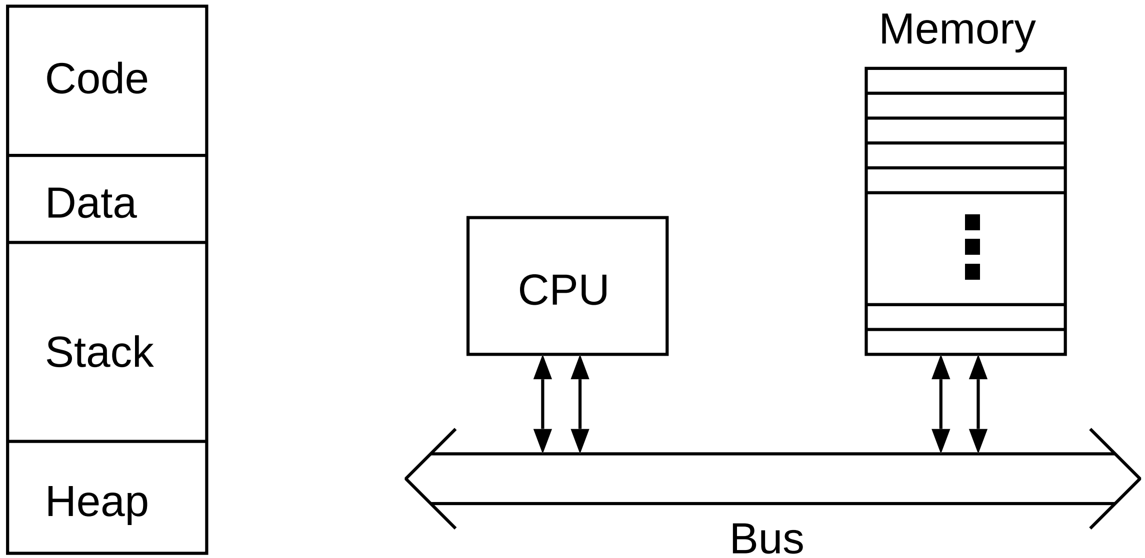
Outline

- Address spaces and address binding
  - compile-time
  - load-time
  - run-time
- Memory management: mapping virtual address to physical addresses
  - contiguous allocation and fragmentation
- Paging
  - paging hardware
  - multi-level and hashed page tables
  - protection and sharing
- Segmentation
- Swapping
- Demand paging
  - page faults
  - page replacement
    - FIFO
    - optimal
    - LRU
    - LRU approximations
    - counting algorithms
- Frame allocation
- Thrashing
- Performance of demand paging: issues and improvements

Address spaces



Program address space

Physical addresses

Address binding: mapping from one address space to another address space

Address binding

Compile-time binding

- Location of program in physical memory must be known at compile time
- Compiler generates *absolute* code
  - compiler binds names to actual physical addresses
- Loading $\equiv$ copying executable file to appropriate location in memory
- If starting location changes, program will have to be recompiled
- Example: .COM programs in MS-DOS

Address binding

Load-time binding

- Compiler generates *relocatable* code
  - compiler binds names to relative addresses (offsets from starting address)
  - compiler also generates relocation table
- Linker resolves external names and combines object files into one loadable module
- (Linking) loader converts relative addresses to physical addresses
- No relocation allowed during execution

Address binding

Run-time binding

- Programs/compiled units may need to be relocated during execution
- CPU generates relative addresses
- Relative addresses bound to physical addresses at runtime based on location of translated units
- Suitable hardware support required

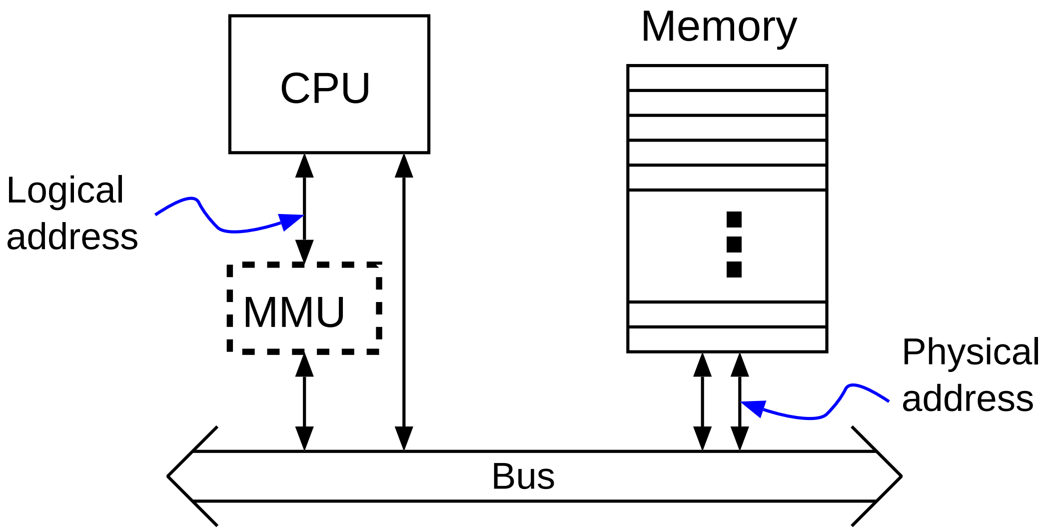
Memory management unit

- **Logical/virtual address:** address generated by CPU
- **Physical address:** address seen by memory hardware
- Compile-time / load-time binding $\Rightarrow$ logical address $=$ physical address

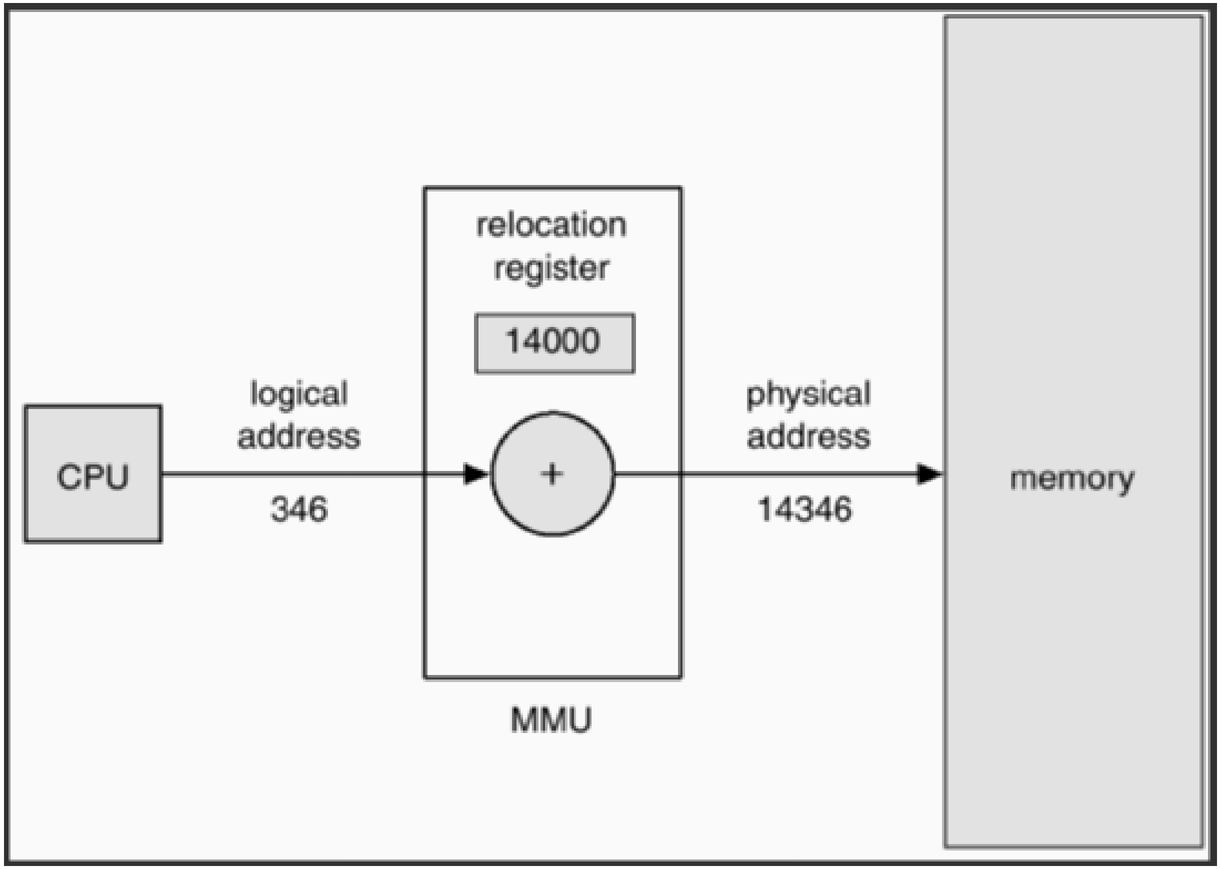
  Run-time binding $\Rightarrow$ logical address $\neq$ physical address

Section 8.2

MMU: h/w device that maps virtual addresses to physical addresses at run time
(also called *address translation hardware*)

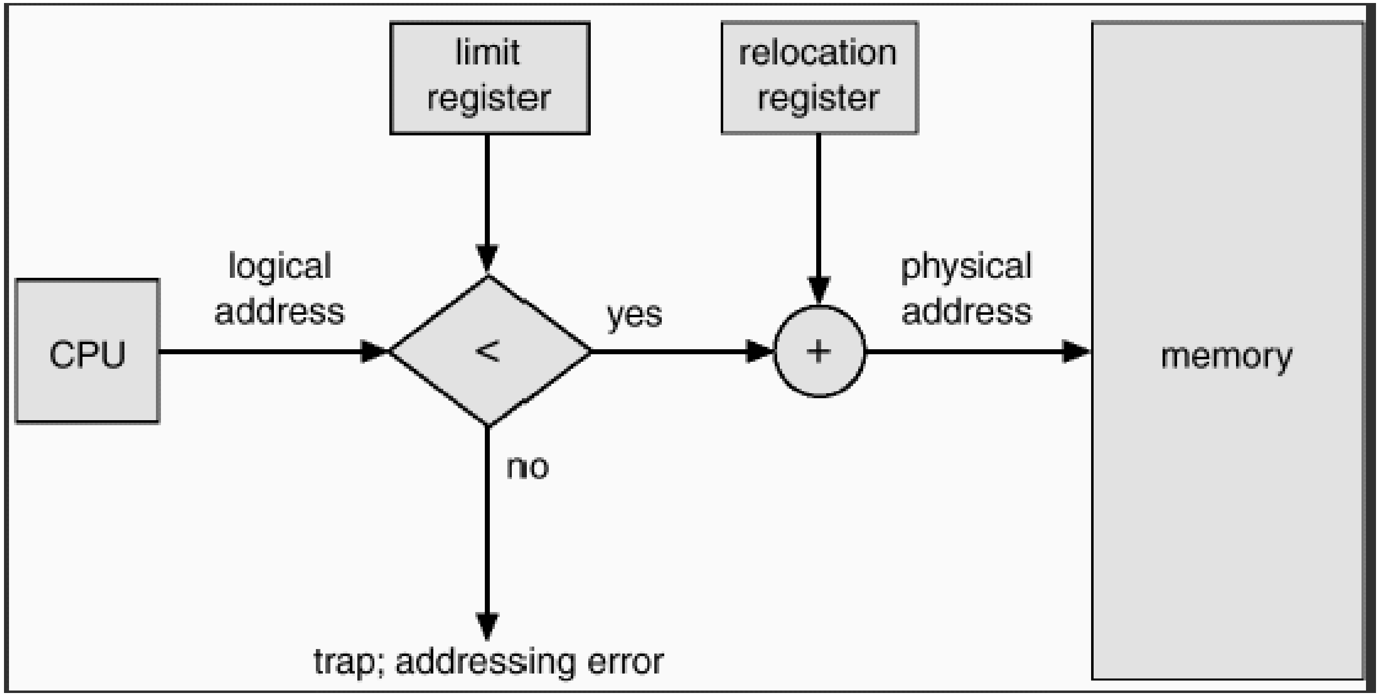


MMU



- Kernel loads relocation register when scheduling a process

Memory protection



- Prevents process from accessing any memory outside its own address space
- Allows OS size to change dynamically
  - *transient* code (code/data corresponding to infrequently used devices / services) may be removed from memory when not in use

Contiguous allocation

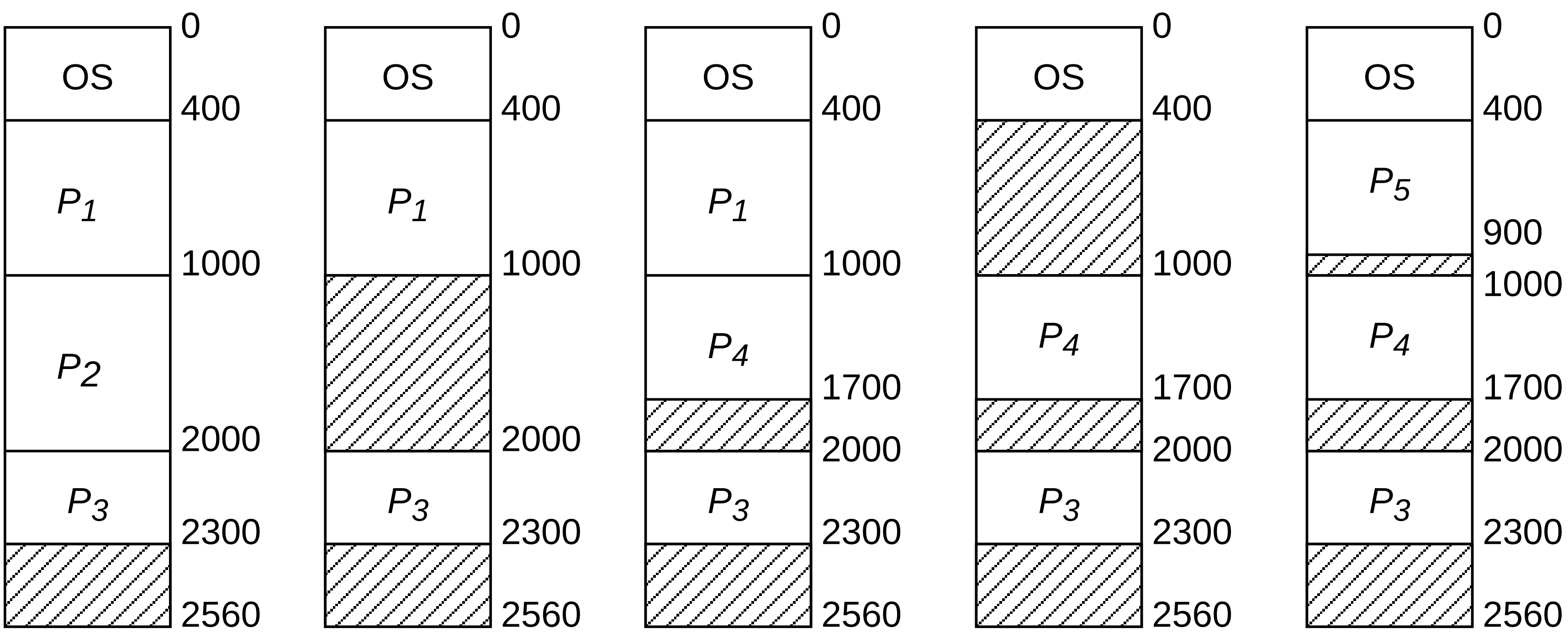
- Memory is divided into variable-sized partitions
- OS maintains a list of allocated / free partitions (*holes*)
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Memory is allocated to processes until requirements of next process in queue cannot be met
  - OS may skip down the queue to allocate memory to a smaller process that fits in available memory
- Hole allocation policies:
  - **First-fit:** allocate the first hole that is big enough
  - **Best-fit:** allocate the smallest hole that is big enough
    - entire free list has to be searched unless sorted
  - **Worst-fit:** allocate the largest hole
- When process exits, memory is returned to the set of holes and merged with adjacent holes, if any

Contiguous allocation

Example:

Process sizes:

P_1 600 P_2 1000 P_3 300 P_4 700 P_5 500



Fragmentation

- **External fragmentation:** memory space to satisfy a request is available, but is not contiguous
  - may be reduced slightly by allocating memory from appropriate end (top/bottom) of hole
- **Internal Fragmentation:** allocated memory may be larger than requested memory
  $\Rightarrow$ memory within partition may be left unused
  - may be used to avoid overhead required to keep track of small holes

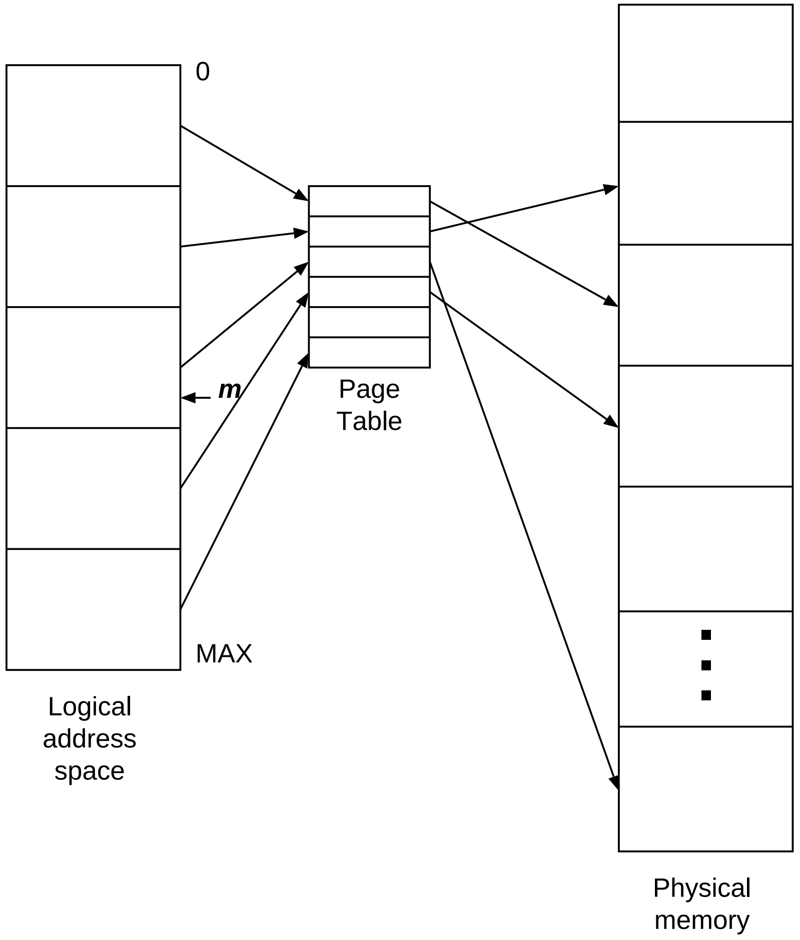
Compaction

- Memory contents shuffled to place all free memory together in one large block
- Reduces external fragmentation
- Dynamic relocation (run-time binding) needed

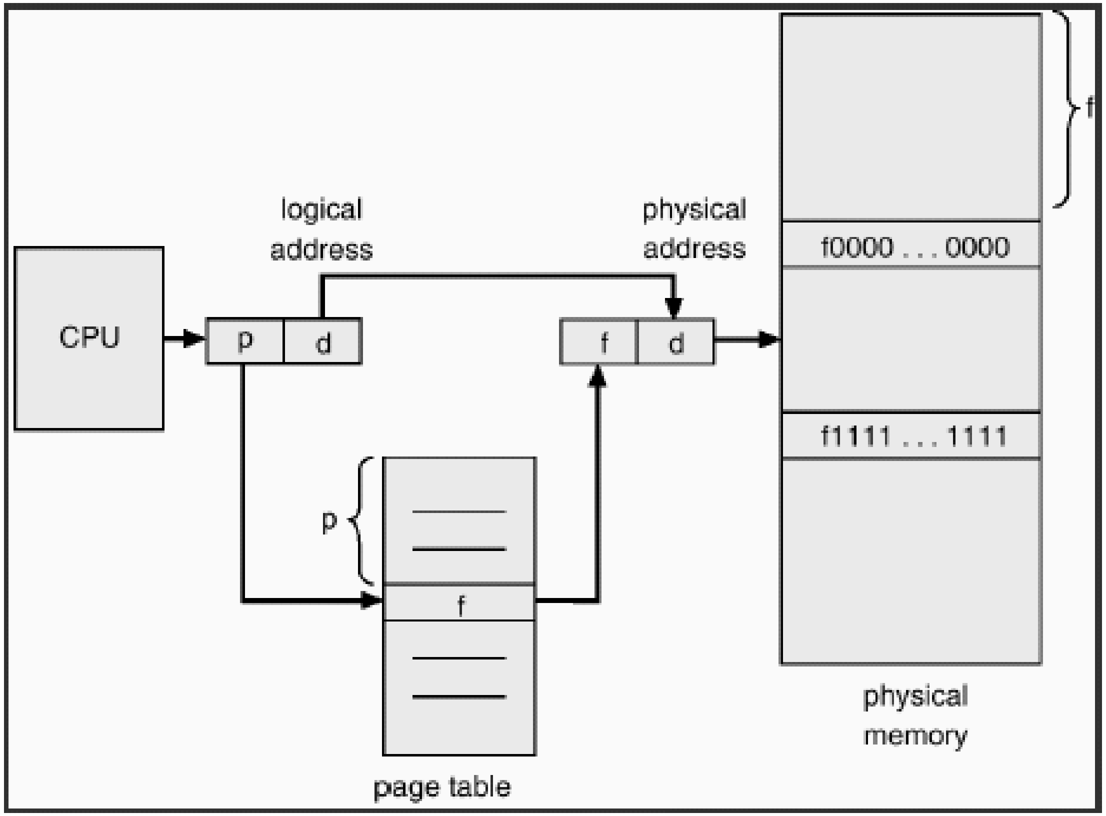
Paging

- Physical memory is partitioned into fixed-size *frames*
- Frame size:
  - defined by hardware
  - should be power of 2
  - typically 512–8192 bytes
- Logical address space is partitioned into *pages* (same size as frames)
- When a process with n pages has to be loaded, n free frames have to be found
- Kernel keeps track of free frames
- Page table translates logical page #s to physical frame addresses

Section 8.5.1



Paging



Let 2^m = size of logical address space

2^n = size of page

Then p = $m - n$ higher order bits of logical address

d = n lower order bits of logical address

Paging

- Page table:
  - part of process context
  - during context switch, saved page table is used to reconstruct hardware page table
  - may be used by some system calls to translate logical addresses to physical addresses in software
- Frame table:
  - maintained by kernel
  - contains 1 entry per physical page frame
    - whether free or allocated
    - allocation information (PID, page#)

Paging

Miscellaneous issues:

- Memory protection is automatic
  - process cannot address memory outside its own address space
- Fragmentation:
  - No external fragmentation
  - Internal fragmentation can happen
    - half a page per process, on average
- Page/frame size:
  - Small frames $\Rightarrow$ less fragmentation
  - Large frames $\Rightarrow$ page table overhead $\downarrow$; I/O is more efficient

Paging hardware

I. Special purpose registers:

- Page table is stored in a set of dedicated, high-speed registers
- Instructions to load/modify PT registers are privileged
- Acceptable solution if page table is small
- Example: DEC PDP-11
  - 16-bit address space
  - 8K page size
  - page table contains 8 entries

Section 8.5.2

Paging hardware

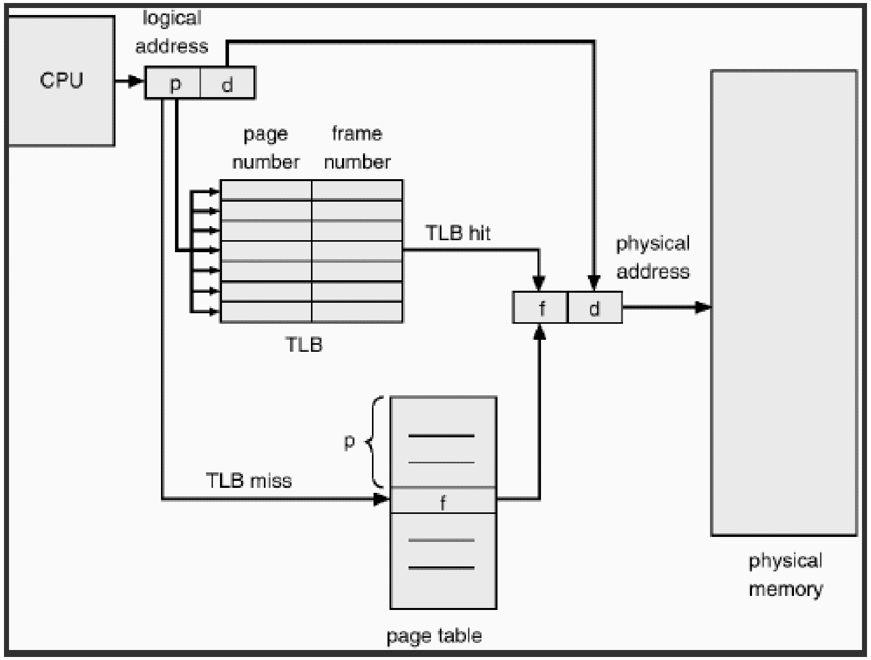
II. Memory + PTBR:

- Needed for large page tables
- PT stored in main memory
- Base address of PT is stored in *page table base register* (PTBR)
  Length of PT is stored in *page table length register* (PTLR)
- Context switch involves changing 1 register only
- Two physical memory accesses are needed per user memory access
  $\Rightarrow$ memory access is slowed by factor of 2

Paging hardware

III. Associative registers/Translation look-aside buffer (TLB):

- TLB $\equiv$ small, fast-lookup hardware cache, built using high-speed memory (expensive)
  - each register holds key + value
  - input value is compared *simultaneously* with all keys
  - on match, corresponding value is returned



- TLB holds subset of page table entries
- TLB hit $\Rightarrow$ additional overhead may be 10% or less
  TLB miss $\Rightarrow$ new $\langle$ page#, frame# $\rangle$ added to TLB
- TLB has to be flushed on context-switch

Paging hardware

- **Hit ratio:** percentage of times that a page# is found in TLB
  - depends on size of TLB
- Effective memory access time: average time for a memory access (including TLB lookup)

  Example:

  TLB lookup: 20ns Memory access: 100ns Hit ratio: 80%

  Effective access time = $0.8 \times 120 \ + \ 0.2 \times 220$ = 140ns

Multi-level paging

- Logical address spaces are usually very large (2^{32} or 2^{64})
  $\Rightarrow$ page tables are very large (how large?)
  $\Rightarrow$ page tables should/can not be allocated contiguously
- Two-level paging:
  - First level (inner) page table is broken into pieces
  - Second level (outer) PT entries point to memory frames holding the pieces of the first level PT

  Example:

$\leftarrow$ page # $\rightarrow$		$\leftarrow$ offset $\rightarrow$
p_1	p_2	d
10 bits	10 bits	12 bits

- 3-, 4-, ... level paging may be required for certain architectures
- Performance: TLB miss $\Rightarrow$ upto 4 extra memory accesses

Section 8.5.3

Memory protection

- Protection bit(s) associated with each frame (via page table entry)
  - protection bit specifies read-only / read-write access
  - protection bit checked in parallel with address computation
  - protection violation (writing to read-only page) causes hardware trap to OS
- Valid/invalid bit indicates whether page is in the process' logical address space
  - set by OS for each page
  - may be used to implement process size restrictions

Section 8.5.2

Sharing pages

- Primarily used for sharing *reentrant* (read-only) code for heavily used programs
  e.g. common utilities, text editors, compilers, window/desktop managers
  NOTE: data for separate processes are stored separately
- PT for each process running a shared program maps code pages to the same physical frames
- Data pages are mapped to different physical frames

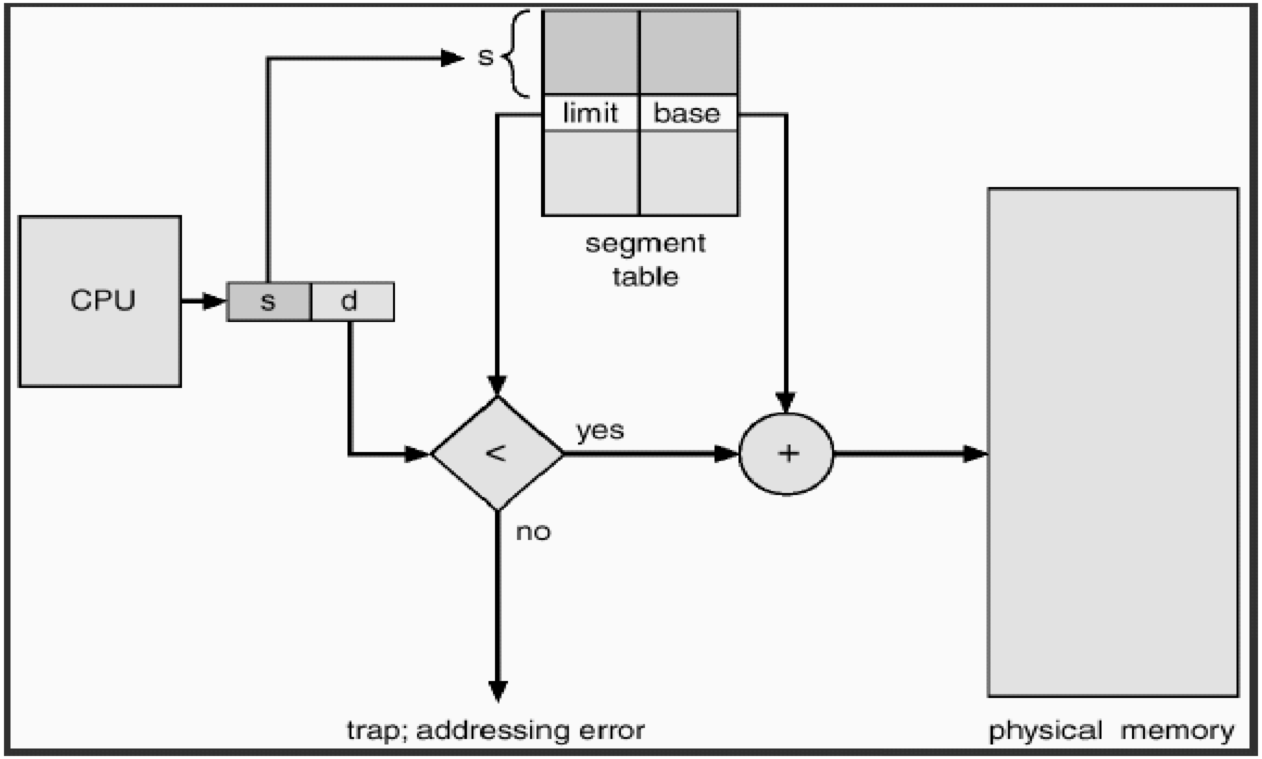
Segmentation

- Intuitively, address space $\neq$ linear array of bytes
- Address space is made up of variable-sized logical **segments**
  e.g. main function, subroutines, some data structures (list, array, stack, etc.), . . .
- Segments are not necessarily ordered
- Elements within a segment are ordered
- Each segment is allocated contiguous memory
- Logical addresses specify $\langle$ segment identifier, offset $\rangle$

NOTE: Segments are usually automatically generated by the compiler

Segment Table

- Maps 2-dimensional logical addresses to 1-dimensional physical memory addresses
- Segment table entry = $\langle$ segment base, segment limit $\rangle$
  Base = starting physical address of segment in memory
  Limit = size of segment



Segmentation

Segment tables:

- Can be stored in fast registers / memory
  - STBR: points to segment table in memory
    STLR: length of segment table
  - ARs hold the most recently used segment-table entries

Protection/sharing:

- Each segment has associated protection/permission bits
- Memory mapping hardware checks protection bits to prevent illegal memory accesses
  - hardware checks can be used to enforce automatic bounds on array indices
- 1 or more segments can be shared between processes by setting segment table entries to point to the same physical location
  - shared code segments should be assigned the same segment # in all processes

Fragmentation:

- Segments are variable-sized $\Rightarrow$ external fragmentation may happen
  - if average segment size is small, fragmentation is low

Swapping

Motivation:

Consider the following situation:

- $P_1, \ldots, P_n$ are resident in memory and occupy all available memory
- P_i forks to create a child

Swapping

Motivation:

Consider the following situation:

$P_1, \ldots, P_n$ are resident in memory and occupy all available memory

P_i forks to create a child

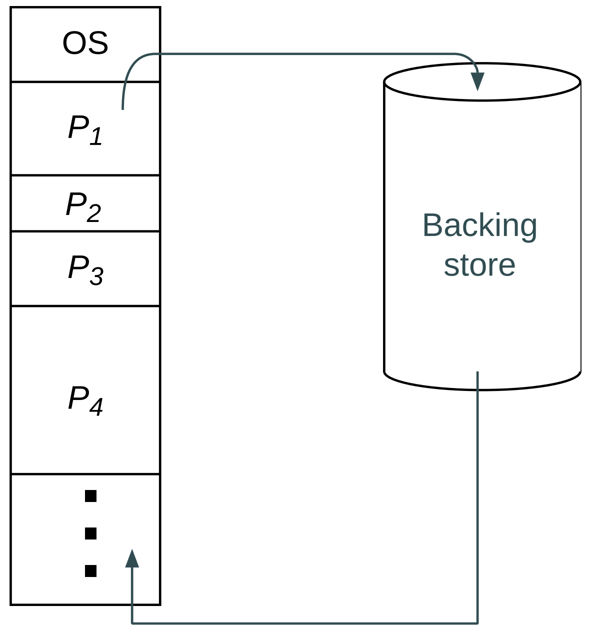
Principle:

- Space on fast disk (also called **Backing Store**) is used as additional / secondary memory
- Process can be *swapped out* temporarily from main memory to backing store; released memory is used for some other process; swapped process is *swapped in* later for continued execution

Swapping

Choosing processes:

- Round-robin
  - when P's quantum expires, it is swapped out, P' is swapped into freed memory
  - scheduler allocates next quantum to some other process in memory



- Priority-based (**roll out, roll in**)
  - when higher priority process arrives, lower-priority process is swapped out
  - when higher priority process finishes, lower priority process can be swapped in

Swapping

Performance:

- Context switch time increases ($\because$ disk transfer is involved)
- Time quantum should be large compared to swap time for good utilization

Example:

Process size: 100K   Transfer rate: 1Mbps
$\Rightarrow$ swap-out + swap-in time = 200ms $(+\ \varepsilon)$

Swapping

Input/output:

- If P is swapped out while waiting for input into buffer in user memory, addresses used by I/O devices may be wrong
- Solutions:
  - process with pending I/O should never be swapped, OR
  - I/O operations are always done using OS buffers (data can be transferred from OS to user buffer when P is swapped in)

Compaction:

1. Processes which have to be moved are swapped out
2. Memory is compacted by merging holes
3. Swapped-out processes are swapped in to different memory locations to minimize fragmentation

Virtual memory

Section 9.1

Background:

- Instructions being executed /addresses being referenced must be in main memory
- Entire logical address space does not have to be loaded into memory
  - some code may be executed rarely
    e.g. error handling routines for unusual error conditions, code implementing rarely used features
  - arrays/tables may be allocated more memory than required
- Virtual memory $\equiv$ mechanism to allow execution of processes without requiring the entire process to be in memory

Virtual memory

Advantages:

- Programs can be larger than physical memory
- More programs can be run at the same time
  $\Rightarrow$ throughput / degree of multiprogramming increases without increase in response time
- Less I/O is needed for loading/swapping
  $\Rightarrow$ programs may run faster (compared to swapping)

Demand paging

- Processes reside on secondary memory (high-speed disk)
- When process is to be executed, only the needed pages are brought into memory (**lazy swapping**)
- Page table should specify location of pages (memory vs. on-disk)
  - valid/invalid bit may be used
  - for page that is not currently in memory, page table entry may contain address of page on disk
- While process accesses pages resident in memory, execution proceeds normally
- When process accesses page not in memory, paging hardware traps to OS (**page fault**)

NOTE: *Swapper* manipulates entire processes

Pager copies individual pages to/from swap space

Section 9.2

Page faults

1. Check internal table to determine whether reference was to valid / invalid page.
2. Invalid access $\Rightarrow$ terminate process.
3. Find a free frame from the free-frame list.
4. Read the desired page from swap device into the free frame.
5. When I/O is complete, update internal table and page table.
6. Restart the instruction that was interrupted by the illegal address trap.
   (state/context of the process is saved so that process can be restarted in exactly the same state)

Restarting instructions

Page Fault	Handling
Instruction fetch	Re-fetch instruction
Operand fetch	1. Re-fetch instruction. 2. Decode instruction. 3. Fetch operand.
`ADD A B C`	1. Fetch, decode instruction 2. Fetch A, B. 3. Add A,B; store sum in C.

Problems:

- `MVC` (IBM System 360/370)
  - moves upto 256 bytes from one location to another
- Auto-increment/auto-decrement addressing modes

Page replacement

Section 9.4

Motivation:

- Pure demand paging: pages are not brought into memory until required
  (process starts executing with no pages in memory)
- Overallocation $\Rightarrow$ free frame list may be empty when a page fault occurs

Method:

1. Find the location of the desired page on disk.
2. Find a free frame. If there is no free frame:
   (i) use page replacement algorithm to select *victim* frame;
   (ii) write victim page to disk; change page/frame tables accordingly.
3. Read the desired page into the (newly) free frame.
4. Update the page and frame tables; restart the process.

Modify/dirty bit

- Modify/dirty bit is associated with each page (via PT)
- Set whenever the page is written
- If dirty bit of victim frame is clear, it is not written to disk
- Reduces time to service page faults
- Also applicable to read-only pages

Page replacement algorithms

- Page replacement algorithm should yield low page-fault rate
- **Reference string:** sequence of memory references
  - used to evaluate PR algorithms
  - may be generated artificially, or by tracing a process
  - memory references are in terms of page #s only
  - sequence of successive references to the same page may be replaced by only one reference
- # of frames allocated to a process $\uparrow \Rightarrow$ page faults $\downarrow$

Section 9.5

FIFO

- Pages are kept in a FIFO queue
  - when a page is brought into memory, it is added at tail of queue
  - when a page has to be replaced, page at head of queue is selected
- Example:

  Reference string: 1 2 3 4 1 2 5 1 2 3 4 5

  # of frames: 3

  # of page faults: 9
- **Belady's anomaly:**

  # of frames allocated to a process $\uparrow \not\Rightarrow$ page faults $\downarrow$

Stack algorithms:

- Pages in memory with n frames $\subseteq$ Pages in memory with $n+1$ frames
- Never exhibit Belady's anomaly

Optimal algorithm

- Replace page that will not be used for the longest period of time
- Minimizes the number of page faults for a fixed number of allocated frames
- Not implementable
- Used to measure other replacement algorithms

LRU algorithm

- Replace page that has not been used for the longest time
- Often used in practice
- Disadvantage: usually requires substantial hardware assistance

Counter implementation:

- Each PT entry contains a time-of-use (counter) field
- On each memory reference, a clock/counter is incremented; counter is copied into the PT entry for the referred page
- When a page has to be replaced, page with the smallest counter is selected
- Disadvantages:
  - each memory reference requires a write to memory
  - entire page table has to be searched to find LRU page
  - counter overflow has to be handled

LRU algorithm

Stack implementation:

- page numbers are maintained in a doubly-linked stack with *head* and *tail* pointers
- on a page reference, the corresponding PT entry is moved to top of stack
  - six pointers have to be changed
- *tail* points to LRU page

LRU approximation algorithms

Background:

- Many architectures do not provide hardware support for true LRU page replacement
- Approximate versions of LRU have to be implemented with the limited hardware support

Reference bit:

- Associated with each PT entry
- All reference bits are initially cleared by OS
- Set by hardware on each page reference
  $\Rightarrow$ distinguishes used pages from unused pages

LRU approximation algorithms

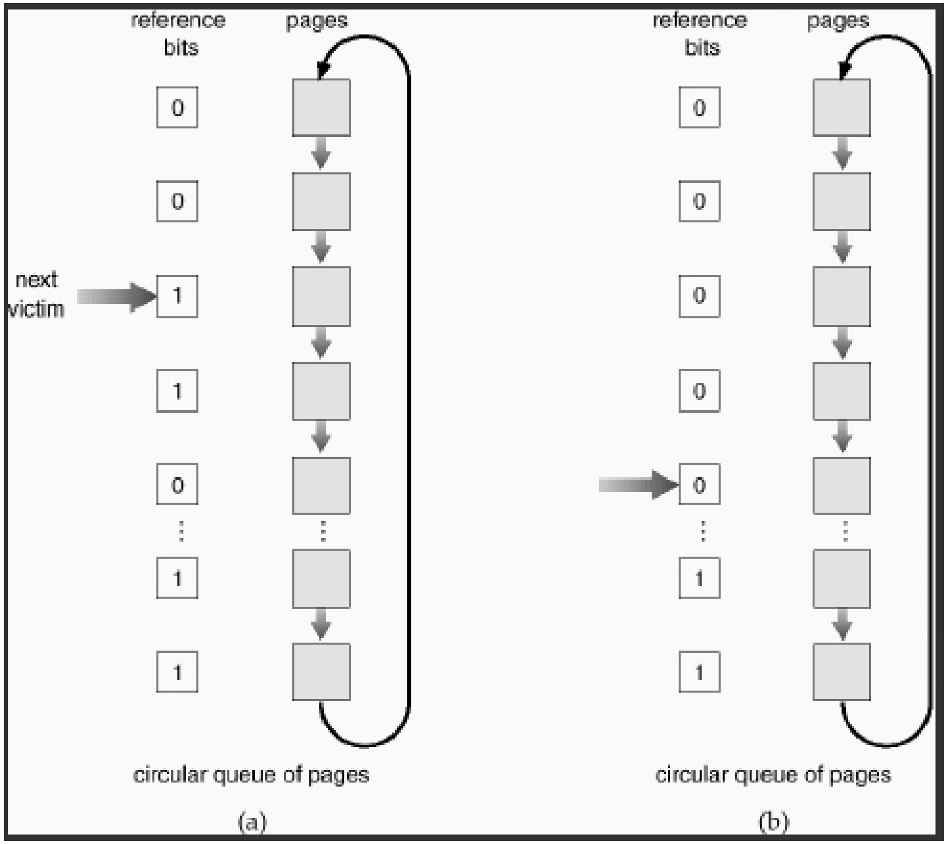
I. Additional-reference-bits algorithm:

- 1 reference byte associated with each PT entry
- On each timer interrupt: reference byte is right-shifted; reference bit is copied into high-order bit of reference byte and cleared
- Reference bytes contain history of page use for 8 most recent intervals
- Reference bytes order PT entries in LRU order (ties may be broken using FIFO ordering)

LRU approximation algorithms

II. Second-chance/clock algorithm:

- Store PT entries in a FIFO queue
- If reference bit of selected page is set:
  - clear reference bit
  - set arrival time to current time
  - continue to next page in FIFO order



- If all bits are set, second-chance replacement reduces to FIFO replacement

LRU approximation algorithms

III. Enhanced second-chance algorithm:

- $\langle$ ref bit, dirty bit $\rangle$ considered as an ordered pair
  - $\langle 0, 0 \rangle$ – best page to replace
  - $\langle 0, 1 \rangle$ – not recently used, but modified (has to be written to disk)
  - $\langle 1, 0 \rangle$ – recently used, but clean (likely to be used again soon)
  - $\langle 1, 1 \rangle$ – recently used and modified
- First page in lowest non-empty class is selected as victim

Counting algorithms

- Each PT entry stores count of the number of references to that page
- **LFU Algorithm:** replaces page with smallest count
  - counter may be right shifted at intervals to form an exponentially decaying average usage count
- **MFU Algorithm:** replaces page with largest count
  - LFU page may have been brought in very recently and is yet to be used
- Performance is not very good

Global vs. local replacement

- Global replacement
  - replacement frame can be selected from all frames (including frames allocated to other processes)
  - generally provides better throughput
- Local replacement: replacement frame can be selected from the frames allocated to the current process

Allocation of frames

Single user system:

- Kernel occupies M frames + some frames for dynamic data structures
- Remaining frames are put on free list for use by a user process

Section 9.6

Multiprogramming:

- Minimum # of frames to be allocated to a process:
  - maximum number of memory references permitted in a single instruction

    Example: PDP-11 `MOV` instruction
    - instruction may occupy $>$ 1 word
    - 2 operands each of which can be an indirect reference
  - if fewer frames are allocated, process should be swapped out, and allocated frames freed

Allocation of frames

Let n = # of processes
M = total # of memory frames
s_i = size of process p_i
a_i = # of frames allocated to p_i

Equal allocation:

$$a_i = M/n$$

Proportional allocation:

$$a_i = M \times s_i/\Sigma s_i$$

Priority-based allocation:

$$a_i = f(P_i,\ M \times s_i/\Sigma s_i)$$

NOTE: Allocation depends on level of multiprogramming

Thrashing

Definition: situation in which a process is spending more time paging than executing

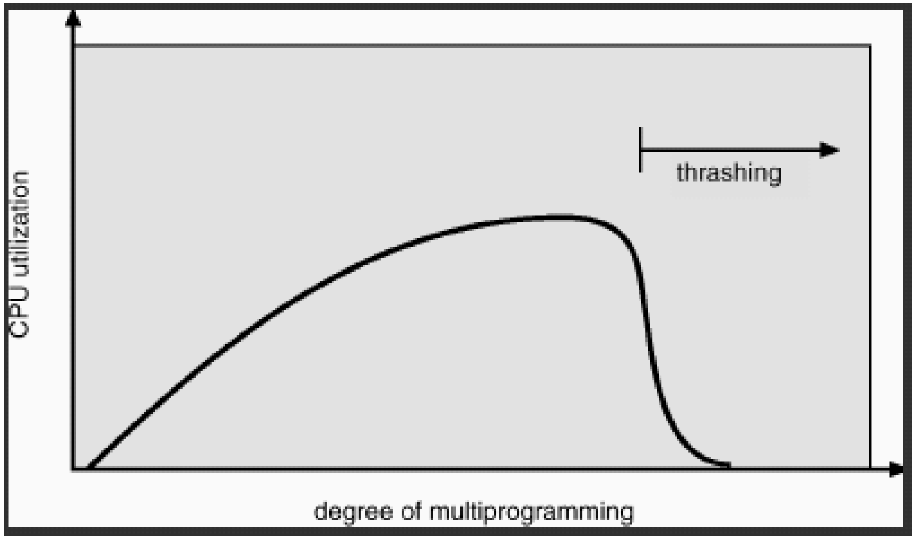
Scenario I:

- Process is not allocated “enough” frames to hold all pages that are in active use
- On a page fault, an active page (p) is replaced
  $\Rightarrow$ process page faults soon to page in p

Thrashing

Scenario II:

- OS monitors CPU utilization to determine degree of multiprogramming
- Global page replacement algorithm is used
- Process enters a phase where it needs a significantly larger # of frames
- Multiple processes start page-faulting
  - ⇒ paging device queue becomes longer, ready queue empties
  - ⇒ CPU utilization decreases
  - ⇒ CPU scheduler increases degree of multiprogramming



Thrashing: remedies

Local/priority page replacement:

+ If one process starts thrashing, it cannot cause other processes to start thrashing
– Thrashing processes use paging device heavily
  $\Rightarrow$ average service time for page fault increases for non-thrashing processes also

Page fault frequency monitoring:

- Upper and lower bounds on “desired” page fault rate are determined
- If PFR $>$ upper limit, process is allocated another frame
  If PFR $<$ lower limit, a frame is removed from the process
- If PFR increases and no free frames are available:
  - a process is selected and suspended
  - freed frames are distributed to processes with high PFRs

Thrashing: remedies

Locality model:

- a set of pages that are actively used together
  e.g. subroutine code, local variables, and some subset of global variables
- process moves from one locality to another (possibly overlapping) locality during execution

Working set model:

- *Working set window* = most recent Δ page references
- *Working set* = set of pages in the working set window
  - approximates the program's current locality
  - Δ too large $\Rightarrow$ working set overlaps several localities
  - Δ too small $\Rightarrow$ working set does not cover entire locality
- Total demand for frames $D = \sum WSS_i$

Thrashing: remedies

Working set model: (CONTD.)

- OS monitors working set of each process and allocates enough frames to accomodate working set
- If extra frames are available, more processes can be loaded into memory
  If D exceeds # of available frames, process(es) must be suspended
- Implementation:
  - Timer interrupt is generated at regular intervals e.g. every 5000 memory references
  - For each page, reference bit is copied into history register and cleared
  - Overhead = Frequency of interrupt, # of history bits

Performance

- Effective access time = $ma + p \times PF\ time$
  where ma - memory access time
  p - probability of a page fault
- Page fault service time:
  - time to service page fault interrupt
  - time for I/O
  - time to restart process

Section 9.3

Example: *PF Time*: 25ms ma: 100ns
EAT $\approx 100 + 25,000,000 \times p$
(for acceptable performance, $<$ 1 memory access in 2,500,000 should fault)

Performance

Swap space:

- Swap space should be allocated in large blocks
  $\Rightarrow$ Disk I/O to swap faster than I/O to file system
- File image can be copied to swap space at process startup
- If swap space is limited: (e.g. BSD UNIX)
  - pages are brought in from file system on demand
  - replaced pages are written to swap space

Page buffering

- Systems may maintain a pool of free frames
- On a page fault:
  - required page is read into a free frame from the pool
  - in parallel, a victim is selected and written to disk
  - victim frame is added to free-frame pool
- Process restarts as soon as possible
- Page information may also be maintained for each free frame
  - if desired page is in free-frame pool, no I/O is necessary
  - used on VAX/VMS systems with FIFO page replacement

- System may maintain a list of modified pages
- When paging device is idle, modified pages are written to disk