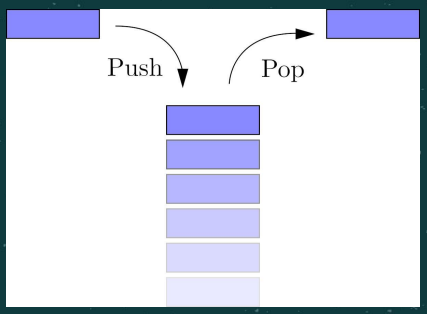
Sistemas Operativos I

Más mecanismos de Sincronización

Definimos rápidamente lo que es un tipo objeto como el conjunto de operaciones más una especificación que describe el correcto funcionamiento, esta puede ser secuencial o no. Un objeto es una entidad computacional que corresponde con esas operaciones y esa especificación. La especificación puede ser descrita de diferentes maneras, se llaman secuenciales a aquellas especificaciones que se definen en base a las secuencias (trazas) de la ejecución del objeto concurrente.
Y un objeto concurrente como algo de un tipo objeto que puede ser accedido (interactúa) con varios procesos de forma concurrente (thread-safe).

Objeto : Stack sin límite



- push
- pop

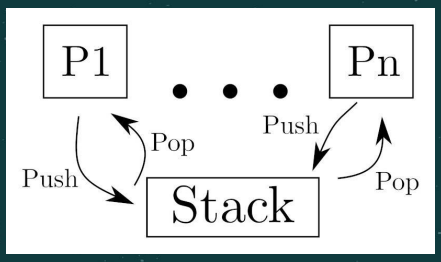
Como el stack no tiene límite siempre se puede invocar a la función `push`, y asumimos que hacer un `pop` de un stack vacío es `bottom`.
Especificación: todas las trazas que correspondan con una FILO (First-In-Last-Out).

Por ejemplo podemos definir rápidamente un objeto Stack sin límite de valores, como un objeto con dos operaciones push/pop, donde la función push agrega un elemento (y como el stack es sin límite) lo puede hacer siempre, y pop lo que hace es eliminarlo. En caso de que el stack esté vacío se devuelve un elemento distinguido llamado bottom.
La especificación está dada por todas las trazas (sequencia de pop/pushs) que se correspondan con un FILO


Veamos una implementación de
un Stack sin límite

Objeto : Stack sin límite Concurrente

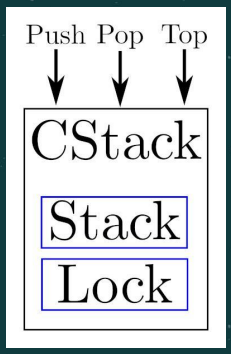
- conc_push
- conc_pop



Donde queremos que se comporte de la misma manera que el stack anterior pero será un stack compartido por varios procesos.

Dada una especificación secuencial de un objeto, es fácil implementar su versión concurrente simplemente utilizando locks.
La implementación la podemos hacer simplemente introduciendo un lock que proteja las regiones críticas (cada operación) del objeto.
Esto está muy bien, porque combina dos objetos en uno… Donde por una lado tenemos la idea de información o data bien concreta, y otro que es el lock que maneja el control del programa. (Operacional y data).
Mostrarlo en JAm!

Objeto : Stack sin límite Concurrente

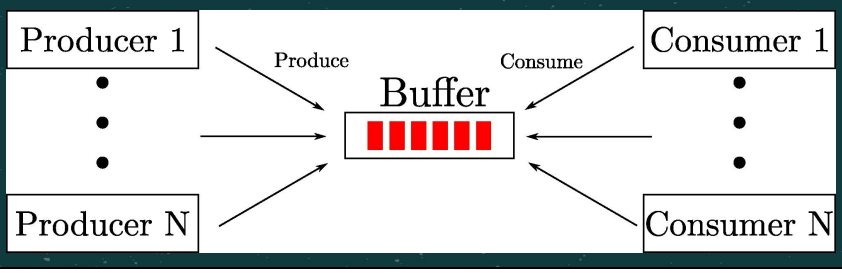


Dada una especificación secuencial de un objeto, es fácil implementar su versión concurrente simplemente utilizando locks.
La implementación la podemos hacer simplemente introduciendo un lock que proteja las regiones críticas (cada operación) del objeto.
Esto está muy bien, porque combina dos objetos en uno… Donde por una lado tenemos la idea de información o data bien concreta, y otro que es el lock que maneja el control del programa. (Operacional y data).

El ejercicio consiste en dada una ‘stack’ implementar un ‘stack concurrente’. Las operaciones del stack concurrente deben ser atómicas. Como veremos más adelante esto no soluciona los problemas.

Problema Productor(es) Consumidor(es)

Asumamos dos tipos de procesos, aquellos que producen un elemento y otros que lo consumen. Asumimos que la producción y el consumo se pueden dar en diferentes momentos, y por ende introducimos un **buffer** de tamaño **K**.



El problema de productor/consumidor ya lo mencioné, hay dos procesos uno que produce y otro que consume. La idea es que los productos que son producidos por el productor son consumidos por el consumidor, y no hay casos raros. Todo lo que produce se consume, y se consume solo lo que se produce.
Para hacerlo más divertido, asumamos que tenemos varios productores y varios consumidores, y como el consumo es diferido de la producción utilizamos un **buffer** donde los productores van poniendo los elementos producidos.

Procesos!

Productores

```
while(true):
   v <~ producir
   buff.depositar(v)
```

Consumidores

```
while(true):
   v <~ buff.tomar()
   consumir(v)
```

Los productores producirán siempre que puedan e invocaran al objeto concurrente **buff** para que deposite donde quiera el producto recién producido.

Objeto Concurrente Buffer(I)

Asumimos que el objeto concurrente Buffer tiene un stack interno llamado **istack** de tamaño **K**.

```
depositar(v)
si hay lugar en istack:
  stack_push(v)
```

```
tomar()
si no es vacio istack:
  v = stack_pop()
  retornar v
```

Comencemos a definir el objeto concurrente Buffer. Primero asumimos que tiene un stack interno (istack), de al menos tamaño **K**.
Definimos a depositar() simplemente como push() si hay lugar, y tomar() como pop() si hay al menos un elemento.
Problemas! Esto es básicamente un stack concurrente! Y podemos llegar a tener problemas entre las llamadas a pop/push.

Observar que al representar al buffer de productor-consumidor con un stack, puede hacer que algunos elementos queden mucho tiempo en el stack sin ser consumidos. Por esto el problema de productor-consumidor suele resolverse con una queue.

Esto igualmente tiene dos problemas que vamos a ignorar por ahora:

- Del lado del Productor, si no lo puede poner con push, porque está lleno, y tira el objeto, y vamos a tener objetos que nunca se van a consumir (PROBLEMA!)
- Del lado del Consumidor, si el buffer está vacío, qué retornamos??!!

Objeto Concurrente Buffer(II)

Asumimos que el objeto concurrente Buffer tiene un stack **concurrente** interno llamado **icstack** de tamaño **K**.

```
depositar(v)
si hay lugar en icstack:
  conc_stack_push(v)
```

```
tomar()
si no es vacio icstack:
  v = conc_stack_pop()
  retornar v
```

Podemos entonces aplicar lo que vimos antes, y encerrar las regiones críticas usando un lock.

Pero todavía tenemos un problema, para el caso de los productores: entre la lectura de que haya lugar en el stack y que realmente se introduzca podemos dejar que varios productores piensen que pueden hacer push sin tener lugar.

De manera análoga, para el caso de los consumidores: entre la verificación de que el buffer no está vacio y el pop() de un elemento puede hacer que varios consumidores piensen que pueden hacer pop cuando en realidad está vacío el buffer.

Recordar que solo queremos producir si hay lugar en el buffer y solo queremos consumir si el buffer no está vacío, estoy no está sucediendo.

Objeto Concurrente Buffer(II')

Asumimos que el objeto concurrente Buffer tiene un stack interno llamado **istack** de tamaño **K** y un lock llamado **lock**.

```
depositar(v)
si hay lugar en icstack:
   tomar(lock)
   stack_push(v)
   soltar(lock)
```

```
tomar()
si no es vacio icstack:
   tomar(lock)
   v = stack_pop()
      soltar(lock)
   retornar v
```

Si expandimos las defincciones de conc_stack_push() y conc_stack_pop().

De esta manera podemos pensar en tomar el lock antes de las condiciones, si el buffer está lleno o si el buffer está vacío.

Objeto Concurrente Buffer(III)

Asumimos que el objeto concurrente Buffer tiene un stack interno llamado **icstack** de tamaño **K** y un lock llamado **lock**.

`depositar(v)`

```
tomar(lock)
si hay lugar en icstack:
   stack_push(v)
soltar(lock)
```

`tomar()`

```
tomar(lock)
si no es vacio icstack:
   v = stack_pop()
soltar(lock)
retornar v
```

Bueno, tomamos el lock y luego hacemos los chequeos necesarios…

Esto igualmente tiene dos problemas

- Del lado del Productor, si no lo puede poner con push, porque está lleno, y tira el objeto, y vamos a tener objetos que nunca se van a consumir (PROBLEMA!)
- Del lado del Consumidor, si el buffer está vacío, qué retornamos??!!

De alguna manera tenemos que verificar en el caso del productor que el elemento fue depositado en el buffer, y desde el lado del consumidor que el elemento fue tomado del buffer.

Objeto Concurrente Buffer(IV)

Asumimos que el objeto concurrente Buffer tiene un stack interno llamado **istack** de tamaño **K** y un lock llamado **lock**

```
depositar(v)
agregado = false
while(!agregado)
   tomar(lock)
   si hay lugar en icstack:
      stack_push(v)
      agregado = true
   soltar(lock)
```

```
tomar()
sacado = false
while (!sacado)
   tomar(lock)
   si no es vacio icstack:
      v = stack_pop()
      sacado = true
   soltar(lock)
retornar v
```

Bien, ya tenemos una solución que parece andar. Pero es super ineficiente, los hilos van a estar compitiendo todo el tiempo por adquirir el acceso a la región crítica. Para esto introduciremos un nuevo mecanismo que permite resolver el problema mucho más elegantemente.

Si analizamos lo que está pasando: los procesos dependen de ciertas condiciones para garantizar su correcta ejecución. Que el stack no esté lleno, o vacío, y para esto podríamos diseñar un mecanismo que los haga `dormir` hasta que estas condiciones se den.


Variables de Condición

Variables de Condición!

Las variables de condición representan la idea que hay un conjunto de procesos que esperan cierta condición, y una vez que esta se cumple, se despiertan para que puedan volver a competir por los recursos.

Y se utilizan en conjunto con mutex lock.

+ wait : pone al hilo que la invoca a la espera de una condición liberando el lock que tiene.
+ signal : despierta a un hilo que esté a la espera
+ broadcast : despierta a todos los hilos esperando.

Variables de Condición!

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t
*restrict mutex, const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict
mutex);
```

La función pthread_cond_timedwait() será equivalente a pthread_cond_wait(), excepto que se devuelve un error si pasa el tiempo absoluto especificado por abstime.

Si se bloquea más de un thread en una variable de condición, la política de programación determinará el orden en que se desbloquean los threads. Cuando cada thread se desbloquea como resultado de un pthread_cond_broadcast() o pthread_cond_signal() retorna de su llamada a pthread_cond_wait() o pthread_cond_timedwait(), el thread poseerá el mutex con el que llamó a pthread_cond_wait() o pthread_cond_timedwait(). Los threads que se desbloquean competirán por el mutex de acuerdo con la política de programación (si corresponde) y como si cada uno hubiera llamado a pthread_mutex_lock().

Observar que el while() es necesario porque podría suceder que haya threads productores esperando en la variable de condición (pthread_cond_wait()), luego se haga un signal por un consumidor (indicando que hay lugar en el buffer) pero justo venga otro productor (que no estaba esperando en la variable de condición) y llame a pthread_mutex_lock() y compita por el lock y se lo gane a los que estaban bloqueados en pthread_cond_wait(). Al ganar este otro thread, chequea que hay lugar en el buffer y lo llena (haciendo que no se cumpla más la condición). Entonces, el thread que se había despertado por el signal, ahora gana el lock y debe rechequear la condición ya que no se cumple más, y deberá bloquearse nuevamente en la variable de condición.

Un thread que está esperando en el pthread_cond_signal() está esperando dos cosas (en orden): primero un signal y luego intentará tomar el mutex. Una vez que reciba el signal pasará a intentar tomar el mutex. Si el mutex no está disponible se queda bloqueado en el mutex, no hace falta que sea signaleado nuevamente. Por este comportamiento, es necesario que se vuelva a rechequear la condición en el while, ya que podría ser que luego de un rato logre tener el mutex pero la condición haya cambiado.

Un thread puede llamar a las funciones pthread_cond_broadcast() o pthread_cond_signal() independientemente de que posea o no el mutex que llama a los subprocesos pthread_cond_wait() o pthread_cond_timed_wait() se han asociado con la variable de condición durante sus esperas; sin embargo, si se requiere un comportamiento de programación predecible, entonces ese mutex debe ser bloqueado por el subproceso que llama pthread_cond_broadcast() o pthread_cond_signal().

Las funciones pthread_cond_broadcast() y pthread_cond_signal() no tendrán efecto si no hay threads actualmente bloqueados en cond.

La subrutina pthread_cond_signal desbloquea al menos un hilo bloqueado, mientras que la subrutina pthread_cond_broadcast desbloquea todos los hilos bloqueados. Si se bloquea más de un subproceso en una variable de condición, la política de programación determina el orden en que se desbloquean los subprocesos.

Objeto Concurrente Buffer Variables de Condición

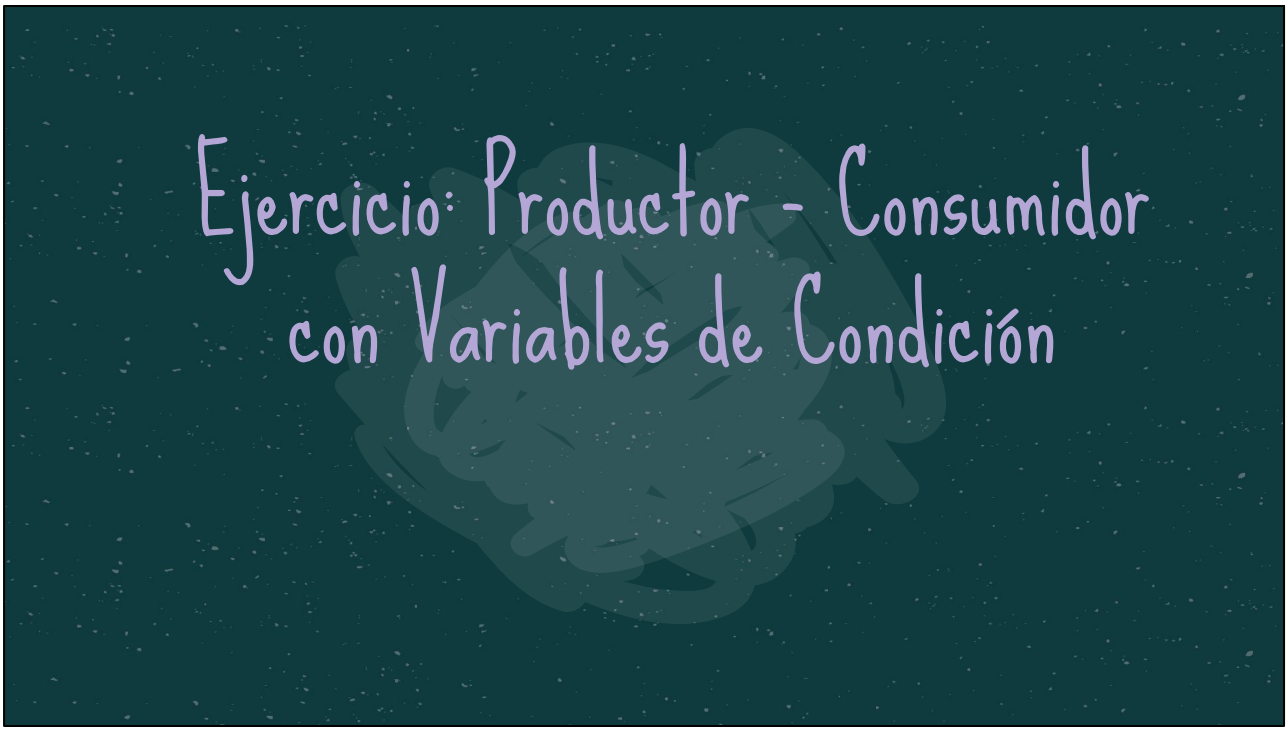
Asumimos que el objeto concurrente Buffer tiene un stack **concurrente** interno llamado **istack,** un semáforo **Lugares(K)** y **Ocupados(0)**.

```
depositar(v)
tomar(lock)
while (isFull())
  cond_wait(non_full_cond, lock)
stack_push(v)
cond_signal(non_empty_cond)
soltar(lock)
```

```
tomar()
tomar(lock)
while (isEmpty())
  cond_wait(non_empty_cond, lock)
v = stack_pop()
cond_signal(non_full_cond)
soltar(lock)
return v
```

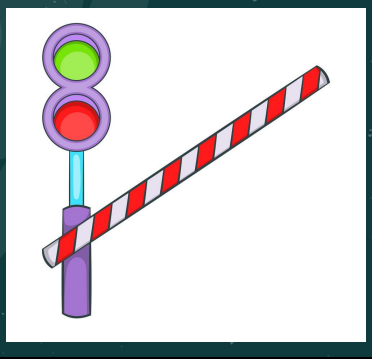
Poniendo todo junto, podemos resolver el objeto concurrente buffer de tamaño K utilizando, un stack concurrente, y dos variables de condición:

+ que el buffer no esté lleno
+ que el buffer no esté vacío.



Mostrar ejemplo minimal de semáforo.

Semáforos



Semáforos!

Los semáforos son un objeto concurrente que representa la idea de un **contador concurrente**. Es decir, es un contador con dos operaciones **atómicas** *down* y *up* .

- Un semáforo S se inicia con un valor s0 no negativo.
- El contador s0 nunca se vuelve negativo
- *down* decrementa atómicamente en uno el contador
- *up* incrementa atómicamente en uno el contador

Invariante: S = s0 + #(up) - #(down)

Es decir, podemos pensar a los semáforos como un contador concurrente, donde se puede incrementar o decrementar su valor en uno.
Como su valor nunca puede ser negativo, la operación down es **bloqueante** en el caso que quiera decrementar un contador en 0.
La operación up siempre se puede realizar sin violar ningún invariante por lo que es **no bloqueante**.
Si hay procesos bloqueados por querer decrementar el valor del contador, y se produce un up, todos compiten normalmente.
Tenemos un variante.

Los semáforos fueron inventados para brindar una solución elegante a un problema, pero hacen mucho y pueden ser complicados de usar.
Se usan en general para dos cosas

+ Proveen un lock
+ Se utilizan para preservar una condición.

Para la primera podemos usar un lock directamente, y para la segunda es más fácil utilizar lo que se conocen como **Variables de Condición!**

Remarcar que signal puede perderse! Si no hay nadie esperando no pasa nada. El hilo que se despierta vuelve a competir por el lock.

Objeto Concurrente Buffer Semáforos

Asumimos que el objeto concurrente Buffer tiene un stack **concurrente** interno llamado **istack,** un semáforo **Lugares(K)** y **Ocupados(0)**.

```
depositar(v)
Lugares.down()
conc_stack_push(v)
Ocupados.up()
```

```
tomar()
Ocupados.down()
t = conc_stack_pop()
Lugares.up()
retornar t
```

Poniendo todo junto, podemos resolver el objeto concurrente buffer de tamaño K utilizando, un stack concurrente, y dos semáforos, uno para cada condición:

+ que el buffer no esté lleno
+ que el buffer no esté vacío.

Semáforos!

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec
*abs_timeout);
int sem_destroy(sem_t *sem);

Link with -pthread.
```

sem_wait() disminuye (bloquea) el semáforo al que apunta sem. Si el valor del semáforo es mayor que cero, entonces el decremento continúa y la función regresa inmediatamente. Si el semáforo actualmente tiene el valor cero, luego la llamada se bloquea hasta que sea posible realizar la disminución (es decir, el valor del semáforo aumenta por encima de cero) o un controlador de señal interrumpe la llamada.

sem_trywait() es lo mismo que sem_wait(), excepto que si el decremento no se puede realizar de inmediato, entonces la llamada devuelve un error (errno establecido en EAGAIN) en lugar de bloquear.

sem_timedwait() es lo mismo que sem_wait(), excepto que abs_timeout especifica un límite en la cantidad de tiempo que la llamada debe bloquearse si la disminución no se puede realizar de inmediato. El argumento abs_time-out apunta a una estructura que especifica un tiempo de espera absoluto en segundos y nanosegundos desde Epoch, 1970-01-01 00:00:00 +0000 (UTC).

Mediante variables de condición se puede también emplear una semántica parecida (aunque no igual) a la de los semáforos, con los métodos var.wait() y var.signal(). En el caso de los monitores, var.wait() suspende al hilo hasta que otro hilo ejecute var.signal(); en caso de no haber ningún proceso esperando, var.signal() no tiene ningún efecto (no cambia el estado de var, a diferencia de lo que ocurre con los semáforos).

Mostrar ejemplo minimal de semáforo.


Monitores

Monitores

```
typedef struct monitor {

  shared_data;    // Shared data to be protected
  sync_mechanism; // mutex, var. cond., semaphores
} monitor_t;

operation1(monitor_t *monitor){...}
operation2(monitor_t *monitor){...}
…
```

- En programación concurrente, un monitor es una estructura de sincronización que impide que los hilos accedan simultáneamente al estado de un objeto compartido y les permite esperar a que este cambie. Proporcionan un mecanismo para que los hilos renuncien temporalmente a un mutex, esperando a que se cumpla una condición, para luego recuperarlo y reanudar su tarea. Un monitor consta de un mutex (bloqueo) y al menos una variable de condición. Una variable de condición se "signalea" explícitamente cuando se modifica el estado del objeto, transfiriendo temporalmente el mutex a otro hilo que la espera.
- La característica que define a un monitor es que sus métodos se ejecutan con exclusión mutua: en cada momento, como máximo un hilo puede estar ejecutando cualquiera de sus métodos. Al usar una o más variables de condición, también se puede permitir que los threads esperen una condición determinada (de ahí el nombre "monitor"). Un Monitor se dice que es un objeto concurrente thread-safe (seguro para múltiples threads).

Los monitores proporcionan una abstracción simple y de alto nivel que puede usarse para implementar sistemas concurrentes complejos. Los monitores también garantizan que la sincronización esté encapsulada dentro del módulo, lo que facilita el razonamiento sobre la corrección del sistema.

```
typedef struct monitor {
  // Shared data
  int value;
  // Mutex for mutual exclusion
  pthread_mutex_t mutex;
  // Condition variable for signaling
  pthread_cond_t cond;
} monitor_t;

void monitor_init(monitor_t *monitor) {
  monitor->value = 0;
  pthread_mutex_init(&monitor->mutex, NULL);
  pthread_cond_init(&monitor->cond, NULL);
}

void monitor_increment(monitor_t *monitor) {
  pthread_mutex_lock(&monitor->mutex);
  monitor->value++;
  pthread_cond_signal(&monitor->cond); // Signal waiting threads
```

```
  pthread_mutex_unlock(&monitor->mutex);
}

int monitor_get_value(monitor_t *monitor) {
  pthread_mutex_lock(&monitor->mutex);
  while (monitor->value == 0) {
    pthread_cond_wait(&monitor->cond, &monitor->mutex);
  }
  int val = monitor->value;
  pthread_mutex_unlock(&monitor->mutex);
  return val;
}
```

El la Literatura existen dos tipos de Monitores: Hoare y Mesa.

La diferencia entre monitores de Hoare y monitores de Mesa tiene que ver con cómo se manejan las variables de condición y qué hilo tiene el control tras una señalización (signal). Ambos modelos son formas de implementar monitores, pero difieren en la semántica de la señalización.

Monitor de Hoare:

- Transferencia inmediata del control al hilo que estaba esperando una condición.
- El hilo que hace signal() cede el control al hilo que estaba esperando en esa condición.
- Se asegura que la condición sigue siendo verdadera justo después de la señalización.

Características de Hoare:

- Precisión en la sincronización: el hilo despertado actúa inmediatamente.
- Más complejo de implementar.
- Útil cuando es muy importante garantizar el estado del monitor al despertar.

Ejemplo de comportamiento con Monitor de Hoare:

1) Hilo A espera en una condición.
2) Hilo B hace signal().
3) B se bloquea, y A toma el control inmediatamente.
4) Cuando A termina, B continúa.

Monitor de Mesa:

- El hilo que hace signal() continúa ejecutándose.
- El hilo que fue despertado por signal() espera su turno para entrar al monitor (va a la cola de entrada).

Características de Mesa:

- Más fácil de implementar con sistemas de planificación modernos.
- El hilo despertado debe volver a verificar la condición (por eso se usan bucles while en lugar de if).
- Puede haber un retraso entre signal() y la ejecución real del hilo despertado.

Ejemplo de comportamiento con Monitor de Mesa:

1) Hilo A espera en una condición.
2) Hilo B hace signal().
3) B sigue ejecutándose.
4) A se despierta, pero espera hasta que B salga del monitor para entrar.

En la materia siempre utilizaremos Monitores Mesa debido a que las variables de condición en pthread son de tipo Mesa.