東南大學
SOUTHEAST UNIVERSITY

OPERATING SYSTEM CONCEPTS

Chapter 9. Virtual Memory

A/Prof. Kai Dong

Warm-up

What Happens when OS is Booting?

OS @boot (kernel mode)	**Hardware**
initialize *trap table*	
	remember addresses of ...
	system call handler
	timer handler
	illegal mem-access handler
	illegal instruction handler
start interrupt timer	
	start timer; interrupt after *X* ms
initialize process table	
initialize free list	

Warm-up

What Happens when OS is Running?

OS @run (kernel mode)	**Hardware**	**Program (user mode)**
To start process A:		
allocate entry in process table		
allocate memory for program		
set *base/limit registers*		
return-from-trap **(into A)**		
	restore registers of A	
	move to *user mode*	
	jump to A's (initial) PC	
		Process A runs
		fetch instruction
	translate virtual address and perform fetch	
		execute instruction
	if explicit load/store:	
	ensure address is in-limit;	
	translate virtual address and perform load/store	
	. . .	
	Timer interrupt	
	move to *kernel mode*	
	jump to interrupt handler	

Warm-up

What Happens when an Exception Takes Place?

OS @run **(kernel mode)**	**Hardware**	**Program** **(user mode)**
	Timer interrupt move to *kernel mode* jump to interrupt handler	
Handle the *trap* call *switch*() routine save regs(A) to proc-struct(A) (including *base/limit*) restore regs(B) from proc-struct(B) (including *base/limit*) ***return-from-trap*** **(into B)**		
	restore regs of B move to *user mode* jump to B's PC	
		Process B runs execute bad load
	load is out-of-limit move to *kernel mode* jump to trap handler	
Handle the *trap* decide to terminate process B de-allocate B's memory free B's entry in process table		

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
- To examine the relationship between shared memory and memory-mapped files
- To explore how kernel memory is managed

Contents

Contents

Background

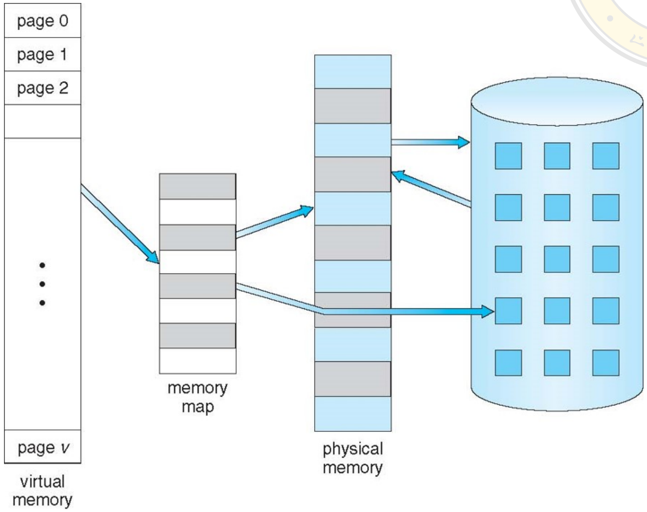
- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each 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

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
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- Backing store
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

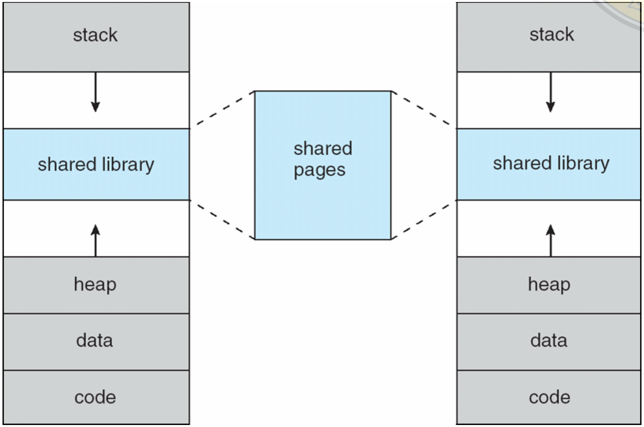
Background

Virtual Memory that is Larger than Physical Memory



Background

Shared Library Using Virtual Memory

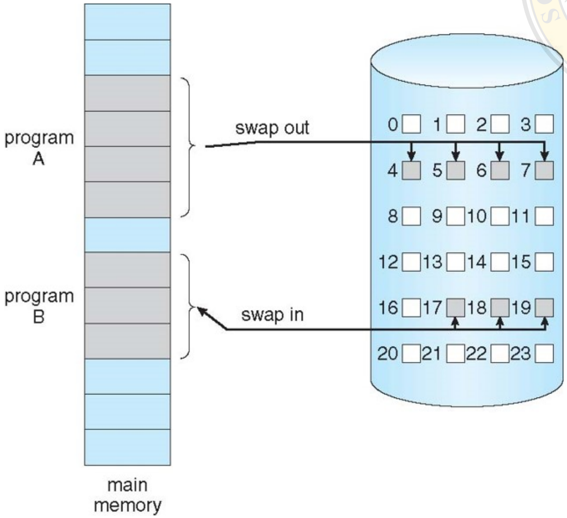


Contents

Demand Paging

- 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



Demand Paging

Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Pager brings in only those pages into memory
- 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
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging

Demand Paging

Valid-Invalid Bit

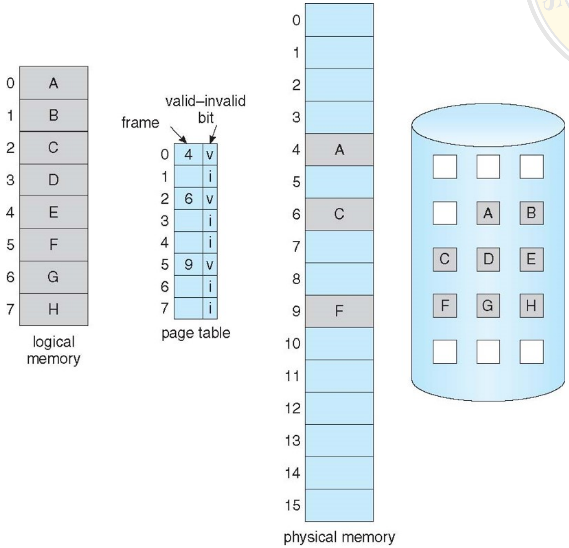
- With each page table entry a valid-invalid bit is associated (*v* ⇒ in-memory — memory resident, *i* ⇒ not-in-memory)
  - Present bit
- Initially valid-invalid bit is set to *i* on all entries
- Example of a page table snapshot:

Frame #	v/i
	v
	v
	i
	v
	i
	i
	i
. . .	

- During MMU address translation, if valid-invalid bit in page table entry is *i* ⇒ page fault

Demand Paging

Page Table when Some Pages are not in Main Memory



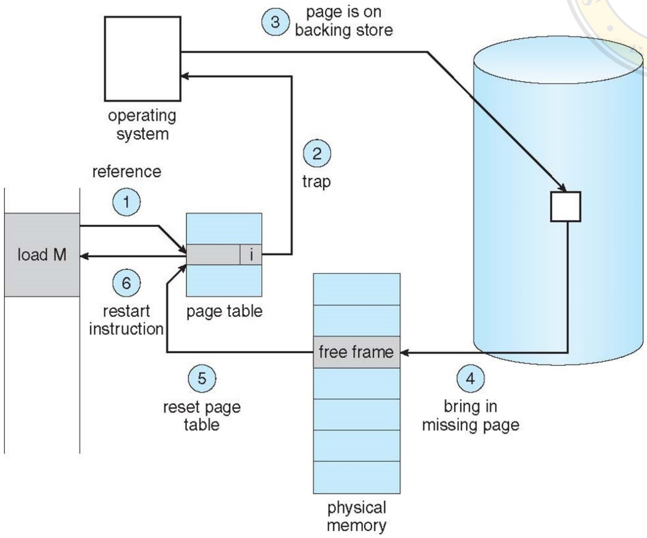
Demand Paging

Page Fault

- **Page fault** — If there is a reference to a page, first reference to that page will trap to operating system.
    1. Operating system looks at an internal table to decide:
        - » Invalid reference $\Rightarrow$ abort
        - » Just not in memory
    2. Find free frame
    3. Swap page into frame via scheduled disk operation
    4. Reset tables to indicate page now in memory
       Set validation bit $= v$
    5. Restart the instruction that caused the page fault

Demand Paging

Page Fault Handling



Demand Paging

Page-Fault Control Flow Algorithm (Hardware)

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)    // TLB hit
        if (CanAccess(TlbEntry.ProtectBits) == True)
                Offset = VirtualAddress & OFFSET_MASK
                PhysAddr = (TlbEntry.PFN<<SHIFT) | Offset
                AccessMemory(PhysAddr)
        else
                RaiseException(PROTECTION_FAULT)
else                    // TLB miss
        PTEAddr PTBR + (VPN *sizeof(PTE))
        PTE = AccessMemory(PTEAddr)
        if (PTE.Valid == False)
                RaiseException(SEGMENTATION_FAULT)
        else if (CanAccess(PTE.ProtectBits) == False)
                RaiseException(PROTECTION_FAULT)
        else if (PTE.Present == True)
                TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
                RetryInstruction()
        else if (PTE.Present == Faulse)
                RaiseException(PAGE_FAULT)
```

Demand Paging

Page-Fault Control Flow Algorithm (Software)

```
PFN = FindFreePhysicalPage()
if (PFN == -1)          // no free page found
        PFN = EvictPage()       // run replacement algorithm
DiskRead(PTE.DiskAddr, pfn)             // sleep (waiting for I/O)
PTE.present = True              // update page table with present
PTE.PFN = PFN           // bit and translation (PFN)
RetryInstruction()              // retry instruction
```

Demand Paging

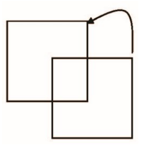
Aspects of Demand Paging

- 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

Demand Paging

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?

Demand Paging

Performance of Demand Paging

- Stages in Demand Paging
  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:
     - 5.1 Wait in a queue for this device until the read request is serviced
     - 5.2 Wait for the device seek and/or latency time
     - 5.3 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

Demand Paging

Performance of Demand Paging (cont.)

- Besides the context switch
- 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)

$$\begin{aligned} EAT = & (1 - p) \times T_{memory\_access} + \\ & p \times (T_{page\_fault\_overhead} + T_{swap\_page\_out} + T_{swap\_page\_in}) \end{aligned}$$

Demand Paging

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

$$\begin{aligned} EAT &= (1-p) \times 200 + p \times (8\ milliseconds) \\ &= (1-p) \times 200 + p \times 8,000,000 \\ &= 200 + p \times 7,999,800 \end{aligned}$$

- 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

$$220 > 200 + 7,999,800 \times p \Rightarrow p < 0.0000025$$

Demand Paging

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)
    - 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

Demand Paging

Inverted Page Tables

- Inverted page table no longer contains complete information about the logical address space of a process.
- That information is required if a referenced page is not currently in memory.
- Demand paging requires this information to process page faults.
- For the information to be available, an external page table (one per process) must be kept (can be on the backing store).
- A page fault may now cause the virtual memory manager to generate another page fault as it pages in the external page table it needs to locate the virtual page on the backing store.

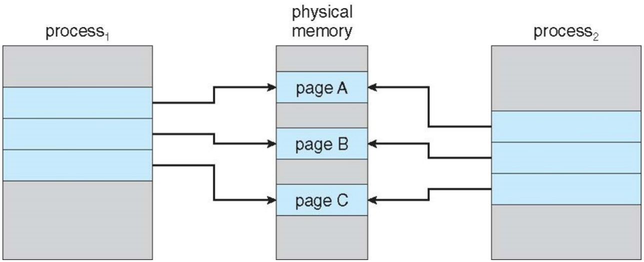
Contents

Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially share the same pages in memory
  - A shared page is copied only when either process modifies it
- 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
- *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

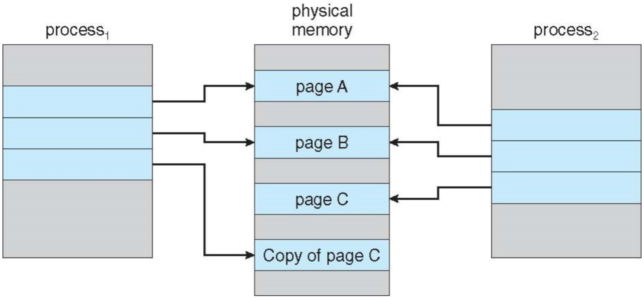
Copy-on-Write

Before Process$_1$ Modifies Page C



Copy-on-Write

After Process$_1$ Modifies Page C



Contents

Page Replacement

What Happens if There is No Free Frame?

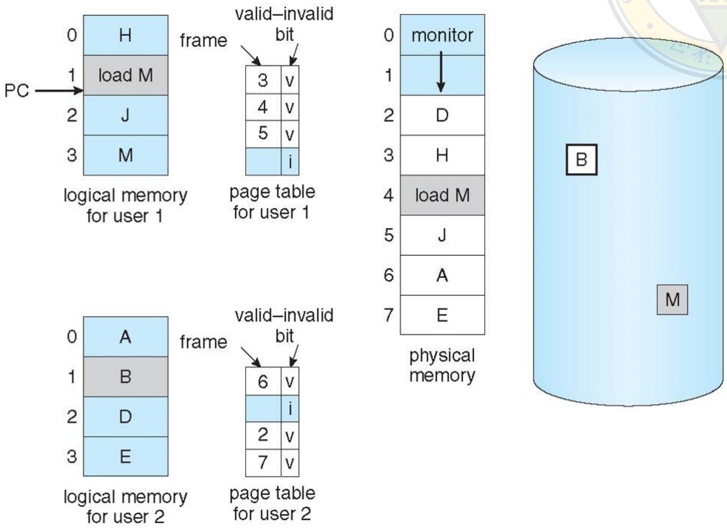
- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement — find some page in memory, but not really in use, page it out
  - Algorithm — terminate? swap out? replace the page?
  - Performance — want an algorithm which will result 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

Page Replacement

Need For Page Replacement



Page Replacement

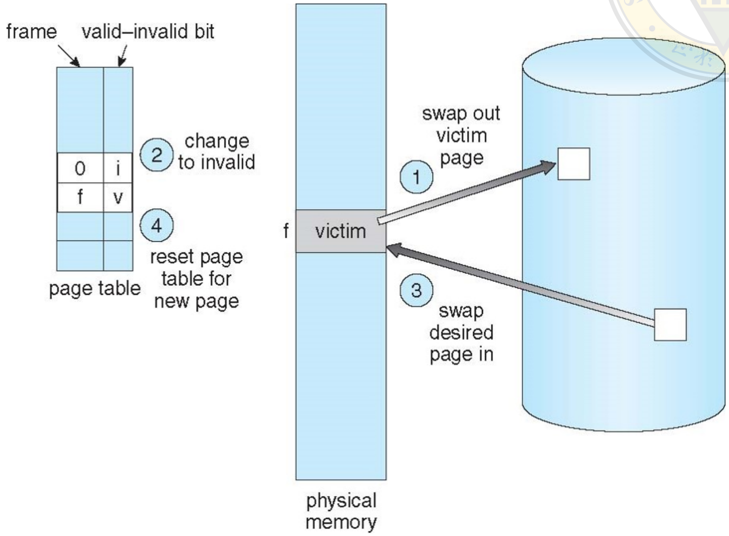
Basic Page Replacement

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
   - 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

Page Replacement



Page Replacement

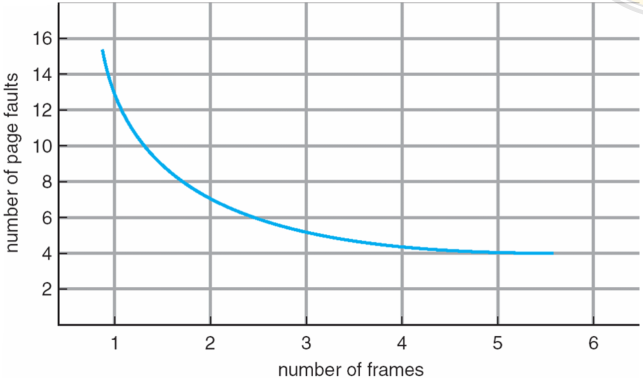
Page and Frame Replacement Algorithms

- **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

Page Replacement

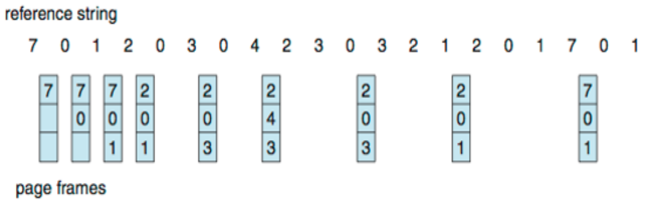
Graph of Page Faults Versus the Number of Frames



Page Replacement

Optimal Algorithm

- Bélády's MIN algorithm in 1966
- Replace page that will not be used for longest period of time
  - 3 frames, 9 page faults

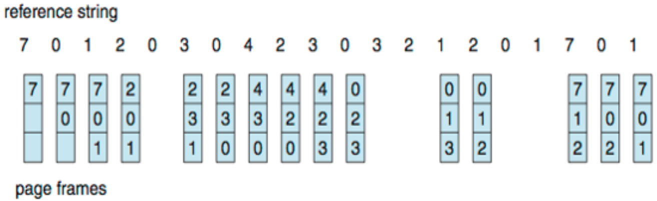


- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

Page Replacement

First-In-First-Out (FIFO) Algorithm

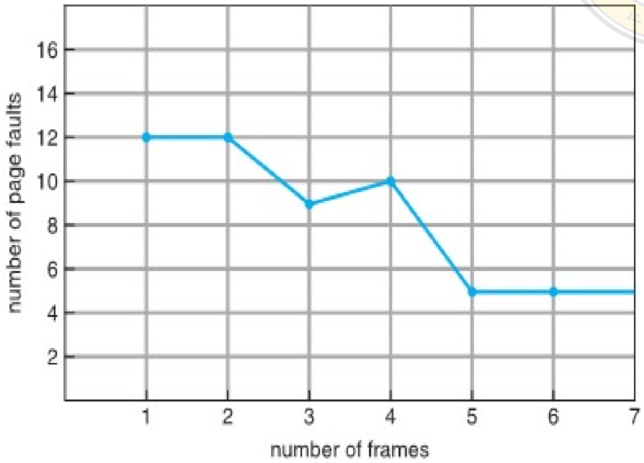
- Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
- 3 frames (3 pages can be in memory at a time per process)



  - 15 page faults
- How to track ages of pages?
  - Just use a FIFO queue
- 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! — **Bélády’s Anomaly**

Page Replacement

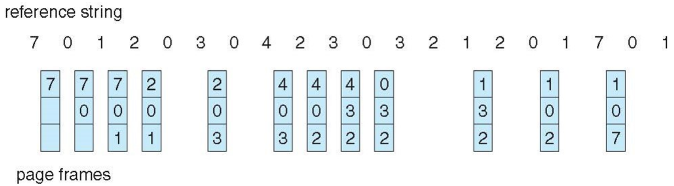
Bélády's Anomaly



Page Replacement

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in 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?

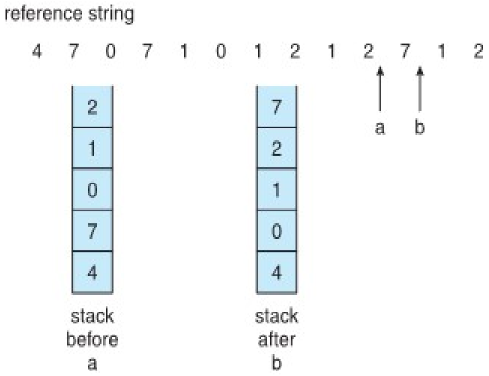
Page Replacement

LRU Algorithm (contd.)

- 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 (why?)
  - But each update more expensive
  - No search for replacement (why?)
    - LRU page is always at the bottom

Page Replacement

Use of a Stack to Record Most Recent Page References



Page Replacement

LRU Algorithm (contd.)

- LRU and OPT are cases of stack algorithms that do **NOT** have Bélády’s Anomaly
- Proof?
  - A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n + 1 frames. For LRU replacement, the set of pages in memory would be the n most recently referenced pages. If the number of frames is increased, these n pages will still be the most recently referenced and so will still be in memory.

Page Replacement

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
- **Additional-Reference-Bits Algorithm**
  - Keep an 8-bit byte for each page
  - At regular intervals shifts the bits right 1 bit, shift the reference bit into the high-order bit
  - Interpret these 8-bit bytes as unsigned integers, the page with lowest number is the LRU page

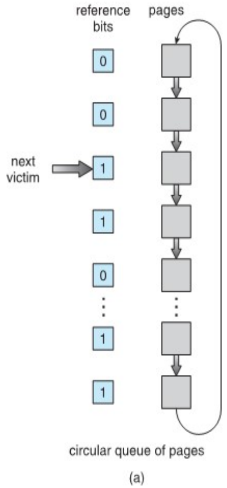
Page Replacement

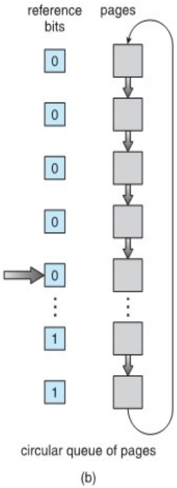
LRU Approximation Algorithms (contd.)

- **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

Page Replacement

Second-Chance (Clock) Page Replacement Algorithm





Page Replacement

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) 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 out before replacement
     - How can a page be modified without used?
  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 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

Page Replacement

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 smallest count
- 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

Page Replacement

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

Page Replacement

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge — e.g., 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 give direct access to the disk, getting out of the way of the applications
  - Raw disk mode
- Bypasses buffering, locking, etc

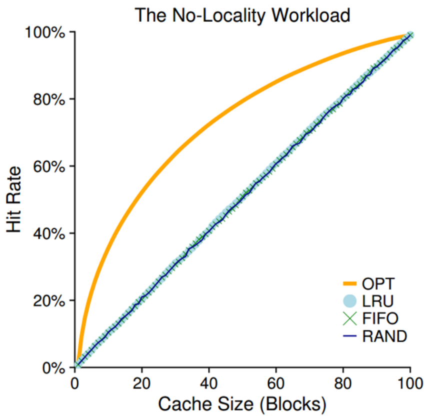
Page Replacement

An Example Benchmark

- Random Algorithm
  - Simply picks a random page to replace.
  - How Random does depends on the luck of the draw
- Workload examples
  - The no locality workload
    - each reference is to a random page within the set of accessed pages.
  - The "80-20" locality workload
    - 80% of the references are made to 20% of the pages (the "hot" pages); the remaining 20% of the references are made to the remaining 80% of the pages (the "cold" pages).
  - The Looping-Sequential Workload
    - Loop for accesses to a sequence of pages.

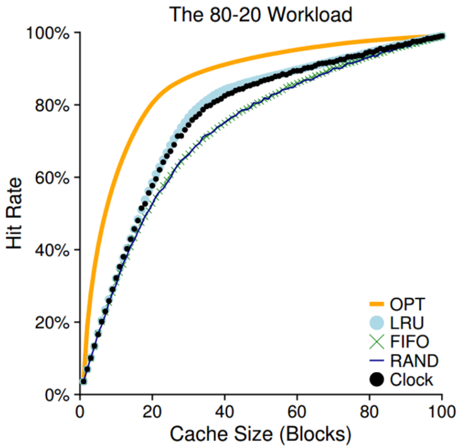
Page Replacement

Performance of Page Replacement Algorithms



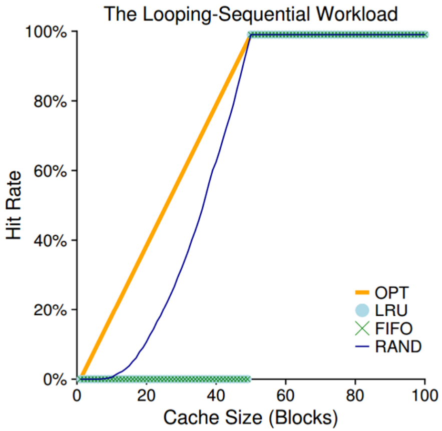
Page Replacement

Performance of Page Replacement Algorithms (contd.)



Page Replacement

Performance of Page Replacement Algorithms (contd.)



Page Replacement

In Class Exercise

Consider the reference page sequence is 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, and the number of page frame is 3.

(a) How many page faults for FIFO algorithm?

(b) How many page faults for LRU algorithm?

(c) How many page faults for OPT algorithm?

Page Replacement

Key

Consider the reference page sequence is 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, and the number of page frame is 3.

(a) How many page faults for FIFO algorithm? Key: 9

	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	4	4	4	5	5	5	5	5	5
		2	2	2	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	2	4	4
PF	○	○	○	○	○	○	○			○	○	

(b) How many page faults for LRU algorithm? Key: 10

	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	4	4	4	5	5	5	3	3	3
		2	2	2	1	1	1	1	1	1	4	4
			3	3	3	2	2	2	2	2	2	5
PF	○	○	○	○	○	○	○			○	○	○

(c) How many page faults for OPT algorithm? Key: 7

	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	1	1	1	1	1	1	3	4	4
		2	2	2	2	2	2	2	2	2	2	2
			3	4	4	4	5	5	5	5	5	5
PF	○	○	○	○			○			○	○	

Contents

Allocation of Frames

- Each process needs minimum number of frames
- Examples:
  - IBM 370 — 6 pages to handle *MVC* instruction:
    - instruction is 6 bytes, might span 2 pages
    - 2 pages to handle from
    - 2 pages to handle to
  - The *MVC* instruction may be the operand of an *EXECUTE* instruction
  - One (or more but limited) level indirect addressing
- Maximum of course is total frames in the system
- Two major allocation schemes
  - **Fixed allocation**
  - **Priority allocation**
- Many variations

Allocation of Frames

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

$$m = \textit{total number of frames} \qquad m = 64$$

$$s_i = \textit{size of process } p_i \qquad s_1 = 10$$

$$s_2 = 127$$

$$a_i = \textit{allocation for } p_i = (s_i / \Sigma s_i) \times m \qquad a_1 = 10/137 \times 62 \approx 4$$

$$a_2 = 127/137 \times 62 \approx 57$$

Allocation of Frames

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, OR
  - select for replacement a frame from a process with lower priority number

Allocation of Frames

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 process cannot control its own page-fault rate
  - But greater throughput so more common
- **Local replacement** — each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Allocation of Frames

Non-Uniform Memory Access (NUMA)

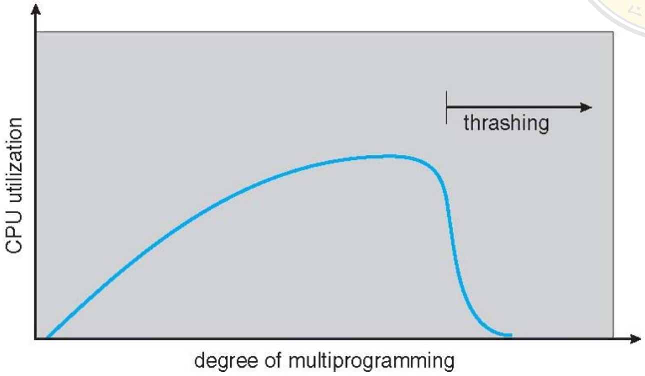
- So far all memory accessed equally
- Many systems are NUMA — speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - 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

Contents

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** — a process is busy swapping pages in and out

Thrashing



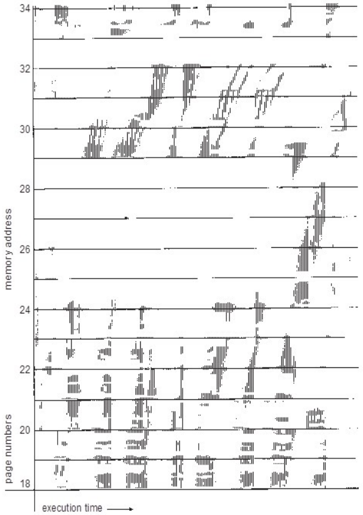
Thrashing

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
  - Limit effects by using local or priority page replacement

Thrashing

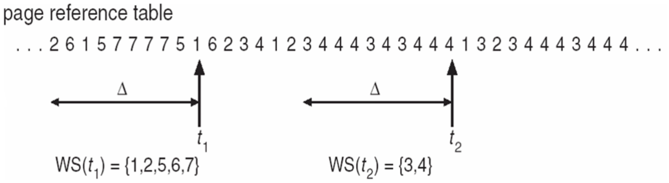
Locality in a Memory-Reference Pattern



Thrashing

Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
- WSS_i (working set size 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
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes



Thrashing

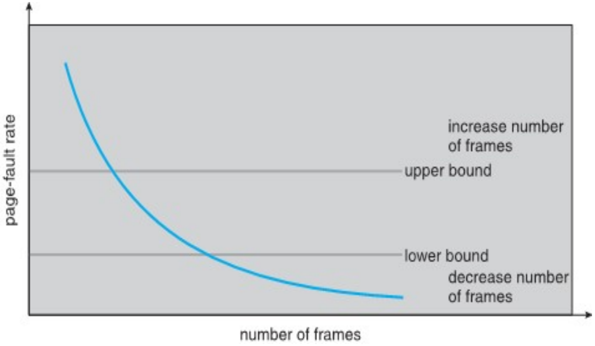
Keeping Track of the Working Set

- Approximate with interval timer interrupt + 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
  - cost to service the interrupts is higher

Thrashing

Page-Fault Frequency

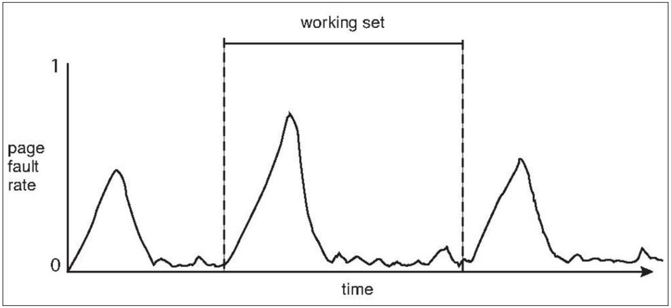
- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (PFF) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Thrashing

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



Contents

Memory-Mapped Files

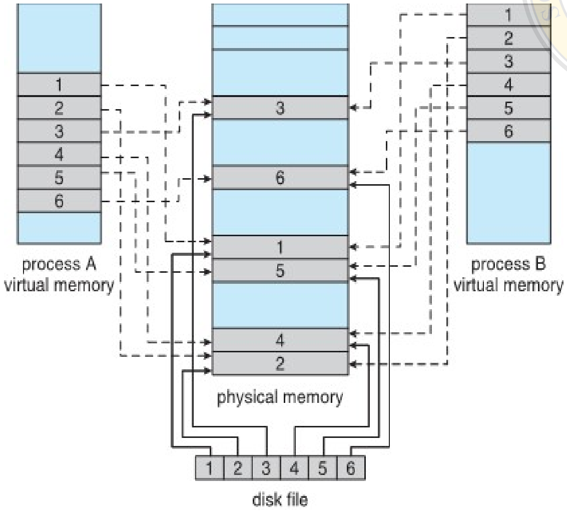
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file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

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*() system 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
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)

Memory-Mapped Files



Contents

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

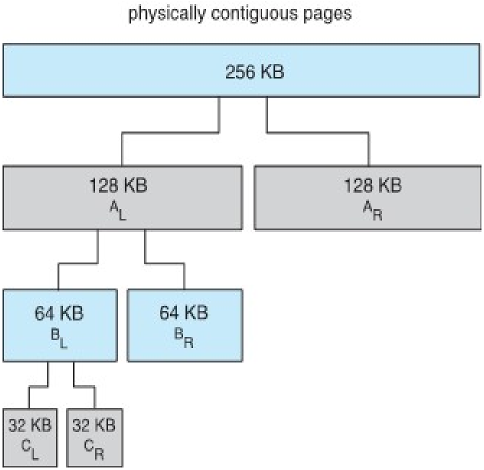
Allocating Kernel Memory

Buddy Allocator

- 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

Allocating Kernel Memory

Buddy Allocator (contd.)



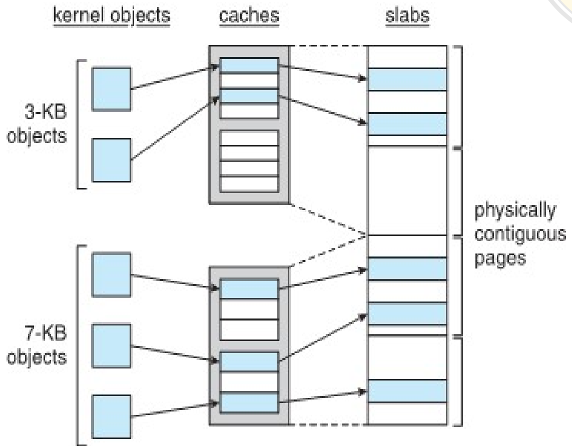
Allocating Kernel Memory

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

Allocating Kernel Memory

Slab Allocator (contd.)



Contents

Other Considerations

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 $s \times \alpha$ save pages faults > or < than the cost of prepaging $s \times (1 - \alpha)$ unnecessary pages?
  - α near zero $\Rightarrow$ prepaging loses

Other Considerations

Program Structure

```
int[128,128] data;
```

- Each row is stored in one page
- Program #1: $128 \times 128 = 16,384$ page faults

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

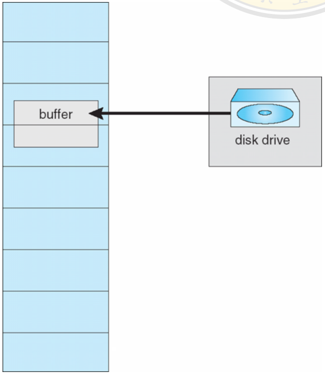
- Program #2: 128 page faults

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

Other Considerations

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


buffer
disk drive