Operating Systems

Lecture 2.3 - Inter Process Communication

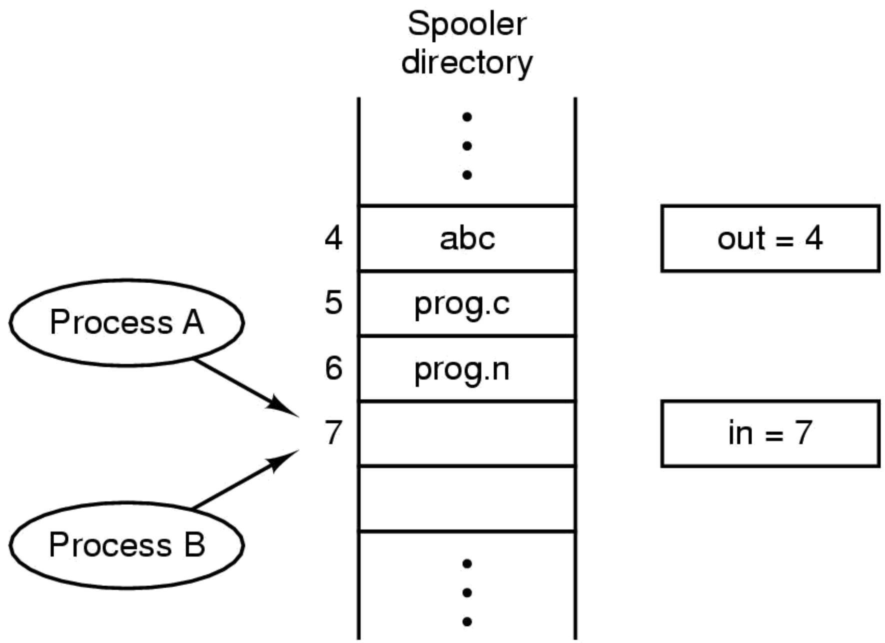
Golestan University

Hossein Momeni
momeni@iust.ac.ir

Classical IPC problems

- Processes frequently need to communicate with other processes
- This is called Inter-Process Communication or IPC.
- Three problems in IPC:
  1. Race Condition
  2. Deadlock
  3. Starvation

Race Conditions



Two processes want to access shared memory at same time. What happens if they try to access it simultaneously?

Race Conditions

- Situations like this are called race conditions.
- What will happen if two processes execute the following code?

```
X=0;
…
Read(x);
X++;
Write(x);
```

Critical Regions (1)

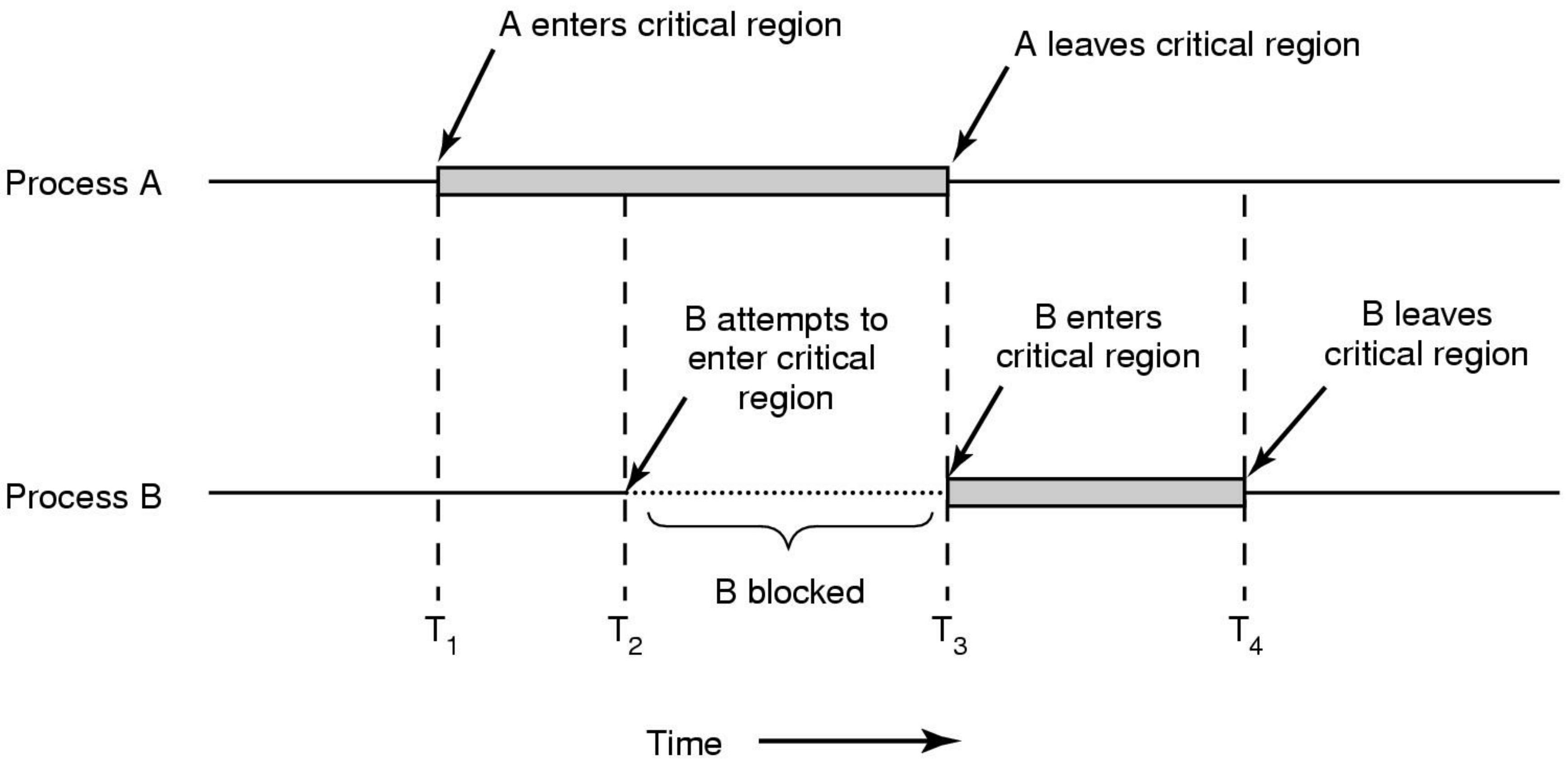
Critical Regions solution: Mutual exclusion*

Four conditions to provide mutual exclusion

1. No two processes simultaneously in critical region (Mutual exclusion)
2. No assumptions made about speeds or numbers of CPUs (Generality)
3. No process running outside its critical region may block another process (Progress)
4. No process must wait forever to enter its critical region (Bounded waiting)

***Mutual exclusion**: A collection of techniques for sharing resources so that different uses do not conflict and cause unwanted interactions

Critical Regions (2)



Mutual exclusion using critical regions

Two ways for waiting: 1- Busy waiting 2- Blocking

Mutual exclusion solutions

- Approaches:
  - Software approaches
  - Hardware Approaches
  - Operating system level approaches
  - Compiler level approaches

- Busy Waiting
  - Interrupts disable
  - Lock variables
  - Strict alternation
  - Peterson’s solution
  - TSL instruction
- Sleep and wake up
  - Semaphor
  - Monitor
  - Message Passing

Interrupts disable

- CPU switches with interrupt
- When Interrupts are disable, other process are disable
- Problems
  - May be interrupt not enable & system halted!
  - Impossible for multi processors system

Lock variables

- Lock variable idea is a software solution:
- P1:
  - while Lock=1 do wait
  - Lock=1
  - Enter to critical region
  - Lock=0
- P2:
  - while Lock=1 do wait
  - Lock=1
  - Enter to critical region
  - Lock=0
- No three conditions (Mutual exclusion, Progress, Bounded waiting)

Strict alternation

```
while (TRUE) {
    while (turn != 0);       /* loop */
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1);       /* loop */
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Proposed solution to critical region problem

(a) Process 0. (b) Process 1.

- **No Progress condition**

Peterson's solution (1982)

Peterson's solution for achieving mutual exclusion

```
#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 = other;                      /* set flag */
     while (turn == other && interested[other] == TRUE)    /* null statement */ ;
}

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

Peterson's solution

- Guarantee three conditions
- Busy waiting
- Two-Process.

TSL instruction

```
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 non zero, 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
```

- Mutual Exclusion, Progress
- No Bounded waiting and Busy waiting
- Not supported by some processor (such as Intel)

Problems of Mutual Exclusion with Busy Waiting

- Busy Waiting problem (waiting loop)

- Priority inversion problem
  - There is two processes H and L.
  - H has higher priority than L.
  - L is in its critical section and H becomes ready.
  - What happens?

Sleep and Wakeup

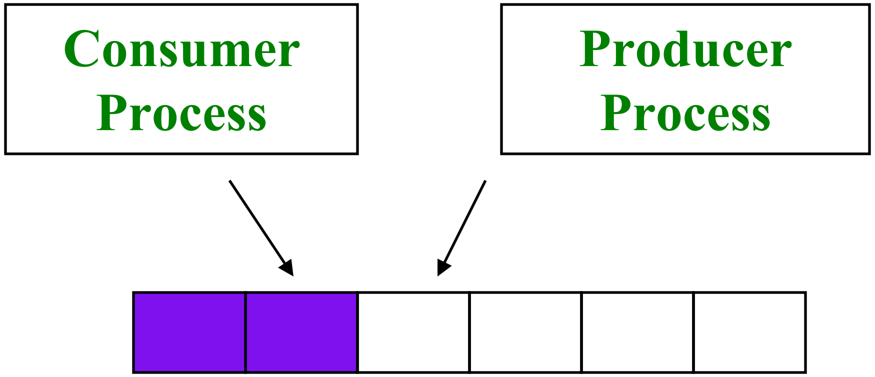
- Semaphores
- Monitors
- Message Passing

Sleep and Wakeup

- Despite busy waiting methods which waste CPU cycles, in this method processes may sleep or wakeup using system calls.
- Let's clarify this approach using an example, namely, producers and consumers.

Sleep and Wakeup

- Consider two processes which produce and consume items from/to a buffer with size N.



Sleep and Wakeup

Producer-consumer problem with fatal race condition

```
#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 */
     }
}
```

Problems

- This solution is also wrong.
- Consider N=0
  - Consumer before sleeping, CPU switch to producer
  - Producer produce till count=N and go to the sleep mode.
  - CPU switch to the Consumer
  - Consumer go to the sleep mode
  - Deadlock!!

Semaphores

- In many problems there is a need to count an event, like producing an item or consuming it.
- Accessing to this counter should be protected against concurrent processes.
- Such a protected counter is called a semaphore which has more features.

Semaphores (cont.)

- Two operators are defined on a semaphore: Down and Up (generalizations of sleep and wakeup)

Down(int& x) {
If (x > 0)
x--;
else
Sleep() };
	Up(int& x) {
	If (there is any waiting process)
	Pick a process from queue and make it ready;
	else
	x++ };

Semaphores (cont.)

- How to protect a critical section using semaphores?

```
int s = 1;
Down(s);
…
Critical Section
…
Up(s);
```

Semaphores (cont.)

- Consider a resource which can be shared by 3 processes. How accessing this device can be protected using semaphores?

```
int x = 3;
Down(x);
…
Accessing the shared resource.
…
Up(x);
```

Semaphores (cont.)

- Mutex Semaphore (Binary)
- Define 2 operation Down & Up
  - Down or Wait or P (Proberen)
  - Up or Signal or V (Verhogen)

Semaphores (Cont)

- We use semaphore for:
  - Mutual Exclusion (initial value=1)
  - Process Synchronization (initial value=0)
    - Example Synchronization:
    - We want to print AB

P1	P2
-----	------
A	wait (s)
Signal (s)	B

Producer & Consumer solution with Semaphores

The producer-consumer problem using semaphores

```
#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 */
     }
}
```

Monitors (Hoare 1974)

- Semaphore solution is difficult
- Semaphore solution is low level
- Process deadlock
  - Exchange down operations in producer
  - Consider the buffer is full

Monitors (cont.)

- To make it easier to write correct programs, a higher level primitive called monitor is introduced.
- It is a collection of procedures, variables and data structures that are all grouped in a package.
- An important property:
  - Only one process can be active in a monitor at any time.

Monitors model

```
monitor monitor_name
{         shared variable declarations
          procedure p1(…) {
          ….
          }
           procedure pn(…) {
          ….
          }
          {
                    initialization code
          }
}
```

Monitors (cont.)

- Monitors are a programming language construct, so the compiler should handle calls to procedures.
- When a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active or not.
- Condition variables
  - Wait and Signal operation

Producer-consumer problem with monitor

```
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;
```

Outline of producer-consumer problem with monitors

- only one monitor procedure active at one time
- buffer has *N* slots

Message Passing

- Monitor Problems:
  - Not Support in some programming language such as C and Pascal
  - Use in single processor systems or share memory multi processor
- Message Passing:
  - Inter process communication without share memory
  - Use send & receive system call for communication (such as semaphores)
  - send(destination,&message)
  - receive(source,&message)
- If message is not exist:
  - Receiver is blocked until message is not reply
  - Return error message

Producer-consumer problem with N messages

```
#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 */
      }
}
```

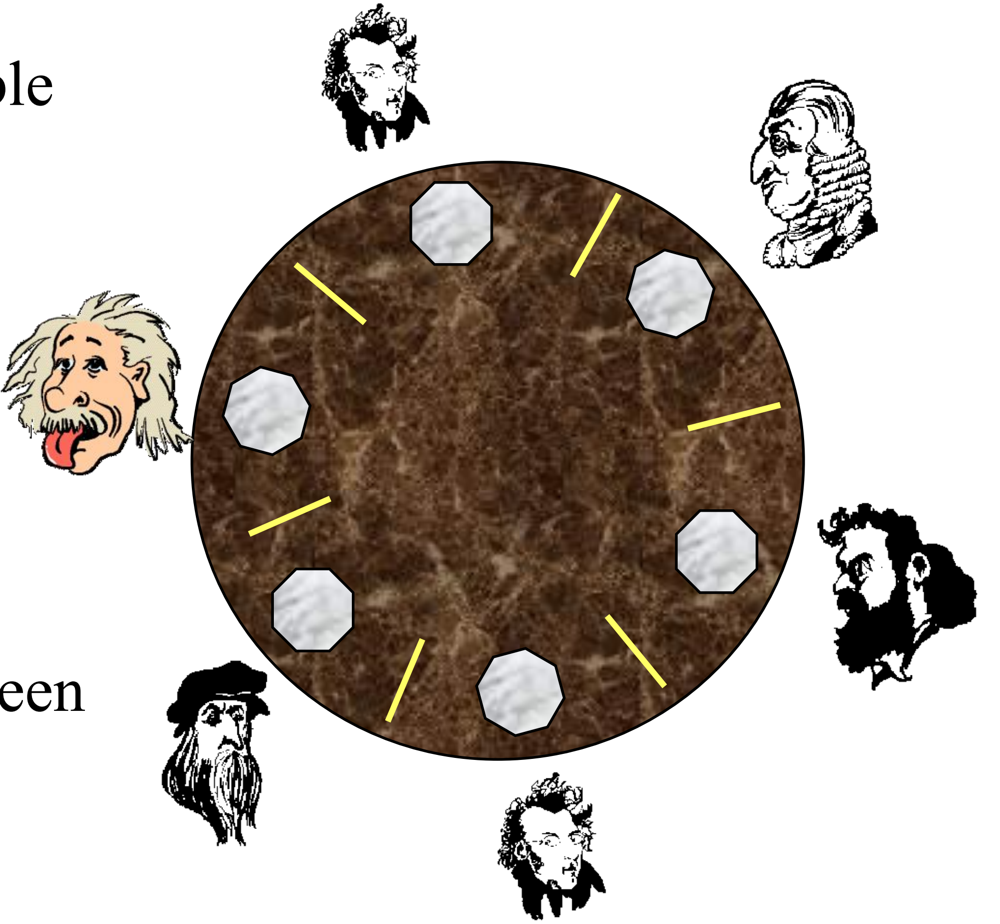
```
void consumer(void)
{
      int item, i;
      message m;
      for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
      while (TRUE) {
            receive(producer, &m);              /* get message containing item */
            item = extract_item(&m);            /* extract item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

Classical IPC problems

Classical synchronization problems

Dining Philosophers

- *N* philosophers around a table
  - All are hungry
  - All like to think
- *N* chopsticks available
  - 1 between each pair of philosophers
- Philosophers need two chopsticks to eat
- Philosophers alternate between eating and thinking
- Goal: coordinate use of chopsticks



A nonsolution to the dining philosophers problem

```
#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 */
     }
}
```

Solution to dining philosophers problem (part 1)

```
#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 */
    }
}
```

Solution to dining philosophers problem (part 2)

```
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]);
     }
}
```

Readers and Writers

- There are two semaphores in this solution.
  - One for writing to database.
  - One for counting the readers.

```
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 */
    }
}
```