ICS 143 - Principles of Operating Systems

Lectures 10,11,12 and13 - Memory Management

Prof. Nalini Venkatasubramanian

nalini@ics.uci.edu

Outline

- Background
- Logical versus Physical Address Space
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging

Background

- Program must be brought into memory and placed within a process for it to be executed.
- Input Queue - collection of processes on the disk that are waiting to be brought into memory for execution.
- User programs go through several steps before being executed.

Names and Binding

- Symbolic names → Logical names → Physical names
  - Symbolic Names: known in a context or path
    - file names, program names, printer/device names, user names
  - Logical Names: used to label a specific entity
    - inodes, job number, major/minor device numbers, process id (pid), uid, gid..
  - Physical Names: address of entity
    - inode address on disk or memory
    - entry point or variable address
    - PCB address

Binding of instructions and data to memory

- Address binding of instructions and data to memory addresses can happen at three different stages.
  - Compile time:
    - If memory location is known apriori, absolute code can be generated; must recompile code if starting location changes.
  - Load time:
    - Must generate relocatable code if memory location is not known at compile time.
  - Execution time:
    - Binding delayed until runtime if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g. base and limit registers).

Binding time tradeoffs

- Early binding
  - compiler - produces efficient code
  - allows checking to be done early
  - allows estimates of running time and space
- Delayed binding
  - Linker, loader
  - produces efficient code, allows separate compilation
  - portability and sharing of object code
- Late binding
  - VM, dynamic linking/loading, overlaying, interpreting
  - code less efficient, checks done at runtime
  - flexible, allows dynamic reconfiguration

Dynamic Loading

- Routine is not loaded until it is called.
- Better memory-space utilization; unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the operating system is required; implemented through program design.

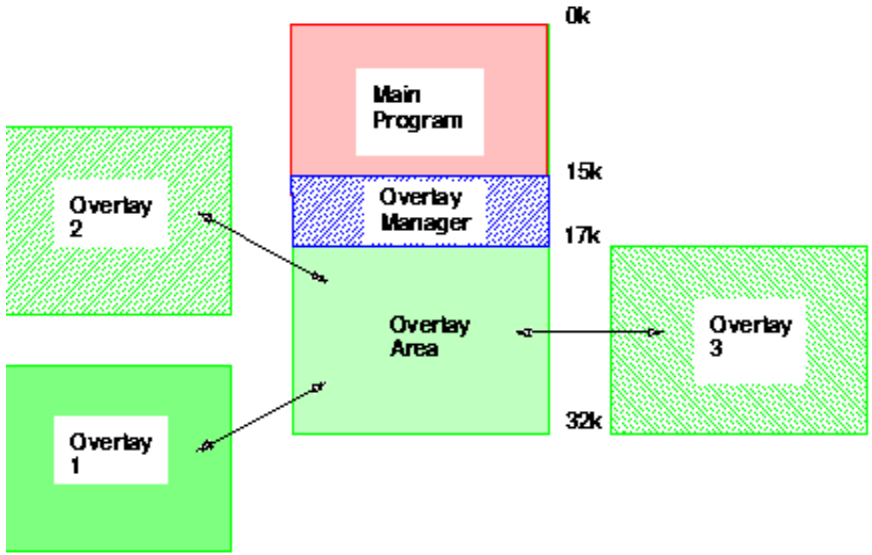
Dynamic Linking

- Linking postponed until execution time.
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine.
- Stub replaces itself with the address of the routine, and executes the routine.
- Operating system needed to check if routine is in processes' memory address.

Overlays

- Keep in memory only those instructions and data that are needed at any given time.
- Needed when process is larger than amount of memory allocated to it.
- Implemented by user, no special support from operating system; programming design of overlay structure is complex.

Overlaying


0k
Main Program
15k
Overlay Manager
17k
Overlay 2
Overlay Area
Overlay 3
Overlay 1
32k

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.
  - Logical Address: or virtual address - generated by CPU
  - Physical Address: address seen by memory unit.
- Logical and physical addresses are the same in compile time and load-time binding schemes
- Logical and physical addresses differ in execution-time address-binding scheme.

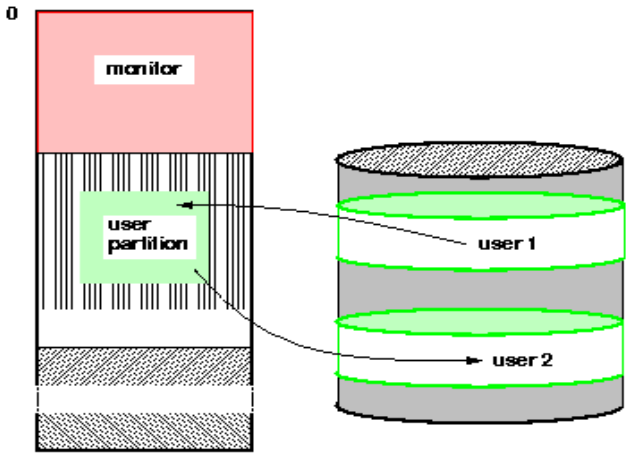
Memory Management Unit (MMU)

- Hardware device that maps virtual to physical address.
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with logical addresses; it never sees the real physical address.

Swapping

- A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
  - Backing Store - fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
  - Roll out, roll in - swapping variant used for priority based scheduling algorithms; lower priority process is swapped out, so higher priority process can be loaded and executed.
  - Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.
  - Modified versions of swapping are found on many systems, i.e. UNIX and Microsoft Windows.

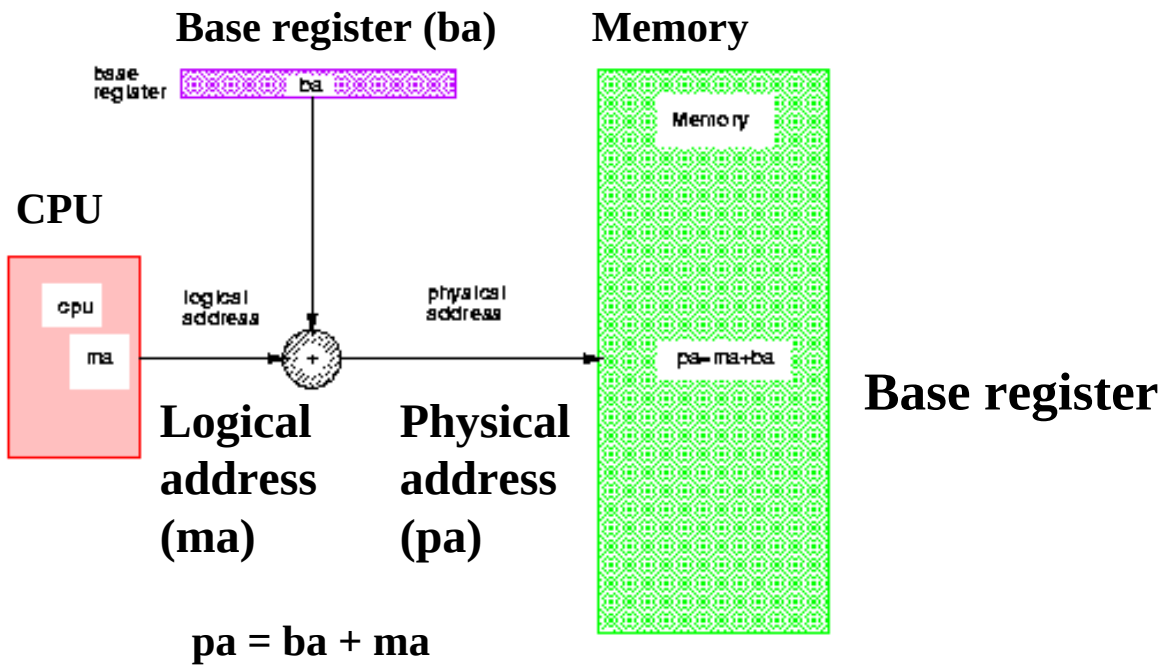
Schematic view of swapping



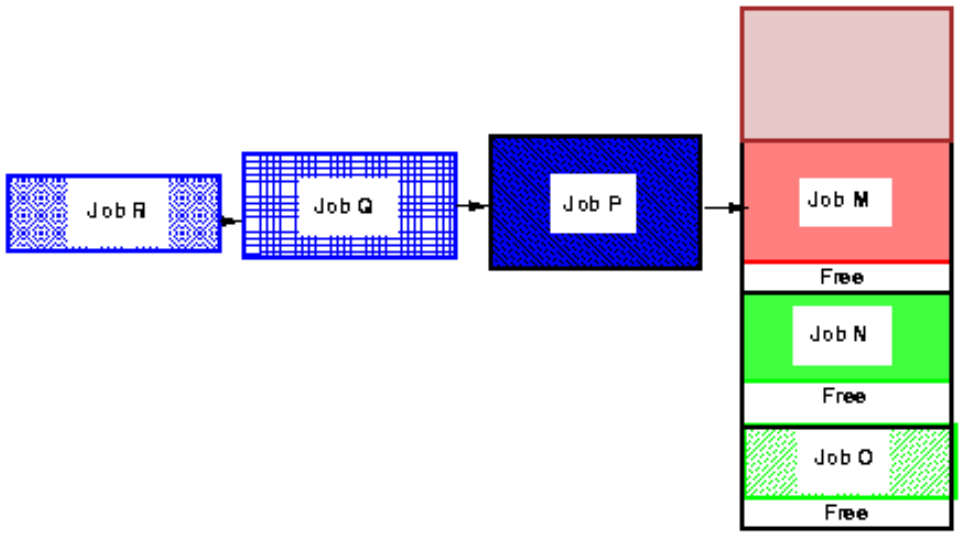
Contiguous Allocation

- Main memory usually into two partitions
  - Resident Operating System, usually held in low memory with interrupt vector.
  - User processes then held in high memory.
- Single partition allocation
  - Relocation register scheme used to protect user processes from each other, and from changing OS code and data.
  - Relocation register contains value of smallest physical address; limit register contains range of logical addresses - each logical address must be less than the limit register.

Relocation Register



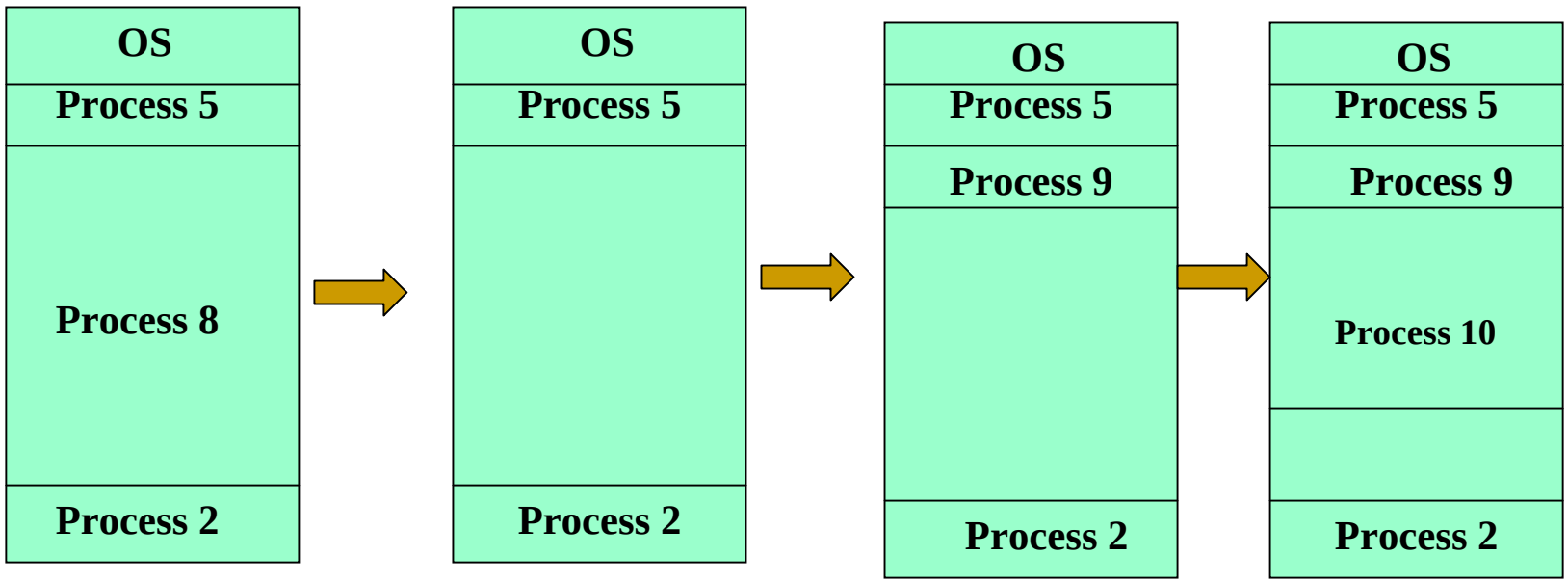
Fixed partitions



Contiguous Allocation (cont.)

- Multiple partition Allocation
  - Hole - block of available memory; holes of various sizes are scattered throughout memory.
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it.
  - Operating system maintains information about
    - allocated partitions
    - free partitions (hole)

Contiguous Allocation example



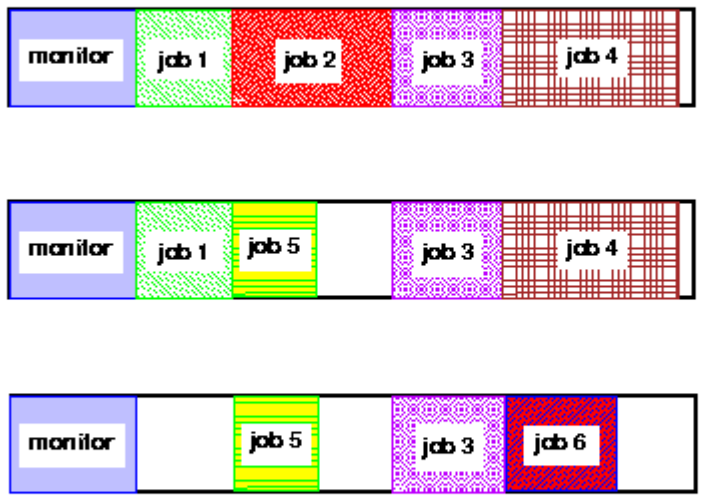
Dynamic Storage Allocation Problem

- How to satisfy a request of size n from a list of free holes.
  - First-fit
    - allocate the first hole that is big enough
  - Best-fit
    - Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
  - Worst-fit
    - Allocate the largest hole; must also search entire list. Produces the largest leftover hole.
- First-fit and best-fit are better than worst-fit in terms of speed and storage utilization.

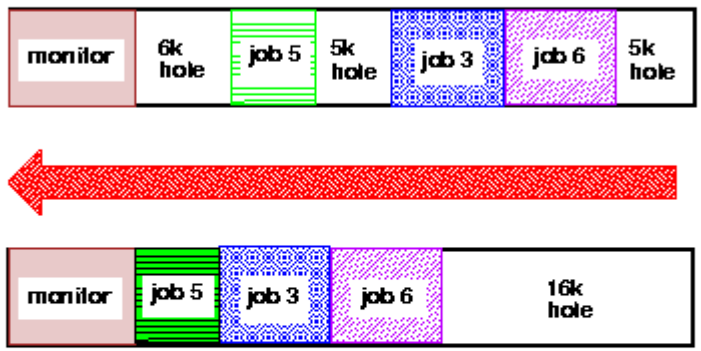
Fragmentation

- External fragmentation
  - total memory space exists to satisfy a request, but it is not contiguous.
- Internal fragmentation
  - allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time.
  - I/O problem - (1) latch job in memory while it is in I/O (2) Do I/O only into OS buffers.

Fragmentation example



Compaction


monitor
6k hole
job 5
5k hole
job 3
job 6
5k hole
monitor
job 5
job 3
job 6
16k hole

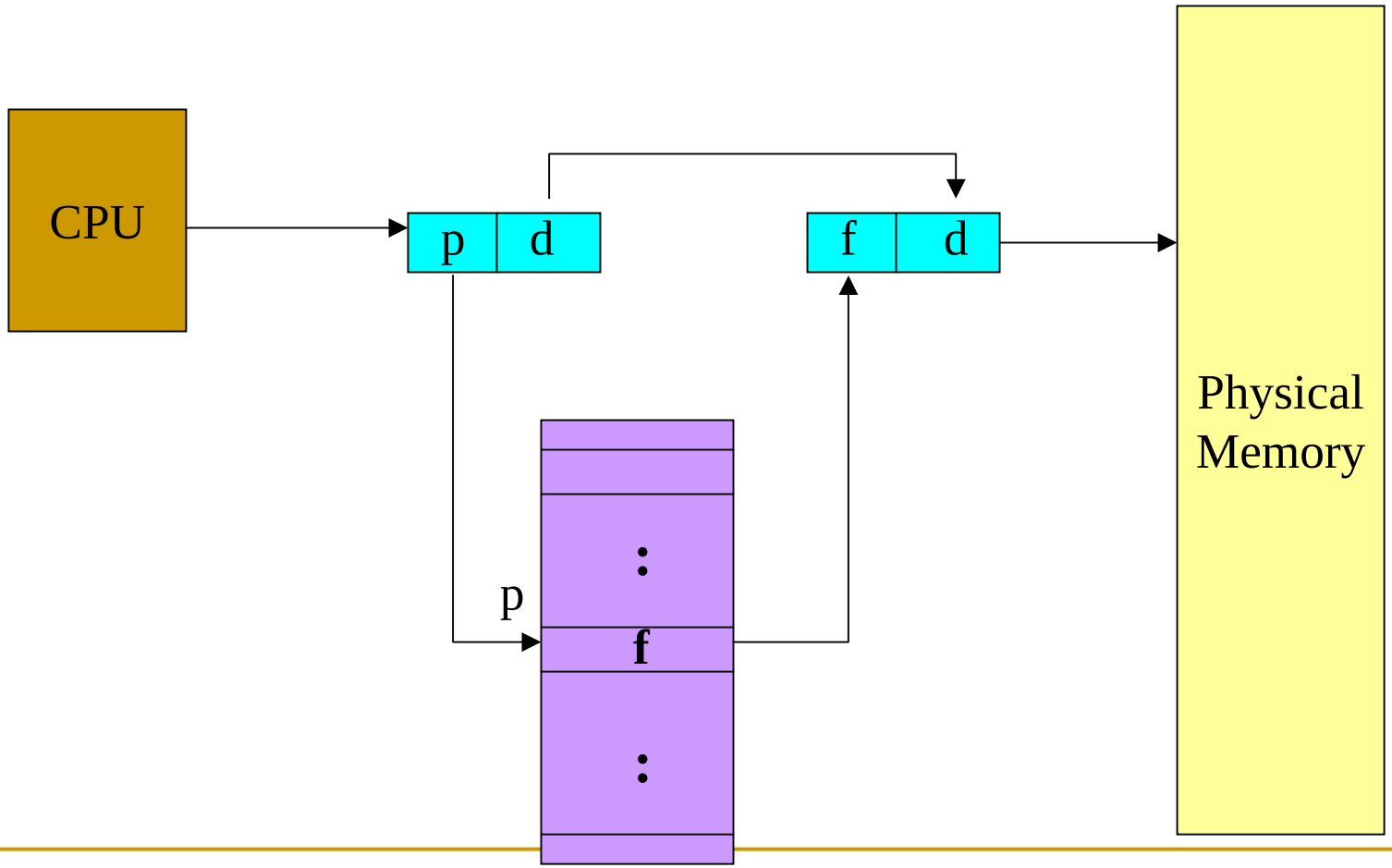
Paging

- Logical address space of a process can be non-contiguous;
  - process is allocated physical memory wherever the latter is available.
  - Divide physical memory into fixed size blocks called ***frames***
    - size is power of 2, 512 bytes - 8K
  - Divide logical memory into same size blocks called ***pages.***
    - Keep track of all free frames.
    - To run a program of size n pages, find n free frames and load program.
  - Set up a page table to translate logical to physical addresses.
  - Note:: Internal Fragmentation possible!!

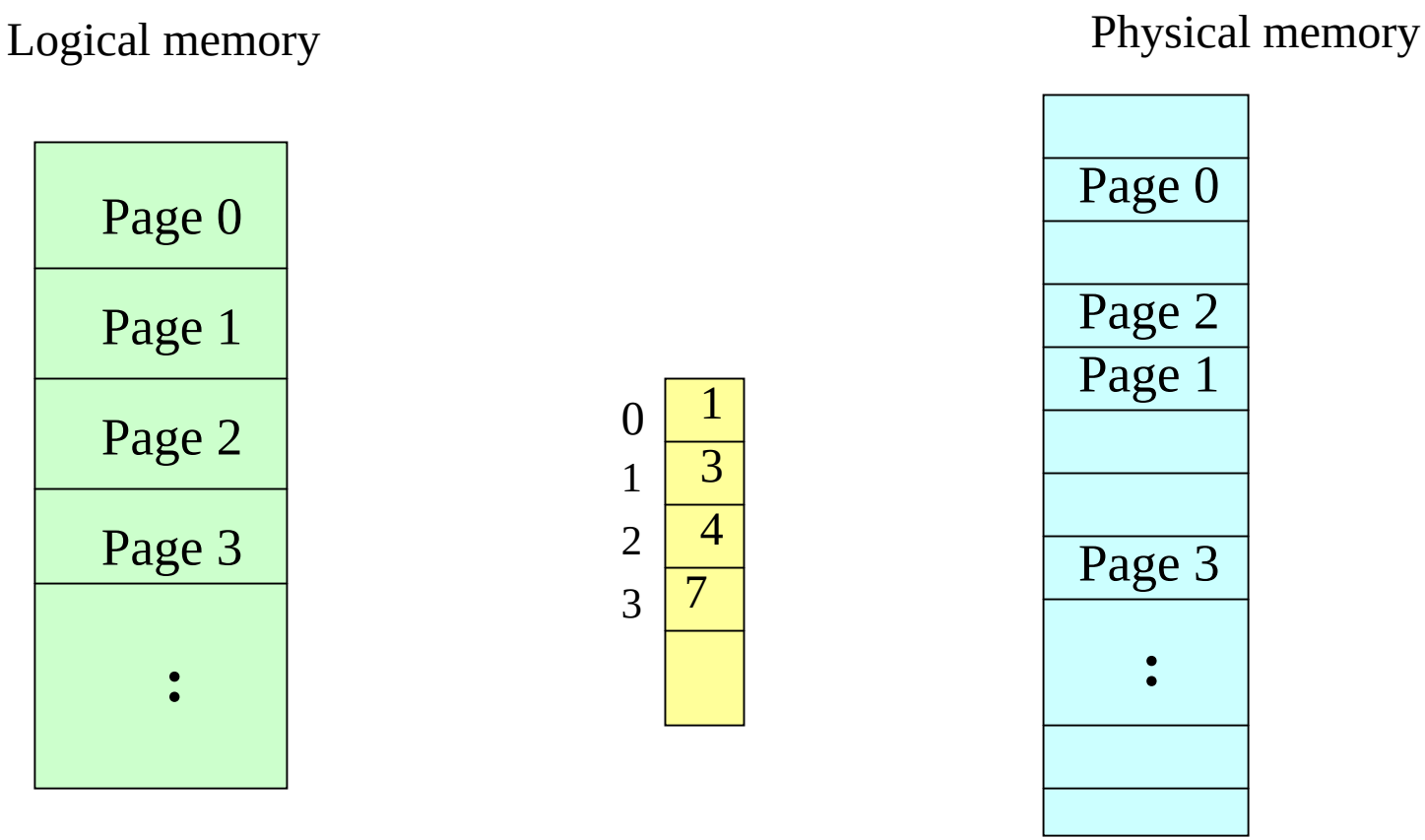
Address Translation Scheme

- Address generated by CPU is divided into:
  - Page number(p)
    - used as an index into page table which contains base address of each page in physical memory.
  - Page offset(d)
    - combined with base address to define the physical memory address that is sent to the memory unit.

Address Translation Architecture



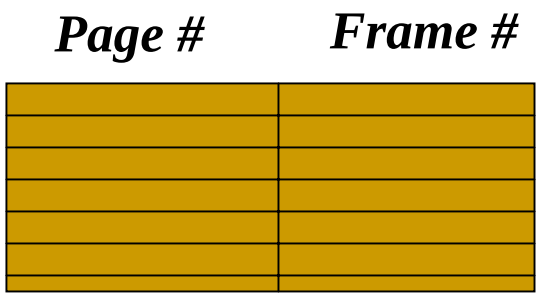
Example of Paging



Page Table Implementation

- Page table is kept in main memory
  - Page-table base register (PTBR) points to the page table.
  - Page-table length register (PTLR) indicates the size of page table.
- Every data/instruction access requires 2 memory accesses.
  - One for page table, one for data/instruction
  - Two-memory access problem solved by use of special fast-lookup hardware cache (i.e. cache page table in registers)
    - associative registers or translation look-aside buffers (TLBs)

Associative Registers



Address Translation
(A, A')

- If A is in associative register, get frame #
- Otherwise, need to go to page table for frame#
  - requires additional memory reference
  - Page Hit ratio - percentage of time page is found in associative memory.

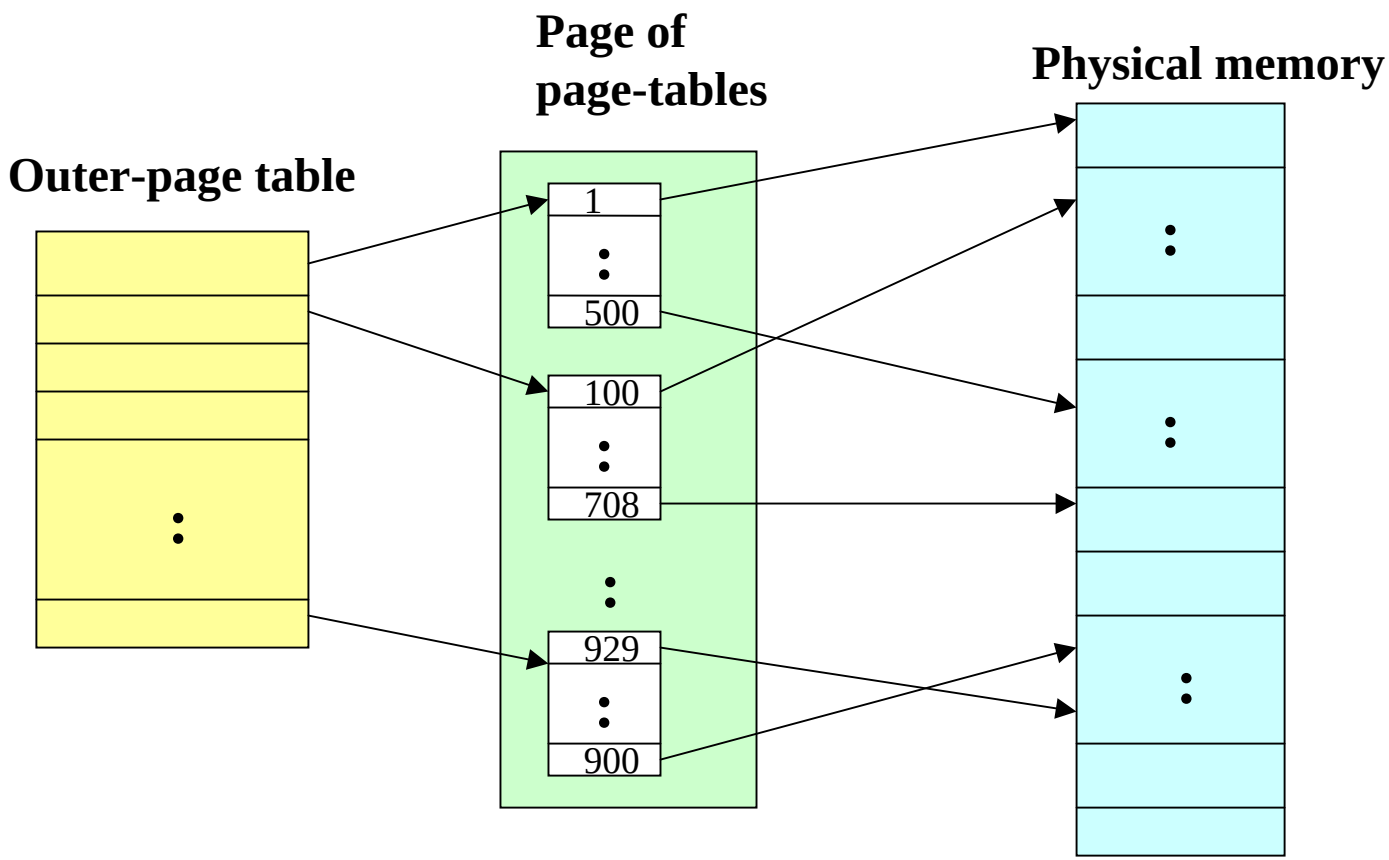
Effective Access time

- Associative lookup time = ε time unit
- Assume Memory cycle time = 1 microsecond
- Hit ratio = α
- Effective access time (EAT)
  - EAT = $(1+\varepsilon)\,\alpha + (2+\varepsilon)\,(1-\alpha)$
  - EAT = $2+\varepsilon-\alpha$

Memory Protection

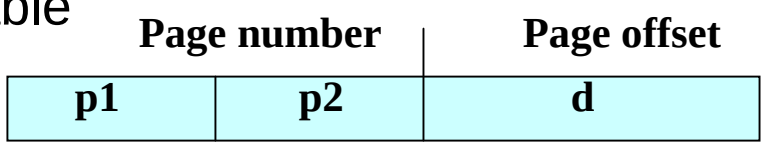
- Implemented by associating protection bits with each frame.
- Valid/invalid bit attached to each entry in page table.
  - Valid: indicates that the associated page is in the process' logical address space.
  - Invalid: indicates that the page is not in the process' logical address space.

Two Level Page Table Scheme



Two Level Paging Example

- A logical address (32bit machine, 4K page size) is divided into
  - a page number consisting of 20 bits, a page offset consisting of 12 bits
- Since the page table is paged, the page number consists of
  - a 10-bit page number, a 10-bit page offset
- Thus, a logical address is organized as (p1,p2,d) where
  - p1 is an index into the outer page table
  - p2 is the displacement within the page of the outer page table



Multilevel paging

- Each level is a separate table in memory
  - converting a logical address to a physical one may take 4 or more memory accesses.
  - Caching can help performance remain reasonable.
    - Assume cache hit rate is 98%, memory access time is quintupled (100 vs. 500 nanoseconds), cache lookup time is 20 nanoseconds
    - Effective Access time = 0.98 * 120 + .02 * 520 = 128 ns
    - This is only a 28% slowdown in memory access time...

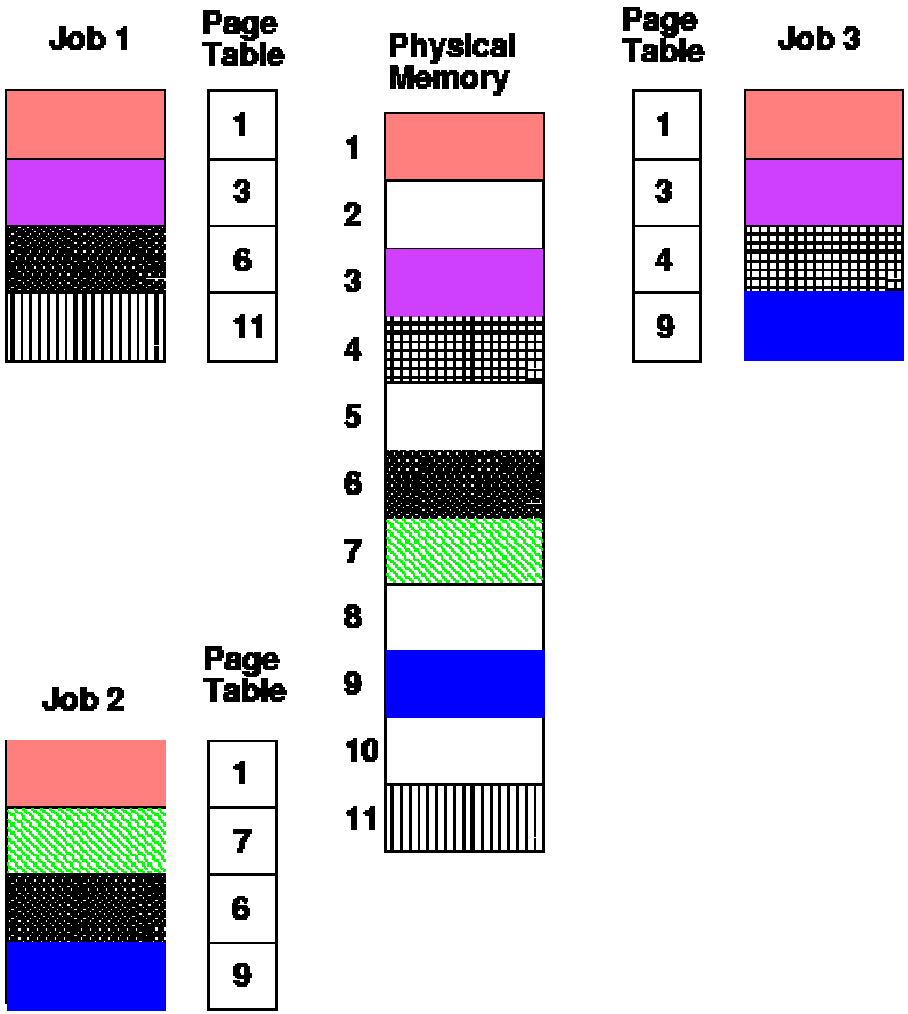
Inverted Page Table

- One entry for each real page of memory
  - Entry consists of virtual address of page in real memory with information about process that owns page.
  - Decreases memory needed to store page table
  - Increases time to search table when a page reference occurs
    - table sorted by physical address, lookup by virtual address
  - Use hash table to limit search to one (maybe few) page-table entries.

Shared pages

- Code and data can be shared among processes
  - Reentrant (non self-modifying) code can be shared.
  - Map them into pages with common page frame mappings
  - Single copy of read-only code - compilers, editors etc..
- Shared code must appear in the same location in the logical address space of all processes
- Private code and data
  - Each process keeps a separate copy of code and data
  - Pages for private code and data can appear anywhere in logical address space.

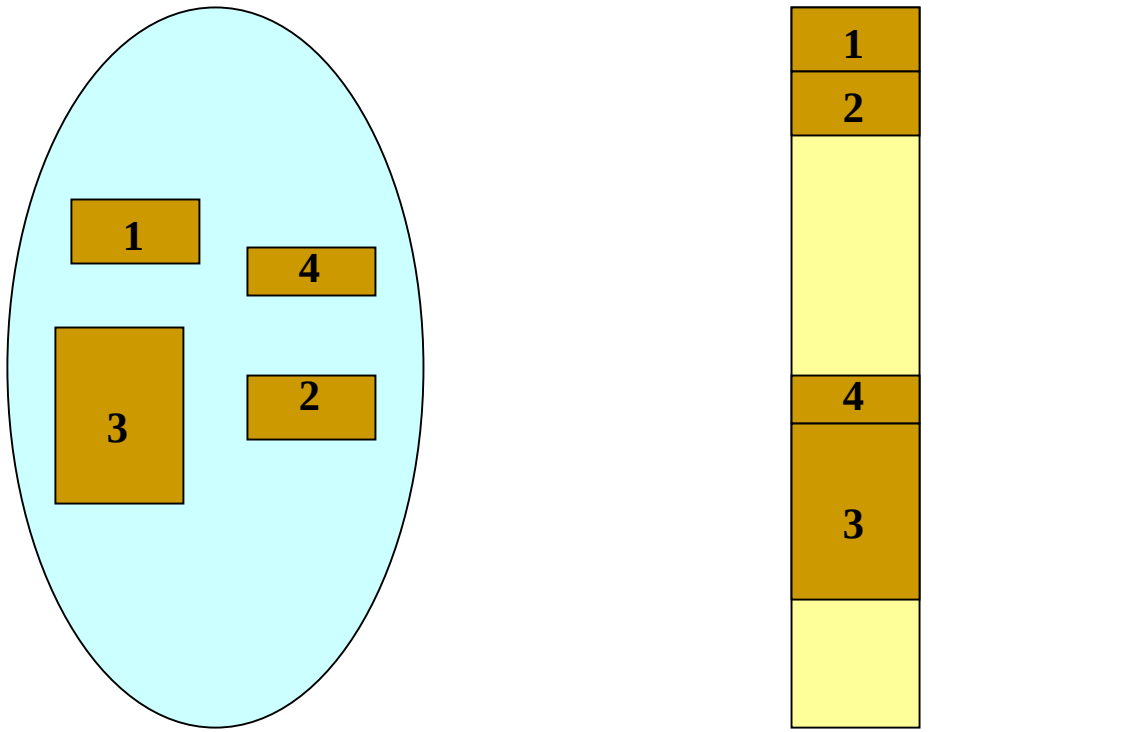
Shared Pages



Segmentation

- Memory Management Scheme that supports user view of memory.
- A program is a collection of segments.
- A segment is a logical unit such as
  - main program, procedure, function
  - local variables, global variables,common block
  - stack, symbol table, arrays
  - Protect each entity independently
  - Allow each segment to grow independently
  - Share each segment independently

Logical view of segmentation



Segmentation Architecture

- Logical address consists of a two tuple

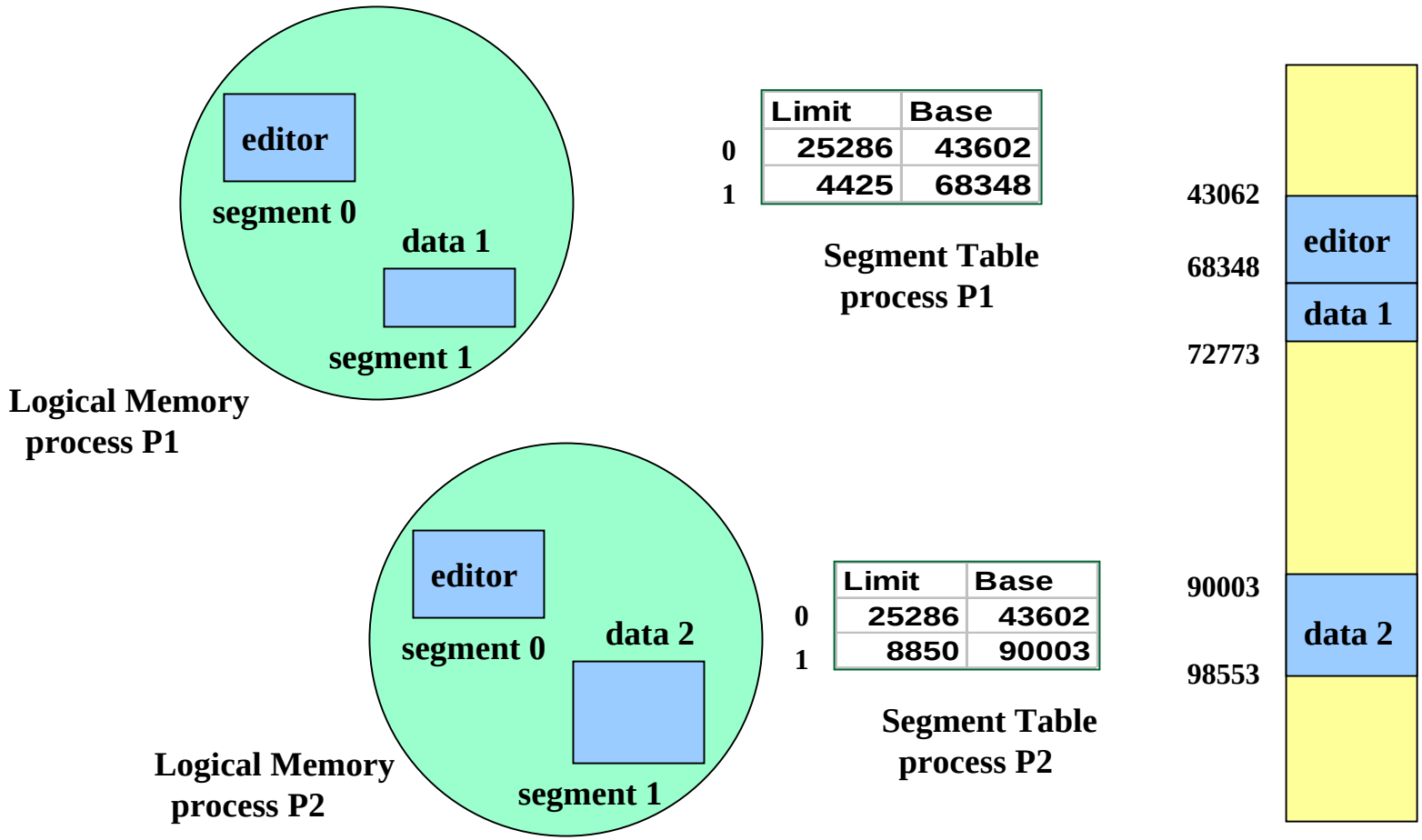
  <segment-number, offset>

- Segment Table
  - Maps two-dimensional user-defined addresses into one-dimensional physical addresses. Each table entry has
    - Base - contains the starting physical address where the segments reside in memory.
    - Limit - specifies the length of the segment.
  - *Segment-table base register* (STBR) points to the segment table's location in memory.
  - *Segment-table length register* (STLR) indicates the number of segments used by a program; segment number is legal if s < STLR.

Segmentation Architecture (cont.)

- Relocation is dynamic - by segment table
- Sharing
  - Code sharing occurs at the segment level.
  - Shared segments must have same segment number.
- Allocation - dynamic storage allocation problem
  - use best fit/first fit, may cause external fragmentation.
- Protection
  - protection bits associated with segments
    - read/write/execute privileges
    - array in a separate segment - hardware can check for illegal array indexes.

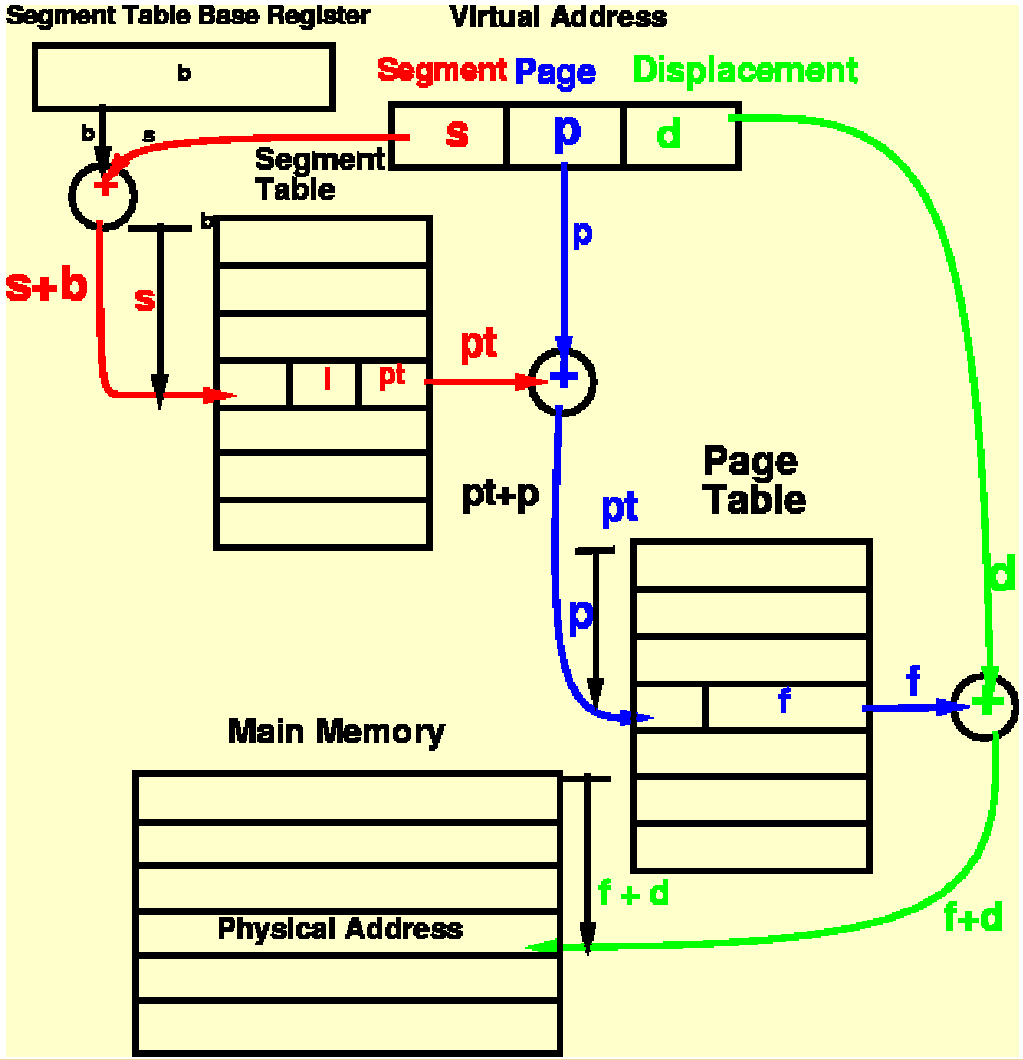
Shared segments



Segmented Paged Memory

- Segment-table entry contains not the base address of the segment, but the base address of a page table for this segment.
  - Overcomes external fragmentation problem of segmented memory.
  - Paging also makes allocation simpler; time to search for a suitable segment (using best-fit etc.) reduced.
  - Introduces some internal fragmentation and table space overhead.
- Multics - single level page table
- IBM OS/2 - OS on top of Intel 386
  - uses a two level paging scheme

MULTICS address translation scheme



Virtual Memory

- Background
- Demand paging
  - Performance of demand paging
- Page Replacement
  - Page Replacement Algorithms
- Allocation of Frames
- Thrashing
- Demand Segmentation

Need for Virtual Memory

- 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.
  - Need to allow pages to be swapped in and out.
- Virtual Memory can be implemented via
  - Paging
  - Segmentation

Paging/Segmentation Policies

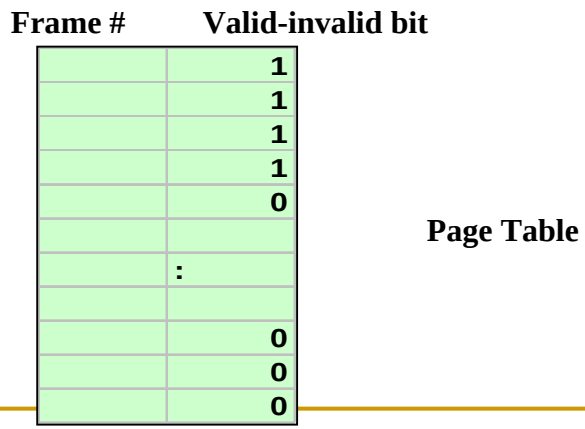
- Fetch Strategies
  - When should a page or segment be brought into primary memory from secondary (disk) storage?
    - Demand Fetch
    - Anticipatory Fetch
- Placement Strategies
  - When a page or segment is brought into memory, where is it to be put?
    - Paging - trivial
    - Segmentation - significant problem
- Replacement Strategies
  - Which page/segment should be replaced if there is not enough room for a required page/segment?

Demand Paging

- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less Memory needed
  - Faster response
  - More users
- The first reference to a page will trap to OS with a page fault.
- OS looks at another table to decide
  - Invalid reference - abort
  - Just not in memory.

Valid-Invalid Bit

- With each page table entry a valid-invalid bit is associated (1 ⇒ in-memory, 0 ⇒ not in memory).
- Initially, valid-invalid bit is set to 0 on all entries.
  - During address translation, if valid-invalid bit in page table entry is 0 --- ***page fault*** occurs.
  - Example of a page-table snapshot



Handling a Page Fault

- Page is needed - reference to page
  - Step 1: Page fault occurs - trap to OS (process suspends).
  - Step 2: Check if the virtual memory address is valid. Kill job if invalid reference. If valid reference, and page not in memory, continue.
  - Step 3: Bring into memory - Find a free page frame, map address to disk block and fetch disk block into page frame. When disk read has completed, add virtual memory mapping to indicate that page is in memory.
  - Step 4: Restart instruction interrupted by illegal address trap. The process will continue as if page had always been in memory.

What happens if there is no free frame?

- Page replacement - find some page in memory that is not really in use and swap it.
  - Need page replacement algorithm
  - Performance Issue - need an algorithm which will result in minimum number of page faults.
  - Same page may be brought into memory many times.

Performance of Demand Paging

- Page Fault Ratio - $0 \leq p \leq 1.0$
  - If $p = 0$, no page faults
  - If $p = 1$, every reference is a page fault
- Effective Access Time

  EAT = (1-p) * memory-access +
  p * (page fault overhead +
  swap page out +
  swap page in +
  restart overhead)

Demand Paging Example

- Memory Access time = 1 microsecond
- 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out.
- Swap Page Time = 10 msec = 10,000 microsec

  EAT = (1-p) *1 + p (15000) ≈1 + 15000p microsec
- EAT is directly proportional to the page fault rate.

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
  - large virtual memory can be provided on a smaller physical memory.

Page Replacement Algorithms

- Want lowest page-fault rate.
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.
- Assume reference string in examples to follow is

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

Page Replacement Strategies

- The Principle of Optimality
  - Replace the page that will not be used again the farthest time into the future.
- Random Page Replacement
  - Choose a page randomly
- FIFO - First in First Out
  - Replace the page that has been in memory the longest.
- LRU - Least Recently Used
  - Replace the page that has not been used for the longest time.
- LFU - Least Frequently Used
  - Replace the page that is used least often.
- NUR - Not Used Recently
  - An approximation to LRU
- Working Set
  - Keep in memory those pages that the process is actively using

First-In-First-Out (FIFO) Algorithm

Reference String: 1,2,3,4,1,2,5,1,2,3,4,5

- Assume x frames (x pages can be in memory at a time per process)

3 frames

Frame 1	1	4	5
Frame 2	2	1	3
Frame 3	3	2	4

9 Page faults

4 frames

Frame 1	1	5	4
Frame 2	2	1	5
Frame 3	3	2	
Frame 4	4	3	

10 Page faults

FIFO Replacement - *Belady's Anomaly* -- more frames does not mean less page faults

Optimal Algorithm

- Replace page that will not be used for longest period of time.
  - How do you know this???
  - Generally used to measure how well an algorithm performs.

4 frames

Frame 1	**1**	**4**
Frame 2	**2**	
Frame 3	**3**	
Frame 4	**4**	**5**

6 Page faults

Least Recently Used (LRU) Algorithm

- Use recent past as an approximation of near future.
- Choose the page that has not been used for the longest period of time.
- May require hardware assistance to implement.
- Reference String: 1,2,3,4,1,2,5,1,2,3,4,5

4 frames

Frame 1	**1**		**5**
Frame 2	**2**		
Frame 3	**3**	**5**	**4**
Frame 4	**4**	**3**	

8 Page faults

Implementation of 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 changes, look at the counters to determine which page to change (page with smallest time value).
- Stack Implementation
  - Keeps a stack of page number in a doubly linked form
  - Page referenced
    - move it to the top
    - required 6 pointers to be changed
  - No search required for replacement

LRU Approximation Algorithms

- Reference Bit
  - With each page, associate a bit, initially = 0.
  - When page is referenced, bit is set to 1.
  - Replace the one which is 0 (if one exists). Do not know order however.
- Additional Reference Bits Algorithm
  - Record reference bits at regular intervals.
  - Keep 8 bits (say) for each page in a table in memory.
  - Periodically, shift reference bit into high-order bit, I.e. shift other bits to the right, dropping the lowest bit.
  - During page replacement, interpret 8bits as unsigned integer.
  - The page with the lowest number is the LRU page.

LRU Approximation Algorithms

- Second Chance
  - FIFO (clock) replacement algorithm
  - Need a reference bit.
  - When a page is selected, inspect the reference bit.
  - If the reference bit = 0, replace the page.
  - If page to be replaced (in clock order) has reference bit = 1, then
    - set reference bit to 0
    - leave page in memory
    - replace next page (in clock order) subject to same rules.

LRU Approximation Algorithms

- Enhanced Second Chance
  - Need a reference bit and a modify bit as an ordered pair.
  - 4 situations are possible:
    - (0,0) - neither recently used nor modified - best page to replace.
    - (0,1) - not recently used, but modified - not quite as good, because the page will need to be written out before replacement.
    - (1,0) - recently used but clean - probably will be used again soon.
    - (1,1) - probably will be used again, will need to write out before replacement.
    - Used in the Macintosh virtual memory management scheme

Counting Algorithms

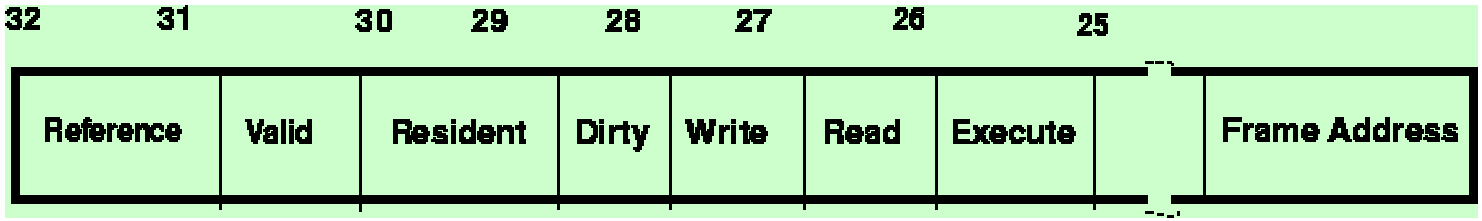
- Keep a counter of the number of references that have been made to each page.
  - LFU (least frequently used) algorithm
    - replaces page with smallest count.
    - Rationale : frequently used page should have a large reference count.
      - Variation - shift bits right, exponentially decaying count.
  - MFU (most frequently used) algorithm
    - replaces page with highest count.
    - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Page Buffering Algorithm

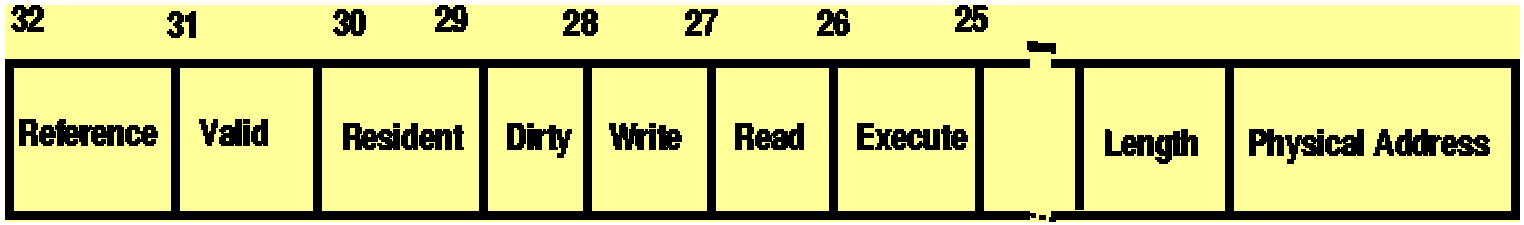
- Keep pool of free frames
  - Solution 1
    - When a page fault occurs, choose victim frame.
    - Desired page is read into free frame from pool before victim is written out.
    - Allows process to restart soon, victim is later written out and added to free frame pool.
  - Solution 2
    - Maintain a list of modified pages. When paging device is idle, write modified pages to disk and clear modify bit.
  - Solution 3
    - Keep frame contents in pool of free frames and remember which page was in frame.. If desired page is in free frame pool, no need to page in.

Protection Bits

Page Protection



Segmentation Protection



Reference - *Page has been accessed*
Valid - *Page exists*
Resident - *Page is cached in primary memory*
Dirty - *Page has been changed since page in*

Allocation of Frames

- Single user case is simple
  - User is allocated any free frame
- Problem: Demand paging + multiprogramming
  - Each process needs minimum number of pages based on instruction set architecture.
  - Example IBM 370: 6 pages to handle MVC (storage to storage move) instruction
    - Instruction is 6 bytes, might span 2 pages.
    - 2 pages to handle ***from***.
    - 2 pages to handle ***to***.
  - Two major allocation schemes:
    - Fixed allocation
    - Priority allocation

Fixed Allocation

- Equal Allocation
  - E.g. If 100 frames and 5 processes, give each 20 pages.
- Proportional Allocation
  - Allocate according to the size of process
    - Sj = size of process Pj
    - $S = \sum Sj$
    - m = total number of frames
    - aj = allocation for $Pj = Sj/S * m$
    - If $m = 64$, $S1 = 10$, $S2 = 127$ then

      $a1 = 10/137 * 64 \approx 5$

      $a2 = 127/137 * 64 \approx 59$

Priority Allocation

- May want to give high priority process more memory than low priority process.
- Use a proportional allocation scheme using priorities instead of size
- If process Pi generates a page fault
  - select for replacement one of its frames
  - select for replacement a frame form a process with lower priority number.

Global vs. Local Allocation

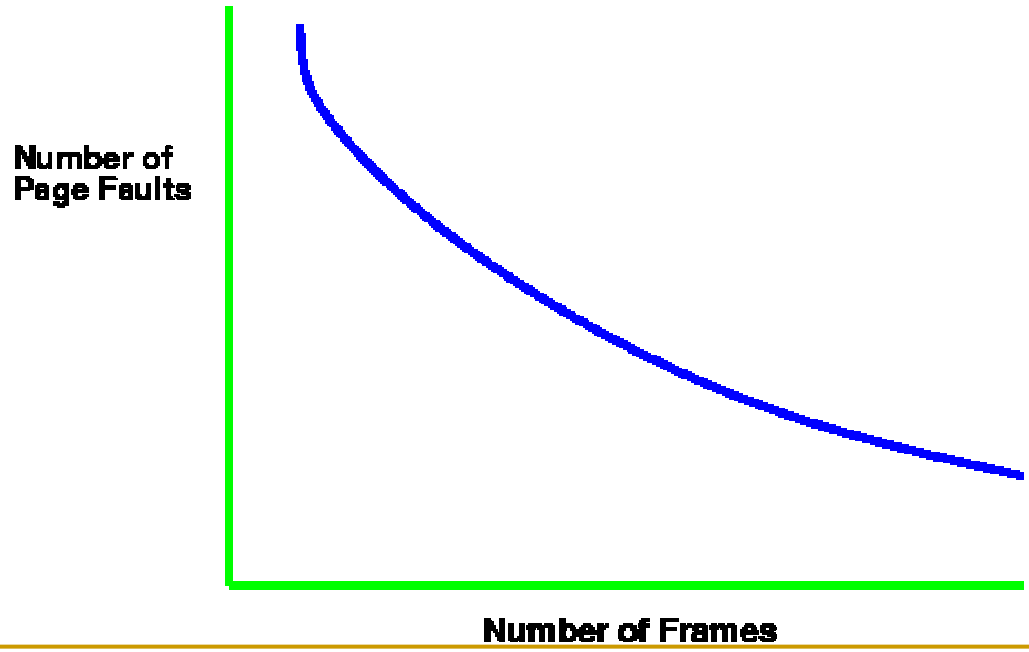
- Global Replacement
  - Process selects a replacement frame from the set of all frames.
  - One process can take a frame from another.
  - Process may not be able to control its page fault rate.
- Local Replacement
  - Each process selects from only its own set of allocated frames.
  - Process slowed down even if other less used pages of memory are available.
- Global replacement has better throughput
  - Hence more commonly used.

Thrashing

- If a process does not have enough pages, the page-fault rate is very high. This leads to:
  - low CPU utilization.
  - OS thinks that it needs to increase the degree of multiprogramming
  - Another process is added to the system.
  - System throughput plunges...
- *Thrashing*
  - A process is busy swapping pages in and out.
  - In other words, a process is spending more time paging than executing.

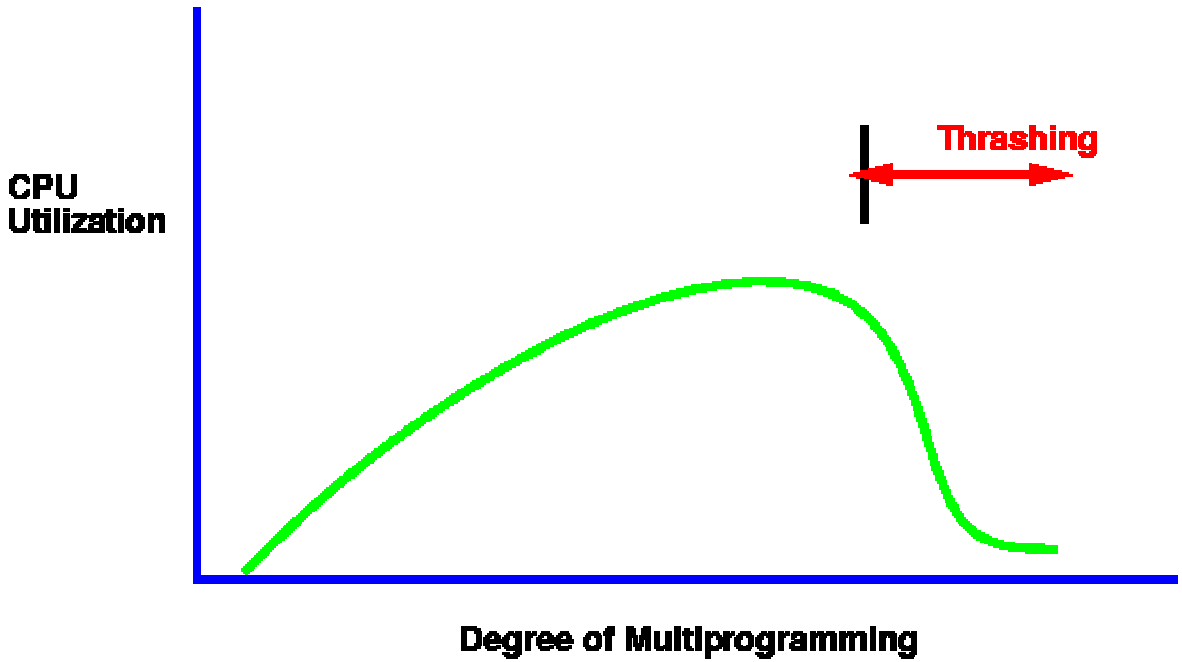
Thrashing (cont.)

- Why does paging work?
  - Locality Model - computations have locality!
  - Locality - set of pages that are actively used together.
  - Process migrates from one locality to another.
  - Localities may overlap.



Thrashing

- Why does thrashing occur?
  - Σ (size of locality) > total memory size



Working Set Model

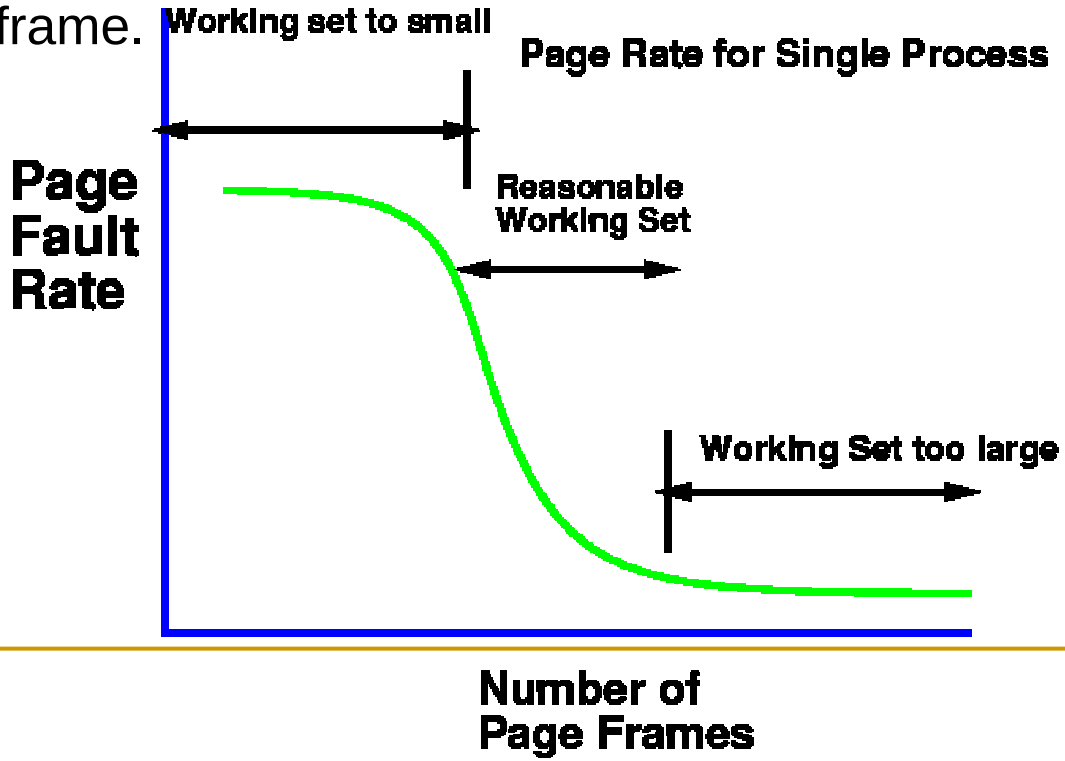
- $\Delta \equiv$ working-set window
  - a fixed number of page references, e.g. 10,000 instructions
  - WSSj (working set size of process Pj) - total number of pages referenced in the most recent Δ (varies in time)
    - If Δ too small, will not encompass entire locality.
    - If Δ too large, will encompass several localities.
    - If $\Delta = \infty$, will encompass entire program.
  - $D = \sum$ WSSj $\equiv$ total demand frames
    - If D > m (number of available frames) $\Rightarrow$thrashing
  - Policy: If D > m, then suspend one of the processes.

Keeping Track of the Working Set

- Approximate with
  - interval timer + a reference bit
  - Example: Δ = 10,000
    - Timer interrupts after every 5000 time units.
    - Whenever a timer interrupts, copy and set the values of all reference bits to 0.
    - Keep in memory 2 bits for each page (indicated if page was used within last 10,000 to 15,000 references).
    - If one of the bits in memory = 1 $\Rightarrow$ page in working set.
  - Not completely accurate - cannot tell where reference occurred.
  - Improvement - 10 bits and interrupt every 1000 time units.

Page fault Frequency Scheme

- Control thrashing by establishing *acceptable* page-fault rate.
  - If page fault rate too low, process loses frame.
  - If page fault rate too high, process needs and gains a frame.



Demand Paging Issues

- Prepaging
  - Tries to prevent high level of initial paging.
    - E.g. If a process is suspended, keep list of pages in working set and bring entire working set back before restarting process.
    - Tradeoff - page fault vs. prepaging - depends on how many pages brought back are reused.
- Page Size Selection
  - fragmentation
  - table size
  - I/O overhead
  - locality

Demand Paging Issues

- Program Structure
  - Array A[1024,1024] of integer
  - Assume each row is stored on one page
  - Assume only one frame in memory
  - Program 1

```
for j := 1 to 1024 do
for i := 1 to 1024 do
        A[i,j] := 0;
```

1024 * 1024 page faults

  - Program 2

```
for i := 1 to 1024 do
for j:= 1 to 1024 do
        A[i,j] := 0;
```

1024 page faults

Demand Paging Issues

- I/O Interlock and addressing
  - Say I/O is done to/from virtual memory. I/O is implemented by I/O controller.
    - Process A issues I/O request
    - CPU is given to other processes
    - Page faults occur - process A's pages are paged out.
    - I/O now tries to occur - but frame is being used for another process.
  - Solution 1: never execute I/O to memory - I/O takes place into system memory. Copying Overhead!!
  - Solution 2: Lock pages in memory - cannot be selected for replacement.

Demand Segmentation

- Used when there is insufficient hardware to implement demand paging.
- OS/2 allocates memory in segments, which it keeps track of through segment descriptors.
  - Segment descriptor contains valid bit to indicate whether the segment is currently in memory.
    - If segment is in main memory, access continues.
    - If not in memory, segment fault.