Multics Virtual Memory — Tutorial and Reflections

Paul Green

Keywords: Virtual Memory, Multics, Tutorial

Summary: Tutorial and Reflections on Multics Virtual Memory Design

This is a summary of the Multics virtual memory design, taken from memory and reflecting Multics from the start of my association in 1969 until 1980 when I left the program. I have had a great deal of help writing this paper, which I deeply appreciate. Bernie Greenberg and Tom Van Vleck have been kind enough to read and comment on drafts. Bernie, in particular, has corrected a number of my lapses in my (now early middle-aged) memory. Many other people have also taken the time to assist me. Any errors that remain are my responsibility; please send comments or corrections to Paul_Green@vos.stratus.com so that I can correct future versions. [1]

Understanding the Multics virtual memory is key to understanding many Multics mechanisms. Storage allocation, addressing, protection, file I/O, dynamic linking, to name but a few, all depend on the Multics virtual memory design. [2]

The Multics virtual memory design is, I believe, the crowning achievement of Multics. The fact that Multics was written in a high-level language (PL/I), or that it had better security than other systems are probably better known, but not nearly as important, in my mind.[1] [3]

I have sprinkled in a few historical references about the hardware where I felt it was relevant. [4]

The Multics papers listed in the bibliography posted periodically by Tom Van Vleck contain a number of descriptions of the Multics Virtual Memory. Please refer to them for precise details. My intent in this posting is to give a high-level view of the features, capabilities, and mechanisms without getting bogged-down in the detail. In fact, since I am writing this from memory, I will probably get one or two of the details wrong. Since Multics evolved continuously from 1964 or so until the present, many of the details also evolved. This is a personal view, not a technical description. [5]

1 Multics Hardware Base

Multics was originally written for a custom-modified GE-635 system, called the GE-645. The classic reference on this version is The Multics System, An Examination of Its Structure, by Elliott I. Organick, MIT Press, 1972, ISBN 0-262-15012-3. [6]

1. Multics was not the first major operating system to be written in a high-level language. I believe that credit has to go to MCP for the Burroughs B5000 that was written in ALGOL.

The major components of the GE-635 were one or more central processing units, one or more I/O controllers, and one or more memory subsystems. Processors and I/O controllers accessed main memory thru ‘‘system controllers’’. All processors and controllers see the same physical address space. The flexibility and expandability of this design has been given as one of the major reasons that MIT chose the General Electric product over the (then brand-new) IBM 360 product as the basis for Multics. [7]

The GE-635 had core memory. Core memory in the mid-60s was not very dense; I believe it had to be hand-made. A cabinet the size of a classic telephone booth (perhaps 3 ft square and 6 ft high) held only 16,384 36-bit words! To get 64K (256KB, using 9-bit bytes of full ASCII, not the 6-bit BCD used in GECOS) took 4 cabinets attached to a square ‘‘dog house’’ frame that held the cables. Each component (CPU, I/O controller, system controller) required several cabinets of logic. Together with the disks, drum, printer, vacuum column tape drives, and Selectric-style operator's console, and motor generator, even a small system took up perhaps 3500 square feet of floor space. All of the cabinets were a lovely powder blue with aluminum corner edging. Lots of lights. It was a beautiful sight. [8]

The GE-635 (and 645) had 36-bit words and 18-bit word-level addressing. It had two 36-bit accumulators (A and Q) and eight 18-bit index registers. [9]

I wish I could remember the instruction times. I'm sure we'd laugh now, but it was a powerful machine in its day. (The time-of-day clock had 52 bits of precision and ticked every microsecond, and had logic to prevent reading the same value twice, even though this was probably unlikely. I'd have to guess that the average processor instruction time was well over 1 microsecond). Since it was very definitely a CISC machine, and asynchronous as well, individual instruction times varied widely. [10]

The native GE-635 supported absolute addressing, pc-relative addressing, indirect addressing, indexed addressing, indirect-then-index, index-then-indirect, autoincrement, and many more. Instructions were 36 bits (See Figure 1, below). The GE-635 was was word-addressed, not byte-addressed, so the byte-ordering issue never arose (it was essentially big-endian). [11]

Organick says that the GE design was very similar to the IBM 7094. I leave it to historians more qualified that I to decide to what extent this is true; for now, I simply point out that both machines are 36-bit, word-addressed designs with the offset in the upper half of the instruction word and the opcode & modifier in the lower half. The number and size of the index registers was also identical. Both used 6-bit BCD characters. One ‘‘improvement’’ GE made was to allow any number of levels of indirection, whereas Organick says that the 7094 stopped after two memory references. (The GE machines used a timer to detect loops in indirect-word chains. When the timer expired the machine took a ‘‘lockup fault’’. A similar timer detected failures in the asynchronously connected subsystems; it caused an ‘‘op not complete’’ fault.) [12]

The GE-635 had two privilege modes: Master Mode (privileged) and Slave Mode (unprivileged). Master Mode programs could reference all of main memory (in ‘‘absolute’’ mode). A base register relocated, and a bound register limited, all memory references made in Slave Mode. [13]

```
   Figure 1. GE-635 Instruction Format

  000000000011111111 112222222 222 333333
  012345678901234567 890123456 789 012345
 +------------------+---------+---+------+
 |      ADDRESS     | OPCODE  |UIB| TAG  |
 +------------------+---------+---+------+

   Field Name   Size      Purpose
   ----------   -------   -------
```

```
ADDRESS      18 bits   word address; 0-262143 (256KW)
OPCODE        9 bits   instruction opcode
U             1 bit    unused
I             1 bit    interrupt inhibit flag
B             1 bit    unused
TAG           6 bits   index/indirection type
```

The interrupt inhibit flag was only obeyed if the program was executing in Master Mode. But you could avoid interrupts in Slave Mode. The 635, 645 (and 6180) only sampled interrupts between pairs of instructions (they fetched even/odd pairs, executed both, then looked at interrupts). This was true whether you were in Master or Slave mode. So, even a user could get a two-instruction sequence that was atomic. Multics made use of this feature, although I can't remember where. [14]

The 635 and 645 also had instructions to execute single instructions or pairs of instructions. These could be used to ensure atomicity as well; interrupts would happen only after the instruction, or pair of instructions, had completed. Yet on the 645 these instructions could also take linkage, segmentation, or page faults...and these faults were restartable! The 635 and 645 repeat (and repeat-double) instructions WERE interruptible, but only after each instruction or pair of instructions. As a result, the 645 machine state information was considerably larger and more complex than the 635 machine state. The ability to handle interrupts and/or restartable faults during these particular instructions is arguably the hairiest aspect of of the (pre-6180-EIS) hardware design, and was the source of many subtle hardware bugs. [15]

General Electric added hardware support for segmentation and paging, and added 4 base (segment number) and 4 offset registers to turn the GE-635 into the GE-645. The 18-bit address field of the instruction became a 3-bit base/offset register number and a 15-bit offset. Bit 29 (formerly unused; see Figure 1) controlled whether the 18-bit form or the 3-bit/15-bit form of operand address was used. [16]

```
   Figure 2. GE-645 Instruction Format

 000 000000011111111 112222222 222 333333
 012 345678901234567 890123456 789 012345
+---+---------------+---------+---+------+
|BR |   ADDRESS     | OPCODE  |UIB| TAG  |
+---+---------------+---------+---+------+

   Field Name   Size      Purpose
   ----------   -------   -------
   BR            3 bits   base register; 0-7
   ADDRESS      18 bits   word address; 0-32767 (32KW)
   OPCODE        9 bits   instruction opcode
   U             1 bit    unused
   I             1 bit    interrupt inhibit flag
   B             1 bit    0=no base register (GE-635 form)
                          1=base register (GE-645 form)
   TAG           6 bits   index/indirect type
```

where TAG is

```
 01 2345
+--+----+
|Tm| Td |
```

```
+--+----+

 Tm        2 bits    Tag modifier
                     00   R-type    register, no indirect
                     01   RI-type   register then indirect
                     10   IT-type   indirect then tally
                     11   IR-type   indirect then register

 Td        4 bits    Tag descriptor
                     0ccc      ccc is a 3-bit code
                     1rrr      rrr is a 3-bit index register
```

The R, RI, and IR tag modifiers could be used with tag descriptors that specified one of the 8 index registers, either half of the A or Q register, the instruction counter, or no register, as follows.

```
The 3-bit "rrr" encoding permits eight index registers; each
is 18 bits of signed offset.  The "ccc" encoding permits 8
forms of direct modification.  They were, in no particular
order (and from memory!):

DU   direct upper        address used as upper half of
                              immediate operand, lower=0
DL   direct lower        address used as lower half of
                              immediate operand, upper=0
AU   accumulator upper   upper half of A register used as
                              index register
AL   accumulator lower   lower half of A register used as
                              index register
QU   quotient upper      upper half of Q register used as
                              register
QL   quotient lower      lower half of Q register used as
                              register
IC   instruction counter IC used as index register
N    no modifier         no indexing
```

The IT tag modifier had its own set of tag descriptors that permitted various types of “tally modification”. This was basically a clever use of indirection to implement auto-incrementing or auto-decrementing index values. It also provided rudimentary character support on top of the basic word-address architecture. One problem for Multics was that the tally words were not extended to work well with segmentation. The tally words (holding just an offset and length) had to be in the same segment as the data being manipulated. This wasn't a problem for the GE-635 because all of main memory was essentially one segment. But it was a problem for Multics, as it meant that tally words could not be used with pure (read-only) segments. Also, tally words didn't fit well into a PL/I model, and so the use of them on Multics was limited to a few, clever assembly-language programs. [17]

Multics needed the ability to have indirect words that contained segment numbers. This was accomplished by taking an unused value in the IT tag space. The tag was named “ITS”; this stands for “indirect thru segment”. It was heavily used. Everyone who worked on Multics knew about “ITS pointers”, and everyone who programmed eventually learned the octal value of the ITS tag: 43. When the ITS tag was present in an indirect word, it meant that this indirect word contained a segment number and the following indirect word contained the word address. There was also an ITB (“Indirect Thru Base”) tag that took the segment number from a base register instead of memory. It was rarely used (I found a use for it in the code generator I wrote for my undergraduate thesis, but that is another story). On the 6180 the ITB tag was renamed ITP but the function stayed the same. [18]

It is a happy coincidence that the ITS and ITB/ITP tags are in the IT region of the tag modifier space. They were placed there because that is where there were unused values, not because of the apparent name similarity, for they are really quite different functions, and implemented by a completely different part of the processor logic. [19]

The existing 635 FT ("fault tag") IT modifier was renamed to FT1 ("Fault Tag 1") and a new tag was added to indicate that an ITS pointer was invalid; it was named FT2 ("Fault Tag 2") and was used within Multics to implement dynamic linking. (The Multics name space management and dynamic linking mechanism is complex and worthy of a paper of its own). [20]

The format of an ITS pointer is as follows: [21]

```
   Figure 3. GE-645 Pointer Format  (72 bits)

  000000000011111111 112222222 222 333333
  012345678901234567 890123456 789 012345
 +------------------+---------+---+------+
 |  SEGMENT #       | UNUSED  |UUU| ITS  |   WORD 0
 +------------------+---------+---+------+

  000000000011111111 112222222 222 333333
  012345678901234567 890123456 789 012345
 +------------------+---------+---+------+
 |  ADDRESS         | UNUSED  |UUU| TAG  |   WORD 1
 +------------------+---------+---+------+

   Field Name   Size      Purpose
   ----------   -------   -------
   SEGMENT #    18 bits   segment number; 0-262143
   UNUSED        9 bits   unused
   UUU           3 bits   unused
   ITS           6 bits   '43'b3

   ADDRESS      18 bits   word address; 0-262143
   UNUSED        9 bits   unused
   UUU           3 bits   unused
   TAG           6 bits   index/indirect type
```

In the case of an unsnapped (unresolved) link, the ITS tag was replaced with a FT2. This tag caused a restartable fault that was handled by the dynamic linker, which would resolve the reference and update the pointer to contain the segment number and address of the target reference. It would replace the FT2 with ITS and restart the instruction. [22]

The 645 could, and did, run native 635 software unchanged when it wasn't running Multics. In fact, most, if not all, of the hardware diagnostics ran in native 635 mode on the bare machine, and tested only the 635 capabilities. It was not unusual to turn the machine over to the field engineers due to a problem, have them run the diagnostics without a problem, and turn the machine back unfixed. [23]

Each of the 8 base registers of the 645 had two modes. In ‘‘unpaired’’ mode a base register held a only a segment number and was therefore a pointer to the base of that segment. In ‘‘paired’’ mode a base register held a word offset and was paired with a different base register that held the segment number. Privileged instructions were provided to manipulate these modes. [24]

Multics ran with 4 base registers paired and 4 unpaired. The even-numbered registers (ap=0, bp=2, lp=4, sp=6) were paired with the next-highest odd register (ab=1, bb=3, lb=5, sb=7). In this way, Multics had the use of 4 general-purpose pointer registers and 4 ‘‘base of same segment’’ registers. The uses of most of them were fairly fixed, at least while in PL/I code. [25]

The ‘‘ap’’ register pointed to the argument list during a call operation, and to the PL/I operators transfer vector once a PL/I program was invoked; the ‘‘ab’’ register followed along accordingly. The ‘‘bp’’ register was the only real temporary, general-use pointer register; the ‘‘bb’’ register, of course, pointed to the base of the segment that ‘‘bp’’ was using. The ‘‘lp’’ register pointed to the static and linkage data area for the executing procedure (PL/I internal static); the ‘‘lb’’ register pointed to the base of the affiliated ‘‘combined linkage segment’’. The ‘‘sp’’ register pointed to the current stack frame; the ‘‘sb’’ register pointed to the base of the stack segment. [26]

By carefully reserving the base of the linkage segment and stack segment for key data structures, the Multics designers were effectively able to squeeze out two more pointer registers (e.g., ‘‘lb’’ and ‘‘sb’’). [27]

Note that 3 base register pairs are used to access 6 data areas (argument list, pl1 operators, current static, base-of-static, current stack frame, base-of-stack), leaving just one base register pair available for every other address! It was a tight fit.[1] [28]

Instructions were provided to store the contents of a base register pair in memory (See Figure 3, above), but the only way to load a base register pair was to use the ‘‘effective address to base register’’ instruction. Load instructions were added on the 6180, but rarely used. Every load of a base register pair (called a ‘‘pointer load’’ for brevity) had the possibility of taking a linkage fault, etc., and ultimately, on the 6180, also had to deal with ring level validation. These instructions were heavily used and were not cheap. [29]

There was a hidden base register to record the segment number of the executing program (procedure base register, ‘‘pbr’’). [30]

Note that there was no direct hardware support for rings on the 645. The ring mechanism was implemented by keeping unique descriptor segments for each ring used by a process, and by forcing a hardware fault when a program wanted to cross rings. The fault handler would switch descriptor segments. [31]

In 1969 (I think), Honeywell purchased the General Electric Computer Division and formed Honeywell Information Systems. General Electric had been working on a replacement for the GE-635, named the GE-655 internally. Honeywell renamed it the 6000 series. [32]

---

1. I have never fully understood why the hardware had the ability to run the base registers unpaired. In hindsight, it would have been much simpler (and MUCH more useful) to have 8 general-purpose pointer registers. Indeed, this is what was eventually done on the 6180, as we shall see. Base registers were a completely new concept to the 645; the 635 had only absolute, indexed or PC-relative addressing using a classic base-and-bounds scheme. Multics itself never used the generality implicit in the pairing scheme; it paired the registers early in initialization and left them that way forever. Only the Bootload Operating System (BOS), which is a mini-OS for booting, dumping and operating Multics, kept all of the base registers unpaired. This gave BOS eight segment registers, and it put a zero in register 0, a one in register 1, and so on. This meant that the assembler syntax 2|47, which normally meant ‘‘offset 47 in the segment in register 2’’ now also meant ‘‘offset 47 in segment 2”! (A clever trick by Stan Dunten, the original author of BOS). Perhaps having the ability to run the base registers unpaired would have been useful had GECOS ever been ported to the 645; it could have used them in unpaired mode, since it did not support segmentation.

In the early 1970s, Multics was ported to the Honeywell 6180, a Honeywell 6080 with Multics segmentation, paging, and ring hardware added. The segmentation and paging hardware was very similar to the 645 implementation (we missed a major opportunity to enlarge segments!), but the ring hardware was all new. [33]

The Honeywell 6180 went thru many evolutions and appeared under a number of different marketing names, but the basic hardware changes required by Multics stayed constant. For simplicity, I use the name 6180 in this paper to cover all of the subsequent versions. Eventually, Honeywell gave way to Bull HN which gave way to Bull. I mostly use ''Honeywell'' here because it is the company I worked for, and the company that existed for the years I am describing. [34]

I don't know of a reference to the 6180 version of Multics in any book. The Honeywell manuals and various papers and technical reports by Honeywell and MIT people are the only source I know about. [35]

We changed the 8 base registers into 8 pointer registers (each could hold a segment number and a word offset) and added hardware support for rings. Segment descriptor words now contained ring brackets and two hidden registers were added, the procedure ring register, and a ring register for the effective address. [36]

The Honeywell 6000 product line had an ''extended instruction set'' (EIS) option that supported character string, bit string, fixed decimal, and floating decimal data. Character strings and bit strings could be as long as a segment! Instructions were even provided to perform COBOL ''edits'' of picture data. EIS was built, I suspect, to support COBOL, but we quickly made it a requirement for Multics and fixed the PL/I compiler to use it. (The decision was quick; the actual implementation took us over a year as we ran into many bugs in the PL/I compiler and the EIS hardware. The first version of the EIS box had so much trouble handling restartable faults that Bill Silver wrote a special test harness to try all of the possible cases of multi-word EIS instructions, descriptors, and operands overlapping page boundaries). [37]

Over the years, the 6000 line was upgraded to use semiconductor memory, faster disks, and all the usual peripheral improvements of the day. An awful lot of staff time went to keeping up with the newer, faster hardware, rather than, say, functional enhancements (some things never change...). [38]

OK, so much for the hardware. How did Multics use it? [39]

2 Multics Virtual Memory Design

Multics has a segmented, paged virtual address space that offers 8 nested levels of privileged (called rings). [40]

The Multics file system supports a tree-structured hierarchy of directories, up to an arbitrary limit of 16 levels. File names and directory names are full ASCII and up to 32 characters long. Files and directories can have multiple names. Symbolic links for both files and directories are supported. Files and directories can have discretionary access control lists as well as nondiscretionary security levels and categories. [41]

Multics (at least in this era; pre-1981) does not speak of ''opening'' files. Multics supports a call to ''initiate'' a segment that maps onto an entire file. Internally, Multics initiated segments onto entire directories as well. Thus, all files are ''memory mapped'' in their entirety into the address space of every process using them. Directories are handled the same way, although since they are ring 0 segments, users cannot read or write them directly. [42]

When a program is done with a segment it issues a call to ''terminate'' the segment. This disconnects the segment number used by the calling process from the file. If no other ''initiations'' remain, the file is effectively closed. [43]

The term ‘‘file’’ and ‘‘segment’’ are often used interchangeably as a result of this one-to-one binding. In hindsight, it seems clear that the term ‘‘segment’’ really applies to the way the data is mapped into the address space, and the term ‘‘file’’ really applies to the way the data is stored on the disk. [44]

The address space of a process is just the set of files and directories that have been initiated (opened) and not yet terminated (closed). While most Multics programs carefully terminate the data files that they initiated, the Multics implementation of dynamic linking leaves all programs that have ever been referenced initiated forever, so a Multics address space tends to grow with time. The Multics kernel also leaves directories initiated (to speed up re-references), and this can also use up the address space. Programs that walk a directory tree must call a special kernel entrypoint to terminate currently-unused directories or they will run out of segment numbers. [45]

Programmers who were accustomed to an operating system in which their programs had a definite start and completion point, and where program completion meant removal from the address space (which was just about everyone), had a tough time initially on Multics. In Multics, the whole process is really one program. Individual segments can come and go dynamically, but the program (process) lives forever. Multics eventually added a concept called ‘‘run units’’ that simulated the conventional behavior. [46]

The Multics compilers and binder (a program to concatenate and prelink multiple compilation units) must call special kernel entrypoints to unlink all references to program segments that they are about to recompile or rebind. A failure to do this would result in previously-snapped links becoming stale. If a compiler takes a linkage fault, it is also possible for the dynamic linker to find the very program that is being compiled; this happened to me a few times when I was bootstrapping the compiler, until I took more care in the placement of the output files. [47]

2.1 Segmentation and Paging

Segments are 0 to 256 pages of 1K words (4K bytes) each, or a megabyte total. Segments are used for everything; programs, data, stacks, temporary storage; segmentation is pervasive and fundamental. There is NO supervisor or user storage other than segments. Segmentation is very definitely visible to the programming model, but pages are invisible, except for efficiency considerations. [48]

A process can have up to 4096 segments. The kernel shares the address space of the user and takes up the first few hundred segments. Segments can be shared with other processes or private to a process. While it is possible to initiate a file in different processes with the same segment number, it is cumbersome, so shared user segments rarely have the same segment number in different address spaces. On the other hand, since every new process inherits the address space of the kernel, it is common (indeed, a requirement) for the initial kernel segments to have the same segment number in all processes. This allowed the kernel to make and use pointers valid in all processes in a way which user-ring programs could only envy. [49]

Another interesting feature of the kernel addressability scheme was the ‘‘process data segment’’ (PDS); a segment in the kernel address space that was created anew for each new processes, but which had exactly the same segment number in each process. Thus, the kernel's fixed references to locations in this segment would resolve to the same offset in different segments as the kernel executed in different processes. The descriptor segment itself was managed in a similar way. There was also a ‘‘processor data segment’’ (PRDS) that was per-CPU, and likewise always had the same segment number in all processes. [50]

Multiple initiations of a file in a single address space are possible but all use the same segment number. [51]

Multics maps segments directly onto files and directories, so that ‘‘opening a file’’ means ‘‘creating an association’’ between a segment number and the file. Reading (or writing) the Nth 4KB disk block of a file is as simple as reading (or writing) the Nth 4KB page of a segment. The binding between a segment number and a file is per-process and exists only for the life of the initiation. [52]

The kernel has segments that are not associated with a file, but the only way that a user can create a segment is to create and open a file somewhere. Multics handily gives every process a temporary directory ("the process directory") where you can create temporary segments. [53]

The processor has a "descriptor segment base register" that points to the "descriptor segment" of the running process. The descriptor segment has one entry per segment. Each entry is called a segment descriptor word (SDW). The 6180 version of the SDW contains (very roughly; I no longer have the manuals available): [54]

```
faulted (invalid) bit
paged/unpaged bit
read access mode bit
execute access mode bit
write access mode bit
privileged bit
unencacheable bit
gate bit
physical address of the page table for the segment
  or physical address of the segment itself (18 bits)
ring brackets (write, read, call) (9 bits)
maximum segment length / call limiter (14 bits)
```

The faulted flag indicates that the SDW is invalid. The fault handler is expected to correct this and restart the instruction. [55]

The read/execute/write mode bits give the access permission modes for the entire segment (in association, naturally, with the ring brackets). [56]

The paged/unpaged bit provides a mechanism for unpaged segments; i.e., segments with no page tables. The SDW points directly to the segment in main memory. The segment length field in the SDW is the high-order 14 bits of the necessary 18 bit length; thus, the resolution of the length of unpaged segments is to a 16-word boundary. Still, this is considerably better than a page (1024-word) boundary. Unpaged segments save both time and main memory. The Multics kernel uses these "unpaged segments" for wired segments; i.e., for segments that are exempt from file system (paging) activity. BOS, the primitive operating system mentioned earlier, uses ONLY unpaged segments. All user segments, and all pageable kernel segments, have page tables. This feature should not be confused with the independent mechanism for "wiring" pages into main memory so that they cannot be paged-out; all unpaged segments are, of necessity, wired, but wired segments can be both paged and unpaged. [57]

Minimizing the use of paged segments was not an issue of beauty. It was a matter of real efficiency. The addressing cycle is much shorter and faster for unpaged segments because the processor (actually, the "appending unit") doesn't have to look up or factor in the PTW. Most collection 1 supervisor segments (whether code or data) are both wired and unpaged. This includes all parts of the paging system itself, even the segment used to hold the system page tables (the "SST" in Multics jargon). No doubt this was even necessary to avoid an unpleasant recursion in address formation. A few segments that have to be accessible thru the file system are paged; this permits a process to initiate and reference these segments normally; they can be wired or unwired as required. For example, "pl1_operators_" is both paged and wired, whereas "slt" and "config_deck" are paged and unwired. [58]

If this sounds funny, recall that the opposite of wired is unwired, and the opposite of paged is unpaged. They are different, though intertwined, concepts. Wired/Unwired refers to whether a segment is permanently resident in main memory or not. Paged/Unpaged refers to whether a segment has a page table and can be page-faulted. Because Multics has no mechanism for swapping unpaged segments in or out of main memory, an unpaged segment must be wired as well. The confusion arises because segments with page tables can be wired, too. [59]

Interestingly, by the time during bootload that Multics is ready to convert unpaged segments into paged segments, it is already using some of the very segments that must be converted. For example, the routine in question, ‘‘make_segs_paged’’, is written in PL/I and can at any point invoke code within pl1_operators_, even as it is converting it. It is a measure of the elegance of the design that it is able to convert ALL of these segments without any special cases. [60]

The privileged bit permits privileged instructions to be executed out of this segment. [61]

The unencacheable bit prevented data from the segment from appearing in a CPU cache. Both the 645 and 6180 versions of Multics supported multiple CPUs; both had per-CPU caches; both made the supervisor ultimately responsible for cache consistency. The 645 CPU encached only SDWs and PTWs (in the ‘‘associative memories’’; one for SDWs and one for PTWs). The supervisor cleared each CPU's associative memory after it changed the main memory copies of SDWs or PTWs. The 6180 CPU had similar PTW and SDW associative memories, along with a general instruction and data cache, and Multics computed encacheability at the segment level as a function of access mode and sharing status. Steve Webber and Riley Dobberstein designed and patented the segmentation aspect of the 6180 cache and Steve won a Honeywell Sweatt award for it. Honeywell had designs for hardware-mediated cache consistency, but the product never went into production. [62]

The gate bit indicates that this segment is a gate. This activates the ‘‘call limit’’ field of the SDW. Obviously, this bit is redundant to the ring brackets, but is presumably easier for the hardware to decode. [63]

2.2 Paging

Multics implements a classical demand-paged virtual memory management system. [64]

Reading a particular page of a segment will read the associated disk block into main memory, if necessary, or will materialize a page of zeros if the disk block has never been written. For obvious reasons, reading is therefore synchronous with disk activity. There is no way to ‘‘preread’’ a page. [65]

Writing a particular page of a segment will first read or materialize the page, if necessary, and will then store the new data and set the ‘‘modified’’ flag in the page table word. At some future time the kernel will notice the modified flag and will write the page back to the disk, resetting the modified bit. [66]

Reading or writing a page sets the ‘‘used’’ bit in the page table word. The kernel uses this bit to determine which pages have been recently referenced (and, by extension, which have not and are therefore candidates for replacement). [67]

Adding a page to a segment (whether zero or nonzero) increments the size of a segment and hence increments the ‘‘disk quota usage’’ of a directory. Thus, it is possible to get a software ‘‘quota overflow’’ fault by simply reading thru many pages of zeros. This fault is restartable, and so waiting a few seconds for these pages to be disconnected by the normal page replacement algorithm is often sufficient to be able to continue. Just another one of the many ‘‘funnies’’ in Multics, and another one of the exceptions to the supposed invisibility of the paging system. [68]

While this mechanism is fine, even superior, for casual use of data, it is insufficient to meet the requirements of a transaction-protected database manager. Such a product needs precise control over the scheduling and ordering of disk writes. One of the most frustrating activities on Multics is to try to write code that recovers corrupted files (i.e., from interrupted file modifications); or simply tries to avoid corruption by carefully ordering writes. Because the ordering of the actual disk writes bears no relation to the ordering of the modification of the pages, and because a system interruption may prevent some of the writes from completing, it is impossible to place an ordering on the modifications that actually appear in the disk file.[69]

Eventually, Multics provided a system call to force all modified pages to disk and not return until this had completed (a ‘‘file sync’’ in Unix terms). This is effective, although expensive. The larger problem is that having to add a ‘‘file sync’’ primitive contradicts the supposed transparency of the paging system and works at cross-purposes to the whole Multics virtual memory architecture. That ‘‘file sync’’ was necessary admits of a major flaw in the design. [70]

Each page table word (PTW) contains (from memory): faulted (invalid) flag physical address of the page in main memory used flag modified flag [71]

There were a number of fields in the PTW that were unused by the hardware but heavily used by the software. [72]

2.3 Pointers

A ‘‘pointer’’ in Multics is a complete address. It has a segment number, word offset, byte offset, and (on the 6180) a ring validation number. The 645 had one pointer format; the unpacked form shown in Figure 3, above. The 6180 has an additional, packed form. An unpacked pointer is 72 bits long; a packed pointer is 36 bits long. Unpacked pointers (and the hardware) support 32K segments. Packed pointers support 4K segments. Packed pointers were introduced with the move to the 6180 processor, and their popularity led to a practical limit of 4K segments per process. This turned out not to be a problem. [73]

```
Figure 4.  Packed Pointer Format (36 bits)

 000000 000011111111 112222222222333333
 012345 678901234567 890123456789012345
+------+------------+------------------+
|OFFSET| SEGMENT #  | ADDRESS          |
+------+------------+------------------+

Field Name   Size      Purpose
----------   -------   -------
OFFSET        6 bits   bit offset; 0-35
SEGMENT #    12 bits   segment number; 0-4095
ADDRESS      18 bits   word address; 0-262143
```

Note that packed pointers do not have an ITS tag and therefore do not support indirection. They do not have a ring number and thus cannot be used as arguments to a lower ring. The 6180 has new instructions for loading and storing packed pointers. [74]

2.4 Privileged vs. Unprivileged Mode

The 645 had Master Mode and Slave Mode. The 6180 has privileged and unprivileged mode. Both master and privileged mode exist to prevent unprivileged code from performing critical functions. [75]

Privileged mode on the 6180 is a segment attribute that is valid only while executing in ring 0. Certain critical processor and system control instructions can only be executed in privileged mode. In keeping with the principle of least privilege, an initial attempt was made to limit the number of kernel segments that had the privileged bit on, and to collect all privileged instructions into modules dedicated to that purpose. [76]

However, as almost all such instructions were in very time-critical paths of the kernel, the assembly-language callers of these routines were eventually changed to invoke the instructions in-line and to make the containing segments privileged. If you have permission to use an privileged instruction, you might as well use it efficiently. To quote Bernie Greenberg (who provided this anecdote), ‘‘If the supervisor is going berserk, a cupboard full of pea-shooters isn't going to help us.’’ [77]

2.5 Rings of Privilege

Instead of the usual 2-level (supervisor/user) protection scheme, Multics has 8 levels of privilege, numbered 0 to 7. Ring 0 is the most privileged. Ring 7 is the least privileged. [78]

As explained in the section above, ring 0 was further divided into privileged and unprivileged code. [79]

The terminology we use to describe parts of an operating system has never been consistently defined, and the usage of some terms has changed over the years. I'll try to use terminology that is fair to Multics and also accurate by today's standards. This may require an occasional explanation. [80]

Ring 0 was effectively two large programs; one that was unpageable or ‘‘wired’’, and one that was itself paged. However, the tools we had did not support the two program concept, and so great, manual, care had to be taken to keep them separate. While parts of the ring 0 code were more basic than others (page control versus directory control versus name space management), and while the major pieces were fairly well separated, nevertheless all of the code in ring 0 shared the same environment and could easily interoperate. We introduced many subtle bugs in the kernel by accidentally calling paged code from the wired code. In hindsight, we should have made the separation explicit. [81]

We used the terms ‘‘ring 0’’, ‘‘hardcore’’ and ‘‘supervisor’’ fairly interchangeably. I'll use ‘‘supervisor’’ because that is vague and all-encompassing and implies ‘‘large’’, which it most certainly was. We did not use the term ‘‘kernel’’ until we had to define a ‘‘security kernel’’ subset of Multics. The common use of ‘‘kernel’’ is probably Unix-based and thus post-dates the early Multics work. [82]

Ring 0 is used by the supervisor. By design, none of the code in ring 0 comes from the file system hierarchy. However, as mentioned above, the supervisor is segmented and a special ‘‘segment loader’’ read the kernel segments off of the boot tape into main memory. A few of these kernel segments were visible thru the file system, and those that were had to be ‘‘patched’’ into it. After initialization, the supervisor mounted the file system and made it available to users. [83]

Ring 1 holds ‘‘extensions’’ to the supervisor; vendor-supplied code that is less privileged than the kernel but more privileged than all users. Ring 1 is within the ‘‘security kernel’’ and is therefore able to avoid the restrictions of the Access Isolation Mechanism (AIM). Message segments are the prime example of a ring 1 subsystem; this facility provides queues that are, in fact, built on top of segments. It does its own AIM validation so that messages in the queues can be of various AIM levels. [AIM is not presently described in this paper, but is a nondiscretionary access control mechanism that implements government-style security levels and classifications.] [84]

Rings 2 and 3 are available to run code that is less privileged than the supervisor but more privileged, and protected from, the normal users. Rings 2 (and higher) are outside the security kernel and cannot escape AIM. (These rings have been used for database managers, TCP/IP drivers, etc. Some of these products also have ring 1 components, making them truly “3-ring circuses”). [85]

By convention, ring 2 is used for vendor-supplied applications and ring 3 is available for user-supplied applications. [86]

Typical user programs run in ring 4. [87]

Rings 5 is available to run less-privileged user programs. A Honeywell-sponsored scout troop was placed in this ring (in a undoubtedly vain attempt to protect the system!) [88]

Rings 6 and 7 have almost no direct access to system functions; they must call user-supplied code in inner rings to accomplish any useful work. [89]

There were originally 4 access permission modes (read, execute, write, and append), later simplified to 3 modes (read, execute, and write). (Append mode was meant to permit a segment to grow but in the end it was easier to permit all segments to grow). [90]

Rings and modes combine to enforce a total ordering of objects, access permissions, and agents (executing programs / processes). [91]

The term “file” was used in several senses over the years. Originally, “file” and “segment” were used interchangeably. This made sense when all files were less than one megabyte and would fit in a segment. For the moment, I will use file in this sense. Later, I will explain the other uses. [92]

All files are mapped into segments when open, and all segments have a single set of _ring brackets_ associated with them. Thus, ring brackets are per file (or per segment). The terms file and segment are interchangeable on Multics. [93]

Ring brackets are a list of 3 ring numbers that specify which rings can use any of the specified access permission modes. [94]

Access permission modes are per user; see the section on “access control lists” for more information.[1] [95]

Generally, if ring N has permission to read, execute, or write a segment, then all rings less than N have the same permission. In other words, rings are nested, monotonic, levels of privilege. (This is not quite true for execute permission, as will be explained later). [96]

Given a file F, ring brackets [R1, R2, R3], where R1<=R2<=R3, and access permission modes read, execute, and write, then the following statements apply. [97]

1. File F can be written by a program executing in rings 0 thru R1. The range 0:R1 is called the write bracket.
2. File F can be read by a program executing in rings 0 thru R2. The range 0:R2 is called the read bracket.

---

1. Early versions of Multics permitted both access permission modes and ring brackets to be per user, i.e., specified in the ACL entry for a file. This proved to be overly-general and hard to use and understand. It was simplified to the scheme described here.

3. File F can be executed in rings R1 thru R2. The range R1:R2 is called the execute bracket. Additionally, File F can be called from rings R2+1 thru R3 (assuming that R2 The ring number of the calling segment (program), and the ring brackets of the called segment (program) are used to calculate the new ring number of execution.
   - ❏ If the caller's ring is within the execute bracket, file F will execute in the same ring as the caller.
   - ❏ If the caller's ring is within the gate bracket, the process will switch to ring R2 before executing file F. This is an ‘‘inward call’’. It increases privilege.
   - ❏ If the caller's ring is below the gate bracket, the process will switch to ring R1 before executing file F. This is an ‘‘outward call’’. It lowers privilege.
4. The Multics hardware and operating system implement an inward call by using the standard subroutine call and return sequence. Both the ring crossing to raise privilege at the call, and the ring crossing to lower privilege at the return are implemented in hardware. I believe that an outward call is implemented via an operating system trap; Multics treats this as a ‘‘goto’’ rather than a call; no corresponding return is permitted (except via a gate, of course). This was because the procedure return location would have been stored in the outer ring, where it would be susceptible to tampering.
5. The segment descriptor word (SDW) for a gate contains a field (‘‘call limiter’’) that defines the offset of the maximum address that can be called by an outer ring. The idea was to put all entrypoints at the beginning of a segment, in a transfer vector, and set the call limiter to the end of the transfer vector. This would prevent less-privileged code from jumping into the middle of privileged code, and would have permitted any procedure segment to be a gate. By the time this facility was available on the 6180, Multics had already standardized on putting gate transfer vectors and associated code in a segment of their own, so the hardware mechanism was used only for the first purpose. We had special assembly language macros to create gates; we never fixed the compilers or the binder to let ordinary programs be gates.

Supervisor segments are typically bracketed (W:0, R:0, G:0). Code segments have read and execute permission; data segments have read and write permission. Only code executing in ring 0 (most privileged) can access these segments. [98]

User segments are typically bracketed (4, 4, 4). This means that all rings from 0 thru 4 have the same read/write access to the segment. Only ring 4 code can execute the segment. Rings 5 thru 7 have no access to it. [99]

Code that was needed by all rings, such as the ‘‘programmed operators’’ used by compiled code, was bracketed (0, 7, 7). Since the execute bracket covers all rings, any call to this segment would execute in the same ring as the caller. [100]

Gates into ring 0 are (0, 0, 5). Code in rings 1 thru 5 can call the gate and begin executing in ring 0. Code in ring 0 that calls the gate remains in ring 0. Rings 6 and 7 have no access to it. The most common gate is ‘‘hcs_’’ (hard core supervisor). [101]

Gates into ring 1 are (1, 1, 5). Code in rings 2 thru 5 can call the gate and begin executing in ring 1. Code in ring 1 remains in ring 1. I think that Multics would trap if ring 0 code called the gate; this would be interpreted as an error in this particular case. As before, rings 6 and 7 are in ‘‘outer space’’ and have no access. The gate for the message segment facility mentioned earlier is ‘‘message_segment_’’. [102]

Data that is read/write to the kernel but only readable to normal users is (0, 5, 5). This was not common (I can't recall any examples), as we did not want user code to be dependent on the layout of supervisor data structures. [103]

Interestingly, segments shared by the supervisor and the user world have to be ‘‘patched’’ into the file system at system boot time, as they exist in the supervisor's address space before the file system is mounted. This resulted in such segments having two separate descriptors (SDWs) in processes where users initiated them — the (0,0,0) descriptor with the same segment number in every process, usable only by the supervisor, and the other descriptor, resulting from the user initiation and usable by the user (and potentially by the supervisor). Although there would be two descriptors, both describe the same segment via the same page table.[1] [104]

In the 645 implementation, rings were implemented by keeping different descriptor segments for each ring, faulting (trapping) on every ring crossing, and switching descriptor segments. In the 6180 implementation, we were able to have a single descriptor segment per process, and have the hardware validate all calls and returns, whether within the same ring or across rings. Much faster. [105]

In a future version of this paper I will describe ring validation levels and how they are used to ‘‘weaken’’ pointers and pointer registers. [106]

2.6 Access Control Lists

Segments and directories are protected by access control lists. This is a list of users (or classes of users) and their access permission modes. The permission modes for directories are ‘‘status’’, ‘‘modify’’, and ‘‘append’’. The permission modes for segments are ‘‘read’’, ‘‘execute’’, and ‘‘write’’. [107]

Status permission grants the ability to list the contents of a directory. Even though a directory is implemented as a ring 0 segment, Multics never permitted users to have direct access. All access is mediated via gates into ring 0. [108]

No access is required to traverse (pass thru) a directory on your way to a lower-level directory. Thus, denying someone status permission at a high level still meant you could grant status permission to a lower level directory. Note that Unix does it the other way; a source of frustration to many Unix users. [109]

Multics enforces no permanent relation between the access granted to a directory and that granted to subdirectories. A new subdirectory is initialized to have the same access lists as its parent (...I think...) but anyone with permission to modify the subdirectory can change its access lists to any setup they desire. [110]

Modify permission grants the ability to delete objects from a directory (segments, sub-directories, ACLs, links, etc.), or to change the attributes of any object. [111]

Append permission grants permission to add new objects. [112]

An ACL name is of the form ‘‘Person.Project.Tag’’ where Person is a person name (or the special name ‘‘anonymous’’), Project is a project name, and Tag is a 1-character code that indicates the type of process (‘‘a’’ for interactive, ‘‘m’’ for absentee (batch)). [113]

So, the access that I would have to my own home directory would be: [114]

```
sma       Green.*.*
```

and the access to a PL/I source file would be: [115]

```
rw        Green.*.*
```

and to an executable program would be: [116]

1. I am presently researching an interesting security breach of Multics involving this dual-descriptor design. I will include this story in a future revision.

```
re        Green.*.*
```

and the access to the system ‘‘whotab’’ listing all logged-in users would be: [117]

```
r         *.*.*
```

I could deny Bob Frankston's absentee jobs (batch jobs) from accessing my files, while still permitting him to access them interactively via: [118]

```
n         Frankston.*.m
r         *.*.*
```

Directories have an Initial Access Control List (Initial ACL or IACL) that is used to initialize the ACL of all new objects. This replaced the Common Access Control Lists (CACLs) of a previous version of Multics. I will explain this in more detail in a future version of this paper. [119]

2.7 Multi-Segment Files

Eventually we needed files larger than a segment (and we did not or could not raise the segment size), so someone invented ‘‘multi-segment files’’ (MSFs). In an MSF, the named, visible object is actually a directory. Within the directory are segments named ‘‘0’’, ‘‘1’’, ‘‘2’’, etc., that hold the data. The otherwise-unused bitcount attribute of the directory specifies the number of segments and serves as the flag to indicate to Multics that this directory is really an MSF. The list command (used to list the contents of directories) listed an MSF as a file, and totaled up the sizes properly. [120]

At first, only a couple of programs knew the convention. Dan Bricklin (yes, the same person who co-wrote VisiCalc with the above-mentioned Bob Frankston), wrote a utility subroutine called msf_manager_ that managed the details and so it became a lot easier for programs to deal with them. (I can remember changing the compiler listing generator to use msf_manager_ so we could get listings longer than a segment). But it was still a kludge because you had to deal with the file a segment at a time. Some users of MSFs could handle MSF components (segments) that were less than a full segment; some could not. Some commands were never fixed to know how to deal with MSFs; they continued to see them as directories, and so you were always at risk. [121]

You could get a lot of useful work done without ever needing MSFs, and they were not very hard to use (post msf_manager_) when you did need them. The primary problem was that command written to work fine on a single segment (e.g, the sort command, which would sort lines in a file) would simply not work on MSFs. [122]

2.8 “True” Files

Still later, true files and file I/O came along. This was after I left in 1980, so I am not familiar with it. I believe that it was finally possible to do classical record-at-a-time file I/O that resulted in direct (though undoubtedly cached) disk I/O. [123]

If anyone in net-land understands this addition and would care to send me a description, I will include it in a future edition of this paper. [124]

3 Commentary and Critique

Segmentation was considered a very useful feature in Multics, not a curse as in some existing systems (no names, but you know which ones I mean). In fact, I can make an even stronger statement. Multics without segmentation would not be Multics. All of the major Multics mechanisms used segmentation; and they were all intertwined. The limit of 1 megabyte on the maximum size of a segment was not, in most cases, a practical constraint, though I believe it was a fairly serious architectural constraint that would have greatly complicated the design and mechanisms had Multics lasted longer. [125]

Segmentation was used: [126]

- ❑ to implement and organize the supervisor
- ❑ to map files and directories directly onto segments
- ❑ to implement dynamic linking
- ❑ by a called procedure to find its linkage (static) pointer
- ❑ to find per-process data (kept at the base of the stack)
- ❑ to control access to data and programs
- ❑ to assign data (segments) to rings
- ❑ for temporary storage
- ❑ for stacks, linkage (static), and areas (heaps)
- ❑ and so on...

Due to a bug in the 6180 hardware (in the character string unit??) some instructions did not work properly against the last word of the last page in a segment. Therefore, Multics limited segments (in software) to 1044480 bytes (255 pages) instead of the full 1048576 bytes (256 pages). [127]

The limit of about a megabyte per segment was not a problem for most ordinary files (source programs, object programs, executables, directories, etc.) but was a problem for anyone building a database, or generating an object code listing of a large source program. Multi-segment files (described above) were our first attempt to get around this limit. Raising the maximum segment size would have been a much better, though still not perfect, solution. [128]

By design, addressing within a segment ‘‘wrapped around’’. If you indexed off the end of a segment, you wrapped around to the beginning of the same segment. This was nice in that programming bugs that generated wild offsets stayed within the intended segment. It was pretty rare to have a bug that randomly affected the segment number portion of an address. [129]

MSFs were the one case where it would have been nice to be able to concatenate segments into larger entities. No matter how big you set the segment size, some application is always going to want more. Concatenating segments would be simple if incrementing the word offset past the last word carried into the segment number. You wouldn't want this by default, but with a bit in the SDW you could control which segments were concatenated and which were not. This was never implemented; too bad, because I feel it would have solved the segment-size problem elegantly and forever. [130]

The fact that segment numbers were temporary handles and no good across processes was an occasional problem. It meant that you couldn't store segment numbers in permanent files and expect them to be useful to anyone else, or even to later instances of yourself. It was possible to get around this limitation within PL/I by defining a PL/I area on top of a segment and by using offset variables instead of pointer variables. Offsets are relative to the base of the area but can be used in the same source-level contexts as pointers (although their underlying implementation was slightly more expensive than pointers). But because offsets were not implemented by the first two generations of PL/I compilers, and because Multics PL/I had special builtin functions to manipulate the segment number or offset value of a pointer, programmers used bit(18) variables as an ad-hoc offset type. [131]

There was no native support for any kind of a permanent pointer with segment numbers; i.e., one that could last across initiations or process boundaries. [132]

Inventing an efficient way to have 8 levels of protection was a major accomplishment of Multics. I wish the people who design microprocessors would copy it. You probably don't need 8 levels; 4 would. But 2 is certainly too few. I guess the microkernel notion of running the supervisor in another address space gets around this problem, but at the expense of message passing. Multics can do a direct procedure call to a higher privilege level, with the full generality of high-level language argument passing, without taking a trap or invoking mediation code. The return from higher privilege (lower ring) to lower privilege (higher ring) is also direct. [133]

The particular instruction set used by Multics was never very important. The addressing is important. Multics needs segmentation, paging, rings, and pointers than can be used (indirected thru) or faulted, fixed up, and restarted. This last mechanism is used to implement dynamic linking. We could probably do without it, but it sure simplified things to have it. [134]

Various people tried to complicate or improve on Multics' virtual memory design, but, to me, Multics found the right balance of features versus complexity. [135]

3.1 Multics Hardware Funnies

The index registers can be used both as classical index registers (in address modification), and on their own as nearly general-purpose (though short) registers. Many opcodes are provided to compare, add, subtract, perform Boolean functions, etc., in index registers or between an index register and memory. I think the only truly useful things you couldn't do in them was shift, multiply or divide. I'm sure they were great for assembly language programmers, but the limited functionality and size forced the compilers to use them only for indexing. [136]

Initially, the designers of the PL/I compiler chose to use only the accumulator/quotient (AQ) register pair to hold arithmetic values. Eventually, the compiler was improved to be able to put short-precision index values into index registers. Looking back, it is obvious that the hardware would have benefited greatly from more general-purpose registers. [137]

Probably 2/3 of the 512 opcodes were devoted to index registers operations. Many opcodes were needed because the index register number was a part of the opcode when an index register was the target of the operation. The compiler probably used 10% of the opcodes and ignored the rest. I'm sure whole sections of the hardware logic never got used by Multics, or got used rarely. [138]

The address modification flavors were baroque. It would make a modern-day instruction set designer scream. Again, I don't think Multics used more than a handful of the possible modes. As explained above, some of the addressing modes flat-out didn't work on Multics because the tally words had to be in the same segment as the data. Pretty hard to do when the segment was read-only, or writeable but shared between processes. This was due to trying to retrofit segmentation onto an existing design and being unable to make it co-exist with some of the existing modes. [139]

3.2 Multics Software Funnies

Insufficient thought was given to the interactions between the maximum file name length (32), the maximum directory depth (16), the possibility of multiple file names, and the maximum pathname length (168). Multics had a tendency to canonicalize the pathname to an object by concatenating all of the primary names together. As a result, by using short addnames and long primary names you could trick Multics into creating directories that would overflow the pathname length limit. Some student hackers were able to crash the MIT Multics this way. [140]

Similar problems existed for the concatenation of the maximum-length person name (22), the maximum-length project name (9), and the tag (1), to form the process group ID (32, but 34 are needed because there are also two periods in the ID). As before, some student hackers were able to crash the "answering service" process on the MIT Multics by registering a user named Funkelhausersteinweitz.SIPBADMIN! [141]

3.3 A Few Words on Portability

I don't have the papers handy to check, but my memory is that the designers of Multics chose PL/I as an implementation language for productivity reasons, not for portability reasons. Even if they had intended some amount of portability, the project itself never followed thru. I would be surprised if a re-reading of the papers showed any concern for this issue. People familiar with computers of the 1990s are used to being able to purchase PCs that are "100% compatible" and to run shrink-wrapped software, but the 1960s were different. The design work on Multics started about the same time that the ANSI standards committees started working on standard versions of COBOL and PL/I. Unix hadn't even been invented yet; its designers were hard at work on Multics itself. [142]

So, while Multics is written in PL/I, and there are arguably many different PL/I compilers around for current machines, how come we just can't port Multics to modern hardware? [143]

First, Multics is still owned by Bull (the successor to Honeywell), and Bull has shown no interest in further development work, or even publication, of the source code. Holding onto the rights to Multics and its source code is a way of holding onto the (dwindling) Multics customer base. [144]

Second, Multics had its own, idiosyncratic, PL/I compiler. It is highly doubtful whether the great body (literally millions of lines) of Multics source code will ever be compiled or compileable by any other PL/I compiler. The Multics PL/I language is the full ANSI PL/I language with many extensions, and, no doubt, many "features" that Multics depends upon. There are very few implementations of full PL/I available today. (Even Stratus, where I work today, has never seen fit to extend our subset-G PL/I compiler to a full language PL/I compiler, even though we have the rights to a full-language version of our compiler). [145]

Lastly, and arguably most important, the Multics source code is rife with declarations that "know" that the word size is 36 bits, that characters are 4 per word and bytes are 9 bits, that the pointer size is 72 bits, that stacks grow up, that character strings are a maximum of 1MB, and so on. Assumptions about the underlying hardware implementation are strewn throughout the source code. [146]

If Bull ever relents and permits the source code to be published and used as intended, the only economically and technically viable way to run it (in my not-so-humble opinion) would be to emulate the Bull (ex-Honeywell) hardware itself, right down to the huge, complex machine conditions. Given the relative cycle times of modern RISC microprocessors, a decent emulation might actually run Multics faster than the current iron. (This alone is probably reason enough for Bull to hang onto the rights to the source code; they are keeping their Multics customers locked up). [147]

The “right way” to pay tribute to Multics is to discuss it, argue over it, and to have it be a source of inspiration for future operating systems. As a usable piece of technology it is stuck in the 1980s (if not the 1970s) and will most likely stay there forever. [148]

Revision History [149]

- Written 18 May 1993
- Revised 8 July 1993 (Corrected errors, added 80306 and NOS/VE comparisons)
- Revised 1 July 1994 (Corrected major errors in ring bracket descriptions, added figures 1-4, expanded treatment significantly, added PRIMOS description).
- Revised 17 September 1994 (Corrected description of unpaired base registers and unpaged segments; added many corrections from reviewers, moved comparisons with other systems to separate files).

Contact: Paul_Green@vos.stratus.com [150]