



OPERATING SYSTEMS

INTERPROCESS  
COMMUNICATION

## 4 Interprocess Communication

2

Processes frequently need to communicate with other processes. 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 processes, preferably in a well-structured way not using interrupts. In this week, we will look at some of the issues related to this Interprocess Communication (IPC).

## 4 Interprocess Communication

3

Briefly, there are three issues here.

- How one process can pass information to another.
- Making sure two or more processes do not get in each other's way.
- Proper sequencing when dependencies are present.

## 4.1 Race Conditions

4

In some operating systems, processes that are working together may share some common storage that each one can read and write. To see how IPC works in practice, let us consider a simple 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 then removes their names from the directory.

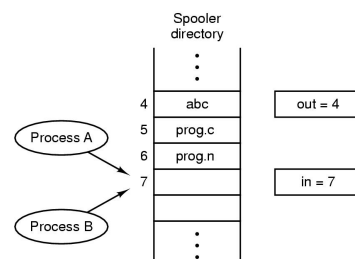
## 4.1 Race Conditions

5

Imagine that our spooler directory has a very 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 and in) which points to the next file to be printed, and to the next free slot in the directory, respectively. These variables might well be kept on a two-word file available to all processes. At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full. More or less simultaneously, processes A and B decide they want to queue a file for printing.

## 4.1 Race Conditions

6



Two processes want to access shared memory at same time

## 4.1 Race Conditions

7

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. It too stores it in its local variable **next\_free\_slot**. At this instant both processes think that the next available slot is 7. Process B now continues to run. It stores the name of its file in slot 7 and updates in to be an 8.

## 4.1 Race Conditions

8

Process A runs again, starting from the place it left off. It writes its file name in slot 7, erasing the name that process B just put there. Then it 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. Situations like this, 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**.

## 4.2 Critical Regions

9

Sometimes a process has to access shared memory or files, or do other critical things that can lead to races. That part of the program where the shared memory is accessed is called the **critical region**.

How do we avoid race conditions? **Mutual exclusion** is a solution which describes some way of making sure that if one process is using a shared data, the other processes will be excluded from doing the same thing.

## 4.2 Critical Regions

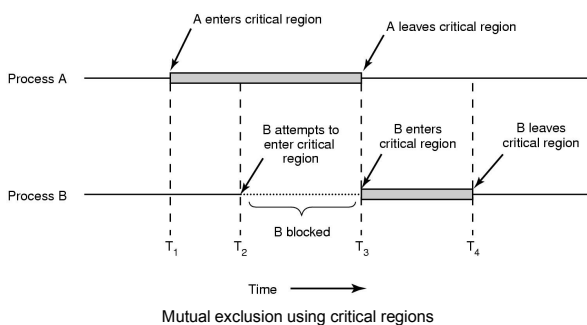
10

Four conditions to provide mutual exclusion:

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

## 4.2 Critical Regions

11



## 4.3 Mutual Exclusion

12

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

### 4.3.1 Disabling Interrupts

13

On a single-processor system, 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.

### 4.3.1 Disabling Interrupts

14

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 (with two or possibly 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.

### 4.3.1 Disabling Interrupts

15

On the other hand, 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. 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.

### 4.3.1 Disabling Interrupts

16

The possibility of achieving mutual exclusion by disabling interrupts -even within the kernel- is becoming less every day due to the increasing number of multi core chips even in low-end PCs. Two cores are already common, four are present in high-end machines, and eight or 16 are not far behind. In a multiprocessor system, disabling the interrupts of one CPU does not prevent other CPUs from interfering with operations the first CPU is performing. Consequently, more sophisticated schemes are needed.

### 4.3.2 Lock Variables

17

As a second attempt, let us look for a software solution. Consider having 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.

### 4.3.2 Lock Variables

18

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.

## 4.3.2 Lock Variables

19

Now you might think that we could get around this problem by first reading out the lock value, then checking it again just before storing into it, but that really does not help. The race now occurs if the second process modifies the lock just after the first process has finished its second check.

## 4.3.3 Strict Alternation

20

A third approach to the mutual exclusion problem is shown in figure.

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

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

## 4.3.3 Strict Alternation

21

In figure, the integer variable “turn” keeps track of whose turn it is to enter the critical region. Initially, Process0 inspects “turn”, finds it to be 0, and enters its critical region. Process1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1. Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time. Only when the wait will be short, using busy waiting may be reasonable. A lock that uses busy waiting is called a **spin lock**.

## 4.3.3 Strict Alternation

22

When Process0 leaves the critical region, it sets “turn” to 1, to allow Process1 to enter its critical region. Suppose that Process1 finishes its critical region quickly, so that both processes are in their noncritical regions, with “turn” set to 0. Now Process0 executes its whole loop quickly, exiting its critical region and setting “turn” to 1. At this point “turn” is 1 and both processes are executing in their noncritical regions.

## 4.3.3 Strict Alternation

23

Suddenly, Process0 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now, because “turn” is 1 and Process1 is busy with its noncritical region. It hangs in its while loop until Process1 sets “turn” to 0. Put differently, taking turns is not a good idea when one of the processes is much slower than the other. This algorithm can not be a solution because it violates condition 3.

## 4.3.4 Peterson's Solution

24

By combining the idea of taking turns with the idea of lock variables and warning variables, Peterson discovered a much simpler way to achieve mutual exclusion in 1981. Before entering the critical region, each process calls “enter\_region” with its own process 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 the other process to enter, if it so desires.

## 4.3.4 Peterson's Solution

25

```
#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 */
}
```

## 4.3.4 Peterson's Solution

26

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 Process1 is not interested, "enter\_region" returns immediately. If Process1 now makes a call to "enter\_region", it will hang there until "interested[0]" goes to FALSE, an event that only happens when Process0 calls "leave\_region" to exit the critical region.

## 4.3.4 Peterson's Solution

27

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 overwritten and lost. Suppose that Process1 stores last, so "turn" is 1. When both processes come to the while statement, Process0 executes it zero times and enters its critical region. Process1 loops and does not enter its critical region until Process0 exits its critical region.

## 4.3.5 The TSL Instruction

28

Some computers, especially those designed with multiple processors in mind, have an instruction like TSL (Test and Set Lock). 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.

## 4.3.5 The TSL Instruction

29

```
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
```

Entering and leaving a critical region using the TSL instruction

## 4.3.5 The TSL Instruction

30

It is important to note that locking the memory bus is very different from disabling interrupts. Disabling interrupts then performing a read on a memory word followed by a write does not prevent a second processor on the bus from accessing the word between the read and the write. In fact, disabling interrupts on processor 1 has no effect at all on processor 2. The only way to keep processor 2 out of the memory until processor 1 is finished is to lock the bus, which requires a special hardware system.

### 4.3.5 The TSL Instruction

31

To use the TSL instruction, we will use a shared variable ("lock") to coordinate 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.

### 4.3.5 The TSL Instruction

32

To prevent two processes from simultaneously entering their critical regions, there are four-instruction subroutines. The first instruction copies the old value of "lock" to the register and then sets "lock" to 1. Then the old value is compared with 0. If it is nonzero, the lock was already set, so the program just goes back to the beginning and tests it again. When the process is done with its critical region, it will become 0, and the subroutine returns, with the lock set. For clearing the lock, the program just stores a 0 in "lock".

### 4.4 Sleep and Wakeup

33

Both Peterson's solution and the solutions using TSL are correct, but both have the defect of requiring busy waiting. In essence, what these solutions do is this: when a process wants to enter its critical region, 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. Not only does this approach waste CPU time, but it can also have unexpected effects.

### 4.4 Sleep and Wakeup

34

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**.

### 4.4 Sleep and Wakeup

35

Now let us look at some interprocess communication primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions. One of the simplest is the pair "sleep" and "wakeup". "Sleep" is a system call that causes the caller to block, that is, be suspended until another process wakes it up. "Wakeup" call has one parameter, the process to be awakened.

### 4.4 Sleep and Wakeup

36

Let us use these primitives in an example of the **producer-consumer** problem. Two processes share a fixed-size buffer. One of them, the producer, puts data 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 has removed an item. If the consumer wants to remove an item from the buffer when the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

## 4.4 Sleep and Wakeup

37

This approach may lead to the same kinds of race conditions. To keep track of the number of items in the buffer, we will use a variable, count. If the maximum number of items the buffer can hold is N, producer's code will first test to see if count is N. If it is not, the producer will add an item and increment count; if it is, producer will go to sleep. Consumer's code is similar: first test count to see if it is 0. If it is, go to sleep; if it is nonzero, remove an item and decrement the counter. Each of the processes also tests to see if the other should be awakened, and if so, wakes it up.

## 4.4 Sleep and Wakeup

38

```
#define N 100 /* number of slots in the buffer */
int count = 0; /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) { /* repeat forever */
        item = produce_item(); /* generate next item */
        if (count == N) sleep(); /* if buffer is full, go to sleep */
        insert_item(item); /* put item in buffer */
        count = count + 1; /* increment count of items in buffer */
        if (count == 1) wakeup(consumer); /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) { /* repeat forever */
        if (count == 0) sleep(); /* if buffer is empty, got to sleep */
        item = remove_item(); /* take item out of buffer */
        count = count - 1; /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item); /* print item */
    }
}
```

## 4.4 Sleep and Wakeup

39

A race condition can occur because access to count is unconstrained; e.g. the buffer is empty and the consumer has just read count to see if it is 0. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. The producer inserts an item in the buffer, increments count, and notices that it is now 1. Reasoning that count was just 0, and thus the consumer must be sleeping, the producer calls wakeup to wake the consumer up.

## 4.4 Sleep and Wakeup

40

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. Let us try to fix by a **wakeup waiting bit**. When a wakeup is sent to a process that is still awake, this bit is set. Later, when the process tries to go to sleep, if the wakeup waiting bit is on, it will be turned off, but the process will stay awake. One (or more) wakeup waiting bit does not work.

## 4.5 Semaphores

41

This was the situation in 1965, when Dijkstra suggested using an integer variable to count the number of wakeups saved for future use. In his proposal, a new variable type, which he called a **semaphore**, was introduced. A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending. Dijkstra proposed having two operations, down and up (generalizations of sleep and wakeup).

## 4.5 Semaphores

42

The down operation checks to see if the value is greater than 0. If so, it decrements the value and just continues. 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, are all done as a single, indivisible **atomic action**. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked.

## 4.5 Semaphores

43

The up operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system and is allowed to complete its down. Thus, after an up on a semaphore with processes sleeping on it, the semaphore will still be 0, but there will be one fewer process sleeping on it. The operation of incrementing the semaphore and waking up one process is also indivisible.

## 4.5 Semaphores

44

Semaphores can be used in the lost-wakeup problem. The way to work them correctly is to implement up and down as system calls, with the operating system briefly disabling all interrupts while it is testing the semaphore, updating it, and putting the process to sleep, if necessary. As all of these actions take only a few instructions, no harm is done in disabling interrupts. If multiple CPUs are being used, each semaphore should be protected by a lock variable, with the TSL instructions used to make sure that only one CPU at a time examines the semaphore.

## 4.5 Semaphores

45

```
/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&mutex);
        down(&empty);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

## 4.5 Semaphores

46

This solution uses three semaphores; "full" for counting the number of full slots, "empty" for counting the number of empty slots, and "mutex" to make sure produce and consumer don't access the buffer at the same time. Initially, "full" is 0, "empty" is the number of slots in the buffer, and "mutex" is 1. Semaphores used by two or more processes, where only one of them can enter its critical region at the same time are called **binary semaphores**. If a process uses down and up correctly, mutual exclusion is guaranteed.

## 4.5 Semaphores

47

Here, semaphores are used in two different ways. At first is for mutual exclusion with the mutex semaphore. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables. The other use of semaphores is for synchronization. The full and empty semaphores are needed to guarantee that certain event sequences do or do not occur. In this case, they ensure that the producer stops running when the buffer is full, and that the consumer stops running when it is empty.

## 4.6 Mutexes

48

When the semaphore's ability to count is not needed, a simplified version of the semaphore, called a mutex, is sometimes used. Mutexes are good only for managing mutual exclusion to some shared resource or piece of code. They are easy and efficient to implement, which makes them especially useful in thread packages that are implemented entirely in user space.



## 4.6 Mutexes

49

A mutex is a variable that can be in one of two states: unlocked or locked. When a process (or thread) needs access to a critical region, it calls `mutex_lock`. If the mutex is currently unlocked, the call succeeds and the calling thread is free to enter the critical region. On the other hand, if the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls `mutex_unlock`. If multiple threads are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.

## 4.6 Mutexes

50

```
mutex_lock:
    TSL REGISTER,MUTEX           | copy mutex to register and set mutex to 1
    CMP REGISTER,#0             | was mutex zero?
    JZE ok                       | if it was zero, mutex was unlocked, so return
    CALL thread_yield           | mutex is busy; schedule another thread
    JMP mutex_lock              | try again later
ok: RET | return to caller; critical region entered

mutex_unlock:
    MOVE MUTEX,#0               | store a 0 in mutex
    RET | return to caller
```

Implementation of `mutex_lock` and `mutex_unlock`

## 4.7 Monitors

51

In source code of semaphore, 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.

## 4.7 Monitors

52

This problem is pointed out to show how careful you must be when using semaphores. To make it easier to write correct programs, a higher-level synchronization primitive is proposed, called a monitor. 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 monitor.

## 4.7 Monitors

53

Monitors have an important property: only one process can be active in a monitor at any instant. Monitor is a programming language construct, so the compiler knows it is special and can handle calls to monitor procedures. Typically, when a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active 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.

## 4.7 Monitors

54

It is up to the compiler to implement mutual exclusion on monitor entries, but a common way is to use a binary semaphore. Because the compiler is arranging for the mutual exclusion, the error probability is very small. It is sufficient to know that if all the critical regions are turned into monitor procedures, no two processes will ever execute their critical regions at the same time. But, how should the producer block when it finds the buffer full?

## 4.7 Monitors

55

The solution lies in two operations, wait and signal. When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a wait on some condition variables, say, "full". This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now. This other process, for example, the consumer, can wake up its sleeping partner by doing a signal on the condition variable that its partner is waiting on.

## 4.7 Monitors

56

To avoid having two active processes in the monitor at the same time, a process doing a signal must exit the monitor immediately. In other words, a signal statement may appear only as the final statement in a monitor procedure. If a signal is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler, is revived. Another solution is to let the signaler continue to run and allow the waiting process to start running only after the signaler has exited the monitor.

## 4.7 Monitors

57

```
monitor ProducerConsumer
condition full, empty;
integer count;
procedure insert(item: integer);
begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty);
end;
function remove: integer;
begin
    if count = 0 then wait(empty);
    remove := remove_item;
    count := count - 1;
    if count = N - 1 then signal(full);
end;
count := 0;
end monitor;

procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item);
    end;
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item);
    end;
end;
```

Outline of producer-consumer problem with monitors

## 4.7 Monitors

58

A problem with monitors (also semaphores) is that they were designed for solving the mutual exclusion problem on one or more CPUs that all have access to a common memory. By using the semaphores, we can avoid races. When we go to a distributed system consisting of multiple CPUs, each with its own private memory, connected by a local area network, these primitives become inapplicable. Also, none of the primitives allow information exchange between machines. Something else is needed.

## 4.8 Message Passing

59

This IPC method uses two primitives, "send" and "receive", which, like semaphores and unlike monitors, are system calls rather than language constructs. The former call sends a message to a given destination and the latter one receives a message from a given source. If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

## 4.8 Message Passing

60

Message passing systems have many challenging problems and design issues that do not arise with semaphores or with monitors, especially if the communicating processes are on different machines connected by a network. For example, messages can be lost by the network. To guard against lost messages, sender and receiver can agree that as soon as a message has been received, the receiver will send back a special acknowledgement message. If the sender has not received **acknowledgement** (ack) in a certain time interval, it retransmits the message.

## 4.8 Message Passing

61

If message is received correctly, but acknowledgement back to the sender is lost. The sender will retransmit the message, so the receiver will get it twice. Usually, this problem is solved by putting consecutive sequence numbers in each original message. If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a duplicate that can be ignored.

## 4.8 Message Passing

62

Message systems also have to deal with the question of how processes are named, so that the process specified in a send or receive call is unambiguous. Authentication is also an issue in message systems: how can the client tell that it is communicating with the real file server, and not with an imposter? Another design issue occurs when the sender and receiver are on the same machine. Copying messages from one process to another is always slower than doing a semaphore operation or entering a monitor.

## 4.8 Message Passing

63

The producer-consumer problem can be solved with message passing. We assume that all messages are the same size and that messages sent but not yet received are buffered automatically by the operating system. The buffer can keep N messages. Consumer starts out by sending N empty messages to the producer. Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one. In this way, the messages can be stored in a given amount of memory known in advance.

## 4.8 Message Passing

64

```
#define N 100          /* number of slots in the buffer */
void producer(void)
{
    int item;
    message m;        /* message buffer */

    while (TRUE) {
        item = produce_item(); /* generate something to put in buffer */
        receive(consumer, &m); /* wait for an empty to arrive */
        build_message(&m, item); /* construct a message to send */
        send(consumer, &m); /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

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

## 4.8 Message Passing

65

If the producer works faster than the consumer, all the messages will end up full, waiting for the consumer; the producer will be blocked, waiting for an empty to come back. If the consumer works faster, then the reverse happens: all the messages will be empties waiting for the producer to fill them up; the consumer will be blocked, waiting for a full message.

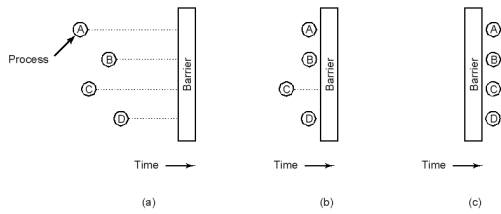
## 4.9 Barriers

66

Our last synchronization mechanism is intended for groups of processes rather than two-process producer-consumer type situations. Some applications are divided into phases and have the rule that no process may proceed into the next phase until all processes are ready to proceed to the next phase. This behavior may be achieved by placing a barrier at the end of each phase. When a process reaches the barrier, it is blocked until all processes have reached the barrier.

## 4.9 Barriers

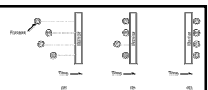
67



- (a) processes approaching a barrier,
- (b) all processes but one blocked at barrier,
- (c) last process arrives, all are let through

## 4.9 Barriers

68



In Figure, we see four processes approaching a barrier. What this means is that they are just computing and have not reached the end of the current phase yet. After a while, the first process finishes all the computing required of it during the first phase. It then executes the barrier primitive, generally by calling a library procedure. The process is then suspended. A little later, a second and then a third process finish the first phase and also execute the barrier primitive. Finally, when the last process, C, hits the barrier, all the processes are released.