CFGS DAM

Primer Curso

Programación de Servicios y Procesos

UT2: PROGRAMACIÓN CONCURRENTE

Problemas típicos de concurrencia

Índice

1.Problema del productor/consumidor Enunciado

- El problema del productor-consumidor escribe dos procesos, productor y consumidor, y ambos comparten un búfer de tamaño finito.
  - La tarea del productor es generar un producto, almacenarlo y comenzar nuevamente;
  - La del consumidor es coger (simultáneamente) productos uno a uno.
- El problema consiste en que el productor no añada más productos que la capacidad del buffer y que el consumidor no intente tomar un producto si el buffer está vacío.

Problema del productor/consumidor
Solución general

- Ambos procesos se ejecutan simultáneamente y se "despiertan" o "duermen" según el estado del buffer. Concretamente:
- El productor agrega productos mientras quede espacio y en el momento en que se llene el buffer se pone a "dormir".
- Cuando el consumidor toma un producto notifica al productor que puede comenzar a trabajar nuevamente.
- En caso contrario, si el buffer se vacía, el consumidor se pone a dormir y en el momento en que el productor agrega un producto crea una señal para despertarlo.

Problema del productor/consumidor Solución 1

- Usamos una variable entera llamada count que guarda el número de elementos en el buffer
  - Inicialmente, count vale 0
  - Es incrementado por el productor cuando produce un nuevo valor y lo almacena en el buffer
  - Es decrementado por el consumidor cuando extrae un elemento del buffer

Problema del productor/consumidor Solución 1 (C)

- Productor:

```
while (true) {
    while (count == BUFFER_SIZE) {
        <<esperar>>
    }
    <<PRODUCIR>>
    <<aviso de que he producido>>
}
```

Produce un elemento y avisa de que lo ha producido.

Ej de código en C:

```
buffer [in] = <<elemento producido>>
in = (in + 1) % BUFFER_SIZE;
count++;
```

- Consumidor:

```
while (true) {
   while (count == 0) {
      <<esperar>>
   }
   <<CONSUMIR>>
    <<aviso de que he consumido >>
}
```

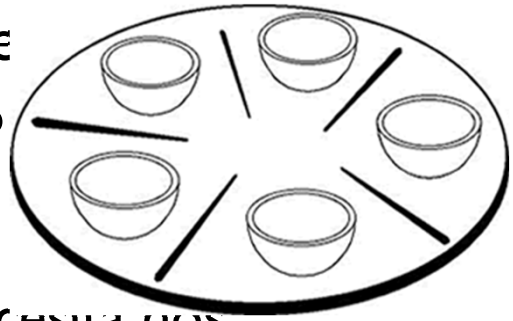
Consume el elemento y avisa de que lo ha consumido.

Ej de código en C:

```
nextConsumed =  buffer[out];
out = (out + 1) % BUFFER_SIZE;
count--;
```

2.Cena de Filósofos

Enunciado



- En 1965 Dijsktra planteó y resolvió este proble
  - 5 filósofos se sientan en la mesa, donde cada uno un plato de spaghetti
  - Entre cada dos platos hay un tenedor.
  - El spaghetti es tan escurridizo que un filósofo necesita dos tenedores para comerlo
  - La vida de un filosofo, consta de periodos alternados de comer y pensar
  - Cuando un filosofo tiene hambre, intenta obtener un tenedor para su mano izquierda y otro para su mano derecha. (primero coge uno y luego el otro sin importar por cual empieza),
  - Si logra obtener los dos tenedores, come un rato y después deja los tenedores y continua pensando,
- Objetivo:
  - Conseguir que cada filósofo lleve a cabo su tarea de forma indefinida, es decir, que el programa "nunca" acabe.

Cena de Filósofos
Solución general

- *El filósofo espera hasta que el palillo/tenedor especificado este disponible.*

```
#define N 5                                  /* número de filósofos */

void filósofo (int i)                        /* i: qué filósofo (desde 0 hasta N-1) */
{
while (1) {
    pensar ( );                             /* el filósofo está pensando */
    coger_tenedor (i);                       /* coge el tenedor izquierdo */
    coger_tenedor ((i + 1) % N); /* coge el tenedor derecho */
    comer ( );
    dejar_tenedor (i);                       /* deja el tenedor izquierdo en la mesa */
    dejar_tenedor ((i + 1) % N);  /* deja el tenedor derecho en la mesa */
}
}
```

  - *Por desgracia esta solución es incorrecta, ya que si, por ejemplo, los 5 filósofos toman sus tenedores izquierdos de forma simultánea se produciría un interbloqueo.*

    (Ya que Ninguno de ellos podría coger su palillo/tenedor derecho).

Cena de Filósofos.
Posibles Soluciones I

A)Por turno cíclico

- Se empieza por un filósofo, que si quiere, puede comer, y después pasa su turno al de la derecha. Cada filósofo sólo puede comer en su turno (es decir si tiene el turno=ficha o testigo).
- Problemas:
  - si el número de filósofos es muy alto, uno puede morir de hambre antes de que le toque el turno.
  - Se desaprovechan recursos, ya que solo puede comer un filósofo a la vez.

B)Varios turnos

- Se establecen varios turnos = se otorgan varias fichas.
  Así, si por ejemplo hay 7 comensales podemos poner 3 fichas en posiciones alternas.
- Para que funcione sin que se hagan cuellos de botella (las fichas de los filósofos que comen rápido pasan rápido a un lado de la mesa e impedirían comer de forma simultánea), **se establecen turnos de tiempo fijo**.
  Por ejemplo cada 5 minutos se pasan las fichas (y los turnos) a la derecha.
  - En base al tiempo que suelen tardar los filósofos en comer y en volver a tener hambre, puede ser mejor o peor el algoritmo.
    Si el tiempo de turno se aproxima al tiempo medio que tarda un filósofo en comer, esta variante da muy buenos resultados.
    Si además el tiempo medio de comer es similar al tiempo medio en volver a tener hambre la solución se aproxima al óptimo.

C)Filósofos pares/Impares

- Unos filósofos empiezan por coger el tenedor de la izquierda y otros el de la derecha.
- Es una solución óptima y la codificaremos más adelante.

D) Usando registros de estado <<ver Siguiente diapositiva>>

Cena de Filósofos.
Posibles Soluciones II

D)Usando registros de Estado

- Esta solución se basa e :
  - Utiliza un vector, estado, para llevar un registro con la actividad de un filósofo en cada momento:
    - El filósofo puede estar comiendo, pensando o hambriento (estado que indica que quiere coger los palillos para empezar a comer).
  - Mirar a los compañeros cuando se desea comer:
    - Un filósofo puede comer únicamente si los vecinos no están comiendo
  - La solidaridad:
    - Cuando un filósofo deja sus palillos, debe avisar a los otros de que ya pueden comer usando sus palillos.

- Nota:
  Si lo implementamos con semáforos (ver 2 diapositivas adelante):
  El vector estado no es un "semáforo original", ya que queremos consultar su estado en cualquier momento. Recuerda q ***sobre un semáforo, por definición****, solo se pueden ejecutar las operaciones de wait y signal una vez se haya inicializado,* ***no habiendo un getValor() ni nada parecido.***

Cena de Filósofos.
(Solución par/impar mediante semáforos en Pascal FC)

Esta solución es bastante efectiva, y consiste en diferenciar a los filósofos sentados en posiciones pares e impares.
De este modo si un filósofo para empezar a comer primero toma su palillo derecho, los dos filósofos de su lado tomarán primero su palillo izquierdo.

Para la siguiente solución definimos palillo como array[0..N] of semaphore siendo N los sitios de la mesa.
Todos los semáforos del array se inicializarán a 1.

```
process type filosofo_par(i:integer);
begin
   repeat
      piensa;
      wait(palillo[(i+1) mod N]);
      wait(palillo[i]);
      come;
      signal(palillo[i]);
      signal(palillo[(i+1) mod N]);
   forever
end;
```

```
process type filosofo_impar(i:integer);
begin
   repeat
      piensa;
      wait(palillo[i]);
      wait(palillo[(i+1) mod N]);
      come;
      signal(palillo[i]);
      signal(palillo[(i+1) mod N]);
   forever
end;
```

Cena de Filósofos.
(Solución con semáforos y RegEstado en C)

Solución comprobada En C++

Compartido

```
#define N               5
#define IZQD            (i-1)%N
#define DCHA            (i+1)%N
#define PENSANDO        0
#define HAMBRIENTO      1
#define COMIENDO        2
```

Los N filósofos estarán en uno de los 3 estados posibles

```
typedef int semaforo;
int estado[N];
semaforo mutex;
semaforo S[N];
```

Semáforo para acceder a la SC (Garantiza que dos filósofos no acceden a ejecutar intentos a la vez cuando están hambrientos), es decir, garantiza la exclusión mutua. ***Se inicia a 1***

*5 semáforos(1 para cada filósofo), **iniciados a 0**, para controlar si el filósofo puede o no comer.*

```
void filosofo(int i)
{
  while (1) {
    piensa();
    coge_palillos(i);
    come();
    deja_palillos(i);
  }
}
```

Coge los 2 tenedores, bloqueándose si no es posible

Con este down Impido que 2 filósofos intenten coger palillos "a la vez"

```
void coge_palillos(int i)
{
  down (&mutex);
  estado[i]= HAMBRIENTO;
  intento(i);
  up (&mutex);
  down (&S[i]);
}
```

Si hago este down sin haber hecho el up de intento, me quedo bloqueado.
(es decir me quedo esperando hasta que pueda comer)
Debo hacerlo después de la SC de exmutua para no bloquear otros procesos.
Cdo me liberen (otro entra en deja_palillos) es xq hay palillos libres y pasaría a comer

Ojo!! Estas líneas no se pueden cambiar de orden, ya que si lo hicieran, el proceso se quedaría colgado sin ejecutar la sentencia que libera el semáforo de la Ex. Mutua

```
void deja_palillos(int i)
{
  down (&mutex);
  estado[i]= PENSANDO;
  intento(IZQD);
  intento(DCHA);
  up (&mutex);
}
```

Intento liberar a mis vecinos de izquierda y de la derecha, dándo una nueva oportunidad de ejecut método intento, teniendo en cue que no soy el único vecino.

```
void intento (int i)
{
  if(estado[i] ==  HAMBRIENTO &&
       estado[IZQD] !=  COMIENDO &&
       estado[DCHA] !=  COMIENDO) {
    estado[i] = COMIENDO;
    up(&S[i]);
  }
}
```

Si estoy sin comer (hambriento) y ni a mi izquierda ni a mi derecha nadie come, me pongo a comer

Nota: Suponemos que trabajamos con semáforos binarios robustos.

Cena de Filósofos.
Solución mediante monitores en C++

- En esta solución se juega con el array “numPalillos” (iniciado a 2), que permite solo comer a los filósofos que lo mantengan (el valor 2) después de efectuar el método “cogerPalillos”.
  - Este método resta en uno el valor de los contrincantes(izda y dcha) impidiendo que puedan comer si lo desean hasta que el que evitó el bloqueo les desbloquee a través del método “dejarPalillos”.

- Notas:
  - En C++ las variables de condición son estructuras (como clases) que admiten 2 métodos: wait() y signal(), para bloquear/desbloquear a un hilo de su sección crítiva
  - Se omite el constructor que inicia el array numPalillos, en todas sus posiciones a 2.)

Cena de Filósofos.
Solución mediante monitores en C++ (II)

```
Monitor CtrlPalillos {

        public cogerPalillos, dejarPalillos;

        private
                int numPalillos[5];
                condition dormir[5];

        void cogerPalillos(int filosofo) {
                if (numPalillos[filosofo] != 2) dormir[filosofo].delay();
                // Quita un palillo al filósofo de la izquierda
                numPalillos[ (filofoso+5-1)%5 ]--;
                // Quita un palillo al filósofo de la derecha
                numPalillos[ (filofoso+1)%5 ]--;
        }
```

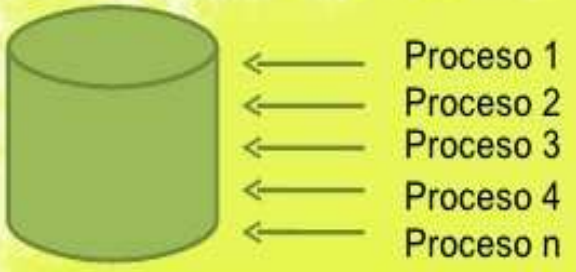
Cena de Filósofos.
Solución mediante monitores en C++ (III)

```
void dejarPalillos(int filosofo) {
        // Deja un palillo para el filósofo de la izquierda
        numPalillos[ (filofoso+5-1)%5 ]++;
        // Deja un palillo para el filósofo de la derecha
        numPalillos[ (filofoso+1)%5 ]++;
        // Despierta al filósofo de la izquierda, si tiene sus dos palillos
        if (numPalillos[(filofoso+5-1)%5]==2)
                dormir[(filofoso+5-1)%5].resume();
        // Despierta al filósofo de la derecha, si tiene sus dos palillos
        if (numPalillos[(filofoso+1)%5 ]==2)
                dormir[(filofoso+1)%5].resume();
}
```

Esta solución es óptima, ya que

- *gracias al monitor aseguramos la Ex. Mutua, y*
- *gracias a los métodos y el array numPalillos, permitimos comer a varios filósofos de forma concurrente sin provocar inanición ni interbloqueo.*

Cena de Filósofos.
Ejemplo de uso en la "vida real"



Base De Datos

FILOSOFOS COMENSALES

Suponemos n=50 (procesos/Hilos)

Enunciado Del Problema:

- El número de cada proceso representa su ID y además el numero de bloqueos que el proceso utiliza de la DB.
- Pueden haber varios procesos usando la DB pero la suma de los bloques utilizado no debe ser mayor a 100.

Cena de Filósofos.
Ejemplo de uso en la "vida real"
Solución mediante semáforos I

```
Filosofos Comensales:
Variables:
      Status[1..N] = 'P' <--- Hace_Cosas()
                     'H' <--- Quiere_usarBD()
                     'C' <--- Usa_BD()
Suma: Integer <--- La suma de bloques utilizados.
Semaforos:
      Exc_Mut <-- Recursos Compartidos
      Sem[1..N] <--  Semaforo X Proceso
Inicializacion:
    suma=0; Is(Exc_Mut,1);
    For i=1 to N{
          status[i]='P'
          Is(Semaforo;0);
    }
```

Cena de Filósofos.
Ejemplo de uso en la "vida real"
Solución mediante semáforos II

```
Proceso(Integer i){
Repeat
      Hace_cosas();
      Solicita_BD(i);
            Usa_BD();
      Devuelve_BDD();
      Hace_MasCosas();
}

Solicita_BDD( Integer i){
   Down(Exc_Mut);
  Status[i]='h';
  test(i);
  up(Exc_Mut);
  Down(              );
}
```

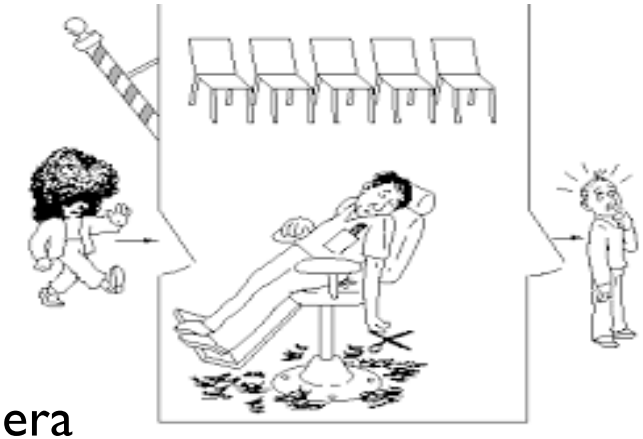
Este número representa el PID y el de transacciones que el proceso solicita sobre la BDD.

```
Devuelve_BDD(Integer i){
     Down(Exc_Mut);
     status[i]='P';
     suma=suma-i;
     for j=     down to 1
              test(j);
     up(Exc_mut);
}
Test(Integer i){
If(suma+i<=100 and status[i]='h')
       suma=suma+i;
       status[i]='c';
        up(Sem[i]);
}
```

Al liberar mi espacio intento que todos los demás procesos bloqueados (estado = 'h') puedan usar la BDD (Q pasen a estado 'c')

3. Problema del barbero dormilón

Enunciado



- Una peluquería tiene
  - un barbero,
  - una silla de peluquero y
  - N sillas para que se sienten los clientes en espera
- Si no hay clientes presentes el barbero se sienta y se duerme.
- Cuando llega un cliente, este debe despertar al barbero dormilón.
- Si llegan mas clientes mientras el barbero corta el cabello de un cliente, ellos se sientan (si hay sillas desocupadas), o, en caso contrario, salen de la peluquería.
- El problema consiste en programar al barbero y los clientes (procesos) sin entrar en condiciones de competencia.

Problema del barbero dormilón

Solución general

- Cuando el barbero abre su negocio por la mañana ejecuta el procedimiento "barbero()", lo que establece un bloqueo en el semáforo "clientes", hasta que alguien llega; después se va a dormir (a esperar ;).
- Cuando llega el primer cliente, éste ejecuta el método "cliente()"
  - Se verifica entonces si el número de clientes que esperan es menor que el número de sillas.
  - Si esto ocurre (es decir, si existe una silla disponible)
    - el cliente incrementa una variable contadora.
    - Luego realiza un "up" en el semáforo "customer" con lo que se despierta al barbero.
  - Si-no:
    - sale sin su corte de pelo
- Notas:
  - Este problema puede tener variantes que lo hagan más complicado, como por ejemplo que haya más de una silla de barbero para cortar el pelo.
  - Se puede resolver de diferentes maneras con las diferentes técnicas vistas, así como con cualquier lenguaje concurrente (ver los pdfs anexos).

Problema del barbero dormilón

Solución con semáforos (Pseudocódigo) I

```
Semáforo barberoListo = 0;        // (Mutex, sólo 1 o 0)
Semáforo sillasAccesibles = 1;    // (Mutex solo 0 o 1)
                                  Cuando sea 1, el número de sillas libres puede aumentar o disminuir
Semáforo clientes = 0;            // Número de clientes en la sala de espera

int sillasLibres = N              // Número total de sillas
```

Función barbero (Proceso/hilo-thread):

```
while(true)                       // Ciclo infinito
{
   wait(clientes);                // Espera la señal de un hilo cliente para despertar.

   wait(sillasAccesibles);        // (Ya está despierto) Espera señal para poder modificar sillasLibres.
   sillasLibres += 1;             // Aumenta en uno el número de sillas libres.

   signal(barberoListo);          // El barbero está listo para cortar y manda señal al hilo cliente.
   signal(sillasAccesibles);      // Manda señal para desbloquear el acceso a sillasLibres
                                  // Aquí el barbero corta el pelo de un cliente (zona de código no crítico).
}
```

Problema del barbero dormilón

Solución con semáforos (Pseudocódigo) II

Función cliente (Proceso/hilo-thread):

```
wait(sillasAccesibles);       // Espera la señal para poder acceder a sillasLibres.

if (sillasLibres > 0)         // Si hay alguna silla libre, se sienta en una.
{
   sillasLibres -= 1;         // Decrementando el valor de sillasLibres en 1.

   signal(clientes);          // Manda señal al barbero de que hay un cliente disponible.
   signal(sillasAccesibles);  // Manda señal para desbloquear el acceso a sillasLibres.
   wait(barberoListo);        // El cliente espera a que el barbero esté listo para atenderlo.

  <<Se le corta el pelo al cliente>>
}
else   // Si no hay sillas libres.
{
   signal(sillasAccesibles);  // Manda señal para desbloquear el acceso a sillasLibres.
                              // El cliente se va de la barbería y no manda la señal de cliente disponible.
}
```

4.Problema de los lectores/escritores

Definición

- Problema:
  - Acceso a recursos compartidos de lectura/escritura.
- Propiedades:
  - Cualquier número de lectores puede leer un archivo simultáneamente.
  - Sólo puede escribir en el archivo un escritor en cada instante.
  - Si un escritor está accediendo al archivo, ningún lector puede leerlo.
- Problema típico:
  - PSs que acceden a una base de datos(BBDD):
    - lectores: consultan la BBDD
    - escritores: consultan y modifican la BBDD
  - Para mantener la consistencia de la BBDD
    - cuando un escritor accede a la BBDD, es el único proceso que la puede usar
    - varios lectores pueden acceder simultáneamente

Problema de los lectores/escritores
Solución con semáforos (Sin Prioridad)

Compartido

```
typedef int semaforo;
int contlect = 0; /* contador de lectores */
semaforo mutex=1; /*controla el acceso a contador de lectura */
semaforo esem=1; /* controla el acceso de escritura */
```

```
void lector(void){
  while (1) {
     down (mutex);
     contlect = contlect + 1;
     if(contlect ==1) down (esem);
     up (mutex);

     lee_recurso();

     down (mutex);
     contlect = contlect - 1;
     if(contlect ==0) up (esem);
     up (mutex);
  }
}
```

```
void escritor(void)
{
   while (1) {
      down (esem);
      escribe_en_recurso();
      up (esem);
   }
}
```

Problema de los lectores/escritores

Solución con semáforos (Prioridad a los escritores)

Compartido

```
typedef int  semaforo;
int contlect = 0;          /* contador de lectores */
int contesc = 0;           /* contador de escritores */
semaforo mutex1=1;         /* controla el acceso a contlec */
semaforo mutex2=1;         /* controla el acceso a contesc */
semaforo mutex3=1;         /* controla el acceso al semáforo lsem */
semaforo esem=1;           /* controla el acceso de escritura */
semaforo lsem=1;           /* controla el acceso de lectura */
```

Lector

```
void lector(void)
{
  while (1) {
    down (mutex3);
    down (lsem);
    down (mutex1);
    contlect = contlect + 1;
    if(contlect ==1) down (esem);
    up (mutex1);
    up (lsem);
    up (mutex3);
    lee_recurso();
    down (mutex1);
    contlect = contlect - 1;
    if(contlect ==0) up (esem);
    up (mutex1);
  }
}
```

Escritor

```
void escritura(void)
{
  while (1) {
    down (mutex2);
    contesc = contesc + 1;
    if(contesc ==1) down (lsem);
    up (mutex2);
    down(esem);
    escribe_en_recurso();
    up (esem);
    down (mutex2);
    contesc = contesc - 1;
    if(contesc ==0) up (lsem);
    up (mutex2);
  }
}
```

Problema de los lectores/escritores

Solución con monitores

Controlador

```
void controlador(void)
{
  while (1) {
    if (cont>0) {
      if (!vacio(terminado)) {
        receive (terminado, msj);
        cont++;
      } else if (!vacio(pedir_escritura)) {
        receive (pedir_escritura, msj);
        escritor_id = msj.id;
        cont = cont-100;
      } else if (!vacio(pedir_lectura)) {
        receive (pedir_lectura, msj);
        cont--;
        send(buzon[msj.id],"OK");
      }
    }
    if (cont==0) {
      send (buzon[escritor_id] , "OK");
      receive (terminado, msj);
      cont==100;
    }
    while(cont<0) {
      receive (terminado, msjl);
      cont++;
    }
  } /* while(1) */
}  /* fin función controlador */
```

Lector

```
void lector(int i)
{
  mensaje msjl;
  while (1) {
    msjl=i;
    send (pedir_lectura, msjl);
    receive (buzon[i], msjl);
    lee_unidad();
    msjl=i;
    send (terminado, msjl);
  }
}
```

Escritor

```
void escritura(int j)
{
  mensaje msje;
  while (1) {
    msje=i;
    send (pedir_escritura, msjl);
    receive (buzon[j], msje);
    escribe_unidad();
    msje=j;
    send (terminado, msje);
  }
}
```