

## 1.1. INTRODUCCIÓN

Todos los ordenadores actuales realizan varias tareas a la vez, por ejemplo ejecutar un programa de procesador de textos, leer información de un disco duro, imprimir un documento por la impresora, visualizar información en pantalla, etc... Cuando un programa se carga en la memoria para su ejecución se convierte en un proceso.

En un sistema operativo **multiproceso** o **multitarea** se puede ejecutar más de un proceso (programa) a la vez, dando la sensación al usuario de que cada proceso es el único que se está ejecutando. La única forma de ejecutar varios procesos simultáneamente es tener varias CPUs (ya sea en una máquina o en varias). En los sistemas operativos con una única CPU se va alternando la ejecución de los procesos, es decir, se quita un proceso de la CPU, se ejecuta otro y se vuelve a colocar el primero sin que se entere de nada; esta operación se realiza tan rápido que parece que cada proceso tiene dedicación exclusiva.

La **programación multiproceso** tiene en cuenta la posibilidad de que múltiples procesos puedan estar ejecutándose simultáneamente sobre el mismo código de programa. Es decir desde una misma aplicación podemos realizar varias tareas de forma simultánea, o lo que es lo mismo, podemos dividir un proceso en varios subprocesos. En este capítulo aprenderemos a ejecutar varios procesos simultáneamente.

## 1.2. PROCESOS Y SISTEMA OPERATIVO

Se puede definir un **proceso** como un programa en ejecución. Consiste básicamente en el código ejecutable del programa, los datos y la pila del programa, el contador de programa, el puntero de pila y otros registros, y toda la información necesaria para ejecutar el programa.

Todos los programas que se ejecutan en el ordenador se organizan como un conjunto de procesos. El sistema operativo decide parar la ejecución de un proceso, por ejemplo, porque ha consumido su tiempo de CPU, y arrancar la de otro. Cuando se suspende temporalmente la ejecución de un proceso debe reanudarse posteriormente en el mismo estado en que encontraba cuando se paró, esto implica que toda la información referente al proceso debe almacenarse en alguna parte.

El **BCP** es una estructura de datos llamada *Bloque de Control de Proceso* donde se almacena información acerca de un proceso:

- **Identificación del proceso.** Cada proceso que se inicia es referenciado por un identificador único.
- **Estado del proceso.**
- **Contador de programa.**
- **Registros de CPU.**
- **Información de planificación de CPU** como la prioridad del proceso.
- **Información de gestión de memoria.**
- **Información contable** como la cantidad de tiempo de CPU y tiempo real consumido.
- **Información de estado de E/S** como la lista de dispositivos asignados, archivos abiertos, etc.

Mediante el comando **ps** (*process status*) de Linux podemos ver parte de la información asociada a cada proceso. El siguiente ejemplo muestra los procesos actualmente vivos en la máquina, se muestran 2 procesos ejecutándose, uno es el shell y el otro es la ejecución de la orden ps:

```
mj@ubuntu-mj:~$ ps
  PID TTY          TIME CMD
 1906 pts/0        00:00:00 bash
 2362 pts/0        00:00:00 ps
```

**PID:** identificador del proceso.

**TTY:** terminal asociado del que lee y al que escribe. Si no hay aparece interrogación.

**TIME:** tiempo de ejecución asociado, es la cantidad total de tiempo de CPU que el proceso ha utilizado desde que nació.

**CMD:** nombre del proceso.

La orden **ps -f** muestra más información:

```
mj@ubuntu-mj:~$ ps -f
  UID          PID   PPID  C  STIME TTY          TIME CMD
  mj           1906   1298  0  18:06 pts/0        00:00:00 bash
  mj           2363   1906  0  18:33 pts/0        00:00:00 ps -f
```

**UID:** nombre de usuario

**PPID:** PID del padre de cada proceso.

**C:** porcentaje de recursos de CPU utilizado por el proceso.

**STIME:** hora de inicio del proceso.

La orden **ps -AF** muestra todos los procesos activos con todos los detalles; como en los ejemplos anteriores se puede observar que la última línea que aparece es la del comando que se está ejecutando:

```
mj@ubuntu-mj:~$ ps -AF
  UID          PID   PPID  C    SZ    RSS  PSR  STIME TTY          TIME CMD
  root           1      0  0   703  1524    0  18:05 ?           00:00:00 /sbin/init
  root           2      0  0     0     0    0  18:05 ?           00:00:00 [kthreadd]
  root           3      2  0     0     0    0  18:05 ?           00:00:00 [migration/0]
  root           4      2  0     0     0    0  18:05 ?           00:00:00 [ksoftirqd/0]
  root           5      2  0     0     0    0  18:05 ?           00:00:00 [watchdog/0]
  root           6      2  0     0     0    0  18:05 ?           00:00:00 [events/0]
  root           7      2  0     0     0    0  18:05 ?           00:00:00 [cpuset]
  root           8      2  0     0     0    0  18:05 ?           00:00:00 [khelper]
  root           9      2  0     0     0    0  18:05 ?           00:00:00 [async/mgr]
  . . . . .
  postfix       1782    968  0  1457  1348    0  18:06 ?           00:00:00 pickup -l -t fi
  root          1786   1510  0   904  1080    0  18:06 ?           00:00:00 hald-addon-stor
  nobody        1846   1324  0  13116 2776    0  18:06 ?           00:00:00 /opt/lampp/bin/
  mj            1847     1  0 10495 9648    0  18:06 ?           00:00:02 /usr/lib/gnome-
  nobody        1849   1324  0 13048 2744    0  18:06 ?           00:00:00 /opt/lampp/bin/
  . . . . .
  mj            2497   2493  0   405   440    0  18:34 ?           00:00:00 heart -pid 2493
  mj            2557   2379  0  4263 10404    0  18:34 ?           00:00:01 /usr/bin/python
  root          2698    324  0   664   424    0  18:39 ?           00:00:00 udevd --daemon
  root          2699    324  0   664   448    0  18:39 ?           00:00:00 udevd --daemon
  mj            2720   1906  0   679  1044    0  18:49 pts/0        00:00:00 ps -AF
```

**C:** porcentaje de CPU utilizado por el proceso.

**SZ:** tamaño virtual de la imagen del proceso.



**RSS:** tamaño de la parte residente en memoria en kilobytes.

**PSR:** procesador que el proceso tiene actualmente asignado.

En Ubuntu escribiendo desde la terminal **sudo gnome-system-monitor** podemos acceder a la interfaz gráfica que nos muestra información sobre los procesos que se están ejecutando, véase Figura 1.1.

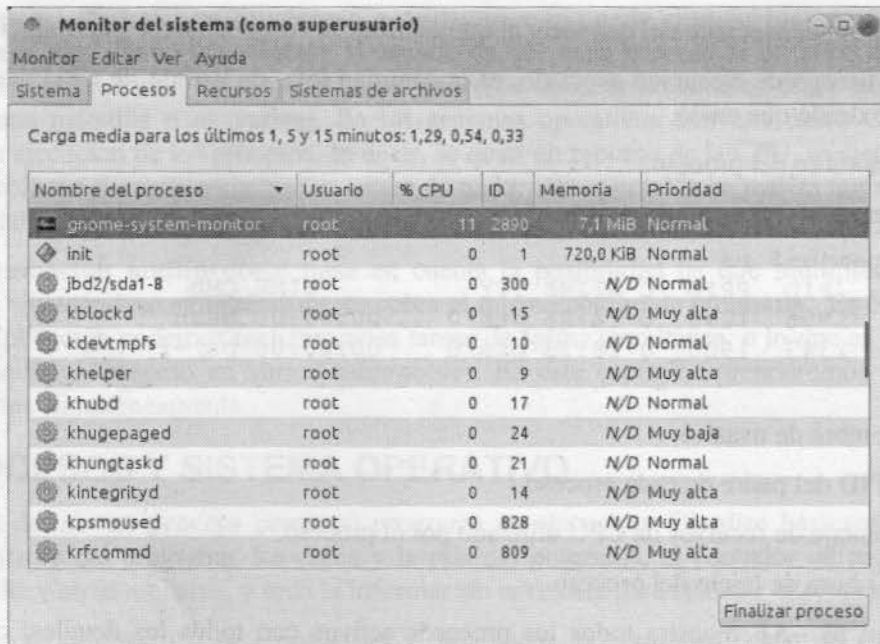


Figura 1.1. Monitor del sistema en Ubuntu.

En sistemas operativos Windows podemos usar desde la línea de comandos del DOS la orden **tasklist** para ver los procesos que se están ejecutando:

```
D:\>tasklist
```

Nombre de imagen	PID	Nombre de sesión	Núm. de	Uso de memor
System Idle Process	0	Console	0	28 KB
System	4	Console	0	340 KB
smss.exe	1528	Console	0	436 KB
csrss.exe	1640	Console	0	8.052 KB
winlogon.exe	1664	Console	0	1.736 KB
services.exe	1708	Console	0	3.676 KB
lsass.exe	1720	Console	0	1.412 KB
nvsvc32.exe	1900	Console	0	5.488 KB
svchost.exe	1956	Console	0	5.536 KB
svchost.exe	2008	Console	0	4.736 KB
svchost.exe	1624	Console	0	31.096 KB
svchost.exe	248	Console	0	4.036 KB
svchost.exe	956	Console	0	4.056 KB
spoolsv.exe	1276	Console	0	6.932 KB
explorer.exe	1540	Console	0	43.888 KB
RTHDCPL.exe	1600	Console	0	23.004 KB
rundll32.exe	1636	Console	0	4.364 KB

La siguiente orden muestra los servicios que se están ejecutando bajo el proceso **svchost.exe**:

```
D:\>tasklist /svc /fi "imagenname eq svchost.exe"
```

Nombre de imagen	PID	Servicios
svchost.exe	1956	DcomLaunch, TermService
svchost.exe	2008	RpcSs
svchost.exe	1624	AudioSrv, Browser, CryptSvc, Dhcp, dmserver, EventSystem, FastUserSwitchingCompatibility, helpsvc, LanmanServer, lanmanworkstation, Netman, Nla, RasMan, Schedule, seclogon, SENS, SharedAccess, ShellHWDetection, TapiSrv, Themes, TrkWks, W32Time, winmgmt, wuauserv, WZCSVC
svchost.exe	248	Dnscache
svchost.exe	956	LmHosts, SSDPSRV
svchost.exe	1608	WebClient
svchost.exe	2872	stisvc
svchost.exe	6108	HTTPFilter

Aunque lo más típico es usar la combinación de teclas [CTRL+ Alt + Supr] para que se muestre la pantalla que da acceso al *Administrador de tareas de Windows*, véase Figura 1.2.

Nombre	10% CPU	40% Memoria	10% Disco	0% Red
<b>Aplicaciones (3)</b>				
Explorador de Windows (3)	0,1%	80,1 MB	0 MB/s	0 Mbps
Firefox (32 bits)	0,3%	468,4 MB	0 MB/s	0,1 Mbps
Task Manager	0,2%	12,8 MB	0 MB/s	0 Mbps
<b>Procesos en segundo plano (70)</b>				
Aislamiento de gráficos de dispositivo de audio ...	0%	4,9 MB	0 MB/s	0 Mbps
Aplicación de subsistema de cola	0%	4,4 MB	0 MB/s	0 Mbps
Application Frame Host	0%	7,0 MB	0 MB/s	0 Mbps
COM Surrogate	0%	1,2 MB	0 MB/s	0 Mbps
Configuración	0%	0,4 MB	0 MB/s	0 Mbps
ControlCenter Main Process (32 bits)	0%	0,6 MB	0 MB/s	0 Mbps
ControlCenter UX System (32 bits)	0%	0,7 MB	0 MB/s	0 Mbps

Figura 1.2. Administrador de tareas de Windows.

### 1.2.1. ESTADOS DE UN PROCESO

Un proceso, aunque es una entidad independiente, puede generar una salida que se use como entrada para otro proceso. Entonces este segundo proceso tendrá que esperar a que el primero termine para obtener los datos a procesar, en este caso debe bloquearse hasta que sus datos de entrada estén disponibles. Un proceso también se puede parar porque el sistema operativo decida asignar el procesador a otro proceso. En definitiva, los estados en los que se puede encontrar un proceso son los siguientes:

- **En ejecución:** el proceso está actualmente ejecutándose, es decir, usando el procesador.
- **Bloqueado:** el proceso no puede hacer nada hasta que no ocurra un evento externo, como por ejemplo la finalización de una operación de E/S.



- **Listo:** el proceso está parado temporalmente y listo para ejecutarse cuando se le de oportunidad.

La Figura 1.3 muestra mediante un diagrama de estados los estados en que se puede encontrar un proceso.



Figura 1.3. Estados de un proceso.

Las transiciones entre los estados son las siguientes:

- **En ejecución - Bloqueado:** un proceso pasa de ejecución a bloqueado cuando espera la ocurrencia de un evento externo.
- **Bloqueado - Listo:** un proceso pasa de bloqueado a listo cuando ocurre el evento externo que se esperaba.
- **Listo - En ejecución:** un proceso pasa de listo a ejecución cuando el sistema le otorga un tiempo de CPU.
- **En ejecución - Listo:** un proceso pasa de ejecución a listo cuando se le acaba el tiempo asignado por el sistema operativo.

### 1.3. PROGRAMACIÓN CONCURRENTE

El diccionario *WordReference.com* (<http://www.wordreference.com/definicion/>) nos muestra varias acepciones de la palabra concurrencia. Nos quedamos con la tercera: "*Acaecimiento o concurso de varios sucesos en un mismo tiempo*". Si sustituimos sucesos por **procesos** ya tenemos una aproximación de lo que es la concurrencia en informática: la existencia simultánea de varios procesos en ejecución.

#### 1.3.1. PROGRAMA Y PROCESO

Al principio del tema se definió un **proceso** como un programa en ejecución. Y ¿qué es un programa?, podemos definir **programa** como un conjunto de instrucciones que se aplican a un conjunto de datos de entrada para obtener una salida. Un proceso es algo activo que cuenta con una serie de recursos asociados, en cambio un programa es algo pasivo, para que pueda hacer algo hay que ejecutarlo.



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

En la Figura 1.12 existe un programa almacenado en 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, tenemos entonces 3 procesos concurrentes.

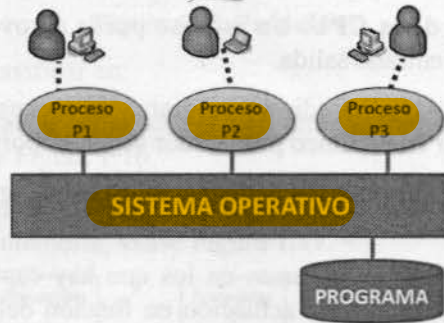


Figura 1.12. Un programa con 3 instancias ejecutándose.

Dos procesos serán concurrentes cuando la primera instrucción de uno de ellos se ejecuta después de la primera instrucción del otro y antes de la última. Es decir, existe un solapamiento o intercalado en la ejecución de sus instrucciones. No hay que confundir el solapamiento con la ejecución simultánea de las instrucciones, en este caso estaríamos en una situación de programación paralela, aunque a veces el hardware subyacente (más de un procesador) sí permitirá la ejecución simultánea.

Supongamos ahora que el programa anterior al ejecutarse da lugar a 2 procesos más, cada uno ejecutando una parte del programa, entonces la Figura 1.12 se convierte en la 1.13. Ya que 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<sup>1</sup>.

Cuando 2 programas son concurrentes, uno se inicia y DESPUES inicia el otro pero siempre dentro del tiempo de ejecución del anterior (es decir entre q inicia y termina este)

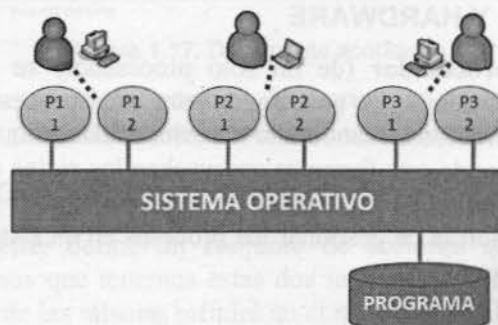


Figura 1.13. Un programa dando lugar a más de un proceso.

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 los recursos exigen mecanismos de comunicación y sincronización entre procesos.

<sup>1</sup> Programación concurrente. José Tomás Palma Méndez y otros. Ed Paraninfo. ISBN: 9788497321846


### 1.3.2. CARACTERÍSTICAS

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).

#### BENEFICIOS

La programación concurrente aporta una serie de beneficios:

**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 importancia. 

**Solución a problemas de naturaleza concurrente.** Existen algunos problemas cuya solución es más fácil utilizando esta metodología:

- **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

En un sistema **monoprocesador** (de un solo procesador) se puede tener una ejecución concurrente gestionando el tiempo de procesador para cada proceso. El S.O. va alternando el tiempo entre los distintos procesos, cuando uno necesita realizar una operación de entrada salida, lo abandona y otro lo ocupa; de esta forma se aprovechan los ciclos del procesador. En la Figura 1.14 se muestra como el tiempo de procesador es 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**.

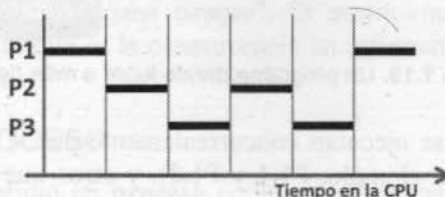


Figura 1.14. Concurrencia.

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, véase Figura 1.15.

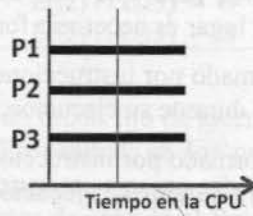


Figura 1.15. Paralelismo.

Estos sistemas se pueden clasificar en:

- **Fuertemente acoplados:** cuando poseen una memoria compartida por todos los procesadores, véase Figura 1.16.
- **Débilmente acoplados:** cuando los procesadores poseen memorias locales y no existe la compartición de memoria, véase Figura 1.17.

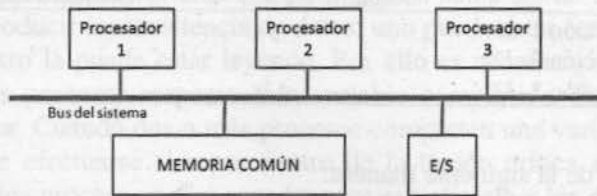


Figura 1.16. Fuertemente acoplados.

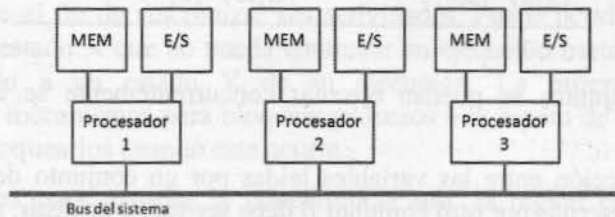


Figura 1.17. Débilmente acoplados.

Se denomina **multiproceso** a la gestión de varios procesos dentro de un sistema multiprocesador, donde cada procesador puede acceder a una memoria común.

### 1.3.3. 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 la ejecución de las mismas influirá en el resultado final:

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

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

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

## CONDICIONES DE BERNSTEIN

Bernstein definió unas **condiciones** para que dos conjuntos de instrucciones se puedan ejecutar concurrentemente. En primer lugar es necesario formar 2 conjuntos de instrucciones:

- **Conjunto de lectura:** formado por instrucciones que cuentan con variables a las que se accede en modo lectura durante su ejecución.
- **Conjunto de escritura:** formado por instrucciones que cuenta con variables a las que se accede en modo escritura durante su ejecución.

Por ejemplo, sean las siguientes instrucciones:

Instrucción 1:	$x := y+1$
Instrucción 2:	$y := x+2$
Instrucción 3:	$z := a+b$

Los conjuntos de lectura y escritura estarían formados por las variables siguientes:

	Conjunto lectura - L	Conjunto escritura - E
Instrucción 1- I1:	y	x
Instrucción 2- I2:	x	y
Instrucción 3- I3:	a,b	z

Se pueden expresar de la siguiente manera:

$L(I1)=\{y\}$	$E(I1)=\{x\}$
$L(I2)=\{x\}$	$E(I2)=\{y\}$
$L(I3)=\{a,b\}$	$E(I3)=\{z\}$

Para que dos conjuntos se puedan ejecutar concurrentemente se deben cumplir estas 3 condiciones:

- La intersección entre las variables leídas por un conjunto de instrucciones  $I_i$  y las variables escritas por otro conjunto  $I_j$  debe ser vacío, es decir, **no debe haber variables comunes**:

$$L(I_i) \cap E(I_j) = \emptyset$$

- La intersección entre las variables de escritura de un conjunto de instrucciones  $I_i$  y las variables leídas por otro conjunto  $I_j$  debe ser nulo, es decir, **no debe haber variables comunes**:

$$E(I_i) \cap L(I_j) = \emptyset$$

- Por último, la intersección entre las variables de escritura de un conjunto de instrucciones  $I_i$  y las variables de escritura de un conjunto  $I_j$  debe ser vacío, **no debe haber variables comunes**:

$$E(I_i) \cap E(I_j) = \emptyset$$

En el ejemplo anterior tenemos las siguientes condiciones, donde se observa que las instrucciones I1 e I2 no se pueden ejecutar concurrentemente porque no cumplen las 3 condiciones:



Conjunto I1 e I2	Conjunto I2 e I3	Conjunto I1 e I3
$L(I1) \cap E(I2) \neq \emptyset$	$L(I2) \cap E(I3) = \emptyset$	$L(I1) \cap E(I3) = \emptyset$
$E(I1) \cap L(I2) \neq \emptyset$	$E(I2) \cap L(I3) = \emptyset$	$E(I1) \cap L(I3) = \emptyset$
$E(I1) \cap E(I2) = \emptyset$	$E(I2) \cap E(I3) = \emptyset$	$E(I1) \cap E(I3) = \emptyset$

En los programas secuenciales hay un orden fijo de ejecución de las instrucciones, siempre se sabe por dónde va a ir el programa. En cambio, en los programas concurrentes hay un orden parcial. Al haber solapamiento de instrucciones no se sabe cuál va a ser el orden de ejecución, puede ocurrir que ante unos mismos datos de entrada el flujo de ejecución no sea el mismo. Esto da lugar a que los programas concurrentes tengan un comportamiento indeterminista donde repetidas ejecuciones sobre un mismo conjunto de datos puedan dar diferentes resultados.

### 1.3.4. PROBLEMAS INHERENTES A LA PROGRAMACIÓN CONCURRENTES

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

- **Exclusión mutua.** En programación concurrente es muy típico que varios procesos accedan a la vez a una variable compartida para actualizarla. Esto se debe evitar, ya que puede producir inconsistencia de datos: uno puede estar actualizando la variable a la vez que otro la puede estar leyendo. Por ello es necesario conseguir la exclusión mutua de los procesos respecto a la variable compartida. Para ello se propuso la **región crítica**. Cuando dos o más procesos comparten una variable, el acceso a dicha variable debe efectuarse siempre dentro de la región crítica asociada a la variable. Sólo uno de los procesos podrá acceder para actualizarla y los demás deberán esperar, el tiempo de estancia es finito.
- **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 pueda continuar su ejecución hasta que otro proceso P2 haya llegado a un estado Y de su ejecución. La programación concurrente proporciona mecanismos para bloquear procesos a la espera de que ocurra un evento y para desbloquearlos cuando este ocurra.

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

#### ACTIVIDAD 1.9

Responde a las siguientes cuestiones:

Escribe alguna característica de un programa concurrente.

¿Cuál es la ventaja de la concurrencia en los sistemas monoprocesador?

¿Cuáles son las diferencias entre multiprogramación y multiproceso?

¿Cuáles son los dos problemas principales inherentes a la programación concurrente?

## 1.4. PROGRAMACIÓN PARALELA Y DISTRIBUIDA

### 1.4.1. PROGRAMACIÓN PARALELA

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. Estos elementos pueden ser un número arbitrario de equipos conectados por una red, un único equipo con varios procesadores o una combinación de ambos. 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 independiente del resto y accesible sólo por él. Para realizar el intercambio de información es necesario que cada procesador realice la petición de datos al procesador que los tiene, y éste haga el envío. El entorno de programación PVM que veremos más adelante utiliza este modelo.

El intercambio de información entre procesadores depende del sistema de almacenamiento que se disponga. Según este criterio las arquitecturas paralelas se clasifican en: **Sistemas de memoria compartida o multiprocesadores:** los procesadores comparten físicamente la memoria; y **Sistemas de memoria distribuida o multicomputadores:** cada procesador dispone de su propia memoria.

Dentro de los sistemas de memoria distribuida o multicomputadores nos encontramos con los **Clusters**. Son sistemas de procesamiento paralelo y distribuido donde se utilizan múltiples ordenadores, cada uno con su propio procesador, enlazados por una red de interconexión más o menos rápida, de tal forma que el conjunto de ordenadores es visto como un único ordenador, más potente que los comunes de escritorio.

Tradicionalmente, el paralelismo se ha utilizado en centros de supercomputación para resolver problemas de elevado coste computacional en un tiempo razonable, pero en la última década su interés se ha extendido por la difusión de los procesadores con múltiples núcleos (combina dos o más procesadores independientes en un solo circuito integrado). Estos procesadores permiten que un dispositivo computacional exhiba una cierta forma del paralelismo a nivel de thread (thread-level parallelism) (TLP) sin incluir múltiples microprocesadores en paquetes físicos separados. Esta forma de TLP se conoce a menudo como multiprocesamiento a nivel de chip (chip-level multiprocessing) o CMP<sup>2</sup>.

## VENTAJAS E INCONVENIENTES

### Ventajas del procesamiento paralelo:

- 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.

Pero no todo son ventajas, algunos inconvenientes son:

<sup>2</sup> [https://es.wikipedia.org/wiki/Procesador\\_multinúcleo](https://es.wikipedia.org/wiki/Procesador_multinúcleo)



- 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 como: predicciones y estudios meteorológicos, estudio del genoma humano, modelado de la biosfera, predicciones sísmicas, simulación de moléculas... En algunos casos se dispone de tal cantidad de datos que serían muy lento o imposible tratar con máquinas convencionales.

### ACTIVIDAD 1.10

Entra en la siguiente URL [https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/) y responde a las siguientes cuestiones:

Cita algunas características de la computación serie.

Cita algunas características de la computación en paralelo.

Ámbitos en los que se usa la computación en paralelo.

¿Cómo hace uso de la computación paralela el proyecto SETI @ home?

## 1.4.2. 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 hacer 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.

Se define un sistema distribuido 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<sup>3</sup>:

- **Concurrencia:** lo normal en una red de ordenadores es la ejecución de programas concurrentes.
- **Inexistencia de reloj global:** cuando los programas necesitan cooperar coordinan sus acciones mediante el paso de mensajes. No hay una temporalización, los relojes de los host no están sincronizados.
- **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 en 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

<sup>3</sup> Sistemas Distribuidos: Conceptos y Diseño. George Coulouris y otros. Ed: Addison-Wesley.



programación que tenga acceso al máximo al hardware del sistema puede manejar la programación distribuida, considerando una buena cantidad de tiempo y código<sup>4</sup>.

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.

Por ejemplo, un cliente web solicita una página, el servidor web envía al cliente la página solicitada. Véase Figura 1.19. La comunicación entre servidores y clientes se realiza a través de la red.

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. En el capítulo 3 se tratarán los sockets en Java.
- **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 servicio 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 o **RMI** (*Remote Method Invocation*). Un objeto que vive en un proceso puede invocar métodos de un objeto que reside en otro proceso. **Java RMI** extiende el modelo de objetos de Java para proporcionar soporte de objetos distribuidos en lenguaje Java.

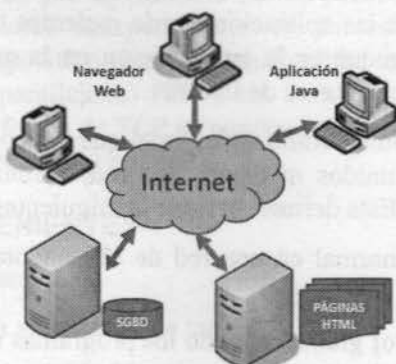


Figura 1.19. Cliente-servidor sobre web.

## VENTAJAS E INCONVENIENTES

Ventajas que aportan los sistemas distribuidos:

- Se pueden compartir recursos y datos.
- Capacidad de crecimiento incremental.

<sup>4</sup> [http://es.wikipedia.org/wiki/Programación\\_distribuida](http://es.wikipedia.org/wiki/Programación_distribuida)

- Mayor flexibilidad al poderse distribuir la carga de trabajo entre diferentes ordenadores.
- Alta disponibilidad.
- Soporte de aplicaciones inherentemente distribuidas.
- Carácter abierto y heterogéneo.

Pero no todo son ventajas, algunos inconvenientes son:

- 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.

### ACTIVIDAD 1.11

Busca en Internet aplicaciones de los sistemas distribuidos.

### PROGRAMACION CONCURRENTES, PARALELA Y DISTRIBUIDA

**Programación Concurrente:** Tenemos varios elementos de proceso (hilos, procesos) que trabajan de forma conjunta en la resolución de un problema. Se suele llevar a cabo en un único procesador o núcleo.

**Programación Paralela:** Es programación concurrente cuando se utiliza para acelerar la resolución de los problemas, normalmente usando varios procesadores o núcleos.

**Programación Distribuida:** Es programación paralela cuando los sistemas están distribuidos a través de una red (una red de procesadores); se usa paso de mensajes.

[Redacted content]