SINCRONIZACIÓN DE PROCESOS

Introducción

- Los procesos acceden a datos o recursos compartidos

- El acceso puede provocar que el estado final de los datos no sea correcto, generando incoherencias debido a manipulaciones concurrentes

- El acceso debe ser controlado, deben emplearse herramientas que garanticen la exclusión mutua

- Existen herramientas que permiten esto, pero tanto el sistema operativo como el hardware deben ofrecer características que soporten las mismas

Instrucción Test-and-Set

- Es una instrucción empleada para testear y escribir en un lugar en memoria de manera atómica (sin interrupción). Lectura y escritura sin interrupciones.

- El testeo puede ser una comparación frente a un valor por ejemplo.
  El valor es dependiente del resultado del test

- Un procesador realizando una instrucción Test-and-Set sobre una dirección de memoria impide que cualquier otro procesador pueda realizarla cerrando el bus de memoria

- Con esta instrucción se facilita la creación de semáforos y otras herramientas de sincronización

- Enlace
  `http://en.wikipedia.org/wiki/Test-and-set`

Semáforos

- Introducidos por Edsger Dijkstra en 1968
- Herramienta de sincronización que brinda una solución al problema de la exclusión mutua restringiendo el acceso simultáneo a los recursos compartidos
- Permiten resolver la mayoría de los problemas de sincronización entre procesos y forma parte del diseño de muchos sistemas operativos y de lenguajes de programación concurrentes
- Si bien son útiles para sincronizar procesos, no pueden evitar deadlocks
- El valor de un semáforo representa la cantidad de instancias libres de un recurso determinado

Semáforos

- Los semáforos consisten en:
  - una variable que puede tomar valores en un rango acotado de enteros positivos
  - una cola de procesos que están en espera del recurso
  - un conjunto de funciones permitidas

- Operaciones

```
P(Semaforo s) { // Adquirir Recurso
  wait until s > 0, then s ← s-1;
                /* Testear y decrementar s, debe ser una operación atómica */
}

V(Semaforo s) { // Liberar Recurso
  s ← s+1;       /* debe ser atómico */
}

Init(Semaforo s, Valor v) {
  s ← v;
}
```

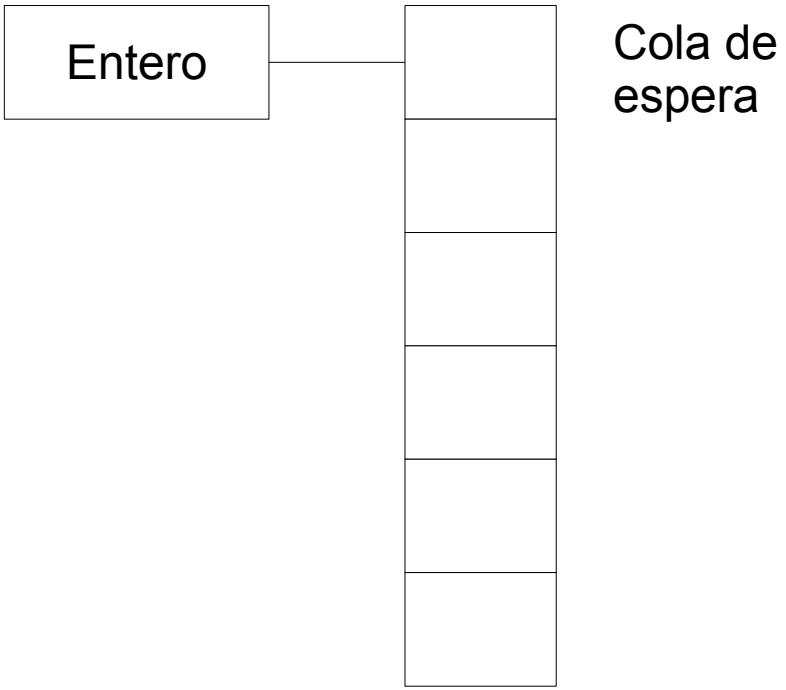
P = down = wait = acquire

V = up = signal = release

- La atomicidad de las operaciones se obtiene mediante el uso de instrucciones tipo *test-and-set* o desactivación de interrupciones, dicha atomicidad es fundamental para evitar las condiciones de competencia

Semáforos

- Para evitar el busy-waiting (http://en.wikipedia.org/wiki/Busy_waiting) un semáforo tiene asociada una cola de procesos



Semáforos binarios

- Actúan como indicador de condición registrando si un recurso está o no disponible
- Son denominados ***mutex*** cuando se inician en 1 pues su función principal es la de garantizar la exclusión mutua
- Un semáforo binario sólo puede tomar dos valores: 0 y 1
  Si, para un semáforo binario S, S = 1 entonces el recurso está disponible y el proceso lo puede utilizar; si S = 0 el recurso no está disponible y el proceso debe esperar
- Si un proceso hace un *down* sobre un semáforo binario que vale:
  1 – decrementará de manera atómica el valor del semáforo dejándolo en 0
  0 – provocará que el proceso invocante quede "dormido" en el semáforo
- Si un proceso hace un *up* sobre un semáforo que vale:
  1 – el semáforo permanecerá con el valor 1 y el proceso seguirá su ejecución
  0 – el semáforo se incrementa sólo si *no* hay procesos "dormidos" en el"
  si tiene procesos dormidos continuará en 0 y se despertará a uno de dichos procesos

Semáforos

- Uso de semáforos para obtener exclusión mutua

```
semaforo mutex = 1
```

```
Proceso P1() {
    down(mutex);
    /* RC */
    up(mutex);
}
```

```
Proceso P2() {
    down(mutex);
    /* RC */
    up(mutex);
}
```

- Uso de semáforos para sincronización de procesos

"P2 debe ejecutarse luego de P1"

```
semaforo s = 0
```

```
Proceso P1() {
    ...
    /* RC */
    up(s);
}
```

```
Proceso P2() {
    down(s);
    /* RC */
    ...
}
```

- **Semáforos n-arios o de conteo o generales**

• Útiles para representar *n* instancias de un tipo de recurso o para permitir el acceso a una región critica a *n* procesos

• Toman valores de 0 a *n*

• Su comportamiento es similar a los de los semáforos binarios

- **Semáforos n-arios implementado con semáforos binarios**

```
SemaforoGeneral {
    int valor = K
    SemaforoBinario wait = 0
    SemaforoBinario mutex = 1
}

down(SemaforoGeneral S) {
    down(S.mutex)
    if (S.valor <= 0) {
        S.valor = S.valor - 1;
        up(S.mutex);
        down(S.wait);
    } else {
        S.valor = S.valor – 1;
        up(S.mutex);
    }
}
```

```
up(SemaforoGeneral S) {
    down(S.mutex)
    if (S.valor < 0) {
        up(S.wait);
    }
    S.valor = S.valor + 1;
    up(S.mutex);
    }
}
```

Fumadores

- Para poder confeccionar un cigarro y fumarlo se precisan 3 ingredientes:
  - Tabaco - Papel - Fósforo
- Hay 3 fumadores deseosos de fumar, cada uno de ellos tiene una cantidad ilimitada de uno de estos ingredientes
- Existe un "arbitro" que permite a los fumadores hacer sus cigarros seleccionando dos fumadores para que coloquen un item cada uno sobre la mesa, luego notifica al tercero para que advierta que los dos componentes faltantes están disponibles. El proceso advertido toma los ítems, arma un cigarro y fuma....
  El arbitro ve la mesa vacía y nuevamente elige dos fumadores...

```
semaforo fumador[3] ={0,0,0};
semaforo mesa = 1;
```

Arbitro

```
while (1) {
  down(mesa);
  //elegir fumadores i y j
  //para que pongan items
  //hacer que k fume
  up(fumador[k]);
}
```

Fumador(i)

```
while true {
    down(fumador[i]);
    //hacer un cigarro
    up(mesa);
    //fumar el cigarro
}
```

Buffer acotado - Productor/Consumidor

• Un proceso, Productor, coloca items en un buffer de determinado tamaño (N)
Cuando el buffer esté completamente lleno "se va a dormir"

• El Consumidor extrae los items, cuando el buffer está vacío "se va a dormir"

`count` = número de items existentes en el buffer

- El Productor chequea `¿count==N?`
  SI → dormir
  NO → `count = count + 1`

- El Consumidor chequea `¿count==0?`
  SI → dormir
  NO → `count = count - 1`

Buffer acotado - Productor/Consumidor

• Cada proceso verifica si el otro proceso debe o no continuar durmiendo

```
#define N 10
int count = 0;

void productor() {
    int item;

    while(1) {
       item = producir_item();
       if (count == N)
          sleep();
       poner_item(item);
       count = count + 1;
       if (count == 1)
          wakeup(consumidor);
    }
}
```

```
void consumidor() {
    int item;

    while(1) {
       if (count == 0)
          sleep();
       item = sacar_item(item);
       count = count - 1;

       if (count == N-1)
          wakeup(productor);
    }
}
```

Buffer acotado - Productor/Consumidor

• Los semáforos resuelven el problema del “despertar perdido”

• Se utilizan tres semáforos
- **full**: cuenta el número de entradas ocupadas en el buffer
- **empty**: cuenta el número de entradas vacías en el buffer
- **mutex**: para garantizar la exclusión mutua

```
#define N 10
semaforo mutex = 1
semaforo empty = N
semaforo full = 0
```

```
void productor() {
    int item;

    while(1) {
       item = producir_item();
       down(empty);
       down(mutex);
       poner_item(item);
       up(mutex);
       up(full);
    }
}
```

```
void consumidor() {
    int item;

    while(1) {
       down(full);
       down(mutex);
       item = sacar_item(item);
       up(mutex);
       up(empty);
    }
}
```

Lectores y Escritores

- Modela el acceso a una base de datos
- Muchos procesos en competencia intentan leer y escribir en ella
- Pueden existir muchos lectores simultáneos
- Cuando actúa un escritor no pueden estar presentes otros lectores ni escritores

```
semaforo mutex = 1;
semaforo db = 1;
int rc = 0;


 void escritor() {
    while (1) {
       ...
       down(db);
       escribir_db();
       up(db);
    }
 }
```

```
void lector() {
   while (1) {
      down(mutex);
      rc = rc + 1;
      if (rc == 1)
         down(db);
      up(mutex);
      leer_db();
      down(mutex);
      rc = rc – 1;
      if (rc == 0)
         up(db);
      up(mutex);
   }
}
```

Prioridad para los lectores

Barbero Dormilón

- 1 barbero
- 1 silla de peluquero
- N sillas
- si no hay clientes el barbero se duerme en la silla
- cuando llega el primer cliente debe despertar al barbero
- si llegan más cliente mientras el barbero corta el cabello de un cliente se sientan en la silla (si las hay) o salen de la peluquería

```
#define SILLAS 5
semaforo clientes = 0   /* Numero de clientes en espera */
semaforo barbero = 0
semaforo mutex = 1
int espera = 0

void barbero() {
    while (1) {
        down(clientes);
        down(mutex);
        espera = espera – 1;
        up(barbero);
        up(mutex);
        cortar_cabello();
    }
}
```

Si no hay clientes se duerme

Barbero listo para trabajar

```
void cliente() {
    down(mutex);
    if (espera < SILLAS) {
        espera = espera + 1
        up(clientes);
        up(mutex);
        down(barbero);
        tomar_corte();
    } else {
        up(mutex);
    }
}
```

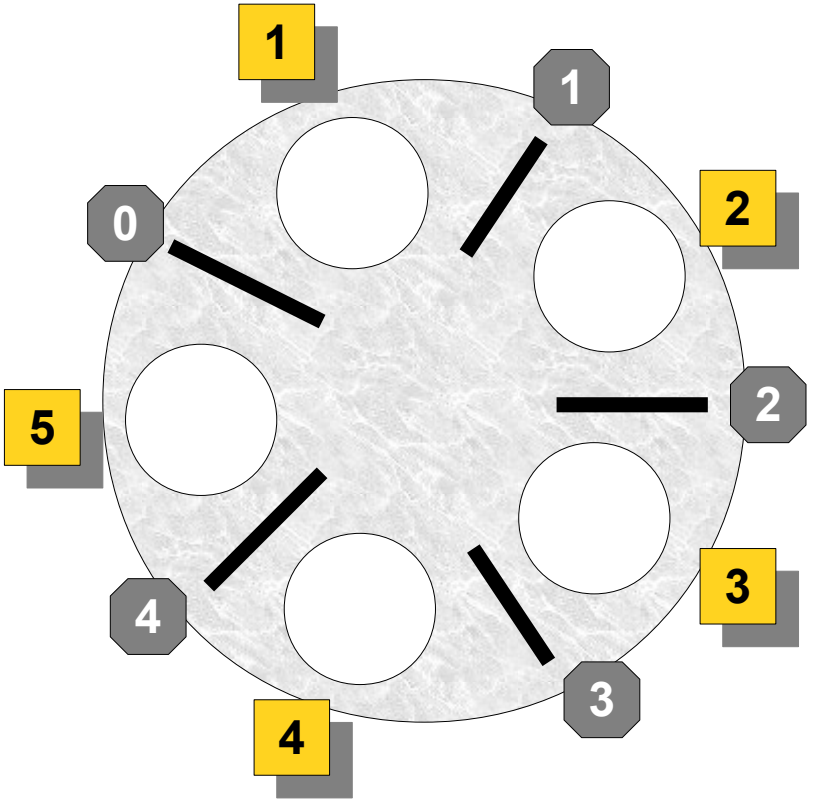
Despierta al barbero

Si el barbero está ocupado espera

Si no hay sillas libres se retira sin tomar servicio

Cena de los filósofos

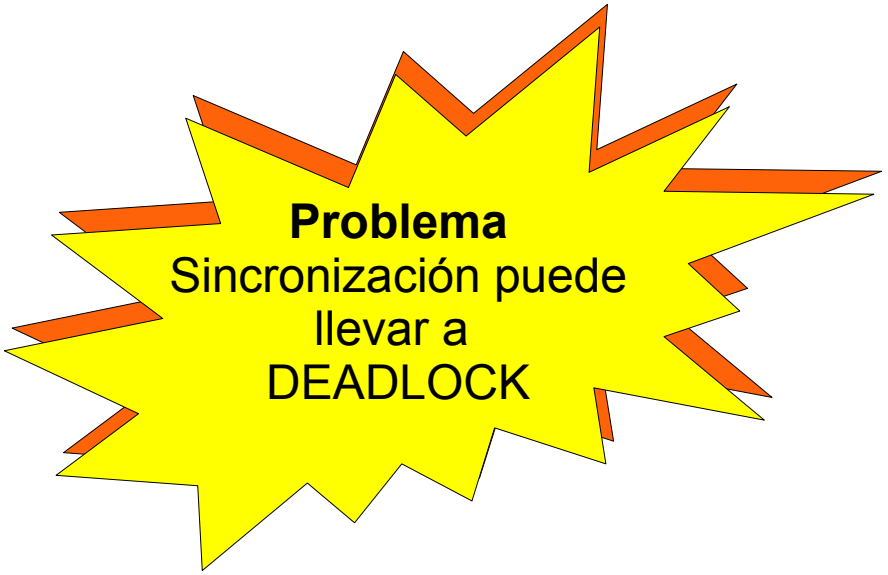
- Típico problema donde un conjunto de procesos compiten por recursos compartidos
- Definido por Dijkstra en 1965, Hoare propuso la fantasía
- 5 filósofos = 5 procesos (`N=5`)
- Cada filósofo precisa 2 palillos (recursos) para comer (ejecutarse) arroz



i	IZQ i%N	DER (i-1)%N
1	1	0
2	2	1
3	3	2
4	4	3
5	0	4

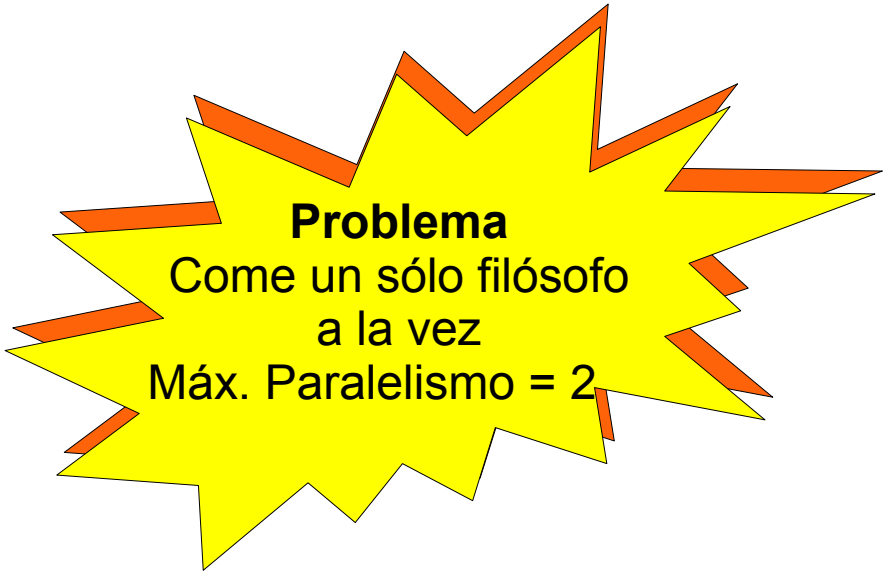
Cena de los filósofos: Solución 1

```
#define N 5
#define IZQ(i) i%N
#define DER(i) (i-1)%N

void filosofo (int i) {
    while (1) {
        pensar();
        tomar_palillo(IZQ(i));
        tomar_palillo(DER(i));
        comer();
        soltar_palillo(IZQ(i));
        soltar_palillo(DER(i));
    }
}
```



Cena de los filósofos: Solución 2 (con semáforos)

```
#define N 5
#define IZQ(i) i%N
#define DER(i) (i-1)%N
semaforo mutex = 1

void filosofo (int i) {
    while (1) {
        pensar();
        down(mutex);
        tomar_palillo(IZQ(i));
        tomar_palillo(DER(i));
        comer();
        soltar_palillo(IZQ(i));
        soltar_palillo(DER(i));
        up(mutex);
    }
}
```



Cena de los filósofos: Solución 3

```
#define N 5
#define IZQ i%N
#define DER (i-1)%N
#define PENSAR 0
#define APETITO 1
#define COMER 2
semaforo mutex=1;
int estado[N];
semaforo s[N]; //todos en cero

void filosofo(int i) {
    while (1) {
        pensar();
        tomar_palillo(i);
        comer();
        poner_palillo(i);
    }
}

void tomar_palillo(int i) {
    down(mutex);
    estado[i] = APETITO;
    test(i);
    up(mutex);
    down(s[i]);
}

void poner_palillo(int i) {
    down(mutex);
    estado[i] = PENSAR;
    test(IZQ(i));
    test(DER(i));
    up(mutex);
}

void test(int i) {
   if (estado[i]==APETITO && estado[IZQ(i)]!=COMER && estado[DER(i)]!=COMER) {
        estado[i] = COMER;
        up(s[i]);
   }
}
```

Monitores

- Se debe prestar especial atención al usar semáforos

```
void barbero() {
    while (1) {
        down(clientes);
        down(mutex);
        espera = espera – 1;
        up(barbero);
        up(mutex);
        cortar_cabello();
    }
}
```

```
void barbero() {
    while (1) {
        down(mutex);
        down(clientes);
        espera = espera – 1;
        up(barbero);
        up(mutex);
        cortar_cabello();
    }
}
```

- Esta primitiva de alto nivel fue introducida por Hoare en 1974 – Hansen 1975
- Con monitores la exclusión muta está garantizada
- Un monitor es una colección de procedimientos, variables y estructuras de datos
- Los procesos pueden invocar a los procedimientos de un monitor cuando lo deseen, pero no tienen acceso directo a las estructuras de datos internas

Monitores

- Sintaxis modelo

```
monitor modelo {
    int i;          //variable local del monitor
    condition c;    //variable de condición

    procedimiento P1() { ...
    }

    procedimiento P2() { ...
    }

    procedimiento P3() { ...
    }

    //inicialización de variables
}
```

- Sólo un proceso puede estar activo dentro del monitor
- Si el monitor está ocupado y un segundo proceso intenta ingresar, éste se bloquea
- Los monitores son construcciones de los lenguajes de programación

Monitores

- Generalmente el compilador del lenguaje que soporta monitores implementa la exclusión mutua mediante semáforos
- Usando monitores, basta con transformar las secciones críticas en procedimientos de monitor con el fin de lograr la exclusión mutua
- *¿Cómo se realiza el bloqueo de procesos?*
  Mediante variables de condición + operaciones WAIT y SIGNAL
  - si un proceso de monitor no puede/debe continuar ejecuta un WAIT en alguna variable de condición
  - el proceso se bloquea y permite que otro proceso ingrese al monitor
  - el nuevo proceso despertará al proceso dormido mediante SIGNAL sobre la misma variable de condición
- Un SIGNAL sobre una variable que no bloquea ningún proceso se pierde, no se mantiene historia
- *¿Así no existen dos procesos activos en el monitor?*
  Hoare: suspender al proceso que hace SIGNAL y reanudar al proceso antes dormido
  Hansen: el proceso que hace SIGNAL debe salir del monitor inmediatamente

Monitores

```
monitor ProductorConsumidor {
    int count;
    condition lleno, vacio;

    procedimiento poner() {
        if (count == 10)
            wait(lleno)
        poner_item()
        count = count + 1
        if (count == 1)
            signal(vacio)
    }

    count = 0;
}
```

```
procedimiento sacar() {
    if (count == 0)
        wait(vacio)
    sacar_item()
    count = count - 1
    if (count == 9)
        signal(lleno)
}
```

```
procedimiento Productor() {
    while(1) {
        producir_item()
        ProductorConsumidor.poner
    }
```

```
procedimiento Consumidor() {
    while(1) {
        producir_item()
        ProductorConsumidor.sacar
    }
```

Monitores: Desventajas

- Los monitores son un concepto a nivel lenguaje de programación

- Muchos lenguajes de programación no poseen monitores