8. Storage Management

DATA STRUCTRES AND ALGORITHMS
[17ECSC204]

PRAKASH HEGADE

SCHOOL OF CSE | KLE Tech

Note:
The contents of this chapter is taken from text book as is.

Dynamic Memory Allocation API's

malloc – allocates and reserves a block of memory, specified in bytes and returns a pointer to the first byte of allocated space.
Example: malloc(size)
char *str;
str = (char *)malloc(10);

calloc – allocates multiple block of same size, initializes all locations to zero and returns a pointer to the first byte of allocated space
Example: calloc(n, size)
char* str=NULL;
str = (char *)calloc(10, sizeof(char));

realloc – used to alter the previously allocated space which is allocated by malloc or calloc
Example:
char *str;
str = (char *)malloc(10);
str = (char *) realloc(str, 40);

free –release the memory space that had been allocated by memory allocation functions
Example: char *str;
str = (char *)malloc(10);
free(str);

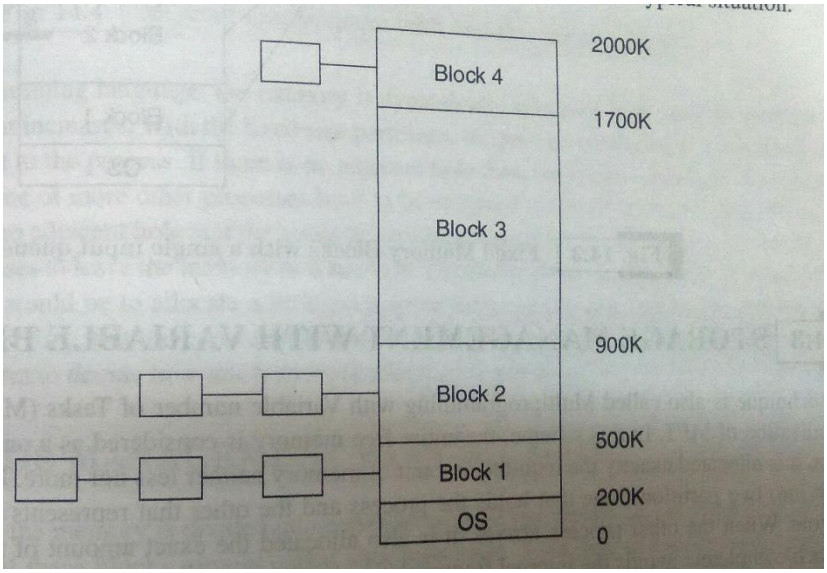
Storage Management with Fixed Blocks

The easiest way of managing memory is to divide it in fixed number of partitions or blocks. The partitions may be of unequal sizes. Since the number of partitions is fixed, the degree of multiprogramming is bounded by the number of partitions. It is easy to understand and equally easy to implement but has become obsolete now.

Example: IBM mainframes in OS/360

In this the memory is partitioned into fixed number of blocks. Each block has an input queue of processes associated with. When the process arrives, the operating system finds its memory requirement and puts in at the tail of the queue of the smallest block big enough to accommodate the process. Whenever the block is free, the process at the head of the corresponding queue is removed from the queue and loaded in the free block and run.

Figure below explains the scenario:



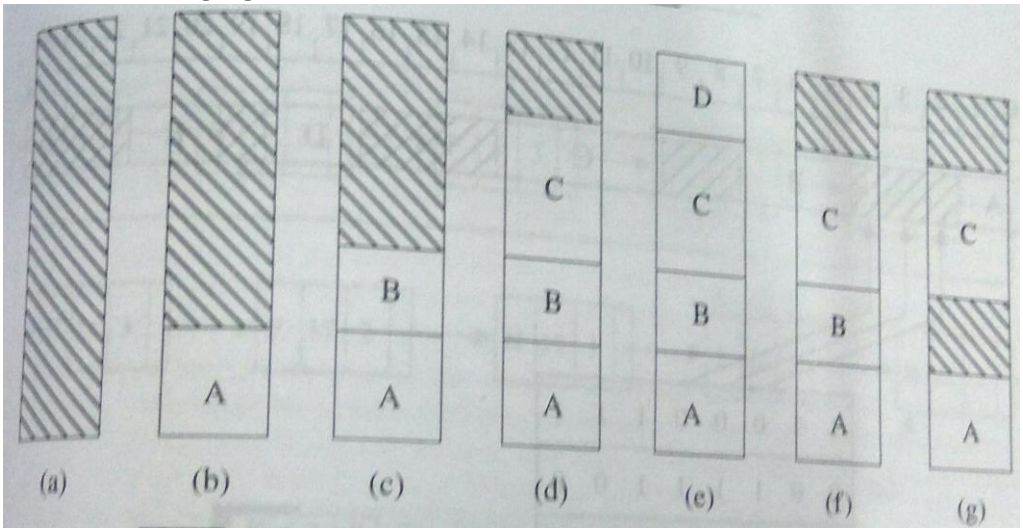
There is inherent disadvantage of this scheme as evident from figure. The queues for small blocks are always large where those for big blocks may be empty. The blocks with empty input queues cannot be used for other processes whereas smaller processes have to wait longer for their turn.

This problem can be solved by maintaining single queue for all blocks together.

Storage Management with Variable Blocks

In this scheme, the entire free memory is considered as one big hole. If the process arrives, it is allocated exactly the required amount of memory, neither less not more. In this way the memory is split into two partitions, one that holds the process and the other one that represents the remaining memory as hole.

Consider the following figure:

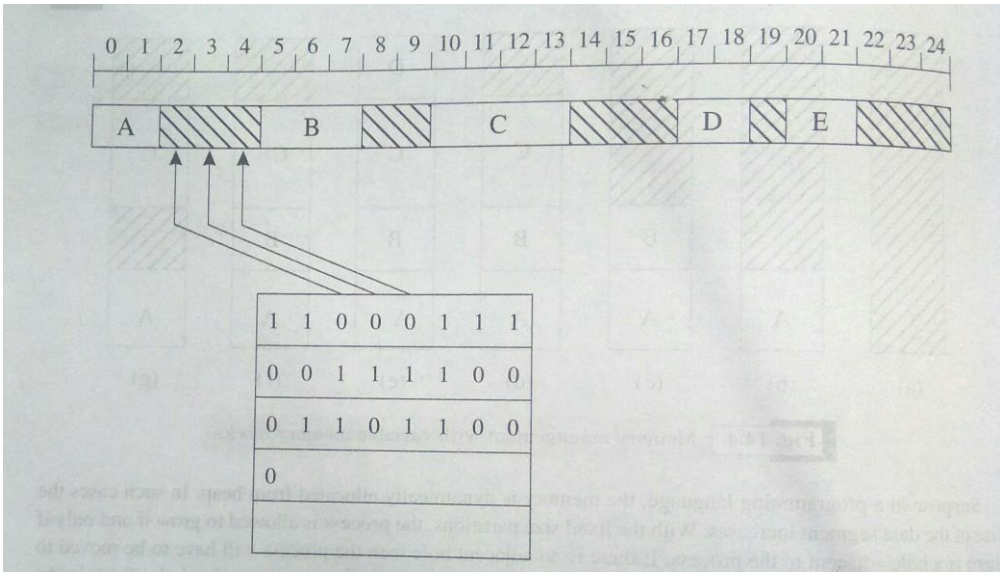


Initially we have 100k available (a). Then 4 processes come with A, B, C and D such that they occupy the available space. The final can be seen in (e). B and D leave the memory creating hole which can be seen in (g).

Think of a situation where a number of processes keep on loading and unloading, then there would be a number of small holes scattered throughout the memory leading to external fragmentation.

Storage Management with Bit Maps

In the bit maps scheme the memory is divided into a number of fixed blocks and processes are allocated memory in multiples of these blocks called allocation units. A bit map is used to keep track of all allocation units. In the bit map, corresponding to each allocation unit, there is a bit indicating whether the allocation unit is free or occupied by the processes. Bit value 1 may represent that the corresponding allocation unit is allocated and 0 may represent that it is free of vice-versa. An example can be seen below:

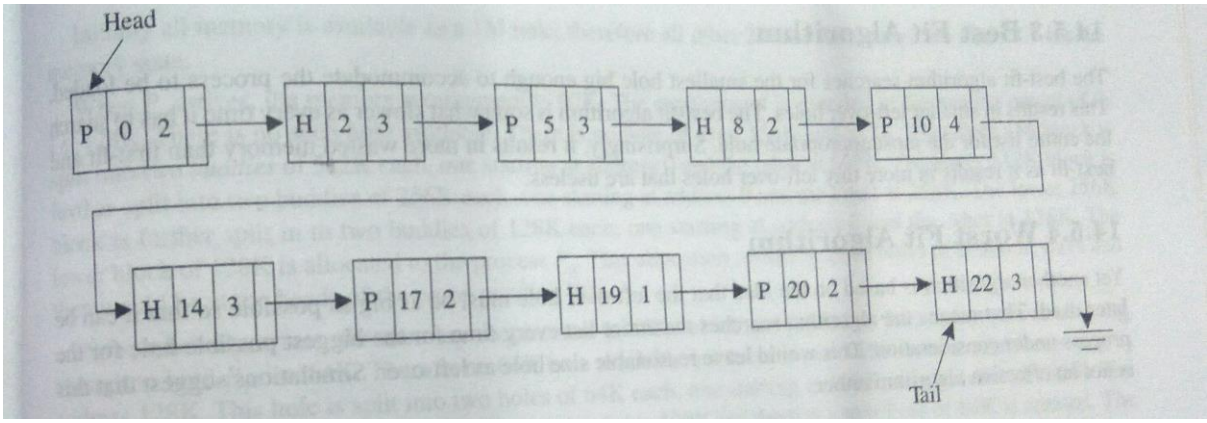


If the main memory size is fixed, then the bit map method is simple to keep track of memory allocation because the bit map depends on only size of memory and that of allocation unit. The disadvantage of the method is that, if a process of m units is to be loaded, then the system has to search for m consecutive 1s in the map which is usually a slow operation. So, practically it is not recommended to use bitmaps very often.

Storage Management Using Linked Lists

In this method, a linked list of holes and processes is maintained. Each node has four fields, an entry indicating P/H, i.e. process/hole, an entry for start location and an entry for size of the process/hole and a pointer to the next entry in the list.

You can refer to the figure below:



The starting locations and the sizes are presented in terms of allocation unit. Though the single linked list is slow, a double linked list is more flexible. Doubly linked list because when a process leaves the memory it is easy to check whether the previous node corresponds to a hole or a process.

When the blocks are sorted by addresses, a number of algorithms are possible for memory management.

First Fit Algorithm:
In this, whenever a process is to be loaded, the linked list is searched from the beginning to find a hole big enough to accommodate the process. The first such hole found is used.

Next Fit Algorithm:
It is based on first fit and only difference is in searching strategy. When a hole is desired for the first time, the algorithm behaves just like the first fit algorithm but this algorithm keeps track of the node in the linked list it is on. When the next requirement comes then the algorithm starts searching from the node it stopped on the previous occasion.

Best Fit Algorithm:
This algorithm searches for the smallest big enough hole to accommodate the process to be loaded.

Worst Fit Algorithm:
Left over hole must be as big as possible so that it can be later used. This means that algorithm searches the entire list every time for the biggest possible hole for the process under consideration.

Quick Fit Algorithm:
In this algorithm we maintain separate lists of some frequently used hole sizes. When a process is to be loaded, it takes minimal time to select the appropriate sized hole.

Storage Release

To manage memory properly we need to handle holes in such a way that the released memory can be used most efficiently. This requires below two points to mention:

1. When a process leaves the memory, a hole is created. We must find the location for this hole in the free list as quickly as possible. This would depend on the way the linked list is organized.
2. When a hole is created due to storage release, how to know whether it can be merged with the previously released one/two holes.

These are addressed by below algorithms:

Liberation Algorithm:

The list of holes is maintained as a double link list in increasing order of the starting addresses of the holes. When a storage block is freed, the hole list is searched from the beginning for the first hole whose starting address is greater than the starting address of the block being released. If no such hole is found then the last existing hole may be contiguous to the block being freed. We check for the appropriate hole whether it can be merged or inserted at appropriate position. On an average, the simple liberation algorithm traverses half of the hole list.

Boundary Tag Method:

This method depends on the marking of block of the memory to distinguish as free/allocated and to decide whether merger of free blocks is possible. Usually a flag is used in each block to indicate whether it is allocated or free. Since it is just a flag and only one bit can do the job and it is packed with a word representing the size.

~*~*~*~*~*~*~