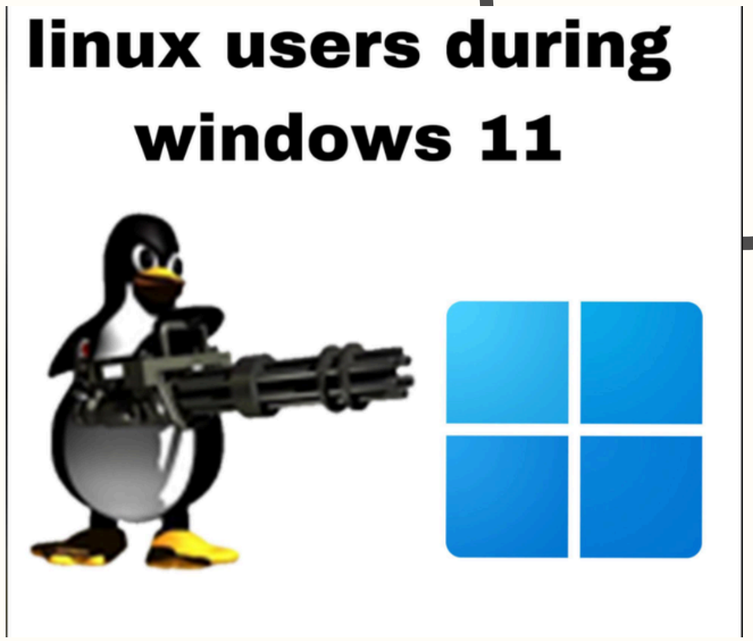
AYUDANTÍA 3: THREADS Y SINCRONIZACIÓN



Profesor : Jonathan Frez

Ayudante : Dante Hortuvia

REGLAS DE LAS TAREAS

- Windows queda estrictamente prohibido en cualquier tipo de clase, ejercicio, tarea y cualquier cosa relacionada al curso

- El uso de IA generativa esta permitido, siempre y cuando no se utilice de manera abusiva

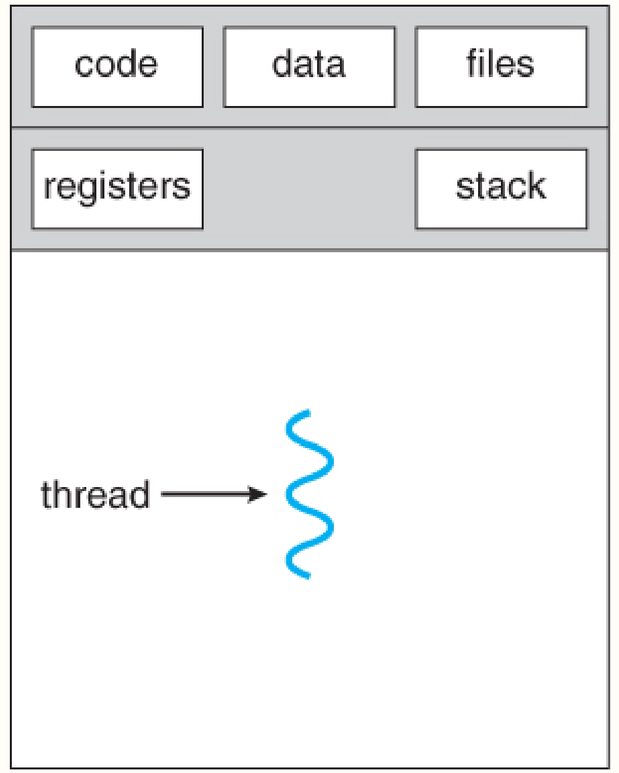
Threads

Los threads son unidades básicas de ejecución dentro de un proceso, diseñadas para realizar tareas de forma concurrente.

Son más ligeros que los procesos, ya que comparten recursos como memoria y archivos dentro del mismo espacio de direcciones.

Cada thread cuenta con un **Thread Control Block (TCB)**, asociado al **PCB** del proceso, para su gestión y control.

Para funcionar correctamente requieren sincronización, evitando conflictos al acceder a los recursos compartidos.



Proceso con un thread

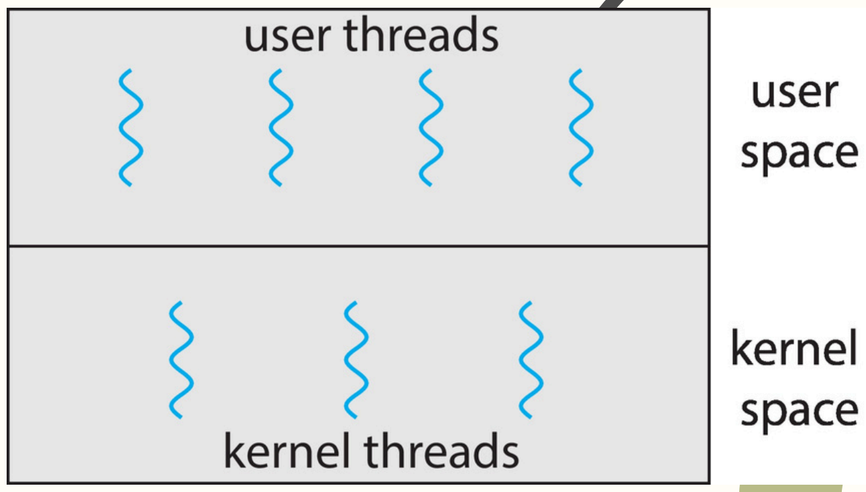
Tipos de Threads

User-Level Threads:

Se crean y gestionan en el espacio de usuario, sin intervención directa del sistema operativo. Son rápidos de crear y ofrecen mayor flexibilidad, adaptándose mejor a necesidades específicas de la aplicación.

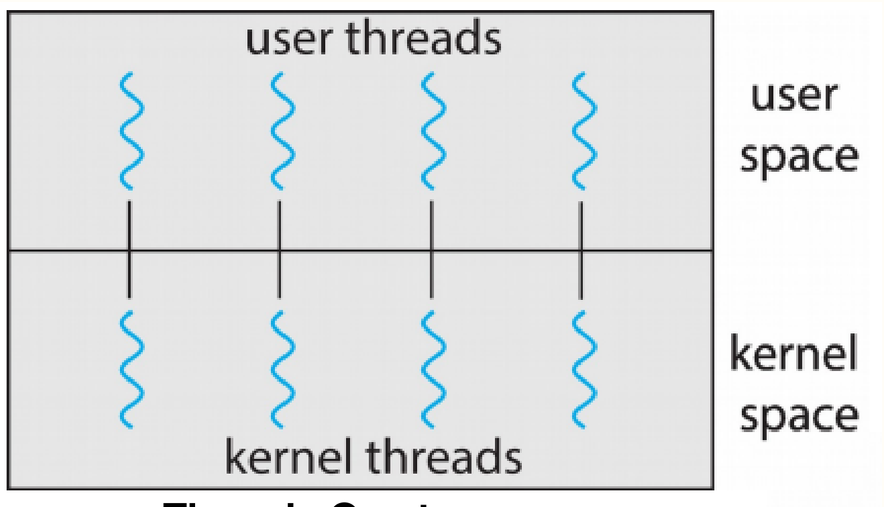
Kernel-Level Threads:

Se crean y gestionan en el espacio del kernel, por lo que dependen del sistema operativo. Su creación consume más recursos y tiempo, pero permiten un mayor control y eficiencia al aprovechar directamente las funcionalidades del kernel.



User Threads y Kernel Threads

MultiThreads



Threads One to one

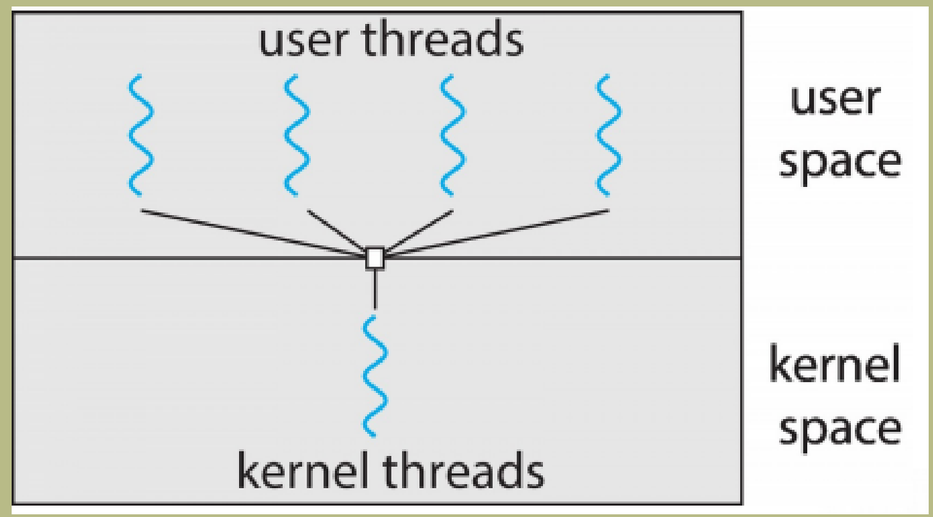
One to One:

En este modelo, cada user-level thread se asigna a un kernel-level thread. Permite paralelismo real, ya que si un thread ejecuta una llamada bloqueante, los demás pueden seguir trabajando. Su desventaja es que un número elevado de kernel threads puede sobrecargar al sistema operativo y afectar el rendimiento.

Many to One:

Varios user-level threads se asocian a un único kernel-level thread. La gestión ocurre en el espacio de usuario, por lo que no existe paralelismo real: solo un thread puede acceder al kernel a la vez. Si un thread realiza una llamada bloqueante, todos los demás quedan detenidos.

.

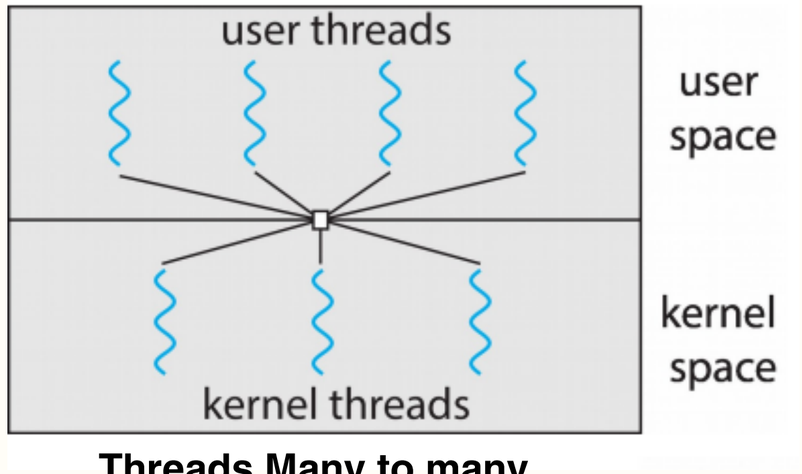


Threads Many to one

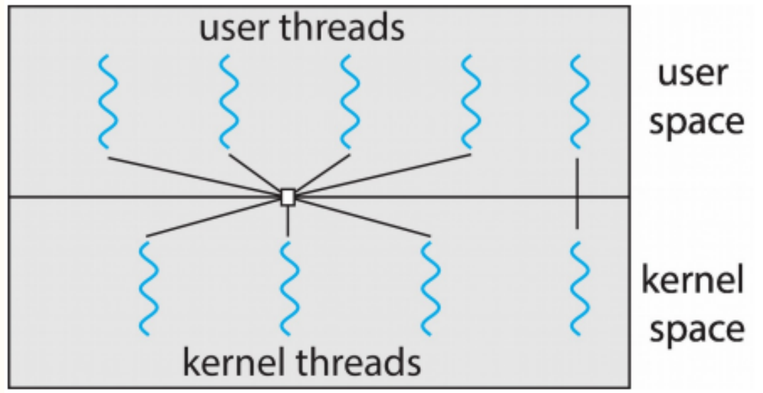
Many to Many:

En este modelo, múltiples user-level threads se asignan a una cantidad igual o menor de kernel-level threads. Ofrece mayor flexibilidad en el uso de recursos y permite aprovechar el paralelismo, evitando las limitaciones de Many to One y el alto costo de One to One.

.



Threads Many to many



Threads Two-Level

Two-Level (Mixto) :

Es una variación del modelo Many to Many que agrega flexibilidad al permitir que ciertos user-level threads se asocien directamente con un kernel-level thread. De esta forma combina las ventajas de ambos enfoques..

.

Sección Crítica

Es una región de la memoria compartida (por ejemplo, variables globales) donde el acceso y la modificación de datos pueden afectar el resultado del programa, generando comportamientos inesperados si no se controla correctamente.

- **Exclusión mutua: Solo un thread o proceso puede estar dentro de la sección crítica a la vez.**
- **Progreso: Si varios threads quieren acceder, al menos uno debe poder entrar en un tiempo limitado.**
- **Ausencia de inanición: Todo thread o proceso que quiera acceder a la sección crítica eventualmente podrá hacerlo.**

Race Conditions

Es una situación que se da cuando dos o más threads compiten por acceder o modificar un recurso compartido, como una variable global. Esta competencia puede llevar a resultados inesperados o inconsistentes, como valores incorrectos o corrupción de datos.

Busy Waiting

Es una técnica donde un thread permanece activo en un bucle verificando repetidamente si ha ocurrido un evento, en lugar de bloquearse. Normalmente se usa mediante instrucciones de comparación o condiciones, pero puede desperdiciar recursos de CPU mientras espera.

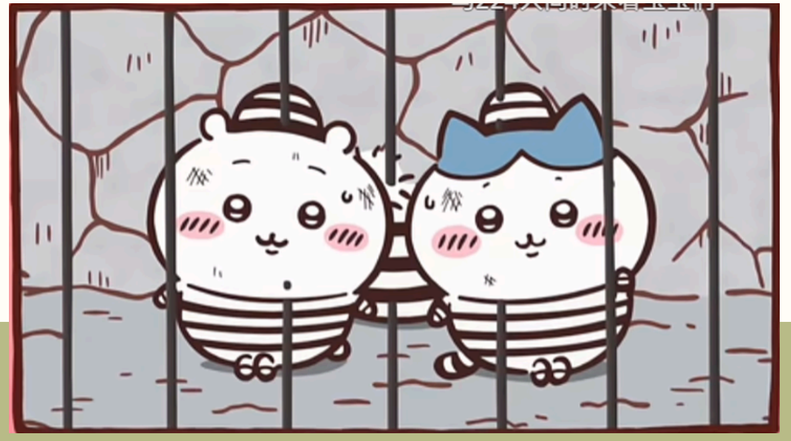
```c
bool evento_ocurrido = false;

void* generar_evento(void* arg) {
    sleep(5);
    evento_ocurrido = true;
    return NULL;
}

void* esperar_evento(void* arg) {
    printf("Esperando evento...\n");
    while (!evento_ocurrido) {
    }
    printf("¡Evento detectado!\n");
    return NULL;
}

int main() {
    pthread_t thread_evento, thread_espera;

    pthread_create(&thread_evento, NULL, generar_evento, NULL);
    pthread_create(&thread_espera, NULL, esperar_evento, NULL);

    pthread_join(thread_evento, NULL);
    pthread_join(thread_espera, NULL);

    return 0;
}
```

Ejemplo Busy Waiting

DeadLock

Un deadlock ocurre cuando dos o más threads quedan bloqueados permanentemente, esperando recursos ocupados entre sí. Se puede producir por varias condiciones:

- Exclusión mutua: Dos o más threads requieren recursos que otros threads poseen, y ninguno está dispuesto a liberarlos.
- Captura y espera: Un thread tiene un recurso y espera por otro que está ocupado por otro thread; este también espera, formando un ciclo.
- Espera circular: Se crea un ciclo donde cada thread espera por un recurso que posee el siguiente thread en el ciclo.
- No expropiación: La ausencia de expropiación permite que un thread retenga un recurso indefinidamente, contribuyendo al deadlock.



Inanición (Starvation)

Ocurre cuando uno o más threads no pueden acceder a los recursos necesarios porque otros threads los monopolizan. A diferencia del deadlock, algunos threads pueden seguir ejecutándose mientras otros quedan esperando indefinidamente.

Causas comunes de la inanición:

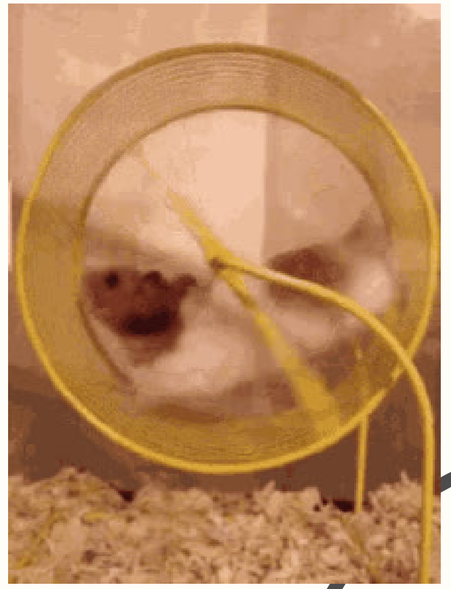
- Prioridades desbalanceadas entre threads.
- Sincronización inadecuada.
- Recursos limitados.
- Captura y espera de recursos por otros threads.

Livelock

Ocurre cuando uno o más threads están activos y respondiendo a cambios en otros threads, pero ninguno progresa realmente en su tarea. A diferencia del deadlock, los threads no están bloqueados; están en constante cambio de estado sin avanzar.

Causas comunes del livelock:

- Manejo inadecuado de la sincronización entre threads.
- Reintentos continuos para acceder a recursos ocupados por otros threads.
- Protocolos de cooperación que generan cambios repetitivos sin progreso real



Mutex

Un mutex es una herramienta que asegura que solo un thread pueda acceder a un recurso compartido a la vez.

Cuando un thread quiere usar el recurso, debe adquirir el mutex (acquire). Si otro thread ya lo posee, debe esperar hasta que esté disponible. Una vez terminado, el thread libera el mutex (release), permitiendo que otros threads accedan al recurso de manera segura.
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

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t lock;

void *thread_func(void *arg){
    pthread_mutex_lock(&lock);
    //  SC
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(){
    pthread_t threads[4];
    pthread_mutex_init(&lock, NULL);
    for(int i = 0 ; i < 4 ; i++){
        pthread_create(threads[i], NULL, thread_func, NULL);
    }
    for(int i = 0 ; i < 4 ; i++){
        pthread_join(threads[i], NULL);
    }
    pthread_mutex_destroy(&lock);
}
```

Semáforos

Inventados por Dijkstra, los semáforos son herramientas de sincronización que controlan el acceso a recursos compartidos entre múltiples threads.

Un semáforo funciona como un contador, que indica cuántos threads pueden acceder simultáneamente a un recurso. Se puede incrementar o decrementar según la disponibilidad del recurso, coordinando el acceso de manera segura

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>

sem_t semaforo;

void* thread_func(void* arg){
    sem_wait(&semaforo);
    //  SC
    sem_post(&semaforo);
    return NULL;
}

int main(){
    sem_init(&semaforo, 0, 2);
    pthread_t threads[4];
    for(int i = 0 ; i < 4 ; i++){
        pthread_create(&threads[i], NULL, thread_func, NULL);
    }
    for(int i = 0 ; i < 4 ; i++){
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

Ejercicios

Ejercicio 1

```c
#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 2

```c
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? Explíquelos y proponga una solución

Ejercicio 3

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 4

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 terminóloga 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 Martın 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 5

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.

Contacto

DANTE.HORTUVIA@MAIL.UDP.CL

+56 9 2236 9606

HTTPS://GITHUB.COM/DOSHUERTOS/AYUDANTIAS_SO_2_2025.GIT

(Las respuestas estan en el git)

Gracias