Multiprogramación

La **Multiprogramación** es una técnica que permite que varios Procesos puedan existir simultáneamente en la memoria de un equipo informático y puedan ser ejecutados de forma concurrente (ya sea real o no).

La idea tras la Multiprogramación es solucionar dos problemas concretos. El primero se refiere a la enorme **diferencia de velocidad** entre el procesador y los dispositivos de Entrada / Salida. Actualmente esta diferencia es grande incluso entre el procesador y la memoria RAM. No puede ser que un Proceso en Ejecución se pase la mitad del tiempo viendo si termina de leerse un fichero del disco duro. El segundo, es aprovechar los tiempos muertos del procesador. Si un Proceso no hace nada, automáticamente se le reemplaza por otro.

La solución es la **Concurrencia de Procesos**. Consiste en cargar tantos Procesos en memoria como sean posibles, e ir alternando entre ellos cada vez que uno realiza una operación que sea lenta, como una Entrada / Salida. Existen varias clases de **Concurrencia**.

- La **Real**, que sólo es posible cuando existen varios procesadores (o núcleos). En estos casos, se pueden ejecutar simultáneamente tantos Procesos como procesadores haya disponibles. Esto es lo que se llama **Procesamiento Paralelo**.

- La **Virtual**, que requiere el uso de **Multiprogramación**. Sólo se dispone de un único procesador, por lo tanto, en cada instante de tiempo sólo puede haber una único Proceso en ejecución. El Sistema Operativo va alternando los Procesos de forma que cada uno disponga de un breve periodo de tiempo de Procesador. Lo que se consigue con esto es el **Procesamiento Concurrente** (no real).

  Nótese que existen procesadores de varios núcleos. Estos procesadores son capaces de ejecutar varias instrucciones a la vez, por lo tanto, se les considera Procesamiento Paralelo. No obstante, se complica tanto la planificación de Procesos que no vamos a meternos a explicarlo.

Programación Distribuida

La **Programación Distribuida** es otro de los paradigmas de la Multiprogramación. En este tipo de arquitecturas, la ejecución de un software se distribuye entre diferentes equipos, de forma que todos colaboran con sus procesadores. Es una forma potente, escalable y desde luego mucho más barata que desarrollar procesadores de alta gama. Para conectar estos equipos se utilizan redes, y el resultado es lo que se denominan *granjas de ordenadores*. No todas las tareas pueden distribuirse. Por tanto, se define el **Procesamiento Distribuido** como aquel en el que un proceso se ejecuta en procesadores independientes conectados y sincronizados.

Programación Multihilo

Todo el código de un Proceso está formado por una serie de **sentencias** que se ejecutan de forma secuencial. Mientras no se termine la primera, no es posible pasar a la segunda; y así hasta que se termine la ejecución del Proceso. Pero ocurre que en ocasiones es posible *trocear* el código en partes más pequeñas y ejecutarlas por separado y en **paralelo**. Esto se conoce como **Programación Multihilo**.

Un Hilo no es lo mismo que un Proceso. Podría decirse (para entendernos) que un Proceso puede tener o un Hilo de ejecución y ser **Monohilo**; o tener varios Hilos y ser **Multihilo**. Al revés no es posible: un Hilo no puede tener Procesos. Veremos Hilos en los temas siguientes.

Bifurcación o fork

La bifurcación o **fork** es una operación que permite generar una copia idéntica de un **Proceso**. Al Proceso original se le llama Padre, y a sus copias Hijos. Todos ellos tienen un PID diferente y su código propio en su espacio de memoria, por lo tanto, no se 'pisan' cuando cambian los valores de las variables etc. Un ejemplo de un fork en Linux (en C) es el siguiente. Veremos más sobre este asunto más adelante.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main () {
    int pid;
    pid = fork();

    // Ambos procesos Padre e Hijo tienen el mismo codigo
    if (pid == 0) {
        // Pero esto solo lo ejecuta el Hijo
        printf ("Soy el Hijo %d del Padre %d\n", getpid(), getppid());
    } else {
        // Y esto solo lo ejecuta el Padre
        printf ("Soy el Padre %d del Hijo %d\n", getpid(), pid);
    }
    printf ("He terminado %d\n", getpid());
    exit(0);
}
```

Comunicación entre Procesos

Por definición, los **Procesos** son elementos estancos. Cada uno tiene su espacio de memoria, su tiempo de procesador asignado y su estado. Si embargo, los Procesos deben de comunicarse entre sí. Por ejemplo, la Salida de un Proceso puede ser la Entrada de otro Proceso que está en espera.

La comunicación entre procesos se llama **IPC** (*Inter-Process Comunication*) y existen varias formas de implementarlas.

- **Sockets**: Son mecanismos de comunicación de bajo nivel. Establecen secuencias de bytes bidireccionales entre diferentes Procesos o incluso máquinas, y pueden ser programados en diferentes lenguajes como Java.

- **Flujos de E/S**: Interceptando los flujos de Entrada y Salida, es posible comunicar dos **Procesos** mientras uno de ellos sea Hijo del otro. Un ejemplo de esto son los Pipes que programaremos en los siguientes apartados.

- **RPC**: Llamadas a procedimientos remotos (*Remote Process Call*). Consiste en realizar llamadas a métodos de otros **Procesos** que pueden estar incluso en una máquina remota. En Java, a esta tecnología se la llama RMI (*Remote Method Invocation*), equivalente a RPC, pero orientada a objetos.

- **Persistencia**: Un **Proceso** escribe en un fichero o base de datos, y otro **Proceso** lee del mismo sitio. Muy sencillo para cuando no es necesario complicarse la vida.

- **Servicios**: Pueden usarse otros Servicios como por ejemplo FTP, servicios web, tecnologías en la nube, etc.