

Development of Multiplatform Applications
2nd Course

Programming of Services and Processes

**MULTIPROCESS
PROGRAMMING**

UNIT 1: PROCESSES

***Professor:** Empar Carrasquer
ecarrasquer@cipfpbatoi.es*

| | |
|--|-----------|
| OBJECTIVES | 3 |
| 1. INTRODUCTION | 4 |
| 2. BASIC CONCEPTS | 5 |
| 3. PROCESSES | 8 |
| 3.1 THE OPERATING SYSTEM AND PROCESSES | 8 |
| 3.2 STATES OF A PROCESS | 9 |
| 3.3 PROCESS SCHEDULING. | 11 |
| 3.4 PROCESS CREATION. | 16 |
| 3.5 TERMINATING A PROCESS | 17 |
| 3.6 PROCESSES AND THREADS | 18 |
| 3.7 MANAGING PROCESSES IN THE OPERATING SYSTEM | 19 |
| 4. MULTI-PROCESS PROGRAMMING IN JAVA | 20 |
| 4.1 INTRODUCTION | 20 |
| 4.2 CREATING AND DESTROYING A PROCESS | 20 |
| 4.3 COMMUNICATING PROCESSES | 23 |
| 4.4 SYNCHRONIZING | 25 |
| 5. EXERCISES | 26 |

OBJECTIVES

- ✓ Understand the basic concepts related to execution of programs.
- ✓ Know how the operating system manages processes.
- ✓ Create simple multi process applications.

1. INTRODUCTION

In the past, the early computers didn't have operating systems; they executed a single program from beginning to end, and that program had direct access to all the resources of the machine. Running only a single program at a time was an inefficient use of expensive computer resources.

Operating systems evolved to allow more than one program to run at once, running individual programs in processes: isolated, independently executing programs to which the operating system allocates resources such as memory, file handles, and security credentials. Processes could communicate with one another through a variety of communication mechanisms: sockets, signal handlers, shared memory, semaphores, and files.

Allowing multiple programs to execute simultaneously was motivated by:

- **Resource utilization.** Programs sometimes have to wait for external operations such as input or output, and while waiting can do no useful work. It is more efficient to use that wait time to let another program run.
- **Fairness.** Multiple users and programs may have equal claims on the machine's resources. It is preferable to let them share the computer via finer grained time slicing than to let one program run to completion and then start another.
- **Convenience.** It is often easier or more desirable to write several programs that each perform a single task and have them coordinate with each other as necessary than to write a single program that performs all the tasks.

Nearly all widely used programming languages today follow the sequential programming model, where the language specification clearly defines "*what comes next*" after a given action is executed.

The sequential programming model is intuitive and natural, as it models the way the humans usually work: do one thing at a time in sequence but also involves some degree of asynchrony. That is, do another task while waiting for others to finish.

Finding the right balance of **sequentiality** and **asynchrony** is often a characteristic of efficient people and the same is true of programs.

The same concerns (resource utilization, fairness, and convenience) that motivated the development of processes also motivated the development of **threads**.

Threads allow multiple streams of program control flow to coexist within a process. They share process wide resources such as memory and file handles, but each thread has its own program counter, stack, and local variables. Threads also provide a natural decomposition for exploiting hardware parallelism on multiprocessor systems; multiple threads within the same program can be scheduled simultaneously on multiple CPUs.

Threads are sometimes called **lightweight processes**, and most modern operating systems treat threads, not processes, as the basic units of scheduling.

In the absence of explicit coordination, threads **execute simultaneously** and **asynchronously** with respect to one another. Threads share the memory address space of their owning process; all threads within a process have access to the same variables and allocate objects from the same heap. Therefore, without explicit synchronization to coordinate access to shared data, a thread might modify variables that another thread is in the middle of using, with unpredictable results.

2. BASIC CONCEPTS

Operating systems support concurrent programming. However, to fully understand how it works, there are some concepts to learn.

Concurrency

It is the execution of two or more **independent**, interacting programs over the same period of time; their execution can be interleaved or even simultaneous. Concurrency is used in many kinds of systems, both small and large.

Example1: Home computer.

In separate windows, a user runs a web browser, a text editor, and a music player all at once. The operating system interacts with each of them.

Almost unnoticed, on the background, the clock and virus scanner are also running. The operating system waits for the user to ask more programs to start, while also handling underlying tasks such as resolving what information from the Internet goes to which program.

Example 2: Online reservation system

Using a web-based hotel reservation system, Anna and Sue want to book a room. Each has a separate computer, and thus a separate web browser. Their two web browsers plus the hotel's web server together comprise the concurrent program, but the term **concurrent system** might be more appropriate and it will view them as a single, distributed entity to be modelled.

A concurrent system may be implemented via:

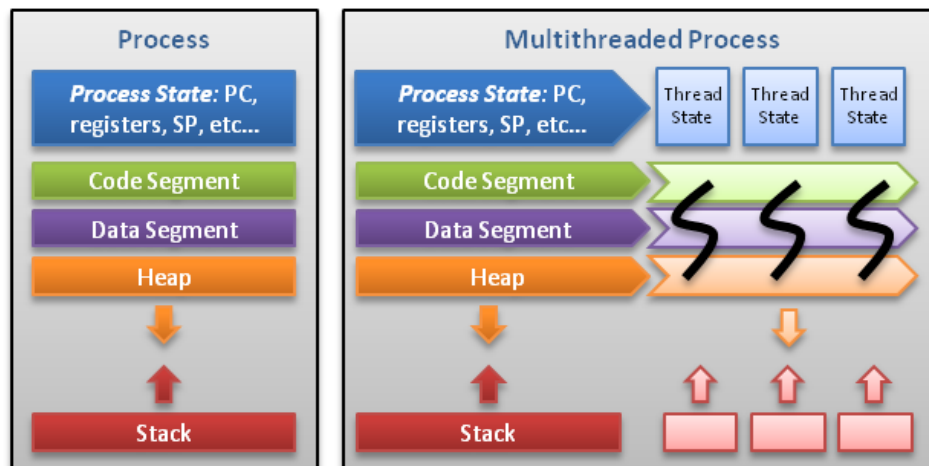
- **Processes** and/or
- **Threads**.

A **process** is a running **program**. The operating system allocates all resources that are needed for the process. A **program**

Example:

We might start two instances of Microsoft Word each modifying a different file. Each instance of Word runs in an independent process with its own memory space.

A **thread** is a dispatching unit within a process. That means that a process can have a number of threads running within the scope of that particular process.



Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

Main difference between threads and processes can be defined as follows:

While a thread has an access to the memory address space and resources of its process, a process cannot access variables allocated for other processes.

Conclusions:

- Communication between threads created within a single process is simple because the threads share all the variables. Thus, a value produced by one thread is immediately available for all the other threads.
- Threads take less time to start, stop or swap than processes, because they use the address space already allocated for the current process.

Different operating systems treat processes and threads differently. For example,

- MS-DOS support a single user process and a single thread,
- UNIX supports multiple user processes with only one thread per process,
- Solaris OS, Linux and Windows XP/7/8, Mac OS X support multiple processes and multiple threads.

So, if an operating system supports at least multiple processes, we can say that the **operating system supports concurrent programming**.

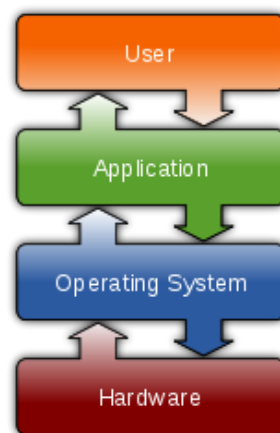
Executable file

It is a computer file that contains instructions that a computer's operating system can understand and follow. Executing this file runs the program as a process.

Operating system:

It is a collection of software that manages computer hardware resources and provides common services for computer programs. Objectives:

- Executing user programs: create and manage processes from executable files and manage their execution.
- Make the computer user-friendly: OS is the interface between user and computer hardware.
- Optimize all the computer resources to users and applications.



Daemon (service)

It is a computer program that runs as a background process, rather than being under the direct control of an interactive user.

Systems often start daemons at boot time and serve the function of responding to network requests, hardware activity, or other programs by performing some task. Daemons can also configure hardware (like *udev* on some GNU/Linux systems)

Parallel computing

It is a form of computation in which many calculations are carried out (executed) simultaneously in a multi-core multiprocessor system. A **parallel programming model** is a concept that enables the expression of *parallel programs*, which can be compiled and executed.

Distributed computing

Studies the *distributed systems*: Software components located on networked computers communicate and coordinate to achieve a common goal. A problem is divided into many tasks and many computers working in parallel solve it.

A computer program that runs in a distributed system is a **distributed program** and **distributed programming** is the process of writing such programs.

Examples of distributed systems: SOA-based systems, massively multiplayer online games, peer-to-peer applications, etc.

Activity: Search for information in the Internet about SOA.

What are their benefits? Is there something to criticize to this architecture?

3. PROCESSES

3.1 The operating system and processes

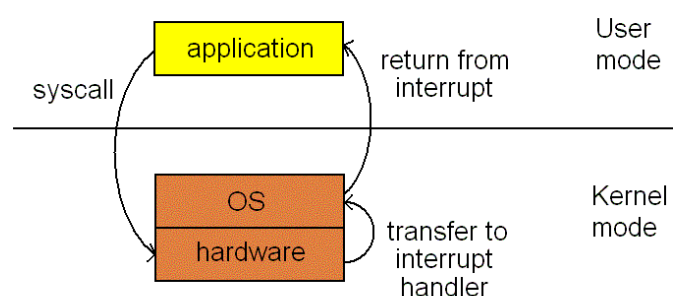
The operating system as a whole includes the **kernel**, and may include other related programs that provide services for applications like:

- Utility programs
- Command interpreters
- Programming libraries

The operating system **kernel** is the part of the operating system that responds to **system calls**, **hardware interruptions** and **program exceptions**.

There are two processor (CPU) modes of operation:

- Kernel (*privileged* or *supervisor* mode)
- User mode



Kernel code runs in a privileged execution mode, while the rest of the operating system does not. It is a program with code and data like any other program that resides in its own address space, separate from the address space of processes that are running on the system.

Kernel uses *privilege* to:

- Control hardware
- Protect and isolate itself from processes

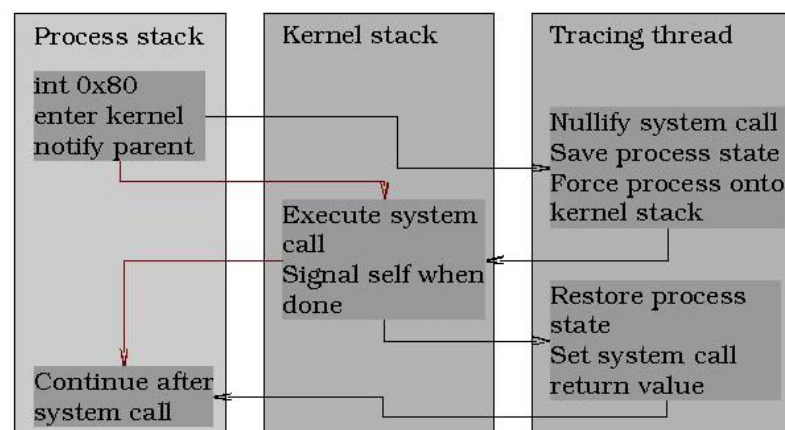
Privileges vary from platform to platform, but may include:

- Ability to execute special instructions (like halt)
- Ability to manipulate processor state (like execution mode)
- Ability to access virtual addresses that can't be accessed otherwise

System calls are the interface between processes and the kernel. A process uses system calls to request operating system services.

This is what happens in a system call:

Virtualization of a system call



3.2 States of a process

The operating system is responsible for executing and managing processes.

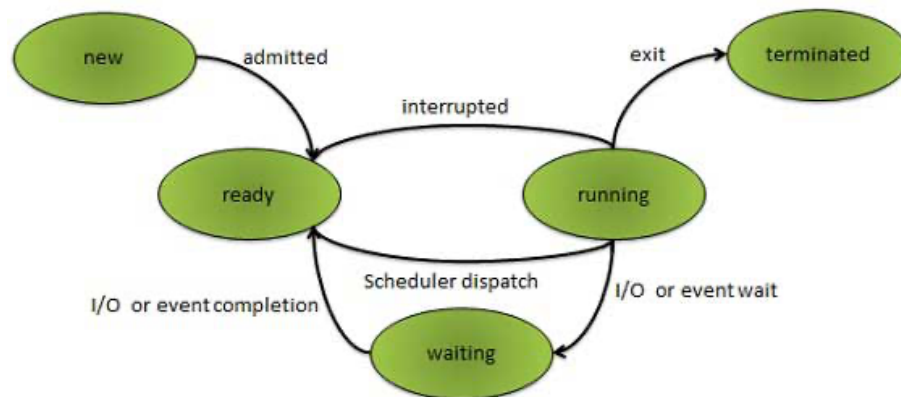
Components of a process are:

- Instructions to be executed (code or program)
- Data to be used while executing.
- Resources needed.
- Status of the process execution.

Activity: Find information about how MS-DOS operating system manages processes.

As a process runs, the process will go through several **states**. The change of state will also occur through the intervention of the operating system.

Process can have one of the following five states at a time: new, ready, running, waiting and terminated.



New: the process is being created.

Ready: The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run.

Running: Process instructions are being executed (i.e. The process that is currently being executed).

Waiting: The process is waiting for some event to occur (such as the completion of an I/O operation).

Terminated: The process has finished execution.

Activity: Look for information about conditions for the transition of process state and complete the following table:

| Transition | Description of conditions |
|--|---------------------------|
| (Admitted) New → Ready | |
| (Scheduler dispatched) Ready → Running | |
| (Interrupted, time expired) Running → Ready | |
| (Blocked) Running → Waiting | |
| (Awakes) Waiting → Ready | |
| (Exits) Running → Terminated | |

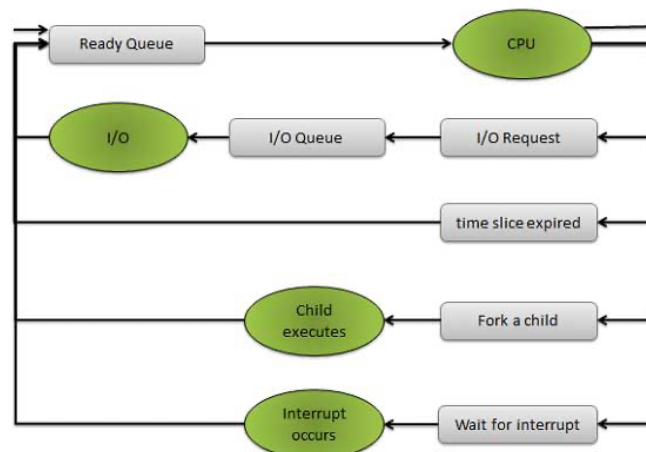
3.3 Process Scheduling.

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular **strategy**.

Process scheduling is an essential part of a *Multiprogramming* operating system that allows more than one process to be loaded into the executable memory at a time and loaded process shares the CPU using *time multiplexing*.

Activity: Search the Internet for the “time multiplexing” concept.

Scheduling queues refers to queues of processes or devices. Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.



A newly arrived process is put in the ready queue. Processes wait in ready queue for allocating the CPU. Once the CPU is assigned to a process, then that process will execute. While executing the process, any one of the following events can occur.

- The process could issue an I/O request and then it would be placed in an I/O queue.
- The process could create new sub process and will wait for its termination.
- The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

Schedulers are special system software that handles process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run.

There are three types of schedulers:

- Long Term Scheduler
- Short Term Scheduler
- Medium Term Scheduler

Long-Term Scheduler

It is also called **job scheduler**. Long-term scheduler determines which programs are admitted to the system for processing.

Job scheduler selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long-term scheduler. When process changes the state from new to ready, then there is use of long-term scheduler.

Short-Term Scheduler

It is also called **CPU scheduler or dispatcher**. It decides which process executes next. Main objective is increasing system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects one process among the processes that are ready to execute and allocates CPU to one of them. Short-term scheduler is faster than long-term scheduler.

Medium-Term Scheduler

Medium term scheduling is part of the **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the *swapped out-processes*.

Running process may become suspended if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the secondary storage. This process is called swapping, and the process is swapped out or rolled out. Swapping may be necessary to improve the process mix.

Context Switch

The *context* switch is the mechanism to **store** and **restore** the **state** or **context** of a CPU *in Process Control Block* so that a process execution can be resumed from the same point at a later time.

Using this technique a **context switcher enables multiple processes to share a single CPU**. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the context switcher saves the content of all processor registers for the process being removed from the CPU, in its process descriptor. The **context of a process** is represented in the process **control block of a process**.

Context switch time is pure overhead. Context switching can significantly affect performance as modern computers have a lot of general and status registers to be saved. Content switching times are highly dependent on hardware support.

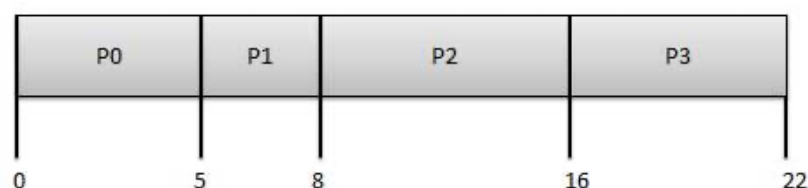
When the process is switched, the following information is stored.

- Program Counter
- Scheduling Information
- Base and limit register value
- Currently used register
- Changed State
- I/O State
- Accounting

Scheduling algorithms:

- First Come First Serves (FCFS) Scheduling
- Shortest-Job-First (SJF) Scheduling
- Priority Scheduling
- Round Robin (RR) Scheduling
- Multilevel Queue Scheduling
- **First Come First Serve (FCFS)**
 - Jobs are executed on first come, first serve basis.
 - Easy to understand and implement.
 - Poor in performance as average waiting time is high.

| Process | Arrival Time | Execute Time | Service Time |
|---------|--------------|--------------|--------------|
| P0 | 0 | 5 | 0 |
| P1 | 1 | 3 | 5 |
| P2 | 2 | 8 | 8 |
| P3 | 3 | 6 | 16 |



Wait time of each process is following:

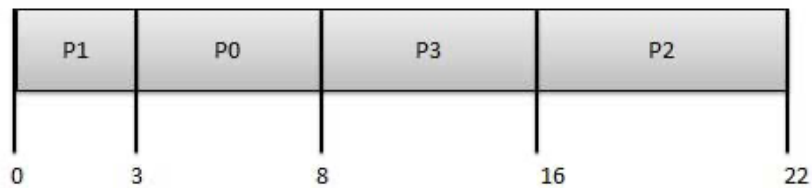
| Process | Wait Time: Service Time - Arrival Time |
|---------|--|
| P0 | 0 - 0 = 0 |
| P1 | 5 - 1 = 4 |
| P2 | 8 - 2 = 6 |
| P3 | 16 - 3 = 13 |

$$\text{Average Wait Time: } (0+4+6+13) / 4 = 5.55$$

- **Shortest Job First (SJF)**

- Best approach to minimize waiting time.
- Impossible to implement
- Processor should know in advance how much time process would take.

| Process | Arrival Time | Execute Time | Service Time |
|---------|--------------|--------------|--------------|
| P0 | 0 | 5 | 0 |
| P1 | 1 | 3 | 3 |
| P2 | 2 | 8 | 8 |
| P3 | 3 | 6 | 16 |



Time of each process is the following:

| Process | Wait Time: Service Time - Arrival Time |
|---------|--|
| P0 | 3 - 0 = 3 |
| P1 | 0 - 0 = 0 |
| P2 | 16 - 2 = 14 |
| P3 | 8 - 3 = 5 |

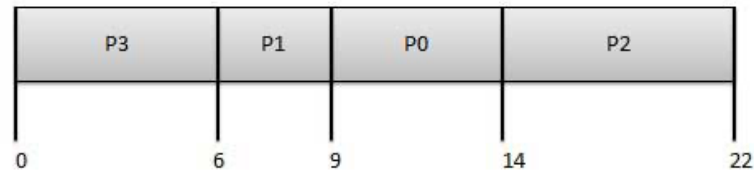
$$\text{Average Wait Time: } (3+0+14+5) / 4 = 5.50$$

- **Priority Based Scheduling**

- Each process is assigned a priority.
- Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first serve basis.

Priority can be decided based on memory requirements, time requirements or any other resource requirement.

| Process | Arrival Time | Execute Time | Priority | Service Time |
|---------|--------------|--------------|----------|--------------|
| P0 | 0 | 5 | 1 | 0 |
| P1 | 1 | 3 | 2 | 3 |
| P2 | 2 | 8 | 1 | 8 |
| P3 | 3 | 6 | 3 | 16 |



Wait time of each process is the following:

| Process | Wait Time: Service Time - Arrival Time |
|---------|--|
| P0 | $0 - 0 = 0$ |
| P1 | $3 - 1 = 2$ |
| P2 | $8 - 2 = 6$ |
| P3 | $16 - 3 = 13$ |

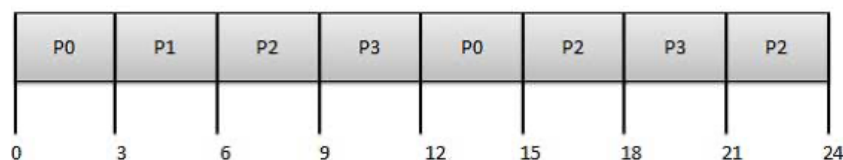
$$\text{Average Wait Time: } (0+2+6+13) / 4 = 5.25$$

- Round Robin Scheduling**

- Each process is provided a fix time to execute called **quantum**.
- Once a process is executed for given time period, process is **preempted** (stopped or paused) and other process executes for given time period. Context switching is used to save states of **preempted** processes.

| Process | Arrival Time | Execute Time |
|---------|--------------|--------------|
| P0 | 0 | 5 |
| P1 | 1 | 3 |
| P2 | 2 | 8 |
| P3 | 3 | 6 |

Quantum = 3

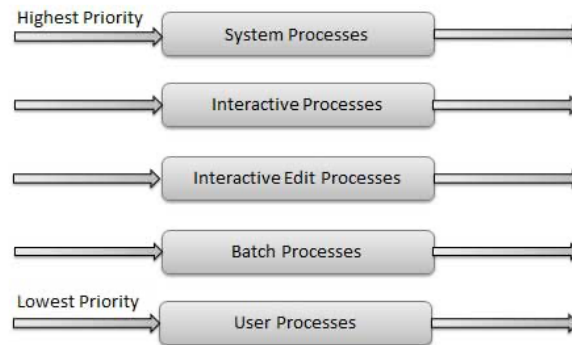


Wait time of each process is the following:

| Process | Wait Time: Service Time - Arrival Time |
|---------|--|
| P0 | $(0-0) + (12-3) = 9$ |
| P1 | $(3-1) = 2$ |
| P2 | $(6-2) + (15-9) = 10$ |
| P3 | $(9-3) + (18-12) = 12$ |

$$\text{Average Wait Time: } (9+2+10+12) / 4 = 8.25$$

- **Multi Queue Scheduling**
 - Multiple queues are maintained for processes.
 - Each queue can have its own scheduling algorithms.
 - Priorities are assigned to each queue.



3.4 Process creation.

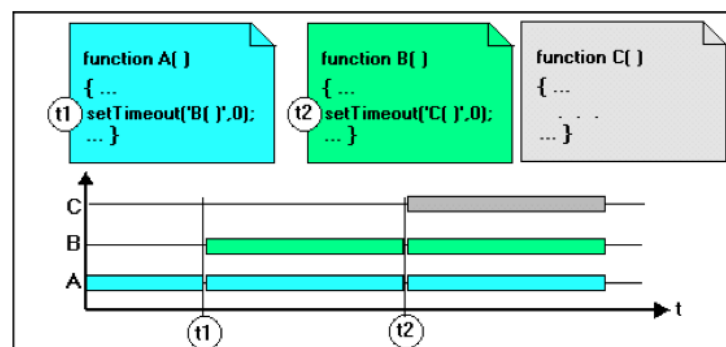
There are four main events that cause process creating:

- System **boot**
- A **system call** from a process to create a **new process**. (→ In Unix, the system call **fork** creates a new process (child) identical to the parent process)
- **User request** to create a process.
- Starting a **batch job**.

Most concurrent languages offer some variant of the following **process creation mechanisms**:

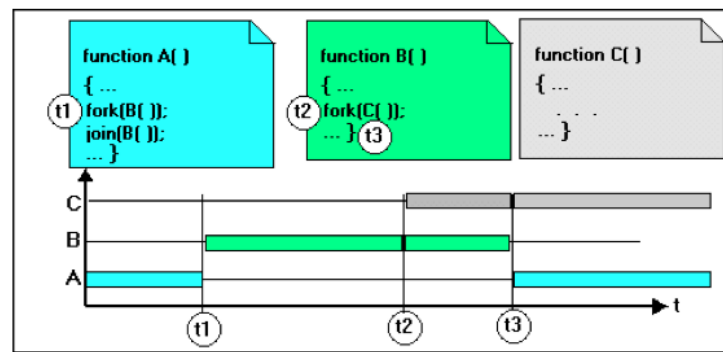
- Co-routines
- Fork and Join
- Cobegin/coend

Co-routines are only pseudo-concurrent and require explicit transfers of control:



Co-routines can be used to implement most high-level concurrent mechanisms.

Fork can be used to create any number of processes:

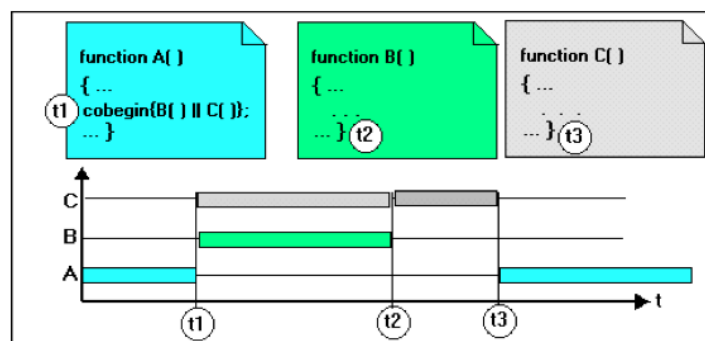


Join waits for another process to terminate.

→ Fork and join are **unstructured**, so require *care and discipline*.

Cobegin/coend blocks are better structured, but they can only create a fixed number of processes.

The caller continues when all of the coblocks have terminated.



3.5 Terminating a process

The live-cycle of a process is simple: creating, executing and terminating. A process may terminate for the following reasons:

- **Normal exit** (voluntary): A process terminates when it finishes executing its final statement and asks the OS to delete it by using the `exit()` system call. At that point, the process may return a status value (typically an integer) to its parent process. All the resources of the process, including physical and virtual memory, opened files and I/O buffers are deallocated by the OS.
- **Abnormal termination**: programming errors, run time errors, I/O, user intervention.
 - **Error exit** (voluntary): An error caused by the process, often due to a program bug (executing an illegal instruction, referencing non-existent memory, or dividing by zero). In some systems (e.g. UNIX), a process can tell the OS that it wishes to handle certain errors itself, in which case the process is signalled (interrupted) instead of terminated when one of the errors occurs.
 - **Fatal error** (involuntary): i.e. no such file exists.
 - **Killed by another process** (involuntary): A process can cause the termination of another process via an appropriate system call.

Answer the question: In Linux, what is the operating system command to terminate a process that is not responding?

3.6 Processes and threads

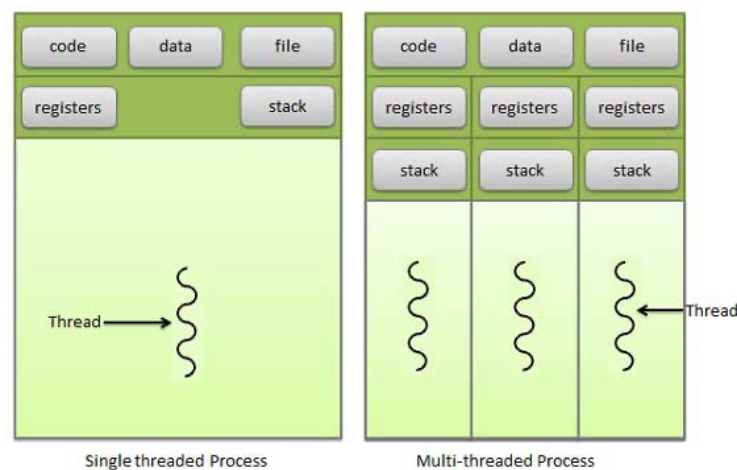
A **thread** is a flow of execution through the process code, with its own program counter, system registers and stack.

A thread is also called a **lightweight process** and represents a separate flow of control.

Each thread belongs to exactly one process and no thread can exist outside a process. The implementation of threads differs from one operating system to another.

Threads have been successfully used in implementing network servers and web servers. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

The following figure shows the working of the single and multithreaded processes.



Differences between process and thread:

| Process | Thread |
|--|--|
| Process is heavy weight or resource intensive. | Thread is lightweight, takes less resources than a process. |
| Process switching needs interaction with operating system as it has separate address space. | Switching among threads in the same process is much faster because the OS only switches registers, not memory mapping. |
| In multiple processing environments, each process executes the same code but it has its own memory and file resources. | All threads can share same code, data and set of opened files. |
| If one process is blocked, then no other process can execute until the first process is unblocked. | While one thread is blocked and waiting, second thread in the same task can run. |
| Multiple processes without using threads use more resources. | Multiple threaded processes use fewer resources. |
| In multiple processes, each process operates independently of the others. | One thread can read, write or change another thread's data. |

Advantages of threads:

- Minimize **context-switching** time.
- Use of threads provides **concurrency** within a process.
- **Efficient** communication, as they share memory and resources.
- **Economy**: more economical to create and context switch, the kernel no acts.
- Utilization of **multiprocessor architectures** to a greater scale and efficiency.

***Activity:** Use of threads is highly recommended while programming. Search Internet and find out examples of situations where you might use threads.*

What are the benefits of threads in multiprocessor systems?

3.7 Managing processes in the Operating system

The operating system creates and manages processes. When the user opens a new program, the operating system is responsible for creating and executing the process for this application.

A **new process** is always **created** required by **another process** or by the **user**.

As each process may create new processes, it appears a hierarchical relationship between every process running in the operating system and we can see them as a **processes tree**. When operating system boots and the kernel is loaded, the initial process of the system is created and then the other processes are hierarchically created: parents, children, grandparents, etc.

The operating system, to identify processes, gives a different **PID** (*process identifier*) to each process.

***PRACTICE:** Process management in Linux.*

Read the related practice document and do the required tasks.

4. MULTI-PROCESS PROGRAMMING IN JAVA

4.1 Introduction

Concurrent programming is an effective way to process information by allowing different events or processes alternating CPU to provide multiprogramming. Multiprogramming can occur between entirely **independent processes**, as might be the appropriate word processor, browser, music player, etc., or **between processes** that can **cooperate** to perform a common task.

As we know, the **operating system is responsible for providing multiprogramming** between all processes in the system, hiding this complexity both users and developers. However, if processes cooperate with each other, the **developer must implement all the communication and synchronization** needed for these processes using any of the technics seen before.

Independently of the platform or language, when **writing multi-process programs** follow the steps below:

1. **Functional decomposition.** Identify the different tasks or functions that the application must do and the relationship between them.
2. **Partition.** Assign a process to do each task and establish the communication and synchronizing scheme between them. The objective is to maximize the independence between processes as minimizing the communication.
3. **Implementation.** Program the application using the facilities given by the platform finally chosen.

To simplify, we are going to use the multiplatform Java Virtual Machine. Java allows only very simple methods to implement the inter-process communication and synchronization.

4.2 Creating and destroying a process

(More information at: <http://docs.oracle.com/javase/6/docs/api/java/lang/Process.html>)

The class **Process** provides methods for performing input from the process, performing output to the process, waiting for the process to complete, checking the exit status of the process, and destroying (killing) the process.

The methods that create processes may not work well for special processes on certain native platforms, such as native windowing processes, daemon processes, Win16/DOS processes on Microsoft Windows, or shell scripts.

By default, the created **subprocess does not have its own terminal or console**. All its standard I/O (i.e. stdin, stdout, stderr) operations will be redirected to the parent process, where they can be accessed via the **streams** obtained using specific methods (**getOutputStream()**, **getInputStream()**, and **getErrorStream()**). The parent process uses these streams to feed input to and get output from the subprocess.

Because some native platforms only provide **limited buffer size** for standard input and output streams, failure to promptly write the input stream or read the output stream of the subprocess may cause the subprocess to **block**, or even **deadlock**.

The subprocess is not killed when there are no more references to the Process object, but rather the **subprocess** continues **executing asynchronously**.

There is no requirement that a process represented by a Process object execute asynchronously or concurrently with respect to the Java process that owns the Process object.

Creating a Process

The ***ProcessBuilder.start()*** and ***Runtime.exec()*** methods create a native process and return an instance of a subclass of Process that can be used to control the process and obtain information about it.

ProcessBuilder.start() is the preferred way to create a Process.

Starting a new process that uses the default working directory and environment is easy:

```
Process p = new ProcessBuilder("myCommand", "myArg").start();
```

The **start** method checks that “myCommand” is a valid operating system command, which is system-dependent, but at the very least it must be a non-empty list of non-null strings.

Among the many things that can go wrong are:

- The operating system program file was not found.
- Access to the program file was denied.
- The working directory does not exist.
- Etc.

In such cases an **exception** will be thrown. The exact nature of the exception is system-dependent, but it will always be a subclass of ***IOException***.

Example:

```

import java.io.IOException;
import java.util.Arrays;

public class ProcessCreating {

    public static void main(String[] args) throws IOException {
        if (args.length <= 0) {
            System.err.println("I need a command/program to execute!!");
            System.exit(-1);
        }

        ProcessBuilder pb = new ProcessBuilder(args);
        try {
            Process process = pb.start();
            int exitValue = process.waitFor();
            System.out.println("Execution of " +
                Arrays.toString(args) + "returns exit value: " + exitValue);
        }
        catch (IOException ex) {
            System.err.println("IO Exception !!");
            System.exit(-1);
        }
        catch (InterruptedException ex) {
            System.err.println("The child process exited with error");
            System.exit(-1);
        }
    }
}

```

Destroying a process

The ***destroy()*** method kills the subprocess. The subprocess represented by this Process object is forcibly terminated and all its resources released. In Java, this is done by the Garbage Collector.

We can see how to destroy a process in the following code. In this case we create the process using ***Runtime*** class:

```

import java.io.IOException;

public class ProcessDestroy {

    public static void main(String[] args) throws IOException {
        if (args.length <= 0) {
            System.err.println("I need a command/program to execute!!");
            System.exit(-1);
        }

        Runtime runtime = Runtime.getRuntime();

        try {
            Process process = runtime.exec(args);
            process.destroy();
        }
        catch (IOException ex) {
            System.err.println("IO Exception !!");
            System.exit(-1);
        }
    }
}

```

4.3 Communicating processes

A process, like other running program, receives some information, processes it and produces a result through:

- **Standard input** (*stdin*): It is usually the keyboard but might be a file, other process, etc.
- **Standard output** (*stdout*): it is usually the display, but could be the printer, other process, etc.
- **Standard error output** (*stderr*): It is usually the same as standard output but might be redirected to a file.

In Java, the child process created from class *Process* does not have its own communication interface so it is no possible a direct communication with it. Therefore, all the process input and output of information is redirected to parent process through the following **data streams**:

- **OutputStream**: source of **standard input**. By default, the subprocess reads input from a *pipe*. Java code can access this pipe via the output stream returned by the *Process* method *getOutputStream()*. [However, standard input may be redirected to another source using the *ProcessBuilder* method *redirectInput()*.*]
- **InputStream** and **ErrorStream**: destination for **standard output** and **standard error**. By default, the subprocess writes output and error to *pipes*. Java code can access these pipes via the inputs streams returned by the *Process* methods *getInputStream()* and *getErrorStream()*. [However, standard output and standard error may be redirected to another source using the *ProcessBuilder* methods *redirectOutput()* and *redirectError()*.*]

* **since v.1.7**

→ Initially, the *standard output* and *error output* of a subprocess are sent to two separate streams. This is set via the *ProcessBuilder* property *redirectErrorStream* (by default set to *false*). If *redirectErrorStream* is set to *true*, then subprocess *stdout* and *stderr* will be merged (this makes it easier to correlate error messages with the corresponding output).

It is important to take into account that the buffer size for **stdin** and **stdout** could be limited in the operating system and eventually causing the child process to block. Therefore, in Java the parent-child process communication is done through defining a *buffer* for the above *data streams*.

Let's see an example of communication between parent and child process:

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Arrays;

public class ProcessCommunication {

    public static void main(String[] args) throws IOException {

        Process process = new ProcessBuilder(args).start();
        InputStream inStream = process.getInputStream();

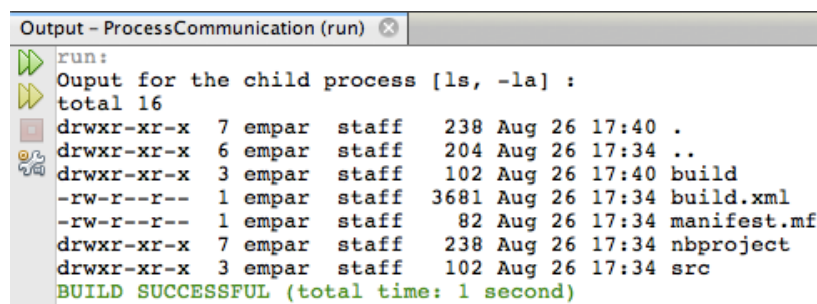
        InputStreamReader inStreamReader = new InputStreamReader(inStream);
        BufferedReader bufReader = new BufferedReader(inStreamReader);

        System.out.println("Output for the child process "
            + Arrays.toString(args) + " : ");

        String line;
        while ((line = bufReader.readLine()) != null) {
            System.out.println(line);
        }
    }
}

```

The **output result** for command **ls -la** could be:



```

Output - ProcessCommunication (run)
run:
Output for the child process [ls, -la] :
total 16
drwxr-xr-x  7 empar  staff   238 Aug 26 17:40 .
drwxr-xr-x  6 empar  staff   204 Aug 26 17:34 ..
drwxr-xr-x  3 empar  staff   102 Aug 26 17:40 build
-rw-r--r--  1 empar  staff  3681 Aug 26 17:34 build.xml
-rw-r--r--  1 empar  staff    82 Aug 26 17:34 manifest.mf
drwxr-xr-x  7 empar  staff   238 Aug 26 17:34 nbproject
drwxr-xr-x  3 empar  staff   102 Aug 26 17:34 src
BUILD SUCCESSFUL (total time: 1 second)

```

As we can see, using *data streams* to communicate processes in Java is quite simple. However, there are other alternatives ways like:

- **Sockets** between processes.
- Using the **Java Native Interface (JNI)**
- Using other non-standard java communication libraries like the open-source library **CLIPC** (<http://clipc.sourceforge.net/>). With this library it is possible to make use of shared memory, pipes, semaphores, etc., that do not come included in Java standard libraries.

Activity: Find information in the Internet about the **JNI** and answer these questions:

1. What is the **JNI** for?
 2. What are the reasons to use the **JNI**?
 3. Search for an example of use and give your opinion:
 - a. Do you think **JNI** is easy to learn? Why?
 - b. Do you think that the native code is portable?
-

4.4 Synchronizing

Data streams can be considered as a way of synchronization between parent and child processes through the interchange of information.

Besides, the parent process can block itself and *wait* for the child process to finish and complete the *exit* operation. When the child process exits it returns an integer value to the parent with the result of execution. An exit value of zero means that the child process ended normally.

The method ***waitFor()*** in the class *Process* waits until the subprocess terminates (if the current process is interrupted while waiting, *InterruptedException* is thrown) and returns the exit value of the terminated process.

The exit value of the child process can be also obtained calling the method *exitValue()*, but if the child process has not finished yet *IllegalThreadStateException* is thrown.

Here is an example:

```
import java.util.Arrays;
import java.io.IOException;

public class ProcessSynchronize {

    public static void main(String[] args) {
        try {
            Process process = new ProcessBuilder(args).start();
            int exitValue = process.waitFor();

            System.out.println("The exit value for " + Arrays.toString(args)
                               + " is: " + exitValue);
        } catch (IOException ex) {
            System.out.println("There was an error while running "
                               + Arrays.toString(args)
                               + " --> " + ex.getMessage());
        } catch (InterruptedException ex) {
            System.out.println("The current process was interrupted --> "
                               + ex.getMessage());
        }
    }
}
```

<http://felinfo.blogspot.com.es/2009/12/ejecutar-comandos-linux-y-ms-dos-desde.html>

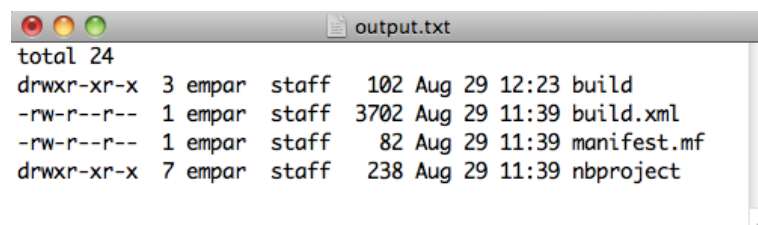
Practical Activity: I/O Redirection

One of the most useful and powerful things you can do with I/O redirection is to connect multiple commands together with what are called *pipes*. With pipes, the standard output of one command is fed into the standard input of another. For example:

```
$ ls -l | head -n5
→ Displays the first 5 lines of the ls -l command
```

In this activity you have to develop a multi-process application that creates two sub-processes (one for each command) and redirects the standard output of the first process (*in the example, ls -l*) to the standard input of the second (*in the example, head -n5, that waits for data in its stdin*):

- Program a new class named *Piping* that will do all the work to communicate and synchronize the two subprocesses. It will consist of:
 - Two Constructors:
 - `public Piping(String[] command1, String[] command2, String outputFile) { }`
 - `public Piping(String[] command1, String[] command2) {..... }` → the default file name is: *output.txt*.
 - New method *start()* that creates the subprocesses, pipe their data streams and writes the final output to a text file:



It **throws** any necessary exception in case of error as well.

- New method *getOutputFileName()* that returns the output file name.
 - Override the base method *toString()*. It must return a String containing the message: "The exit value of execution is: <exitValueofSecondProcess>"
 - Try the programmed class *Piping* from the main project class, *main()* method. Create an instance of *Piping* and catch all the exceptions thrown from the *Piping* class.
-

5. EXERCISES