Virtual Memory

Politecnico di Torino

Department of Control and Computer Engineering

System Programming - Sarah Azimi

CAD & Reliability Group
DAUIN– Politecnico di Torino

Outline

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

Objectives

- Memory Management techniques has the goal to keep many process in memory simultaneously.
  - Allowing multiprogramming
- However, they tend to require that an entire process is in memory before execution.
- Virtual Memory allows the execution of processes that are not completely in memory.
- Major advantages:
  - The program can be larger than physical memory.
  - Allowing processes to share files and libraries, and to implement shared memory.

Background

- The instructions being executed must be in physical memory.
  - It limits the size of the program to the size of physical memory.
- In many cases, the entire program is not needed.
  - Programs often have code to handle unusual error conditions which happens barely. Therefore, this code is never executed.
  - Array, lists and tables often are allocated more memory than they actually required.
  - Certain options and features of a program may not be needed at the same time.

Background

- The ability to execute a program that is only partially in memory would confer more benefits:
  - A program would no longer be constrained by the amount of physical memory that is available.
  - Since each program could take less physical memory, more programs could run at the same time, with corresponding increase in CPU utilization with no increase in response.
  - Less I/O would be needed to load or swap portions of programs into memory, so each program can run faster.
- Running program that is not entirely in memory would benefit both system and its user.

Virtual Memory

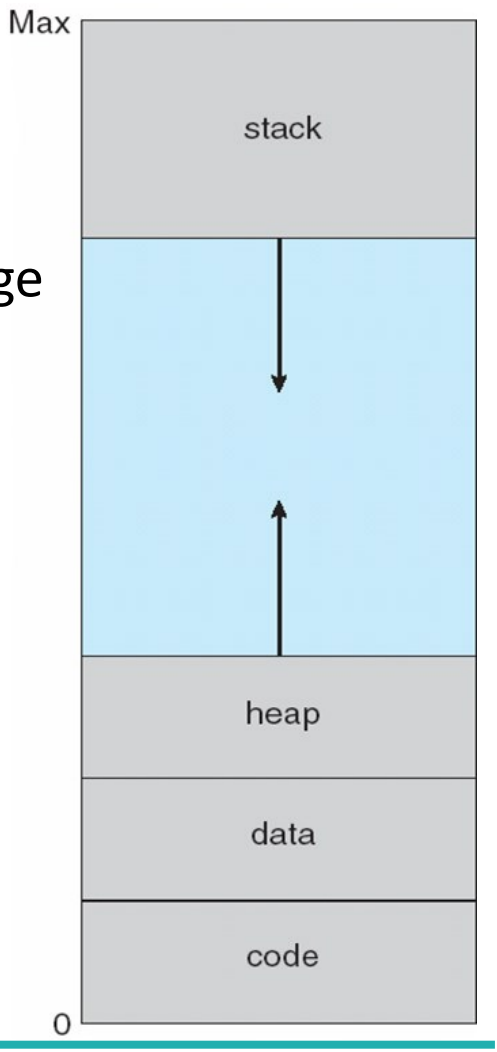
- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for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

Virtual Address Space

- **Virtual address space** – refers to the 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 and the physical pages assign to the process may not be contiguous
  - MMU must map logical to physical frames in memory
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

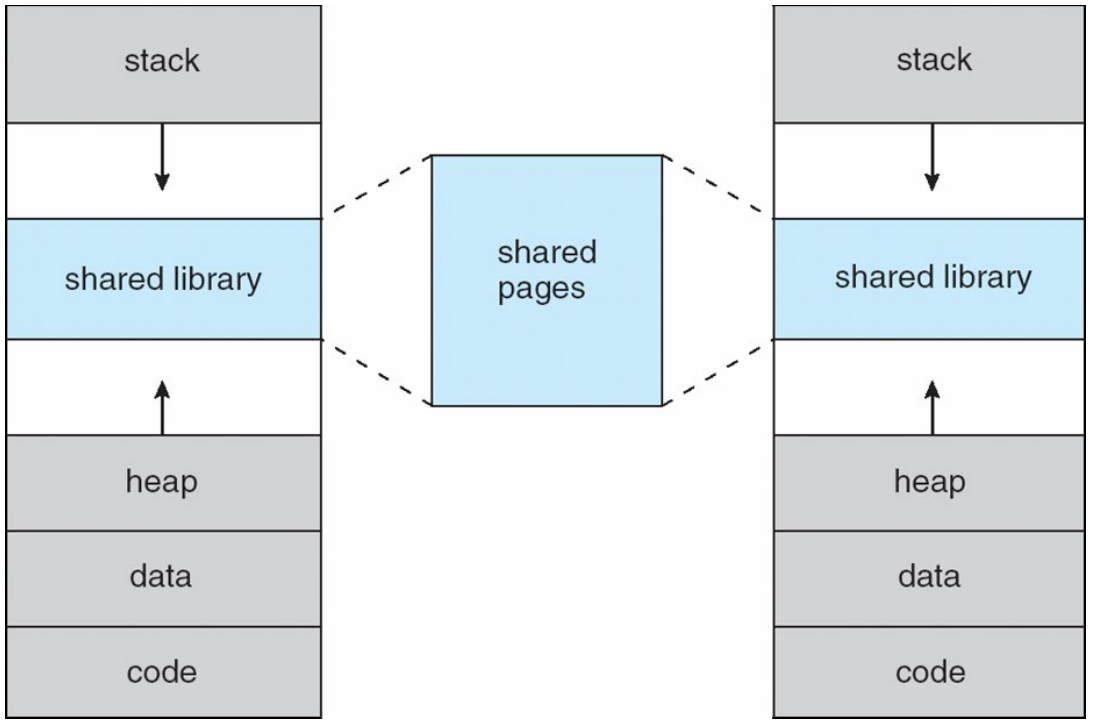
Virtual Address Space

- 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



Virtual Address Space

- Virtual Memory allows files and memory to be shared by two or more processes through page sharing:
  - System libraries such as the standard C library can be shared by several processes through mapping of the shared object into a virtual address space.



Virtual Address Space

- Virtual Memory allows files and memory to be shared by two or more processes through page sharing:
  - System libraries such as the standard C library can be shared by several processes through mapping of the shared object into a virtual address space.
  - Libraries are mapped read-only into the space of each process that is linked with that.
  - Processes can share memory. Virtual memory allows one process to create a region of memory that it can share with another process.
  - Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared.
  - System libraries shared via mapping into virtual address space

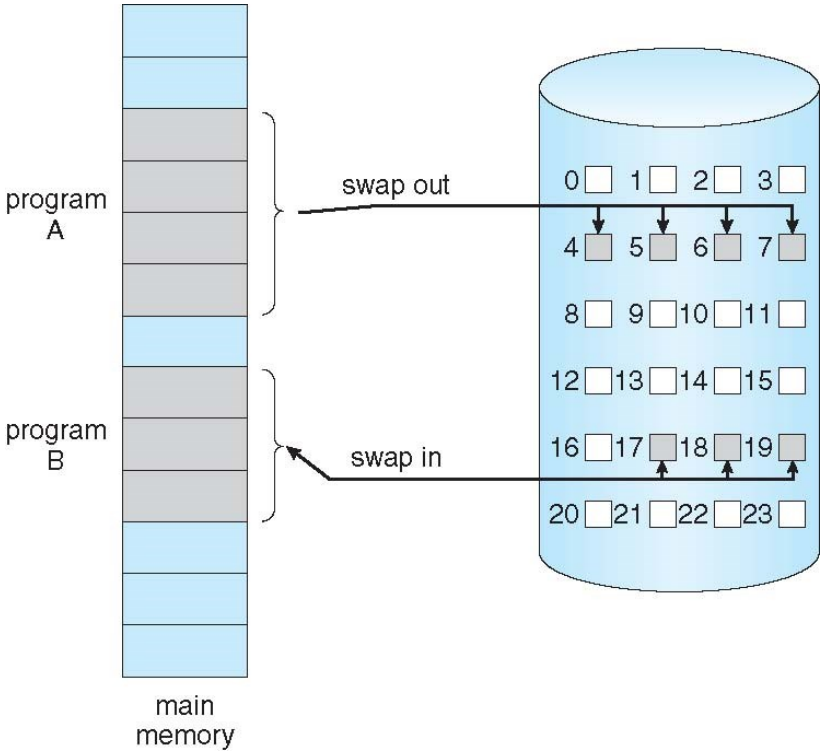
Implementation of Virtual Memory

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Demand Paging

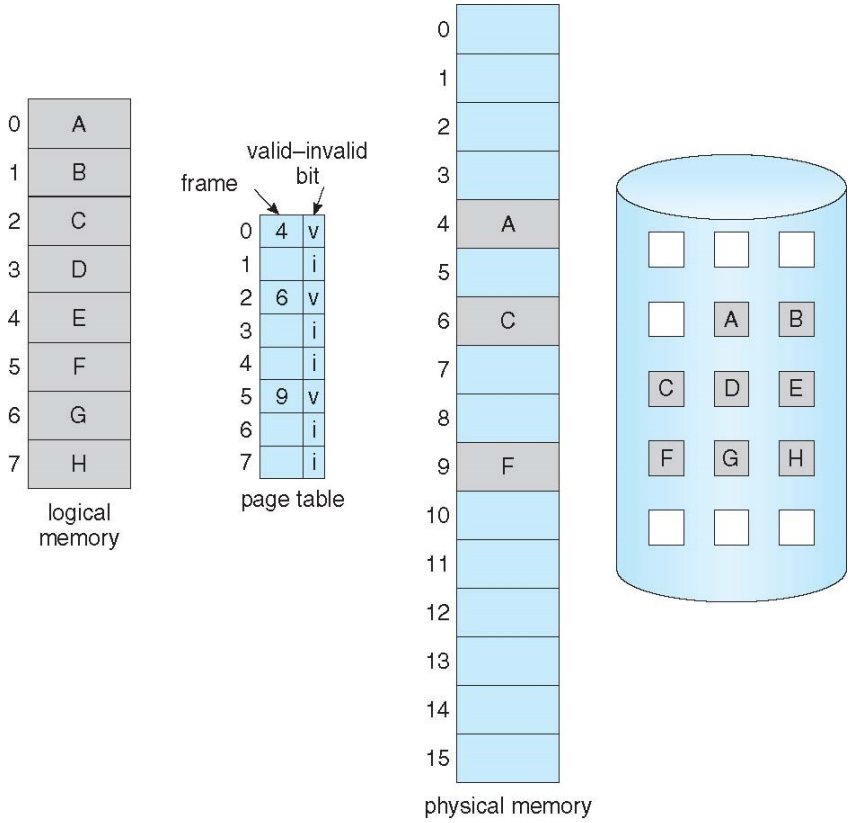
- First option for executing a program is brining all to the memory.
  - Initially, we may not need all the program in the memory.

- Loading pages only as they are demanded known as Demand Paging.
  - Pages that are never accesses are thus never loaded into physical memory.
  - A demand paging system is similar to a paging system with swapping where processes reside in secondary memory.



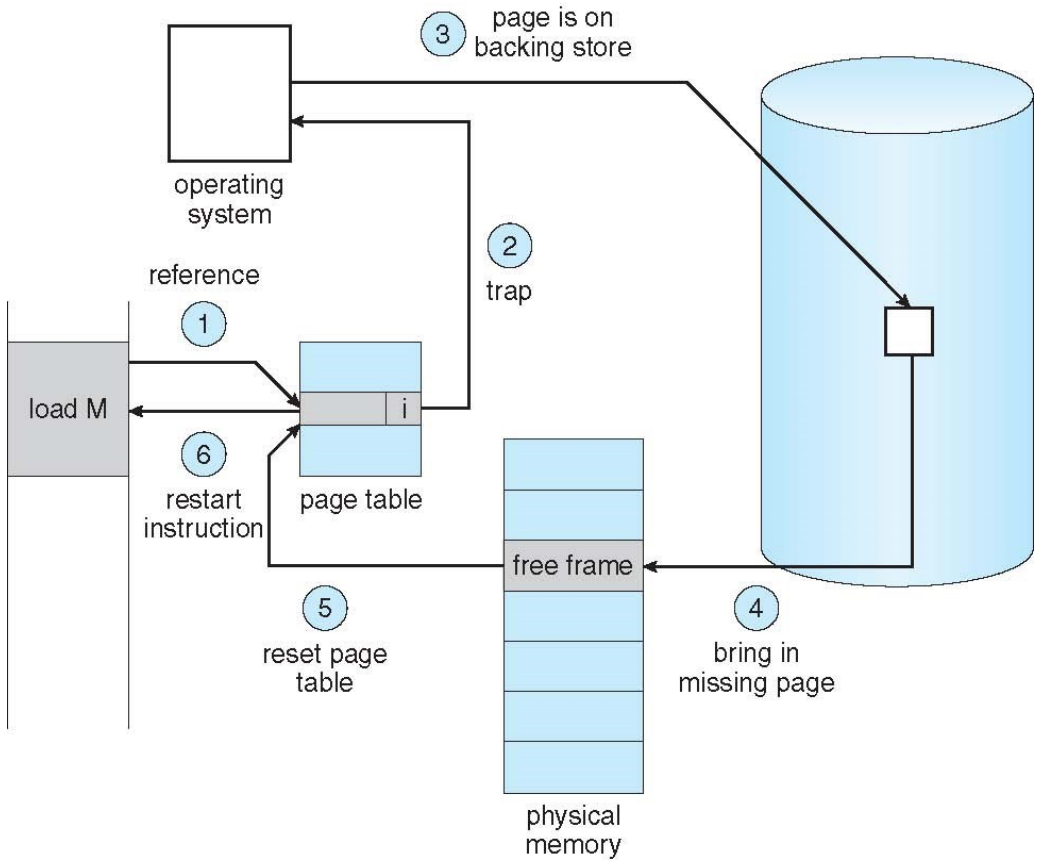
Demand Paging – Basic Concepts

- During process execution, some pages will be in memory, and some will be in secondary storage.
- How to distinguish them? Asking support from hardware through valid-invalid bit.
  - When the bit is set to **valid**, the associated bit is both legal and in memory.
  - If the bit is set to **invalid**, the page is either not in the logical address space of the process or is currently in secondary storage.



Demand Paging – Basic Concepts

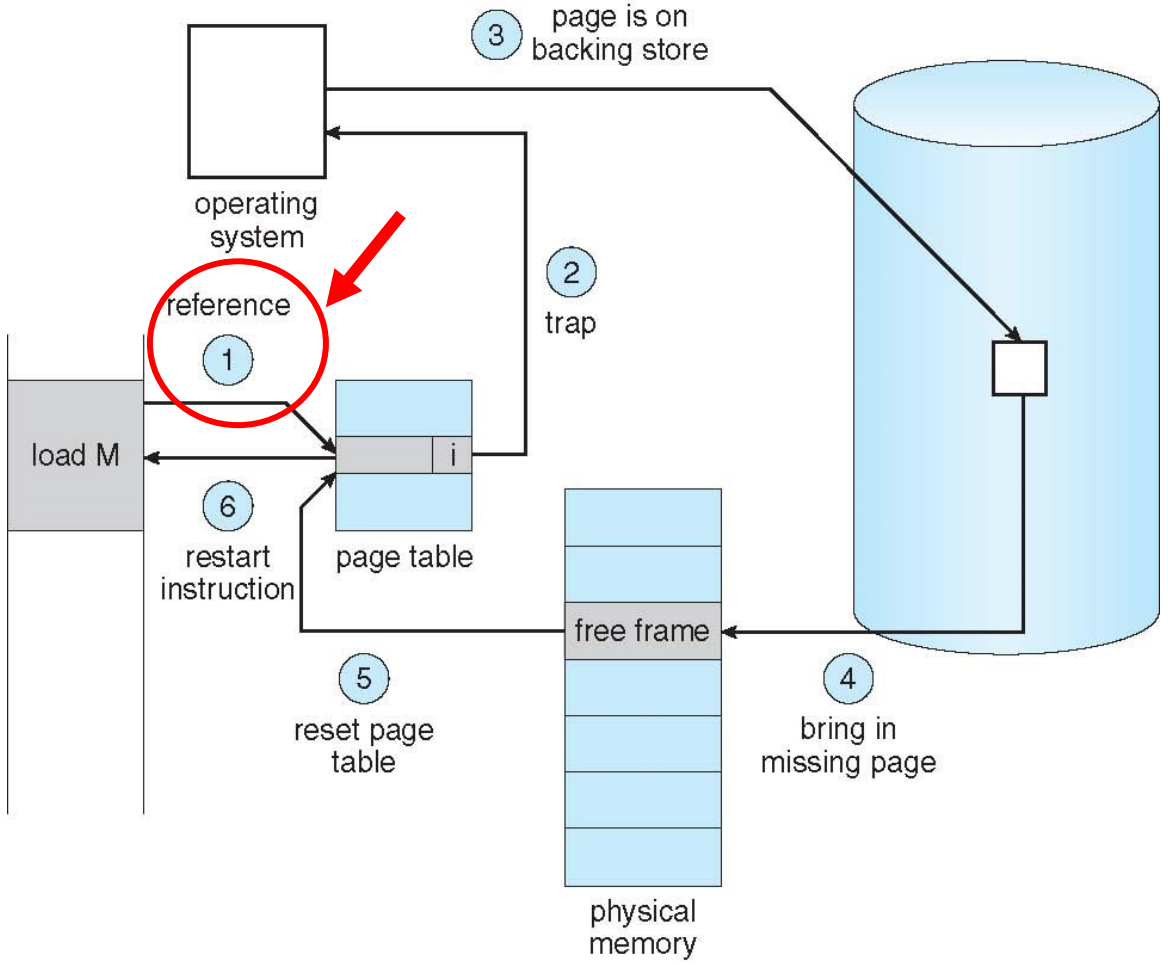
- What happens if the process tries to access a page that was not brought into memory?
  - An access to a page marked invalid caused a **page fault**.
- How is the page fault handled?
  1. Reference
  2. Trap
  3. Page is on backing store
  4. Bring in missing page
  5. Reset page table
  6. Restart instruction



Demand Paging – Basic Concepts

- How is the page fault handled?

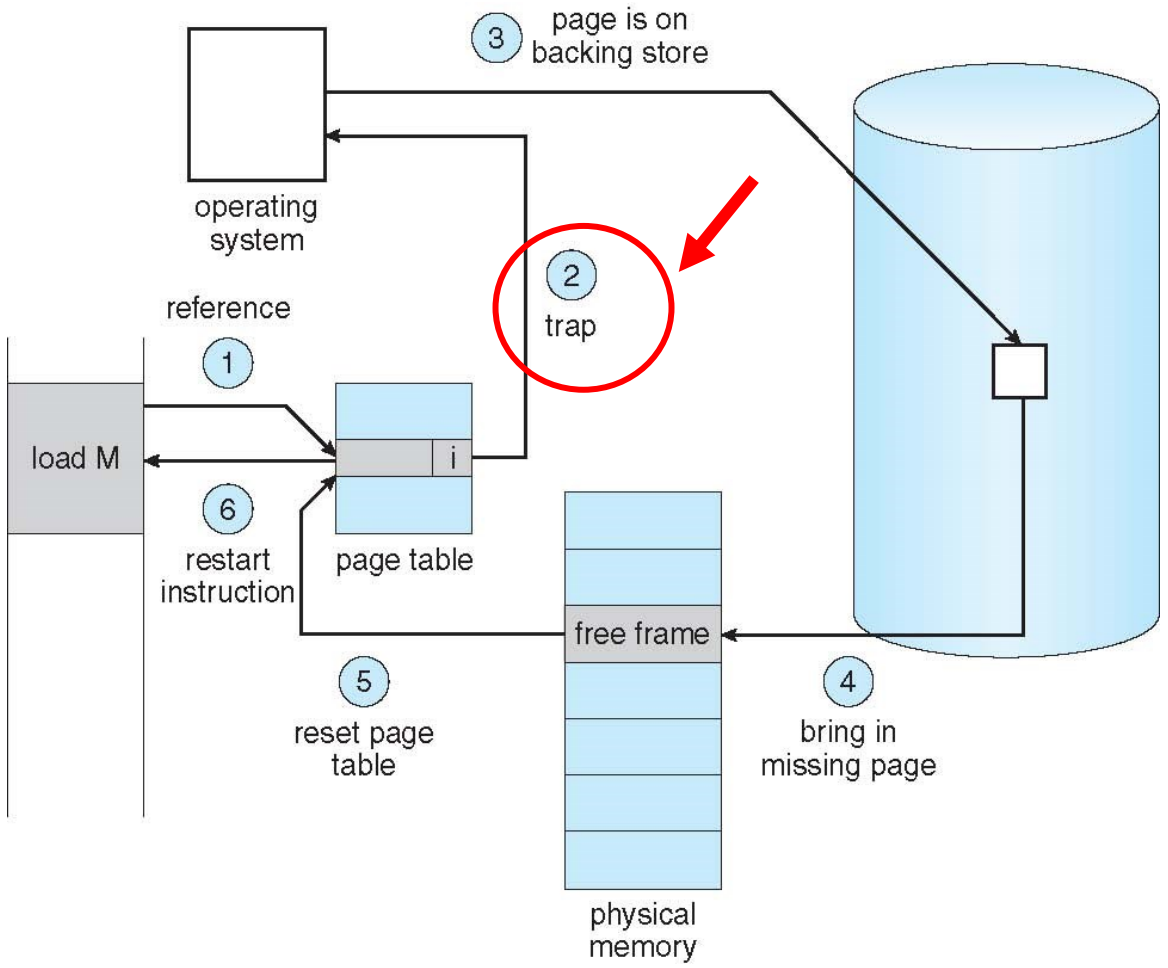
  1. We check an internal table for this process to determine whether the reference was a valid or an invalid memory.



Demand Paging – Basic Concepts

- How is the page fault handled?

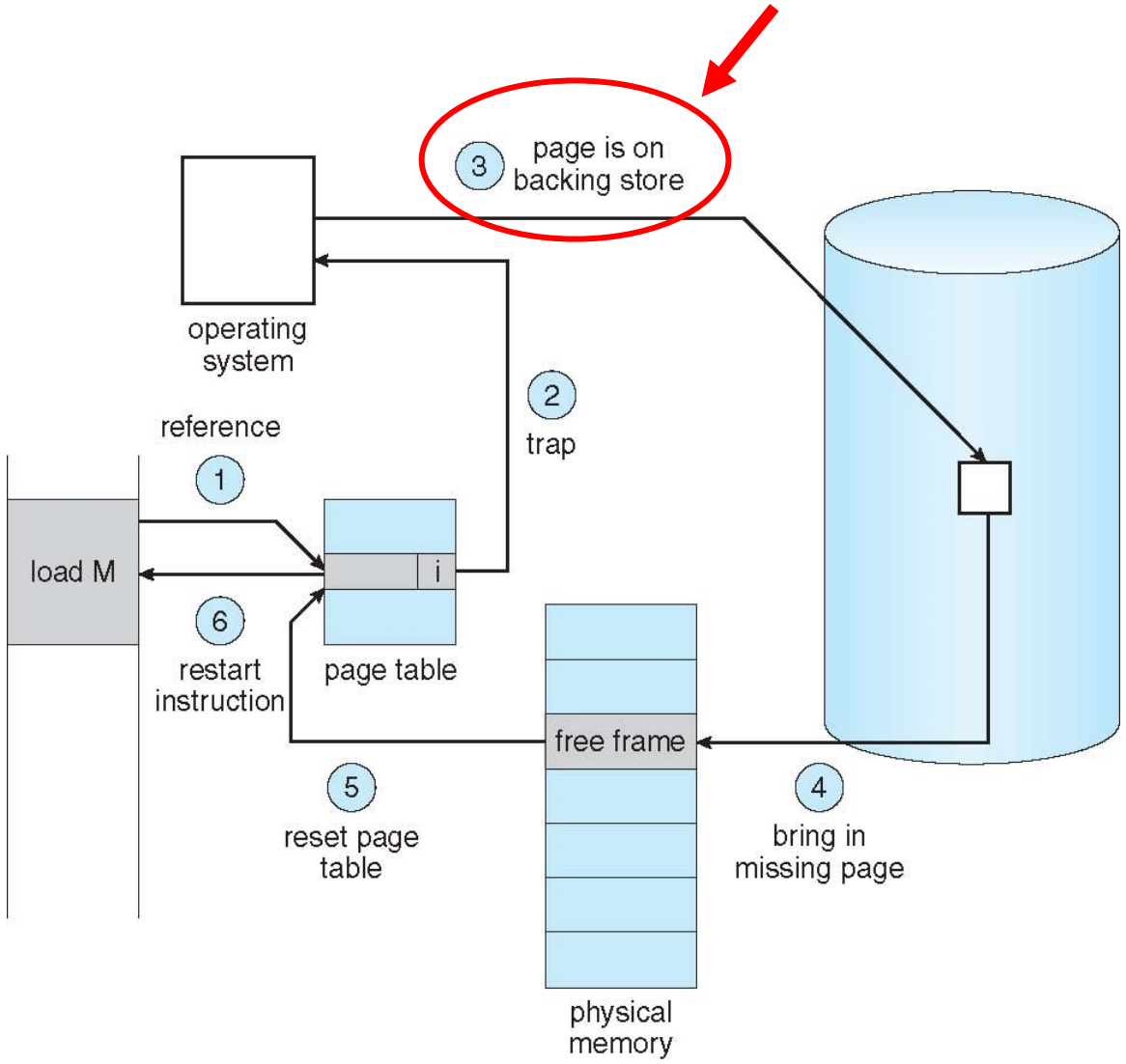
  1. Reference
  2. If the reference was invalid, we terminate the process, If it was valid but we have not yet brought in that page, we now page it in.



Demand Paging – Basic Concepts

- How is the page fault handled?

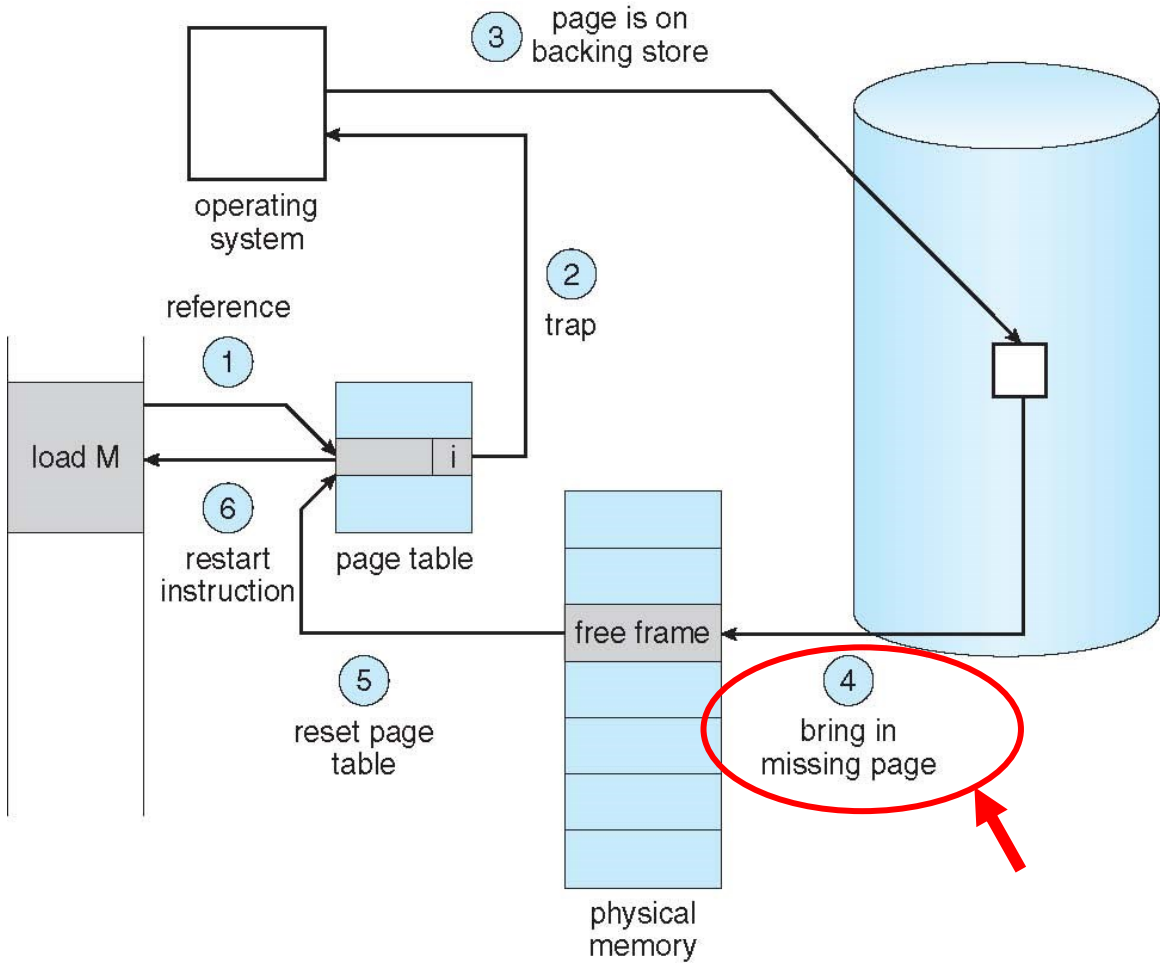
  1. Reference
  2. Trap
  3. We find a free frame



Demand Paging – Basic Concepts

- How is the page fault handled?

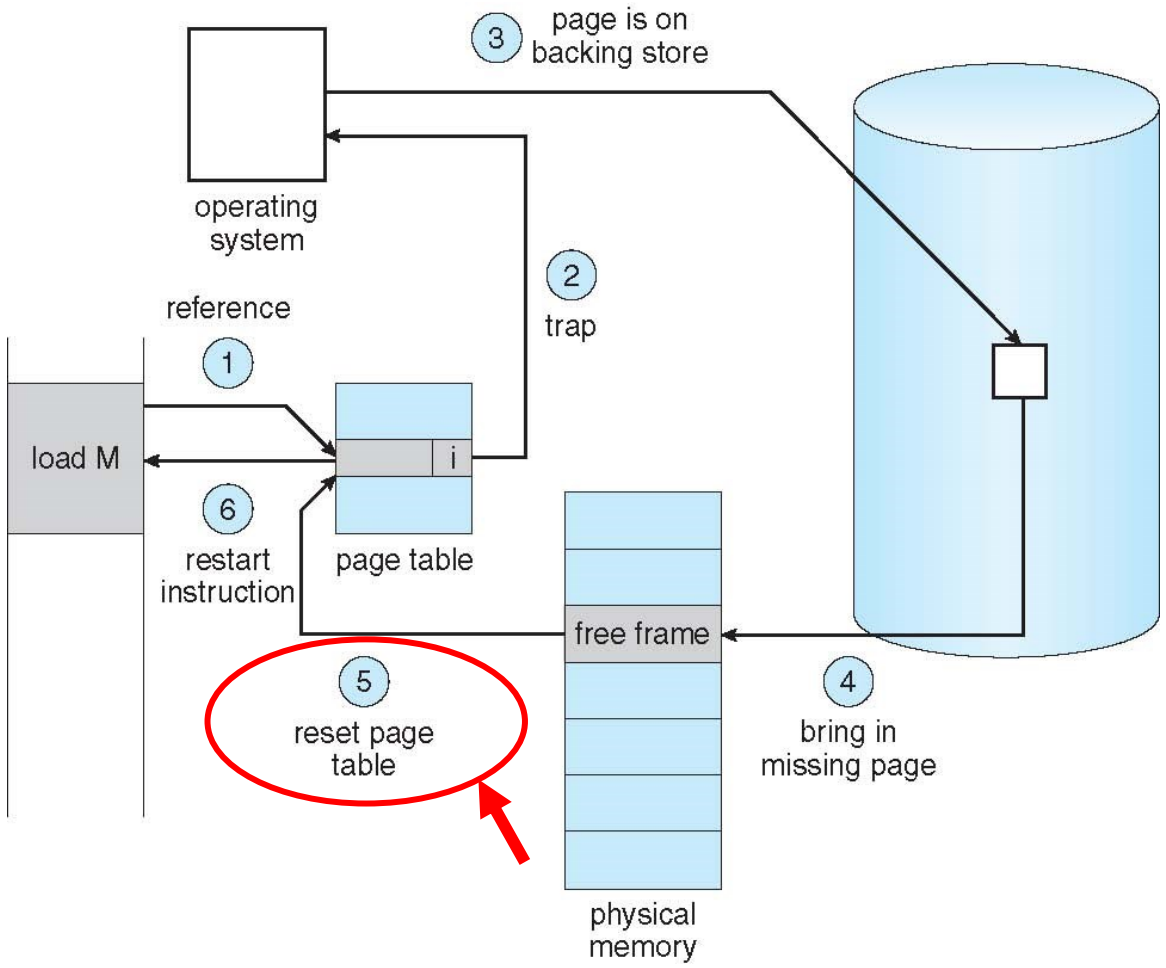
  1. Reference
  2. Trap
  3. Page is on backing store
  4. We schedule a secondary storage operation to read the desired page into the newly allocated frame.



Demand Paging – Basic Concepts

- How is the page fault handled?

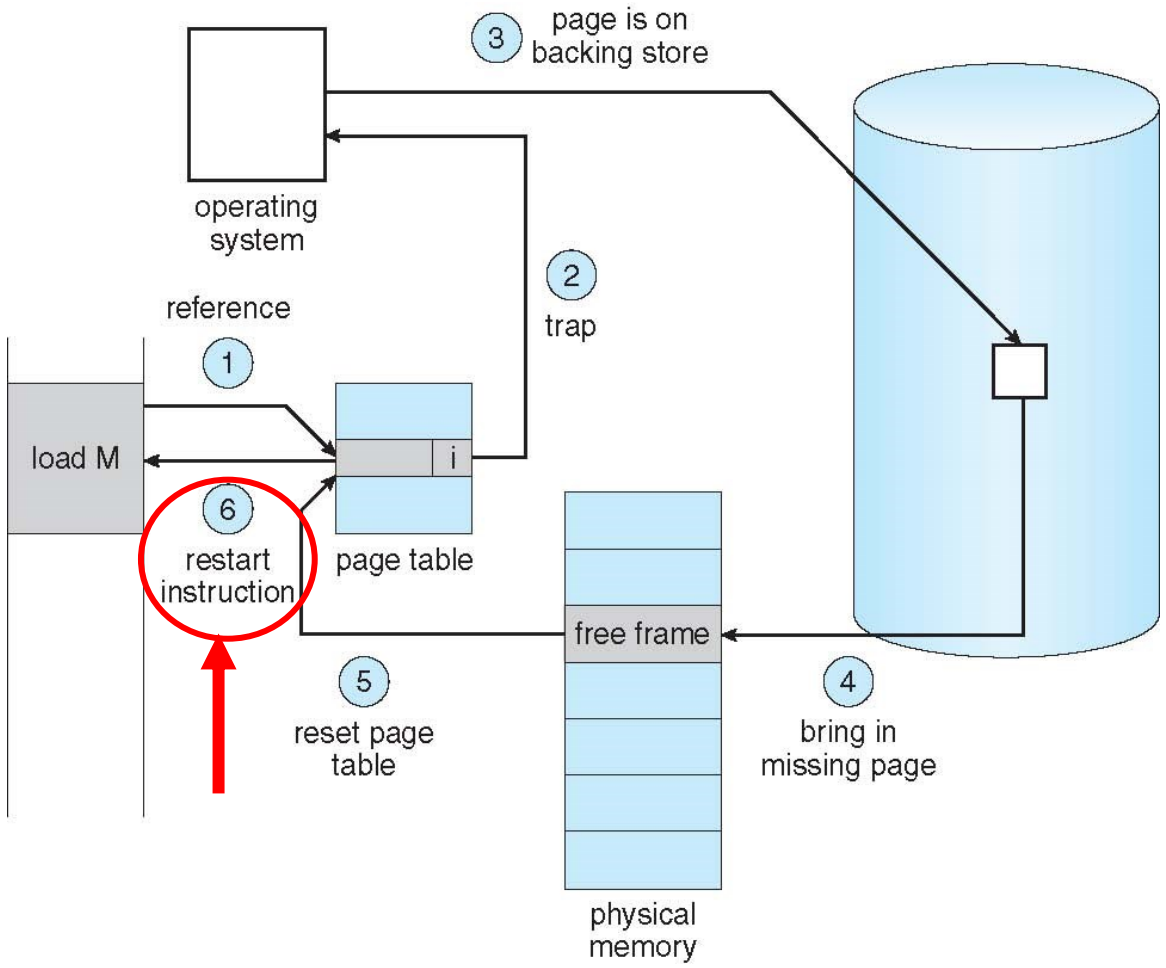
  1. Reference
  2. Trap
  3. Page is on backing store
  4. bring in missing page
  5. When the storage read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.



Demand Paging – Basic Concepts

- How is the page fault handled?

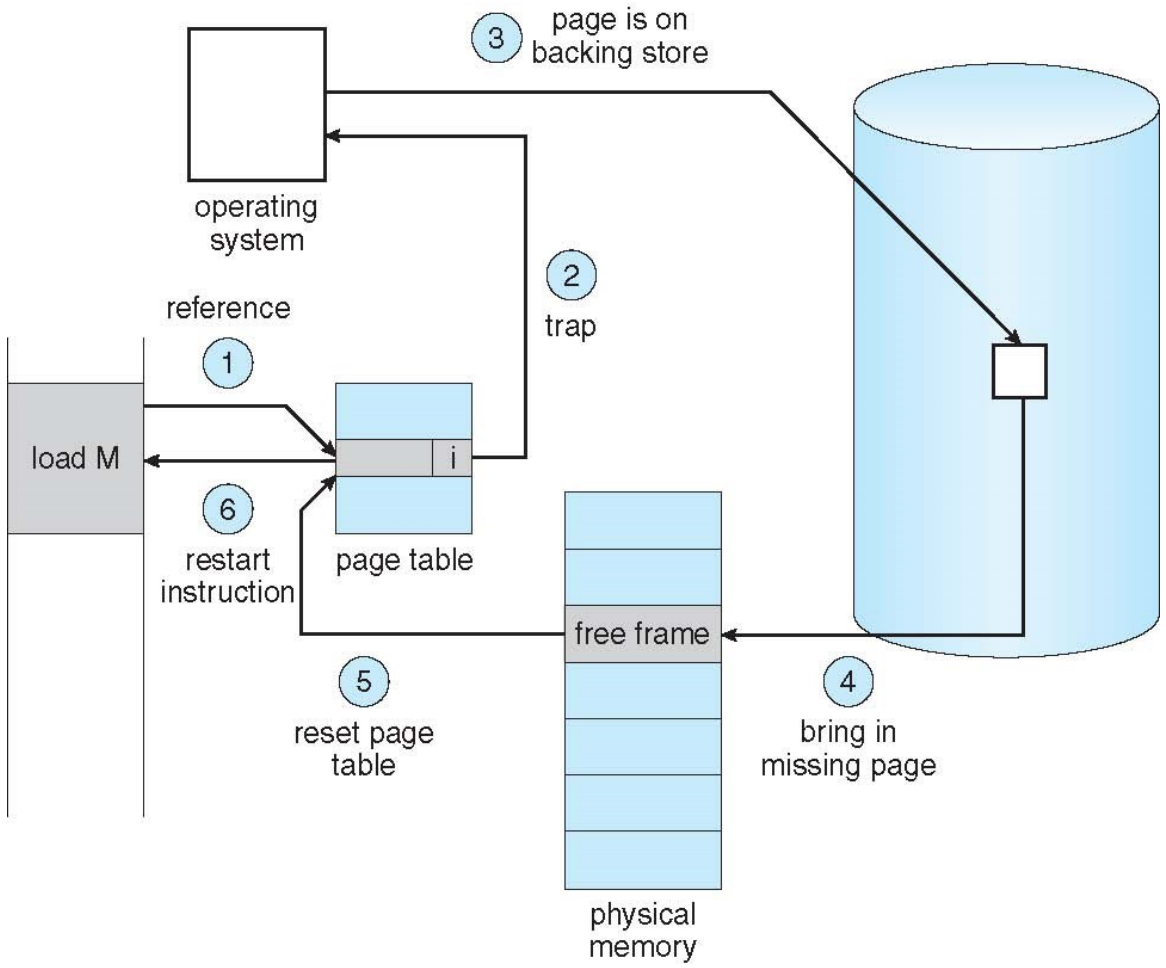
1. Reference
2. Trap
3. Page is on backing store
4. bring in missing page
5. reset page table
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.



Demand Paging – Basic Concepts

- How is the page fault handled?

  1. Reference
  2. Trap
  3. Page is on backing store
  4. bring in missing page
  5. reset page table
  6. Restart instruction



Aspects of Demand Paging

- Extreme case – start executing a 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**
- 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
  - This situation would result in unacceptable system performance.
  - However, it is not likely, because programs tend to have **locality of reference** which result in reasonable performance from demand paging.

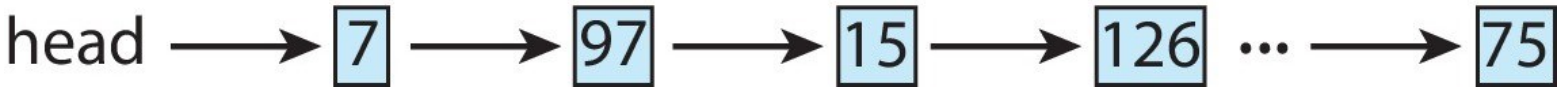
Aspects of Demand Paging

- Hardware support needed for demand paging is the same as the one needed for paging and swapping:
  - **Page table** with valid / invalid bit
  - **Secondary memory** is usually a high-speed disc knows as swap device with **swap space**

- A crucial requirement for demand paging is the ability to restart any instruction after a page fault. We must be able to restart the process in exactly the same place and state, except the desired page is now in memory and is accessible.

Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.



Free-Frame List

- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.
- As free frames are requested, the size of the free-frame list shrinks.

Performance of Demand Paging

- Computing the **Effective Access Time** for a demand-paged memory.
  - Assuming the memory access-time (ma) is 10 ns.
  - As long as we have no page faults, the effective access time is equal to the memory access time.
  - If a page fault occurs, we must first read the relevant page from secondary storage and then access the desired word.
  - Considering P as the probability of a page fault $(0 \leq p \leq 1)$

$$\text{Effective Access time} = (1\text{-}p) * ma + p * \text{page fault time}$$

Stages in Demand Paging – Worse Case

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

Stages in Demand Paging – Worse Case

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

Stages in Demand Paging – Worse Case

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

Not all these steps are necessary in every case.

Performance of Demand Paging

- 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 $0 <= p <= 1$
      - 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)

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

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

- The effective access time is directly proportional to the **page-fault rate**.

Demand Paging Example

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

- The effective access time is directly proportional to the **page-fault rate**.
  - If one access out of 1,000 causes a page fault, then EAT = 8.2 microseconds.
  - This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent

$$220 > 200 + 7{,}999{,}800 * p \quad 20 > 7{,}999{,}800 * p$$

$$p < .0000025$$

$$P < \text{one page fault in every } 400{,}000 \text{ memory accesses}$$

Exercise

- An operating system supports a paged virtual memory. The central processor has a cycle time of 1 microsecond. It costs an additional 1 microsecond to access a page other than the current one. Pages have 1,000 words, and the paging device is a drum that rotates at 3,000 revolutions per minute and transfer 1 million words per second. The following statistical measurements were obtained from the system:
  - One percent of all instructions executed accessed a page other than the current page.
  - Of the instructions that accessed another page, 80 percent accessed a page already in memory.
  - When a new page was required, the replaced page was modified 50 percent of the time.
  - Calculate the effective instruction time on this system, assuming that the system is running one process only and that the process is idle during drum transfers.

$$EAT = 0.99 * (1\mu s + 0.008 * (2\mu s) + 0.002 * (10{,}000\mu s + 1{,}000\mu s) + 0.001 *(10{,}000\mu s + 1{,}000\mu s) =$$
$$(0.99 + 0.016 + 22.0 + 11.0)\mu s = 34{,}5\mu s$$

Demand Paging Optimizations

- Another aspect of demand paging is the overall use of swap space.
  - First option for the system to gain better paging throughput is by copying an entire file image into the swap space at process startup and then performing demand paging from the swap space.
  - Second option is to demand-page from the file system initially but to write the pages to swap space as they are replaced.
    - Only needed pages are read from the file system but al the subsequent paging is done from swap space.
  - Demand pages are brought directly from the file system. However, when page replacement is called for, these frames can simply be overwritten.
    - Using this approach, the file system serves as the backing store, however, swap space must still be used for pages not associated with a file, known as **anonymous memory**.

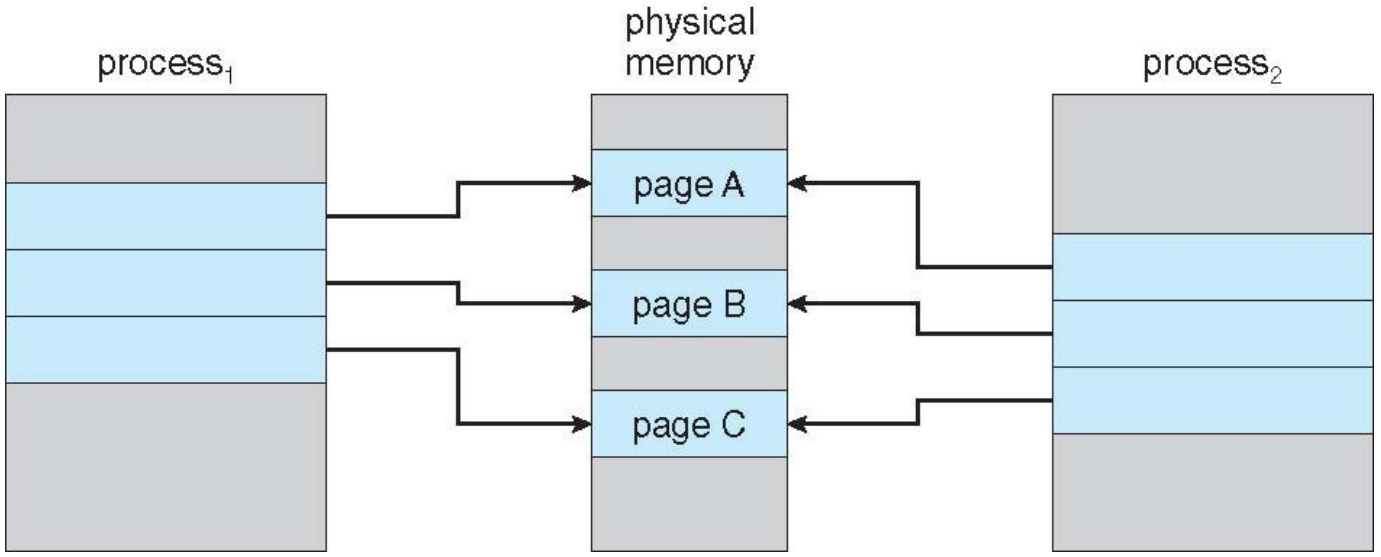
Copy-on-Write

- Process creation using the fork() system call may initially bypass the need for demand paging.
- Using a technique called **copy-on-write (COW)** that allows the parent and child processes to initially share the same pages.
  - If either process modifies a shared page, only then is the page copied.

Copy-on-Write

- Assume that the child process attempts to modify a page containing portion of the slack, with the pages set to be copy-on-write.
- The operating system will obtain a frame from the free-frame list and create a copy of this page, mapping it to the address space of the child process.
- The child will then modify its copied page and not the page belonging to the parent process.

- Only pages that are modified by either process are copies.
- All unmodified pages can be shared by the parent and child processes.

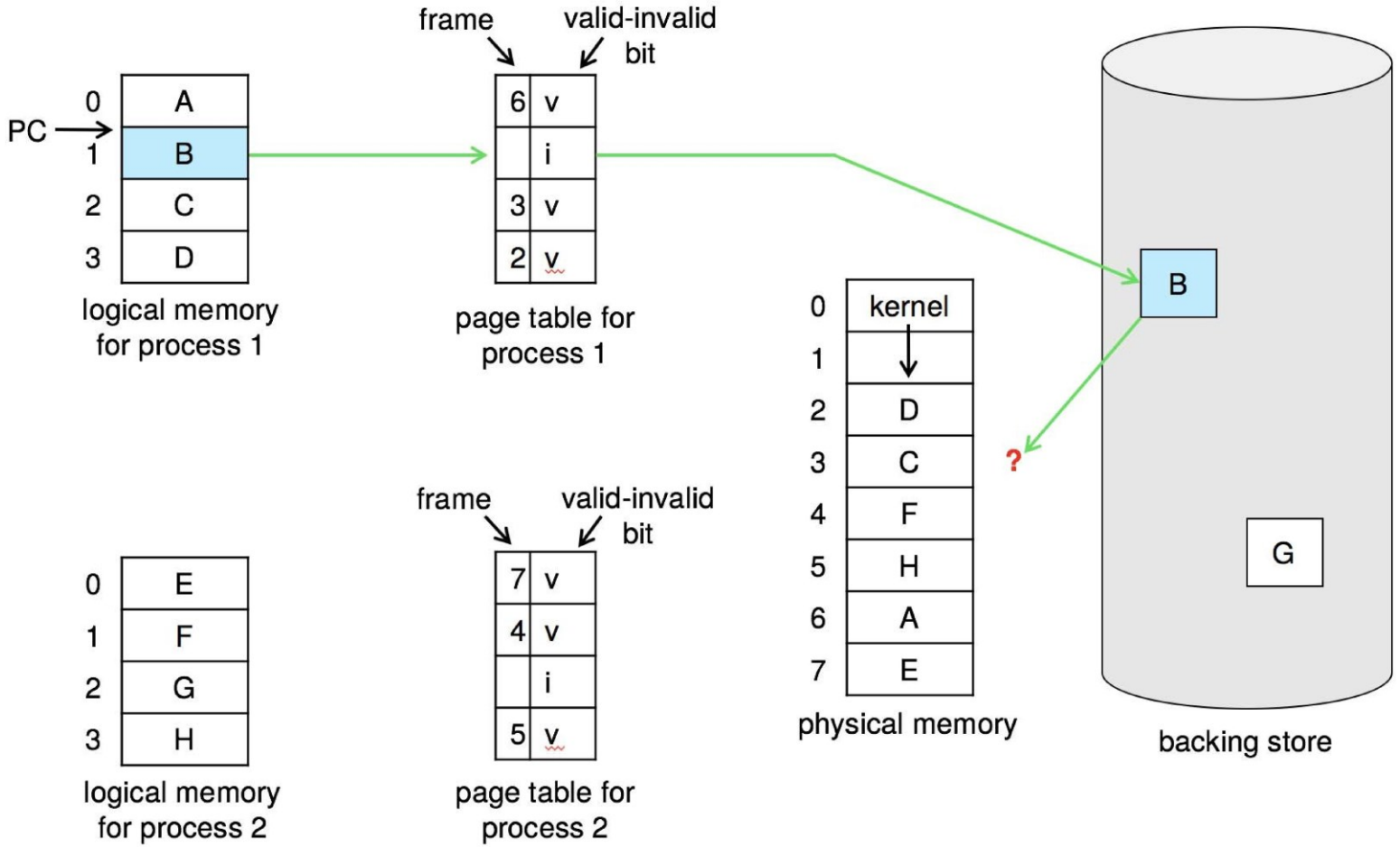


Page Replacement

- If we increase the degree of multiprogramming, we are **over-allocating memory**.
- While a process is executing, a page fault occurs.
- The operating system determines where ethe desired page is residing on secondary storage.
- There are no free frames, all memory is in use.
- Operating system can do standard swapping and swap out a process.
  - Standard swapping leads to overhead of copying entire processes between memory and swap space.
- Most operating system now combine swapping pages with **Page Replacement**.

Page Replacement

- Most operating system now combine swapping pages with **Page Replacement**.



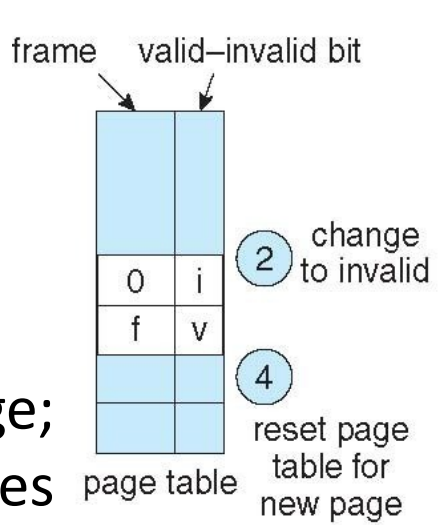
Basic Page Replacement

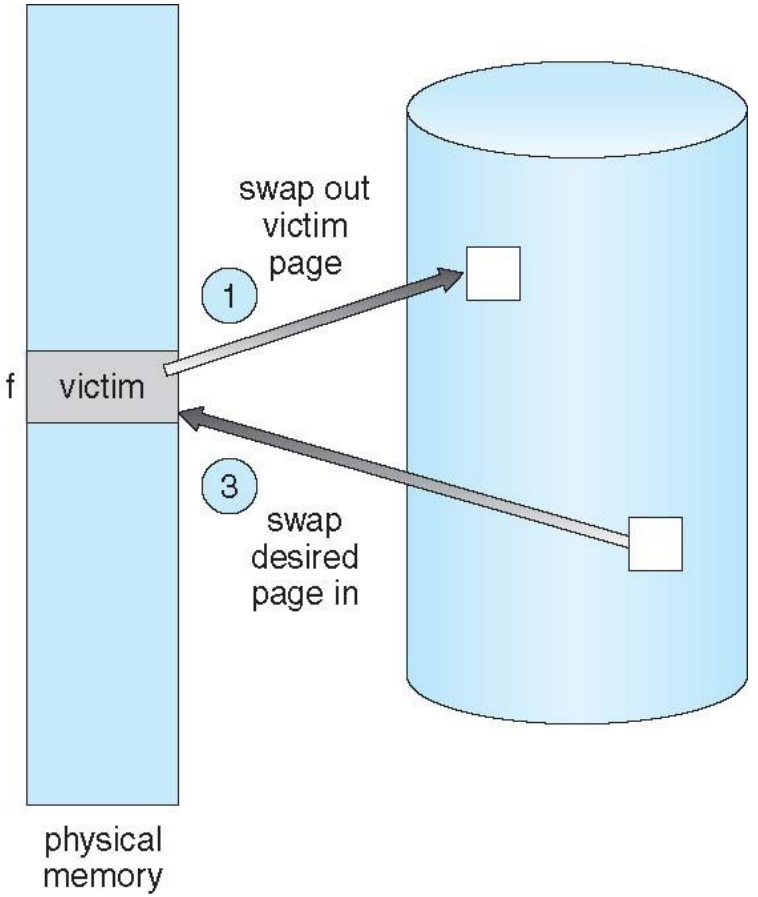
- Page replacement is: if no frame is free, we find one that is not currently being used and free it.
- Free frame by writing its content to swap space and changing the page table.
- Using the free frame to hold the page for which the process faulted.

Basic Page Replacement

- We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on secondary storage.
2. Find a free frame.
   A. If the is a free frame, use it,
   B. If not, select a **victim frame**.
3. Write the victim frame to secondary storage; change the page and frame tables accordingly.
4. Read the desired page into the freed frame.
5. Continue the process from where the page fault occurred.





Page Replacement

- If no frames are free, two-page transfers are required.
  - This doubles the page-fault service time.

- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers.
  - The modify bit for a page is set by the hardware whenever any bytes in the page is written into, indicating that the page has been modified.
  - If the bit is set, the page is modified since it was read, we must write the page to storage.
  - If the bit is not set, we do not need to write the page to the storage, it is already there.

Page and Frame Replacement Algorithms

- Two major problems to be solved for implementing demand paging:
- **Frame-allocation algorithm**
  - If we have multiple process in memory, we must decide how many frames to allocate to each process.
- **Page-replacement algorithm**
  - When page replacement is required, we must select the frames that are to be replace.
  - Want lowest page-fault rate

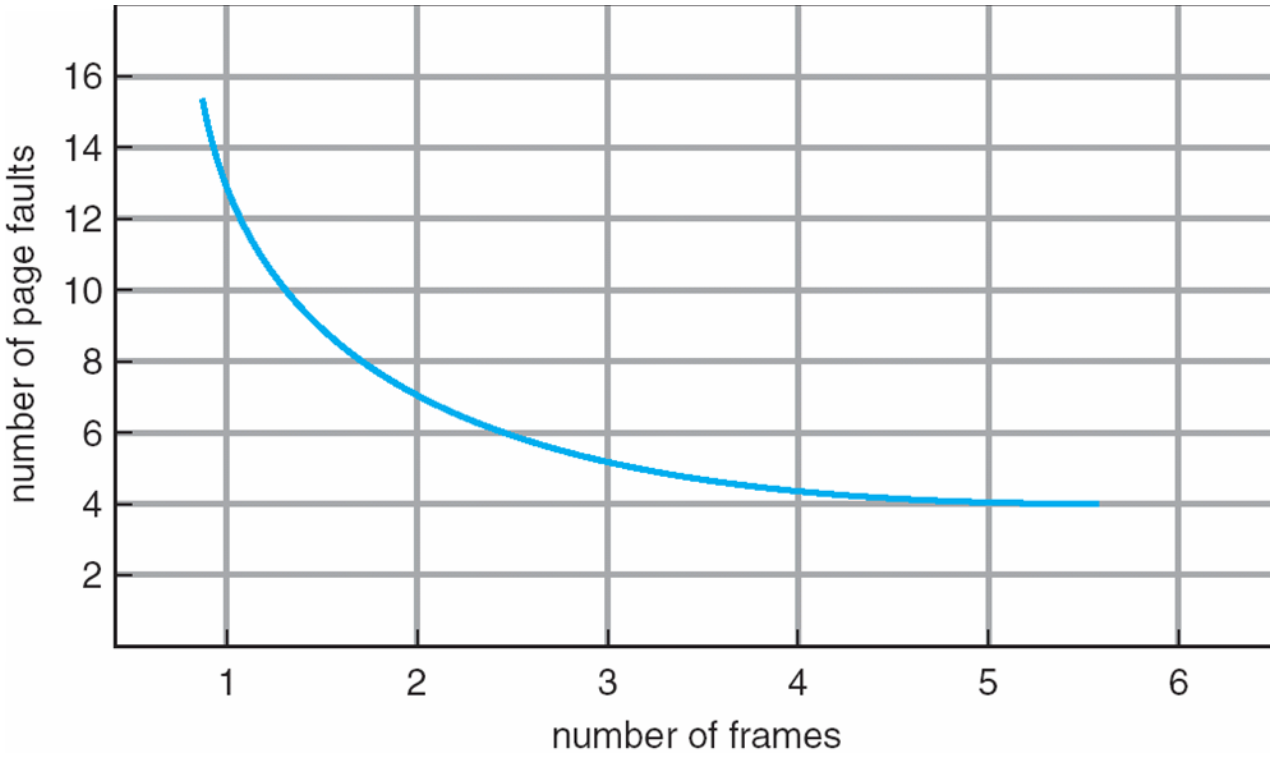
Page and Frame Replacement Algorithms

- How to select a particular replacement algorithm?
- 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

- As the number of frames increases, the number of page faults drops to some minimal level.



Page replacement strategies

Page Fault Frequency (empirical probability)

$$f(A,m) = \sum_{\forall w} p(w) \frac{F(A,m,w)}{len(w)}$$

- *A* page replacement algorithm under evaluation
- *w* a given reference string
- *p*(*w*) probability of reference string *w*
- *len*(*w*) length of reference string *w*
- *m* number of available page frames
- *F*(*A*,*m*,*w*) number of page faults generated with the given reference string (*w*) using algorithm *A* on a system with *m* page frames.

Page Replacement Algorithms

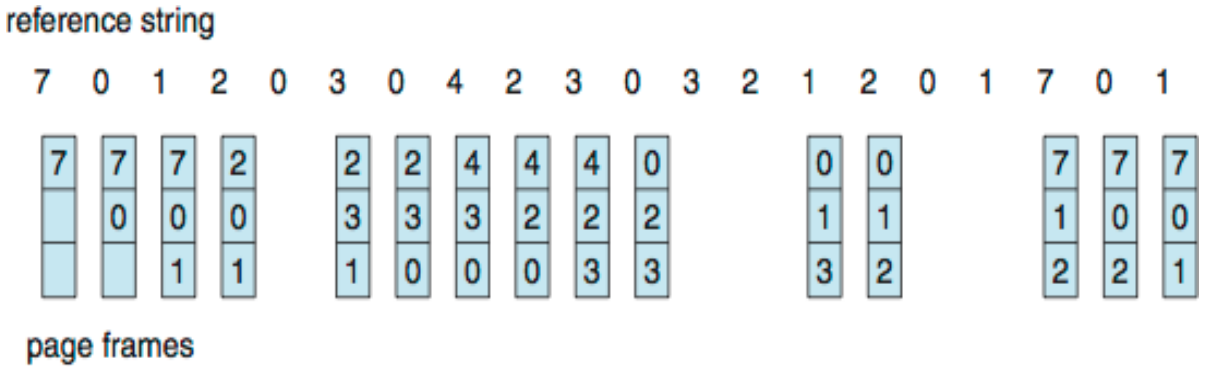
- There are many different page-replacement algorithms:
  - FIFO Page Replacement
  - Optimal Page Replacement
  - LRU Page Replacement
  - LRU-approximation Page Replacement
  - Counting-based Page Replacement
  - Page-Buffering Algorithm
  - Applications and Page Replacement

First-In-First-Out (FIFO) Algorithm

- The FIFO replacement algorithm associates with each page the time when the page was brought into memory.
- When a page must be replaced, the oldest page is chosen.
- It is not necessary to record the time when a page is brought in.
  - Creating a FIFO queue to hold all pages in memory.
  - Replacing the page at the head of the queue
  - Inserting the page that is brought into memory at the tail of the queue.

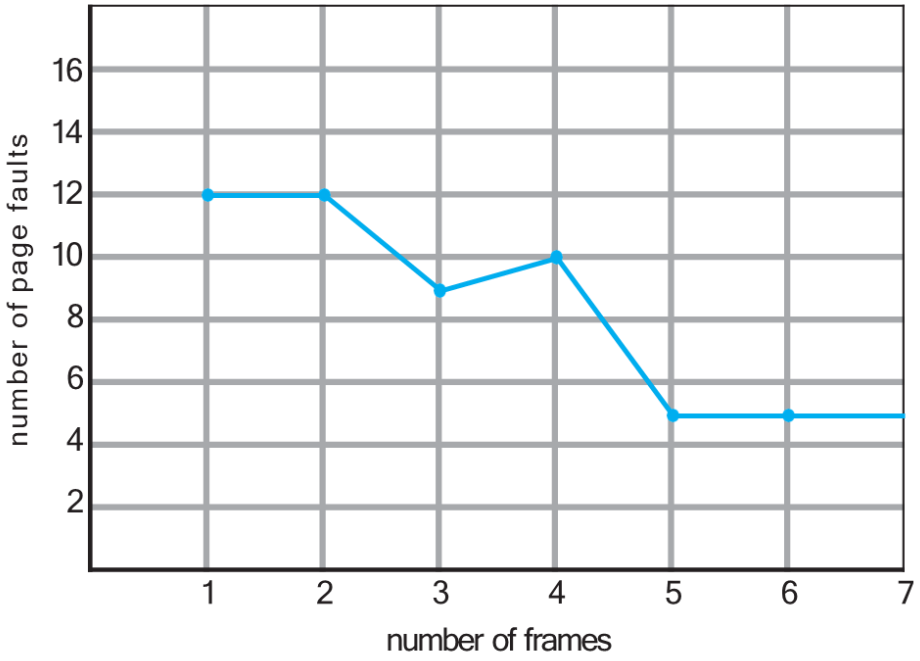
First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**



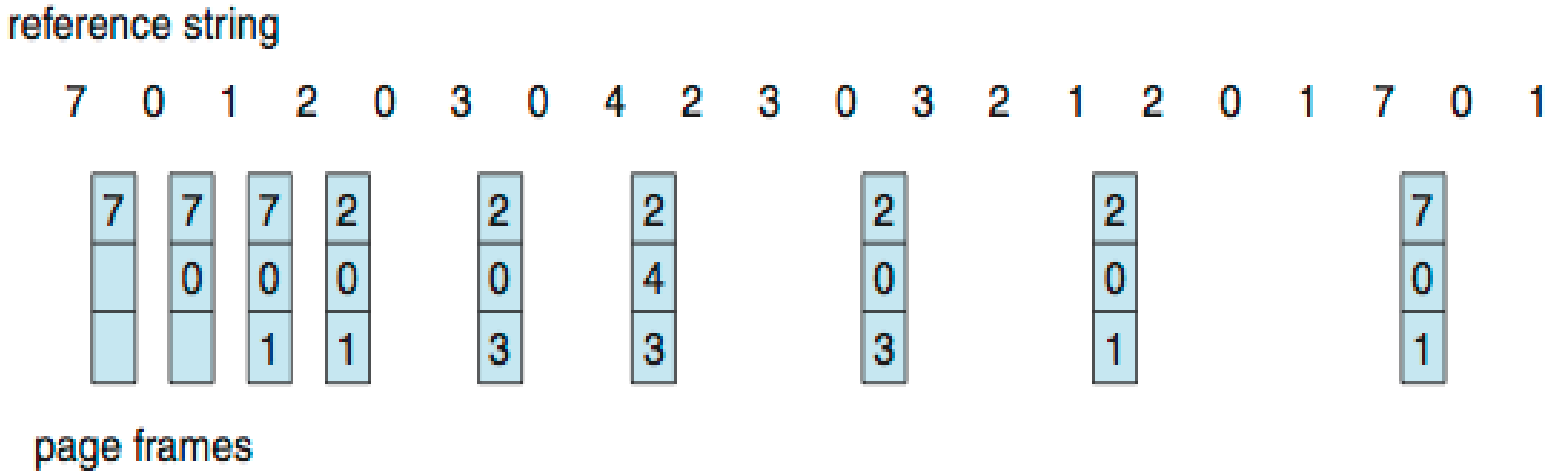
FIFO Illustrating Belady's Anomaly

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - The number of faults for four frames (10) is greater than the number of faults for three frames (9).
  - This effect is called **Belady's anomaly**.
    - For some page-replacement algorithms, the page-fault rate ay increase as the number of allocated frames increase.



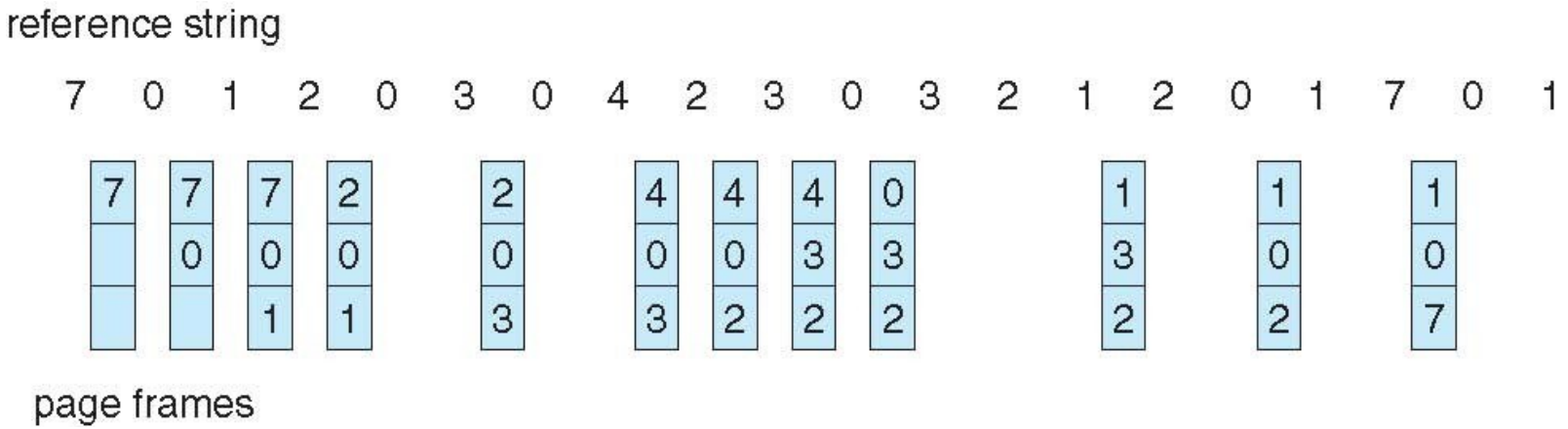
Optimal Algorithm

- Replace page that will not be used for longest period of time
  - Guaranteeing the lowest possible page-fault rate for a fixed number of frames.
  - Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1 >> 9 page fault
  - Difficult to implement since it requires future knowledge of the reference string.
    - It is used mainly for comparison study, measuring how well a new algorithm performs.



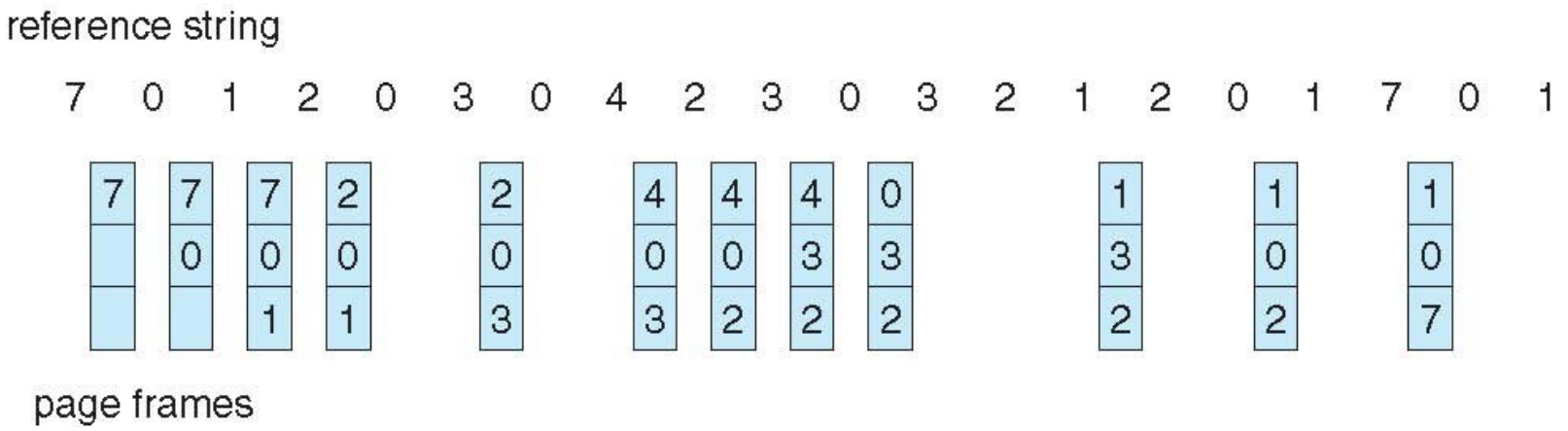
Least Recently Used (LRU) Algorithm

- 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
  - The main difference between the FIFO and OPT algorithms is that FIFO uses the time when a page was brought into memory, OPT uses the time when a page is to be used.



Least Recently Used (LRU) Algorithm

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



LRU Algorithm Implementation

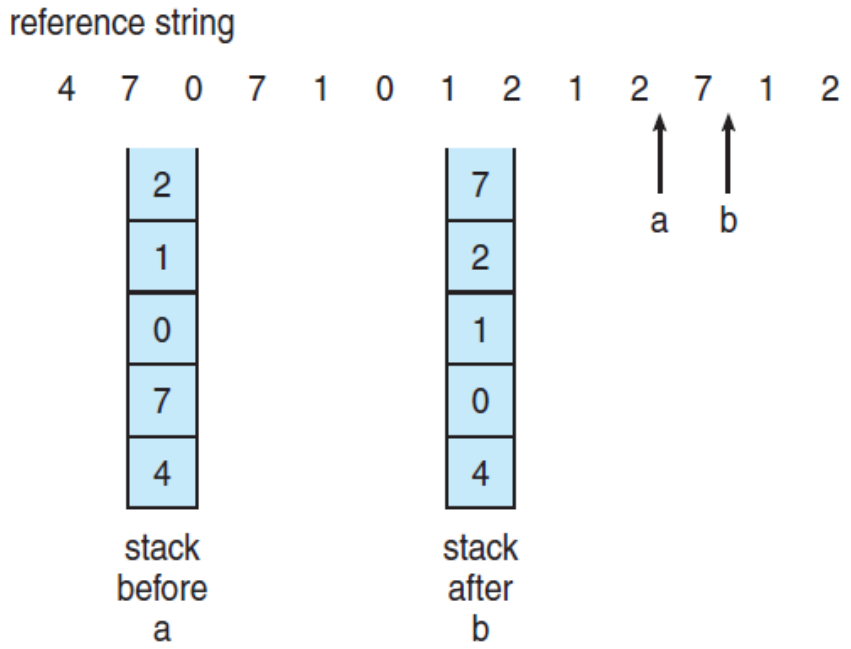
- Two implementation is feasible:
  - Counters
    - Associating with each page-table entry, a time-of-use field and ass to the CPU a logical clock or counter.
  - Stack
    - Keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top.

LRU Algorithm Implementation - Counter

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
  - In this way, we always have the time to the last reference to each page.
  - When a page needs to be changed, look at the counters to find smallest value
    - This scheme requires the search through the page table to find the LRU page and write to memory for each memory access.

LRU Algorithm - Stack

- Stack implementation
  - Keep a stack of page numbers
  - Whenever a page is referenced, it is removed from the stack and put on the top.
  - In this way, the most recently used page is always at the top of the stack and the least recently on the bottom.



LRU Algorithm - Stack

- Stack implementation
  - Since entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer.
  - Removing a page and putting it on the top of the stack requires changing six pointers at worst.
  - Each update is a little more expensive, but there is no search for a replacement.
  - The tail pointer points to the bottom of the stack, which is the LRU page.

LRU Algorithm - Stack

- Stack implementation
  - Since entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer.
  - Removing a page and putting it on the top of the stack requires changing six pointers at worst.
  - Each update is a little more expensive, but there is no search for a replacement.
  - The tail pointer points to the bottom of the stack, which is the LRU page.

  - LRU and OPT are cases of **stack algorithms** that do not have Belady's Anomaly.

LRU Approximation Algorithms

- LRU needs special hardware and still slow. Many systems provide some help in form of **Reference bit**
  - Reference bits are associated with each entry in the page table.
  - Initially, all bits are cleared (to 0) by the operating system.
  - When page is referenced, the bit set to 1 by the hardware.
  - After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the order of use.
  - This is referred to as approximate LRU algorithm.

Second-chance Algorithm

- Second-chance algorithm is a FIFO replacement algorithm
- When a page has been selected, however, we inspect its reference bit.
  - If the value is 0, we proceed to replace this page.
  - If the value is 1:
    - We give the page a second chance and move on to select the next FIFO page.
    - When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time.
  - replace next page, subject to same rules

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit
- With these two bits, we have the following four possible classes:
  - (0, 0) neither recently used nor modified – best page to replace
  - (0, 1) not recently used but modified – not quite as good, must write out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement is called for, use the clock scheme but use the four classes to replace the page in the lowest non-empty class
  - Might need to search circular queue several times

Counting-based 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
  - The problem is when a page is heavily used during the initial phase of a process but then is never used again.

- **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

- Systems commonly keep a pool of free frames, always
  - When a page fault occurs, a victim frame is chosen as before.
  - The desired page is read into a free frame from the pool before the victim is written out.
  - The process restart as soon as possible, without waiting for the victim page to be written out.
  - The victim is the written out, its frame is added to the free-frame pool.

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

Exercise:

- Consider the following page reference string:

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

- How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames?
- Remember that all frames are initially empty, so your first unique pages will cost one fault each.
  1. LRU replacement
  2. FIFO replacement
  3. Optimal replacement

Frame #	LRU	FIFO	Optimal
1	20	20	20
2	18	18	15
3	15	16	11
4	10	14	8
5	8	10	7
6	7	10	7
7	7	7	7

Exercise:

- Considering the following page reference string: 7,2,3,1,2,5,3,4,6,7,7,1,0,5,4,6,2,3,0,1
- Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms:
  - LRU Replacement
  - FIFO Replacement
  - Optimal Replacement

- LRU Replacement >> 18
- FIFO Replacement >> 17
- Optimal Replacement >> 13

Exercise:

Consider the two-dimensional array A: int A[][] = new int [100][100];

- Where A[0][0] is at location 200 in a paged memory system with pages of size 200. A small process that manipulates the matrix resides in page 0 (location 0 to 199). Thus, every instruction fetch will be from page 0.
  - For three-page frames, how many page fault are generated by the following array_initialization loop? Use LRU replacement and assume that page frame 1 contains the process and the other two are initially empty.

a.
```
For (int j = 0; j < 100; j++)
        for (int i = 0; i < 100; i ++){
                A[i][j] = 0;
        }
```

answer = 5,000

b.
```
For (int i = 0; i < 100; i++)
        for (int j = 0; j < 100; j ++){
                A[i][j] = 0;
        }
```

answer = 50

Allocation of Frames

- How do we allocate the fixed amount of free memory among various processes?
  - If we have 93 free frames and two processes, how many frames does each process get?
    - Operating system may take 35 frames, 93 frames for user space.
    - Pure demand paging, all 93 frame be put on free-frame list.
    - The first 93 pages faults would all get free frames from the free-frame list.
    - For a new page demand, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with 94th.
    - When the process terminated, the 93 frames would be placed on free-frame list.

Allocation of Frames

- How do we allocate the fixed amount of free memory among various processes?
  - If we have 93 free frames and two processes, how many frames does each process get?
    - Operating system may take 35 frames, 93 frames for user space.
    - Pure demand paging, all 93 frame be put on free-frame list.
    - The first 93 pages faults would all get free frames from the free-frame list.
    - For a new page demand, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with 94^{th}.
    - When the process terminated, the 93 frames would be placed on free-frame list.
- There are many variation on this simple strategy.

Allocation of Frames

- How do we allocate the fixed amount of free memory among various processes?
  - If we have 93 free frames and two processes, how many frames does each process get?
- Each process needs ***minimum*** number of frames
  - When a page fault occurs before an execution of the instruction is complete, the instruction must be restarted.
  - We need enough frames to hold all the different pages that any single instruction can reference.
- ***Minimum*** number of frames per process is defined by the architecture.
- ***Maximum*** is the total frames in the system.

Allocation of Frames

- Two major allocation schemes
  - Equal Allocation
  - Proportional Allocation

Equal Allocation

- **Equal allocation** – to split ***m*** frames among ***n*** processes is to give everyone an equal share, ***m/n*** frames (ignoring frames needed from OS for now).
  - For example, if there are 93 frames (after allocating frames for the OS) and 5 processes, give each process 18 frames
  - Keep the 3 left as free frame buffer pool.

Proportional Allocation

- **Proportional allocation** – allocation of available memory to each process according to its size.
  - Considering a system with a 1KB frame size.
  - If a small student process of 10KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames.
    - The student process does not need more than 10 frames.

Proportional Allocation

- **Proportional allocation** – allocation of available memory to each process according to its size.

s_i = size of virtual memory for process p_i

$S = \sum s_i$

m = total number of available frames

Allocating a_i frames to the process p_i, where a_i is approximately >> $a_i = S_i / S * m$

a_i should be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding m.

Proportional Allocation

- **Proportional allocation** – allocation of available memory to each process according to its size.

s_i = size of virtual memory for process p_i

$S = \sum s_i$

m = total number of available frames

Allocating a_i frames to the process p_i, where a_i is approximately >> $a_i = S_i / S * m$

Splitting 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames.

Proportional Allocation

- **Proportional allocation** – allocation of available memory to each process according to its size.
  - If the multiprogramming level increases, each process will lose some frames to provide the memory needed for the new process and vice versa.
  - Either in equal or proportional allocation, a high-priority process is treated the same as a low-priority process.
  - However, you may want to give the high-priority process more memory to speed its execution.
    - One solution is to use a proportional allocation scheme wherein the ration of frames depend not on the relative sizes of process but rather on the priorities of processes or on a combination of size and priority.

Global vs. Local Allocation

- With multiple processes competing for frames, we can classify the page-replacement algorithm into two broad categories:
  - **Global replacement** – process selects a replacement frame from the set of all frames even if the frame is currently allocated to some other process
    - But greater throughput so more common
  - **Local replacement** – each process selects from only its own set of allocated frames

Global vs. Local Allocation

- With multiple processes competing for frames, we can classify the page-replacement algorithm into two broad categories:
  - **Global replacement** – process selects a replacement frame from the set of all frames even if the frame is currently allocated to some other process
    - Set of pages in memory for a process depends not only on the paging behavior of that process, but also on the paging behavior of other processes.
    - The process execution time can vary greatly.

Global vs. Local Allocation

- With multiple processes competing for frames, we can classify the page-replacement algorithm into two broad categories:
  - **Local replacement** – each process selects from only its own set of allocated frames.
    - The set of pages in memory for a process is affected by the paging behavior of only that process.
    - More consistent per-process performance.

Global vs. Local Allocation

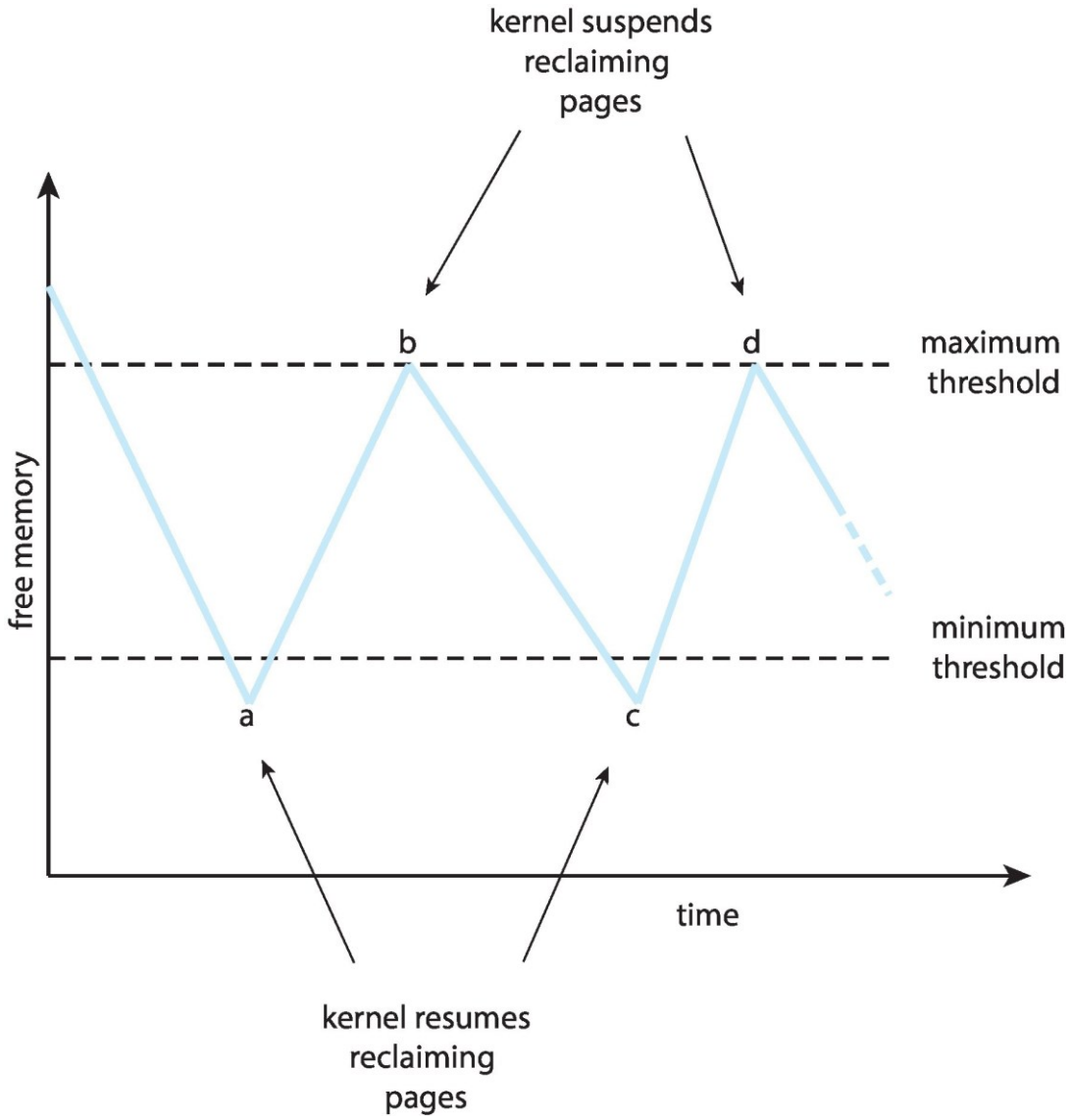
- With multiple processes competing for frames, we can classify the page-replacement algorithm into two broad categories:
  - **Local replacement** – each process selects from only its own set of allocated frames.
    - The set of pages in memory for a process is affected by the paging behavior of only that process.
    - More consistent per-process performance.
- Generally, the global replacement results in greater system throughput, therefore, it is used more commonly.

Reclaiming Pages

- A strategy to implement a global page-replacement policy
- All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero, before, we begin selecting pages for replacement.
- Page replacement is triggered when the list falls below a certain threshold.
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.

Reclaiming Pages Example

- When it drops below this threshold, a kernel routine is triggered known as **reapers**.
  - It begins reclaiming pages from all processes in the system.
  - When the amount of free memory reaches the max threshold, the reaper routing is suspended.



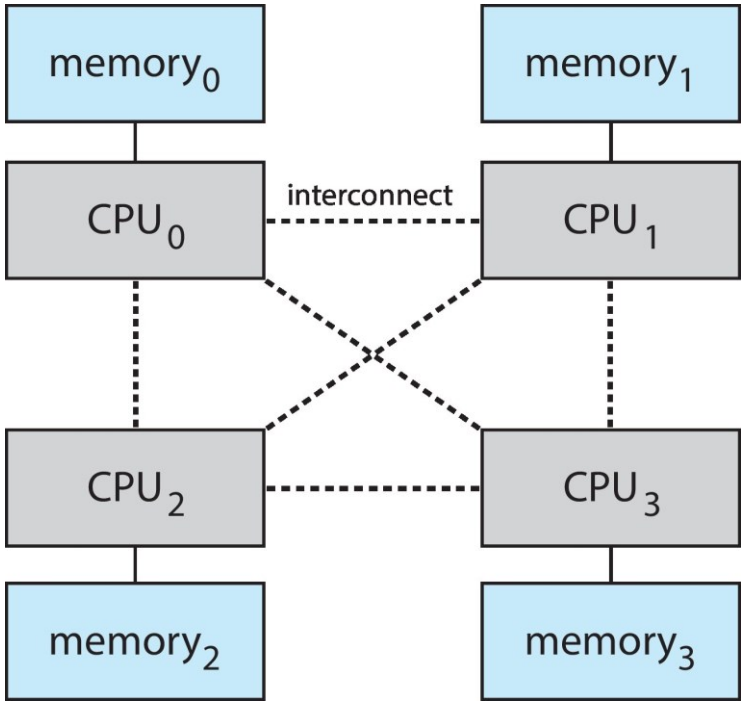
Non-Uniform Memory Access

- So far all memory accessed equally.
- On **Non-Uniform Memory Access** (NUMA), systems with multiple CPU, that is not the case.
  - a given CPU can access some section of main memory faster than it can access others.

Non-Uniform Memory Access

- On **Non-Uniform Memory Access** (NUMA), systems with multiple CPU.

- A CPU can access its local memory faster than local memory of another CPU.
- NUMA systems are slower than systems in which all accesses to main memory are treated equally.



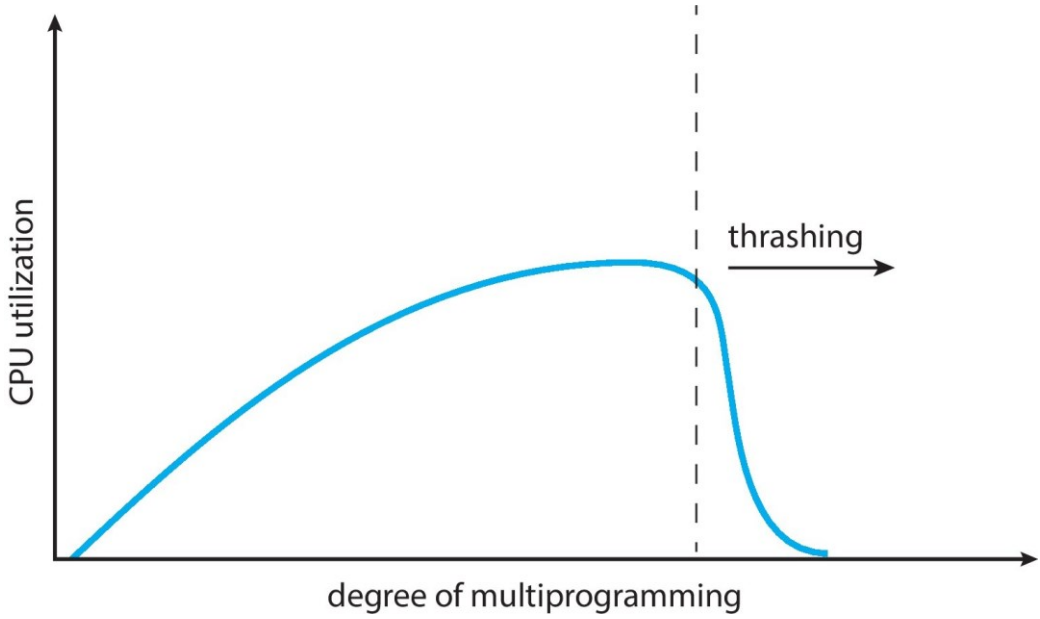
Trashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace an active page
  - But quickly need replaced page back >> quickly faults again.
- A process is **trashing** if it is spending more time paging than executing.
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

Trashing

- Operating system monitors CPU utilization.
- Low CPU utilization low, increasing the level of multiprogramming by a new process.
- It starts faulting and taking frames from other processes while other processes need those frames.
- These faulting must use the paging device to swap pages in and out.

- The CPU scheduler sees a decrease in CPU utilization, increasing the level of multiprogramming even more.
- At this point, trashing is happening, and we must decrease the degree of multiprogramming to increase CPU utilization.



Demand Paging and Thrashing

- Limiting the effect of trashing by using **Local Replacement Algorithm.**
  - It requires that each process select from only its own set of allocated frames.
  - If one process starts trashing, it cannot steal frames from another process and cause the other to thrash as well.
- It will keep the process in the queue for a long time, increasing the access time even for a process that is not trashing.
- To prevent trashing, providing a process with as many frames as it needs
  - **Locality model**

Demand Paging and Thrashing

- **Locality model:** The locality is a set of pages that are actively used together.
  - A running program is generally composed of several different localities.

- If we allocate enough frames to a process to accommodate its current locality, it will fault for the pages in its locality until all these pages are in memory.
- Then, it will not page fault again until it changes localities.
- If we do not allocate enough frames to accommodate the size of the current locality, the process will trash.

Working-Set Model

- The **working set model** is based on an assumption of locality.
- It is using a parameter Δ to define the **working-set window**.
- The set of pages in the most recent Δ page references is **working set**.
- The working set is the approximation of the program's locality.

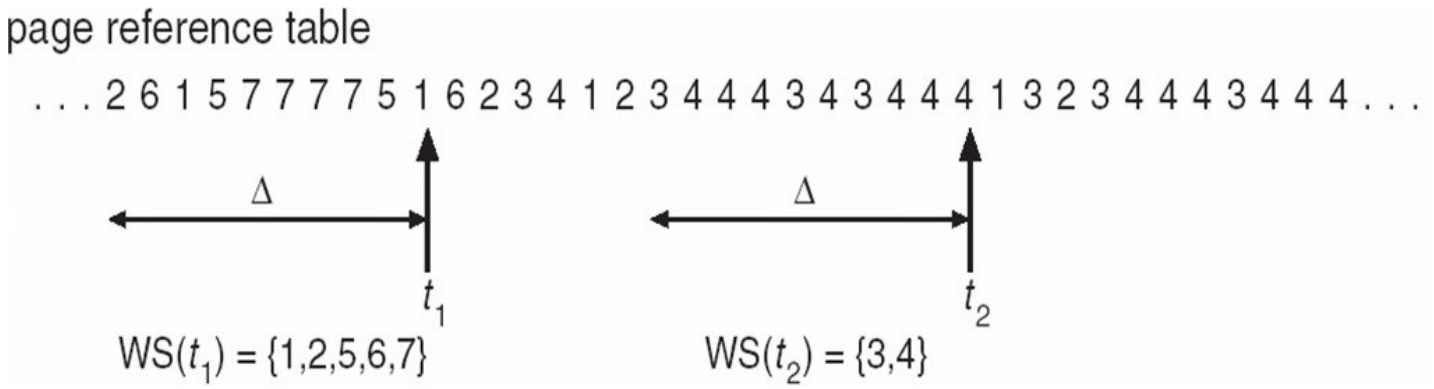
Working-Set Model

- The **working set model** is based on an assumption of locality.
- The accuracy of the working set depends on the selection of 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
- The most important property of the working set is its size.

- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality

Working-Set Model

- if *D* > m (the demand is greater than the total number of available frames) >> Thrashing
- Policy if *D* > m, then suspend or swap out one of the processes



Working-Set Model

- Once Δ is chosen, the OS allocates to the working sets, enough frames to provide it with its working-set size.
- If there are enough extra frames, another process can be initiated.
- If the sum of working-set sizes exceed the total number of available frames, the operating system selects a process to suspend and swap out the pages, reallocating the frames to other processes.
- This prevents thrashing while keeping the degree of multiprogramming as high as possible.
- **Challenge: How to keep track of working sets?**

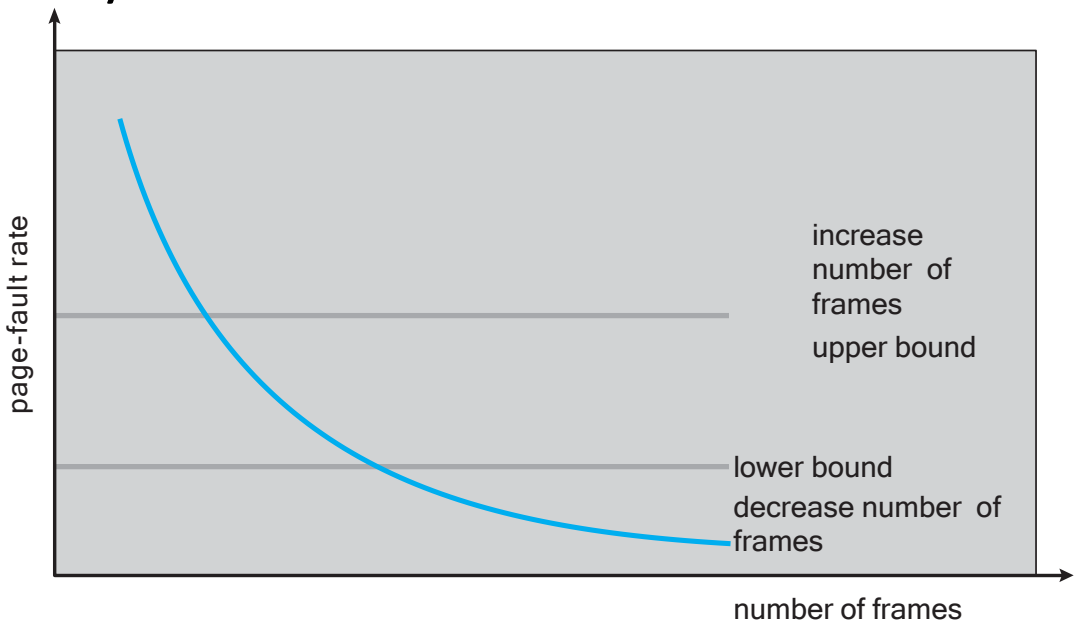
Keeping Track of the Working Set

- Approximation of the working-set model with a fixed interval timer interrupt and a reference bit
- Example: assuming Δ = 10,000 references
  - Timer interrupts after every 5,000 references
  - When we get a timer interrupt, we copy and clear the reference-bit values for each page.
  - Thus, if a page fault occurs, we can examine the current reference bit and two in-memory bits to determine whether a page was used within the last 10,000 to 15,000 references.
- Why is this not completely accurate?
  - We cannot tell where, within an interval of 5,000, a reference occurred.
  - Reducing the uncertainty by increasing the number of history bits and the frequency of interrupts.

Page-Fault Frequency

- More direct approach than WSS by controlling page fault rate.
- Establish "acceptable" **page-fault frequency (PFF)** rate and use local replacement policy
  - When PFF is too high, the process needs more frames.
  - When PFF is too low, the process may have too many frames.

Establishing upper and lower bounds on the desired page-fault rate.



Allocating Kernel Memory

- When a process running in user mode requests additional memory, pages are allocated from the list of free page frames maintained by the kernel.

- Kernel memory is often allocated form a free-memory pool different from the list used to satisfy ordinary user-mode processes.

Allocating Kernel Memory

- Kernel Memory is often allocated from a free-memory pool for two reasons:
  - The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. So, the kernel must minimize the waste due to fragmentation. This is important because many operating systems do not subject kernel code or data to the paging system.
  - Some kernel memory needs to be contiguous. Certain HW interacts directly with physical memory, without the benefits of a virtual memory interface.

Allocating Kernel Memory

- Kernel Memory is often allocated from a free-memory pool for two reasons:
  - The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. So, the kernel must minimize the waste due to fragmentation. This is important because many operating systems do not subject kernel code or data to the paging system.
  - Some kernel memory needs to be contiguous. Certain HW interacts directly with physical memory, without the benefits of a virtual memory interface.

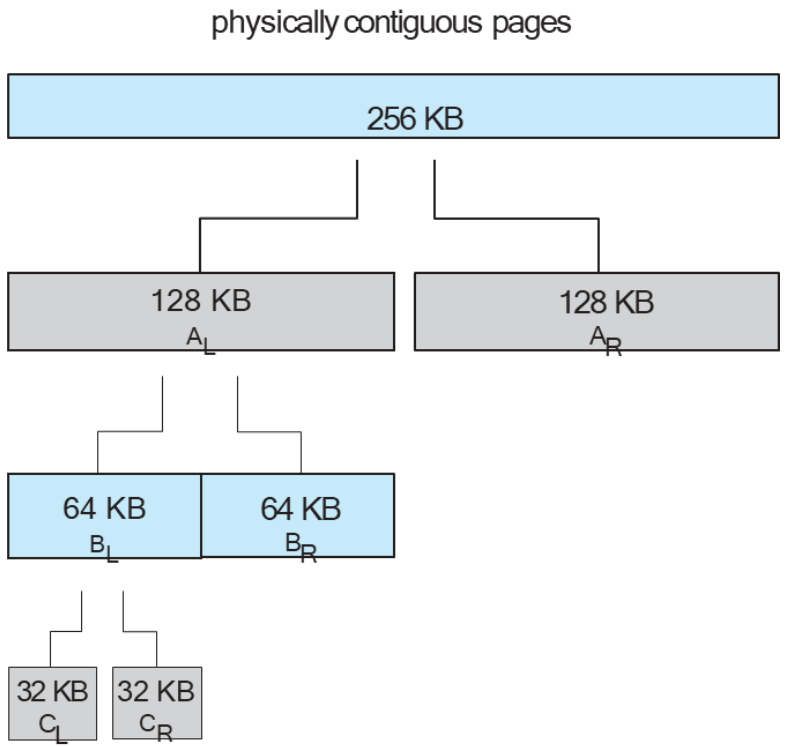
- Two strategies for allocating Kernel Memory:
  - **Buddy System**
  - **Slab Allocation**

Buddy System

- Allocates memory from fixed-size segment consisting of physically contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until an appropriately sized chunk available

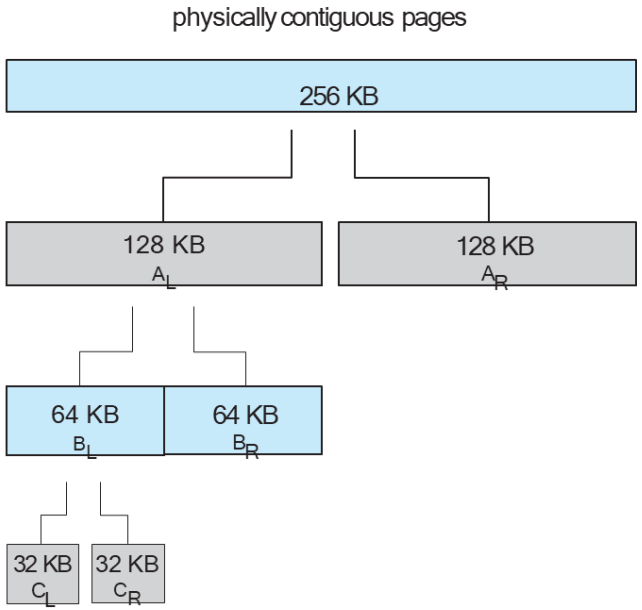
Buddy System

- For example, assuming the size of a memory segment is 256KB, kernel requests 21KB
  - The segment is initially divided into two **buddies** >> A_L and A_R, each 128 KB in size
  - One of these buddies is further divided into two 64KB buddies >> B_L and B_R
  - Either BL or BR is divided again into two 32 KB buddies >> C_L and C_R
    - One of these buddies is used to satisfy the 21KB request.



Buddy System

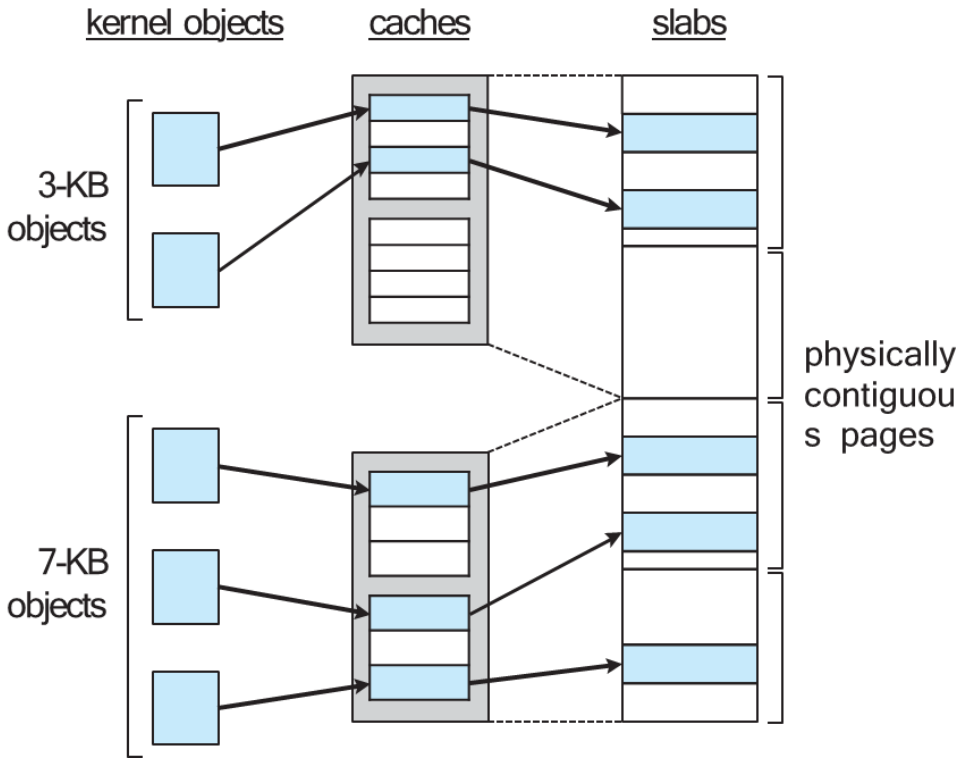
- An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as **Coalescing**.
  - In the previous example, when the kernel release C_L unit, the system can coalesce C_L and C_R into a 64 KB segment.
  - This segment, B_L, can be coalesced with its buddy, B_R, to form a 128 KB segment.
  - Ultimately, we can end up with the original 256 KB segment.



Slab Allocation

- Another strategy for allocating kernel memory is **slab allocation**.
  - **Slab** is made up of one or more physically contiguous pages.
  - **Cache** consists of one or more slabs.

- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the kernel data structure the cache represents.



Slab Allocation

- When aa cache is created, a number of objects, which are initially marked as **free**, are allocated to the cache.
- When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request.
  - The object assigned from the cache is marked as **used**.

- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated from contiguous physical pages and assign to cache.
- Benefits include no fragmentation, fast memory request satisfaction

Exercise (25/06/2018):

Si consideri la seguente sequenza di riferimenti a pagine in memoria: **57751711434123**.

Si utilizzi un algoritmo di sostituzione pagine di tipo working-set (versione esatta) con finestra **Δ = 3**, assumendo che siano disponibili al **massimo 3 frame**. Determinare **quali e quanti page fault** (accessi a pagine non presenti nel resident set) e **page out** (rimozioni di pagine dal resident set) si verificheranno. Si richiede la visualizzazione (dopo ogni accesso) del resident set.

Riferimenti	5	7	7	5	1	7	1	1	1	4	3	4	1	2	3
Resident Set	5	5	5	5	5	5				4	4	4	4	4	3
		7	7	7	7	7	7	7		3	3	3	3	2	2
					1	1	1	1	1	1	1		1	1	1
Page Fault	X	X			X					X	X		X	x	x
Page Out							X		X			X		X	X

Total number of Page faults = 8

Total number of Page Out = 5

Exercise (25/06/2018):

Si vuole inoltre definire una misura di località del programma svolto, basato sulla "Reuse Distance". La Reuse Distance al tempo Ti (RDi), in cui si accede alla pagina Pi, viene definita come il numero di pagine (distinte e diverse da Pi) a cui si è fatto accesso a partire dal precedente accesso a Pi (assumendo, convenzionalmente, per il primo accesso a una pagina, il numero totale di pagine cui si è fatto accesso sino a quell'istante). Ad esempio, al tempo 3, in cui si fa il secondo accesso alla pagina 5, RD3 = 1, in quanto tra i due accessi alla pagina 5 si è fatto accesso, due volte, a una sola pagina (7). Dati i vari RDi, se ne calcoli il valor medio RDavg. La località del programma svolto viene definita come L = 1 / (1+ RDavg). Si calcolino i valori RDi, RDavg e L.

Riferimenti	5	7	7	5	1	7	1	1	1	4	3	4	1	2	3
Resident Set	5	5	5	5	5	5				4	4	4	4	4	3
		7	7	7	7	7	7	7		3	3	3	3	2	2
					1	1	1	1	1	1	1		1	1	1
Page Fault	X	X			X					X	X		X	x	x
Page Out							X		X			X		X	X
RD	0	1	0	1	2	2	1	0	0	3	4	1	2	5	3

$RD_{ave} = 25/15 = 1.67$

Exercise (24/07/2019):

Sia data la stringa di riferimenti a pagine 3, 4, 1, (3, 1, 4, 4, 3, 1, 1)*10, in cui la sintassi (…)*n indica che la stringa tra parentesi viene ripetuta/iterata n volte (la stringa può ad esempio, derivare da un costrutto iterativo).

Si utilizzi un algoritmo di sostituzione pagine di tipo working-set (versione esatta) con finestra di durata $\Delta = 3$. Si supponga di denominare page-out la rimozione di una pagina dal resident set (in quanto esce dal working set). Si visualizzino nello schema che segue i riferimenti e il resident set dopo ogni riferimento, indicando i page-fault (accessi a pagine non presenti nel resident set) e i page-out. Si richiede la visualizzazione solo fino alle prime due iterazioni della sotto-stringa ripetuta 10 volte. ATTENZIONE: ogni riga del resident set rappresenta un frame, quindi una pagina presente in un frame non può cambiare riga quando rimane nel resident set. Nel caso di page-fault senza page-out, si utilizzi il primo frame libero dall'alto. Nel caso di page-out e (contemporaneo) page-fault, viene riutilizzato il frame appena liberato (da page-out). Qualora page-out e page-fault siano relativi alla stessa pagina, l'algoritmo di sostituzione ne tiene conto, evitando sia page-out che page-fault.

Exercise (24/07/2019):

Sia data la stringa di riferimenti a pagine **3, 4, 1, (3, 1, 4, 4, 3, 1, 1)*10**, in cui la sintassi (…)*n indica che la stringa tra parentesi viene ripetuta/iterata n volte. Si utilizzi un algoritmo di sostituzione pagine di tipo working-set (versione esatta) con **finestra di durata Δ = 3**.

Tempo	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Riferimenti	3	4	1	3	1	4	4	3	1	1	3	1	4	4	3	1	1
Resident Set	3	3	3	3	3	3		3	3	3	3	3	3		3	3	3
		4	4	4		4	4	4	4				4	4	4	4	
			1	1	1	1	1		1	1	1	1	1	1		1	1
Page Fault	X	X	X			X		X	X				X		X	X	
Page Out					X		X	X		X				X	X		X

Exercise (05/07/2019):

A) Si consideri la seguente sequenza di riferimenti in memoria nel caso di un programma in cui, per ogni accesso (indirizzi in esadecimale, si indirizza il Byte), si indica se si tratta di lettura (R) o scrittura (W): **R 3F5, R 364, W 4D3, W 47E, R 4C8, W 2D1, R 465, W 2A0, R 3BA, W 4E6, R 480, R 294, R 0B8, R 14E**.

Supponendo che sia indirizzi fisici che logici siano su **12 bit**, che si usi paginazione con pagine di dimensione **128 Byte** e che il massimo indirizzo utilizzabile dal programma sia **C10**, si dica quante pagine sono presenti nello spazio di indirizzamento del programma e se ne calcoli la frammentazione interna.

Exercise (05/07/2019):

B) Si determini la stringa dei riferimenti a pagine (Si consiglia di passare da esadecimale a binario, per determinare correttamente il numero di pagina e, se necessario, il displacement/offset). Si utilizzi un algoritmo di sostituzione pagine di tipo **LRU** (Least Recently Used). Si assuma che siano disponibili 3 frame, agli indirizzi fisici (espressi in esadecimale) **780, A00, B00**. Si richiede la visualizzazione (dopo ogni accesso) del resident set (i frame fisici contenenti pagine logiche).

Determinare quali e quanti page fault (accessi a pagine non presenti nel resident set) si verificheranno. Si dica infine a quali indirizzi fisici vengono effettuati gli accessi (tra quelli sopra elencati) R 3F5, W 4D3, R 3BA

Riferimenti		7	6	9	8	9	5	8	5	7	9	9	5	1	2
Resident Set	780	7	7	7	8		8			8	9			9	2
	A00		6	6	6		5			5	5			5	5
	B00			9	9		9			7	7			1	1
Page Fault		X	X	X	X		X			X	X			X	X

Exercise (011/09/2018):

A) Si consideri la seguente sequenza di riferimenti in memoria nel caso di un programma di 4K parole in cui, per ogni accesso (indirizzi in esadecimale), si indica se si tratta di lettura (R) o scrittura (W): W 3A1, R 3F5, R A64, W BD3, W 57E, R A08, R B85, W 3A0, R A1A, W A36, R B20, R 734, R AB8, R C4E, W B64.

Si determini la stringa dei riferimenti a pagine, supponendo che la loro dimensione sia di 512 parole. Si utilizzi un algoritmo di sostituzione pagine di tipo Enhanced Second-Chance, per il quale, al bit di riferimento (da inizializzare a 0 in corrispondenza al primo accesso a una nuova pagina dopo il relativo page fault), si unisce il bit di modifica (modify bit). Si assuma che una pagina venga sempre modificata in corrispondenza a una scrittura (write), che siano disponibili 3 frame e che l'algoritmo operi con il criterio seguente: dato il puntatore alla pagina corrente (secondo la strategia FIFO) si fa un primo giro, senza modificare il reference bit, sulle pagine per localizzare la vittima (l'ordine di priorità è (reference,modify):

(0,0), (0,1), (1,0), (1,1)); una volta determinata la vittima, si fa un secondo giro per azzerare i reference bit delle pagine "salvate" (comprese tra la posizione di partenza e la vittima).

Determinare quali e quanti page fault (accessi a pagine non presenti nel resident set) si verificheranno. Si richiede la visualizzazione (dopo ogni accesso) del resident set, indicando per ogni frame i bit di riferimento e modifica. Si numerino le pagine a partire da 0.

Exercise (011/09/2018):

A) zzare, per questa domanda, lo schema seguente per svolgere l'esercizio, indicando nella prima riga la stringa dei riferimenti a pagine (rappresentate a scelta in esadecimale o decimale), nella seconda Read o Write, nelle tre successive (che rappresentano i 3 frame del resident set), le pagine allocate nei corrispondenti frame, indicando per ognuna i bit (reference,modify). Indicare inoltre (sottolineandola, circolettandola o ponendo una freccia), quale pagina si trova in testa al FIFO. Nell'ultima riga si indichi la presenza o meno di un Page Fault.

Riferimenti	1	1	5	5	2	5	5	1	5	5	5	3	5	6	5
Read/Write	W	R	R	W	W	R	R	W	R	W	R	R	R	R	W
Resident Set	1_{01}	1_{11}	1_{11}	1_{11}	1_{11}	1_{11}	1_{11}	1_{11}	1_{11}	1_{11}	1_{11}	1_{01}	1_{01}	1_{01}	1_{01}
			5_{00}	5_{11}	5_{11}	5_{11}	5_{11}	5_{11}	5_{11}	5_{11}	5_{11}	5_{01}	5_{11}	5_{01}	5_{11}
					2_{01}	2_{01}	2_{01}	2_{01}	2_{01}	2_{01}	2_{01}	3_{00}	3_{00}	6_{00}	6_{00}
Page Fault	X		X									X		X	

Other Considerations

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking

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 α of the pages is used
  - Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging
- $s * (1 - \alpha)$ unnecessary pages*?*
  - α near zero prepaging loses

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**
  - 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

TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (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

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

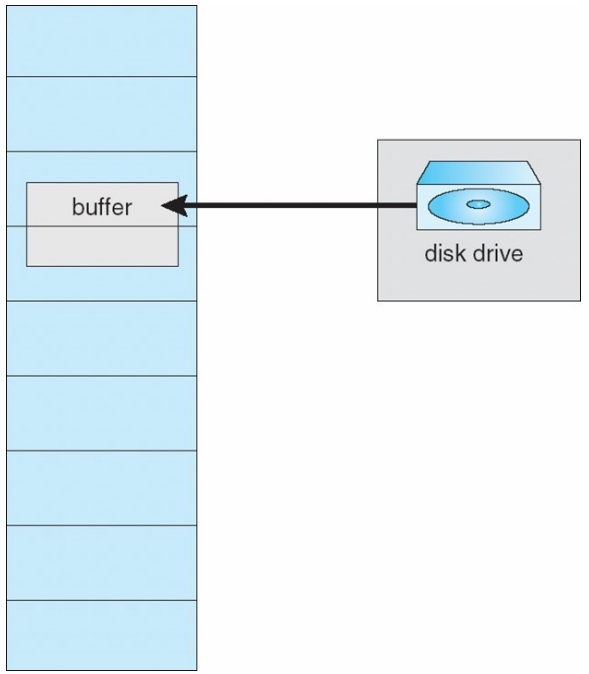
  - Program 2

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

- 128 page faults

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



Operating System Examples

- Windows
- Solaris

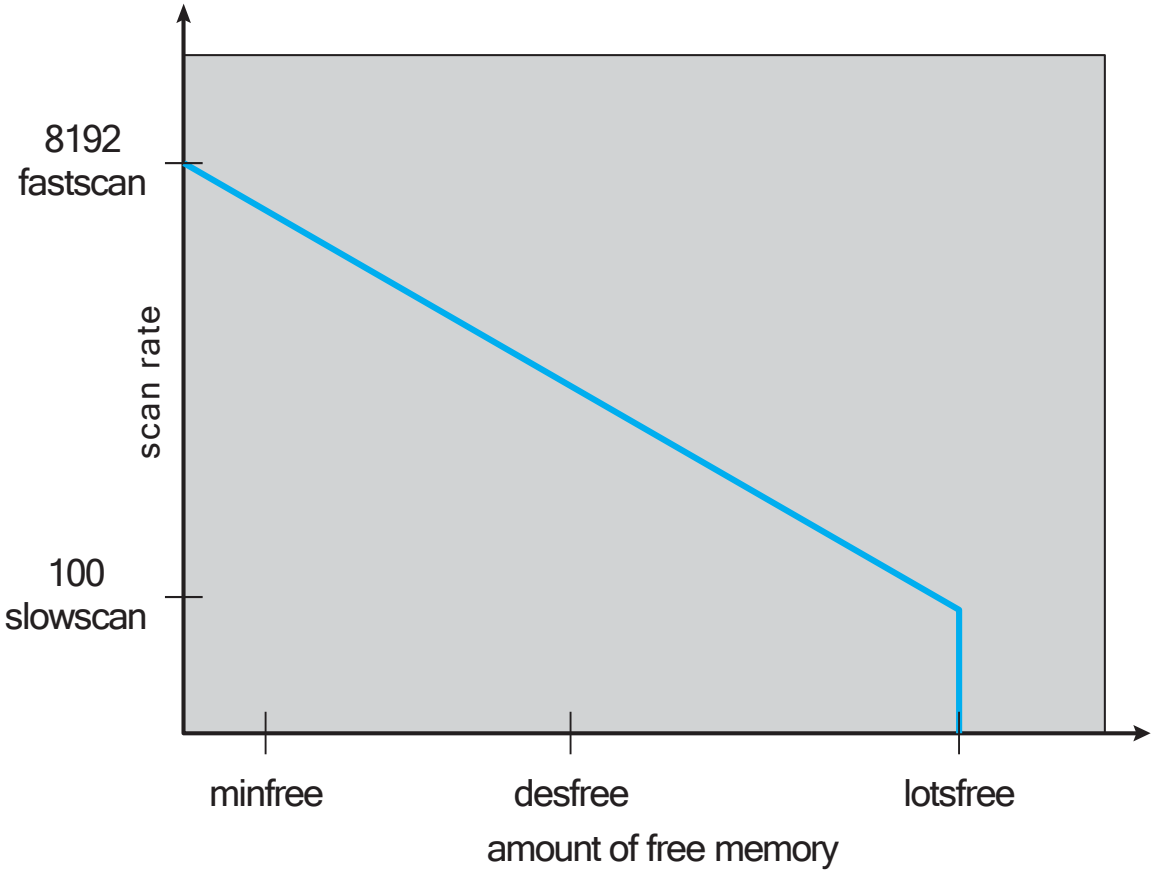
Windows

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

Solaris

- Maintains a list of free pages to assign faulting processes
- **`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. This ranges from **`slowscan`** to **`fastscan`**
- **`Pageout`** is called more frequently depending upon the amount of free memory available
- **Priority paging** gives priority to process code pages

Solaris 2 Page Scanner



Thanks

Politecnico di Torino

Department of Control and Computer Engineering

Questions?

sarah.azimi@polito.it