

1.- Estudie el siguiente código que trata de resolver, únicamente por software y sin mediación del sistema operativo, el problema de la exclusión mutua entre dos hilos:

```
turno = 0

Hilo 0
while(TRUE){
    //Espera ocupada
    while(turno!=0);
    sección_crítica_H1();
    turno = 1;
    otro_código_H1()
}

Hilo 1
while(TRUE){
    //Espera ocupada
    while(turno!=1);
    sección_crítica_H2();
    turno = 0;
    otro_código_H2()
}
```

¿Resuelve correctamente el problema de la sección crítica garantizando exclusión, progreso y espera limitada?. Justifique su respuesta

2.- Escriba un programa que cree tres hilos que se comunicarán entre ellos. El hilo 1 genera los 1000 primeros números pares y el hilo 2 los 1000 números impares. El hilo 3 irá leyendo esos números e imprimiéndolos en pantalla. Se debe garantizar que los números escritos por pantalla estén en orden: 1,2,3,4,5... Implementar dicha sincronización mediante:

- a) Cerrojos y variables condicionales.
- b) Semáforos.

H1: 1000 primeros nº pares.
H2: 1000 impares.
H3: leer los nº e imprimirlos.

a)

```
mutex_t mutex;
var_cond vcPar, vcImpar, vcConsumir;

#define IMPARES 1
#define PARES 2
#define CONSUMIR -1

int turno = IMPARES;
int dato; // Variable compartida, var. global.
```

H1() {

```
for (int i=2; i < 1000; i+=2) {
    // comprobamos si es su turno, si no a esperar.
    lock (&mutex);
    while (turno != PARES) {
        cond_wait (&vcPares, &mutex);
        // dentro de cond-wait, se realiza unlock y lock
    }
    dato = i;
    turno = CONSUMIR;
    cond_signal (&vcConsumir);
    unlock (&mutex);
}
```

}

}

H2() // Impar

```
int i;
for (i=1; i < 1000; i+=2) {
    lock (&mutex);
    while (turno != IMPARES) {
        cond_wait (&vcImpar, &mutex);
    }
    dato = i;
    turno = CONSUMIR;
    cond_signal (&vcConsumir);
    unlock (&mutex);
}
```

7

H3() // Imprimir

```
int i;
for (i=1; i < 1000; i++) {
    lock (&mutex);
    while (turno != CONSUMIR)
```

```

        cond - wait (&vcConsumir, &mutex);
    {
        printf("Dato: %d\n", dato);
        // Si el dato es par, toca generar el impar.
        if (dato % 2 == 0)
        {
            turno = IMPARES;
            cond - signal (&vcImpar);
        }
        else
        {
            turno = PARES;
            cond - signal (&vcPar);
        }
        unlock (&mutex);
    }
    // for
}
// función.

```

b) main()

```

void * sh - mem;
sem_t * p - sem;
int * p - data;

```

```

sh - mem = mmap (NULL, 3 * sizeof (sem_t) + sizeof (int),
PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANON, (-1, 0));

```

// Solicitar región de memoria compartida para compartir los recursos de sincronización.

```
p - sem = sh - mem;
```

```
p - data = sh - mem + 3 * sizeof (sem_t);
```

// Hay que inicializar los semáforos de Impar, Par, y Consumidor

Pares sem - init (&p - sem[0], 1, 0); → el valor que desea inicializar.
impares sem - init (&p - sem[1], 1, 1); → 0 para que se quede bloqueado
consumidor sem - init (&p - sem[2], 1, 0);

if (!fork()) // Devuelve 0, ejecuta el hijo.

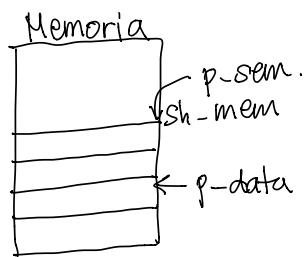
```
pares (p - data, p - sem[0]);
```

```
exit(0)
```

?else if (!fork()) // Vuelve a crear otro proceso de hijo

```
impares (p - data, p - sem[1]);
```

```
exit(0);
```



MAP_SHARED → memoria compartida
→ los hijos pueden acceder.
pueden ir elementos
cualesquier porque se ignoran

MAP_ANON → memoria que no tiene nombre.

```

} else // el padre sigue ejecutándose
    consumidor(p-data, p-sem[2]);
sem - destroy (&p-sem[0]);
sem - destroy (&p-sem[1]);
sem - destroy (&p-sem[2]);
UnMap (sh-mem, 3*sizeof(Sem_t)+sizeof(int));
// liberar la memoria que hemos reservado ↑
exit(0)
}
// main

```

```

void Pares(int *data, sem_t *p-sem) {
    int i;
    for (i = 2; i < 1000; i += 2) {
        sem - wait (&p-sem[0]);
        // al inicializarlo al 0 se queda bloqueado.
        *data = i;
        sem - post (&p-sem[2]);
    }
}

```

```

void Impares(int *data, sem_t *p-sem) {
    int i;
    for (i = 1; i < 1000; i += 2) {
        sem - wait (&p-sem[1]);
        *data = i;
        sem - post (&p-sem[2]);
    }
}

```

```

void consumidor(int *data, sem_t *p-sem) {
    int i;
    for (i = 1; i < 1000; i++) {
        // se comunica a través de la memoria compartida
        sem - wait (&p-sem[2]);
    }
}

```

```
printf ("Data: %d\n", *data);
if (data % 2 == 1)
    sem-post (&p-sem[0]);
else
    sem-post (&p-sem[1]);
{
}
```

4.- El Problema de los Fumadores [Suhas Patil]: considerar un sistema con tres procesos fumadores y un proceso agente. Continuamente cada fumador se prepara un cigarrillo y lo fuma. Para hacer un cigarrillo se necesitan tres ingredientes: tabaco, papel y cerillas. Uno de los procesos tiene infinito papel, otro tabaco y el tercero cerillas. El agente tiene provisión infinita de los tres ingredientes.

El agente coloca dos ingredientes distintos y de forma aleatoria en la mesa. El fumador que tiene el ingrediente que falta puede proceder a preparar y fumar un cigarrillo, haciendo una señal al agente cuando acaba, el agente en este instante repite la operación poniendo otros dos de los tres ingredientes en la mesa. La operación se repite indefinidamente.

Escribir un programa que sincronice al agente y los fumadores mediante semáforos o mutexes y variables condicionales. Asúmase que el agente no tiene forma de consultar los ingredientes que posee cada fumador.

Necesitamos variables de 3 ingredientes y una agente.

Tiene que ser global para que todos los hilos puedan acceder,

Para acceder, necesitamos una sección crítica → crear mutex global.

```
int mesa[3] = {0, 0, 0} // mesa[0] = cerillas, mesa[1] = papel, mesa[2] = tabaco  
mutex_t mutex;           1 = true (hay), 0 = false (no hay)  
vc_t vcmesa; // 0 con 2 variables.
```

// Coger aleatoriamente dos componentes.

1. Generar dos aleatorios que tienen que ser distintos en [0, 2]
2. Esperar a que mesa quede libre.
3. Si está libre la mesa, tiene que poner ingredientes
4. La forma sencilla: avisar a todos.
5. Volver a ejecutarlo.

```

    Agente()
    {
        while(1)
        {
            i = random % 3; // un n° aleatorio [0,2]
            do
            {
                j = random % 3;
            } while(i == j); // tenemos dos n° distintos
            lock(&mutex);
            while(mesa[0] || mesa[1] || mesa[2])
            {
                // mientras haya (hay que esperar a que
                // se quede vacío
                cond-wait(&vcMesa, &mutex);
            }
            mesa[i] = mesa[j] = 1
            cond-broadcast(&vcMesa);
            unlock(&mutex);
        }
    }

```

en C.

Void FumadorPapel()
 {
 1º Acceder a la mesa y comprobar si hay tabaco y cerilla.
 2º Esperar si no están los dos ingredientes que necesitamos
 3º Retirar los ingredientes en la mesa
 4º Avisar / Desbloquear a todos
 5º Liberamos la mesa
 6º Fumar
 }

```

    while(1)
    {
        lock(&mutex);
    }

```

```

while (!mesa[0] && !mesa[2]) {
    cond-wait (&vcMesa, &mutex);
}
mesa[0] = mesa[2] = 0;
cond-broadcast (&vcMesa);
unlock (&mutex); → cond-signal (&vcAgente);
Fumar();
}
// while
}
// función

```

```

void FumadorCerilla() {
    while (1) {
        lock (&mutex);
        while (!mesa[1] && !mesa[2]) {
            cond-wait (&vcMesa, &mutex);
}
mesa[1] = mesa[2] = 0;
cond-broadcast (&vcMesa);
unlock (&mutex); → cond-signal (&vcAgente);
Fumar();
}
}

```

```

void FumadorTabaco() {
    while(1) {
        lock(&mutex);
        while (!mesa[0] && !mesa[1])
            cond-wait (&vcMesa, &mutex);
        mesa[0]=mesa[1]=0;
        cond-broadcast (&vcMesa);
        unlock (&mutex);
        Fumar();
    }
}

```

CON SEMÁFOROS

```

sem_t sem_cons(0), sem_agente, sem_mesa(1);
int nBloqueados=0;
int mesa[3]={0,0,0} // Tabaco, Cerilla, Papel

```

→ consumidor
↓
Al principio
está libre.

→ usaremos
como un mutex.

```

void Agente() {
    int i,j;
    while(1) {
        i = random % 3;
        do {
            j = random % 3;
        } while (i==j)
        wait (&sem_mesa);
        mesa[i]=1;
    }
}

```

Despertar a
 todos hilos que
 estaban bloqueados
 porque semáforos
 no tienen broadcast. }
 ?
 mesa[j] = i;
 while (nBloqueados > 0) {
 SEM - POST (&SEM - CONS);
 nBloqueados --;
 ?
 POST (&SEM - mesa);
 ? → WAIT (&SEM - agente);
 ?
 }

void FumadorTabacoCerilla (int ing1, int ing2) {
 while (1) {
 ?
 wait (&SEM - mesa);
 if (mesa[ing1] && mesa[ing2]) {
 // VAMOS A RETIRAR LOS INGREDIENTES
 mesa[ing1] = mesa[ing2] = 0;
 post (&SEM - mesa); // Liberar mutex
 post (&SEM - agente); // Avisear. →
 fumar();
 }
 ? else {
 // QUEDARSE BLOQUEADO ESPERANDO
 nBloqueados++;
 post (&SEM - mesa);
 wait (&SEM - CONS);
 }
 } // Función.

5.- Una tribu de salvajes se sirven comida de un caldero con M raciones de estofado de misionero. Cuando un salvaje desea comer, se sirve una ración del caldero a menos que esté vacío. Si está vacío deberá avisar al cocinero para que reponga otras M raciones de estofado, y entonces se podrá servir su ración. Un cocinero y número arbitrario de salvajes se comportan del siguiente modo:

```
//Cocinero:  
while (True){  
    putServingsInPot(M)  
}  
  
//Salvajes:  
while (True){  
    getServingFromPot()  
    eat()  
}
```

Sobre este código realice los siguiente:

- 1) Añada el código necesario para garantizar la correcta sincronización, usando semáforos POSIX y variables de tipo entero, booleano...
 - 2) Añada el código necesario para garantizar la correcta sincronización, usando cerrojos y variables condicionales

Se deben cumplir las siguientes restricciones:

- a) Los salvajes no pueden invocar `getServingFromPot()` si el caldero está vacío
 - b) El cocinero sólo puede invocar `putServingsInPot(M)` si el caldero está vacío

```

int raciones = 0;
    CON VC y CEROJOS
mutex_t mutex;
vc_t calderoVacio, calderoLleno;

void salvaje() {
    while (1) {
        lock (&mutex);
        while (raciones == 0) {
            → cond-signal (&calderoVacio);
            → cond-wait (&calderoLleno, &mutex);
        }
        raciones--;
        getServingFromPot();
        unlock (&mutex);
        eat(); // Se ejecuta concurrentemente.
    }
}

```

```

void cocinero() {
    while (1) {
        lock (&mutex);
        while (raciones > 0)
            cond - wait (&calderoVacio, &mutex); →
        raciones = M;
        putServingsInPot();
        cond - broadcast (&calderoLleno); →
        unlock (&mutex);
    }
}

```

SEMAFORO UNO COMO SI FUERA mutex,
 Y DOS SEMS PARA COLA DE ESPERA.
 VARIABLE COMPARTIDA → ACCEDER A LA SECCIÓN CRÍTICA.

int raciones = 0
 sem_t sem_caldoro(1), sem_caldoroVacio(0), sem_caldoroLleno(0);
 mutex.

```

void Salvaje() {
    while (1) {
        // ACCEDER AL CALDERO
        lock (&mutex);
        wait (&sem_caldoro); →
        if (raciones == 0) {
            // ME QUEDO ESPERANDO
            post (&sem_caldoroVacio); →
            → wait (&sem_caldoroLleno);
            // AQUÍ NO PASEMOS EL mutex, POR LO TANTO
            // NO SE LIBERA. NO DEBERÍA HACER EL
            // cocinero?
            raciones = 1; →
            Lo ponemos aquí para que se
            cambie cuando tiene el mutex.
        }
    }
}

```

raciones →
getServingsFromPot();
post (& Sem - Calder); unlock (mutex)
eat();

Void Cinerot) {

while(1){}

while(1){}

→ wait(&Sem - Calderolacio);

→ putServingsInfo();

post (& sem- Caldero Lieno); →

2 L'Asia al finim salvis que

metex

۳

Con semáforo, se garantiza el orden, mientras el otro no.

12.- El Barbero Dormilón: una barbería está compuesta por una sala de espera, con n sillas, y la sala del barbero, que tiene un sillón para el que está siendo atendido. Las condiciones de atención a los clientes son las siguientes:

- a) Si no hay ningún cliente, el barbero se va a dormir
- b) Si entra un cliente en la barbería y todas las sillas están ocupadas, el cliente abandona la barbería
- c) Si hay sitio y el barbero está ocupado, se sienta en una silla libre
- d) Si el barbero estaba dormido, el cliente le despierta
- e) Una vez que el cliente va a ser atendido, el barbero invocará `cortarPelo()` y el cliente `recibirCortePelo()`

Escriba un programa que coordine al barbero y a los clientes utilizando mutex y semáforos POSIX para la sincronización entre procesos.

```
int numSillasOcupadas = 0;
const int MAX = 5;
mutex_t mutex;
sem_t sillón(0), afeitado(0);
int dormido = 0 // boolean false

void cliente() {
    lock(&mutex);
    // Si hay silla libre, ocupo una y entro
    // Si no salgo
    if (numSillasOcupadas < MAX) {
        // Ocupar la silla
        numSillasOcupadas++;
        if (dormido) {
            post(&sillón);
            dormido = 0;
        }
    }
    unlock(&mutex);
}
```

```

    wait (&afeitado);
    recibirCortePelo();
} else {
    unlock (&mutex);
}
// Cliente

```

```

void Barbero() {
    while (1) {
        // Comprobar si hay clientes para atender.
        lock (&mutex);
        if (numSillasOcupadas == 0) {
            dormido = 1;
            unlock (&mutex);
            wait (&sillon);
            unlock (&mutex);
        }
        numSillasOcupadas--;
        unlock (&mutex);
        Post (&afeitado);
        CortarPelo();
    }
    // White
}
// Barbero

```

j porque
 porque
 siempre

while(1)?
 el barbero
 está esperando al cliente
 en cambio el cliente
 viene de vez en
 cuando.