Chapter 7

VIRTUAL MEMORY

Contents

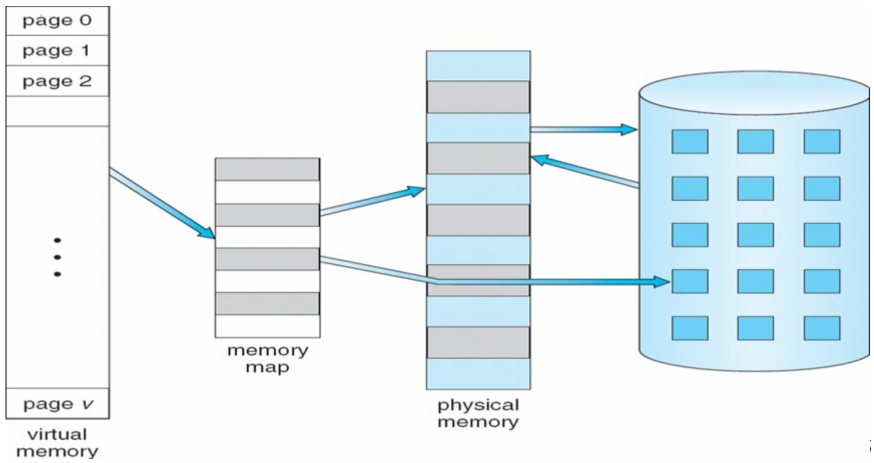
Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model

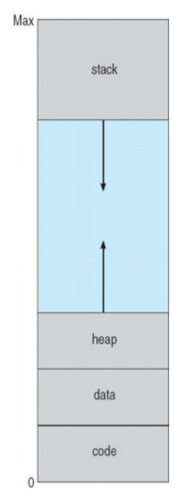
Background

- 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
- Virtual memory can be implemented via:
  - ✓ Demand paging
  - ✓ Demand segmentation

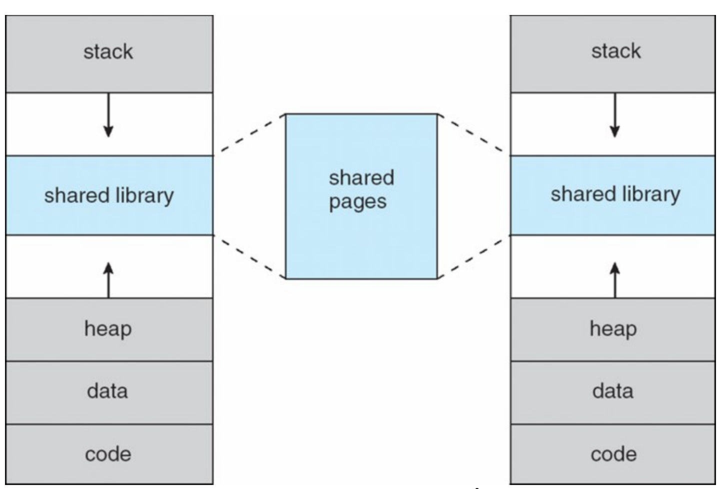
- Virtual Memory That is Larger Than Physical Memory



Virtual-address Space



- Shared Library Using 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
- 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

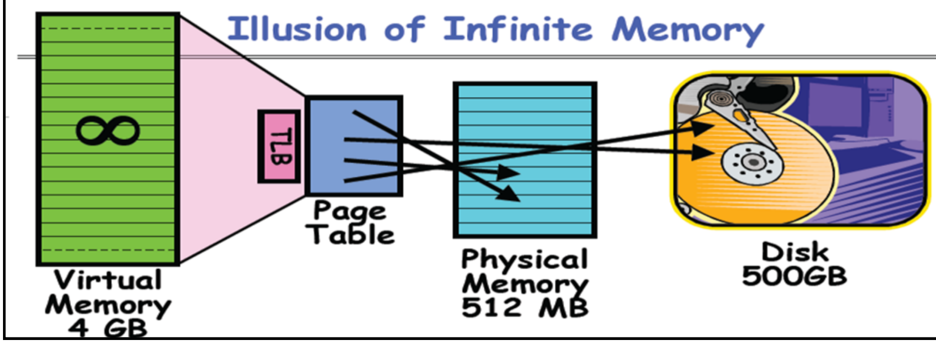
Đinh Công Đoan 7

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30% per year
- But they don't use all their memory all the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory
- Solution: use main memory as a cache for disk

Đinh Công Đoan 8

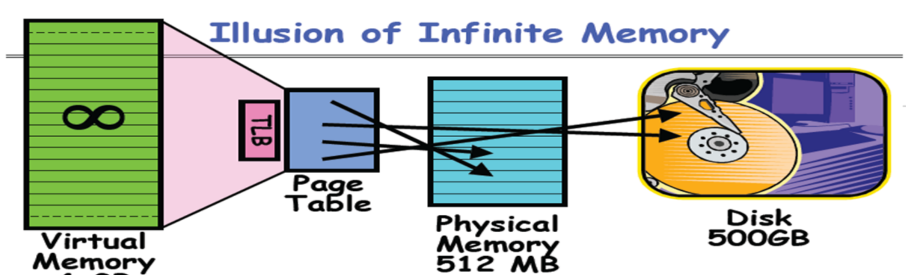
Demand Paging

- Disk is larger than physical memory, so in-use virtual memory can be bigger than physical memory
  - ✓ Combined memory of running processes can also be much larger than physical memory
  - ✓ More programs fit into memory, allowing more concurrency



Demand Paging

- Principle: transparent level of indirection (page table)
  - ✓ Supports flexible placement of physical data
    - Data could be on disk, or even across network
  - ✓ Variable location of data transparent to user program
    - Performance issue, not correctness issue



Demand Paging

- Demand Paging is Caching:
  - What is block size? (1 page)
  - What is the organization?
    - Fully associative: arbitrary virtual → physical mapping
  - How do we find a page in the cache?
    - Check TLB, then page-table
  - What is page replacement policy (LRU, random, etc.)?
    - This is today's topic!
  - What happens on a miss?
    - Page fault: go to disk, and possibly swap a page out to disk to make room
  - What happens on a write (write-through, write-back)
    - Definitely write-back! Need a dirty bit!

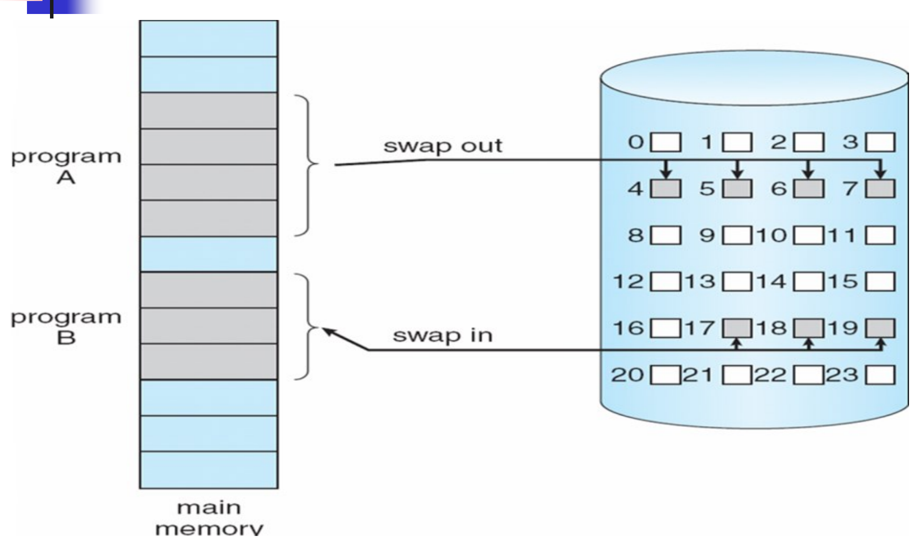
Demand Paging

- PTE helps us implement paging
  - Valid/invalid bit indicates if page is in memory or on disk
- Suppose user references page with invalid PTE?
  - MMU traps to OS as a page fault
  - What does the OS do?
    - Choose an old page to replace
    - If old page is dirty, write contents back to disk
    - Invalidate PTE and any cached TLB entry for old page
    - Load new page into memory from disk
    - Update PTE
    - Continue thread from original faulting location
  - TLB for new page will be loaded when thread continues
  - While pulling pages off disk for one process, OS runs another process from ready queue
    - After all, the faulting process is essentially blocking on disk I/O
    - Suspended process sits on wait queue

Review: Software-Loaded TLB

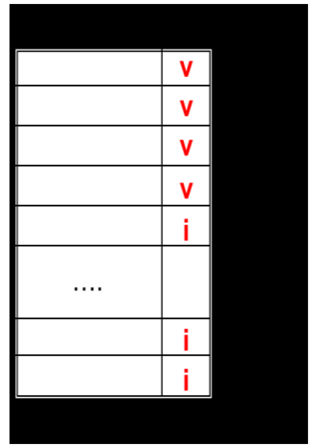
- MIPS/SIMICS/Nachos TLB is loaded by software
  - High TLB hit rate: ok to trap to software to fill the TLB, even if slower
  - Simpler hardware and added flexibility: software can maintain translation tables in whatever convenient format
- How can a process run without access to page table?
  - Fast path (TLB hit with valid = 1)
    - Translation to physical page done by hardware
  - Slow path (TLB hit with valid = 0 or TLB miss)
    - Hardware receives a TLB fault
  - What does the OS do on a TLB Fault?
    - Traverse page table to find appropriate PTE
      - If valid = 1, load PTE into TLB, continue thread
      - If valid = 0, perform Page Fault, then continue thread
- Everything is transparent to the user process
  - It doesn't know about paging to/from disk
  - It doesn't even know about software TLB handling

Transfer of a Paged Memory to Contiguous Disk Space

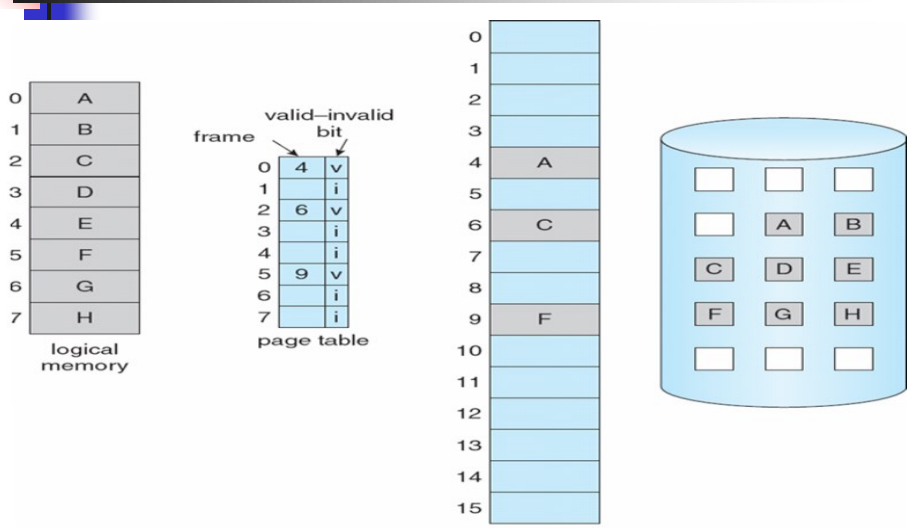


Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
(v ⇒ in-memory, i ⇒ not-in-memory)
- Initially valid–invalid bit is set to i on all entries
- Example of a page table snapshot:
- During address translation, if valid–invalid bit in page table entry is I ⇒ page fault



Page Table When Some Pages Are Not in Main Memory

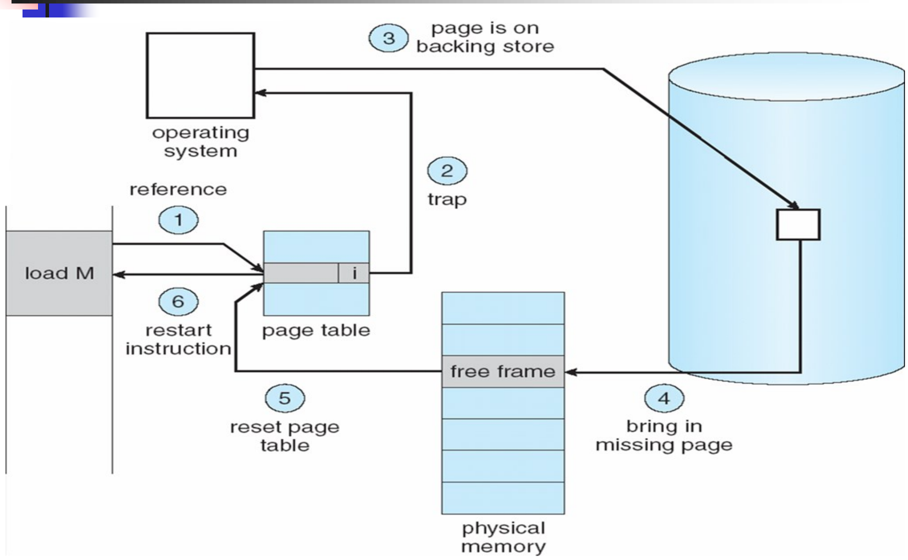


Page Fault

- 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:
   - ✓ Invalid reference ⇒ abort
   - ✓ 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

Steps in Handling a Page Fault



Re-Run the Faulting Instruction?

- Yes, but what if the instruction has side effects?
  - Options: Unwind side-effects or finish them off?
  - Example: mov (sp)+, 10
    - What if page fault occurs when writing to the stack pointer? Was sp already incremented?
  - Example: strcpy (r1), (r2)
    - Source and destination overlap: can't unwind in principle!
    - IBM S/370 and VAX solution: execute twice – once read-only
  - Example: RISC delayed branches
    div r1, r2, r3
    ld r1, sp
    - Need to track PC and nPC
  - Example: Delayed Exceptions
    div r1, r2, r3
    ld r1, sp
    - Takes many cycles to detect divide-by-zero, but ld caused the page fault

Precise Exceptions

- State of the machine is preserved as if the program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations
  - Difficult in the presence of pipelining, out-of-order execution, etc.
  - MIPS takes this position
- Imprecise: system software figures out where it is and puts everything back together
- Performance goals often lead designers to forsake precise interrupts (unfortunately)
- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - ✓ if $p = 0$ no page faults
  - ✓ if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

$$\begin{aligned} EAT = (1 - p) \text{ x memory access} \\ + p \text{ (page fault overhead} \\ + \text{ swap page out} \\ + \text{ swap page in} \\ + \text{ restart overhead)} \end{aligned}$$

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!!

Demand Paging

- Does software-loaded TLB need a use bit?
- Two options
  - ✓ Hardware sets use bit in TLB
    - When TLB entry is replaced, software copies use bit back to page table
  - ✓ Software manages TLB entries as FIFO list
    - Everything not in TLB is Second-Chance list, managed as strict LRU
- Core Map
  - ✓ Page tables map virtual page → physical page
  - ✓ Do we need a reverse mapping (i.e. physical page → virtual page)?
    - Yes, clock algorithm runs through page frames. If sharing, then multiple virtual-pages per physical frame can exist
    - Can't push page out to disk without invalidating all of the PTEs

Process Creation

- Virtual memory allows other benefits during process creation:
  - ✓ Copy-on-Write
  - ✓ Memory-Mapped Files (later)
- 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
  - ✓ Free pages are allocated from a pool of zeroed-out pages

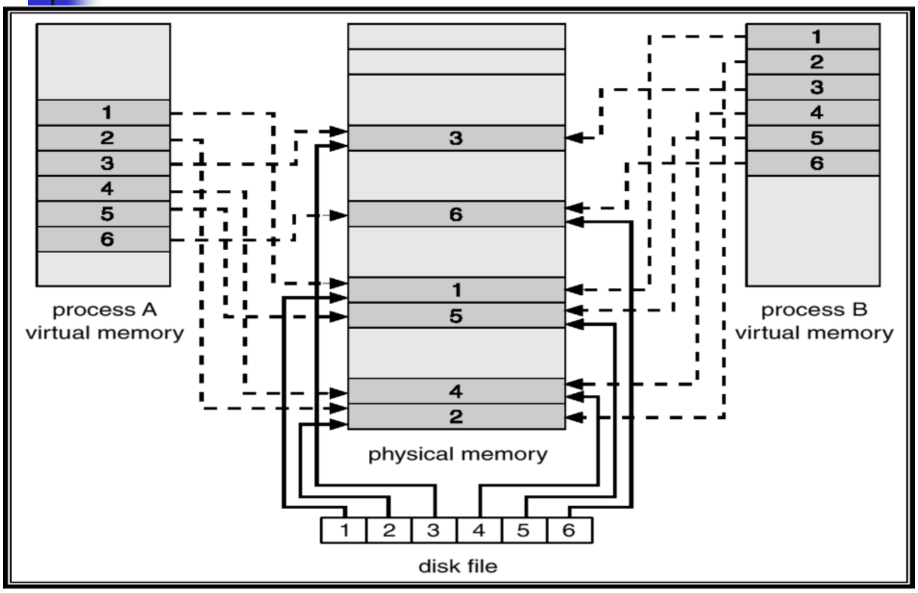
Copy – on - write

- 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
- Free pages are allocated from a pool of zeroed-out pages

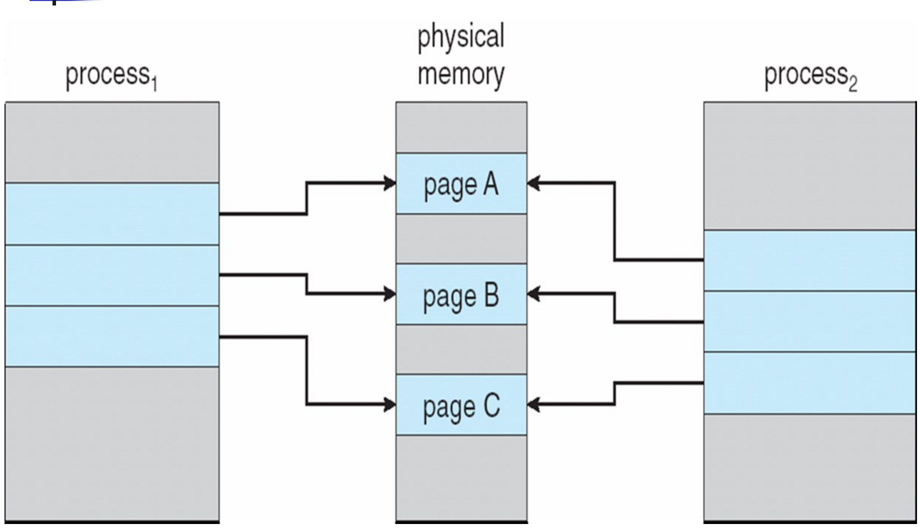
Memory – mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory.
- A file is initially read using demand paging. A pagesized 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 file access by treating file I/O through memory rather than **read() write()** system calls.
- Also allows several processes to map the same file allowing the pages in memory to be shared

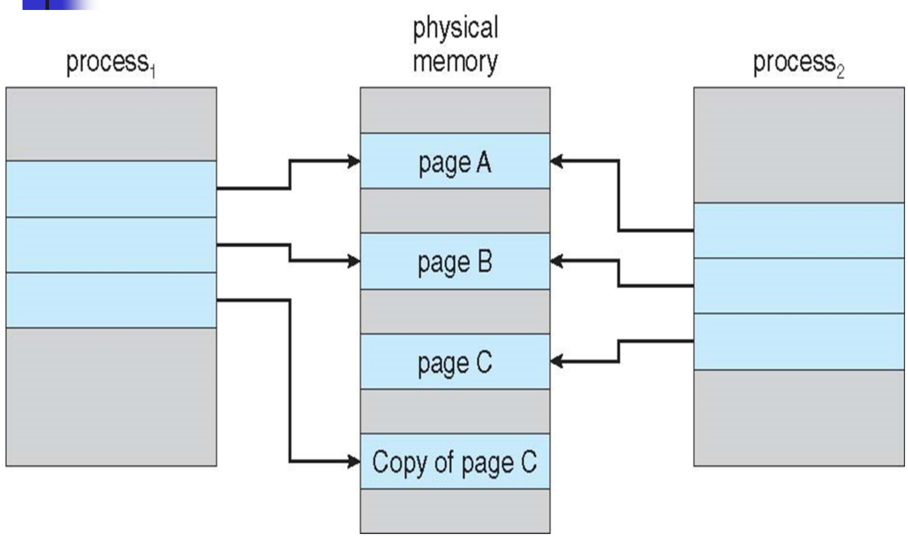
Memory mapped files



Before Process 1 Modifies Page C



After Process 1 Modifies Page C



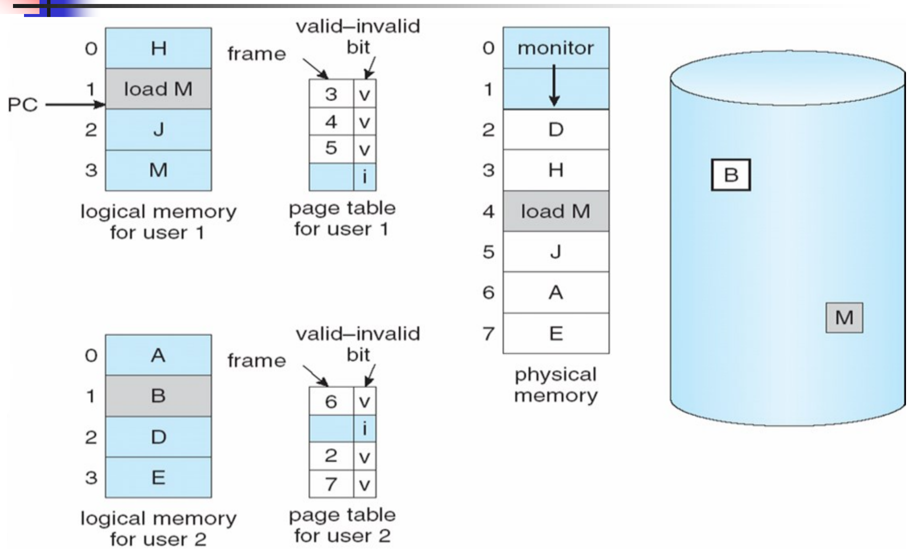
What happens if there is no free frame?

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

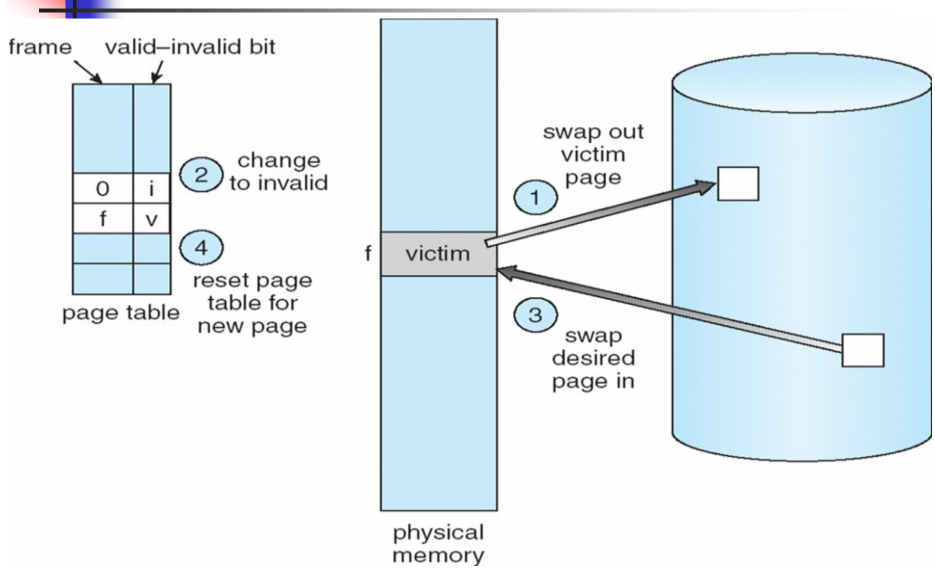
Need For Page Replacement



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

- Replacement is a performance issue: the cost of being wrong is high (going to disk)
- Keep "important" pages in memory, not toss them out
- FIFO
  - ✓ Throw out oldest page. Be fair – let every page live in memory for same amount of time
  - ✓ Bad, because throws out heavily used pages instead of infrequently used ones
- MIN
  - ✓ Replace page that won't be used for the longest time
  - ✓ Great, but can't know the future
  - ✓ Used for comparison
- RANDOM
  - ✓ Simple: typical solution for TLB's
  - ✓ Unpredictable, makes it hard to make real-time guarantees

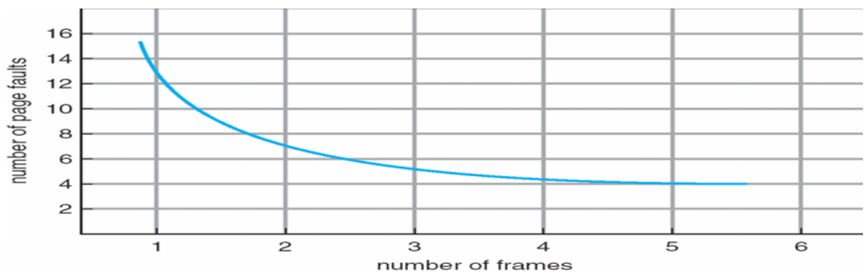
Page Replacement Algorithms

- 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
- In all our examples, the reference string is

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

Graph of Page Faults Versus The Number of Frames

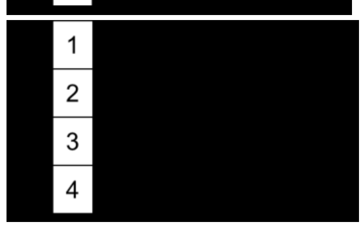
- Ideally, adding memory, the miss rate should go down (above)
- Is this always the case?
  - It would seem so!
- Belady's Anomaly: Some replacement algorithms don't have this obvious property
  - FIFO



First-In-First-Out (FIFO) Algorithm

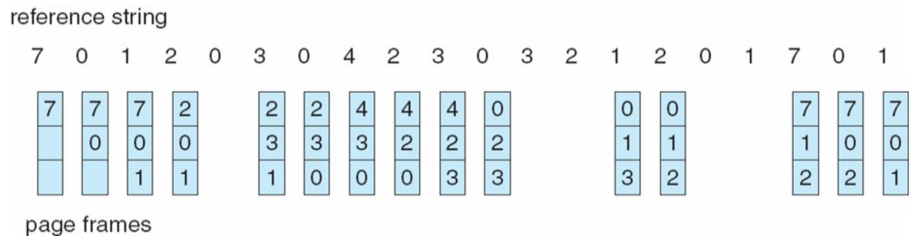
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

- 4 frames

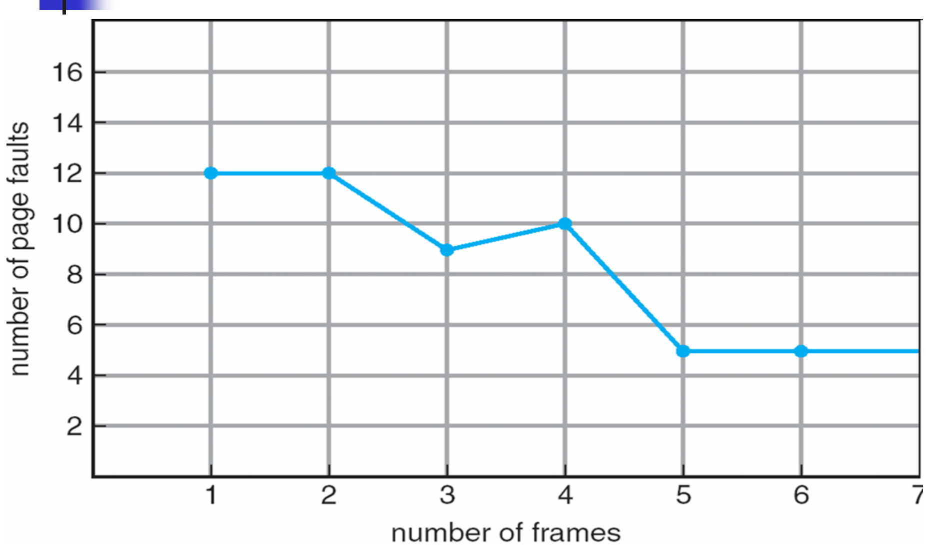


- Belady's Anomaly: more frames ⇒ more page faults

FIFO Page Replacement



FIFO Illustrating Belady's Anomaly



Belady's Anomaly

Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

MIN: Optimal Algorithm

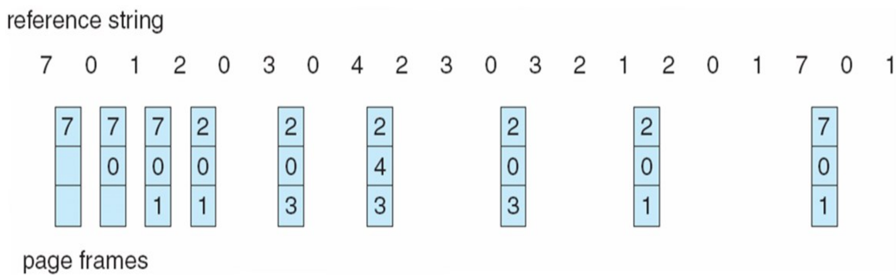
- Replace page that will not be used for longest period of time
- 4 frames example

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

1
2
3
4

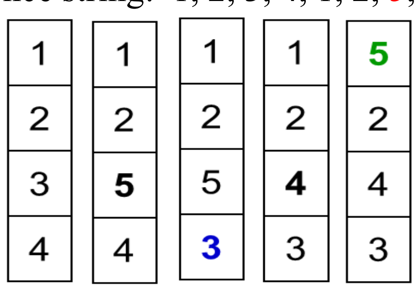
- How do you know this?
- Used for measuring how well your algorithm performs

MIN: Optimal Page Replacement



Least Recently Used (LRU) Algorithm

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



- List Implementation
  - ✓ On each use, remove page from list and place it at head
  - ✓ LRU page is at the tail
- But need to traverse the list to move a page from the middle to the head

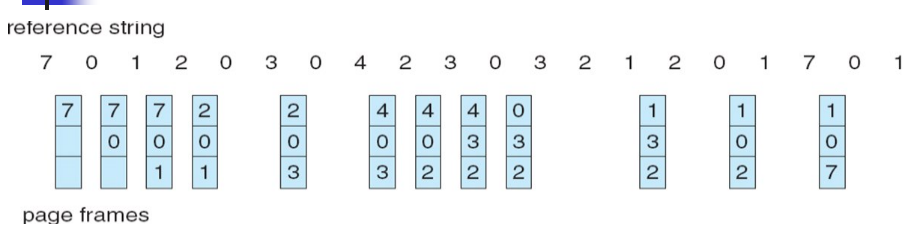
Least Recently Used (LRU) Algorithm

- 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

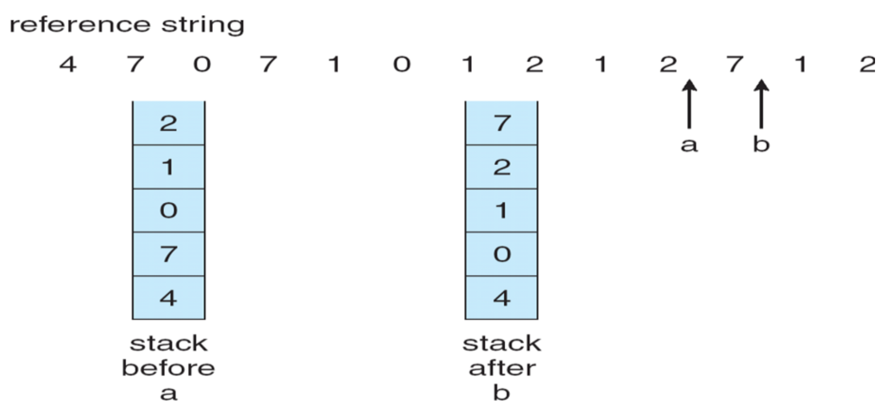
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

LRU Page Replacement



- 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

- Use Of A Stack to Record The Most Recent Page References



Comparing the Algorithms

- FIFO: Suppose we have 3 frames, 4 pages, and the following reference stream
  - ✓ A B C A B D A D B C B
- 7 Faults
  - ✓ When referencing D, replacing A is a bad choice, since we need A again right away

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

Comparing the Algorithms

- MIN: Suppose we have 3 frames, 4 pages, and the following reference stream
  - ✓ A B C A B D A D B C B
- 5 Faults
  - ✓ D is brought in to the page that will not be referenced for the longest time
- How does this compare to LRU?

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

When Will LRU Perform Poorly?

- Consider
  - ✓ A B C D A B C D A B C D
- Every LRU reference is a page fault!

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

- Consider
  - ✓ A B C D A B C D A B C D
- MIN does much better

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A									B		
2		B					C					
3			C	D								

Least Recently Used (LRU) Algorithm

- Due to locality, it would seem that LRU would be a good candidate to approximate MIN
- But LRU is hard to implement, takes up a lot of space or time.
- In practice, we approximate LRU instead.

LRU Approximation Algorithms

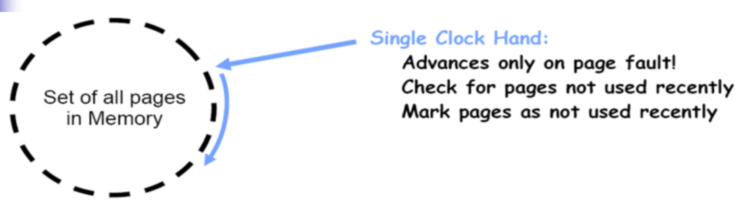
- Reference bit
  - ✓ With each page associate a bit, initially = 0
  - ✓ When page is referenced bit set to 1
  - ✓ Replace the one which is 0 (if one exists)
    - We do not know the order, however
- Second chance
  - ✓ Need reference bit
  - ✓ Clock replacement
  - ✓ If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

Clock Algorithm

- Arrange physical pages in circle with single clock hand
  - ✓ Replace an old page, not the oldest page
  - ✓ Approximation to LRU, hence an approximation to an approximation of MIN
- Details

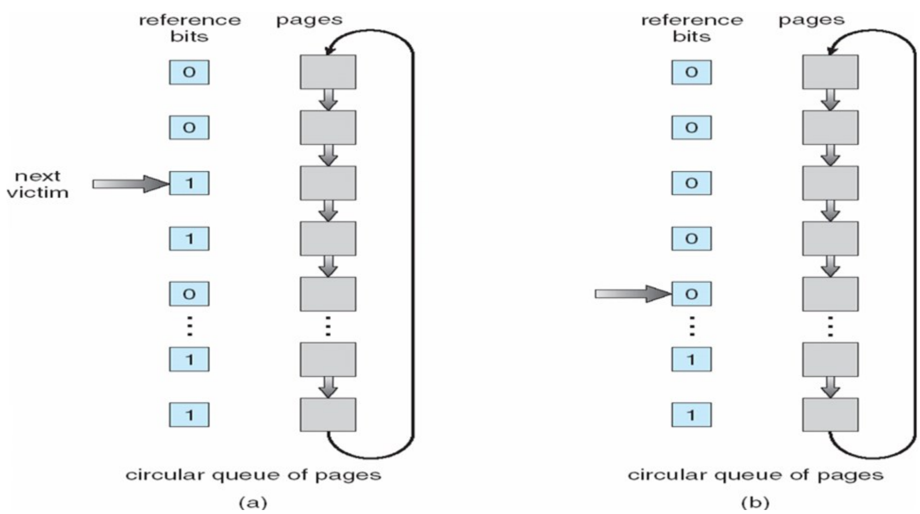
  - ✓ Hardware "use" bit per physical page
    - Hardware sets use bit on each reference
    - If use bit isn't set, means not referenced for a long time
    - Nachos hardware sets use bit in the TLB, you have to copy this back to page table when TLB entry gets replaced
  - ✓ On Page Fault
    - Advance clock hand (not real time)
    - Check use bit: 1 == used recently; clear and leave alone, 0 == selected candidate for replacement
    - Even if all use bits set, will eventually loop around; in which case, resort to FIFO

Clock Algorithm



- What if hand moving slowly?
  - ✓ Good sign or bad sign?
    - Not many page faults and/or find page quickly
- What if hand moving quickly?
  - ✓ Lots of page faults and/or lots of reference bits set
- One way to view clock algorithm
  - ✓ Crude partitioning of pages into two groups: young and old
  - ✓ Why not partition into more groups?
    - Clock Algorithm a.k.a. "Second Chance" – could have Nth chance, too!
      - Counting algorithms

Second-Chance (clock) Page-Replacement Algorithm



Counting Algorithms: Nth Chance Version of Clock Algorithm

- Nth Chance Algorithm: OS keeps counter per page: # sweeps
- On page fault, OS checks use bit:
  - 1 == clear use and also clear counter (used in last sweep)
  - 0 == increment counter, if count = N, replace page
- So the clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N?
  - Why pick small N?
- What about dirty pages?

- Nth Chance Algorithm: OS keeps counter per page: # sweeps
- On page fault, OS checks use bit:
  - 1 == clear use and also clear counter (used in last sweep)
  - 0 == increment counter, if count = N, replace page
- So the clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N?
    - Better approximation to LRU: If N ~ 1K, really good approximation
  - Why pick small N?
    - Might have to look a long way to find a free page with large N
- What about dirty pages?

Counting Algorithms: Nth Chance Version of Clock Algorithm

- What about Dirty Pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach
    - Clean pages, use N = 1
    - Dirty pages, use N = 2, and write back to disk when N = 1
- 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

Clock Algorithms: Details

- Which bits of a PTE entry are useful to us?
  - Use: set when a page is referenced; cleared by clock algorithm
  - Modified (Dirty): set when a page is modified, cleared when a page is written to disk
  - Valid: ok for program to reference this page
  - Read-Only: ok for program to read page, but not modify
    - For example, for catching modifications to code pages
- Do we really need hardware-supported modified bit?
  - No, we can emulate it using a read-only bit

Clock Algorithms: Details

- Which bits of a PTE entry are useful to us?
  - Use: set when a page is referenced; cleared by clock algorithm
  - Modified (Dirty): set when a page is modified, cleared when a page is written to disk
  - Valid: ok for program to reference this page
  - Read-Only: ok for program to read page, but not modify
    - For example, for catching modifications to code pages
- Do we really need hardware-supported modified bit?
  - No, we can emulate it (BSD Unix) using a read-only bit
    - Initially, mark all pages as read only (even the data pages)
    - On write, trap to the OS, OS software sets software modified bit, and marks page as read-write
    - When page comes back in from disk, mark read-only

Clock Algorithms: Details

- Do we really need a hardware-supported use bit?
  - No, we can emulate it similar to the modified bit
    - Mark all pages as invalid, even if in memory
    - On read to invalid page, trap to OS
    - OS sets use bit, and marks page read-only
  - Get modified bit in same way as previous
    - On write, trap to OS (either invalid or read-only)
    - Set use and modified bits, mark page read-write
  - When clock hand passes by, reset use and modified bits and mark page as invalid again
- Remember, however, that clock is just an approximation of LRU
  - Can we do a better approximation, given that we have to take page faults on some reads and writes to collect use information?
  - Need to identify old page, not the oldest page!
  - Answer: second chance list

Second-Chance List Algorithm

- VAX/VMS approach
- Split memory in two: Active List (RW), Second Chance list (Invalid)
- Access pages in Active List at full speed
- Otherwise, Page Fault
  - Move overflow page from end of Active List to front of Second Chance list
  - Mark overflow page invalid
  - If desired page is on the Second Chance list:
    - Move to front of Active List
    - Mark RW
  - Else:
    - Page in to front of Active List
    - Mark RW
    - Page out LRU victim at the end of the Second Chance list

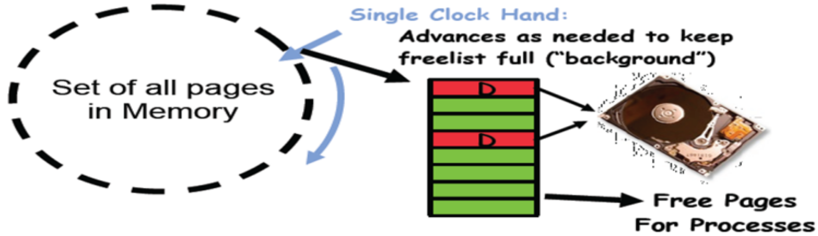
Second-Chance List Algorithm

- How many pages for second chance list?
  - If 0 ➔ FIFO
  - If all ➔ LRU, but page fault on every reference
- Pick intermediate value. Result is:
  - Few disk accesses (page only goes to disk if unused for a long time) (+)
  - Increased overhead trapping to OS (software/hardware tradeoff) (-)
- With page translation, we can adapt to any kind of access the program makes
  - Can use page translation / protection to share memory between threads on widely separated machines
- Why didn't VAX just include a "use" bit?
  - Strecker (architect) asked OS people, they said they didn't need it, so didn't implement it
  - He later got blamed, but VAX did ok anyway

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

Free List



- Keep set of free frames ready for use in demand paging
  - Free list filled in background by clock algorithm or other technique (clock daemon)
  - Dirty pages start copying back to disk when they enter the list
- Like VAX second-chance list
  - If page needed before reused, just return to the active set
- Advantage: faster for page fault
  - Can always use page (or pages) immediately on fault

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

$$s_i = \text{size of process } p_i$$
$$S = \sum s_i$$
$$m = \text{total number of frames}$$
$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$
$$m = 64$$
$$s_i = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

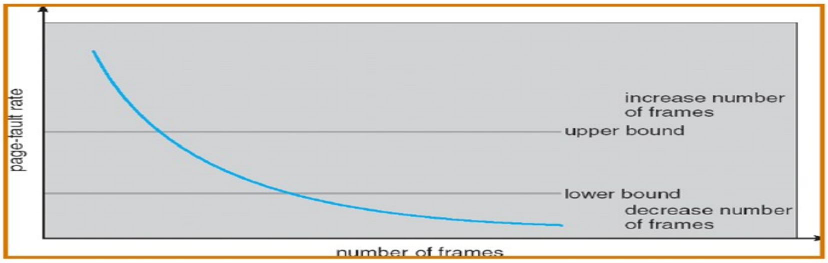
- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global vs. Local Replacement

- 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
- Priority Allocation may, if using global replacement, select for replacement a frame from a process with lower priority.

Adaptive Allocation based on Page Fault Frequency

- Can we reduce capacity misses by dynamically changing the number of pages per application?

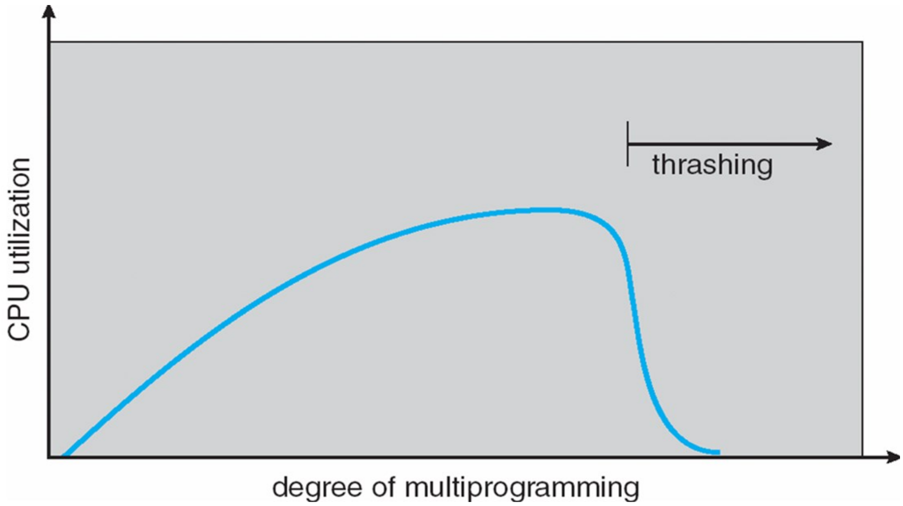


- Establish "acceptable" page fault rate
  - ✓ If rate is too low, process loses a frame to a process that needs it
  - ✓ If rate is too high, process gains a frame
- But what if we just don't have enough memory?

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" (number of ready processes)
  - ✓ another process added to the ready queue
    - Which requires more memory, leading to more page faults, leading to lower CPU utilization, and the cycle continues…
- Thrashing ≡ a process is busy swapping pages in and out
- How do we detect and respond to thrashing?

Thrashing (Cont.)



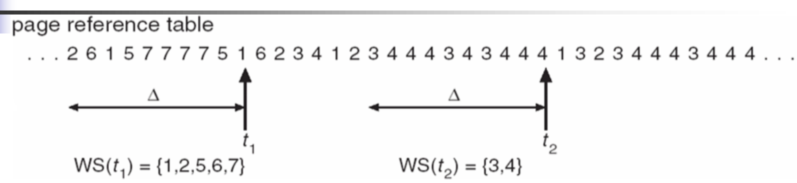
Demand Paging and Thrashing

- 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

- Program memory access patterns have temporal and spatial locality
  - ✓ Group of pages accessed along a given time slice is called the "Working Set"
  - ✓ Working Set defines the minimum number of pages needed for the process to "behave well"
- Not enough memory for the working set ➔ Thrashing
  - ✓ Better to swap out process?

Working-Set Model



- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references Example: 10,000 instruction
- 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
- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend one of the processes

What About Compulsory Misses?

- Recall that compulsory misses are misses that occur the first time that a page is seen
  - ✓ Page that are touched for the first time
  - ✓ Pages that are touched after process is swapped out/swapped back in
- Clustering:
  - ✓ On a page fault, bring in multiple pages “around” the faulting page
  - ✓ Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
- Working Set Tracking
  - ✓ Use algorithm to try to track working set of an application
  - ✓ When swapping process back in, swap in working set

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - ✓ 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

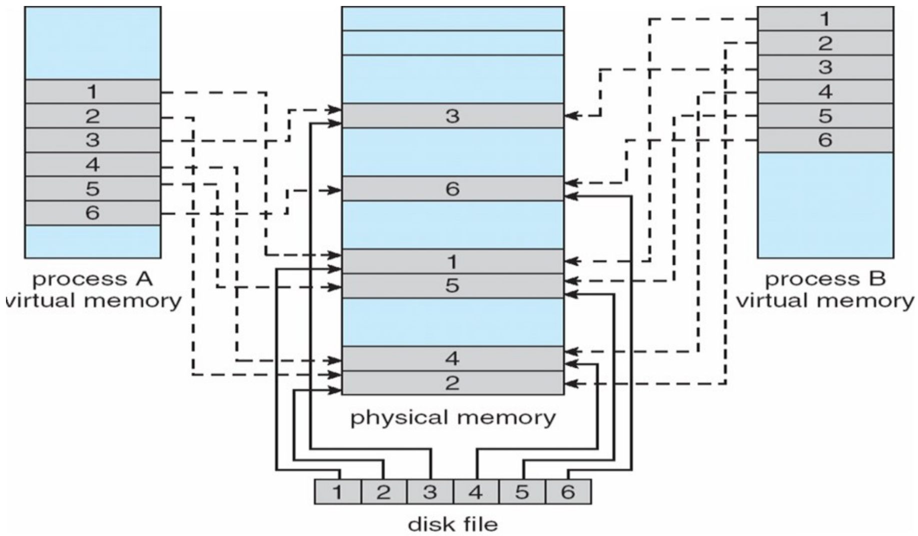
Paging Summary

- Replacement Policies
  - FIFO
  - MIN
  - LRU
    - Clock Algorithm
    - Nth Chance Clock Algorithm
      - Second-Chance List Algorithm
- Working Set
  - Set of pages touched by a process recently
- Thrasing
  - A process is busy swapping pages in and out
  - Process will thrash if working set doesn't fit in memory
  - Need to swap out a process

Memory-Mapped Files

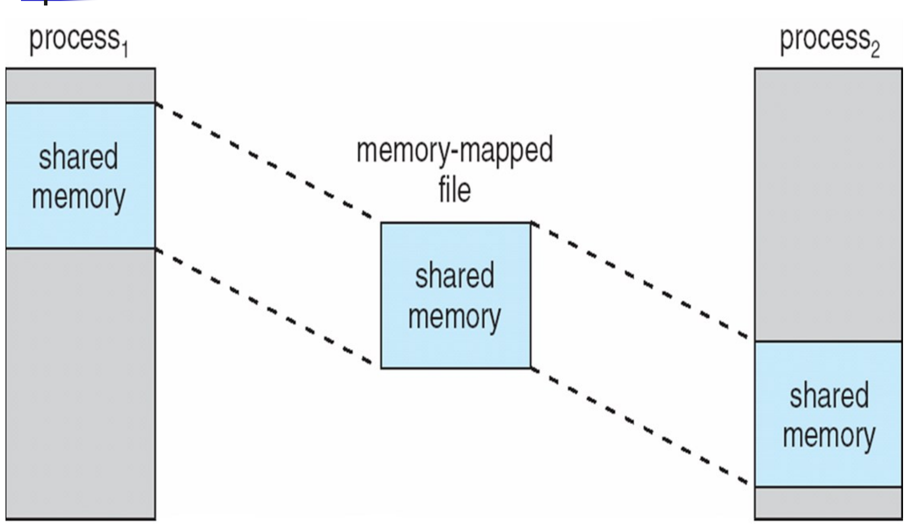
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than `read()` `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

Memory Mapped Files



Đinh Công Đoan 79

Memory-Mapped Shared Memory in Windows



Đinh Công Đoan 80

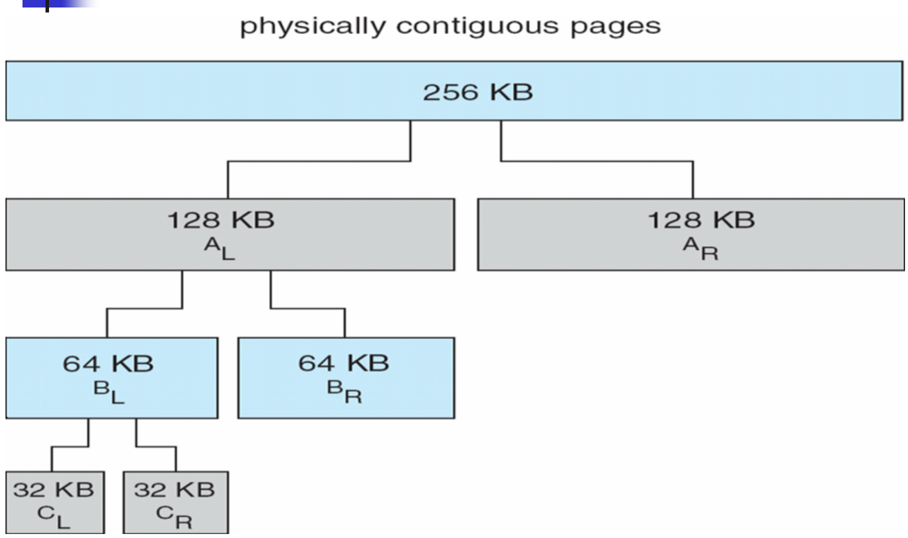
Allocating Kernel Memory

- 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

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

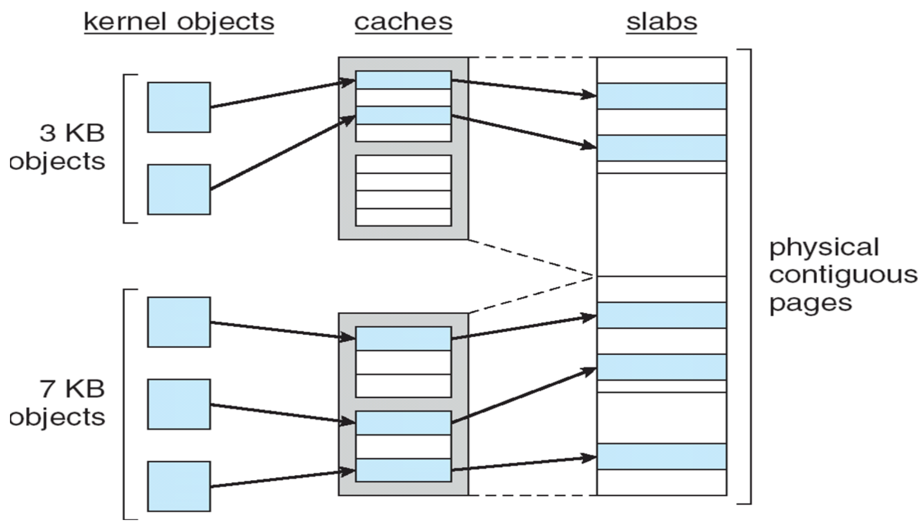
Buddy System Allocator



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



Other Issues -- Prepaging

- 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

Other Issues – Page Size

- Page size selection must take into consideration:
  - ✓ fragmentation
  - ✓ table size
  - ✓ I/O overhead
  - ✓ locality

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 page size
- Provide Multiple Page Sizes
  - ✓ This allows applications that require larger page sizes the opportunity to use them without 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

Reason Why Frames Used For I/O Must Be In Memory