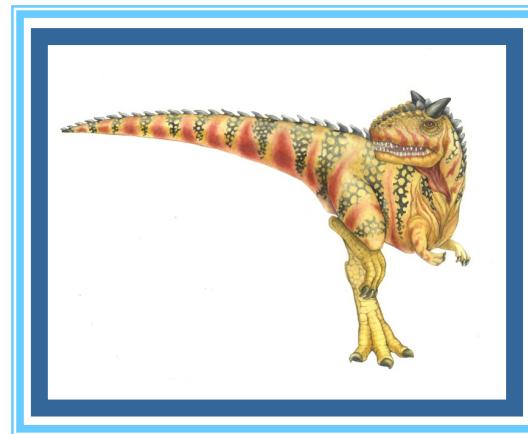
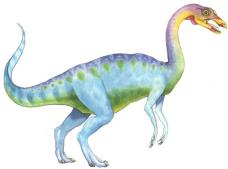


Chapter 10: Virtual Memory





Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames





Background

- Code needs to be in memory to execute, but entire program rarely used
 - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
 - Program no longer constrained by limits of physical memory
 - Each program takes less memory while running -> more programs run at the same time
 - ▶ Increased CPU utilization and throughput with no increase in response time or turnaround time
 - Less I/O needed to load or swap programs into memory -> each user program runs faster





Background (Cont.)

- **Virtual memory** – separation of user logical memory from physical memory
 - Only part of the program needs to be in memory for execution
 - Logical address space can therefore be much larger than physical address space
 - Allows address spaces to be shared by several processes
 - Allows for more efficient process creation
 - More programs running concurrently
 - Less I/O needed to load or swap processes

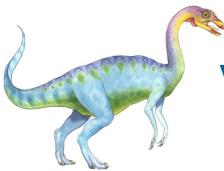




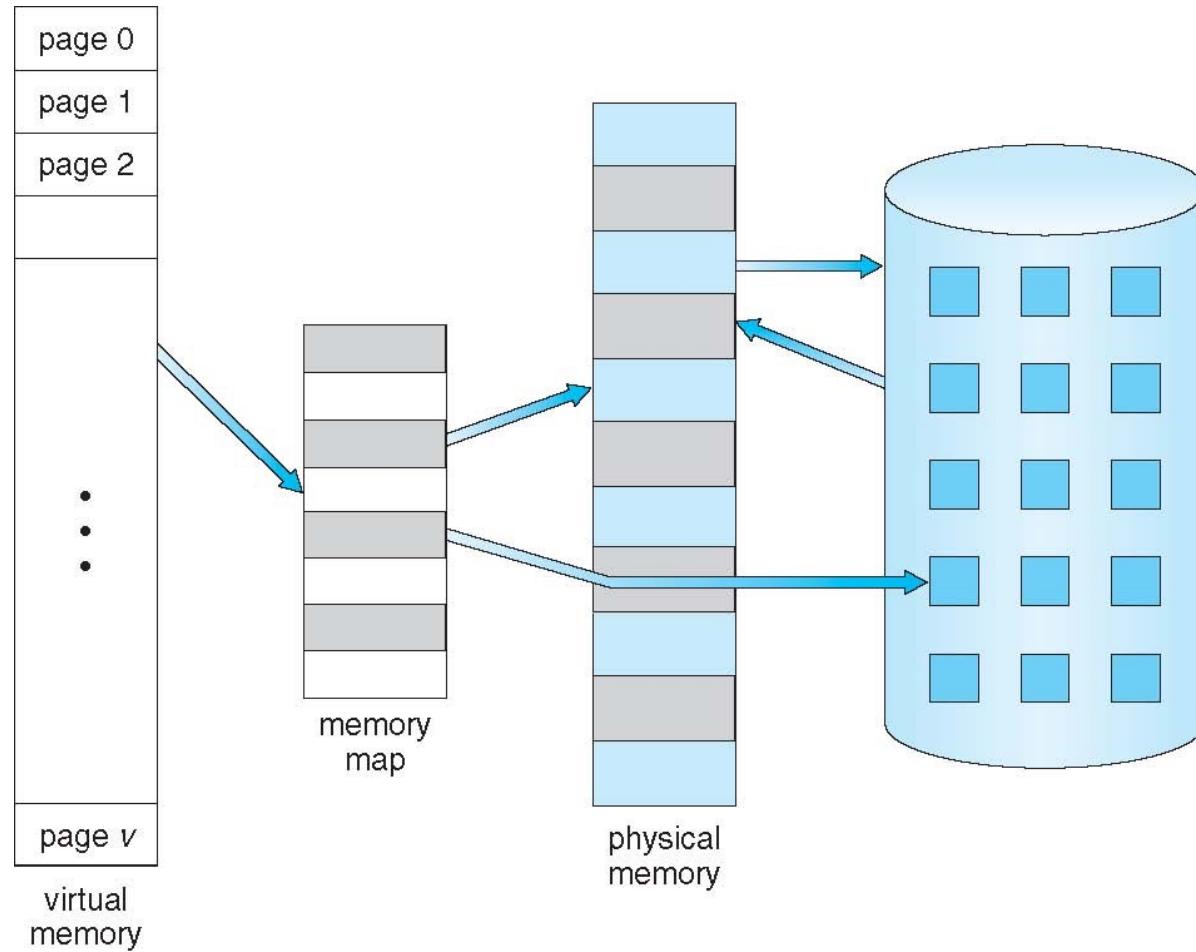
Background (Cont.)

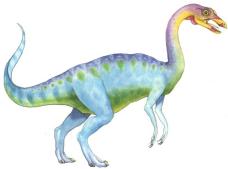
- **Virtual address space** – logical view of how process is stored in memory
 - Usually start at address 0, contiguous addresses until end of space
 - Meanwhile, physical memory organized in page frames
 - Memory Management Unit (MMU) must map logical to physical
- **Virtual memory** can be implemented via:
 - Demand paging
 - Demand segmentation





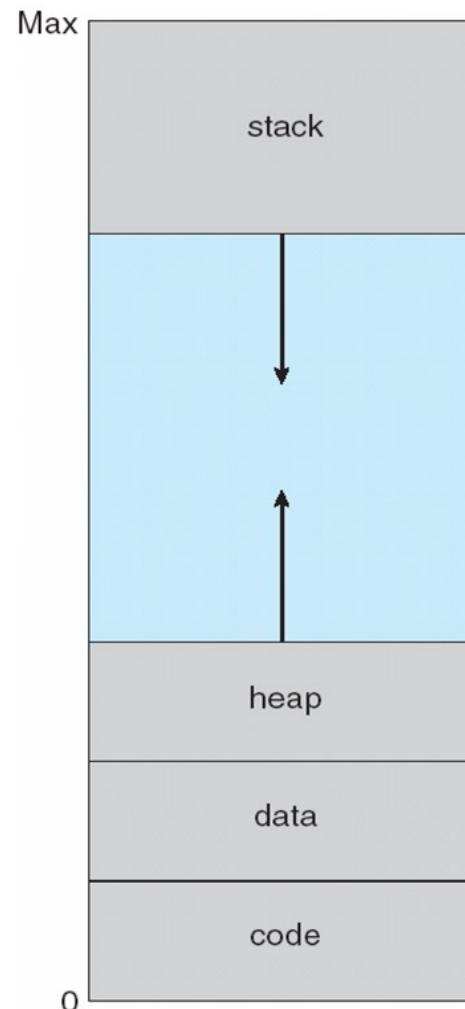
Virtual Memory That is Larger Than Physical Memory





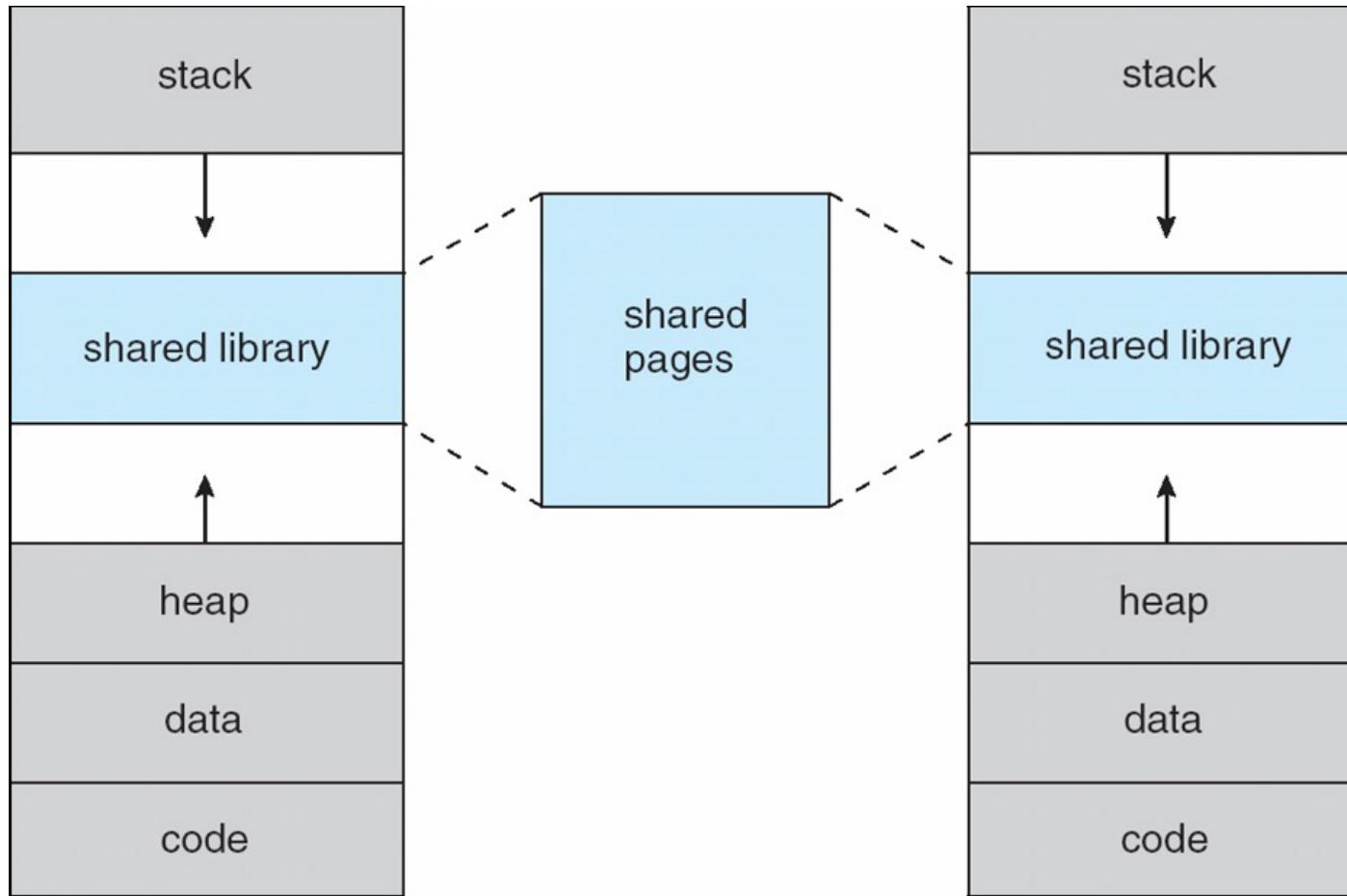
Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
 - Maximizes address space use
 - Unused address space between the two is hole
 - ▶ No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation





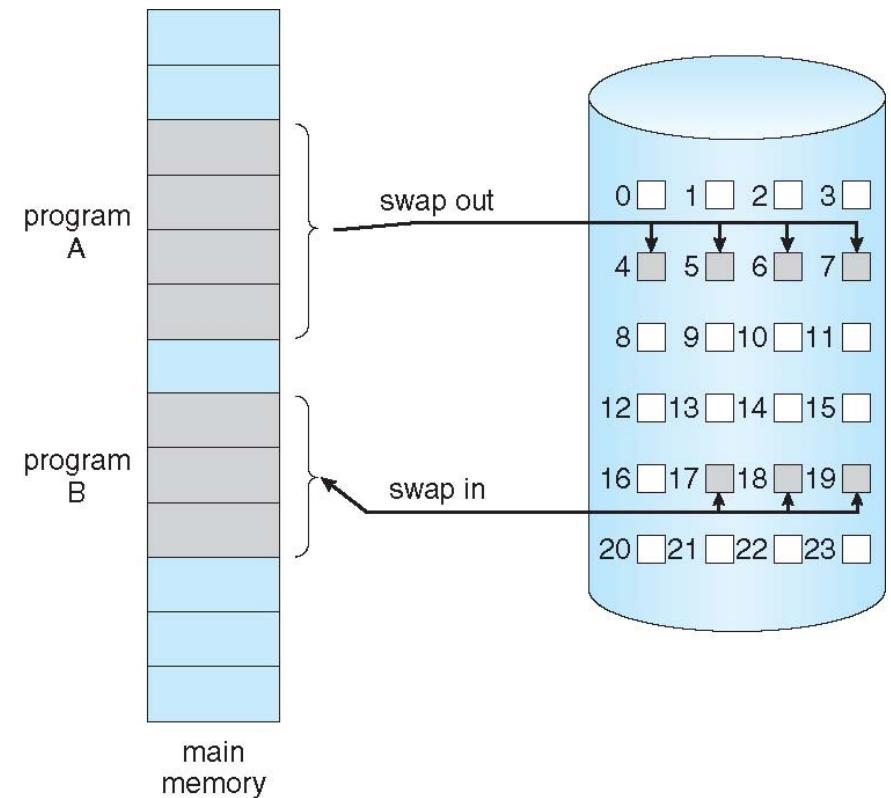
Shared Library Using Virtual Memory





Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
 - Less I/O needed, no unnecessary I/O
 - Less memory needed
 - Faster response
 - More users
- Similar to paging system with swapping (**diagram on right**)
- Page is needed \Rightarrow reference to it
 - invalid reference \Rightarrow abort
 - not-in-memory \Rightarrow bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
 - Swapper that deals with pages is a **pager**

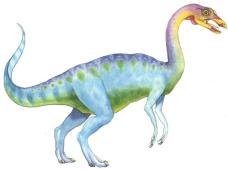




Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
 - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
 - No difference from non demand-paging
- If page needed and not memory resident
 - Need to detect and load the page into memory from storage
 - ▶ Without changing program behavior
 - ▶ Without programmer needing to change code





Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

| Frame # | valid-invalid bit |
|---------|-------------------|
| | v |
| | v |
| | v |
| | i |
| ... | |
| | i |
| | i |

page table

- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

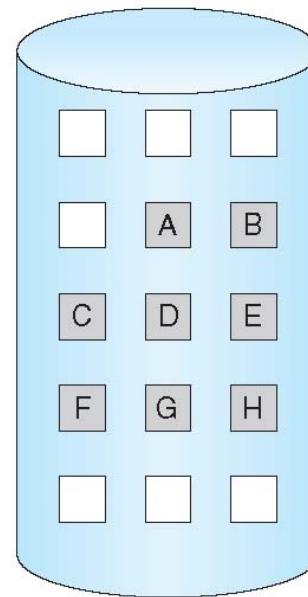
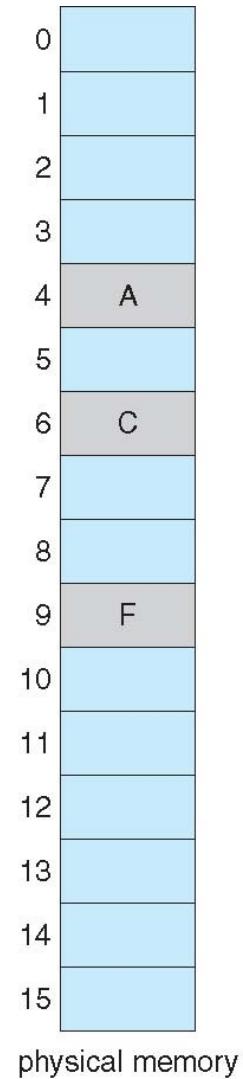
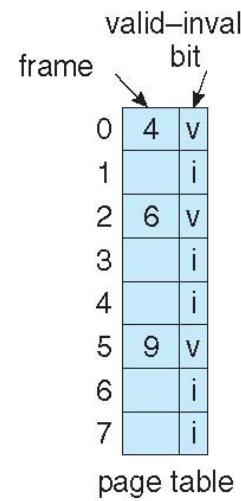


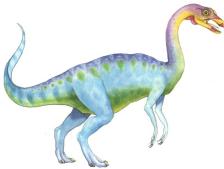


Page Table When Some Pages Are Not in Main Memory

| | |
|---|---|
| 0 | A |
| 1 | B |
| 2 | C |
| 3 | D |
| 4 | E |
| 5 | F |
| 6 | G |
| 7 | H |

logical memory



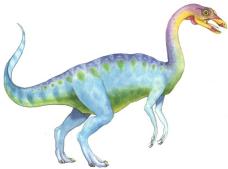


Page Fault

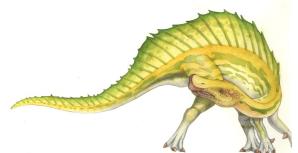
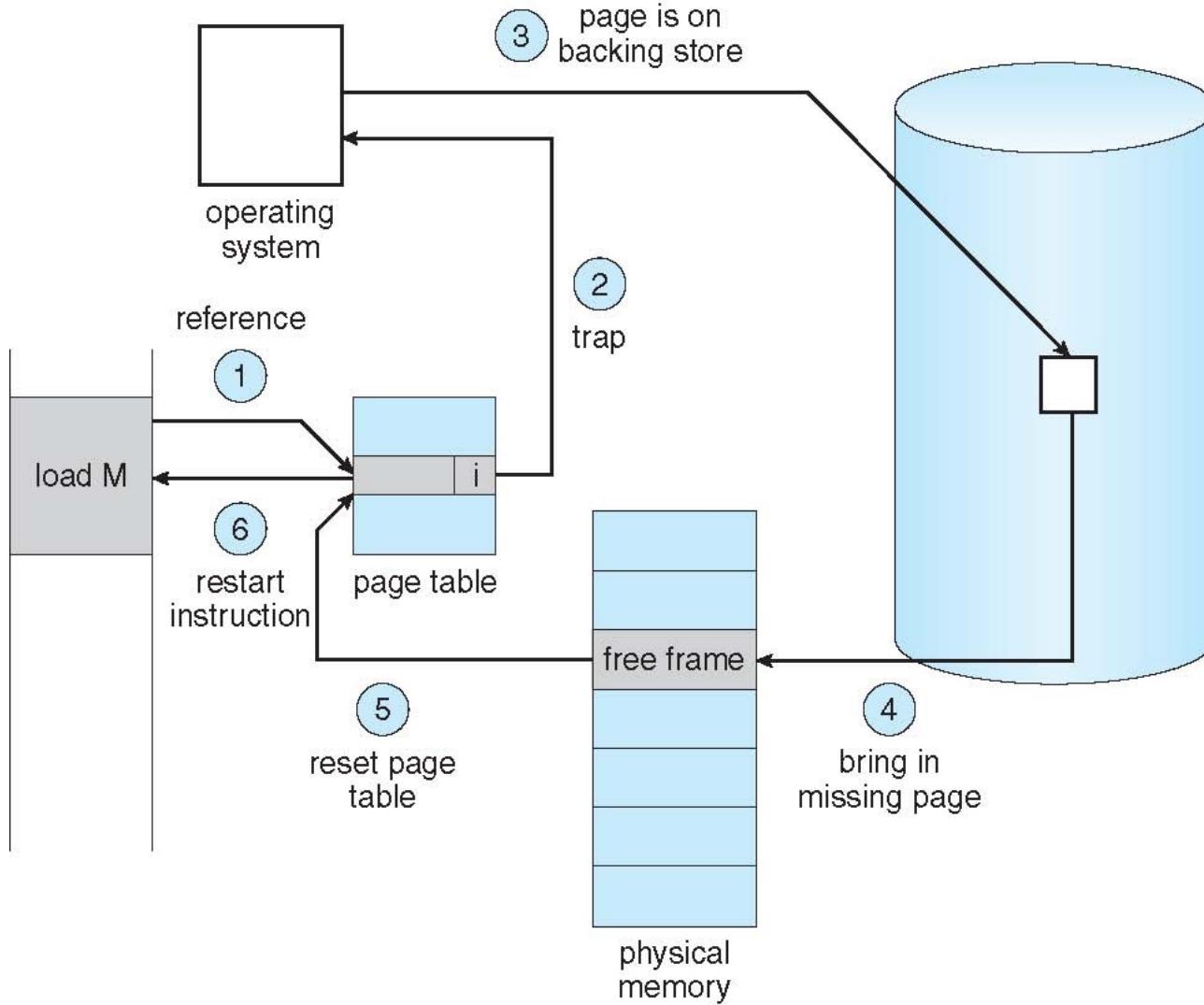
- If the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a **page fault**.
 - The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system.

 1. Operating system looks at another table (usually kept in the Process Control Block) to decide:
 - Just not Invalid reference \Rightarrow abort
 - in memory
 2. Find free frame
 3. Swap page into frame via scheduled disk operation
 4. Reset tables to indicate page now in memory
Set validation bit = **V**
 5. Restart the instruction that caused the page fault





Steps in Handling a Page Fault

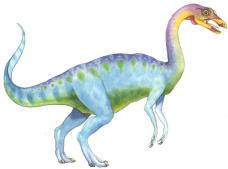




Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
 - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
 - And for every other process pages on first access
 - **Pure demand paging** → never bring a page into memory until it is required
- Actually, a given instruction could access multiple pages -> multiple page faults
 - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
 - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
 - Page table with valid / invalid bit
 - Secondary memory (swap device with **swap space**)
 - Instruction restart





Example

- Consider a three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps to execute this instruction:
 1. Fetch and decode the instruction (ADD).
 2. Fetch A.
 3. Fetch B.
 4. Add A and B.
 5. Store the sum in C. (**suppose page fault happened here**)



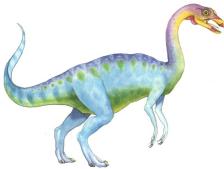


Performance of Demand Paging

■ A page fault causes the following sequence to occur:

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
 1. Wait in a queue for this device until the read request is serviced
 2. Wait for the device seek and/or latency time
 3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction





Performance of Demand Paging (Cont.)

- In any case, There are three major task components of the page-fault service time:
 - Service the interrupt – careful coding means just several hundred instructions needed
 - Read the page – lots of time
 - Restart the process – again just a small amount of time
- Page Fault Rate $0 \leq p \leq 1$
 - if $p = 0$ no page faults
 - if $p = 1$, every reference is a fault
- **Effective Access Time (EAT)**

$$\begin{aligned} EAT &= (1 - p) \times \text{memory access} \\ &\quad + p (\text{page fault overhead} \\ &\quad \quad \quad + \text{swap page out} \\ &\quad \quad \quad + \text{swap page in}) \end{aligned}$$





Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $EAT = (1 - p) \times 200 + p (8 \text{ milliseconds})$
 $= (1 - p \times 200 + p \times 8,000,000)$
 $= 200 + p \times 7,999,800$
- If one access out of 1,000 causes a page fault, then
 $EAT = 8.2 \text{ microseconds.}$
This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
 - $220 > 200 + 7,999,800 \times p$
 $20 > 7,999,800 \times p$
 - $p < .0000025$
 - < one page fault in every 400,000 memory accesses



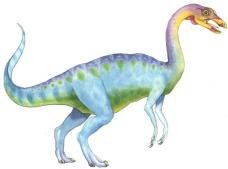


Check your understanding

- 1) Referencing a page that is not yet loaded in main memory is known as _____.
 - demand paging
 - a page fault
 - a trap

- 2) Handling a page fault requires _____.
 - setting the valid/invalid bit to valid
 - requires checking the value of the valid/invalid bit
 - bringing the page from the backing store and storing it in a frame in physical memory



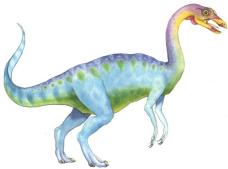


Check your understanding

- 3) On a system with demand-paging, a process will experience a high page fault rate when the process begins execution.
 - True
 - False

- 4) If a memory access requires 100 ns and servicing a page fault requires 6 ms, what is the effective access time with page fault rate of $p = 0.25$?
 - 4,500,025 ms
 - 1,500,075 ms
 - 1,500,025 ms

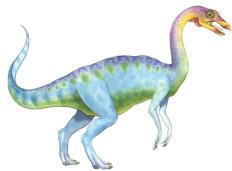




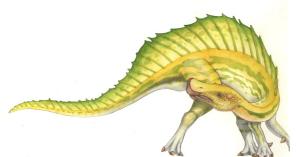
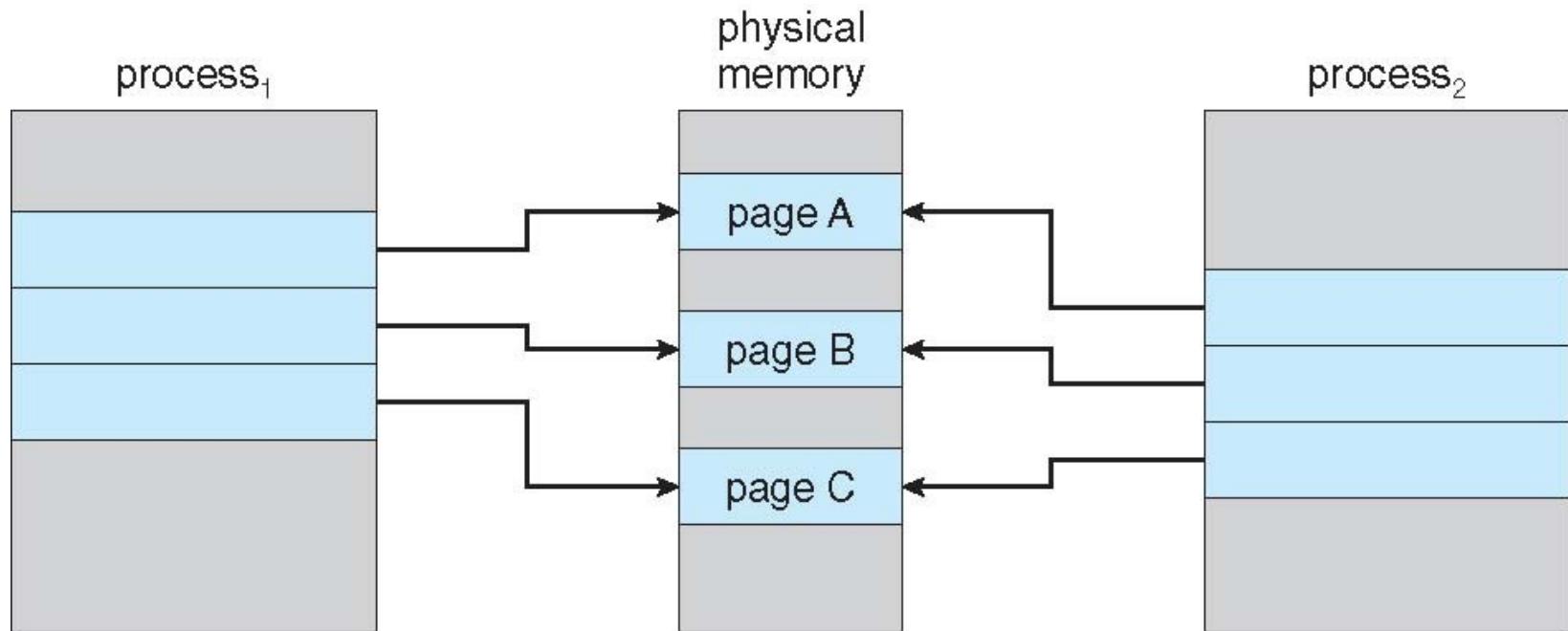
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially **share** the same pages in memory
 - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
 - Pool should always have free frames for fast demand page execution
 - ▶ Don't want to have to free a frame as well as other processing on page fault
 - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent



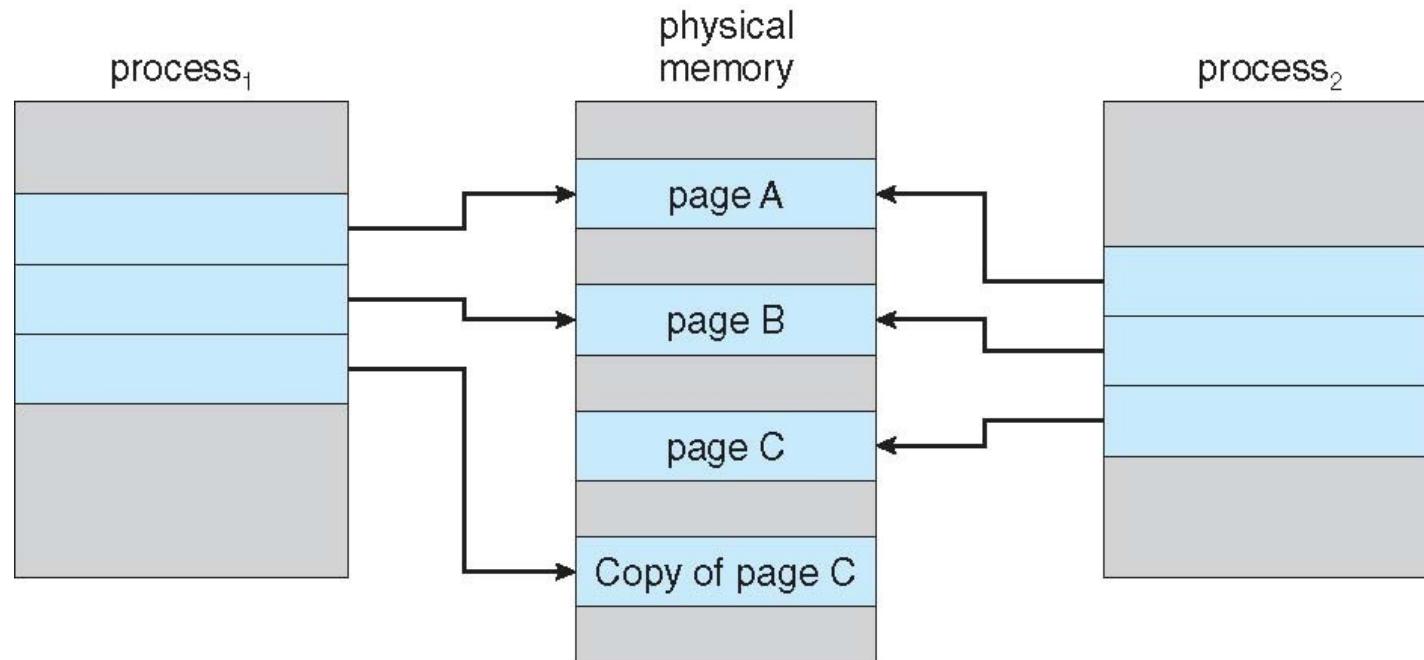


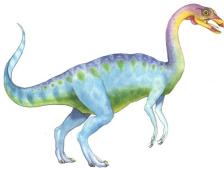
Before Process 1 Modifies Page C





After Process 1 Modifies Page C





What Happens if There is no Free Frame?

- Used up by process pages
- Also, in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
 - Algorithm – terminate? swap out? replace the page?
 - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times





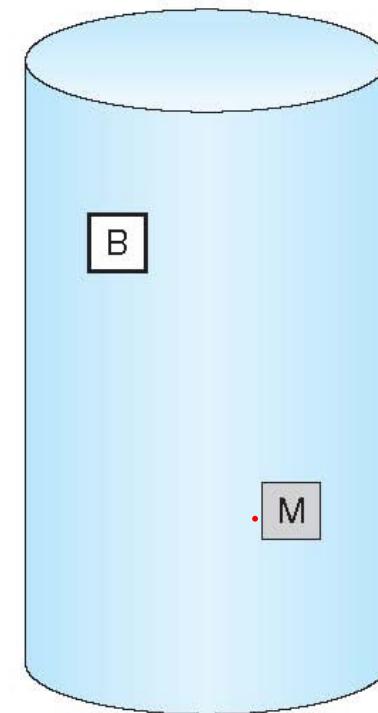
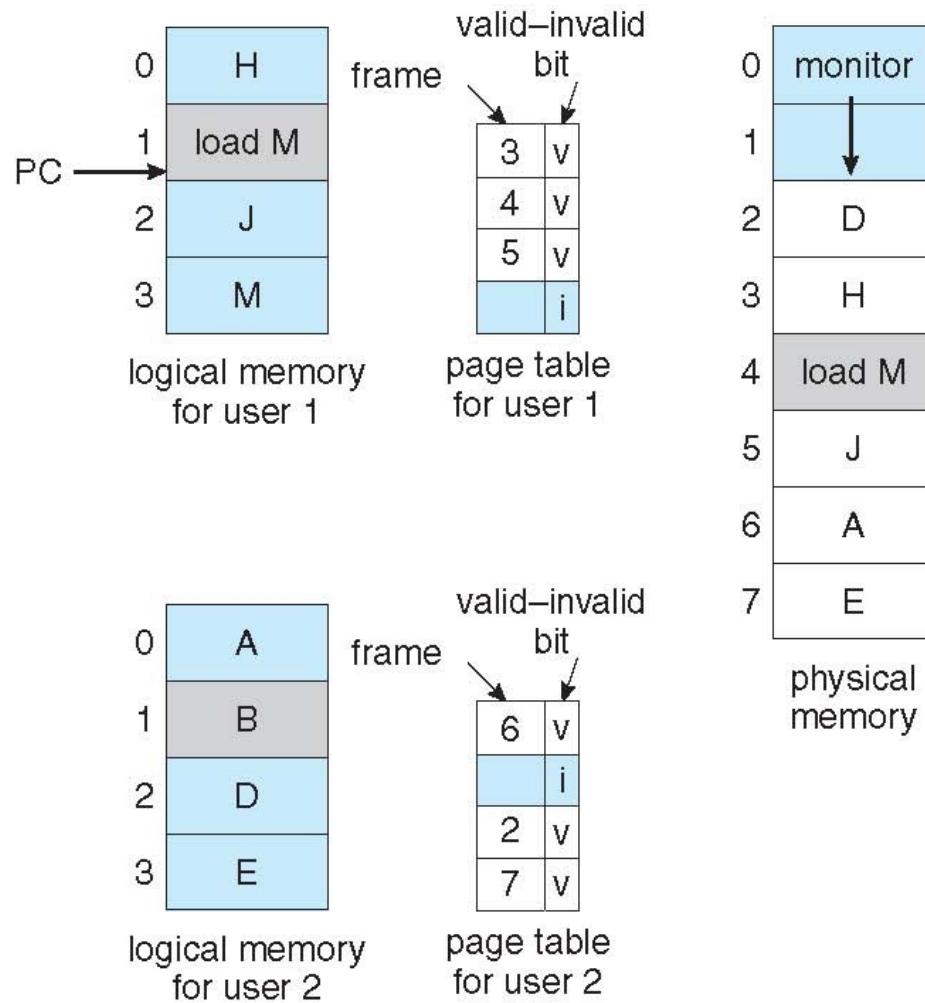
Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory





Need For Page Replacement





Basic Page Replacement

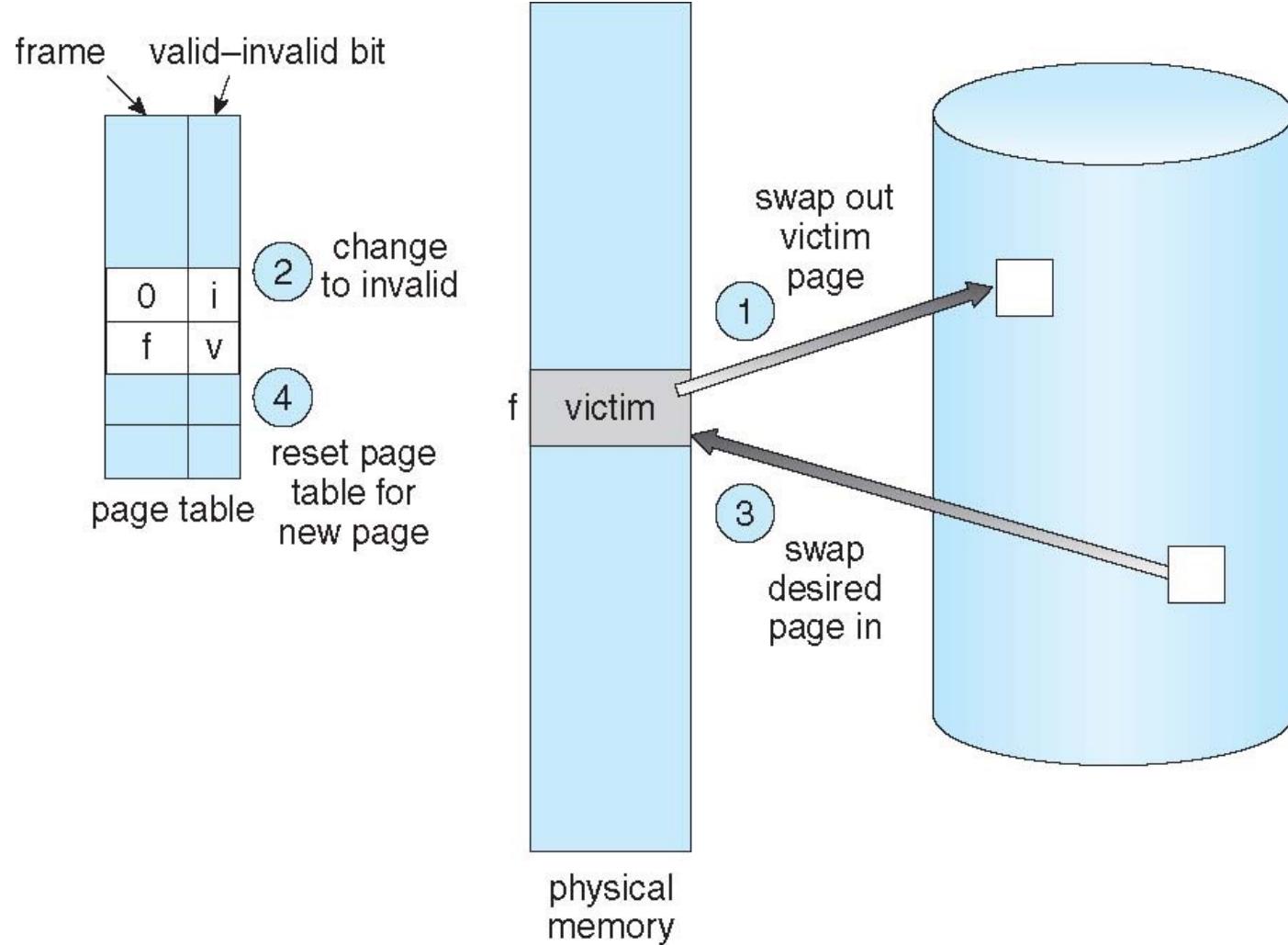
1. Find the location of the desired page on disk
2. Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a **victim frame**
 - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT





Page Replacement





Page and Frame Replacement Algorithms

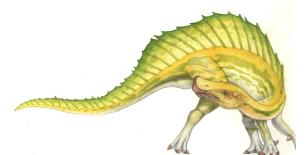
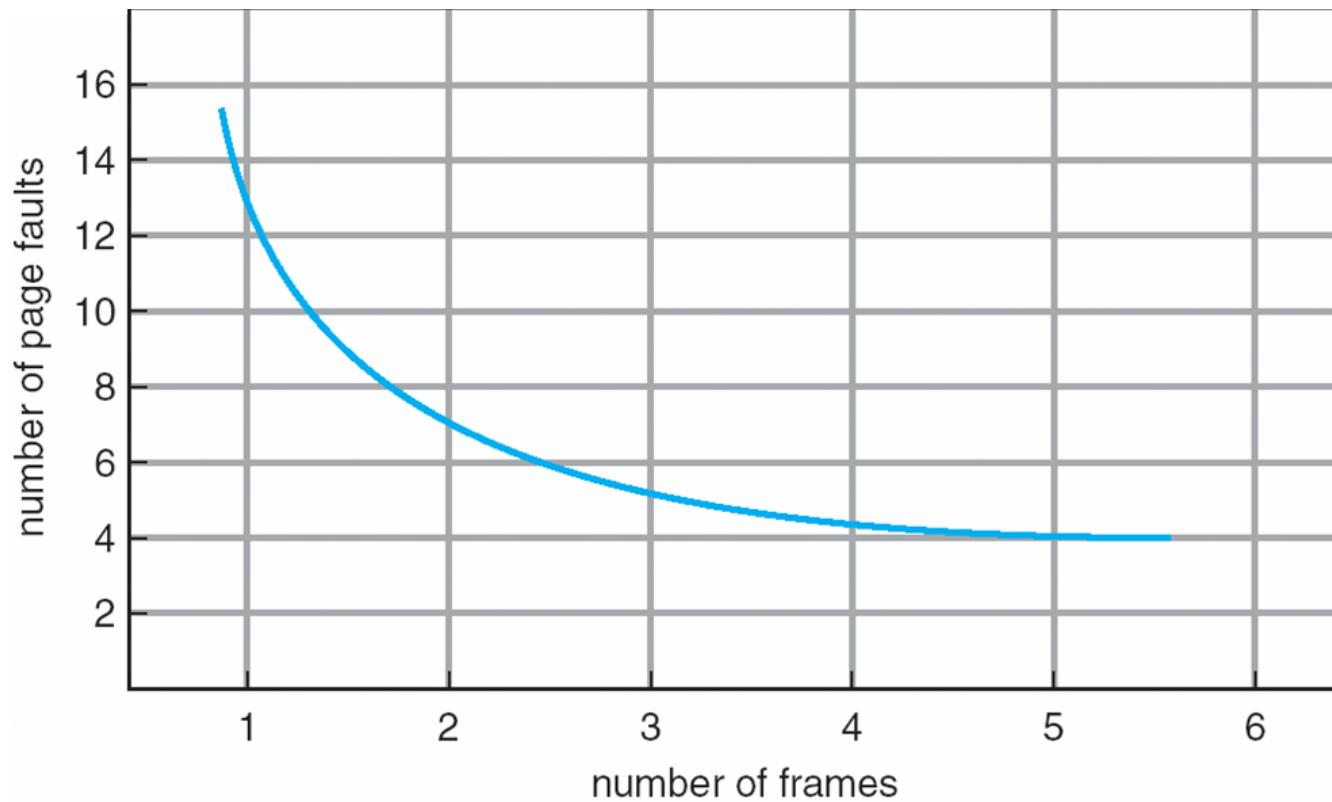
- **Frame-allocation algorithm** determines
 - How many frames to give each process
 - Which frames to replace
- **Page-replacement algorithm**
 - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
 - String is just page numbers, not full addresses
 - Repeated access to the same page does not cause a page fault
 - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

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





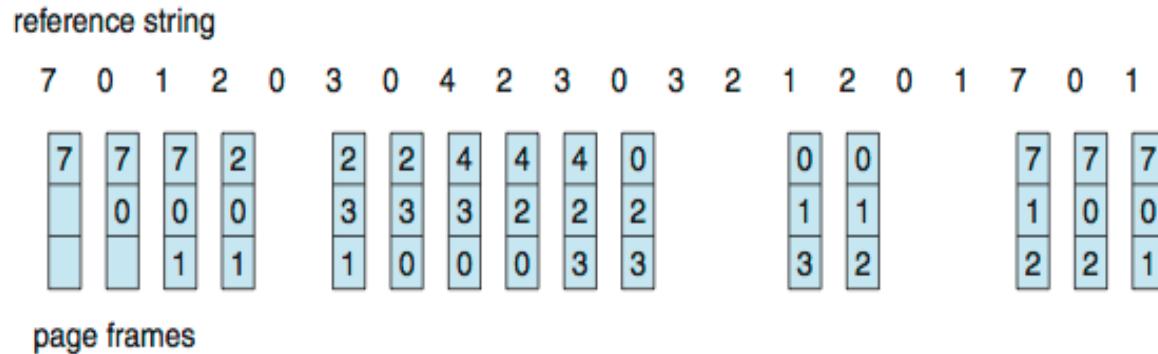
Graph of Page Faults Versus The Number of Frames





First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



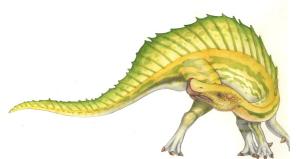
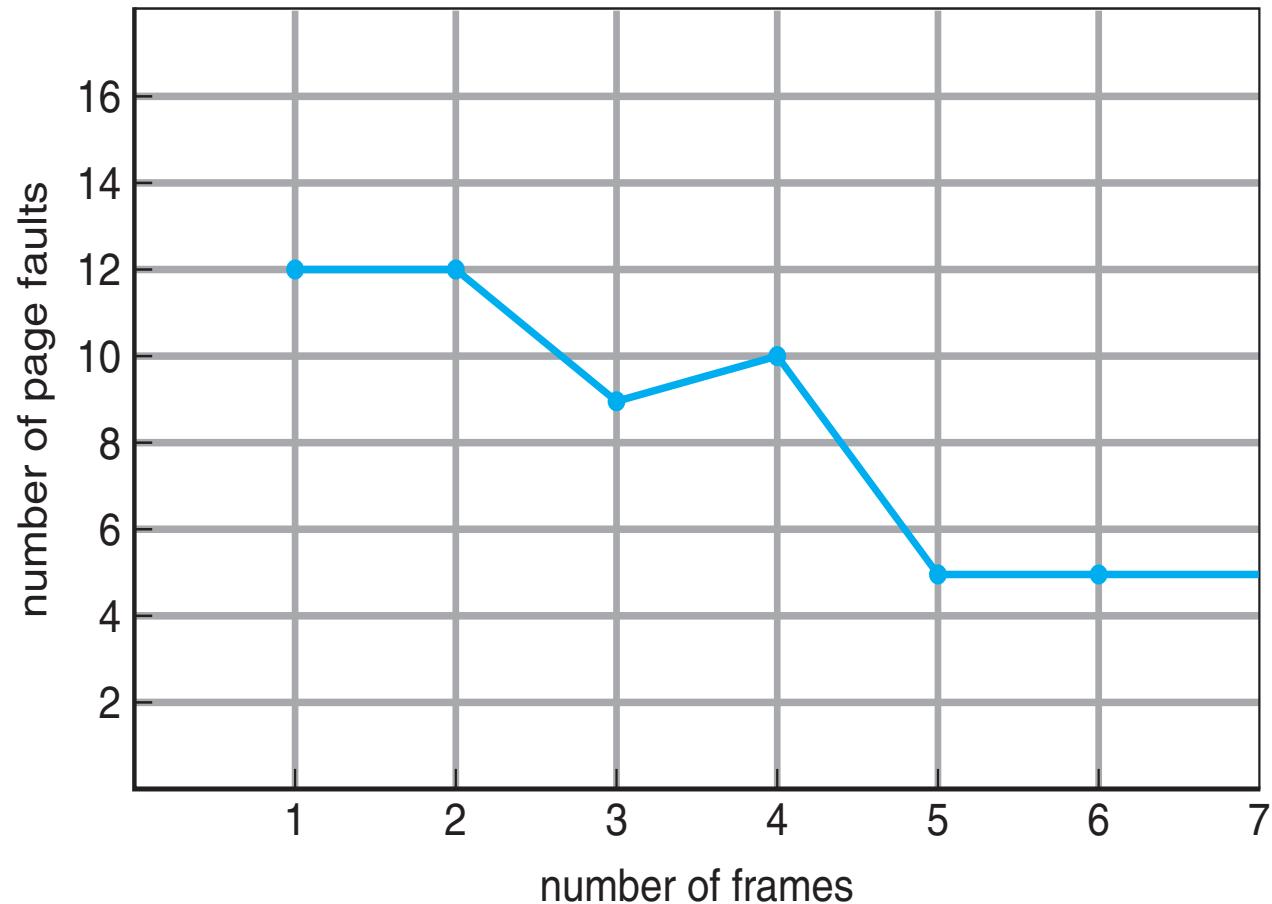
15 page faults

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
 - Adding more frames can cause more page faults!
 - ▶ **Belady's Anomaly**
- How to track ages of pages?
 - Just use a FIFO queue





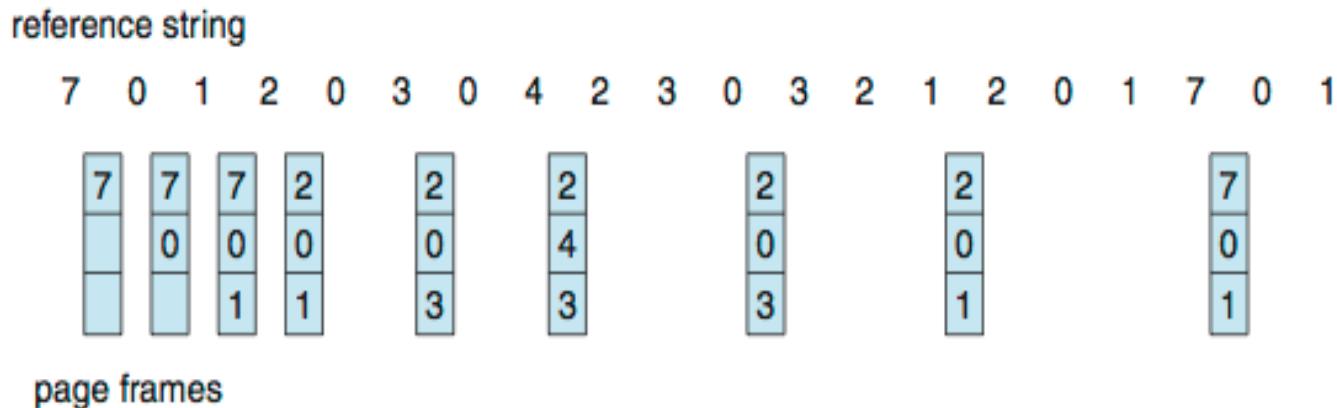
FIFO Illustrating Belady's Anomaly





Optimal Algorithm

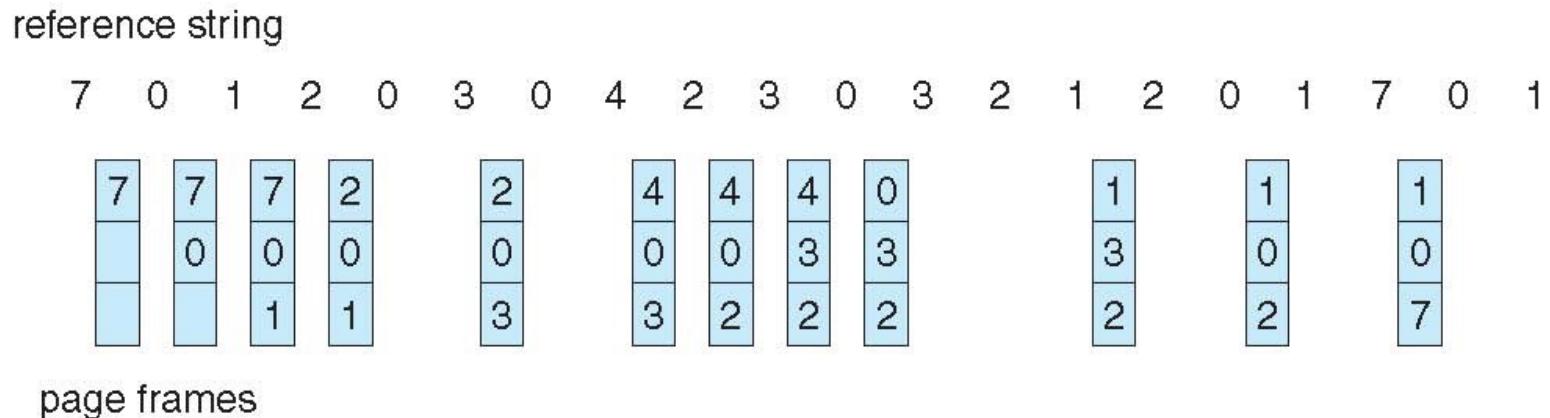
- Replace page that will not be used for longest period of time
 - 9 is optimal for the example
- How do you know this?
 - Can't read the future
- Used for measuring how well your algorithm performs





Least Recently Used (LRU) Algorithm

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



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

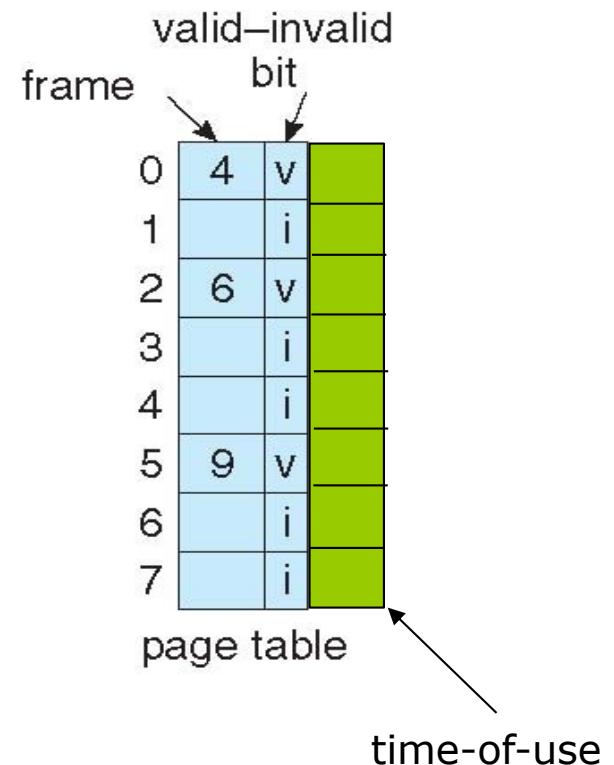




LRU Algorithm (Cont.)

■ Counter implementation

- associate each page-table entry with a time-of-use field and add to the CPU a logical clock or counter.
- the clock is incremented for every memory reference.
- whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page.
- In this way, we always have the "time" of the last reference to each page.
- replace the page with the smallest time value.
 - ▶ This scheme requires a search of the page table to find the LRU page and a write to memory



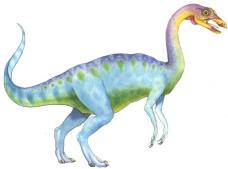


LRU Algorithm (Cont.)

■ Stack implementation

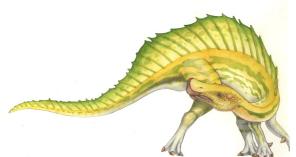
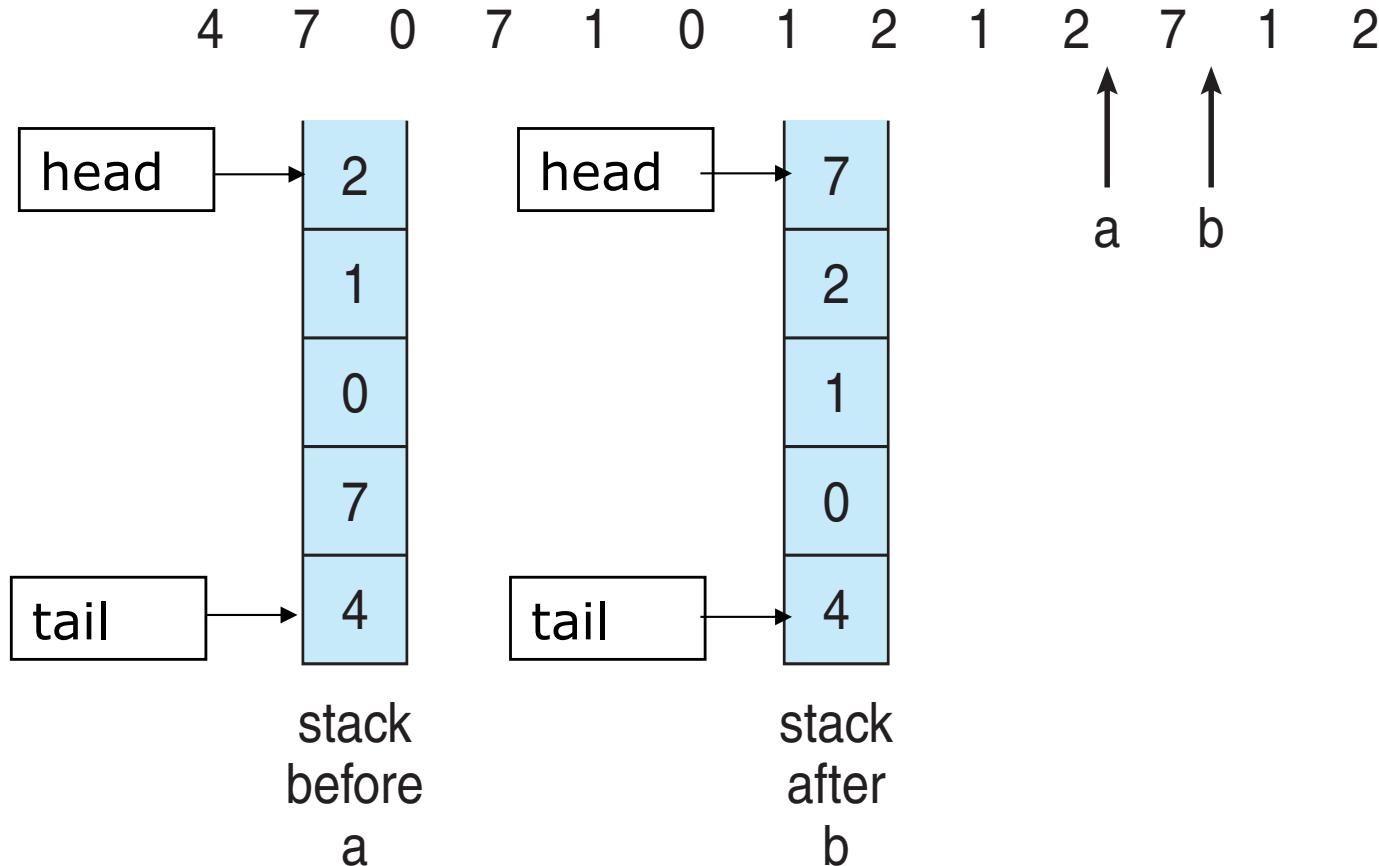
- Keep a stack of page numbers in a double link form (**doubly linked list with a head pointer and a tail pointer**):
 - ▶ top of the stack for the most recently used page
 - ▶ bottom of the stack for the least recently used page
- Page referenced (**see next page's example**):
 - ▶ move it to the top
 - ▶ requires 6 pointers to be changed
- But each update more expensive
- No search (on the page table) for replacement





Use Of A Stack to Record Most Recent Page References

reference string





LRU Algorithm (Cont.)

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly
- **Note:** neither implementation of LRU would be conceivable without hardware assistance beyond the standard TLB registers.
 - The updating of the clock fields or stack must be done for **every** memory reference.
 - If we were to use an interrupt for every reference to allow software to update such data structures, it would slow every memory reference by a factor of at least ten, hence slowing every process by a factor of ten.





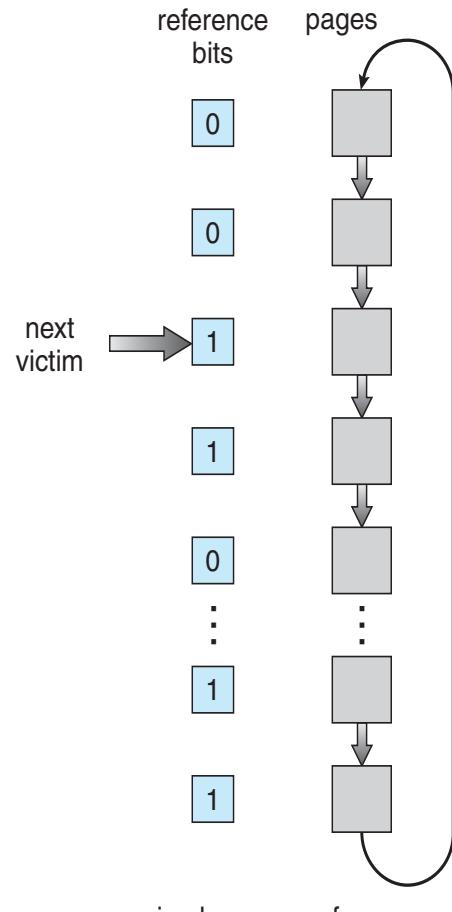
LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
 - With each entry in the page table is associated with a bit, initially = 0
 - When page is referenced, bit set to 1 by the hardware
 - Replace any with reference bit = 0 (if one exists)
 - ▶ We do not know the order of use, however
- **Second-chance algorithm**
 - Generally FIFO, plus hardware-provided reference bit
 - **Clock** replacement
 - If page to be replaced has
 - ▶ Reference bit = 0 -> replace it
 - ▶ reference bit = 1 then:
 - set reference bit 0, leave page in memory
 - replace next page, subject to same rules

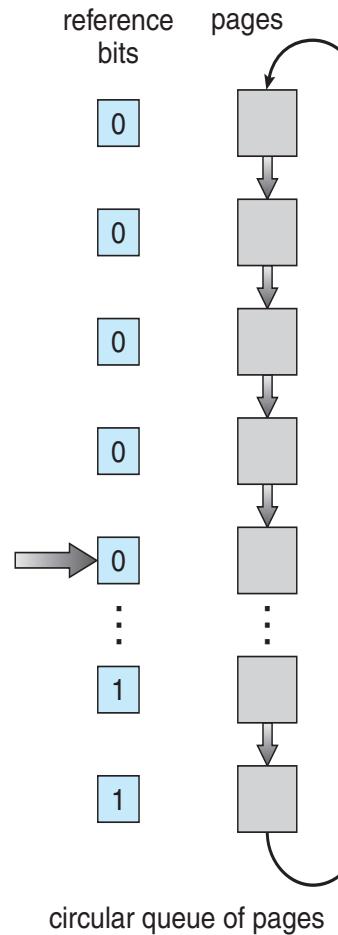




Second-Chance (clock) Page-Replacement Algorithm



(a)



(b)

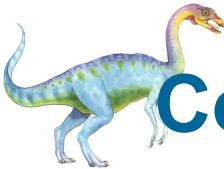




Enhanced Second-Chance Algorithm

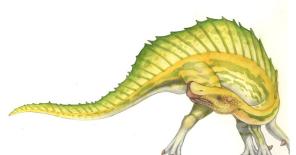
- Improve algorithm by using **reference bit and modify bit** (if available) in concert
- Take ordered pair (reference, modify)
 1. (0, 0) neither recently used nor modified – best page to replace
 2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
 3. (1, 0) recently used but clean – probably will be used again soon
 4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
 - Might need to search circular queue several times

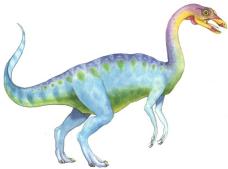




Counting Based Page Replacement Algorithms

- Keep a counter of the number of references that have been made to each page
 - Not common
- **Least Frequently Used (LFU) Algorithm:** replaces page with smallest count
- **Most Frequently Used (MFU) Algorithm:** based on the argument that the page with the smallest count was probably just brought in and has yet to be used





Page-Buffering Algorithms

- Keep a pool of free frames, always
 - Then frame available when needed, not found at fault time
 - Read page into free frame and select victim to evict and add to free pool
 - When convenient, evict victim
- Possibly, keep list of modified pages
 - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
 - If referenced again before reused, no need to load contents again from disk
 - Generally useful to reduce penalty if wrong victim frame selected





Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
 - OS keeps copy of page in memory as I/O buffer
 - Application keeps page in memory for its own work
- Operating system can give direct access to the disk, getting out of the way of the applications
 - **Raw disk** mode
- Bypasses buffering, locking, etc





Allocation of Frames

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
 - instruction is 6 bytes, might span 2 pages
 - 2 pages to handle *from*
 - 2 pages to handle *to*
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
 - fixed allocation
 - priority allocation
- Many variations

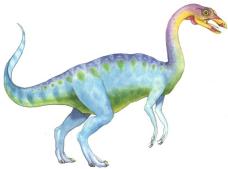




Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
 - Keep some as free frame buffer pool
 - Proportional allocation – Allocate according to the size of process
 - Dynamic as degree of multiprogramming, process sizes change
 - s_i = size of process p_i
 - $S = \sum s_i$
 - m = total number of frames
 - a_i = allocation for $p_i = \frac{s_i}{S} \times m$
- $m = 64$
 $s_1 = 10$
 $s_2 = 127$
 $a_1 = \frac{10}{137} \times 62 \approx 4$
 $a_2 = \frac{127}{137} \times 62 \approx 57$

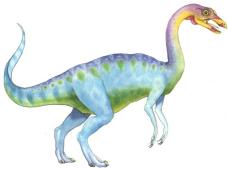




Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
 - select for replacement one of its frames
 - select for replacement a frame from a process with lower priority number



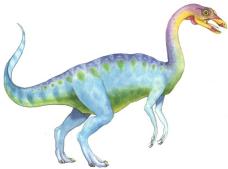


Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
 - But then process execution time can vary greatly
 - But greater throughput so more common

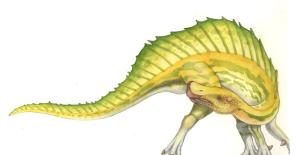
- **Local replacement** – each process selects from only its own set of allocated frames
 - More consistent per-process performance
 - But possibly underutilized memory





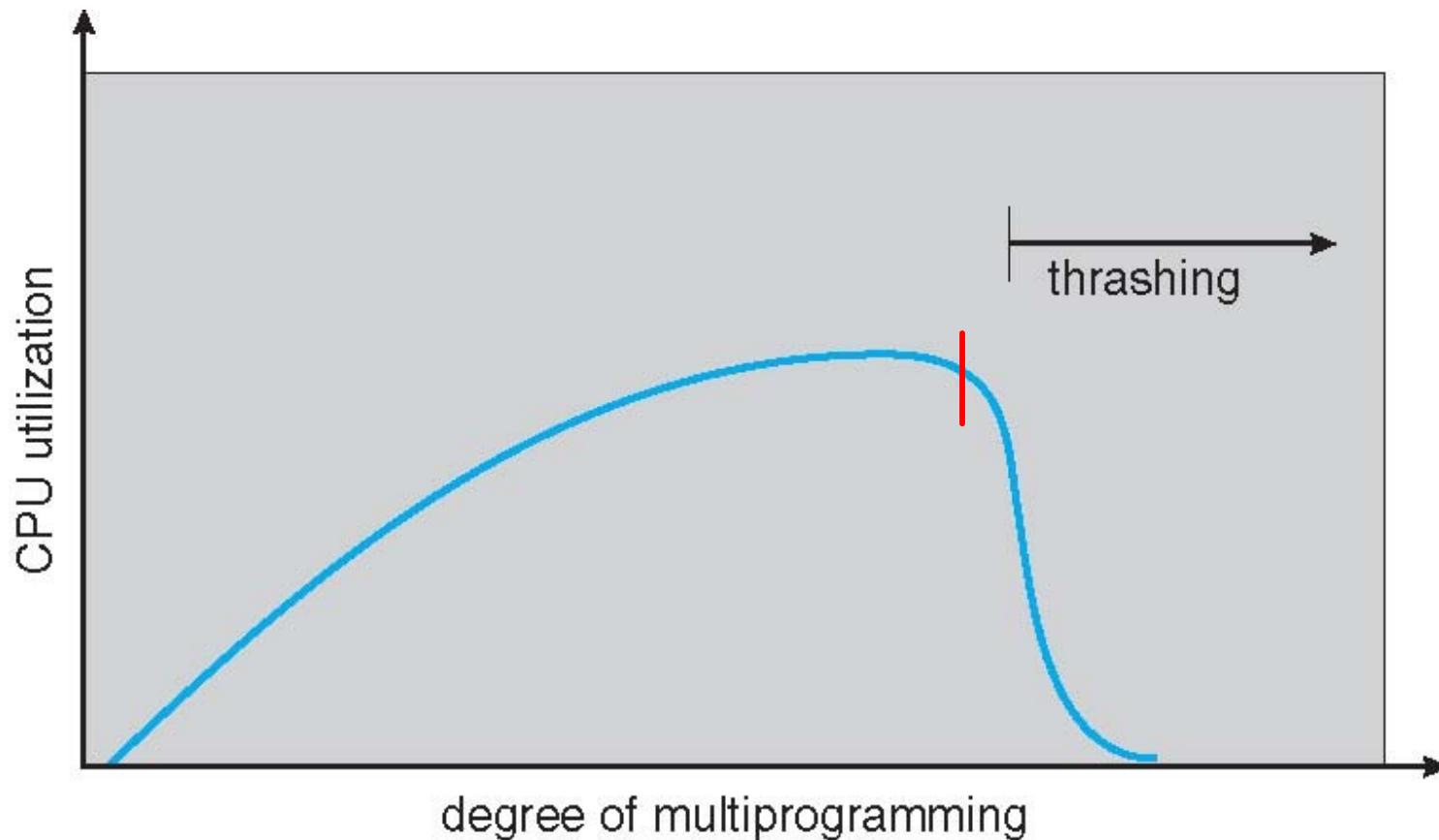
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
 - Page fault to get page
 - Replace existing frame
 - But quickly need replaced frame back
 - This leads to:
 - ▶ Low CPU utilization
 - ▶ Operating system thinking 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





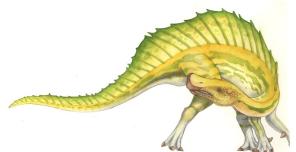
Thrashing (Cont.)



增加程式數量 (multiprogramming) 通常會提高 CPU 利用率

但超過某個臨界點後，因為記憶體不足 → page fault 暴增

CPU 利用率急遽下降





Demand Paging and Thrashing

■ Why does demand paging work?

- **Locality Model:** Programs tend to access a relatively small set of memory locations repeatedly over short periods. These clusters of memory references are known as **localities**.
- Process migrates from one locality to another
- Localities may overlap

■ Why does thrashing occur?

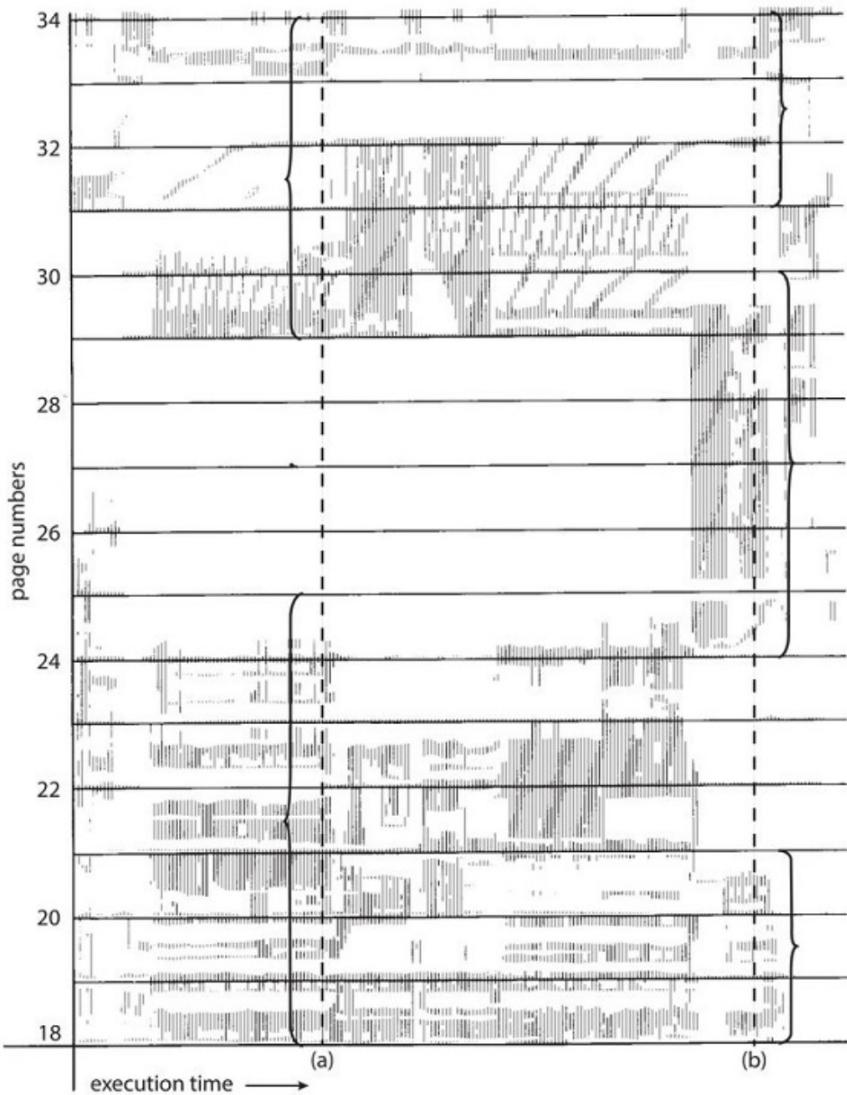
Σ size of locality > total memory size

- Limit effects by using local or priority page replacement





Locality In A Memory-Reference Pattern



A locality is a set of pages that are actively used together.

At time (a), the locality is the set of pages {18, 19, 20, 21, 22, 23, 24, 29, 30, 33}

At time (b), the locality changes to {18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33}



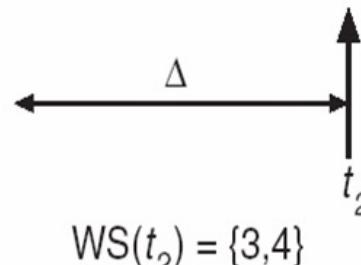
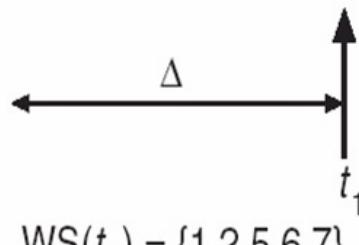


Working-Set Model

- parameter $\Delta \equiv$ working-set window \equiv a fixed number of page references
Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
total number of pages referenced in the most recent Δ (varies in time)
 - 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
- $D = \sum WSS_i \equiv$ total demand frames
 - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

page reference table

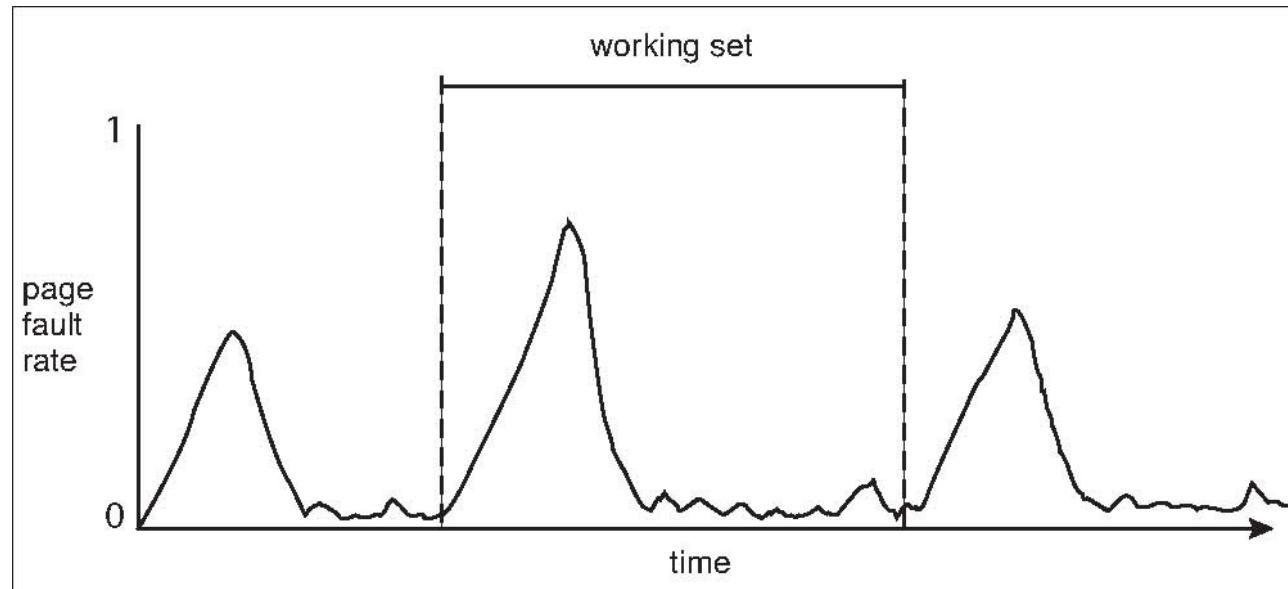
... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...





Working Sets and Page Fault Rates

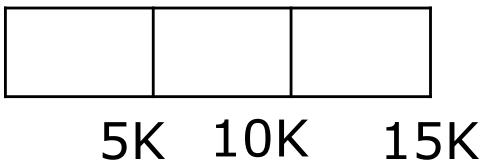
- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time





Keeping Track of the Working Set

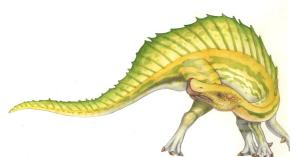
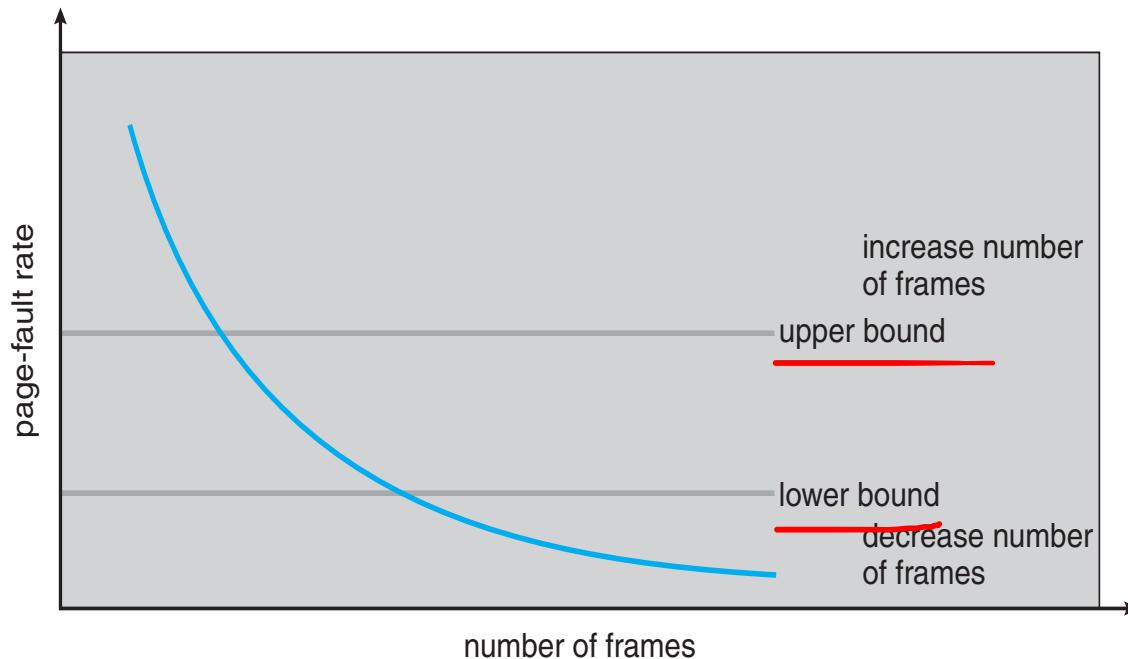
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page
 - Whenever a timer interrupts copy and sets the values of all reference bits to 0
 - If one of the bits in memory = 1 \Rightarrow page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

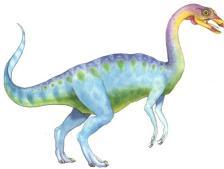




Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame





Other Considerations -- Prepaging

■ Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume s pages are prepaged and a of the pages is used
 - ▶ Is cost of $s * a$ save pages faults > or < than the cost of prepaging
 - ▶ $s * (1 - a)$ unnecessary pages?
 - ▶ a near zero \Rightarrow prepaging loses





Other Issues – Page Size

- Sometimes OS designers have a choice
 - Especially if running on custom-built CPU
- Page size selection must take into consideration:
 - Fragmentation
 - Page table size
 - **Resolution** (granularity of memory allocation)
 - I/O overhead
 - Number of page faults
 - Locality
 - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

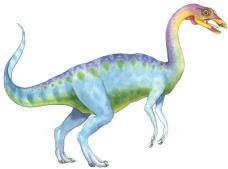




Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- Ideally, the working set of each process is stored in the 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;`
- For pages of 128 words, each row takes one page
- If the operating system does not have enough frames (specifically, fewer than 128 frames), it won't be able to hold all pages at once.
- Program 1

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

$128 \times 128 = 16,384$ page faults

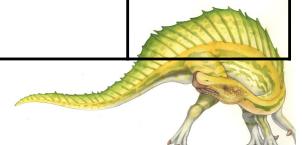
| | | |
|---|--|--|
| 1 | | |
| 2 | | |
| 3 | | |

- Program 2

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

128 page faults

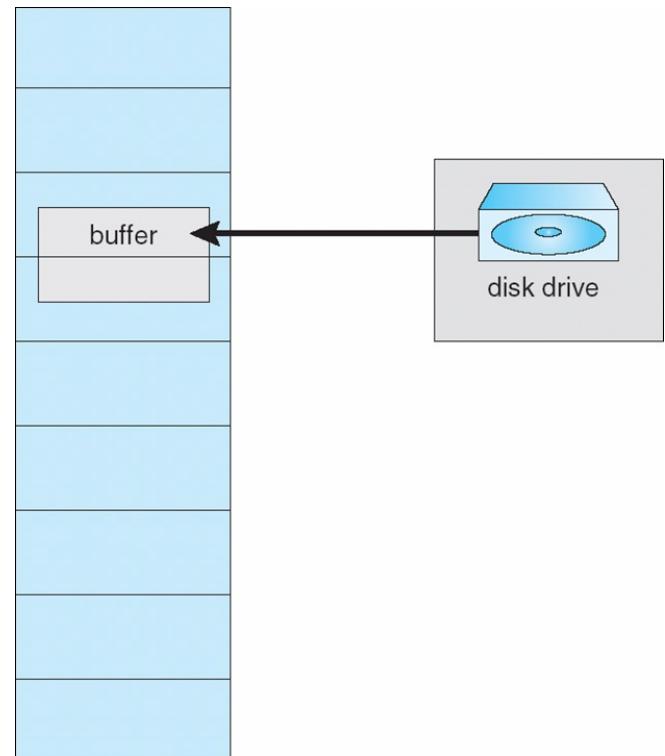
| | | |
|---|---|---|
| 1 | 2 | 3 |
| | | |
| | | |





Other Issues – I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory



End of Chapter 10

