Comunicación y sincronización entre procesos e hilos

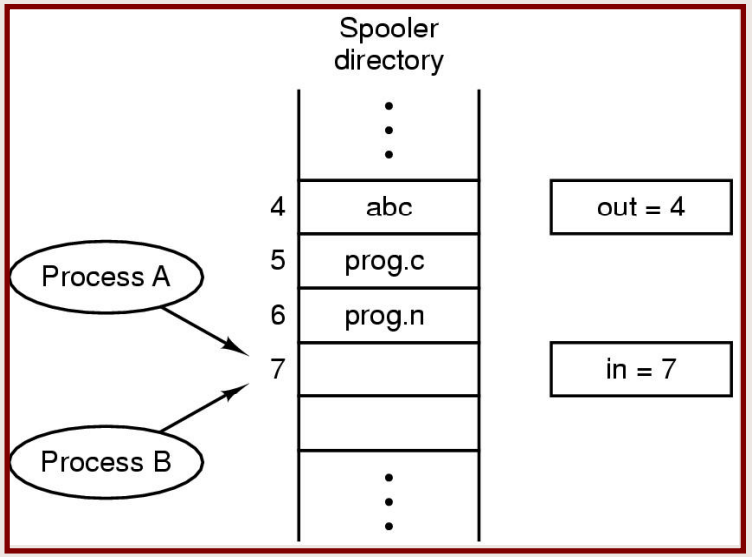
Comunicación entre procesos (IPC)

- Ejemplo: pipe (canalización en el shell)
- Cuestiones
  - Pasar información
  - Evitar interferencias
  - Garantizar orden correcto (dependencias)
- Igual para procesos que para hilos (la primera más sencilla para ellos)
- Multiprogramación

Sección 2.3

Condiciones de carrera (condiciones de competencia)

- Ejemplo: spooler de impresión
  - Directorio de spooler
  - Demonio de impresión



- La ley de Murphy
- Depurar programas con condiciones de carrera es complejo

Sección 2.3.1

Otro ejemplo de condición de carrera

```
int suma=0;
…
void suma(int ni, int nf)
   {
   int j;
   for (j=ni; j<=nf; j++)
        suma = suma + j; }
```

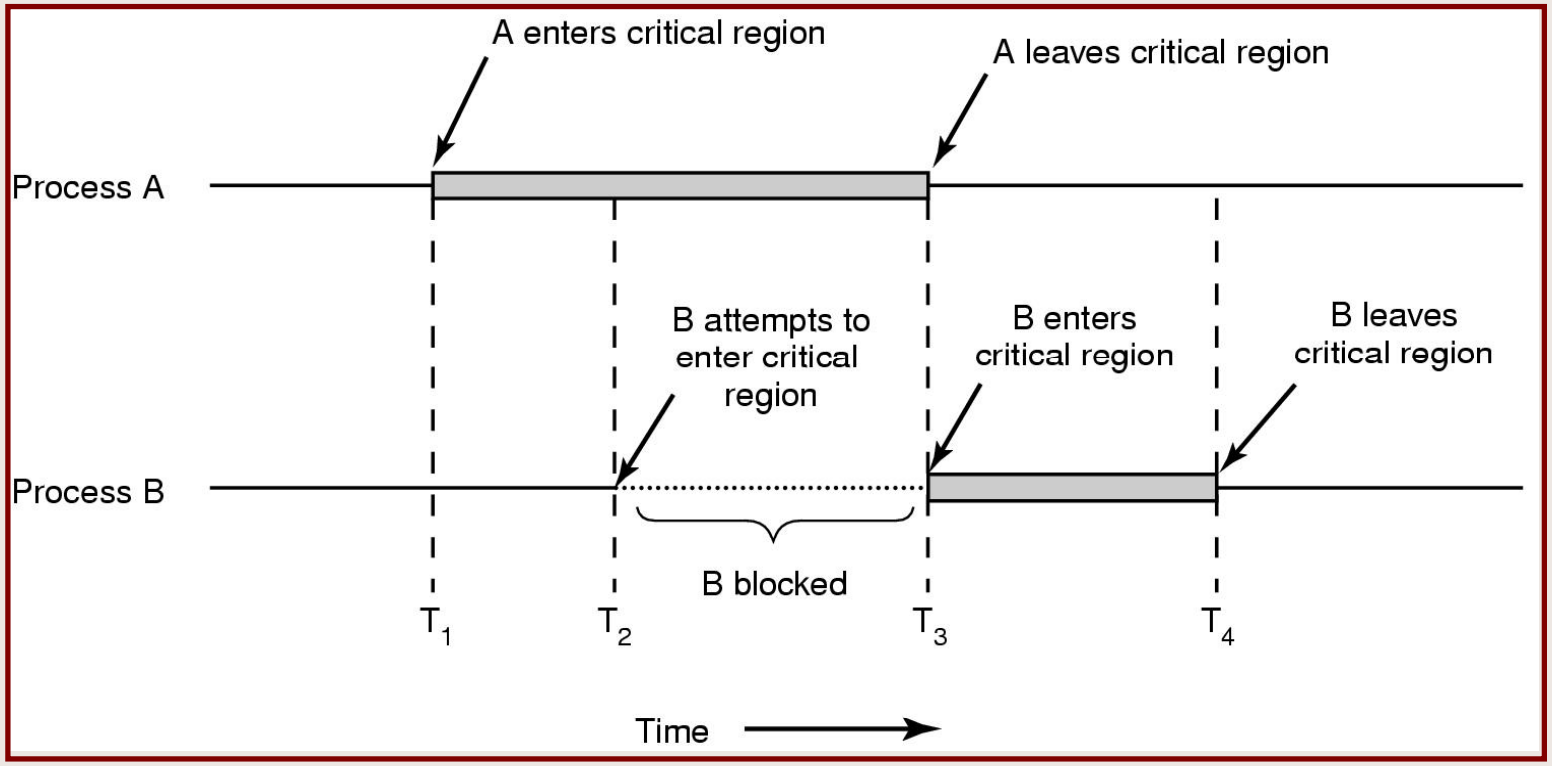
```
int suma=0;
…
void suma(int ni, int nf) {
   int j, suma_parcial=0;
   for (j=ni; j<=nf; j++)
        suma_parcial = suma_parcial + j;
   suma = suma + suma_parcial; }
```

Regiones críticas

- Parte del programa que accede a la memoria compartida
- Se necesita **exclusión mutua**
- Condiciones para una buena solución:
  - No puede haber dos procesos simultáneos en la región crítica
  - No se pueden hacer suposiciones sobre velocidades o número de CPUs
  - Un proceso no puede bloquear a otro sin estar en la región crítica
  - Ningún proceso debe esperar indefinidamente al acceso a la región crítica

Sección 2.3.2

Exclusión mutua



Exclusión mutua con espera ocupada

- Deshabilitando interrupciones
  - No se puede expropiar a un proceso (no hay cambios de contexto)
  - El kernel lo hace con frecuencia, pero es peligroso en procesos del usuario
  - Es inútil en sistemas multiprocesador o en sistemas multicore

Sección 2.3.3

Variables de candado (o bloqueo)

- Intento de solución software
  - Candado=0 (ningún proceso está en su región crítica)
  - Candado=1 (algún proceso está en su región crítica)
- No funciona!

Alternancia estricta

- Una variable establece el turno

```
While (TRUE) {
    while (turno !=0) ;
    region_critica();
    turno=1;
    region_no_critica();
}
```

```
While (TRUE) {
    while (turno !=1) ;
    region_critica();
    turno=0;
    region_no_critica();
}
```

- Espera ocupada (espera activa)
- No se cumple la condición 3
- Los procesos trabajan al ritmo del más lento

Solución de Peterson

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
        int other;                      /* number of the other process */

        other = 1 – process;            /* the opposite of process */
        interested[process] = TRUE;     /* show that you are interested */
        turn = process;                 /* set flag */
        while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
        interested[process] = FALSE;    /* indicate departure from critical region */
}
```

¿Espera activa?

La instrucción TSL

- Ayuda del hardware: Es una instrucción atómica: bloquea el bus de memoria
- Funciona para multicores

tsl R1, candado

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was nonzero, lock was set, so loop
        RET                        | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET                        | return to caller
```

¿Espera activa?

Dormir y despertar

- Usa llamadas al sistema de comunicación
  - sleep
  - wakeup
- Evita esperas activas que pueden dar lugar al problema de inversión de prioridades
- Usan una posición en memoria para asociarse

Sección 2.3.4

El problema del productor-consumidor

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Semáforos

- Es una variable que toma valores enteros positivos.
- Regida por dos operaciones atómicas:
  - **down**: si el semáforo es mayor que 0 lo decrementa sin más, si es 0 se bloquea el proceso
  - **up**: incrementa el semáforo, si algún proceso estaba bloqueado por el semáforo se despierta para ejecutar su down

Sección 2.3.5

El problema del productor-consumidor resuelto con semáforos

```
#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 */
        }
}
```

Mutexes

- Es una variable que toma valores 0 ó 1
- Regida por dos operaciones atómicas: lock y unlock
- Implementación de lock y unlock con TSL:

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Sección 2.3.6

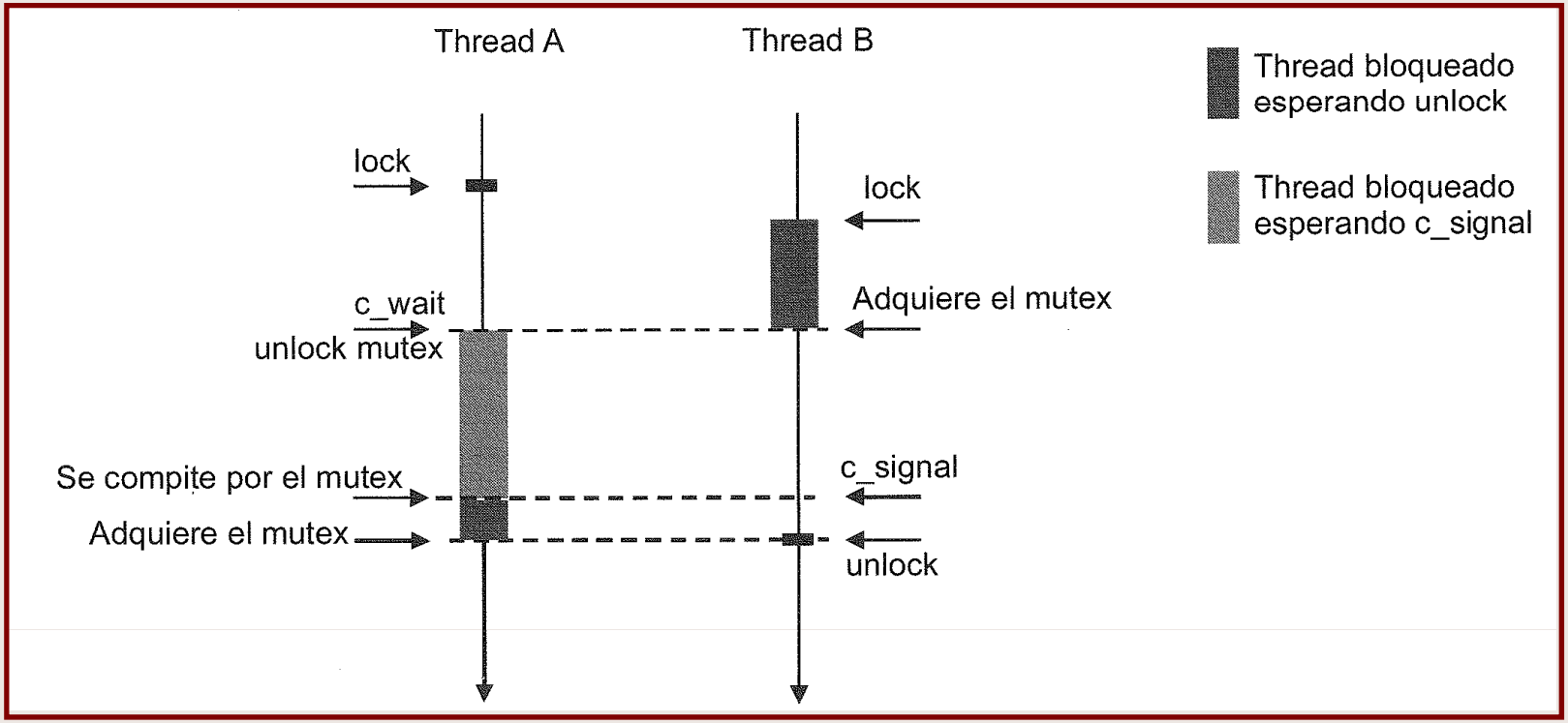
Mutexes con pthreads

- Funciones:
  - Pthread_mutex_init
  - Pthread_mutex_destroy
  - Pthread_mutex_lock
  - Pthread_mutex_trylock
  - Pthread_mutex_unlock

Variables de condición

- Sirven para bloquear threads hasta que se cumpla una determinada condición
- Están asociadas a mutexes
- Funciones para pthreads:
  - Pthread_cond_init
  - Pthread_cond_destroy
  - Pthread_cond_wait
  - Pthread_cond_signal
  - Pthread_cond_broadcast

Variables de condición



El problema del productor-consumidor resuelto con mutexes y variables de condición (1/3)

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                        /* put item in buffer */
                pthread_cond_signal(&condc);       /* wake up consumer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}
```

El problema del productor-consumidor resuelto con mutexes y variables de condición (2/3)

```
void *consumer(void *ptr)                                   /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                                 /* take item out of buffer */
                pthread_cond_signal(&condp);        /* wake up producer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}
```

El problema del productor-consumidor resuelto con mutexes y variables de condición (3/3)

```
int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Monitores (1/2)

- Es un mecanismo de alto nivel: menor posibilidad de error pero menos versátil
- El monitor tiene una estructura similar a la de un objeto en el que se definen procedimientos (las regiones críticas) con exclusión mutua garantizada (lo hace el compilador seguramente con semáforos o mutexes)

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Sección 2.3.7

Monitores (2/2)

- Para hacerlos más versátiles suelen ir acompañados del uso de variables de condición

```
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;
```

Paso de mensajes

- Funciones send y receive
- Acuse de recibo (acknowledgement)
- Autenticación
- Buffers de envío y recepción

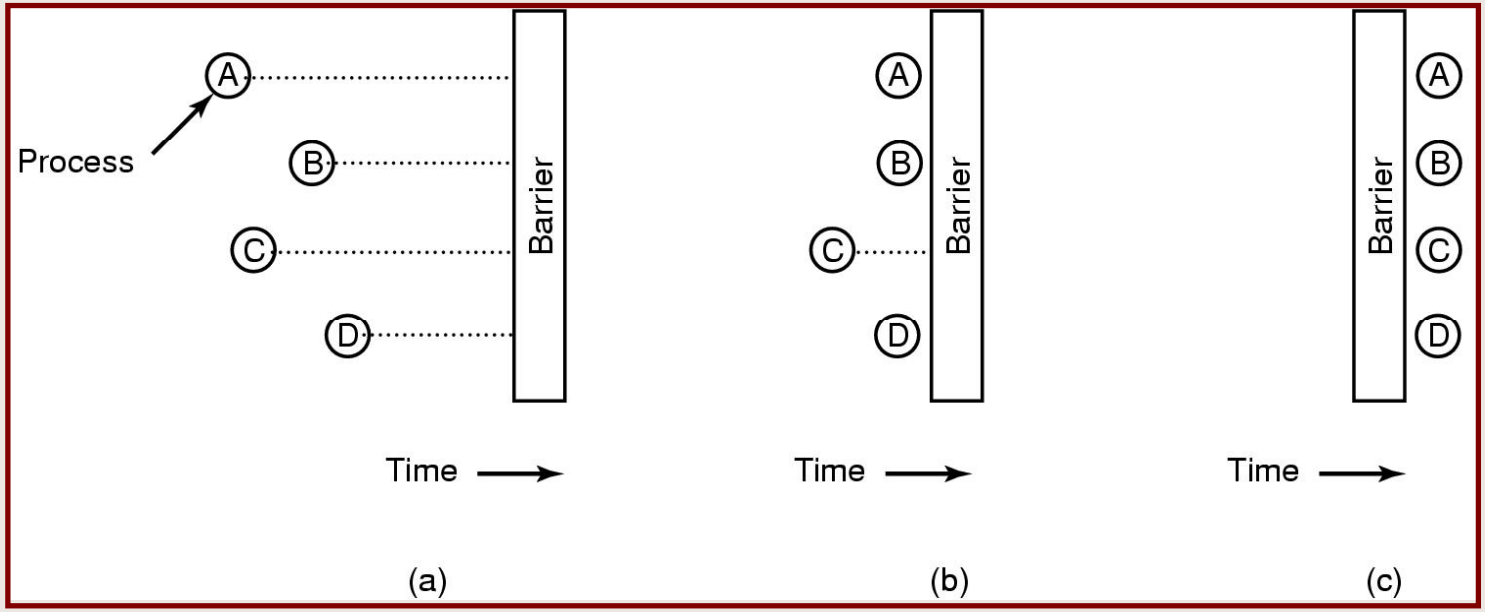
Sección 2.3.8

El problema del productor-consumidor resuelto con paso de mensajes

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}
```

```
void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Barreras



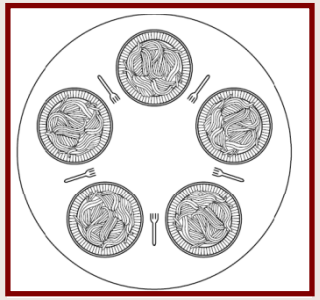
Sección 2.3.9

Problemas clásicos de comunicación entre procesos

- El problema de los filósofos
- El problema de los lectores y escritores

Sección 2.5

El problema de los filósofos. Planteamiento



```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
        while (TRUE) {
                think( );                       /* philosopher is thinking */
                take_fork(i);                   /* take left fork */
                take_fork((i+1) % N);           /* take right fork; % is modulo operator */
                eat( );                         /* yum-yum, spaghetti */
                put_fork(i);                    /* put left fork back on the table */
                put_fork((i+1) % N);            /* put right fork back on the table */
        }
}
```

Sección 2.5.1

El problema de los filósofos. Solución (1/3)

```
#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 */
        }
}
```

El problema de los filósofos. Solución (2/3)

```
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 */
}
```

El problema de los filósofos. Solución (3/3)

```
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 problema de los lectores y escritores (1/2)

```
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 */
        }
}
```

Sección 2.5.2

El problema de los lectores y escritores (2/2)

```
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 */
        }
}
```