Virtual Memory

Reading:

Silberschatz

chapter 10

Reading:

Stallings

chapter 8

Outline

- Introduction
- Advantages
- Thrashing
- Principal of Locality
- VM based on Paging/Segmentation
- Combined Paging and Segmentation
- Working Set, Paging Behavior
- VM Policies
- Load Control
- OS Examples

Introduction

- All memory references within Process
  - *Logical addresses*
  - Dynamically translated into physical addresses at run time
  - Process may be swapped in ↔ out of main memory at any time
- A process may be broken up into pieces
  - Page or Segments
  - These pieces need not be contiguously located in main memory
- Thus it is not necessary → All the pages or segments of process be in memory during execution
  - If the piece that holds next instruction to be fetched and next data location to be accessed are in main memory
  - Proceed for time being
- How can this be achieved? Lets see in general terms ->

Sample Program Execution

OS begins by bringing in one of few pieces, contains start of program

- Portion of process in main memory at any time → Resident set, smooth execution
- Process → Logical address that is not in main memory, generates an interrupt indicating memory access fault
- Blocking state → OS puts the interrupted process in blocking state and takes control
- Piece of process that contains the logical address that caused access fault is brought into main memory

Advantages

- **More processes can be maintained in main memory** → Efficient utilization of processor!
- **A process may be larger than all of main memory** → Restrictions in programming are lifted, huge size associated with disk storage
- With virtual memory based on paging or segmentation, OS and hardware determines the maximum memory available

Memory Types

- **Real Memory** → Main Memory
- **Virtual Memory** → Memory which is allocated on disk, allows effective multiprogramming and relieves the user of unnecessarily tight constraints of main memory

Thrashing

- Swapping → OS brings one pieces of process in, it must through another out
- If it throws a piece out just it is about to be used? Gets back again almost immediately!
- Thrashing → Above condition when processor spends most of its time in swapping pieces rather than executing instructions
- Variety of complex but effective algorithms
  - Based on recent history, the OS tries to guess ->
  - Which pieces are least likely to be used in near future

Principle of Locality

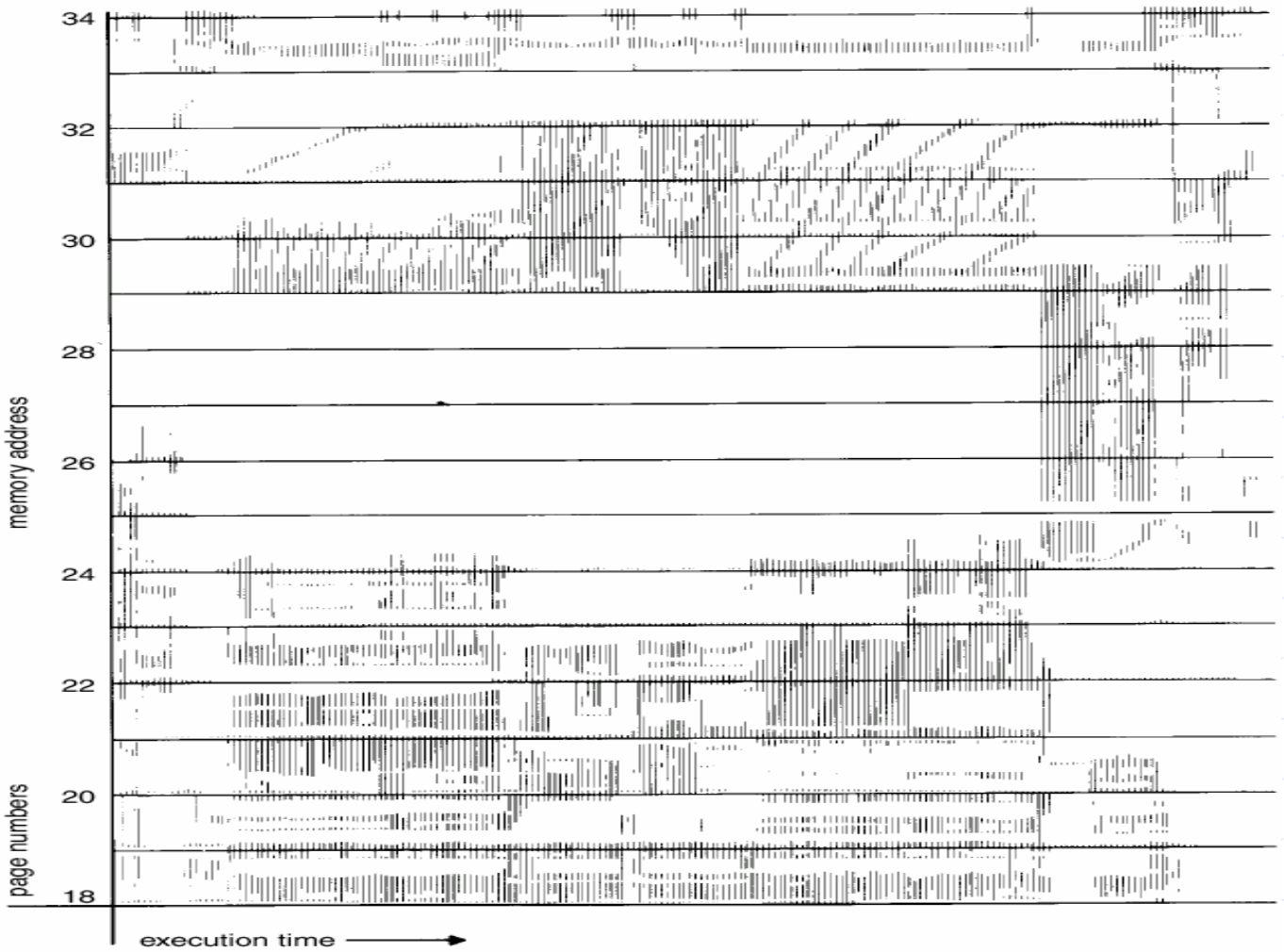
90/10 rule comes from empirical observation

A program spends 90% of its time in 10% of its code

- Locality
  - Set of pages that are actively used together
  - As process executes, it moves from locality to locality
  - Program → several different localities, may overlap, example
- Localities are defined by program structure and its data structure
  - All programs will exhibit this basic memory reference structure
  - If data accesses were random rather than patterned, caching → Useless
- Thus it is possible to make intelligent guesses about which pieces will be needed over a short period of time, which avoids thrashing
- This suggests that a virtual scheme may work efficiently

Locality in Memory Reference Pattern

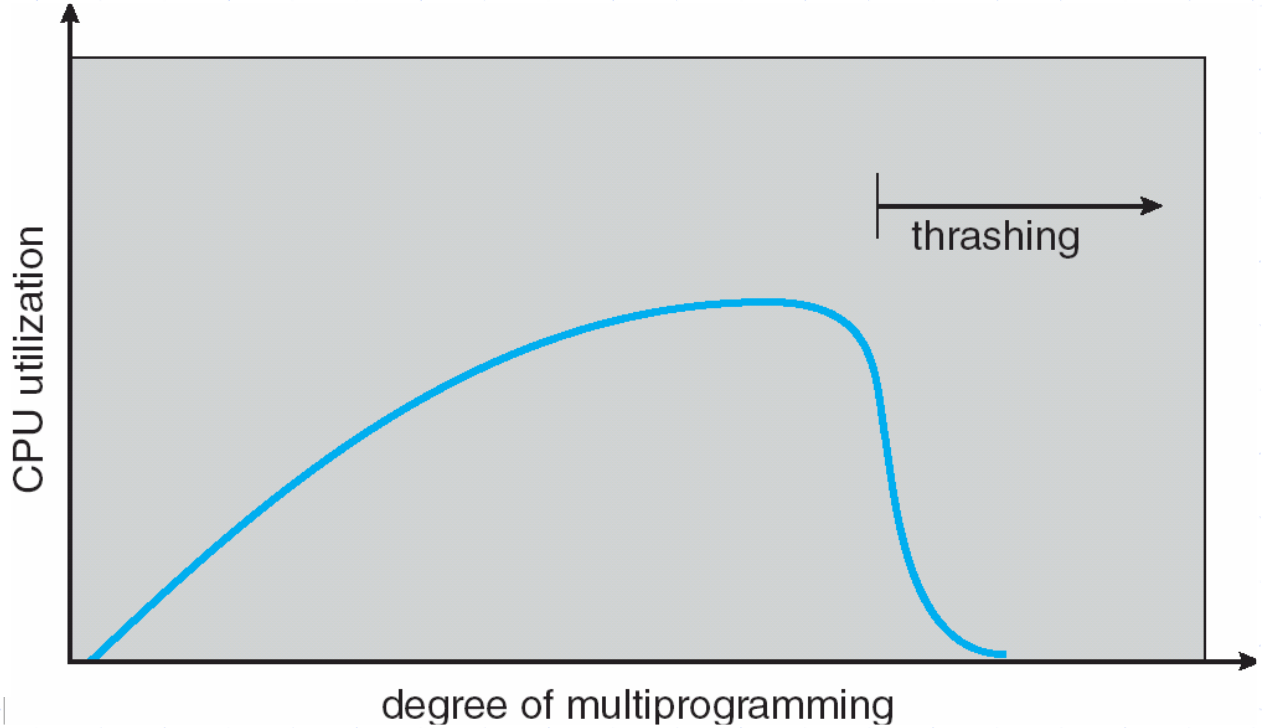
During lifetime of a process, references are confined to a subset of pages



Thrashing

- Process $\rightarrow$ not have "enough" pages, 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 added to the system
- Thrashing
  - a process is busy swapping pages in and out
  - Σ size of locality > total memory size

Thrashing



Effective Access Time (EAT)

- Page Fault Rate $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

EAT = (1 – *p*) x memory access
+ *p* (page fault overhead
+ [swap page out]
+ swap page in
+ restart overhead)

Effective Access Time - Example

- For most computer systems, the memory access time (*ma*) $\rightarrow$ 10 ~ 200 nanoseconds
- If no *page fault* occurs, reads page from disk and then access desired word
- Example
  - If average page-fault service time of 8 milliseconds and memory access time of 200 nanoseconds, EAT?

$$EAT = (1 - p) \times 200 + p \times 8{,}000{,}000 \text{ milliseconds}$$
$$= 200 + 7{,}999{,}800 \times p$$

EAT is directly proportional to fault rate, which should be kept low in paging

Support

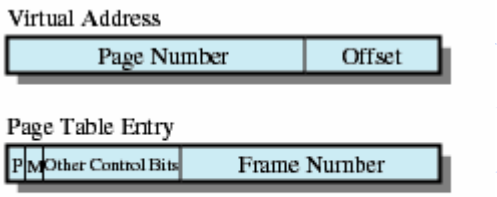
- Support needed for the Virtual Memory
  - *Hardware* → Support paging and segmentation
  - *OS* → Manage the movement of pages and/or segments between secondary and main memory

Virtual Memory based on Paging

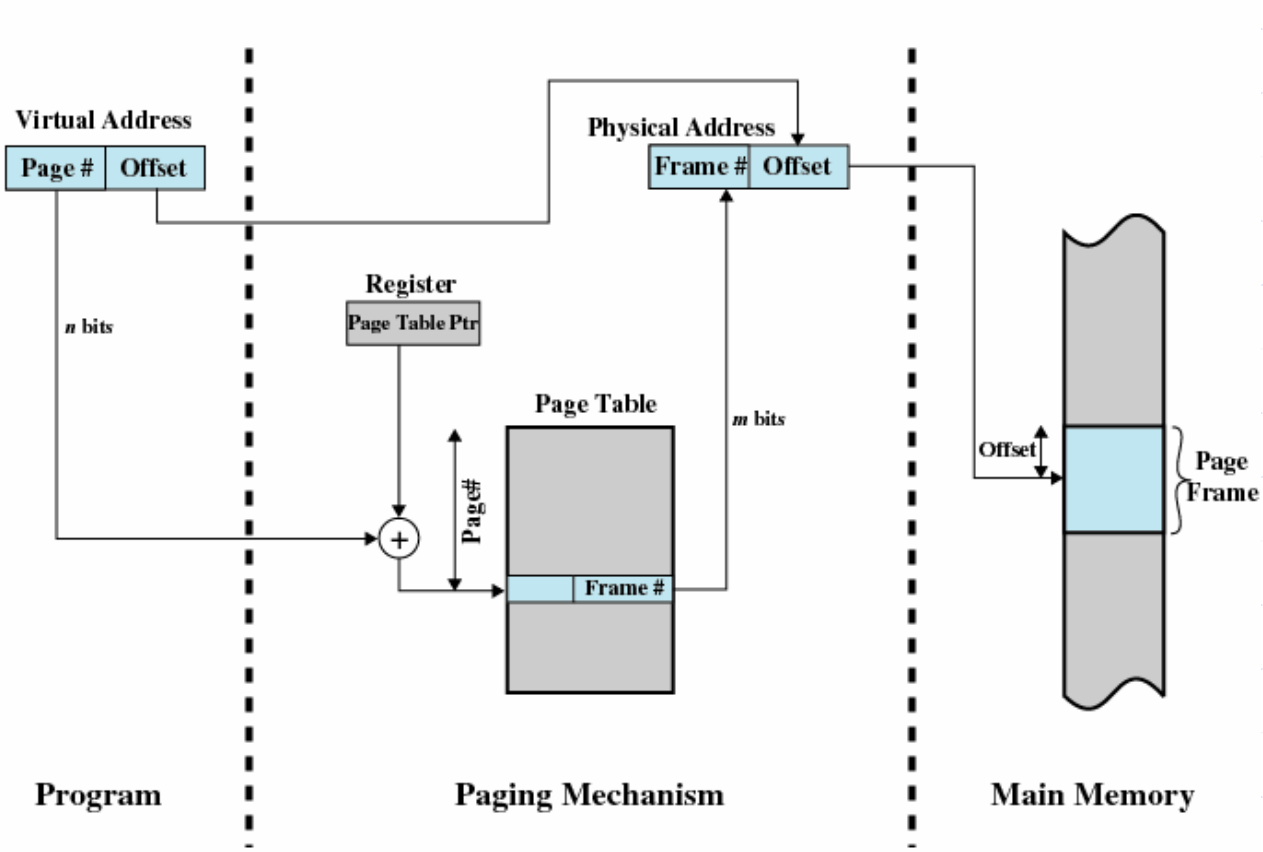
- In most systems, there is one page table per process
- Each page table contains the page number of the corresponding page in the main memory
- Only some pages of process are in main memory
  - A bit is needed in PTE → page present (P) in main memory or not
- Another control bit in *PTE* is modify (M) bit
  - Indicates if this page has been altered since the page was last loaded into memory
  - Why needed?

Other Control Bits

- Used Bit
  - Indicates whether the page has been accessed recently
  - Used by the page replacement algorithm
- Access permissions bit
  - Indicates whether the page is read-only or read-write
- Unix copy-on-write bit
  - Set whether more than one process shares a page
  - If one of the processes writes into the page, a separate copy must first be made for all other process sharing the page
  - Useful for optimizing *fork*()



Address Translation in Paging System



Page Tables

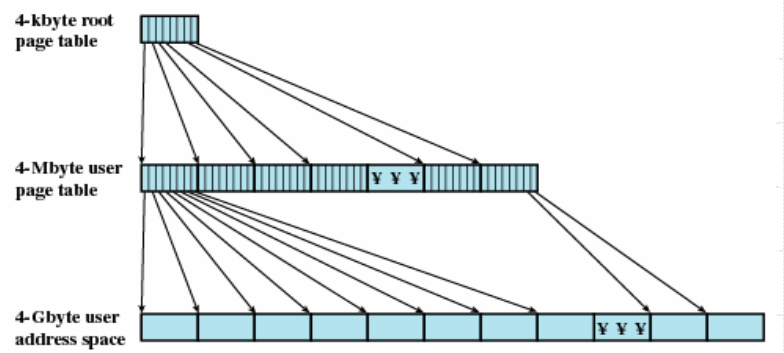
- Main memory for page tables could be unacceptably high
- Therefore most virtual memory schemes also store page tables in virtual memory (instead of real memory)
- When a process is running a part of its PTE must be in main memory, including PTE of currently executing page
- Some processors make use of two-level schemes to organize large page tables, e.g. Pentium processor

Example

Two-level scheme for 32 bit address

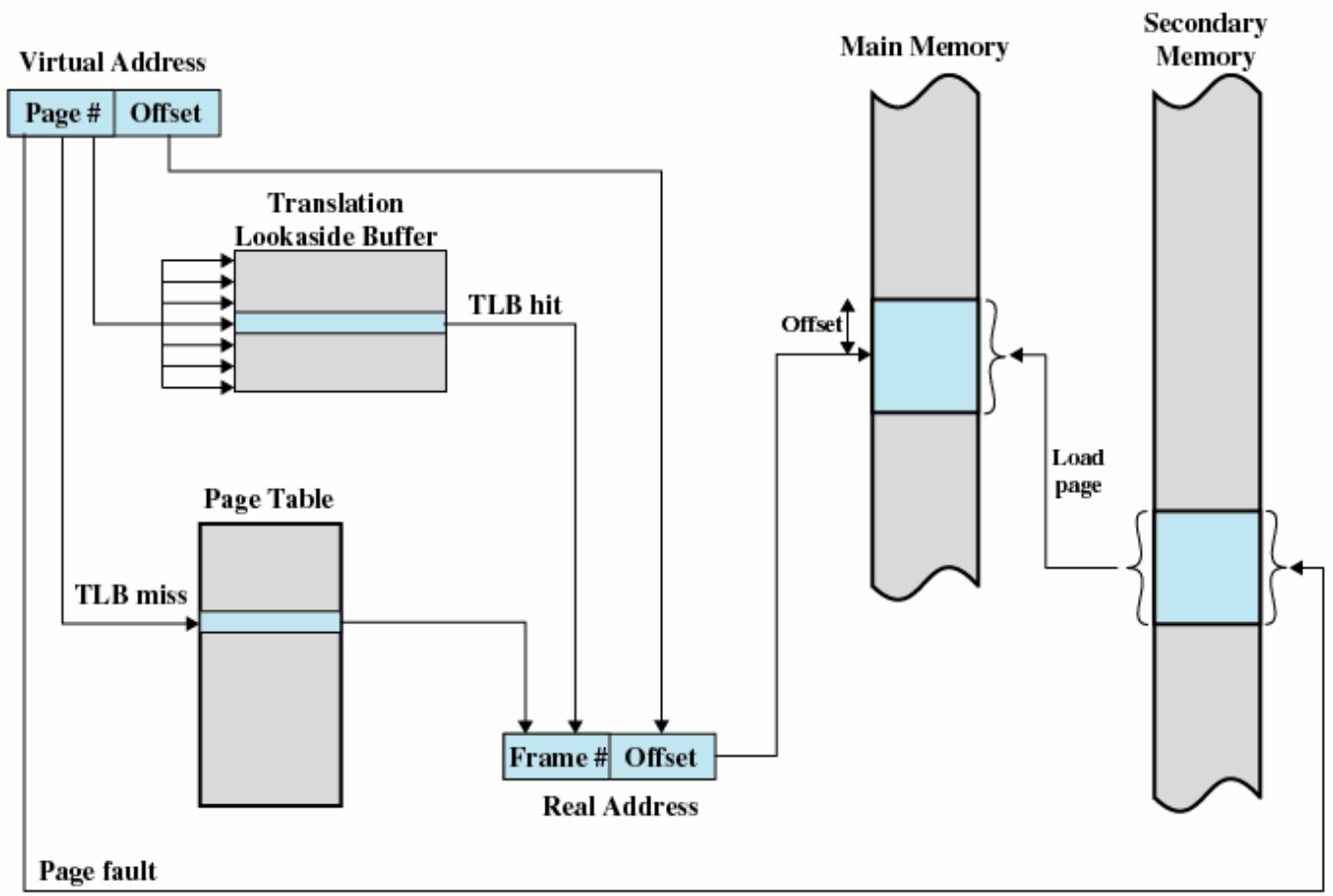
- 32 bit addresses, If 4K (2^{12}) page size, $2^{32}/2^{12} = 2^{20}$ entries, If each PTE is 4 bytes, 4 MB (2^{10}x 2^{10}x4) is required for page table

- The table with 2^{10} pages $\rightarrow$ VM, mapped by a root page table with 2^{10} PTEs $\rightarrow$ 4 KB ($4x2^{10}$) of main memory



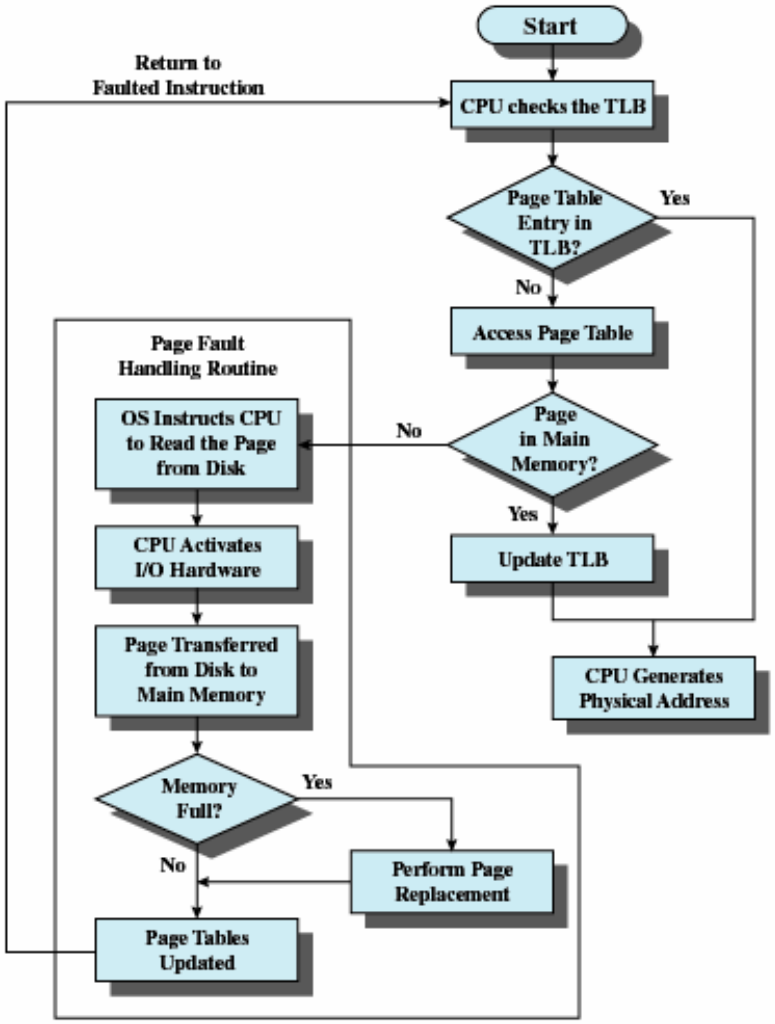
Translation Lookaside Buffer

TLB → Special high-speed cache for PTEs

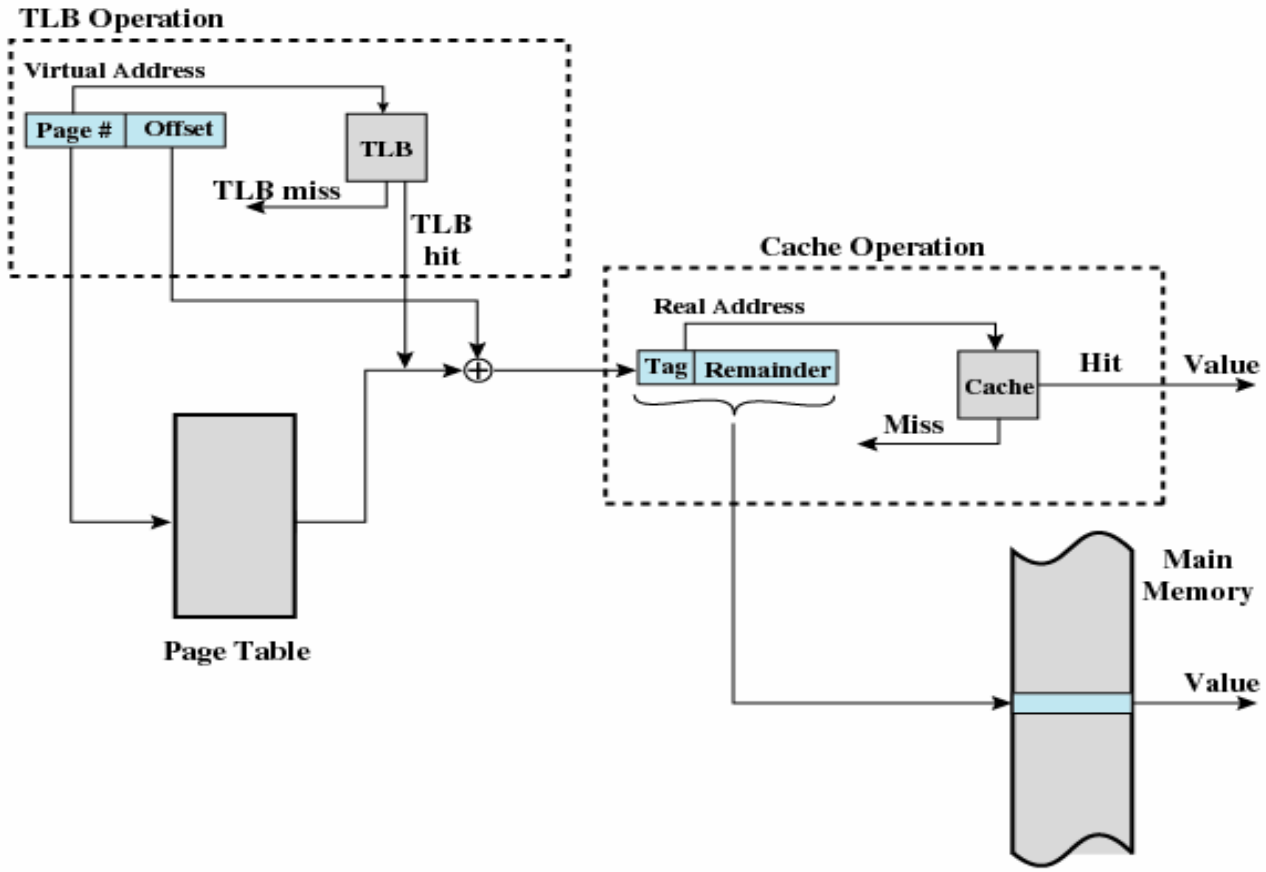


Translation Lookaside Buffer

Operation of Paging and TLB



Cache and TLB Operation



Combined Paging and Segmentation

- Advantages of Paging
  - Any page → Placed in any frame in physical memory
  - Fast to allocate and free
    - Alloc → No searching for suitable free space
    - Free → Doesn't have to coallesce with adjacent free space
    - Just use bitmap to show free/allocated page frames
  - Simple to swap-out portions of memory to disk
    - Page size matches disk block size
    - Can run process when some pages are on disk
    - Add *present* bit to PTE
  - Enables sharing of portions of address space
    - To share a page, have PTE point to same frame

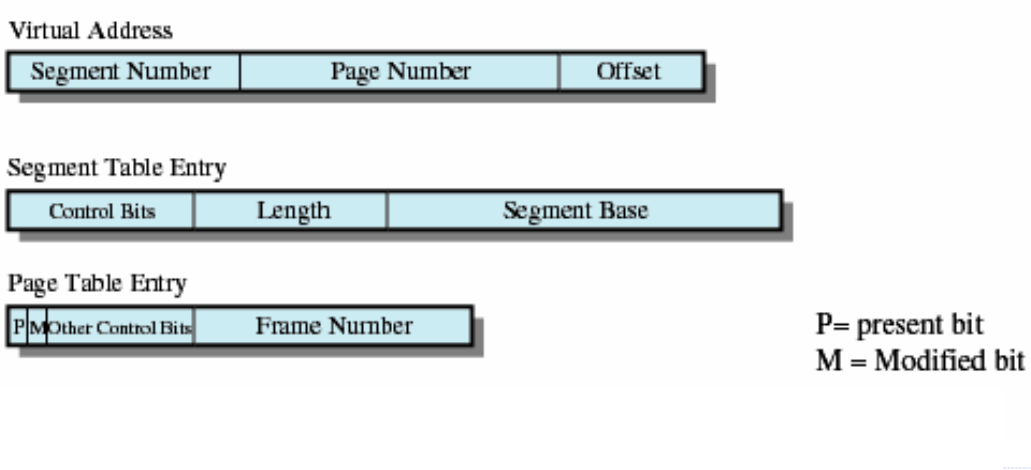
Combined Paging and Segmentation

- Disadvantages of Paging
  - Internal fragmentation
    - Wasted memory grows with larger pages
  - Additional memory reference to look up in page table → Very inefficient
  - Storage for page tables may be substantial
  - Requires PTE for all pages in address space
    - Entry needed even if page not allocated
  - Problematic with dynamic stack and heap within address space

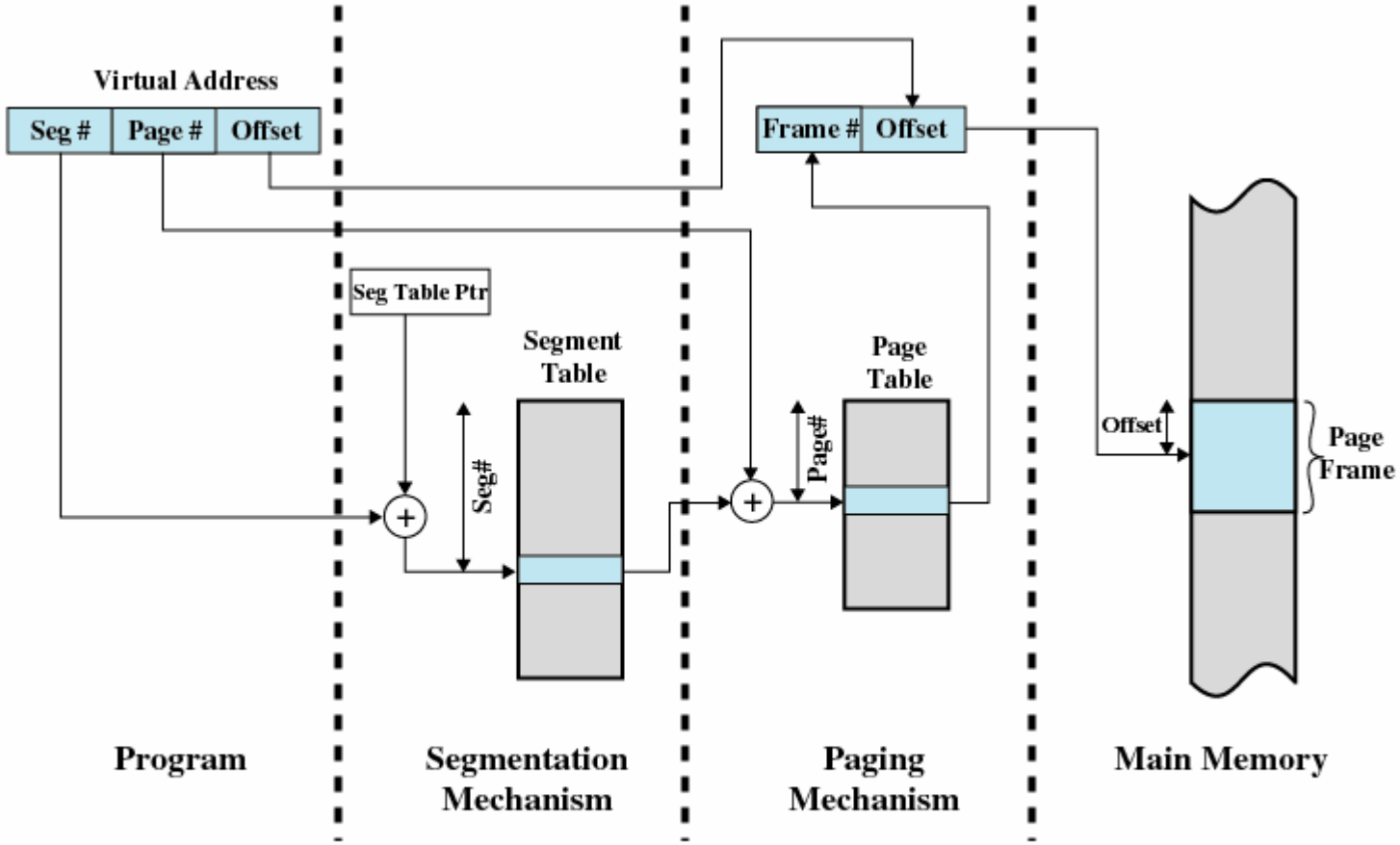
Combined Paging and Segmentation

- Goal → More efficient support for address spaces
  - Divide address space into segments (code, heap, stack)
    - Segments can be variable length
  - Divide each segment into fixed-sized pages
  - Solves the problems of external fragmentation and lengthy search times by paging the segments
    - MULTICS approach

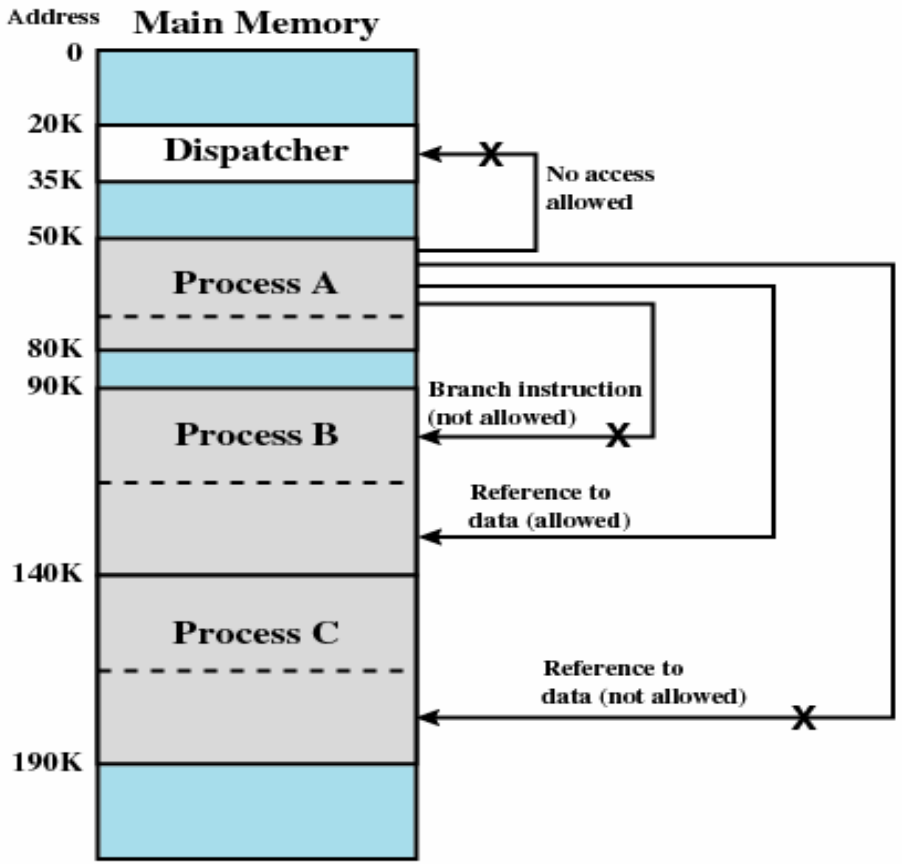
Combined Segmentation and Paging



Combined Segmentation and Paging

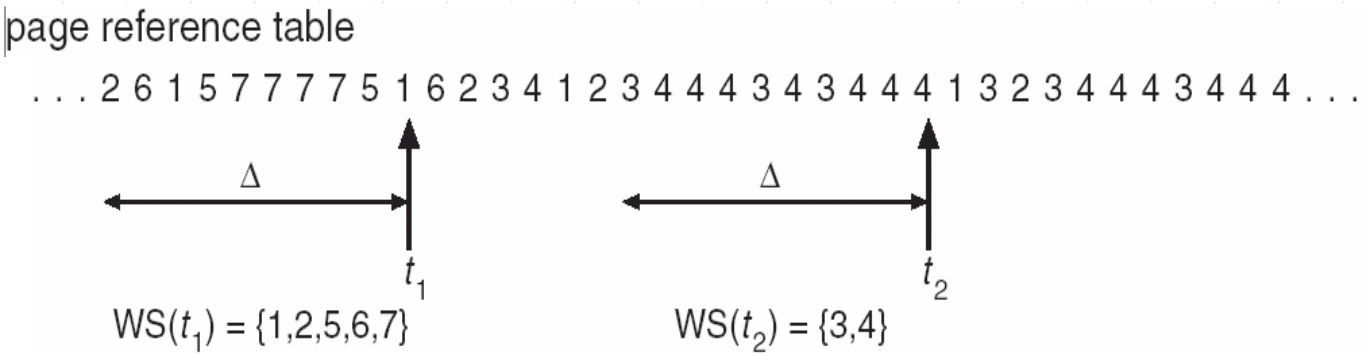


Protection and Sharing



Working Sets

- Appox of program locality → *Working Set* $WS(t,\Delta)$
  - Set of pages in last Δ memory references at time t



  - Accuracy of working set → Δ
    - if Δ too small will not encompass entire locality
    - if Δ too large will encompass several localities
    - if $\Delta = \infty \Rightarrow$ will encompass entire program
  - $D = \Sigma\ WSS_i \rightarrow$ total demand frames
    - if $D > m \Rightarrow$ Thrashing, if $D >$ m, then suspend one of the proces

Working Sets

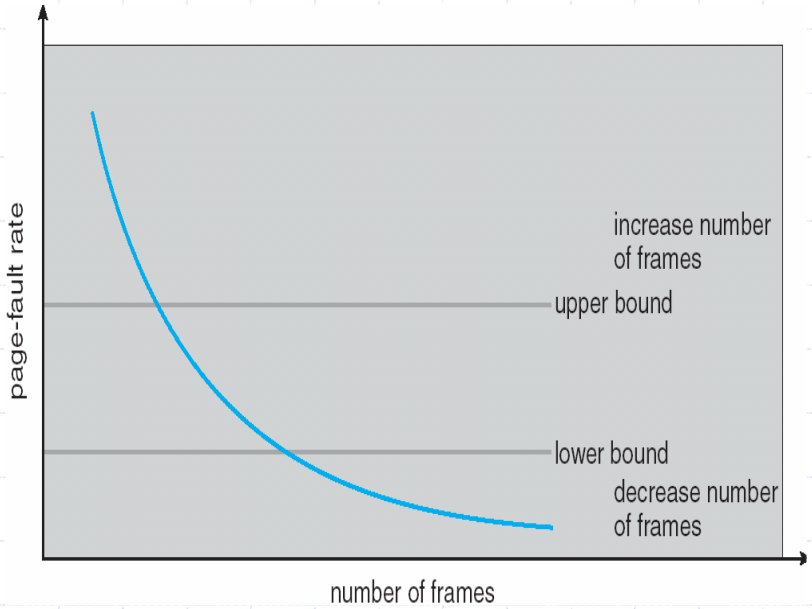
- Denning's *working set* principle
  - A program should run if its working set is in memory
  - A page may not be victimized if it is a member of the current working set of any runable (not blocked) program
- Keeping track of *working set* - Approx
  - Example Δ = 10,000
    - Timer interrupts after every 5000 time units
    - Keep in memory 2 bits for each page
    - Whenever a timer interrupts copy and sets the values of all reference bits to 0
    - If one of the bits in memory = 1 $\Rightarrow$ page in working set
  - Accurate?

Working Sets

- Software scheme
  - use referenced bit
  - periodically shift the reference bit
  - If OR(bits) = 1, the page is in the $W(t, \Delta)$
- Scheduling
  - Schedule *checker* process together with user processes. As a result, checker will periodically
    - kick out pages not in $W(t, \Delta)$ by using the software scheme above
    - decrease $W(t, \Delta)$ by the number of pages kicked out
  - Page fault handler will increase $W(t, \Delta)$ by one after bringing a page into the memory

Page-Fault Frequency

- Details of working set → High overhead
- Page-Fault Frequency (PFF) → Monitor Threshing
  - PFF → 1/T, T→ Average inter-page fault time



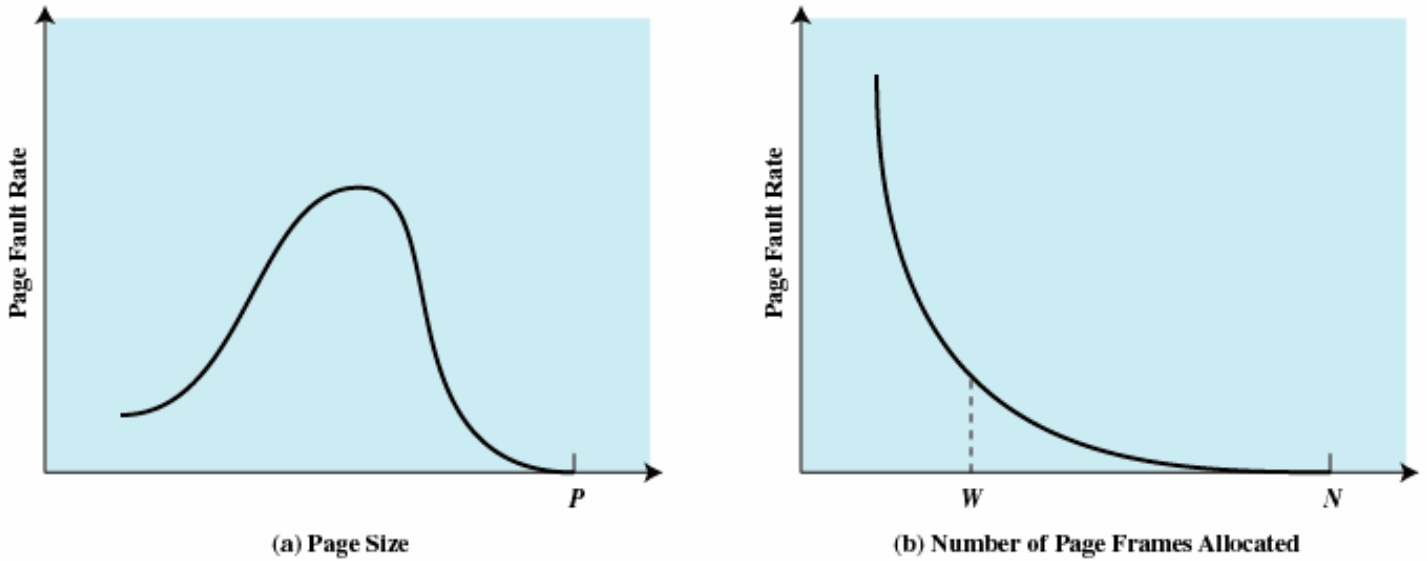
  - PFF > T_U → Increase # pages allocated to the process by 1
  - PFF < T_L → Decrease # pages allocated to the process by 1
  - T_L < PFF < T_U → Replace a page in memory by some other reasonable policy; e.g., LRU

Page Size

- Smaller page size
  - Less amount of internal fragmentation
  - Greater is the number of pages required *per process*
  - More pages per process → Larger page tables, double page fault

- Larger page size
  - Rotational devices → More efficient block transfer of data

Paging Behavior



Policies for VM

The hardware only provides some basic capabilities for virtual memory. The OS must make decisions on the various aspects of memory management.

- **Whether or not use VM techniques**?
- **Use Paging or Segmentation or both?**
- **Algorithms for memory management techniques?**
  - Key Issue – Performance
  - Minimize the page faults rate → Considerable software overhead
  - Which page to replace, I/O
  - No policy works the best!
  - Performance of policies → main memory size, relative speed, size and number of processes competing for resources, execution behavior
  - No final answers, smaller OS → policies that are good for wide range of conditions, larger systems → monitoring and control tools that allow tuning at site

Policies for VM

- **Allocation:** How much real memory should be allocated to each (ready) program?
- **Fetching:** When a page should be bought into main memory?
- **Placement:** Where in real memory the fetched page should be loaded?
- **Replacement:** Which page in memory should be replaced when a new page must be bought in?

Allocation Policy

Conflicting Requirements

- If fewer frames are allocated for a program
  - Page fault rate ↑
  - More programs can reside in memory, decreasing the need for swapping
- Allocating additional frames to a program beyond a certain number
  - Results in little or only moderate increase in performance

The number of allocated pages (*resident set size*) can be *fixed or variable* during the execution of program

Fetch Policy

When pages should be bought into main memory?

- **Demand Paging**
  - Bring into main memory only when a reference is made
  - Principle of locality $\rightarrow$ most future references will be pages that have recently been bought in
  - Most common approach used in paging systems
- **Prepaging**
  - Pages other than the one demanded by a page fault are bought in
  - Exploits the seek time and rotational latency of disks, but its utility has not been established

Cleaning Policy

When a modified page should be written out to secondary memory?

- Demand Cleaning → CPU utilization
- Pre Cleaning → Modified again
- Better approach → Page buffering
  - Replaced pages are placed in two lists
    - Modified and Unmodified
  - Modified list → Periodically written out in batches
  - Unmodified list → Pages are either reclaimed if referenced again or lost when its frame is assigned to another page

Placement Policy

Where in real memory a process piece should reside?

This policy usually follows the rules about paging/segmentation discussed earlier

- In pure segmentation system, policies such as *best-fit*, *first-fit*, *etc*., discussed earlier are possible alternatives
- In systems with pure paging or paging combined with segmentation, placement is straightforward using address translation and memory access hardware

Replacement Policy

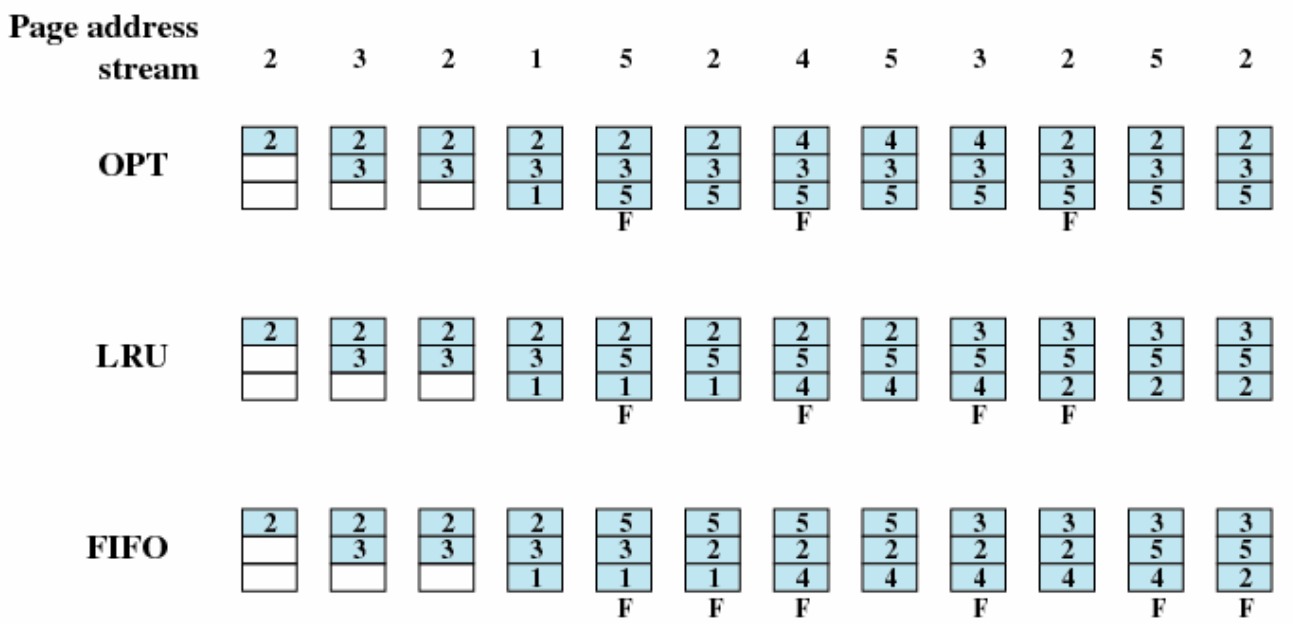
Selection of page in main memory to be replaced when a new page must be bought in. Most policies try to predict future behavior based on past behavior.

Frame Locking → If a frame is locked it may not be replaced, frame locking bit

Examples: Kernel of OS, I/O buffers, etc.

- **OPT**
  - The page for which the time to NEXT reference is longest is replaced
  - Impossible to implement → OS cannot have perfect knowledge of future events
  - Used as standard to judge others
- **FIFO**
  - Pages are treated a circular list, longest resident (oldest) page is replaced
  - Simplicity, Replaces the page that has been longest, Reasoning?
- **LRU**
  - Page in the memory that has NOT been referenced for longest time is replaced
  - Nearly as OPT, Problem → Implementation

Example - Page Replacement Algorithms



Clock Policy

- LRU → Nearly as well as optimal policy
  - Difficult to implement → Significant overhead
- FIFO → Simple to implement but performs poorly
- No of other algorithms → Appox LRU performance with little overhead
  - Variants of scheme referred to as Clock Policy

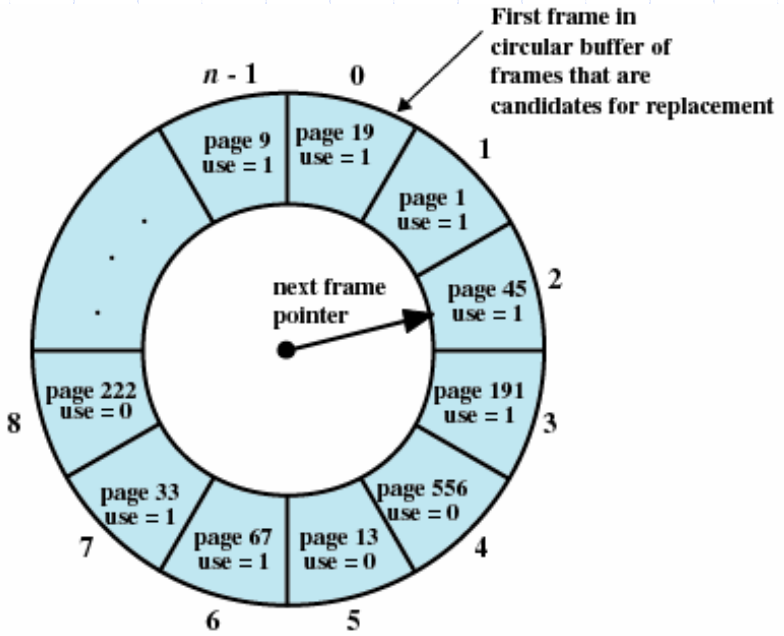
Clock Policy

Requires the association of an additional bit with each frame → *use bit*

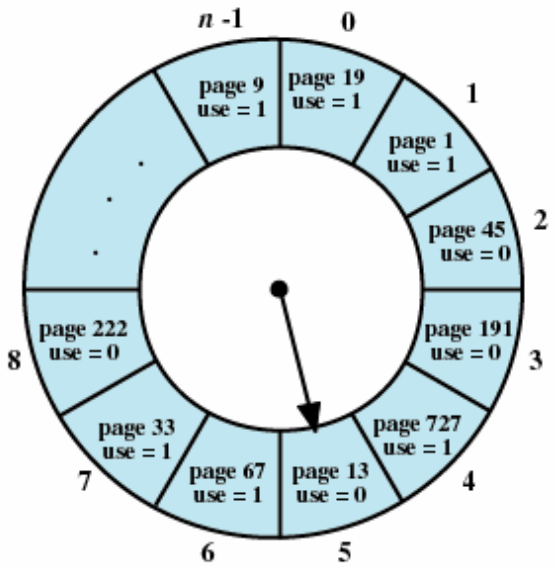
All the frames, along with *use bit* are kept in a circular queue

- When a page is first loaded into frame in memory → *use bit* is set to 1
- When a page is subsequently referenced → *use bit* is set to 1
- When a frame is needed, the pointer is advanced to first frame with a zero *use bit*
- As the pointer advances, it clears the *use bit* (1 is changed to 0)
- This algorithm is also known as 2nd chance algorithm

Clock Policy

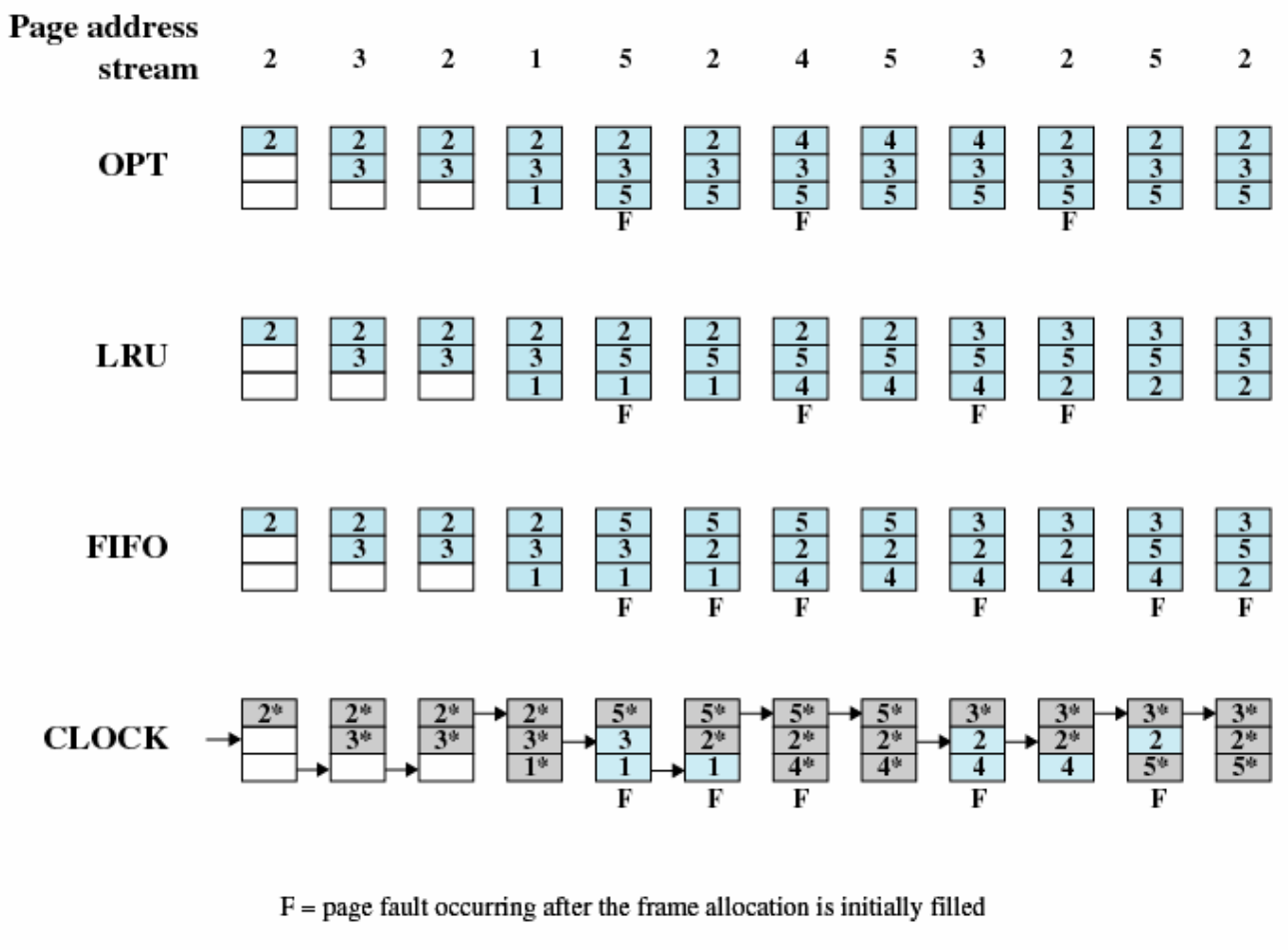


(a) State of buffer just prior to a page replacement

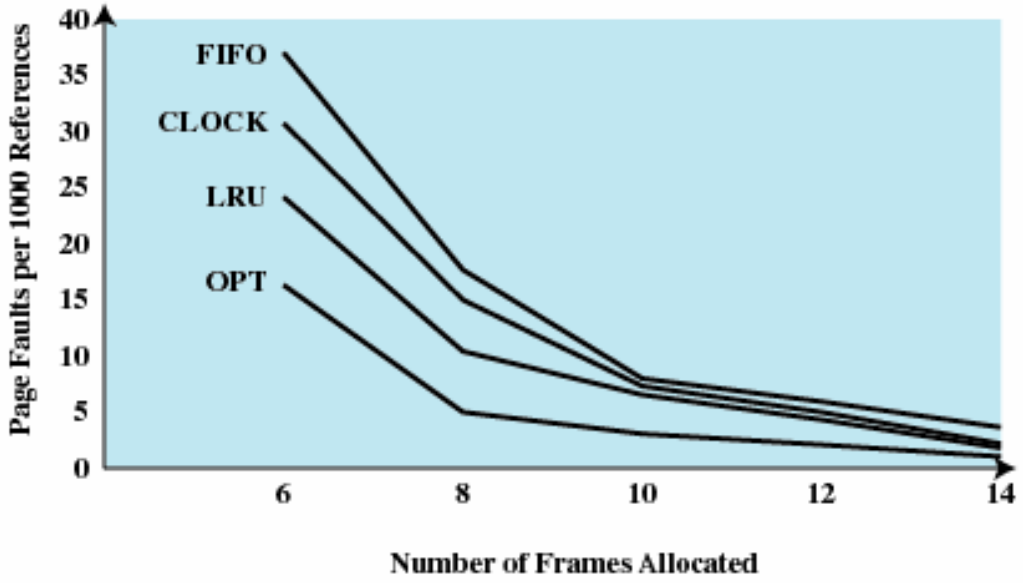


(b) State of buffer just after the next page replacement

Example - Page Replacement Algorithms



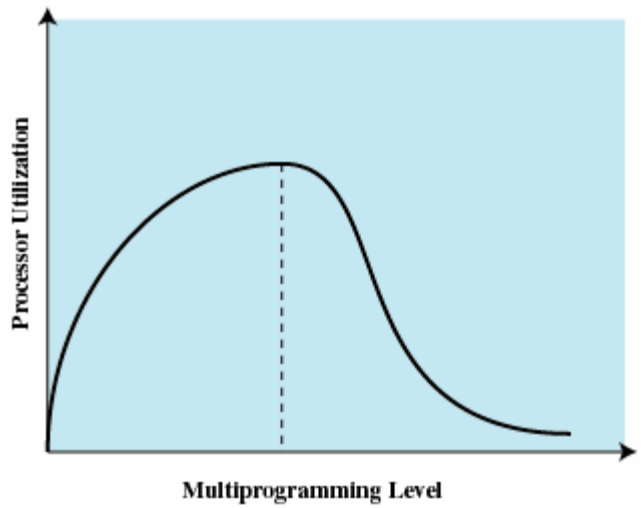
Comparison of Placement Algorithms



Load Control

Determines the number of process that will be resident in the main memory, which is referred to as multiprogramming level.

- If too few process are resident at any time, many occasions when all process are blocked and much time will be spent in swapping.
- Too many process will result in Thrashing.



Load Control

The only way to eliminate thrashing is to reduce the multiprogramming level by suspending one or more process(es). The Victim process(es) can be:

- Lowest priority process
- Faulting process
- Newest process
- Process with smallest resident set
- Process with largest resident set

Operating System Examples

- Windows XP
- Solaris

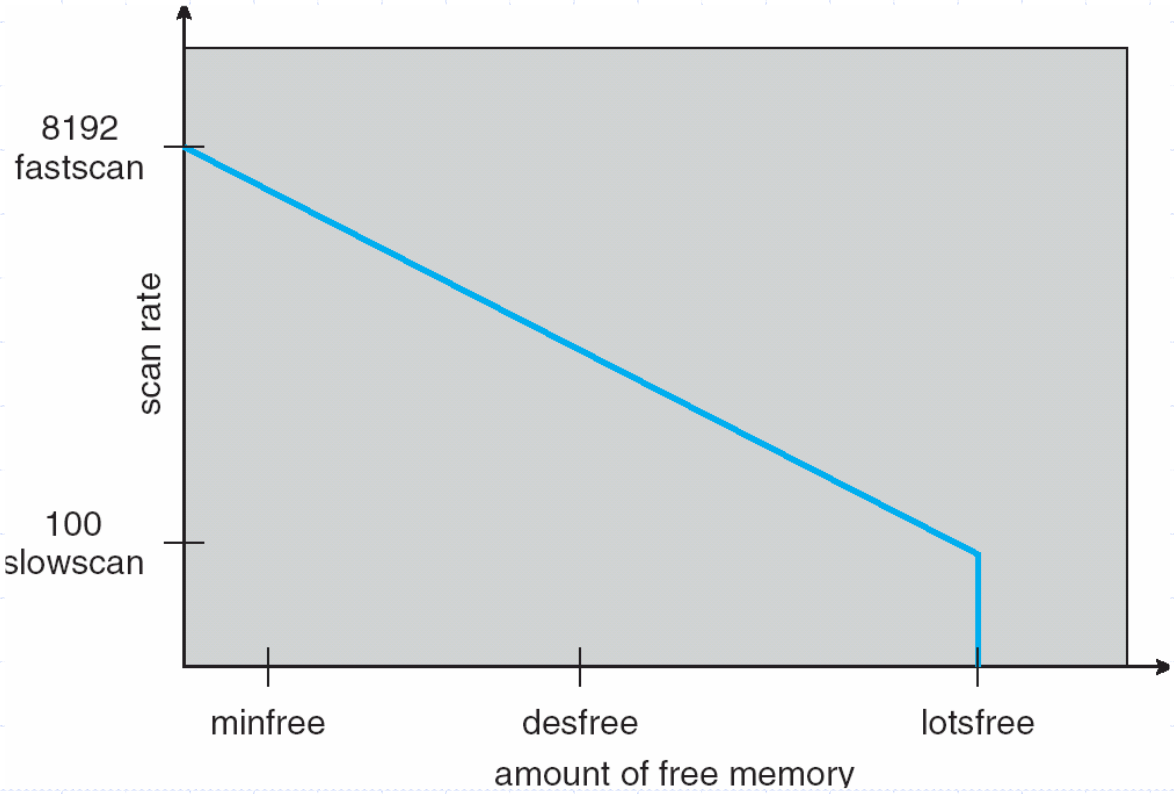
Windows XP

- Demand paging with **clustering**
  - Clustering → brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum** (50-345)
- Amount of free memory in the system ↓
  - Threshold → **automatic working set trimming**
  - How?
    - Single Processor → variation of clock algorithm
    - Multiprocessor → variation of FIFO algorithm

Solaris

- Page fault → Kernel assigns a page to faulting thread
  - Keeps sufficient amount of memory
- Parameters
  - Lotsfree → Threshold parameter, (amount of free memory) to begin paging
  - Desfree → Threshold parameter to increasing paging
  - Minfree → Threshold parameter to being swapping
- Paging is performed by *pageout* process
  - Pageout → scans pages using modified clock algorithm
  - *Scanrate* is the rate at which pages are scanned
  - Ranges from *slowscan* ↔ *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available

Solaris 2 Page Scanner



Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
  - TLB Reach = (TLB Size) $\times$ (Page Size)
- Ideally, the working set of each process $\rightarrow$ TLB; Otherwise there is a high degree of page faults
- Increase the Page Size. This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes. This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.

Other Issues – Program Structure

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

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

128 x 128 = 16,384 page faults

  - Program 2

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

128 page faults