Unit 4

Main Memory Management

1. Background

Memory is central to the operation of a modern computer system. Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.
A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore ***how*** a program generates a memory address.

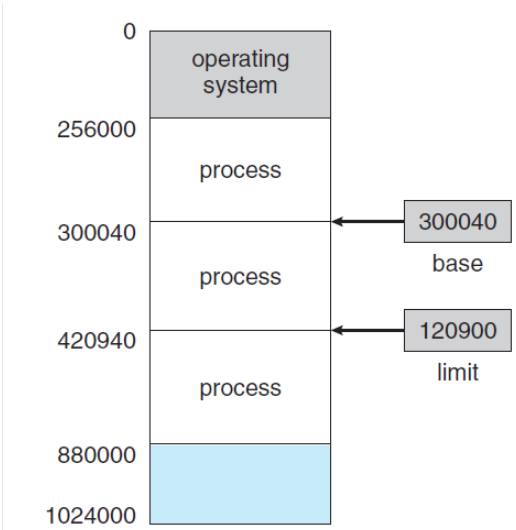
1) Basic Hardware

Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.
Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Accessing data from main memory can take many CPU clock cycles, causing the processor to pause or "stall" because it doesn't have the data it needs to continue executing instructions. This delay happens often since memory access is frequent, making it a major issue.
To solve this, fast memory called a cache is added between the CPU and the main memory. This cache is usually placed on the CPU chip itself, allowing much quicker access to frequently used data and reducing the need for the processor to wait.

To ensure the system runs smoothly, the operating system must be protected from unauthorized access by user processes. On multiuser systems, it's also important to prevent user processes from interfering with each other. This protection must be handled by hardware because relying on the operating system for every memory access would slow the system down.
A simple way to implement this protection is to give each process its own memory space. This separation ensures that processes don't interfere with one another and allows multiple processes to run simultaneously.
To enforce this separation, the hardware uses two registers:

1. **Base Register**: Stores the starting address of the process's legal memory area.
2. **Limit Register**: Specifies the size of the allowed memory range.

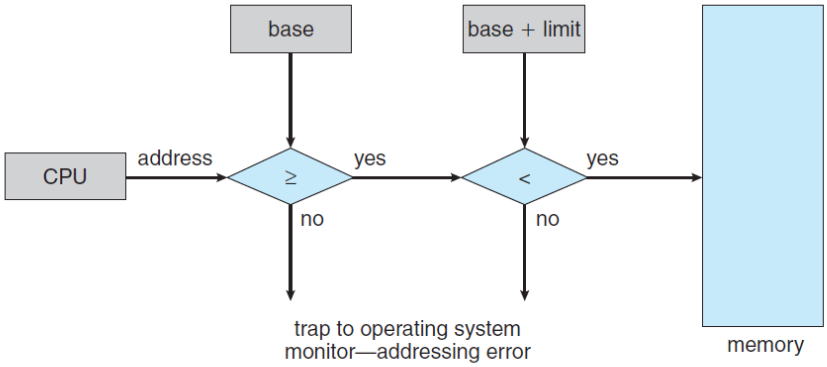
For example, if the base register holds **300040** and the limit register specifies a size of **120900**, the process can only access memory addresses from **300040 to 420939**. Any attempt to access addresses outside this range will be blocked.



A base and a limit register define a logical address space.

To protect memory, the CPU hardware checks every memory address generated in **user mode** against the base and limit registers. If a program in user mode tries to access memory outside its allowed range (like operating system memory or another user's memory), the hardware triggers a **trap** to the operating system. This is treated as a critical error and stops the program.

The **base** and **limit** registers can only be updated by the operating system. This is done using a **special privileged instruction** that can only run in **kernel mode**. Since only the operating system operates in kernel mode, only it has the authority to change these registers. This ensures user programs cannot alter them and bypass memory protection.



Hardware address protection with base and limit registers.

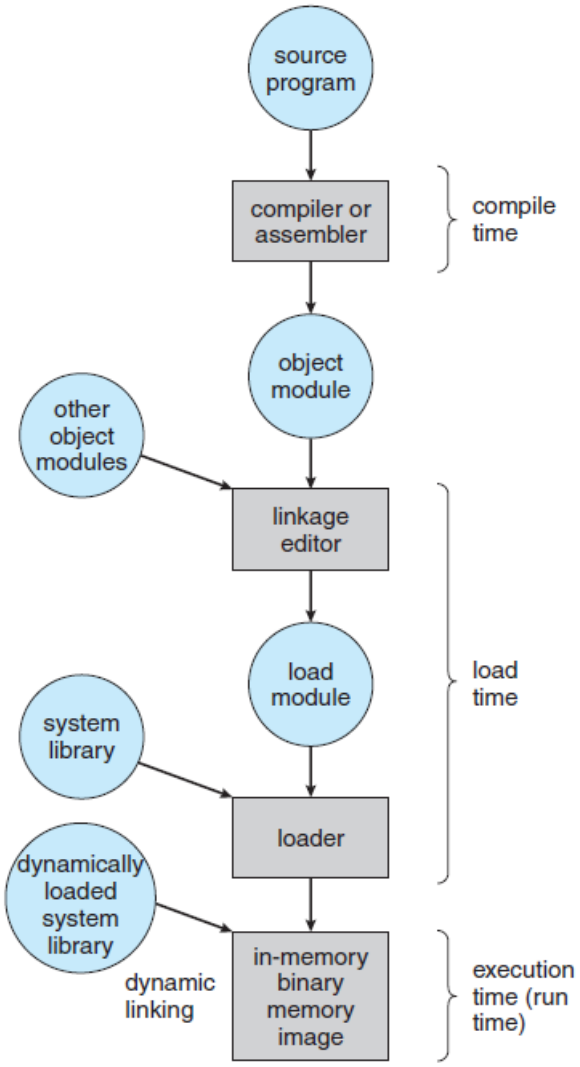
This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents. The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services.

2) Address Binding

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the **input queue**. The normal single-tasking procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available. In most cases, a user program goes through several steps—some of which may be optional—before being executed (below Figure). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as the variable count). A compiler typically **binds** these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader in turn binds the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

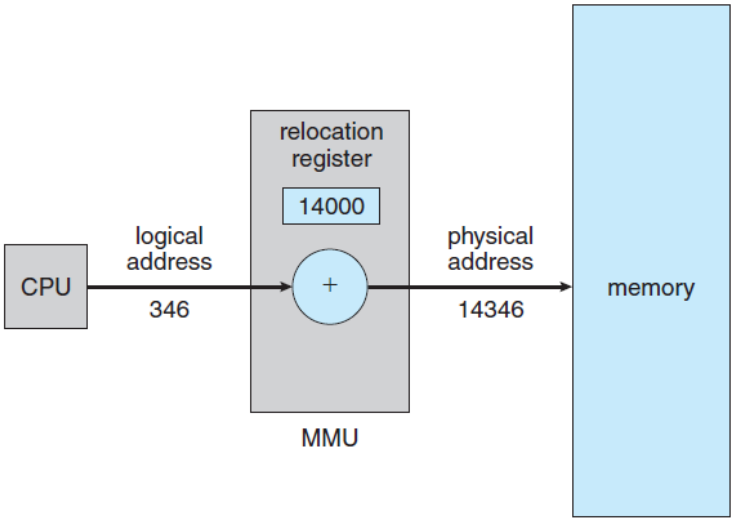
- **Compile Time**:
  - If the process's memory location is known during compilation, the compiler creates **absolute code** starting at that location.
  - If the location changes later, the code must be **recompiled**.
  - Example: MS-DOS .COM programs are bound at compile time.
- **Load Time**:
  - If the memory location isn't known at compile time, the compiler generates **relocatable code**.
  - The final memory location is determined when the program is **loaded** into memory.
  - If the location changes, the program just needs to be **reloaded**.
- **Execution Time**:
  - If a process may move between memory segments during execution, binding is done at **runtime**.
  - This requires **special hardware** to handle dynamic address translation.
  - Most general-purpose operating systems use this approach.



Multistep processing of a user program.

3) Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit—that is, the one loaded into the **memory-address register** of the memory—is commonly referred to as a **physical address**.



Dynamic relocation using a relocation register.

Aspect	Logical Address Space	Physical Address Space
Definition	Set of addresses generated by the CPU for a program.	Set of actual addresses in physical memory (RAM).
Visibility	Seen by the user or program.	Hidden from the user; managed by the OS and hardware.
Binding	Determined during execution (runtime binding).	Determined by the memory management unit (MMU).
Range	Starts from 0 to maximum logical address (per process).	Depends on the size of the installed physical memory.
Purpose	Used by the program to reference memory indirectly.	Used by hardware to access actual memory locations.
Translation	Translated to physical addresses by the MMU.	Directly accessed by the hardware.
Independence	Independent of the physical memory size.	Limited by the physical memory size.
Example	A pointer in the program referring to memory.	The actual RAM location where the pointer points.

Example of Logical and Physical Address:

Imagine a program is running, and the operating system is managing its memory. The program uses **logical addresses** (relative to its own address space), and the MMU translates these to **physical addresses** in the main memory.

Example Details:

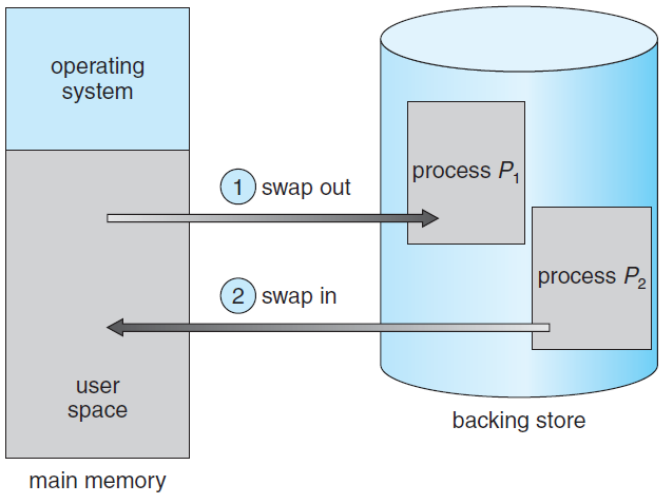
1. **Base Address in Physical Memory**: The program is loaded at physical address **3000** in main memory.
2. **Logical Address**: The program accesses **logical address 200**.
3. **Translation**:
   - The MMU adds the base address **3000** to the logical address **200**.
   - Physical address = Base address + Logical address = **3000 + 200 = 3200**.
4. **Access**:
   - The MMU translates the logical address **200** into the physical address **3200** and accesses the required memory.

This approach ensures the program can operate in its own address space without worrying about the actual physical location in memory.

Feature	Logical Address	Physical Address
Generated By	CPU (Program)	Memory (RAM or Cache)
Visibility	Visible to the program and OS	Invisible to the program
Translation	Translated to physical address by MMU	Direct memory location in hardware
Memory Type	Virtual Memory	Physical Memory (RAM)
Scope	Virtual address space	Actual memory hardware location
Usage	Program accesses data	Hardware accesses data

2. Swapping

A process must be in memory to be executed. A process, however, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution (below Figure). Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.



Swapping of two processes using a disk as a backing store.

Swapping is a memory management technique that allows the operating system to move processes between **main memory (RAM)** and **secondary storage (usually a disk)**. This process enables the system to **simulate a larger amount of memory** than is physically available, which can **increase the degree of multiprogramming** (the number of processes that can be loaded and executed concurrently).

- When a process is not actively using the CPU, it can be swapped out (written to disk) to free up space in physical memory.
- When a process needs to run again (for example, when it is scheduled by the CPU), it can be swapped back into memory from the disk.

Example:

- Suppose a system has only 4 GB of RAM, but there are 10 processes that need to run, and each process requires 1 GB of memory. Without swapping, only 4 processes could be loaded into memory at once.
- With swapping, the operating system can load some processes into memory and swap others out when needed. For example, while Process 1 is in memory, Process 2 could be swapped out to disk. When Process 1 is finished or not in use, Process 2 can be

swapped back in, allowing the system to run more processes concurrently than would be possible with physical RAM alone.

Swapping allows a system to use **secondary storage (like a disk)** as **virtual memory**, enabling processes to run even when the physical RAM is insufficient. By **moving processes in and out of memory**, swapping **increases the degree of multiprogramming**, allowing more processes to be executed concurrently than the system's physical memory would otherwise permit. However, the trade-off is that frequent swapping can reduce performance due to slower access speeds of secondary storage.

i) Standard Swapping

Standard swapping involves moving processes between main memory and a backing store. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a **ready queue** consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process. The context-switch time in such a swapping system is fairly high.

To calculate the **total context-switch time** in a swapping system, we need to break it down into two main parts: the time taken to **swap out** a process from **main memory** to the **backing store** (disk) and the time taken to **swap in** the process from the backing store back into memory.

Let's go through the calculation step by step:

Given Information:

- **Process size** = 100 MB
- **Backing store (disk) transfer rate** = 50 MB per second
- **Swap time (overhead)** = 200 milliseconds (this might include time to update data structures, manage the swap, etc.)

1. Time to transfer process to or from disk (transfer time):

The time to transfer the entire process (100 MB) to or from the disk is calculated as follows:

$$\text{Transfer time} = \frac{\text{Process size}}{\text{Transfer rate}}$$

$$\text{Transfer time} = \frac{100\ \text{MB}}{50\ \text{MB/second}} = 2\ \text{seconds}$$

2. Total time for a swap (swap out + swap in):

Since both the swap out (moving the process from memory to disk) and swap in (moving the process back from disk to memory) must be performed during a context switch, we double the transfer time:

$$\text{Total transfer time} = 2 \text{ seconds} \times 2 = 4 \text{ seconds}$$

3. Adding the swap overhead:

The swap overhead includes the time spent on operations like updating page tables, saving the state of the process, and loading it back. This time is given as 200 milliseconds, or 0.2 seconds.

$$\text{Total swap time} = \text{Total transfer time} + \text{Swap overhead}$$

$$\text{Total swap time} = 4 \text{ seconds} + 0.2 \text{ seconds} = 4.2 \text{ seconds}$$

Conclusion:

The total **context-switch time** for swapping in and out a 100 MB process is approximately **4.2 seconds**.

The time it takes to swap a process largely depends on how much memory needs to be transferred. For example, if a computer system has 4 GB of memory and the operating system uses 1 GB, then the maximum memory available for a user process is 3 GB. However, many user processes might be much smaller than that—such as 100 MB. Swapping out a 100 MB process would only take around 2 seconds, compared to 60 seconds for a full 3 GB process. Therefore, it's useful to know the exact memory a process is using, rather than its maximum possible use, so we can swap out only the memory actually in use and save time. For this to work, processes with changing memory needs would need to notify the operating system when they need more or less memory using system calls (e.g., request memory() and release memory()).

Swapping also has other constraints. For example, a process should be completely idle before being swapped. If a process has pending I/O (input/output) operations, it can't be swapped out immediately because it may still be accessing its memory for I/O operations. If the I/O operation is delayed because the device is busy, and we swap out that process, it could cause issues if a new process is swapped into that memory space.

There are two main ways to handle this problem:

1. Avoid swapping out processes that have pending I/O.
2. Perform I/O only through the operating system's own buffers. The operating system ensures that processes perform I/O operations only through the operating system's own **buffers**. These buffers are temporary memory areas controlled by the OS where data is stored while being transferred between a process's memory and I/O devices (e.g., hard disk, network).

The second approach, though, adds extra overhead since it involves copying data twice—from the OS buffer to the user process memory—before the process can use it.

Isolation of I/O from Process Memory:

- Normally, if a process performs I/O directly with its own memory, that memory must remain accessible until the I/O operation completes. If the process is swapped out while the I/O is still pending, the I/O device might lose access to the required memory, leading to failures or data corruption.
- By performing I/O through **OS buffers**, the data involved in the I/O operation is transferred to a memory area controlled by the OS. This buffer remains in physical memory regardless of whether the process itself is swapped out.

2. Independent I/O Handling:

- Once the data is moved to the OS buffer, the I/O device interacts only with this buffer, not the process's memory.
- If the process is swapped out, the OS can continue managing the I/O operation independently, ensuring that the data flow to or from the device is not disrupted.

3. Contiguous memory allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well. In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process.

An **interrupt vector** is a data structure used by the operating system to handle and manage **interrupts**—signals sent to the CPU to indicate an event requiring immediate attention. The interrupt vector contains pointers to **interrupt service routines (ISRs)**, which are specific routines designed to handle different types of interrupts.

The terms **low memory** and **high memory** refer to specific regions of a computer's main memory (RAM), typically determined by their starting addresses. Here's the distinction:

Low Memory:

- **Definition**: The memory region that starts from the lowest address (address 0) and extends upward.
- **Usage**: Historically, this area is often reserved for the **operating system kernel**, interrupt vectors, or other essential system functions because these components need to be accessible at predictable locations during system startup.
- **Example**: In many systems, the operating system is loaded into low memory to ensure it can manage hardware and software efficiently.

High Memory:

- **Definition**: The memory region that starts from higher addresses and extends downward toward low memory.

- **Usage**: This area is typically allocated for **user processes** or additional system data. Placing the operating system in high memory leaves more room in low memory for user programs, which might be necessary for certain system architectures.
- **Example**: On systems that prefer user processes to reside in low memory for faster access, the operating system may be placed in high memory.

Choice of Low or High Memory for the OS:

- **Low Memory**: Ensures simplicity and direct access during the boot process and hardware interrupts.
- **High Memory**: Leaves the lower, faster memory regions available for user programs, which might improve performance on systems where user program speed is critical.

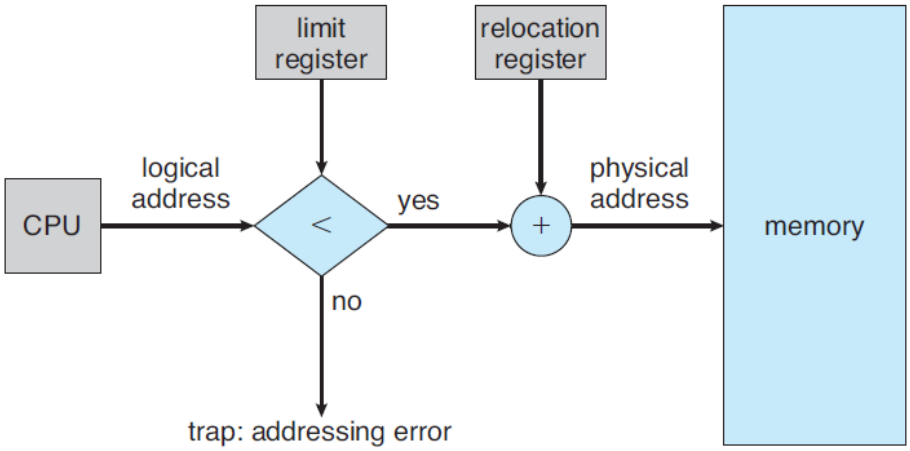
The choice depends on the hardware architecture and design goals of the operating system.

i) Memory Protection

We can prevent a process from accessing memory it doesn't own by using a **relocation register** and a **limit register**. The relocation register stores the starting physical address of the process, while the limit register defines the maximum range of addresses the process can use. For instance, if the relocation register holds 100040 and the limit register holds 74600, then the process can access addresses only within this range. The **Memory Management Unit (MMU)** uses these values to map logical addresses to physical addresses, ensuring each address falls within allowed limits before sending it to memory.

When the CPU scheduler chooses a process to run, it loads the relocation and limit registers with the process's values during a context switch. This way, every address generated by the process is checked against these registers, which protects the operating system and other processes' memory from being accidentally accessed or modified.

The relocation register setup also makes it easy for the operating system's size to change as needed. For example, the operating system has code and memory space for device drivers. If a device driver or other OS service isn't used frequently, its code doesn't need to stay in memory. This is called **transient code**, meaning it's loaded only when needed, allowing the OS to adjust its size dynamically to free up memory for other uses.



Hardware support for relocation and limit registers.

The **base address** in the context of memory management is essentially the same as the value stored in the **relocation register**. It represents the starting physical address where a process's memory segment resides in physical memory. Here's how the terms relate and their usage:

Base Address vs. Relocation Register

- **Base Address**: Refers conceptually to the starting address of a process's memory segment in physical memory. It defines where the process's memory begins.
- **Relocation Register**: A hardware register that holds the base address. It is used during address translation to map logical addresses to physical addresses at runtime.

Role of the Base Address

When a process generates a **logical address** (e.g., `200`), this address is **offset** within the process's memory space. To compute the corresponding **physical address**, the system adds the base address (from the relocation register) to the logical address:

$$\text{Physical Address} = \text{Base Address (Relocation Register)} + \text{Logical Address}$$

For example:

- If the **relocation register** (base address) contains `1000` and the process generates a logical address of `200`, the physical address becomes `1000 + 200 = 1200`.

Protection with the Limit Register

The **limit register** is used to define the range of memory the process can access. It prevents a process from accessing memory beyond its allocated range. For every memory access, the hardware checks:

1. If the **logical address** is less than the value in the limit register.
2. If true, the physical address is computed by adding the base address (relocation register) to the logical address.
3. If false, the system generates a **trap** (error), as the process is attempting to access unauthorized memory.

ii) Memory Allocation

One simple way to allocate memory is by dividing it into several fixed-sized partitions, with each partition holding exactly one process. This limits the number of processes in memory to the number of partitions available. When a partition is empty, a process from the input queue is loaded into it. Once the process finishes, the partition becomes free for another process. This fixed-partition method was used in older systems like the IBM OS/360 (known as MFT) but is no longer widely used.

A more flexible method, called the variable-partition scheme (or MVT-Multiprogramming with a Variable number of Tasks), allows memory to be divided into different-sized blocks. The operating system keeps track of which parts of memory are free and which are occupied. Initially, all memory is free and viewed as one large block (a "hole"). As processes enter, they are loaded into available memory blocks that match their size needs, which are then marked as occupied. When a process completes, its memory is released and can be reused by another process from the input queue. (hotel with variable seaters)

At any time, the operating system maintains a list of available memory blocks and an input queue of processes waiting for memory. It allocates memory to processes until no available block is large enough to meet the next process's memory requirements. The OS can either wait until a suitable block is available or check for smaller processes that might fit into available memory instead.

Over time, the available memory is scattered across various-sized holes throughout memory. When a new process arrives, the system looks for a hole large enough to accommodate it. If the hole is too big, it splits the hole, giving part to the process and keeping the rest as a free hole. When a process finishes, its memory block is freed and added back to the list of holes. If this hole is next to other free holes, they merge to form a larger hole, which may meet the needs of other processes in the queue.

This approach to managing free memory is part of a broader dynamic storage allocation problem, which is about finding a hole of size *n* from a list of available holes. Common strategies to handle this include **first-fit** (using the first hole that fits), **best-fit** (choosing the smallest hole that fits), and **worst-fit** (choosing the largest hole).

- **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fit**. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

First-Fit:

- **Logic**: Scans memory from the beginning and selects the first available free block (hole) that is large enough to fit the process.
- **Advantages**:
  - Simple and fast since it stops searching as soon as a suitable block is found.
  - Minimal search time compared to other strategies.
- **Disadvantages**:
  - May leave small unusable fragments at the beginning of memory (external fragmentation).
  - Does not always utilize memory optimally.

---

2. Best-Fit:

- **Logic**: Searches the entire memory and selects the smallest hole that is large enough to fit the process.
- **Advantages**:
  - Minimizes wasted space in the selected block, potentially reducing external fragmentation.
- **Disadvantages**:
  - Slower because it must scan the entire memory.
  - Can lead to many small, unusable holes scattered throughout memory.

---

3. Worst-Fit:

- **Logic**: Searches the entire memory and selects the largest available hole to fit the process.
- **Advantages**:
  - Leaves larger leftover holes, which may be more useful for future processes.
- **Disadvantages**:
  - Often creates larger fragmentation over time as big holes are broken into smaller, less useful pieces.
  - Inefficient for systems where smaller holes are more commonly needed.

Strategy	Search Time	Fragmentation	Suitability
First-Fit	Fast	Moderate (external)	Good for quick allocation
Best-Fit	Slow	Small holes (external)	Good when minimizing waste is crucial
Worst-Fit	Slow	Large holes (external)	Good if larger future allocations are expected

iii) Fragmentation

Fragmentation refers to the inefficient utilization of memory due to its division into non-contiguous blocks, making it challenging to allocate contiguous memory for processes even when there is sufficient total memory available. Fragmentation can occur in **both main memory (RAM)** and **secondary storage**.

Types of Fragmentation

1. **External Fragmentation**
    - **Definition**: Occurs when free memory is scattered across different locations, but none of the free blocks are large enough to meet the memory request of a process.
    - **Cause**:
        - Processes are allocated and deallocated dynamically, leaving behind small unused memory blocks between allocated spaces.
    - **Example**:
        - A memory space has free blocks of 10KB, 15KB, and 20KB, but a process requiring 25KB cannot be allocated, even though the total free memory is 45KB.
    - **Solution**:
        - **Compaction**: Combine all free memory blocks into a single large block by moving processes to make contiguous memory.
        - **Paging or Segmentation**: Divide memory into fixed-size blocks or logical divisions to avoid fragmentation.

2. **Internal Fragmentation**
    - **Definition**: Occurs when allocated memory exceeds the process's requirements, leaving unused memory within an allocated block.
    - **Cause**:
        - Fixed-size memory blocks are allocated, and the process does not use the entire allocated block.
    - **Example**:
        - If memory is allocated in fixed-size blocks of 4KB and a process needs only 3KB, the remaining 1KB is wasted.
    - **Solution**:
        - **Dynamic Allocation**: Allocate memory based on exact process requirements.
        - **Variable Partitioning**: Allocate blocks based on process size to reduce waste.

Feature	External Fragmentation	Internal Fragmentation
Definition	Free memory is split into small, scattered blocks.	Unused memory within an allocated block.
Cause	Non-contiguous memory allocation; insufficiently large blocks to hold a process.	Block allocation is slightly larger than the process needs.
Impact	Total memory is enough for a process but is non-contiguous, preventing allocation.	Wasted memory within blocks, reducing overall memory efficiency.
Occurrence	Happens in dynamic memory allocation (e.g., first-fit, best-fit) when processes leave gaps.	Occurs when fixed-size blocks are used, and processes don't use all assigned memory.
Solution	Compaction, segmentation, or paging.	Allocating memory in units or smaller block sizes to match process needs.
Memory Type Affected	Unused space **between** allocated blocks (holes in memory).	Unused space **within** allocated blocks.
Severity	Can be severe as it restricts large allocations despite available memory.	Less severe, as unused space is contained within each block.

4. Segmentation, Paging

Segmentation is a memory management technique that divides a process's memory into different segments based on its logical divisions, such as code, data, and stack segments. Each segment represents a logical unit that a programmer or operating system can manage independently. Unlike paging, where memory is divided into fixed-size blocks, segmentation allows for variable-sized segments, which align more naturally with how programs are structured.

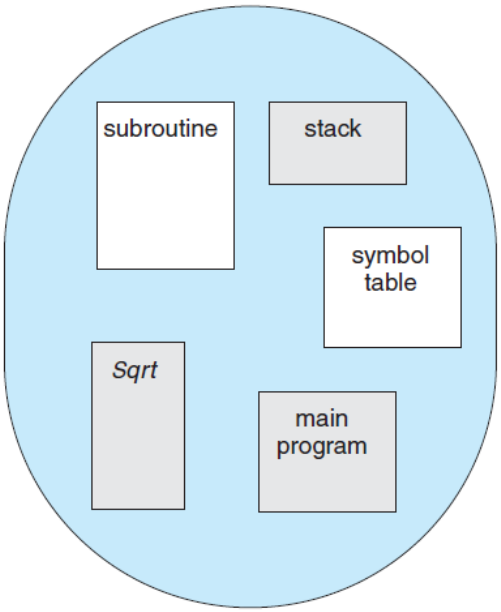
i) Basic Method

Most programmers don't think of memory as a simple, continuous line of bytes. Instead, they see memory as a collection of separate sections, or "segments," each with a specific purpose and size.

When writing code, programmers think in terms of a main program with functions or methods, along with data structures like arrays, variables, and stacks. Each part of the program is referenced by name, like "the stack," "the math library," or "the main function," without worrying about the exact memory addresses these parts occupy. The order of these segments in memory doesn't matter to the programmer; they just care about what each segment does.

Within each segment, elements are located by their position from the start of that segment—like the first line of code, or the fifth step in a function. This approach is more intuitive and matches the logical structure of programs.

Segmentation is a memory-management scheme that supports this programmer view of memory. A logical address space is a collection of segments.



Programmer's view of a program.

Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The programmer therefore specifies each address by two quantities: a segment name and an offset.

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a ***two tuple:***

<segment-number, offset>.

Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.

A C compiler might create separate segments for the following:

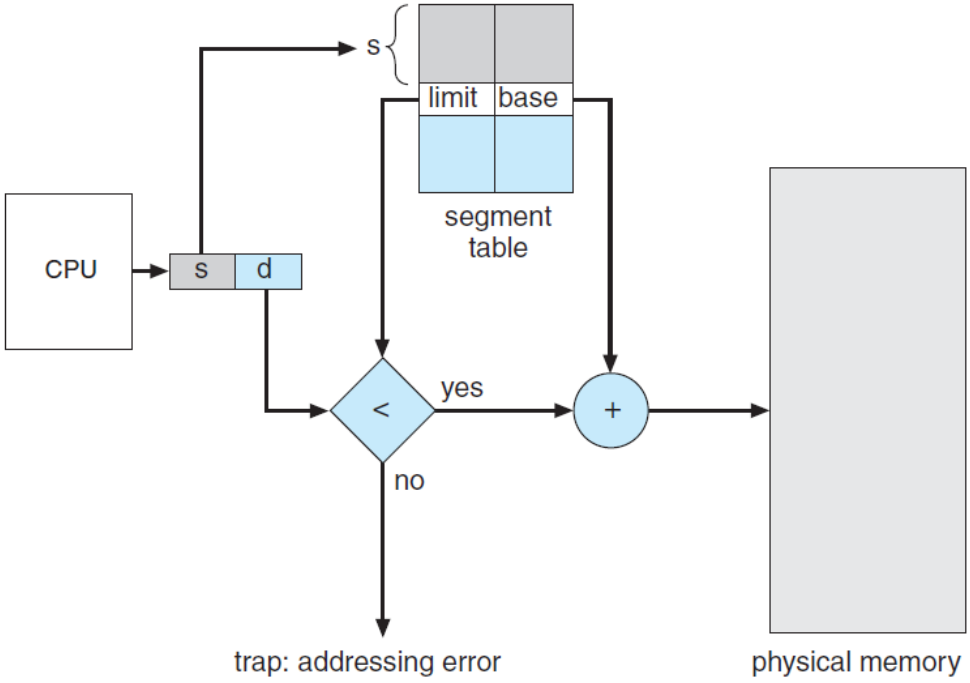
1. The code
2. Global variables
3. The heap, from which memory is allocated
4. The stacks used by each thread
5. The standard C library

Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

ii) Segmentation Hardware

Although the programmer can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one dimensional sequence of bytes. Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses.

This mapping is affected by a **segment table**. Each entry in the segment table has a **segment base** and a **segment limit**. The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.



Segmentation hardware.

The use of a segment table is illustrated in above Figure. A logical address consists of two parts: a segment number, *s*, and an offset into that segment, *d*.
The segment number is used as an index to the segment table. The offset *d* of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment).
When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base–limit register pairs.

Advantages of Segmentation

1. **Better Logical Organization**:
   - Programs are divided into meaningful units, making it easier to manage.
2. **Protection**:
   - Each segment can have distinct access rights (e.g., read-only for code, read-write for data).
3. **Efficient Memory Utilization**:
   - Segments are allocated memory based on their actual size, reducing internal fragmentation.
4. **Supports Sharing**:
   - Segments like code can be shared among processes if they are read-only.

Disadvantages of Segmentation

1. **External Fragmentation**:
   - Since segments are of variable size, dynamic allocation can leave scattered free memory blocks.
2. **Complex Management**:
   - Maintaining segmentation tables and performing address translation can be more complex compared to paging.

Let us consider a program divided into 3 segments:

Segment Number	Base Address	Limit (Size)
0	3000	100
1	5000	200
2	7000	150

Logical Address

A logical address is given as <s, d>, where:

- **s**: Segment number
- **d**: Offset within the segment

For example, the logical address <1, 120> refers to **Segment 1** with an offset of **120**.

Steps for Address Translation

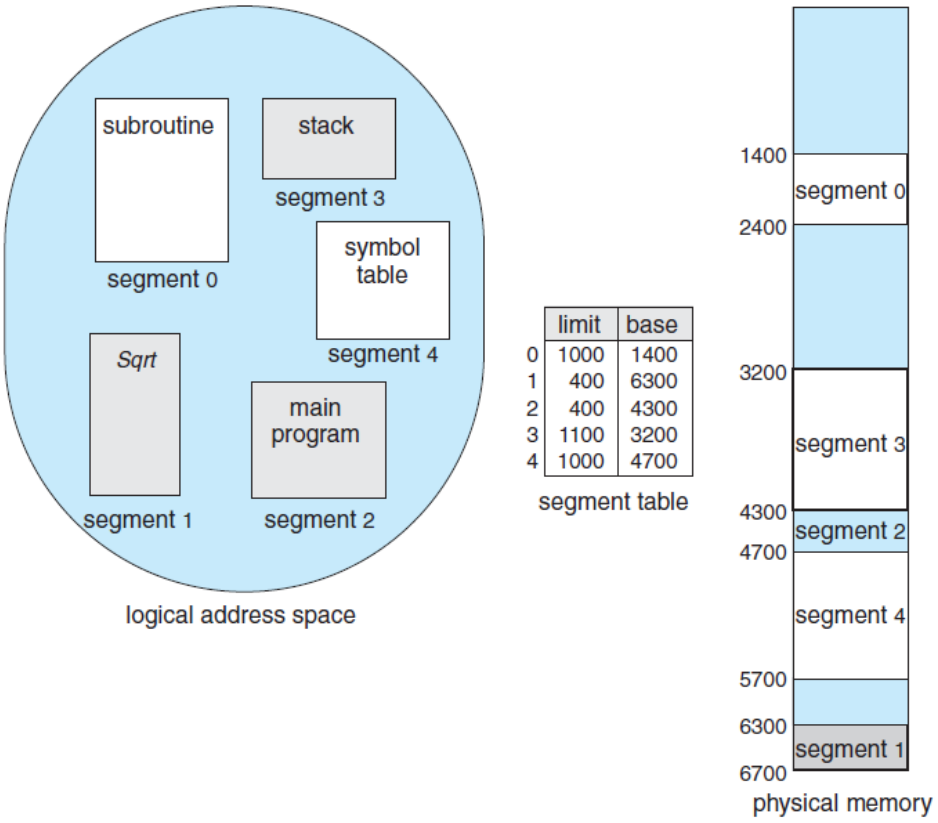
1. **Look up the Segment Table**:
    - Segment number **1** corresponds to:
        - **Base Address = 5000**
        - **Limit = 200**
2. **Validate Offset**:
    - Check if the offset **120** is less than the **Limit (200)**.
    - Since 120 < 200, the offset is **valid**.
3. **Calculate Physical Address**:
    - Add the **Base Address** to the **Offset**:
        - Physical Address = Base (5000) + Offset (120) = **5120**

Invalid Offset Example

If the logical address is <1, 250>:

- Segment **1** has a limit of **200**.
- Since 250 > 200, the offset is invalid, and the system traps to the OS for an error.

This ensures the program accesses only the memory allocated for its segments, enforcing memory protection.



Example of segmentation.

In the example described, we have five segments, each with a unique segment number from 0 to 4. These segments are stored at different starting locations in physical memory. To keep track of each segment's location and size, a *segment table* is used. Each entry in the table contains:

- The *base*, which is the starting address of the segment in physical memory.
- The *limit*, which is the length of the segment.

For instance:

- Segment 2 starts at address 4300 and is 400 bytes long. So, if we reference byte 53 of segment 2, we calculate its actual memory location by adding 53 to the base of 4300, resulting in address 4353.
- Similarly, a reference to segment 3, byte 852, maps to address 3200 (base of segment 3) + 852, which equals 4052.

If we try to access byte 1222 of segment 0, it would cause an error (or *trap* to the OS) because segment 0 is only 1000 bytes long, so byte 1222 is out of bounds. This way, the segment table helps ensure each reference stays within the segment's limits.

Aspect	Paging	Segmentation
Definition	Divides logical memory into fixed-sized blocks called pages and maps them to physical memory frames.	Divides logical memory into variable-sized segments based on the program's logical divisions.
Size	Fixed size (determined by hardware).	Variable size (based on logical structure like functions, arrays, etc.).
Address Components	Logical address consists of a page number and page offset.	Logical address consists of a segment number and segment offset.
Purpose	Optimizes memory utilization by eliminating external fragmentation.	Provides a logical way to manage memory based on program modules.
Mapping	Page table is used to map pages to physical frames.	Segment table is used to map segments to physical memory blocks.
Fragmentation	Can cause internal fragmentation due to fixed page sizes.	Can cause external fragmentation because of variable-sized segments.
Visibility to User	Transparent to the user; the user doesn't see or manage pages.	Visible to the user; the user defines and organizes memory into segments.
Memory Access	Access requires page table lookups (may use a TLB for faster translation).	Access requires segment table lookups.
Logical Structure	Does not consider logical divisions of the program.	Matches the program's logical structure (e.g., code, stack, heap).
Usage	Commonly used in virtual memory systems for efficient memory management.	Used in older systems or special scenarios requiring logical organization.
Protection	Provides protection at the page level (read/write/execute permissions).	Provides protection at the segment level (based on logical divisions).
Overhead	Overhead due to maintaining page tables.	Overhead due to maintaining segment tables and dealing with external fragmentation.
Examples	Most modern operating systems (e.g., Windows, Linux) use paging for virtual memory.	Early systems like Intel 8086 and architectures relying on logical memory divisions used segmentation.

Paging and Internal Fragmentation

Internal fragmentation occurs in paging because memory is allocated in fixed-sized blocks (pages), which might not be fully utilized by the process.

Example:

- Assume a system with a page size of **4 KB**.
- A process requires **10 KB** of memory.
- The process is divided into three pages:
  - Page 1: 4 KB (fully utilized).
  - Page 2: 4 KB (fully utilized).
  - Page 3: 2 KB (remaining portion of the process).

The third page has **2 KB of unused memory**, leading to internal fragmentation. This wasted space arises because the page size is fixed, and the process size doesn't always perfectly match page boundaries.

Segmentation and External Fragmentation

External fragmentation occurs in segmentation because memory is allocated in variable-sized blocks (segments) based on the program's needs. Over time, as segments are allocated and deallocated, free memory is scattered into non-contiguous blocks, making it difficult to allocate large segments despite having sufficient total free memory.

Example:

- Consider a memory of **100 KB** initially free.
- A process requests three segments:
  - Segment 1: 20 KB
  - Segment 2: 30 KB
  - Segment 3: 40 KB

After allocation, the memory looks like this:

```
| Segment 1 (20 KB) | Segment 2 (30 KB) | Segment 3 (40 KB) | Free (10 KB) |
```

Now, Segment 2 (30 KB) is deallocated:

```
| Segment 1 (20 KB) | Free (30 KB) | Segment 3 (40 KB) | Free (10 KB) |
```

A new process requests a segment of **35 KB**. Even though there is **40 KB of free memory** (30 KB + 10 KB), the segment cannot be allocated because the free memory is not contiguous. This results in external fragmentation.

- **Paging (Internal Fragmentation)**: Fixed page sizes waste space within individual pages if the process doesn't perfectly fit the page size.

- **Segmentation (External Fragmentation)**: Variable-sized memory blocks create gaps over time, making it hard to utilize free memory effectively despite having sufficient total free space.

Paging

Segmentation permits the physical address space of a process to be non-contiguous.

i) Basic Method

Paging is another memory-management scheme that offers this advantage. However, paging avoids external fragmentation and the need for compaction, whereas segmentation does not.

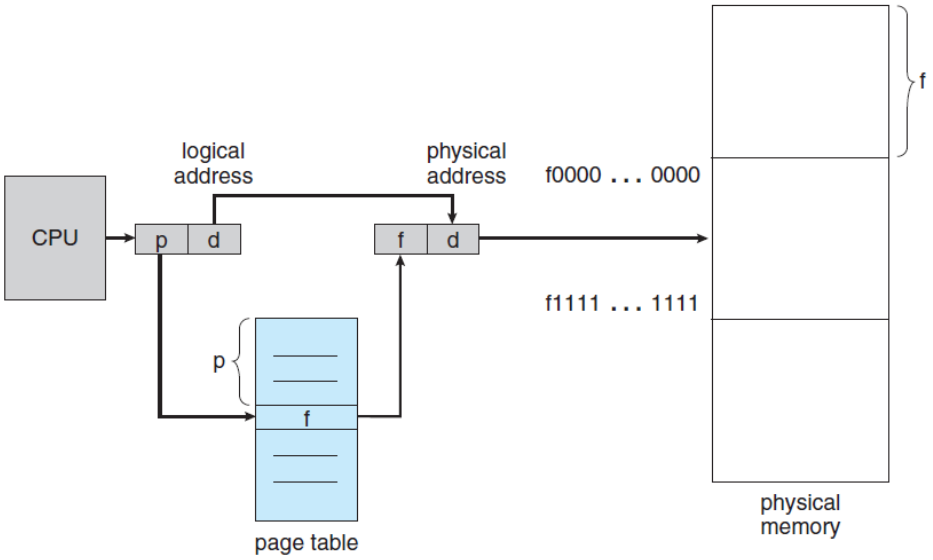
It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store. Most memory-management schemes used before the introduction of paging suffered from this problem. The problem arises because, when code fragments or data residing in main memory need to be swapped out, space must be

found on the backing store. The backing store has the same fragmentation problems discussed in connection with main memory, but access is much slower, so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is used in most operating systems, from those for mainframes through those for smartphones. Paging is implemented through cooperation between the operating system and the computer hardware.

Paging is a memory management technique where both physical and logical memory are divided into fixed-sized blocks. Here's a breakdown of how it works:

1. **Frames and Pages**:
    - Physical memory is divided into blocks called *frames*.
    - Logical memory is divided into blocks of the same size, known as *pages*.
    - When a process is to be executed, its pages are loaded into available frames in physical memory. This ensures that the process's pages can fit into the frames as needed.
2. **Backing Store**:
    - The *backing store* (usually disk storage) is also divided into blocks the same size as frames.
    - Pages of a process are stored here initially and are loaded into memory as frames become available.
3. **Separation of Address Spaces**:
    - One of the advantages of paging is that the logical address space of a process is completely separate from the physical address space.
    - For example, even if the physical memory is limited, a process could have a 64-bit logical address space, meaning it can utilize a large range of addresses within its own address space without requiring equivalent physical memory.
4. **Address Translation Using Page Tables**:
    - The CPU generates addresses that are divided into two parts: a *page number* (p) and a *page offset* (d).
    - The page number serves as an index to a *page table*, which stores the base address of each page in physical memory.
    - To get the actual physical address, the base address from the page table is combined with the page offset, and this complete physical address is sent to the memory unit.

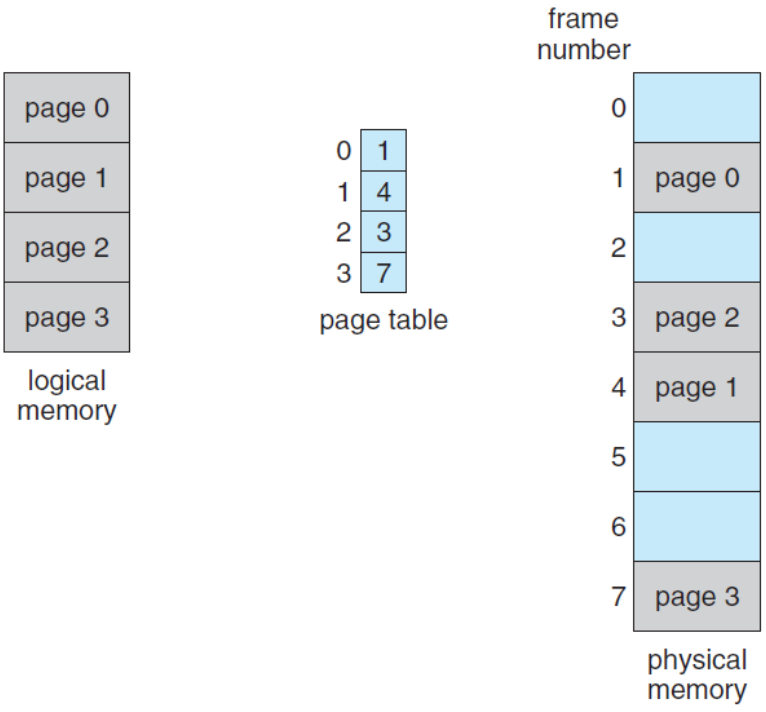
By separating logical and physical memory through paging, the system gains flexibility and can use memory more efficiently, even supporting large logical address spaces on systems with limited physical memory. This mechanism is essential for enabling virtual memory in modern computing systems.



Paging hardware.

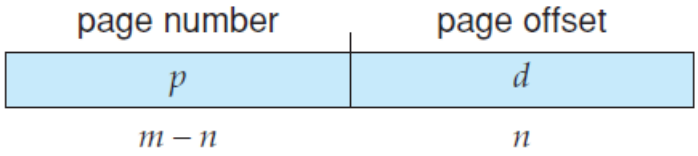
Feature	Segmentation	Paging
Division	Variable-sized logical segments	Fixed-size pages
Fragmentation	Prone to external fragmentation	Prone to internal fragmentation
Addressing	`<segment_number, offset>`	`<page_number, offset>`
Purpose	Logical program division	Simplified memory allocation

Segmentation provides a logical view of memory that aligns with program structure, while paging focuses on fixed-size blocks for efficient allocation.

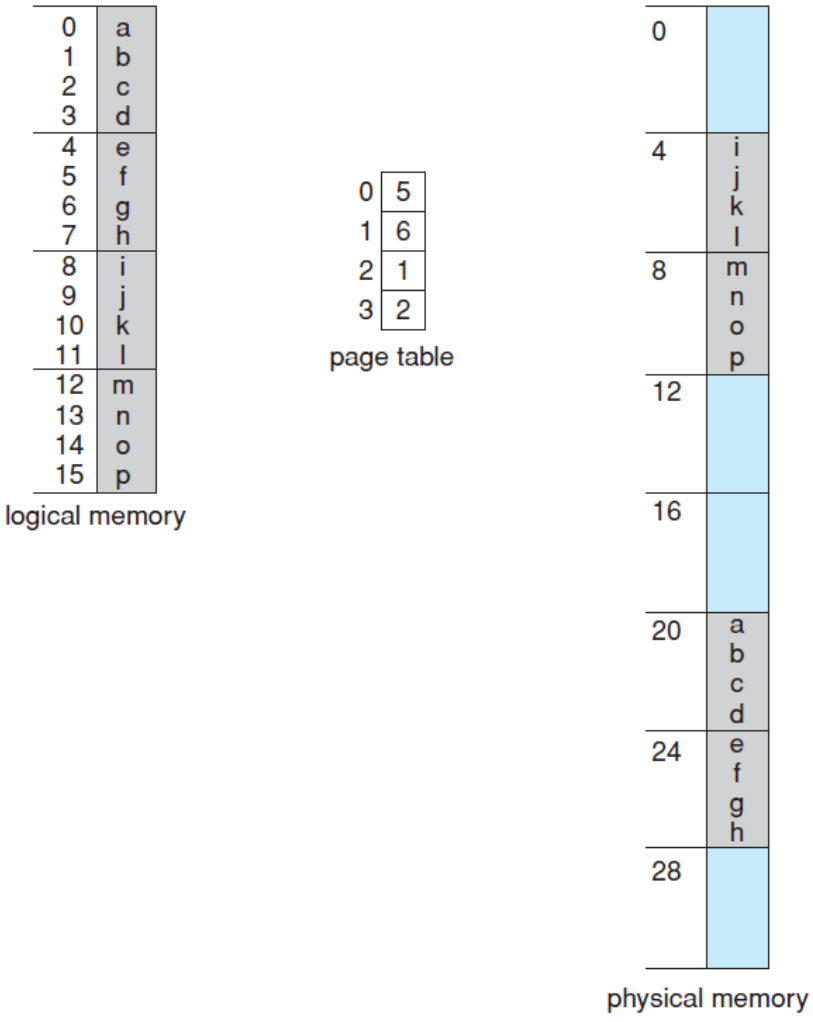


Paging model of logical and physical memory.

The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1 GB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the high-order $m - n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:



where p is an index into the page table and d is the displacement within the page. As a concrete (although minuscule) example, consider the memory in below Figure. Here, in the logical address, $n = 2$ and $m = 4$. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5.



Paging example for a 32-byte memory with 4-byte pages.

Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3]. Logical address 4 is page

1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0]. Logical address 13 maps to physical address 9.
You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory. When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires *n* pages, at least *n* frames must be available in memory. If *n* frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on(below figure).

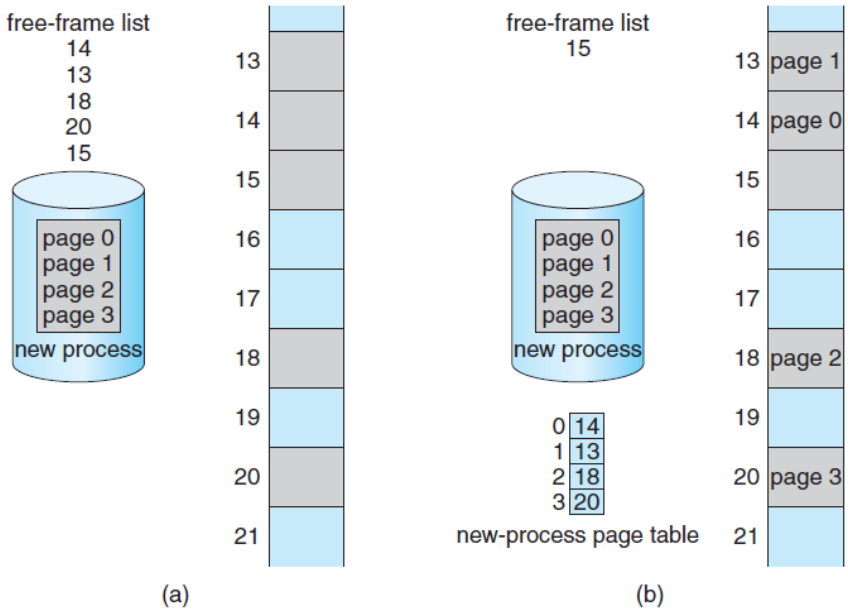


Figure 8.13 Free frames (a) before allocation and (b) after allocation.

Paging separates the programmer's view of memory from the actual physical memory layout. Here's a simplified summary:

Programmer's View vs. Physical Reality

- The programmer sees memory as a single, continuous space dedicated to their program. However, in reality, their program's pages are scattered throughout physical memory, along with pages from other programs.
- **Address translation hardware** maps these logical addresses (from the programmer's view) to actual physical addresses. This mapping is done by the operating system and is hidden from the programmer.

Memory Protection

- Each program can only access its own pages, as defined by its page table. It cannot access memory outside of this table, keeping processes secure from one another.

Frame Table

- The operating system manages memory through a **frame table**. This table tracks each frame (memory block) to see if it's free or in use, and if in use, which program is using it.

System Calls and Address Mapping

- When a program makes a system call, such as for input/output, it may pass a memory address (like a buffer location). The OS then translates this logical address to the correct physical address.
- The OS keeps a copy of each program's page table, similar to how it saves the program's instructions and register values. This page table copy is used by the **CPU dispatcher** when switching processes.

Paging and Context Switching

- Since each process has its own page table, paging slightly increases **context-switch time**. However, it allows the OS to effectively manage memory while keeping processes isolated from each other's data.

ii) Hardware Support

Operating systems handle page tables in different ways:

1. **Separate Page Table per Process**:
    - Some OSs create an individual page table for each process. A pointer to this table is saved with other important register values (like the instruction counter) in the process control block (PCB).
    - When a process starts, the system reloads this page table and other registers to match.
2. **Shared Page Tables**:
    - Other OSs may use one shared page table or only a few, which reduces the work needed during context switches.

Hardware Implementation of Page Tables

1. **Dedicated Registers**:

    - For small page tables, the OS can store the page table directly in dedicated, high-speed registers. This makes address translation faster since each memory access uses this page table.
    - Only the OS can modify these registers (this is a privileged operation). For instance, the DEC PDP-11 uses this method, with 8 entries kept in fast registers.
2. **Page Table in Main Memory**:
    - For large page tables, the OS stores the table in main memory, with a **Page Table Base Register (PTBR)** pointing to its location.
    - Changing the active page table for a new process only requires updating the PTBR, which speeds up context switching.

Accessing Memory with PTBR

- When using the PTBR, accessing memory involves two steps:
    1. First, use the PTBR to find the page table entry for the page containing the target memory.
    2. Then, use the frame number and offset to get the actual address.
- This setup requires two memory accesses, which doubles the access time, potentially slowing performance.

The **Translation Lookaside Buffer (TLB)** is a specialized, fast cache used in memory management to improve the speed of virtual address translation. It stores recent translations of virtual memory addresses to physical memory addresses, reducing the need to access the slower page table in memory.

Example Workflow

1. The CPU generates a virtual address (e.g., page number + offset).
2. The MMU checks the TLB:
    - If the page number is in the TLB (**hit**), the corresponding frame number is retrieved.
    - If the page number is not in the TLB (**miss**), the MMU accesses the page table in memory, retrieves the mapping, and updates the TLB.
3. The physical address is computed using the frame number and the offset, and memory is accessed.

The **Page Table Base Register (PTBR)** is a hardware register used in memory management systems to store the starting address of the page table for the currently running process. It plays a critical role in virtual memory systems by enabling the operating system to manage and translate virtual addresses into physical addresses efficiently.

Example Workflow with PTBR

1. The CPU generates a virtual address, consisting of a **page number** and a **page offset**.
2. The MMU retrieves the PTBR's value, which points to the base address of the page table.

3. Using the page number as an index, the MMU locates the corresponding entry in the page table.
4. The page table entry provides the physical frame number for the page.
5. The physical address is calculated by combining the frame number with the page offset.
6. The physical address is used to access memory.

To speed up address translation, modern systems use a small, fast hardware cache called a **Translation Lookaside Buffer (TLB)**. The TLB is a high-speed, associative memory that quickly matches entries without needing to search through all of them sequentially. Each TLB entry has two parts: a *key* (or tag) and a *value*. When a page number (key) is given to the TLB, it compares it to all keys at once. If a match is found, the corresponding frame number (value) is returned immediately, adding almost no delay. Since TLBs are small, typically holding between 32 and 1,024 entries, they work within the CPU's instruction pipeline without slowing it down. Some CPUs even split the TLB into separate parts for instructions and data, effectively doubling the entries.

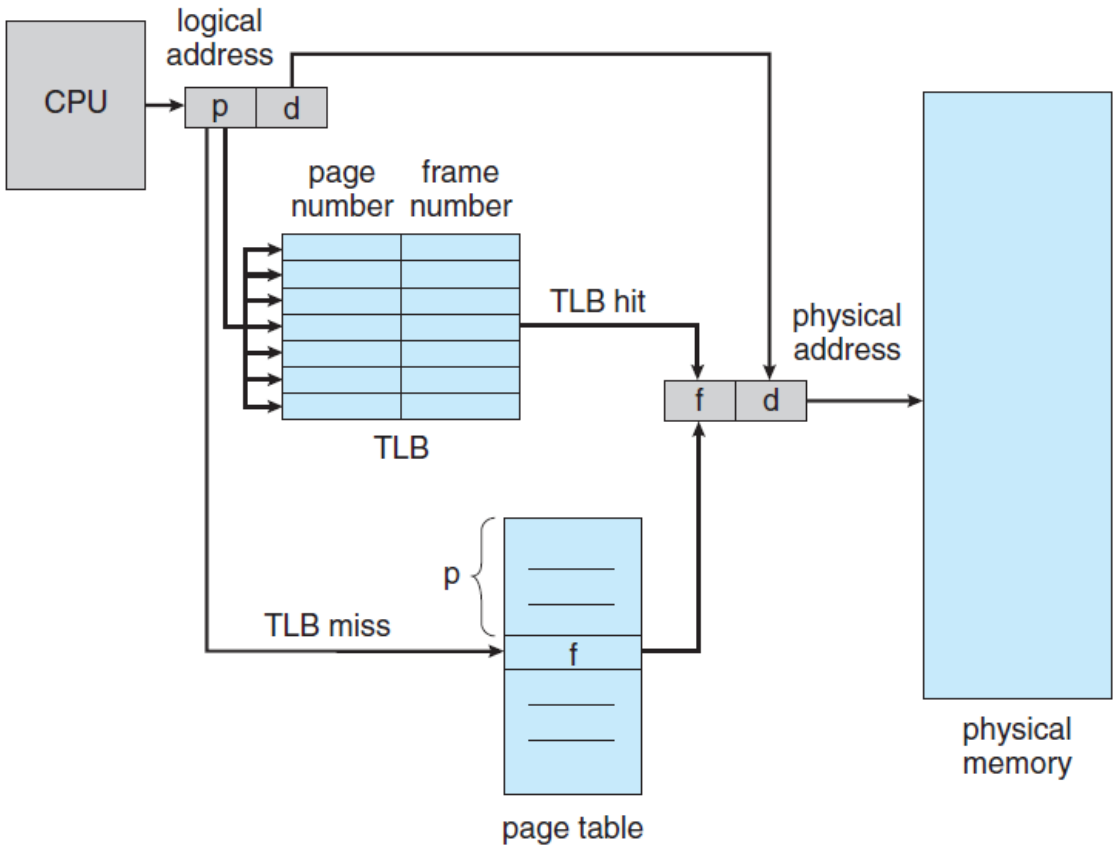
The TLB works with page tables as follows:

1. When the CPU generates a logical address, the page number is sent to the TLB.
2. If the TLB has this page number *(TLB hit)*, it returns the frame number directly, and memory can be accessed without extra delay.
3. If the TLB does not have the page number (a *TLB miss*), it retrieves the frame number from the page table in memory. The new page and frame numbers are then added to the TLB for quicker access next time.

When the TLB is full, an older entry must be replaced. Replacement strategies vary, from **Least Recently Used (LRU)** to **round-robin** or **random** replacements.

To manage multiple processes, some TLBs include *Address Space Identifiers (ASIDs)*, which identify each process and ensure only the current process's addresses are accessed. Without ASIDs, the TLB must be cleared (or *flushed*) on a context switch to avoid incorrect address translations from leftover entries.

Aspect	TLB	Page Table
Location	Located in the MMU as a hardware cache.	Stored in main memory.
Capacity	Stores a small number of entries (limited by hardware).	Can store all mappings for the process's virtual address space.
Hit/Miss	A TLB hit results in fast translation; a TLB miss requires a page table lookup.	No concept of a "hit" or "miss"; always accessed if the TLB misses.



Paging hardware with TLB.

The effectiveness of the TLB is measured by the *hit ratio* — the percentage of times the desired page number is found in the TLB. For example, an 80% hit ratio means 80% of the time, the TLB has the required page, allowing fast access. If a TLB hit takes 100 nanoseconds, but a TLB miss takes 200 nanoseconds (since it requires accessing both the page table and memory), the higher the hit ratio, the faster the overall memory access will be.

To find the **effective memory-access time**, we weight the case by its probability:
effective access time = 0.80 × 100 + 0.20 × 200 = 120 nanoseconds
In this example, we suffer a 20-percent slowdown in average memory-access time (from 100 to 120 nanoseconds).
For a 99-percent hit ratio, which is much more realistic, we have effective access time = 0.99 × 100 + 0.01 × 200= 101 nanoseconds

This increased hit rate produces only a 1 percent slowdown in access time.

The **effectiveness of a TLB** is crucial to improving memory access times in a virtual memory system. It depends heavily on the **TLB hit ratio**, which represents the proportion of memory accesses that are resolved directly by the TLB, avoiding the need for slower page table lookups.

Effective Memory Access Time

The **effective memory-access time (EAT)** is the average time taken to access memory, accounting for both TLB hits and misses. It is calculated as follows:

$$\text{EAT} = (\text{Hit Ratio}) \times (\text{Time for TLB Hit}) + (\text{Miss Ratio}) \times (\text{Time for TLB Miss})$$

Where:

- **Hit Ratio** = Fraction of memory accesses resolved by the TLB.
- **Miss Ratio** = 1−Hit Ratio, fraction requiring a page table lookup.
- **Time for TLB Hit** = Time to access memory directly using TLB.
- **Time for TLB Miss** = Time to handle a TLB miss, which includes the time to access the page table and resolve the address.

Example Calculations

1. **Hit Ratio = 80% (0.80)**:
    - TLB Hit Time = 100 nanoseconds.
    - TLB Miss Time = 200 nanoseconds.
    - Effective Access Time:
    - EAT=(0.80×100)+(0.20×200)=80+40=120 nanoseconds
    - **Result**: 20% slower than ideal TLB hit time (100 nanoseconds).
2. **Hit Ratio = 99% (0.99)**:
    - TLB Hit Time = 100 nanoseconds.
    - TLB Miss Time = 200 nanoseconds.
    - Effective Access Time:
    - EAT=(0.99×100)+(0.01×200)=99+2=101 nanoseconds
    - **Result**: Only 1% slower than the ideal TLB hit time.

Analysis of Results

- A higher **hit ratio** results in a more efficient memory system because most memory accesses are resolved quickly through the TLB.
- The **penalty for a TLB miss** is significant because it involves accessing both the page table and memory, leading to slower performance.

iii) Protection

In a paged environment, memory protection is managed using protection bits tied to each frame, typically stored in the page table. A simple bit can indicate if a page is read-write or read-only. Every memory access checks these protection bits to ensure no writes are made to read-only pages; if a write is attempted, a hardware trap alerts the operating system to a memory protection error. This system can be extended to finer protection levels, allowing pages to be marked as read-only, read-write, or execute-only, or even combinations of these. Any illegal access attempt is trapped to the OS for handling. Each page table entry also has a *valid-invalid bit*, which specifies whether the page belongs to the process's logical address space. A page marked "invalid" triggers an error if accessed, as it falls outside the legal address range for that process. Most processes don't use their entire address space, so creating a page table with entries for every possible page would waste memory. To prevent this, some systems use a *page-table length register (PTLR)*, which records the actual size of the page table. This register ensures addresses stay within the valid range by verifying each address against the PTLR limit.

Page Table Length Register (PTLR)

The **Page Table Length Register (PTLR)** is a hardware component used in systems with paging memory management. It indicates the size of the page table for a given process. This helps the system know the valid range of page table entries and ensures efficient memory access while preventing illegal memory access.

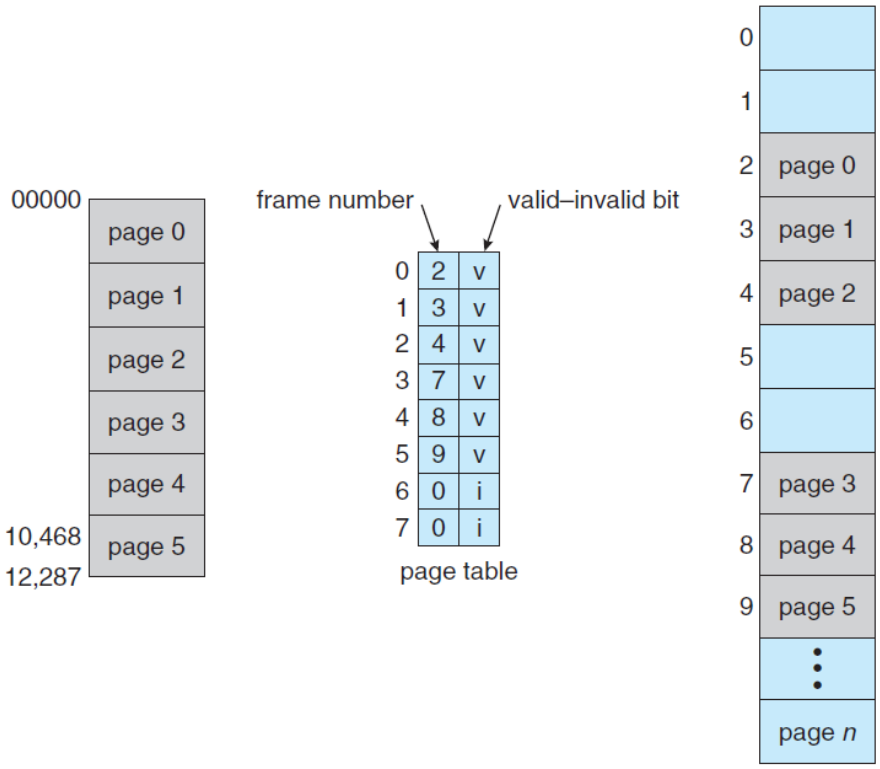
Functions of PTLR:

1. **Defining Page Table Boundaries**:
   PTLR specifies the total number of pages that belong to a process. This is particularly useful in systems where processes have varying memory requirements.
2. **Preventing Invalid Access**:
   During a memory reference, the system checks the page number against the PTLR to ensure the page number is within the valid range of the page table. If the page number exceeds this range, it is considered invalid, and a trap (exception) is raised.
3. **Dynamic Process Support**:
   PTLR allows processes with different memory demands to coexist by accommodating page tables of different sizes.

Example:

- Suppose a process has a page table containing 10 entries. The PTLR is set to 10, indicating that the page table has valid entries for pages 0 to 9.
- If a memory reference attempts to access page 15, the system checks the PTLR and determines that this is an invalid page since it exceeds the defined range. An exception is raised to handle this invalid memory access.

This register enhances memory protection and ensures that the system operates within the bounds of allocated resources for each process.



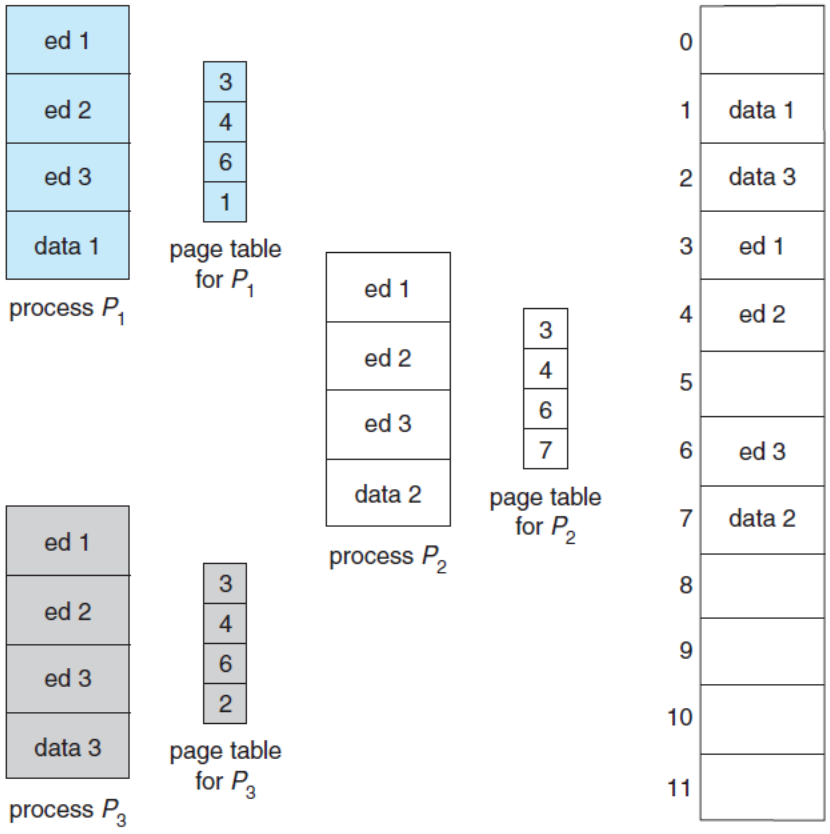
Valid (v) or invalid (i) bit in a page table.

iv) Shared Pages

Paging enables sharing of common code, which is especially useful in a time-sharing system. For instance, imagine 40 users using a text editor. If each editor instance has 150 KB of code and 50 KB for data, running 40 instances would require 8,000 KB in total. However, if the code is re-entrant (non-modifiable), it can be shared among users.

In this setup, each user accesses the same physical copy of the editor's code (150 KB), but each has a unique data space (50 KB). This means we only need one copy of the editor in memory, plus the individual data pages for each user. For 40 users, this requires only 2,150 KB, a major memory saving compared to 8,000 KB.

Other commonly used programs, like compilers or libraries, can also be shared. For code to be shared, it must be reentrant—meaning it doesn't modify itself during execution. The OS enforces read-only access on shared code to prevent accidental modifications, ensuring multiple processes can safely execute the same code at once.



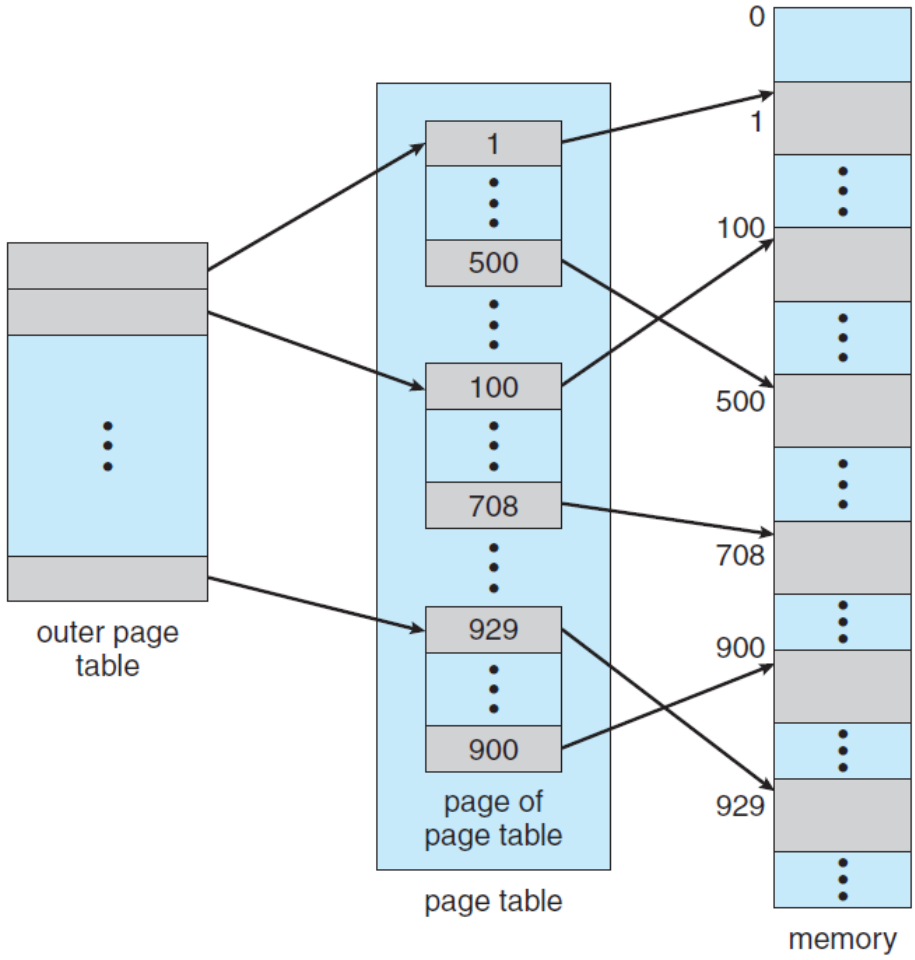
Sharing of code in a paging environment.

5. Structure of page table

The structure of a page table varies based on the technique used to manage virtual memory efficiently. The three most common techniques are **hierarchical paging**, **hashed page tables**, and **inverted page tables**.

i) Hierarchical Paging

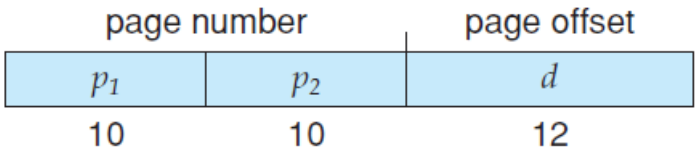
Modern computer systems often support very large logical address spaces (up to 32 or 64 bits), leading to extremely large page tables. For example, in a 32-bit address space with a page size of 4 KB (2^{12}), the page table could require up to 1 million entries $(2^{32}/2^{12})$. If each entry takes 4 bytes, each process might need up to 4 MB just for its page table. Allocating such a large page table contiguously in memory is impractical. To address this, the page table can be broken down into smaller, manageable pieces. One approach is **two-level paging**, where the page table itself is paged. Here's how it works:



A two-level page-table scheme.

1. **Dividing the Address**: Consider a 32-bit logical address with a 4 KB page size. The address is split into:
   - A **20-bit page number** and
   - A **12-bit page offset**.
2. **Paging the Page Table**: The 20-bit page number is further divided into:
   - A **10-bit outer page number**, which indexes a first-level page table.
   - A **10-bit inner page offset**, which is used in the second-level page table.

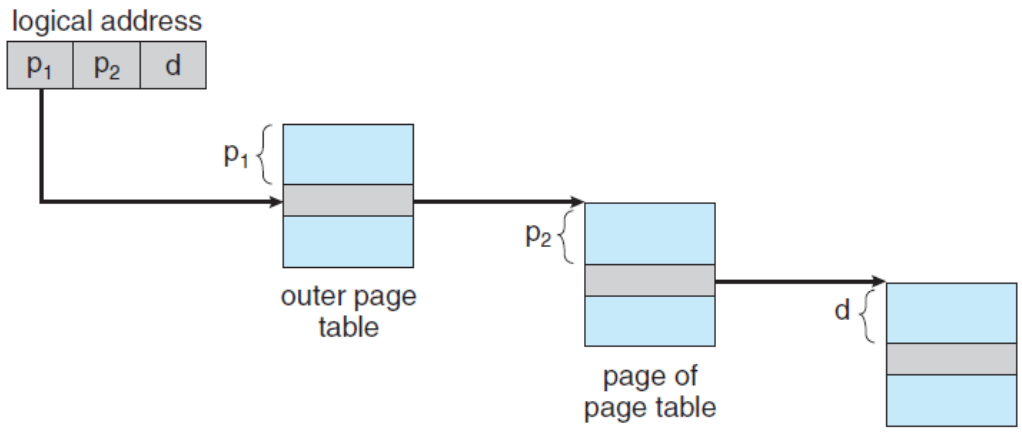
Thus, a logical address is as follows:



where $p1$ is an index into the outer page table and $p2$ is the displacement within the page of the inner page table.

With this two-level approach, each smaller part of the page table can be loaded separately, reducing memory requirements by only loading needed sections. This method allows efficient handling of large address spaces without requiring one massive contiguous page table in main memory.

The address-translation method for this architecture is shown below



Address translation for a two-level 32-bit paging architecture.

ii) Hashed Page Tables

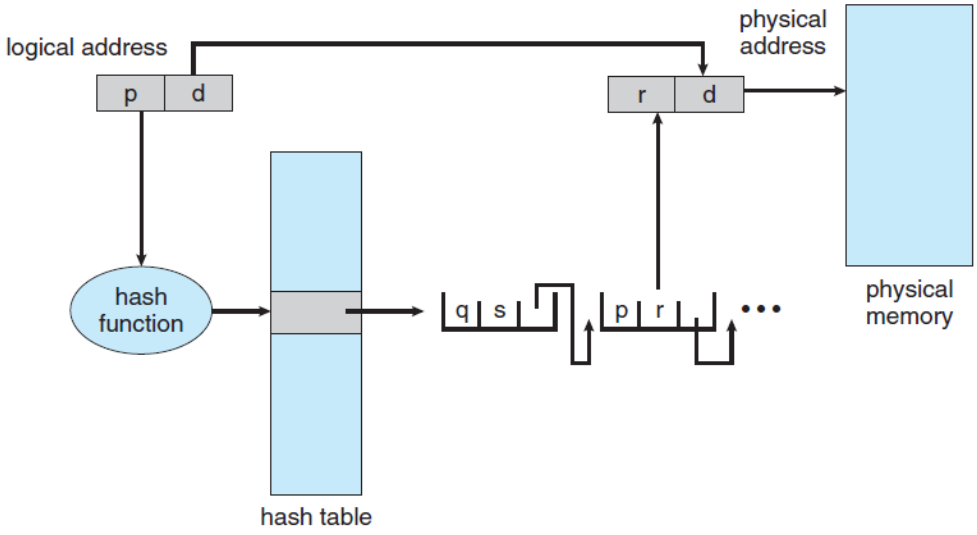
One way to manage address spaces larger than 32 bits is by using a **hashed page table**. In this approach, the virtual page number is hashed to determine the location in a hash table. Each entry in the hash table contains a **linked list** to handle cases where multiple entries hash to the same location (collisions). Each element in the linked list has three fields:

1. The **virtual page number**.
2. The **mapped page frame**.
3. A pointer to the **next element** in the list.

Here's how the process works:

- The virtual page number from the virtual address is hashed into the hash table.
- The hash table entry is checked. If the **virtual page number** matches the one in the first element of the linked list, the corresponding **page frame** (from the second field) is used to calculate the physical address.
- If no match is found, the algorithm continues to check the remaining entries in the linked list.

This system is shown in below Figure



Hashed page table.

For larger address spaces, such as 64-bit addresses, a variation of this method called **clustered page tables** has been proposed. In clustered page tables, each entry in the hash table maps to multiple pages (for example, 16 pages at once), instead of just one. This allows a single entry to store the mappings for multiple physical page frames, making it more efficient for sparse address spaces where memory references are scattered across non-contiguous areas.

iii) Inverted Page Tables

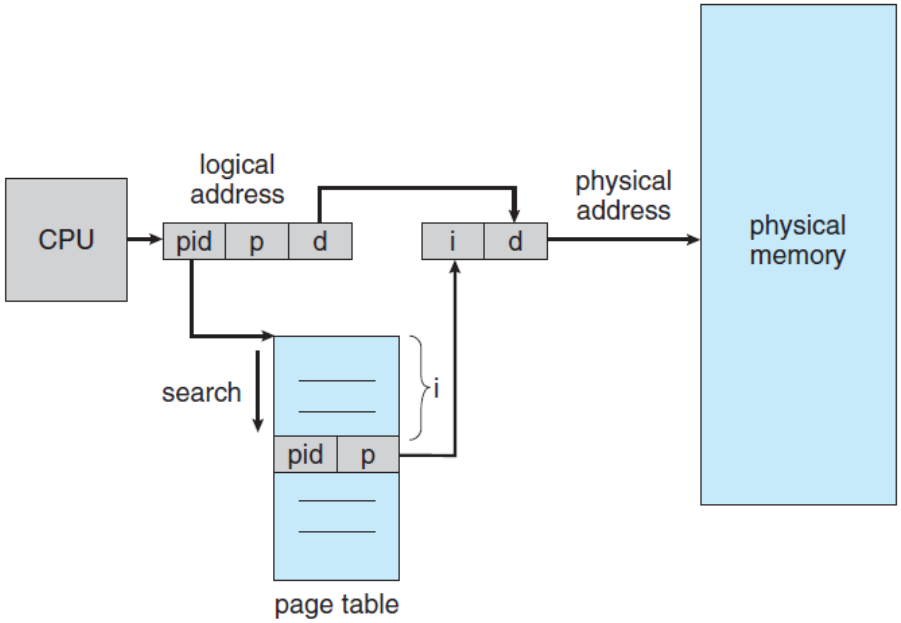
In a typical system, each process has its own **page table** that maps virtual addresses to physical memory. The page table contains an entry for each page the process is using, which makes it easy to look up the corresponding physical address when needed. The table is organized by virtual address, allowing the operating system to quickly find the physical address. However, this method can become inefficient because a process may use millions of virtual pages, leading to large page tables that consume a significant amount of memory.

To address this issue, **inverted page tables** are used. Instead of having one entry for each virtual page, an inverted page table has one entry for each physical page (or memory frame). Each entry contains:

- The **virtual address** of the page stored in that physical memory location.
- Information about the **process** that owns the page.

With this system, only one page table is needed for the entire system, and it tracks physical memory usage, rather than each process's virtual memory. An **address-space identifier (ASID)** is often included in each entry to help distinguish between different processes, ensuring that each virtual page is mapped to the correct physical memory for the right process.

Inverted page tables are used in systems like the **UltraSPARC (64-bit)** and **PowerPC**, where multiple address spaces might be mapped to the same physical memory.



Inverted page table.

To explain how the **inverted page table** works, let's look at a simplified version used in the **IBM RT** system. IBM was a pioneer in using inverted page tables, starting with the **IBM System 38** and continuing with the **RS/6000** and **IBM Power CPUs**.

In the IBM RT, each virtual address consists of three parts: <process-id, page-number, offset>. The inverted page table stores entries as pairs: <process-id, page-number>, with the **process-id** acting as the **address-space identifier (ASID)**. When a memory reference is made, the system presents the <process-id, page-number> portion of the virtual address to the memory subsystem. The inverted page table is then searched to find a matching entry. If a match is found (let's say at entry i), the physical address is generated as <i, offset>. If no match is found, it means the virtual address is invalid.

While inverted page tables save memory compared to traditional page tables, they introduce a performance issue. Since the table is sorted by physical address but lookups are based on virtual addresses, the system may need to search through the entire table to find a match, which can be slow. To solve this, a **hash table** is often used to speed up the search. The virtual address is hashed into the hash table, and the table quickly points to the relevant page-table entry. However, using the hash table still requires an extra memory access, so a memory reference may require at least two real memory accesses: one for the hash table and one for the page table.

A challenge with inverted page tables is implementing **shared memory**. Normally, shared memory is implemented by mapping multiple virtual addresses (one for each process) to the same physical address. However, in an inverted page table, there is only one entry for each physical page, so multiple virtual addresses cannot map to the same physical page. One solution is to allow the inverted page table to store just one mapping of a virtual address to the shared physical address. As a result, when a process references a virtual address that is not mapped to a physical address, a **page fault** occurs.

Virtual memory

1. Background

Virtual Memory:

Virtual memory is a technique that separates a process's **logical memory** (what the program sees) from **physical memory** (actual hardware RAM). This allows programmers to work with a **large logical memory space** without worrying about the limited size of physical memory.

- **Benefits**:
  - Provides an **illusion** of more memory than physically available.
  - Allows **multi-programming** by sharing physical memory among multiple processes.
  - Simplifies programming since developers don't need to manage memory constraints manually.

Virtual Address Space:
The **virtual address space** is the range of logical addresses assigned to a process. It represents how the process perceives memory, typically starting from a base address like 0 and appearing as contiguous.

- **Key Features**:
  - The program assumes its memory exists in a **single contiguous block**.
  - The operating system maps virtual addresses to physical addresses, often **non-contiguously**.
  - Portions of the process that aren't currently needed can be stored on the disk (backing store) and loaded into physical memory only when required.

Example:

- A process has a **virtual address space** of 4 GB.
- Physical memory is only **2 GB**.
- The OS ensures that the **active parts** of the process (e.g., the code currently running, critical data) are kept in physical memory, while inactive parts remain on the disk.

This allows a process to run as if it has the full 4 GB memory, even though the physical RAM is much smaller. The mapping between virtual and physical memory is handled by the **memory management unit (MMU)** with techniques like **paging** or **segmentation**.

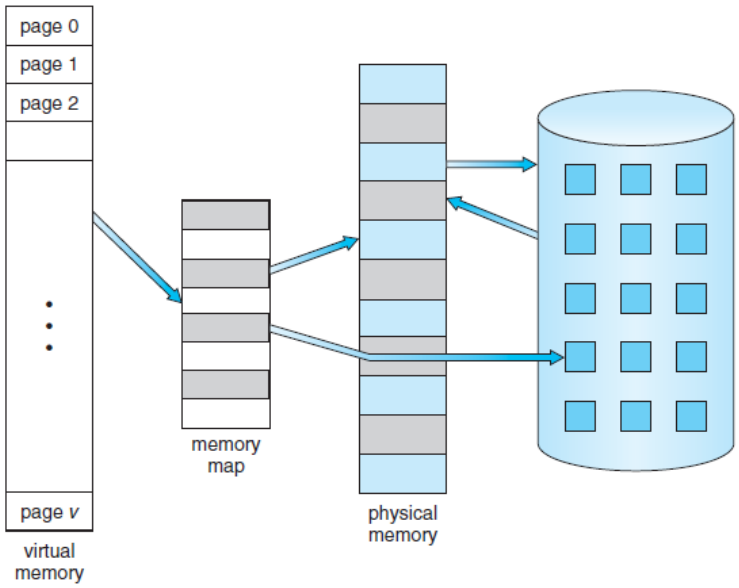
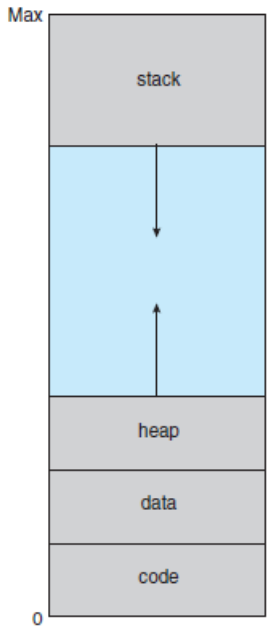


Diagram showing virtual memory that is larger than physical memory.

Note in the below Figure that we allow the heap to grow upward in memory as it is used for dynamic memory allocation.



Virtual address space.

Virtual Address Spaces

Virtual address space is the range of virtual addresses that an operating system gives to a process. It's a private space for each process, and other processes can't access it unless it's shared

Key Concepts

1. **Virtual vs. Physical Memory:**
   - **Virtual memory**: The memory addresses used by a program to access memory.
   - **Physical memory**: The actual RAM in the system.
   - A **mapping** process (managed by the operating system and hardware) translates virtual addresses to physical addresses.
2. **Address Space Layout:** A process's virtual address space is typically divided into regions:

- **Code Segment**: Contains executable instructions of the program.
- **Heap**: For dynamic memory allocation (grows upward).
- **Stack**: For function calls and local variables (grows downward).
- **Data Segment**: Contains global and static variables.
- **Memory-Mapped Files**: For libraries and files mapped into memory.

3. **Sparse Address Spaces:**
   - Parts of the virtual address space are "holes" with no physical memory mapping, allowing efficient memory usage.

Aspect	Logical Address	Virtual Address
Definition	Address generated by the CPU during program execution.	Address in the virtual memory space managed by the OS.
Scope	A subset of the virtual address space.	Includes logical addresses and unmapped regions (holes).
Generated By	CPU during instruction execution.	Operating system as part of virtual memory abstraction.
Visibility	Visible to the programmer or application.	Usually invisible to the programmer.
Usage	Used directly by the CPU to fetch instructions or data.	Used by the OS to manage memory allocation and mapping.
Mapping	Translated to physical address by the Memory Management Unit (MMU).	Maps to physical memory or swap space on disk.
Representation	Represents a valid memory location as seen by the program.	Represents a broader memory space, including unused areas.
Example Context	An instruction address in assembly or higher-level code.	An address in the entire process memory layout.
Errors	Invalid logical addresses cause segmentation faults.	Virtual addresses are mapped dynamically, reducing errors.

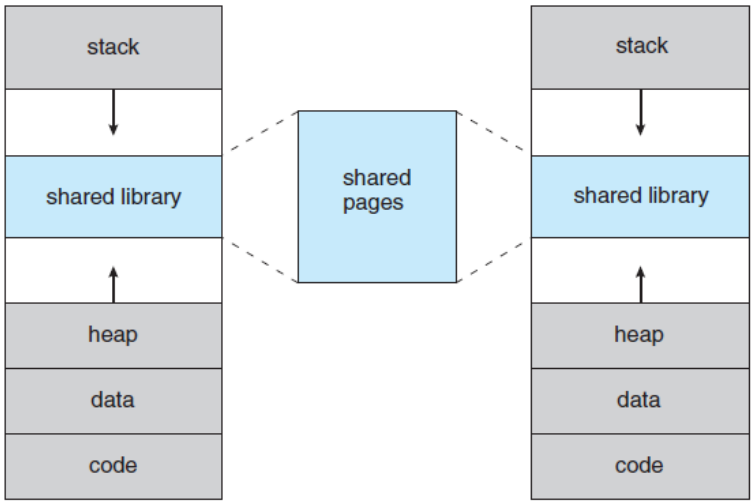
Similarly, we allow for the stack to grow downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as **sparse** address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution. In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through page sharing. This leads to the following benefits:

• System libraries can be shared by several processes through mapping of the shared object into a virtual address space. Although each process considers the libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes (Figure below). Typically, a library is mapped read-only into the space of each process that is linked with it.

• Similarly, processes can share memory. Two or more processes can communicate through the use of shared memory. Virtual memory allows one process to create a

region of memory that it can share with another process. Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared, much as is illustrated in below Figure.

• Pages can be shared during process creation with the fork() system call, thus speeding up process creation.



Shared library using virtual memory.

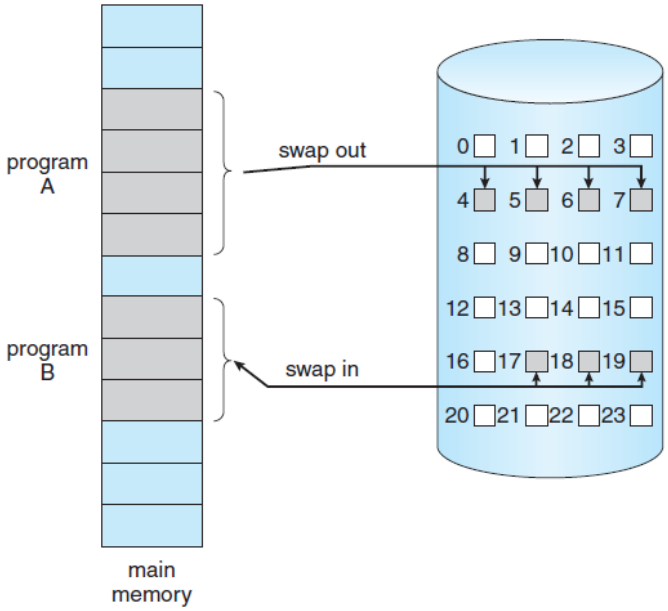
2. Demand Paging

When an executable program is loaded from disk into memory, one approach is to load the entire program into physical memory at the start of execution. However, this method can be inefficient because not all parts of the program may be immediately required. For example, if a program begins by displaying a list of options for the user to select from, loading the entire program would include executable code for all options, even if some are never used.

An alternative strategy is demand paging, a technique commonly employed in virtual memory systems. With demand-paged virtual memory, program pages are loaded into memory only when they are needed during execution. As a result, pages that are never accessed remain on the disk and are not loaded into physical memory.

A demand-paging system operates similarly to a paging system with swapping, where processes initially reside in secondary memory (typically a disk). When a process is executed, it is swapped into memory. However, unlike traditional swapping, demand paging employs a *lazy swapper*. This means pages are not brought into memory unless they are specifically needed.

In the context of demand paging, the term *swapper* is not entirely accurate, as a swapper typically handles entire processes. Instead, the term *pager* is used, as it manages the individual pages of a process, loading them into memory only when required.



Transfer of a paged memory to contiguous disk space.

1. Basic Concepts

When a process is swapped in, the pager predicts which pages are likely to be used before the process is swapped out again. Instead of loading the entire process into memory, the pager selectively loads only those pages it anticipates will be needed. This approach reduces both swap time and the physical memory required, as unnecessary pages are left on disk.

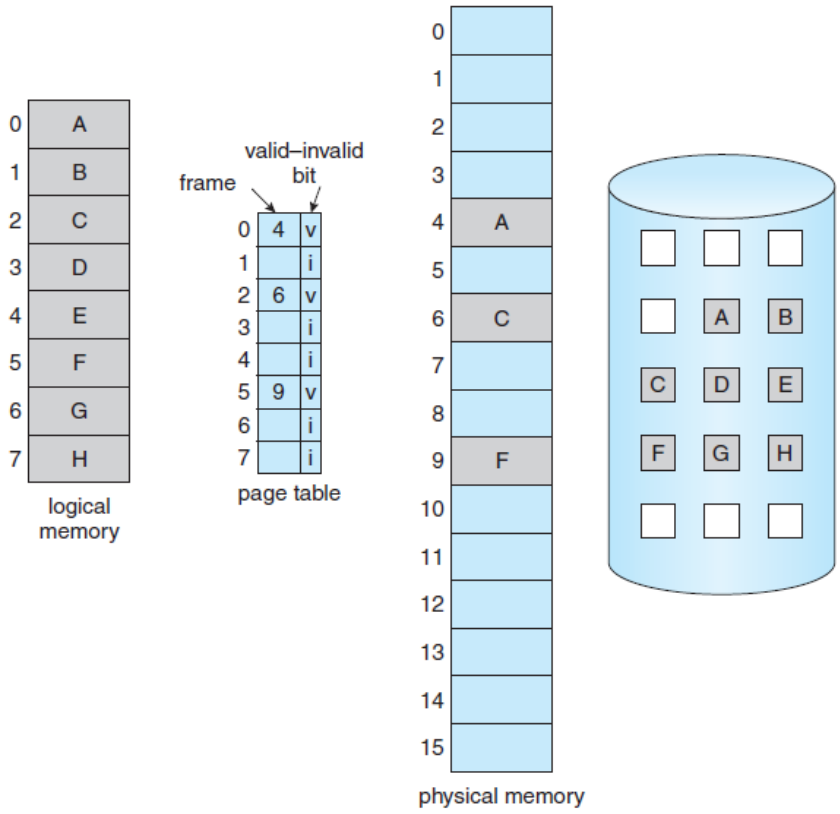
To implement this scheme, hardware support is needed to differentiate between pages in memory and those on disk. The valid–invalid bit mechanism serves this purpose. In this context, a "valid" bit indicates that the associated page is both part of the process's logical address space and is currently in memory. An "invalid" bit signifies either that the page is not part of the logical address space or that it is part of the address space but resides on the disk.

Valid–Invalid Bit:

- Each page table entry in the process's page table contains a **valid–invalid bit**:
  - **Valid Bit (1)**:
    - The page is in memory.
    - The page can be accessed directly without additional actions.
  - **Invalid Bit (0)**:
    - The page is not in memory but may exist on disk.
    - Accessing the page triggers a **page fault**, prompting the OS to fetch the page from disk.

For pages in memory, the page-table entry is configured as usual. For pages on disk, the page-table entry is either marked invalid or contains the disk address where the page is stored. This setup is illustrated in below Figure.

If the process does not attempt to access a page marked as invalid, no impact occurs. Consequently, if the pager accurately predicts and loads only the pages that will be used, the process runs as if all pages were initially loaded. During execution, as long as the process accesses memory-resident pages, it proceeds without interruption.



Page table when some pages are not in main memory.

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a **page fault**. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is straightforward (below Figure).

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.

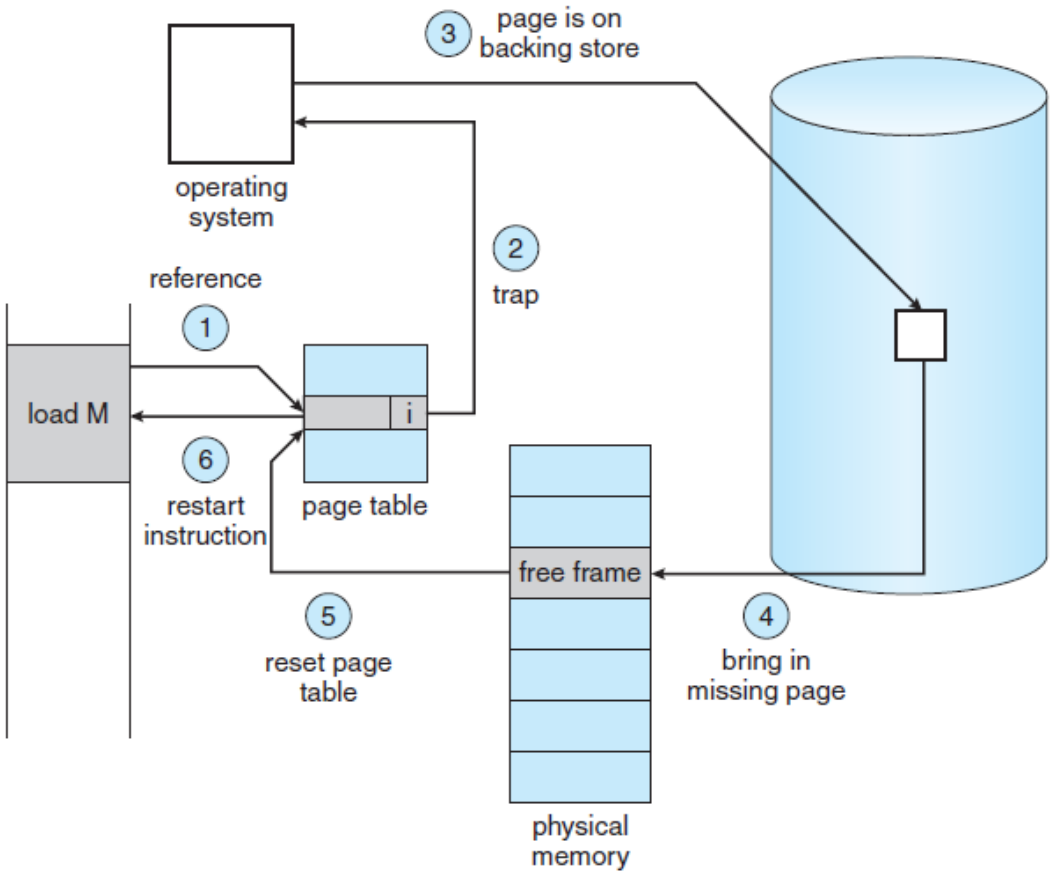
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.

3. We find a free frame (by taking one from the free-frame list, for example).

4. We schedule a disk operation to read the desired page into the newly allocated frame.

5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.



Steps in handling a page fault.

In the extreme case, a process can begin execution without any pages in memory. When the operating system sets the instruction pointer to the first instruction, which resides on a page not currently in memory, a page fault occurs. The operating system then loads this page into memory. As the process continues, it generates additional page faults as needed, with each required page being brought into memory. Once all necessary pages are loaded, the process can run without further faults. This approach is known as **pure demand paging**, as pages are only brought into memory when explicitly needed.

In theory, some programs could cause multiple page faults per instruction if each instruction accesses several new pages (one for the instruction and many for data). This would lead to poor performance due to excessive page faults. However, real-world programs typically exhibit **locality of reference** meaning that they tend to access a small set of memory pages repeatedly over time. This property helps demand paging provide acceptable system performance in practice.

The hardware required to support demand paging is similar to that for paging and swapping:

- **Page table**: This table holds entries that can be marked invalid using a valid–invalid bit or special protection bits to indicate whether a page is in memory or on disk.
- **Secondary memory**: This storage holds pages that are not in physical memory. It is typically a high-speed disk referred to as the **swap device**, and the portion of the disk used for this purpose is called **swap space**.

Aspect	Pure Demand Paging	Demand Paging
Definition	In pure demand paging, no page is loaded into memory at the start of program execution; every page is loaded only when referenced for the first time.	In demand paging, some pages (e.g., initial program instructions) may be pre-loaded into memory, while others are loaded on-demand.
Initial Memory State	Memory is empty at the start of the process; all pages are brought in only after a page fault.	Memory may contain some pre-loaded pages, reducing initial page faults.
Page Faults	Experiences a high number of page faults initially, as every required page triggers a fault.	Experiences fewer initial page faults if frequently used pages are pre-loaded.
Startup Time	Higher startup time due to the need to fetch every page from secondary storage.	Faster startup time as some necessary pages may already be loaded.
Efficiency	More efficient in terms of memory usage, as only actively used pages are loaded.	Balances memory usage and performance by pre-loading some frequently used pages.
Use Case	Suitable for systems where memory conservation is critical, and page access patterns are unpredictable.	Suitable for general-purpose systems where performance and startup time are important.

2. Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system. To see why, let's compute the **effective access time** for a demand-paged memory. For most computer systems, the memory-access time, denoted *ma*, ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from disk and then access the desired word.

Let p be the probability of a page fault ($0 \le p \le 1$). We would expect p to be close to zero—that is, we would expect to have only a few page faults. The **effective access time** is then

effective access time = $(1 - p) \times ma + p \times$ page fault time.

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
    a. Wait in a queue for this device until the read request is serviced.
    b. Wait for the device seek and/or latency time.
    c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).

9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

Not all of these steps are necessary in every case. For example, we are assuming that, in step 6, the CPU is allocated to another process while the I/O occurs. This arrangement allows multiprogramming to maintain CPU utilization but requires additional time to resume the page-fault service routine when the I/O transfer is complete.
In any case, we are faced with three major components of the page-fault service time:
1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

With an average page-fault service time of 8 milliseconds and a memory access time of 200 nanoseconds, the effective access time in nanoseconds is
effective access time = $(1 - p) \times (200) + p$ (8 milliseconds)
$= (1 - p) \times 200 + p \times 8{,}000{,}000$
$= 200 + 7{,}999{,}800 \times p.$
We see, then, that the effective access time is directly proportional to the **page-fault rate**. If one access out of 1,000 causes a page fault, the effective access time is 8.2 microseconds. The computer will be slowed down by a factor of 40 because of demand paging! If we want performance degradation to be less than 10 percent, we need to keep the probability of page faults at the following level:
$220 > 200 + 7{,}999{,}800 \times p,$
$20 > 7{,}999{,}800 \times p,$
$p < 0.0000025.$
That is, to keep the slowdown due to paging at a reasonable level, we can allow fewer than one memory access out of 399,990 to page-fault. In sum, it is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically.

A system has the following parameters:

- Memory access time = **250 nanoseconds**
- Page fault service time = **6 milliseconds (6,000,000 nanoseconds)**
- Page-fault rate = p

Find the effective access time (EAT) formula and calculate it if p=0.001 (1 page fault per 1,000 memory accesses).

Formula for Effective Access Time (EAT):

$$EAT = (1 - p) \times \text{Memory Access Time} + p \times \text{Page Fault Service Time}$$

Substituting the values:

$$EAT = (1 - p) \times 250 + p \times 6,000,000$$

$$EAT = 250 - 250p + 6,000,000p$$

$$EAT = 250 + (6,000,000 - 250)p$$

$$EAT = 250 + 5,999,750p$$

Calculate EAT for p=0.001
Substitute p=0.001

$$EAT = 250 + 5,999,750 \times 0.001$$

$$EAT = 250 + 5,999.75$$

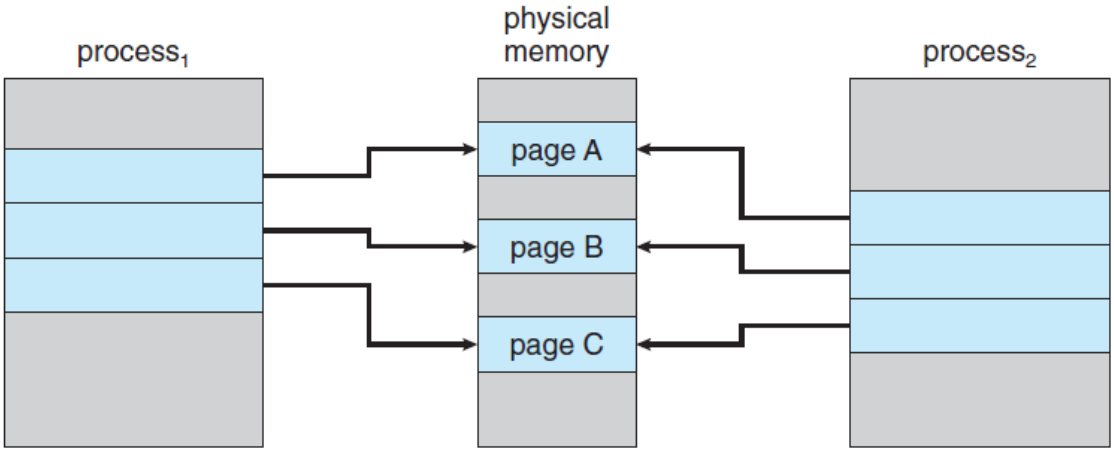
$$EAT = 6,249.75 \text{ nanoseconds}$$

- $1 \text{ millisecond} = 1,000 \text{ microseconds}$
- $1 \text{ microsecond} = 1,000 \text{ nanoseconds}$
- So, $1 \text{ millisecond} = 1,000,000 \text{ nanoseconds}$

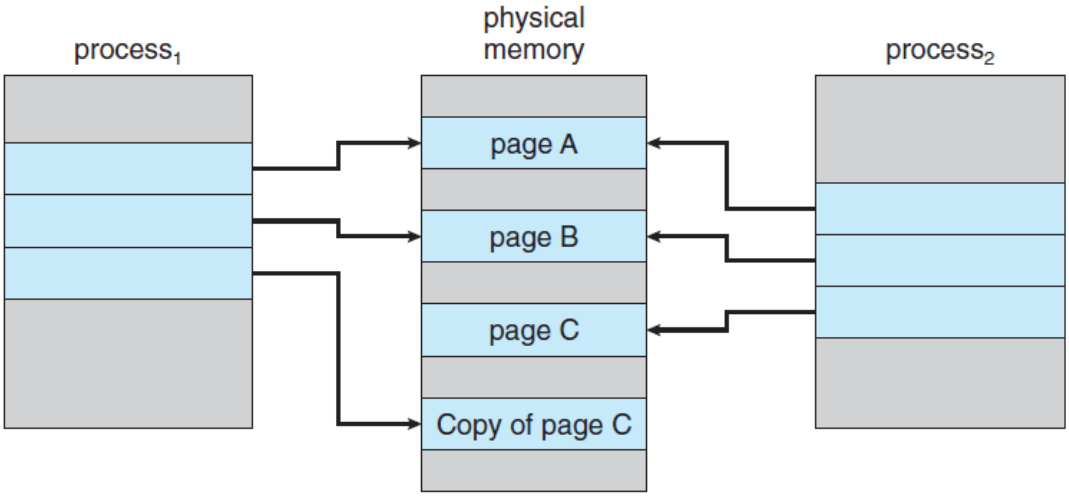
3. Copy-on-write

The fork() system call is used to create a child process that duplicates its parent process. Traditionally, this involved creating a complete copy of the parent's address space for the child, duplicating all the parent's memory pages. However, since many child processes often invoke the exec() system call immediately after creation, duplicating the parent's address space becomes unnecessary. To optimize this, a technique called **copy-on-write** is used. This approach allows the parent and child processes to initially share the same memory pages. These shared pages are marked as "copy-on-write," meaning that if either process attempts to modify a shared page, a separate copy of the page is created for that process.

For example, assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write. The operating system will create a copy of this page, mapping it to the address space of the child process. The child process will then modify its copied page and not the page belonging to the parent process. Obviously, when the copy-on-write technique is used, only the pages that are modified by either process are copied; all unmodified pages can be shared by the parent and child processes. Note, too, that only pages that can be modified need be marked as copy-on-write. Pages that cannot be modified (pages containing executable code) can be shared by the parent and child. Copy-on-write is a common technique used by several operating systems, including Windows XP, Linux, and Solaris.



Before process 1 modifies page C.



After process 1 modifies page C.

When it is determined that a page is going to be duplicated using copy on- write, it is important to note the location from which the free page will be allocated. Many operating systems provide a **pool** of free pages for such requests. These free pages are typically allocated when the stack or heap for a process must expand or when there are copy-on-write pages to be managed. Operating systems typically allocate these pages using a technique known as **zero-fill-on-demand**. Zero-fill-on-demand pages

have been zeroed-out before being allocated, thus erasing the previous contents. vfork() (for **virtual memory fork**)— operates differently from fork() with copy-on-write. With vfork(), the parent process is suspended, and the child process uses the address space of the parent. Because vfork() does not use copy-on-write, if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes. Therefore, vfork() must be used with caution to ensure that the child process does not modify the address space of the parent. vfork() is intended to be used when the child process calls exec() immediately after creation. Because no copying of pages takes place, vfork() is an extremely efficient method of process creation and is sometimes used to implement UNIX command-line shell interfaces.

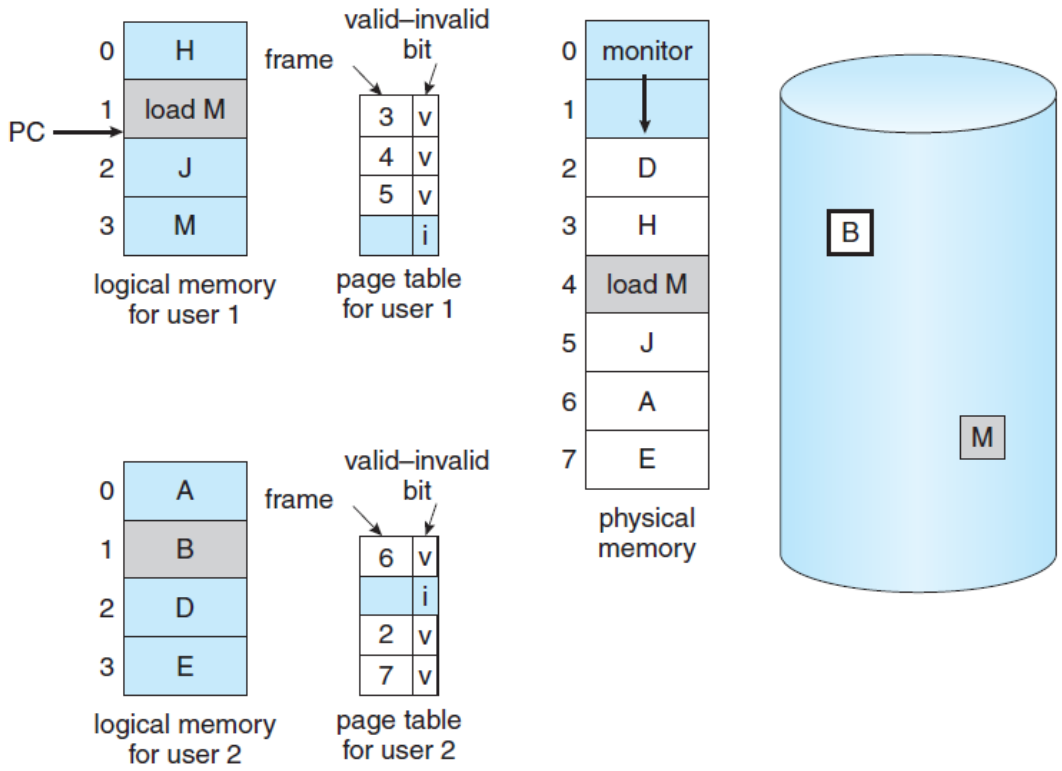
Aspect	`fork()`	`vfork()`
Address Space	Duplicates the parent's address space.	Shares the parent's address space.
Parent Behavior	Parent continues execution after `fork()`.	Parent is blocked until the child exits or calls `exec()`.
Performance	Higher overhead due to memory duplication.	More efficient for use cases involving `exec()`.
Use Case	General-purpose process creation.	Optimized for child processes immediately calling `exec()`.

4. Page replacement

Page replacement is a mechanism used in operating systems to manage memory when a process requests a page that is not currently in physical memory (i.e., a page fault occurs). If the memory is full, the operating system must select a page to remove (replace) from memory to make space for the requested page. This is crucial for efficient memory utilization in systems that use virtual memory.

Why Page Replacement is Needed

1. **Limited Physical Memory:** The physical memory (RAM) may not be large enough to hold all the pages a process requires.
2. **Demand Paging:** In demand paging, only the required pages are loaded into memory, and others are stored on disk. When memory is full, replacement ensures a balance between loaded and swapped-out pages



Need for page replacement.

Steps in Page Replacement

1. **Page Fault Occurs:** The requested page is not in physical memory.
2. **Page Selection:** A replacement algorithm decides which page to evict.
3. **Eviction:** The selected page is written back to disk (if modified) and removed from memory.
4. **Loading the New Page:** The requested page is brought into memory, and the page table is updated.

Page Replacement Algorithms

1. **First-In-First-Out (FIFO)**
   - Replaces the oldest page in memory.
   - Simple but can lead to Belady's Anomaly (more frames causing more page faults).
2. **Optimal Page Replacement**
   - Replaces the page that will not be used for the longest time in the future.
   - Provides the minimum page faults but is impractical as it requires future knowledge of page references.
3. **Least Recently Used (LRU)**
   - Replaces the page that has not been used for the longest time.
   - Approximates the optimal algorithm.

Over-allocation of Memory

Over-allocation of memory refers to a scenario where the operating system allocates more virtual memory to processes than the actual physical memory (RAM) available on the system.

This concept is closely tied to virtual memory management and relies on mechanisms like **paging** and **swapping** to ensure that processes can run effectively, even when memory demands exceed physical capacity.

How Over-allocation Works

1. **Virtual Memory System**: The operating system provides each process with a large, contiguous address space, regardless of the actual available RAM.
2. **Page Table Mapping**: Logical addresses are mapped to physical addresses using a page table. Pages that do not fit into RAM are stored on disk (swap space).
3. **Demand Paging**: Pages are loaded into RAM only when they are accessed. If the memory is full, page replacement techniques are used to free up space.

Benefits of Over-allocation

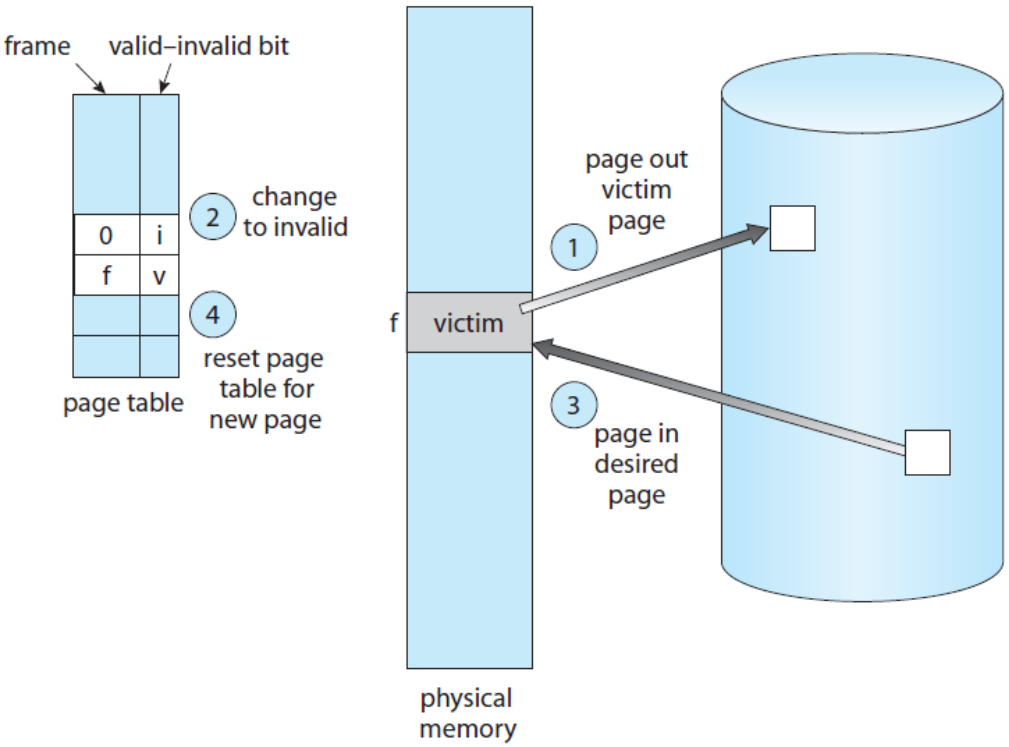
1. **Efficient Memory Utilization**: Allows the system to support more processes than the physical memory size.
2. **Increased Flexibility**: Processes can request large amounts of memory without being constrained by the physical RAM size.
3. **Better Resource Sharing**: Multiple processes can share memory resources dynamically.

Risks and Challenges

1. **Thrashing**: If too many processes are active and require frequent swapping, the system spends more time moving pages between disk and memory than executing processes.
2. **Performance Overhead**: Disk I/O is much slower than RAM, leading to significant delays if over-allocation is poorly managed.
3. **Out-of-Memory (OOM) Errors**: If the swap space is also exhausted, the operating system might terminate processes to free up memory.

Basic Page Replacement

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the age is no longer in memory (below Figure). We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:



Page replacement.

1. Find the location of the desired page on the disk.
2. Find a free frame:
    a. If there is a free frame, use it.
    b. If there is no free frame, use a page-replacement algorithm to select a **victim frame**.
    c. Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the user process from where the page fault occurred.

Notice that, if no frames are free, ***two*** page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.

We can reduce this overhead by using a **modify bit** (or **dirty bit**). When this scheme is used, each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk. If the modify bit is not set, however, the page has ***not*** been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there. This technique also applies to read-only pages (for example, pages

of binary code). Such pages cannot be modified; thus, they may be discarded when desired.

This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half ***if*** the page has not been modified. Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. With no demand paging, user addresses are mapped into physical addresses, and the two sets of addresses can be different. All the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory. If we have a user process of twenty pages, we can execute it in ten frames simply by using demand paging and using a replacement algorithm to find a free frame whenever necessary. If a page that has been modified is to be replaced, its contents are copied to the disk. A later reference to that page will cause a page fault. At that time, the page will be brought back into memory, perhaps replacing some other page in the process. We must solve two major problems to implement demand paging: we must develop a **frame-allocation algorithm** and a **page-replacement algorithm**. That is, if we have multiple processes in memory, we must decide how many frames to allocate to each process; and when page replacement is required, we must select the frames that are to be replaced. Designing appropriate algorithms to solve these problems is an important task, because disk I/O is so expensive. Even slight improvements in demand-paging methods yield large gains in system performance.

We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string**.

1. **FIFO Page Replacement**
2. **Optimal Page Replacement**
3. **Least recently used (LRU) algorithm.**

   The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is ***how*** to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

Counters Approach:

- Each page entry in the page table has a **time-of-use** field.
- A **logical clock or counter** in the CPU increases with every memory access.
- When a page is accessed, the current clock value is saved in the page's time-of-use field.
- To replace a page, the operating system searches for the page with the **smallest time value** (the one used the longest time ago).
- **Downside**: Searching for the least recently used (LRU) page and updating the clock for each access can significantly slow down processes.

Stack Approach:

- A **stack** is used to track page numbers.
- When a page is accessed, it is **moved to the top** of the stack.
- The **most recently used page** is always at the top, and the **least recently used page** is at the bottom.
- Pages can be replaced quickly by removing the page at the bottom of the stack (tail pointer).
- The stack is often implemented using a **doubly linked list** to make updates efficient.
- **Downside**: Updating the stack requires changing multiple pointers, making it slightly more complex.

1. Both methods ensure the **Least Recently Used (LRU)** page is replaced.
2. These approaches avoid **Belady's anomaly**, where increasing memory frames causes more page faults.
3. **Hardware support** is critical to make these methods practical. Without it, frequent updates would slow memory operations significantly.

Real-World Context:

Imagine a bookshelf where you track which books are used.

- **Counter Approach**: Write the last time each book was read on its cover, and remove the oldest one when space is needed.
- **Stack Approach**: Move the most recently read book to the top of the stack. The least read book is always at the bottom and removed when necessary.

The Optimal replacement, LRU replacement does not suffer from Belady's anomaly. Both belong to a class of page-replacement algorithms, called stack algorithms, that can never exhibit Belady's anomaly. A stack algorithm is a algorithm for which it can be shown that the set of pages in memory for n frames is always a *subset* of the set of pages that would be in memory with $n + 1$ frames.

Stack Algorithms are a class of page-replacement algorithms that follow a key property:

The set of pages in memory for n frames is always a subset of the set of pages in memory for n+1 frames.

This means that if a process has n+1n+1n+1 frames available, it will contain at least the same pages that it would have with nnn frames, plus potentially some additional ones. As a result, adding more frames will never increase the number of page faults, which ensures that stack algorithms do not exhibit Belady's Anomaly.

Let's illustrate the subset property of stack algorithms using the Least Recently Used (LRU) page replacement algorithm as an example.

Example

Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Frames: We consider n=3 and n+1=4

Case 1: n=3n = 3n=3 Frames

1. Initially empty frames:
   Reference 1 → Pages: {1} (Page fault)
   Reference 2 → Pages: {1, 2} (Page fault)
   Reference 3 → Pages: {1, 2, 3} (Page fault)
2. Page replacement starts:
   Reference 4 → Pages: {2, 3, 4} (Page fault, evict 1)
   Reference 1 → Pages: {1, 3, 4} (Page fault, evict 2)
   Reference 2 → Pages: {1, 2, 4} (Page fault, evict 3)
   Reference 5 → Pages: {2, 4, 5} (Page fault, evict 1)
3. No more page faults for repeated references within the frames:
   Reference 1, 2, 3, 4, 5 → Frames updated as needed.

Case 2: n+1=4 Frames

1. Initially empty frames:
   Reference 1 → Pages: {1} (Page fault)
   Reference 2 → Pages: {1, 2} (Page fault)
   Reference 3 → Pages: {1, 2, 3} (Page fault)
   Reference 4 → Pages: {1, 2, 3, 4} (Page fault)
2. No evictions yet because more frames are available:
   Reference 1 → Pages: {1, 2, 3, 4} (No page fault)
   Reference 2 → Pages: {1, 2, 3, 4} (No page fault)
3. Page replacement starts only after all frames are filled:
   Reference 5 → Pages: {2, 3, 4, 5} (Page fault, evict 1)
   Reference 1, 2, 3, 4, 5 → Frames updated as needed.

Verification of Subset Property

- Frames with n=3: {2, 4, 5} at the end.
- Frames with n + 1=4: {2, 3, 4, 5} at the end.

Clearly, the set of pages in memory with n=3 ({2, 4, 5}) is a subset of the set of pages in memory with n + 1=4 ({2, 3, 4, 5}).

LRU-Approximation Page Replacement

Few computer systems provide sufficient hardware support for true LRU page replacement. In fact, some systems provide no hardware support, and other page-replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a reference bit. The **reference bit** for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.

Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the order of use. This information is the basis for many page-replacement algorithms that approximate LRU replacement.

Additional-Reference-Bits Algorithm

We can gain additional ordering information by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, for example, then the page has not been used for eight time periods. A page that is used at least once in each period has a shift register value of 11111111. A page with a history register value of 11000100 has been used more recently than one with a value of 01110111. If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced. Notice that the numbers are not guaranteed to be unique, however. We can either replace (swap out) all pages with the smallest value or use the FIFO method to choose among them. The number of bits of history included in the shift register can be varied, of course, and is selected (depending on the hardware available) to make the updating as fast as possible. In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the **second-chance page-replacement algorithm**.

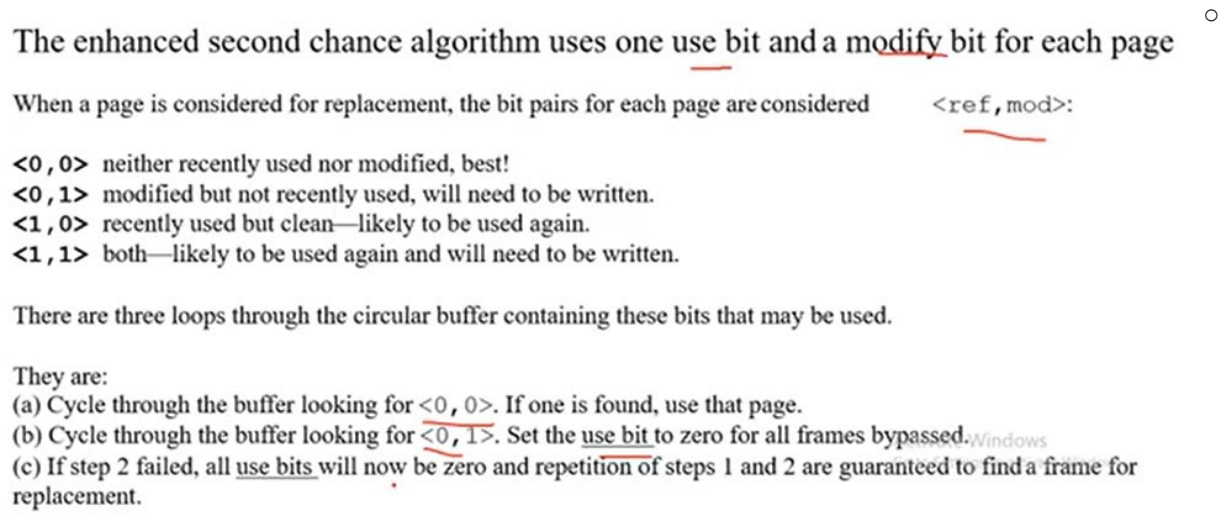
LRU Approximation Algorithms

LRU (Least Recently Used) page replacement is an effective algorithm but can be expensive to implement due to the need to track the precise order of memory references. LRU approximation algorithms are designed to approximate the behaviour of LRU with reduced overhead. These algorithms use hardware support and simpler data structures to estimate which pages are least recently used.

Types of LRU Approximation Algorithms

1. **Reference Bit Algorithm**
    - Each page has a reference bit (initially set to 0).
    - When a page is accessed, its reference bit is set to 1.
    - At regular intervals, the operating system resets reference bits to 0.
    - Pages with a reference bit of 0 are considered candidates for replacement as they are less likely to have been recently used.
2. **Second-Chance Algorithm (Clock Algorithm)**
    - Pages are arranged in a circular queue (like a clock).
    - Each page has a reference bit.
    - When a replacement is needed, the algorithm scans the queue:
        - If a page's reference bit is 0, it is replaced.
        - If the reference bit is 1, it is cleared, and the page is given a "second chance" as the algorithm moves to the next page.

3. **Enhanced Second-Chance Algorithm**
    - Extends the second-chance algorithm by using both the reference bit and a modification (dirty) bit.
    - Pages are classified into four categories:
        - Not recently used, not modified.
        - Not recently used, modified.
        - Recently used, not modified.
        - Recently used, modified.
    - Pages are replaced in the order of their category, prioritizing those that are neither recently used nor modified.



The enhanced second chance algorithm uses one use bit and a modify bit for each page

When a page is considered for replacement, the bit pairs for each page are considered <ref,mod>:

<0,0> neither recently used nor modified, best!
<0,1> modified but not recently used, will need to be written.
<1,0> recently used but clean—likely to be used again.
<1,1> both—likely to be used again and will need to be written.

There are three loops through the circular buffer containing these bits that may be used.

They are:
(a) Cycle through the buffer looking for <0, 0>. If one is found, use that page.
(b) Cycle through the buffer looking for <0, 1>. Set the use bit to zero for all frames bypassed.
(c) If step 2 failed, all use bits will now be zero and repetition of steps 1 and 2 are guaranteed to find a frame for replacement.

The additional reference bit approach faces challenges related to complexity, overhead, and scalability. The **Second-Chance Algorithm** overcomes these issues by offering a lightweight, hardware-compatible, and efficient mechanism for approximating LRU behaviour while avoiding the pitfalls of excessive bookkeeping or computational overhead.

A **dirty page** refers to a page in memory that has been modified (written to) but has not yet been written back to the disk (or secondary storage). In the context of virtual memory management, when a page is swapped out of physical memory, if it is "dirty," the operating system must ensure that the page is written back to its corresponding location in the backing store before being replaced or swapped out again.

Aspect	Additional Reference Bit Algorithm	Second-Chance Algorithm
Purpose	Approximation of the Least Recently Used (LRU) algorithm.	Simpler approximation of LRU using a single reference bit.
Reference Bits Used	Uses multiple bits (e.g., 8 bits) per page to track detailed usage history.	Uses a single reference bit per page.
Implementation Complexity	Requires periodic shifting and updating of the reference bit vectors for all pages.	Simpler to implement; evaluates pages in a circular queue.
Memory Overhead	Requires more memory to store multiple reference bits for each page.	Minimal memory overhead as only one bit per page is used.
Computational Overhead	Higher due to the need for shifting and comparing bit-vectors.	Lower as it only checks and resets a single bit per page.
Efficiency in Large Systems	Less scalable due to the need for maintaining and updating multiple bits.	Scales better as it operates efficiently with a single bit.
Decision Process	Replacement decisions depend on the binary patterns of reference bits.	Replacement decisions are made by giving pages a "second chance."
Replacement Order	Based on the detailed history captured by the bit vectors.	Follows a circular queue, resetting the reference bit if set.
Hardware Dependency	May require additional hardware support for multiple reference bits.	Works with standard reference bits already supported in hardware.
Use Cases	Suitable when precise tracking of page usage is essential.	Ideal for systems needing a simple and practical LRU approximation.

L6.47: LRu Approximation | Reference Bit | Enhanced Reference Bit - YouTube

Page Replacement Algorithms-III - YouTube

Lec52:Second Chance Page Replacement Algorithm. - YouTube

ENHANCED SECOND CHANCE page replacement algorithm(11)

Enhanced Second Chance Algorithm - YouTube

Aspect	Second-Chance Algorithm	Enhanced Second-Chance Algorithm
Reference Bits Used	Uses a single reference bit per page.	Uses both reference and modification (dirty) bits for decision-making.
Decision Criteria	Checks the reference bit only to determine whether a page gets a "second chance."	Considers both reference and modification bits to prioritize replacement.
Replacement Order	Pages are replaced in a circular queue based on reference bit status.	Pages are replaced based on a priority matrix of reference and modification bits.
Efficiency	Simpler but may not always replace the least useful page.	More efficient as it uses additional information (dirty bit) to make better decisions.
Complexity	Simple to implement with minimal overhead.	Slightly more complex due to the need to track and evaluate the dirty bit.
Handling Dirty Pages	Treats all pages equally without considering modification status.	Gives lower priority to dirty pages (defers their replacement).
Replacement Priority	Based solely on whether the reference bit is set.	Pages with `R = 0` and `M = 0` are replaced first, followed by other combinations.
Memory Overhead	Minimal memory usage (only one bit per page).	Requires additional memory for tracking the dirty bit.
Use Cases	Suitable for simple systems with moderate memory demands.	Ideal for systems with a mix of frequently accessed and modified pages.
Effectiveness	May lead to suboptimal replacements if dirty pages are not accounted for.	More effective in reducing disk I/O by prioritizing clean pages for replacement.

Counting-Based Page Replacement

There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes.

- The **least frequently used (LFU)** page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

Least Frequently Used (LFU) Page Replacement Algo - YouTube

- The **most frequently used (MFU)** page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Most Frequently Used MFU Page Replacement Algorithm

5. Allocation of frames

How do we allocate the fixed amount
of free memory among the various processes? If we have 93 free frames and two processes, how many frames does each process get?

6. Thrashing

Thrashing is a situation in a computer system where excessive paging or swapping of data between the main memory (RAM) and disk (secondary storage) severely degrades system performance. It occurs when the operating system spends more time swapping data in and out of memory than executing the actual processes. Thrashing occurs when the operating system starts swapping pages in and out of memory too often, leading to severe performance issues. Here's a simplified breakdown of how it happens:

- **Increasing CPU Utilization**: The operating system monitors how much the CPU is being used. If CPU usage becomes too low, the system adds more processes to keep the CPU busy.

- **Page Faults and Memory Strain**: When a process enters a new phase that needs more memory, it starts requesting more pages (frames). These requests cause "page faults." However, there is limited memory, so other processes lose their pages to meet the demand. As a result, these other processes also experience page faults, causing more paging and swapping in and out of memory.

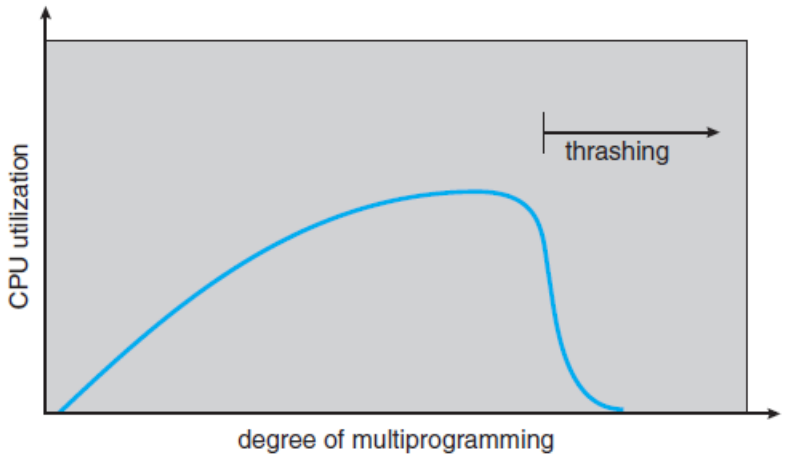
- **Queue for the Paging Device**: As more processes wait for pages to be swapped, they queue up for the paging device. This delays the processes, and the CPU ends up sitting idle more often.

- **Increase in Degree of Multiprogramming**: Seeing the CPU become idle, the system decides to add more processes to the system (increasing the degree of multiprogramming). This results in even more page faults because the new processes take up memory, causing more swapping.

- **Cycle of Thrashing**: This cycle continues, where processes fault, wait for the paging device, and the system keeps adding more processes to the system. As a result, CPU utilization drops drastically because the CPU is waiting for memory pages instead of performing useful tasks.

- **System Performance Drops**: Eventually, the system becomes overwhelmed with page faults and memory swapping. The page-fault rate increases significantly, and the

effective access time for memory also increases. No meaningful work is being done because the system is too busy swapping data.



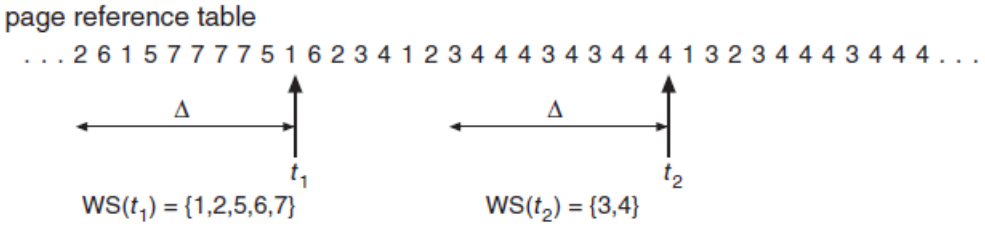
Thrashing.

How to Limit Thrashing

1. Local Replacement Algorithm: In this approach, each process manages its own memory. If one process starts thrashing, it can't steal frames from another process. However, the problem isn't fully solved, because even if a process is not thrashing, it might still experience delays in accessing pages due to the long queues for the paging device.
2. **Working-Set Model**

The **Working-Set Model** is based on the idea of locality in program execution, where a program tends to repeatedly use a small set of pages (memory locations) over a short period of time. This model helps optimize memory allocation and prevent **thrashing** by monitoring how many pages a process actively uses.

- It is based on the assumption of locality
- It uses △ , working set window.
- It examines the most recent page reference
- If the page is in active use it will be in the working set window.
  If it is no longer being used, it will drop from the working set window.



Working-set model.

How It Works

1. **Working-Set Window (Δ)**:
    - The working set is determined by a parameter Δ, which defines a "window" of the most recent page references made by the process.
    - The working set consists of the pages that were referenced within the last Δ page references. For example, if Δ = 10 memory references, and a process accesses pages 1, 2, 5, 6, and 7, those pages would be part of the working set at that time.
    - As time progresses, the working set changes, dropping pages that are no longer being used.
2. **Working Set Size**:
    - The size of the working set (WSSi) is the number of frames (pages) a process is actively using within its working set.
    - For example, if a process needs 5 pages actively, its working-set size is 5.
3. **Calculating Total Demand**:
    - The **total demand for frames (D)** is the sum of the working-set sizes of all processes in the system. If the total demand exceeds the total available frames (memory), **thrashing** occurs, because processes cannot keep their working sets in memory.
4. **Managing Working Sets**:
    - The operating system monitors the working set of each process and ensures that each process has enough frames for its working set. If enough extra frames are available, the system can launch more processes.
    - If the total working-set size exceeds the total available memory, the OS may **suspend** a process. This means swapping out its pages to free up memory for other processes. The suspended process can be restarted later.

Advantages of the Working-Set Model

- **Prevents Thrashing**: By ensuring each process gets enough memory for its working set, the system avoids thrashing, where processes are spending all their time swapping pages instead of executing.
- **Maximizes CPU Utilization**: The system can keep the degree of multiprogramming high while preventing excessive page swapping.

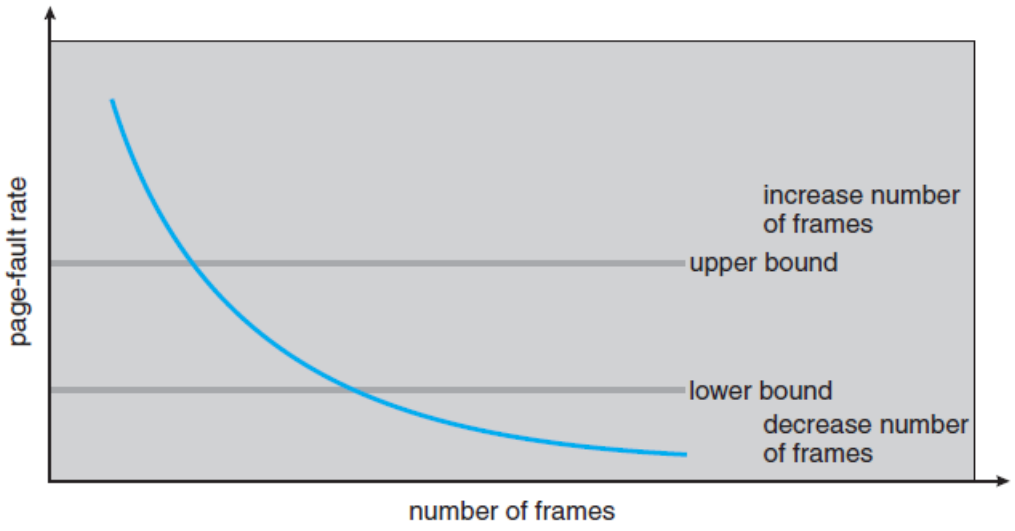
Implementation Challenges

- **Tracking the Working Set**: The working-set window is dynamic—it shifts as the program runs. For every memory access, the window "moves" to include the new reference and drop the oldest reference.
- **Approximating the Working Set**: It can be hard to track exactly which pages are in the working set. The working-set window is typically approximated using a timer interrupt and reference bits:
    - **Timer Interrupt**: A timer interrupt can be set to occur every few thousand references. When the interrupt happens, the reference bits are copied and cleared.
    - **Reference Bits**: Each page has a reference bit, indicating if the page was accessed in the last window of time. If any reference bit is set within a certain period (e.g., 10,000 references), that page is considered part of the working set.

This method isn't perfectly accurate, but it's a good approximation. More frequent interrupts or more bits could improve accuracy but would also increase the overhead of managing the working set.

3. **Page-Fault Frequency (PFF)** strategy is an alternative approach to preventing **thrashing**, which focuses directly on controlling the **page-fault rate** of processes rather than tracking their working set. This strategy simplifies managing memory by adjusting the number of frames allocated to processes based on their page-fault behaviour.

The working-set model is successful, and knowledge of the working set can be useful for prepaging, but it seems a clumsy way to control thrashing. A strategy that uses the **page-fault frequency (PFF)** takes a more direct approach. The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page-fault rate (below Figure). If the actual page-fault rate exceeds the upper limit, we allocate the process another frame. If the page-fault rate falls below the lower limit, we remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing. As with the working-set strategy, we may have to swap out a process. If the page-fault rate increases and no free frames are available, we must select some process and swap it out to backing store. The freed frames are then distributed to processes with high page-fault rates.



Page-fault frequency.

L6.50: Thrashing in OS | Working set window using example - YouTube

thrashing in operating system | Techniques to handle Thrashing | working set model - YouTube

Lec55.4: Working Set Model and Page Fault Frequency Model | Memory Management | OS - YouTube

Working Set Model and Page Fault Frequency| Thrashing | Memory Management - YouTube