Integrantes

Estefanía Goez Moreno

Jaime Andrés Muñoz

Hilos

- son componentes básicos de los procesos que permiten dividir su ejecución en partes más pequeñas y ejecutar tareas de manera concurrente. Son especialmente útiles cuando se busca mejorar el rendimiento de aplicaciones, aprovechar al máximo los recursos del hardware (como procesadores multinúcleo) o mantener la responsividad de una interfaz mientras se realizan tareas intensivas. **Usos: concurrencia, paralelismo, tareas asíncronas.**

Proceso: Es una instancia de un programa en ejecución. Tiene su propio espacio de memoria y recursos asignados por el sistema operativo. Es independiente de otros procesos.

Hilo: Es una unidad de ejecución dentro de un proceso. Los hilos de un mismo proceso comparten el espacio de memoria y los recursos del proceso.

Sistema multiprocesador

Retos:

- Concurrencia: Varias tareas se ejecutan simultáneamente. Es esencial evitar errores como condiciones de carrera, donde múltiples hilos acceden o modifican datos compartidos de manera no controlada.
- Sincronización: Coordinar la ejecución de hilos o procesos para garantizar el acceso ordenado a recursos compartidos.
- Escalabilidad: Asegurar que el programa aproveche eficientemente un mayor número de procesadores sin disminuir el rendimiento debido a cuellos de botella.
- Balanceo de carga: Distribuir equitativamente las tareas entre los procesadores para evitar sobrecarga en algunos núcleos mientras otros están infrautilizados.

Paralelismo vs concurrencia

Paralelismo: Físicamente se tiene la capacidad de ejecutar varias actividades simultáneas en el tiempo. (varias CPUS).

Concurrencia: En la misma CPU, varias actividades se están ejecutando a la vez.

Tipos de paralelismo

Paralelismo a nivel de instrucción (ILP): Ejecuta múltiples instrucciones de un programa en paralelo utilizando técnicas como pipelining y ejecución superscalar en procesadores.

Paralelismo a nivel de datos: Aplica la misma operación a diferentes datos de manera simultánea. Es típico en aplicaciones de procesamiento masivo, como gráficos o inteligencia artificial.

Paralelismo a nivel de tareas: Divide un problema en subtareas independientes, que pueden ejecutarse en paralelo en diferentes procesadores o núcleos.

Paralelismo a nivel de memoria o procesos: Cada proceso tiene su propia memoria y ejecuta su tarea de forma independiente. Común en sistemas distribuidos.

En clase se discutió acerca del navegador Google Chrome y su técnica de abrir cada nuevo sitio web usando un proceso diferente. ¿Se habrían logrado los mismos beneficios si Chrome hubiera sido diseñado para abrir cada nuevo sitio web en un hilo separado?

No, ya que los hilos comparten el mismo espacio de memoria. Entonces, Si un hilo tiene un fallo crítico, podría corromper datos compartidos y comprometer todo el navegador. Además, los hilos no pueden ejecutarse de forma aislada debido a que comparten recursos, lo que abre brechas de seguridad.

Condiciones de carrera (RACE CONDITION)

Se produce cuando 2 o más hilos intentan acceder a posiciones de memoria que son compartidas y si acceden de manera no sincronizada, los resultados son inconsistentes (debido al orden en que se pueden dar las interrupciones de SO). Lo que ocasiona que los resultados no sean deterministas.

Ejemplo: Dos hilos (hilo1 y hilo2) intentan incrementar la variable contador simultáneamente.

Sección crítica: Sección del código en dónde puede presentarse una condición de carrera; por lo tanto, debe ejecutarse como si fuera una instrucción atómica.

Criterios de evaluación

- Exclusión mutua: Garantiza que solo un hilo o proceso puede estar dentro de la sección crítica (accediendo a los recursos compartidos) a la vez.
- Equidad: Asegura que cada hilo tiene un límite en la cantidad de tiempo que puede esperar para entrar en la sección crítica, evitando injusticias (starvation).
- Desempeño: Si ningún hilo está en la sección crítica, los hilos que deseen entrar deben poder decidir cuál lo hará en un tiempo finito, evitando bloqueos innecesarios.

La solución de sincronización por hardware consiste en utilizar instrucciones especiales del procesador para garantizar que las operaciones críticas se ejecuten de manera atómica (indivisible). Estas soluciones son implementadas directamente por el hardware y permiten resolver problemas de exclusión mutua y sincronización en sistemas concurrentes.

Mutex

Nombre que la librería pthreads ha dado al lock

Funciones.

- Lock: Solicita el acceso exclusivo al recurso compartido.
- Unlock: Libera el Mutex, permitiendo que otros hilos accedan al recurso compartido.
- Park: Pone el hilo rn modo sleep e una cola mientras no puede tener acceso a la región crítica.
- Unpark: despierta el hilo correspondiente para que acceda a la región crítica.

El uso de park y unpark facilita gestionar el manejo de una cola que permite establecer un orden de los hilos que necesitan acceder a la región crítica. De esta manera, se evita que los hilos esperen infinitamente su turno (spin waiting).

Estas funciones deben ser atómicas, seguras (evitar el acceso simultáneo) y equitativas (garantizar que los hilos obtengan el Mutex en el orden en que lo solicitaron).

Spin Waiting: Espera activa. Los demás hilos se quedan esperando activamente a que se libere el lock para acceder a la región crítica.

Variables de condición vs locks

Las variables de condición permiten establecer un orden de ejecución; por ejemplo, en el caso de que un hilo padre cree un hilo hijo en su ejecución, entonces, es necesario que el hilo hijo finalice completamente

para continuar con el padre. Las variables de condición, además de un Lock también requieren un flag (bandera) y una señal.

Los Locks garantizan exclusión mutua en una sección crítica.

Similitudes

- Ambas primitivas se utilizan para coordinar la ejecución de múltiples hilos o procesos que comparten recursos, asegurando un acceso ordenado y seguro.
- Las variables de condición suelen usarse junto con locks para gestionar acceso controlado a recursos compartidos en escenarios más complejos.

Semáforo

Es una estructura de sincronización que tiene un contador en ella. Permite representar tanto el comportamiento de un Lock como el de una condición de variable.

Mutex	Semáforo
Exclusión mutua: permite que un único hilo acceda a un recurso.	Sincronización y control del acceso a recursos múltiples.
Binario (bloqueado/desbloqueado).	Contador entero (puede ser mayor a 1).
Tiene un dueño: el hilo que lo bloquea debe liberarlo.	No tiene dueño: cualquier hilo puede realizar Signal.

Wait: Decrementa el valor del contador. Si el número resultante es negativo, el hilo se pone a dormir.

Post: Incrementa el valor del contador. Si hay hilos a la espera del semáforo, despierta alguno de ellos.

Deadlock (interbloqueo)

Se produce cuando 2 o más hilos se bloquean a la espera de un evento que va a suceder.

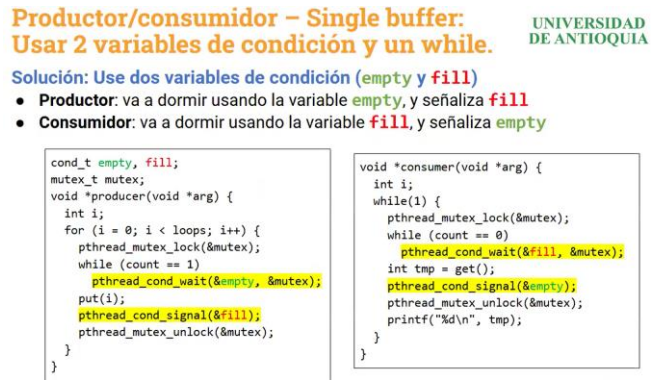
Starvation (inanición)

Se ocasiona cuando uno o más hilos esperan indefinidamente el acceso a un recurso compartido, debido a que este aún no ha sido liberado.

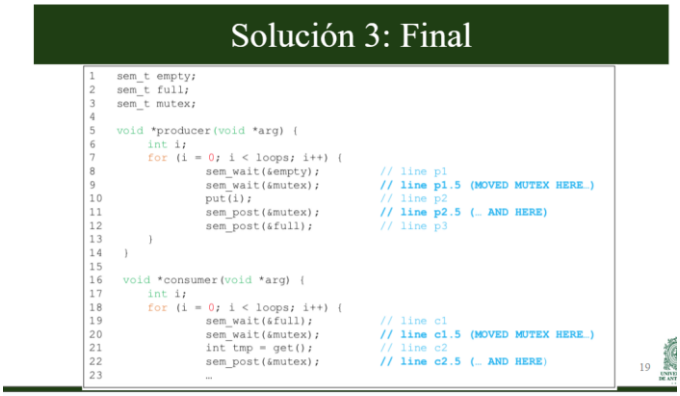
Problemas clásicos de sincronización

1. **El problema del productor-consumidor (o del buffer limitado)**
   Un productor genera datos y los coloca en un buffer compartido, mientras que un consumidor extrae datos del mismo buffer. El desafío está en garantizar que:
   - El productor no intente agregar datos cuando el buffer esté lleno.
   - El consumidor no intente retirar datos cuando el buffer esté vacío.

   Un productor despierta consumidores y un consumidor despierta productores para evitar **DEADLOCKS.**



Solución con semáforos



Solución típica:

- Utilizar semaforos o monitores para sincronizar el acceso al buffer.
- Dos contadores: uno para rastrear elementos llenos y otro para rastrear espacios vacíos.
- Exclusión mutua para proteger las operaciones sobre el buffer.

2. **El problema de los lectores-escritores**
   Un recurso compartido (como una base de datos) puede ser leído por múltiples procesos simultáneamente (lectores), pero solo uno puede escribir (escritor), y los escritores no pueden acceder al recurso mientras haya lectores activos.

   Requisitos:
   - Los lectores no deben interferir entre sí.
   - Un escritor debe tener acceso exclusivo.
   - Puede priorizarse a los lectores o a los escritores dependiendo del diseño.

   Solución típica:

   - Usar **lectura-escritura locks** (rw-locks) que permitan acceso compartido para lectores y acceso exclusivo para escritores.

   - Manejo cuidadoso de prioridades para evitar **inversión de prioridades** o **inanición**.

Solución con semáforos

Reader-Writer Lock

```
15      rw->readers++;
16      if (rw->readers == 1)
17              sem_wait(&rw->writelock); // first reader acquires writelock
18      sem_post(&rw->lock);
19  }
20
21  void rwlock_release_readlock(rwlock_t *rw) {
22      sem_wait(&rw->lock);
23      rw->readers--;
24      if (rw->readers == 0)
25              sem_post(&rw->writelock); // last reader releases writelock
26      sem_post(&rw->lock);
27  }
28
29  void rwlock_acquire_writelock(rwlock_t *rw) {
30      sem_wait(&rw->writelock);
31  }
32
33  void rwlock_release_writelock(rwlock_t *rw) {
34      sem_post(&rw->writelock);
35  }
```

23

3. El problema de los filósofos comensales

Cinco filósofos están sentados alrededor de una mesa con cinco tenedores, uno entre cada par de filósofos. Para comer, un filósofo necesita ambos tenedores adyacentes, pero solo puede tomar un tenedor a la vez. Esto puede causar interbloqueo (deadlock) si todos los filósofos toman un tenedor simultáneamente y esperan por el otro.

Solución típica:

- Evitar el interbloqueo introduciendo reglas como:
- Limitar el número de filósofos que pueden intentar comer al mismo tiempo.
- Hacer que cada filósofo tome ambos tenedores o ninguno.
- Numerar los tenedores y hacer que los filósofos los tomen en un orden específico (por ejemplo, primero el tenedor con el número más bajo).

Solución con semáforos

- Solución:
  - Cambiar el orden en el que los semáforos son adquiridos

```
1   void getforks() {
2       if (p == 4) {
3               sem_wait(forks[right(p)]);
4               sem_wait(forks[left(p)]);
5       } else {
6               sem_wait(forks[left(p)]);
7               sem_wait(forks[right(p)]);
8       }
9   }
```

29

```
6   void putforks() {
7       sem_post(forks[left(p)]);
8       sem_post(forks[right(p)]);
9   }
```

```
void* productor(void* arg) {
   for (i = 0; i < num; i++) {
      sem_wait(&empty);
      sem_wait(&mutex);
      put(i);
      sem_post(&mutex); // LÍNEA-A
      sem_post(&full);  // LÍNEA-B
   }
}
```

```
void* consumidor(void* arg) {
   while (!done) {
      sem_wait(&full);  // LÍNEA-C
      sem_wait(&mutex); // LÍNEA-D
      int tmp = get(i);
      sem_post(&mutex);
      sem_post(&empty);
      // hacer algo con tmp ...
   }
}
```

El valor inicial de empty debería ser del tamaño del buffer

El valor inicial de full debería ser 0, ya que indica la cantidad de elementos disponibles.

El valor inicial de mutex debería ser 1, ya que asegura que el primer hilo en llegar pueda adquirir el lonk

Se intercambia el orden de la LÍNEA-C con la LÍNEA-D

Resultado: La aplicación dejaría de ser funcional. Si el consumidor adquiere el mutex después de verificar full, varios consumidores podrían entrar a la sección crítica simultáneamente, lo que rompería la exclusión mutua. Esto podría llevar a condiciones de carrera al acceder al búfer compartido.

Se elimina la LÍNEA-A

Resultado: La aplicación dejaría de ser funcional. Sin liberar el mutex después de que el productor realiza su tarea, otros hilos (productores o consumidores) no podrán acceder al búfer, lo que resultará en un deadlock.

Se elimina la LÍNEA-B

Resultado: La aplicación dejaría de ser funcional. Al no incrementar el semáforo full, el consumidor nunca será notificado de que hay elementos disponibles en el búfer, lo que provocará que quede bloqueado indefinidamente en sem_wait(&full).

Se intercambia el orden de la LÍNEA-A con la LÍNEA-B

Resultado: La aplicación dejaría de ser funcional. Incrementar primero el semáforo full podría permitir que el consumidor acceda al búfer antes de que el productor haya terminado su operación (no ha liberado el mutex). Esto podría provocar una condición de carrera o el consumo de datos incorrectos.

Spinlocks

Los spinlocks son mecanismos de sincronización que funcionan haciendo que un hilo "gire" en un bucle mientras espera a que un recurso compartido se libere. Esto implica que el hilo permanece activo, realizando constantemente comprobaciones, en lugar de ceder el control de la CPU.

En un sistema monoprocesador, solo un hilo puede ejecutarse a la vez. Si un hilo adquiere el spinlock y otro hilo intenta adquirirlo, este último entrará en un bucle infinito (spinning), pero nunca tendrá la oportunidad de ejecutarse porque el primer hilo no cede el procesador hasta que termine. Esto resulta en una pérdida total de tiempo de CPU.

Desabilitar interrupciones

Deshabilitar interrupciones es un mecanismo que puede utilizarse en sistemas de un solo procesador para garantizar que una sección crítica se ejecute sin ser interrumpida. Sin embargo, no es adecuado para implementar primitivas de sincronización en sistemas multiprocesador por varias razones fundamentales:

1. No detiene a otros procesadores:

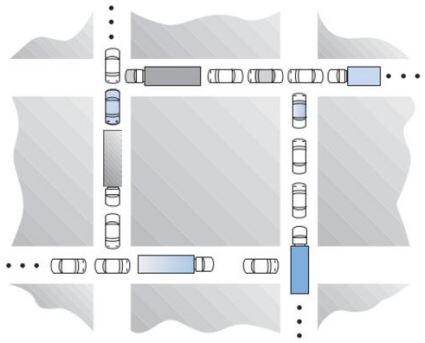
En un sistema multiprocesador, cada procesador tiene su propia unidad de ejecución y puede estar ejecutando hilos independientes en paralelo. Si deshabilitas las interrupciones en un procesador, los otros procesadores continúan ejecutándose normalmente, lo que significa que el acceso a los recursos compartidos entre procesadores no se sincroniza. Deshabilitar interrupciones en un solo procesador no previene que otros procesadores accedan a las secciones críticas, lo que puede llevar a condiciones de carrera y resultados impredecibles.

2. Riesgo de interbloqueo (deadlock):

Si deshabilitas las interrupciones, los procesadores pueden quedar bloqueados esperando que otro procesador libere un recurso, lo que podría generar un interbloqueo. Esto es particularmente problemático en sistemas multiprocesador, ya que se pierde la capacidad de gestionar adecuadamente las situaciones en las que dos o más procesadores esperan por un recurso bloqueado por otro. Sin la capacidad de interrumpir el

proceso y reprogramar el acceso al recurso, los procesadores pueden quedarse bloqueados indefinidamente.

Interbloqueo



Indique el nombre de cada una de las cuatro condiciones necesarias para que exista un interbloqueo. Muestre que cada condición se presenta en este ejemplo.

Mutua exclusión.

Los recursos (en este caso, las intersecciones de tráfico) solo pueden ser ocupados por un vehículo a la vez. Cada vehículo ocupa una sección específica de la intersección, y ningún otro puede entrar mientras esta esté ocupada.

Retención y espera.

Un proceso (vehículo) retiene un recurso (una sección de la intersección) mientras espera por otro recurso (otra sección). Cada vehículo está reteniendo su posición actual mientras espera a que la sección delante de él se desocupe.

No expropiación (no apropiatividad).

Los recursos no pueden ser liberados por la fuerza; deben ser liberados voluntariamente por el proceso que los posee. Ningún vehículo puede retroceder o liberar su posición actual hasta que pueda avanzar.

Espera circular.

Existe una cadena cerrada de procesos donde cada proceso está esperando un recurso que está siendo ocupado por el siguiente proceso en la cadena. Los vehículos están bloqueados en un ciclo: cada uno espera que el vehículo delante de él se mueva, pero ninguno puede avanzar.

¿Cómo evitar el interbloqueo?

Regla: No entrar a la intersección hasta que haya espacio suficiente para cruzarla completamente.

Técnica aplicada: Implementar una política de entrada condicional basada en disponibilidad. Esto se conoce como el protocolo de "no bloqueo anticipado". Si los vehículos no ingresan a la intersección hasta que puedan atravesarla completamente, se elimina la posibilidad de formar una cadena de espera circular, ya que los recursos (secciones) no se bloquean innecesariamente.

Variables de condición como semáforos

```c
int done = 0;
mutex_t m = MUTEX_INIT;
cond_t c = COND_INIT;
void *child(void *arg) {
  printf("child\n");
  mutex_lock(&m);
  done = 1;
  cond_signal(&c);
  mutex_unlock(&m);
}
int main(int argc, char *argv[]) {
  pthread_t c;
  printf("parent: begin\n");
  pthread_create(c, NULL, child, NULL);
  mutex_lock(&m);
  while(done == 0)
    cond_wait(&c, &m);
  mutex_unlock(&m);
  printf("parent: end\n");
}
```

Implementación con variables de Condición (CV)

```c
sem_t s;
void *child(void *arg) {
  printf("child\n");
  sem_post(&s); // increment
}
int main(int argc, char *argv[]) {
  sem_init(&s, 0, 0);
  pthread_t c;
  printf("parent: begin\n");
  pthread_create(c, NULL, child, NULL);
  sem_wait(&s); // decrement, wait for counter == 0
  printf("parent: end\n");
}
```

Implementación con semáforos

Lock como semáforo

Código de implementación del lock empleando semáforos

```c
sem_init(sem_t *s, int initval) {
  s->value = initval
}

sem_wait(sem_t *s) {
  while (s->value <= 0)
    put_self_to_sleep();
  s->value--;
}

sem_post(sem_t *s) {
  s->value++;
  wake_one_waiting_thread();
}
```

```c
typedef struct __lock_t {
  // whatever data structs you need goes here
  sem_t value;
} lock_t;

void init(lock_t *lock) {
  // init code goes here
  sem_init(&lock->value, 0, 1);
}

void acquire(lock_t *lock) {
  // lock acquire code goes here
  sem_wait(&lock->value);
}

void release(lock_t *lock) {
  // lock release code goes here
  sem_post(&lock->value);
}
```

Problemas de concurrencia

Violación de atomicidad

- Hay una violación en la **serialización** deseada para los **accesos a memoria**.
  - Ejemplo en MySQL
    - Dos hilos diferentes acceden al campo `proc_info` en la estructura `struct thd`.

```
1   Thread1::
2   if(thd->proc_info){
3     ...
4     fputs(thd->proc_info , ...);
5     ...
6   }
7
8   Thread2::
9   thd->proc_info = NULL;
```

7

Violación de Orden

- El **orden de ejecución** deseado para dos accesos a memoria **se invierte!**
  - **A** debería ejecutarse siempre antes de **B**, pero **el orden no se asegura** en tiempo de ejecución.
  - Ejemplo:

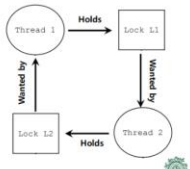
```
1   Thread1::
2   void init(){
3       mThread = PR_CreateThread(mMain, ...);
4   }
5
6   Thread2::
7   void mMain(...){
8       mState = mThread->State
9   }
```

10

Problemas con Interbloqueos (*deadlocks*)

```
Thread 1:        Thread 2:
lock(L1);        lock(L2);
lock(L2);        lock(L1);
```

- Presencia de un ciclo:
  - Thread1 adquiere el *lock* L1 y espera por el *lock* L2
  - Thread2 adquiere el *lock* L2 y espera por el *lock* L1



13