/exec to determine whether the program is reentrant-or not and to pass these arguments to the program. In line with the hierarchy of processes under UNIX, overaying programs inherit certain attributes of the antecedent process. The chief attributes inherited are:

1. The working directory is inherited.
2. All open files of the preceding process are inherited.
3. The age and process id are inherited. In short, all Process Table information is inherited.
4. All per process information (U block) with the exception of signals and context information are inherited. Ignored signals are inherited but user handled signals are not.

The steps in doing an overlay are:

1. The system determines what program is to be executed. The first argument to the exec system call is the name of the overlaying program. The i-node for the file containing the program is brought into the Inode Table (nami.c/nami). If no such named program exists, the exec is terminated and an error is posted by nami.c/nami is returned to the user.

2. In order to forestall the occurrence of a deadly embrace, the Count of the number of processes currently doing an exec "execnt") is checked to see if it has reached the limit NEXEC (4). If it has, the process calling for the overlay is roadblocked until the count drops below NEXEC. (The count is decremerited as each process finishes the exec and a wakeup is issued if any process is roadblocked awaiting use of the exec function.) This procedure does not absolutely prevent the occurrence of a deadly embrace, but does decrease its probability significantly.

3. Once entrance to the exec function is permitted, a system buffer is allocated (bio.c/getblk) to hold the remaining arguments from the exec call. The buffer is allocated to the "bfreelise queue ("b_forw", "b_back" list), so that no device association occurs. The arguments are then fetched from the user's address space (using mch.s/fubyte). (The exec system call passes an gray of arguments. The array contains pointers to the actual arguments. The arguments themselves must be ASCII strings terminated by nulls, octal 0, or must appear to be strings terminated by nulls.) The pointers to the actual arguments are fetched and eventually discarded. Only the arguments themselves, separated by null characters, are placed in the argument buffer. Only 511 characters of argument strings and their (null) separators are allowed. If this limit is exceeded, the overlay is terminated and the system error E2BIG is posted in "u_error". The arguments are taken from the program in the same order that they appear in the system call.

4. The first 8 bytes of the program to be executed is read into the argument array ("u_arg[]") in the U block. This is done in a manner similar to that in which a core image of a process is produced (sig.c/core). The variables "u_count", "u_offset[]", "u_segflg" and "u_base" are set to the number of bytes for the transfer, the offset into the file (0), whether the transfer is into the system (1) and the virtual destination address ("u_uarg[0]") in the system before calling rdwri.c/readi. If any system errors occur in doing this read, the exec is terminated. The header (first 16 bytes) of every object file contains the following information:

   a. Majic number. This indicates whether the program is reentrant (410) or not (407) and in the future will indicate whether the program is to be separated into I and D space (411).

   b. Text size in bytes.

   c. Data size in bytes.

   d. Bss size in bytes.

   e. The remaining 4 words contain information about the symbol table and relocation bits (see UNIX Programmers Manual, A.OUT(V)).

   These arguments are checked and regrouped. Any file not beginning with one of the majic numbers is assumed to be unexecutable and the exec is terminated. For nonreentr ant programs, the text and data area size are combined in what is called the data. This insures that they will not be separated when the prototype Memory Management registers are set up (main.c/estabur).

5. The prototype Memory Management registers are set up after rounding the text and data sizes up to the nearest memory block (64 bytes). The initialstack size (SSLZE) is set up for 20 memory blocks (1280 bytes). If the program is too big or there is physically nor enough user memory on the machine, the exec call will be terminated and the system error ENOMEM will be posted by main.c/estabur. A check is also made to insure that no program is updating the file which is to be overlaid. Rather than wait for the update to occur, the exec is terminated.

At this point, the program has satisfied all of the criteria for being allowed into the system. The procedure for bringing the program into memory is as follows:

1. Any text associated with the old process is freed (text.c/xfree).

2. The remainder of the old process is truncated to the size of the U block, (slp.c/expand).

3. Any reentrant portion of the program is read in and a copy placed on the swap device if there is not already a copy in the system (text.c/xalloc). This may result in the exec call being delayed while necessary I/O occurs.

4. The U block (truncated from the original process) is expanded to the size of the data and stack. (The data size consists of data and

bss for reentrant programs and includes the size of the text for nonreentrant programs.) The allocated memory is cleared. Because of the way memory expansions are done, a swap may occur at this point to grow the process size.

5. The user's Memory Management registers are set up so that there is only a data area (no stack), and the data (and bss) portion of the program is read in. The read is accomplished by setting "u_base", "u_offsetn", "u_count" and "u_segflg" to indicate that the read is to be done into the user's virtual address space (starting at virtual 0), is to begin in the object file after the 16 byte header, is to include all data (and text for nonreentrant programs) and is to be done into the user's virtual address space.
6. The correct virtual address space is set up for the program (main.c/estabur) , the User Memory Management registers are loaded and the size of the process is recorded (in "p_size"). When setting the correct virtual address space, the reentrant text is included and an initial stack area (SSIZE) of 1280 bytes is used.
7. The arguments to the overlaid program are set up. This is done by storing the (ASCII string) arguments onto the user's stack. The argument strings are placed on the stack in the same order that they were taken from the calling program. Pointers to each string are loaded directly below this.
8. There are 2 indicators in the file access permissions associated with each file. These two indicators allow the user id or group id of a program to assume those associated with the i-node rather than those of the user executing the program. These two indicators are referred to as the "set-user-id" and "set-group-id" bits and sys1.c/exec changes the user or group id to the appropriate value if they are set.
9. As mentioned previously, only ignored signals are inherited by the orverlaid processes. User processed signals are reset so that to the standard system action is taken.
10. The user's registers must be zeroed (including floating point registers), so that the overlaid process starts out with all registers initialized to zero. Since the system and user processes use the same set of registers (General Register Set 0), the registers cannot be zeroed directly. Rather the context information saved when the exec system call is made is zeroed. The PC on the stack is set to zero so that when the overlaid program begins executing it will start executing at virtival address zero.
11. The i-node for the overlaying file can be released and the buffer used for holding arguments can be released (bio.c/brelse) to the pool of available buffers.
12. Any processes that were roadblocked because they attempted an exec and there were too many processes in the midst of doing an exec are awakened. The number of processes doing an exec ("execnt") is decremented.

exit

CALL

exit()

RETURNS

No value is returned.

SYNOPSIS

Terminates a process, that is, makes a process a ZOMBIE.

DESCRIPTION

When a process terminates, it enters the zombie state ("p_star" = SZOMB) until a parent find it. Besides the need to pass back termination status of children processes, the cpu time (user and system) used by the child is accumulated by the parent. As CPU times are kept in the U block and as a parent may have many children processes, it is convenient to keep the U block of the deceased process around until the parent disposes of it.

Since the parent process may disappear from the system before any of its children, a mechanism must exist so that the children can be disposed of. The INIT process in the system is the pro cess that spawns the line monitor programs. It allows user's to log on and off so it is always in the system. (Since it is a user process though, it can be killed.) If a parent dies before any of it's children, then INIT is made their parent and will dispose of them when they terminate.

The steps in the termination of a process are:

1. All of the process's signals are reset so that they are ignored. This is done since subsequent steps may require the process to

roadblock, at which time a signal might be caught.

2. All of the files that the process had open are closed.
3. The i-node corresponding to the working directory is released from the Inode Table.
4. Any reentrant text is abandoned (text.c/xfree).
5. Space is allocated on the swap device to place the zombie. The zombie is only 256 bytes in size, but 8 times that much space is allocated to reduce fragmentation. This is done because although most processes dispose of zombies quickly (by waiting for them), a zombie remains in the system until a parent finds it. Overallocating space on the swap device reduces fragmentation (hopefully).
6. A system buffer is obtained to copy the first 512 bytes of the U block into. Since the relevant information in the U block is in the first 512 bytes ("u" array), buffered I/O can be used rather than doing physical I/O to the swap device. A synchronous buffered write is performed to insure that the data reaches the swap device and does not linger in the I/O subsystem.
7. The memory occupied by the process is freed and the process state is changed to that of a zombie ("p_stat" = SZOMB), and the address of the process on the swap device is set up ("p_addr"). The data and stack areas could have been freed (malloc.c/mfree) before the copy was made in 6, but the U block would then have to be freed and fragmentation in memory would probably be increased.
8. Arrangements are made for the disposition of the terminating process and all of it's children.
   a. The Process Table is searched for the parent of the terminating process. If the parent is found, INIT is awakened first_This is done because INIT will inherit all of the terminating process's children as its children. As some of these children may already be deceased, INIT can dispose of them. A wakeup (slp.c/wakeup) is also issued to the parent process. The. Process Table is rescanned and all of the children of the terminating process are made children of INIT (process number 1). Finally, the processor is relinquished by the terminated process.
   b. If the parent of the terminating process cannot be found, then the terminating process is made a child of-INIT and the procedure in a is repeated.

      If the INIT is somehow destroyed in the Process Table, a system panic occurs ("PANIC NO INIT PROCESS"). INIT may be killed and become a zombie itself, however, since there will no longer be a process to remove zombies as described above they will accumulate in the system.

fork

CALL

fork()

RETURNS

Posts an error if there is not room in the system to create a new process. Also, returns the identity of the created child process to the parent and the identity of the parent to the child.

SYNOPSIS

Fork system call. Creates a new process in the system.

DESCRIPTION

Fork is the mechanism by which a new process enters the system. It creates a mirror image copy of the process making the fork call.

Most of the work in creating the mirror image process is done by slp.c/newproc. This function creates the new image and sets up a new Process Table entry for the child. After it completes it's work, there will be two processes in the system which will return from the slp.c/newproc function. The parent will return directly from slp.c/newproc, returning a zero, however, the child will return through the process Switcher (slp.c/swtch) and will return a one. Sys1.c/fork uses this distinction to allow it to perform some additional initialization for the child. In particular, it zeroes the cumulative user and system times ("u_utime" and "u_stime") of the child and the cumulative user and system time of the child's children ("u_cutime[]" and "u_cstime[]"). In addition, it returns the ID of the parent to the child. (The actual C library interface for the fork system

call causes a zero to be returned to the child.) For the parent process, sys1.c/fork returns the ID of the child and advances the PC for the parent process (on the stack frame), so that a different point in the C library is entered.

Before starting the creation of the child, sys1.c/fork scans the Process table to see if there is a slot available. If there is none, then an error EAGAIN is posted.

rexit

CALL

rexit()

RETURNS

No value is returned.

SYNOPSIS

The exit system call.

DESCRIPTION

This is the System Entry point corresponding to the exit system call. The exit system call can pass a one byte status indicator as an argument. Sys1.c/rexit saves this value (passed in R0) in "u_arg[0]" for convenience and calls sys1.c/exit to terminate the process.

wait

CALL

wait 0

RETURNS

A system error is posted if a process waits, but has no children.

SYNOPSIS

The wait system call. Wait for a child process to die.

DESCRIPTION

When a process dies, it becomes a zombie until the parent finds and disposes of it. If a parent does not wait for the child, then the zombie will remain in the system for the lifetime of the parent. If the parent leaves the system before the child dies, then the child is made a child of the INIT process.

Making a process a zombie allows status and execution time of the child to be examined. In particular, the exit status of the deceased child is passed back to the parent and the cumulative execution time (user and system time) of the deceased child and all of it's children is added to that of the parent.

The wait system call does not wait for the death of a particular child process. Rather, the call returns when the first child process terminates.

When sys1.c/wait is called, a linear search of the Process Table is done. If the process making the wait system call has no children then a system error (ECHILD) is posted (in "u_error") and the wait is terminated. If the process does have children, but none are yet deceased, the process is roadblocked (at low priority, "p_pri" = PWAIT).

when a child does ,terminate the roadblocked parent is reawakened and the linear search is repeated.

When a zombie child is found, the following steps are performed to dispose of it.

1. The process ID of the deceased child is returned in register R0 and the status of the dead child is returned in register R1. The status consists of two bytes. The low order byte contains the number of any signal that may have been received by the process to cause termination and an indication of whether a core image was produced. The high order byte contains any status inforrnation returned by the child.
2. The U block of the zombie is read into one of the system buffers by doing a synchronous buffered read (bio.c/bread) from the swap device and the area occupied by the zombie on the swap area is freed. It should be remembered that this area was overallocated (8 disk blocks rather than one) to reduce fragmentation on the swap area.
3. The Process Table entry of the deceased child is cleared. Only the important entries are zeroed; "p_stat" - process state; "p_pid" - process ID; "pppid" - parent ID; "p_sig" - indication of pending signal; "p_ttyp" - controlling teletype; "p_flag" - location of process and flag indicators for process.
4. The execution of the deceased child and all of it's children is added to that of the parent

Associated with every process there are several time values kept.

1. "u_utime" - This is a one word entry which records the cumulative user CPU time of the process. (that is, time actually spent executing the user's program). The entry is kept in

sixtieths of a second and is only an approximation since it cannot discount any interrupt handling processed between clock ticks (1/60 second).

2. "u_stime" - This one word entry records the cumulative system CPU time used by the process. It records all of the time spent by the system in handling system calls for the process and is subject to the same limitations as 1.
3. "ti_cutimen" - This is a two word entry (long integer) used to record the cumulative user time of all children of the process.
4. "u_cstimen" - This is a two word entry used to record the cumulative system time of all children processes.

When a process terminates, the (user and system) execution time of the deceased are added to the cumulative times ("u_cutime[]" and "u_cstime[]") of the parent. That is, the parent's times are adjusted as follows:

"u_cutime[]" = "u_utime"+"u_cutime[]"

"u_cstime[]" = "u_stime"+"u_cstime"

In both of these equations, values on the left correspond to parent times, while all values on the right correspond to child times.

The system buffer containing the zombie U block is released to the buffer pool (bio.c/brelse).

close

CALL

close()

RETURNS

None

SYNOPSIS

Close system call interface - close a file.

DESCRIPTION

The argument to close is the file descriptor of the file to be closed. Close calls fio.c/getf to check that the file descriptor is a valid value and points to a open file. Close then zeros the file descriptor entry in the per user control block (u.u_ofile). Close calls fio.c/closef to decrement the usage count in the System INODE Table (inode.h) to see if the entry can be purged (if the i-node is not shared by other processes). If the. count goes to zero, the i-node is updated in the ilist and the entry is freed.

creat

CALL

creat()

RETURNS

On error the appropriate code has been set in the per user control block (u.u_error). On success the file descriptor for this file is returned to the user in register R0.

SYNOPSIS

Creat system call interface - to creat new files.

DESCRIPTION

Like OPEN, creat takes a string of characters representing a pathname and calls nami.c/namei to decode the pathname into the appropriate locked and incremented i-node. If the filename does not exist (creating a new file) then iget.c/maknode is called to allocate a i-number and free i-node, put the i-node into the System INODE Table (i-node.h) and write a directory entry.

Sys2.c/open1 is called to complete linkage from the System File Table (file.h) and the user’s File Descriptor Table (u.u_ofile).

link

CALL

link()

RETURNS

If successful none. An error occurs if the file already sexists (EEXIST) or a link to a file on another device is requested (EXDEN). The error code is set in the per user control block (u.u_error).

SYNOPSIS

Link system call interface - a link is a pointer in a directory to a file.

DESCRIPTION

Link is called with two arguments: the pathnarne of the file to be linked to, and the name to call the link. Nami.c/namei is called with the first argument to translate the pathname into a i-node pointer. The returned pointer to a incremented, locked i-node. Links to directories are illegal (except by the super-user) so if the file is a directory and the user is not super-user, an error flag is set, iget.c/iput is called to unlock this i-node and decrement the usage count by one.

The i-node is unlocked so that nami.c/namei can be called again with the second argument to return a locked, incremented i-node for the second argument pathname. If the file already exists the error code EEXIST is set in the per user control block (u.u_error) and iget.c/iput is called to unlock the i-node and decrement the use count. Links cannot exist across devices because identical i-numbers appear in separate file systems and the link is only an i-number reference. If the directory i-node of the second argument is not on the same device as the first argument file then the error code EXDEV is set in the per user control block (u.u_error). Otherwise, iget.c/wdir is called to write a directory entry from the information left behind by the call to narnei. The link count of the i-node is incremented by one and the i-node is marked so that the "last modified" date and time can be updated. Iget.c/iput is called to unlock the i-node, and reduce the use count by one.

mknod

CALL

mknod()

RETURNS

On success none. On failure, if the name already exists sets the error code EEXIST in the per user controi block (u.u_error).

SYNOPSIS

Mknod system call interface - make a directory (called from mkdir) or a special file (/etc/rnknod).

DESCRIPTION

Mknod is called with three user supplied arguments: a pointer to the pathname of the file, the mode of the new file, the major and minor device classes for special file (zero for directory).

Mknod can only be called by the super-user for directory creation. Nami.c/namei is called to decode the pathname, create a i-node entry in the System INODE Table (i-node.h), and return a pointer to the locked, incremented i-node. If the pathname does not exist the error code (EEXIST) is set- in the per user control block. iget.c/maknode is called to fill in the new i-node with the "mode" from the second argument. Maknod also calls iget.c/wdir to write the directory entry for this i-node. The first address field in the i-node, addr[0], is set to the third argument.

open

CALL

open()

RETURNS

On error the appropriate code has been set in per user control block (u.u_error). On success the file descriptor for this file is returned to the user in register R0.

SYNOPSIS

Open system call interface - to open existing files.

DESCRIPTION

Open takes a string of characters representing a pathname and on successful return has brought in the i-node for that file into the System INODE Table (i-node.h), built an entry in the System File Table ffile.h), and has built a file descriptor entry in the File Descriptor Table in the per user control block (u.u_ofile). Open calls nami.c/namei to decode the pathname, find the i-node of the associated file, and is returned a pointer to the appropriate i-node in the System

INODE Table (i-node.h). The pointer is to a locked, incremented i-node. Sys2.c/openl is

called to complete the Ihikage from the System File Table (file.h) and the user's File Descriptor Table (u.u_ofile).

openl

CALL

openl (ip, node, trf)
int *ip, node, trf;

RETURNS

If successful return nothing. If permissions are illegal set the error code in the per user control block (u.u_error).

SYNOPSIS

Complete linkage of control blocks tq open a file.

DESCRIPTION

Openl checks the requested "mode" against the file access permission by calling fio.c/access with the pointer, "ip", to the i-node in the System INODE Table (i-node.h). If, the access is illegal the error code is set in the per user control block (u.u_error) and iput.c/iput is called to decrement the usage count in the System INODE Table and unlock the i-node. Anytime i-nodes must be changed they are locked to prevent simultaneous change by several processes.

If openl was called by CREAT (sys2.c/creat), the file is new or to be re-written. The "trf" flag is set to indicate this and openl calls iget.c/itrunc to truncate the file. Pipe.c/prele is called to unlock this i-node.

This file may be a special file (a physical device) and fio.c/openi is call to make this check and open the device if necessary. Fio.c/falloc is called to find the next available file descriptor in the user's File Descriptor Table (u.u_ofile), and the first available slot in the System File Table (file.h). The System File Table is then set to point to the allocated i-node in the System INODE Table (inode.h).

rdwr

CALL

rdwr (mode)
int mode;

RETURNS

On an error, rdwr will set the appropriate error code in the per user control block (u.u_error). On success the count of the number of bytes read/written is returned is user register R0.

SYNOPSIS

Read or write data from a file pointed to by a file descriptor.

DESCRIPTION

Arguments to the read or write system call are the file descriptor, the address of a buffer, and the number of bytes to read or write. The file descriptor is in the user's register R0 and the buffer address and number of bytes are in the per user control block (u.u_arg[0], u.u_arg[1]). The file descriptor is connected to a pointer to the System File Table entry (file.h) for this open file by fio.c/getf. If the file descriptor is invalid or the mode of the file does not match the mode input argument then the per user control block error code is set (u.u_error) to bad file description (EBADF).

If the System File Table entry indicates that this file is a pipe then the read or write is accomplished by pipe.c/readp or pipe.c/writep. Otherwise the read or write is done by ordinary file I/O routines rdwri.c/readi or rdwri.c/writei. In either case, the number of bytes read written is placed in the per user control block (u.u_count) and returned to the user in R0.

read

CALL

read()

RETURNS

None

SYNOPSIS

Read system call interface.

DESCRIPTION

Read call the sys2.c/rdwr routine with a read mode. Rdwr does all the work.

seek

CALL

seek()

RETURNS

If successful none. If failure the error code is set in the per user control block.

SYNOPSIS

Seek system call interface - moves the read/write position pointer by blocks or bytes absolutely or relative to the current positions.

DESCRIPTION

Seek is called with three arguments: the file descriptor which is placed in user register R0, the amount of the offset (u.u_arg[0]) and a flag defining blocks or bytes, and abslute or relative (u.u_arg[1]). The flag is as follows:

	Bytes	Blocks
Absolute Position	0	3
Relative to Current Position	1	4
Relative to End of File	2	5

Seek calls fio.c/getf to check the validity of the file descriptor value and that it points to a open file. A pointer to the entry in the System File Table (file.h) is returned.

If seek was called on a file descriptor that points to a pipe the error code (ESPIPE) is set in the per user control block (u.u_error) because seeks are illegal on pipes. Seek uses the flag argument (u.u_arg[1]) to determine how to apply the offset to the read/write position pointer in the slot in the System File Table (file.h).

write

CALL

write()

RETURNS

None

SYNOPSIS

Write system call interface.

DESCRIPTION

Write calls sys2.c/rdwr routine with a write mode. Rdwr does all the work.

dup

CALL

dup()

RETURNS

System primitive.

SYNOPSIS

Duplicates an open file descriptor.

DESCRIPTION

Dup is the system primitive used to duplicate a user's open file descriptor. This is easily accomplished by placing another pointer to the system file table entry (file[]) for the open file in the user's open file table (u_ofile) and increasing the file's use count (f_count).

fstat

CALL

fstat()

RETURNS

System primitive.

SYNOPSIS

Allows the caller to obtain the vital statistics of an open file by using the file descriptor.

DESCRIPTION

Fstat is a system primitive that enables a user to obtain certain information about an open file by using the file descriptor. Therefore, the file's name (which could be that of a special file or a pipe) need not be known. The real work of obtaining this data is done by sys3.c/stat1, and fstat's principle duty is to translate the user's fstat arguments (file descriptor and buffer address) into a correct stat1 function call. A secondary responsibility of fstat before returning is to compute the correct file length if the file is a pipe, and replace stat1's view of file length with a more meaningful value. In particular, from the read end of a pipe, the file length returned by fstat is the number of unread bytes in the pipe; from the write end, the length is the number of bytes currently in the pipe, whether read or not.

getmdev

CALL

getmdev()

RETURNS

The device number of a block device.

SYNOPSIS

Used while mounting and unmounting file systems to determine the device number of a block special file.

DESCRIPTION

The routine getmdev is used internally by the system when mounting and dismounting file systems (sys3.c/smount and sys3.c/sumount, respectively) to translate the special file's pathname into a device number. It is expected that this name resides in user space. Certain checks are performed to ensure that the special file is that of a valid block device that is configured into the system. The device number is returned unless the device fails to pass the validation tests, in which case the appropriate error bits are set (u_error).

smount

CALL

smount()

RETURNS

System primitive.

SYNOPSIS

Mounts a file system on a specified directory or regular file.

DESCRIPTION

Smount is the syitem primitive that effects the mounting of a file system on an existing file or directory (which is called the mountpoint). The restrictions regarding this procedure are: the mount point may not be currently in use by the requester or another user, the mount point may not be a special file, and the file system (device) being mounted may not already be mounted. Failure to fulfill any one of these requirements results in the file system not being mounted. If these restrictions have been satisified, the mount is accomplished by placing the file system's superblock (struct filsys) in memory in a block I/O buffer, building an entry in the mount table

(mount[]) and marking the mount point i-node as a mount point.

stat

CALL

stat()

RETURNS

System primitive.

SYNOPSIS

Allows the caller to obtain the vital statistics of a file by using the file's name.

DESCRIPTION

Stat is a system primitive that enables a user to obtain information about a file whose name is known. Since sys3.c/stat1 does the real work in obtaining this data, stat's principle function is to translate the file's pathname into a meaningful stat1 function call.

stat1

CALL

stat1(ip, addr)
struct inode *ip;
int *addr;

RETURNS

No value returned.

SYNOPSIS

Returns a file's vital statistics to a user in response to a fstat or stat system call.

DESCRIPTION

Stat1 does most of the work involved in fulfilling fstat and stat system calls. To ensure that the most current information regarding the file (especially the access and update times) is available, the file's inode table entry, pointed to "ip", is first written to the file system. The file's inode is then read directly from the file system. The information requested by the fstat or stat call is transferred directly to the user supplied area starting at "addr".

sumount

CALL

sumount()

RETURNS

System primitive.

SYNOPSIS

Dismounts a mounted file system.

DESCRIPTION

Sumount is the system primitive that unmounts a file system previously mounted by sys3.c/smount. This can be done only if no files in the file system are being used (i.e., have no system inode table entries). A sync is performed to ensure that all in-core data regarding the file system is actually written to the device. The dismount is then completed by clearing the mount table entry (mount[]), freeing the buffer used for the superbiock, and removing the mount point stigma from the mount point file.

chdir

CALL

chdir()

RETURNS

System primitive.

SYNOPSIS

Changes a process's current working directory.

DESCRIPTION

Chdir is the system primitive that allows a process to change its working directory. The new working directory must be a bona fide directory for which the user has execute permission. The actual switching of directories is easily accomplished by altering the U block's current directo ry pointer (u_cdir) to the new directory's inode table entry.

chmod

CALL

chmod()

RETURNS

System primitive.

SYNOPSIS

Alters a file's mode (i.e., permissions).

DESCRIPTION

Chmod is a system primitive that allows the mode of a file (i.e., its permissions) to be restated. This may be done only by super-user or the owner of the file (as determined by the effective userid). The actual mode changing is completed by placing the new mode in the file's mode field (i_mode).

chown

CALL

chown()

RETURNS

System primitive.

SYNOPSIS

Changes the owner of a file.

DESCRIPTION

Chown is the system primitive that enables a user to change the ownership of a file. Only the super-user or the file's owner may change the ownership (that is, a file may only be given away, it cannot be taken). The userid of the new owner is duly recorded in the file's ownership field (Luid). An interesting side effect is that, unless done by super-user, changing file ownership negates the set userid aspect of the file's mode.

getgid

CALL

getgid()

RETURNS

System primitive.

SYNOPSIS

Enables a process to determine the real group id under which it is running.

DESCRIPTION

Getgid is a system primitive that returns tb the user the real (as distinguished from the effective) group id (u_rgid) under which the process is running.

getpid

CALL

getpid()

RETURNS

System primitive.

SYNOPSIS

Enables a process to determine its process id.

DESCRIPTION

Getpid is a systim primitive that returns to an invoking process its process id (p_pid).

getswit

CALL

getswit()

RETURNS

System primitive.

SYNOPSIS

Reads the console switches.

DESCRIPTION

Getswit is a system primitive that returns the contents of the console switches to the user.

getuid

CALL

getuid()

RETURNS

System primitive.

SYNOPSIS

Enables a process to determine the real userid under which it is running.

DESCRIPTION

Getuid is a system primitive that returns to a process the real (as distinguished from the effective) userid (u_ruid) under which it is running.

gtime

CALL

gtime()

RETURNS

System primitive.

SYNOPSIS

Returns time of day to user.

DESCRIPTION

Gtime is a system primitive that returns the time of day (time[]), in one second granularity, to the user.

kill

CALL

kill()

RETURNS

System primitive.

SYNOPSIS

Sends a specified signal to another process.

DESCRIPTION

Kill is the system primitive that enables one process to send a signal to another. Before the signal is sent, verification is made that the sending process is either the super-user or that both the sending and receiving processes have the same (effective) userid.

nice

CALL

nice()

RETURNS

System primitive.

SYNOPSIS

Allows a process to alter its priority.

DESCRIPTION

Nice is a system primitive that provides a process a limited amount of flexibility in altering its priority. For a normal user, the argument to nice must be a value between zero and twenty. This value (u_nice) is ultimately added to the process’s system determined priority value, effectively lowering its priority. Super-user is allowed to specify, nice values between -220 and 20; hence, a raising of priority, is possible.

profil

CALL

profil()

RETURNS

System primitive.

SYNOPSIS

Activates the execution profiling of a process.

DESCRIPTION

Profil is the system primitive that activates process execution profiling. To do so, it records, for later system use (see mch.s/incupc), the values (u_prof[]) used by the system in fulfilling the profiling request.

setgid

CALL

setgid

RETURNS

System primitive.

SYNOPSIS

Changes the effective, and possibly the real, group id of a user process.

DESCRIPTION

Setgid is a system primitive that enables a process to change the group id under which it is executing. A normal user may change only the effective group id (u_gid) of the process to whatever the real group id (u_rgid) already is; superuser may change both the real and effective group id to anything desired.

setuid

CALL

setuid()

RETURNS

System primitive.

SYNOPSIS

Changes the effective, and possibly the real, userid of a user process.

DESCRIPTION

Setuid is a system primitive that enables a process to change the userid under which it is executing. A normal user is only allowed to change the effective userid (u_uid) of the process to whatever the real userid (u_ruid) already is; super-user may change both the real and effective userid to anything desired.

smdate

CALL

smdate()

RETURNS

System primitive.

SYNOPSIS

Alters the last modified time for a file.

DESCRIPTION

Smdate is a system primitive that enables a user to change a file's last modified time (i_mtime) to any time desired. Only the super-user or the file's owner may do this. This is done immediately to the i-node entry in the file system and an interesting by-product is that it forces the incore inode table entry for the file to be written to the file system.

ssig

CALL

ssig()

RETURNS

System primitive.

SYNOPSIS

Permits a process to specify the action to be taken when it is sent a signal.

DESCRIPTION

Ssig is a system primitive that allows a process to specify which of three ways a particular signal sent to it should be handled: default system action, ignored, or control be given to a user specified function (catch the signal). This is accomplished by entering the necessary value (zero, an odd integer, or the address of the function, respectively) in the proper u_signal[] slot. The kill signal may not be caught or ignored, and if the signal in question has already been sent to the process, but has not yet been fielded by the system, it is thrown away.

stime

CALL

stime()

RETURNS

System primitive.

SYNOPSIS

Sets the time of day.

DESCRIPTION

Stime is a system primitive used to set the time of day (time[]). The caller must be super-user. A secondary function performed is to wakeup all processes sleeping on a time of day. This ensures that none oversleep.

sync

CALL

sync()

RETURNS

System primitive.

SYNOPSIS

Syncs all mounted file systems.

DESCRIPTION

Sync is a system primitive that syncs all mounted file systems. That is, it causes all incore information about the file systems to be actually written to their devices.

times

CALL

times()

RETURNS

System primitive.

SYNOPSIS

Provides execution time infoanation about a process and its children.

DESCRIPTION

Times is a system primitive that provides to a process the user and system CPU time used by itself and the cumulative user and system CPU time used by all of its terminated child processes. All of the times returned have 1/60 second granularity.

unlink

CALL

unlink()

RETURNS

System primitive.

SYNOPSIS

Removes a link to a file from a directory.

DESCRIPTION

Unlink is a system primitive that removes a directory entry to a file (unlinks it). Super-user is the only person permitted to unlink a directory. The process of unlinking involves zeroing the file's i-number in its entry in its parent directory and decreasing its link count (i_nlink) by one. The file is physically removed from the file system when its link count becomes zero and all processes using it have closed it. There is no verification that the user owns the file, but the user must have write permission for the parent directory.

lflags

CALL

lflags()

RETURNS

No value returned.

SYNOPSIS

System call interface for semaphores.

DESCRIPTION

Semaphores are currently implemented as lock/unlock conditions. Lflags is called with an action code in u_arg[0] and the semaphore number in u_arg[1]. The action code is one of the following:

1. Lock - If the semaphore is locked, wait until it becomes unlocked. When the semaphore becomes unlocked (or if originally unlocked) lock it and return.
2. Unlock - Clear the semaphore and wake up all processes waiting on it.
3. Tlock - If the semaphore is unlocked, lock it and in either case (locked or unlocked) return.

The process id of the locking process is saved in the semaphore to allow the system to know who has certain semaphores. If the semaphore value is out of range return the EINVAL error. If the action code is not one of the above return the ENOENT error.

sysent

CALL

RETURNS

SYNOPSIS

System Entry Point Table.

DESCRIPTION

This table is used by the trap handler (trap.c/trap) to call the appropriate system function when a system call is made. The table consists of two word entries. The first word of each entry is the number of arguments that are to be fetched from the user's program as arguments to the system function. This number does not correspond to the number of arguments to a system call as specified in the UNIX Programmers Manual since some arguments are passed in registers (see trap.c/trap). Individual descriptions of system functions should be consulted for the manner in which arguments are passed. The second word of each entry is the address of the function in the system that implements the system call. Currently, there are 64 entries in the table.

The TRAP instruction is executed when a system call is made (SYS pseudo-op in assembly language). This instruction contains an 8 bit field reserved for a trap number. UNIX uses the lower six bits of this number as an identifier for the type of system call and hence an index into the "sysent[]" table.

Unused system call entries have the function trap.c/nosys entered in the table. This function posts an error if a system call is made using that as the index. The first entry in the table (index 0) is filled in with the trap.c/nullsys function. A zero index in the trap instruction is recognized as an indirect system call. The trap.c/nullsys entry in the table insures that multiple levels of indirect system call cannot be made.

xalloc

CALL

xalloc(ip)
struct mode *ip;

RETURNS

No value is explicitly returned. It does, however, set a process's text pointer ("p_textp").

SYNOPSIS

Creates a reentrant text segment in the system or finds the text if it is already present.

DESCRIPTION

Text.c/alloc is a subroutine of sys1.c/exec. It is used to create the Text Table entry for a process that has reentrant text and to read the reentrant text into memory. The Text Table contains the following entries for each reentrant text segment.

"x_count" - This is a count of the number of processes using the text.

"x_ccount" - This is a count of the number of processes in memory that are using a text.

"x_caddr" - This is the memory address (in memory blocks) of the text. This entry is only valid if the memory usage count ("x_ccount") is nonzero.

"x_daddr" - This is the address (in 512 byte blocks) of the text on the swap area. Since reentrant text is managed separately on the swap area from the nonreentrant portion, it need never be swapped out. Thus, reentrant text remains in the same position on the swap device until the last process using it leaves the system at which time the text is destroyed.

"x_size" - This is the size in bytes of the text.

"x_iptr" - This is a pointer to the Inode Table entry containing the i-node from which the text is to be read.

In setting up a reentrant text segment in the system, the following steps are taken:

1. The Text Table is scanned to see whether the reentrant text from i-node "ip" has already been set up. If it has, then steps 2-6 are skipped. The text pointer ("p_textp") in the Process Table is set up and the use count "x_count" is incremented in this case. If the text is not already known to the system, a Text Table entry is allocated. If there are too many entries in the Text Table the system panics ("PANIC OUT OF TEXT").
2. The Text Table entry is initialized by setting the use count ("x_count") to one and the in-memory use count ("x_ccount") is zeroed since the text has not been read into nemory as yet. Since text.c/xalloc is a subroutine of the exec system call (sys1.c/exec), the size of the text is available in "u_arg[]". (Sys1.c/exec has already read the 16 byte header of the object file.) This byte value is rounded up to the nearest memory block, converted to memory block granularity and placed in "x_size".
3. Space is allocated on the swap device to hold the text. If there is no space available, the system panics ("PANIC OUT OF SWAP SPACE"). The address of the text ("x_daddr") on the swap area is set at this time.
4. Before calling text.c/xalloc, sys1.c/exec freed all of the memory space occupied by the overlaid process. Only the U block is retained so that the overlaying process may inherit some of the characteristics of the parent. This memory space is now expanded to encompass both the U block and the space required for the text. In addition, the User Memory Management registers are loaded so that a virtual address space that can be read or written (equal to the size of the text) is set up in this area (directly below the U block).
5. The text is read into memory by setting up the "u_base", "u_count", "u_segflg", and "u_offset[]" entries so that rdwr.c/readi can be called. "u_segflg" has already been set by sys1.c/exec to indicate that the I/O will be transfered to the user's virtual address space. The offset ("u_offset[]") is set so that the 16 byte header4is skipped and the destination is set up so that the data is copied to virtual address 0 in the user's address space.
6. Once the reentrant text is in memory, a copy is made on the swap device of the text (only). While this is being done the process must be locked in memory.
7. After the copy of the text is made on the swap device, the process doing the exec is unlocked. The text pointer is set ("p_textp") and the -Inode Table entry for the text is marked (ITEXT), so that the text cannot be modified on the filesystem until the last process using the text leaves the system.

8. The process is now truncated to the size of the U block (that is, the text in memory is discarded.)
9. For text that must be created fresh or for existing text that must be read in because it is not in memory, the following procedure is used:
    a. The context of the process is saved (in "u_rsav" and "u_ssav") and the process is swapped out. At this time, the process consists only of it's U block.
    b. The process is marked (SSWAP in "p_flag"), so that when the process resumes it will return to sys1.c/exec and the processor is relinquished (slp.c/swtch). Thus, the text is effectively abandoned until the process is again brought into memory by the Scheduler.
10. For processes which are executing a reentrant text that is already in memory, none of the procedures 2-9 need be done. Only the in-memory count "x_ccount" need be updated (incremented by 1).

xccdec

CALL

xccdec(xp)
struct text *xp;

RETURNS

No value is returned.

SYNOPSIS

Decrements the in-memory usage count of a reentrant text.

DESCRIPTION

Reentrant text is maintained as a separate entity on the swap device and in-memory. It is therefore necessary to keep track of the number of processes that are using the in-memory copy so that the memory can be freed when all of the processes using it leave the system. Text.c/xccdec decrements the in-memory usage count ("x_count") and if it becomes zero, frees the memory occupied by that text.

xfree

CALL

xfree()

RETURNS

No value is returned.

SYNOPSIS

Decrements the usage count of reentrant text and removes the text from the system when the last process using that text leaves the system.

DESCRIPTION

This function decrements the usage count of a reentrant text. It is the complimentary function to text.c/xalloc. Associated with each reentrant text in the system is a Text Table entry. (These are described under text.c/xalloc) Text.c/xfree simply determines whether a process has a reentrant text portion ("p_textp" is nonzero) and decrements both the in-memory usage count ("x_count") and the usage count ("x_count"). The in-memory usage count is decremented by calling text.c/xccdec. A check is made by text.c/ifree to see if all of the processes using the reentrant text have disappeared from the system. When this occurs, text.c/xfree deallocates the space that was occupied by the text on the swap area andreleases the i-node for that text. While a reentrant program is being executed, it's i-node is kept in the mode Table. This is done to eliminate the need to read that i-node if another instance of the program is invoked and to prevent the reentrant text from being overwritten while there is a copy of that text executing in the system.

xswap

CALL

xswap(p,freeilag,oldsize)
struct proc *p;

RETURN

No value is returned.

SYNOPSIS

Performs housekeeping required to swap a process out.

DESCRIPTION

Two indicators are passed to text.c/xswap. The first ("freeflag") indicates whether the memory occupied by the swapped process should be freed.

The only time that the memory occupied by a process is to be retained after a swap is when a parent creates a child by forking. The second argument ("oldsize") tells text.c/xswap what the former size of the, rocess was. This is necessary when swapping is used to grow the size of a process (slp.c/expand) so that while a space large enough to hold the new size is allocated on the swap area, only the former size will be placed in that area. The steps in swapping a process out are,

1. Space is allocated on the swap area to place the process. Since a copy -of reentrant text remains on the swap area even when a reentrant process is brotrght into memory, there is no need to swap it Out If there is no space on the swap area for the process, the system panics ("PANIC OUT OF SWAP SPACE").
2. The memory usage count ("x_ccount") of any reentrant text for the process is decremented and if there are no processes in memory using the text, the memory occupied by the text is freed.
3. The process is locked in memory ("p_flag" = SLOCK) until I/O for the swap is completed.
4. The swap function (bio.c/swap) is called to swap out the process and if any errors occur, the system panics ("PANIC SWAP ERROR").

   /TI -4 5. The memory associated with the process (not including reentrant text) is freed unless a swap to do a fork is occurring.
6. The location of the process ("p_addr") is set to it’s location on the swap area and the process is marked as non-resident ("p_flag" = SLOAD) and is unlocked.
7. Since a process has moved to the swap device, the Scheduler is notified if there were formerly no ready processes on the swap device.

nosys

CALL

nosys()

RETURNS

A fatal system error is posted ("in u_error").

SYNOPSIS

A placeholding entry in function tables which results in a fatal error being posted.

DESCRIPTION

The address of this function is placed in any table of functions (System Entry Point Table, Block Device Switch Table, Character Device Switch Table, etc.), where no entry is required and where calling the function corresponding to that entry is an error. The fatal error code 100 is posted in "u_error" so that the trap handler can properly notify the process which originated the bad system call or table reference (i.e., terminate it or allow it to handle the Bad System Call signal which is sent).

nullsys

CALL

nullsys()

RETURNS

No value is returned.

SYNOPSIS

A system call that does nothing. It's address is used to fill in tables of functions in the system where no error should be posted when the entry is called.

DESCRIPTION

This entry is used in any table of functions in the system (System Entry Point Table, Block and Character Device Tables, etc.), where calling the function corresponding to that entry should result in no action and no error being posted. A deliberate use of the trap.c/nullsys entry is made as the first entry in the System Entry Point Table. This prevents multiple levels of indirect system calls (see trap.c/trap).

trap

CALL

trap (dev,sp,r1,newps,r0,pc,oldps)
char *sp;

RETURNS

Any minor errors encountered in processing a system call are reported to the user process and any major errors result in the process being signalled (and usually terminated).

SYNOPSIS

The system trap handler.

DESCRIPTION

All system calls are made by trapping to the operating system using the TRAP instruction. This is the most common trap generated. Hardware traps are also generated for illegal instructions, addressing violations, etc. UNIX can recover a user process from these errors either by terminating the process or allowing the process itself to regain control and recover from the error. The operating system does not, however, recover if any of these traps are generated while system functions are executed. These result in a system panic ("PANIC TRAP").

For every process in the system, UNIX maintains a set ("u_signal[]" in each U block) of signal actions (20) that can be received by that process. A signal may be sent to a process via the kill system call, however, within the system, many of the signals correspond to specific hardware events (traps) and result in a signal being sent to a process. The following is the list of traps that can be generated by the PDP-1l hardware and the signals that are sent (if any),

1. Bus error - signal 10 - SIGBUS.
2. Illegal Instruction - signal 4 - SIGINS.
3. Trace trap - signal 5 - SIGTRC.
4. IOT instruction - signal 6 - SIGIOT.
5. Power Fail,
6. EMT instruction - signal 7 - SIGEMT.
7. Trap instruction (i.e., system call).
8. Programmed Interrupt.
9. Floating Point Violation - signal 8 - SIGFP.
10. Segmentation Violation - signal 11 - SIGSEG.

The remainder of the signals are reserved for user initiated or system initiated communication with a process. As mentioned previously, signals corresponding to hardware errors may be sent via the kill system call and a process may choose to

ignore or to handle any or all of these traps (by using the signal system call). Only one signal cannot be caught or ignored (the kill signal, SIGKIL = 9).

Like the clock interrupt handler, the trap handler performs a variety of functions. The functions will be described in the order of their most frequent use.

The trap handler is called to interface every system call to the operating system. For the most part, a user program involks a library (C library or assembly language library) routine to properly set up the arguments to the system call, however, there is no reason why a program need involk the library routines. There are two basic forms of system calls. The first form is the direct system call:

```
sys 22      /trap instruction
arg 1       /argument number 1
arg 2       /argument number 2
```

The SYS pseudo instruction is recognized by the UNIX assembler as the TRAP instruction. The number (22 in this case) is an index into the System Entry Point ("sysent[]"). This instruction is assembled as a one word instruction with the lower 8 bits reserved for the trap number. UNIX has only 64 system entry points at present, so only 6 bits of the trap number are recognized. Immediately following this are the (one word) arguments to the system call. Some system calls pass the first argument (as specified by the system call descriptions in the UNIX Programmers Manual) in register R0. (The C compiler recognizes the fact that registers R0 and R1 are scratch..) Arguments assembled after the trap instruction are not necessarily in the order specified by the format of the system call in the UNIX programmers Manual. The number of arguments assembled after the trap instruction must be the number expected by the system as the trap handler advances the Program Counter to skip over these arguments.

Since some arguments are assembled after the trap instruction and these arguments are typically not constant values, they must be set up just before the system call is made. This, however, proves a problem for reentrant programs as the text segment is write protected. Thus, for all practical purposes, this form is useful only for system calls which have no argument or one argument which is passed in R0.

The second form of system call is the indirect system call:

```
.text
      sys 0  /indirect system call
      addr   /address of real
             /system call
.data
9:
    sys 22   /system call
    arg 1    /argument 1
    arg 2    /argument 2
```

System call zero is reserved for the indirect system call. This system call passes as its one and only (constant value) argument, the address of the real system call. A template for the real systern call is assembled in the data area just as it would have been if the first form bad been used. The advantage is, however, that since the template is in the data area it may be modified even if the program is reentrant. This is used for all multiple argument system calls by the C library to insure that any program may be made reentrant. The first entry in the System Entry Point Table is for the trap.c/nullsys function.. This function does nothing and its presence as the first entry in the table insures that multiple levels of indirect system calls can not be done.

The System Entry Point Table is an array consisting of 64 two word entries. The first word of each entry contains the number of arguments that must be fetched from the user’s virtual address space and the second is the address of the function within the system which corresponds to the system call. Once it has been ascertained that the trap is actually a system call, the type (direct or indirect) of system call is determined by checking the trap number.

For the direct systern calls, the number of arguments (specified in the System Entry Point Table) are fetched from the user’s virtual address space. The Program Counter is saved automatically on the system’s stack (U block) by the hardware trap mechanism and is available as an argument to trap.c/trap so that it can be found easily. Since the TRAP instruction is a one word instruction, the PC was automatically advanced to the next location when the trap occurred, so that it is only necessary to advance the PC to skip over the arguments following the TRAP initruction. The per process information in each U block can accomodate up to 5 arguments (in the array "u_arg[]") making a total of 6 that may be passed. (R0 is usually used to pass an argument by saving and restoring registers in the library interface. More could be passed in registers and in any event, the Argument Array "u_arg[]" could be enlarged.)

Each argument fetched from the user's address space is placed in the Argument Array.

For indirect system calls it is only necessary to step the PC by 2 to bypass the one argument to the indirect system call. Arguments are loaded into the Argument Array as with the direct system call.

Some system calls pass the address of a string as a file name or a path name. Ultimately, the nami.c/nami function is called to do pattuaame searches on these strings so that the variable "u_dirp" is set to "u_arg[0]" . (the first argument fetched from the user's address space) as a convenience. This means that a string argument for calls like open, create, link, etc. follows the convention that a pointer to the string name is setup as the first argument in the system call. Another convention used is that for system calls that involve file descriptors, the descriptor is passed in register R0.

Once all of the arguments have been set up, the system call can be made by selecting its address from the System Entry Point Table. In order to allow system calls to be prematurely terminated, as when a signal is sent to a process, entry into the system must be made in a particular manner. The function specified in the System Entry Table cannot be called directly. Rather, a dummy routine (trap.c/trap1) must be used. This is done because the execution of nonlocal goto's (mch.s/aretu) within the operating system depend on a variation of the context saving and restoring method to operate. Mch.s/aretu manipulates registers R5 and SP which are saved (by trap.c/trap1) in "u_qsav". The manipulation makes the execution of mch.s/aretu appear as if the function trap.c/trap1 returned (prematurely to trap.c/trap). The two word array "u_qsav" in the U black is used to save the stack position of the dummy function trap.c/trap1 and to restore it when mch.s/aretu is called (by slp.c/sleep).

The chief use of this setup is to terminate system calls prematurely when signals are caught. In particular, for processes that relinquish the processor at low system priority (wait priorities, "p_pri" value is greater than zero) a check is made before the processor is relinquished and after it is regained to see whether there is a signal pending. If there is a signal pending, then the Signal Processor sig.c/psig is called. The Signal Processor determines whether signals are to have the standard system action (see sig.c/psig) or are to be handled by the user process. If the signals are to be handled by the user, then slp.c/sleep executes a non-local goto to return to the trap handler so that the user's program is entered at the proper point. No error is posted when the system call is aborted.

For processes that relinquish the processor at high priority, the system does not have to worry about a long period of time that the process might asleep, so that it is sufficient to allow the system call to complete and to check in the trap handler to see if there are any signals pending. Before making this check however, the trap handler checks to see if any errors were posted (in "u_error") by the system call and if a fatal error ("u_error" > = 100) has been posted, a bad system call signal (signal 12 - SIGSYS) is sent to the process. This will result in the process being terminated if standard system action is in effect . When a minor error occurs ("u_error" < 100) in processing a system call, the error number is returned by the trap handler in register r0 and the C bit is set in the Processor Status word so that it can be tested by the C library interface. The library function can thus determine whether an actual value is being returned or whether an error has occurred.

The second most frequent use of the trap handler is in resolving Stack Violations, that is in allocating more stack space to a process. When the Stack Violation occurs, the instruction that was executing aborts. The abort only occurs at the end of a fetch cycle, so that it can be guaranteed that at least part of the instruction was executed. The Program Counter (PC) is updated before any fetches occur, so that it will indicate the next location or next portion of the instruction after the abort. (The length of art instruction depends on the addressing modes used and the type of instruction; single operand, double operand, etc. Before each portion of an instruction is fetched, the PC is incremented.). Determining what type of instruction aborted and how much to adjust it is not an easy matter since the PDP-11 possesses several addressing modes, autoincrement and autodecrement, which cause automatic updating of registers.

Once it has been determined that the trap is a Segmentation Violation and that the user's stack pointer is indeed beyond the area that has been allocated to him, it is necessary to determine whether the PC can be backed up so that the aborted instruction can be restarted. The mch.s/backup function backs up instruction. This is easily done for 11/45 and 11/70 processor's, as they possess a hardware register (Memory

Management Status Register 2), which contains the amount by which any of the registers have been autoincremented or autodecremerited. The registers can easily be adjusted to their old values and the instruction restarted. On 11/40 processor's., this register is not present and the fetch cycles of the instruction must be simulated to determine which fetch caused the abort. The autoincrement, autodecrement addressing modes pose problems for instruction backup on certain forms of instruction. In particular, instructions of the form:

cmp -(sp),-(sp)

cannot be backed up (see mch.x/backup for details).

If the mch.s/backup routine indicates that the instruction can be (and has been) backed up, a stack growth procedure may be started. The first step in this procedure is to determine whether adding stack space to the user's virtual address space will produce any address overlap or overflow the virtual address space already allocated to the process. This is done by calling main.c/estabur to make these checks and actually set up an image of the new Memory Management registers in the U block.

The amount of stack space added each time a stack violation occurs is 20 memory blocks (1280 bytes). Once it has been determined that there is enough user virtual space for the addition, the expansion can be done by calling slp.c/expand. This function will copy the existing process (including U block) to a new (larger by the increment) area of memory, or if not enough memory is available, will swap the process out to an area encompassing the new size. In either case, the copy results in the addition of the new area to the end of the physical area occupied by the process. Since stack segments grow downward in physical and virtual address space, the added memory should be lower (in physical core) and contiguous to the existing stack area. This is accomplished by simply copying the existing stack to the added physical memory. (Effectively the stack is shifted down by 1280 bytes in memory.)

The third most frequent use of the trap handler is for handling traps generated by executing the SETD instruction on processor's that do not have the Floating Point Processor option (all 11/40's). The start off function (crt0.s) within every C language program issues the SETD instruction. This instruction turns on double precision floating point mode. On processor's not having Floating Point hardware this instruction generates an illegal instruction trap, so it must be ignored. There is a Floating Point software package (fp.s) which may be loaded with any program not possessing floating point hardware to simulate the operation of floating point instructions. In order to use this package the user program must use the signal system call to direct any illegal instruction traps to the floating point package. The trap handler sends an illegal instruction signal to the process and calls the Signal Processor sigc/psig. every time a floating point instruction is executed on a processor not having floating point hardware. The floating point interpreter will execute a BPT (which produces a BPT trap) instruction if any non-floating point illegal instructions are executed.

Execution of the IOT, EMT, and BPT instructions by user processes produce a trap and a fatal error unless other arrangements have been made (via signal system call). Similarly, all bus errors and unrecoverable Segmentation Violations produce fatal errors.

All traps occurring while the processor is in Kernel mode result in a system panic. It is assumed that the operating system is completely debugged and cannot produce errors unless the hardware is in trouble. When this occurs, the message

KA6 = num1
APS = num2

is printed on the system console. The quantity numl is the contents of the Memory Management Address Register that maps the system's stack (U block - for non I & D space system Kernel Instruction Address Register 6 is printed, while for I & D space systems Kernel Data Address Register 6 is printed). This is the physical memory block (64 byte granularity), which corresponds to the beginning of the U block (virtual address 0140000 or 24K word). The second quantity, *num2*, is the virtual address within the U block of the stack frame built for the trap ba.ndler. The stack frame is described under mch.s/call and contains such things as the PC and PS at the time of the trap, the type of trap, etc. The physical address of the start of the stack frame can be computed as follows:

addr = KA6*0100 + APS - 0140000

The physical 18 bit address from this calculation may be used at the processor console to examine the stack frame when the system crashes. The stack frame is simply the arguments to the trap handler (see CALL above) in order from right to

left (oldps to dev).

Once this message is printed, the processor is placed in the WAIT state. If the continue switch on the processor console is stepped, the message "PANIC TRAP" is printed.

The last function performed by the trap handler is in conjunction with penalizing processes that use too much system time. Processes that hog system resources are divided into two categories; system bound and CPU bound.

CPU bound processes are identified and penalized by the clock interrupt handler. System bound processes spend most of their time executing system calls. In order to prevent system bound processes from monopolizing the processor, a scheme whereby the number of consecutive system calls made by a process before it roadblocks is kept. If a process makes seventeen consecutive system calls before being roadblocked, the process is preempted and it's priority is lowered by one point. This is a one time penalty and the next time the process has the opportunity to run, its priority is restored to its normal value.

The trap handler insures that a process's priority is restored to it's user mode priority when it returns to user mode execution. When a process is awakened, it receives the (software) priority that it roadblocked at. It is selected by the process Switcher (slp.c/swtch) on this basis and retains that priority until it roadblocks again or until the process returns to User mode execution. At this time, the priority is set to PUSER + u_nice. FUSER is defined as a low value (100) and "u_nice" is any user specified penalty or reward for the process (specified via the nice system call).

trap1

CALL

trap1 (function)
int (*function)();

RETURNS

No value is returned.

SYNOPSIS

Calls the appropriate entry point in the system when a system call is made. Necessary in order to terminate a system call to process a signal.

DESCRIPTION

This is a dummy system call made necessary by the way nonlocal goto's are executed within the operating system. When a signal is caught before a process roadblocks (slp.c/sleep) or just after if is awakened, the system call is terminated prematurely by using the mch.s/aretu function. Execution of this function by slp.c/sleep causes the trap.c/trap1 function to appear to have returned. (That is, control is passed to trap.c/trap.) In order to be able to execute the nonlocal goto, the stack position of trap.c/trap1 must be saved. R5 and SP are saved in the U block in "u_qsav". (See trap.c/trapi for a more complete explanation.)

COMMON SYSTEMS

UNIX OPERATING SYSTEM

DEVICE DRIVERS SEC.1

This index lists the authorized issues of the sections that form a part of the current issue of this specification.

ISSUE 1 1/30/76

1. GENERAL

This document describes functions contained in pidents from PR-1C302-01 as follows:

BIO01	BLOCK I/O
DC01	DC-11 COMMUNICATIONS INTERFACE
DH01	DH-11 COMMUNICATIONS MULTIPLEXER
DHDM01	DHDM MODEM CTL INTERFACE
DHFDM01	DHFDM NULL MODEM CTL INTERFACE
DN01	DN-11 ACU INTERFACE
DP01	DP-11 201 SYNCHRONOUS INTERFACE
HP01	RP04 MOVING HEAD DISK INTERFACE
HS01	RS03/04 INTERFACE
HT01	TU16 MAGNETIC TAPE INTERFACE
KL01	KL-11 OR DL-11 ASYNCHRONOUS INTERFACE
LP01	LINE PRINTER INTERFACE

2. PROGRAM CONVENTIONS

A. System calls are made with the first argument in register R0. When the system call is made, the contents of register R0 are moved to the per user control block (user.h) in the variable called u.u_R0. The remaining arguments of a system call are moved into the per user control block array u.u_arg (this means u.u_arg[0] is the second argument).

B. Arguments or results of executing some functions are often left behind in the per user control block. For example, nami.c/namei decodes a pathname into an inode pointer. In the process, a pointer to the inode of the parent directory is left in u.u_pdir. This means it is eady to make a directory entry for a file since the inode for the directory is available. (See the documented header user.h in PR-1C301.)

C. Inodes are always locked during manipulations to prevent simultaneous update by two processv:. The procedure is to always lock and increment the usage count of an inode even if it turns out that a user does not have access to that file. At the end of processing of the inode, the usage count is reduced by 1 if there was an error, and in either success or failure, the inode is unlocked.

D. Error processing that reflects errors back to the user are set in the per user control block error flag (u.u_error). These error conditions can be referenced by the user program through the external variable "errno". (See Section 2 of Programer’s Manual for list of error conditions.)

E. If I/O processing is to be done on a device, the particular driver for that device must be called. Devices are known by major and minor numbers stored in an inode. The system calls the particular device driver indirectly through the major device number. A block switch table and character switch table are defined at system generation time. The major device number is used as a displacement into this table and the appropriate routine is called. For example, the code:

```
(*bdevsw[maj].d_close)
```

will call the close entry point for the driver associated mith major device "maj".

bawrite

CALL

bawrite(bp)
struct buf *bp;

RETURNS

No value is returned.

SYNOPSIS

Performs an asynchronous write to a block device.

DESCRIPTION

Most write operations to a block device under UNIX are either delayed or performed asynchronously. This means that a process may not experience any of the latency time or transfer time associated with outputting to a device. Only delay time due to scarcity of available system buffers will be experienced. In addition, performing the write asynchronously allows efficiency gains when multiple references to the same block occur. Bio.c/bawrite write sets up an asynchronous write for one 512 byte block., The process requesting the write does not wait for completion.

When bio.c/bawrite is called, the block to be written has already been moved to the appropriate device queue by a higher level function so that no allocation of a buffer to a device queue is needed as for reads. The device strategy routine (rp.c/rpstrategy, rf.c/rfstrategy, etc.) is called to queue the block for I/O by bio.c/bawrite. Errors detected by the strategy routine are posted (in "u_error") by calling bio.c/geterror. Bio.c/bawrite uses common code from bio.c/bwrite to call the strategy routine and report any errors. When I/O is completed, the block will be returned to the queue of available blocks on the freelist (by the higher level routine calling bio.c/brelse), however, it will also remain linked to the device queue so that any future reference to that block (by bio.c/getblk) will find it in the system.

bdwrite

CALL

bdwrite(bp)
struct buf *bp;

RETURNS

No value is returned.

SYNOPSIS

Performs a delayed write to a block device.

DESCRIPTION

Most write operations need not be done immediately. It is convenient in most cases to merely return a buffer that is to be written onto a block device to the pool of free buffers ("bfreelist") until some more convenient time to write it out. In this way, a subsequent read or write request which required data within that block will find the data within the system and extraneous operations can be eliminated. The buffer, however, must be marked ("b_flags") with an indicator (B_DELWRI), so that if free buffers are required by other processes, the block will be written out (asynchronously by bio.c/bawrite) and another chosen. The delayed write block would then be returned to the free queue but would not be a candidate for reallocation.

The buffer that is to be written will thus experience the following movements:

1. The buffer will be allocated to a device queue by some higher level function using bio.c/getblk. This means a buffer is removed from the freelist and placed on the device’s queue. (The block may already be in the system so that there would be no need for allocation.)
2. The data is then copied into the buffer by the higher level routine.
3. When the delayed write is issued, the buffer is marked (B_DELWRI in "b_flags") and placed at the end of the queue of available blocks on the freelist. The buffer is still, however, linked on the device queue.
4. The block will linger on the freelist until another request for that block is made (by bio.c/getblk) or until a free buffer is needed. If a buffer is on the free list long enough to reach the head of the free queue, and free buffers are needed, an asynchronous write (bio.c/bawrite) is issued for the delayed write

block and the next available block is chosen. The asynchronous write will result in the buffer being unlinked from the free list and placed on the device's I/O queue until the write is completed.

5. When the write is completed, the buffer is returned to the end of the free list, however, it is still linked to the device queue so that if that block is referenced by some process, the block will be found in the system.
6. The block will linger on the available list as in 3 until either another free buffer is needed and the buffer has reached the head of the free list or until some other process requests the data in that buffer. If a free buffer is needed, the buffer no longer must be written out so that it can be allocated to the requester as a free buffer. Only at this time does the block disappear from the device queue.
7. In order to prevent very active devices on the system from accumulating a large number of delayed write blocks, the sys3.c/sync function causes all delayed write blocks to be written out at least at the frequency that the UPDATE process runs (in multi-user every 30 seconds). This also minimizes discrepancies between in memory data and device data if the system crashes.

Since magnetic tape is a sequential medium, delayed writes must be disallowed as blocks must be written sequentially and allowing blocks to accumulate on the freelist as delayed write blocks runs the risk of disturbing the order in which the blocks reach the magnetic tape. Delayed writes are, however, allowed for DEC Tape (TC11) since it is really a random access device even though it is a sequential medium. This makes it necessary to flush out delayed write blocks when the DEC Tape is closed (tc.c/tcclose). (This is done by calling bio.c/bdwrite from bio.c/bflush.)

bflush

CALL

bflush(device)

RETURNS

No value is returned.

SYNOPSIS

Flushes write behind blocks out of the I/O subsystem for a particular device or for all devices.

DESCRIPTION

The inertia built into the block I/O subsystem by use of the write behind feature (delayed write - see bio.c/bdwrite) allows a number of redundant I/O operations on the same block to be eliminated. This is done by allowing write behind blocks to be returned to the system's available buffer queue until a later time when they are written out. This could, however, produce problems if the system crashes, as those blocks would not be updated since they were allowed to lie in the I/O subsystem. To flush these write behind blocks out of the I/O subsystem, the bio.c/bflush function is called by the the UPDATE (alloc.c/update) process once every 30 seconds. The delayed write blocks are written out (by bio.c/bflush) asynchronously (see bio.c/bawrite).

When each dealyed write buffer is written, it is temporarily removed (bio.t/notavail) from the available list and made busy (B_BUSY in "b_flags"). It is returned to the end of the available list once the write has been completed. Since the block is returned to the available list and its linkage on the device queue ("b_forw", "b_back" on "devtab") is undisturbed, the block still appears on the device queue. The only difference is that it has been assured that the block has been updated on the block device.

Detaching any filesystem or closing a DEC Tape (the TC11 is a random access device even though it is a sequential media) requires that all write behind blocks for that device be flushed out. Bio.c/bflush is called by both sys3.c/sumount and tc.c/tcclose.

If bio.c/bflush is called with "device" number NODEV (-1) then the write behind blocks associated with devices are flushed out, otherwise only those write behind blocks associated with "device" are flushed out. The UPDATE process calls bio.c/bflush with NODEV as an argument. Since the available queue of buffers ("bfreelist") is examined to find delayed write blocks and this queue is the one from which all allocation and deallocation of buffers occurs, the processor's priority must be raised to 6 to prevent interrupts from changing the status of buffers or altering linkages.

binit

CALL

binit()

RETURNS

No value is returned.

SYNOPSIS

Initializes the block device buffers and determines the number of block devices on the system.

DESCRIPTION

UNIX possesses a pool of 512 byte buffers ("buffers") which are used for buffering reads or writes from any device or can be used by the system for holding any data and whose size does not exceed 512 .bytes. Associated with each block device is a device queue ("devtab"), which links together all of the blocks that have currently been read or written from that device. There is also a queue of available buffers ("bfreelist") which chain buffers together. The linkages on the device queue ("devtab") and available queue are arranged so that a buffer may appear on both queues at the same time.

Each buffer has a header ("buf") which contains linkages, byte count information, status flags,and a pointer to an associated 512 byte buffer. A buffer header ("bfreelist") which has no associated buffer is used as the anchor linking together all available buffers in a ring. All buffer headers contain two pairs of pointers; "av_forw", "av_back" and "b_forw", "b_back". Each pair contains a forward and backward pointer so that in searching through the list, neighboring buffers on any queue may be found immediately. One pair of pointers ("av_forw" "av_back") is used to link together a ring anchored by "bfreelist" containing all of the buffers that are currently available for allocation. The second pair of pointers ("b_forw" and "b_back") are used to link the buffer onto a device queue. "Bfreelist" also uses these pointers so that it may act as a device queue. This queue is used for linking together blocks which are allocated for some purpose other than device I/O and for which it is undesirable to have them associated with a device.

The device queue also has two pairs of pointers but uses them for different purposes. The first pair ("b_forw", "b_back") are used to link together all buffers that have been used or are being used for I/O to the device. This queue is also arranged as a ring with "devtab" as anchor. The second pair of pointers ("d_actf", "d_actl") are used to order buffers that are currently queued to be read or written. The arrangement here varies with the device but is usually a single thread chain ordered according to the strategy used to access that device (First Come First Served - FCFS, SCAN, SSTF, etc.).

A block may appear on both the available list and the "b_forw", "b_back" list of the device (since the block, though available for use by other processes was last accessed from the device). This allows the elimination of many I/O operations since the desired block may be found on the device queue.

Initializing the block device buffers is done by setting up the pointers in the buffer headers and allocating each buffer to the free list. The algorithm used by bio.c/binit is essentially as follows. First, a ring with no buffers is created with "bfreelist" as its sole member by initializing all of its pointers to itself. Thereafter, a buffer is allocated to this queue (by calling bio.c/brelse) and header information is added to each buffer. Coincident with this, each buffer is allocated to the null device queue ("b_forw", "b_back" pointers in "bfreeelist") so that the "b_forw" and "b_back" pointers are initialized. The process is repeated for each of the system buffers. The header informal that is initialized is as follows:

The "b_dev" entry must be set up so that no buffer starts out as associated with any device ("b_dev" = -1).

The buffers ("buffers") are allocated separately from the buffer headers("buf[]") so that a pointer ("b_addr") in the buffer header must be set to point to the address of the buffer. Buffer headers and buffers are allocated as an array and buffer header i ("buf[i]") is associated .with buffer i ("buffers[i]"). The reason that the headers are not allocated as part of the buffer is so that physical I/O may be interfaced to the block I/O subsystem by simply using a special buffer header which points to the data in the user’s process. Also, when debugging a core dump, all of the information about buffer status can be obtained by dumping the headers without the necessity of dumping buffers.

The buffer must be marked ("b_flags") busy (B_BUSY) until all of the linkages have been properly set up. Bio.c/brelse will reset the busy flag once the available pointers "av_forw", "av_back" have been set up.

Another function performed at initialization time is to determine how many block devices there are on the system so that major device numbers may be checked by higher level function. Table ("bdevsw") contains a 4 word entry for each device on the system and one blank (zero) entry following all of these entries to indicate the end of the table. Bio.c/binit scans the table looking for the first zero entry, counting each entry as it scans. The total number of block devices is loaded in an external variable ("nblkdev").

bread

CALL

bread(dev,blkno)

RETURNS

A pointer to a suffer containing the block "blkno" is returned. (Actually, a pointer to the buffer header is returned.)

SYNOPSIS

Performs a synchronous 512 byte read of a block device.

DESCRIPTION

Since a program cannot operate on data until it is available, only synchronous reads are used under UNIX (i.e., any higher level request for a block is forced to roadblock until the data is available). A certain amount of inertia is built into the block I/O buffering scheme so that there is a possibility that the desired block "blkno" may already be in one of the system's buffers. This is more likely to be true if a process is attempting to read data in 512 byte chunks. If the block is in the system it can be grabbed before it leaves the system thus saving a read operation. A new buffer can be allocated from the pool of free buffers ("bfreelist") if the block is not already in the system. All reads result in a request for 512 bytes of data, even though the higher level function calling for the read may only need a small portion of this data. The buffer is set up for a 512 byte read (the word count "b_wcount" is set to 256) and the buffer is marked for a read (B_BREAD set in "b_flags"). In both cases, the buffer is marked ("b_flags") as busy (B_BUSY) for the period of time that the read is scheduled to take place and/or while the data is required for use by a process.

If the block must be read, the buffer is placed on the appropriate block device queue by calling the device strategy routine. The argument "device" indicates the major device number so that the proper device strategy routine may be selected from the Block Device Switch Table ("d_strategy" in "bdevsw"). There is no possibility of a reference to the table being out of bounds since the major device number was checked at higher levels of software (against "nblkdev"). The process that requested the read is roadblocked until the read, has completed by calling bio.c/iowait. When the read is complete,d the device interrupt handler will mark the buffer as having been filled by setting the done indicator (B_DONE) in the "b_flags" entry of the buffer. Any errors occurring in the read will be reported by the device interrupt handler.

breada

CALL

breada(device,blkno,rablkno)

RETURNS

A pointer to a buffer containing the block "blkno" is returned. (Actually, a pointer to a buffer header is returned.)

SYNOPSIS

Performs read ahead on "device".

DESCRIPTION

Read ahead is a technique whereby an attempt is made to anticipate where the next read request on a device will be and to preread that data In this manner, the program requesting the read will not be subjected to positional and rotational latency or device queuing, if read ahead is completed before the next block is requested. There are different strategies that can be used for-doing read ahead. On UNIX, all reads (not including physical I/O) result in a full 512 byte block being read from the device. Smaller amounts of data could be read if a program requested it, however, since disks transfer times are small in comparison to positional and rotational latency times, any extra transfer is inconsequential. Also, most DEC disks are designed around a 512 byte sector and while fewer than 512 bytes may be specified, the disk controller is busy until a full sector has been transferred. By reading the full 512 bytes, any subsequent read or write which references data within that block will not have to be read (if the block does not leave the system). Besides the advantage gained by reading a minimum of 512 bytes instead of the desired quantities, the next

block is anticipated and read under certain conditions. Thus, one request will spawn several read requests to bring data into the system. A routine for finding a block that is already in memory (bio.c/incore) must be available to determine whether any reads need be done and the read ahead strategy must be capable of determining when read ahead should be discontinued so that superfluous reads are not generated.

The strategy adopted under UNIX is to pursue read ahead as long as a process is reading (512 byte blocks) sequentially through a file or a device. When the first non sequential read is requested, read ahead is discontinued and is not restarted until sequential accesses begin again.

Bio.c/breada carries out the read ahead opera-_tion. Starting and stopping read ahead and determining which block number in a file or on a device is the read ahead block ("rablkno") is done by the higher level function rdwri.c/readi.

In implementing the read ahead strategy, bio.c/breada makes use of bio.c/incore to determine whether a block is already in memory. For the desired block "blkno", the bio.c/breada function behaves exactly like bio.c/bread. That is, a synchronous read is performed and the process requesting the read is roadblocked until it is completed. Since the desired block may already be within system, bio.c/incore is called to look for that block among the buffers on the freelist ("bfreelist"). If the block is already in memory, bio.c/bread is called to get the buffer. If the desired block has not already been read by a previous read operation then bio.c/getblk is called to see if the block is possibly on a device queue waiting for its turn to be read. If that is not the case, a buffer is allocated for the read and the appropriate device strategy routine is called. Bio.c/breada does not wait (yet) for the read to complete. Rather, it goes through a similar operation for the read ahead block "rablkno". Bio.c/incore is called to search the free list of buffers ("bfreelist") to see if the block was read in a previous read operation. Nothing will be done, of course, if the read ahead block is in memory. If it is not in memory, bio.c/getblk is called to search the device queue for it or to allocate a block so that it may be read. The device strategy module is called to read the read ahead block, however, the buffer will be marked (B_ASYNC in "b_flags") so that when the read completes the buffer is returned to the pool of available buffers. Bio.c/breada then waits for the read of the the desired block to complete. It does not wait for the read ahead block.

An external variable "raflg" is available for turning of all read ahead on aildevices. "Rang" is initialized to one, however, by changing it to zero read ahead is eliminated. As with bio.c/bread any error detection is done as a result of the interrupt handler indicating an error to bio.c/incore and a system error (in "u_error") being posted. These errors are of no concern to bio.c/breada or bio.c/bread and are used only at higher levels of software to return errors to the user.

brelse

CALL

brelse(bp)
struct buf *bp;

RETURNS

No value is returned.

SYNOPSIS

Releases a buffer to the queue of available buffers on the freelist ("bfreelist").

DESCRIPTION

This function takes the buffer "bp" and places it on the queue ("devtab") of available buffers. Neither the data in the buffer nor it’s linkage to the device queue are destroyed, however, so that the block appears on both queues. (The available queue "bfreelist" and the queue it was released from, "devtab".) In this way, the block appears as available for allocation, yet it retains the identity of the data that resides in the buffer. It is this fact that allows subsequent references to the block to eliminate unnecessary 110 operations.

The "av_forw" and "av_back" pointers in the buffer headers link together available blocks on the freelist ("bfreelist"). By simply inserting the buffer "bp" at the end of the freelist (it will become the last block on the "av_forw" chain) it becomes an available buffer. Changing buffer linkages must not be interrupted so that the processor’s priority is raised to 6 to lock out interrupts from all block devices.

Several conditions must be checked before a block is released to the available list.

1. A check must be made to see if there is a request for this block by some other process (B_WANTED set in "b_flags"). The B_WANTED flag is set if a process requests a block when it is busy (B_BUSY set in

"b_flags"). The busy flag is set when a buffer is found or allocated by bio.c/getblk and is not reset until the buffer is released (bio.c/brelse). Thus, any reference to the block while it is busy will result in the wanted flag being set in the buffer and the process that references the busy block being roadblocked. It is the duty of bio.c/brelse to awaken all processes that are waiting for this buffer (by calling slp.c/wakeup).

2. If the available queue of buffers on the freelist is empty, then bio.c/brelse must notify bio.c/getblk that one buffer is now available. (Bio.c/getblk will roadblock any process that requires a buffer if there are none available.)
3. If the buffer being released was never read or written properly (B_ERROR set in "b_flags") then in order to prevent any other process from finding the buffer containing bad data the minor device number ("d_minor") of the device number ("b_dev") is destroyed (set to -1 to destroy any associativity). The buffer will still be handled as any other block, that is, it will appear on two queues, but the device number which is checked by bio.c/getblk is destroyed so that even though the block is still on the device queue it will not be recognized. It can, however, be allocated as a free buffer.

bwrite

CALL

bwrite(bp)
struct buf *bp;

RETURNS

No value is returned.

SYNOPSIS

Performs a synchronous 512 byte write on a block device.

DESCRIPTION

Most write operations in the block I/O subsystem are performed asynchronously, however, there are several operations (updating superbiocks, updating i-nodes, updating freelists, etc.) which cannot be delayed. Functions within the system that require writes have already allocated a buffer (by calling bio.c/getblk) so that there is no need to pass a device number to bio.c/bwrite as is done with the read functions. The device number is already set in the buffer header ("b_dev") so that bio.c/bwrite need only set the write flag in the buffer header ("b_flags") and queue the buffer on the device for writing by calling the device strategy routine. (Actually, since a buffer may be used only for reading or writing, the absence of the read flag, B_READ, indicates that a write is to be performed.) Since the write is to be performed synchronously, the process requesting the write is roadblocked until the write is completed.

Bio.c/bwrite provides common code for bio.c/bdwrite and bio.c/bdwrite, however, they will be discussed under their respective functions.

clrbuf

CALL

clrbuf(bp)
struct buf *bp;

RETURNS

No value is returned.

SYNOPSIS

Zeros a 512 byte system buffer.

DESCRIPTION

Clearing a buffer of its contents is really a service provided to higher level functions and is not used by any of the functions in bio.c. It is used by the magnetic tape strategy routine to pad out a block with zeros if a block smaller than 512 bytes is to be written. It is also used every time a block is allocated to a file in the file system. This is done so that there is no old data residing in a file if the file’s length is not a multiple of 512 bytes.

devstart

CALL

devstart(bp, devloc,devblk,hbcom)
struct buf *bp:
int *devloc;

RETURNS

No value is returned.

SYNOPSIS

Loads a block device controller’s registers to initiate a transfer.

DESCRIPTION

Since the controller registers on most block devices manufactured by Digital Equipment Corporation have the same form and relative positioning, a common routine may be used to load the device

registers. The new PDP-11 common controller RH11 has a slightly different format so that a comparable routine rh.c/rhstart is used for these devices.

The parameters passed and the register which the are loaded are:

1. "bp" - This is the address of the buffer header which contains information about the buffer to be written. The header contains location, word count, memory extension, device number, operation, etc.
2. "devloc" - This is an address in the device controller. It is the Unibus address of the cylinder (or possibly sector) register of the controller. Only controllers which have the following four registers in the following order may use bio.c/devstart.)
   a. Command and status
   h. Word Count
   c. Bus Address
   d. Cylinder, Sector or Track Address Register.
3. "devblk" - This is either the Cylinder, Sector or Track to be loaded into register 2d above. This value is computed by the device startup routine (rp.c/rpstart, rk.c/rkstart, etc.).
4. "hbcom" - This is a flag indicating whether a read or a write is to be issued to the controller. The Command and Status Register is the last register to be loaded as it actually initiates the transfer.

Bio.c/devstart is called by block device start routines (rp.c/rpstart, rf.c/rfstart, etc.).

getblk

CALL

getblk(device, blkno)

RETURNS

Returns a pointer to a system buffer. If a device number that is out of range is passed to getblk, a system panic will occur ("PANIC BLKDEV").

SYNOPSIS

Determines whether a given block from a device is already in the system and if not, allocates a buffer.

DESCRIPTION

Bio.c/getblk is used by any function within the system that must do 512 byte I/O. It retrieves the desired block if it-is already within the system or allocates a fresh buffer if it is not within the system. With the linkage setup as described under bio.c/binit, a block remains associated with the device that it was last used for even though it has been returned to the available queue. In this way, the presence of a buffer which had previously been read or written may be detected and unnecessary I/O eliminated. In addition, write behind operations (see bio.c/bdwrite) actually return buffers to the available list without being written. This is done in the hope that the block will be accessed soon afterwards, so that several writes to the same block will result in only one transfer to the device. Write behind blocks cannot be allowed to lie in the I/O subsystem forevthe buffer will remain busy until after the process has finished using the data When the buffer is released (bio.c/brelse) any other processes requiring the data in that buffer will be notified (via slp.c/wakeup). In order to prevent a redundant wakeup being issued by bio.c/brelse when busy buffers are released, bio.c/iodone resets the B_WANTED bit before issuing a wakeup. For asynchronous I/O, the buffers are released immediately by calling bio.c/brelse, so this is not a problem.

iowait

CALL

iowait(bp)
struct buf *bp;

RETURNS

No value is explicitly returned, however, bio.c/iowait does cause any I/O errors to be posted ("u_error").

SYNOPSIS

Roadblocks a process until the block "bp" has been read or written.

DESCRIPTION

All synchronous reads or writes must wait for I/O to complete. Waiting for I/O to complete is done by the higher level routines (bio.c/bread, bio.c/breada, bio.c/bwrite) calling bio.c/iowait to roadblock the process requesting the read or write

until the I/O is completed. The process is road-blocked at priority PBIO (-50) to decrease its likelihood of being swapped. The process is road-blocked until the B_DONE bit in the "b_flags" entry of the buffer is set by the interrupt handler. If the I/O cannot be completed by the device driver, the B_DONE bit and the B_. ERROR bit in "b_flags" is set by the device interrupt handler. Bio.c/iodone is called to set the completion flag (B DONE) but the interrupt handler sets the error indication (B_ERROR) itself. Bio.c/iodone sends a wakeup to all processes waiting (in the bio.c/iowait function) for the buffer. When a process that is waiting for the buffer is awakened, bio.c/iowait will find the B_DONE bit set and will call bio.c/geterror to post (in "u_error") any error that may have occurred.

physio

CALL

physio(strategy, bp, device, rdflg)
struct buf *bp;
int (*strategy)();

RETURNS

No value is explicitly returned, however, if an error occurs it is posted ("u_error").

SYNOPSIS

Performs address mapping and checking for physical I/O.

DESCRIPTION

Physical unbuffered I/O is the only means by which I/O can be done. There are a number of advantages and disadvantages to using physical I/O.

1. In addition to no filesystern mapping being performed, the I/O is unbuffered. This means that the data is read or written directly from the user's address space. To allow this two things must be done.
   a. The start address and end address of the user's buffer area must be checked to see that it lies within the user's virtual address space. For normal buffered I/O this need not be done. The rdwri.c/iomove function catches any memory violations when copying data from a user's process into one of the system's buffers.
   b. The bio.c/physio function has no idea as to which block devices are word oriented so that transfers must specify an even number of bytes.
2. Since the I occurs directly from the user's program, tt program must be locked in core.
3. Since I/O occurs directly to or from a user program, all of the advantages of read ahead and write behind are lost to a program doing physical I/O. This means that all of the latency (positional and rotational) to perform the I/O will be experienced by processes doing physical I/O. While it is true that none of the overhead of copying data from the user's program to a system buffer is encountered, this is a small amount of time in comparison with the average positional or even the average rotational latency of the secondary storage devices that are currently available. There is a point at which physical I/O will give more throughput to a device than buffered UNIX I/O, however, this point varies with the speed of the processor (11/40, 11/45, 11/70) and the average rotational latencies of the devices (RK11, RF11, RSO4, RP03, RP04, etc.). Additional positional delays on spiral reads may also effect the crossover point on the smaller devices (RK11).
4. Physical I/O like swap I/O uses a special buffer header for each device ("rptab", "rktab", etc.). Since there is only one of these headers per (major) device, only one physical I/O operation to each ,controller at a time can be queued. (Most controllers are busy - i.e. not available while read or write operations are occurring.)

With the above background in mind the operation of bio.c/physio will be described.

When any I/O operation is requested, a common function sys2.c/rdwr is called to determine what the target file (i-node) is. There are three other quantities which can be determined by sys2.c/rdwr.

1. The current position in the file. This may be found in the "f_offset[]" entry in the File Table and is transferred to the "u_offset[]" entry in the U block to avoid repeated indirect addressing when referencing it.
2. The virtual address (in user space) where the transfer is to begin can be obtained from the second argument of the read or write system call. This is a byte address and is placed in "u_base" for convenience.

3. The number of bytes to be transferred can be obtained from the third argument of the read or write system call. This byte count is placed in "u_count". With the values in "u_base" and "u_count", the virtual area from or to which I/O is to be directed is defined.

Since all block device controllers work from physical addresses instead of virtual addresses, it is necessary to relocate "u_base". (For regular 512 byte UNIX I/O, relocation need not be performed because the system buffers have the same physical and virtual addresses.) Since the operating system assigns physical space to a process when it is created and since a process may only initiate I/O on data within its own address space and there are no holes in physical memory (that is, it is contiguous from beginning to end) bio.c/physio need not check to see that the buffer area is beyond the limits of physical memory. It is sufficient to insure that the buffer is within the virtual address space of the user's process.

One additional restriction is placed on the user's buffer. The buffer must not be within the text area of the user's process. It may only be within the data or bss area of the user's process. This is done because requesting a physical write into the text area of a reentrant program would cause a Segmentation Violation by the controller. (The ability of a program to write it's own text out is also lost.) No equivalent restriction exists for regular 512 byte UNIX I/O, so that a program may read or write into it's own text area. (If the prgram is reentrant, requesting a write into the text area will result in a Segmentation Violation since reentrant text is write protected by the system. The rdwri.c/iomove function would detect this when the system buffer was copied into the write protected text.) At present, when user programs are separated into I and D space areas, the issue of reading or writing the text area will become superfluous.

Checking whether the buffer ,pis within the data, stack or text area can be done by checking the quantities "u_dsize", "u_ssize" and "u_tsize". These are the size of the text, data and bss area in memory blocks (64 bytes). In order to do the checking, bio.c/physio must know how the text, data and stack areas are loaded (by main.c/estabur) in the system. The text area is loaded in the low physical (and virtual address space), followed by the data (and bss) area. The stack area is allocated (physically) directly below the data area (however, in virtual address terms the stack is in the high virtual address area). The checks that are made are:

1. A check is made to see that the user's buffer begins on an even address boundary and that the byte count specifies an even number of bytes (i.e., ends on a word boundary). This done is because most block devices cannot transfer an odd number of bytes or fetch from an odd address.
2. A check is made to see that the buffer does not extend beyond the limits of the virtual address space of the program. This is done by checking to see that the address of the end of the buffer ("u_base" + "u_count") is greater than the address of the beginning of the buffer ("u_base").

   A check is also made to see that the buffer is not within the text area. This is done by examining the address of the user's buffer ("u_base") to see if it is beyond the last address in the text area ("u_tsize"). Before this check can be made, the "u_tsize" address must be rounded up to the nearest 4K virtual memory address. This is due to the way virtual address space is allocated by main.c/estabur and the way programs are loaded by the UNIX loader. This address is compared with the starting address of the buffer "u_base" to see if the buffer begins within the (adjusted) virtual address space of the text.
3. Another check tests to see that the transfer does not span the virtual address gap between the end of the data area.and the beginning of the stack area. This is done by determining whether the end of the buffer is within the data area of the program (the start of the buffer was checked above) or the start of the buffer is within the stack area (extending beyond the virtual address space of the program was checked in 2).

With the above checks made, it can be guaranteed that the I/O will not abort due to addressing errors unless there is a genuine hardware problem. (The I/O may also abort due to an illegal disk address specified - i.e., "u_offset[]".)

After the above checks have been made, the buffer header can be set up for the transfer. Since there is only one buffer header per major device, use of this buffer is restricted to one process at a time. To accomplish this, access to the physical buffer header is restricted so that only one process can do I/O to a particular device at a time. This is achieved by setting the busy (B_BUSY) flag in the "b_flage entry of the buffer header when a

process enters the address computation part of bio.c/physio. The busy flag is reset by bio.c/physio only after the I/O has been completed. (The device interrupt handier will indicate the completion of I/O by setting the B_DONE flag.

Any processes which tries to do physical I/O to a device on which physical I/O is already being done (B_BUSY set) is roadblocked and the wanted flag (B_WANTED) is set in the header. When the process doing physical I/O on the device has finished using the buffer header all processes waiting to do physical I/O on that device are awakened and the B_WANTED and B_BUSY flags are reset. (Under physical I/O, there is no need to copy the data into the user's address space as for normal I/O, so the buffer header is busy only as long as it takes to set up and do the I/O.)

Setting up the physical I/O buffer header is done in a manner similar to that of bio.c/getblk for buffered I/O The chief difference is that memory extension bits must be calculated and the virtual address specified in the read must be relocated to a physical address. The physical address is determined by consulting the User Memory Management registers and determining what physical area of memory is currently mapped by the virtual addresses and adjusting the buffer address appropriately. The quantities that are set in the buffer header are:

"b_addr" is set to the physical address corresponding to the start of the user's buffer.

"b_flags" is set up to contain any memory extension bits, to indicate that the buffer header is busy (B_BUSY) and to indicate whether a read or write (the argument wrdfle) is to be performed.

"b_blkno" is set to the block on the device where the transfer is to start. This means that reads or whites are restricted to begin on 512 byte boutitlaries on the device (on record boundaries for magnetic tape). Requesting information that does not begin on a 512 byte boundary results in the read or write starting at the nearest 512 byte boundary anyway.

"b_wcount" is set to the word count for the transfer. This has implications for use on certain block devices(all disks) as transfers that involve transfers smaller than a sector size (sector sizes vary among disks from the word addressable RF11 to the 512 byte larger disks RP03, RK11, R1304, etc.). On these devices, a request for I/O that is not a multiple of the sector size will result in zeros being padded in the last sector written. (This would destroy any data that existed in the remainder of that sector.)

"b_error" is set to zero. This is for future use in reporting individual error numbers from devices.

"bdev" is set to the device number for the transfer "device".

Once all of these parameters have been set up in the buffer header, the device strategy routine ("d_strategy" in "bdevsw") can be called to queue the header on the device. It should be noted that unlike buffered I/O whete the buffers are chained onto the device queue (by bio.c/getblk) so that they may be found later, the physical I/O headers are not queued on the "b_forw", "b_back" chain of the device queue.

One additional precaution must be taken when doing physical I/O. That is, the process must be locked in core so that it is not swapped out while the I/O is occurring. To do this, the SLOCK bit is set in the appropriate Process Table entry ("p_flag"). Once the I/O is completed the lock bit is reset.

Error reporting is done in the same manner as for normal I/O (by calling bio.c/geterror). Any of the addressing errors checked are reported directly in "u_error" by setting the system error EFAULT.

Physical I/O uses the residual word count entry ("b_resid") in order to allow the device drivers to report exactly how many words were read or written.

swap

CALL

swap(blkno, coreaddr, count, rdflg)

RETURNS

Returns a 1 on error.

SYNOPSIS

Performs physical I/O to the swap device.

DESCRIPTION

This is essentially a stripped down version of bio.c/physio (which does I/O directly from the user's address space). The reason that it is a separate function is that the physical I/O function is serially reusable (per device) and multiple

requests for I/O to the same device are queued (i.e., only one can take place to a device at a time). Using only one common function might mean that the swap would compete with physical I/O if the swap device was on the same device as the physical I/O. Another reason for their separation is that since the system has control of its resources (i.e., it knows how much memory or swap space is available and where it is), there is no need to worry about validating any of the addresses or word counts.

A special buffer header ("swbuf") is used for the swap device. The information in this header is filled in as is done by bio.c/getblk for normal I/O. The data filled in is:

"b_flags" - The B_BUSY flag is used to indicate whether the buffer is in use or not and the B_WANTED flag is set if any other process requires use of the swapper. (The Scheduler is not the only process that does swapping. A process may swap itself out.) The B_READ flag is also set based on the value of "rdflg" passed to bio.c/swap. (A 0 in the B_READ bit position indicates that a write is to be done.) Address extension bits are computed for placing in "b_flags" based on the value of "coreaddr". This argument contains the number of memory blocks (32 words/block) that are to be swapped out. When a process is swapped it is swapped in one of two forms.

1. If it is reentrant, the U block and data (data, bss and staci) are swapped as one piece. When the process was created the reentrant text was created and remains separately on the swap area so that it need never be swapped out.
2. If it is nonreentrant the U block, text and data (data, bss, stack) are swapped as one piece.

The parameters loaded are:

"b_dev" - set to the swap device "swapdev" which was specified when the system was compiled (conf.c).

"b_addr" - set to the lower 16 bits of the address from or to which data is to be swapped. The argument "coreaddr" is in granularity of memory block (32 words/block) and so must be adjusted to a byte address (left shift 6 bits).

"b_count" - set to the word count for the transfer. It is obtained from the "count" argument which is in memory blocks (left shift 5 bits for word granularity).

"bbikno" - this is the logical block number of the destination from "blkno".

The process requesting the swap is roadblocked until the swap is completed and any other process requesting a swap while the swap buffer is busy is roadblocked until the swap buffer is free. Swapping has the highest software priority in the system when roadblocking a process(even higher than normal I/O). Any error occurring during the swap is reported by returning a nonzero value to the caller and in the case of the Scheduler any error results in a system panic ("PANIC SWAP DEVICE").

dcclose

CALL

dcclose(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Logically closes a terminal attached with a DC11. asynchronous interface.

DESCRIPTION

In order to logically close a terminal attached with a DC11 interface, dcclose only has to change the software state (tstate) for the proper minor device (dc11[rdev.d_minor]) to closed and, after forcing any data on the output queue (t_outq) to be transmitted, flush the device's I/O queues (see tty.c/wflushtty).

dcopen

CALL

dcopen(dev, flag)
int dev, flag;

RETURNS

No value returned.

SYNOPSIS

Logically opens a terminal attached with a DC11 asynchronous interface.

DESCRIPTION

The deepen routine does a logical open of a terminal attached with a DC11 interface. Dcopen must first filter out requests to open invalid devices. Assuming that the device number "dev" is satisfactory, the device's control structure (dc11[dev.d_minor]) is selected and the device is placed into a waiting-to-open state (t_state). Note that for the purpose of selecting an actual DC11, the minor device number plus one corresponds directly to the DC11's position in the system. For example, minor device two causes the third DC11 on the system to be selected.

If the device is not already in the open state (i.e. the device was closed; it is permissible to invoke dcopen more than once without intervening closes), the terminal control structure (struct tty) for the device and the DC11's hardware status registers are initialized, and the state of the device is set to open. Note that the initial line speed selection is always for the next to lowest that is available on the DC11.

At this point, the device, whether previously opened or being opened for the first time, is simultaneously in the software states (tstate) of open and waiting-to-open. All that remains for dcopen to do is to await a carrier signal on the DC11 (see dc.c/dcrint) if one is not already present. Once carrier is detected, the waiting-to-open stigma is removed and the opening of the device is complete.

dcread

CALL

dcread(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Read routine for terminals attached with a DC11 asynchronous interface.

DESCRIPTION

The dcread routine is the DC11 driver's interface to the general purpose functions of UNIX that handle terminal 110. All read requests to terminals attached with a DC11 interface must pass through dcread. If the DC11 device "dev" still has a carrier signal, then dcread merely has to invoke tty.c/ttread to obtain input data from the canonical queue (t_canq).

dcrint

CALL

dcrint(dev)
in dev;

RETURNS

No value returned.

SYNOPSIS

Handles interrupts that occur in the receive portion of the DC11 asynchronous interface.

DESCRIPTION

As the interrupt handler on the receive side of a DC11 interface, dcrint must diagnose the DC11's

state in order to decide what to do about the interrupt.

In particular, dcrint must first determine if a carrier signal is still present on the DC11 device "dev". If there is no such signal, the software state of the device (t_state) is altered to reflect loss of carrier. Additionally, if the device is not in the waiting-to-open state (see dc.c/dcopen), then it was open for some user and the carrier dropped for some reason. This necessitates disabling the receiver, flushing the device's I/O queues (see tty.c/flushtty), and signaling to the user that carrier has been lost.

The action taken when a carrier signal is present on the line depends on whether or not the DC11 has sensed an error. If an error is indicated (which could signify a carrier transition or be a ring indication) and the DC11 is in the waiting-to-open state (see dc.c/dcopen), the device's state is changed to reflect presence of the carrier and dc.c/dcopen is awakened so that it may conclude open processing. Otherwise, the interrupt is assumed to have been caused by receipt of a character by the device. If the character's parity is allowable under the mode of the terminal flags), it is placed on the device's raw input queue (t_rawq; see tty.c/ttyinput); otherwise, it is discarded.

dcsgtty

CALL

dcsgtty(dev, array)
int dev, *array;

RETURNS

No value returned.

SYNOPSIS

Determines or modifies the mode of a terminal attached with a DC11 asynchronous interface.

DESCRIPTION

A nonzero value of "array" signifies that information about the current state of the terminal attached to the DC11 device "dev" is desired in response to a gtty system call (see tty.c/gtty). In this case, "array" is assumed to be a three word array into which dcsgtty places the current line speeds (t_speeds), zero, and the current state (t_flags) of the device and its associated terminal.

Conversely, a zero value of "array" implies that the state of the device is to be respecified in response to a stty system call (see tty.c/stty). Pending output (t_outq) is first written and the device's I/O queues are flushed (see tty.c/wflushtty). The speeds kspeeds) and mode (t_flags) of the terminal are then respecified as requested. If the requested speeds are reasonable (that is, the DC11 is wired for the speeds specified by the user), the DC11's transmitter and receiver status registers are altered to reflect the new line speeds. Whether or not a speed for the transmitter or receiver is reasonable is determined by the presence of a nonzero entry in the proper position, of the DC11 driver's dctstab[] or dcrstabn table, respectively.

dcwrite

CALL

dcwrite (dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Write routine for terminals attached with a DC asynchronous interface.

DESCRIPTION

The dcwrite routine is the DC11 driver's interface to the general purpose functions of UNIX that handle terminal I/O. All write requests to terminals attached with a DC11 interface must pass through dcwrite. If the DC11 device "dev" still has a carrier signal, they dcwrite merely invokes tty.c/ttwite to place the data on the output queue (t_outq) and initiate transmission.

dcxint

CALL

dcxint(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Interrupt handler for the transmit portion of a DC11 asynchronous interface.

DESCRIPTION

The dcxint routine receives control whenever an interrupt on the transmit side of a DC11 interface occurs. Transmission of the next character in the

output queue (t_outg) of device "dev" is initiated (see tty.c/ttstart). If this queue is empty or has reached its low water mark, all processes waiting for such events are awakened. The former event is waited on by tty.c/wilushtty; the latter, by processes waiting for the -queue to shrink to a reasonable length before putting more data on it.

dhclose

CALL

dhclose(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Logically closes a terminal attached with a DH11 asynchronous interface.

DESCRIPTION

In order to logically close a terminal attached with a DH11 interface, dhclose has to change the software state (t_state) for the proper device (dh11[dev.d_minor] to closed, and, after forcing any data on the output queue (t_outq) to be transmitted, flush the devices I/O queues (see tty.c/wflushtty).

dhopen

CALL

dhopen(dev, flag)
int dev, flag;

RETURNS

No value returned.

SYNOPSIS

Logically opens a terminal attached with a DH11 asynchronous interface.

DESCRIPTION

Before actually attempting to open a terminal connected with a DH11 device, dhopen must filter out requests to open invalid devices by checking the plausibility of the device number "dev". Assuming that this test is passed, the control structure for the device (i.e., terminal) is selected (dh11[dev.d_minor]) and the device is placed in a waiting-to-open state (t_state). When mapping device numbers to actual communications lines, the minor device number is taken to be the corresponding line in the DH11. For example, DH11 minor device three is taken to be line three in the DH11 itself. It should be noted that this version of the DH11 driver does not support multiple DH11s on a system.

If the line (i.e., minor device) is not already open (i.e., the device was closed; it is permissible to open the device more than once without intervening closes), the device's (terminal's) control structure (struct tty) is initialized and the hardware line parameters are set (see dh.c/dhparam). All that remains to be done in order to conclude the open is to wait for a carrier signal on the line. Exactly how this is done depends on whether the device has a DM11 modem control (see dhdm.c/dmopen) or not (see dhfdm.c/dmopen). Once a carrier signal has been detected, the opening procedure is completed by changing the device's state from waiting-to-open to open.

dhparam

CALL

dhparam(tp)
struct tty *tp;

RETURNS

No value returned.

SYNOPSIS

Sets line parameter information in the DH11 asynchronous interface.

DESCRIPTION

Information about the characteristics of a communications line must be set in the DH11 interface for each line that is to be active. This is the purpose of the dhparam routine. In particular, given a terminal descriptor structure (struct tty) pointed to by "tp", dhparam must analyze the line speed (t_speeds) and parity (t_flags) information. The hardware control information corresponding to this software line speed and parity is then loaded into the DH11's Line Parameter Regiiter. The net effect is to force agreement between the software and the DH11 hardware for the speed and parity of the line in question.

dhread

CALL

dhread (dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Read routine for terminals attached with a DH11 asynchronous interface.

DESCRIPTION

The dhread routine is the DH11 driver's interface to the general purpose functions of UNIX that handle terminal I/O. All read requests to terminals attached with a DH11 interface must pass through dhread. If the DH11 device "dev" still has a carrier signal, then dhread only has to invoke tty.c/ttread to obtain input data from the canonical queue (t_canq).

dhrint

CALL

dhrint()

RETURNS

No value returned.

SYNOPSIS

Handles interrupts that occur in the receive portion of the DH11 asynchronous interface.

DESCRIPTION

As the receive interrupt handler for the DH11 interface, dhrint is called whenever a character is received by the device. (Note that this version of the DH11 driver does not use the silo feature of the DH11 to its fullest extent, as the silo alarm level is left at zero.) Dhrint determines on which line the character was received, and discards the character if either the corresponding device has not yet been opened or there was a parity error in receiving the character. Otherwise, the received character is placed on the raw input queue (t_rawq) of the device corresponding to the line on which it was received (see tty.c/ttyinput).

dhsgtty

CALL

dhsgtty (dev, array)
int dev, *array;

RETURNS

No value returned.

SYNOPSIS

Determines or modifies the mode of a terminal attached with a DH11 asynchronous interface.

DESCRIPTION

A nonzero value of "array" signifies that information about the current state of the terminal attached to the DH11 device "dev" is desired in response to a gtty system call (see tty.c/gtty). In this case "array" is assumed to be a three word array into which dhsgtty places the current line speeds (t_speeds), zero, and the current state (t_flags) of the device and its associated terminal.

Conversely, a zero value of "array" implies that the state, of the device is to be respecified in response to a stty system call (see tty.c/stty). Untransmitted data on the output queue (toutq) is transmitted and the device's input queues are flushed (see tty.c/wflushtty). The speeds (t_speeds) and mode (t_flags) of the terminal are then reset as requested. This also entails resetting the DH11 hardware's concept of the speed and mode of the communications line (see dh.c/dhparam).

dhstart

CALL

dhstart(tp)
struct tty *tp;

RETURNS

No value returned.

SYNOPSIS

Special start routine for initiating the transmission of characters on a DH11 asynchronous interface.

DESCRIPTION

Because the DH11 is a sixteen line multipixer, the general purpose device start routine of UNIX (tty.c/ttstart) is not sufficient for initiating transmission. The routine dhstart is invoked by tty.c/ttstart to fulfill this device start function for the DH11 interface. Dhstart initiates transmission of an output character to the terminal described by the control structure pointed to by "tp". No action is taken, however, if the line is already busy with a transmission or there are no characters on the output queue (t_outq). Once a character has been obtained from the device's output queue, a determination must be made as to whether, or not

it is a delay character (e.g., a delay character would precede a carriage return). For "normal", nondelay characters, transmission of the character on the appropriate communications line is initiated, the state (t_state) of the device (i.e., line) is changed to show that it is busy, and the driver's bit map of DH11 lines (dhsar) is altered to show that transmission was initiated on that line. This bit map is used by the DH11 transmitter interrupt handler (dh.c/dhxint) to determine the lines for which transmission has completed. Delay characters are handled by requesting a system timeout (see clock.c/timeout) after the delay time has expired.

Regardless of the nature of the character obtained from the output queue, dhstart's final responsibility is to check the length of the device's output queue. If sufficiently short, any processes waiting for it to shrink before putting more characters on it (i.e., write to the device) are awakened.

dhwrite

CALL

dhwrite(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Write routine for terminals attached with a DH11 asynchronous interface.

DESCRIPTION

The dhwrite routine is the DH11 driver's interface to the general purpose functions of UNIX that handle terminal I/O. All write requests to terminals attached with a DH11 interface must pass through dhwrite. If the DH11 device "dev" still has a carrier signal, then dhwrite merely invokes tty.c/ttwrite to place the data on the device's output queue (t_outq) and initiate transmission.

dhxint

CALL

dhxint()

RETURNS

No value returned.

SYNOPSIS

Handles interrupts that occur from the transmit portion of the DH11 asynchronous interface.

DESCRIPTION

Because the DH11 is a multiplexer and a transmitter interrupt may be caused by any one of the sixteen lines, the DH11 driver's transmitter interrupt handler, dhxint, must first determine exactly which line(s) caused the interrupt. This is done by comparing the current state of the DH11's Buffer Active Register (which shows the lines for which transmission is active) against a bit map (dhsar) that shows the lines that have had transmissions initiated (see dh.c/dhstart). The product of this comparison is a bit map of all the lines whose transmission has completed since the last interrupt. The software status of each such line is changed to inactive (i.e., not busy) and transmission is reinitiated on the line (dh.c/dhstart).

dmint

CALL

dmint()

RETURNS

No value returned.

SYNOPSIS

Handles interrupts from the DM11 modem control.

DESCRIPTION

As the interrupt handler for the DM11 modem control, dmint must determine which line caused the interrupt and what the carrier transition was (i.e., was carrier attained or lost). The action to be taken depends on the line's new carrier state.

If carrier is not present, the control structure for the line is changed to reflect absence of carrier (tstate). Additionally, if the device was not waiting-to-open (i.e., was open, see dh.c/dhopen), the loss of carrier indicates a hangup situation. In this case, a hangup is sig naled to the process(es) associated with the line and the device's I/O queues are flushed (see tty.c/flushtty).

The only other possibility is that a carrier signal is present. The device's software state (t_state) is changed to reflect this fact and dhdm.c/dmopen, which is awaiting a carrier signal before continuing open processing for the device, is awakened.

dmopen

CALL

dmopen(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Logically opens a DM11 modem control.

DESCRIPTION

This version of the dmopen routine is used in conjunction with the DH11 driver (dh.c) whenever the DM11 modem control is used on the DH11. A fake routine, dhfdm.c/dmopen, is used in lieu of this routine for those DH11s that do not have a DM11.

Dmopen is invoked by dh.c/dhopen each time a line attached to the DH11 is opened. It must check to see if a carrier signal is already present on the line corresponding to device "dev". If this is not the case, dmopen is obligated to wait until carrier has been received (see dhdm.c/dmint) before continuing. In any event, once the line has a carrier signal, dmopen may return to dhopen for the conclusion of open processing.

dmopen

CALL

dmopen(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Fake open routine for the DM11 modem control.

DESCRIPTION

When a DH11 device is opened, the dh.c/dhopen routine calls upon dmopen to logically open the DM11 and await a carrier signal on the line being opened. The version of dmopen found in dhfdm.c is used in conjunction with those DH11s that do not have a DM11 modem control; the routine dhdm.c/dmopen is used whin modem control is available.

Since dhfdm.c/dmopen is essentially a fake routine that emulates the real DM11 driver, at least as far as the DHII driver is concerned, it need only change the state (t_state) of device "dev" to indicate presence of a carrier signal.

dnclose

CALL

dnclose(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Logically closes the DN11 ACU interface.

DESCRIPTION

In order to logically close the DN11 device "dev", dnclose only has to clear the device's status register.

dnint

CALL

dnint(dev)
int dev;

RETURNS

No value'returned.

SYNOPSIS

Interrupt handler for the DN11 ACU interface.

DESCRIPTION

As the interrupt handler for the DN11, dnint only has to awaken dn.c/dnwrite so that the next digit of the call may be dialed.

dnopen

CALL

dnopen (dev, flag)
int dev, flag;

RETURNS

No value returned.

SYNOPSIS

Logically opens the DN11 ACU interface.

DESCRIPTION

In order to logically open the DN11, dnopen must choose the proper ACU as determined by the device number "dev" and verify that the unit is available for use. If unavailable, an error is indicated (u_error). Otherwise, dnopen enables the chosen ACU. It should be noted that the argument "flag" only serves to maintain syntax compatibility with other device open routines.

dnwrite

CALL

dnwrite(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Write routine for the DN11 ACU interface.

DESCRIPTION

As the write routine for the DN11, dnwrite is responsible for interpreting and dialing a telephone number written to the DN11 device "dev". This is accomplished by serially obtaining each digit of the number and presenting it to the DN11 for dialing. Before obtaining each digit, however, dnwrite must first ensure that the DN11 is ready to accept a digit for dialing. If the device is not yet ready, then dnwrite is, of course, obligated to wait until it is (see dn.c/dnint). Once the unit is ready, a digit may be obtained from the user's buffer and given to the DN11 for dialing. This sequence is repeated for each digit of the telephone number until either an error occurs or the call is completed (answered).

A special case in the handling of the telephone number's digits is that of the hyphen (-) character, which is not written to the DN11. Rather, this character causes dnwrite to delay eight seconds before continuing the dialing sequence. This is useful for such things as waiting for a second dial tone.

dpclose

CALL

dpclose()

RETURNS

No value returned.

SYNOPSIS

Logically closes a DP11 synchronous interface.

DESCRIPTION

In order to logically close the DP11, dpclose must disable transmit and receive interrupts from the device, indicate the device is no longer in use by any process (dp_proc), and, relinquish the I/O buffer that was obtained by dp.c/dpopen (dp_buf).

dpopen

CALL

dpopen(dev, flag)
int dev, flag;

RETURNS

No value returned.

SYNOPSIS

Logically opens a DP11 synchronous interface.

DESCRIPTION

Before logically opening the DP11 interface, dpopen must verify that it is not already open for some other user (actually, process); in which case, an error condition is indicated (u_error) and the open is not done. The user process that first opens the DP11 may open it as many times as desired without intervening closes. On the initial open, an I/O buffer is obtained (see bio.c/getblk) and the driver's internal timer (dp_timer) is set for a minute (see dp.c/dptimeout). Dpopen sets the sync character to octal 26 and initializes the DP11 device's transmitter and receiver registers.

The arguments "dev" and "flag" are not used by this routine, but serve to maintain syntax compatibility with other device open routines.

dpread

CALL

dpread()

RETURNS

No value returned.

SYNOPSIS

Read routine for the DP11 synchronous interface.

DESCRIPTION

As the read routine for the DP11 driver, dpread is invoked each time the user reads from the device. An immediate return is done if a check of the DP11 device's status reveals that it is no longer in a ready state (see dp.c/dpwait). If any characters have been received and placed in the I/O buffer (dp_buf; see dp.c/dprint), then they are moved to the user's I/O buffer and the read request is considered complete. Note that the number of characters actually given to the user is always the minimum of the number requested and the number available.

However, it may be that no data has yet been received. In this case, dpread checks to see if a timeout has occurred (see dp.c/dptimeout); a timeout reveals that at least five seconds have passed without any data being received. Rather than continue to wait for receipt of data, a return is done with no characters returned to the user. Otherwise, the routine waits for a device status change or timeout to awaken it (see dp.c/dpturnaround), at which time it goes back through the algorithm just described.

dprint

CALL

dprint()

RETURNS

No value returned.

SYNOPSIS

Handles interrupts on the receive side of the DP11 synchronous interface.

DESCRIPTION

As the receive interrupt handler for the DP11, dprint is given control whenever a character (seven data bits plus one parity bit) is received by the device. The character is retained only if the

device is in a READ state (dp state) (to prevent overwriting the I/O buffer if it contains data being transmitted; see dp.c/dpwrite) and the I/O buffer is not already full (a full buffer results if the user is lax or unable to read data that has been received). Before the character is placed in the I/O buffer, it is converted to odd parity if not already so.

Note that after synchronization has been achieved by receipt of the initial two (consecutive) sync characters (octal 26), this DP11 driver causes any further sync characters to be retained and treated as normal data characters until the synchronization is broken (see dp.c/dpturnaround).

dpstart

CALL

dpstart()

RETURNS

No value returned.

SYNOPSIS

Transmits a character on the DP11 synchronous interface.

DESCRIPTION

Every character that is transmitted on the DP11 device must pass through dpstart before actually being sent. This routine resets the driver's internal timer (dp_timer) to five seconds; this allows ample time to do an actual transmission. A failure to complete the transmission in this amount of time results in a timeout and the flushing of any untransmitted characters that are in the I/O buffer (see dp.c/dpturnaround). Dpstart then checks to see if there are more characters to be sent (dp_nxmit). If there are none, it switches the device state (dp_state) to READ to indicate that the buffer is now free to be reused; otherwise, the first untransmitted character in the buffer ("dp_buf") is converted to odd parity and placed in the DP11's transmitter buffer.

dptimeout

CALL

dptimeout()

RETURNS

No value returned.

SYNOPSIS

Performs timing function for the DP11 synchronous interface driver.

DESCRIPTION

The dptimeout routine is called once a second by the system (see clock.c/clock and dock.c/timeout). It decrements the driver's internal timer (dp_timer). If the timer goes to zero, the DP11 is turned around (see dp.c/dpturnaround) and the timer is reset to one second to ensure that dp.c/dpread knows that a timeout occurred. Its last responsibility to request another wakeup by the system in one second.

dpturnaround

CALL

dpturnaround()

RETURNS

No value returned.

SYNOPSIS

Turns around the DP11 synchronous interface.

DESCRIPTION

The dpturnaround routine is invoked either because of a timeout situation in the DP11 driver's clocking mechanism (see dp.c/dptimeout) or because of a change in the device's status that may affect data reliability (see dp.c/dpxint). The driver's timer (dp_timer) is reset to five seconds and the device's receive active bit is turned off (the ramifications of this are described below). In addition, if the device was in the WRITE state (dp_state), the state is changed to READ and any untransmitted characters in the I/O buffer are discarded. Finally, those waiting to read or write to the device are awakened.

The net effect of the above actions is that, insofar as reading from the device is concerned, a resynchronization of the device (two consecutive sync characters) is required before data may be received (i.e., reenable the receive active bit). As far as writing goes, any characters written to

(Page missing from the original material)

hpstart

CALL

hpstart()

RETURNS

No value returned.

SYNOPSIS

Initiates the actual I/O to an RP04 device.

DESCRIPTION

If there are any I/O requests on the RP04 queue (chained from d_actf in hptab), hpstart marks the RP04 as active (hptab.d_active) and initiates I/O for the first request in the queue.. The real work of activation is done by rh.c/rhstart but hpstart must fill in an RP04 controller register that is not filled in by rhstart. Although the physical block address has already been computed by hp.c/hpstrategy, hpstart must complete the logical to physical device mapping by selecting the actual physical drive.

hpstrategy

CALL

hpstrategy(bp)
struct buf *bp;

RETURNS

No value returned.

SYNOPSIS

Places an I/O buffer on the RP04's queue of I/O buffers to read/write.

DESCRIPTION

The strategy routines for disk and tape drivers provide two major services: to place I/O requests on the device's queue of pending requests in an order that is most efficient for that particular device, and, to verify that the request's logical block address conforms to the logical (i.e., minor) device policy of the device driver. In particular, disk and tape strategy routines do the following specific things for each I/O request, which the driver sees in terms of a pointer "bp" to a buffer header (struct buf):

1. Verify that the block address given in the I/O request is a plausible address for the logical device being read/written. That is, ensure conformity to the logical device policy of the driver. Since there are 22 sectors per track and 19 tracks per cylinder, 418 blocks are on each cylinder
2. For devices that have several logical devices on a single physical device, translate the block address on the logical (i.e., minor) device to a true block address on a physical device. The remainder of the translation (selecting the physical drive) is peformed elsewhere, usually in the driver's start routine. Note that for drivers that map a logical device to one or more physical devices this step is omitted and the address translation is done elsewhere, usually in the driver's start routine.
3. Place the I/O request in the device's queue of pending I/O requests (work to do queue). The location within the queue where the request is placed depends on the queuing strategy being employed for the device. The queue itself is chained from d_actf in the device's devtab (e.g., hptab, hstab, etc.) Immediately prior to rechaining the queue to insert the - request, the processor's hardware priority must be raised to that of the device to disable interrupts from the device.
4. Cause physical I/O to be initiated if there are no previous requests currently being serviced.

The strategy routine for the RP04 disk, hpstrategy, performs all of the above functions. The logical to physical device mapping is accomplished by dividing the logical (i.e., the minor) device number by eight. The quotient of this division is interpreted as the controller drive number and the remainder is the logical file system on that drive. For example, a minor device number of 13 is construed to be logical file system five on physical drive one. Each physical drive is divided into eight logical file systems as follows:

file	cylinders	# of blocks
0	0 - 23	9614
	24 - 43	UNUSED
	(Can be swap area)	
1	44 - 200	65535
2	201 - 357	65535
3	358 - 407	20900
4	0 - 99	40600
5	100 - 199	40600
6	200 - 299	40600
7	300 - 399	40600

The queuing strategy used is the "elevator" technique. That is, when a request is made, if there are none or one other request on the queue, the new request is placed at the end of the queue (First In First Out (FIFO) strategy). If there are already two or more pending requests, the new request is inserted so that all requests on the queue are in cylinder number order. Whether this order is increasing or decreasing by cylinder number is established when there are two requests on the queue. All requests for the same cylinder are handled in a FIFO manner.

hpwrite

CALL

hpwrite(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Interface to RP04 driver for "raw" mode write requests.

DESCRIPTION

The write routines for disk and tape drivers are the functions that handle "raw" mode write requests for their respective devices. That is, they are the interface between users making such requests and the I/O subsystem, and are called whenever a write is done to the raw device. They usually do little more than invoke bio.c/physio to do the real work involved with "raw" I/O, but are vitally necessary, as they inform physio of such things as the device strategy routine to invoke to make the I/O request. Hpwrite merely verifies (see hp.c/hpphys) that the "raw" write request will remain entirely within the bounds of the RP04 logical device in question before calling physio.

hsintr

CALL

hsintr()

RETURNS

No value returned.

SYNOPSIS

Handles interrupts from the RS03/RS04.

DESCRIPTION

Hsintr, as the interrupt handler for the RS03/RS04 disk, performs the functions described in rp.c/rpintr.

hsread

CALL

hsread(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Interface to RS03/RS04 driver for "raw" mode read requests.

DESCRIPTION

Hsread handles raw mode read requests to the RS03/RS04 disk by calling bio.c/physio. See rp.c/rpread for a discussion of raw mode device read routines.

hsstart

CALL

hsstart()

RETURNS

No value returned.

SYNOPSIS

Initiates the actual I/O to an HS03/HS04 device.

DESCRIPTION

If there are any I/O requests on the HS03/HS04 queue (chained from d_actf in hstab), hsstart marks the HS03/HS04 as active (hstab.d_active). After filling in the disk address extension error register, hsstart initiates I/O for the first request in the queue by invoking rh.c/rhstart. The arguments to devstart reflect the logical device policy of the driver (see hs.c/hsstrategy), as they are the translation of a logical device block number to a physical device address.

hsstrategy

CALL

hsstrategy(bp)
struct buf *bp;

RETURNS

No value returned.

SYNOPSIS

Places an I/O buffer on the RS03/RS04's queue of I/O buffers to read/write.

DESCRIPTION

Hsstrategy performs for the RS03/RS04 disk the general strategy functions described in rp.c/rpstrategy. The I/O buffer queuing strategy employed is strictly First In First Out (FIFO). The logical device policy of the driver is that the logical device number (minor device) is used to identify the type of disk and the physical drive. Minor device numbers 0-7 define eight drives respectively on an RS03, whereas minor devices 8-15 define 8 drives on an RS04.

hswrite

CALL

hswrite(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Interface to RS04/RS03 driver for "raw" mode write requests.

DESCRIPTION

Hswrite handles raw mode write requests to the RS04/RS03 disk by calling bio.c/physio. See rp.c/rpwrite for a discussion of raw mode device write routines.

hcommand

CALL

hcommand(unit, com)
int unit, com;

RETURNS

No value returned.

SYNOPSIS

Issues a command to the TU16 magnetic tape controller.

DESCRIPTION

Before issuing a command to the TU16 controller, hcommand ensures that the controller is ready and that no other magnetic tape I/O requests are active (d_active in httab). The logical "unit" is mapped onto a physical unit (Unit 4-7 map into drive 0-3). Once these conditions are met, the controller command "com" is issued fo drive "unit".

htclose

CALL

htclose(dev, flag)
int dev, flag;

RETURNS

No value returned.

SYNOPSIS

Performs a logical close for the TU16 magnetic tape.

DESCRIPTION

Certain cleanup functions must be performed after a user has completed use of a TU16 magnetic tape device. These are done by htclose, which is called when the magnetic tape file is closed. In particular, the drive "dev" is marked as not in use (h_openf[dev]) so that others may now use it, a double end-of-file is written if the device was opened for writing ("flag" = 2). If the unit is 0-3 or 8-11 the tape is rewound.

htintr

CALL

htintr()

RETURNS

No value returned.

SYNOPSIS

Handles interrupts from the TU16 magnetic tape.

DESCRIPTION

As the interrupt handler for TU16 magnetic tape, htintr's first responsibility is to determine if an error occurred on the controller command. There are many possibilities if an error occurred.

1. If the error was not an end-of-file, is deemed recoverable, and was from an actual I/O request (as distinguished from the tape positioning initiated by ht.c/htstart), then the number of I/O errors that have already occurred for this request is checked. If fewer than ten errors have occurred, then action is taken to reposition the tape and retry the request. Otherwise, the request is abandoned as hopeless, is not retried, and becomes subject to the checks described in items 2 and 3 below.
2. Any error that does not result in a retry and which was not an end-of-file or from a raw mode I/O request causes the drive to be marked as unusable. Note that tape positioning errors and unrecoverable I/O errors fall into this category. This stigma holds until the device is closed.
3. All requests that are not retried are marked as in error and then ultimately handled as if they were successfully completed actual I/O requests (the handling of which is described below). This includes an end-of-file condition.

All successfully completed actual I/O requests are marked as complete and removed from the queue; for successfully completed tape positioning commands (done as a prelude to the actual I/O for those requests needing tape positioning, see ht.c/htstart) the current block number counter (h_blknon) is adjusted to reflect the tape's position and the request remains as the first in the queue. Ht.c/htstart is then invoked for the first request in the queue.

On an end of file if the request was for block I/O, the device is shut down (made unavailable for further reads) to prevent read ahead from moving the tape forward. This is to allow multifile tape processing.

htopen

CALL

htopen(dev, flag)
int dev, flag;

RETURNS

No value returned.

SYNOPSIS

Performs a logical open for the TU16 magnetic tape.

DESCRIPTION

Because of magnetic tape's sequential nature, only one user is allowed to access a given drive at one time. The function of htopen is to enforce this restriction by checking the availability of the specified device "dev". If the drive in question is already in use (h_openftp, then appropriate error bits are set (u_error). Otherwise, the drive is marked in use (h_openftclevi) and the block counters (h_biknon and h_nxrecfl) are initialized. The flag "flag", which indicates whether the open is for reading and/or writing, is not used and only serves to maintain syntax compatibility with other device open routines.

Units 0-7 are mapped into 800 BPI where as units 8-15 are mapped into 1600 BPI and the appropriate controller bits are set.

htphys

CALL

htphys (dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Computes the starting block number for raw mode I/O requests to TU16 magnetic tape.

DESCRIPTION

Prior to doing raw mode I/O to magnetic tape, the starting block number for the transfer must be calculated. This is necessary so that the tape may be properly positioned before starting the I/O (see ht.c/htstart). This duty is performed by htphys, which then records its findings in the block number indicator for drive "dev" (i.e., h_blkno[].

htread

CALL

htread(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Interface to TU16 driver for "raw" mode read requests.

DESCRIPTION

Htread handles raw mode read requests to TU16 magnetic tape. See rp.c/rpread for a discussion of raw mode read routines. Before calling bio.c/physio to do the real work, the starting block number of the requesto must be calculated by ht.c/htphys.

htstart

CALL

htstart

RETURNS

No value returned.

SYNOPSIS

Initiates the actual I/O procedure for TU16 magnetic tape.

DESCRIPTION

If there are any I/O requests on the TU16 queue (chained from d_actf in httab), htstart initiates the I/O procedure for the first request in the queue. This entails first verifying that any previous I/O to the device did not result in an unrecoverable I/O error (see ht.c/htintr) and that the controller is ready to accept commands. Failure to pass these two tests results in the I/O request being marked complete but in error and removing it from the I/O queue. Otherwise, a contrailer command is issued for the situation that pertains to the request.

1. If the tape is not correctly positioned to do the read or write, then a command is issued to advance or rewind the tape to the correct block.

2. In all other cases the actual I/O for the first request in the queue is initiated.

In either case the function being performed is recorded (in d_active) for later use by the interrupt handler (see ht.c/htintr).

htstrategy

CALL

htstrategy(bp)
struct buf *bp;

RETURNS

No value returned.

SYNOPSIS

Places an I/O buffer on the TU16's queue of I/O buffers to read/write.

DESCRIPTION

Htstrategy performs for TU16 magnetic tape the general strategy functions described in rp.c/rpstrategy. The I/O buffer queuing strategy employed is strictly First In First Out (FIFO). As might be expected, the logical device number is taken to be the physical drive number.

The block address verification for magnetic tape is because of the medium's sequential nature and the attempt to have it emulate a disk file system, more complex than that of most strategy routines and therefore merits elaboration. A block number counter (h_nxrec[]) is maintained for each drive. An I/O request may not be initiated to a block number exceeding the value of the drive's counter. Specifically, when the device is opened (see ht.c/htopen), the value of this counter is set to the size of the largest permissible file (65535 blocks). As long as only read requests are made, the value of the counter does not change. However, whenever a write request is made, the counter's value is set to the requested block number plus one (which will be, for almost all cases, the number of the block that will be written next). For every read or write request, the block number of the request is checked against the counter. If the request's block number exceeds the counter value, then the request is marked as complete but in error (u_error). Otherwise the request is considered valid. A special case is the situation where a read request is made for a block number that equals the counter's value. This could occur, for example, if records are being read and written and a read is issued for block n + 1 immediately after writing block n. In this case no actual I/O takes place, but the user's I/O buffer is zeroed and the I/O request is masked as completed.

htwrite

CALL

htwrite(dcv)
int dev;

RETURNS

No value returned.

SYNOPSIS

Interface to TU16 driver for "raw" mode write requests:

DESCRIPTION

Htwrite handles raw mode write requests to TU16 magnetic tape. See rp.c/rpwrite for a discussion of raw mode write routines. Before calling bio.c/physio to do the real work, the starting block number of the request must be calculated by ht.c/htphys.

klclose

CALL

klclose(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Performs a logical close of a terminal attached with a KL11 or DL11A asynchronous interface.

DESCRIPTION

Kiciose logically closes terminals attached with KL11 or DL11AA interfaces by invoking tty.c/wflushtty for the appropriate device "dev". This causes all characters on the device's output queue (t_outq) to be transmitted and flushes any characters on the device's input queues t_rawq, t_canq).

klopen

CALL

klopen(dev, flag)
int dev, flag;

RETURNS

No value returned.

SYNOPSIS

Performs a logical open of a terminal attached with a KL11 or DL11A asynchronous interface.

DESCRIPTION

The klopen routine does a logical open of a terminal attached with a KL11 or DL11A interface. This primarily entails the initialization of the terminal control structure (struct tty) for the appropriate minor device (kl11[dev.d_minor]). This entails computing the address of the device's registers, and certain assumptions are made in this respect. In particular, minor device zero is assumed to be the system console, since the console is typically connected with a KL11 or DL11A interface. Further, it is assumed that the minor device numbers for any additional terminals attached with a KL11 or DL11A interface have been assigned consecutively starting with minor device one. For example, if system has three KL11's, then they must be given minor device numbers zero (system console), one, and two. A final responsibility of klopen is to enable interrupts in the device's read and write status registers and to enable the reader, so that characters may be received from paper tape on the terminal (e.g., a teletype terminal).

The argument "flag" is nut used by klopen since the terminal is always opened for both reading and writing, but serves to maintain syntax compatibility with other device open routines.

klread

CALL

klread(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

The read routine for terminals attached with KL11 or DL11A asynchronous interfaces.

DESCRIPTION

The kiread routine is the KL11/DL11A driver's interface to the general purpose functions of UNIX that handle terminal I/O. In particular, all read requests to terminals attached with all or DL11A interfaces must pass through kiread, which does nothing more than invoke tty.c/ttread for the appropriate device "dev".

klrint

CALL

klrint(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Handles interrupts that result when a character is received from a terminal attached with a KL11 or DL11A asynchronous interface.

DESCRIPTION

The klrint routine receives control whenever an interrupt on the receive side of the KL11 or DL11A interface occurs. The character is retrieved from the read data buffer and placed on the raw input queue (t_rawq) for device "dev" (see tty.c/ttyinput). The reader is then reenabled,

so that characters may continue to be received if they are coming from paper tape (e.g., a teletype terminal).

klsgtty

CALL

klsgtty(dev, array)
int dev, *array;

RETURNS

No value returned.

SYNOPSIS

Determines or modifies the mode of a terminal attached with a KL11 or DL11A asynchronous interface.

DESCRIPTION

A nonzero value of "array" implies that information about the current state of the terminal is desired in response to a guy system call (see tty.c/guy) by the user. In this case klsgtty assumes that "array" is a three word array and places the current state (t_flags) of the terminal into the last word.

Conversely, a zero value of "array" implies that the terminal's state is to be changed in response to a stty system call (see tty.c/stty), but only after any pending output to the device has been physically transmitted.

Note that klsgtty totally disregards device speeds, both when determining and modifying the state of the device.

klwrite

CALL

klwrite(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

The write routine for terminals attached with KL11 or DL11A asynchronous interfaces.

DESCRIPTION

The klwrite routine is the KL11/DL11 driver's interface to the general purpose functions of UNIX that handle terminal I/O. With one exception all write requests to terminals attached with KL11 or DL11A interfaces must pass through klwrite, which does nothing more than invoke tty.c/ttwrite for the appropriate device "dev".

Tile system console is normally attached with a KL11 or DL11A interface. Since UNIX itself manipulates the device registers directly when printing system messages on the console (see prf.c/putchar), system writes to the console do not go through klwrite.

klxint

CALL

klxint(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Handles interrupts that result from transmitting to a terminal attached with a KL11 or DL11A asynchronous interface.

DESCRIPTION

The klxint routine receives control whenever an interrupt on the transmit side of the KL11 or DL11A interface occurs. Transmission of the next character in the output queue (t_outq) for device "dev" is initiated (see tty.c/ttstart). If the output queue is short enough, then processes waiting for it to shrink to an acceptable length before placing more characters on it (i.e., write to the device; see tty.c/ttwrite) or to empty (see tty.c/wflushtty) are awakened.

lpcanon

CALL

lpcanon(c)
int c;

RETURNS

No value returned.

SYNOPSIS

Edits characters written to the LP11 line printer.

DESCRIPTION

The lpcanon routine edits (i.e., translates as required) the character "c" being written to the LP11 printer and causes the resultint character(s) (which may include new lines or page ejects) to be placed on the printer's output queue. There are essentially two phases of editing, the first being done only for half ASCII (64 character) printers, and the second being done for all printers.

Lpcanon must be compiled for either a 64 or 96 character printer; it is not able to dynamically determine which it is being invoked for. In particular, a parameter within the driver is available for specifying the character set (CAP). If lpcanon was compiled for a 64 character printer, then an initial editing of certain characters is done as follows.

Character	Mapped Into
a thru z	A thru Z
{	(-
}	)-
‘	’-
|	!-
~	^-

The overstruck characters are actually generated by invoking lpcanon for the first character of the overstrike, backing up the perceived column counter (see below) to give the impression of backspace, and then proceeding into the second editing phase with the second character of the overstrike.

Regardless of the printer's character set, all characters pass through a "second" editing phase (although, strictly speaking, it is really the first and only phase for 96 character printers). Three counters are maintained: a perceived column counter (ccc), that indicates into which column the next character should go; the actual column counter (mcc), that indicates into which column the next character will actually go (i.e., the column position of the printer itself); and a line counter (mlc), that keeps track of the number of lines printed on a page. Each character being printed has a different affect on these counters, as shown below.

Character	Mapping
Tab	Set perceived column counter to next tab stop (every eight columns).
New Page	Put "new page" character on output queue. Reset both column counters and the line counter to zero.
New Line	Put new line on output queue, unless the per page line limit has been reached. If so, change new line to new page character and put on output queue. Adjust the line counter as necessary and reset both column counters to zero.
CR	Reset perceived column counter to zero.
Backspace	Back up perceived column counter one position.
Blank	Advance perceived column counter one position.
All Others	If the perceived column exceeds the characters per line limit, merely add one to the perceived column counter and discard the character. Otherwise, adjust the perceived and actual counters as necessary to make them agree. (That is, if the perceived precedes the actual, place a carriage return and enough blanks on the output queue to align the counters. If the actual precedes the perceived, put enough blanks on the output queue to realign the counters.) Once the counters are aligned, the character is placed on the output queue and the perceived and actual column counters are incremented.

Notice that the notion of a perceived and an actual column counter permits the printing of overstruck characters and attempts to minimize the number of blanks being printed (lines with all blanks are translated to just a new line and trailing blanks on a line are not printed).

lpclose

CALL

lpclose(dev, flag)
int dev, flag;

RETURNS

No value returned.

SYNOPSIS

Logically closes the LP11 line printer.

DESCRIPTION

Lpclose logically closes the LP11 by causing the paper to be ejected to top-of-form and marking the device as closed (lp11.flag).

lpint

CALL

lpint()

RETURNS

No value returned.

SYNOPSIS

Interrupt handler for the LP11 line printer.

DESCRIPTION

As the interrupt handler for the LP11 printer, lpint must reactivate the printer for the next character in the output queue, if any. A second responsibility is to awaken lp.c/lpoutput if the output queue length has shrunk to an acceptable length, so that more characters may be placed on it.

lpopen

CALL

lpopen(dev, flag)
int dev, flag;

RETURNS

No value returned.

SYNOPSIS

Logically opens the LP11 line printer.

DESCRIPTION

The lpopen routine logically opens the LP11 line printer. In particular, it ensures that the device is not already being used by another user and that the printer itself is ready to be used (on line, power on, etc.). The device is not opened and an error is indicated (u_error) if one of these tests fail. If all is well, the device is enabled, marked as open (lp11.flag), and it is ensured that the paper is at top-of-form (or, at least, insofar as the driver is concerned).

lpoutput

CALL

lpoutput(c)
int c;

RETURNS

No value returned.

SYNOPSIS

Places a character on the LP11’s output queue and activates printing.

DESCRIPTION

Before placing the character "c" on the LP11’s output queue, lpoutput verifies that there are not already too many unprinted characters on the queue. If the queue is too long, it waits until the queue shrinks to an acceptable length (see lp.c/lpint). The character is then placed on the output queue and steps are taken to activate actual printing (see lp.c/lpstart).

lpstart

CALL

lpstart()

RETURNS

No value returned.

SYNOPSIS

Activates the LP11 line printer for a character.

DESCRIPTION

If the LP11 printer is ready to print a character and there are any characters to be printed in the output queue, the printer is passed the first character in the queue.

lpwrite

CALL

lpwrite()

RETURNS

No value returned.

SYNOPSIS

The write routine for the LP11 line printer.

DESCRIPTION

The lpwrite routine is the interface between the user writing to the LP11 line printer and the device driver itself. Therefore, it is called whenever the user issues a write request to this device. Lpwrite collects from the user's output buffer as many characters as necessary to satisfy the write request's byte count and causes them to be translated (see lp.c/lpcanon), as necessary, and placed on the LP11's output queue.

COMMON SYSTEMS

UNIX OPERATING SYSTEM

DEVICE DRIVERS SEC.2

This index lists the authorized issues of the sections that form a part of the current issue of this specification.

ISSUE 1 1/30/76

1. GENERAL

This document describes functions contained in pidents from PR-1C303-01 as follows:

MALLOC01	MEMORY ALLOCATOR
MEM01	CORE MEMORY
PARTAB01	TABLE OF TTY PARAMETERS
PC01	PC-11 PAPER TAPE READ/PUNCH INTERFACE
PIPE01	INTERPROCESS CHANNEL
RF01	RF11/RS11 FIXED HEAD DISK FILE
RH01	RH DEVICE INTERFACE
RK01	RK-11/11K03(05) DISK INTERFACE
RP01	RP-11/RP03 MOVING HEAD DISK INTERFACE
TC01	TC-11/TU56 DEC TAPE INTERFACE
TM01	TM-11/TU-10 MAG. TAPE INTERFACE
TTY01	GENERAL TYPEWRITER SUBROUTINES

2. PROGRAM CONVENTIONS

A. System calls are made with the first argument in register R0. When the system call is made, the contents of register R0 are moved to the per user control block (user.h) in the variable called u.u_R0. The remaining arguments of a system call are moved into the per user control block array u.u_arg (this means u.u_arg[0] is the second argument).

B. Arguments or results of executing some functions are often left behind in the per user control block. For example, nami.c/namei decodes a pathname into an inode pointer. In the process, a pointer to the inode of the parent directory is left in u.u_pdir. This means it is eady to make a directory entry for a file since the inode for the directory is available. (See the documented header user.h in PR-1C301.)

C. Inodes are always locked during manipulations to prevent simultaneous update by two processv:. The procedure is to always lock and increment the usage count of an inode even if it turns out that a user does not have access to that file. At the end of processing of the inode, the usage count is reduced by 1 if there was an error, and in either success or failure, the inode is unlocked.

D. Error processing that reflects errors back to the user are set in the per user control block error flag (u.u_error). These error conditions can be referenced by the user program through the external variable "errno". (See Section 2 of Programer’s Manual for list of error conditions.)

E. If I/O processing is to be done on a device, the particular driver for that device must be called. Devices are known by major and minor numbers stored in an inode. The system calls the particular device driver indirectly through the major device number. A block switch table and character switch table are defined at system generation time. The major device number is used as a displacement into this table and the appropriate routine is called. For example, the code:

```
(*bdevsw[maj].d_close)
```

will call the close entry point for the driver associated mith major device "maj".

malloc

CALL

malloc(mp, size)
int size, *mp;

RETURNS

Address of a memory/disk area of the desired size. Zero if requested amount of memory/disk space is not available.

SYNOPSIS

Malloc is used to obtain (allocate) space from either main memory or the swap device.

DESCRIPTION

The malloc and malioc.c/mfree functions are used to allocate and deallocate, respectively, space either in main memory or on the swap device using a first fit, low address first algorithm. Two tables of free space are maintained by the system, one for free core (the array coremapfl) and the other for free swap space (the array swapmapn). Each table entry contains the free area size, in 64 byte granularity for memory and 512 byte granularity for swap space, and the free area's starting address, expressed as either a memory segment number or disk block number. The entries within each table are ordered by ascending starting address.

When called, malloc does a linear search of the table pointed to by the first function argument ("mp") for the first entry whose siie is equal to or greater than the second argument ("size"). Because no conversion of "size" to the next 64 or 512 byte value is performed and because of the table updating requirement, the granularity of "size" must already be in the proper units. When a free area of sufficient size is found, the size and starting address of the table entry are adjusted to reflect the allocation of the requested space by decreasing the size and increasing the starting address. In the case where the requested "size" was equal to the size of the free area available (i.e. the updated size value is zero), that entry in the table is eliminated and all of the following entries are moved up. The starting address of the allocated space (either a memory segment number or a disk block address) is returned. A zero is returned in the case that all free areas were smaller than the requested size.

mfree

CALL

mfree(mp, size, addr)
int size, addr, *mp;

RETURNS

No value returned.

SYNOPSIS

Mfree is used to free (deallocate) space from either main memory or the swap device.

DESCRIPTION

Mfree is the complementary function of malloc.c/malloc, in that it frees (deallocates) the memory/swap area of "size" that begins at "addr". In deallocating memory, both "size" and "addr" are expressed in memory management units (64 byte), whereas for deallocation of swap space, "size" is in 512 byte units and "addr" is a disk block number. See the description of malloc for an explanation of the tables used to maintain a mapping of free space in memory and on the swap device.

A linear search of the free space table pointed to by the first argument ("mp") is made to find the first entry Ei whose starting address is greater than that of the area being freed ("addr"). If there is no such entry, then Ei is the first unused slot in the table. The space being freed (the new free area) is then entered in the table (deallocated) in one of four ways.

1. If the free areas described by entries Ei-1 and Ei do not abut the area being freed, then entry Ei and all following entries are moved down in the table and the entry for the new free area is inserted in the ith slot.
2. If the preceding entry Ei-1 abuts the new free area, then "size" is added to the size of entry Ei-1.
3. If the entry Ei abuts the new free area, then its size is increased and its starting address decreased by "size".
4. If the new free area abuts both Ei-1 and Ei (i.e. the area being freed is the hole between these two free areas), the size of Ei and "size" are both added to the size of Ei-1. Entry Ei is then removed and all following entries are moved up in the table.

mmread

CALL

mmread(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Read routine for the memory device driver.

DESCRIPTION

Because of the hardware enforced segmentation of physical memory into virtual address spaces, it is not possible for a user process to directly reference any memory locations outside its own address space. The memory device driver mem.c, which consists of mmread and rnmwrite, provides the capability to make such references. In particular, once entries for mmread and mmwrite are placed in the configuration table conf.c (in the cdevswn table, since these two routines emulate raw mode device driver routines) and the appropriate special filei have been created, the user is free to treat physical memory as if it were a regular file (read, write, seek, open, close). Of course, the usual restrictions regarding file access permissions (for the memory special files) still apply. The minor device numbers recognized by this memory device driver are as follows.

0 All addresses (i.e., file offsets) are interpreted as physical memory addresses.

1 All addresses (i.e., file offsets) are interpreted as kernel space addresses.

2 All read requests result in a no bytes read condition; this emulates a zero length file. Although all write requests appear to complete, no writing is ever done; this is the personification of the "bit bucket".

On any request to read minor device two (low byte of "dev"), mmread does an immediate return, thereby indicating to the I/O subsystem an unfulfilled read request (end-of-file).

For minor devices zero and one it is first necessary to compute the memory segment number (i.e., 64 byte boundary) of the segment containing the physical or kernel address, respectively, to be read. When this value has been placed into user space memory management register zero, a character (byte) may be read (see mch.s/fubyte). After restoring the user space memory management register to its original contents, the character is passed to the user's I/O buffer. This algorithm continues until either there is an error or the byte count of the read request is satisfied.

mmwrite

CALL

mmwrite(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Write routine for the memory device driver..

DESCRIPTION

See mem.c/mmread for an overview of the memory device driver, of which the mmwrite routine is the write portion.

For all write requests to minor device two (low byte of "dev"), the user's I/O request parameters (u_count, u_base, and u_offset) are modified to reflect the completion of a write request, but no data is ever actually written (i.e., the "bit bucket").

For minor devices zero and one, a character is retrieved from the user's I/O buffer. The memory segment number (i.e., 64 byte boundary) of the segment containing the physical or kernel address, respectively, to be written is computed. After this value is placed in user space memory managempnt register zero, the character may be written (see mch.s/subyte) to the desired location in memory. The user's memory management register is then restored to its original contents. This algorithm continues until either there is an error or the byte count of the write request is satisfied.

partab

CALL

None

RETURNS

No value returned.

SYNOPSIS

Translation Table to distinguish legal and illegal ASCII characters.

DESCRIPTION

Several character drivers and tty.c use partab to differentiate map ASCII characters into special functions. The table is made up of translation values with 0200 ORed in appropriately to provide the parity bit to the seven bit ASCII character. The lower seven bits have the following meaning:

0	regular character
1	non-printing character
2	backspace
3	newline
4	horizontal tab
5	vertical tab
6	carriage return

pcclose

CALL

pcclose(dev, flag)
int dev, flag;

RETURNS

No value returned.

SYNOPSIS

Logically closes the PC11 paper tape reader/punch.

DESCRIPTION

If the PC11 was open for writing ("flag" nonzero), pcclose logically closes it by punching a leader in the tape (see pc.c/pcleader). However, if the PC11 was being used for reading ("flag" = 0), then all characters that were received from the device but not yet read by the user are flushed from the input queue (pcin), the reader is disabled, and the device's state (pcstate) is changed to indicate that the reader may now be used by other users.

pcleader

CALL

pcleader()

RETURNS

No value returned.

SYNOPSIS

Punches a leader on paper tape.

DESCRIPTION

Whenever paper tape is used for output, pcleader is invoked at the time the PC11 paper tape reader/punch is logically opened (see pc.c/pcopen) or closed (see pc.c/pcclose). This routine merely punches a leader of a hundred zero characters.

pcopen

CALL

pcopen(dev, flag)
int dev, flag;

RETURNS

No value returned.

SYNOPSIS

Logically opens the PC11 paper tape reader/punch for input or output.

DESCRIPTION

The pcopen routine logically opens the PC11 paper tape reader/punch. The specific actions that are taken depend on whether the device is being opened for reading ("flag" = 0) or writing ("flag" nonzero).

If the PC11 is being opened for reading, pcopen verifies that the device is not already open for reading by some other user (pcstate). If it is, pcopen returns with an error (u error), since allowing multiple users to read the_same tape will usually prove unacceptable to all of them. Otherwise, the device's state (pcstate) is set to WAITING and pcopen waits for the state to change. A change of state signifies that a character has been actually been received (see pc.c/pcrint); this ploy ensures that there is tape in the reader and that it is online.

If the PC11 is opened for writing, pcopen merely punches a leader in the tape (see pc.c/pcleader).

pcoutput

CALL

pcoutput(c)
int c;

RETURNS

No value returned.

SYNOPSIS

Places a character on the PC11's output queue and activates punching.

DESCRIPTION

Before placing the character "c" on the PC11's output queue (pcout), pcoutput must verify that no device errors have occurred punching any previous characters. If an error did occur, pcoutput indicates that fact (u_error) and returns, as any

further attempts to punch would be futile. In the absence of errors, this routine places the character "c" on the output queue and activates the punch (sec pc.c/pcstart). Note that if the queue is too long, pcoutput may have to wait until it shrinks to an acceptable length (see pc.c/pcpint) before placing the character on it.

pcpint

CALL

pcpint()

RETURNS

No value returned.

SYNOPSIS

Interrupt handler for the PC11 paper tape punch.

DESCRIPTION

As the interrupt handler for the PC11 paper tape punch, pcpint must reactiviate the punch for the next character in the output queue (pcout), if any. It is also responsible for awakening pc.c/pcoutput if the output queue length has shrunk to a reasonable length, so that more characters may be placed on the queue.

pcread

CALL

pcread()

RETURNS

No value returned.

SYNOPSIS

The read routine for the PC paper tape reader/punch.

DESCRIPTION

The pcread routine is the interface between the user reading from the PC11 device and the device driver itself. As such, it is called whenever the user issues a read request to this device. Pcread attempts to pass to the user's buffer as many characters from the input queue (pcin) as are necessary to satisfy the read request's byte count. If there are not enough characters available on this queue, pcread activates the PC reader, waits for more characters to be received and placed on the input queue (see pc.c/pcrint), and then continues to pass characters to the user. Pcread will return once the user's request is satisfied; however, it will return prematurely if an end-of-file is sensed on the PC11 reader (EOF in pcstate, see pc.c/pcrint) or an error occurs while passing characters to the user's buffer.

pcrint

CALL

pcrint()

RETURNS

No value returned.

SYNOPSIS

Interrupt handler for the PC11 paper tape reader.

DESCRIPTION

The pcrint routine receives control when interrupts are received from the PC11 reader. If the state of the reader (pcstate) is WAITING and a character was successfully received from the PC11, the state is changed to READING (see pc.c/pcopen for an explanation of the significance of this). This character is then handled as is any other chSracter that is read when the PC is in the READING state (see below).

When an interrupt is received (i.e., a character is received) with the PC11 in the READING state, perint checks to see if an error occurred. Any error is interpreted as end-of-file and the device's state is changed to reflect this fact. Correctly received characters are placed on the input queue (pcin) and the reader is reactiviated if there are not already too many characters on the input queue. Whether the interrupt was from a properly received character or an end-of-file (i.e., an error), pc.c/pcread is awakened in case it was waiting for more characters to be received and placed on the input queue.

pcstart

CALL

pcstart()

RETURNS

No value returned.

SYNOPSIS

Activates the PC11 paper tape punch for a charac-

ter.

DESCRIPTION

If the PC11 punch is ready to punch a character and there are any characters in the output queue (pcout), then the punch is passed the first character in the queue.

pcwrite

CALL

pcwrite()

RETURNS

No value returned.

SYNOPSIS

The write routine for the PC11 paper tape reader/punch.

DESCRIPTION

The pcwrite routine is the interface between the user writing to the PC11 device and the device driver itself. As such, it is called whenever the user issues a write request to this device. Pcwrite collects as many characters from the user's output buffer as necessary to satisfy the write request's byte count and causes them to be placed on the PC11's output queue (pcout; see also pc.c/pcoutput).

(Section missing from the original material)

rfintr

CALL

rfintr()

RETURNS

No value returned.

SYNOPSIS

Handles interrupts from the RF11.

DESCRIPTION

Rfintr, as the interrupt handler for the RF11 disk, performs the functions described in rp.c/rpintr.

rfread

CALL

rfread(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Interface to RF11 driver for "raw" mode read requests.

DESCRIPTION

Rfread handles raw mode read requests to the RF11 disk by calling bio.c/physio. See rp.c/rpread for a discussion of raw mode device read routines.

rfstart

CALL

rfstart()

RETURNS

No value returned.

SYNOPSIS

Initiates the actual I/O to an RF11 device.

DESCRIPTION

If there are any I/O requests on the RF11 queue (chained from d_actf in rftab), rfstart marks the RF11 as active (rftab.d_active). After filling in the disk address extension error register, rfstart initiates I/O for the first request in the queue by invoking bio.c/devstart. The arguments to devstart reflect the logical device policy of the driver (see rf.c/rfstrategy), as they are the translation of a logical device block number to a physical device address.

rfstrategy

CALL

rfstrategy(bp)
struct bur *bp;

RETURNS

No value returned.

SYNOPSIS

Places an I/O buffer on the RF11’s queue of I/O buffers to read/write.

DESCRIPTION

Rfstrategy performs for the RF11 disk the general strategy functions described in rp.c/rpstrategy. The I/O buffer queuing strategy employed is strictly First In First Out (FIFO). The logical device policy of the driver is that the logical device number (minor device) is taken to be that value plus one physical devices. For example, RF logical device two is viewed as a single device consisting of the concatenation of the three physical drives 0, 1, and 2. The actual logical device block number to physical device address translation is performed by, rf.c/rfstart.

rfwrite

CALL

rfwrite(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Interface to RF11 driver for "raw" mode write requests.

DESCRIPTION

Rfwrite handles raw mode write requests to the RF11 disk by calling bio.c/physio. See rp.c/rpwrite for a diqcussion of raw mode device write routines.

(Section missing from the original material)

(Page missing from the original material)

2. If greater than 7, say, 7+n, then it is the block interleaving of the first n RK05 drives. Thus, logical device 9 is the interleaving of blocks on drives 0 and 1.

The actual logical block number to physical device address translation is performed by rk.c/rkaddr.

rkwrite

CALL

rkwrite(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Interface to RK05 driver for "raw" mode write requests.

DESCRIPTION

Rkwrite handles raw mode write requests to the RK05 disk by calling bio.c/physio. See rp.c/rpwrite for a discussion of raw mode device write routines.

rpintr

CALL

rpintr()

RETURNS

No value returned.

SYNOPSIS

Handles interrupts from the RP03.

DESCRIPTION

The block device driver interrupt handling routines receive control when an interrupt from their respective devices occurs. All of these routines perform the same basic functions.

1. Mark the device as inactive (d_active in the device's devtab).
2. Check to see if an error occurred on the transfer. If so, certain actions unique to that particular device may need to be taken (e.g., reset the controller). The transfer is usually then reinitiated and is reggrded as incomplete. An I/O request may be retried up to ten times before being abandoned as hopeless (B_ERROR in the buffer header's b_flags).
3. Mark completed requests as such. This is done both to requests that have completed successfully and to those that have been deemed hopeless because of I/O errors.
4. Remove the completed requests from the device's I/O queue and initiate I/O for the next request in the queue.

rpphys

CALL

rpphys(dev)
int dev;

RETURNS

Zero if a "raw" I/O request will exceed the logical device's upper bound; one if it remains in bounds.

SYNOPSIS

Verifies that a raw mode I/O request to an RP03 logical device will remain within the upper bound of that device.

DESCRIPTION

Because I/O requests to RP03s are made to a logical (i.e., minor) device and there are many logical devices on a single physical device, it is necessary to impose software checks to ensure that a raw mode I/O request can be fulfilled within the confines of the logical device. Whereas rp.c/rpstrategy ensures that the starting block address is valid, rpphys verifies that the number of bytes to be transferred will not cause the transfer to overrun the end of the logical device. A one is returned for a valid 110 request, otherwise, zero.

rpread

CALL

rpread(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Interface to RP03 driver for "raw" mode read requests.

DESCRIPTION

The read routines for disk and ,tape drivers are the functions that handle "raw" mode read requests for their respective devices. That is, they are the interface between users making such requests and the I/O subsystem, and are called whenever a read is done to the raw device. They usually do little more than invoke bio.c/physio to do the real work involved with "raw" I/O, but are vitally necessary, as they inform physio of such things as the device strategy routine to invoke to make the I/O request.

Rpread merely veriies (see rp.c/rpphys) that the "raw" read request will remain entirely within the bounds of the RP03 logical device in question before calling physio.

rpstart

CALL

rpstart()

RETURNS

No value returned.

SYNOPSIS

Initiates the actual I/O to an RP03 device.

DESCRIPTION

If there are any I/O requests on the RP03 queue (chained from d_actf in rptab), restart marks the RP03 as active (rptab.d_active) and initiates I/O for the first request in the queue. The real work of activation is done by bio.c/devstart, but rpstart must fill in an RP03 controller register that is not filled in by devstart. Although the physical block address has already been computed by rp.c/rpstrategy, rpstart must complete the logical to physical device mapping by selecting the actual physical drive.

rpstrategy

CALL

rpstrategy(bp)
struct buf *bp;

RETURNS

No value returned.

SYNOPSIS

Places an I/O buffer on the RP03's queue of I/O buffers to read/write.

DESCRIPTION

The strategy routines for disk and tape drivers provide two major services: to place I/O requests on the device's queue of pending requests in an order that is most efficient for that particular device, and, to verify that the request's logical block address conforms to the logical (i.e., minor) device policy of the device driver. In particular, disk and tape strategy routines do the following specific things for each 110 request, which the driver sees in terms of a pointer "bp" to a buffer header (struct buf):

1. Verify that the block address given in the I/O request is a plausible address for the logical device being read/written. That is, ensure conformity to the logical device policy of the driver.
2. For devices that have several logical devices on a single physical device, translate the block address on the logical (i.e., minor) device to a true block address on a physical device. The remainder of the translation (selecting the physical drive) is peforuied elsewhere, usually in the driver's start routine. Note that for drivers that map a logical device to one or more physical devices this step is omitted and the address translation is done elsewhere, usually in the driver's start routine.
3. Place the I/O request in the device's queue of pending 110 requests (work to do queue). The location within the queue where the request is placed depends on the queuing strategy being employed for the device. The queue itself is chained from d_actf in the device's devtab (e.g., rptab, rftab, etc.) Immediately prior to rechaining the queue to insert the request, the processor's hardware priority must be raised to that of the device to disable interrupts from the device.
4. Cause physical I/O to be initiated if there are no previous requests currently being serviced.

The strategy routine for dig RP03 disk, rpstrategy, performs all of the above functions. The logical to physical deVice mapping is accomplished by dividing the logical (i.e., the minor) device number by eight. The quotient of this division is interpreted as the controller drive number and the remainder is the logical file system on that drive. For example, a minor device number of 13 is construed to be logical file system five on physical drive one. Each physical drive is divided into eight logical file systems as follows:

File #	Cylinders	# of blocks
0	0 - 202	40600
1	203 - 405	40600
2	0 - 45	9200
3	360 - 405	9200
4	0 - 327	65535
5	78 - 405	65535
6	unused	
7	unused	

The queuing strategy used is the "elevator" technique. That is, when a request is made, if there are none or one other request on the queue, the new request is placed at the end of the queue (First In First Out (FIFO) strategy). If there are already two or more pending requests, the new request is inserted so that all requests on the queue are in

cylinder number order. Whether this order is increasing or decreasing by cylinder number is established when there are two re, quests on the queue. All requests for the same cylinder are handled in a FIFO manner.

rpwrite

CALL

rpwrite(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Interface to RP03 driver for "raw" mode write requests.

DESCRIPTION

The write routines for disk and tape drivers are the functions that handle "raw" mode write requests for their respective devices. That is, they are the interface between users making such requests and the I/O subsystem, and are called whenever a write is done to the raw device. They usually do little more than invoke bio.c/physio to do the real work involved with "raw" I/O, but are vitally necessary, as they inform physio of such things as the device strategy routine to invoke to make the I/O request.

Rpwrite merely verifies (see rp.c/rpphys) that the "raw" write request will remain entirely within the bounds of the RP03 logical device in question before calling physio.

tcclose

CALL

tcclose(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Logically closes the TC11 DECtape.

DESCRIPTION

Tcclose is the device close routine for DECtape. It accomplishes its function by merely forcing all of the write behind buffers for the device "dev" to be written (see bio.c/bflush).

tcintr

CALL

tcintr()

RETURNS

No value returned.

SYNOPSIS

Handles interrupts from the TC11 DECtape.

DESCRIPTION

As the interrupt handler for DECtape, tcintr's first responsibility is to determine if an error occurrred on the controller command. If this is the case and it is the twentieth error to occur for the current DECtape I/O request, then the I/O is marked as complete but in error and abandoned as hopeless. Otherwise, the error is handled by issuing a read block number command for the block for which the transfer failed. This necessitates reversing the tape's direction, which in essence means that the block search is reinitiated for the block number of the I/O request. Tcintr then returns, as it will receive control again when the read block number command completes and causes an interrupt.

In most cases, however, an error will not occur. Tcintr continues the block search that was initiated by tc.c/tcstart, or, as described above, by tcintr itself. This entails merely comparing the current block number with that of the current pending I/O request, and issuing a read block number command for the forward or reverse direction as appropriate. Thus, the tape is advanced in the proper direction to locate the desired block. Since a DECtape interrupt is gencrated anew for each read block number command, it is necessary for tcintr to return after each such command to prevent recursive stacking of tcintr calls.

When the block number of the desired block is encountered while advancing the tape in the forward direction, a command to actually read or write the user's data is issued and tcintr returns. If the tape block search is proceeding in the reverse direction, then the search is actually made for the desired block number minus three. When this block is encountered, the tape direction is reversed (to the forward direction) and the search resumed. Of course, the desired block is found almost immediately and the I/O is performed as previously described. This three block tactic is vital to ensure that the tape is up to speed in the forward direction before attempting the actual I/O.

When tcintr receives control after a completed read or write, the I/O request is marked as completed and, if there are any requests remaining on the queue, tc.c/tcstart is called to initiate I/O for the next request in the queue.

tcstart

CALL

tcstart()

RETURNS

No value returned.

SYNOPSIS

Initiates the actual I/O procedure for TC11 DECtape.

DESCRIPTION

If there are any I/O requests on the TC11 queue (chained from d_actf in tctab), tcstart initiates the actual I/O procedure for the first request in the queue. Because of the random block access property of DECtape, it is first necessary to determine the tape's current position. Tcstart merely initiates this process; the-remainder of the positioning to the proper tape block is carried out by tc.c/tcintr.

If the last commmand to the DECtape controller was not to the same logical device as the current I/O request, a command to stop all transports is first issued. Note that for DECtape, the logical

device number is interpreted as the physical drive number. The controller is then issued the read block number command for the drive the current I/O request is for. This command is usually for the forward direction, but if the drive is not up to speed (i.e., the stop all transports command was previously issued), it is for the reverse direction. In any event, the command is always issued with interrupts enabled, so that upon command completion the DECtape interrupt handler tc.c/tcintr receives control and evaluates the results.

tcstrategy

CALL

tcstrategy (bp)
struct buf *bp;

RETURNS

No value returned.

SYNOPSIS

Places an I/O buffer on the TC11's queue of I/O buffers to read/write.

DESCRIPTION

Tcstrategy performs for DECtape the general strategy functions described in rp.c/rpstrategy. The I/O buffer queuing strategy employed is strictly First In First Out (FIFO). As might be expected, the logical device number is taken to be the physical drive number.

(Page missing from the original material)

(Page missing from the original material)

The block address verification for magnetic tape is, because of the medium's sequential nature and the attempt to have it emulate a disk file system, more complex than that of most strategy routines and therefore merits elaboration. A block number counter (t_nxrec[]) is maintained for each drive. An I/O request may not be initiated to a block number exceeding the value of the drive's counter. Specifically, when the device is opened (see tm.c/tmopen), the value of this counter is set to the size of the largest permissible file (65535 blocks). As long as only read requests are made, the value of the counter does not change. However, whenever a write request is made, the counter's value is set to the requested block number plus one (which will be, for almost all cases, the number of the block that will be written next). For every read or write request, the block number of the request is checked against the counter. If the request's block number exceeds the counter value, then the request is marked as complete but in error (u_error). Otherwise the request is considered valid. A special case is the situation where a read request is made for a block number that equals the counter's value. This could occur, for example, if records are being read and written and a read is issued for block n + 1 immediately after writing block n. In this case no actual I/O takes place, but the user's I/O buffer is zeroed and the I/O request is marked as completed.

tmwrite

CALL

tmwrite(dev)
int dev;

RETURNS

No value returned.

SYNOPSIS

Interface to TM11 driver for "raw" mode write requests.

DESCRIPTION

Tmwrite handles raw mode write requests to TM11 magnetic tape. See rp.c/rpwrite for a discussion of raw mode write routines. Before calling bio.c/physio to do the real work, the starting block number of the request must be calculated tm.c/tmphys.

canon

CALL

canon(atp)
struct tty *atp;

RETURN

An indication of whether any characters have been transferred from an teletype input queue to a teletype canonical queue is returned.

SYNOPSIS

This is the Canonicalizer. It translates a line of input into a standard Form (called Canonical Form) and performs erase-kill processing,

DESCRIPTION

Basically, a teletype may select one of two processing modes; line at a time processing or character at a time processing (raw character I/O). The difference between the two as far as the response seen by a user is that for line at a time processing, a read of a teletype does not return until a whole line of input is accumulated while for character at a time processing, a read returns one character, regardless of whether a whole line has been received or not. Erase-kill processing is performed for line at a time processing but not for character at a time processing, and special characters, such as *quit*, *interrupt* and *EOT* lose their meaning in raw mode

The input queue (or raw queue "t_rawq") contains delimiters to mark off the amount of input that is to be examined by tty.c/canon. The delimiter used is 0377 (octal). For character at a time processing, the delimiter is placed after each character, while for line at a time processing the delimiter is placed after each line feed or carriage return (by tty.c/ttyinput). Tty.c/canon is called by tty.c/ttread to read a teletype and put one delimited string of input in Canonical Form so that it can be transferred to the user process. Processing of a delimited string is handled as follows:

1. A check is made to see if any delimited string has bee’h accumulated from the teletype. If none has been accumulated, tty.c/canon roadblocks the process that requested the Canonicalization of input. The process is roadblocked at priority TTIPRI and remains roadblocked until a delimited input string has been accumulated. The tty.c/ttyinput function awakens a process when a delimited string has been received. Before roadblocking the process, a check is made to see if carrier has been dropped on that teletype by checking the "t_state" flag in the associated teletype structure. The receive interrupt handler for the individual line interface drivers can detect whether carrier has been dropped and will set an indicator (CARRIER) in the teletype structure(in "t_state"). To prevent the status of the teletype from changing while the check is made, interrupts from character devices are locked out by setting the processors priority to 5.
2. Characters are transferred one at a time to the Canonical Buffer "canonb" until one of the following occurs,

   a. There are no more characters on the input queue("t_rawq").

   b. The first delimiter(0377) is reached.

   c. The size of the Canonical Buffer has been exceeded (currently 256 characters).
3. As the characters are transferred to the Canonical Buffer, they are put into Canonical Form and erase-kill processing is performed. The following transformations are made:

   a. If a teletype is set for upper case only mapping, then the following transformations are made

Input (2 Characters)	Canonical Form (1 character)
\’	’
\!	|
\^	^
\\	\
\(	{
\)	}
\lower case alphabetic	upper case alphabetic

   b. The character ’#’ erases the previous character from the input stream. No character before the first character on a line may be deleted.

   c. The character ’@’ kills an entire line of input. All of the input up to and including the @ is deleted from the input string. No lines before the line in which the ’@’ appears can be deleted.
4. As characters are placed in the Canonical Buffer ("canonb"), they are deleted from the input queue ("t_rawq"). Once a string has been placed in Canonical Form and erase-kill

processing performed, the canonicalized string is transferred to the canonical queue ("t_canq") associated with the teletype and an indication that processing has been completed is returned to the calling routine (tty.c/ttread).

cinit

CALL

cinit()

RETURNS

No value is returned.

SYNOPSIS

The character buffers ("dist") are initialized.

DESCRIPTION

The character buffering scheme used in the UNIX Operating System utilizes a pool of six byte buffers that are available to all of the character devices on the system. The buffers are organized as a linked list ("clist") and each six byte buffer contains one extra word which is used as a pointer for queuing a buffer on a device. One hundred buffers are normally allocated for character storage. A globally known pointer ("cfreelist") contains a pointer to the first free buffer. Each free buffer contains a pointer to the next buffer and the last on the "cfreelist" queue contains a zero in the pointer entry.

After UNIX is booted into memory the "dist" is initialized as part of the startup procedure. (Main.c/main calls it.)

In order to simplify the work that must be done by the assembly language functions mch.s/putc and mch.s/getc in allocating and deallocating character buffers, a trick is used in initializing the storage that is used for the "clist" buffers. The method forces the allocation of each character buffer to occur on an eight byte memory address even though the buffer definition may not occur on an eight byte boundary. This means that even though the overall storage called for is defined as one hundred character buffers, only ninety-nine may be present because of the boundary adjustment.

Another function performed by tty.c/cinit is to determine how many character devices there are on the system so that higher level functions may check major device numbers to insure that they are within range when accessing the Character Device Switch Table. The global variable "nchrdev" is set to the number of character devices (i.e., effectively the maximum value that the major device number may be for a character device).

flushtty

CALL

flushtty(atp)
struct tty *atp;

RETURN

No value is returned.

SYNOPSIS

The input queue, canonical queue and output queue associated with a particular character device is emptied and the buffer storage returned to the freelist.

DESCRIPTION

There are a number of reasons why the queues associated with a character idevice should be flushed out. For character devices which are connected to a computer over common carrier lines, the possibility exists that the connection may be broken at any time so that outputting characteri to that teletype is impossible. Characters accumulated in the input ("t_rawq"), canonical ("t_canq") and output ("t_outq") queues would then be a stranded. They would be queued on a character device which would no longer be able to accept output or stimulate input processing and thus empty their queues. The tty.c/flushtty ftinction allows the interrupt handlers associated with each line interface driver to clean out the queues if they detect that the connection has been broken (carrier dropped).

Flushing everything in the queues associated with one teletype is also done when some process closes a teletype. This must be done so that if input is accumulated before the close is issued, the characters in the input queue will not be stranded. A teletype’s queues are also flushed when either of the special characters quit (delete) or interrupt (control FS) are received from a teletype. These characters have a special meaning to the system when the teletype is set up for line at a time processing. They indicate to the system that processes which are controlled by the teletype issuing the quit or interrupt should be terminated (unless they have made other arrangement via the signal system call). In this case it is desirable to flush the output to the controlling teletype as well

as input from the teletype so that output from the terminated program is not printed and any commands that have not yet been acted upon can be dropped.

Flushing out of the three queues is a delicate operation. Both the canonical ("t_canq") and output queues ("t_outq") may be flushed without any precautions, however, all processes waiting for input from the teletype or waiting to do output to the teletype must be awakened to insure that there will be no processes hung in the system waiting for I/O that can never occur. In order to flush the input queue, interrupts from character teletypes must be locked out so that a race between flushing and accumulating characters does not occur. Once the input queue is flushed, the delimiter count ("t_delct") is zeroed to indicate that there are no lines of input in the raw queue.

gtty

CALL

gtty()

RETURNS

The tty.c/gtty routine does not explicitly return a value however, a three word array containing the speeds and mode of a teletype is returned as a result of calling it.

SYNOPSIS

Implements the gtty system call. Obtains the modes and line speed of a teletype.

DESCRIPTION

This function performs the inverse operation of the tty.c/stty function, that is, it returns to a process the line speed and modes for a given character device. The three word array that is returned to the process making the gtty system call is as described under tty.c/stty. Both tty.c/gtty and tty.c/stty use the tty.c/sgtty function as a common subroutine to interface to the sgtty function associated with a character device.

sgtty

CALL

```
sgtty(vector)
struct {
   int v[3];
} *vector;
```

RETURNS

No values are returned

SYNOPSIS

interfaces tty.c/stty and tty.c/gtty to the appropriate character device sgtty function (dh.c/dhsgtty, dc.c/dcsgtty, etc.).

DESCRIPTION

Since the functions of the stty and gtty system calls are similar, a common subroutine is used to interface them to the character device sgtty functions (dh.c/dhsgtty, dc.c/dcsgtty, etc.). The argument "vector" is either the address of a three word array within the system in the case of tty.c/gtty (so that the device characteristics may be returned in the array) or in the case of tty.c/stty, "vector" is zero. The pointer "vector" is passed to the character device sgtty function and is used to determine whether the existing characteristics are to be returned or whether new characteristics are to be set up.

As part of the stty or gtty system call, the file descriptor and hence the device number for the character device is passed. The calls have the following form,

```
stty(fildes, arg)
int fildes;
struct{
   char ispeed, ospeed;
   int unussd;
   int mode;
} *arg;
```

and

```
gtty(fildes, arg)
```

The file descriptor is passed in register R0 when the system call is made and the tty.c/sgtty function must obtain it from the stack frame so that it can be used to find the major and minor device number of the device. The function fio.c/getf is used to get the i-node associated with this file descriptor. In order to protect the devices on the system or the system itself from a bad file descriptor passed in the system call, tty.c/sgtty

(Page missing from the original material)