CS415 Module 7 Part A - How Does Virtual Memory Work?

Athens State University

Outline

Contents

Why Does Virtual Memory Work?

- Two fundamental characteristics of memory management
  1. All memory references are logical addresses that all 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
- These characteristics imply that it is not required for all pages or segments of a process to be in memory during execution

Important Definitions

Virtual memory A storage allocation scheme in which secondary memory can be addressed as though it is part of main memory

Virtual address The address assigned to a location in virtual memory to allow that location to be accessed as though it were part of main memory

Virtual address space The virtual storage assigned to a process

Address space The range of memory addresses available to a process

Real address The address of a storage location in main memory

Virtual Memory and Execution of a Process

- The OS brings only a few pieces of the program into main memory when a process starts
- **Resident set**: portion of process that is in main memory
- An interrupt is generated when an address that is needed is not in main memory
- The operating system schedules loading of this data using standard device operations
- Which means the OS places the process into a blocked state

For now, we can talk in general terms, and we will use the term piece to refer to either page or segment, depending on whether paging or segmentation is employed. Suppose that it is time to bring a new process into memory. The operating system begins by bringing in only one or a few pieces, to include the initial program piece and the initial data piece to which those instructions refer.

The portion of a process that is actually in main memory at any time is defined to be the resident set of the process. As the process executes, things proceed smoothly as long as all memory references are to locations that are in the resident set. Using the segment or page table, the processor always is able to determine whether this is so. If the processor encounters a logical address that is not in main memory, it generates an interrupt indicating a memory access fault. The operating system puts the interrupted process in a blocking state and takes control.

And then what happens is...

- The OS issues a disk I/O Read request
- Another process is dispatched to run while the disk I/O takes place
- An interrupt is issues when disk I/O is complete, which causes the OS to place the process that caused the page fault back onto the ready queue

For the execution of this process to proceed later, the operating system will need to bring into main memory the piece of the process that contains the logical address that caused the access fault. For this purpose, the operating system issues a disk I/O read request. After the I/O request has been issued, the operating system can dispatch another process to run while the disk I/O is performed. Once the desired piece has been brought into main memory, an I/O interrupt is issued, giving control back to the operating system, which places the affected process back into a Ready state.

Implications

- More processes may be maintained in main memory
  - Only load in needed portions of each process
  - With so many processes in main memory, it is highly likely that a process will be in the Ready state at any particular time
- A process may be larger than all of main memory

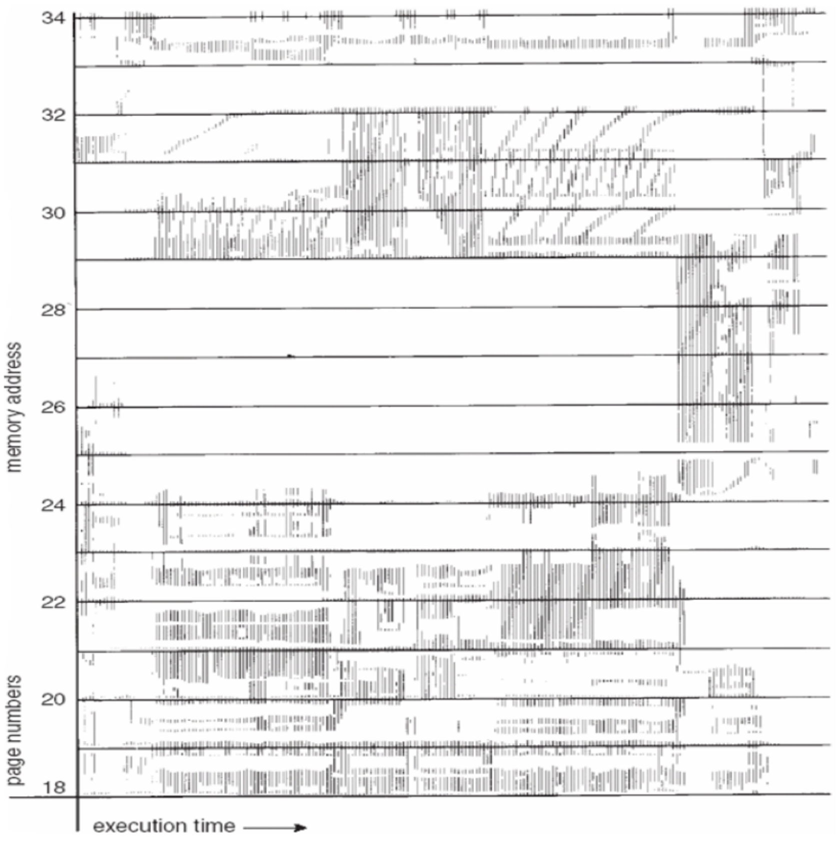
Thrashing and the Principle of Locality

- **Thrashing**: A state in which the system spends most of its time swapping process pieces rather than executing instructions
- Avoid thrashing by trying to guess, based on recent history, which pieces are least likely to be used in the near future

Principle of Locality

- Program and data references within a program tend to cluster
- Only a few pieces of a process will be needed over a short period of time
- Result is that we can make intelligent guesses about which pieces will be needed in the future

Locality In A Memory Reference Pattern



What do we need?

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 or segments between secondary memory and main memory

1 Policies for Virtual Memory

Paging

Determines when a page should be brought into memory

- Two main types:
  - Demand paging
  - Prepaging

Demand Paging

- Only bring pages into main memory when a reference is made to a location on the page
- Many page faults when process is first started
- Principle of locality implies that as more pages are brought into memory, the number of page faults should drop to a very low level

With demand paging, a page is brought into main memory only when a reference is made to a location on that page. If the other elements of memory management policy are good, the following should happen. When a process is first started, there will be a flurry of page faults. As more and more pages are brought in, the principle of locality suggests that most future references will be to pages that have recently been brought in. Thus, after a time, matters should settle down and the number of page faults should drop to a very low level.

Performance of Demand Paging

- Let
$$0 \leq p \leq 1$$
  be the page fault rate
  - At $p = 0$, no page faults
  - At $p = 1$, every reference is a fault
- Effective Access Time
  - Let E be the Effective Access Time
  - Let m be time required to get data from memory
  - Let f be page fault overhead time
  - Let s_o be time to swap page out of memory

  - Let s_i be time to swap page into memory
  - let r be the time required to restart an interrupted instruction
  - Therefore,

$$E = (1-p) * m + p * (f + s_o + s_i + r)$$

Demand Paging Example

- Assume that memory access time, m is 200ns
- Average page-fault service time, $f + s_o + s_i + r$ is 8ms

$$\begin{aligned} E &= (1-p) * 200 + p * 8e6 \\ &= 200 + 7.9998e6 * p \end{aligned}$$

- If one access out of 1000 causes a page fault, then $E = 8.2$microseconds
  - This is a slowdown by a factor of 40
- If want keep performance impact less than 10 percent, we must have less than one page fault in every 400,000 memory accesses

$$\begin{aligned} 200 &> 200 + 7.9998e6 * p \\ 20 &> 7.9998e6 * p \\ p &> 0.0000025 \end{aligned}$$

-

Demand Paging Optimizations

- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used on older versions of the BSD operating systems
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current versions of BSD (FreeBSD, OpenBSD, NetBSD)

2 Page Replacement Algorithms

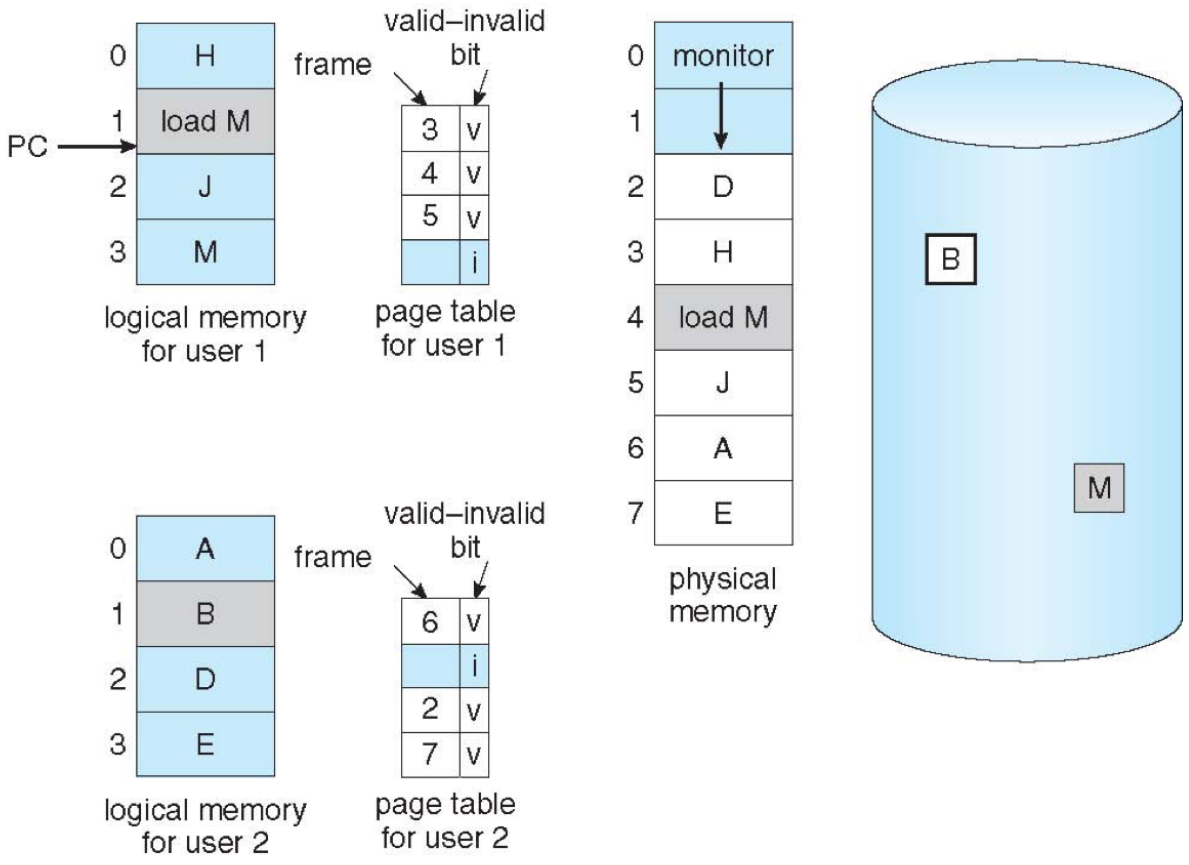
What Happens If There No Free Frame

- So, lot more pages than frames
  - Processes use frames, but so does the kernel, I/O buffering,...
  - How much to allocate to each?
- **Page replacement**: Find a page in memory not currently in use and page it to secondary memory
  - Algorithm - Terminate? Swap out? Replace the page?
  - Performance - Need algorithm that results in the least number of page faults
- The same page may be brought into memory several times.

Page Replacement

- Prevent over-allocation of memory by modifying the page-fault service routine to include page replacement
- Add a *modify (dirty) bit* to the page table to reduce overhead of page transfers by writing only modified pages to disk
- Page replacement completes separation of logical and physical memory
  - Large virtual memory can be provided on a smaller physical memory
  - Think Intel x64 - Processes have a 2^{64} bit address space on a physical memory of 4GB

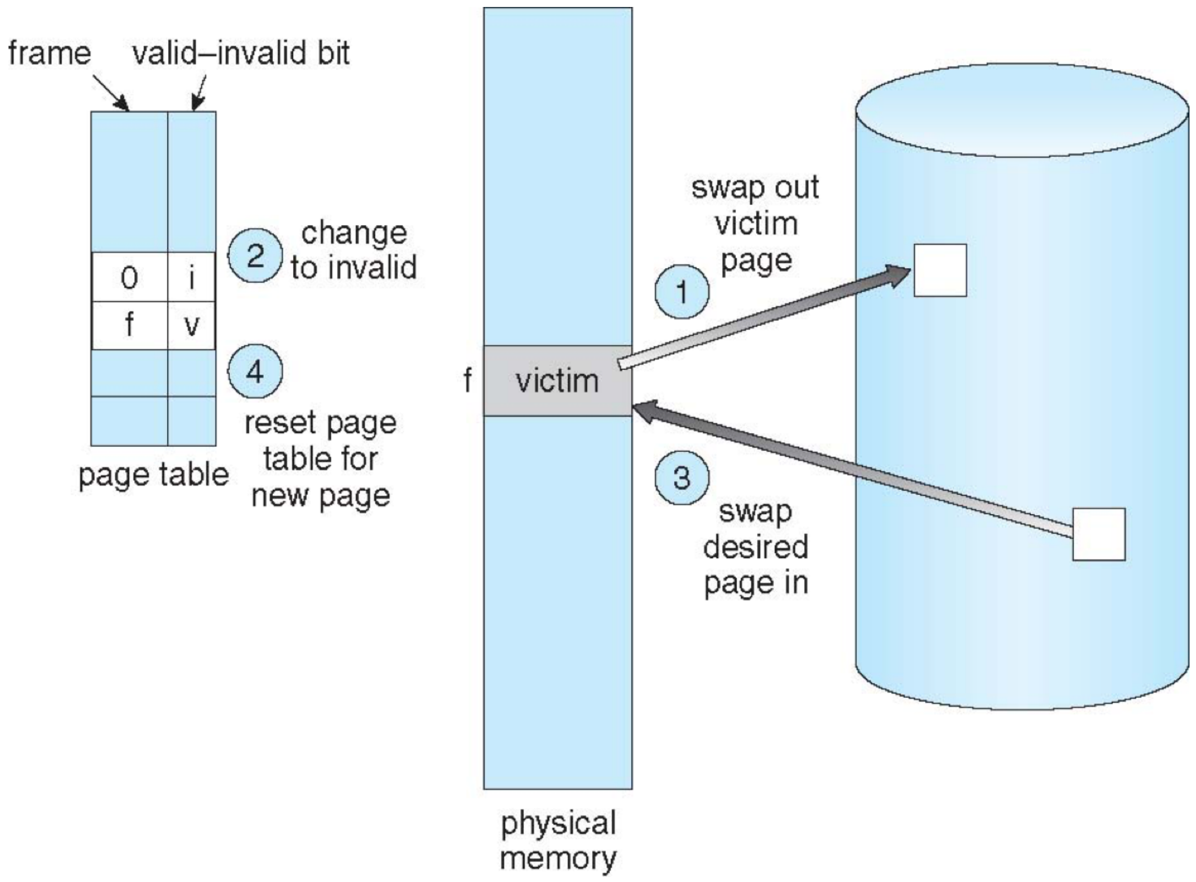
The Need For Page Replacement



Basic Page Replacement Algorithm

1. Find the location of the desired page on disk
2. Find a free frame
   (a) If there is a free frame, use it
   (b) If there is no free frame, use a *page replacement algorithm* to select a *victim frame*
   (c) Write victim frame to disk if dirty
3. Load page into the (newly) free frame and upate the page and frame tables
4. Restart the instruction that caused the trap

Page Replacement Steps



Page Replacement and Frame Allocation Algorithms

- **Frame allocation algorithm**: determines how many frames to give each process and which frame to replace
- **Page replacement algorithm**: Want lowest page-fault rate on both first access and subsequent accesses
- Evaluate algorithm by running on a *reference string*, a particular string of memory references indexed by page number, and computing the number of faults on that string
- In the following examples, the reference string is:

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

First-in, First-out Page Algorithme

- Reference string:

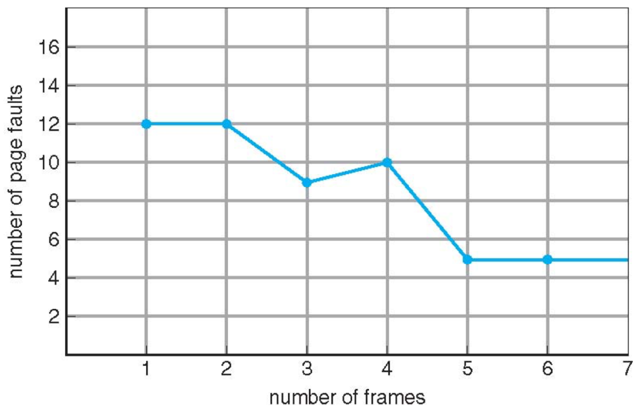
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

- Allow each process to use 3 frames:

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

- This scheme has 15 page faults, but can vary by reference string
  - Consider 1,2,3,4,1,2,5,1,2,3,4,5
- **Belady's Anomaly**: Adding more frames can cause more page faults
- How to track ages of pages: Just use a queue!

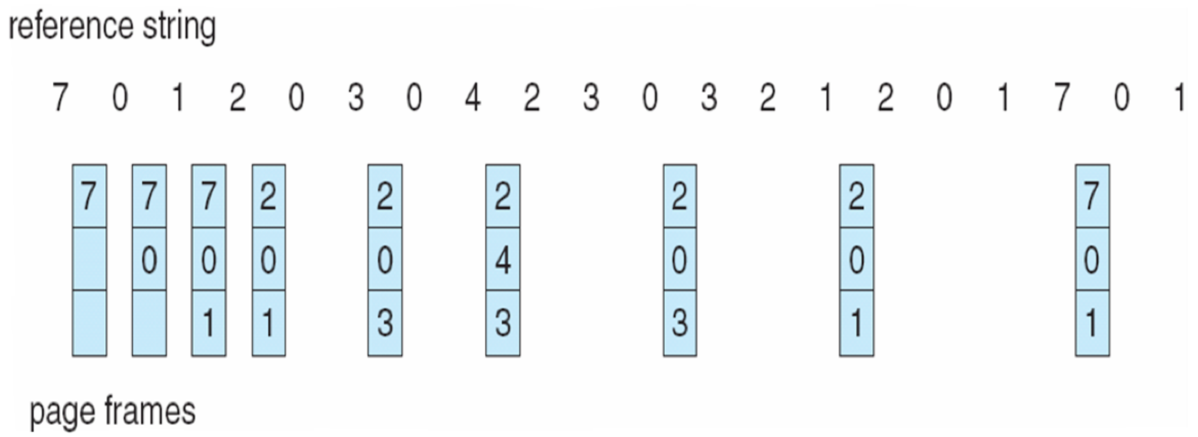
Belady's Anomaly



Optimal Algorithm

- Replace the page that will not be used in the longest period of time
- But this requires perfect prediction of the future
- Useful for measuring how well other PRAs work

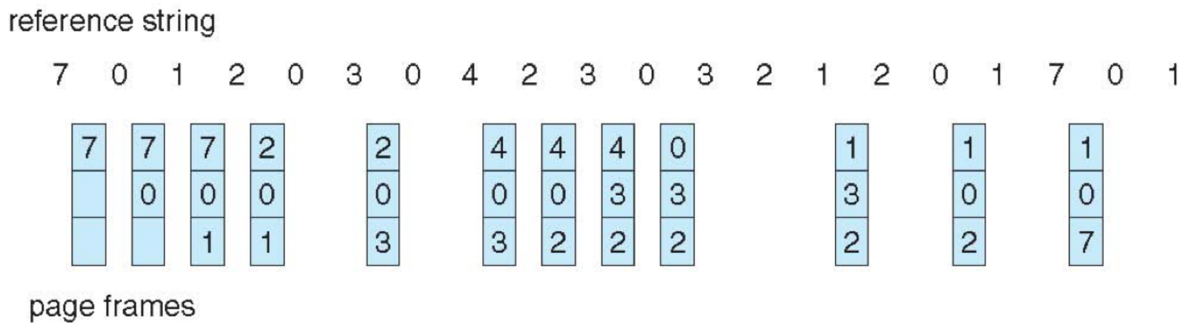
Optimal Page Replacement



For our example reference string, the optimal algorithm suffers from 9 page faults

Least Recently Used (LRU) Algorithm

- Use page knowledge rather than future prediction
- Replace page that has not been used in the most amount of time
- Keep track of the time of last use with each page



- So, 12 faults, better than FIFO but worse than OPT
- Good algorithm that is used often
- But how to implement?

LRU Algorithm

- Option 1: Counter implementation
  - Each page entry has a counter, dump time stamp into counter each time page is accessed
  - Search through table to find counter with smallest value
- Option 2: Stack implementation
  - Keep a stack of page numbers (using a double-linked stack)
  - If page referenced, move to top of stack. This requires 6 pointers to be changed
  - Updates are more expensive but no need to search
- LRU and OPT are examples of *stack PRAs* that do not suffer from Belady's Anomaly

LRU Approximation Algorithms

- LRU implementation needs special hardware and still slow
- Reference Bit Replacement
  - Associate a bit to each page, initially set to 0
  - Set the bit to 1 when page is referenced
  - Replace any page with reference bit set to 0
  - Note this doesn't take order into account
- Second-chance Algorithm
  - FIFO, but with hardware reference bit

  - Clock replacement
  - If page to be replaced has
    * Reference bit set 0, then replace
    * Otherwise, set bit to 0, leave page in memory
    * Look at following page, and repeat

Counting Algorithms

- Not common, but keep a counter of number of references made to each page
- **LFU Algorithm**: replace page with smallest count
- **MFU Algorithm**: based on argument that page with smallest count was just brought in and has yet to be used, so replace page with largest count

Page-Buffering Algorithms

- Optiion 1: Keep a pool of free frames
  - Frame available when needed, not found at fault time
  - Read page into free frame, select victim to evict and add that frame to free pool
  - Evict victim when convenient
- Option 2: Keep pool and leavefree frame contents intact and note what is in them
  - If frame referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Replacement Policy and CPU Cache Size

- Page replacement can have performance impacts if using a large cache
- If page frame selected for replacement is in cache, replacement invalidates the cache block *AND* replaces the frame
- Page buffering moderates this by considering cache behavior in policy for page replacement in the buffer
- Most operating systems place pages by selecting an arbitrary page frame from the page buffer

Main memory size is getting larger and the locality of applications is decreasing. In compensation, cache sizes have been increasing. Large cache sizes, even multimegabyte ones, are now feasible design alternatives. With a large cache, the replacement of virtual memory pages can have a performance impact. If the page frame selected for replacement is in the cache, then that cache block is lost as well as the page that it holds.

In systems that use some form of page buffering, it is possible to improve cache performance by supplementing the page replacement policy with a policy for page placement in the page buffer. Most operating systems place pages by selecting an arbitrary page frame from the page buffer; typically a first-in-first-out discipline is used.

3 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

With paged virtual memory, it is not necessary and indeed may not be possible to bring all of the pages of a process into main memory to prepare it for execution. Thus, the operating system must decide how many pages to bring in, that is, how much main memory to allocate to a particular process. Several factors come into play:

- The smaller the amount of memory allocated to a process, the more processes that can reside in main memory at any one time. This increases the probability that the operating system will find at least one ready process at any given time and hence reduces the time lost due to swapping.
- If a relatively small number of pages of a process are in main memory, then, despite the principle of locality, the rate of page faults will be rather high.
- Beyond a certain size, additional allocation of main memory to a particular process will have no noticeable effect on the page fault rate for that process because of the principle of locality.

Fixed allocation

- Gives a process a fixed number of frames in main memory within which to execute
- On page fault, one of the pages owned by that process must be replaced

Variable Allocation

- Allows the number of pages allocated to a process to be varied over the lifetime of the process

With these factors in mind, two sorts of policies are to be found in contemporary operating systems. A fixed-allocation policy gives a process a fixed number of frames in main memory within which to execute. That number is decided at initial load time (process creation time) and may be determined based on the type of process (interactive, batch, type of application) or may be based on guidance from the programmer or system manager. With a fixed-allocation policy, whenever a page fault occurs in the execution of a process, one of the pages of that process must be replaced by the needed page.

A variable-allocation policy allows the number of page frames allocated to a process to be varied over the lifetime of the process. Ideally, a process that is suffering persistently high levels of page faults, indicating that the principle of locality only holds in a weak form for that process, will be given additional page frames to reduce the page fault rate; whereas a process with an exceptionally low page fault rate, indicating that the process is quite well behaved from a locality point of view, will be given a reduced allocation, with the hope that this will not noticeably increase the page fault rate.

The variable-allocation policy would appear to be the more powerful one. However, the difficulty with this approach is that it requires the operating system to assess the behavior of active processes. This inevitably requires software overhead in the operating system and is dependent on hardware mechanisms provided by the processor platform.

4 Other Polices

Cleaning Policy

Concerned with determining when a modified page should be written out to secondary memory

Demand Cleaning
A page is written out to secondary memory only when it has been selected for replacement

Precleaning
Allows the writing of pages in batches

A cleaning policy is the opposite of a fetch policy; it is concerned with determining when a modified page should be written out to secondary memory. Two common alternatives are demand cleaning and precleaning. With demand cleaning , a page is written out to secondary memory only when it has been selected for replacement. A precleaning policy writes modified pages before their page frames are needed so that pages can be written out in batches.

There is a danger in following either policy to the full. With precleaning, a page is written out but remains in main memory until the page replacement algorithm dictates that it be removed. Precleaning allows the writing of pages in batches, but it makes little sense to write out hundreds or thousands of pages only to find that the majority of them have been modified again before they are replaced. The transfer capacity of secondary memory is limited and should not be wasted with unnecessary cleaning operations.

Load Control

- Determines the number of processes that will be resident in main memory
  - Multiprogramming level
- Critical in effective memory management
- Too few processes, many occasions when all processes will be blocked and much time spent in swapping
- Too many processes will lead to thrashing

Load control is concerned with determining the number of processes that will be resident in main memory, which has been referred to as the multiprogramming level. The load control policy is critical in effective memory management. If too few processes are resident at any one time, then there will be many occasions when all processes are blocked, and much time will be spent in swapping. On the other hand, if too many processes are resident, then, on average, the size of the resident set of each process will be inadequate and frequent faulting will occur. The result is thrashing.

5 Key Points

Key Points

- Overall design of virtual memory
- Page Replacement Algorithms
- Other policies