Concurrencia y Paralelismo

Un thread es un hilo de ejecución de un proceso. Los thread comparten:

- Memoria
- File descriptors
- Sockets
- Signals

Los thread comparten la tabla de páginas, por tanto, tienen el mismo heap t zona de datos (comparten variables globales y dinámicas), pero cada thread tiene su propio stack (variables locales). Usaremos la librería <threads.h>.

Creación de threads

```
#include <threads.h>

struct args{
        int i;
};

int thread_function(void *p){
        struct args *args=p; //hacemos que el puntero sea del tipo del
struct de datos que vamos a usar
        args->i++; //relizamos las tareas que tengan que hacerse con los
datos
        return 0;
}

int main(){
        thrd_t t;
        struct args *args=malloc(sizeof(struct args));
        int result;
        args->i=3;

        thrd_create(&t, thread_function, args); //Creamos el thread
        thrd_join(t, &result); //Pausamos la ejecución del código principal
y esperamos a que el thread termine
        free(args);
}
```

Sección crítica y exclusión mutua

La sección crítica es la parte de un código que accede a un recurso compartido entre varios threads. Por ejemplo, si en el ejemplo anterior tuviéramos dos threads que incrementan el valor de i, si los dos leen i=3 a la vez y suman 1, al final del programa tendríamos i=4, cuando nuestro objetivo era i=5. Este problema se conoce como lost update. Para evitar esto se usan los mútex, lo que se conoce como solucionarlo con exclusión mutua. Por ejemplo:

```
#include <threads.h>

struct args{
        mtx_t mutex;
        int i;
};

int thread_function(void *p){
        struct args *args=p;
        mtx_lock(&(args->mutex)); //bloqueamos sección crítica
        args->i++;
        mtx_unlock(&(args->mutex)); //desbloqueamos sección crítica
        return 0;
}

int main(){
        thrd_t t1, t2;
        struct args *args=malloc(sizeof(struct args));
        int result;
        args->i=3;
        mtx_init(&(args->mutex)); //creamos mutex

        thrd_create(&t1, thread_function, args);
        thrd_create(&t2, thread_function, args);
        thrd_join(t1, &result);
        thrd_join(t2, &result);

        mtx_destroy(&(args->mutex)); //destruíumos mutex
        free(args);
}
```

Semáforos

Otra forma de controlar el acceso a la sección crítica son los semáforos. Tienen un valor numérico que les se asigna al crearlos. Al intentar bloquearlo (V o post) se incrementa ese valor. Al intentar desbloquearlo (P o wait) se decrementa ese valor. Al llegar a 0 el proceso se bloquea. Se usan así:

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

#define NUM_THREADS 3

sem_t semaphore;

void* thread_function(void* arg) {
    int thread_id = *(int*)arg;
    printf("Thread %d is waiting\n", thread_id);
    sem_wait(&semaphore);
    printf("Thread %d has acquired the semaphore\n", thread_id);
    printf("Thread %d is releasing the semaphore\n", thread_id);
    sem_post(&semaphore);
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    sem_init(&semaphore, 0, 2);

    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i + 1;
        pthread_create(&threads[i], NULL, thread_function, &thread_ids[i]);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&semaphore);

    return 0;
}
```

Interbloqueo e inanición

El interbloqueo es la situación donde dos o más procesos están esperando por recursos que otro tiene ocupado. Cuando un proceso lleva mucho tiempo esperando el acceso a un recurso compartido se denomina inanición. Existen dos formas simples de prevenir el interbloqueo. Por ejemplo, en esta función que sumará dos posiciones aleatorias de un array:

- Evitando mantener recursos reservados (hold and wait)

```
int suma_protegida(int *v1, int *v2, mtx_t *m1, mtx_t *m2){
        int x;
        while(1){
                mtx_lock(m1);
                if(mtx_trylock(m2)){ //si no puede liberar m2, libera m1
                        mtx_unlock(m1);
                        continue; //vuelve al while
                }
                x=(*v1)+(*v2);
                unlock(m1);
                unlock(m2);
                break;
        }
        return x;
}
```

- Con reserva ordenada

```
int suma_protegida(int *v1, int *v2, mtx_t *m1, mtx_t *m2, int orden1, int
orden2){
        int x;
        if(orden1<orden2){
                mtx_lock(m1);
                mtx_lock(m2);
        } else{
                mtx_lock(m2);
                mtx_lock(m1);
        }
        x=(*v1)+(*v2);
        unlock(m1);
        unlock(m2);
        return x;
}
```

Productores y consumidores

Este problema se da cuando hay un buffer compartido entre procesos que insertan elementos y problemas que los eliminan. Si el buffer se llena, los producores deben esperar a que los consumidores consuman elementos. Si el buffer está vacío, los consumidores deben esperar a que los productores produzcan nuevos elementos.

La solución más simple a priori es comprobando continuamente con un contador la cantidad de elementos, pero esto produce un alto consumo de CPU. Por eso las mejores soluciones son con condiciones o con semáforos.

- Con condiciones:

```c
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>

#define BUFFER_SIZE 5
#define NUM_PRODUCERS 2
#define NUM_CONSUMERS 2

int buffer[BUFFER_SIZE];
int buffer_index = 0;

mtx_t mutex;
cnd_t buffer_not_full;
cnd_t buffer_not_empty;

void produce(int value) {
    buffer[buffer_index++] = value;
}

int consume() {
    return buffer[--buffer_index];
}

void* producer(void* arg) {
    int producer_id = *(int*)arg;

    for (int i = 0; i < BUFFER_SIZE; i++) {
        mtx_lock(&mutex);

        while (buffer_index == BUFFER_SIZE) {
            cnd_wait(&buffer_not_full, &mutex);
        }

        int value = rand() % 100;
        produce(value);
        printf("Producer %d produced: %d\n", producer_id, value);

        cnd_signal(&buffer_not_empty);
        mtx_unlock(&mutex);
    }
```

```c
    return NULL;
}

void* consumer(void* arg) {
    int consumer_id = *(int*)arg;

    for (int i = 0; i < BUFFER_SIZE; i++) {
        mtx_lock(&mutex);

        while (buffer_index == 0) {
            cnd_wait(&buffer_not_empty, &mutex);
        }

        int value = consume();
        printf("Consumer %d consumed: %d\n", consumer_id, value);

        cnd_signal(&buffer_not_full);
        mtx_unlock(&mutex);
    }

    return NULL;
}

int main() {
    thrd_t producers[NUM_PRODUCERS];
    thrd_t consumers[NUM_CONSUMERS];

    mtx_init(&mutex, mtx_plain);
    cnd_init(&buffer_not_full);
    cnd_init(&buffer_not_empty);

    int producer_ids[NUM_PRODUCERS];
    int consumer_ids[NUM_CONSUMERS];

    for (int i = 0; i < NUM_PRODUCERS; i++) {
        producer_ids[i] = i + 1;
        thrd_create(&producers[i], producer, &producer_ids[i]);
    }

    for (int i = 0; i < NUM_CONSUMERS; i++) {
        consumer_ids[i] = i + 1;
        thrd_create(&consumers[i], consumer, &consumer_ids[i]);
    }

    for (int i = 0; i < NUM_PRODUCERS; i++) {
```

```
        thrd_join(producers[i], NULL);
    }

    for (int i = 0; i < NUM_CONSUMERS; i++) {
        thrd_join(consumers[i], NULL);
    }

    mtx_destroy(&mutex);
    cnd_destroy(&buffer_not_full);
    cnd_destroy(&buffer_not_empty);

    return 0;
}
```

- Con semáforos:

```
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>
#include <semaphore.h>

#define BUFFER_SIZE 5
#define NUM_PRODUCERS 2
#define NUM_CONSUMERS 2

int buffer[BUFFER_SIZE];
int buffer_index = 0;

sem_t mutex;
sem_t buffer_not_full;
sem_t buffer_not_empty;

void produce(int value) {
    buffer[buffer_index++] = value;
}

int consume() {
    return buffer[--buffer_index];
}

void* producer(void* arg) {
    int producer_id = *(int*)arg;

    for (int i = 0; i < BUFFER_SIZE; i++) {
        sem_wait(&buffer_not_full);
        sem_wait(&mutex);
```

```
        int value = rand() % 100;
        produce(value);
        printf("Producer %d produced: %d\n", producer_id, value);

        sem_post(&mutex);
        sem_post(&buffer_not_empty);
    }

    return NULL;
}

void* consumer(void* arg) {
    int consumer_id = *(int*)arg;

    for (int i = 0; i < BUFFER_SIZE; i++) {
        sem_wait(&buffer_not_empty);
        sem_wait(&mutex);

        int value = consume();
        printf("Consumer %d consumed: %d\n", consumer_id, value);

        sem_post(&mutex);
        sem_post(&buffer_not_full);
    }

    return NULL;
}

int main() {
    thrd_t producers[NUM_PRODUCERS];
    thrd_t consumers[NUM_CONSUMERS];

    sem_init(&mutex, 0, 1);
    sem_init(&buffer_not_full, 0, BUFFER_SIZE);
    sem_init(&buffer_not_empty, 0, 0);

    int producer_ids[NUM_PRODUCERS];
    int consumer_ids[NUM_CONSUMERS];

    for (int i = 0; i < NUM_PRODUCERS; i++) {
        producer_ids[i] = i + 1;
        thrd_create(&producers[i], producer, &producer_ids[i]);
    }

    for (int i = 0; i < NUM_CONSUMERS; i++) {
```

```
        consumer_ids[i] = i + 1;
        thrd_create(&consumers[i], consumer, &consumer_ids[i]);
    }

    for (int i = 0; i < NUM_PRODUCERS; i++) {
        thrd_join(producers[i], NULL);
    }

    for (int i = 0; i < NUM_CONSUMERS; i++) {
        thrd_join(consumers[i], NULL);
    }

    sem_destroy(&mutex);
    sem_destroy(&buffer_not_full);
    sem_destroy(&buffer_not_empty);

    return 0;
}
```

Lectores y escritores

Este problema se da cuando hay procesos que leen y procesos que escriben. Por tanto, todos los procesos podrán leer la zona compartida, pero solo un escritor puede escribir a la vez. Existen dos formas de resolver este problema:

- Si se desean hacer muchas consultas se les da la prioridad a los lectores

```
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>

#define NUM_READERS 5
#define NUM_WRITERS 2

int shared_data = 0;
int readers_count = 0;
int writers_waiting = 0;
mtx_t mutex;
mtx_t resource_mutex;
cnd_t readers_cond;
cnd_t writers_cond;

void* reader(void* arg) {
    int reader_id = *(int*)arg;
```

```
    mtx_lock(&mutex);
    while (writers_waiting > 0) {
        cnd_wait(&readers_cond, &mutex);
    }
    readers_count++;
    mtx_unlock(&mutex);

    // Leer datos compartidos
    printf("Reader %d reads data: %d\n", reader_id, shared_data);

    mtx_lock(&mutex);
    readers_count--;

    if (readers_count == 0) {
        cnd_signal(&writers_cond);
    }

    mtx_unlock(&mutex);

    return NULL;
}

void* writer(void* arg) {
    int writer_id = *(int*)arg;

    mtx_lock(&mutex);
    writers_waiting++;

    while (readers_count > 0) {
        cnd_wait(&writers_cond, &mutex);
    }

    writers_waiting--;
    mtx_unlock(&mutex);

    // Escribir en los datos compartidos
    shared_data = writer_id;
    printf("Writer %d writes data: %d\n", writer_id, shared_data);

    cnd_signal(&readers_cond);
    cnd_signal(&writers_cond);

    return NULL;
}

int main() {
```

```
    thrd_t readers[NUM_READERS];
    thrd_t writers[NUM_WRITERS];

    mtx_init(&mutex, mtx_plain);
    mtx_init(&resource_mutex, mtx_plain);
    cnd_init(&readers_cond);
    cnd_init(&writers_cond);

    int reader_ids[NUM_READERS];
    int writer_ids[NUM_WRITERS];

    for (int i = 0; i < NUM_READERS; i++) {
        reader_ids[i] = i + 1;
        thrd_create(&readers[i], reader, &reader_ids[i]);
    }

    for (int i = 0; i < NUM_WRITERS; i++) {
        writer_ids[i] = i + 1;
        thrd_create(&writers[i], writer, &writer_ids[i]);
    }

    for (int i = 0; i < NUM_READERS; i++) {
        thrd_join(readers[i], NULL);
    }

    for (int i = 0; i < NUM_WRITERS; i++) {
        thrd_join(writers[i], NULL);
    }

    mtx_destroy(&mutex);
    mtx_destroy(&resource_mutex);
    cnd_destroy(&readers_cond);
    cnd_destroy(&writers_cond);

    return 0;
}
```

- Si se desea mantener información actualizada se les da prioridad a los escritores

```
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>

#define NUM_READERS 5
#define NUM_WRITERS 2
```

```c
int shared_data = 0;
int readers_count = 0;
int writers_count = 0;
int writers_waiting = 0;
mtx_t mutex;
mtx_t resource_mutex;
cnd_t readers_cond;
cnd_t writers_cond;

void* reader(void* arg) {
    int reader_id = *(int*)arg;

    mtx_lock(&mutex);
    while (writers_count > 0 || writers_waiting > 0) {
        cnd_wait(&readers_cond, &mutex);
    }
    readers_count++;
    mtx_unlock(&mutex);

    // Leer datos compartidos
    printf("Reader %d reads data: %d\n", reader_id, shared_data);

    mtx_lock(&mutex);
    readers_count--;

    if (readers_count == 0 && writers_waiting > 0) {
        cnd_signal(&writers_cond);
    }

    mtx_unlock(&mutex);

    return NULL;
}

void* writer(void* arg) {
    int writer_id = *(int*)arg;

    mtx_lock(&mutex);
    writers_waiting++;

    while (readers_count > 0 || writers_count > 0) {
        cnd_wait(&writers_cond, &mutex);
    }

    writers_waiting--;
    writers_count++;
```

```
    mtx_unlock(&mutex);

    // Escribir en los datos compartidos
    shared_data = writer_id;
    printf("Writer %d writes data: %d\n", writer_id, shared_data);

    mtx_lock(&mutex);
    writers_count--;

    if (writers_waiting > 0) {
        cnd_signal(&writers_cond);
    } else {
        cnd_broadcast(&readers_cond);
    }

    mtx_unlock(&mutex);

    return NULL;
}

int main() {
    thrd_t readers[NUM_READERS];
    thrd_t writers[NUM_WRITERS];

    mtx_init(&mutex, mtx_plain);
    mtx_init(&resource_mutex, mtx_plain);
    cnd_init(&readers_cond);
    cnd_init(&writers_cond);

    int reader_ids[NUM_READERS];
    int writer_ids[NUM_WRITERS];

    for (int i = 0; i < NUM_READERS; i++) {
        reader_ids[i] = i + 1;
        thrd_create(&readers[i], reader, &reader_ids[i]);
    }

    for (int i = 0; i < NUM_WRITERS; i++) {
        writer_ids[i] = i + 1;
        thrd_create(&writers[i], writer, &writer_ids[i]);
    }

    for (int i = 0; i < NUM_READERS; i++) {
        thrd_join(readers[i], NULL);
    }
```

```
    for (int i = 0; i < NUM_WRITERS; i++) {
        thrd_join(writers[i], NULL);
    }

    mtx_destroy(&mutex);
    mtx_destroy(&resource_mutex);
    cnd_destroy(&readers_cond);
    cnd_destroy(&writers_cond);

    return 0;
}
```