Module 4: Memory Management

-by
Asst Prof Rohini M. Sawant

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

BASIC HARDWARE

- Memory consists of large array of words or 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 & storing to specific memory addresses.
- A typical instruction-execution cycle, first fetches an instruction from memory.
- After the instruction has been executed on the operands, results may be stored back in memory

BASIC HARDWARE

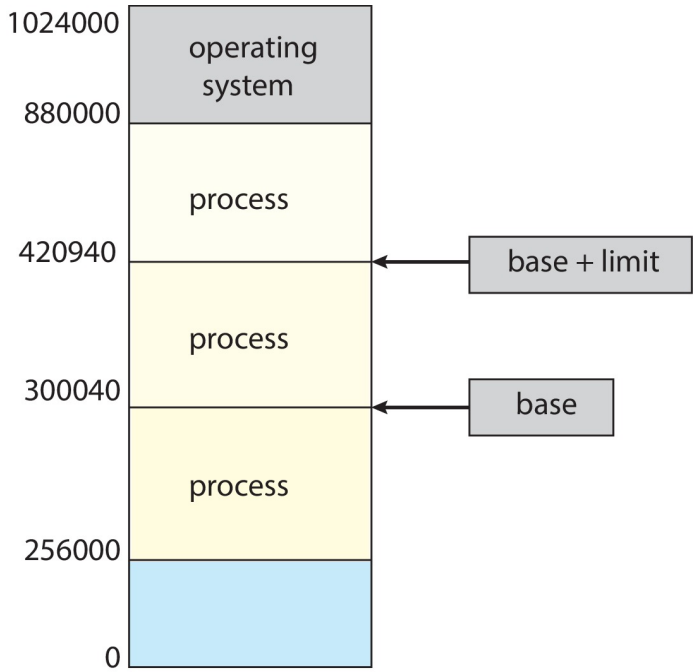
- The CPU can directly access only the Registers built into the processors and the Main Memory.
- There are machine instructions that take memory addresses as arguments, but none that take disk addresses.
- So any instructions in execution and any data being used by the instructions must be in one of these direct access storage devices.
- If the data are not in memory, they must be moved there before the CPU can operate on them.

PROTECTION

- The operating system has to be protected from access by user processes.
- In addition, user processes must be protected from one another.
- This protection must be provided by the hardware.
- The above protection can be done by ensuring that each process has a separate memory space.
- To do this, we need the ability to determine the range of legal addresses that the process may access & to ensure that the process can access only these legal address.
- For this we can use two registers- BASE & REGISTERS

PROTECTION

- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process

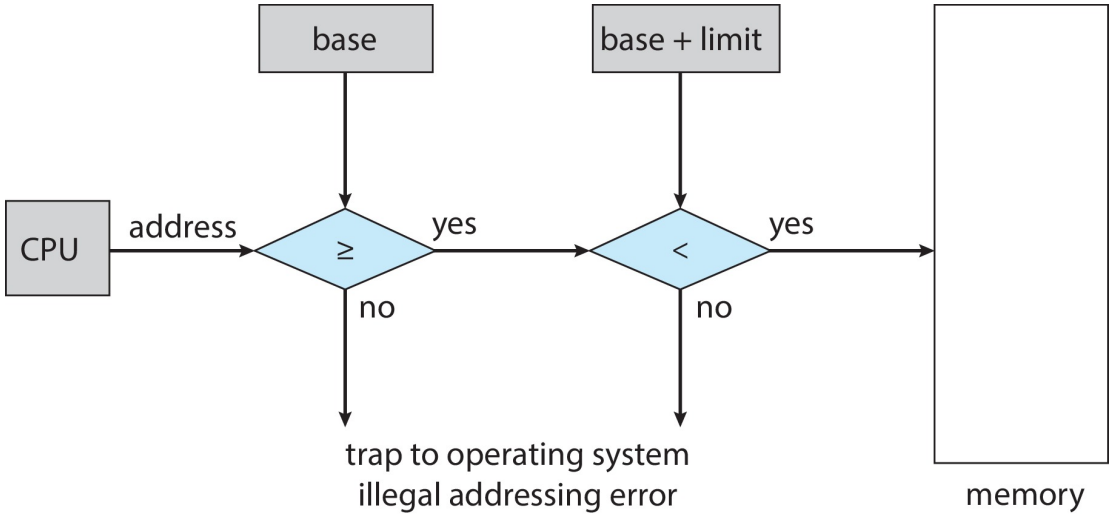


PROTECTION

- A base & limit register define a Logical Address Space.
- The base register holds the smallest legal physical memory address.
- The Limit register specifies the size of the range.
- For example, if the base register holds 30040 & limit register is 120900, then the program can legally access all addresses from 30040 through 420940 (inclusive).

Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



- The instructions to loading the base and limit registers are privileged

ADDRESS BINDING

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e., "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e., 74014
  - Each binding maps one address space to another

ADDRESS BINDING

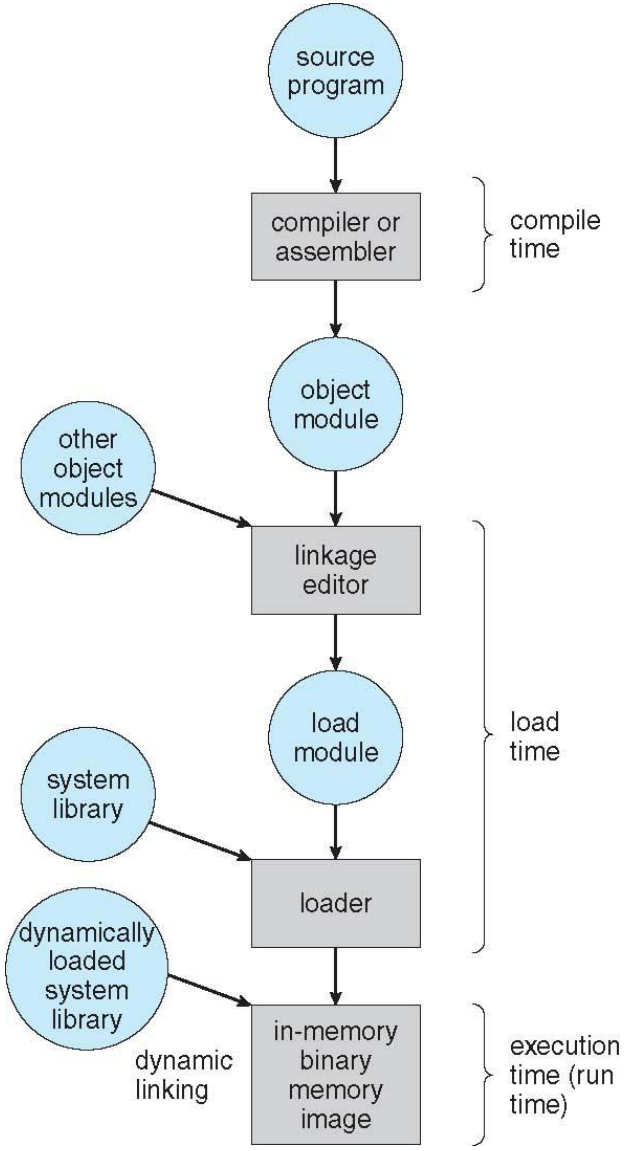
- Usually A Program resides on a disk as a Binary Executable File.
- To be executed, the program must be brought into memory & placed within a process.
- Depending on the memory management in the use, the process may be moved between disk & memory during its execution.
- The processes on the disk that are waiting to be brought into memory for execution form the input queue.
- Input Queue → Select a process → Loads it into Memory → Executes & Accesses instructions & data from Memory → Process terminates Memory space is declared available

ADDRESS BINDING

- Most system allow a user process to reside in any part of the physical memory.
- Though address space of the computer starts at 00000 at the first address of user process need not be 00000.
- In most cases, a process will go through several steps during COMPILE TIME, LOAD TIME & EXECUTION TIME before being executed.
- Addresses may be represented in different ways during these steps
- Source Program- Addresses are general symbolic(such as count).
- Compiler- typically binds these symbolic addresses to relocatable addresses (such as “14 bytes from the beginning of this module”)
- Linkage Editor or Loader- Binds the relocatable addresses to absolute addresses (such as 74014)
- Each binding is a mapping from one address space to another.

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

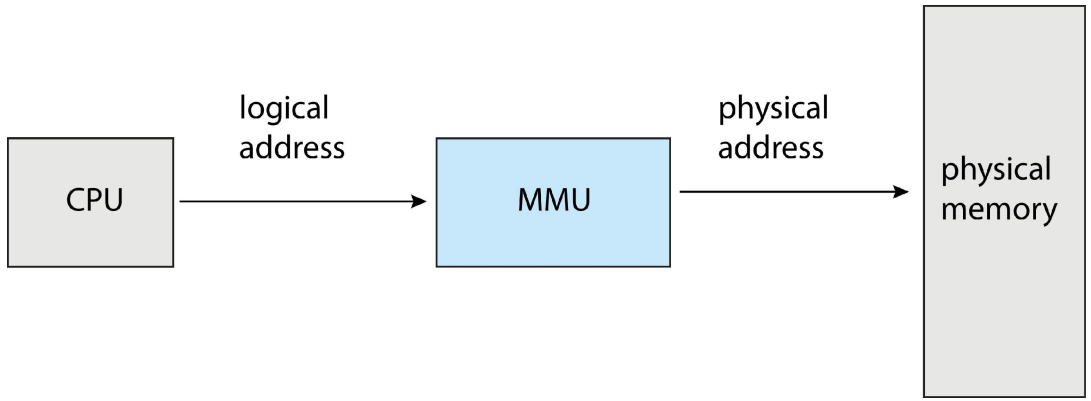


Logical vs Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

MEMORY MANAGEMENT UNIT

- Hardware device that at run time maps virtual to physical address



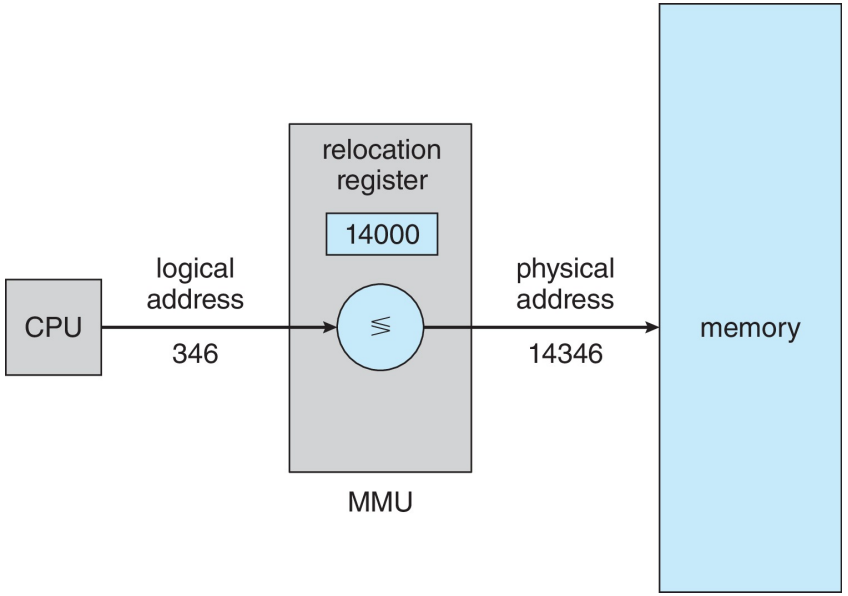
- Many methods possible, covered in the rest of this chapter

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

Memory-Management Unit (MMU)

- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory



We have two different types of addresses:

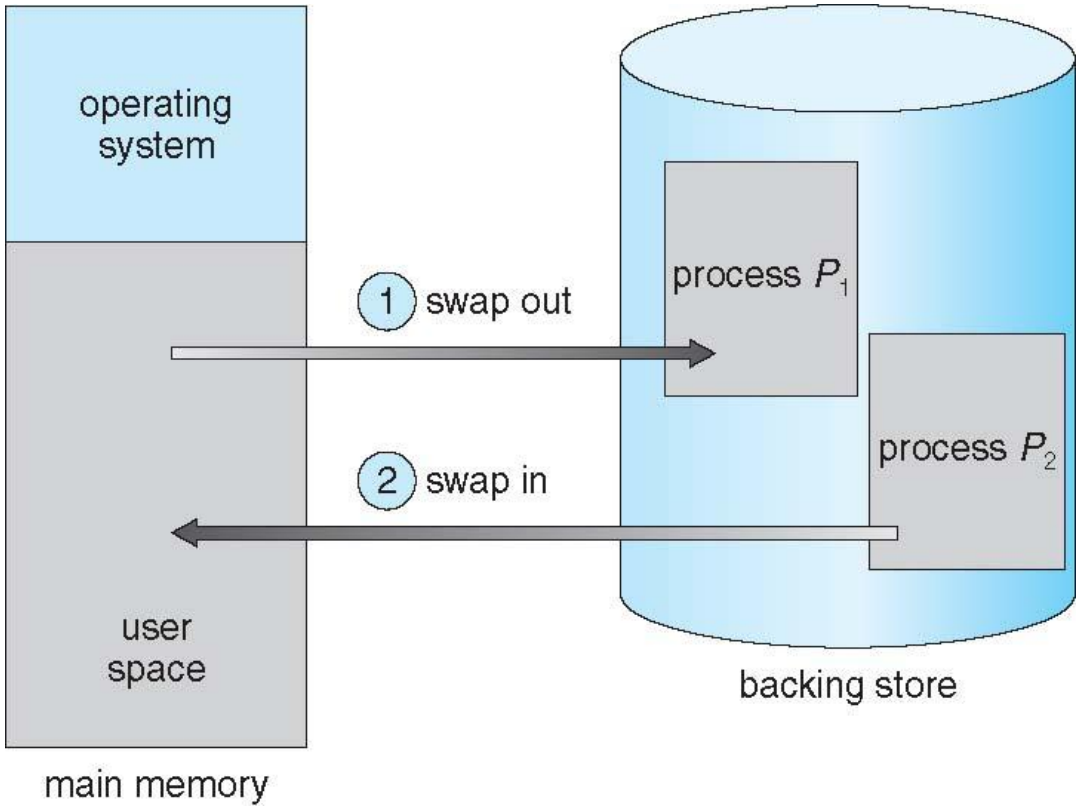
- Logical addresses (in the range of 0 to max)
- Physical addresses (in the range of R+0 to R+max for a base value R).
- The user generates only the logical addresses & thinks that the process runs in locations 0 to max.
- The user program supplies logical addresses, these logical addresses must be mapped to physical addresses before they are used.
- The base & limit registers can be loaded only by the OS, which uses a special privileged instructions.
- Since privileged instructions can be executed only in kernel mode, and since only the OS executes in kernel mode, only the OS can load the base & limit registers.
- This scheme allows the OS to change the value of the registers but prevents user programs from changing the registers contents

SWAPPING

- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

SWAPPING

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold



BACKING STORE

- Swapping requires a backing store to a fast disk.
- It must be large enough to accommodate copies of all memory images for all users.
- 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 in memory & are ready to run.
- Whenever the CPU Scheduler decides to executes 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, & if there is no free memory region, the dispatcher swaps out a process currently in memory & swaps in the desired process.
- It then reloads registers & transfers control to the selected process.

Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`
- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low

SWAP TIME

- The context switch time in a swapping system is fairly high.
- Example:
- Let User process size= 10 MB
- Transfer rate of Backing store= 40 MB per second
- Actual transfer of the 10 MB process to or From main memory takes-

=10000 KB/40000KB per second

=1/4 second = 250 millisecond

- Assuming that no head seeks are necessary, & assuming an average latency of 8 milliseconds, the swap time=258 ms.
- Since we must both swap out & swap in, the total swap time is = 258 x 2=516 ms

MEMORY ALLOCATION

- One of the simplest methods of allocating memory is to assign processes to variably sized partitions in memory, where each partition may contain exactly one process.
- In general, as mentioned, the memory blocks available comprise a set of holes of various sizes scattered throughout memory.
- When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts.
- One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes.
- If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.
- This procedure is a particular instance of the general dynamic storage allocation problem, which concerns how to satisfy a request of size n from a list of free holes.
- There are many solutions to this problem which is: The first-fit, best-fit , and worst-fit strategies

Dynamic Storage-Allocation Problem

- **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.

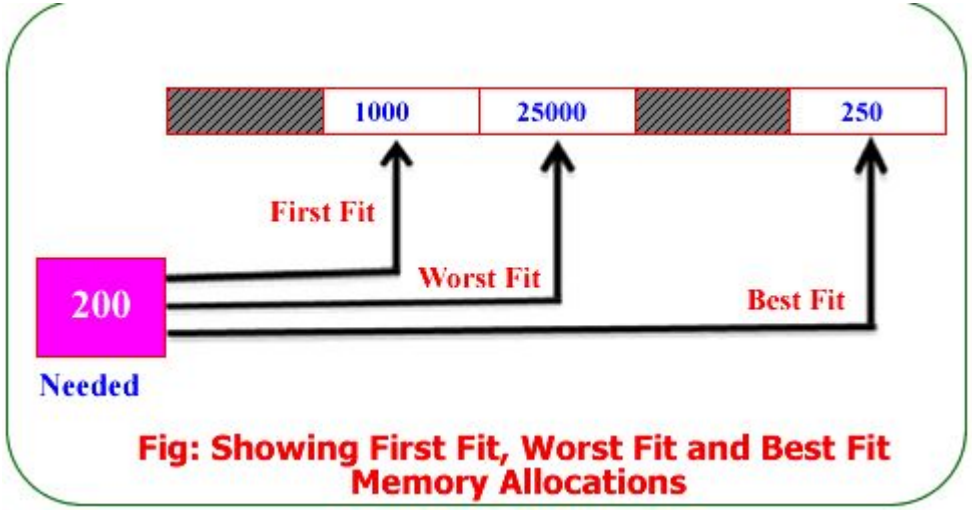


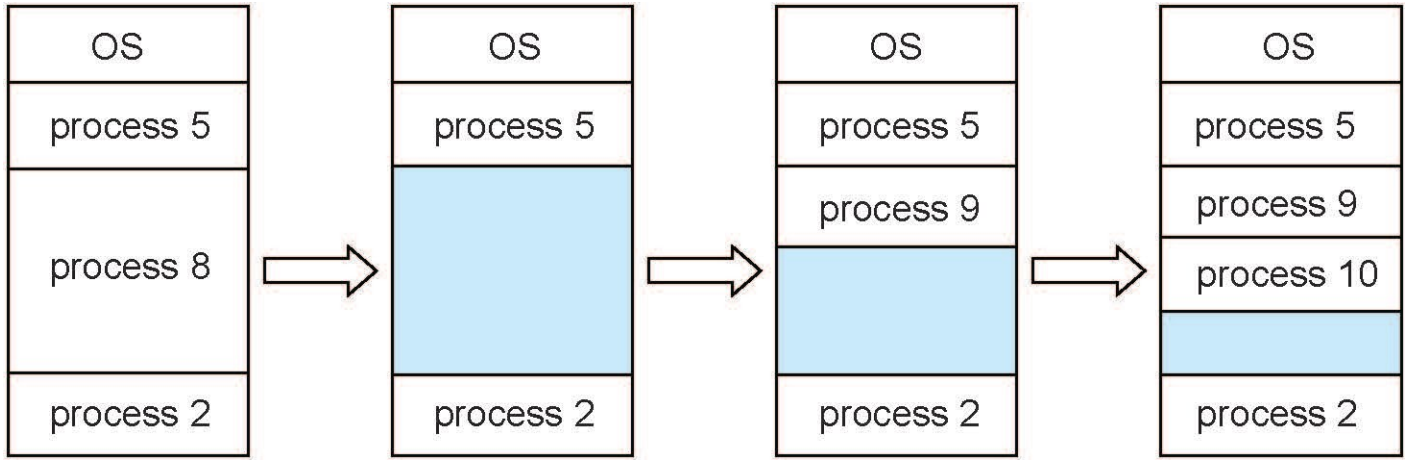
Fig: Showing First Fit, Worst Fit and Best Fit Memory Allocations

CONTIGUOUS ALLOCATION

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
- Base register contains value of smallest physical address
- Limit register contains range of logical addresses – each logical address must be less than the limit register
- MMU maps logical address *dynamically*

Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process' needs)
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  a) allocated partitions b) free partitions (hole)



NUMERICAL ON MEMORY ALLOCATION

- Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 122K, and 426K (in order)? Which algorithm makes the most efficient use of memory?

- **First-fit:**
  212k -> 500K (288 left)
  417k -> 600k (183 left)
  122k -> 288k (166k left)
  426k -> nowhere big enough left.

- **Best-fit:**
  212k -> 300k (88k left)
  417k -> 500k (83k left)
  122k -> 200k (88k left)
  426k -> 600k (174k left)

- **Worst-fit:**
  212k -> 600k (388k left)
  417k -> 500k (83k left)
  122k -> 388k (188k left)
  426k -> nowhere big enough again!

S.NO.	Fixed partitioning	Variable partitioning
1.	In multi-programming with fixed partitioning the main memory is divided into fixed sized partitions.	In multi-programming with variable partitioning the main memory is not divided into fixed sized partitions.
2.	Only one process can be placed in a partition.	In variable partitioning, the process is allocated a chunk of free memory.
3.	It does not utilize the main memory effectively.	It utilizes the main memory effectively.
4.	There is presence of internal fragmentation and external fragmentation.	There is external fragmentation.
5.	Degree of multi-programming is less.	Degree of multi-programming is higher.
6.	It is more easier to implement.	It is less easier to implement.
7.	There is limitation on size of process.	There is no limitation on size of process.

FRAGMENTATION

- Fragmentation is an unwanted problem in the operating system in which the processes are loaded and unloaded from memory, and free memory space is fragmented.
- Processes can't be assigned to memory blocks due to their small size, and the memory blocks stay unused.
- It is also necessary to understand that as programs are loaded and deleted from memory, they generate free space or a hole in the memory.
- These small blocks cannot be allotted to new arriving processes, resulting in inefficient memory use.
- As the process is loaded and unloaded from memory, these areas are fragmented into small pieces of memory that cannot be allocated to incoming processes. It is called fragmentation.
- User processes are loaded and unloaded from the main memory, and processes are kept in memory blocks in the main memory.
- Many spaces remain after process loading and swapping that another process cannot load due to their size.
- Main memory is available, but its space is insufficient to load another process because of the dynamical allocation of main memory processes.

Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

S.NO	Internal fragmentation	External fragmentation
1.	In internal fragmentation fixed-sized memory, blocks square measure appointed to process.	In external fragmentation, variable-sized memory blocks square measure appointed to the method.
2.	Internal fragmentation happens when the method or process is smaller than the memory.	External fragmentation happens when the method or process is removed.
3.	The solution of internal fragmentation is the best-fit block.	The solution to external fragmentation is compaction and paging.
4.	Internal fragmentation occurs when memory is divided into fixed-sized partitions.	External fragmentation occurs when memory is divided into variable size partitions based on the size of processes.
5.	The difference between memory allocated and required space or memory is called Internal fragmentation.	The unused spaces formed between non-contiguous memory fragments are too small to serve a new process, which is called External fragmentation.
6.	Internal fragmentation occurs with paging and fixed partitioning.	External fragmentation occurs with segmentation and dynamic partitioning.
7.	It occurs on the allocation of a process to a partition greater than the process's requirement. The leftover space causes degradation system performance.	It occurs on the allocation of a process to a partition greater which is exactly the same memory space as it is required.
8.	It occurs in worst fit memory allocation method.	It occurs in best fit and first fit memory allocation method.

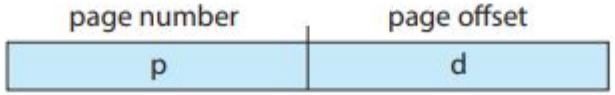
PAGING

- Paging is a memory management scheme that permits a process's physical address space to be noncontiguous.
- Paging avoids external fragmentation and the associated need for compaction, two problems that plague contiguous memory allocation.
- Because it offers numerous advantages, paging in its various forms is used in most operating systems, from those for large servers through those for mobile devices.
- Paging is implemented through cooperation between the operating system and the computer hardware.

PAGING

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames. This rather simple idea has great functionality and wide ramifications. For example, the logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 2^{64} bytes of physical memory.

Every address generated by the CPU is divided into two parts: a **page number** (p) and a **page offset** (d):



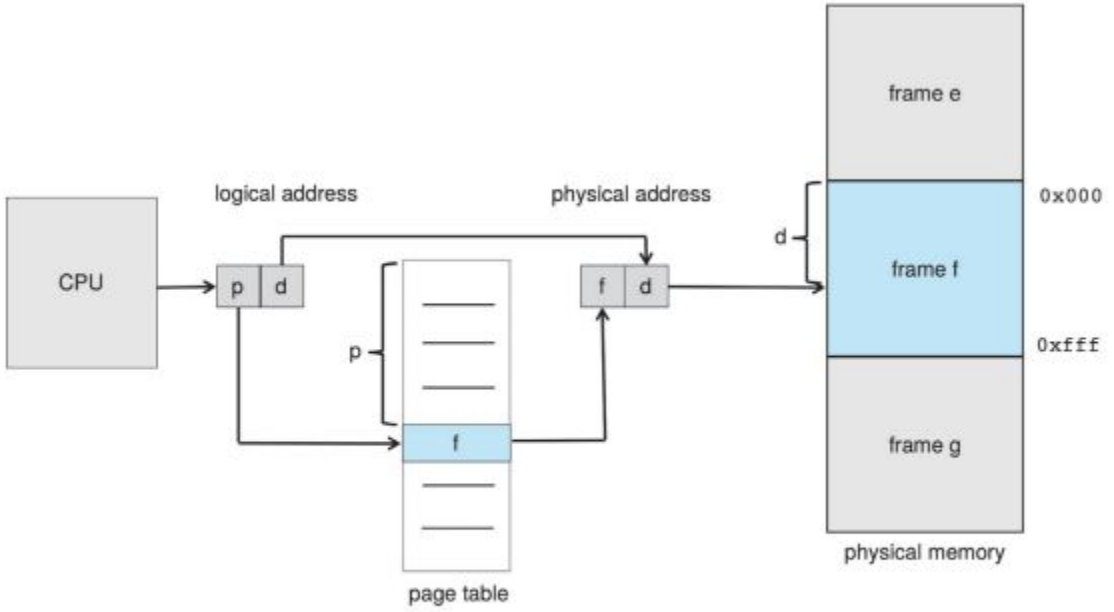


Figure 9.8 Paging hardware.

The page number is used as an index into a per-process **page table**. This is illustrated in Figure 9.8. The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being referenced. Thus, the base address of the frame is combined with the page offset to define the physical memory address. The paging model of memory is shown in Figure 9.9.

The following outlines the steps taken by the MMU to translate a logical address generated by the CPU to a physical address:

1. Extract the page number p and use it as an index into the page table.
2. Extract the corresponding frame number f from the page table.
3. Replace the page number p in the logical address with the frame number f.

As the offset d does not change, it is not replaced, and the frame number and offset now comprise the physical address.

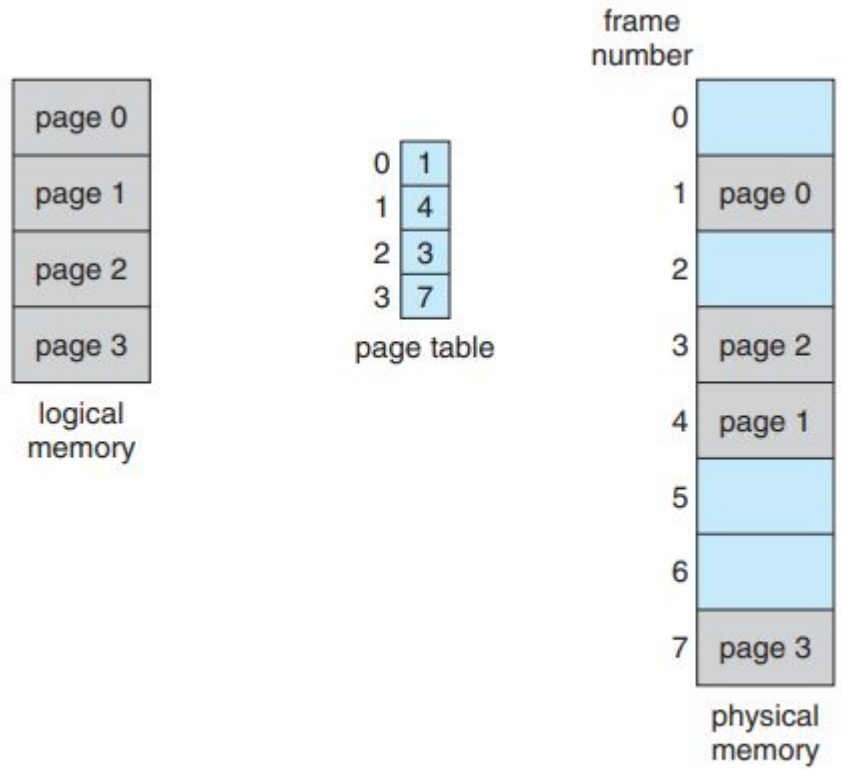


Figure 9.9 Paging model of logical and physical memory.

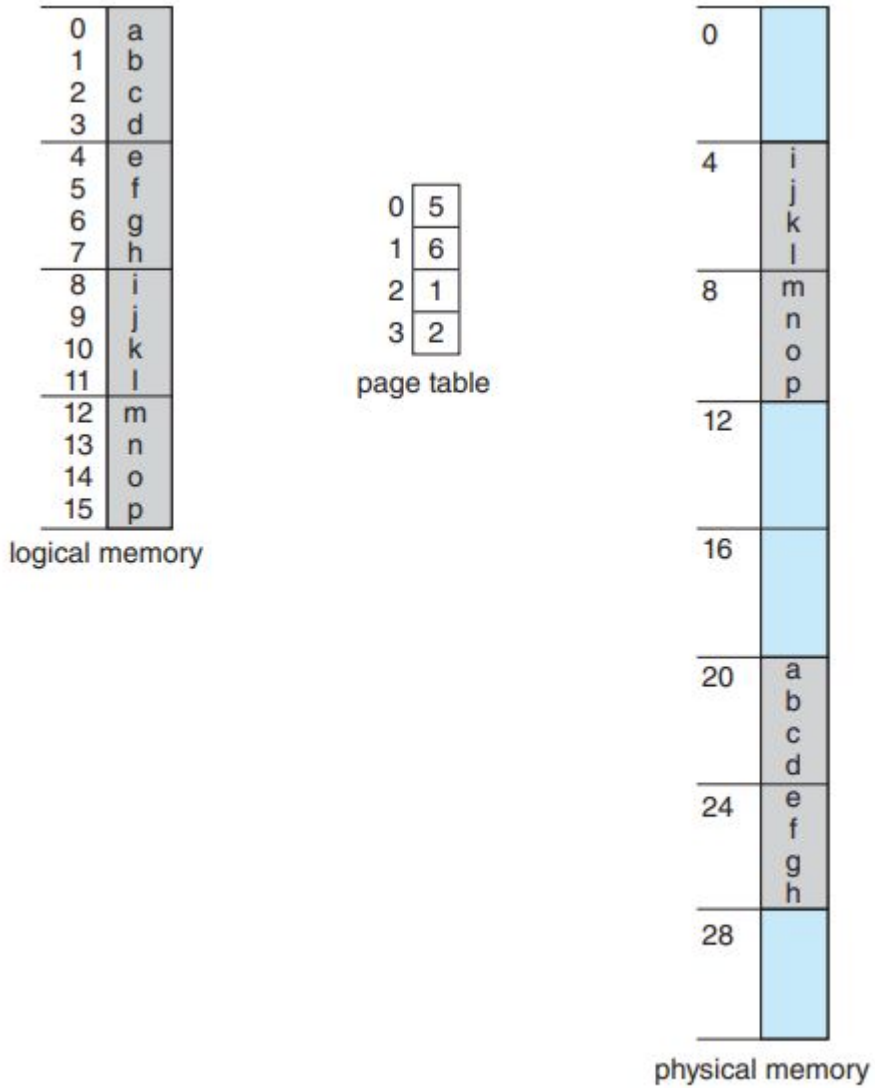


Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.

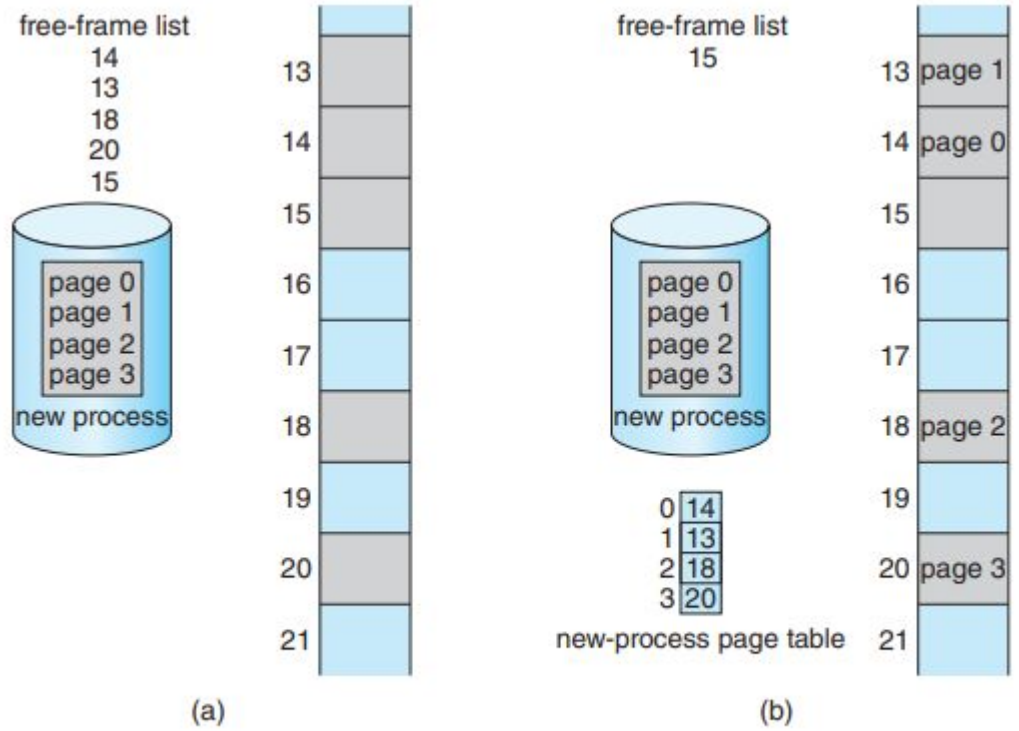


Figure 9.11 Free frames (a) before allocation and (b) after allocation.

HARDWARE IMPLEMENTATION OF PAGE TABLE

The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated high-speed hardware registers, which makes the page-address translation very efficient. However, this approach increases context-switch time, as each one of these registers must be exchanged during a context switch.

The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary CPUs, however, support much larger page tables (for example, 2^{20} entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a **page-table base register** (**PTBR**) points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

HARDWARE IMPLEMENTATION OF PAGE TABLE

- Translation Look-aside Buffer (TLB). The TLB is associative, high-speed memory.
- Each entry in the TLB consists of two parts: a key (or tag) and a value.
- When the associative memory is presented with an item, the item is compared with all keys simultaneously.
- If the item is found, the corresponding value field is returned. The search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty.
- To be able to execute the search within a pipeline step, however, the TLB must be kept small. It is typically between 32 and 1,024 entries in size.

- The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB.
- If the page number is found, its frame number is immediately available and is used to access memory.
- As just mentioned, these steps are executed as part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging.
- If the page number is not in the TLB (known as a TLB miss), address translation proceeds following the steps, where a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory.
- In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.
- If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) through round-robin to random.

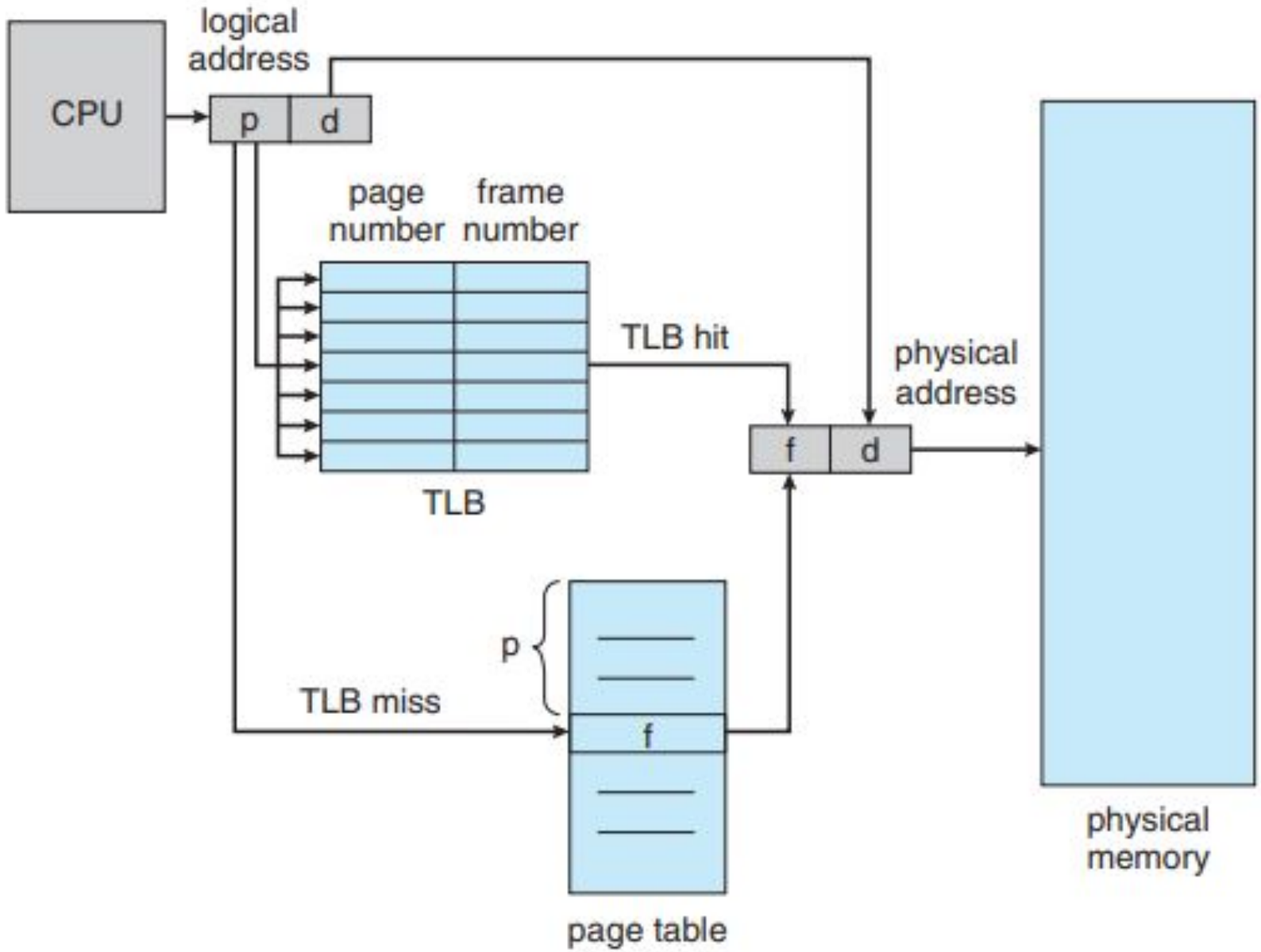
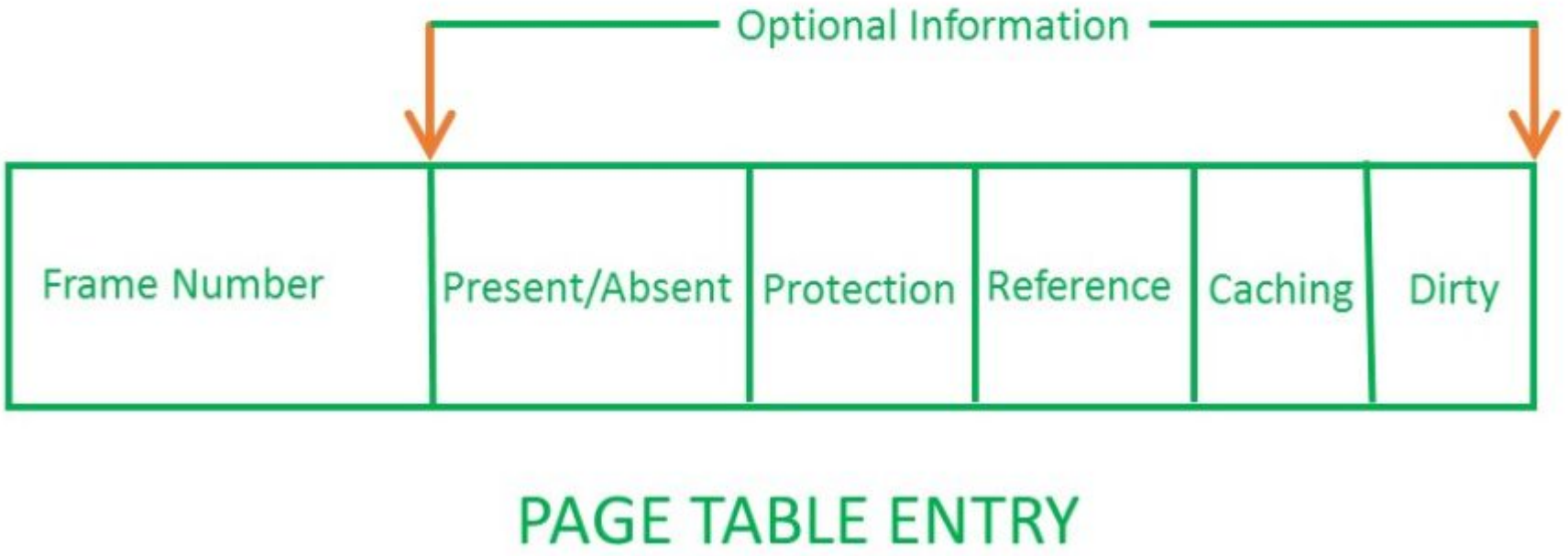


Figure 9.12 Paging hardware with TLB.

PAGE TABLE ENTRIES

Page table has page table entries where each page table entry stores a frame number and optional status (like protection) bits. Many of status bits used in the virtual memory system. The most **important** thing in PTE is **frame Number**.

Page table entry has the following information –



1. **Frame Number** – It gives the frame number in which the current page you are looking for is present. The number of bits required depends on the number of frames.Frame bit is also known as address translation bit.

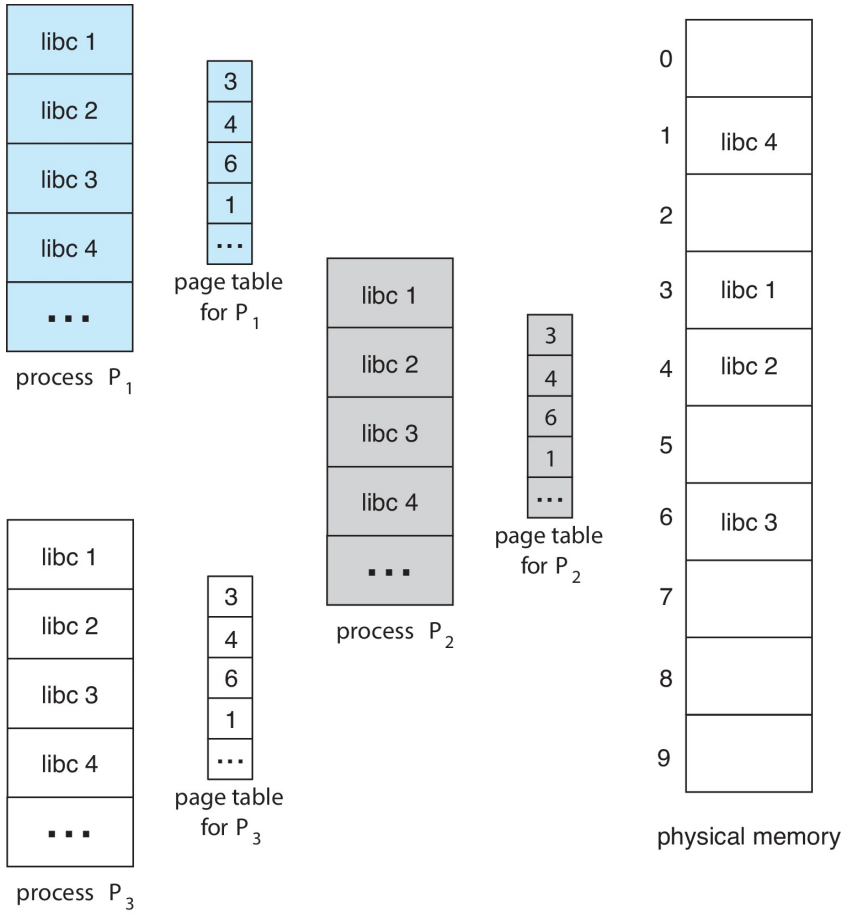
```
Number of bits for frame = Size of physical memory/frame size
```

2. **Present/Absent bit** – Present or absent bit says whether a particular page you are looking for is present or absent. In case if it is not present, that is called Page Fault. It is set to 0 if the corresponding page is not in memory. Used to control page fault by the operating system to support virtual memory. Sometimes this bit is also known as **valid/invalid** bits.
3. **Protection bit** – Protection bit says that what kind of protection you want on that page. So, these bit for the protection of the page frame (read, write etc).
4. **Referenced bit** – Referenced bit will say whether this page has been referred in the last clock cycle or not. It is set to 1 by hardware when the page is accessed.
5. **Caching enabled/disabled** – Some times we need the fresh data. So whenever freshness is required, we don't want to go for caching or many levels of the memory.The information present in the closest level to the CPU and the information present in the closest level to the user might be different. So we want the information to be consistent, which means whatever information user has given, CPU should be able to see it as first as possible. That is the reason we want to disable caching. So, this bit **enables or disable** caching of the page.

Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example

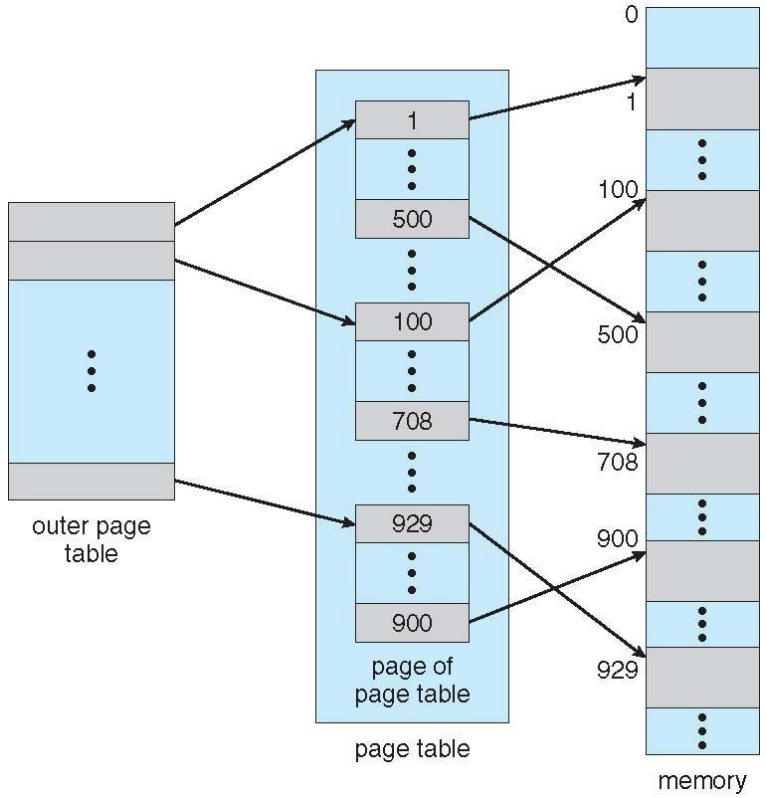


Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries ($2^{32} / 2^{12}$)
  - If each entry is 4 bytes □ each process 4 MB of physical address space for the page table alone
    - Don't want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller units
    - Hierarchical Paging
    - Hashed Page Tables
    - Inverted Page Tables

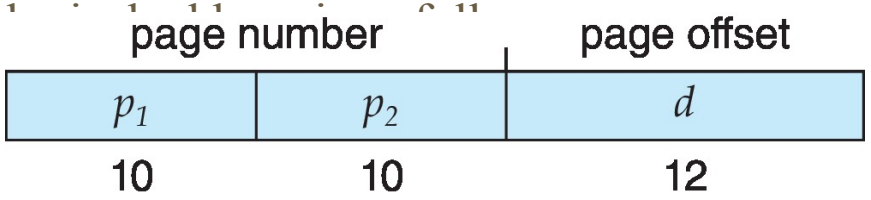
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table



Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a

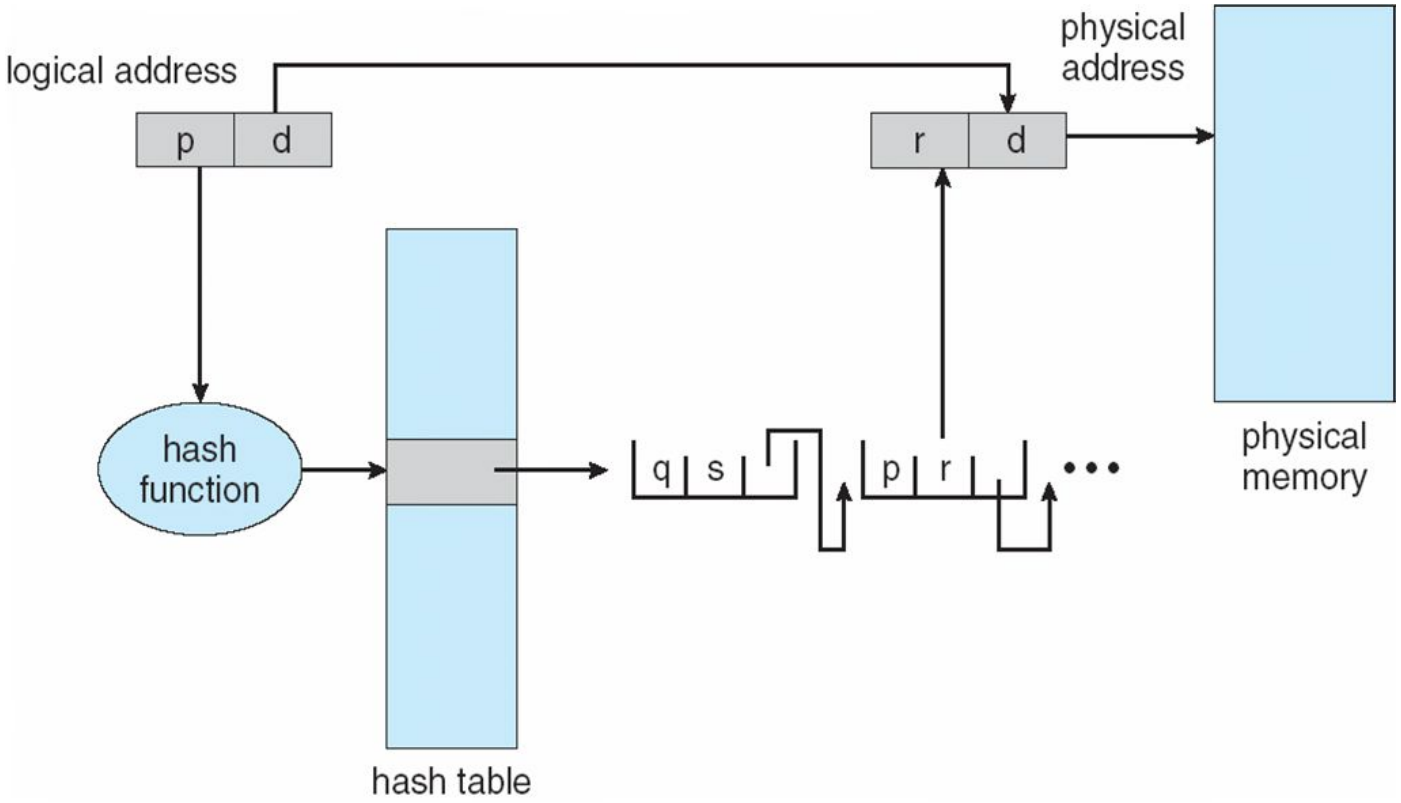


- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

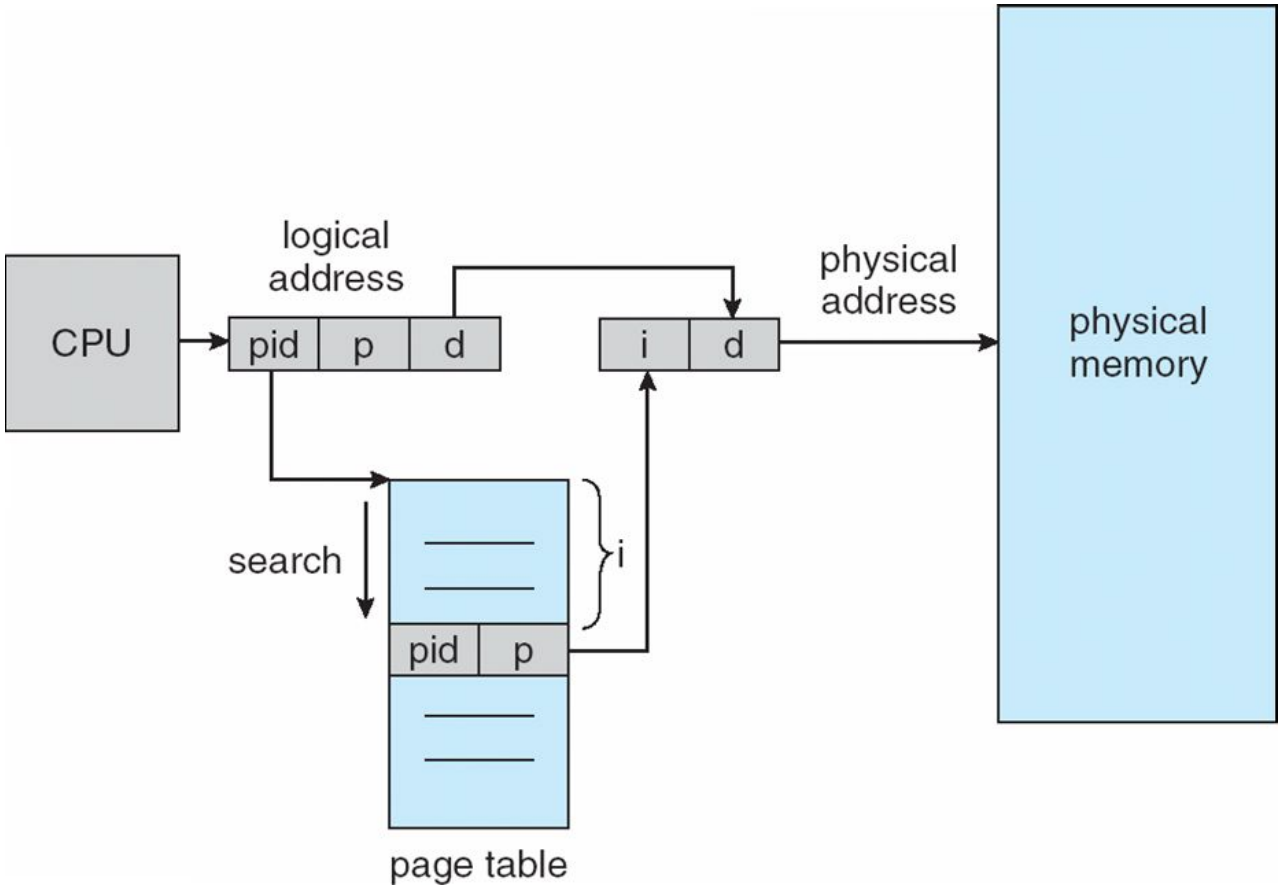
Hashed Page Table



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture

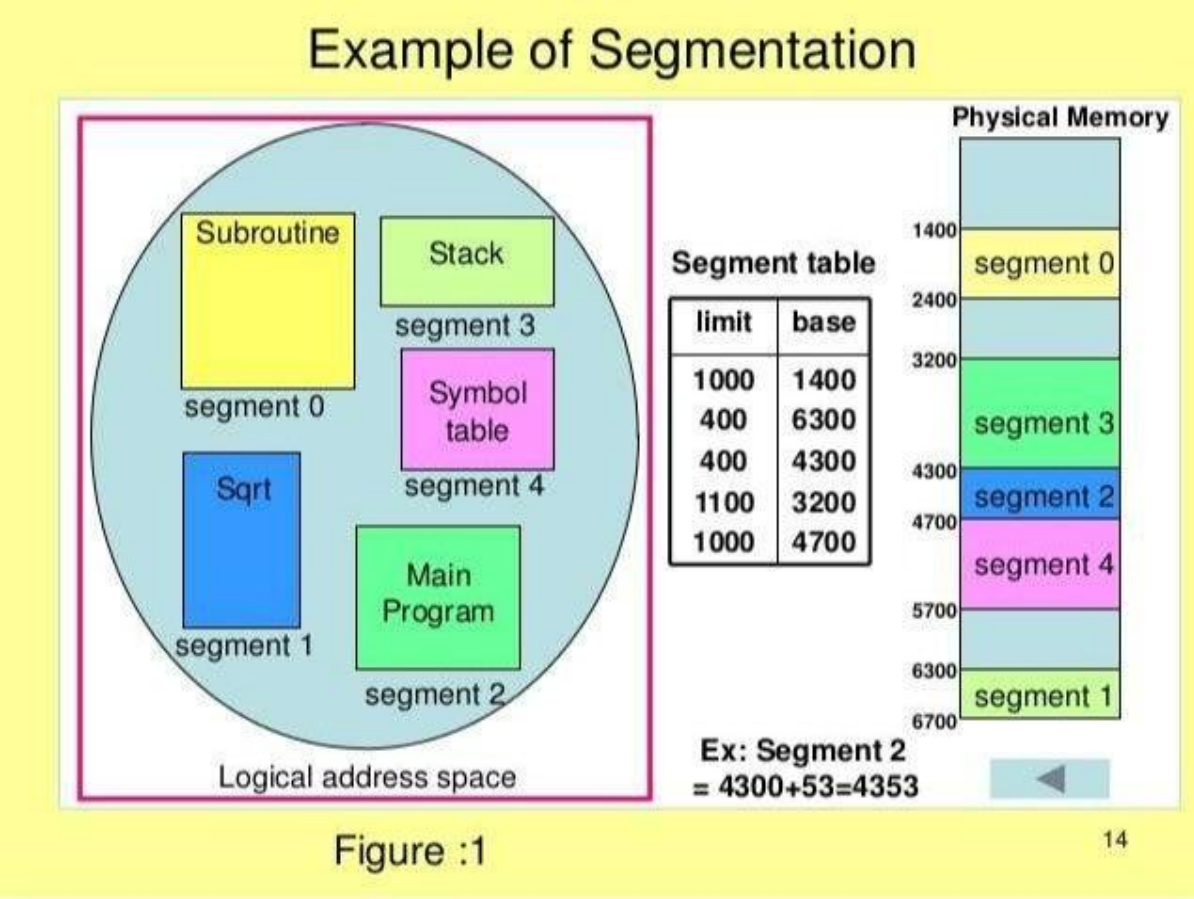


SEGMENTATION

- Segmentation is another non contiguous memory allocation technique like paging.
- Unlike Paging, in segmentation, the process are not divided into fixed size pages.
- Instead, the processes are divided into several modules called segments which improves the visualization for the users.
- So here, both secondary memory & main memory are divided into partitions of unequal sizes.
- A logical address space is a collection of segments.
- Each Segment has a name & a length.
- The addresses specify both the segment name & the offset within the segment.
- The user therefore specifies each address by two quantities: **A segment name** & **an offset.**
- For simplicity of implementation, segments are numbered & are referred to by a segment number, rather than by a name.

SEGMENTATION

- We must define an implementation to map two dimensional user defines addresses into one dimensional physical address.
- This mapping is done by using a segment table.
- Each entry in the segment table has a **segment base** & a **segment limit.**
- The segment base contains the starting point physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.



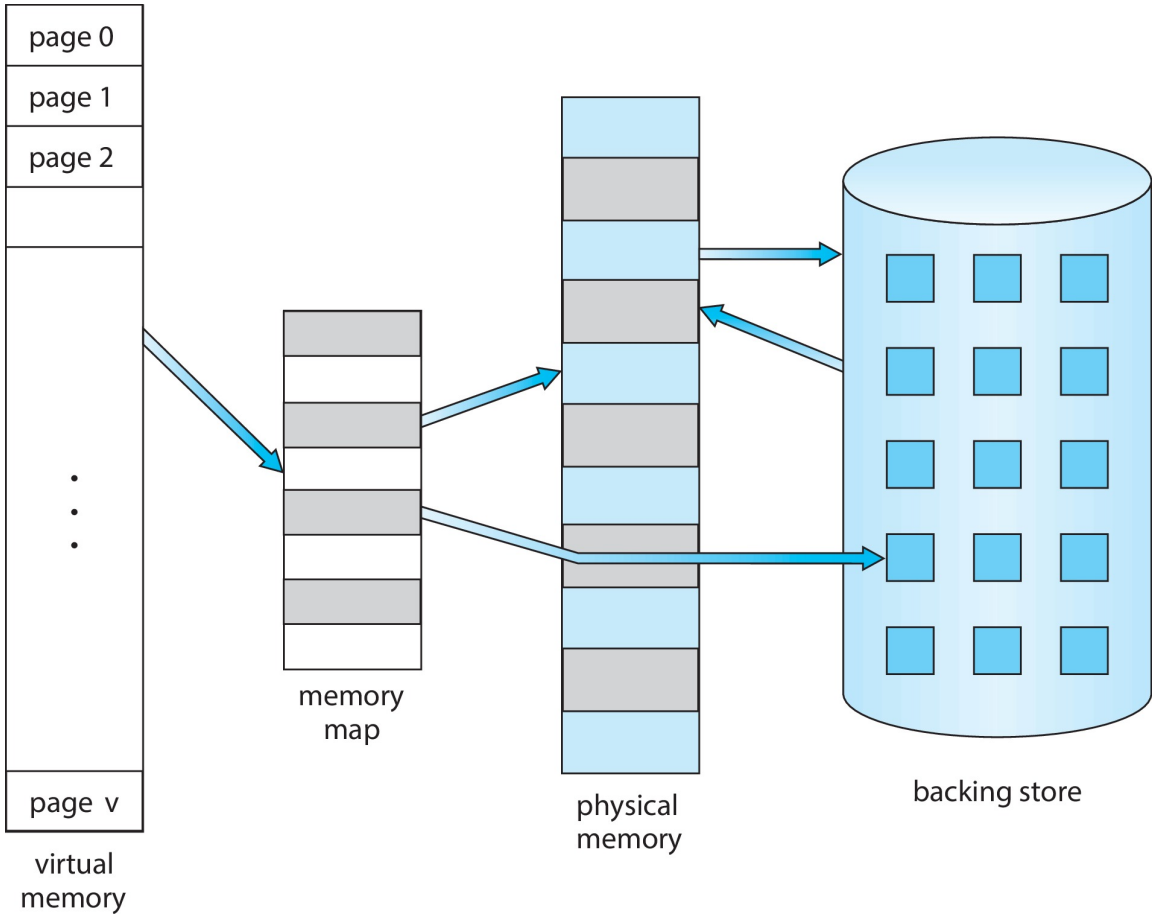
BACKGROUND OF VIRTUAL MEMORY

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

VIRTUAL MEMORY

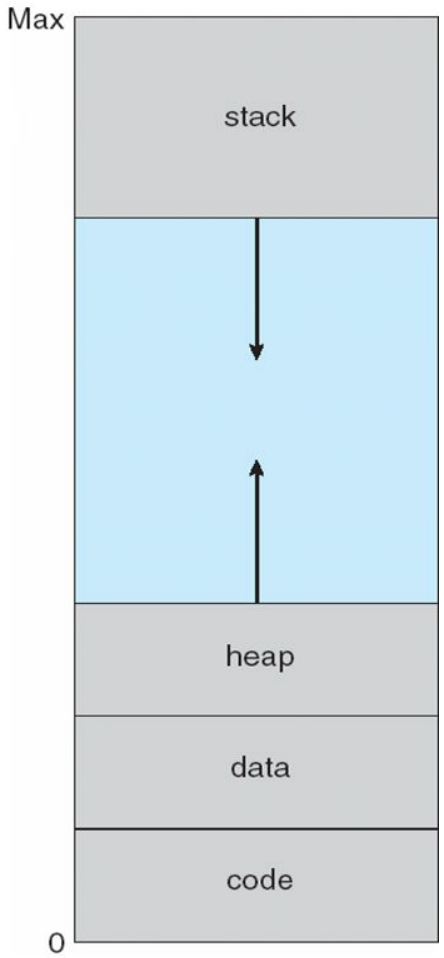
- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

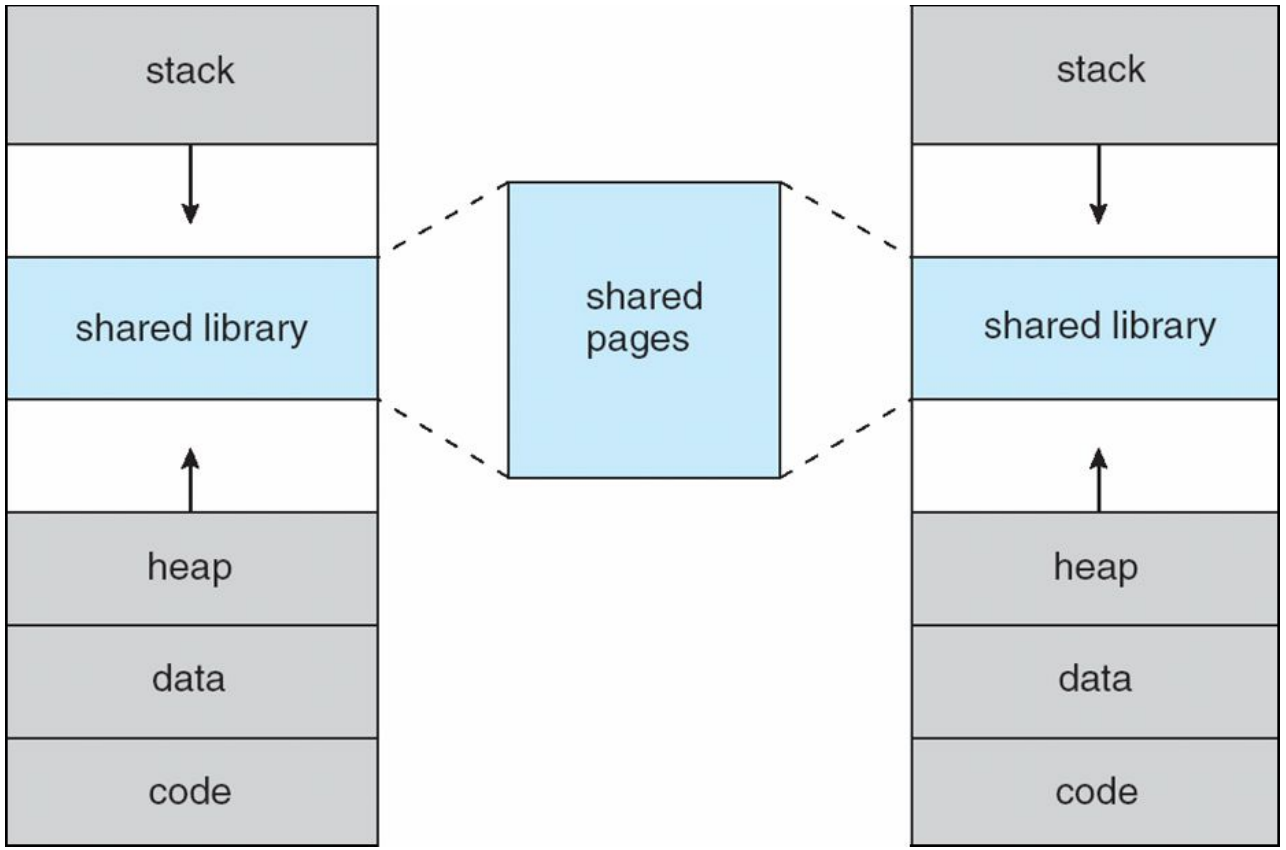


Virtual-address Space

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



Shared Library Using Virtual Memory



DEMAND PAGING

- Consider how an executable program might be loaded from secondary storage into memory.
- One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially need the entire program in memory.
- Suppose a program starts with a list of available options from which the user is to select.
- Loading the entire program into memory results in loading the executable code for all options, regardless of whether or not an option is ultimately selected by the user.
- An alternative strategy is to load pages only as they are needed. This technique is known as demand paging and is commonly used in virtual memory systems.
- With demand-paged virtual memory, pages are loaded only when they are demanded during program execution.
- Pages that are never accessed are thus never loaded into physical memory.
- A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually an HDD).
- Demand paging explains one of the primary benefits of virtual memory—by loading only the portions of programs that are needed, memory is used more efficiently

DEMAND PAGING

- While a process is executing, some pages will be in memory, and some will be in secondary storage.
- Thus, we need some form of hardware support to distinguish between the two. The valid – invalid bit scheme described in can be used for this purpose
- This time, however, when the bit is set to “valid,” the associated page is both legal and in memory.
- If the bit is set to “invalid,” the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently in secondary storage.
- The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is simply marked invalid.
- 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.

DEMAND PAGING

The procedure for handling this page fault is straightforward:

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 secondary storage operation to read the desired page into the newly allocated frame.
5. When the storage read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
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.

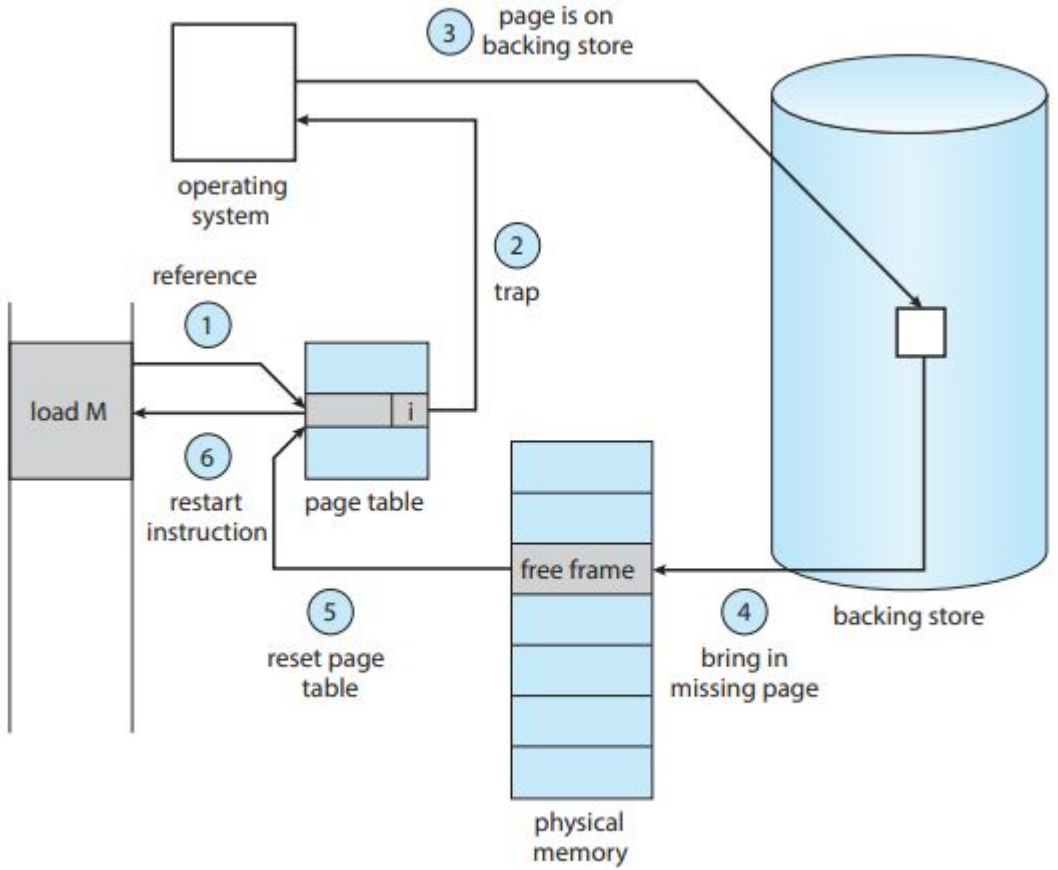
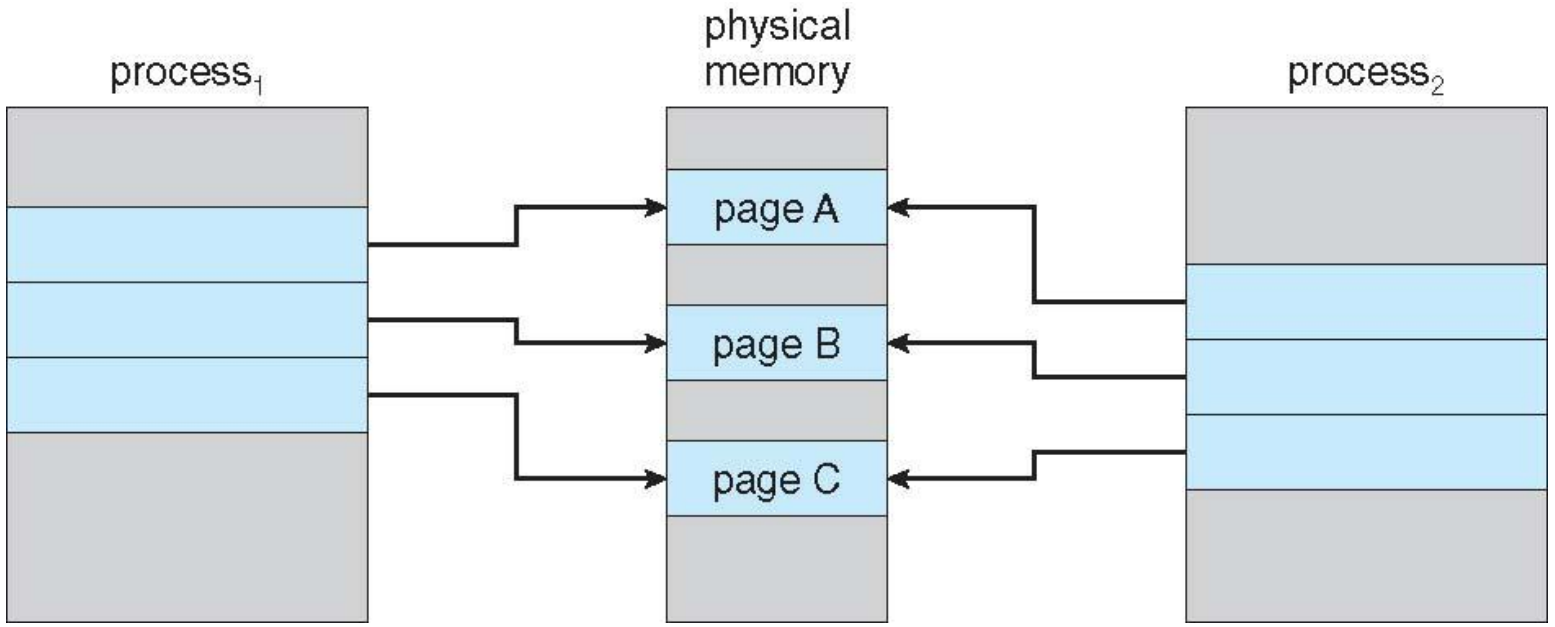


Figure 10.5 Steps in handling a page fault.

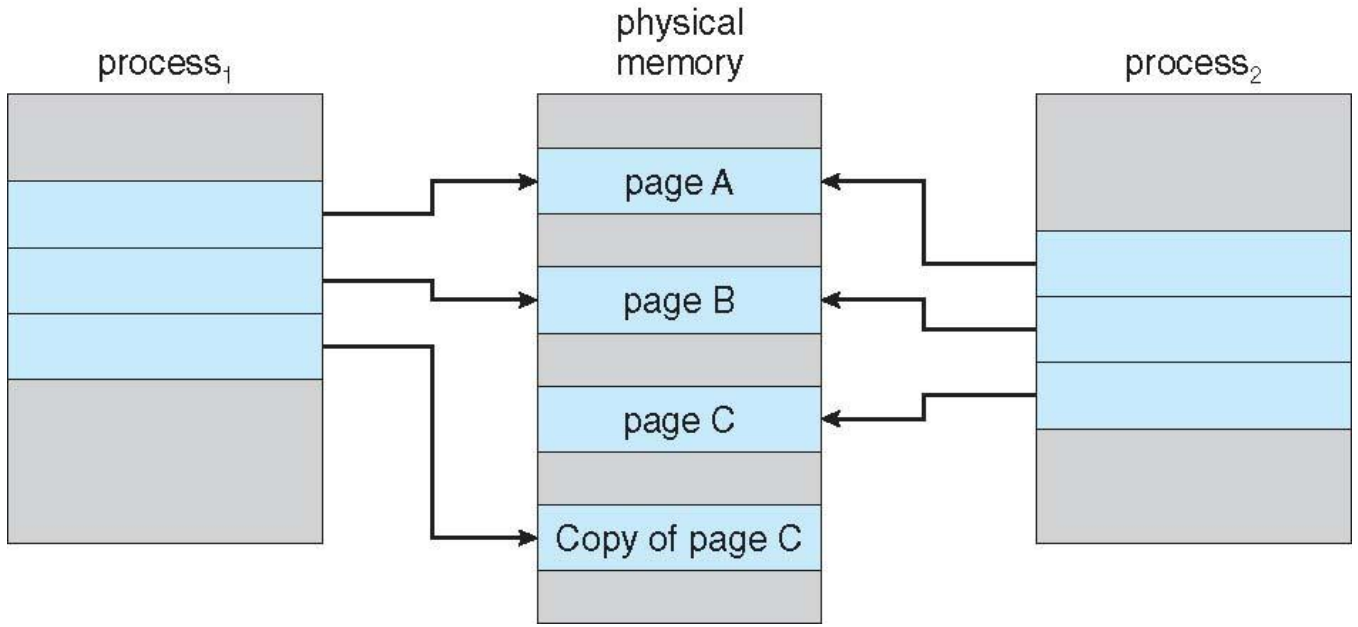
Copy-on-Write

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

Before Process 1 Modifies Page C



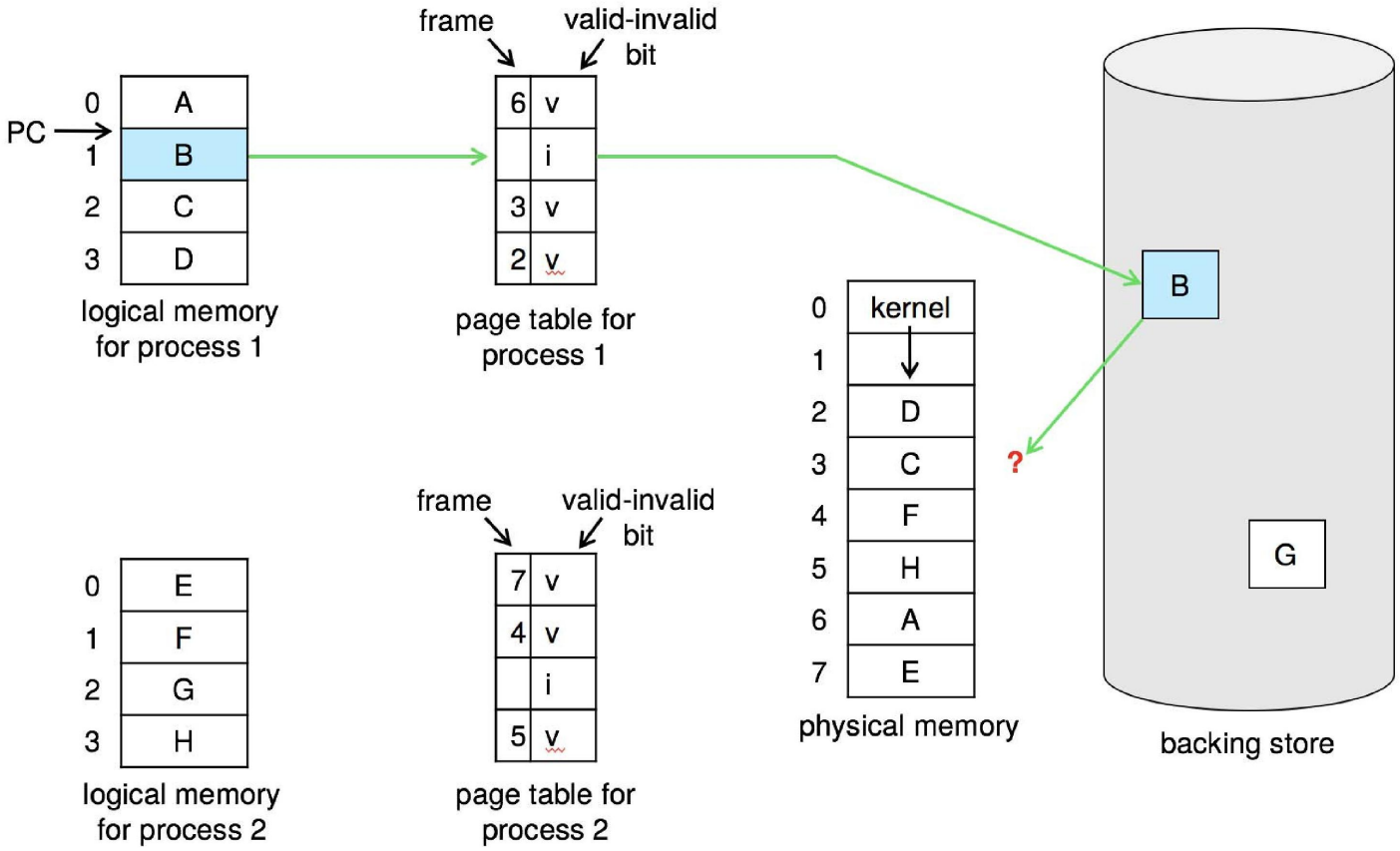
After Process 1 Modifies Page C



Page Replacement

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

Need For Page Replacement

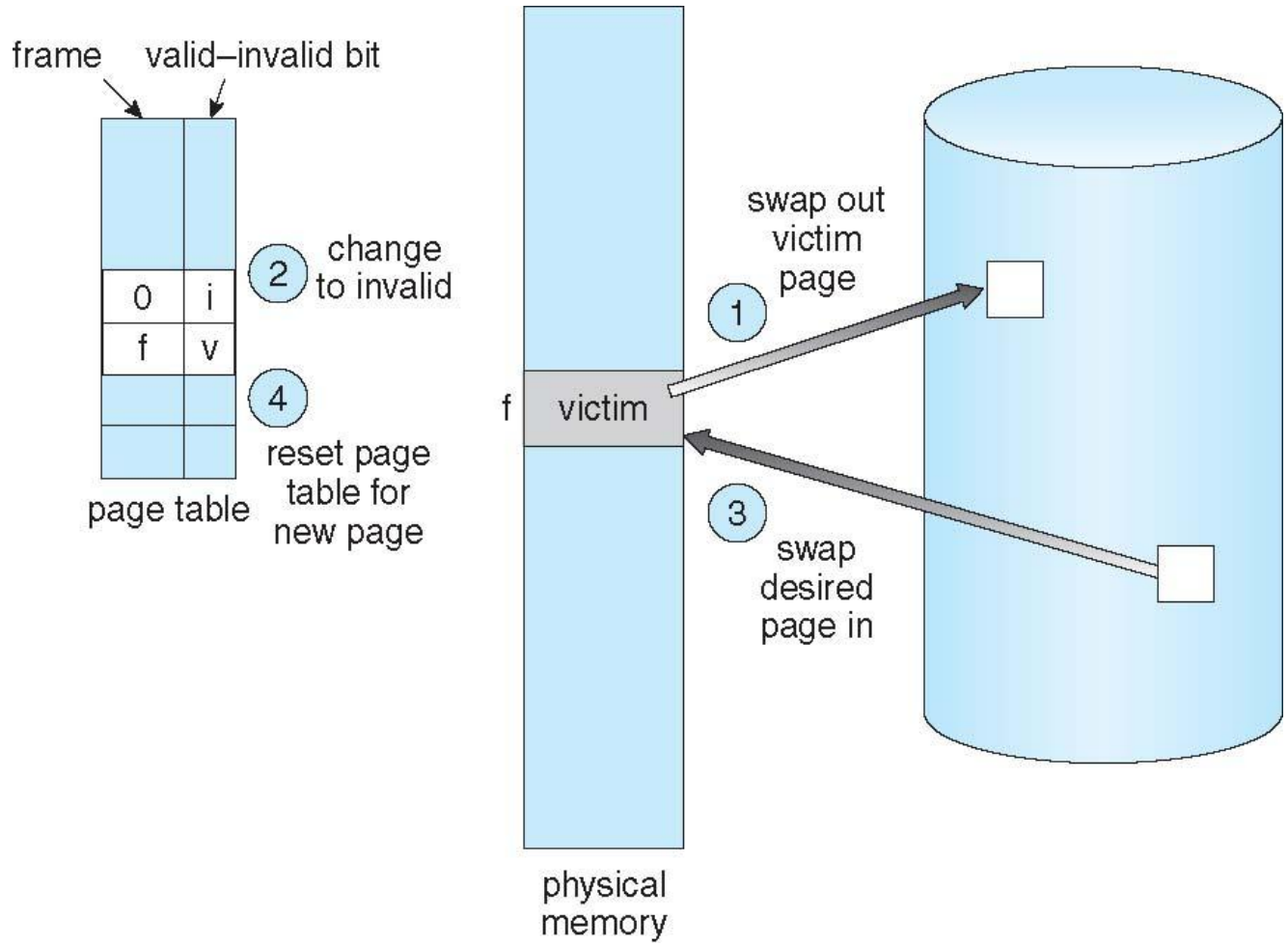


Basic Page Replacement

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

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

Page Replacement

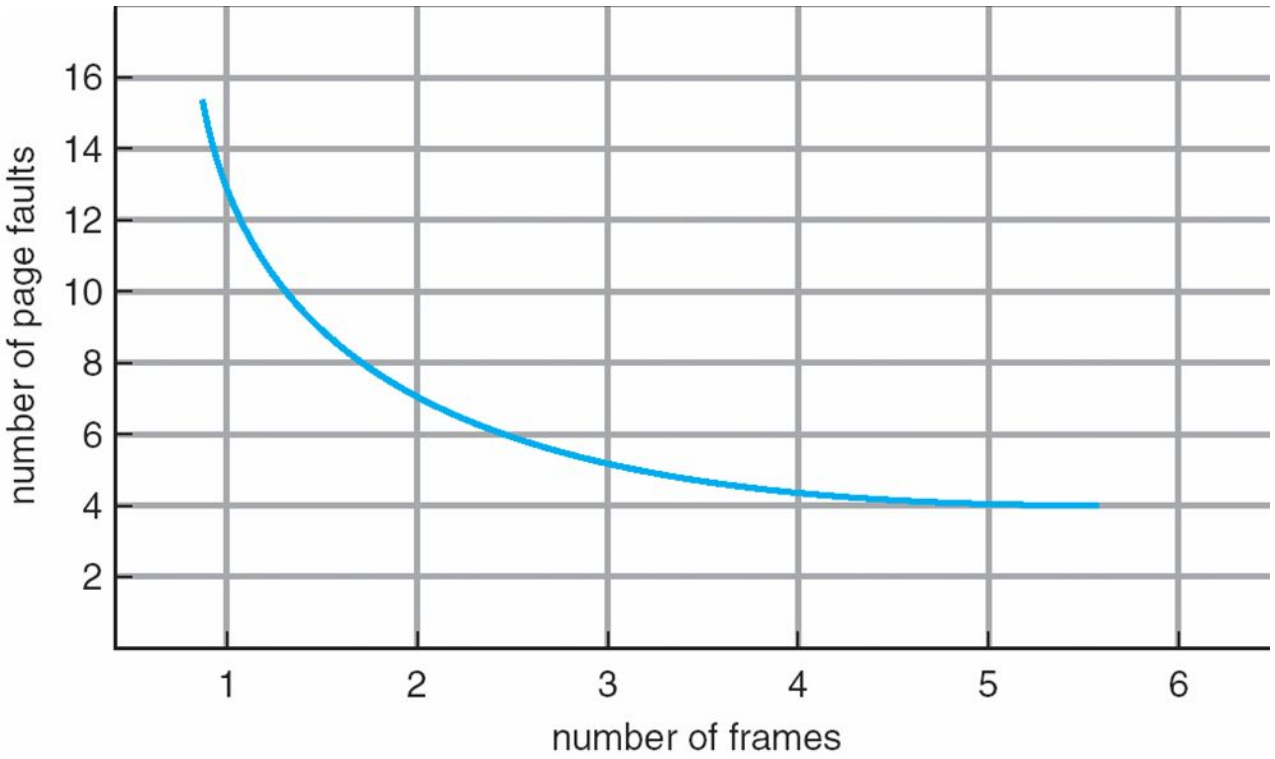


Page and Frame Replacement Algorithms

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

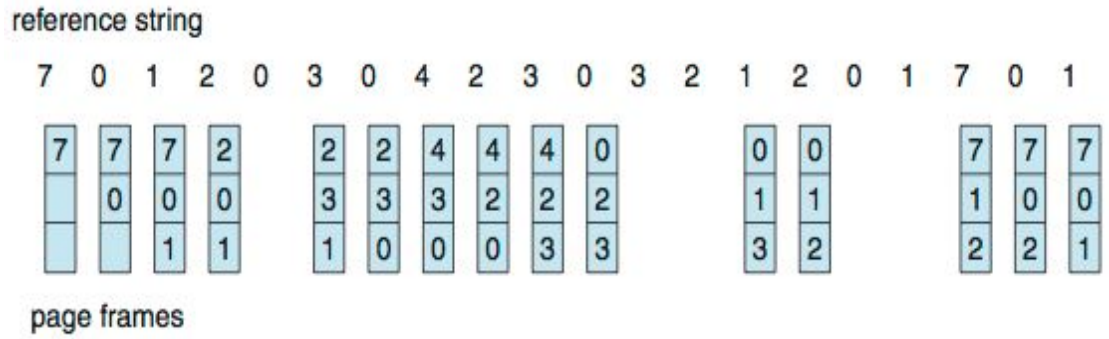
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus the Number of Frames



First-In-First-Out (FIFO) Algorithm

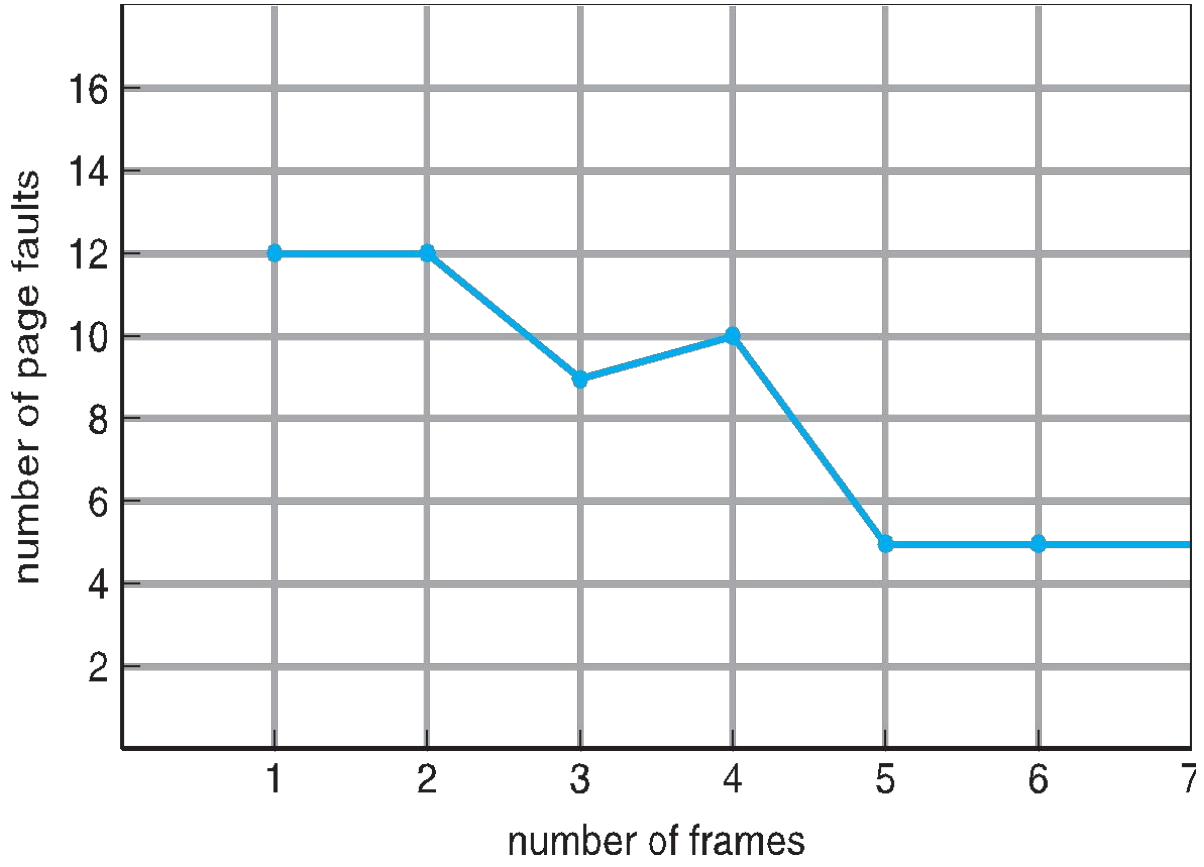
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

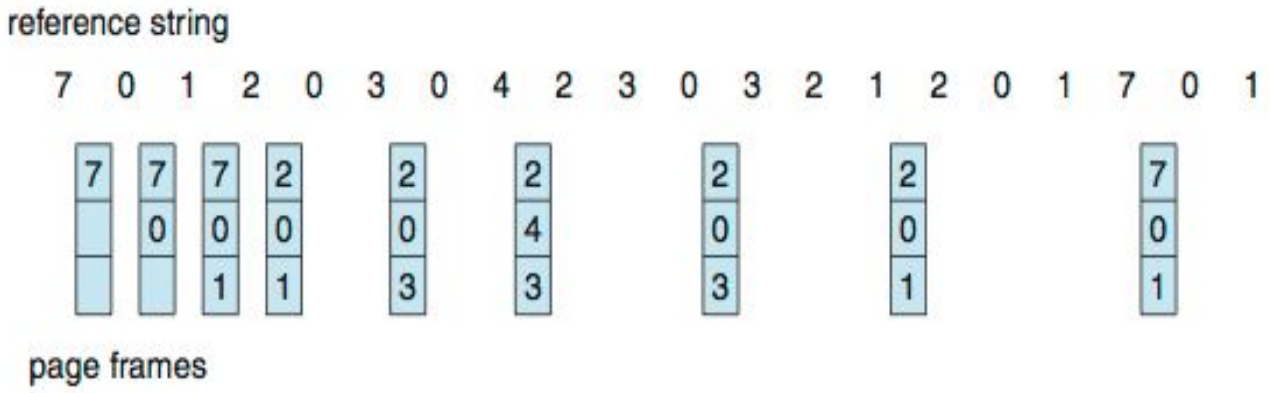
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

FIFO Illustrating Belady's Anomaly



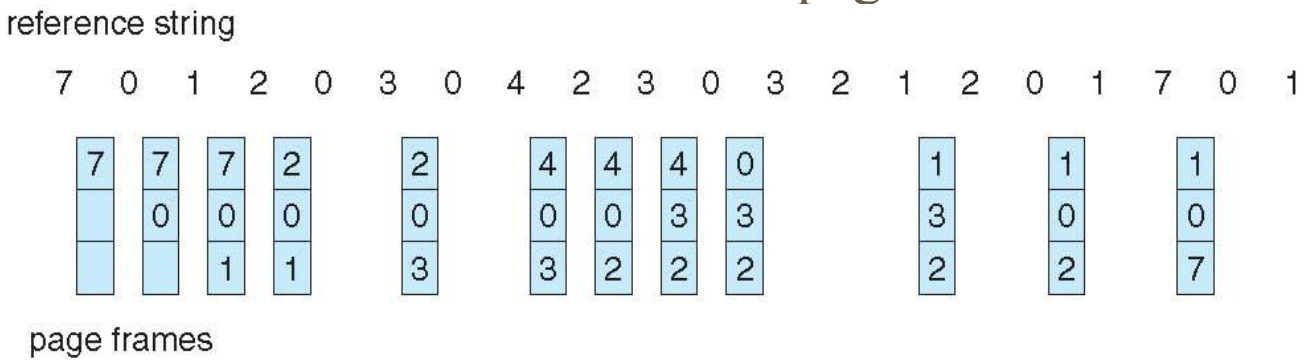
Optimal Algorithm

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



Least Recently Used (LRU) Algorithm

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



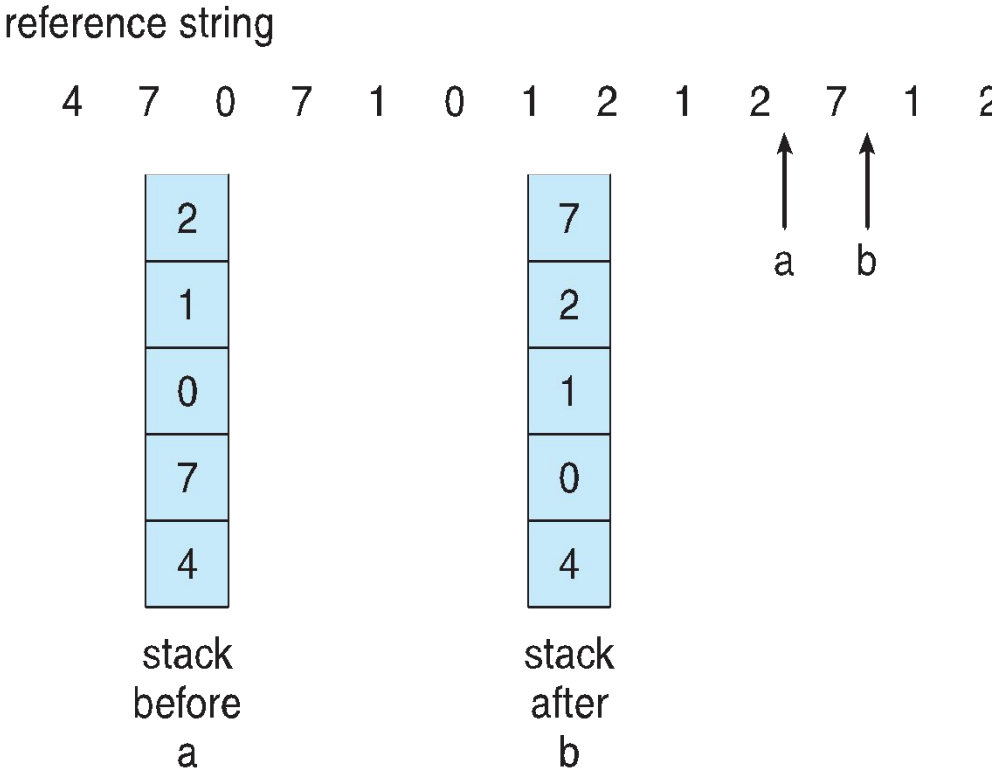
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement

LRU Algorithm (Cont.)

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly
- Use Of A Stack to Record Most Recent Page References



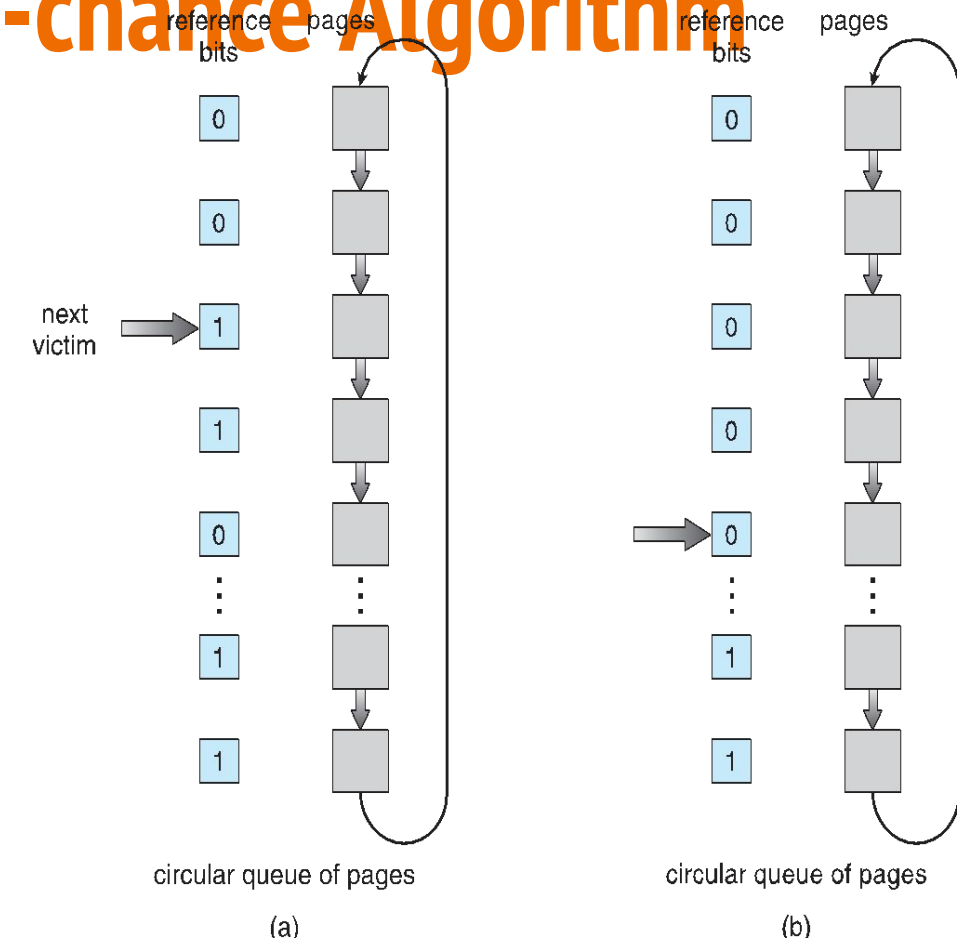
LRU Approximation Algorithms

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

LRU Approximation Algorithms (cont.)

- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

Second-chance Algorithm



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used not modified – best page to replace
  - (0, 1) not recently used but modified – not quite as good, must write out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times

Counting Algorithms

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

Page-Buffering Algorithms

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

Applications and Page Replacement

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

Allocation of Frames

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

Fixed Allocation

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

$m = 64$

$s_1 = 10$

$s_2 = 127$

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

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

Global vs. Local Allocation

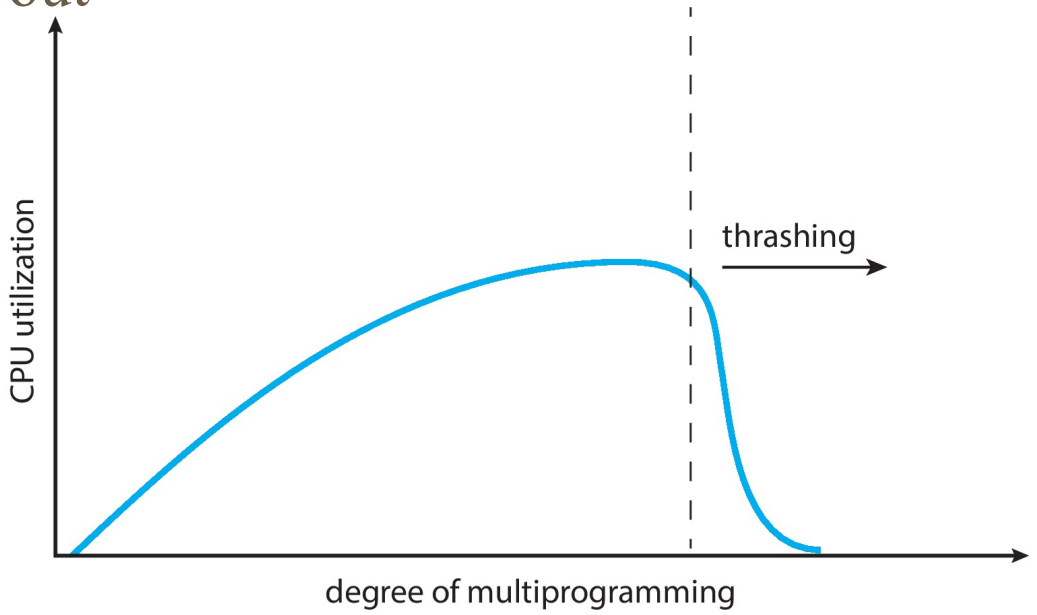
- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

Thrashing (Cont.)

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



Demand Paging and Thrashing

- Why does demand paging work?

  Locality model

  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?

  Σ size of locality > total memory size
- Limit effects by using local or priority page replacement