MagicSix Programmer's Logic Manual.

Why it works!
Why is doesn't work!

This manual is divided into various sections, each describing either an important logical section of the system, or describing the contents of a binder input segment, if no better logical division is deemed possible.

The sections are: 1) ns: The low-level interrupt handlers and other low-level routines. 2) ns2: More of the same, assembled in a different module. 3) rim: The core shuffler. 4) disk: The disk routines and some user callable core shuffler interface

routines, such as hcs$terminate. 5) switch: The address space switcher. 6) fs2: The buffer managers. 7) fs3: More of the same, and vtoc-level file system operations. 8) fsi: Initiate routines. 9) fsn: File creation transfer vectors to cfile and fsn segments. 10) fsr: File deletion transfer vector to segment fsr. 11) fss: File system assembly routines. 12) fst: Low level buffer manager stuff.

In addition, there is a discussion at the beginning of this manual of various important concepts that do not fall into a small number of the above sections, such as the locking strategies used by the system. These are as follows:

1) General file system organization. 2) Locking strategies used by the system. 3) Address space switching.

Chapter 1: Preliminaries.

Section 1: General file system organization.

The file system is a Multics-like tree structured file system containing at the lowest level, a group of segments. Certain segments are actually directories and contain information describing other segments. For each segment that exists on the disk, there is a file map describing the pages allocated to the file. The way that this is done is by having a volume table of contents, or a vtoc, which is just a large file. Every 64 bytes describes a new file, and the 64 byte long file maps are called vtoces. Each vtoce is of one of three types. The first type is an allocation map for vtoces, and this type consists of a simple 512 bit array, one bit for each vtoce. As could be expected, these allocation vtoces occur every 512 vtoces, at numbers 0, 512, 1024, etc. They always have bit 0 (the first bit) on so that each

allocation vtoce marks itself as in use. Thus vtoce 0 is an allocation map for the first 512 (0-511) vtoces in the vtoc. Vtoce 1 is the file map for the vtoc itself, so that it may be dynamically grown after the file system is created. Vtoce 2 is the file map for the disk page allocation map, which is simply a large bit array, one bit for each page on the disk, 1 meaning used and 0 meaning a free block. Vtoce 3 is the file map for the root directory. To start things off, the convention is made that the first page (page 0) of vtoce index 0, (the vtoc) is at page 0 of the disk. This enables the file system to find any vtoce simply by dividing 64*index by 2048, getting the page number, relative to the start of the vtoc, of the page containing the vtoce being searched for, and then computing the offset from the remainder of the division. The second type of vtoce is the initial file map vtoce, which describes a file by giving its length, bit count, and (perhaps part of) its position on the disk. The third type of vtoce is the continuation of the above if the description does not fit in one 64 byte vtoce.

The format of a file map vtoce is as follows:

```
1 filemap based unaligned,
        2 type fix(7), /* 1 for vtoce's of the second type, three for extensions */
        2 pad bit(24), /* garbage to force alignment of the following*/
        2 date_time_modified bit(64), /* updated on swap out through
hcs$core_to_index */
        2 maxents fix(7), /* maximum number of data descriptors that fit in this
vtoce */
        2 curents fix(7), /* same as above, only the number currently in use */
        2 extension fixed(15), /* next vtoce, or zero if this is the last */
        2 data(8),
              3 runlen fix(7), /* number of pages starting at...*/
              3 daddr fix(23), /* this page number*/
        2 bit_count fix(31), /* the segments bit count */
        2 length fix(31), /* the number of pages allocated to this file */
        2 lock fix(31), /* obsolete.*/
        2 last_index fix(15), /* the index number of the last vtoce describing this
file */
```

If this is a continuation vtoce, then filemap.data is dimensioned to be 11 long instead of 8 and everything after filemap.data does not exist.

Here is a description of the fields: type: This is 1 for the first vtoce describing a particular file, and 3 if this is a continuation for extra-long file maps.

pad: This is total trash.

date_time_modified: This is the date and time, in hcs$read_clock format, of the last time that the segments was written through to the disk. This occurs shortly after terminating a segment that was modified, if you are the last user using it. The update is done by hcs$core_to_index when the core shuffler gets a write through request telling it to write a segment through to the disk, but to leave it in core also, in case it is needed again shortly.

maxents: This gives the value of the dimension of filemap.data in this particular vtoce.

curents: This gives the number of the entries in filemap.data that are currently in use by the file.

data: This array describes where the file is actually situated on the disk. It is basically an array of pointers and lengths, since files are often contiguous, telling where each page or group of pages is located. Filemap.data.daddr is the page number of the first page in the block being described and filemap.data.runlen is the number of pages in the block, including the first one, i.e. the values
02000007
03000033
in a two element vtoce indicates that pages 7,8,33,34,35 are allocated to the file in that order.

bit_count: This is the bit count of the segment.

length: This is redundant information, but is simpler to get at then calculating the length of a file by counting the number of pages allocated in filemap.data in the current vtoce and in all of its extensions. Note that the length field counts all of the pages in the extensions, also.

lock: Obsolete.

last_index: A vtoce index (starting from 0) of the last vtoce describing this file. For a vtoce without extension vtoces, it is the index of the only vtoce describing the file.

Note that a vtoce index is a number between 0 and 65576 which tells which 64 byte vtoce in the vtoc is being referenced.

Thus to swap a segment in or out, all that is needed is the segment's vtoce index and a pointer to the segment. The core shuffler deals exclusively with vtoc indexes and never with directories.

It should be obvious that an extension vtoce is only added when the disk page allocation is so fragmented that there are more than eleven different contiguous areas of the disk belonging to the file in question.

At this level, the file system looks like a simple collection of files without any structure, names or properties. All of this information is contained in the containing directory of a segment. Each directory contains a threaded list of all the segments contained in it, and also information on their names, accesses, address space names, and also a bit if the segment is actually a directory. Directories are simply ring 1 segments that are writable from ring 0 only, except that they have a bit set in the containing directory so that certain operations can not be performed on them, such as deletion without deleting their contents, truncation and generally writing upon.

The structure of a directory is as follows. There are a number of important blocks.

```
1 directory based,
        2 first_fcb offset(dirarea),
        2 dlock bit(32),
        2 cst(32) offset(dirarea),
        2 dirarea area (30000),

1 fcb based,
        2 next_fcb offset,
        2 last_fcb offset,
        2 uid bit(64),
        2 flags,
              3 f1 bit(16),
              3 f2 bit(16),
        2 numberofnames fixed(15),
        2 primary_name offset,
        2 index_in_vtoc fix(15),
```

```
        2 dtm bit(64),
        2 dtu bit(64),
        2 domain fixed,
        2 zo_acl_hw bit(16),
        2 dtb bit(64),
        2 dtc bit(64),
        2 sname char(4),

1 nameblock based,
        2 next_name offset,
        2 file_descr offset,
        2 hash_fault offset,
        2 name char(32)varying,
```

The basic aim of a directory is to associate a name with a vtoc index, and thus a file, and secondarily, to associate the extra information in the fcb with this same file.

The general structure of a directory is best illustrated by stepping through the operation of listing all of the names of a particular segment, given its name and a pointer to the containing directory. This will be done later.

The fields in a directory are as follows: first_fcb: A pointer to the first fcb in a directory. It is never null, as there are dummy blocks on each end of the fcb chain for simplicity in chaining.

dlock: This is a lock that is locked whenever a user is modifying a directory in a critical way, such as creating or deleting a file in it.

cst: This is the central hash table. Each name hashes to one of the offsets in this array. This offset points at a name block.

dirarea: This is the area in which all allocations of fcbs and nameblocks are made.

The fields in an fcb (file control block) are as follows: next_fcb: since the fcbs are doubly threaded lists, this is one of the
two pointers in the list, this one pointing to the next fcb.

last_fcb: similar to the preceding, except of course for direction.

uid: not currently used.

flags.f1: The type of the segment. 8000=dir, 4000=link and 2000=normal segment.

flags.f2: not currently used.

number_of_names: the number of names currently on the segment.

primary_name: the offset into the dirarea where the first name in the bunch on this segment is kept.

index_in_vtoc: the vtoce index of the first vtoce that describes this file.

dtm: the date and time modified, i.e. the last time the core job wrote this segment back on to the disk. It is updated by hcs$core_to_index.

dtu: the date time used. This field is not updated since it would require writing through immense numbers of dirs on every call to hcs$initiate.

domain: either 0 or 1 depending on the ring number that this segment should run in if it is called. This has no effect on non-executable segments.

zo_acl_hw: these are the access bits that determine the access of both rings 0 and 1 to this segment. It is divided into two bytes. The first byte is the access that ring 0 has to the segment, and the second byte is the access that ring 1 has to the segment. Each byte has the following format:

bit 0: This bit is set if the appropriate ring is to take an execute fault if execution is attempted. This bit describes what the hardware should do, now whether or not the process will get a no-execute fault if execution is attempted. For example, this bit must be on to get the linkage section copied, or to switch rings from the ring that this describes when entering this segment from a different ring.

bits 1 and 2: These are the bits that are copied directly into the write protection bits of the mac register for this segment. Thus 00 means writable, 11 means take a fault on write, and 01 and 10 are not used. As with bit 0, these bits do not necessarily mean that a no-write fault will be generated when the mac interrupts, since one use of these bits is in maintaining the modified flag, to prevent

unnecessary write-throughs.

Bits 3 through 7 are basically software flags, not mac register fields, as the preceding bits were.

bit 3: This bit tells whether or not the linkage section has been copied. It is set whenever we take a no-execute fault in the ring that the segment should run in (the domain of the segment) and execution is allowed and this bit is off. At this time, the linkage section is copied into the linkage segement, and the execute fault flag is cleared to save time in intra-ring calls.

bit 4: This bit is on when gating from another domain is allowed. For example, if a segment is a ring 0 segment, and ring 1 is supposed to be allowed to gate into it, then the ring 1 access byte will have this bit set so that the ring change from ring 1 to ring 0 will be allowed. This bit is only examined when a no-execute fault occurs due to bit 0 being set in the access byte, thus having bit 0 off in the from-domain's access byte will prevent ring crossing faults from occuring, and the branch will complete without any ring change. This mechanism is used to switch from both rings 0 and 1.

bit 5: This is the read permission bit. It is currently not implemented, as the mac does not support read protection unless the entire segment is non-present.

bit 6: This is the execution permission bit. If this bit is off, and bit 0 is set, and an intra-ring call is made, it will fail because of a no-execute fault being raised. On cross-domain calls, this bit is ingnored in the caller's ring, as the controlling bit is the gate bit in this case.

bit 7: This is the write permission bit. If this bit is off, and a write fault is generated by the mac, then a no-write fault is raised by the system is the running proess.

dtb: This is the date-time backed up field, which is set by a ring 0 routine by the mag tape backup system.

dtc: This is the date-time created field. This is set by hcs$append_seg (or dir) when the segment is created.

sname: This is the address space name that the segment normally runs in. A name of all ascii zeroes means the segment should run in the caller's address space.

The fields in the nameblock are now described:

next_name: This is a pointer to the next name on the segment, if there is one, otherwise it is nullo. This field has nothing to do with the hashing, but rather is set up so that by starting at the fcb, you can get to the primary name, which is a name block, and by then following the next_name chain, you can get all of the names on the segment.

file_descr: This is a pointer to the file_control_block (fcb) on whose next_name chain this name may be found.

hash_fault: This is a pointer to the next name (or nullo) that hashes to the same value (by hcs$hash_cv).

name: This is the name of the segment. It is a char varying string. Although by pl1 conventions, "foo" equals "foo ", they hash to different values and are thus on different hash chains. Thus two such files could be created, since only the appropriate hash chain is checked, during creation and lookups, for the segment's name block.

The way that a file is looked-up can now be easily explained. First, the name is hashed by calling hcs$hash_cv. This returns the hash bucket number that the name is in. Starting at cst(i) where i is the value returned by hcs$hash_cv, we go down the name chain, looking to see if a name block with a name that is equal to the name that we are searching for is present. If there is none, then we report this, otherwise we get the fcb pointer out of the name block and then we have all of the information we need, since all of this is kept in the fcb.

Locking Strategies:

Most of the critical code in the system is protected by the use of locks and by three subroutines, hcs$lockr, hcs$lockw and hcs$unlock. The general format of a lock is:

```
1 lock based,
        2 ts_halfword,
        2 count,
```

The ts_halfword is used by the test and set instruction to prevent two different users from attempting to process the lock at the same time. Only one bit is needed for this, but the ts instruction seems to smash the entire halfword. The right half has one of two values. If the lock is locked for reading, then it is the count of the number of readers of the object being locked. If the lock is locked for writing, then lock.count is equal to a halfword -1. The ts_halfword is locked (non-zero) only for a short time at a time, when we are examining the lock. The basic idea is that if someone is modifying an object, no one else can read or write to it, but if someone is reading an object, others may read, but writers must wait for all readers to complete. The subroutines hcs$lockr, hcs$lockw and hcs$unlock all take one argument, a lock, and respectively lock for read, lock for write and unlock the lock they are passed. The two lock routines do not return until the lock is grabbed. During the attempt to grab the lock, the process uses a little bit of cpu time, just doing a ts and an svc 2, which gives the cpu away for a clock tick.

There are a large number of locks in the system, most of which are kept in the buffer header in segment one. These locks are:

Core lock. Ast lock. Selch lock. Dalloc lock. Get_disk_block lock. Individual sdb locks.

The core lock is a lock for the page allocation table. The lock is at 400 hex and the table starts at 404 hex. This lock is locked by the core job when it runs, and is for all intents and purposes obsolete, as no one else modifies the page allocation table, except for hcs$remove_pages, which will not cause any trouble, as hcs$remove_pages runs with the inhibit flag on and waits for the sdb's lock to be unlocked.

The ast lock is the lock that is locked when someone is adding a segment to the ast. The ast is the active segment table, and is a list that associates vtoce indexes with segment descriptor block addresses. Its structure is an array of 32 pointers (the ast) to active segment table entries (aste's). An aste looks like this: 1 aste based,

```
        2 thread ptr,
        2 sdb ptr,
        2 size fixed,
```

2 index fixed,

The ast is 32 pointers to different lists of aste's. The list in which the aste is to be added to is determined by taking the index number modulo 32. The thread pointer points to the next aste in the list, or is null. The sdb pointer is a pointer to the corresponding sdb. The size field is obsolete. The index field is the vtoce index of the segment whose sdb is at the address pointed at by aste.sdb.

The routine hcs$initiate locks the ast when it tries to set up the address space with the new segment being initiated. First initiate gets the vtoce index of the segment to be initiated from the containing dir. Then it locks the ast and calls find_sdb to search the ast for the index and return the sdb if the index is found. Since the sdb could have zero users and thus be garbage collected at any time by the core job, the routine hcs$find_sdb adds one to the sdb's usage count before returning the sdb so that it is impossible to use a deleted sdb. The routine hcs$release_sdb will decrement the user count in the sdb when you are done with the pointer to the sdb. After find_sdb is called, if the sdb does exist, hcs$map_sdb maps the sdb into the current address space, bumps the user count by 1 if the sdb was not already mapped in, and then hcs$release_sdb and hcs$unlock (on the ast) are called. If an sdb does not exist for the given index, the ast is left locked, and hcs$index_to_core is called to load the segment into core and get an sdb for it. Then hcs$add_in is called to add the sdb into the ast and then the ast is unlocked. Add_in should not be called until the sdb is perfectly set up, since find_sdb does not look at the ast lock. It is safe to add to the ast while searching it, but two people should not add at the same time. The ast lock is designed to prevent two people from loading the same segment into core twice, in different sdb's.

The selch lock is a simple lock. It is locked when someone is doing selch transfers. This prevents two people from using the selch at the same time. Since the mag tape driver uses the selch, and the disk driver uses it too, and only one can be active at a time, this lock is necessary. One must be careful not to request the core job to do anything for the process that has the selch lock locked, since the core job may have to do disk i/o and it will wait for the selch to be unlocked before performing it.

The dalloc lock is another simple lock. It is locked by someone when they are trying to allocate a vtoce index when creating a file. This prevents the lockups that occured when we locked the entire disk buffer containing the vtoce allocation mask being searched.

The get_disk_block lock is locked whenever a new page is being loaded into the vtoc buffers in segment one. Since the buffer strategy module is fairly complex

and on occaision has to wait for a buffer to free up, a lock is needed to prevent two different users from interfering with the common database.

The last locks that exist are the per-sdb locks. There are two bits per sdb, kept in the sixteen bit field sdb.states. There are two states bits that are relevent. The 1000 bit is the "wired" bit and the 2000 bit is the "total lock" bit. The 1000 bit is turned on when the disk server is doing disk i/o into or out from the segment. This bit used to be very important when the user loaded segments for himself, but now that the core job does this loading, this bit has become obsolete, except for things like the save command, which do disk i/o directly into scratch segments. The disk server itself actually sets this bit before starting the transfer and clears it immediately after the transfer completes. The 2000 bit is still a critical bit. This bit basically means "the sdb is being processed by another user and should not be looked at". What happens when a process encounters a locked segment, what it does depends on what the process is trying to do. The core job ignores this bit, and is careful not to use the information in it. If a running user process encounters such an sdb because it touched a swapped out segment or is trying to grow a segment that is being grown, or written through, then the process threads itself on the colusers chain. This is a list of users to be woken up when the 2000 bit is cleared. The field sdb.luser points to the first task control block (tcb) in this list and the field tcb.coluser points to the next one in the chain, until a null field is encountered. If the core job tries to wake up a job whose tty field is 0, the system will crash with a code 4, so this is a good debugging tool to see if jobs are being woken after they have been killed, especially after a fatal error.

One thing that is important to maintain in a non-zero user count for all sdb's to which pointers exist. This is why hcs$find_sdb adds one to the user count field in the sdb. Any time an sdb's user count goes to zero, the core job may garbage collect that sdb. In general, sdb's are deleted at various times. The core job deletes sdb's when it swaps out a segment if the user count is 0 for that segment. When a user logs out or newprocs, i.e, when a process is killed, all the sdb's are deleted that are swapped out and have zero user counts. Similarly, when a segment is terminated and it is swapped out, the sdb is deleted. In all cases when an sdb is deleted, what I mean is that hcs$ast_delete is called on the sdb, which removes it from the ast, and then hcs$sfree is called to actually free the storage that was used by the sdb.

No discussion of locks would be complete without mentioning two other favorites for preventing interference, turning interrupts off in the psw, and the inhibit flag. Both the disk server process and the core shuffler run with interrupts masked. The core job enables them again when it does operations that are likely to

take a large amount of time, such as moving a segment, however, most critical code runs with interrupts disabled. Note, however, that svc 2 will still give up the processor.

The inhibit flag is used when you need interupts enabled, so, for instance, characters get echoed, but do not want to loose the processor for anything else. The inhibit flag basically tells the processor to return to the current process as soon as possible after handling an interrupt. As soon as possible means immediately after a clock interrupt, and after waking up the disk server for disk interrupts. If the interrupt is a pasla interrupt, control is returned after doing the echoing of the character(s) typed. When handling the interrupt, it is necessary never to save the registers of the currently running process, since the condition stack-frame restorer needs to be able to ensure that the registers will not be saved. Thus interrupt routines should use register set 0, leaving the register set F alone so that the process can be restored with an lpswr 0 instruction.

The Address Space Switcher

One of the biggest problems with this machine is the small number of segments (16) in an address space. Since dynamic linking is accomplished by mapping each called file into a new segment, and each command is treated as a separate call, we would have a limit of sixteen different commands typed in a process, before the address space would be filled. To get around this problem, we run different programs in different address spaces. Each segment on the disk is associated with a four character address space name. In each process, for each address space that exists, separate address space is created for each different address space name that is dynamically linked to. The way that the address space switcher works is that when the dynamic linker snaps a link to foo$bar, when the pointer to foo is made, the address space name of the segment foo, which is stored in the sdb, is compared with the name of the current address space, which is stored in the tcb. If the name of the segment is non-zero and not equal to the name of the current space, then the segment is mapped into the new space, which will be created if it does not exist. Along with the segment being called, any arguments and descriptors are also mapped, and a new descriptor list and argument list is consed up. The program remembers which segments were mapped in this way, and terminates them when a return is made from the address space switcher. Thus there is a major different in the way the link snapper works, depending on whether or not a cross-space link is being snapped. If it is not a cross-space link, then the link snapper returns after snapping the link, but if it is a cross-sppace link, then the address space switcher is called, which in

turn calls the callee, and does the mappings described above. When the called program returns, the argument segments are terminated, and a return address space switch is performed. This basically consists of saving the state of the current address space and switching to the previous address space. An address space consists of the sdb pointers and protection bits for each segment, the lot and the hash table for the reference names. Currently, all of these are simply copied into the system, but this is subject to further improvements to speed up a cross-space switch. When an address space switch occurs, the linkage segment, stack, pds, pl1 operators, scs and hcs segments are all in the new address space when it is created. One obscurity is that since the only way to find out how many arguments are being passed is by looking at the argument list, each call must have an argument list, i.e. calling subroutines with 0 arguments will lose unless a dummy argument list is created. The address space switcher will recognize a header that says 0 arguments (4 or -4) but a general call generated by the pl1 compiler will use an undefined argument list pointer and perhaps an undefined descriptor pointer (depending on the argument list pointer). Note that currently the address space switcher will not map the target of input pointers, so that these objects must be passed either as dummy arguments, or the routine must be re-written to accept the object instead of a pointer to it.

The Binder Input Segments.

The files ns.sysin, ns2.incl.sysin, ns0.incl.sysin and ns1.incl.sysin are assembled in that order to get ns. Ns contains the lowest level system routines and interrupt vectors. I will go through all of the routines in ns one at a time describing them.

Starting at 400 hex, we have the core allocation lock and tables, which starts out filled with x'ffff'. The memory test fills in 0's for the available memory, where each 2K page is associated with a halfword that contains either a 0 for free memory, a x'ffff' for memory that is either non-existant or used by the operating system, or an sdb address for a page that belongs to an sdb.

The routine "entry_point" is the assembly language stack frame creator. It, being in hcs, causes a gate into ring 0, even though it is called by all assembly routines. This is subject to change. It first decides which stack it should be running on, which is kept in the d0s field of the current task's tcb. It then loads the linkage pointer from the lot in the pds and then does some complex decision making on whether or not to try to extend the stack. If we are running on the pds, no attempt is made to touch deeper on the stack. Similarly, if the inhibit flag is on, then again no attempt is made to grow the stack. Otherwise, the first_frame field in the stack header of segment 15 is set to the current stack frame (the name

is obsolete) and the stack is touched 500 bytes past the end of the current stack frame. Note that registers 0 and 1 must be reloaded if we are not running on the pds, for basically unknown reasons.

Next are two entry points into the scheduler, clock and crock. Clock is entered only when the real time clock goes off, since we use this clock to keep track of system uptime. Crock is a similar entry point, except that the uptime count is not updated. Next, if no task was running (cur_task = 0) at the time the interrupt occured, an attempt is made to start the disk server process. If the inhibit flag is on, the system does an lpswr 0 to return immediately to the process that was interrupted. Otherwise, the psw is saved in the current job's tcb and save is called to save the state of the current process. Save switches to register set 15, saves register 13 in rsamm and then loads 13 with the pointer to the current tcb and saves the rest of the registers, and then copies register 13 from rsamm into where it should go to be saved. Save then switches back to register set 0 and returns via register 12. Note that for each call to save, the psw must be saved independently, since the psw comes in various registers after a fault, depending on what type of a fault occured.

After saving the state of the machine, the clock/crock entr point flashes the lights and then branches to sr10 with register 10 set to the next user found by chasing the circular list pointer tcb.next, which gives the structure to the round-robin scheduler.

Sr10 is an imporant entry point to the scheduler. It assumes tha register 10 points to a valid tcb, and attempts to start that user. What sr10 does is first, if the user is blocked, the user is not started, but rather sr10 iterates, trying to start the user whose tcb is in tcb.next. If the 4000 bit is on in tcb.job_state, then the process will be started, otherwise, we first check that segments 13 and 15 are present, i.e. will not cause faults when referenced. The 4000 bit is provided so that when a process starts up, it can grow its own segments 13 and 15. Sr10 then loads the mac registers from the sdb's and protection halfwords in the address space. The 10 bit of tcb.job_state is or'd in with each mac register, so that the core job can reference segments that cause other processes to take non-presence faults so that it can load them. After the mac registers are loaded, the real registers in register set 15 are loaded and the process is resumed from its saved psw. One further point is that if the scheduler goes around the tcb.next loop 100 times and does not find a runnable task, then it zeroes out cur_task, indicating that there is no runnable task, and then it halts the cpu with interrupts enabled, so that i/o interrupts can be processed.

The routine svc4 is entered when an svc 4 is executed by the processor. This svc is supposed to try to warm start the cpu. If MagicSix has crashed because of a

spurious interrupt or some other basically harmless fault, this will succeed, otherwise it will fail miserably. What this routine does is zero out the inhibit flag, and then try to illegal instruction (with a valid psw) the currently running task. If there is no currently running task, this routine simply enters the scheduler loop looking for a runnable task.

The routine svc7 is entered when an svc 7 is executed. This svc is supposed to crash the system quickly with a code number in the lights. When executed, the system puts x'c0de' in the left half of the lights, and the parameter block address of the svc in the right half of the lights, enabling simple crashes to be easily understood by the operator.

The routine copy is a routine which copies from register 14 to register 8, and the bal to copy through register 12 is followed by a halfword byte count, which if zero means that the count is in register 5. It only copies an integral number of fullwords and should only be passed a length that is a multiple of four.

The routine cold_start is where the system comes to come up immediately after it is loaded by the ipl rom. When the system is first loaded, the illegal instruction trap address is pointed here, so that to start the system after loading, one need only get an illegal instruction. The first thing that is done by cold start is to stop the selch, in case it is still going from the ipl transfer. We then perform a simple memory test to find the size of available memory, modifying the page use table at x'400' correspondingly. The memory test consists of storing a halfword of x'ffff' in the first hw of each page and reading it back to ensure that it is still there. If this fails, the page is marked as bad (x'ffff' in the page use map). Next, we read the console lights and store the initial console for further use in the tcb of the disk server process. We then set the hw at x'78' to console*64+x'ff' so that magic 5 will not interfere with our processor. We then check to see if the upper byte of the lights was x'6', in which case we use the emergency file system instead (this information is passed in location 8 to the disk server process). We then set x'e000' to an empty area for obsolete reasons. We also set 13800 to a 40000 byte empty area, since this is where the linkage segment for the core job is going to be mapped. We set x'd000' to be an 8Kbyte empty area, the central free storage pool. It then sets the clock speed and starts the clock going. We then copy the bit counts of segments 0 and 1 into their respective sdb's, set the illegal instruction trap address to point to the normal ii handler, set the full word at 0 to x'22002200' for historical reasons (one cpu used to have a broken wire that made it difficult to stop if it was in an illegal instruction loop, so I set 0 to a bs * to try to cut down on these ii loops). We then branch to sr10 with register 10 set to start the core job.

Next we have the routines that are used to initialize the various processes when the system comes up and during newprocs.

The first of these is the routine "test_job" which is the place where the core shuffler starts running. Basically, all that it does is add the name hcs to segment 0, add an historical name to segment 1 (this is not necessary any more), and then call the main core shuffler entry point which sets up and then enters the main core shuffler command loop. Note that register ap (15) has a pointer to the halfword that must be zeroed in order to start the disk server process, to complete system initialization.

The routine "db_task" is similar, this is the place where the disk server starts running when it is made runnable by the core shuffler. This routine adds the name hcs to segment 0, creates the user task that comes up on the ipl console, sets the console field of this task, sets the psw of the new task to start at std_init, and then calls the main disk server entry point.

Std_init is the most complicated of all of the process initialiaztion routines. This routine must function for newprocs and for initial process creation. A newly created process has a zero length pds, linkage segment and stack. Since in order to be able to handle out of bounds faults to grow these segments, the pds must be 3K long, we must first grow the pds, at least. The reason that the create tasks primitive does not create tasks with reasonably large sized pds's is that since newprocs are handled by the disk server process, and growing these segments required sending messages to the core job, which could be waiting for the disk server to process a later message, the system would deadlock quite often when people would newproc. Thus the first thing that a normal process does when it starts up is send messages to the core job to grow segments 13,14 and 15 to normal sizes. The normal messsage sending facility requires allocating and freeing free storage, a fairly hairy operation to do since the routines to do this can not be called as the stack and pds have not been grown yet. Thus a special message buffer has been provided for starting up. It has a lock field so that if two people are newprocing at the same time, the system does not become inconsistant.

The first thing that a process does at std_init is to lock the message buffer. When it has gotten the buffer locked, we then call thread, after setting up the message with the sdb of the segment to be grown. We call thread a total of three times, once for each of the linkage segment, pds and stack. When a process is newly created, it must have its saved registers 6 7 and 8 set to be pointers to the sdb's of its pds, linkage segment and stack, so tat std_init can find these sdb's easily for sending these messages. Thread basically turns on the inhibit flag, threads the message on the core job's channel 0 ipc list of messages and sends a wakeup to the core job. We then thread ourselves on the wakeup list of the sdb, set our processes block/wakeup indicator to blocked and turn off the inhibit flag. We then do an svc 2, which runs the process scheduler, which will cause the block to take effect

immediately. We then return after the core job wakes us up.

After the three segments are grown, we copy a template stack header into the base of the stack and the pds. We clear the bit that indicates that we need not wait for the pds and stack to become present before scheduling our process, and set the areas in the pds and linkage segment to empty. We then unlock the message buffer and set the access on segment zero so that we take a linkage segment copying fault, which was heretofore disabled. We then do an svc 2, which runs us through the scheduler, so that we know that the next time we are scheduled, our mac registers will have been reloaded. We then load the psw with x'44f0' and then do the standard process startup procedure, namely adding the refname hcs to segmen 0, the refname nplo to segment 1 and then initiating "scs" from ">sl1" so that the link snapper can use make_ptr in this process. We then call the command processor at the process initialization entry, which prints the familiar greeting.

The next major piece of code in the system is the link snapper. When an svc 0 is encountered, the system simply signals the condition "SVC0" in the running process. Under normal circumstances, there is no handler for this condition, and so control transfers to the label real_sdh, which goes to real_no_handler to print an error message, unless the condition is "SVC0", in which case we fall through into the link snapper.

The first thing that the link snapper decides is whether the link is a new-style link or an old-style link. An old-style link is a 6 byte instruction, while a new-style link is 4 bytes long and instead of having the name of the link immediately following the svc, there is just an offset into the caller's segment where the name is contained. The way that the caller's segment is found is by searching the linkage offset table (lot) for the largest pointer to a linkage section whose address is less than that of the link. The names "hcs" and "scs" are special cased, and all other segment names are sent directly to scs$make_ptr, to find both the segment and the entry point. For scs and hcs, the segment numbers are known (2 and 0 respectively), and the entry point is found by using a special routine, find_map, which searches object segments for entry points. This duplication of code is needed so that we can initiate scs and find the entry point make_ptr in it.

For each style link, there are 4 different types of links, type 0,1,2 and 4. Type 0 is the normal link that gets snapped into a branch instruction and is threaded on a list of links to enable the link to be unsnapped when the segment it points to is terminated. Type 1 is the same, except that the thread field is not present, and of course the link is not threaded on the list. Type 2 links are links that are never snapped, but cause branches to the entry point to be re-resolved on each call. These, too, are not threaded on the list of links to be unsnapped on termination.

Type 4 links are external static links and are snapped into links that load register 2 with a pointer to the external static object and then return via a br 14. If the link is a type 2 link, then the link snapper simply returns to the place where we want to transfer control to. All other types of links simply return to the link after it is snapped, unless we are crossing an address space. In this case, no matter what type of link it is, we do not snap the link, but rather call hcs$switch which does a cross-address space call, and when it is returned to, a return across the address space boundary. In a cross-address space call, when the return is finished, the gate segment is terminated from the caller's address space.

The next routine to be described is the disk/selch interrupt handlers. The selch interrupt routine simple returns to the interrupted process. The disk controller and disk drive interrupt routines wakeup the disk server process, and will, if the inhibit flag is off, save the state of the currently running process and schedule the disk server process instead.

Then entry point hcs$gettab is used to get system variables from programs that are assembled in ns.sysin. The program takes a fix(15) argument, the index of the table entry desired, and this argument is bashed to the return value, which is also a halfword. The different indexes are:

0: The address of the core use lock.
1: The address of the selch interrupt count.
2: The address of cur_task.
3: The address of the ptr to the tcb of the disk server (the user to wakeup on a disk interrupt).
4: Db_task.
5: Std_init.
6: The address of a pointer to the ast.

The entry point get_time returns a fullword value, the number of clock ticks that have elapsed since the system has come up.

The entry point hcs$salloc is a non-user callable routine that manages free storage for the general system heap at x'd000'. In this area are allocated all tcb's, sdb's, aste's, hash tables for reference names (this may change), the ast and interrupt vectors for devices attached by the io system. This routine does most of its critical work with the inhibit flag on, to prevent the screw-ups that would inevitably occur if two users were to try to allocate and/or free storage in the central pool at the same time. There are two different standard crashes that can be

signalled from the free storage managers. The first is crash code 8, which means that there is no block in free storage large enough to hold the requested block. The other is crash code 7, which means that the free storage area has been apparently garbaged. The blocks returned by hcs$salloc must be double word aligned, as this is necessary to be able to do lpsw instructions from tcb's allocated in this area. There is an eight byte header in front of each block of storage, allocated or not. If it is an allocated block, then the format is

```
1 allocated_header based,
        2 one fixed(31) init(1),
        2 size fix(31), /* including 8 bytes for header */
```

otherwise the block is in this format:

```
1 free_block_header based,
        2 free_list_next_ptr ptr,
        2 size fix(31), /* including 8 bytes for header */
```

The algorithm used is a best-fit search for the free block to be returned. Experiments have shown that this works better for this application by far compared with best fit, as far as fragmentation is concerned. If a block is larger than is requested, it will be broken into an allocated block and a free block, and the allocated one returned.

The arguments to hcs$salloc are a ptr (returned) and a length (fixed bin(31)).

The entry hcs$sfree takes a ptr to an area returned by hcs$salloc, and frees it. This entry, too, runs mostly with the inhibit flag on. It consolidates free blocks maximally to minimize fragmentation. The length of the block to free is determined from the header of the block being freed.

The routine "popret" is the inverse of "entry_point", i.e. this is the return operator for assembly language stack frames. This operator is usually invoked by branching to !13+x'e4', unless the caller is assembled with popret. Basically, what popret does is to remove the stack frame, reload the registers from the removed stack frame's saved register 14.

The next routine is msf, which is the hardware condition signaller. It has three different ways to enter it. All assume that the routine "save" has been called to save the state of the process. Msf and msf2 store registers 14 and 15 in the tcb

of the user whose tcb is pointed to by r10 (which must be set to cur_task for all of these entries), while msaved does not do this. The other difference is that msf loads r3 with 0, while the other entries do not. R3 controls which routine is to be called. If it is zero, it is the standard pl1 condition handler. It r3 is 4, the routine called is the linkage segment copier, and if it is 8, then we call the routine that sends messages to the core job, to swap in or grow segments. These routines copy the registers at the time of the fault, which are saved in tcb.regs, into tcb.sigregs, to enable them to be saved on the stack. It then saves the psw in tcb.sigpsw, and saves the domain of execution in the high order halfword of tcb.sigpsw. We then switch to domain 0, by setting tcb.process_domaiin to 0, and set tcb.regs(3) to the value of register 3 that we were passed when msf* was called. We copy the condition name from the eight byte temporary "condition" into tcb.regs(14) and tcb.regs(15), set tcb.psrw to 0|svproc and schedule the user by branching to sr10. This is how we crawl out into mapped mode when a fault occurs. After we are running mapped at svproc, we load the stack pointer from first_frame+!15, which points to the stack frame deepest on the stack, in case the user's sp was garbage, and we then turn on the inhibit flag and push a stack frame. Then we shut off the inhibit flag. The reason for this is that when the inhibit flag is on, the stack frame push operator does not touch 500 bytes deeper on the stack, which could cause a fault that would garbage the information saved in the tcb. We then copy the saved registers from tcb.sigregs, the saved psw from tcb.sigpsw and also the saved domain of execution, in case we are returning to the pl1 operators after the fault, as they will not gate back to their old domain of execution automatically, into the stack frame that we have built. We store the condition name that is passed to us in registers 14 and 15 in the stack frame also, and then we branch through sigptr+!13(r3) to get the appropriate signaller. When this returns, we touch the stack (obsolete, I believe), and then turn the inhibit flag on and copy the saved machine conditions that we had saved in our stack frame, and restore them directly into the tcb. This is why interrupt routines must never save the registers of a process when it is running with the inhibit flag on, as that would confuse this section of code. We have popped the stack frame previous to doing this copy in to the tcb, and then we restore d0s to indicate that stack frames are to be built on the segment 15 stack, by setting it to !15. We then do an svc 1, which turns off the inhibit flag and branches to the scheduler without saving the state of the process, so that the conditions that we just restored into the tcb will be used the next time that the process is scheduled.

The next non-trivial routine in ns is the memory access controller fault handler. This is a great kludge, due to the fact that the mac itself:

1) Does not provide the referencing address when a fault occurs.
2) Except in branch and store type instructions, the cpu executes the offending instruction anyway, causing occaisional register garbaging, which is detected.
3) Does not provide the address of the faulting opcode, necessitating the running of a clever, but not perfect algorithm to back up across an instruction, since the psw is already updated by the time the fault has occured. Note that there is no 100% correct algorithm for backing up on these machines, although there would exist one if all of the opcodes whose first nibble is 4 were renumbered so that their first nibble was an eight.

The routine that is entered when the mac decides to warn of troubles is boundf. This routine calls the save routine to save the state of the process that was running, and then stores registers 14 and 15 in the psw field of the tcb. We then load a register with the mac status register and store a zero in to the mac status register. Storing a zero in to the mac status register is necessary, despite the claims of the 7/32 manual, to clear the interrupt flag in the mac, and if this is not done, then no writes to memory will complete from then on. We then load certain registers with convenient values, namely r5 gets cur_task, r11 gets the sdb pointer, and r12 gets the access byte for the ring that the process was running in. We then test for the presence of fault bits in the mac status register. We check no_execute faults first, followed by out_of_bounds faults, non_presence faults and no_write faults.

In order to signal error conditions, we load register ten with a pointer to a char(8) aligned name and then branch to do_fault, which will then signal that condition in the current process. This is done by branching to msf.

There are two routines that are called by most of the fault handlers, machac and badluckq. When you bal link,machac, register 4 is returned pointing to the referenced address. Machac also backs up the field tcb.psrw so that it points to the beginning of the opcode that took the fault. Machac has to do some guessing, the assumptions that it makes are as follows:

RX3 instructions with negative offsets that take faults will be taken as only 4 bytes instructions.

Opcodes whose first nibble is a 4 and which use register 0 in their R1 field will be taken as 6 byte instructions.

Generally, how the backup is guessed is as follows. It looks 6 bytes back from the fault address, and if the opcode could not have taken a fault (i.e. not a sis or somesuch nonsense) then we assume that we have a 4 byte opcode. In addition, the halfword 4 bytes before the psw after the fault is anded with f0f0 and if the result is not 4000 we assume we have a 4 byte instruction. The bit array macbac is a bit array which has a 1 if the opcode could not be a faulting opcode and is a zero if it could.

After the backup is completed, many routines call badluckq, which tests to see if the cpu has bashed a register while executing an instruction that should have faulted. This is not necessary for branching into segments that are swapped out, or where the process does not have execute permission, but is necessary for all faults where the psw is not the fault-causing address. It uses two bit maps, the first, mfq, has a one on for every opcode that bashes a valuable register when executed, such as "a", and for every illegal opcode also. The bit map macfail has a one on for every opcode that fails only if it has bashed an index register, such as load or load halfword. The opcodes in mfq are badluck faulted any time that they take faults, while those in macfail raise badluck faults only if r1 equals an index register field.

The routine to handle write faults first calls machac to evaluate the referenced address, then tests the 2000 bit of the sdb and if it is on, the process threads itself onto the coluser chain for that segment. Otherwise, we check bit 7 of the protection byte for the appropriate ring, and if it is off, we signal a nowrite condition. Otherwise, we must have a first-write trap fault occuring, and so we and out the bits 1 and 2 from the protection byte for this ring, and set the modified flag for the segment.

The execution fault handler is somewhat easier since the address reference that faulted does not have to be evaluated, it is in the psw. We first get the sdb address of the referenced segment,

and if the 2000 bit is on, we thread our process on the coluser chain and go blocked. Otherwise, we have two cases, one where the domain of execution is the same as the domain of the segment that we are entering, and one where they are different. If they are different, we first check to see if our domain has gate permission into the segment we have branched into (bit 4 of the protection byte). If we do not have gate permission, we signal the condition gate_err, otherwise, we check the 800 bit in sdb.states, and if it is on, it means that we are supposed to do argument validation. We then check to see if the stack pointer is in a valid range (between f0000 and 100000), and give a fatal error if it is not. Next, we check the segment to which we just branched. If the halfword is a x'5810' where the psw points, then we know that this is a pl1 entry point, and can do no more validation. Otherwise, we look back one halfword for the entry descriptor halfword (unless the offset of the psw is 0, in which case we give a gate_err), and check the argument count and writability of all of the arguments, i.e. ensure that all arguments that we will modify are writable to the caller also. The entry descriptor halfword is a 4 bit count followed by a 12 bit bitmap, one bit for each argument, where a 1 indicates that we will modify that argument and thus the caller had better have write permission on the segment containing it, and a zero bit connotes an input-only argument. If the halfword contains x'ffff', the meaning is special, i.e. do not allow this call. Similarly, a halfword of 0 means that all calls are to be allowed at this entry point. If all of these tests are passed, then the gate is allowed, the process_domain field of the tcb is changed to indicate running in the new ring, and the process is rescheduled. Note that in order to get domain-crossing faults, it is necessary for a segment in one domain to deny execution permission to all other domains, but to allow gate permission to all those domains that are to be able to call into this domain.

If the domain of execution is the same as the domain of the segment being entered, then there are only two choices, either a real fault should be taken, or the linkage section of this segment should be copied into the linkage segment. Thus, first we test bit 3, the linkage section copied bit, and if this bit is on, then we take a no-execute fault, otherwise we check the execute permission bit, and if it is off, we take an execute fault. If the linkage section has to be copied and we have

execution permission in this ring, then we clear bit 0 of the protection byte, so that no more execution faults are taken by this ring when calling this segment, set the linkage section copied bit in the protection byte, load r3 with a 4 and go to msf2, which causes hcs$signal2 to be called to copy the linkage section and then return to the program at the point where it took the fault.

Another type of fault that can be detected by the mac is the non-presence fault, raised when a reference is made to a segment whose x'10' bit is off in its mac register. This means that either a reference to a non-existant segment has occured, or that a segment has to be swapped in from the disk. The first thing that the handler does is to load the sdb address of the segment in which the process was running at the time of the fault. If this segment is non-existant, then we know that we should take a nonexist fault immediately. If the 2000 bit is on in sdb.states, then we thread ourselves on the coluser chain and try again after the current operation on the segment is completed. If the 4000 bit is on, then we go to swis, which is where to go to swap in a segment. If none of these bits are on, then the segment in which we were running must have been present in memory, and the referencing address must be the cause of the non-presence fault. We now call machac to evaluate this referencing address. If the sdb corresponding to the segment number of this address is 0, then we signal a nonexist fault. Otherwise, we call badluckq, which checks to see if we have a badluck fault. If badluckq returns, then we are safe, and then check to see if the 2000 bit is on in the sdb for the segment being referenced. If it is, again, we just thread ourselves on the coluser chain and go blocked. Otherwise, if the 4000 bit is on, we swap in the segment that is being referenced. In all cases, either the 4000 or the 2000 bit must be on.

The last routine in the mac fault handler is the handler for out of bounds faults. There are two cases, the first is if the psw is out of bounds, and the second is if the referencing address is out of bounds. The handler first checks the sdb of the segment that the process is running in, and if the psw is out of bounds in that segment, then we raise the oobounds condition (This is currently not checked, but will be shortly). If we pass this first test, then we call machac to evaluate the referencing address. Next we check the inhibit flag, and if it is on, we give a fatal process error, as the process has probably faulted

in some fairly critical piece of code. Next, we call badluckq, to see if we have a badluck fault. We then load the sdb address of the referenced segment, and if we do not have write access to it in the current domain, then we give an oobounds fault. We then check the 2000 bit in the sdb, and if it is on, then we block ourselves on the coluser chain. Otherwise, we check to see if we will get a maxlength error when we call hcs$set_seg_size. If we decide that we will, then we cause an oobounds fault to be raised (as usual, by going to msf), otherwise we call signal3 by loading an eight into register three, and going to msf2. This will send a message to the core job telling it what size to grow the segment to, and also blocking us on the sdb. When we call signal3, there is a machine condition stack frame (msf) above it on the stack. The condition name in that frame contains information for signal3. The first two bytes are the segment number of the segment that is being grown. The next two bytes are the new size for the segment. The next four bytes are the sdb address of the segment that is being grown. We then set tcb.d0s to !13 so that all future frames are pushed on the pds, in case we are growing the stack. Note that the code that checks if set_seg_size will complain is more than just checking if we will be growing a segment past its maxlength, since we must do the special processing for the stack also. If we are growing a stack segment (determined by checking the segment number of the referencing address against 15), then if we try to grow the stack past x'f9000', we cause a fatal process error, otherwise, we bump the maxlength of the stack to 7c00 and give a warning by signalling recurse.

The routine branched to to swap in a segment is very simple, it just stores the above information in the condition field for the msf, turns the 2000 bit on in the sdb , switches to the pds for growing stack frames, saves the psw from registers 14 and 15 in tcb.psw and goes to msf2 with register three containing an eight.

The entry hcs$push_address_space is a nearly obsolete entry that was supposed to alleviate the shortage of segment numbers by allowing a stack of segments to be used instead. This entry takes one argument, a bit(16) bit map telling which segments are supposed to be preserved over the address space change. Bit 0 corresponds to segment 0, and a 1 in a particular position means to push the segment, while a zero in that position means that the segment that is currently in that slot should be preserved in the new address space. Segments 0 and 15 can not ever be pushed. This entry has nothing to do with

the usual address space switcher, and is only called by hcs$create_task to ensure that segments 10, 11 and 12 are available for creating the pds, linkage segment and stack for the new process. When this entry is called, a new address space is allocated by calling hcs$salloc, and this address space is threaded on the list of address spaces that comes from tcb.pa_address_space after it is formatted correctly. We then call hcs$reschedule to force the reloading of the mac registers for this process, an operation that must be done every time that the mac registers are inconsistant with the address space of a process, or with an sdb.

The entry hcs$pop_address_space simply removes the current address space, threads on the next address space on the stack, and does a call to hcs$reschedule to reload the mac registers. It then frees the old address space with hcs$sfree.

The entry hcs$reschedule turns on the clock for one instruction, then restores the psw, and then does an svc 2 to give the processor away for a clock tick. The reason that we turn the clock on is that if the clock is off and we are waiting for some even that requires an interrupt to be processed before it occurs, then we will wait forever, since the clock would otherwise never be enabled.

The entry svc2 is where an svc 2,** instruction causes a trap to. It simulates a clock interrupt, even if interrupts are disabled.

The entry hcs$signal_defaults is the default signaller. This is the signaller that is called through the vector in the stack header. It basically finds the machine condition frame by looking a set number of frames up from its own frame, and then searches from there on up looking for an on-handler for the ccondition being signalled. Each time that a type four stack frame is encountered during the search, the name of the address space that we are now "entering" (in terms of the stack frames belonging to a program in a different address space), is saved so that when we finally find the on-handler, we know exactly what address space we should be in when we call the on-handler. Once we have found the handler, if it is in the same address space that we are in, we simply call it, otherwise we call hcs$switch to do the call to the correct address space. If the address space names are the same, then

we also load the linkage pointer for the routine that we are entering from the lot into register 12. If we do not find the condition name in an on-handler on the stack, then we call through !13+sdh, which is initially a standard default handler (see the description of the link snapper). If this pointer is null, then the process is given a fatal error.

The next routine is the fatal error/svc 5 handler. The label svc5 is where the svc 5 handler goes. If a program is running in register set 15 and wants to cause a fatal error, that process must do an svc 5,**. If you are in register set 0, (i.e. an interrupt-level routine) and want to cause a fatal error, you sh uld branch to svc5, although an svc 5 should work also, since svc's interrupt even in interrupt mode.

The handler is the same in either case. We first load register 10 with cur_task, and if it is zero, then we will crash the system (not implemented yet). Otherwise, we put x'ff' into the high order byte of tcb.dls, to indicate that a fatal error has occured. We then check cur_task against !1+x'11c8', which is the current owner of the ast, and if we are the owner, then we unlock the ast. We then load a "reasonable" value into the stack pointer and switch to ring 0, clear the inhibit flag if it was on, and wakeup our process if it was asleep. We set d0s to !13 so that stack frames are built on the pds, and load x'44f0' into the psw left word. We load the address of "newp" into the right word of the psw, and then for the stack and the pds, we modify their sdbs so that the states bits and the luser chain is zero for them. We also set ssize to size in each of these sdbs. This is to prevent the core shuffler from actually doing any requests for our process, since our process will soon be logged out. Otherwise, we might get a wakeup that we do not want. We then go to sr10, which will start our process again. The routine "newp" will set !15+first_frame (the current frame) to !13+x'fc' and then branches through a special word that is loaded by the disk server to point to hcs$newproc. We do not snap a link since we want to do the bare minimum in this process as we can.

The routine signal2 is the routine that runs on the pds and copies the linkage section of a segment into the linkage segment, using the free area in the pds or the linkage segment, depending on the ring of the segment whose linkage section is being copied. The reason

that this routine can run before hcs's and nplo's linkage section is copied is that the noexecute bit is turned off at the time of the fault, so that as long as the routine to copy the linkage segment of a segment does not use any static or links, the routine will run successfully. Note that this routine recurses, as it uses a pl1 operator to do the allocation in the pl1 area, and so while copying segment 0's linkage section, it will recurse in order to copy the pl1 operator's linkage section. This routine also sets the lot pointer to point to the linkage section that was just allocated.

Before documenting the block and wakeup calls, a brief word on the interprocess communication system is in order.

Each process has sixteen inter-process communication channels for sending an receiving messages and wakeups on. Each channel is completely independent of any actions taken on any of the other channels. The following operations may be done on a channel:
1: Creation. A channel must be created before any operations are perfomed on it by any process, including the owning process.

2: Blocking. A process may go blocked on a channel. This is similar to the "p" semaphore, in that the block may come before or after the wakeup. This operation stops the calling process until a wakeup has been sent. The wakeup may have been sent previously or it may still have yet to be sent.

3: Wakeups. A process may be woken up by any process.

4: A message may be read by the owning process. This consists of first calling block on the channel, and then removing a message from the threaded list of messages for that channel.

5: A message may be sent by any process. This consists of threading a message on the per-channel message list and then sending a wakeup on that channel.

Basically, a wakeup or a block is accomplished as follows. For each channel, a count is maintained of the number of wakeups that are pending on the channel. Each wakeup increments the count by one,

and each block decrements the count by one. Initially, the count is zero. When a block changes the count from 0 to -1, the process is also put to sleep, i.e. the field tcb.bw_count is set to -1 and an svc 2 is executed to cause the scheduler to take the processor from this process. Furthermore, when a wakeup changes the channel wakeup count from -1 to 0, the process sending the wakeup also sets the channel owner's tcb.bw_count field to 0. Thus when the number of blocks passes the number of wakeups, a process is stopped, and when this condition is rectified, the process is woken up again.

A channel looks like this:

```
1 ipc_channel based,
        2 count fix(31),
        2 first_msg ptr,
        2 last_msg ptr,
```

The reason for the two pointers is so that message insertion runs in constant time.

The entry hcs$block is called to do a normal block operation. It takes one argument, a fixed bin (15) channel number that must already have been created in the caller's process. This call performs a block operation as described above.

The entry hcs$block2 is a special cheap call for special types of blocks. This call simply sets the tcb.bw_count field of the tcb to -1 and does an svc 2 to give up the cpu. This call can only be used with hcs$wakeup2 or with interrupt routines that wake up a process by simply zeroing the tcb.bw_count field. It does not work if the wakeup is sent before the block is performed, nor does it separate wakeups into different channels, as does the normal calls.

The entry hcs$wakeup2 is a call that works with hcs$block2. It takes one argument, a char(8) aligned process name who is to be woken. No checks are made, except to ensure that the process being named exists. The tcb.bw_count field is zeroed and then this routine returns.

The entry hcs$wakeup takes two arguments, a char(8) name and a fixed bin(15) channel number in that process on which the wakeup is to

be sent. A wakeup is sent as described in the ipc section above.

The entry hcs$read_ipc is an entry that is used for sending messages to other processes. First, we call hcs$block on the channel, and then we turn on the inhibit flag and copy the first message in the message queue for this particular channel into the message buffer. If there is no message waiting, even though the wakeup was sent, then the system is crashed with a code of 2. The second argument to this program is the fix(15) channel number on which the message is expected to arrive. The first is a standard message block:

```
1 standard_message_block based,
        2 who char(8),
        2 garbage char(4),
        2 message_length fixed(31),
        2 message char(message_len),
```

The who field is filled in with the user name of the sender of the message. The message_length field is filled in with the size in bytes of the message. The reason for the garbage field is that when a message is in free storage, it has a pointer to the next message in the chain, and this pointer is copied out also, into the field labelled "garbage". If this field is non-null, then there is definitely another message in the message chain for that channel. If the field is null, there may still be a message there, which had been threaded on the list after the read_ipc that returned this message had already returned.

The entry hcs$write_ipc is a similar entry point. The arguments are the same as to hcs$read_ipc, except that the message buffer is an input argument in this call, and the channel number is a channel number in the receiving process. A block of free storage is gotten by use of the hcs$alloc routine, and then the message is threaded on to the list of messages for the appropriate channel. Then hcs$wakeup is called to send the wakeup to that process.

The entry hcs$start_task is called to start a blocked process at a given location in its address space. The first argument is a char(8) aligned process name and the second argument is a pointer in the blocked process's address space of where to start the process. The low order

word of the tcb.psrw field of the process to be started is set to the second argument, tcb.bw_count is zeroed, and the subroutine returns.

The entry point hcs$create_ipc takes one argument, a fix(15) channel number, and creates it. If the channel has already been created, i.e. if tcb.ipc_chain(channel#) is not null(), then this routine immediately returns. Otherwise, we allocate the channel header block, initialize it to 0 for the count, and null() for the first and last messages pointer.

The entry point hcs$signal3 is the routine that is called to run on the pds that sends a message to the core shuffler to either swap in a segment or to grow a segment when it is being extended. This routine first looks back along the stack one frame to get a pointer to the msf frame. 88 off of this stack frame is a pointer to the sdb, and 92 off of the frame is the pointer that was being referenced through at the time of the fault. If the 4000 bit is off in the states halfword in the sdb being processed, then we are growing the segment. We then call hcs$set_seg_size to grow the segment. This routine just sends a message to the core_job to do the real work. If the 4000 bit is on, we call hcs$swap_in to swap in the segment. The only difference is in the message that is sent to the core job. See the documentation on swap_in and hcs$set_seg_size. Before returning, we set the tcb.d0s field back to !15, so that all future frames are allocated on the segment 15 stack.

The entry point hcs$get_seg_ptr takes two arguments, a pointer and a char(32)varying name, and searches the current address space's hash table for the reference name and if it is found, the first argument is set to a pointer to the base of the segment. If no such name is found then the pointer is set to a fixed bin (31) minus 1, i.e. x'ffffffff'. The way that this routine works is by searching the name hash table for the name. The hash algorithm is unfortunately far from centralized, and consists of adding the rightmost byte of the length field to the first byte find names added to segments in other address spaces.

The next two entries are hcs$create_seg and hcs$alt_seg. These entries take one argument, a pointer, and ensure that the segment exists. If there is a segment in that slot, the routine simply

returns. Otherwise a new zero length swapped out segment is created. It is not associated with a disk segment, i.e. its vtoc index field in the sdb is zero. The difference between alt_seg and create_seg is that create_seg touches the base of the new segment before it returns, while alt_seg does not. The reason for this is to minimize bad_luck faults. The reason for the existance of the alt_seg entry is that there are certain operations which must never require the core shuffler's intervention, such as creating a process. Thus all that is needed is for hcs$create_task to use the hcs$alt_seg entry to create the new process's stack, pds and linkage segments and have the new process actually do the segment growing. The entry is a straightforward routine that simply fills in the fields of the sdb being created. The initial access for the segment is 1f1f and it has initially the 4000 bit set on. This is to indicate that it is a zero length segment, since the hardware does not support zero length segments. Thus we get a non-presence fault on the segment initially, and the core shuffler understands this special case.

The create_task entry is used to create new processes. It is usually called by the disk server, except when the cons command is used to set up a process on a new terminal. This is also a straightforward routine, as it simply fills in various fields in the tcb, address space and the other blocks that are allocated by this routine. This calls hcs$alt_seg since if we called create_seg instead, we would send a request to the core shuffler, which might need the disk server's services to complete the request. Since many of the calls to create_task are made by the disk server in response to newproc requests, this could easily cause a dead-lock situation. This entry is the only entry to actually call hcs$push_address_space in order to have enough segments so that it can create the dummy segments for the new process's pds, linkage and stack segments. The process that is created is blocked, has tcb.d0s set to indicate a login rather than a newproc or a fatal error, has the tcb.job_state field set to x'4000' which tells the scheduler that this process should be scheduled even though its pds and stack segments are non-present at the time. This is clearly necessary since the first action that the process will take will be to grow and format the header of these segments. This entry takes only one argument, which is a char(8) name for the new process. It does not check if a process

by the same name already exists.