Curso: Sistemas Operativos
Tema 2. Administración de procesos.

2. Administración de procesos.

Objetivo.
El alumno clasificará los tipos de procesos y sus estados mediante las diferentes técnicas de comunicación y sincronización de procesos concurrentes

Contenido.
2.1 Procesos. Concepto y estructura.
2.2 Hilos. Concepto y estructura.
2.3 Estados de un proceso-hilo.
2.4 Interrupciones y suspensiones.
2.5 Concurrencia.

2.1 Procesos y estructura

Iniciaremos este tema con entender y definir lo que es un proceso. Sabemos que el proceso es el recurso software más importante de un sistema de cómputo-comunicación.

El término ***proceso*** fue utilizado por primera vez por los diseñadores del sistema operativo Multics en los años 1960. Desde entonces, el término proceso, referido también como ***tarea***, ha tenido muchas definiciones; he aquí algunas de ellas:

- Programa en ejecución.
- Actividad asíncrona.
- "Espíritu animado" de un procedimiento.
- "Centro de control" de un procedimiento en ejecución.
- La entidad a la que se le asignan los procesadores.

La más aceptada, para el común de los programadores, es la definición de proceso como un programa en ejecución. Sin embargo, como veremos más adelante, un programa en ejecución se puede dividir en varios procesos, por lo que un proceso puede estar constituido por solo una sección de código ejecutable.

- Diferencias entre programa y proceso.

Un programa es una entidad ***estática*** ya que se encuentra almacenado en memoria secundaria como un tipo de archivo. En tanto, un proceso es ***dinámico*** porque para ser ejecutado, o sea, asignado al procesador, debe ubicarse en memoria principal.

- Creación de procesos.

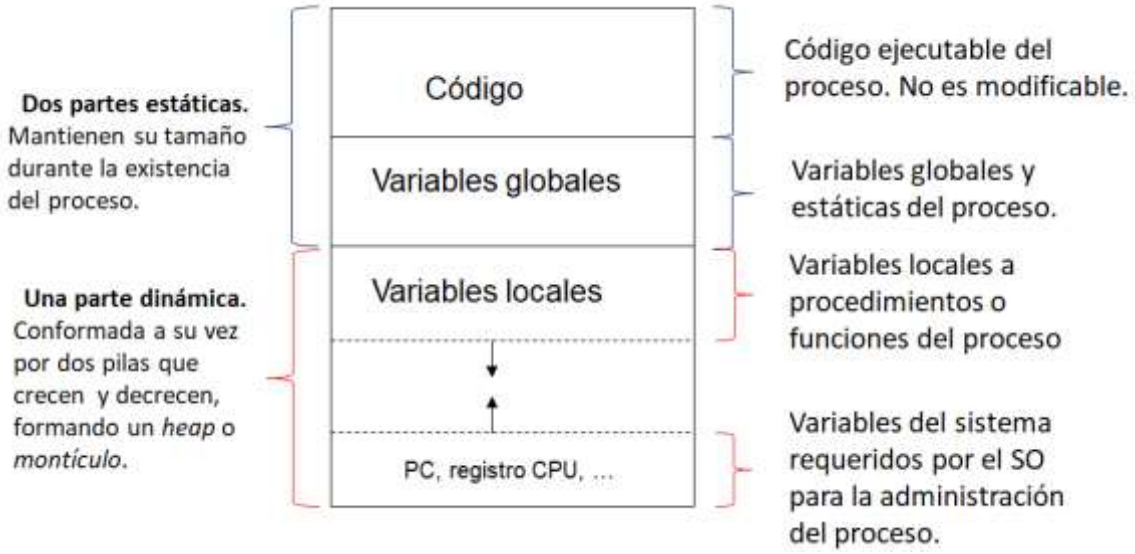
Los procesos se pueden crear por las siguientes maneras:

- Como parte del arranque del sistema operativo. En el sistema operativo Linux, el primer proceso que se crea es init( ).
- Dando la orden de ejecución de un programa o aplicación, mediante un intérprete de comandos o una interfaz gráfica (por ejemplo IDE).
- A través de un programa en ejecución utilizando llamadas al sistema. En Unix/Linux con la llamada al sistema fork( ).
- Como parte del procesamiento por lotes en un sistema que lo realice de manera automática.

Cuando se crea un proceso, se le asignan recursos del sistema, como memoria principal, dispositivos de entrada/salida, entre otros.

- Estructura de un proceso.

Un proceso está conformado básicamente por tres partes:



- Bloque de control de proceso.

El bloque de control de proceso (BCP), también nombrado como *descriptor de proceso*, es una estructura de datos que contiene información importante acerca de un proceso. El BCP generalmente contiene:

- El estado actual del proceso (lo describiré en el punto 2.3)
- Un identificador único (PID).
- Un apuntador hacia al proceso padre (el que lo creó).
- Apuntadores a los procesos hijo (creados por el proceso).

- Prioridad del proceso.
- Apuntadores hacia zonas de memoria del proceso
- Apuntadores a los recursos asignados al proceso.
- Un área de salvaguarda de los registros.
- El procesador en el que se está ejecutando el proceso, en caso que sea un sistema de múltiples procesadores (multiprocesamiento).

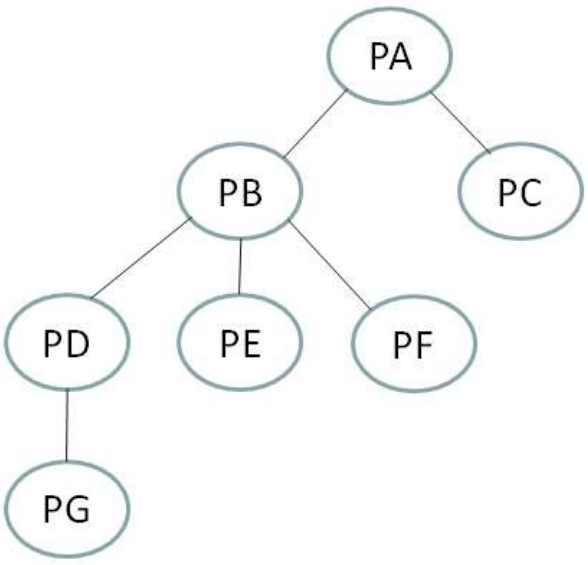
Cuando el sistema operativo hace cambio de contexto en un sistema multitareas, utiliza las áreas de salvaguarda de los BCP para guardar la información que se necesita para reiniciar un proceso cuando a éste le sea asignado de nuevo el procesador.

Por lo que el BCP es la entidad que define un proceso al sistema operativo. Debido a que los BCP deben ser manejados con eficiencia por el sistema operativo, muchos sistemas de cómputo tienen un registro de hardware que siempre apunta hacia el BCP del proceso que se está ejecutando. A menudo existen instrucciones de hardware que cargan en el BCP información acerca del estado, la cual recuperan con rapidez.

- Estructura jerárquica de procesos.

Un proceso puede crear un nuevo proceso. Si lo hace, el proceso creador se denomina *proceso padre*, y el proceso creado, *proceso hijo*. Un proceso puede crear *n* nuevos procesos, por lo que tendrá *n* procesos hijo. Sin embargo, un proceso sólo va a tener un proceso padre. A esta forma de creación de procesos se le llama *estructura jerárquica de procesos*.

De forma esquemática, la estructura jerárquica de procesos se puede mostrar como un árbol invertido.



Podemos observar que:

- El proceso A (PA) tiene dos procesos hijo: PB y PC.
- El proceso B (PB) tiene 3 procesos hijo: PD, PE y PF.
- El proceso D (PD) tiene un proceso hijo: PG
- Y los procesos PC, PE, PF y PG no han creado ningún proceso, por lo que no tienen procesos hijo.
- Todos los procesos tienen un proceso padre, excepto el proceso

Procesos en Linux

- Los procesos en el sistema operativo Linux, están compuestos por tres bloques o segmentos:

  - ❖ Segmento de texto. Código ejecutable del programa.
  - ❖ Segmento de datos. Variables globales y estáticas.
  - ❖ Segmento de heap (montículo). Dinámico, conformado por dos pilas:
    - ✓ pila en modo usuario. Variables locales
    - ✓ pila en modo kernel. Variables del sistema

- Linux utiliza una *Tabla de procesos,* equivalente al BCP*,* requerido para el control de éstos. Su contenido es:

  * Estado
  * Apuntadores a memoria ocupada
  * Campo de señales
  * Temporizadores para cálculo de prioridad dinámica
  * Parámetros de planificación
  * UID. Identificador de usuario
  * PID. Identificador de proceso
  * Descriptores de eventos

- Todos los procesos, excepto el primero (init), son creados mediante la llamada al sistema *fork( )*

La llamada al sistema fork() crea un proceso nuevo; el proceso que hace la llamada a fork() es conocido como padre, y el creado, como hijo. El proceso hijo realiza el mismo código del padre, con la diferencia de que fork() devuelve al proceso hijo el valor de 0 y al padre el valor del identificador (pid) del hijo.

La declaración de fork() de manera implícita, es decir, ya definida, es:

```
#include <sys/types.h>
#include <unistd.h>
pid_t  fork();
```

Es conveniente invocar estos #include para asegurar que use el estándar de unix/Linux; el tipo de dato que regresa fork() está definido en sys/types.

Como fork() devuelve un valor entero, ya que puede devolver un 0 o el identificador del proceso hijo, cuyo valor también es un entero, podemos usar una variable de tipo entera para guardar el valor que devuelve.

Revisemos un pequeño programa en C con la llamada al sistema fork().

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

void main()
{
   fork();
   printf("Hola mundo");
}
```

Para asegurar el buen funcionamiento de fork()

Al regreso de la llamada fork(), se tiene creado el proceso hijo que ejecutará el mismo código del proceso padre a partir del fok().

Actividad 2.1.1 Crea un programa en C que contenga el código arriba mostrado. Compílalo en la plataforma Linux (te recomiendo usar gcc) y ejecútalo; observa lo que pasa. Agrega otras sentencias después del printf("Hola mundo"); que incluya mensajes de salida. Compílalo y ejecútalo.

- Características del proceso padre versus las del proceso hijo.

Una vez creado el proceso hijo, se hace la devolución del control al sistema, para seguir la ejecución de los procesos tanto del padre como del hijo.

Es así que el sistema los ve como dos procesos los cuales tienen las siguientes características:

- ❖ Cada uno cuenta con identificador único de proceso, que generalmente se le nombra *pid*. Es decir, el *pid* del hijo es distinto al del padre. El valor del *pid* es un número entero y conforme se van creando los procesos se les va asignando el siguiente número consecutivo. Si al efectuar una llamada a fork() no se puede crear un proceso, entonces al proceso que hizo la llamada le regresa el valor de -1.
- ❖ El valor que devuelve la llamada al sistema fork() a los procesos padre e hijo es diferente: devuelve un 0 al proceso hijo y un valor positivo distinto de 0, el cual es el *pid* del hijo, al padre.
- ❖ Si el proceso padre termina antes que el proceso hijo, el proceso hijo es adoptado por el proceso *init* de Linux. Recordemos que el proceso *init* es el proceso inicial del sistema; su *pid* siempre es 1. Esto se realiza porque Linux, maneja una estructura jerárquica de procesos donde todos los procesos deben tener un proceso padre.

- Recursos compartidos entre los proceso Padre e Hijo
  - ❖ La imagen del código del proceso hijo es la del proceso padre. Es decir comparten esa misma zona de memoria.

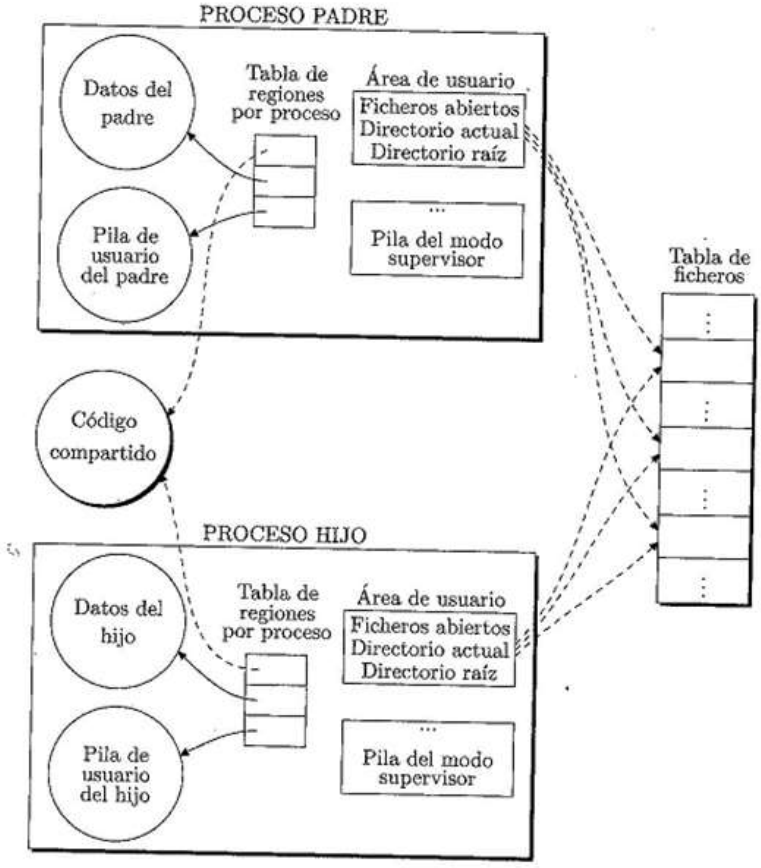
- ❖ Los dispositivos y archivos que tuviera abiertos el proceso padre antes de crear el proceso hijo, serán compartidos. Incluyen el stdin, stdout y stderr.
  - Esto quiere decir que la salida estándar y salida de error estándar de padre e hijo aparecerán mezcladas.

- Recursos no compartidos entre los procesos Padre e Hijo

  - ❖ No se comparten las variables de memoria.
    - Las variables globales y estáticas del proceso padre, junto con sus valores, se copian a otra área de memoria asignada al proceso hijo. De tal manera que si el proceso padre hace cambio de valores a sus variables globales o estáticas, ese cambio no se hace en las variables globales y estáticas del proceso hijo y viceversa.

    - Al igual que el caso anterior, las variables locales y del sistema se copian en otra área de memoria que es asignada al proceso hijo. Es así, que los dos procesos, padre e hijo, pueden ejecutar diferentes partes del código porque sus contadores de programa (PC) tienen diferente valor.

- Esquema de recursos compartidos y no compartidos entre procesos Padre e Hijo usando fork().



Revisemos el siguiente programa:

```
/*   Programa 1
     Programa que hace uso de las llamadas al sistema fork(), getpid() y getppid().

    Ejemplifica la creación de un proceso hijo y la obtención del identificador de
    proceso para cada proceso.
*/
#include <sys/types.h>
#include < unistd.h>
#include <stdio.h>

main()
{
   int   pidP, pidH;

   pidH = fork(); /* Se crea el proceso hijo */
   if (pidH!=0) { /* código del padre */
     pidP=getpid(); //obtengo el pid del padre
     printf("\n Soy el padre mi identificador de proceso es:%d", pidP);
     printf ("\n Mi  hijo tiene el pid=%d", pidH);
   }
   else{ /* código del hijo */
     printf("\n Soy el hijo mi pid es: %d\n", getpid());
     printf("\n El pid de mi padre es: %d\n", getppid());
   }
}
```

Observamos en el programa anterior:

- El uso de la llamada al sistema getpid() que devuelve el identificador del proceso que la llama.
- El uso de la llamada al sistema getppid() que devuelve el identificador del proceso padre de quien la llama.
- Cada proceso realizará una porción de código diferente. Esto se logra con la estructura *if*, la cual compara el valor regresado por la llamada al sistema:
    - Si el valor pidH en la asignación pidH = fork(); es diferente de 0, entonces el que la está ejecutando es el proceso padre. De otro modo, si pidH es 0 el proceso que está ejecutando el if es el proceso hijo.

Actividad 2.1.2 Crea un programa en C, con el código del Programa 1. Compílalo y ejecútalo. Observa los mensajes de salida. Revisa lo que escribe el proceso hijo y lo que escribe el proceso padre.

Actividad 2.1.3 Hazle modificaciones al programa anterior manejando variables locales y globales, dales un valor inicial antes del fork(). Después cámbiales su valor tanto en el hijo como en el padre y observa cómo se manejan en zonas de memoria distinta a pesar de tener el mismo nombre.

2.3 Estados de un proceso.

Un proceso pasa por una serie de estados discretos. Un proceso cambia de estados a causa de varios eventos.

Podemos hablar de tres estados básicos en los que un proceso puede estar:

- **Estado de ejecución**. Al proceso se le ha asignado la unidad central de procesamiento y se está ejecutando.
- **Estado listo**. El proceso está listo para que el planificador de procesos del sistema operativo lo considere para asignarle una unidad central de procesamiento.
- **Estado bloqueado**. El proceso está esperando que suceda algún evento antes de estar listo para que se le asigne nuevamente la unidad central de procesamiento.

Hay sistemas operativos que manejan otros estados; algunos se complementan con los tres básicos antes mencionados. Por esto, en este tema nos enfocaremos a trabajar solamente con dichos estados.

Como vimos en la evolución de los sistemas operativos, en la actualidad, la mayoría son multitarea. Es decir, el sistema operativo atiende a varios procesos de forma simultánea. Por lo que al tratar este tema consideraremos a un sistema multitarea, y, por sencillez, un sistema con sólo una unidad central de procesamiento.

Entonces, en un sistema multitarea y monoprocesador, sólo un proceso puede estar en estado de ejecución, varios en estado listo y varios en estado bloqueado. En estos dos últimos estados, los procesos pueden convivir de diferentes formas, esto dependerá de la técnica que use el planificador de procesos en el caso de procesos en estado listo, o bien, dependerá mucho de cómo vayan presentándose los eventos a los procesos que están en estado de bloqueado.

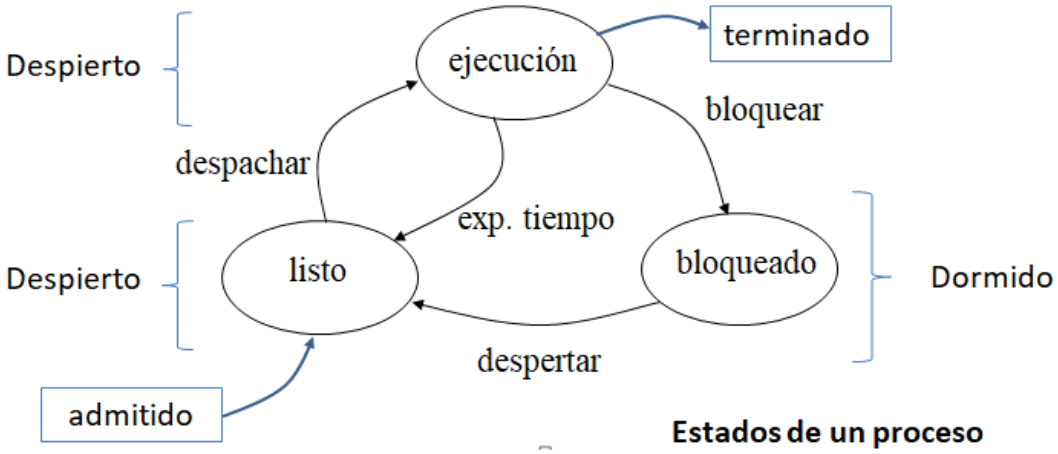
Transición de estados de los procesos.

Cuando un proceso pasa de un estado a otro, se dice que ocurre una transición. Cada transición tiene un nombre que identifica de qué a qué estado pasa un proceso. Veamos cuáles son estas transiciones:

- **Despachar**. Es la transición que pasa un proceso de estado listo a ejecución. El planificador de procesos de bajo nivel es el encargado de seleccionar qué proceso es al que se le asignará el procesador.
- **Bloquear**. Ocurre cuando un proceso pasa de ejecución a bloqueado porque ocurre un evento donde el procesador no puede seguir con la ejecución del proceso hasta que otro hardware o software atienda ese evento. Por lo que el procesador se aprovecha para

ejecutar otro proceso. Un caso muy típico de bloqueo es cuando el proceso va a realizar una operación de entrada/salida.

- **Despertar.** Transición que ocurre cuando un proceso recibe la señal de que el evento, por el que tuvo que ser bloqueado, termina y pasa a estado listo.
- **Expiración tiempo**. Cuando en un sistema multitareas se desea ser justo en la asignación y estancia de un proceso en el procesador, entonces se le asigna un tiempo de ejecución. Si no termina el proceso en ese tiempo, pasa a estado de listo.



Estados de un proceso

Del diagrama de Estados de un proceso, podemos observar que hay dos estados encerrados en recuadros: admitido y terminado. Revisémoslos:

- **Admitido**. Es un estado transitorio de muy poca duración. Es cuando se crea el proceso y se le asigna espacio en memoria para colocarlo en la estructura de los procesos listos.

- **Terminado.** También es un estado transitorio de poca duración. Ocurre cuando el proceso terminó de ejecutarse y sus estructuras están a la espera de ser limpiadas por el sistema operativo. La terminación de un proceso puede ser:
  - ✓ *Salida normal*. El proceso es el que decide en su algoritmo que ya terminó,
  - ✓ *Por un error crítico*. El proceso quiere acceder a zonas de memoria que no le corresponde o quiere hacer una operación no adecuada que afecte su funcionalidad.
  - ✓ *Por una condición de excepción*. Cuando el proceso procede a terminar ante diversas situaciones inesperadas.
  - ✓ *Recibir una señal de otro proceso*. Otro proceso con autoridad elimina al proceso.

También podemos observar en el diagrama, que los procesos que están en estado listo y el que está en ejecución, se dice que están **despiertos**. Mientras que los que están en estado bloqueado están **dormidos**.

Los procesos que están en estos tres estados básicos se dice que están **activos**.

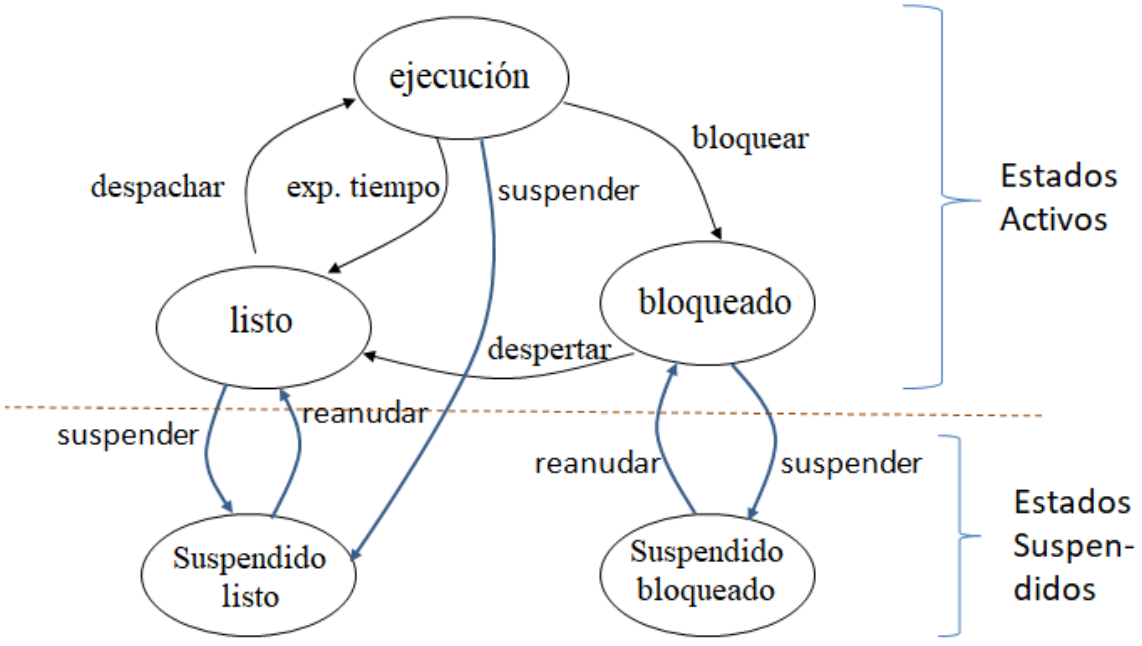
2.4 Interrupciones y suspensiones.

Un proceso suspendido no puede proseguir sino hasta que lo reanuda otro proceso.

La suspensión dura por lo regular sólo periodos breves. Cuando hay suspensiones largas se deben liberar los recursos del proceso. Pero la decisión de liberar o no los recursos depende mucho de la naturaleza de cada recurso como la memoria principal o un dispositivo de e/s. Reanudar (o activar) un proceso implica reiniciarlo a partir del punto en el que se suspendió.

Razones para suspender a un proceso:

- Si un sistema está funcionando mal y es probable que falle, se pueden suspender los procesos activos para reanudarlos cuando se haya corregido el problema.
- Un usuario que desconfíe de los resultados parciales de un proceso puede suspenderlo (en vez de abortarlo) hasta que verifique si el proceso funciona correctamente o no.
- Algunos procesos se pueden suspender como respuesta a las fluctuaciones a corto plazo de la carga del sistema y reanudarse cuando las cargas regresen a niveles normales.



Estados de un proceso, incluidos estados de suspensión

Actividad 2.4.1 Completa la siguiente tabla poniendo en la celda vacía el correspondiente estado y/o transición.

PID	Estado actual	Transición	Estado final
215		despachar	
217			Suspendido listo
215	ejecución		listo
218	bloqueado		listo
220		reanudar	
221		suspender	

Operaciones sobre procesos.

Los sistemas que administran procesos deben ser capaces de realizar ciertas operaciones sobre procesos y con ellos. Por lo general, estas operaciones incluyen:

- Crear un proceso. Al crear un proceso se realizan estas operaciones:
  - Dar un identificador al proceso
  - Insertarlo en lista de procesos del sistema.
  - Crear su bloque de control de proceso
  - Asignar los recursos iniciales al proceso
  - Determinar la prioridad inicial del proceso
- Destruir un proceso
- Suspender un proceso
- Reanudar un proceso
- Cambiar la prioridad de un proceso
- Bloquear un proceso
- Despertar un proceso
- Despachar un proceso
- Permitir que un proceso se comunique con otro.

Interrupciones.

Una interrupción suele ser un evento asíncrono, que es un evento que puede ocurrir en cualquier momento, por lo que no está sincronizado con el reloj del sistema ni con el ciclo de ejecución de instrucciones del procesador.

La interrupción indica al procesador que necesita manipular un evento de alta prioridad. El hardware del procesador suele incluir uno o más registros de interrupción que establece el evento de interrupción.

El procedimiento que realiza el sistema operativo ante una interrupción es el siguiente:

- El sistema operativo toma el control, ante la recepción de una señal de interrupción.
- El sistema operativo guarda el estado del proceso interrumpido. En muchos sistemas esta información se guarda en el bloque de control de procesos del proceso interrumpido.
- El sistema operativo analiza la interrupción y transfiere el control a la rutina apropiada para atenderla; en muchos casos, el hardware es el que se encarga de esto automáticamente.
- La rutina del manejador de interrupciones procesa la interrupción.
- Se restablece el estado del proceso interrumpido.
- Continúa la ejecución del proceso interrumpido.

Existen cinco clases de Interrupciones:

- *Interrupciones de llamadas al sistema o supervisor*. Son iniciadas por el proceso que está en ejecución; a través de las cuales, el usuario genera una petición de servicio particular al sistema. Así el sistema operativo evita que los usuarios intenten rebasar límites y puede rechazar ciertas peticiones si el usuario no tiene los privilegios necesarios.

- *Interrupciones de E/S*. Son iniciadas por hardware de entrada y salida. Estas interrupciones indican al procesador el cambio de estado de un canal o dispositivo. Por ejemplo, la terminación de una transferencia de E/S hacia o desde disco a memoria principal.

- *Interrupciones externas de E/S*. Cuando hay una incidencia por parte de un usuario u otro procesador. Por ejemplo, que el usuario mueva el ratón, haga clik en un botón del ratón, o teclee algún dato.

- *Interrupciones de verificación de la máquina. Son ocasionadas por el mal funcionamiento del hardware.*

- *Interrupciones de verificación del programa*. Son ocasionadas por una amplia clase de problemas que pueden ocurrir cuando se ejecutan las instrucciones de un proceso. Por ejemplo:

    - La división entre cero
    - La presencia de datos con formatos erróneos
    - El intento de ejecutar un código de operación inválido
    - El intento de hacer referencia a una localidad de memoria fuera de la zona asignada.
    - El intento de ejecutar una instrucción privilegiada.
    - Hacer referencia a un recurso protegido.

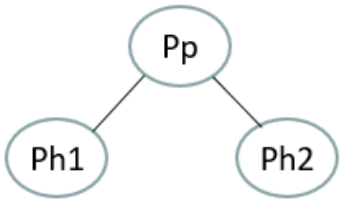
Actividad 2.4.2 Realiza un análisis comparativo entre Suspensiones vs Interrupciones. Menciona tres grandes diferencias.

Procesos en Linux (continuación)

Hasta el momento, hemos trabajado con las llamadas al sistema: fork(), getpid() y getppid. Estas llamadas y otras que se verán más adelante, propician que el proceso que las hace se interrumpa.

Realizaremos algunos ejercicios para mostrar cómo se pueden crear árboles jerárquicos de procesos.

Ejercicio 2.4.1 Elaborar un programa en C, con llamadas al sistema, que construya el siguiente árbol de procesos. Cada proceso deberá escribir qué proceso es y, al menos, su identificador de proceso.



Pp → Proceso padre
Ph1 → Proceso hijo 1 (primero en crearse)
Ph2 → Proceso hijo 2 (segundo en crearse)

Respuesta:

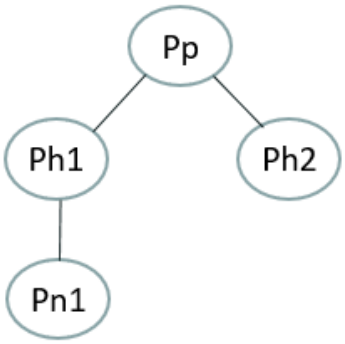
```
#include <sys/types.h>
#include < unistd.h>
#include <stdio.h>

main()
{
   int   pidP, pidH1, pidH2;

   pidH1 = fork(); /* Se crea el proceso hijo1 */
   if (pidH1 != 0) { /* código del padre */
      pidH2 = fork(); /* el padre crea el proceso hijo2 */
      if (pidH2 != 0) { /* sigue siendo el código del padre */
         pidP= getpid();
         printf("\n Soy el padre mi identificador de proceso es:%d", pidP);
         printf ("\n Mis hijos tienen los pidH1=%d y pidH2=%d ", pidH1, pidH2);
      }else{ /* código del hijo2*/
         printf("\n Soy el hijo2 mi pid es: %d\n", getpid());
         printf("\n El pid de mi padre es: %d\n", getppid());
      }
   }else{ /* código del hijo1*/
         printf("\n Soy el hijo1 mi pid es: %d\n", getpid());
         printf("\n El pid de mi padre es: %d\n", getppid());
   }
}
```

Actividad 2.4.3 crea el programa de respuesta del ejercicio; compílalo y ejecútalo. Podrás observar que los identificadores de procesos son consecutivos. Posiblemente cuando alguno de los hijos escriba el pid del padre, éste sea 1 u otro proceso que sea la raíz de inicio de sesión del usuario. Esto ocurriría si el código que ejecuta el padre ha terminado, entonces lo adopta el proceso init() del usuario. Para ver esto, coloca la instrucción *sleep(3);* antes de escribir los mensajes en alguno de los hijos.

Podemos observar en el programa de respuesta, que los procesos hijo hacen uso de getpid() y getppid() para escribir los identificadores de procesos en el printf. ¿Qué valores crees que escribirían si se sustituyen dichas llamadas por pidH1 o pidH2 (según corresponda) y pidP?

Recordemos que al crear los procesos hijo las variables (locales y globales o estáticas) se copian a otra área de memoria, este sería el caso de pidP. En los procesos hijo, ¿qué valor reciben de fork()?

Ejercicio 2.4.2 Elaborar un programa en C, con llamadas al sistema, que construya el siguiente árbol de procesos. Cada proceso deberá escribir qué proceso es y, al menos, su identificador de proceso.



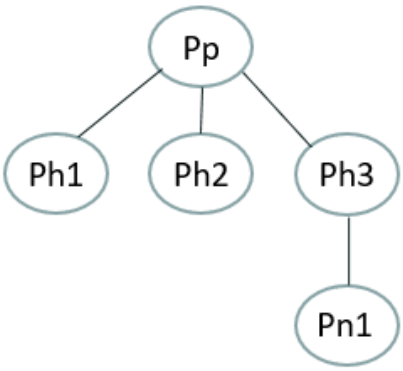
Pp → Proceso padre
Ph1 → Proceso hijo 1 (primero en crearse)
Ph2 → Proceso hijo 2 (segundo en crearse)
Pn1 → Proceso nieto 1 (proceso hijo del proceso hijo 1)

Respuesta:

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

main() {
   int   pidP, pidH1, pidH2, pidN1;

   pidH1 = fork(); /* Se crea el proceso hijo1 */
   if (pidH1 != 0) { /* código del padre */
      pidH2 = fork(); /* el padre crea el proceso hijo2 */
      if (pidH2 != 0) { /* sigue siendo el código del padre */
         pidP= getpid();
         printf("\n Soy el padre mi identificador de proceso es:%d", pidP);
         printf ("\n Mis hijos tienen los pidH1=%d y pidH2=%d ", pidH1, pidH2);
```

```
      }else{ /* código del hijo2*/
         printf("\n Soy el hijo2 mi pid es: %d\n", getpid());
         printf("\n El pid de mi padre es: %d\n", getppid());
      }
   }else{ /* código del hijo1*/
      pidN1 = fork(); /* el hijo 1 crea un proceso hijo (Pn1) */
       if (pidN1 != 0){ /* sigue siendo el hijo 1 */
         printf("\n Soy el hijo1 mi pid es: %d\n", getpid());
         printf("\n El pid de mi padre es: %d\n", getppid());
      }else{ /* código del proceso nieto 1 (hijo del proceso hijo 1)*/
         printf("\n Soy el nieto1 mi pid es: %d\n", getpid());
         printf("\n El pid de mi padre (hijo 1) es: %d\n", getppid());
      }
   }
}
```

Actividad 2.4.4. Elaborar un programa en C, con llamadas al sistema, que construya el siguiente árbol de procesos. Cada proceso deberá escribir, al menos, qué proceso es y su identificador de proceso.



Pp → Proceso padre
Ph1 → Proceso hijo 1 (primero en crearse)
Ph2 → Proceso hijo 2 (segundo en crearse)
Ph3 → Proceso hijo 3 (tercero en crearse)
Pn1 → Proceso nieto 1 (proceso hijo del proceso hijo 3)

Otras llamadas al sistema en Linux.

Una llamada al sistema muy empleada es wait(). Se usa para que el proceso padre detenga su ejecución hasta que algún proceso hijo termine.

Generalmente la llamada al sistema wait() se usa con la llamada exit(). La llamada exit() indica al sistema que termina el proceso que la invoca.

La definición de estas llamadas es:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int *stat_loc);
```

Si *stat_loc* es NULL, wait() sólo espera que termine un hijo, sin que éste le envíe información. *stat_loc* ocupa 16 bits.
Esta llamada devuelve el *pid* del proceso que terminó.

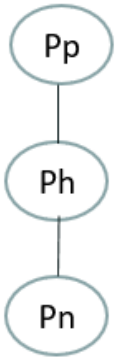
```
#include <stdlib.h>
void exit(int status);
```

Termina el proceso que la invoca.

Devuelve el valor de *status* tanto al sistema operativo como a la llamada wait() en su argumento.

El argumento *status* ocupa sólo 8 bits.

Ejercicio 2.4.3 Elaborar un programa en C, con llamadas al sistema, que construya el siguiente árbol de procesos. Cada proceso deberá escribir el mensaje indicado cuando su hijo termine.



Pp → Soy la ficha roja
Ph → Soy la ficha verde
Pn → Soy la ficha negra

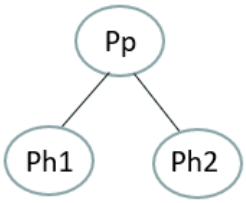
Entonces la salida debe ser:
Soy la ficha negra
Soy la ficha verde
Soy la ficha roja

Respuesta:

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

main()
{   int   pidH, pidN;

   pidH = fork(); /* Se crea el proceso hijo */
   if (pidH != 0) { /* código del padre */
         wait(NULL); /* espera a que su hijo termine */
         printf("\n Soy la ficha roja”);
   }else{ /* código del hijo*/
      pidN = fork(); /* el hijo crea un proceso hijo (Pn) */
       if (pidN != 0){ /* sigue siendo el hijo */
         wait(NULL); /* el hijo espera a que su hijo termine */
         printf("\n Soy la ficha verde");
         exit(0); /* indica que termina el hijo */
      }else{ /* código del proceso nieto  */
         printf("\n Soy la ficha negra”);
         exit(0); /* indica que termina el nieto */
      }
   }
}
```

Ejercicio 2.4.4 Elaborar un programa en C, con llamadas al sistema, que construya el siguiente árbol de procesos. El proceso padre deberá esperar a que sus dos procesos hijo terminen para que escriba su mensaje.

Cada proceso deberá escribir el mensaje indicado:



Pp → Soy la ficha roja
Ph1 → Soy la ficha verde
Ph2 → Soy la ficha azul

Respuesta:

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

main()
{
   int   pidP, pidH1, pidH2;

   pidH1 = fork(); /* Se crea el proceso hijo1 */
   if (pidH1 != 0) { /* código del padre */
      pidH2 = fork(); /* el padre crea el proceso hijo2 */
      if (pidH2 != 0) { /* sigue siendo el código del padre */
         wait(NULL); /* espera a que un hijo termine */
         wait(NULL); /* espera a que el otro hijo termine */
         printf("\n Soy la ficha roja");
      }else{ /* código del hijo2*/
         printf("\n Soy la ficha azul");
         exit(0); /* indica que termina el hijo 2 */
      }
   }else{ /* código del hijo1*/
      printf("\n Soy la ficha verde);
      exit(0); /* indica que termina el hijo 1 */
   }
}
```

Comunicación básica entre procesos.

Utilizando la dupla de llamadas al sistema wait()-exit() se puede hacer la comunicación entre un proceso hijo hacia su proceso padre.

Revisemos los argumentos de ambas llamadas:

- En wait(*stat_loc*), su argumento es un apuntador a entero (*int **) que ocupa 16 bits.
- En exit(*status*), su argumento es de tipo entero (*int*) que ocupa 8 bits.

Entonces, si se tiene el siguiente código:

```
int i=4,            /* en binario, 4 es 100 */
    estado;
    .
    .
/* código del padre */
    wait(&estado);
    .
    .
/* código del hijo */
    exit(i);        /* envía 4 en 8 bits: 00000100
```

Recibe en *estado* el valor de 4 en binario en 8 bits. Lo justifica a la izquierda ocupando 16 bits: 0000010000000000; por lo que este número es el 1024 decimal. Por lo tanto, para tener el valor que realmente envía la llamada exit(), se debe hacer un corrimiento de 8 bits a la derecha.

Revisa el siguiente programa que hace una comunicación básica de proceso hijo a proceso padre.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
main()
{
  int i= 10,
     j= 12,
     pid,
     estado;

   pid = fork();  /* Se crea el proceso hijo */
   if (pid) { /*código del padre*/
     printf("\n Soy el padre y el identificador de mi proceso hijo es:%d\n", pid);
     printf ("\n Ahora esperaré a que termine mi hijo");
     wait(&estado); /* recibe el valor enviado por el hijo en la variable estado */
     estado = estado >> 8; /* corrimiento de 8 bits necesario*/
     printf("\n el valor que me envió mi hijo es: %d", estado);
   }
   else{ /* código del hijo */
     i=i+j;
     exit(i); /* termina el hijo y envía el valor de la variable i al padre */
   }
}
```

Actividad 2.4.5 Crea el programa anterior; compílalo y ejecútalo. Dale diferentes valores al argumento de exit(), pueden ser una constantes entera o una variable entera. ¿Cuál es el valor más grande que puede enviar exit() para que en wait() se pueda recibir adecuadamente después de hacer el corrimiento?

Ahora ya sabemos cómo hacer una comunicación entre proceso hijo y el proceso padre, con el uso de wait()-exit(). Sin embargo, esta comunicación es muy precaria, sólo se puede hacer de hijo a padre y el valor enviado es un entero de 8 bits.

2.5 Concurrencia

En la administración de procesos en un sistema multitareas y/o multiusuarios, un tema de suma importancia es la **sincronización** que debe haber entre los procesos para su correcto procesamiento.

Esto es, cuando hay varios procesos que están compartiendo recursos tales como el procesador, la memoria, o los dispositivos de E/S, debe haber mecanismos que los coordinen para que no se produzcan fallas debidas a cómo y cuándo se asignan los recursos compartidos.

Los procesos que están conviviendo y compartiendo recursos de manera simultánea, se les llaman procesos **concurrentes**.

Podemos hablar de tres tipos de procesos concurrentes:

- **Totalmente independientes**. Cuando los procesos pueden ser ejecutados sin la dependencia de la ejecución de otros procesos.
- **Asíncronos**. Es cuando los procesos, en ocasiones, requieren cierta sincronización y cooperación.
- **Síncronos**. Cuando los procesos requieren una total sincronización para que puedan ejecutarse en tiempo y forma.

En un ambiente de procesos concurrentes hay que cuidar los siguientes aspectos:

- La comunicación entre procesos. La cual se debe realizar con el canal de comunicación adecuado para que la transferencia de información sea exitosa.
- La asignación de recursos. Debe realizarse bajo ciertos mecanismos para hacer una asignación justa.
- Que los procesos se ejecuten en cierto orden. Por ejemplo, si el proceso A requiere de un dato que genera el proceso B, entonces el proceso A debe esperar a que el proceso B le indique o le transfiera el dato que generó.

Hasta el momento hemos hecho comunicación entre procesos creados con la llamada al sistema fork( ), utilizando las llamadas al sistema wait( ) y exit( ) para una comunicación muy simple, o bien usando tuberías con la llamada al sistema pipe( ). Estas comunicaciones se realizan implícitamente de manera síncrona.

Entonces, para revisar las diversas situaciones que se presentan en procesos concurrentes, las cuales puedan propiciar fallas por falta de sincronización, generaremos un ambiente de muchos procesos que comparten variables globales; me refiero a los **hilos**.

Los hilos que aprenderemos a manipular son los hilos Posix (pthread).

Hilos Posix

POSIX es el acrónimo de Portable Operating System Interface. Históricamente, cada fabricante de hardware implementaba sus propios hilos y la forma de gestionarlos. Las implementaciones variaban notablemente, lo que dificultaba a los programadores desarrollar aplicaciones con hilos portables. Para aprovechar al máximo las bondades que ofrecen los hilos era necesario hacer una interfaz de programación estándar.

En 1995 se especificó la interfaz estándar IEEE 1003.1c. Las implementaciones basadas en este estándar son conocidas como hilos POSIX o Pthreads. Actualmente la mayoría de los fabricantes ofrecen en su hardware Pthreads, además de sus propias APIs (Intel, AMD). A pesar de que se creó en 1995, este estándar ha pasado por varios cambios y revisiones, la más reciente fue en el año 2008.

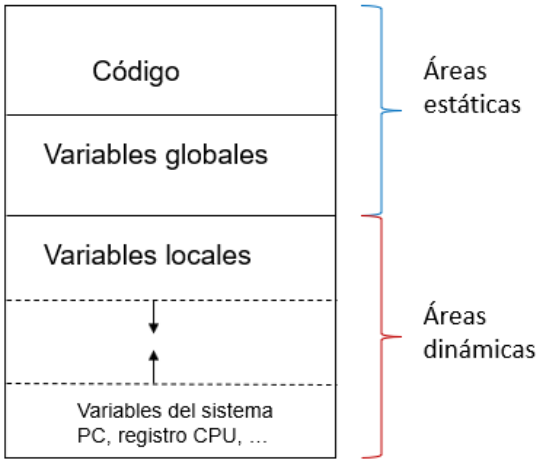
Los hilos permiten la ejecución concurrente de instrucciones asociadas a diferentes funciones dentro de un mismo proceso. Es decir, cuando se crea un hilo, se le indica qué función debe ejecutar, la cual está dentro del código del hilo o proceso que creó este nuevo hilo.

Los hilos Posix son *procesos ligeros* ya que no tienen un identificador de proceso (pid) que a través de él, el sistema operativo los pueda manipular. Esta manipulación es tarea del hilo que lo crea. Más adelante veremos cómo se realiza este manejo, quedando a la responsabilidad del programador.

Como los hilos son procesos ligeros, su estructura es la misma que la de los procesos:

Recordemos que las áreas del código y de variables globales mantienen su tamaño. Las áreas que crecen y decrecen a manera de pila, son las que almacenan las variables locales y las variables del sistema.

A diferencia de los procesos creados con fork( ) (donde hay procesos padre e hijo), cuando un proceso de un programa en ejecución crea un hilo, el programa en ejecución es el hilo principal y el hilo que se crea se llama hilo secundario.

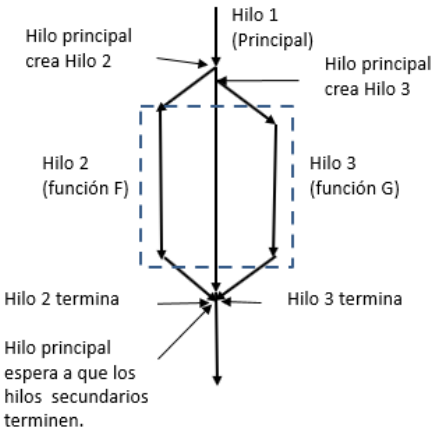


Entonces, en un ambiente de multihilos podemos tener cierta jerarquía: empezamos con el *hilo principal* que se genera cuando lanzamos a ejecución un programa, éste puede crear hilos, generándose *hilos secundarios*, que a su vez, éstos generen hilos que pudieran ser nombrados *hilos terciarios*, y así sucesivamente. Esquemáticamente lo podemos ver así:

En el esquema observamos que el hilo principal (Hilo 1) crea dos hilos secundarios (uno a la vez), nombrados Hilo 2 e Hilo 3.

Como lo había mencionado antes, al crear un hilo se indica qué función del código del hilo principal deberá ejecutar en primera instancia. Entonces en el esquema observamos que al crear el hilo 2, se le indica que debe ejecutar la función F y al hilo 3 debe ejecutar la función G.

Observemos que en el recuadro punteado, enmarca el momento en que se tienen tres hilos ejecutándose simultáneamente.



Veamos qué comparten y no comparten los hilos:

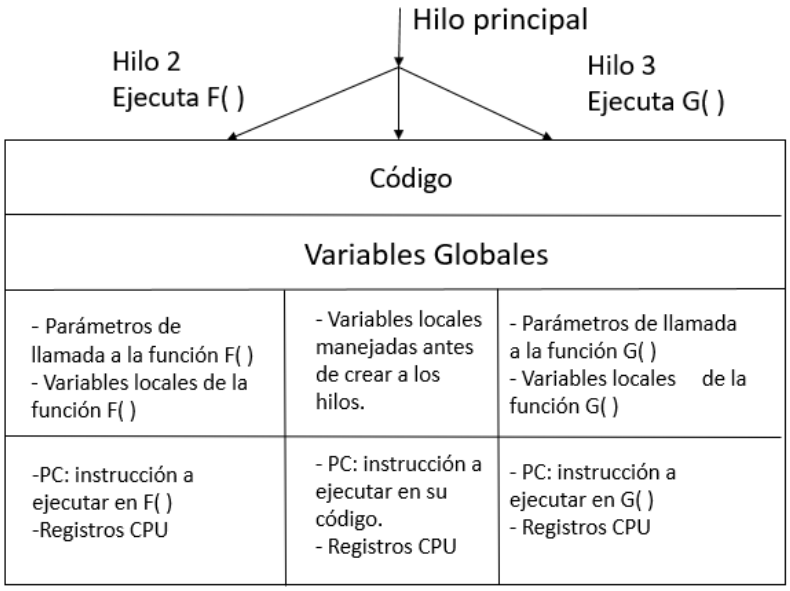
Los hilos **sí comparten** la misma imagen de memoria de:

- Código,
- Variables de memoria globales y
- Los dispositivos y archivos que tuviera abierto el hilo principal.

Los hilos **no comparten**:

- El Contador de Programa: cada hilo podrá ejecutar una sección distinta de código.
- Los registros de CPU.
- La pila en la que se crean las variables locales de las funciones a las que se va llamando después de la creación del hilo.
- El estado: puede haber hilos en ejecución, listos, o bloqueados en espera de un evento. Esto es muy importante hacerlo notar ya que en un sistema multihilo es un sistema de multitareas ya que los hilos pueden estar en cualquiera de los 3 estados básicos: ejecución, listo o bloqueados.

Veamos el siguiente diagrama de uso de memoria correspondiente al esquema anterior donde existe un hilo principal y dos hilos secundarios:



Características de los hilos Posix.

- Consumen menos memoria que los procesos creados con fork( ). Debido a que comparten, además del código y archivos abiertos, las variables globales en las mismas áreas de memoria. Y no hace copia de las variables locales del hilo principal porque sólo maneja las variables locales de la función que ejecutará al crearse.

- Cuesta menos crearlos. Ocupa menos tiempo, ya no tiene que esperar que el gestor de memoria busque memoria para copiar variables locales y globales de quien lo crea.

- Ya tienen disponible un mecanismo de comunicación entre ellos: las variables globales que son compartidas.

Si bien el uso de variables globales está totalmente desaconsejado en otras circunstancias, en concurrencia a través de hilos es el método idóneo para compartir información.

- Es más sencillo hacer un cambio de contexto entre hilos, lo que los hace ideales para la realización de tareas concurrentes cooperativas.

Pasemos ahora a crear ambientes de multitareas usando hilos Posix en lenguaje C.

Para ello necesitamos conocer qué bibliotecas, variables y funciones básicas utilizaremos en los programas:

- Deben incluirse los siguientes archivos de cabecera:

```
#include   <stdio.h>
#include   <pthread.h>
```

- Declarar, en la función main( ), tantas variables de tipo *pthread_t* como hilos se crearán. Por ejemplo:

  pthread_t hilo1, hilo2;

  Pero supóngase que se van a manejar 100 hilos para dividir el trabajo de sumar dos vectores de 1,000,000 de elementos, entonces en lugar de definir una variable por cada hilo, podemos definir un arreglo de 100 variables de tipo pthread_t:

  pthread_t hilo[100];

  En este caso se recomienda que los hilos realicen la misma función, pero con diferentes datos. Ya que, si no es así, habría que definir 100 funciones, lo que propiciaría la poca eficiencia del uso de hilos.

- Para crear un hilo, se usa la función *pthread_create( )* que está definida como sigue:

  int pthread_create (*pthread_t* *thread, *const pthread_attr_t* *attr,

  void *(*start_routine)(void*), *void* *arg);

  Donde:

  - El primer argumento *pthread_t* *thread, es la dirección de la variable de tipo *pthread_t* declarada en la función main( ) y va a contener el identificador del hilo.

  - El segundo argumento *const pthread_attr_t* *attr, define los atributos del hilo. Si attr vale NULL se crea con los atributos por defecto. Más adelante revisaremos qué constantes podemos poner en este argumento para definir atributos específicos.

  - El tercer argumento *void* *(*start_routine)(void*), es la función que el hilo ejecutará al momento de crearse. Por lo que en el programa (hilo principal) se deberá definir una función de tipo apuntador a void, con un solo argumento que será de tipo apuntador a void. Por ejemplo:

    void *codigo_del_hilo(void *arg);

    Cuando definimos una función o una variable de tipo void, hacemos que sean de tipo genérico, para después obtener su valor del tipo que deseemos.

  - El cuarto argumento void *arg, es el único argumento que puede recibir la función. Como es de tipo apuntador a void, nos da una gama amplia de tipos de argumento.

- Para indicar que el hilo secundario terminó, se debe usar la función *pthread_exit( )* al final de la ejecución de él. Será la última sentencia ejecutada por el hilo. Esta función está definida como sigue:

void pthread_exit (*void* *value_ptr)

Devuelve al hilo que esté esperando su terminación, los datos referenciados por el apuntador value_ptr.

Si no se usa esta función de forma explícita, entonces es llamada implícitamente cuando el hilo termina de ejecutar su función.

- La función *pthread_join( )* suspende la ejecución del hilo que la invoca y espera hasta la terminación del hilo con identificador thread. Esta función está definida como sigue:

  int pthread_join (*pthread*_t thread, *void* **value_ptr);

  El apuntador devuelto a este hilo, por la terminación del hilo que hizo la llamada a la función pthread_exit, es almacenado en la dirección referenciada por value_ptr.

Una vez que hemos conocidos estos elementos básicos de manejo de hilos Posix, realizaremos algunos programas.

```
/* Programa donde se genera un hilo secundario.  */
#include <pthread.h>
#include <stdio.h>
void *codigo_del_hilo(void *id)
{
  int  valor;
  valor = *(int *)id;
  printf("Soy el hilo, recibí como argumento el valor de: %d\n",valor);
  pthread_exit(NULL); /* el hilo secundario termina su ejecución */
}
main()
{
    pthread_t hilo;    /* la variable hilo será el identificador del hilo */
    int error;  /* se usa para recibir lo que devuelve la función de manejo de hilos  */
    int valArg=5;   /* valor que se enviará a la función del hilo como argumento */

    error = pthread_create(&hilo,NULL,codigo_del_hilo,&valArg);   /* se crea el hilo y se envía la
                                                                    dirección de  valArg como argumento */
    error = pthread_join(hilo, NULL); /* el hilo principal queda en espera a que su hilo termine */
    printf("\nHilo principal terminado\n");
}
```

Actividad 2.5.1 Transcribe el programa anterior. Compílalo enlazándolo con la biblioteca pthread: *gcc prog.c –lpthread,* y ejecútalo. Hazle algunos cambios probando el manejo de argumentos.

Una vez que conocemos cómo se crea un ambiente de multitareas y multihilos, revisemos qué situaciones de sincronización se presentan entre los procesos o hilos concurrentes.

- **Condición de carrera o competencia**.

La condición de carrera (race condition) ocurre cuando dos o más procesos o hilos acceden un **recurso compartido sin control**, es decir, bajo circunstancias en la que la sincronía de estos accesos no puede predecirse de forma práctica. Esto genera variaciones impredecibles en la salida que a menudo constituyen errores de operación y pueden corromper la información que se maneja.

Veamos un caso práctico, basándonos en el ejercicio 2.5.1:

Ejercicio 2.5.1 Elaborar un programa donde el hilo principal crea dos hilos secundarios: hilo1 e hilo2. Las tareas de cada hilo se definen a continuación.

- El hilo principal deberá definir un vector de 10 elementos enteros, creará dos hilos secundarios. Esperará a que terminen los dos hilos y escribirá el resultado del cálculo de cada hilo.
- El hilo secundario hilo1 calculará el elemento menor del arreglo.
- El hilo secundario hilo2 calculará el valor promedio de los elementos del arreglo.

Recordemos que en la solución 2, definimos 3 variables compartidas por los tres hilos y que se definieron como globales:

```
int vector[ ]= {23,15,42,3,72,38,12,91,57,64};  /* vector como variable global */
int elemMenor; /* variable global para obtener el resultado del hilo 1 */
float promedio; /* variable global para obtener el resultado del hilo 2 */
```

Estas tres variables son recursos compartidos, pero ¿se acceden sin control por los tres hilos? Se pudiera decir que no están controladas explícitamente porque cada hilo las puede acceder en cualquier momento. Sin embargo, no propician ningún error de operación por lo siguiente:

vector[ ] : Sólo lo consultan los tres hilos, es decir, ninguno lo modifica.
elemMenor: Sólo lo modifica el hilo 1; y el hilo principal sólo la escribe.
promedio: Sólo lo modifica el hilo 2; y el hilo principal sólo lo escribe.

De aquí podemos deducir que pudiera haber errores de operación sobre esas variables sólo cuando dos o más hilos las **modificaran** sin ningún control.

Para que se dé el caso, vamos a declarar como global a la variable ***i*** que la ocupan los hilos secundarios, dentro de la estructura ***for***, para realizar su respectivo cálculo. Como la variable ***i*** será global, la definiremos en mayúscula (***I***):

Ejercicio 2.5.3 Variante del ejercicio 2.5.1 agregando a la variable I como global para ser compartida con los dos hilos secundarios.

```
#include <pthread.h>
#include <stdio.h>

int vector[ ]= {23,15,42,3,72,38,12,91,57,64};  /* vector como variable global */
int elemMenor; /* variable global para obtener el resultado del hilo 1 */
float promedio; /* variable global para obtener el resultado del hilo 2 */
int I; /*variable global para usarla en los hilos secundarios dentro del for correspondiente*/
void *calcula_elem_menor(   );  /* prototipo de la función del hilo 1 */
void *calcula_promedio(  ); /* prototipo de la función del hilo 2 */

main()
{
    pthread_t  hilo1, hilo2;  /* se declaran los identificadores de los hilos */
    int error;

    error = pthread_create(&hilo1,NULL,calcula_elem_menor,NULL); /* se crea el hilo 1, no se
    envía argumento*/
    error = pthread_create(&hilo2,NULL,calcula_promedio,NULL); /* se crea el hilo 2, no se
    envía argumento */

    error = pthread_join(hilo1,NULL); /* se espera a que termine el hilo 1 */
    error = pthread_join(hilo2,NULL); /* se espera a que termine el hilo 2 */

    printf("Hilo 1 terminado, menor = %d\n",elemMenor);  /* Escribe el resultado del hilo 1 */
    printf("Hilo 2 terminado, promedio = %f\n",promedio); /* Escribe el resultado del hilo 2 */
}
```

Ahora mostremos el uso de la variable global I en la función de cada hilo.

```
void *calcula_elem_menor(  ) /* no recibe argumento */
{
  elemMenor=vector[0];
  for (I =1;I<10; I++)
    if (elemMenor > vector[I])
      elemMenor=vector[I];
  pthread_exit(NULL);
}
```

Se calcula el elemento menor directamente en la variable global elemMenor, usando la variable global I.

```
void *calcula_promedio(  ) /* no recibe argumento */
{
  int suma=0;

  for (I=0;I<10; I++)
    suma+=vector[I];
  promedio=suma/10.0;
  pthread_exit(NULL);
}
```

Se calcula el promedio directamente en la variable global promedio, usando la variable global I.

Actividad 2.5.5 Transcribe el programa del ejercicio 2.5.3, compílalo con la biblioteca pthread: gcc prog.c –lpthread . Ejecútalo varias veces; podrás observar que los resultados de salida varían. Da una conclusión de porqué varían los resultados.

- **Barreras**.

Son mecanismos que se requieren para que un proceso o hilo espere a que otro hilo o proceso termine su ejecución o cálculo, para que pueda continuar ya que necesita del dato calculado para realizar operaciones con él.

Un ejemplo muy sencillo es el siguiente: si el proceso A produce datos y el proceso B los imprime, B tiene que esperar hasta que A haya producido algunos datos antes de empezar a imprimir.

En nuestro curso, estas barreras las hemos aplicado usando la llamada al sistema wait() para el caso de procesos creados con fork(); para los hilos hemos usado la función pthread_join( ).

Actividad 2.5.6 De cualquiera de los programas de solución del ejercicio 2.5.1, las dos llamadas a la función pthread_join() de la función main(), conviértelas en comentario o quítalas. Compílalo con la biblioteca pthread: gcc prog.c –lpthread. Ejecútalo varias veces; podrás observar que los resultados de salida son erróneos; menciona por qué.

- **Regiones críticas y exclusión mutua**.

Muchas de las situaciones por las que hay problemas de condición de carrera es porque dos o más procesos o hilos acceden sin control a los datos compartidos que sean modificados por al menos un proceso o hilo. Para evitar las consecuencias de las condiciones de carrera antes descritas, emplearemos dos conceptos: región crítica y exclusión mutua.

Región o sección crítica: Es la parte del proceso o hilo en la que tiene acceso a la memoria o archivos compartidos que sean modificables. Por lo que para evitar condiciones de carrera dentro de la región crítica se debe cumplir:

- Cuando un proceso acceda a la región crítica, los demás no podrán acceder a ella hasta que lo permita el que lo accedió primero; sin embargo, estos procesos pueden seguir ejecutándose fuera de la región crítica.
- El proceso que accede a la región crítica entra en un estado especial y debe ejecutar rápidamente sus operaciones en esta región.
- No se debe bloquear el proceso que está en una región crítica; es decir no debe realizar una operación de E/S.

Exclusión mutua: La condición de exclusión mutua establece que solamente se permite a un proceso estar dentro de la misma *región crítica*. Esto es, que en cualquier momento solamente un proceso puede usar un recurso a la vez.

En la implementación de esta condición se emplean dos **primitivas** de exclusión mutua:

- **Primitiva de entrada**: Es aquella que controla que sólo un proceso o hilo entre a la región crítica.
- **Primitiva de salida**: Es aquella que le avisa a los demás procesos o hilos que el proceso o hilo que estaba en la región crítica, ya la desocupó.

Para la realización de dichas primitivas, según Edsger Dijkstra, se deben satisfacer las siguientes restricciones:

- Escribirse en software general.
- No pueden hacerse suposiciones acerca de las velocidades o el número de CPUs.
- Ningún proceso que se ejecute fuera de su región crítica puede bloquear otros procesos.
- Ningún proceso debe tener que esperar de manera indefinida para entrar a su región crítica.

Existen varios algoritmos y técnicas que implementan los mecanismos de la exclusión mutua: los algoritmos de Dekker y de Peterson, semáforos, monitores, candados. Revisaremos a detalle algunos de ellos.

Algoritmo de Dekker.

El algoritmo de Dekker es un algoritmo de programación concurrente para exclusión mutua, que permite a **dos procesos o hilos** de ejecución compartir un recurso sin conflictos. Fue uno de los primeros algoritmos de exclusión mutua inventados, implementado por Edsger Dijkstra.

Si ambos procesos intentan acceder a la sección crítica simultáneamente, el algoritmo elige un proceso según una variable de turno. Si el otro proceso está ejecutando en su sección crítica, deberá esperar su finalización.

Existen cinco versiones del algoritmo de Dekker, teniendo ciertos fallos los primeros cuatro. La versión 5 es la que trabaja más eficientemente, siendo una combinación de la 1 y la 4. Les presento algunos.

NOTA: Transcribí los algoritmos en lenguaje C. Algunas funciones sólo están indicadas y no implementadas, pero hacen alusión a lo que van a realizar. Las instrucciones que están encerradas entre < > son instrucciones que se deben realizar para creación de los hilos. Se emplean hilos para compartir la(s) variable(s) global(es) que se emplean para el control del acceso a la región crítica.

Algoritmo de Dekker Versión 1.

Garantiza la exclusión mutua usando la variable global *num_proc* en la implementación de las primitivas de entrada y salida. Tiene algunas desventajas que mencionaré más adelante.

```
int   num_proc;  /*variable global para el manejo de las primitivas de exclusión mutua */

void *proc_uno( )
{
  while (1) /* se ejecutará indefinidamente */
  {
       tareas_iniciales_uno();  /* operaciones fuera de la región crítica */
       while (num_proc == 2 );  /* espera activa (Primitiva de entrada) */
       region_critica_uno( );  /* entra a su región crítica */
       num_proc = 2;  /* da turno al proc_dos de entrar a la región crítica (Primitiva de salida) */
       otras_tareas_uno( );  /* operaciones fuera de la región crítica */
  }
}

void *proc_dos( )
{
  while (1) /* se ejecutará indefinidamente */
  {
       tareas_iniciales_dos();  /* operaciones fuera de la región crítica */
       while (num_proc == 1 );  /* espera activa (Primitiva de entrada) */
       region_critica_dos( );  /* entra a su región crítica */
```

```
        num_proc = 1;  /* da turno al proc_uno de entrar a la región crítica (Primitiva de salida) */
        otras_tareas_dos( );  /* operaciones fuera de la región crítica */
    }
}

main( )
{
        num_proc = 1;  /* se le da turno inicial al proc_uno */
        <crea proc_uno()>
        <crea proc_dos()>
}
```

Podemos observar que los dos procesos hilos secundarios se ejecutarán de forma indefinida, esto se hace para que ambos hilos estén trabajando de manera simultánea por mucho tiempo y se vea la competencia por entrar a la región crítica en varias ocasiones.

Revisemos algunas líneas del algoritmo de Dekker versión 1:

a) En la función main( )

```
num_proc = 1;  /* se le da turno inicial al proc_uno */
```

Como los dos hilos van a estar despachándose (entrar a estado de ejecución) sin un orden establecido, se le dará al proc_uno el turno de acceder primero a la región crítica aunque el proc_dos quiera entrar primero.

b) En las funciones de los hilos secundarios

```
while (num_proc == 1 );  /* espera activa (Primitiva de entrada del proc_dos) */
while (num_proc == 2 );  /* espera activa (Primitiva de entrada del proc_uno) */
```

Estas líneas están implementando una “espera activa”. Estas esperas activas actúan como primitivas de entrada de la siguiente manera: Supongamos que el proc_dos es el primero que quiere acceder a la región crítica, pero en ella está proc_uno por haberle dado el turno inicial, entonces el “while (num_proc == 1);” es verdadero y se queda en un ciclo hasta que el proc_uno cambie el valor de num_proc a 2, la cual es la primitiva de salida del proc_uno.

```
num_proc = 2;  /* da turno al proc_dos de entrar a la región crítica (Primitiva de salida del proc_uno) */
```

De manera análoga pasa con la espera activa del proc_uno: Si el proc_uno quiere acceder a la región crítica, pero en ella está proc_dos, entonces el “while (num_proc == 2);” es verdadero y se queda en un ciclo hasta que el proc_dos cambie el valor de num_proc a 1, la cual es la primitiva de salida del proc_dos.

```
num_proc = 1;  /* da turno al proc_dos de entrar a la región crítica (Primitiva de salida del proc_dos) */
```

Una vez que entendimos cómo funciona esta versión del algoritmo de Dekker, veamos qué desventajas tiene:

- Se tiene que indicar qué proceso entra por primera vez a la región crítica, sin importar si es el primero en ser despachado.
- Entran y salen de la región crítica de manera alternada, es decir primero entra al que se le dio su turno de primera vez, luego el otro, después el primero que entró, y así sucesivamente. No hay opción de que un mismo proceso entre dos veces seguidas a la región crítica, que es un aspecto que sí se puede dar.
- Si un proceso termina, el otro sólo puede entrar una vez más a la región crítica y se quedaría en espera activa de manera indefinida la siguiente ocasión que quiera acceder a la región crítica. Lo cual no cumple con la última restricción establecida por Dijkstra.

Es entonces que surge la segunda versión del Algoritmo de Dekker para solventar las desventajas de su primera versión.

Actividad 2.5.7 Analiza el Algoritmo de Dekker versión 1 y comprueba que tiene las tres desventajas arriba mencionadas.

Algoritmo de Dekker Versión 2.

Soluciona las desventajas de la versión 1, pero surge otro problema. Utiliza dos variables globales para indicar cuándo un hilo está en la región crítica.

```
int    proc1_entro, /* variable para indicar que el proc_uno está en la región critica */
       proc2_entro; /* variable para indicar que el proc_dos está en la región critica */

void *proc_uno( )
{
   while (1) /* se ejecutará indefinidamente */
   {
       tareas_iniciales_uno(); /* operaciones fuera de la región crítica  */
       while (proc2_entro == 1 ); /* espera activa del proc_uno*/
       proc1_entro = 1;  /* indica al  otro proceso que proc_uno está en la región crítica */
       region_critica_uno( );  /* entra a su región crítica */
       proc1_entro = 0;  /* da turno al proc_dos de entrar a la región crítica (Primitiva de salida) */
       otras_tareas_uno( );  /* operaciones fuera de la región crítica  */
   }
}

void *proc_dos( )
{
   while (1) /* se ejecutará indefinidamente */
   {
```

```
        tareas_iniciales_dos(); /* operaciones fuera de la región crítica  */
        while (proc1_entro == 1); /* espera activa del proc_dos */
        proc2_entro = 1;  /* indica al  otro proceso que proc_dos está en la región crítica */
        region_critica_dos( ); /* entra a su región crítica */
        proc2_entro = 0; /* da turno al proc_uno de entrar a la región crítica (Primitiva de salida) */
        otras_tareas_dos( ); /* operaciones fuera de la región crítica  */
    }
}
main( )
{
    proc1_entro = 0;  /* se le da valor inicial de 0 porque proc_uno no está en la región crítica */
    proc2_entro = 0;  /* se le da valor inicial de 0 porque proc_dos no está en la región crítica */
    <crea proc_uno()>
    <crea proc_dos()>
}
```

Revisemos algunas líneas del algoritmo de Dekker versión 2:

a) En la función main( )

```
proc1_entro = 0;  /* se le da valor inicial de 0 porque proc_uno no está en la región crítica */
proc2_entro = 0;  /* se le da valor inicial de 0 porque proc_dos no está en la región crítica */
```

Como los dos hilos van a estar despachándose (entrar a estado de ejecución) sin un orden establecido, ahora con estas variables se le permitirá el acceso a la región crítica al primer proceso que vaya a acceder a ella.

b) En las funciones de los hilos secundarios

```
while (proc2_entro == 1 ); /* espera activa del proc_uno*/
proc1_entro = 1;  /* indica al  otro proceso que proc_uno está en la región crítica */
while (proc1_entro == 1); /* espera activa del proc_dos */
proc2_entro = 1;  /* indica al  otro proceso que proc_dos está en la región crítica */
```

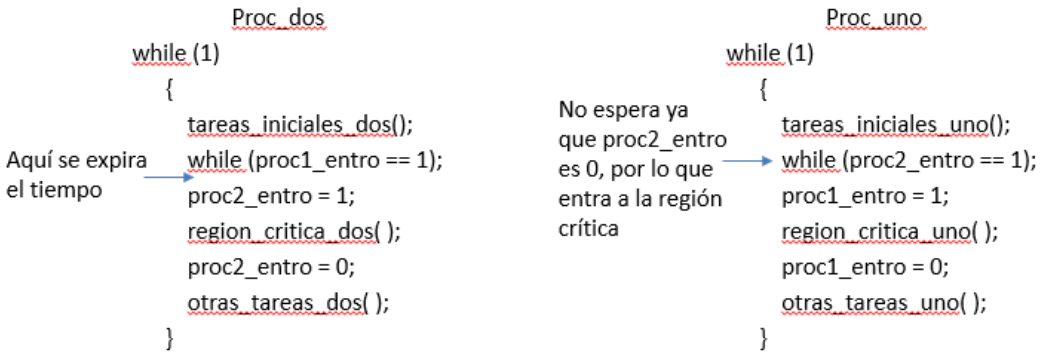
Estas líneas están implementando las **primitivas de entrada** de cada hilo, las cuales están compuestas por dos líneas: la espera activa y la de indicación de que el hilo está en la región crítica. De esta forma la región crítica es accedida por primera vez, por cualquier hilo.

Las primitivas de salida están dadas por las líneas:

```
proc1_entro = 0;  /* da turno al proc_dos de entrar a la región crítica (Primitiva de salida) */
proc2_entro = 0; /* da turno al proc_uno de entrar a la región crítica (Primitiva de salida) */
```

Veamos ahora cómo funciona esta versión del algoritmo de Dekker y encontremos el problema de concurrencia que se presenta.

- Como es un ambiente de multitareas o multihilos, puede suceder que, por ejemplo, el proc_dos al estar ejecutando su primitiva de entrada, no llega a terminarla porque se le expira el tiempo, sólo pudo ejecutar la espera activa y no realizó la asignación de 1 a proc2_entro, por lo que al entrar a ejecución el proc_uno y ejecutar su primitiva de entrada, como no se actualizó la variable proc2_entro hace que sí entre a su región crítica. Por lo que los dos hilos están al mismo tiempo en la región crítica. Entonces hay una colisión en la región crítica; **no garantiza** la exclusión mutua.



Algoritmo de Dekker Versión 3.

Esta versión utiliza dos variables para que cada hilo avise que va a entrar a la región crítica. Soluciona la colisión en la región crítica, pero surge otro problema.

```
int    proc1_solicita, /* para indicar que el proc_uno solicita entrar a la región critica */
       proc2_solicita; /* para indicar que el proc_dos solicita entrar a la región critica */

void *proc_uno( )
{
   while (1) /* se ejecutará indefinidamente */
   {
        tareas_iniciales_uno(); /* operaciones fuera de la región crítica  */
        proc1_solicita = 1;  /* indica al  otro proceso que proc_uno quiere entrar a región crítica */
        while (proc2_solicita == 1 ); /* espera activa del proc_uno */
        region_critica_uno( );  /* entra a su región crítica */
        proc1_solicita = 0;  /* da turno al proc_dos de entrar a región crítica (primitiva de salida) */
        otras_tareas_uno( );  /* operaciones fuera de la región crítica  */
   }
}
```

```
void *proc_dos( )
{
   while (1) /* se ejecutará indefinidamente */
   {
        tareas_iniciales_dos(); /* operaciones fuera de la región crítica  */
        proc2_solicita = 1;  /* indica al  otro proceso que proc_dos quiere entrar a región crítica */
        while (proc1_solicita == 1); /* espera activa del proc_dos */
        region_critica_dos( ); /* entra a su región crítica */
        proc2_solicita = 0; /* da turno al proc_uno de entrar a región crítica (Primitiva de salida) */
        otras_tareas_dos( ); /* operaciones fuera de la región crítica  */
   }
}

main( )
{
    proc1_solicita = 0;  /* se le da valor inicial de 0 porque proc_uno no está en la región crítica */
    proc2_solicita = 0; /* se le da valor inicial de 0 porque proc_dos no está en la región crítica */
    <crea proc_uno()>
    <crea proc_dos()>
}
```

Revisemos algunas líneas del algoritmo de Dekker versión 3:

a) En la función main( )

```
proc1_solicita = 0;  /* se le da valor inicial de 0 porque proc_uno no está en la región crítica */
proc2_solicita = 0; /* se le da valor inicial de 0 porque proc_dos no está en la región crítica */
```

Como los dos hilos van a estar despachándose (entrar a estado de ejecución) sin un orden establecido, con estas variables los hilos solicitarán entrar a la región crítica antes de la espera activa para evitar la colisión.

b) En las funciones de los hilos secundarios

```
proc1_solicita = 1;  /* indica al  otro proceso que proc_uno quiere entrar a la región crítica */
while (proc2_solicita == 1 ); /* espera activa del proc_uno */
proc2_solicita = 1;  /* indica al  otro proceso que proc_dos quiere entrar a la región crítica */
while (proc1_solicita == 1); /* espera activa del proc_dos */
```

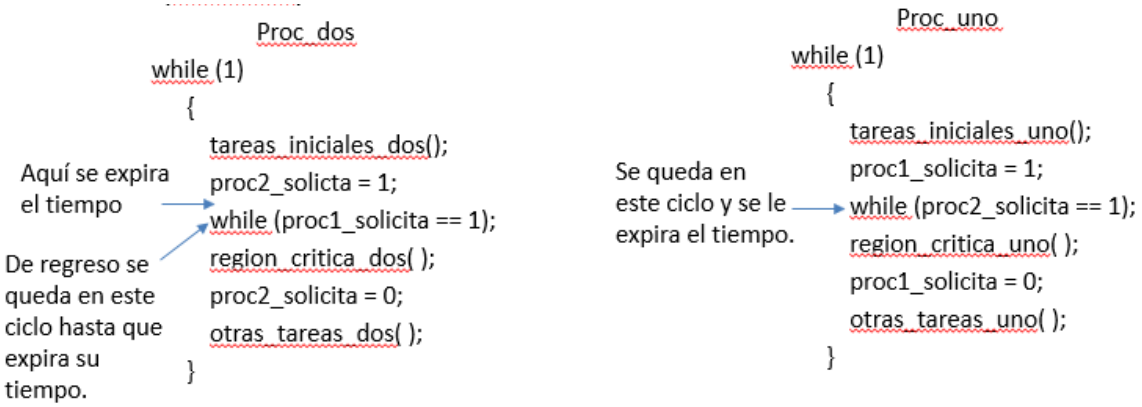
Estas líneas están implementando las **primitivas de entrada** de cada hilo, las cuales están compuestas por dos líneas: la de indicación de que el hilo quiere entrar a la región crítica y la espera activa. Como primero está la asignación de 1 a la variable que solicita entrar a la región crítica, se evita así la colisión en la región crítica.

Las primitivas de salida están dadas por las líneas:

```
proc1_solicita = 0;  /* da turno al proc_dos de entrar a la región crítica (Primitiva de salida) */
proc2_solicita = 0;  /* da turno al proc_uno de entrar a la región crítica (Primitiva de salida) */
```

Veamos cómo funciona la versión 3 del algoritmo de Dekker y encontremos el problema de concurrencia que se presenta.

- Siendo un ambiente de multitareas o multihilos, puede suceder que, por ejemplo, el proc_dos al estar ejecutando su primitiva de entrada, sólo ejecuta la asignación a la variable proc2_solicita y no ejecuta la espera activa porque se le expira el tiempo, por lo que al entrar a ejecución el proc_uno y ejecutar su primitiva de entrada, hace la asignación a proc1_solicita y se queda en el ciclo del while ya que proc_dos actualizó la variable proc2_solicita y ahí consume todo el tiempo. Entonces regresa a ejecución proc_dos y se queda en el ciclo while hasta terminar su tiempo. Por lo que los dos hilos que quedan es espera infinita. A esta situación se le llama "bloqueo mutuo" (deadlock).



Algoritmo de Dekker Versión 5 (final).

La versión 4 del algoritmo de Dekker mejora la versión 3 bajando la probabilidad de que ocurra un bloqueo mutuo, pero es hasta la versión 5 y final la que hace que funcionen de manera satisfactoria las primitivas de exclusión mutua.

```
int   proc1_solicita, /* para indicar que el proc_uno solicita entrar a la región critica */
      proc2_solicita, /* para indicar que el proc_dos solicita entrar a la región critica */
     proc_favorecido; /* en caso de cercanía a un bloqueo mutuo se favorece a este proceso */

void *proc_uno( )
{
   while (1) { /* se ejecutará indefinidamente */
        tareas_iniciales_uno(); /* operaciones fuera de la región crítica  */
        proc1_solicita = 1;  /* indica al  otro proceso que proc_uno quiere entrar a la región crítica */
        while (proc2_solicita == 1 )
             if (proc_favorecido == 2) {
                  proc1_solicita = 0;
```

```
                while (proc_favorecido== 2);  /* espera activa del proc_uno */
                proc1_solicita = 1 }
        region_critica_uno( );  /* entra a su región crítica */
        proc_favorecido = 2;
        proc1_solicita = 0;  /* da turno al proc_dos de entrar a la región crítica (Primitiva de salida) */
        otras_tareas_uno( );  /* operaciones fuera de la región crítica  */
  }
}

void *proc_dos( )
{
    while (1) { /* se ejecutará indefinidamente */
        tareas_iniciales_dos(); /* operaciones fuera de la región crítica  */
        proc2_solicita = 1;  /* indica al  otro proceso que proc_dos quiere entrar a la región crítica */
        while (proc1_solicita == 1 )
              if (proc_favorecido == 1) {
                proc2_solicita = 0;
                while (proc_favorecido== 1);  /* espera activa del proc_dos */
                proc2_solicita = 1 }
        region_critica_dos( ); /* entra a su región crítica */
        proc_favorecido = 1;
        proc2_solicita = 0; /* da turno al proc_uno de entrar a la región crítica (Primitiva de salida) */
        otras_tareas_dos( ); /* operaciones fuera de la región crítica  */
   }
}

main( )
{
        proc1_solicita = 0;  /* se le da valor inicial de 0 porque proc_uno no está en la región crítica */
        proc2_solicita = 0; /* se le da valor inicial de 0 porque proc_dos no está en la región crítica */
        proc_favorecido = 1;
        <crea proc_uno()>
        <crea proc_dos()>
}
```

Actividad 2.5.8 Analiza la versión 5 del algoritmo de Dekker e identifica:

- La primitiva de entrada de cada proceso. Describe su funcionamiento y justifica porqué no puede haber bloqueo mutuo ni colisión en la región crítica.
- La primitiva de salida. Describe su funcionamiento.

En el estudio de las primitivas de exclusión mutua, se desarrolló otro algoritmo que muestro a continuación.

Algoritmo de Peterson.

Peterson desarrolló el primer algoritmo (1981) para dos procesos que fue una simplificación del algoritmo de Dekker. Revisemos dicho algoritmo.

La solución está basada en utilizar variables para indicar qué proceso puede entrar, pero además usa un proceso *favorecido* para *desempatar* en caso de que ambos procesos busquen entrar a la vez. Si bien estas variables se emplean en el Algoritmo de Dekker, en el Algoritmo de Peterson, en cierto modo, el comportamiento es más *amable*: si un proceso detecta que el otro fue primero en actualizar el proceso favorecido, entonces lo deja pasar.

Al manejar así las variables de control de acceso a la región crítica, no es necesario darles un valor inicial en la función main().

Veamos el algoritmo e identifiquemos las primitivas de exclusión mutua y su funcionamiento.

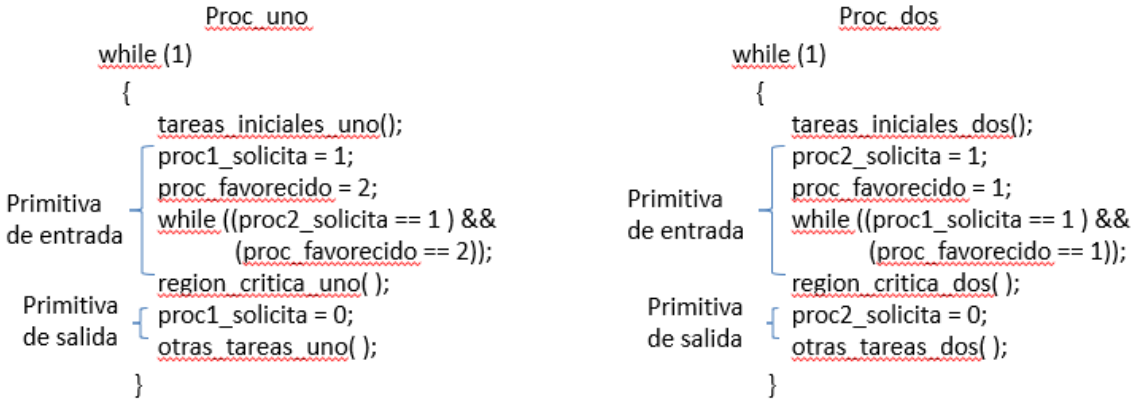
Algoritmo de Peterson.

```
int     proc1_solicita, /* para indicar que el proc_uno solicita entrar a la región critica */
        proc2_solicita, /* para indicar que el proc_dos solicita entrar a la región critica */
        proc_favorecido; /* en caso de cercanía a un bloqueo mutuo se favorece a este proceso */

void *proc_uno( )
{
   while (1) /* se ejecutará indefinidamente */
   {
        tareas_iniciales_uno(); /* operaciones fuera de la región crítica  */
        proc1_solicita = 1;  /* indica al  otro proceso que proc_uno quiere entrar a la región crítica */
        proc_favorecido = 2; /* favorece al otro proceso para desempatar el acceso a la vez */
        while ((proc2_solicita == 1 ) && (proc_favorecido == 2)); /* espera activa del proc_uno */
        region_critica_uno( );  /* entra a su región crítica */
        proc1_solicita = 0;  /* avisa que ya no está en la región crítica (Primitiva de salida) */
        otras_tareas_uno( );  /* operaciones fuera de la región crítica  */
   }
}
```

```
void *proc_dos( )
{
  while (1) /* se ejecutará indefinidamente */
  {
        tareas_iniciales_dos(); /* operaciones fuera de la región crítica  */
        proc2_solicita = 1;  /* indica al  otro proceso que proc_dos quiere entrar a la región crítica */
        proc_favorecido = 1; /* favorece al otro proceso para desempatar el acceso a la vez */
        while ((proc1_solicita == 1 ) && (proc_favorecido == 1)); /* espera activa del proc_dos */
        region_critica_dos( ); /* entra a su región crítica */
        proc2_solicita = 0; /* avisa que ya no está ocupando la región crítica (Primitiva de salida) */
        otras_tareas_dos( ); /* operaciones fuera de la región crítica  */
  }
}

main( )
{
      <crea proc_uno()>
      <crea proc_dos()>
}
```

Identifiquemos ahora las primitivas de exclusión mutua y su funcionamiento.

a) Las primitivas de exclusión mutua de cada proceso son:



b) Funcionamiento de las primitivas de entrada. Cada proceso primeramente, avisa que quiere entrar a la región crítica, posteriormente cede su favoritismo al otro proceso para que en la espera activa, si el otro proceso ya había avisado que quería entrar a la región crítica lo pueda hacer mientras éste se queda en espera.

c) Funcionamiento de las primitivas de salida. Simplemente cada proceso avisa que ya no está ocupando la región crítica.

Semáforos.

Dijkstra extrajo los conceptos fundamentales de la exclusión mutua en su concepto de *semáforos*. Un semáforo es una variable *protegida* cuyo valor sólo puede ser leído y alterado mediante las operaciones **P** y **V**, y una operación de asignación de valores iniciales que llamaremos **inicia_semaforo**. Estas operaciones son indivisibles, es decir, no se puede interrumpir su ejecución hasta terminarla.

Existen dos tipos de semáforos:

- *Binarios*. Los semáforos sólo pueden tener los valores de 0 o 1.
- *Contadores*. También llamados semáforos generales, pueden tener valores enteros no negativos; es decir valor de 0, 1, 2, 3 …

Ahora veamos cómo funcionan las operaciones sobre los semáforos.

- *Operación **P***.

P viene del vocablo holandés "*proberen te verlagen*" (intentar decrementar). Si el semáforo tiene un valor mayor que 0, lo decrementa, en caso contrario bloquea el proceso que lo llamó. Esta operación también es conocida como wait, acquire, down o lock.

Suponiendo que tenemos una variable de tipo semáforo nombrada S, la operación P sobre el semáforo S, escrita P(S), opera como sigue.

```
if (S > 0)
   S = S-1;
else
   esperar S;
```

Significa que el proceso continúa su ejecución y se decrementa el valor del semáforo

Significa que el proceso que hace la operación P(S) pasa a una cola de espera hasta que el semáforo le dé paso.

- *Operación **V***.

V viene del vocablo holandés "*verhoog*" (incrementar). Si hay algún proceso en la cola de espera del semáforo, deja pasar el que está en la cabecera; en caso contrario, incrementa el valor de la variable. Esta operación también es conocida como signal, release, up o post. La operación V sobre el semáforo S, escrita V(S), opera como sigue.

```
if (hay procesos en la cola de espera de S)
   deja que prosiga el que está en la cabecera de la cola
else
   S = S+1;
```

- *Semáforos en exclusión mutua***.**

La implementación del mecanismo de exclusión mutua se realiza usando P(S) como primitiva de entrada y V(S) como primitiva de salida, siendo S un semáforo binario. Por ejemplo, si en el algoritmo de Peterson usáramos un semáforo para implementar las primitivas de exclusión mutua, se vería así.

```
Proc_uno
while (1)
  {
     tareas_iniciales_uno();
     P(S);
     region_critica_uno( );
     V(S);
     otras_tareas_uno( );
   }
```

```
Proc_dos
while (1)
  {
     tareas_iniciales_dos();
     P(S);
     region_critica_dos( );
     V(S);
     otras_tareas_dos( );
   }
```

Previamente haber definido a la variable S como semáforo y haberle dado un valor inicial de 1 con la operación inicia_semaforo(S,1), en la función main(). ¿Porqué darle el valor inicial de 1? Para que sin importar qué proceso quiera entrar primero a la región crítica lo haga.

Actividad 2.5.9 ¿Qué sucedería si al semáforo S se le da el valor inicial de 0? ¿Se presentaría un problema de concurrencia? ¿Cuál?

- *Semáforos en el protocolo bloquear/despertar*

Otro caso de uso de semáforos binarios es para llevar a cabo el sencillo mecanismo de sincronización *bloquear/despertar* de dos procesos. Este protocolo se refiere a que un proceso desea que se le avise si ocurre un evento en específico, y que algún otro proceso sea capaz de detectar la ocurrencia de ese evento y le avise al primero. Veamos el algoritmo que realiza este protocolo usando un semáforo.

- Descripción del algoritmo bloquear/despertar.

Se define una variable *evento* de tipo semáforo (por lo pronto solo se indica así, después veremos el manejo de semáforos).

El proceso espera_evento ejecuta ciertas tareas iniciales antes de que ocurra el evento; después ejecuta P(evento) para esperar a que ocurra el evento.

Se ha asignado al semáforo el valor inicial 0 (por lo pronto solo se indica así, después veremos cómo se da valor inicial), de manera que el proceso debe esperar.

El proceso detecta_evento ejecuta V(evento) para señalar al otro proceso que ha ocurrido el evento. Esto permite que pueda continuar el proceso espera_evento (aunque el semáforo sigue en cero).

Algoritmo bloquear/despertar con semáforo.

```
semaforo evento; /* es pseudocódigo */

void *espera_evento( )
{
        tareas_iniciales_espera(); /* operaciones antes de que ocurra el evento  */
        P(evento);    /* espera a que ocurra el evento para continuar */
        otras_tareas_espera( );  /* operaciones después de ocurrido el evento */
}

void *detecta_evento( )
{
        tareas_iniciales_detecta(); /* operaciones que detecten que ocurrió el evento  */
        V(evento); /* avisa al otro proceso que ocurrió el evento */
        otras_tareas_detecta( ); /* operaciones después de ocurrido el evento  */
}

main( )
{
       inicia_semaforo(evento, 0);  /* se le da valor inicial al semáforo evento */
       <crea espera_evento()>
       <crea detecta_evento()>
}
```

Actividad 2.5.10 ¿Qué sucedería si al semáforo *evento* se le da el valor inicial de 1? ¿Se presentaría un problema de concurrencia? ¿Cuál?

Continuando con el manejo de semáforos, revisemos ahora otro tipo de semáforos:

- *Semáforos contadores*.

Los semáforos contadores son útiles cuando hay que asignar un recurso a partir de un banco de recursos idénticos. Veamos su funcionamiento:

- El semáforo tiene como valor inicial el número de recursos contenidos en el banco de recursos idénticos.
- Cada operación **P** decrementa en uno el semáforo, indicando que se ha retirado un recurso del banco y que lo está utilizando algún proceso.
- Cada operación **V** incrementa en uno el semáforo, lo que indica la devolución de un recurso al banco y que el recurso está disponible para ser asignado a otro proceso.
- Si se intenta una operación **P** cuando el semáforo tiene valor 0, el proceso deberá esperar hasta que se devuelva un recurso al banco mediante una operación **V**.

Hagamos un ejercicio para entender su uso.

Ejercicio 2.5.4 Un cierto sistema cuenta con un banco de 3 recursos idénticos que son compartidos por varios procesos. Indica, de acuerdo con las solicitudes (P) y liberaciones (V) de recursos que van realizando los procesos, el valor que va obteniendo el semáforo R, así como la situación de la cola de procesos en espera a que se libere un recurso.

Respuesta

inicia_semáforo(R,3) ⟹ Se inicia con el valor de 3 porque el banco es de 3 recursos

Proceso	Operación	Valor de R	Situación cola	Comentarios
A	P(R)	2	vacía	Se asigna un recurso a A y quedan 2 libres
D	P(R)	1	vacía	Se asigna un recurso a D y queda 1 libre
C	P(R)	0	vacía	Se asigna un recurso a C y no hay libres
B	P(R)	0	B	Como no hay recursos, B entra a la cola
E	P(R)	0	E, B	Como no hay recursos, E entra a la cola
D	V(R)	0	E	D libera recurso y se le asigna a B

Actividad 2.5.11 Un cierto sistema cuenta con un banco de 4 recursos idénticos que son compartidos por varios procesos. Indica, de acuerdo con las solicitudes (P) y liberaciones (V) de recursos que van realizando los procesos, el valor que va obteniendo el semáforo R, así como la situación de la cola de procesos en espera a que se libere un recurso.

inicia_semáforo(R,)

Proceso	Operación	Valor de R	Situación cola	Comentarios
D	P(R)			
A	P(R)			
B	P(R)			
A	V(R)			
E	P(R)			
C	P(R)			
D	V(R)			

F	P(R)			
G	P(R)			

- *Semáforos en la relación productor-consumidor.*

La relación productor-consumidor es un ejemplo clásico de sincronización de procesos. Consiste en que un proceso, el *productor*, genera información que utiliza un segundo proceso, el *consumidor*.

- Descripción del algoritmo productor-consumidor y posibles problemas de concurrencia.

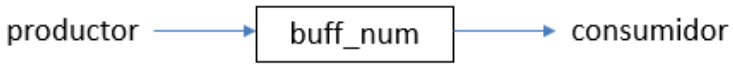
Supongamos que la información compartida es por una variable entera global *buffer_num*, entonces el productor realiza algunos cálculos y escribe el resultado en dicha variable *buffer_num*, el consumidor va leyendo los datos de dicha variable y los imprime.

Si cada vez que el productor deposita un resultado en *buffer_num* el consumidor lo lee y lo imprime de inmediato, entonces la salida impresa representará fielmente la secuencia de números generados por el productor.

Pero supóngase que las velocidades de los procesos son muy distintas. Si el consumidor opera más rápido que el productor, podría leer e imprimir el mismo número dos veces (o muchas) antes de que el productor deposite el siguiente número.

Si el productor opera más rápido que el consumidor, podría escribir sobre su resultado anterior antes de que el consumidor tuviera oportunidad de leerlo e imprimirlo; de hecho, un productor rápido podría realizar esto muchas veces, con lo cual se perderían muchos resultados.

Por lo que el comportamiento deseado aquí es que el productor y el consumidor coopere de la manera que los datos escritos en *buff_num* no se pierdan ni se dupliquen.



- Implementación del algoritmo con semáforos

Se emplean dos semáforos: el productor indica *num_depositado* (con V) y el consumidor lo verifica (con P); el consumidor no puede proseguir en tanto no se deposite un número en *buffer_num*.

El consumidor indica *num_recuperado* con una operación V y el productor lo verifica con una operación P; el productor no puede proseguir hasta que se haya recuperado un número de *buffer_num*.

Los valores iniciales de los semáforos obligan al productor a depositar un valor en *buffer_num* antes de que el consumidor pueda proseguir.

Algoritmo Productor-Consumidor.

```
int       buffer_num; /* variable global compartida */
semaforo  num_depositado, /* para indicar que el dato está depositado en el buffer*/
          num_recuperado; /* para indicar que el dato ha sido leído del buffer */

void *productor( )
{
   int datoAdepositar;
   while (1) /* se ejecutará indefinidamente */
   {
        calcula_dato_a_depositar();
        P(num_recuperado);
        buffer_num = datoAdepositar;
        V(num_depositado);
   }
}

void *consumidor( )
{
   int datoRecuperado;
   while (1)    /* se ejecutará indefinidamente */
   {
        P(num_depositado);
        datoRecuperado = buffer_num;
        V(num_recuperado);
        escribir(datoRecuperado);
   }
}

main( )
{
        inicia_semaforo(num_depositado, 0);
        inicia_semaforo(num_recuperado, 1);
        <crea productor>
        <crea consumidor>
}
```

Actividad 2.5.12 Analiza el algoritmo Productor-Consumidor e identifica:

- La región crítica de cada proceso.
- La primitiva de entrada de cada proceso. Describe su funcionamiento.
- La primitiva de salida de cada proceso. Describe su funcionamiento.
- La razón de los valores iniciales de los semáforos.

Bloqueos mutuos

Uno de los problemas que se presentan en concurrencia y que ha merecido mucho estudio, son los *bloqueos mutuos,* también nombrados *interbloqueos, abrazos mortales o deadlock*. Es por eso que los revisaremos a detalle.

Vimos que en el Algoritmo de Dekker versión 3 se podía presentar un bloqueo mutuo al quedar ambos procesos en su correspondiente espera activa. Esta situación puede suceder cuando dos o más procesos están compartiendo recursos y ninguno pueden continuar su proceso porque suceden las:

- *Condiciones de Coffman*

En un artículo escrito en 1971 por Edward. G. Coffman, describe cuatro condiciones que deben cumplirse simultáneamente y que no son totalmente independientes entre ellas, para que ocurra un bloqueo mutuo.

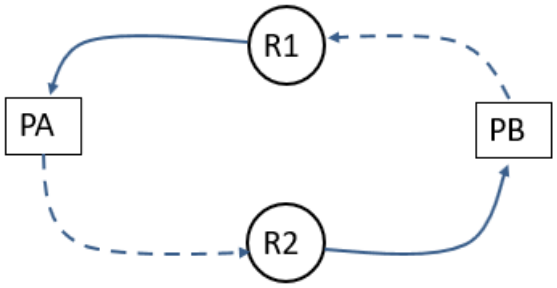
- **Exclusión mutua**. Los procesos reclaman acceso exclusivo de los recursos.
- **Retención y espera**. Los procesos mantienen los recursos que ya les habían sido asignados mientras esperan recursos adicionales.
- **No apropiatividad**. Los recursos no pueden ser arrebatados de los procesos que los tienen hasta su completa utilización.
- **Espera circular.** Existe una cadena circular de procesos en que cada uno mantiene a uno o más recursos que son requeridos por el siguiente en la cadena.

- *Bloqueo Mutuo Simple*.

Un bloque mutuo simple se da cuando dos procesos (PA y PB) compiten por dos recursos (R1 y R2), que solo pueden ser utilizados por un proceso a la vez y se presentan la siguiente secuencia de acciones:

1. PA solicita el recurso R1 de forma exclusiva; como está libre, se le asigna.
2. PB solicita el recurso R2 de forma exclusiva; como está libre, se le asigna.
3. Luego, PA solicita el recurso R2 ya asignado a PB; entonces queda en espera de que PB lo libere.
4. PB, que sigue ejecutándose, solicita el recurso R1; como PA tiene asignado el recurso R1, PB se queda en espera a que R1 sea liberado por PA.
5. Los procesos se quedan en espera circular.

De forma esquemática, esta situación se representa así



Nota: Las flechas sólidas representan que el recurso ha sido asignado al proceso. En tanto las flechas punteadas representan la solicitud que hace un proceso de un recurso.

Bloqueo mutuo simple

- *Estrategias para resolver bloqueos mutuos*.

Los resultados de la investigación sobre el bloqueo mutuo han sido satisfactorios en cuanto a que se han encontrado métodos limpios y rápidos para manejar la mayoría de los problemas más comunes. Existen cuatro áreas de interés relacionadas con los interbloqueos que pueden resumirse como prevención, técnicas para evitarlos, detección y recuperación de los mismos.

- **Prevención**. En la prevención del bloqueo mutuo interesa ajustar el sistema para *eliminar toda posibilidad de que ocurra*. La prevención suele funcionar pero sus métodos ocasionan, en general, un aprovechamiento pobre de los recursos. No obstante, estos métodos se utilizan con frecuencia.

- **Evitar**. Las técnicas que tienen como objetivo evitar el interbloqueo imponen *condiciones menos restrictivas* que en la prevención, para tratar de obtener un *mejor aprovechamiento de los recursos*. No elimina como las técnicas de prevención todas las posibilidades de que se produzca un bloqueo mutuo, pero se esquiva cuanto está a punto de suceder.

- **Detección**. Los métodos de detección del interbloqueo se utilizan en sistemas que permiten la ocurrencia de los mismos, ya sea de manera voluntaria o involuntaria. Su objetivo es determinar si ha ocurrido un bloqueo mutuo y saber exactamente cuáles son los procesos y recursos implicados en él.

- **Recuperación**. Los métodos de recuperación están íntimamente ligados a los de detección. Sirven para eliminar los interbloqueos detectados en un sistema para poder seguir trabajando y para que los procesos implicados puedan terminar su ejecución y liberen sus recursos. La recuperación es un problema complejo, en el mejor de los casos, los sistemas se recuperan de un bloqueo mutuo eliminando completamente uno o varios de los procesos implicados. Después, se inician de nuevo los procesos eliminados, perdiéndose la mayor parte o todo el trabajo previo realizado por el proceso.

- *Prevención de bloqueos mutuos.*

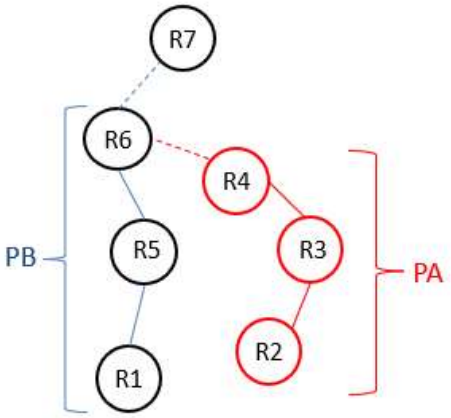
La estrategia empleada con más frecuencia por los diseñadores para tratar el bloqueo mutuo es la prevención. Examinaremos los métodos de prevención, junto con los efectos que tienen sobre los usuarios y los sistemas, sobre todo desde la perspectiva del rendimiento. *Havender* definió 3 estrategias independientes:

1. Cada proceso deberá pedir todos sus recursos al mismo tiempo y no podrá seguir la ejecución hasta haberlos recibido todos; esto hace que pueda quedarse en espera indeterminada (inanición).

2. Si a un proceso que tiene recursos se le niegan los demás, ese proceso deberá liberar sus recursos y, en caso necesario, pedirlos de nuevo junto con los recursos adicionales. Esto implica que dicho proceso pueda estar reiniciándose con frecuencia.

3. Se impondrá un ordenamiento lineal a los recursos. Los procesos irán solicitando recursos cuyo ordenamiento sea creciente. Esto se realiza de la siguiente forma:

   - Todos los recursos se numeran globalmente, siguiendo un orden creciente del 1 al número de recursos compartidos.
   - Los procesos pueden solicitar los recursos en cualquier momento, según un cierto orden numérico (creciente) de recurso; debido a lo cual la gráfica de asignación de recursos no tendrá ciclos.
   - En cada instante uno de los recursos asignados tendrá el número más grande, y así, el proceso que lo posea no pedirá un recurso ya asignado.
   - El proceso terminará o solicitará recursos con números mayores, que estarán disponibles:
     - ✓ Al concluir liberará sus recursos.
     - ✓ Otro proceso tendrá el recurso con el número mayor y también podrá terminar.
     - ✓ Todos los procesos podrán terminar y no habrá bloqueo.



Proceso A tiene asignados los recursos R2, R3 y R4 y solicita R6, pero deberá esperar a que el PB lo libere. No hay espera circular.

Esta estrategia de Havender tiene las siguientes desventajas: a) Es un ordenamiento demasiado estricto para muchas situaciones del mundo real y b) Lleva a los procesos a acaparar recursos de baja jerarquía.

Actividad 2.5.13 Realiza un análisis comparativo de las 3 estrategias de Havender de prevención de bloqueos mutuos. Compara en cuanto a mejor aprovechamiento de recursos, presencia de problemas de inanición (espera indeterminada), acercamiento al mundo real, etc. Justificándolo.

- *Evitar bloqueos mutuos*

Aún presentándose las condiciones para un bloqueo mutuo, todavía es posible evitarlo mediante una asignación cuidadosa de los recursos. Tal vez el algoritmo más famoso para evitar el interbloqueo sea el *algoritmo del banquero* de Dijkstra (73), el cual lo revisaremos a continuación.

Algoritmo del Banquero

En principio, este algoritmo supone que todos los recursos son del mismo tipo. Considérese la asignación de una cantidad *n* de recursos idénticos. Entonces el proceso es el siguiente:

- Un sistema operativo comparte un número fijo *n*, de recursos entre un número fijo de *p*, de procesos.
- Cada proceso especifica por adelantado el número máximo de recursos que necesitará durante su ejecución.
- El sistema operativo aceptará la petición de un usuario si la necesidad máxima de ese proceso no es mayor que *n*.
- Un proceso puede obtener o liberar recursos uno a uno.
- Algunas veces un usuario puede verse obligado a esperar para obtener un recurso adicional, pero el sistema operativo garantiza una espera finita.
- El número real de recursos asignados a un proceso nunca será superior a la necesidad máxima declarada por ese usuario.
- Si el sistema operativo es capaz de satisfacer la necesidad máxima del proceso, entonces éste debe garantizar al sistema operativo que los recursos serán utilizados y liberados en un tiempo finito.

Se dice que el estado del sistema es **seguro** si el sistema operativo puede garantizar que todos los procesos terminan en un tiempo finito. En otro caso, el sistema está en un estado **inseguro**.

Sea préstamo (i) la representación del préstamo actual de recursos para el proceso i. Sea máx(i) la necesidad máxima de recursos de un proceso y, por último, sea petición (i) la petición actual del usuario, que es igual a su necesidad máxima menos el préstamo actual. Por ejemplo, el proceso 7 tiene una necesidad máxima de 6 recursos y un préstamo actual de 5, entonces tiene

petición(7) = máx(7) - préstamo(7) = 6 - 5 = 2

El sistema operativo controla *n* recursos. Sea *d* el número de recursos todavía disponibles para asignar. Entonces *d* es igual a *n* menos la suma de los préstamos de los usuarios.

El algoritmo del banquero permite la asignación de recursos a los usuarios solamente cuando la asignación conduzca a *estados seguros*, y *no a estados inseguros*. Un *estado seguro* es una situación tal en la que todos los procesos son capaces de terminar en algún momento. Un *estado inseguro* es aquel en el cual puede presentarse un bloqueo mutuo.

Veamos los siguientes casos que se presentan en un determinado momento trabajando con este algoritmo.

- *Caso 1.* Supóngase que un sistema tiene doce recursos y tres procesos que los comparten.

	Préstamo actual	Necesidad Máxima
Proceso A	1	4
Proceso B	4	6
Proceso C	5	8
Disponibles: 12-10 = 2		

La tabla anterior representa un **estado seguro** porque el proceso B tiene un préstamo de 4 recursos y necesita como máximo 6, o sea, 2 más. El sistema tiene 12 de los cuales 10 están en uso y mantiene 2 disponibles. Si los que están disponibles se asignan al proceso B, cubriendo su demanda máxima, este proceso podrá terminar. Al acabar, devolverá todos los 6 recursos, y el sistema podrá asignarlos al proceso A y al C. De esta forma, la clave de un estado seguro es que exista al menos una forma en la que terminen todos los procesos.

- *Caso 2.* Supóngase el mismo sistema del caso 1 que tiene doce recursos y tres procesos que los comparten.

	Préstamo actual	Necesidad Máxima
Proceso A	8	10
Proceso B	2	5
Proceso C	1	3
Disponibles: 12-11 = 1		

Ahora 11 de los 12 recursos están asignados y solamente hay uno disponible. En este momento, no se puede garantizar que terminen los tres procesos. Si el proceso A pide y

obtiene el último recurso y los tres vuelven a solicitar un recurso más se produciría un bloqueo triple. Por lo que es un **estado inseguro**.

Es importante señalar, que un estado inseguro no implica la existencia, ni siquiera eventual, de un interbloqueo. Lo que sí implica un estado inseguro es la posibilidad de que ocurra por una desafortunada secuencia de eventos.

- *Caso 3.* Transición de estado seguro a estado inseguro. Saber que un estado es seguro no implica que serán seguros todos los estados futuros. La política de asignación de recursos debe considerar cuidadosamente todas las peticiones antes de satisfacerlas. Por ejemplo supongamos la situación que se muestra en la siguiente tabla en un sistema con 10 recursos.

	Préstamo actual	Necesidad Máxima
Proceso A	2	9
Proceso B	4	6
Proceso C	2	9
Disponibles: 10-8 = 2		

Vemos que los dos recursos que están disponibles se le pueden asignar al Proceso B para que cubra su necesidad máxima y pueda terminar, y así liberar los 6 recursos que le fueron asignados. Sin embargo, estos 6 recursos disponibles no cubren la necesidad máxima si se los diéramos al Proceso A o al Proceso C. Por lo que el **estado es inseguro**.

Entonces hay que ver más allá de que se pueda cubrir la necesidad máxima de un proceso.

Ejercicio 2.5.5 En cierto momento, la situación de asignación de recursos en un sistema de 12 recursos idénticos se muestra en la siguiente tabla:

	Préstamo actual	Necesidad Máxima
Proceso A	2	5
Proceso B	2	4
Proceso C	4	6
Proceso D	2	5
Disponibles: 12-10 = 2		

Observamos que está en un **estado seguro**, porque los dos recursos disponibles se le pueden asignar al proceso B o al proceso C para cubrir su necesidad máxima y cuando terminen, los recursos liberados se les pueden asignar a los procesos restantes.

Recordemos que para hacer la asignación de recursos idénticos a los procesos que los comparten, se usan *semáforos contadores*. Supongamos que usamos el semáforo R, para indicar los recursos disponibles, que la operación P(R) es para solicitar un recurso y la operación V(R) para liberar un recurso; ve indicando la situación de la asignación de recursos para que el sistema **esté siempre en estado seguro**, de acuerdo a la siguiente secuencia de operaciones de los procesos.

Respuesta

El valor del semáforo R es de 2 porque el semáforo indica el número de recursos disponibles (esa es su función).

Proceso	Operación	Valor de R	Cola	Comentarios
A	P(R)	2	A	No se le da recurso porque se volvería estado inseguro.
D	P(R)	2	D,A	No se le da recurso porque se volvería estado inseguro
C	P(R)	1	D,A	Se asigna un recurso a C y mantiene estado seguro
B	P(R)	1	B,D,A	No se le da recurso porque se volvería estado inseguro
C	P(R)	0	B,D,A	Se asigna un recurso a C y cubre su necesidad máxima
C	Termina	6	B,D,A	Al terminar C libera 6 recursos. Ahora se revisa la cola y se ve si puede dársele un recurso a los procesos que están en la cola, de acuerdo al orden de entrada.
		5	B,D	Se asigna un recurso a A porque se mantiene estado seguro
		4	B	Se asigna un recurso a D porque se mantiene estado seguro
		3	vacía	Se asigna un recurso a B porque se mantiene estado seguro

Actividad 2.5.14 Partiendo del estado final del ejercicio 2.5.5, llena la tabla siguiente de acuerdo a la secuencia de operaciones de los procesos, cuidando que siempre esté en estado seguro.

Proceso	Operación	Valor de R	Cola	Comentarios
A	P(R)			
B	P(R)			
D	P(R)			
B	Termina			
A	P(R)			

- Detectar bloqueos mutuos.

La detección del bloqueo mutuo es el proceso de determinar si realmente existe un interbloqueo e identificar los procesos y recursos implicados en él. Los algoritmos de detección determinan por lo general si existe una espera circular.

Para facilitar la detección de interbloqueos, utilizaremos una notación en la que un grafo dirigido indica las asignaciones y peticiones de recursos. Los cuadrados representan procesos; los círculos grandes, clases de recursos idénticos; los círculos pequeños en el interior de los grandes indican el número de recursos de cada clase.

Por ejemplo, si un círculo grande etiquetado como R1 contiene tres círculos pequeños, significa que hay tres recursos del tipo R1 para asignárseles a los procesos.

De forma esquemática, la asignación y solicitud de recursos se representa de la siguiente forma:

Nótese que cuando un proceso solicita un recurso, la punta de la flecha se queda en la circunferencia mayor. Cuando un recurso es asignado a un proceso, la flecha tiene su origen de la circunferencia pequeña.

Una vez que entendemos cómo se representa la asignación y solicitud de recursos en un grafo dirigido, una de las técnicas para detección de bloqueos mutuos es a través de la reducción del grafo, la cual veremos a continuación.

Reducción de grafos

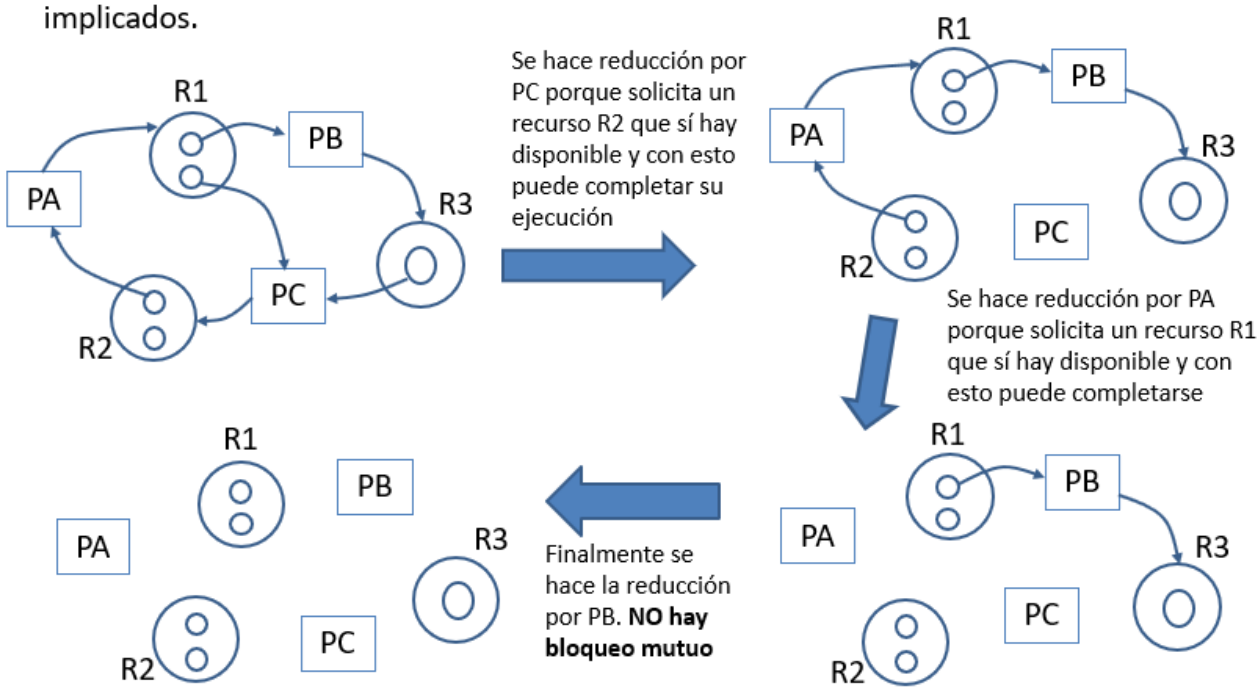
Es una técnica que consiste en ir reduciendo una gráfica determinando los procesos que pueden completar su ejecución:

- Si pueden atenderse las peticiones de recursos de un proceso, se dice que la gráfica puede ser reducida por ese proceso.
- Esta reducción es equivalente a mostrar la gráfica como si el proceso hubiese acabado y hubiera devuelto los recursos al sistema.

- La reducción se efectúa eliminando las flechas de los recursos hacia ese proceso y eliminando las flechas de ese proceso hacia los recursos.
- Si una gráfica puede ser reducida por todos sus procesos, entonces no hay interbloqueo.
- Si una gráfica no puede ser reducida por todos sus procesos, los procesos irreductibles constituyen el conjunto de procesos en bloqueo mutuo de la gráfica.

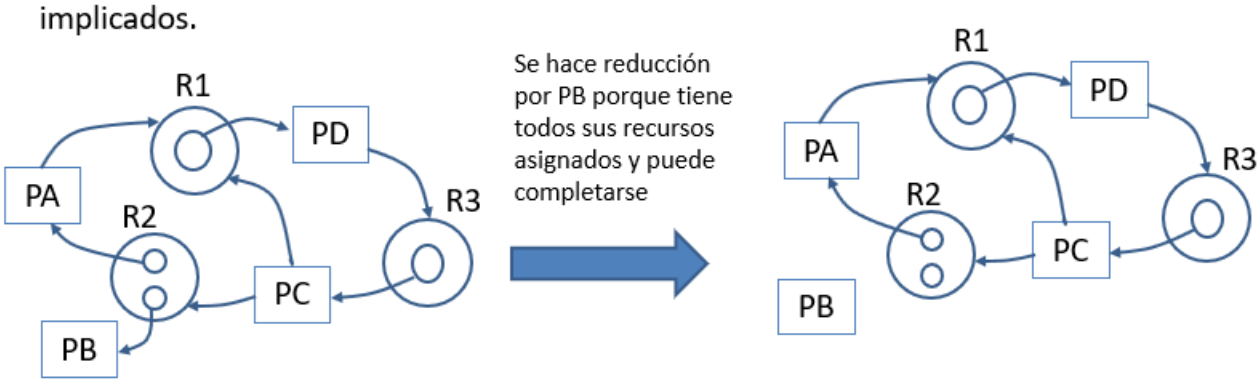
Ejercicio 2.5.6

Para el siguiente grafo de asignación y solicitud de recursos, aplica la técnica de reducción e indica si hay bloqueo mutuo y cuáles procesos son los que están implicados.



Ejercicio 2.5.7

Para el siguiente grafo de asignación y solicitud de recursos, aplica la técnica de reducción e indica si hay bloqueo mutuo y cuáles procesos son los que están implicados.

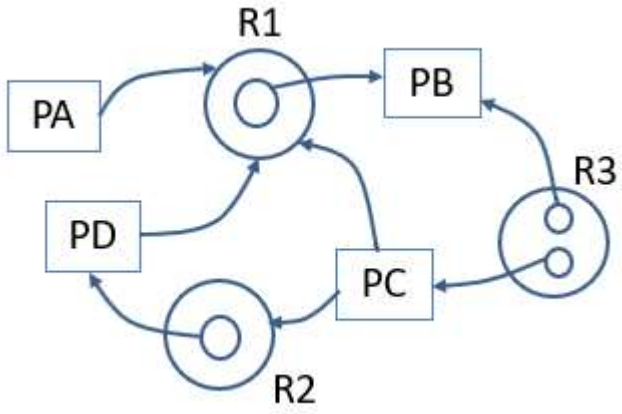


Podemos observar que ya no se puede reducir más el grafo ya que, aunque se le pueda asignar un recurso R2 a PC, éste está solicitando un recurso R1 que no se le puede asignar.

Entonces **SÍ hay bloqueo mutuo** y los procesos implicados son PA, PC y PD.

Actividad 2.5.15

Para el siguiente grafo de asignación y solicitud de recursos, aplica la técnica de reducción e indica si hay bloqueo mutuo y qué procesos son los que están implicados.



- Recuperación ante un bloqueo mutuo

En los sistemas actuales la recuperación se suele realizar eliminado un proceso y arrebatándole sus recursos. Por lo general, el proceso eliminado se pierde pero ahora es posible concluir los procesos restantes. Algunas veces es necesario eliminar varios procesos hasta que se hayan liberado los recursos suficientes para que terminen los procesos restantes.

Básicamente hay dos técnicas de recuperación:

a) Los procesos pueden eliminarse de acuerdo con algún orden de *prioridad*. La prioridad es un valor que se le puede asignar a un proceso y de acuerdo a su valor puede tener mayor o menor jerarquía ante otros procesos. Entonces puede eliminarse el proceso con menor jerarquía entendiéndose que su eliminación no es tan grave y se puede iniciar después.

b) Aplicando un mecanismo efectivo de suspensión/reanudación, implicaría suspender temporalmente los procesos y reanudarlos después sin pérdida de trabajo productivo. Para ello sería deseable la posibilidad de especificar puntos de verificación/reinicio. De este modo, se facilita la suspensión/reanudación, que se hará desde el último punto de verificación (es decir, la última grabación del estado del sistema).

Pero muchos sistemas de aplicación se diseñan sin aprovechar las ventajas de las funciones de punto de verificación/reinicio. Por lo general, se requiere un esfuerzo consciente por parte de los diseñadores para incorporar la verificación/reinicio, y a menos que las tareas requieran muchas horas de ejecución, su uso es poco común.

- *Algoritmo del avestruz*.

Otra forma de percibir el problema de los bloqueos mutuos, que es la más común empleada en todos los sistemas operativos de propósito general, es el llamado *algoritmo del avestruz*: ignorar las situaciones de bloqueo (escondiéndose de ellas como avestruz que mete la cabeza bajo la tierra), esperando que su ocurrencia sea poco frecuente, o si ocurre, que su efecto no repercuta en el sistema.

Hay que comprender que esto ocurre porque las condiciones impuestas por las demás estrategias resultan demasiado onerosas, el conocimiento previo resulta insuficiente, o los bloqueos simplemente pueden presentarse ante recursos externos y no controlados (o conocidos siquiera) por el sistema operativo.

La realidad del cómputo marca que es **el programador** de aplicaciones quien debe prever las situaciones de carrera, bloqueo e inanición en su código; el sistema operativo empleará ciertos mecanismos para asegurar la seguridad en general entre los componentes del sistema, pero el resto recae en las manos del programador.

- *Implementación de mecanismos de sincronización*.

Los mecanismos de sincronización permiten forzar a un proceso a detener su ejecución hasta que ocurra un evento en otro proceso.

La implementación de estos mecanismos de sincronización, por parte de los programadores de aplicaciones, se puede realizar usando servicios que ofrece el sistema operativo, los cuales trabajan, algunos con procesos (creados con fork()) y otros con hilos (por ejemplo, hilos POSIX). Estos servicios son:

- Señales (asincronismo)

- Tuberías (pipes, FIFOS)
- Semáforos
- Mutex y variables condicionales
- Paso de mensajes

Ya revisamos el uso de las tuberías bajo procesos creados con fork(). Revisar a detalle el resto de los servicios nos llevaría otro curso completo. Por lo que sólo veremos de manera general el concepto de *mutex* y su uso en la implementación de un tipo de semáforos.

Actividad. 2.5.16 Elabora un cuadro sinóptico de la clasificación de las áreas de estudio de los bloqueos mutuos y sus correspondientes algoritmos, ventajas y desventajas.

Sincronización de hilos en POSIX: Mutex

Mutex (mutual exclusion) es un mecanismo de sincronización de hilos POSIX que actúa como semáforos binarios optimizados para exclusión mutua. Cuenta con las siguientes características que se deben considerar al momento de su implementación en aplicaciones:

- Inicializados a 1 por defecto.
- Tiene un propietario, lo que significa que sólo el hilo que hace el lock puede hacer el unlock; es decir, cuando un hilo hace la operación P() y entra a la región crítica, sólo él puede hacer la operación V() para avisar que ya salió de la región crítica.

Por estas características, el mutex para hilos Posix, está limitado a usarse para la implementación de sólo ciertos algoritmos vistos en clase. ¿Puedes numerar y justificar en cuáles no se podría usar?

- Ejercicio de aplicación de mutex en hilos Posix

Descripción:

Es común que muchos sistemas manejen datos en vectores y realicen operaciones sobre ellos. Cuando resultan vectores muy grandes, es conveniente realizar las operaciones dividiendo el trabajo entre varios hilos.

Supongamos que se requiere obtener la suma de los N elementos de un vector usando N_H hilos. Cada hilo hará una suma parcial de N/N_H elementos.

Por ejemplo, si N = 1000 y N_H = 4, entonces cada hilo realizará la suma parcial de los siguientes elementos:

hilo[0] = del 0 al 249 (suma 250 elementos)
hilo[1] = del 250 al 499 (suma 250 elementos)
hilo[2] = del 500 al 749 (suma 250 elementos)
hilo[3] = del 750 al 999 (suma 250 elementos)

La suma se irá calculando en la variable global Gsuma.
Presento dos versiones de programas que “resuelven” el problema:

Versión 1. El cálculo de la suma de los elementos se realiza usando la variable global Gsuma compartida por todos los hilos (región crítica). No se realiza ningún control de acceso a dicha variable por lo que salen diferentes resultados al ejecutarlo varias veces.

Versión 2. Se utiliza mutex para controlar el acceso a la variable global Gsuma, por lo que el cálculo se realiza de forma adecuada.

Programa Versión 1 (sin uso de mutex).

```
/* Programa que suma los elementos de un vector de N números enteros. */
/* Se usan N_H hilos para hacer sumas parciales equitativas */
#include <stdio.h>
#include <pthread.h>
#define N_H 4 /* Numero de hilos en los que se repartirá el trabajo */
#define N 1000 /* Tamaño del vector a sumar */

int A[N], /* Vector al que se sumarán sus elementos */
Gsuma = 0; /* Variable global donde se tendrá la suma de los elementos del vector */

void *codigo_del_hilo(void *id)
{
        int i,j;
        j=*(int *)id;
        for (i=j;i<j+N/N_H;i++)
                Gsuma+=A[i]; /* El hilo va sumando los elementos depositándolo en Gsum*/
        pthread_exit(id);
}
main()
{
        int i, h, e[N_H];
        pthread_t hilo[N_H];
        int error;
        int *salida;
        for (i=0;i<N;i++) /* Inicialización del vector a sumar */
        {
                A[i]=1;
        }
        for (h=0; h<N_H; h++) /* Crea los hilos entre los que reparte la tarea */
        {
                e[h]=N/N_H*h; /* Cálculo del índice inicial a sumar */
                error = pthread_create(&hilo[h],NULL,codigo_del_hilo,&e[h]);
        }
        for (h=0; h<N_H; h++) /* Espera a que terminen los n hilos */
        {
                error = pthread_join(hilo[h],(void **)&salida);
                printf ("\nHilo terminado que suma de %d a %d",*salida,*salida+N/N_H-1);
        }
        printf("\nGsuma=%d\n",Gsuma);
}
```

Programa Versión 2 (con uso de mutex).

```
/* Programa para sumar los elementos de un vector de N números enteros. */
/* Se usan N_H para hacer sumas parciales equitativas */
#include <stdio.h>
#include <pthread.h>
#define N_H 4 /* Numero de hilos en los que se repartira el trabajo */
#define N 1000 /* Tamaño del vector a sumar */

pthread_mutex_t gMutex; /* Se declara una variable de tipo mutex */
int A[N], /* Vector al que se sumarán sus elementos */
Gsuma =0; /* Variable global donde se tendrá la suma de los elementos del vector */

void *codigo_del_hilo(void *id)
{
        int i,j;
        j=*(int *)id;
        for (i=j;i<j+N/N_H;i++){
                pthread_mutex_lock(&gMutex); /* equivale a la operación P(gMutex) */
                Gsuma+=A[i]; /* Región crítica */
                pthread_mutex_unlock(&gMutex); /* equivale a la operación V(gMutex) */
        }
        pthread_exit(id);
}
main()
{
        int i, h, e[N_H];
        pthread_t hilo[N_H];
        int error;
        int *salida;
        for (i=0;i<N;i++) /* Inicializacion del vector a sumar */
        {
                A[i]=1;
        }
        pthread_mutex_init(&gMutex, NULL); /* Se inicia la variable mutex */
        for (h=0;h<N_H;h++) /* Crea los hilos entre los que reparte la tarea */
        {
                e[h]=N/N_H*h; /* se calcula el índice inicial del vector en cada hilo para realizar la
                suma */
                error = pthread_create(&hilo[h],NULL,codigo_del_hilo,&e[h]);
        }
        for (h=0; h<N_H; h++) /* Espera a que terminen los n hilos */
        {
                error = pthread_join(hilo[h],(void **)&salida);
                printf ("\nHilo terminado que suma de %d a %d",*salida,*salida+N/N_H-1);
        }
        printf("\nGsum=%d\n",Gsum);
}
```