Why am I lecturing about Multics?
  Origin of many ideas in today's OSes
  Motivated UNIX design (often in opposition)
  Motivated x86 VM design
  This lecture is really "how Intel intended x86 segments to be used"

Multics background
  design started in 1965
    very few interactive time-shared systems then: CTSS
  design first, then implementation
  system stable by 1969
    so pre-dates UNIX, which started in 1969
  ambitious, many years, many programmers, MIT+GE+BTL

Multics high-level goals
  many users on same machine: "time sharing"
  perhaps commercial services sharing the machine too
  remote terminal access (but no recognizable data networks: wired or phone)
  persistent reliable file system
  encourage interaction between users
    support joint projects that share data &c
  control access to data that should not be shared

Most interesting aspect of design: memory system
  idea: eliminate memory / file distinction
  file i/o uses LD / ST instructions
  no difference between memory and disk files
  just jump to start of file to run program
  enhances sharing: no more copying files to private memory
  this seems like a really neat simplification!

GE 645 physical memory system
  24-bit phys addresses
  36-bit words
  so up to 75 megabytes of physical memory!!!
    but no-one could afford more than about a megabyte

[per-process state]
  DBR
  DS, SDW (== address space)
  KST
  stack segment
  per-segment linkage segments

[global state]
  segment content pages
  per-segment page tables
  per-segment branch in directory segment
  AST

645 segments (simplified for now, no paging or rings)
  descriptor base register (DBR) holds phy addr of descriptor segment (DS)
  DS is an array of segment descriptor words (SDW)
  SDW: phys addr, length, r/w/x, present
  CPU has pairs of registers: 18 bit offset, 18 bit segment #
    five pairs (PC, arguments, base, linkage, stack)
  early Multics limited each segment to 2^16 words

    thus there are lots of them, intended to correspond to program modules
  note: cannot directly address phys mem (18 vs 24)
  645 segments are a lot like the x86!

645 paging
  DBR and SDW actually contain phy addr of 64-entry page table
  each page is 1024 words
  PTE holds phys addr and present flag
  no permission bits, so you really need to use the segments, not like JOS
  no per-process page table, only per-segment
    so all processes using a segment share its page table and phys storage
    makes sense assuming segments tend to be shared
    paging environment doesn't change on process switch

Multics processes
  each process has its own DS
  Multics switches DBR on context switch
  different processes typically have different number for same segment

how to use segments to unify memory and file system?
  don't want to have to use 18-bit seg numbers as file names
  we want to write programs using symbolic names
  names should be hierarchical (for users)
    so users can have directories and sub-directories
    and path names

Multics file system
  tree structure, directories and files
  each file and directory is a segment
  dir seg holds array of "branches"
    name, length, ACL, array of block #s, "active"
  unique ROOT directory
  path names: ROOT > A > B
  note there are no inodes, thus no i-numbers
  so "real name" for a file is the complete path name
    o/s tables have path name where unix would have i-number
    presumably makes renaming and removing active files awkward
    no hard links

how does a program refer to a different segment?
  inter-segment variables contain symbolic segment name
  A$E refers to segment A, variable/function E
  what happens when segment B calls function A$E(1, 2, 3)?

when compiling B:
    compiler actually generates *two* segments
    one holds B's instructions
    one holds B's linkage information
    initial linkage entry:
      name of segment e.g. "A"
      name of symbol e.g. "E"
      valid flag
    CALL instruction is indirect through entry i of linkage segment
    compiler marks entry i invalid
    [storage for strings "A" and "E" really in segment B, not linkage seg]

when a process is executing B:

```
    two segments in DS: B and a *copy* of B's linkage segment
    CPU linkage register always points to current segment's linkage segment
    call A$E is really call indirect via linkage[i]
    faults because linkage[i] is invalid
    o/s fault handler
      looks up segment name for i ("A")
      search path in file system for segment "A" (cwd, library dirs)
      if not already in use by some process (branch active flag and AST
knows):
        allocate page table and pages
        read segment A into memory
      if not already in use by *this* process (KST knows):
        find free SDW j in process DS, make it refer to A's page table
        set up r/w/x based on process's user and file ACL
        also set up copy of A's linkage segment
      search A's symbol table for "E"
      linkage[i] := j / address(E)
      restart B
    now the CALL works via linkage[i]
      and subsequent calls are fast

how does A get the correct linkage register?
  the right value cannot be embedded in A, since shared among processes
  so CALL actually goes to instructions in A's linkage segment
    load current seg# into linkage register, jump into A
    one set of these per procedure in A

all memory / file references work this way
  as if pointers were really symbolic names
  segment # is really a transparent optimization
  linking is "dynamic"
    programs contain symbolic references
    resolved only as needed -- if/when executed
  code is shared among processes
  was program data shared?
    probably most variables not shared (on stack, in private segments)
    maybe a DB would share a data segment, w/ synchronization
  file data:
    probably one at a time (locks) for read/write
    read-only is easy to share

filesystem / segment implications
  programs start slowly due to dynamic linking
  creat(), unlink(), &c are outside of this model
  store beyond end extends a segment (== appends to a file)
  no need for buffer cache! no need to copy into user space!
    but no buffer cache => ad-hoc caches e.g. active segment table
  when are dirty segments written back to disk?
    only in page eviction algorithm, when free pages are low
  database careful ordered writes? e.g. log before data blocks?
    I don't know, probably separate flush system calls

how does shell work?
  you type a program name
  the shell just CALLs that program, as a segment!
  dynamic linking finds program segment and any library segments it needs
  the program eventually returns, e.g. with RET
```

```
  all this happened inside the shell process's address space
  no fork, no exec
  buggy program can crash the shell! e.g. scribble on stack
  process creation was too slow to give each program its own process

how valuable is the sharing provided by segment machinery?
  is it critical to users sharing information?
  or is it just there to save memory and copying?

how does the kernel fit into all this?
  kernel is a bunch of code modules in segments (in file system)
  a process dynamically loads in the kernel segments that it uses
  so kernel segments have different numbers in different processes
    a little different from separate kernel "program" in JOS or xv6
  kernel shares process's segment# address space
    thus easy to interpret seg #s in system call arguments
  kernel segment ACLs in file system restrict write
    so mapped non-writeable into processes

how to call the kernel?
  very similar to the Intel x86
  8 rings. users at 4. core kernel at 0.
  CPU knows current execution level
  SDW has max read/write/execute levels
  call gate: lowers ring level, but only at designated entry
  stack per ring, incoming call switches stacks
  inner ring can always read arguments, write results
  problem: checking validity of arguments to system calls
    don't want user to trick kernel into reading/writing the wrong segment
    you have this problem in JOS too
    later Multics CPUs had hardware to check argument references

are Multics rings a general-purpose protected subsystem facility?
  example: protected game implementation
    protected so that users cannot cheat
    put game's code and data in ring 3
    BUT what if I don't trust the author?
      or if i've already put some other subsystem in ring 3?
    a ring has full power over itself and outer rings: you must trust
  today: user/kernel, server processes and IPC
    pro: protection among mutually suspicious subsystems
    con: no convenient sharing of address spaces

UNIX vs Multics
  UNIX was less ambitious (e.g. no unified mem/FS)
  UNIX hardware was small
  just a few programmers, all in the same room
  evolved rather than pre-planned
  quickly self-hosted, so they got experience earlier

What did UNIX inherit from MULTICS?
  a shell at user level (not built into kernel)
  a single hierarchical file system, with subdirectories
  controlled sharing of files
  written in high level language, self-hosted development

What did UNIX reject from MULTICS?
```

files look like memory
instead, unifying idea is file descriptor and read()/write()
memory is a totally separate resource
dynamic linking
instead, static linking at compile time, every binary had copy of libraries
segments and sharing
instead, single linear address space per process, like xv6
(but shared libraries brought these back, just for efficiency, in 1980s)
Hierarchical rings of protection
simpler user/kernel
for subsystems, setuid, then client/server and IPC

The most useful sources I found for late-1960s Multics VM:

1. Bensoussan, Clingen, Daley, "The Multics Virtual Memory: Concepts and Design," CACM 1972 (segments, paging, naming segments, dynamic linking).
2. Daley and Dennis, "Virtual Memory, Processes, and Sharing in Multics," SOSP 1967 (more details about dynamic linking and CPU).
3. Graham, "Protection in an Information Processing Utility," CACM 1968 (brief account of rings and gates).