Lecture 9 (Implementation of Virtual Memory Issues & segmentation)

Implementation of virtual memory issues:

- Instruction Backup
- Locking Pages in Memory
- Backing Store
- Separation of Policy and Mechanism

Page-related work for OS

- Process creation (determine size, allocate memory for page table, allocate space in swap area, initialize swap area with program text and data, record info about swap area and page table in process table)
- Process execution (reset MMU for new process and flush TLB, make new process' page table current, possibly bring some process' pages into memory)
- Page fault (read hardware register to determine virtual address which causes fault; determine the page and locate it on disk; find available page frame; read page into page frame; back up program counter)
- Process termination (release process's page table, pages and disk space)

Page Fault Handling

1. The computer hardware traps to the kernel and program counter (PC) is saved on the stack. Current instruction state information is saved in CPU registers.
2. An assembly program is started to save the general registers and other volatile information to keep the OS from destroying it.

3. Operating system finds that a page fault has occurred and tries to find out which virtual page is needed. Some times hardware register contains this required information. If not, the operating system must retrieve PC, fetch instruction and find out what it was doing when the fault occurred.
4. Once virtual address caused page fault is known, system checks to see if address is valid and checks if there is no protection access problem.
5. If the virtual address is valid, the system checks to see if a page frame is free. If no frames are free, the page replacement algorithm is run to remove a page.
6. If frame selected is dirty, page is scheduled for transfer to disk, context switch takes place, fault process is suspended and another process is made to run until disk transfer is completed.
7. As soon as page frame is clean, operating system looks up disk address where needed page is, schedules disk operation to bring it in.
8. When disk interrupt indicates page has arrived, page tables are updated to reflect its position, and frame marked as being in normal state.
9. Faulting instruction is backed up to state it had when it began and PC is reset. Faulting is scheduled, operating system returns to routine that called it.
10. Assembly Routine reloads register and other state information, returns to user space to continue execution.

Kernel traps & save PC in stack → general registers saved → find virtual page with fault → check validness of address (0=kill, 1=find frame to evict) → if selected frame is dirty then initial page write on disk & context switch & another process run → page clean →schedules disk operation & run another process → page arrived → frame in normal state → faulting process is scheduled → registers reloads.

Instruction backup

When a program references a page that is not in memory, the instruction causing the fault is stopped partway through and a trap to the OS occurs. After the OS has fetched the page needed, it must restart the instruction causing the trap.

Autoincrementing: one or more registers increment while instruction executed. OS must decrement register before restarting instruction.

Solution: have **internal register** in which program counter is copied. Second register may store some information about incremental/decremental

---

Locking Pages in memory

OS try to remove from memory page which currently access by I/O device.

Solution: lock the pages engaged in I/O. Locking a page is often called **pinning** it in memory

Another solution: all I/O to kernel buffers and then copy the data to user pages later

---

Backing Store

Swap partition used when size of physical memory < size of program for run

To allocate page space on the disk UNIX systems have swap partition on the disk or on separate disk. Block number relative to the the start of used partition.

System is booted ⇒ partition = single empty entry

First process started ⇒ chunk of partition size of first process is reserved & remaining area reduced by that amount

New process started ⇒ chunks of swap partitions equal in size to their core images. Finish ⇒ disk space freed.

Disk address's of process swap area is associated with each process. This info in process table.

Swap area initialized before the process starts: copy entire process image to swap partition(brought in as needed) or load entire process in memory(page out as needed). Calculating address to write a page: add the offset of page within the virtual address space to start of swap area.

Better to allocate separate swap areas for the text, data and stack, as data & stack can grow. Also possible to allocate disk space for each page & deallocate when swapped back.

(a) allocate fixed size per process (b)allocated chunk can grow

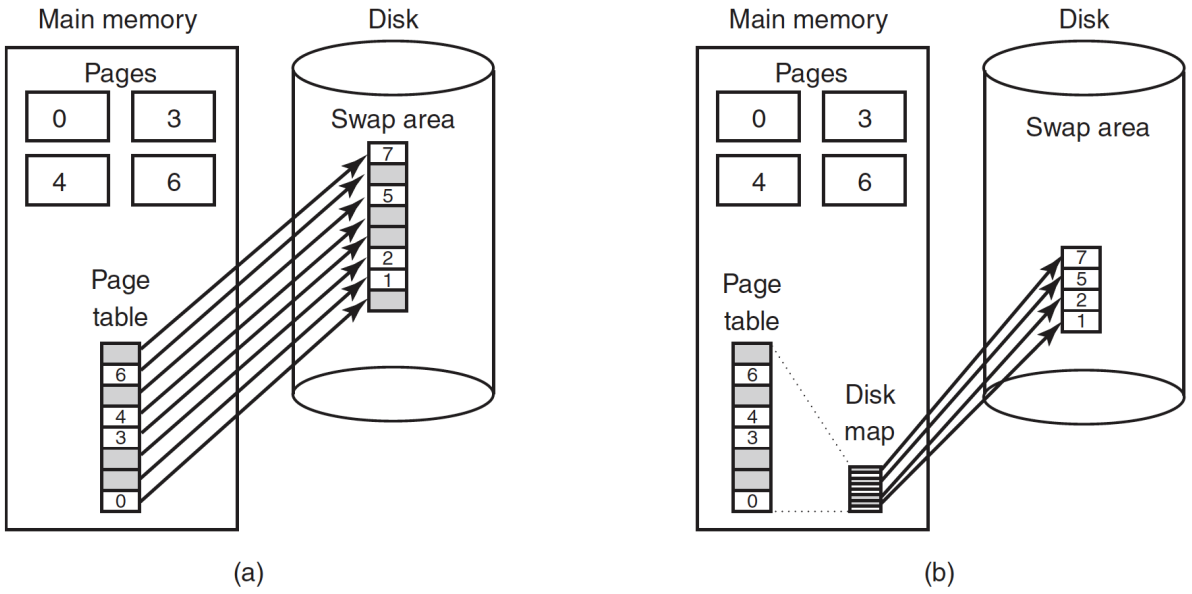


Figure 3-28. (a) Paging to a static swap area. (b) Backing up pages dynamically.

Separation of Policy and Mechanism

Memory management system divided into:

- Low-level MMU handler - machine dependent code, all details
- Page fault handler (part of kernel) - machine-independent code
- External pager (user space) - determines policy, runs as a user process.

Process start → external pager notified to set up process' page map & allocate backing store on the disk → process runs → map new objects into address space → external pager notified → occur some events:

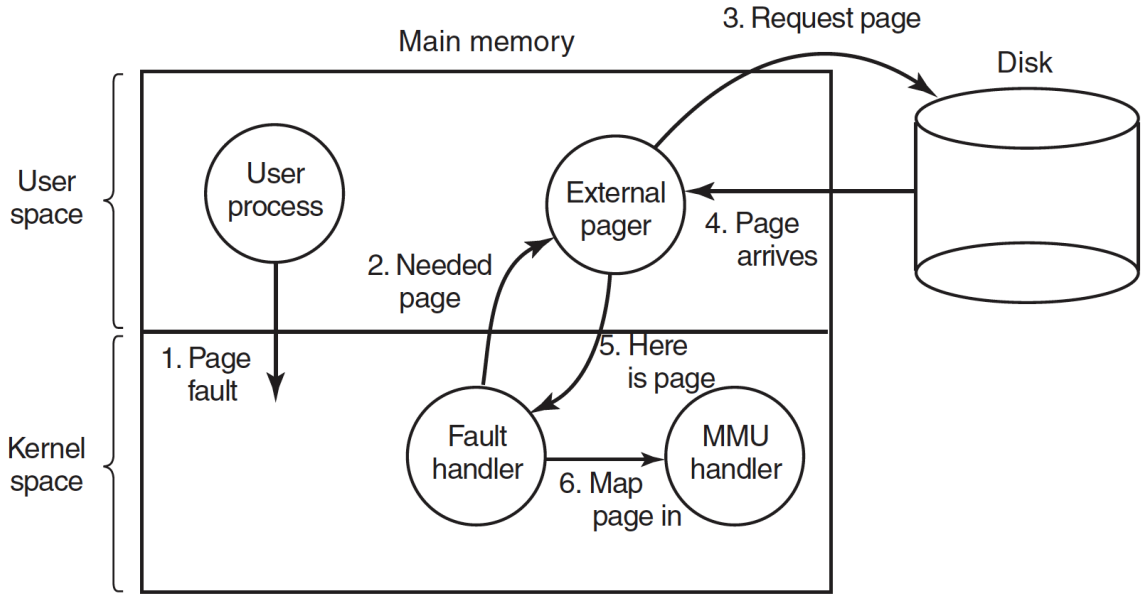


Figure 3-29. Page fault handling with an external pager.

Page fault → page fault handler determines page & send message to pager → pager request page from disk → pager copies it to its own address space → pager to handler where page is → handler unmaps page from pager's address space & MMU put it at right place in user space.

Pros: flexibility, modular code. **Cons**: extra overheads of crossing user-kernel boundary

---

Segmentation

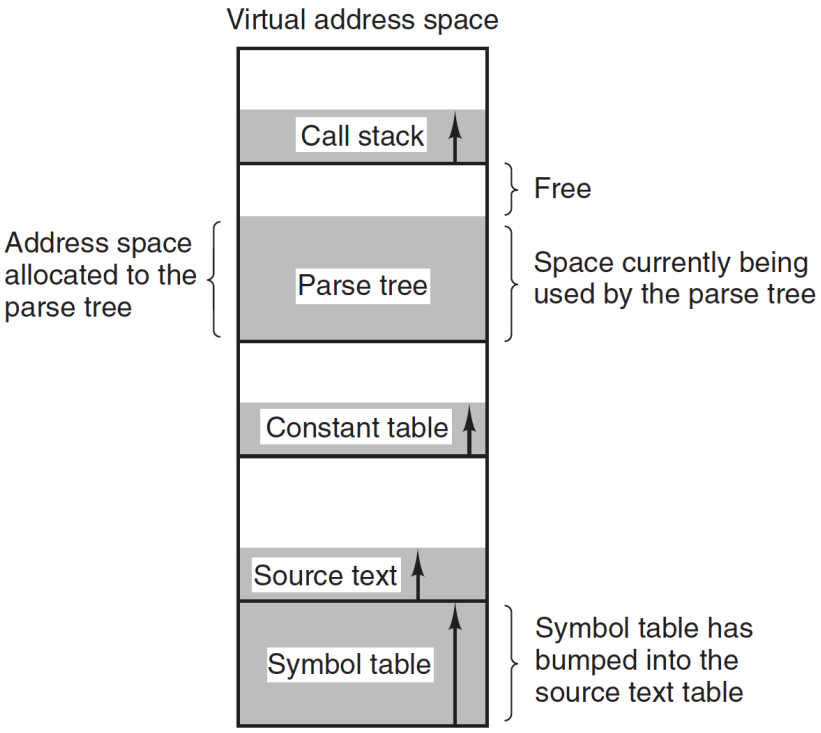


Figure 3-30. In a one-dimensional address space with growing tables, one table may bump into another.

Segments - completely independent address spaces. Segment = linear sequence if addresses from to max_value. Segment length may change while execution. Segments grow & shrink independently. Address - two-part = segment number + address within segment

Advantages: linking of procedures are simple, no need to recompile all segments after modification in one, help to sharing libraries, diff segments = diff kinds of protection

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Implementation of pure segmentation

External fragmentation (checkerboarding) - after the system has been running for a while, memory will be divided up into a number of chunks, some containing segments and some containing holes. Usually external fragmentation dealt with compaction.

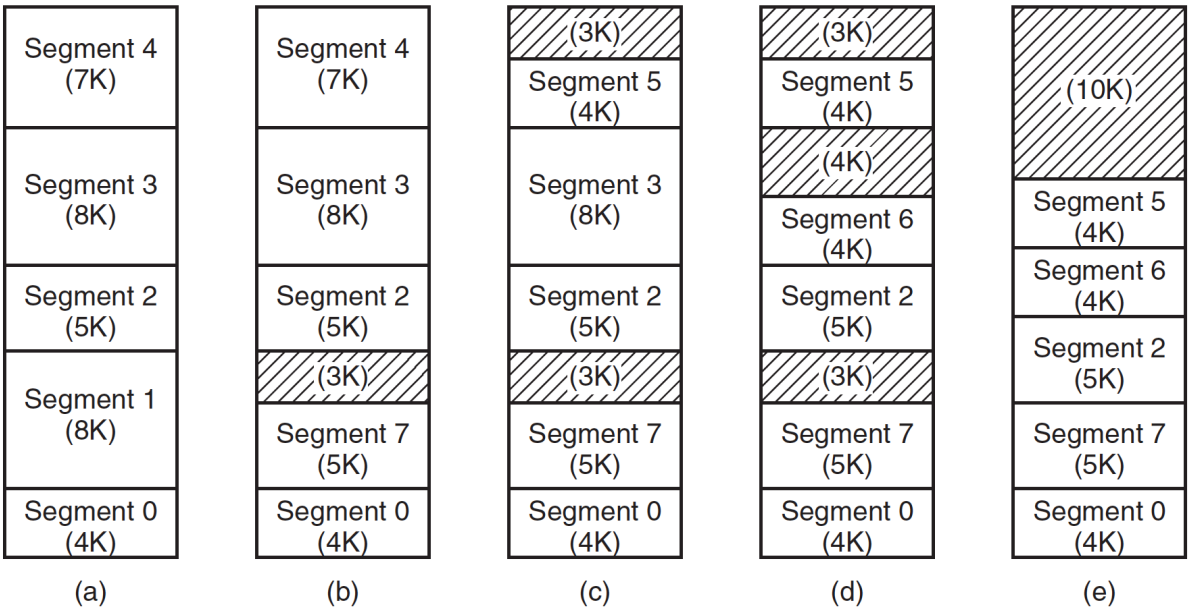


Figure 3-33. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

Segmentation with paging

If segments are large ⇒ only those pages of a segment that are actually needed have to be around.

MULTICS: treat each segment as virtual memory & page it. Combining advantages of paging & segmentation.

Segment table per program, descriptor per segment. Segment descriptor - indicates whether segment in main memory or not. Part of segment in memory⇒segment in memory⇒page table in memory. Segment in memory ⇒descriptor = 18-bit pointer to page table.

Virtual address = segment number + page number + offset within page = 18 + 6 + 10 = 34

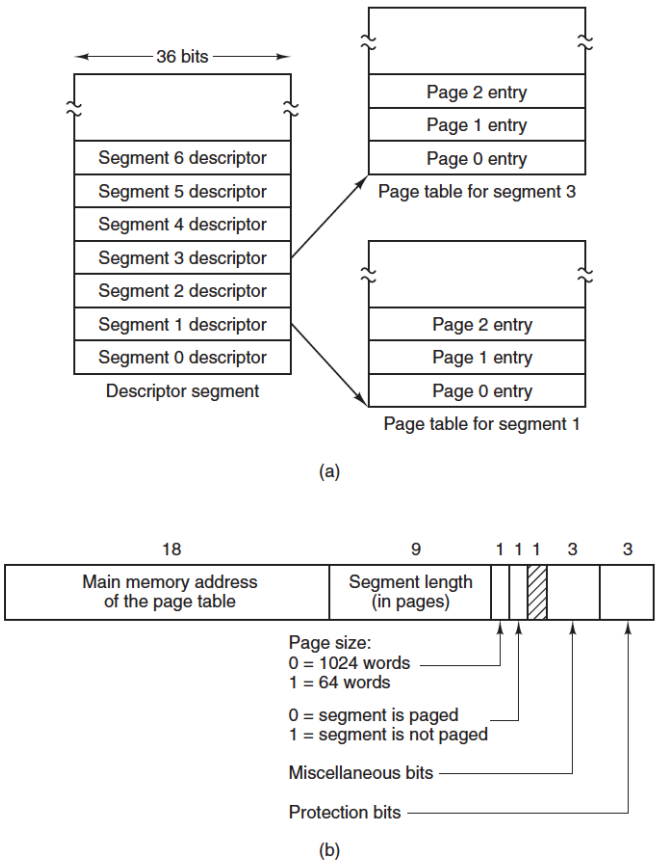


Figure 3-34. The MULTICS virtual memory. (a) The descriptor segment pointed to the page tables. (b) A segment descriptor. The numbers are the field lengths.

Conversion virtual address into physical address: segment number → segment descriptor → page number(segment in memory) or segment fault → page frame → page origin + offset = main memory address→r/w operations

TLB: 16-word & can search entries in parallel. Address first checked for presence in TLB, if so then directly from TLB.

INTEL X86:

x86-64: Segmentation is considered obsolete and is no longer supported, except in legacy mode.

x86-32: It has 16K segments, each holding up to 1 billion 32-bit words:

- **LDT(local descriptor table)** describes segments local to each program: code, data, stack... Each program has its own LDT

- **GDT(global descriptor table)** describes system segments including OS itself. Shared by all programs.

Access to segment: program loads selector into 1 of 6 machine' segment registers → corresponding descriptor fetched from LDT or GDT → descriptor stored in microprogram registers →

CS register holds selector for code & **DS register** holds selector for data

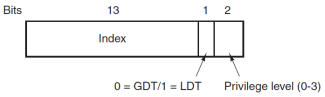


Figure 3-38. An x86 selector.

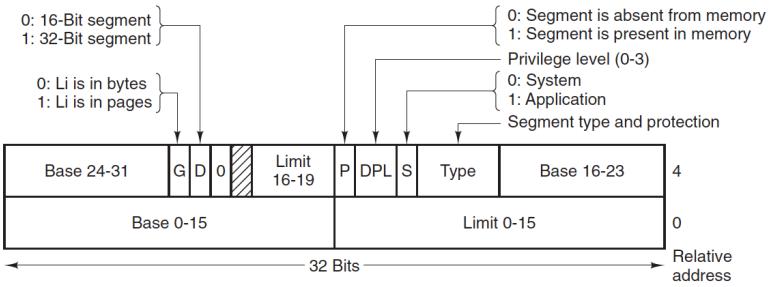
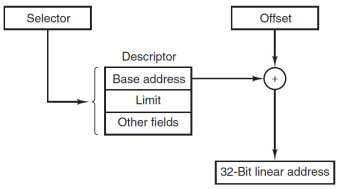
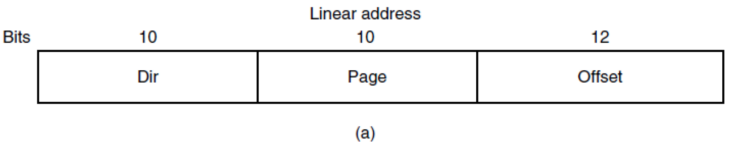


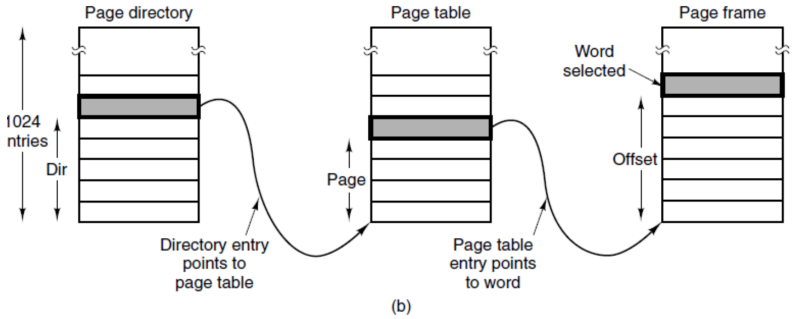
Figure 3-39. x86 code segment descriptor. Data segments differ slightly.

Conversion to linear address:





(a)



(b)

If paging is disabled, the linear address is interpreted as the physical address, otherwise interpreted as a virtual address and mapped onto the physical address using page tables