COP 4600 – Summer 2011

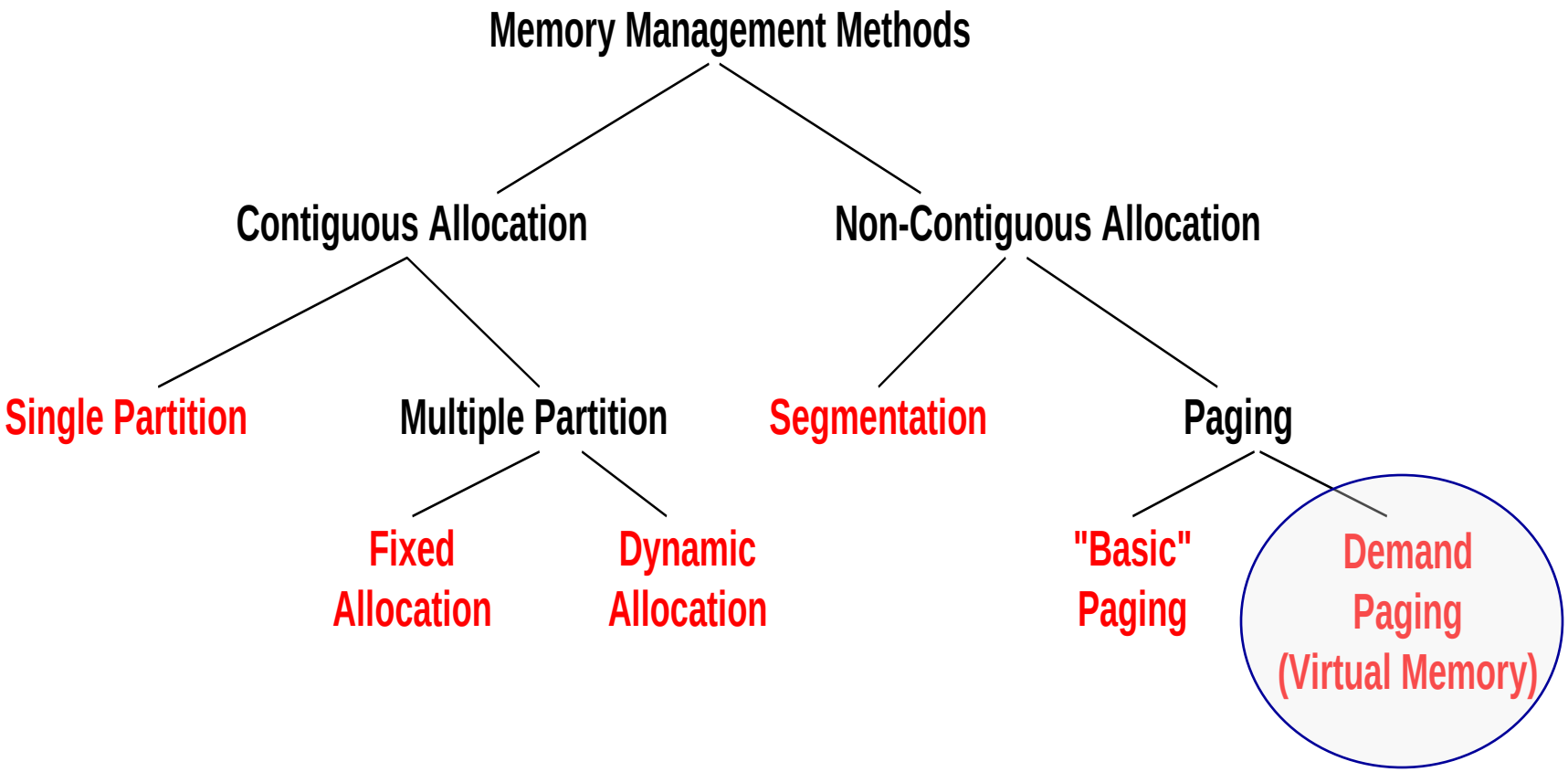
Introduction To Operating Systems

Memory Management – Part 3

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop4600/sum2011

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida

Memory Management



Virtual Memory

- The memory management schemes that we have seen so far are necessary because of one basic requirement: The instructions being executed by the processor must be in the physical memory of the machine.
- The first approach to meeting this requirement is to place the entire logical address space (i.e., the entire process) into the main memory.
- Comparing simple paging and simple segmentation, on one hand, with fixed and dynamic partitioning, on the other, we have the foundations for a more sophisticated memory management system.
- There are two characteristics of paging and segmentation which are key:

1. All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process can be swapped in and out of main memory such that it occupies different regions of main memory at different times during its lifetime.
2. A process may be broken up into a number of chunks (either pages or segments) and these chunks need not be contiguously located in main memory during execution. The combination of dynamic run-time address translation and the use of a page or segment table permits this.

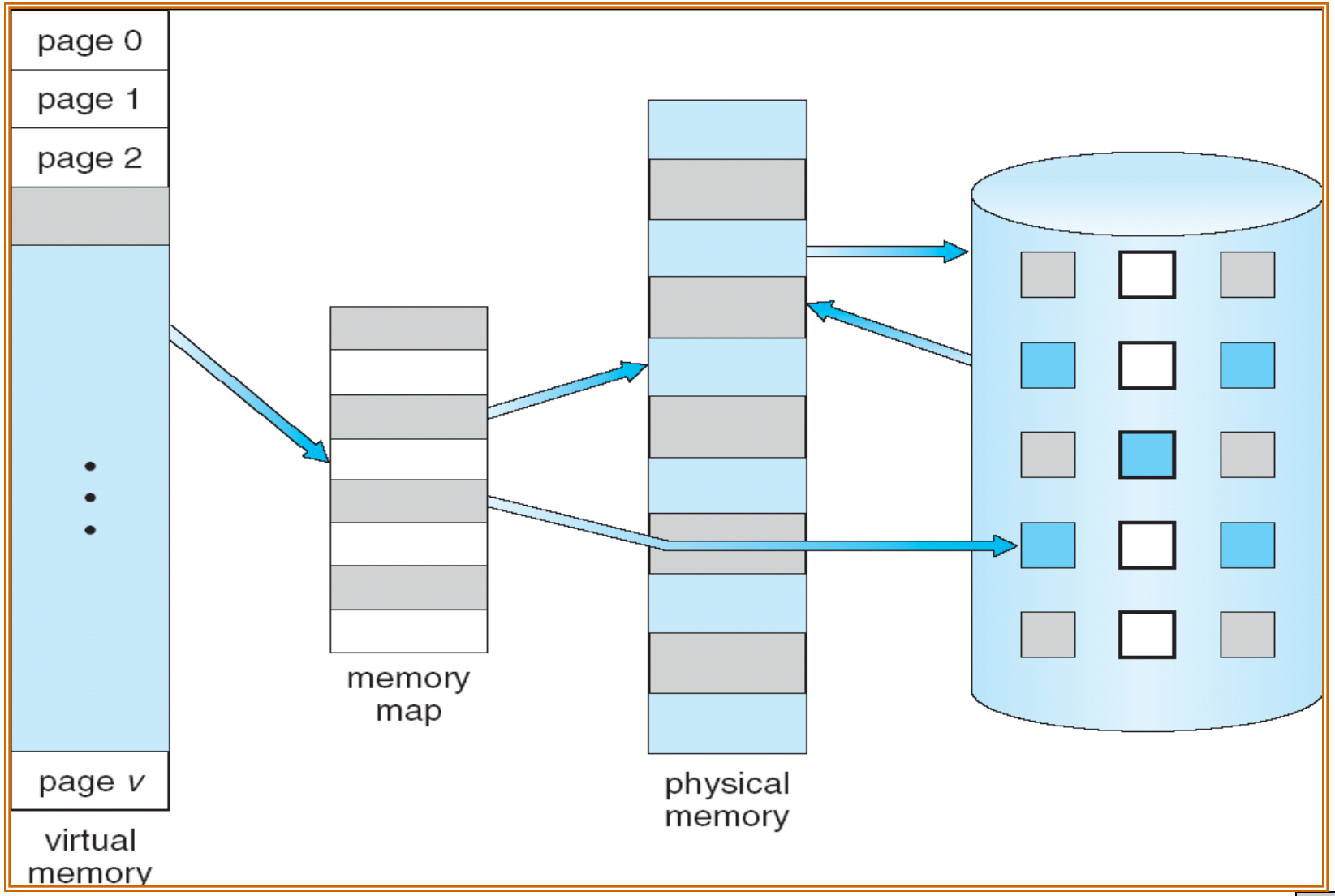
Virtual Memory (cont.)

- Putting these two key elements together allows us to realize that it is not necessary that all of the pages or segments of a process be in main memory during execution.
- If the page or segment that holds the next instruction to be fetched and the page or segment that holds the next data location to be access are both in main memory, then at least for a time, the execution may proceed.
- The ability to execute a program that is only partially in main memory provides many benefits, some of which are:
  - A program is no longer constrained in size by the amount of physical memory. Programmers could write programs for a very large virtual address space.
  - The degree of multiprogramming can be increased since each user program could take less physical memory.
  - Less I/O is required to load or swap each user program into memory, so the program would execute faster.

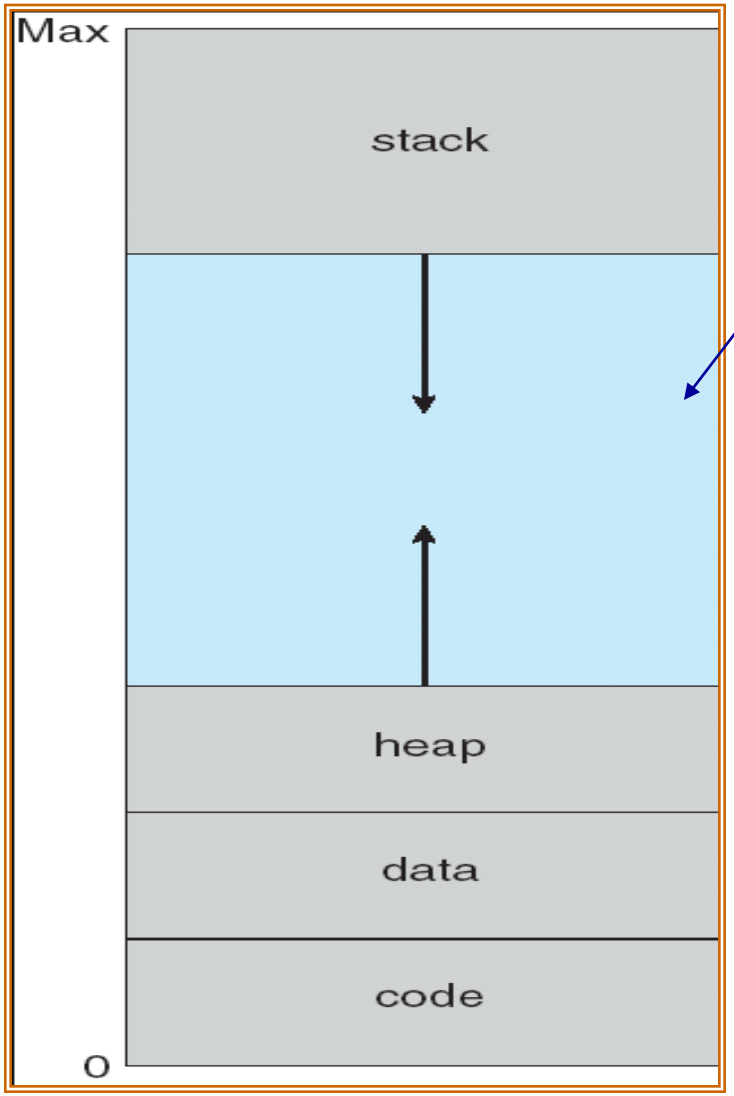
Virtual Memory (cont.)

- Virtual memory involves the separation of logical memory as perceived by users from the physical memory.
- This separation allows an extremely large virtual memory to be provided for programmers when only a much smaller physical memory is available.
- The drawing on page 6 illustrates the basic concept of virtual memory.
- The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory. The figure on page 7 illustrates the virtual address space of a process.

Virtual Memory That is Larger Than Physical Memory



Virtual-address Space

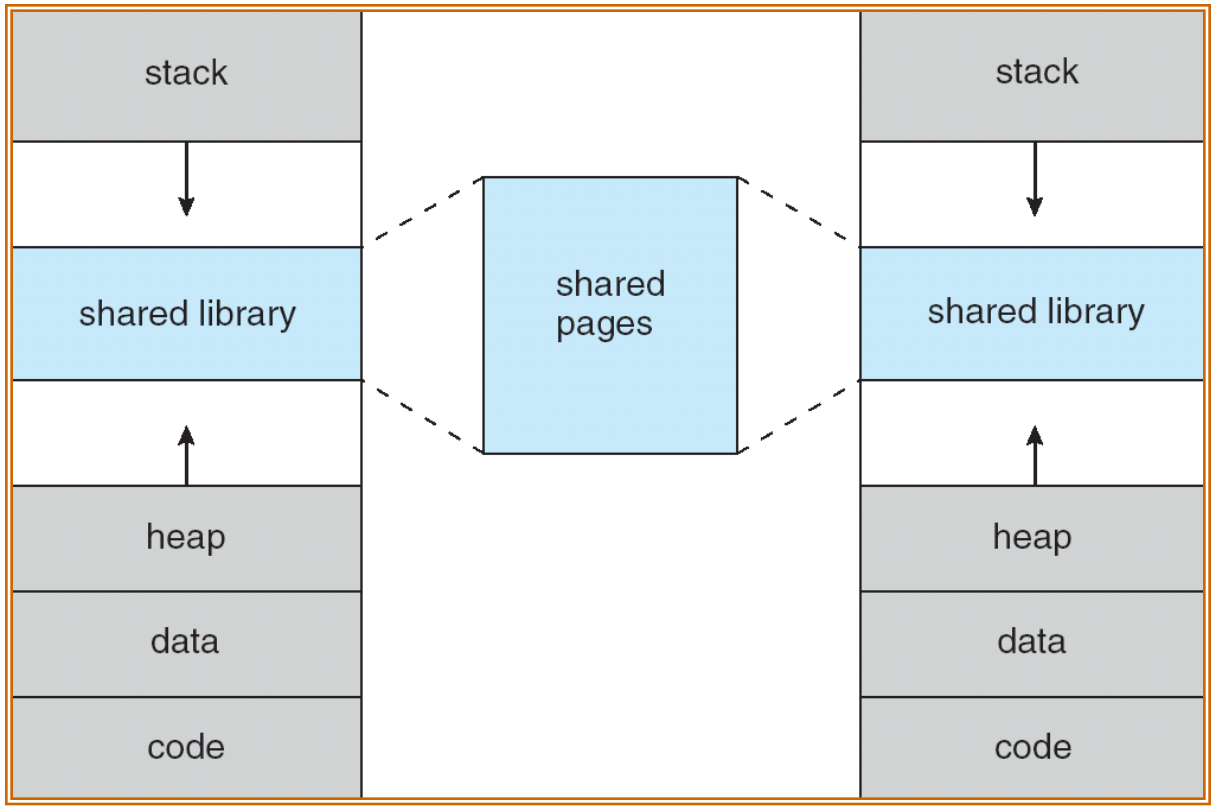


The run-time stack (function calls, etc.) grows vertically downward in memory and the heap (dynamic allocation) grows vertically upwards. The blank space between the heap and the stack is part of the virtual address space, but requires actual physical pages only if the heap or stack grows.

Virtual Memory (cont.)

- In addition to separating logical memory from physical memory, virtual memory also 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 shared libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all of the processes. (See figure on next page.) Typically, a library is mapped read-only into the space of each process that is linked to it.
  - Processes can share memory, by allowing 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.
  - Virtual memory can allow pages to be shared during process creation with the `fork()` system call, thus speeding up process creation.

Shared Library Using Virtual Memory



Virtual Memory (cont.)

- Virtual memory can be implemented using two techniques:
  - Demand paging – As with simple paging, all memory pages and process pages are of the same fixed size. The difference is that in demand paging, only part of the process needs to be in main memory. Process pages are loaded "on demand" as they are needed.
  - Demand segmentation – Memory segments are of varying size as with simple segmentation, however, as with demand paging, process segments are loaded "on demand" as they are needed.
- Other than the addressing issues that we examined for both simple paging and simple segmentation, the implementations of demand paging and demand segmentation are very similar. As such, we will focus primarily on demand paging.

Demand Paging

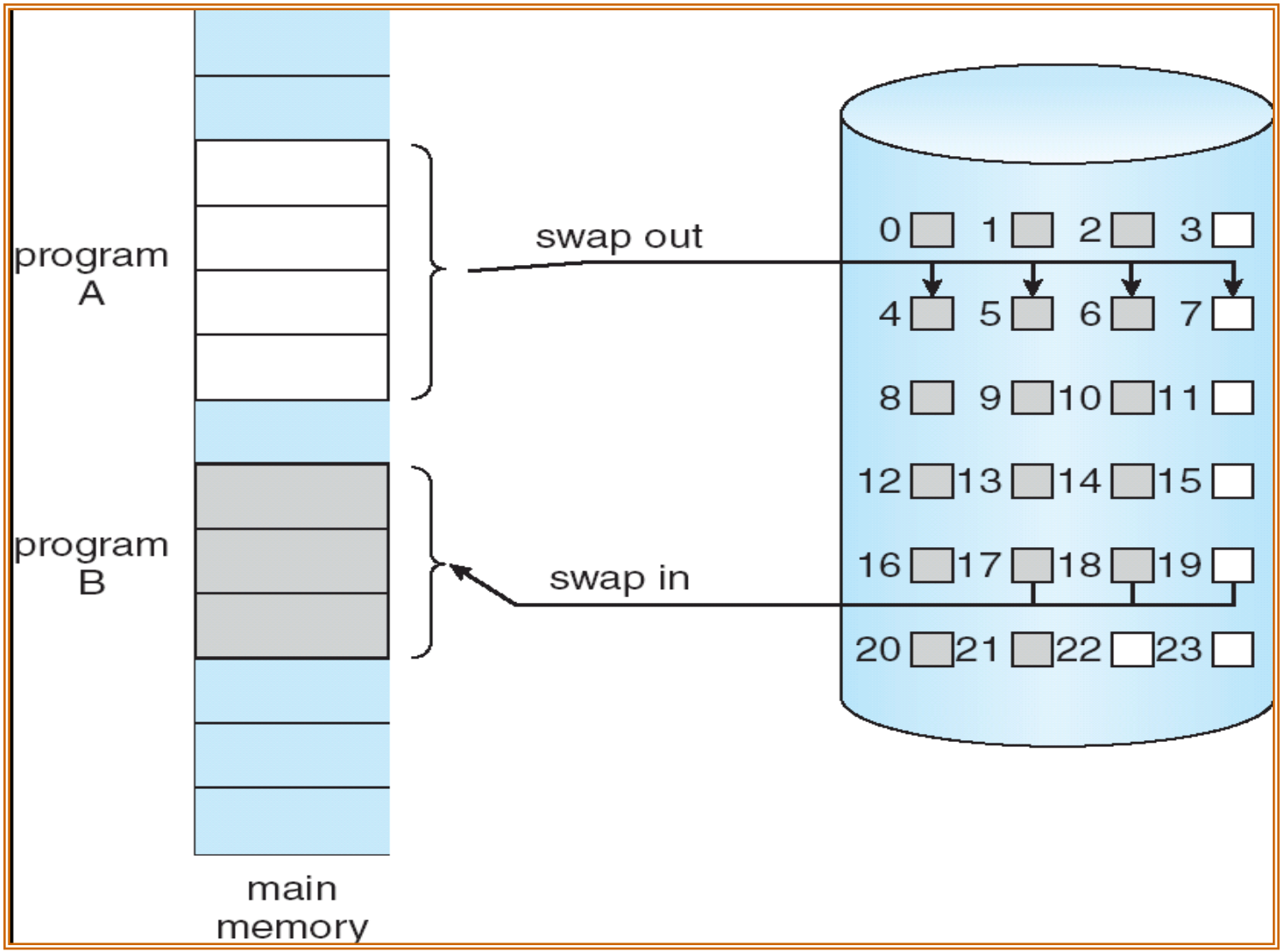
- Demand paging is similar to simple paging with swapping where processes reside in secondary memory (usually a disk, commonly referred to as the backing store).
- When a process is scheduled to be executed, it is swapped into memory. Rather than swapping the entire process into memory, a lazy swapper is used.
- A lazy swapper never swaps a page into memory unless that page will be needed.
- Note: Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of the term *swapper* is technically incorrect. A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. In conjunction with demand paging the term pager should be employed.

Demand Paging (cont.)

- Since a page is loaded into main memory only when it is needed, the question arises as to how does the system know when a page is needed?
- A page is needed whenever a program statement makes a reference (addresses a location) to a page.
- When a reference is made to a page one of three situations will arise:

1. The referenced page is already in main memory. Action: do nothing.
2. The referenced page is not already in main memory. Action: load page.
3. The reference is invalid (out of range, etc.). Action: abort process.

Transfer of a Paged Memory to Contiguous Disk Space

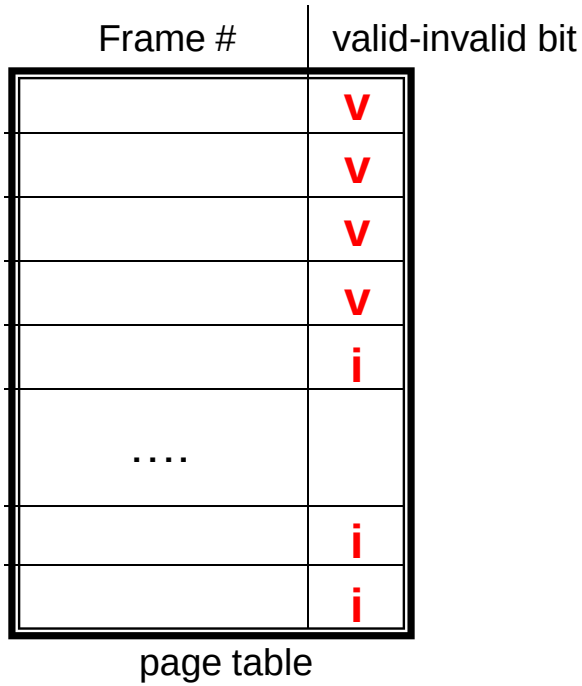


Demand Paging – Basic Concepts

- When a process is to be swapped in, the pager "guesses" which pages will be used before the process is swapped out again.
- By swapping in only those pages that will be used, the swap time is decreased as is the amount of main memory that is required.
- Hardware support is required to distinguish the pages that are in memory and those that are on the disk (backing store).
- The valid-invalid bit scheme that we discussed in the context of simple paging can be extended for use in this situation.

Valid-Invalid Bit

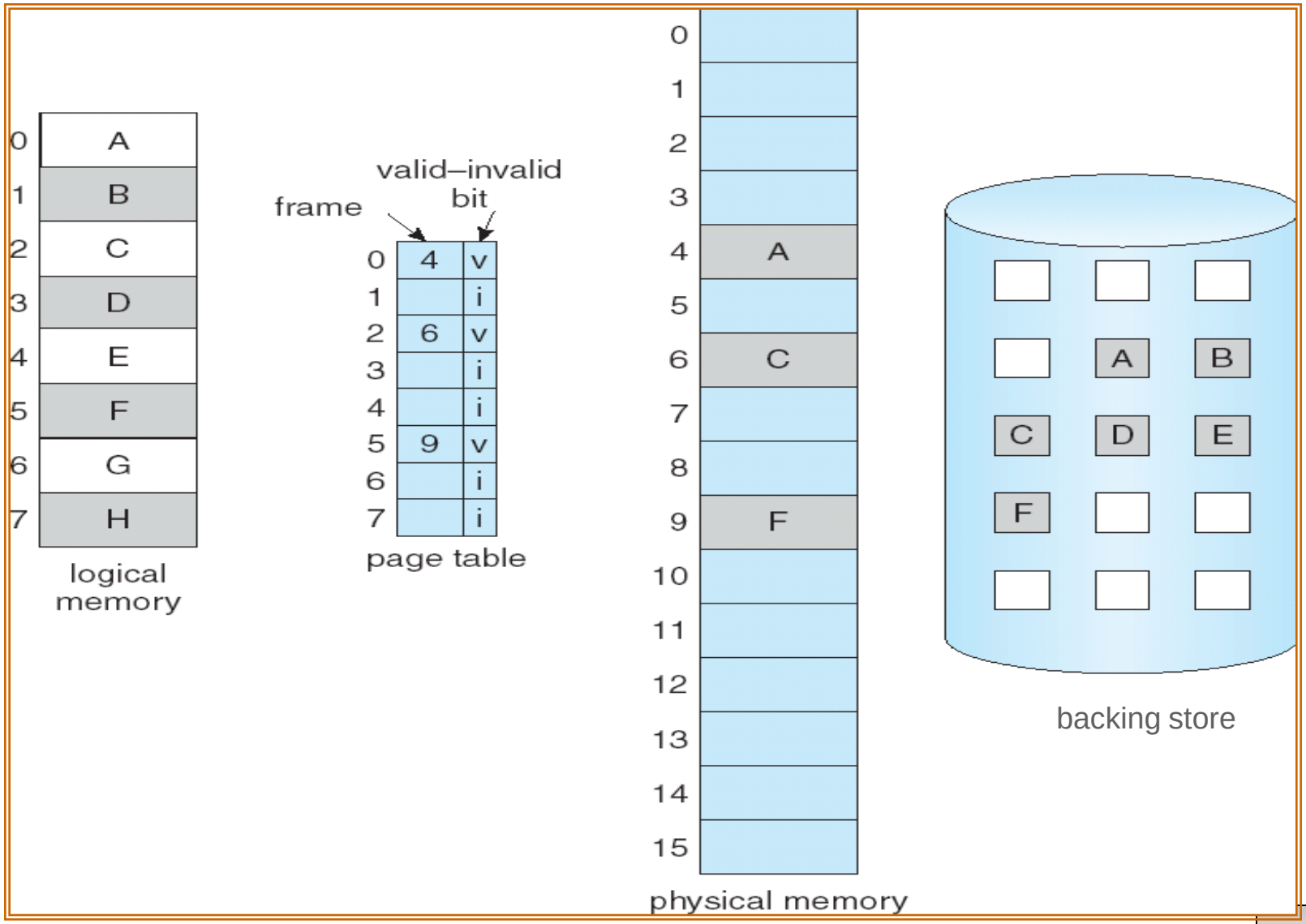
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:
- During address translation, if valid–invalid bit in page table entry is **i** ⇒ page-fault



Demand Paging – Basic Concepts (cont.)

- With demand paging, if the bit is set to "valid", the associated page is both legal (within the logical address space of the process) and in memory.
- If the bit is set to "invalid", the page is either not within the logical address space of the process or is a valid address but is currently not in the main memory (i.e., it is on the disk).
- A page table entry for a page that is currently in memory is set in the same manner as with the simple paging scheme, but the page table entry for a page that is not currently in memory is either simply marked invalid, or contains the address of the page on disk.
- This is illustrated in the figure on the next page.

Page Table When Some Pages Are Not in Main Memory



Demand Paging – Basic Concepts (cont.)

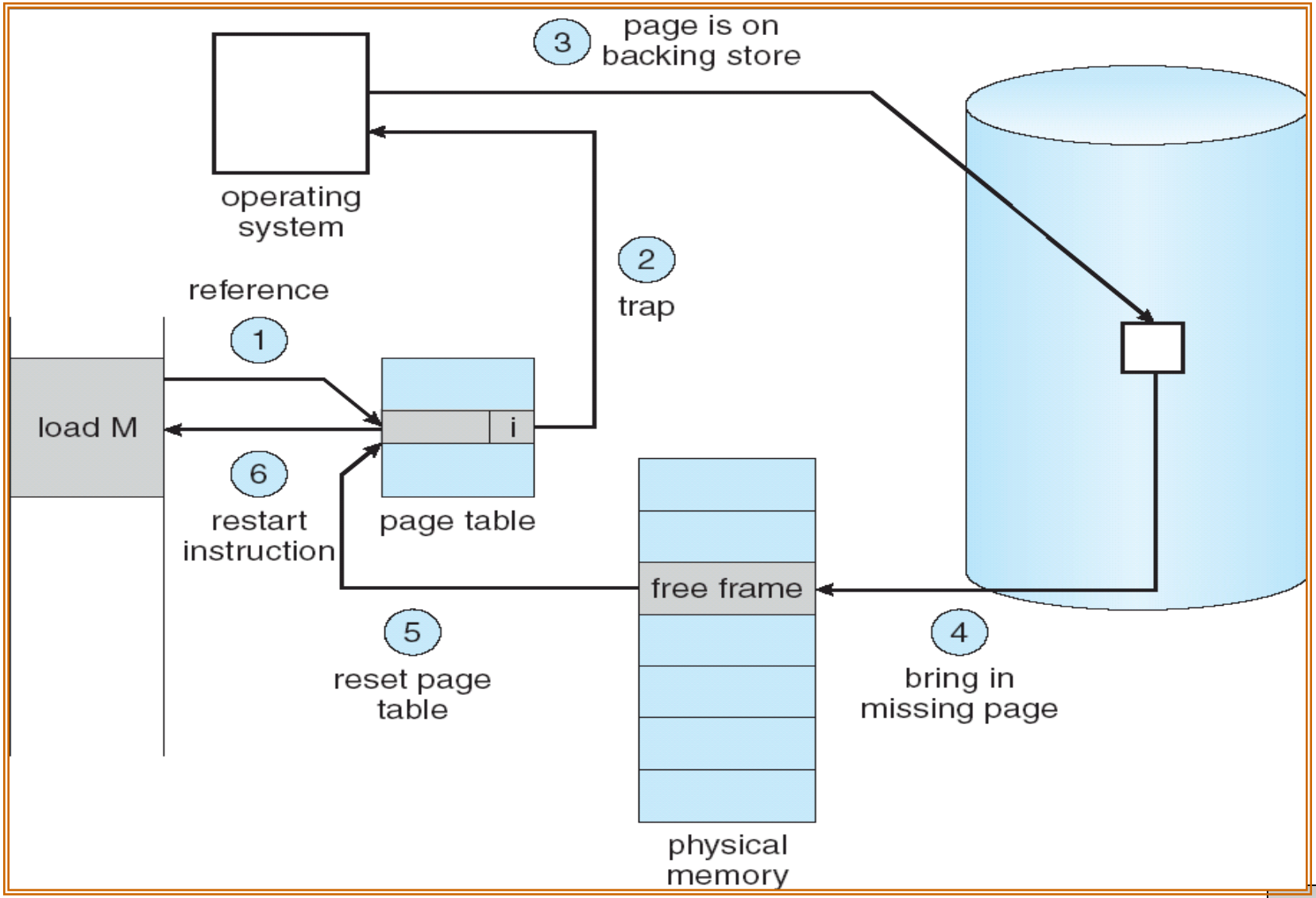
- Notice that marking a page invalid will have no effect if the process never attempts to reference that page.
- Thus, if the pager guesses correctly and pages in only those pages that are actually needed, the process will run exactly as though all of its pages had been loaded.
- As long as the process executes and references pages that are memory resident, execution will proceed normally.
- What happens if the process references a page that is not memory resident? A page-fault occurs.
- A page-fault occurs whenever the paging hardware, in translating the logical address through the page table, encounters the invalid bit set. This generates a page-fault trap. This trap is the result of the OS's failure to bring the desired page into memory.

Page-Faults

- The procedure for handling a page-fault is straightforward and consists of the following steps:

1. Check an internal table (usually maintained with the PCB) for this process to determine whether the reference was a valid or invalid memory reference.
2. If the reference was invalid, terminate the process. If it was valid, but the page is not resident in memory, it must be paged in now.
3. Find a free frame in memory.
4. Schedule a disk operation (I/O) to read the desired page into the newly allocated frame.
5. When the disk read is complete, modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. Restart the instruction that was interrupted by the page-fault. The process will now access the page as though it had always been in memory.

Steps in Handling a Page-Fault



Pure Demand Paging

- In the extreme case, a process can begin executing with no pages in memory. When the OS sets the instruction pointer to the first instruction of a process, which is on a non-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that is needed is in memory. At that point it will execute with no more page-faults.
- This scheme is known as pure demand paging: never bring a page into memory until it is referenced.

Pure Demand Paging (cont.)

- Theoretically, a process could access several new pages of memory with each instruction execution (one page for the instruction and several for data), possibly causing multiple page-faults per instruction.
- Such a situation would result in very poor system performance.
- Fortunately, analysis of running processes shows that this type of behavior is exceedingly unlikely.
- Programs tend to have a locality of reference, which results in reasonable performance from demand paging. We'll deal with issues surrounding locality of reference a bit later.

Hardware Support For Demand Paging

- The hardware required to support demand paging is the same as the hardware for simple paging and swapping.
  - Page table – This table has the ability to mark an entry invalid through a valid-invalid bit or special value of protection bits.
  - Secondary memory – This memory holds those pages that are not present in the main memory. This is typically a high-speed disk. Commonly referred to as the backing store or swap disk. The portion of this disk that is actually used for the swapping operations is referred to as the swap space.

Performance Of Demand Paging

- Demand paging can significantly affect the performance of a computer system.
- In order to understand how this occurs, we'll compute a factor known as the effective access time.
- For most computer systems, the memory-access time, denoted *ma*, ranges from 10 to 200 nanoseconds (10-200 x 10^{-9} seconds).
- As long as there are no page-faults, the effective access time is the same as the memory access time.

Performance Of Demand Paging (cont.)

- If there is a page-fault, first the relevant page must be read from disk and then the desired word within that page must be read.
- Let p be the probability of a page-fault ($0 \leq p \leq 1$). We would expect p to be close to 0, indicating only a few page-faults.
- The effective access time is then:

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

- Thus, to compute the EAT, we must know how much time is required to service a page-fault.

Performance Of Demand Paging (cont.)

- A page-fault causes the following sequence of events to occur:

1. Trap to the OS.
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 location of page on 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 CPU to some other user (only if multiprogramming).
7. Receive an interrupt from the disk I/O subsystem (I/O complete).
8. Save the registers and process state for the other user if step 6 is used.
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to reflect new page in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, new page table, and resume interrupted instruction.

Performance Of Demand Paging (cont.)

- Summarizing the process of handling a page-fault as shown on the previous page, we realize that the process consists of three major components:

  1. Service the page-fault interrupt.
  2. Read in the requested page.
  3. Restart the process.

- The fist and third tasks can be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 μsec each.
- The page-switch time, however, will be closer to 8 msec.
- A typical hard disk has an average latency of about 3 msec., a seek time of around 5 msec., and a transfer time of 0.05 msec.

Performance Of Demand Paging (cont.)

- Thus, the total paging time is around 8 msec., including hardware and software time.
- Remember, that this time includes only the device-service time. If a queue of processes is waiting for the device (other processes that have caused page-faults), the device-queuing time must also be added to our time, further increasing the time required to effect a page swap.
- If we assume an average page-fault service time of 8 msec, and a memory access time of 200 nsec., then the EAT in nanoseconds is: $EAT = (1 - p) * 200 + p\ (8\ msec) = (1 - p) * 200 + p * 8{,}000{,}000$

$= 200 + p * 7{,}999{,}800$

Performance Of Demand Paging (cont.)

- Thus, the effective access time is directly proportional to the page-fault rate.
- If one access out of 1,000 causes a page-fault, then

  EAT = 8.2 msec. This is a slowdown by a factor of 40!!
- If we want the system to be degraded less than 10%, then we need the following to hold:

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

  $20 > 7{,}999{,}800 * p$

  $p < 0.0000025$
- In other words, to keep the slowdown due to paging at a reasonable level, we must allow fewer than one memory reference out of 399,990 to page-fault.
- The page-fault rate must be kept low!

Additional Factors Affecting Demand Paging Performance

- The page size is an important hardware decision that will directly affect the performance of a demand paged system.
- There are several factors to consider.
- One is internal fragmentation. The smaller the page size, the less the amount of internal fragmentation. To optimize the use of main memory we'd like to minimize the amount of internal fragmentation. On the other hand, the smaller the page, the greater the number of pages that will be required per process. More pages per process means larger page tables. For large programs in a heavily multi-programmed environment, this may mean that some portion of the page tables of active processes must be in virtual memory, not in main memory.

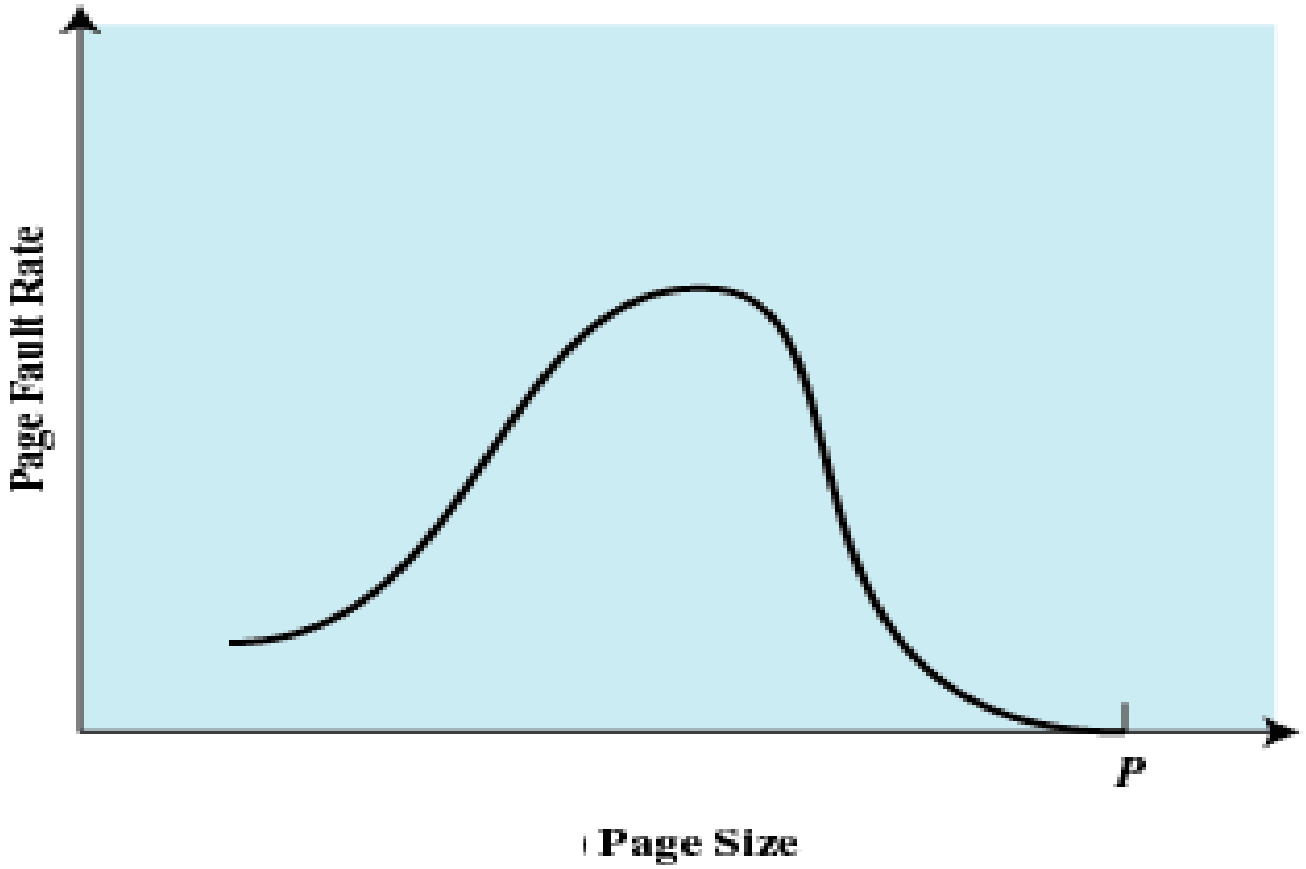
Additional Factors Affecting Demand Paging Performance

- This leads to the unfortunate situation of a double page fault for a single reference to memory: first to bring in the needed portion of the page table and a second to bring in the process page.
- Another factor is that the physical characteristics of most secondary memory devices m which are rotational, favor a larger page size for more efficient block transfer of data.
- Complicating these matters is the effect of page size on the rate at which page faults occur. This behavior is linked closely to the locality of reference (we'll see this in more detail later), but can be summarized in the following manner:

Additional Factors Affecting Demand Paging Performance

- If the page size is very small, the ordinarily a relatively large number of pages will be available in main memory for a process.
- After a time, the pages in memory will all contain portions of the process near recent references. Thus, the page fault rate should be low.
- As the size of the page is increased, each individual page will contain locations further and further from any particular recent reference. Thus the effect of the principle of locality is weakened and the page fault rate begins to rise.
- Eventually, however, the page fault rate will begin to fall as the size of a page approaches the size of the entire process. When a single page encompasses an entire process, there will be no page faults.

Additional Factors Affecting Demand Paging Performance

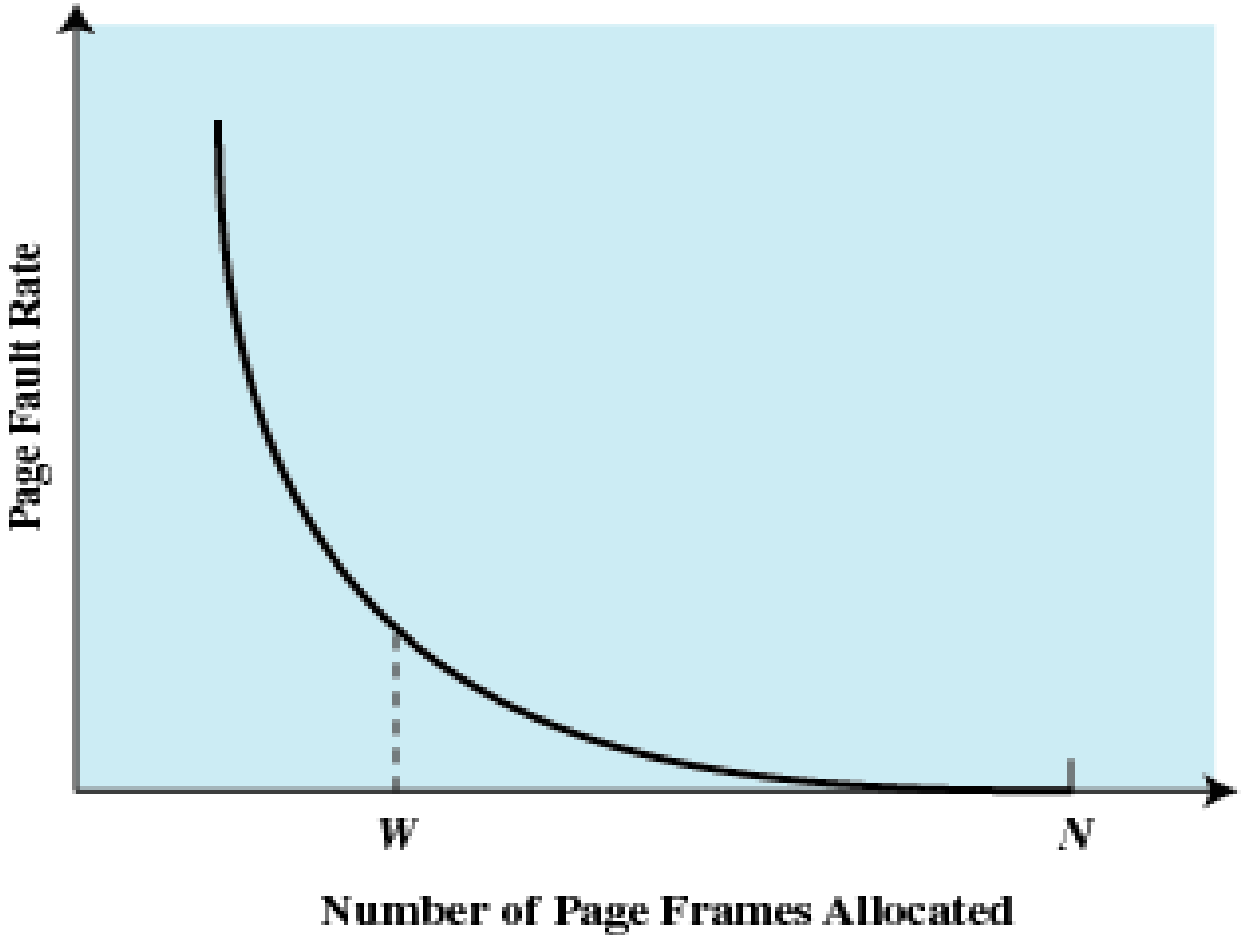


P = size of entire process

Additional Factors Affecting Demand Paging Performance

- A further complication is that the page fault rate is also determined by the number of page frames allocated to a process.
- The diagram on the next page illustrates that, for a fixed page size, the fault rate drops as the number of pages maintained in main memory increases.
- Thus, a software policy (the amount of memory to allocate to each process) interacts with a hardware design decision (page size).

Additional Factors Affecting Demand Paging Performance



Operating System Software

- The design of the memory management portion of an OS depends on three fundamental areas of choice:
  - Whether or not to use virtual memory techniques.
  - The use of paging or segmentation or both.
  - The algorithms employed for various aspects of memory management.
- The choices made in the first two areas depend almost exclusively on the hardware platform available.
  - For example, earlier UNIX implementations did not provide virtual memory because the processors on which the system ran did not support paging or segmentation. Neither of these techniques is practical without hardware support for address translation and other basic functions.

Operating System Software

- The third area is summarized in the table below:

Fetch Policy Demand Pre-paging	**Resident Set Management** Resident set size Fixed Variable Replacement Scope Global Local
Placement Policy	**Cleaning Policy** Demand Pre-cleaning
Replacement Policy Basic Algorithms Optimal LRU FIFO Clock Page Buffering	**Load Control** Degree of Multi-programming