OPERATING SYSTEMS

VIRTUAL MEMORY

Hardware and Control Structures

Two characteristics fundamental to memory management

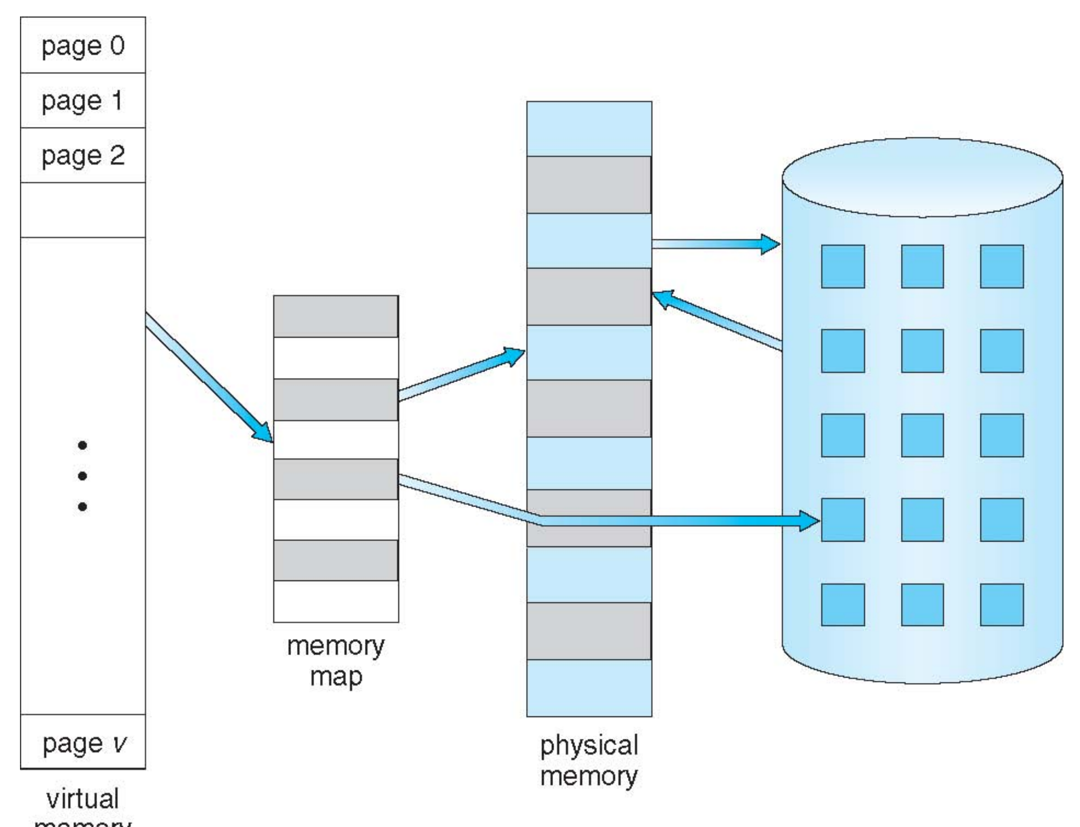
1) **All memory references are logical addresses** that are dynamically translated into physical addresses at run time
2) **A process may be broken up into a number of pieces** that don't need to be contiguously located in main memory during execution

It is not necessary that all the pages or segments of a process be in main memory during execution

Implications

- More processes may be maintained in main memory
  - Only some of the pieces of any particular process are loaded
  - This leads to more efficient utilization of the processor because it is more likely that at least one of the processes will be in a ready state at any particular time
- A process may be larger than all of main memory
  - Solution with overlays: the programmer must devise ways to structure the program into pieces that can be loaded separately
  - With virtual memory based on paging or segmentation, that job is left to the OS and the hardware

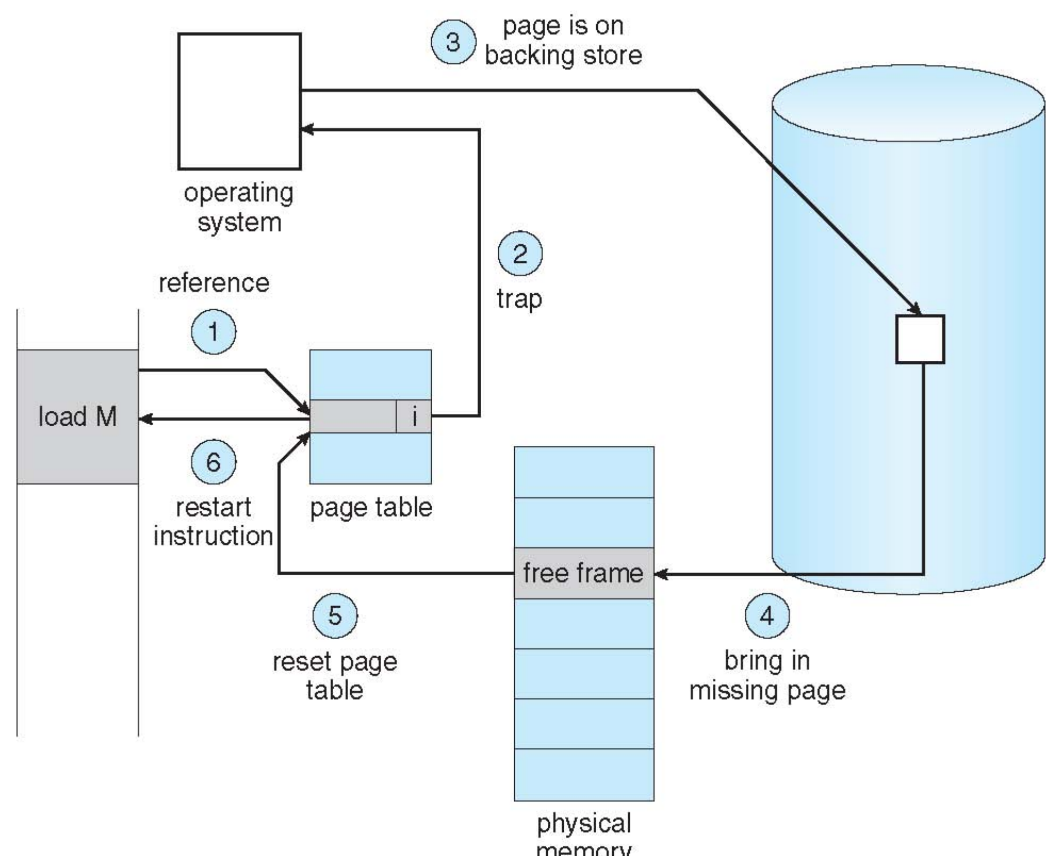
Virtual Memory That is Larger Than Physical Memory



Execution of a Process

- The OS brings into main memory a few pieces of the program
- **Resident set**
  Portion of process that is in main memory
- When an **address** is needed that is **not in main memory**, the OS generates an **interrupt**
- The OS places the **process** in a **blocking state**

Steps in Handling a Page Fault



Real and Virtual Memory

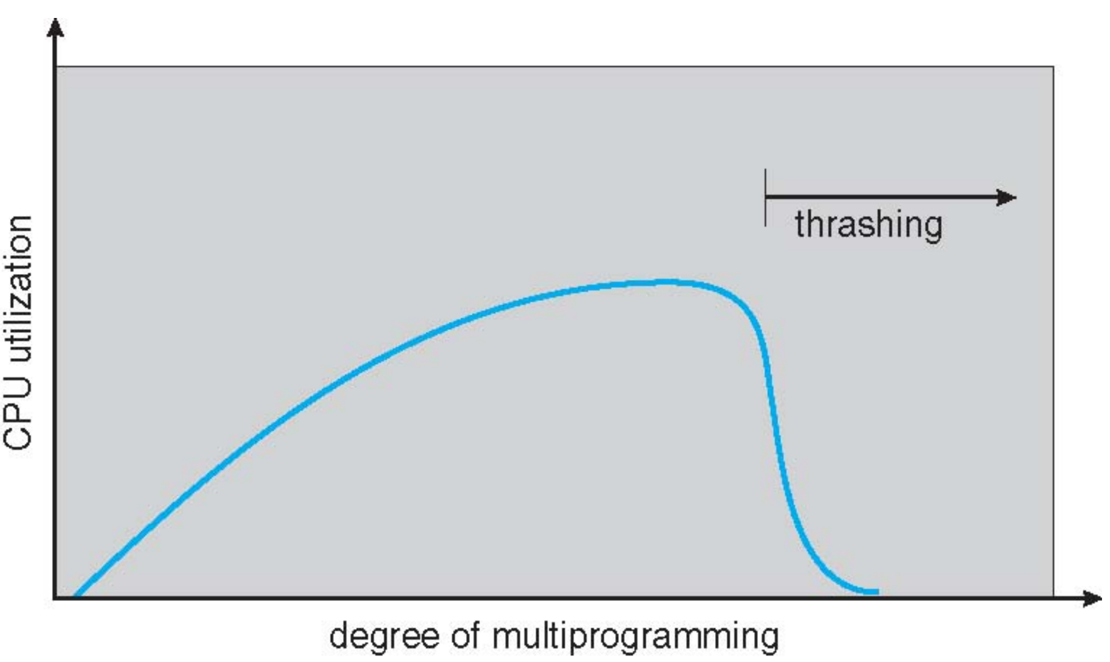
- **Real memory**
  - Main memory, the actual RAM
- **Virtual memory**
  - Memory on disk
  - Allows for effective multiprogramming and relieves the user of tight constraints of main memory

Trashing

A state in which **the system spends most of its time swapping process pieces rather than executing instructions**

To avoid this, **the OS tries to guess,** based on recent history, which **pieces are least likely to be used in the near future**

Thrashing

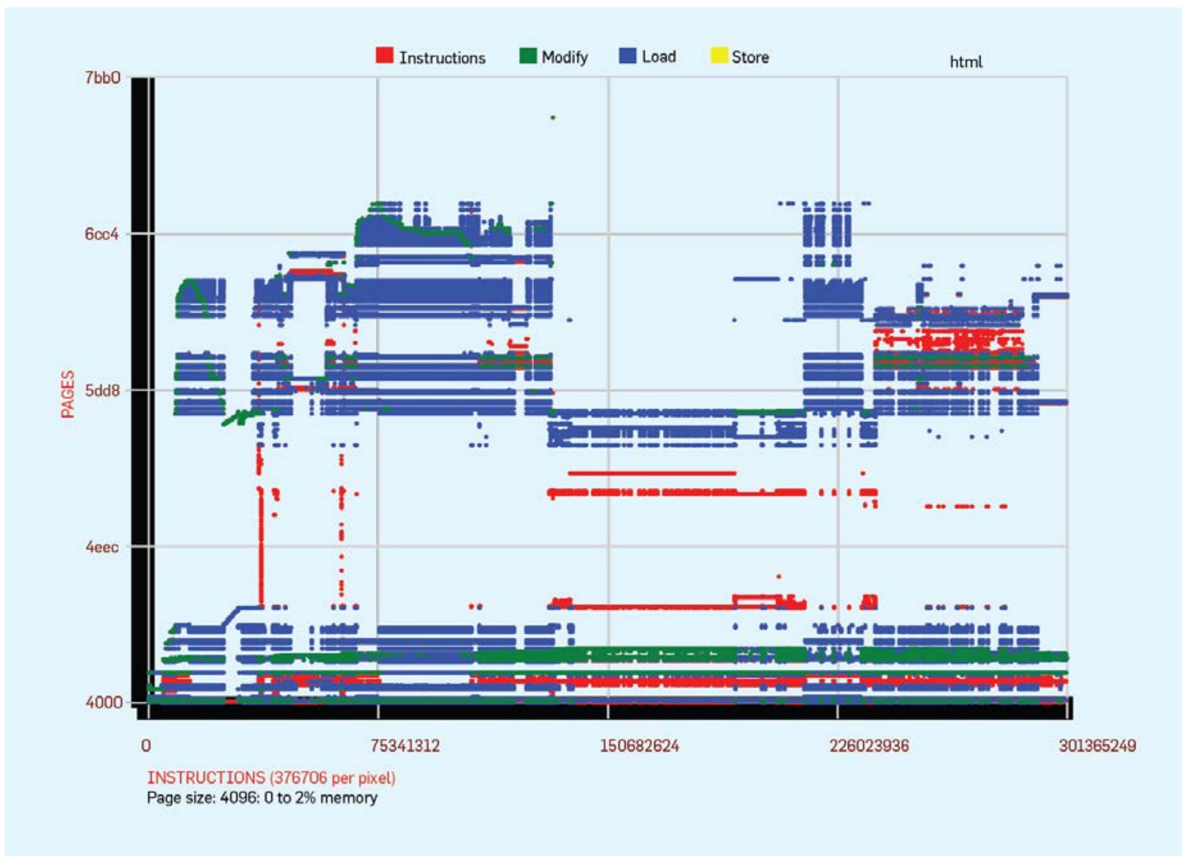


Principle of Locality

- Program and data references within a process tend to cluster
- Only a few pieces of a process will be needed over a short period of time
- Therefore it is possible to make intelligent guesses about which pieces will be needed in the future
- Avoids thrashing

Principle of Locality

Memory map of a Firefox Browser in Linux



Communications of the ACM, 09/2017

Support Needed for Virtual Memory

For virtual memory to be practical and effective:

- Hardware must support paging and segmentation
- Operating system must include software for managing the movement of pages and/or segments between secondary memory and main memory

Demand Paging

Demand Paging

- A page is loaded 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
- 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

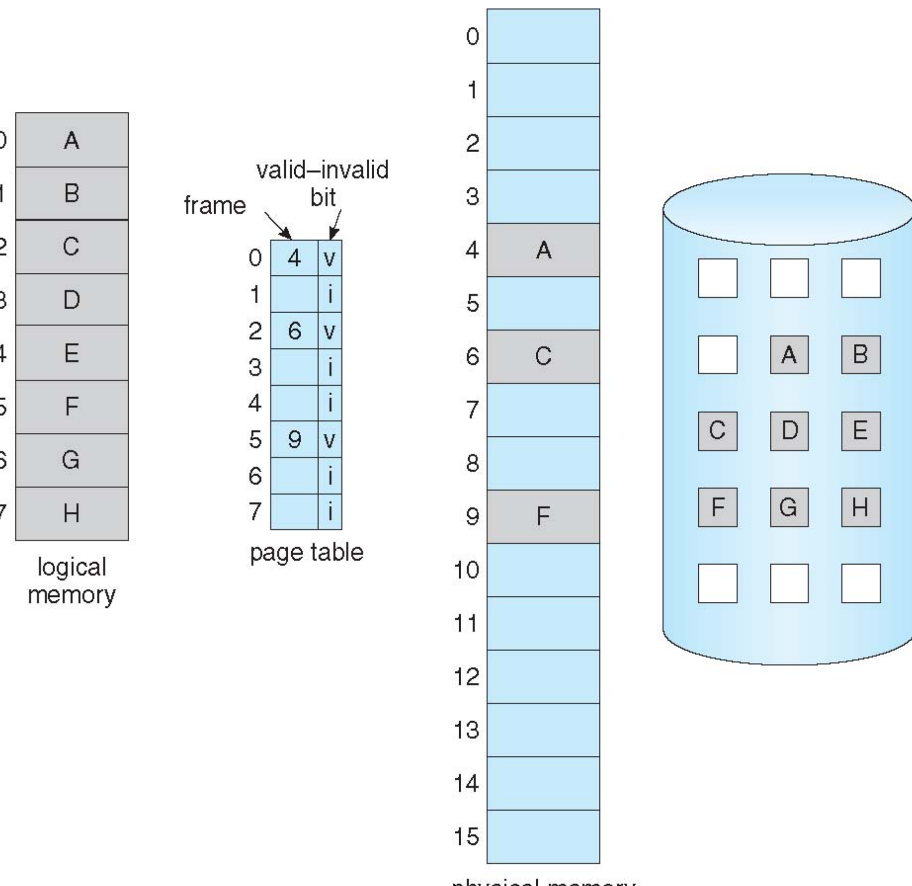
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  v ⇒ in-memory,
  i ⇒ not-in-memory
- During MMU address translation,
  if valid–invalid bit in page table entry is i ⇒ page fault
- Initially valid–invalid bit is set to i on all entries

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

page table

Page Table When Some Pages Are Not in Main Memory



Performance of Demand Paging

- Three major activities
  - Service the interrupt
    careful coding means just several hundred instructions needed
  - Read the page
    very slow operation
  - Restart the process
    fast operation
- 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)

EAT = (1 – p) x memory access

+ p page fault overhead

+ swap page out

+ swap page in

Demand Paging Example

Memory access time = 200 nanoseconds

Average page-fault service time = 8 milliseconds

EAT = (1 – p) x 200 + p (8 milliseconds)

= (1 – p) x 200 + p x 8,000,000

= 200 + p x 7,999,800

If one access out of 1,000 causes a page fault, then

EAT = 8.2 microseconds.

This is a slowdown by a factor of 40!!

If we want performance degradation < 10 percent

220 > 200 + 7,999,800 x p
20 > 7,999,800 x p

p < .0000025

< one page fault in every 400,000 memory accesses

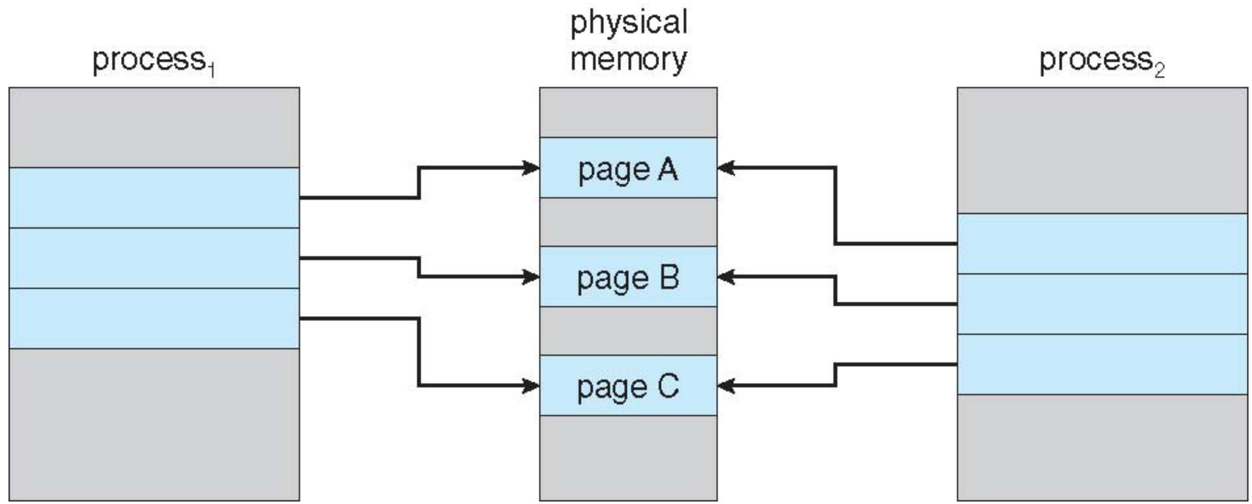
Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks
- Copy entire process image to swap space at process load time (*used in older BSD Unix*)
  - Then page in and out of swap space
- Discard rather than paging out when freeing frames containing program binaries (*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 (such as code)

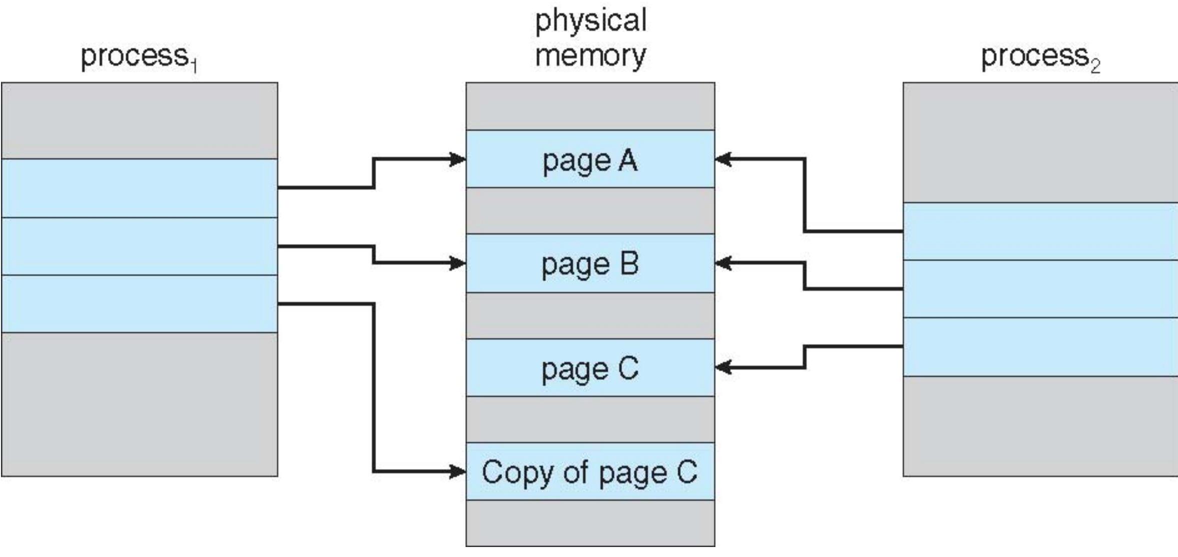
Copy-on-Write

- 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

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



What Happens if There is no Free Frame?

- Main memory demanded non only 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**
    an algorithm which will result in minimum number of page faults
- Drawback
  Same page may be brought into memory several times

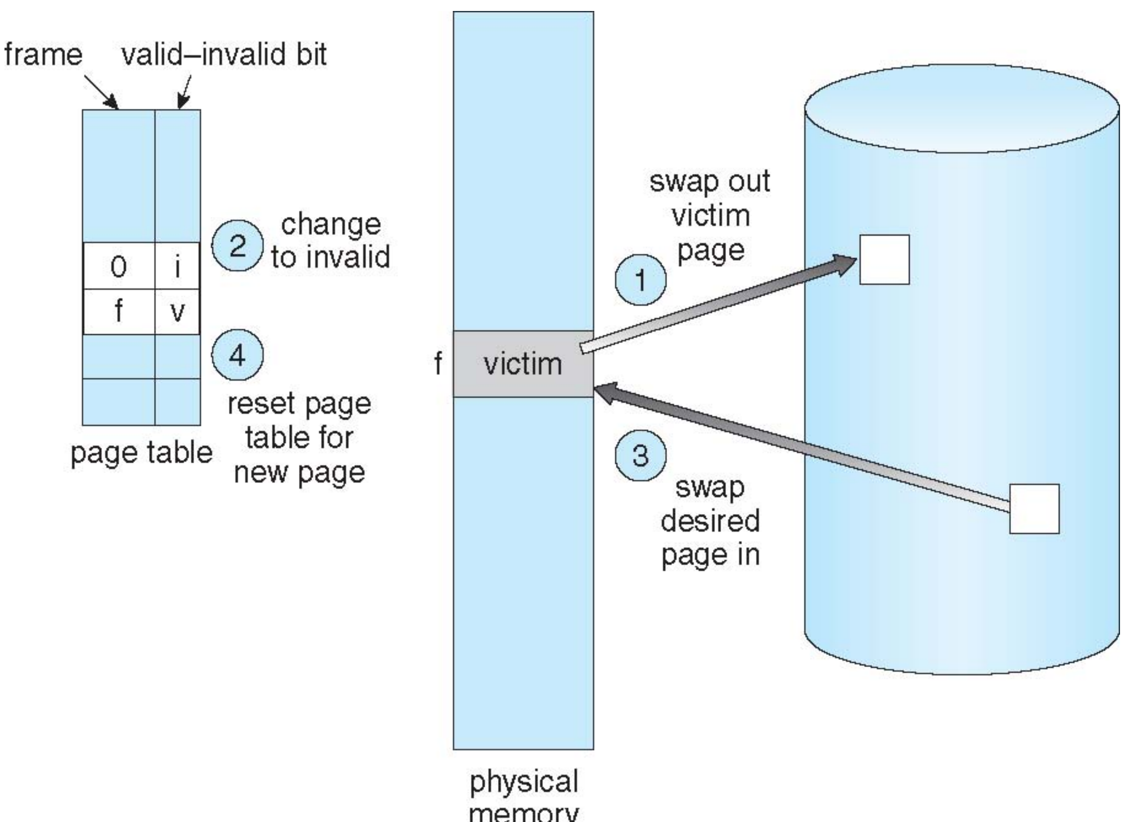
Page Replacement

- 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

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
5. *Note:* now potentially 2 page transfers for page fault – increasing EAT

Page Replacement

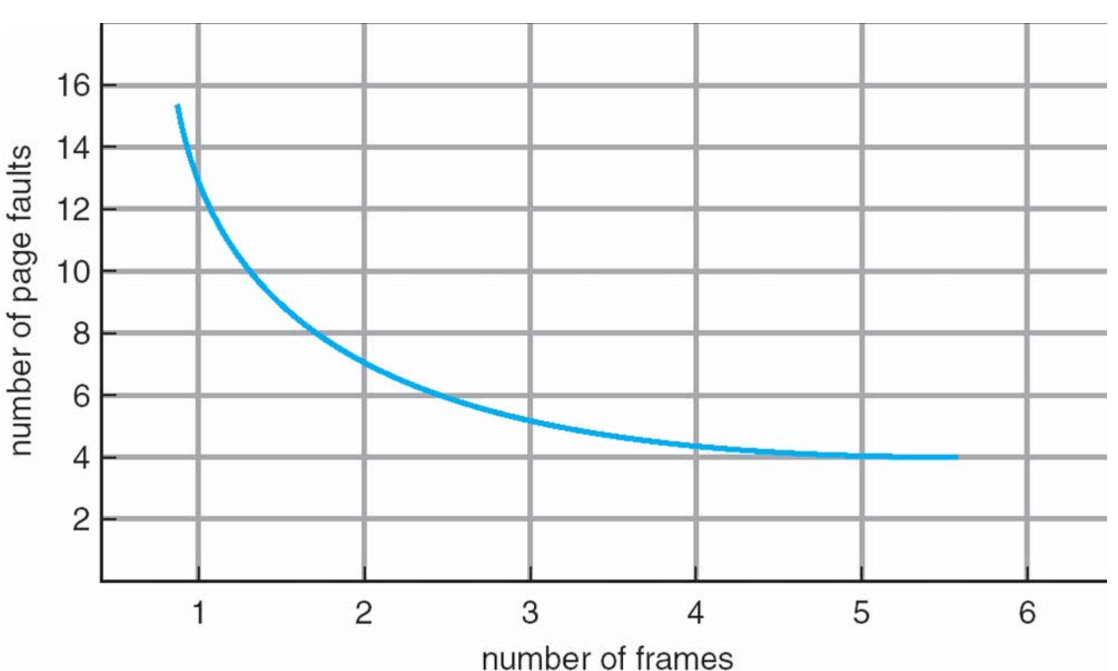


Page and Frame Replacement Algorithms

- Frame-allocation algorithm determines
  - How many frames to give each process
  - Which frames to replace
- Page-replacement algorithm goal
  - 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

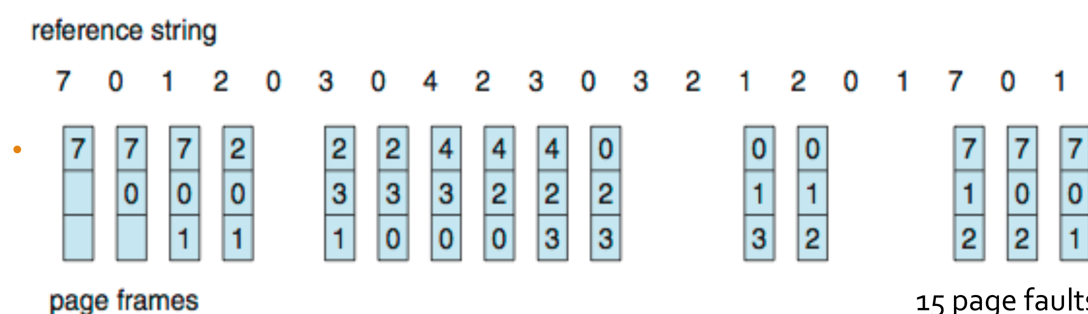
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Desired Behaviour of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

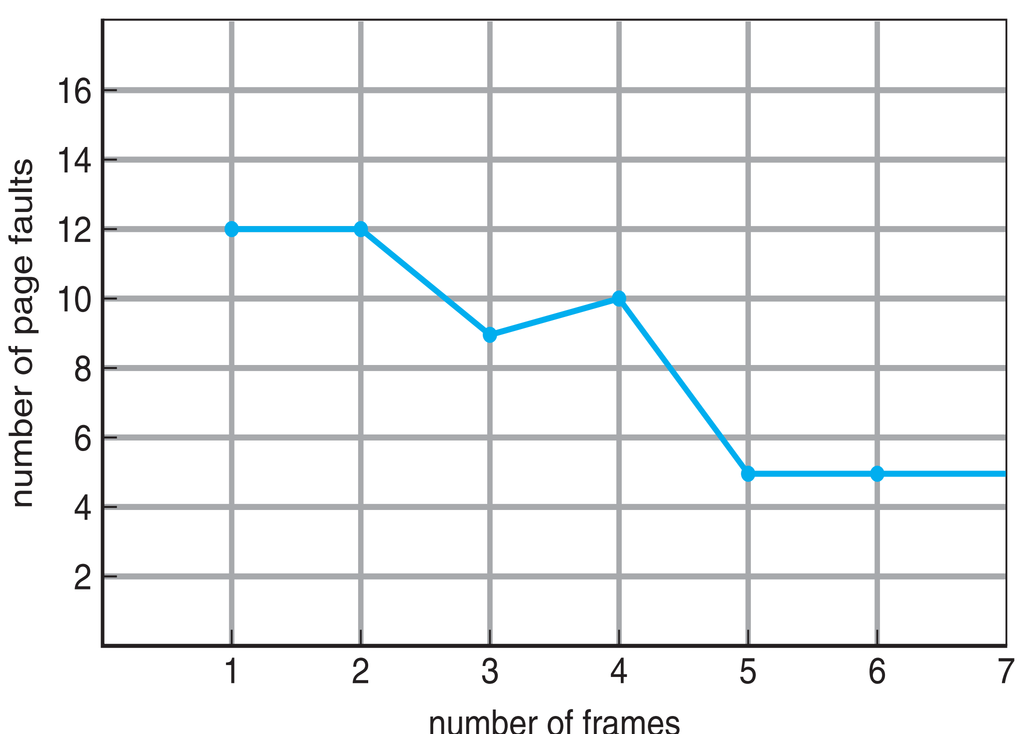
- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

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

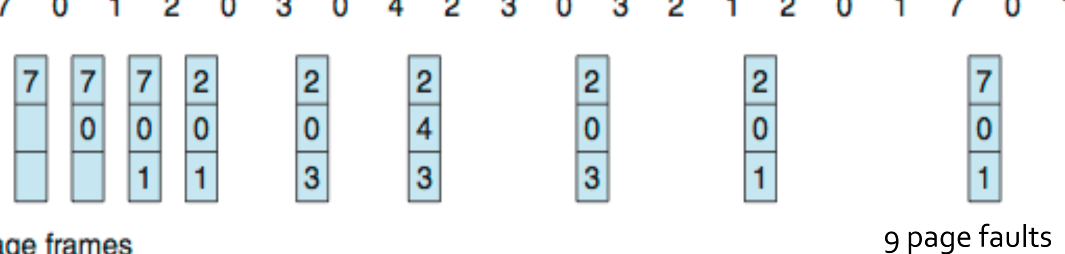
FIFO Illustrating Belady's Anomaly



Optimal Algorithm

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

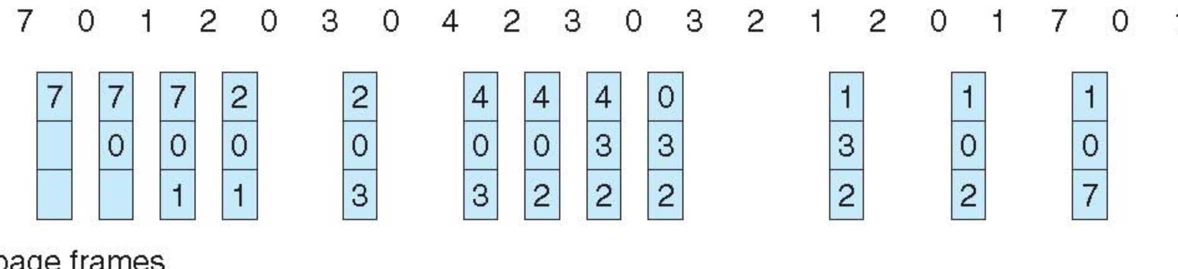
reference string



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

Least Recently Used (LRU) Algorithm

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

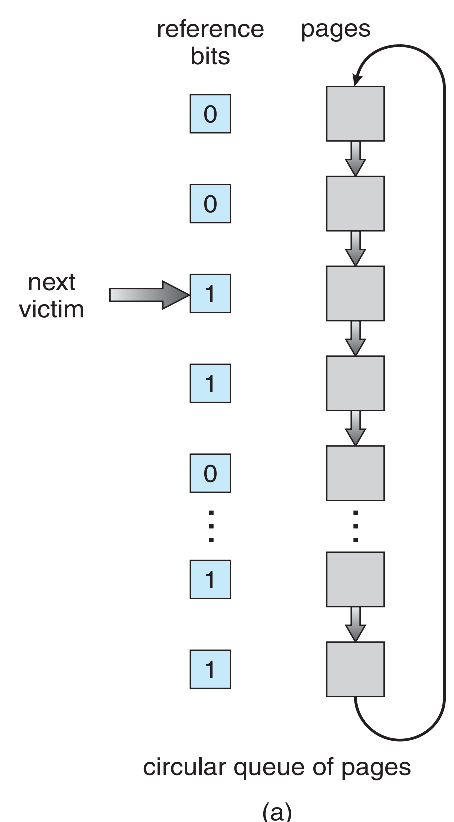
LRU Algorithm

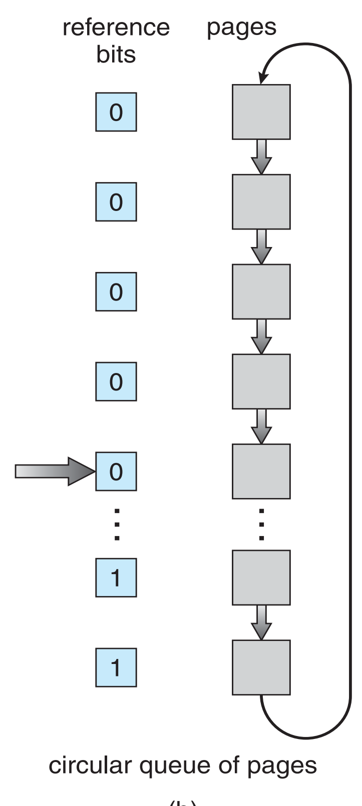
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of stack algorithms that don't have Belady's Anomaly

LRU Approximation Algorithms

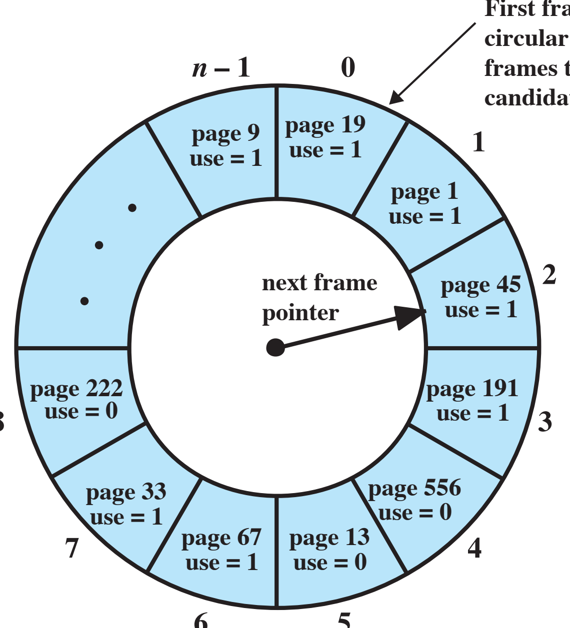
- LRU needs special hardware and still slow
- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - Clock replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm

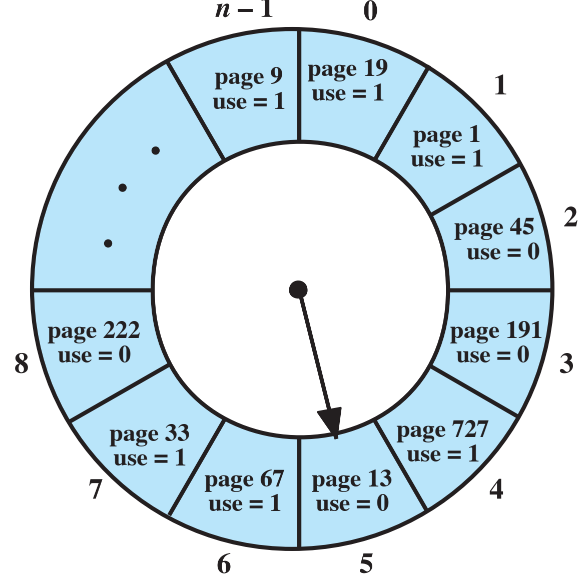




Second-Chance (clock) Page-Replacement Algorithm



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



(b) State of buffer just after the next 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
  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

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-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

Prepaging

- Pages other than the one demanded by a page fault are brought in
- Exploits the characteristics of most secondary memory devices
  - If pages of a process are stored contiguously in secondary memory it is more efficient to bring in a number of pages at one time
- Ineffective if extra pages are not referenced
- Should not be confused with “swapping”

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge
  - i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can give direct access to the disk, getting out of the way of the applications
  - Raw disk mode
  - Bypasses buffering, locking, etc

Resident Set Management

Resident Set Management

- The OS must decide how many pages to bring into main memory
  - The smaller the amount of memory allocated to each process, the more processes can reside in memory
  - Small number of pages loaded increases page faults
- Beyond a certain size, further allocations of pages will not effect the page fault rate

Resident Set Size

- **Fixed Allocation**
  Gives a process a fixed number of frames in main memory within which to execute
  - When a page fault occurs, one of the pages of that process must be replaced
- **Variable Allocation**
  Allows the number of page frames allocated to a process to be varied over the lifetime of the process

Replacement Scope

- The scope of a replacement strategy can be categorized as **global** or **local**
  - Both types are activated by a page fault when there are no free page frames

Local

- Chooses only among the resident pages of the process that generated the page fault

Global

- Considers all unlocked pages in main memory

Fixed Allocation, Local Scope

- Necessary to decide ahead of time the amount of allocation to give a process
- If allocation is too small, there will be a high page fault rate

If allocation is too large, there will be too few programs in main memory

- Increased processor idle time
- Increased time spent in swapping

Variable Allocation Global Scope

Easiest to implement

Adopted in a number of operating systems

OS maintains a list of free frames

Free frame is added to resident set of process when a page fault occurs

If no frames are available the OS must choose a page currently in memory

One way to counter potential problems is to use page buffering

Variable Allocation Local Scope

When a new process is loaded into main memory, allocate to it a certain number of page frames as its resident set

When a page fault occurs, select the page to replace from among the resident set of the process that suffers the fault

Reevaluate the allocation provided to the process and increase or decrease it to improve overall performance

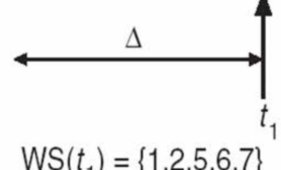
Variable Allocation Local Scope

- Decision to increase or decrease a resident set size is based on the assessment of the likely future demands of active processes

Key elements:
• Criteria used to determine resident set size • The timing of changes

Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ
  - 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

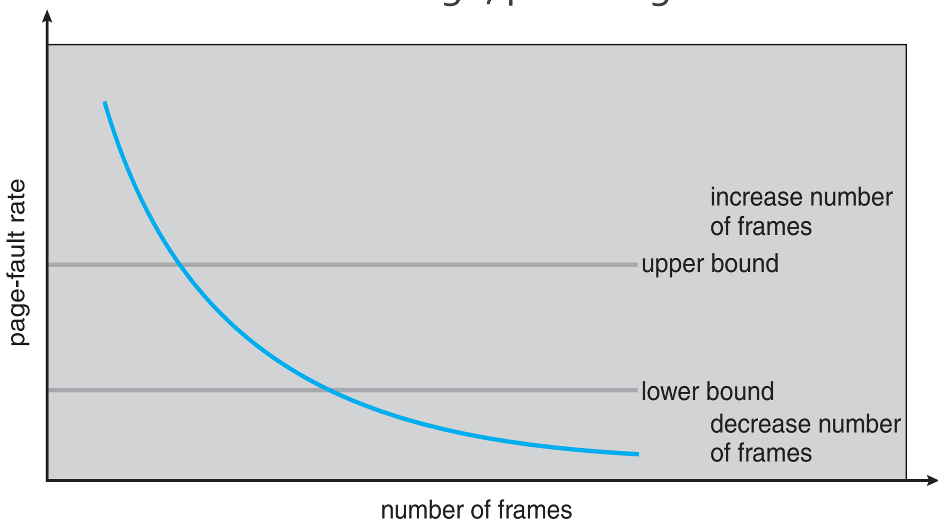


Working-Set Model

- $D = \Sigma \, WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

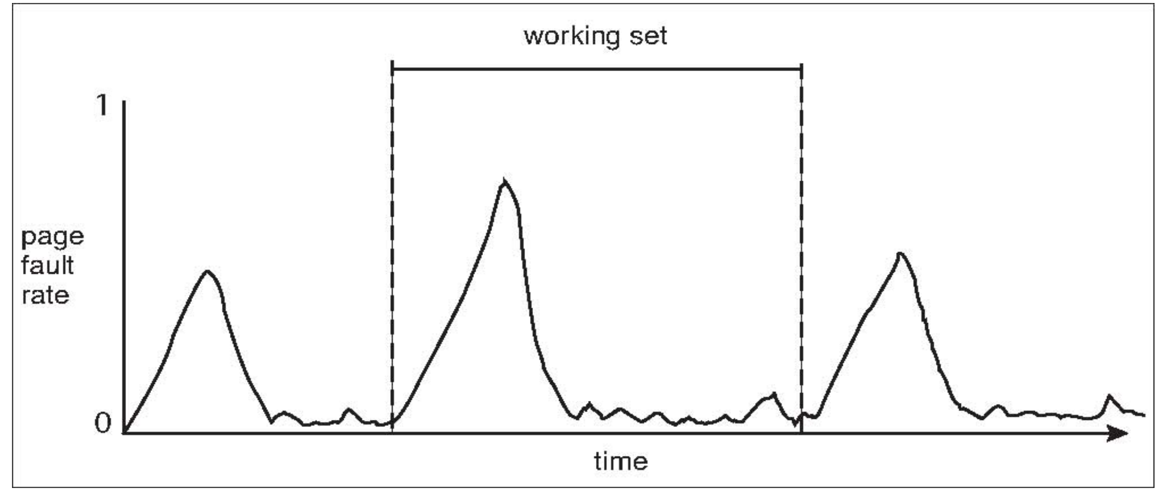
Page-Fault Frequency

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



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Kernel Memory Allocation

Allocating Kernel Memory

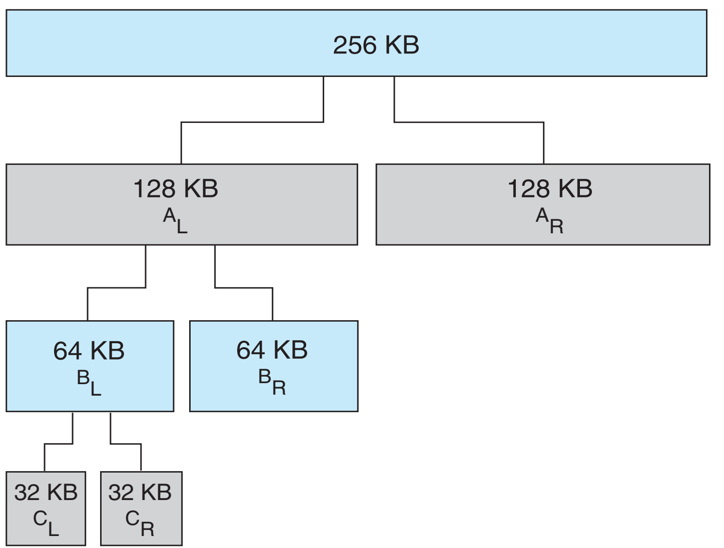
- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O

Buddy System

- Allocates memory from fixed-size segment 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

For example, assume 256KB chunk available, kernel requests 21KB



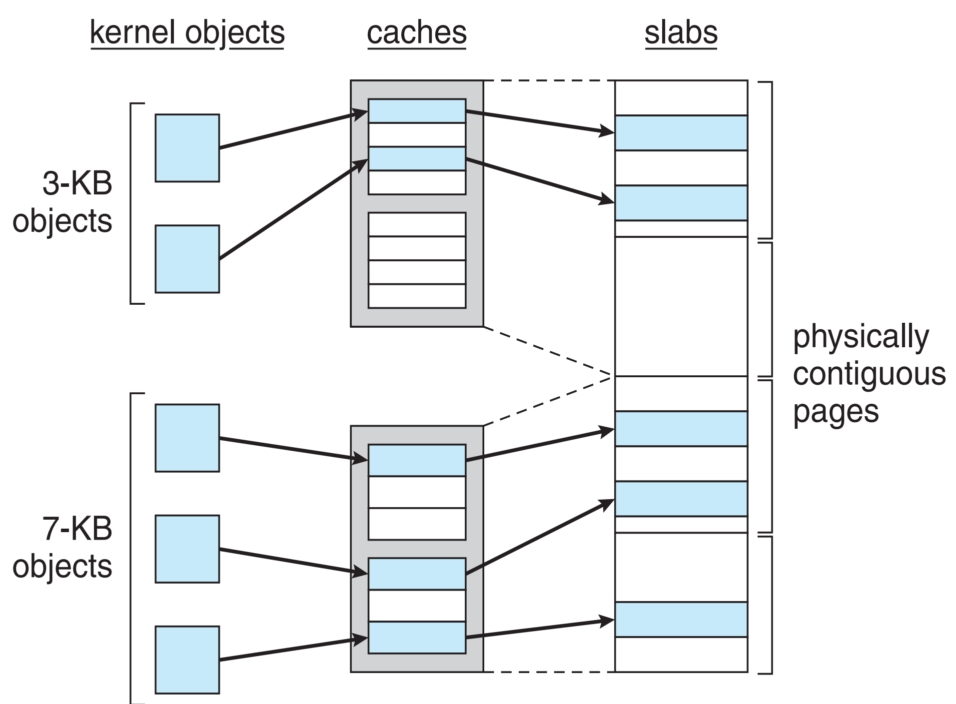
Advantage – quickly coalesce unused chunks into larger chunk

Disadvantage – fragmentation

Slab Allocator

- **Slab** is one or more physically 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



Design Issues

Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration
  - Fragmentation
  - Page table size
  - Resolution
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

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

Program Structure

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

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

  - 128 x 128 = 16,384 page faults

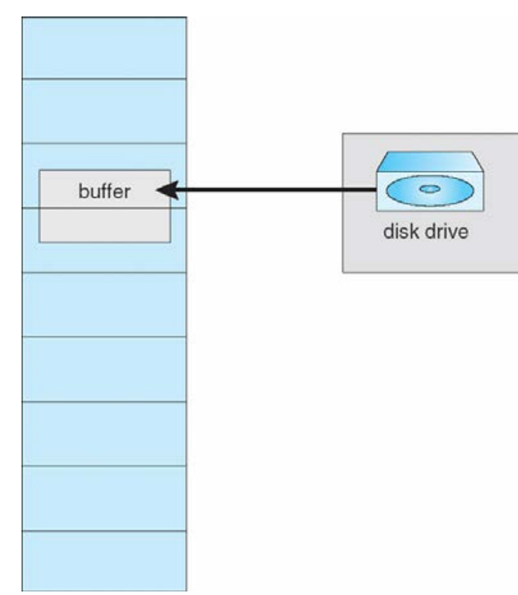
  - Program 2

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

  - 128 page faults

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



Operating System Examples

Windows

- Uses demand paging with clustering. Clustering brings in pages surrounding the faulting page
- Processes are assigned working set minimum and working set maximum
  - Working set minimum is the minimum number of pages the process is guaranteed to have in memory
  - A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, automatic working set trimming is performed to restore the amount of free memory
  - Working set trimming removes pages from processes that have pages in excess of their working set minimum

Solaris

- Maintains a list of free pages to assign faulting processes
- **Lotsfree** threshold parameter (amount of free memory) to begin paging
- **Desfree** threshold parameter to increasing paging
- **Minfree** threshold parameter to being swapping
- Paging is performed by pageout process
- Pageout scans pages using modified clock algorithm
- **Scanrate** is the rate at which pages are scanned. This ranges from slowscan to fastscan
- Pageout is called more frequently depending upon the amount of free memory available
- Priority paging gives priority to process code pages

Linux Virtual Memory

Three level page table structure

Page directory

- Process has a single page directory
- Each entry points to one page of the page middle directory
- Must be in main memory for an active process

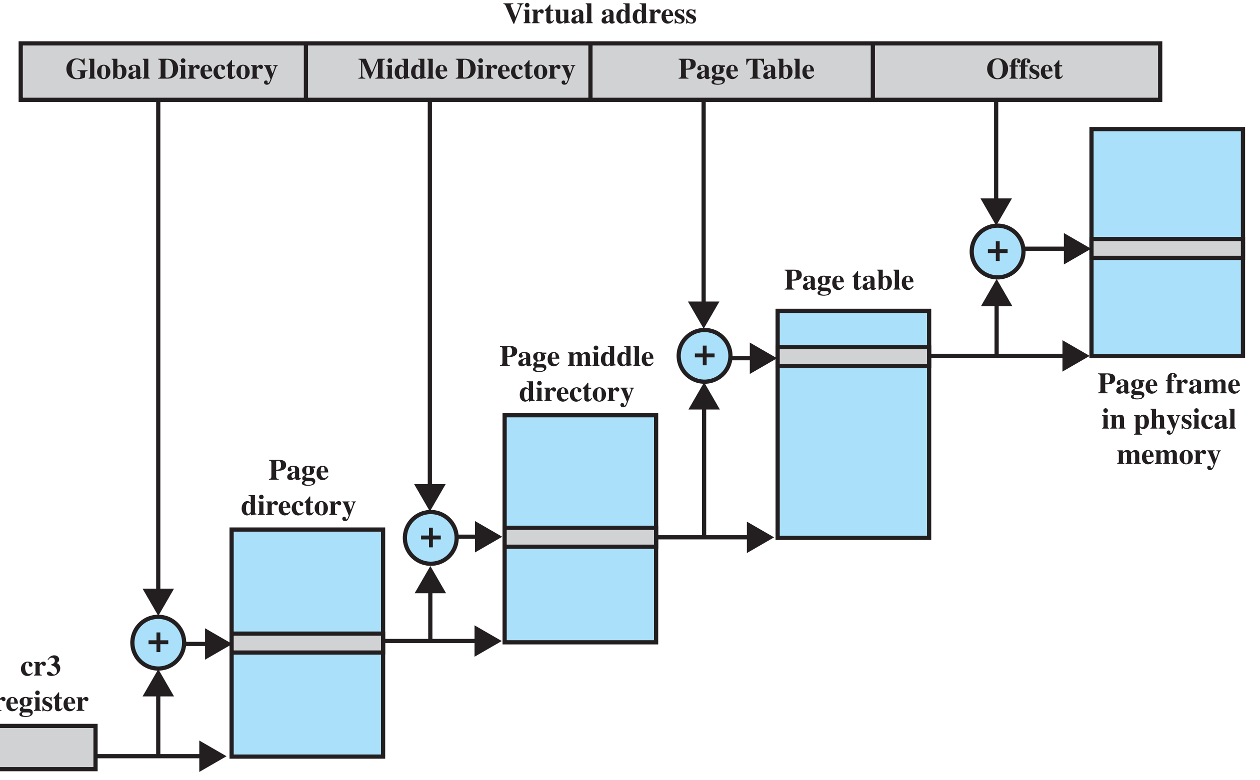
Page middle directory

- May span multiple pages
- Each entry points to one page in the page table

Page table

- May also span multiple pages
- Each entry refers to one virtual page of the process

Address Transaltion in Linux Virtual Memory Scheme



Linux Page Replacement

Based on the clock algorithm

The use bit is replaced with an 8-bit age variable

Incremented each time the page is accessed

Periodically decrements the age bits

A page with an age of 0 is an "old" page that has not been referenced in some time and is the best candidate for replacement

A form of least frequently used policy