Chapter 3
InterProcess Communication (IPC)

IPC is a mechanism for processes to communicate and to synchronize their actions. The responsibility for providing communication is by O.S

3.1 The Critical-Section Problem

พิจารณาระบบที่ประกอบด้วย n โพรเซส { P_0, P_1, …, P_{n-1}} แต่ละโพรเซสจะมีส่วนของรหัสคำสั่ง ที่เรียกว่า ส่วนวิกฤต (critical section) ซึ่งใช้ในการเปลี่ยนแปลงข้อมูลหรือทรัพยากรที่ใช้ร่วมกัน ถ้าโพรเซสเหล่านั้นสามารถทำงานพร้อม ๆ กันได้ ถ้าไม่มีการป้องกัน อาจจะเกิดปัญหาความไม่คงเส้นคงวา (inconsistency) เกิดขึ้นได้ เช่น ปัญหาผู้ผลิต-ผู้บริโภค ที่มี buffer จำกัด (bounded-buffer producer-consumer problem) ที่ผู้บริโภคต้องรอถ้า buffer ว่าง และผู้ผลิตต้องรอถ้า buffer เต็ม ต่อไปนี้

ข้อมูลร่วม

```
var n;
        type item = …;
        var buff : array [0..n-1] of item;
           in, out : 0..n-1;
           counter : 0..n;
```

ผู้ผลิต

```
        Repeat
          …
         produce an item in nextp
          …
         while counter = n do no-op;
         buffer[in] := nextp;
         in := in + 1 mod n;
         counter := counter + 1;
        until false;
```

ผู้บริโภค

```
Repeat
    while counter = 0 do no-op;
   nextc := buff[out];
   out := out + 1 mod n;
   counter := counter – 1;
   …
 consume the item in nextc
   …
until false;
```

ถ้าผู้ผลิตปฏิบัติการกับคำสั่ง counter := counter + 1 พร้อมกับ (concurrent) ผู้บริโภคปฏิบัติการกับคำสั่ง counter := counter – 1 สมมุติว่าขณะนั้น counter มีค่าเท่ากับ 5 หลังจากปฏิบัติการแล้ว counter อาจจะมีค่าเป็น 4, 5 หรือ 6 ซึ่งไม่ถูกต้อง ค่าที่ถูกต้องควรเป็น 5 เหตุที่เป็นเช่นนี้เพราะคำสั่งระดับสูงข้างต้น เมื่อแปลเป็นภาษาเครื่องแล้วจะมีชุดคำสั่ง ดังนี้

counter := counter + 1 ➔

$register_1$:= counter

$register_1$:= $register_1$ + 1

counter := $register_1$

counter := counter - 1 ➔

$register_2$:= counter

$register_2$:= $register_2$ -1

counter := $register_2$

และถ้ามีลำดับการปฏิบัติการดังต่อไปนี้

T0: ผู้ผลิต ปฏิบัติการ	$register_1$:= counter	{ $register_1$ = 5}
T1: ผู้ผลิต ปฏิบัติการ	$register_1$:= $register_1$ + 1	{ $register_1$ = 6}
T2: ผู้บริโภค ปฏิบัติการ	$register_2$:= counter	{ $register_2$= 5}
T3: ผู้บริโภค ปฏิบัติการ	$register_2$:= $register_2$ –1	{ $register_2$= 4}
T4: ผู้ผลิต ปฏิบัติการ	counter := $register_1$	{ counter =6}
T5: ผู้บริโภค ปฏิบัติการ	counter := $register_2$	{ counter = 4}

ผลสุดท้าย counter มีค่าเท่ากับ 4

แต่ถ้ามีลำดับการปฏิบัติการดังต่อไปนี้

T0: ผู้ผลิต ปฏิบัติการ	$register_1$:= counter	{ $register_1$ = 5}
T1: ผู้ผลิต ปฏิบัติการ	$register_1$:= $register_1$ + 1	{ $register_1$ = 6}
T2: ผู้บริโภค ปฏิบัติการ	$register_2$:= counter	{ $register_2$= 5}
T3: ผู้บริโภค ปฏิบัติการ	$register_2$:= $register_2$ –1	{ $register_2$= 4}
T4: ผู้บริโภค ปฏิบัติการ	counter := $register_2$	{ counter= 4}
T5: ผู้ผลิต ปฏิบัติการ	counter := $register_1$	{ counter =6}

ผลสุดท้าย counter มีค่าเท่ากับ 6

แต่ถ้ามีลำดับการปฏิบัติการดังต่อไปนี้

T0: ผู้ผลิต ปฏิบัติการ	$register_1$:= counter	{ $register_1$ = 5}
T1: ผู้ผลิต ปฏิบัติการ	$register_1$:= $register_1$ + 1	{ $register_1$ = 6}
T2: ผู้ผลิต ปฏิบัติการ	counter := $register_1$	{ counter =6}
T3: ผู้บริโภค ปฏิบัติการ	$register_2$:= counter	{ $register_2$= 6}
T5: ผู้บริโภค ปฏิบัติการ	$register_2$:= $register_2$ –1	{ $register_2$= 5}
T6: ผู้บริโภค ปฏิบัติการ	counter := $register_2$	{ counter= 5}

ผลสุดท้าย counter มีค่าเท่ากับ 5

Where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called **race conditions.**

การแก้ปัญหาส่วนวิกฤตที่ดี จะต้องสอดคล้องกับข้อกำหนดต่อไปนี้

Mutual Exclusion: ต้องไม่มีโพรเซสอยู่ในส่วนวิกฤตพร้อม ๆ กันมากกว่าหนึ่งโพรเซส หรือมีเพียงโพรเซสเดียวที่อยู่ในส่วนวิกฤตในขณะใดขณะหนึ่ง

ต้องไม่ตั้งข้อสมมุติเกี่ยวกับ ความเร็วและจำนวนของซีพียู

Progress: ถ้าไม่มีโพรเซสใดเลยอยู่ในส่วนวิกฤต และมีโพรเซสตั้งแต่หนึ่งโพรเซสขึ้นไป ต้องการที่จะเข้าสู่ส่วนวิกฤต โพรเซสเหล่านั้นจะต้องมีส่วนร่วมในการถูกคัดเลือก ให้เข้าสู่ส่วนวิกฤตต่อไปอย่างยุติธรรม โดยไม่มีการกีดกันตัวเอง หรือกีดกันระหว่างกัน

Bounded Waiting: เวลาที่ต้องรอเข้าสู่ส่วนวิกฤต ตั้งแต่แสดงความจำนง จนได้รับอนุญาติ จะต้องอยู่ในขีดจำกัด และไม่มีการรออย่างไม่สิ้นสุด

In this section we will examine various proposals for achieving mutual exclusion, so that while one process is updating shared memory in its critical region, no other process will enter its critical region and cause trouble.

3.1.1 Disabling Interrupts

The simplest solution is to have each process disable all interrupts just after entering its critical region (with interrupts turned off the CPU will not be switched to another process) and re-enable them just before leaving it. This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did it, and never turned them on again? That could be the end of the system. Furthermore, if the system is a multiprocessor, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

On the other hand, it is frequently convenient for the kernel itself to disabling interrupts for a few instructions while it is updating variable or lists. The conclusion is: disabling interrupts is often a useful technique within the operating system itself but is not appropriate as a general mutual exclusion mechanism for user processes.

3.1.2 Busy Waiting

Consider software solution for the two-process.

Algorithm 1 : Lock Variable

```
                 /* initially shared variable lock = 0 */

while (true) {                               while (true) {
  while (lock!=0); /*  busy waiting */          while (lock !=0); /*  busy waiting */
  lock = 1;                                     lock = 1;
  critical_region();                            critical_region();
  lock = 0;                                     lock = 0;
  noncritical_region();                         noncritical_region();
}                                            }

        Process 0                                    Process 1
```

The mutual exclusion is not ok. P0 read lock = 0, process switch to P1 read lock = 0, what happen?

Algorithm 2: **Strict Alternation**

```
/* initially shared variable turn = 0 */

while (true) {                                   while (true) {
   while (turn !=0); /*  busy waiting */            while (turn !=1); /*  busy waiting */
   critical_region();                               critical_region();
   turn = 1;                                        turn = 0;
   noncritical_region();                            noncritical_region();
}                                                }
         Process 0                                        Process 1
```

The mutual exclusion is ok but it violates condition Progress of critical section problem.

หมายเหตุ เพราะเมื่อตัวเองออกจาก critical section แล้ว จะกำหนดให้ turn เป็นของอีกฝ่ายหนึ่ง ถ้าอีกฝ่ายยังไม่เข้า critical section แต่ตัวเองต้องการเข้า critical section อีก ไม่สามารถทำได้ เพราะ turn เป็นของอีกฝ่ายหนึ่ง ทำให้กีดกันตัวเอง จึงขัดแย้งกับ condition Progress

Algorithm 3: **Peterson's Solution** (The correct solution)

```
/* initially shared variable flag[0] = flag[1] = false and turn = 0 or 1*/
while (true) {                                      while (true) {
   flag[0] = true;                                     flag[1] = true;
   turn = 1;                                           turn = 0;
   while (flag[1] and turn ==1); /*  busy waiting */   while (flag[0] and turn ==0); /*  busy waiting */
   critical_region();                                  critical_region();
   flag[0] = false;                                    flag[1] = false;
   noncritical_region();                               noncritical_region();
}                                                   }
         Process 0                                           Process 1
```

For multiple-Process Solutions see, S. Abraham and B.G. Peter, "**Operating System Concepts**", 5th ,Addison-Wesley , 1998.

The TSL Instruction

Need a little help from hardware, especially multiple processor. Many computers have an instruction TEST AND SET LOCK (TSL) that works as follows. It reads the contents of the memory word into a register and then stores a nonzero value at that memory address. The operations of reading the word and storing into it are guaranteed to be indivisible- no other processor can access the memory word until the instruction is finished.

Before entering its critical region, a process calls enter_region,

```
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 (busy waiting) */
        ret                       /* return to caller; critical region entered
```

which does busy waiting until the lock is free; then it acquires the lock and returns. After the critical region the process calls leave_region, which stores a 0 in lock:

```
leave_region:
        move lock,0               /* store a 0 in lock
        ret                       /* return to caller
```

3.1.2 Sleep and Wakeup

Both Peterson's solution and the solution using TSL are correct, but both have the defect of requiring busy waiting. Not only does this approach waste CPU time, but it can also have unexpected effects. Consider a computer with two processes, H, with high priority and L, with low priority. The scheduling rules are such that H is run whenever it is in ready state. At a certain moment, with L in its critical region, H becomes ready to run (e.g., an I/O operation completes). H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever. This situation is sometimes referred to as the **priority inversion problem.**

Now let us look at some interprocess communication primitives that block instead of waiting CPU time when they are not allowed to enter their critical regions.

3.1.2.1 Semaphores

Dijkstra proposed having two operations, P and V (generalizations of SLEEP and WAKEUP, respective). In the text book use DOWN and UP, or wait and signal respectively.

A semaphore S is an integer variable, apart from initialization, is accessed only through two standard **atomic** operations: **wait** and **signal.** The classical definitions of wait and signal in busy waiting style are

```
wait(S) : while S <= 0 do no-op;
              S := S – 1;

signal(S) : S := S + 1;
```

This type of semaphore is also called a **spinlock**. Spinlocks are useful in multiprocessor systems. The advantage of a spinlock is that no context switch is required when a process must wait on a lock (a context switch may take considerable time). Thus ,when locks are expected to be held for very short times, spinlocks are quite useful.

To overcome the need for busy waiting, we can modify the definition of the wait and signal semaphore operations. To implement semaphores under this definition, we define a semaphore as a record:

```
type semaphore = record
                     value : integer;
                     L : list of process;
                 end;
```

The **counting semaphore** operations can now be defined as

```
wait(S) : S.value := S.value –1;
          if S.value < 0
            then begin
                   add this process to S.L; block;
                 end;
signal(S) : S.value := S.value + 1;
            if S.value <= 0
              then begin
                     remove a process P from S.L; wakeup(P);
                   end;
```

OR

```
wait(S) : if S.value > 0
            then S.value := S.value – 1;
            else begin
                   add this process to S.L; block;
                 end;
signal(S) : if S.L is not empty, one or more processes are waiting on S
            then begin
                   remove a process P from S.L;  wakeup(P);
                 end;
            else S.value := S.value + 1;
```

Semaphores that have only value 0 or 1 are called **binary semaphores**, usually use to do mutual exclusion in critical region problem.

Usage

- n-process critical-section problems

```
/* note  mutex is initialized to 1 */
while (1) {
  wait(mutex);
  critical_region();
  signal(mutex);
  noncritical_region();
}
```

- synchronozation problems

  For example, consider two concurrently running processes: P1 with a statement S1, and P2 with a statement S2. Suppose that we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0, and by inserting the statements

  ```
  S1;
  signal(synch);
  ```

  in process P1, and the statements

  ```
  wait(synch);
  S2;
  ```

  in process P2.

Disadvantage: Although semaphores provide a convenient and effective mechanism for process synchronization, their incorrect use can still result in timing errors that are difficult to detect.
For example,

```
P0              P1
wait(S);        wait(Q)
wait(Q);        wait(S);
  .               .
signal(S);      signal(Q);
signal(Q);      signal(S);
```

Suppose that P0 executes wait(S) and then P1 executes wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal operations cannot be executed, P0 and P1 are deadlocked.
หมายเหตุ deadlock คือเหตุการณ์ที่ต่างคนต่างรอ ไม่สามารถดำเนินการอะไรต่อไปได้

3.2.1.2 Monitors

To make it easier to write correct programs, Hoare and Brinch Hansen proposed a high level synchronization primitive called a **monitor**. A monitor is characterized by a set of programmer-defined operators. The representation of a monitor type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type. The syntax of a monitor is

```
type monitor_name = monitor
  variable declarations

   procedure entry P1( … );
      begin … end;

   procedure entry P2( … );
      begin … end;

  …
   procedure entry Pn( … );
      begin … end;

  begin
    initialization code
   end .
```

The monitor construct ensures that only one process at a time can be active within the monitor (programmer does not need to code this constraint explicitly , Figure 3.1).

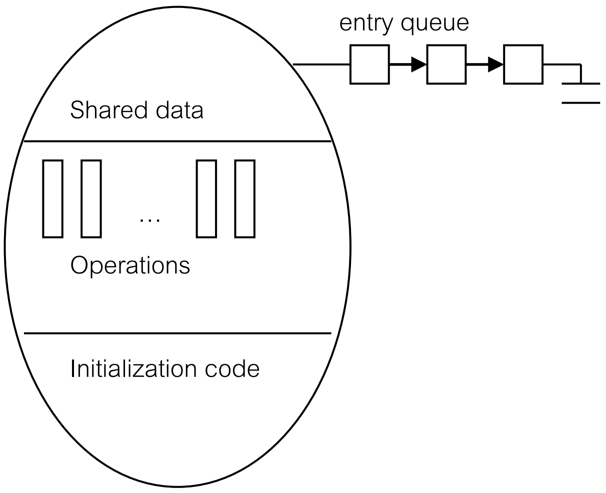

Figure 3.1 Scheme view of a monitor

The critical region is not sufficiently powerful for modeling some synchronization schemes. We need additional mechanisms with monitors for synchronization. These mechananisms are provided by the **condition** construct. A programmer who needs to write her own tailor-made synchronization scheme can define one or more variables of type condition:

var x,y : condition;

The only operations that can be invoked on a condition variable are **wait** and **signal**. The operation

wait(x);

means that the process invoking this operation is suspended until another process invokes

signal(x);

The signal(x) operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect; that is, the state of x is as though the operation was never executed (Figure 3.2). Contrast this with the signal operation associated with semaphores, which always affects the state of the semaphore.

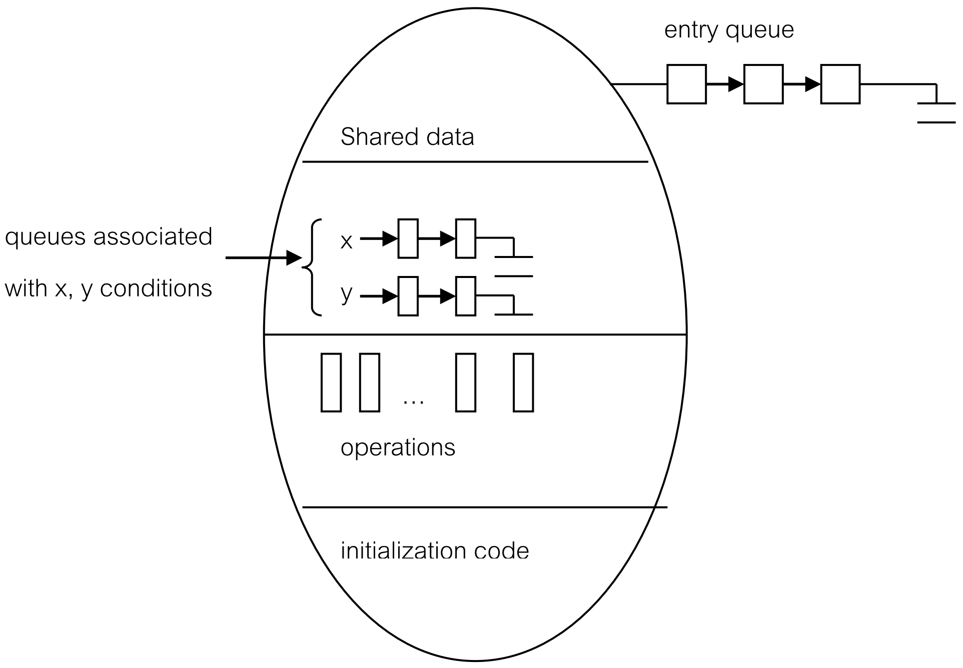

Figure 3.2 Monitor with condition variables

Now suppose that, when the x.signal operation is invoked by a process P, there is a suspended process Q associated with condition x. Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q will be active simultaneously within the monitor (which is not allowed). Note, however, that both processes can conceptually continue with their execution. Two possibilities exist:

1. P waits until Q either leaves the monitor, or waits for another condition.
2. Q waits until P either leaves the monitor, or waits for another condition.

Note! Choice (1) was advocated by Hoare. A compromise between these two choices was adopted in the language Concurrent Pascal. When process P executes the signal operation, it immediately leaves the monitor. Hence, Q is immediately resumed. This scheme is less powerful than Hoare's, since a process cannot signal more than once during a single procedure procedure call.

3.2.1.3 Message Passing

This method of interprocess communication uses two primitives SEND and RECEIVE, which, like semaphores and unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as

```
        send(destination, &message);
```

and

```
        receive(source, &message);
```

The former call sends a message to a given destination and the latter one receives a message from a given source (or from ANY, if the receiver does not care). If no message is available, the receiver could block until one arrives.

Example: The 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 (1) {
     produce_item(&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 (1) {
       receive(producer, &m);        /* get message containing item */
       extract_item(&m, &item);      /* extract item from message */
       send(producer, &m);           /* send back empty reply */
       consume_item(item);           /* do something with the item */
  }
 }
```

3.2.2 Classical IPC Problems

The Bounded-Buffer Problem (ดูหัวข้อ 3.1)

Producer:

```
            while (1) {
            …
            produce an item in nextp
            …
            wait(empty);
            wait(mutex);
            …
            add nextp to buffer
            …
            signal(mutex);
            signal(full);
      }
```

Consumer:

```
      while (1) {
            wait(full);
            wait(mutex);
            …
            remove an item from buffer to nextc
            …
            signal(mutex);
            signal(empty);
            …
            consume the item in nextc
```

```
        …
}
```

The Dining Philosophers Problem

Five philosophers are seated around a circular table. Each philosopher has a plate of especially slippery spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each plate is a fork as shown in Figure 3.3.

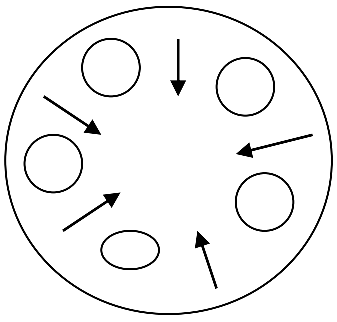

Figure 3.3

The life of a philosopher consists of alternate periods of eating and thinking. When a philosopher gets hungry, she tries to acquire her left right fork, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks and continues to think. The key question is : can you write a program for each philosopher that does what it is supposed to do and never gets stuck?

Incorrect solution:

```
#define N        5
philosopher(i)
int i;
{
   while (1) {
          think();               /* philosopher is thinking */
          take_fork(i);          /* take left fork */
          take_fork((I+1)%N);    /* take right fork */
          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 */
   }
}
```

Problems:

1. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, deadlock.
2. If we modify the program so that after talking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. This proposal too, fails. Starvation may occur. With a little bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, picking up their left forks again simultaneously, and so on, forever. **หมายเหตุ** starvation หมายถึงการรอคอยอย่างไม่มีที่สิ้นสุด ในกรณีนี้แม้ยังคงมีการดำเนินการอยู่แต่ไม่เกิดผลงานอะไรทั้งสิ้น กรณีอื่นเช่น เรารอคอยอย่างไม่มีที่สิ้นสุด แต่คนอื่นดำเนินการต่อไปได้
3. One improvement to this program that has no deadlock and starvation is to protect the five statements following the call to think by a binary semaphore, mutex. Before starting to acquire forks, a philosopher would do a wait(mutex). After replacing the forks, she

would do an signal(mutex). From a theoretical viewpoint, this solution is adequate. From a practical one, it has performance bug: only one philosopher can be eating at any instant. With five forks available, we should be able to allow two philosophers to eat at the same time.

Correct Solution:

```
#define N          5                /* number of philosophers */
#define LEFT       (i-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 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];
void philosopher(int i)
{
  while (1) {                       /* 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 */
{
  wait(&mutex);                     /* enter critical region */
  state[i] = HUNGRY;                /* record fact that philosopher i is hungry */
  test(i);                          /* try to acquire 2 forks */
  signal(&mutex);                   /* exit critical region */
  wait(&s[i]);                      /* block if forks were not acquired */
}
void put_forks(i)                   /* i : philosopher number, from 0 to N-1 */
{
  wait(&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 */
  signal(&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;
    signal(&s[i]);
  }
}
```

The Readers and Writers Problem

Imagine a big data base, such as an airline reservation system, with many completing processes wishing to read and write it. It is acceptable to have multiple processes reading the data base at the same time, but if one process is writing (i.e., changing) the data base, no other processes may have access to the data base, not even readers.

Two Solutions from paper of P.J. Courtois, F.Heymans, and D.L. Parnas, **Concurrent Control with “Readers” and “Writers”**, communication of the ACM, volume 14, October 1971, pp.667-668.

Solution 1: reader has higher priority than writer (writer may starvation)
No reader be kept waiting unless a writer has already obtained permission to use the resource

Integer readcount; (initial value = 0)
Semaphore mutex, w; (initial value for both = 1)

```
READER                            WRITER
wait(mutex);
readcount := readcount + 1;
If readcount = 1 then wait(w);
signal(mutex);
                                  wait(w);
…                                 …
reading is performed              writing is performed
…                                 …
wait(mutex);                      signal(w);
readcount := readcount – 1;
if readcount = 0 then signal(w);
signal(mutex);
```

Solution 2: writer has higher priority than reader (reader may starvation)
Once a write is ready to write, he performs his “write” as soon as possible.

Integer readcount, writecount; (initial value = 0)
Semaphore mutex1, mutex2, mutex3, w, r; (initial value = 1)

```
READER                            WRITER
wait(mutex3);                     wait(mutex2);
wait(r);                          writecount := writecount + 1;
wait(mutex1);                     if writecount = 1 then wait(r);
readcount = readcount + 1;        signal(mutex2);
if readcount = 1 then wait(w);    wait(w);
signal(mutex1);
signal(r);
signal(mutex3);
…                                 …
reading is done                   writing is performed
…                                 …
wait(mutex1);                     signal(w);
readcount := readcount – 1;       wait(mutex2);
if readcount = 0 then signal(w);  writecount := writecount – 1;
signal(mutex1);                   if writecount = 0 then signal(r);
                                  signal(mutex2);
```

Solution 3: reader and writer have the same priority

```
type readersandwriters monitor;
var
  readers : integer;
  someoneiswriting: boolean;
  readingallowed, writingallowed: condition;
```

```
procedure beginreading;
  begin
    if someoneiswriting or queue(writingallowed)
       then wait(readingallowed);
    readers := readers + 1;
    signal(readingallowed)
  end;
procedure finishedreading;
  begin
    readers := readers – 1;
    if readers = 0 then signal(writingallowed)
  end;
procedure beginwriting;
  begin
    if readers > 0 or someoneiswriting
       then wait(writingallowed);
    someoneiswriting := true
  end;
procedure finishedwriting
  begin
    someoneiswriting := false;
    if queue(readingallowed)
        then signal(readingallowed)
        else signal(writingallowed)
  end;
begin
  readers := 0;
  someoneiswriting := false
end;
```

The Sleeping Barber Problem

See text book section 2.3.3 pp. 80.

Final Remark! For variety of IPC implementing in UNIX see, W. Richard Stevens, “UNIX Network Programming”, Prentice-hall, 1991.