Operating Systems (OS) Concepts - IT 308: Lecture 21
Memory Management

**Batch:** B.Tech III year

**Instructor:** Rahul Muthu

DA-IICT
In a multiprogramming environment, several processes must be run concurrently in order to maximise resource utilisation.
What is memory management

- In a multiprogramming environment, several processes must be run concurrently in order to maximise resource utilisation.
- Thus all (or many) of these processes must be present in main memory at the same time.
What is memory management

- In a multiprogramming environment, several processes must be run concurrently in order to maximise resource utilisation.
- Thus all (or many) of these processes must be present in main memory at the same time.
- Since memory is usually limited to less than the total requirement of all processes present in the job pool, processes need to be selected from secondary storage to bring into main memory.
In a multiprogramming environment, several processes must be run concurrently in order to maximise resource utilisation.

Thus all (or many) of these processes must be present in main memory at the same time.

Since memory is usually limited to less than the total requirement of all processes present in the job pool, processes need to be selected from secondary storage to bring into main memory.

Processes also sometimes need to be moved out of main memory to accommodate another process, and moved in again.
memory manager’s perspective

- Every instruction executed by a computer system involves a memory access.
Every instruction executed by a computer system involves a memory access.

In the execution cycle, an instruction is fetched from the location indicated by the program counter. The instruction is then decoded and then executed.
Every instruction executed by a computer system involves a memory access.

In the execution cycle, an instruction is fetched from the location indicated by the program counter. The instruction is then decoded and then executed.

The execution phase often involves manipulation of one or more variables stored in memory. Thus, those locations may be accessed as well.
Every instruction executed by a computer system involves a memory access.

In the execution cycle, an instruction is fetched from the location indicated by the program counter. The instruction is then decoded and then executed.

The execution phase often involves manipulation of one or more variables stored in memory. Thus, those locations may be accessed as well.

This means that these accesses to memory may be for instructions or data.
Every instruction executed by a computer system involves a memory access.

In the execution cycle, an instruction is fetched from the location indicated by the program counter. The instruction is then decoded and then executed.

The execution phase often involves manipulation of one or more variables stored in memory. Thus, those locations may be accessed as well.

This means that these accesses to memory may be for instructions or data.

The memory manager is designed to handle these two types of memory content identically not distinguishing between whether the access is to instructions or data.
A memory manager needs to control the regions of memory accessed by any particular process, in order to ensure one process does not modify or access the region of another process.
A memory manager needs to control the regions of memory accessed by any particular process, in order to ensure one process does not modify or access the region of another process.

Provision must also be made to allow cooperating processes to access common areas.
A memory manager needs to control the regions of memory accessed by any particular process, in order to ensure one process does not modify or access the region of another process.

Provision must also be made to allow cooperating processes to access common areas.

Facilities for both these requirements need to be in place.
There are many situations where it is not possible to accommodate an entire program in memory at the same time.
There are many situations where it is not possible to accommodate an entire program in memory at the same time.

A similar scenario is when the data to be operated on by a program may not be loadable into memory at the same time.
There are many situations where it is not possible to accommodate an entire program in memory at the same time. A similar scenario is when the data to be operated on by a program may not be loadable into memory at the same time. Both these constitute the two facets of a process’s memory requirement.
There are many situations where it is not possible to accommodate an entire program in memory at the same time.

A similar scenario is when the data to be operated on by a program may not be loadable into memory at the same time.

Both these constitute the two facets of a process’s memory requirement.

It is necessary in such situations to be able to run the program by storing only some fraction of it in memory at any time and bringing in the relevant portions when the need arises.
There are many situations where it is not possible to accommodate an entire program in memory at the same time.

A similar scenario is when the data to be operated on by a program may not be loadable into memory at the same time.

Both these constitute the two facets of a process’s memory requirement.

It is necessary in such situations to be able to run the program by storing only some fraction of it in memory at any time and bringing in the relevant portions when the need arises.

It is often possible to create a process corresponding to a program, with different amounts of memory requirement.
There are many situations where it is not possible to accommodate an entire program in memory at the same time. A similar scenario is when the data to be operated on by a program may not be loadable into memory at the same time. Both these constitute the two facets of a process’s memory requirement. It is necessary in such situations to be able to run the program by storing only some fraction of it in memory at any time and bringing in the relevant portions when the need arises. It is often possible to create a process corresponding to a program, with different amounts of memory requirement. There is a bare-minimum, less than which cannot accommodate the particular program.
There are many situations where it is not possible to accommodate an entire program in memory at the same time. A similar scenario is when the data to be operated on by a program may not be loadable into memory at the same time. Both these constitute the two facets of a process’s memory requirement.

It is necessary in such situations to be able to run the program by storing only some fraction of it in memory at any time and bringing in the relevant portions when the need arises.

It is often possible to create a process corresponding to a program, with different amounts of memory requirement. There is a bare-minimum, less than which cannot accommodate the particular program.

If more memory is available, then the performance is better and the excess should be used to improve effectiveness.
The portion of memory allocated to a process is called its *address space*. 
The portion of memory allocated to a process is called its *address space*.

*Contiguous allocation* is an allocation of a continuous sequence of physical address for the entire requirement of a process.
The portion of memory allocated to a process is called its *address space*.

*Contiguous allocation* is an allocation of a continuous sequence of physical address for the entire requirement of a process.

*Noncontiguous allocation* distributes the address space into separate parts scattered through the memory.
One approach to handling the integrity of address spaces is to partition the memory into pieces and load a process into exactly one of the parts.
One approach to handling the integrity of address spaces is to partition the memory into pieces and load a process into exactly one of the parts.

The number of processes which may be present in the main memory at any stage is restricted to be at most the number of such parts. This is therefore a limit on the degree of multiprogramming.
One approach to handling the integrity of address spaces is to partition the memory into pieces and load a process into exactly one of the parts.

The number of processes which may be present in the main memory at any stage is restricted to be at most the number of such parts. This is therefore a limit on the degree of multiprogramming.

Memory partition may be *static* or *dynamic*.
Partitioning

- One approach to handling the integrity of address spaces is to partition the memory into pieces and load a process into exactly one of the parts.
- The number of processes which may be present in the main memory at any stage is restricted to be at most the number of such parts. This is therefore a limit on the degree of multiprogramming.
- Memory partition may be *static* or *dynamic*.
- Static partitioning is done at the time of system generation and the sizes of each part may not be altered without reinstalling the entire computer system.
One approach to handling the integrity of address spaces is to partition the memory into pieces and load a process into exactly one of the parts.

The number of processes which may be present in the main memory at any stage is restricted to be at most the number of such parts. This is therefore a limit on the degree of multiprogramming.

Memory partition may be *static* or *dynamic*.

Static partitioning is done at the time of system generation and the sizes of each part may not be altered without reinstalling the entire computer system.

Dynamic partitioning allocates any requesting process precisely the amount of memory it demands at the time of requirement.
One approach to handling the integrity of address spaces is to partition the memory into pieces and load a process into exactly one of the parts. The number of processes which may be present in the main memory at any stage is restricted to be at most the number of such parts. This is therefore a limit on the degree of multiprogramming.

Memory partition may be *static* or *dynamic*.

Static partitioning is done at the time of system generation and the sizes of each part may not be altered without reinstalling the entire computer system.

Dynamic partitioning allocates any requesting process precisely the amount of memory it demands at the time of requirement.

Clearly the sizes of the parts are fixed in the former case, but may alter in the latter.
If we use static partitioning, then, since the sizes of most programs are unlikely to match any part’s size precisely, an allocation of memory normally results in wastage of the excess in the corresponding part.
If we use static partitioning, then, since the sizes of most programs are unlikely to match any part’s size precisely, an allocation of memory normally results in wastage of the excess in the corresponding part.

This is called *internal fragmentation*.
If we use static partitioning, then, since the sizes of most programs are unlikely to match any part’s size precisely, an allocation of memory normally results in wastage of the excess in the corresponding part.

This is called *internal fragmentation*.

In the case of dynamic partitioning, the allocation made is precise. However, in a long sequence of allocations and releases, the whole contiguous memory may have a lot of holes of free space.
If we use static partitioning, then, since the sizes of most programs are unlikely to match any part’s size precisely, an allocation of memory normally results in wastage of the excess in the corresponding part.

This is called *internal fragmentation*

In the case of dynamic partitioning, the allocation made is precise. However, in a long sequence of allocations and releases, the whole contiguous memory may have a lot of holes of free space.

In a contiguous allocation scheme, this means that programs bigger than the sizes of these holes cannot be accommodated there.
Fragmentation

- If we use static partitioning, then, since the sizes of most programs are unlikely to match any part’s size precisely, an allocation of memory normally results in wastage of the excess in the corresponding part.

  This is called **internal fragmentation**

- In the case of dynamic partitioning, the allocation made is precise. However, in a long sequence of allocations and releases, the whole contiguous memory may have a lot of holes of free space.

  In a contiguous allocation scheme, this means that programs bigger than the sizes of these holes cannot be accommodated there.

  This problem, also a form of memory wastage is called **external fragmentation**. Even if the total memory available is sufficient it may not be possible to load a process in this case.
Physical address and logical address

- A program is often written and compiled and stored as an executable file on disc before it is loaded and executed.
Physical address and logical address

- A program is often written and compiled and stored as an executable file on disc before it is loaded and executed.
- All instructions as well as data access are made relative to earlier instructions or a particular point in a data file, or variables manipulated by the program.
Physical address and logical address

- A program is often written and compiled and stored as an executable file on disc before it is loaded and executed.
- All instructions as well as data access are made relative to earlier instructions or a particular point in a data file, or variables manipulated by the program.
- These are called *logical addresses* used by a program.
A program is often written and compiled and stored as an executable file on disc before it is loaded and executed.

All instructions as well as data access are made relative to earlier instructions or a particular point in a data file, or variables manipulated by the program.

These are called *logical addresses* used by a program.

However, when a process is actually executing in memory, the exact point where each instruction or piece of data resides depends on the memory management scheme in place, as well as the configuration of the memory at the time the process was last loaded.
A program is often written and compiled and stored as an executable file on disc before it is loaded and executed.

All instructions as well as data access are made relative to earlier instructions or a particular point in a data file, or variables manipulated by the program.

These are called *logical addresses* used by a program.

However, when a process is actually executing in memory, the exact point where each instruction or piece of data resides depends on the memory management scheme in place, as well as the configuration of the memory at the time the process was last loaded.

The actual locations of any instruction/data is called the physical address.
A program is often written and compiled and stored as an executable file on disc before it is loaded and executed.

All instructions as well as data access are made relative to earlier instructions or a particular point in a data file, or variables manipulated by the program.

These are called logical addresses used by a program.

However, when a process is actually executing in memory, the exact point where each instruction or piece of data resides depends on the memory management scheme in place, as well as the configuration of the memory at the time the process was last loaded.

The actual locations of any instruction/data is called the physical address.

Thus for any program there is a physical address space and a logical address space.
The process of translating logical addresses into physical addresses when a program is executing, or present in main memory for execution is called *binding*. 
Address binding

- The process of translating logical addresses into physical addresses when a program is executing, or present in main memory for execution is called *binding*.
- The binding must clearly take place before any instruction is executed or data accessed, otherwise it could lead to illegal or incorrect accesses.
The process of translating logical addresses into physical addresses when a program is executing, or present in main memory for execution is called *binding*.

The binding must clearly take place before any instruction is executed or data accessed, otherwise it could lead to illegal or incorrect accesses.

There is still some flexibility as to the time of address binding, since there is a chain of steps between writing a program and its actual execution on the computer.
A user program is first written in a specific programming language.
A user program is first written in a specific programming language.

The program is then compiled using a compiler for the particular language, resulting in an *object module*. 
User program processing

1. A user program is first written in a specific programming language.

2. The program is then compiled using a compiler for the particular language, resulting in an object module.

3. The object module often makes use of several predefined common libraries with functions present in them. Thus at the time of loading the module, several other modules are loaded. Several routines for system calls are also loaded. This is called the linking phase, and here the relationship between the various parts must be made clear.
A user program is first written in a specific programming language.

The program is then compiled using a compiler for the particular language, resulting in an object module.

The object module often makes use of several predefined common libraries with functions present in them. Thus at the time of loading the module, several other modules are loaded. Several routines for system calls are also loaded. This is called the linking phase, and here the relationship between the various parts must be made clear.

Finally the process begins execution.
Various levels of address binding

- *Static* (or compile-time) binding translates the logical addresses into physical addresses at the time the program is compiled. This necessitates prior knowledge as to where exactly the program will reside in main memory.
Various levels of address binding

- **Static** (or compile-time) binding translates the logical addresses into physical addresses at the time the program is compiled. This necessitates prior knowledge as to where exactly the program will reside in main memory.

- **Load binding** translates the logical addresses into physical addresses at the time of loading into main memory. This is necessary if the actual address of loading is known only at the time of loading. The compiler needs to generate *relocatable* code in this case.
Various levels of address binding

- **Static** (or compile-time) binding translates the logical addresses into physical addresses at the time the program is compiled. This necessitates prior knowledge as to where exactly the program will reside in main memory.

- **Load binding** translates the logical addresses into physical addresses at the time of loading into main memory. This is necessary if the actual address of loading is known only at the time of loading. The compiler needs to generate *relocatable* code in this case.

- **Execution binding** is the process of translating logical addresses into physical addresses at execution time. This is necessary in order that the process be movable from one portion of physical memory to another during its life-cycle.
Various levels of address binding

- **Static** (or compile-time) binding translates the logical addresses into physical addresses at the time the program is compiled. This necessitates prior knowledge as to where exactly the program will reside in main memory.

- **Load binding** translates the logical addresses into physical addresses at the time of loading into main memory. This is necessary if the actual address of loading is known only at the time of loading. The compiler needs to generate *relocatable* code in this case.

- **Execution binding** is the process of translating logical addresses into physical addresses at execution time. This is necessary in order that the process be movable from one portion of physical memory to another during its life-cycle.

- If runtime binding is not provided, then memory management becomes more difficult since any process swapped out, has to be brought in to the same portion of the memory in order to execute correctly.
Wasted memory is one measure of the performance of a memory manager. This includes the fraction of unusable memory due to fragmentation (internal or external) and also the tables and other data-structures used for the memory management itself. This element is referred to as table fragmentation.
Wasted memory is one measure of the performance of a memory manager. This includes the fraction of unusable memory due to fragmentation (internal or external) and also the tables and other data-structures used for the memory management itself. This element is referred to as table fragmentation.

Memory wastage also includes loading of duplicate copies of shared routines or data by multiple processes.
Wasted memory is one measure of the performance of a memory manager. This includes the fraction of unusable memory due to fragmentation (internal or external) and also the tables and other data-structures used for the memory management itself. This element is referred to as table fragmentation.

Memory wastage also includes loading of duplicate copies of shared routines or data by multiple processes.

*Time complexity* is the cost associated with the execution of the memory management algorithms and their decision making procedures.
Wasted memory is one measure of the performance of a memory manager. This includes the fraction of unusable memory due to fragmentation (internal or external) and also the tables and other data-structures used for the memory management itself. This element is referred to as table fragmentation.

Memory wastage also includes loading of duplicate copies of shared routines or data by multiple processes.

*Time complexity* is the cost associated with the execution of the memory management algorithms and their decision making procedures.

*Memory-access overhead* is the extra time used, by comparison with the same scenario in the absence of dynamic memory management.
Virtual memory

- This is used when actual physical memory is less than necessary for a program to execute.
Virtual memory

- This is used when actual physical memory is less than necessary for a program to execute.
- Since every portion of a program is not simultaneously in use, execution is possible by moving various parts in and out dynamically.
Virtual memory

- This is used when actual physical memory is less than necessary for a program to execute.
- Since every portion of a program is not simultaneously in use, execution is possible by moving various parts in and out dynamically.
- Some parts of a program may be used throughout, whereas others need to be present at only particular times.
Virtual memory

- This is used when actual physical memory is less than necessary for a program to execute.
- Since every portion of a program is not simultaneously in use, execution is possible by moving various parts in and out dynamically.
- Some parts of a program may be used throughout, whereas others need to be present at only particular times.
- If the memory is large enough to accommodate at any stage the fraction of a program which is needed at that time, then alone is it possible to execute the program on that system.
Virtual memory

- This is used when actual physical memory is less than necessary for a program to execute.
- Since every portion of a program is not simultaneously in use, execution is possible by moving various parts in and out dynamically.
- Some parts of a program may be used throughout, whereas others need to be present at only particular times.
- If the memory is large enough to accommodate at any stage the fraction of a program which is needed at that time, then alone is it possible to execute the program on that system.
- In this situation, some parts of a program are overwritten by other parts (in memory) as the execution evolves. This process is called *overlaying*.
In static partitioning, clearly decisions need to be made on the basis of the likely memory requirements of user programs.
In static partitioning, clearly decisions need to be made on the basis of the likely memory requirements of user programs. The current state of the memory here is stored in a *partition description table*. 
In static partitioning, clearly decisions need to be made on the basis of the likely memory requirements of user programs.

The current state of the memory here is stored in a *partition description table*.

When a request comes in for a program to be loaded, various algorithms can be used to allocate a partition. Each has its own advantages and disadvantages.
In static partitioning, clearly decisions need to be made on the basis of the likely memory requirements of user programs.

The current state of the memory here is stored in a *partition description table*.

When a request comes in for a program to be loaded, various algorithms can be used to allocate a partition. Each has its own advantages and disadvantages.

Possible algorithms are first-fit, best-fit and worst-fit.
In static partitioning, clearly decisions need to be made on the basis of the likely memory requirements of user programs.

The current state of the memory here is stored in a *partition description table*.

When a request comes in for a program to be loaded, various algorithms can be used to allocate a partition. Each has its own advantages and disadvantages.

Possible algorithms are first-fit, best-fit and worst-fit.

If some process is unallocatable and another process is idling then it is moved out and brought back in for later execution. This is called *swapping*. 
The *swapper* needs to select a process to swap out.
The *swapper* needs to select a process to swap out.

It also needs to swap in a process.
The *swapper* needs to select a process to swap out.

It also needs to swap in a process.

It needs to manage swap space. The swapper performs the majority of the functions of the medium-term scheduler.
The *swapper* needs to select a process to swap out.

It also needs to swap in a process.

It needs to manage swap space. The swapper performs the majority of the functions of the medium-term scheduler.

Relocatable binding is necessary for swapping.
The swapper needs to select a process to swap out.

It also needs to swap in a process.

It needs to manage swap space. The swapper performs the majority of the functions of the medium-term scheduler.

Relocatable binding is necessary for swapping.

Static relocation is like static binding (or load binding) during the reload.
The *swapper* needs to select a process to swap out.

It also needs to swap in a process.

It needs to manage swap space. The swapper performs the majority of the functions of the medium-term scheduler.

Relocatable binding is necessary for swapping.

Static relocation is like static binding (or load binding) during the reload.

Dynamic relocation means again all binding is done at runtime.
Protection and Sharing

- Protection needs to be enforced particularly from overwriting the OS parts by user programs.
Protection and Sharing

- Protection needs to be enforced particularly from overwriting the OS parts by user programs.
- One way of doing this is by storing important code like the *resident monitor* in Read-only memory.
Protection and Sharing

- Protection needs to be enforced particularly from overwriting the OS parts by user programs.
- One way of doing this is by storing important code like the resident monitor in Read-only memory.
- Another way is to have a bit mode, which indicates against every word in memory whether it is protected (belongs to monitor/OS) or it is unprotected.
Protection needs to be enforced particularly from overwriting the OS parts by user programs.

One way of doing this is by storing important code like the resident monitor in Read-only memory.

Another way is to have a bit mode, which indicates against every word in memory whether it is protected (belongs to monitor/OS) or it is unprotected.

Sharing of address spaces by multiple user-processes is regulated by the system.