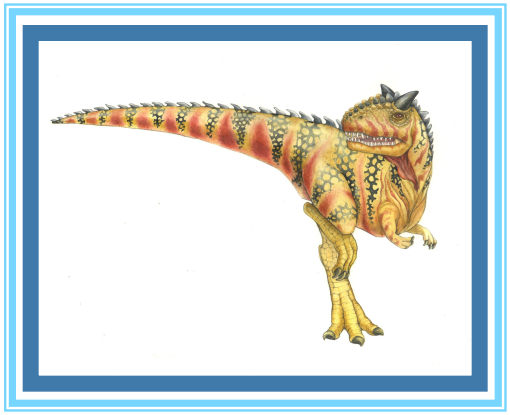
Chapter 10: Virtual Memory



Chapter 10: Virtual Memory

- Background — Virtual memory is a technique that allows the execution of processes that may not be completely in memory. It separates the logical memory as perceived by users from the physical memory, enabling more efficient and flexible use of memory.
- Demand Paging — Demand paging is a method of loading pages into memory only when they are needed, rather than loading the entire program into memory at once. This reduces the amount of physical memory required and can improve system performance
- Copy-on-Write — Copy-on-Write (COW) is an optimization strategy used in virtual memory systems. When a process creates a copy of its address space (e.g., during a fork operation), the pages are not immediately duplicated. Instead, both processes share the same pages, and a copy is made only when one of the processes modifies a shared page.
- Page Replacement — When a page fault occurs and there is no free frame available, the operating system must select a page to remove from memory to make space for the new page. This process is known as page replacement. Various algorithms are used to decide which page to replace, such as FIFO, LRU, and Optimal.
- Allocation of Frames — This involves determining how many frames to allocate to each process and which frames to allocate. The allocation can be done using fixed or dynamic allocation strategies.
- Thrashing — Thrashing occurs when a system spends more time swapping pages in and out of memory than executing actual processes. This happens when there is insufficient memory, causing a high rate of page faults.
- Memory-Mapped Files — Memory-mapped files allow files to be accessed as if they were part of the virtual memory. This can improve file I/O performance and simplify file access by mapping a file's contents directly into the address space of a process.
- Allocating Kernel Memory — Kernel memory allocation is different from user memory allocation. The kernel often requires contiguous memory and has different allocation strategies, such as buddy systems and slab allocation.
- Other Considerations
- Operating-System Examples — This section likely provides examples of how different operating systems implement virtual memory. Examples might include Windows, Linux, and macOS, each with its own specific strategies and optimizations.

Objectives

Virtual Memory: A technique that allows the execution of processes that may not be completely in memory.
Benefits: It enables more efficient use of physical memory, allows larger programs to run on systems with limited physical memory, and provides isolation between processes.

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.

  Demand Paging: Pages are loaded into memory only when they are needed, reducing the amount of physical memory required and improving system performance.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.

  Apply the FIFO, optimal, and LRU page-replacement algorithms:
  FIFO (First-In-First-Out): Replaces the oldest page in memory.
  Optimal: Replaces the page that will not be used for the longest period of time in the future (theoretical).
  LRU (Least Recently Used): Replaces the page that has not been used for the longest period of time.
- Describe the working set of a process, and explain how it is related to program locality.

  Describe the working set of a process, and explain how it is related to program locality:
  Working Set: The set of pages that a process is currently using.
  Program Locality: The tendency of a program to access a relatively small set of pages repeatedly over a short period of time.
- Describe how Linux, Windows 10, and Solaris manage virtual memory.

  Describe how Linux, Windows 10, and Solaris manage virtual memory:
  Each operating system has its own strategies and optimizations for managing virtual memory, which will be covered in detail.
- Design a virtual memory manager simulation in the C programming language.

Background

This slide provides an overview of the fundamental concepts and benefits of virtual memory:

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures

Error code, unusual routines, large data structures: Parts of the program that are not frequently used do not need to be loaded into memory initially.

- Entire program code not needed at same time

Only the parts of the program that are currently being executed need to be in memory.

- 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

Program no longer constrained by limits of physical memory: Virtual memory allows programs to run even if they are larger than the available physical memory.

Each program takes less memory while running -> more programs run at the same time: This increases the system's multitasking capabilities.
Increased CPU utilization and throughput with no increase in response time or turnaround time: More efficient use of CPU resources without negatively impacting performance.

Reducing the need for I/O operations to load or swap pages improves the overall speed and efficiency of user programs.

Virtual memory

Virtual memory is a system that separates the logical memory (what the user sees) from the physical memory (the actual hardware). This allows the system to use more memory than is physically available by using disk space to simulate additional RAM.

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
    Not all parts of a program are used at the same time. For example, error handling code or certain functions might not be needed immediately. Virtual memory allows only the necessary parts to be loaded into RAM, saving space.
  - Logical address space can therefore be much larger than physical address space
    The logical address space (the addresses used by the program) can be much larger than the physical memory available. This means programs can be larger than the actual RAM, as parts of them can be stored on disk and loaded as needed.
  - Allows address spaces to be shared by several processes
    Multiple processes can share the same physical memory. For example, if two programs use the same library, that library can be loaded into memory once and shared, rather than being loaded separately for each program.
  - Allows for more efficient process creation
    When creating new processes, virtual memory can make this more efficient. For example, using techniques like Copy-on-Write, where the system only makes copies of memory pages when they are modified, rather than copying everything at once.
  - More programs running concurrently
    Since not all parts of all programs need to be in memory at the same time, more programs can run simultaneously. This improves the overall utilization of the system.
  - Less I/O needed to load or swap processes
    By only loading the necessary parts of programs into memory, the system reduces the amount of input/output operations needed to load or swap programs. This makes the system faster and more efficient.

Virtual memory (Cont.)

- **Virtual address space** – logical view of how process is stored in memory

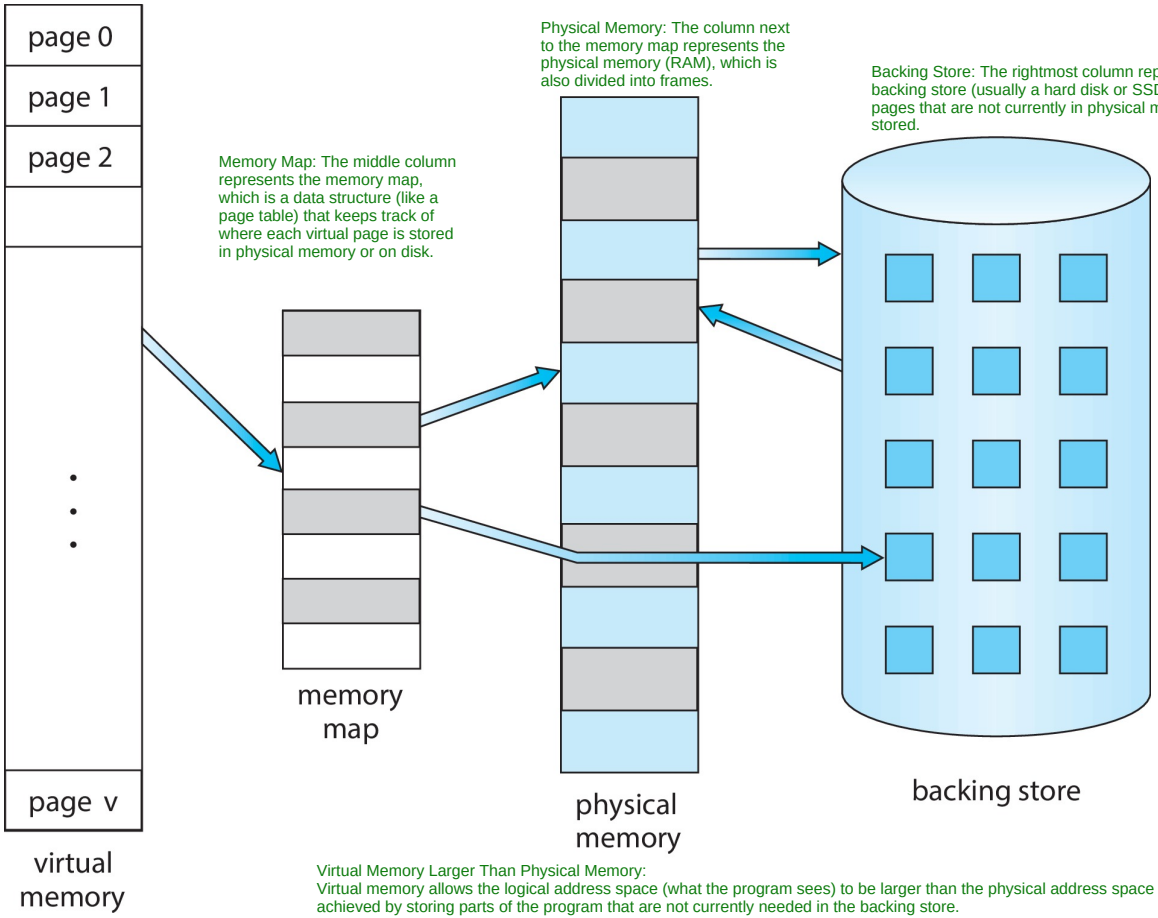
  The virtual address space is the logical view of how a process's memory is organized. It is the range of addresses that a process can use.

  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

1. Usually start at address 0, contiguous addresses until end of space:
The virtual address space typically starts at address 0 and continues in a contiguous manner. This makes it simple for the program to manage memory, as it sees a continuous block of addresses.
2. Meanwhile, physical memory organized in page frames:
Physical memory (the actual RAM) is divided into fixed-size blocks called page frames. The virtual memory system maps these logical addresses to physical addresses in these frames.
3. MMU must map logical to physical:
The Memory Management Unit (MMU) is a hardware component that handles the translation of logical addresses (used by the program) to physical addresses (used by the hardware). This mapping is crucial for the virtual memory system to work.

Virtual Memory Implementation:
1. Demand Paging:
Definition: A technique where pages are loaded into memory only when they are needed. If a program tries to access a page that is not in memory, a page fault occurs, and the operating system loads the required page from disk.
Benefit: This reduces the amount of memory needed and allows for more efficient use of physical memory.
2. Demand Segmentation:
Definition: Similar to demand paging, but instead of pages, segments (which can be of variable size) are loaded on demand. Segments can represent different logical parts of a program, such as code, data, and stack.
Benefit: This can provide more flexibility and efficiency, especially for programs that have distinct segments with different usage patterns.

Virtual Memory That is Larger Than Physical Memory



Virtual Memory: The leftmost column represents the virtual memory, which is divided into pages (page 0, page 1, page 2, ..., page v).

Memory Map: The middle column represents the memory map, which is a data structure (like a page table) that keeps track of where each virtual page is stored in physical memory or on disk.

Physical Memory: The column next to the memory map represents the physical memory (RAM), which is also divided into frames.

Backing Store: The rightmost column represents the backing store (usually a hard disk or SSD), where pages that are not currently in physical memory are stored.

Virtual Memory Larger Than Physical Memory:
Virtual memory allows the logical address space (what the program sees) to be larger than the physical address space (the actual RAM). This is achieved by storing parts of the program that are not currently needed in the backing store.
2. Page Mapping:
The memory map (page table) keeps track of where each virtual page is located. When a page is needed, it is either found in physical memory or loaded from the backing store into physical memory.
3. Efficient Use of Memory:
This system allows for efficient use of physical memory by only keeping the necessary pages in RAM and storing the rest on disk. This way, larger programs can run on systems with limited physical memory.

Virtual-address Space

The logical address space is designed to maximize address space use. The stack starts at the maximum logical address and grows downward, while the heap grows upward. This design leaves a large unused address space (hole) between the stack and heap, which can be used as needed.

- 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

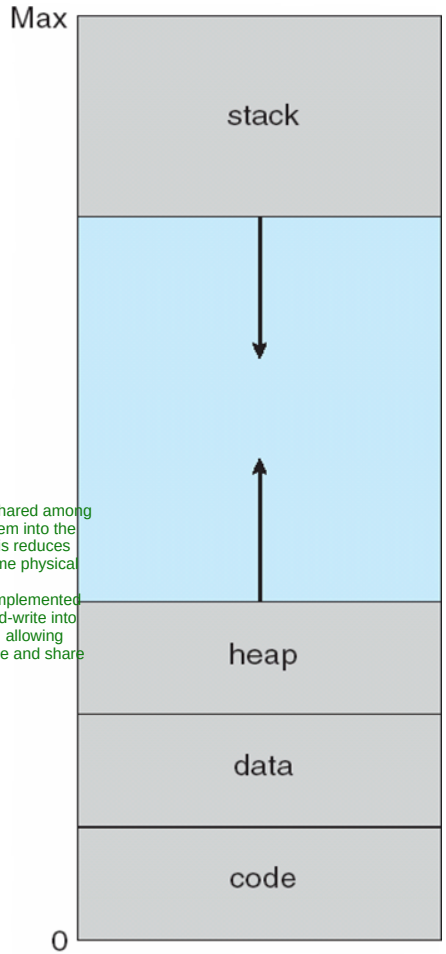
This design enables sparse address spaces, where there are gaps (holes) left for growth. This is useful for dynamically linked libraries, system libraries, and other dynamically allocated memory.

- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc

This design enables sparse address spaces, where there are gaps (holes) left for growth. This is useful for dynamically linked libraries, system libraries, and other dynamically allocated memory.
Sparse address spaces mean that not all of the logical address space is mapped to physical memory. Only the parts that are actually used are mapped, which saves physical memory.

- 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

. Max: This represents the maximum logical address in the virtual address space.



Stack: The stack is used for function calls, local variables, and control flow. It grows downward from the maximum logical addr towards lower addrs. For eg, if a function is called, its local variables and return address are pushed onto the stack

System libraries can be shared among processes by mapping them into the virtual address space. This reduces memory usage as the same physical memory can be shared.
Shared memory can be implemented by mapping pages as read-write into the virtual address space, allowing processes to communicate and share data efficiently.

The heap is used for dynamic mem allocation (e.g using malloc in C or new in C++/ JAVA). It grows upwards from the end of the data segment towards higher addresses

The data segment contains global and static variables that are initialized by the programmer. For eg, a global var defined outside of any func would be stored in the data segment.

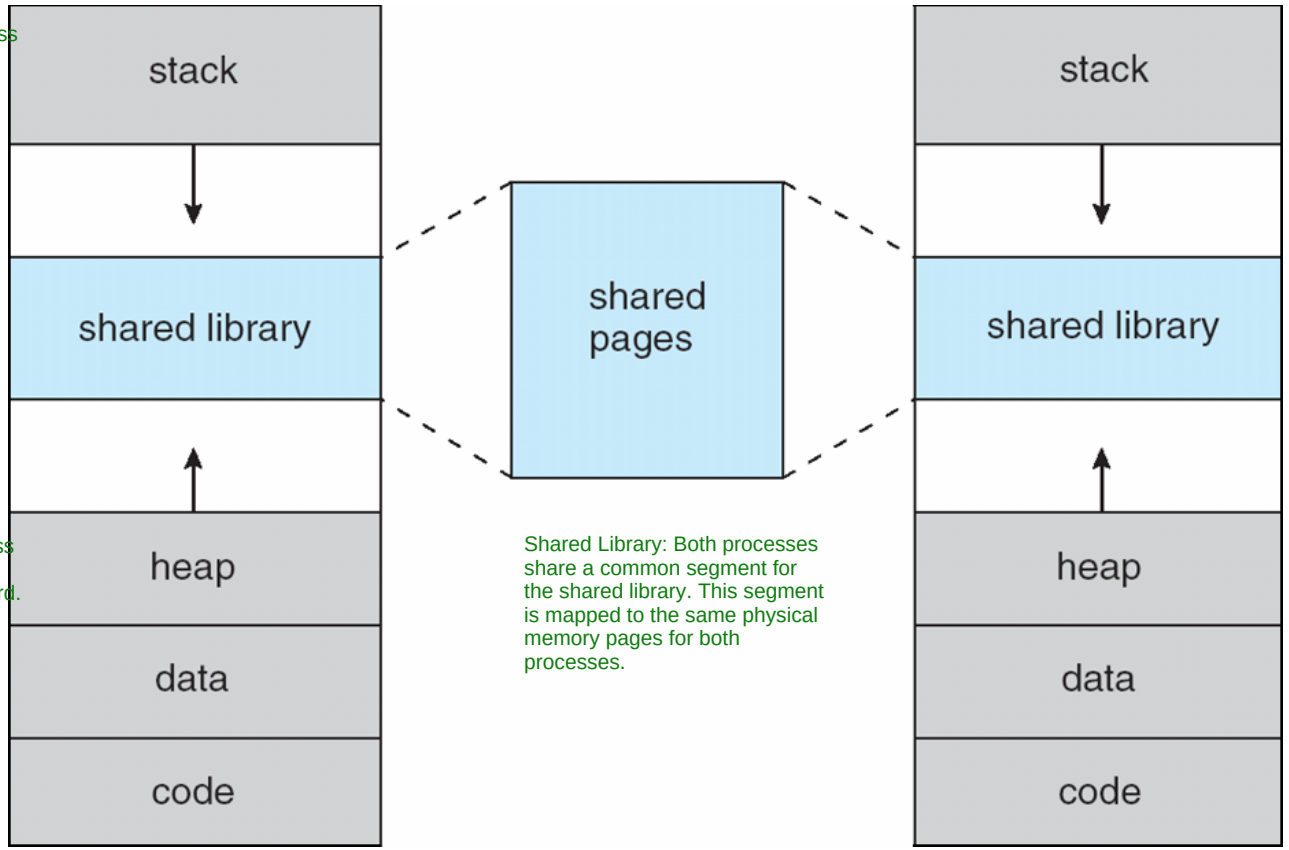
The code segment contains the executable instructions of the program. For example, the compiled machine code of the program's functions and methods is stored here.

System Libraries and Shared Memory:
System libraries can be shared among processes by mapping them into the virtual address space. This reduces memory usage as the same physical memory can be shared.
Shared memory can be implemented by mapping pages as read-write into the virtual address space, allowing processes to communicate and share data efficiently.

When a process is created using fork(), the pages can be shared between the parent and child processes. This speeds up process creation and reduces memory usage by sharing the same physical pages until they are modified (Copy-on-Write).

Shared Library Using Virtual Memory

The diagram shows two processes, each with its own virtual address space.



Stack: Each process has its own stack, which grows downward.

Heap: Each process has its own heap, which grows upward.

Data: Each process has its own data segment.

Code: Each process has its own code segment.

Shared Library: Both processes share a common segment for the shared library. This segment is mapped to the same physical memory pages for both processes.

1. Shared Library:
A shared library is a collection of code that can be used by multiple programs simultaneously. Examples include standard libraries like libc in C or msvcrt in Windows.
By mapping the shared library into the virtual address space of multiple processes, the same physical memory pages can be used by all processes. This reduces memory usage and improves efficiency.
2. Shared Pages:
The shared library segment in each process's virtual address space points to the same physical memory pages. This means that the code and data in the shared library are loaded into memory only once, but can be accessed by multiple processes.
3. Benefits:
Memory Efficiency: Reduces the overall memory footprint by sharing common code and data.
Faster Process Creation: When a new process is created, it can share the already loaded libraries, speeding up the process creation time.
Consistency: Ensures that all processes use the same version of the shared library, reducing potential inconsistencies.

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

Loading Pages on Demand:
Instead of loading the entire process into memory at once, demand paging loads pages only when they are needed. This reduces the amount of memory required and improves system performance.
Less I/O Needed: Only the necessary pages are loaded, reducing the number of I/O operations.
Less Memory Needed: Only the required pages are kept in memory, freeing up space for other processes.
Faster Response: The system can start executing a process without waiting for the entire process to be loaded into memory.
More Users: More processes can be kept in memory simultaneously, allowing more users to run programs concurrently.

- Similar to paging system with swapping (diagram on right)
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

When a process tries to access a page that is not in memory, a page fault occurs. The operating system then loads the required page from the backing store (disk) into memory.
Invalid Reference: If the process tries to access an invalid address, the operating system aborts the process.
Not-in-Memory: If the page is not in memory but is a valid address, the operating system brings the page into memory.

- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

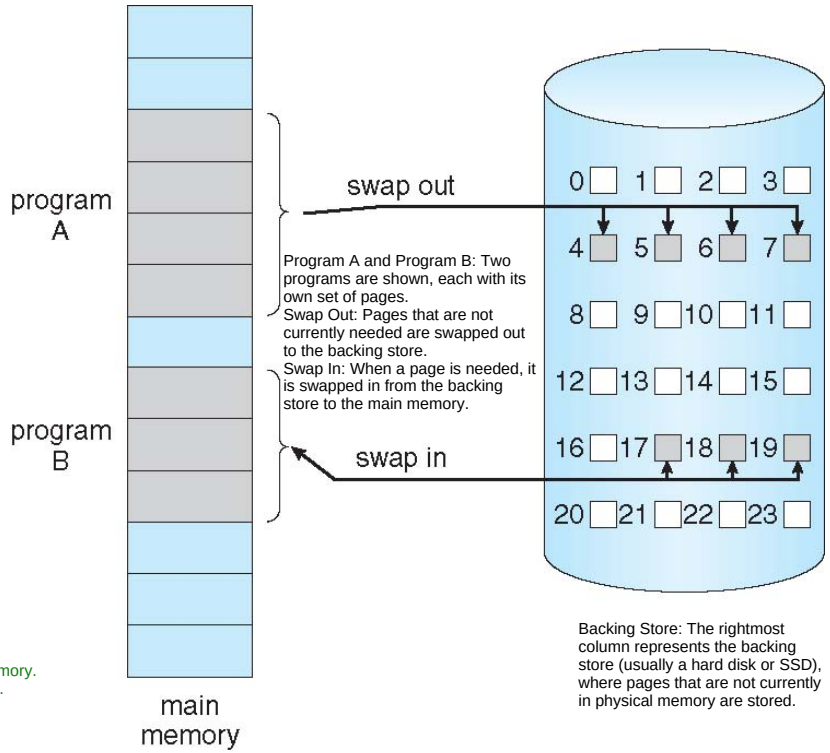
Lazy Swapper:
A lazy swapper (or pager) is a component that only loads pages into memory when they are needed. It never swaps a page into memory unless the page will be used.
Pager: The term "pager" refers to the component that handles the loading of pages into memory on demand.

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)

less I/O is needed because with demand paging, only the pages that are actually needed by the process are loaded into memory. This means that the system performs I/O operations (reading from disk) only for the required pages, rather than loading the entire process into memory at once. This reduces the total number of I/O operations, as unnecessary pages are not loaded, leading to more efficient use of I/O resources.

Entire Process at Load Time: One approach is to load the entire process into memory when it starts. This can be inefficient if the process does not use all of its pages immediately.
Page on Demand: A more efficient approach is to load pages into memory only when they are needed. This is known as demand paging.



Program A and Program B: Two programs are shown, each with its own set of pages.
Swap Out: Pages that are not currently needed are swapped out to the backing store.
Swap In: When a page is needed, it is swapped in from the backing store to the main memory.

Backing Store: The rightmost column represents the backing store (usually a hard disk or SSD), where pages that are not currently in physical memory are stored.

Benefits of Demand Paging:
Less I/O Needed: Only the necessary pages are loaded, reducing the number of I/O operations.
Less Memory Needed: Only the required pages are kept in memory, freeing up space for other processes.
Faster Response: The system can start executing a process without waiting for the entire process to be loaded into memory.
More Users: More processes can be kept in memory simultaneously, allowing more users to run programs concurrently.

3. Similar to Paging System with Swapping:
Demand paging is similar to a paging system that uses swapping. The diagram on the right illustrates this concept.

Main Memory: The middle column represents the main memory (RAM), which is divided into frames.

Basic Concepts

Swapping:
With traditional swapping, the pager guesses which pages will be used before swapping them out again. This can be inefficient if the guesses are incorrect.

- With swapping, pager guesses which pages will be used before swapping out again

Demand Paging:
Instead of guessing, demand paging brings in only those pages that are actually needed. This reduces unnecessary I/O and memory usage.

- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging

Determining the Set of Pages:
To implement demand paging, the system needs new functionality in the Memory Management Unit (MMU) to detect page faults and load the required pages.

- If pages needed are already **memory resident**
  - No difference from non demand-paging

If the pages needed by a process are already in memory (memory resident), there is no difference from non-demand paging. The process can access the pages directly.

- 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

Handling Page Faults:
If a page needed by a process is not in memory (not memory resident), the system needs to detect this and load the page into memory from storage.
This should be done without changing the program's behavior and without requiring the programmer to change the code.

Basic Concepts: Demand paging requires new MMU functionality to handle page faults and load pages on demand. It ensures that pages are loaded only when needed, reducing unnecessary I/O and memory usage.

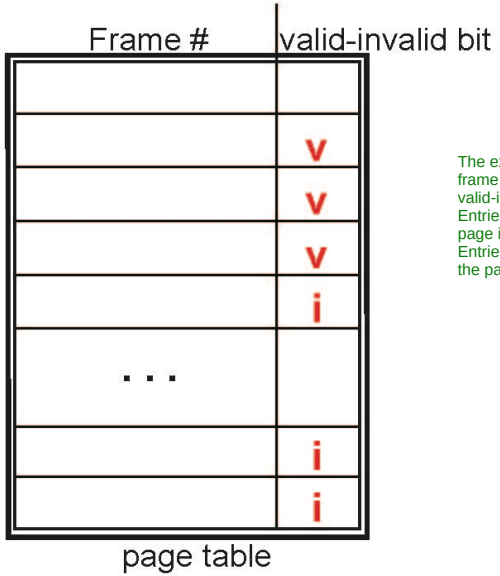
Valid-Invalid Bit

Valid-Invalid Bit:
Each entry in the page table has a valid-invalid bit associated with it.
Valid (v): Indicates that the page is in memory (memory resident).
Invalid (i): Indicates that the page is not in memory (not memory resident).

- 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

Initially, the valid-invalid bit is set to invalid (i) for all entries, meaning no pages are in memory at the start.

- Example of a page table snapshot:



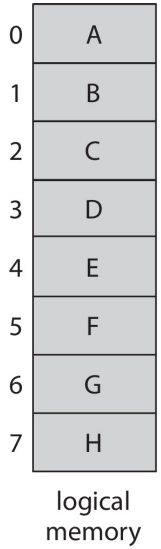
The example shows a page table with frame numbers and their corresponding valid-invalid bits.
Entries with a valid bit (v) indicate that the page is currently in memory.
Entries with an invalid bit (i) indicate that the page is not in memory.

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

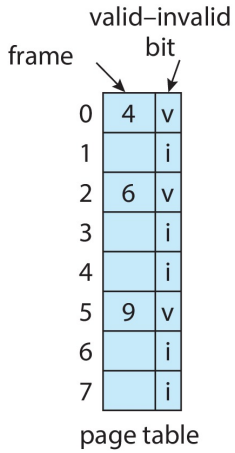
MMU Address Translation:
During address translation, if the MMU encounters an entry with an invalid bit (i), it triggers a page fault.
This means the page needs to be loaded from the backing store into memory.

Page Table When Some Pages Are Not in Main Memory

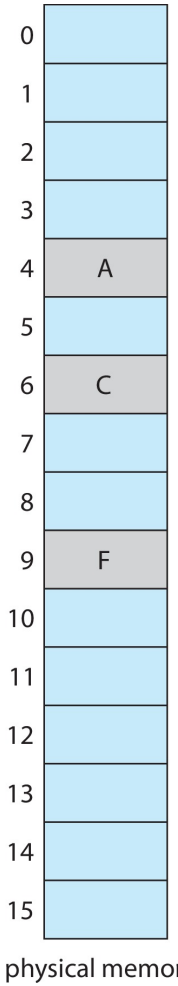
When the process modifies A the copy on disk is not the same as the copy on memory.



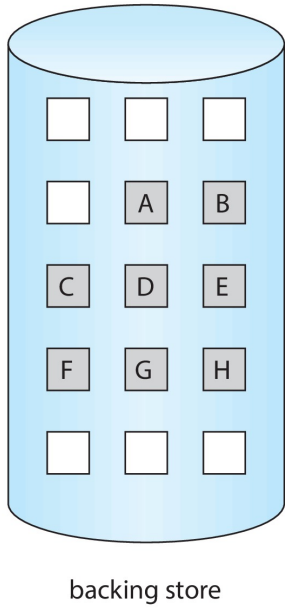
Logical Memory: The left column represents the logical memory, divided into pages (A, B, C, D, E, F, G, H).



The middle column shows the page table, which maps logical pages to physical frames.
Each entry in the page table has a frame number and a valid-invalid bit.
For example, page A is mapped to frame 4 and is valid(v), meaning it is in memory



Physical Memory: The next column represents the physical memory (RAM), divided into frames.
Frames 4, 6, and 9 are occupied by pages A, C, and F, respectively.



Backing Store: The rightmost column represents the backing store (disk), where pages that are not in memory are stored.
Pages B, D, E, G, and H are stored in the backing store.

Key Points:
1. Page Fault Handling:
When a process tries to access a page that is not in memory (invalid bit), a page fault occurs.
The operating system then loads the required page from the backing store into physical memory.
2. Example Scenario:
If the process tries to access page B, the MMU will find that the valid-invalid bit for page B is invalid (i).
This triggers a page fault, and the operating system will load page B from the backing store into an available frame in physical memory.
3. Consistency:
When a process modifies a page (e.g., page A), the copy in memory is updated. The copy on disk (backing store) may not be the same as the copy in memory until it is written back.

Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to operating system
   - Page fault

Page Fault Occurrence:
When a process references a page that is not in memory, the first reference to that page will trap to the operating system, causing a page fault.

2. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory

The operating system looks at another table (often part of the page table) to decide the nature of the fault:
Invalid Reference: If the reference is invalid (e.g., the address is outside the process's address space), the operating system aborts the process.
Not in Memory: If the page is valid but not currently in memory, the operating system proceeds to handle the page fault.

3. Find free frame

Find Free Frame:
The operating system finds a free frame in physical memory to load the required page.

4. Swap page into frame via scheduled disk operation

Swap Page into Frame:
The operating system schedules a disk operation to read the required page from the backing store (disk) and load it into the free frame in physical memory.

5. Reset tables to indicate page now in memory
   Set validation bit = **v**

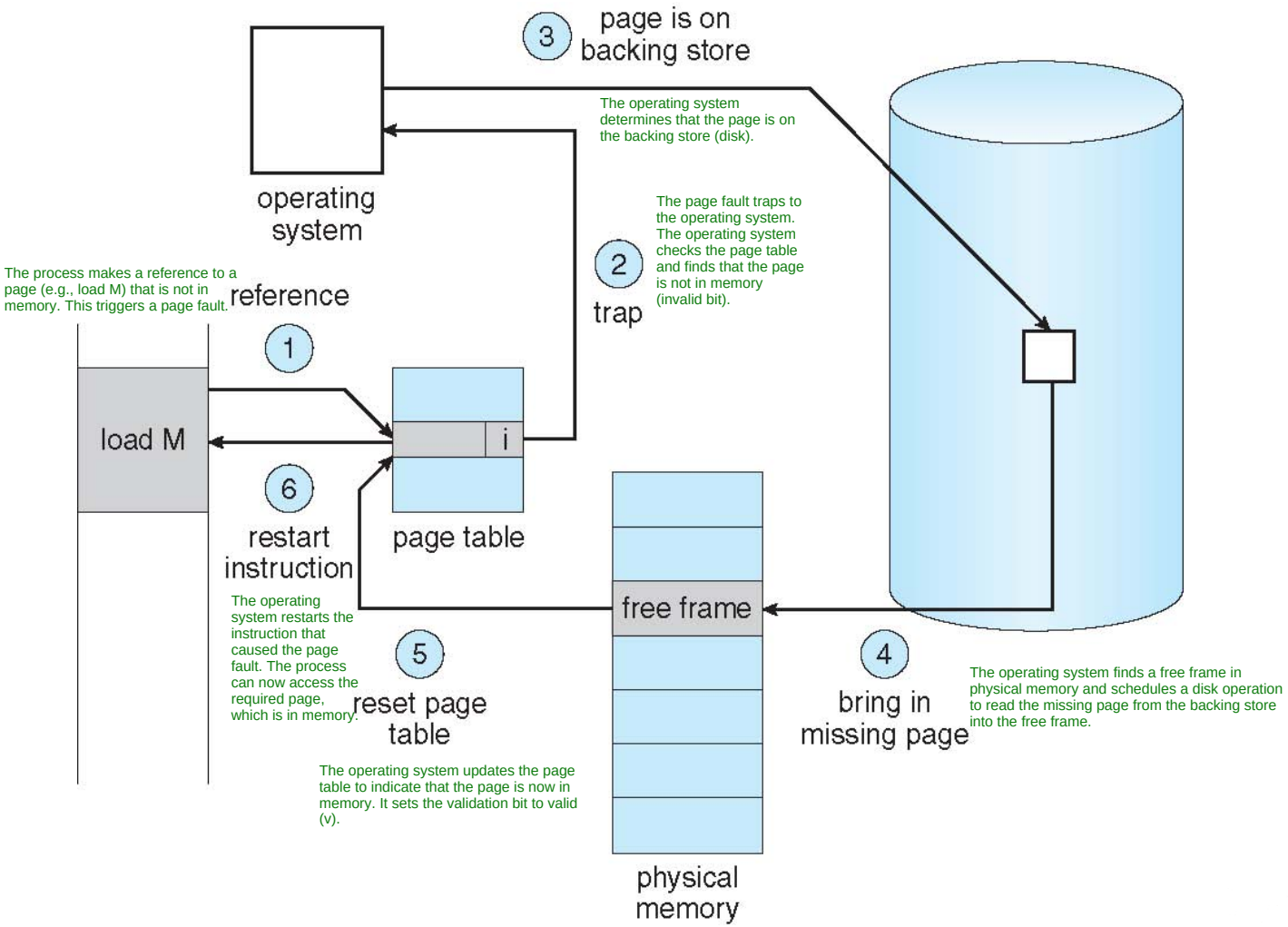
Update Page Table:
The operating system updates the page table to indicate that the page is now in memory. It sets the validation bit to valid (v).

6. Restart the instruction that caused the page fault

Restart Instruction:
The operating system restarts the instruction that caused the page fault. The process can now access the required page, which is in memory.

Steps in Handling a Page Fault (Cont.)

The diagram illustrates the steps involved 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**

Starting with No Pages in Memory:
Imagine a process starts running, but none of its pages are loaded into memory initially.
The operating system sets the instruction pointer to the first instruction of the process.
Since this instruction is not in memory, it causes a page fault.
This process continues for every page the process tries to access for the first time, causing a page fault each time.
Pure Demand Paging: This is called pure demand paging because pages are only loaded into memory when they are accessed for the first time.

- 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

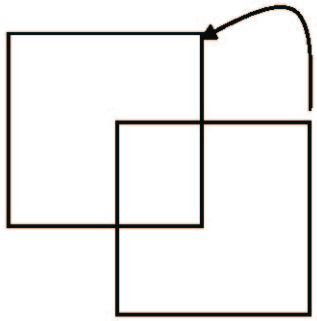
Multiple Accesses:
A single instruction might need to access multiple pages. For example, an instruction that adds two numbers from different memory locations and stores the result back to another location.
If none of these pages are in memory, each access will cause a separate page fault.
Example:
Consider an instruction that fetches two numbers from memory and stores the result back to memory.
This could cause three page faults if the pages containing the two numbers and the result are not in memory.
Locality of Reference:
The concept of locality of reference helps mitigate this issue. Programs tend to access a small set of pages repeatedly over a short period, so once the necessary pages are loaded, fewer page faults will occur.

. Hardware Support Needed for Demand Paging:
Page Table with Valid/Invalid Bit:
The page table must have valid/invalid bits to indicate whether a page is in memory (valid) or not (invalid).
Secondary Memory (Swap Space):
There must be secondary memory (swap space) to store pages that are not currently in memory.
Instruction Restart:
The hardware must support restarting instructions that caused a page fault. This means the system needs to be able to resume the execution of an instruction after the required page has been loaded into memory.

Instruction Restart

- Consider an instruction that could access several different locations
  - Block move

Complex Instructions:
Some instructions can access several different memory locations. For example, a block move instruction that copies a block of memory from one location to another.
Block Move: The diagram illustrates a block move operation where the source and destination blocks overlap.



  - Auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?

Instructions that automatically increment or decrement memory locations can also cause issues if they span multiple pages.

Restarting the Whole Operation:
When a page fault occurs, the operating system needs to restart the instruction that caused the fault.
Challenges: Restarting the whole operation can be challenging if the source and destination overlap or if the instruction accesses multiple pages.
The system must ensure that the instruction is restarted correctly and that the program's behavior remains consistent.

Free-Frame List

When a page fault occurs, the operating system must bring the desired page from secondary storage (disk) into main memory (RAM).

- 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.

Most operating systems maintain a free-frame list, which is a pool of free frames available for allocation when a page fault occurs. The free-frame list helps the operating system quickly find available memory to load the required page.

The diagram shows a linked list of free frames with the head pointing to frame 7, which points to frame 97, and so on, until frame 75. This list allows the operating system to efficiently manage and allocate free frames.

- 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.

The operating system typically allocates free frames using a technique known as zero-fill-on-demand. This means that the contents of the frames are zeroed out (set to zero) before being allocated to ensure that no residual data from previous processes remains.

- When a system starts up, all available memory is placed on the free-frame list.

When the system starts up, all available memory is placed on the free-frame list. This ensures that the operating system has a pool of free frames ready for allocation as needed.

Stages in Demand Paging – Worse Case

1. Trap to the operating system

   Trap to the Operating System:
   When a page fault occurs, the process traps to the operating system. This means that control is transferred from the user process to the operating system to handle the fault.

2. Save the user registers and process state

   Save User Registers and Process State:
   The operating system saves the current state of the user process, including the contents of the CPU registers and other relevant information. This is necessary to resume the process later.

3. Determine that the interrupt was a page fault

   The operating system determines that the interrupt was caused by a page fault. This involves checking the page table to see if the page is marked as invalid (not in memory).

4. Check that the page reference was legal and determine the location of the page on the disk

   The operating system checks that the page reference was legal (i.e., within the process's address space) and determines the location of the page on the disk.

5. Issue a read from the disk to a free frame:

   The operating system issues a read request to load the required page from the disk into a free frame in memory. This involves several steps:

   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

   1. Wait in Queue: The read request may need to wait in a queue if the disk is busy servicing other requests.
   2. Device Seek/Latency Time: The disk must seek to the correct location and wait for the appropriate latency time before reading the data.
   3. Transfer Page: The page is transferred from the disk to a free frame in memory.

Stages in Demand Paging (Cont.)

6. While waiting, allocate the CPU to some other user

While waiting for the disk I/O operation to complete, the operating system can allocate the CPU to another process. This ensures that the CPU is not idle and can continue executing other tasks.

7. Receive an interrupt from the disk I/O subsystem (I/O completed)

Once the disk I/O operation is completed, an interrupt is generated to notify the operating system that the requested page has been loaded into memory.

8. Save the registers and process state for the other user

The operating system saves the state of the currently running process (which was using the CPU while waiting for the I/O operation) so that it can be resumed later.

9. Determine that the interrupt was from the disk

The operating system determines that the interrupt was caused by the completion of the disk I/O operation.

10. Correct the page table and other tables to show page is now in memory

The operating system updates the page table and other relevant tables to indicate that the page is now in memory. The valid-invalid bit for the page is set to valid (v).

11. Wait for the CPU to be allocated to this process again

The process that caused the page fault must wait for the CPU to be allocated to it again. This involves scheduling the process to run once the CPU is available.

12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

The operating system restores the user registers, process state, and the updated page table for the process that caused the page fault. The instruction that caused the page fault is then resumed, and the process continues execution.

Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
    When a page fault occurs, the interrupt must be serviced. With careful coding, this can be done with just a few hundred instructions.
  - Read the page – lots of time
    Reading the page from disk takes a significant amount of time compared to memory access.
  - Restart the process – again just a small amount of time
    Once the page is loaded, the process must be restarted. This involves a small amount of time to restore the process state and resume execution.
- Page Fault Rate $0 \le p \le 1$
  The page fault rate (p) is the fraction of memory accesses that result in a page fault.
  - if $p = 0$ no page faults
    If p = 0, there are no page faults.
  - if $p = 1$, every reference is a fault
    If p = 1, every memory reference causes a page fault.
- Effective Access Time (EAT)
  The effective access time (EAT) is the average time required to access memory, considering both memory accesses and page faults.

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

(1−p)×memory access time: The time for memory accesses that do not cause page faults.

p×(page fault overhead+swap page out time+swap page in time): The time for memory accesses that cause page faults, including the overhead of handling the page fault and the time to swap pages in and out of memory.

Demand Paging Example

- Memory access time = 200 nanoseconds

  . Memory Access Time:
  The time it takes to access memory is given as 200 nanoseconds.

- Average page-fault service time = 8 milliseconds

  The time it takes to handle a page fault, including reading the page from disk and updating the page table, is given as 8 milliseconds (8,000,000 nanoseconds).

- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p x 200 + p x 8,000,000 (in nanoseconds)

  = 200 + p x 7,999,800

- If one access out of 1,000 causes a page fault, then

  If one access out of 1,000 causes a page fault (p = 0.001) (remember p is the rate of page faults).

  EAT=200+0.001×7,999,800=200+7,999.8 = 8,199.8 nanoseconds = 8.2 microseconds

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!

  This is a slowdown by a factor of 40 compared to the original memory access time of 200 nanoseconds.

- If want performance degradation < 10 percent

  To keep performance degradation below 10%, we set up the inequality:

  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025

    This means there should be less than one page fault in every 400,000 memory accesses to keep the performance degradation below 10%.
  - < one page fault in every 400,000 memory accesses

Demand Paging Optimizations

Swap space I/O is faster than file system I/O, even if they are on the same device. This is because swap space is allocated in larger chunks and requires less management than the file system.

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix

In older systems like BSD Unix, the entire process image is copied to swap space at process load time. Pages are then paged in and out of swap space as needed.

- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system

In systems like Solaris and current BSD, pages are demand-paged directly from the program binary on disk. When a frame is freed, the page is discarded rather than being paged out.
This approach still requires writing to swap space for pages not associated with a file (like stack and heap), known as anonymous memory.
Pages modified in memory but not yet written back to the file system also need to be written to swap space.

- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

Mobile systems typically do not support swapping due to limited storage and power constraints.
Instead, they demand page from the file system and reclaim read-only pages (such as code) when needed.

Copy-on-Write

Definition: Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory. This is particularly useful when a process creates a child process using fork().
Modification: If either the parent or child process modifies a shared page, only then is the page copied. This means that the actual copying of pages is deferred until it is necessary.

- **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

  COW allows for more efficient process creation because only the pages that are modified are copied. This reduces the overhead of creating a new process.
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages

  Zero-Fill-on-Demand:
  Free pages are generally allocated from a pool of zero-fill-on-demand pages. This means that the content of the frames is zeroed out before being allocated to ensure no residual data from previous processes remains.
  The pool should always have free frames available for fast demand page execution to avoid the need to free a frame during a page fault.
  - 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
  - Designed to have child call `exec()`
  - Very efficient

  vfork():
  vfork() is a variation of the fork() system call. It has the parent process suspend execution while the child process uses the copy-on-write address space of the parent.
  This is designed to have the child process call exec() soon after creation, making it very efficient.

Before Process 1 Modifies Page C

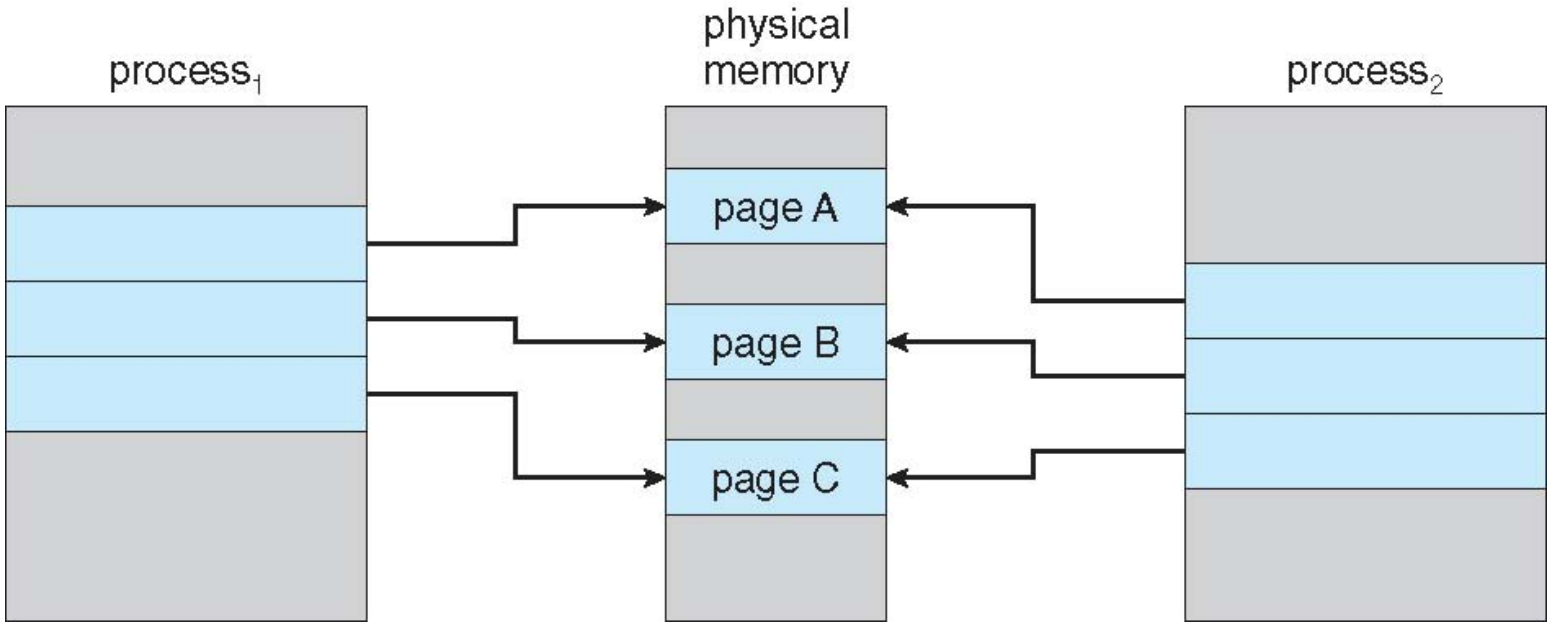


Diagram Explanation:
The diagram shows two processes, process1 and process2, sharing pages in physical memory before any modifications are made.

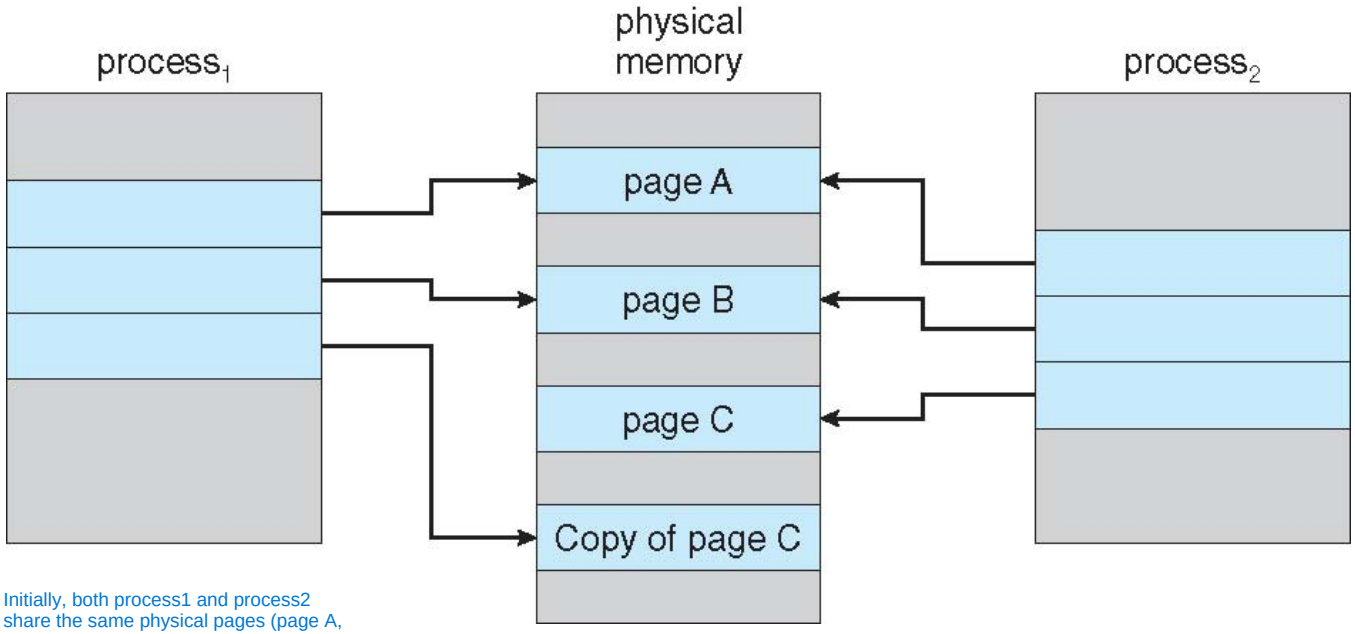
Shared Pages:
Both process1 and process2 share the same physical pages (page A, page B, and page C) in memory.
The arrows indicate that both processes have pointers to the same physical pages.

Initial State:
In the initial state, both processes can read from the shared pages without any issues. No copying of pages has occurred yet.

If process1 or process2 attempts to modify any of the shared pages (e.g., page C), the operating system will create a copy of the page for the modifying process. This ensures that the other process continues to see the original, unmodified page.

After Process 1 Modifies Page C

The diagram shows the state of memory after process1 modifies page C.



Initially, both process1 and process2 share the same physical pages (page A, page B, and page C) in memory.

When process1 modifies page C, the operating system creates a copy of page C for process1. This is the essence of Copy-on-Write (COW). The original page C remains unchanged and is still shared by process2.

New State:
process1 now has its own copy of page C, while process2 continues to use the original page C.
The physical memory now contains page A, page B, the original page C, and the copy of 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

Memory Usage:
Free frames can be used up by process pages, kernel, I/O buffers, etc.
The challenge is to determine how much memory to allocate to each.

Page Replacement:
When there are no free frames available, the operating system must use a page replacement algorithm to free up space.
Page Replacement Algorithm: The algorithm must find a page in memory that is not currently in use and page it out (write it to disk) to free up a frame.
Decisions: The algorithm must decide whether to terminate a process, swap out a page, or replace a page.

Performance Considerations:
The goal is to use an algorithm that minimizes the number of page faults, as frequent page faults can significantly degrade performance.
Repeated Page Faults: The same page may be brought into memory multiple times if the algorithm is not efficient, leading to repeated page faults.

Here we coming to the point of saying if there is no free frames what does it mean? free fram overall or ina process?
Lets just consider a given process has a certain number of valuable frames and all of them are used. How many do we allocate to each process?

Page Replacement

an idea we need to consider to provide a more flexible strategy.

To prevent over-allocation of memory, the page-fault service routine is modified to include page replacement. This ensures that the system does not run out of memory by managing the allocation and deallocation of pages dynamically.

The last step to provide virtual memory. Fully dynamic management of frames.

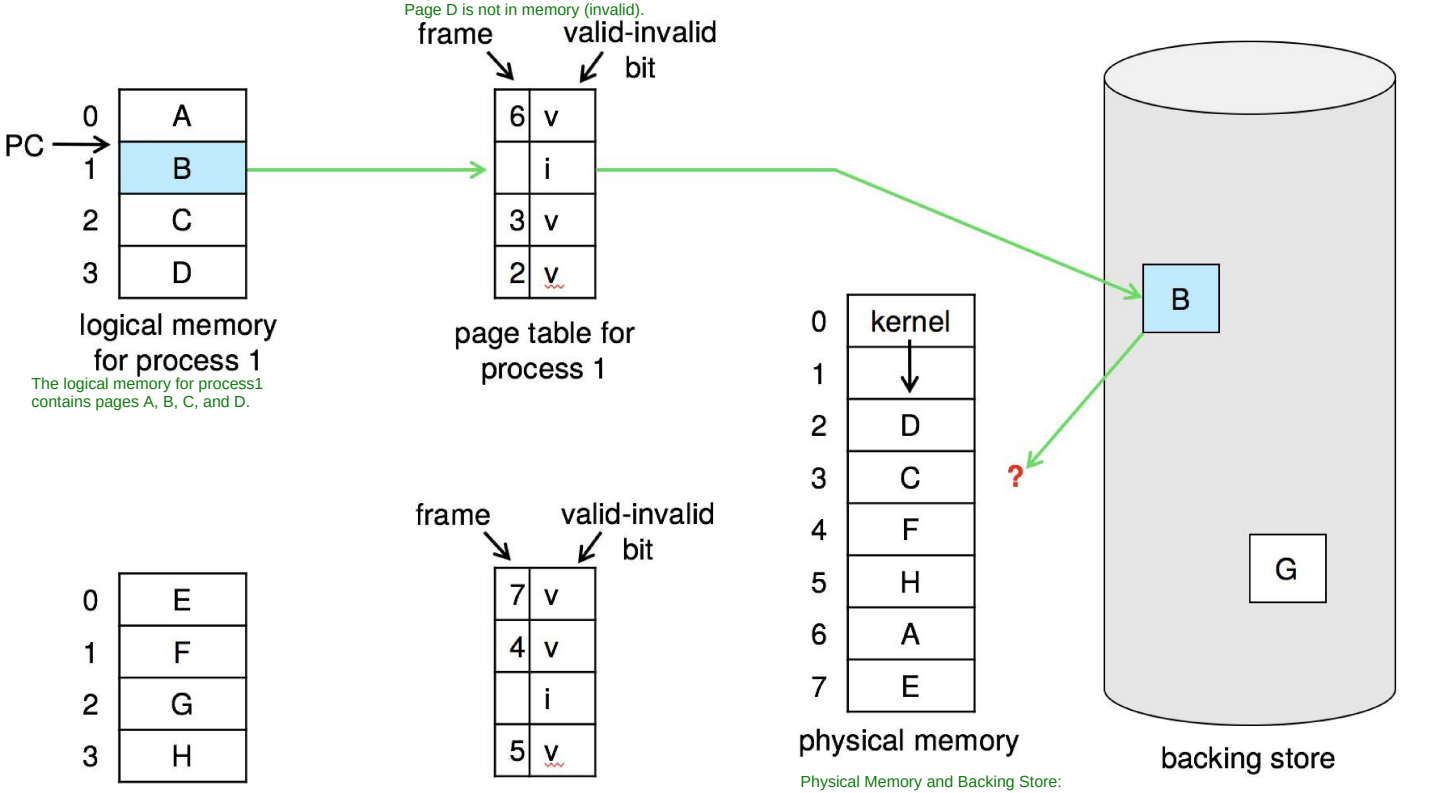
- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory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

The modify (or dirty) bit is used to reduce the overhead of page transfers. This bit indicates whether a page has been modified while in memory.
Usage: Only modified pages (those with the dirty bit set) need to be written back to disk. Unmodified pages can be discarded without writing them back, reducing the number of I/O operations.

Page replacement completes the separation between logical memory (the memory as seen by the process) and physical memory (the actual RAM).
This allows the system to provide a large virtual memory space on a smaller physical memory by dynamically managing which pages are in memory and which are on disk.

Need For Page Replacement

The diagram illustrates the need for page replacement when there are no free frames available.



Physical Memory and Backing Store:
Physical memory contains frames used by the kernel and various processes.
The backing store (disk) contains pages that are not currently in memory.
In this example, page B from process1 is in the backing store and needs to be brought into memory.

*****Page Replacement:
When process1 needs to access page B, a page fault occurs because page B is not in memory.
The system must find a free frame to load page B. If no free frames are available, the system must use a page replacement algorithm to free up a frame.
The algorithm will select a page that is not currently in use (e.g., page G) and page it out to the backing store to make room for page B.

capital letters represent the content

Basic Page Replacement

Find the Location of the Desired Page on Disk:
When a page fault occurs, the operating system must locate the desired page on the disk (backing store).

1. Find the location of the desired page on disk

once we have loaced the page on disk lets say B

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**

If there is a free frame: Use it to load the desired page.
If there is no free frame: Use a page replacement algorithm to select a victim frame (a frame currently in use that will be replaced).
Victim Frame: The selected frame that will be replaced.
Write to Disk if Dirty: If the victim frame has been modified (dirty), write it back to the disk before replacing it.

We need to find a page or cadidate for replacement if we dont find it.

   - Write victim frame to disk if dirty

Bring the Desired Page into the Free Frame:
Load the desired page from the disk into the (newly) free frame.
Update the page table and frame table to reflect the changes.

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

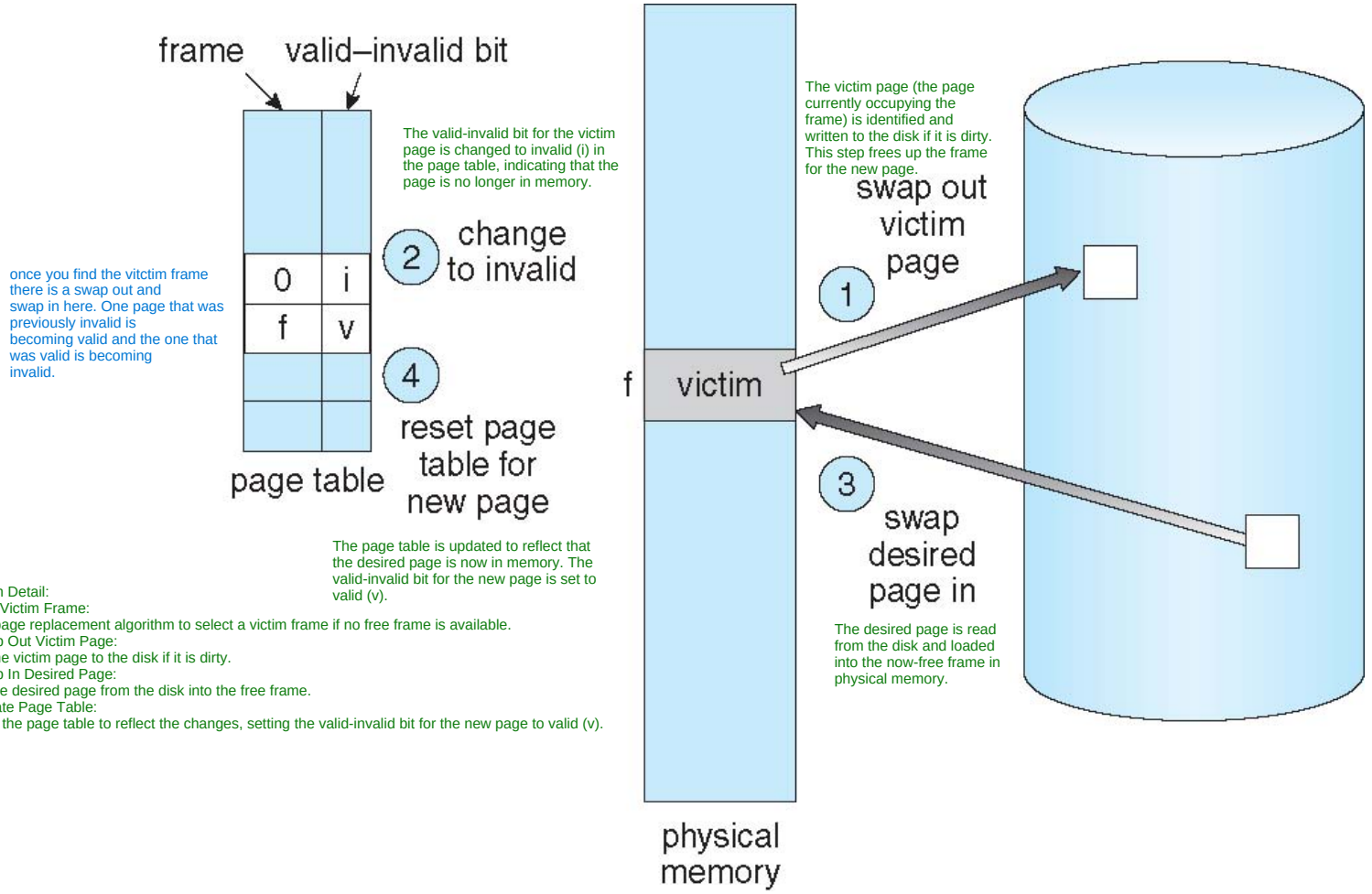
Restart the Instruction:
Continue the process by restarting the instruction that caused the page fault.

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

There may be two page transfers for a single page fault: one to swap out the victim page and another to swap in the desired page. This increases the Effective Access Time (EAT).

Page Replacement

The diagram illustrates the detailed steps involved in page replacement.



once you find the vitctim frame there is a swap out and swap in here. One page that was previously invalid is becoming valid and the one that was valid is becoming invalid.

The valid-invalid bit for the victim page is changed to invalid (i) in the page table, indicating that the page is no longer in memory.

The victim page (the page currently occupying the frame) is identified and written to the disk if it is dirty. This step frees up the frame for the new page.

The page table is updated to reflect that the desired page is now in memory. The valid-invalid bit for the new page is set to valid (v).

The desired page is read from the disk and loaded into the now-free frame in physical memory.

Steps in Detail:
1. Find Victim Frame:
Use a page replacement algorithm to select a victim frame if no free frame is available.
2. Swap Out Victim Page:
Write the victim page to the disk if it is dirty.
3. Swap In Desired Page:
Load the desired page from the disk into the free frame.
4. Update Page Table:
Update the page table to reflect the changes, setting the valid-invalid bit for the new page to valid (v).

Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace

Replace one page that is not necessary for you in furture. It is a predicative algo. How do we know the algo is good? Is future = past? depends? Evaluation should be experimental. training test and test set should be similar.
note highlights that frame-allocation algorithms often use predictive techniques to decide which pages to replace. The effectiveness of these algorithms can be evaluated experimentally by using similar training and test sets.

How many frames to allocate to each process. Which frames to replace when a page fault occurs.

The only meaningful data that we need for algo are the ref string. A program is going to make several read and writes to mem, reads can be instructions like fetch. Could be a list of mem locations that have been accessed for reading and writing.That could be a ref string. Reference string that has already filtered out page numbers

- **Page-replacement algorithm**

The objective is to achieve the lowest page-fault rate on both the first access and re-access of pages.

  - 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

Algorithms are evaluated by running them on a specific string of memory references (reference string) and computing the number of page faults that occur.
Reference String:
A sequence of page numbers that a process accesses.
Repeated access to the same page does not cause a page fault if the page is already in memory.
The results depend on the number of frames available for the process.

- In all our examples, the **reference string** of referenced page numbers is

what we can see certain pages are accessed several times like page 7 or 1 and few are used very less.

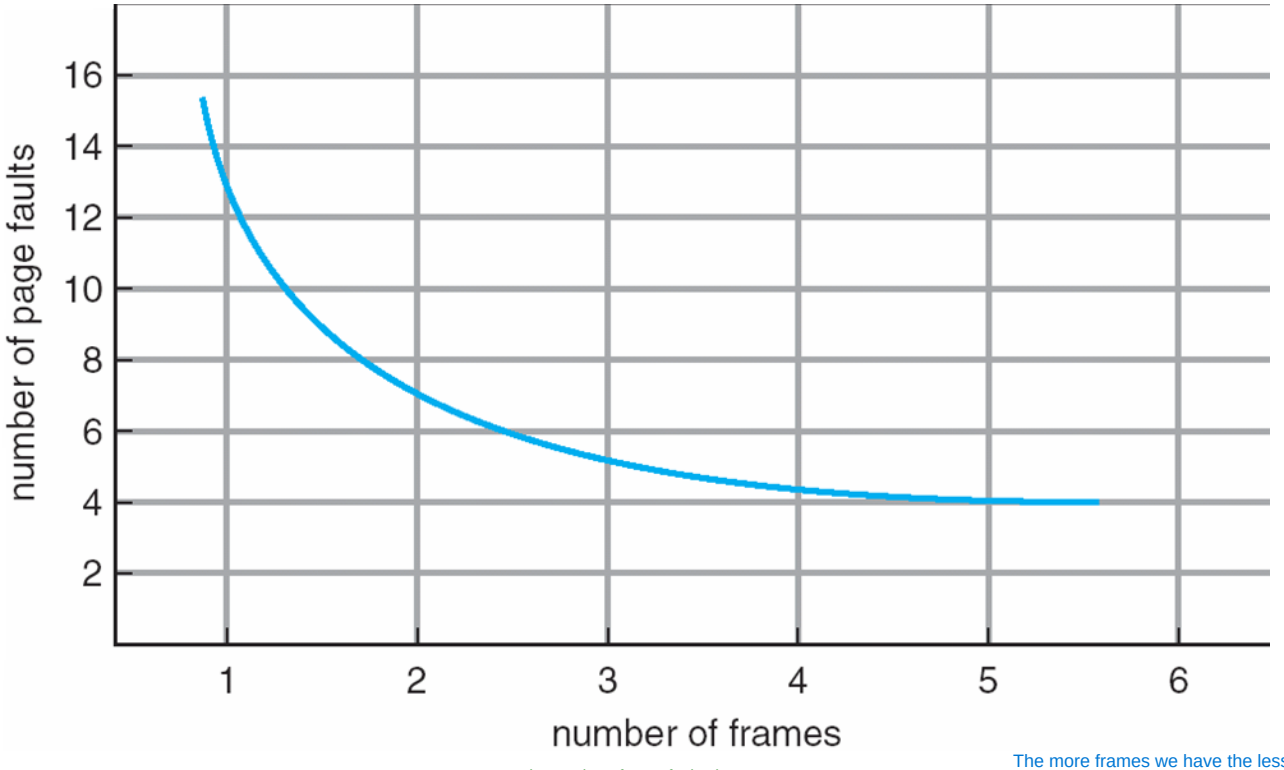
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

page 7, 0, 1.....

This string represents the sequence of page numbers accessed by a process.

Graph of Page Faults Versus The Number of Frames

The graph shows the relationship between the number of frames allocated to a process and the number of page faults that occur.



. X-Axis (Number of Frames): Represents the number of frames allocated to the process.

Represents the number of page faults that occur.

As the number of frames increases, the number of page faults decreases. With fewer frames, the process experiences more page faults because there is less space to hold the pages in memory.
As more frames are allocated, the process can hold more pages in memory, reducing the likelihood of page faults.

The more frames we have the less page faults we will trigger.
If you have only one frame it means that every time the program is changing the page there will be a page fault.

how many frames is that program going to use?

Page replacement strategies

Page Fault Frequency (empirical probability)

The formula for page fault frequency is given by:

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

Benchmarking Analogy:
Think of evaluating a page replacement algorithm like benchmarking a car. You measure various aspects (like speed and travel distance) and take an average. Similarly, you evaluate the algorithm using multiple reference strings, count the page faults, and calculate the frequency.
Variables Influencing \(f\):
The frequency of page faults (f) is influenced by how rapidly pages change and how well the algorithm predicts which pages will not be used in the future. If pages change rapidly, more page faults are likely. Choosing the right page to replace (one that won't be used soon) reduces page faults.
Weighted Average:
When calculating the average page fault frequency, you might use a weighted average. This means giving different weights to different reference strings based on their probability or importance.
Formula Usage:
The formula is important for exams. It helps compare different algorithms by calculating the weighted average of page faults across multiple reference strings.

The page replacement algorithm being evaluated.

- *A* page replacement algorithm under evaluation
- *w* a given reference string

A specific reference string, which is a sequence of page accesses.

- *p*(*w*) probability of reference string *w*

The probability of encountering the reference string

- *len*(*w*) length of reference string *w*

The length of the reference string w, i.e., the number of page accesses in the string.

benchmarking of a car for eg would mean longer travel, speed, then take an avg of everything. lets say we have multiple reference string W. Ref string like the one we saw in the above slides. If you count the num of page faults and call it F and devide by length of ref string you get the frequency of page fault. Num of page faults over total num of accesses.

what are the variables that are influencing f?
If you have pages changing very rapidly you will prolly have
more page faults. if you choose the right victim not to be used in future you will end up with less page faullts.

- *m* number of available page frames

The 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.

The number of page faults generated by algorithm A with m frames for the reference string w.

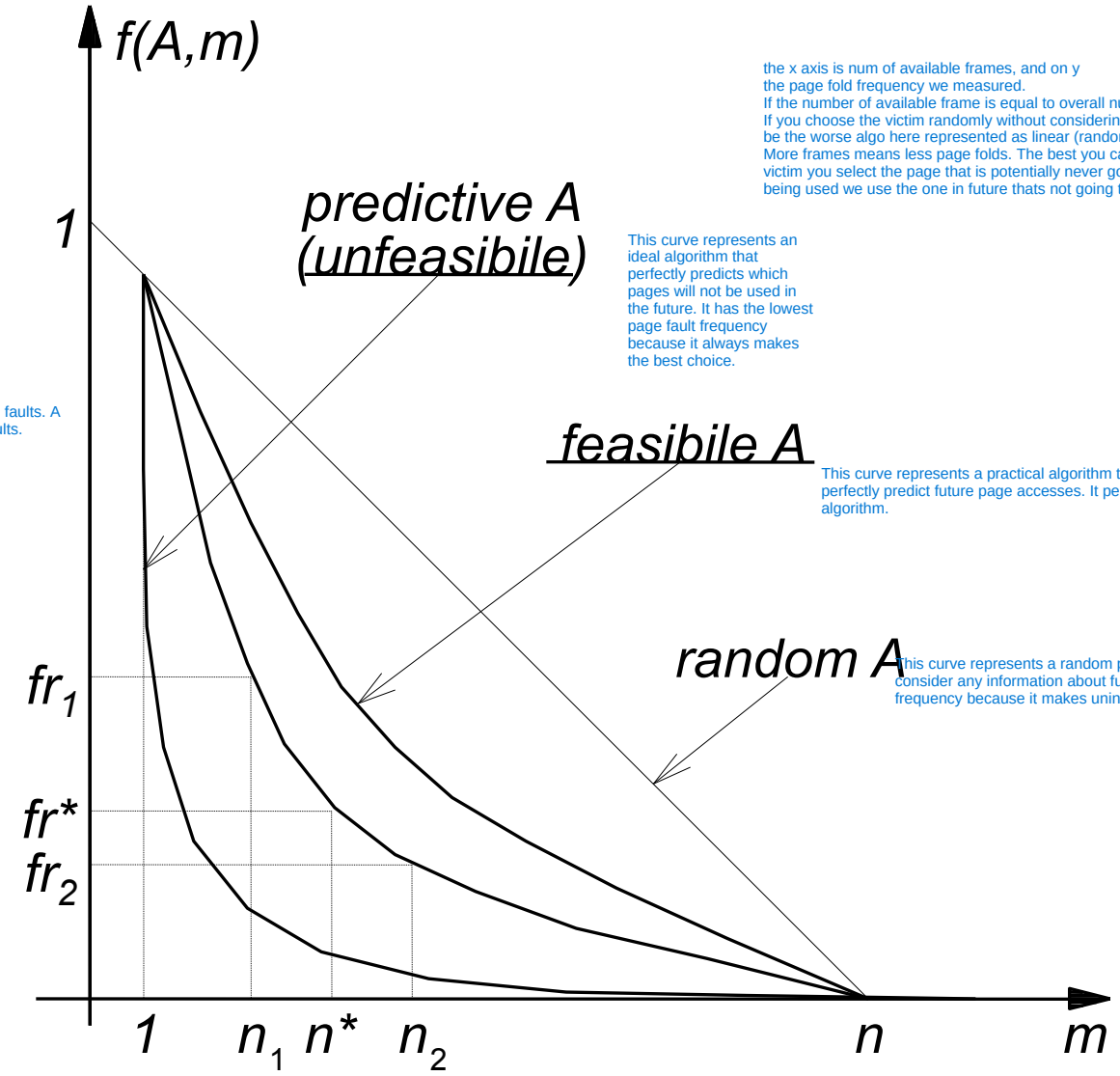
Once you have this fraction for ref string 1, ref string 2, ref string 3..... How to consider an average? weighted average? You are assuming all ref strings are meaningful.

Lets say 1st ref string 50% of y behavior, 2nd 25% and 3rd 10%. It is basically a weight of probabily. You compare it and choose the best one.

formula used in exam so remember.

The graph shows how the number of available frames (m) affects the page fault frequency f(A,m)) for different page replacement algorithms

page fault frequency as a function of m



the x axis is num of available frames, and on y
the page fold frequency we measured.
If the number of available frame is equal to overall num of pages that means we will minimize.
If you choose the victim randomly without considering anything, you should expect that random should be the worse algo here represented as linear (random A).
More frames means less page folds. The best you can obtain is predictive A, if you have to select the victim you select the page that is potentially never going to be used in the future. if all the pages are being used we use the one in future thats not going to be used. it end up with best curve.

This curve represents an ideal algorithm that perfectly predicts which pages will not be used in the future. It has the lowest page fault frequency because it always makes the best choice.

Represents the frequency of page faults. A lower value means fewer page faults.

This curve represents a practical algorithm that aims to minimize page faults but cannot perfectly predict future page accesses. It performs well but not as perfectly as the predictive algorithm.

This curve represents a random page replacement algorithm, which does not consider any information about future page accesses. It has the highest page fault frequency because it makes uninformed decisions.

X and Y Axes:
The x-axis shows the number of available frames, and the y-axis shows the measured page fault frequency.
Minimizing Page Faults:
If the number of available frames equals the total number of pages, page faults are minimized because all pages can be kept in memory.
Random Algorithm:
A random page replacement algorithm is expected to perform poorly because it does not use any information to make informed decisions about which page to replace.
More Frames, Fewer Page Faults:
More frames generally lead to fewer page faults because more pages can be kept in memory.
The best algorithm (predictive A) would perfectly predict future page accesses and select the page that will not be used again, resulting in the lowest page fault frequency.

Represents the number of available page frames. More frames mean more memory available to hold pages.

Select victim using modify bit

Reference Bit: Indicates whether a page has been accessed recently.
0: The page has not been accessed since the bit was last cleared.
1: The page has been accessed.

Modify Bit: Indicates whether a page has been modified (written to).
0: The page has not been modified.
1: The page has been modified.

Reference bit	Modify bit
0	0
0	1
1	0
1	1

Using the Bits to Select a Victim:
The combination of the reference bit and modify bit helps the operating system decide which page to replace.
Pages with a reference bit of 0 and a modify bit of 0 are ideal candidates for replacement because they have not been accessed or modified recently.

what kinda data are the algo allowed to observe to make predictions.
Reference bit 0 says no access to page since bit was cleared. modify bit 0 means most of the writes modify it is very rare that a write is rewriting the data. The modify bit means no modification .
1 1 means access and modification means okay. 0,1 means how can the page be modified and not accessed? Ref bit and modify bit are not necessarily cleared or sent at the same moment. it can mean modify bit was cleared long time ago and ref bit was cleared recently. The reference bit will be used by algorithms and reset frequently. simply because ref is cleared frequently we can have 0,1.

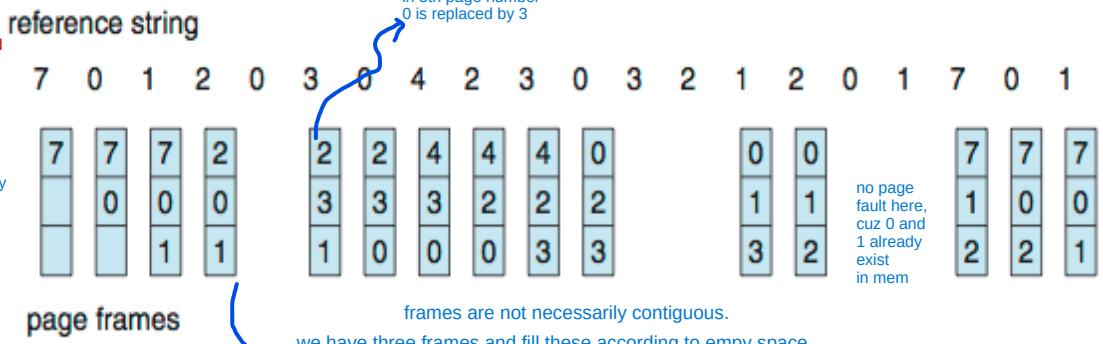
Data for Algorithms:
Algorithms use the reference and modify bits to make predictions about which pages to replace.
Reference Bit 0: Indicates no access to the page since the bit was cleared.
Modify Bit 0: Indicates no modification to the page.
Reference Bit 1 and Modify Bit 1: Indicates the page has been accessed and modified.
Reference Bit 0 and Modify Bit 1: Indicates the page has been modified but not accessed recently.
**Using the Bits to Select a Victim:
The combination of the reference bit and modify bit helps the operating system decide which page to replace.
Pages with a reference bit of 0 and a modify bit of 0 are ideal candidates for replacement because they have not been accessed or modified recently.
Clearing Bits:
The reference and modify bits are not necessarily cleared at the same time.
For example, the modify bit might have been cleared a long time ago, while the reference bit was cleared recently.
The reference bit is used frequently by algorithms and reset often, which is why you can have combinations like 0 (reference bit) and 1 (modify bit).

First-In-First-Out (FIFO) Algorithm

The FIFO algorithm replaces the oldest page in memory (the one that was loaded first) when a page fault occurs.

- 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)

The diagram shows the sequence of page replacements and the resulting page faults. Each time a new page is loaded into memory and a page is replaced, a page fault occurs.



in 5th page number 0 is replaced by 3

The three frames are represented close to each other in the diagram for clarity, but they can be located anywhere in memory.

no page fault here, cuz 0 and 1 already exist in mem

Typical queue. If you have a ref string, when i need a replacement the victim will be the page that among all pages in memory. the page that was entered in the frame earliest in the past.

The FIFO algorithm uses a queue to keep track of the order in which pages were loaded. The oldest page (at the front of the queue) is replaced first.Example: When page 7 is replaced by page 2, the head of the queue moves, and the new page is added to the end of the queue.

frames are not necessarily contiguous.

we have three frames and fill these according to empy space.

in 4th(3 frames) one page num 7 is replaced by 2 as fifo.

15 page faults\

The diagram shows 15 page faults for the given reference string. Each rectangle with three frames indicates a page fault. No rectangles mean no page fault occurred.

- 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**

Adding more frames can sometimes cause more page faults, which is counterintuitive. This phenomenon is known as Belady's Anomaly.

- How to track ages of pages?
  - Just use a FIFO queue

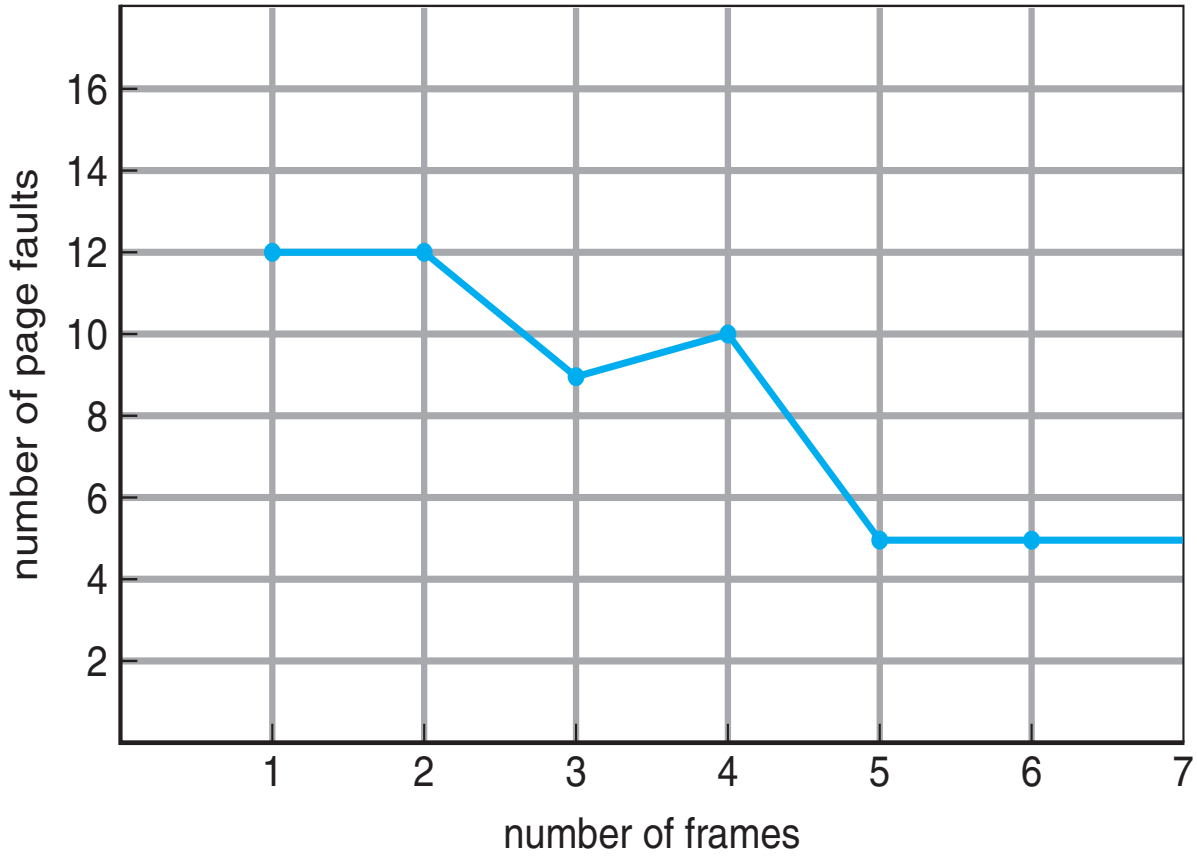
The FIFO algorithm uses a queue to keep track of the order in which pages were loaded. The oldest page (at the front of the queue) is replaced first.

Initial state all frames empty
Page 7: Loaded in the first frame.
Page 0: Loaded into the second frame.
Page 1: Loaded into the third frame.
Page 2: Replaces page 7 (FIFO).
Page 0: Already in memory, no page fault.
Page 3: Replaces page 0 (FIFO).
Page 0: Replaces page 1 (FIFO).
Page 4: Replaces page 2 (FIFO).
Page 2: Replaces page 3 (FIFO).
Page 3: Replaces page 0 (FIFO).
Page 0: Replaces page 4 (FIFO).
Page 3: Already in memory, no page fault.
Page 2: Replaces page 3 (FIFO).
Page 1: Replaces page 0 (FIFO).
Page 2: Already in memory, no page fault.
Page 0: Replaces page 1 (FIFO).
Page 1: Replaces page 2 (FIFO).
Page 7: Replaces page 0 (FIFO).
Page 0: Replaces page 1 (FIFO).
Page 1: Replaces page 7 (FIFO).

FIFO Illustrating Belady's Anomaly

Belady's Anomaly is a phenomenon where increasing the number of page frames results in an increase in the number of page faults. This is counterintuitive because one would expect that having more frames would always reduce the number of page faults.

Graph Explanation:
The graph shows the number of page faults on the y-axis and the number of frames on the x-axis.
The reference string used for this example is not explicitly mentioned, but it demonstrates how the FIFO algorithm can exhibit Belady's Anomaly.



With 1 frame, there are 12 page faults.
With 2 frames, the number of page faults remains the same.
With 3 frames, the number of page faults decreases to 9.
With 4 frames, the number of page faults increases to 10, illustrating Belady's Anomaly.
With 5, 6, and 7 frames, the number of page faults decreases again.

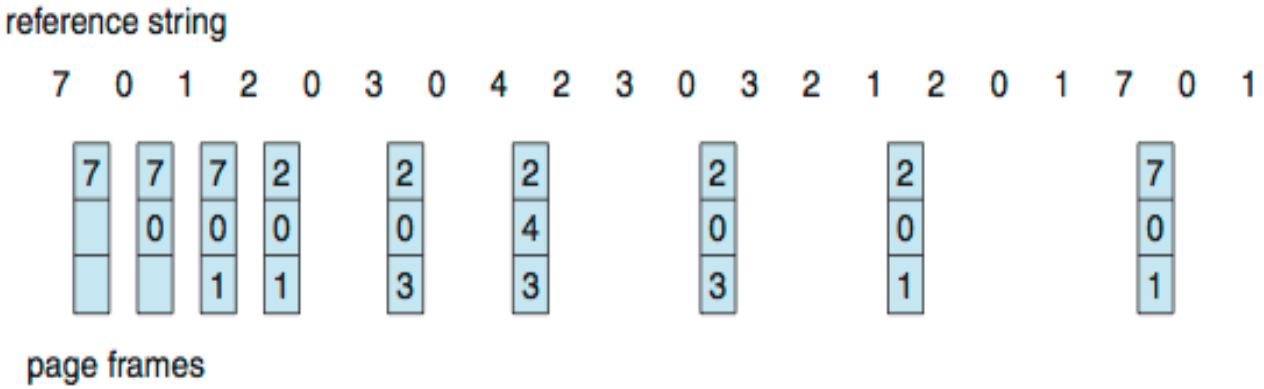
Optimal Algorithm

Optimal Page Replacement Algorithm:
The optimal page replacement algorithm replaces the page that will not be used for the longest period of time in the future.
This algorithm is theoretical because it requires knowledge of future page references, which is not possible in practice.

- 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

The algorithm looks ahead in the reference string to determine which page will not be used for the longest period of time and replaces that page.
For example, after the third page (2) is loaded, all frames are full. When the next page (0) needs to be loaded, the algorithm looks ahead and determines which of the current pages (7, 0, 1) will not be used for the longest time and replaces that page.

** The optimal algorithm is used as a benchmark to measure how well other page replacement algorithms perform. Since it is not feasible to implement in practice, it serves as an ideal standard.



Initial State: All frames are empty.
Page 7: Loaded into the first frame.
Page 0: Loaded into the second frame.
Page 1: Loaded into the third frame.
Page 2: Replaces page 7 (optimal choice).
Page 0: Already in memory, no page fault.
Page 3: Replaces page 1 (optimal choice).
Page 0: Already in memory, no page fault.
Page 4: Replaces page 2 (optimal choice).
Page 2: Replaces page 3 (optimal choice).
Page 3: Replaces page 0 (optimal choice).
Page 0: Replaces page 4 (optimal choice).
Page 3: Already in memory, no page fault.
Page 2: Replaces page 3 (optimal choice).
Page 1: Replaces page 0 (optimal choice).
Page 2: Already in memory, no page fault.
Page 0: Replaces page 1 (optimal choice).
Page 1: Replaces page 2 (optimal choice).
Page 7: Replaces page 0 (optimal choice).
Page 0: Replaces page 1 (optimal choice).
Page 1: Replaces page 7 (optimal choice).

If you already had the string of ref from now to the end. If I knew the future what could I do? I know when the page will be used in the future. after the third one we need a replacement cuz all frames are full so for 2 we need replacement. We look in the future which is the farthest???

Least Recently Used (LRU) Algorithm

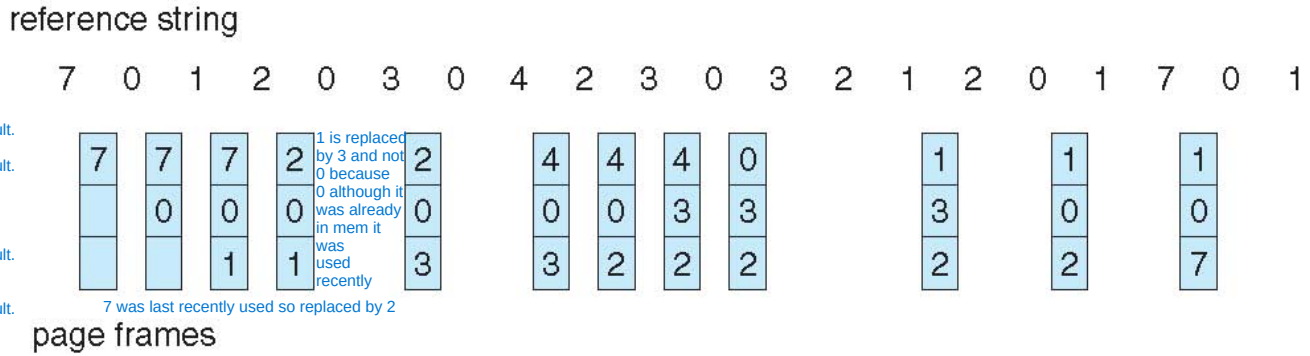
The Least Recently Used (LRU) algorithm replaces the page that has not been used for the longest period of time.

Unlike the optimal algorithm, which looks into the future, LRU uses past knowledge to make decisions.

I need to look at the past and not consider when a page entered in the system but the last time it was used cuz it is indicative of how much the page has been used in the near past. The closer this page has been used the higher probability that it is still used.

- 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

How It Works:
The algorithm keeps track of the last time each page was used. When a page fault occurs and there are no free frames, the page that has not been used for the longest time is replaced.



1 is replaced by 3 and not 0 because 0 although it was already in mem it was used recently

7 was last recently used so replaced by 2

Page 7: Loaded into the first frame.
Page 0: Loaded into the second frame.
Page 1: Loaded into the third frame.
Page 2: Replaces page 7 (LRU).
Page 0: Already in memory, no page fault.
Page 3: Replaces page 1 (LRU).
Page 0: Already in memory, no page fault.
Page 4: Replaces page 2 (LRU).
Page 2: Replaces page 3 (LRU).
Page 3: Replaces page 0 (LRU).
Page 0: Replaces page 4 (LRU).
Page 3: Already in memory, no page fault.
Page 2: Replaces page 3 (LRU).
Page 1: Replaces page 0 (LRU).
Page 2: Already in memory, no page fault.
Page 0: Replaces page 1 (LRU).
Page 1: Replaces page 2 (LRU).
Page 7: Replaces page 0 (LRU).
Page 0: Replaces page 1 (LRU).
Page 1: Replaces page 7 (LRU).

- 12 faults – better than FIFO but worse than OPT

The LRU algorithm results in 12 page faults for the given reference string. This is better than the FIFO algorithm but worse than the optimal algorithm.

- Generally good algorithm and frequently used
- But how to implement?

Implementing LRU can be challenging because it requires keeping track of the order of page accesses. This can be done using data structures like stacks or linked lists.

On a page fault with no free frame, the LRU algorithm selects the page that has not been used for the longest time as the victim for replacement.

Least Recently Used Algorithm

On page fault with no free frame, victim is page that has not been used in the most amount of time

The table shows a sequence of page accesses and the resulting page faults using the LRU algorithm.

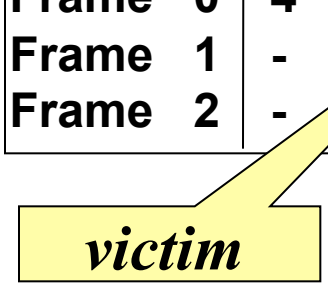
Time		1	2	3	4	5	6	7	8	9	10	11	12
String		4	3	2	1	4	3	5	4	3	2	1	5
Fault		*	*	*	*	*	*	*			*	*	*
Frame	0	4	4	4	1	1	1	5	5	5	2	2	2
Frame	1	-	3	3	3	4	4	4	4	4	4	1	1
Frame	2	-	-	2	2	2	3	3	3	3	3	3	5

Reference String

Three frames are available.

The table shows the state of the frames after each page access.
A page fault occurs when a page needs to be loaded into a frame, and the frame is already occupied by another page.
The victim page (the one to be replaced) is highlighted in red.

The table shows the sequence of page replacements for the given reference string using the LRU algorithm:
Time: The time step for each page access.
String: The reference string of page accesses.
Fault: Indicates whether a page fault occurred ().
Frames: The state of the frames after each page access.
Victim: The page that is replaced (highlighted in red).



$$f = 10 / 12 = 0.83 \Rightarrow 83\ \%$$

The page fault frequency is calculated as the number of page faults divided by the total number of page accesses.
In this example, there are 10 page faults out of 12 accesses, resulting in a page fault frequency of 83%.

LRU Algorithm (Cont.)

- Counter implementation

How It Works:
Every page entry has a counter.
Each time a page is referenced, the current time (clock) is copied into the counter for that page.

  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value

When a page needs to be replaced, the algorithm searches through the table of counters to find the page with the smallest value (i.e., the page that was used the longest time ago).

    - Search through table needed

Searching through the table to find the smallest counter value can be time-consuming.

- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:

A stack of page numbers is maintained in a double-linked list form.
When a page is referenced, it is moved to the top of the stack.
This requires updating pointers to maintain the stack order.

    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement

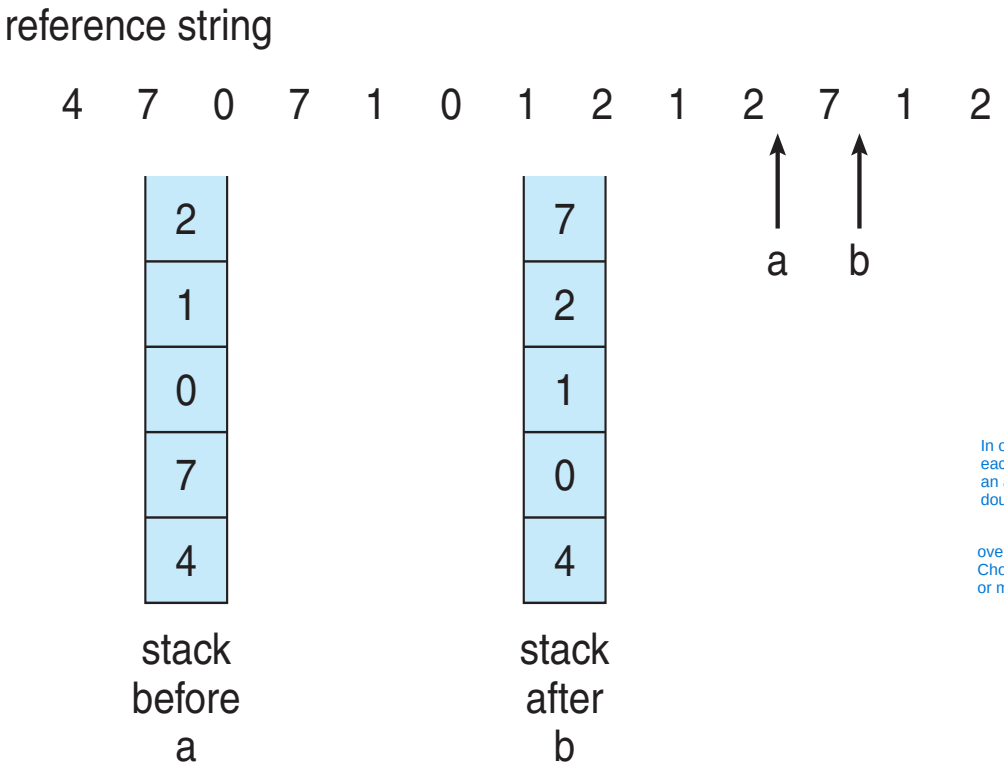
Page Replacement:
The page at the bottom of the stack (the least recently used page) is replaced when a page fault occurs.
Advantages:
No need to search for the page to replace, as the least recently used page is always at the bottom of the stack.
Drawback:
Each update is more expensive because it requires changing multiple pointers (six pointers for each memory access).

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

Both LRU and the optimal algorithm (OPT) are examples of stack algorithms.
Stack algorithms do not suffer from Belady's Anomaly, meaning that increasing the number of frames will not increase the number of page faults.

Use Of A Stack to Record Most Recent Page References

1. Reference 4:
Stack: [4]
2. Reference 7:
Stack: [7, 4]
3. Reference 0:
Stack: [0, 7, 4]
4. Reference 7:
Page 7 is already in the stack, move it to the top.
Stack: [7, 0, 4]
5. Reference 1:
Stack: [1, 7, 0, 4]
6. Reference 0:
Page 0 is already in the stack, move it to the top.
Stack: [0, 1, 7, 4]
7. Reference 1:
Page 1 is already in the stack, move it to the top.
Stack: [1, 0, 7, 4]
Reference 2:
Stack: [2, 1, 0, 7, 4]
9. Reference 1:
Page 1 is already in the stack, move it to the top.
Stack: [1, 2, 0, 7, 4]
10. Reference 2:
Page 2 is already in the stack, move it to the top.
Stack: [2, 1, 0, 7, 4]
Stack Before 'a'
At this point, the stack before operation 'a' is:
Stack: [2, 1, 0, 7, 4]
11. Reference 7 (Operation 'a'):
Page 7 is already in the stack, move it to the top.
Stack: [7, 2, 1, 0, 4]
12. Reference 1:
Page 1 is already in the stack, move it to the top.
Stack: [1, 7, 2, 0, 4]
13. Reference 2 (Operation 'b'):
Page 2 is already in the stack, move it to the top.
Stack: [2, 1, 7, 0, 4]
Stack After 'b'
After operation 'b', the stack is:
Stack: [2, 1, 7, 0, 4]



In order to implement this you need a double linked list cuz each page should exactly have 2 points. Instead of having an array of counters we have an array of pointers. we need double linked list so that removal is possible.

overall six pointers to be modified for every memory access. Choose the stack implementation, either restore the counter or modify the pointers.

Double-Linked List:
To implement the stack, a double-linked list is used because each page should have two pointers (one to the previous page and one to the next page).
Instead of using an array of counters, an array of pointers is used.
A double-linked list allows for efficient removal and insertion of pages.
Pointer Updates:
Overall, six pointers need to be modified for every memory access.
The choice is between restoring the counter or modifying the pointers.

The stack implementation of the LRU algorithm involves moving referenced pages to the top of the stack. This ensures that the most recently used pages are at the top, and the least recently used pages are at the bottom. The implementation requires updating multiple pointers for each memory access.

LRU Approximation Algorithms

- LRU needs special hardware and still slow

Implementing the exact LRU algorithm requires special hardware to keep track of the order of page accesses, which can be complex and slow.

- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however

Reference Bit:
How It Works:
Each page is associated with a reference bit, initially set to 0.
When a page is referenced, its reference bit is set to 1.
Page Replacement:
When a page needs to be replaced, any page with a reference bit of 0 can be chosen.
The order of pages with reference bit 0 is not known, so any of them can be replaced.

- **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

This algorithm is an enhancement of the FIFO algorithm, incorporating a reference bit.
It is also known as the clock replacement algorithm because it can be visualized as a circular buffer (clock).
Page Replacement:
If the page to be replaced has a reference bit of 0, it is replaced.
If the page to be replaced has a reference bit of 1, the reference bit is set to 0, and the page is given a "second chance." The algorithm then moves to the next page and repeats the process.

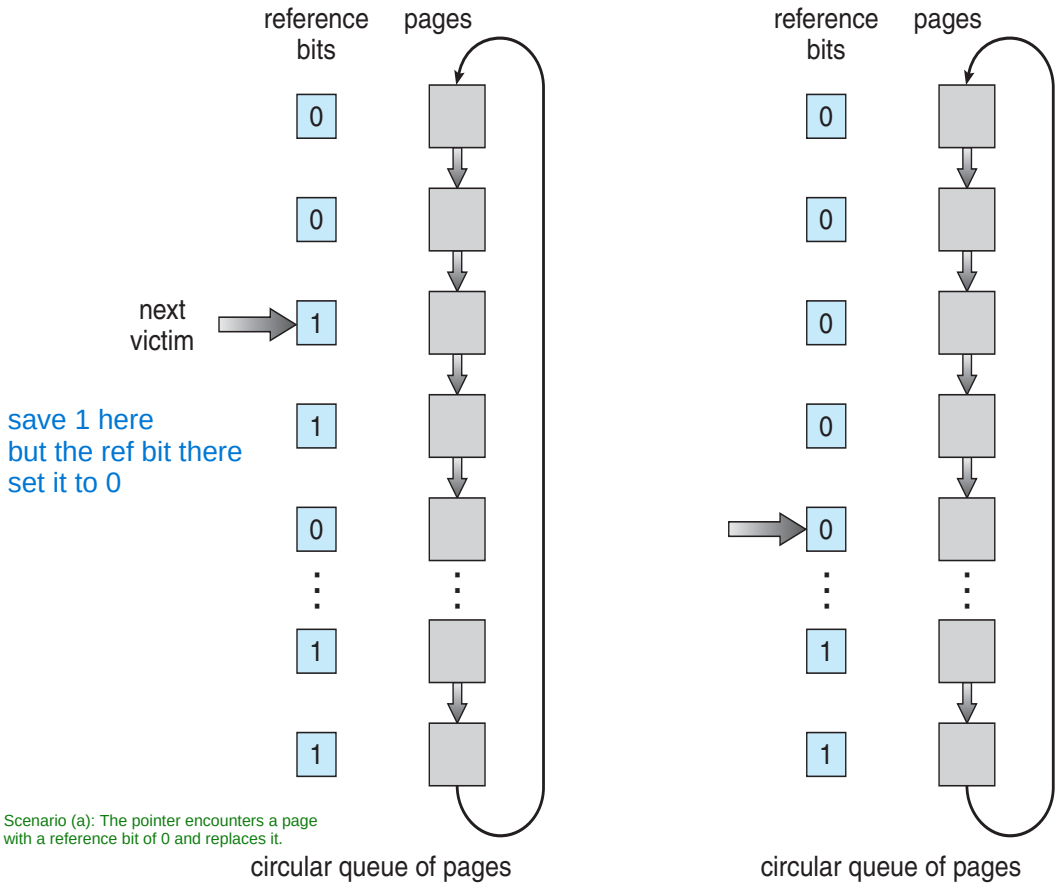
this one is more evolved as it is less vague! When do we clear the ref bit?
it is fifo plus management of ref bit! Clock replacement means circular buffer

The victim in priciple is the oldest that entered the memory.

Advantages:
This algorithm is more efficient than pure FIFO because it takes into account whether a page has been used recently.

Second-Chance (clock) Page-Replacement Algorithm

Clock Replacement:
The clock replacement algorithm is a practical implementation of the second-chance algorithm.
It uses a circular queue of pages, with a pointer that moves around the circle to find a page to replace.



Reference Bits:
Each page has an associated reference bit.
The reference bit is set to 1 when the page is referenced.
Page Replacement:
The pointer moves around the circular queue.
If it encounters a page with a reference bit of 0, that page is replaced.
If it encounters a page with a reference bit of 1, the reference bit is set to 0, and the pointer moves to the next page.

Scenario (a): The pointer encounters a page with a reference bit of 0 and replaces it.

Scenario (b): The pointer encounters pages with reference bits of 1, sets them to 0, and continues until it finds a page with a reference bit of 0 to replace.

In the diagram, the pointer moves around the circular queue.
In the first scenario (a), the pointer encounters a page with a reference bit of 0 and replaces it.
In the second scenario (b), the pointer encounters pages with reference bits of 1, sets them to 0, and continues until it finds a page with a reference bit of 0 to replace.

Enhanced Second-Chance Algorithm

Improving the Algorithm:
The second-chance algorithm can be improved by using both the reference bit and the modify bit (if available) together.
This allows for a more informed decision when selecting a page to replace.

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used not 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 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

rdered Pair (Reference, Modify):
The algorithm considers the combination of the reference bit and the modify bit to determine which page to replace:
(0, 0): The page has not been recently used and has not been modified. This is the best candidate for replacement because it is not in use and does not need to be written back to disk.
(0, 1): The page has not been recently used but has been modified. This page is not as good a candidate for replacement because it needs to be written back to disk before being replaced.
(1, 0): The page has been recently used and is clean (not modified). This page is likely to be used again soon, so it is not a good candidate for replacement.
(1, 1): The page has been recently used and modified. This page is also likely to be used again soon and needs to be written back to disk before being replaced.

When a page replacement is needed, the algorithm uses the clock scheme but considers the four classes of pages (based on the reference and modify bits). The algorithm replaces the page in the lowest non-empty class. This may require searching the circular queue several times to find the best candidate for replacement.

Counting 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

Keeping a Counter:
Counting algorithms keep a counter of the number of references made to each page.
This is not a common approach but can be useful in certain scenarios.

Least Frequently Used (LFU) Algorithm:
How It Works:
The LFU algorithm replaces the page with the smallest count (i.e., the page that has been used the least frequently).
Rationale:
The idea is that pages that have been used less frequently in the past are less likely to be used in the future.

How It Works:
The MFU algorithm replaces the page with the largest count (i.e., the page that has been used the most frequently).
Rationale:
The argument is that the page with the smallest count was probably just brought in and has yet to be used, so it should not be replaced.

Counting how many times a page has been used. The other counter was a counter timer. So many references that the page had and instead of using the least. If the page was used less frequenntly than other pages in the past than it will be used less frequently even in the future. We can have least frequently and most frequently. It is an idea what could be considered as another potential.

Counting References:
Counting algorithms keep track of the number of references made to each page.
This can be done using a counter for each page.
LFU Algorithm:
The LFU algorithm replaces the page with the smallest count, based on the idea that pages used less frequently in the past are less likely to be used in the future.
MFU Algorithm:
The MFU algorithm replaces the page with the largest count, based on the idea 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

If you don't find a free page, take the victim replace it and free the frame. Why don't we have a pool of free frames and choose from it when required. There should be some moments at time when there are not enough free frames.

Always maintain a pool of free frames. This ensures that a frame is available when needed, avoiding the need to find a free frame at the time of a page fault.

When a page fault occurs, read the page into a free frame from the pool. Select a victim page to evict and add its frame to the free pool. Evict the victim page when convenient, not necessarily immediately.

Free frame is available when needed but its not created at full time. In other moments we are taking care of creating free frames. The problem with freeing a frame when the page is modified is saving that page back in storage. So the idea is the following: I have a page fault and i need a free frame for that page fault, I use the free frame here and label a victim.
So the idea is i take the free frame and find a candidate for another free frame.

Keep a list of modified (dirty) pages. Write these pages to the backing store (disk) when the system is otherwise idle. This helps to reduce the time needed to write out modified pages during a page replacement.

Keep Free Frame Contents Intact:
Concept:
Keep the contents of free frames intact and note what is in them.
If a page is referenced again before the frame is reused, there is no need to load the contents again from the disk.
Benefit:
This reduces the penalty if the wrong victim frame is selected, as the page can be quickly re-referenced without a disk read.

Another advtg of this kinda buffering is: Suppose the op system has not yet saved the page, you can access the page number in that frame that's marked ready to be freed.

Page-Buffering Algorithms Slide:
Free Frames:
If a free page is not found, take the victim page, replace it, and free the frame.
Maintain a pool of free frames to choose from when required.
There should be moments when there are not enough free frames.
Free Frame Availability:
Free frames are available when needed but are not created at all times.
The idea is to take a free frame and find a candidate for replacement.
Modified Pages:
Keeping a list of modified pages helps manage the backing store efficiently.
Write pages to the backing store when it is idle and set them to non-dirty.
Free Frame Contents:
Keeping free frame contents intact allows for quick re-referencing if the page is needed again.
This reduces the penalty if the wrong victim frame is selected.

Applications and Page Replacement

OS Guessing About Future Page Access:
All page replacement algorithms involve the operating system making guesses about future page accesses.
The accuracy of these guesses affects the performance of the algorithms.

- All of these algorithms have OS guessing about future page access

Some applications, such as databases, have better knowledge about their own access patterns.
These applications can provide hints to the operating system to improve page replacement decisions.

- 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

when you have many copies better to use intermediate buffer as you don't req full synconization

Memory-intensive applications can cause double buffering, where both the operating system and the application keep copies of the same data in memory.
This can lead to inefficiencies.

- Operating system can given direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc

The operating system can give applications direct access to the disk, bypassing the OS's buffering mechanisms.
This is known as raw disk mode and can improve performance by reducing overhead.

In raw disk mode, applications can bypass the operating system's buffering, locking, and other mechanisms.
This can lead to more efficient data access for certain applications.

A page is considered imp or not imp based on times its accessed.

OS Guessing:
The operating system guesses about future page accesses, which affects the performance of page replacement algorithms.
Intermediate Buffer:
When there are many copies, it is better to use an intermediate buffer if full synchronization is not required.
Page Importance:
A page is considered important or not based on how frequently it is accessed.

Allocation of Frames

Each process needs a minimum number of frames to function properly.
A frame is a fixed-size block of memory that holds a page.

- 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

The IBM 370 computer system needs 6 pages to handle an SS MOVE instruction.
Instruction Details:
The instruction is 6 bytes long and might span 2 pages.
2 pages are needed to handle the "from" part of the instruction.
2 pages are needed to handle the "to" part of the instruction.
This means a total of 6 pages are needed to handle this instruction.

The maximum number of frames a process can use is the total number of frames available in the system.

There are two major schemes for allocating frames to processes:
Fixed Allocation: Each process is given a fixed number of frames.
Priority Allocation: Frames are allocated based on the priority of the process.
There are many variations of these schemes.

are you going to trigger 6 page faults yes, but you need 6 frames. So this is simply telling you don't think olnly 1 frame is req.

The location of frames? We have seen process with three frames, extreme case 1 frame or enough frame for all pages. Which is the minimum num of frames req by a process? Suppose an instruction triggers a page fault! Is it sufficient to get one frame for it?

Its not true! One fault doesn't require one frame. There are instuction that need upto six pages. An instuction typically needs the page where it is, an instruction of 6 bytes could overlap of ending of one and beginning of another page. so for instuction fetch you need two pages. so for 1 fetch you could need 2. most of the instructions when you execute them typically they move data.

Page Faults:
If an instruction triggers a page fault, it may need multiple frames to handle the fault.
For example, an instruction that is 6 bytes long might need 2 pages, so 2 frames are required.

when allocating frames to processes.

Fixed Allocation

In equal allocation, each process is given an equal number of frames

If there are 100 frames available and 5 processes, each process gets 20 frames.

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames

equal allocation is fair or unfair based on criteria you are choosing.

Free Frame Buffer Pool:
Some frames are kept as a free frame buffer pool to handle page faults and other needs.

  - Keep some as free frame buffer pool

if a larger process will take more frames the smaller process will take less frames.

- Proportional allocation – Allocate according to the size of process

In proportional allocation, frames are allocated based on the size of the process.

The allocation can change as the degree of multiprogramming (number of processes) and process sizes change.

  - Dynamic as degree of multiprogramming, process sizes change

If there are 64 frames and two processes with sizes 10 and 127:

$m = 64$

2 processes here one with 10 pages and one with 127 pages.

$s_1 = 10$

$s_2 = 127$

S = sum of the sizes of all processes

— $S = \sum s_i$

proportional to the relative size of the process

— m = total number of frames

— a_i = allocation for $p_i = \frac{s_i}{S} \times m$

alocate for process p(i)

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Equal Allocation:
Equal allocation can be fair or unfair depending on the criteria used.
If a larger process takes more frames, a smaller process will get fewer frames.
Proportional Allocation:
Proportional allocation is based on the relative size of the process.
The formula ensures that larger processes get more frames.
The example shows how frames are allocated based on the size of the processes.

instead of having 127 we have 64 frames less than half of what is needed!!! its a bug prolly 62 or 64.

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

In global replacement, a process can select a replacement frame from the set of all frames in the system.
This means one process can take a frame from another process if needed

Greater flexibility and optimization because the process can choose from a larger pool of frames.
This can lead to greater overall system throughput, making it a more common approach.

Disadvantages:
The execution time of a process can vary greatly because its frames can be taken by other processes.
A process may be highly impacted by the actions of other processes, leading to potential performance issues.

The process could select frame from all frames.

this opens more flexibility and optimization but also means that a given process can highly be impacted by other processes as well.

Local rep is very simple,

Local Replacement:
Concept:
In local replacement, each process can only select from its own set of allocated frames. A process cannot take frames from other processes.
Advantages:
More consistent per-process performance because each process has a fixed set of frames. Processes are not affected by the actions of other processes.
Disadvantages:
Memory may be underutilized because a process cannot use free frames from other processes. Less flexibility and optimization compared to global replacement.

G

it is clear that global replacement has more flexibility and room for improvement but its just impacted by other processes.

Global vs. Local Allocation:
In global replacement, a process can select a replacement frame from all available frames, providing greater flexibility and optimization but potentially leading to variable execution times and performance impacts from other processes. In local replacement, each process can only select from its own set of allocated frames, providing consistent performance but potentially leading to underutilized memory.

Reclaiming Pages

- A strategy to implement 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.

Global Page-Replacement Policy:
Concept:
A strategy to implement a global page-replacement policy involves managing the free-frame list effectively.
Memory Requests:
All memory requests are satisfied from the free-frame list.
Instead of waiting for the free-frame list to drop to zero, the system begins selecting pages for replacement when the list falls below a certain threshold.
Page Replacement:
Page replacement is triggered when the free-frame list falls below a certain threshold.
This ensures that there is always sufficient free memory to satisfy new requests.

This strategy helps maintain a pool of free frames, ensuring that memory requests can be satisfied quickly.
It prevents the system from running out of free frames, which can lead to performance issues.

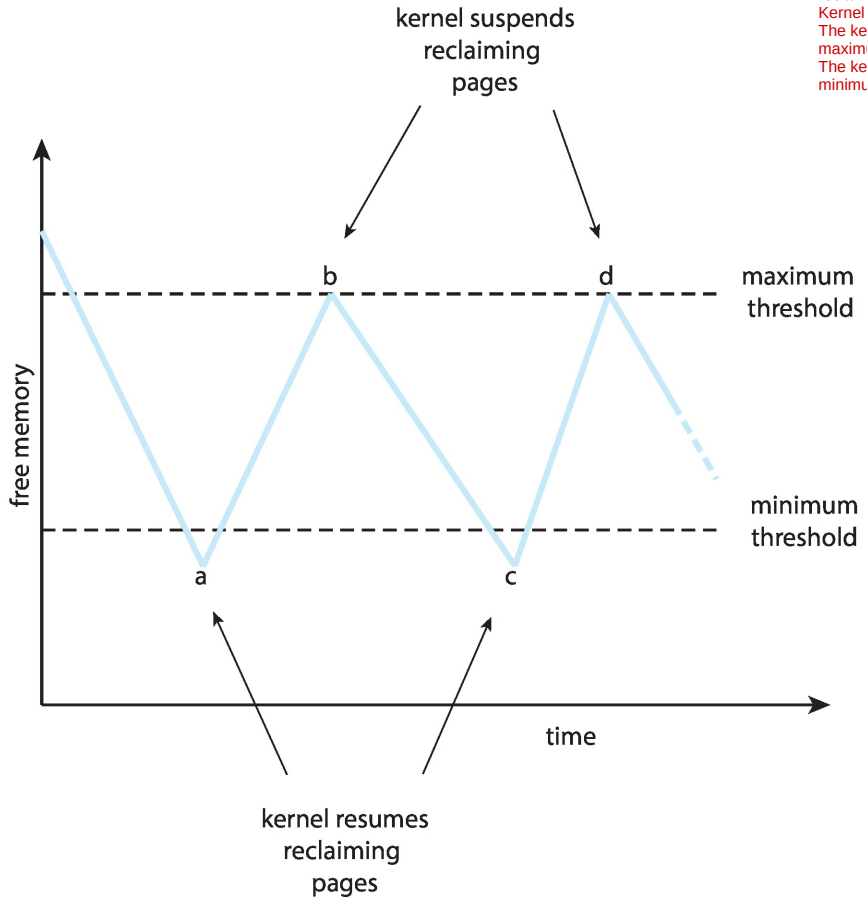
One possibility of global replacement is the following:

Reclaiming Pages:
A global page-replacement policy involves managing the free-frame list effectively. Memory requests are satisfied from the free-frame list, and page replacement is triggered when the list falls below a certain threshold. This strategy ensures there is always sufficient free memory to satisfy new requests.

Reclaiming Pages Example

Free Frame List:
At certain moments, the free frame list becomes empty, triggering the need to reclaim pages.
Kernel Actions:
The kernel suspends reclaiming pages when the free memory reaches the maximum threshold.
The kernel resumes reclaiming pages when the free memory drops to the minimum threshold.

Free Memory Management:
The graph shows how the system manages free memory over time.
The y-axis represents the amount of free memory, and the x-axis represents time.



Thresholds:
Maximum Threshold: The upper limit of free memory. When the free memory reaches this level, the kernel suspends reclaiming pages.
Minimum Threshold: The lower limit of free memory. When the free memory drops to this level, the kernel resumes reclaiming pages.

Process:
Point 'a': The free memory drops to the minimum threshold, triggering the kernel to start reclaiming pages.
Point 'b': The free memory increases to the maximum threshold, causing the kernel to suspend reclaiming pages.
Point 'c': The free memory drops again to the minimum threshold, and the kernel resumes reclaiming pages.
Point 'd': The free memory increases again to the maximum threshold, and the kernel suspends reclaiming pages.

Non-Uniform Memory Access

Uniform Memory Access (UMA):
So far, we have assumed that all memory is accessed equally by all processors.
This means that the speed of access to memory is the same for all processors.

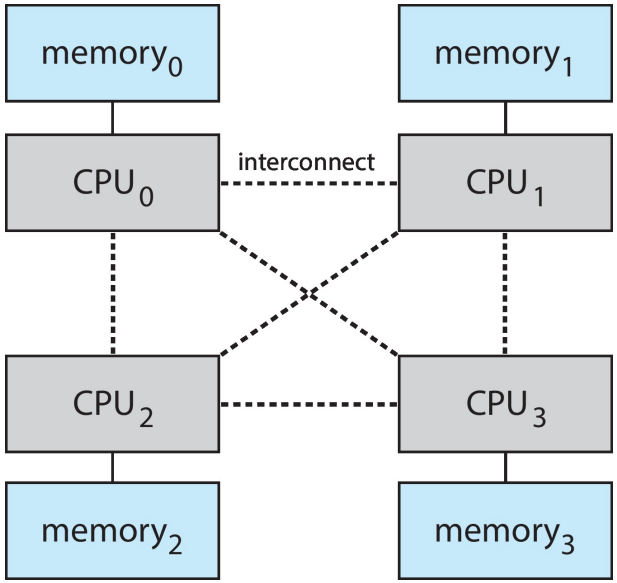
- So far all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus

Non-Uniform Memory Access (NUMA):
In NUMA systems, the speed of access to memory varies depending on the memory's location relative to the processor.
NUMA Architecture:
Consider system boards containing CPUs and memory, interconnected over a system bus.
Each CPU has its own local memory, but it can also access memory connected to other CPUs.
Accessing local memory is faster than accessing memory connected to other CPUs.

- NUMA multiprocessing architecture

One should also be aware of the fact that in multicore platforms we have one shared memory which can rather be seen as a network in which we have a distributed memory. Part of the memory can be considered near and fast to access

The diagram shows a NUMA architecture with four CPUs (CPU0, CPU1, CPU2, CPU3) and their respective local memories (memory0, memory1, memory2, memory3).



cpu0 and cpu1 have local memory thats also accessed by other cpu but with more time.

The CPUs are interconnected, allowing them to access each other's memory, but with varying access speeds.

Shared Memory in Multicore Platforms:
In multicore platforms, there is shared memory that can be seen as a network of distributed memory.
Part of the memory is considered near and fast to access.
Local Memory Access:
CPUs have local memory that is faster to access.
Other CPUs can access this local memory, but it takes more time compared to accessing their own local memory

Non-Uniform Memory Access (Cont.)

- Optimal performance comes from allocating memory "close to" the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by Solaris by creating **lgroups**
    - Structure to track CPU / Memory low latency groups
    - Used my schedule and pager
    - When possible schedule all threads of a process and allocate all memory for that process within the lgroup

Optimal performance in a NUMA (Non-Uniform Memory Access) system comes from allocating memory "close to" the CPU on which the thread is scheduled.
This means that the memory used by a thread should be physically close to the CPU executing that thread to reduce access time.

To achieve this, the scheduler (the part of the operating system that decides which thread runs on which CPU) should be modified to schedule the thread on the same system board whenever possible.
This ensures that the thread's memory is close to the CPU, improving performance.

Solaris Solution: lgroups:
The Solaris operating system addresses this by creating "lgroups" (latency groups).
lgroups:
These are structures that track CPU and memory groups with low latency.
The scheduler and pager (the part of the OS that manages memory) use these lgroups.
When possible, all threads of a process are scheduled within the same lgroup, and all memory for that process is allocated within the lgroup.

Optimal Performance:
Optimal performance is achieved by allocating frames to processes running on the same CPU or close to the memory.
When scheduling a process, it should be placed closer to the memory or the frames should be allocated closer to the CPU.

Optimal performance should be provided if you allocate frames to processes running on 0.??? not sure bout this one.

when we schedule a process should we place it closer to memory or should be allocate the frames closer in the cpu.

Thrashing

Thrashing is a phenomenon that occurs when a process does not have "enough" pages, leading to a very high page-fault rate. A page fault occurs when a process tries to access a page that is not currently in memory, requiring the OS to fetch it from disk.

- 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

Page Fault to Get Page:
The process experiences a page fault and the OS fetches the required page from disk.
Replace Existing Frame:
To make room for the new page, an existing frame in memory is replaced.
Quickly Need Replaced Frame Back:
Soon after, the process needs the replaced frame back, causing another page fault.

The CPU spends most of its time waiting for pages to be fetched from disk, leading to low utilization.

The OS may think it needs to increase the degree of multiprogramming (number of processes running) to improve CPU utilization.

Adding more processes exacerbates the problem, leading to even more thrashing.

Thrashing is a phenomenon in computer operating systems where the system is spending a significant amount of time swapping data between memory (RAM) and virtual memory (disk), due to excessive paging. When a process doesn't have enough pages in memory to execute efficiently, it leads to a high rate of page faults. Page faults occur when the operating system needs to fetch a page from disk into memory because it's not currently present.

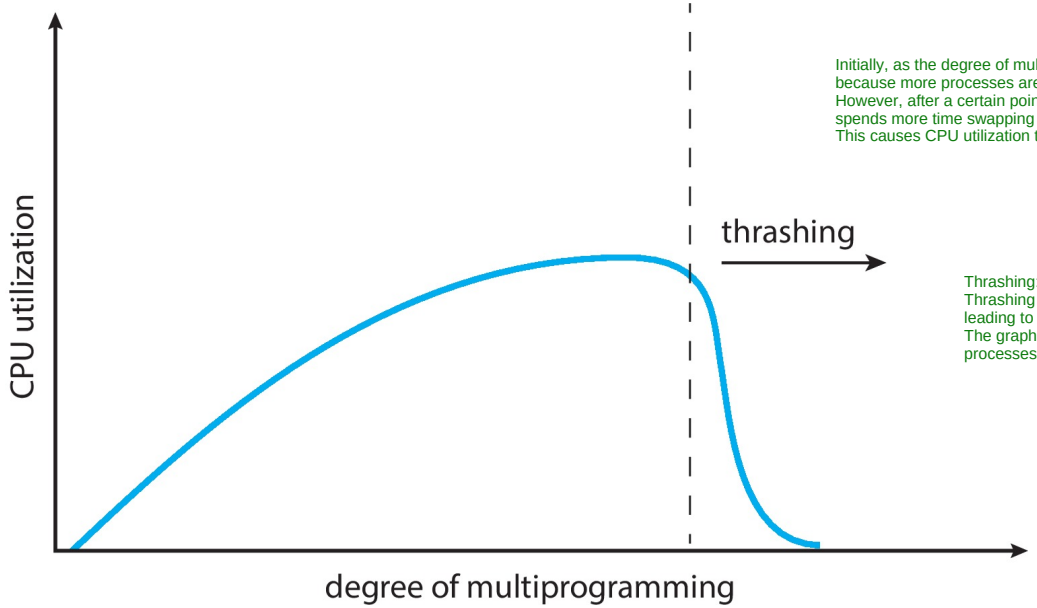
Thrashing occurs when the system spends a significant amount of time swapping data between memory (RAM) and virtual memory (disk) due to excessive paging.
When a process doesn't have enough pages in memory to execute efficiently, it leads to a high rate of page faults.
Page faults occur when the OS needs to fetch a page from disk into memory because it's not currently present.

Thrashing (Cont.)

Thrashing occurs when a process is busy swapping pages in and out of memory, leading to a high page-fault rate.
This happens when the process does not have enough pages to work efficiently.

- **Thrashing**. A process is busy swapping pages in and out

The graph shows the relationship between CPU utilization and the degree of multiprogramming (number of processes running).



The y-axis represents CPU utilization, which is a measure of how effectively the CPU is being used.

Initially, as the degree of multiprogramming increases, CPU utilization also increases because more processes are running and using the CPU.
However, after a certain point, adding more processes leads to thrashing. The system spends more time swapping pages in and out of memory than executing processes.
This causes CPU utilization to drop sharply, as shown by the steep decline in the graph.

Thrashing:
Thrashing occurs when the system is busy swapping pages in and out, leading to a significant drop in CPU utilization.
The graph illustrates how CPU utilization initially increases with more processes but then drops sharply due to thrashing.

The x-axis represents the degree of multiprogramming, which is the number of processes running in the system.

Demand Paging and Thrashing

Locality Model:
Imagine you are working on a project. You don't need all your books and notes at once; you only need a few of them that are relevant to what you're currently doing. This is similar to how a computer program works.
Process Migration:
A program (or process) doesn't need all its data at once. It only needs a small part of it, called a "locality." As the program runs, it moves from one locality to another, just like you might switch from one set of notes to another while working on different parts of your project.
Overlapping Localities:
Sometimes, the data needed for different parts of the program overlap. This means some data is used in multiple localities, just like some notes might be useful for multiple parts of your project.

- Why does demand paging work?

  Locality model

  Demand paging works because of the locality model, which states that processes tend to access a relatively small portion of their address space at any given time.

  - Process migrates from one locality to another

    Processes migrate from one locality (a set of pages that are actively used together) to another.

  - Localities may overlap

    Localities may overlap, meaning that some pages are used in multiple localities.

- Why does thrashing occur?

  Thrashing occurs when the total size of all localities exceeds the total memory size available.
  This means that there are not enough pages in memory to accommodate all the active localities, leading to excessive paging and thrashing.

Why Does Thrashing Occur?:
Total Size of Localities:
Thrashing happens when the total amount of data (localities) that all programs need is more than the computer's memory can hold.
Imagine you have a small desk and too many books. If you keep switching between books, you spend more time organizing your desk than actually studying. This is what happens in a computer when it thrashes.
The computer spends more time moving data in and out of memory (swapping pages) than actually running programs.

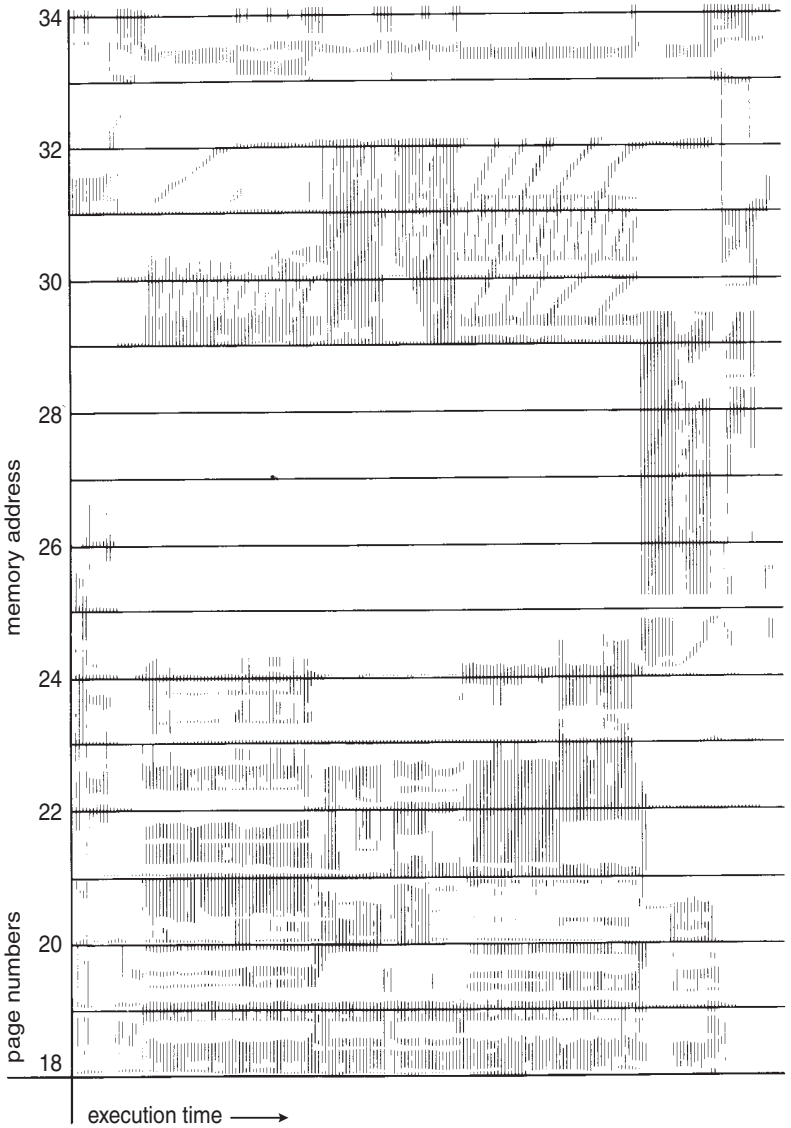
$$\Sigma \text{ size of locality} > \text{total memory size}$$

- Limit effects by using local or priority page replacement

  The effects of thrashing can be limited by using local or priority page replacement strategies.
  Local Page Replacement:
  Each process can only replace pages from its own set of allocated frames, reducing the impact on other processes.
  Priority Page Replacement:
  Pages are replaced based on their priority, ensuring that more important pages are kept in memory.

Locality Model:
The locality model explains why demand paging works. Processes tend to access a small portion of their address space at a time, migrating from one locality to another.
Thrashing:
Thrashing occurs when the total size of all localities exceeds the total memory size, leading to excessive paging.
Limiting Thrashing:
Thrashing can be limited by using local or priority page replacement strategies to manage memory more effectively.

Locality In A Memory-Reference Pattern



Working-Set Model

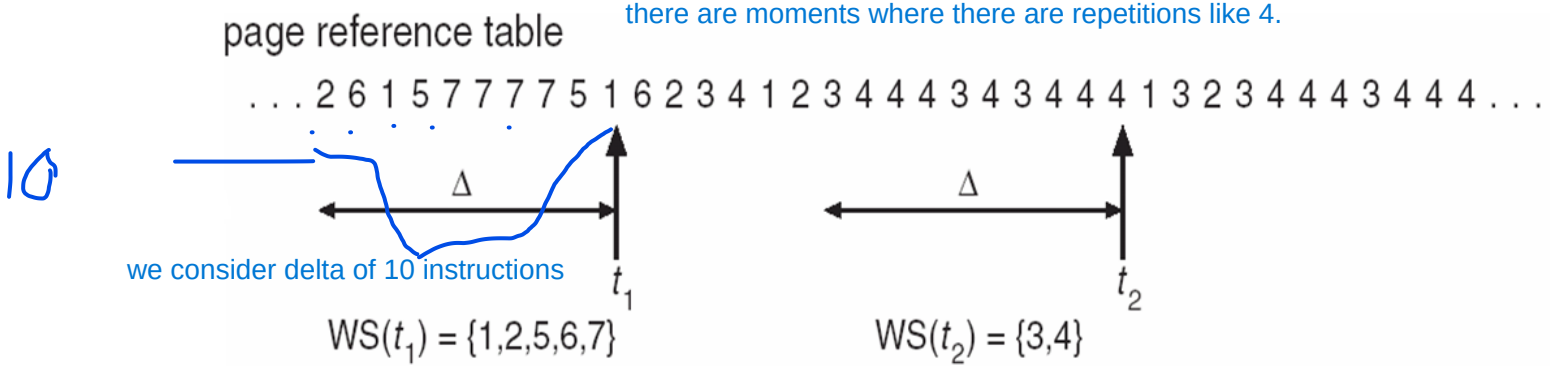
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality

Working set is the set of pages that represent the locality of program. The set of pages where the program is working. How to define this? In some interval time we can say the program was working in that place. A window in time (working set window), over 10000 instructions my working set is been working over those time. working set size is the size of the working set how many pages. 10000 instr how many pages used here? it could be any number upto 10000 (working set size).

Working-Set Model (Cont.)

- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend or swap out one of the processes

there are moments where there are repetitions like 4.



the working set referred to t1 is the amount of pages that were used in the last delta instructions.

The ws size is 5.

the working set model says lets try to keep in memory the pages i am working on. So far we consider the number of frames is fixed. Now consider if we are able to assign a given process 5 frames here maybe its enough to assign 2 frames in the next one.

Keeping Track of the Working Set

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

every 5000-time units you try to remove from memory pages not in the working set. By keeping 2 bits in each page,

Page-Fault Frequency

The Page-Fault Frequency (PFF) approach is more straightforward compared to the Working Set Size (WSS) approach.

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy

The system sets an acceptable rate of page faults and uses a local replacement policy to manage memory.

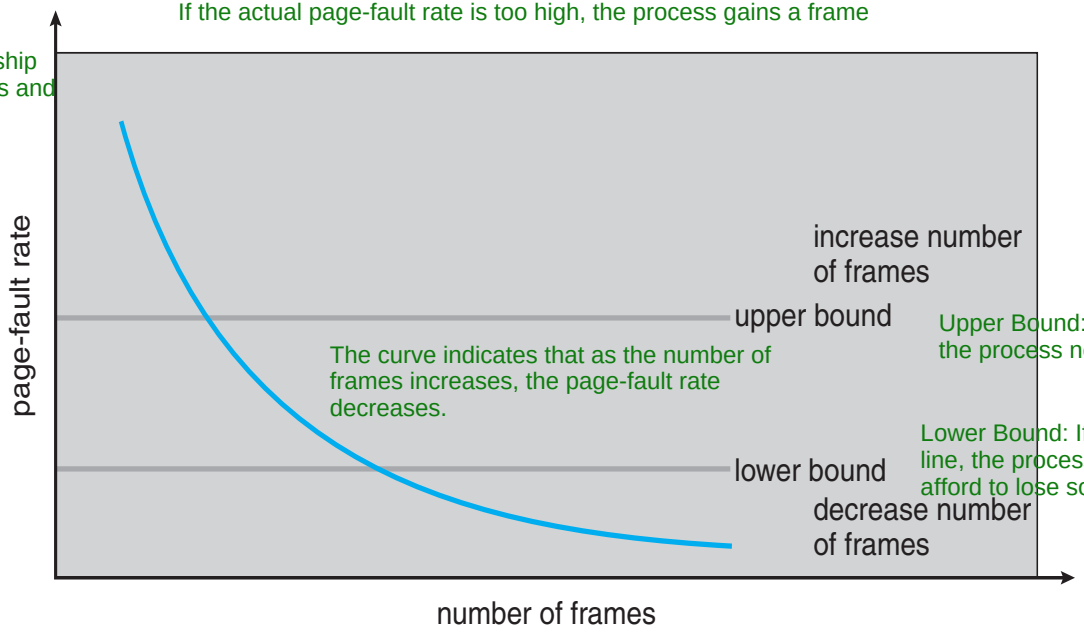
instead of trying to predict observe the result.

- If actual rate too low, process loses frame
- If actual rate too high, process gains frame

actual rate is below threshhold it means we have too many frames so we can remove frames and if its too high we need to gain frames

If the actual page-fault rate is too low, the process loses a frame.
If the actual page-fault rate is too high, the process gains a frame

The graph shows the relationship between the number of frames and the page-fault rate.



The curve indicates that as the number of frames increases, the page-fault rate decreases.

Upper Bound: If the page-fault rate is above this line, the process needs more frames.

Lower Bound: If the page-fault rate is below this line, the process has too many frames and can afford to lose some.

Instead of predicting, the system observes the actual page-fault rate.
If the actual rate is below the threshold, it means there are too many frames, so some can be removed.
If the rate is too high, more frames are needed.

Page Fault Frequency Algorithm

The algorithm is activated at every page fault, not at every page reference.

- Activated at every page fault, not at page reference.

The algorithm uses the time interval (T) from the previous page fault to determine actions.

- Based on time interval τ from previous page fault.

If T<c (where is a constant), the page fault frequency is higher than desired. In this case, a new frame is added to the Resident Set of the page-faulting process.

- if $\tau < c$, i.e. page fault frequency greater than desired, add a new frame to the Resident Set of the page faulting process.

T>=c, the page fault frequency is acceptable. In this case, frames with a reference bit of 0 are removed from the Resident Set, and the reference bit of all other pages in the Resident Set is cleared (set to 0).

- if $\tau \geq c$, i.e. page fault frequancy OK, remove from Resident Set of page faulting process all pages with reference bit at 0;

then clear (set 0) reference bit of all other pages in Resident Set.

This algorithm will also be used in exam. Lets measure the distance between two faults the last and the present page fault. Tou (T) is less than a constant c it means two page faults are closer than a threshold whihc means frequency high. if page frequency hihger distance is more.

if T>=c it means im happy because page fault freq is smaller or equal to constant so i can release frames. if freq is okay i will remove from the resident sent all frames that i can consider out of the working set. How? if they have ref bit 0 it means they have not been used recently so not in working set. All pages that i keep whihc i have reference bit 0 i clear them.

The Page-Fault Frequency approach dynamically adjusts the number of frames allocated to a process based on the observed page-fault rate.
The algorithm ensures that the system maintains an acceptable page-fault rate by adding or removing frames as needed.
The time interval T between page faults is a critical factor in determining whether to add or remove frames.
This detailed explanation should help you understand the concepts and the diagrams presented in the slides.

Page Fault Frequency

access with no page fault

Tempo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Stringa	**6**	**4**	**4**	**3**	**2**	**4**	**4**	**4**	**1**	**3**	**3**	**2**	**4**	**4**	**5**	**1**	**2**	**2**	**2**	**6**	**1**	**2**	**5**	**4**
Frame 0	6	6	6	6	6	6	6	6	6	6	6	6	6	6	5	5	5	5	5	6	6	6	6	6
Frame 1	-	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	2	2	2	2	2
Frame 2	-	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	1	1	1	1	1
Frame 3	-	-	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	-	-	-	5	5
Frame 4	-	-	-	-	-	-	-	-	1	1	1	1	1	1	-	1	1	1	1	-	-	-	-	4
Fault	*	*		*	*				*						*	*				*			*	*
Ref Bit									0						0					0				

close together so high freq

simulation can be seen here.

C=3

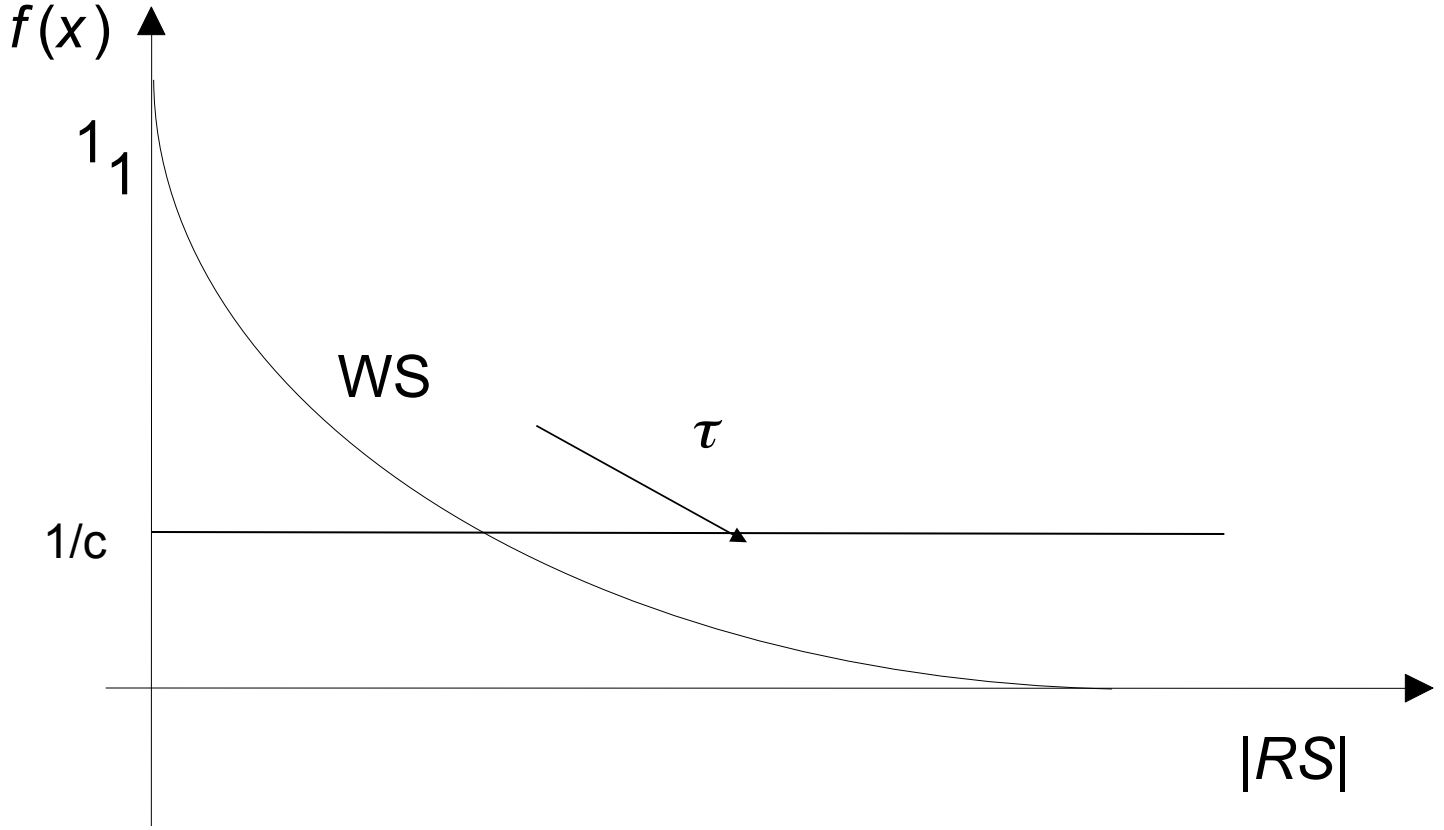
Page referenced

victim

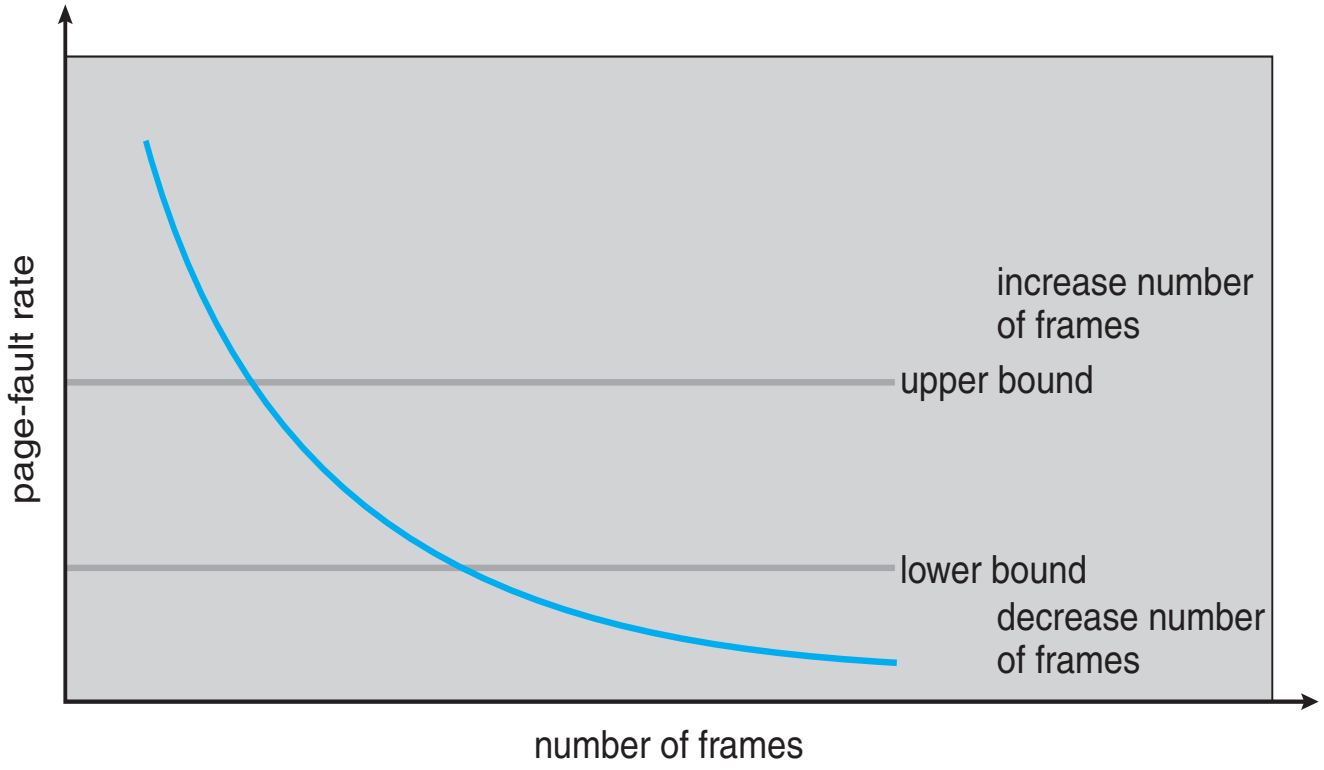
far enough from previous page fault so we activate the algorithm.

the victim are removed from resident set

Page fault frequency function



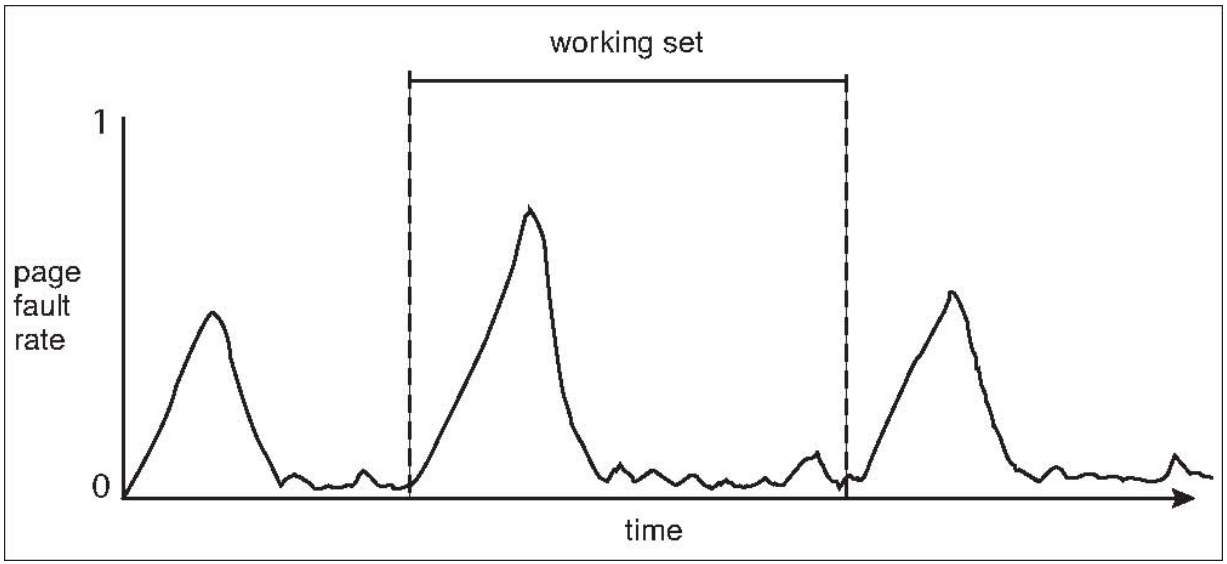
Page Fault Frequency with two thresholds



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

You have increasing time that means new pages have been introduced. Page faults are increasing or decreasing. Delta should be close to this interval. If you take a delta smaller than this interval you will prolly trigger more page faults.



Allocating Kernel Memory

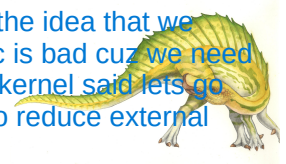
- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O

at the end of the chapter we are saying the kernel is palying another game, the kernel is not necessarrily using the memory simply becasue the kernel has some requirement on contiguous allocation of memory! We will see in os161 the microprocessor has diff partition of memory, for the kernel we will see part of memory is contiguous allocation.
Kernel has other requirements. In most os the kernel will have a set of available frames or memory partition distinct from the user partition. Kernel is needing contiguous allocation. Suppose DMA controllers and other simple devices.

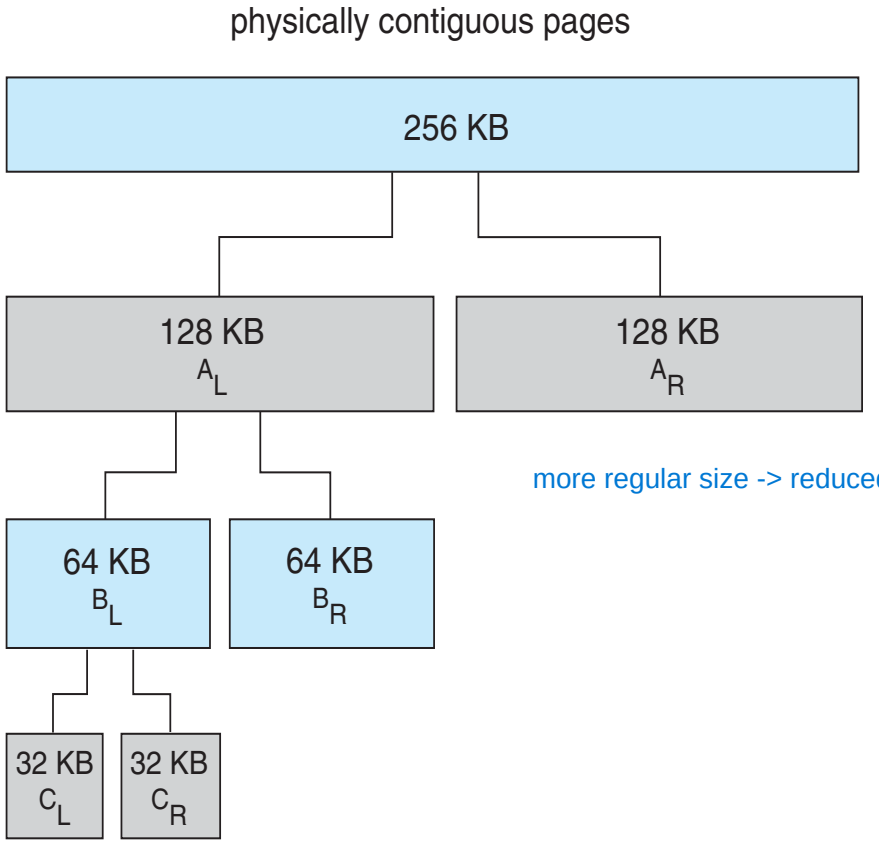
Buddy System

- 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 appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation

We see two strategies, the buddy system is based on the idea that we would like contiguous alloc for kernel. Contiguous alloc is bad cuz we need compaction which is expensive and we don't like. The kernel said lets go contiguous but pay something internel fragmentation to reduce external fragmention.



Buddy System Allocator

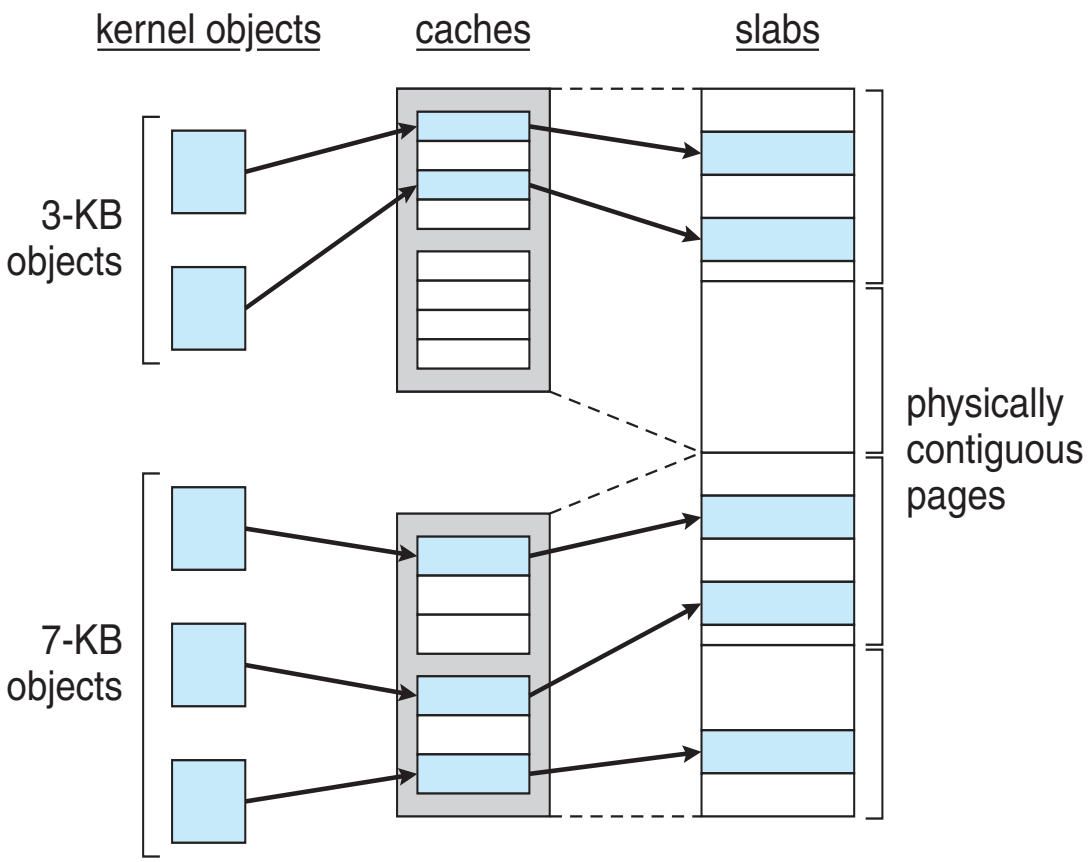


more regular size -> reduced external fragmentation.

Slab Allocator

- Alternate strategy
- **Slab** is 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 data structure
- When cache created, filled with objects marked as **`free`**
- When structures stored, objects marked as **`used`**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Slab is one or more physicall contiguous pages, but multiple pages can be not enough or flexible in terms of potential fragmention so you group slabs in caches and you play alloc internally in caches. You can see how k malloc is implemented in os161. Each cache for a particular kernel obj.

Slab Allocator in Linux

- For example process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty

Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited memory
    - Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

Other Considerations

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

We have considered how use memory is alloc, paging and demand paging.
What can we do to gurantee good performance? We consider few things:
A small variant to demang paging with replacement scheme 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 *α* of the pages is used
  - Is cost of ***s * α*** save pages faults > or < than the cost of prepaging
    s * (1- α) unnecessary pages*?*
  - ***α*** near zero ⇒ prepaging loses

In order to decrease the num of page faults, try to find a good victim when you need a victim. If you know which is the policy, choosing a good victim is to reducing the num of page faults. Another possiblily to reduce the num of page faults when u have time to do it is bringing something in memory which is not in the memory but will be used in the future. Bing able to guess a page not in the memory but will be used in future bring in advance can avoid the next page fault. If prepaging done when I am already playing with a page fault and while bringing page in a memory i bring for eg page b! I speculate when bringing this page in i say okay the next one might be needed as well so this the cost in data transfer is reduced. Instead of starting with zero paging let's guess the initial pages that's needed to start. Is this worse than the cost of what i have been doing. Suppose s pages are prepaged. 100 pages have been prepaged for eg. what percentage of those pages was actually beneficial? Lets say a the ratio of pages effectively used. It means all pages were effectively needed. 1-a is the num or ratio of pages bought in.

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

Can we adjust page size for best performance? Fragmentation means that if we want to minizmize internal fragmentation prolly small pages are better. The only fragmented pages of a given page is the last case. Average case 50% . Small page means. More entry in page table -> larger pages.

I/O overhead will generally prefer larger pages cuz once you pay a certain overhead in order to startup and activate its better to have more data to transfer.

Num o fpage faults, in principle you can say suppose you have large pages, one process stays in a given large page. In principle you can say if pages are very large i am reducing the um of page faults the side effect is onhe big page is prolly going to bring in memory. In a given large page you will prolly have both instruction and data that are being used and ones not being used. Eg one only page in a given process you are only using 30 percent and 70 percent not used. Smaller pages tend to only contain used things

Locality similar reasoning as page faults. If page size larger than locality it means you are including things in mem not needed. Smaller in locality means better cuz locality will be covered by some pages.

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

Tlb the num of memory addressees covered by the tlb is given by the num of entries in tlb multiplied by how many pages is a given page covering. Let's say a tlb is not adjustable! Let's say tlb is a part of the microprocessor.

It is clear that if you have 100 entries in tlb, smaller page tables means tlb is covering smaller num of addresses. Most microprocessors especially if the designer or vendor is tryna cover diff platforms.

Program Structure

- Program structure
  - `int[128,128] data;` matrix 128x128
  - 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

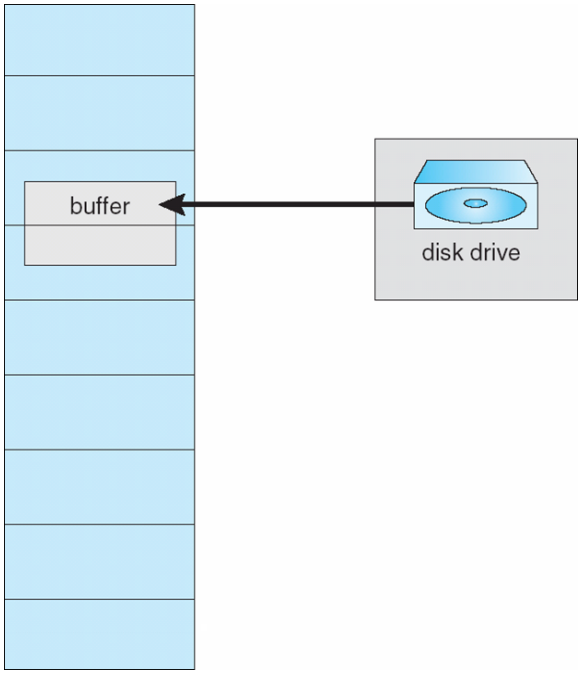
```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults

Each row is contained in one page, or each row is spanning over three or four pages. If you want to traverse all the matrix better to go by rows. Suppose if one row is one page. It is clear you trigger more page faults if you traverse by columns. So hence should be by rows.

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



The problem of I/O we have talked about in two models. If you have seen two solutions do not move that process. Use kernel buffers.
Consider when talking about swapping in the framework of both contiguous allo and paging. In order to make room for other process to start. swapping could be critical if process you trying to move is in IO. But swapping involed entire process but here we say even with paging when you find a victim there is possible that victime can be locked in io so we can't move back in store.

Operating System Examples

- Windows
- Solaris

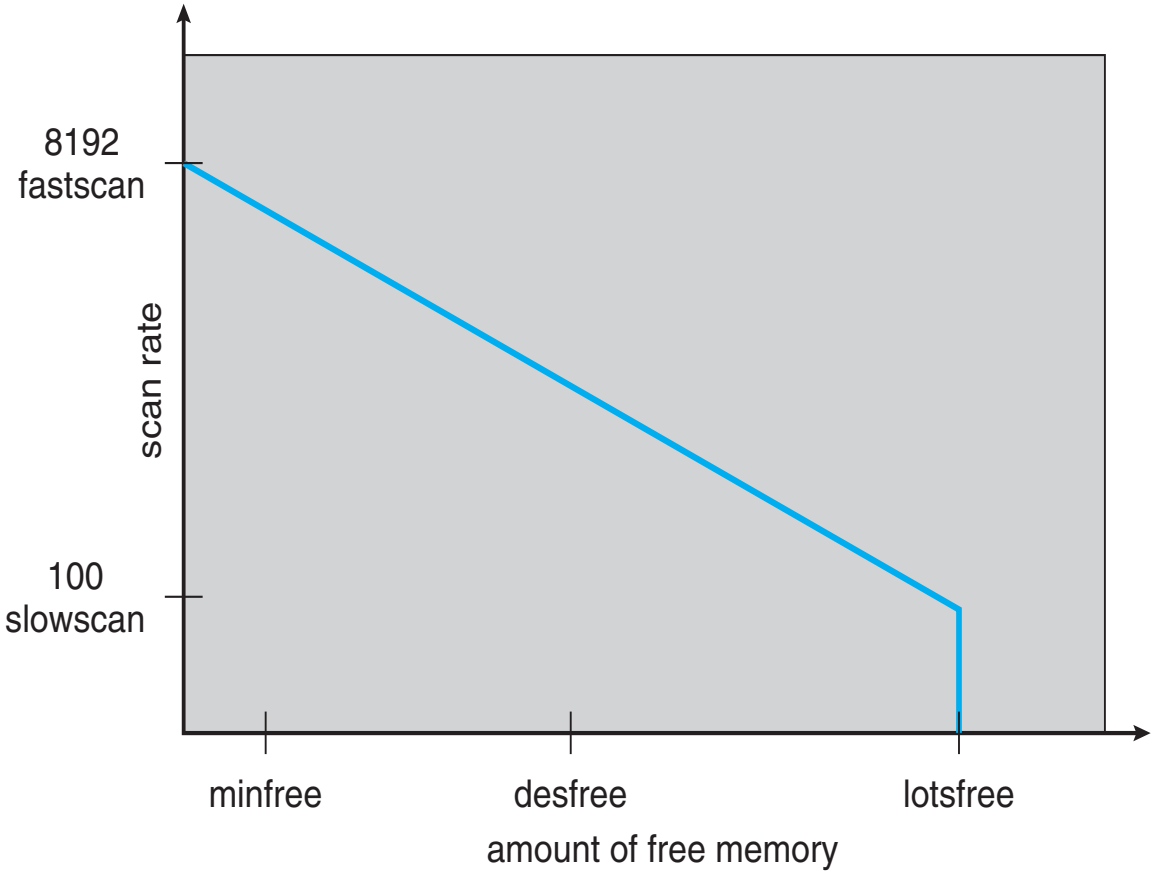
Windows

- 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

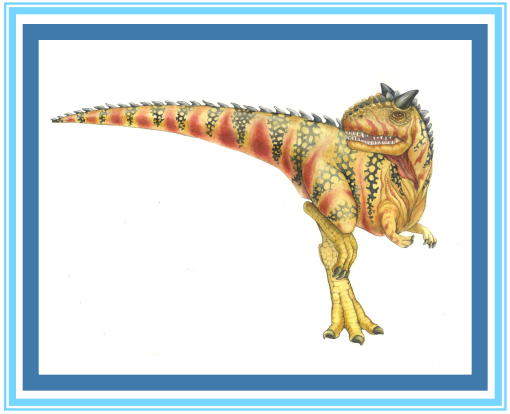
Solaris

- 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



End of Chapter 10

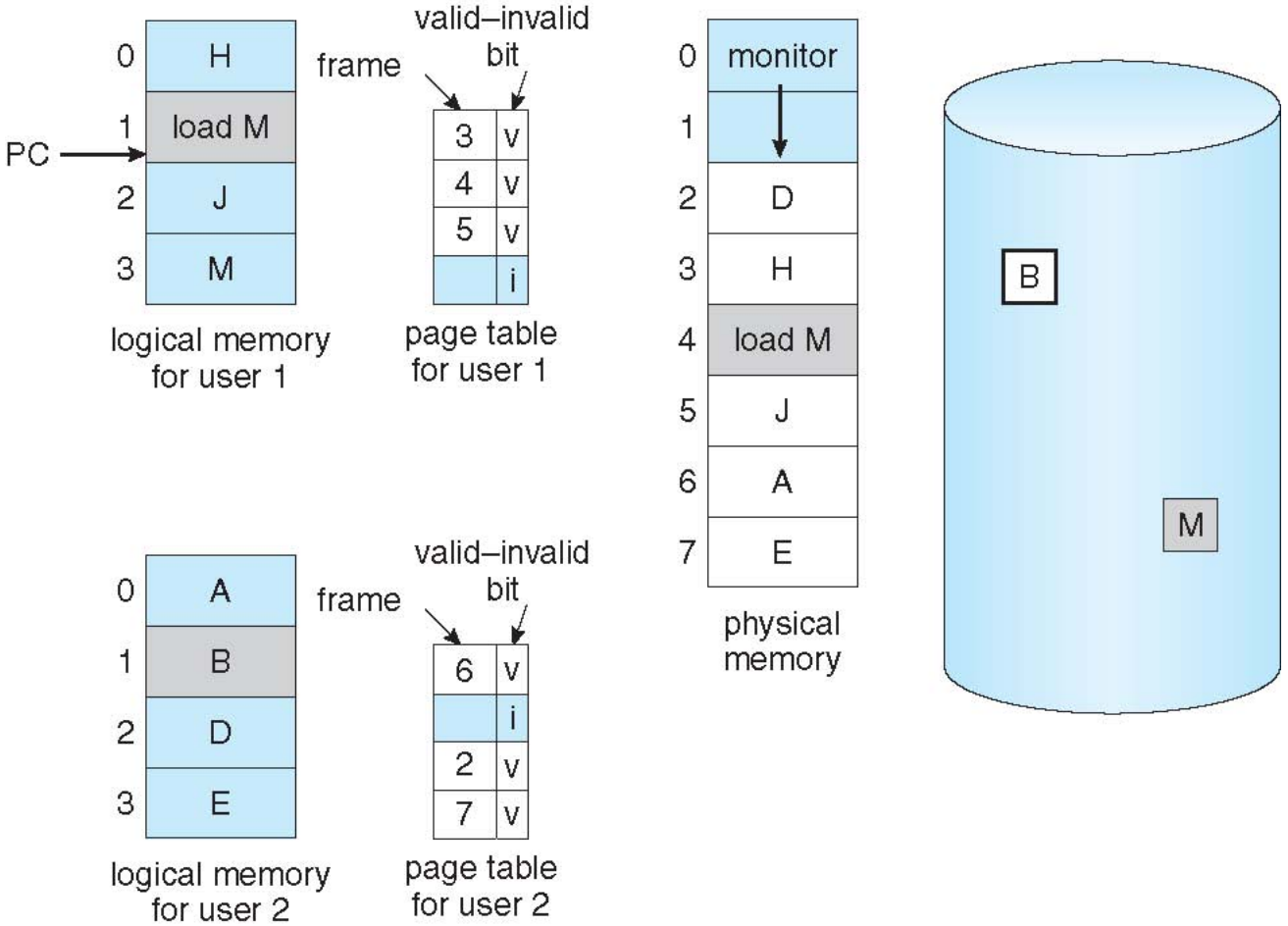


Performance of Demand Paging

- 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
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

Need For Page Replacement

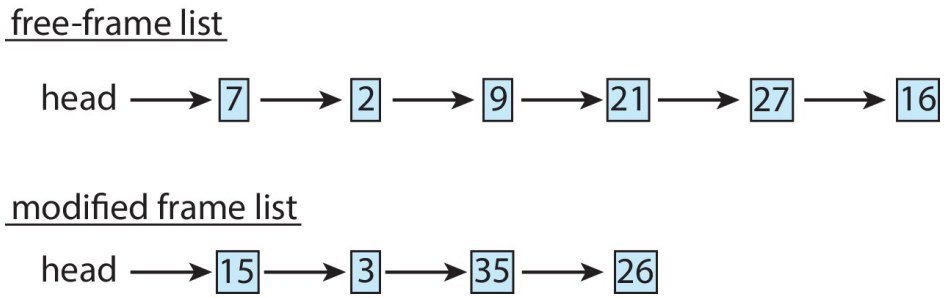


Priority Allocation

- 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

Memory Compression

- **Memory compression** -- rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.
- Consider the following free-frame-list consisting of 6 frames



- Assume that this number of free frames falls below a certain threshold that triggers page replacement. The replacement algorithm (say, an LRU approximation algorithm) selects four frames -- 15, 3, 35, and 26 to place on the free-frame list. It first places these frames on a modified-frame list. Typically, the modified-frame list would next be written to swap space, making the frames available to the free-frame list. An alternative strategy is to compress a number of frames{\mdash}say, three{\mdash}and store their compressed versions n a single page frame.

Memory Compression (Cont.)

- An alternative to paging is **memory compression**.
- Rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.