Memory Management

Chapter 3

Memory

- Paraphrase of Parkinson's Law, ‘‘*Programs expand to fill the memory available to hold them*.’’
- Average home computer nowadays has 10,000 times more memory than the IBM 7094, the largest computer in the world in the early 1960s

Cache Memory (Computer Arch.)

- Principle of Locality
- Memory Hierarchy Levels
- Cache Memory
  - Direct Mapped Cache
  - Cache Misses, Write-Through and Write-Back
  - Associative Caches
  - Replacement Policy
  - Multilevel Caches

Virtual Memory (Computer Arch.)

- Virtual Memory
  - Virtual and Physical Address
  - Page Tables
  - Address Translation Using a Page Table
  - Page Fault Penalty
  - Fast Translation Using a TLB

No Memory Abstraction

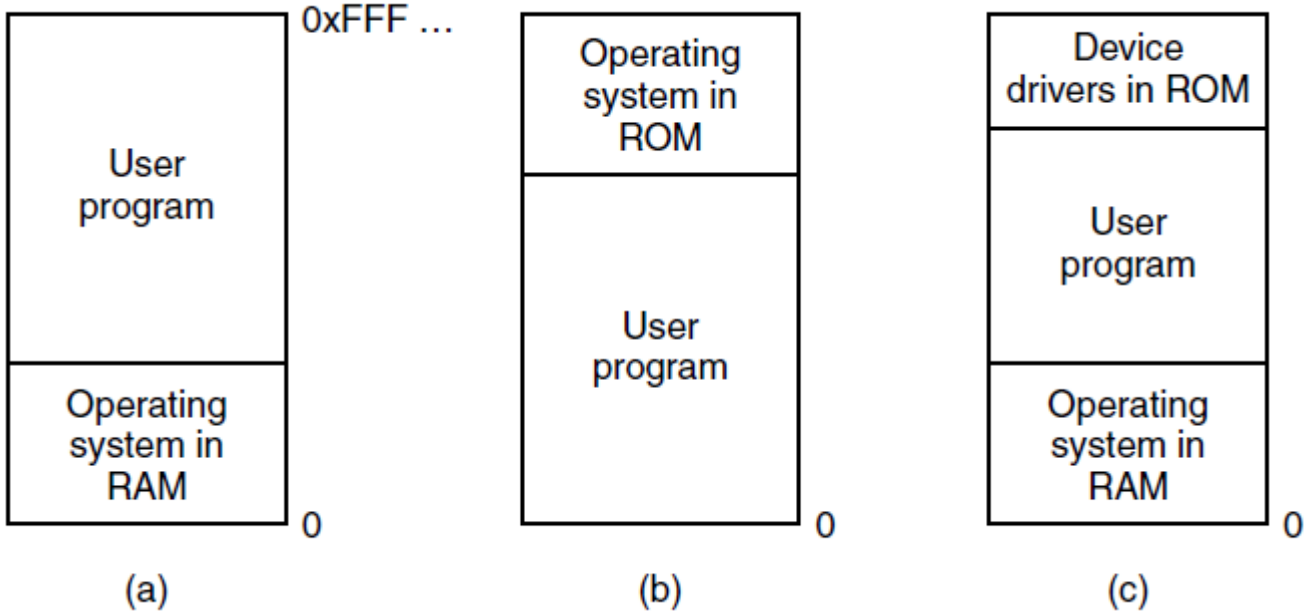


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist

Running Multiple Programs Without a Memory Abstraction

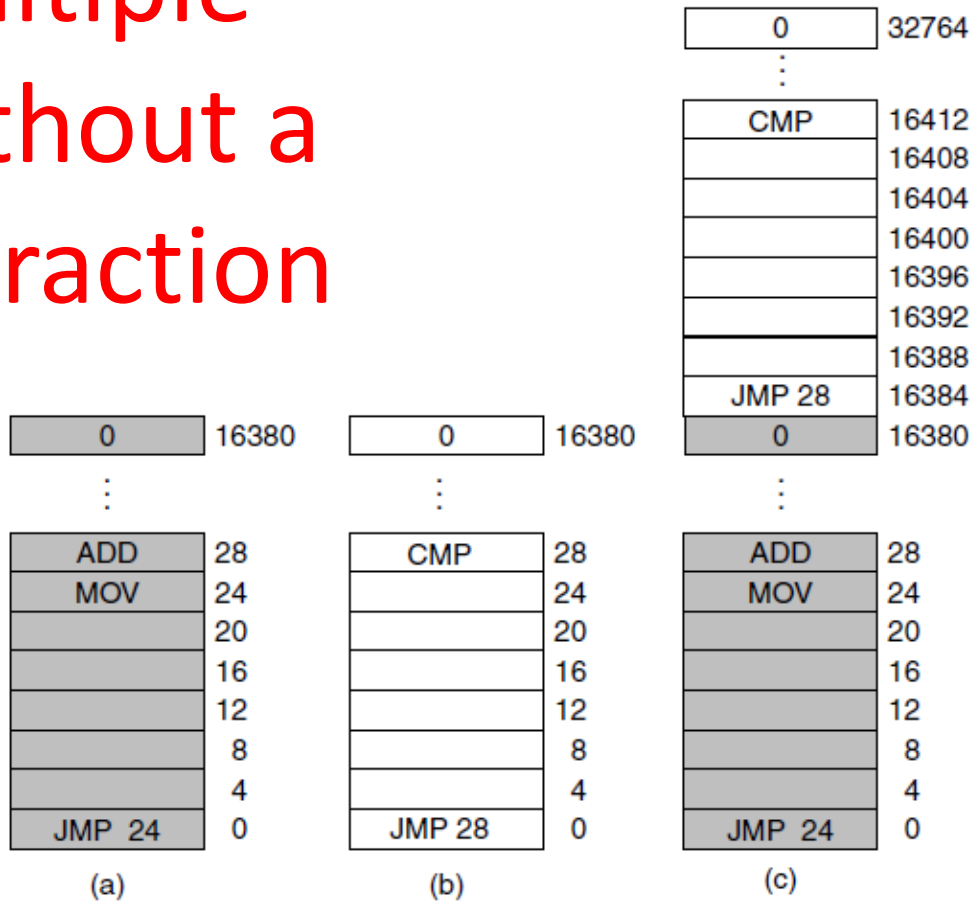


Figure 3-2. Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

Base and Limit Registers

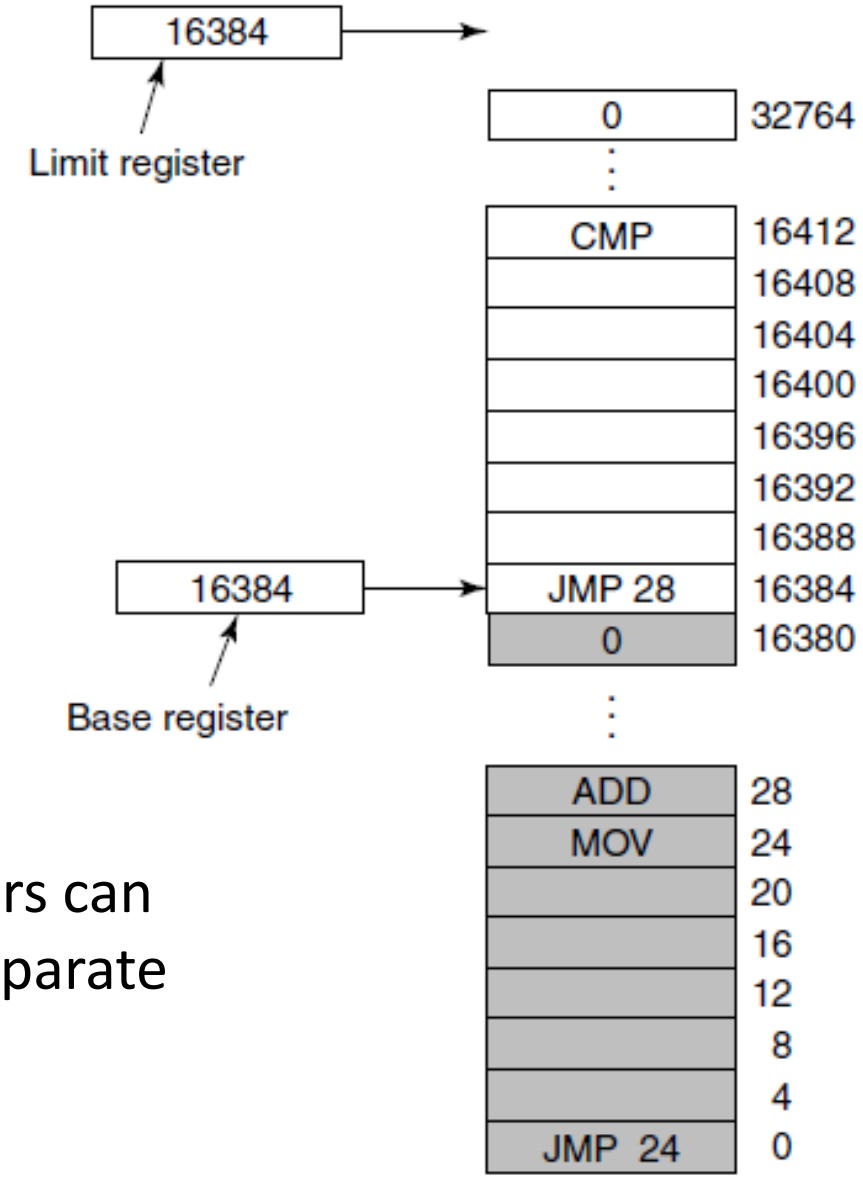


Figure 3-3. Base and limit registers can be used to give each process a separate address space.

Large Memory Footprint

- Previous schemes require the physical memory of the computer is large enough to hold all the processes
- Keeping all processes in memory all the time requires a huge amount of memory
- Solution
  - Swapping (process entirely in memory)
  - Virtual Memory (process partially in memory)

Swapping (1)

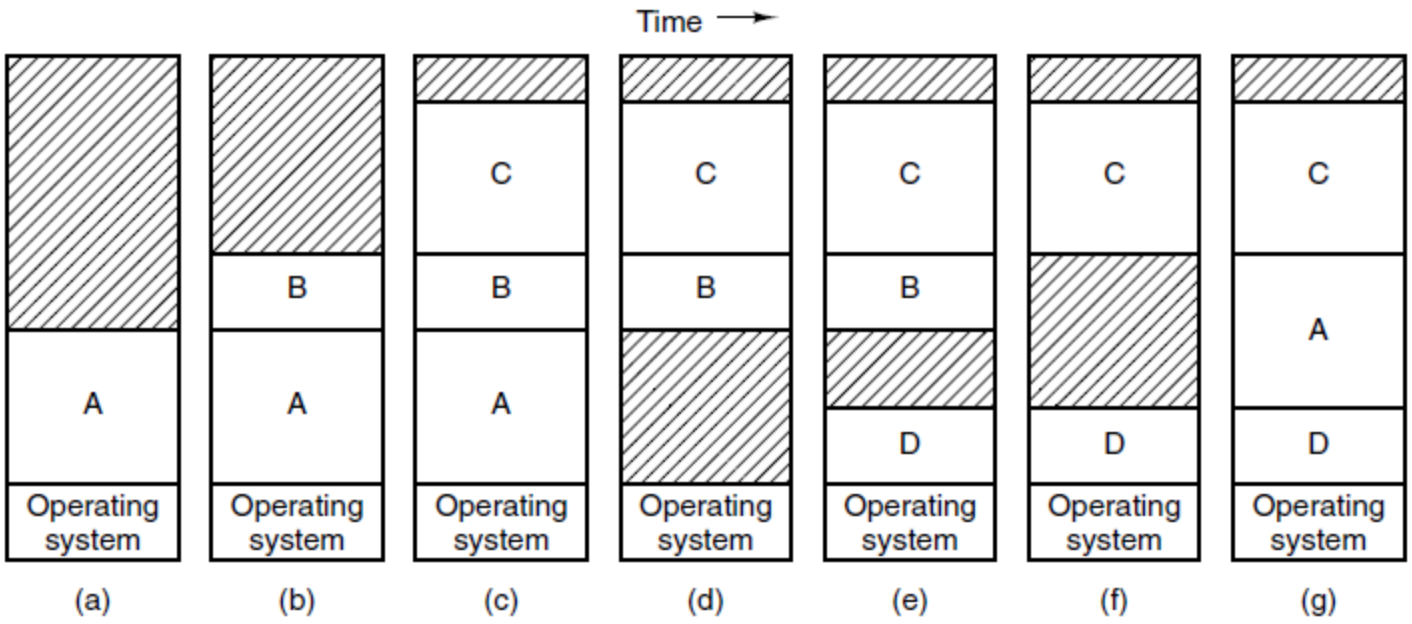


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory

Swapping (2)

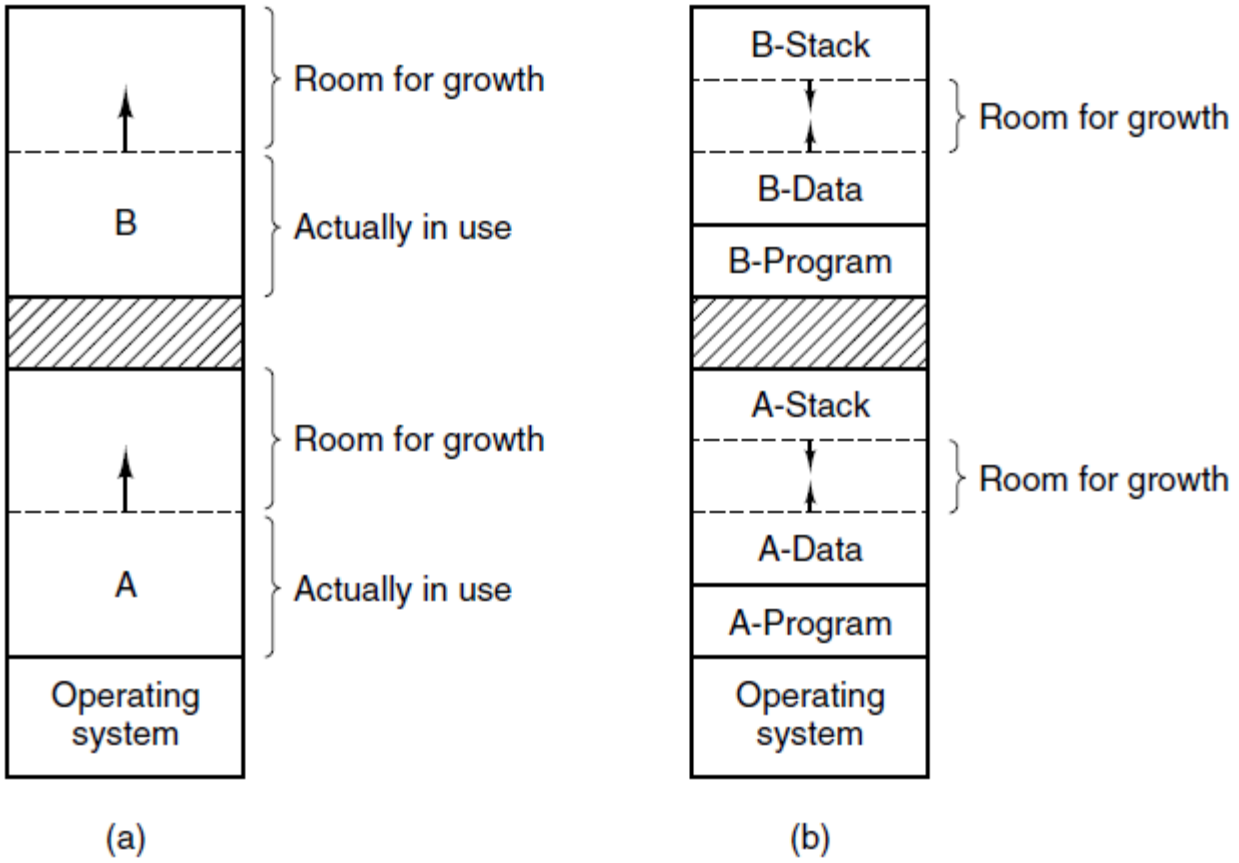


Figure 3-5. (a) Allocating space for a growing data segment.
(b) Allocating space for a growing stack and a growing data segment.

Swapping Issues

- Memory compaction might be needed when swapping creates multiple holes In memory
- If the growing process is adjacent to a hole
  - it can be allowed to grow into the hole
- If the growing process is adjacent to a process
  - it has to be moved to a hole large enough for it
  - or one or more processes have to be swapped out to create a large enough hole
  - If both fail, the growing process will be suspended

Memory Management with Bitmaps

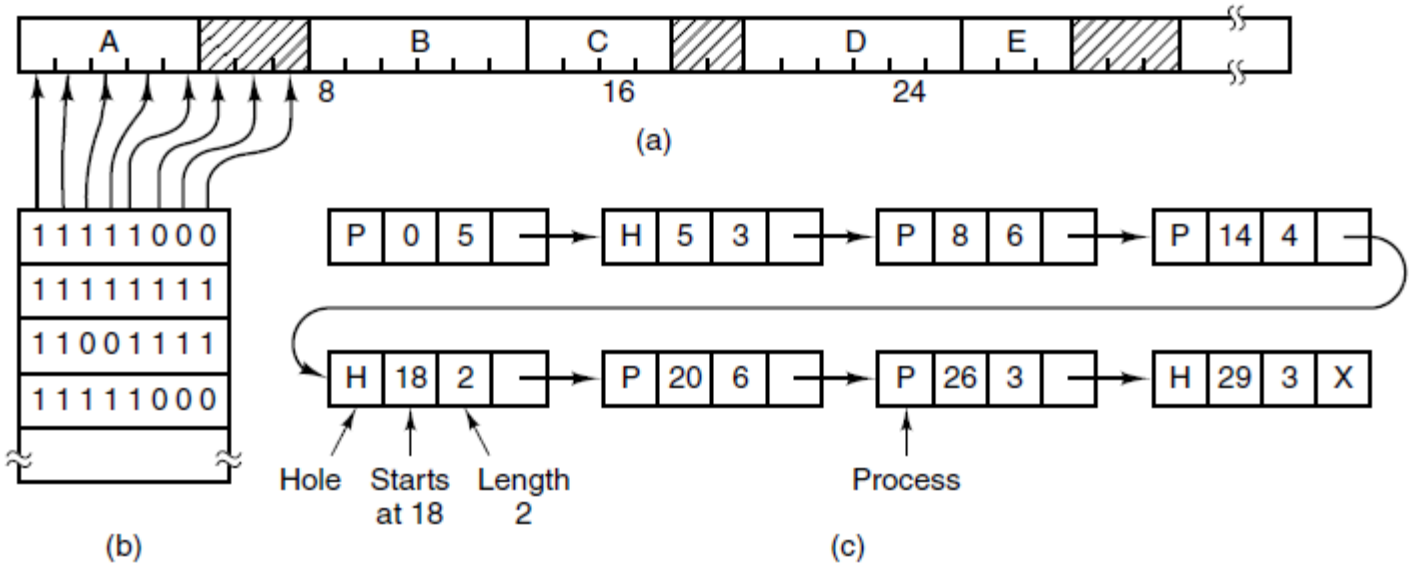


Figure 3-6. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

Memory Management with Linked Lists

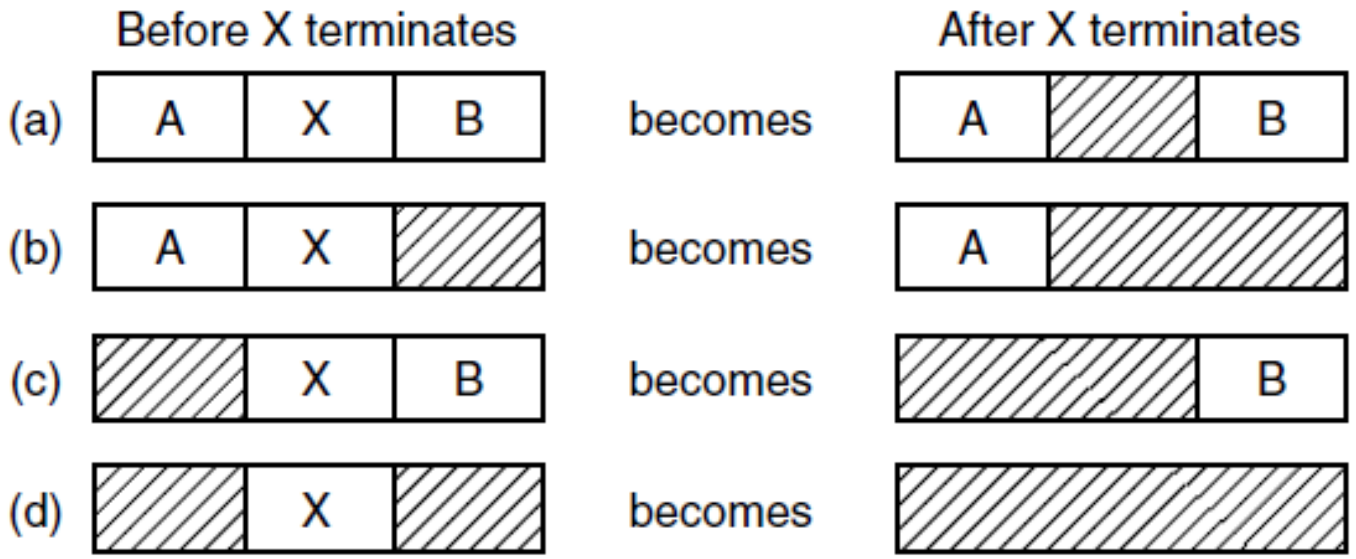


Figure 3-7. Four neighbor combinations for the terminating process, *X*. (double-linked list more convenient)

Memory Management Algorithms

- First fit
- Next fit
- Best fit
- Worst fit
- Quick fit

Memory Management Algorithms

- **First fit**
  - Scans along the list of segments until it finds a hole that is big enough
  - Fast because it searches as little as possible
- **Next fit**
  - Same as first fit, except it keeps tracks of where it is whenever it finds a suitable hole
  - Next search start from where it left off last time
  - Slightly worse than first fit [Bays, 1977]

Memory Management Algorithms

- **Best fit**
  - Searches the entire list and takes the smallest hole that is adequate
  - Slower than first fit because of searching entire list
  - More wasted memory due to tiny, useless holes
- **Worst fit**
  - Searches the entire list and takes the largest available hole
  - Not a very good idea

Memory Management Algorithms

- Maintaining separate lists for processes and holes can speed up all four algorithms
  - Hole list may be kept sorted on size
- **Quick fit**
  - Maintains separate lists for some of the more common size requested (4KB, 8KB, 12KB etc.)
  - Finding a hole of the required size is very fast
  - When a process terminates or swapped out, finding neighbors for merge is quite expensive

Virtual Memory

- There is a need to run programs that are too large to fit in memory
- Solution adopted in the 1960s, split programs into little pieces, called overlays
  - Kept on the disk, swapped in and out of memory
- Virtual memory : each program has its own address space, broken up into chunks called pages

Paging (1)

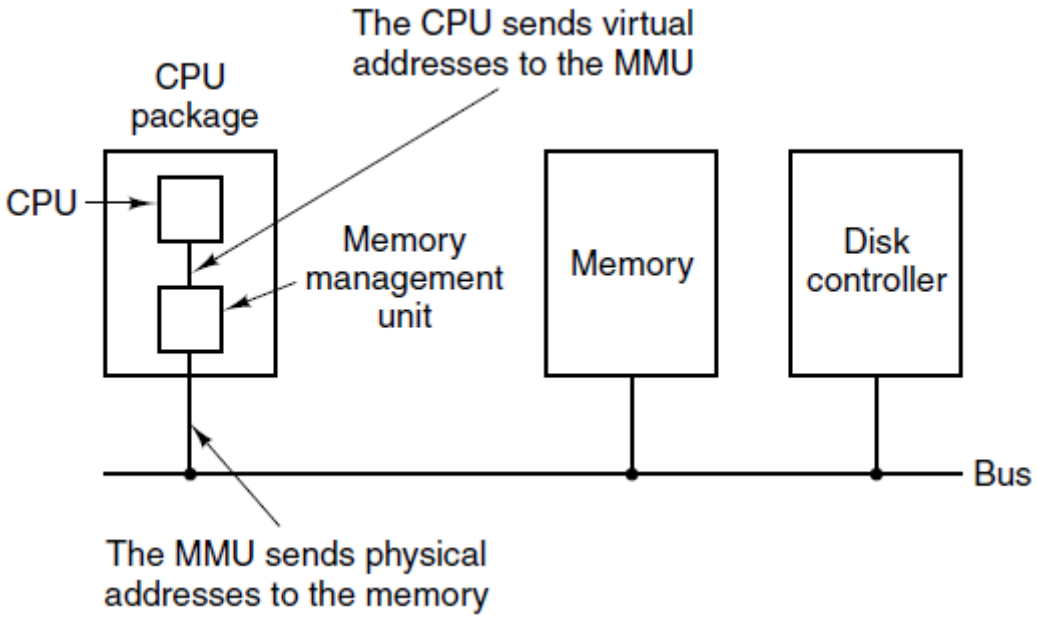


Figure 3-8. The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays. However, logically it could be a separate chip and was years ago.

Paging (2)

MOV REG, 0 (MMU receive)

Virtual page 0 (0K-4K)

Page frame 2 (8K-12K)

MOV REG, 8192 (MMU transmit)

MOV REG, 32780 (MMU receive)

Virtual page 8 (32K-36K)

Page frame X

Page fault and OS starts operation

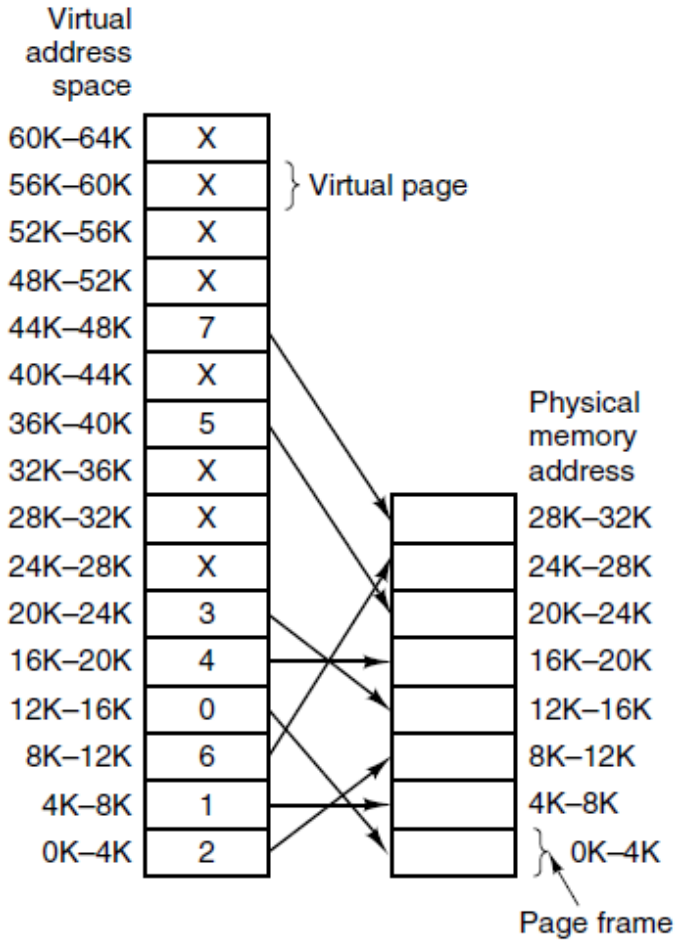


Figure 3-9. The relation between virtual addresses and physical memory addresses is given by the page table. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287

Paging (3)

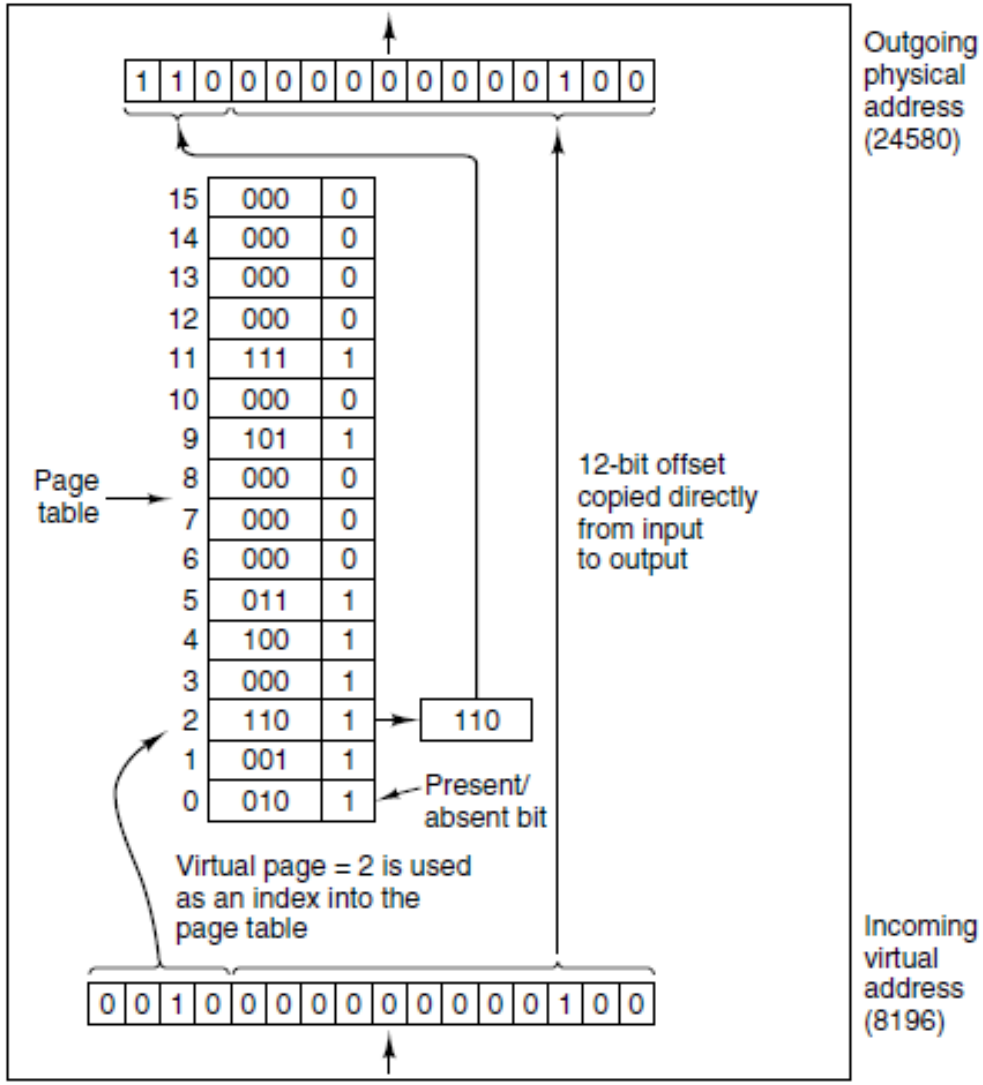


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

Structure of a Page Table Entry

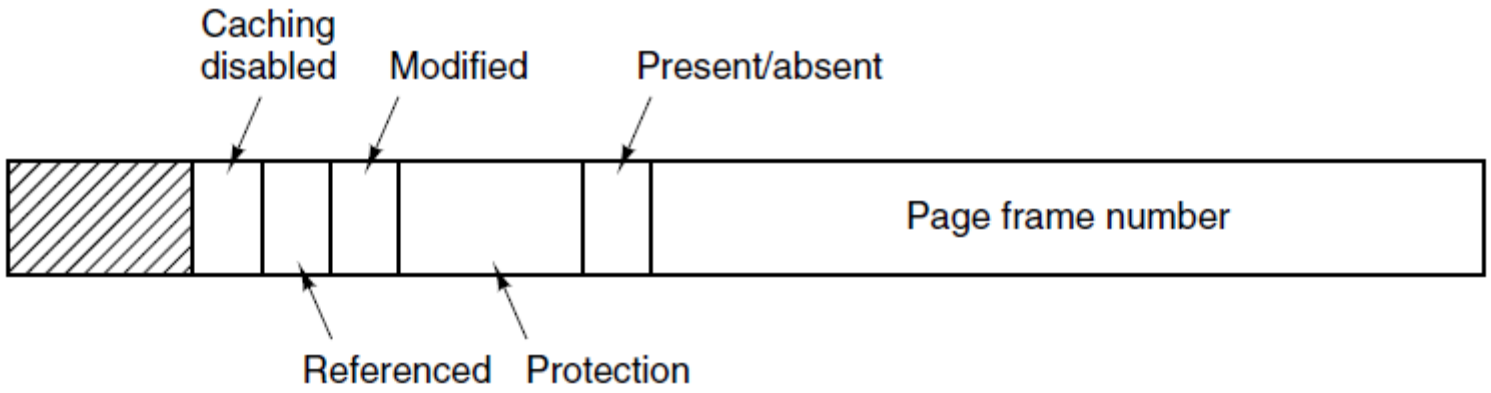


Figure 3-11. A typical page table entry.

Speeding Up Paging

Major issues faced:

1. The mapping from virtual address to physical address must be fast.
2. If the virtual address space is large, the page table will be large.

Each process needs its own page table (because it has its own virtual address space)

Simpler Design

- Single page table consisting of an array of fast hardware registers (one for each virtual page)
- When a process is started up, OS loads the registers with the process page table
- During process execution, no more memory references
- If page table is large then very expensive
- Load full page table at every context switch

Alternate Design

- The page table can be entirely in the main memory
- Hardware needs a single register pointing to the start of page table
- Context switch just by reloading one register
- Require more memory references to read page table entries, making it very slow

Translation Lookaside Buffers

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 3-12. A TLB to speed up paging.

Software TLB Management

- TLB entries are explicitly loaded by OS
- Various strategies to reduce TLB miss and cost of a TLB miss
  - OS can use it intuition to figure out which pages are likely to be used next
- Soft miss - page not in TLB but in memory
- Hard miss - page neither in TLB nor in memory
- Hard miss is million times slower than soft one

Page Fault Cost Varies

- Page table walk doesn't find the page in process' page table, so page fault
  - Minor page fault if the page in memory but not in the process' page table (brought in from disk by another process), no disk access, simple map
  - Major page fault if the page needs to be brought from disk
  - Program simply accessed an invalid address, so no mapping needed, OS kills with segmentation fault

Multilevel Page Tables

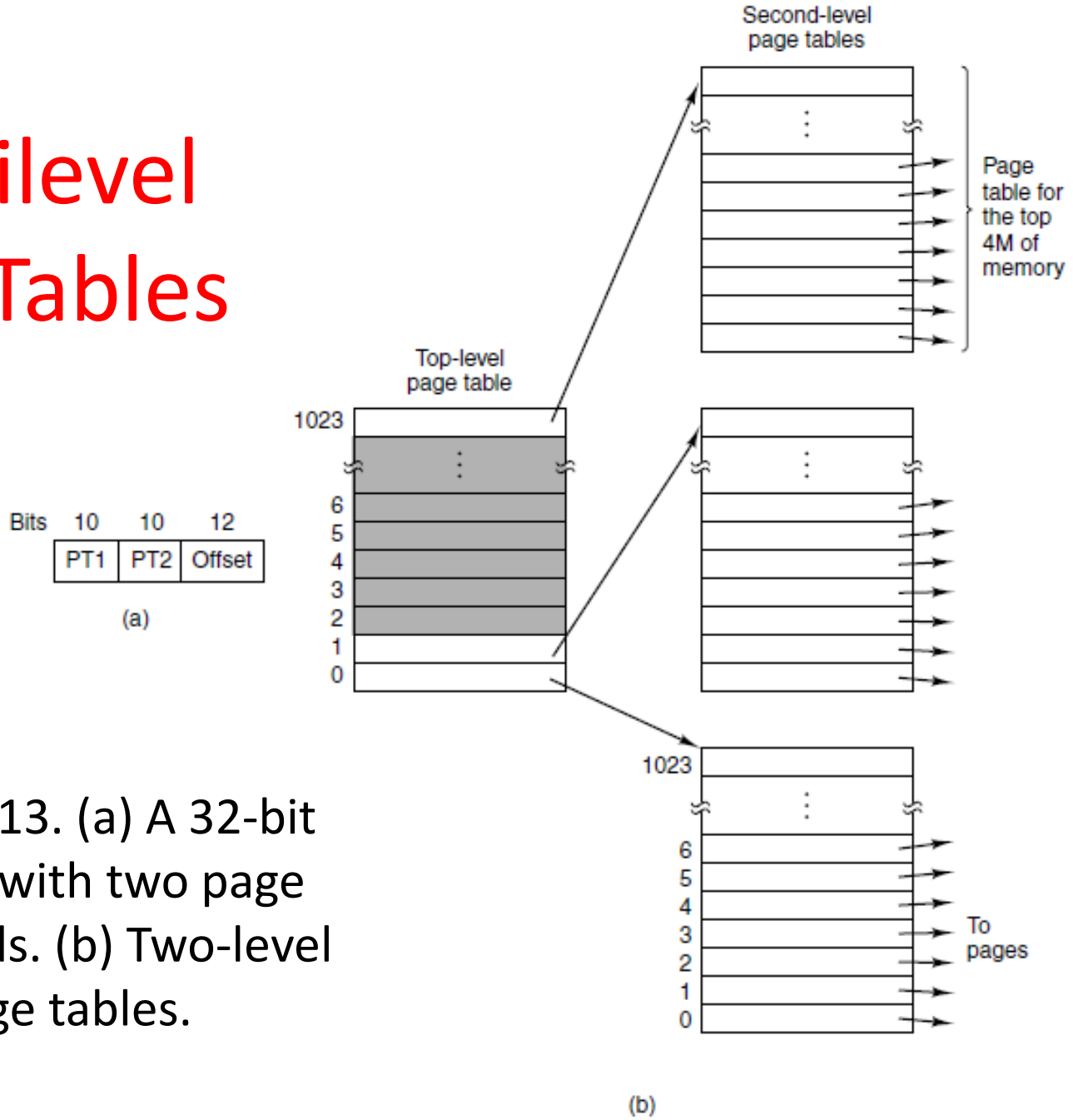


Figure 3-13. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

Inverted Page Tables

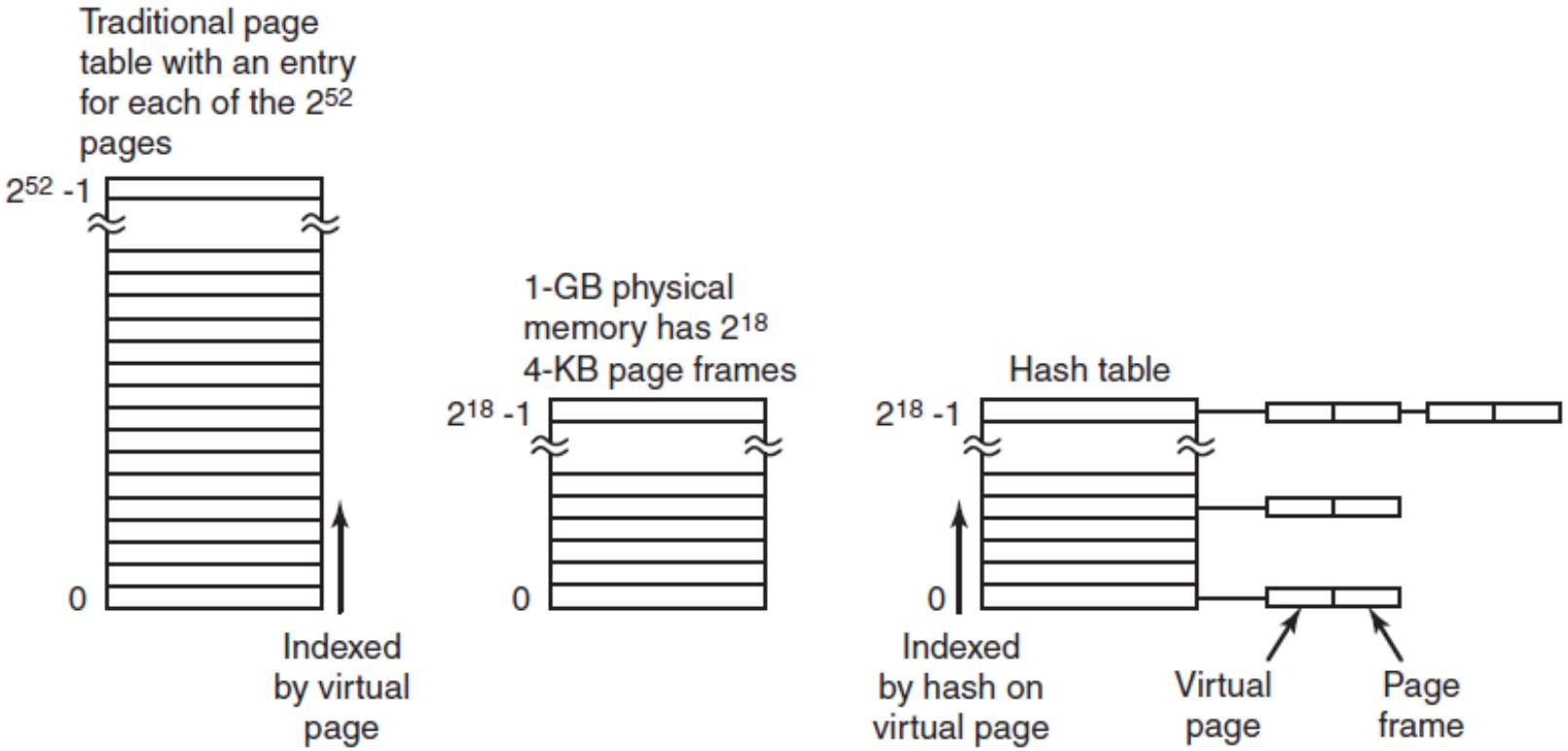


Figure 3-14. Comparison of a traditional page table with an inverted page table.

Page Replacement Algorithms

- Optimal algorithm
- Not recently used algorithm
- First-in, first-out (FIFO) algorithm
- Second-chance algorithm
- Clock algorithm
- Least recently used (LRU) algorithm
- Working set algorithm
- WSClock algorithm

Optimal Algorithm

- When a page fault occurs, some set of pages in memory
  - One will be referenced on the next instruction
  - Others may not be referenced until 10, 100 or perhaps 1000 instructions later
  - Each page can be labeled with the number of instructions executed before it is first referenced
- The optimal algorithm removes the page with the highest label

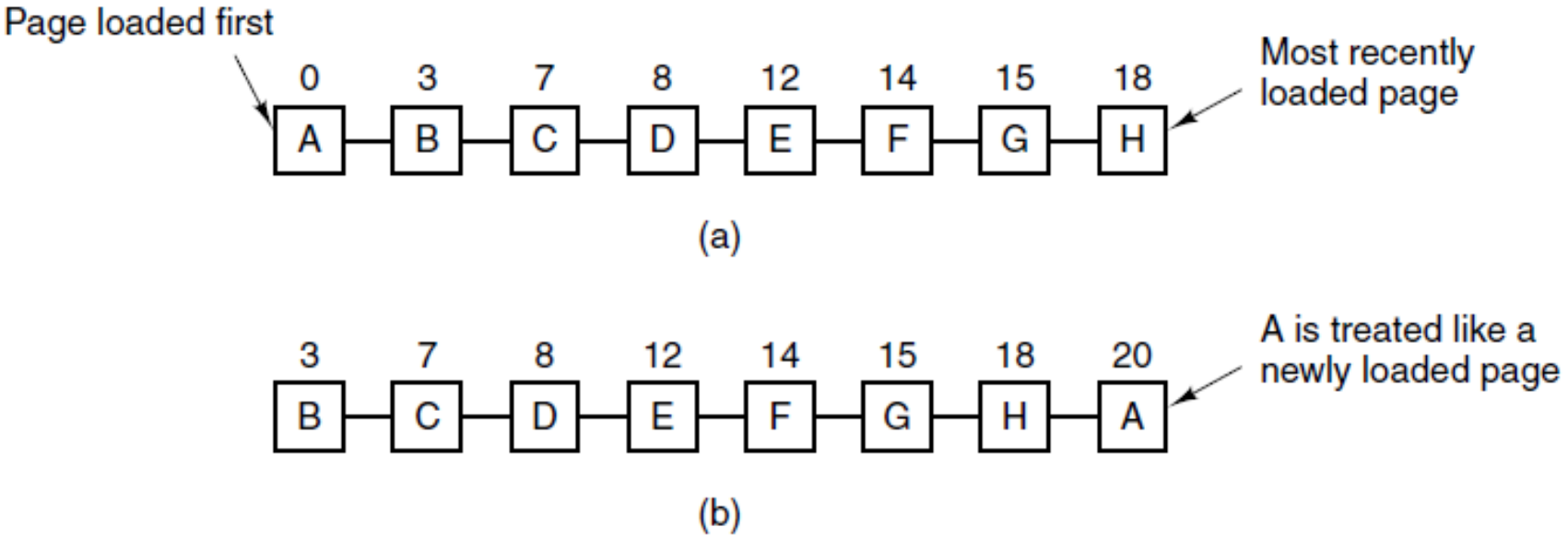
Not Recently Used Algorithm

- At page fault, system inspects pages
- Categories of pages based on the current values of their *R* and *M* bits:
  - Class 0: not referenced, not modified.
  - Class 1: not referenced, modified.
  - Class 2: referenced, not modified.
  - Class 3: referenced, modified.
- Removes a page at random from the lowest numbered non-empty class

FIFO Algorithm

- OS maintains a list of all pages in memory
  - The most recent arrival at the tail
  - The least recent arrival at the head
- On a page fault
  - The page at the head is removed
  - The new page added to the tail of the list
- Oldest page may still be useful
  - FIFO in its pure form is rarely used

Second-Chance Algorithm

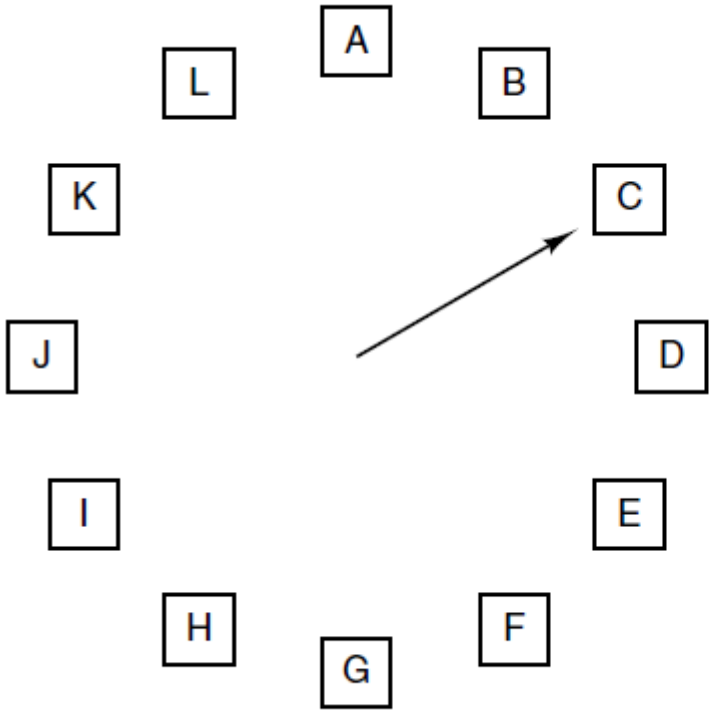


If R = 0 for the oldest page , replace immediately

If R = 1, clear R bit and put at the end

Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their load times.

Clock Page Replacement Algorithm



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

Figure 3-16. The clock page replacement algorithm.

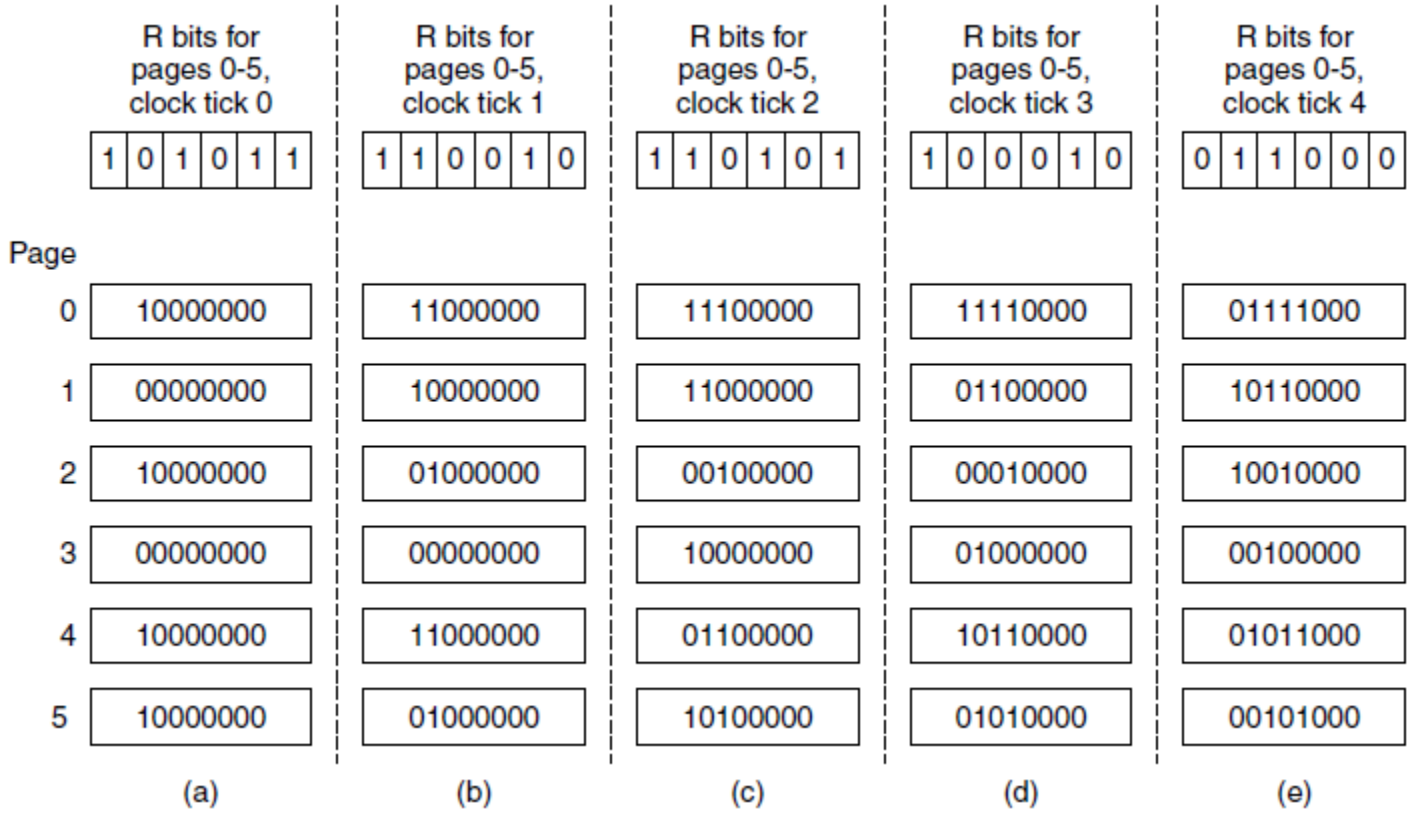
LRU Algorithm

- When a page fault occurs, throw out the page that has been unused for the longest time
- OS needs to maintain a linked list of all pages in memory
  - Most recently used page at the front, least recently used at the rear
  - Update the list on every memory references
  - Finding a page in the list, deleting it, and then moving it to the front

LRU Algorithm

- Alternate is a special hardware with a 64 bit counter (C)
  - C is automatically incremented after each instruction
  - After each memory reference, the current value of C is stored in the page table entry for the page just referenced
  - When a page fault occurs, throw out the page with lowest C

Simulating LRU in Software



Aging algorithm is a modification of NFU (Not Frequently Used)

Figure 3-17. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

Demand Paging

- In the purest form, processes are started up with none of their pages in memory
- CPU tries to fetch the first instruction
  - It gets a page fault
  - OS brings the page containing the first instruction
- After a while, the process has most of the pages it needs and settles down to run with relatively few page faults

Working Set

- The set of pages that a process is currently using is its **working set**
- If the entire working set is in memory, the process will run without causing many faults
- But If the available memory is too small to hold the entire working set, the process will cause many page faults and run slowly
- Page faults every few instructions - **thrashing**

Working Set

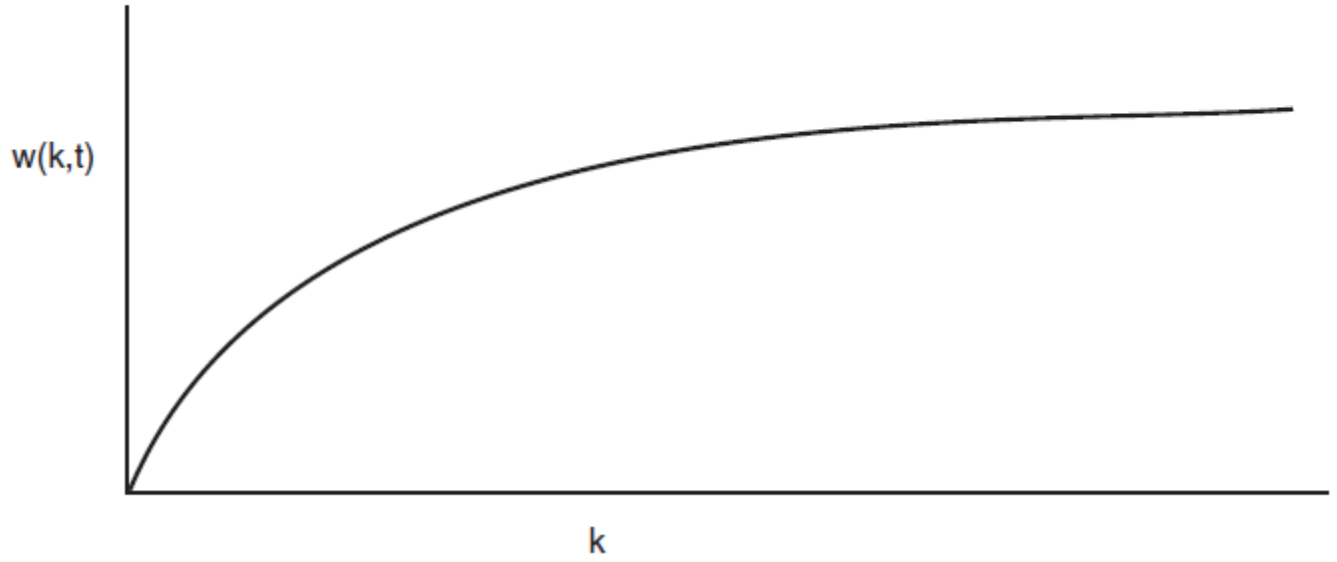


Figure 3-18. The working set is the set of pages used by the *k* most recent memory references. The function *w(k, t)* is the size of the working set at time *t*.

Working Set Model

- Many paging systems try to keep track of each process' working set and make sure it is in memory before letting the process run
- This approach is called the working set model
- Loading the pages before letting processes run is called prepaging

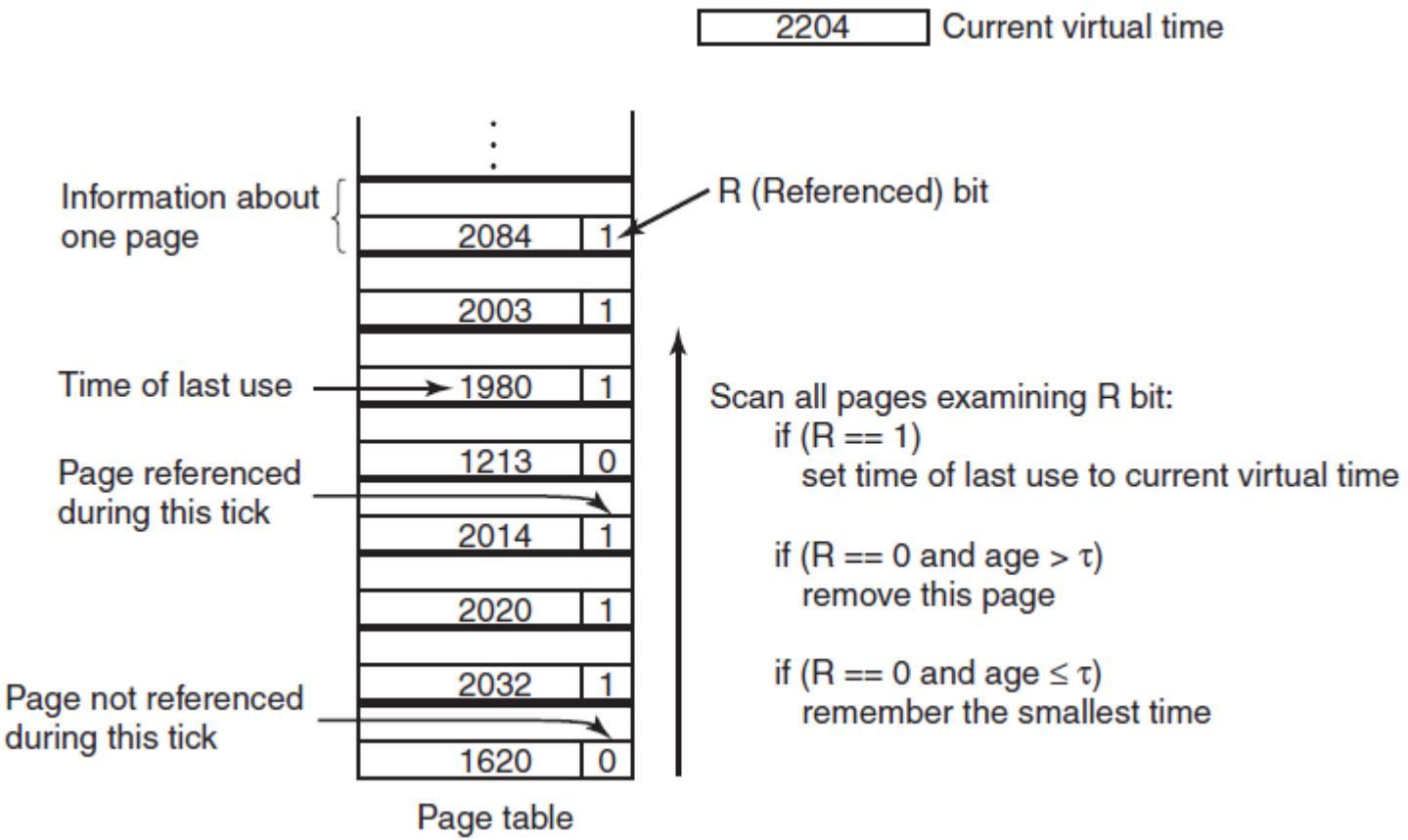
Working Set Model

- To implement working set model, OS keeps track of which pages are in the working set
- This leads to a page replacement algorithm
  - When a page fault occurs, find a page not in the working set and evict it
- Working set is the set of pages used in the k most recent memory references
  - To implement any working set algorithm, some value of k must be chosen in advance

Working Set Algorithm (1)

- We can define working set as the set of pages used during the past T secs. of execution time
  - For each process, only its own execution time counts
- The amount of CPU time a process has actually used since it started – current virtual time
  - Working set of a process is the set of pages it has referenced during past τ seconds of virtual time

Working Set Algorithm (2)



age = current virtual time – time of last use

smallest time of last use implies greatest age

Figure 3-19. The working set algorithm.

WSClock Algorithm (1)

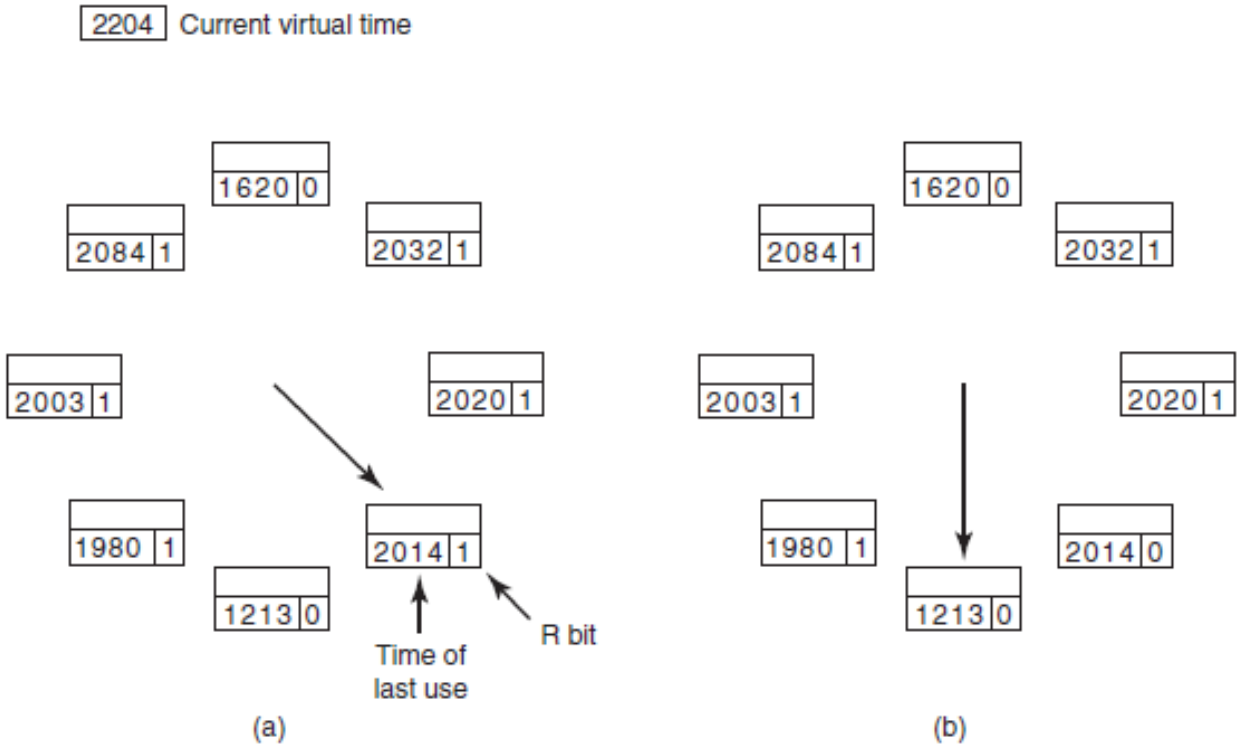


Figure 3-20. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when R = 1.

WSClock Algorithm (2)

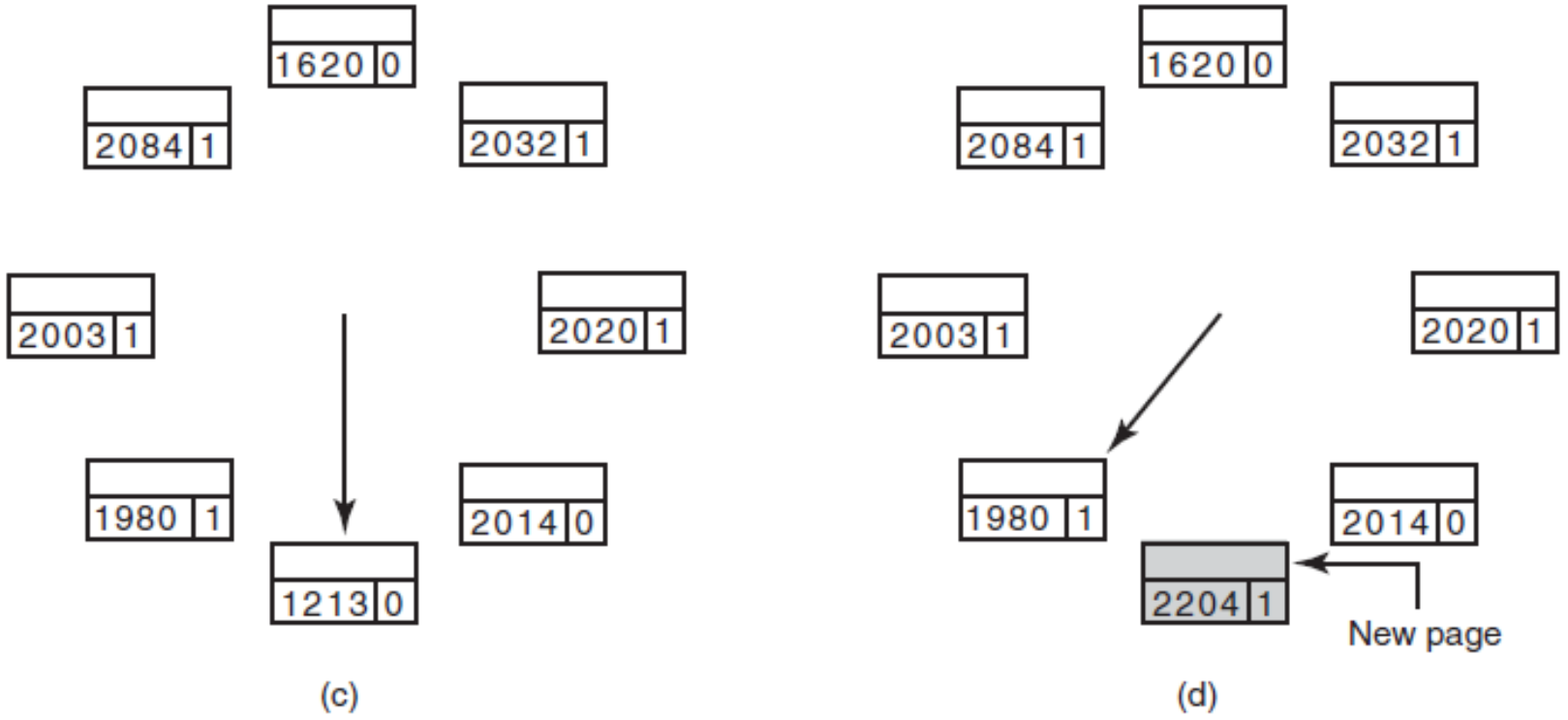


Figure 3-20. Operation of the WSClock algorithm. (c) and (d) give an example of R = 0.

Summary of Page Replacement Algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second, chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Figure 3-21. Page replacement algorithms discussed in the text.

Local versus Global Allocation Policies (1)

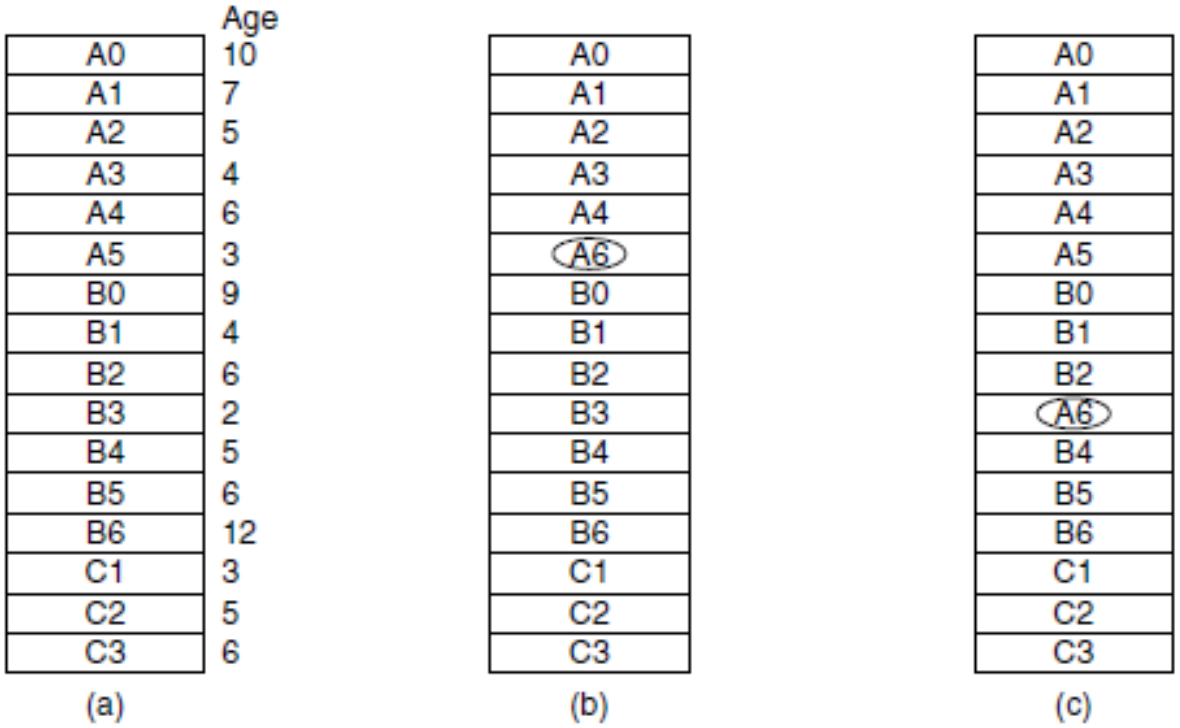


Figure 3-22. Local versus global page replacement.
(a) Original configuration. (b) Local page replacement.
(c) Global page replacement.

Local versus Global Allocation Policies (2)

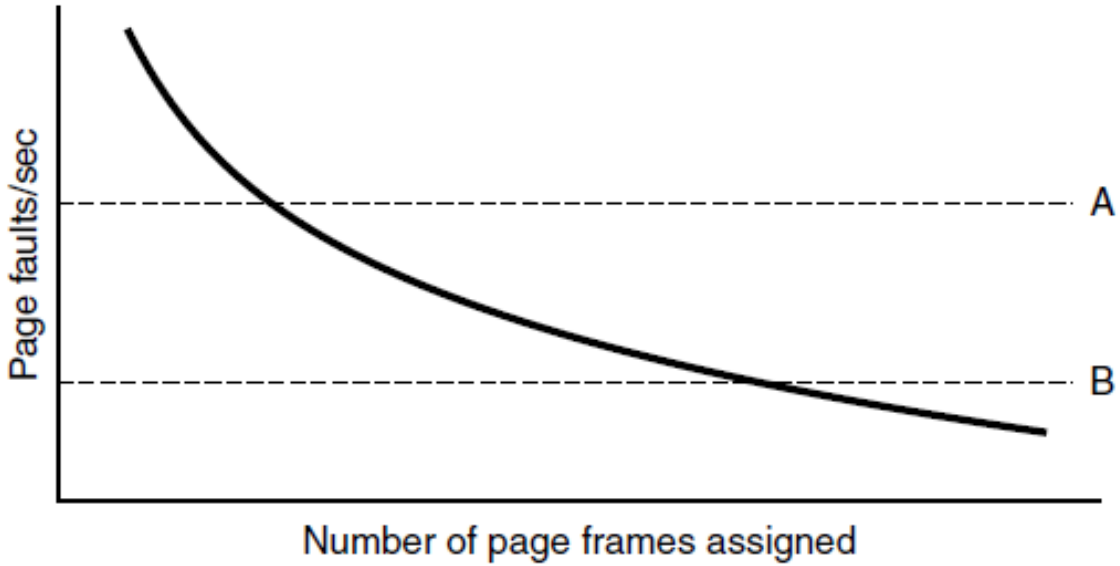


Figure 3-23. Page fault rate as a function of the number of page frames assigned.

Page Size

- Page size is a parameter chosen by OS
- Determining the best page size requires balancing several competing factors
  - so no overall optimum
- Small page size
  - Randomly chosen text, data, or stack segment will not fill an integral number of pages
  - n segments in memory and page size of p bytes, $\frac{np}{2}$ bytes will be wasted on internal fragmentation

Page Size

- Small page size
  - Program consisting of 8 sequential phases of 4 KB each, 4 KB page size is enough
  - With page size of 16 KB or 32 KB, more waste
  - Program will need many pages, and thus a large page table
  - Transfer to and from disk are generally a page at a time, seek and rotation dominate, so small page transfer takes almost as much time as large page

Page Size

- Small page size
  - Small pages use up much valuable space in the TLB
  - A program uses 1 MB of memory with working set of 64 KB, with 4 KB pages 16 entries in TLB, with larger page size much less TLB entry
- To balance all these trade-offs, OS sometimes use different page sizes for different parts of the system

Page Size

- Average process size: s bytes, the page size: p bytes, each page entry: e bytes
- The approximate number of pages needed per process is then $\frac{s}{p}$, occupying $\frac{se}{p}$ bytes of space
- The wasted memory in the last page of the process for internal fragmentation: $\frac{p}{2}$
- Total overhead: $\frac{se}{p} + \frac{p}{2}$

Page Size

- The first term is large when the page size is small, the second term is large when page size is large
- The optimum must lie somewhere in between
- By taking the first derivative with respect to p and equating it to zero, we get

$-\frac{se}{p^2} + \frac{1}{2} = 0$, optimum page size, $p = \sqrt{2se}$

- For s = 1 MB, e = 8 bytes, optimum size 4 KB

Separate Instruction and Data Spaces

- Most computers have a single address space that holds both program and data
- Another approach is to have separate address spaces for instructions (program text) and data (I-space and D-Space)
- Each one has its own page table, with its own mapping
  - Instruction fetch must use I-space page table, data must go through D-space page table

Separate Instruction and Data Spaces

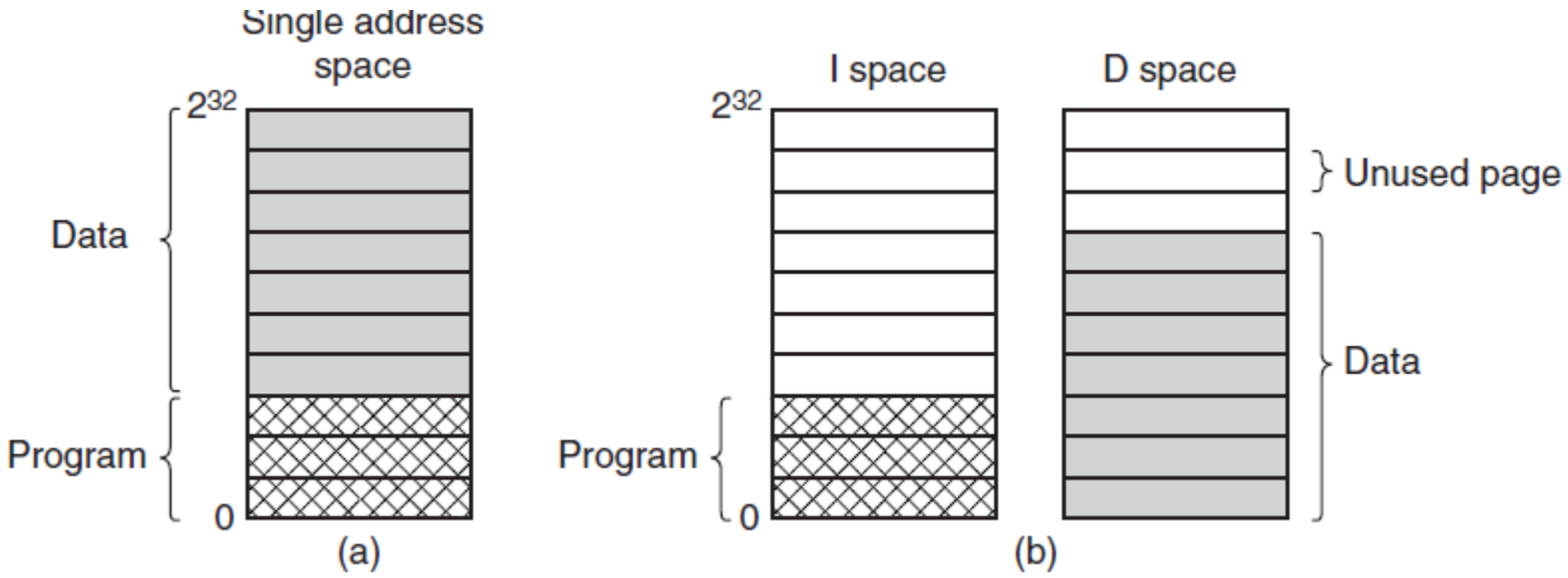


Figure 3-24. (a) One address space.
(b) Separate I and D spaces.

Shared Pages

- It is common for several users to be running same program at the same time
- Single user may be running several programs that use the same library
- Page sharing is more efficient
- With separate I-space and D-space, each process has two pointers in its process table
  - one to I-space page table and another to D-space page table

Shared Pages

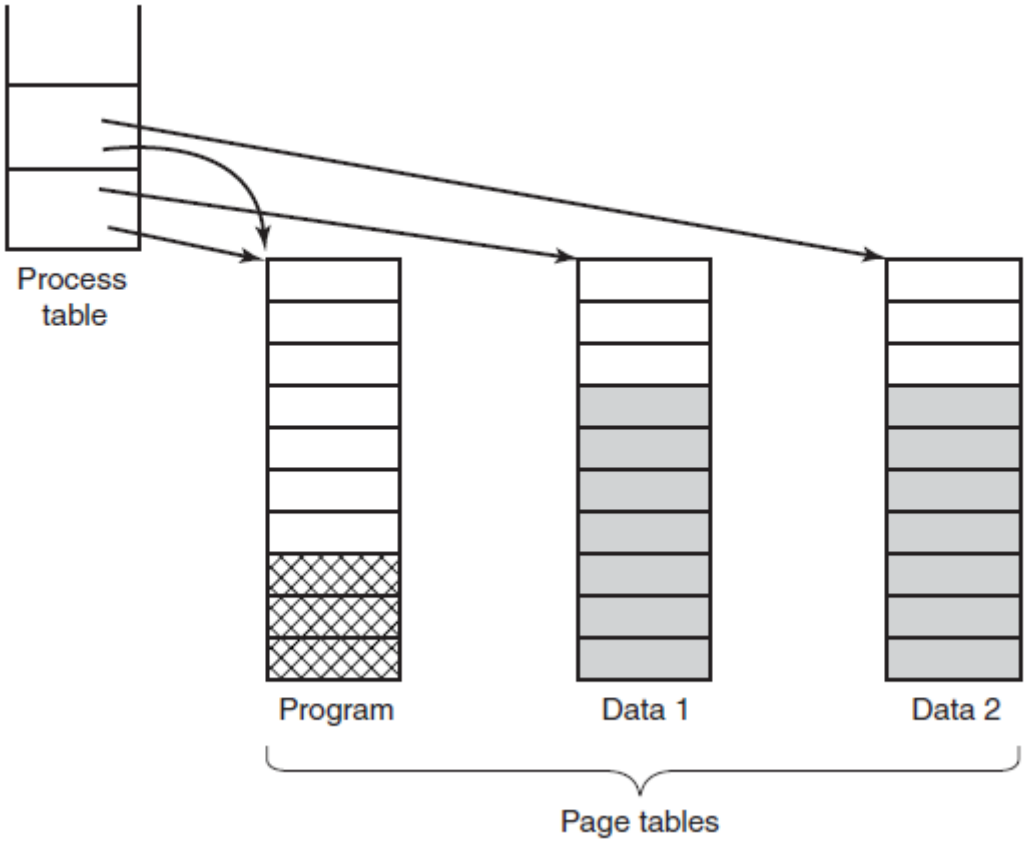


Figure 3-25. Two processes sharing the same program sharing its page table.

Shared Pages

- A and B, both running editor, sharing pages
- Scheduler decides to remove A from memory
  - Evicts all its page
  - Filling the empty pages with some other programs
  - B will generate a large number of page faults to bring them back in again
- When A terminates, need to find shared pages
  - Searching page table is too expensive, separate data structures to keep track shared pages

Shared Pages

- What happened with fork() **– copy on write**
  - After a fork, the parent and child are required to share both program text and data
  - Each process with own page table and both of them point to same set of pages
  - No copying of pages at fork time
  - All the data pages are mapped as READ ONLY
  - Any update, the read-only violation trap to the OS
  - Copies made with both as READ/WRITE

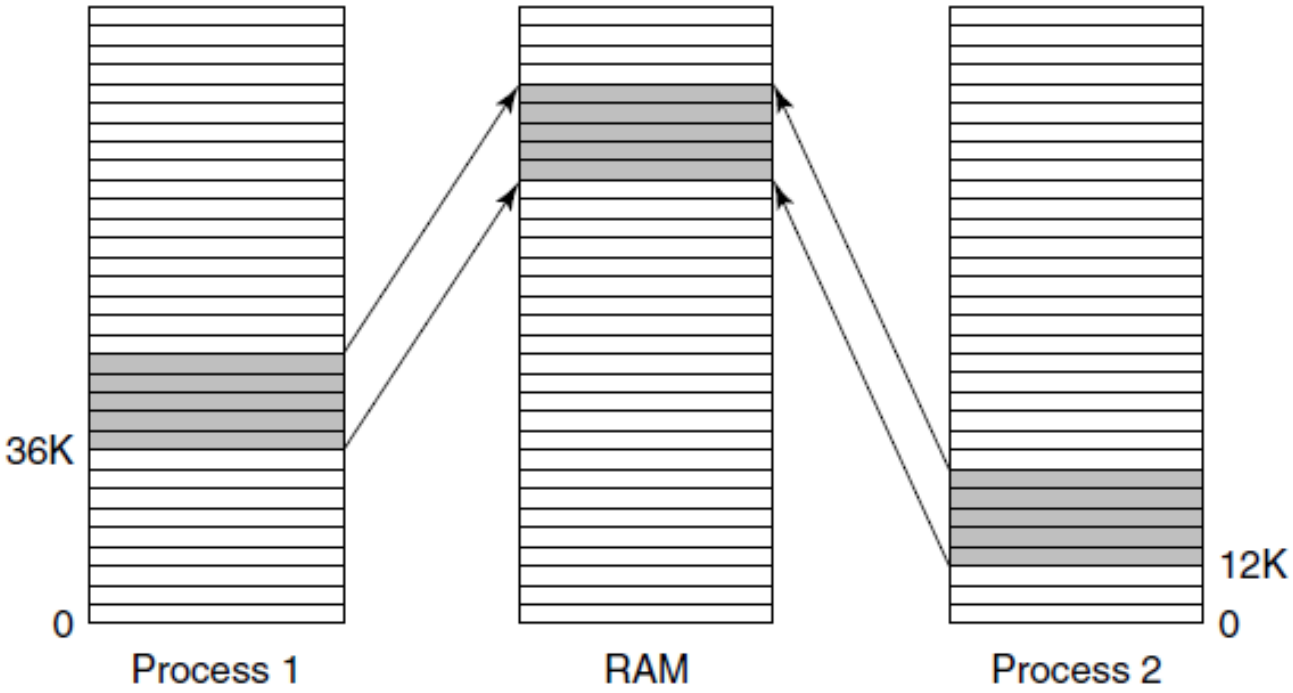
Shared Libraries

- In modern systems, there are many large libraries used by many processes
- Statically binding all these libraries to every executable program is not feasible
- ld *.o –lc –lm (UNIX command)
  - Links all the .o (object) files and then scans two libraries /usr/lib/libc.a and /usr/lib/libm.a
  - Any functions called in the object files but not present there are sought in the libraries

Shared Libraries

- When a program is linked with shared libraries, linker includes a small stub routine that binds to the called function at run time
- Shared library (DLL/Windows) can be loaded
  - when the program is loaded or
  - when the functions are called for the first time
  - If already loaded, no need to load it again
- If a function is updated, it is not necessary to recompile the programs that call it

Shared Libraries



Compile shared libraries with a special compiler flag telling the compiler not to produce any instructions with absolute address, rather use relative addresses (position-independent code)

Figure 3-26. A shared library being used by two processes.

Memory Mapped Files

- Process can issue a system call to map a file onto a portion of its virtual address space
  - Generally, no pages are brought in at the time of the mapping
  - As the pages are touched, they are demand paged in one page at a time
  - When process exists or explicitly unmaps the file, all the modified pages are written back to the file on disk

Cleaning Policy

- Paging works best when there is an abundant supply of free page frames that can be claimed as page fault occurs
- Paging system generally have a background process, paging daemon
  - If two few page frames are free, it begins selecting pages to evicts
  - If these pages have been modified, they are written to disk

Cleaning Policy

- Implemented with two-handed clock
- Front hand controlled by paging daemon
  - When it points to a dirty page, that page is written back to disk and front hand is advanced
  - When it points to a clean page, it is just advanced
- Back hand is used for page replacement
  - Same as standard clock algorithm
  - The probability of the back hand hitting a clean page is increased due to paging daemon

OS Involvement with Paging

- **Process Creation Time**
  - When a new process is created, the OS has to determine the initial size of program and data and create page table for them
  - Space has to be allocated in memory for the page table and it has to be initialized
  - The page table need not to be resident when the process is swapped out but has to be in the memory when the process is running

OS Involvement with Paging

- **Process Creation Time**
  - Space has to be allocated in the swap area on disk so that when a page is swapped out, it has somewhere to go
  - The swap area also has to be initialized with program text and data so that when the new process starts getting page faults, the pages can be brought in
  - Information about the page table and swap area on disk must be recorded in the process table

OS Involvement with Paging

- **Process Execution Time**
  - When a process is scheduled for execution, the MMU has to be reset for the new process and the TLB flushed, to get rid of traces of the previously executing process
  - The new process' page table has to be made current, usually by copying or a pointer to it to some hardware registers
  - Optionally, some or all of the process' pages can be brought into memory to reduce page faults

OS Involvement with Paging

- **Page Fault**
  - When a page fault occurs, OS has to read out hardware registers to determine which virtual address caused the fault
  - It must compute which page is needed and locate the page in the disk
  - It must then find an available page frame in which to put the new page, evicting some old page if needed

OS Involvement with Paging

- **Page Fault**
  - Then it must read the needed page into the page frame
  - It must back up the program counter to have it point to the faulting instruction and let that execute again

OS Involvement with Paging

- **Process Termination Time**
  - When a process exits, OS must release its page table, its pages and the disk space that the pages occupy when they are on disk
  - If some of the pages are shard with other processes, the pages in memory and on disk can be released only when the last process using them terminated

Page Fault Handling (1)

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

Page Fault Handling (2)

5. If frame selected dirty, page is scheduled for transfer to disk, context switch takes place, suspending faulting process
6. As soon as frame clean, operating system looks up disk address where needed page is, schedules disk operation to bring it in.
7. When disk interrupt indicates page has arrived, tables updated to reflect position, and frame marked as being in normal state.

Page Fault Handling (3)

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

Instruction Backup

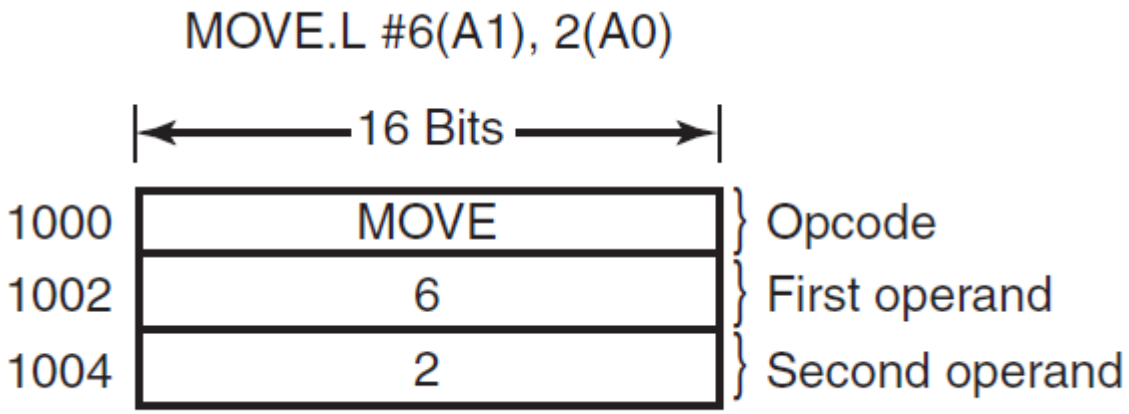


Figure 3-27. An instruction causing a page fault.

Instruction Backup

- To restart the instruction, OS must determine where the first byte of instruction is
- Difficult/Impossible for OS to determine unambiguously where the instruction began
- Some use autoincrementing
- CPU designers provide a solution
  - PC is copied to a hidden internal register before each instruction is executed
  - Second register to tell which register have inc/dec

Locking Pages in Memory

- A process issued a system call to read from some file to a buffer within its address space
  - Process is suspended, another is allowed to run
  - This other process gets a page fault
- With global algorithm, there is chance that the page involving I/O of the suspended process will be chosen to be removed from memory
  - Page can be locked, known as **pinning**

Backing Store

- The simplest algorithm for allocating page on the disk is a special swap partition
  - On a separate disk from the file system
  - Most UNIX systems work like this
- Initialize swap area before a process can start
  - Copy the entire process image to swap area, so that it can be brought in as needed
  - Load the entire process in memory and let it be paged out as needed

Backing Store

- But process can increase in size after starting
  - Text usually fixed, data and stack can grow
  - Better to reserve separate swap areas
- The other extreme is to allocate nothing in advance
  - Allocate disk space for page when swapped out
  - Deallocate when swapped back in
  - Pages in memory do not tie up any swap space

Backing Store

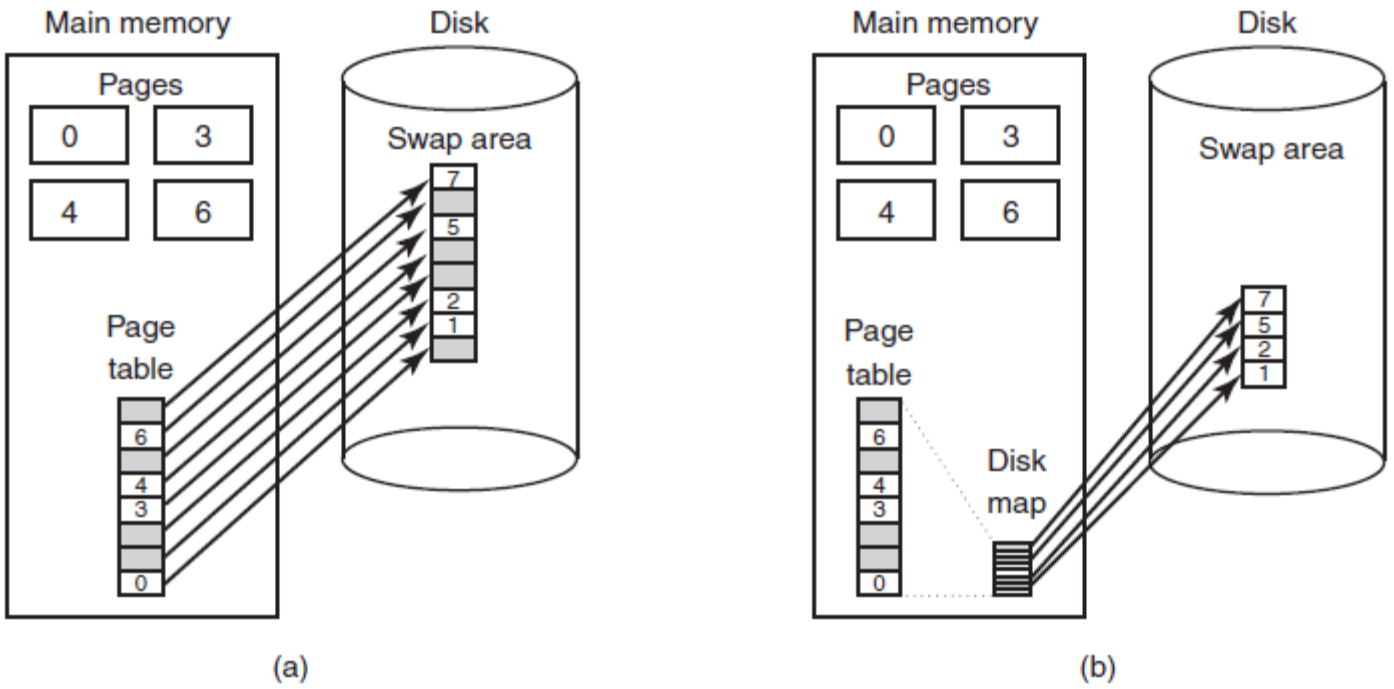


Figure 3-28. (a) Paging to a static swap area. (b) Backing up pages dynamically.

Separation of Policy and Mechanism

Memory management system is divided into three parts

1. A low-level MMU handler (machine-dependent)
2. A page fault handler that is part of the kernel (machine-independent)
3. An external pager running in user space (large determines the policy)

Separation of Policy and Mechanism – Process Starts Up

- The external pager is notified 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

Separation of Policy and Mechanism – Page Fault Occurs

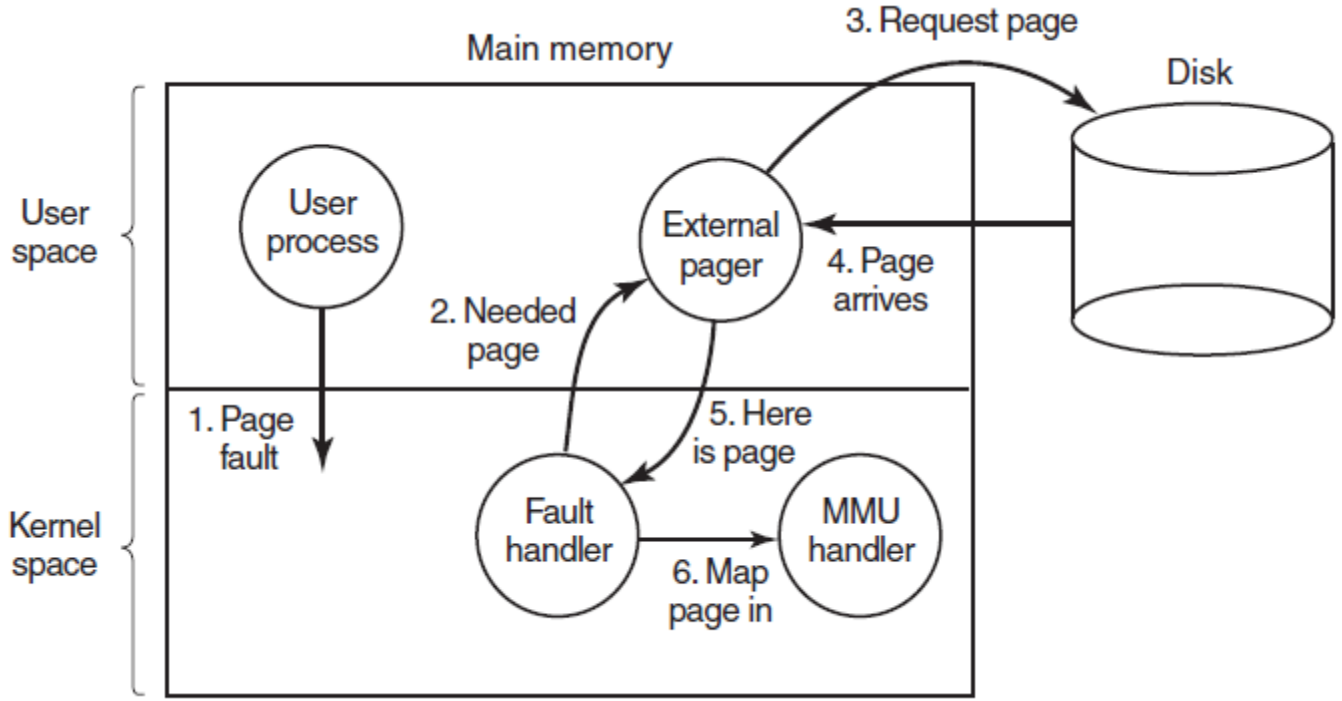


Figure 3-29. Page fault handling with an external pager.

Separation of Policy and Mechanism – Page Replacement

- Where to put the page replacement algorithm
  - Cleanest to have it in the external pager
  - But pager doesn't have access to the R and M bits
  - Either some mechanism is needed to pass this information to the external pager
  - Or page replacement algorithm must go to kernel
  - Fault handler tells the pager the selected page for eviction and data, pager writes to the disk
  - More modular code and greater flexibility

Segmentation

- The virtual memory discussed so far is one-dimensional
  - the virtual addresses go from 0 to some maximum address
  - one address after another
- Having two or more separate virtual address spaces may be much better then having only one

Segmentation (1)

Examples of tables generated by compiler:

1. The source text being saved for the printed listing
2. The symbol table, names and attributes of variables.
3. The table containing integer and floating-point constants used.
4. The parse tree, syntactic analysis of the program.
5. The stack used for procedure calls within compiler.

Segmentation (2)

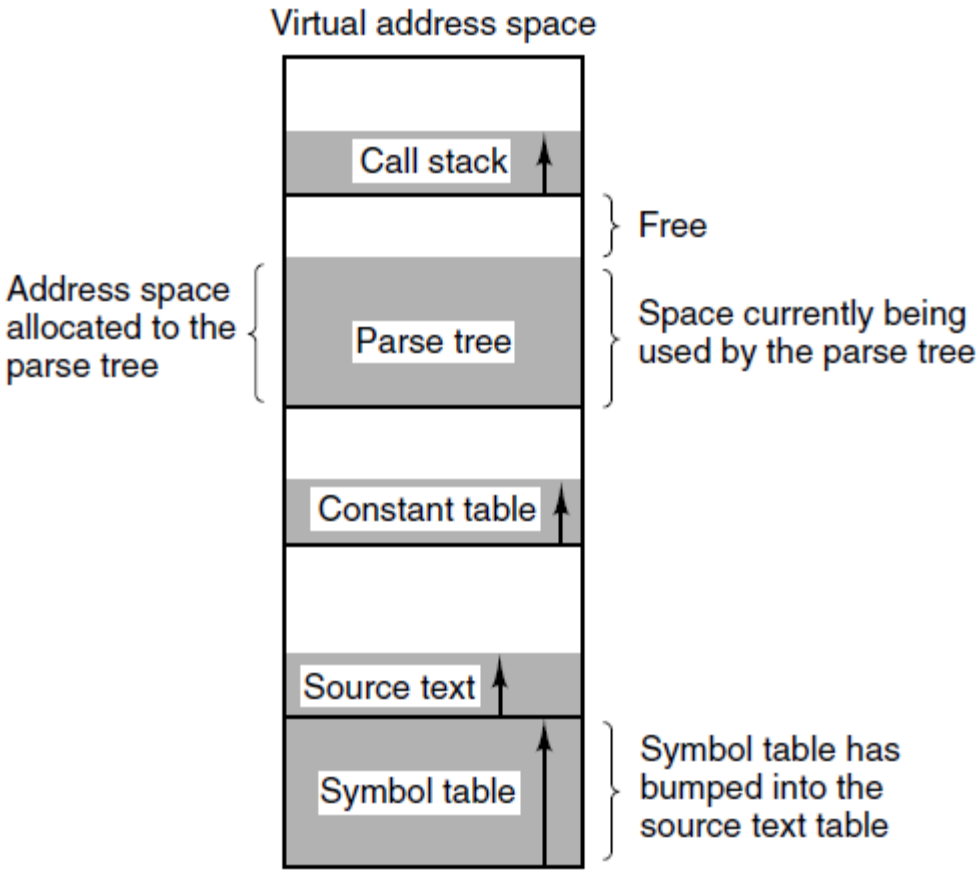
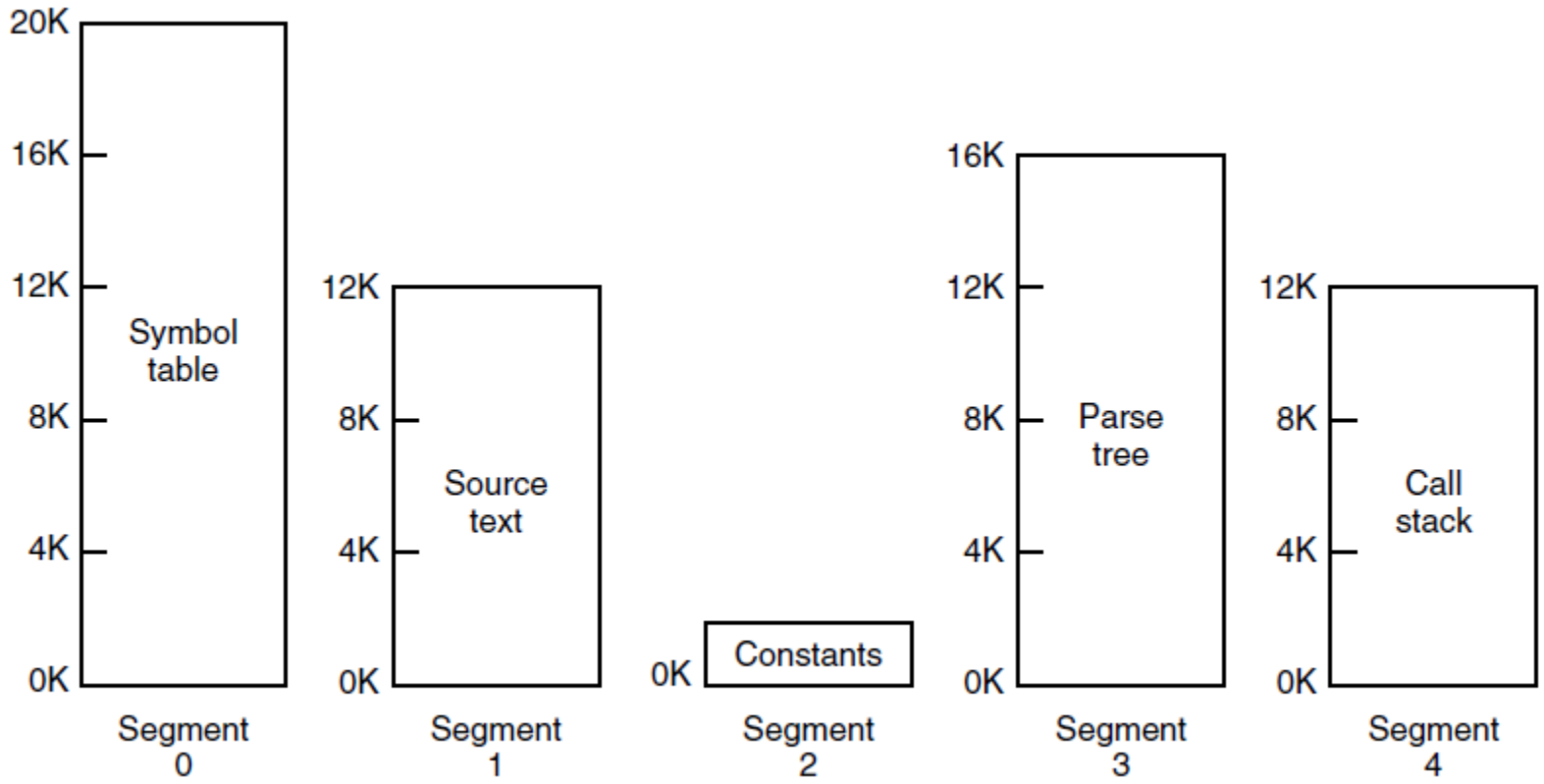


Figure 3-30. In a one-dimensional address space with growing tables, one table may bump into another.

Segmentation (3)



A straightforward solution is to provide many completely independent address spaces, which are called segments

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

Segmentation (4)

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

Figure 3-32. Comparison of paging and segmentation

Implementation of Pure Segmentation

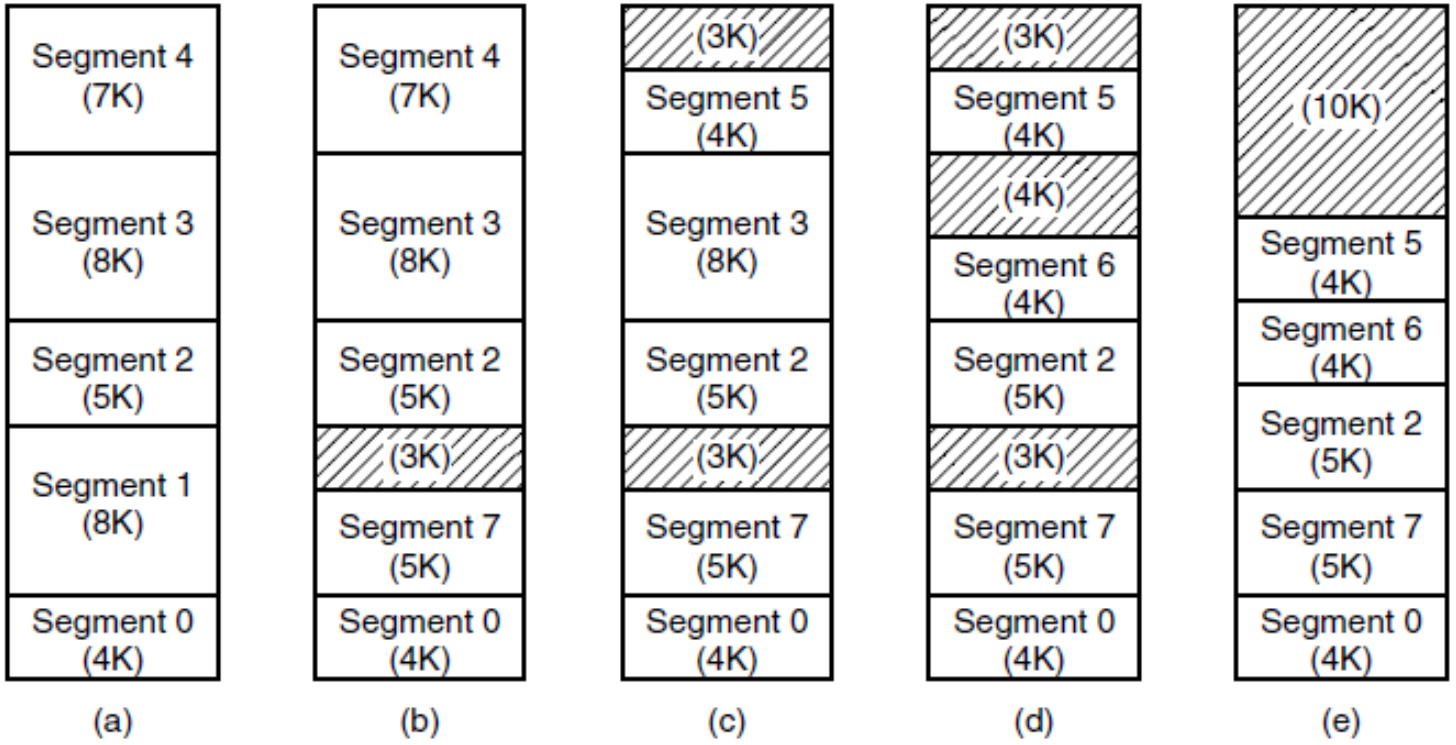


Figure 3-33. (a)-(d) Development of checkerboarding.
(e) Removal of the checkerboarding by compaction.

MULTICS

- One of the most influential OS ever having had a major influence on
  - UNIX, x86 memory architecture, TLB
- Research project at MIT in 1969
  - last MULTICS system was shutdown in 2000, a run of 31 years
- Ideas developed in MULTICS are as valid and useful now as they were in 1965

Segmentation with Paging: MULTICS (1)

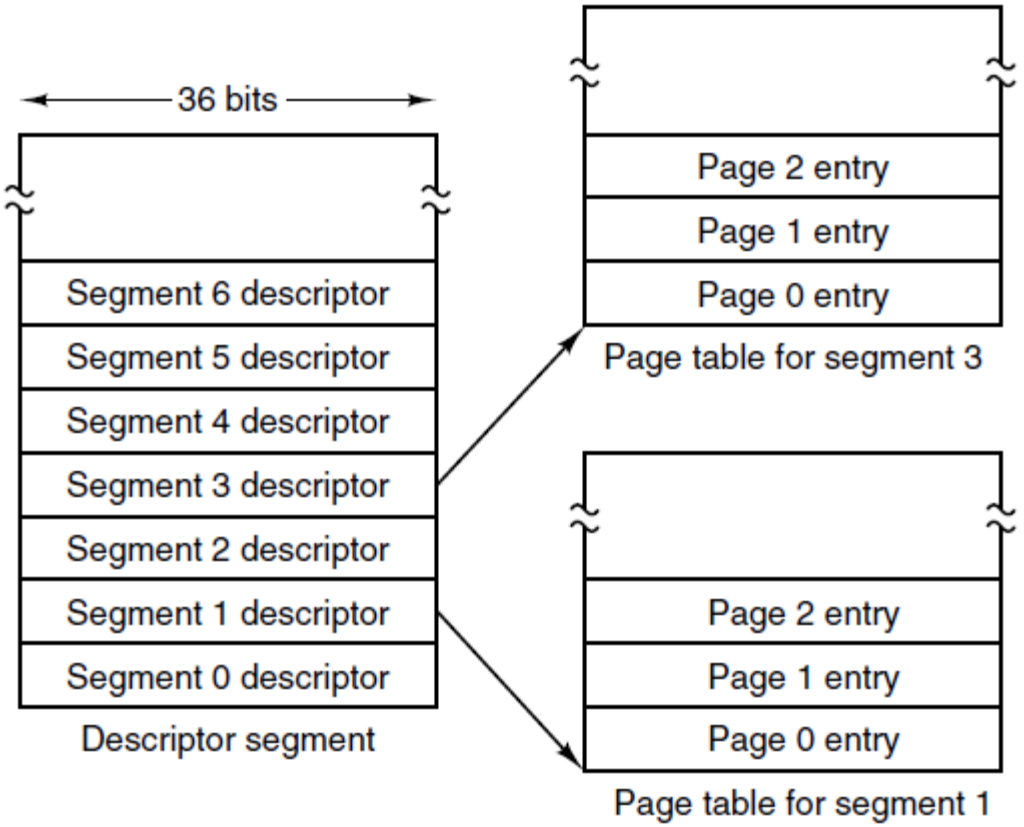


Figure 3-34. The MULTICS virtual memory. (a) The descriptor segment pointed to the page tables.

Segmentation with Paging: MULTICS (2)

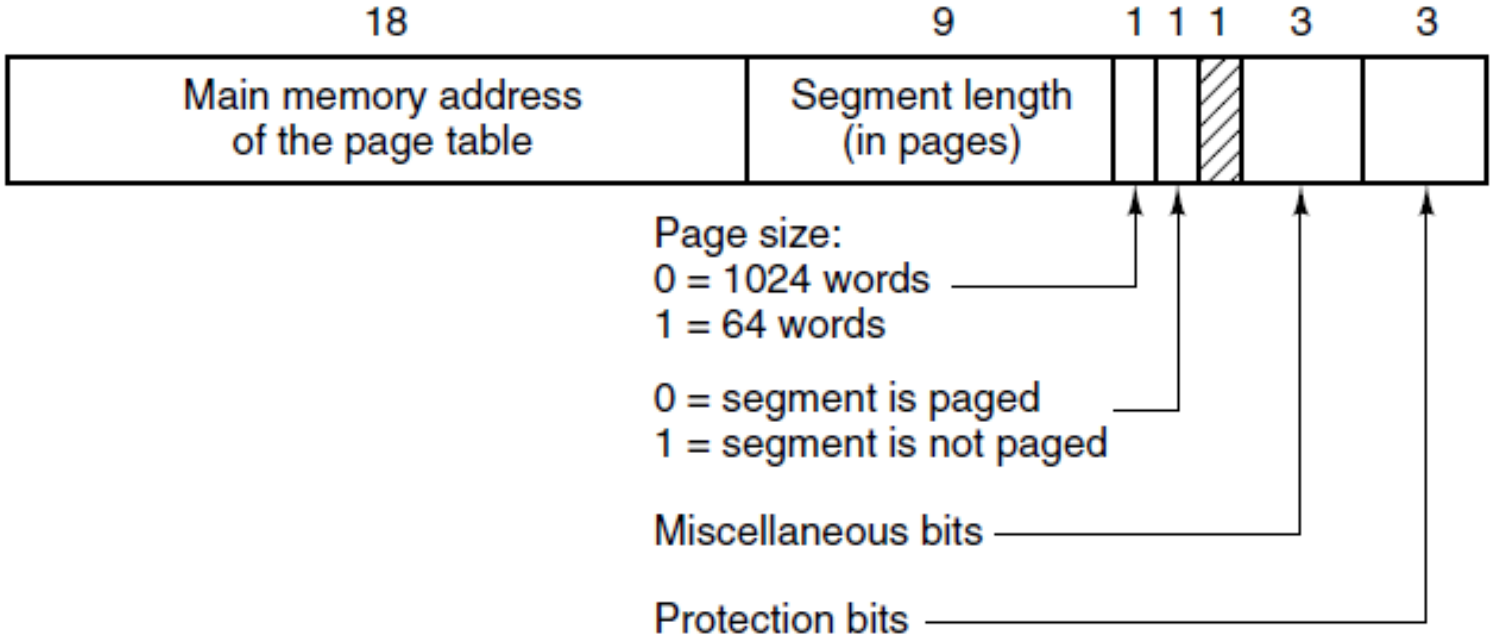


Figure 3-34. The MULTICS virtual memory. (b) A segment descriptor. The numbers are the field lengths.

Segmentation with Paging: MULTICS (3)

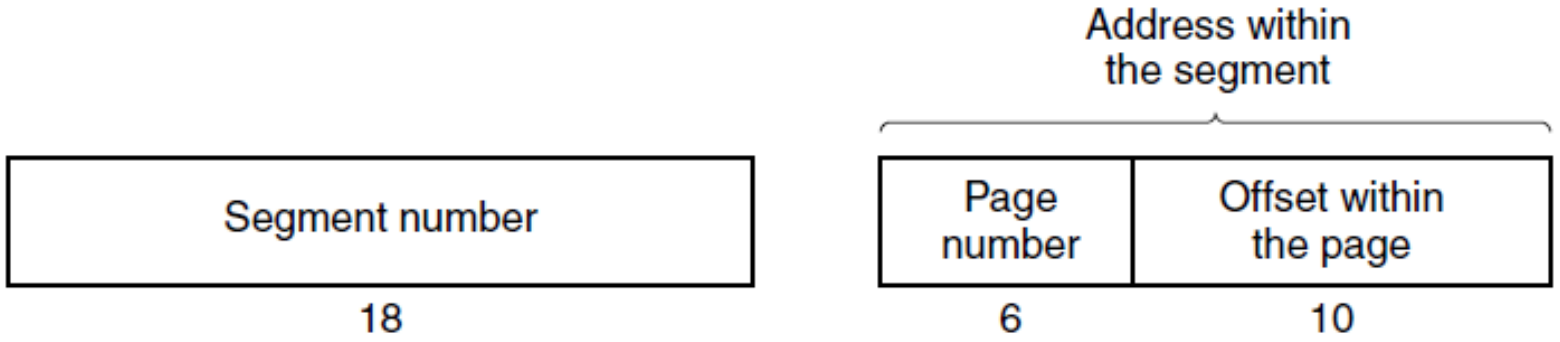


Figure 3-35. A 34-bit MULTICS virtual address.

Segmentation with Paging: MULTICS (4)

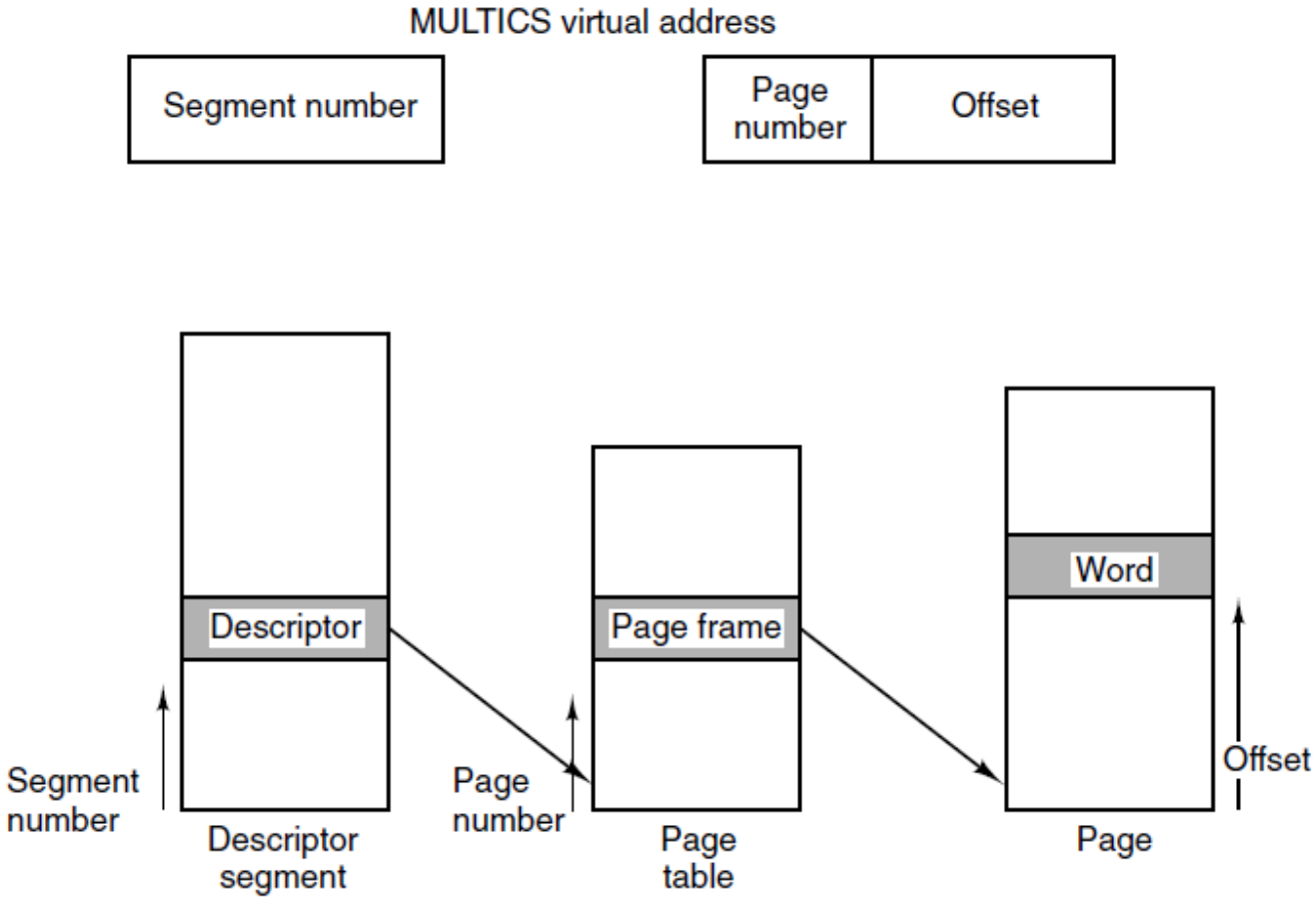


Figure 3-36. Conversion of a two-part MULTICS address into a main memory address.

Segmentation with Paging: MULTICS (5)

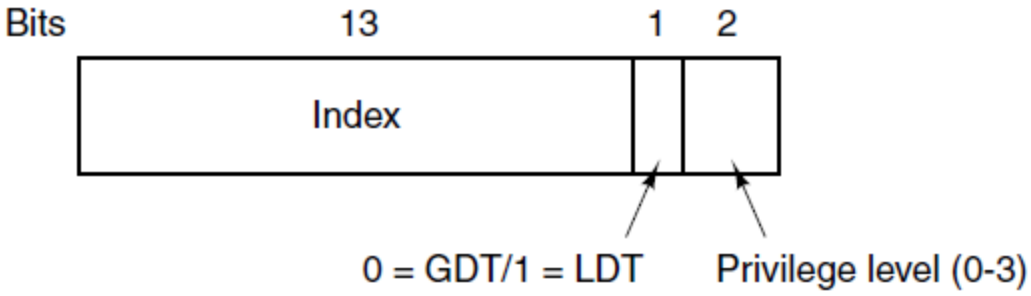
Comparison field: Segment number	Comparison field: Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

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

The Intel x86

- Virtual memory of x86 resembled that of MULTICS in many ways including segmentation and paging
  - 256K segments, each up to 64K words in MULTICS
  - 16K segments, each up to 1 billion words in x86
- In x86-64, segmentation is considered obsolete and is no longer supported
- In x86-32, segmentation is still used

Segmentation with Paging: The Intel x86 (1)



LDT (Local Descriptor Table) and GDT (Global Descriptor Table)

Each program has its own LDT

There is a single GDT shared by all the programs in the computer

Figure 3-38. An x86 selector.

Segmentation with Paging: The Intel x86 (2)

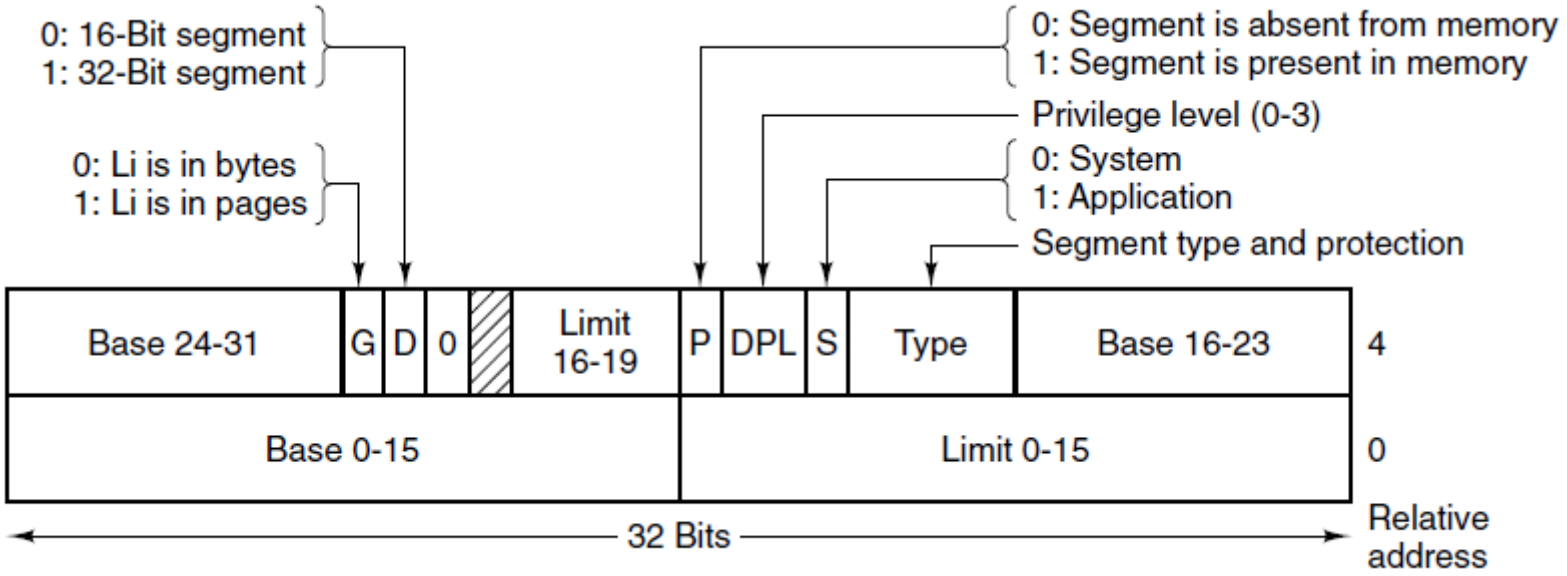


Figure 3-39. x86 code segment descriptor. Data segments differ slightly.

Segmentation with Paging: The Intel x86 (3)

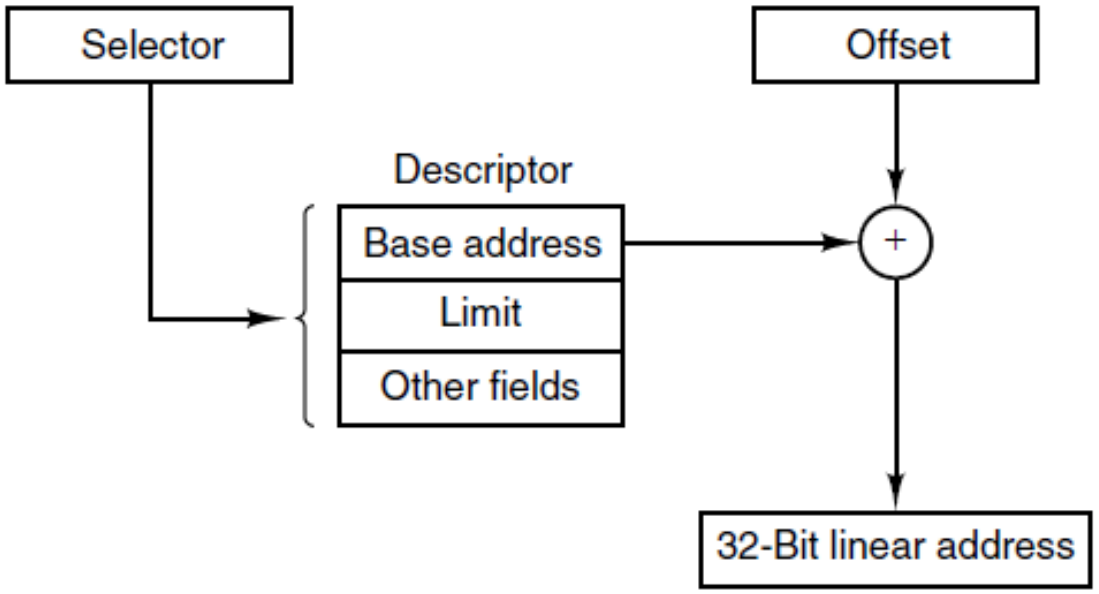


Figure 3-40. Conversion of a (selector, offset) pair to a linear address.

Segmentation with Paging: The Intel x86 (4)

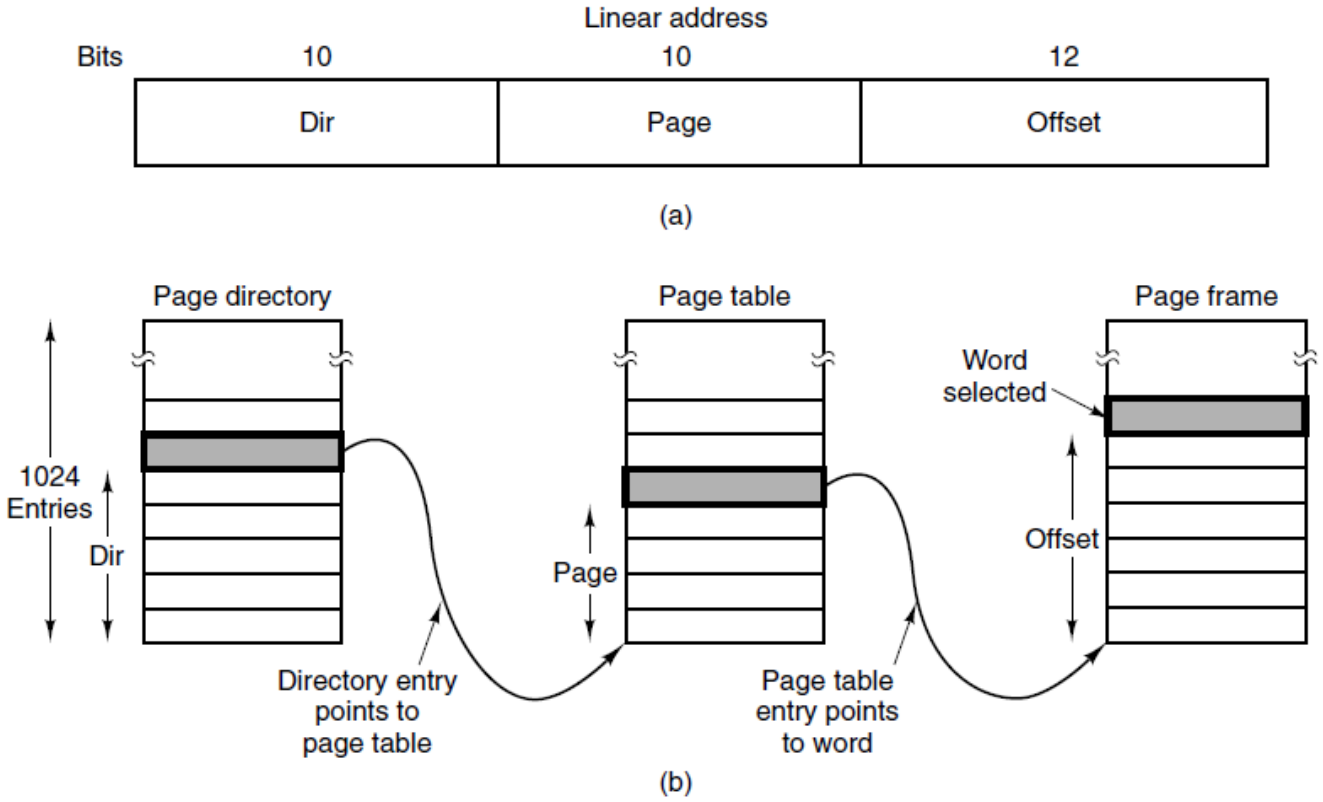


Figure 3-41. Mapping of a linear address onto a physical address.

End

Chapter 3