Inter-Process Communication (IPC)

1. **Critical Regions and conditions**
2. **Mutual Exclusion**
3. **Mutual Exclusion Primitives and Implementation**
    - **3.1. Dekker's algorithm**
    - **3.2. Peterson's algorithm**
    - **3.3. TSL (Test and Set Lock)**
    - **3.4. locks**
4. **Producer and Consumer problem**
5. **Monitors**
6. **Message Passing**
7. **Classical IPC problems**
    - **7.1. The Dining Philosophers problem**
    - **7.2. The readers and writers problem**
    - **7.3. The sleeping barber problem**

1. Introduction:

Processes frequently need to communicate with other processes so, there is a need for a well-structured communication among processes. Inter-process communication (IPC) is a capability supported by an operating system that allows one process to communicate with another process. The processes can be running on the same computer or on different computers connected through a network. An operating system provides **inter-process communication** to allow processes to exchange information. Inter-process communication is useful for creating *cooperating* processes. The main communication methods are;

- **Shared memory (Shared Variable):** In this method, process communicates through some shared variable.
- **Message passing:** In this method, messages are exchanged among the processes.

IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. IPC is required in all multiprocessing systems, but it is not generally supported by single-process operating systems such as DOS, OS/2, etc.

2. Race condition:

In operating systems, processes that are working together share some common storage (main memory, file etc.) that each process can read and write. When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, these conditions are called race conditions. OR, the situation where several process access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access take place is called race condition. To guard

against the race condition we need to ensure that only one process at a time can be manipulating the shared variables.

3. Critical regions or critical section

The part of program in which shared variable is updated is called critical region or critical section. If we arrange the process in such a way that no two processes enter the critical section at the same time, then we could avoid race. When process is accessing shared data, the process is said to be in critical section. When one process is in its critical section all other processes are excluded from entering in its critical section. Each process must request permission to enter its critical section.

We need four conditions that hold a good solution for avoiding race condition:

- No two processes may be simultaneously inside their critical section.
- No process running outside its critical section may block other processes.
- When no process is in the critical section, any process requesting for entry in critical section must be permitted without delay.
- A process is granted entry in critical section for finite time only.

4. Mutual Exclusion:

When number of processes executes concurrently, the critical section of one process should not be executed concurrently with the critical section of another process. When a process enters into its critical section, all other processes are restricted until the current running process is completed. Thus, only one process at a time is allowed to access the shared resource. This is known as mutual exclusion. Mutual exclusion is a mechanism to ensure that only one process is doing certain thing at a time and other processes are prevented from modifying the shared variable until the current process is finished.

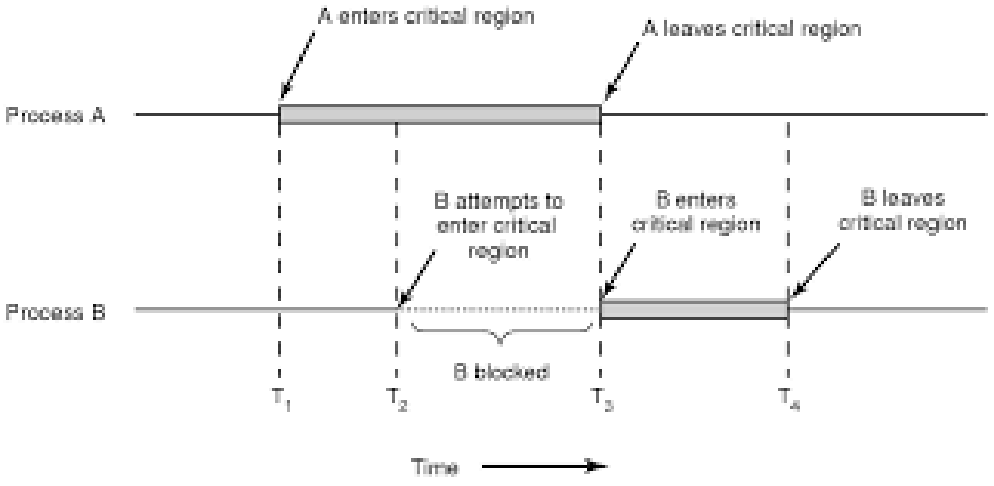


Fig: Mutual Exclusion using critical regions

In above figure, process A enters its critical section at time T1. At time T2, process B attempts to enter its critical section but denied because another process is already in its critical section and allows only one process at a time. So, process B is temporarily suspended until T3 when process A leaves its critical section and allows process B to enter immediately. Process B leaves its critical section at time T4.

5. Mutual Exclusion primitives and implementation:

5.1. Dekker's Algorithm:

Dekker's algorithm is the first known correct solution to the mutual exclusion problem in concurrent programming. Originally developed by Dekker in a different context, it was applied to the critical section problem by Dijkstra. It allows two threads to share a single-use resource without conflict, using only shared memory for communication. It avoids the strict alternation of an unaffected simplicity of nature or absence of artificiality of turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.

The mutual exclusion requirement is assured. No process will enter its critical section without setting its flag. Every process checks the other flag after setting its own. If both are set, the turn variable is used to allow only one process to proceed. The progress requirement is assured. The turn variable is only considered when both processes are using, or trying to use, the resource. If two processes attempt to enter a critical section at the same time, the algorithm will allow only one process in, based on whose turn it is. If one process is already in the critical section, the other process will busy wait for the first process to exit. This is done by the use of two flags, wants_to_enter[0] and wants_to_enter[1], which indicate an intention to enter the critical section on the part of processes 0 and 1, respectively, and a variable turn that indicates who has priority between the two processes. Dekker's algorithm can be expressed in pseudocode, as follows.

```
variables
    wants_to_enter : array of 2 booleans
    turn : integer

wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0 // or 1
```

```
p0:
  wants_to_enter[0] ← true
  while wants_to_enter[1] {
```

```
p1:
  wants_to_enter[1] ← true
  while wants_to_enter[0] {
```

```
  if turn ≠ 0 {
    wants_to_enter[0] ← false
    while turn ≠ 0 {
      // busy wait
    }
    wants_to_enter[0] ← true
  }
}

// critical section
...
turn ← 1
wants_to_enter[0] ← false
// remainder section
```

```
  if turn ≠ 1 {
    wants_to_enter[1] ← false
    while turn ≠ 1 {
      // busy wait
    }
    wants_to_enter[1] ← true
  }
}

// critical section
...
turn ← 0
wants_to_enter[1] ← false
// remainder section
```

Dekker's algorithm guarantees mutual exclusion, freedom from deadlock, and freedom from starvation.

5.2. 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 we 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 */
}
```

Initially neither process is in 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_region, it will hang there until interested[0] goes to FALSE, an event that only happens when process 0 calls leave_region to exit the critical region.

Now consider the case that both processes call enter_region almost simultaneously. Both will

store their process number in turn. Whichever store is done last is the one that counts; the first one is lost. Suppose that process 1 stores last, so turn is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region.

5.3. TSL (Test and Set Lock)

TSL RX,LOCK

(Test and Set Lock) that works as follows: it reads the contents of the memory word LOCK into register RX and then stores a nonzero value at the memory address LOCK. The operations of reading the word and storing into it are guaranteed to be indivisibleno other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

```
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
```

One solution to the critical region problem is now straightforward. Before entering its critical region, a process calls enter_region, 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. As with all solutions based on critical regions, the processes must call enter_region and leave_region at the correct times for the method to work. If a process cheats, the mutual exclusion will fail.

5.4. Locks:

- A single, shared, (lock) variable, initially 0.
- When a process wants to enter its critical region, it first tests the lock.
- If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region and a 1 means that some process is in its critical region.

Drawbacks:

Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

Problems with mutual Exclusion:

The above techniques achieve the mutual exclusion using busy waiting. Here while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble.

Mutual Exclusion with busy waiting just check to see if the entry is allowed when a process wants to enter its critical region, if the entry is not allowed the process just sits in a tight loop waiting until it is

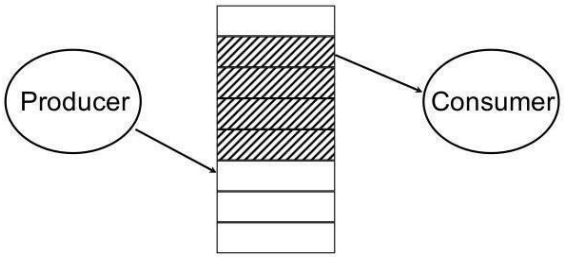
- This approach waste CPU time
- There can be an unexpected problem called priority inversion problem.

6. Sleep and Wakeup:

Sleep and wakeup are system calls that blocks process instead of wasting CPU time when they are not allowed to enter their critical region. sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened.

7. Producer-consumer problem (Bounded Buffer):

Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out.



Problem arises when

1. The producer wants to put a new data in the buffer, but buffer is already full.
   Solution: Producer goes to sleep and to be awakened when the consumer has removed data.
2. The consumer wants to remove data the buffer but buffer is already empty.
   Solution: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

```
#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); count                */ /* put item in buffer */
        = 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 */
    }
}
```

Fig: The producer-consumer problem with a fatal race condition.

N → Size of Buffer
Count → a variable to keep track of the number of items in the buffer.

Producer code:
The producers code is 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 an item and increment count.

Consumer code:
It is similar as of producer. First test count to see if it is 0. If it is, go to sleep; if it nonzero remove an item and decrement the counter.
Each of the process also tests to see if the other should be awakened and if so wakes it up.
This approach sounds simple enough, but it leads to the same kinds of race conditions as we saw in the spooler directory.

1. The buffer is empty and the consumer has just read count to see if it is 0.
2. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. (Consumer is interrupted and producer resumed)

- The producer creates an item, puts it into the buffer, and increases count.
- Because the buffer was empty prior to the last addition (count was just 0), the producer

tries to wake up the consumer.

- Unfortunately, the consumer is not yet logically asleep, so the **wakeup signal** is lost.
- When the consumer next runs, it will test the value of count it previously read, find 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.

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. For temporary solution we can use wakeup waiting bit to prevent wakeup signal from getting lost, but it can't work for more processes.

8. Semaphore:

In computer science, a semaphore is a protected variable or abstract data type that constitutes a classic method of controlling access by several processes to a common resource in a parallel programming environment. Synchronization tool does not require busy waiting. **A semaphore is a special kind of integer variable which can be initialized and can be accessed only through two atomic operations. P and V. If S is the semaphore variable, then,**

P operation: Wait for semaphore to become positive and then decrement

P(S):

While (S<=0)

do no-op;

S = S-1

V Operation: Increment semaphore by 1

V(S):

S=S+1;

Atomic operations: When one process modifies the semaphore value, no other process can simultaneously modify that same semaphores value. In addition, in case of the P(S) operation the testing of the integer value of S (S<=0) and its possible modification (S=S-1), must also be executed without interruption.

Semaphore operations:

- P or Down, or Wait
- V or Up or Signal

Counting semaphore – integer value can range over an unrestricted domain

Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement

Also known as mutex locks

Advantages of semaphores:

- Processes do not busy wait while waiting for resources. While waiting, they are in a "suspended" state, allowing the CPU to perform other chores.
- Works on (shared memory) multiprocessor systems.
- User controls synchronization.

Disadvantages of semaphores:

- ❖ They can only be invoked by processes not interrupt service routines.
- ❖ The user controls synchronization and they could mess up.

9. Monitors:

In concurrent programming, a **monitor** is an object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared to reasoning about parallel code that updates a data structure.

Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met.

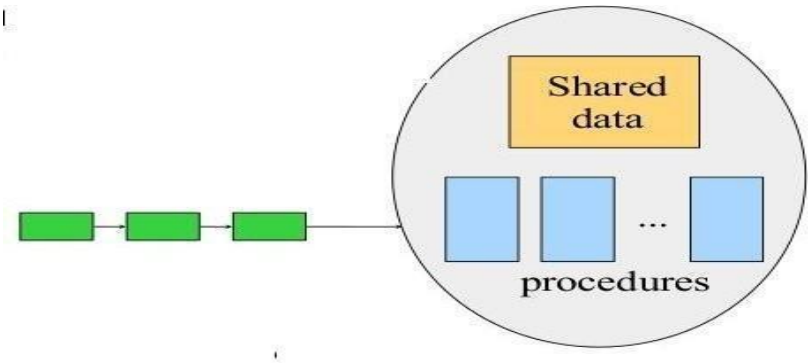


Fig: Queue of waiting processes trying to enter the monitor

With semaphores IPC seems easy, but suppose that the two downs in the producer's code were reversed in order, so mutex was decremented before empty instead of after it. If the buffer were completely full, the producer would block, with mutex set to 0. Consequently, the next time the consumer tried to access the buffer, it would do a down on mutex, now 0, and block too. Both processes would stay blocked forever and no more work would ever be done. This unfortunate situation is called a deadlock.

Monitors can be defined as follows;

- A higher level synchronization primitive.
- A monitor is a collection of procedures, variables, and data structures that are 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.

monitor example

```
integer i;
condition c;
procedure producer (x);
  .
  .
  .
 end;
 procedure consumer (x);
 .
 .
 end;
end monitor;
```

Fig: A monitor

10. Message Passing:

Message passing in computer science, is a form of communication used in parallel computing, object-oriented programming, and interprocess communication. In this model processes or objects can send and receive messages (comprising zero or more bytes, complex data structures, or even segments of code) to other processes. By waiting for messages, processes can also synchronize.

Message passing is a method of communication where messages are sent from a sender to one or more recipients. Forms of messages include **(remote) method invocation, signals, and data packets**. When designing a message passing system several choices are made:

- Whether messages are transferred reliably
- Whether messages are guaranteed to be delivered in order
- Whether messages are passed one-to-one, one-to-many (multicasting or broadcasting), or many-to-one (client–server).
- Whether communication is synchronous or asynchronous.

This method of inter-process 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);
```

Synchronous message passing systems requires the sender and receiver to wait for each other to transfer the message. Asynchronous message passing systems deliver a message from

sender to receiver, without waiting for the receiver to be ready.

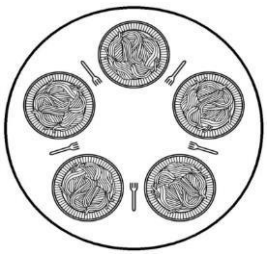
11. Classical IPC Problems

A. Dining Philosophers Problem
B. The Readers and Writers Problem
C. The Sleeping Barber Problem

11.1. Dining philosophers problems:

There are N philosphers sitting around a circular table eating spaghetti and discussing philosphy. The problem is that each philosopher needs 2 forks to eat, and there are only N forks, one between each 2 philosophers. Design an algorithm that the philosophers can follow that insures that none starves as long as each philosopher eventually stops eating, and such that the maximum number of philosophers can eat at once.

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock



The problem was designed to illustrate the problem of avoiding deadlock, a system state in which no progress is possible. One idea is to instruct each philosopher to behave as follows:

- think until the left fork is available; when it is, pick it up
- think until the right fork is available; when it is, pick it up
- eat
- put the left fork down
- put the right fork down
- repeat from the start

This solution is incorrect: it allows the system to reach deadlock. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

We could modify the program so that after taking 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, although for a different reason. With a little bit of 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. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called starvation

The solution presented below is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses 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 only if neither neighbor is eating. Philosopher i's neighbors are defined by the macros LEFT and RIGHT. In other words, if i is 2, LEFT is 1 and RIGHT is 3.

Solution:

```
#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 N1 */
{
    while (TRUE){               /* repeat forever */
        think();                /* philosopher is thinking */
        take_forks(i);          /* acquire two forks or block */
        eat();                  /* yum-yum, spaghetti */
        put_forks(i);           /* put both forks back on table */
    }
}
void take_forks(int i)          /* i: philosopher number, from 0 to N1 */
{
    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 N1 */
{
    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 N1* /
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
        EATING) { state[i] = EATING;
        up(&s[i]);
    }
```

}

11.2. Readers Writer problems:

The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem which models access to a database. Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other process may have access to the database, not even a reader. The question is how do we program the readers and the writers? One solution is shown below.

Solution to Readers Writer problems

```
typedef int semaphore;                /* use our 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 */
    }
}
```

In this solution, the first reader to get access to the data base does a down on the semaphore db. Subsequent readers merely have to increment a counter, rc. As readers leave, they decrement the

counter and the last one out does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

11.3. Sleeping Barber Problem

Customers arrive to a barber, if there are no customers the barber sleeps in his chair. If the barber is asleep then the customers must wake him up.

The analogy is based upon a hypothetical barber shop with one barber. The barber has one barber chair and a waiting room with a number of chairs in it. When the barber finishes cutting a customer's hair, he dismisses the customer and then goes to the waiting room to see if there are other customers waiting. If there are, he brings one of them back to the chair and cuts his hair. If there are no other customers waiting, he returns to his chair and sleeps in it.

Each customer, when he arrives, looks to see what the barber is doing. If the barber is sleeping, then the customer wakes him up and sits in the chair. If the barber is cutting hair, then the customer goes to the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits his turn. If there is no free chair, then the customer leaves. Based on a naive analysis, the above description should ensure that the shop functions correctly, with the barber cutting the hair of anyone who arrives until there are no more customers, and then sleeping until the next customer arrives. In practice, there are a number of problems that can occur that are illustrative of general scheduling problems.

The problems are all related to the fact that the actions by both the barber and the customer (checking the waiting room, entering the shop, taking a waiting room chair, etc.) all take an unknown amount of time. For example, a customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. While he is on his way, the barber finishes

the haircut he is doing and goes to check the waiting room. Since there is no one there (the customer not having arrived yet), he goes back to his chair and sleeps. The barber is now waiting for a customer and the customer is waiting for the barber. In another example, two customers may arrive at the same time when there happens to be a single seat in the waiting room. They observe that the barber is cutting hair, go to the waiting room, and both attempt to occupy the single chair.

Solution:

Many possible solutions are available. The key element of each is a mutex, which ensures that only one of the participants can change state at once. The barber must acquire this mutex exclusion before checking for customers and release it when he begins either to sleep or cut hair. A customer must acquire it before entering the shop and release it once he is sitting in either a waiting room chair or the barber chair. This eliminates both of the problems mentioned in the previous section. A number of semaphores are also required to indicate the state of the system. For example, one might store the number of people in the waiting room.

A multiple sleeping barbers problem has the additional complexity of coordinating several barbers among the waiting customers