

# Memory Management

Week 09 – Lecture

Implementation Issues &  
Segmentation

# Team

- Instructor
  - Giancarlo Succi
- Teaching Assistants
  - (Vladimir Ivanov)
  - Luiz Araujo (also Tutorial Instructor)
  - Nikita Lozhnikov
  - Nikita Bogomazov

# Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013  
Prentice-Hall, Inc.
- and customised for the needs of this course.
- Additional input for the slides are detailed later

# Implementation Issues

- Involvement with Paging
- Page Fault Handling
- Instruction Backup
- Locking Pages in Memory
- Backing Store
- Separation of Policy and Mechanism

# OS Involvement with Paging (1)

- Four situations for paging-related work:
  - Process creation
  - Process execution
  - Page fault
  - Process termination

# OS Involvement with Paging (2)

- Process creation:
  - **Determine** how large the program and data will be (initially) and **create** a page table for them
  - **Allocate** and **initialize** space in memory for the page table
  - **Allocate** space in the swap area on disk so that when a page is swapped out, it has somewhere to go
  - **Initialize** the swap area with program text and data so that when the new process starts getting page faults, the pages can be brought in
  - **Record** information about the page table and swap area on disk in the process table

# OS Involvement with Paging (3)

- Process execution:
  - **Reset** the MMU for the new process and **flush** TLB to get rid of traces of the previously executing process
  - **Make** the new process' page table to be current, usually by copying it or a pointer to it to some hardware register(s)
  - Optionally, **bring** some or all of the process' pages into memory to reduce the number of page faults initially

**MMU (Memory Management Unit):** maps the virtual addresses onto the physical memory addresses

**TLB (Translation Lookaside Buffer):** a small hardware device for mapping virtual addresses to physical addresses without going through the page table.

# OS Involvement with Paging (4)

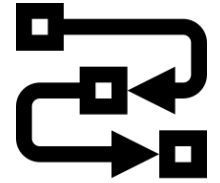
- Page fault:
  - Read out hardware registers to **determine** which virtual address caused the fault
  - From this information, **compute** which page is needed and **locate** that page on disk
  - **Find** an available page frame in which to put the new page, evicting some old page if need be
  - **Read** the needed page **into** the page frame
  - **Back up** the program counter to have it point to the faulting instruction and let that instruction execute again



# OS Involvement with Paging (5)

- Process termination:
  - **Release** the page table, its pages, and the disk space that the pages occupy when they are on disk
  - If some of the pages are shared with other processes, the pages in memory and on disk can be released only when the last process using them has terminated

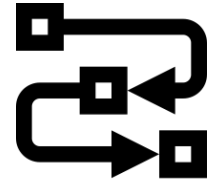
# Page Fault Handling (1)



Sequence of events on a page fault:

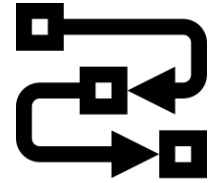
- The hardware **traps** to kernel, **saving** program counter on stack.
- An assembly code routine is started to **save** general registers and other volatile info
- OS discovers page fault has occurred, tries to **discover** which virtual page needed
- Once virtual address caused fault is known, system **checks** to see if address valid and the protection consistent with access

# Page Fault Handling (2)



- If frame selected (to be replaced) is dirty, page is scheduled for **transfer** to disk, context switch takes place, suspending faulting process
- As soon as frame is clean, OS **looks up** disk address where needed page is and **schedules** disk operation to bring it in
- When disk interrupt indicates that page has arrived, tables are **updated** to reflect position, and frame **marked** as being in normal state

# Page Fault Handling (3)



- Faulting instruction is **backed up** to state it had when it began and program counter is reset
- Faulting process is **scheduled**, operating system returns to routine that called it
- Routine **reloads** registers and other state information, returns to user space to **continue** execution

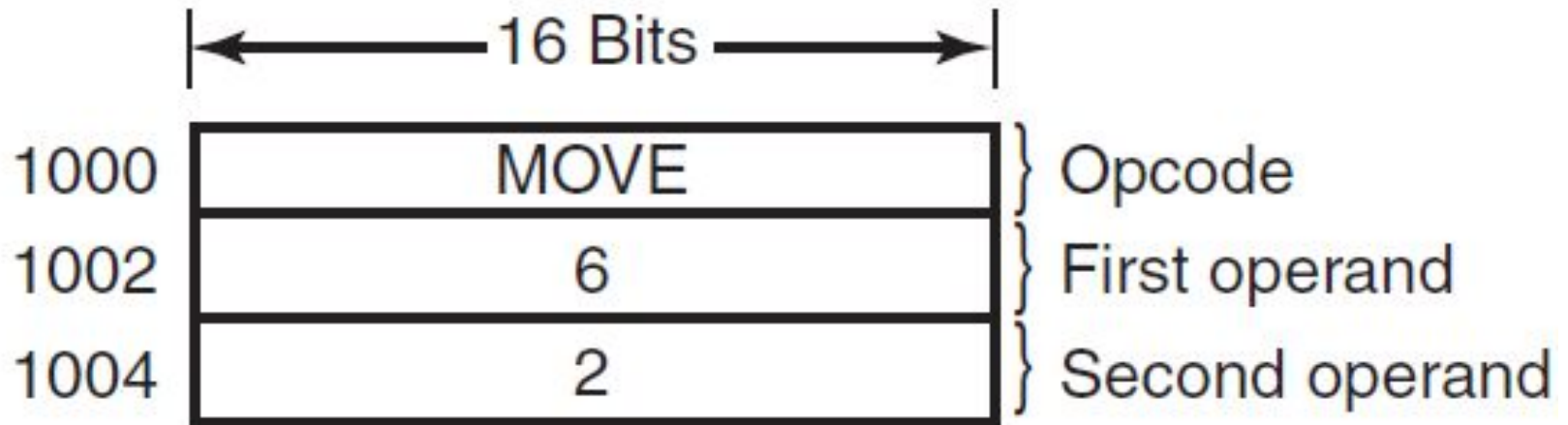
# Instruction Backup (1)

- 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

# Instruction Backup (2)

- Let's consider a CPU which is used in embedded systems and has instructions with two addresses
- An example of such an instruction:

MOVE.L #6(A1), 2(A0)



# Instruction Backup (3)

- The instruction starts at address 1000 and makes three memory references: the instruction word and two offsets for the operands
- To restart the instruction the OS must be able to detect where the first byte of the instruction is
- It is not easy since the value of the program counter at the time of the trap depends on which operand faulted and how the CPU's microcode has been implemented

# Instruction Backup (4)

- In our case the program counter might be 1000, 1002, or 1004 at the time of the fault
- It is frequently impossible for the OS to determine unambiguously where the instruction began
- If the program counter is 1002 at the time of the fault, the OS has no way of telling whether the word in 1002 is a memory address associated with an instruction at 1000 (e.g., the address of an operand) or an opcode



# Instruction Backup (5)

- Some addressing modes use **autoincrementing**: one or more registers are incremented when an instruction is executed
- The increment may be done before or after the memory reference, depending on the details of the microcode
  - In former case, the OS must decrement the register before restarting the instruction
  - Otherwise, increment **must not be undone** by the OS
- Autodecrement mode also exists and causes a similar problem

# Instruction Backup (6)



- A possible solution is to have a **hidden internal register** into which the program counter is copied just before each instruction is executed
- The second register may store the information about **registers that have already been autoincremented** or autodecremented and by how much
- Given this information, the OS can unambiguously undo all the effects of the faulting instruction so that it can be restarted

# Locking Pages in Memory (1)

- Consider a process that has just issued a system call to **read from some file** or device into a buffer within its address space
- While waiting for the I/O to complete, the process is **suspended** and **another process** is allowed to run. This other process **gets a page fault**

# Locking Pages in Memory (2)

- If the **paging algorithm is global**, there is a small, but nonzero, chance that the page containing the I/O buffer will be chosen to be removed from memory
- If an I/O device is currently in the process of doing a Direct Memory Access (DMA) transfer to that page, the part of the data will be written in the buffer where they belong, and **part of the data will be written over the just-loaded page**

# Locking Pages in Memory (3)

- One solution to this problem is to **lock pages engaged in I/O** in memory so that they will not be removed
- Locking a page is often called **pinning** it in memory
- **Another solution** is to do all I/O to kernel buffers and then copy the data to user pages later


# Backing Store (1)

- Where on disk a page selected for replacement is put?  
Answer: page space allocated on the disk
- Most UNIX systems have a **special swap partition** on the disk or even a separate disk
- This partition doesn't have a normal file system. Instead, block numbers relative to the start of the partition are used → This eliminates all the overhead of converting offsets in files to block addresses

# Backing Store (2)

- When the system is booted, this swap partition is empty and is **represented** in memory as a single entry giving its origin and size
- When the first process is started, a chunk of the partition area the size of the first process is reserved and the remaining area reduced by that amount
- **New processes are assigned chunks of the swap partition** equal in size to their core images. As they finish, their disk space is freed

# Backing Store (3)

- Each process has associated to it a disk address utilized as swap area. This information is kept in the process table
- Calculating the address to write a page to is as simple as adding the offset of the page within the virtual address space to the start of the swap area 
- The swap area must be initialized before the process starts:
  - One way is to copy the entire process image to the swap area, so that it can be *brought in as needed*
  - The other way is to load the entire process in memory and let it be *paged out as needed*



# Backing Store (4)

- The processes can increase in size after they start. Both the data area and the stack can grow
- It may be better to reserve **separate swap areas** for the text, data and stack and allow each of these areas to consist of more than one chunk on the disk (Fig. 3-28a)
- It is also possible to **allocate disk space for each page** when it is swapped out and deallocate it when it is swapped back in. However, there must be a table per process telling for each page on disk where it is (Fig. 3-28b)

← See Figure 3-28(b) in TB

# Backing Store (5)

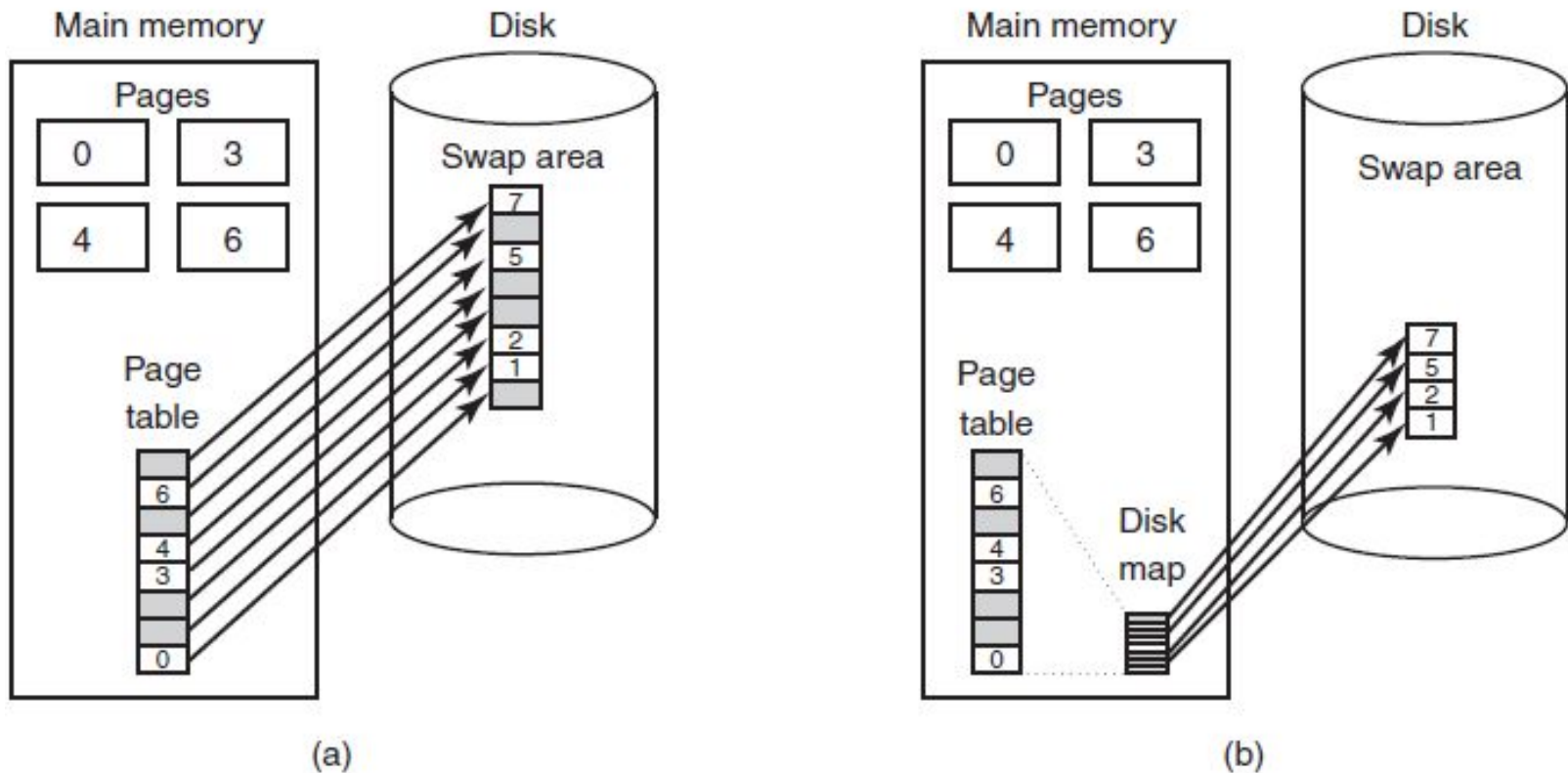


Figure 3-28. (a) Paging to a static swap area.  
(b) Backing up pages dynamically.

# Backing Store (6)

- Having a fixed swap partition is not always possible. In this case, one or more large, preallocated files within the normal file system can be used as it is done in Windows.
- Since the program text of every process came from some (executable) file in the file system, the executable file can be used as the swap area
- Since the program text is generally read only, when program pages have to be evicted from memory, they are just discarded and read in again from the executable file when needed
- Shared libraries can also work this way

# Separation of Policy and Mechanism (1)

- Tool for managing the complexity of any system
- Memory management system is divided into three parts:
  - A low-level MMU handler
  - A page fault handler that is part of the kernel
  - An external pager running in user space

# Separation of Policy and Mechanism (2)

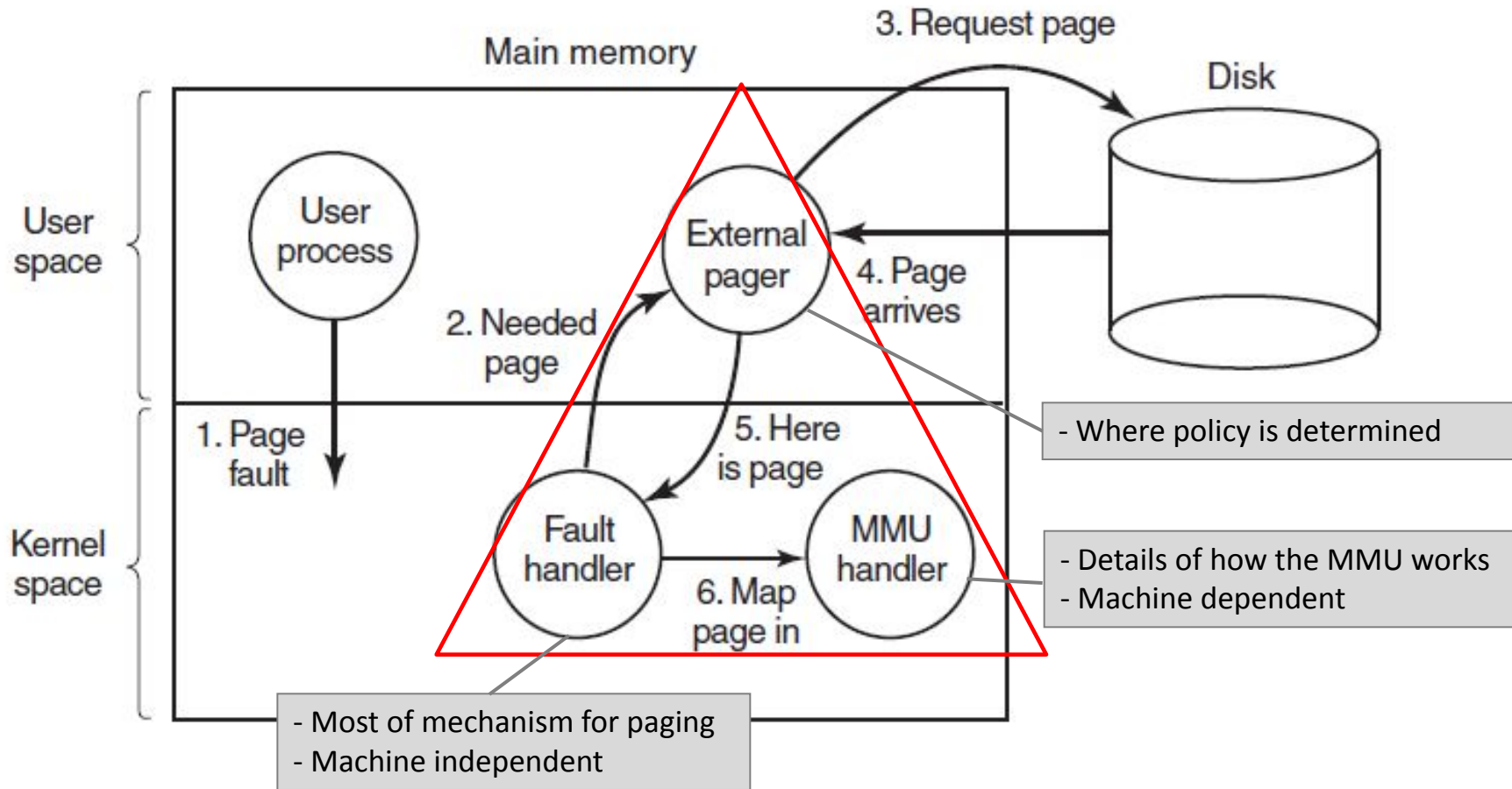


Figure 3-29. Page fault handling with an external pager.

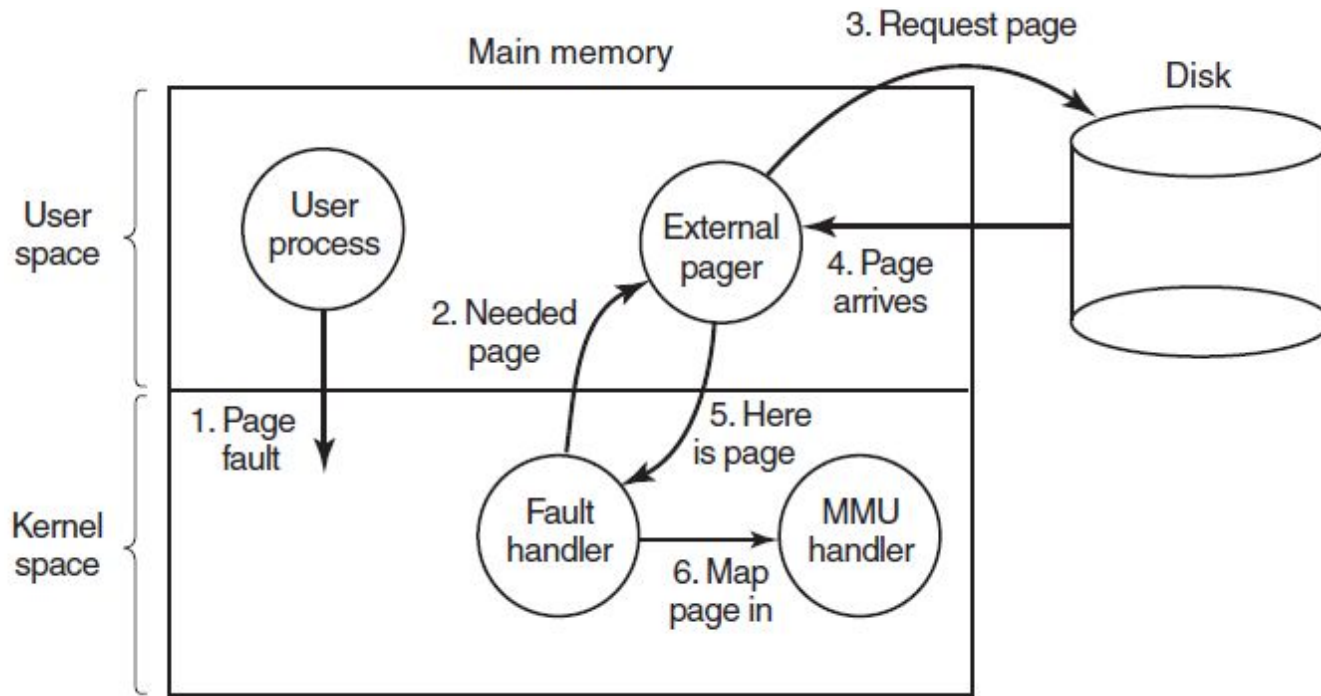
## Separation of Policy and Mechanism (3)

- All the details of how the MMU works are encapsulated in the MMU handler, which is machine-dependent code and has to be rewritten for each new platform the OS is ported to
- The page-fault handler is machine-independent code and contains most of the mechanism for paging
- The policy is largely determined by the external pager, which runs as a user process

## Separation of Policy and Mechanism (4)

- When a process starts up, the external pager is notified in order to set up the process' page map and allocate the necessary backing store on the disk
- As the process runs, it may map new objects into its address space, so the external pager is once again notified
- Then, the following events occur (Fig. 3-29)

# Separation of Policy and Mechanism (5)

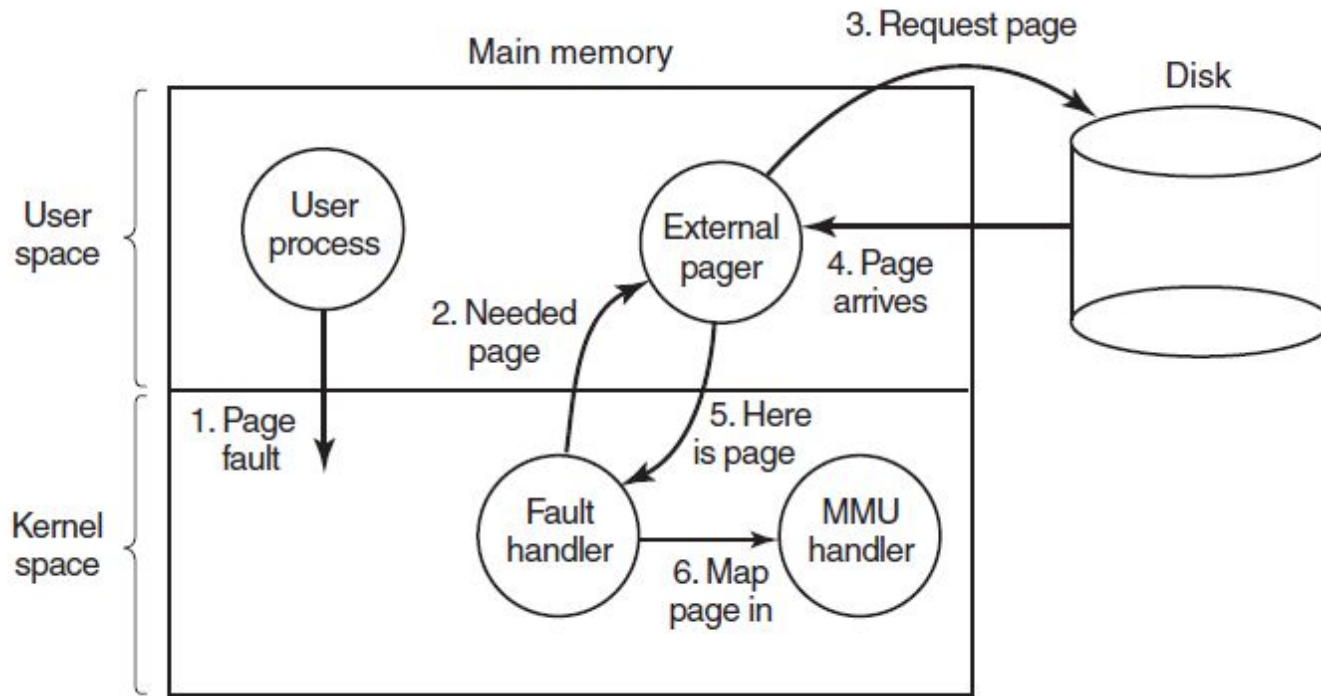


1. The running process gets a page fault

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

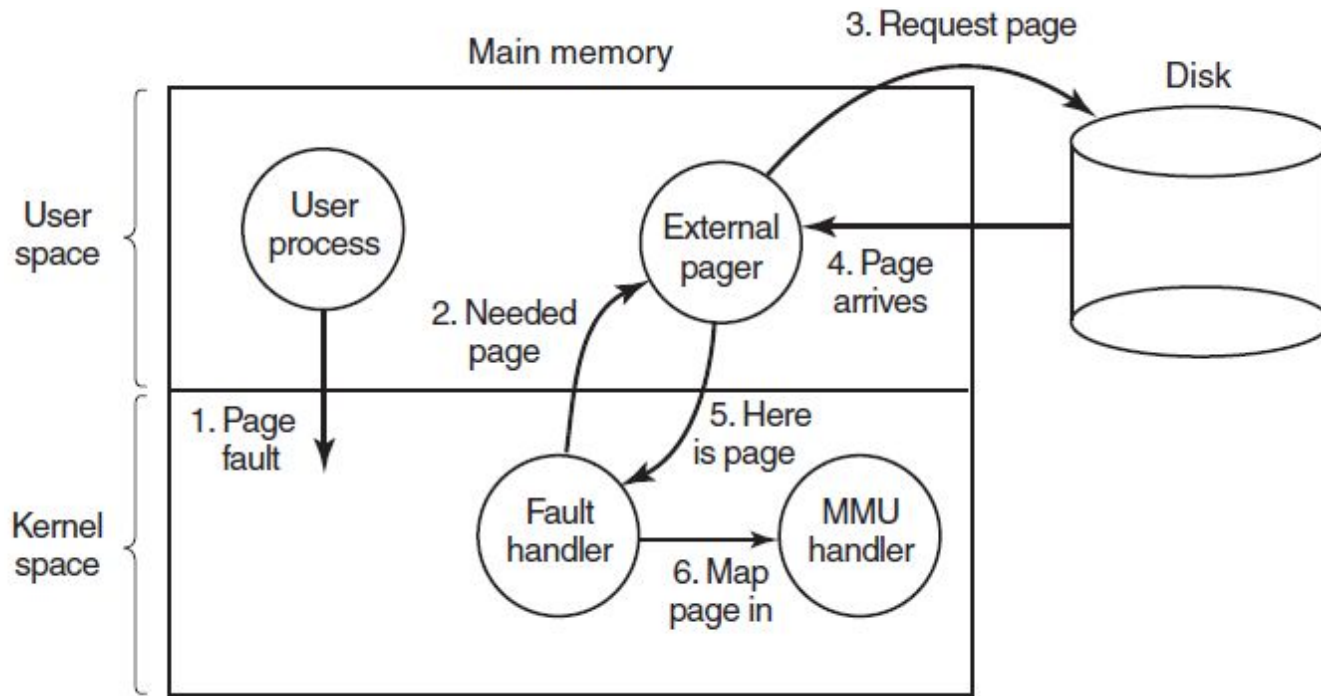


# Separation of Policy and Mechanism (6)



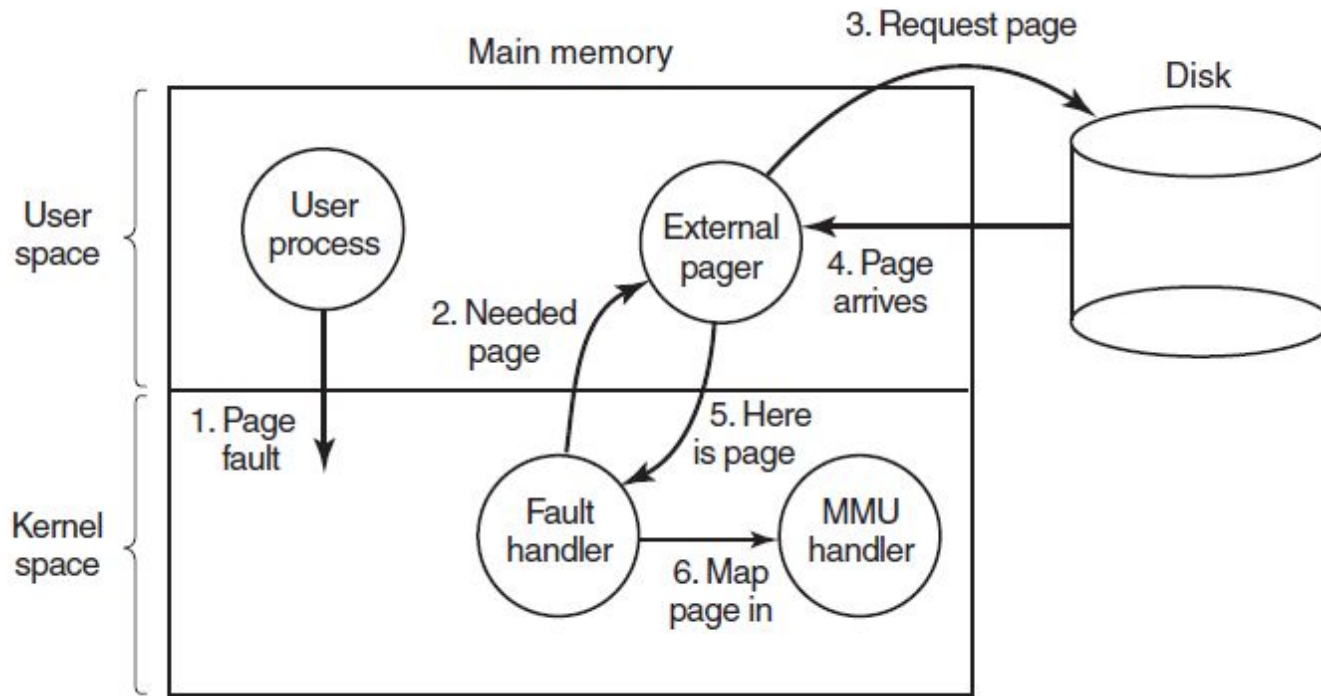
2. The fault handler figures out which virtual page is needed and sends a message to the external pager

# Separation of Policy and Mechanism (7)



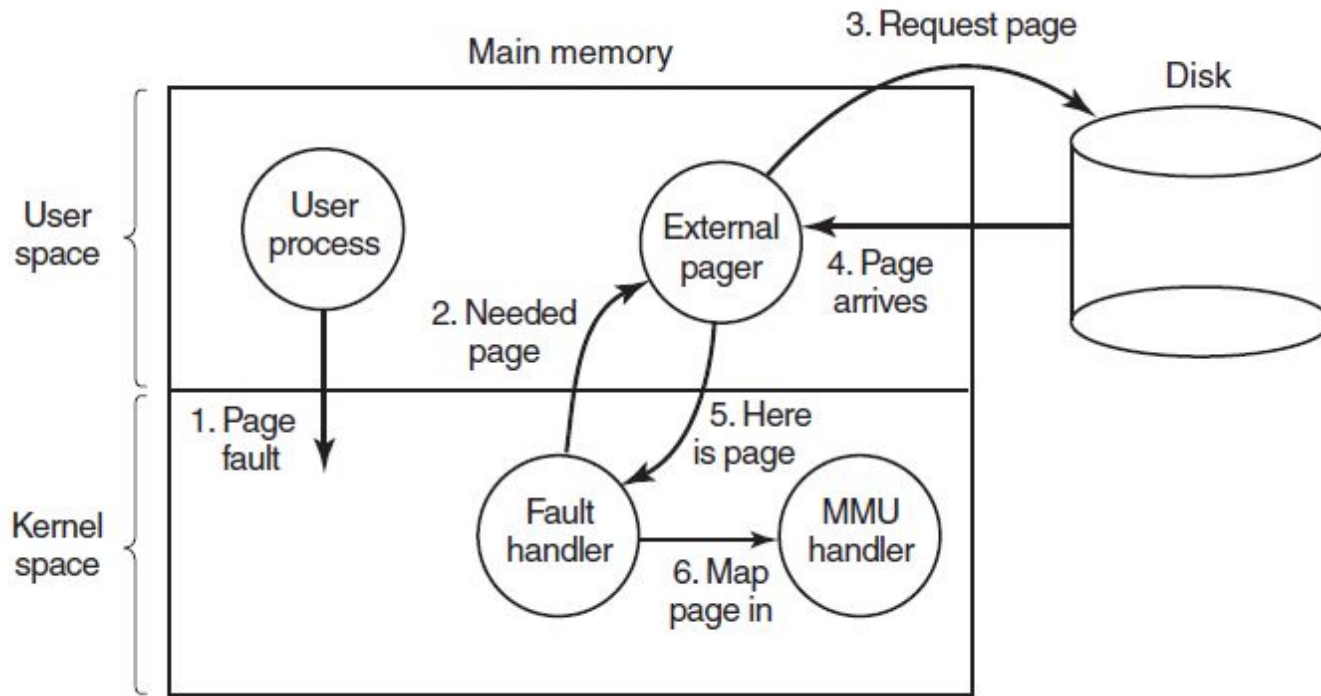
3. The external pager reads the page in from the disk and...
4. ... copies it to a portion of its own address space

# Separation of Policy and Mechanism (8)



5. The external pager informs the fault handler where the page is

# Separation of Policy and Mechanism (9)



6. The fault handler unmaps the page from the external pager's address space and asks the MMU handler to put it into the user's address space at the right place

(The user process can now be restarted)

# Separation of Policy and Mechanism (10)

- The page replacement algorithm can be put in the external pager, but there are some issues:
  - the external pager does not have access to  $R$  and  $M$  bits of all the pages
  - either some mechanism is needed to pass this information up to the external pager, or the page replacement algorithm must go in the kernel

# Separation of Policy and Mechanism (11)

- The main **advantage** of this implementation is more modular code and greater flexibility
- The main **disadvantage** is the extra overhead of crossing the user-kernel boundary several times and the overhead of the various messages being sent between the pieces of the system

# Segmentation (1)

- Examples of several tables generated by compiler:
  - The source text being saved for the printed listing
  - The symbol table, names and attributes of variables
  - The table containing integer and floating-point constants used
  - The parse tree, syntactic analysis of the program
  - The stack used for procedure calls within compiler

# Segmentation (2)

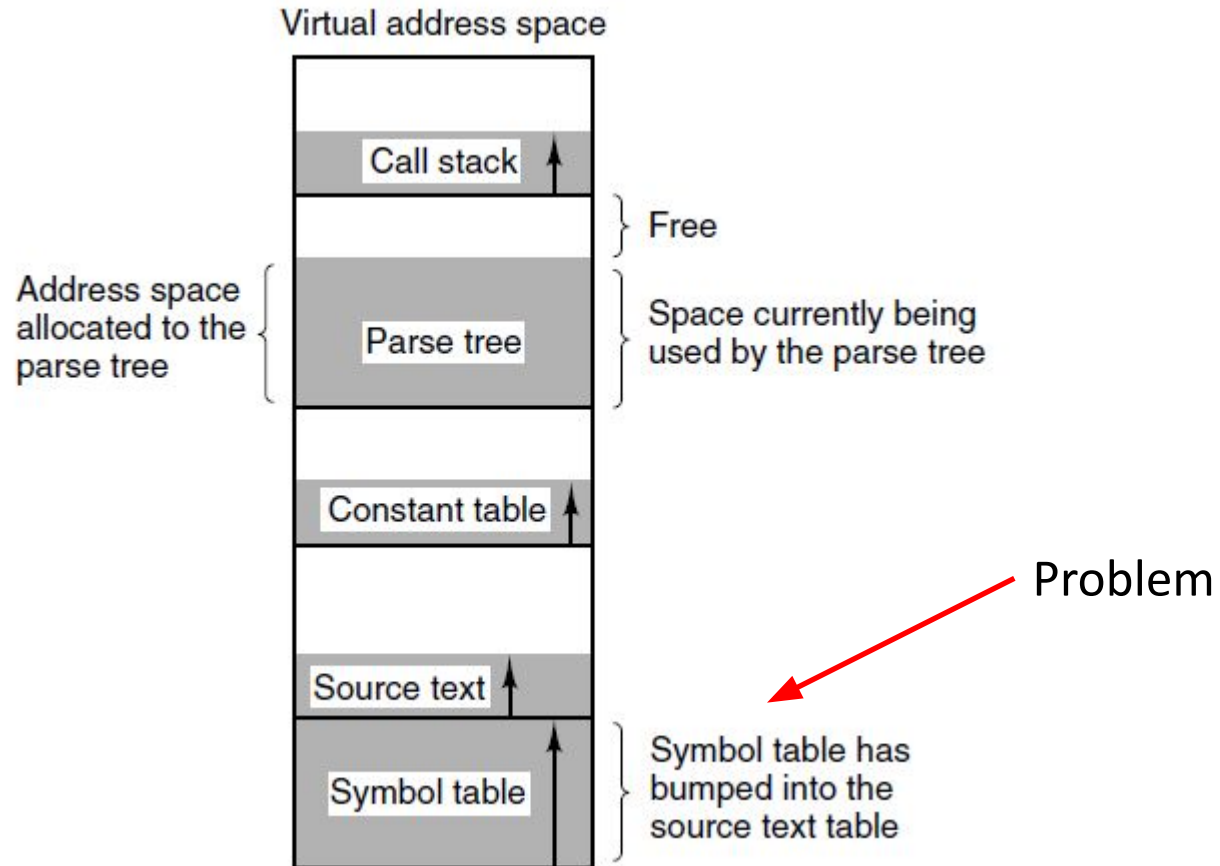


Figure 3-30. In a one-dimensional address space with growing tables, one table may bump into another.



# Segmentation (3)

- What is needed is a way of freeing the programmer from having to manage the expanding and contracting tables
- The solution is to provide the machine with many completely independent address spaces, which are called **segments**
- Each segment consists of a linear sequence of addresses, starting at 0 and going up to some maximum value. The length of each segment may be anything from 0 to the maximum address allowed
- Segment lengths may change during execution

# Segmentation (4)

- Because each segment constitutes a separate address space, different segments can grow or shrink independently without affecting each other
- A segment can fill up, but segments are usually very large, so this occurrence is rare
- To specify an address in this segmented or two-dimensional memory, the program must supply a two-part address, a segment number, and an address within the segment (Fig. 3-31)

# Segmentation (5)

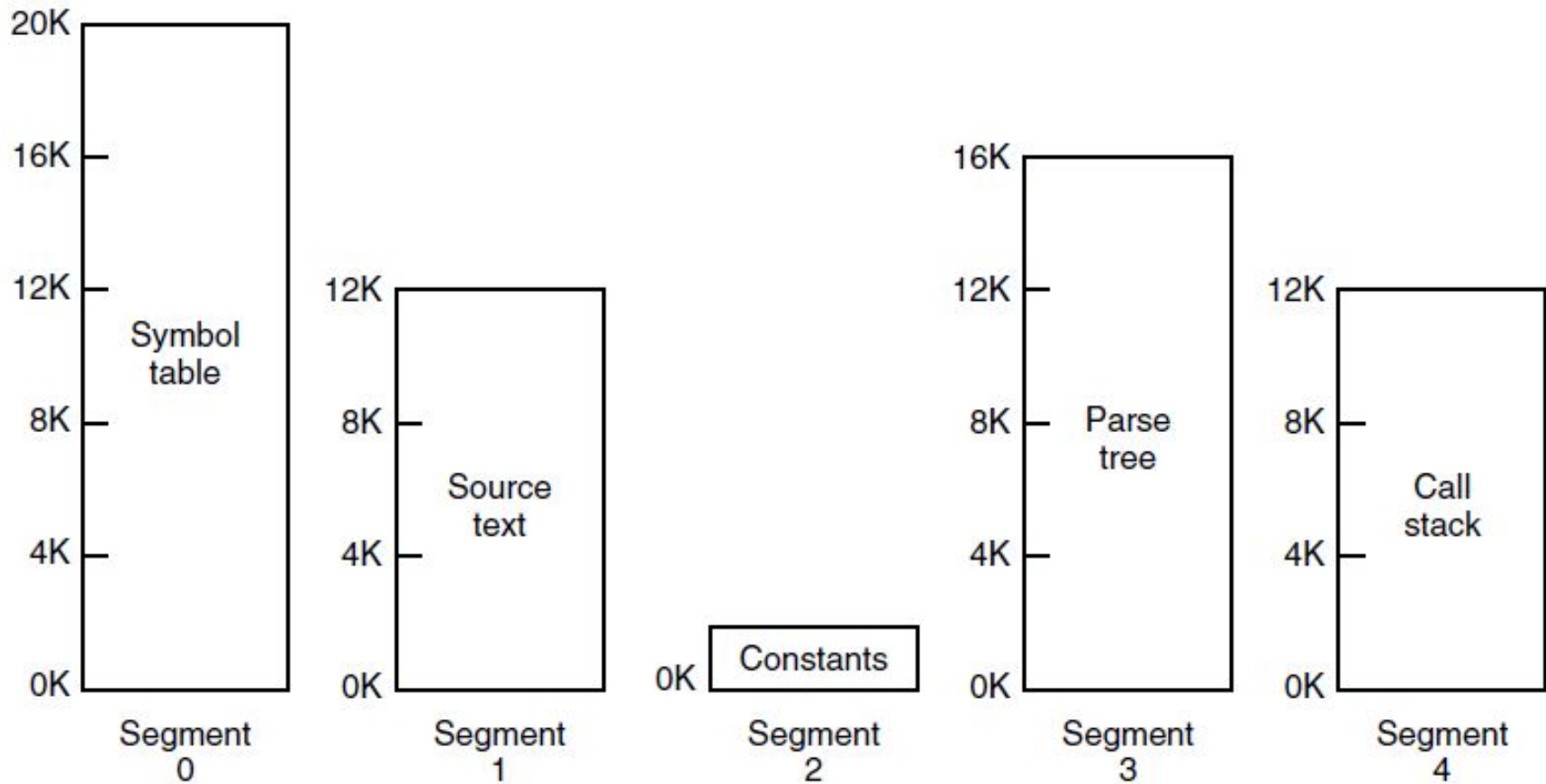


Figure 3-31. A segmented memory allows each table to grow or shrink independently of the other tables.

# Segmentation (6)

- A segment is a logical entity that might contain a procedure, or an array, or a stack, or a collection of scalar variables, but usually it does not contain a mixture of different types
- Some advantages of segments are:
  - If each procedure occupies a separate segment, with address 0 as its starting address, the linking of procedures compiled separately is greatly simplified. After all the procedures that constitute a program have been compiled and linked up, a procedure call to the procedure in segment  $n$  will use the two-part address  $(n, 0)$  to address word 0 (the entry point)

# Segmentation (7)

- Advantages of segments (cont.):
  - If the procedure in segment  $n$  is subsequently modified and recompiled, no starting addresses are modified.  
Consequently, no other procedures need be changed, even if the new version is larger than the old one
  - Segmentation also facilitates sharing procedures or data between several processes (the shared libraries)
  - Different segments can have different kinds of protection

# Segmentation (8)

Consideration	Paging	Segmentation
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

Figure 3-32. Comparison of paging and segmentation

# Segmentation (9)

Consideration	Paging	Segmentation
Does the programmer need to be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	One	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

Figure 3-32. Comparison of paging and segmentation

# Implementation of Pure Segmentation (1)

- Consider a piece of memory containing five segments (Fig. 3-33a)
- If a relatively large segment is evicted and another segment, which is smaller, is put in its place there will be a hole between two segments (Fig. 3-33b)
- 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
- This is called **checkerboarding** or **external fragmentation**. It wastes memory in the holes and can be dealt with by compaction (Fig. 3-33e)



# Implementation of Pure Segmentation (2)

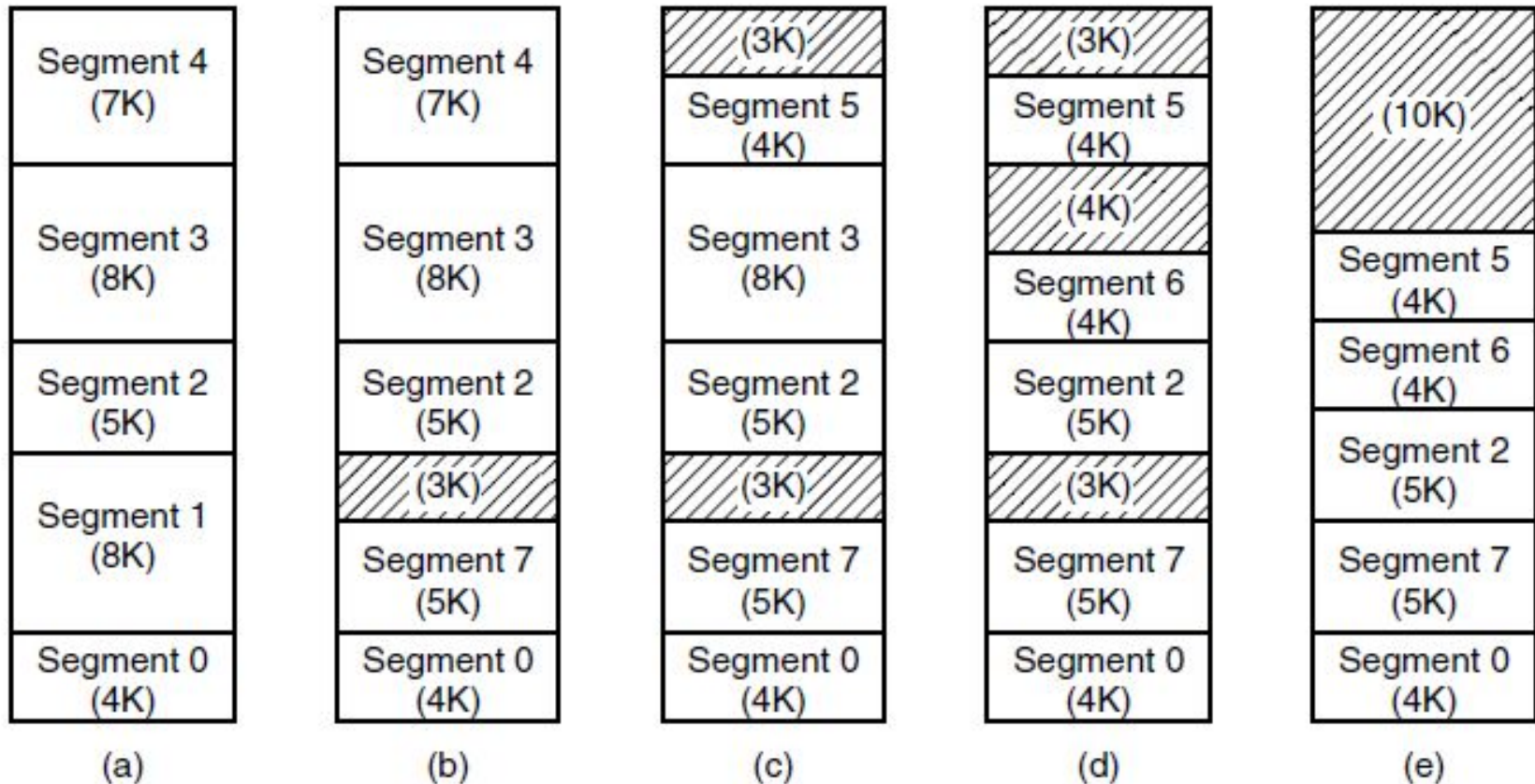


Figure 3-33. (a)-(d) Development of checkerboarding.  
(e) Removal of the checkerboarding by compaction.

# Segmentation with Paging

- If the segments are large, it may be inconvenient or impossible, to keep them in main memory as a whole
- This leads to the idea of paging them, so that only those pages of a segment that are actually needed have to be around
- We will cover two examples: MULTICS and Intel x86

# Segmentation with Paging: MULTICS (1)

- MULTICS ran on the Honeywell 6000 machines and their descendants and provided each program with a virtual memory of up to  $2^{18}$  segments, each of which was up to 65,536 (36-bit) words long
- To implement this, the MULTICS designers chose to treat each segment as a virtual memory and to page it, combining the advantages of paging with the advantages of segmentation

# Segmentation with Paging: MULTICS (2)

- Each MULTICS program had a segment table, with one descriptor per segment
- The segment table with potentially more than a quarter of a million entries was itself a segment and was paged
- A segment descriptor contained an indication of whether the segment was in main memory or not. If any part of the segment was in memory, the segment was considered to be in memory, and its page table was in memory
- If the segment was in memory, its descriptor contained an 18-bit pointer to its page table (Fig. 3-34a)

# Segmentation with Paging: MULTICS (3)

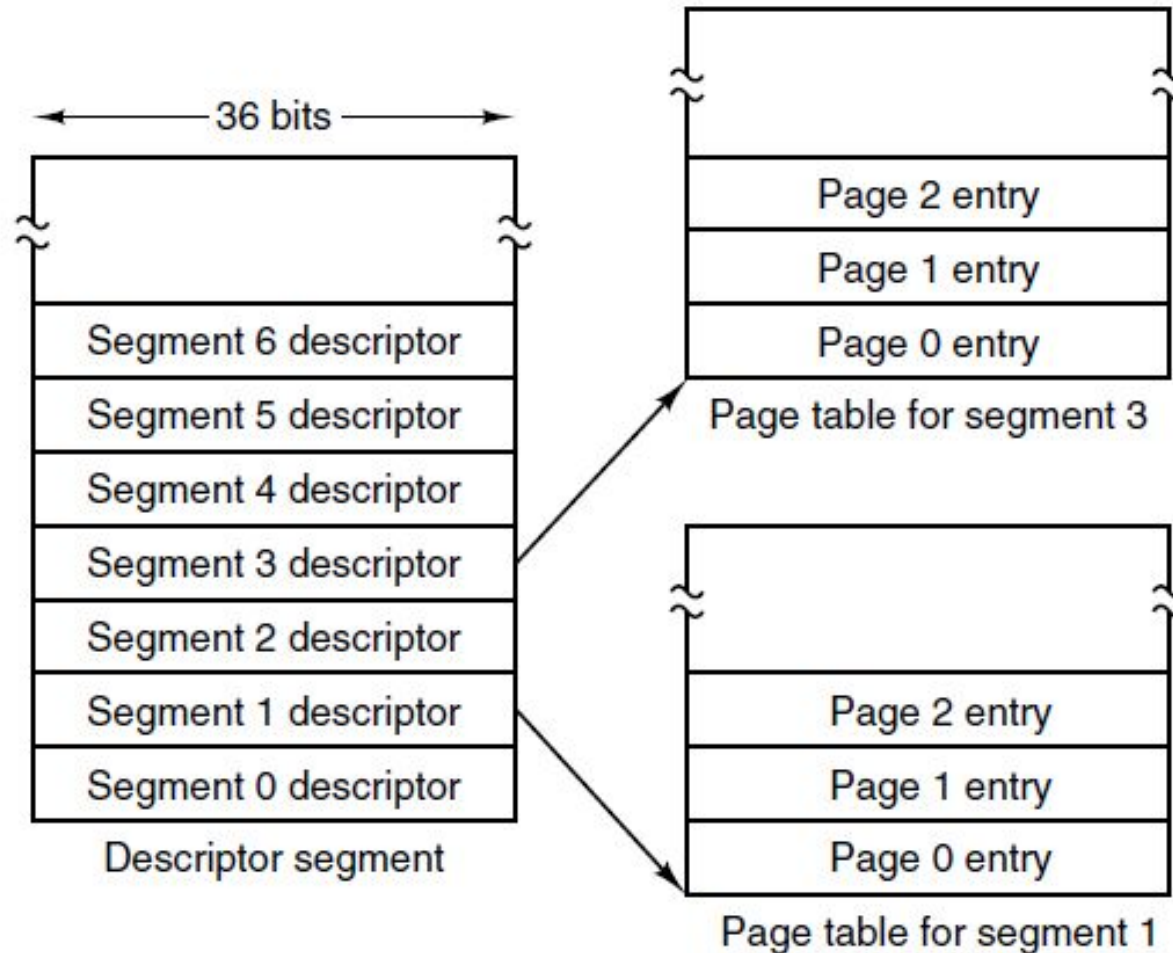


Figure 3-34a. The descriptor segment pointed to the page tables

# Segmentation with Paging: MULTICS (4)

- Physical addresses were 24 bits and pages were aligned on 64-byte boundaries (the low-order 6 bits of page addresses were 000000), only 18 bits were needed in the descriptor to store a page table address
- The descriptor also contained the segment size, the protection bits, and other items (Fig. 3-34b)
- The address of the segment in secondary memory was not in the segment descriptor but in another table used by the segment fault handler

# Segmentation with Paging: MULTICS (5)

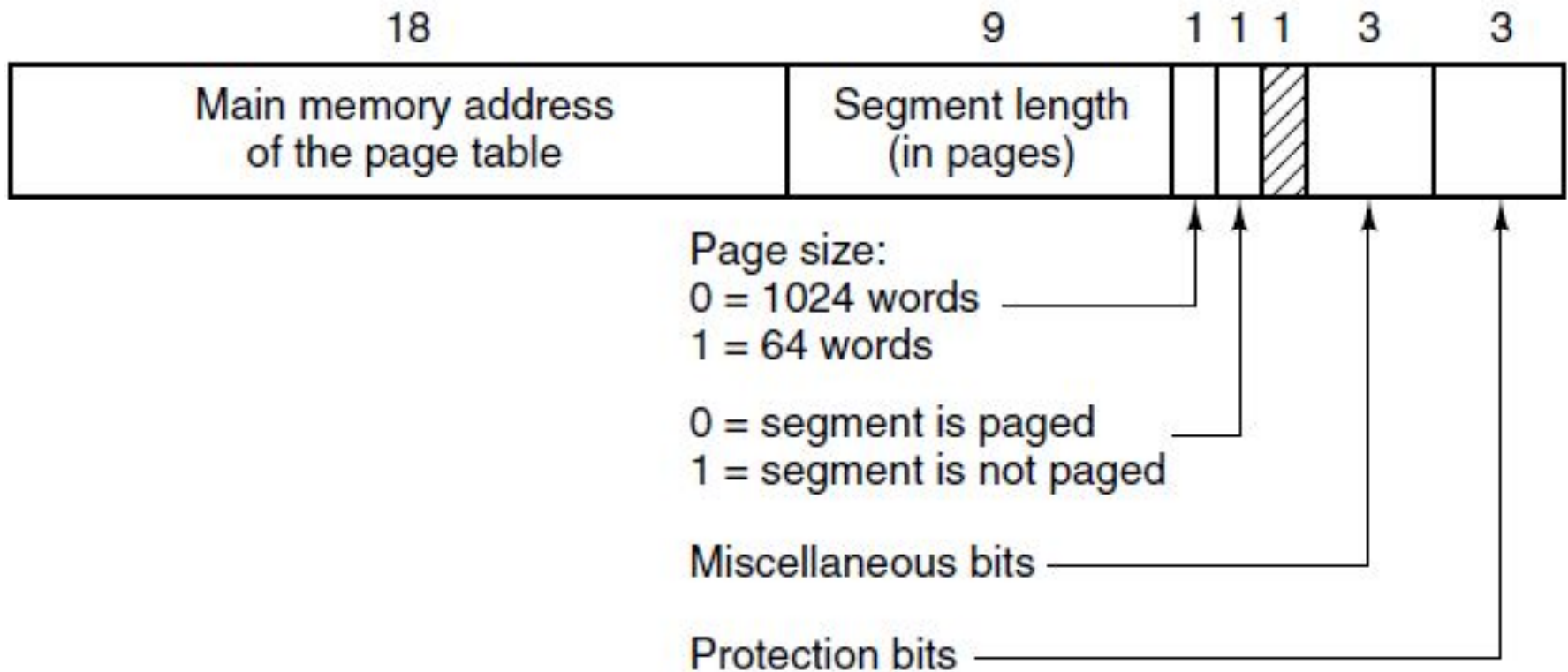


Figure 3-34. A segment descriptor.

The numbers are the field lengths

# Segmentation with Paging: MULTICS (6)

- Each segment was an ordinary virtual address space and was paged in the same way as the non-segmented paged memory. The normal page size was 1024 words
- An address in MULTICS consisted of two parts: the segment and the address within the segment which was divided into a page number and a word within the page (Fig. 3-35)



# Segmentation with Paging: MULTICS (7)

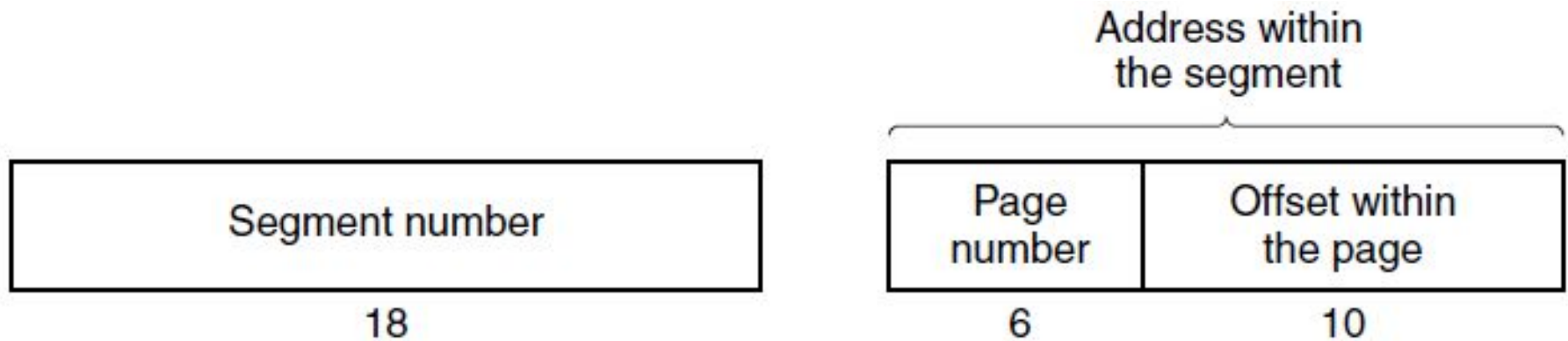


Figure 3-35. A 34-bit MULTICS virtual address.

# Segmentation with Paging: MULTICS (8)

- When a memory reference occurred, the following algorithm was carried out (Fig. 3-36):
  - The segment number was used to find the segment descriptor
  - A check was made to see if the segment's page table was in memory. If it was not, a segment fault occurred. If there was a protection violation, a fault (trap) occurred

# Segmentation with Paging: MULTICS (9)

- Memory reference with segments (cont.):
  - The page table entry for the requested virtual page was examined. If the page itself was not in memory, a page fault was triggered. If it was in memory, the main-memory address of the start of the page was extracted from the page table entry
  - The offset was added to the page origin to give the main memory address where the word was located
  - The read or store finally took place

# Segmentation with Paging: MULTICS (10)

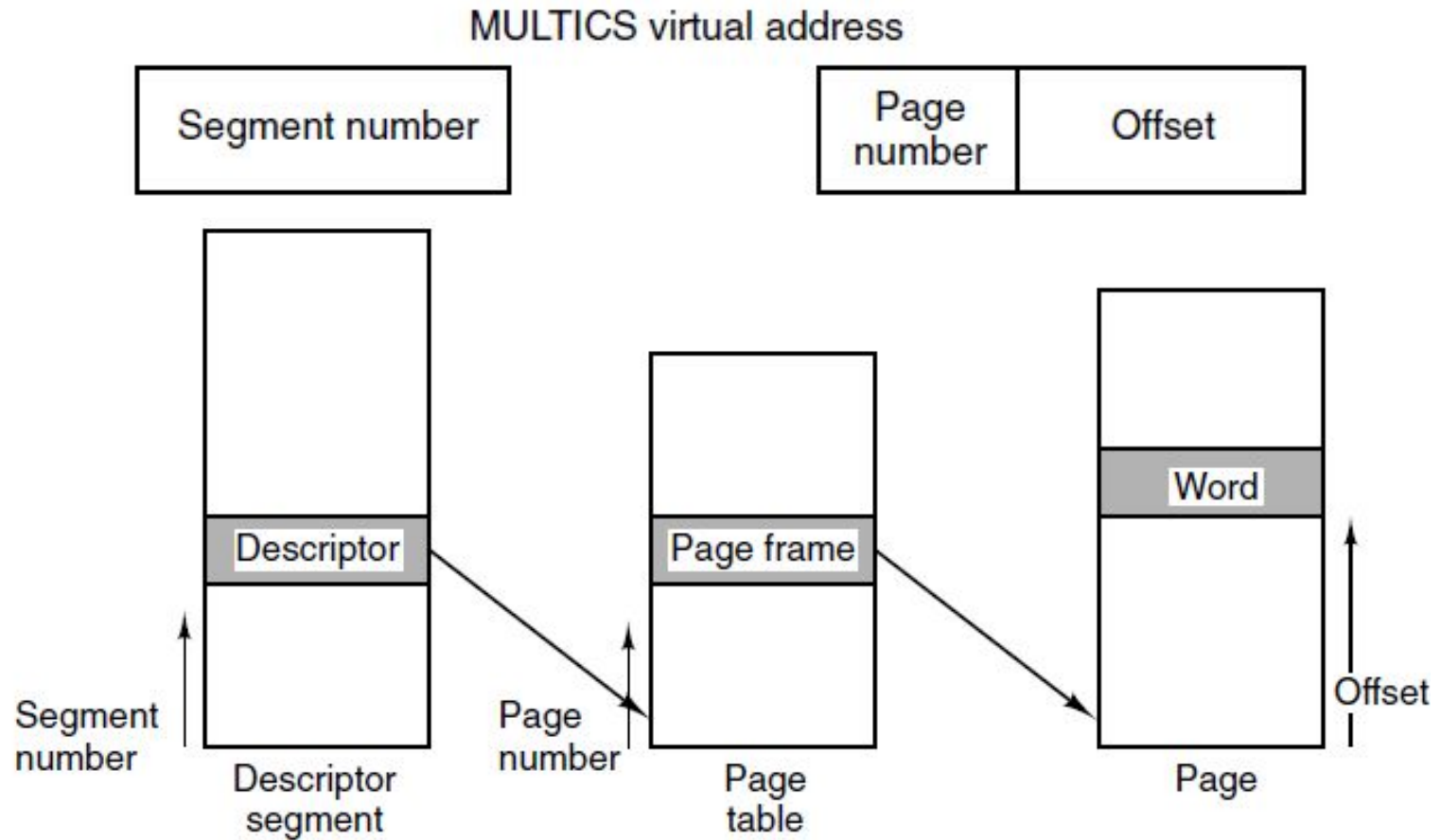


Figure 3-36. Conversion of a two-part MULTICS address into a main memory address.

# Segmentation with Paging: MULTICS (11)

- The MULTICS hardware contained a 16-word high-speed TLB that could search all its entries in parallel for a given key
- When an address was presented to the computer, the addressing hardware first checked to see if the virtual address was in the TLB
- If so, it got the page frame number directly from the TLB and formed the actual address of the referenced word without having to look in the descriptor segment or page table (Fig. 3-37)

# Segmentation with Paging: MULTICS (12)

Comparison field		Page frame	Protection	Age	Is this entry used? ↓
Segment number	Virtual page				
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Figure 3-37. A simplified version of the MULTICS TLB. The existence of two page sizes made the actual TLB more complicated.

# Segmentation with Paging: The Intel x86 (1)

- In x86-64 CPUs, segmentation is considered obsolete and is no longer supported, except in legacy mode
- We will discuss x86-32. It has 16K segments, each holding up to 1 billion 32-bit words
- The larger segment size is important since few programs need more than 1000 segments, but many programs need large segments

# Segmentation with Paging: The Intel x86 (2)

- x86 virtual memory model contains two tables:
  - **Local Descriptor Table (LDT)** describes segments local to each program, including its code, data, stack, and so on. Each program has its own LDT
  - **Global Descriptor Table (GDT)** describes system segments, including the OS itself. It is shared by all the programs on the computer



# Segmentation with Paging: The Intel x86 (3)

- To access a segment, an x86 program first loads a selector for that segment into one of the machine's six segment registers
- During execution, the **CS register** holds the selector for the code segment and the **DS register** holds the selector for the data segment
- Each selector is a 16-bit number (Fig. 3-38)
- Descriptor 0 is forbidden and causes a trap if used. It may be safely loaded into a segment register to indicate that the segment register is not currently available

# Segmentation with Paging: The Intel x86 (4)

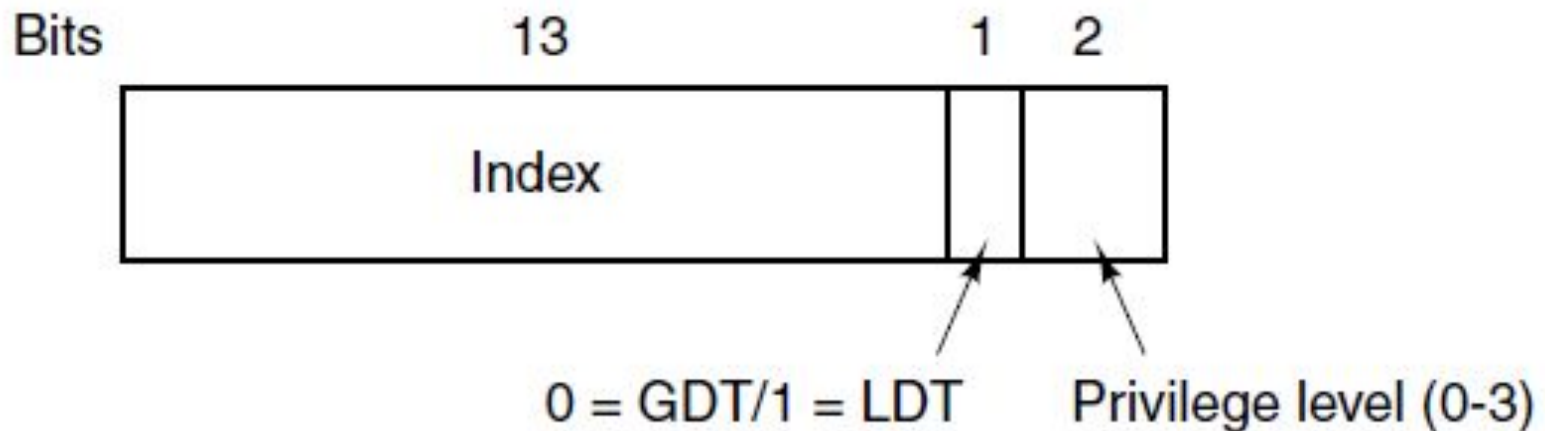


Figure 3-38. An x86 selector.

# Segmentation with Paging: The Intel x86 (5)

- At the time a selector is loaded into a segment register, the corresponding descriptor is fetched from the LDT or GDT and stored in microprogram registers
- The format of the selector allows to locate the descriptor easily:
  - Either the LDT or GDT is selected, based on selector bit 2
  - The selector is copied to an internal scratch register, and the 3 low-order bits set to 0
  - The address of either the LDT or GDT table is added to it, to give a direct pointer to the descriptor

# Segmentation with Paging: The Intel x86 (6)

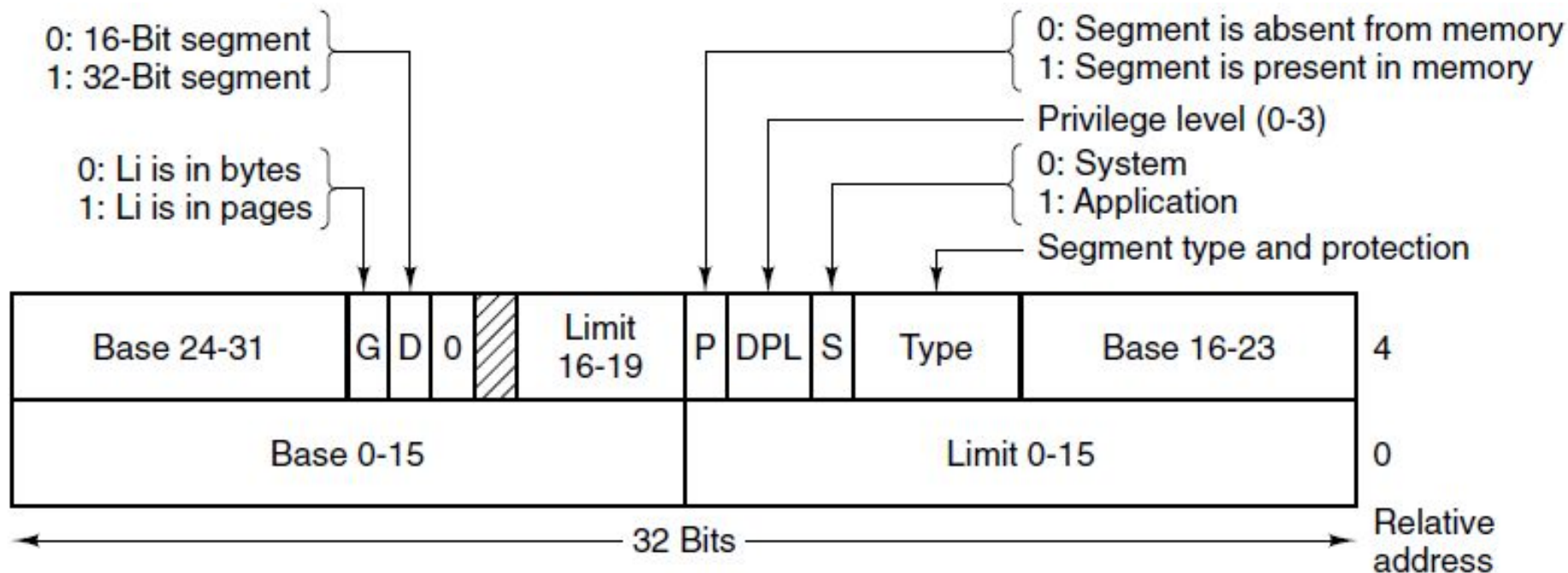


Figure 3-39. x86 code segment descriptor.  
Data segments differ slightly.

# Segmentation with Paging: The Intel x86 (7)

- Step-by-step conversion of a (selector, offset) pair to a physical address:
  - The microprogram can find the complete descriptor corresponding to the selector in its internal registers
  - If the segment does not exist (selector 0), or is currently paged out, a trap occurs
  - The hardware uses the *Limit* field to check if the offset is beyond the end of the segment, in which case a trap also occurs

# Segmentation with Paging: The Intel x86 (7)

- Conversion (cont.):
  - If the *Gbit* (Granularity) field is 0, the *Limit* field is the exact segment size, up to 1 MB. Otherwise, the size is in pages
  - The x86 then adds the 32-bit *Base* field in the descriptor to the offset to form what is called a **linear address** (Fig. 3-40)
  - If paging is disabled, the linear address is interpreted as the physical address

# Segmentation with Paging: The Intel x86 (8)

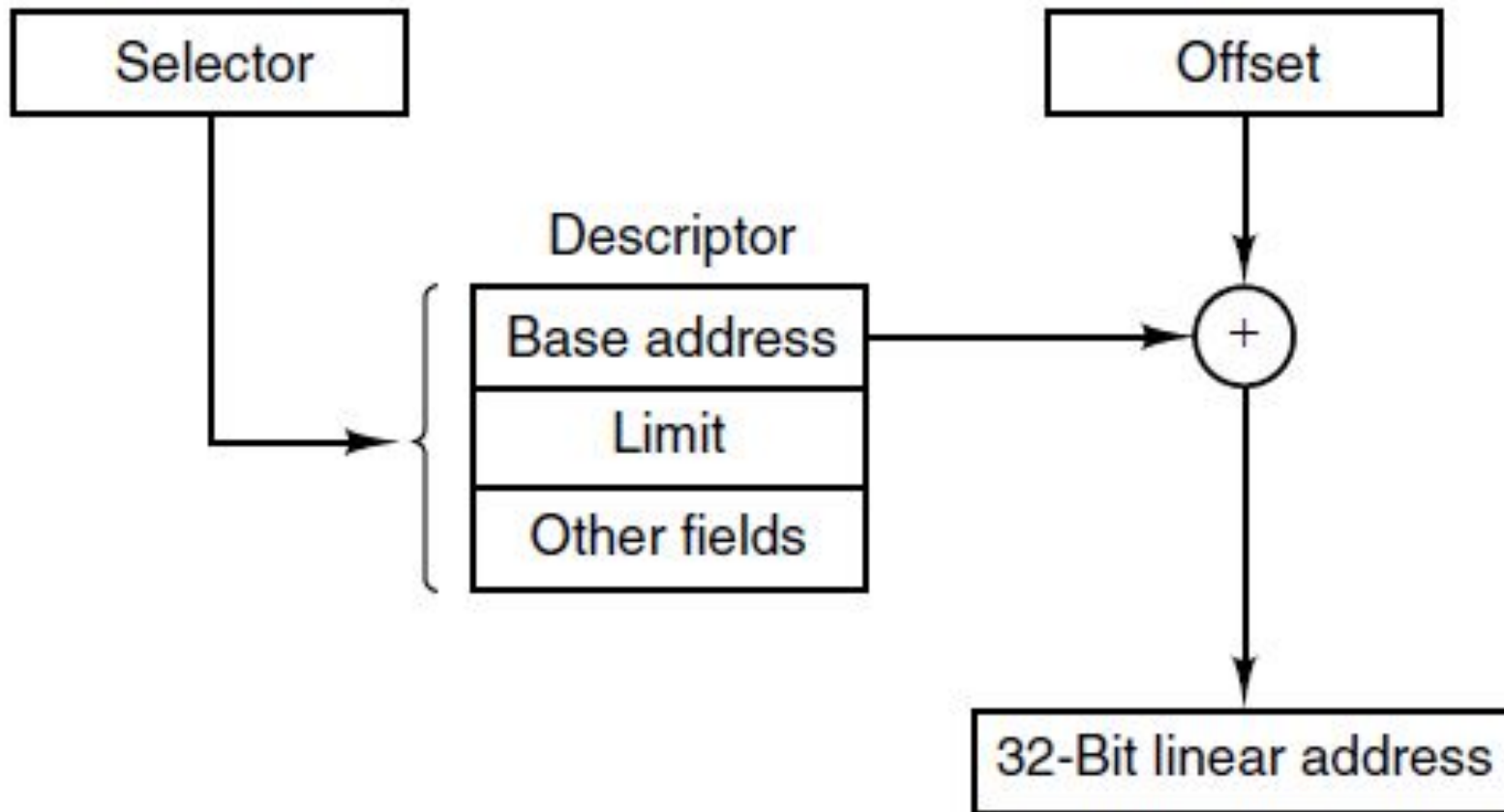


Figure 3-40. Conversion of a (selector, offset) pair to a linear address.

# Segmentation with Paging: The Intel x86 (9)

- Conversion (cont.):
  - If paging is enabled, the linear address is interpreted as a virtual address and mapped onto the physical address using page tables
  - Each running program has a page directory consisting of 1024 32-bit entries:
    - It is located at an address pointed to by a global register.
    - Each entry in this directory points to a page table also containing 1024 32-bit entries.
    - The page table entries point to page frames



# Segmentation with Paging: The Intel x86 (10)

- Conversion (cont.):
  - The *Dir* field of a linear address (Fig. 3-41a) is an index into the page directory to locate a pointer to the proper page table
  - The *Page* field is an index into the page table to find the physical address of the page frame
  - *Offset* is added to the address of the page frame to get the physical address of the byte or word needed
  - To avoid making repeated references to memory, the x86 has a small TLB that directly maps the most recently used *Dir-Page* combinations onto the physical address of the page frame

# Segmentation with Paging: The Intel x86 (11)

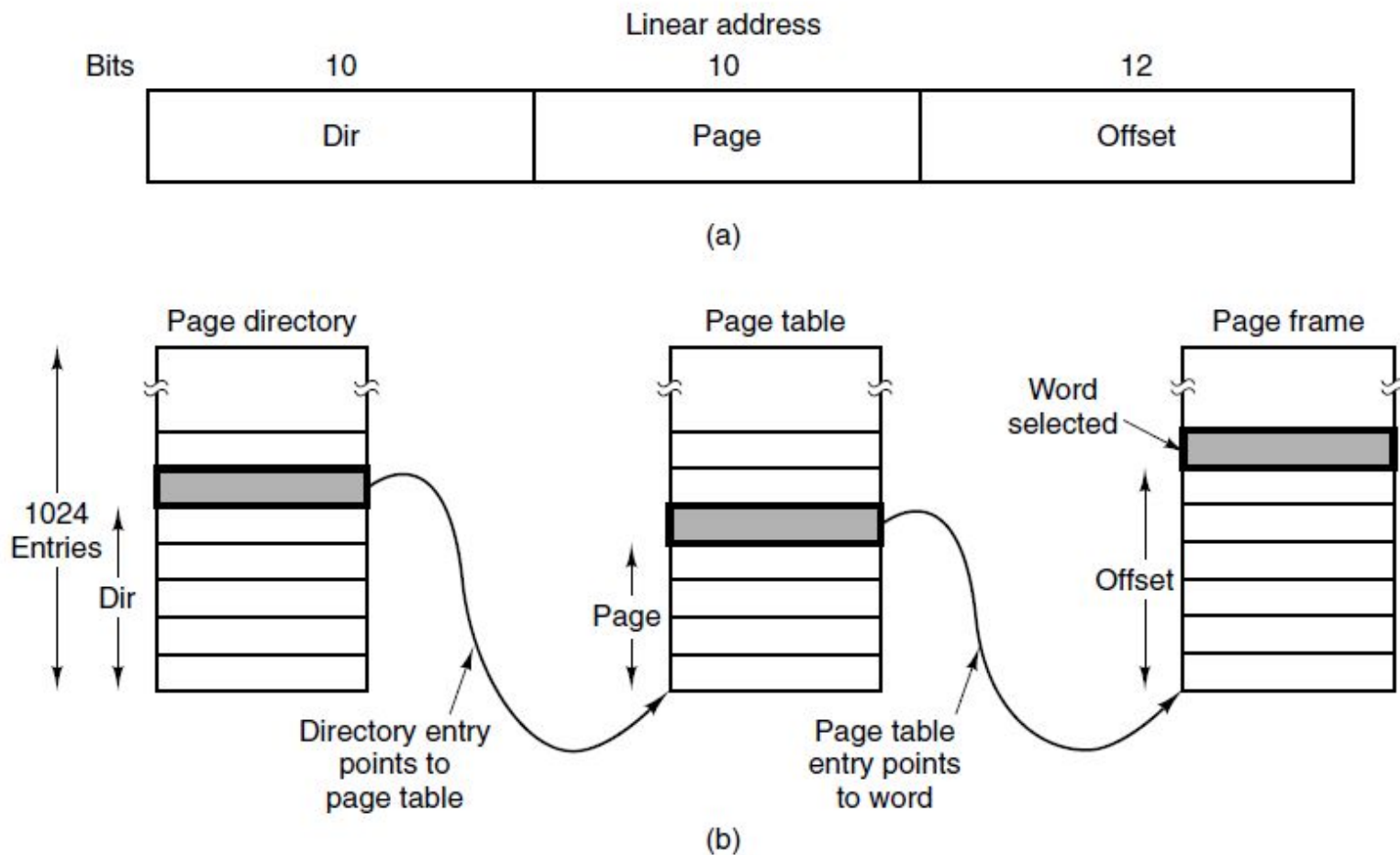


Figure 3-41. Linear to physical address mapping.

End

Week 09 – Lecture 1

# References

- Tanenbaum & Bo, Modern Operating Systems:  
4th edition, 2013  
Prentice-Hall, Inc.