**EJERCICIOS DE FINAL – CONCURRENCIA**

**¿Qué es un thread, cual es su estructura y qué estados posee ?**

Un thread es una secuencia de ejecución atómica que representa una tarea planificable en ejecución.

* Secuencia de ejecución atómica, cada thread ejecuta una secuencia de instrucciones como lo hace un bloque de código en el modelo de programación secuencial.
* Tarea planificable en ejecución: El sistema operativo tiene injerencia sobre el mismo en cualquier momento y puede ejecutarlo, suspenderlo y continuarlo cuando el desee.

En criollo, un thread es una secuencia de instrucciones independiente el cual el SO controla y planifica su ejecución, ejecutándose dentro de un proceso. El sistema operativo provee la ilusión de que cada uno de estos threads se ejecuta en su propio procesador de manera inmediata. Un thread se caracteriza por:

* Thread ID
* Un conjunto de los valores de registros
* Stack propio
* Política y prioridad de ejecución
* Propio errno
* Datos específicos del thread

Cada thread tiene una estructura que representa su estado, esta es llamada Thread Control Block (TCB), se crea una entrada por cada thread. La TCB almacena el estado per-thread de un thread.

Para poder crear múltiples Threads, pararlos y re-arrancarlos, el sistema operativo debe poder guardar el estado actual del bloque de ejecución en la TCB:

* Puntero al stack del thread.
* Una copia de sus registros del procesador.

Además se guarda metadata sobre el thread en esta estructura:

* ID
* Prioridad de Scheduling
* Status

El sistema operativo para permitir esta abstracción también se debe guardar la información de estado que es compartida por varios threads (estado compartido).

* Código.
* Variables globales.
* Heap.

**Diagrama

Descripción generada automáticamente**

Los estados que puede tomar un thread son los siguientes:

**Diagrama

Descripción generada automáticamente**

* **Init**: Un thread se encuentra en este estado cuando se está inicializando el estado per-thread y se está reservando el espacio necesario para las estructuras. Una vez que esto finaliza se lo pasa a estado READY y se lo pone en una lista llamada ready-list en la cual están esperando todos los threads listos para ejecutar.
* **Ready**: Thread está listo para correr pero no está siendo ejecutado en ese instante. La TCB está en la ready-list, en cualquier momento el thread scheduler puede transicionar al estado RUNNING
* **Running**: Thread está siendo ejecutado por el procesador en ese instante y los valores de los registros de la TCB están cargados en el procesador, en ese momento el thread puede pasar a estado READY de dos formas:
  + El scheduler lo pasa de estado mediante el desalojo o preemption del mismo mediante el guardado de los valores de registro y cambiando el thread que se está ejecutando por el próximo en la lista.
  + Voluntariamente un thread puede solicitar abandonar la ejecución mediante la utilización de thread\_yield, por ejemplo.
* **Waiting**: El thread está esperando que algún evento suceda. Un thread no puede pasar de Waiting a Running directamente, por lo tanto estos threads se almacenan en la lista de waiting-list. Una vez que el evento ocurre el scheduler se encarga de pasar el thread de Waiting a Running, moviendo la TCB desde el waiting-list a la ready-list.
* **Finished**: Un thread que se encuentra en este estado jamás podrá volver a ser ejecutado. Existe una lista llamada finished-list en la que se encuentran los TCB que han terminado.

**Describa qué es un thread y use su API para crear un programa que use 5 threads para incrementar una variable compartida por todos en 7 unidades/thread hasta llegar a 1000.**

Un thread es una secuencia de instrucciones atómicas panificable por el SO que pertenecen a un proceso, esté comparte algunos recursos como el heap, el código o variables globales entre otros thread (estado compartido) pero cada thread posee su propio stack, stack pointer y registros de propósito general. Esta información se contextualiza y guarda en una estructura llamada Thread Control Block (estado per-thread).

int counter **=** 0**;**

void **\***incrementTo1000**(**void **\*** args**)** **{**

mutex\_t **\*** lock **=** **(**mutex\_t **\*)** args**;**

pthread\_mutex\_lock**(**lock**);** // Evita que haya condiciones de carrera en la variable

**if** **(**counter **<** 1000**)** **{**

counter **+=** 7**;**

**}**

pthread\_mutex\_unlock**(**lock**);**

**}**

int main**()** **{**

pthread\_t threads**[**5**];**

mutex\_t **\*** lock**;**

pthread\_mutex\_init**(**lock**);**

**for(**int i **=** 0**;** i**<**5**;**i**++)** **{**

pthread\_create**(&**threads**[**i**],** **NULL,** incrementTo1000**,** lock**);**

**}**

**for(**int i **=** 0**;** i**<**5**;**i**++)** **{**

pthread\_join**(&**threads**[**i**]);**

pthread\_destroy**(&**threads**[**i**]);**

**}**

pthread\_mutex\_destroy**(**lock**);**

**return** 0**;**

**}**

**Defina y dé ejemplos: race-condition, heisenbug, dead-lock, interleave.**

* **Race-Condition**: Cuando más de un thread quiere acceder a un recurso compartido, y al no ser determinístico cual llega primero dado que depende del scheduler se dice que es una condición de carrera dado que depende de quien llegue primero al recurso lo puede dejar en un estado.

Ejemplo:

Thread 1: Thread 2:

x = 2\*x x = x+1

Si al principio x = 2 entonces hay dos resultados posibles:

x = 5 o x = 6

* **Heisenbug**: Son bugs indeterministicos, que dependen del intercalado del código de distintos threads, es decir depende del scheduler. Por lo tanto a veces funciona todo bien y a veces los resultados no son los esperados.

**if** **(**beer **==** 0**)** **{** // Si no hay beer

**if** **(**nota **==** 0**)** **{** // Si no hay nota

nota **=** 1**;** // Dejar la nota

HayBeer**++;** // Comprar beer

nota **=** 0**;** // Sacar nota

**}**

**}**

**if** **(**beer **==** 0**)** **{**

**if** **(**beer **==** 0**)** **{**

**if** **(**nota **==** 0**)** **{**

nota **=** 1**;**

beer**++;**

nota **=** 0**;**

**}**

**}**

**if** **(**nota **==** 0**)** **{**

nota **=** 1**;**

beer**++;**

nota **=** 0**;**

**}**

**}**

En ese caso se esperaba que le llegue la nota del thread 2 al primero para que no comprara cerveza, pero al ejecutarse todo el código y recién switchear de threads luego de que el segundo ya comprara cerveza, el primero nunca recibió la nota entonces volverá a comprar cerveza...

* **Deadlock**: Esto sucede cuando entre varios threads uno obtiene el lock y por alguna razón no lo libera entonces el resto de los threads se quedan esperando a que se libere el recurso cosa que nunca sucede...

thread1

lock**(**l2**)**

lock**(**l**)**

unlock**(**l2**)**

// do awesome stuff

unlock**(**l**)**

thread2

lock**(**l**)**

// do awesome stuff

unlock**(**l**)**

thread3

lock**(**l**)**

// do awesome stuff

lock**(**l2**)**

En este caso si el scheduler pone al thread3 primero, este nunca liberará el recurso dejando a los otros threads trabados por siempre. O si entra primero por thread1 y después switchea al thread3 como no se liberó el lock de l2 también va a quedar trabado los 3 threads.

**Interleave**: Es el entrelazado entre códigos de distintos threads, depende de cómo el scheduler ejecute los distintos threads.

**Zona crítica**: Es la porción de código de un thread donde se debe asegurar que se pueda ejecutar atómicamente, como por ejemplo operaciones sobre un recurso compartido.

**Exclusión mutua**: La exclusión mutua es la forma de asegurar la ejecución atómica de esta zona crítica, dado que cuando un thread tiene el candado o lock ningún otro puede tenerlo

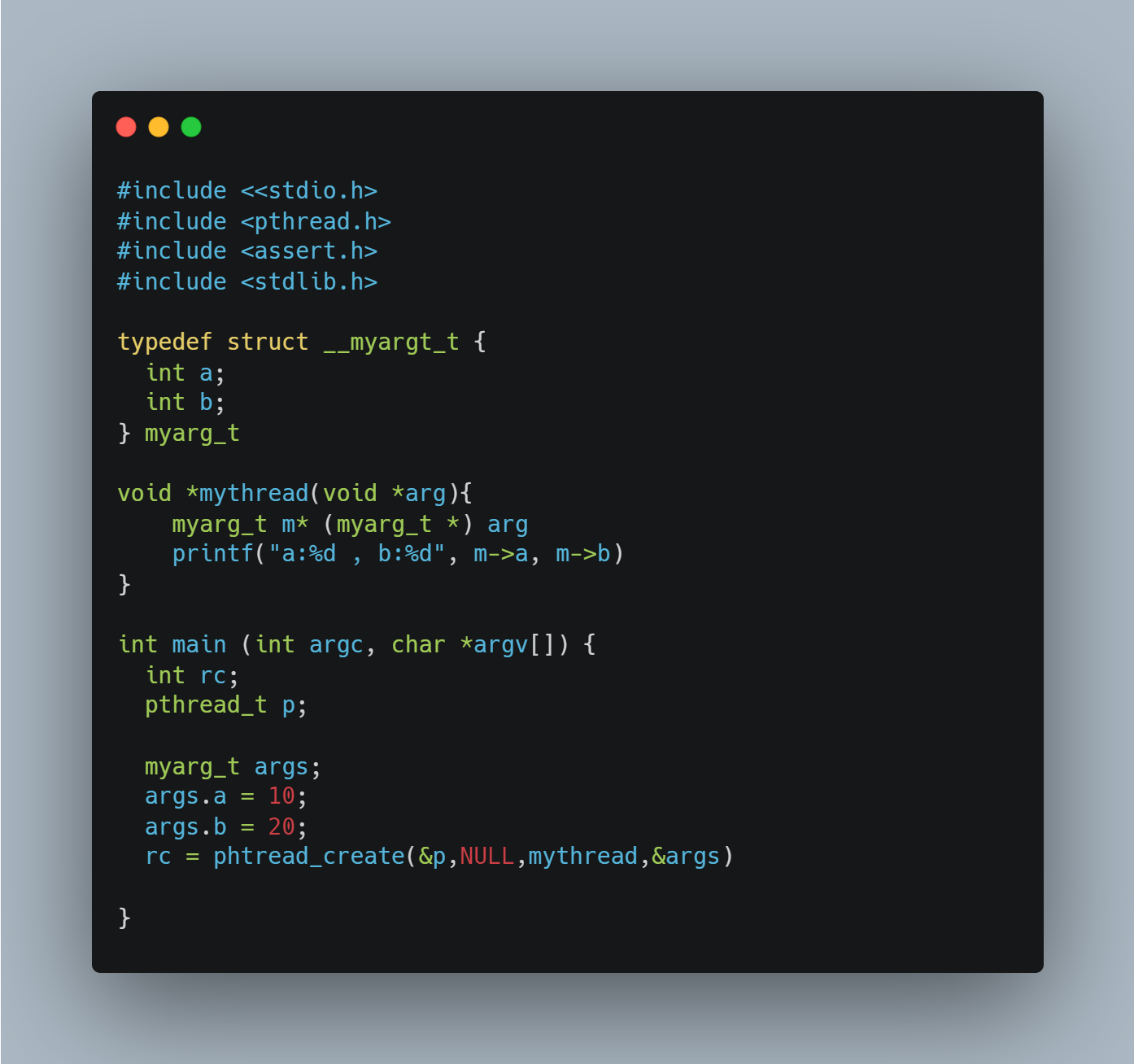
**Defina la abstracción que mejor maneja la \*\*granularidad más fina de cómputo\*\* (thread). Cómo se denomina, sus características principales. Y en un ejemplo que muestre cómo funciona y cuáles son las precauciones que un programador debe tener en cuenta a la hora de utilizar dicha abstracción.**

**Sin el ejemplo es inválido el ejercicio*.***

Un thread es una secuencia de ejecución atómica (atómica quiere decir que inicia y finaliza en un mismo bloque de código, es decir, no se puede particionar) que representa una tarea planificable de ejecución (con planificable quiere decir que el SO puede decidir si ejecutarlo, pausarlo y/o detenerlo).

Las características de un thread son:

1. Thread id
2. Un conjunto de valores de registros
3. Stack propio
4. Política de prioridad de ejecución
5. Propio errno (error number)
6. Datos específicos del thread



**Dado el siguiente tipo de dato, Identificar la sección critica.:**

**struct QNode {**

**int key;**

**struct QNode\* next;**

**};**

**struct Queue {**

**struct QNode \*front, \*rear;**

**};**

**void deQueue(struct Queue\* q){**

**if (q->front == NULL) return;**

**struct QNode\* temp = q->front;**

**q->front = q->front->next;**

**if (q->front == NULL) q->rear = NULL;**

**free(temp);**

**}**

struct QNode {

int key;

struct QNode\* next;

};

struct Queue {

struct QNode \*front, \*rear;

};

void deQueue (struct Queue\* q) {

**\*Inicio de sección crítica\***

if (q->front == NULL)

return;

struct QNode\* temp = q->front->next;

q->front = q->front->next;

if (q->front == NULL)

q->rear = NULL;

free (temp);

**\*Fin de sección crítica\***

}

La sección se vuelve crítica dado que si muchos threads utilizan la cola, estarían constantemente accediendo a la misma porción de memoria del heap, lo que de no hacer un uso correcto de threads provocaría por ejemplo que el nodo se pierda, que se libere memoria que ya se había liberado, etc. Dependiendo de qué thread utilice los métodos pueden darse condiciones de carrera para obtener los nodos.  
  
La solución a esto sería el uso de locks y mutex, donde varios threads pueden hacer uso de esta cola pero uno a la vez, tendrá el acceso a esta porción de código aquel hilo que posea el lock.

**Muestre una implementación de un tipo de dato en el cual una variable que es compartida por varios threads es modificada y accedida de forma sincronizada.**

Un ejemplo de tipo de dato en el que una variable que es compartida por varios threads, es modificada y accedida de forma sincronizada podría ser la del contador con locks:

typedef struct counter\_t{

    int value;

    phthread\_mutex\_t lock;

} counter\_t;

void init (counter\_t \*c){

    c->value = 0;

    Pthread\_mutex\_init(&c->lock, NULL);

}

void increment (counter\_t \*c){

    Pthread\_mutex\_lock(&c->lock);

    c->value++;

    Pthread\_mutex\_unlock(&c->lock);

}

void decrement (counter\_t\* c){

    Pthread\_mutex\_lock(&c->lock);

    c->value--;

    Pthread\_mutex\_unlock(&c->lock);

}

int get (counter\_t \*c){

    Pthread\_mutex\_lock(&c->lock);

    int rc = c->value;

    Pthread\_mutex\_unlock(&c->lock);

    return rc;

}

**Dado la siguiente estructura de datos, implemente la función insertar y la función crear, de forma tal que sea thread safe:**

**int** **List\_Insert**(**list\_t** **\***L, **int** key) {

**node\_t** **\***new **=** malloc(**sizeof**(**node\_t**));

**if** (new **==** NULL) {

perror("malloc");

**return** **-**1; *// fail*

}

new**->**key **=** key;

new**->**next **=** L**->**head;

L**->**head**=** new;

**return** 0; *// success*

}

*// nodo*

**typedef** **struct** **\_\_node\_t** {

**int** key;

**struct** **\_\_node\_t** **\***next;

} **node\_t**;

*// lista*

**typedef** **struct** **\_\_list\_t** {

**node\_t** **\***head;

} **list\_t**;

**Explique detalladamente que es un thread. Cuales son sus componentes. Que metadata se debe almacenar. Cual es su relacion con el paralelismo.**

*// La estructura de la lista deberia tener un mutex:*

**typedef** **struct** **\_\_list\_t** {

**node\_t** **\***head;

**pthread\_mutex\_t** mutex;

}

**int** insert\_node\_safe(**list\_t** **\***list, **int** key) {

**node\_t** **\***new\_node **=** malloc(**sizeof**(**node\_t**));

**if** (new **==** NULL) {

perror("malloc");

**return** **-**1;

}

new\_node**->**key **=** key;

*// Start Lock*

pthread\_mutex\_lock(**&**(lista**->**mutex));

new\_node**->**next **=** list**->**head;

list**->**head **=** new\_node;

pthread\_mutex\_unlock(**&**(lista**->**mutex));

*// End Lock*

**return** 0;

}

**list\_t** **\***create\_safe\_list() {

**list\_t\*** list **=** malloc(**sizeof**(**node\_t**) **+** **sizeof**(**pthread\_mutex\_t**));

pthread\_mutex\_init(**&**(lista**->**mutex), NULL);

**return** list;

}

Un thread es una secuencia de instrucciones ejecutable, atómica y panificable. En Linux, a un thread se lo trata de la misma manera que a un proceso, de hecho, a ambos se los crea con la misma syscall (clone), pero con diferentes parámetros y flags. A diferencia de los procesos, los threads comparten ciertas cosas entre ellos, por ejemplo, el Heap, Code Segment y Variables globales.

Cuando se crea un thread, es necesario guardar cierta metadata como el estado del mismo, el ID y su prioridad.

Los threads son buenos para realizar paralelismo por esto de compartir memoria. Un ejemplo claro es la mutliplicacion de dos matrices. Al compartir la informacion de ambas matrices, se puede acceder desde varios núcleos a la vez para realizar operaciones independientes entre si. Esto podría ahorrar mucho tiempo.