

Cheat Sheet - Sistemas operativos UDEA 2025-1

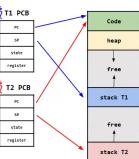
HILOS | CONCURRENCIA - LOCKS

Introducción

Un hilo es un punto de ejecución de un programa. Un programa puede tener uno o muchos hilos de ejecución. Todos los hilos de ejecución de un programa comparten AS.

Los hilos de un programa comparten AS pero cada hilo tiene su propio stack, su propio Program counter y sus propios registros. Esto significa que el código y el heap del proceso puede ser consultado desde cualquier hilo.

Thread Control Block: Esta estructura almacena el estado de cada hilo. Así no se pierde información en los cambios de contexto. Se encuentra dentro del PCB del proceso.



En el contexto Switch se guarda la información en el TCB del proceso que sale y se carga el TCB del proceso que entra.

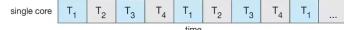


Múltiples hilos de ejecución

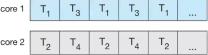
Paralelismo

Concurrencia vs Paralelismo

Concurrencia: Varias tareas se ejecutan en un solo procesador. El scheduler se encarga de proporcionarla.



Paralelismo: Varias tareas se ejecutan simultáneamente. Propio de sistemas multicore.



Tipos de paralelismo:

Paralelismo de datos: Divide los datos en subconjuntos. Cada subconjunto lo procesa un core y cada core aplica las mismas operaciones que los demás.



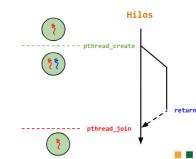
Paralelismo de tareas: Distribuye hilos a través de los núcleos de tal manera que cada hilo realiza una operación.



Creación de hilos

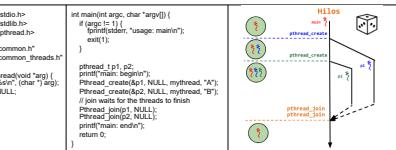
Creación de hilos

Similar a crear procesos (Llamada al sistema fork)



Supongamos que tenemos un programa con 2 funciones. En cierto momento llamamos pthread_create y se crea un hilo. Le decimos qué función queremos que ejecute. Al mismo tiempo se ejecutarán ambas funciones del código.

Un hilo padre puede esperar un hilo hijo, para que cuando ambos acaben continúe la ejecución normalmente.



El hilo padre de otro hilo, es siempre el que llame la función create. El orden de ejecución de los hilos es aleatorio ya que depende del scheduler.

Problemas de la creación de hilos:

Comportamiento no determinista: no sabemos el orden de ejecución de los hilos.

Datos compartidos: Por ejemplo cuando tenemos variables globales, y se crean 2 hilos que deben leer y modificar esa variable, se presentan comportamientos inesperados ya que se presenta la Condición de carrera.

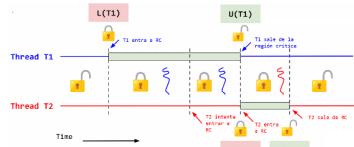
Condición de carrera: Es un problema que se presenta cuando se tienen varios hilos accediendo todos simultáneamente a variables modificándolas. El espacio donde generalmente se presentan estas competencias, se llama Sección crítica. Para solucionarlo, se plantea la condición mutua. Consiste en asegurar que si un proceso está accediendo a la Sección crítica, no puede ser interrumpida. Para esto necesitamos soportar atomicidad (Código que no puede ser interrumpido. Una sola instrucción):

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c, $0x1
```

Locks

Es un objeto en memoria que proporciona exclusión mutua. Que cuando algo acceda a una porción de memoria nadie más pueda acceder a ese mismo recurso. Si se encuentra adquirido un hilo adquirió el lock y se está ejecutando en la sección crítica. Si se encuentra disponible es porque nadie está accediendo a la región crítica y se puede acceder sin ningún problema.

Las funciones son:
acquire(): la cual espera a que un lock esté libre y lo toma
release(): la cual libera el lock



Métricas de implementación de locks

Mutual exclusion: Si hay un hilo en una RC, debe ser único y no se puede quitar.
Fairness: Cada hilo tiene igual probabilidad de adquirir el lock.
Performance (Desempeño): Tiempo tomado en la adquisición/liberación con o sin competencia.

Locks con inst. load/store

```
1 typedef struct __lock_t {
2     int flag;
3 } lock_t;
4
5 void init(lock_t *mutex) {
6     // &lock -> lock es available, 1 -> held
7     mutex->flag = 0;
8 }
9
10 void lock(lock_t *mutex) {
11     while (lock->flag == 1) // TEST the flag
12         ; // spin-wait (do nothing)
13     mutex->flag = 1; // now SET it!
14 }
15
16 void unlock(lock_t *mutex) {
17     mutex->flag = 0;
18 }
```

Este método para implementación de locks se basa una variable global llamada flag la cual está en 1 si un proceso está accediendo a la RC. Y 0 si no.

Desventajas:
 Se producen condiciones de carrera por interrupciones para cambios de contexto.

La exclusión mutua no se satisface. Presenta un mal desempeño porque hay un ciclo que está gastándose solo preguntando.

Instrucciones atómicas test-and-set

```
int TestAndSet(int *old_ptr, int new_) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new; // store 'new' into old_ptr
    return old; // return the old value
}
```

```
1 typedef struct __lock_t {
2     int flag;
3     lock_t;
4 }
5
6 void init(lock_t *lock) {
7     // => lock is available, 1: lock is held
8     lock->flag = 0;
9 }
10
11 void lockAndSet(lock_t *lock) {
12     while (TestAndSet(&lock->flag, 1) == 1)
13         ; // spin-wait (do nothing)
14 }
15
16 void unlock(lock_t *lock) {
17     /lock->flag = 0;
18     TestAndSet(&lock->flag, 0);
19 }
```

Estas instrucciones garantizan que las operaciones de lectura, modificación y escritura se hagan con una sola instrucción, la cual no puede ser interrumpida. Devuelve el valor anterior y actualiza de manera simultánea el valor de una posición de memoria.

Variables de condición

Es una manera de implementar sincronización entre hilos.

Funciones wait y signal.
 Estas funciones usan una variable de condición y un mutex para enviar a "dormir" o "despertar" los hilos en cuestión.

wait(cond_t *cv, mutex_t *mutex)

- Un hilo se pone en una cola (a dormir) a la espera de que un estado deseado suceda.
- Recibe un mutex como parámetro.
- Cuando se llama el wait el lock es liberado y se pone el hilo a dormir.

signal(cond_t *cv)

- Despierta un único hilo de los que se encuentran en la cola de espera (if >= 1 el hilo está esperando).
- Si no hay hilos esperando, solo retorna sin hacer nada.
- Cuando el hilo se despierte se debe adquirir de nuevo el lock.

Con Pthread podemos implementar mutex y variables de condición:

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

Ejemplo de implementación en C y prueba de escritorio en el siguiente link.

Ticket-lock usando Fetch-and-Add

La instrucción Fetch and Add es la siguiente:

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

Luego, para implementar locks con esta instrucción, tendremos 2 flags (ticket y turn) y:

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn) {
        ; // spin
    }
}

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

El ticket para cada hilo representa como su boleto para la región crítica. El turn representa a cuál se debe atender. Entonces el ticket del hilo es 3, entonces solo podrá acceder a la región crítica cuando el turn sea 3.

Ambos inician en 0.

Instrucciones atómicas compare-And-Swap

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}
```

Estas instrucción atómica actualiza una localización de memoria con un valor anterior sólo cuando este es igual al esperado. Trabaja similar al test-and-set.

Si *ptr == esperado → actualiza la memoria al valor new.

Si *ptr != esperado → retorna el valor actual.

Instrucciones atómicas LL&SC

Load Linked: Opera como un load típico.
Store Conditional: El almacenamiento solo funciona si no se ha realizado otro almacenamiento previo en el mismo lugar.

Éxito: Retorna 1 y actualiza *ptr a new.

Falla: Retorna 0 y no actualiza memoria

```
void lock(lock_t *lock) {
    while (1) {
        while (Loadlinked(&lock->flag) == 1)
            ; // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
        else // otherwise: try it all over again
    }
}

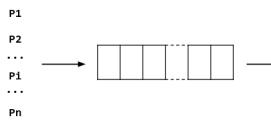
void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Cheat Sheet - Sistemas operativos UDEA 2025-1

VARIABLES DE CONDICIÓN - SEMÁFOROS

Problema del Productor - Consumidor

Varios productores que ubican ítems en un buffer. Los consumidores retiran estos elementos del buffer y los consumen. Tendremos entonces 2 funciones: `put()` que la usan los productores para meter elementos al buffer y `get()` usada por los consumidores para obtener elementos del buffer. La RC que definimos en este problema está conformada por: El buffer y la variable count.



Hay varias estrategias que se pueden implementar:

- Única CV y sentencia if (Un solo productor y un solo consumidor)
- Única CV y sentencia while (Un solo productor y varios consumidores)
- Single buffer con 2 CV y sentencia while (Un solo productor y varios consumidores)

Semáforos

Permite la sincronización entre recursos compartidos. Es una estructura de control que permite tener exclusión mutua y permite realizar la sincronización. Usando esta estructura se puede implementar locks y variables de condición. Un semáforo es un valor entero que tiene dos operaciones clave: `wait()`: disminuye el valor del semáforo y si el valor es menor que cero, el proceso se bloquea (espera) y `signal()`: Aumenta el valor del semáforo y si hay procesos esperando, despierta uno de ellos.

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}  
  
signal(S) {  
    S++;  
}
```

Semáforo como CV

Aunque los semáforos y las variables de condición tienen propósitos distintos, es posible usar semáforos para implementar un comportamiento similar al de una variable de condición. En este caso, los hilos bloqueados llaman a `wait()` y quedan detenidos hasta que otro hilo ejecuta `signal()`, actuando como si fuera una notificación. Sin embargo, a diferencia de las variables de condición, los semáforos mantienen el conteo incluso si no hay hilos esperando, lo que puede generar señales "perdidas" o no deseadas si no se gestionan cuidadosamente. Por eso, aunque es viable, no es la opción más precisa ni segura para sustituir directamente una variable de condición.

```
int done = 0;  
mutex_t m = MUXEX_INIT;  
cond_t c = COND_INIT;  
void *child(void *arg) {  
    printf("child\n");  
    sem_post(&s); // increment  
}  
int main(int argc, char *argv[]) {  
    pthread_t t;  
    printf("parent: begin\n");  
    pthread_create(&t, 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)
```

Problema de los filósofos

El problema de los filósofos, planteado por Dijkstra, representa un escenario clásico de sincronización donde cinco filósofos se turnan para pensar y comer, necesitando dos palillos chinos (uno a cada lado) para alimentarse. Cada palillo está protegido por un semáforo, y los filósofos deben adquirir ambos antes de entrar a la sección crítica (comer).



La implementación básica puede llevar a un interbloqueo (deadlock) si todos toman un palillo y esperan el segundo indefinidamente.

```
Basic loop of each philosopher  
while(1) {  
    think();  
    getforks();  
    eat();  
    putforks();  
}  
  
Helper functions (Downey's solutions)  
// helper functions  
int left(int p) {  
    return p;  
}  
  
int right(int p) {  
    return (p + 1) % 5;  
}
```

Única CV y sentencia if

La CV a usar tiene como propósito hacer que los consumidores esperen cuando el buffer esté vacío o que los productores esperen cuando el buffer esté lleno. Esta CV solo verifica una vez antes de esperar, además si el hilo es despertado, asume que puede continuar sin verificar otra vez, lo que puede llevar a que acceda a un buffer vacío o lleno incorrectamente.

```
int loops; // must initialize somewhere...  
cond_t cond;  
mutex_t mutex;  
void *producer(void *arg) {  
    int i;  
    for (i = 0; i < loops; i++) {  
        pthread_mutex_lock(&mutex); // p1  
        if (count == 1) // p2  
            pthread_cond_wait(&cond, &mutex); // p3  
        tmp = get(); // p4  
        pthread_cond_signal(&cond); // p5  
        pthread_mutex_unlock(&mutex); // p6  
        printf("%d\n", tmp);  
    }  
}  
  
void *consumer(void *arg) {  
    int i;  
    while (1) {  
        pthread_mutex_lock(&mutex); // p1  
        if (count == 0) // p2  
            pthread_cond_wait(&cond, &mutex); // p3  
        tmp = get(); // p4  
        pthread_cond_signal(&cond); // p5  
        pthread_mutex_unlock(&mutex); // p6  
        printf("%d\n", tmp);  
    }  
}
```

Tipos de Semáforos

Existen dos tipos de semáforos: el semáforo binario, que solo puede tener los valores 0 o 1 y se usa para controlar acceso exclusivo a recursos, funcionando de forma similar a un mutex; y el semáforo contador (o general), que puede tener valores mayores a 1 y permite gestionar múltiples instancias de un recurso compartido, como por ejemplo varios espacios en un buffer. Ambos se usan para sincronizar procesos o hilos en entornos concurrentes.

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}  
  
signal(S) {  
    S++;  
}
```

Semáforo como lock

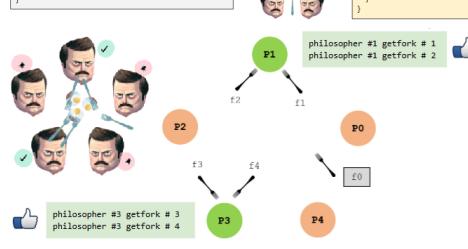
Un semáforo, especialmente el binario, puede utilizarse como un mecanismo de bloqueo o lock para controlar el acceso exclusivo a secciones críticas en entornos concurrentes. Al usar `wait()` para bloquear y `signal()` para liberar, cumple una función similar a la de un mutex. Sin embargo, a diferencia del mutex, el semáforo no tiene propiedad, lo que significa que cualquier hilo puede liberar el recurso, incluso si no fue quien lo bloqueó. Por esta razón, aunque un semáforo puede actuar como lock, se recomienda usar mutexes cuando se requiere un control más estricto sobre la exclusión mutua.

```
sem_t lock;  
void *worker(void *arg) {  
    int i;  
    sem_wait(&lock); // What goes here?  
    for (i = 0; i < 1e6; i++)  
        counter++;  
    sem_post(&lock); // What goes here?  
    return NULL;  
}  
  
int main(int argc, char *argv[]) {  
    int num = atoi(argv[1]);  
    pthread_t pid[PMAX];  
    sem_init(&lock, 0, 1 /* What goes here? */);  
    for (int i = 0; i < num; i++)  
        Pthread_create(&pid[i], 0, worker, 0);  
    for (int i = 0; i < num; i++)  
        Pthread_join(pid[i], NULL);  
    printf("counter: %d\n", counter);  
    return 0;  
}
```

Problema de los filósofos - Solución 1

Consiste en representar cada palillo como un semáforo binario inicializado en 1, de modo que cada filósofo, al querer comer, debe adquirir los dos palillos vecinos usando `sem_wait()` y luego liberarlos con `sem_post()` al terminar de comer. Esta implementación permite que los filósofos piensen y coman alternadamente, pero tiene un defecto importante: si todos los filósofos toman el palillo de su izquierda al mismo tiempo, se produce un interbloqueo (deadlock), ya que todos quedan esperando eternamente el palillo derecho. Esta solución es funcional pero insegura sin mecanismos adicionales para romper el ciclo de espera circular.

```
int main(int argc, char *argv[]) {  
    for (int i = 0; i < 5; i++) {  
        sem_init(forks[i], 0, 1); // initialized to 1  
    } // ...  
}  
  
void philosopher(int i) {  
    while(1) {  
        think();  
        getforks();  
        eat();  
        putforks();  
    }  
}
```



Única CV y sentencia while

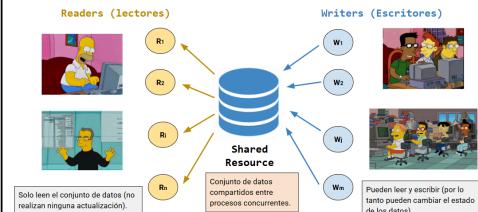
La CV a usar tiene como propósito hacer que los consumidores esperen cuando el buffer esté vacío o que los productores esperen cuando el buffer esté lleno. Esta CV es compartida por ambos roles, y siempre se emplea junto con una sentencia while para reevaluar la condición tras cada espera, ya que un hilo puede ser despertado sin que la condición haya cambiado, evitando así errores de concurrencia.

```
int loops; // must initialize somewhere...  
cond_t cond;  
mutex_t mutex;  
void *producer(void *arg) {  
    int i;  
    for (i = 0; i < loops; i++) {  
        pthread_mutex_lock(&mutex); // p1  
        if (count == 1) // p2  
            pthread_cond_wait(&cond, &mutex); // p3  
        tmp = get(); // p4  
        pthread_cond_signal(&cond); // p5  
        pthread_mutex_unlock(&mutex); // p6  
        printf("%d\n", tmp);  
    }  
}
```

```
void *consumer(void *arg) {  
    int i;  
    for (i = 0; i < loops; i++) {  
        pthread_mutex_lock(&mutex); // c1  
        if (count == 0) // c2  
            pthread_cond_wait(&cond, &mutex); // c3  
        tmp = get(); // c4  
        pthread_cond_signal(&cond); // c5  
        pthread_mutex_unlock(&mutex); // c6  
        printf("%d\n", tmp);  
    }  
}
```

Problema de Lectores - Escritores

Es un problema clásico de concurrencia que surge cuando múltiples hilos necesitan acceder a un recurso compartido, como una base de datos. Los lectores solo leen el recurso, por lo que pueden hacerlo simultáneamente sin conflictos; sin embargo, los escritores modifican el recurso y requieren acceso exclusivo. La dificultad radica en permitir múltiples lecturas concurrentes sin bloquear innecesariamente, pero garantizando que cuando un escritor acceda, ningún otro lector (o escritor) interfiere. Además, debe evitarse que lectores o escritores queden esperando indefinidamente, lo que lleva a variantes del problema según a quién se le dé prioridad.

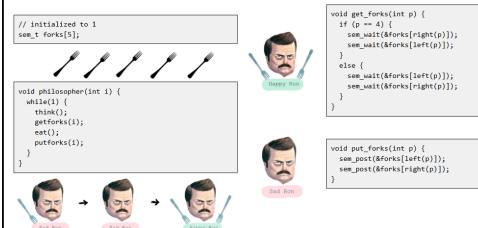


Para resolverlo, se implementa una estructura llamada `rwlock_t`, que usa semáforos para coordinar el acceso: un semáforo de escritura (`rwritelock`), uno de lectura (`lock`) y un contador de lectores activos. Cuando un lector entra, incrementa el contador y, si es el primero, bloquea el acceso a escritores; al salir, el último lector libera ese bloqueo. Los escritores, por su parte, solo pueden entrar cuando ningún lector ni otro escritor esté activo. Esta solución permite múltiples lectores concurrentes sin interferencia, pero puede generar starvation de escritores si los lectores no paran de llegar.

```
typedef _struc_t rwlock_t {  
    sem_t lock; // binary semaphore (basic lock)  
    sem_t writelock; // used to allow ONE writer or MANY  
    int readers; // count of readers reading in  
} rwlock_t;  
  
void rwlock_init(rwlock_t *rw) {  
    rw->readers = 0;  
    sem_init(&rw->lock, 0, 1);  
    sem_init(&rw->writelock, 0, 1);  
}
```

Problema de los filósofos - Solución al interbloqueo

Para evitarlo, se propone una solución simple: hacer que uno de los filósofos (por ejemplo, el último) tome primero el palillo derecho en lugar de la izquierdo, rompiendo así el ciclo circular de esperas. Esta estrategia elimina el deadlock, aunque no garantiza que todos los filósofos puedan comer siempre, dejando abierto la posibilidad de inanición (starvation), otro de los desafíos clásicos en este problema de concurrencia.



Creado por:

Cristian Daniel Muñoz Botero
cristian.munoz3@udea.edu.co

Brandon Duque García
brandon.duque@udea.edu.co