Virtual Memory

Chapter 8

Characteristics of Paging and Segmentation

- **Memory references are dynamically translated into physical addresses at run time**
  - a process may be swapped in and out of main memory such that it occupies different regions
- **A process may be broken up into pieces (pages or segments) that do not need to be located contiguously in main memory**
- **Hence: all pieces of a process do not need to be loaded in main memory during execution**
  - computation may proceed for some time if the next instruction to be fetch (or the next data to be accessed) is in a piece located in main memory

Process Execution

- **The OS brings into main memory only a few pieces of the program (including its starting point)**
- **Each page/segment table entry has a present bit that is set only if the corresponding piece is in main memory**
- **The resident set is the portion of the process that is in main memory**
- **An interrupt (memory fault) is generated when the memory reference is on a piece not present in main memory**

Process Execution (cont.)

- **OS places the process in a Blocking state**
- **OS issues a disk I/O Read request to bring into main memory the piece referenced to**
- **another process is dispatched to run while the disk I/O takes place**
- **an interrupt is issued when the disk I/O completes**
  - this causes the OS to place the affected process in the Ready state

Advantages of Partial Loading

- **More processes can be maintained in main memory**
  - only load in some of the pieces of each process
  - With more processes in main memory, it is more likely that a process will be in the Ready state at any given time
- **A process can now execute even if it is larger than the main memory size**
  - this is possible by using more bits for logical addresses than the bits needed for addressing the physical memory

Virtual Memory: large as you wish!

  - Ex: 16 bits are needed to address a physical memory of 64KB
  - lets use a page size of 1KB so that 10 bits are needed for offsets within a page
  - For the page number part of a logical address we may use a number of bits larger than 6, say 22 (a modest value!!)
- **The memory referenced by a logical address is called virtual memory**
  - is maintained on secondary memory (ex: disk)
  - pieces are bring into main memory only when needed

Virtual Memory (cont.)

- For better performance, the file system is often bypassed and virtual memory is stored in a special area of the disk called the swap space
  - larger blocks are used and file lookups and indirect allocation methods are not used
- **By contrast, physical memory is the memory referenced by a physical address**
  - is located on DRAM
- **The translation from logical address to physical address is done by indexing the appropriate page/segment table with the help of memory management hardware**

Possibility of trashing

- **To accommodate as many processes as possible, only a few pieces of each process is maintained in main memory**
- **But main memory may be full: when the OS brings one piece in, it must swap one piece out**
- **The OS must not swap out a piece of a process just before that piece is needed**
- **If it does this too often this leads to trashing:**
  - The processor spends most of its time swapping pieces rather than executing user instructions

Locality and Virtual Memory

- Principle of locality of references: memory references within a process tend to cluster
- Hence: only a few pieces of a process will be needed over a short period of time
- Possible to make intelligent guesses about which pieces will be needed in the future
- This suggests that virtual memory may work efficiently (ie: trashing should not occur too often)

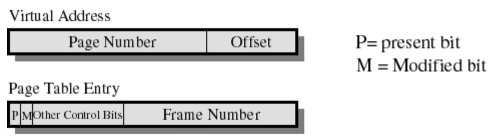
Support Needed for Virtual Memory

- Memory management hardware must support paging and/or segmentation
- OS must be able to manage the movement of pages and/or segments between secondary memory and main memory

- We will first discuss the hardware aspects; then the algorithms used by the OS

Paging

- **Typically, each process has its own page table**



- **Each page table entry contains a present bit to indicate whether the page is in main memory or not.**
  - If it is in main memory, the entry contains the frame number of the corresponding page in main memory
  - If it is not in main memory, the entry may contain the address of that page on disk or the page number may be used to index another table (often in the PCB) to obtain the address of that page on disk

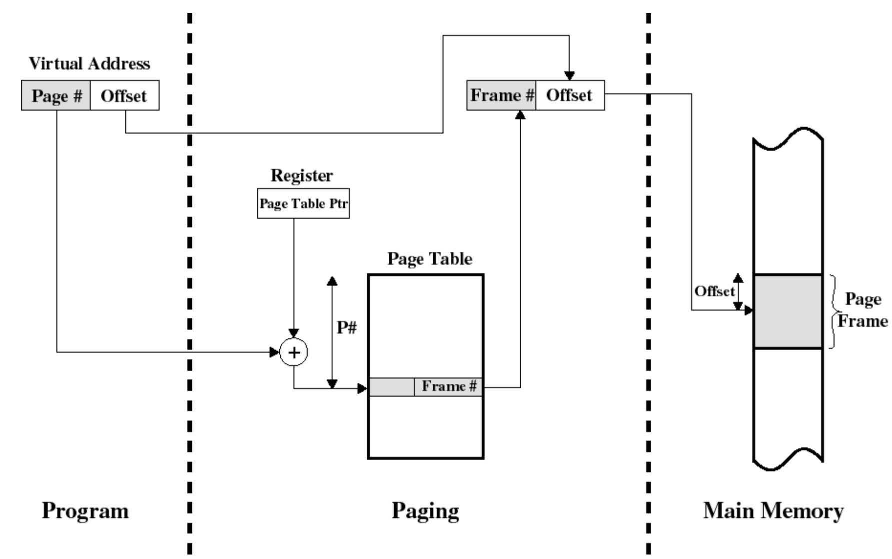
Paging

- **A modified bit indicates if the page has been altered since it was last loaded into main memory**
  - If no change has been made, the page does not have to be written to the disk when it needs to be swapped out
- **Other control bits may be present if protection is managed at the page level**
  - a read-only/read-write bit
  - protection level bit: kernel page or user page (more bits are used when the processor supports more than 2 protection levels)

Page Table Structure

- **Page tables are variable in length (depends on process size)**
  - then must be in main memory instead of registers
- **A single register holds the starting physical address of the page table of the currently running process**

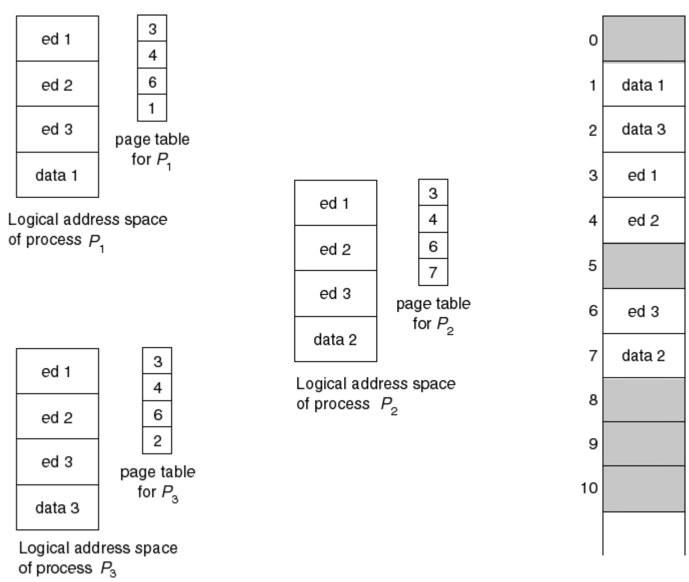
Address Translation in a Paging System



Sharing Pages

- If we share the same code among different users, it is sufficient to keep only one copy in main memory
- Shared code must be reentrant (ie: non self-modifying) so that 2 or more processes can execute the same code
- If we use paging, each sharing process will have a page table who's entry points to the same frames: only one copy is in main memory
- But each user needs to have its own private data pages

Sharing Pages: a text editor



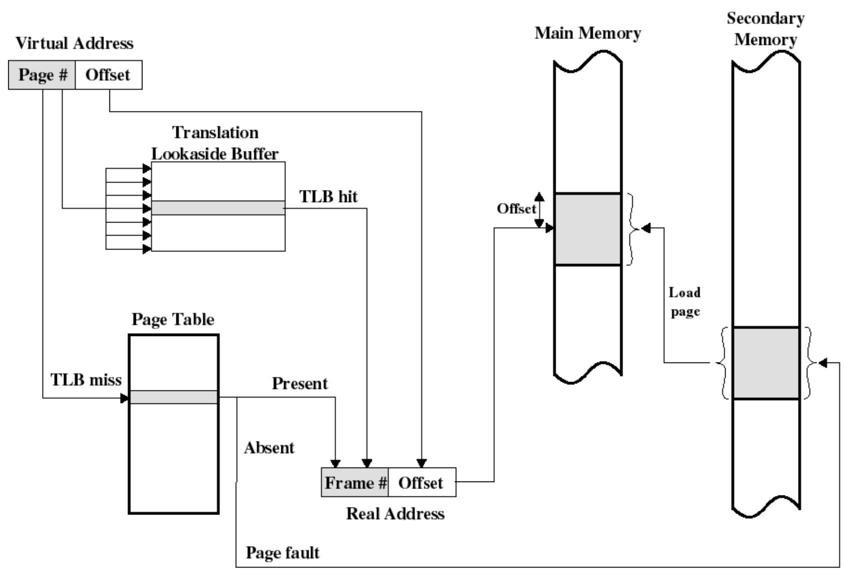
Translation Lookaside Buffer

- **Because the page table is in main memory, each virtual memory reference causes at least two physical memory accesses**
  - one to fetch the page table entry
  - one to fetch the data
- **To overcome this problem a special cache is set up for page table entries**
  - called the TLB - Translation Lookaside Buffer
    - Contains page table entries that have been most recently used
    - Works similar to main memory cache

Translation Lookaside Buffer

- **Given a logical address, the processor examines the TLB**
- **If page table entry is present (a hit), the frame number is retrieved and the real (physical) address is formed**
- **If page table entry is not found in the TLB (a miss), the page number is used to index the process page table**
  - if present bit is set then the corresponding frame is accessed
  - if not, a page fault is issued to bring in the referenced page in main memory
- **The TLB is updated to include the new page entry**

Use of a Translation Lookaside Buffer



TLB: further comments

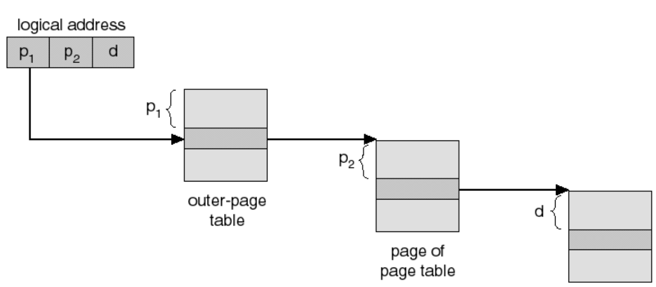
- **TLB use associative mapping hardware to simultaneously interrogates all TLB entries to find a match on page number**
- **The TLB must be flushed each time a new process enters the Running state**
- **The CPU uses two levels of cache on each virtual memory reference**
  - first the TLB: to convert the logical address to the physical address
  - once the physical address is formed, the CPU then looks in the cache for the referenced word

Page Tables and Virtual Memory

- **Most computer systems support a very large virtual address space**
  - 32 to 64 bits are used for logical addresses
  - If (only) 32 bits are used with 4KB pages, a page table may have 2^{20} entries
- **The entire page table may take up too much main memory. Hence, page tables are often also stored in virtual memory and subjected to paging**
  - When a process is running, part of its page table must be in main memory (including the page table entry of the currently executing page)

Multilevel Page Tables

- **Since a page table will generally require several pages to be stored. One solution is to organize page tables into a multilevel hierarchy**
  - When 2 levels are used (ex: 386, Pentium), the page number is split into two numbers p1 and p2
  - p1 indexes the outer paged table (directory) in main memory who’s entries points to a page containing page table entries which is itself indexed by p2. Page tables, other than the directory, are swapped in and out as needed



Windows NT Virtual Memory

- **Uses paging only (no segmentation) with a 4KB page size**
- **Each process has 2 levels of page tables:**
  - a page directory containing 1024 page-directory entries (PDEs) of 4 bytes each
  - each page-directory entry points to a page table that contains 1024 page-table entries (PTEs) of 4 bytes each
  - so we have 4MB of page tables per process
  - the page directory is in main memory but page tables containing PTEs are swapped in and out as needed

Windows NT Virtual Memory

- **Virtual addresses (p1, p2, d) use 32 bits where p1 and p2 are each 10 bits wide**
  - p1 selects an entry in the page directory which points to a page table
  - p2 selects an entry in this page table which points to the selected page
- **Upon creation, NT commits only a certain number of virtual pages to a process and reserves a certain number of other pages for future needs**
- **Hence, a group of bits in each PTE indicates if the corresponding page is committed, reserved or not used**

Windows NT Virtual Memory

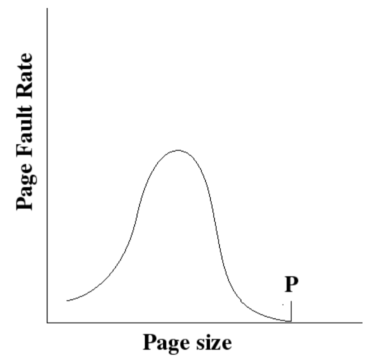
- **A memory reference to an unused page traps into the OS (protection violation)**
- **Each PTE also contains:**
  - a present bit
    - If set: 20 bits are used for the frame address of the selected page.
    - Else these bits are used to locate the selected page in a paging file (on disk)
  - some bits identify the paging file used
  - a dirty bit (ie: a modified bit)
  - some protection bits (ex: read-only, or read-write)

The Page Size Issue

- **Page size is defined by hardware; always a power of 2 for more efficient logical to physical address translation. But exactly which size to use is a difficult question:**
  - Large page size is good since for a small page size, more pages are required per process
    - More pages per process means larger page tables. Hence, a large portion of page tables in virtual memory
  - Small page size is good to minimize internal fragmentation
  - Large page size is good since disks are designed to efficiently transfer large blocks of data
  - Larger page sizes means less pages in main memory; this increases the TLB hit ratio

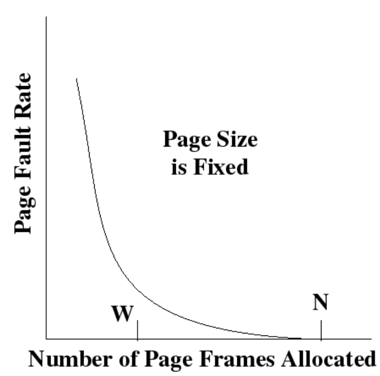
The Page Size Issue

- With a very small page size, each page matches the code that is actually used: faults are low
- Increased page size causes each page to contain more code that is not used. Page faults rise.
- Page faults decrease if we can approach point P were the size of a page is equal to the size of the entire process



The Page Size Issue

- Page fault rate is also determined by the number of frames allocated per process
- Page faults drops to a reasonable value when W frames are allocated
- Drops to 0 when the number (N) of frames is such that a process is entirely in memory

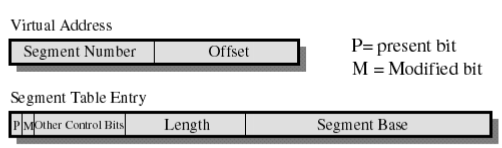


The Page Size Issue

- **Page sizes from 1KB to 4KB are most commonly used**
- **But the issue is non trivial. Hence some processors are now supporting multiple page sizes. Ex:**
  - Pentium supports 2 sizes: 4KB or 4MB
  - R4000 supports 7 sizes: 4KB to 16MB

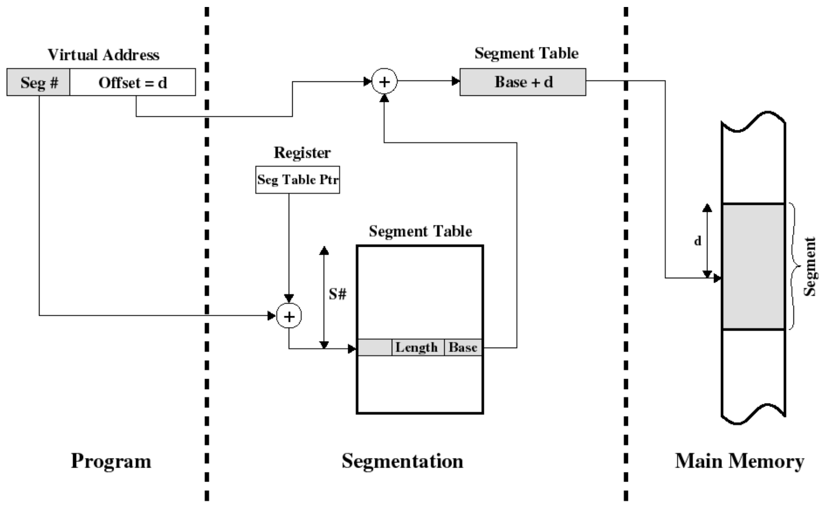
Segmentation

- **Typically, each process has its own segment table**



- **Similarly to paging, each segment table entry contains a present bit and a modified bit**
- **If the segment is in main memory, the entry contains the starting address and the length of that segment**
- **Other control bits may be present if protection and sharing is managed at the segment level**
- **Logical to physical address translation is similar to paging except that the offset is added to the starting address (instead of being appended)**

Address Translation in a Segmentation System



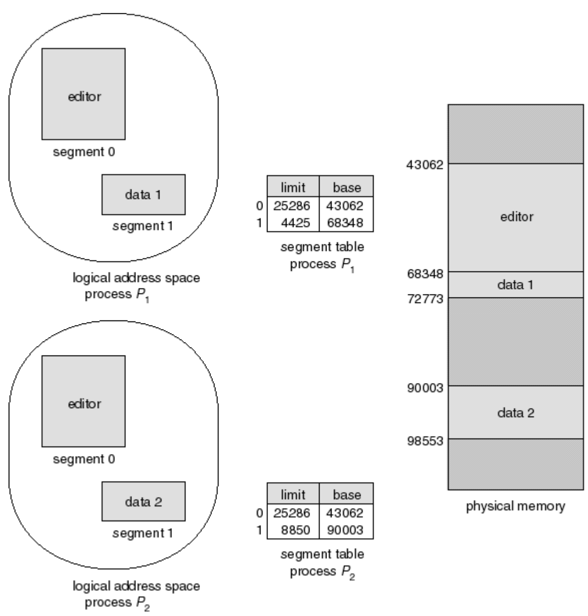
Segmentation: comments

- **In each segment table entry we have both the starting address and length of the segment**
  - the segment can thus dynamically grow or shrink as needed
  - address validity easily checked with the length field
- **But variable length segments introduce external fragmentation and are more difficult to swap in and out...**
- **It is natural to provide protection and sharing at the segment level since segments are visible to the programmer (pages are not)**
- **Useful protection bits in segment table entry:**
  - read-only/read-write bit
  - Supervisor/User bit

Sharing in Segmentation Systems

- **Segments are shared when entries in the segment tables of 2 different processes point to the same physical locations**
- **Ex: the same code of a text editor can be shared by many users**
  - Only one copy is kept in main memory
- **but each user would still need to have its own private data segment**

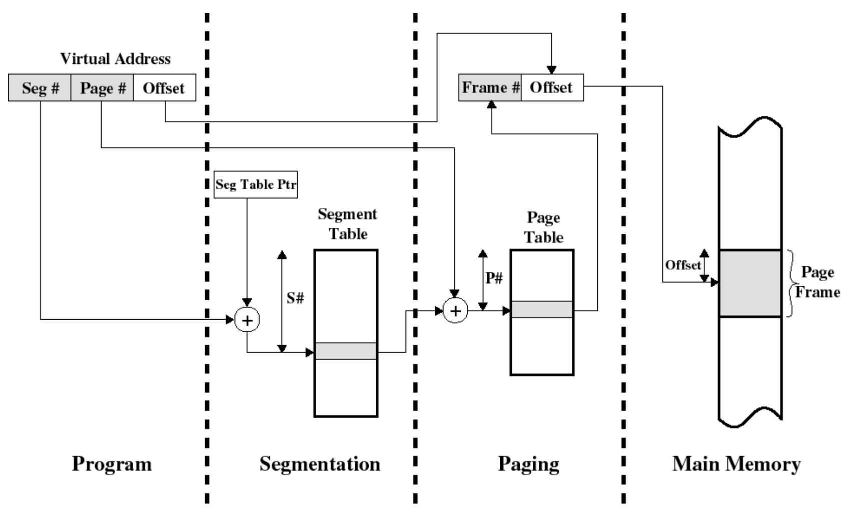
Sharing of Segments: text editor example



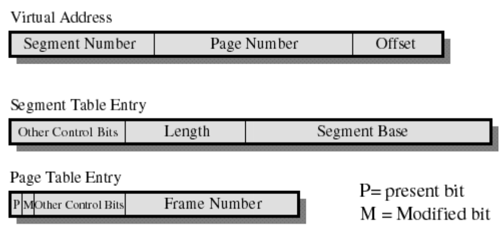
Combined Segmentation and Paging

- **To combine their advantages some processors and OS page the segments.**
- **Several combinations exists. Here is a simple one**
- **Each process has:**
  - one segment table
  - several page tables: one page table per segment
- **The virtual address consist of:**
  - a segment number: used to index the segment table who's entry gives the starting address of the page table for that segment
  - a page number: used to index that page table to obtain the corresponding frame number
  - an offset: used to locate the word within the frame

Address Translation in a (simple) combined Segmentation/Paging System



Simple Combined Segmentation and Paging



- **The Segment Base is the physical address of the page table of that segment**
- **Present and modified bits are present only in page table entry**
- **Protection and sharing info most naturally resides in segment table entry**
  - Ex: a read-only/read-write bit, a kernel/user bit...

Operating System Software

- **Memory management software depends on whether the hardware supports paging or segmentation or both**
- **Pure segmentation systems are rare. Segments are usually paged -- memory management issues are then those of paging**
- **We shall thus concentrate on issues associated with paging**
- **To achieve good performance we need a low page fault rate**

Fetch Policy

- **Determines when a page should be brought into main memory. Two common policies:**
  - Demand paging only brings pages into main memory when a reference is made to a location on the page (ie: paging on demand only)
    - many page faults when process first started but should decrease as more pages are brought in
  - Prepaging brings in more pages than needed
    - locality of references suggest that it is more efficient to bring in pages that reside contiguously on the disk
    - efficiency not definitely established: the extra pages brought in are “often” not referenced

Placement policy

- **Determines where in real memory a process piece resides**
- **For pure segmentation systems:**
  - first-fit, next fit... are possible choices (a real issue)
- **For paging (and paged segmentation):**
  - the hardware decides where to place the page: the chosen frame location is irrelevant since all memory frames are equivalent (not an issue)

Replacement Policy

- **Deals with the selection of a page in main memory to be replaced when a new page is brought in**
- **This occurs whenever main memory is full (no free frame available)**
- **Occurs often since the OS tries to bring into main memory as many processes as it can to increase the multiprogramming level**

Replacement Policy

- **Not all pages in main memory can be selected for replacement**
- **Some frames are locked (cannot be paged out):**
  - much of the kernel is held on locked frames as well as key control structures and I/O buffers
- **The OS might decide that the set of pages considered for replacement should be:**
  - limited to those of the process that has suffered the page fault
  - the set of all pages in unlocked frames

Replacement Policy

- **The decision for the set of pages to be considered for replacement is related to the resident set management strategy:**
  - how many page frames are to be allocated to each process? We will discuss this later
- **No matter what is the set of pages considered for replacement, the replacement policy deals with algorithms that will choose the page within that set**

Basic algorithms for the replacement policy

- **The Optimal policy selects for replacement the page for which the time to the next reference is the longest**
  - produces the fewest number of page faults
  - impossible to implement (need to know the future) but serves as a standard to compare with the other algorithms we shall study:
    - Least recently used (LRU)
    - First-in, first-out (FIFO)
    - Clock

The LRU Policy

- **Replaces the page that has not been referenced for the longest time**
  - By the principle of locality, this should be the page least likely to be referenced in the near future
  - performs nearly as well as the optimal policy
- **Example: A process of 5 pages with an OS that fixes the resident set size to 3**

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
OPT	2	2	2	2	2	2	4	4	4	2	2	2
		3	3	3	3	3	3	3	3	3	3	3
				1	5	5	5	5	5	5	5	5
					F		F			F		
LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
					F		F		F	F		

Implementation of the LRU Policy

- **Each page could be tagged (in the page table entry) with the time at each memory reference.**
- **The LRU page is the one with the smallest time value (needs to be searched at each page fault)**
- **This would require expensive hardware and a great deal of overhead.**
- **Consequently very few computer systems provide sufficient hardware support for true LRU replacement policy**
- **Other algorithms are used instead**

The FIFO Policy

- **Treats page frames allocated to a process as a circular buffer**
  - When the buffer is full, the oldest page is replaced. Hence: first-in, first-out
    - This is not necessarily the same as the LRU page
    - A frequently used page is often the oldest, so it will be repeatedly paged out by FIFO
  - Simple to implement
    - requires only a pointer that circles through the page frames of the process

Comparison of FIFO with LRU

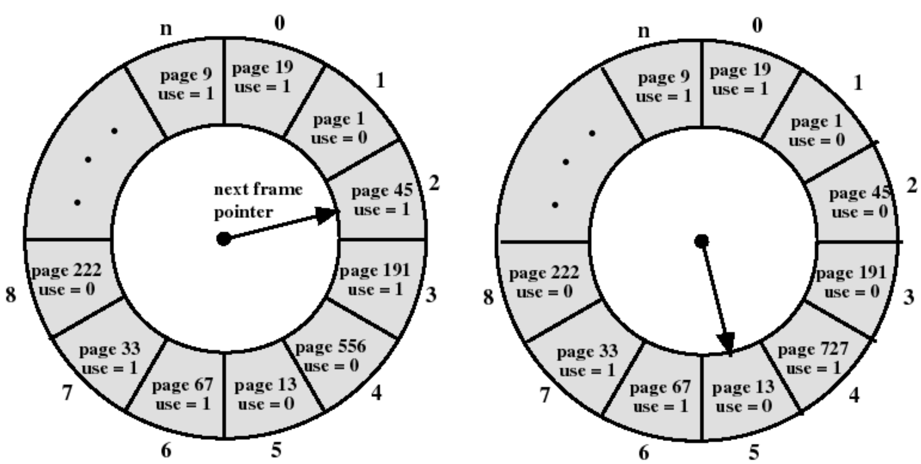
Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
LRU	2	2 3	2 3	2 3 1	2 5 1 F	2 5 1	2 5 4 F	2 5 4	3 5 4 F	3 5 2 F	3 5 2	3 5 2
FIFO	2	2 3	2 3	2 3 1	5 3 1 F	5 2 1 F	5 2 4 F	5 2 4	3 2 4 F	3 2 4	3 5 4 F	3 5 2 F

- **LRU recognizes that pages 2 and 5 are referenced more frequently than others but FIFO does not**
- **FIFO performs relatively poorly**

The Clock Policy

- **The set of frames candidate for replacement is considered as a circular buffer**
- **When a page is replaced, a pointer is set to point to the next frame in buffer**
- **A use bit for each frame is set to 1 whenever**
  - a page is first loaded into the frame
  - the corresponding page is referenced
- **When it is time to replace a page, the first frame encountered with the use bit set to 0 is replaced.**
  - During the search for replacement, each use bit set to 1 is changed to 0

The Clock Policy: an example



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

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

Comparison of Clock with FIFO and LRU

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
LRU	2	2 3	2 3	2 3 1	2 5 1 F	2 5 1	2 5 4 F	2 5 4	3 5 4 F	3 5 2 F	3 5 2	3 5 2
FIFO	2	2 3	2 3	2 3 1	5 3 1 F	5 2 1 F	5 2 4 F	5 2 4	3 2 4 F	3 2 4	3 5 4 F	3 5 2 F
CLOCK	2*	2* 3*	2* 3*	2* 3* 1*	5* 3 1 F	5* 2* 1 F	5* 2* 4* F	5* 2* 4*	3* 2 4 F	3* 2* 4	3* 2 5* F	3* 2* 5*

- Asterisk indicates that the corresponding use bit is set to 1
- Clock protects frequently referenced pages by setting the use bit to 1 at each reference

Comparison of Clock with FIFO and LRU

- **Numerical experiments tend to show that performance of Clock is close to that of LRU**
- **Experiments have been performed when the number of frames allocated to each process is fixed and when pages local to the page-fault process are considered for replacement**
  - When few (6 to 8) frames are allocated per process, there is almost a factor of 2 of page faults between LRU and FIFO
  - This factor reduces close to 1 when several (more than 12) frames are allocated. (But then more main memory is needed to support the same level of multiprogramming)

Page Buffering

- **Pages to be replaced are kept in main memory for a while to guard against poorly performing replacement algorithms such as FIFO**
- **Two lists of pointers are maintained: each entry points to a frame selected for replacement**
  - a free page list for frames that have not been modified since brought in (no need to swap out)
  - a modified page list for frames that have been modified (need to write them out)
- **A frame to be replace has a pointer added to the tail of one of the lists and the present bit is cleared in corresponding page table entry**
  - but the page remains in the same memory frame

Page Buffering

- **At each page fault the two lists are first examined to see if the needed page is still in main memory**
  - If it is, we just need to set the present bit in the corresponding page table entry (and remove the matching entry in the relevant page list)
  - If it is not, then the needed page is brought in, it is placed in the frame pointed by the head of the free frame list (overwriting the page that was there)
    - the head of the free frame list is moved to the next entry
  - (the frame number in the page table entry could be used to scan the two lists, or each list entry could contain the process id and page number of the occupied frame)
- **The modified list also serves to write out modified pages in cluster (rather than individually)**

Cleaning Policy

- **When does a modified page should be written out to disk?**
- **Demand cleaning**
  - a page is written out only when it’s frame has been selected for replacement
    - but a process that suffer a page fault may have to wait for 2 page transfers
- **Precleaning**
  - modified pages are written before their frame are needed so that they can be written out in batches
    - but makes little sense to write out so many pages if the majority of them will be modified again before they are replaced

Cleaning Policy

- **A good compromise can be achieved with page buffering**
  - recall that pages chosen for replacement are maintained either on a free (unmodified) list or on a modified list
  - pages on the modified list can be periodically written out in batches and moved to the free list
  - a good compromise since:
    - not all dirty pages are written out but only those chosen for replacement
    - writing is done in batch

Resident Set Size

- **The OS must decide how many page frames to allocate to a process**
  - large page fault rate if to few frames are allocated
  - low multiprogramming level if to many frames are allocated

Resident Set Size

- **Fixed-allocation policy**
  - allocates a fixed number of frames that remains constant over time
    - the number is determined at load time and depends on the type of the application
- **Variable-allocation policy**
  - the number of frames allocated to a process may vary over time
    - may increase if page fault rate is high
    - may decrease if page fault rate is very low
  - requires more OS overhead to assess behavior of active processes

Replacement Scope

- **Is the set of frames to be considered for replacement when a page fault occurs**
- **Local replacement policy**
  - chooses only among the frames that are allocated to the process that issued the page fault
- **Global replacement policy**
  - any unlocked frame is a candidate for replacement
- **Let us consider the possible combinations of replacement scope and resident set size policy**

Fixed allocation + Local scope

- **Each process is allocated a fixed number of pages**
  - determined at load time and depends on application type
- **When a page fault occurs: page frames considered for replacement are local to the page-fault process**
  - the number of frames allocated is thus constant
  - previous replacement algorithms can be used
- **Problem: difficult to determine ahead of time a good number for the allocated frames**
  - if too low: page fault rate will be high
  - if too large: multiprogramming level will be too low

Fixed allocation + Global scope

- **Impossible to achieve**
  - if all unlocked frames are candidate for replacement, the number of frames allocate to a process will necessary vary over time

Variable allocation + Global scope

- **Simple to implement--adopted by many OS (like Unix SVR4)**
- **A list of free frames is maintained**
  - when a process issues a page fault, a free frame (from this list) is allocated to it
  - Hence the number of frames allocated to a page fault process increases
  - The choice for the process that will loose a frame is arbitrary: far from optimal
- **Page buffering can alleviate this problem since a page may be reclaimed if it is referenced again soon**

Variable allocation + Local scope

- **May be the best combination (used by Windows NT)**
- **Allocate at load time a certain number of frames to a new process based on application type**
  - use either prepaging or demand paging to fill up the allocation
- **When a page fault occurs, select the page to replace from the resident set of the process that suffers the fault**
- **Reevaluate periodically the allocation provided and increase or decrease it to improve overall performance**

The Working Set Strategy

- **Is a variable-allocation method with local scope based on the assumption of locality of references**
- **The working set for a process at time t, W(D,t), is the set of pages that have been referenced in the last D virtual time units**
  - virtual time = time elapsed while the process was in execution (eg: number of instructions executed)
  - D is a window of time
  - at any t, |W(D,t)| is non decreasing with D
  - W(D,t) is an approximation of the program's locality

The Working Set Strategy

- **The working set of a process first grows when it starts executing**
- **then stabilizes by the principle of locality**
- **it grows again when the process enters a new locality (transition period)**
  - up to a point where the working set contains pages from two localities
- **then decreases after a sufficient long time spent in the new locality**

296

The Working Set Strategy

- **the working set concept suggest the following strategy to determine the resident set size**
  - Monitor the working set for each process
  - Periodically remove from the resident set of a process those pages that are not in the working set
  - When the resident set of a process is smaller than its working set, allocate more frames to it
    - If not enough free frames are available, suspend the process (until more frames are available)
      - ie: a process may execute only if its working set is in main memory

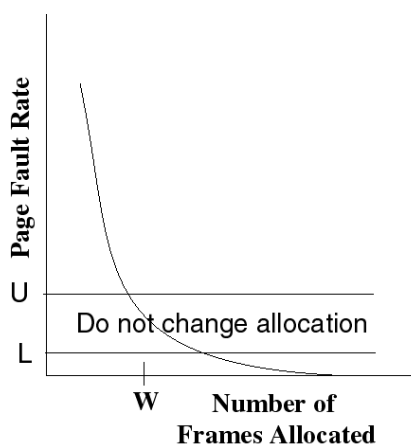
297

The Working Set Strategy

- **Practical problems with this working set strategy**
  - measurement of the working set for each process is impractical
    - necessary to time stamp the referenced page at every memory reference
    - necessary to maintain a time-ordered queue of referenced pages for each process
  - the optimal value for D is unknown and time varying
- **Solution: rather than monitor the working set, monitor the page fault rate!**

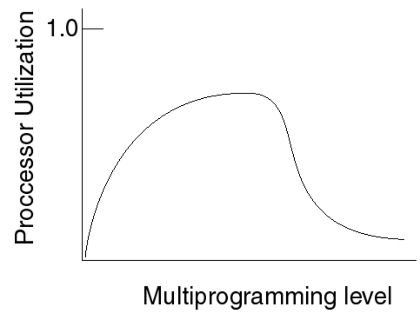
The Page-Fault Frequency Strategy

- **Define an upper bound U and lower bound L for page fault rates**
- **Allocate more frames to a process if fault rate is higher than U**
- **Allocate less frames if fault rate is < L**
- **The resident set size should be close to the working set size W**
- **We suspend the process if the PFF > U and no more free frames are available**



Load Control

- **Determines the number of processes that will be resident in main memory (ie: the multiprogramming level)**
  - Too few processes: often all processes will be blocked and the processor will be idle
  - Too many processes: the resident size of each process will be too small and flurries of page faults will result: thrashing



Load Control

- **A working set or page fault frequency algorithm implicitly incorporates load control**
  - only those processes whose resident set is sufficiently large are allowed to execute
- **Another approach is to adjust explicitly the multiprogramming level so that the mean time between page faults equals the time to process a page fault**
  - performance studies indicate that this is the point where processor usage is at maximum

Process Suspension

- **Explicit load control requires that we sometimes swap out (suspend) processes**
- **Possible victim selection criteria:**
  - Faulting process
    - this process may not have its working set in main memory so it will be blocked anyway
  - Last process activated
    - this process is least likely to have its working set resident
  - Process with smallest resident set
    - this process requires the least future effort to reload
  - Largest process
    - will yield the most free frames