CAS CS 552
Intro to Operating Systems

Richard West

Virtual Memory

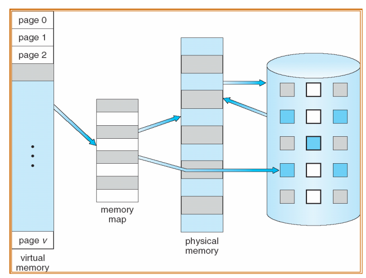
Objectives

- 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

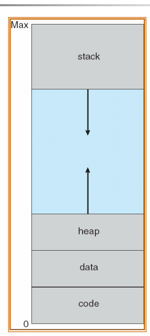
Background

- **Virtual memory** – separation of user's (logical) view of memory from physical memory
  - Only part of a program needs to be in physical/main memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by multiple processes
  - Allows for more efficient process creation
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

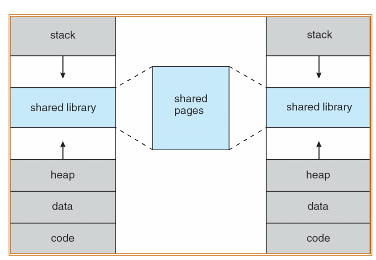
Virtual Memory Pages Mapped to Physical (Page) Frames



Virtual-address Space



Shared Library Using Virtual Memory



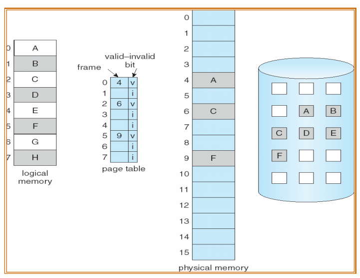
Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - Higher degree of multiprogramming
- Page is needed ⇒ a reference is made to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapping** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

Page Validation

- Each page table entry has a valid/invalid bit
  - Used to indicate whether or not a page is in memory
  - Let **v** ⇒ in-memory, **i** ⇒ not-in-memory
  - NOTE: Some architectures refer to the valid bit as a "present" bit
  - Initially valid–invalid bit is set to **i** on all entries

Example Page Table w/ Some Unmapped Pages



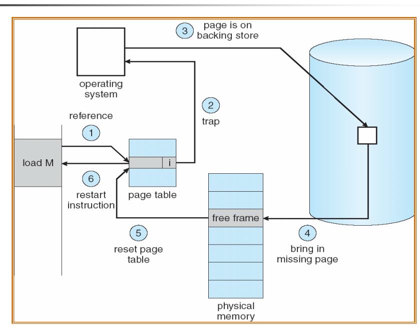
Page Faults

- If a process executes and accesses all pages in memory (i.e., all memory references are to valid pages), execution proceeds normally
- If a process tries to access a page, **P**, not in memory:
  - Paging hardware will notice invalid bit for **P** when translating an address via the active page table
    - This causes a **page fault** trap to the OS

Handling Page Faults

- OS checks info in the process control block (PCB) for the current process causing the page fault, to determine whether or not the memory reference was within the process's address space
  - PCB has info about the contiguous virtual memory areas making up the process's address space
  - If reference was to an address outside process's address space, the process is terminated (a memory protection fault)
  - If reference was within process's address space but page is *not* in memory, page is mapped to a free frame
  - The page table is updated with a valid entry for the page that faulted
    - Info about the contiguous VM areas of the process are updated as necessary
  - The instruction that caused the page fault is restarted as though it had always been in memory

Steps in Handling a Page Fault



Performance of Demand Paging

- Let p = Pr{page fault} | $0 \le p \le 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

  EAT = $(1 - p)$ x memory access
  + p (page fault overhead incl.
  time to swap page out,
  time to swap page in, &
  time to restart instruction)

Demand Paging Example

- Memory access time = 200 nanoseconds (nS)
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p) x 200 + p x 8000000 nS
  = 200 + p x 7999800 nS
- If one access out of 1000 causes a page fault, then
  EAT = 8.2 microseconds
  This is a slowdown by a factor of about 40
  Need to keep page fault rate low

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
- To do this, pages have a read-only bit set that causes a trap to the kernel when either parent or child attempts to write to the corresponding page
- COW allows more efficient process creation as only modified pages are copied

Page Replacement 1/2

- The story so far: page-based memory…
  - eliminates external fragmentation
  - enables virtual memory to be larger than physical memory
  - provides memory protection for different address spaces (at granularity of a page)
- To maintain efficient use of CPU, it is desirable to have a *high degree of multiprogramming*
- Increasing degree of multiprogramming increases the # of processes at least partially loaded in memory
  - Nearly all physical memory may be in use
  - If a process requires a page to be loaded into memory (e.g., due to demand paging), there might *not* be any free frames
  - **In this case, on a page fault, the OS must replace an existing page in memory**

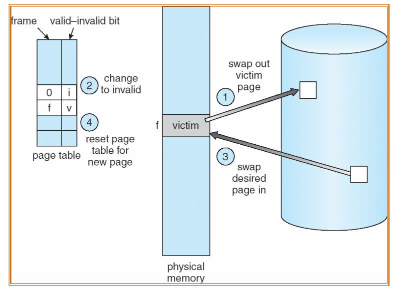
Page Replacement 2/2

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use **modified (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk to free up space for new pages

Basic Page Replacement Approach

- 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
- Bring the desired page into one of the free frames
- Update the page and frame tables accordingly
- Restart the process

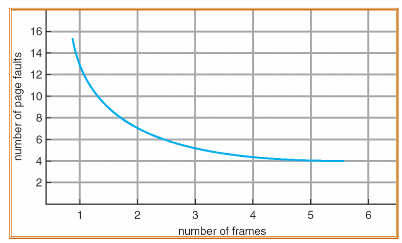
Diagrammatic View of Page Replacement



Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references and computing the number of page faults on that string
  - For simplicity, a reference string will refer to a series of page numbers corresponding to the virtual memory addresses being referenced

Typical Page-Fault Distribution



First-In-First-Out (FIFO) Algorithm

- OS maintains a queue (for all currently loaded) pages based on the time they are brought into memory
- Oldest page is at the head of the queue & newest is at the back
  - When a page is replaced, it is taken from the front of the queue
- FIFO replacement can suffer from Belady's Anomaly
  - more frames ⇒ more page faults

Example FIFO Page Replacement

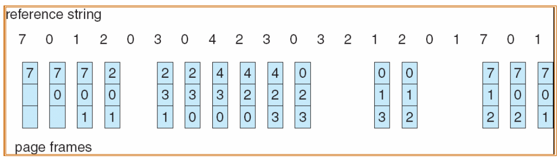
- Reference string:
  - **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
  - 3 frames:

1	4	5	
2	1	3	9 page faults
3	2	4	

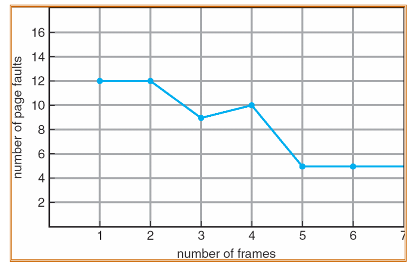
  - 4 frames:

1	5	4	
2	1	5	10 page faults
3	2		
4	3		

Additional FIFO Page Replacement Example



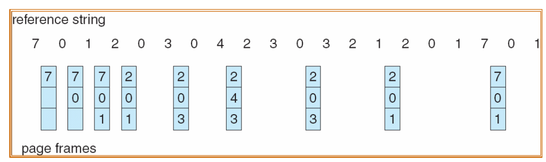
FIFO Replacement Illustrating Belady's Anomaly



Optimal Algorithm

- Achieves lowest page fault rate
- Aim: replace the page that will not be used for the longest period of time (i.e., whose next reference is furthest in the *future*)
- Problem: Optimal algorithm requires future knowledge of memory, and hence, page references which is not usually possible to know
- If we use the recent past to predict the future, then we can replace the page that has not been used for the longest time
  - Least recently used (LRU) replacement

Optimal Page Replacement



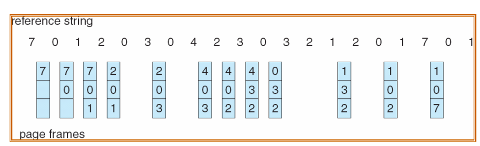
LRU Page Replacement Example 1

- Let there be 4 frames to use
- 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

LRU Page Replacement Example 2

- Here, there are 3 frames to use



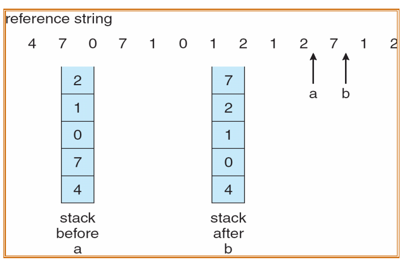
LRU Implementation 1/2

- (1) Counter implementation:
  - A logical clock (or counter) is incremented every memory reference
  - Each page has a "time of use" field
  - The "time of use" field is updated with the counter value when the page is referenced
  - The page to replace is the one with the lowest counter value
  - Problems:
    - counter overflow
    - must search the counter values for every page to determine lowest counter value

LRU Implementation 2/2

- (2) Stack implementation:
- keep a stack of page numbers for all loaded pages
  - Most recently referenced page is removed from the stack and placed on top of the same stack
  - Over time, the page at the bottom of the stack is the LRU page
- Problems: rearranging the stack can require more overhead than the counter to keep track of page references
- Both stack- and counter-based LRU implementations require h/w assistance in practice, since memory references can be in the multiple millions per second

Example Stack Implementation



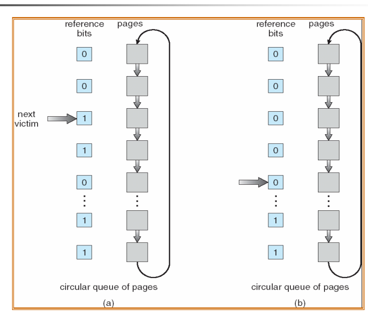
LRU Approximation

- With limited hardware support many computer systems implement alternatives, or approximations, to LRU
  - Some systems use a **reference bit** (for each page) set by the hardware when the page is referenced
  - Can replace a page whose reference bit is not set (if one exists)
    - ...but this loses historical ordering info about page references

Second Chance (Clock) Algorithm

- Like FIFO replacement but first check the reference bit of a page
  - If bit=0, replace page
  - If bit=1, clear bit and check next page
    - This gives page a "second chance" to stay in memory
  - Worst-case: all bits are 1 for all pages in memory
    - Here, we cycle through all pages, clearing their reference bits, and return to the first page in the cycle (which is replaced)

Second-Chance (Clock) Page Replacement Algorithm



Enhanced Second Chance Algorithm

- a.k.a "not recently used", or NRU
- This uses a **reference bit** and a **modify bit** per page
- Each page can be in one of four classes:

referenced	modified
0	0
0	1
1	0
1	1

Most desirable to replace

Not recently ref'd or modified

Recently ref'd but not recently modified

A page may be modified but not recently ref'd if it is awaiting write-back to disk

Counter Algorithms

- Using multiple reference bits (e.g., 8) for each page in memory, we can approximate a finite history of references
  - At regular intervals (e.g., 100mS) a timer interrupt causes the OS to right-shift by 1-bit an n-bit counter for each page, and then insert the reference bit of that page into the most significant bit of the counter
    - Least significant bits are discarded
    - e.g. 1, if counter = 00000000 page has not been ref'd for last eight clock intervals
    - e.g. 2, a page with counter=10001100 has been more recently ref'd than one page with counter 01111111

Counter Algorithm Variants

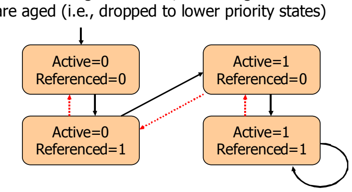
- **LFU** – replace page with smallest counter value
  - Problem?
    - A page might be referenced frequently and then not at all but may remain in memory long after it is needed
- **MFU** – replace page w/ largest counter value as smallest one may have only just been brought into memory (and may need to be used again in the future)

Global vs. Local Replacement

- **Global replacement**
  - System may replace a page from any process/address space
- **Local replacement**
  - System replaces a page associated with the local process/address space

Example - Linux

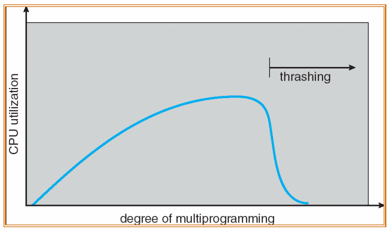
- Paging policy is a variant of the clock algorithm
- Linux maintains **active** and **inactive** lists for all pages in memory
- Pages move through the following states depending on the rate of them being referenced, while at given time intervals pages are aged (i.e., dropped to lower priority states)



Thrashing 1/2

- Suppose a process address space contains two pages, P1 and P2
- Let P2 be brought into memory at the cost of P1 being replaced
  - What happens if the process requires P1 shortly after it is swapped out?
    - Will end up with another page fault
  - If there is a lot of paging activity, a process may spend more time involved in paging than executing
    - Such a process is said to be **thrashing**

Thrashing 2/2



Ways to Limit Thrashing

- Decrease the degree of multiprogramming
- Use only local page replacement algorithms
  - i.e., replace pages from the local process and not other processes
- Provide a process with as many frames as it needs
  - How do we do this?

Locality of Reference

- "90:10" Rule
  - A program spends 90% of its time executing 10% of its code
  - Loops, subroutines, procedures/functions, basic code blocks define "localities of reference"
- If system allocates enough frames to a process for its current locality, page faults will be reduced until the process changes locality

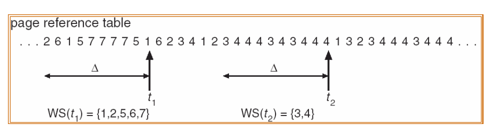
Working Set Model 1/2

- The set of pages in the most recent Δ page references is the **working set**
- The working set is an approximation of the program's locality
  - It is the set of pages currently being used by a process and the entire working set should be in memory to reduce page faults
- Δ should be large enough to encompass a process's current locality, but no larger

Working Set Model 2/2

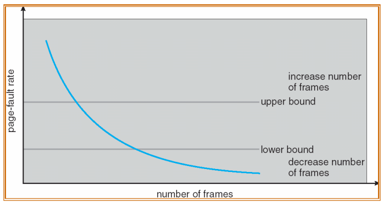
- Let $\alpha_i(t)$ be the working set size of process i at time, t
- Total demand for page frames from n processes is:
  - $D=\sum_{\forall i} \alpha_i(t)$ | 1<=i<=n
- If D > total frames in system, thrashing will occur
- Solution:
  - OS monitors working set for each process and allocates enough frames for the working set corresponding to the current locality
  - If there are spare frames, a new process can begin (thereby increasing the degree of multiprogramming)
  - If D exceeds available frames, OS suspends one or more processes, by writing pages of suspended processes to disk

Working-set model



Page-Fault Frequency Scheme

- To reduce thrashing the OS may take the following approach:
  - Upper and lower bounds can be placed on the desired page-fault rate of a process
  - If a process's page fault rate exceeds the upper bound, it is allocated another frame
  - If a process's page fault rate drops below the lower bound, it loses a frame



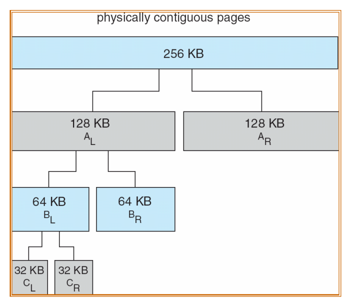
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
    - e.g., for file objects and process control blocks
  - Some kernel memory needs to be contiguous
    - e.g., for DMA transfers

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

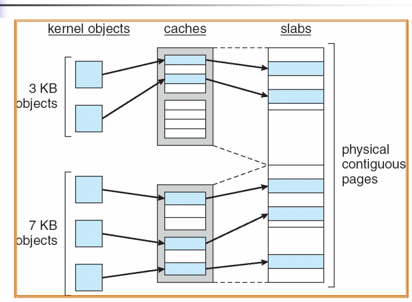
Buddy System Allocator



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 (fraction) of the pages are used
    - Is cost of $s * \alpha$ saved pages faults > or < than the cost of prepaging $s * (1- \alpha)$ unnecessary pages?
    - α near zero $\Rightarrow$ prepaging less beneficial

Other Issues – Page Size

- Page size selection must take into consideration:
  - fragmentation
  - page 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 – 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