CSC 314: Memory management

Dr NYAMSI

Memory management: aim

- The main purpose of a computer system is to execute programs.
- These programs, together with the data they access, must be at least partially in main memory during execution.
- To improve both the utilization of the CPU and the speed of its response to users, a general-purpose computer must keep several processes in memory.
- Many memory-management schemes exist, reflecting various approaches, and the effectiveness of each algorithm depends on the situation.
- Selection of a memory-management scheme for a system depends on many factors, especially on the hardware design of the system.

Memory management: outline

- Background
- Some Notions
- Algorithms
  - Swapping
  - Contiguous memory allocation
  - Segmentation
  - Paging (SPW= Student Personal Work)

Memory management: 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.

Memory management: Background

- A typical instruction-execution cycle:
  1. Fetches instruction from memory
  2. Decode instruction
  3. Fetches operand from memory
  4. Execute instruction on the operands
  5. Store results back in memory
- We are interested only in the sequence of memory addresses generated by the running program.

Memory management: Background

- Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly.
- 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.

Memory management: Background

- Registers that are built into the CPU are generally accessible within one **cycle** of the **CPU clock**.
- Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.
- We cannot say the same for main memory, which is accessed via a transaction on the **memory bus**.
- The processor normally needs to **stall**, since it does not have the data required to complete the instruction that it is executing.

Memory management: Background

- This situation is intolerable because of the frequency of memory accesses.
- The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access.
- We call such a memory **cache memory**

Memory management: Background

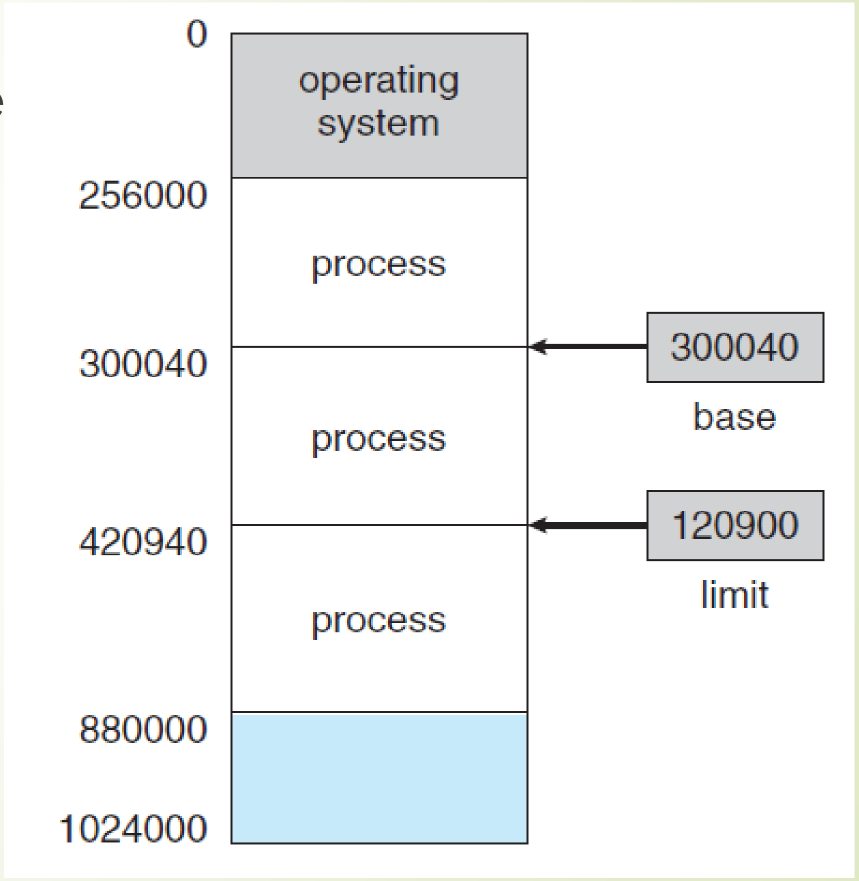
- We are concerned with the relative speed of accessing physical memory,
- We also must ensure correct operation.
- For proper system operation we must protect the operating system from access by user processes.
- On multiuser systems, we must additionally protect user processes from one another.
- This protection must be provided by the hardware because the operating system doesn’t usually intervene between the CPU and its memory accesses (performance penalty).

Memory management: Background

- Hardware implements this production in several different ways. Let's outline one possible implementation.
- We first need to make sure that each process has a separate memory space.
- Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution.
- To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
- We can provide this protection by using two registers, usually a base and a limit.

Memory management: Background

- The **base register** holds the smallest legal physical memory address;
- the **limit register** specifies the size of the range
- Base and limit registers define a logical address space
- the program can legally access all addresses from 300040 through 420939 (inclusive)



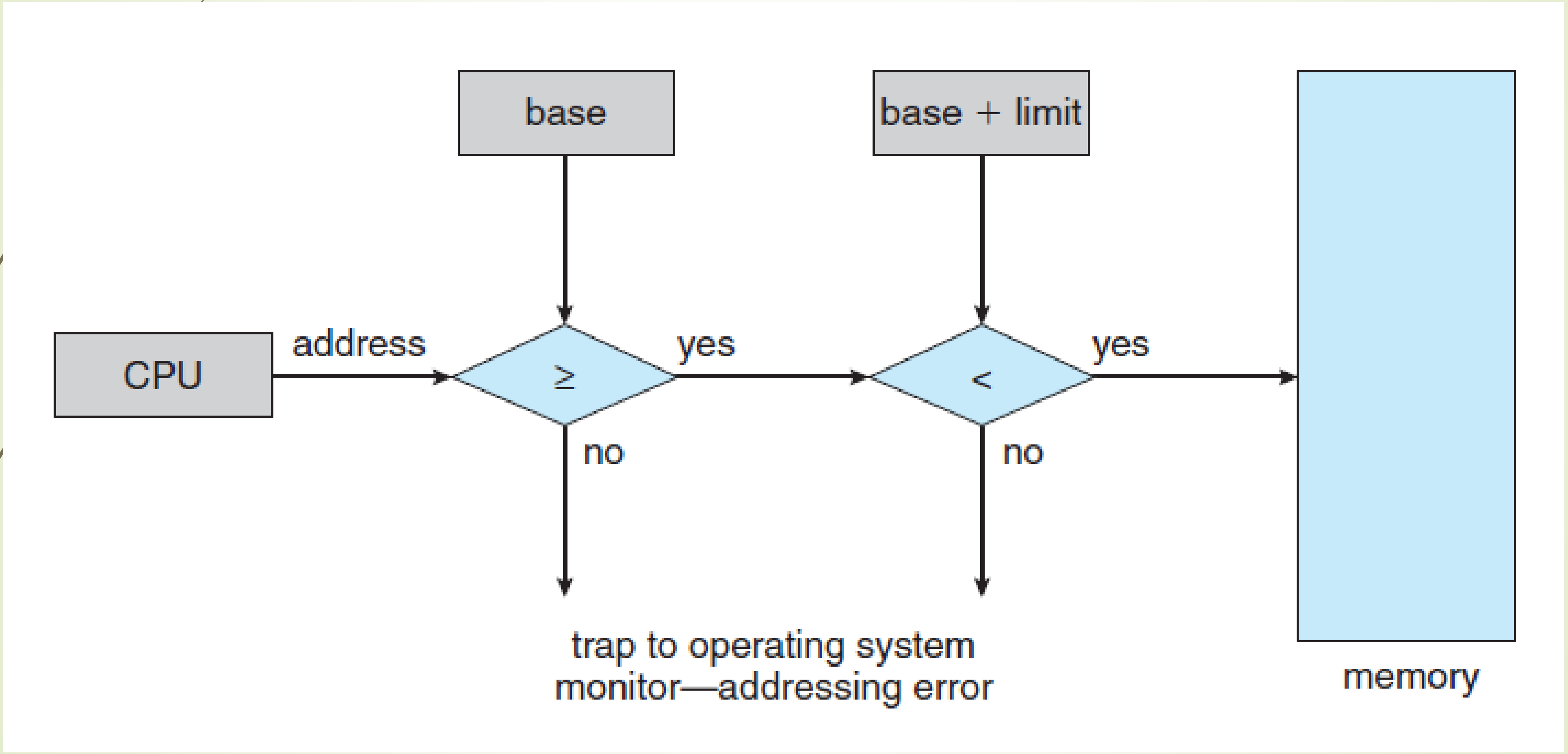
Memory management: Background

- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error.
- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

Memory management: Background

- The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction.
  - Privileged instructions can be executed only in kernel mode,
  - Only the operating system executes in kernel mode,
  - Then, only the operating system can load the base and limit registers.

Memory management: Background



Memory management: Background

- 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.

Memory management: Background

- Usually, a program resides on a disk as a binary executable file.
- The processes on the disk that are waiting to be brought into memory for execution form the input queue.
- The 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.

Memory management: Background

- Most systems allow a user process to reside in any part of the physical memory
- In most cases, a user program goes through several before being executed
- 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.

Memory management: Background

- The binding of instructions and data to memory addresses can be done at any step along the way.
  - **Compile time**: If you know at compile time where the process will reside in memory, then absolute code can be generated.
  - **Load time**: If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code.
  - **Execution time**: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Most general-purpose operating systems use this method.

Memory management: Background

- An address generated by the CPU is commonly referred to as a **logical address**,
- 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**.
- The compile-time and load-time address-binding methods generate identical logical and physical addresses.
- the execution-time address-binding scheme results in differing logical and physical addresses.
  - We usually refer to the logical address as a **virtual address**.

Memory management: Background

- The set of all logical addresses generated by a program is a **logical address space**.
- The set of all physical addresses corresponding to these logical addresses is a **physical address space**.
- In the execution-time address-binding scheme, the logical and physical address spaces differ.
- The **run-time mapping** from virtual to physical addresses is done by a hardware **device** called the memory-management unit (MMU).

Memory management: Background

- The base register is now called a **relocation register.**
- The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory.
- If the base is at 14000, then
  - An attempt by the user to address location 0 is dynamically relocated to location 14000;
  - An access to location 346 is mapped to location 14346.

Memory management: Background

- The user program never sees the real physical addresses.
- The user program deals with logical addresses.
- The memory-mapping hardware converts logical addresses into physical addresses.
- The final location of a referenced memory address is not determined until the reference is made.
- All these is true if the size of a process has thus been limited to the size of physical memory. Which is not reality!!

Memory management: Background

- To obtain better memory-space utilization, we can use **dynamic loading.**
- With dynamic loading, a routine is not loaded until it is called.
- All routines are kept on disk in a relocatable load format.
  - The main program is loaded into memory and is executed.
  - When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
  - If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change.
  - Then control is passed to the newly loaded routine.

Memory management: Background

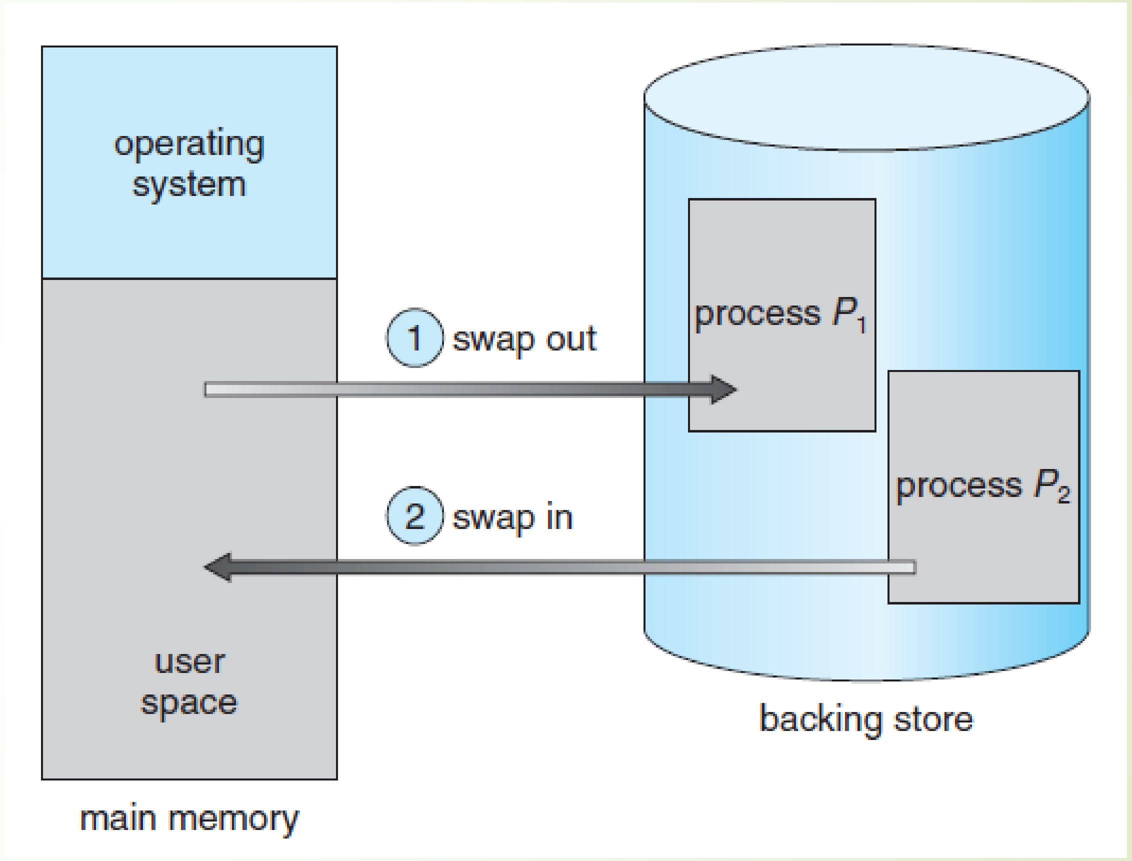
- We will now see how OS and hardware works together to guarantee that a program can run, even if it is 1000 times large than the memory.
- Many methods exists. We focused on
  - Swapping,
  - Contiguous memory allocation (CMA)
  - Segmentation
- We leave paging to you, as personal work

Memory management: Algorithms-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.
- 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.

Memory management: Algorithms-Swapping

- Standard Swapping
- The backing store is commonly a fast disk (Cache)
- 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.



Memory management: Algorithms-Swapping

- 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.
- Notice that the major part of the swap time is transfer time.
  - The total transfer time is directly proportional to the amount of memory swapped.

Memory management: Algorithms-Swapping

- Swapping is constrained by other factors as well.
- If we want to swap a process, we must be sure that it is completely idle.
- Transfers between operating-system buffers and process memory occur only when the process is swapped in.
- This double buffering itself adds overhead.
  - We now need to copy the data again, from kernel memory to user memory, before the user process can access it.
- Standard swapping is not used in modern operating systems

Memory management: Algorithms-CMA

CMA stands 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.
- The major factor affecting this decision is the location of the **interrupt vector**.

Memory management: Algorithms-CMA

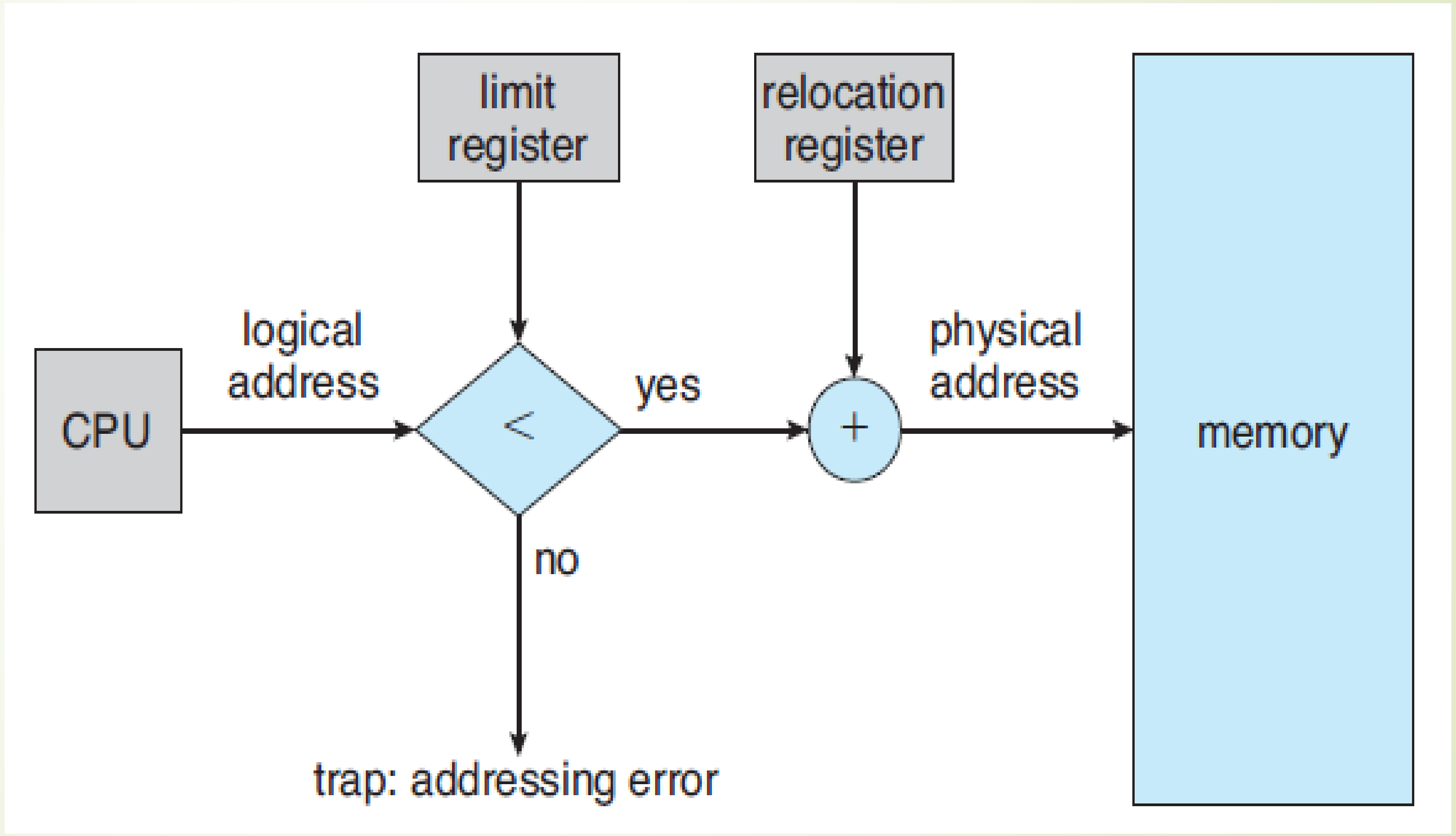
- In CMA, each process is contained in a single section of memory that is contiguous to the section containing the next process.
- We can prevent a process from accessing memory it does not own by combining two ideas: relocation register system and limit register method!

Memory management: Algorithms-CMA

- The relocation register contains the value of the smallest physical address;
- the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600).
- Each logical address must fall within the range specified by the limit register.
- The MMU maps the logical address dynamically by adding the value in the relocation register.
- This mapped address is sent to memory.

Memory management: Algorithms-CMA

- CMA



Memory management: Algorithms-CMA

- One of the simplest methods for allocating memory is to divide memory into several **fixed-sized partitions**.
- Each partition may contain exactly one process.
- Thus, the degree of multiprogramming is bound by the number of partitions.
- In this method, when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.

Memory management: Algorithms-CMA

- In the **variable-partition scheme**, the operating system keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**.
- Eventually, memory contains a set of **holes** of various sizes.

Memory management: Algorithms-CMA

- Allocation method: time and space bound method
  - Processes enter the system, they are put into an input queue.
  - The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated 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 is allocated space, it is loaded into memory, and it can then compete for CPU time.
  - When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

Memory management: Algorithms-CMA

- Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied.
- The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.
- If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.
- The system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

Memory management: Algorithms-CMA

- 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: the first-fit, best-fit, and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes.

Memory management: Algorithms-CMA

- 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.

Memory management: Algorithms-CMA

- 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.

Memory management: Algorithms-CMA

- 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.

Memory management: Algorithms-CMA

- Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation.
- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes.

Memory management: Algorithms-CMA

- Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem.
- Memory fragmentation can be internal as well as external.
- Internal fragmentation is defined as unused memory that is internal to a partition.
- One solution to the problem of external fragmentation is compaction.
- The goal is to shuffle the memory contents so as to place all free memory together in one large block.

Memory management: Algorithms-Segmentation

- As we've already seen, the user's view of memory is not the same as the actual physical memory.
- This is true of the programmer's view of memory.
- What if the hardware could provide a memory mechanism that mapped the programmer's view to the actual physical memory?
- The system would have more freedom to manage memory, while the programmer would have a more natural programming environment.
- **Segmentation** provides such a mechanism.

Memory management: Algorithms-Segmentation

- When writing a program, a programmer thinks of it as a main program with a set of methods, procedures, or functions.
- The programmer talks about "stack," "math library," and "main program" without caring what addresses in memory these elements occupy.
- Segments vary in length, and the length of each is intrinsically defined by its purpose in the program.

Memory management: Algorithms-Segmentation

- Elements within a segment are identified by their offset from the beginning of the segment:
  - the first statement of the program,
  - the seventh stack frame entry in the stack,
  - the fifth instruction of the Sqrt(),
  - … and so on.
- **Segmentation** is a memory-management scheme that supports this programmer view of memory.
- **A logical address space** is a collection of segments.

Memory management: Algorithms-Segmentation

- 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, 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>

Memory management: Algorithms-Segmentation

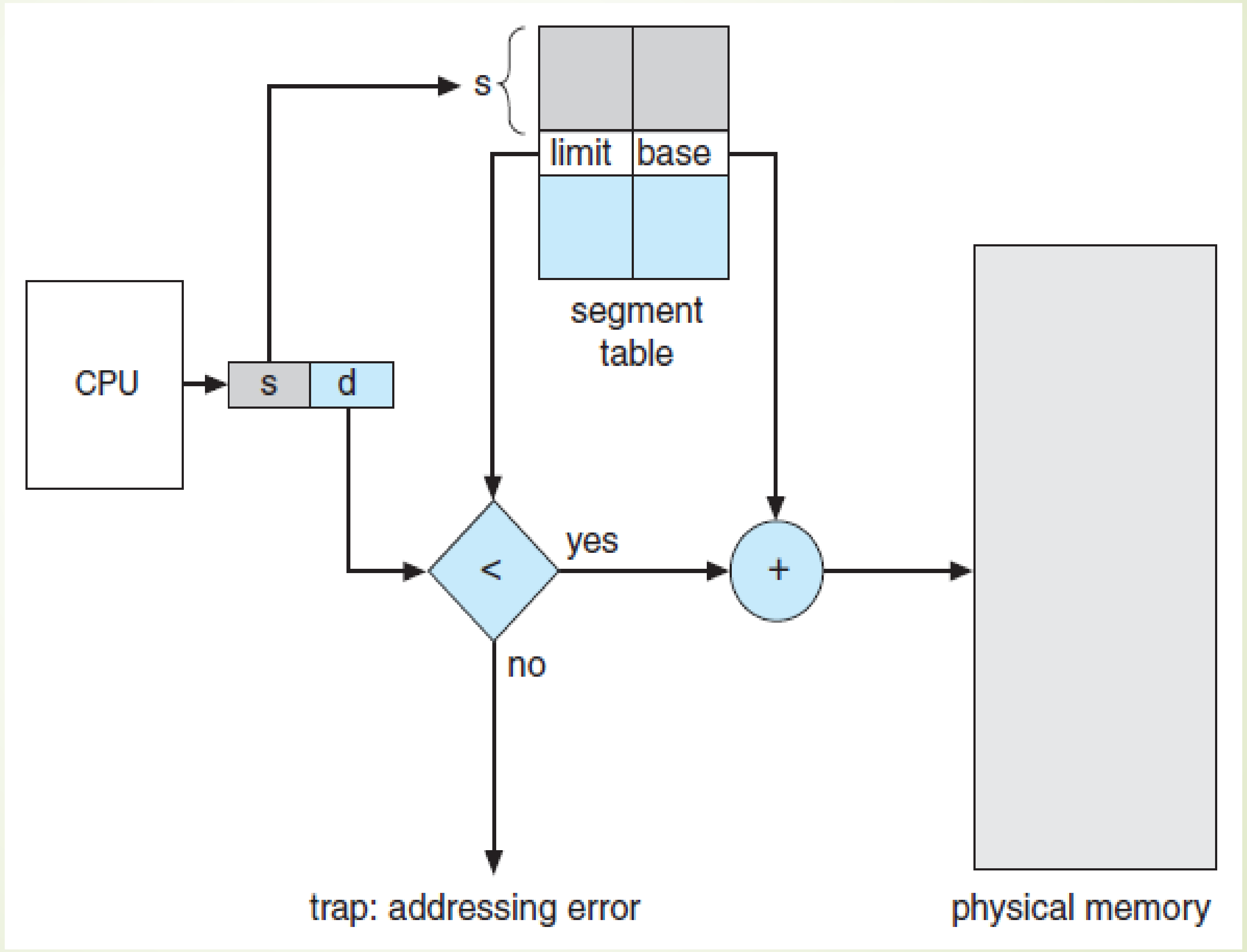
- We must now define an implementation to map two-dimensional user-defined addresses into one-dimensional physical address (hardware address)
- This mapping is effected by a segment table.
- Each entry in the segment table has a segment **base** and a segment **limit**.
  - The segment base contains the starting physical address where the segment resides in memory,
  - the segment limit specifies the length of the segment.

Memory management: Algorithms-Segmentation

- 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.

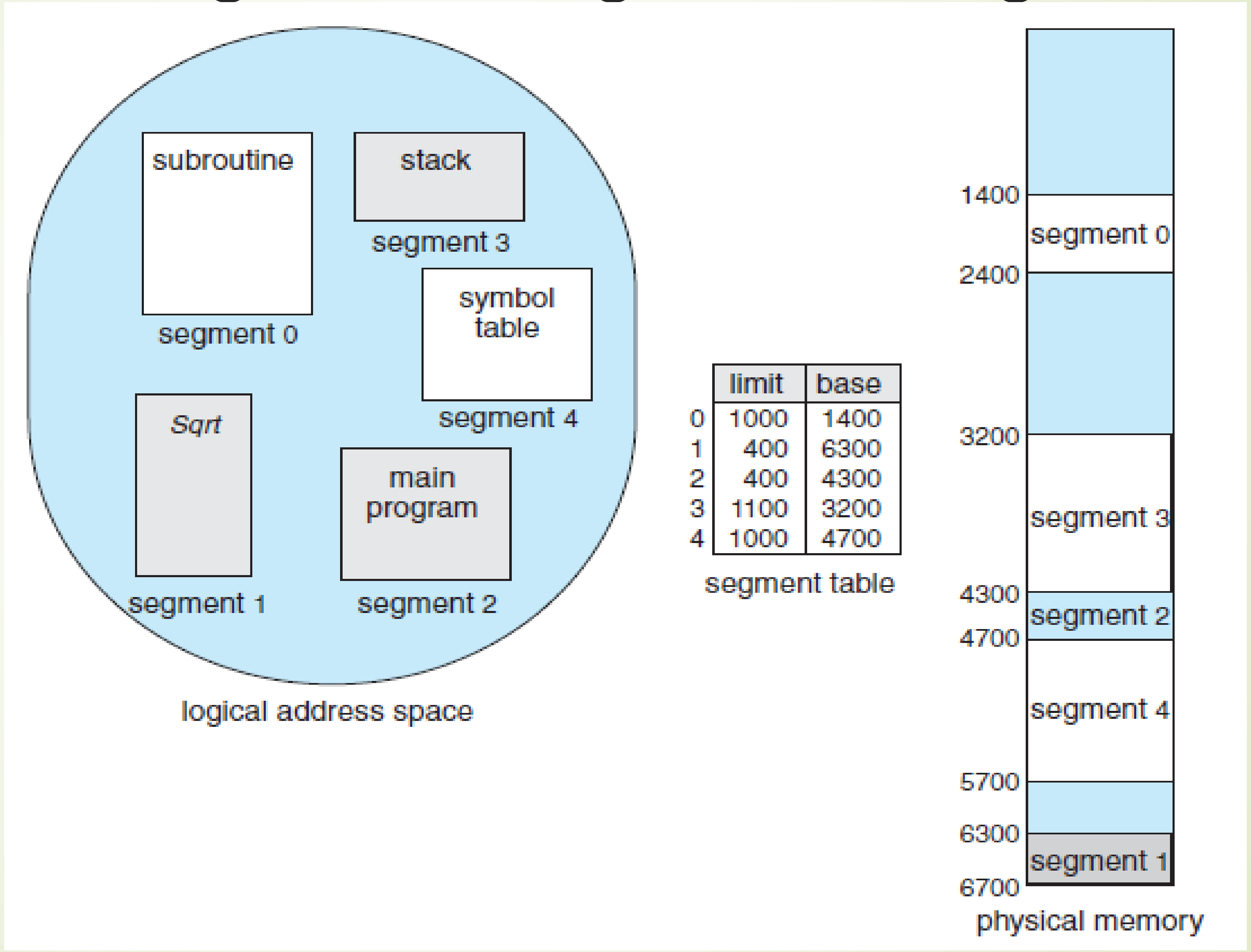
Memory management: Algorithms-Segmentation

- Segmentation hardware.



Memory management: Algorithms-Segmentation

- Example:



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

Memory management: Algorithms-Segmentation

- We have five segments numbered from 0 through 4
- The segments are stored in physical memory as shown
- The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (**base**) and the length of that segment (**limit**).
  - segment 2 is 400 bytes long and begins at location 4300
  - a reference to byte 53 of segment 2 is mapped onto location 4353 (4300 + 53)
  - A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052.
  - A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

Memory management: Algorithms-Paging

- Segmentation permits the physical address space of a process to be non-contiguous.
- Segmentation needs external fragmentation and compaction
- Paging is another memory-management scheme that offers this advantage.
- Paging avoids external fragmentation and compaction.
- It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store.

Memory management: Algorithms-Paging

- Most memory-management schemes used before (that is segmentation, CMA) 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.

Memory management: Algorithms-Paging

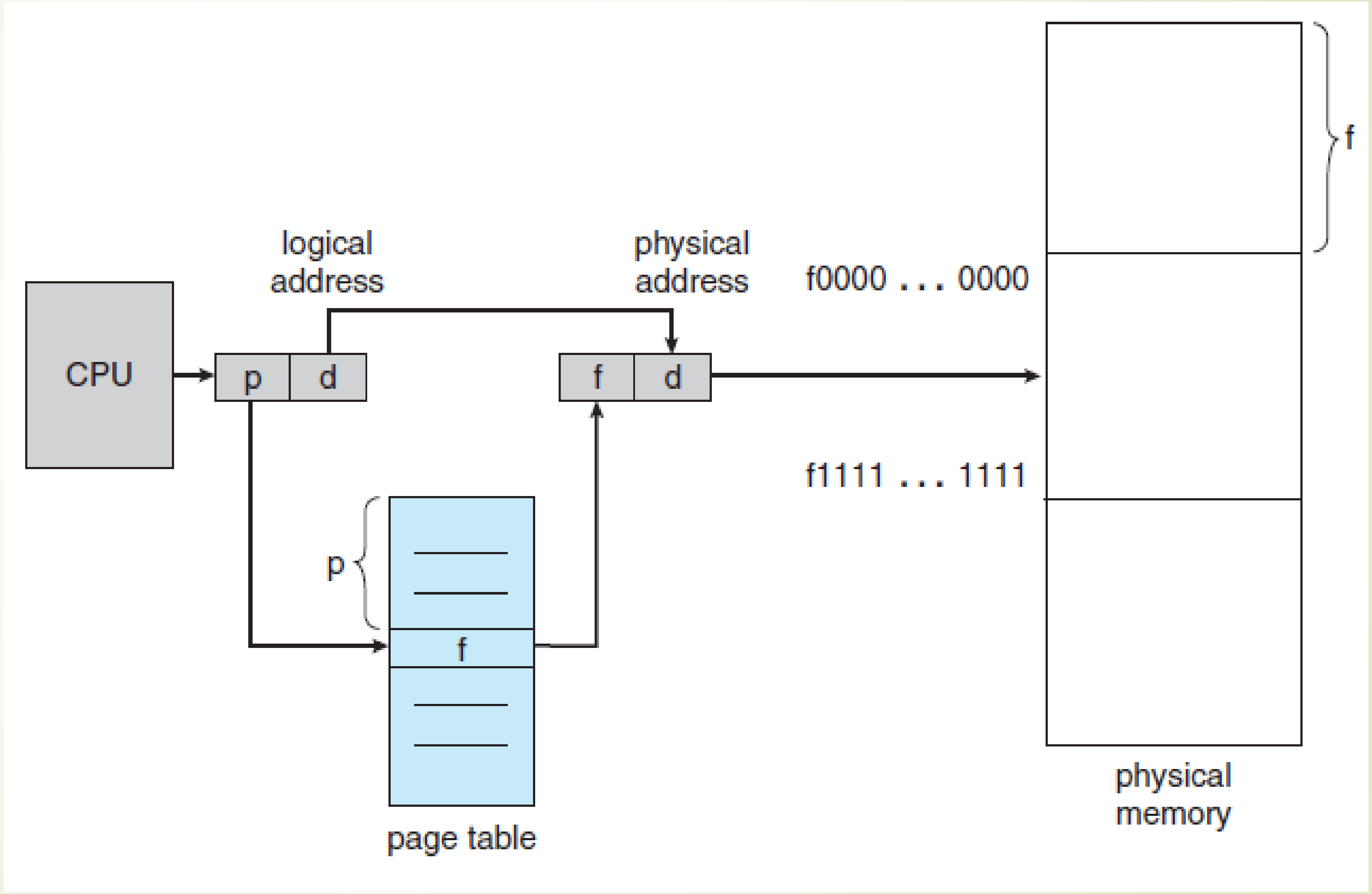
- Paging solves these limitations (or inconveniences)
- Paging in its various forms is used in most operating systems, from those for mainframes through those for smartphones.
- Paging is implemented through **cooperation** between the **operating system** and the **computer hardware**.

Memory management: Algorithms-Paging

- The basic method:
  - breaking physical memory into **fixed-sized** blocks called **frames**
  - 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.

Memory management: Algorithms-Paging

- Representation

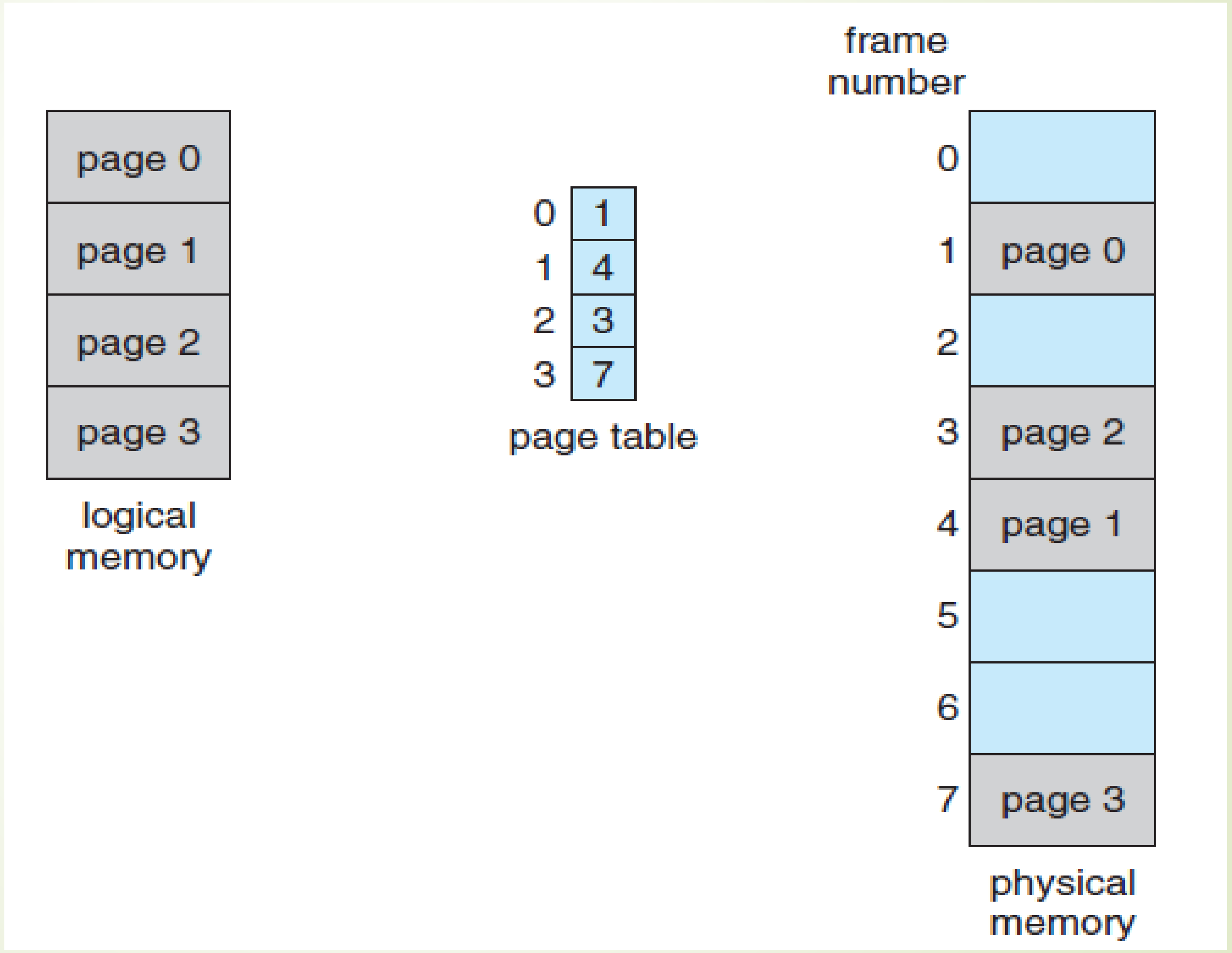


Memory management: Algorithms-Paging

- Every address generated by the CPU is divided into two parts:
  - a page number (**p**)
  - a page offset (**d**).
- The page number is used as an index into a page table.
- The page table contains the base address of each page in physical memory.
- This base address is combined with the page offset to define the **physical memory address** that is sent to the memory unit.

Memory management: Algorithms-Paging

- Example:



Memory management: Algorithms-Paging

- The page size (like the frame size) is defined by the hardware.
- The size of a page is a power of 2 (between 512 bytes and 1 GB per page, depending on the computer architecture).
- 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**=p**) bits of a logical address designate the page number,
  - the (n**=d**) low-order bits designate the page offset

Memory management: Algorithms-Paging

- 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=**p**) bits of a logical address designate the page number,
  - the (n=**d**) low-order bits designate the page offset
- p is an index into the **page table** and d is the **displacement** within the page.

Memory management: Algorithms-Paging

- As assignment
  - Explain the principle of paging
  - Associative paging
  - Use an example to present paging and associative paging

Memory management: summary

- Memory-management algorithms for multi-programmed operating systems range from the simple single-user system approach to segmentation and paging.
- Every memory address generated by the CPU must be checked for legality and possibly mapped to a physical address.
- The checking cannot be efficiently implemented in software.
- The various memory-management algorithms (contiguous allocation, paging, segmentation, and combinations of paging and segmentation) differ in many aspects.

Memory management: summary

Hardware support:

- A simple base register or a base–limit register pair is sufficient for the single- and multiple-partition schemes.
- Paging and segmentation need mapping tables to define the address map.

Memory management: summary

Performance:

- As the memory-management algorithm becomes more complex, the time required to map a logical address to a physical address increases.
- For the simple systems, we need only compare or add to the logical address—operations that are fast.
- Paging and segmentation can be as fast if the mapping table is implemented in fast registers.
- If the table is in memory, however, user memory accesses can be degraded substantially.
- A TLB can reduce the performance degradation to an acceptable level.

Memory management: summary

Fragmentation:

- A multi-programmed system will generally perform more efficiently if it has a higher level of multiprogramming.
- For a given set of processes, we can increase the multiprogramming level only by packing more processes into memory.
- To accomplish this task, we must reduce memory waste, or fragmentation.
- Systems with fixed-sized allocation units, such as the single-partition scheme and paging, suffer from internal fragmentation.
- Systems with variable-sized allocation units, such as the multiple-partition scheme and segmentation, suffer from external fragmentation.

Memory management: summary

Relocation:

- One solution to the external-fragmentation problem is compaction.
- Compaction involves shifting a program in memory in such a way that the program does not notice the change.
- This consideration requires that logical addresses be relocated dynamically, at execution time.
- If addresses are relocated only at load time, we cannot compact storage.

Memory management: summary

Swapping:

- Swapping can be added to any algorithm.
- At intervals determined by the operating system, usually dictated by CPU-scheduling policies, processes are copied from main memory to a backing store and later are copied back to main memory.
- This scheme allows more processes to be run than can be fit into memory at one time.
- In general, PC operating systems support paging, and operating systems for mobile devices do not.

Memory management: summary

Sharing:

- Another means of increasing the multiprogramming level is to share code and data among different processes.
- Sharing generally requires that either paging or segmentation be used to provide small packets of information (pages or segments) that can be shared.
- Sharing is a means of running many processes with a limited amount of memory, but shared programs and data must be designed carefully.

Memory management: summary

Protection:

- If paging or segmentation is provided, different sections of a user program can be declared **execute-only**, **read-only**, or r**ead–write**.
- This restriction is necessary with shared code or data and is generally useful in any case to provide simple run-time checks for common programming errors.

Memory management: Questions

1. Name two differences between logical and physical addresses.
2. Why are page sizes always powers of 2?
3. Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?
4. On a system with paging, a process cannot access memory that it does not own; why? How could the operating system allow access to other memory? Why should it or should it not?

Memory management: Questions

5. Compare paging with segmentation with respect to the amount of memory required by the address translation structures in order to convert virtual addresses to physical addresses.
6. Consider the following segment table below.
   a. Draw a graphic to represent it
   b. What are the physical addresses for the following logical addresses?
      i. 0,430
      ii. 1,10
      iii. 2,500
      iv. 3,400
      v. 4,112

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96