**3.4 Inter-process Communication (IPC) & Synchronization**

**3.4.1 Introduction**

***Inter-process Communication***

Processes frequently need to communicate with each other. For example in a shell pipeline, the output of the first process must be passed to the second process and so on down the line. Thus there is a need for communication between the processes, preferably in a well-structured way not using the interrupts.

IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. Inter-process communication (IPC) is a set of

techniques for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computers connected by a network.

Co-operating process requires IPC. There are two fundamental ways of IPC.

**a. Shared Memory**

**b. Message Passing**



***Fig: Communication Model (a) Message Passing (b) Shared Memory***

1. **Shared Memory**

* Here a region of memory that is shared by co-operating process is established.
* Process can exchange the information by reading and writing data to the shared region.
* Shared memory allows maximum speed and convenience of communication as it can be done at the speed of memory within the computer.
* System calls are required only to establish shared memory regions. Once shared memory is established no assistance from the kernel is required, all accesses are treated as routine memory access.

1. **Message Passing**

* Communication takes place by means of messages exchanged between the co-operating

process.

* Message passing is useful for exchanging the smaller amount of data since no conflict need to be avoided.
* Easier to implement than shared memory.
* Slower than that of Shared memory as message passing system are typically implemented using system call which requires more time consuming task of Kernel intervention.

**3.4.2 Race Condition**



***Figure: Two processes want to access shared memory at the same time***

The situation where two or more processes are reading or writing some shared data & the final results depends on who runs precisely and when, are called **race conditions.**

To see how inter-process communication works in practice, let us consider a simple but common example, a print spooler. When a process wants to print a file, it enters the file name 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 removes their names from the directory.

Imagine that our spooler directory has a large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables,

**out:** which points to the next file to be printed

**in:** which points to the next free slot in the directory.

At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files to be printed). More or less simultaneously, processes A and B decide they want to queue a file for printing as shown in the fig.

Process A reads in and stores the value, 7, in a local variable called **next\_free\_slot**. Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B.

Process B also reads in, and also gets a 7, so it stores the name of its file in slot 7 and updates in to be an 8. Then it goes off and does other things.

Eventually, process A runs again, starting from the place it left off last time. It looks at **next\_free\_slot**, finds a 7 there, and writes its file name in slot 7, erasing the name that process B just put there. Then it computes **next\_free\_slot + 1**, which is 8, and sets **in** to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output.

*Spooling: Simultaneous peripheral operations online*

**3.4.3 Critical Section (or critical region)**

To avoid race condition we need **Mutual Exclusion (MUTEX). Mutual Exclusion** is some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same things.

That part of the program where the shared memory is accessed is called the **critical region or critical section**. If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid race conditions. Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data.

**(Rules for avoiding Race Condition) Solution to Critical section problem:**

1. No two processes may be simultaneously inside their critical regions. (MUTEX)

2. No assumptions may be made about speeds or the number of CPUs.

3. No process running outside its critical region may block other processes.

4. No process should have to wait forever to enter its critical region.

****

***Fig: Mutual Exclusion using Critical Region***

**3.4.5 Proposals for achieving Mutual Exclusion (Techniques for**

**avoiding Race Condition)**

1. Disabling Interrupts

2. Lock Variables

3. Strict Alteration

4. Peterson's Solution

5. TSL instruction

6. Sleep and Wakeup

7. Semaphores

8. Monitors

9. Message Passing

1. **Disabling Interrupts**

The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur.

The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts 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 memory without fear that any other process will intervene.

***Disadvantages:***

1. It is unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did, and then never turned them on again?
2. Furthermore, if the system is a 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.

***Advantages:***

It is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists. If an interrupt occurred while the list of ready processes, for example, was in an inconsistent state, race conditions could occur.

1. **Lock Variables**

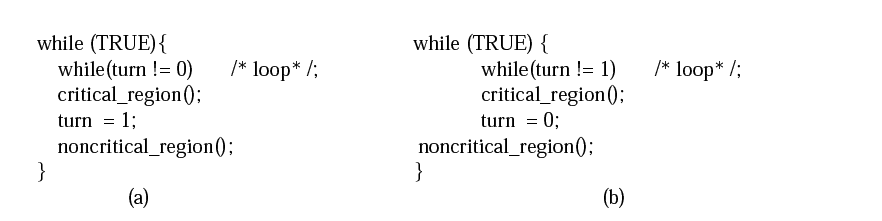
* A single, shared, (lock) variable, initially 0.
* When a process wants to enter its critical region, it first tests the lock.
* If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process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

***Drawbacks:***

Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory.

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.

1. **Strict Alteration**



***Figure: A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.***

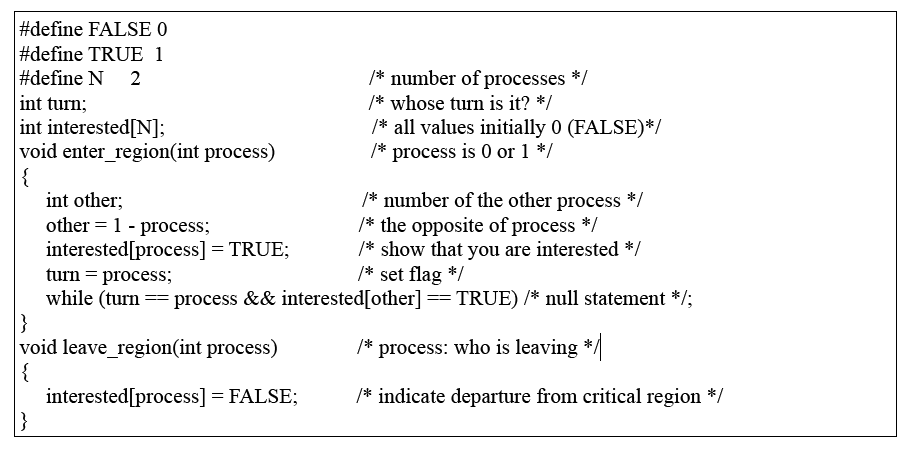
Processes share a common integer variable turn. If turn == i then process Pi is allowed to execute in its CR, if turn == j then process Pj is allowed to execute.

**Advantages:** Ensures that only one process at a time can be in its CR.

**Problems:** Strict alternation of processes in the execution of the CR.

What happens if process i just finished CR and again need to enter CR and the process j is still busy at non-CR work? (violate condition 3)

1. **Peterson's Solution**



***Figure: Peterson's solution for achieving mutual exclusion.***

* Before entering its CR, each process call *enter\_region()* with its own process number, 0 or 1 as parameter.
* Call will cause to wait, if need be, until it is safe to enter.
* When leaving CR, the process calls *leave\_region()* to indicate that it is done and to allow other process to enter CR.

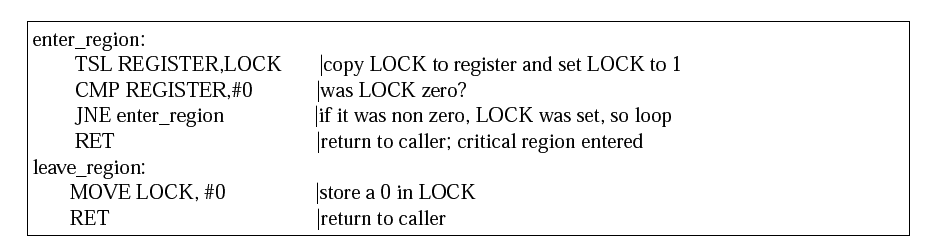
**Advantage:** Preserves all conditions.

**Problems:** Difficult to program for n-processes system and less efficient.

1. **The TSL Instruction**

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 indivisible no 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.



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.

**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.

***Examples to use Sleep and Wakeup primitives:***

**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.



Trouble arises when

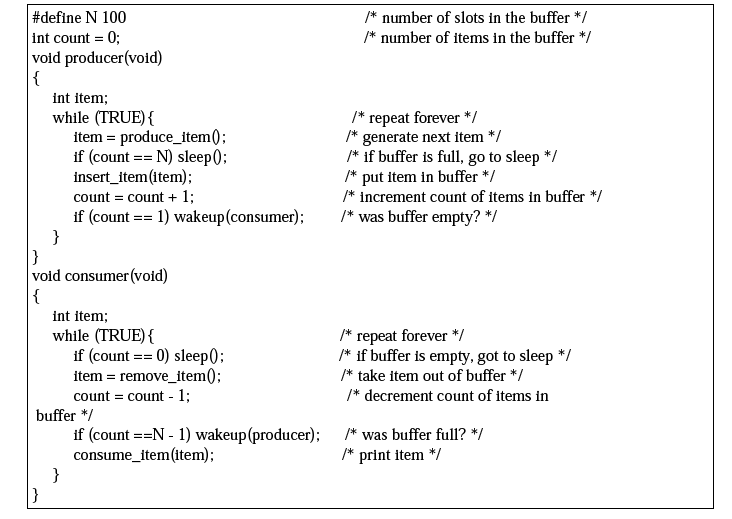
**1.** The producer wants to put a new data in the buffer, but buffer is already full.

**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.



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

N → Size of Buffer

Count--> a variable to keep track of the no. of items in the buffer.

**Producers 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)

3. The producer creates an item, puts it into the buffer, and increases count.

4. Because the buffer was empty prior to the last addition (count was just 0), the producer tries to

wake up the consumer.

5. Unfortunately, the consumer is not yet logically asleep, so the **wakeup signal** is lost.

6. When the consumer next runs, it will test the value of count it previously read, find it to be 0,

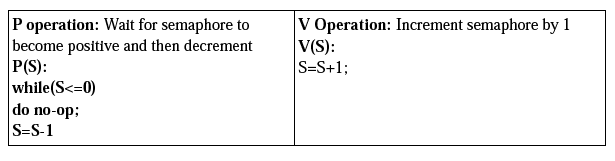
and go to sleep.

7. 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.

1. **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 that 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,*

**Semaphore operations:** 

***P or Down, or Wait:*** P stands for *proberen* for "to test

***V or Up or Signal:*** Dutch words. **V** stands for *verhogen* ("increase")

***wait(sem):*** Decrement the semaphore value. if negative, suspend the process and place in queue.

(Also referred to as *P(), down* in literature.)

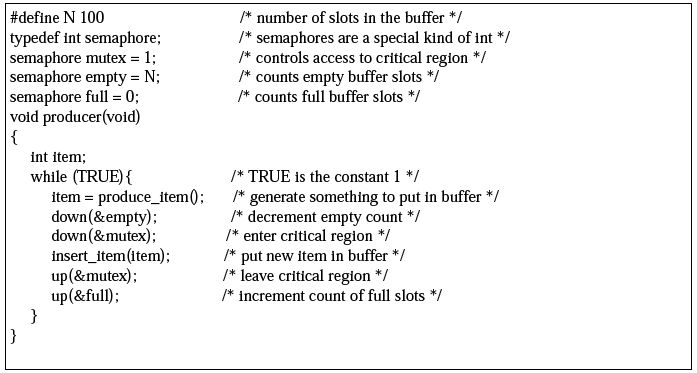
***signal(sem):*** Increment the semaphore value, allow the first process in the queue to continue.

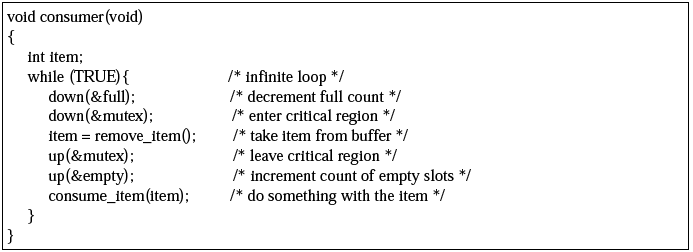
(Also referred to as *V(), up* in literature.)

**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





***Fig: The producer-consumer problem using semaphores.***

This solution uses three semaphores.

1. **Full:** For counting the number of slots that are full, initially 0
2. **Empty:** For counting the number of slots that are empty, initially equal to the no. of slots

in the buffer

1. **Mutex:** To make sure that the producer and consumer do not access the buffer at the

same time, initially 1.

Here in this example semaphores are used in two different ways.

**1. For mutual Exclusion**: The mutex semaphore is for mutual exclusion. It is designed to

guarantee that only one process at a time will be reading or writing the buffer and the

associated variable.

**2. For synchronization:** The full and empty semaphores are needed to guarantee that certain

certain event sequences do or do not occur. In this case, they ensure that producer stops

running when the buffer is full and the consumer stops running when it is empty.

The above definition of the semaphore suffers the problem of busy wait. To overcome the need for busy waiting, we can modify the definition of the P and V operation of the semaphore. When a Process executes the P operation and finds that the semaphores value is not positive, it must wait. However, rather than busy waiting, the process can block itself.

**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:**

• Can only be invoked by processes--not interrupt service routines because interrupt routines

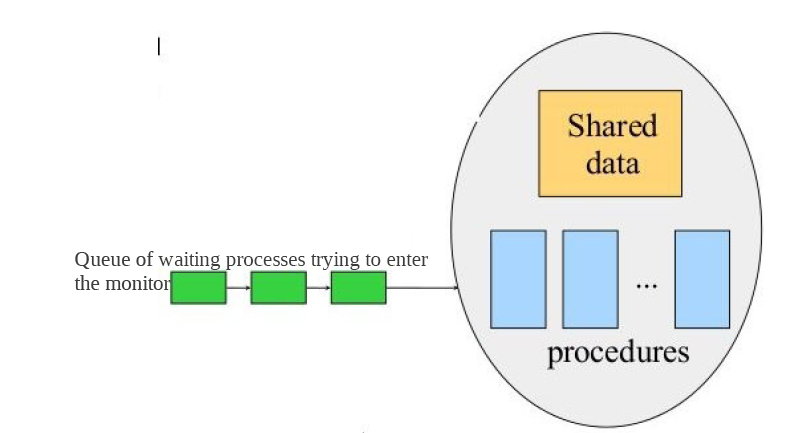
cannot block.

• user controls synchronization--could mess up.

1. **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.



● 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.

● This rule, which is common in modern object-oriented languages such as Java, was relatively

unusual for its time,

1. **Message Passing**

Message passing in computer science, is a form of communication used in parallel computing, object oriented programming, and inter-process 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

Semaphore 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.

***Producer-Consumer Problem with Message Passing***

#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. Dining Philosophers Problem

2. The Readers and Writers Problem

3. The Sleeping Barber Problem

**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]);

}

}

1. **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 (Courtois et al., 1971). 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 you program the readers and the writers? One solution is shown below.

**Solution to Readers Writer problems**

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 \*/

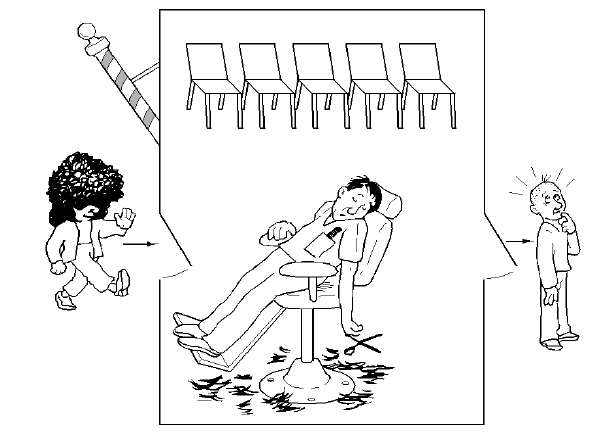
}

}

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.

1. **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.