

Memory Management

Chapter 7

Lecture 20

Memory Management

- Accommodates multiple processes in main memory
 - ◆ Hardware support
 - ◆ OS support
- If only a few processes can be kept in main memory at a time, then a high percentage of the time all processes will be waiting for I/O and the CPU will be idle
- Memory needs to be allocated efficiently in order to have as many processes in main memory as possible
- In most OSs, the kernel occupies a fixed portion of the address space and the rest is shared by processes

Memory Management Requirements

- Relocation
 - ◆ Programmer will *not* know where the program will be placed in memory when it is executed
 - ◆ A process may be (often) **relocated** in main memory due to swapping
 - ◆ Swapping enables the OS to have a larger pool of ready-to-execute processes
 - ◆ OS must know: location of process information, execution stack, entry point to begin execution
 - ◆ Memory references in code (for both instructions and data) must be translated to actual physical memory address

Memory Management Requirements

- Protection
 - ◆ Processes should not be able to reference memory locations in another process without permission
 - ◆ Memory protection leads to better security and more reliable operation of the system
 - ◆ It is impossible to check addresses at compile time in programs since the program could be relocated
 - ◆ Address references must be checked at run time by hardware

Memory Management Requirements

■ Sharing

- ◆ Must allow several processes to access a common portion of main memory without compromising protection.

Reasons:

- Cooperating processes may need to share access to the same data structure
- Allow each process to access the same copy of the program code rather than have their own separate copy

- ◆ Mechanisms to support relocation support sharing as well

Memory Management Requirements

■ Logical Organization

- ◆ Users write programs in modules with different characteristics

- Instruction modules are execute-only
- Data modules are either read-only or read/write
- Some modules are private others are public

- ◆ To effectively deal with user programs, the OS and hardware should support a basic form of module to provide the required protection and sharing

- ◆ Segmentation

Memory Management Requirements

■ Physical Organization

- ◆ Main memory holds program and data currently in use:

- high cost and volatile

- ◆ Secondary memory is the long term store for programs and data:

- slower, cheaper and nonvolatile

- ◆ Moving information between these two levels of memory is a major concern of memory management (OS)

- it is highly inefficient to leave this responsibility to the application programmer (a technique called overlays)

Dynamic Linking (1)

■ Libraries

- ◆ **Static linking** – libraries combined a priori into binary program image

- ◆ **Dynamic linking** – linking of libraries is postponed until execution time

■ How:

- ◆ Stub for each library reference

- ◆ Stub indicates how to located memory resident library routine or how to load it if not present

- ◆ Stub replaces itself with address of target routine and executes routine

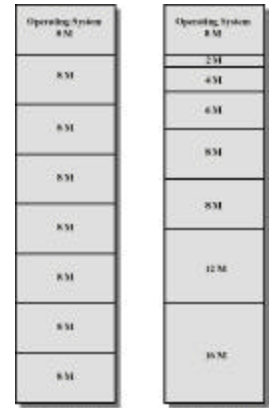
■ Routine is not loaded until called

Dynamic Linking (2)

- What is loaded corresponds to what is used often
 - ◆ Unused routine is never loaded
 - ◆ Good for new versions of libraries
 - ◆ If old programs are statically linked old programs need to be relinked
 - ◆ Dynamically linked programs get the latest libraries during typical execution
 - ◆ Disadvantage: Performance can be poor

Fixed Partitioning

- Partition main memory into a set of non overlapping regions called **partitions**
- Partitions can be of equal or unequal sizes
- A process whose size is less than or equal to a partition size can be loaded into the partition
- If all partitions are occupied, the operating system can swap a process out of a partition

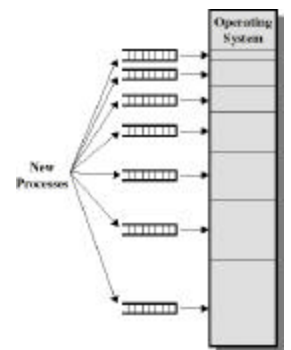


Fixed Partitioning: Problems

- A large program may not fit in a partition. The programmer must then design the program with overlays
- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called **internal fragmentation**.
- Unequal-size partitions lessens these problems but they still remain
- Equal-size partitions were used in early IBM's OS/MFT (Multiprogramming with a Fixed number of Tasks)

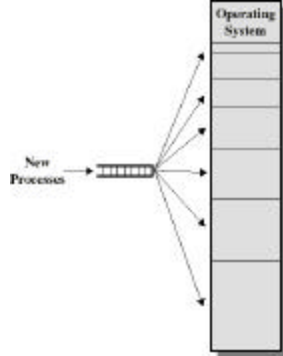
Placement Algorithm with Partitions

- Unequal-size partitions: use of multiple queues
 - ◆ Assign each process to the smallest partition within which it will fit
 - ◆ A queue for each partition size
 - ◆ tries to minimize internal fragmentation
 - ◆ Problem: some queues will be empty if no processes within a size range is present
 - ◆ Effective memory management



Placement Algorithm with Partitions

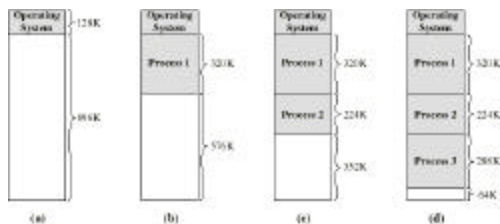
- **Unequal-size partitions: use of a single queue**
 - ◆ When its time to load a process into main memory the smallest available partition that will hold the process is selected
 - ◆ Increases the level of multiprogramming at the expense of internal fragmentation



Dynamic Partitioning

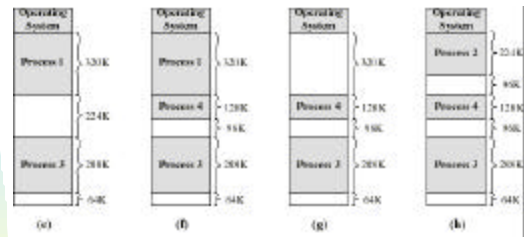
- Partitions are of variable length and number
- Each process is allocated exactly as much memory as it requires
- Eventually holes are formed called **external fragmentation**
- Must use **compaction** to shift processes so they are contiguous and all free memory is in one block
- Used in IBM's OS/MVT (Multiprogramming with a Variable number of Tasks)

Dynamic Partitioning: an example



- A hole of 64K is left after loading 3 processes: not enough room for another process
- Eventually each process is blocked. The OS swaps out process 2 to bring in process 4

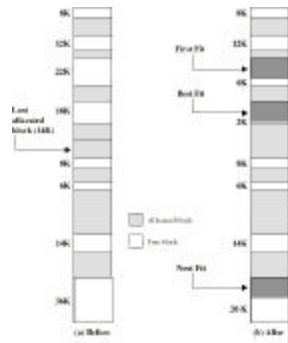
Dynamic Partitioning: an example



- Another hole of 96K is created
- Eventually each process is blocked. The OS swaps out process 1 to bring in again process 2 and another hole of 96K is created...
- Compaction would produce a single hole of 256K

Placement Algorithm

- Used to decide which free block to allocate to a process
- Goal: to reduce usage of compaction (time consuming)
- Possible algorithms:
 - ◆ Best-fit: choose smallest hole
 - ◆ First-fit: choose first hole from beginning
 - ◆ Next-fit: choose first hole that fits starting from last placement
 - ◆ Worst-fit: choose hole that fits the least well



Example Memory Configuration Before and After Allocation of 16 Kbyte Block

Placement Algorithm: comments

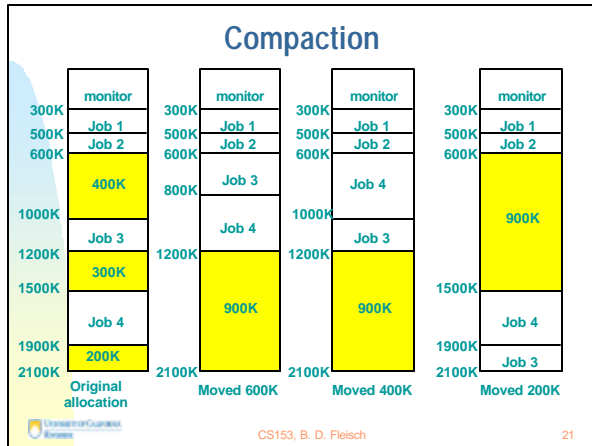
- Next-fit often leads to allocation of the largest block at the end of memory
- First-fit favors allocation near the beginning: tends to create less fragmentation than Next-fit
- Best-fit searches for smallest block: the fragment left behind is small as possible
 - ◆ main memory quickly forms holes too small to hold any process: compaction generally needs to be done more often
- Worst-fit finds the block that is the largest leaving behind a fragment that is as large as possible

Replacement Algorithm

- When all processes in main memory are blocked, the OS must choose which process to replace
 - ◆ A process must be swapped out (to a Blocked-Suspend state) and be replaced by a new process or a process from the Ready-Suspend queue
 - ◆ We will discuss later such algorithms for memory management schemes using virtual memory

Compaction

- Place used blocks and unused blocks in contiguous locations
- Compaction – only works if jobs can be moved to new locations → dynamic relocation
- Determining the best way to compact memory may be difficult



- ### Relocation
- Because of swapping and compaction, a process may occupy different main memory locations during its lifetime
 - Hence physical memory references by a process cannot be fixed
 - This problem is solved by distinguishing between **logical address** and **physical address**
 - Compile time and load time address-binding methods result in an environment where the logical and physical addresses are the same
 - Execution time address-binding results in the logical and physical address are different
- University of Guelph
 CS153, B. D. Fleisch 22

- ### Binding: Options
- **Compile Time**: if it is known at compile time where the program will reside – absolute code could be generated
 - **Load Time** – Compiler must generate relocatable code. Binding is delayed until load time. If starting address changes, reload the code
 - **Run Time** – If program can be moved during its execution, then binding must be delayed until run time. Special hardware must be used. Most general purpose OSs use this method.
- University of Guelph
 CS153, B. D. Fleisch 23

- ### Address Types
- A **physical address** (absolute address) is a physical location in main memory
 - A **logical address** is a reference to a memory location independent of the physical structure/organization of memory
 - Compilers produce code in which all memory references are logical addresses
 - A **relative address** is an example of logical address in which the address is expressed as a location relative to some known point in the program (ex: the beginning)
- University of Guelph
 CS153, B. D. Fleisch 24

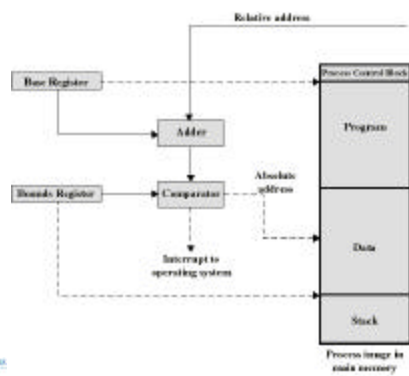
Address Translation

- Relative address is the most frequent type of logical address used in program modules (ie: executable files)
- Such modules are loaded in main memory with all memory references in relative form
- Physical addresses are calculated “on the fly” as the instructions are executed
- For adequate performance, the translation from relative to physical address must be done by hardware

Simple example of hardware translation of addresses

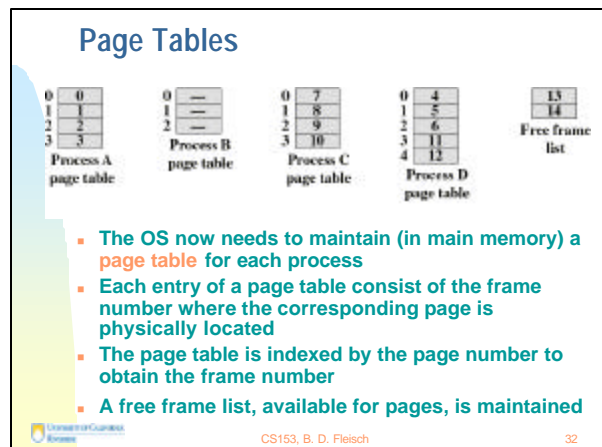
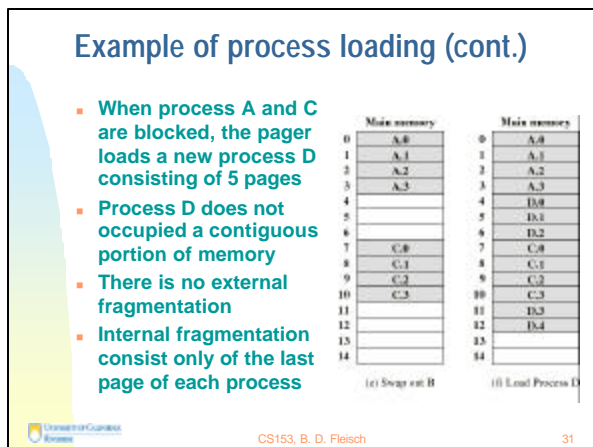
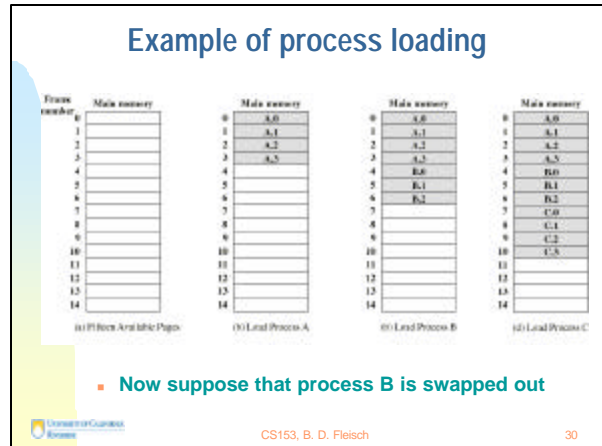
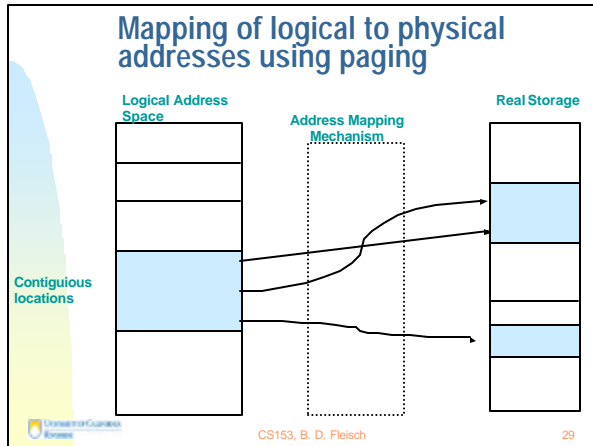
- When a process is assigned to the running state, a base register (in CPU) gets loaded with the starting physical address of the process
- A bound register gets loaded with the process's ending physical address
- When a relative address is encountered, it is added with the content of the base register to obtain the physical address which is compared with the content of the bound register
- This provides hardware protection: each process can only access memory within its process image

Example Hardware for Address Translation



Simple Paging

- Main memory is partition into equal fixed-sized chunks (of relatively small size)
- Trick: each process is also divided into chunks of the same size called **pages**
- The process pages can thus be assigned to the available chunks in main memory called **frames** (or page frames)
- Consequence: a process does not need to occupy a contiguous portion of memory

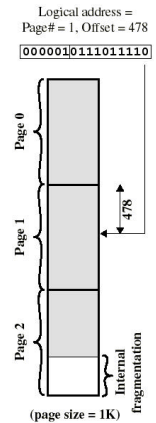


Logical address used in paging

- Within each program, each logical address must consist of a **page number** and an **offset** within the page
- A CPU register always holds the starting physical address of the page table of the currently running process
- Presented with the logical address (page number, offset) the processor accesses the page table to obtain the physical address (frame number, offset)

Logical address in paging

- The logical address becomes a relative address when the page size is a power of 2
- Ex: if 16 bits addresses are used and page size = 1K, we need 10 bits for offset and have 6 bits available for page number
- Then the 16 bit address obtained with the 10 least significant bit as offset and 6 most significant bit as page number is a location relative to the beginning of the process



Logical address in paging

- By using a page size of a power of 2, the pages are invisible to the programmer, compiler/assembler, and the linker
- Address translation at run-time is then easy to implement in hardware
 - logical address (n,m) gets translated to physical address (k,m) by indexing the page table and appending the same offset m to the frame number k

Paging Example

