

Sistemas operativos

Ayudantía 4: Threads

Profesores: Martín Gutiérrez
y Víctor Reyes

Ayudantes: Diego Banda y
Dante Hortuvia



Contacto



dante.hortuvia@mail_udp.cl



doshuertos



github.com/Doshuertos/Ayudantias_SO_CIT2010

Sección 1



diego.banda@mail_udp.cl



darklouds



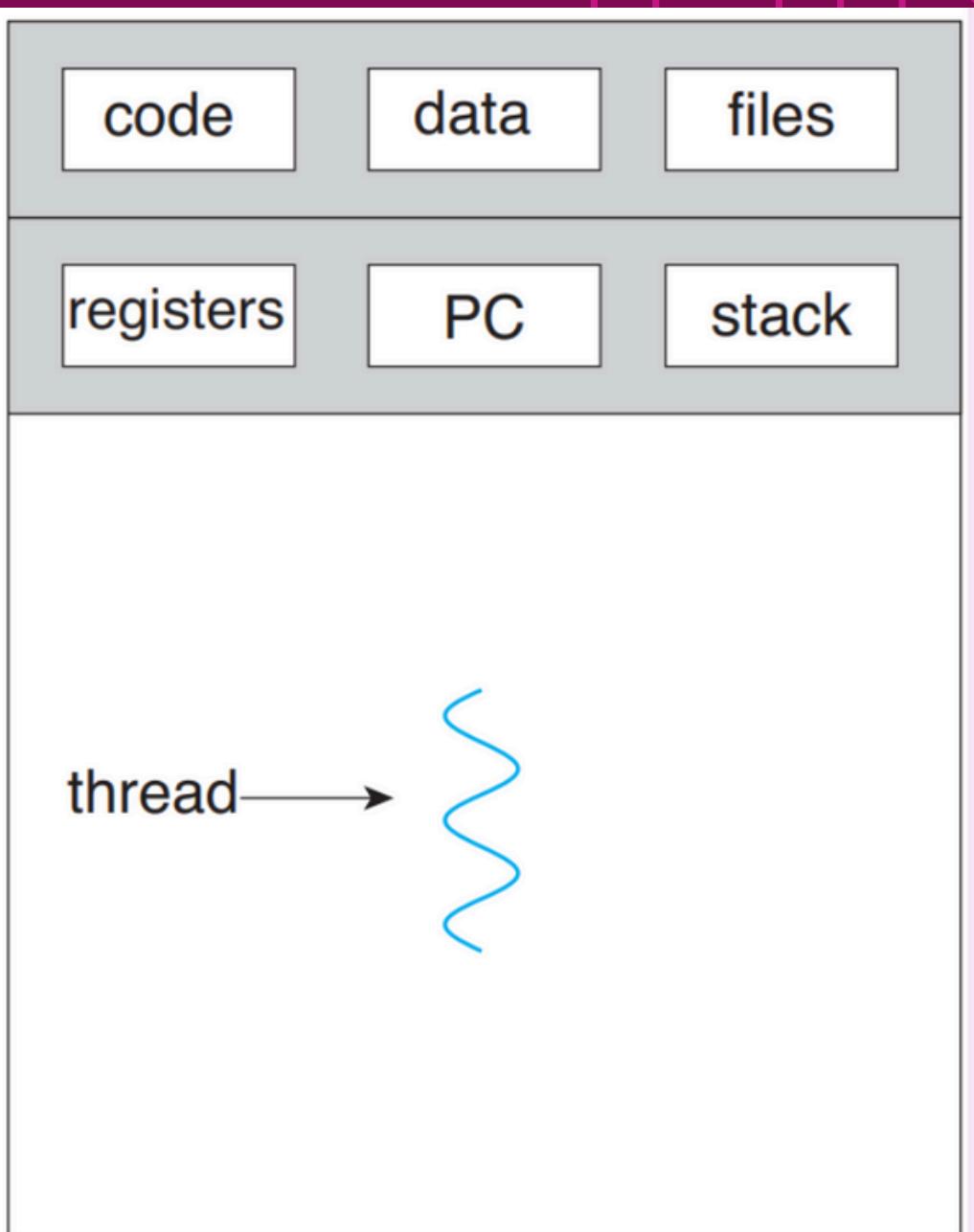
github.com/DiegoBan/SO2025-1

Sección 2

¿Qué es un thread?

Se puede decir que es como un mini-proceso dentro del un proceso (como los antes vistos) hechos para realizar tareas específicas (como una función).

Son más ligeros que realizar otro proceso nuevo, ya que comparten datos y recursos con el proceso en el que se ejecutan.



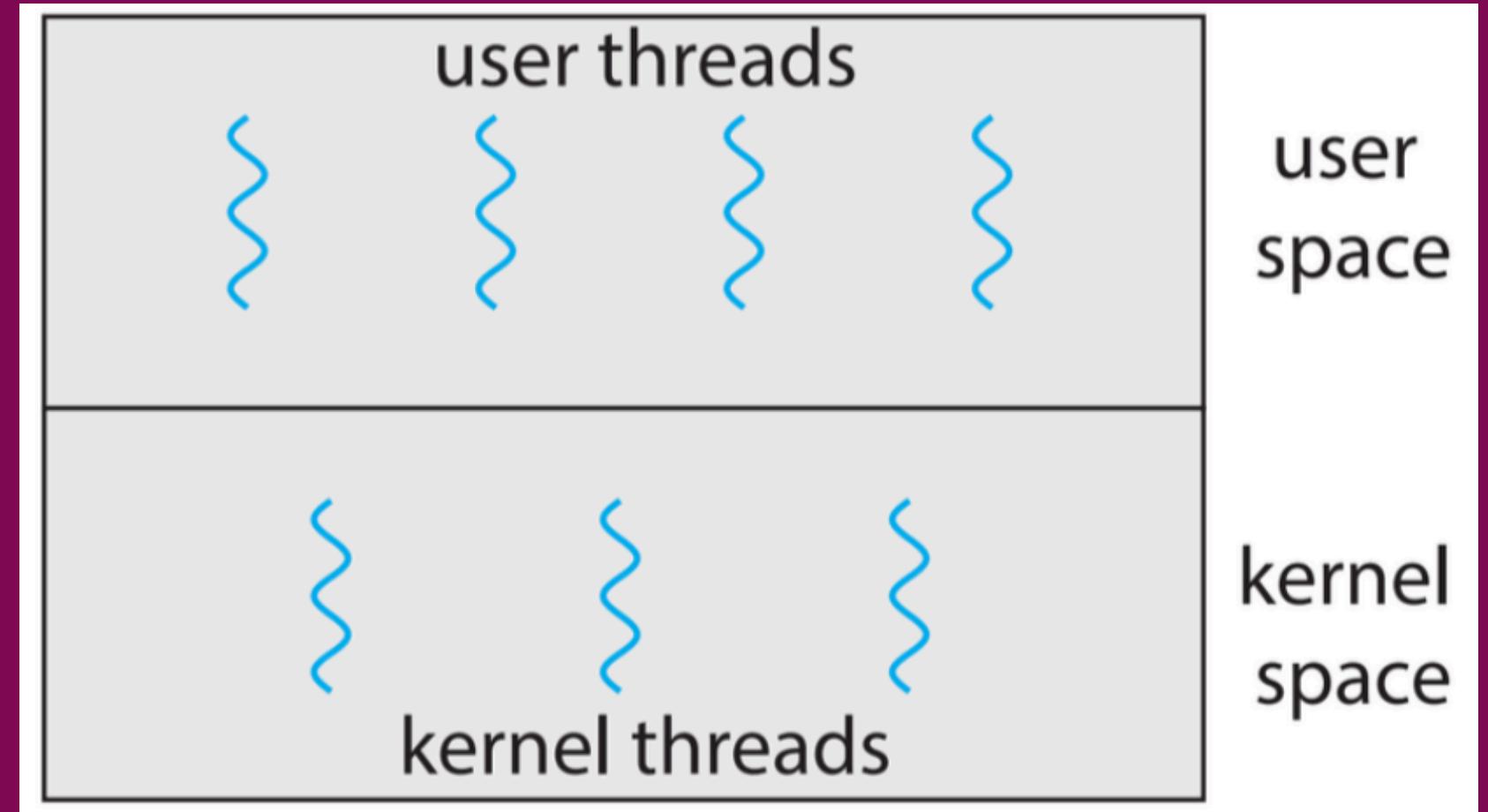
Tipos de threads

User level threads

Creados en el user space, no necesitan intervención del SO. Son más rápidos y flexibles.

Kernel level threads

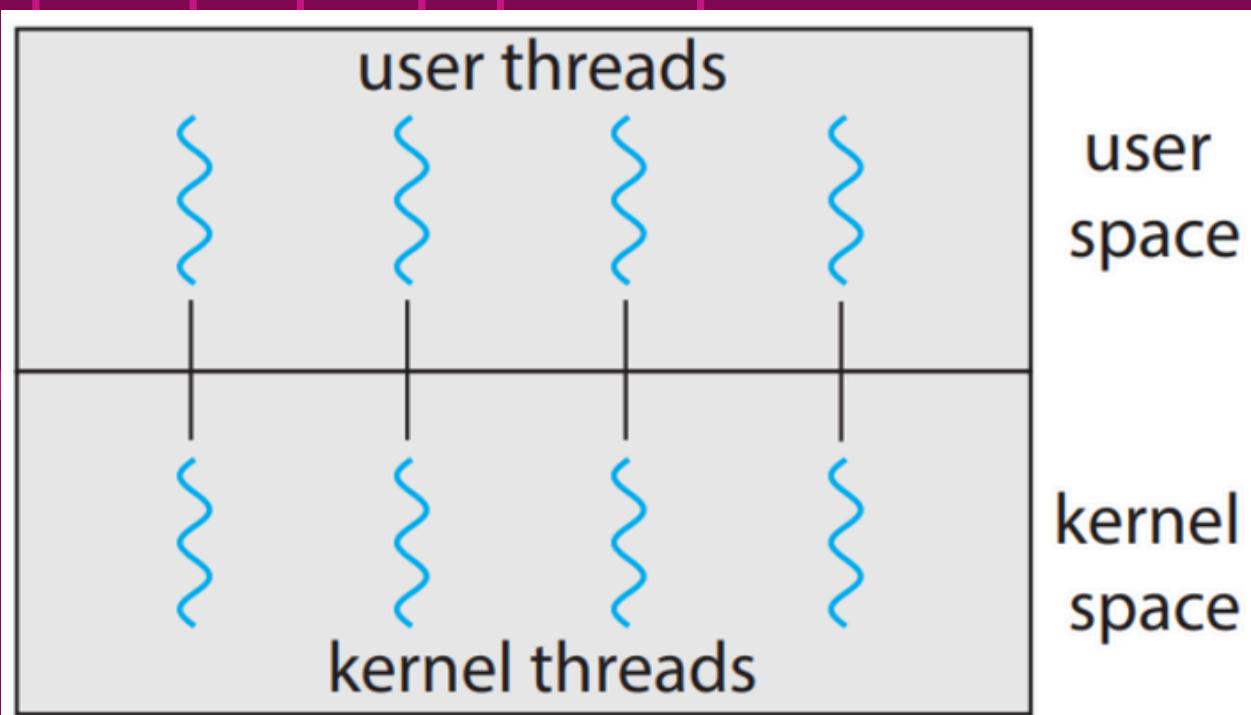
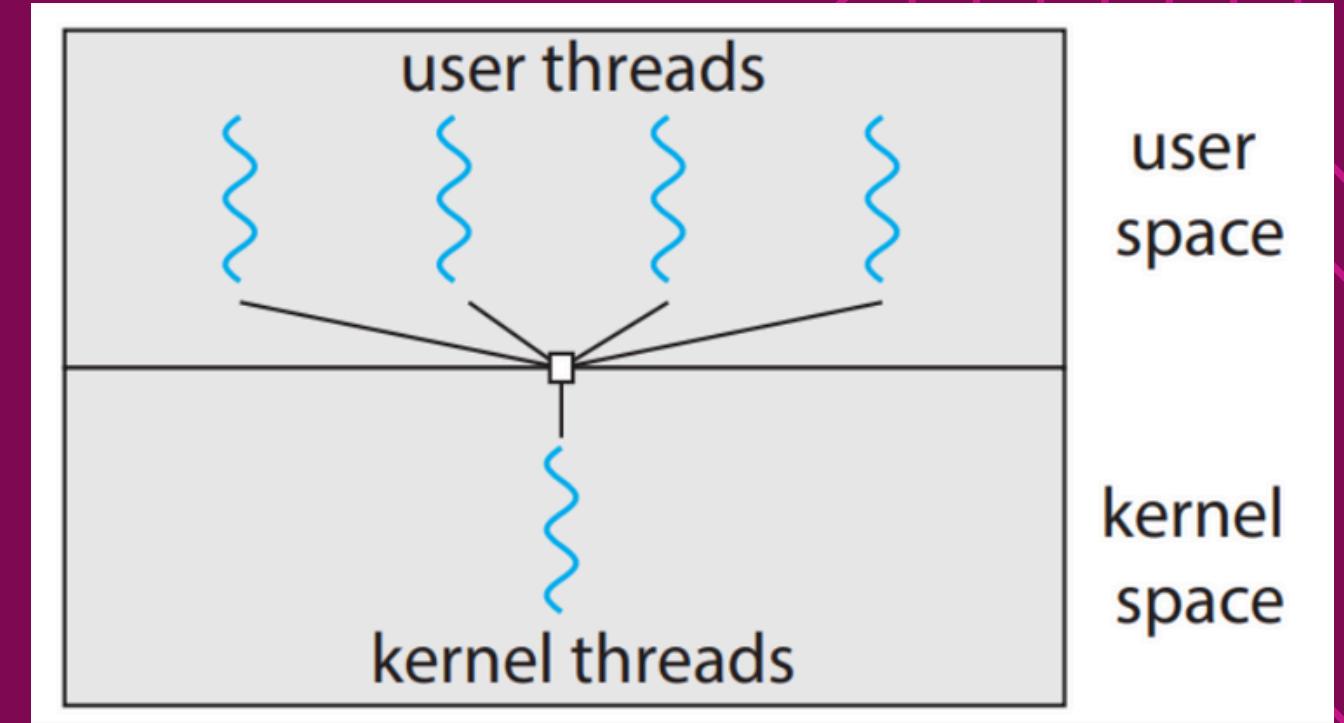
Creados y usados en el kernel space, dependen del SO, son más costosos y lentos de crear, sin embargo, se tiene más control sobre ellos al aprovechar funcionalidades de kernel.



Multithreads!!

Many to One

Multiples user level threads asociados a un kernel level thread, la gestión se realiza a nivel usuario, solo uno puede entrar al modo kernel a la vez. No hay paralelismo, todo se bloquea si se realiza una syscall bloqueante.



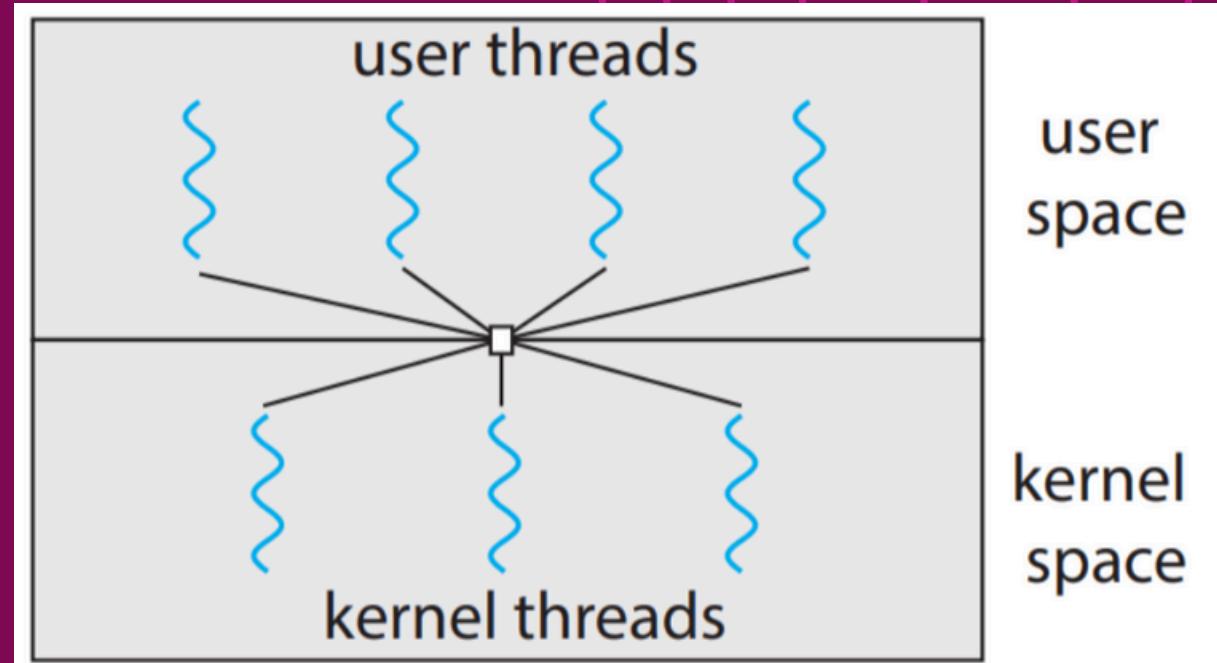
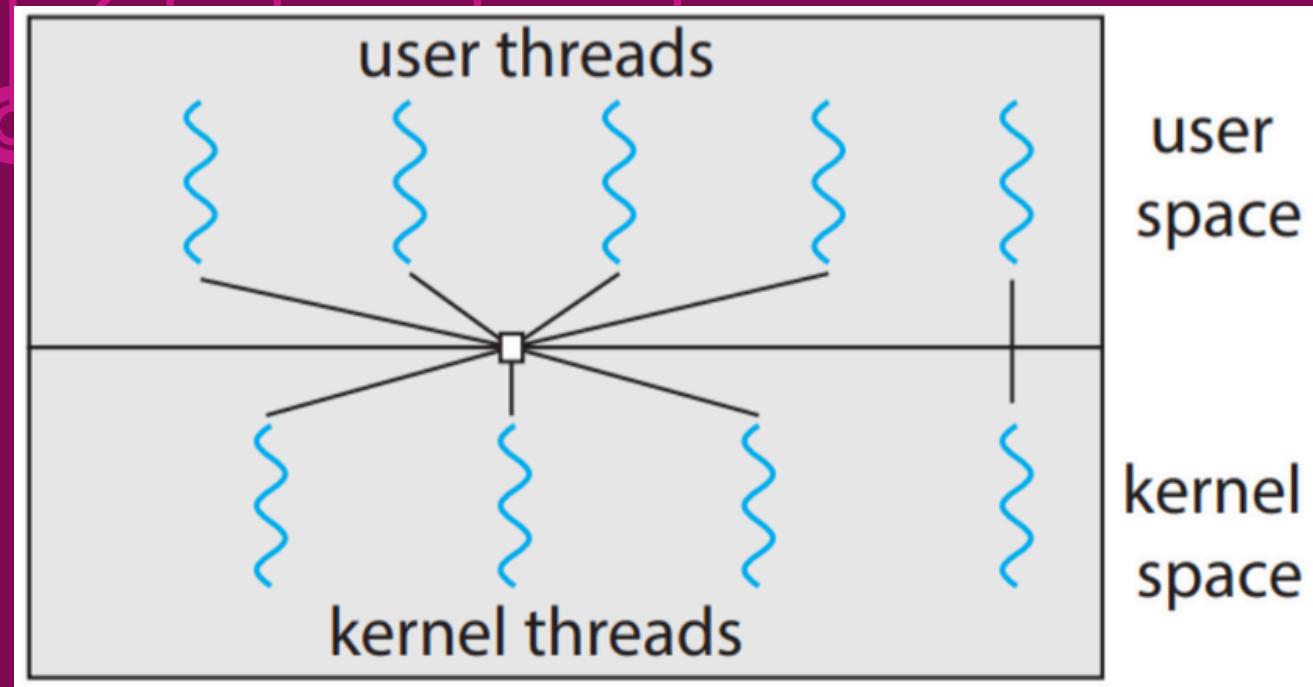
One to one

Cada thread tiene su contraparte, para cada kernel level thread hay un user level thread. Permite paralelismo ante syscalls bloqueantes, pero muchos kernel level threads pueden afectar al rendimiento del SO.

Multithreads!!

Many to many

x cantidad de user level threads conectados a una igual o menor cantidad de kernel level threads, da flexibilidad a la asignación de recursos y parallelismo.



Two level (mixto)

Variación de many to many, agrega la opción de tener user level threads asociados directamente a un kernel level thread.

Sección Crítica

Región de la memoria de acceso compartido (variables globales), el acceso y modificación de estas variables puede cambiar el resultado del código, dando como resultado outputs que no sabemos (cosas que no queremos que sucedan jeje).

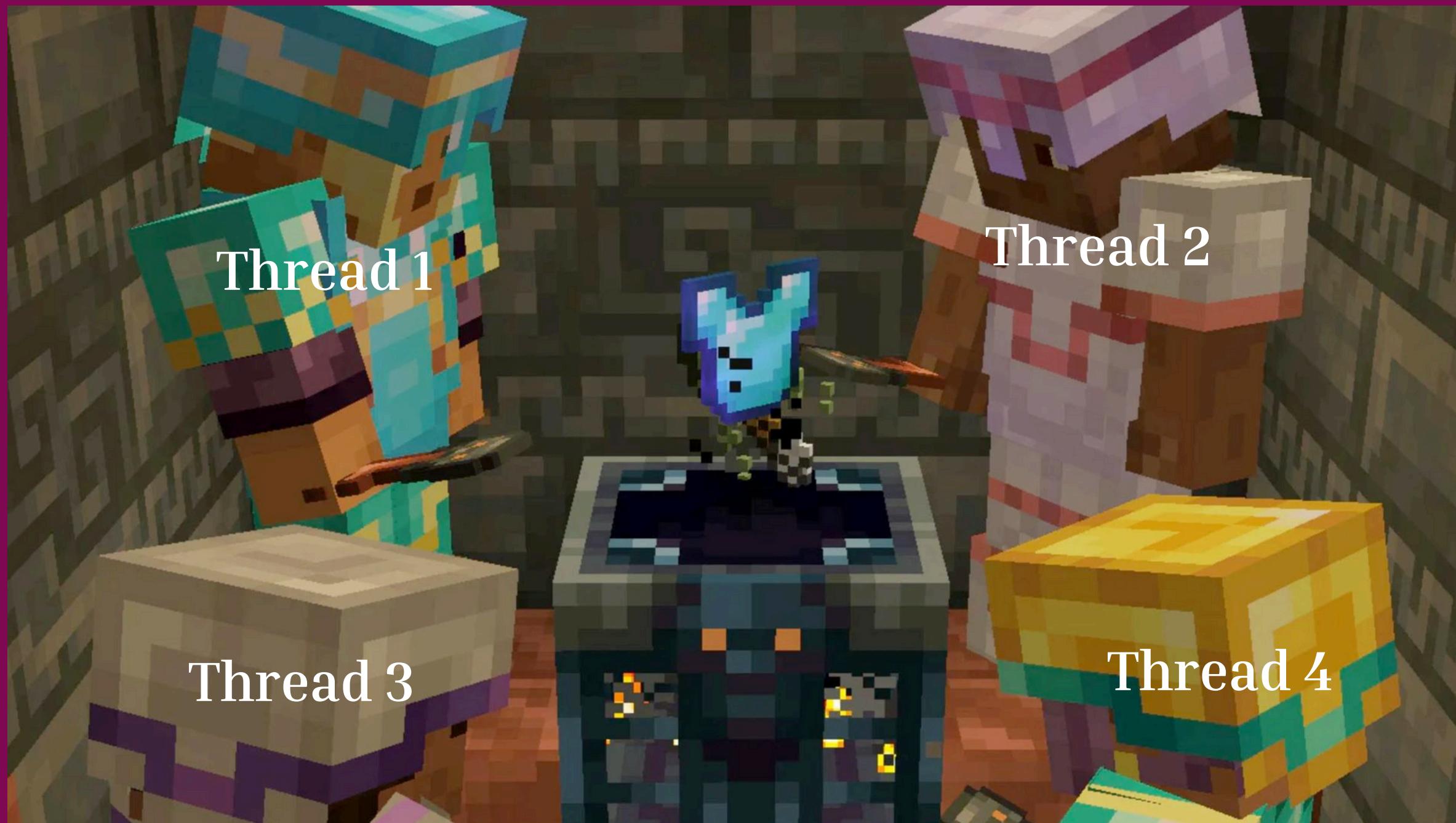
Respecto a la sección critica se busca idealmente:

- **Exclusión Mutua:** A lo más un thread/proceso en la SC.
- **Progreso:** Al menos un thread/proceso puede entrar en la SC, en caso que 2 o más quieran acceder a esta SC, se debe decidir cual en un tiempo acotado.
- **Ausencia de Inanición:** Si un thread/proceso quiere acceder a su SC, eventualmente podrá hacerlo después de un tiempo acotado.



Race Conditions

Cuando dos o más threads/procesos compiten por modificar una variable global, llevando a outputs erroneos y posiblemente imposibles de predecir.



Sincronización

¿Cómo se evita todo lo anterior? Sincronizando procesos. Opciones disponibles:

- Desactivar las interrupciones de tu SO (no recomendable jeje)

- Spinlock:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4
5 int lock = 0;
6
7 void *thread_func(void *arg){
8     while(lock);
9     lock = 1;
10    // SC
11    lock = 0;
12 }
13
14 int main(){
15     pthread_t threads[4];
16     for(int i = 0 ; i < 4 ; i++){
17         pthread_create(&threads[i], NULL, thread_func, NULL);
18     }
19     for(int i = 0 ; i < 4 ; i++){
20         pthread_join(threads[i], NULL);
21     }
22 }
23 }
```

Sincronización

- Algoritmo de peterson (Para 2 threads):

```
bandera[0] = false
bandera[1] = false
turno      // No es necesario asignar un turno
p0: bandera[0] = true
turno = 1
while( bandera[1] && turno == 1 );
//{ no hace nada; espera. }
// sección crítica

// fin de la sección crítica
bandera[0] = false

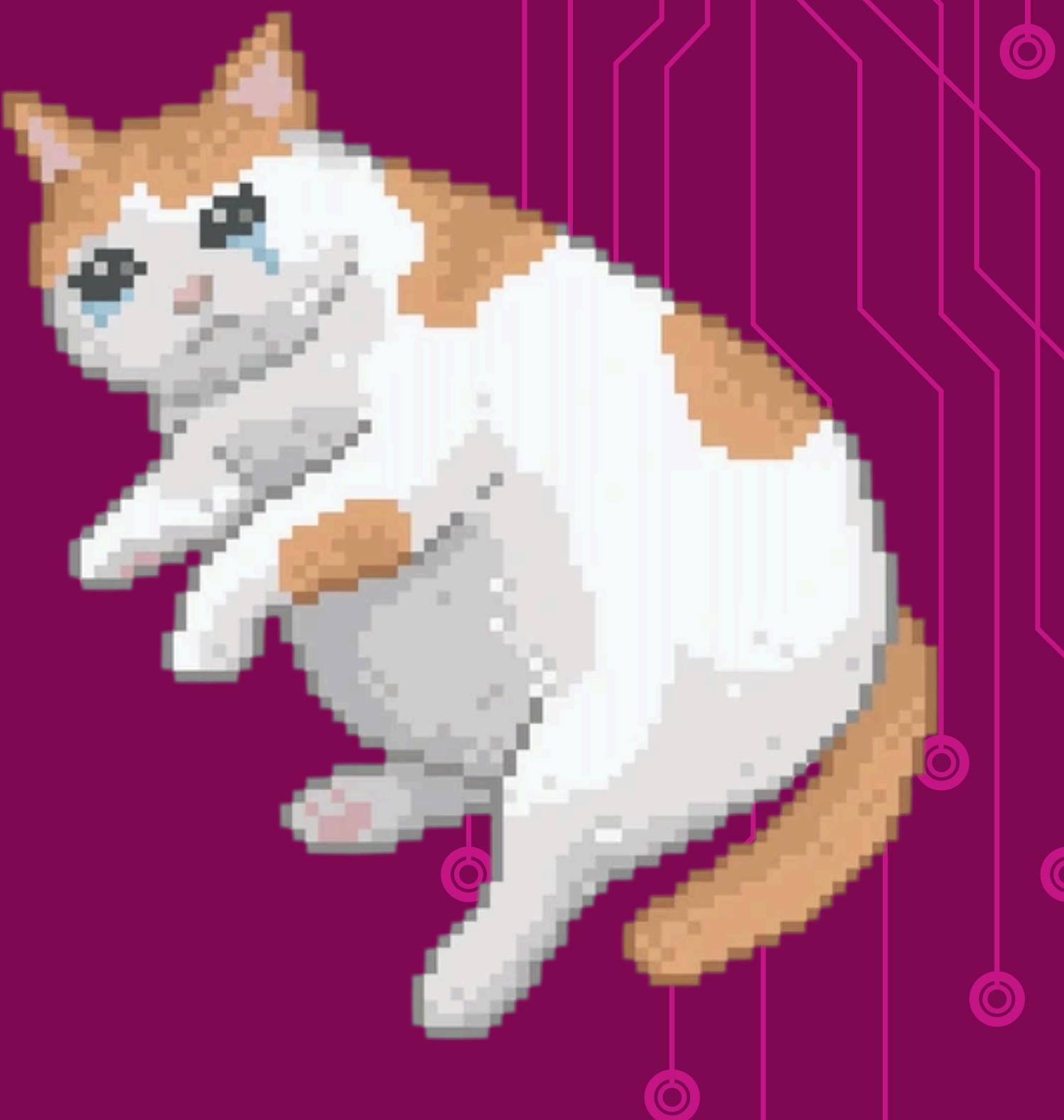
p1: bandera[1] = true
turno = 0
while( bandera[0] && turno == 0 );
//{ no hace nada; espera. }
// sección crítica

// fin de la sección crítica
bandera[1] = false
```

Pueden fallar si hay cambios de contexto, ya que no son atómicos, son instrucciones ejecutadas en distintas líneas, para evitar esto hay técnicas como “test_and_set” o “compare_and_swap”

Busy Waiting

Estas técnicas generan busy waiting, que en palabras simples, los threads que estén en el while esperando su turno de entrar a la SC estarán gastando tiempo de CPU también, aunque en realidad no estén haciendo absolutamente nada.

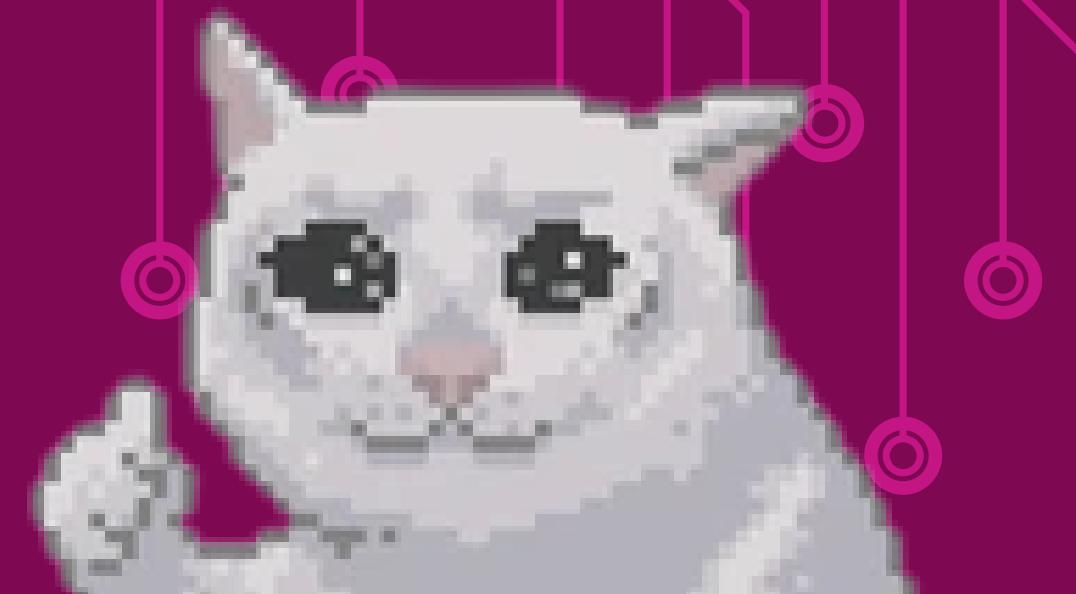


¿Sincronización sin Busy Waiting?

Existen técnicas implementadas en los lenguajes de programación (C y C++) que son atómicas y no generan Busy Waiting, pero, ¿como es esto posible?

Estas técnicas ponen a dormir los threads que no entran en la SC y cuando sea su turno se reactivan.

MUTEX!!! :D



Mutex

Hace que solo un thread/proceso entre en la SC.

Utiliza:

- `pthread_mutex_t lock`: Para creación de variable que bloqueará x segmento.
- `pthread_mutex_init(&lock, NULL)`: Para inicialización de Mutex.
- `pthread_mutex_lock(&lock)`: Para bloquear segmento.
- `pthread_mutex_unlock(&lock)`: Para desbloquear segmento anteriormente bloqueado.
- `pthread_mutex_destroy(&lock)`: Para destruir mutex al finalizar código y ejecución.



Mutex

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 pthread_mutex_t lock;
7
8 void *thread_func(void *arg){
9     pthread_mutex_lock(&lock);
10    // SC
11    pthread_mutex_unlock(&lock);
12    return NULL;
13 }
14
15 int main(){
16     pthread_t threads[4];
17     pthread_mutex_init(&lock, NULL);
18     for(int i = 0 ; i < 4 ; i++){
19         pthread_create(threads[i], NULL, thread_func, NULL);
20     }
21     for(int i = 0 ; i < 4 ; i++){
22         pthread_join(threads[i], NULL);
23     }
24     pthread_mutex_destroy(&lock);
25 }
```



Semáforos

Lo mismo que los mutex, pero utilizados cuando más de un thread debe entrar a la SC o a x segmento, funciona como un contador.

Utiliza:

- **sem_t nombre:** Creación de variable a utilizar.
- **sem_init(&nombre, 0, numero de threads a entrar a la SC):**
Inicialización y selección de cuantos procesos podrán pasar por el semáforo antes de que este los bloquee.
- **sem_wait(&nombre):** Baja en 1 el contador, en caso de que el contador ya esté en 0, el thread queda en espera.
- **sem_post(&nombre):** Aumenta en 1 el contador, indica que un proceso ya salió de la SC.
- **sem_destroy(&nombre):** Para eliminar semaforo.



Semáforos

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <unistd.h>
5  #include <semaphore.h>
6
7  sem_t semaforo;
8
9  void* thread_func(void* arg){
10     sem_wait(&semaforo);
11     // SC
12     sem_post(&semaforo);
13     return NULL;
14 }
15
16 int main(){
17     sem_init(&semaforo, 0, 2);
18     pthread_t threads[4];
19     for(int i = 0 ; i < 4 ; i++){
20         pthread_create(&threads[i], NULL, thread_func, NULL);
21     }
22     for(int i = 0 ; i < 4 ; i++){
23         pthread_join(threads[i], NULL);
24     }
25 }
26 }
```



Deadlock

Dos o más threads se bloquean mutuamente, quedando atrapados para toda la eternidad esperando que el otro los desbloquee. Se pueden dar en distintas situaciones, normalmente por un mal manejo de la sincronización para la SC.

Livelock

Livelock es más complejo de ocurrir y raro que un deadlock, en este se bloquean y **no progresan** en las instrucciones, sin embargo, si están cambiando de estado y/o realizando tareas.

Ejercicio

```
#include <stdio.h>

int lock = 0;

void* mythread(void* arg) {
    while (lock);
    lock = 1;
    // sección crítica
    lock = 0;
    return NULL;
}
```

¿Qué problema(s) podrían existir? Explíquelos y proponga una solución para uno de ellos .



Ejercicio

```
pthread_mutex_t pl;
int arr[10];
int flag = 0;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

void* rp() {
    int i = 0;
    printf("\nIngrese valores\n");
    for (i = 0; i < 4; i++) {
        scanf("%d", &arr[i]);
    }
    pthread_mutex_lock(&pl);
    pthread_cond_wait(&cv, &pl);
    pthread_mutex_unlock(&pl);
    pthread_exit(NULL);
}

void* ev() {
    int i = 0;
    pthread_mutex_lock(&pl);
    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&pl);
    printf("Los valores ingresados en el arreglo son:");
    for (i = 0; i < 4; i++) {
        printf("\n%d\n", arr[i]);
    }
    pthread_exit(NULL);
}
```

¿Qué problema(s) tiene el código?
Explíquelo(s) y proponga una solución



Ejercicio

Viene un evento épico, el Monsters of Rock 2023. En dicho evento se presentaran varias bandas de rock old-school como Scorpions, Helloween, Kiss, entre otros. Se cuenta con dos tipos de entrada para los asistentes: cancha (sin enumeración) y galería (con enumeración, en donde el sistema asigna el lugar). Por protocolo de la organización, todo asistente que se encuentre en galería debe estar en su asiento correspondiente. Suponga que usted debe implementar el sistema de compras para dicho evento, en donde M asientos son asignados para galería y N cupos para cancha. Asuma que de manera concurrente podrían existir un numero indeterminado de personas interesadas en comprar entradas. En base a lo anterior, responda las siguientes preguntas:

1. Identifique los elementos del problema y abstráigalos a la terminología que hemos empleado en la asignatura
2. Escriba snippets de código que permitan implementar una solución al problema.
3. El evento ha tenido tan nivel de éxito, que la organización piensa realizar múltiples conciertos. Cada vez que un concierto quede sin entradas disponibles, se deberá reiniciar todas las compras, es decir, nuevamente quedaran disponibles M entradas de galería y N cupos de cancha. Suponga que automáticamente las compras anteriores quedan almacenadas en una base de datos. Escriba snippets de código que consideren este nuevo requerimiento.

Ejercicio

Los profesores Mauricio Hidalgo y Víctor Reyes de Sistemas Operativos, amantes de la buena mesa y los asados, planean organizar una parrillada en la EIT en los meses venideros (ñam ñam). La idea es que a medida que los profesores vayan preparando los platos, estos sean dispuestos en una mesa, para así ser retirados por los participantes al evento. Asuma que solo N platos pueden ser dejados en la mesa, que existen M comensales en el evento y que dada la cantidad de carne disponible, los profesores podrán hacer K platos durante todo el evento ($K > M > N$). Por ultimo, es importante tomar en consideración que todos los asistentes al evento deben al menos comer un plato del asado.

1. Identifique los elementos del problema y abstráigalos a la terminología que hemos empleado en la asignatura
2. Escriba fragmentos de código que permitan implementar una solución al problema.
3. A sido tal el éxito del evento, que se ha sumado también el profesor Martín Gutiérrez. Entre los tres profesores, montan dos parrillas y dos mesas para ir dejando los platos. Cada una de estas mesas tiene capacidad N y cada parrilla podrá hacer $K/2$ platos durante el evento. Escriba fragmentos de código que permitan implementar una solución a este nuevo problema.

Ejercicio

Suponga que 10 threads, y de manera concurrente, comparten acceso a dos stacks LIFO diferentes, Stack-A y Stack-B. Hay dos funciones que los threads utilizan para mover elementos entre los dos stacks:

- AtoB(): esta función saca un elemento del Stack-A y lo añade al Stack-B.
- BtoA(): esta función saca un elemento del Stack-B y lo añade al Stack-A.

Se sabe que inicialmente el Stack-A contiene K elementos y que el Stack-B esta vacío. Las implementaciones parciales de estas funciones son las siguientes:

```
void AtoB() {
    int x = pop(StackA);      // Saca un elemento del stack A
    push(x, StackB);         // Agrega el elemento al stack B
}

void BtoA() {
    int x = pop(StackB);      // Saca un elemento del stack B
    push(x, StackA);         // Agrega el elemento al stack A
}
```

Continuación de ejercicio anterior

El ejercicio consiste en modificar estas funciones, usando semáforos, para asegurar que se cumplan los siguientes requisitos de sincronización:

- La operación `pop()` nunca debe ejecutarse en un stack vacío.
- Solo un thread a la vez debe estar utilizando cada stack. Por ejemplo, si un thread esta ejecutando `pop()` en el Stack-A, entonces ningun otro thread debe estar ejecutando `pop()` o `push()` en el Stack-A.
- Los threads nunca deben bloquearse mutuamente.

Además, debe ser posible, al menos en algunas situaciones, que diferentes threads utilicen los stacks al mismo tiempo. En particular, no se acepta una solución que utilice un solo semáforo para bloquear ambos stacks.

1. Defina todos los semáforos que debe utilizar su solución, en conjunto al valor de inicialización.
2. Agregar operaciones de semáforos a las funciones anteriormente descritas, para que se cumplan los requisitos de sincronización. No utilizar ningun otro mecanismo de sincronización que no sean los semáforos definidos anteriormente. No realizar cambios en el código, aparte de insertar llamadas a operaciones de semáforos.