PSP02.- PROGRAMACIÓN MULTIPROCESO

1.- CONCEPTOS BÁSICOS.

Un **programa** es una secuencia de instrucciones escritas en un lenguaje de programación para realizar una tarea o necesidad de un usuario. Ejemplos: Word, Firefox...

Un **proceso** es un programa en ejecución. Un proceso necesita los siguientes **recursos** del sistema para poder realizar una tarea:

- Tiempo de CPU.
- Memoria.
- Memoria.
- Archivos.
- Dispositivos de E/S.

Los sistemas operativos actuales son **multiproceso o multitarea**, ya que pueden ejecutar varios procesos simultáneamente. Para ello la CPU va alternando la ejecución de los diferentes procesos. En ordenadores con varias CPU podemos ejecutar varios procesos simultáneamente.

La **programación multiproceso** permite que múltiples procesos se puedan ejecutar simultáneamente sobre el mismo código de programa. Cuando tenemos dos procesos en ejecución de un determinado programa (por ejemplo, Microsoft Word), podemos trabajar con diferentes documentos.

Las **funciones del Sistema Operativo** como gestor de procesos son las siguientes:

- Creación y eliminación de procesos.
- Planificación de procesos (procurando la ejecución de múltiples procesos para maximizar la utilización del procesador).
- Establecimiento de mecanismos para la sincronización y comunicación de procesos.
- Manejo de bloqueos mutuos.

La información sobre los procesos que están pendientes de ejecutar y su estado se debe guardar en algún lugar. El **BCP** es una estructura de datos llamada Bloque de Control de Proceso donde se almacena la información acerca de los procesos:

- Identificación del proceso: cada proceso es referenciado por un identificador único.
- Estado del proceso.
- Contador del programa: en qué número de instrucción se encuentra el programa.
- Registros de la CPU.
- Información de planificación de la CPU: la prioridad del proceso, por ejemplo.
- Información de gestión de memoria.
- Información contable: cantidad de tiempo de CPU y tiempo real consumido.
- Información de estado de E/S: lista de dispositivos asignados, archivos abiertos, etc.

2.- PROCESOS EN LINUX.

Mediante el **comando ps** (process status) de Linux se puede ver información asociada a los procesos:

aitor@aitor-VirtualBox:~$ ps

PID	TTY	TIME CMD
2336	pts/1	00:00:00 bash
2402	pts/1	00:00:00 ps

La información que muestra el comando ps es la siguiente:

identificador del proceso

PID:	identificador del proceso
TTY:	terminal asociado al proceso. Si aparece interrogación es que el proceso no se ha ejecutado desde un terminal
TIME:	tiempo de ejecución asociado. El tiempo total que el proceso ha utilizado la CPU
CMD:	nombre del proceso

En el caso anterior, hay una terminal abierta (pts/1) y en ella se ejecuta el comando ps.

El **comando ps -f** muestra más información sobre los procesos:

aitor@aitor-VirtualBox:~$ ps -f

UID	PID	PPID C	STIME	TTY	TIME	CMD
aitor	2433	2425 0	05:50	pts/1	00:00:00	bash
aitor	2569	2433 0	05:57	pts/1	00:00:00	ps -f

UID:	nombre del usuario
PPID:	PID del padre de cada proceso
C:	porcentaje de recursos de la CPU utilizado por el proceso
STIME:	hora de inicio del proceso

El **comando ps -ef** muestra todos los procesos, incluidos los que no se han ejecutado desde el terminal.

aitor@aitor-VirtualBox:~$ ps -ef

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	05:43	?	00:00:01	/sbin/init
root	2	0	0	05:43	?	00:00:00	[kthreadd]
root	3	2	0	5:43	?	0:00:00	[ksoftirqd/0]
root	5	2	0	5:43	?	0:00:00	[kworker/0:0H]
aitor	2425	1493	0	5:50	?	0:00:00	gnome-terminal
aitor	2432	2425	0	5:50	?	0:00:00	gnome-pty-helper
aitor	2433	2425	0	5:50	pts/1	0:00:00	bash
root	2470	2	0	5:50	?	0:00:00	[kworker/0:0]
root	2480	1	0	5:50	?	0:00:00	/lib/systemd/systemd-hostnamed
aitor	2484	1493	21	5:50	?	0:00:08	/usr/lib/firefox/firefox
aitor	2511	1493	0	5:51	?	0:00:00	/usr/lib/libunity-webapps/unity-webapp....

aitor	2554	2433	0	5:51	pts/1	0:00:00	ps -ef

Con el **comando ps –AF,** además de mostrar todos los procesos, muestra más información sobre ellos.

aitor@aitor-VirtualBox:~$ ps -AF

UID	PID	PPID	C	SZ	RSS	PSR	STIME	TTY	TIME	CMD
root	1	0	0	8443	2088	0	05:43	?	00:00:01	/sbin/init
root	2	0	0	0	0	0	05:43	?	00:00:00	[kthreadd]
aitor	2433	2425	0	6747	4032	0	05:50	pts/1	00:00:00	bash
root	2470	2	0	0	0	0	05:50	?	00:00:00	[kworker/0:0]
root	2480	1	0	4380	1228	0	05:50	?	00:00:00	/lib/systemd/s...
aitor	2484	1493	7	224155	176684	0		0	05:50 ?	00:00:20
/usr/lib/firefox/firefox										
aitor	2511	1493	0	71806	3644	0	05:51	?	00:00:00	/usr/lib/libuni....
aitor	2565	2433	0	5677	2488	0	05:55	pts/1	00:00:00	ps -AF

SZ:	tamaño real de la imagen del proceso
RSS:	tamaño de la parte residente del proceso en memoria en KB
PSR:	procesador que el proceso tiene actualmente asignado

En la distribución GNU/Linux Ubuntu es posible obtener de forma gráfica esta información relativa a los procesos mediante la aplicación llamada **Monitor del sistema.**

Caso práctico

Actividad 1 Comando **top** de Linux. Prueba este comando y averigua la información que se obtiene a partir de él.

Actividad 2 Comando **free** de Linux. Prueba este comando y averigua la información que se obtiene a partir de él.

Actividad 3 Ejecuta el **Monitor del Sistema en Ubuntu** y averigua la información que se obtiene a partir de él.

3.- PROCESOS EN WINDOWS.

A través de la herramienta Administrador de tareas (combinación de teclas `Control` + `Alt` + `Supr`), es posible conocer, entre otros, los procesos que se están ejecutando en un momento dado en un sistema operativo Windows.

El comando **tasklist** de Windows nos muestra los procesos que se están ejecutando. Para ello vamos a ir al intérprete de comandos **(Inicio → Ejecutar → cmd)** y lo ejecutamos:

```
C:\> tasklist
```

Podemos visualizar los servicios que se estén ejecutando bajo el proceso svchost.exe de la siguiente forma:

```
C:\>  tasklist /svc /fi “imagename eq svchost.exe”
```

4.- ESTADOS DE UN PROCESO.

A medida que un proceso va evolucionando en su ciclo de vida, puede ir cambiando de estados. Cada proceso puede estar en alguno de los siguientes estados:

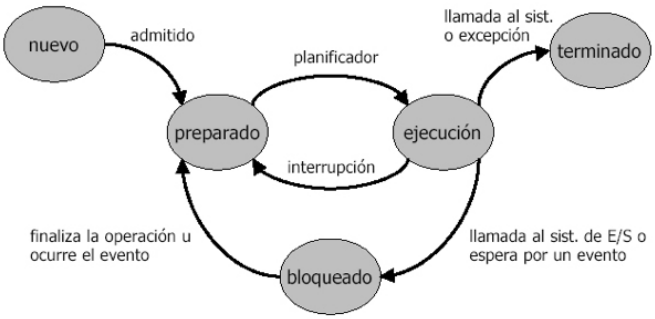
- **Nuevo (new):** el proceso se está creando.
- **En ejecución (running):** el proceso está en la CPU ejecutando instrucciones.
- **Bloqueado (waiting, en espera):** proceso esperando a que ocurra un suceso (ej. terminación de una operación de E/S o recepción de una señal).
- **Preparado (ready, listo):** esperando que se le asigne a un procesador.
- **Terminado (terminated):** finalizó su ejecución, por tanto no ejecuta más instrucciones y el sistema operativo le retirará los recursos asignados.

Debes conocer

Sólo un proceso puede estar ejecutándose en cualquier procesador en un momento dado, aunque varios procesos pueden estar listos y esperando.

0.

Estados de un proceso y posibles transiciones entre ellos



E *(link: https://www.google.es)* laboración propia

Para que un programa se ejecute, el SO debe crear un proceso para él. En un sistema con multiprogramación, el procesador ejecuta código de distintos programas que pertenecen a distintos procesos.

Aunque dos procesos estén asociados al mismo programa, se consideran dos secuencias de ejecución separadas, por lo que cada una de ellas se considera un proceso diferente.

Se denomina **traza de un proceso** al listado con la secuencia de instrucciones que se ejecutan para el mismo.

5.- CREACIÓN DE PROCESOS EN C.

Los procesos se crean mediante una llamada al sistema de “crear proceso”, durante el curso de su ejecución. El proceso creador se denomina proceso padre, y el nuevo proceso, proceso hijo.

Cuando un proceso crea un nuevo proceso, existen 2 posibilidades en términos de ejecución del mismo:

- Padre e hijo se ejecutan concurrentemente
- El padre espera a que finalice la ejecución del hijo

En el sistema operativo UNIX existen dos funciones básicas para la creación de procesos:

Función fork()

Cuando es llamada, crea un proceso hijo que es una copia casi exacta del proceso padre (duplicado del padre). Ambos procesos continúan ejecutándose desde el punto en el que se hizo la llamada a la función fork().

En UNIX los procesos se identifican mediante un **"identificador de proceso" (PID)** que es un número entero único. Ambos procesos continúan su ejecución en la instrucción que sigue al **fork()**, pero con la siguiente diferencia:

- El código que el proceso hijo recibe del fork() es cero
- El código que el proceso padre recibe del fork() es el PID del proceso hijo

Función exec()

Tras crear un nuevo proceso después de llamar a la función **fork()**, Linux llama a una función de la familia **exec()**. Esta función reemplaza el programa que se está ejecutando en el proceso por otro programa. Cuando un programa llama a una función del tipo **exec()**, su ejecución cesa de inmediato y comienza a ejecutarse el nuevo programa desde el principio, suponiendo que no se haya producido ningún error durante la llamada a la función **exec()**.

Habitualmente es uno de los dos procesos (el padre o el hijo) quien llama a la función **exec()**, después de haberse creado un proceso hijo mediante el uso de la función **fork()**.

Todo proceso en Linux lleva asociado un identificador (nº de 16 bits que se asigna secuencialmente por cada nuevo proceso que se crea), que es único para cada proceso y que se conoce como **PID**. A excepción del proceso raiz (init), el resto de procesos tienen asociado un proceso padre, cuyo identificador de proceso se conoce como **PPID**.

6.- IDENTIFICACIÓN DE PROCESOS EN C.

Un programa puede obtener el identificador del proceso en el que se está ejecutando por medio de la función de llamada al sistema **getpid()**, mientras que el identificador del proceso padre puede obtenerse por medio de la función de llamada al sistema **getppid()**.

Al finalizar el **fork()**, tanto el proceso padre como el hijo continúan su ejecución a partir de la siguiente instrucción. Si un padre quiere esperar a que su hijo termine, deberá utilizar la llamada al sistema **wait().** wait() detiene la ejecución del proceso (lo pasa al estado bloqueado) hasta que su hijo termine. wait() regresa de inmediato si el proceso no tiene procesos hijos. Cuando wait() regresa por finalización de la ejecución de su hijo, el valor devuelto es positivo e igual al PID de dicho proceso. En caso contrario, devuelve –1 y pone un valor en errno.

Debes conocer

El siguiente programa en C escribe el identificador de un proceso (PID) y el identificador de su proceso padre (PPID).

```
//getpid.c
#include <stdio.h>
#include <unistd.h>
int main ()
        {
        printf("El identificador de este proceso es PID = %d\n",(int)getpid ());
        printf("El identificador del proceso padre es PPID = %d\n",(int)getppid ());
        }
```

En sistemas GNU/Linux, para **compilar y ejecutar un programa escrito en C:**

$ gcc programa.c -o programa	**PARA COMPILAR**
$ ./programa	**PARA EJECUTAR**

En el lenguaje de programación C, **pid_t** es un tipo de datos que se usa para definir y manejar variables que van a contener el PID de un determinado proceso. Ejemplo de declaración de una variable llamada pid con tipo de datos pid_t:

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

void main ()
{
    pid_t pid;
    pid = fork();
    if (pid == -1) ......
```

7.- FUNCIONES EN C PARA LA GESTIÓN DE PROCESOS.

7.1.- LA FUNCIÓN FORK.

Debes conocer

Crear un nuevo proceso hijo que es una copia exacta en código y datos del proceso que lo ha llamado. Las variables del proceso hijo son una copia de las variables del proceso padre, pero éstas utilizan un espacio de memoria diferente, por lo que si se modifica una variable del proceso hijo, no afecta a las del proceso padre.

Valores devueltos:

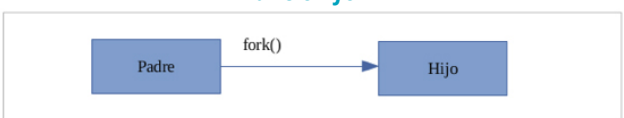
- Devuelve **-1** si se produce algún error en la ejecución.
- Devuelve **0** si no se produce ningún error y nos encontramos en el proceso hijo.
- Devuelve el **PID** asignado al proceso hijo, si no se produce ningún error y nos encontramos en el proceso padre.

Librería que necesita: unist.h

Ejercicio Resuelto

Ejemplo: un proceso padre crea un proceso hijo.

Función fork



E *(link: https://www.google.es)* laboración propia

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

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

void main () {
    pid_t pid;
    pid = fork();
    if (pid == -1)
    {
        printf("No se ha podido crear el proceso hijo...");
        exit (-1);
    }
    if (pid == 0)   ////////// Nos encontramos en el proceso hijo
        {
        printf ("Soy el proceso hijo \n\t Mi PID es %d, El PID de mi padre es: %d.\n", getpid(), getppid());
        }
        else ////// Nos encontramos en el proceso padre
        {
        printf ("Soy el proceso padre \n\t Mi PID es %d, El PID de mi padre es: %d.\n\t Mi hijo : %d terminó. \n"
        }
        exit(0);
}
```

7.2.- LA FUNCIÓN EXECL.

Debes conocer

Permite ejecutar un proceso, pasándole como parámetros un programa junto con sus parámetros asociados.

Libreríaque necesita: unist.h

Sintaxis:

```
int execl (const char *fichero, const char *arg 0, ….char *argn, (char *) NULL );
```

Ejercicio Resuelto

Ejemplo: ejecutar el comando Linux /bin/ls -l:

```
execl ("/bin/ls","ls", "-l", (char *)NULL);

// funcionexec.c
#include <unistd.h>
#include <stdio.h>
void main ()
{
    printf("Ejemplo de uso de exec ():");
    printf("Los archivos en el directorio son: \n");
    execl ("/bin/ls","ls", "-l", (char *)NULL);
    printf (" Esto no se ejecuta !!!!!");
}
```

7.3.- LA FUNCIÓN WAIT.

Debes conocer

Permite que un proceso padre se quede bloqueado hasta que su proceso hijo termine la ejecución. Devuelve la identificación del proceso hijo cuya ejecución ha finalizado.

Librería que necesita: sys/wait.h

Sintaxis:

```
pid_t wait (int *status);
```

Ejercicio Resuelto

Ejemplo 2: el proceso padre crea un proceso hijo y éste, a su vez, crea un proceso nieto.

Ejercicio Resuelto

Ejemplo 1: el proceso padre espera a la finalización del proceso hijo.

```
// funcionesforkwait.c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h.h>

void main ()
{
    pid_t pid, Hijo_pid;
    pid = fork();
    if (pid == -1)
    {
        printf("No se ha podido crear el proceso hijo...");
        exit (-1);
    }
    if (pid == 0)   ////////// Nos encontramos en el proceso hijo
        {
                printf ("Soy el proceso hijo \n\t Mi PID es %d, El PID de mi padre es: %d.\n", getpid(), getppid(
        }
        else ////// Nos encontramos en el proceso padre
        {
        Hijo_pid = wait(NULL);
        printf ("Soy el proceso padre \n\t Mi PID es %d, El PID de mi padre es: %d.\n\t Mi hijo : %d terminó. \n"
        }
        exit(0);
```

Ejercicio Resuelto

Ejemplo 2: el proceso padre crea un proceso hijo y éste, a su vez, crea un proceso nieto.

```
// funcionesforkwait2.c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h.h>

void main ()
{
    pid_t pid, Hijo_pid, pid2, Hijo2_pid;
    pid = fork();
    if (pid == -1)
    {
        printf("No se ha podido crear el proceso hijo...");
        exit (-1);
    }
    if (pid == 0)   ////////// Nos encontramos en el proceso hijo
        {
                pid2 = fork(); ///////// Soy el hijo creo el nieto
                switch (pid2)
                {
                        case -1:
                                printf("No se ha podido crear el proceso nieto...");
                                exit (-1);
                                break;
                        case 0:
                                printf ("Soy el proceso NIETO \n\t Mi PID es %d, El PID de mi padre es: %d.\n", g
                                break;
                        default:  ///////// Proceso padre
                                Hijo2_pid = wait (NULL);
                        printf ("Soy el proceso HIJO \n\t Mi PID es %d, El PID de mi padre es: %d.\n\t Mi hijo :
                                                }
        }
        else ////// Nos encontramos en el proceso padre
        {
                Hijo_pid = wait(NULL);
                printf ("Soy el proceso PADRE \n\t Mi PID es %d, El PID de mi padre es: %d.\n\t Mi hijo : %d term
        }
        exit(0);
}
```

7.4.- LA FUNCIÓN EXIT.

Debes conocer

Provoca que un proceso termine, devolviendo un estado al proceso padre. Si el estado devuelto es 0 la terminación del proceso ha sido satisfactoria. Cualquier otro estado que se devuelva (-1, 1, ...), significa que la terminación del proceso no ha sido satisfactoria.

Librería que necesita: stdlib.h

Sintaxis:

```
void exit (int status);
```

Ejercicio Resuelto

Ejemplo 1: el proceso padre espera a la finalización del proceso hijo.

```
// funcionesforkwait.c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h.h>

void main ()
{
    pid_t pid, Hijo_pid;
    pid = fork();
    if (pid == -1)
    {
        printf("No se ha podido crear el proceso hijo...");
        exit (-1);
    }
    if (pid == 0)   ////////// Nos encontramos en el proceso hijo
        {
                printf ("Soy el proceso hijo \n\t Mi PID es %d, El PID de mi padre es: %d.\n", getpid(), getppid(
        }
        else ////// Nos encontramos en el proceso padre
        {
        Hijo_pid = wait(NULL);
        printf ("Soy el proceso padre \n\t Mi PID es %d, El PID de mi padre es: %d.\n\t Mi hijo : %d terminó. \n"
        }
        exit(0);
```

8.- COMUNICACIÓN ENTRE PROCESOS.

Existen varias formas de comunicación entre procesos en GNU/Linux: pipes, colas de mensajes, semáforos y segmentos de memoria compartidos. En esta unidad didáctica se van a ver los pipes (tuberías).

8.1.- PIPES SIN NOMBRE.

Debes conocer

Definición: especie de falso fichero que sirve para conectar dos procesos.

Funcionamiento

- El proceso A quiere enviar un mensaje al proceso B. El proceso A escribe en el pipe como si fuese un fichero de salida y el proceso B lee del pipe como si fuese un fichero de entrada.
- Cuando un proceso quiere leer del pipe y éste está vacío, tendrá que esperar hasta que otro proceso escriba en él.
- Cuando un proceso intenta escribir en un pipe y éste está lleno, el proceso se bloqueará hasta que el pipe se vacíe.
- El pipe es bidireccional, pero cada proceso lo utiliza en una sola dirección (el kernel gestiona la sincronización).

Dos procesos conectados por un pipe

Elaboración propia

Función pipe

- Sintaxis para la creación de un pipe (tubería):

```
#include <uninstd.h>
int pipe(int fd[2]);
```

- Esta función recibe un solo argumento, que es un array de dos enteros: fd[0] contiene el descriptor para la lectura del pipe y fd[1] contiene el descriptor para la escritura del pipe.
- Esta función recibe un solo argumento, que es un array de dos enteros: fd[0] contiene el descriptor para la lectura del pipe y fd[1] contiene el descriptor para la escritura del pipe.
- Devolución de valores de la función:
  - Si la función tiene éxito, devuelve 0 y el array contendrá dos nuevos descriptores de archivos para ser usados por la tubería
  - Si la función no tiene éxito, devuelve -1

Funciones de apoyo

- **read** (para leer datos de un pipe): intenta leer count bytes del descriptor de fichero definido en fd, para guardarlos en el buffer buf. Devuelve el número de bytes leídos. Si se compara este valor con la variable count, podremos saber si se ha conseguido leer tantos bytes como se pedían. int read (int fd, void *buf, int count);
- **write** (para escribir datos en un pipe): a buf se le da el valor de lo que queramos escribir, definimos su tamaño en count y especificamos el fichero en el que escribiremos en fd. int write (int fd, void *buf, int count);
- **open** (para abrir un fichero): abre el fichero indicado en la cadena fichero según el modo de acceso indicado en el entero modo (0 para lectura, 1 para escritura, 2 para lectura y escritura, etc). Devuelve -1 si ocurre algún error. int open (const char *fichero, int modo);
- **close** (para cerrar un fichero): cierra el fichero cuyo descriptor se indique entre paréntesis. int close (int fd);

Ejercicio Resuelto

Ejemplo 1: abrir, escribir, leer y cerrar un fichero de nombre texto.txt

```
// lecturaescritura.c
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>  // funcion strlen
#include <unistd.h>

int main(){
  int fd, fd2, numbytes;
  char buffer[1024];
  char saludo[]="Un saludo!!!";

//// Escritura
  fd=open ("texto.txt",1); //// fichero se abre solo para escritura
  if (fd == -1)
     {
                printf("Error al abrir el fichero...\n");
                exit (-1);
     }
   printf("Escribo el saludo .... \n");
   write(fd, saludo, strlen(saludo));
   close(fd);

//// Lectura
  printf("Contenido del Fichero \n");
  fd2 = open("texto.txt", 0);
  while ((numbytes = read(fd2, &buffer, sizeof(char))) > 0){
        printf("%s",buffer);
  }
```

```
  close(fd2);
}
```

Ejercicio Resuelto

Ejemplo 2: el proceso hijo envía un mensaje al proceso padre.

```
                    Padre                                          Hijo
        fd[0]      fd[1]                                fd[0]        fd[1]
        ^
         I----------------------------     pipe  <---------------------------I
                        lee                              escribe
//pipes.c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h.h>

void main ()
{
        int fd[2];
        char buffer[30];
        pid_t pid;
        pipe(fd); // Se crea el pipe o tubería
        pid = fork();
        switch (pid) {
        case -1: // error
                printf("No se ha podido crear el proceso hijo...\n");
                exit (-1);
                break;
        case 0:   // Hijo
                printf ("El hijo escribe en el pipe...\n");
                write (fd[1], "Hola padre", 10);
                break;
        default: // Padre
                wait (NULL);
                printf ("El padre lee del pipe...\n");
                read (fd[0], buffer, 10);
                printf ("\tMensaje leído: %s\n", buffer);
                break;
        }
        exit(0);
}
```

Ejercicio Resuelto

Ejemplo 3: el proceso padre envía un mensaje al proceso hijo.

```
Padre                                          Hijo
fd[0]      fd[1]                                    fd[0]        fd[1]
                                                                 ^
I----------------->     pipe   ----------------I
escribe                 lee
```

Los procesos deben tratar el pipe de forma unidireccional, aunque éste sea bidireccional. Aunque en el ejemplo anterior no lo hemos hecho, deberíamos cerrar una de las direcciones, es decir, deberíamos decidir hacia qué dirección se envía.

Cuando la información va del padre al hijo, se recomienda que:

- El padre cierre el descriptor de lectura fd[0]

- El hijo cierre el descriptor de escritura fd[1]

Cuando la información va del hijo hacia el padre, se recomienda que:

- El padre cierre el descriptor de escritura fd[1]
- El hijo cierre el descriptor de lectura fd[0]

```
//pipes2.c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h.h>

void main ()
{
        int fd[2];
        char buffer[80];
        char saludoPadre[]="Hola hijo";
        pid_t pid;
        pipe(fd); // Se crea el pipe o tubería
        pid = fork();
        switch (pid) {
        case -1: // error
                printf("No se ha podido crear el proceso hijo...\n");
                exit (-1);
                break;
        case 0:   // Hijo recibe
                close(fd[1]); // Cierra el descriptor de escritura
                read (fd[0], buffer, sizeof(buffer)); // leo el pipe
                printf ("\tMensaje leído: %s\n", buffer);
                break;
        default: // Padre Envía
                close(fd[0]); // Cierra el descriptor de lectura
                write (fd[1], saludoPadre, sizeof(saludoPadre));
                printf ("El padre envía un mensaje al hijo,..\n");
                wait (NULL); // Espera al proceso hijo
                break;
        }
        exit(0);
}
```

8.2.- PIPES CON NOMBRE O FIFOS.

Definición:

- Los pipes vistos hasta ahora permiten enviar mensajes entre procesos emparentados o procesos padre-hijo.
- Para poder enviar mensajes entre procesos no emparentados, es necesario utilizar los pipes con nombre o FIFOS.
- Un FIFO es como un fichero con nombre que existe en el sistema de ficheros y que múltiples procesos pueden abrir, leer y escribir.

Funcionamiento:

- Los datos escritos en él se leen como en una cola (primero en entrar, primero en salir) y, una vez leídos, no pueden ser leídos de nuevo.
- Diferencias con los ficheros:
  - Una operación de escritura en el FIFO queda a la espera hasta que un proceso realice la lectura.
  - Solo se permite la escritura de información cuando un proceso la va a recoger.

Formas de crear un FIFO:

- **Ejecutando el comando mknod desde la línea de comandos de Linux:**

```
$ mknod FIFO1 p
donde FIFO1 es el nombre del FIFO y p indica se cree el FIFO
$ ls -al FIF*
prw-rw-r-- 1 ubuntu ubuntu 0    sept 12 15:20   FIFO1
```

Abrimos dos terminales. En el primero ejecutamos:

```
        $ cat FIFO1
```

Se puede apreciar que la ejecución del comando cat se queda a la espera. En el segundo terminal ejecutamos el siguiente comando:

```
$ ls -al > FIFO1
```

Una vez que el segundo comando se ejecuta, el primero deja de estar en espera.

- **Usando la función mknod() desde un programa en C:**

  Su **sintaxis** es la siguiente:

```
        int mknod (const char *pathname, mode_t modo, dev_t dev);
```

  donde:

  pathname: es el nombre del dispositivo creado, en nuestro caso el FIFO de nombre FIFO2

  modo: especifica tanto los permisos de uso como el tipo de modo que se creará; en nuestro caso S_IFIFO de tipo FIFO y 0666 lectura y escritura

- **Valores que devuelve:** 0 si ha funcionado correctamente y -1 si ha producido un error.

Ejercicio Resuelto

Ejemplo Ejecutar los programas fifocrea y fifoescribe en dos terminales diferentes. Cada vez que se ejecute el programa fifoescribe, enviar al programa fifocrea un mensaje para que éste lo visualice

```
//fifocrea.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h.h>
#include <sys/stat.h.h>
#include <fcntl.h>
#include <unistd.h>

int main (void)
{
        int fp;
        int p, bytesleidos;
        char saludo[] = "Un saludo !!!!!\n", buffer [10];
        p=mknod("FIFO2", S_IFIFO|0666, 0); /// permiso de lectura y escritura

        if (p== -1) {
                printf("Ha ocurrido un error.... \n"); // recuerda borrarlo la segunda vez...
                exit (0);
        }

        while (1) { // bucle infinito
                fp = open ("FIFO2", 0); // abro el pipe FIFO2 en modo lectura
                bytesleidos = read(fp, buffer, 1); // leo el primer byte
                printf("Obteniendo información...\n");
                while (bytesleidos != 0) {
                        printf("%s", buffer);
                        bytesleidos = read (fp, buffer, 1); // leo otro byte
```

```
        }
        close (fp);
}
return(0);
}


//fifoescribe.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

int main (void)
{
        int fp;
        int p, bytesleidos;
        char saludo[] = "Un saludo !!!!!\n";
        fp = open ("FIFO2", 1); /// abro el pipe FIFO2 con permiso de escritura
        if (fp == -1) {
                printf("Error al abrir el fichero... \n");
                exit (1);
        }
        printf("Mandando información al FIFO...\n");
        write (fp, saludo, sizeof(saludo));
        close (fp);
        return(0);
}
```

Consideraciones de prueba de los programas:

Si la ejecución del programa fifocrea da un error, probar a borrar el pipe con nombre FIFO2 con el siguiente comando Linux:

```
$ rm FIFO2
```

9.- SINCRONIZACIÓN ENTRE PROCESOS.

Introducción:

- En ejercicios anteriores se han visto los mecanismos más sencillos de comunicación entre procesos.
- Pero para que interactúen entre ellos, necesitan estar sincronizados; es decir, es necesario que haya un funcionamiento coordinado entre los procesos a la hora de ejecutar alguna tarea.
- Se pueden utilizar **señales** para la sincronización entre procesos.
- Existen varias funciones para que un proceso padre y un proceso hijo se comuniquen de forma síncrona mediante la utilización de señales: **signal, kill y sleep.**

Función signal:

- Una señal es como un aviso que un proceso envía a otro proceso.
- Esta función sirve para sincronizar procesos, está definida en la librería signal.h y tiene la siguiente sintaxis:

  ```
  void (signal (int Señal, void (*Func)(int)))(int);
  ```

  donde:

  - **Señal:** contiene el número de señal que se quiere capturar; puede tener, entre otros, los siguientes valores: SIGUSR (señal definida por el usuario para ser usada en programas de aplicación) o SIGKILL (señal que se utiliza para terminar con un proceso)
  - **Func:** contiene la función a la que se quiere llamar y se le suele conocer como manejador de la señal
- Para ver el resto de señales de esta función, ejecutar el comando man de GNU/Linux de la siguiente forma:

```
$ man 7 signal
```

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

- La función devuelve un puntero al manejador de la señal previamente instalado para esa señal
- Ejemplo de uso de la función:

```
signal(SIGUSR1,gestion_padre)
```

significa que cuando el proceso padre reciba una señal de tipo SIGUSR1, se realizará una llamada a la función gestión_padre()

Función kill:

- Esta función sirve para suspender al proceso que realiza la llamada la cantidad de segundos indicada o hasta que se reciba una señal.
- Está definida en la librería unistd.h y tiene la siguiente sintaxis:

```
int sleep (int Seconds);
```

donde:

Seconds: contiene la cantidad de segundos que se quiere suspender el proceso

Función sleep:

- Esta función sirve para suspender al proceso que realiza la llamada la cantidad de segundos indicada o hasta que se reciba una señal.
- Sintaxis:

```
        kill -s VALUE 1234
```

donde:

```
VALUE: es el valor de la señal
1234: es el ID del proceso
```

Ejercicio Resuelto

Ejemplo 1– Crear un proceso hijo y que el proceso padre le envíe al hijo 2 señales SIGUSR1. Definir la función manejador() para gestionar la señal, la cual será visualizada en un mensaje cuando el proceso hijo la reciba. En el proceso hijo realizar la llamada a la función signal(), donde se decide lo que se hará en el caso de recibir una señal (enviar un mensaje a la salida estándar del sistema). Después, hacer un bucle infinito que no hace nada. En el proceso padre, hacer las llamadas a la función kill() para enviar las señales. Con la función sleep() hacer que los procesos esperen un segundo antes de continuar.

```
//sincronizar1.c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
/* ------------------------------------------------*/
/* gestion de señales en proceso HIJO               */
void manejador (int signal )
{
        printf ("Hijo recibe señal...%d\n", signal);
}
/*************************************************/
void main ()
{
        int pid_hijo;
        pid_hijo = fork(); // creamos el hijo
        switch (pid_hijo) {
        case -1: // error
                printf("No se ha podido crear el proceso hijo...\n");
                exit (-1);
                break;
        case 0:   // Hijo
                signal (SIGUSR1, manejador); // Manejador de señal en hijo
                while(1) {}; // Bucle infinito
                break;
        default: // El padre envia 2 señales
                sleep (1);
                kill (pid_hijo, SIGUSR1); //Envía señal al hijo
                sleep (1);
                kill (pid_hijo, SIGUSR1); //Envía señal al hijo
                sleep (1);
                break;
        }
        exit(0);
}
```

Ejercicio Resuelto

Ejemplo 2 – El proceso padre envía señales al hijo y el proceso hijo envía señales al padre de forma indefinida

```
//sincronizar2.c
#include <stdlib.h>
```

```c
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
/* ------------------------------------------------*/
/* gestion de señales en proceso PADRE             */
void gestion_padre (int signal )
{
        printf ("Padre recibe señal...%d\n", signal);
}
/* gestion de señales en proceso HIJO              */
void gestion_hijo (int signal )
{
        printf ("Hijo recibe señal...%d\n", signal);
}
/************************************************/
int main (){
int pid_padre, pid_hijo;
        pid_padre= getpid ();
        pid_hijo = fork(); // creamos el hijo
        switch (pid_hijo) {
        case -1: // error
                printf("No se ha podido crear el proceso hijo...\n");
                exit (-1);
                break;
        case 0:   // Hijo
                signal (SIGUSR1, gestion_hijo); // Manejador de señal en hijo
                while(1) {
                        sleep(1);
                        kill(pid_padre, SIGUSR1); /// Envia señal al padre
                        pause (); // El hijo espera hasta que llegue una señal de respuesta
                };
                break;
        default: // Padre
                signal (SIGUSR1, gestion_padre);
                while(1) {
                        pause (); // El padre espera hasta que llegue una señal del hijo
                        sleep(1);
                        kill(pid_hijo, SIGUSR1); /// Envia señal al hijo
                };
                break;
        }
        return 0;
}
```

10.- TERMINACIÓN DE PROCESOS.

Procesos zombie:

- Son procesos que han finalizado su ejecución pero que todavía no han sido eliminados del sistema.
- Ejemplo: un programa crea un proceso hijo y luego llama a la función wait(). Si el proceso hijo no ha finalizado en este punto, el proceso padre se bloqueará en la llamada hasta que el proceso hijo finalice su ejecución. Pero si el proceso hijo finaliza antes de que el proceso padre haya llamado a la función wait(), el proceso hijo se convierte en un proceso zombie.

Finalización de procesos:

- Un proceso finaliza cuando, tras ejecutar su última instrucción, le pide al sistema operativo que lo elimine utilizando una llamada al sistema “exit”.
- Un proceso finaliza cuando su proceso padre emite una llamada al sistema para abortarlo.
- Algunas de las razones por las que un proceso padre podría terminar la ejecución de sus procesos hijo son las siguientes:
  - El hijo se excedió en la utilización de alguno de los recursos que se le asignaron
  - La tarea que se asignó al hijo ya no es necesaria
  - El padre va a salir, y el sistema operativo no permite que un hijo continúe si su padre termina (terminación en cascada)

Matar procesos:

- En ocasiones sucede que se necesita finalizar la ejecución de un determinado proceso por alguna razón: el proceso está consumiendo demasiado tiempo del procesador, el proceso está bloqueado, el proceso no genera información, el proceso genera demasiada información, etc.
- Para matar un proceso utilizaremos la orden kill, que resulta muy útil en el caso de procesos que no tienen asociada ninguna terminal de control.
- Para matar un proceso necesitaremos su PID, por lo que la sintaxis de la orden kill es:

```
kill PID(s)
```

11.- CREACIÓN DE PROCESOS EN C#.

Debes conocer

C# dispone en el paquete System.Diagnostics de la clase Process ya la gestión (iniciar, matar, ...) de los procesos. Para definir la información de inicio de los procesos se dispone de la clase ProcessStratInfo. Entre las propiedades de esta clase se encuentran los siguientes:

- **arguments:** permite pasar parámetros al proceso
- **createNoWindow**
- **useShellExecute:** permite indicar si se va a ejecutar el proceso en un shell
- **fileName:** indica la ruta al ejecutable. Si el ejecutable ha sido añadido a la varible de entorno path, no hace falta indicar la ruta completa.

A la hora de trabajar con procesos existen diferentes excepciones que pueden aparecer. A diferencia de otros lenguajes de programación en C# no hay obligación de utilizar bloques try catch con llamadas a funciones o métodos que puedan devolver una excepción. El compilador permite que no se traten y en caso de no tenerlas en cuenta, es equivalente a lanzar a un nivel superior. Por lo tanto, las excepciones llegarían hasta el usuario final.

Para la comunicación entre diferentes procesos existen varias técnicas entre las que se encuentran los pipes anónimos, pipes con nombre, WCF o sockets.

En el siguiente ejemplo de C# se muestra cómo lanzar un proceso:

```
using System;
using System.Diagnostics;
using System.ComponentModel;

namespace MyProcessSample
{
    class MyProcess
    {
        public static void Main()
        {
            try
            {
                using (Process myProcess = new Process())
                {
                    myProcess.StartInfo.UseShellExecute = false;
                    myProcess.StartInfo.FileName = "C:\\HelloWorld.exe";
                    myProcess.StartInfo.CreateNoWindow = true;
                    myProcess.Start();
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
```

```
}
```

12.- PROGRAMACIÓN CONCURRENTE.

Una sencilla definición de este concepto es la siguiente: *"Existencia simultánea de varios procesos en ejecución"*.

12.1.- PROGRAMA Y PROCESO.

Proceso: programa en ejecución.

Programa: conjunto de instrucciones que se aplican a un conjunto de datos de entrada para obtener una salida.

Mientras que los procesos son algo **activo** (cuentan con una serie de recursos asociados), los programas se puede decir que son algo **pasivo** (hay que ejecutarlos para que hagan algo).

Un programa con 3 instancias ejecutándose

Autor *(link: https://www.google.es)* (Licencia) *(link: https://www.google.es)* Procedencia *(link: https://www.google.es)*

Pero cuando un programa se pone en ejecución, puede dar lugar a más de un proceso, cada uno de ellos ejecutando una parte del programa. Por ejemplo el programa asociado a un navegador web; por un lado, está controlando las acciones del usuario con la interfaz y, por otro lado, hace las peticiones al servidor web. Por lo tanto, cada vez que se ejecuta este programa crea 2 procesos.

En la siguiente figura se puede apreciar que hay un programa almacenado en el disco y 3 instancias del mismo ejecutándose; por ejemplo, por 3 usuarios diferentes. Cada instancia del programa es un proceso; por tanto, existen 3 procesos independientes ejecutándose al mismo tiempo sobre el sistema operativo. En este caso tenemos 3 procesos concurrentes.

Un programa dando lugar a más de un proceso

Autor *(link: https://www.google.es)* (Licencia) *(link: https://www.google.es)* Procedencia *(link: https://www.google.es)*

Supongamos ahora que al ejecutarse el programa anterior da lugar a dos procesos, cada uno de ellos ejecutando una parte del programa. Entonces, la figura anterior se convierte en otra diferente. Como un programa puede estar compuesto por diversos procesos, una definición más acertada de proceso es la de una "*actividad asíncrona susceptible de ser asignada a un procesador*".

Cuando varios procesos se ejecutan concurrentemente, puede haber procesos que colaboren para un determinado fin (por ejemplo, P1.1 y P1.2) y otros que compitan por los recursos del sistema (por ejemplo, P2.1 y P3.1). Estas tareas de colaboración y competencia por recursos exigen mecanismos de comunicación y sincronización entre procesos.

12.2.- CARACTERÍSTICAS DE LA PROGRAMACIÓN CONCURRENTE.

Debes conocer

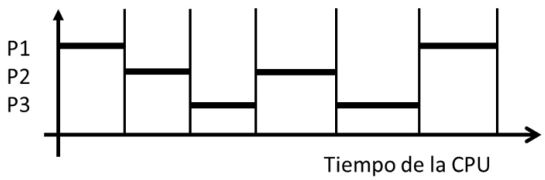
La **programación concurrente** es la disciplina que se encarga del estudio de las notaciones que permiten especificar la ejecución concurrente de las acciones de un programa, así como las técnicas para resolver los problemas inherentes a la ejecución concurrente (comunicación y sincronización).

Los **beneficios** que aporta la programación se pueden resumir en:

- **Mejor aprovechamiento de la CPU:** un proceso puede aprovechar ciclos de CPU mientras otro realiza una operación de entrada/salida.
- **Velocidad de ejecución:** al subdividir un programa en procesos, éstos se pueden "repartir" entre procesadores o gestionar en un único procesador, según la prioridad de los procesos.
- **Solución a problemas de naturaleza concurrente:** existen algunos problemas cuya solución es más fácil utilizando este tipo de programación. Por ejemplo:
  - Sistemas de control: son sistemas en los que hay captura de datos, normalmente a través de sensores, análisis y actuación en función del análisis. Un ejemplo son los sistemas de tiempo real.
  - Tecnologías web: los servidores web son capaces de atender múltiples peticiones de usuarios concurrentemente, también los servidores de chat, correo, los propios navegadores web, etc.
  - Aplicaciones basadas en GUI: el usuario puede interactuar con la aplicación mientras la aplicación está realizando otra tarea. Por ejemplo: el navegador web puede estar descargando un archivo mientras el usuario navega por las páginas.
  - Simulación: programas que modelan sistemas físicos con autonomía.
  - Sistemas Gestores de Bases de Datos: los usuarios interactúan con el sistema, cada usuario puede ser visto como un proceso.

Concurrencia y hardware

Concurrencia



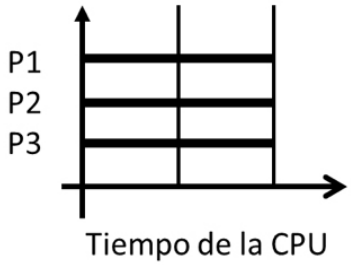
Autor *(link: https://www.google.es)* (Licencia) *(link: https://www.google.es)* Procedencia *(link: https://www.google.es)*

En un sistema monoprocesador (de un solo procesador), se puede tener una ejecución concurrente gestionando el tiempo de procesador para cada proceso. El sistema operativo va alternando el tiempo entre los distintos procesos. Cuando un proceso necesita realizar una operación de entrada/salida, lo abandona y otro lo ocupa. De esta forma se aprovechan los ciclos del procesador. En la siguiente figura se muestra cómo el tiempo de procesador está repartido entre 3 procesos. En cada momento sólo hay un proceso. Esta forma de gestionar los procesos en un sistema monoprocesador recibe el nombre de multiprogramación.

Sistemas monoprocesador multiprocesador

- En un sistema **monoprocesador** todos los procesos comparten la misma memoria. La forma de comunicar y sincronizar procesos se realiza mediante variables compartidas.
- En un sistema **multiprocesador** (existe más de un procesador), podemos tener un proceso en cada procesador. Esto permite que exista paralelismo real entre los procesos (ver la siguiente figura). Estos pueden ser de memoria compartida (fuertemente acoplados) o con memoria local a cada procesador (débilmente acoplados). Se denomina multiproceso a la gestión de varios procesos dentro de un sistema multiprocesador, donde cada procesador puede acceder a una memoria común.

Paralelismo



Autor *(link: https://www.google.es)* (Licencia) *(link: https://www.google.es)* Procedencia *(link: https://www.google.es)*

12.3.- PROGRAMAS CONCURRENTES.

Un programa concurrente define un conjunto de acciones que pueden ser ejecutadas simultáneamente.

Supongamos que tenemos estas dos instrucciones en un programa. Está claro que el orden de ejecución de las mismas influirá en el resultado final:

```
x=x+1;          La primera instrucción se debe
y=x+1;          ejecutar antes de la segunda
```

En cambio, si tenemos estas otras, el orden de ejecución es indiferente:

```
x=1;            El orden no interviene
y=2;            en el resultado final
z=3;
```

12.4.- PROBLEMAS INHERENTES A LA PROGRAMACIÓN CONCURRENTE.

A la hora de crear un programa concurrente podemos encontrarnos con los siguientes problemas:

- **Exclusión mutua:** consiste en que varios procesos acceden a la vez a una variable compartida para actualizarla. Esto debe evitarse, ya que produce inconsistencia de datos: un proceso estar actualizando la variable, a la vez que otro proceso puede estar leyéndola. Por ello es necesario conseguir la exclusión mutua de los procesos respecto a la variable compartida.
- **Condición de sincronización:** hace referencia a la necesidad de coordinar los procesos con el fin de sincronizar sus actividades. Puede ocurrir que un proceso P1 llegue a un estado X que no puede continuar su ejecución hasta que otro proceso P2 haya llegado a un estado Y de su ejecución. La programación concurrente proporciona mecanismos para bloquear los procesos a la espera de que ocurra un evento y para desbloquearlos cuando esto ocurra.

Algunas herramientas para manejar la concurrencia son: la región crítica, los semáforos, región crítica condicional, buzones, sucesos, monitores y sincronización por rendez-vous.

12.5.- PROGRAMACIÓN CONCURRENTE EN C#.

En C# la programación concurrente se resuelve mediante el concepto de los hilos y tareas, los cuales son como una secuencia de control dentro de un proceso, que ejecuta sus instrucciones de forma independiente.

Entre procesos e hilos existen ciertas diferencias:

- Los hilos comparten el espacio de memoria del usuario, muchos comparten datos y espacios de direcciones. Sin embargo, los procesos generalmente poseen espacios de memoria independientes e interactúan a través de mecanismos de comunicación dados por el sistema.
- Hilos y procesos pueden encontrarse en diferentes estados, pero los cambios de estado en los procesos son más costosos.

Para programar concurrentemente podemos dividir nuestro programa en hilos o tareas. C# proporciona la construcción de programas concurrentes mediante la clase Thread o la clase Task. La principal diferencia entre las tareas y los hilos es que las tareas se ejecutan de forma asíncrona por defecto en un pool de hilos. Es decir, la gestión de la tarea no está en manos de programador. Además, las tares se pueden sincronizar fácilmente con las palabras reservadas async y await. Aunque, si es necesario un rendimiento elevado, el uso de hilos se debe tener en cuenta.

13.- PROGRAMACIÓN PARALELA Y DISTRIBUIDA.

13.1.- PROGRAMACIÓN PARALELA.

Debes conocer

Un **programa paralelo** es un tipo de programa concurrente diseñado para ejecutarse en un sistema multiprocesador. El procesamiento paralelo permite que muchos elementos de proceso independientes trabajen simultáneamente para resolver un problema. El problema a resolver se divide en partes independientes, de tal forma que cada elemento pueda ejecutar la parte de programa que le corresponda a la vez que los demás.

Recordemos que en un sistema multiprocesador, donde existe más de un procesador, podemos tener un proceso en cada procesador y todos juntos trabajan para resolver un problema. Cada procesador realiza una parte del problema y necesita intercambiar información con el resto. Según cómo se realice este intercambio podemos tener **modelos distintos de programación paralela:**

- **Modelo de memoria compartida:** los procesadores comparten físicamente la memoria, es decir, todos acceden al mismo espacio de direcciones. Un valor escrito en memoria por un procesador puede ser leído directamente por cualquier otro.
- **Modelo de paso de mensajes:** cada procesador dispone de su propia memoria, la cual sólo es accesible por él. Para hacer el intercambio de información es necesario que cada procesador realice la petición de datos al procesador que los tiene y éste haga el envío.

Las **ventajas** más importantes del procesamiento paralelo son las siguientes:

- Proporciona ejecución simultánea de tareas.
- Disminuye el tiempo total de ejecución de una aplicación.
- Resolución de problemas complejos y de grandes dimensiones.
- Utilización de recursos no locales, por ejemplo, los recursos que están en una red distribuida, una WAN o la propia red internet.
- Disminución de costos, en vez de gastar en un supercomputador muy caro, se pueden utilizar otros recursos más baratos disponibles remotamente.

Algunas de las desventajas del procesamiento paralelo son las siguientes:

- Los compiladores y entornos de programación para sistemas paralelos son más difíciles de desarrollar.
- Los programas paralelos son más difíciles de escribir.
- El consumo de energía de los elementos que forman el sistema.
- Mayor complejidad en el acceso a los datos.
- La comunicación y la sincronización entre las diferentes subtareas.

La computación paralela resuelve problemas tales como: predicciones y estudios metereológicos, estudio del genoma humano, modelado de la biosfera, predicciones sísmicas, simulación de moléculas, etc.

13.2.- PROGRAMACIÓN DISTRIBUIDA.

Uno de los motivos principales para construir un sistema distribuido es compartir recursos. Probablemente, el sistema distribuido más conocido por todos es Internet que permite a los usuarios, donde quiera que estén, hacen uso de la World Wide Web, el correo electrónico y la transferencia de ficheros. Entre las aplicaciones más recientes de la computación distribuida se encuentra el **Cloud Computing**, que es la computación en la nube o servicios en la nube, que ofrece servicios de computación a través de Internet.

Un **sistema distribuido** se define como aquel en el que los componentes hardware o software, localizados en computadores unidos mediante una red, comunican y coordinan sus acciones mediante el paso de mensajes. Esta definición tiene las siguientes consecuencias:

- **Concurrencia:** lo normal en una red de ordenadores es la ejecución de programas concurrentes.
- **Inexistencia de reloj global:** cuando los programas necesitan coordinar sus acciones mediante el paso de mensajes.
- **Fallos independientes:** cada componente del sistema puede fallar independientemente, permitiendo que los demás continúen su ejecución.

La programación distribuida es un paradigma de programación enfocado a desarrollar sistemas distribuidos, abiertos, escalables, transparentes y tolerantes a fallos. Este paradigma es el resultado natural del uso de las computadoras y las redes. Casi cualquier lenguaje de programación que tenga acceso al máximo al hardware del sistema, puede manejar la programación distribuida, considerando una buena cantidad de tiempo y de código.

Una arquitectura típica para el desarrollo de sistemas distribuidos es la **arquitectura cliente-servidor**. Los clientes son elementos activos que demandan servicios a los servidores, realizando peticiones y esperando la respuesta. Los servidores son elementos pasivos que realizan las tareas bajo requerimientos de los clientes.

Existen varios modelos de programación para la comunicación entre los procesos de un sistema distribuido:

- **Sockets:** proporcionan los puntos extremos para la comunicación entre procesos. Es actualmente la base de la comunicación. Pero al ser de muy bajo nivel de abstracción, no son adecuados a nivel de aplicación.
- **Llamada de procedimientos remotos o RPC (Remote Procedure Call):** permite a un programa cliente llamar a un procedimiento de otro programa en ejecución en un proceso servidor. El proceso servidor define en su interfaz de usuario los procedimientos disponibles para ser llamados remotamente.
- **Invocación remota de objetos:** el modelo de programación basado en objetos ha sido extendido para permitir que los objetos de diferentes procesos se comuniquen uno con otro por medio de una invocación a un método remoto.

Las **ventajas** que aportan los sistemas distribuidos son las siguientes:

- Se pueden compartir recursos y datos.
- Capacidad de crecimiento incremental.
- Mayor flexibilidad al poderse distribuir la carga de trabajo entre diferentes ordenadores.
- Alta disponibilidad.
- Soporte de aplicaciones inherentemente distribuidas.
- Carácter abierto y heterogéneo.

Algunas de las **desventajas** de este tipo de sistemas son las siguientes:

- Aumento de la complejidad, se necesita nuevo tipo de software
- Problemas con las redes de comunicación: pérdida de mensajes, saturación del tráfico.
- Problemas de seguridad, como por ejemplo, ataques de denegación de servicio en la que se “bombardea” un servicio con peticiones inútiles de forma que un usuario interesado en usar el servicio no pueda usarlo.