6 SINCRONIZACIÓN DE PROCESOS

1 ANTECEDENTES

2 PROBLEMA DE LA SECCIÓN CRÍTICA

3 HARDWARE DE SINCRONIZACIÓN

4 SEMÁFOROS

5 PROBLEMAS CLÁSICOS DE SINCRONIZACIÓN

6 REGIONES CRÍTICAS

7 MONITORES

9 TRANSACCIONES ATÓMICAS

6.1 ANTECEDENTES

- El acceso concurrente a datos compartidos puede tener como resultado la inconsistencia de datos.
- La mantención de la consistencia de los datos requiere mecanismos que aseguren la ejecución ordenada de procesos cooperativos, que comparten un espacio de direcciones lógicas.

6.2 PROBLEMA DE LA SECCIÓN CRÍTICA

- N procesos compitiendo por usar datos compartidos.
- Cada proceso P_i (i : 0 .. n - 1) tiene un segmento de código, llamado *sección crítica*, en el cual son accesados los datos compartidos.
- Problema: si un proceso está ejecutando su sección crítica, asegurar que ningún otro ejecute la propia.
- Estructura del proceso P_i

repeat
sección de entrada
sección crítica
sección de salida
sección restante
until *falso*;

6.2 PROBLEMA DE LA SECCIÓN CRÍTICA

La solución pasa por satisfacer 3 requisitos:

1. **Exclusión Mutua**. Si P_i está ejecutando su sección crítica, ningún otro puede ejecutar la propia.

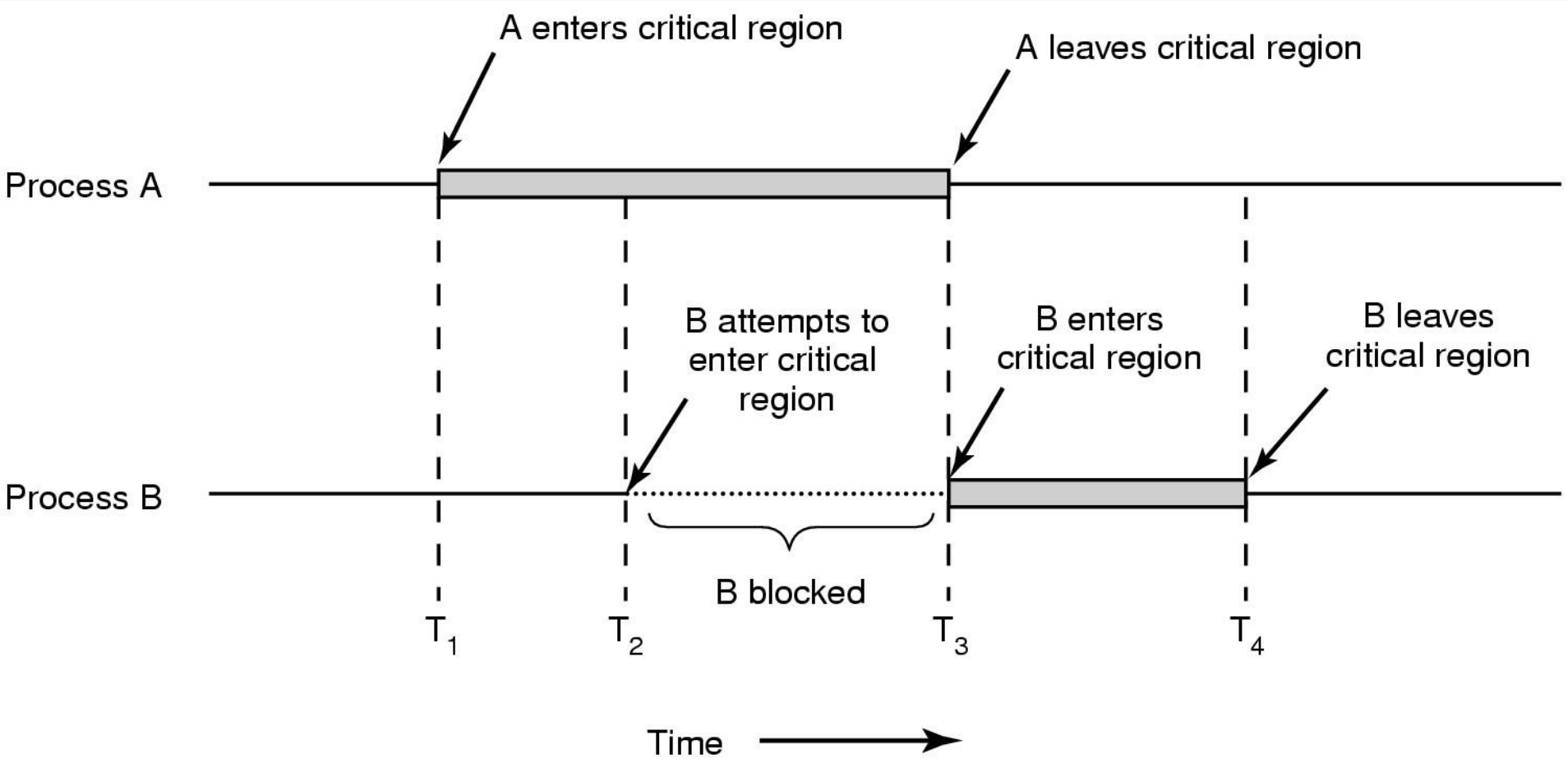
2. **Progreso**. Si ningún proceso está ejecutando su sección crítica y hay alguno que quiere entrar en ella, sólo los procesos que no están en la **sección restante** pueden participar de la decisión de qué proceso entrará próximamente y esta selección no puede ser pospuesta indefinidamente.

6.2 PROBLEMA DE LA SECCIÓN CRÍTICA

3. **Espera Limitada**. Un proceso no puede quedar esperando indefinidamente a entrar en su sección crítica.

Se asume que cada proceso se ejecuta a velocidad ≠ 0, pero sin suposiciones sobre la velocidad relativa de los procesos.

6.2 PROBLEMA DE LA SECCIÓN CRÍTICA
exclusión mutua usando regiones críticas



6.2.1 Soluciones para dos Procesos

- Sólo 2 procesos (P_0 y P_1) a la vez.
- Si un proceso es P_i , el otro es P_j (con j = 1 - i).
- Los procesos pueden compartir algunas variables comunes para sincronizar sus acciones.
- Se proponen 3 algoritmos:
  - ALGORITMO 1: Usando una variable entera global que representa el turno que le corresponde a cada proceso. No satisface el progreso. Supongamos turno = 0, P1 está listo para entrar a su sección crítica, pero no puede porque P0 todavía esta en la **sección restante.**

6.2.1 Soluciones para dos Procesos

- ALGORITMO 2: El Algoritmo 1 no retiene información sobre el estado interno de cada proceso. Se soluciona esto reemplazando la variable turno, por un arreglo boolean en que cada elemento representa si el proceso está listo para entrar a su sección crítica. No satisface el progreso.
  - *Si flag[i]== true; significa que Pi está listo para ingresar a su sección crítica. También es variable global.*
  - *Satisface la condición de exclusión mutua, pero no la de progreso.*

6.2.1 Soluciones para dos Procesos

- ALGORITMO 3: Combinando las ideas del Algoritmo 1 y 2 obtenemos una solución que cumple los tres criterios.
  - *Los procesos usan tres variables compartidas:*
  - *Bool flag[2]; // arreglo de dos booleanos*
  - *Int turno; // turno toma los valores 0 y 1*
  - *Inicialmente flag[0]=flag[1]=false y turno puede tomar cualquier valor (0 o 1)*

6.2.1 Soluciones para dos Procesos

- ALGORITMO 3:
- ¿Es correcto este algoritmo?
  - *¿se preserva la exclusión mutua?*
  - *Para que Pi entre a su sección crítica, se requiere que flag[j]=false o turno=i*
  - *Aún si ambos procesos entran porque flag[0]=flag[1]=true, ambos procesos no cumplen la condición del while por que turno está en 0 o 1, pero no ambos*
  - *Las condiciones 2 y 3 se preservan*

6.2.2 Soluciones para N Procesos

- Antes de entrar en su sección crítica, cada proceso recibe un número. Quien tenga el número menor entra en su sección crítica.
- Si los procesos P_i y P_j reciben el mismo número, si $i < j$, entonces P_i es atendido primero; sino P_j es atendido primero.
- El esquema de numeración genera siempre números en orden creciente; ej., 1, 2, 3, 3, 3, 3, 4, 5 ...

6.3 HARDWARE DE SINCRONIZACIÓN

- La solución planteada en el algoritmo 3 es posible generalizarla para múltiples procesos. Sin embargo es una solución de software.
- Una solución distinta y más eficiente es solicitar ayuda del hardware. Existen dos formas:
  - Deshabilitando el sistema de interrupciones al entrar a una sección crítica para evitar el context switch o cambio de contexto. (Es peligroso, ineficiente y difícil en ambiente de multiprocesadores)
  - A través de una instrucción especial que permite leer y modificar una dirección de memoria en forma atómica (TAS test and set).

6.4 SEMÁFOROS

- Para problemas de mayor complejidad, se usan herramientas más flexibles llamadas semáforos
- Es una variable entera. Las modificaciones a la variable se ejecutan en forma indivisible.
- Sólo puede ser accesado vía dos operaciones atómicas
  - *(espera) wait* (*S*) : **WHILE** $S \leq 0$ **DO** *nada*;
    $S := S - 1$;
  - *(señal) signal* (*S*) o Up(S): $S := S + 1$;
- Tipos de Semáforo:
  - Contador: valor entero con dominio no restringido.
  - Binario (0, 1): más simple de implementar.
- Se puede implementar un contador como binario.

6.4.1 Uso de semáforos

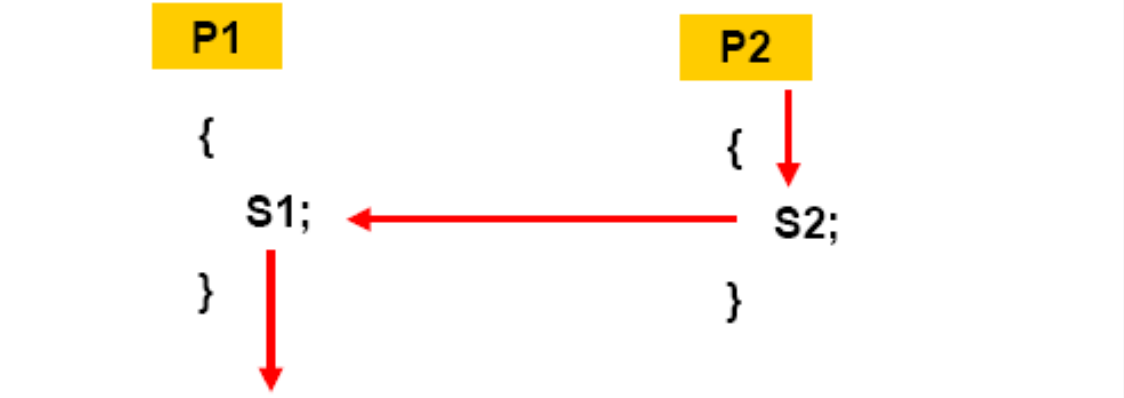
- Se puede resolver el problema de la sección crítica con n procesos, compartiendo un semáforo llamado mutex para manejar la exclusión mutua. Se inicializa en 1.

```
Do {
Wait (mutex);
    Sección crítica
Signal (mutex);
    Sección restante
```

- También se pueden resolver problemas de sincronización compartiendo un semáforo.

6.4.1 Uso de semáforos

- Además de resolver problemas de exclusión mutua de secciones críticas, se pueden resolver problemas de sincronización de procesos compartiendo un semáforo.
- Esto significa forzar un orden en su ejecución



Queremos obligar a que S1 se ejecute sólo después que S2 se haya ejecutado. ¿Cómo se puede hacer?

6.4.1 Uso de semáforos

Semaforo s =0;

- P1

{

Wait (s);

S1;

}

- P2

{

S2;

Signal (s);

}

6.4.2 Implementación

- Las soluciones de exclusión mutua requieren de *espera activa*, donde el proceso desarrolla un loop, lo cual genera overhead, pues se derrochan ciclos de CPU.
- Se puede solucionar con semáforos *spinlock (cerradura de giro)*, que no necesitan hacer cambio de contexto durante la espera. (A nivel de hilos de ejecución)
- Otra solución es la *espera inactiva*, donde el proceso se bloquea durante la espera, "despertando" al término de ésta.

6.4.3 Bloqueo Mutuo e Inanición

- Al implementar semáforos, se puede producir una situación donde dos o más procesos están esperando indefinidamente por un evento que puede ser causado sólo por uno de los procesos en espera. Esto se conoce como *bloqueo mutuo* o deadlock.
- También puede llevar a los procesos involucrados a un estado de *bloqueo indefinido* o *inanición*, donde un proceso puede no ser nunca removido de la cola en la cual se encuentra.

6.4.4 Semáforos Binarios

- Los semáforos descritos anteriormente son del tipo de conteo.
- Un semáforo *binario* es un valor entero que sólo puede variar entre 0 y 1.
- Puede ser más simple de implementar que el de conteo.
- Un semáforo de conteo puede ser implementado mediante semáforos binarios.

6.5 PROBLEMAS CLÁSICOS

1 Buffer Limitado

2 Lectores y Escritores

3 Filósofos Cenando

4 Barbero dormilón

6.5.1 Buffer Limitado

- En el contexto de los procesos cooperativos, se considera el problema de los productores y consumidores, donde hay procesos que producen información, la cual es consumida por otros procesos.
- Para que éstos puedan ejecutarse, debe existir un buffer que el productor pueda llenar y el consumidor pueda vaciar, el cual tiene capacidad fija.
- Por ello, el productor debe esperar si el buffer está lleno y el consumidor debe esperar si el buffer está vacío.

Productor_Consumidor

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );               /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

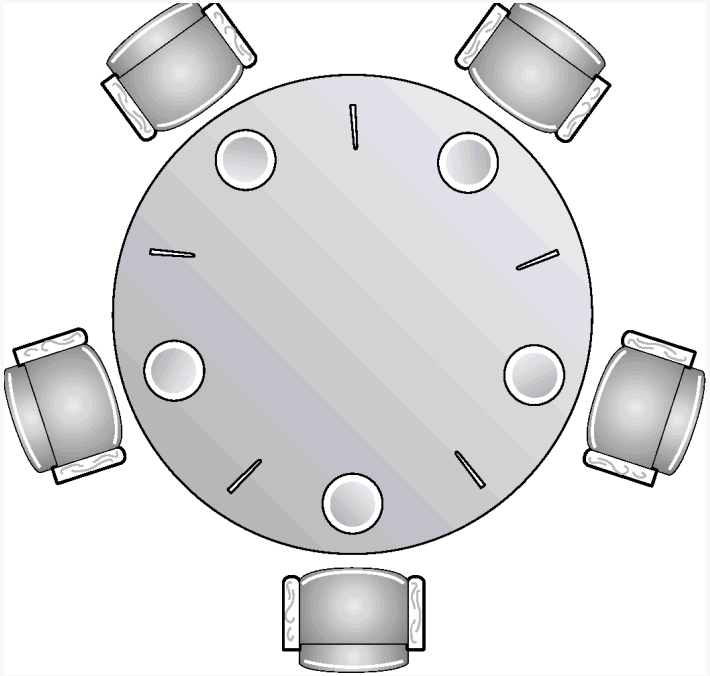
6.5.2 Lectores y Escritores

- Objeto de datos compartido entre varios procesos concurrentes, algunos sólo leerán (lectores) y otros actualizarán (escritores).
- La dificultad está en coordinar a los que actualizan, por lo que el acceso para actualización es exclusivo.
- La solución a este problema tiene variantes, en las que intervienen prioridades.
- También se puede provocar inanición.

6.5.2 Lectores y Escritores

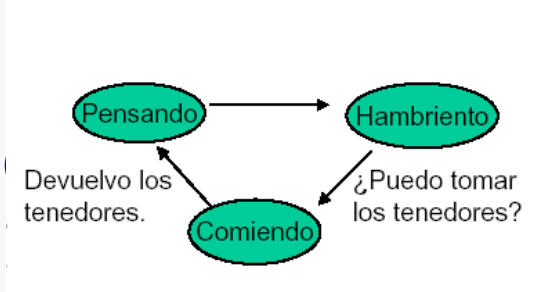
```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {                     /* repeat forever */
          down(&mutex);                 /* get exclusive access to 'rc' */
          rc = rc + 1;                  /* one reader more now */
          if (rc == 1) down(&db);       /* if this is the first reader ... */
          up(&mutex);                   /* release exclusive access to 'rc' */
          read_data_base( );            /* access the data */
          down(&mutex);                 /* get exclusive access to 'rc' */
          rc = rc – 1;                  /* one reader fewer now */
          if (rc == 0) up(&db);         /* if this is the last reader ... */
          up(&mutex);                   /* release exclusive access to 'rc' */
          use_data_read( );             /* noncritical region */
     }
}


void writer(void)
{
     while (TRUE) {                     /* repeat forever */
          think_up_data( );             /* noncritical region */
          down(&db);                    /* get exclusive access */
          write_data_base( );           /* update the data */
          up(&db);                      /* release exclusive access */
     }
}
```

6.5.3 Filósofos Cenando



- 5 filósofos están sentados en una mesa circular y su actividad consiste en pensar y comer.
- Al centro de la mesa hay un solo plato con comida y 5 tenedores ubicados entre éstos.
- Cuando un filósofo piensa, no actúa. Si tiene hambre, trata de tomar el tenedor más cercano (a derecha o izquierda), el cual puede estar ocupado, en ese caso, debe esperar. Cuando lo obtiene, come y luego devuelve el tenedor.

6.5.3 Filósofos Cenand



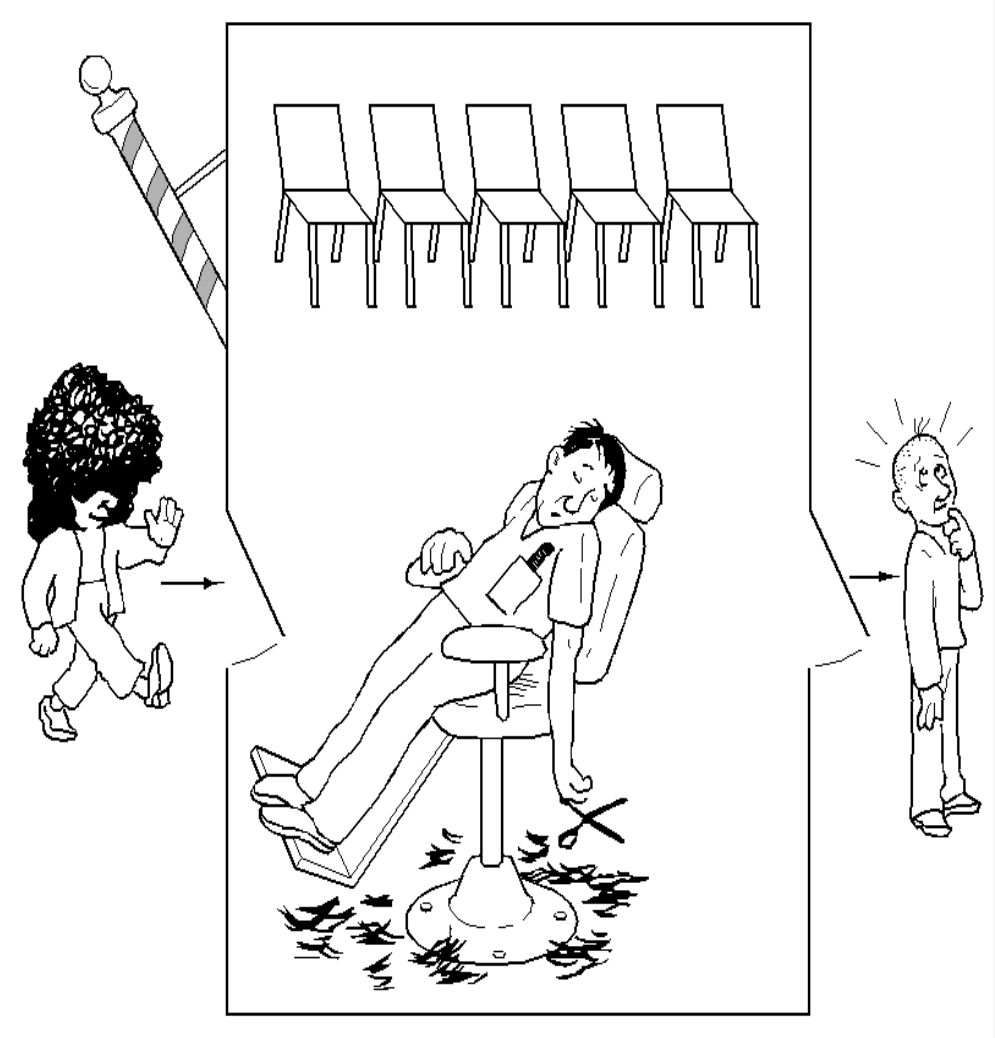
- Es un problema de sincronización clásico, donde se debe controlar la concurrencia, asignando recursos sin provocar bloqueo mutuo ni inanición.
- Una solución simple es representar cada tenedor mediante un semáforo, lo cual garantiza que dos filósofos no usarán simultáneamente un tenedor, pero puede provocar bloqueo mutuo.
- Se mejora restringiendo el número de filósofos que comen simultáneamente a 4 o que un filósofo sólo pueda comer si ambos tenedores están desocupados.
- Resolver el bloqueo mutuo no necesariamente resuelve la inanición.

6.5.3 Filósofos Cenando

```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher is trying to get forks */
#define EATING      2           /* philosopher is eating */
typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];                 /* one semaphore per philosopher */

void philosopher(int i)         /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {              /* repeat forever */
        think( );               /* philosopher is thinking */
        take_forks(i);          /* acquire two forks or block */
        eat( );                 /* yum-yum, spaghetti */
        put_forks(i);           /* put both forks back on table */
    }
}
```

```
void take_forks(int i)          /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = HUNGRY;          /* record fact that philosopher i is hungry */
    test(i);                    /* try to acquire 2 forks */
    up(&mutex);                 /* exit critical region */
    down(&s[i]);                /* block if forks were not acquired */
}

void put_forks(i)               /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = THINKING;        /* philosopher has finished eating */
    test(LEFT);                 /* see if left neighbor can now eat */
    test(RIGHT);                /* see if right neighbor can now eat */
    up(&mutex);                 /* exit critical region */
}

void test(i)                    /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

El Barbero Dormilón



El Barbero Dormilón

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting – 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair( );                /* cut hair (outside critical region) */
    }
}


void customer(void)
{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut( );             /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

6.6 REGIONES CRÍTICAS

- El uso de semáforos no siempre es exitoso para sincronizar procesos, generándose diversos problemas.
- Como mejora, se introduce el uso de un constructor de sincronización de alto nivel.
- El constructor utiliza una variable v compartida, que sólo puede ser accesada dentro de la región, de manera que mientras una sentencia de ésta está siendo ejecutada, ningún otro proceso puede accesar la variable v.

6.6 REGIONES CRÍTICAS

- Las regiones que hacen referencia a la misma variable compartida, se excluyen unas a otras en el tiempo.
- Cuando un proceso trata de ejecutar la región, evalúa una expresión B booleana. Si es verdadera, la sentencia es ejecutada, sino el proceso es postergado hasta que B sea verdadera y no haya otro proceso en la región asociada con v.

6.7 MONITORES

- Constructor de sincronización de alto nivel que permite la compartición segura de un tipo de dato abstracto entre procesos concurrentes.
- Para permitir que un proceso espere dentro del monitor, debe declararse una variable tipo *condition*.
- La variable sólo puede ser usada con *wait* y *signal*.
  - La operación *x.wait*; implica que el proceso invocador de esta operación está suspendido hasta que otro proceso invoque *x.signal*;
  - La operación *x.signal* resume exactamente un proceso suspendido. Si no hay, la operación no tiene efecto.

6.7 MONITORES

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

6.7 MONITORES – Productor/Consumidor

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count – 1;
                if count = N – 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

6.9 TRANSACCIONES ATÓMICAS

- Al trabajar con Sistemas de Bases de Datos, la mantención de la consistencia de los datos es crucial. Los Sistemas Operativos han tomado prestada esta capacidad, aplicándola a sus propios fines.
- Se define una **Transacción** como unidad operacional, la cual realiza una función lógica a través de un conjunto de instrucciones, el cual accede a los datos, con el propósito de leerlos/actualizarlos.
- El conjunto de instrucciones procede secuencialmente (serialmente) y termina con una instrucción especial de confirmación (commit) o descarte (rollback o abort), la cual valida/invalida a todo el conjunto.

6.9 TRANSACCIONES ATÓMICAS

- Esta serialización debe quedar registrada, de manera de permitir la recuperación, en caso de caídas.
- Para reforzar lo anterior, se introducen puntos de control (checkpoints), que son instancias intermedias de verificación.
- Para hacer más eficiente el sistema, se ejecutan las transacciones en forma concurrente, lo cual implica serializar las operaciones de las distintas transacciones, en forma intercalada.