Memory Management

Dickinson College
Computer Science 354

slides courtesy of Professor Grant Braught

Road Map

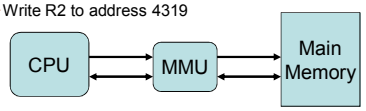
- Past:
  - ✓ What an OS is, why we have them, what they do.
  - ✓ Base hardware and support for operating systems
  - ✓ Process Management
  - ✓ Process Scheduling
  - ✓ Multi-Threading
  - ✓ Thread Synchronization
  - ✓ File Systems
- Present:
  - ✓ Memory management
- Future:
  - ✓ Protection and Security

Memory Management Outline

- With respect to memory management, we'll examine the following topics:
  - ✓ Basic hardware capabilities
    - ❖ Logical vs. Physical addresses
    - ❖ Absolute vs. relative programs
    - ❖ Address binding
  - ✓ Multiprogramming and Memory
  - ✓ Virtual Memory

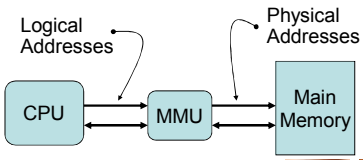
Basic Hardware Capabilities

- Assumptions about basic computer hardware:
  - ✓ For instructions to be executed, or data to be accessed, they must be contained in main memory.
  - ✓ Program execution generates a stream of memory references that are sent from the CPU to the *memory management unit*, which controls access to the main memory.
    - ❖ Load address 1000 into R3
    - ❖ Write R2 to address 4319



Logical vs. Physical Addresses

- The addresses issued by a program to the MMU are called *logical addresses*.
- Addresses issued by the MMU to the main memory are called *physical addresses*.



Absolute vs. Relative Programs

- In an *absolute program* the memory addresses contained in the program correspond directly to physical memory addresses
  - ✓ logical addresses = physical addresses
    - ❖ Very intuitive and easy to implement.
    - ❖ Drawbacks?
- In a *relative program* the memory addresses contained in the program are relative to the start of the program.

Address Binding

- Address binding is the process of translating logical addresses to physical addresses.
  - ✓Address binding techniques:
    - ❖Compile time
    - ❖Load time
    - ❖Run time

Compile Time Address Binding

- With compile time address binding, the compiler produces an absolute program.
  - ✓The base address at which the executable program will be loaded must be provided to the compiler.
  - ✓The resulting program must always be loaded at the same location in physical memory.
    - ❖MS-DOS .COM format.

Load Time Address Binding

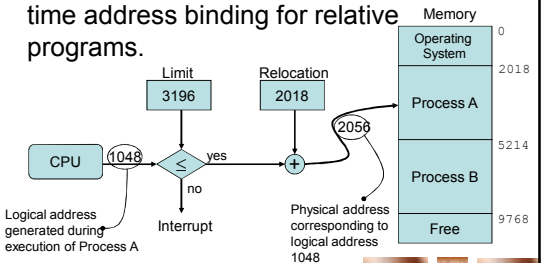
- With load time address binding, a special program called a *loader* produces an absolute program as a program is loaded into physical memory.
  - ✓The loader translates all logical addresses within the program to physical addresses.

Run Time Address Binding

- With run time address binding, logical addresses are translated to physical address as the program executes.
  - ✓The program issues relative addresses to the MMU which translates them to physical addresses.

Relocation Register

- A relocation register provides a simple means of performing run time address binding for relative programs.



Note: Assume CPU is executing Process A

Tradeoffs

- What tradeoffs exist between the different address binding schemes?

Compile time:
Do it once -loads fast / runs fast.
Can't relocate program.
Load time:
Have to translate every time the program is loaded which will cause programs to load slowly.
Once loaded it runs fast.
Run time:
Programs load fast.
Translation during run time slows down execution.

Unix / Linux Bashing

- "Linux is only free if your time is worthless."
- "Unix is user-friendly. It's just very selective about who its friends are."

Multiprogramming and Memory

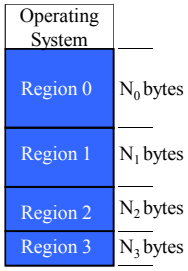
- With multiprogramming, the OS shares the physical memory among multiple processes that are running concurrently.
  - The OS can allocate memory to processes using two types of schemes:
    - Contiguous Allocation Schemes
    - Non-contiguous memory allocation schemes

Contiguous Allocation Schemes

- With contiguous allocation each process is stored as a whole in a contiguous range of physical memory addresses.
  - Approaches:
    - Fixed size partitions
    - Variable size partitions

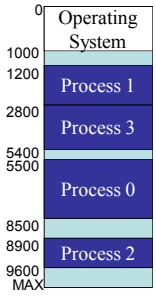
Fixed Size Partitions

- The OS divides physical memory into a number of fixed size regions.
  - When a process is to be loaded into memory, the OS loads it into one of these regions.
    - **Best Fit**: The process is loaded into the smallest free region that satisfies its space requirements.



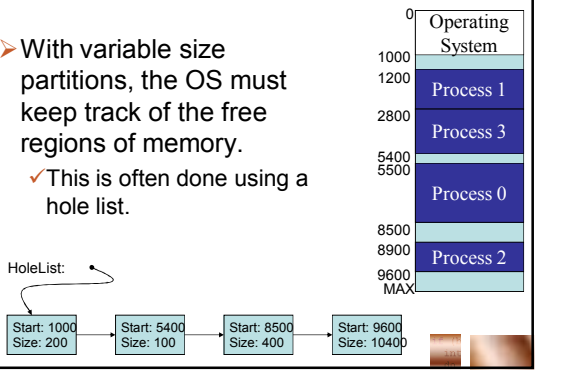
Variable Size Partitions

- The OS allocates variable sized regions of memory based on the size requirements of each process.
  - Each time a process is to be loaded into memory a free region of sufficient size must be found.



Variable Size Partitions

- With variable size partitions, the OS must keep track of the free regions of memory.
  - This is often done using a hole list.



Allocation Strategies

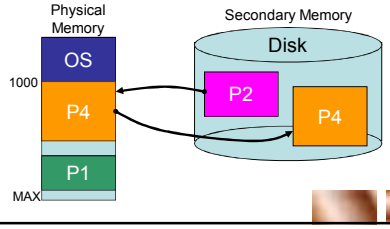
- Allocating space using variable size partitions requires that the OS have a policy for selecting the hole for a process:
  - ✓ **Best Fit**: Select the smallest hole that satisfies the request.
  - ✓ **Worst Fit**: Select largest hole that satisfies the request.
  - ✓ **First Fit**: Select the first hole that satisfies the request.
    - ❖ Always start checking holes from the start of the list.
  - ✓ **Next Fit**: Select the next hole that satisfies the request.
    - ❖ Use a pointer to keep track of the last hole that was checked, and continue checking holes from that point for the next request.

Dealing with Fragmentation

- With variable size partitions, external fragmentation can become a significant issue.
  - ✓ Mitigating external fragmentation:
    - ❖ Hole compaction
    - ❖ Memory compaction

Swapping

- Partitioning can be combined with swapping to further increase the number of programs available for multiprogramming.

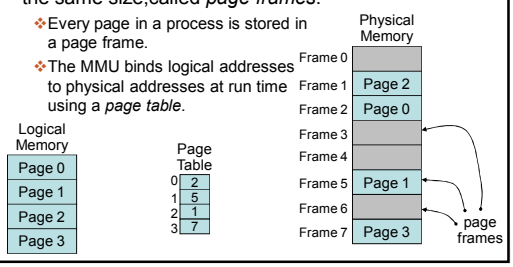


Non-Contiguous Allocation

- With non-contiguous allocation, a process can be divided into pieces (*pages* or *segments*), each of which is stored in a different area of physical memory.
  - ✓ Approaches:
    - ❖ Paging
    - ❖ Segmentation

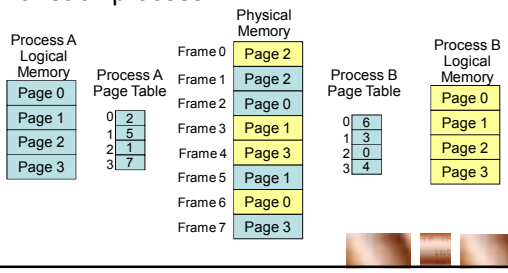
Paging

- In Paging:
  - ✓ Processes are divided into fixed size chunks called *pages.*
  - ✓ Physical memory is divided into fixed size areas, of the same size,called *page frames*.
    - ❖ Every page in a process is stored in a page frame.
    - ❖ The MMU binds logical addresses to physical addresses at run time using a *page table*.



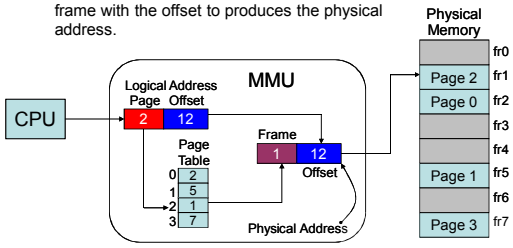
Paging

- To share physical memory among multiple processes, the OS maintains a page table for each process.



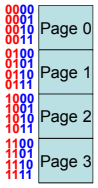
Address Binding With Paging

- We can think of logical addresses as being divided into two parts, the *page* and the *offset* within the page.
  - The MMU uses the page table to translate the page into the page frame. Combining the page frame with the offset to produces the physical address.



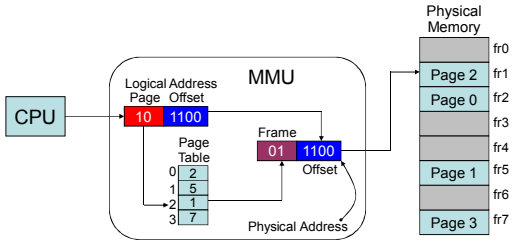
Address Binding with Paging

- In a paging system the logical address space is viewed as a contiguous array of bytes.
  - Logical addresses are actually specified in binary.
    - We must now think of the high order bits as specifying the **page** and the low order bits as specifying the **offset** within the page.
  - Example:
    - Logical address space with 4 pages of 4 bytes each.
      - 2 bits needed to specify the page.
      - 2 bits needed to specify the offset.
    - E.g. Address 1101 (13_{10}) is at offset 1 in page 3.



Address Binding with Paging

- The example from earlier, now in binary.



Operating System Role in Paging

- Address binding with paging is performed in hardware by the MMU. However, the OS must:
  - Keep track of free frames
  - Allocate frames to processes
    - Create page tables for each process
  - Swap the page table during each context switch
  - Perform manual address translations

Operating System Role in Paging

- With respect to memory allocation and paging, what must the OS do when a new process is created?
  - Look in the exe file to determine the size of the process (it can do so because the exe file is a structured file that indicates the size of the code, the global variables, the heap and the stack.)
  - Figure out how many pages are needed to hold the whole process.
  - See if a sufficient number of free frames are available.
  - Allocate the frames to the process
  - Load the pages into the frames from disk.
  - Fill in the process' page table.
  - Mark the frames as used in bit vector or frame table.

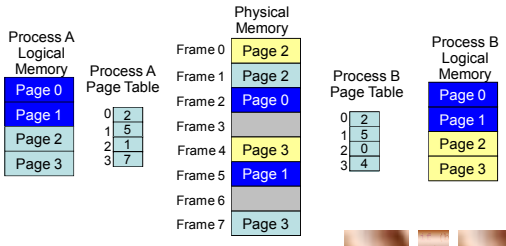
Memory Protection with Paging

- The traditional base/limit register approach to memory protection does not work with non-contiguous allocation.
  - For memory protection with paging, the page table can be augmented with bits to indicate:
    - Validity of page
    - Read-Only / Execute
  - A logical address that references an invalid page or accesses it in a way that is not permitted results in a trap to the OS.

	Frame	V	RO	X	
0	1011	1	1	1	Code
1	1010	1	1	1	
2	0001	1	1	1	
3	0111	1	0	0	Heap
4	0000	0	0	0	
5	0000	0	0	0	
6	0000	0	0	0	
7	1100	1	0	0	Stack

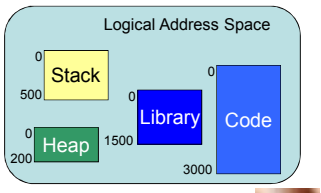
Shared Memory with Paging

- With paging, shared memory can be implemented by including the same frame in the page table of multiple processes.



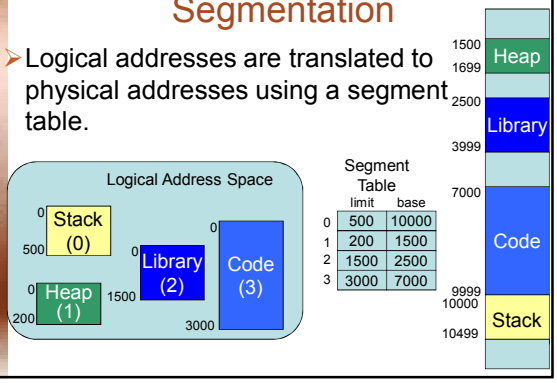
Segmentation

- With segmentation the logical address space is viewed, not as a contiguous array of bytes, but as a collection of *segments*.
  - Logical addresses are specified using a *segment* and an *offset* within the segment.

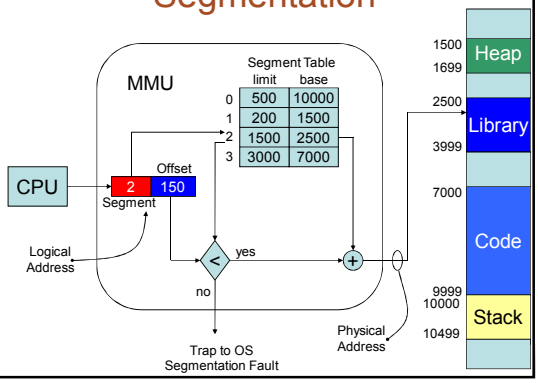


Address Binding with Segmentation

- Logical addresses are translated to physical addresses using a segment table.



Address Binding with Segmentation



Implementing Segmentation

- In a system that uses segmentation, there is close cooperation between the hardware, the compiler/assembler and the operating system.

Memory Management Problems

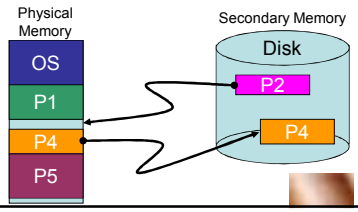
- Memory management, as we've studied it so far faces two significant problems:
  - How to allow significantly more programs to execute concurrently in the same amount of physical memory?
  - How to allow for the execution of programs that exceed the size of the physical memory (RAM)?
- Ultimate solution: *virtual memory*

Approaches to the Problems

- There are several approaches that address either one or both of the problems targeted by virtual memory:
  - ✓Approaches:
    - ❖Swapping
    - ❖Overlays
    - ❖Dynamic Link Libraries
    - ❖Virtual Memory

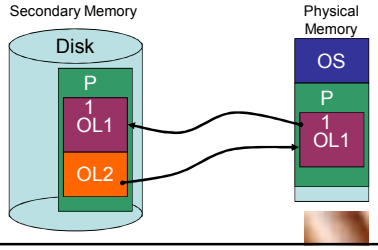
Swapping

- Swapping allows an increase in the number of concurrently executing processes by copying inactive or blocked processes to disk (*swapping them out*).
  - ✓This achieves the first benefit of VM but not the second.



Overlays

- Overlays allow programs to exceed the size of physical memory by enabling the programmer to identify relatively independent portions of a program that can be *overlaid* on top of one another in physical memory.



Dynamic Link Libraries

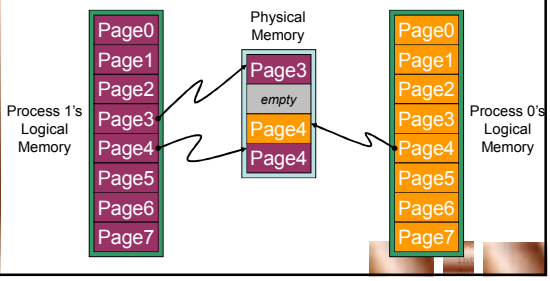
- Dynamic link libraries (DLLs) allow an increase in the number of concurrently executing programs by reducing the memory footprint of processes that use common code.
  - ✓Contrast with statically linked libraries.

Virtual Memory

- The idea behind virtual memory is to use physical memory to hold only the portions of each executing process that are currently being used.
  - ✓The portions of each executing process that are not currently being used are held on secondary storage until they are needed.

Virtual Memory

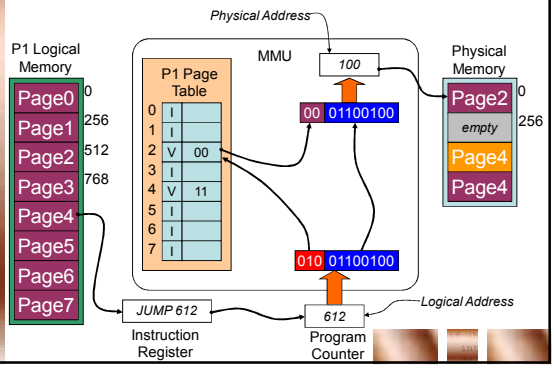
- Virtual memory is most often implemented using *paging*.



Address Binding with Paged VM

- In virtual memory with paging, address binding is done using a page table.
  - ✓The address translation is identical to the paging that we discussed earlier.
  - ✓Except:
    - ❖All pages that are not currently in physical memory are marked as invalid.
    - ❖Any logical address that references a page that is marked invalid causes the MMU to generate a trap to the OS.
      - This type of trap is called a *page fault*.

Address Binding with Paged VM



Page Faults

- When a logical address references a page that is not currently in physical memory a page fault occurs.
  - ✓What must the OS do to handle a page fault?
    - ❖Load the referenced page into physical memory.
    - ❖Update the page table to indicate where the new page has been loaded.
    - ❖Restart the instruction that generated the page fault.
    - ❖This time the instruction will succeed because the referenced page is in the physical memory.
    - ❖If there are no empty page frames:
      - The OS must select a page to be removed from physical memory.
      - Update the page table of the process whose page was removed from physical memory.
      - It then follows the above process as if there were an empty frame (because there is now!)

The Page Fault Penalty

- Because page faults require disk accesses the effective time for a memory reference can increase dramatically when VM is used.
  - ✓Sample system:
    - ❖Access to physical memory requires 1ns (10^{-9}sec)
    - ❖Reading a page from disk requires 10ms (10^{-2} sec)
    - ❖5% of all logical addresses result in a page fault.
    - ❖What is the effective memory access time on this system?

The Page Fault Penalty

- Being careful about the number of page frames available to each process (*frame allocation policy*) and the set of pages from each process that are held in those page frames (*page replacement policy*) can reduce the rate at which page faults occur.

Locality of Reference

- The design of frame allocation polices and page replacement polices is influenced by a property of programs called *locality*.
  - ✓**Spatial Locality**: If a program uses an instruction/datum, then instructions/data that are close by will tend to be used soon.
  - ✓**Temporal Locality**: If a program uses an instruction/datum then that instruction/datum will tend to be used again soon.
    - ❖**90/10 Rule**: Typical programs spend 90% of their time executing 10% of their instructions.

Typical Program Execution

Image scanned from Silberschatz, Galvin & Gagne.

Locality of Reference

- How does locality of reference relate to the selection of page replacement and frame allocation policies?
- Page Replacement:
  - If we use a data or instruction it is likely to be used again soon we shouldn't kick out pages that have been used recently.
  - Also, if a page hasn't been used recently, then it is unlikely to be used again so it is a good candidate to be removed.
- Frame Allocation:
  - Programs execute in one locality for a while and then shift to another locality for a while. So ideally we might try to ensure that there are enough frames allocated to a process to hold all of the pages that are part of the current locality.

Page Replacement Policy

- The page replacement policy dictates how a *victim frame* is selected when a page fault occurs and there are no empty page frames.
  - The page in the victim frame is removed from memory and is replaced by the page that caused the fault.
  - Possible Policies:
    - FIFO: First-in-first-out
    - LRU: Least recently used

Evaluating Page Replacement Policies

- The number of page faults serves as a good metric for evaluating page replacement policies.
  - Replacement policies are often compared by using *page reference strings* and counting the number of resulting page faults:
    - **Page Reference String**: A list of the pages referenced by a program.
      - Randomly generated
      - Program tracing
    - Example:
      1,1,2,3,3,3,4,4,1,3,3,4,2,2,5,5,1,2,2,3,4,4,4,5,5
      - Page reference strings can be compressed by removing duplicates. Why?

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

Simulating FIFO

- Assuming a frame allocation of 3 frames and the following page reference string, how many page faults occur with FIFO?

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

Optimal Page Replacement (OPT)

- Optimal Page Replacement:
  - Replace the page that will not be used for the longest period of time.
    - Guarantees the fewest number of page faults.
  - Problem?
  - Solution?

Least Recently Used (LRU)

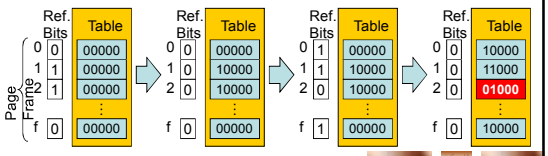
- The LRU replacement policy, replaces the page that has not been used in the longest period of time.
  - ✓Attempts to approximate OPT.
  - ✓Implementation issues:
    - ❖Requires hardware support to keep track of the time a page was last referenced.
      - Counter based approach.
    - ❖Requires a search for the frame containing the page with the earliest reference time.

Approximating LRU

- Most virtual memory hardware provides two features that are helpful for approximating LRU page replacement.
  - ✓Associated with each page frame are two bits:
    - ❖**Reference Bit**: this bit is set to 0 when a page is loaded into the frame and is flipped to 1 when the page is referenced.
    - ❖**Dirty Bit**: this bit is set to 0 when a page is loaded into the frame and is flipped to 1 when the page is modified.

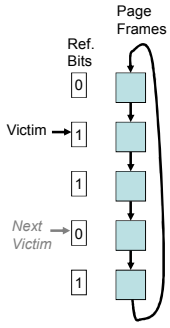
Shifting Implementation of LRU

- Strategy:
  - ✓Keep a table with a k bit number for each page frame.
  - ✓Periodically, shift the referenced bits for each page frame into the MSb of its associated k bit number and reset the referenced bits to 0.
  - ✓Select the page in the page frame with the smallest k bit number as the victim.



Second Chance LRU

- Strategy:
  - ✓View the page frames as a circular queue.
  - ✓Keep a pointer to frame containing the most recently replaced page.
  - ✓Traverse the reference bits:
    - ❖Zero each bit as it is visited.
    - ❖Replace first unreferenced page.



Enhanced Second Chance LRU

- Similar to second chance but considers both the referenced bit (r) and the dirty bit (d):
  - ✓Page frames are divided into categories based on the bit pair rd:
    - ❖00: neither recently used nor modified.
      - Best choice for replacement.
    - ❖01: not recently used but modified.
    - ❖10: recently used but not modified.
    - ❖11: recently used and modified.
      - Worst choice for replacement.
  - ✓Replace the page in the next frame in the lowest existing category.

Page Buffering

- Performance of a paging system can be improved by keeping a pool of unused page frames.
  - ✓On a page fault load the requested page into one of the unused page frames.
  - ✓Then select a victim page:
    - ❖ Write its contents back to disk (if necessary).
    - ❖ Add its frame to the pool of unused frames.
  - ✓An enhancement:
    - ❖Keep a record of what pages are in the pool of unused frames.

Page Frame Allocation Policy

- The page frame allocation policy determines the number of page frames that each process is allowed to use.
  - Allocation Policies:
    - Fixed Allocations:
      - Equal
      - Proportional
    - Dynamic Allocations:
      - Working set
      - Page fault frequency

Working Set Frame Allocation

- The working set policy attempts to allocate frames based on the size of a process' current locality.
  - Example:

Page Reference String

…2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3

w *w*

t_1 t_2

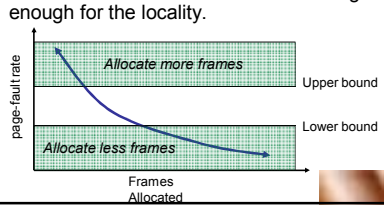
$WS(t_1)=\{2,6,1,5,7\}$ $WS(t_2)=\{3,4\}$

$|WS(t_1)| = \mathbf{5}$ $|WS(t2)| = \mathbf{2}$

$w \equiv$ working set window size

Page Fault Frequency Allocation

- The page fault frequency policy also attempts to allocate frames based on the size of a process' current locality.
  - However, it does not attempt to measure the locality size directly.
  - Instead it relies on the frequency of page faults to indicate if the allocation is either too large or not large enough for the locality.



Thrashing

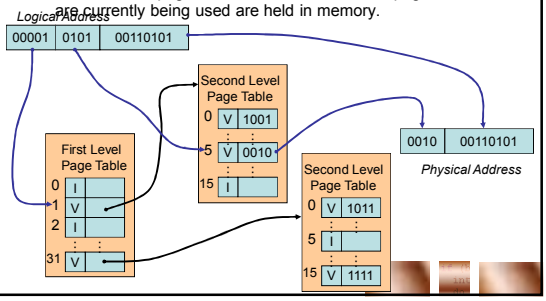
- If a process is not allocated a sufficient number of frames to hold its current locality it will *thrash*.
  - A process that is *thrashing*, spends more time waiting for page-faults than it spends processing.
    - How might an OS deal with thrashing?

Problems With Page Tables

- What issues might exist with the use of a page table for run-time address binding on a modern system?

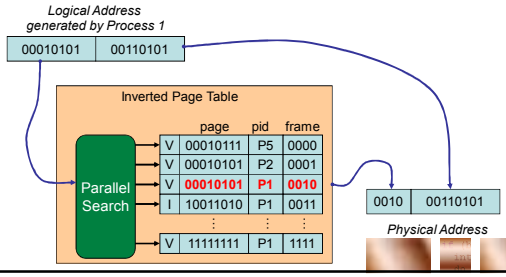
Multilevel Page Table

- A multilevel page table breaks the page table up into pages.
  - The first level page table and the second level page tables that are currently being used are held in memory.



Inverted Page Table

- An inverted page table keeps track of which page from which process is stored in each page frame.
  - ✓ Parallel search hardware is then used to search the inverted page table for the frame containing the referenced page of the current process.



Translation Look-Aside Buffer (TLB)