**Memory management techniques**

**Introduction**

Memory management is critical. The performance of the entire system was directly dependent on two things:

1. How much memory was available
2. How the memory was optimized while jobs were being processed.

**The Main Memory**

The main memory (RAM) is a large array of words or bytes, ranging in size from hundreds of thousands to billions. Each word or byte has its own address.

The main memory is the only large storage device that the CPU is able to address and access directly.

**Memory Allocation Schemes**

We shall discuss the following memory allocation schemes:

* Single-User Configurations
* Fixed Partitions
* Dynamic Partitions
* Re-locatable Dynamic Partitions

**Single-User Contiguous Systems**

Single-user systems in a non-networked environment allocate, to each user, access to all available main memory for each job, and jobs are processed sequentially, one after the other.

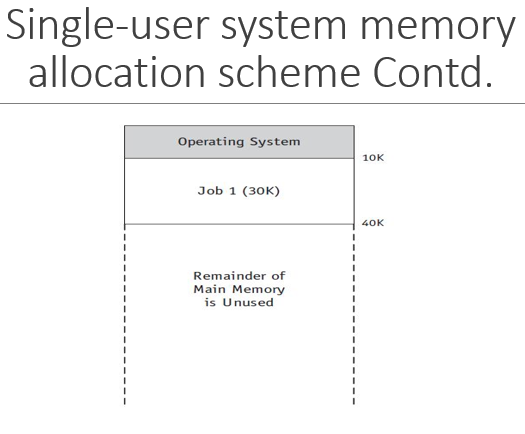
**How the Single-user system memory allocation scheme works**

This memory allocation scheme works as follows:

Before execution can begin, each job or program is loaded in its entirety into memory and allocated as much contiguous space in memory as it needs.

If the program is too large to fit into the available memory space, it cannot begin execution.

This scheme demonstrates a significant limiting factor of all computers—they have only a finite amount of memory. If a program doesn’t fit, then either the size of the main memory must be increased, or the program must be modified to fit, often by revising it to be smaller.



**Memory allocation in SUCS**

To allocate memory, the amount of work required from the operating system’s Memory Manager is minimal, as described in these steps:

1. Evaluate the incoming process to see if it is small enough to fit into the available space. If it is, load it into memory; if not, reject it and evaluate the next incoming process,
2. Monitor the occupied memory space. When the resident process ends its execution and no longer needs to be in memory, make the entire amount of main memory space available and return to Step 1, evaluating the next incoming process.
3. Once the program is entirely loaded into memory, it begins its execution and remains there until execution is complete, either by finishing its work or through the intervention of the operating system, such as when an error is detected.

**Problem with this scheme**

One major problem with this type of memory allocation scheme is that it doesn’t support multiprogramming (multiple jobs or processes occupying memory at the same time); it can handle only one at a time

**Fixed Partitions**

**Fixed (Static) Partitions**

The first attempt to allow for multiprogramming used fixed partitions (also known as static partitions) within main memory—each partition could be assigned to one job.

A system with four partitions could hold four jobs in memory at the same time.

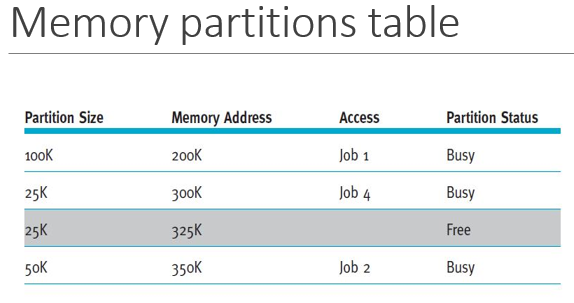
An important factor was introduced with this scheme: protection of the job’s memory space. Once a partition was assigned to a job, the jobs in other memory partitions had to be prevented from invading its boundaries, either accidentally or intentionally.

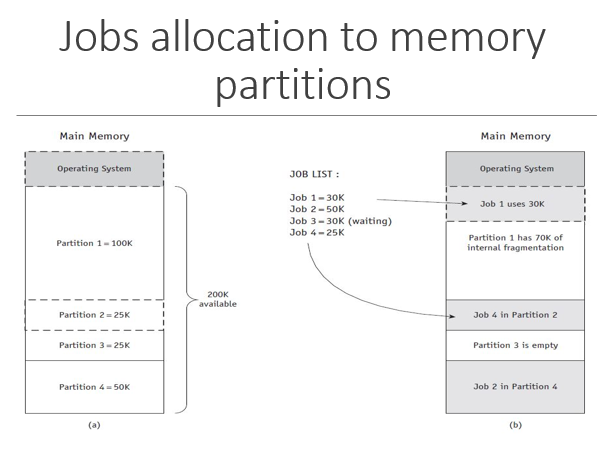
This problem of partition intrusion didn’t exist in single-user contiguous allocation schemes because only one job was present in main memory at any given time—only the portion of main memory that held the operating system had to be protected. However, for the fixed partition allocation schemes, protection was mandatory for each partition in main memory.

**Example of a two-partition system**

The Memory Manager could perform these steps in a two-partition system:

1. Check the incoming job’s memory requirements. If it’s greater than the size of the largest partition, reject the job and go to the next waiting job. If it’s less than the largest partition, go to Step 2.
2. Check the job size against the size of the first available partition. If the job is small enough to fit, see if that partition is free. If it is available, load the job into that partition. If it’s busy with another job, go to Step 3.
3. Check the job size against the size of the second available partition. If the job is small enough to fit, check to see if that partition is free. If it is available, load the incoming job into that partition. If not, go to Step 4.
4. Because neither partition is available now, place the incoming job in the waiting queue for loading at a later time. Return to Step 1 to evaluate the next incoming job.
5. This partition scheme is more flexible than the single-user scheme because it allows more than one program to be in memory at the same time. However, it still requires that the entire program be stored contiguously and in memory from the beginning to the end of its execution
6. In order to allocate memory spaces to jobs, the Memory Manager must maintain a table which shows each memory partition’s size, its **address**, its access restrictions, and its current status (free or busy).





**Problem with Fixed Partition**

Because the partitions were static, so the systems administrator had to turn off the entire system to reconfigure their sizes, and any job that couldn’t fit into the largest partition could not be executed.

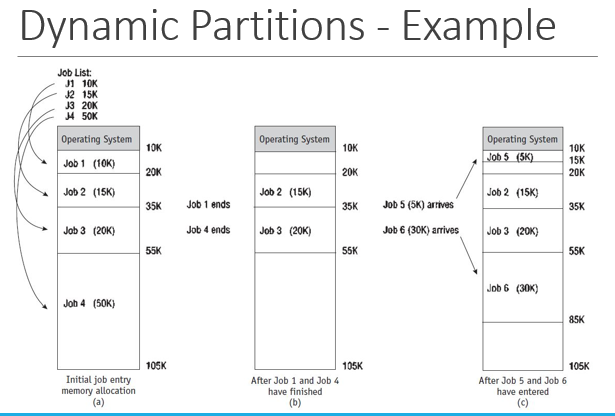
It also leads to ***Internal Fragmentation***; a phenomenon of less-than-complete use of memory space. And it is a major drawback to this memory allocation scheme

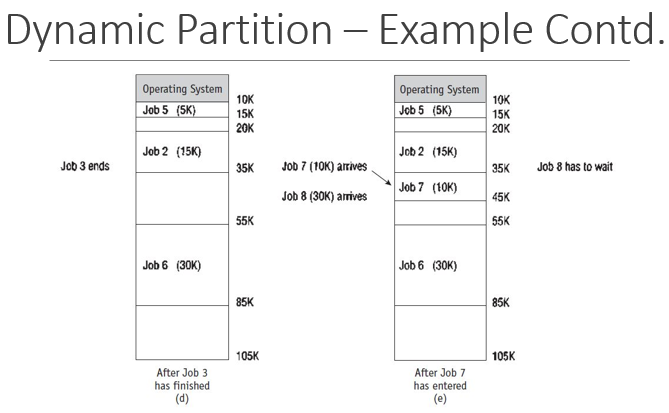
**Dynamic Partitions**

With the introduction of the dynamic partition allocation scheme, memory is allocated to an incoming job in one contiguous block, and each job is given only as much memory as it requests when it is loaded for processing.

A dynamic partition scheme allocates memory efficiently as each of the first few jobs are loaded, but when those jobs finish and new jobs enter the system (which are not the same size as those that just vacated memory), the newer jobs are allocated space in the available partition spaces on a priority basis; demonstrates first-come, first-served priority—that is, each job is loaded into the first available partition.

Therefore, the subsequent allocation of memory creates fragments of free memory *between* partitions of allocated memory. This problem is called **external fragmentation** and, it allows memory to be wasted





**Free partitions**

Notice three free partitions of 5K, 10K, and 20K—35K in all—enough to accommodate Job 8, which requires only 30K. However, because the three memory blocks are separated by partitions, Job 8 cannot be loaded in a contiguous manner. Therefore, this scheme forces Job 8 to wait.

Although the dynamic partition scheme is a significant improvement over fixed partitions because memory is no longer wasted inside each partition, it introduces another problem – ***External Fragmentation***

**Best-fit and first-fit methods**

Memory partitions may be allocated on the basis of first-fit memory allocation or best-fit memory allocation. For both schemes, the Memory Manager keeps detailed lists of the free and busy sections of memory either by size or by location.

The **best-fit allocation method keeps** the free/busy lists in order by size, from smallest to largest.

The **first-fit allocation method** keeps the free/busy lists organized by memory locations, from low-order memory to high-order memory.

Each has advantages depending on the needs of the particular allocation scheme.

Best-fit usually makes the best use of memory space; first-fit is faster.



**De-allocation**

De-allocation refers to the release of memory space.

Until now, we’ve considered only the problem of how memory blocks are allocated, but eventually there comes a time for de-allocation.

De-allocation is very necessary to allow incoming or waiting jobs to execute.

**De-allocation in Fixed Partition**

For a fixed partition system, the process is quite straightforward. When the job is completed, the Memory Manager immediately de-allocates it by resetting the status of the entire memory block from “busy” to “free.”

Any code—for example, binary values with 0 indicating free and 1 indicating busy—may be used, so the mechanical task of de-allocating a block of memory is relatively simple.

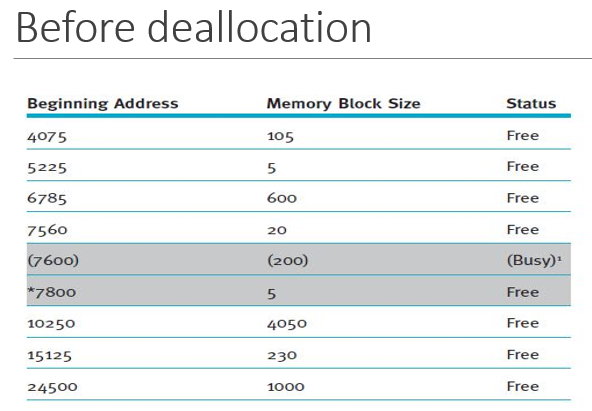
**De-allocation in Dynamic Partition**

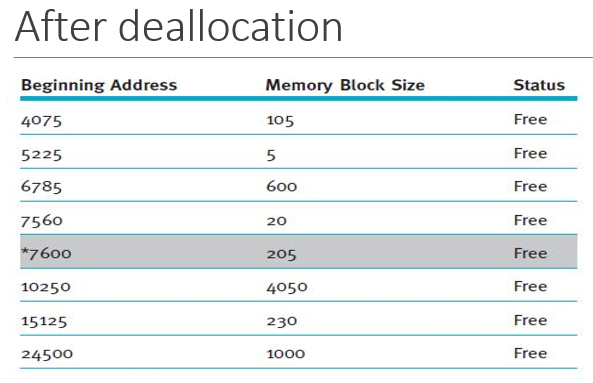
A dynamic partition system uses a more complex algorithm because it tries to combine free areas of memory whenever possible. Therefore, the system must be prepared for three alternative situations:

* Case 1: When the block to be de-allocated is adjacent to another free block
* Case 2: When the block to be de-allocated is between two free blocks
* Case 3: When the block to be de-allocated is isolated from other free blocks

**Joining Two Free Blocks**

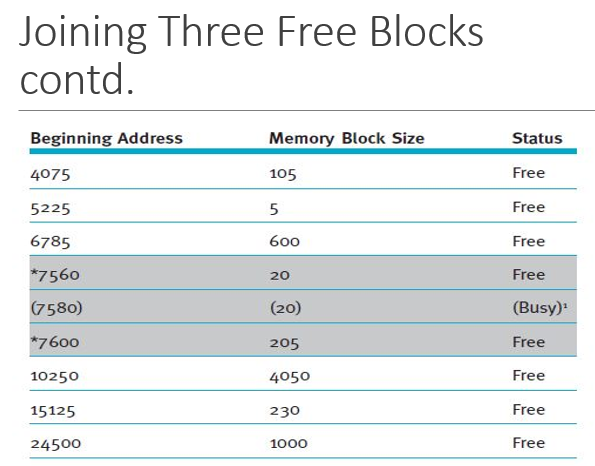
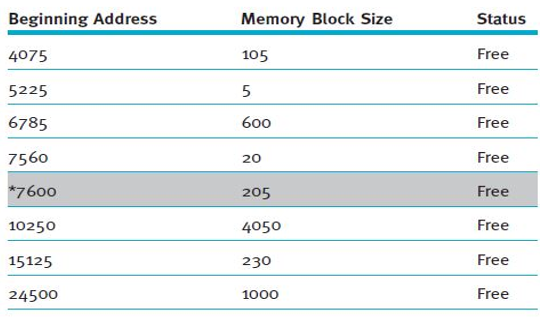
The table below shows how de-allocation occurs in a dynamic memory allocation system when the job to be de-allocated is next to one free memory block before the de-allocation

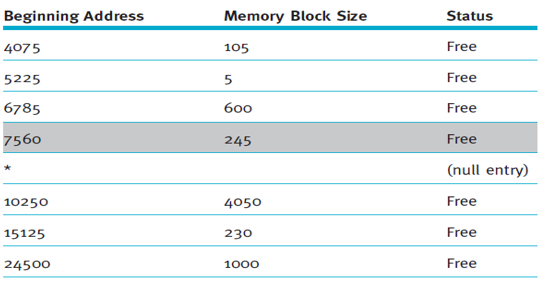




**Joining Three Free Blocks**

Using the de-allocation algorithm, the system learns that the memory to be de-allocated is between two free blocks of memory. Therefore, the sizes of the three free partitions (20 + 20 + 205) must be combined and the total stored with the smallest beginning address, 7560.



**De-allocating an Isolated Block**

Reading assignment

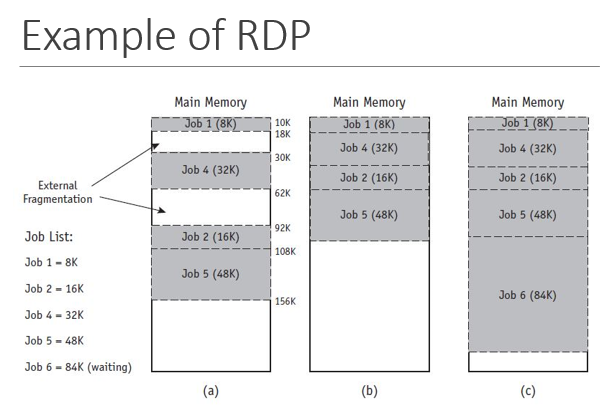
**Re-locatable Dynamic Partition (RDP)**

All of the memory allocation schemes described thus far shared some unacceptable fragmentation characteristics that had to be resolved as the number of waiting jobs became unwieldy and demand increased to use all the slivers of memory often left unused

**How the RDP Works**

In this memory allocation scheme, the Memory Manager relocates programs to gather together all of the empty blocks and compact them to make one block of memory large enough to accommodate some or all of the jobs waiting to get in.

The compaction of memory, sometimes referred to as memory defragmentation, is performed by the operating system to reclaim fragmented space.



Compaction isn’t an easy task. Most or all programs in memory must be relocated so they’re contiguous, and then every address, and every reference to an address, within each program must be adjusted to account for the program’s new location in memory.

However, all other values within the program (such as data values) must be left alone. In other words, the operating system must distinguish between addresses and data values, and these distinctions are not obvious after the program has been loaded into memory.

**Example - Assembly Language**

Here, we show example of an assemble language program instruction.

The instruction to add the integer 1 to I is coded as:

ADDI I, 1

However, after it has been translated into machine code, it could look like this:

000007 271 01 0 00 000001

Now which elements are addresses, references to the address and which are instruction codes or data values?

It’s not obvious at first glance.

The address is the number on the left (000007),

The instruction code is next (271), and the data value is on the right (000001)

Therefore, if this instruction is relocated 200 places, then the address would be adjusted (added to or subtracted) by 200, but the instruction code and data value would not be.