Chapter 10: Virtual Memory

Chapter 10: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

Objectives

- Define virtual memory and describe its benefits.

- Illustrate how pages are loaded into memory using demand paging.

- Apply the FIFO, optimal, and LRU page-replacement algorithms.

- Describe the working set of a process, and explain how it is related to program locality.

- Describe how Linux, Windows 10, and Solaris manage virtual memory.

- Design a virtual memory manager simulation in the C programming language.

Background

- Code needs to be in memory to execute, but entire program rarely used

  - Error code, unusual routines, large data structures

- Entire program code not needed at same time

- Consider ability to execute partially-loaded program

  - Program no longer constrained by limits of physical memory

  - Each program takes less memory while running -> more programs run at the same time

    - Increased CPU utilization and throughput with no increase in response time or turnaround time

  - Less I/O needed to load or swap programs into memory -> each user program runs faster

Virtual memory

- **Virtual memory** – separation of user logical memory from physical memory
    - Only part of the program needs to be in memory for execution
    - Logical address space can therefore be much larger than physical address space
    - Allows address spaces to be shared by several processes
    - Allows for more efficient process creation
    - More programs running concurrently
   - Less I/O needed to load or swap processes

Virtual memory (Cont.)

- **Virtual address space** – logical view of how process is stored in memory

  - Usually start at address 0, contiguous addresses until end of space

  - Meanwhile, physical memory organized in page frames

  - MMU must map logical to physical

- Virtual memory can be implemented via:

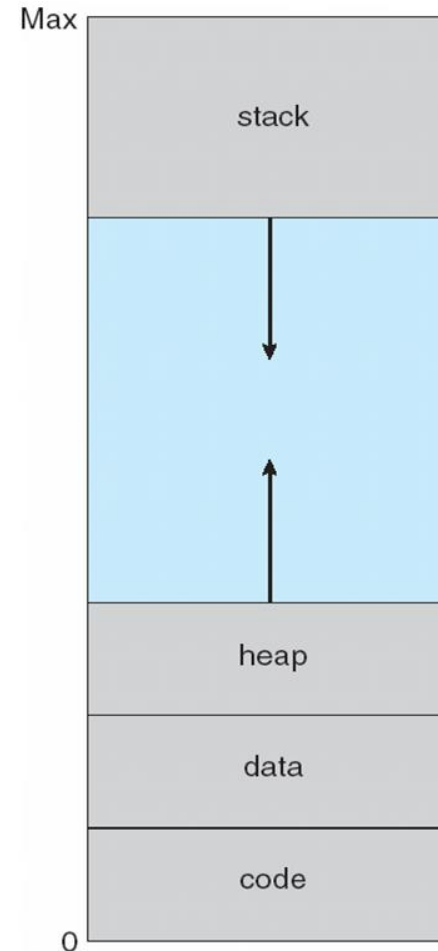 - Demand paging

  - Demand segmentation

virtual memory

memory map

physical memory

backing store

page 0
page 1
page 2
page v
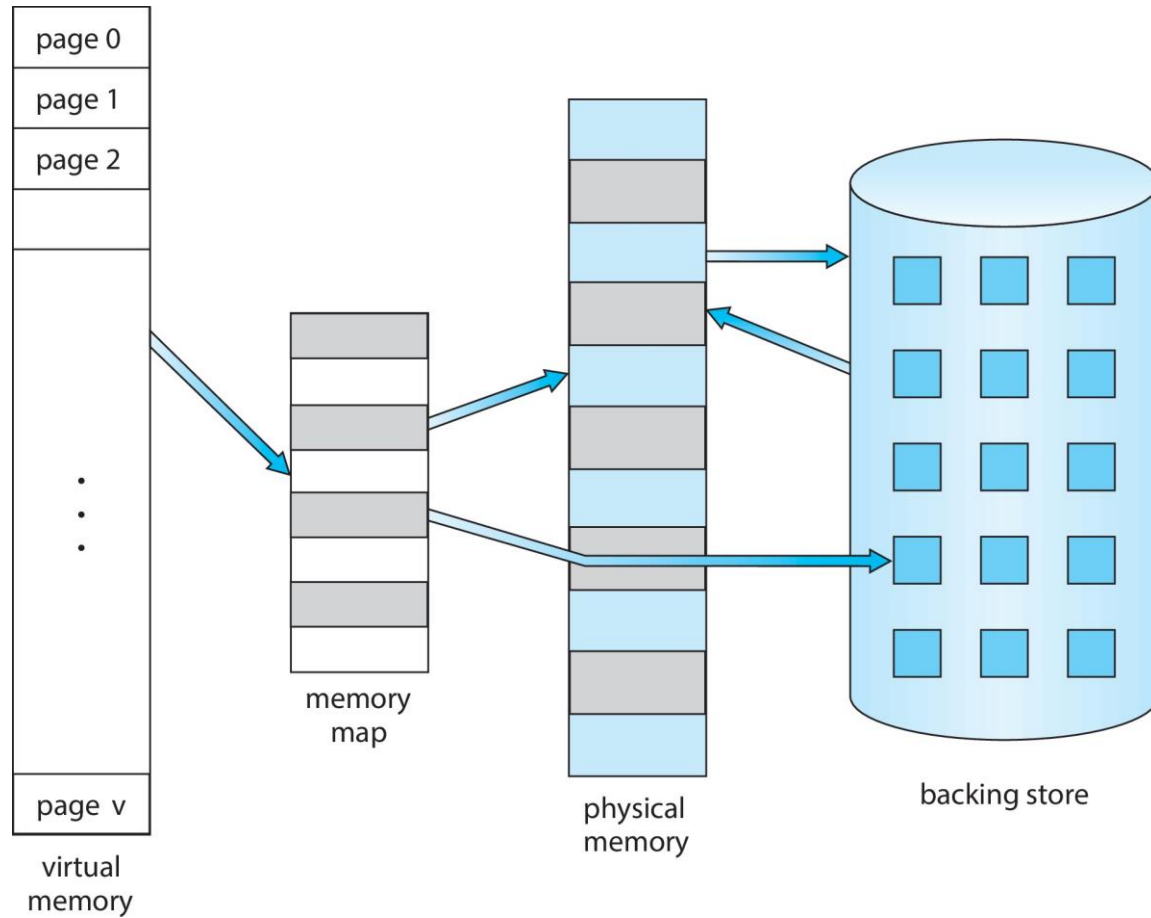
Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow "down" while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two is hole
    - ▸ No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation
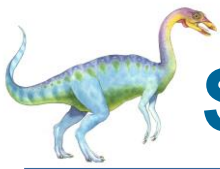
Shared Library Using Virtual Memory
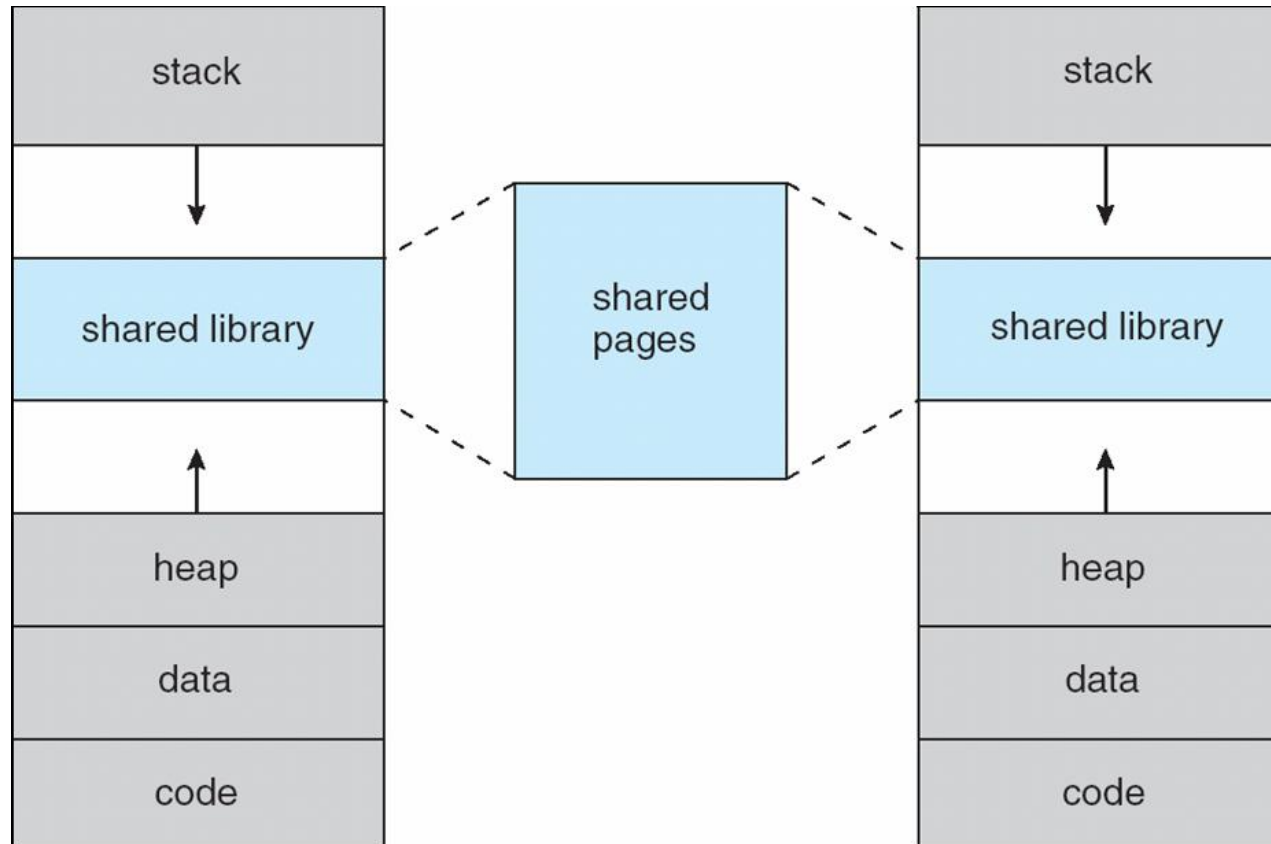
Demand Paging

- Could bring entire process into memory at load time

- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users

- Similar to paging system with swapping (diagram on right)

- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory

- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**
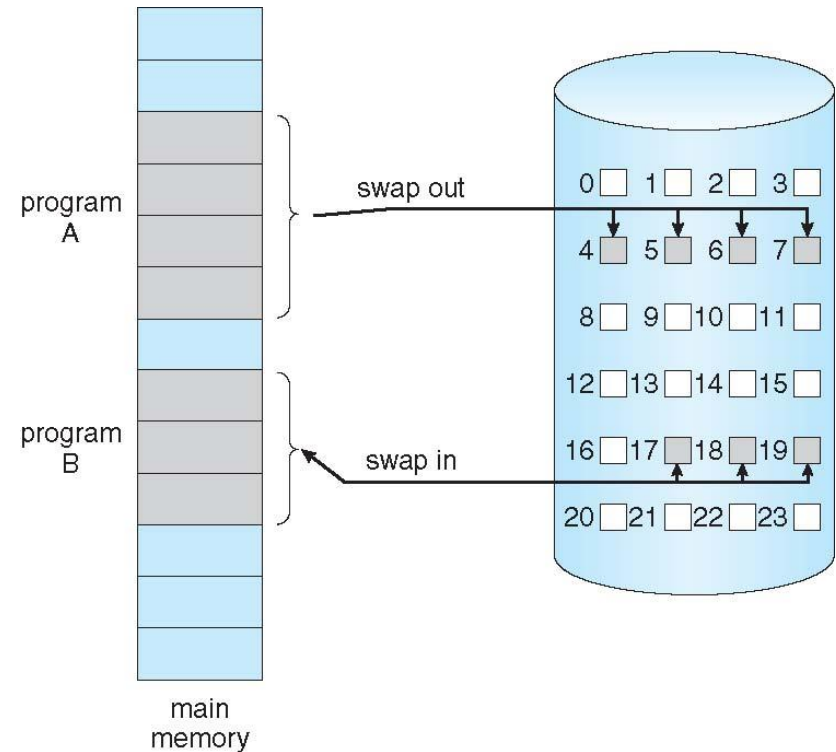
Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)

Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again

- Instead, pager brings in only those pages into memory

- How to determine that set of pages?

  - Need new MMU functionality to implement demand paging

- If pages needed are already **memory resident**

  - No difference from non demand-paging

- If page needed and not memory resident

  - Need to detect and load the page into memory from storage

    ▸ Without changing program behavior

    ▸ Without programmer needing to change code

Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  (**v** $\Rightarrow$ in-memory – **memory resident**, **i** $\Rightarrow$ not-in-memory)

- Initially valid–invalid bit is set to **i** on all entries

- Example of a page table snapshot:

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

page table

- During MMU address translation, if valid–invalid bit in page table entry
  is **i** $\Rightarrow$ page fault

Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to operating system

    - Page fault

2. Operating system looks at another table to decide:

    - Invalid reference ⇒ abort

    - Just not in memory

3. Find free frame

4. Swap page into frame via scheduled disk operation

5. Reset tables to indicate page now in memory
   Set validation bit = **v**

6. Restart the instruction that caused the page fault
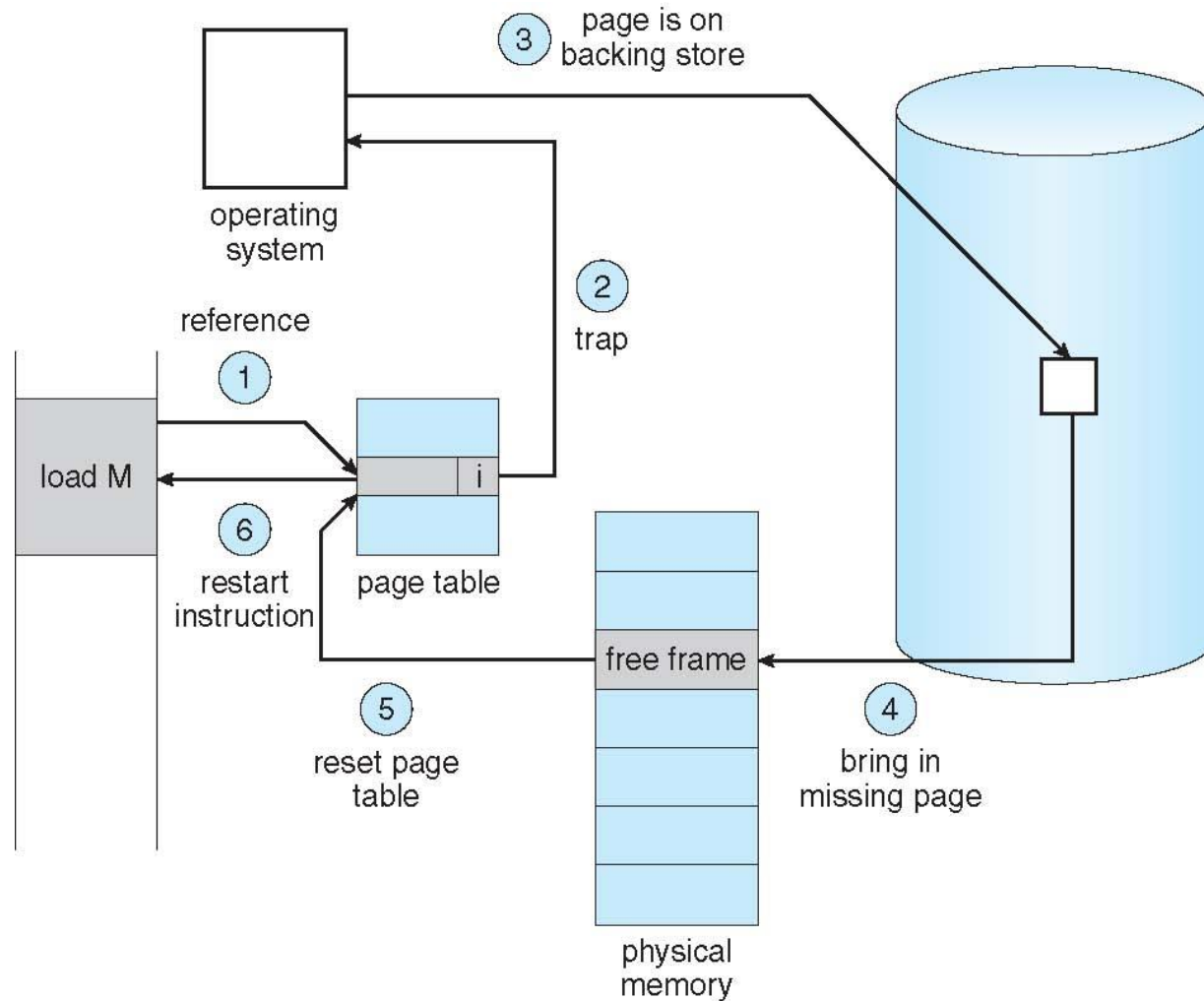
Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Instruction Restart

- Consider an instruction that could access several different locations
  - Block move
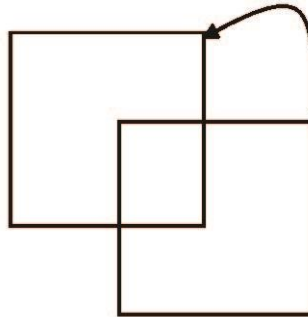
  - Auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?

Solutii pt restartarea instructiunilor

- depind de la un procesor la altul

- (1) PC-ul e copiat intr-un registru intern CPU inaintea executiei fiecarei instructiuni + registrele interne care salveaza incrementul/decrementul

- (2) microcodul CPU stocheaza starea interna pe stiva cand apare page-fault-ul

- (3) microcodul CPU are capacitate de roll-back la starea dinaintea executiei instructiunii (pp. existenta unui checkpoint intern)

Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.

- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.

head ⟶ 7 ⟶ 97 ⟶ 15 ⟶ 126 ... ⟶ 75
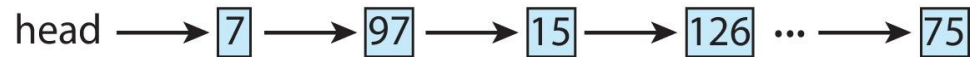
- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.

- When a system starts up, all available memory is placed on the free-frame list.

Stages in Demand Paging – Worse Case

1. Trap to the operating system

2. Save the user registers and process state

3. Determine that the interrupt was a page fault

4. Check that the page reference was legal and determine the location of the page on the disk

5. Issue a read from the disk to a free frame:

   a) Wait in a queue for this device until the read request is serviced

   b) Wait for the device seek and/or latency time

   c) Begin the transfer of the page to a free frame

Stages in Demand Paging (Cont.)

6.   While waiting, allocate the CPU to some other user

7.   Receive an interrupt from the disk I/O subsystem (I/O completed)

8.   Save the registers and process state for the other user

9.   Determine that the interrupt was from the disk

10. Correct the page table and other tables to show page is now in memory

11. Wait for the CPU to be allocated to this process again

12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging

- Three major activities

  - Service the interrupt – careful coding means just several hundred instructions needed

  - Read the page – lots of time

  - Restart the process – again just a small amount of time

- Page Fault Rate $0 \leq p \leq 1$

  - if $p = 0$ no page faults

  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

  EAT = $(1 - p)$ x memory access

  $+ p$ (page fault overhead

  + swap page out

  + swap page in)

Demand Paging Example

- Memory access time = 200 nanoseconds

- Average page-fault service time = 8 milliseconds

- EAT = (1 − p) x 200 + p (8 milliseconds)

  = (1 − p x 200 + p x 8,000,000

  = 200 + p x 7,999,800

- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!

- If want performance degradation < 10 percent

  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p

  - p < .0000025

  - < one page fault in every 400,000 memory accesses

Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system

- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix

- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system

- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially *share* the same pages in memory

  - If either process modifies a shared page, only then is the page copied

- COW allows more efficient process creation as only modified pages are copied

- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages

  - Pool should always have free frames for fast demand page execution

    - Don't want to have to free a frame as well as other processing on page fault

  - Why zero-out a page before allocating it?

- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent

  - Designed to have child call exec()

  - Very efficient

process₁   physical memory   process₂
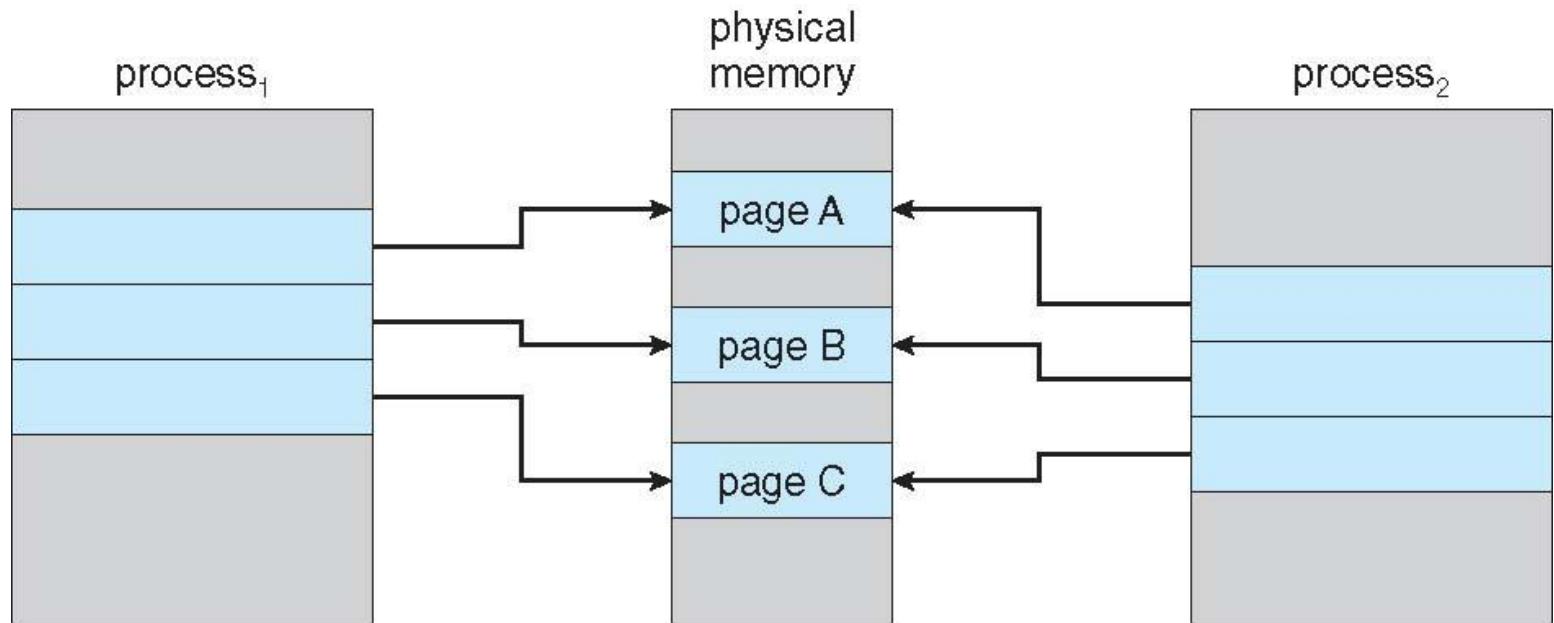
page A
page B
page C
Copy of page C
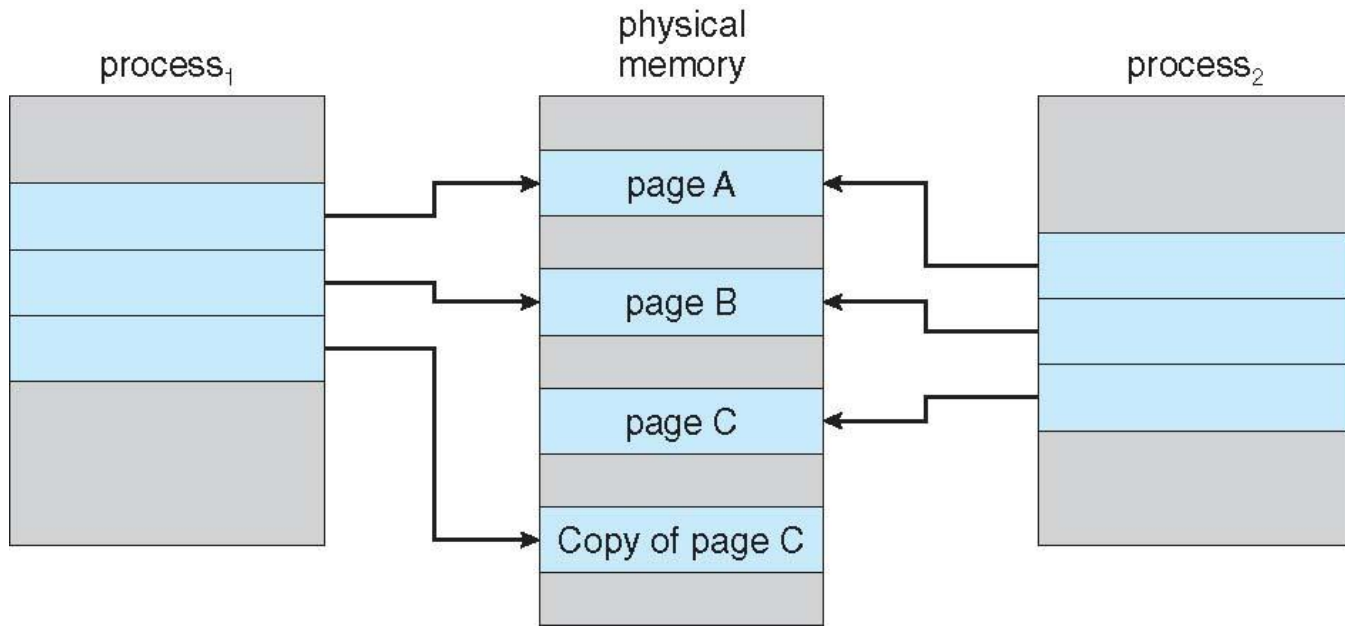
What Happens if There is no Free Frame?

- Used up by process pages

- Also in demand from the kernel, I/O buffers, etc

- How much to allocate to each?

- Page replacement – find some page in memory, but not really in use, page it out

  - Algorithm – terminate? swap out? replace the page?

  - Performance – want an algorithm which will result in minimum number of page faults

- Same page may be brought into memory several times

Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement

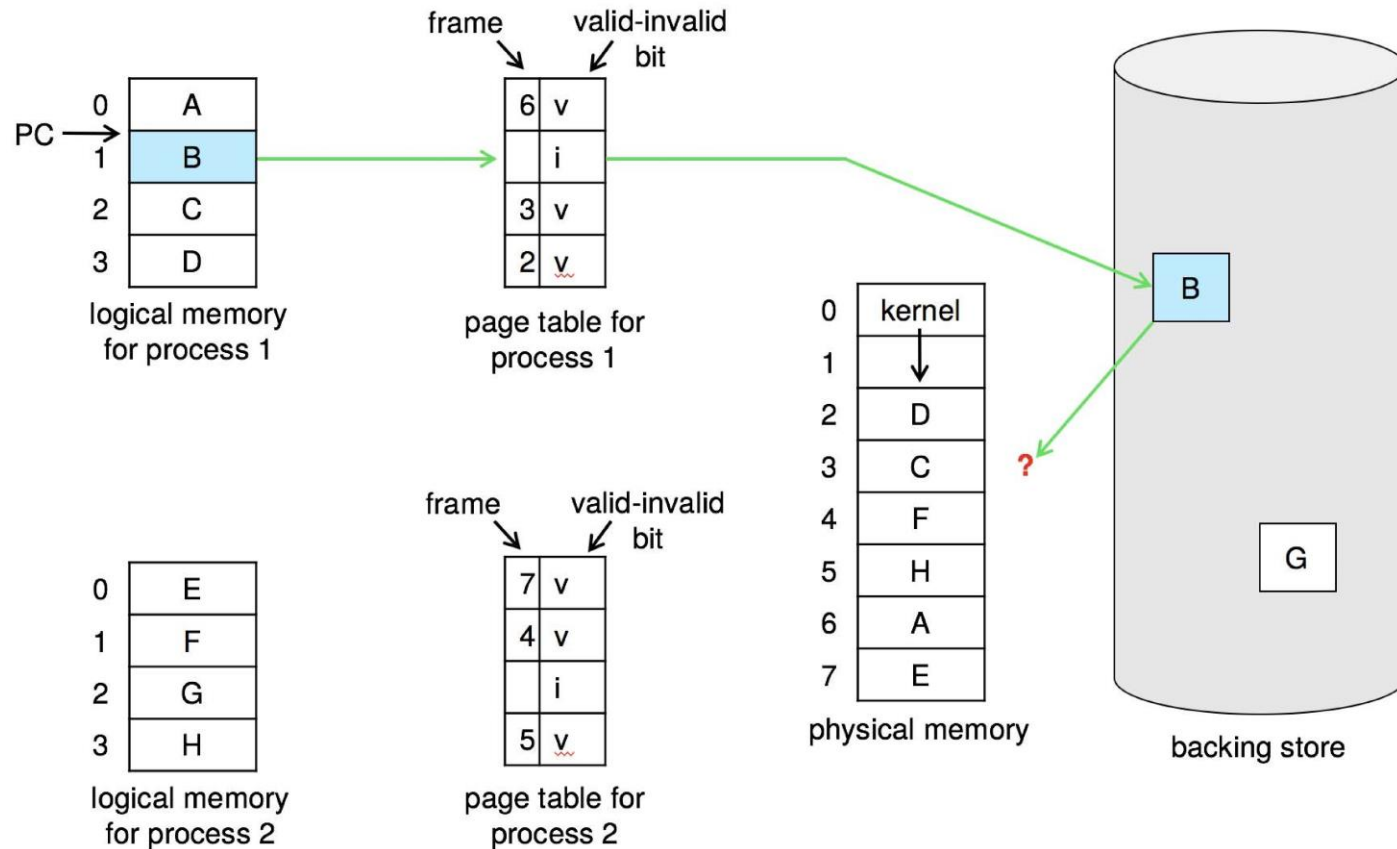- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk

- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

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**
        - Write victim frame to disk if dirty

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

4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement

- frame
- valid–invalid bit

- 0 | i
- f | v

(2) change to invalid

(4) reset page table for new page

page table

(1) swap out victim page

f | victim

physical memory

(3) swap desired page in
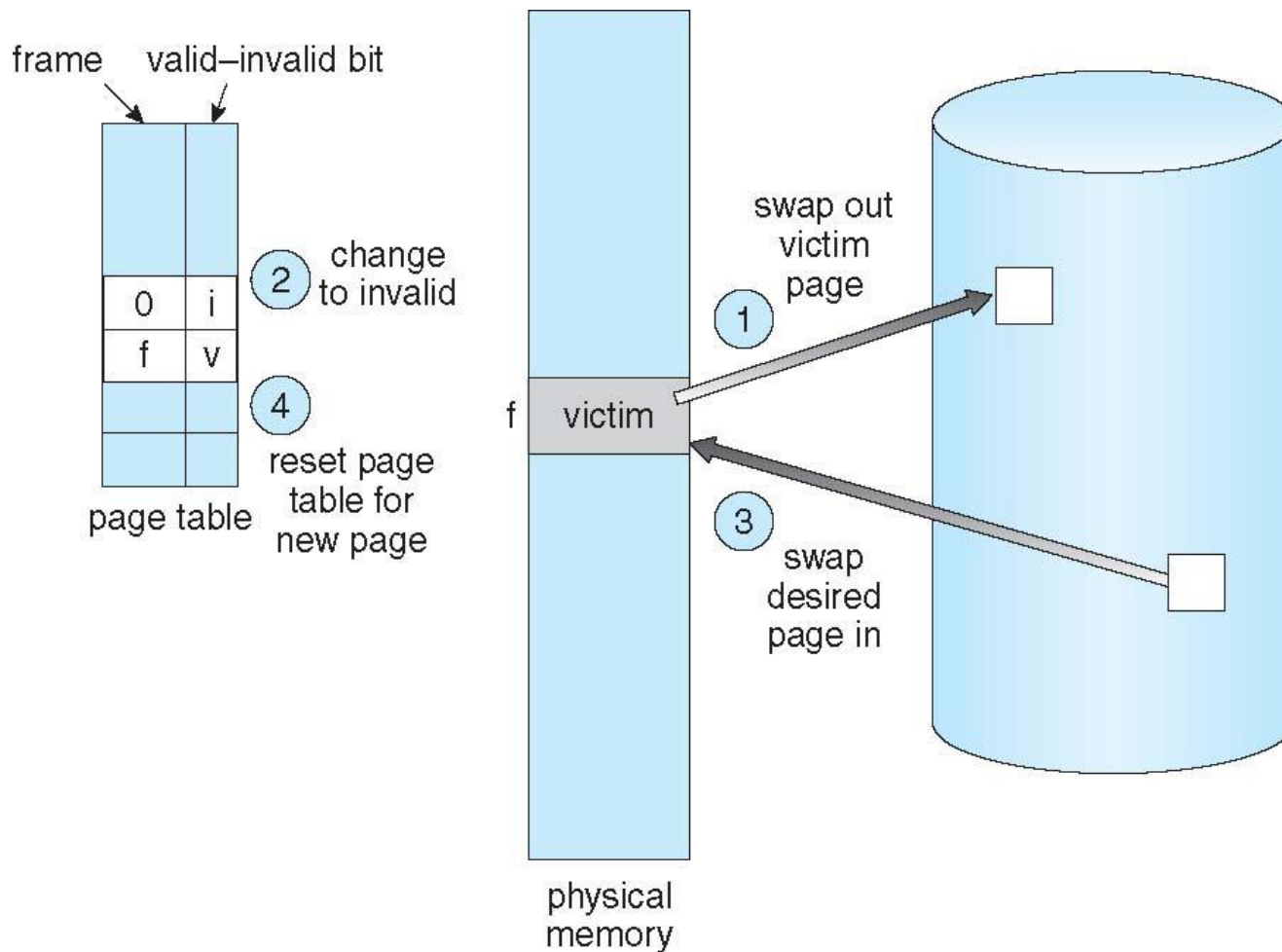
Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines

  - How many frames to give each process

  - Which frames to replace

- **Page-replacement algorithm**

  - Want lowest page-fault rate on both first access and re-access

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

  - String is just page numbers, not full addresses

  - Repeated access to the same page does not cause a page fault

  - Results depend on number of frames available

- In all our examples, the **reference string** of referenced page numbers is

  7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
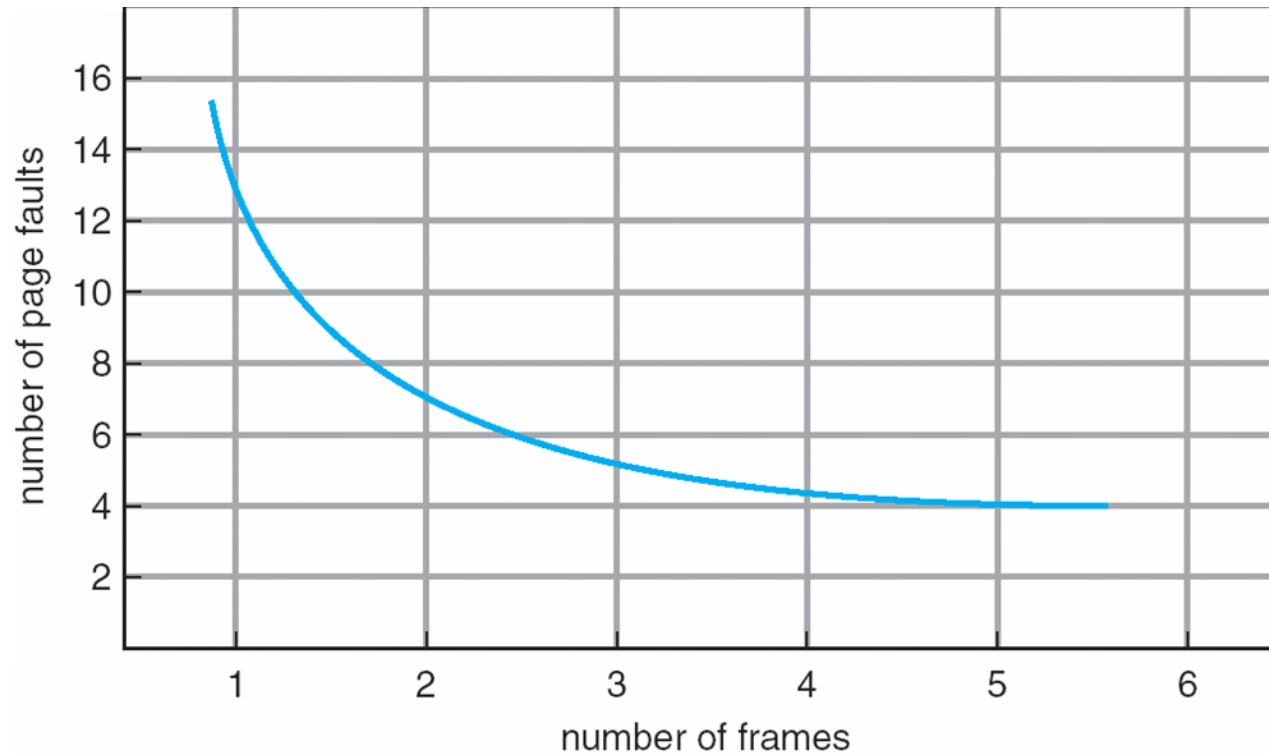
First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

- 3 frames (3 pages can be in memory at a time per process)

reference string

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

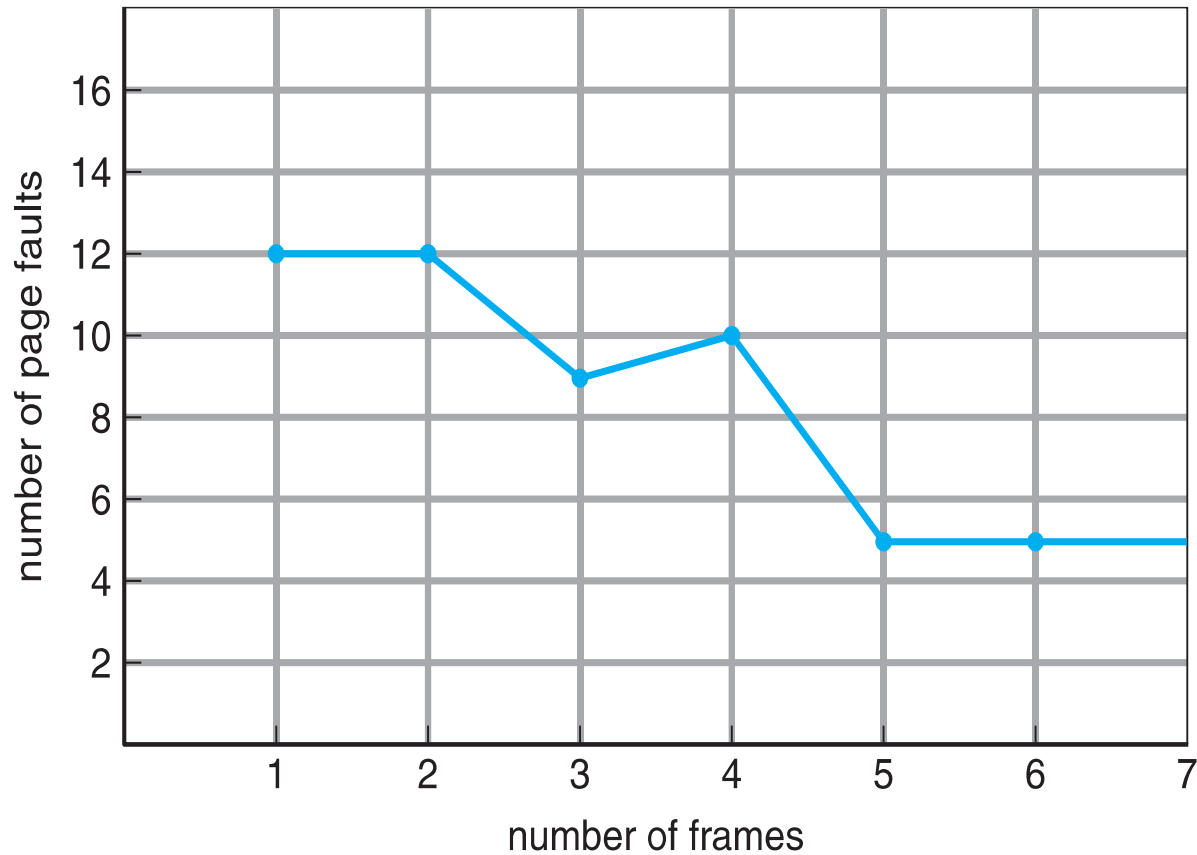| 7 | 7 | 7 | 2 | | 2 | 2 | 4 | 4 | 4 | 0 | | | 0 | 0 | | | 7 | 7 | 7 |
| | 0 | 0 | 0 | | 3 | 3 | 3 | 2 | 2 | 2 | | | 1 | 1 | | | 1 | 0 | 0 |
| | | 1 | 1 | | 1 | 0 | 0 | 0 | 3 | 3 | | | 3 | 2 | | | 2 | 2 | 1 |

page frames

15 page faults

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5

  - Adding more frames can cause more page faults!

    ▸ **Belady's Anomaly**

- How to track ages of pages?

  - Just use a FIFO queue

Explicatie anomalia lui Belady

- FIFO nu e un algoritm de tip stiva (*stack algorithm*)

  - algoritmi pt care M(m,r) ⊆ M(m+1,r)

  unde M(m,r) = multimea frame-urilor dintr-o memorie de dimensiune *m* (nr. total de frame-uri) dupa un nr. *r* de referinte la memorie

  - la marirea dimensiunii memoriei cu un frame si rexecutarea procesului, la orice moment in timp, toate paginile care erau in memorie dupa *r* referinte in cazul memoriei mai mici sunt si acum in memorie **plus** o pagina aditionala

- ex algoritmi stiva: OPT (algoritmul optimal), LRU (Least Recently Used)

- pt. algoritmii stiva e convenabil sa se reprezinte secventa de referinte la memorie *prin distanta fata de varful stivei* a locului paginii *inainte* de referinta

- paginile nereferite inca nu sunt in stiva si au distanta ∞

Algoritmi stiva

- secventa de distante fata de varful stivei nu depinde doar de secventa de referinte de memorie, ci si de algoritmul de paginare

- proprietatile statistice ale secventei de distante induc performanta algoritmului de paginare

  - daca densitatea de probabilitate a intrarilor din secventa de distante e "stransa" (i.e., majoritatea distantelor < K) => K frame-uri sunt suficiente pt o rata redusa de page-fault-uri

  - daca, denstitatea de probabilitate e o functie "plata" => singura solutie pt a evita un nr mare de page-fault-uri este sa ai tot atatea frame-uri cate pagini sunt

Predictia ratei de page-fault-uri

- secventa de distante poate fi folosita la predictia ratei de page-fault-uri atunci cand variaza nr de pagini *n*

- algoritm:

  - scaneaza secventa de distante si calculeaza

    C_i = nr de aparitii ale distantei *i* in secventa

    C_∞ = nr de aparitii ale distantei ∞ in secventa

  - calculeaza vectorul F

$$F_m = \sum_{k=m+1}^{n} C_k + C_\infty$$

adica nr de page-fault-uri care apar datorita secventei de distante intr-o memorie cu *m* frame-uri

Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

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

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future

- Replace page that has not been used in the most amount of time

- Capitalizes on the locality of reference principle

- Associate time of last use with each page

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

- 12 faults – better than FIFO but worse than OPT

- Generally good algorithm and frequently used

- But how to implement?

LRU Algorithm (Cont.)

- Counter implementation

  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter

  - When a page needs to be changed, look at the counters to find smallest value

    ▸ Search through table needed

- 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

  - But each update more expensive

  - No search for replacement

LRU Algorithm (Cont.)

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

- Use Of A Stack to Record Most Recent Page References

reference string

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

2
1
0
7
4

stack
before
a

7
2
1
0
4

stack
after
b

LRU Approximation Algorithms

- LRU needs special hardware and still slow

- **Reference bit**

    - With each page associate a bit, initially = 0

    - When page is referenced bit set to 1

    - Replace any with reference bit = 0 (if one exists)

        ‣ We do not know the order, however
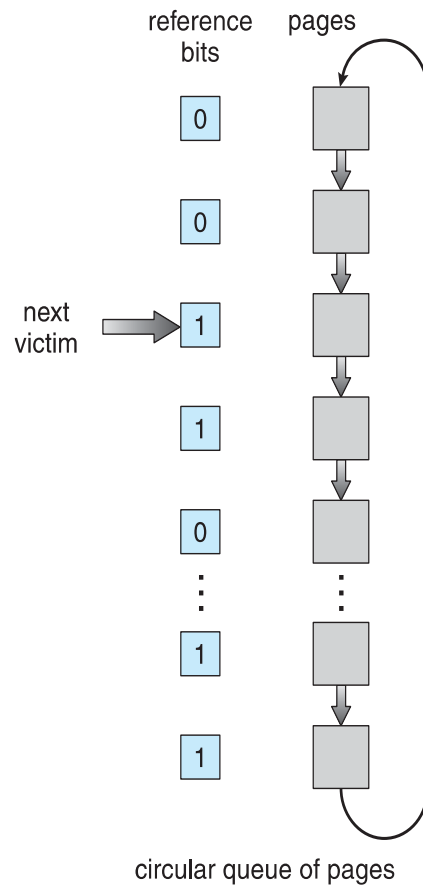
LRU Approximation Algorithms (cont.)

- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules
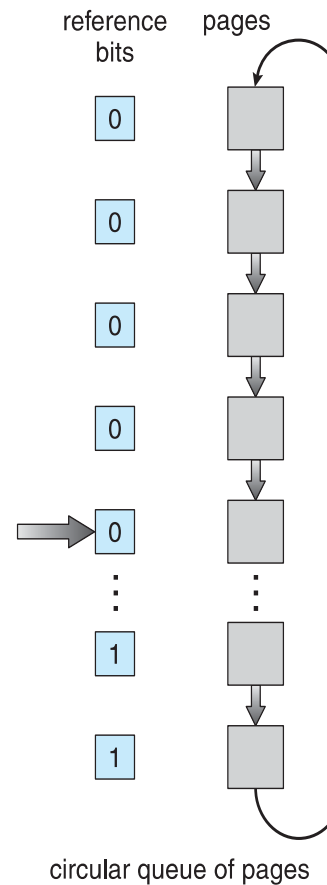
Second-chance Algorithm

reference bits | pages

next victim → 0, 0, **1**, 1, 0, ⋮, 1, 1

circular queue of pages

(a)

reference bits | pages

0, 0, 0, 0, → 0, ⋮, 1, 1

circular queue of pages

(b)

Enhanced Second-Chance Algorithm

- Sometimes also called Not Recently Used (NRU)

- Improve algorithm by using reference bit and modify bit (if available) in concert

- Algorithm starts with R=0 and M=0, periodically the R bits are set to 0

- Take ordered pair (reference, modify):

  - (0, 0) neither recently used not modified – best page to replace

  - (0, 1) not recently used but modified – not quite as good, must write out before replacement

  - (1, 0) recently used but clean – probably will be used again soon

  - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class

  - Might need to search circular queue several times

Counting Algorithms

- Keep a counter of the number of references that have been made to each page (basically add the R bit, on each clock tick)

  - Not common

- **Least Frequently Used** (**LFU**) **Algorithm (sometimes also called Not Frequently Used, NFU)**:

  - Replaces page with smallest count

  - doesn't properly reflect locality of reference (long-time ago accessed frames are kept in memory, while recently and frequently accessed frames risk eviction, "NFU doesn't forget anything")

- **Most Frequently Used** (**MFU**) **Algorithm**:

  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Aging

- ajuta NFU sa reflecte mai corect localitatea temporala si sa aproximeze mai bine LRU

- la fiecare tact de ceas, contorul frame-ului e shiftat la dreapta cu 1 bit, iar R se aduna la cel mai semnificativ bit al rezultatului shiftarii

  $\Rightarrow$ shiftarea micsoreaza ponderea acceselor departate in timp, iar adunarea lui R tinde sa actualizeze corect gradul de utilizare recenta al paginii

- la page-fault se elimina frame-ul cu contorul cel mai mic

- diferente fata de LRU

  - la eliminarea a doua pagini cu acelasi prefix (eg 001) nu stim care pagina a fost referita ultima intre tactul 1 si 2

  - contorul frame-ului are reprezentare limitata (eg 8 biti) => nu se distinge intre doua pagini cu contor 0, desi una poate a fost referita mai recent decat alta

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access

- Some applications have better knowledge – i.e. databases

- Memory intensive applications can cause double buffering

  - OS keeps copy of page in memory as I/O buffer

  - Application keeps page in memory for its own work

- Operating system can given direct access to the disk, getting out of the way of the applications

  - **Raw disk** mode

- Bypasses buffering, locking, etc.

Allocation of Frames

- Each process needs *minimum* number of frames
- 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*
- *Maximum* of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames

  - Keep some as free frame buffer pool

- Proportional allocation – Allocate according to the size of process

  - Dynamic as degree of multiprogramming, process sizes change

  - Prioritatea poate influenta fractiunea de pagini alocate

  - s_i = size of process p_i

  - $S = \sum s_i$

  - m = total number of frames

  - a_i = allocation for $p_i = \dfrac{s_i}{S} \times m$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 62 \approx 4$$
$$a_2 = \frac{127}{137} \times 62 \approx 57$$

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
  - But then process execution time can vary greatly
    - rata de page-fault-uri a procesului nu depinde doar de comportamentul sau, ci si de cele ale altor procese
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
    - rata de page-fault-uri depinde exclusiv de comportamentul procesului
 - But possibly underutilized memory

Reclaiming Pages

- A strategy to implement global page-replacement policy

- All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement,

- Page replacement is triggered when the list falls below a certain threshold.

- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.
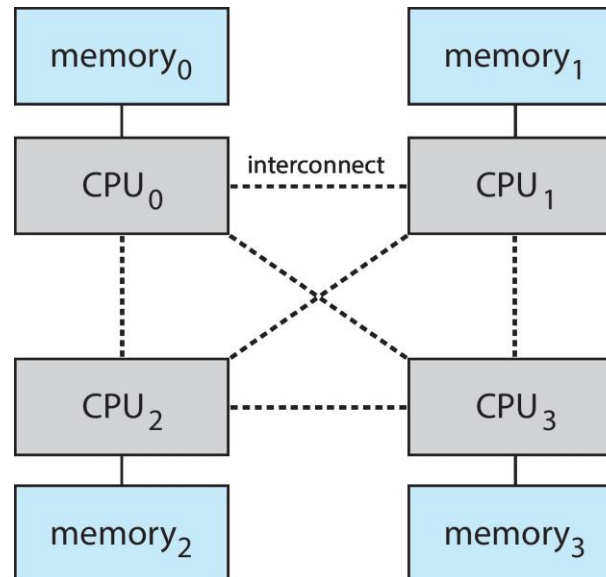
Reclaiming Pages Example
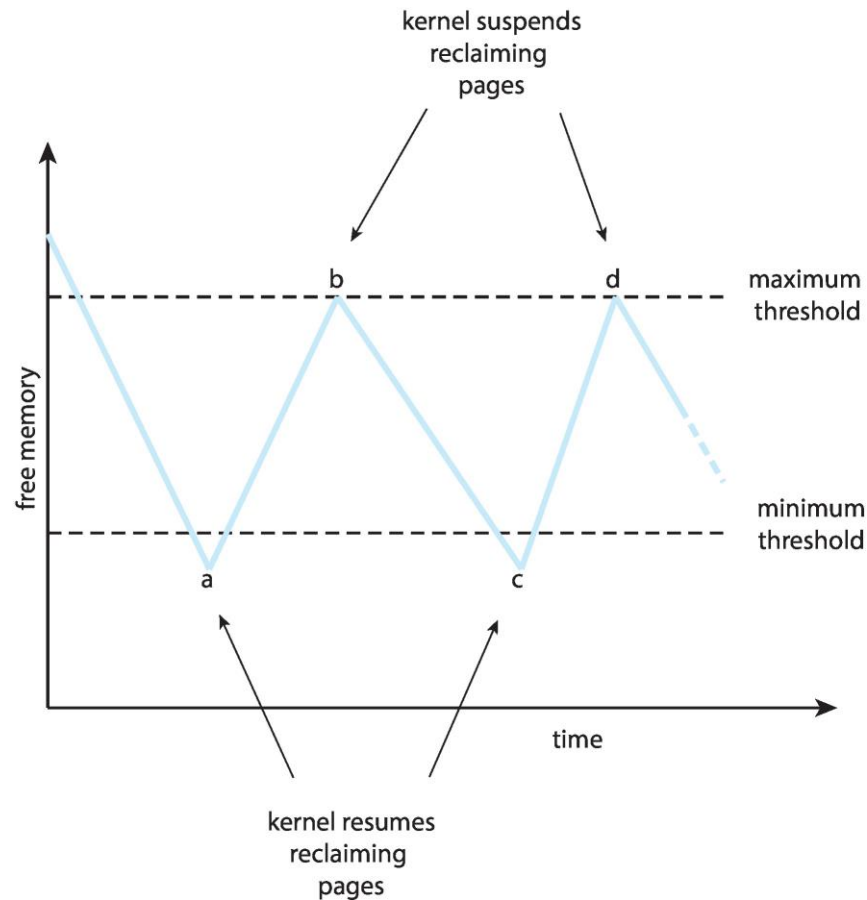
Non-Uniform Memory Access

- So far, we assumed that all memory accessed equally

- Many systems are **NUMA** – speed of access to memory varies

  - Consider system boards containing CPUs and memory, interconnected over a system bus

- NUMA multiprocessing architecture

```
  ┌──────────┐              ┌──────────┐
  │ memory_0 │              │ memory_1 │
  └────┬─────┘              └────┬─────┘
  ┌────┴─────┐  interconnect ┌───┴──────┐
  │  CPU_0   │- - - - - - - -│  CPU_1   │
  └────┬─────┘              └────┬─────┘
       │     ╲            ╱      │
       │      ╲          ╱       │
       │       ╲        ╱        │
  ┌────┴─────┐  ╲      ╱   ┌──────┴───┐
  │  CPU_2   │- - - - - - -│  CPU_3   │
  └────┬─────┘             └────┬─────┘
  ┌────┴─────┐             ┌────┴─────┐
  │ memory_2 │             │ memory_3 │
  └──────────┘             └──────────┘
```

Non-Uniform Memory Access (Cont.)

- Optimal performance comes from allocating memory "close to" the CPU on which the process/thread is scheduled

  - And modifying the scheduler to schedule the thread on the same system board when possible

  - Alternativ, utilizatorul poate seta afinitatea de procesor (soft/hard)

  - Consecinta: cache hit ratio marit si timp de acces la memorie redus

  - For threads, solved by Solaris by creating **lgroups**

    ▸ Structure to track CPU / Memory low latency groups

    ▸ Used my schedule and pager

    ▸ When possible schedule all threads of a process and allocate all memory for that process within the lgroup
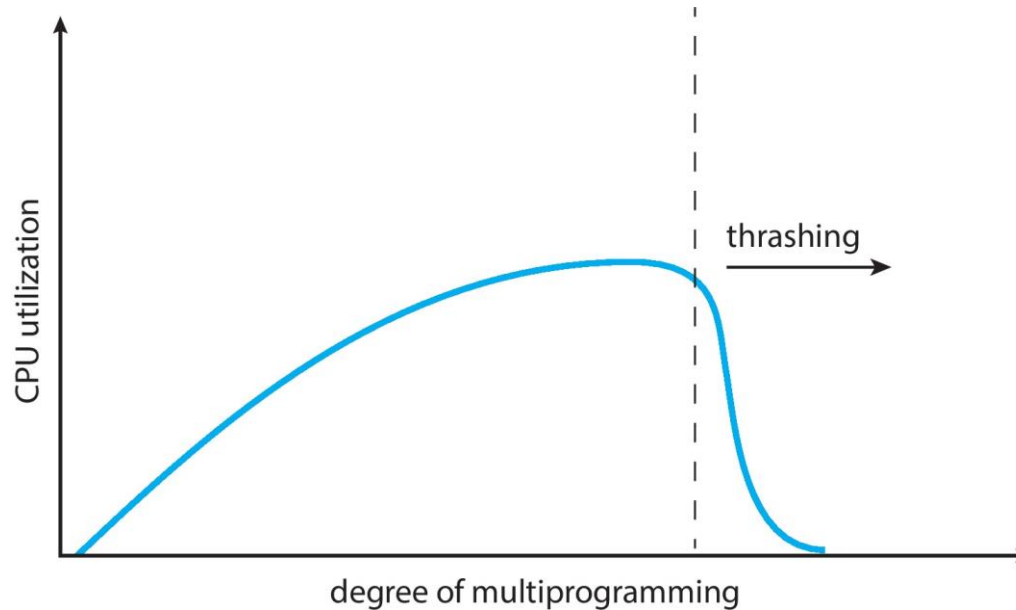
Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high

  - Page fault to get page

  - Replace existing frame

  - But quickly need replaced frame back

  - This leads to:

    - ▸ Low CPU utilization

    - ▸ Operating system thinking that it needs to increase the degree of multiprogramming

    - ▸ Another process added to the system

Thrashing (Cont.)

- **Thrashing.** A process is busy swapping pages in and out

Demand Paging and Thrashing

- Why does demand paging work?

  Locality model

  - Process migrates from one locality to another

  - Definita ca seturi de pagini care se folosesc impreuna la un moment dat

    ‣ Ex: apel de functie => localitatea se refera la accesele la instructiunile functiei, variabilele locale (stiva) si subsetul de date globale (data)

    ‣ La iesirea din functie se paraseste localitatea

  - Localities may overlap (ex: datele globale pot apartine mai multor localitati)

- Why does thrashing occur?

  Σ size of locality > total memory size

- Limit effects by using local or priority page replacement
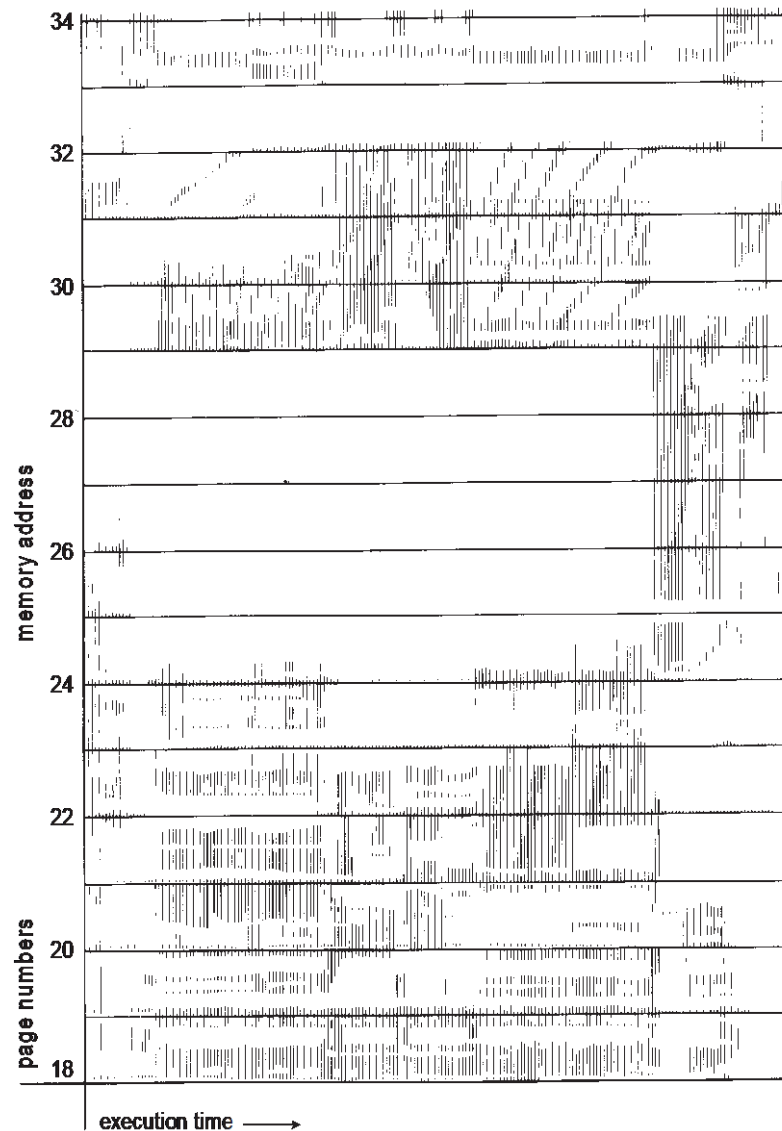
Working-Set Model

- Pp. alocam suficiente frame-uri pt localitatea curenta => nr limitat de page-fault-uri pana cand se schimba localitatea din nou

- Model bazat pe localitatea de referinta

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references Example: 10,000 instructions
  - Idee: se examineaza cele mai recente Δ referinte la pagini
  - Daca pagina e activa, va fi in working set, altfel este eliminata

- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program

- $D = \Sigma \ WSS_i \equiv$ total demand frames
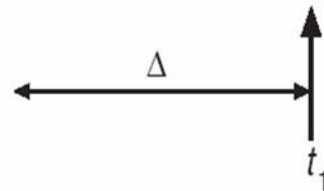 - Approximation of locality
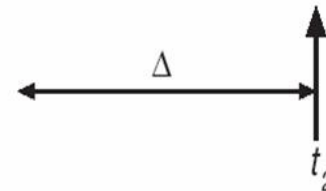
Working-Set Model (Cont.)

- if $D > m \Rightarrow$ Thrashing

- Policy if $D > m$, then suspend or swap out one of the processes

page reference table

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

$WS(t_1) = \{1,2,5,6,7\}$ $WS(t_2) = \{3,4\}$

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit

- Example: Δ = 10,000

  - Timer interrupts after every 5000 time units

  - Keep in memory 2 bits for each page

  - Whenever a timer interrupts, copy and sets the values of all reference bits to 0

  - If one of the bits in memory = 1 $\Rightarrow$ page in working set

- Why is this not completely accurate?

- Improvement = 10 bits and interrupt every 1000 time units

 - Dezavantaj: nr crescut de intreruperi si overheadul aferent

Page-Fault Frequency

- More direct approach than WSS

- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy

  - If actual rate too low, process loses frame

  - If actual rate too high, process gains frame
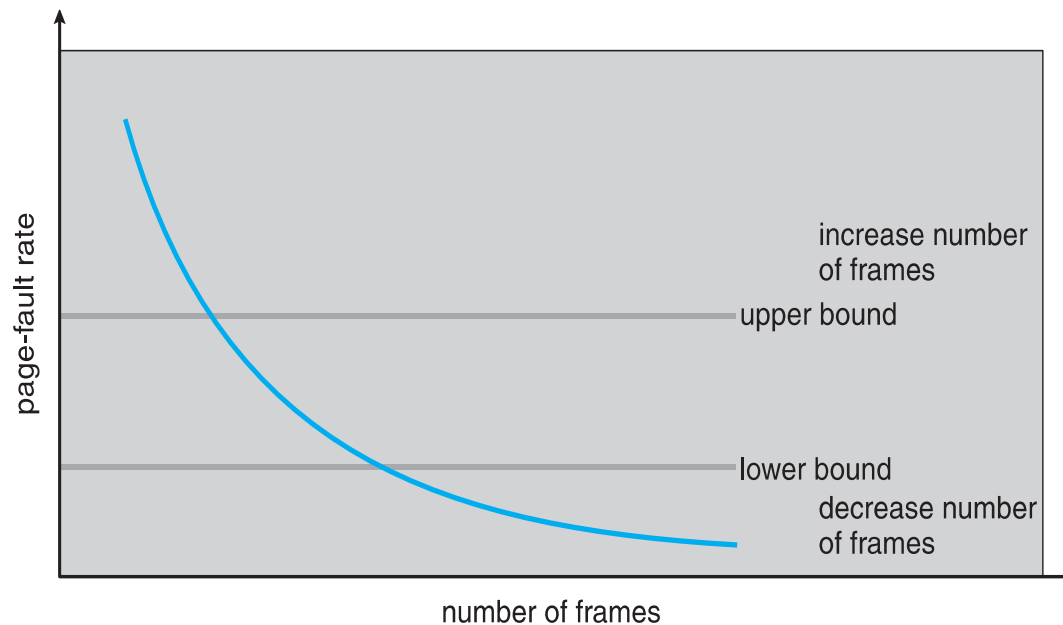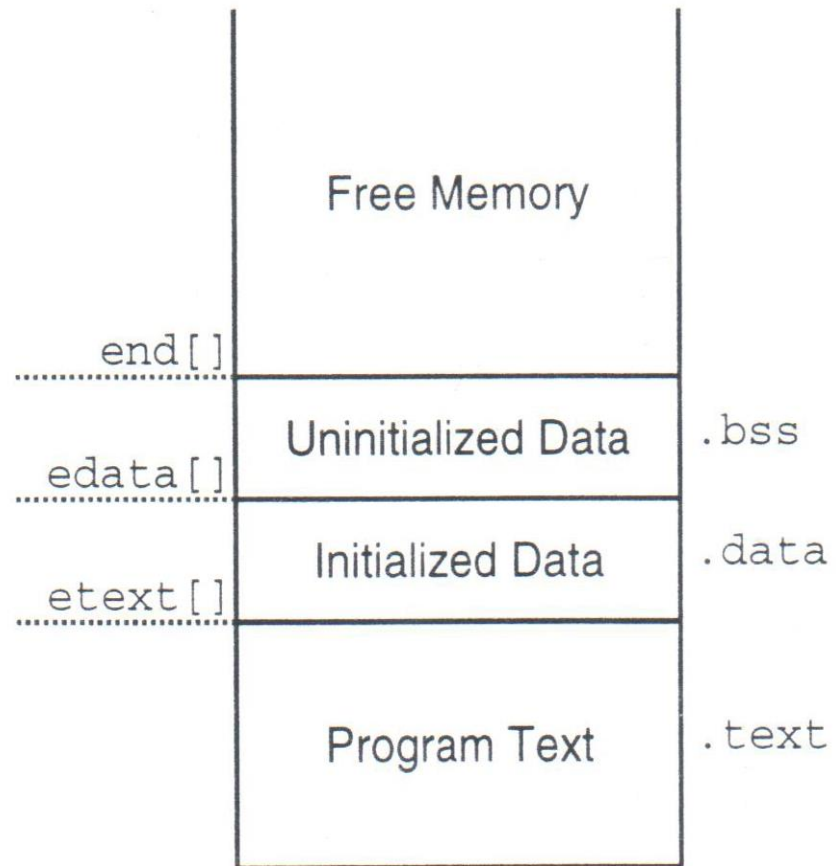
Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate

- Working set changes over time

- Peaks and valleys over time
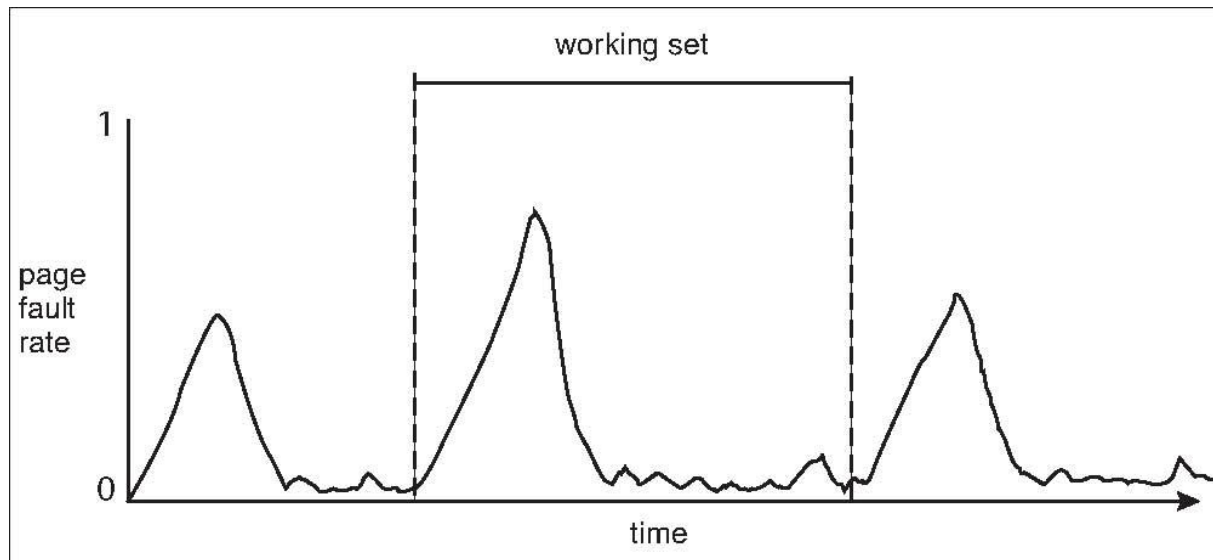
Alocarea memoriei kernel

- harta memoriei dupa ce loaderul a incarcat imaginea kernelului

- editorul de legaturi a generat simboluri care delimiteaza sectiunile imaginii kernel: _etext, _edata, _end

- codul de initializare a sistemului poate folosi _end pt a localiza adresa de start a memoriei libere

- memoria disponibila se afla
  - fie ca valoare pasata codului de bootstrap
  - fie proband adresele dincolo de _end

Free Memory	
end [ ]	
Uninitialized Data	.bss
edata [ ]	
Initialized Data	.data
etext [ ]	
Program Text	.text

Alocare memoriei kernel (cont'd)

- memoria disponibila e impartita in frames, si pusa in free list

  - paginile sunt gestionate de un alocator de pagini (*page-level allocator*)

- in timpul executiei, cand kernelul are nevoie de memorie pt structurile sale de date interne (PCB-uri, buffere de retea, buffere pt blocuri de disc, samd) cere o pagina (sau mai multe) alocatorului de pagini

- structurile de date ale kernelului pot fi adresate virtual, dar nu sunt swapabile

  - i.e., nu exista page-fault pt paginile kernel

  - page-fault-urile ar fi critice pe durata tratarii intreruperilor sau in timpul sectiunilor critice

Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - i.e., for device I/O

Alocatorul de memorie kernel

- Kernel Memory Allocator (KMA)

- componenta a sistemului de operare care obtine pagini de la alocatorul de pagini si le foloseste pt satisfacerea cererilor de memorie neswapabila ale kernelului

- trebuie sa fie rapid

  - stiva kernel e mica si echivalentul variabilelor automatice din programe foloseste memoria KMA

  - KMA e apelat din handlere de intrerupere care au constrangeri critice de timp

  - KMA lent => degradarea performantei intregului sistem

- trebuie sa aiba interfata de programare simpla

  - de regula, memoria se aloca intr-o parte a kernelului si se dealoca in alta => free() trebuie proiectat a.i. sa nu stie dimensiunea alocarii

Alocatorul de memorie kernel (cont'd)

- trebuie sa fie eficient

  - sa aiba factor de utilizare mare(ideal 100%, in practica 50% e acceptabil)

  - factor de utilizare: $\dfrac{memoria\ totala\ ceruta}{memoria\ totala\ necesara\ pt\ a\ satisface\ cererile}$

- trebuie sa fie capabil sa obtina pagini noi de la alocatorul de pagini cand free list e goala, respectiv sa dea inapoi pagini atunci cand nu mai sunt folosite

- trebuie sa fie capabil sa compacteze blocurile nealocate adiacente in vederea satisfacerii unor cereri viitoare de dimensiuni mari
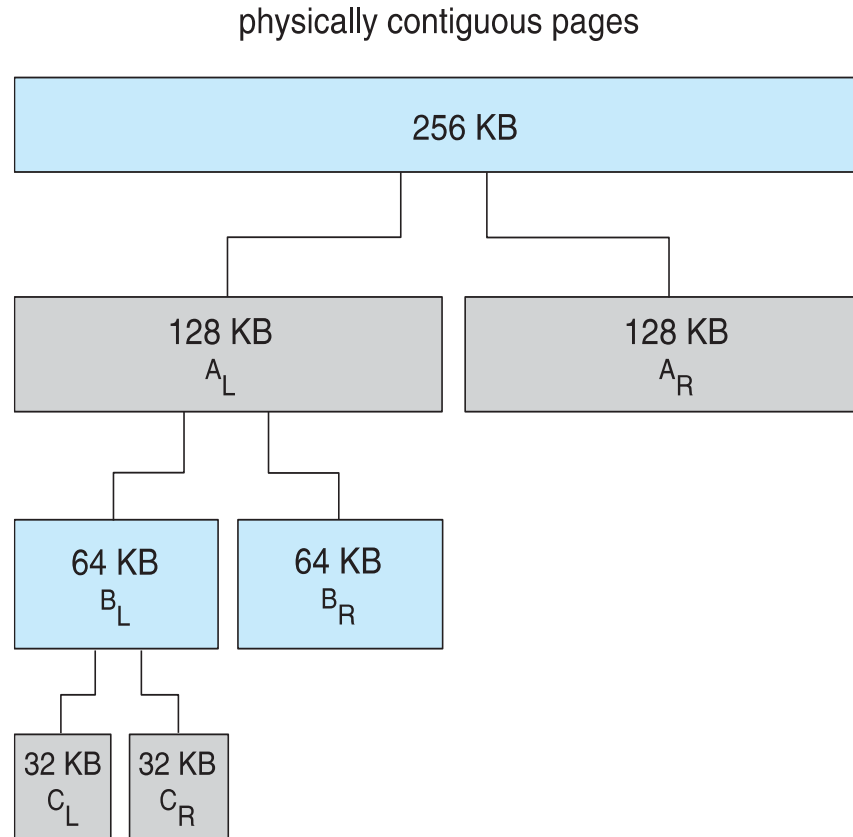
Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages

- Memory allocated using **power-of-2 allocator**

  - Satisfies requests in units sized as power of 2

  - Request rounded up to next highest power of 2

  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2

    - Continue until appropriate sized chunk available

- For example, assume 256KB chunk available, kernel requests 21KB

  - Split into A_{L} and A_{R} of 128KB each

    - One further divided into B_{L} and B_{R} of 64KB

      - One further into C_{L} and C_{R} of 32KB each – one used to satisfy request

- Advantage – quickly **coalesce** unused chunks into larger chunk

- Disadvantage - fragmentation

Buddy System Allocator

physically contiguous pages

```
                    256 KB

        128 KB                  128 KB
         A_L                     A_R

    64 KB       64 KB
     B_L         B_R

  32 KB  32 KB
   C_L    C_R
```

Implementare Buddy System

- managerul de memorie mentine *free lists* pt toate puterile lui 2 pana la dimensiunea memoriei

- alocarea

  - rotunjita la urmatoarea putere a lui 2

  - daca nu exista in free lists segmentul corespunzator se merge la puteri mai mari; blocurile mari alocate se sparg succesiv in 2 buddies

- dealocarea

  - rotunjita la urmatoarea putere a lui 2

  - daca segmentul dealocat are un buddy adiacent, se compacteaza intr-un segment mai mare; daca segmental rezultat are si el un buddy, se compacteaza in continuare,samd

- regula de detectare buddies:

  B_1, B_2 buddies $\Leftrightarrow$ size(B_1) = size(B_2) = 2^k && $2^{k+1}|$ adr$(B_1 + B_2)$

Exemplu Buddy System

Figure 5: Buddy system operation.

Buddy System, pros & cons

- avantaje

  - dealocare rapida (segmentele compactabile se identifica prin adresa si buddies nu se cauta decat in lista segmentelor cu aceeasi dimensiune ca si segmentul eliberat)

- dezavantaje

  - fragmentare interna (datorita rotunjirii de la alocare)

  - fragmentare externa

    ex: dupa alocarea celor 1536 bytes, o cerere de 5KB nu poate fi satisfacuta, desi exista 5KB liberi in total
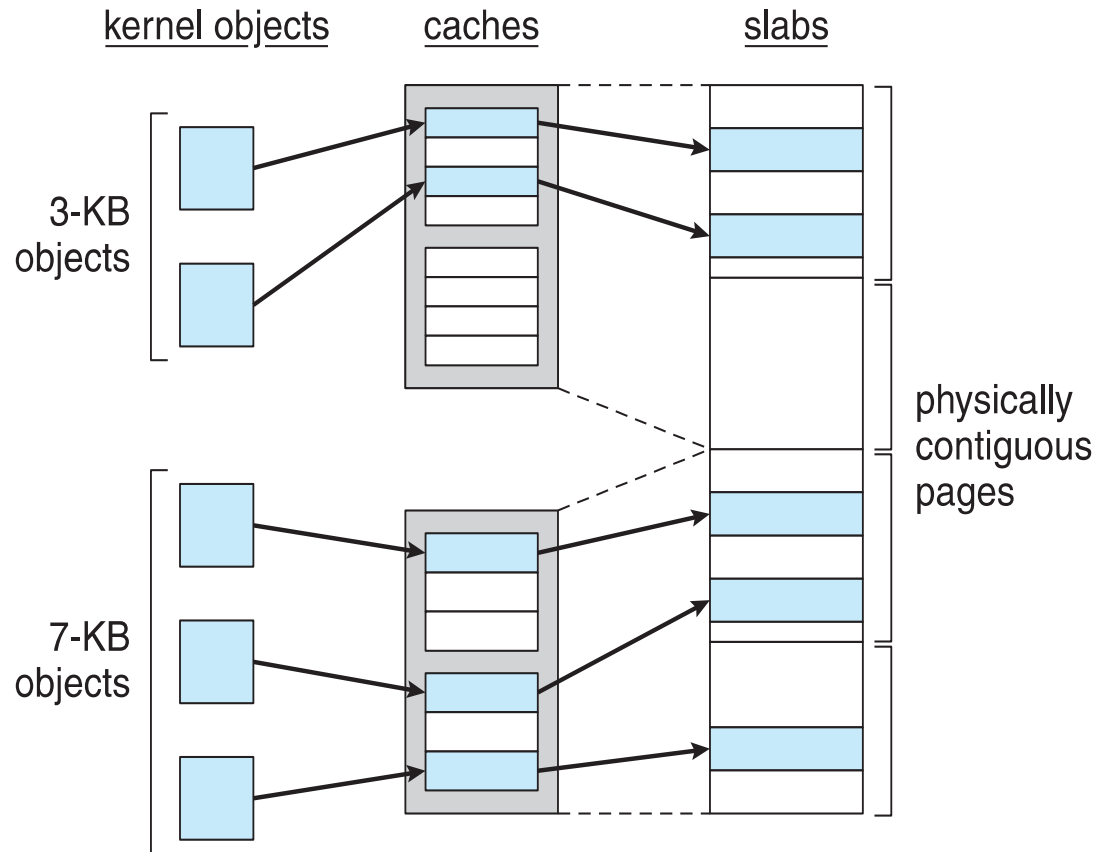
Slab Allocator

- Alternate strategy

- **Slab** is one or more physically contiguous pages

- **Cache** consists of one or more slabs

- Single cache for each unique kernel data structure

  - Each cache filled with **objects** – instantiations of the data structure

- When cache created, filled with objects marked as `free`

- When structures stored, objects marked as `used`

- If slab is full of used objects, next object allocated from empty slab

  - If no empty slabs, new slab allocated

- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation

kernel objects | caches | slabs

3-KB objects

7-KB objects

physically contiguous pages

Slab Allocator in Linux

- For example process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty

Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes

- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators

  - SLOB for systems with limited memory

    - Simple List of Blocks – maintains 3 list objects for small, medium, large objects

  - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

Other Considerations

- Prepaging

- Page size

- TLB reach

- Inverted page table

- Program structure

- I/O interlock and page locking

Prepaging

- To reduce the large number of page faults that occurs at process startup

- Prepage all or some of the pages a process will need, before they are referenced

- But if prepaged pages are unused, I/O and memory was wasted

- Assume s pages are prepaged and α of the pages is used

    - Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging $s * (1- \alpha)$ unnecessary pages?

    - α near zero $\Rightarrow$ prepaging loses

Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

TLB Reach

- TLB Reach - The amount of memory accessible from the TLB

- TLB Reach = (TLB Size) X (Page Size)

- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults

- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size

- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Program Structure

- Program structure
  - `int[128,128] data;`
  - Each row is stored in one page
  - Program 1

    ```
    for (j = 0; j <128; j++)
        for (i = 0; i < 128; i++)
            data[i,j] = 0;
    ```

    128 x 128 = 16,384 page faults

  - Program 2

    ```
    for (i = 0; i < 128; i++)
        for (j = 0; j < 128; j++)
            data[i,j] = 0;
    ```
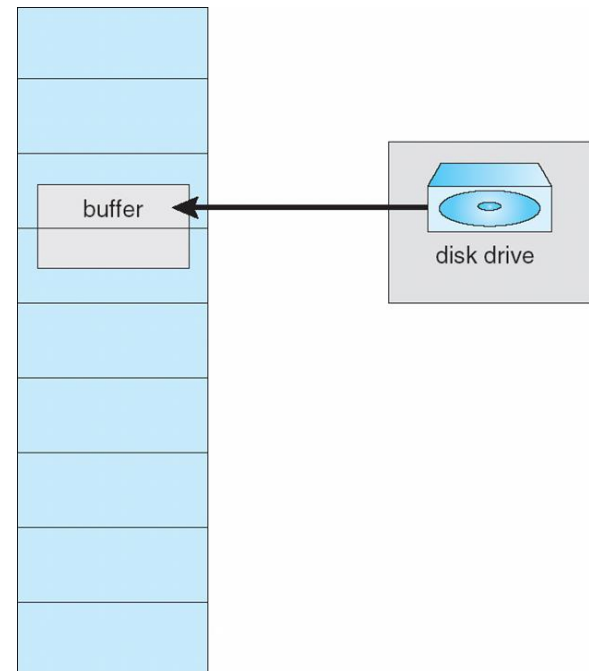
    128 page faults

I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory

- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm

- **Pinning** of pages to lock into memory

Operating System Examples

- Windows

- Solaris

Windows

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page

- Processes are assigned **working set minimum** and **working set maximum**

- Working set minimum is the minimum number of pages the process is guaranteed to have in memory

- A process may be assigned as many pages up to its working set maximum

- When the amount of free memory in the system falls below a threshold**, automatic working set trimming** is performed to restore the amount of free memory

- Working set trimming removes pages from processes that have pages in excess of their working set minimum

Solaris

- Maintains a list of free pages to assign faulting processes
- `Lotsfree` – threshold parameter (amount of free memory) to begin paging
- `Desfree` – threshold parameter to increasing paging
- `Minfree` – threshold parameter to being swapping
- Paging is performed by `pageout` process
- `Pageout` scans pages using modified clock algorithm
- `Scanrate` is the rate at which pages are scanned. This ranges from `slowscan` to `fastscan`
- `Pageout` is called more frequently depending upon the amount of free memory available
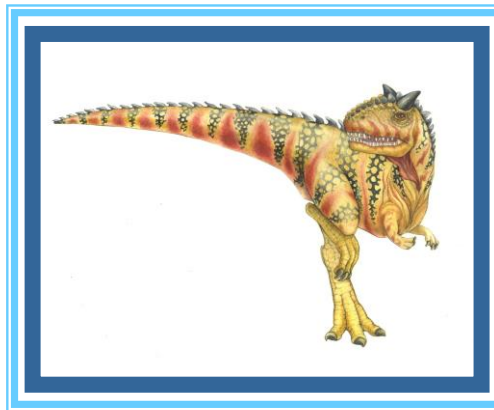- *Priority paging* gives priority to process code pages

# Solaris 2 Page Scanner
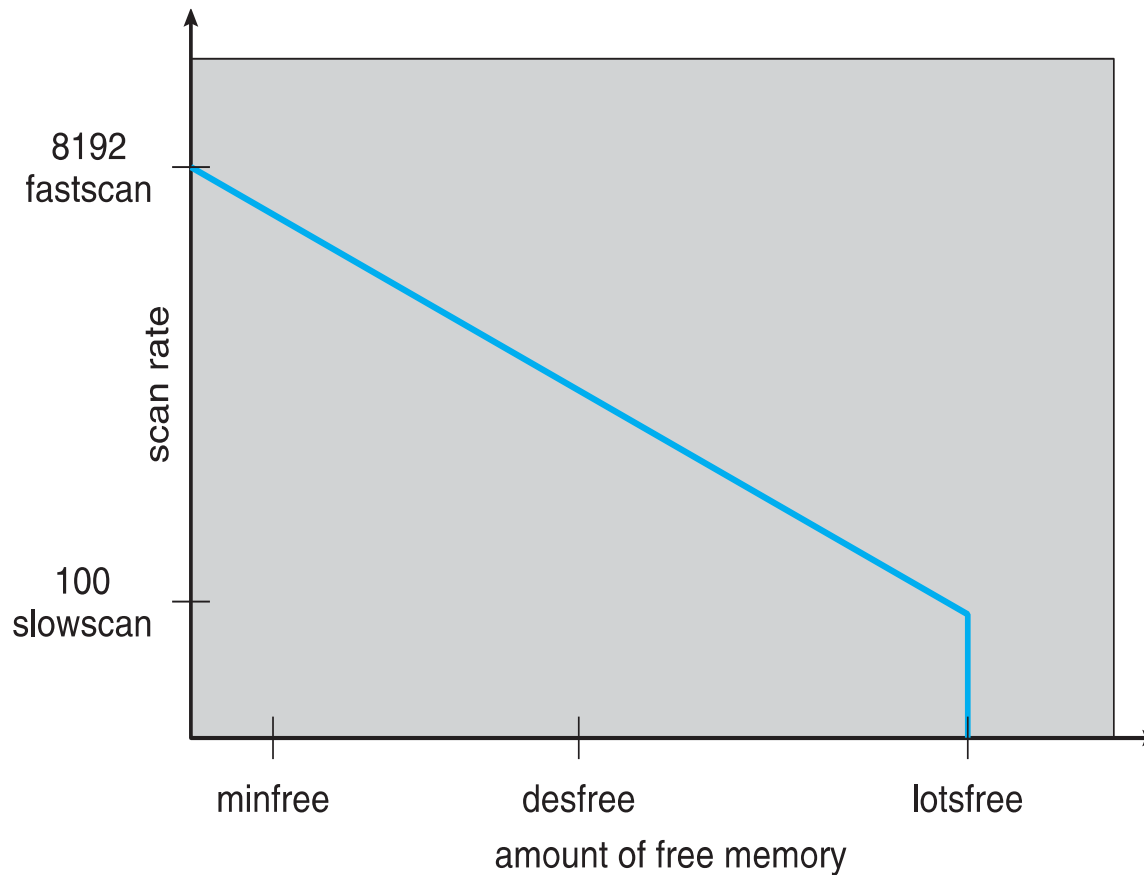
End of Chapter 10

Performance of Demand Paging

- Stages in Demand Paging (worse case)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
   1. Wait in a queue for this device until the read request is serviced
   2. Wait for the device seek and/or latency time
   3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction
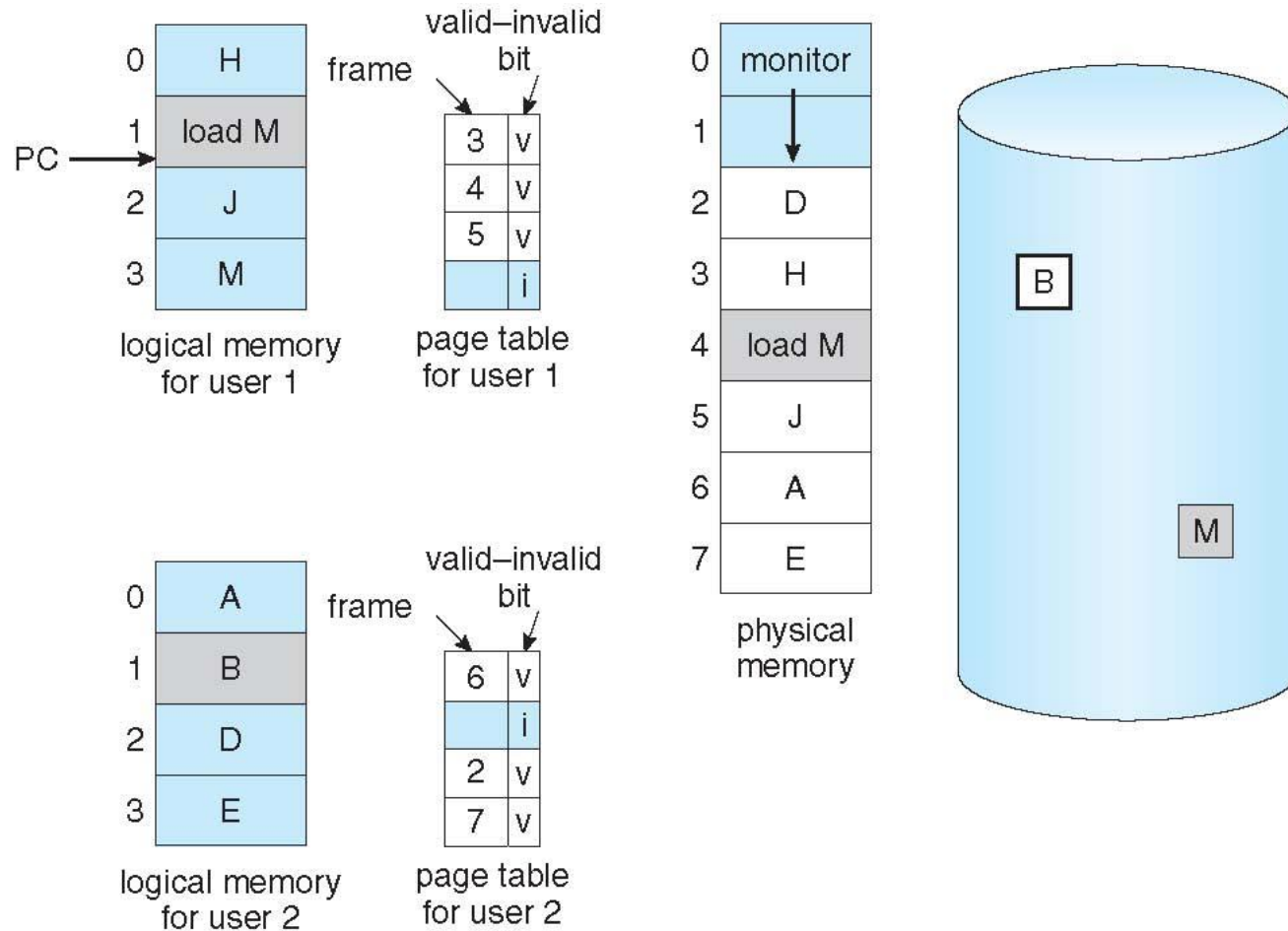
Priority Allocation

- 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

Memory Compression

- **Memory compression** -- rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.

- Consider the following free-frame-list consisting of 6 frames

free-frame list

head → 7 → 2 → 9 → 21 → 27 → 16

modified frame list

head → 15 → 3 → 35 → 26

- Assume that this number of free frames falls below a certain threshold that triggers page replacement. The replacement algorithm (say, an LRU approximation algorithm) selects four frames -- 15, 3, 35, and 26 to place on the free-frame list. It first places these frames on a modified-frame list. Typically, the modified-frame list would next be written to swap space, making the frames available to the free-frame list. An alternative strategy is to compress a number of frames{\mdash}say, three{\mdash}and store their compressed versions n a single page frame.

Memory Compression (Cont.)

- An alternative to paging is **memory compression**.

- Rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.

free-frame list

head ⟶ 2 ⟶ 9 ⟶ 21 ⟶ 27 ⟶ 16 ⟶ 15 ⟶ 3 ⟶ 35

modified frame list

head ⟶ 26

compressed frame list

head ⟶ 7