Operating Systems

Virtual Memory

Giorgio Grisetti
grisetti@diag.uniroma1.it

Department of Computer Control and Management Engineering
Sapienza University of Rome

Facts

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

Facts

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 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:

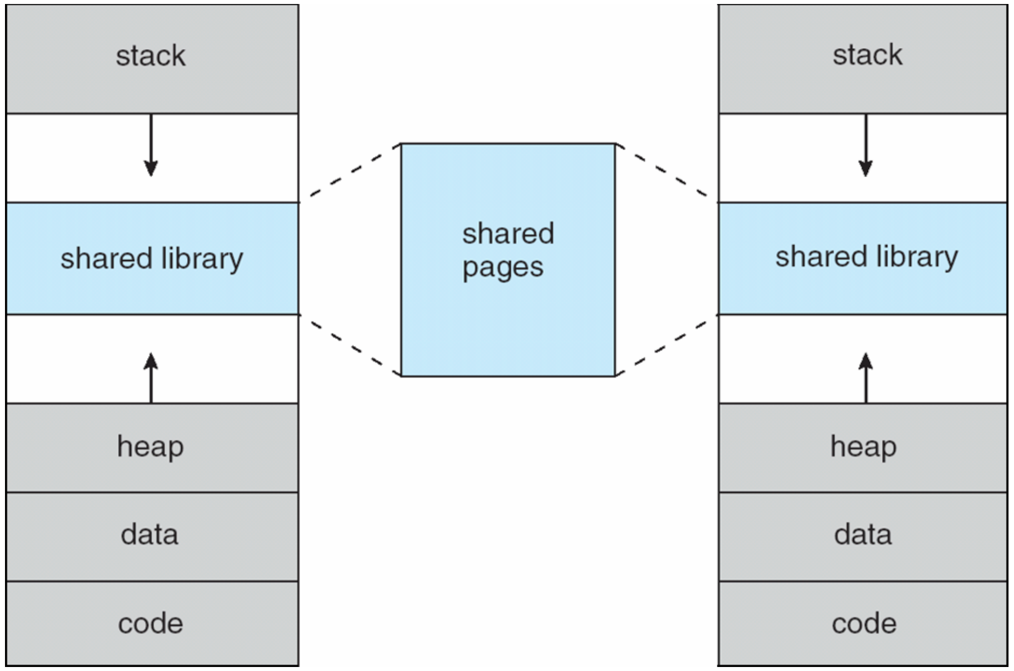
- Demand paging
- Demand segmentation

Virtual 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

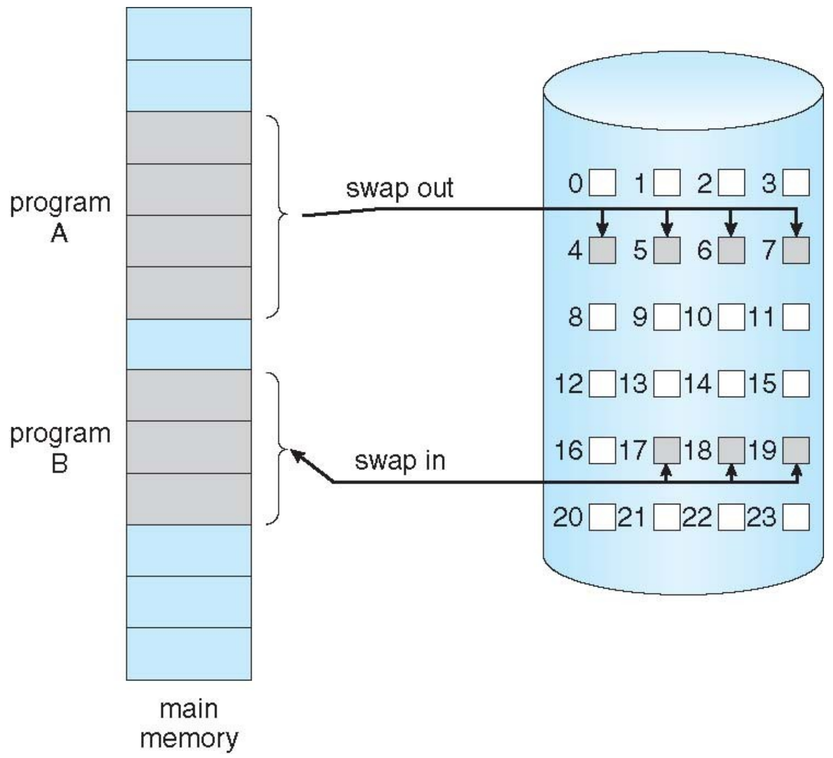


Demand Paging

Istead of bringing the entire process into memory at load time, 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 -> reference to it
  - invalid reference -> abort
  - not-in-memory -> 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

Pure demand paging:extreme case, start process with *no* pages in memory

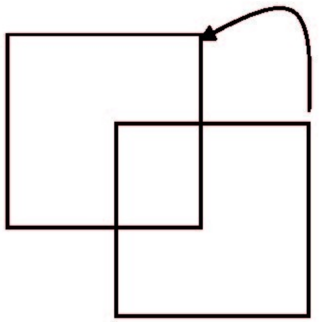
- OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault

Issue: 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**

Critical case: block move

- auto increment/decrement location
- Restart the whole operation?
- What if source and destination overlap?



Hardware support needed for demand paging

- Page table with valid / invalid bit
- Secondary memory (swap device with **swap space**)
- Instruction restart

Demand Paging

Worst case (access to page not in RAM)

- Context switch to OS (Page Fault trap, save state PRIOR instruction execution)
- Check that the page reference was legal and determine the location of the page on the disk
- Issue a read from the disk to a free frame:
  - Wait in a queue for this device until the read request is serviced
  - Wait for the device seek and/or latency time
  - Begin the transfer of the page to a free frame
- While waiting, allocate the CPU to some other user
- Receive an interrupt from the disk I/O subsystem -> Context switch to OS
- Correct the page table and other tables to show page is now in memory
- Switch back to faulting process

Performances:

- Measured with EAT
- Effective Access Time (EAT)

  EAT =

  (1 – p) x memory access

  + p (page fault overhead

  + swap page in)

  - Three major activities
  - Service the interrupt (~1k instructions), goes in overhead
  - Read/Write the page – lots of time
  - Restart the process – (~1k istructions), goes in overhead

- Page Fault Rate $0 \le p \le 1$
  - if p = 0 no page faults
  - if p = 1, every reference is a fault
-

Demand Paging Performance

Effective Access Time (EAT)

EAT =

(1 – *p*) x memory access

+ *p* (page fault overhead

+ swap page in)

- Overhead (~hundreds of instructons):
  - Service the interrupt
  - Restart the process
- Swap Page Out/In: lots of time

p: page fault rate $0 \le p \le 1$

  - if *p* = 0 no page faults
  - if *p* = 1, every reference is a fault

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.

  Slowdown by a factor of 40!!

- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
  - p < .0000025 -> (1/400,000)

Demand Paging Optimizations

Swap space

Disk area without a file system (raw mode)

- I/O is faster than file system I/O even if it resides on the same device
    - NO filesystem overhead
- On startup: copy entire process image to swap space at process load time
- On execution: swap in and out of swap space
- When swapping out read only memory don’t write back the data
- RW pages need to be written back when swapped out

Copy on Write (COW)

- When forking, replicate only the page table, to point to parent frames, but toggle a flag on the pages
- When forking, and set a "trap_on_write" flag on pages to 1
- on write a trap is generated
    - the frame is copied, and the bit is cleared so that further accesses will not trap
    - use reference counters on frames in OS to handle multiple forks

Free frames

- The system keeps a list of free frames (similar to a SLAB), to quickly get a free page when needed
- For security: free pages are zeroed (otherwise a new process might read the data of a dead process)

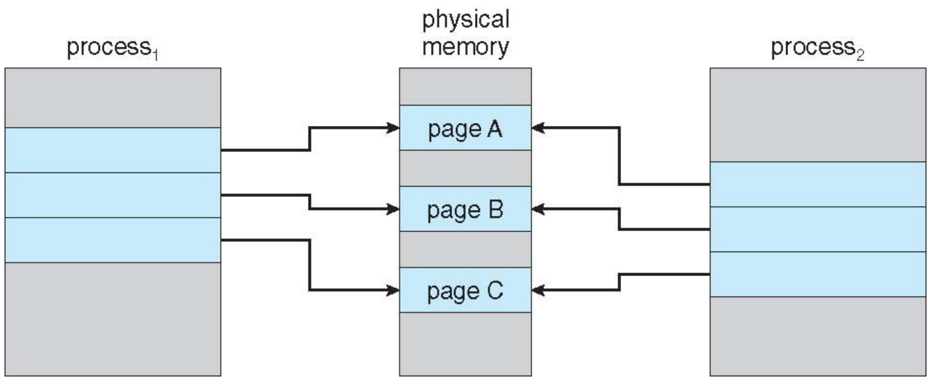
COW Example

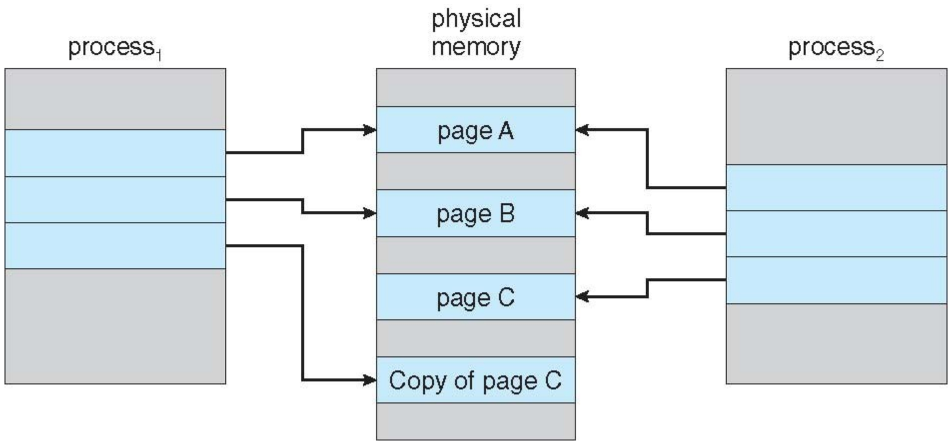
Before

- process 1 forks, and generates process 2
- page table is just copied, and trap_on_write bit is set
- frames are not copied

After

- when process 1 writes on page c, a trap is generated
- the frame "C" is copied, and a the value in the page table of process 1 is updated





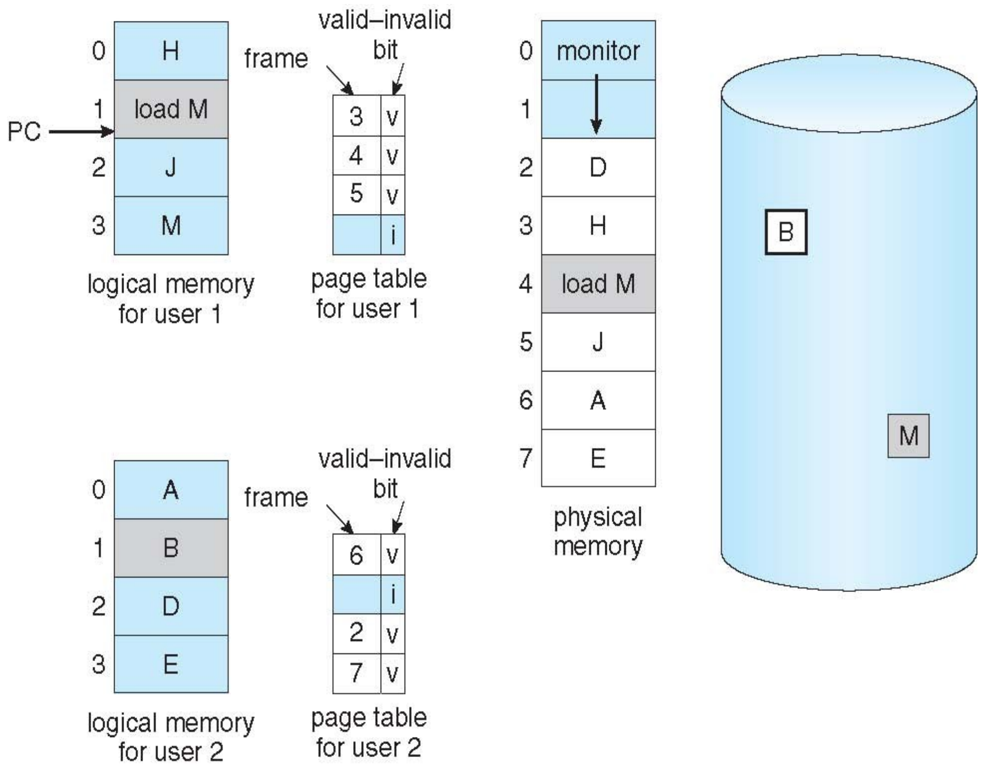
Page Replacement

Choose a used frame in memory to be swapped out (victim).

- Used when no free frame is available
- Optimality: choose the page that will be accessed latemost
- requires knowledge about the future

Pages that have not been altered in RAM do not need to be written back

- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers
- They can be dismissed at lower cost



Basic Page Replacement

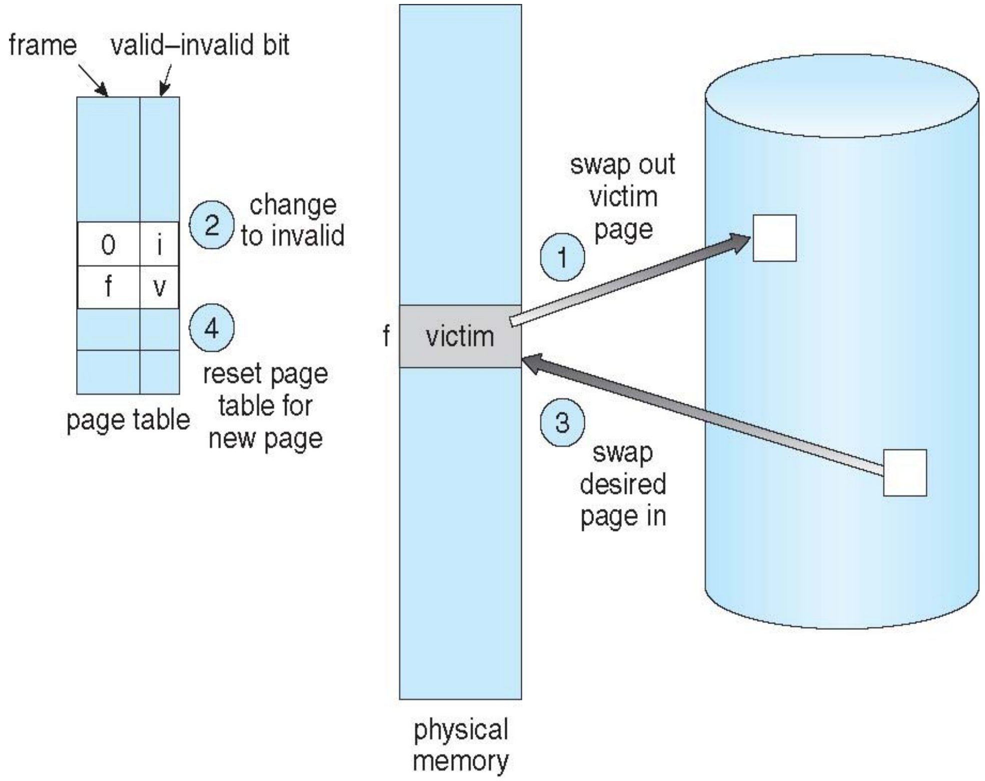
- Find the location of the desired page on disk

- 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

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

- Continue the process by restarting the instruction that caused the trap

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



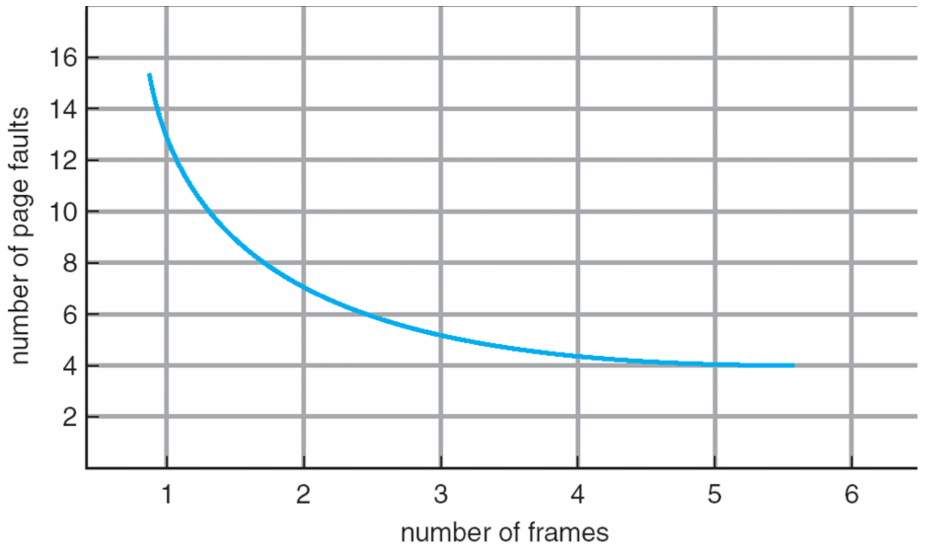
Page Replacement Algorithms

Goal: minimizing the page fault rate

- Evaluation:
  - through simulation, using an array (string) encoding the access pattern
  - s[i]=x, means that at time i, the system uses page x
  - Example of reference string:

<7,0,1,2,0,3,0,4,2,3,0,3,0, 3,2,1,2,0,1,7,0,1>

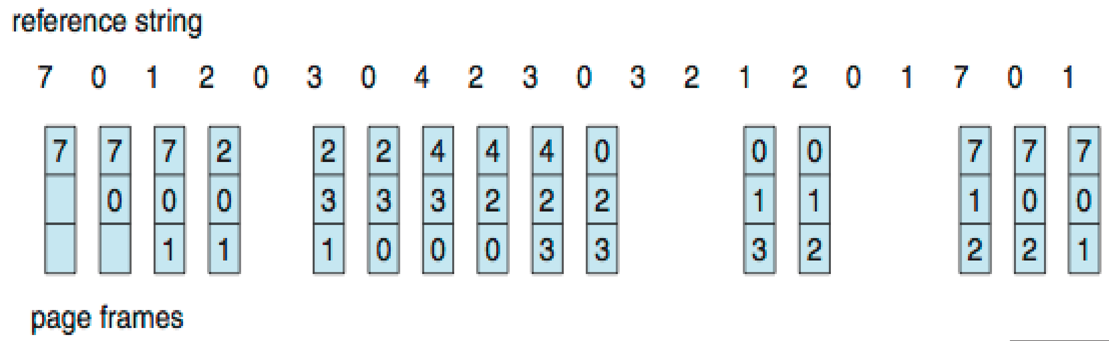
The more the frames, the less the page faults



PR: FIFO algorithm

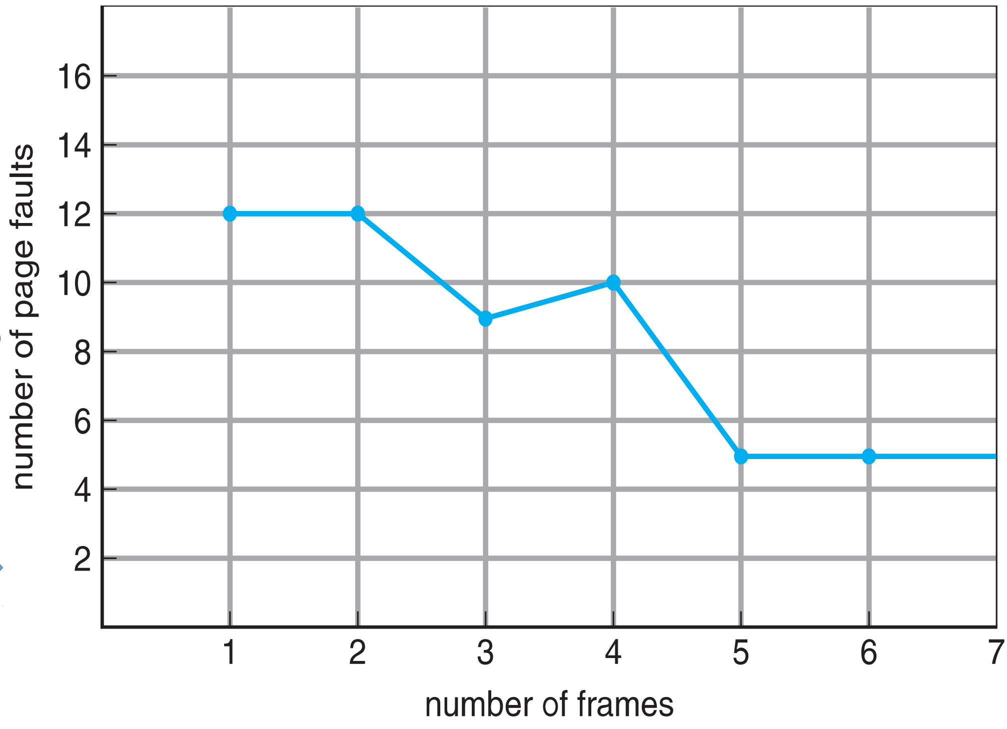
Idea: Choose as victim the page that was swapped in last

- Example (3 frames)



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

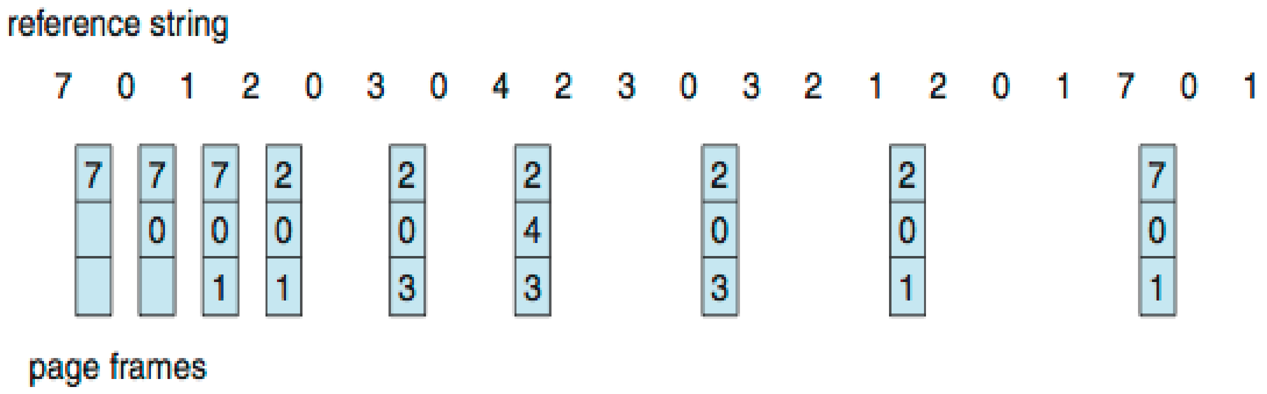
- Adding more frames can cause more page faults!
  - **Belady's Anomaly**



Optimal Page Replacement

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

- can't be done in practice
- provides an upper bound
  - all algorithms will be worse than optimal



Optimal value for this configuration is 9

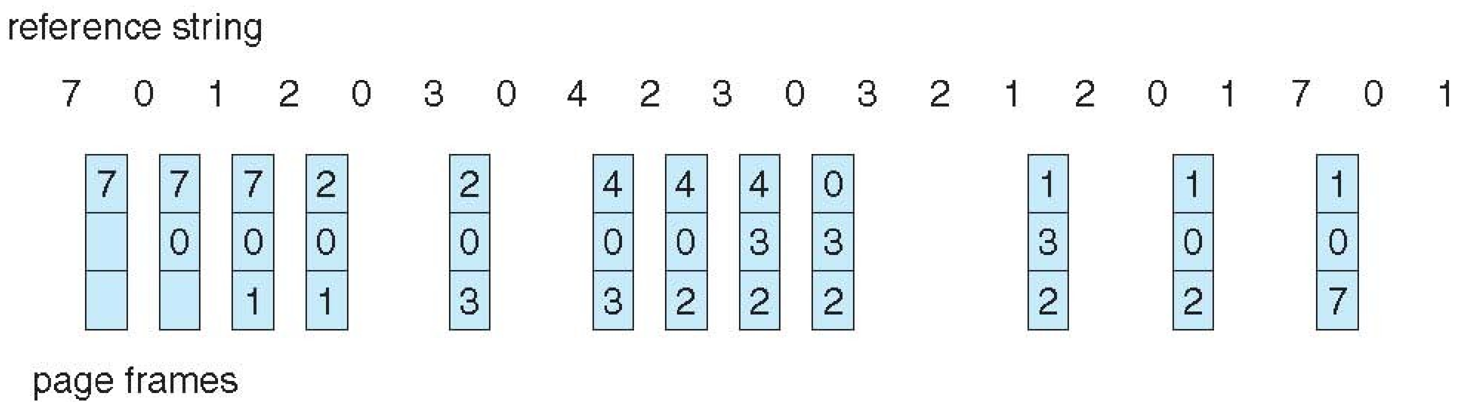
LRU page replacement

Idea: approximate optimal by predicting which page will be used last.

Use prior knowledge to get the prediction: history repeats

Evicted page: the page that has not been used since longer

LRU not subject to Belady anomaly



12 faults

LRU Implementations

- Counter:
  - each has a counter, when accessed copy the clock in the counter
  - on evicition: scan the page table
- List:
  - keep a list of pages. Each time a page is accessed, move it s entry on top of the list.
  - expensive

Shortcomings: LRU requires special hardware, but it is still slow.

Full implementations not used.

Approximated implementations are.

LRU Approximations

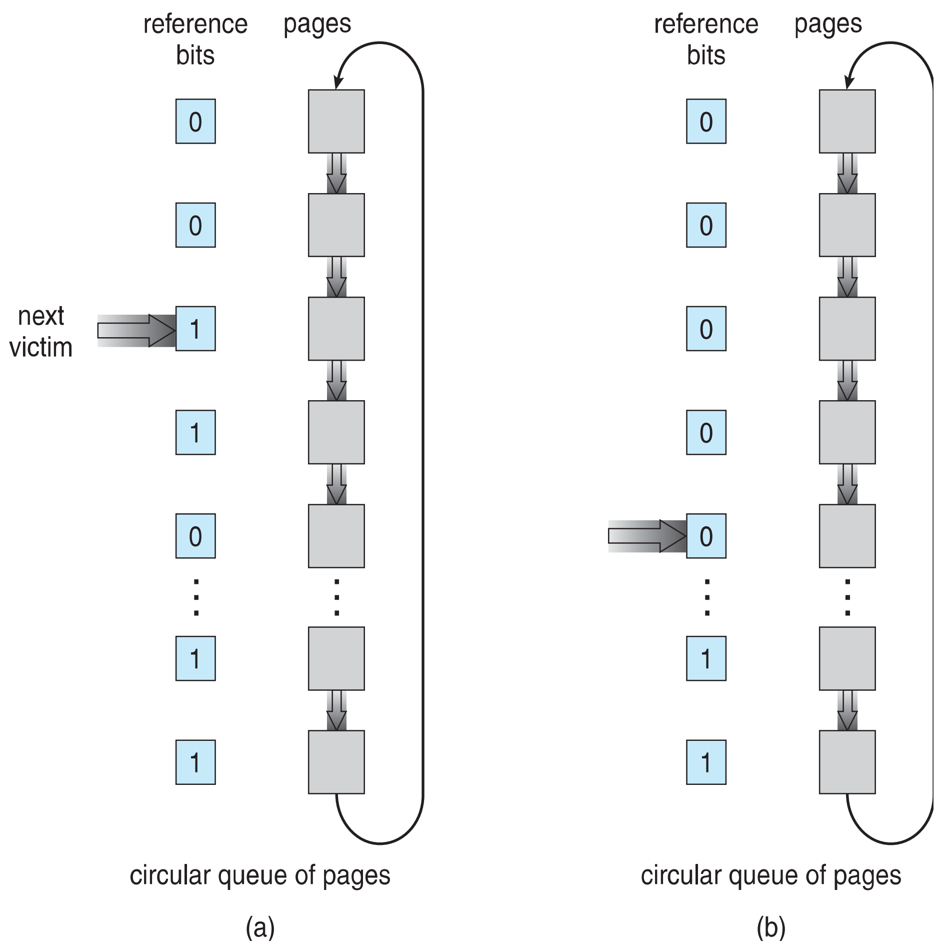
Need reference bit in page table (HW support)

- 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

Second-chance algorithm

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



LRU approximations

Hardware Support:reference bit **and modify bit** in page table (HW support)

- When accessing a page set modify bit to 1

Enhanced Second-Chance Algorithm

Clock replacement

- rank pages based on acces and modify bit
  - 0,0: best candidate (no write)
  - 0,1: write, but used long ago
  - 1,0: used recently, but no write
  - 1,1: worst case

Thrashing

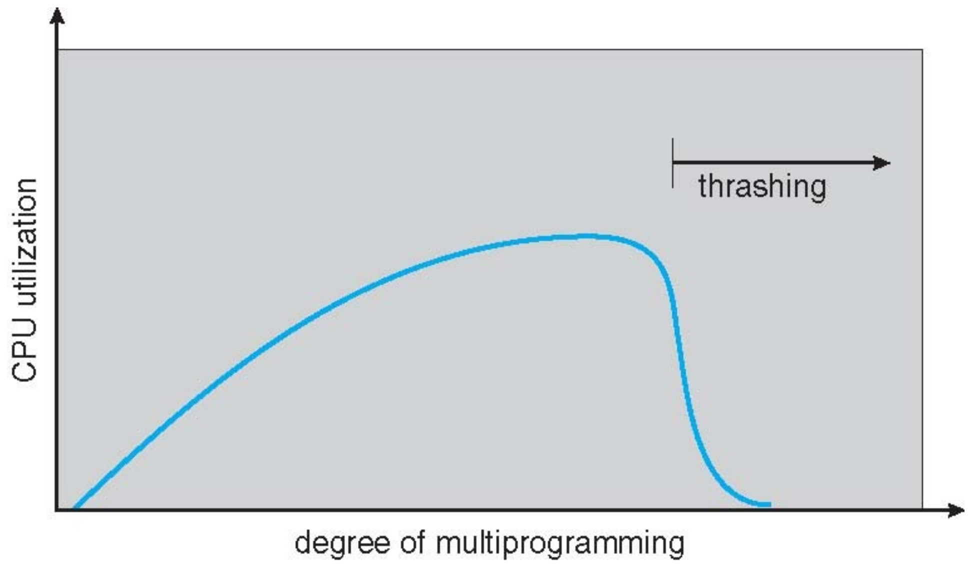
Thrashing: a process is busy swapping pages in and out

Happens when 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

Consequences:

- Low CPU utilization
- Operating system thinking that it needs to increase the degree of multiprogramming
- Another process added to the system



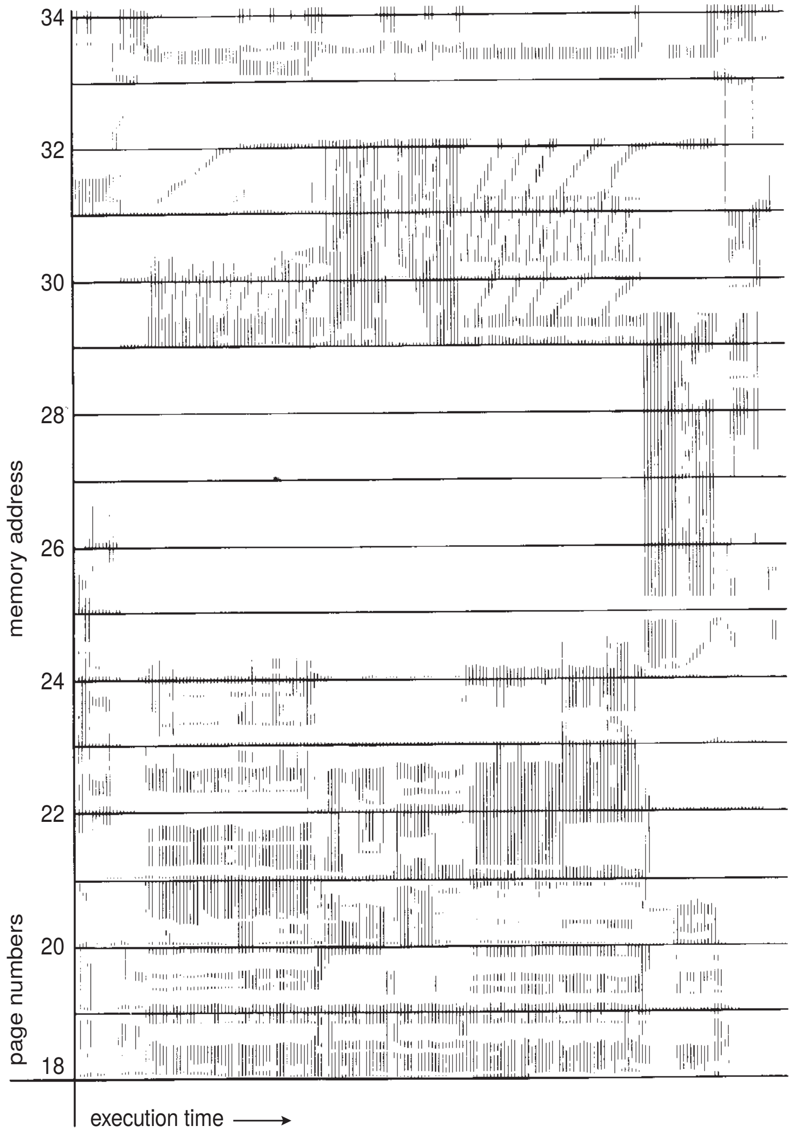
Memory Access Pattern

On the right we see the "pattern" of pages accessed as time evolves

Nearby columns show similar "black stripes": the regions of memory accessed as the system evolves changes smoothly

Localiity principle:

- If I have accessed something short ago, it is very likely I will peek on it again in the near future



Working Set Model

Used to model access patterns and locality

- wss(p,t): function of two parameters:
  - t: the "epoch" (time interval) under analysis
  - p: the process id
  - wss(p,t): set of pages accessed in the epoch t

Depends on the "duration" of an epoch

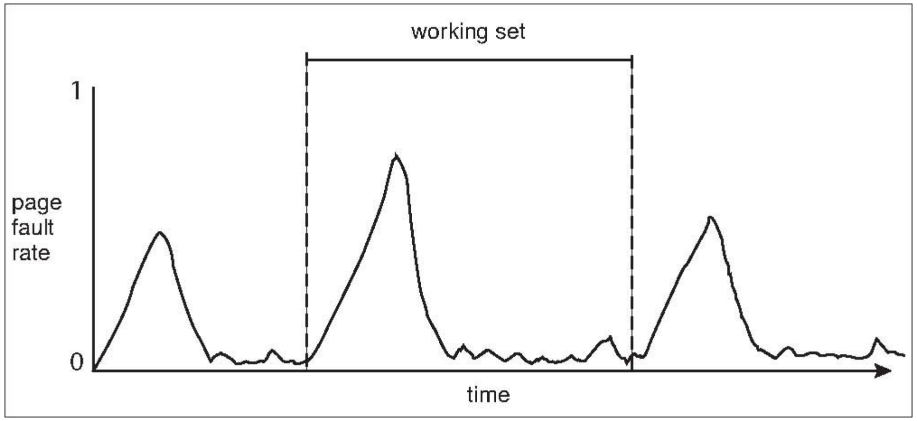
  - if duration too small, not representative for the locality
  - if duration too large, captures several localities

Number of frames required at time t

$D(t)=\Sigma_p wss(p,t)$

Trashing when too little frames available and too many required

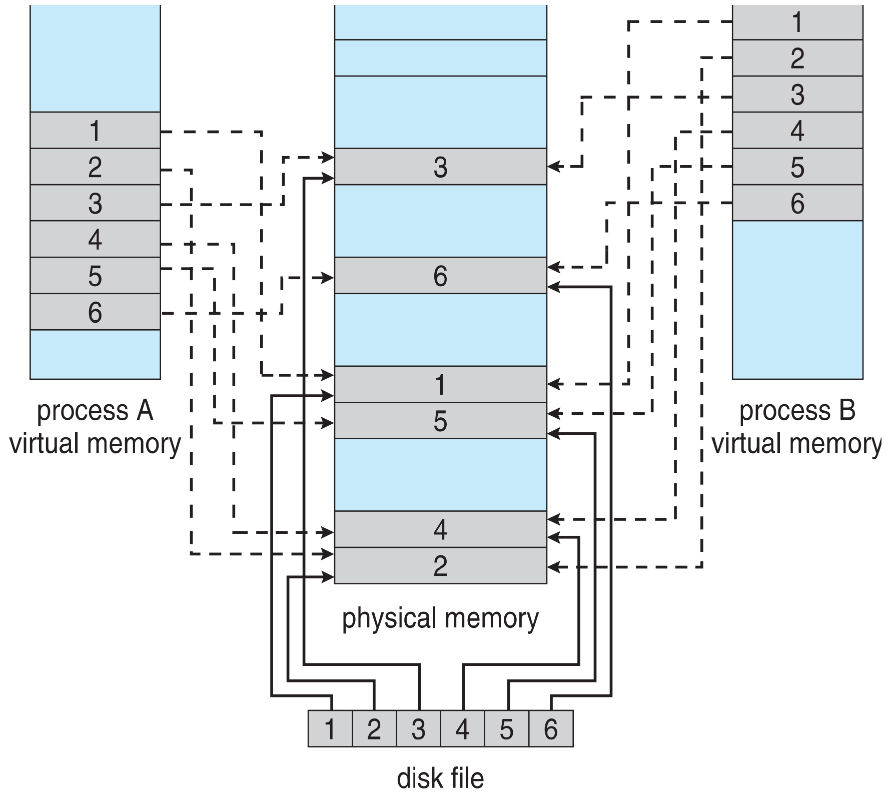
$D(t)>max\_frames$



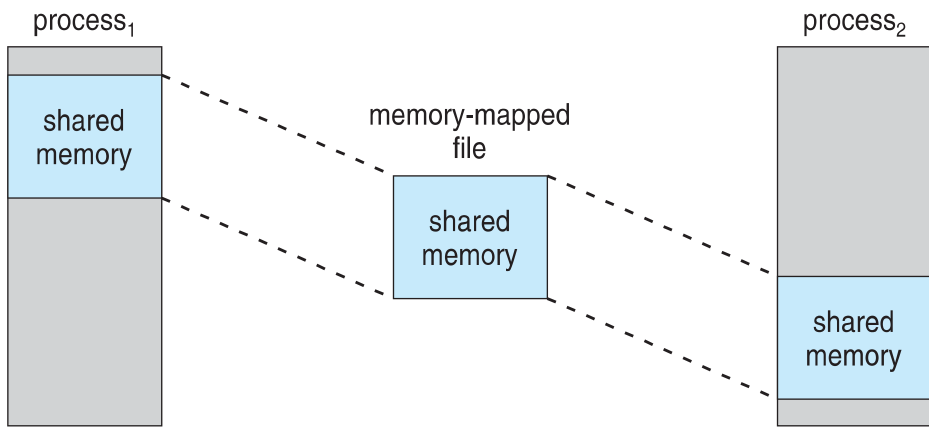
fault rate and working set are correlated

Constructs relying on VM

Memory mapped file (mmap)



Shared Memory


process$_1$
shared memory
memory-mapped file
shared memory
process$_2$
shared memory