Memory Management

Memory Management

- 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

Memory Hierarchy

- » How does the operating system create abstractions from memory, and how does it manage them?
- » Memory hierarchy:
  - » Few megabytes of very fast, very expensive, volatile cache memory
  - » A few gigabytes of medium-speed, medium priced, volatile main memory
  - » A few terabytes of slow, cheap, nonvolatile magnetic or solid state disk storage.

Memory Hierarchy

- » Memory Manager manages it
  - » Tracks which parts of memory are in use
  - » Allocates memory to processes
  - » Deallocates memory when processes are done.

No Memory Abstraction

» Early mainframes (before 1960), early minicomputers (before 1970), early personal computers (before 1980) had no memory abstraction

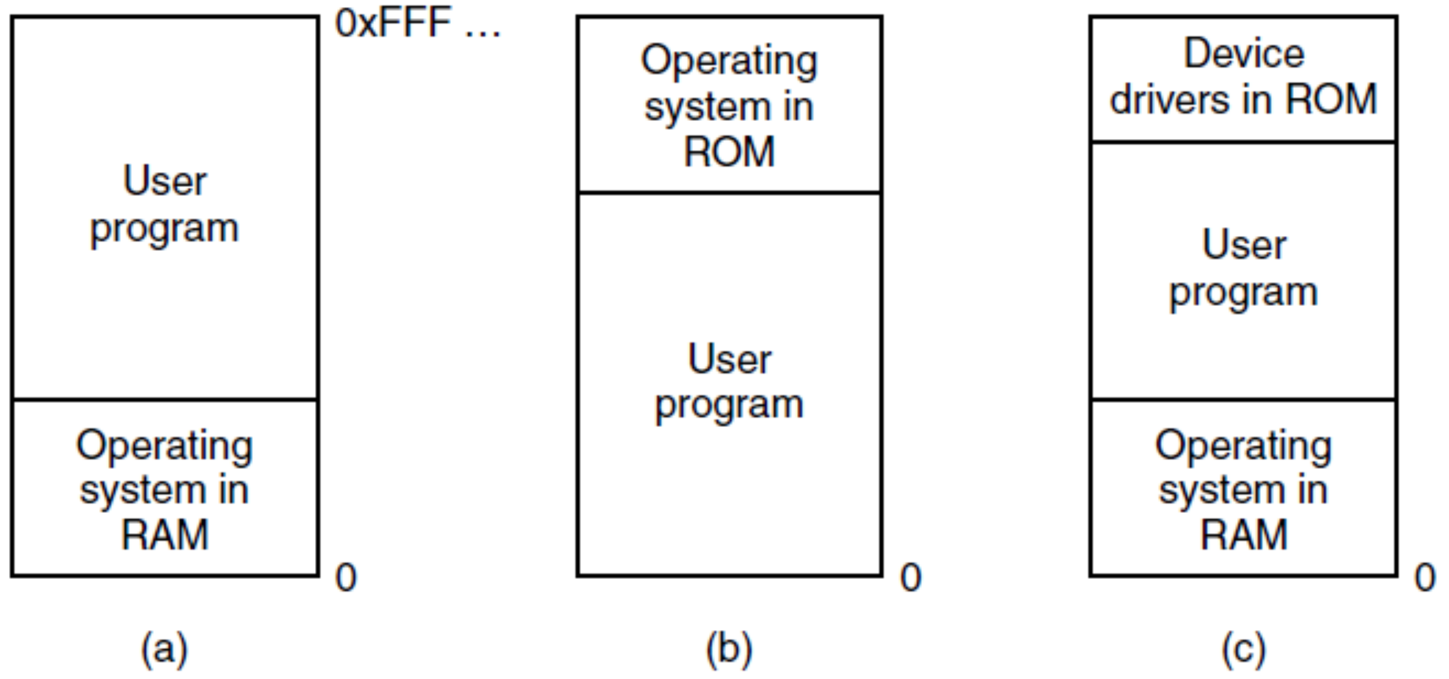
» Every program saw physical memory.

» Programmers saw a set of addresses from 0 to some maximum.

No Memory Abstraction

- » Two programs could not be resident in memory at the same time.
  - » Address conflicts
  - » Could run multi-threaded since threads share address space.
  - » Limited use. Users want unrelated programs to be running at once.

Memory Layout

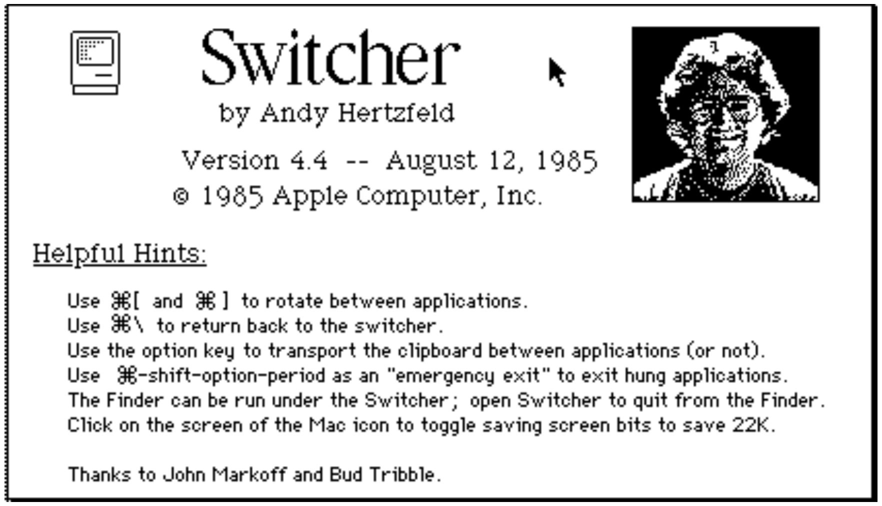


No Memory Abstraction

- » Multiple programs possible
  - » Save entire contents of memory to a disk file
  - » Bring in new and run it
  - » Swapping.
- » Additional hardware will allow concurrency without swapping.

Switcher

- Switcher worked by designating a number of fixed "slots" in memory
    - Applications could be loaded into those slots
    - The user could then switch between these applications by clicking a button above the menu bar. The current application would horizontally slide out of view, and the next one would slide in.
    - Worked with existing memory management so changes were transparent to the OS and to the running programs



Switcher
by Andy Hertzfeld
Version 4.4 -- August 12, 1985
© 1985 Apple Computer, Inc.

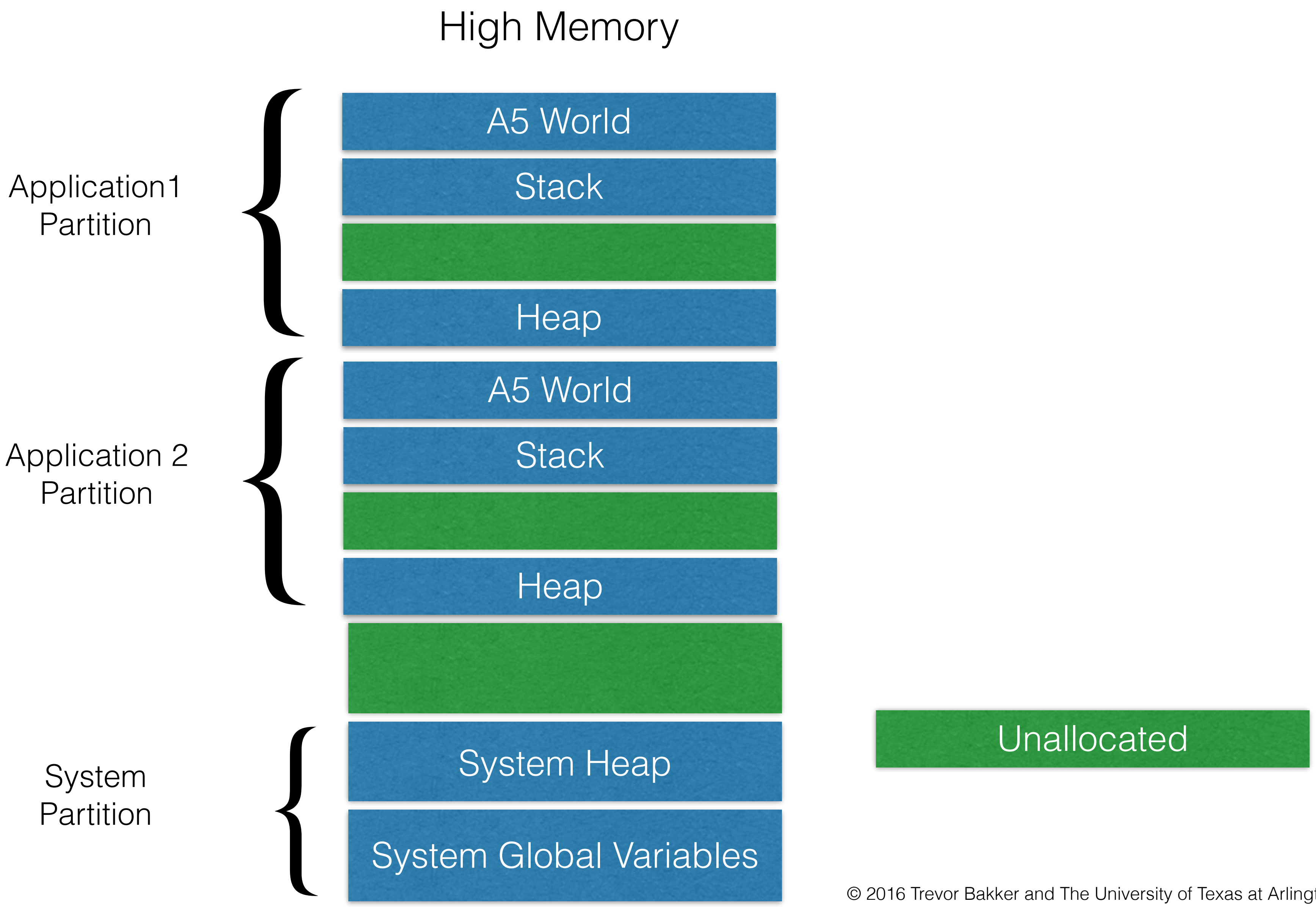
Helpful Hints:

Use ⌘[and ⌘] to rotate between applications.
Use ⌘\ to return back to the switcher.
Use the option key to transport the clipboard between applications (or not).
Use ⌘-shift-option-period as an "emergency exit" to exit hung applications.
The Finder can be run under the Switcher; open Switcher to quit from the Finder.
Click on the screen of the Mac icon to toggle saving screen bits to save 22K.

Thanks to John Markoff and Bud Tribble.

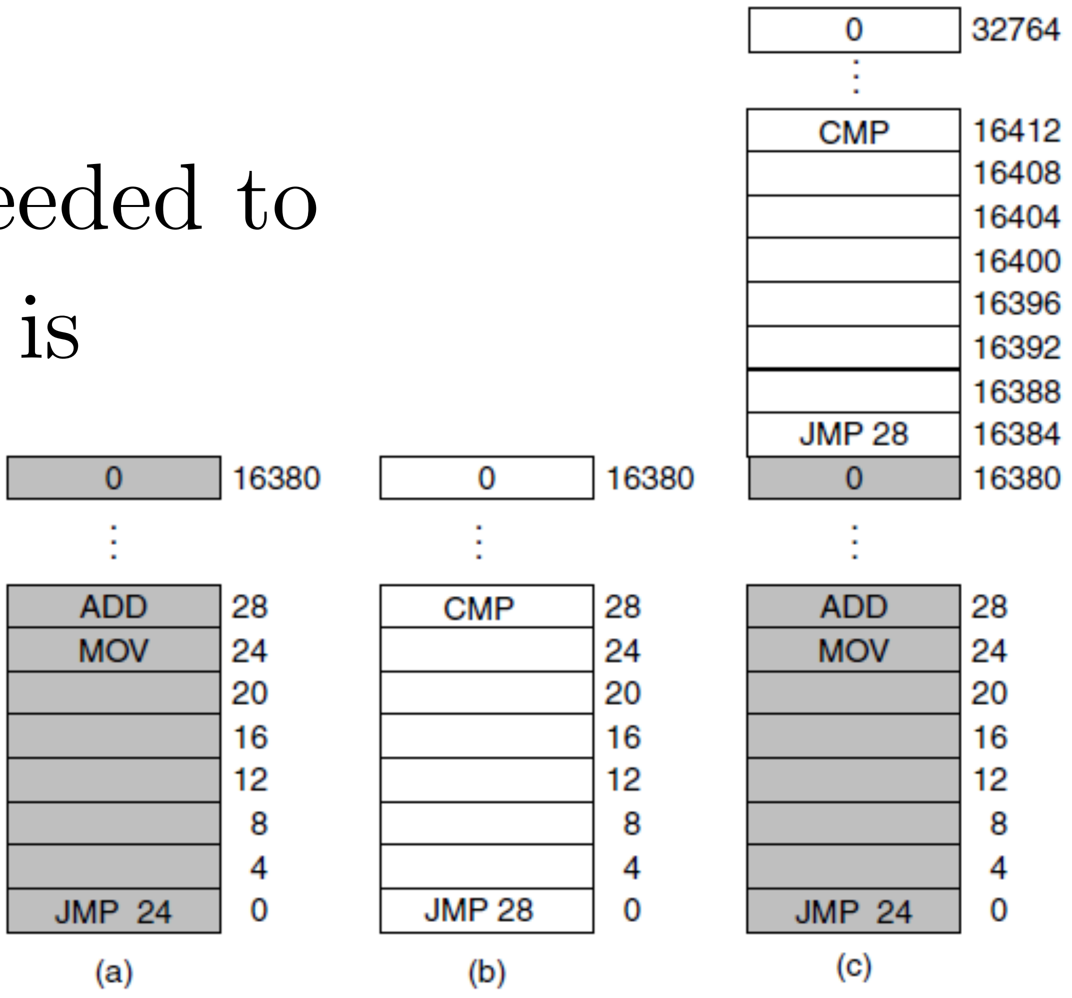
http://www.folklore.org/StoryView.py?project=Macintosh&story=Switcher.txt

Switcher Memory Layout



No Memory Abstraction

- IBM 360 solution: static relocation
  - Extra info needed to specify what is relocatable



Address Spaces

- » Exposing memory to processes have several drawbacks
  - » If user programs can address every byte of memory, the OS can be trashed intentionally or by accident.
  - » Difficult to have multiple programs running at once

Address Spaces

- » Abstract memory
  - » Process creates a virtual CPU abstraction
  - » Address space is abstract memory for a process to live in
- » Address space: Set of all addresses that a process can see to address memory
- » Each process has an independent address space

Address Spaces

» Need to map address 28 in one process and address 28 in another process to a separate physical address.

» Dynamic relocation

» Older solution: two special registers

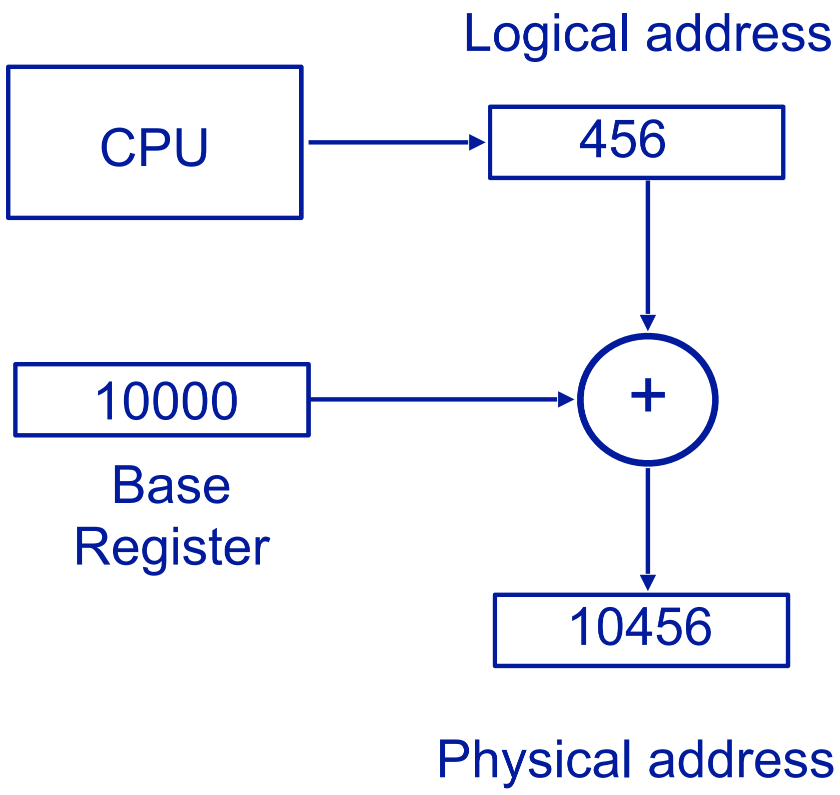
  » Base register

  » Limit register

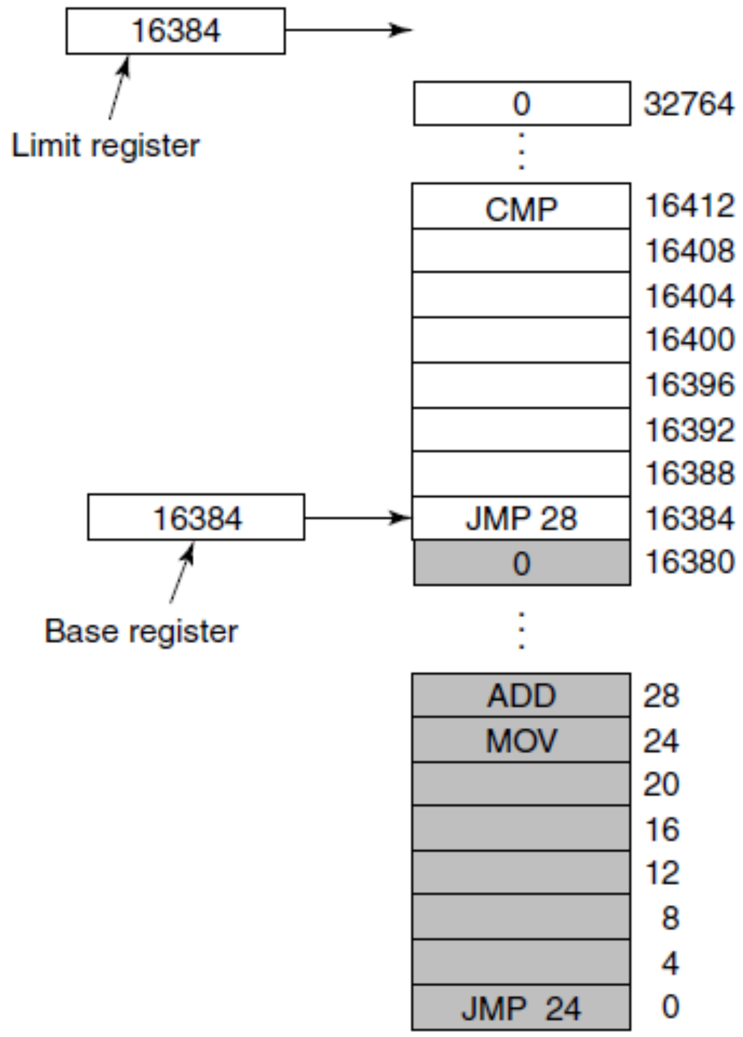
Physical v. Logical Addresses

- » Originally programs were compiled to reference addresses that corresponded one to one with physical memory addresses.
- » Unknowingly using concept of logical and physical memory
  - – Logical addresses - Set of addresses the CPU generates as the program is executed.
  - – Physical addresses - Set of addresses used to reference physical memory.

Dynamic Relocation



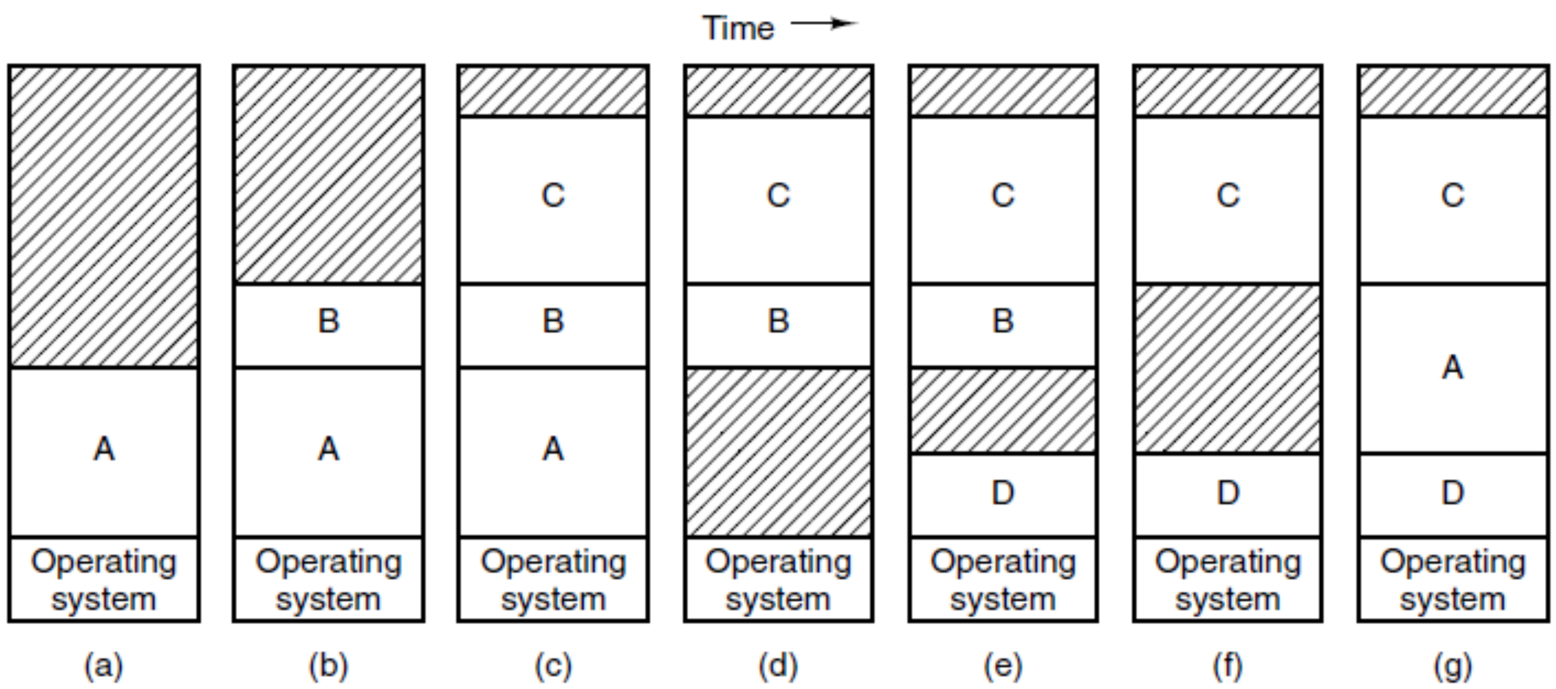
Dynamic Relocation



Swapping

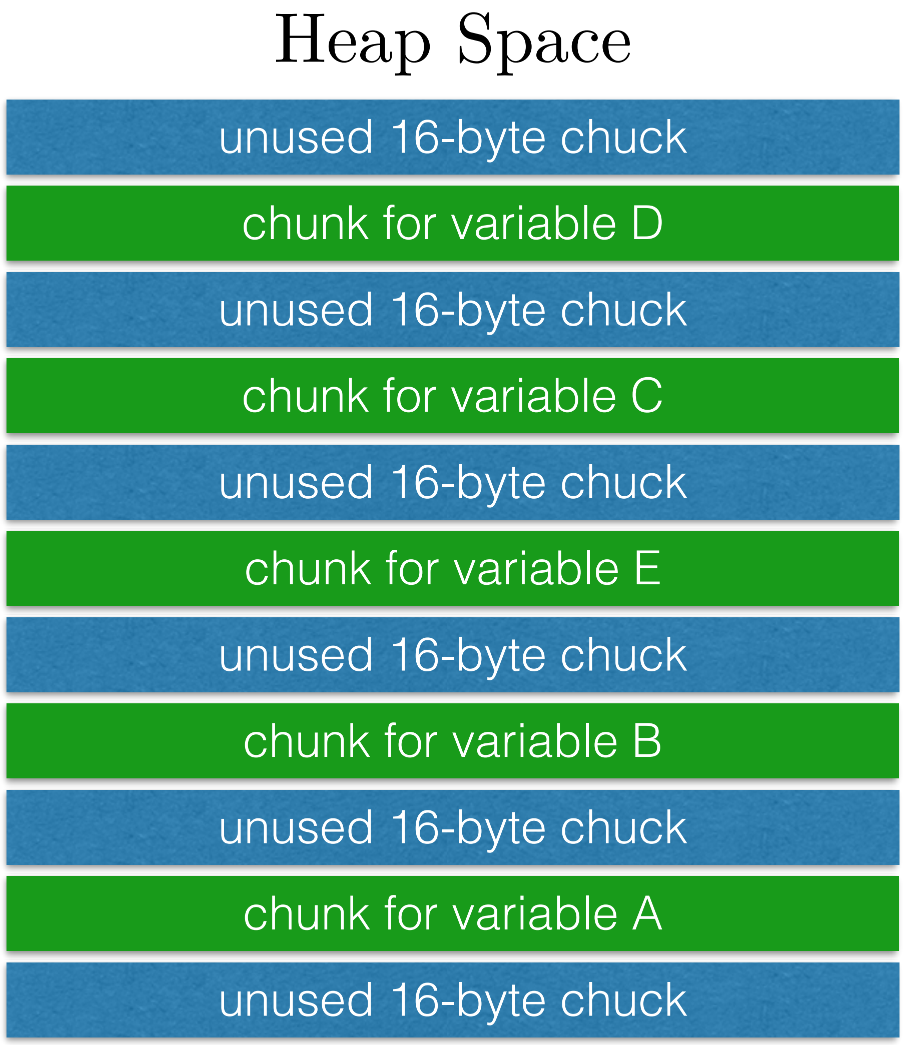
- If physical memory is large enough to hold all the processes, the previous schemes will do
- In practice, total amount of RAM is more that can fit in memory.
- Two approaches:
  - Swapping: Bring in each process in its entirety, run it for a while, then put it back onto disk
  - Virtual Memory: Run processes when they are partially in memory

Swapping



Software or Hardware relocation needed

Heap Fragmentation



96 bytes of free memory but we can't allocate any chuck larger than 16 bytes

External Fragmentation

Compaction

- » The operating system can reorganize the fragmented space so that the free space is contiguous.
- » Expensive

Allocation

- » How much should the OS allocate?
  - » If processes can’t grow, then allocate exactly the amount needed
  - » If the process can grow, and there is a hole next to the process, it can be allocated and the process grow into it.

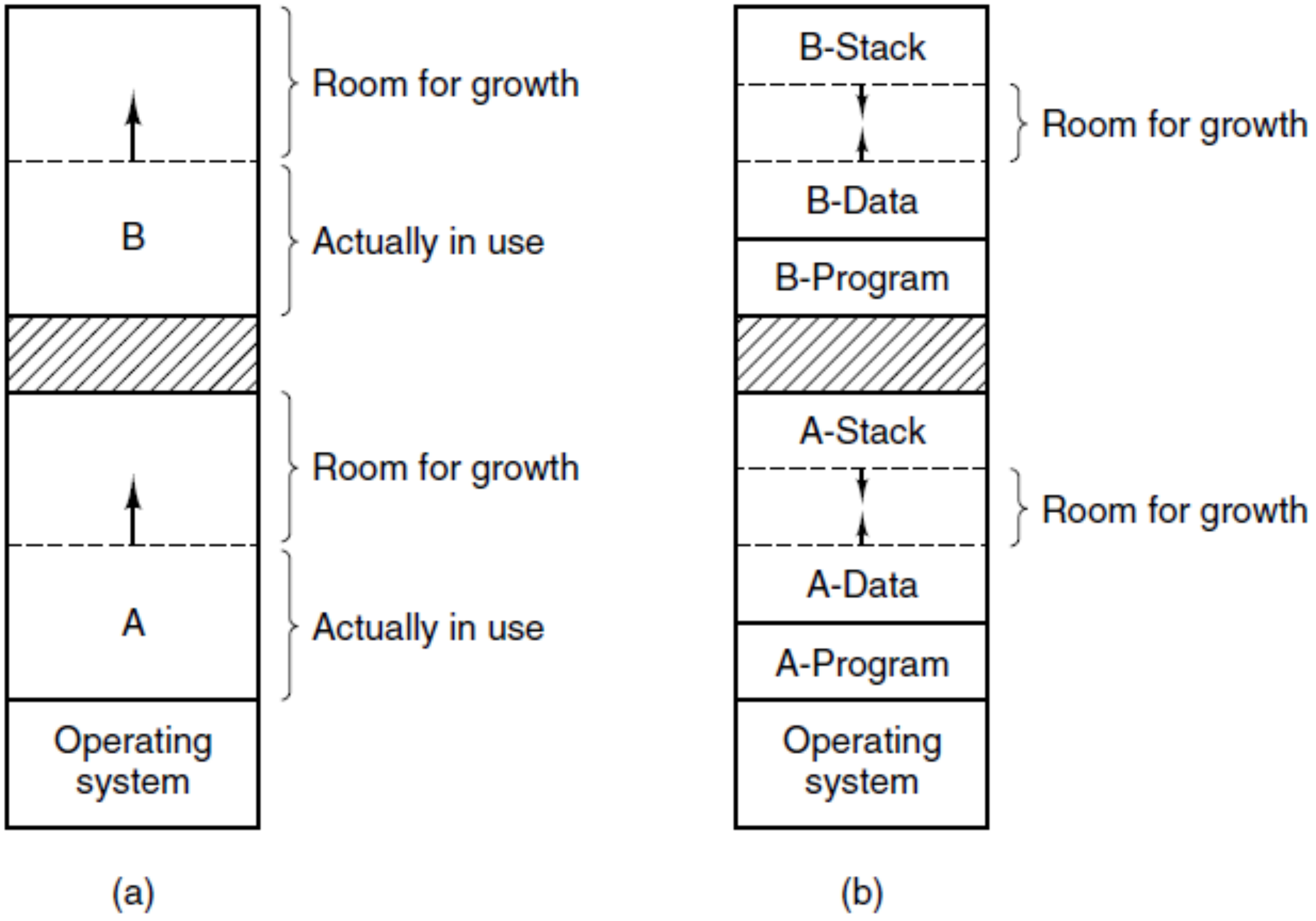
Allocation

- » How much should the OS allocate?
  - » If processes doesn’t have an adjacent hole, it will have to move to a hole in memory large enough or
  - » Swap out one or more processes.
  - » If it can’t grow and no free swap space, suspend or kill the process

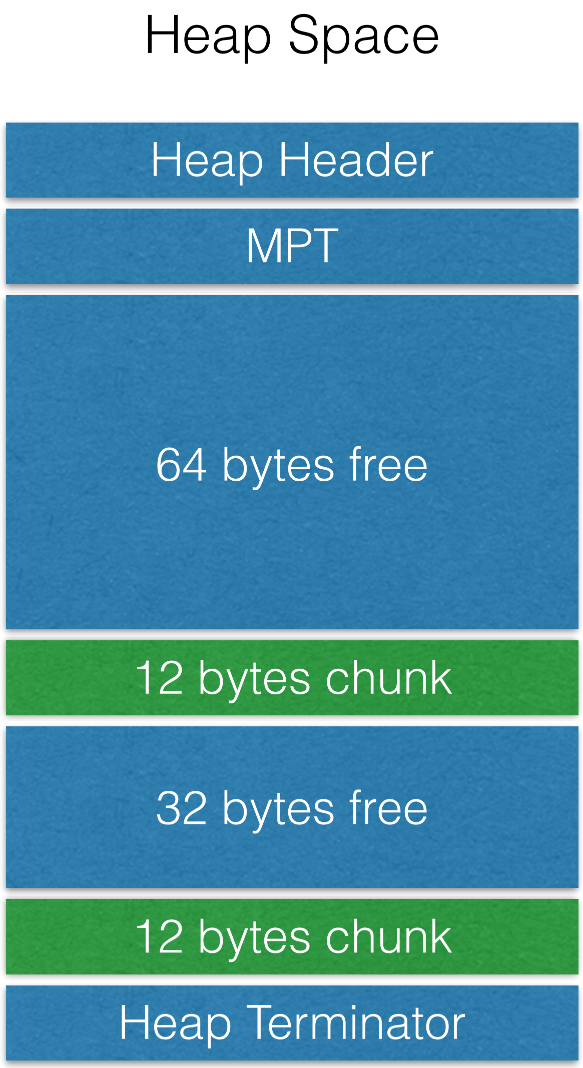
Allocation

- » Good idea to allocate a little extra when a process is swapped in to reduce the overheard with moving or swapping processes.
- » Only memory used should be swapped, otherwise wasteful.

Allocation


B-Stack
Room for growth
B
Actually in use
B-Data
B-Program
Room for growth
A-Stack
Room for growth
A
Actually in use
A-Data
A-Program
Room for growth
Operating system
Operating system
(a)
(b)

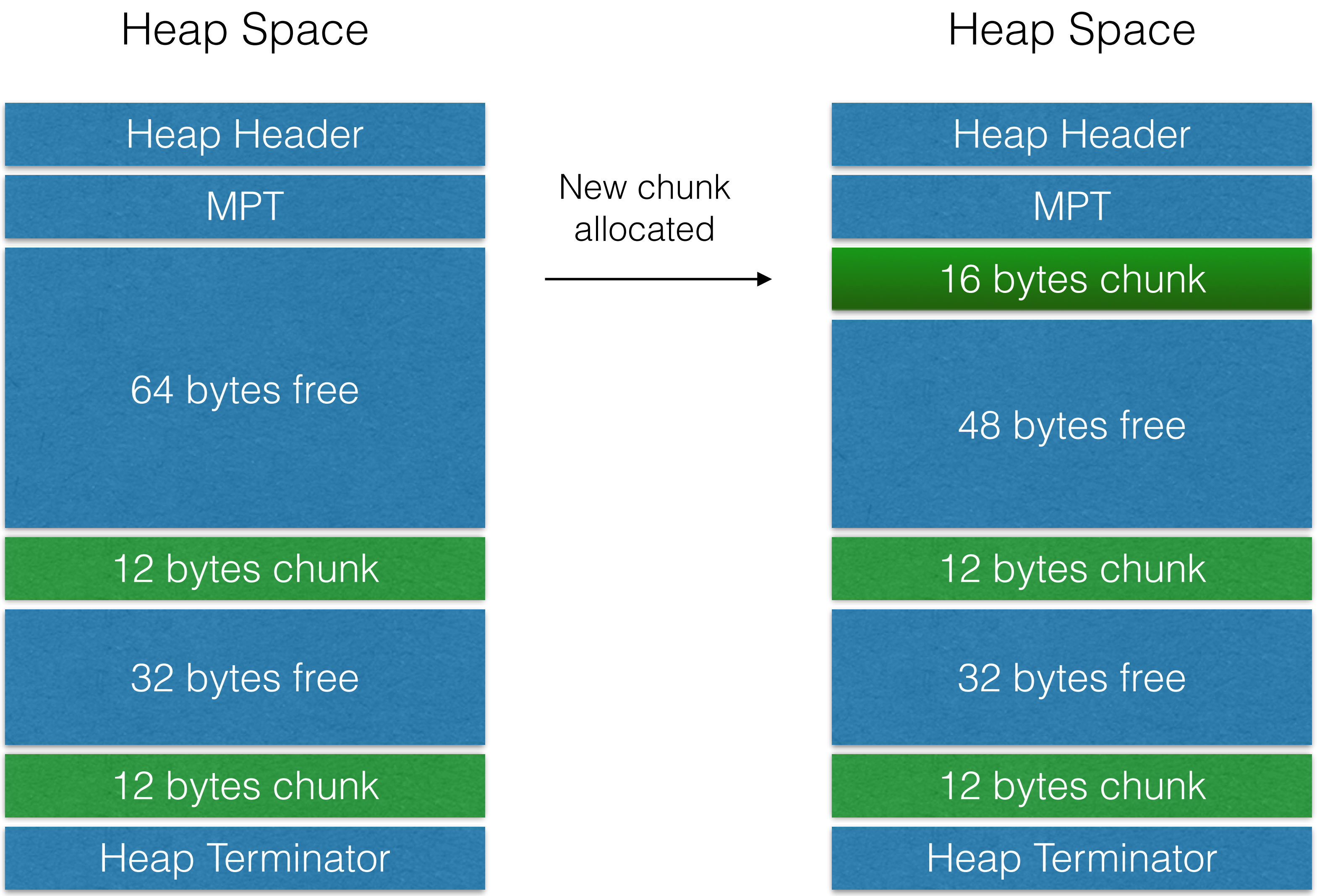
Allocating Memory



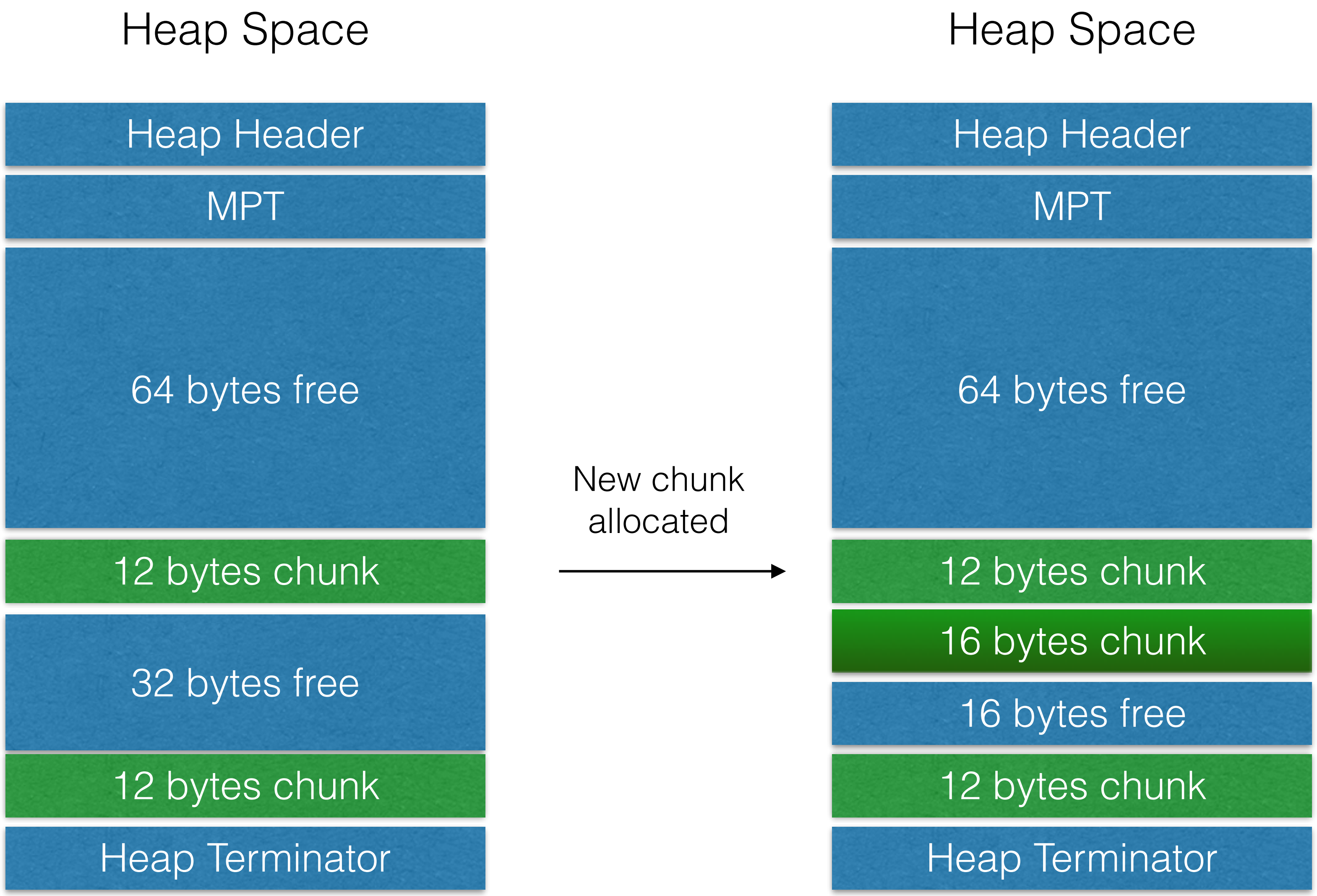
Example: We want to allocate a new 16 byte block.

Which free block does the OS choose?

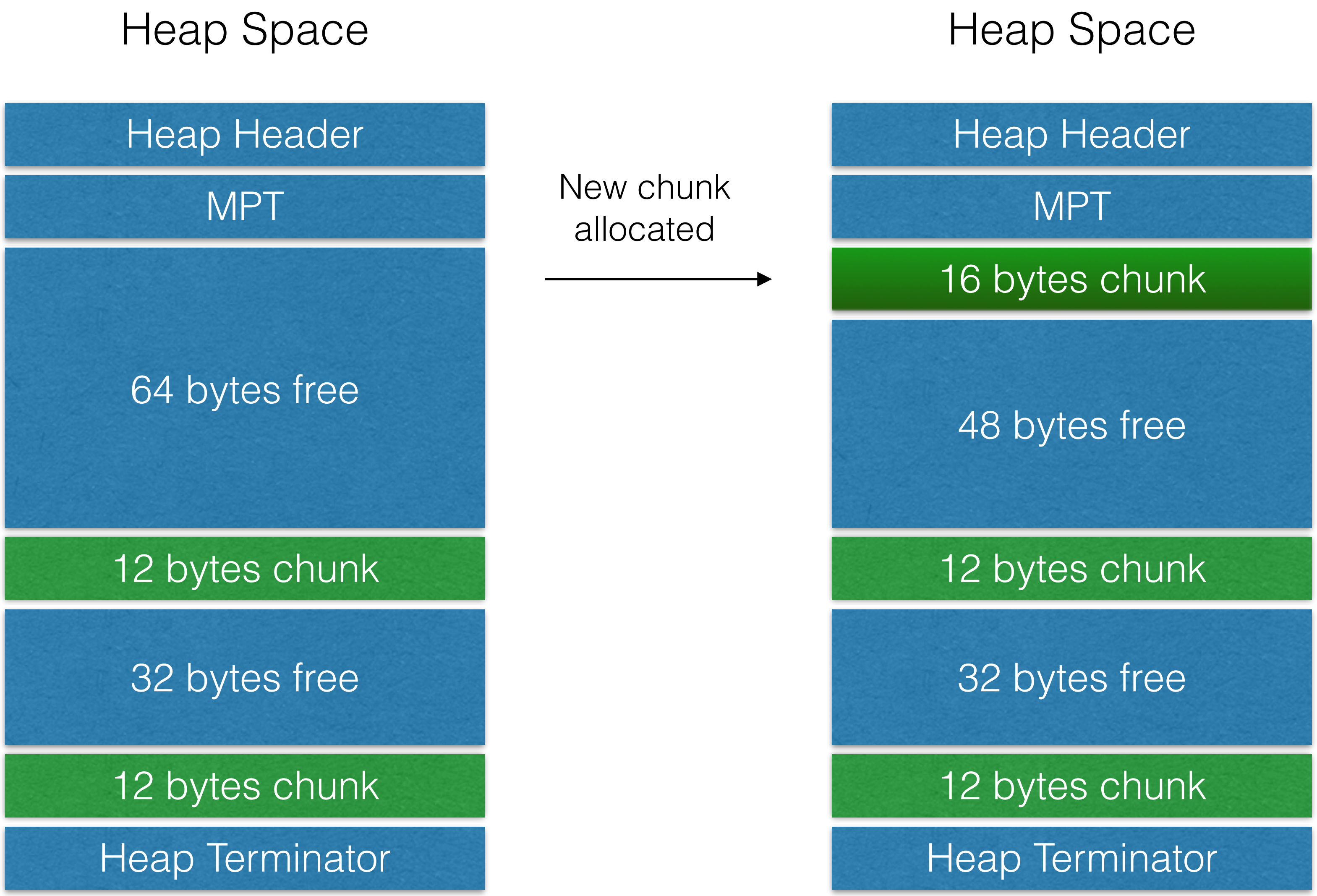
First Fit


Heap Space
Heap Header
MPT
64 bytes free
12 bytes chunk
32 bytes free
12 bytes chunk
Heap Terminator
New chunk allocated
Heap Space
Heap Header
MPT
16 bytes chunk
48 bytes free
12 bytes chunk
32 bytes free
12 bytes chunk
Heap Terminator

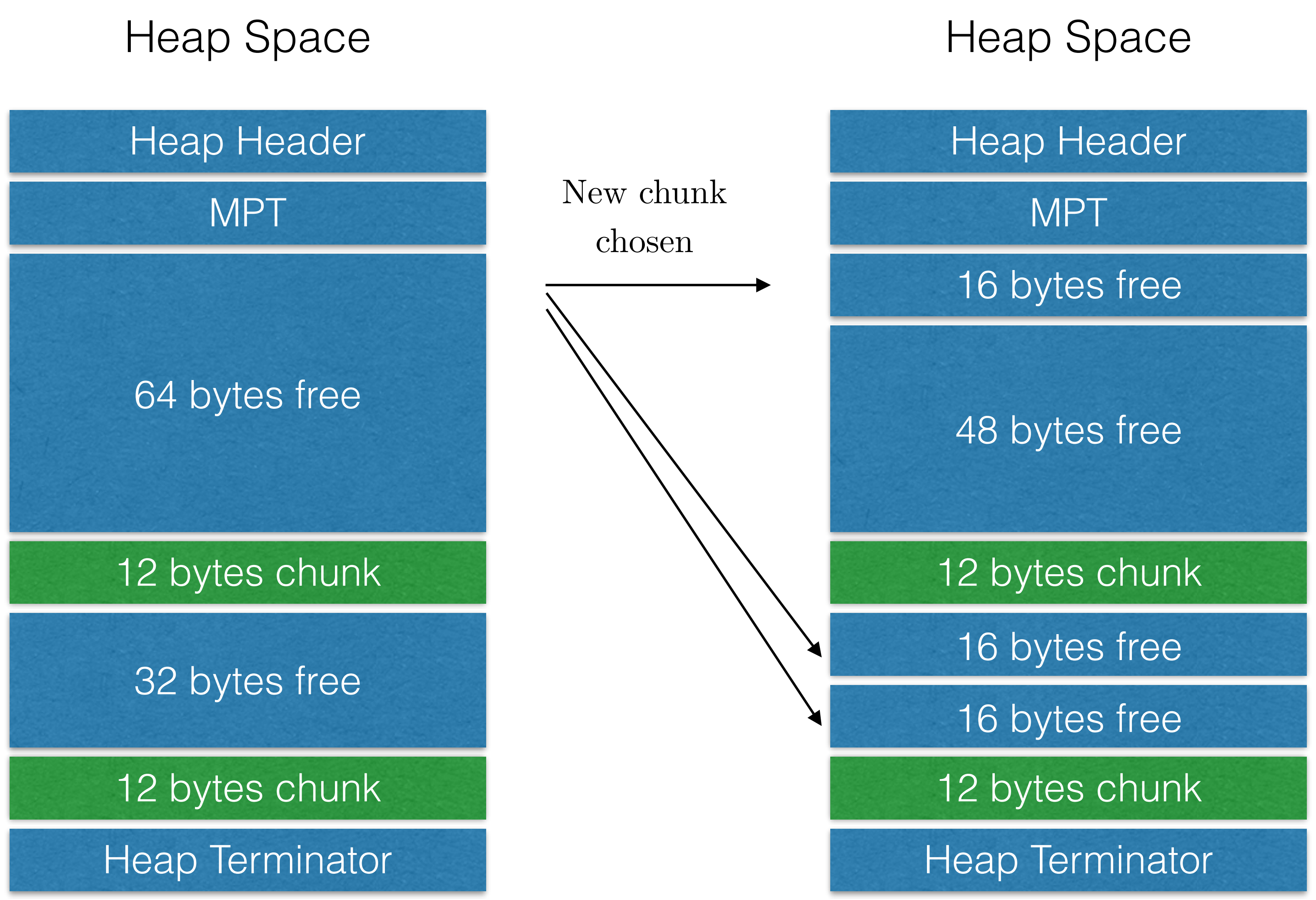
Best Fit


Heap Space
Heap Header
MPT
64 bytes free
12 bytes chunk
32 bytes free
12 bytes chunk
Heap Terminator
New chunk allocated
Heap Space
Heap Header
MPT
64 bytes free
12 bytes chunk
16 bytes chunk
16 bytes free
12 bytes chunk
Heap Terminator

Worst Fit


Heap Space
Heap Header
MPT
64 bytes free
12 bytes chunk
32 bytes free
12 bytes chunk
Heap Terminator
New chunk allocated
Heap Space
Heap Header
MPT
16 bytes chunk
48 bytes free
12 bytes chunk
32 bytes free
12 bytes chunk
Heap Terminator

Quick Fit



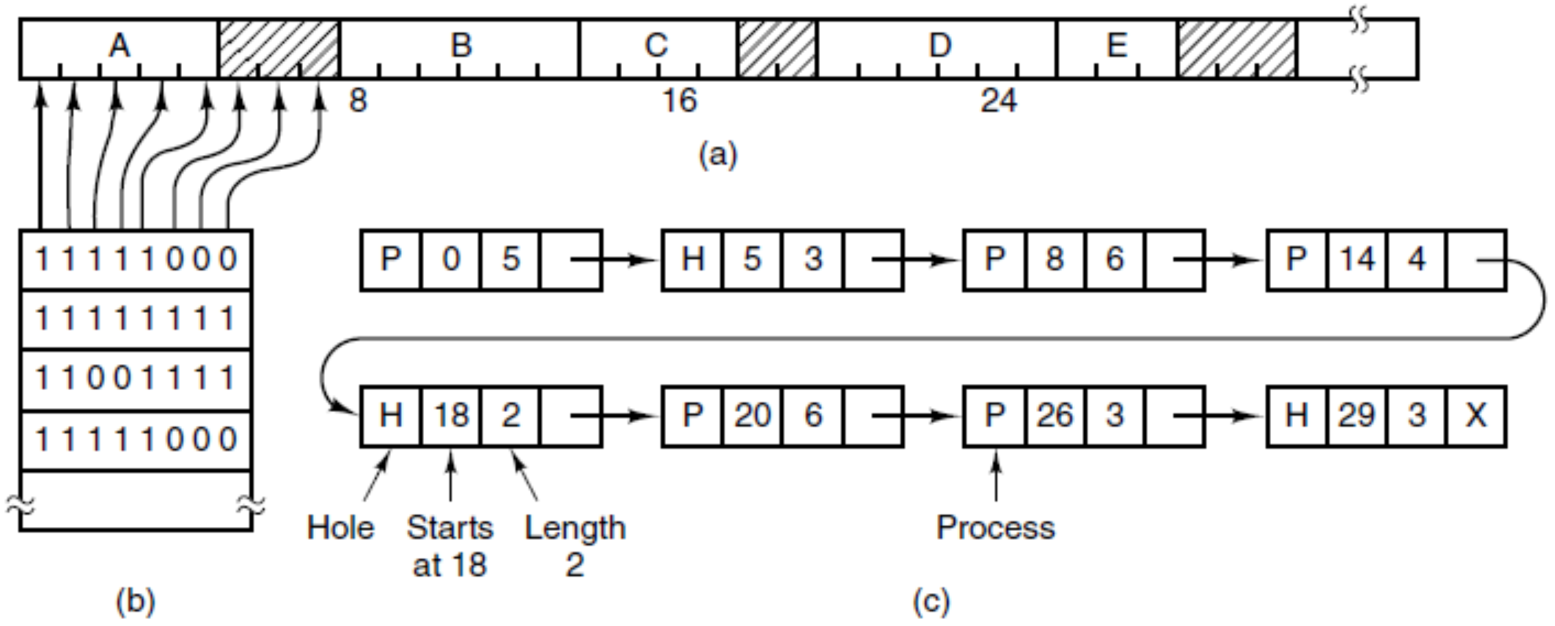
Maintain list for common sizes

Managing Free Memory

- » Two ways to track memory usage
  - » Bit maps
  - » Free lists

- » Chapter 10: Linux memory allocators such as buddy and slab.

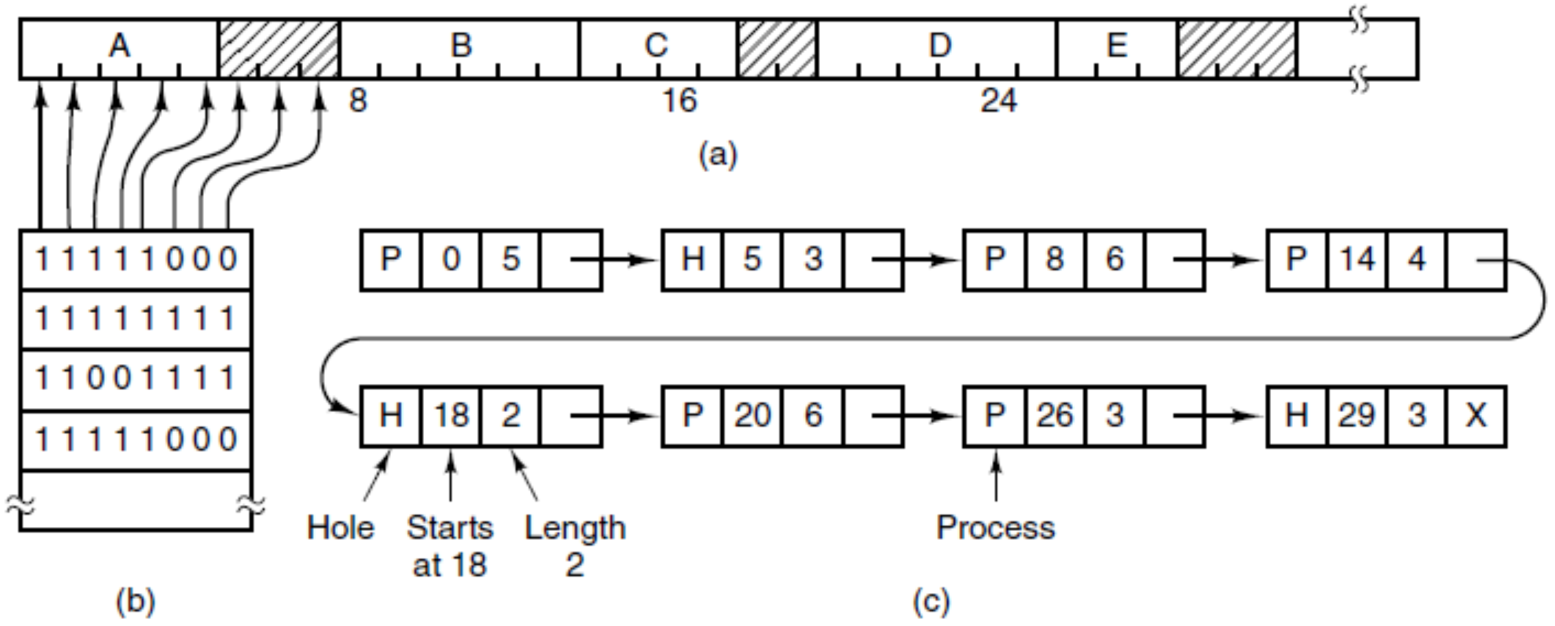
Bitmap



» With bitmap, memory divided into allocation units as small as a few words to a couple kilobytes.

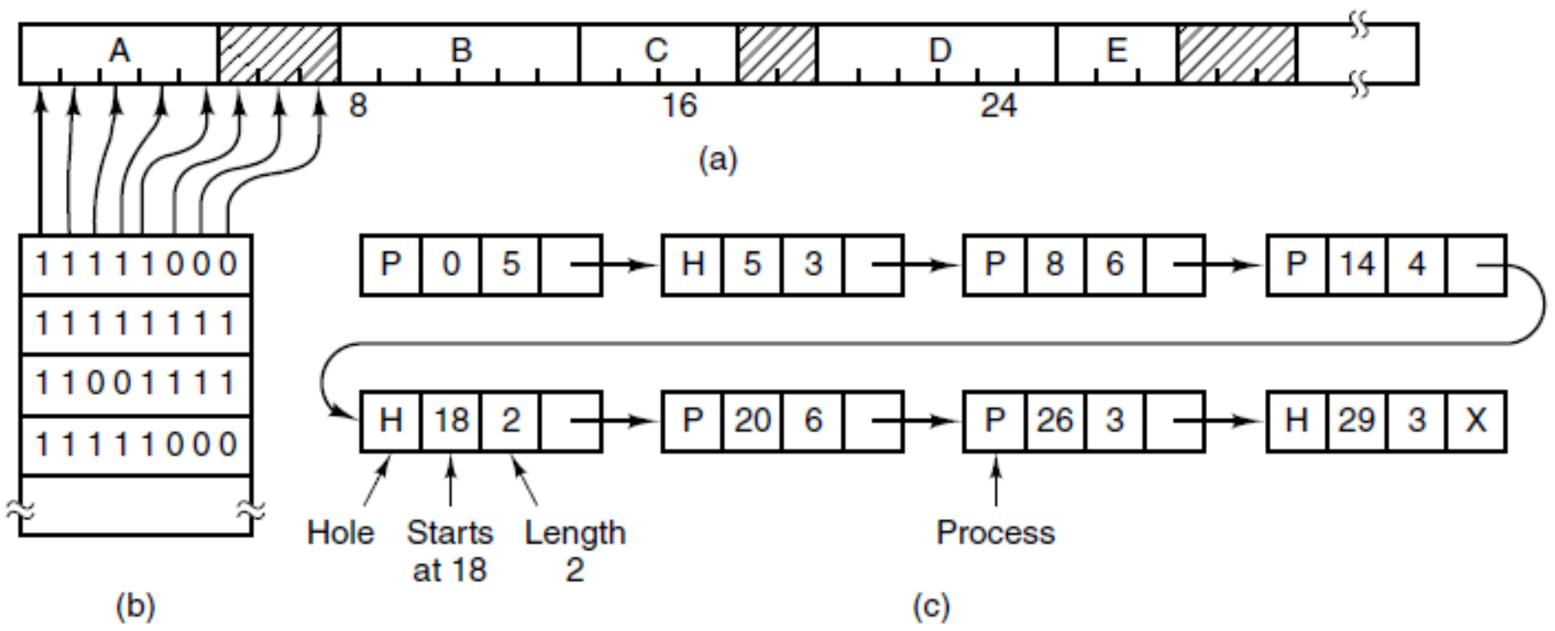
» Each allocation unit corresponds to a bit

Bitmap



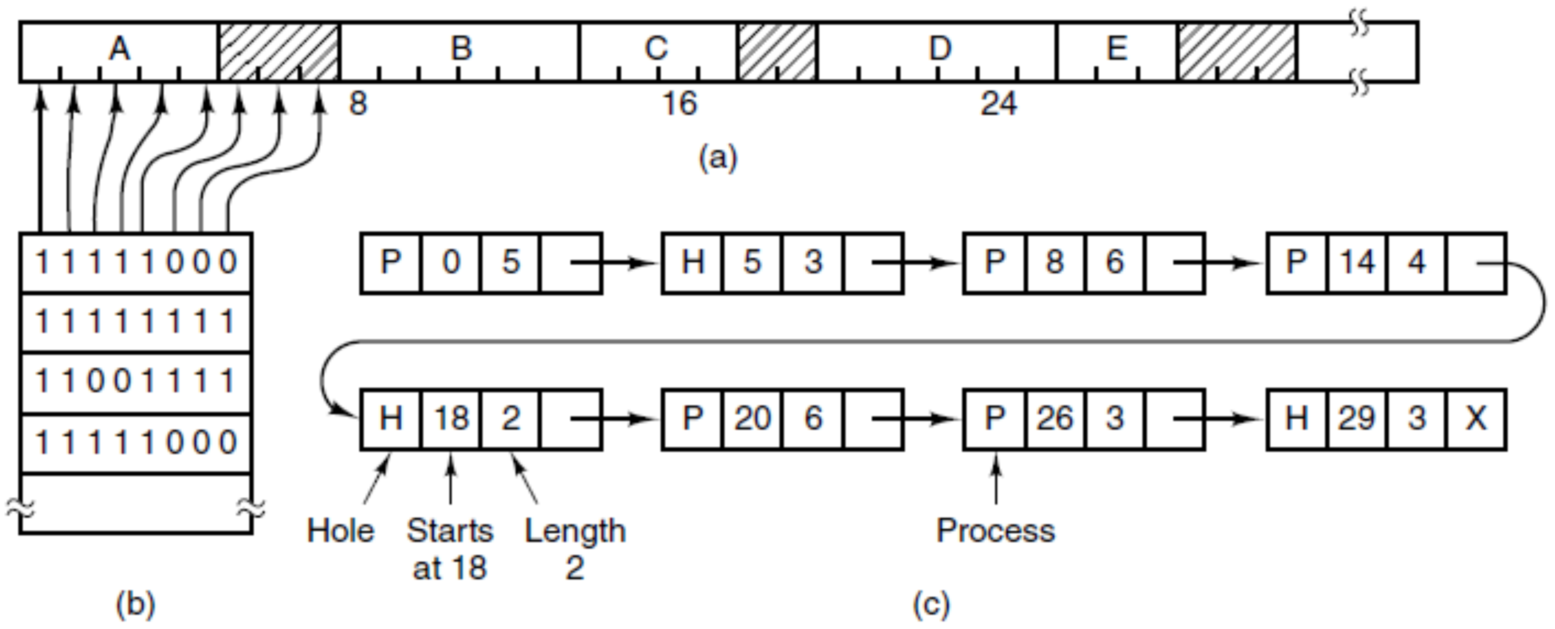
» Problem with bitmap: When bringing in K-units, the memory manager must search the bitmap for a run of k consecutive 0’s

Bitmap



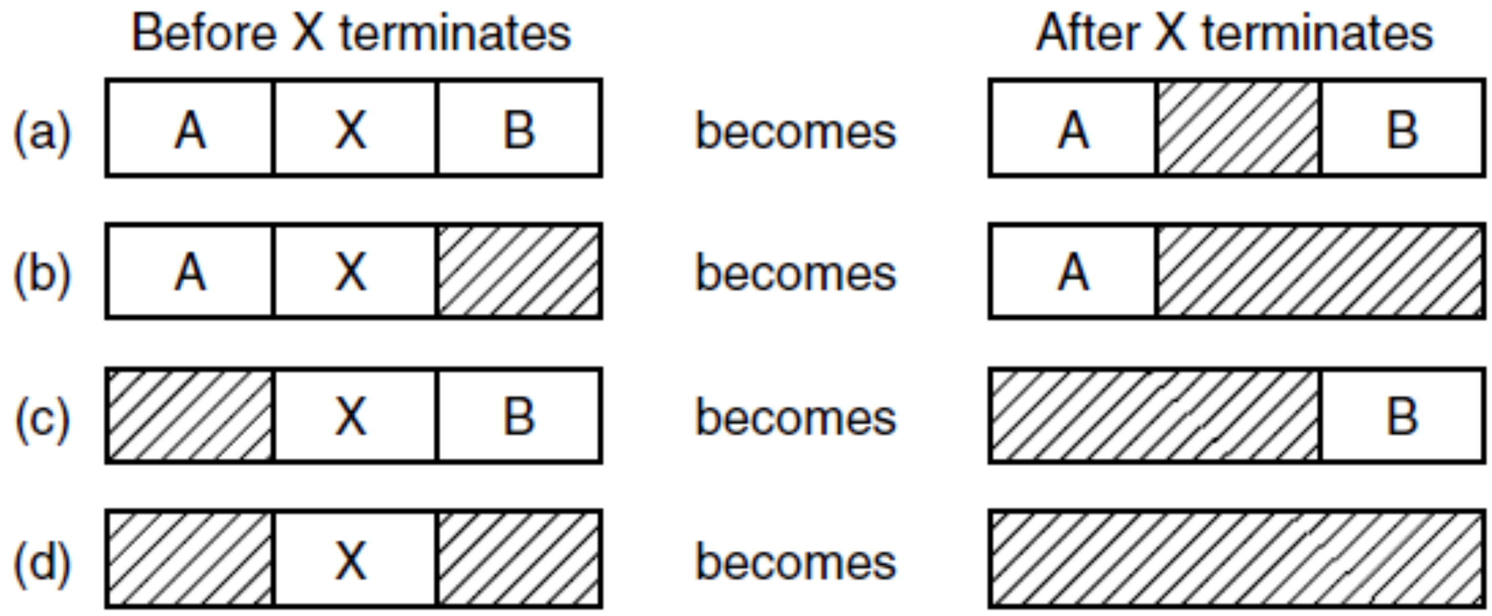
» Problem with bitmap: When bringing in K-units, the memory manager must search the bitmap for a run of k consecutive 0’s

Linked List



» Linked list: list of allocated and free memory segments.

Linked List


Before X terminates
After X terminates
(a)
A
X
B
becomes
A
B
(b)
A
X
becomes
A
(c)
X
B
becomes
B
(d)
X
becomes

Memory Allocation

- » Can be optimized by keeping separate lists for processes and holes.
- » If separate lists kept then the hole list should be sorted by size

What if the program size exceeds physical RAM?

Overlays

- » Memory wasn't always cheap. Could be a large part of total system price.
- » How do you write large programs for small memories?
- » init, main, finalize - common programming pattern
  - – Don't have to be in memory at the same time.

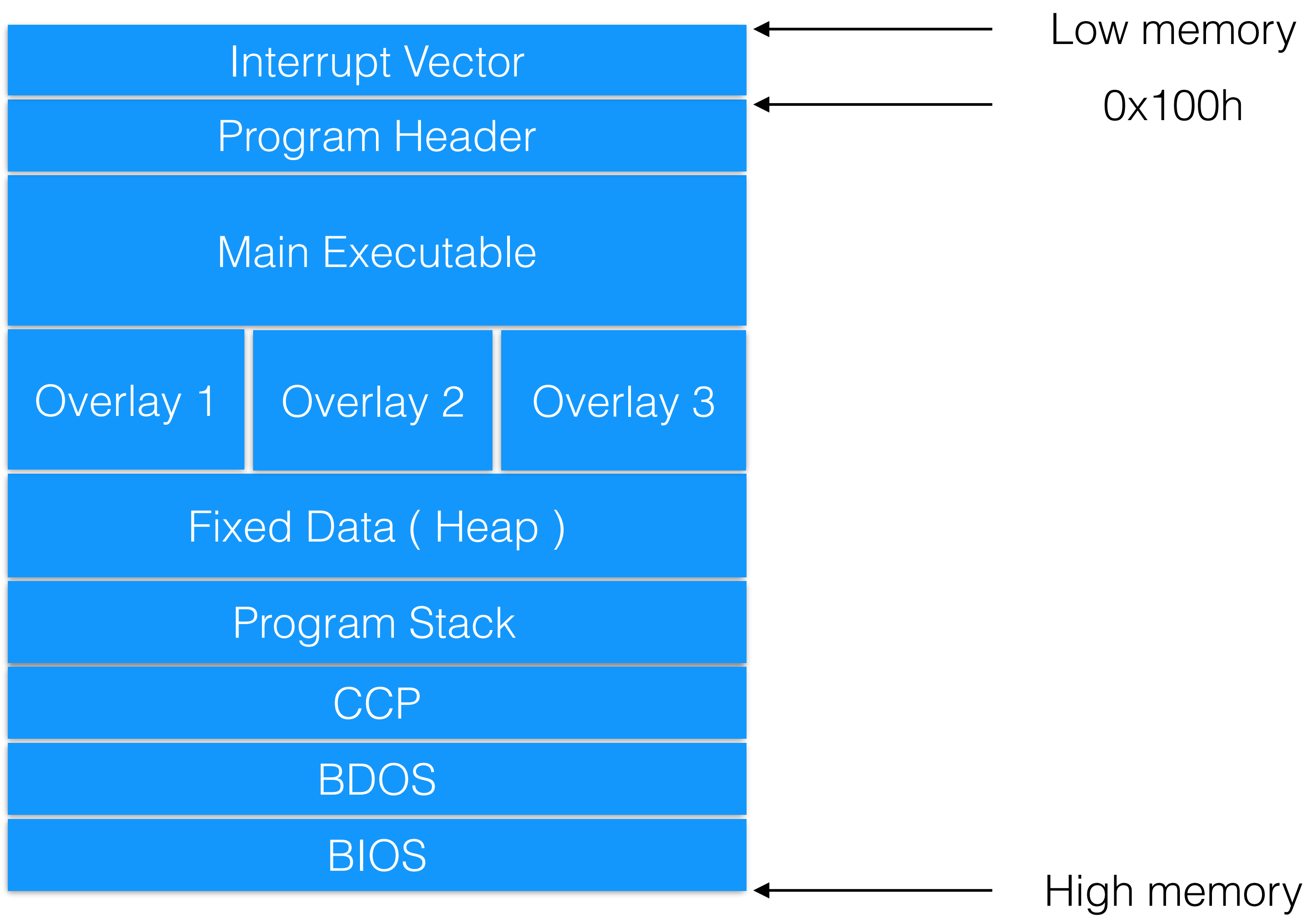
Overlays

» The programmer specifies the overlays

» An overlay can not make calls into another overlay.

» Actual loading of overlays into memory would be done by the runtime library

Program Overlay Layout



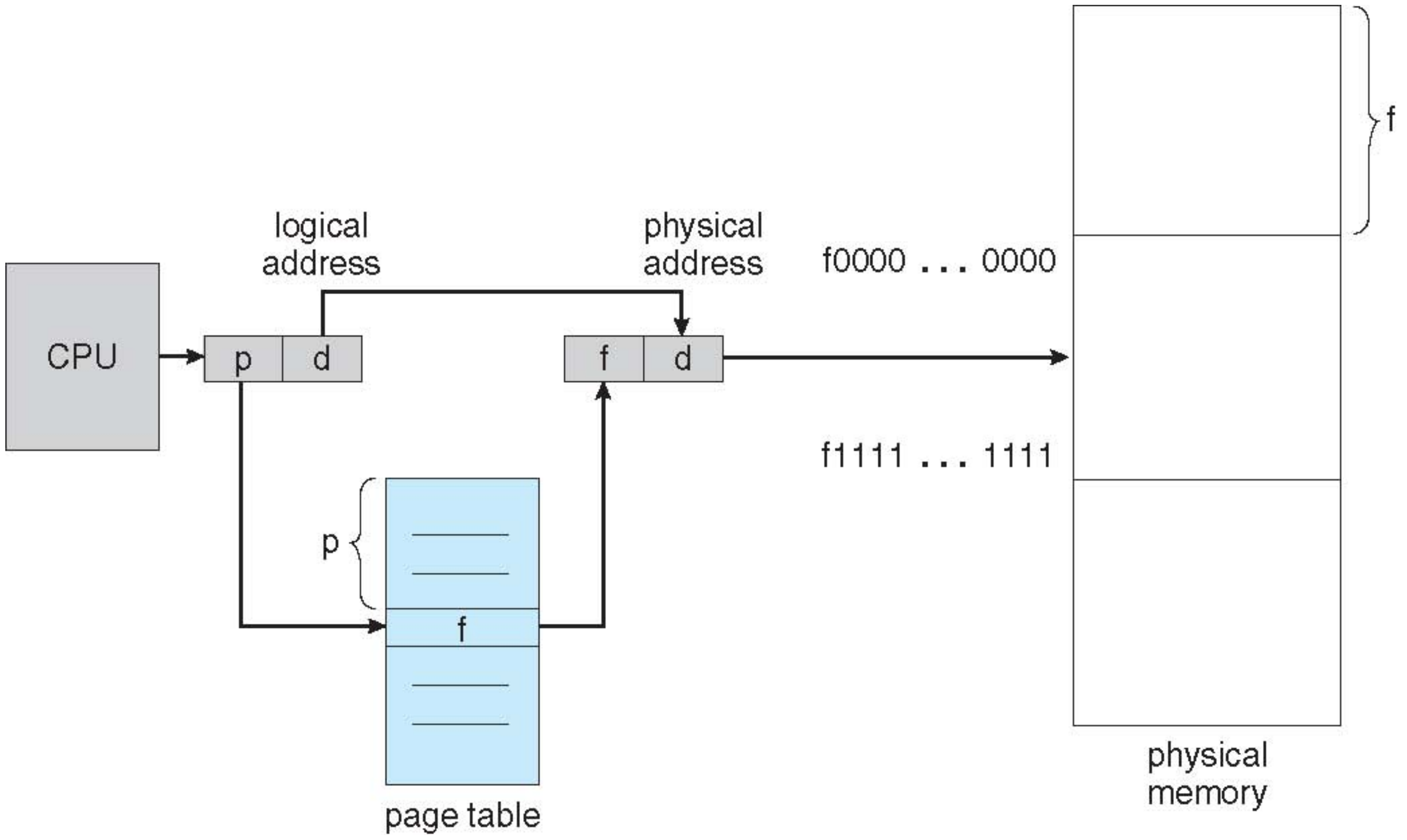
We need help, but why?

- » Multiprocessing causes external fragmentation
- » Compaction wastes resources
- » Solution – break RAM into fixed parts
- » Do not need to be contiguous
- » Map from logical to physical spaces
- » Need hardware help

Paging

- » Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- » Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8,192 bytes)
  - – Commonly 4 KB
- » Divide logical memory into blocks of same size called pages
- » Keep track of all free frames
- » To run a program of size n pages, need to find n free frames and load program
- » Set up a page table to translate logical to physical addresses
  - – Holds the map to the physical address
- » Internal fragmentation

Paging Hardware

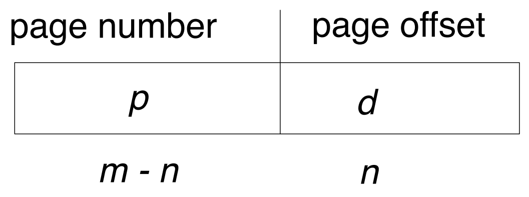


Address Translation Scheme

» Address generated by CPU is divided into:

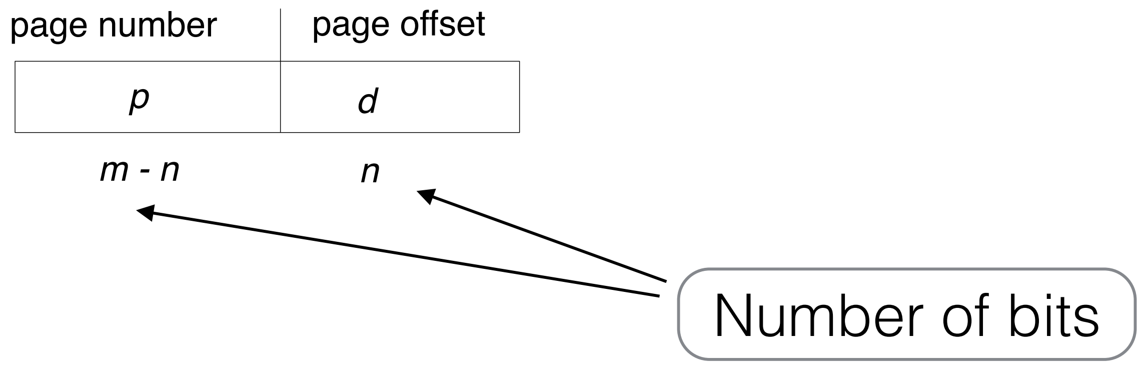
- **Page number (*p*)** – used as an index into a ***page table*** which contains base address of each page in physical memory
- **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

- For given logical address space 2^m and page size 2^n



Address Translation Scheme

- For given logical address space 2^m and page size 2^n



Address Translation Scheme

- For a system with 4 GB of addressable logical address space and a page size of 4 KB, the logical address is:

page number	page offset
p	*d*
m - n	*n*

Page size: 4096 = 2^{12}

Address space: 4294967296 = 2^{32}

page number	page offset
20	*12*

Address Translation Scheme

– Number of pages:

4294967296 / 4096 = 1048576

» And to verify our numbers:

$$1048576 = 2^{20}$$

page number	page offset
20	*12*

Address Translation Scheme

For a system with 12 bit addressing:

1. What is the maximum size of addressable real memory?

If the addresses are split into a page (6 bits) and an offset (displacement, 6 bits)

1. How large is a memory frame?
2. How many entries (maximum) is the page table?

Address Translation Scheme

For a system with 12 bit addressing:

1. What is the maximum size of addressable real memory?

2^{12} = 4096 bytes

Address Translation Scheme

If the addresses are split into a page (6 bits) and an offset (6 bits)

1. How large is a memory frame?

2^6=64 bytes

2. How many entries (maximum) is the page table?

2^6 = 64 pages

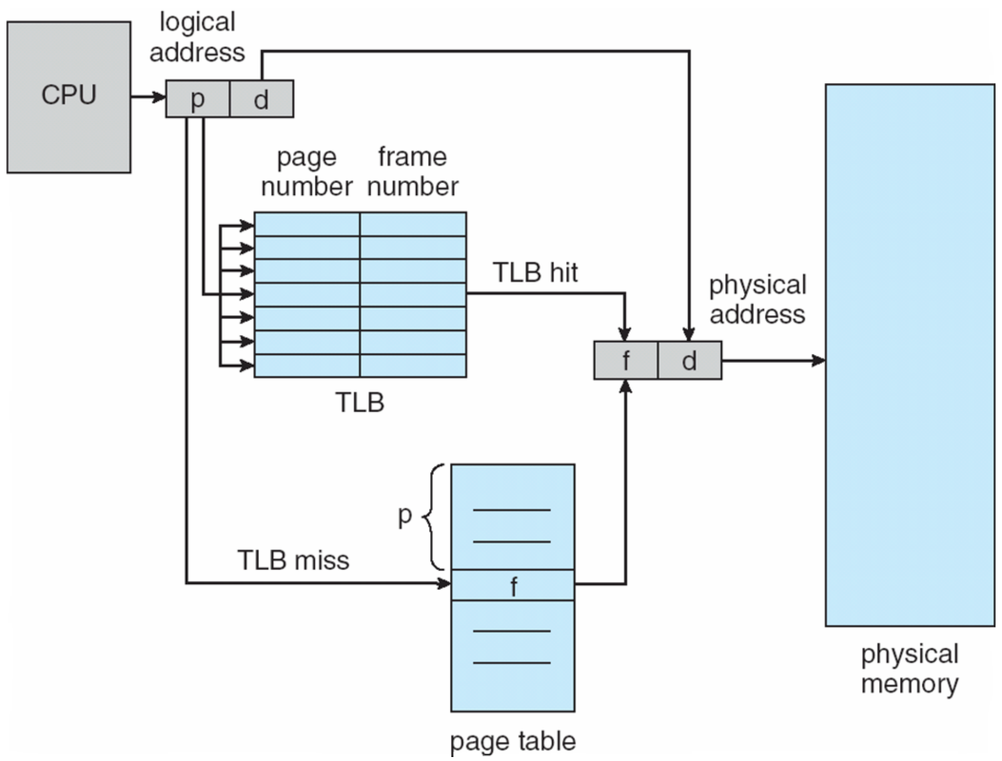
Implementation of Page Tables

» In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

  – Half speed

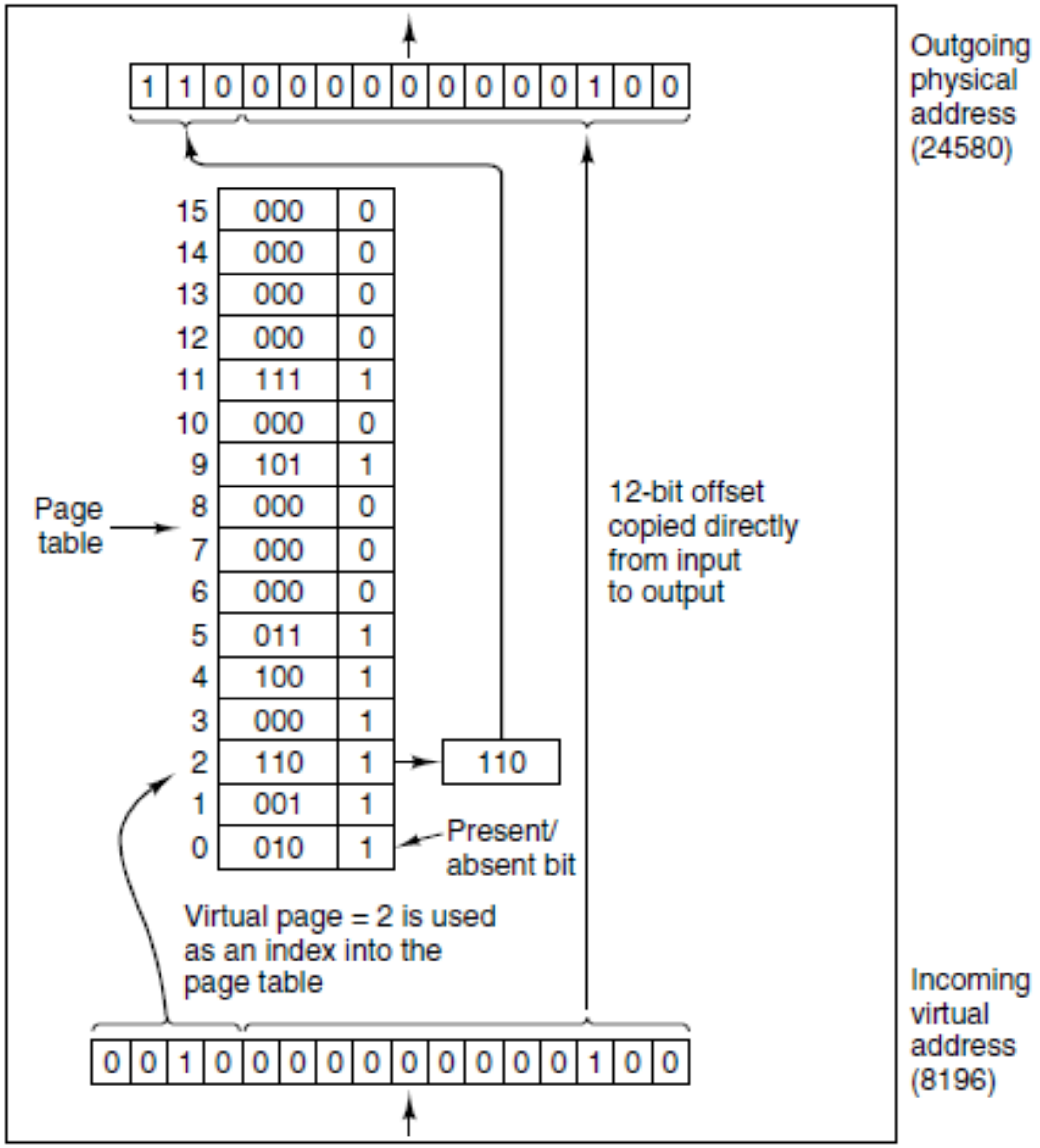
» The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)

Translation Lookaside Buffer



- All entries search in parallel
- Complex circuit so it’s small
- Rarely over 1000 entries

Inside the TLB



Implementation of Page Tables

- Nehalem has a two-level TLB:
  - The first level of TLB is shared between data and instructions.
    - The level 1 data TLB now stores 64 entries for small pages (4K) or 32 for large pages (2M/4M),
    - The level 1 instruction TLB stores 128 entries for small pages (the same as with Core 2) and seven for large pages.
    - The second level is a unified cache that can store up to 512 entries and operates only with small pages.

Effective Access Time

- »Associative Lookup = ε time unit
- »Assume memory cycle time = M
- »Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- »Hit ratio = α

Effective Access Time (EAT)

$$\text{EAT} = (M + \varepsilon)\,\alpha + (2M)(1 - \alpha)$$

2M because a miss requires us to get the frame number out of the page table (M cost) and get the normal memory reference (M cost)

Effective Access Time Example

»Associative Lookup = ε = 5 nanoseconds

»Assume memory cycle time is 100 nanoseconds

»Hit ratio = α = 80%

Effective Access Time (EAT)

EAT = (100 + 5) .8 + (200)(1 – .8)

= 124 nanoseconds

Effective Access Time

» First equation assumed TLB and Page Table lookups happened in parallel.

» If not:

Effective Access Time (EAT)

$$\mathrm{EAT} = (\mathrm{M} + \varepsilon)\,\alpha + (2\mathrm{M} + \varepsilon)(1 - \alpha)$$

Context Switches

- » Most systems TLB doesn’t concern itself with which process is running.
  - – Context switch means TLB must be flushed.
  - – First few memory accesses of the new process will not get any TLB hits so the process will run at half speed for memory references.
    - • Threads vs. processes
- » Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Demand Paging

- » Our previous methods had one requirement:
  - – The instructions being executed must be in physical memory.
  - – The first approach to meeting this requirement is to place the entire logical address space in physical memory.
  - – Dynamic loading can help to ease this restriction
    - • Requires special precautions and extra work by the programmer.

Demand Paging

- Up until now we have assumed all programs fit fully into memory before they execute.
- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
- One major advantage of this scheme is that programs can be larger than physical memory.

Demand Paging

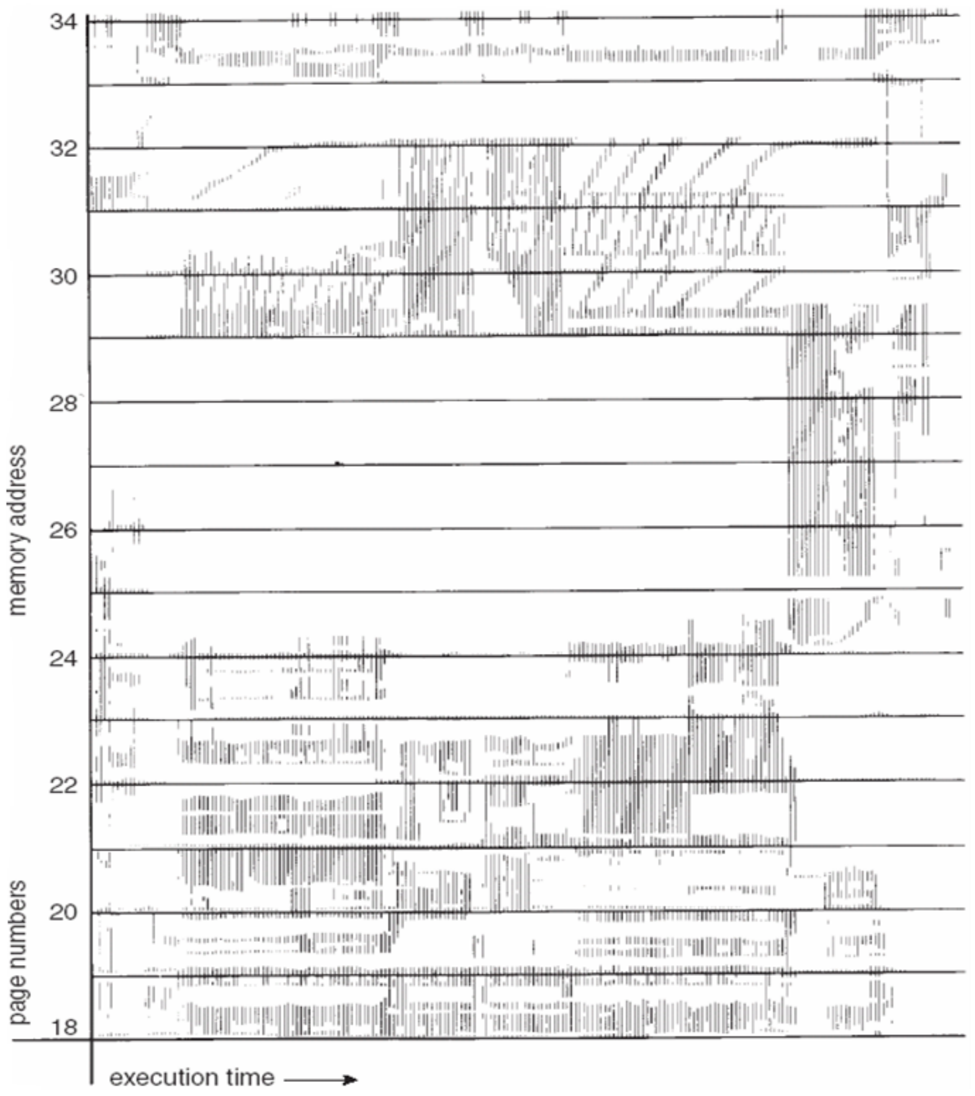
- » The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable;
  - – Also unfortunate, since it limits the size of a program to the size of physical memory.
- » In fact, an examination of real programs shows us that, in many cases, the entire program is not needed. For instance, consider the following:
  - – Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
  - – Arrays,lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements. An assembler symbol table may have room for 3,000 symbols, although the average program has less than 200 symbols.
  - – Certain options and features of a program may be used rarely

Demand Paging

» Even in those cases where the entire program is needed, it may not all be needed at the same time.

– Locality of Reference - Only a few pages of memory are accessed by the process in any given time slot.

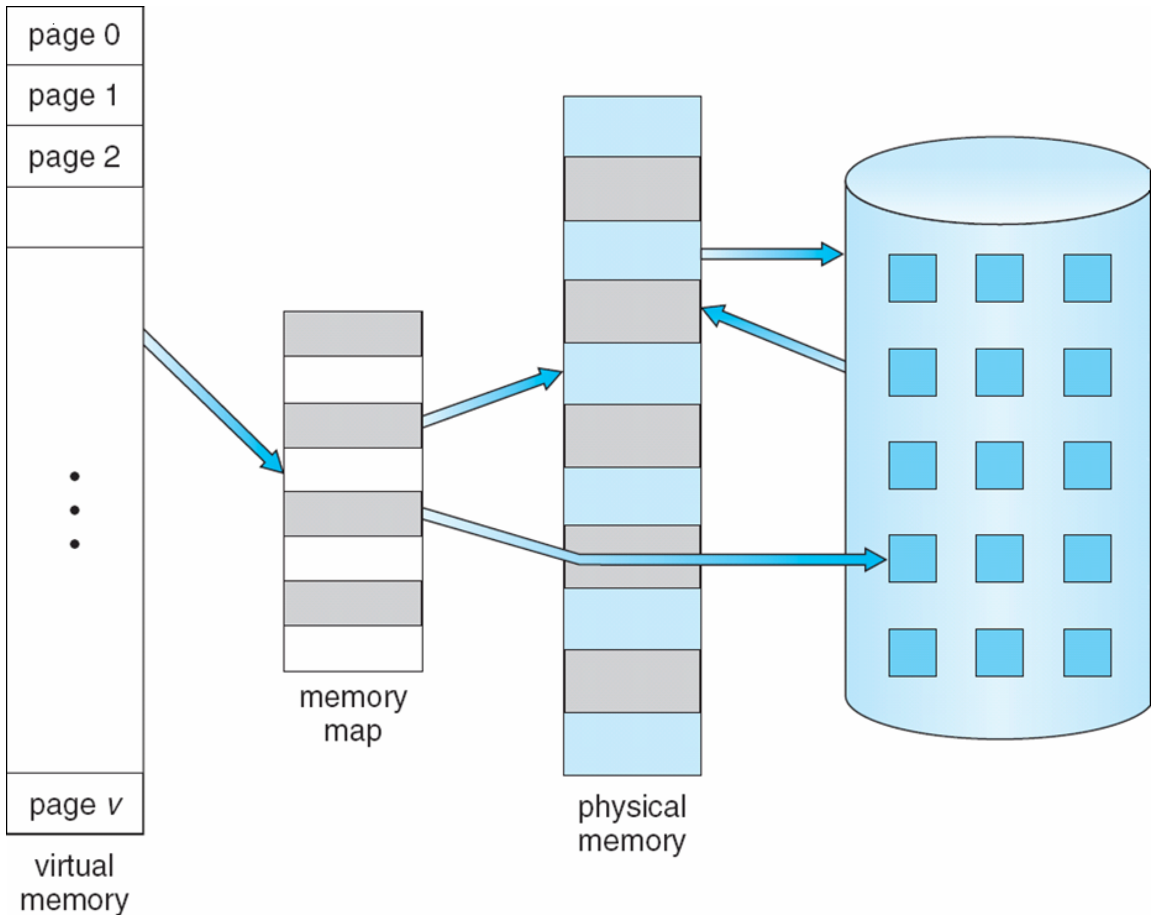
Locality In A Memory-Reference Pattern



Demand Paging

- » The ability to execute a program that is only partially in memory would confer many benefits:
  - A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large virtual address space, simplifying the programming task.
  - Because each user program could take less physical memory, more programs could be run at the sance time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
  - Less I/O would be needed to load or swap user programs into memory, so each user program would run faster.

Virtual Memory



Demand Paging

- » Bring a page into memory only when it is needed
  - – Less I/O needed
  - – Less memory needed
  - – Faster response
  - – More users

Demand Paging

» Page is needed $\Rightarrow$ reference to it

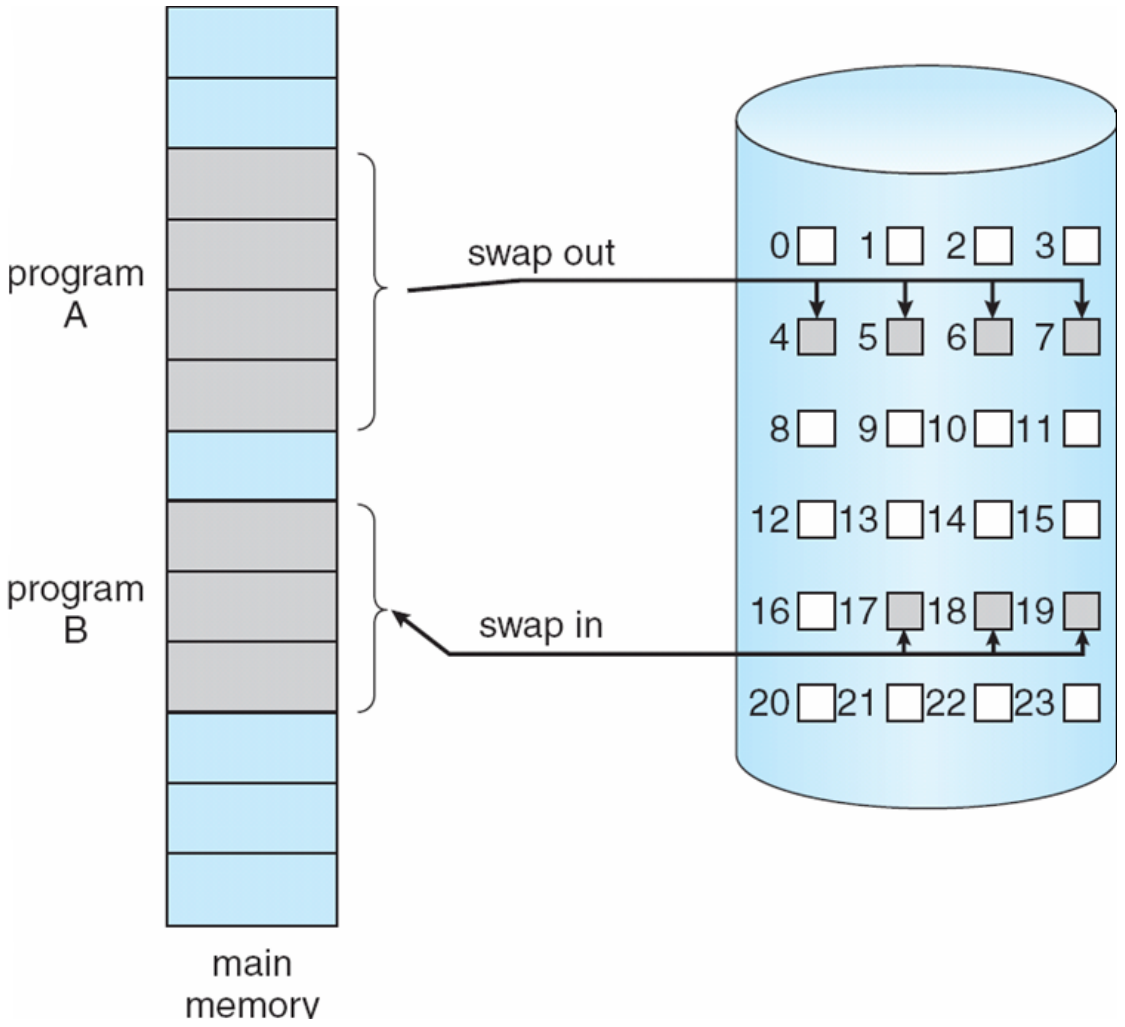
» invalid reference $\Rightarrow$ abort

» not-in-memory $\Rightarrow$ bring to memory

» Lazy loading – never swaps a page into memory unless page will be needed

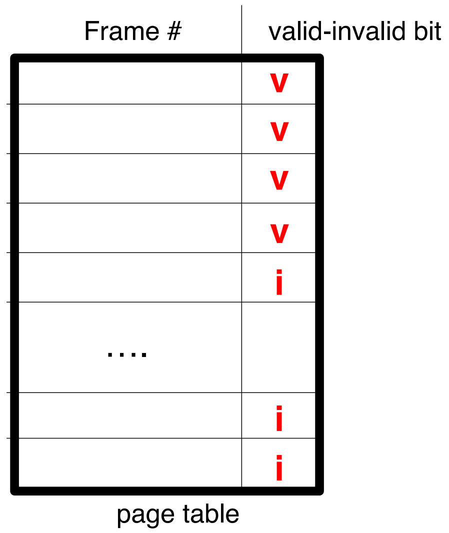
» Swapper that deals with pages is a pager

Demand Paging

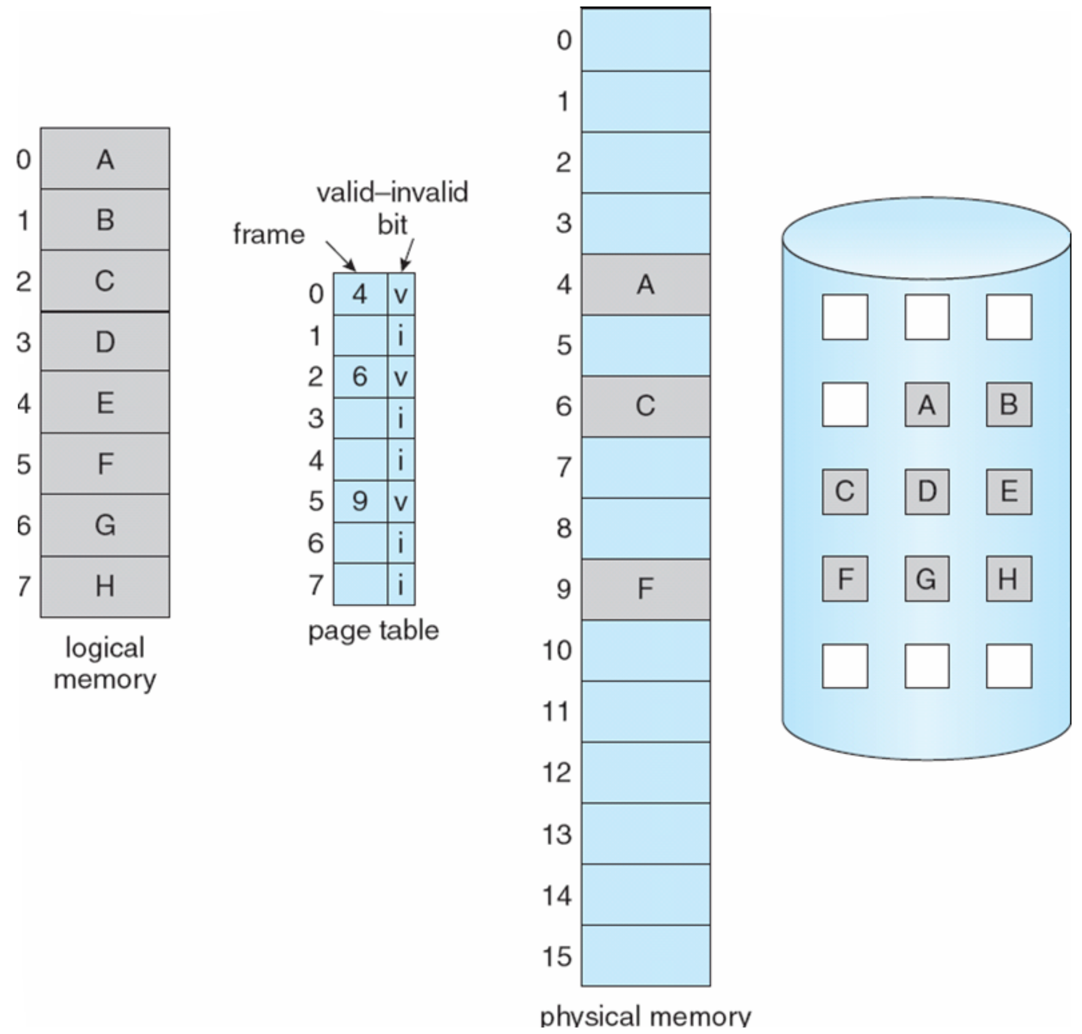


Demand Paging

- With each page table entry a valid–invalid bit is associated ($v \Rightarrow$ in-memory, $i \Rightarrow$ not-in-memory)
  - Present / Absent in book parlance
- Initially valid–invalid bit is set to i on all entries
- Example of a page table snapshot:



Demand Paging



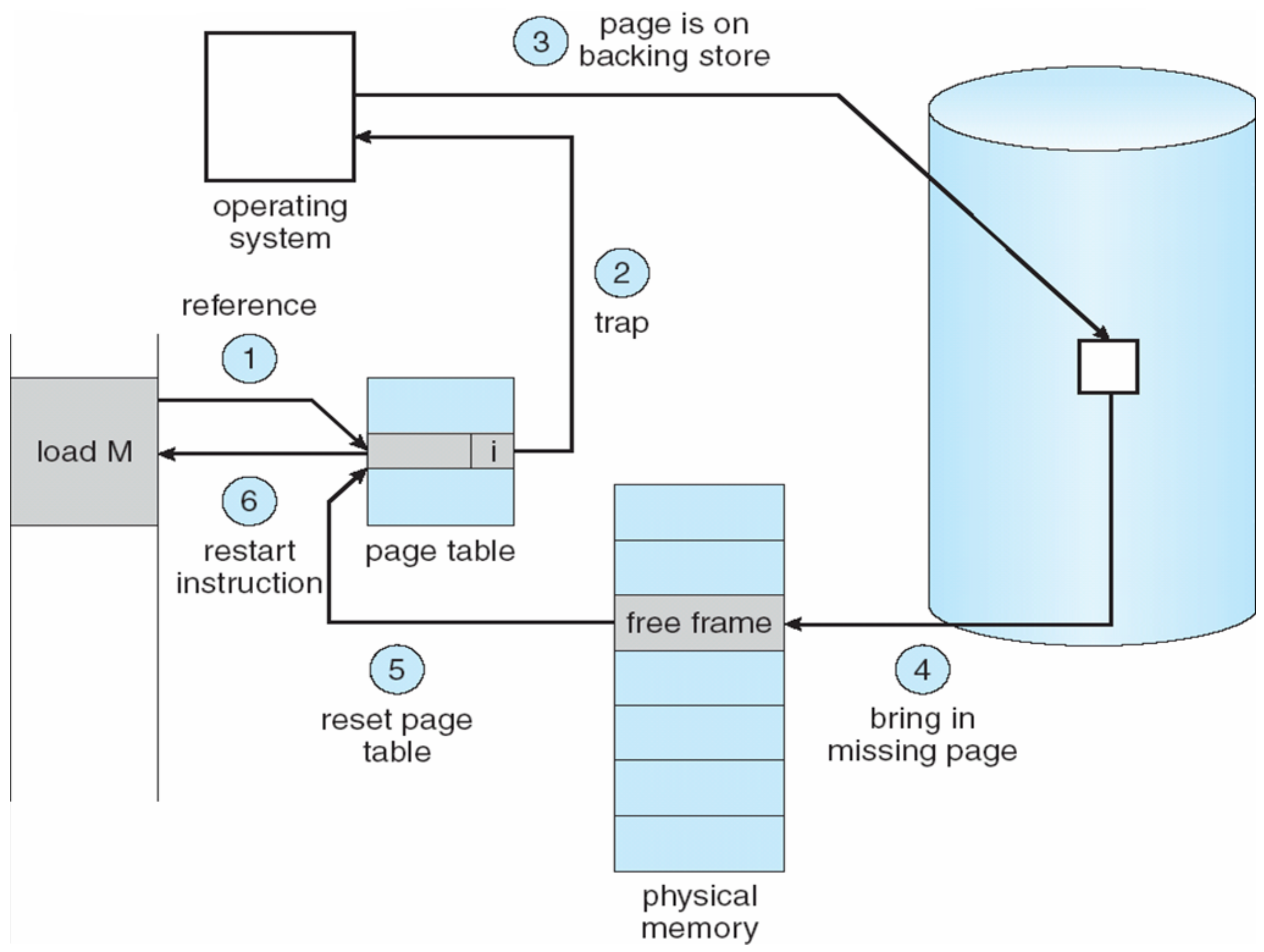
Demand Paging

» If there is a reference to a page, first reference to that page will trap to operating system:

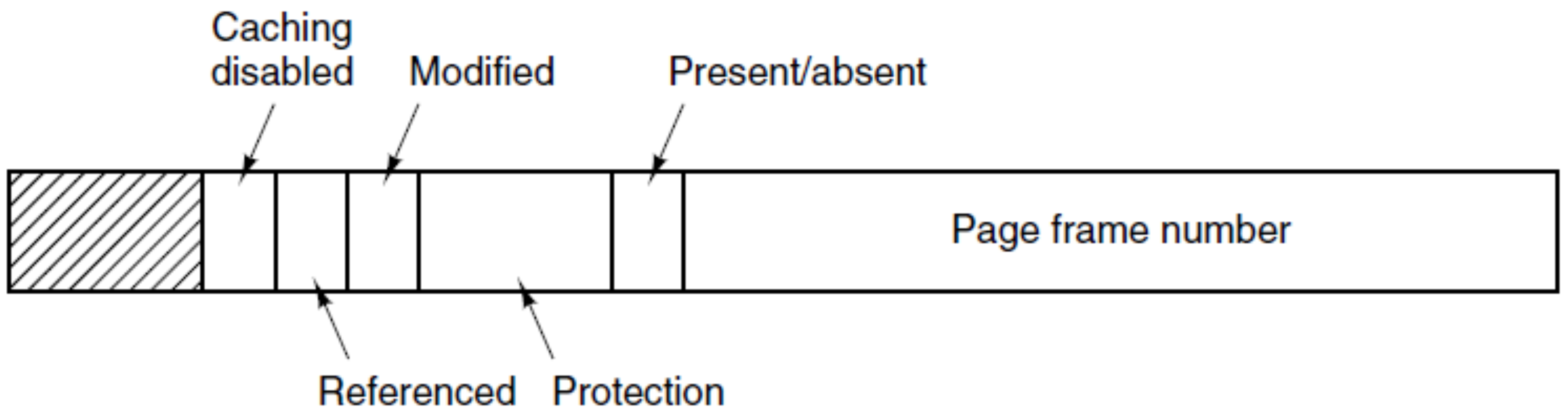
» page fault

1. Operating system looks at another table to decide:
    1. Invalid reference $\Rightarrow$ abort
    2. Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = v

Demand Paging



Page Table Entry



Types of Page Faults

» Soft miss: 10-20 instructions

» Hard miss: several milliseconds

» Minor page fault

» Major page fault

» Segmentation fault

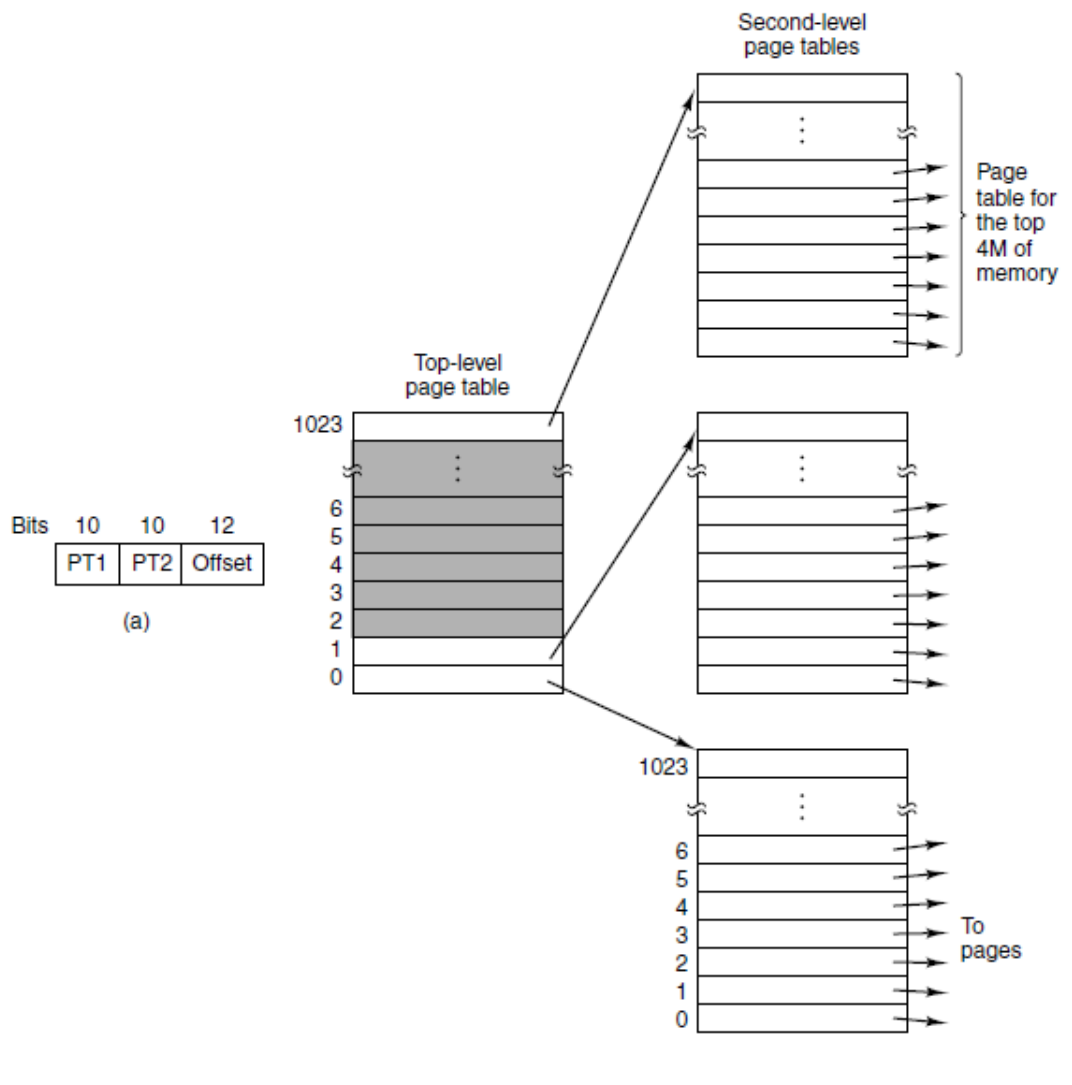
Multilevel Page Tables

» Single page table contains one entry per virtual page per process

  » 4 MB per process. 2^{32} / 4096 * 4

» Multilevel page tables avoid keeping all page tables in memory at a time.

Multilevel Page Tables



(b)

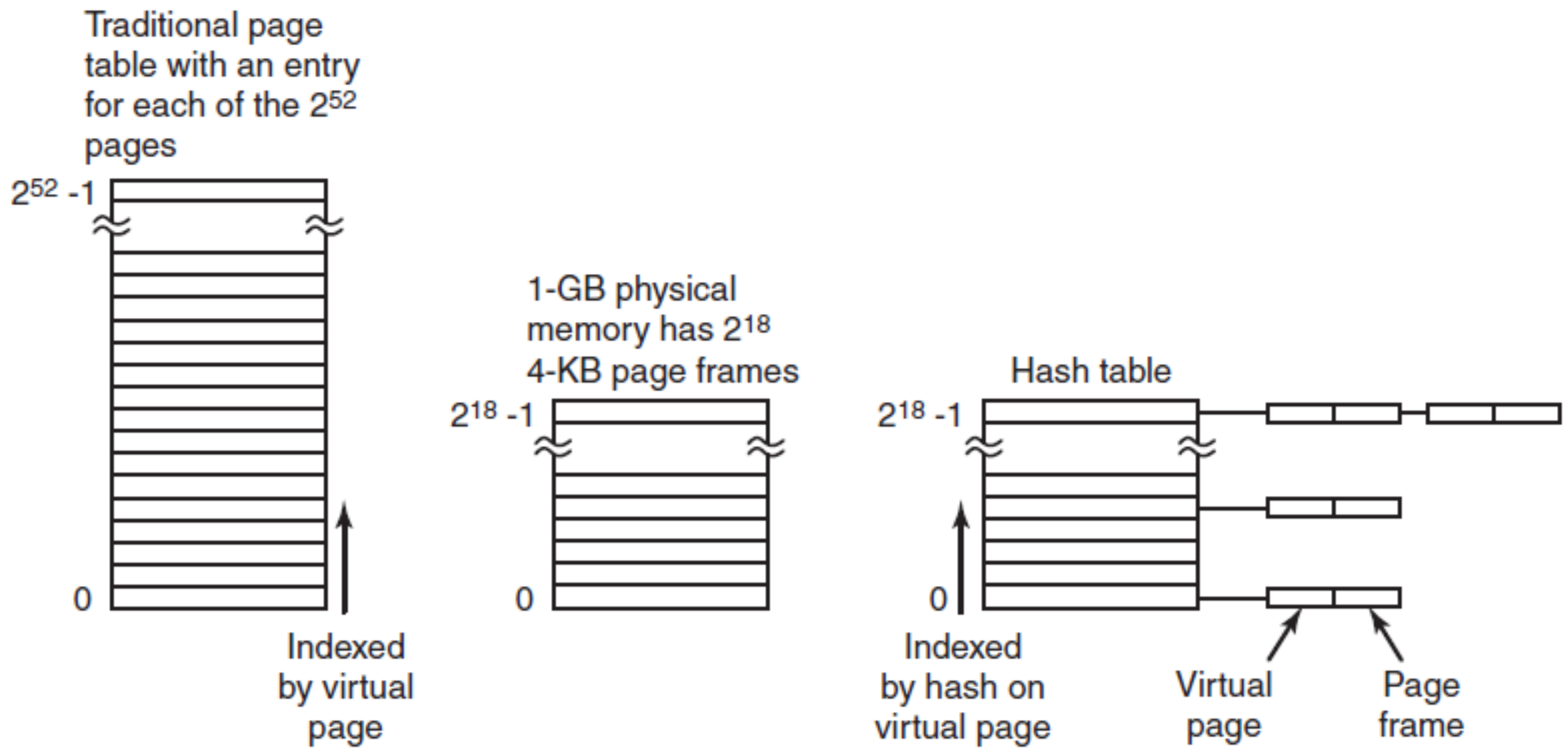
Multilevel Page Tables

- » 80386 addressed 4 GB of memory using a two-level page table.
  - » Page directory -> page tables -> frames
- » Pentium Pro
  - » Page directory pointer table
    - » 64 bit so it could address over 4GB
    - » Page Map Level 4
    - » $2^{48} = 256$ TB

Inverted Page Tables

- First used on PowerPC, UltraSPARC and Itanium
- 64bit page tables are large
  - 4MB page and 64 bit virtual addresses mean 2^{42} page table entries
- Instead of an entry per page of virtual address pace, one entry per page frame of real memory

Inverted Page Tables


Traditional page table with an entry for each of the 2^{52} pages
2^{52} -1
0
Indexed by virtual page
1-GB physical memory has 2^{18} 4-KB page frames
2^{18} -1
0
Hash table
2^{18} -1
0
Indexed by hash on virtual page
Virtual page
Page frame

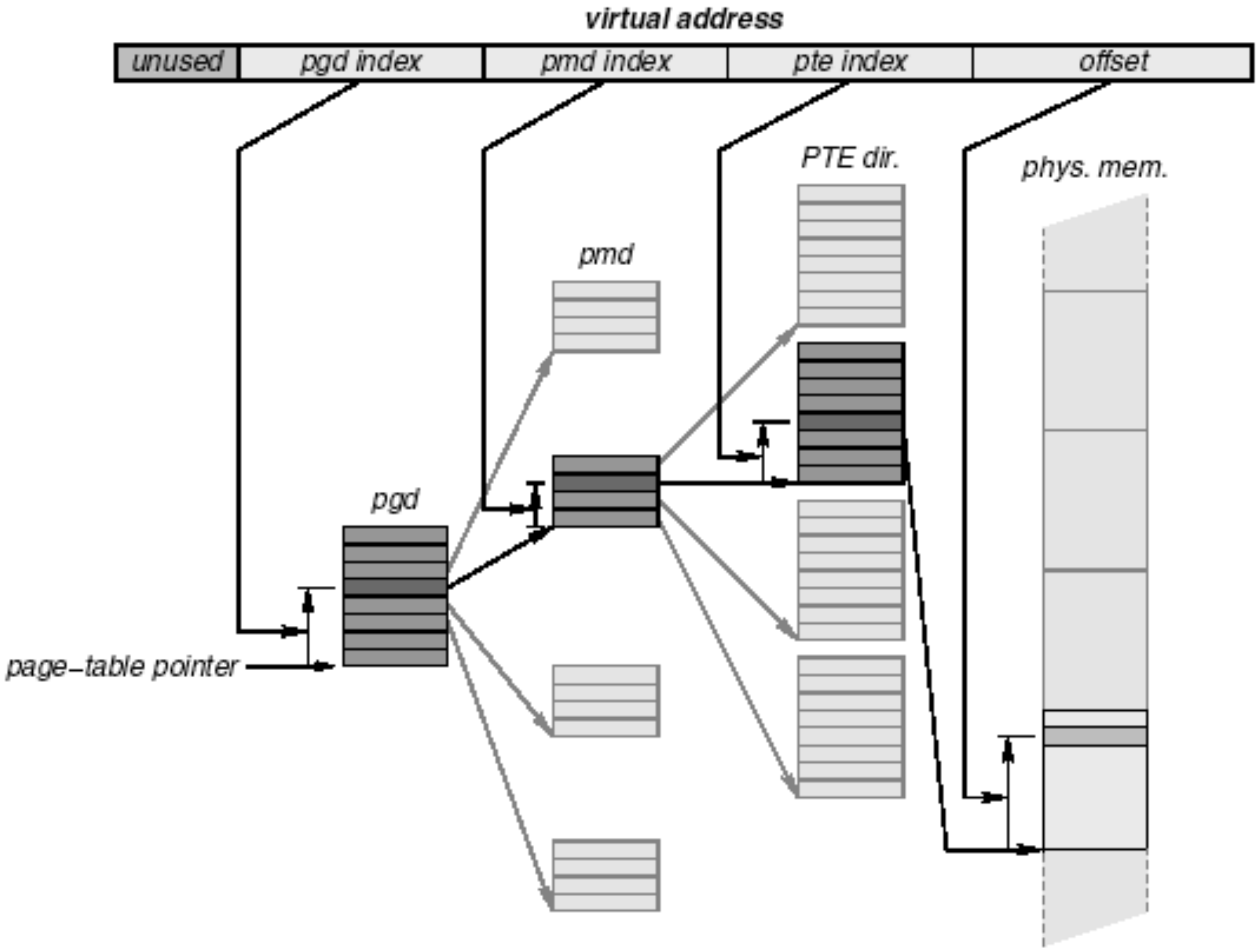
Inverted Page Tables

- » 64-bit virtual addresses and 4 KB page size with 4 GB of RAM, inverted page table requires on 1,048,476 entries.
- » Each entry tracks (virtual address, process) pair
- » Saves a lots of space, but big down side
  - » Virtual -> Physical mapping much harder
  - » Every memory reference must search the page table not just on page faults

Inverted Page Tables

- » TLB can help by holding heavily used pages.
- » On TLB miss, still have to search the inverted table
  - » Feasible way to accomplish is to hash on the virtual address.
  - » If the hash table has as many slots as the machine has physical pages, the bucket will hold a single frame

Linux Page Table



Page Replacement

- What if we are out of frames?
    - Have to discard one
    - May have pages no longer used
- How to identify unneeded?

Working Set

- Can’t assume all the pages we need will be in memory when we need them.
- Working set - pages a process is referencing in a short period, e.g the set of pages a process is currently using
- Sliding window - fixed interval over which the working set is measured.
- Page replacement - removing a page we may not need anymore.

Working Set

Page reference string:

1 2 1 5 7 1 6 3 7 1 6 4 2 7

We never reference page 5 again after step 4

Working set	Event Number
1	1
12	2
12	3
125	4
1257	5
1257	6
1567	7
1367	8
1367	9
1367	10
1367	11
1467	12
1246	13
2467	14

How about removing 5 after it falls out of the 4 step sliding window?

Working Set

Page reference string:

1 2 1 5 7 1 6 3 7 1 6 4 2 7

2 is not referenced between steps 2 and 13

Working set	Event Number
1	1
12	2
12	3
125	4
1257	5
1257	6
1567	7
1367	8
1367	9
1367	10
1367	11
1467	12
1246	13
2467	14

How about removing 5 after it falls out of the 4 step sliding window?

Basic Page Replacement

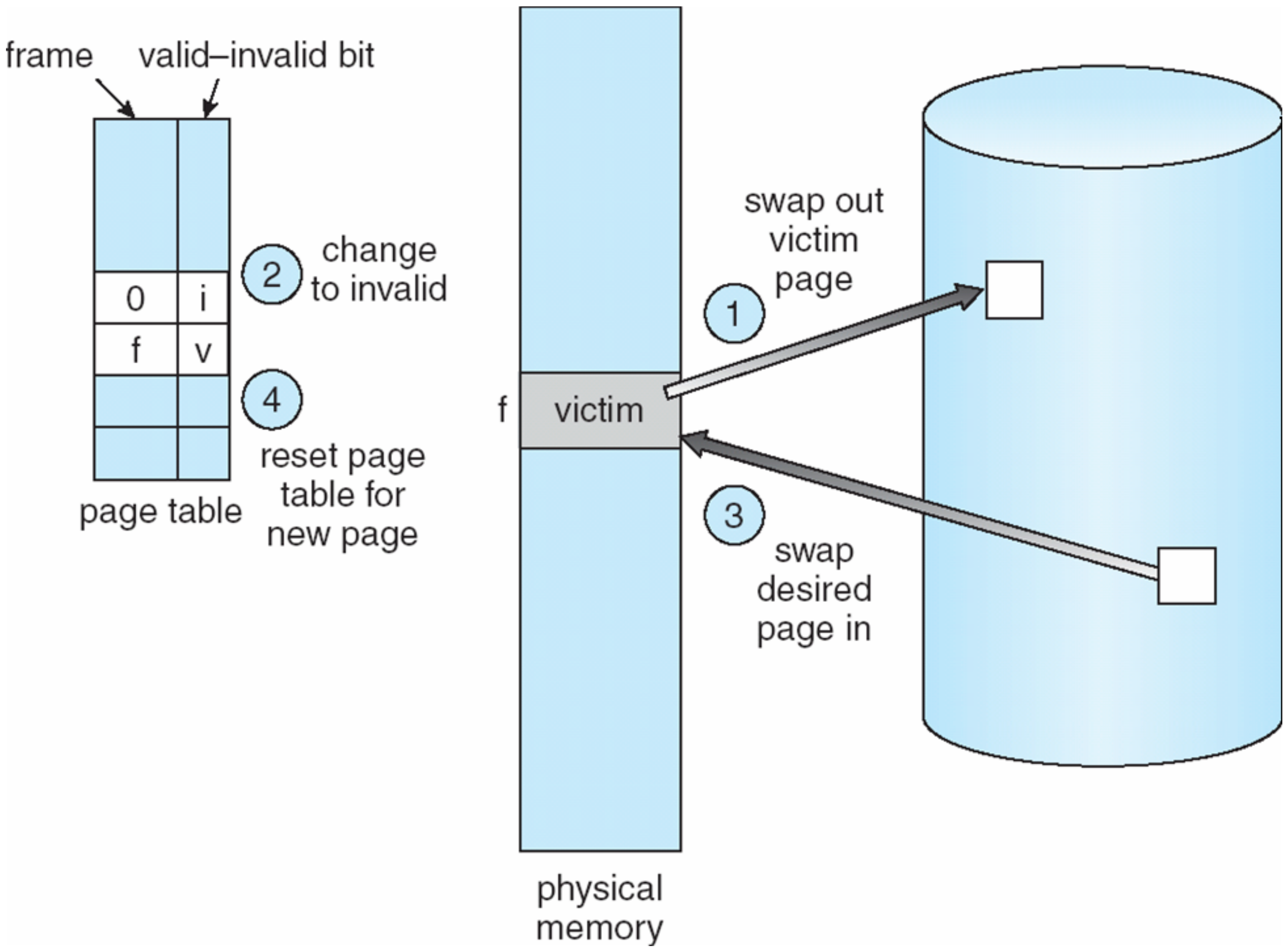
1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement
   algorithm to select a **victim** frame

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Restart the process

Page Replacement



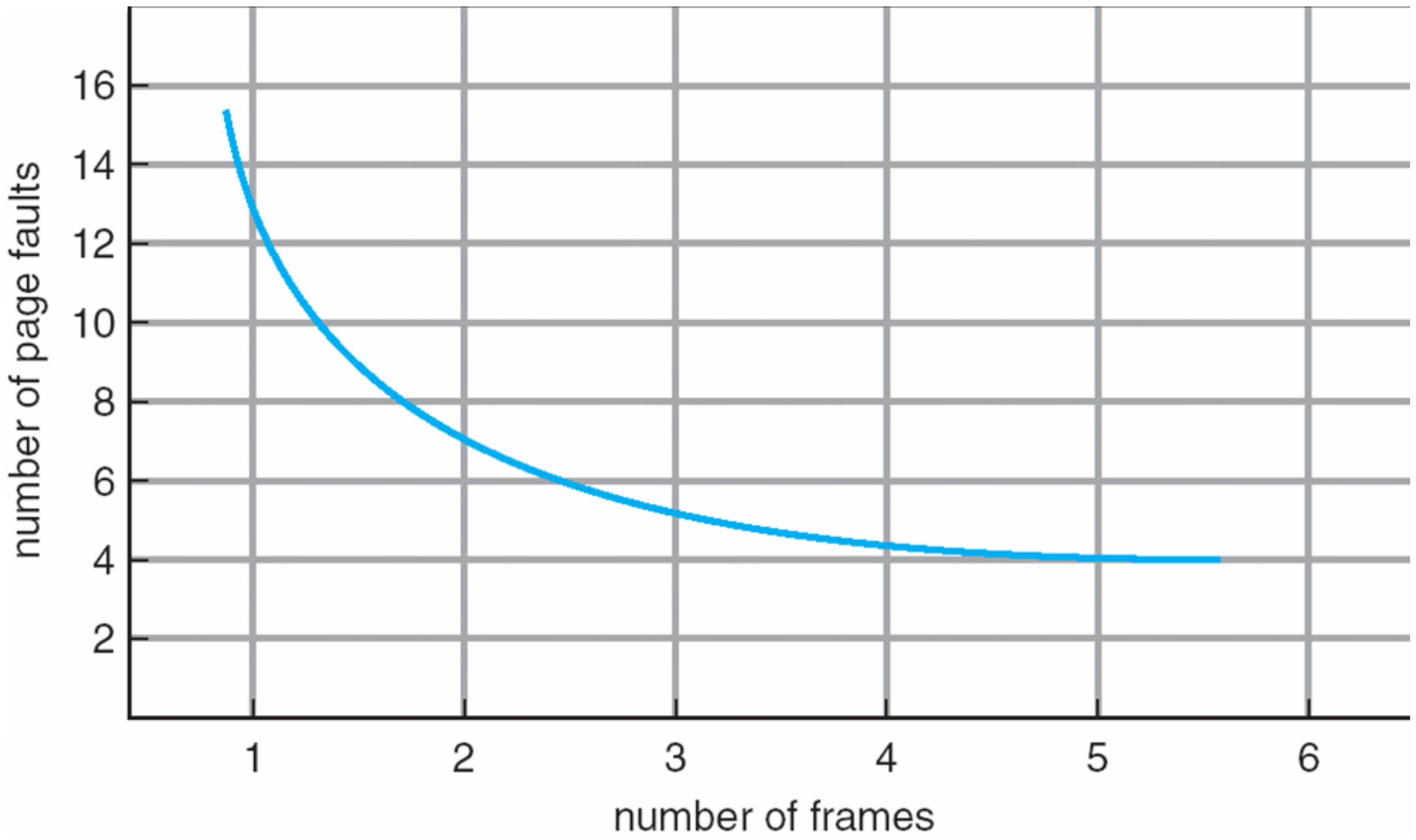
Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

First In, First Out

- Eject the oldest page.
- Very low overhead
- Easy to implement
- Poor performance and erratic behavior.
    - Can end up evicting the most frequently used page
- Subject to Bélády's anomaly

Graph of Page Faults Versus The Number of Frames



Bélády’s anomaly

- Increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern.

Page Requests	3	2	1	0	3	2	4	3	2	1	0	4
Newest Page	3	2	1	0	3	2	4	4	4	1	0	0
		3	2	1	0	3	2	2	2	4	1	1
Oldest Page			3	2	1	0	3	3	3	2	4	4

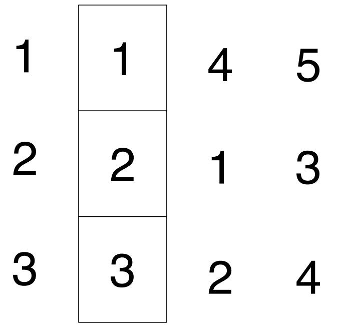
3 page frames
9 faults (red)

Page Requests	3	2	1	0	3	2	4	3	2	1	0	4
Newest Page	3	2	1	0	0	0	4	3	2	1	0	4
		3	2	1	1	1	0	4	3	2	1	0
			3	2	2	2	1	0	4	3	2	1
Oldest Page				3	3	3	2	1	0	4	3	2

4 page frames
10 faults (red)

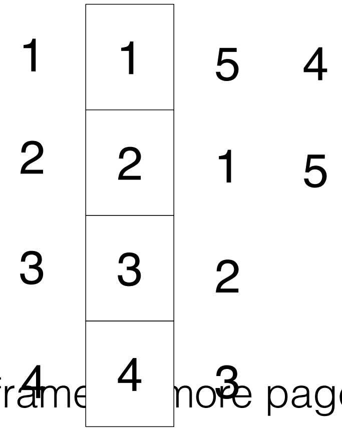
First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)



9 page faults

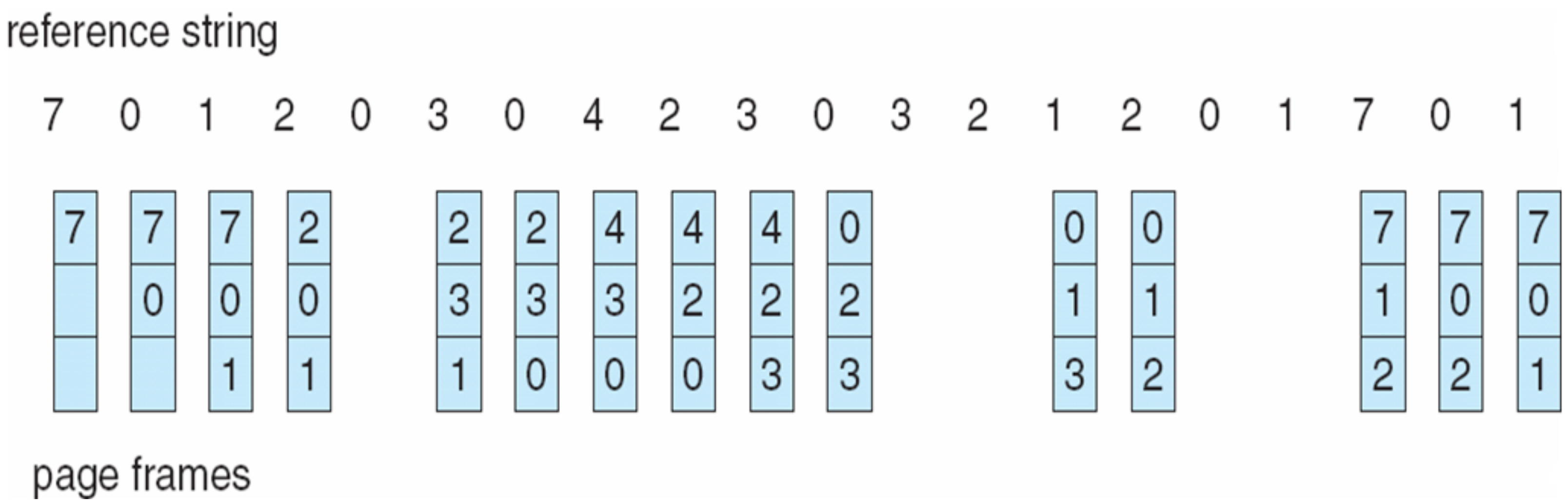
- 4 frames



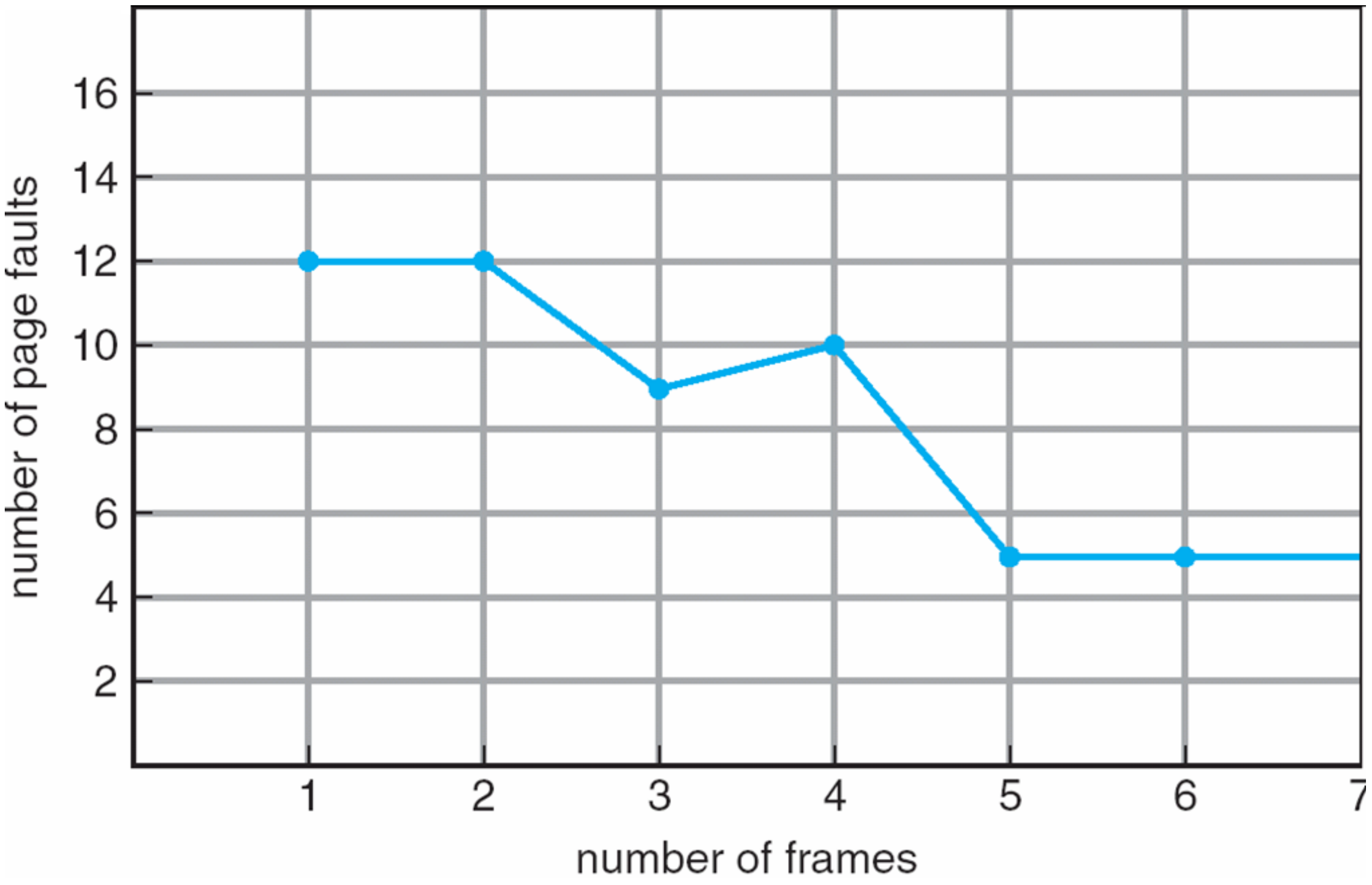
10 page faults

- Belady’s Anomaly: more frame more page faults

FIFO Page Replacement



FIFO Illustrating Belady’s Anomaly

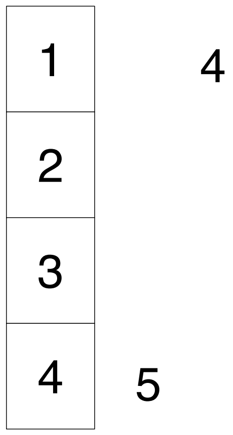


Optimal Algorithm

Replace page that will not be used for longest period of time

4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

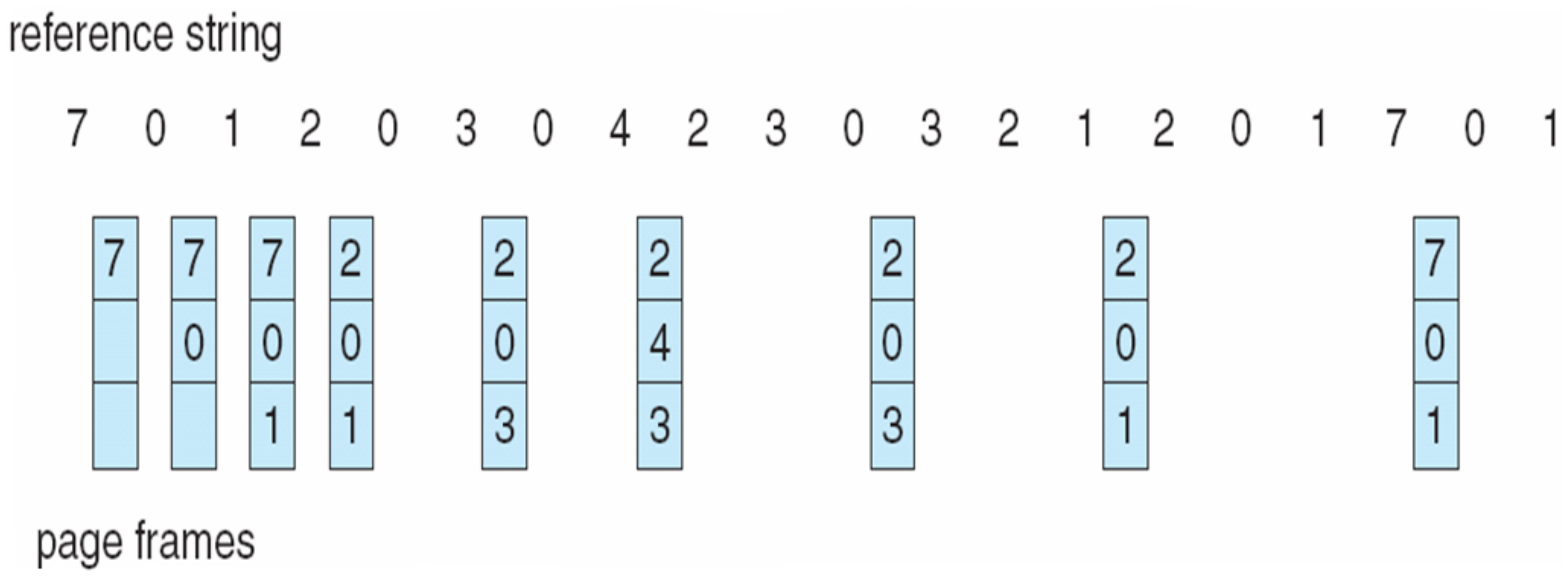


6 page faults

How do you know this?

Used for measuring how well your algorithm performs against a best-case

Optimal Page Replacement

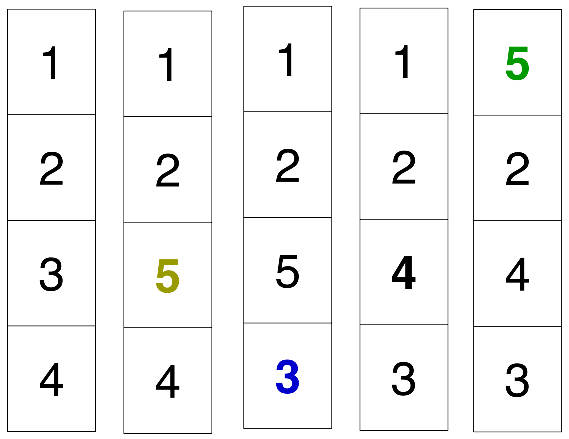


Optimal

- Unrealizable
- Have to be able to read the future
- Provides a best-case

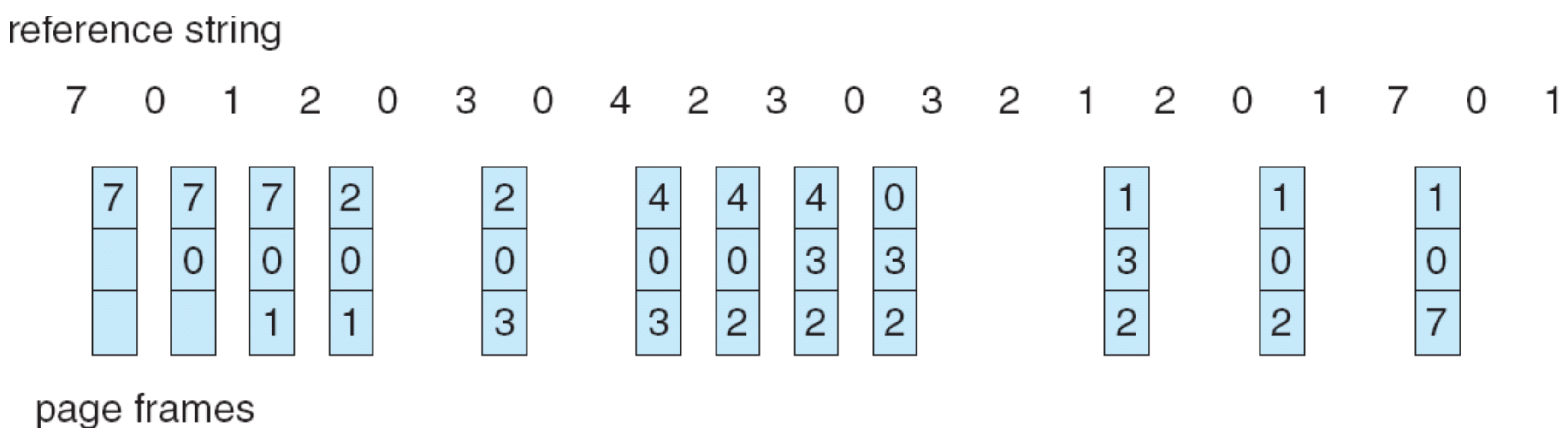
Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**



- Impractical to track.
  - Would need one additional memory access for each memory access or
  - Linked list of each page with most recent in front.
- Can approximate, similar to NRU
  - Need hardware assistance
  - Page reference bit in page table

LRU Page Replacement



LRU Algorithm

- Stack implementation – keep a stack of page numbers in a double link form
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement

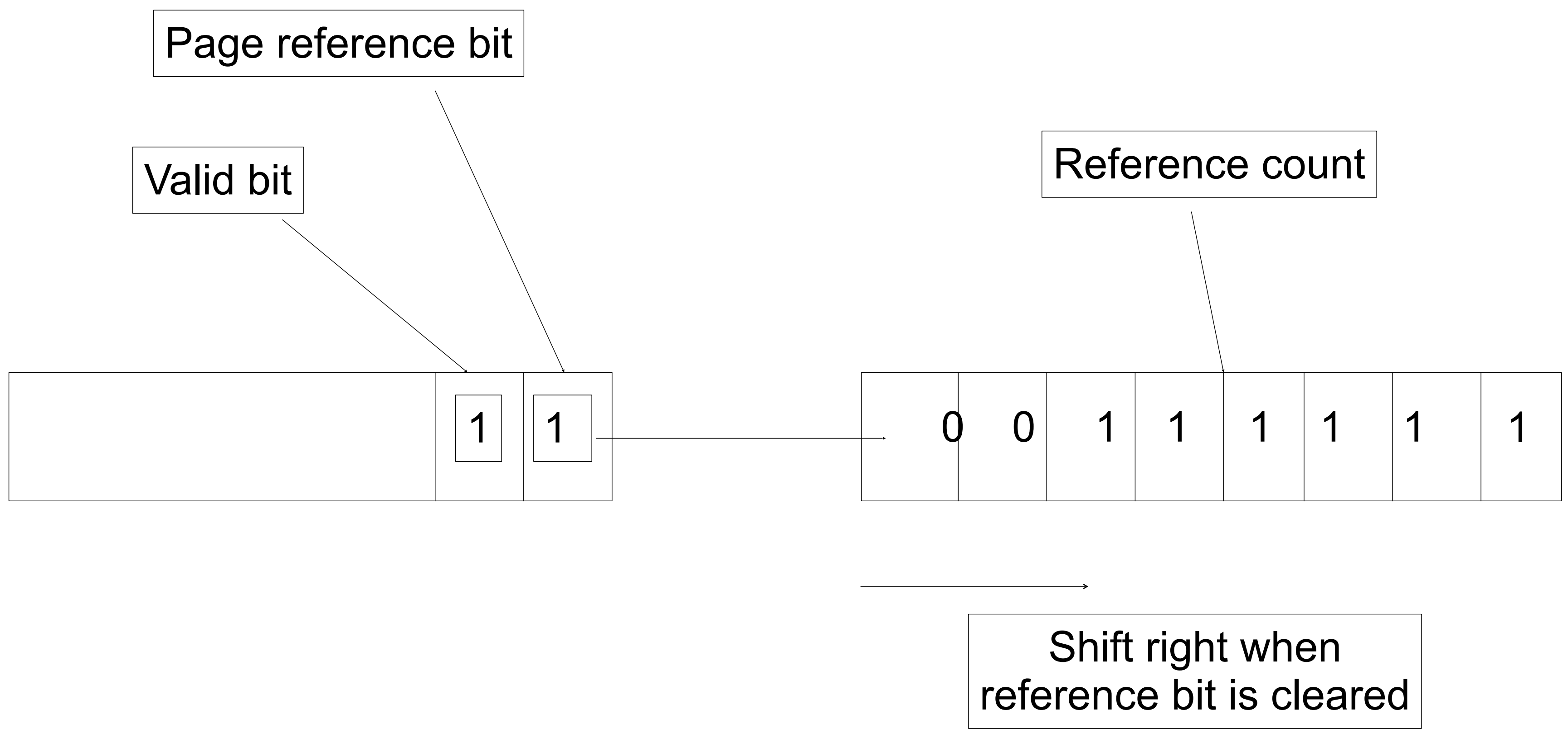
LRU Algorithm

- Stack implementation – keep a stack of page numbers in a double link form
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement

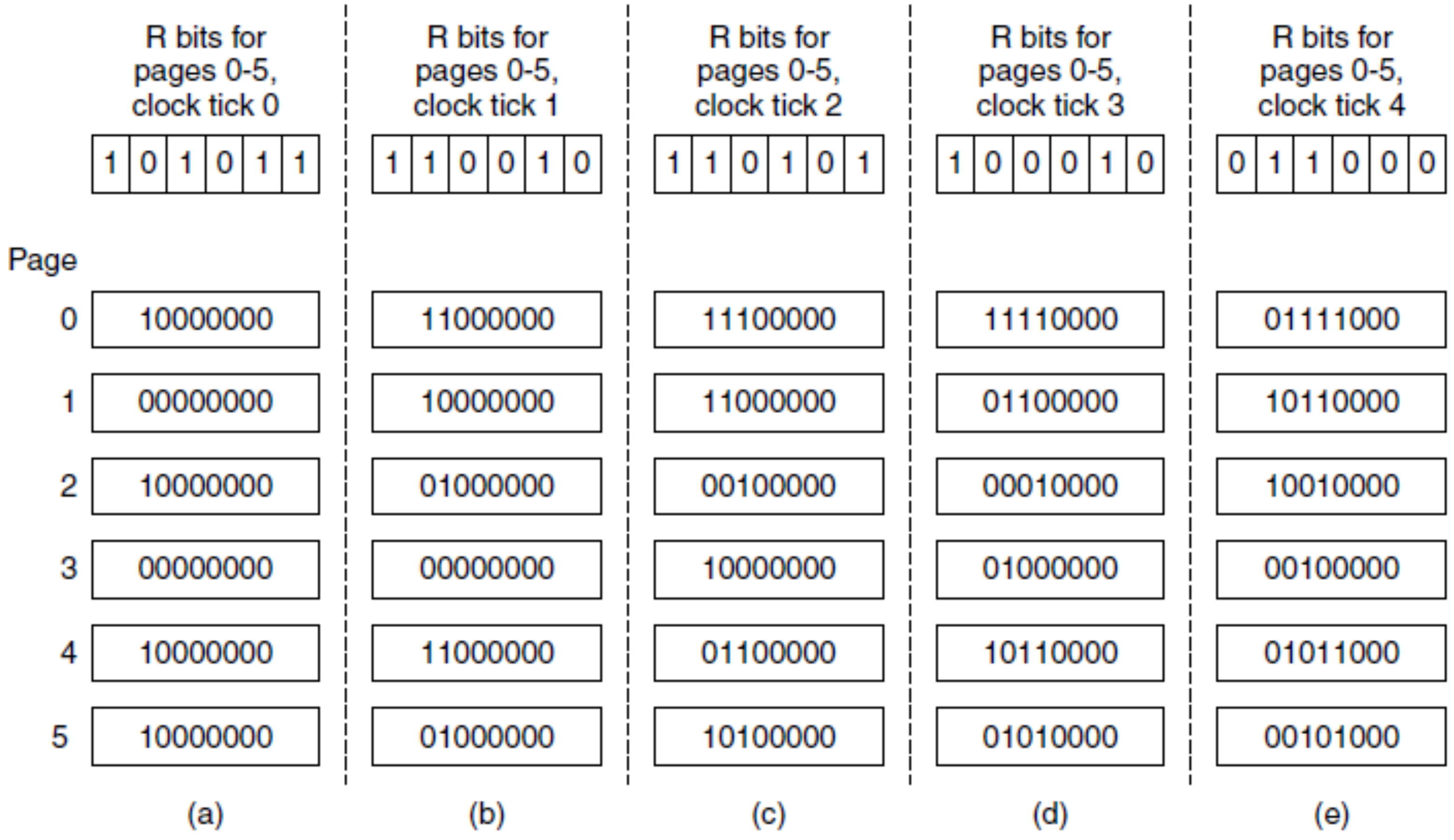
Reference Counter Variation

- Also known as Aging
- As reference bits are cleared, shift into a counter
  - one for each page
- Smallest counter referenced is least recently

Page Reference Bit & Counter

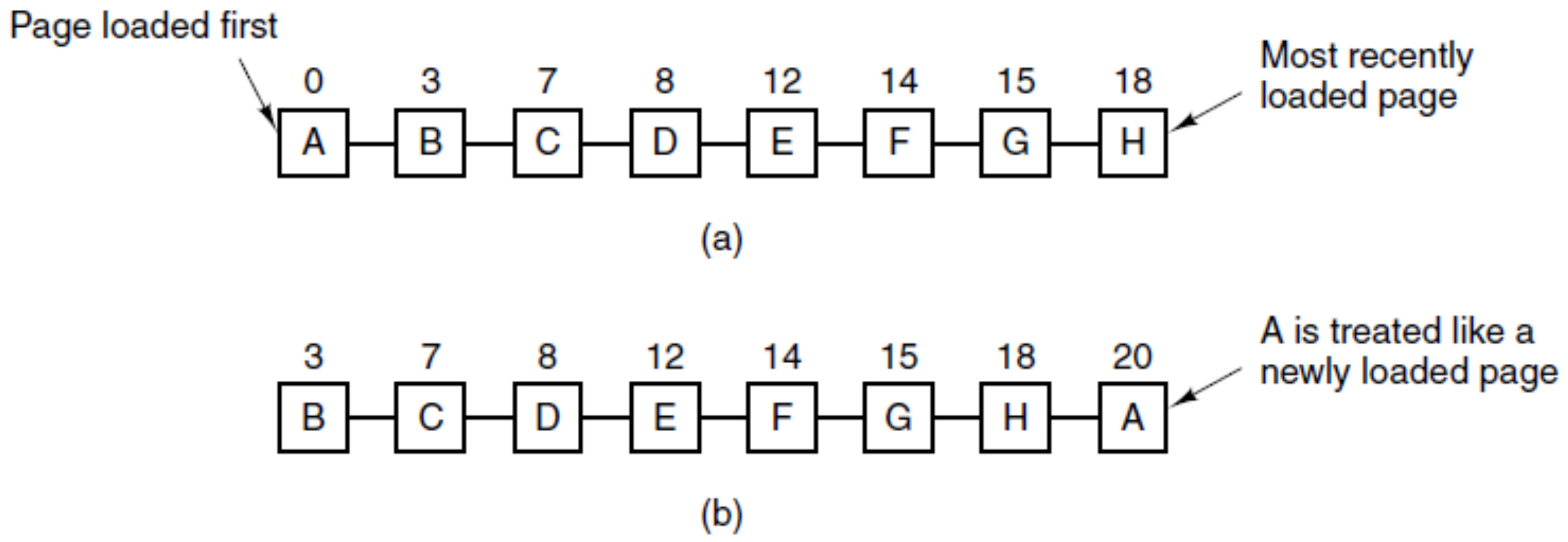


Page Reference Bit & Counter


R bits for pages 0-5, clock tick 0
1 0 1 0 1 1
R bits for pages 0-5, clock tick 1
1 1 0 0 1 0
R bits for pages 0-5, clock tick 2
1 1 0 1 0 1
R bits for pages 0-5, clock tick 3
1 0 0 0 1 0
R bits for pages 0-5, clock tick 4
0 1 1 0 0 0
Page
(a)
(b)
(c)
(d)
(e)

Page	(a)	(b)	(c)	(d)	(e)
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00010000	10010000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000

Second Chance Algorithm

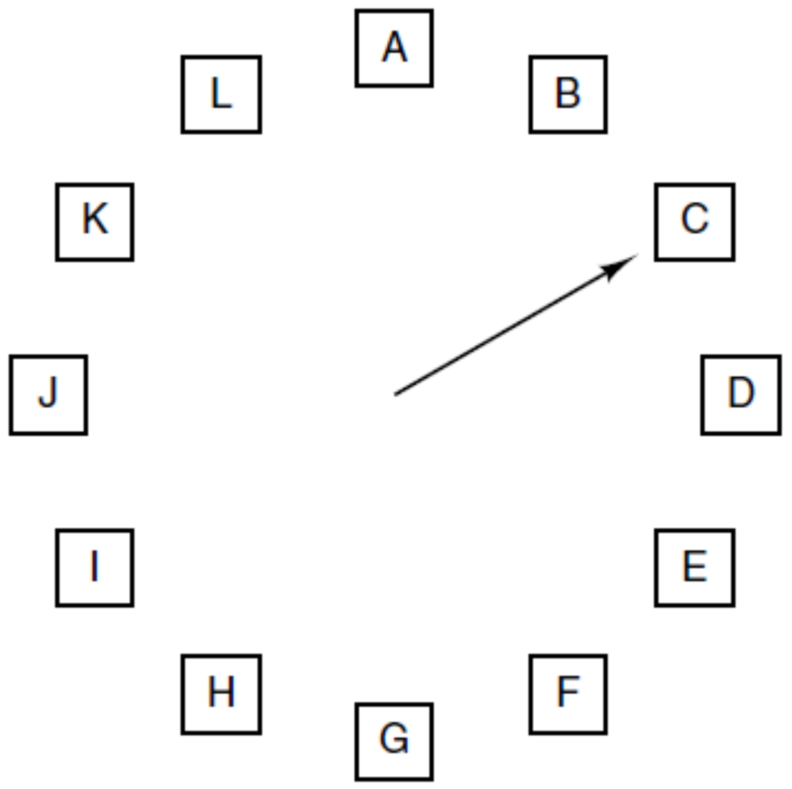


- Sorted in reference time order.
- When page referenced, set R bit
- When faulted check oldest page, if R bit set then set page time to current time, clear R bit and move to the end.
- Repeat until page found with no R bit set.

Not Recently Used

- Better to removed a modified page than a clean page frequently used.
- Easy to understand
- Moderately efficient to implement
- Not optimal, but adequate.

Clock Page 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

Not Recently Used

- Each page has two status bit, R (read) and M (modified)
- On startup all bits are set to 0.
- Every page reference the R bit is set, every dirty page the M bit is set
- Every clock tick R bits are cleared

Not Recently Used

- On a page fault, OS inspects all pages and puts them into 4 categories
- Class 0: not referenced, not modified
- Class 1: not referenced, modified
- Class 2: referenced, not modified
- Class 3: referenced, modified
- NRU removes a page at random from the lowest class

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- LFU Algorithm: replaces page with smallest count
- MFU Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Allocation of Frames

- Each process needs ***minimum*** number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle ***from***
  - 2 pages to handle ***to***
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process

  - s_i = size of process p_i
  - $S = \sum s_i$
  - m = total number of frames
  - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_i = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size

- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global vs. Local Allocation

- Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another

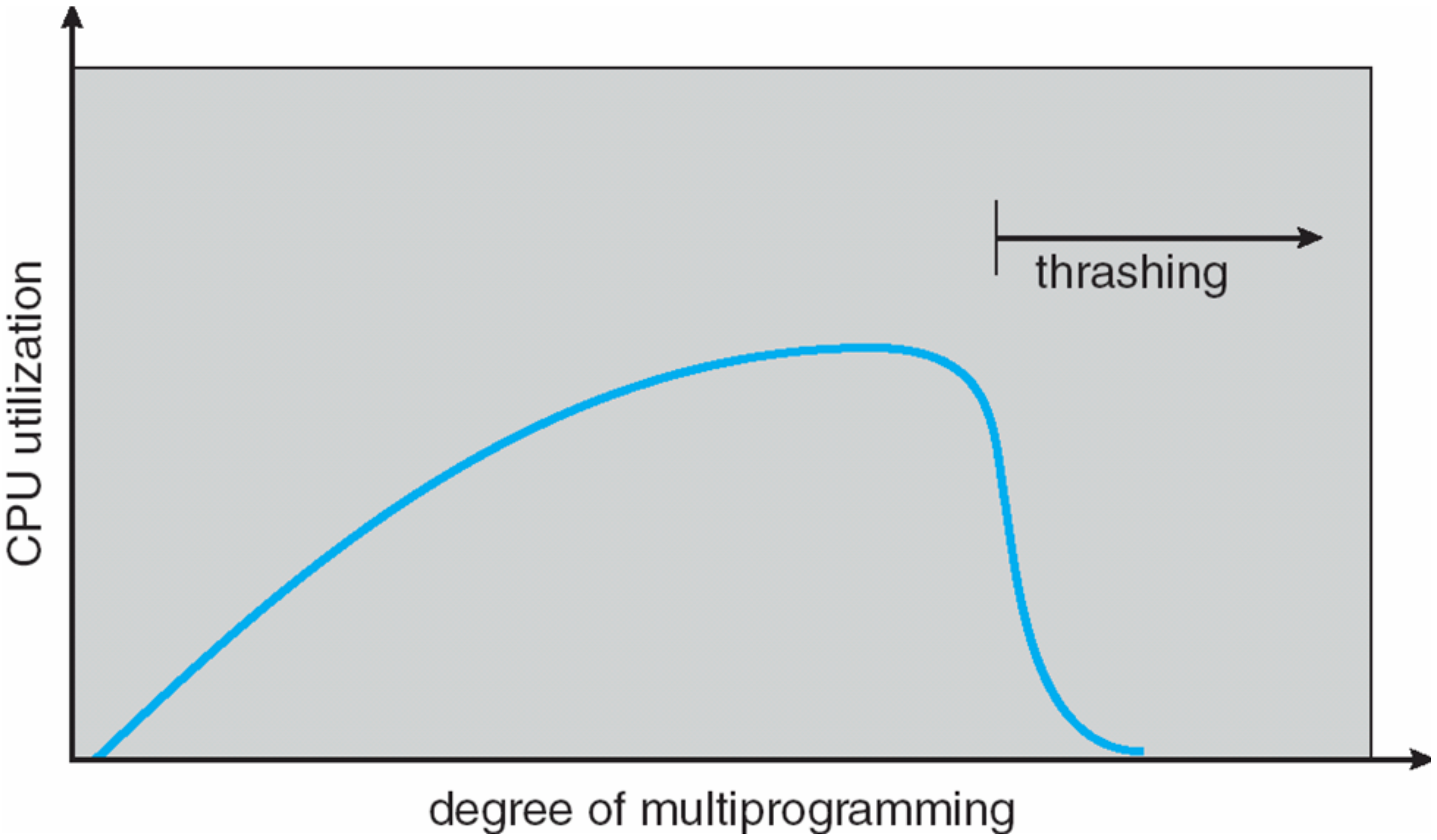
- Local replacement – each process selects from only its own set of allocated frames

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
    - low CPU utilization
    - operating system thinks that it needs to increase the degree of multiprogramming
    - another process added to the system

- **Thrashing** - a process is busy swapping pages in and out

Thrashing (Cont.)



Demand Paging and Thrashing

Why does demand paging work?

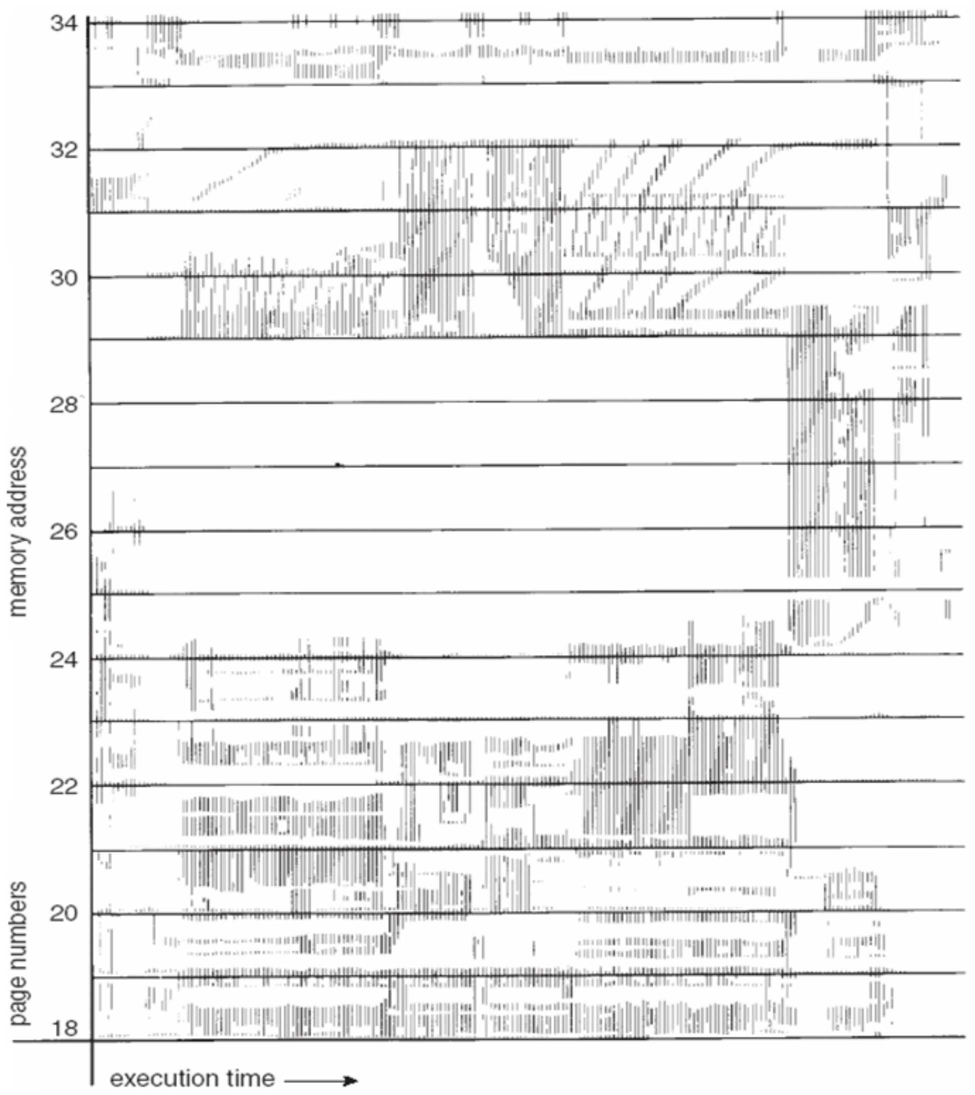
Locality model

- Process migrates from one locality to another
- Localities may overlap

Why does thrashing occur?

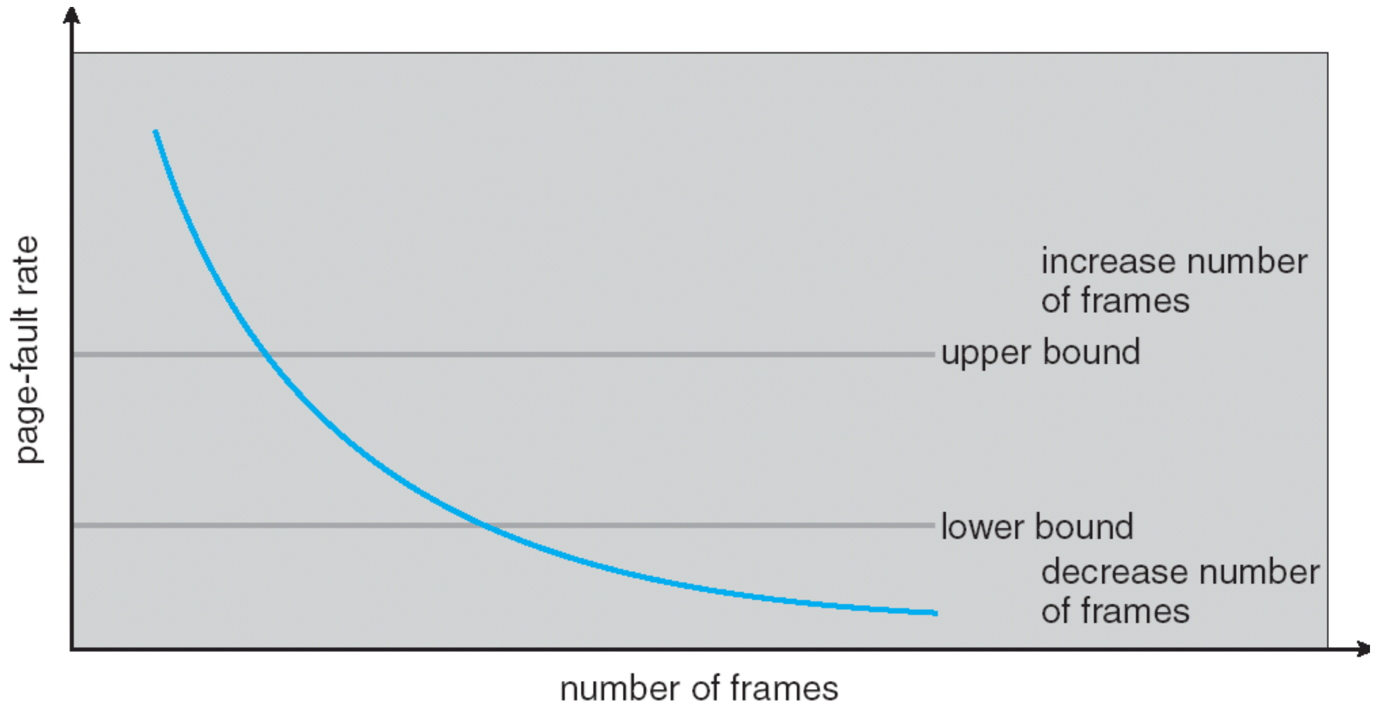
Σ size of locality > total memory size

Locality In A Memory-Reference Pattern



Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame


page-fault rate
increase number of frames
upper bound
lower bound
decrease number of frames
number of frames