Inter-Process Communication (IPC)

1. Introduction

Process frequently need to communicate with other process e.g in a shell pipeline the output of the first process must be passed to second process and so on down line. Thus there is a need for communication between processes in a well structured way not using the interrupt.

IPC provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same addresses space, IPC is particularly useful in a distributed environment where the communicating processes may reside on different computers connected with a network.

There are three issues related to IPC:

1. How one process can pass the information to another?
2. How to make sure two or more processes do not get into each Other's way when engaging in critical activities?
3. How to maintain the proper sequence when dependencies are Presents?

2. Race Condi tons

In some operating system processes that are working together may share some common storage that each one can read or write. The shared storage may be in main memory (in kernel data structure) or it may be a shared file.

Let us consider an example: a print spooler. When a process wants to print a file, it enters the filename in a special spooler directory. Another process, the printer daemon, periodically checks to see if there are any files to be printed; and if there are, it prints them and then removes their name form the directory.

Imagine that there are two variables "out" which points the next file to be printed and "in" which points to the next free slot in the directory. At any instant slots 0 to 3 are empty and slots 4 to 6 are full.

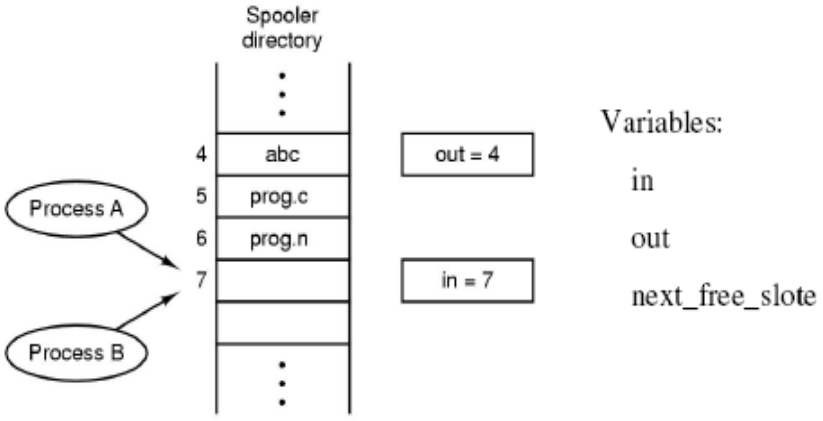


Fig: two processes want to access shared memory at the same time

More or less simultaneously, process A and B decide they want to queue a file for printing. Let process A reads in and stores the value 7 in a local variable called next_free_slot. Just then clock interrupt occurs and CPU switches to process B. process B also reads in and gets a 7. It also stores it in a local variable next_free_slot. It stores the name of its file in slot 7 and updates it to be 8. Eventually, process A runs again starting from the place it left off. It looks at next_free_slot; finds 7, there and writes its filename in slot 7, erasing the name process B, just put there.
Thus process B will never receive any output. The situation like this where two or more processes are reading or writing some shared data and the final result depends on who run precisely is called race condition.

3. How to avoid Race Condition

We know that the race condition are occurred when

- One process reading and another process writing same data
- Two or more process writing same data

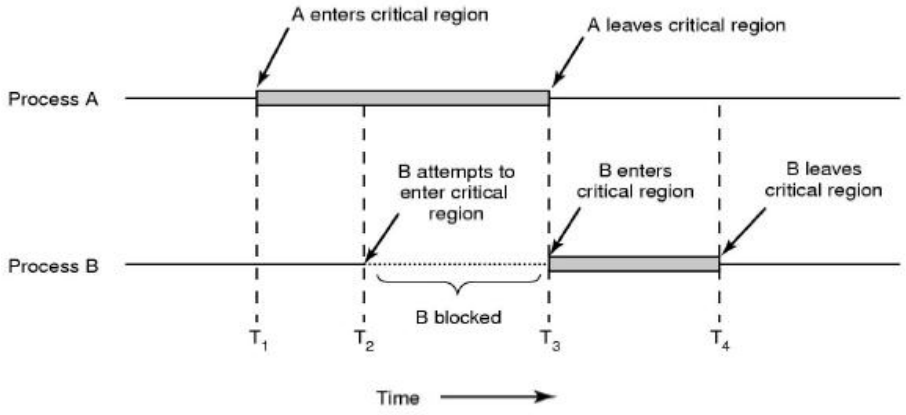
Mutual Exclusion:

Prohibiting more than one process from reading, writing the same data at the same time is called Mutual Exclusion. i.e. some way of making sure that if one process is using a shared variables or files, the other process will be excluded from doing the same thing.
Example one process (say A) increments (use) the shared variables, all other processes desiring to do so at the same moment should be kept waiting, when that process (A) has finished accessing the shared variable, one of the processes waiting to do so should be allowed to processed. In this fashion, each process accessing the shared data excluded all other from doing from doing so simultaneously. This is called mutual exclusion.

Critical Regions:

Mutual exclusion needs to be enforced only when processes access shared modifiable data, when processes are performing operations that do not conflict with one another they should be allowed to proceed concurrently. When a process is accessing shared modifiable data, the process is in critical section; all other processes (at least those that access the same modifiable data) are excluded from their own critical section.
While a process is in its critical section, other process may certainly continue executing outside their critical sections. When a process leaves its critical sections, then other one proceeds (if indeed there is a waiting process).
If we could arrange processes such that no two processes were ever in their critical regions at the same time, we could avoid race.
We need four conditions to hold to have good solutions of race conditions:

- No two processes may be simultaneously inside their CRs (Mutual exclusion).
- No assumptions may be made about the speeds or number of CPUs.
- No process running outside its CR may block other process.
- No process should have to wait forever to enter its CR.



Mutual exclusion using critical regions

Here process A enters its CR at time T_1. A little later, at time T_2 process B attempts to enter its CR but fails because another process is already in its CR and we allow only one at a time. Consequently, B is temporarily suspended until time T_3 when A leaves its CR, allowing B to enter immediately. Eventually B leaves at T_4 and we are back to the original situation with no process in their critical regions.

4. Mutual Exclusion with Busy Waiting

There are various ways or methods for achieving mutual exclusion, so that while one process is busy updating shared variable in its critical region, no other will enter its critical regions and cause the problems.

1. **Interrupt Disabling**

   The simples solutions for achieving ME is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disable, no clock interrupts can occur. The CPU is only switches from process to process as a result of clock interrupts or other interrupts, but when interrupts are turned off the CPU will not be switched to another process. Thus once a process has disabled interrupts, it can examine and update the shared variable without fear that any process will intervene.

   DisableInterrupt()
   //perform CR task
   EnableInterrupt()

   Advantages: ME can be achieved by implementing operating system primitives to disable and enable interrupts

   Disadvantages: it is unwise to give user process the power to turn off interrupts. Suppose that one of them did it and never turned on again. That could be end of the system.

   If the system is multiprocessor, with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory so it only works in single processor environment.

2. Locked variable

In this method a single shared (lock) variable is initially set to zero. When a process wants to enter its CR, it first checks the lock. If lock is 0, the process set it to 1 and enter the CR. If the lock us already 1, the process just waits until it becomes 0. Thus 0 means that no process is in its critical regions and 1 means that some process is in its critical regions.

Advantages:
Seems no problem.
ME can be achieved

Disadvantages: this idea contains exactly same fatal in spooler directory. Suppose that one process reads the lock and sees that it is 0, before it can set it to 1, another process scheduled, enter the critical regions, and set lock to 1. When first process runs again, it will set the lock to 1 and two processes will be in their critical regions at the same time. This violates the Mutual Exclusion.

3. Strict Alternation

In this method, process share a common integer variable *turn*. If *turn* = = 0, then process P_0 is allowed to execute in its critical region. If *turn* = = 1 then process P_1 is allowed to execute in its critical region.
Initially process P_0 inspects turn, find it to be 0 and enter its CR. Process P_1 also finds it to be 0 and therefore sits in a tight loop continually testing a variable until it becomes 1. Continuously testing a variable until some value appears is called ***busy waiting***. It should be avoided because it wastes CPU time.

```
While (true){
    while(turn!=i); /*loop */
    critical_section();
    turn = j;
    noncritical_section();
    }
```
Process P_i

```
While (true){
    while(turn!=j); /*loop*/
    critical_section();
    turn = i;
    noncritical_section();
    }
```
Process P_j

Advantages: Ensures that only one process at a time can be in its critical regions.

Disadvantages: Taking *Turn* is not a good idea when one of the processes is much slower than the other.
If process P_0 just finishes critical regions and again need to enter critical regions and the process P_1 is still busy at non critical regions, in this situation *turn* = = 1, but process P_1 is busy at non critical work while process P_0 wants to enter critical regions. This process P_0 is blocked by process P_1, which violates the conditions of mutual exclusions (condition 3)

4. PETERSON'S ALGORITHM

```
#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 */
}
```

Before using the shared variables (i.e. before entering its CR), each process calls *enter_region* with its own parameter number 0 or 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls *leave_region* to indicate that it is done and to allow other process to enter, if it so desires.

Let us see how this algorithm works. Initially neither process is in its critical region. Now, process 0 calls *enter_region*. It indicates its interest by setting its array element and sets turn to 0. Since process 1 is not interested, *enter_region* returns immediately. If process 1 now calls *enter_regions*, it will hang there until *interested [0]* goes to FALSE, an event that only happens when process 0 calls *leave_region* to exit the CR.

Advantages: preserves all condition of ME.

Disadvantages: Difficulty to program for n-process system and less efficient.

5. Hardware Solutions –TSL (Test and Set Lock)

As with other aspects of software, hardware features can make the programming tasks easier and improve system efficiency. We use a single hardware instruction that reads a variable, stores its value in a save area and sets the variable to a certain value. This instruction is called test and set; once initiated will complete all of these functions without interrupt. The indivisible test and set instruction-

TSL Rx, Lock

Test and Set Lock works as follows:

It reads the contents of memory word lock into register Rx and then stores a non-zero value at the memory address lock. The operation of reading the word and storing into it are guaranteed to be indivisible no other processor can access the memory word until the

instructions is finished. The CPU executing TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.
To use the TSL instruction, we will use a shared variable lock, to coordinate to access to shared memory. When lock is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done the process sets lock back to 0, using an ordinary move instruction.
The solution for critical section problem using this mechanism is shown below:

```
enter_region:
        TSL register, lock      |copy lock to register and set lock to 1
        CMP register, #0        |was lock 0?
        JNE enter_region        |if it was non zero, lock was set, so loop
        RET                     |return to caller;
critical_region();
leave_region:
        MOVE lock, #0           |store a 0 in lock
        RET                     |return to caller
noncritical_section();
```

Advantages: *Preserves all condition, easier programming task and improve system efficiency.*

Problems: *difficulty in hardware design.*

The first instruction copies the old value of lock to the register and then set lock to 1. Then the old value is compared with 0. If it is non-zero, the lock was already set, so the program just goes back to the beginning and tests it again. Sooner or later it will become 0 and the subroutine returns, with the lock set. Clearing the lock is simple. The program just stores 0 in lock.

Advantages: Preserves all conditions, easier programming task and improve system efficiency.

Problem: Difficulty in hardware design.

Alternative of Busy waiting

Both Peterson's solution and the solution using TSL are correct, but both have the defect of requiring busy waiting.
When a process want to enter its CR, it checks to see if the entry is allowed, if it is not, the process just sits in a tight loop waiting until it is, This cause waste of CPU Timer for noting.

1. Sleep and Wake up:

The one of the alternative to the busy waiting is to use sleep and wake up pair. Sleep is a system call that causes the caller to block, that is be suspended until another process wakes it up. The

wake up call has one parameter, the process to be awakened. Alternatively, both sleep and wake up each have one parameter, a memory address used to match up sleeps with wakeups.

Producer – Consumer problem

An example where sleep and wake up can be used is producer and consumer problem (also called bounded buffer problem).

Two process share a fixed sized buffer. One of them the producer puts information into the buffer, and the other one, the consumer takes it out. Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer and wakes it up.

To keep track of number of items in buffer, count variable is used. If the maximum number of items the buffer can hold is N, the producer's code will first test to see if count is N. if it is, the producer will go to sleep, if it is not, the producer will add and increment counter. The consumer's code is similar. First test count to see if it is 0. If it is, go to sleep, if it is non-zero, remove an item and decrement the counter.

```
#define N = 100     /* number of slots in the buffer*/
 int count = 0;      /* number of item 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 */
  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 go to sleep*/
 item = remove_item(); /*take item out
                                  of buffer*/
 count = count -1; /*decrement count*/
 if (count == N-1)wakeup(producer);
                        /*was buffer full ?*/
 consume_item();     /*print item*/
}
}
```

Problems: leads to race as in spooler directory.

The following situation can occur- The buffer is empty and consumer has read count, to see if it is zero. At that instant quantum expired and producer begins execution. The producer inset an item in buffer, increment counter and notice that it is 1, then producer calls "wake up" signal to wake the consumer up. Unfortunately, the consumer is not yet logically a sleep, so the wake up signal is lost. When the consumer next runs, it will test the value of count it previously read, finds it to be 0 and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

2. Semaphore:

The main drawback of sleep and wake up mechanism for solving producer-consumer problem is that, the wake up signal could be lost, in some conditions. Semaphores solve the lost wake up problem.

Semaphore are integer variable. They could have value 0, indicating that no wake ups were saved or some positive value if one or more wakeups were pending. In these method two operations down and up are used instead of sleep and wake up operations. The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements and just continue. If the value is 0, the process is put to sleep without completing the down for the moment. Checking the value changing it and possibly going to sleep is all done as a single indivisible atomic action.

The up operation increases the value of semaphore addressed. If one or more process were sleeping on that semaphore, unable to complete on earlier down operation, one of them is chosen by the system and is allowed to complete its down.

```
typedef int semaphore S;
void down(S)
        { if(S> 0)  S--;
          else      sleep();
        }
void up(S)
        {if(one or more processes are sleeping on S)
                one of these process is proceed;
         else    S++;
        }
while(TRUE){
        down(mutex)
        critical_region();
        up(mutex);
        noncritical_region();
        }
```

Solving producer- consumer problem using semaphores:

```
#define N 100                       /*number of slots in buffer*/
typedef int semaphore;              /*defining semaphore*/
semaphore mutex = 1;                /* control access to the CR */
semaphore empty = N                 /*counts empty buffer slots*/
semaphore full = 0;                 /counts full buffer slots*/

void producer(void)
{
     int item;
     while(TRUE){                                /*repeat forever */
                  item = produce_item();         /*generate something */
                  down(empty);                   /*decrement empty count*/
                  down(mutex);                   /*enter CR */
                  insert_item();                 /* put new item in buffer*/
                  up(mutex);                     /* leave CR*/
                  up(full);                      /*increment count of full slots*/
          }
}
void consumer(void)
{
     int item;
     while(TRUE){                                 /*repeat forever*/
          down(full);                             /*decrement full count */
          down(mutex);                            /*enter CR*/
          item = remove_item();                   /*take item from buffer*/
          up(mutex);                              /*leave CR*/
          up(empty);                              /*increment count of empty slots*/
          consume_item();              /*print item*/
     }
}
```

This solution uses three semaphores:

- Full: for counting the number of slots that are full.
- Empty: for counting the number of slots that are empty.
- Mutex: to make sure the producer and consumer do not access the buffer at same time.

Initially full is 0, empty is equal to the number of slots in the buffer, and mutex is initially 1. Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called binary semaphores. If each process does a down just before entering its CR and an up just after leaving it, mutual exclusion is guaranteed.

Use of semaphores

1. *To deal with n-process critical-section problem.*
   The n processes share a semaphore, (e. g. mutex) initialized to 1.
2. *To solve the various synchronizations problems.*
   For example two concurrently running processes: P1 with statement S1 and P2 with statement S2, suppose, it required that S2 must be executed after S1 has completed. This problem can be implemented by using a common semaphore, synch, initialized to 0.

```
P1:  S1;
     up(synch);
P2:  down(synch);
     S2;
```

3. *for achieving mutual exclusion.*

Criticality Using Semaphores

All process share a common semaphore variable mutex, initialize to 1. Each process must execute down(mutex) before entering CR, and up(mutex) afterward. *What happens when this sequence is not observed?*

1. When a process interchange the order of *down* and *up* operation: causes the multiple processes in CR simultaneously.
2. When a process replace *up* by *down*: causes the dead lock.
3. When a process omits *down* or *up* or both: violated mutual exclusion and deadlock occurs.

3. Monitors

Codes in semaphores are like assembly language code. To make easier to write correct programs, Hoare and Brinch Hansen proposed a higher-level synchronization primitive called a monitor.

A monitor is a collection of procedures, variables and data structures that all grouped together in a special kind of module or package. Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor. Following program illustrates monitors.

```
Monitor monitor_name
 {
   shared variable declarations;
   procedure p1(){ .....  }
   procedure p2(){ ....  }
                ....
   procedure pn(){ ...}
   {initialization code;}
 }
```

Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant. Monitors are a programming language construct, so the compiler knows they are special and can handle calls to monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor. If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter. So, no two processes will were executing their critical regions at the same time.

Solving producer- consumer problem using monitors:

```
Monitor ProducerConsumer
  {
    int count;
    condition full, empty;
    void insert(int item){
          if (count == N ) wait(full);
          insert_item(item);
          count++;
          if (count ==1) signal(empty)
          }
     void remove(){
          if(count ==0) wait(empty);
          remove_item();
          count--;
          if(count ==N-1) signal(full);
          }
    count =0;
    };
```

```
Void producer(){
     while(TRUE){
          item = produce_item();
          ProcedureConsumer.insert(item);
          }
     }
void consumer(){
     while(TRUE){
          item = ProduceConsumer.remove();
          consume_item();
          }
     }
```

Criticality using monitors

1. Lack of implementation in most commonly used programming languages.

   How to implement in C?

2. Both semaphore and monitors are used to hold mutual exclusion in multiple CPUs that all have a common memory, but in distributed system consisting multiple CPUs with its own private memory, connected by LAN, non of these primitives are applicable.

Semaphores are too low level and monitors are not usable except in a few programming language.

4. Message Passing

With the trend of distributed operating system, many operating systems are used to communicate through internet, intranet and remote data processing etc.
Message passing method of inter-process communication uses two primitives send and receive.

Interprocess communication based on two primitives: send and receive.

send(destination, &message);

receive(source, &message);

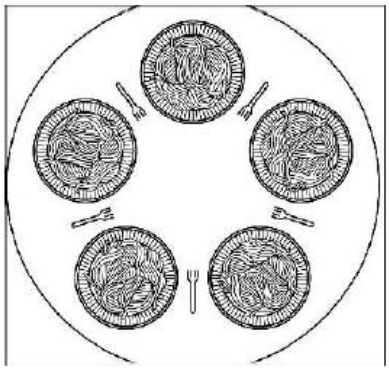
The send and receive calls are normally implemented as operating system calls accessible from many programming language environments.

Solving producer- consumer problem using monitors:

```
#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 */
          receive(consumer, &m);       /*wait for an empty to arrive*/
          build_message(&m, item);  /*construct a message to send*/
          send(consumer, &m); }
}
void consumer(void)
{    int item;
     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 item*/
     }}
```

Classical IPC Problems

1. **The dinning philosopher problems Problem**



Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork.
The life of philosopher consists of alternative period of eating and thinking. When philosopher gets hungry, she tries to acquire her left and right forks; she eats for a while, then puts down the forks and continues to think.

The problem is can we write a program for each philosopher that does what is supposed to do and never gets stuck?

Solution 1
When the philosopher is hungry it picks up a fork and wait for another fork, when get it eats for a while and put both forks back to the table.

```
#define N 5                         // Number of photospheres
void philosopher (int i)                    //i: photosphere number; from 0 to 4
{
    while (TRUE)
    {
            think();                // photosphere is thinking
            take_fork(i);           // take left fork
            take_fork((i+1)%N);     // take right fork
            eat();                  // eat spaghetti
            put_fork(i);            // put left fork back on table
            put_fork((i+1)%N);      // put right fork back on table
    }
}
```

Here, take_fork waits until the specified fork is available and then seizes it.
Problems:
Suppose that all the philosophers take their left fork simultaneously none will be able to take their right fork and therefore will be a deadlock.
Solution 2:
After taking the left fork, the program checks for right fork, if it is not available, the philosopher puts down the left one, waits for some time and then repeats the whole process.
Problem:
All philosophers could start the process of picking left fork simultaneously and so on. In this case starvation may occur.

Solution 3:

Using a binary semaphore mutex before starting to acquire a fork she would do down on mutex. After replacing the fork she would do up on the mutex.

Problem:

From theroritical view point the solution is adequate but from practical one, it has a performance bug; only one philosopher can be eating at any instance with five forks available, we should be able to allow two should be able to allow two philosophers to eat at the same time.

Solution 3: (A perfect solution)

```
#define N 5                     // Number of photospheres
#define  LEFT   (i+N-1)%N       // Number of i's left neighbors
#define  RIGHT  (i+1)%N         // Number of i's right neighbors
#define  THINKING    0          // photosphere is thinking
#define   HUNGRY     1          // photosphere is hungry
#define   EATING     2          // photosphere is eating
typedef int semaphore;          // semaphores are special kinds of int
int state[N];                   // array to keep track of everyone's state
semaphore mutex = 1; // ME for CR
semaphore S[N];                 // one semaphore per photosphere

void philosopher (int i)                //i: photosphere number; from 0 to 4
{
    while (TRUE)
    {
        think();                // photosphere is thinking
        take_forks(i);          // acquire two forks
        eat();                  // eat spaghetti
        put_forks(i);           // put both fork back on table
    }
}
void take_forks(int i)          //i: photosphere number; from 0 to 4
{
    down (& mutex);
    state[i] = HUNGRY;
    test [i];
    up (& mutex);
    down ( &S[i]);
}
void put_fokrs(int i)           //i: photosphere number; from 0 to 4
{
    down (&mutex);
    state [i] = THINKING;
    test (LEFT);
    test (RIGHT);
    up (&mutex);
}
void test(int i)                //i: photosphere number; from 0 to 4
{
```

```
    if ( state [i] = = HUNGRY && state [LEFT] != EATING && state [RIGHT] != EATING )
    {
            state [i] = EATING;
            up (&s[i]);
    }
}
```

Here, we use an array state to keep track of whether a philosopher is EATING, THINKING or HUNGRY (trying to acquire forks). A philosopher may move into EATING state if neither neighbor is EATING. Philosopher i's neighbor are defined by LEFT and RIGHT. In other words, if i is 2, LEFT is 1 and RIGHT is 3.

The problem uses an array of semaphores, one per philosopher, so HUNGRY philosophers can block if the needed forks are busy. Here each process runs the procedure philosopher as its main code, but the other procedure, take_forks, put_forks and test are ordinary procedure and not separated processes.