Lecture 7: The Multics Virtual Memory

Mothy Roscoe and Joe Hellerstein

September 21, 2005

1 Background and History

Multics:

> **M**any
> **U**nnecessary
> **L**arge
> **T**ables
> **I**n
> **C**ore
> **S**imultaneously

Alternatively, "**Mult**iplexed **I**nformation and **C**omputing **S**ervice".

History

Canonical example of Fred Brooks' "second system" syndrome: the first system is minimal and conservative, the second system includes any and all cool ideas and attempts to do everything.

The first system in this case was the MIT CTSS (*Compatible Time-Sharing System*) - the first true time-sharing system (1959-1965).

- Interative working on-line
- Online storage of information
- Tape-drives for both storage and swapping
- 4 consoles (110-baud teleprinters), 2 tape drives per user.
- No protection
- Enabled lots of new ideas in interactive computing: editors, interactive debugging, etc.

Multics mostly developed 1963-1970, ran through the mid 1980s, but never really finished.

- New hardware: Honeywell 645 initially
- Everything built from scratch (mostly written in PL/1)
- Large. Very large.
- Very complex.
- Never really took the world by storm: eclipsed by Unix (written as a reaction to it).
- Nevertheless: pioneered many concepts which are now taken for granted in operating systems, and also many others which are rarely seen these days.

One way to approach Multics

One difference between good Systems Research and good Software Engineering: investigate the consequences of an idea by taking it to its logical extreme in the design of a system.

- Bound to break: you don't necessarily want to build a product this way
- But when it does, you'll understand the nature of the idea (including in more moderate, applicable forms) much more deeply.
- Think of Multics in this way: sharing, protection, addressing.

Alternative is LPU (*Least Publishable Increment*: safe, more applicable short-term, much less fun, and with little long-term impact.

New ideas in Multics:

- Single-level store (memory-mapped file system, or really no files at all, only memory segments).
- Dynamic linking
- On-line reconfiguration
- Shared-memory multiprocessing
- Three-level physical store
- Hierarchical namespace
- Almost all the system written in a high-level language (PL/1)

2 The Virtual Memory System

Main goals at the time

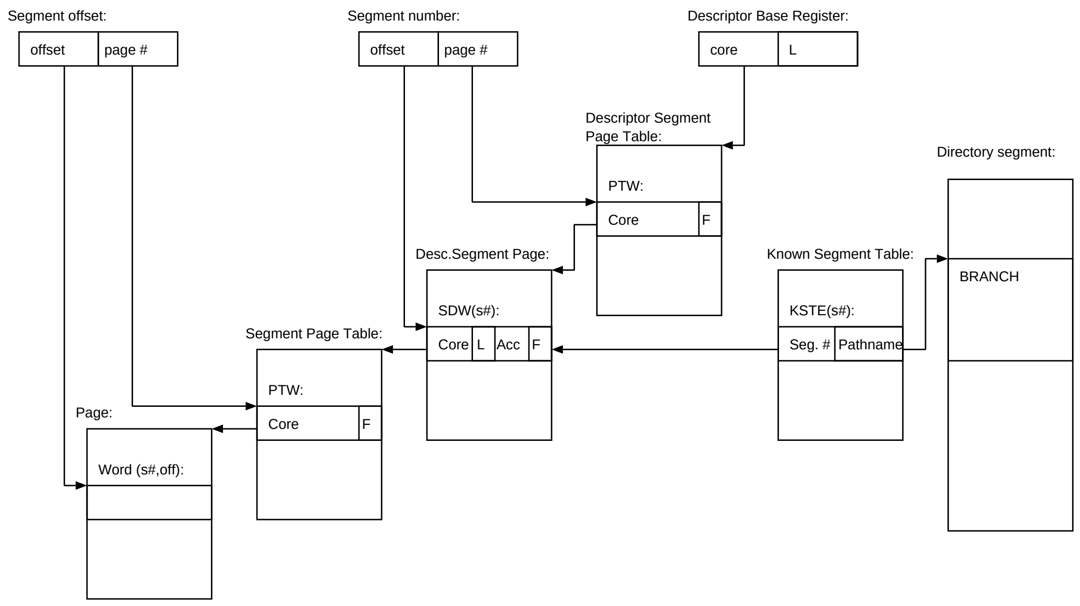
- Single-level store
  - Segments as the basic units of storage, sharing and protection.
  - Referred to in 2 ways: symbolic name (pathname) and virtual address (segment number) within a process.
  - Segments are variable size.
- Fine-grained sharing - in particular, sharing of all code in the system at the level of *procedures and variables*. Extreme.
- Dynamic linking: all references to variables and procedures are resolved *late*, in fact, when first referenced by the program.
- Autonomy of address spaces: each process should be able to resolve a different set of segment names.

Why would you want to do this?

- Many users ⇒ many copies of programs / procedures ⇒ want sharing
- Lots of packages / shared components ⇒ where to load package into address space?
  - Could always load at same address (c.f. early Unix dynamic linking, doesn't scale with # packages)
  - ... or could have autonomous address spaces.
- Leads to requirement for dynamic linking
- Problem: when is a package available for linking? Answer: always (combine VM and file system)

Basics

- Process = address space
- Address space = segment table (actually KST & DS)
- Generalized address: segment index + offset
- Per-address space segment numbers
- Everything except the DS Page Table Segment is (potentially) paged underneath
- Translation cached using 2 lookaside buffers

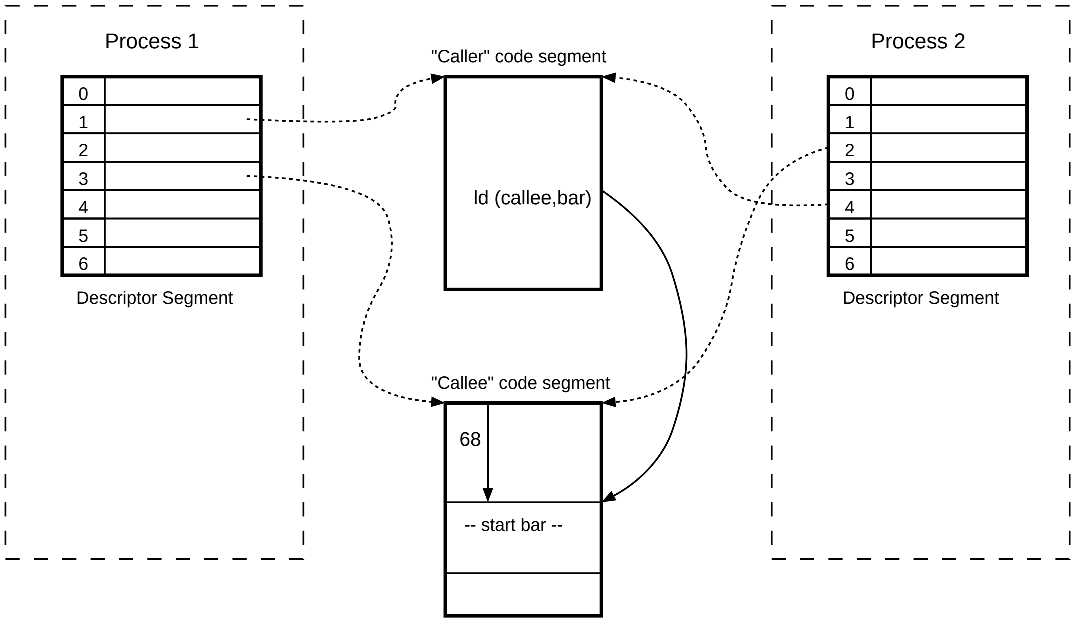


See the paper for details on algorithms for setting these up.

3 Dynamic Linking

- All cross-references to symbols are initially, well, symbolic (filename, symbol name)
- *link trap* on first reference
- Trap *rewrites* the symbolic address with real generalized address for the current process
- Subtle business: see below

Link trap is taken for *every* different external cross reference - a good example of “lazy evaluation”.



Problem 1: Per-process table for *links* (inter-segment references)

Every process has a different mapping from symbolic addresses to generalized addresses.

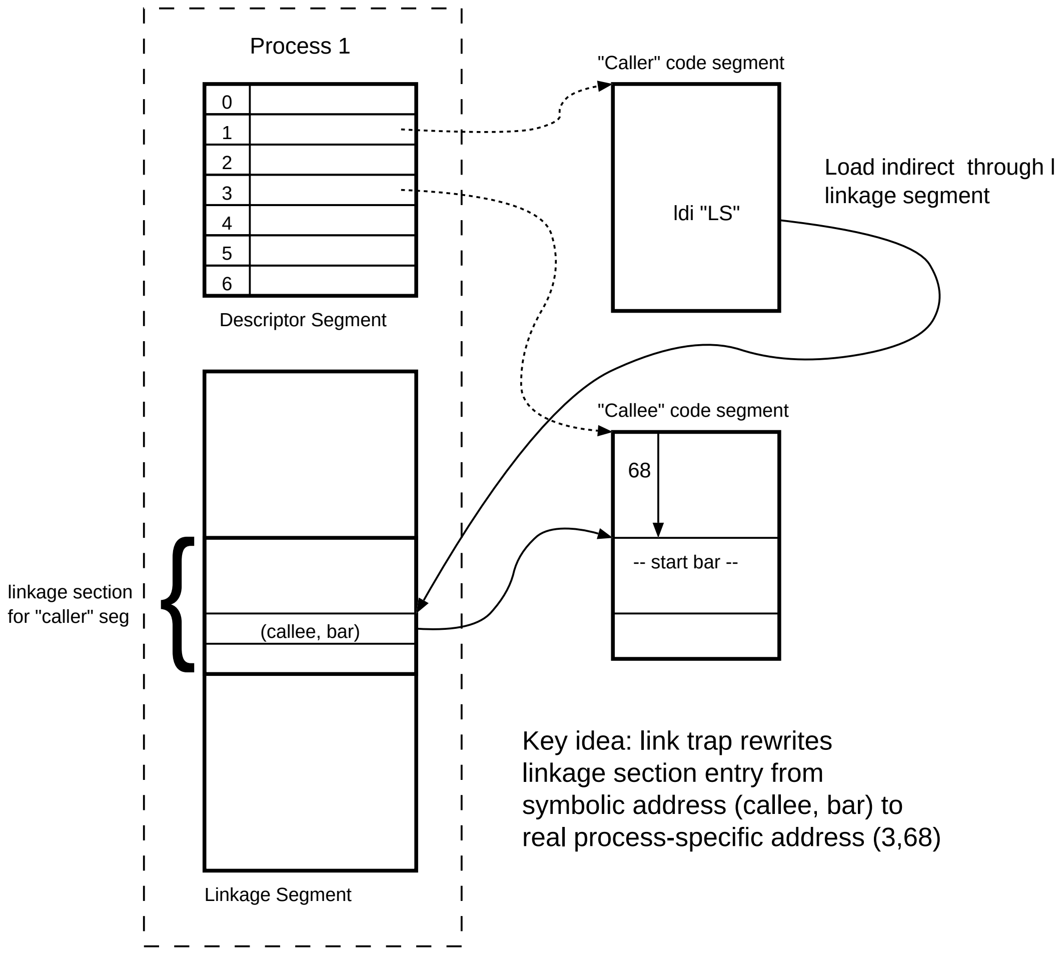
Linkage section: all external references for a code segment (both imports *and* exports).

- Initially a *symbol table*: (callee filename, symbol name)
- becomes an indirection table over time: (segment #, offset)
- One linkage section in a process for each code segment

Linkage segment: Segment to hold all linkage sections for a process

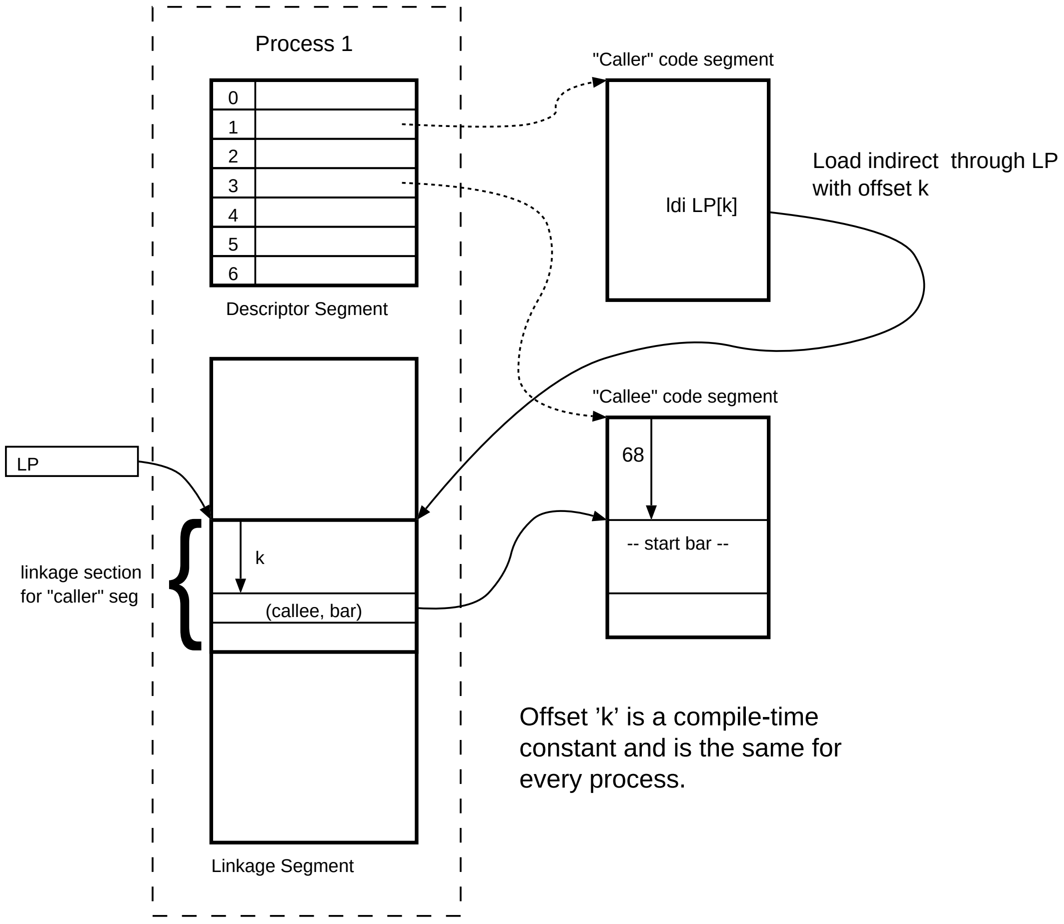
- All rewrites occur within this (private, per-process) segment.

Process layout with linkage segment:


Process 1
0
1
2
3
4
5
6
Descriptor Segment
"Caller" code segment
ldi "LS"
Load indirect through l linkage segment
"Callee" code segment
68
-- start bar --
linkage section for "caller" seg
(callee, bar)
Linkage Segment
Key idea: link trap rewrites linkage section entry from symbolic address (callee, bar) to real process-specific address (3,68)

Problem 2: How do you find the right place in the linkage segment?

- The `caller` code knows nothing about the structure of the (per-process) linkage segment. How does it find the right entry in the linkage section for `caller` to rewrite it?
- Note that the format of the linkage *section* for a code segment (like `caller`) is fixed and known when that segment is compiled.
- Solution: process has a register `LP` (linkage pointer) which points to start of linkage section for the current segment (i.e., the segment where the program counter currently is).
- Compiled code in the current segment can use a fixed offset from this to find `(callee, bar)`.

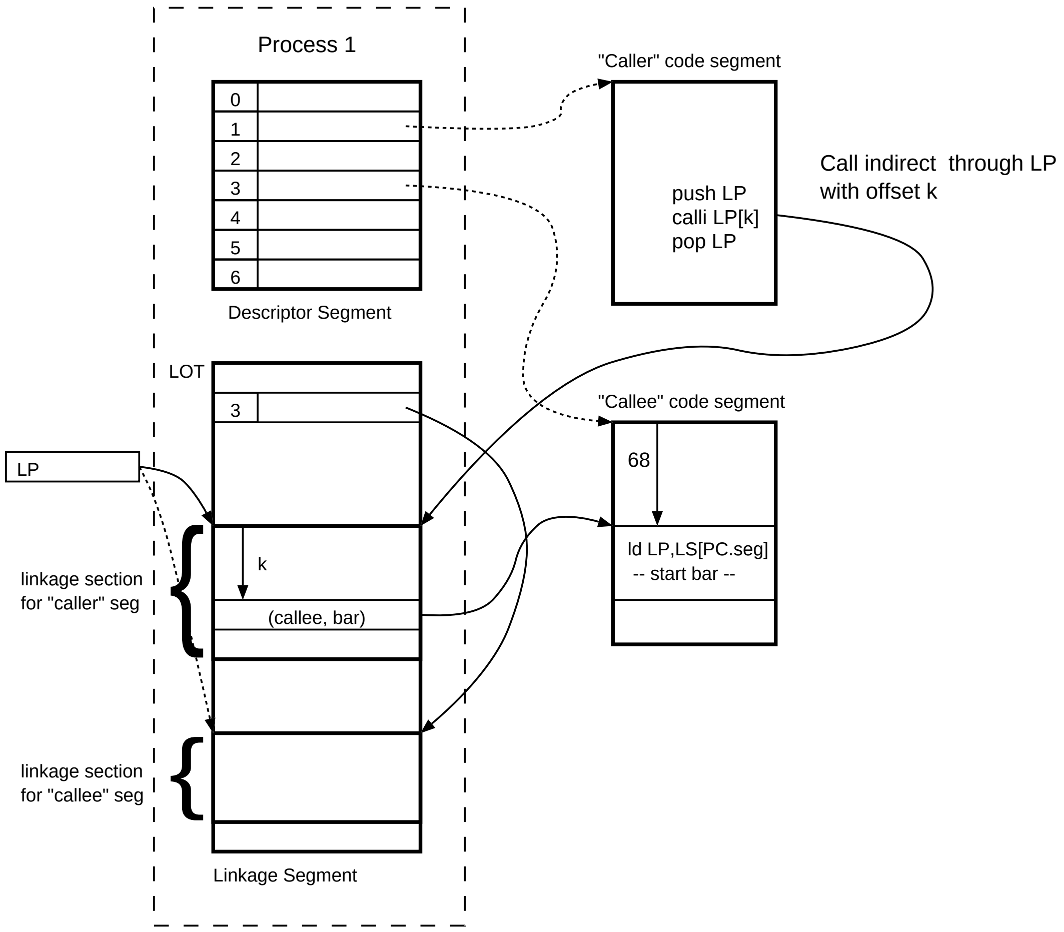

Process 1
0
1
2
3
4
5
6
Descriptor Segment
"Caller" code segment
ldi LP[k]
Load indirect through LP with offset k
"Callee" code segment
68
-- start bar --
LP
linkage section for "caller" seg
k
(callee, bar)
Linkage Segment
Offset 'k' is a compile-time constant and is the same for every process.

Problem 3: LP needs to be reloaded on each inter-segment call. How do we know what value?

The LP must be reloaded after the call (since we need to old one to look up the offset!). Hence, the first few instructions at each entry point load the LP. But from where?

Enter the LOT:

- The Linkage Offset Table, located at the start of the Linkage Segment.
- Indexed by segment number of the code segment (which we can get from the PC).
- Holds the appropriate LP value for the linkage section for that code segment.
- After subroutine jump into a new segment, push the old LP, load new one from LOT(PC segment #).



Historical note: In the published papers, a different approach is described. The linkage segment contains *code* for each entry point: load the LP, and jump to a real procedure. The caller indirects through its own linkage section into a jump table in the callee's linkage section. Has the advantage that the LP isn't reloaded for inter-segment calls.

Resolving symbolic names

So how do we convert `(callee, bar)` to (3, 68)?

1. We now know that the target entry that needs to be rewritten is at LP[k], which points to somewhere in `caller`'s linkage section. We first need to distinguish symbolic addresses from real addresses. This is done with a bit set in the indirect address (at LP[k]). The *link trap* occurs exactly if this bit is set, otherwise it is a real address and the load/call works just fine. Assuming we have a trap, we need to rewrite the target with the real address.

2. First we have to see if this segment is already loaded. We check the known segment table (KST) for this. The KST is an array of pointers indexed by segment number (much like the descriptor segment, but separate). The pointers point to a data structure that contains the name, length, and attributes of the segment. The linker scans the KST to see if any of the names match `callee`, if so we have found the right segment, goto step 4.

3. The segment is not already known (at least not for this process). The symbolic name, `callee`, is relative to a specific path (following some PATH-like "search rules") and thus fully specifies the file to be initiated. The supervisor (i.e. kernel) assigns it a (local) segment number, and adds it to the KST. If the file was already loaded for another process (i.e. it is an *active* segment, its directory entry ("branch" info) points to its page table, which is shared by the new process. If the file is not already mapped, there will be a "missing segment fault", which will actually create its page table; later there will be page faults as well . . .

4. At this point, we know that `callee` equals segment 3, but we don't know the value of `bar`, which is an exported symbol for the file `callee`. Note that the value for `bar` is just an offset into the segment. Since Multics allows independent recompilation, we need to get the current offset of `bar` from the symbol table, which is at the head of the code segment `callee`. Since the operation is read-only, it is OK to read it directly from `callee`, which means we don't have to copy this part of the symbol table into the linkage section. Like the KST, the symbol table is just an array of symbols that is scanned linearly for a match. No match implies a linking error (presumably non-recoverable). At this point we know that `bar` means offset 68, and we know the full generalized address.

5. The only remaining trick is that if the symbol is an entrypoint (i.e. this is a call), then we must load the new LP value from the LOT, using the new PC to tell us the new segment number.

6. Rewrite the symbolic address and restart the faulting instruction.

Conclusion

Why all this dynamic linking rigmarole?

- The original UNIX approach was to statically link each binary: each address space had 3 segments (text, data, stack). Unit of sharing was entire program binary.
- Later, a growable heap was added (bss) separate from the data segment.
- Then, shared libraries (X Windows probably the forcing factor): each library had to be loaded at an address fixed at compile time . . .
- Some systems (e.g. Sun Labs' Spring) introduced copy-on-write dynamically loadable object files which were patched at load for each address space. Then they had to cache partially patched images for performance.
- RISC machines' *calling conventions* by now passed "frame pointers" and other context anyway in registers . . . .
- Eventually: UNIX link-loading now posesses all the complexity of Multics, except that segments are emulated by allocating ranges of virtual addresses.

Moral? . . .

A note from Paul Green about using Multics:

> *By the way, the dynamic linking feature of Multics worked extremely well and was a very popular capability of the system. It was another reason that application developers enjoyed working on the system. The output of the compilers was directly executable. You didn't need to use the static linker ("binder") until you were ready to optimize your work and give it to someone else. Many development projects kept all of the separately-compiled object files lying around in a common directory for everyone to search. You put your own private directory, containing the pieces that you were changing, earlier in your search rules. The very first time you started up the subsystem, things would kind of chug along while the segments got activated and the linking took place. But then the test / debug / edit / compile cycle was fast and efficient for the rest of the day. Every now and then something would screw up and a program wouldn't get unsnapped correctly, or you'd do something dumb and wipe out the process and have to start over, but in general, it all worked very well.*