Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencias de la Computación
R-322 — Sistemas Operativos I

Prctica 3
Sincronización

Nota: El busy-waiting es **inaceptable** para todo ejercicio de esta práctica.

Ej. 1 (Pan y Manteca). Supongamos que tenemos un proceso A que quiere hacerse un pan con manteca. La presencia del pan en la mesa se simboliza con una variable booleana `P`, e ídem para la manteca con `M`. Ambas variables están protegidas por un mutex `lk`. El proceso A, al haber observado que no había ninguna de las dos cosas, se puso a esperar en una variable de condición `cv` a que alguien lo despierte cuando ambas cosas existan sobre la mesa. Si se despierta a A con un solo ingrediente (un estado inaceptable), A destruye la casa en un ataque de furia.

Ahora, el proceso B, habiendo obtenido ambos ingredientes, intenta despertar a A vía la siguiente secuencia:

```
pthread_mutex_lock(&lk);
P = true;
pthread_cond_signal(&cv);
M = true;
pthread_mutex_unlock(&lk);
```

¿Es esto correcto? ¿Qué problema puede haber?

Ej. 2 (Filósofos Comensales, Dijkstra). Cinco filósofos se sientan alrededor de una mesa redonda y viven la buena vida alternando entre comer y pensar. Cada filósofo tiene su plato de pasta, un tipo particular de fideos que requieren dos tenedores para ser comidos. Entre cada par de filósofos consecutivos hay un tenedor, y cada filósofo sólo puede alcanzar los que están a su izquierda y derecha. Para comer, los filósofos acordaron primero tomar el tenedor a su derecha y luego el de a su izquierda. Si cualquier filósofo toma un tenedor y el otro está ocupado, se quedará esperando, con un tenedor en la mano, hasta que pueda tomar el otro tenedor, para luego empezar a comer. Una vez que un filósofo terminar de comer deja los tenedores sobre la mesa y piensa hasta que le vuelve a dar hambre, cuando repite el procedimiento.

Una implementación de esta situación con `pthreads` es como sigue:

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

#define N_FILOSOFOS 5
#define ESPERA 5000000

pthread_mutex_t tenedor[N_FILOSOFOS];
```

```
pthread_mutex_t * izq(int i) { return &tenedor[i]; }
pthread_mutex_t * der(int i) { return &tenedor[(i+1) % N_FILOSOFOS]; }

void pensar(int i)
{
        printf("Filosofo %d pensando...\n", i);
        usleep(random() % ESPERA);
}

void comer(int i)
{
        printf("Filosofo %d comiendo...\n", i);
        usleep(random() % ESPERA);
}

void tomar_tenedores(int i)
{
        pthread_mutex_lock(der(i));
        pthread_mutex_lock(izq(i));
}

void dejar_tenedores(int i)
{
        pthread_mutex_unlock(der(i));
        pthread_mutex_unlock(izq(i));
}

void * filosofo(void *arg)
{
        int i = arg - (void*)0;
        while (1) {
                tomar_tenedores(i);
                comer(i);
                dejar_tenedores(i);
                pensar(i);
        }
}

int main()
{
        pthread_t filo[N_FILOSOFOS];
        int i;

        for (i = 0; i < N_FILOSOFOS; i++)
                pthread_mutex_init(&tenedor[i], NULL);

        for (i = 0; i < N_FILOSOFOS; i++)
                pthread_create(&filo[i], NULL, filosofo, i + (void*)0);

        pthread_join(filo[0], NULL);
        return 0;
}
```

a) Este programa puede terminar en deadlock. Explique cómo.

b) Cansados de no comer los filósofos deciden pensar una solución a su problema. Uno razona que esto no sucedería si alguno de ellos fuese zurdo y tome primero el tenedor de su izquierda. Implemente

esta solución y explique por qué funciona.

c) Otro filósofo piensa que tampoco tendrían el problema si todos fuesen diestros pero sólo intentasen comer a lo sumo $N - 1$ de ellos a la vez.

Implemente esta solución y explique por qué funciona. Para ello va a necesitar un semáforo de Dijkstra. Puede utilizar los *POSIX Semaphores*. En la cabecera `semaphore.h` puede encontrar los prototipos de las funciones necesarias:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

d) Un filósofo, ya sufriendo secuelas del hambre, sugiere que los comensales suelten su tenedor derecho si encuentran que el izquierdo ya está tomado, posiblemente esperando un tiempo antes de reintentar tomarlo. Implemente esta variante e investíguela. ¿Funciona? ¿Es eficiente? Cuantifique su respuesta.

Ej. 3 (Productor-Consumidor, Dijkstra). En un sistema, hay M procesos llamados "productores" que generan valores (digamos, de tipo `int`) y N procesos "consumidores" que los leen y realizan alguna computación con ellos. Se debe garantizar que:

- cada valor producido sea consumido, es decir, que no se pierda
- cada valor producido no sea tomado por más de un consumidor

Los procesos se comunican mediante un buffer compartido, insertando y tomando elementos del mismo. En este ejemplo (ver `prod-cons.c`), los productores alocan un entero con `malloc()` y los consumidores los liberan con `free()`. Perder elementos producidos implica una fuga de memoria, y consumirlos dos veces implica un doble-free.

a) Implemente una solución usando semáforos para llevar las cantidades de elementos en el buffer, y la cantidad de elementos libres.

b) Implemente una solución usando variables de condición.

Ej. 4 (Lectores y Escritores, Parnas). El problema de los lectores y escritores consiste en M hilos lectores y N escritores tratando de acceder a un arreglo en memoria compartida con las siguientes restricciones:

- No puede haber un lector accediendo al arreglo al mismo tiempo que un escritor.
- Varios lectores pueden acceder al arreglo simultáneamente.
- Sólo puede haber un escritor a la vez.

El siguiente código es una implementación del problema utilizando POSIX Threads.

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

#define M 5
#define N 5
#define ARRLEN 10240
```

```c
int arr[ARRLEN];

void * escritor(void *arg)
{
        int i;
        int num = arg - (void*)0;
        while (1) {
                sleep(random() % 3);
                printf("Escritor %d escribiendo\n", num);
                for (i = 0; i < ARRLEN; i++)
                        arr[i] = num;
        }
        return NULL;
}

void * lector(void *arg)
{
        int v, i;
        int num = arg - (void*)0;
        while (1) {
                sleep(random() % 3);
                v = arr[0];
                for (i = 1; i < ARRLEN; i++) {
                        if (arr[i] != v)
                                break;
                }
                if (i < ARRLEN)
                        printf("Lector %d, error de lectura\n", num);
                else
                        printf("Lector %d, dato %d\n", num, v);
        }
        return NULL;
}

int main()
{
        pthread_t lectores[M], escritores[N];
        int i;

        for (i = 0; i < M; i++)
                pthread_create(&lectores[i], NULL, lector, i + (void*)0);

        for (i = 0; i < N; i++)
                pthread_create(&escritores[i], NULL, escritor, i + (void*)0);

        pthread_join(lectores[0], NULL); /* Espera para siempre */
        return 0;
}
```

La solución a este problema es esencialmente un *read-write lock*. Los siguientes apartados piden implementar distintas variantes. Las mismas deben ser **reusables**: el read-write lock no debe estar atado de ninguna forma a este problema en particular (e.g. no debe saber las cantidades de escritores y lectores, ni asumir que un proceso es siempre lector o siempre escritor). Organice su respuesta separando **claramente** la *implementación* del lock de su *uso* en el problema (idealmente en archivos separados). La interfaz debería ser similar a la ofrecida por los mutex de `pthread`.

a) Implemente una solución y explíquela. Para este apartado, siempre que el lock esté tomado por un lector y aparezca un segundo lector, el segundo debe poder entrar inmediatamente (*read-preferring*).

b) Si hay varios lectores que continuamente intentan leer el arreglo, esto puede llevar a starvation de los escritores (explique por qué). Modifique su solución para que, cuando un escritor desee entrar a su RC, ningún lector pueda entrar hasta que ese escritor termine (*write-preferring*).

c) Ahora, podemos tener el problema inverso: si muchos escritores intentan entrar, pueden dejar fuera a los lectores. Implemente una variante *justa* que respete el orden en el que los threads pidieron entrar a la RC.

Ej. 5 (Problema del Barbero, Dijkstra). Una barbería tiene una sala de espera con N sillas y un barbero. Si no hay clientes para atender, el barbero se pone a dormir. Si un cliente llega y todas las sillas están ocupadas, se va. Si el barbero está ocupado pero hay sillas disponibles, se sienta en una y espera a ser atendido. Si el barbero está dormido, despierta al barbero. El cliente y el barbero deben ejecutar concurrentemente las funciones `me_cortan()` y `cortando()` y al terminar los dos ejecutar concurrentemente `pagando()` y `me_pagan()`.

Escriba un programa que coordine el comportamiento del barbero y los clientes y explíquelo.

Ej. 6 (Problema de los Fumadores, Patil). Tres procesos tratan de fumar cada vez que pueden. Para hacerlo necesitan tres ingredientes: tabaco, papel y fósforos. Cada uno tiene una cantidad ilimitada de uno de estos ingredientes. Esto es, un fumador tiene tabaco, otro tiene papel y el último tiene fósforos.

Los fumadores no se prestan los ingredientes entre ellos, pero hay un cuarto proceso, el agente, con cantidad ilimitada de todos los ingredientes, que repetidamente pone a disposición de los fumadores dos de los tres ingredientes, eligiéndolos al azar. Cada vez que esto pasa, el fumador que tiene el ingrediente restante debería proceder a hacerse un cigarrillo y fumar. Al terminar de fumar (y no antes) el fumador avisa al agente que terminó, para que el mismo pueda seguir atendiendo.

A continuación se puede ver una implementación ingenua con `pthreads`:

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

sem_t tabaco, papel, fosforos, otra_vez;

void agente()
{
        while (1) {
                sem_wait(&otra_vez);
                int caso = random() % 3;
                if (caso != 0) sem_post(&fosforos);
                if (caso != 1) sem_post(&papel);
                if (caso != 2) sem_post(&tabaco);
        }
}

void fumar(int fumador)
{
        printf("Fumador %d: Puf! Puf! Puf!\n", fumador);
        sleep(1);
```

```
}

void * fumador1(void *arg)
{
        while (1) {
                sem_wait(&tabaco);
                sem_wait(&papel);
                fumar(1);
                sem_post(&otra_vez);
        }
}

void * fumador2(void *arg)
{
        while (1) {
                sem_wait(&fosforos);
                sem_wait(&tabaco);
                fumar(2);
                sem_post(&otra_vez);
        }
}

void * fumador3(void *arg)
{
        while (1) {
                sem_wait(&papel);
                sem_wait(&fosforos);
                fumar(3);
                sem_post(&otra_vez);
        }
}

int main()
{
        pthread_t s1, s2, s3;

        sem_init(&tabaco,   0, 0);
        sem_init(&papel,    0, 0);
        sem_init(&fosforos, 0, 0);
        sem_init(&otra_vez, 0, 1);

        pthread_create(&s1, NULL, fumador1, NULL);
        pthread_create(&s2, NULL, fumador2, NULL);
        pthread_create(&s3, NULL, fumador3, NULL);

        agente();

        return 0;
}
```

El problema consiste en garantizar que el sistema progresa normalmente, es decir que los fumadores toman los ingredientes cuando les corresponde, fuman, y luego notifican al agente que se puede continuar. Los fumadores deben *sincronizar entre ellos*; no debe modificarse al agente en lo más minímo. Sí pueden crearse nuevos semáforos, mutexes, y threads.

a) ¿Cómo puede ocurrir un deadlock en esta implementación ingenua?

b) ¿Pueden *ordenarse* los recursos para evitar el problema?

c) Implemente una solución (recuerde: sin modificar al agente) y explíquela.

d) Piense soluciones alternativas si fuera aceptable modificar al agente.

e) Extra: ¿cómo se relaciona este ejercicio al servidor de turnos de la práctica anterior?

Nota histórica: Patil introdujo este problema en 1971 problema como una crítica a la universalidad de los semáforos. Dijkstra planteaba que sus semáforos podían tomarse como la única primitiva necesaria para escribir programas concurrentes, construyendo todo el resto (mutex, CV, etc) a partir los mismos. Para este problema de los fumadores, Patil demuestra que **no** puede resolverse usando semáforos, mientras se den dos condiciones:

1. No puede modificarse al agente. Esta restricción proviene de pensar que el agente puede ser el Sistema Operativo, que notifica la disponibilidad de ciertos recursos. No es realista modificar al SO para cada problema que aparece en el espacio de usuario.
2. Que no puedan usarse sentencias condicionales. Esta última restricción no es sensata, y la vamos a ignorar, permitiendo resolver el problema. La justificación original era que los semáforos se introdujeron explícitamente para evitar el testeo repetido de una condición (i.e. busy-waiting).

Patil concluye que los semáforos son insuficientes, y propone una nueva operación $P(s_1, \ldots, s_n)$ que bloquea haciendo P de n semáforos a la vez, y los toma de manera atómica cuando sea posible.

Algunos años después, en 1975, David Parnas presenta una solución cumpliendo ambas condiciones, exactamente lo que Patil había demostrado imposible. ¿El truco? Parnas usó un array de semáforos, algo que la prueba de Patil excluía de manera implícita. A la vez, Parnas criticaba la restricción (2), argumentando que no es realista prohibir condicionales, por más que sí sea realista prohibir el busy-waiting.

Extra: ¿Cómo resolvería el problema con el P n-ario? ¿Cómo resolvería el problema usando un array de semáforos?

Ej. 7 (Cola Lock-Free, Lamport). El siguiente fragmento implementa una cola concurrente entre dos procesos. La variable `s` mantiene la cantidad de elementos escritos y `r` los leídos. El valor `K` está para que estas variables no crezcan demasiado (`s` y `r` siempre están entre 0 y `K-1`).

```
#define B 2
#define K (2*B)

volatile int s, r, buf[B];

static inline int diff() { return (K + s - r) % K; }
void * prod(void *_arg) {
        int cur = 0;
        while (1) {
                while (diff() == B)
                        ;
                buf[s % B] = cur++;
                s = (s+1) % K;
        }
}
void * cons(void *_arg) {
        int cur;
        while (1) {
```

```
                while (diff() == 0)
                        ;
                cur = buf[r % B];
                r = (r+1) % K;

                printf("Leí %d\n", cur);
        }
}
```

Explique por qué funciona sin usar mutexes ni ninguna primitiva de sincronización. ¿Puede generalizar a n procesos?

Ej. 8 (Barreras). Una barrera para n threads tiene una única operación `barrier_wait()` que causa que los threads se pausen hasta que *todos* lleguen a la barrera. Son usadas, generalmente, para asegurar que las iteraciones de varios bucles en paralelo proceden a un mismo paso. Implemente una librería de barreras, exponiendo las funciones:

```
void barrier_init(struct barrier *b, int n);
void barrier_wait(struct barrier *b);
```

Úselas para corregir el siguiente fragmento. La función `calor()` simula la transferencia de calor en un material, haciendo que cada elemento del array se "acerque" a sus vecinos, dejando el resultado de la transformación en un nuevo array.

```
float arr1[N], arr2[N];
void calor(float *arr, int lo, int hi, float *arr2) {
        int i;
        for (i = lo; i < hi; i++) {
                int m = arr[i];
                int l = i > 0   ? arr[i-1] : m;
                int r = i < N-1 ? arr[i+1] : m;
                arr2[i] = m + (l - m)/1000.0 + (r - m)/1000.0;
        }
}

/* Dado un array de [n], devuelve el punto de corte [i] de los [m] totales. */
static inline int cut(int n, int i, int m) {
        return i * (n/m) + min(i, n%m);
}

void * thr(void *arg) {
        int id = arg - (void*)0; /* 0 <= id < W */
        int lo = cut(N, id, W), hi = cut(N, id+1, W);
        int i;
        for (i = 0; i < ITERS; i++) {
                calor(arr1, lo, hi, arr2);
                calor(arr2, lo, hi, arr1);
        }
}
```

Ej. 9 (extra). Implemente barreras *sin* usar ninguna primitiva salvo un spinlock. Compare la performance al aplicarla al ejercicio anterior.

Ej. 10 (Canales Síncronos). Un *canal* es una primitiva que permite el envío de un valor entre threads. Que sea *síncrono* implica que no sólo el lector espera al escritor (obviamente) sino que el escritor no avanza hasta que haya aparecido un lector. Implemente canales síncronos con la siguiente interfaz:

```
struct channel { ... };
void channel_init(struct channel *c);
void chan_write(struct channel *c, int v);
int chan_read(struct channel *c);
```

Use su implementación para implementar una solución al problema de los (múltiples) productores y consumidores. ¿Qué ventajas y desventajas tiene?

Ej. 11. Implemente una librería de semáforos usando variables de condición.

Ej. 12. Implemente una librería de variables de condición usando semáforos.