Memory Management

Introduction

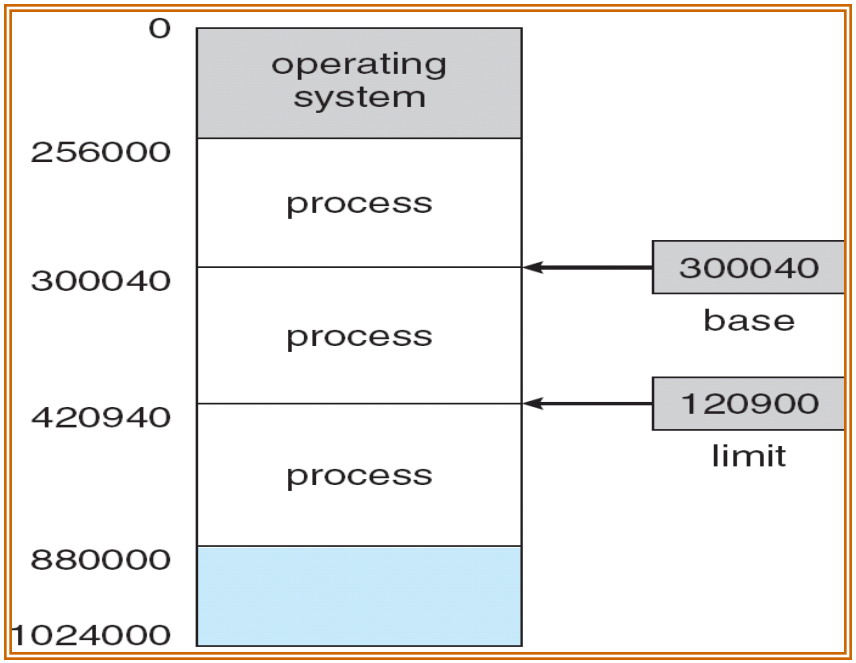
- Program must be brought into memory and placed within a process for it to be run
- **Input queue** – collection of processes on the disk that are waiting to be brought into memory to run the program
- User programs go through several steps before being run

Basic Hardware

- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.
- There are machine instructions that take memory addresses as arguments but there are no instructions that take disk addresses as arguments.
- Thus any instruction in execution and any data used by the instruction must be in one of these direct access storage devices.
- If the data are not in memory, then they must be moved there befor the CPU can operate on them.

- We must be concerned not only with the relative speed of accessing physical memory, but also ensure correct operation to protect the OS from access by user processes from one another.
- This protection must be provided by the hardware.
- It can be implemented in several ways.
- First of all we must make sure 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 and to ensure that the process can access only these legal addresses.

- This protection can be provided by using two registers called a base and a limit



- The base register holds the smallest legal physical address and the limit register specifies the size of the range.
- The program can legally access all addresses from……. To ……

- **Address binding** is the process of mapping the program's logical or virtual **addresses** to corresponding physical or main memory **addresses**.

- The binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location of the process is known at compile time , *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**: 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.

Logical vs. Physical Address Space

- **Logical address –** generated by the CPU
- **Physical address –** address seen by the memory unit. The one loaded into the MAR of the memory.
- The compile time and load time address binding methods generate identical logical and physical addresses.
- ETAB results in differing logical and physical addresses, the the logical address is called the virtual address.
- Set of all logical address generated by a program is a logical address space.
- Set of all physical address corresponding to these logical addresses is a physical address space.

- The runtime mapping from virtual to physical addresses is done by a hardware device called memory management unit(MMU).
- Such mapping we can accomplish through different methods.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.

Dynamic Loading

- The entire program and all data of a process must be in physical memory for the process to execute.
- The size of a process is limited to the size of memory.
- To get better utilization of memory we can use dynamic loading.
- Suppose we are having a program and usually it consists of many routines or modules.
- All modules are kept on disk in a relocatable load format.
- Dynamic loading says that load the main module first and then load other modules when they will be asked for.
- Thus by dynamic loading, all modules will not get jumbled in memory which may or may not get execution in the near future.

Dynamic Linking & Shared Libraries

- With dynamic linking, a stub is included in each library routine reference.
- The stub is a small piece of code that indicates how to locate the appropriate memory resident library routine or how to load the library if the routine is not present.
- When the stub is executed, it checks to see whether the needed routine is already in memory.
- If not, the program loads the routine into memory.
- The stub replaces itself with the address of the routine and execute the routine.

- Thus, next time that particular code segment is reached, the library routine is executed directly.

Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.
- Assume a programming environment with a round robin CPU scheduling algorithm.
- When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into memory space that became free.
- In the mean time the CPU scheduler allocate a time slice to some other process in memory. .

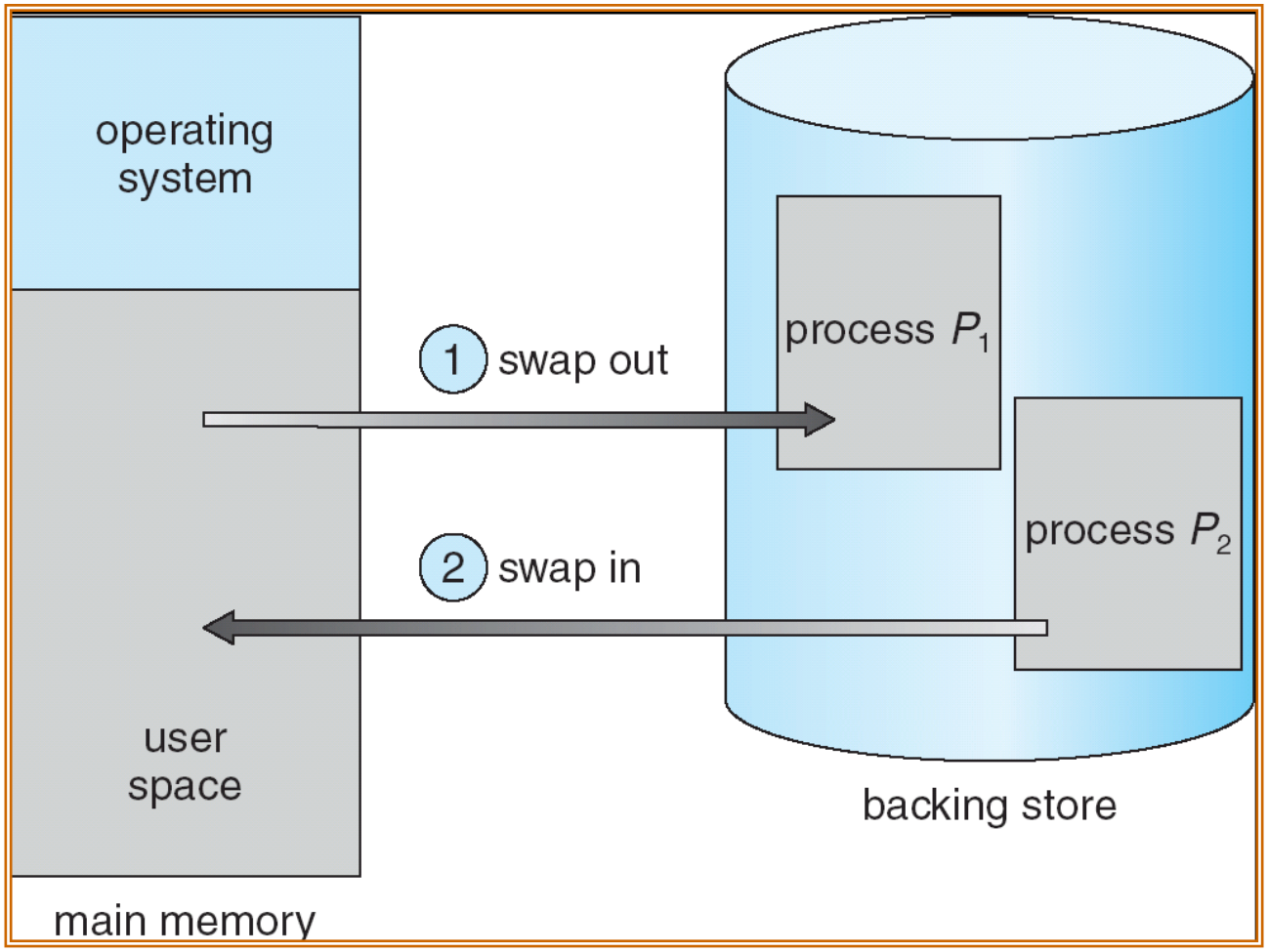
- When each process finishes its quantum, it will be swapped with another process.

- A variant of this swapping process is used for priority based scheduling algorithm.
- If a higher priority process arrives and wants service, the memory manager can swap out the lower priority process and then load and execute higher priority process.
- When the higher priority process finishes, the lower priority process can be swapped back and continued.
- This variant of swapping is called roll-out roll in
- **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.

- A process that is swapped out will be swapped back into the same memory space it occupied previously.
- This is dictated by the method of address binding.
- If binding is done at assembly or load time, then the process cannot be easily moved to a different location.
- If execution time binding is used, then a process can be swapped into a different space because the physical addresses are computed during execution time.

- Swapping requires a swapping store.
- It is a fast disk.
- It must be large enough to accommodate copies of all images for all users.
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk.
- Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.
- The dispatcher checks to see whether 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.

Schematic View of Swapping



Contiguous Memory Allocation

- The main memory must accommodate both the operating system and the various user processes.
- We have to allocate the parts of memory in most efficient manner.
- Contiguous memory allocation is one such allocation method.
- Main memory is usually divided into two partitions:
  - Resident operating system.
  - User processes.

- We can place the OS in either low memory or high memory.
- The major factor affecting this decision is the location of the interrupt vector.
- Interrupt vector is often in the low memory.
- The operating system also resides in the low memory.
- We want several user process to reside in memory at the same time.

- The allocation of available memory to these processes waiting to be brought into memory are to be considered.
- In contiguous memory allocation, each process is contained in a single contiguous section of memory.

Memory Allocation

- The simplest method for allocating memory is to divide memory into several fixed size partitions.
- Each partition may contain exactly one process.
- When a partition is free, a process is selected from the input queue and loaded into the free partition.
- When the process terminates, the partition becomes available for another process.

- In a fixed partition scheme, the OS keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user process and is considered as one large block of available memory.
- It is called a hole.

- When a process arrives and needs memory, we search for a hole which is large enough to fit for that process.
- If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.
- As processes enter the system, they are put into an input queue.

- The operating system checks the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.
- When a process is allocated space, it is loaded into memory, and it can then compete for the CPU.
- When a process terminates, it releases its memory, and the operating system fill this space with another process from the input queue.

- At any given time, we have a list of available block sizes and the input queue.
- The operating system can order the input queue according to a scheduling algorithm.
- Memory is allocated to processes until the memory requirements of the next process cannot be satisfied.
- That is, if no hole is large enough to hold that process.

- 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.
- In general, at any given time we have 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.

- At this point, 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.
- 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.
- The solutions to this problem are
  - first-fit
  - best-fit,
  - worst-fit

- **First fit -**Allocate the first hole that is big enough.
- Searching can start either at the beginning of the set of holes or 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

- Given a memory partition of 150KB, 550KB, 250KB, 350KB and 650KB in order. How would each of the First fit, Best fit and worst fit algorithms place process 220KB, 420KB, 120KB, 430KB in order. Which algorithm will make worst use of memory?