Chapter 3
Memory Management

- Basic memory management
- Swapping
- Virtual memory
- Page replacement algorithms
- Design issues for paging systems
- Segmentation

Memory Management

- Ideally programmers want memory that is
  - large
  - fast
  - non volatile
- Memory hierarchy
  - small amount of fast, expensive memory – cache
  - some medium-speed, medium price main memory
  - gigabytes of slow, cheap disk storage
- Memory manager handles the memory hierarchy

Memory Management

Memory manager keeps track of memory usage:

- allocate memory
- deallocate memory
- swapping

- Issues in memory management
  - Transparency:
    - want to let several processes coexist in main memory.
    - no process should need to be aware of the fact that memory is shared.
    - each must run regardless of the number and/or location of processes.
  - Safety:
    - processes must not be able to corrupt each other.
  - Efficiency:
    - both CPU and memory should not be degraded badly by sharing.

No Memory Abstraction

Mono-programming:

- One user process + O.S. are in memory.
- When the user process is waiting for I/O, CPU is idle.

No Memory Abstraction

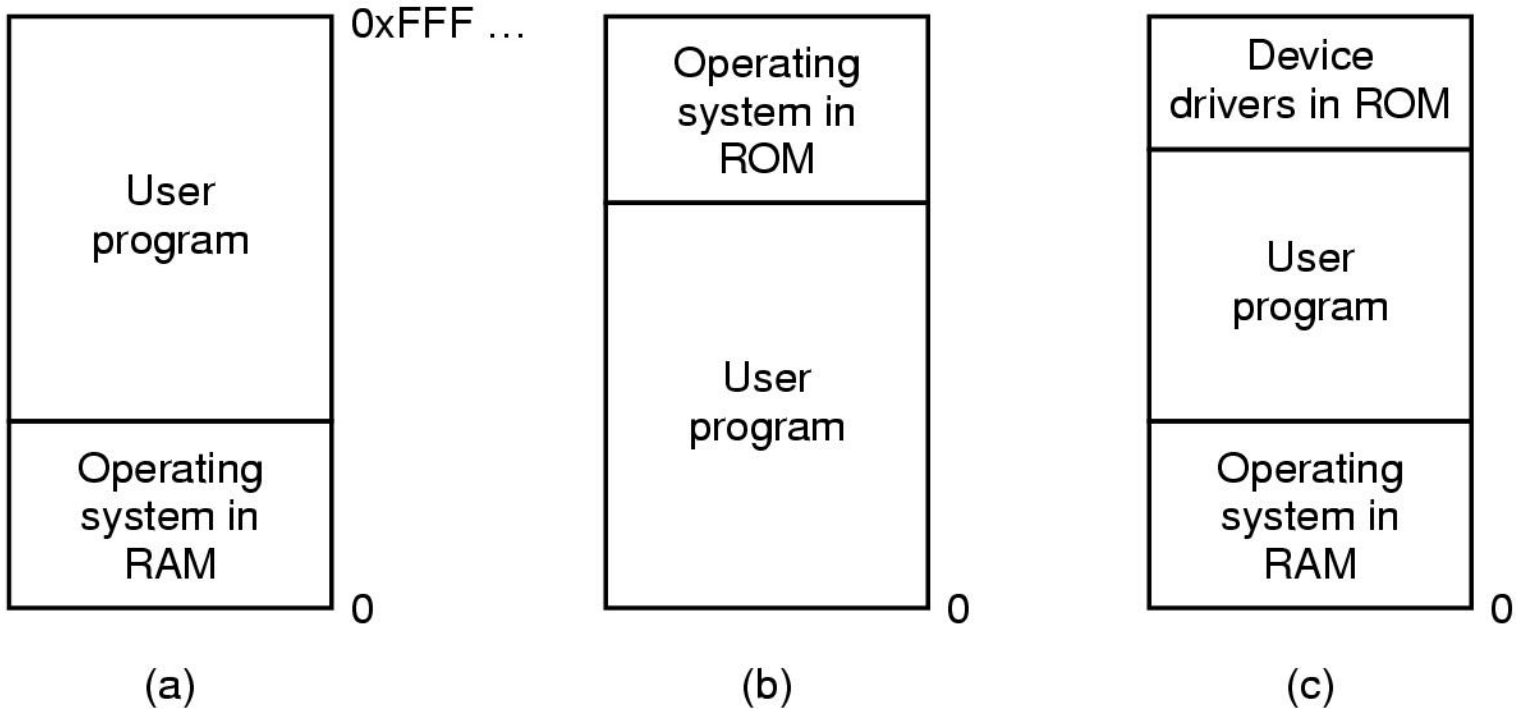


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process.

No Memory Abstraction

Multiprogramming:

- Several user processes are in memory.
- While one process is waiting for I/O, another process can use CPU. Increase CPU utilization.
- Design issue in multiprogramming:

  Decide how many processes should be in memory to keep CPU busy.

Modeling Multiprogramming

Probabilistic model for multiprogramming

n processes in memory

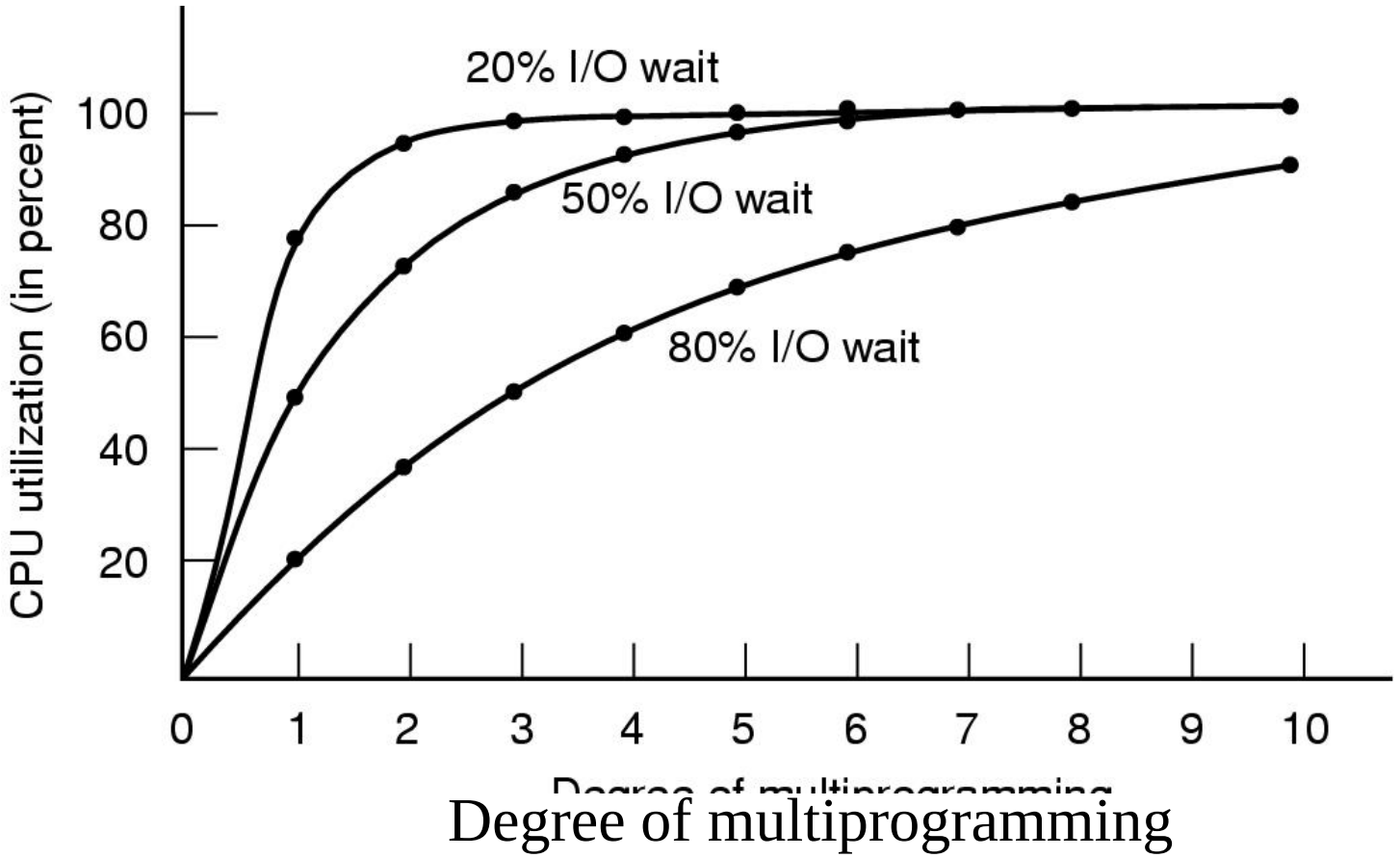
Each process spends a fraction p of its time in I/O wait state

Probability of all n processes are waiting for I/O: p^n
CPU utilization: $1 - p^n$.

n is called the **degree of multiprogramming**.

Approximation of the model

Modeling Multiprogramming



CPU utilization as a function of number of processes in memory

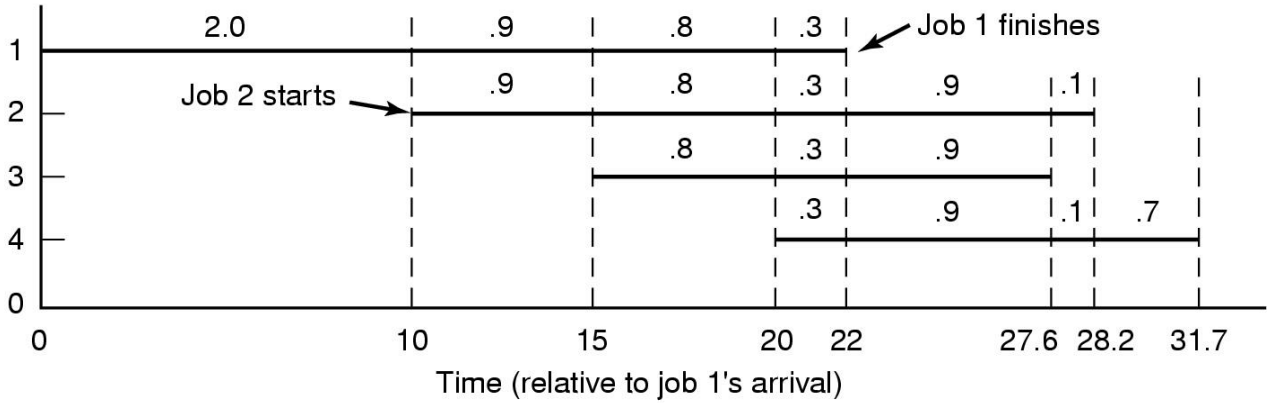
Analysis of Multiprogramming System Performance

Job	Arrival time	CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

(a)

	# Processes 1	2	3	4
CPU idle	.80	.64	.51	.41
CPU busy	.20	.36	.49	.59
CPU/process	.20	.18	.16	.15

(b)



(c)

- Arrival and work requirements of 4 jobs
- CPU utilization for 1 – 4 jobs with 80% I/O wait
- Sequence of events as jobs arrive and finish
- Note numbers show amount of CPU time jobs get in each interval

Analysis of Multiprogramming System Performance

Example: Use this model to analyze the performance of a multiprogramming system. Four jobs, p = 80%.

1. Only job 1 in memory.
20% of time CPU busy. 2 min CPU time finished within 10 min.

2. Jobs 1 and 2 in memory.
36% of time CPU busy. Using round robin, each process uses CPU time
5 x 0.36/2 = 0.9 min within 5 min.

3. Jobs 1, 2 and 3 in memory.
CPU busy = 0.49. Each process uses CPU 5 x 0.49/3 = 0.8 min.
...
Total time: **31.7 min** for all four jobs.

If no multiprogramming, run four jobs one by one.
The total time: (4+3+2+2)/0.2 = **55 min.**

Implementation of Multiprogramming

Fixed partitions without swapping

Divide memory into n partitions (may be unequal)
- Multiple input queues
- Single input queue

Relocation and protection

- Relocation: Adjust a program to run in a different location of the memory

- Protection: Prevent one job from accessing the memory location of another job

Relocation and Protection

Implementation:

Relocation only:
modify the instructions as the program is loaded into memory.

Protection only (e.g. IBM 360):
- Divide memory into 2K blocks
- Assign 4-bit protection code to each block
- PSW (Process State Word) contains a 4-bit key
- When key is not equal to protection code, hardware trap
- Only O.S. can change PSW key.

Relocation and Protection

Both relocation and protection

Two special hardware registers: base register and limit register.
Base register: stores the start address of a partition
Limit register: stores the length of a partition

When a process is scheduled, O.S. sends the start address of this process to base register and the partition length to limit register.

Physical memory address = (base) + memory adders generated by program

O.S. checks the physical address against the limit register

After starting execution, still relocatable.

Relocation

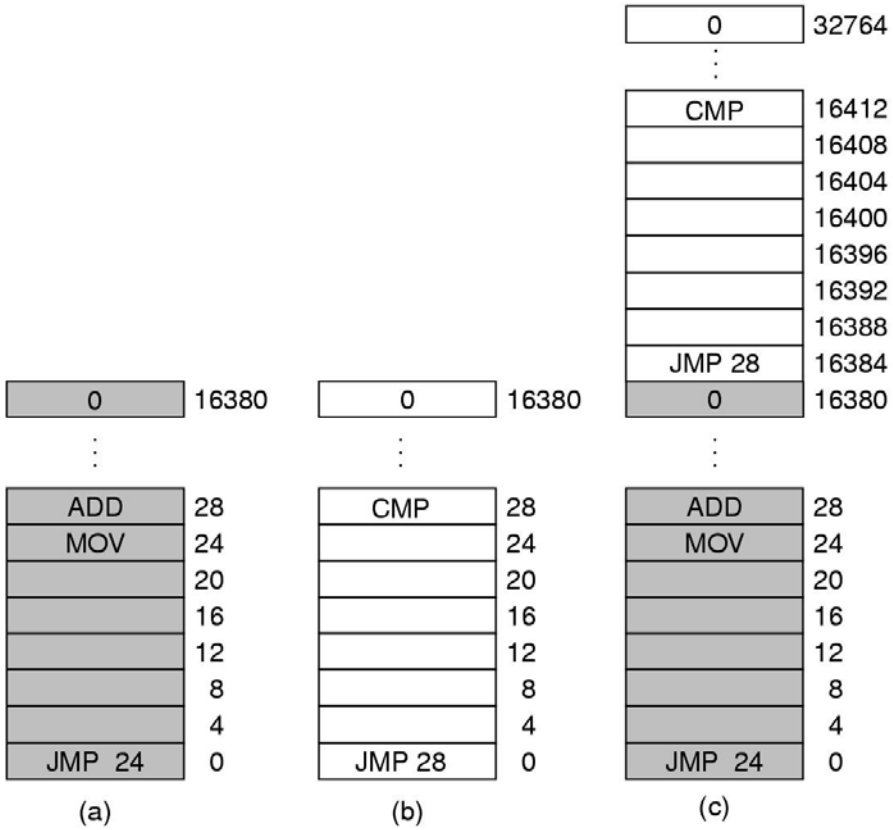


Figure 3-2. Illustration of the relocation problem.

Base and Limit Registers

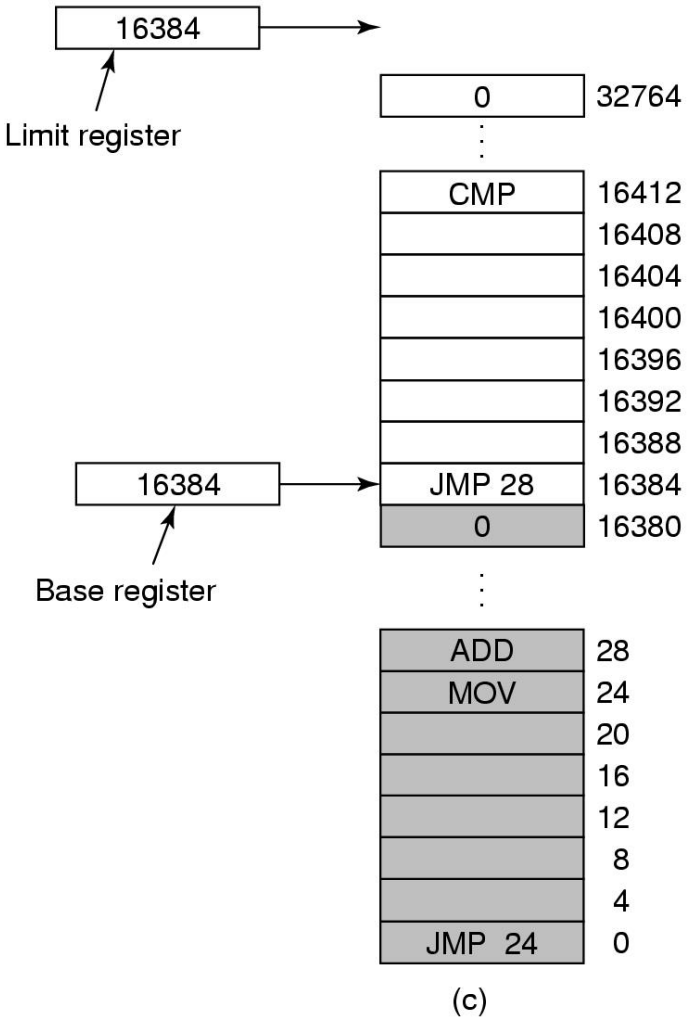
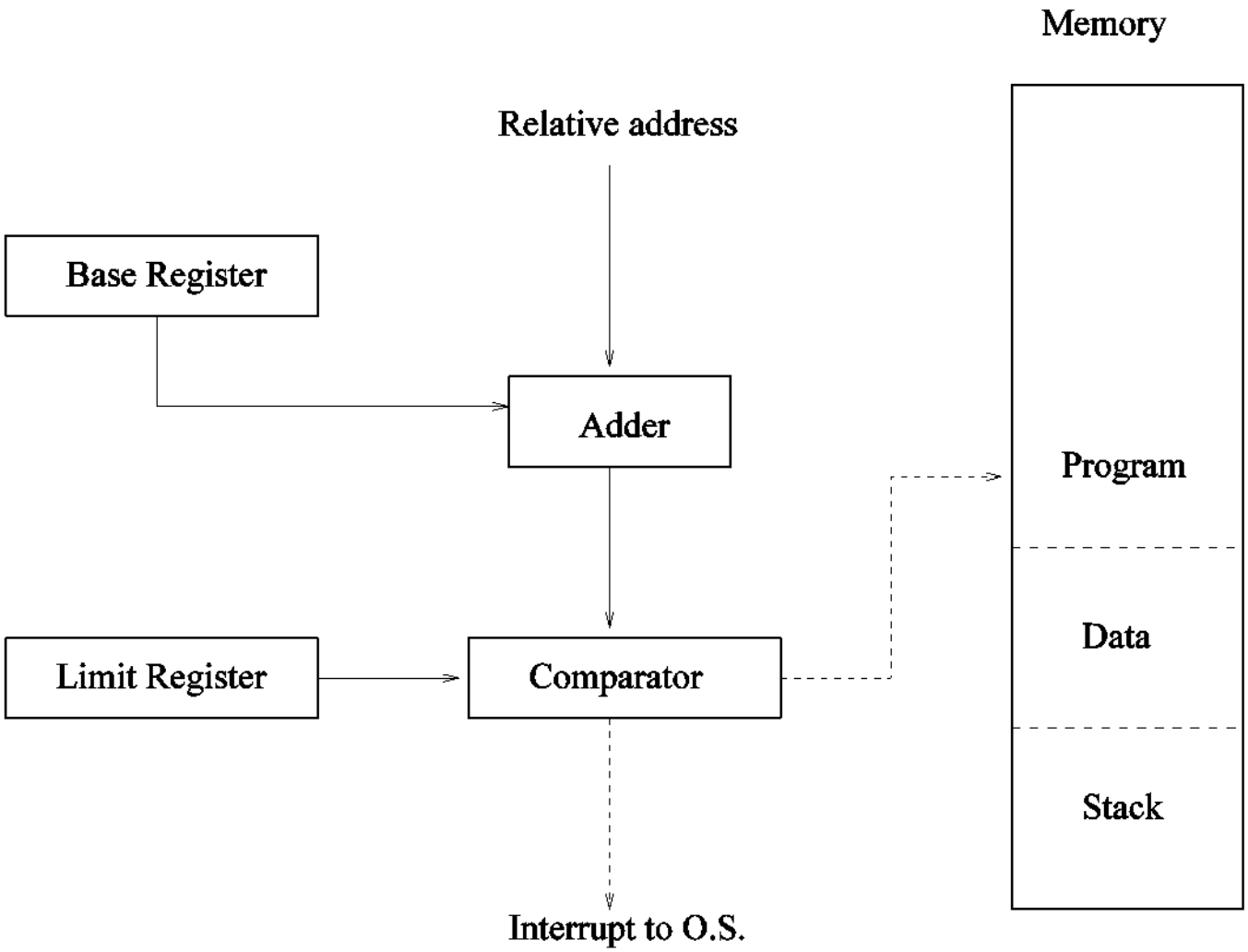


Figure 3-3. Base and limit registers can be used to give each process a separate address space.

Base and Limit Registers



Multiprogramming with Variable Partition

Different from fixed partition: the number, location and size of the partitions vary dynamically.

Concepts:

- Swapping: moving processes between memory and disk.

- Memory compaction: merge all processes together to get a big hole.

How to deal with processes with growing data segment: Allocate a little extra memory.

Process grows too large: kill or swap out.

Swapping

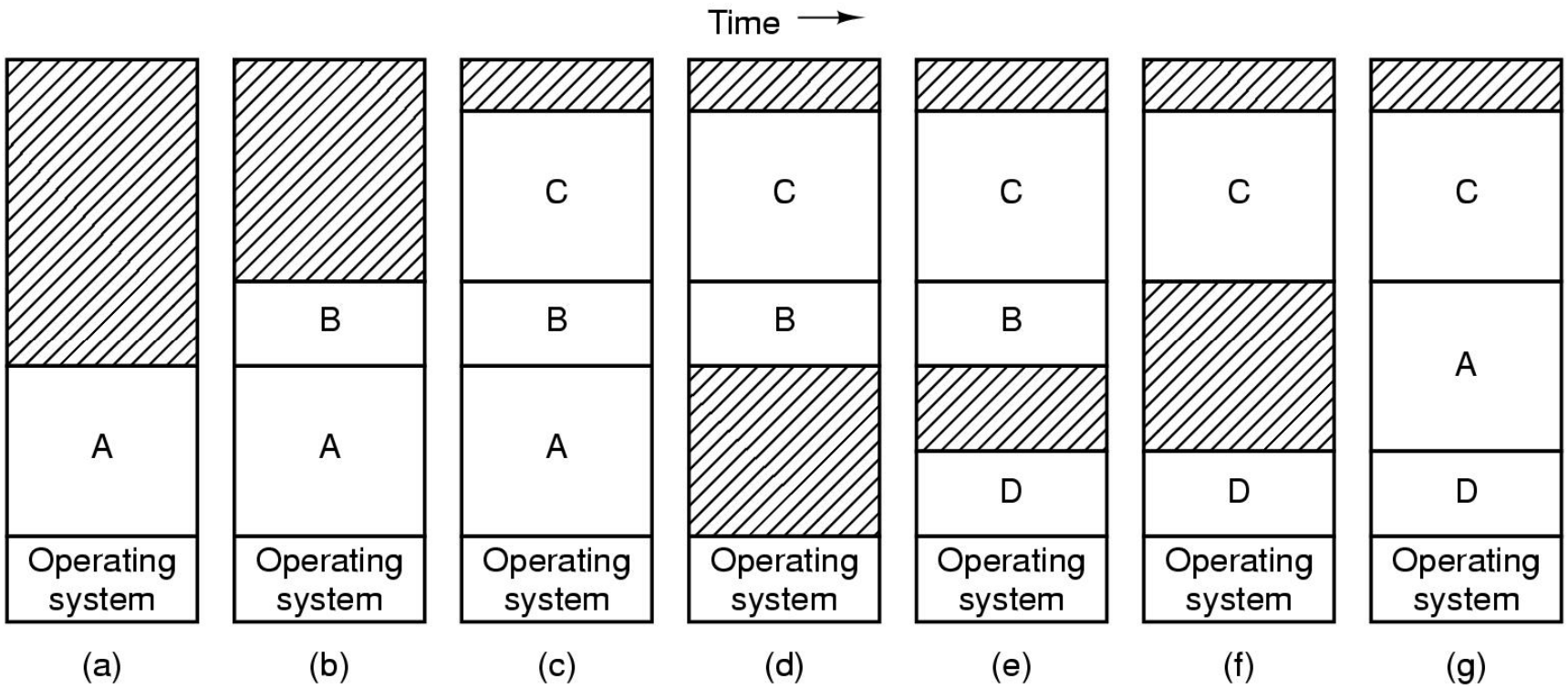


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

Swapping

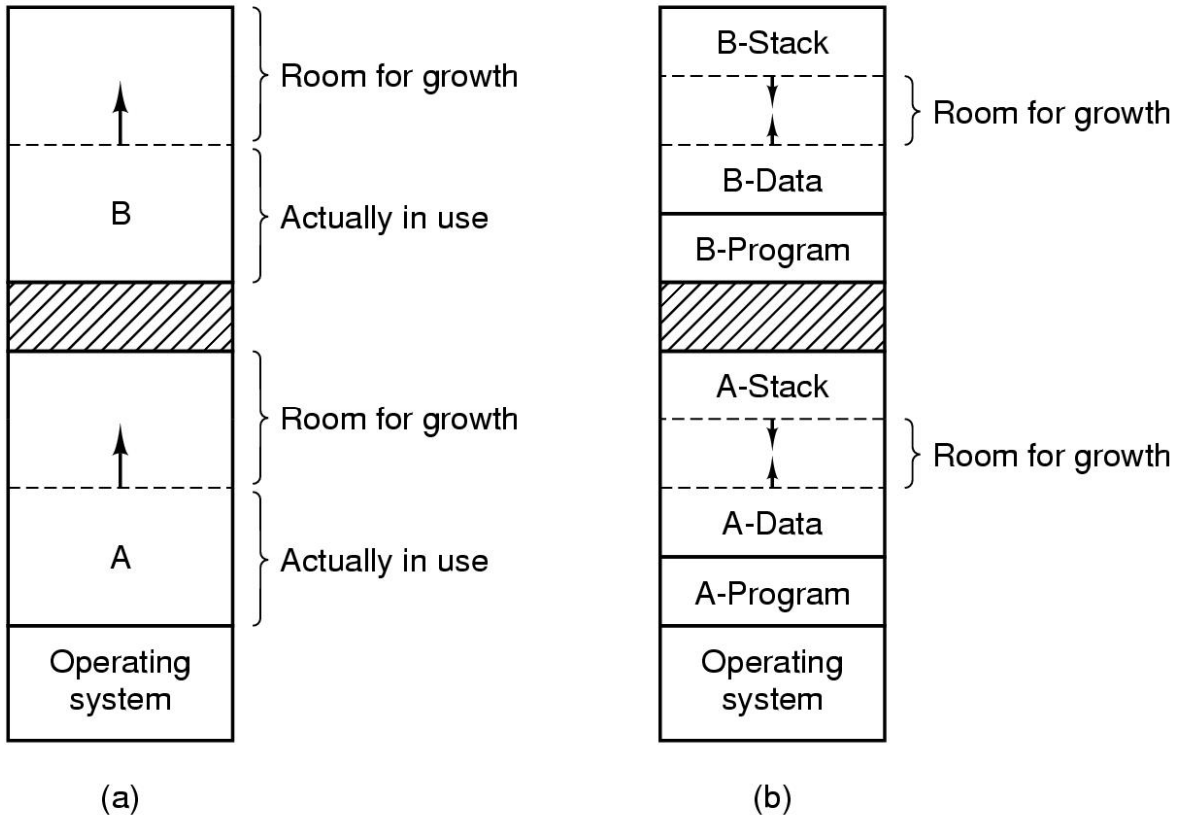


Figure 3-5. (a) Allocating space for growing data segment. (b) Allocating space for growing stack, growing data segment.

Keeping Track of Memory Usage

(1) Bit maps

Divide memory into fixed size chunks (allocation units).

Corresponding to each allocation unit is a bit in the bit map:

1: the unit is used
0: the unit is not used

Memory allocation:
Find k consecutive 0 bits in the map for a process that needs k allocation units of memory.

Major drawback: slow

Memory Management with Bitmaps

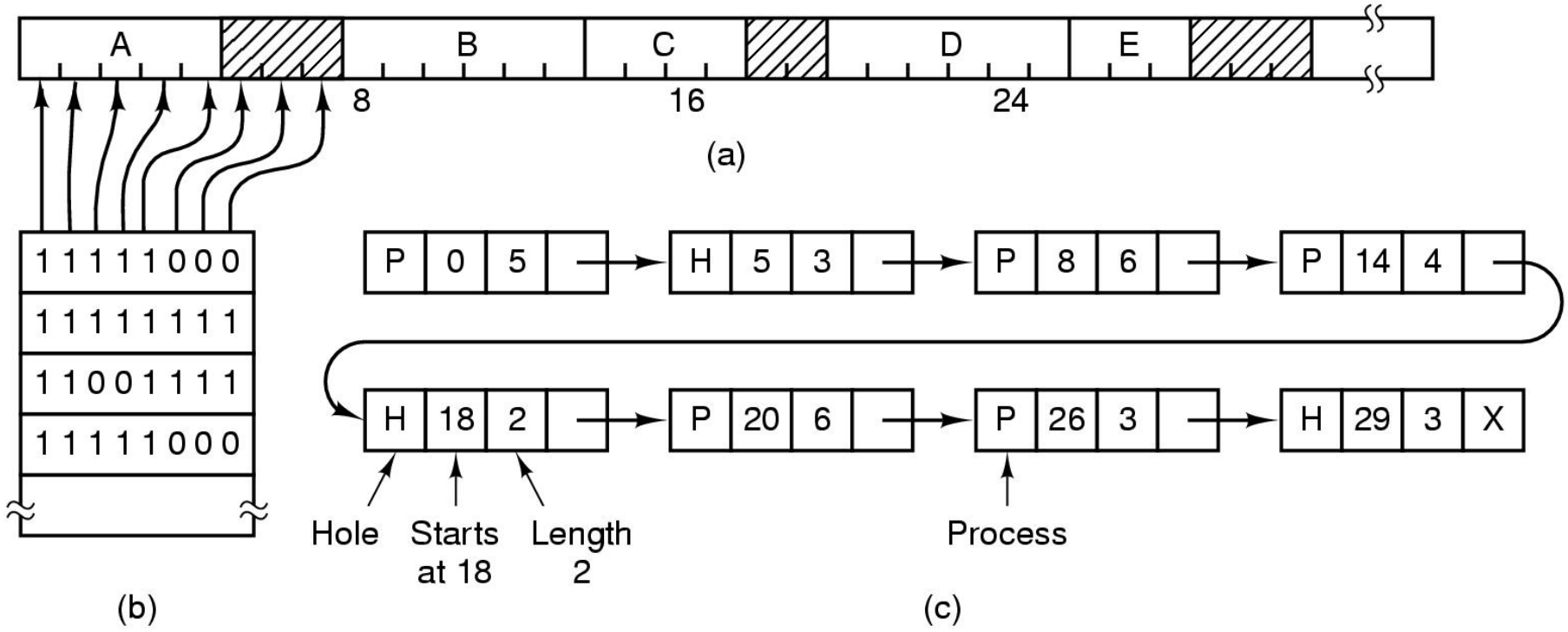
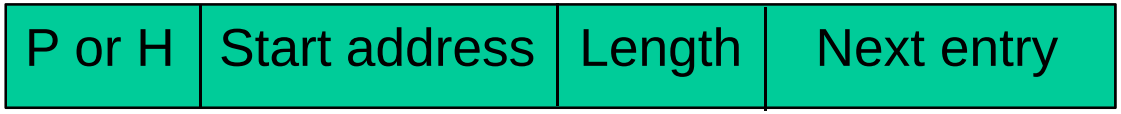


Figure 3-6. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

Memory Management with Linked Lists

(2) Linked lists

Data structure:



P: process
H: hole
Sorted by address

When a process terminates, four possibilities to merge

Memory Management with Linked Lists

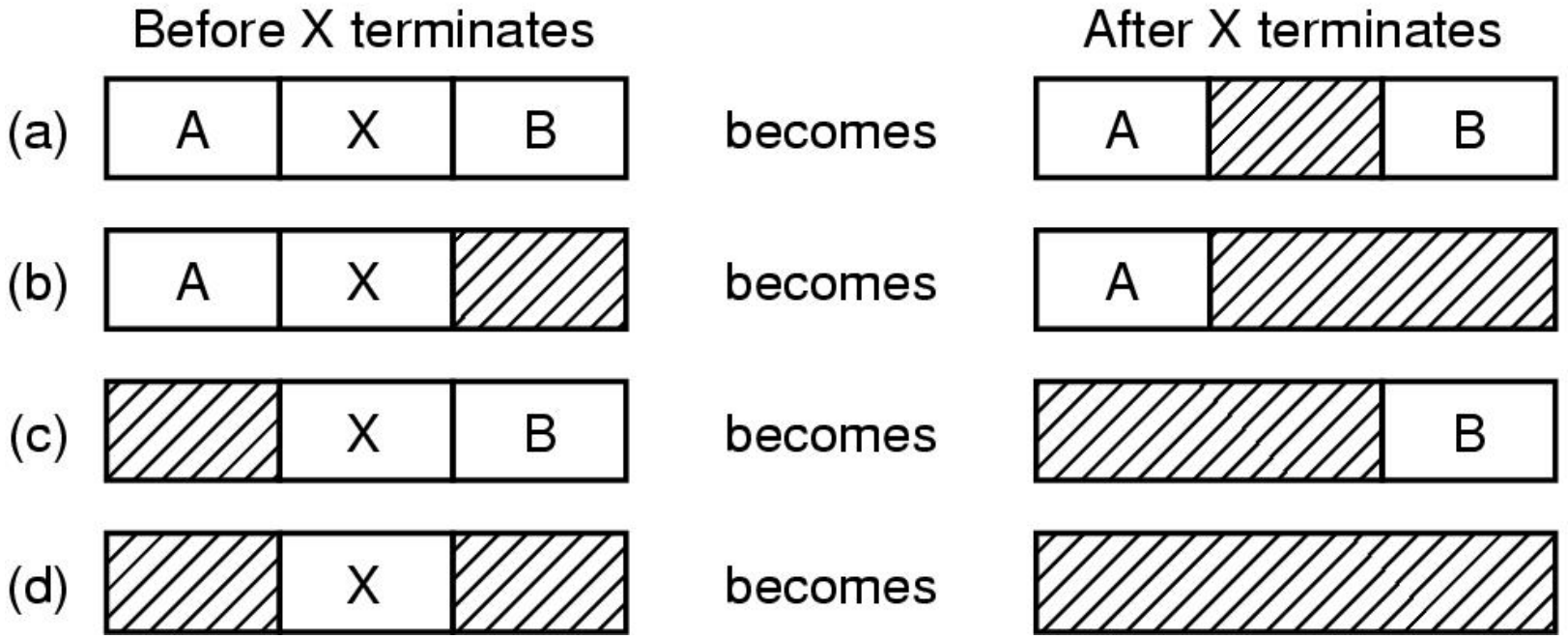


Figure 3-7. Four neighbor combinations for the terminating process, X.

Memory Allocation Algorithms

(a) First fit
Scan from the beginning and find the first hole that is big enough.

(b) Next fit
Similar to first fit except that search starts from where it left off.

(c) Best fit
Search the entire list and take the smallest hole that is adequate.

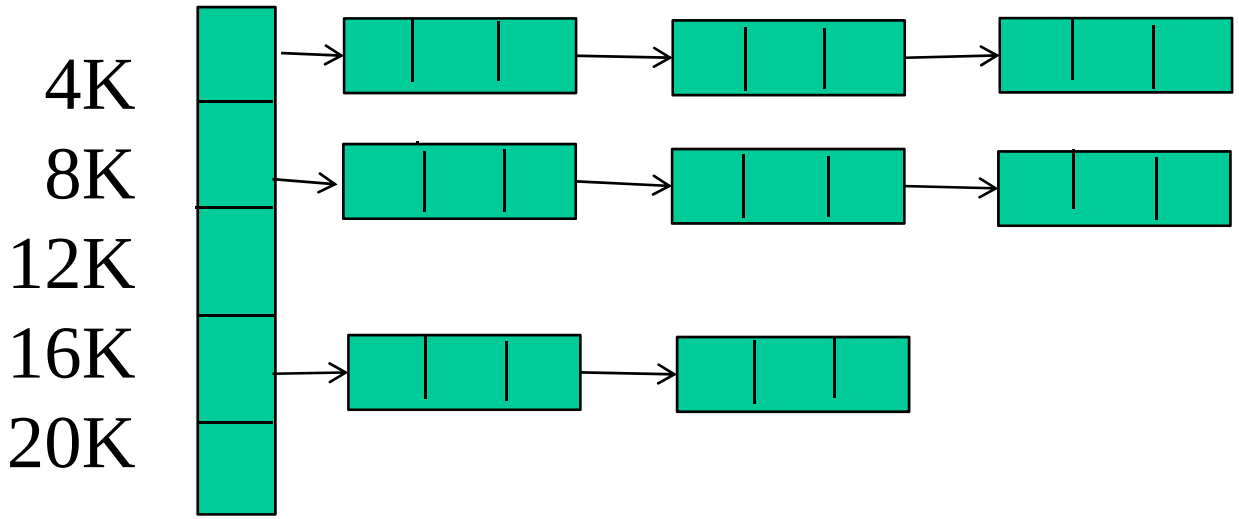
(d) Worst fit
Take the largest hole.

Memory Allocation Algorithms

(e) Quick fit

Maintain separate lists of holes for commonly used sizes. Sorted by hole size.

Finding a hole is fast. What about merging holes? May go through the entire list.



Memory Allocation Algorithms

(f) Buddy system

Maintain a list of free blocks of size 2^0, 2^1, 2^2, , …, 2^k, …, bytes, up to the size of memory.

Example: 1 MB memory, 21 lists needed from 1 byte to 1 MB. Process A requests 70K, need 128K hole. 1MB is split into two 512K blocks (buddies).

Allocation: If the needed size is available, done.
If not, look at the next large size (double size). If available, split it into two blocks and use one.
If not available, look at next large size . . . .
At most log N steps for N bytes memory.

Memory Allocation Algorithms

Deallocation:

First search the same size queue to see if a merge is possible. If not, done.

Otherwise merge them and then search next large size queue until no merge possible.
At most log N steps.

Drawback:
Low memory utilization.
Large internal fragmentation (wasted memory internal to the allocated segments).

External fragmentation: wasted holes between segments.

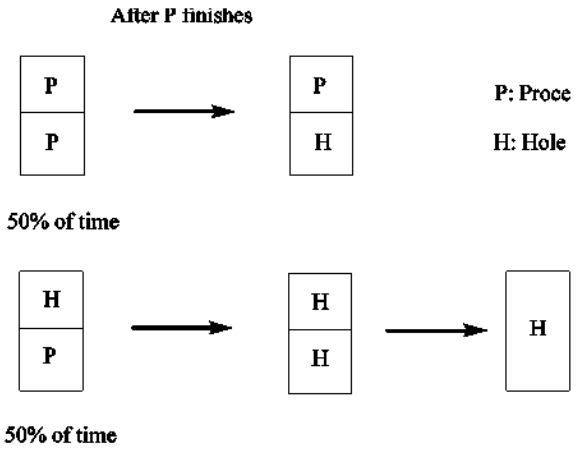
Analysis of Swapping Systems

Estimate how much portion of memory is wasted as holes at any time.

Consider an average process.

Holes can be merged but processes cannot.

Fifty percent rule: At steady state, if there are n processes then there are n/2 holes.



Analysis of Swapping Systems

Unused memory rule:

f: faction of memory occupied by holes
s: average size of a process
ks: average size of a hole
m: total memory bytes

$$(n/2)ks = m - ns$$

$$m = ns(1 + k/2)$$

$$f = (n/2)ks/m = (k/2)m/(1 + k/2)/m = k/(k+2)$$

$$k = hole\_size / process\_size$$

Examples:
k = ½: f= 20%
k = ¼: f = 11%
k = 2: f = 50%

Virtual Memory

•Main idea: allow users to have a large logical address space without worrying about the small physical memory.

•Goal: try to achieve that access to disk is almost as fast as access to memory.

•How to do it: when the size of program exceeds the size of physical memory, O.S. keeps part of the program currently in use in memory, and the rest on disk, with pieces of the program swapped between disk and memory as needed.

•Concepts:

- Virtual address: program generated address
- Physical address: real memory address
- Without virtual memory: virtual address = physical address

Virtual Memory – Paging System

Paging virtual memory (linear address space)

Memory management unit (MMU):
Maps logical address to physical address with a page table

- Page size is fixed, and must be power of 2

Example:
16 bit virtual address (64K), 32K physical memory
Divide virtual address space into 4K pages
Divide physical address space into the same size page frames

Virtual Memory – Paging

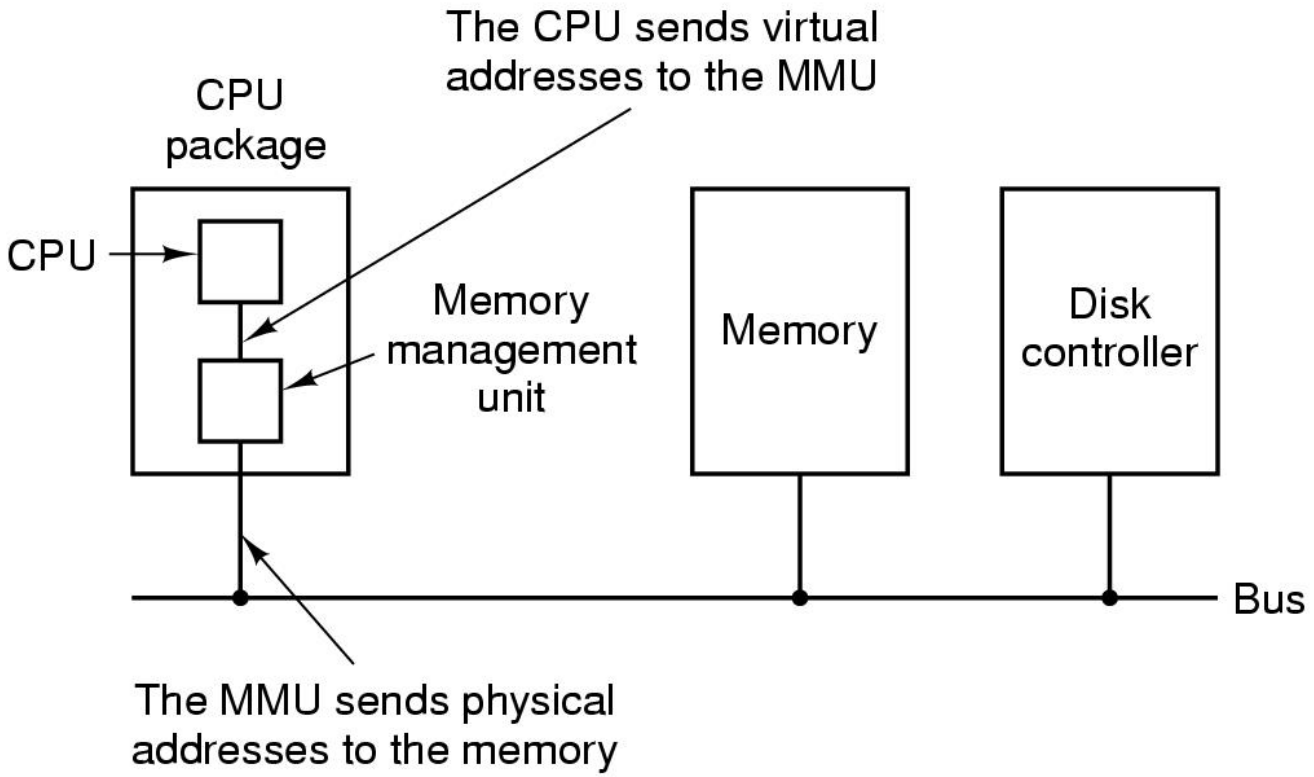


Figure 3-8. The position and function of the MMU – shown as being a part of the CPU chip (it commonly is nowadays). Logically it could be a separate chip, was in years gone by.

Paging

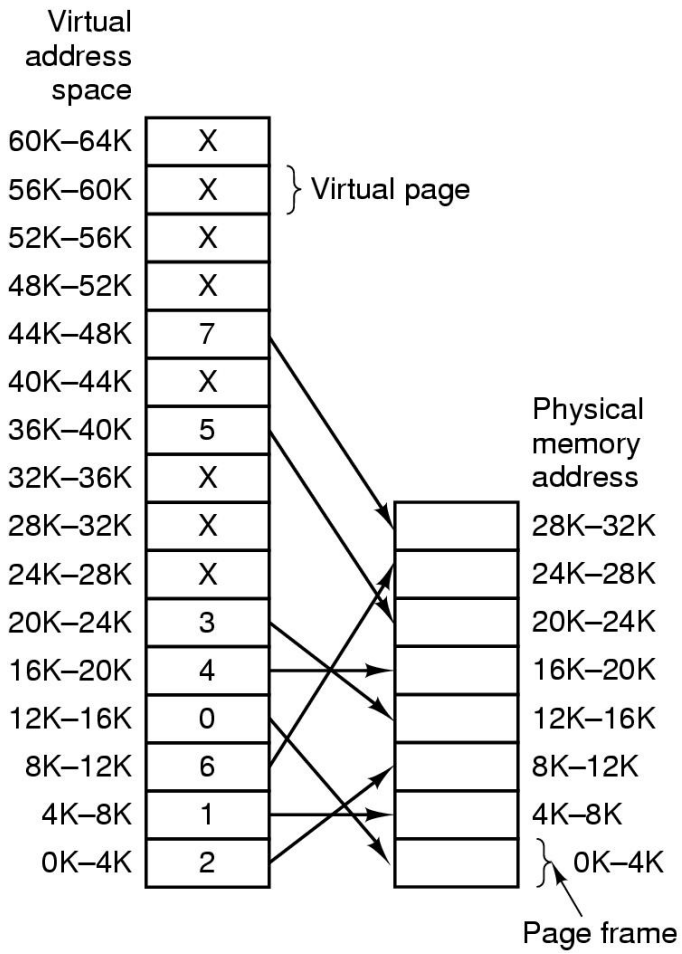


Figure 3-9. Relation between virtual addresses and physical memory addresses given by page table.

Paging

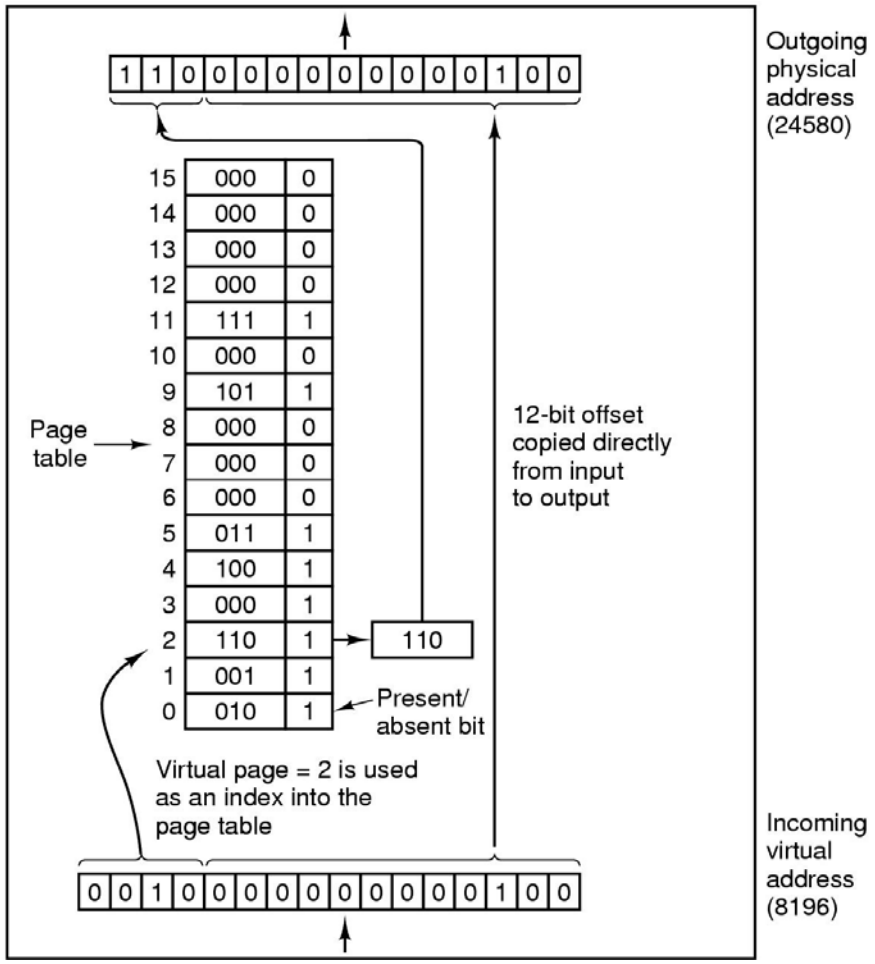


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

Structure of Page Table Entry

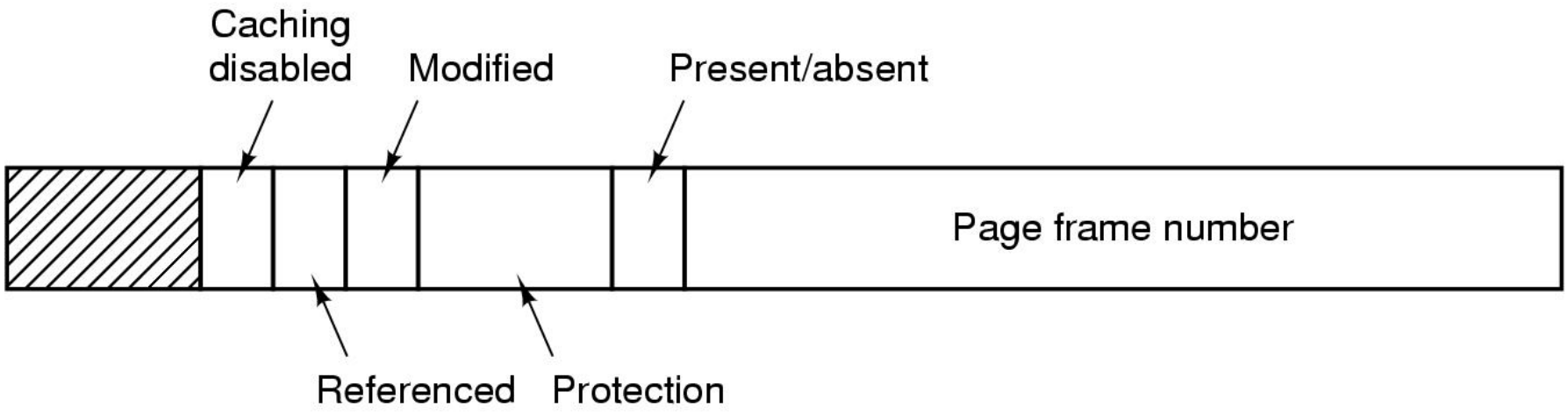


Figure 3-11. A typical page table entry.

Paging

Problems with paging

Efficiency of access:
Even small page tables are generally too large to store in fast memory. So page table is kept in main memory.
One memory access requires two real memory accesses.

Table space: smaller pages, larger page table

Internal fragmentation: larger pages, more internal fragmentation.

General page size: 512-8K bytes

Speeding Up Paging

Paging implementation issues:

- The mapping from virtual address to physical address must be fast.
- If the virtual address space is large, the page table will be large.
- Use translation lookaside buffer to store part of page table

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.

Page Replacement Algorithms

Page fault: the page to be accessed is not in memory (a trap to CPU).

What O.S. to do when a page fault occurs

- Bring a page into memory
- Update page table
- Continue execution of the process

What to do when memory is full:

- Remove one page already in memory.

Page replacement:

When a page fault occurs, O.S. removes a page from memory to make a room for the page that has to be brought in.

Page Replacement Algorithms

- Page fault forces choice
  - which page must be removed
  - make room for incoming page
- Modified page must first be saved
  - unmodified just overwritten
- Better not to choose an often used page
  - will probably need to be brought back in soon

Why virtual memory can be successful: because of **locality of reference**. The memory reference tends to be local. Access the nearby locations, not crossover the entire memory space.

Page Replacement Algorithms

- Random algorithm (pick up any page at random)
- Optimal page replacement algorithm
- Not recently used page replacement
- First-In, First-Out page replacement
- Second chance page replacement
- Clock page replacement
- Least recently used page replacement

Optimal Page Replacement Algorithm

- Replace the page needed at the farthest point in future (Select for replacement that page for which the time to the next reference is the longest)
  - Optimal but unrealizable
  - Results in the fewest number of page faults
  - Impossible to implement
  - Good for comparison
- Estimate by …
  - logging page use on previous runs of process, although this is impractical

Not Recently Used Page Replacement Algorithm

Based on locality of reference. Use the past to predict the future.

Need hardware support:

Two bits associated with each page:
R: reference bit. Set by hardware on any memory read/write to the page.
M: modified bit. Set by hardware when a page is written.

O.S. can reset these two bits in software.

Not Recently Used Page Replacement Algorithm

Algorithm:
(1) When a process is started, R and M bits for all its pages are set to 0.
(2) On each clock tick interrupt, clear the R bit.
(3) When a page fault occurs, choose a page from the lowest numbered nonempty class:

Class	R	M	
0	0	0	not referenced, not modified
1	0	1	not referenced, modified
2	1	0	referenced, not modified
3	1	1	referenced, modified

Easy to implement. Performance is okay.
No history of how long the page is used.

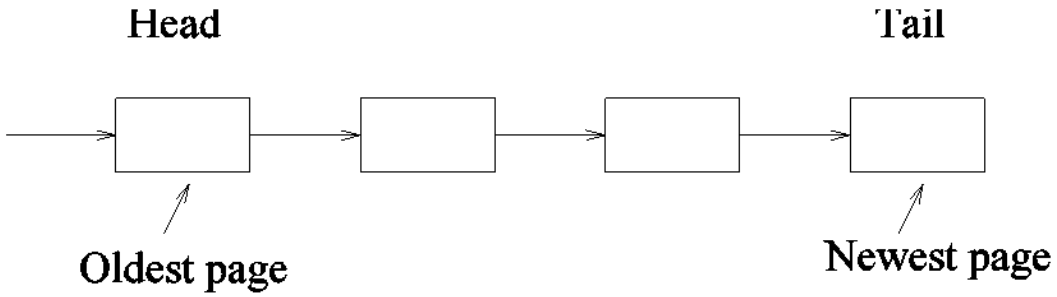
FIFO Page Replacement Algorithm

- Maintain a linked list of all pages in order they came into memory
- When page fault occurs, page at beginning of list replaced
- Disadvantage

  may remove a heavily used as a page in memory the longest may be often used.

  Solution:

  Inspect R and M bits of all pages. Remove the oldest page with R = 0.



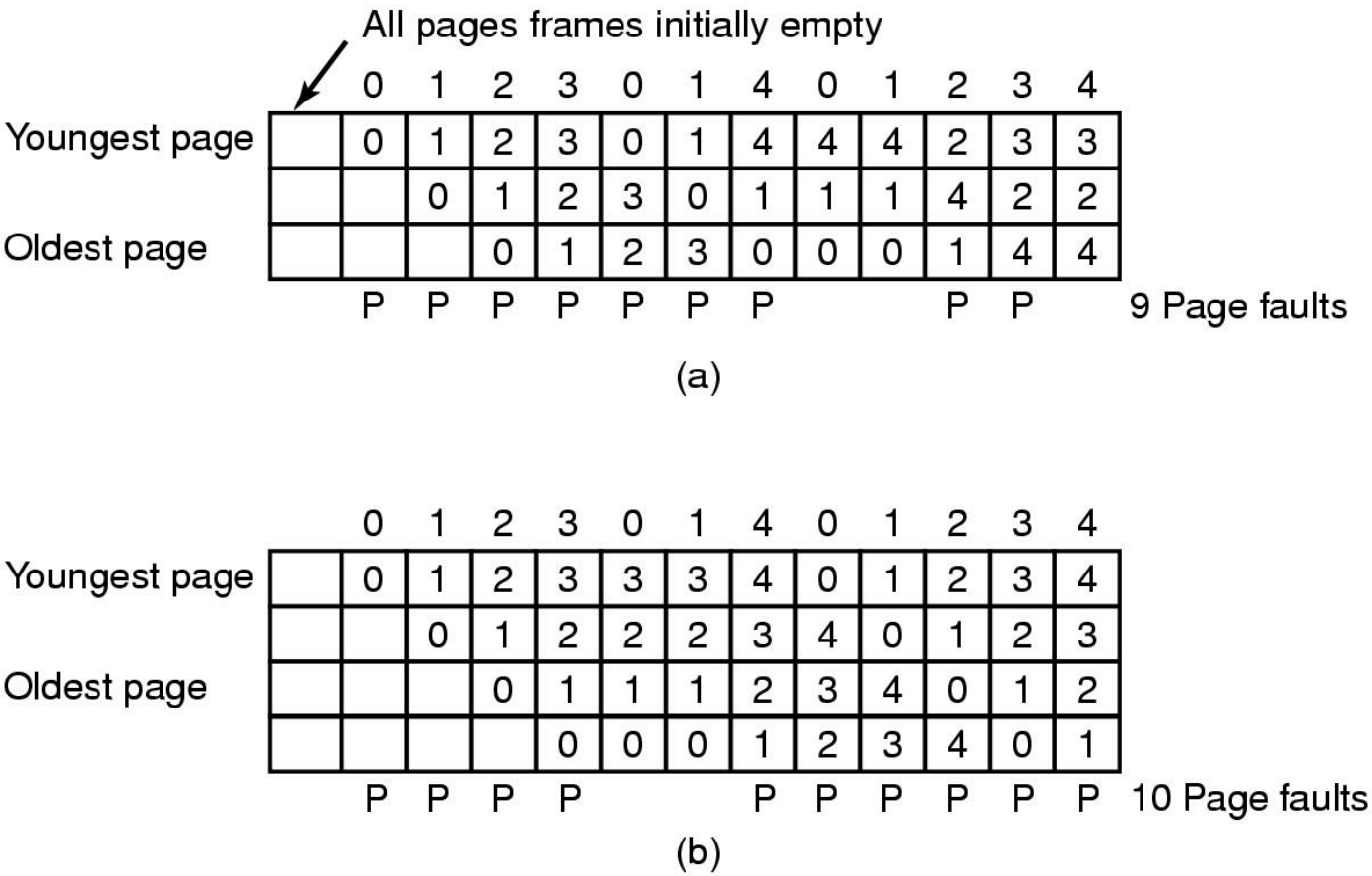
Two Variants of FIFO

Second chance
Look at the oldest page. if R = 0, replace it. Otherwise, clear the R bit and put the page at the end of the list.

Clock
Keep all the pages on a circular list.

An interesting thing in FIFO (Belady's Anomaly): More page frames may lead to more page faults.

Belady's Anomaly



- FIFO with 3 page frames
- FIFO with 4 page frames

- *P*'s show which page references show page faults

Second Chance Algorithm

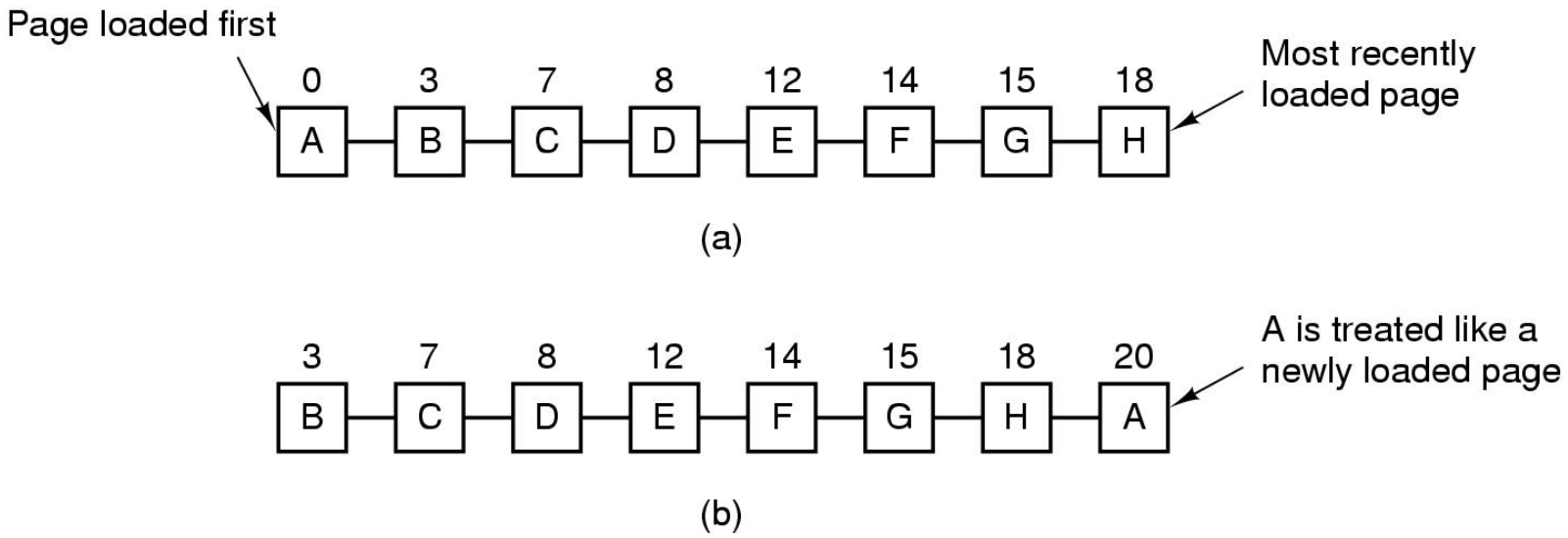
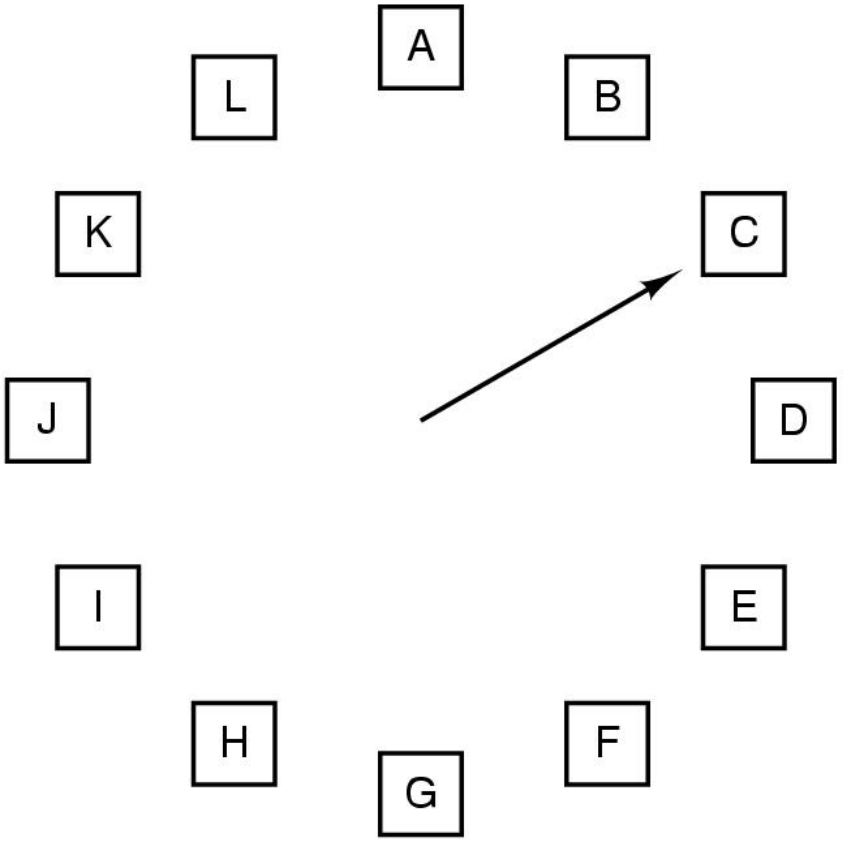


Figure 3-15. Operation of second chance.
(a) Pages sorted in FIFO order.
(b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their load times.

The 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

Least Recently Used (LRU)

- Assume pages used recently will be used again soon
  - throw out page that has been unused for longest time
- Must keep a linked list of pages
  - most recently used at front, least at rear
  - update this list every memory reference !!
- Alternatively keep counter in each page table entry
  - choose page with lowest value counter
  - periodically zero the counter

Least Recently Used (LRU)

Replace the page in memory that has not been referenced for the longest time.

Performance close to optimal algorithm

Implementation is not cheap. Either requires special hardware or approximate software simulation.

Hardware implementation I (Counter)

- A 64-bit counter, C, automatically incremented after each instruction
- After each memory reference, the current value of C is stored in page table for the page just referenced
- Lowest number in the page table is the least recently used

Least Recently Used (LRU)

Hardware implementation II (Matrix)

- For n page frames, use an n x n bit matrix, initially all 0
- After page frame k is referenced, the hardware first sets row k to 1 and then sets column k to 0
- The row whose binary value is lowest is the least recently used

LRU Page Replacement Algorithm

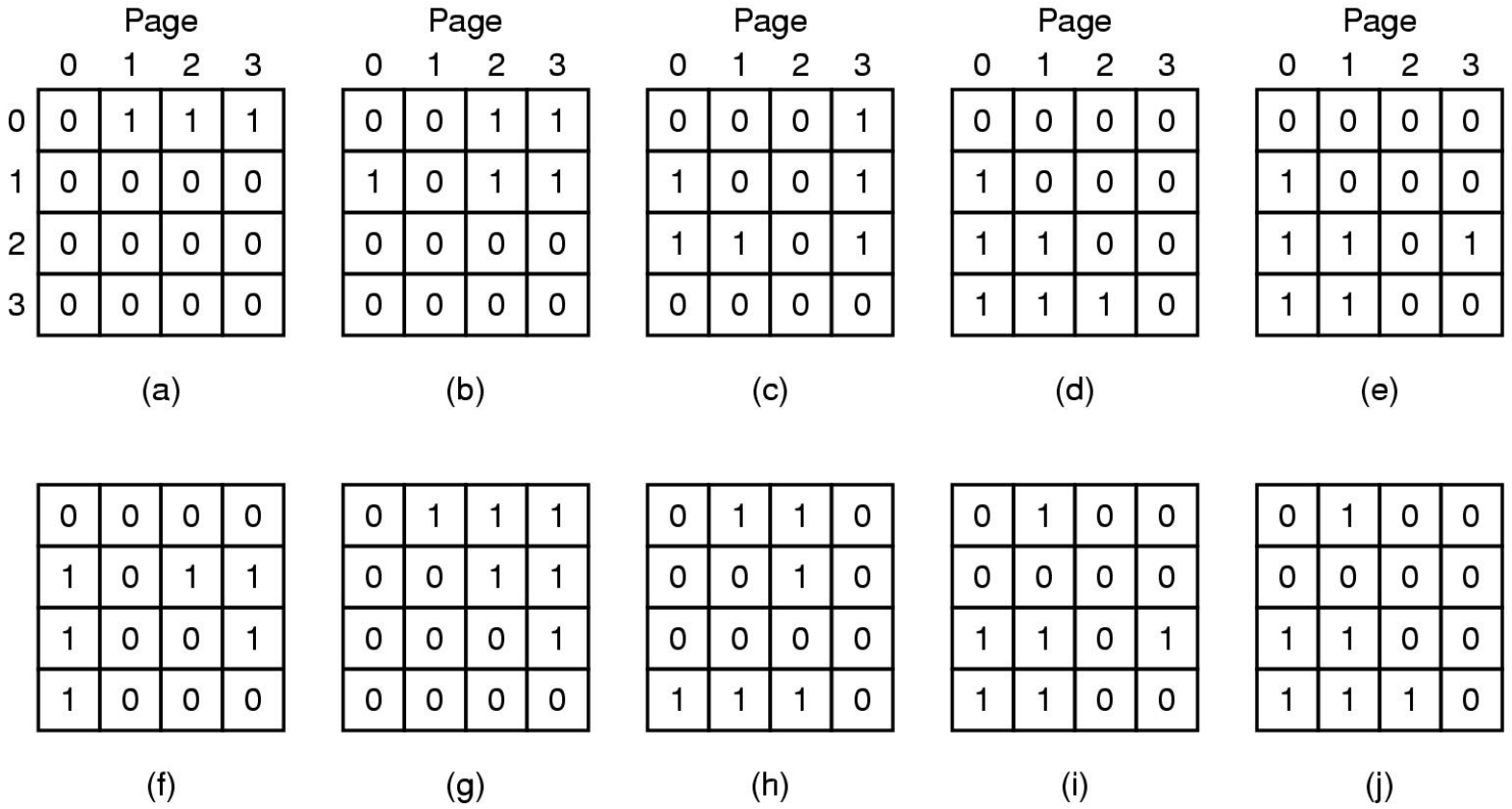


Figure 3-17. LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

LRU Page Replacement Algorithm

Software approximation I (Not-Frequently-Used, NFU)

- Associate with each page a software counter, initially 0
- At each clock tick interrupt, O.S. adds each page's R bit to the counter associated with that page
- When a page fault occurs, replace the page with the lowest counter
- Problem: never forgets anything

LRU Page Replacement Algorithm

Software approximation II (Aging Algorithm)

Modify NFU as follows:

- Counters are shifted right 1 bit before R bits are added
- R bit is added to the leftmost
- Remove the lowest counter page
- Different from LRU:
  Cannot distinguish the references within the same clock tick
- Counters have a finite number of bits, say, 8 bits, and can remember only the history of 8 clock ticks.

Simulating LRU in Software

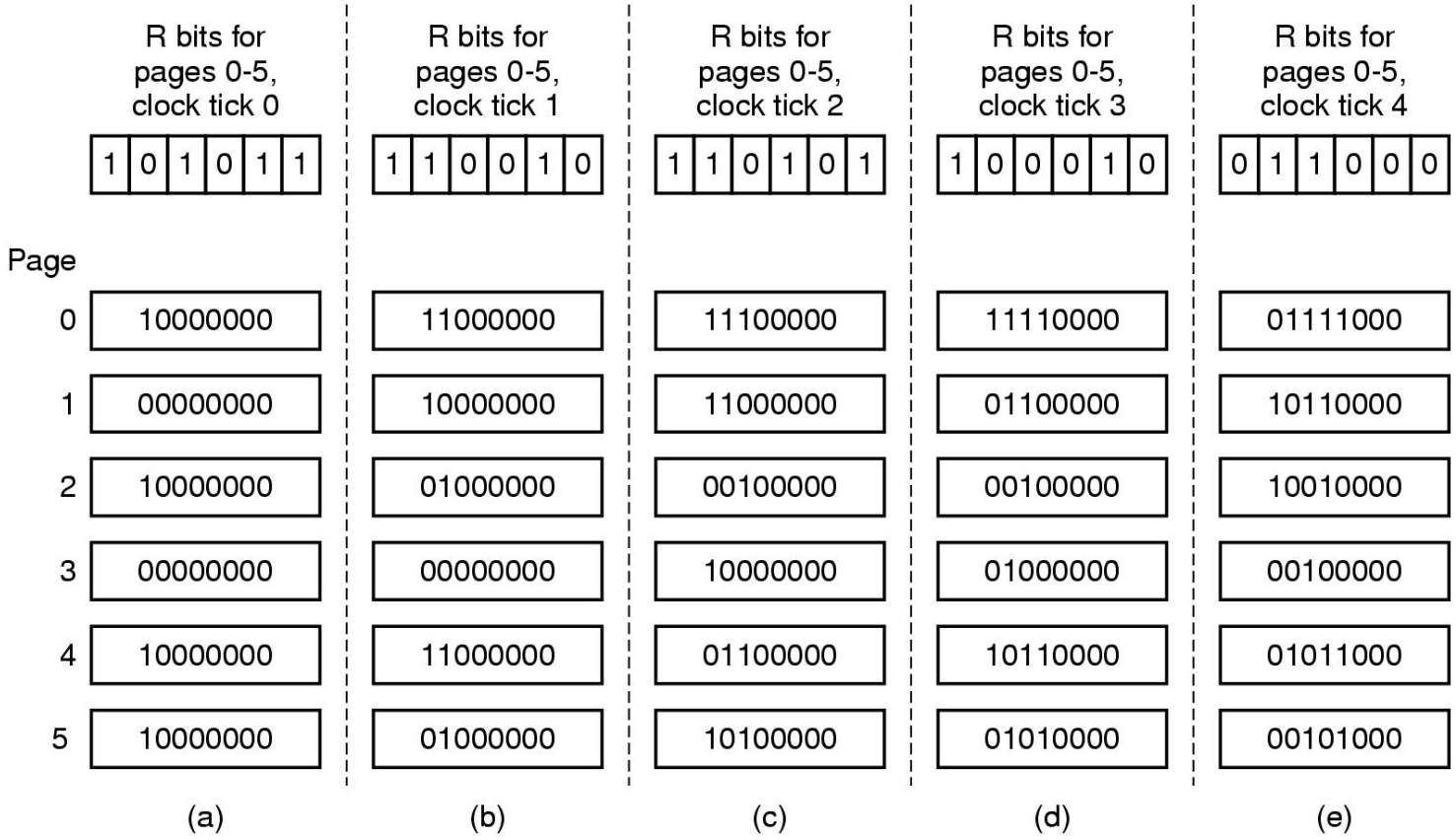


Figure 3-18. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

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 3-22. Page replacement algorithms discussed in the text.

Design Issues for Paging System

Aiming at good performance.

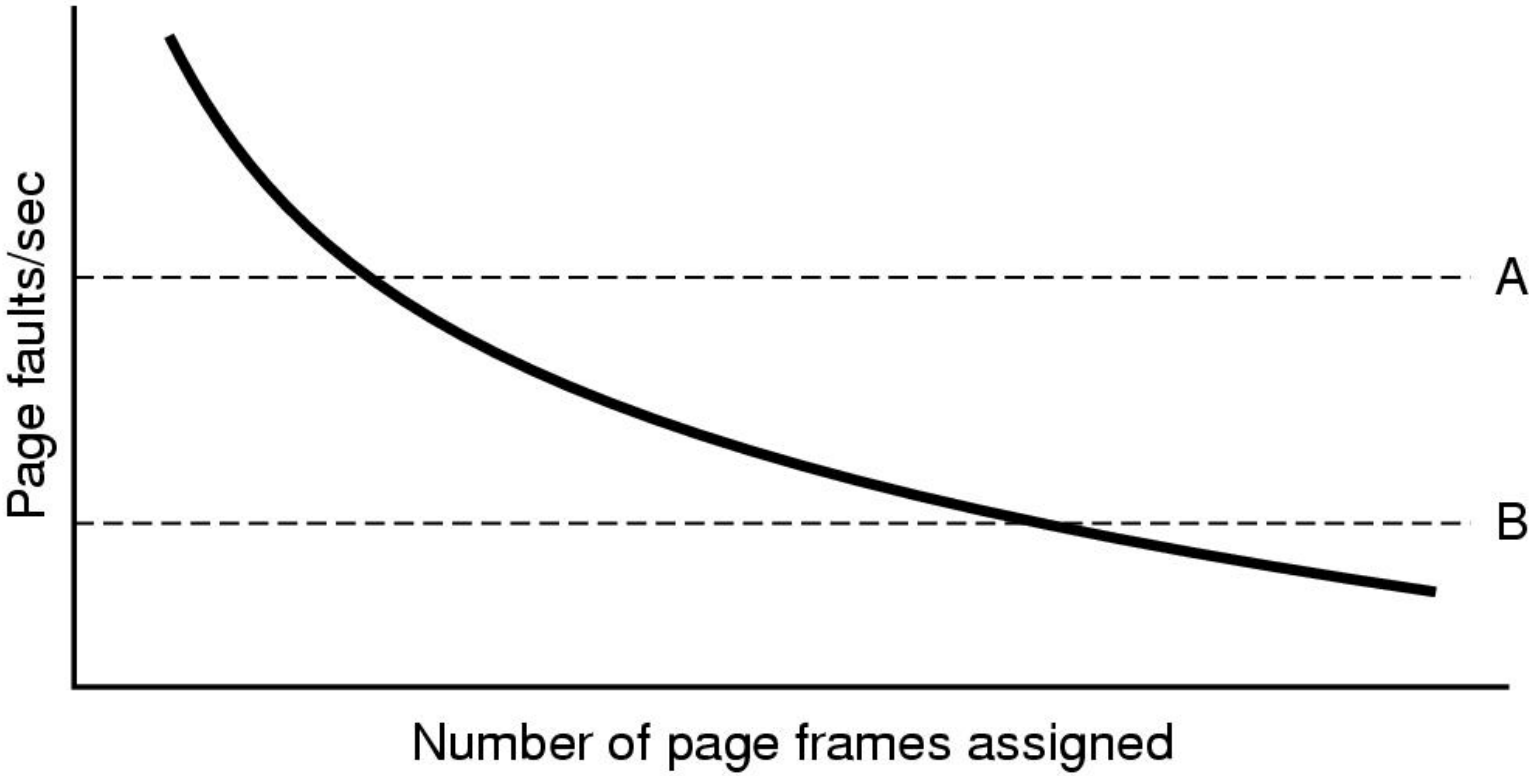
Working set: the set of pages a process is currently using

Thrashing: a program causes page fault every few instructions

How to control thrashing:
- Keep the entire working set in memory
- Page fault frequency allocation algorithm (PFF)

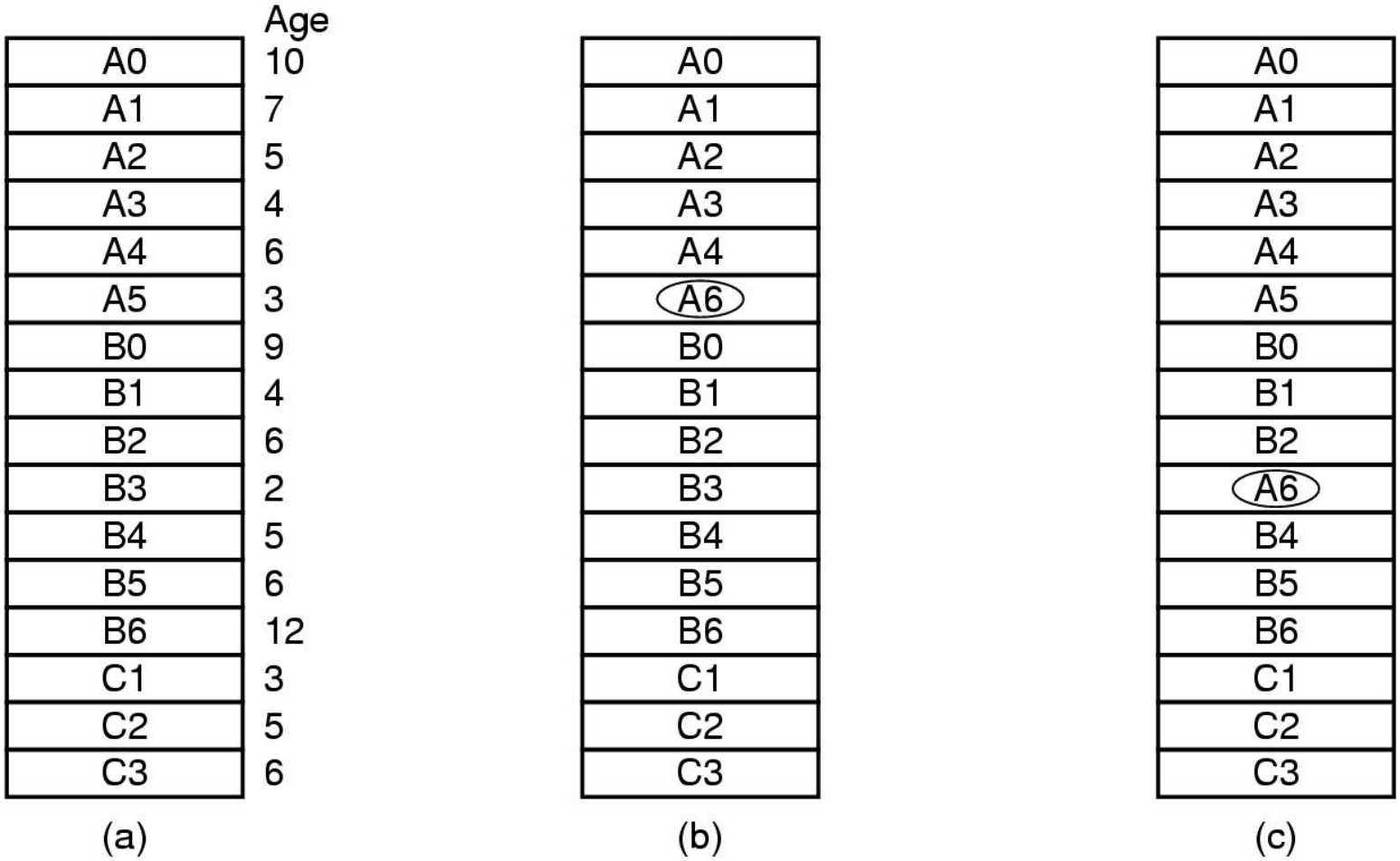
In most page replacement algorithms, we can choose a certain number of page frames for each process to keep a certain page fault rate. Dynamically allocate page frames to each process.

Page Fault Rate vs. Page Frames



Page fault rate as a function of the number of page frames assigned

Local versus Global Allocation Policies



(a) Original configuration. (b) Local page replacement. (c) Global page replacement.

Separate Instruction and Data Spaces

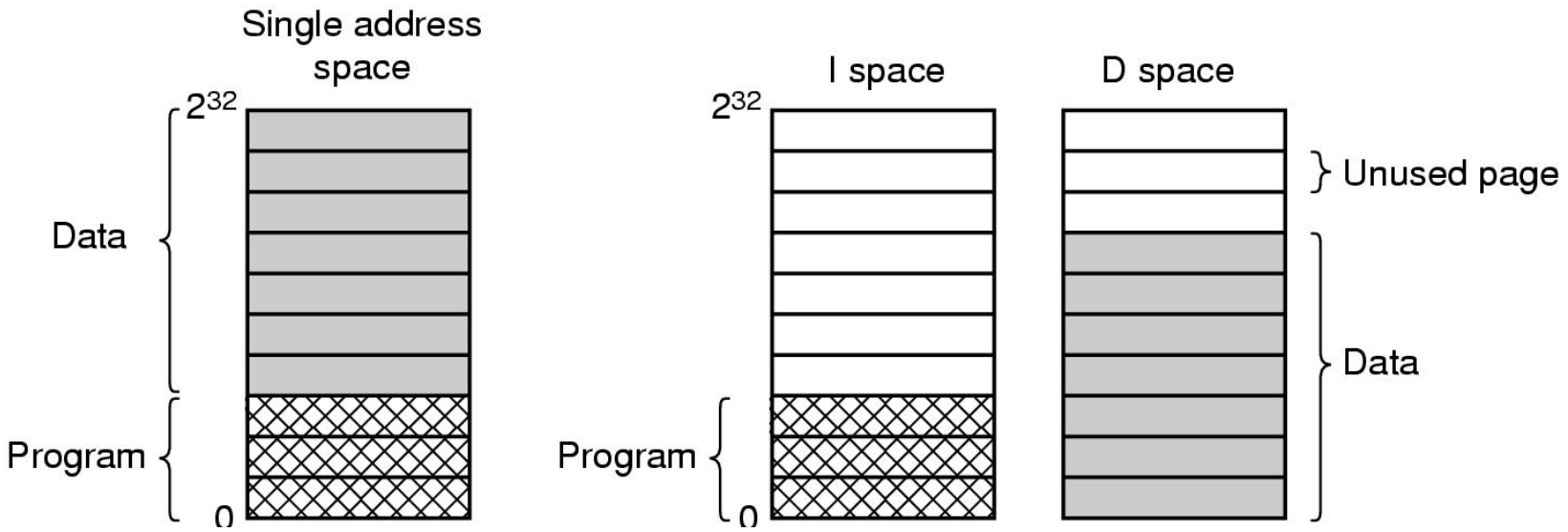


Figure 3-25. (a) One address space.
(b) Separate I and D spaces.

Page Fault Handling (1)

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

Page Fault Handling (2)

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

Page Fault Handling (3)

- Faulting instruction backed up to state it had when it began and program counter reset to point to that instruction.
- Faulting process scheduled, operating system returns to the (assembly language) routine that called it.
- This routine reloads registers and other state information and returns to user space to continue execution, as if no fault had occurred.

Page Size

Small page size

- Advantages
  - less internal fragmentation
  - better fit for various data structures, code sections
  - less unused program in memory
- Disadvantages
  - programs need many pages, larger page tables

Page Size

Page size: p bytes
Average internal fragmentation: p /2 bytes

Smaller page -> smaller internal fragmentation
But smaller page -> larger page table

Assume:
n segments (logical units) in memory
Average process size s
Each page entry (in page table) requires e bytes

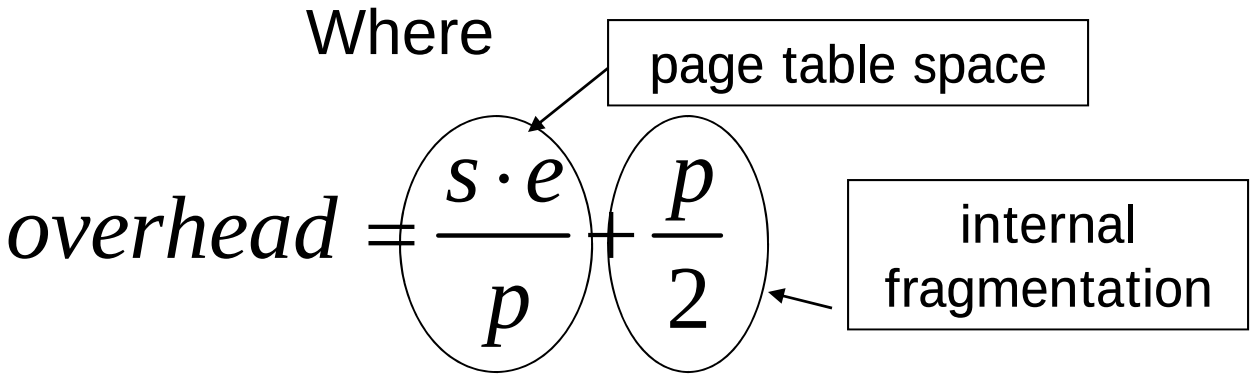
Memory wasted:
Overhead = (s/p)e + p/2
Optimized when

$$p = \sqrt{2se}$$

Example:
s = 32K and e = 8 -> p = 724 bytes.
Take 512 or 1024.

Page Size

Overhead due to page table and internal fragmentation



- s = average process size in bytes
- p = page size in bytes
- e = page entry

Optimized when

$$p = \sqrt{2se}$$

Segmentation

A compiler has many tables that are built up as compilation proceeds, possibly including:

- The source text being saved for the printed listing (on batch systems).
- The symbol table – the names and attributes of variables.
- The table containing integer, floating-point constants used.
- The parse tree, the syntactic analysis of the program.
- The stack used for procedure calls within the compiler.

Segmentation

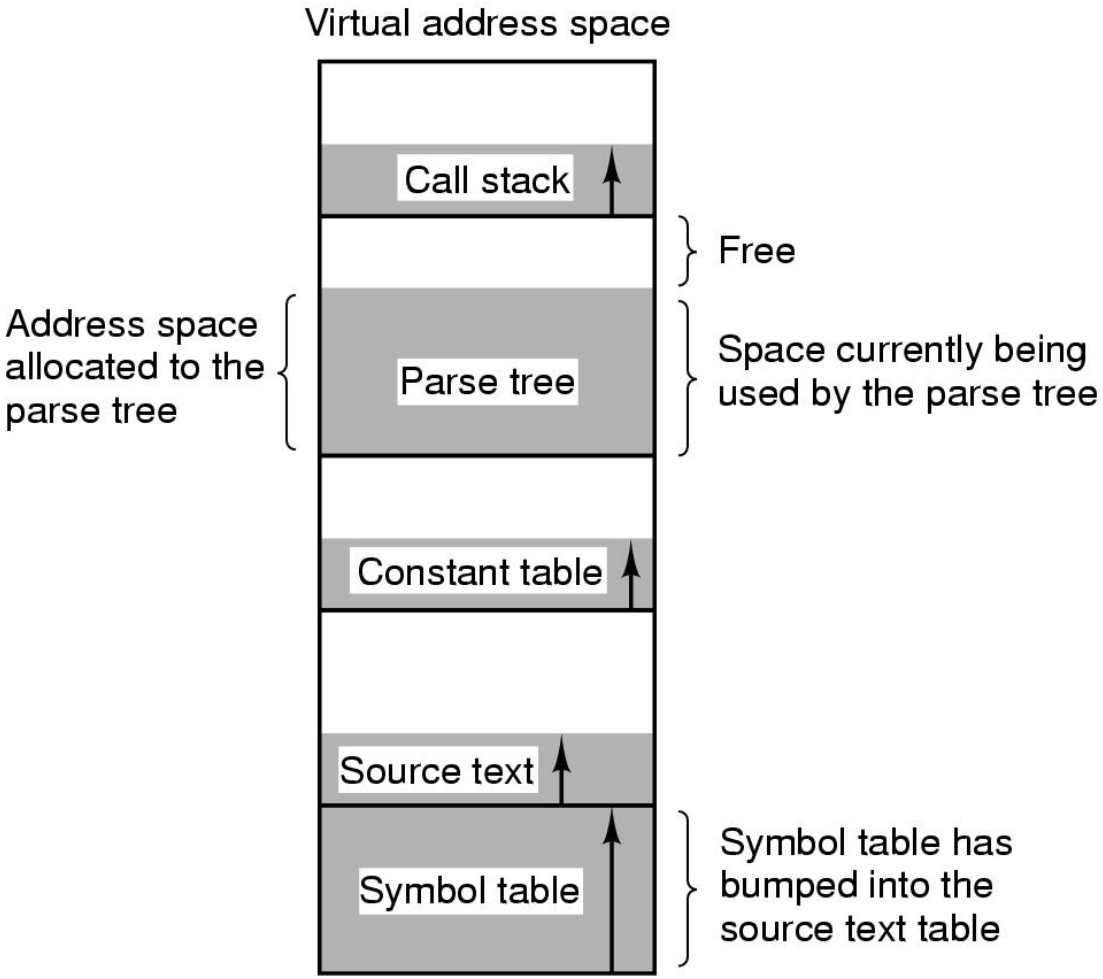


Figure 3-31. In a one-dimensional address space with growing tables, one table may bump into another.

Segmentation

Segmentation system

– idea: access files using memory address
– two dimension address space: segment and offset
– divide logical address space into segments. Each segment is a logical unit, such as data, code, and file.
– segment size is variable, and often large
– Segment table shows the mapping between logical addresses and physical addresses.

Segmentation

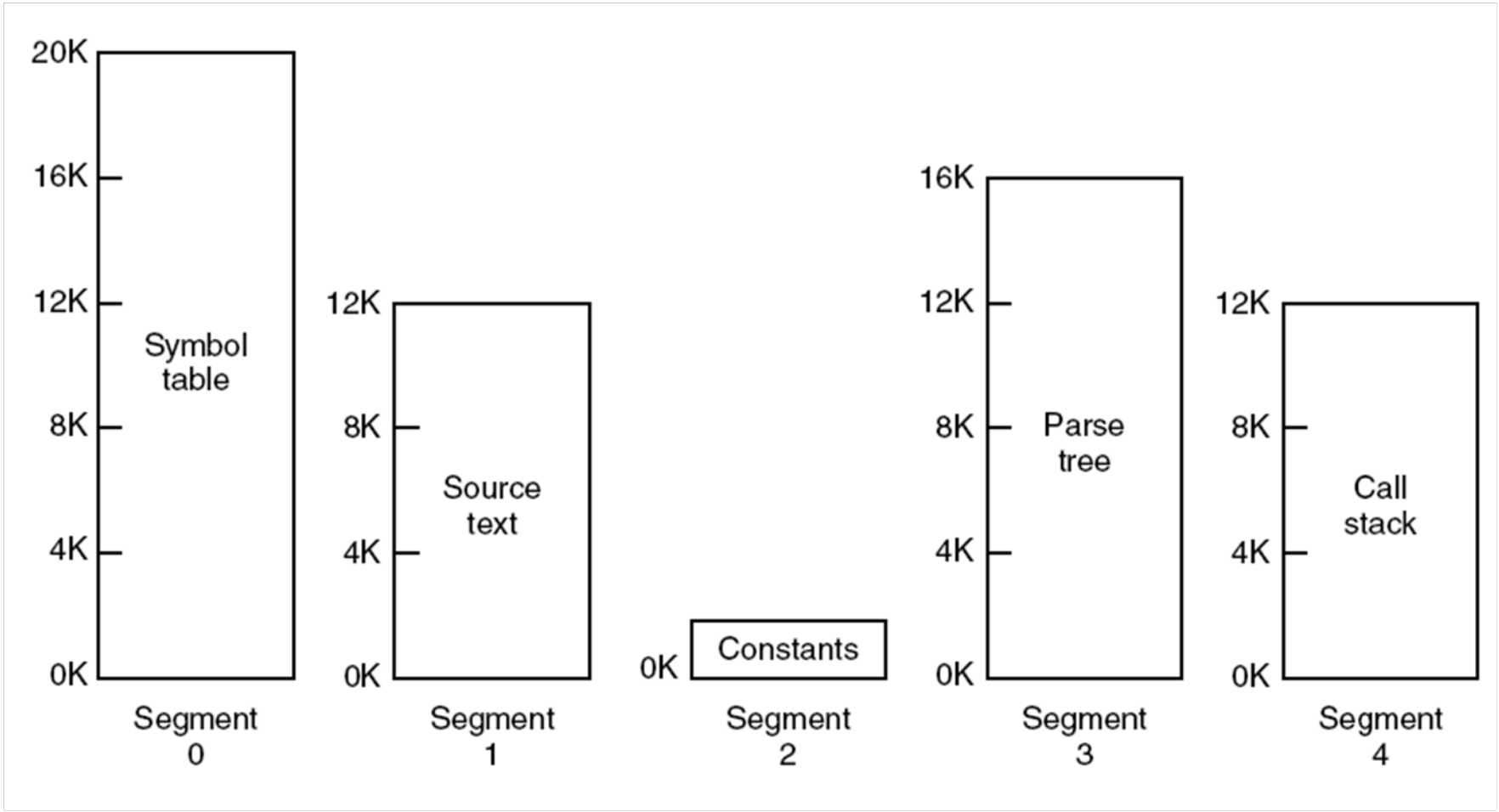


Figure 3-32. A segmented memory allows each table to grow or shrink independently of the other tables.

Implementation of Pure Segmentation

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 3-33. Comparison of paging and segmentation.

Segmentation with Paging: MULTICS (1)

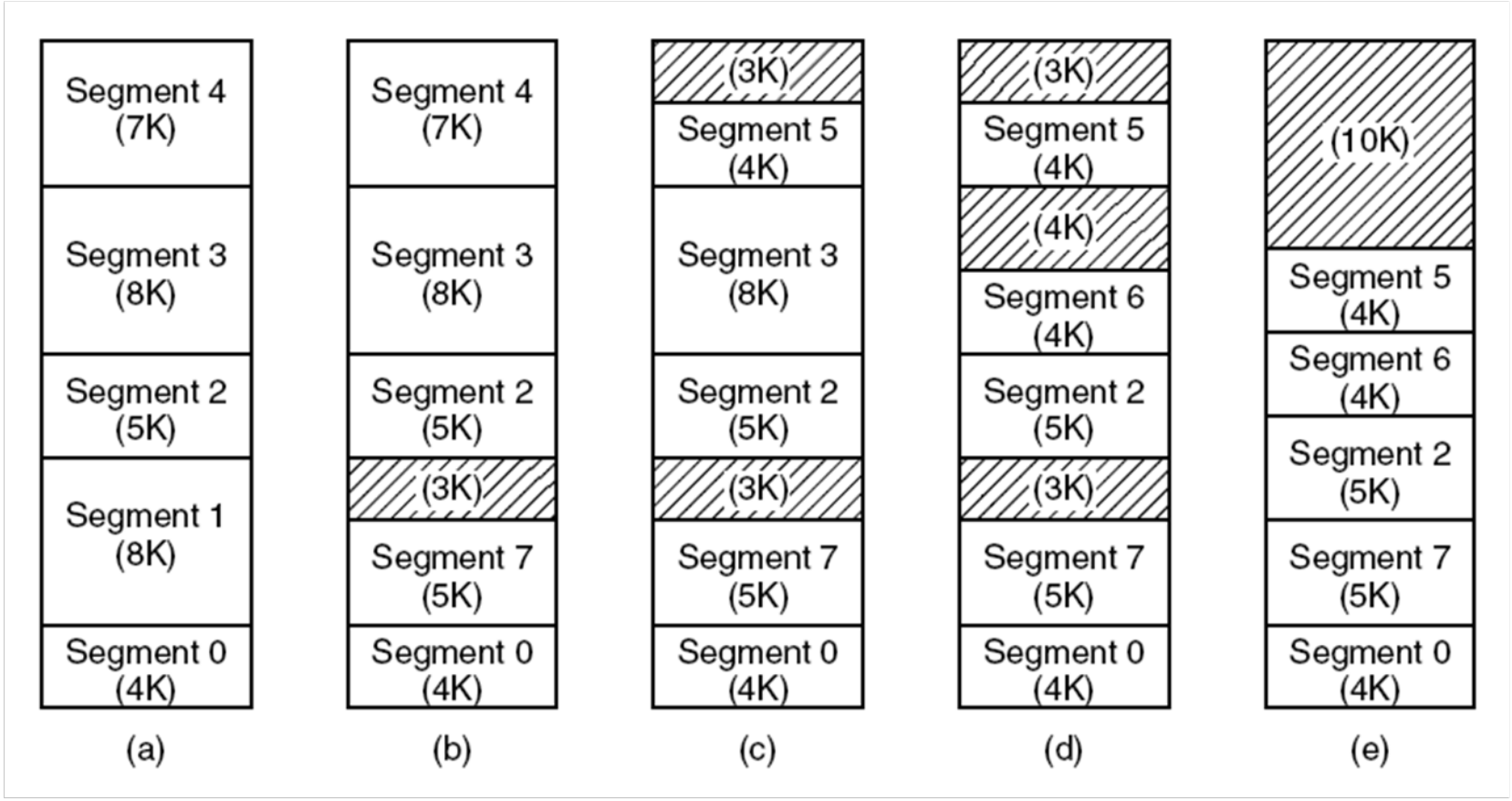


Figure 3-34. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

Segmentation with Paging

Combine segmentation with paging system

– Each segment must start at a page boundary. Divide each segment into pages and each segment has a page table.

– Easy to share a file.

Segmentation with Paging: MULTICS (2)

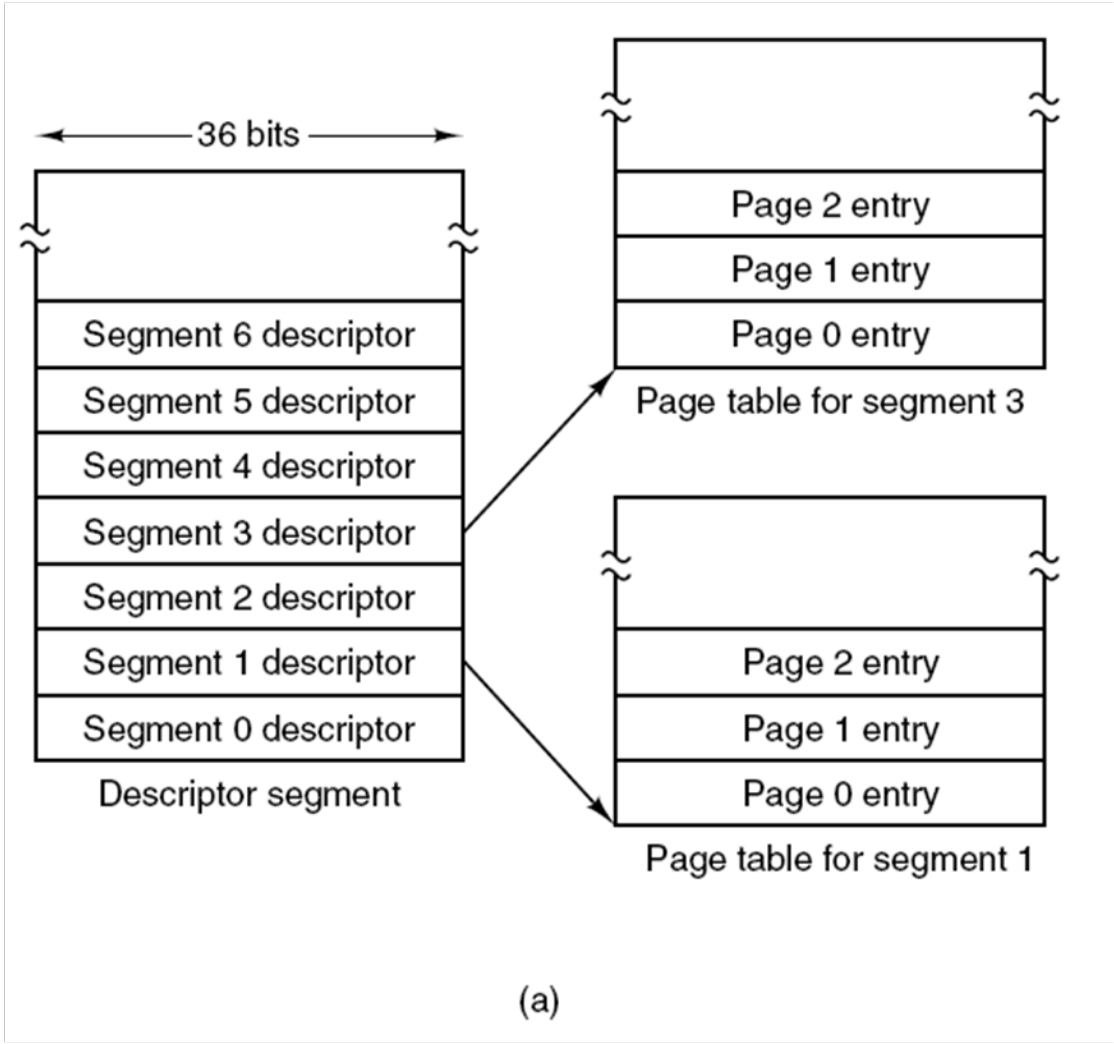
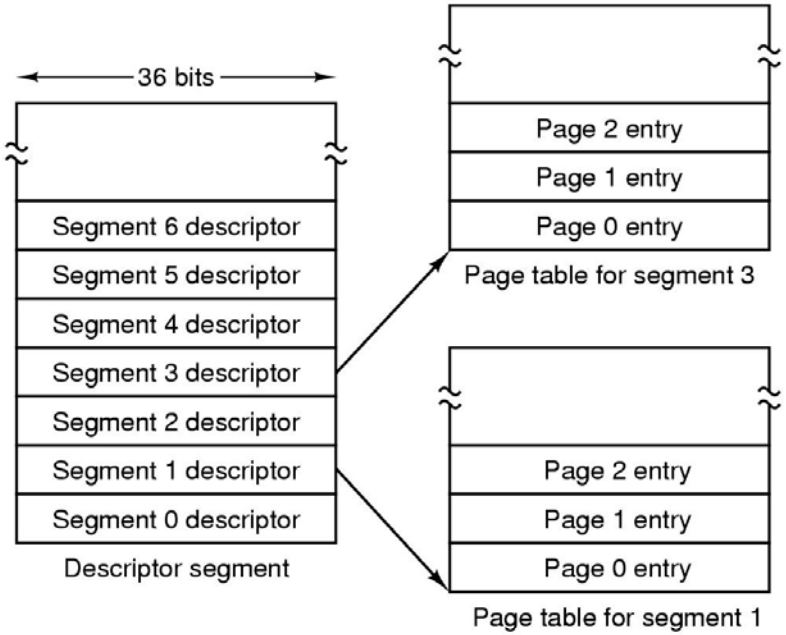


Figure 3-35. The MULTICS virtual memory. (a) The descriptor segment points to the page tables.

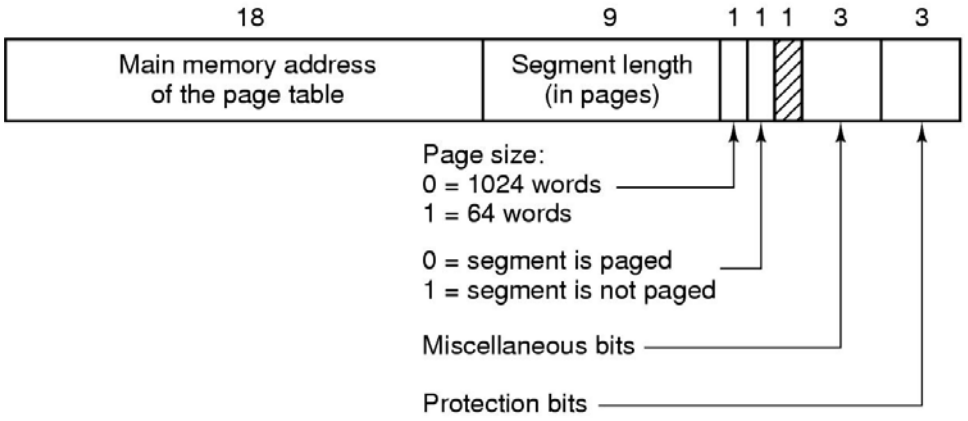
Segmentation with Paging: MULTICS (5)

(a) MULTICS virtual memory.

(b) A segment descriptor. The numbers are the field lengths.



(a)



(b)

Segmentation with Paging: MULTICS (6)

When a memory reference occurs, the following algorithm is carried out:

- The segment number used to find segment descriptor.
- Check is made to see if the segment's page table is in memory.
  - If not, segment fault occurs.
  - If there is a protection violation, a fault (trap) occurs.

Segmentation with Paging: MULTICS (7)

- Page table entry for the requested virtual page examined.
  - If the page itself is not in memory, a page fault is triggered.
  - If it is in memory, the main memory address of the start of the page is extracted from the page table entry
- The offset is added to the page origin to give the main memory address where the word is located.
- The read or store finally takes place.

Segmentation with Paging: MULTICS (8)

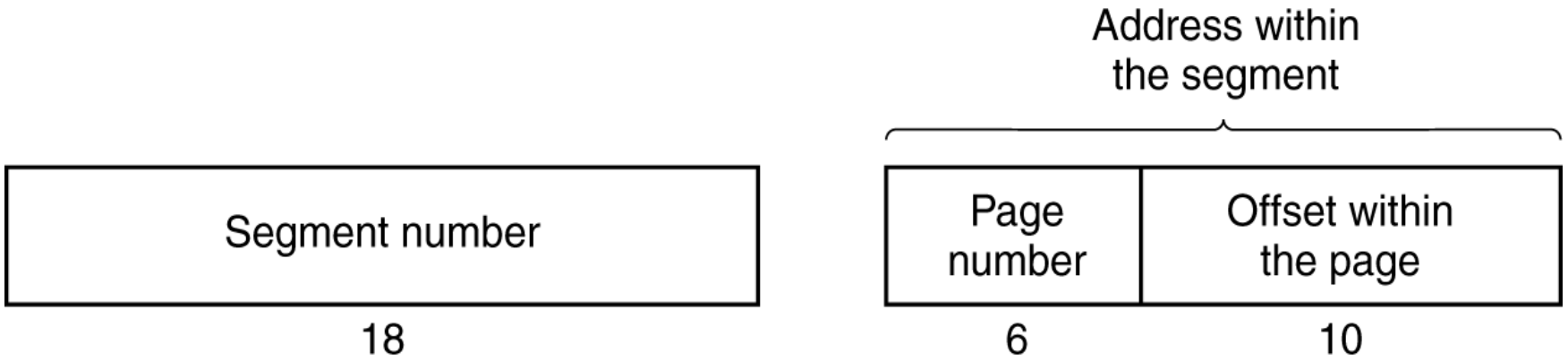


Figure 3-36. A 34-bit MULTICS virtual address.

Segmentation with Paging: MULTICS (9)

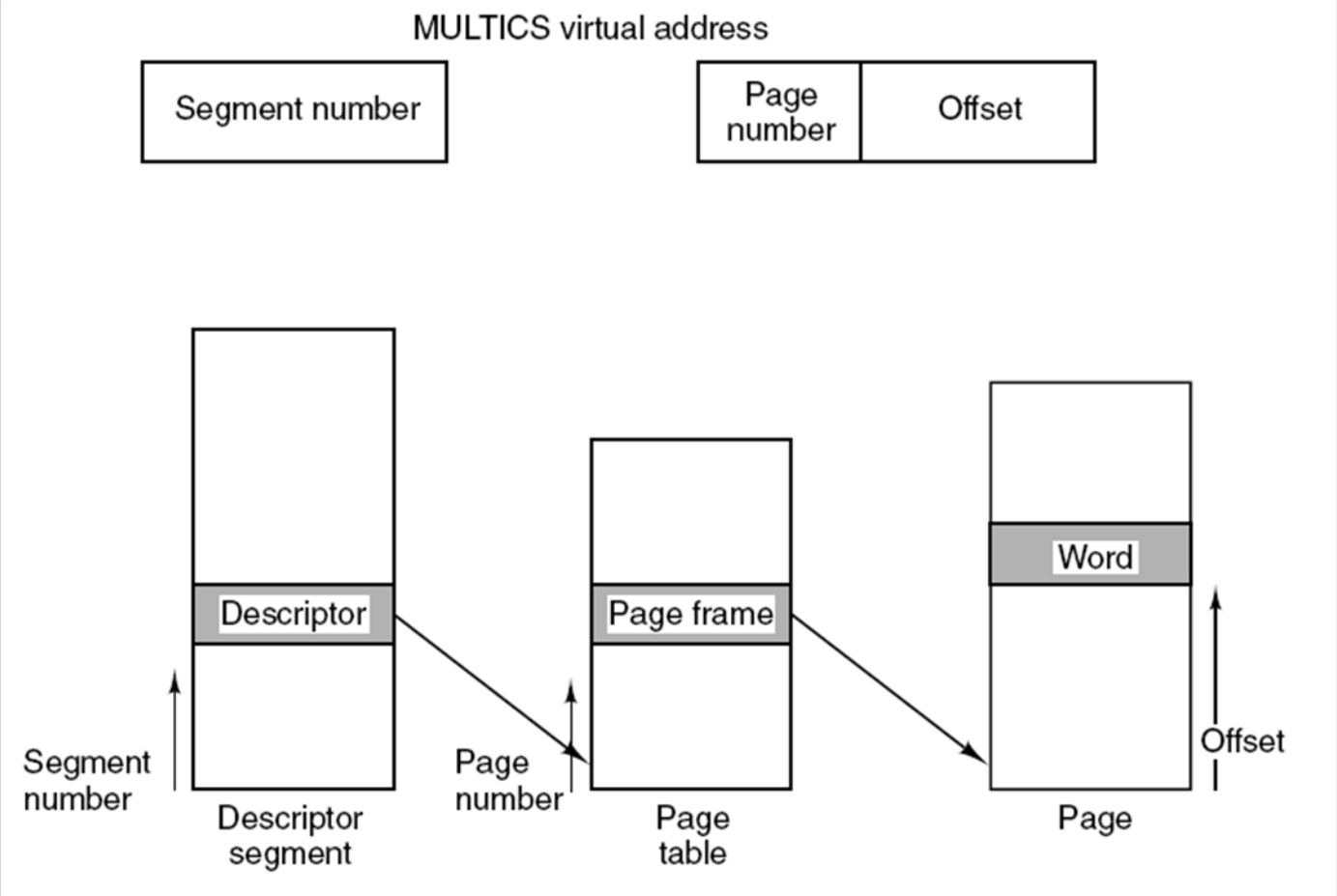
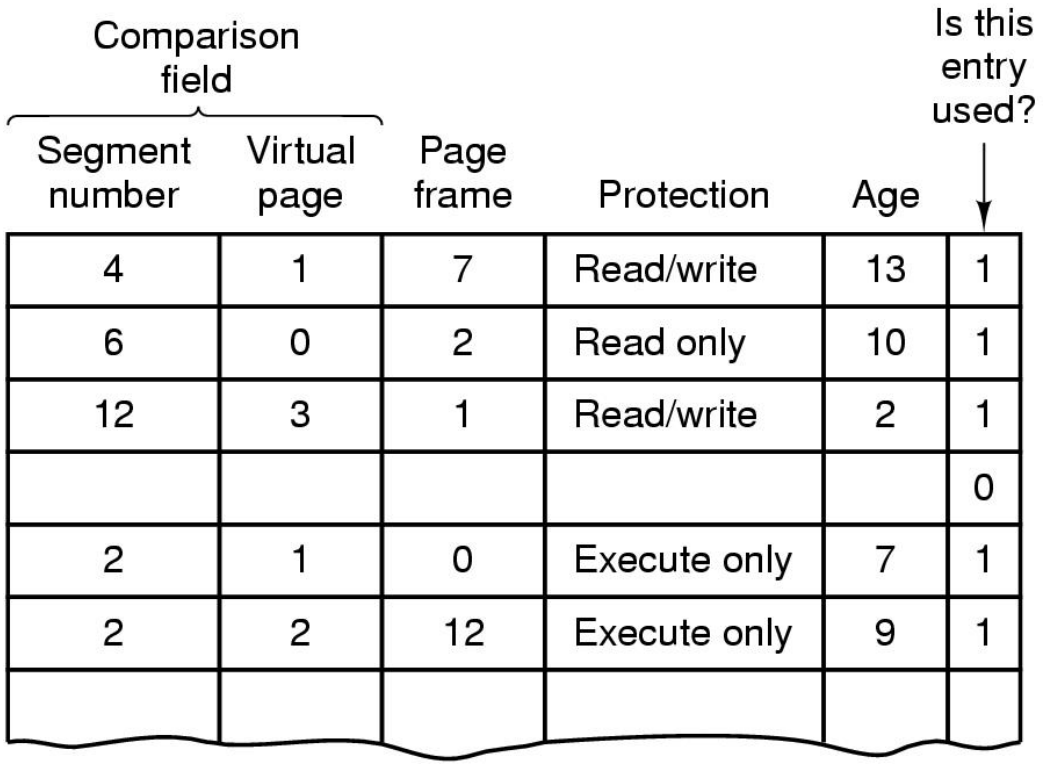


Figure 3-37. Conversion of a two-part MULTICS address into a main memory address.

Segmentation with Paging: MULTICS (10)



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 3-38. A simplified version of the MULTICS TLB. The existence of two page sizes makes the actual TLB more complicated.

Systems Calls for Semaphore and Shared Memory

- Semaphore system calls

Get a set of semaphores

semid = semget(key, nsems, permags)

key: a user defined name for the semaphore set
nsems: number of semaphores in the set
permags: permission state (read, write)
semid: semaphore set identifier associated with key, used by other semaphore operations.

Four ways to use it

Systems Calls for Semaphore and Shared Memory

•Create a private semaphore set
semid = semget(IPC PRIVATE, nsems, 0600|IPC_CREAT|IPC_ EXCL)
Return a unique semid system wide, private to the process.

•Find key if already defined.
semid = semget(key, nsems, 0)
key not equal to IPC PRIVATE, e.g. 0X200.

•Create only if key is not already defined
semid = semget(key, nsems, 0600|IPC_CREAT|IPC_EXCL)
If other processes specify the same key, they will get the same semid.

•Find key if already defines, otherwise create
semid = semget(key, nsems, 0600|IPC_CREAT)

SHELL commands for IPC:
ipcs: check ipc state
ipcrm -s semid: remove a semaphore.

Systems Calls for Semaphore and Shared Memory

Semaphore control operations

semval= semctl(semid, index, GETVAL, val)

Get the value of the semaphore
index: index in the set, e.g. 0 means the first semaphore in the set.

semval= semctl(semid, index, SETVAL, val)

Set the semaphore value to val.

pid= semctl(semid, 0, GETPID, val)
Return the process id of the last process that performs an operation on the semaphore.

Systems Calls for Semaphore and Shared Memory

Semaphore operations (up and down)

semop(semid, op_array, somevalue)

op_array: an array of semaphore operations to perform
somevalue: the number of semaphore operation records
struct sembuf op_array[somevalue] has three fields:
sem num: index to semaphore in the set
sem op: -1: down; +1: up
sem ag: usually set to SEM UNDO,
automatically ``undo'' all operations after process exits.

Example:
sem num= 0;
sem op = -1;
sem ag = SEM UNDO

Systems Calls for Semaphore and Shared Memory

Shared memory system calls

Create shared memory segment

shmid = shmget(key, size, 0600|IPC CREAT|IPC_EXCL)

size: number of bytes.

Systems Calls for Semaphore and Shared Memory

Shared memory operations

shmat(shmid, dataptr, flag)
Attach the memory segment identified by shmid to process's logical data space

dataptr = 0: the segment is attached to the first available address selected by the system

dataptr nonzero: attach to user specified address, depending on flag:

flag & SHM RND is true. shmat will round dataptr to a page boundary
flag & SHM RND is false. attach to the exact values of dataptr
flag & SHM RDONLY is true. Read only.

Example:

```
struct databuf *pp;
pp = (struct databuf  *) shmat(shmid, 0, 0);
```

Systems Calls for Semaphore and Shared Memory

Shared memory control operations

shmctl(shmid, command, &shm_stat)

After your are done, remove the shared memory identifier specified by shmid from the system and destroy the shared memory segment and data structures associated to it:

shmctl(shmid, IPC_RMID, (struct shmid_ds *)0)

Systems Calls for Signals

Signals are called ``software interrupts.'' One process can send signals to another process.

Signal names: SIGINT, SIGKILL, SIGALRM, ...

What to do with a signal system call?

•Catch a signal: signal(sig, func).
Provide a function that is called whenever a specific type of signal occurs. Need to re-enable signal catching.

•Ignore a signal: signal(sig, SIG_IGN).
All signasl, other than SIGKILL, can be ignored.

•Allow the default to happen: signal(sig ,SIG_DEF).
Normally, a process is terminated when receiving a signal.

Send a signal: kill(pid, sig).