CIS2380 Week 5 Lecture

Memory Management (1)

This week we will cover:
(1) Functions of the Memory Manager
(2) Fixed Partitions in Memory
(3) Relocation and Protection
(4) Swapping
(5) Virtual Memory (i): Segmentation

(1) Functions of the Memory Manager It is the job of the operating system - the memory manager, to coordinate memory usage, to provide extra memory when required and to release it when not. This is just one aspect of memory management which can be thought of in terms of five main functions

(a) Memory Allocation Programs are structured into procedures or functions and each has a data and program area. The allocation of memory should reflect this structure and store them as separate units and not just one big block.

(b) Multi-level storage We saw in week 1 that memory is in the form of a hierarchy - where the top layer consists of the registers internal to the CPU, with storage typically about 1KB. The next layer is a small (around 1MB) amount of very fast, expensive cache memory, succeeded by main memory (500MB), and backing store (disk (50GB) and tape (100GB)). Each level is a compromise between speed and cost, and furthermore for parts of a program there will be four copies which have to be consistent with each other. At run time the proportions of program in each of the four layers could be as follows:

- backing store 100 %
- main memory 10%
- cache 1%

registers - one instruction and a few data items. Organising the movement of data between different levels of the pyramid is vital.

(c) Address Mapping or Relocating Only on the most primitive machines can you be sure that a program will be loaded into memory at the same place each time it runs. In general a given job will be run at unpredictable addresses. For example suppose that the first instruction is a call to a procedure at absolute address 100 within the binary (program) file. This absolute address is known as the virtual or logical address. At load time the memory manager might find a suitable free space at 2500 (say) so that the procedure is at address 2600, its physical address. We need a method of mapping between virtual and
physical addresses. This relocation problem could be solved by modifying each instruction as the program is loaded into memory. Thus the relocating loader will change that instruction to a call to 2600. This is called static relocation. However this is cumbersome for a multi-use environment for a program may change its location many times.

(d) Memory protection and sharing This means that users must not interfere with each others memory spaces. For example in the case of the instruction change described above the memory manager would have to ensure that other parts of memory were not unprotected against a malicious (or incompetently written) program. Thus the memory address owned by user B must not be accessed by user A without permission.

(e) Memory Extension The physical address space is limited by (e.g. hardware and cost) and the logical address space of the program is limited by the number of address bits in an instruction. Most modern memory managers allow the logical address space to be larger than the physical address space - and what is more many programs run at the same time. All of this must be carefully controlled

Most systems now allow multiple processing and are not restricted to monoprocessing. Monoprocessing has one program sharing memory between that program and the OS - the only systems using this approach now would be embedded systems. We will confine ourselves to different form of memory management in multiple processing.

(2) Fixed partitions in Memory Multiple processing increases CPU utilisation where one process might be waiting for I/O while another is using the CPU. The easiest way to do this is for memory to be divided into (not necessarily equal) fixed partitions (see Figure 1).

When a job arrives it can be put into the input queue for the smallest partition largest enough to hold it - however any space not used in a partition is lost and a large partition might be empty with at the same time a queue for a small one. An alternative way is to have a single queue - whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run. When a job is loaded into a partition it stays in main memory until it has finished. The disadvantage of fixed partitions is that there is in general not enough main memory to hold all currently active processes so excess processes must be kept on disc. A way must be found of sharing memory and we will look at other ways later.

(3) Relocation and Protection Problems: However we deal with memory management there are relocation problems. It was pointed out earlier that relocation during loading (changing each address from logical to physical) is not very practical and is subject to protection problems. For a
Figure 1: (a) Separate input queues for each partition. (b) A single input queue.

Figure 2: Protection and Address Modification

Further example see Figure 1. If a program with an instruction CALL 100 is loaded in partition 1 (at address 100K) that instruction should be modified to CALL 100K + 100. Sometimes this is done by equipping the machine with two hardware registers called the base and limit registers. (Figure 2.) When a process is scheduled the base register is loaded with the address of the start of its partition (e.g., 100K) and the limit register is loaded with the length of its partition (e.g., 100K). Every memory address has the base added to it - thus a CALL 100 is effectively turned into a CALL 100K + 100 instruction. Addresses are also checked against the limit register to make sure they do not exceed the partition limit and are > 0. A negative logical address would take us into the operating system.
For example Intel processors have registers allowing base addresses including CODE, DATA STACK segments. For many processes process control blocks are used as there are not enough registers. A disadvantage of this approach is the time for addition. We shall return to this problem of relocation later when we look at swapping and segmentation.

(4) **Swapping** Because of the relative smallness of main memory, non-running processes must be kept on disc and brought in to run dynamically. The simplest strategy is **swapping** which consists of bring all of each process to main memory running it for a time, then putting it back on the disk. This can be contrasted with the other approach **virtual memory** which allows programs to run even when they are only partially in main memory. The operation of a swapping system is illustrated in Figure 3:

Initially only process A is in memory. Then processes B and C are created or swapped in from disk. In figure 3(d) A is swapped out to disk. Then D comes in and B goes out and finally A comes in again (g). Note that A will have a different starting address than previously and that the partitions are **variable** in contrast to **fixed** partitions which we saw earlier. When swapping creates multiple holes in memory, it is possible to combine them by moving processes down as far as possible - known as **memory compaction.** In general processes will grow - and may have to be swapped out if there is not enough room.

It is necessary to keep track of where programs are in memory and two methods are used: **bitmaps** and **linked lists.** These can be seen in Figure 4. A bitmap provides a simple way of keeping track of memory - the size of the bitmap depends on the size of memory and the allocation unit. However searching the bitmap is a slow operation for a given amount of free memory requires a given number of consecutive 0s. The linked list also shown is a good way of recording memory usage - for the list can be sorted by address and when a section of memory is free the list can be easily updated. In this
Figure 4: (a) A part of memory with 5 processes and 3 holes - the shaded regions are free. (b) The corresponding bitmap where 0 denotes free (c) The same information as a list

Before X terminates | After X terminates
--- | ---
(a) [A X B] | [A ■ B]
(b) [A X ■] | [A ■ ■]
(c) [■ X B] | [■ ■ B]
(d) [■ X ■] | [■ ■ ■]

Figure 5: 4 neighbour combinations for process X

way merges of free memory are easier (see Figure 5). The situation in Figure 5 (a) requires replacing a P by an H when updating the list as X finishes. In (b) and (c) two entries are coalesced into 1 and in (d) 3 entries are merged. A double-linked list makes it easier to manipulate.

**Algorithms for allocating memory** Given several distinct holes in memory there are several algorithms for determining which free area to allocate to a process. The tutorial examples in week 6 will provide examples.

**First Fit** is the simplest - the memory manager scans along the list of segments until it finds a hole that is big enough. The hole is then filled up from the beginning - a new hole is created of the remaining memory. This is a fast algorithm because there is little searching.

**Next Fit** This keeps track of where it was when it found the last hole. The next hole is chosen starting from there. This performance is slightly worse than first fit.
Best Fit This searches the whole list and takes the smallest that is adequate. For example - in figure 4 if a block of size 2 is needed first fit will allocate a hole at 5 but best fit will allocate one at 18. Best fit is slower than first fit and (surprisingly) wastes more space - because the 'left over' spaces are too small to be useful.

Worst Fit takes the largest available hole (to allow bigger 'left over spaces'). This is not a good choice.

Quick Fit is an example of an algorithm which keeps separate lists for some of the common memory requirements - for example 4KB holes, 8KB holes. Quick fit has the advantage of being extremely fast - but it suffers from the disadvantage (as with the others) that merging newly available 'holes' is expensive. However if merging is not done memory quickly fragments into a large number of useless small holes.

(5) Virtual Memory Usage (i) Segmentation

Virtual memory is a strategy whereby processes can run even when they are only partially in main memory. Since the combined size of program, data, stacks may exceed the amount of physical memory available the operating system keeps those parts of program currently in use in main memory - the rest is on the disk. For example a 16MB program can run on a 4MB machine by carefully choosing which 4MB to keep in memory at each instant. In order to map between virtual and physical addresses a hardware device called the memory management unit (MMU) is used Figure 6 shows the position and function of the MMU indicated as being part of the CPU chip. This is common nowadays - previously it was a separate chip. A general solution is to provide the machine with many independent address spaces called segments . The length of each segment may be anything from 0 to the maximum allowed. Different segments may, and usually do have different lengths. Also the length of a segment may be increased whenever something is pushed onto the stack and decreased whenever anything is popped. The program is treated as an arbitrarily large number of seg-
Figure 7: A segmented memory allows each table to grow or shrink independently of other tables.

ments - each segment reflects a particular part of a program. See Figure 6 which shows a segmented memory for compiler tables which are built up as compilation proceeds.

Each of these segments requires protection. One way would be to implement an arbitrarily large number of base/limit register pairs in the CPU - but this is not feasible. The solution is to implement these registers in ordinary memory - as a table of base/length pairs. This is generally known as a segment table and the elements as segment descriptors. The CPU now has a register which points to the beginning of the segment table of the current process. This changes at every context switch. Each segment begins its logical address at 0. Each segment has a unique number (the high order part of the address) and an offset (the low order part of the address. For example a logical address of 4321 might be interpreted as byte 321 in segment 4. The mapping consists of the following steps:

1. Break the memory reference number into a segment number $s$ and an offset $o$ within the segment.

2. Use $s$ to index into the segment table, find $b$ (the base of this segment in physical memory) and $l$ the segment length.

3. If $o < 0$ or $o > l$ then error.

4. The required address is $b + o$.

For example given logical address of 4321 might be segment 4 and offset 321. We look at the 4th entry in the table and find base address (say 12500) and its length (say 400). The offset is not $< 0$ or $> 400$. It passes both tests. It is then added to the base to give 12821. The result can be summed up: 

logic address 4321 $\rightarrow$ physical address 12821.
Protection and sharing: can be accomplished by adding extra bits to each segment descriptor which signify read write execute access. Each sector has format

\[ r \mid w \mid e \mid \text{base address} \mid \text{length} \mid \]

Cache Memory In order to speed up processing descriptors of most recently accessed segments are cached in the CPU. The table is different from that in figure - as segments are numbered. The cache is cleared on each context switch.

Next Week we shall look at Paging (virtual memory (ii))