Chapter 9: Virtual Memory Management

Contents

- Background
- Demand Paging
- Copy-On-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Allocation Kernel Memory

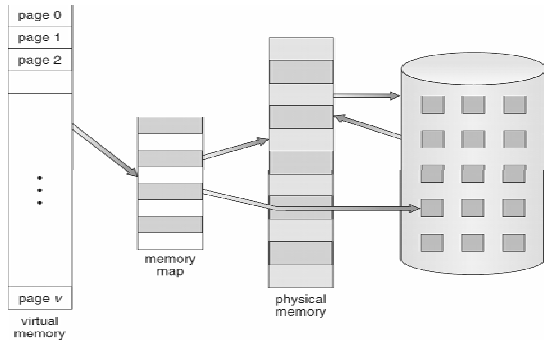
Motivation

- Every time a program is run all parts of a process are not needed at all times during the execution
  - many routines/functions are rarely used
  - programmers make data structures too large
    - e.g. char line[500];
- Unnecessary parts can be kept on secondary storage (hard disk)
  - they need to be brought into main memory when needed
- Advantages of being able to run a program that is only partially loaded into memory:
  - programs are no longer limited by physical memory size
  - more programs can be loaded and run
    - increased CPU utilization, throughput
  - the OS can pretend to have a larger *virtual address space*
- Must be managed carefully to avoid substantial decrease in performance

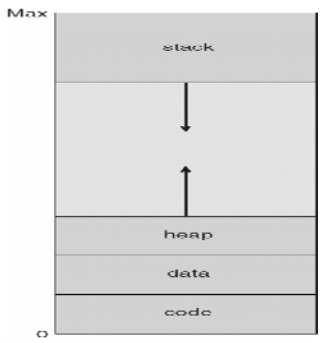
Background

- *Virtual memory*, is a technique that allows the execution of processes that may not be completely in memory (RAM)
- Virtual memory separates the user's logical view of memory from the physical memory.
- Virtual memory appears to be much larger, and more uniform, than physical memory
  - requires the hardware to support swapping in/out from secondary storage

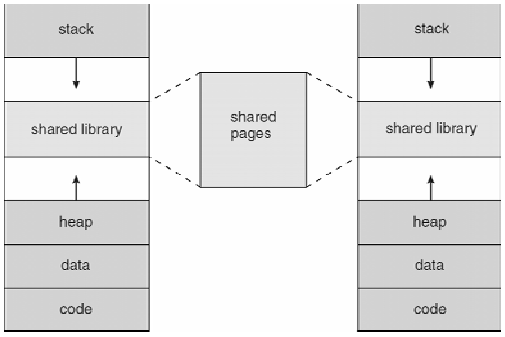
Virtual Memory That is Larger Than Physical Memory



Virtual-address Space



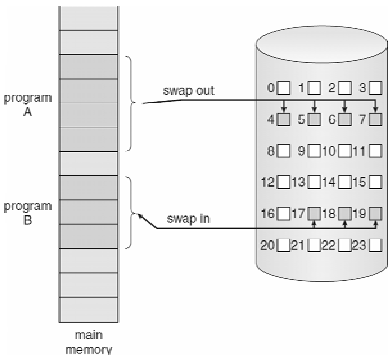
Shared Library Using Virtual Memory



Demand Paging

- Virtual memory is commonly implemented by demand paging
  - similar to a page system with swapping
- Load a page into memory only when it is needed, called demand paging
  - Pages that are never used are never loaded
  - Less I/O needed for load and swap processes (only parts of the process need to be transferred)
  - Less memory needed
- Similar to page table system with swapping, but demand paging doesn't swap entire process into memory, instead uses a lazy swapper
- Lazy swapper – never swaps a page into memory unless it will be needed
  - A swapper that deals with pages is called a pager

Transfer of a Paged Memory to Contiguous Disk Space



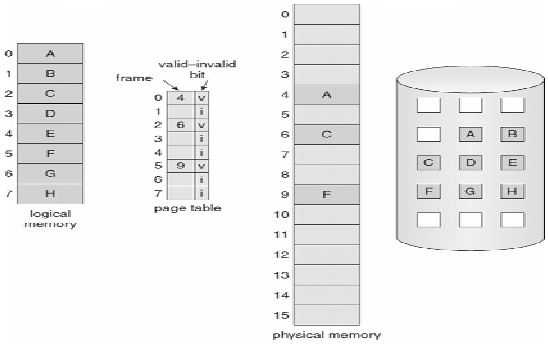
Implementation

- How to distinguish between the pages that are in memory and the pages that are on the disk?
- One approach is to add a *valid-invalid bit* to each page in the page table
  - Valid means that the process is allowed to access that page, AND it is in physical memory
  - Invalid means that the process may not be allowed to access the requested page, or the page may not yet be in physical memory
- Initially valid–invalid bit is set to i on all entries

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

page table

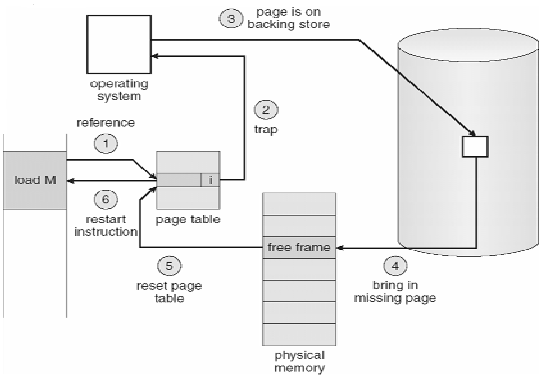
Page Table When Some Pages Are Not in Main Memory



Page fault

- If requested page is memory resident, then process can operate normally
- If requested page not in physical memory (i.e. page is invalid in page table), then a page fault occurs
- If page fault occurs, first check to see if the requested page was an illegal request or if it just isn't in physical memory
  - If an illegal reference, the process terminates (seg fault, bus error, ...)
  - If simply not in memory:
    - Get a free frame of physical memory from the free-list
    - Swap page from disk into the frame via a scheduled disk operation
    - Update tables to indicate that the page is now in physical memory (i.e. set valid-invalid bit to 'v')
    - Restart the instruction that caused the page fault
- When the page fault occurs we save state (registers, condition code, instruction counter) of interrupted process, we must restart the process in exactly the same place and state, except that desired page is now in memory and is accessible

Steps in Handling a Page Fault



Aspects of Demand paging

- Extreme case of demand paging – start a process with no pages in memory
  - Never bring a page into physical memory until it is needed
  - A page fault will occur each time a new page is requested, including on the very first page
  - The scheme is called pure demand paging
- Demand paging (and pure demand paging) require hardware support
  - Page table with valid / invalid bit
  - Secondary memory (swap device with swap space)
  - Ability to restart an instruction

Demand Page Performance

- Let p = probability of a page fault
  ma = memory access time (typically 10 - 200 nsecs; *use 100 nsecs*)
- *Effective Access Time* (EAT)
  = ((1-p) * ma) + (p * page-fault-rate)
- Main components:
  (1) service the page fault interrupt
  (2) read the page into memory
  (3) restart the process
- (1) and (3) take ~1-100 microsecs each
- (2) Reading in the page ~=
  - hard disk latency (~8 msecs) +
    seek time (~15 msecs) +
    transfer time (~1 msec)
    ≈ 24 msecs
- Page fault time = (1) + (2) + (3)
  = about 25 msecs

Demand Page Performance

- Effective Access Time (in nsecs)
  = ((1-p) * 100) + p * 25,000,000

  = 100 + p*24,999,900 nsecs
- Assume 1 access out of 1000 causes a page fault.
- EAT = 100 + 0.001 * 24,999,900
  = about 25,000 nsecs
- Slowdown is about *250 times*!
- What is the page fault rate to make the slowdown less than 10%?
- 110 > 100 + p*24,999,900
  so 10 > p*24,999,900
  so p < ~ 1/2,500,000
- A page fault must occur less than once in 2,500,000 memory accesses.

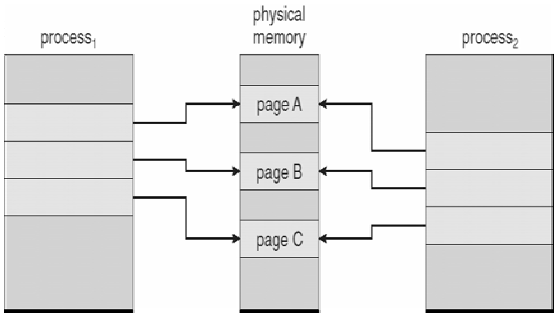
Observations

- The page fault rate must be very low in a demand paging system or the effective access time increases *dramatically*.
- Transfer time is a small part of the total time for reading in a page.

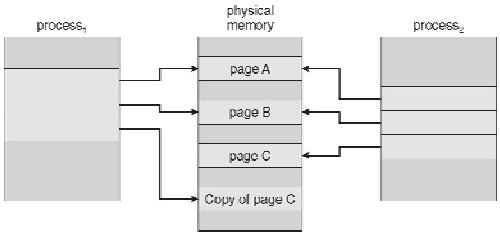
Copy-on-Write

- During process creation Copy-On-Write (COW) allows both parent and child processes to initially share the same pages in memory
- These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of shared page is created
- COW allows more efficient process creation as only modified pages are copied
- Using COW it is important to note the location from which the free page will be allocated
  - Free pages are typically allocated from a pool of zero-fill-on-demand pages (demand pages zeroed-out before being allocated)

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



Virtual memory fork- vfork() system call

- A variation of fork() exists called vfork()- that operates differently from fork() with copy-on-write (Rather than copying the whole image when fork() is executed, copy-on-write techniques are used).
- fork() generates two identical processes with separate memory.
- vfork() generates two processes that share the same memory.
- vfork ()- create a child process and block parent. So child can use parent's address space. Because vfork() does NOT use copy-on-write, if child modifies parent's address space, those changes will be visible to parent
- The intent of vfork() was to eliminate the overhead of copying the whole process image if you only want to do an exec() in the child. Because exec() replaces the whole image of the child process, there is no point in copying the image of the parent.
- vfork() – Does not use COW. Changes made by child visible to parent

Note: vfork() must used with caution to ensure that the child process does not modify the address space of the parent.

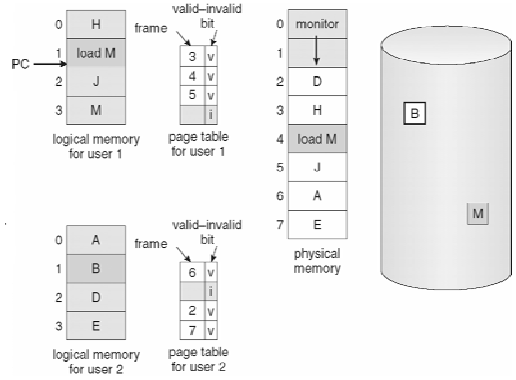
Over-allocation of memory

- Assume an OS with 4 frames.
- If a process p_0 of 4 pages in size actually uses only 2 pages
- Knowing this, the OS can allocate 2 pages to p_0 and give the other 2 pages to another process p_1
  - increases CPU utilization and throughput
- This is Called *over-allocation* of memory
- However it is possible, the process p_0 for a particular data set, may suddenly try to use all 4 of its pages
  - but no longer any free frames
- What to do if there is no free frame?

Possible Solutions

- Terminate the process
  - restart it later
- Swap out the process
  - swap it back when there are free frames
- Page replacement
  - find a page in memory that is not really in use and swap it out

Need For Page Replacement



Page and Frame Replacement Algorithms

- Frame-allocation algorithm determines
  - How many frames to give each process
- Page-replacement algorithm determines
  - Which frames to replace
  - For performance reasons want a page replacement algorithm which will result in minimum number of page faults

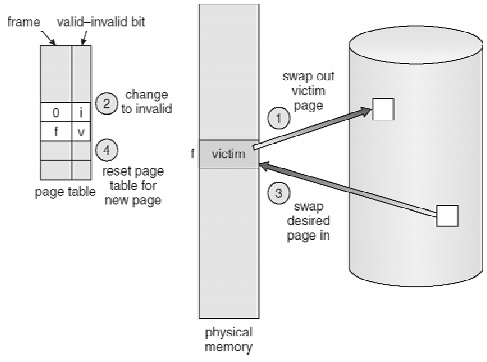
Basic Page Replacement

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

Reducing the Overhead

- If no frames are free two page transfers are required (one out and one in)
  - doubles the page fault rate
  - makes the EAT worse
- Use **modify (dirty) bit** to reduce overhead of page transfers
- Each page can have a modify bit (*dirty bit*)
  - indicates if the page has been modified
- Only modified pages are written back to disk from physical memory
- No need to swap out an unmodified victim page.

Page Replacement



Page Replacement Algorithms

- Many different page replacement algorithms exist
  - FIFO Page Replacement
  - Optimal Page Replacement (theoretical best case)
  - Least-Recently Used (LRU) Page Replacement
  - Counting-Based Page Replacement

FIFO Page Replacement

- The page that was brought in first will be replaced first
- **Pros:**
  - Very simple and easy to understand
- **Cons:**
  - Performance may not be very good
  - the oldest page may have been used very recently
  - there is a good chance that the replaced page will be needed again shortly
  - May result in a high number of page faults

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)

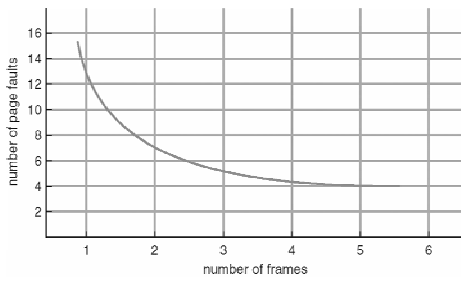
1	1	4	5		
2	2	1	3	9 page faults	
3	3	2	4		

- 4 frames

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

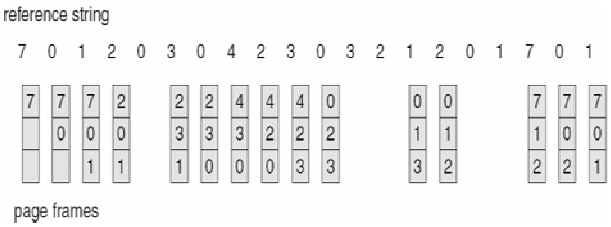
- Belady's Anomaly: more frames ⇒ more page faults
  - More memory does not lead to better performance

Graph of Page Faults Versus The Number of Frames



FIFO Page Replacement Example

A total of 15 page faults

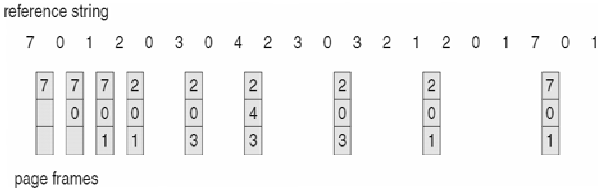


Optimal Page Replacement

- Replace the page that will not be used for longest period of time in the future
- Guarantees the lowest possible page-fault rate for a fixed number of frames
- Provably optimal algorithm
  - useful for comparison with other algorithms
- Impractical for real systems (very difficult to implement)
  - it is not possible to know which page won't be used for the longest period of time (can't see the future)

Optimal Page Replacement Example

A total of 9 page faults

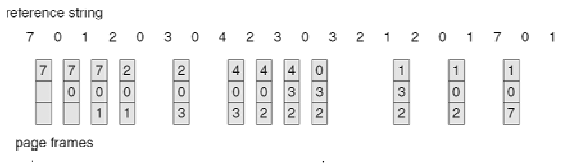


Least Recently Used (LRU) Algorithm

- Replace the page that is the least recently used page
- Similar to the optimal algorithm, but practical
- Requires additional information about the pages
  - Associate time of last use with each page

LRU Page Replacement Example

A total of 12 page faults better than FIFO, but not optimal



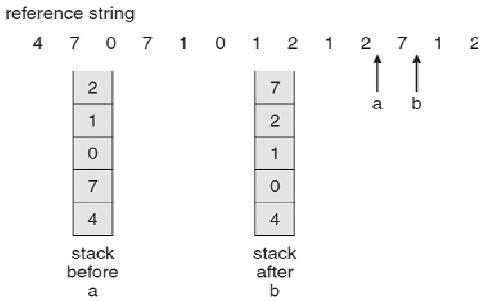
LRU and Belady's Anomaly

- Neither OPT nor LRU suffers from Belady's anomaly.
- In general, LRU gives good (low) page-fault rates.

LRU Page Replacement

- How should a least-recently-used algorithm be implemented?
  - requires special hardware support
- Option #1 - Counter implementation
  - Every page entry has a counter field; every time page is referenced through this entry, copy the system clock into the counter field
  - When a page needs to be replaced, look at the counters to determine which one to replace
    - The one with the smallest counter value will be replaced
  - Problem: have to search all pages/counters! (SLOW)
- Option #2 - Stack implementation
  - Keep a stack of page numbers in a double link form:
  - If a page is referenced, move it to the top of the stack (not a pure stack implementation)
  - Most recently used page is always at the top of the stack and the least recently used page is always at the bottom of the stack
  - Problem: too expensive
    - At worst case requires 6 pointer updates for each page reference
  - No search for replacement (replacement fast)

Use Of A Stack to Record The Most Recent Page References



LRU Approximation Algorithms (1)

- Most system will simply provide a Hardware support in the form of a **reference bit** in the page table for each page
- **Reference bit**
  - With each page in the page table associate a bit, initially = 0 (not referenced/used)
  - When page is referenced , bit set to 1.
  - Replace the one which is 0 (if one exists).
    - Although we do not know the order of use (several pages may have 0 value)

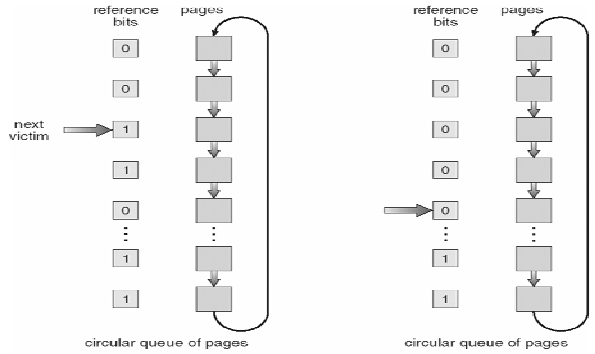
LRU Approximation Algorithms (2)

- **Additional reference byte (e.g., 8 bits) algorithm**
- Keep a ***reference byte*** for each page initialized to 00000000
- Every time a page is referenced
  - Shift the reference bits to the right by 1 bit and discarding low–order bit
  - Place the reference bit (1 if being referenced 0 otherwise) into the high order bit of the reference bits
  - The page with the lowest reference bits value is the one that is Least Recently Used, thus to be replaced
- Example 1 : the page with ref bits 11000100 is more recently used than the page with ref bits 01110111 (11000100 > 01110111)
- Example 2: 00000000 means this page not been used for 8 periods of time. So, it is LRU.
- Example 3: 11111111 means this page have been used (referenced) 8 times.
- If numbers are not unique, then
  - Replace (swap out) all pages with the smallest value (have same value), or use a FIFO selection among them

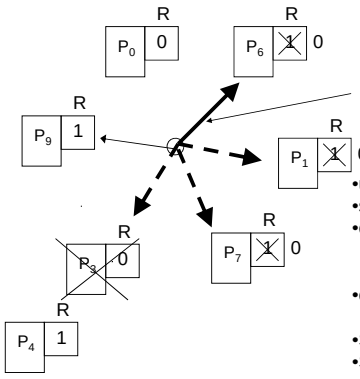
LRU Approximation Algorithms (3)

- **Second Chance algorithm (or clock algorithm)**
- The basic algorithm of second-chance is a FIFO.
- A reference bit for each frame is set to 1 whenever:
  - a page is first loaded into the frame.
  - the corresponding page is referenced.
- When a page has been selected for replacement, inspect its reference bit:
- If a page's reference bit is 1
  - set its reference bit to zero and skip it (give it a second chance)
- If a page's reference bit is set to 0
  - select this page for replacement
- Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chance)
- Problem of clock algorithm: does not differentiate modified (dirty) vs. clean pages
  - More expensive to replace dirty pages than clean pages

Clock Page-Replacement Algorithm

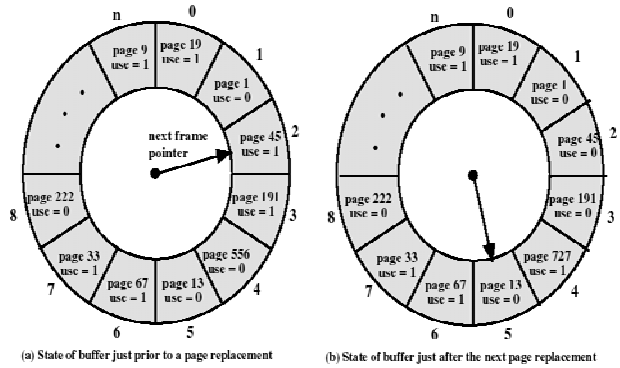


Clock Algorithm



- user refs P_4 - not currently paged
- start at P_6
- check P_6, P_1, P_7 and set their reference bits to zero (give them a second chance)
- check P_3 and notice its ref bit is 0
- Select P_3 for replacement
- Set pointer to P_9 for next search

The Clock Policy: Another Example



(a) State of buffer just prior to a page replacement

(b) State of buffer just after the next page replacement

Enhanced Second-Chance Algorithm (2-bits).

- Very similar to Clock Algorithm
- Consider also the reference bits and the modified bits of pages
  - Reference (R) bit: page is referenced in the last interval
  - Modified (M) bit: page is modified after being loaded into memory
- Four possible cases (R,M):
  1. (0,0)– neither recently used nor modified (best page to replace).
  2. (0,1)– not recently used but modified (not quite as good, must write out before replacement)
  3. (1,0)– recently used but clean (probably will be used again).
  4. (1,1)– recently used and modified (probably will be used again and need to write out before replacement)
- Major difference between this algorithm and clock algorithm is that here we give preference to those pages that have been modified

Counting-Based Page Replacement

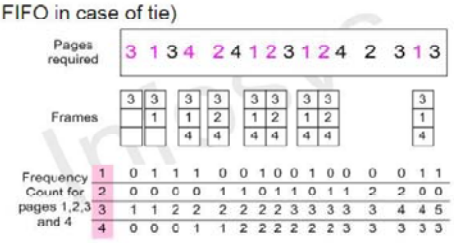
- Keep a counter of the number of references that have been made to each page, and develop the following two schemes:
- **The least frequently used (LFU) Algorithm**: replaces page with smallest count
  - Suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.
- **The most frequently used (MFU) Algorithm**: replaces the page with the largest count
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used

LFU

Page request sequence

3 1 3 4 2 4 1 2 3 1 2 4 2 3 1 3

LFU (use FIFO in case of tie)



Number of page fault is 9

Page-Buffering Algorithms

- In addition to a specific page-replacement algorithm, other procedures are also used.
- **Always maintain a pool of free frames**
  - No need to search for a free frame when the page-fault occurs
  - Load page into free frame, allowing the process to restart as soon as possible
  - The victim page is written out to the disk later, and its frame is added to the free-frame pool.
- Other modifications also exit
  - Possibly, keep list of modified pages
  - Possibly, keep free frame contents intact and note what is in them

Allocation of Frames

- How many frames to allocate to each process?
  - Each process gets the minimum number of frames which is defined by the architecture
  - The maximum number is defined by the amount of available physical memory
- **Two major frame allocation schemes**
  - Fixed allocation (this is a kind of local allocation)
    - Equal allocation
    - Proportional allocation
  - Priority allocation (this is a kind of global allocation)

Equal allocation

- If the number of frames is m and the number of processes is n, each process is allocated m/n frames
- For example, if there are 93 frames and 5 processes, each process will get 18 frames. The 3 leftover frames can be used as a free-frame buffer pool
- Pros: Easy to implement
- Cons:
  - Why allocate 20 frames to a process that might actually only need 5?
  - Can be very wasteful
  - A higher priority process doesn't get any more frames than a lower priority process

Proportional Allocation

- Allocate available frames according to the size of a process
  - Larger processes are allocated more frames than smaller processes
- Let the size of the virtual memory for process p_i be s_i:
  - S = total size of virtual memory
    = sum(s_i)
- Let the total number of available frames be m.
- Allocate a_i frames to process p_i where
  $a_i = (s_i / S) * m$
- Example: Assume:
  - m = 62 frames
  - p_1 uses 10 pages; p_2 uses 127 pages
- Allocation:
  - a_1 = (10/(10+127)) * 62 = about 4 frames
  - a_2 = (127/(10+127)) * 62 = about 57 frames
- Pros: Better allocation scheme than equal allocation
- Cons: A higher priority process doesn't get any more frames than a lower priority process

Priority Allocation

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

Global Vs Local Allocation

- When a page fault occurs for a process and we need page replacement, there are two general approaches:
  - **Global replacement** – select a victim frame from the set of all frames;
    - one process can take a frame from another
  - **Local replacement** – select a victim frame only from the frames allocated to the process.
    - A process uses always its allocated frames

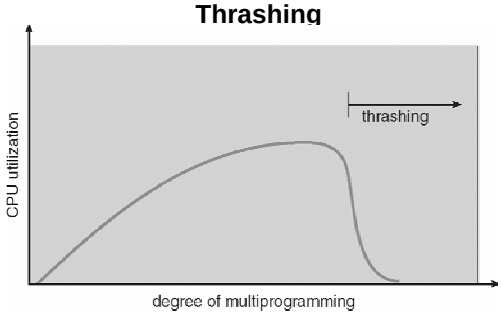
Non-Uniform Memory Access (NUMA)

- Many systems are **NUMA** – Systems in which memory access times vary significantly
  - System is made up of several system boards, each containing multiple CPUs and some memory.
  - The system boards are interconnected in various ways, ranging from system busses to high-speed network connections
  - The CPUs on a particular board can access the memory on that board with less delay than they can access memory on other boards in the system

Thrashing

- High paging activity is called thrashing
- A process is thrashing when it spends more time paging than executing (a process is busy swapping pages in and out)
- Cause of Thrashing
- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - OS spends most of its time swapping to disk

Thrashing

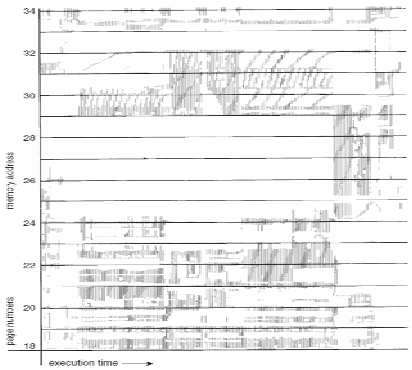


As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply.
At this point, to increase CPU utilization and stop thrashing, we must *decrease the degree of* multiprogramming.

Thrashing

- To prevent thrashing, *we must provide a process with as many frames as* it needs. But how do we know how many frames it "needs"?
- There are several techniques.
- The working-set strategy starts by looking at how frames a process is actually using. This approach defines the locality of process execution.
- The locality model states that, as a process executes, it moves from locality to locality.
- *A locality is a set of pages that are actively used together*.
- A program is generally composed of several different localities, which may overlap
- For example, when a function is called, it defines a new locality. When we exit the function, the process leaves this locality
- If we allocate enough frames to a process to accommodate its current locality, the process will not thrash.

Locality In A Memory-Reference Pattern

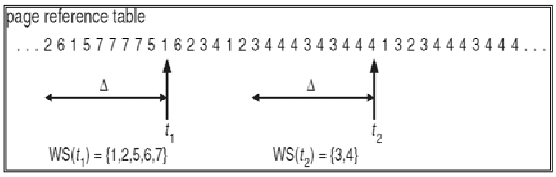


Working-Set Model (WSM)

- A method for deciding
  a) how many frames to allocate to a process, and also
  b) for selecting which page to replace.
- Working set defines minimum number of pages needed for process to behave well.
- Maintain a Working Set (WS) for each process.
  - Look to the past Δ page references
  - $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
- WSS_i (working set size of Process P_i) = total number of distinct pages referenced in the most recent Δ
  - WSS varies in time
  - Value of Δ is important
    - 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

Working-Set Model

- $D = \Sigma\ WSS_i \equiv$ total demand for frames
- if $D > m \Rightarrow$ Thrashing (m: #frames in memory)
- A possible policy: if $D >$ m, then suspend one of the processes.
- For example if $\Delta = 10$ memory references, then the WS at time t_1 is { 1, 2, 5, 6, 7} and at t_2, WS has changed to {3, 4}



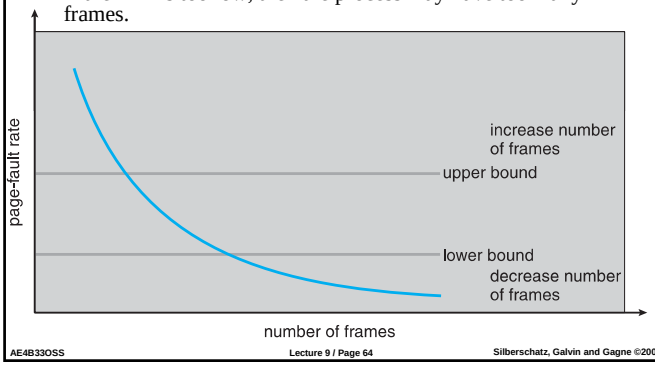
- If we give each process enough frames to accommodate its current locality, we would only get page faults when the locality changes.

Working–Set Model

- Once Δ has been selected, use of the working-set model is simple.
- The operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size. If there are enough extra frames, another process can be initiated
- If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process's pages are written out (swapped), and its frames are reallocated to other processes. The suspended process can be restarted later.
- This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization.
- The difficulty with the working-set model is keeping track of the working set.

Page fault frequency (PFF)

- Instead of working sets, we can monitor page fault frequency (PFF).
- If the PFF is too high, then the process needs more frames.
- If the PFF is too low, then the process may have too many frames.

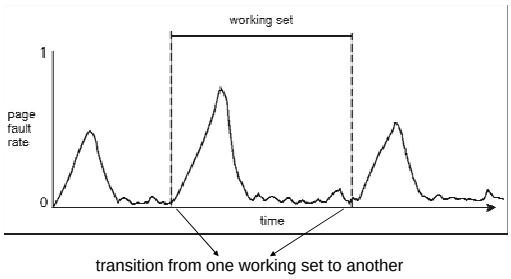


Page fault Frequency

- We can establish upper and lower bounds on the desired PFF.
- If the actual PFF exceeds the upper limit, we allocate the process another frame; if the PFF falls below the lower limit, we remove a frame from the process.
- Thus, we can directly measure and control the PFF to prevent thrashing.
- As with the working-set strategy, we may have to suspend a process.
  - If the PFF increases and no free frames are available, we must select some process and suspend it. The freed frames are then distributed to processes with high PFF.

Relationship between Working Sets and PFF

A peak in PFF occurs when we begin demand paging a new locality. However, once the working set of this new locality is in memory, the PFF falls
Time between the start of one peak to start of the next peak represents the transition from one working set to another



Allocating Kernel Memory

- Kernel memory treated differently from user memory
- OS do not subject kernel code or data to the paging system
- Kernel memory often allocated from a free-memory pool different from the list used to satisfy user processes.
- Two primary reasons for this is:
  - Kernel requests memory for data structures of varying sizes, some of which are less than a page size
  - Some kernel memory needs to be contiguous
- Kernel memory allocators
  - Buddy system
  - Slab allocation

Buddy System

- Buddy system allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated from this segment using power-of-2 allocator
  - Which satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2 (for example, if a request for 11 KB is made, it is satisfied with a 16 KB segment)
  - 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

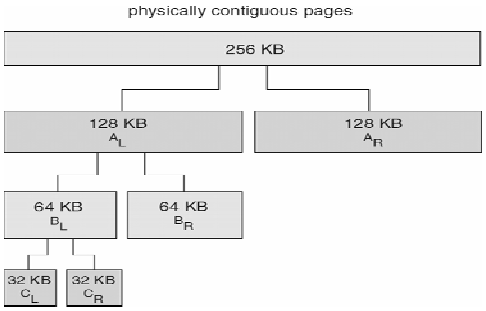
Buddy System Allocator

For example, assume 256KB memory segment is initially available and kernel requests 21KB of memory

The segment is initially divided into two buddies A_L and A_R each of size 128KB.

One of these buddies is further divided into buddies B_L and B_R of 64KB

Again One of these buddies is further into C_L and C_R of 32KB each – one of these used to satisfy 21 KB request



Buddy System

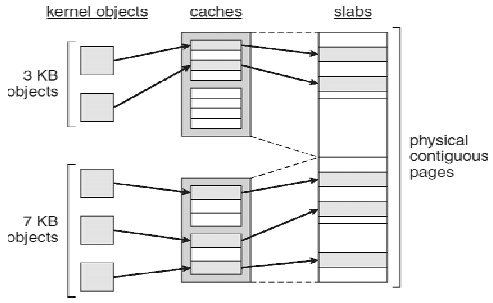
- Advantage – adjacent buddies can be combined to form larger segments using a technique known as coalescing
  - For example, when the kernel releases the C_L unit it was allocated, the system can coalesce C_L and C_R into a 64-KB segment. This segment, B_L, can in turn be coalesced with its buddy B_R to form a 128-KB segment. Ultimately, we can end up with the original 256-KB segment.
- Disadvantage – internal fragmentation
  - For example, a 33-KB request can only be satisfied with a 64-KB segment.

Slab Allocator

- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - For example, a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores, and so forth
  - 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
  - Minimize fragmentation of kernel memory
  - Most kernel memory requests can be satisfied quickly

Slab Allocation

The relationship between slabs, caches, and objects is shown below:



The figure shows two kernel objects 3 KB in size and three objects 7 KB in size. These objects are stored in their respective caches.

The number of objects in the cache depends on the size of the associated slab. For example, a 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects.

Slab Allocator in Linux

- For example In Linux system process descriptor is of type struct task_struct which requires Approx 1.7KB of memory
- When the Linux kernel creates a new task, it requests the necessary memory for the struct task_struct object from the cache
- The cache will fulfill the request using a struct task_struct object that has already been allocated in a slab and is marked as free.
- In Linux, a slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty