LABORATORIO SISTEMAS OPERATIVOS 2

CONSIDERACIONES DE DISEÑO DE HILOS

PROCESOS COOPERATIVOS

Un proceso cooperativo es aquel que puede afectar o ser afectado por otros procesos que se ejecutan en el sistema. Los procesos que cooperan pueden compartir directamente un espacio de direcciones lógicas (es decir, código y datos) o se les puede permitir compartir datos solo a través de archivos o mensajes.

El acceso concurrente a datos compartidos puede provocar inconsistencia en los datos. Por lo que se necesita de mecanismos para garantizar la ejecución ordenada de procesos cooperativos que comparten un espacio de direcciones lógicas, de modo que se mantenga la coherencia de los datos.

CONDICIÓN DE CARRERA

La condición de carrera ocurre cuando más de un proceso intenta acceder y modificar los mismos datos o recursos compartidos. Debido a que muchos procesos intentan modificar los datos o recursos compartidos, existen grandes posibilidades de que un proceso obtenga resultados o datos incorrectos.

Por lo tanto, cada proceso corre para decir que tiene datos o recursos correctos y esto se denomina condición de carrera.

SECCIÓN CRITICA

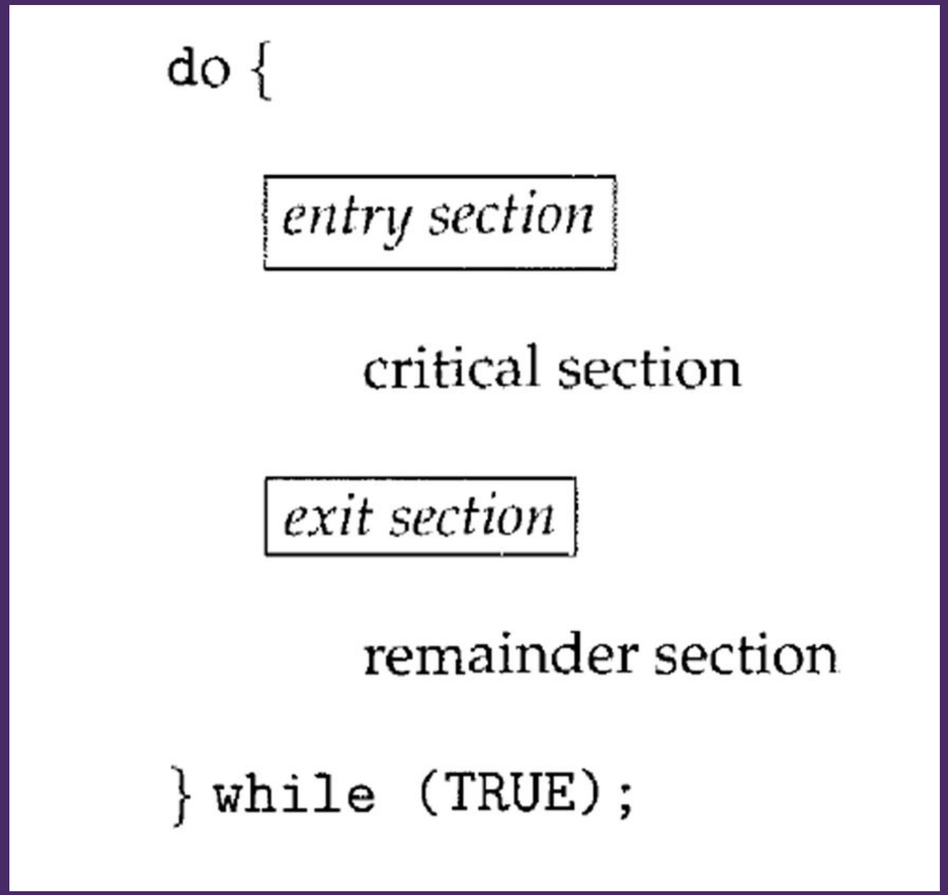
Considere un sistema que consta de n procesos {Po, P1, ..., Pn-1}. Cada proceso tiene un segmento de código, llamado sección crítica, en el que el proceso puede estar cambiando variables comunes, actualizando una tabla, escribiendo un archivo, etc.

La característica importante del sistema es que, cuando un proceso se ejecuta en su sección crítica, no se permite que ningún otro proceso se ejecute en su sección crítica.

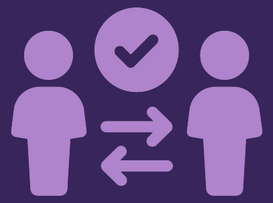
El uso de la sección crítica es un permite que los procesos puedan cooperar sin modificar los datos de otro proceso mientras este se esta ejecutando.

Cada proceso debe solicitar permiso para ingresar a su sección crítica.

La sección de código que implementa esta solicitud es la sección de entrada. La sección crítica puede ir seguida de una sección de salida. El código restante es la sección restante.



Implementaciones de sección crítica deben satisfacer los tres requisitos siguientes:



EXCLUSIÓN MUTUA

Si el proceso P se está ejecutando en su sección crítica, entonces ningún otro proceso puede ejecutarse en sus secciones críticas.

PROGRESO

Si ningún proceso se está ejecutando en su sección crítica y algunos procesos desean ingresar a sus secciones críticas, entonces solo aquellos procesos que no se están ejecutando en sus secciones restantes pueden participaren la decisión sobre cuál ingresará a su sección crítica.

ESPERA LIMITADA

Existe un límite en el número de veces que otros procesos pueden ingresar a sus secciones críticas después de que un proceso haya realizado una solicitud para ingresar a su sección crítica y antes de que se conceda esa solicitud.

LA SOLUCIÓN DE PETERSON

El informático llamado Gary L. Peterson dio un enfoque muy utilizado para resolver el problema de la sección crítica.

En esta solución, cuando un proceso se ejecuta en la sección crítica al mismo tiempo, otros procesos pueden acceder al resto del código y también es posible lo contrario. Lo importante es que esta solución garantiza que solo un proceso esté ejecutando una sección crítica al mismo tiempo. Entendamos esta solución con la ayuda de un ejemplo.

La solución de Peterson requiere que se compartan dos elementos de datos entre los procesos:

```
int turn;
boolean flag[n];
```

La variable turno indica a quién le toca entrar en su sección crítica. Eso es, si turn == i, entonces el proceso Pi puede ejecutar se en su sección crítica.

El arreglo flag se utiliza para indicar si un proceso está listo para ingresar a su sección crítica. Por ejemplo, si flag[i] es verdadero, este valor indica que Pi está listo para ingresar su sección crítica.

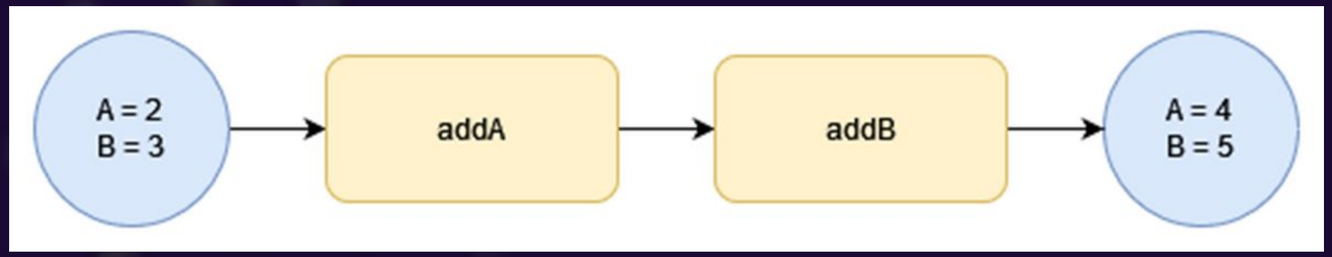
P1:

```
do{
    flag[0]=True;
    turn = 1;

    while( flag[1] == True && turn == 1){
        // Espera...
    }

    // Seccioncritica
      ...
    // Fin de seccioncritica

    flag[0]=False

} while(True)
```

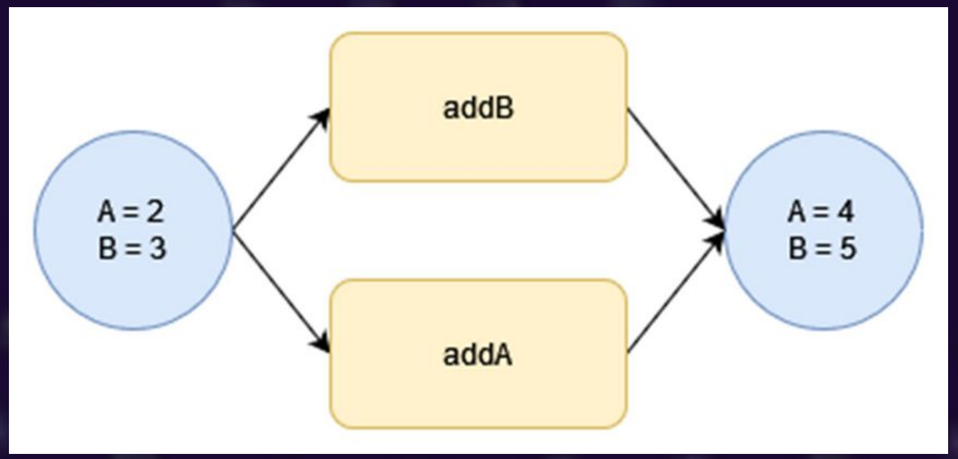
P2:

```
do{
    flag[1]=True;
    turn = 0;

    while( flag[0] == True && turn == 0){
        // Espera...
    }

    // Seccioncritica
      ...
    // Fin de seccioncritica

    flag[1]=False

} while(True)
```

Al comportamiento de addA es independiente del comportamiento de addB y, por lo tanto, no tiene problemas para ejecutarse de manera concurrente



Ni ocurren problemas al ejecutarse de manera paralela.



SEMAFOROS

El científico holandés E.W. Dijkstra mostró cómo resolver el problema de la sección crítica a mediados de los años 60 e introdujo el concepto de semáforo para controlar la sincronización.

Un semáforo es una variable entera a la que se accede mediante dos operaciones especiales, llamadas wait y signal.

Los semaforos son capaces de imponer la exclusión mutua, evitar condiciones de carrera e implementar la sincronización entre procesos.

1

Un semáforo S es una variable entera que, además de la inicialización, es accedida únicamente a través de dos operaciones atómicas: wait() y signal().

La operación de wait disminuye el valor del semáforo y la operación de signal incrementa el valor del semáforo.

Cuando el valor del semáforo es cero, cualquier proceso que realice una operación de espera será bloqueado hasta que otro proceso realice una operación de señal.

```
wait(S) {
    while S <= 0
        ; // no-op
    S--;
}
```

```
signal(S) {
    S++;
}
```

2

Cuando un proceso realiza una operación de wait en un semáforo, la operación verifica si el valor del semáforo es > 0. Si es así, disminuye el valor del semáforo y deja que el proceso continúe su ejecución; de lo contrario, bloquea el proceso en el semáforo.

Una operación de signal en un semáforo activa un proceso bloqueado en el semáforo, si lo hay, o incrementa el valor del semáforo en 1.

Debido a esta semántica, los semáforos también se denominan semáforos de conteo. El valor inicial de un semáforo determina cuántos procesos pueden pasar la operación de espera.

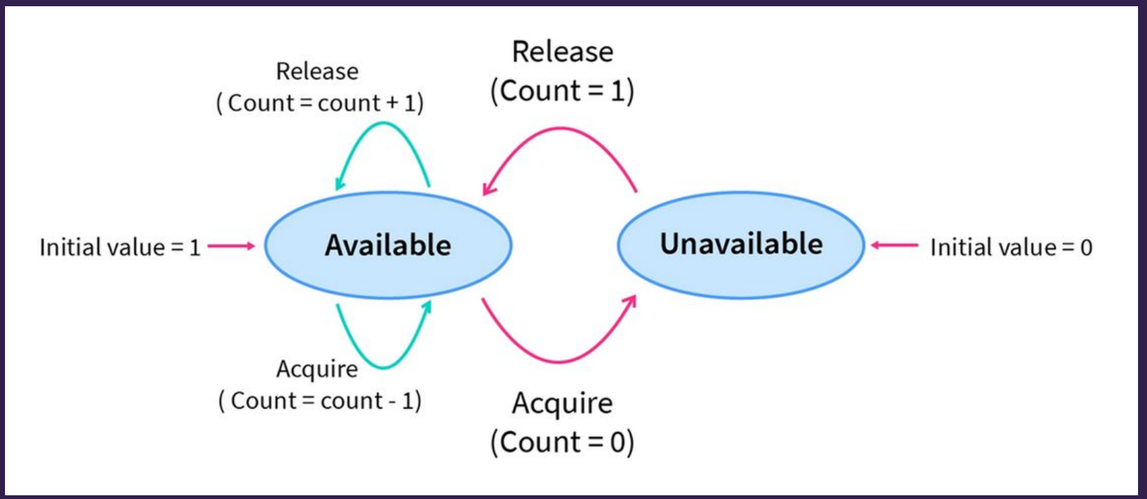
```
wait(S) {
    while S <= 0
        ; // no-op
    S--;
}
```

```
signal(S) {
    S++;
}
```

Un semáforo puede tomar valores mayores que uno y se pueden utilizar para controlar el acceso a un recurso determinado que consta de un número finito de instancias.

Se inicializa S con el numero de recursos disponibles, si el valor del semáforo está por encima de 0, los procesos pueden acceder a la sección crítica o a los recursos compartidos. La cantidad de procesos que pueden acceder a los recursos/código es el valor del semáforo.

Sin embargo, si el valor es 0, significa que no hay recursos disponibles o que varios procesos ya están accediendo a la sección crítica, por lo que no pueden acceder más procesos.



OPERACIONES ATÓMICAS

El problema que tenemos con las soluciones anteriores es que algunas líneas de código que deben ejecutarse sin interferencia de otros procesos; Esto no es un problema siempre que un proceso tenga el control de la CPU, pero si el sistema operativo le quita la CPU antes de que pueda terminar de ejecutar su sección de código, otro proceso puede tener la oportunidad de arruinar su trabajo.

El código debe ejecutarse de forma atómica: en una unidad ininterrumpible. Entonces, para que los semáforos resuelvan el problema, la implementación de las dos funciones de espera y señal debe ser atómica: no puede haber interrupciones mientras se implementan estas operaciones.

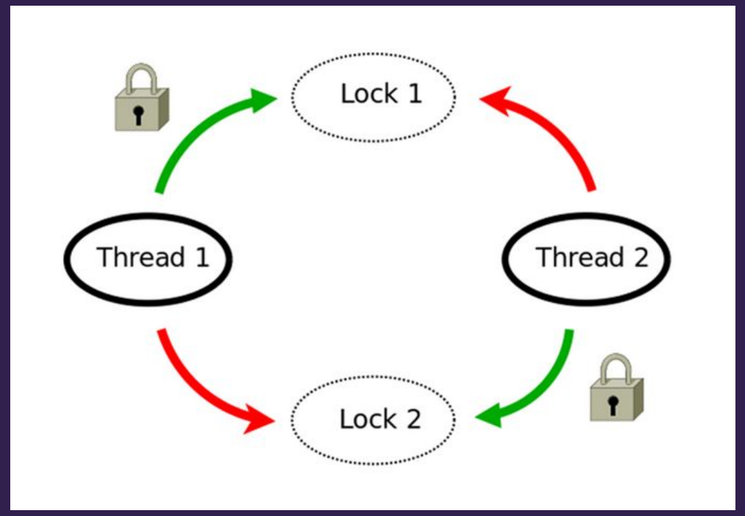
MUTEX

Cuando no se necesita la habilidad del semáforo de contar, algunas veces se utiliza una versión simplificada, llamada mutex. Los mutexes son buenos sólo para administrar la exclusión mutua para cierto recurso compartido o pieza de código.

Se implementan con facilidad y eficiencia, lo cual hace que sean especialmente útiles en paquetes de hilos que se implementan en su totalidad en espacio de usuario.

Un mutex es una variable que puede estar en uno de dos estados: abierto (desbloqueado) o cerrado (bloqueado).

Se utilizan dos procedimientos con los mutexes. Cuando un hilo (o proceso) necesita acceso a una región crítica, llama a mutex_lock. Si el mutex está actualmente abierto (lo que significa que la región crítica está disponible), la llamada tiene éxito y entonces el hilo llamador puede entrar a la región crítica.

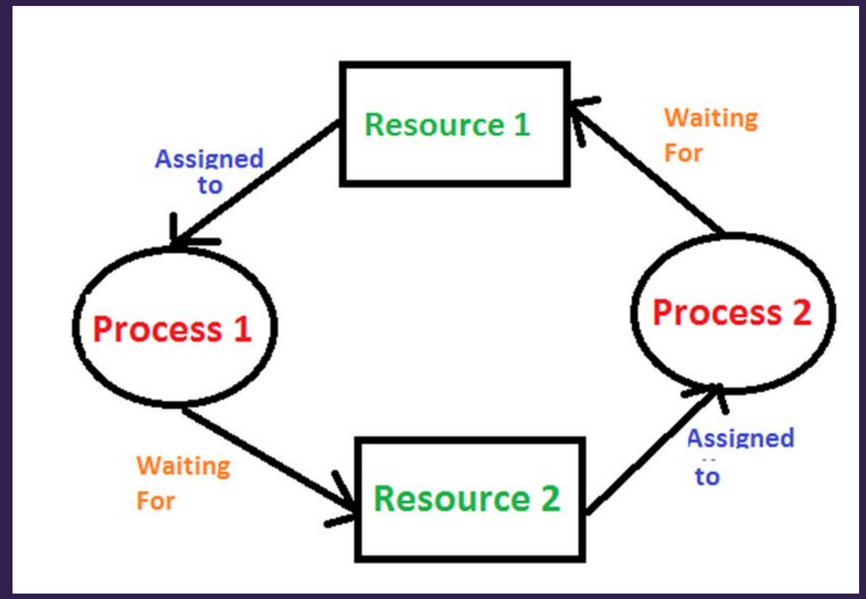


Por otro lado, si el mutex ya se encuentra cerrado, el hilo que hizo la llamada se bloquea hasta que el hilo que está en la región crítica termine y llame a mutex_unlock. Si se bloquean varios hilos por el mutex, se selecciona uno de ellos al azar y se permite que adquiera el mutex.

DEADLOCKS

Un deadlock o bloqueo mutuo es el bloqueo permanente de un conjunto de procesos o hilos de ejecución en un sistema concurrente que compiten por recursos del sistema.

Ocurre cuando un proceso entra en un estado de espera porque un recurso del sistema solicitado está retenido por otro proceso en espera, que a su vez está esperando otro recurso retenido por otro proceso en espera. Si un proceso permanece indefinidamente incapaz de cambiar su estado porque los recursos solicitados por él están siendo utilizados por otro proceso que está esperando, entonces se dice que el sistema está en un deadlock.



Puede surgir un deadlock si las siguientes cuatro condiciones se cumplen simultáneamente:

- Exclusión mutua: dos o más recursos no se pueden compartir (solo se puede usar en un proceso a la vez)
- Retener y esperar: un proceso retiene al menos un recurso y espera recursos.
- Sin preferencia: no se puede tomar un recurso de un proceso a menos que el proceso lo libere.
- Espera circular: conjunto de procesos que se esperan unos a otros en forma circular.

SOLUCIONAR DEADLOCKS

En términos generales, podemos resolver el problema del punto muerto en una de tres formas:

- Se puede utilizar un protocolo para prevenir o evitar bloqueos, asegurando que el sistema nunca entrará en un deadlock.
- Podemos permitir que el sistema entre en un estado de deadlock, detectarlo y recuperarse.
- Se puede ignorar el problema por completo y pretender que los deadlocks nunca ocurren en el sistema.

MENSAJES

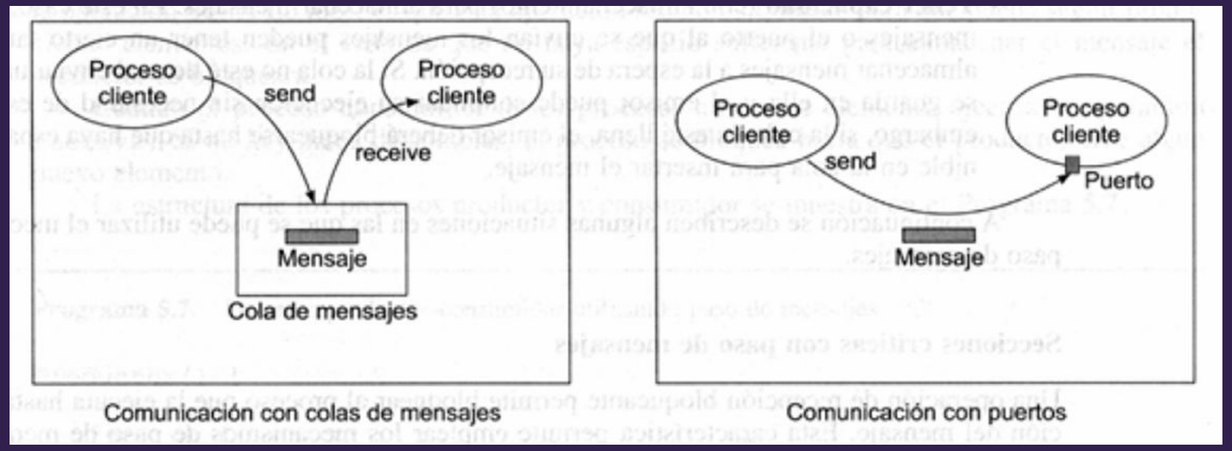
Todos los mecanismos vistos hasta ahora para la sincronización de procesos requieren que los procesos que se desean sincronizar se ejecutan desde el mismo dispoitivo.

Cuando se quiere comunicar y sincronizar procesos que ejecutan en máquinas distintas es necesario recurrir al paso mensajes. En este tipo de comunicación los procesos intercambian mensajes entre ellos.

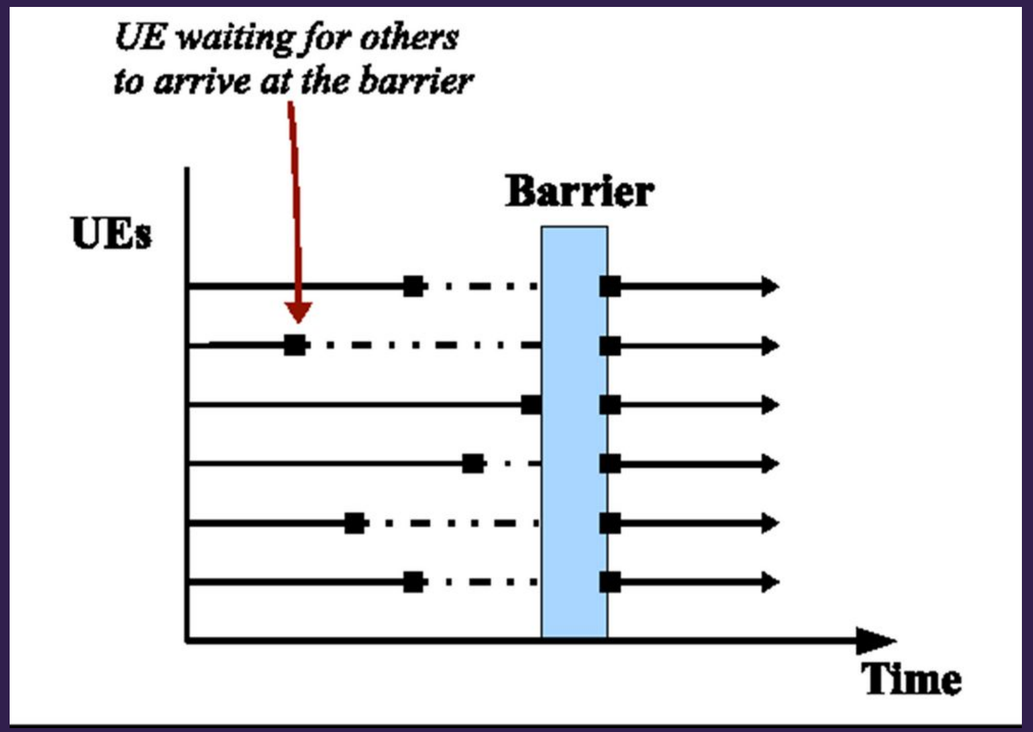
Este esquema también puede emplearse para comunicar y sincronizar procesos que ejecutan en la misma máquina, en este caso los mensajes son locales a la máquina donde ejecutan los procesos.

Utilizando paso de mensajes como mecanismo de comunicación entre procesos no es necesario recurrir a variables compartidas, únicamente debe existir un enlace de comunicación entre ellos. Los procesos se comunican mediante dos operaciones básicas:

- send(destino, mensaje): envía un mensaje al proceso destino.
- receive (origen, mensaje): recibe un mensaje del proceso origen.



BARRERAS



Otro mecanismo de sincronización es el uso de barreras.

Algunas aplicaciones se dividen en fases y tienen la regla de que ningún proceso puede continuar a la siguiente fase sino hasta que todos los procesos estén listos para hacerlo. Para lograr este comportamiento, se coloca una barrera al final de cada fase.

Cuando un proceso llega a la barrera, se bloquea hasta que todos los procesos han llegado a ella.

¡GRACIAS POR LA ATENCIÓN!

¿Dudas?