

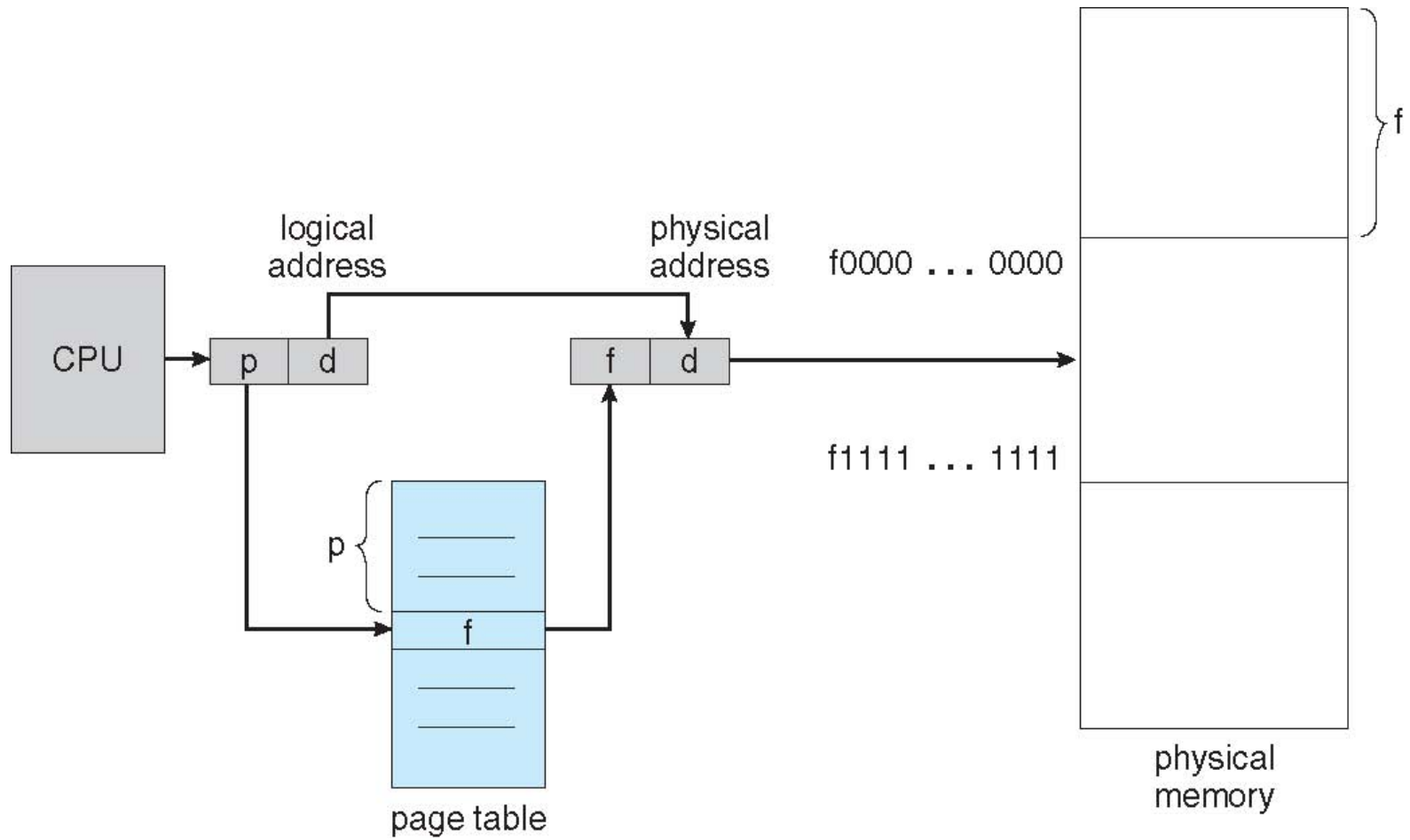
Memory Management

Chapter 11

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



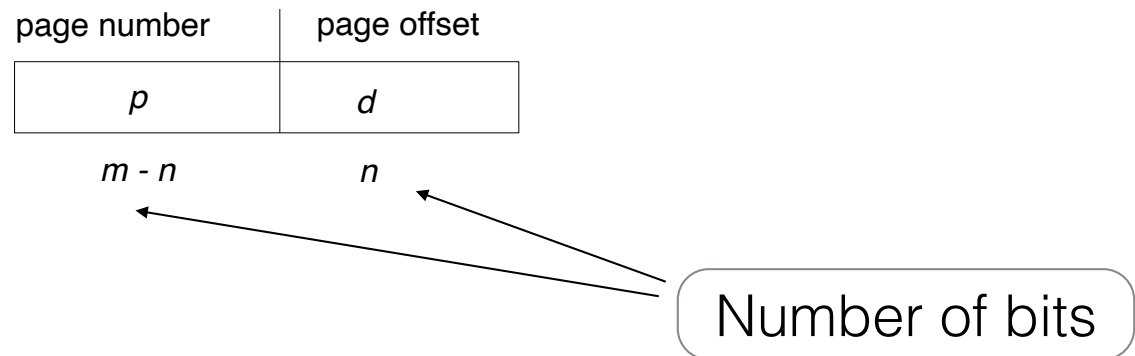
Address Translation Scheme

- » 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

page number	page offset
p	d
$m - n$	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
<i>20</i>	<i>12</i>

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?

$$2^{12} = 4096 \text{ bytes}$$

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?

$$2^6 = 64 \text{ bytes}$$

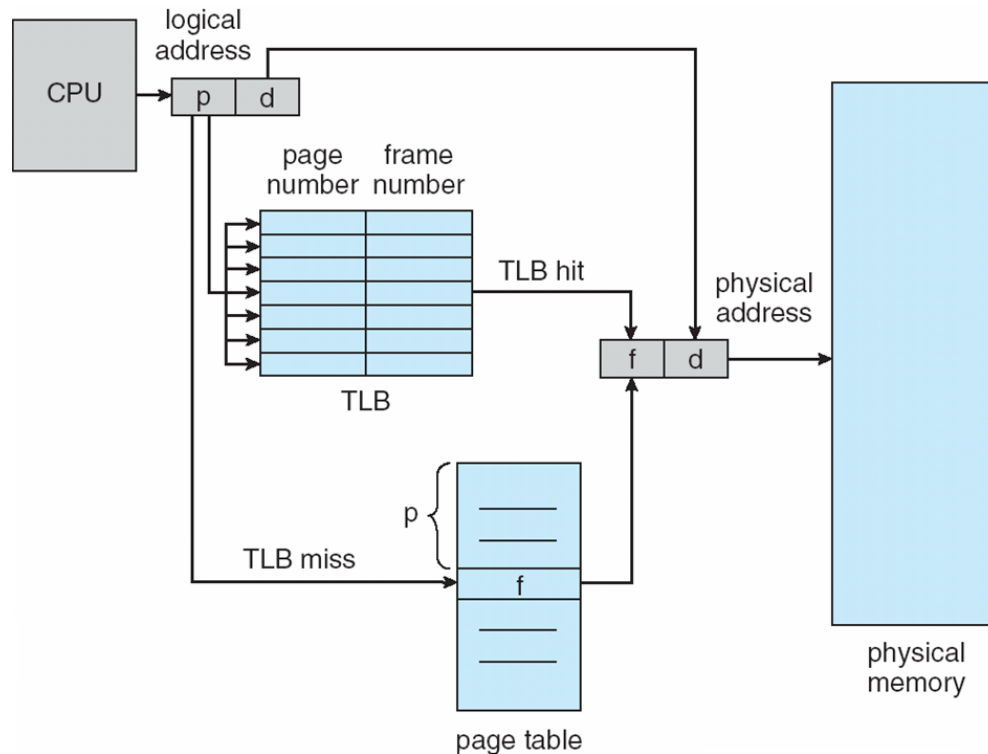
2. How many entries (maximum) is the page table?

$$2^6 = 64 \text{ pages}$$

Implementation of Page Tables

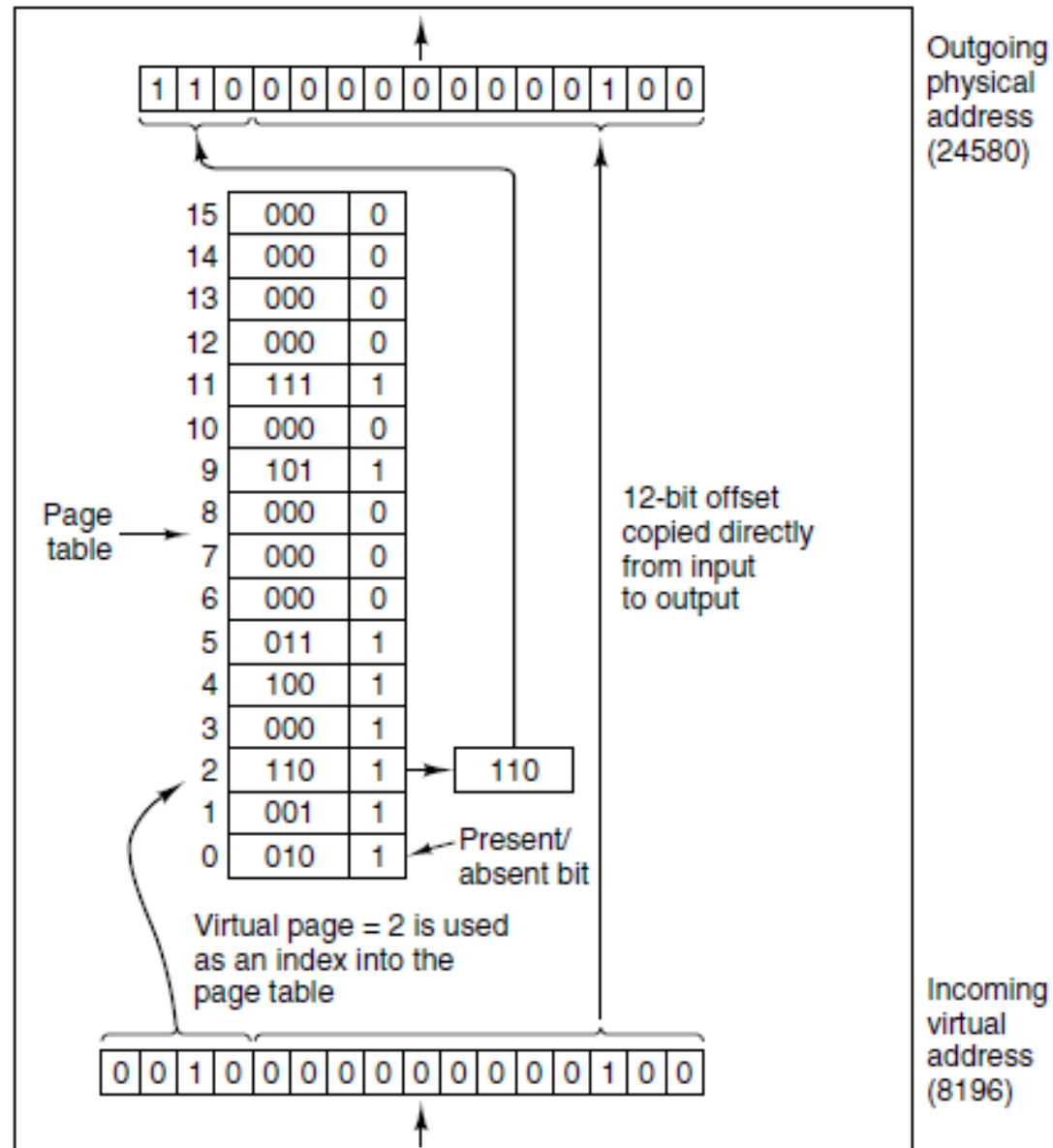
- » In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
 - Half speed
- » The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside 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

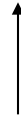
- » Nehalem has a two-level TLB:
 - » The first level of TLB is shared between data and instructions.
 - » The level 1 data TLB now stores 64 entries for small pages (4K) or 32 for large pages (2M/4M),
 - » The level 1 instruction TLB stores 128 entries for small pages (the same as with Core 2) and seven 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)

$$\begin{aligned} \text{EAT} &= (100 + 5) \cdot 0.8 + (200)(1 - 0.8) \\ &= 124 \text{ nanoseconds} \end{aligned}$$

Effective Access Time

- » First equation assumed TLB and Page Table lookups happened in parallel.
- » If not:

Effective Access Time (EAT)

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

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.

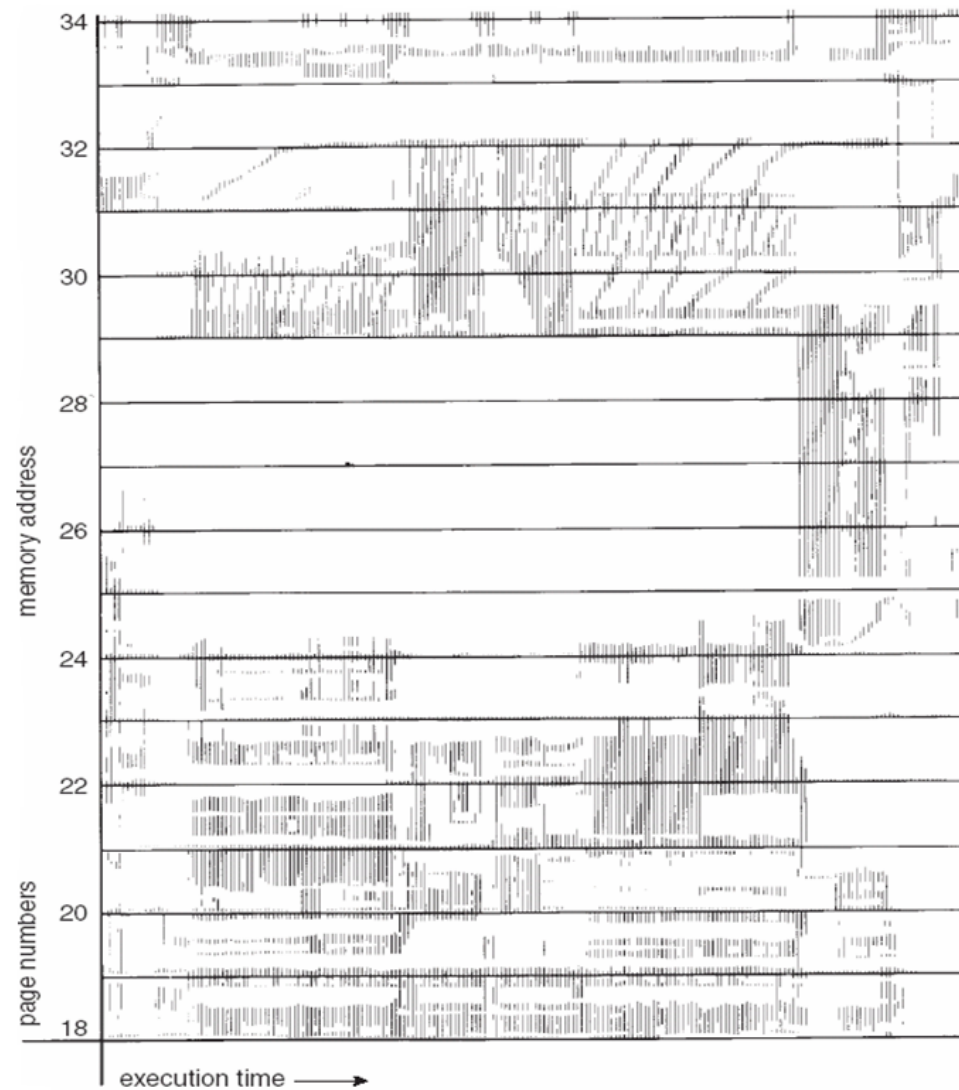
Demand Paging

- » 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.

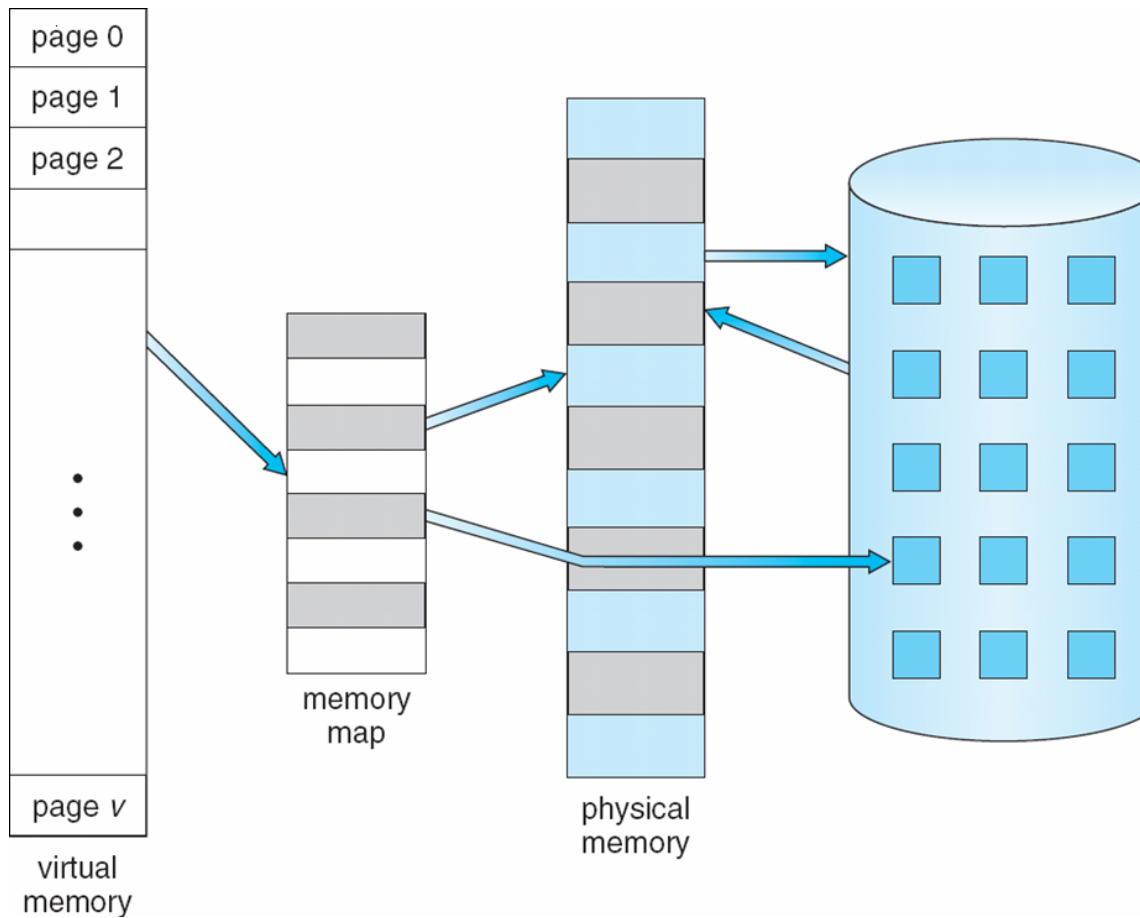
Locality In A Memory-Reference Pattern



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 same 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



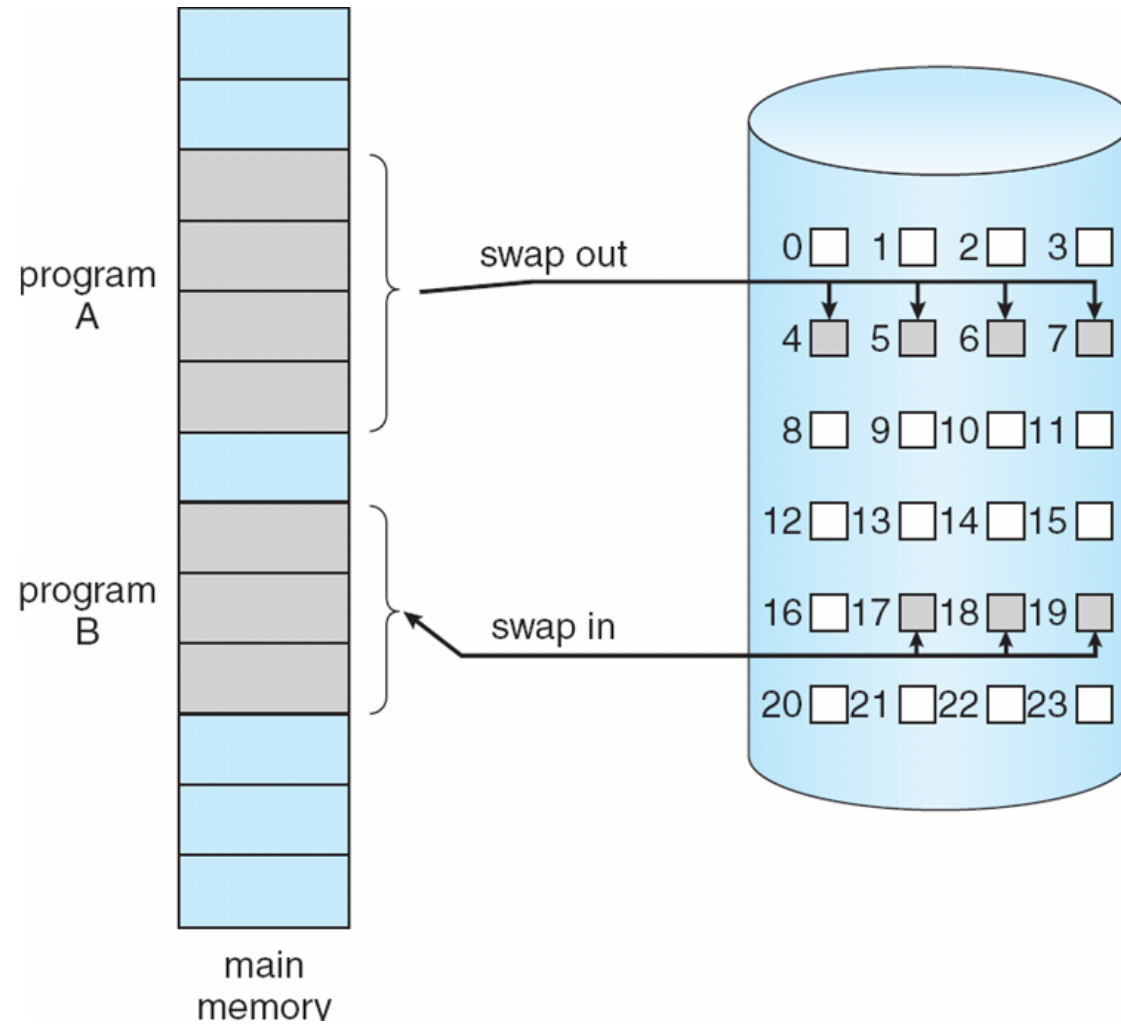
Demand Paging

- » 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



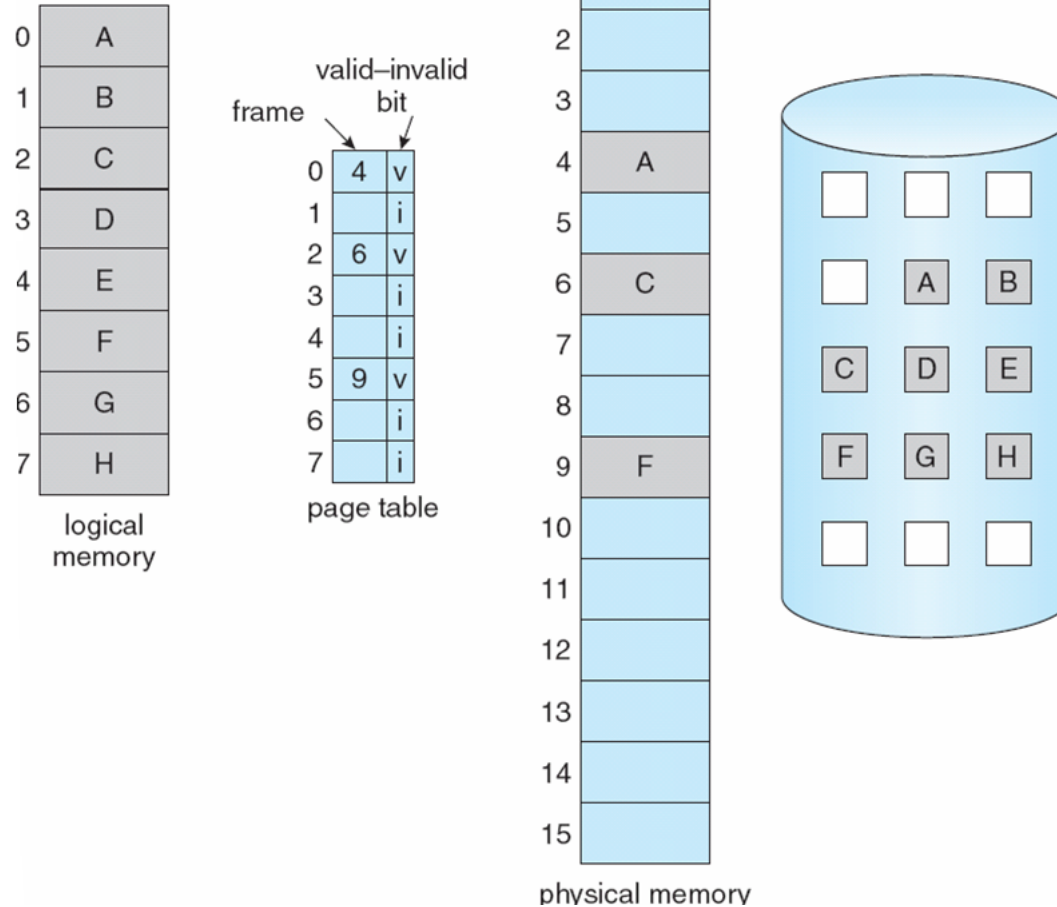
Demand Paging

- » 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:

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
....	
	i
	i

page table

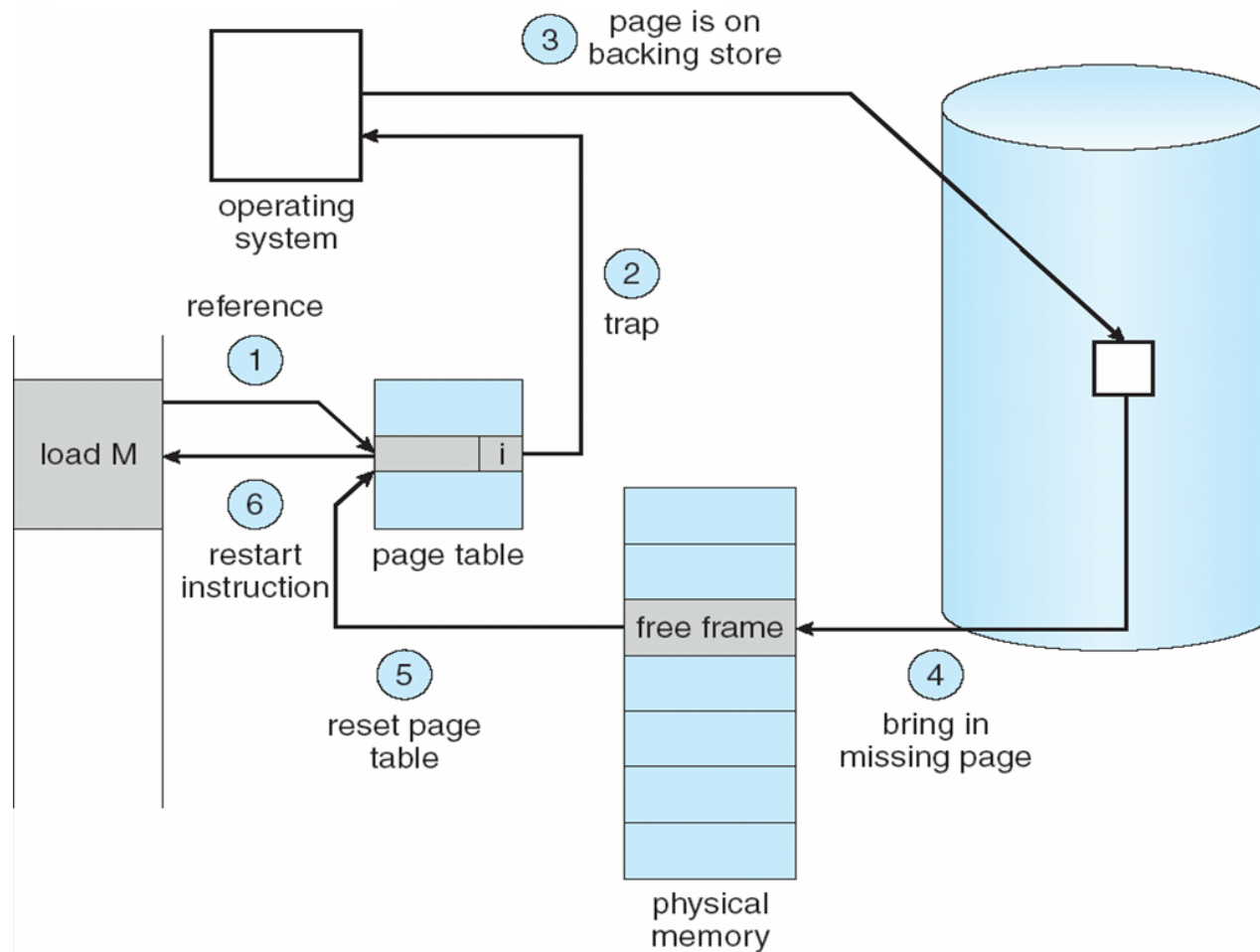
Demand Paging



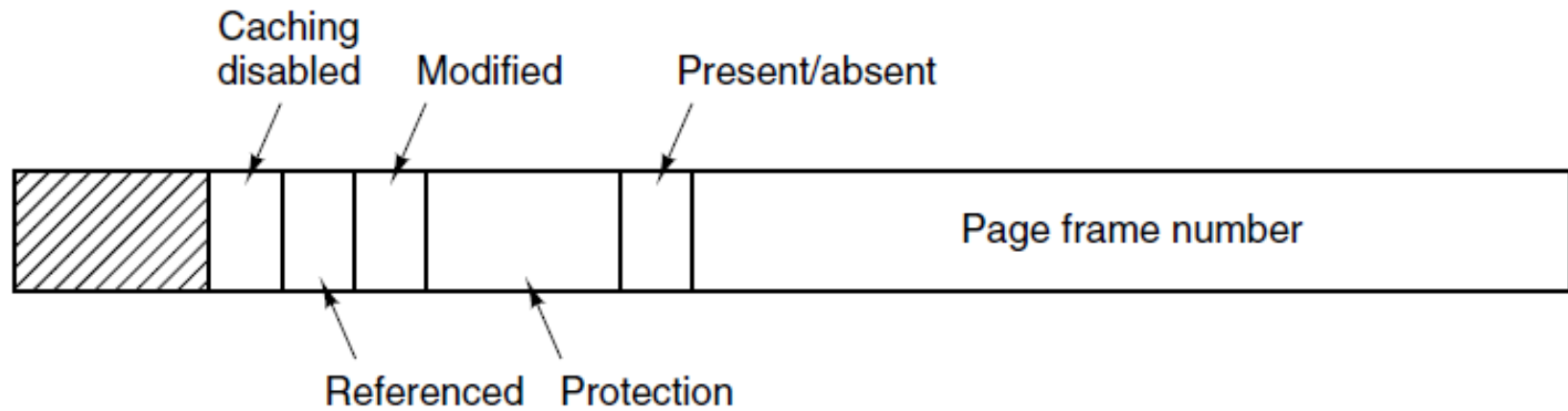
Demand Paging

- » 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
- 6. Restart the instruction that caused the page fault

Demand Paging



Page Table Entry



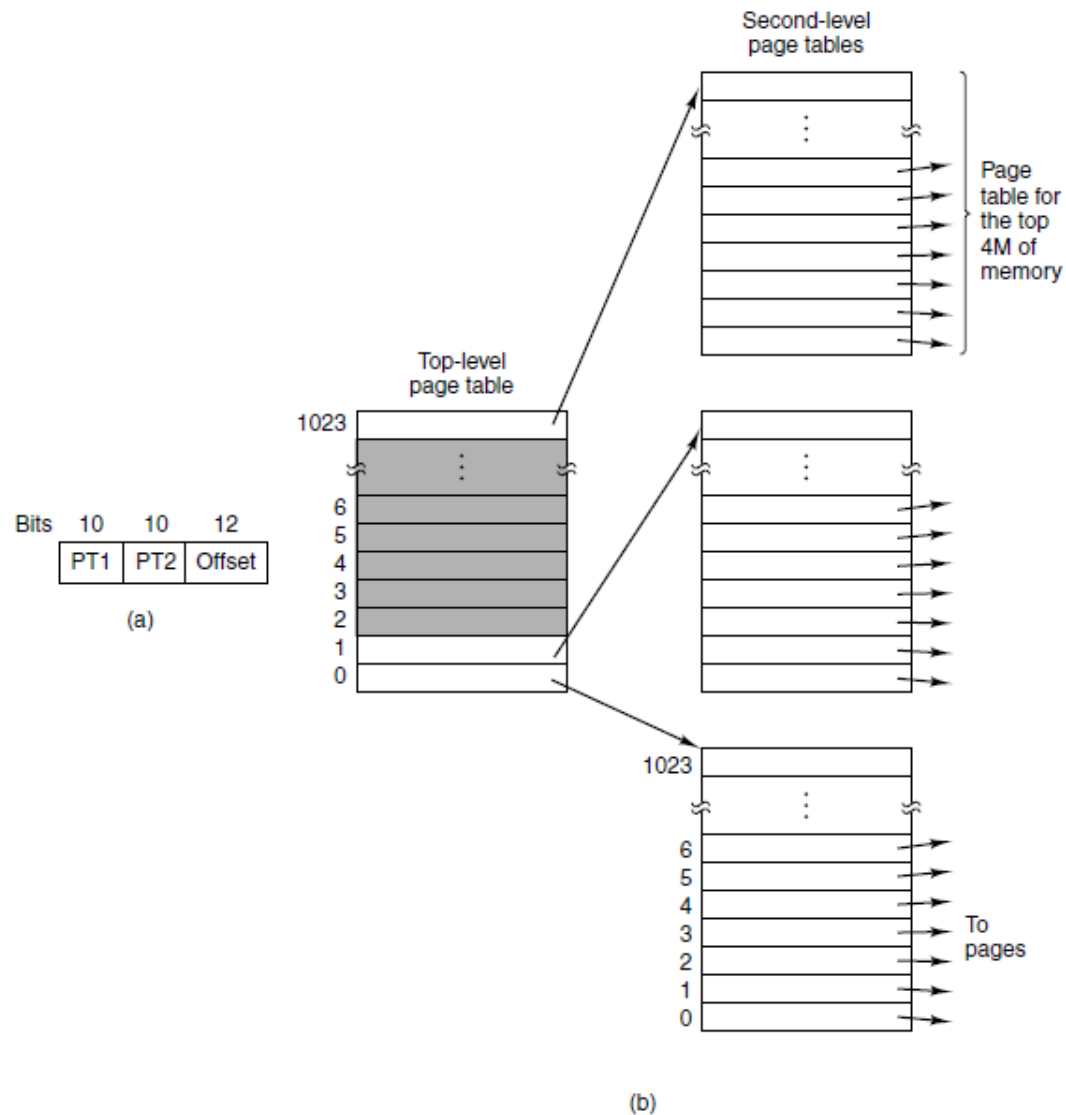
Types of Page Faults

- » 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



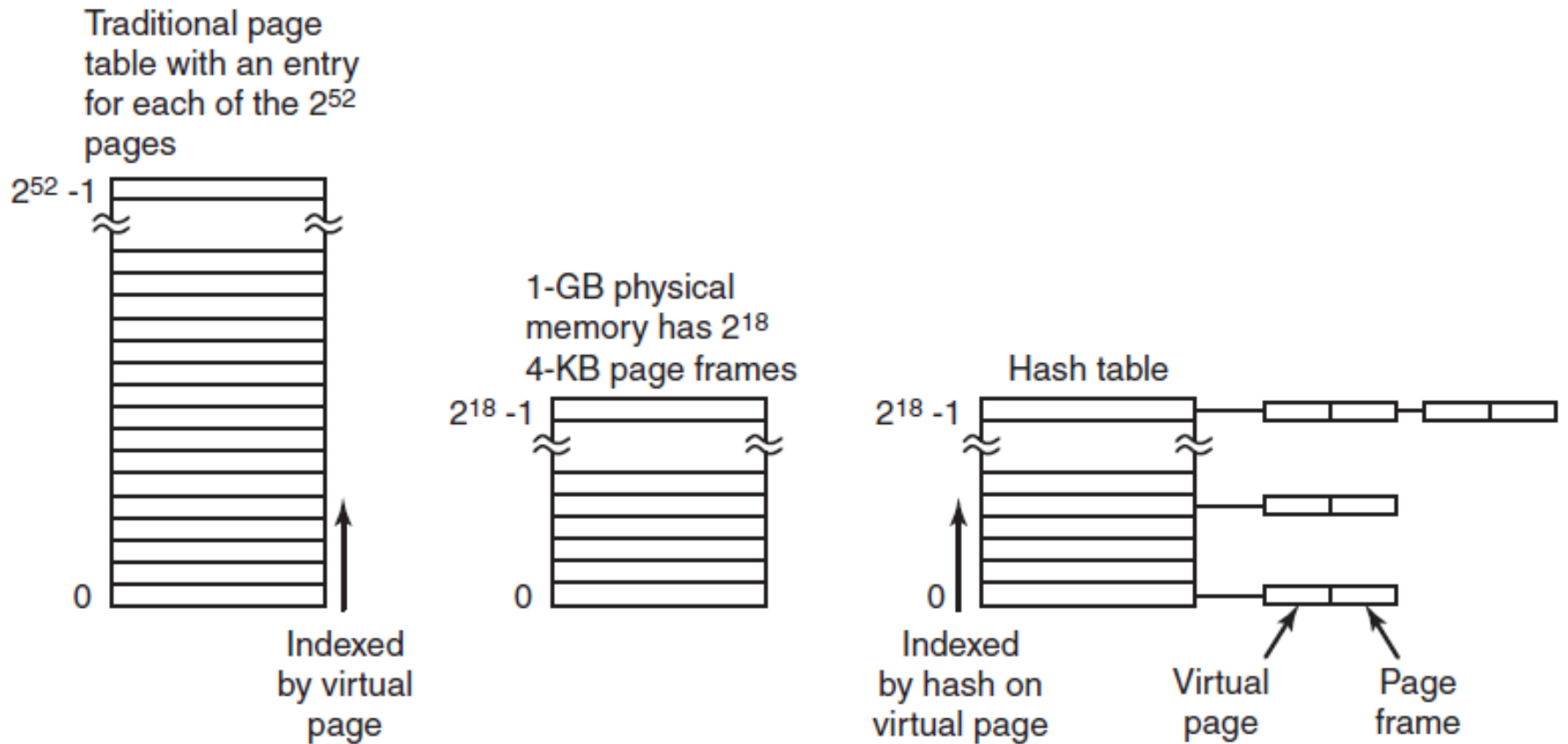
Multilevel Page Tables

- » 80386 addressed 4 GB of memory using a two-level page table.
 - » Page directory \rightarrow page tables \rightarrow 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 space, one entry per page frame of real memory

Inverted Page Tables



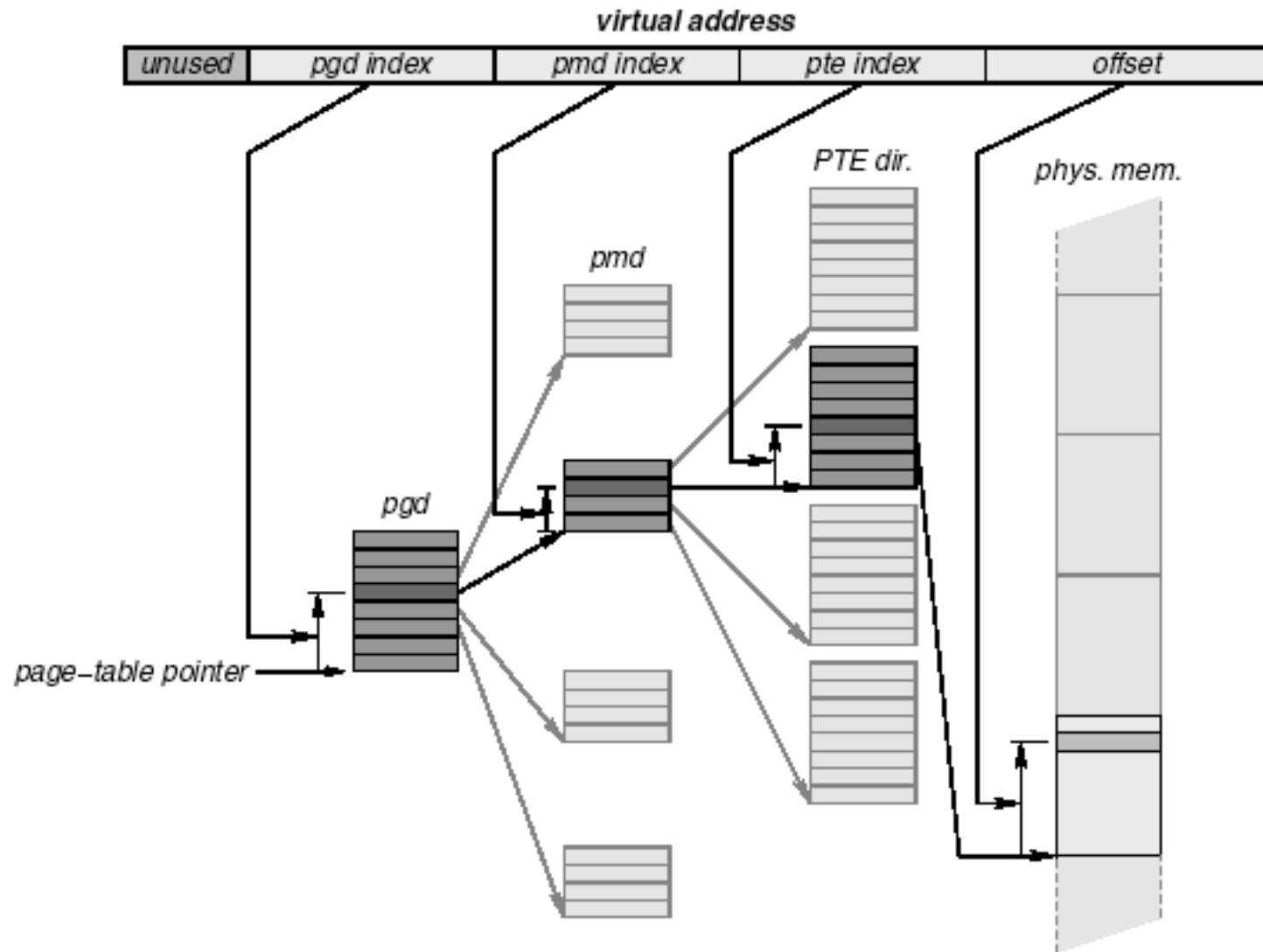
Inverted Page Tables

- » 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 \rightarrow 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	1
12	2	2
12	3	3
125	4	4
1257	5	5
1257	6	6
1567	7	7
1367	8	8
1367	9	9
1367	10	10
1367	11	11
1467	12	12
1246	13	13
2467	14	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

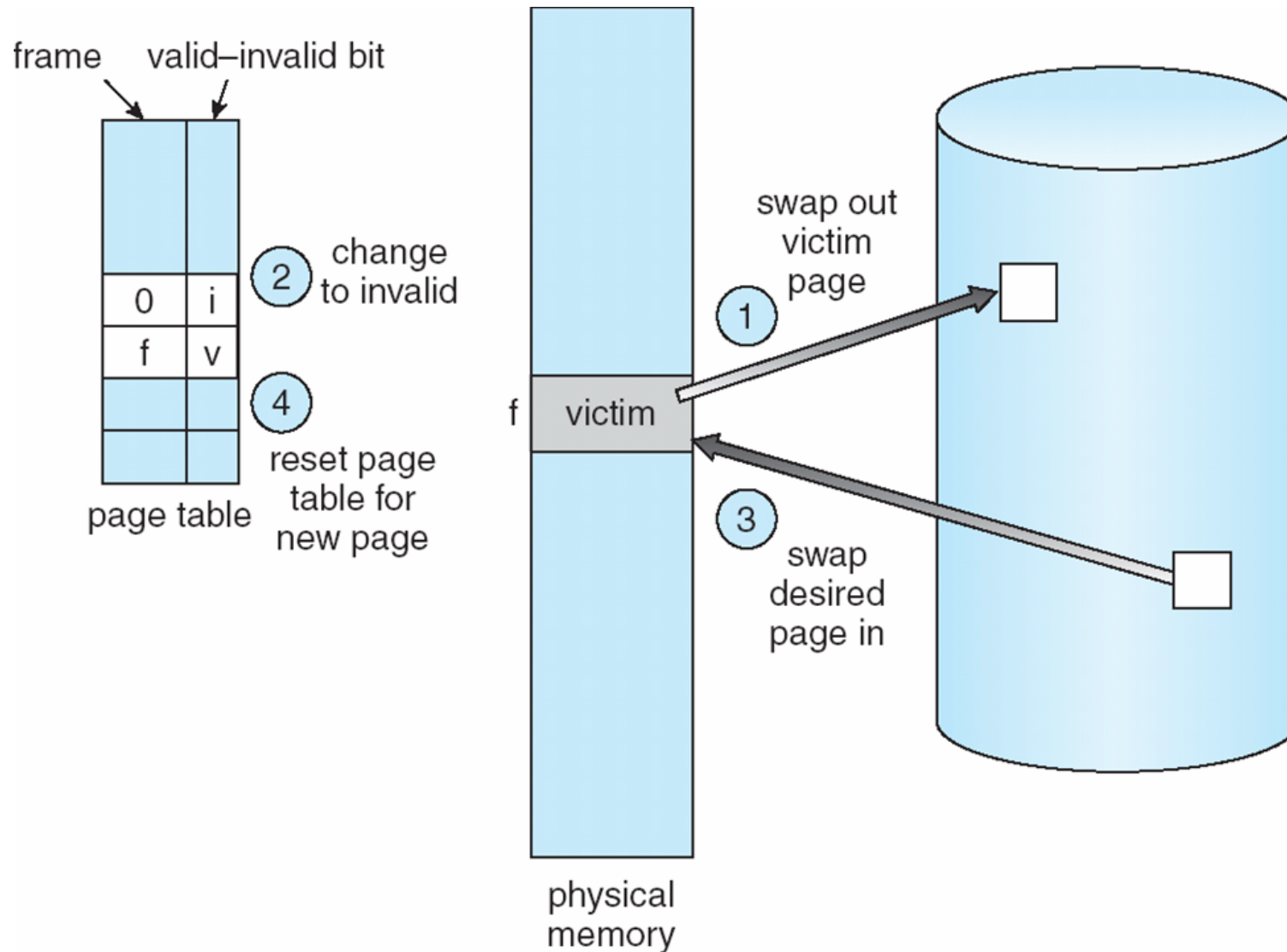
Working set		Event Number
1	1	1
12	2	2
12	3	3
125	4	4
1257	5	5
1257	6	6
1567	7	7
1367	8	8
1367	9	9
1367	10	10
1367	11	11
1467	12	12
1246	13	13
2467	14	14

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



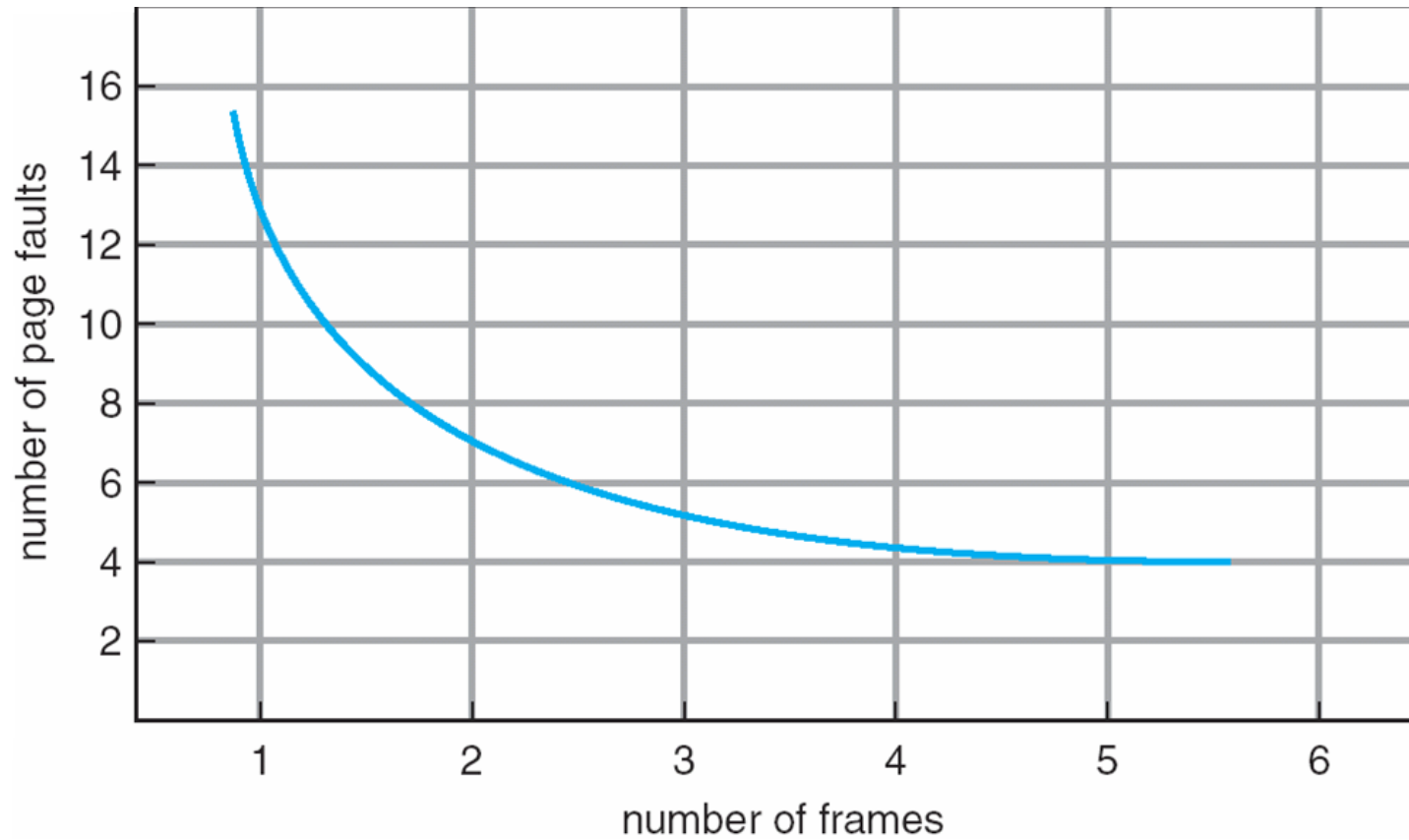
Page Replacement Algorithms

- 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)

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

9 page faults

- 4 frames

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

10 page faults

- Belady's Anomaly: more frames = more page faults

FIFO Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2																
	0	0	0																
		1	1																

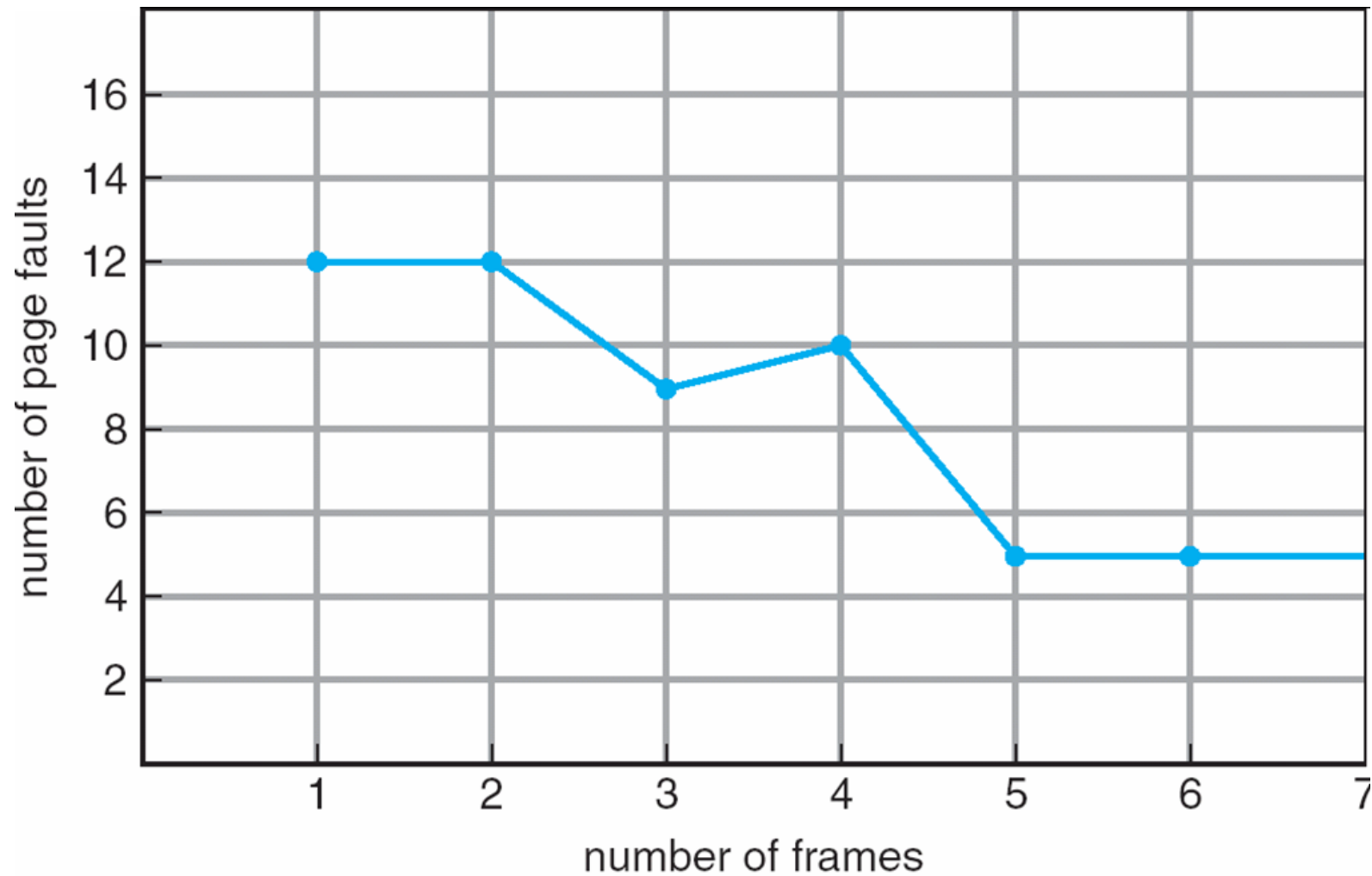
2	2	4	4	4	0														
3	3	3	2	2	2														
1	0	0	0	3	3														

0	0																		
1	1																		
3	2																		

7	7	7																	
1	0	0																	
2	2	1																	

page frames

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

1
2
3
4

4

6 page faults

5

How do you know this?

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

Optimal Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2			2		2			2		2				7		
	0	0	0			0		4			0		0				0		
		1	1			3		3			3		1				1		

page frames

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

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

- 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

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2		2		4	4	4	0			1		1		1	
	0	0	0		0		0	0	3	3			3		0		0	
		1	1		3		3	2	2	2			2		2		7	

page frames

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

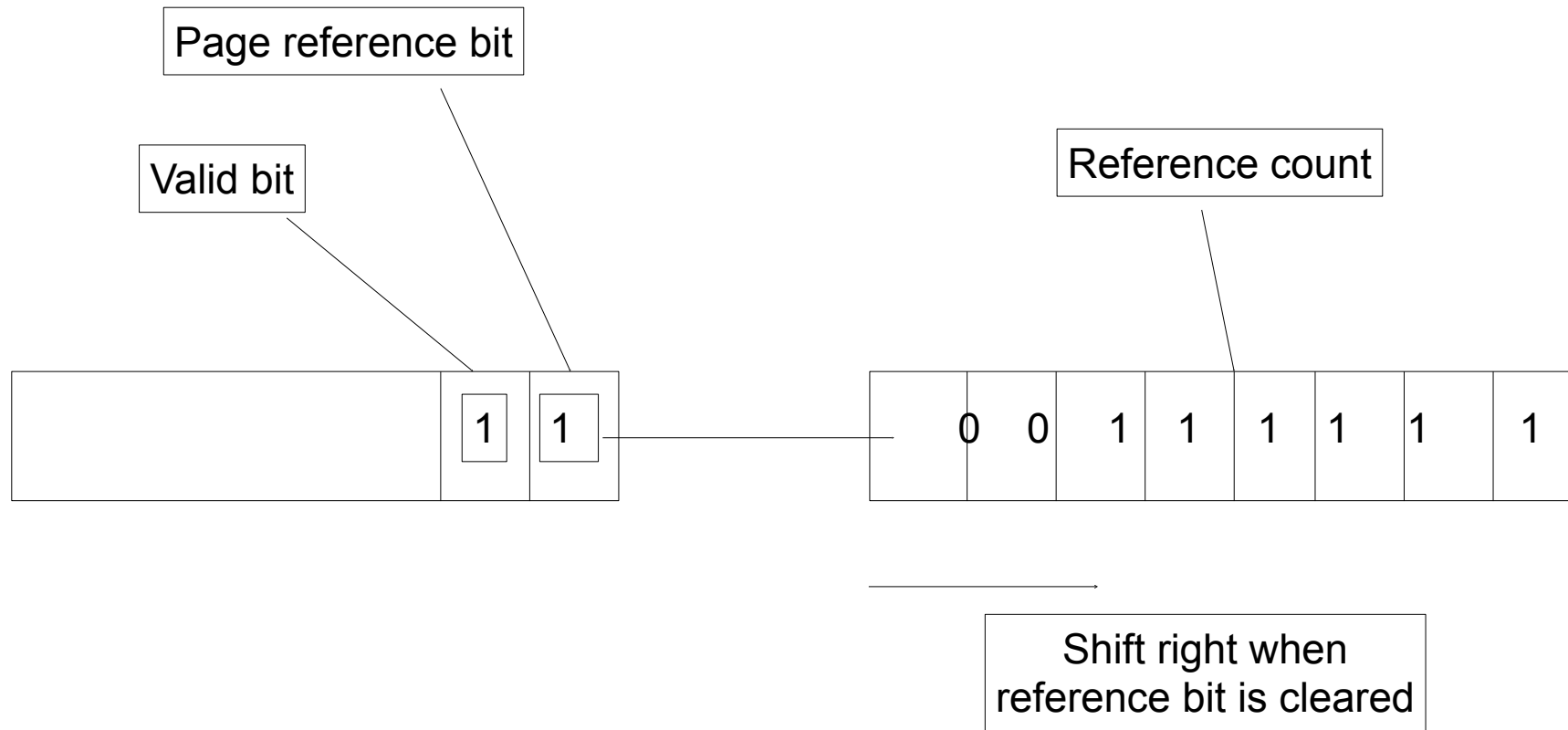
LRU Algorithm

- 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

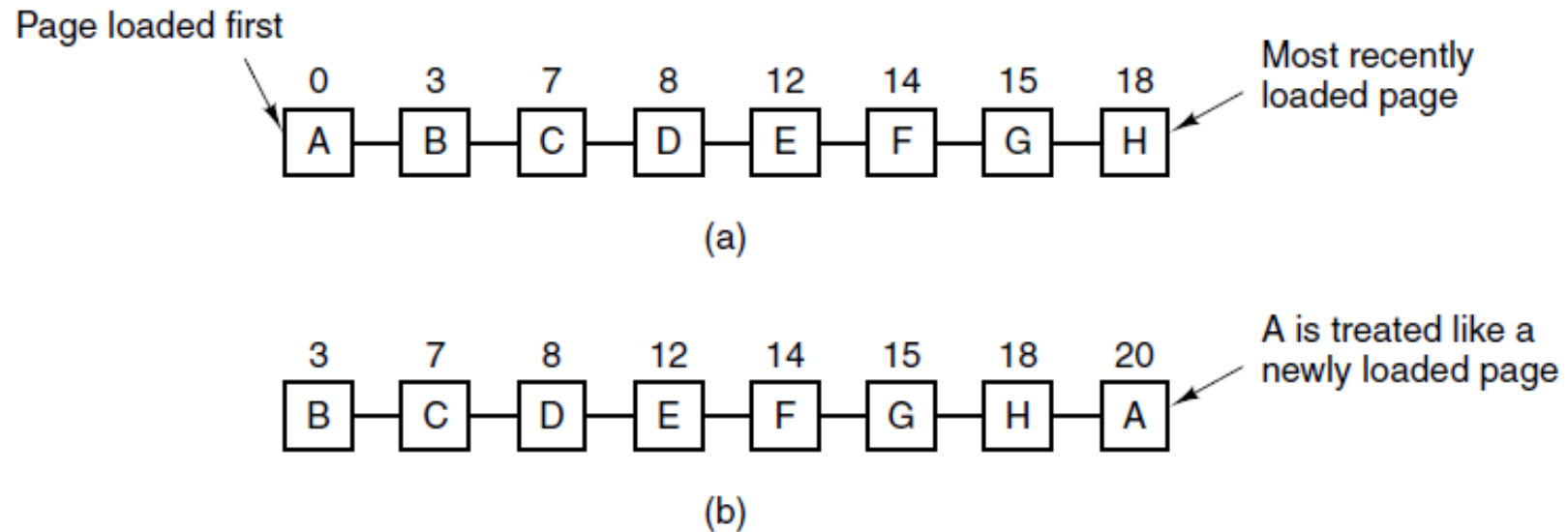
Page Reference Bit & Counter



Page Reference Bit & Counter

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

Second Chance Algorithm

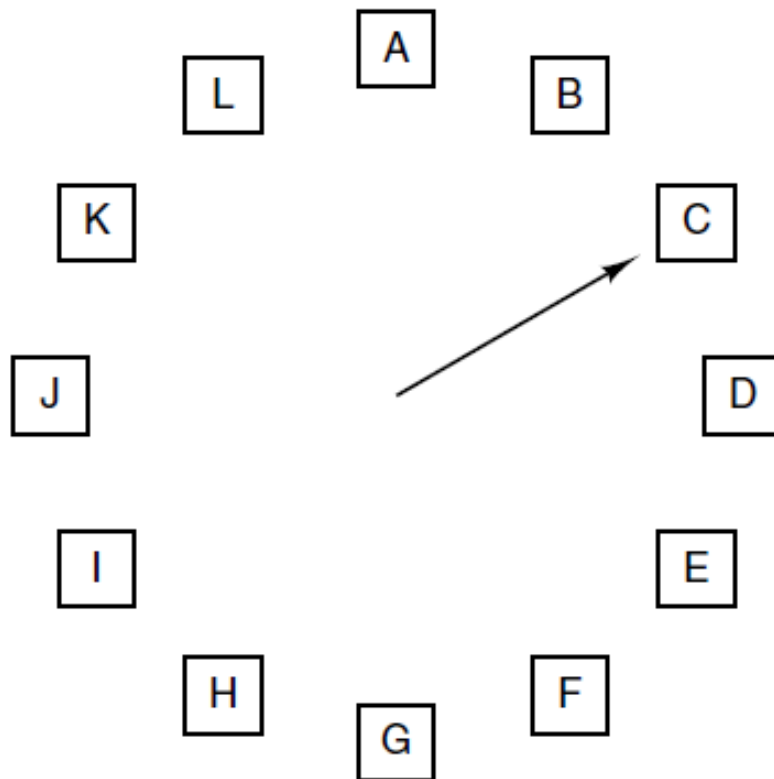


- Sorted in reference time order.
- When page referenced, set R bit
- When faulted check oldest page, if R bit set then set page time to current time, clear R bit and move to the end.
- Repeat until page found with no R bit set.

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

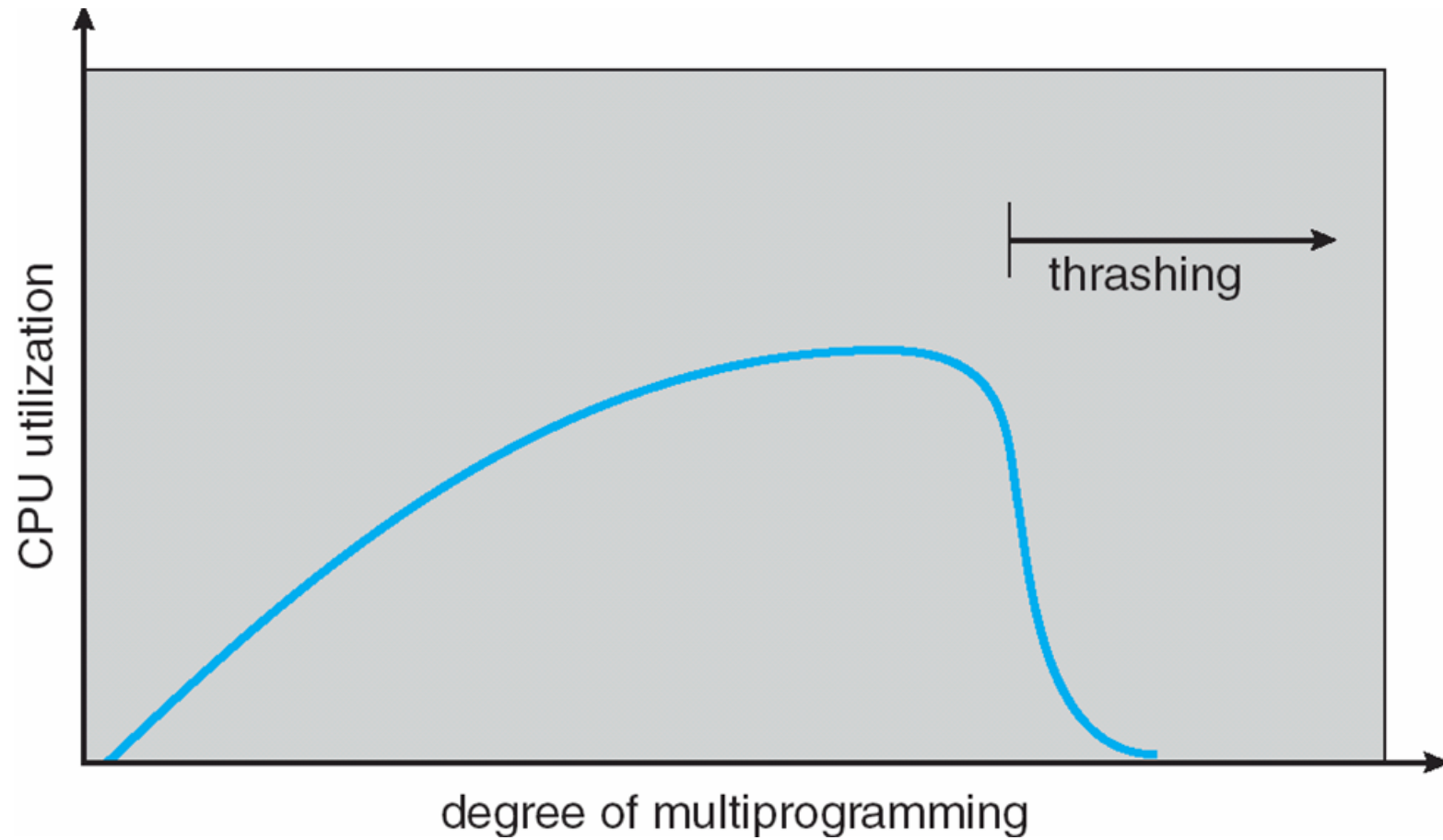
Global vs. Local Allocation

- **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

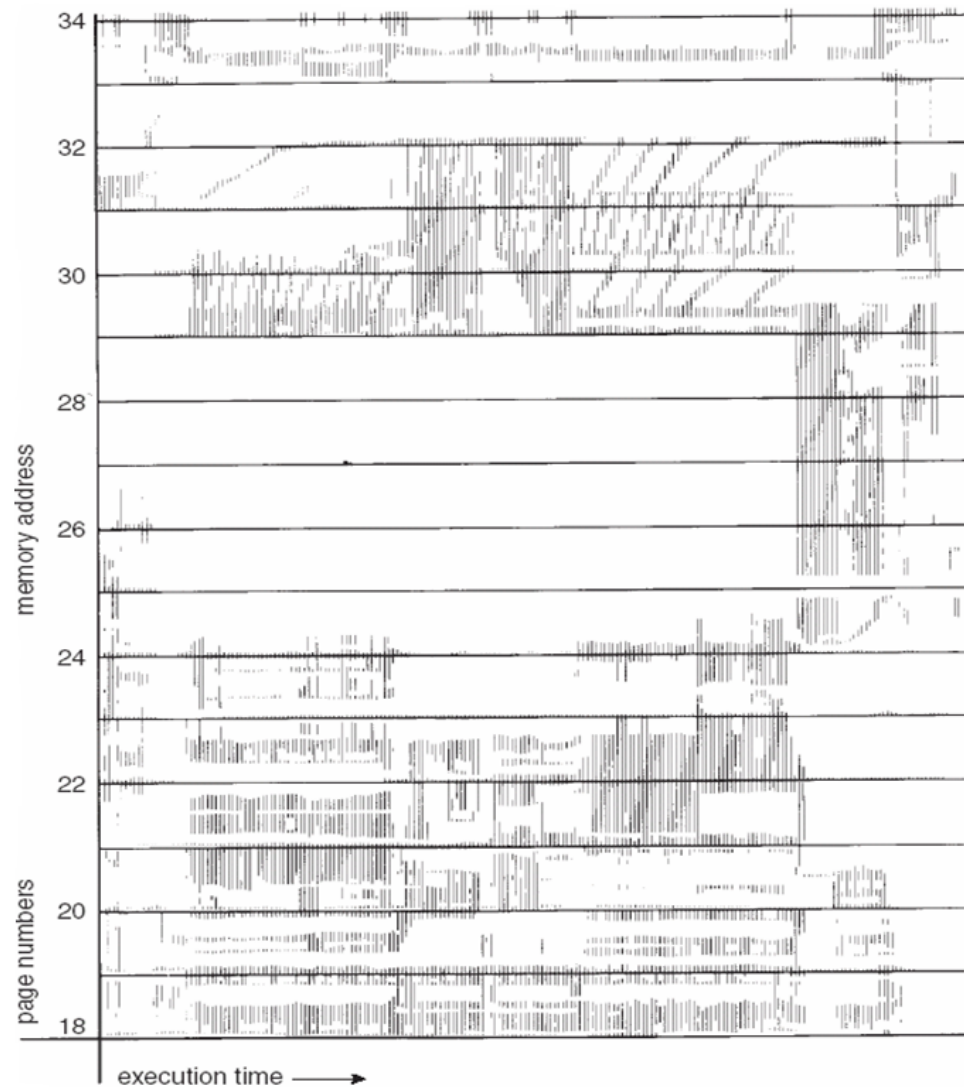
Process migrates from one locality to another

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

