**What Is an Operating System?**

A computer system consists of **software** (programs) and **hardware** (the physical machine and its electronic components). The **operating system** software is the chief piece of software, the portion of the computing system that manages all of the hardware and all of the other software. To be specific, it controls every file, every device, every section of main memory, and every nanosecond of processing time. It controls who can use the system and how. In short, it’s the boss.

Therefore, each time the user sends a command, the operating system must make sure that the command is executed; or, if it’s not executed, it must arrange for the user to get a message explaining the error. Remember: This doesn’t necessarily mean that the operating system executes the command or sends the error message—but it does control the parts of the system that do.

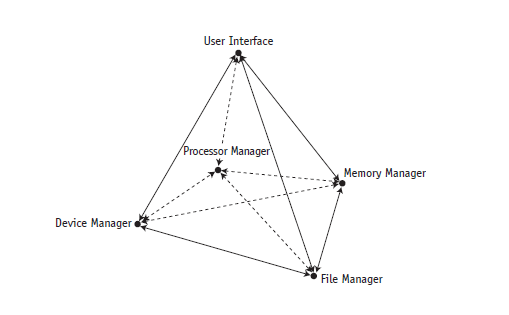
**Operating System Software**

The pyramid is an abstract representation of an operating system and demonstrates how its major components work together.

At the base of the pyramid are the four essential managers of every operating system:

1. The **Memory Manager**
2. The **Processor Manager**
3. The **Device Manager**
4. The **File Manager**.

In fact, these managers are the basis of all operating systems and each is discussed in detail throughout the first part of this book. Each manager works closely with the other managers and performs its unique role regardless of which specific operating system is being discussed. At the top of the pyramid is the User Interface, from which users issue commands to the operating system. This is the component that’s unique to each operating system—sometimes even between different versions of the same operating system.



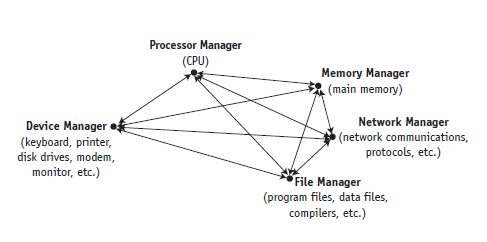
A **network** was not always an integral part of operating systems; early systems were

Self-contained with all network capability added on top of existing operating systems.

Now most operating systems routinely incorporate a **Network Manager**. The base of a pyramid for a networked operating system.

Regardless of the size or configuration of the system, each of the subsystem managers, must perform the following tasks:

1. Monitor its resources continuously
2. Enforce the policies that determine who gets what, when, and how much
3. Allocate the resource when appropriate
4. DE allocate the resource when appropriate



**Main Memory Management**

The Memory Manager is in charge of main memory, also known as RAM, short for Random Access Memory. The Memory Manager checks the validity of each request for memory space and, if it is a legal request, it allocates a portion of memory that isn’t already in use. In a multiuser environment, the Memory Manager sets up a table to keep track of who is using which section of memory. Finally, when the time comes to reclaim the memory, the Memory Manager de-allocates memory.

A primary responsibility of the Memory Manager is to protect the space in main memory occupied by the operating system itself—it can’t allow any part of it to be accidentally or intentionally altered.

**Processor Management**

The Processor Manager decides how to allocate the central processing unit (CPU). An important function of the Processor Manager is to keep track of the status of each process. A process is defined here as an instance of execution of a program.

The Processor Manager monitors whether the CPU is executing a process or waiting for a READ or WRITE command to finish execution. Because it handles the processes’ transitions from one state of execution to another, it can be compared to a traffic controller.

Once the Processor Manager allocates the processor, it sets up the necessary registers and tables and, when the job is finished or the maximum amount of time has expired, it reclaims the processor.

**The Processor Manager has two levels of responsibility.**

1. One is to handle jobs as they enter the system and the other is to manage each process within those jobs. The first part is handled by the **Job Scheduler**, the high-level portion of the Processor Manager, which accepts or rejects the incoming jobs.
2. The second part is handled by the **Process Scheduler**, the low-level portion of the Processor Manager, which is responsible for deciding which process gets the CPU and for how long.

**Device Management**

Device Management monitors every device, channel, and control unit. Its job is to choose the most efficient way to allocate all of the system’s devices, printers, ports, disk drives, and so forth, based on a scheduling policy chosen by the system’s designers.

The Device Manager does this by allocating each resource, starting its operation, and, finally, de-allocating the device, making it available to the next process or job.

**The File Manager**

The File Manager keeps track of every file in the system, including data files, program files, compilers, and applications. By using predetermined access policies, it enforces restrictions on who has access to which files. The File Manager also controls what users are allowed to do with files once they access them.

For example, a user might have read-only access, read-and-write access, or the authority to create and delete files. Managing access control is a key part of file management. Finally, the File Manager allocates the necessary resources and later

De-allocates them.

**Network Management**

Network Managementoperating systems with Internet or networking capability have a fifth essential managercalled the Network Manager that provides a convenientway for users to share resources while controlling users’ access to them. Theseresources include hardware (such as CPUs, memory areas, printers, tape drives,modems, and disk drives) and software (such as compilers, application programs, anddata files).

**The User Interface**

**The user interface** is the portion of the operating system that users interact withdirectly. In the old days, the user interface consisted of commands typed on a keyboardand displayed on a monitor. Now most systems allow users tochoose a menu option from a list. The user interface, desktops, and formats varywidely from one operating system to another.

**Cooperation Issues**

However, it is not enough for each manager to perform its individual tasks. It must also be able to work harmoniously with every other manager. Here is a simplified example. Let’s say someone chooses an option from a menu to execute a program. The following major steps must occur in sequence:

1. The Device Manager must receive the electrical impulses from the mouse or keyboard, form the command, and send the command to the User Interface, where the Processor Manager validates the command.
2. The Processor Manager then sends an acknowledgment message to be displayed on the monitor so the user realizes the command has been sent.
3. When the Processor Manager receives the command, it determines whether the program must be retrieved from storage or is already in memory, and then notifies the appropriate manager.
4. If the program is in storage, the File Manager must calculate its exact location on the disk and pass this information to the Device Manager, which retrieves the program and sends it to the Memory Manager.
5. The Memory Manager then finds space for it and records its exact location in memory. Once the program is in memory, the Memory Manager must track its location in memory (even if it’s moved) as well as its progress as it’s executed by the Processor Manager.
6. When the program has finished executing, it must send a finished message to the Processor Manager so that the processor can be assigned to the next program waiting in line.
7. Finally, the Processor Manager must forward the finished message to the Device Manager, so that it can notify the user and refresh the screen. Although this is a vastly oversimplified demonstration of a complex operation, it illustrates some of the incredible precision required for the operating system to work smoothly. So although we’ll be discussing each manager in isolation for much of this text, no single manager could perform its tasks without the active cooperation of every other part.

**Types of Operating Systems**

Operating systems for computers large and small fall into five categories distinguished by response time and how data is entered into the system: batch, interactive, real-time, hybrid, and embedded systems.

1. **Batch systems** date from the earliest computers, when they relied on stacks of punched cards or reels of magnetic tape for input. Jobs were entered by assembling the cards into a deck and running the entire deck of cards through a card reader as a group—a batch. The efficiency of a batch system is measured in **throughput**—the number of jobs completed in a given amount of time (for example, 550 jobs per hour).
2. **Interactive systems** give a faster turnaround than batch systems but are slower than the real-time systems we talk about next. They were introduced to satisfy the demands of users who needed fast turnaround when debugging their programs. The operating system required the development of time-sharing software, which would allow each user to interact directly with the computer system via commands entered from a typewriter-like terminal. The operating system provides immediate feedback to the user and response time can be measured in fractions of a second.
3. **Real-time systems** are used in time-critical environments where reliability is key and data must be processed within a strict time limit. The time limit need not be ultra-fast (though it often is), but system response time must meet the deadline or risk significant consequences. These systems also need to provide contingencies to fail gracefully—that is, preserve as much of the system’s capabilities and data as possible to facilitate recovery. For example, real-time systems are used for space flights airport traffic control, fly-by-wire aircraft, critical industrial processes, certain medical equipment, and telephone switching, to name a few.

There are two types of real-time systems depending on the consequences of missing the deadline:

* **Hard real-time systems** risk total system failure if the predicted time deadline is missed.
* **Soft real-time systems** suffer performance degradation, but not total system failure, as a consequence of a missed deadline.

Although it’s theoretically possible to convert a general-purpose operating system into a real-time system by merely establishing a deadline, the unpredictability of these systems can’t provide the guaranteed response times that real-time performance requires (Dougherty, 1995). Therefore, most embedded systems and real-time environments require operating systems that are specially designed to meet real-time needs.

1. **Hybrid systems** are a combination of batch and interactive. They appear to be interactive because individual users can access the system and get fast responses, but such a system actually accepts and runs batch programs in the background when the interactive load is light. A hybrid system takes advantage of the free time between high-demand usage of the system and low-demand times. Many large computer systems are hybrids.
2. **Embedded systems** are computers placed inside other products to add features and capabilities. For example, you find embedded computers in household appliances, automobiles, digital music players, elevators, and pacemakers. In the case of automobiles, embedded computers can help with engine performance, braking, and navigation.

For example, several projects are under way to implement “smart roads,” which would alert drivers in cars equipped with embedded computers to choose alternate routes when traffic becomes congested.

**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.

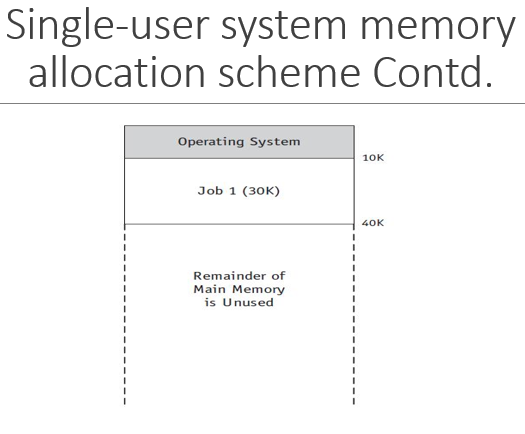
**How the Single-user system memory allocation scheme works**

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.

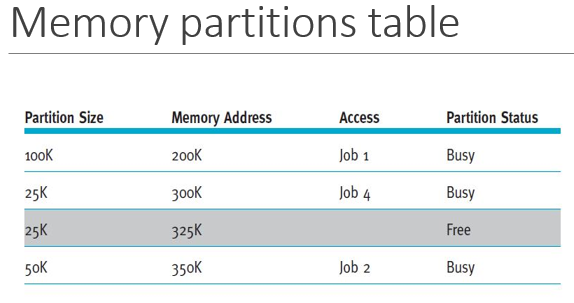
An important factor was introduced with this scheme: protection of the job’s memory space. Once a partition was assigned to a job, the jobs in other memory partitions had to be prevented from invading its boundaries, either accidentally or intentionally.

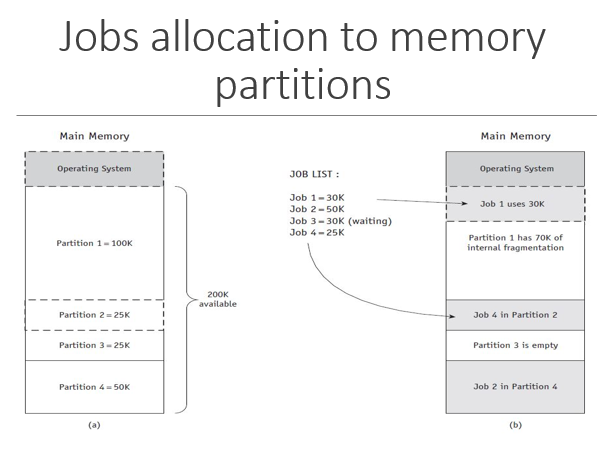
This problem of partition intrusion didn’t exist in single-user contiguous allocation schemes because only one job was present in main memory at any given time—only the portion of main memory that held the operating system had to be protected. However, for the fixed partition allocation schemes, protection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

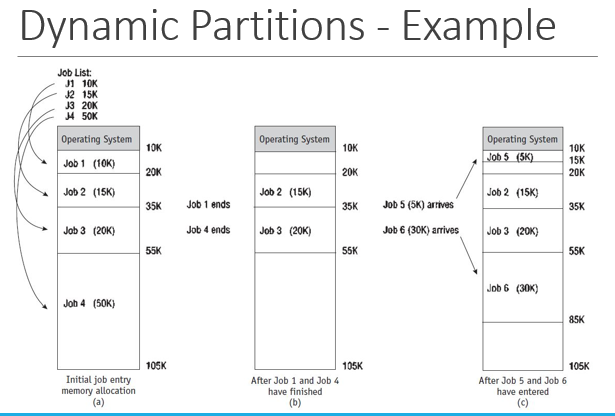
It also leads to ***Internal Fragmentation***; a phenomenon of less-than-complete use of memory space. And it is a major drawback to this memory allocation scheme

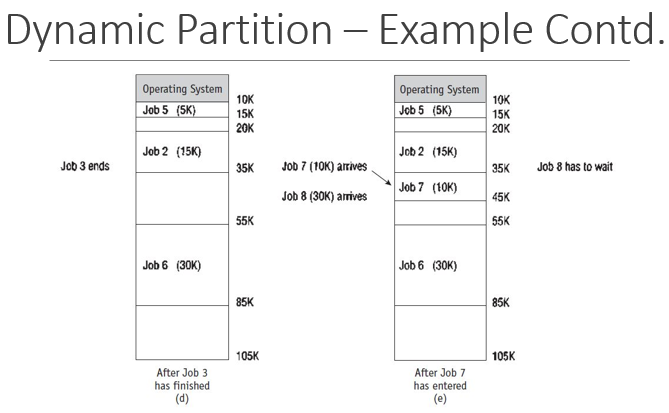
**Dynamic Partitions**

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.

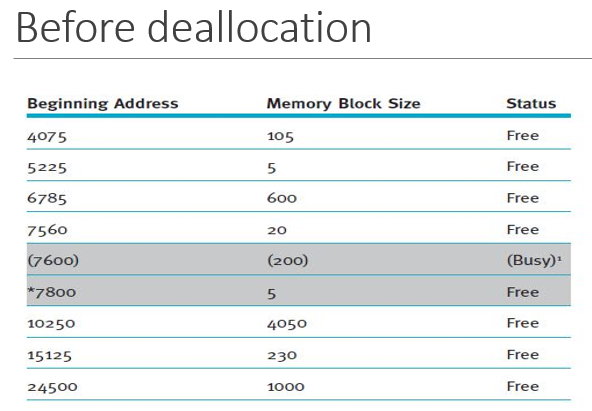
**De-allocation in Dynamic Partition**

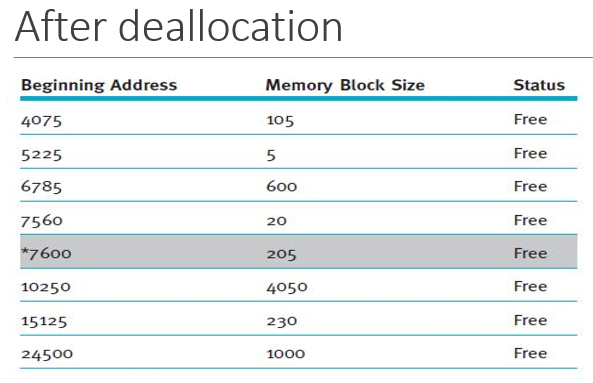
A dynamic partition system uses a more complex algorithm because it tries to combine free areas of memory whenever possible. Therefore, the system must be prepared for three alternative situations:

* 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

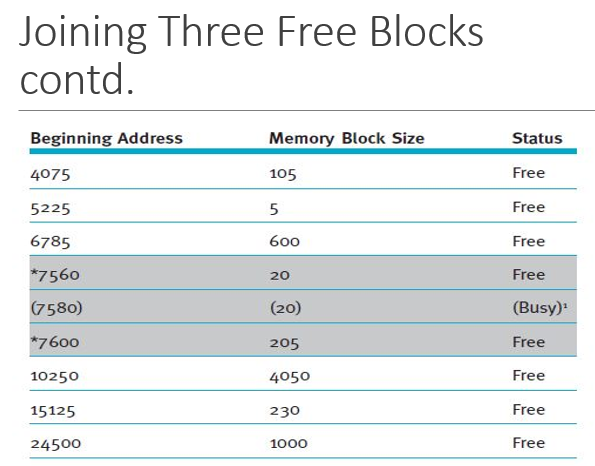
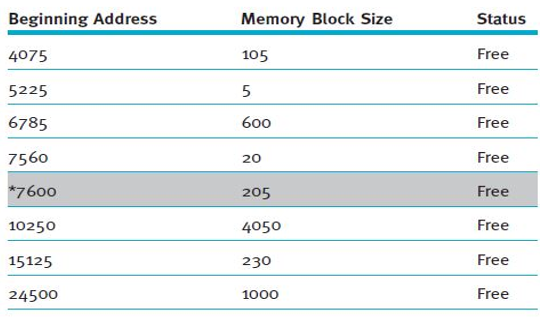
**Joining Two Free Blocks**

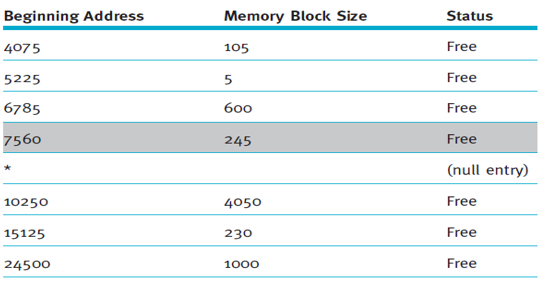
The table below shows how de-allocation occurs in a dynamic memory allocation system when the job to be de-allocated is next to one free memory block before the de-allocation





**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, 7 



**De-allocating an Isolated Block**

Reading assignment

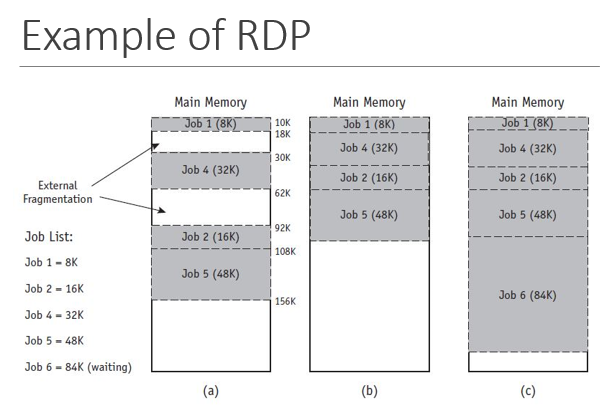
**Re-locatable Dynamic Partition (RDP)**

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.

**Memory management techniques**

Memory management evolve with four new memory allocation schemes.

* Paged Memory Allocation (PMSA)
* Demand Paging Memory Allocation (DPMA)
* Segmented Memory Allocation (SMA)
* Segmented/Demand Paging Memory Allocation (S/DPMA)

Theses memory allocation schemes remove the restriction of storing the programs contiguously, and most of them eliminate the requirement that the entire program reside in memory during its execution.

**Paged Memory Allocation (PMA)**

**Paged memory allocation** is based on the concept of dividing jobs into units of equal size and each unit is called a **page**. Some operating systems choose a page size that is the exact same size as a section of main memory, which is called a **page frame.** Likewise, the sections of a magnetic disk are called **sectors** or **blocks**.

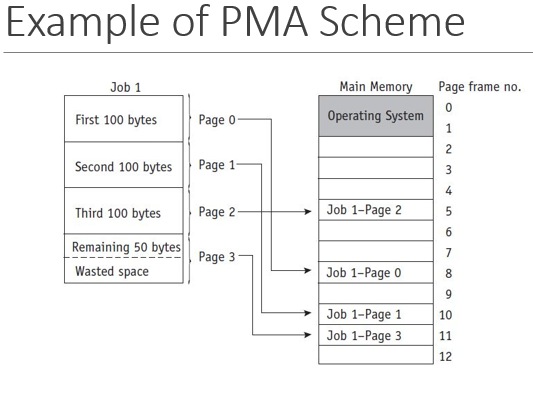
The paged memory allocation scheme works quite efficiently when the pages, sectors, and page frames are all the same size.

**Jobs Ready for Execution**

Before executing a program, a basic Memory Manager prepares it by:

* Determining the number of pages in the program
* Locating enough empty page frames in main memory
* Loading all of the program’s pages into those frames

When the program is initially prepared for loading, its pages are in logical sequence — the first pages contain the first instructions of the program and the last page has the last instructions. These jobs can be loaded in noncontiguous page frames. In fact, each page can be stored in any available page frame anywhere in main memory



**Table of PMA**

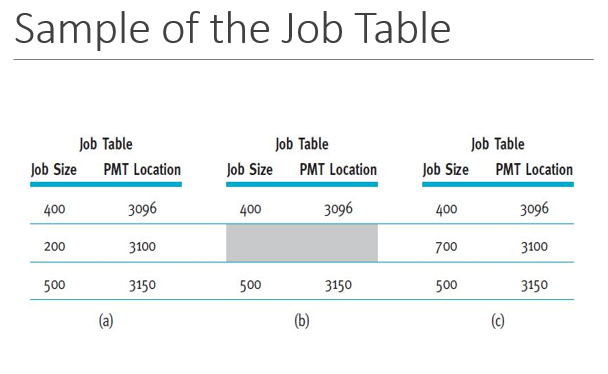
Memory Manager uses some three tables;

1. Job Table,
2. Page Map Table
3. Memory Map Table

To keep track of the pages as they are fit into page frames

The Job Table (JT) contains two values for each active job: the size of the job (shown on the left) in the figure below and the memory location where its Page Map Table is stored (on the right)

The Job Table is a dynamic list that grows as jobs are loaded into the system and shrinks, as shown in (b) in the figure below, as they are later completed.



Each active job has its own Page Map Table (PMT), which contains the vital information for the page—the page number and its corresponding memory address of the page frame.

A simple Memory Map Table (MMT) has one entry for each page frame and shows its location and its free/busy status.

**Advantages of PMAA**

A huge advantage of a paging scheme is that:

1. It allows jobs to be allocated in noncontiguous memory locations.
2. It allows memory to be used more efficiently.

**Disadvantage of PMA**

1. Overhead is increased, and
2. Internal fragmentation is still a problem, although it occurs only in the last page of each job.
3. It still requires that the entire job be stored in memory during its execution

**Demand Paging Memory Allocation**

Demand paging introduced the concept of loading only a part of the program into memory for processing.

It was the first widely used scheme that removed the restriction of having the entire job in memory from the beginning to the end of its processing.

**Concept of DPMA**

The concept of demand paging is that jobs are still divided into equally-sized pages that initially reside in secondary storage.

When the job begins to run, its pages are brought into memory only as they are needed, and if they’re never needed, they’re never loaded.

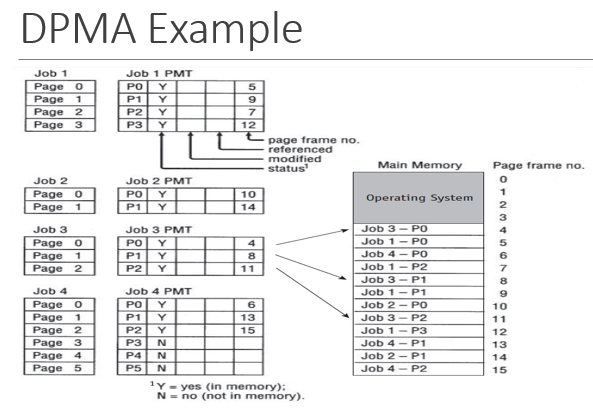
It takes advantage of the fact that programs are written sequentially so that while one section, or module, is processed, other modules may be idle. Not all the pages are accessed at the same time, or even sequentially.

**Scenarios of DPMA**

Instructions to handle errors are processed only when a specific error is detected during execution. For instance, these instructions can indicate that input data was incorrect or that a computation resulted in an invalid answer. If no error occurs, and we hope this is generally the case, these instructions are never processed and never need to be loaded into memory.

Many program modules are mutually exclusive. For example, while the program is being loaded (when the input module is active), then the processing module is inactive because it is generally not performing calculations during the input stage. Similarly, if the processing module is active, then the output module (such as printing) may be idle.

Certain program options are either mutually exclusive or not always accessible. For example, when a program gives the user several menu choices, it allows the user to make only one selection at a time. If the user selects the first option, then the module with those program instructions is the only one that is being used, so that is the only module that needs to be in memory at this time



Demand paging requires that the Page Map Table for each job keep track of each page as it is loaded or removed from main memory. Each PMT tracks the status of the page, whether it has been modified, whether it has been recently referenced, and the page frame number for each page currently in main memory

**Advantages of DPMA**

The demand paging scheme allows the user to run jobs with less main memory than is required if the operating system is using the paged memory allocation scheme.

Demand paging scheme can give the appearance of vast amounts of physical memory when, in reality, physical memory is significantly less than vast.

**Disadvantages of DPMA**

When there is an excessive amount of page swapping between main memory and secondary storage, the operation becomes inefficient. This phenomenon is called **thrashing**.

Thrashing is similar to the problem that students face when comparing explanations of a complex problem in two different textbooks. The amount of time you spend switching back and forth between the two books could cause you to spend more time figuring out where you left off in each book than you spend actually solving the issue at hand.

**Segmented Memory Allocation (SMA)**

The concept of segmentation is based on the common practice by programmers of structuring their programs in modules—logical groupings of code.

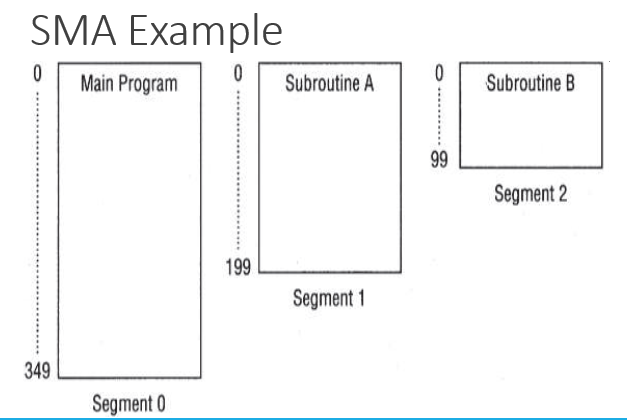
With segmented memory allocation, each job is divided into several segments of ***different sizes***, one for each module that contains pieces that perform related functions

This is fundamentally different from a paging scheme, which divides the job into several pages all of the same size, each of which often contains pieces from more than one program module.

In this memory allocation scheme also, the main memory is no longer divided into page frames, because the size of each segment is different ranging from quite small to large.

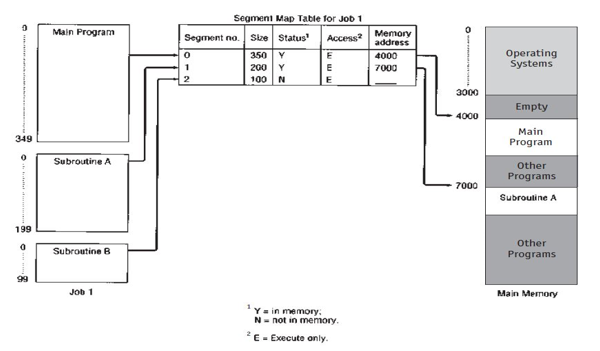
**How the SMA Works**

When a program is compiled or assembled, the segments are set up according to the program’s structural modules. Each segment is numbered and a Segment Map Table (SMT) is generated for each job; it contains the segment numbers, their lengths, access rights, status, and (when each is loaded into memory) its location in memory.



The figure below shows a job, Job 1 that’s composed of a main program and two subroutines (for example, one subroutine calculates the normal pay rate, and a second one calculates the overtime pay or commissions)

It is a single job that is structurally divided into three segments of different sizes.



The Memory Manager needs to keep track of the segments in memory. This is done with three tables, combining aspects of both dynamic partitions and demand paging memory management:

* The Job Table lists every job being processed (one for the whole system).
* The Segment Map Table lists details about each segment (one for each job).
* The Memory Map Table monitors the allocation of main memory (one for the whole system).

**Disadvantage of SMA**

The disadvantage of any allocation scheme in which memory is partitioned dynamically is the return of external fragmentation. Therefore, re-compaction of available memory is necessary from time to time.

**Segmented/Demand Paged Memory Allocation**

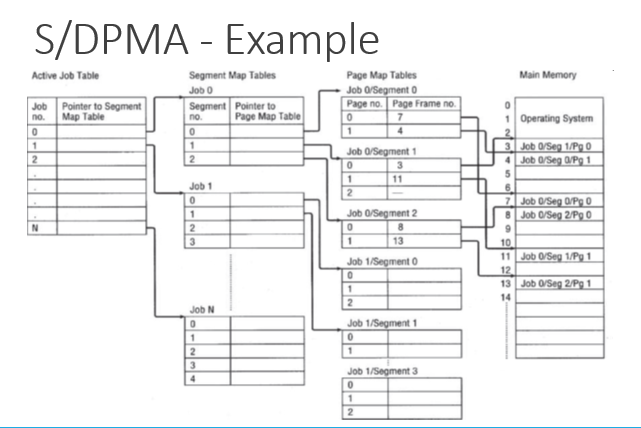
The **segmented/demand paged memory allocation** scheme evolved from a combination of **segmentation** and **demand paging**, and it offers the logical benefits of segmentation, as well as the physical benefits of paging.

This allocation scheme doesn’t keep each segment as a single contiguous unit, but subdivides it into pages of equal size that are smaller than most segments and more easily manipulated than whole segments. Therefore, many of the problems of segmentation (compaction, external fragmentation, and secondary storage handling) are removed because the pages are of fixed length.

**S/DPMA Tables**

This scheme, requires four types of tables as in the figure below:

* The Job Table lists every job in process (there’s one JT for the whole system).
* The Segment Map Table lists details about each segment (one SMT for each job).
* The Page Map Table lists details about every page (one PMT for each segment).
* The Memory Map Table monitors the allocation of the page frames in main memory (there’s one for the whole system).



**Disadvantages of S/DPMA**

The major disadvantages of this memory allocation scheme are twofold:

1. The overhead that is required to manage the tables (Segment Map Tables and the Page Map Tables), and
2. The time required to reference them.