Chapter 8: Virtual Memory

virtual memory system

demand paging, page-replacement, and allocation of page frames

working-set model

shared memory and memory-mapped files

Contents

8.1 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 r un 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 us er program runs faster

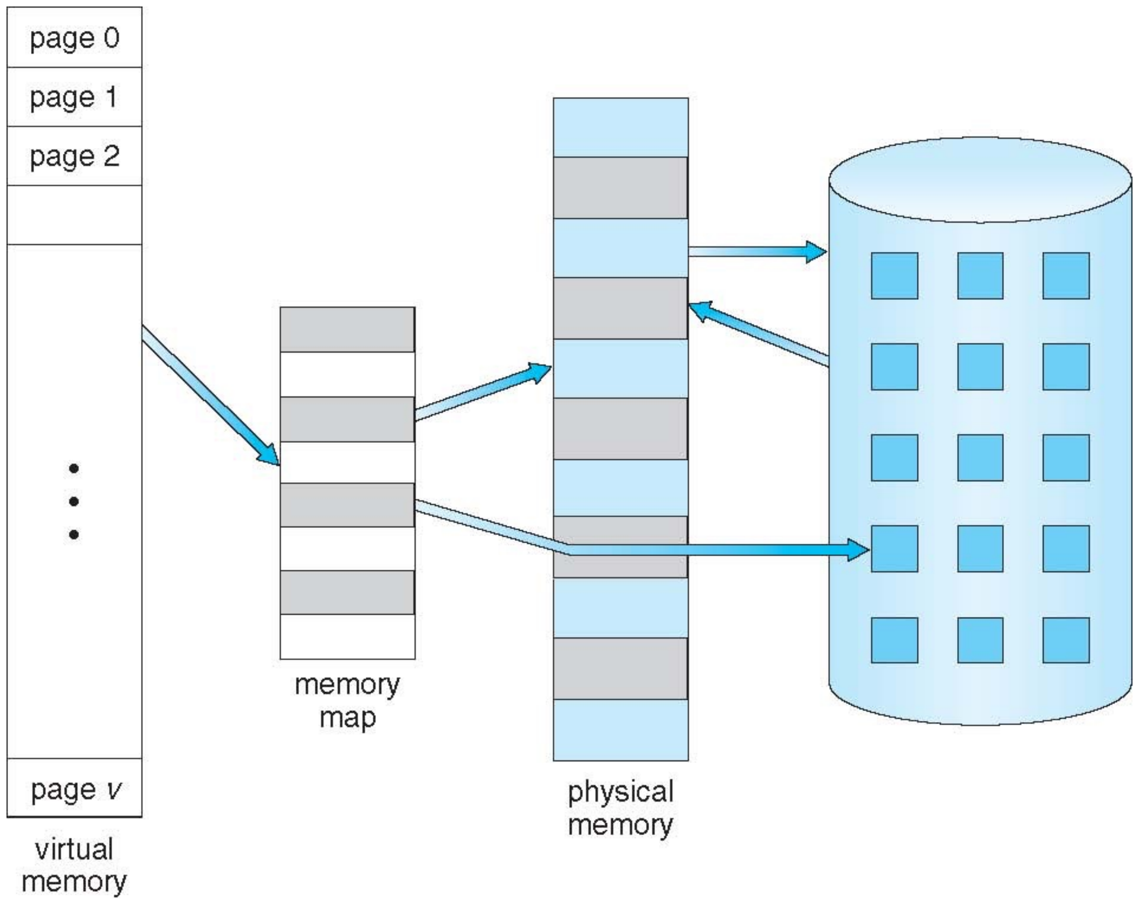
Background (Cont.)

- 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

Background (Cont.)

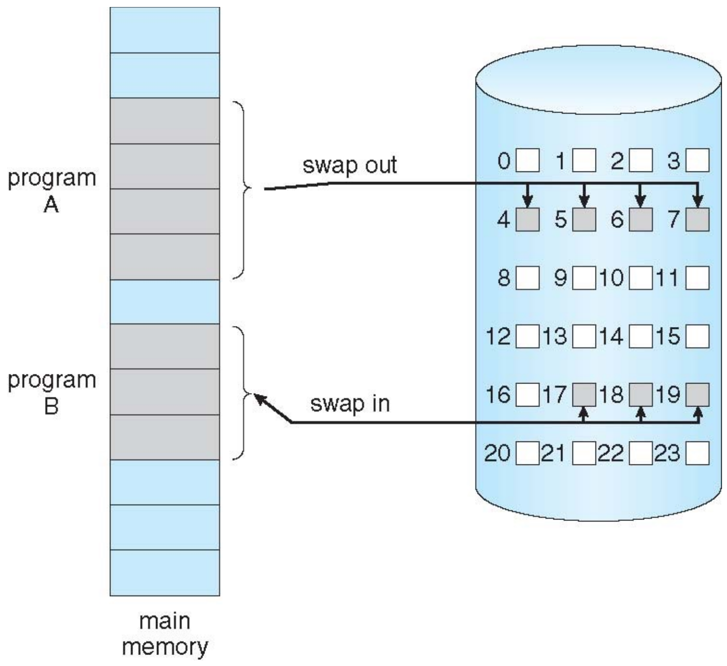
- Virtual address space – logical view of how process is stored i n memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory



8.2 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 ⇒ 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

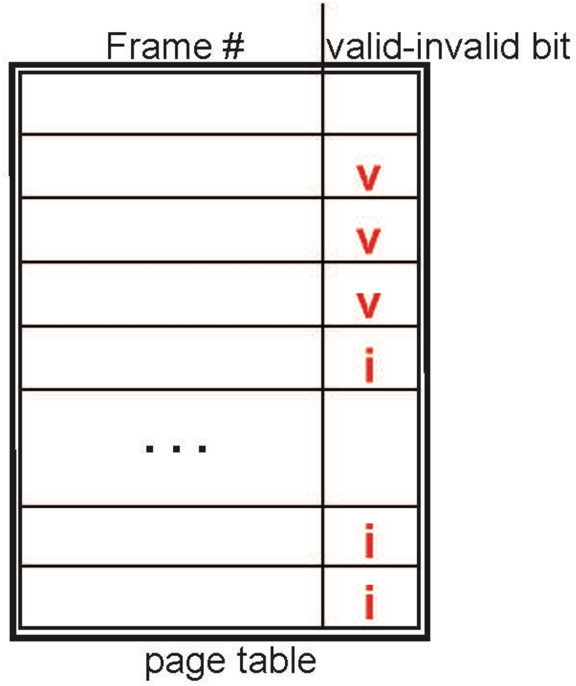


Basic Concepts

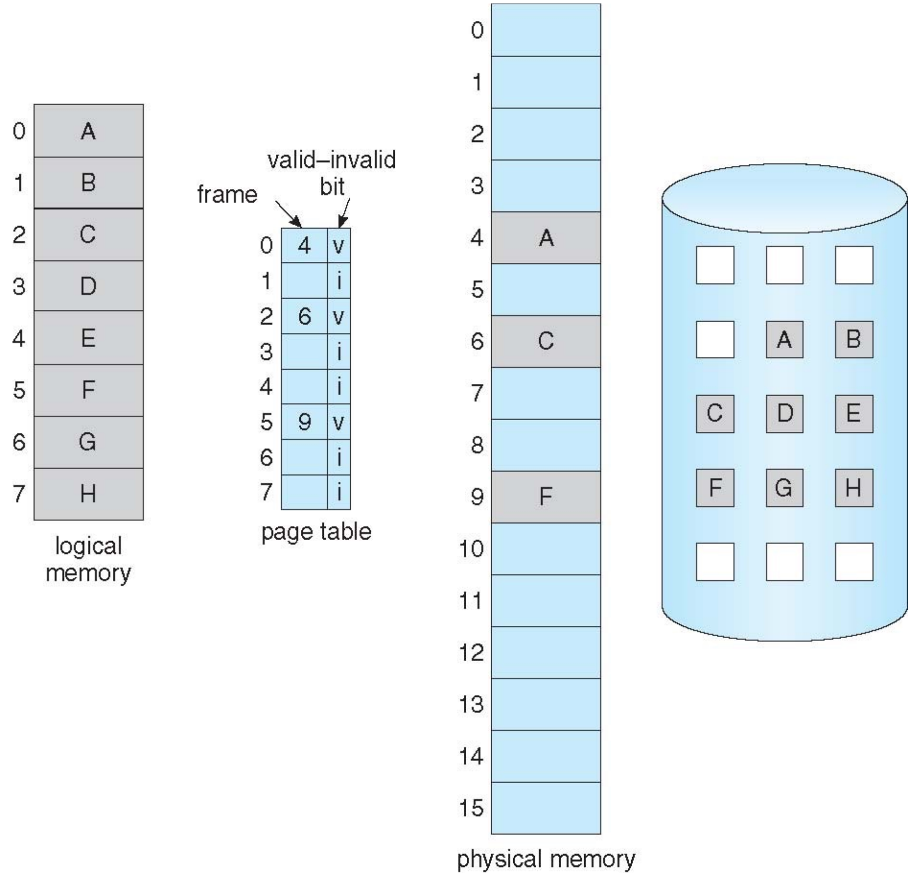
- Pager brings in only those pages into memory; how to deter mine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already memory resident
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the 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 ⇒ in-memory – memory resident, i ⇒ not-in-memory)
- Initially valid–invalid bit is set to i on all entries
- During MMU address translation, if valid–invalid bit in page table entry is i ⇒ page fault

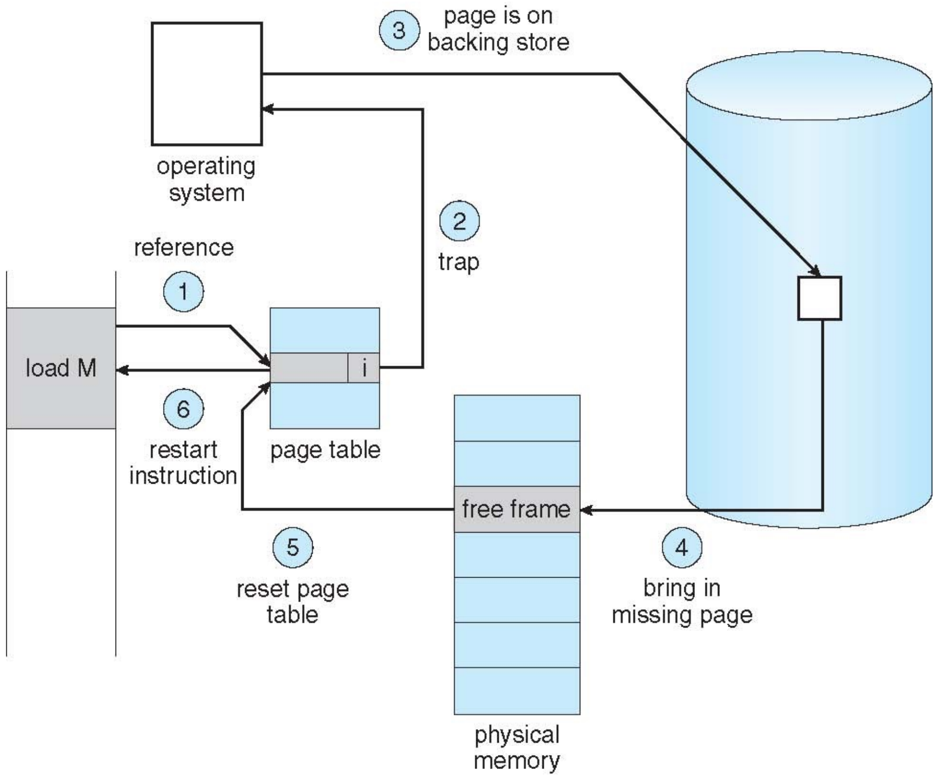


Page Table When Some Pages Are Not in Main Memory



Page Fault

1. If there is a reference to a page
2. First reference to that page will trap to operating system: page fault
3. Locate the page in disk
4. Find free frame and Swap pag e into frame via scheduled disk operation
5. Reset tables to indicate page n ow in memory; set validation bi t = v
6. Restart the instruction that caus ed the page fault

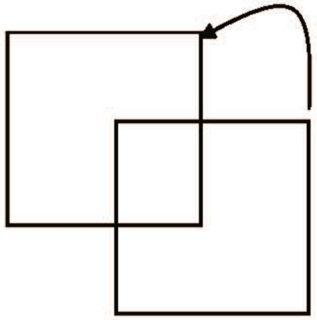


Aspects of Demand Paging

- 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 location s
  - block move



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

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

Performance of Demand Paging (Cont.)

- Three major activities
  1. Service the interrupt – careful coding means just several hundred i nstructions needed
  2. Read the page – lots of time
  3. Restart the process – again just a small amount of time
- Page Fault Rate $0 \le p \le 1$
  - if p = 0 no page faults
  - if p = 1, every reference is a fault
- Effective Access Time (EAT)

$$effective\ access\ time = (1-p) \times ma + p \times page\ fault\ time$$

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 !!

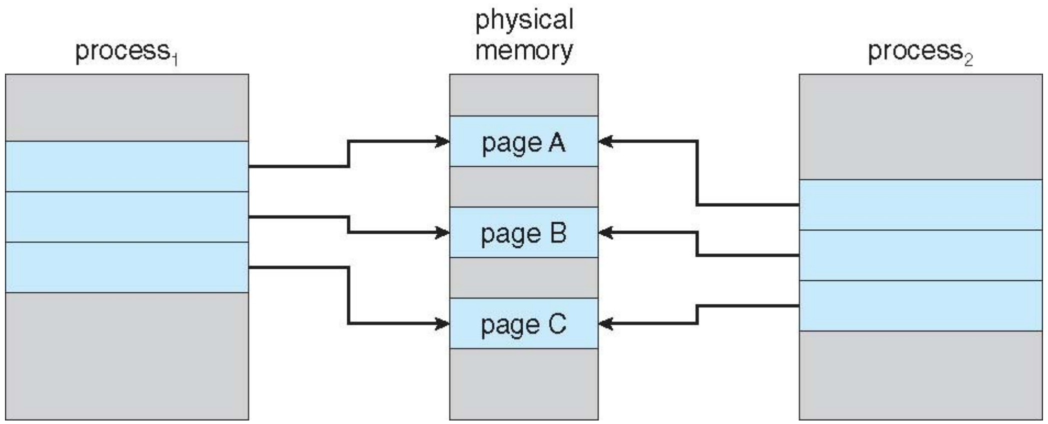
Demand Paging Optimizations

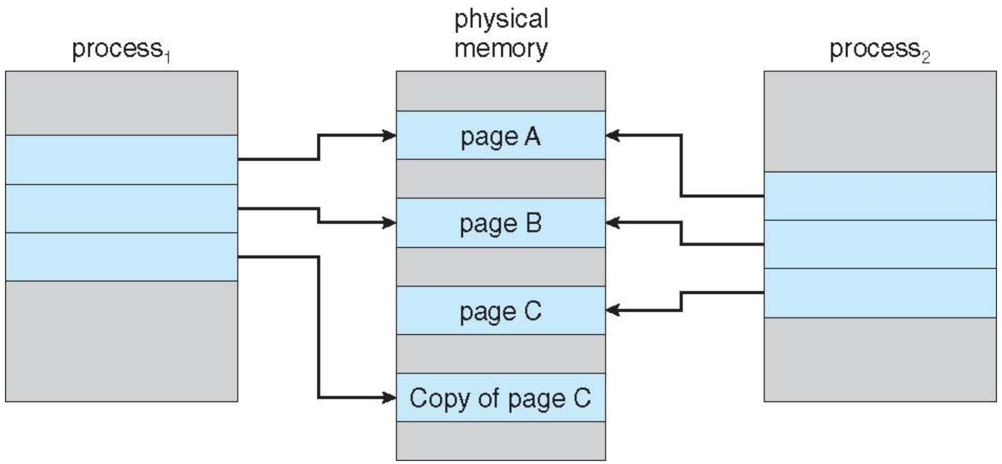
- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program 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)

8.3 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 usin g copy-on-write address space of parent
  - Designed to have child call exec()
  - Very efficient

Before and After Process Modifies Page C





8.4 Page Replacement

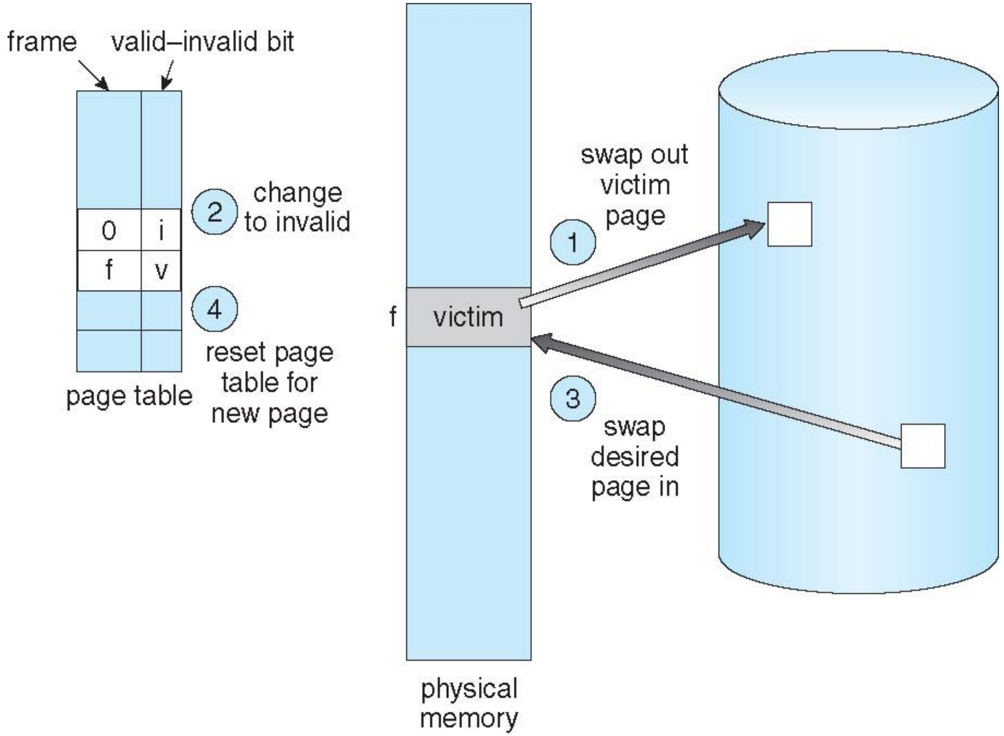
- What Happens if There is no Free Frame?
  - Used up by process pages
  - Also in demand from the kernel, I/O buffers, etc
- Page replacement – find some page in memory, but not reall y in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum num ber of page faults
- Same page may be brought into memory several times

Page Replacement (Cont'd)

- 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

- 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

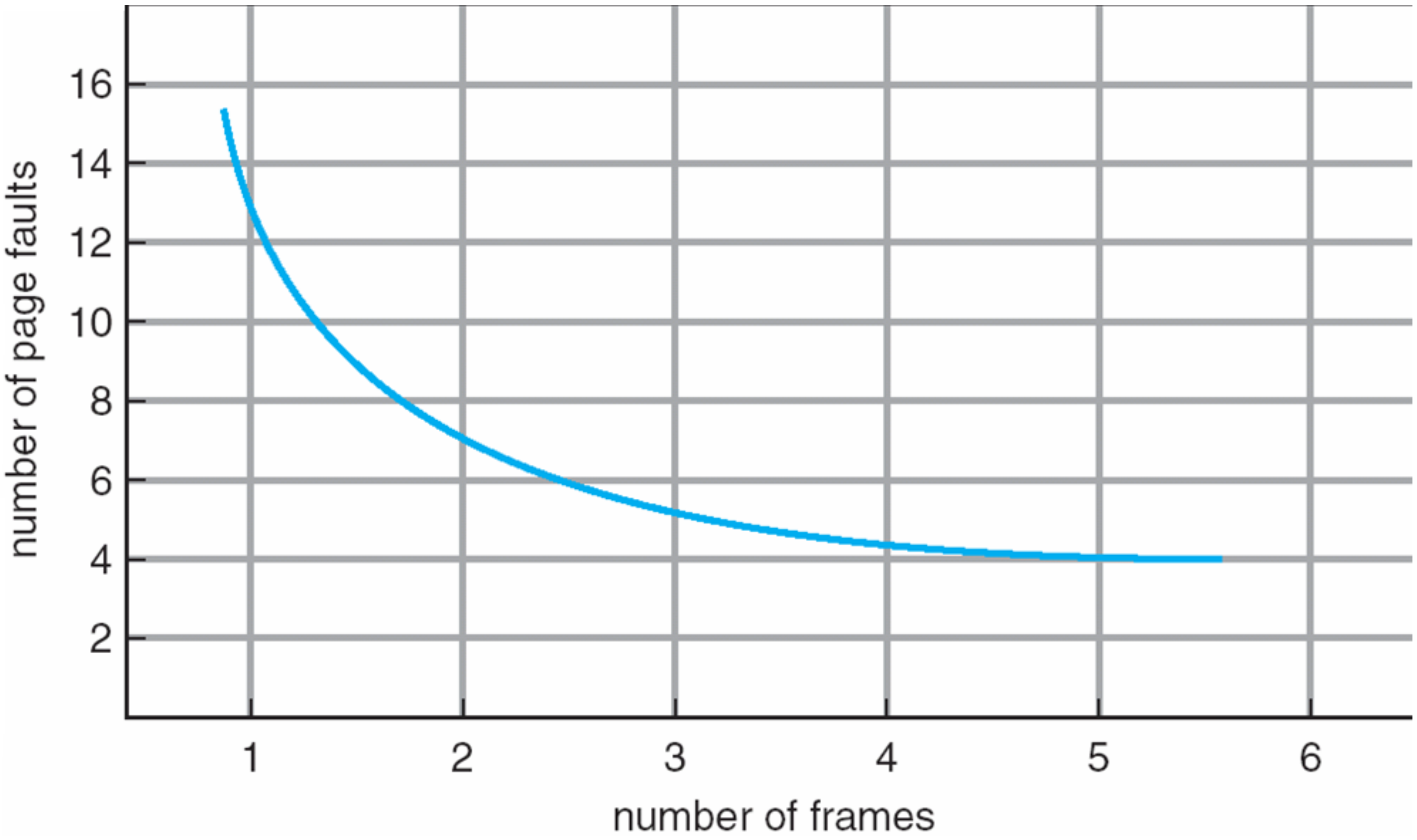


Page and Frame Replacement Algorithms

- Frame-allocation algorithm determines
  - How many frames to give each process
- 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 f aults 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 numb ers is

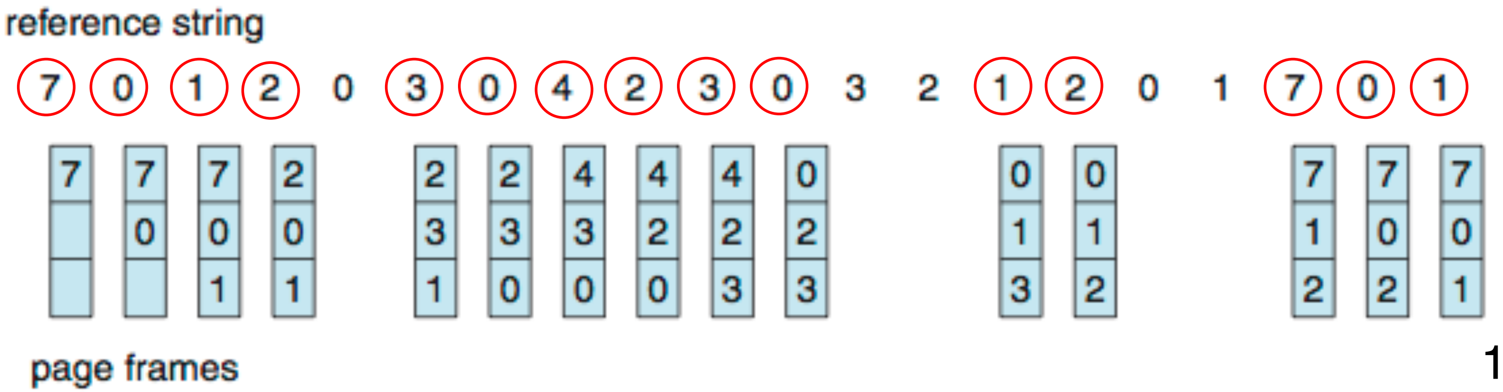
  7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

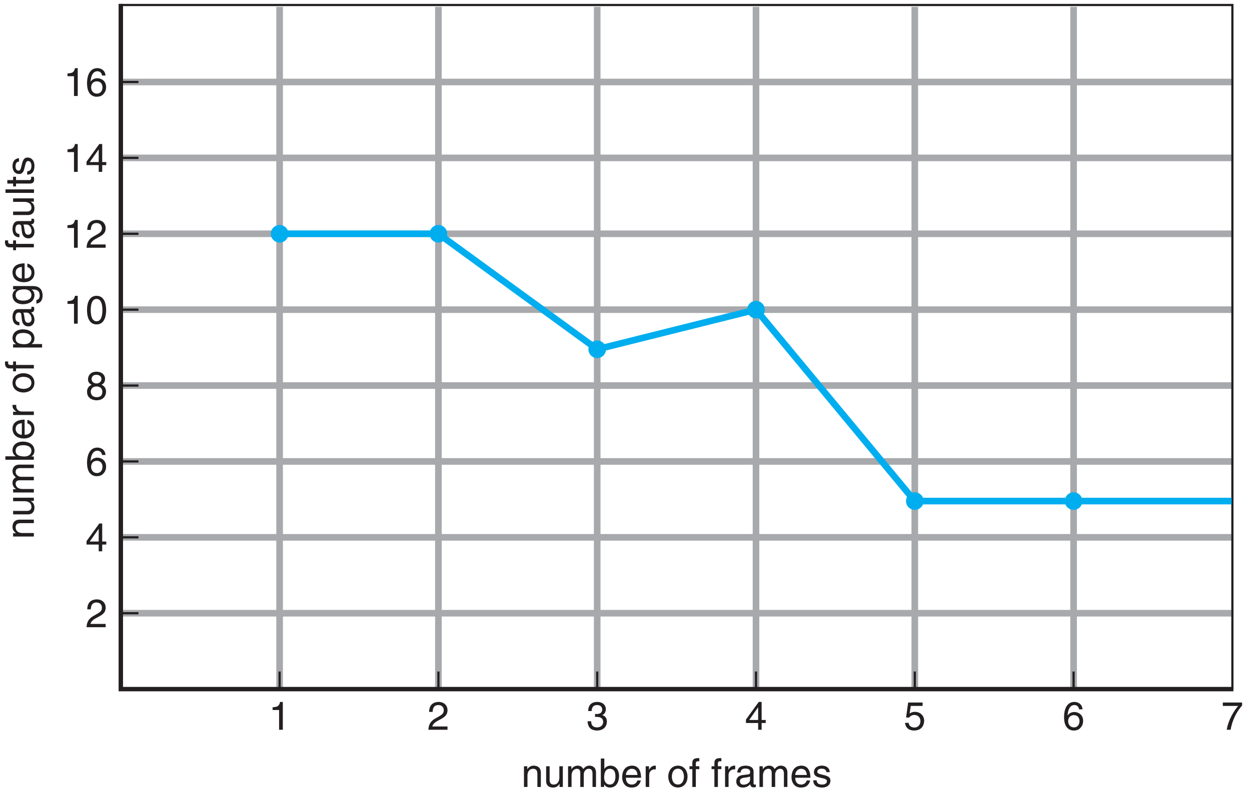
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

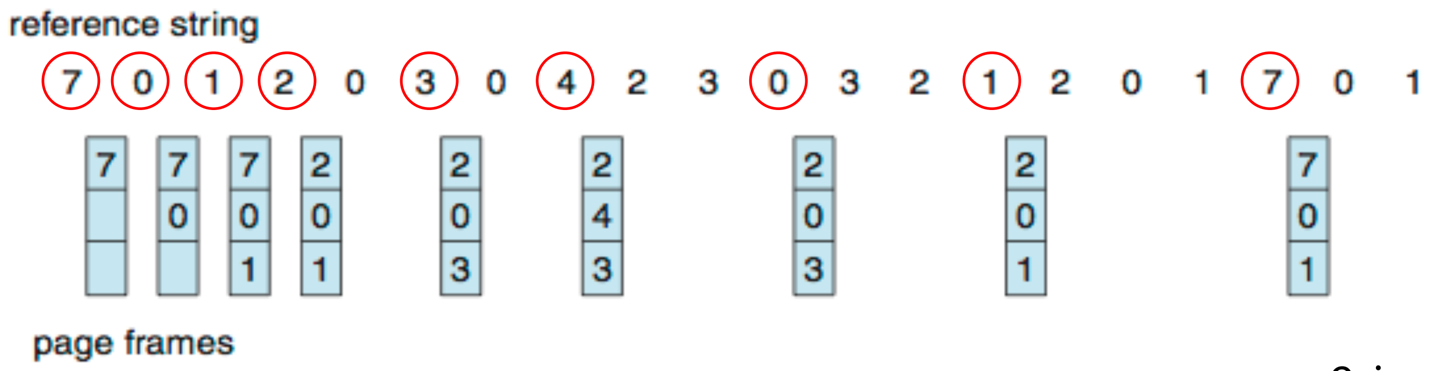
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults! ➔ Belady's Anomaly
- How to track ages of pages?
  - Just use a FIFO queue

FIFO Illustrating Belady's Anomaly



Optimal Algorithm

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

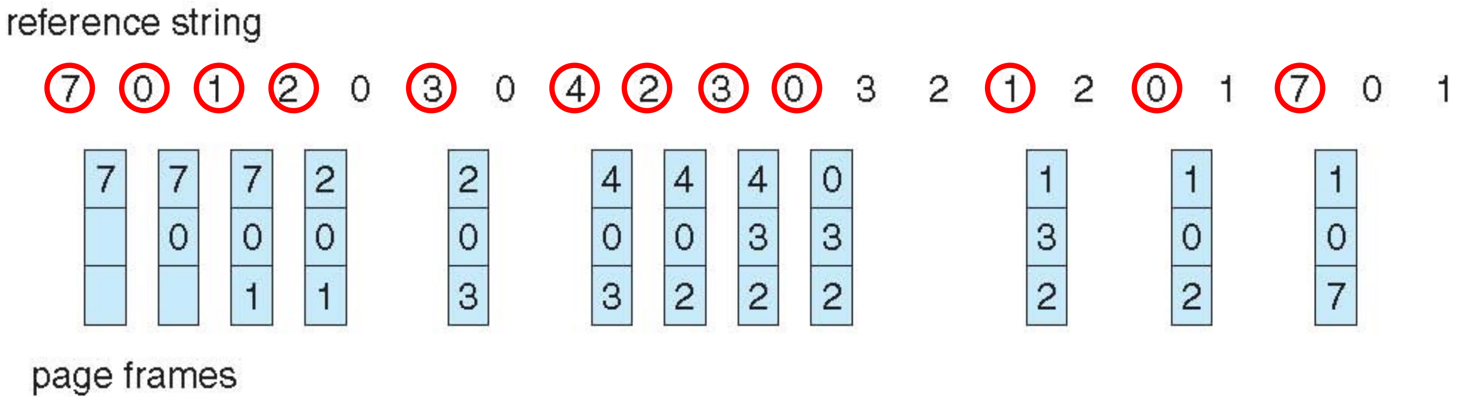


9 is optimal for the example

- How do you know this?
  - Can’t read the future
- Used for measuring how well your algorithm performs

Least Recently Used (LRU) Algorithm

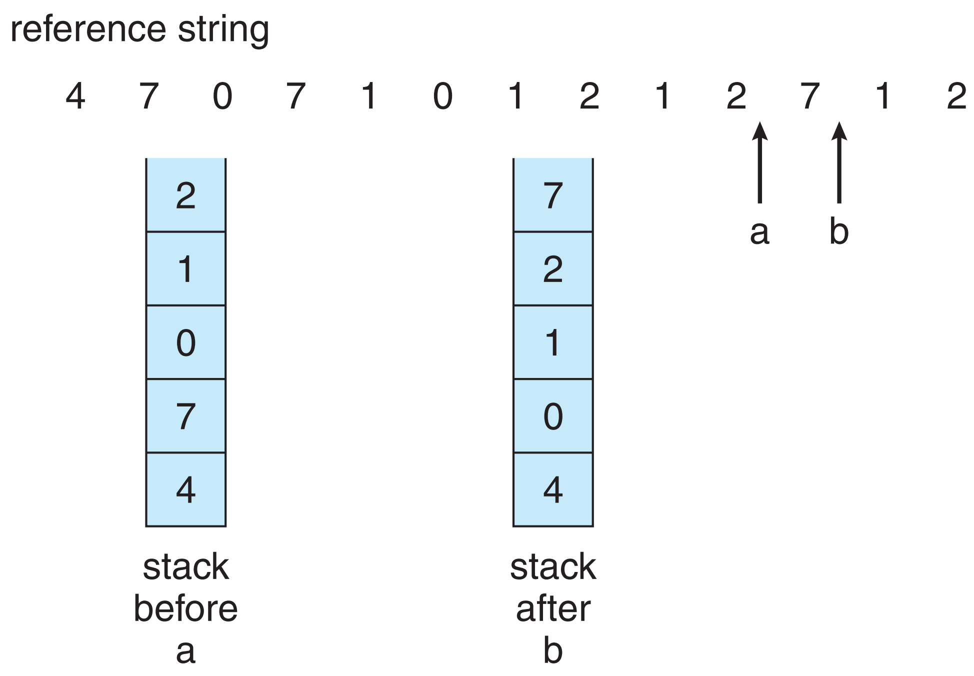
- Use past knowledge rather than future; replace page that has not been used i n the most amount of time
- Associate time of last use with each page



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

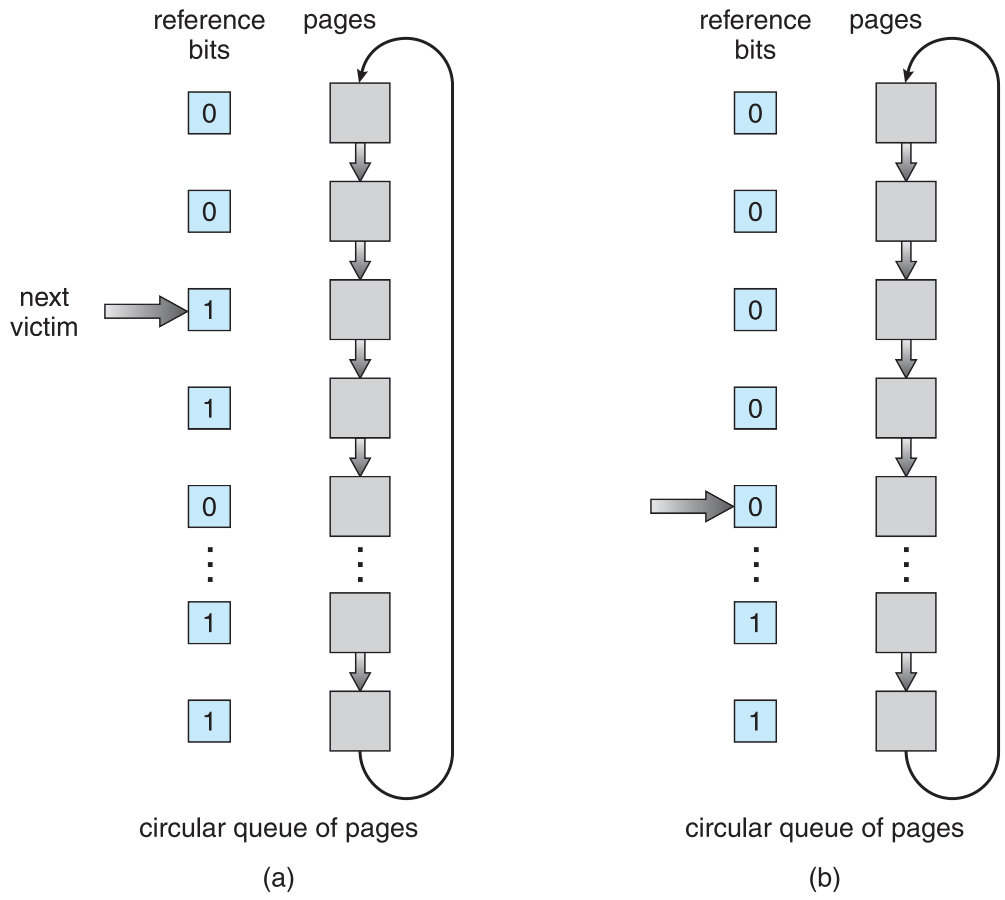
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of 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 and OPT are cases of stack algorithms that don't have Belady's Anomaly



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
- 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 (clock) Page-Replacement Algorithm



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if availab le) in concert
- Take ordered pair (reference, modify)
    1. (0, 0) neither recently used not modified – best page to replace
    2. (0, 1) not recently used but modified – not quite as good, must write ou t before replacement
    3. (1, 0) recently used but clean – probably will be used again soon
    4. (1, 1) recently used and modified – probably will be used again soon an d 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
  - Not common
- Least Frequently Used (LFU) Algorithm: replaces page with s mallest count
- Most Frequently Used (MFU) Algorithm: based on the argume nt that the page with the smallest count was probably just br ought in and has yet to be used

Page-Buffering Algorithms

- Keep a pool of free frames
  - 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 dis k
  - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All of these algorithms have OS guessing about future page a ccess
- 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; bypasses buffering, lockin g, etc
  - Raw disk mode

8.5 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

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 62 \approx 4$$

$$a_2 = \frac{127}{137} \times 62 \approx 57$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather t han size

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

Global vs. Local Allocation

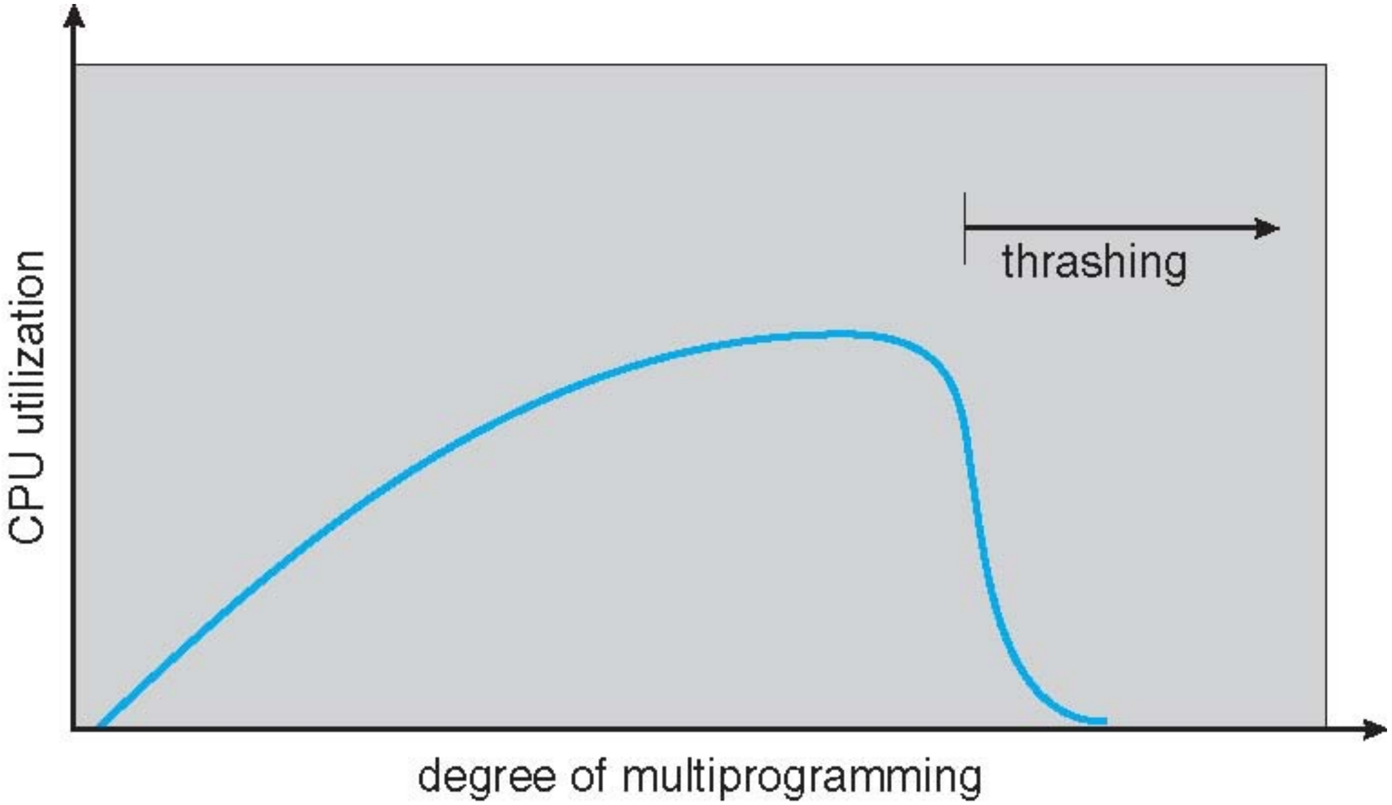
- Global replacement – process selects a replacement frame fro m the set of all frames; one process can take a frame from an other
  - But then process execution time can vary greatly
  - But greater throughput so more common

- Local replacement – each process selects from only its own s et of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

8.6 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 multipro gramming
  - Another process added to the system
- Thrashing ≡ a process is busy swapping pages in and out

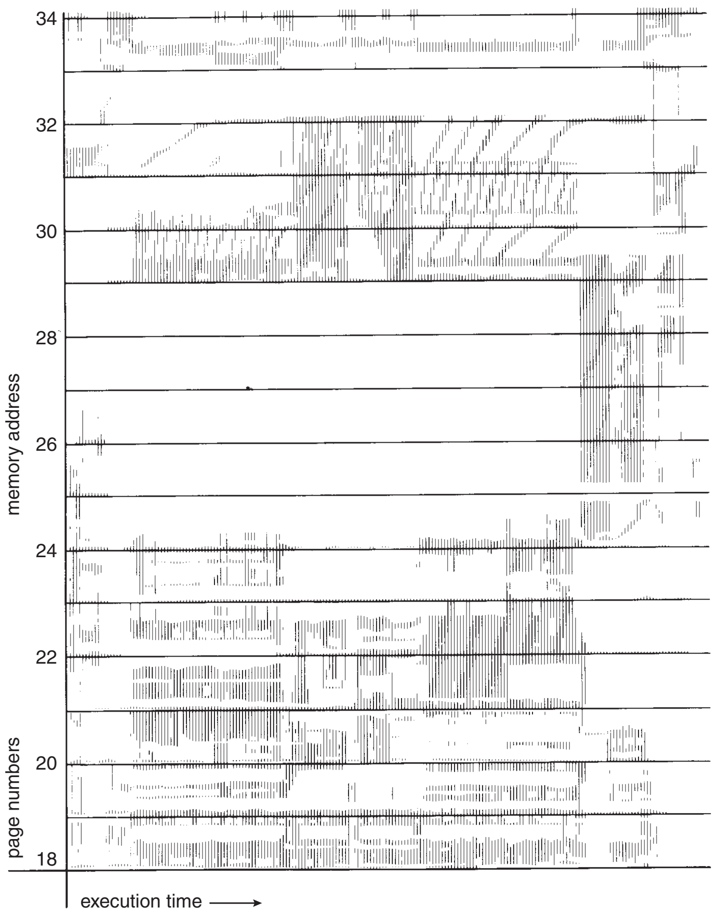
Thrashing (Cont.)



Demand Paging and Thrashing

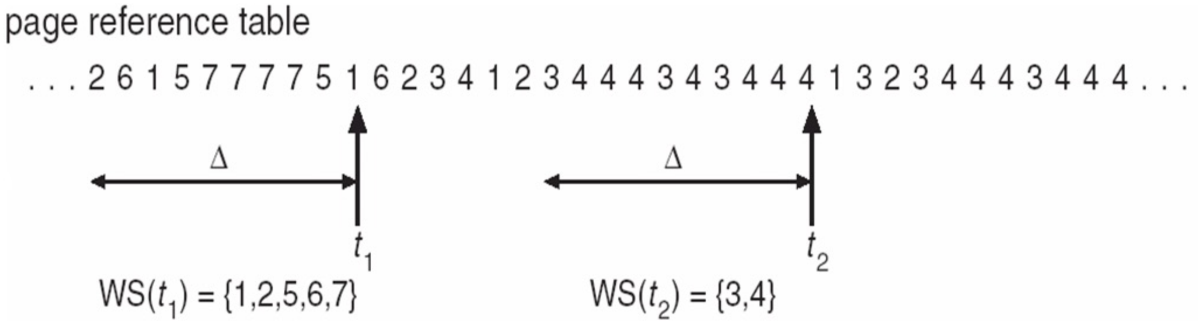
- Why does demand paging w ork?
  Locality model
  - Process migrates from one loc ality to another
  - Localities may overlap

- Why does thrashing occur?
  Σ size of locality > total me mory size



Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSi (working set of Process Pi) =
  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
- $D = \Sigma$ WSi $\equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

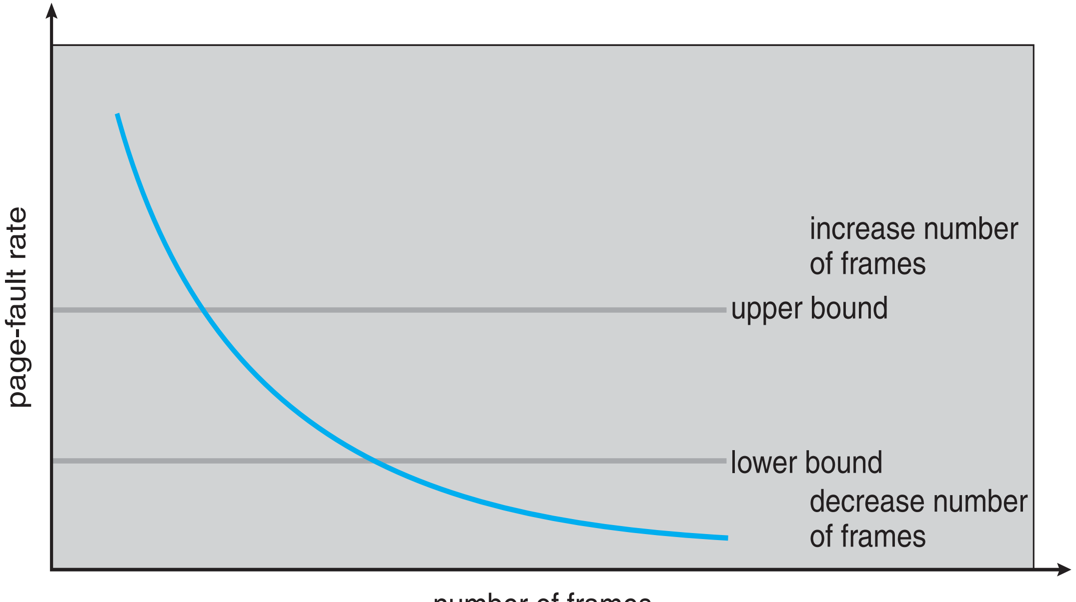


Keeping Track of the Working Set

- 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 referenc e 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

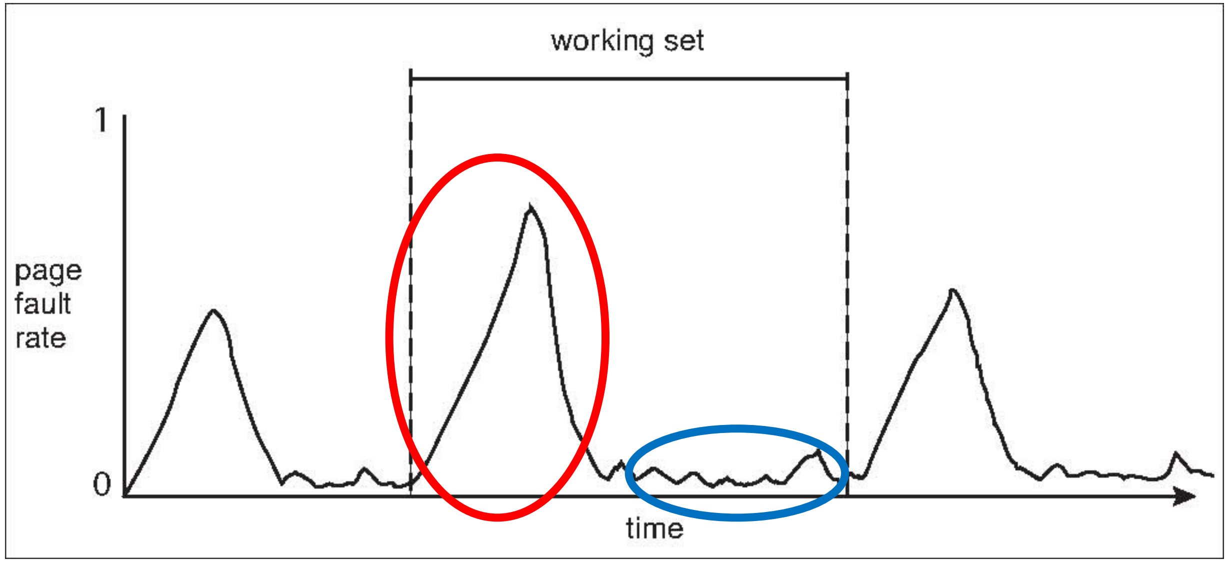
Page-Fault Frequency

- 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



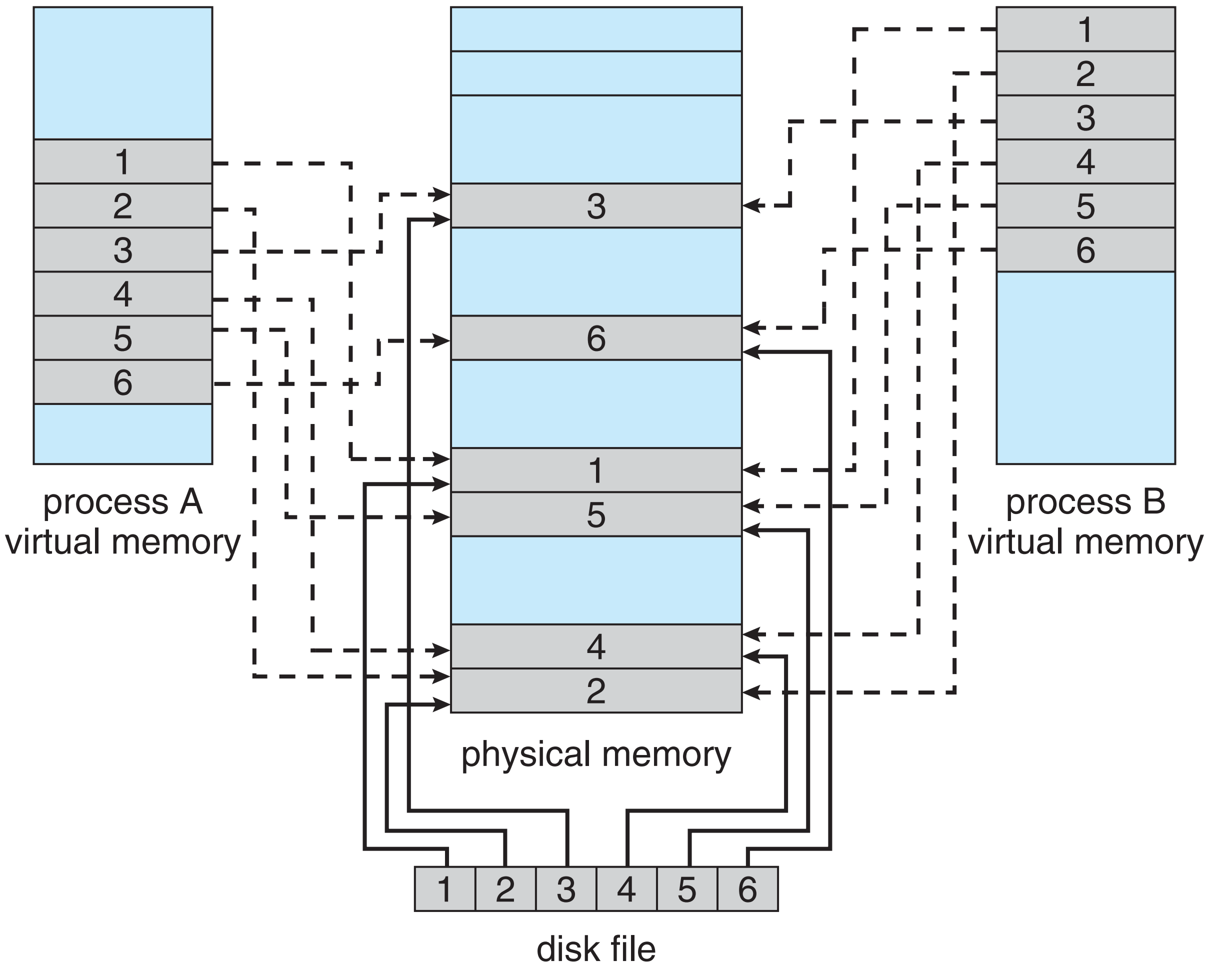
8.7 Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memor y access by mapping a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rat her than read() and write() system calls
- Also allows several processes to map the same file allowing the pages i n memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file close() time
  - For example, when the pager scans for dirty pages

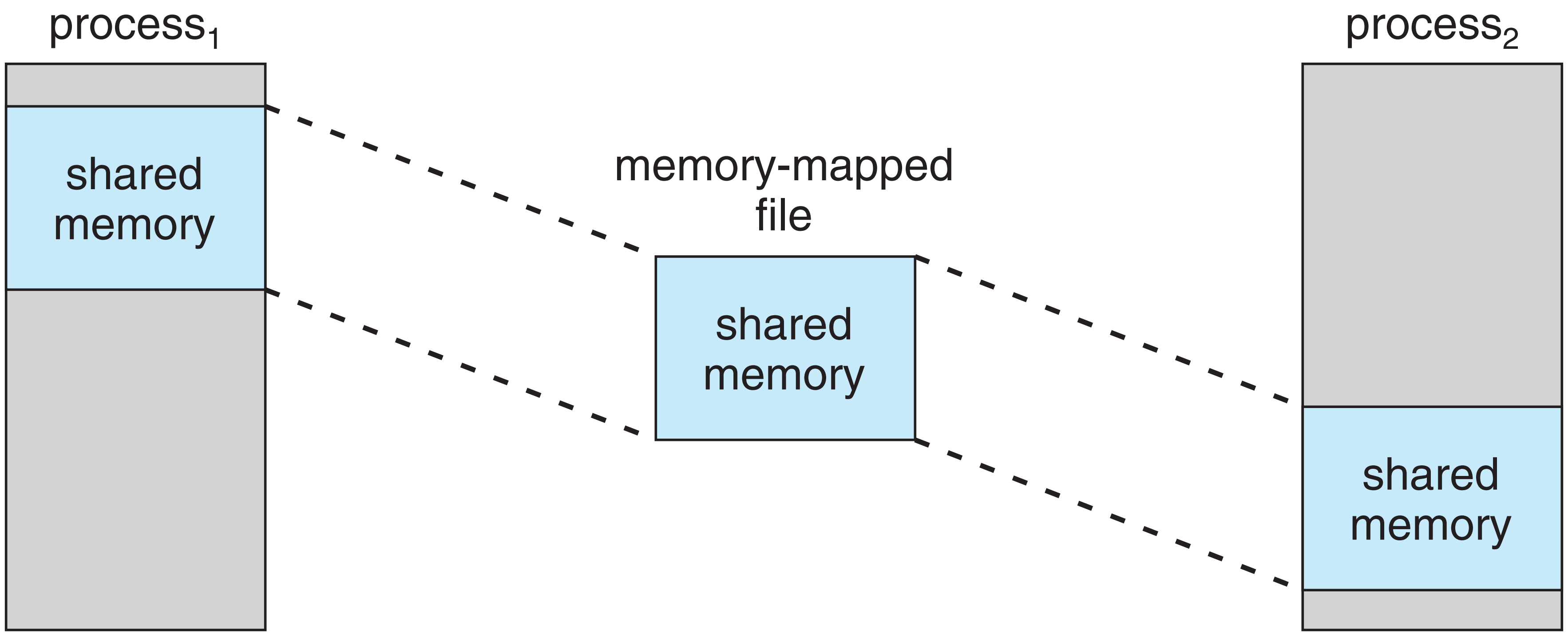
Memory-Mapped File Technique for all I/O

- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via mmap() s ystem call
    - Now file mapped into process address space
- For standard I/O (open(), read(), write(), close()), mmap anyway
    - But map file into kernel address space
    - Process still does read() and write()
        - Copies data to and from kernel space and user space
    - Uses efficient memory management subsystem
        - Avoids needing separate subsystem
- Memory mapped files can be used for shared memory (although again via separate system calls)

Memory Mapped Files



Shared Memory via Memory-Mapped I/O

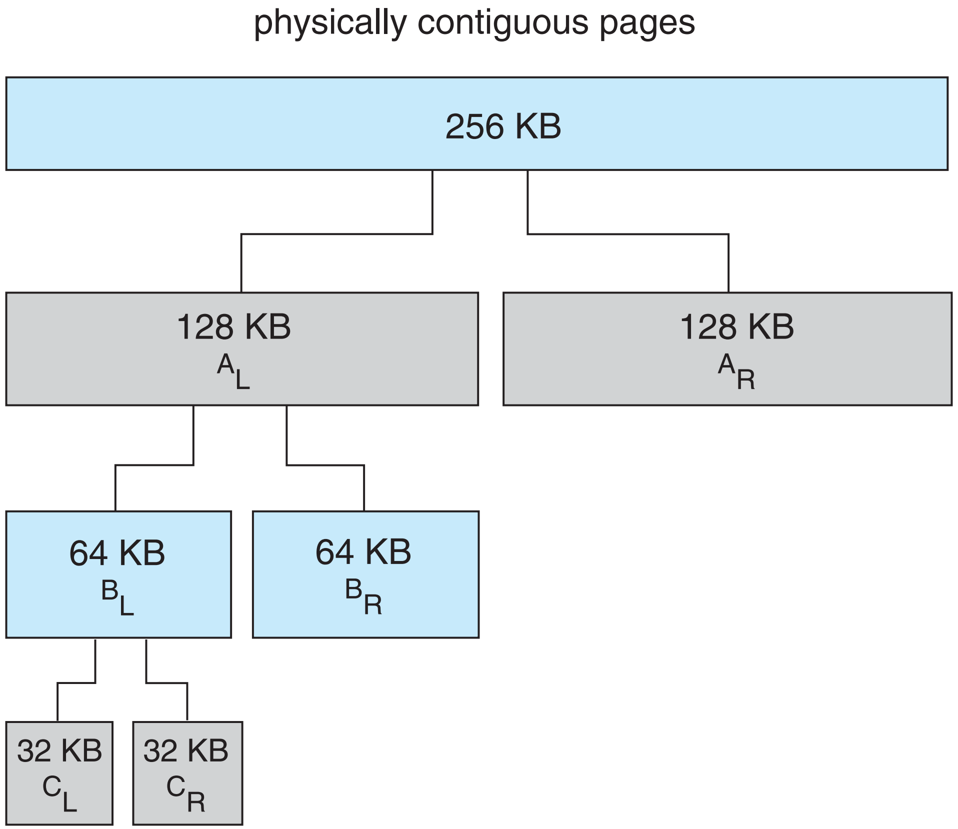


8.8 Allocating Kernel Memory

- 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

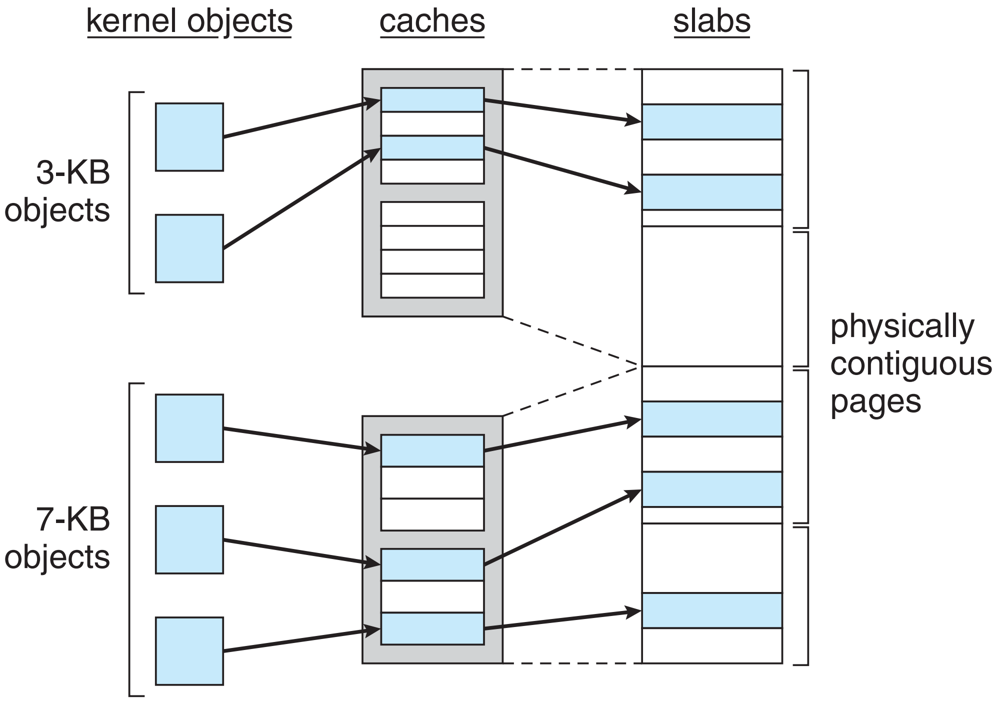
Buddy System

- Allocates memory from fixed-size segment c onsisting 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-lowe r power of 2
    - Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation



Slab Allocator

- Slab is one or more physically contiguou s pages
- Cache consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with objects – instantiatio ns 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 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
  - Full – all used
  - Empty – all free
  - Partial – mix of free and used
- Upon request, slab allocator
  - Uses free struct in partial slab
  - If none, takes one from empty slab
  - If no empty slab, create new empty

8.9 Other Considerations -- Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process sta rtup
  - Prepage all or some of the pages a process will need, before they ar e 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

Other Issues – Page Size

- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - I/O overhead
  - Resolution
  - Number of page faults
  - Locality
- 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

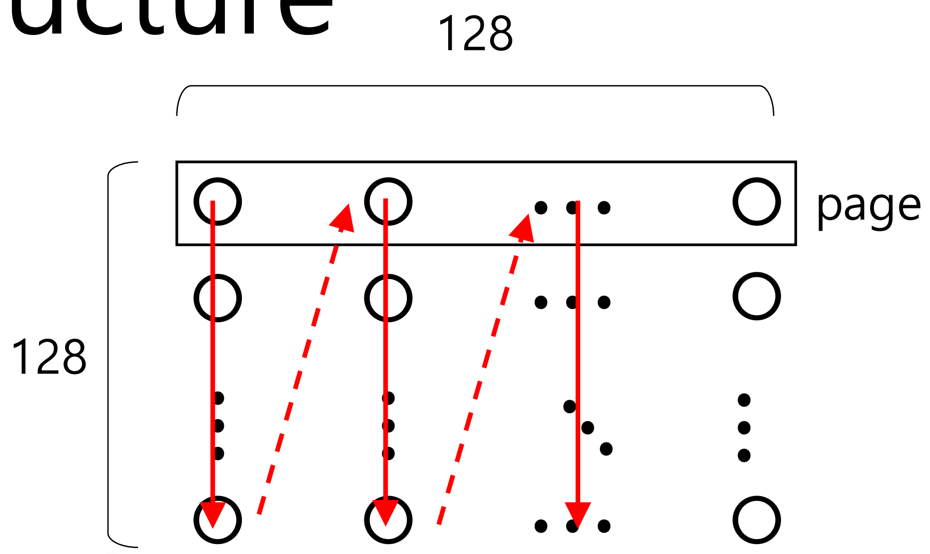
Other Issues – TLB Reach

- 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 pa ge size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them with out an increase in fragmentation

Other Issues – Program Structure

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

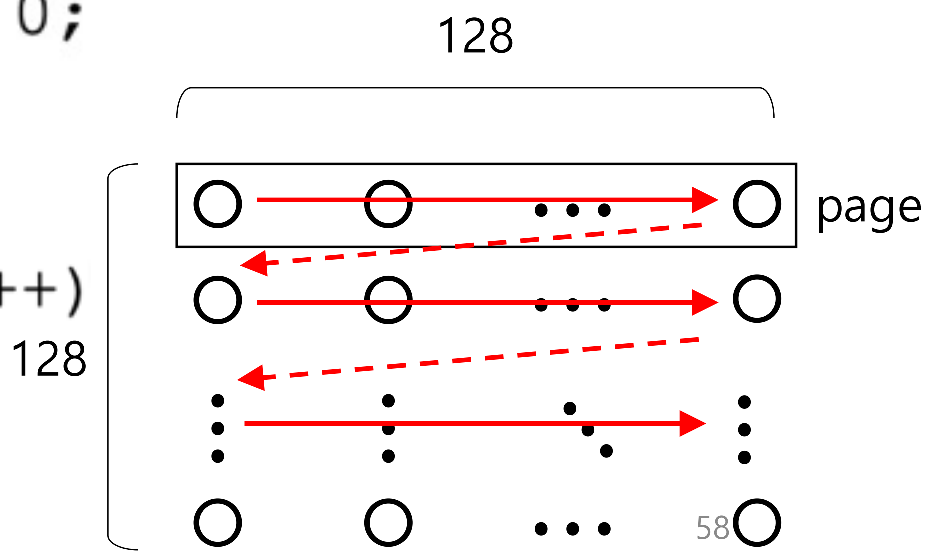
```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```



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

Other Issues – 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