CS 221 Computer Architecture

Week 9: Storage Management

Fall 2001

Course Schedule

W1	Sep 11- Sep 14	Introduction	
W2	Sep 18- Sep 21	Information Representation (1)	(Chapter 3)
W3	Sep 25- Sep 28	Information Representation (2)	(Chapter 3)
W4	Oct 2- Oct 5	Computer Architecture Pep/6	(Chapter 4)
W5	Oct 9- Oct 12	Assembly Language	(Chapter 5)
W6*	**Oct 16- Oct 19**	**High-Level Languages (1)**	**(Chapter 6)**
W7*	**Oct 23- Oct 26**	**High-Level Languages (2)**	**(Chapter 6)**
	Midterm*		
W8	Oct 30- Nov 2	Instruction Sets	(Tanenbaum)
W9	Nov 6 - Nov 9	Storage Management	(Chapter 9)
W10*	**Nov 13- Nov 16**	**Devices and Interrupts**	**(Chapter 8)**
W11	Nov 20- Nov 23	Combinational Logic	(Chapter 10)
W12	Nov 27- Nov 30	Sequential Logic & Review	(Chapter 11)

**Marked classes will be taught by Dave Dove, same place same slot*

Announcements

Thursday's and Friday's classes will be taught by Dave Dove.

Marks will be posted later today.

Plan for Week 9

- **Discuss midterm solution.**
- **Different techniques for managing memory amongst processes**

- **Readings:**
  - Chapter 9 book
  - Next week Chapter 8

I. Intro to Memory Management

Managing Memory Resources

The Operating System provides

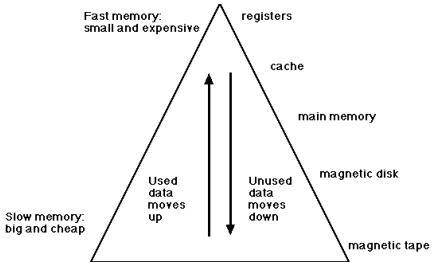
- A convenient programming environment
- Efficient allocation of system resources
- Is responsible for managing the memory programs require

There are a number of issues in memory management:

- ***Memory hierarchy***
  - How do you extend small, fast, expensive memory with big, slow, cheap memory?
- ***Memory allocation***
  - How to divide up memory among different processes so that it is used most effectively?
  - minimizing both the amount of memory that goes unused,
  - and the amount of time that processes can't run because there isn't enough memory to run them?
- ***Memory protection***
  - How do you keep one process from mucking about in the memory assigned to another process?

Memory Hierarchy

- **A memory hierarchy consists of multiple forms of memory of varying speeds and capacities.**



Memory Hierarchy (2)

- **The computer hardware and system software distribute data in memory so that it looks like the system has lots of fast memory.**
  - The hierarchy is economical in that large memory tends to be slow and cheap, while fast memory tends to be small and expensive.
- **Data migration between registers and main memory is handled by the compiler or the programmer.**
  - Both try to use fast registers for local variables and parameters
- **Data migration between cache and main memory is handled by hardware.**
  - A cache is a small amount of especially fast memory placed between the CPU and main memory. It contains copies of data in main memory that is likely to be needed soon by the CPU.
  - Cache memory differs from main memory not only in that it is faster, but also in the way that it is accessed.
  - Cache is organized into key-value pairs, with each key being an address in main memory, and the value the contents stored in main memory
  - When the CPU requests a read from an address in main memory, the cache is searched for an entry that has that address as its key. If such an entry is found the corresponding value is returned to the CPU.
  - If there is no cache entry for the requested address, the data is read from the requested address in main memory, and added to the cache.

Memory Hierarchy (3)

- **Data migration between main memory and magnetic disk is handled by the operating system with support from hardware**
  - With virtual memory, the operating system provides processes with a large block of contiguous memory that is actually on disk.
  - As an address that resides on disk is accessed, hardware reads it in memory first.
  - This process is transparent to program.

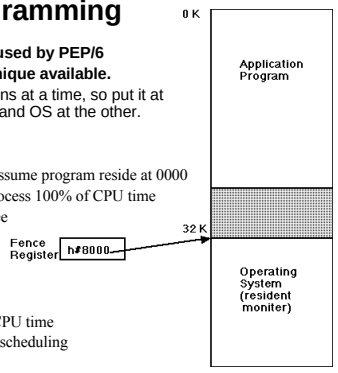
Memory Allocation

- **There are a number of different techniques for allocating memory among processes.**

1. Uniprogramming
2. Fixed partition multiprogramming
3. Variable partition multiprogramming
4. Segmentation
5. Paging
6. Demand-paged virtual memory

Uniprogramming

- **This is the technique used by PEP/6**
- **It is the simplest technique available.**
  - Only one process runs at a time, so put it at one end of memory and OS at the other.
  - Advantages
    - Compiler always assume program reside at 0000
    - Little overhead: process 100% of CPU time
    - Simple and bug-free
  - Disadvantages:
    - Inefficient use of CPU time
    - Inflexibility of job scheduling



If application address >= fence, protection violation.

Uniprogramming (2)

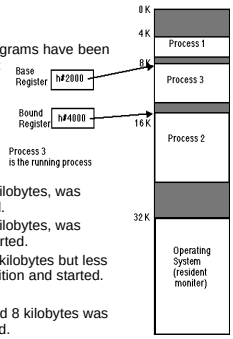
- **Allocation**
  - Since there is only one process in memory at a time, allocating memory among processes is not an issue.
- **Hierarchy**
  - Uniprogramming does permit the use of a storage hierarchy to stretch main memory. A program that is to big to fit could be split up.
  - A system which kept only one process in memory at a time could swap whole processes between the memory and disk.
  - The obvious disadvantage of swapping is that the context switch time is high because you have to read and write secondary storage whenever you switch from one process to another.
- **Protection**
  - It is also possible to have protection. The address where the user program ends and the OS begins is put into a special fence register.
  - To prevent the application program from modifying the operating system, every access to memory while the processor is in user mode can be checked against the value in the fence register, and if it is greater, the hardware triggers a protection violation trap.

Fixed-Partition Multiprogramming

- **With only one application process active**
  - CPU is left idle whenever that process is waiting for input or output.
- **Fixed Partition Multiprogramming is a simple way to have multiple processes active.**
  - More than one process is in memory at one time, with the OS switching between them.
  - The physical memory is divided into different fixed size partitions
  - Each program is loaded into one of the partitions.

Fixed-Partition Multiprogramming (2)

- **The diagram shows a system where**
  - 32 kilobytes available to application programs have been divided up into four fixed-size partitions:
  - 2 of 4 kilobytes each,
  - one of 8 kilobytes, and one of 16.



- **The diagram depicts the situation after**
  1. Process 0, which required less than 4 kilobytes, was loaded into the first partition and started.
  2. Process 1, which required less than 4 kilobytes, was loaded into the second partition and started.
  3. Process 2, which required more than 8 kilobytes but less than 16, was loaded into the fourth partition and started.
  4. Process 0 exited.
  5. Process 3 which requires between 4 and 8 kilobytes was loaded into the third partition and started.

physical address = logical address + base
If physical address >= bound, protection violation

Fixed-Partition Multiprogramming (3)

Allocation Issues

- **Internal fragmentation**
  - occurs when a process is smaller than the smallest available unit of allocation to run it in, so that unusable space is left within the unit of allocation.
  - Processes 1, 2, and 3 all provide examples of internal fragmentation since they are all smaller than the partitions where they are running.
  - Internal fragmentation is an unavoidable consequence of using fixed-sized allocation to store items of varying sizes.
- **External fragmentation**
  - occurs when the total space available is large enough for some process, but because the holes are discontinuous, it isn't possible to fit the process into the available space.
  - If process 3 were to exit, and then a process requiring 11 kilobytes entered the system, our example would exhibit external fragmentation
  - the total free memory (4 + 8 = 12 kilobytes) would be sufficient to run the new process, but because the free memory was divided into two discontinuous holes, it couldn't be used.

Fixed-Partition Multiprogramming (4)

- **Protection : bound or limit register**
  - Protection is provided by the combination of the base register with a bound register which stores the upper limit of the partition into which the process is loaded.
  - If an application tries to access memory that is at an address lower than the value in the base register, or higher than that in the bound register, it causes a protection fault which traps into the operating system.
- **Relocation : base register**
  - Relocation becomes an issue with multiprogramming, since processes don't all start at address 0, and you don't really want the programmer to have to specify ahead of time which address they do start at.
  - Relocation can be dealt with by introducing a distinction between the logical addresses used within the program code, and the physical addresses in memory.
  - When the application program executes an instruction to access memory the hardware adds the value stored in the base register to the operand address, to produce a physical address.

Fixed-Partition Multiprogramming (5)

- **Relocation continued**
  - For example, the instruction `LOADA h#0100,d`
    in process 3 would actually access the physical memory address h#2100.
  - If there were no hardware support the operating system could do relocation as part of loading the process into memory, adding the starting location of the program to every absolute memory reference within the program
- **Note that every time the OS gives the CPU to a new process**
  - the base and bound registers must be loaded with the values appropriate for the new process.

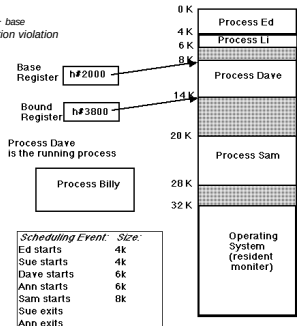
II. Variable Partioning

Variable Partition Multiprogramming

- **Besides fragmentation, fixed partition multiprocessing involves difficult scheduling and partition-sizing decisions.**
  - should a small process be started in a large partition if it is the only one available?
  - should the memory space be divided into a few large partitions (worsening internal fragmentation, but allowing larger processes to run) or many small ones?
- **An alternative is variable partition multiprogramming.**
  - Again, multiple processes are kept in memory
  - But partitions are resized to exactly match the size of the jobs to be run.

Variable Partition Multiprogramming (2)

physical address = logical address + base
If physical address >= bound, protection violation



Scheduling Event:	*Size:*
Ed starts	4k
Sue starts	4k
Dave starts	6k
Ann starts	6k
Sam starts	8k
Sue exits	
Ann exits	
Li starts	2k
Billy starts?	7k

Analysis of VPM

Allocation
- No internal fragmentation: Jobs are loaded into memory using exactly as much space as they require.
- External fragmentation still possible. As processes exit, they leave holes throughout memory.

Algorithms for choosing where to allocate a process include:

best-fit put a new process into the smallest hole big enough to accommodate its memory requirements.
- leave the larger holes intact, available for future large processes.
- by minimizing the difference between the size of the process and the size of hole into which it is loaded, best-fit tends to produce many very small holes.

first-fit put a new process into the first hole found that is big enough to accommodate its memory requirements.
- faster than best-fit, since it doesn't require checking the size of every hole
- produces equally effective use of memory.

worst-fit put a new process in largest available hole, so that space left over after starting the process will also be large.
- less effective in minimizing wasted memory

Analysis of VMP (2)

- **Since all allocation algorithms tend to produce smaller holes than one started out with, there needs to be some method for recreating big holes from the smaller ones.**
  - Recombining adjacent holes into larger holes when a process exits is one method.
  - moving processes about in memory to collect the unallocated memory in one contiguous block, an operation known as compaction.
- **Since the partition sizes are not fixed, with variable partition multiprogramming it is possible for a process to grow over its lifetime.**
  - Implementing is difficult since not all processes are next to a convenient hole.

Hierarchy
- Note that we still don't have any OS support for keeping only the parts of a program actually being used in memory.
- Swapping could be used not only to have more active processes than would fit into memory at one time, but also as a way to manage the memory allocation problems.
- The lack of a sufficiently large hole could be met by swapping out a process, and adding the space that it had been using to the hole adjacent to it.

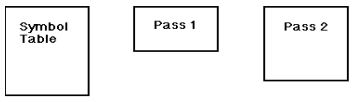
Protection and relocation
- Protection and relocation work just as in fixed partition multiprogramming.

Segmentation

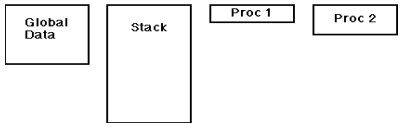
- **A problem with partition multiprogramming is that it puts whole processes into memory**
  - but only a portion of a program's code and data is required at any given time.
  - A solution is to split the program into parts, and have the operating system allocate memory for the pieces of the program instead of the whole thing.
  - This technique is called segmentation
- **With segmentation the operating system provides support for splitting the program up into logical units**
  - that can be allocated memory individually, and which can grow as necessary.
- **A compiler dividing programs into segments might put**
  - global variables into one segment
  - the stack into another
  - and each procedure into a segment of its own.

Example

Assembler divided into segments by programmer:

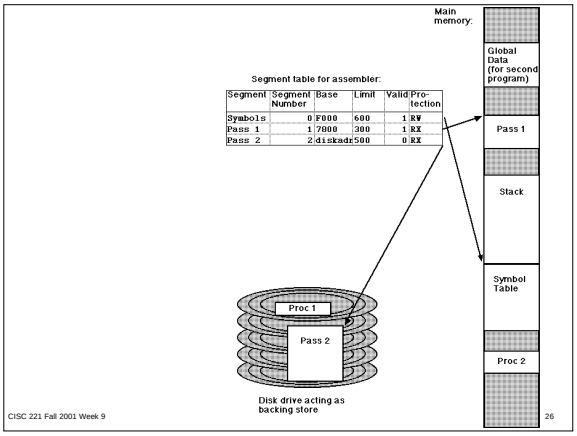


Some other program divided into segments by compiler:



Segment Logical and Physical Addresses

- **Logical addresses in a segmented memory system consist of two parts:**
  - a segment number;
  - an offset within the segment.
- **In a system with 16-bit logical addresses**
  - consisting of a 4-bit segment number and a 12-bit offset (
  - the address h#21f0 is byte h#01f0 in segment 2.
- **The segment numbers are looked up in a segment table.**
  - In a system with a segment table for each process,
  - As part of every context switch in such a system, the operating system would have to set a segment table register to point to the segment table associated with the new process.
- **Here's how the memory might look in a system using segmentation that is running the two programs we divided up before.**
  - The assembler is the current process, running its first pass. Some segments of each program are in memory, and some are not.



Segmentation: Referencing Memory

- **Steps made by the hardware to implement a reference to memory:**
  1. Divide logical address into segment number (2 in our example) and offset (h#01f0), and find segment table entry
  2. If offset is greater than segmentTable[segmentNum].limit, or less than 0, then the reference is to an address outside of the segment. hardware triggers a protection fault that traps into the OS.
  3. If process is trying to write data in a read-only segment, or branch to an address in a segment that is not executable, hardware triggers a protection fault that traps into the operating system.
  4. if segmentTable[segmentNum].valid is not true, that means that the segment is not currently loaded into main memory.
     - The hardware raises a segment fault that traps into the operating system.
     - The operating system will find the segment on disk, read it into memory, set the valid bit to true and the base field to the address in main memory where it has loaded the segment
     - restart the application process with the program counter set so that the application retries the same instruction that faulted the first time.
  - Reading the segment into memory might require ejecting some other segment out of memory to make room, writing to disk
  - 5. Finally, the hardware reads or writes the location whose physical address is given by segmentTable[segmentNum].base + offset

Segmentation: Summary

- **allocation:**
  - no internal fragmentation
  - External fragmentation remains a problem with segmentation, since it faces exactly the same difficulties of allocating variable-sized regions of memory as in variable partition multiprogramming
  - The problem is less severe, since the regions being allocated are smaller: portions of processes rather than entire processes.
- **hierarchy:**
  - only segments being used need to be in memory;
  - segments can be swapped in and out of memory as needed.
- **protection:**
  - implemented using *limit* or *bound* from the segment table.
  - finer grained: segments are smaller than entire processes.
  - can distinguish types of access (i.e, read, write, execute).
  - can share segments between processes.
- **relocation:**
  - performed with base from segment table.

Paging

- **The difficulties with segmentation suggest the idea of paging**
  - divide the program up again, but this time into fixed-sized units, without regard to the logical structure of the program.
  - Thus, you can cut the program in little bits that fit the holes due to fragmentation
- **In Paging, programs are divided into fixed-size units or *pages*.**
  - a typical page size is 4 or 8 kilobytes.
  - Main memory is divided into units of the same size, called frames.
  - Logical addresses consist of page number, and offset within the page.
  - Physical addresses consist of a frame number, and an offset within the frame (which is the same as an offset within a page).

See Figure 9.7 in the text for an example of a paged memory system.

Paging: logical to physical addresses

- **Steps performed by hardware to access a page:**

1. Find entry for the page number in the page table.
   - Each process has its own page table. During a context switch, the operating system sets the registers in the memory management unit to point to the page table for the new current process.
2. If access is inappropriate, trigger a protection fault that traps into OS
   - As with segmentation, each page can have protection bits associated with it that allow a process to read, write, or execute the contents of the page.
3. If the page table entry's valid bit is not set, that means that the page is not currently in main memory.
   - hardware will raise an exception called "page fault", and trap into the OS
   - The operating system will read the missing page in from disk, possibly replacing a page that is already in memory.
   - OS restarts the application program at the instruction that caused the fault.
4. Finally, if the memory reference passes the protection checks and when the desired page is in memory
   - hardware accesses the value at the physical address given by

```
pageTable[pageNum].framenum << offsetWidth (10bits) + offset
```

Replacing Pages

- **The frame table is the data structure that the OS uses to choose where to load a newly required page.**
  - When a page needs to be read into memory to service a page fault, the operating system's page fault handler looks in the frame table for an unused frame.
  - If there aren't any, it needs to choose a page that is already in memory that can be removed from memory, freeing up space for the new page.
- **Choosing which page to replace requires a page replacement algorithm.**
  - What you would like to do is to replace the page in memory that will not be accessed again for the longest time possible. This needs prediction.
- **Two page replacement algorithms are**
  - FIFO: First-in first-out.
  - Replace the page which has been in memory the longest.
  - This is not a particularly reliable guide to which page is unlikely to be accessed soon in the future.
  - LRU: Least recently used.
  - Replace the page in memory which hasn't been accessed for the longest time. Works better than FIFO, but is expensive to implement.
  - Most systems implement some approximation of LRU.

Paging: Summary

- **allocation:**
  - no external fragmentation, since memory is allocated in fixed size units.
  - there is internal fragmentation, but less severe because the allocation units (pages) are relatively small.
- **hierarchy:**
  - can move code and data between backing store and main memory in page-sized units.
- **protection: on a page basis.**
- **relocation:**
  - through the frame numbers in the page table.
  - A page which gets loaded back into memory after having been paged out for a while does not need to be returned to the same frame as it occupied before.

Demand Paged Virtual Memory

- **Virtual memory gives programs the impression that they are working with a large physical memory with a contiguous space**
  - when in fact much of their code and data are stored in secondary storage rather than in main memory,
  - and the portions that are in main memory are not contiguous.
  - The application program's illusion is maintained by making a distinction between the logical or virtual addresses used in application program instructions, and the physical addresses used to access data in main memory.
- **Difference between paging and virtual memory demand paging lies in number of pages a process has in a main memory at any time.**
  - In the case of paging the entire process is loaded into the main memory for execution.
  - In case of demand paging only few pages are initially loaded into the main memory and whenever an access to an unloaded page is requested such page is loaded into the main memory.
  - This leads to a 'working set' of pages of a given process being in the memory at a given time. This set changes as the execution continues.

Demand Paging: Summary

- **allocation:**
  - no external fragmentation, since memory is allocated in fixed size units.
  - there is internal fragmentation, but less severe because the allocation units (pages) are relatively small.
- **hierarchy:**
  - can move code and data between backing store and main memory in page-sized units.
  - Pages are only read in on demand
  - Apparently unused pages are flushed out of memory as new pages are read
- **protection:**
  - on a page basis.
- **relocation:**
  - through the frame numbers in the page table.
  - A page which gets loaded back into memory after having been paged out for a while does not need to be returned to the same frame as it occupied before.

Questions?