select one to be executed in some fair manner (such as first-come-first-serve). If only one of the conditions is true, and the procedure named in an `accept` statement in the body of the `when` statement is open, that one will be executed. If both of the `when` conditions are false, an error condition occurs (this usually terminates the process.)
Producer Consumer Problem

Introduction
This algorithm uses ADA to solve the producer/consumer (or bounded-buffer) problem.

Algorithm
This process (task, to ADA) manages the buffer.

```ada
1 task boundedbuffer is
2    entry deposit(data: in item);
3    entry extract(data: out item);
4 end;
5 task body boundedbuffer is
6    buffer: array[0..n-1] of item;
7    count: integer range 0..n := 0;
8    in, out: integer range 0..n-1 := 0;
9 begin
10    loop
11       select
12          when count < n =>
13             accept deposit(data: in item) do
14                buffer[in] := data;
15                 in := (in + 1) mod n;
16                 count := count + 1;
17          or when count > 0 =>
18             accept extract(data: out item) do
19                data := buffer[out];
20                  out := (out + 1) mod n;
21                 count := count - 1;
22       end select;
23 end loop;
24 end.
```

The producer deposits an item into the buffer with
```
boundedbuffer.deposit(nextp);
```
and the consumer extracts an item from the buffer with
```
boundedbuffer.extract(nextc);
```

Comments
lines 1-4 This indicates that the procedures deposit and extract may be called outside the task, and that extract will return something in its parameter list (the out).

lines 6-8 As usual, buffer is the buffer, and count the number of items currently in the buffer; in and out are the indices indicating where deposits go or where extractions come from.

lines 13-17 If there is room in the buffer (when count < n) this process will accept a request to deposit an item in it (accept deposit ...); it then updates its variables.

lines 18-23 If there is an item in the buffer (when count > 0) this process will accept a request to extract an item from the buffer (accept extract ...); the item is returned via the parameter list. This procedure then updates its variables.

line 24 If both of the above two when conditions are true, and both a producer and consumer has invoked a procedure named by an accept statement (called “an open accept statement”), the system will
Producer Consumer Problem

Introduction

This algorithm uses blocking send and receive primitives to solve the producer/consumer (or bounded-buffer) problem. In this solution, the buffer size depends on the capacity of the link.

Algorithm

```
1 var nextp, nextc: item;
2 procedure producer;
3 begin
4 while true do begin
5 (* produce item in nextp *)
6 send("Consumerprocess", nextp);
7 end;
8 end;
9 procedure consumer;
10 begin
11 while true do begin
12 receive("Producerprocess", nextc);
13 (* consume item in nextc *)
14 end;
15 end;
16 parbegin
17 Consumerprocess: consumer;
18 Producerprocess: producer;
19 parend
20 end.
```

Comments

line 1 Here, nextp is the item the consumer produces, and nextc the item that the consumer consumes.

lines 2-8 This procedure simply generates items and sends them to the consumer process (named Consumerprocess). Suppose the capacity of the link is n items. If n items are waiting to be consumed, and the producer tries to send the n+1-st item, the producer will block (suspend) until the consumer has removed one item from the link (i.e., done a receive on the producer process). Note the name of the consumer process is given explicitly, so this is an example of “explicit naming” or “direct communication.” Also, since the send is blocking, it is an example of “synchronous communication.”

lines 9-15 This code simply receives items from the producer process (named Producerprocess) and consumes them. If when the receive statement is executed there are no items in the link, the consumer will block (suspend) until the producer has put an item from the link (i.e., done a send to the consumer process). Note the name of the producer process is given explicitly; again this is an example of “explicit naming” or “direct communication.” Also, since the receive is blocking, it is an example of “synchronous communication.”

lines 17-20 This starts two concurrent processes, the Consumerprocess and the Producerprocess.
send/receive Chart

Introduction
These charts summarize the actions of the send and receive primitives using both blocking and non-blocking mode and explicit and implicit naming.

Charts
This chart summarizes how naming and blocking affects the send primitive.

<table>
<thead>
<tr>
<th>send</th>
<th>blocking</th>
<th>non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit</td>
<td>send message to receiver; wait until message</td>
<td>send message to receiver</td>
</tr>
<tr>
<td>naming</td>
<td>accepted</td>
<td></td>
</tr>
<tr>
<td>implicit</td>
<td>broadcast message; wait until all processes</td>
<td>broadcast message</td>
</tr>
<tr>
<td>naming</td>
<td>accept message</td>
<td></td>
</tr>
</tbody>
</table>

This chart summarizes how naming and blocking affects the receive primitive.

<table>
<thead>
<tr>
<th>receive</th>
<th>blocking</th>
<th>non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit</td>
<td>wait for message from named sender</td>
<td>if there is a message from the named</td>
</tr>
<tr>
<td>naming</td>
<td></td>
<td>sender, get it; otherwise, proceed</td>
</tr>
<tr>
<td>implicit</td>
<td>wait for message from any sender</td>
<td>if there is a message from any</td>
</tr>
<tr>
<td>naming</td>
<td></td>
<td>sender, get it; otherwise, proceed</td>
</tr>
</tbody>
</table>
Monitors and Priority Waits

Introduction

This is an example of a monitor using priority waits. It implements a simple alarm clock; that is, a process calls `alarmclock.wakeme(n)`, and suspends for `n` seconds. Note that we are assuming the hardware invokes the procedure `tick` to update the clock every second.

Algorithm

```pascal
1  alarmclock: monitor;
2  var now: integer;
3      wakeup: condition;
4  procedure entry wakeme(n: integer);
5      begin
6          alarmsetting := now + n;
7          while now < alarmsetting do
8              wakeup.wait(alarmsetting);
9          wakeup.signal;
10     end;
11  procedure entry tick;
12     begin
13        now := now + 1;
14        wakeup.signal;
15     end.
```

Comments

- **lines 2-3** Here, `now` is the current time (in seconds) and is updated once a second by the procedure `tick`. When a process suspends, it will do a wait on the condition `wakeup`.
- **line 6** This computes the time at which the process is to be awakened.
- **lines 7-8** The process now checks that it is to be awakened later, and then suspends itself.
- **line 9** Once a process has been woken up, it `signals` the process that is to resume next. That process checks to see if it is time to wake up; if not, it suspends again (hence the `while` loop above, rather than an `if` statement). If it is to wake up, it `signals` the next process...
- **line 14** This is done once a second (hence the addition of 1 to `now`). The processes to be woken up are queued in order of remaining time to wait with the next one to wake up first. So, when `tick` signals, the next one to wake up determines if it is in fact time to wake up. If not, it suspends itself; if so, it proceeds.
signals some other process that that process may proceed (using the same priority as above). It sus-
pends on the semaphore xcond. When restarted, it indicates it is done with the x.wait by sub-
tracting 1 from xcondcount, and proceeds. Note that the down(xcond) will always suspend
the process since, unlike semaphores, if no process is suspended on x.wait, then x.signal is
ignored. So when this is executed, the value of the semaphore xcond is always 0.

lines 16-21  First, the process indicates it will be executing an x.signal by adding 1 to urgentcount. It
then checks if any process is waiting on condition variable x(xcondcount > 0), and if so signals
any such process (up(xcondsem)) before suspending itself (down(urgent)). When restarted,
the process indicates it is no longer suspended (by subtracting 1 from urgentcount).
Monitors and Semaphores

Introduction

This handout describes how to express monitors in terms of semaphores. If an operating system provided semaphores as primitives, this is what a compiler would produce when presented with a monitor.

Algorithm

1 var mutex, urgent, xcond: semaphore;
2 urgentcount, xcondcount: integer;

The body of each procedure in the monitor is set up like this:

3 down(xmutex);
4 (* procedure body*)
5 if urgentcount > 0 then
6 up(urgent)
7 else
8 up(mutex);

Each \texttt{x.wait} within the procedure is replaced by:

9 xcondcount := xcondcount + 1;
10 if urgentcount > 0 then
11 up(urgent)
12 else
13 up(mutex);
14 down(xcond);
15 xcondcount := xcondcount - 1;

Each \texttt{x.signal} within the procedure is replaced by:

16 urgentcount := urgentcount + 1;
17 if xcondcount > 0 then begin
18 up(xcondsem);
19 down(urgent);
20 end;
21 urgentcount := urgentcount - 1;

Comments

line 1 The semaphore \texttt{mutex} is initialized to 1 and ensures that only one process at a time is executing within the monitor. The semaphore \texttt{urgent} is used to enforce the requirement that processes that \texttt{signal} (and as a result are suspended) are to be restarted before any new process enters the monitor. The semaphore \texttt{xcond} will be used to block processes doing \texttt{waits} on the condition variable \texttt{x}. Note that if there is more than one such condition variable, a corresponding semaphore for each condition variable must be generated. Both \texttt{urgent} and \texttt{xcond} are initialized to 0.

line 2 The integer \texttt{urgentcount} indicates how many processes are suspended as a result of a \texttt{signal} operation (and are therefore waiting on the semaphore \texttt{urgent}); the counter \texttt{xcondcount} is associated with the condition variable \texttt{x}, and indicates how many processes are suspended on that condition (i.e., suspended on the semaphore \texttt{xcond}).

lines 3-8 Since only one process at a time may be in the monitor, the process entering the monitor procedure must wait until no other process is using it (\texttt{down(mutex)}). On exit, the process signals others that they may attempt entry, using the following order: if any other process has issued a signal and been suspended (i.e., \texttt{urgentcount} = 0), the exiting process indicates that one of those is to be continued (\texttt{up(urgent)}). Otherwise, one of the processes trying to enter the monitor may do so (\texttt{up(mutex)}).

lines 9-15 First, the process indicates it will be executing an \texttt{x.wait} by adding 1 to \texttt{xcondcount}. It then
more readers are reading, it indicates a writer may go ahead by signalling on the condition variable
oktowrite.

lines 19-21 In this routine, the writer first sees if any readers or writers are accessing the file; if so, it waits until
they are done. Then it indicates that it is writing to the file by setting the boolean writing to
true.

lines 26-31 Here, the writer first announces it is done by setting writing to false. Since readers have pri-
ority, it then checks to see if any readers are waiting; if so, it signals all of them (as many readers
can access the file simultaneously). If not, it signals any writers waiting.

line 34 This initializes the variables.
First Readers Writers Problem

Introduction
This algorithm uses a monitor to solve the first readers-writers problem.

Algorithm

```pascal
1 readerwriter: monitor
2 var  readcount: integer;
3      writing: boolean;
4      oktoread, oktowrite: condition;
5 procedure entry beginread;
6 begin
7    readcount := readcount + 1;
8    if writing then
9       oktoread.wait;
10 end;
11 procedure entry endread;
12 begin
13    readcount := readcount - 1;
14    if readcount = 0 then
15       oktowrite.signal;
16    end;
17 procedure entry beginwrite;
18 begin
19    if readcount > 0 or writing then
20       oktowrite.wait;
21       writing := true;
22    end;
23 procedure entry endwrite;
24 var  i: integer;
25 begin
26    writing := false;
27    if readcount > 0 then
28       for i := 1 to readcount
29          oktoread.signal;
30 else
31       oktowrite.signal;
32 end;
33 begin
34    readcount := 0; writing := false;
35 end.
```

Comments

lines 1-4 Here, readcount contains the number of processes reading the file, and writing is true when a writer is writing to the file. oktoread and oktowrite correspond to the logical conditions of being able to access the file for reading and writing, respectively.

lines 7-9 In this routine, the reader announces that it is ready to read (by adding 1 to readcount). If a writer is accessing the file, it blocks on the condition variable oktoread; when done, the writer will signal on that condition variable, and the reader can proceed.

lines 13-15 In this routine, the reader announces that it is done (by subtracting 1 from readcount). If no
them. If there are no blocked consumers, this is effectively a no-op.

line 14 As with the previous procedure, this is called from outside the monitor by `buffer.extract(...)`.

lines 16-17 This code checks to see if there is any unconsumed item in the buffer. If not, the process blocks on the condition `notempty`; when some other process does deposit an element in the buffer, then there will be something for the consumer to extract and that producer process will signal on the condition `notempty`, allowing the blocked one to proceed. Note that while blocked on this condition, other processes may access procedures within the monitor.

lines 18-20 This code actually extracts the item from the buffer. Note that the monitor guarantees mutual exclusion.

line 21 Just as a consumer will block on an empty buffer, a producer will block on a full one. This indicates to any such producer process that the buffer is no longer full, and unblocks exactly one of them. If there are no blocked producers, this is effectively a no-op.

lines 23-25 This is the initialization part.
Producer Consumer Problem

Introduction
This algorithm uses a monitor to solve the producer/consumer (or bounded-buffer) problem.

Algorithm

```plaintext
buffer: monitor;
var slots: array [0..n-1] of item;
    count, in, out: integer;
    notempty, notfull: condition;
procedure entry deposit(data: item);
begin
    if count = n then
        notfull.wait;
    slots[in] := data;
    in := in + 1 mod n;
    count := count + 1;
    notempty.signal;
end;
procedure entry extract(var data: item);
begin
    if count = 0 then
        notempty.wait;
    data := slots[out];
    out := out + 1 mod n;
    count := count - 1;
    notfull.signal;
end;
begin
    count := 0; in := 0; out := 0;
end.
```

Comments

lines 2-4 Here, `slots` is the actual buffer, `count` the number of items in the buffer, and `in` and `out` the indices of the next element of `slots` where a deposit is to be made or from which an extraction is to be made. There are two conditions we care about: if the buffer is not full (represented by the condition variable `notfull`), and if the buffer is not empty (represented by the condition variable `notempty`).

line 5 The keyword `entry` means that this procedure may be called from outside the monitor. It is called by placing the name of the monitor first, then a period, then the function name; so, `buffer.deposit(...)`. 

lines 7-8 This code checks to see if there is room in the buffer for a new item. If not, the process blocks on the condition `notfull`; when some other process does extract an element from the buffer, then there will be room and that process will signal on the condition `notfull`, allowing the blocked one to proceed. Note that while blocked on this condition, other processes may access procedures within the monitor.

lines 9-11 This code actually deposits the item into the buffer. Note that the monitor guarantees mutual exclusion.

line 12 Just as a producer will block on a full buffer, a consumer will block on an empty one. This indicates to any such consumer process that the buffer is no longer empty, and unblocks exactly one of
readcount is incremented by 1; if this is the only reader that will access the file, it waits until any writers have finished (down(wrt)). It then indicates other processes may access readcount (down(mutex)) and proceeds to read from the file.

lines 16-20 Now the reader is done reading the file (for now.) It must update the value of readcount to indicate this, so it waits until no-one else is accessing that variable (down(mutex)) and then decrements readcount. If no other readers are waiting to read (readcount = 0), it signals that any reader or writer who wishes to access the file may do so (up(wrt)). Finally, it indicates it is done with readcount (up(mutex)).

line 24 Since the file is not accessed here, we don't need to put semaphores around this part.

lines 25-26 The writer process waits (down(wrt)) until no other process is accessing the file; it then proceeds to write to the file.

line 27 When the writer is done writing to the file, it signals that anyone who wishes to access the file may do so (up(wrt)).
First Readers Writers Problem

Introduction

This algorithm uses semaphores to solve the first readers-writers problem.

Algorithm

1 var wrt, mutex: semaphore;
2 readcount: integer;
3 begin
4   readcount := 0;
5   wrt := 1;
6   mutex := 1;
7 parbegin
8   repeat (* reader process *)
9     (* do something *)
10    down(mutex);
11    readcount := readcount + 1;
12    if readcount = 1 then
13       down(wrt);
14       up(mutex);
15     (* read the file *)
16    down(mutex);
17    readcount := readcount - 1;
18    if readcount = 0 then
19       up(wrt);
20       up(mutex);
21     (* do something else *)
22   until false;
23 repeat (* writer process *)
24     (* do something *)
25    down(wrt);
26     (* write to the file *)
27    up(wrt);
28     (* do something else *)
29   until false;
30 parend;
31 end.

Comments

lines 1-2 Here, readcount contains the number of processes reading the file, and mutex is a semaphore used to provide mutual exclusion when readcount is incremented or decremented. The semaphore wrt is common to both readers and writers and ensures that when one writer is accessing the file, no other readers or writers may do so.

lines 4-6 This just initializes all the semaphores. It is the only time anything other than a down or an up operation may be done to them. As no readers are yet reading the file, readcount is initialized to 0.

line 9 Since the file is not accessed here, we don't need to put semaphores around this part.

lines 10-15 Since the value of the shared variable readcount is going to be changed, the process must wait until no-one else is accessing it (down(mutex)). Since this process will read from the file,
Extracting an item from the buffer, however, does require that the consumer process obtain exclusive access to the buffer. First, the consumer checks that there is a slot in the buffer with an item deposited and, if not, waits until there is \( \textit{down(full)} \). When there is, it waits until it can obtain exclusive access to the buffer \( \textit{down(mutex)} \). Once both these conditions are met, it can safely extract the item.

As the consumer is done with the buffer, it signals that any other process needing to access the buffer may do so \( \textit{up(mutex)} \). It then indicates it has extracted another item into the buffer \( \textit{up(empty)} \).

Since the buffer is not accessed while the item is consumed, we don't need to put semaphores around this part.
Producer/Consumer Problem

Introduction

This algorithm uses semaphores to solve the producer/consumer (or bounded buffer) problem.

Algorithm

1 var
2   buffer: array [0..n-1] of item;
3   full, empty, mutex: semaphore;
4   nextp, nextc: item;
5 begin
6   full := 0;
7   empty := n;
8   mutex := 1;
9   parbegin
10   repeat  (* producer process *)
11     (* produce an item in nextp *)
12     down(empty);
13     (* deposit nextp in buffer *)
14     down(mutex);
15     up(mutex);
16     up(full);
17     until false;
18   repeat  (* consumer process *)
19     (* extract an item in nextc *)
20     down(full);
21     down(mutex);
22     up(mutex);
23     up(empty);
24     (* consume the item in nextc *)
25     until false;
26   parend;
27 end.

Comments

lines 1-3 Here, buffer is the shared buffer, and contains n spaces; full is a semaphore the value of which is the number of filled slots in the buffer, empty is a semaphore the value of which is the number of empty slots in the buffer, and mutex is a semaphore used to enforce mutual exclusion (so only one process can access the buffer at a time). nextp and nextc are the items produced by the producer and consumed by the consumer, respectively.

line 5-7 This just initializes all the semaphores. It is the only time anything other than a down or an up operation may be done to them.

line 10 Since the buffer is not accessed while the item is produced, we don't need to put semaphores around this part.

lines 11-13 Depositing an item into the buffer, however, does require that the producer process obtain exclusive access to the buffer. First, the producer checks that there is an empty slot in the buffer for the new item and, if not, waits until there is (down(empty)). When there is, it waits until it can obtain exclusive access to the buffer (down(mutex)). Once both these conditions are met, it can safely deposit the item.

lines 14-15 As the producer is done with the buffer, it signals that any other process needing to access the buffer may do so (up(mutex)). It then indicates it has put another item into the buffer (up(full)).
A solution must prevent both.

Figure. The Dining Philosopher's Table
Classical Synchronization Problems

Introduction

This handout states three classical synchronization problems that are often used to compare language constructs that implement synchronization mechanisms and critical sections.

The Producer-Consumer Problem

In this problem, two processes, one called the producer and the other called the consumer, run concurrently and share a common buffer. The producer generates items that it must pass to the consumer, who is to consume them. The producer passes items to the consumer through the buffer. However, the producer must be certain that it does not deposit an item into the buffer when the buffer is full, and the consumer must not extract an item from an empty buffer. The two processes also must not access the buffer at the same time, for if the consumer tries to extract an item from the slot into which the producer is depositing an item, the consumer might get only part of the item. Any solution to this problem must ensure none of the above three events occur.

A practical example of this problem is electronic mail. The process you use to send the mail must not insert the letter into a full mailbox (otherwise the recipient will never see it); similarly, the recipient must not read a letter from an empty mailbox (or he might obtain something meaningless but that looks like a letter). Also, the sender (producer) must not deposit a letter in the mailbox at the same time the recipient extracts a letter from the mailbox; otherwise, the state of the mailbox will be uncertain.

Because the buffer has a maximum size, this problem is often called the bounded buffer problem. A (less common) variant of it is the unbounded buffer problem, which assumes the buffer is infinite. This eliminates the problem of the producer having to worry about the buffer filling up, but the other two concerns must be dealt with.

The Readers-Writers Problem

In this problem, a number of concurrent processes require access to some object (such as a file.) Some processes extract information from the object and are called readers; others change or insert information in the object and are called writers. The Bernstein conditions state that many readers may access the object concurrently, but if a writer is accessing the object, no other processes (readers or writers) may access the object. There are two possible policies for doing this:

First Readers-Writers Problem. Readers have priority over writers; that is, unless a writer has permission to access the object, any reader requesting access to the object will get it. Note this may result in a writer waiting indefinitely to access the object.

Second Readers-Writers Problem. Writers have priority over readers; that is, when a writer wishes to access the object, only readers which have already obtained permission to access the object are allowed to complete their access; any readers that request access after the writer has done so must wait until the writer is done. Note this may result in readers waiting indefinitely to access the object.

So there are two concerns: first, enforce the Bernstein conditions among the processes, and secondly, enforcing the appropriate policy of whether the readers or the writers have priority.

A typical example of this occurs with databases, when several processes are accessing data; some will want only to read the data, others to change it. The database must implement some mechanism that solves the readers-writers problem.

The Dining Philosophers Problem

In this problem, five philosophers sit around a circular table eating spaghetti and thinking. In front of each philosopher is a plate and to the left of each plate is a fork (so there are five forks, one to the right and one to the left of each philosopher's plate; see the figure). When a philosopher wishes to eat, he picks up the forks to the right and to the left of his plate. When done, he puts both forks back on the table. The problem is to ensure that no philosopher will be allowed to starve because he cannot ever pick up both forks.

There are two issues here: first, deadlock (where each philosopher picks up one fork so none can get the second) must never occur; and second, no set of philosophers should be able to act to prevent another philosopher from ever eating.
Test and Set Solution

Introduction

This algorithm solves the critical section problem for $n$ processes using a Test and Set instruction (called TaS here). This instruction does the following function atomically:

```plaintext
function TaS(var Lock: boolean): boolean;
begin
  TaS := Lock;
  Lock := true;
end;
```

Algorithm

```plaintext
1  var waiting: shared array [0..n-1] of boolean;
2    Lock: shared boolean;
3    j: 0..n-1;
4    key: boolean;
...
5  repeat (* process $P_i$ *)
6      waiting[i] := true;
7      key := true;
8      while waiting[i] and key do
9          key := TaS(Lock);
10         waiting[i] := false;
11         (* critical section goes here *)
12        j := i + 1 mod n;
13        while (j <> i) and not waiting[j] do
14            j := j + 1 mod n;
15        if j = i then
16            Lock := false
17        else
18            waiting[j] := false;
19  until false;
```

Comments

lines 1-2: These are global to all processes, and are all initialized to false.
lines 3-4: These are local to each process $P_i$ and are uninitialized.
lines 5-10: This is the entry section. Basically, $waiting[i]$ is true as long as $P_i$ is trying to get into its critical section; if any other process is in that section, then Lock will also be true, and $P_i$ will loop in lines 8-9. Once $P_i$ can go on, it is no longer waiting for permission to enter, and sets $waiting[i]$ to false (line 10); it then proceeds into the critical section. Note that Lock is set to true by the TaS instruction in line 9 that returns false.
lines 12-18: This is the exit section. When $P_j$ leaves the critical section, it must choose which other waiting process may enter next. It starts with the process with the next higher index (line 12). It checks each process to see if that process is waiting for access (lines 13-14); if no-one is, it simply releases the lock (by setting Lock to false; lines 15-16). However, if some other process $P_j$ is waiting for entry, $P_j$ simply changes $waiting[j]$ to false to allow $P_j$ to enter the critical section (lines 17-18).
Bogus Bakery Algorithm

Introduction

Why does Lamport's Bakery algorithm used a variable called `choosing` (see the algorithm, lines 1, 4, 6, and 8)? It is very instructive to see what happens if you leave it out. This gives an example of mutual exclusion being violated if you don't put `choosing` in. But first, the algorithm (with the lines involving `choosing` commented out) so you can see what the modification was:

Algorithm

```
1 var (*choosing: shared array [0..n-1] of boolean; *)
2 number: shared array [0..n-1] of integer;
...
3 repeat
4 (* choosing[i] := true; *)
5 number[i] := max(number[0],number[1],...,number[n-1]) + 1;
6 (* choosing[i] := false; *)
7 for j := 0 to n-1 do begin
8 (* while choosing[j] do; *)
9 while number[j] <> 0 and
10 (number[j], j) < (number[i],i) do
11 (* nothing *);
12 end;
13 (* critical section *)
14 number[i] := 0;
15 (* remainder section *)
16 until false;
```

Proof of Violation of Mutual Exclusion

Suppose we have two processes just beginning; call them p_0 and p_1. Both reach line 5 at the same time. Now, we'll assume both read `number[0]` and `number[1]` before either addition takes place. Let p_1 complete the line, assigning 1 to `number[1]`, but p_0 block before the assignment. Then p_1 gets through the `while` loop at lines 9-11 and enters the critical section. While in the critical section, it blocks; p_0 unblocks, and assigns 1 to `number[0]` at line 5. It proceeds to the while loop at lines 9-11. When it goes through that loop for j = 1, the condition on line 9 is true. Further, the condition on line 10 is false, so p_0 enters the critical section. Now p_0 and p_1 are both in the critical section, violating mutual exclusion.

The reason for `choosing` is to prevent the `while` loop in lines 9-11 from being entered when process j is setting its `number[j]`. Note that if the loop is entered and then process j reaches line 5, one of two situations arises. Either `number[j]` has the value 0 when the first test is executed, in which case process i moves on to the next process, or `number[j]` has a non-zero value, in which case at some point `number[j]` will be greater than `number[i]` (since process i finished executing statement 5 before process j began). Either way, process i will enter the critical section before process j, and when process j reaches the `while` loop, it will loop at least until process i leaves the critical section.
Bakery Algorithm

Introduction

This algorithm solves the critical section problem for \( n \) processes in software. The basic idea is that of a bakery; customers take numbers, and whoever has the lowest number gets service next. Here, of course, “service” means entry to the critical section.

Algorithm

1 var choosing: shared array [0..n-1] of boolean;
2 number: shared array [0..n-1] of integer;

repeat
    choosing[i] := true;
    number[i] := max(number[0],number[1],...,number[n-1]) + 1;
    choosing[i] := false;
    for j := 0 to n-1 do begin
        while choosing[j] do (* nothing *);
        while number[j] <> 0 and (number[j], j) < (number[i],i) do (* nothing *);
    end;
(* critical section *)
number[i] := 0;
(* remainder section *)
until false;

Comments

lines 1-2: Here, \( choosing[i] \) is true if \( P_i \) is choosing a number. The number that \( P_i \) will use to enter the critical section is in \( number[i] \); it is 0 if \( P_i \) is not trying to enter its critical section.

lines 4-6: These three lines first indicate that the process is choosing a number (line 4), then try to assign a unique number to the process \( P_i \) (line 5); however, that does not always happen. Afterwards, \( P_i \) indicates it is done (line 6).

lines 7-12: Now we select which process goes into the critical section. \( P_i \) waits until it has the lowest number of all the processes waiting to enter the critical section. If two processes have the same number, the one with the smaller name – the value of the subscript – goes in; the notation “\( (a,b) < (c,d) \)” means true if \( a < c \) or if both \( a = c \) and \( b < d \) (lines 9-10). Note that if a process is not trying to enter the critical section, its number is 0. Also, if a process is choosing a number when \( P_i \) tries to look at it, \( P_i \) waits until it has done so before looking (line 8).

line 14: Now \( P_i \) is no longer interested in entering its critical section, so it sets \( number[i] \) to 0.
Interprocess Synchronization and Communication

1. Goal: To understand how these are implemented we need to look at what they are and why they are necessary.

2. Bernstein conditions
   a. can’t read, write or write, write from/to same location concurrently

3. Parallel Programming Constructs
   a. *fork* splits a process, *join* resynchronizes them, *quit* stops one
   b. *parbegin, parend* bracket concurrent statements

4. Critical Section Problem
   a. mutual exclusion
   b. progress
   c. bounded wait

5. Software Solutions
   a. Peterson’s solution
   b. Lamport’s bakery algorithm

6. Hardware Solutions
   a. Test and Set

7. Semaphores
   a. non-negative integer variables that can be accessed only using the *wait* and *signal* operations

8. Testing synchronization primitives
   a. bounded buffer problem
   b. readers-writers problem
   c. dining philosophers’ problem

9. Abstract datatypes
   a. classes
   b. instances

10. Monitors
    a. mutual exclusion and coordination through *wait, signal* operations
    b. condition variables
    c. priority waiting

11. Event counters and sequencers
    a. allow synchronization without mutual exclusion
    b. sequencers vs. eventcounters
    c. *read, advance, await, ticket*

12. Problems
    a. lack of hardware support
    b. shared memory vs. message-based synchronization schemes

13. Interprocess communication (IPC)
    a. *send, receive*
    b. blocking vs. non-blocking
    c. capacity
    d. explicit vs. implicit naming
    e. acknowledgements
    f. process terminations
    g. lost or corrupted messages

14. Remote Procedure Calls (RPC)
    a. like a regular subroutine to programmer, but the IPC details are all within the subroutine

15. ADA™
    a. *accept, select* mechanisms
    b. fair internal policy