Ejercicios Tema 2:

David Martinez Diaz GII-ADE

Ejercicio 61:

En un pueblo hay una pequeña barbería con una puerta de entrar y otra de salida, dichas puertas son tan estrechas que sólo permiten el paso de una persona cada vez que son utilizadas. En la barbería hay un número indefinalizar (nunca se agotan) de sillas de 2 tipos:

(a) sillas donde los clientes esperan a que entren los barberos,

(b) sillas de barbero donde los clientes esperan hasta que termina su corte de pelo. No hay espacio suficiente para que más de una persona (barbero o cliente) pueda moverse dentro de la barbería en cada momento, p.e., si los clientes se dan cuenta que ha entrado un barbero, entonces sólo 1 cliente puede levantarse y dirigirse a una silla de tipo (b); un barbero, por su parte, no podría moverse para ir a cortarle el pelo hasta que el cliente se hubiera sentado. Cuando entra un cliente en la barbería, puede hacer una de estas 2 cosas:

- Aguarda en una silla de tipo (a) y espera a que existan barberos disponibles para ser atendido, cuando sucede esto último, el cliente se levanta y vuelve a sentarse, esta vez en una silla de tipo (b), para esperar hasta que termine su corte de pelo.
- Se sienta directamente en una silla de tipo (b), si hay barberos disponibles.

Un cliente no se levanta de la silla del barbero hasta que este le avisa abriéndole la puerta de salida. Un barbero, cuando entra en la barbería, aguarda a que haya clientes sentados en una silla de tipo (b) esperando su corte de pelo. Después revisa el estado de los clientes, siguiendo un orden numérico establecido, hasta que encuentra un cliente que espera ser atendido, cuando lo encuentra comienza a cortarle el pelo y él mismo pasa a estar ocupado. Cuando termina con un cliente, le abre la puerta de salida y espera a que el cliente le pague, después sale a la calle para refrescarse. El barbero no podrá entrar de nuevo en la barbería hasta que haya cobrado al cliente que justamente acaba de atender. No se admite que un cliente pague a un barbero distinto del que cortó el pelo.

Resolver el problema anterior utilizando un solo monitor que defina los siguientes procedimientos:

```
Monitor Barberia;

int numero_barberos;
int numero_clientes;

int total_numero_clientes;

CondVar silla_A = new CondVar();
CondVar espera_barbero = new CondVar();

CondVar sala_espera[];

CondVar corte_hecho[];
bool termina_corte[];

CondVar espera_cobrar[];
bool cobrar[];

CondVar permiso_irse[];
bool tiene_permiso[];

//-------------------------------------------------------
void corte_de_pelo(int numero_de_cliente){

    // Si no hay barberos lo sentamos en una silla tipo a).
    if(numero_barberos == 0)
        silla_A.wait();

    numero_clientes++;
    total_numero_clientes++;

    espera_barbero.signal();
    // Si hay barberos disponibles sentarse en una silla del tipo b) y esperar a
que venga un barbero.

    sala_espera[numero_de_cliente].wait();

    // Espera a que le corten el pelo
    if(termina_corte[numero_de_cliente] == false)
        corte_hecho[numero_de_cliente].wait();

    cobrar[numero_de_cliente] = true;
    espera_cobrar[numero_de_cliente].signal();

    if(tiene_permiso[numero_de_cliente] == false)
        permiso_irse[numero_de_cliente].wait();

    numero_clientes--;
}
//-------------------------------------------------------

int siguiente_cliente(){
```

```
    numero_barberos++;

    if(numero_clientes == 0){
        espera_barbero.wait();
    }
    silla_A.signal();

    int cliente_esperando;
    bool check = false;
    for(int n=0; n<total_numero_clientes && !check; n++){

        if(!(sale_espera[n].empty())){
            cliente = n;
            check = true;

            sala_espera[n].signal();
        }
    }

    return cliente;
}

//------------------------------------------------------

void termina_corte_de_pelo(int numero_de_cliente){

    termina_corte[numero_de_cliente] = true;
    corte_hecho[numero_de_cliente].signal();

    // Espera a cobrar
    if(cobrar[numero_de_cliente] == false)
        espera_cobrar[numero_de_cliente].wait();

    tiene_permiso[numero_de_cliente] = true;
    permiso_irse[numero_de_cliente].signal();

    numero_barberos--;

}

void funcion_hebra_barbero(MRef <Barberia> barberia){

    while(true){

        int cliente = barberia->siguiente_cliente();
        barberia->termina_corte_de_pelo(cliente);
    }
}

void funcion_hebra_cliente(MRef <Barberia> barberia, int num_cliente){

    while(true){
        barberia->corte_de_pelo(num_cliente);
    }
```

```
}
```

Ejercicio 62:

Suponer un garaje de lavado de coches con dos zonas: una de espera y otra de lavado con 100 plazas. Un coche entra en la zona de lavado del garaje sólo si hay (al menos) una plaza libre, si no, se queda en la cola de espera. Si un coche entra en la zona de lavado, busca una plaza libre y espera hasta que es atendido por un empleado del garaje.

Los coches no pueden volver a entrar al garaje hasta que el empleado que les atendió les cobre el servicio. Suponemos que hay más de 100 empleados que lavan coches, cobran, salen y vuelven a entrar al garaje.

Cuando un empleado entra en el garaje comprueba si hay coches esperando ser atendidos (ya situados en su plaza de lavado), si no, aguarda a que haya alguno en esa situación. Si hay al menos un coche esperando, recorre las plazas de la zona de lavado hasta que lo encuentra, entonces lo lava, le avisa de que puede salir y, por último, espera a que le pague.

Puede suceder que varios empleados han finalizar de lavar sus coches y están todos esperando el pago de sus servicios, no se admite que un empleado cobre a un coche distinto del que ha lavado. También hay que evitar que al entrar 2 ó más empleados a la zona de lavado, habiendo comprobado que hay coches esperando, seleccionen a un mismo coche para lavarlo.

```
Monitor Garaje;

int numero_plazas;
int numero_empleados;

int total_numero_coches;

CondVar espera_empleado = new CondVar();
CondVar parking_espera = new CondVar();

CondVar parking_lavado[];
CondVar lavado_hecho[];

bool termina_lavado[];

CondVar espera_cobrar[];
bool cobrar[];

CondVar permiso_irse[];
bool tiene_permiso[];


void lavado_de_coche(int num_coche){

    // Si no hay plazas lo dejamos en la sala de espera.
    if(numero_plazas >= 100)
```

```
        parking_espera.wait();

    numero_plazas++;
    total_numero_coches++;

    espera_empleado.signal();

    // Si hay empleados disponibles esperar en la zona de lavado y esperar a que
venga un empleado.
    parking_lavado[num_coche].wait();

    // Espera a que laven el coche

    if(termina_lavado[num_coche] == false)
        lavado_hecho[num_coche].wait();

    cobrar[num_coche] = true;
    espera_cobrar[num_coche].signal();

    if(tiene_permiso[num_coche] == false)
        permiso_irse[num_coche].wait();

    numero_plazas--;

}

int siguiente_coche(){


    numero_empleados++;

    // Si no hay ninguna plaza ocupada, el empleado espera
    if(numero_plazas == 0 || numero_plazas >= 100){
        espera_empleado.wait();
    }

    parking_espera.signal();

    int coche_esperando;
    bool check = false;
    for(int n=0; n<total_numero_coches && !check; n++){

        if(!(parking_lavado[n].empty())){
            coche_esperando = n;
            check = true;
        }
    }

    return coche_esperando;
}

void terminar_y_cobrar(int num_coche){

    termina_lavado[num_coche] = true;
```

```
    lavado_hecho[num_coche].signal();

    // Espera a cobrar
    if(cobrar[num_coche] == false)
        espera_cobrar[num_coche].wait();

    tiene_permiso[num_coche] = true;
    permiso_irse[num_coche].signal();

    numero_empleados--;
}

void funcion_hebra_empleado(MRef <Garaje> garaje){

    while(true){

        int coche = garaje->siguiente_coche();
        garaje->terminar_y_cobrar(coche);
    }
}

void funcion_hebra_coche(MRef <Garaje> garaje, int num_coche){

    while(true){
        garaje->lavado_de_coche(num_coche);
    }
}
```

Ejercicio 63:

Demostrar que el monitor "productor-consumidor" es correcto utilizando las reglas de corrección de la operación c.wait() y c.signal() para señales desplazantes. Considerar el siguiente invariante del monitor IM: I == { 0 ≤ n ≤ MAX }.

```
Monitor Bufer;

int n;
int frente;
int atras;

CondVar no_vacio;
CondVar no_lleno;

tipo_dato array[MAX];

Bufer::Bufer(){

    frente=1;
    atras= 1;
```

```
    n= 0;
}

void insertar(tipo_dato d){

    if ((atras+1) % MAX == frente){

        // {n == MAX ∧ IM}
        no_lleno.wait()
        // {n = MAX-1} → {IM}
    }

    else NULL;

    buf[atras]= d;
    atras = (atras+1)%MAX ;

    n++;

    //  { 0 < n <= MAX }
    no_vacio.signal();
    // {0 <= n < MAX }
}

void retirar(tipo_dato x){

    if ( frente==atras){

        // { n = 0 } → {IM}
        no_vacio.wait();
        // { 0 < n <= MAX }
    }

    else NULL;

    x = buf[frente];
    frente = (frente+1) % MAX;
    n--;

    // {0 <= n < MAX }
    no_lleno.signal();
    // { 0 < n <= MAX }
}
```

Ejercicio 64:

Demostrar la corrección de un monitor que implemente las operaciones de acceso al buffer circular para el problema del productor-consumidor utilizando el siguiente invariante:

- $0 \leq n \leq N$; No hay procesos bloqueados
- $0 > n$; Hay $|n|$ consumidores bloqueados

- n > N; Hay (n − N) productores bloqueados

Escribir un monitor que cumpla el invariante anterior, es decir:

- Se haga novacio.signal() solamente cuando haya consumidores bloqueados.
- Se haga nolleno.signal() solamente cuando haya productores bloqueados.

Se supone que el buffer tiene N posiciones y se utilizan las dos señales mencionadas anteriormente. No está permitido utilizar la operación c.queue() para saber si hay procesos bloqueados en alguna cola de variable condición. Nótese que el invariante del monitor Bufer comprende ahora 3 propiedades.

La interpretación de este nuevo invariante es más amplia que la del IM del problema 63: los valores negativos de "n" representan (en valor absoluto) el número de procesos consumidores bloqueados, cuando el buffer está vacío y (n-N) es idénticamente igual al número de productores bloqueados cuando está lleno; cuando el valor de "n" se mantiene entre los límites: 0, N tendríamos, por tanto, y como caso particular, que se cumpliría el IM en las mismas condiciones del problema 63.

```
Monitor Bufer;

int n;
int frente;
int atras;

CondVar no_vacio;
CondVar no_lleno;

tipo_dato array[MAX];

Bufer::Bufer(){

    frente=0;
    atras= 0;
    n= 0;
}

void insertar(tipo_dato d){

    n++;

    if(n > MAX){

        // { n > N } → {IM} y Al menos un productor bloqueado (n-N).
        no_lleno.wait();
        // { 0 ≤ n ≤ N } V { n > N } No hay procesos bloqueados o hay al menos un
productor bloqueado (n-N).
    }


    buf[atras]= d;
    atras = (atras%MAX) + 1;
```

```
    // { 0 ≤ n ≤ N } V { n > N } V { 0 > n }
    no_vacio.signal();
    // { 0 ≤ n ≤ N } V { n > N } V { 0 > n }
}

void retirar(tipo_dato x){

    n--;

    if(n<0){

        // { 0 > n } → {IM} y Al menos un consumidor bloqueado |n|.
        no_vacio.wait();
        // { 0 ≤ n ≤ N } V {0 > n} No hay procesos bloqueados o hay al menos un
consumidor bloqueado |n|.
    }


    x = buf[frente];
    frente = (frente%MAX) + 1;

    // { 0 ≤ n ≤ N } V { n > N } V { 0 > n }
    no_lleno.signal();
    // { 0 ≤ n ≤ N } V { n > N } V { 0 > n }
}
```

Ejercicio 65

Considerar el programa concurrente mostrado más abajo. En dicho programa hay 2 procesos, denominados P1 y P2, que intentan alternarse en el acceso al monitor M. La intención del programador al escribir este programa era que el proceso P1 esperase bloqueado en la cola de la seãl p, hasta que el proceso P2 llamase al procedimiento M.sigue() para desbloquear al proceso P1; después P2 se esperaría hasta que P1 terminase de ejecutar M.stop(), tras realizar algún procesamiento se ejecutaría q.signal() para desbloquear a P2. Sin embargo el programa se bloquea.

- (a) Encontrar un escenario en el que se bloquee el programa.

a) Un caso donde se bloquea es cuando accede antes la funcion de la hebra P2, cuando llama a M.sigue(), por lo que se quedaria bloqueado cuando luego la funcion de la hebra P1, al llamar a M.stop(), quedandose en bucle infinito porque ha perdido el signal().

```


Monitor M () {

    Condvar p = new CondVar();
    Condvar q = new CondVar();

    void stop(){
```

```
        p.wait();
        q.signal();
    }

    void sigue(){

        p.signal();
        q.wait();
    }

}

funcion_hebra_P1(){

    while(true) do{
        M.stop();
        //...
    }
}

funcion_hebra_P2(){

    while(true) do{
        M.sigue();
        //...
    }
}
```

- (b) Modificar el programa para que su comportamiento sea el deseado y se eviten interbloqueos.

```
Monitor M () {

    Condvar p = new CondVar();
    Condvar q = new CondVar();

    bool check = false;

    void stop(){

        if(!check){
            p.wait();
        }

        check = false;
        q.signal();
    }

    void sigue(){

        check = true;
```

```
        p.signal();
        q.wait();
    }

}

funcion_hebra_P1(){

    while(true) do{
        M.stop();
        //...
    }
}

funcion_hebra_P2(){

    while(true) do{
        M.sigue();
        //...
    }
}
```

Ejercicio 66:

Indicar con qué tipo (o tipos) de señales de los monitores (SC,SW o SU) sería correcto el código de los procedimientos de los siguientes monitores que intentan implementar un semáforo FIFO.

Modificar el código de los procedimientos de tal forma que pudieran ser correctos con cualquiera de los tipos de señales anteriormente mencionados.

- **Monitor A)**

Funciona para: Monitor SU (Señalar y Espera Urgente) y tambien SW (Señalar y Esperar).

No funciona para Monitor SC (Señalar y Continuar).

Para arreglarlo simplemente sustituir el if(s==0) por un while(s==0).

```
Monitor Semaforo{

    int s;
    CondVar c;

    Semaforo::Semaforo(){
        s = 0;
    }

    void* P(void* arg){

        // Modificacion
```

```
        while (s == 0)
            c.wait();

        s= s-1;
    }

    void* V(void* arg){

        s = s+1;
        c.signal();
    }
}
```

- **Monitor B)**

Funciona para: Monitor SC (Señalar y Continuar).

No funciona para Monitor SU y SW.

Para arreglarlo simplemente sustituir el if(s==0) por un while(s==0).

```
Monitor Semaforo{

    int s;
    CondVar c;

    Semaforo::Semaforo(){
        s= 0;
    }
    void* P(void* arg){

        // Modificacion
        s= s-1;

        while (s == 0){
            c.wait();
        }


    }

    void* V(void* arg){

        // Modificacion
        s= s+1;
        c.signal();

    }
}
```

- **Monitor C)**

-

El invariante del monitor es IM = { s >= 0 }

Funciona para: Monitor señales SW y SU.

No funciona para Monitor SC, ya que puede abandonar el monitor antes de que termine su ejecución.

Para arreglarlo simplemente sería crear “una cola de urgentes” para marcar las hebras que están esperando.

```
Monitor Semaforo{

    int s;
    CondVar c;

    Semaforo::Semaforo(){
        s= 0;
    }

    void* P(void* arg){

        if (s == 0){
            c.wait();
        }
        else {
            s = s-1;
        }

    }

    void* V(void* arg){

        if(c.queue()){
            c.signal();
        }
        else {
            s= s+1;
        }
    }

}
```

Ejercicio 67:

Escribir el código de los procedimientos P() y V() de un monitor que implemente un semáforo general con el siguiente invariante: {s ≥ 0} ∨ {s = s0 + nV − nP } y que sea correcto para cualquier semántica de señales. La implementación ha de asegurar que nunca se puede producir “robo de señal" por parte de las hebras que llamen a las operaciones del monitor semáforo anterior.

```
Monitor Semaforo{

    int s;
    CondVar c;

    Semaforo::Semaforo(){
        s = 0;
    }

    void* P(void* arg){

        // Modificacion
        if (s = 0){

            c.wait();
        }
        else{

            s= s-1;
        }
    }

    void* V(void* arg){

        if(c.queue()){

            c.signal();
        }
        else{

            s = s+1;
        }
    }

}
```

Ejercicio 68:

Suponer un número desconocido de procesos consumidores y productores de mensajes de una red de comunicaciones muy simple. Los mensajes se envían por los productores llamando a la operación broadcast(int m) (el mensaje se supone que es un entero), para enviar una copia del mensaje m a las hebras consumidoras que previamente hayan solicitado recibirlo, las cuales están bloqueadas esperando.

Otra hebra productora no puede enviar el siguiente mensaje hasta que todas las hebras consumidoras no reciban el mensaje anteriormente envíado. Para recibir una copia de un mensaje enviado, las hebras consumidoras llaman a la operación int fetch(). Mientras un mensaje se esté transmitiendo por la red de comunicaciones, nuevas hebras consumidoras que soliciten recibirlo lo reciben inmediatamente sin esperar.

La hebra productora, que envió el mensaje a la red, permanecerá bloqueada hasta que todas las hebras consumidoras solicitantes efectivamente lo hayan recibido. Se pide programar un monitor que incluya entre

sus métodos las operaciones: broadcast(int m), int fetch(), suponiendo una semántica de señales desplazantes y SU.

```
Monitor ProdCons;

CondVar enviar_mensaje;
CondVar esperar;

int solicitantes;
int mensaje;

ProdCons::ProdCons(){

    enviar_mensaje = new CondVar();
    esperar = new CondVar();
}

void broadcast(int m){

    if(!enviar_mensaje.empty())
        esperar.wait();

    mensaje = m;
    while(solicitantes != 0){

        enviar_mensaje.signal();
        solicitantes--;
    }

}

int fetch(){

    int mensaje_recibido;
    solicitantes++;

    if (enviar_mensaje.empty())
        esperar.signal();

    enviar_mensaje.wait();
    mensaje_recibido = mensaje;

    return mensaje_recibido;
}

//-------------------------------------------------------

void funcion_hebra_productor(MRef <ProdCons> monitor){

    int mensaje;
    while(true){
```

```
        mensaje = NextRandomInt();
        monitor->broadcast(mensaje);
    }
}

void funcion_hebra_consumidor(MRef <ProdCons> monitor){

    int mensaje_recibido;

    while(true){

        mensaje_recibido = monitor->fetch();
        cout << "He recibido el mensaje " << mensaje_recibido;
    }
}
```

Ejercicio 69:

Suponer un sistema básico de asignación de páginas de memoria de un sistema operativo que proporciona 2 operaciones: adquirir(int n) y liberar(int n) para que los procesos de usuario puedan obtener las páginas que necesiten y, posteriormente, dejarlas libres para ser utilizadas por otros.

Cuando los procesos llaman a la operación adquirir(int n), si no hay memoria disponible para atenderla, la petición quedaría pendiente hasta que exista un número de páginas libres suficiente en memoria. Llamando a la operación liberar(int n) un proceso convierte en disponibles n páginas de la memoria del sistema.

Suponemos que los procesos adquieren y devuelven páginas del mismo tamaño a un área de memoria con estructura de cola y en la que suponemos que no existe el problema conocido como fragmentación de páginas de la memoria. Se pide programar un monitor que incluya las operaciones anteriores suponiendo semántica de señales desplazantes y SU.

(a) Resolverlo suponiendo orden FIFO estricto para atender las llamadas a la operación de adquierir páginas por parte de los procesos.

```
Invariante del monitor : Memoria_disponible ≤ M0 ∧ peticiones_pendientes →
obtener.queue() ∧ Memoria_disponible ≥ 0;

Condiciones de sincronización:
- Memoria_disponible > 0 (señalar obtener)
- ¬peticiones_pendientes (señalar esperar)

Monitor MemoriaFIFO;

const M0;

int mem_disponible;
bool peticiones_pendientes;
```

```
CondVar esperar;
CondVar obtener;

MemoriaFIFO::MemoriaFIFO(){
    mem_disponible = M0;
    peticiones_pendientes = false;
}

void solicitar(){

    if (peticiones_pendientes || obtener.queue())
        esperar.wait();

    peticiones_pendientes = true;
}

void adquirir(int n){

    while (mem_disponible < n) do{
        obtener.wait();
    }

    mem_disponible = mem_disponible - n;
    peticiones_pendientes = false;

    if (esperar.queue())
        esperar.signal();

}

void liberar(int n){

    mem_disponible = mem_disponible + n;

    if (obtener.queue())
        obtener.signal();

}
```

(b) Relajar la condición anterior y resolverlo atendiendo las llamadas según el orden: petición pendiente con "menor número de páginas primero" (SJF).

```
Monitor MemoriaSJF;

const M0;

int mem_disponible;
bool peticiones_pendientes;
```

```
CondVar esperar;
CondVar obtener;

MemoriaSJF::MemoriaSJF(){

    mem_disponible = M0;
    peticiones_pendientes = false;
}

void adquirir(int n){

    while (mem_disponible < n) do{
        obtener.wait();
    }

    mem_disponible = mem_disponible - n;

    if (obtener.queue())
        obtener.signal();
}

void liberar(int n){

    mem_disponible = mem_disponible + n;

    if (obtener.queue())
        obtener.signal();
}
```

Ejercicio 70:

Diseñar un controlador para un sistema de riegos que proporcione servicio cada 72 horas. Los usuarios del sistema de riegos obtienen servicio del mismo mientras un depósito de capacidad máxima igual a C litros tenga agua. Si un usuario llama al procedimiento abrir_cerrar_valvula(cantidad:positive) y el depósito está vacío, entonces ha de señalarse al proceso controlador y la hebra que soporta la petición del citado usuario quedará bloqueada hasta que el depósito esté otra vez completamente lleno.

Si el depósito no se encontrase lleno o contiene menos agua de la solicitada, entonces el riego se llevará a cabo con el agua que haya disponible en ese momento. La ejecución del procedimiento: control_en_espera() mantiene bloqueado al controlador mientras el depósito no esté vacío. El llenado completo del depósito se produce cuando el controlador llama al procedimiento: control_rellenando, de tal forma que cuando el depósito esté completamente lleno (=C litros) se ha de señalar a las hebras-usuario bloqueadas para que terminen de ejecutar las operaciones de "abrir y cerrar válvula" que estaban interrumpidas.

- (a) Programar las 3 operaciones mencionadas en un monitor que asumen semántica de señales desplazantes y SU.
- (b) Demostrar que las hebras que llamen a las operaciones del monitor anterior nunca pueden llegar a entrar en una situación de bloqueo indefinido.

```
Monitor Riegos;

int cantidad_agua;
const int CANTIDAD_TOTAL_AGUA = 20;

CondVar esperar;
CondVar esta_vacio;

bool esta_vacio;

Riegos::Riegos(){

    esta_vacio = false;
    esperar = new CondVar();
    esta_vacio = new CondVar();
}

void abrir_cerrar_valvula(int litros){

    if(cantidad_agua <= litros){

        cout << "Riego con toda la capacidad disponible: "<< cantidad_agua;
        cantidad_agua -= cantidad_agua;
        esta_vacio = true;
    }
    else{

        cout << "Riego con litros solicitados: "<< litros;
        cantidad_agua -= litros;
    }

    if(esta_vacio){

        esta_vacio.signal();
        esperar.wait();
    }

}

void control_espera(){

    if(!esta_vacio)
        esta_vacio.wait();
}

void control_rellenando(){

    cout <<"Llenamos el tanque de agua";
    cantidad_agua = CANTIDAD_TOTAL_AGUA;

    esta_vacio = false;
```

```
    esperar.signal();
}

void funcion_hebra_controlador(MRef <Riegos> Riegos){

    while (true){

        Riegos.control_en_espera;
        Riegos.control_rellenando;
    }
}

void funcion_hebra_P (MRef <Riegos> Riegos, int i){

    while (true){

        Riegos.abrir_cerrar_valvula(litros);
    }
}
```