Programación multiproceso

Sumario

En un ordenador es posible ejecutar múltiples programas a la vez. El objetivo de este bloque es aprender a programar aplicaciones capaces de ejecutar varias tareas a la vez. Se diferenciará entre proceso, hilo, multiproceso, programación distribuida,…

1 Programas, ejecutables y servicios

Un **programa informático**, secuencia de instrucciones que una computadora puede interpretar y ejecutar.

Por ejemplo, un navegador web es un programa informático.

Los programas se distribuyen en forma de **ejecutables**, que serán archivos que el sistema operativo puede interpretar como un programa cuando el usuario lo requiera. Los ".exe" de Windows o los archivos con permisos de ejecución de Unix (Linux) son ejemplos de ejecutables.

También se destacarán los **servicios** que son programas que es necesario que se estén ejecutando de forma constante para hacer su tarea. Ejemplo de servicio puede ser un servidor web que debe ejecutarse de forma constante para servir las páginas web a los usuarios. Si se parase el servidor web, los usuarios no serían atendidos.

Otros ejemplos de servicios son los servidores de bases de datos, correo electrónico, DNS, FTP, SSH,…

2 Conceptos básicos sobre concurrencia

Hasta ahora los programas que se han realizado son secuenciales, es decir, que para ejecutar una instrucción tiene que haber terminado de ejecutarse la anterior. Si nos fijamos en el siguiente pseudocódigo:

```
1: int a = 3;
2: int b = 2;
3: int c = a + b;
```

Se ejecuta primero la línea 1, después la 2 y después la línea 3. No hay posibilidad de ejecutar el programa en otro orden una vez escrito, ni de ejecutar dos líneas a la vez. Por ejemplo, las líneas 1 y 2 se podrían ejecutar de forma simultánea y no se alteraría el resultado. ¿No sería genial poder decirle al ordenador que estas dos líneas las puedes ejecutar a la vez? Sí, se puede. Ésto va a plantear una serie de ventajas y problemas que se verán en este texto.

Supongamos ahora que cada línea del código anterior tardase un segundo en ejecutarse. Si se ejecuta el programa secuencialmente, una línea tras otra, va a tardar 3 segundos en ejecutarse. Si se le pudiese decir al ordenador que las líneas 1 y 2 se ejecuten simultánemente, el programa tardaría sólo 2 segundos en ejecutarse.

Actualmente se disponen de dispositivos en los que es posible tener uno o varios procesadores:

- Los **sistemas monoprocesador** son aquellos que sólo disponen de un procesador con un solo núcleo.
- Los **sistemas multiprocesador** son aquellos que disponen de varios procesadores o varios núcleos.

En ambos casos va a ser posible realizar concurrencia:

*Un **programa concurrente** es aquel que permite definir un conjunto de acciones que pueden ser ejecutadas simultáneamente.*

Es decir, si se lanza un juego, este puede programarse para que de forma simultánea haga sonar el sonido, mostrar los héroes y villanos y responder a las peticiones del jugador. Si no se hiciera la programación concurrente, hasta que no terminase de sonar la música (que puede durar 2 minutos), no se mostrarían los héroes y villanos. Hasta que no se mostrasen los héroes y villanos, no se responderían las peticiones del jugador. Esto haría que la experiencia de juego fuese arruinada totalmente.

¿Es posible tener concurrencia en un sistema monoprocesador? Respuesta corta: sí.

Respuesta larga: Al tener un solo procesador no va a ser posible tener a dos programas ejecutándose en el mismo instante de tiempo en el procesador. Si se dispone de un sistema operativo adecuado, este puede alternar el tiempo que ocupa cada programa en el procesador, de forma que se deja ejecutar un programa, llamémosle P1, un par de ciclos y los siguentes 3 ciclos se permite ejecutar a otro programa, P2. Después se vuelve a ejecutar P1 otro ciclo de procesador más y vuelve a P2,… Como la alternancia entre programas se hace muy rápidamente, el usuario va a tener la sensación de que los programas se ejecutan de forma simultánea.

Este forma de gestionar los procesos recibe el nombre de **multiprogramación**. Tiene la ventaja de hacer un aprovechamiento más eficiente del procesador.

Los sistemas multiprocesador tienen más riqueza a la hora de hacer concurrencia. Se distinguirá entre:

- **Sistemas fuertemente acoplados:** Todos los procesadores comparten la misma memoria.
- **Sistemas débilmente acoplados:** Cada procesador tiene su propia memoria local que no comparten con otros procesadores.

Un ejemplo de sistema fuertemente acoplado puede ser un ordenador con un procesador multinúcleo. Todos los núcleos comparten la misma memoria.

Un ejemplo de sistema débilmente acoplado se tiene en una red de servidores que tengan que generar páginas web para los usuarios que se conectan.

La memoria compartida entre procesadores permite que los programas puedan compartir información de forma rápida. En el sistema débilmente acoplado se tendrá que recurrir a otras técnicas como la conexión por protocolo TCP-IP a otros equipos.

Al intercambio de información entre los programas ya sea por memoria compartida o pro otros procedimientos se le denomina **paso de mensaje**.

Suele denominarse **multiproceso** a la gestión de varios procesos dentro de un sistema fuertemente acoplado y **procesamiento distribuido** a la gestión de varios procesos dentro de un sistema débilmente acoplado.

Por último las siguientes definiciones son importantes:

- Un **programa concurrente** es aquel que tiene partes que pueden ser ejecutadas simultáneamente.
- Un **programa paralelo** es el que está diseñado para ser ejecutado en un sistema multiprocesador.
- Un **programa distribuido** es aquel que está diseñado para ejecutarse en una red de procesadores que no tienen una memoria común.

3 Procesos

Se denomina **proceso** a un programa en ejecución. Por ejemplo, cuando se ejecuta el comando “ls” en la consola de Linux, el sistema operativo busca el ejecutable correspondiente a “ls” se lanza un proceso ejecutando las órdenes de dicho programa.

En los sistemas operativos multitarea, como es el caso de Unix/Linux, un único procesador puede ejecutar varios procesos de forma concurrente. Para ello se utilizan algoritmos de planificación, **planificador de procesos,** que deciden cuándo se debe desalojar a un proceso de la CPU para asignársela a otro.

A cada proceso se le va a asociar un número que servirá para identificarlo, el **PID**.

Por ejemplo, en Unix (por Linux o MacOS) se dispone del comando “ps” que sirve para ver los procesos en ejecución y ver su **PID:**

```
$ ps -A
  PID TTY          TIME CMD
    1 ?        00:00:01 systemd
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 rcu_gp
    4 ?        00:00:00 rcu_par_gp
    6 ?        00:00:00 kworker/0:0H-kblockd
    9 ?        00:00:00 mm_percpu_wq
   10 ?        00:00:02 ksoftirqd/0
   11 ?        00:00:01 rcu_sched
```

La opción “-A” sirve para ver todos los procesos del sistema. Sin la opción “-A” se ven sólo los procesos lanzados en la sesión actual.

Con el símbolo “&” después de un comando se puede hacer que se ejecute en segundo plano. Por ejemplo:

```
$ less &
[1] 6369
```

El comando “less” se ha ejecutado en 2º plano y se indica el PID que se le ha asignado, en este caso el 6369. Con “ps” se puede consultar el PID:

```
$ ps
   PID TTY          TIME CMD
  6091 pts/0    00:00:00 bash
  6369 pts/0    00:00:00 less
  6382 pts/0    00:00:00 ps
```

Con el comando “fg”, se recupera el proceso en segundo plano:

```
fg
```

Para matar al proceso se usará kill + el número del proceso. Por ejemplo, se ejecuta el comando “nano” en un terminal:

```
$ nano
```

Desde otro terminal se ejecuta “ps -A” para ver el número del proceso:

```
$ ps -A
…
  6109 ?        00:00:00 kworker/u4:5-btrfs-endio-write
  6431 pts/0    00:00:00 nano
  6433 ?        00:00:00 qterminal
  6436 pts/1    00:00:00 bash
  6447 pts/1    00:00:00 ps
```

Se ejecuta kill para matar al proceso:

```
$ kill 6431
```

Se verá que el proceso ha desaparecido.

Cada proceso puede lanzar otros procesos. Estos procesos se denominan **procesos hijo**. Al proceso que lanza a los procesos hijos se le denomina **proceso padre**.

Con el comando “pstree” se puede ver el árbol de procesos padres e hijos:

```
$ pstree
systemd─┬─ModemManager───2*[{ModemManager}]
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─agetty
        ├─at-spi-bus-laun─┬─dbus-daemon
        |                 └─3*[{at-spi-bus-laun}]
        ├─at-spi2-registr───2*[{at-spi2-registr}]
```

```
├─avahi-daemon───avahi-daemon
├─bluetoothd
├─cron
├─dbus-daemon
├─firefox─┬─Web Content───17*[{Web Content}]
|        ├─WebExtensions───17*[{WebExtensions}]
|        └─48*[{firefox}]
```

Curiosidad: El comando, en lenguaje C, para lanzar un proceso hijo es “fork”. La función “fork” lanza un proceso hijo que es exactamente igual al padre pero sólo se diferencia en el PID. Por ello se habla muchas veces de “hacer un fork”.

Curiosidad: Al siguiente código, en lenguaje C, se le denomina la bomba fork:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
   while(1)
            fork();
}
```

Lo que hace es lanza infinitos procesos hijos, y cada hijo hace lo mismo, hasta que el procesador se satura y el usuario pierde el control del equipo. La única solución es forzar el apagado de la máquina (botonazo).

Cuando practique haciendo hilos o procesos tenga en cuenta de no crear una “bomba fork” por error.

El PID de un proceso nunca cambia durante el tiempo de vida de éste. Sin embargo, sí puede verse modificado el valor de su **PPID**, el PID de su proceso padre. Cuando el proceso padre muere, el **proceso huérfano** pasa a considerarse como hijo del proceso **init**, en UNIX es el primer proceso que lanza el sistema operativo (en Linux ha sido sustituido por “systemd”), entonces el valor de su PPID toma el valor 1.

Un proceso pasa por diversos estados, los estados de un proceso en Unix son:

- D Uninterruptible sleep – (En pausa ininterrumpible). Por ejemplo por alguna operación IO.
- R Running – el proceso se encuentra ejecutándose
- S interruptible sleep Pausa interrumpible (espera que se complete un evento)
- T Stopped parado

- X Dead – muerto. Esto nunca deberíamos verlo, ya que si el proceso está muerto, no aparecerá ..
- Z Defunct -Proceso Zombie, terminado pero su proceso padre sigue vivo, y no ha detectado que el proceso ya murió.

Simplificando los estados que interesarán de un proceso son:

Un proceso es creado y pasa a esperar, listo para ejecución, a que el sistema operativo le asigne un turno en el procesador. Una vez que le asigna turno, pasa a ejecución. Para salir de la ejecución pueden suceder 4 casos:

- El programa termina su ejecución y pasa a estado “Terminado”.
- Al proceso se le acaba el turno en el procesador y vuelve a “Listo para ejecución” a esperar a que le asignen un nuevo turno.
- Abandona de forma voluntaria la ejecución y pasa a “Listo para ejecución”. Por ejemplo, un programa con una opción de “no molestar” que trata de minimizar su impacto en el sistema en casos de carga alta del procesador.
- Un proceso quiere acceder a un recurso, por ejemplo, la impresora y debe esperar a que esté disponible. Pasará entonces a estado “Bloqueado” y se desbloqueará cuando reciba el evento de que la impresora está disponible.

El sistema operativo usará un planificador de procesos para asignar los turnos a cada proceso.

3.1 El planificador de procesos

El sistema operativo crea una cola en la que se colocan los procesos a esperar a que se les asigne un turno para poder ejecutarse en el procesador. Imagine que tiene que ir a comprar al supermercado y en la línea de cajas se forma una cola para pagar.

Para clasificar los posibles algoritmos se tiene que tener en cuenta:

- Tiempo de espera es el tiempo que un proceso permanece en espera en la cola de ejecución.
- Tiempo de retorno es el tiempo que va desde que se lanza un proceso hasta que finaliza.
- Tiempo de respuesta es el tiempo que un proceso bloqueado tarda en entrar en ejecución.
- Uso de CPU es el porcentaje de tiempo que la CPU está ocupada.
- Productividad es el número de procesos realizados en una unidad de tiempo.

Los posibles algoritmos son:

- Primero en llegar, primero en ser servido. Es llamado también FIFO (First In, First Out). Consiste en que los procesos se ejecutan en el orden en el que van llegando. No hay multiproceso. Es típico de sistemas operativos como el MsDOS o FreeDOS en los que no existe el multiproceso. Como curiosidad, actualmente el sistema operativo FreeDOS se puede encontrar en ordenadores nuevos como reemplazo a Microsoft Windows:

- Primero la tarea más corta: En este caso se ejecuta primero la tarea que menos tiempo le quede para terminar. Tiene la ventaja de que se ejecutan muchos procesos en poco tiempo, pero se lastran los procesos más largos.
- Primero la tarea más corta con expulsión: Similar al anterior pero se puede expulsar del procesador una tarea si llega otra más corta. Tiene la desventaja que las tareas largas pueden no llegar a terminarse pues se expulsan muchas veces del procesador.
- Cinta o round-robin: Se define un quantum que es el tiempo que podrá estar ejecutándose un proceso en el procesador. Cada proceso es ejecutado por orden siguiendo la cola durante un quantum de tiempo y vuelve de nuevo al final de la cola para esperar su turno. Se considera que todos los procesos tienen la misma prioridad.

Además se puede añadir prioridad de forma que los procesos con más prioridad estén más tiempo ejecutándose en el procesador (usen más quantums) que los que tengan menos prioridad.

Como se puede ver los algoritmos descritos pueden ser:

- Apropiativos son los que permiten la expulsión de procesos para ejecutar un nuevo proceso, poniendo en cola al anterior. Ej. “Primero la tarea más corta con expulsión” y “Round-Robin”.
- No Apropiativos son los que no permiten la expulsión, por lo que un proceso nuevo no entrará hasta que termine el anterior. Ej., “Primero en llegar, primero en ser servido” y “Primero la tarea más corta”

4 Sincronización entre procesos. Exclusión mutua. Condiciones de sincronización.

Se ha comentado que un conjunto de instrucciones se pueden ejecutar de forma concurrente, por ejemplo, en el código:

```
1: int a = 3;
2: int b = 2;
3: int c = a + b;
```

Ya se ha visto que las líneas 1 y 2 se podrían ejecutar a la vez y el funcionamiento del programa seguiría siendo correcto. El problema que se tiene es cuándo decidir qué líneas se pueden ejecutar de forma concurrente y qué problemas se pueden causar.

4.1 Grafos de precedencia

Lo que se hace es representar cada instrucción en un grafo indicando las instrucciones que se van ejecutar de forma simultánea. Por ejemplo:

indica que las instrucciones “a= 2” y “b=2” se ejecutarán de forma simultánea. Hasta que no se hayan ejecutado ambas, no se ejecutará la instrucción “c = a + b”.

4.1.1 Orden total y orden parcial

En los programas que se ejecutan secuencialmente, hay un orden en la ejecución y siempre es el mismo. Cuando el orden de ejecución es siempre el mismo se dice que se tiene un **orden total**.

En los programas concurrentes el orden de ejecución es **orden parcial** en el que el orden de ejecución puede variar cada vez que se ejecuta. Por ejemplo, sea el siguiente grafo:

Como las tres instrucciones son simultáneas el orden de ejecución de las mismas puede variar:

Supongamos que el valor de las variables a y b es cero antes de que se ejecute el código concurrente. En el primer caso la variable c valdrá 4 al final de la ejecución, en el segundo valdrá cero:

Primer caso			
Instrucción	**Memoria**		
	a	**b**	**c**
Estado inicial	0	0	0
b=2	0	**2**	0
a=2	**2**	2	0
c=a+b	2	2	**4**

Segundo caso			
Instrucción	**Memoria**		
	a	**b**	**c**
Estado inicial	0	0	0
c=a+b	0	0	**0**
a=2	**2**	0	0
b=2	2	**2**	0

Como se puede ver en las tablas, el orden de ejecución variará el valor de las variables de la RAM dando resultados diferentes. Esto genera un indeterminismo en la ejecución de un programa concurrente.

4.1.2 Indeterminismo

Del ejemplo anterior se deduce que la ejecución de un programa concurrente puede ser **indeterminista**, el mismo código puede producir resultados diferentes cuando se ejecuta repetidamente sobre el mismo conjunto de datos de entrada.

Esto es un auténtico dolor de cabeza cuando se produce un error en un código que se ha de ejecutar de forma concurre. El proceso de depurar la aplicación es muy complejo.

En una película de mafiosos se indicaba que:

“La primera norma de hacer tratos con el diablo es… no hagas nunca tratos con el diablo.”

Se podría aplicar a la programación concurre:

“La primera norma de la programación concurre es… no hagas nunca programación concurrente.”

4.2 Condiciones de Bernstein

Para poder determinar si dos conjuntos de instrucciones se pueden ejecutar de forma concurre, se definen los siguientes conjuntos:

- Conjunto de lectura: Es el conjunto de de todas las variables que se consultan durante la ejecución de un conjunto de instrucciones.
- Conjunto de escritura: Es el conjunto de todas las variables que son actualizadas con nuevos valores durante la ejecución de un conjunto de instrucciones.

Para que dos conjuntos de instrucciones se puedan ejecutar de forma concurrente se tiene que cumplir que:

- No se lea y escriba la misma variable en los dos conjuntos de instrucciones. Es decir, que la intersección del conjunto de lectura de un conjunto de instrucciones con el conjunto de escritura del otro conjunto de instrucciones sea vacío.
- Que no se escriba la misma variable en los dos conjuntos de instrucciones. Es decir, que la intersección del conjunto de escritura de un conjunto de instrucciones con el conjunto de escritura del otro conjunto de instrucciones sea vacío.

Por ejemplo, sea el código que se quiere transformar en otro equivalente que se ejecute de forma concurrente:

```
1: a = c + d;
2: b = d + 2;
3: c = a + b;
4: d = b + 1;
```

Lo primero que hay que preguntarse es, ¿qué líneas de código se pueden ejecutar de forma concurrente?

Se hace la siguiente tabla:

Instrucción	Lee	Escribe
1: a = c + d	c, d	a
2: b = d +2	d	b
3: c = a + b	a, b	c
4: d = b +1	b	d

Las instrucciones 1 y 2 se pueden ejecutar concurrentemente, pues en las variables de su conjunto de lectura {c, d} no hay ninguna variable del conjunto de escritura {a, b}.

Las instrucciones 3 y 4 se pueden ejecutar concurrentemente, pues en las variables de su conjunto de lectura {a, b} no hay ninguna variable del conjunto de escritura {c, d}.

Ya se ha visto que las instrucciones {1, 2} se pueden ejecutar concurrentemente y {3, 4} se pueden ejecutar también concurrentemente. ¿Qué grupo de instrucciones se debe ejecutar primero?

¿Primero {1, 2} y después {3, 4}? ¿Primero {1, 2} y después {3, 4}? La respuesta es fácil, se desea que el código genera la misma respuesta que su versión no concurrente, por lo que primero se deben ejecutar las instrucciones que estuviesen primero en su versión no concurrente {1, 2} y después {3, 4}:

Hay códigos que no se pueden ejecutar de forma concurre. Supongamos que en el código anterior se cambian dos líneas de orden de la siguiente forma:

```
1: a = c + d;
2: c = a + b;
3: b = d + 2;
4: d = b + 1;
```

Se hace la tabla de los conjuntos de lectura y escritura:

Instrucción	Lee	Escribe
1: a = c + d	c, d	a
2: c = a + b	a, b	c
3: b = d +2	d	b
4: d = b +1	b	d

Las instrucciones 1 y 3 se pueden ejecutar concurrentemente, pues en las variables de su conjunto de lectura {c, d} no hay ninguna variable del conjunto de escritura {a, b}.

Las instrucciones 2 y 4 se pueden ejecutar concurrentemente, pues en las variables de su conjunto de lectura {a, b} no hay ninguna variable del conjunto de escritura {c, d}.

Por lo tanto {1, 3} se podrían ejecutar concurrentemente y {2, 4} también. Pero para que el código de la versión concurrente pueda generar el mismo resultado que el código de la versión no concurrente se ha de respetar el orden original de las líneas del código no concurrente. Puesto para que se genere un resultado correcto la instrucción 1 se debe ejecutar antes que la 2, este código no se puede ejecutar de forma concurrente, pues tanto {1, 3} como {2, 4} trabajan sobre las mismas variables {a, b, c, d}.

El ejemplo anterior se puede ampliar, se añaden más instrucciones:

```
1: a = c + d;
2: c = a + b;
3: b = d + 2;
4: d = b + 1;
5: x = a + b;
6: y = c + d;
```

Ya se ha visto que las instrucciones {1, 2, 3, 4} no se pueden ejecutar de forma concurrente. Pero las instrucciones {5, 6}, sí pues trabajan sobre un conjunto de escritura diferente. {1, 2, 3, 4} escribe en las variables {a, b, c, d}, {5, 6} escriben en las variables {x, y}. Por todo ello, se puede llegar al siguiente grafo de precedencia:

4.3 Sincronización entre procesos

Del apartado anterior se desprenden dos problemas que se deben resolver para poder realizar la programación concurrente con garantías, la exclusión mutua y el de la condición de sincronización.

4.3.1 Exclusión mutua

Se llamará **sección crítica** a una sección de código a la que dos procesos no pueden acceder de forma simultánea pues se provocarían resultados erróneos.

Interesa que las secciones críticas se ejecuten en **exclusión mutua**:

Sólo un proceso puede estar ejecutando la sección crítica como máximo.

Por ejemplo, suponga que se está programando el contador de visitas de una página web. En este caso la sección crítica va a ser el código que incrementa en uno el contador.

Supongamos que el contador está a cero y se producen dos visitas simultáneas. Se va a suponer que cada visita es gestionada por un proceso diferente, es muy común en los servidores web. Si los dos procesos entraran a la vez en la sección crítica sin respetar la exclusión mutua, sucedería lo siguiente:

Tiempo	**Proceso 1**	**Proceso 2**	**Variable en RAM: contador**
1	Accede a la sección crítica y la CPU se baja a los registros el valor de contador = 0	Accede a la sección crítica y la CPU se baja a los registros el valor de contador = 0	0
2	Se incrementa en uno la copia del registro de la la variable contador. contador = 1	Se incrementa en uno la copia del registro de la la variable contador. contador = 1	0
3	Se sube a la RAM el valor del registro de la CPU	Se sube a la RAM el valor del registro de la CPU	**¡¡¡¡ 1 !!!!**

Importante: *Se debe tener en cuenta que cuando un proceso, o un hilo (ya se verá más adelante), consulta el valor de una variable, este valor se copia en alguna caché o registro de la CPU, por lo que no se está trabajando con la variable de la RAM sino con una copia de la misma*. Cada proceso o hilo trabajará con su copia por unos instantes y subirá el resultado de la caché del procesador a la RAM.

Por lo tanto en el tiempo 1 lo que sucede es que cada núcleo de la CPU obtiene una copia de la variable contador de la RAM:

En el tiempo 2, cada núcleo trabaja con su copia de la variable:

En el tiempo 3, cada núcleo de la CPU copia en la RAM el resultado de la operación:

Como se puede ver aunque se han recibido dos visitas, el contador sólo se ha incrementado en 1, debido a que la programación de los procesos se ha realizado de forma incorrecta. En esta caso la sección crítica es la variable contador, con la cual no podrán trabajar los dos procesos a la vez.

4.3.2 Condición de sincronización

El problema de la **condición de sincronización** se produce cuando un proceso, P2, depende de que otro proceso, P1, haya terminado una determinada tarea para hacer su trabajo de forma correcta.

Por ejemplo, si P1 genera una factura en pdf y P2 la manda por correo electrónico, si P1 no ha terminado de generar la factura P2 no podrá enviarla o la enviará de forma incorrecta o incompleta.

El proceso P2 deberá esperar a que P1 termine para poder comenzar con su trabajo.

4.4 Propiedades que deben cumplir los programas concurrentes

Los programas concurrentes, para ejecutarse de forma correcta deben tener las siguientes propiedades:

- **Propiedades de seguridad**: son aquellas que aseguran que no se van a producir errores durante la ejecución del programa.
- **Propiedades de viveza**: son aquellas que aseguran que la ejecución será la correcta.

Hay que fijarse en que aunque no haya errores, la ejecución del programa puede no ser correcta.

Entre las propiedades de seguridad se tienen:

- **Exclusión mutua.**
- **Condición de sincronización.**
- **Interbloqueo o deadlock**: se produce interbloqueo cuando un proceso, P1, tiene un recurso que otro proceso necesita, P2, y P2 tiene un recurso que P1 necesita y no los liberan. Ambos procesos se quedan esperando eternamente a que los recursos que necesitan se liberen. Esta situación se denomina “abrazo mortal” o **deadlock.**

Entre las propiedades de viveza:

- **Interbloqueo activo o livelock**: es un interbloqueo en el que los procesos ejecutan instrucciones sin hacer ningún progreso. Por ejemplo, supongamos que se necesitan dos recursos, R1 y R2, para hacer una tarea. P1 tiene un recurso, R1, que P2 necesita y P2 tiene un recurso, R2, que P1 necesita. P1 libera el recurso R1 y P2 libera el recurso R2. P1 adquiere R2 y P2 adquiere R1. Se puede ver que se vuelve a la situación inicial. Un livelock es más difícil de detectar que un deadlock.
- **Inanición o starvation**: Se produce cuando uno o varios procesos necesitan un recurso y nunca se les llega a conceder. Por ejemplo, supongamos que los procesos P1, P2 y P3 necesitan imprimir de forma constante documentos. Supongamos que el sistema operativo es injusto y sólo concede la impresora a P1 y a P2, de forma que P3 nunca adquiere el recurso. P3 no progresará en su tarea.

5 Comunicación y sincronización entre procesos

Los procesos para comunicarse y sincronizarse pueden usar:

- Tuberías con nombre.
- Tuberías sin nombre.
- Señales.
- Memoria compartida.
- Semáforos.
- Región crítica condicional.
- Monitores.
- Paso de mensajes.

5.1 Tuberías con nombre