Operating Systems (OS) Concepts - IT 308: Lecture 8
Concurrency- Overview

**Batch:** B.Tech III year

**Instructor:** Rahul Muthu

DA-IICT
Modern computer systems (and several earlier versions) use multiprogramming extensively.
Modern computer systems (and several earlier versions) use multiprogramming extensively.

The resource allocation problem, to competing processes, is dealt with by scheduling schemes, which are composite scheduling algorithms.
Modern computer systems (and several earlier versions) use multiprogramming extensively.

The resource allocation problem, to competing processes, is dealt with by scheduling schemes, which are composite scheduling algorithms.

The unit of execution in a multiprogramming environment is a process, which is a program in execution.
Two processes are said to be *cooperating processes*, if they have access to shared data.
Two processes are said to be *cooperating processes*, if they have access to shared data.

Other pairs of processes are called *independent processes*. 
Concurrency

- Two processes are said to be *cooperating processes*, if they have access to shared data.
- Other pairs of processes are called *independent processes*.
- The area of operating systems theory which studies the action of *cooperating processes* and their consequences, in a concurrent system, is called *concurrency*.
Processes in systems execute concurrently and must be dynamically created by the systems.
Processes in systems execute concurrently and must be dynamically created by the systems.

Processes may be created by a program directly loaded in memory, or be created by an existing process.
Processes in systems execute concurrently and must be dynamically created by the systems.

Processes may be created by a program directly loaded in memory, or be created by an existing process.

The latter is achieved by a system call.
Processes in systems execute concurrently and must be dynamically created by the systems.

Processes may be created by a program directly loaded in memory, or be created by an existing process.

The latter is achieved by a system call.

Each of the created processes can further create processes, leading to a tree structure of the processes.
When a process creates several processes, it is called a *parent process* and the created processes are individually called a *child process*.
When a process creates several processes, it is called a parent process and the created processes are individually called a child process.

All processes require system resources like CPU cycles, Files, memory, I/O devices, during their execution.
When a process creates several processes, it is called a *parent process* and the created processes are individually called a *child process*.

All processes require system resources like CPU cycles, Files, memory, I/O devices, during their execution.

When a process creates other processes, the new processes can either obtain resources directly from the operating system or the parent may partition a subset of its resources among its children.
When a process creates several processes, it is called a *parent process* and the created processes are individually called a *child process*.

All processes require system resources like CPU cycles, Files, memory, I/O devices, during their execution.

When a process creates other processes, the new processes can either obtain resources directly from the operating system or the parent may partition a subset of its resources among its children.

In some cases the parent process *shares* its resources with its children, rather than *partitioning* them.
When a process creates several processes, it is called a *parent process* and the created processes are individually called a *child process*.

All processes require system resources like CPU cycles, Files, memory, I/O devices, during their execution.

When a process creates other processes, the new processes can either obtain resources directly from the operating system or the parent may partition a subset of its resources among its children.

In some cases the parent process *shares* its resources with its children, rather than *partitioning* them.

When a process is created, in addition to the physical and logical resources it needs, it also often requires some initialisation data passed as a parameter.
When a process creates several processes, it is called a *parent process* and the created processes are individually called a *child process*.

All processes require system resources like CPU cycles, Files, memory, I/O devices, during their execution.

When a process creates other processes, the new processes can either obtain resources directly from the operating system or the parent may partition a subset of its resources among its children.

In some cases the parent process *shares* its resources with its children, rather than *partitioning* them.

When a process is created, in addition to the physical and logical resources it needs, it also often requires some initialisation data passed as a parameter.

The process ID of all its children is maintained by any process at the time of creation, until the return of the child processes.
When a process creates other processes in one scenario it can continue executing concurrently with the children.
When a process creates other processes in one scenario it can continue executing concurrently with the children.

It can also suspend itself pending the return of one or more of its children.
The child process maybe a *duplicate* of the parent process.
• The child process maybe a *duplicate* of the parent process.
• The child process has an independent program loaded into it.
A process terminates when it has executed its last instruction.
A process terminates when it has executed its last instruction.

In this case it asks the operating system to delete it and deallocate the resources associated with it, using an *exit* system call.
A process terminates when it has executed its last instruction. In this case it asks the operating system to delete it and deallocate the resources associated with it, using an `exit` system call.

At this stage the process may return its output to the parent process using the `wait` system call.
A process terminates when it has executed its last instruction.

In this case it asks the operating system to delete it and deallocate the resources associated with it, using an *exit* system call.

At this stage the process may return its output to the parent process using the *wait* system call.

A process may also be terminated prematurely by an *abort* system call.
This abort action may only be taken by the process itself due to some error, or by the parent process.
- This abort action may only be taken by the process itself due to some error, or by the parent process.
- This ensures protection against random process termination by unauthorised processes.
- This abort action may only be taken by the process itself due to some error, or by the parent process.
- This ensures protection against random process termination by unauthorised processes.
- The parent may terminate a process because the child process has exceeded the resources allocated to it.
This abort action may only be taken by the process itself due to some error, or by the parent process.

This ensures protection against random process termination by unauthorised processes.

The parent may terminate a process because the child process has exceeded the resources allocated to it.

It may also do so because the child process is no longer performing a useful function.
This abort action may only be taken by the process itself due to some error, or by the parent process. This ensures protection against random process termination by unauthorised processes.

The parent may terminate a process because the child process has exceeded the resources allocated to it.

It may also do so because the child process is no longer performing a useful function.

Finally, in many systems, the termination of a process must be preceded by the termination of all its children. Thus the operating system terminates the children of any terminating process. This is called cascading termination.
Advantages of cooperating processes

- Information can be shared without duplication, if several processes are interested, for example in the contents of a single file.
Advantages of cooperating processes

- Information can be shared without duplication, if several processes are interested, for example in the contents of a single file.
- Computational speedup is achieved by parallelising processes which can then run concurrently, if there is a logical way to breakup the functioning.
Advantages of cooperating processes

- Information can be shared without duplication, if several processes are interested, for example in the contents of a single file.
- Computational speedup is achieved by parallelising processes which can then run concurrently, if there is a logical way to breakup the functioning.
- The separation of a big program into independent logical units, called modularisation, facilitates writing of clear code.
Advantages of cooperating processes

- Information can be shared without duplication, if several processes are interested, for example in the contents of a single file.
- Computational speedup is achieved by parallelising processes which can then run concurrently, if there is a logical way to breakup the functioning.
- The separation of a big program into independent logical units, called *modularisation*, facilitates writing of clear code.
- A single user using multiple programs often needs to use resources of one process in another.
Communication means among cooperating processes is called *Interprocess communication* (IPC).
Communication means among cooperating processes

- Communication between two cooperating processes is called *Interprocess communication* (IPC).
- One way to to achieve this is by a message passing system.
Communication means among cooperating processes

- Communication between two cooperating processes is called *Interprocess communication* (IPC).
- One way to achieve this is by a message passing system.
- Shared memory (data structure) is another way to achieve the same goal.