Outline for January 9, 2001

1. Greetings and felicitations!
   a. First part of project due Friday
   b. Web page up and running!
2. What is concurrency?
   a. Concurrent (parallel) vs. sequential (serial)
   b. Logical vs.actual concurrency
   c. Process creation: statically declare all subprocesses (created at execution) or dynamically spawn them
   d. Can view OS as a collection of concurrent processes
3. Simple parallel constructs
   a. fork, join, quit
   b. cobegin/coend
4. Process models
   a. P(p1, p2); S(p1,p2)
   b. Proper and improper nesting
   c. $\rightarrow$ (precedence) relation: $p_i \rightarrow p_j$ means p_i must complete before p_j starts
   d. Domain, range of processes
   e. Equivalence of systems of processes
   f. Determinate system of processes
   g. Mutually noninterfering system of processes
   h. Theorem: If a system is mutually noninterfering, it is determinate.
   i. Theorem: Let f_p be an interpretation of process p. Let $\prod$ be a system of processes, with $p \in \prod$. If for all such p, $domain(p) \neq \emptyset$ and $range(p) \neq \emptyset$, but f_p unspecified, is determinate for all f_p, then all processes in $\prod$ are mutually noninterfering
   j. Maximally parallel system: determinate system for which the removal of any pair from the relation $\rightarrow$ makes the two processes in the pair interfering processes.
5. Critical section problem
   a. Mutual exclusion
   b. Progress
   c. Bounded wait
6. Classical problems
   a. Producer/consumer
   b. Readers/writers (first: readers priority; second: writers priority)
   c. Dining philosophers
7. Basic language constructs
   a. Semaphores
   b. Send/receive
8. Evaluating higher-level language constructs
   a. Modularity
   b. Constraints
   c. Expressive power
   d. Ease of use
   e. Portability
   f. Relationship with proram structure
   g. Process failures, unanticipated faults (exception handling)
   h. Real-time systems
9. Higher-level language constructs
   a. Monitors
   b. Crowd monitors

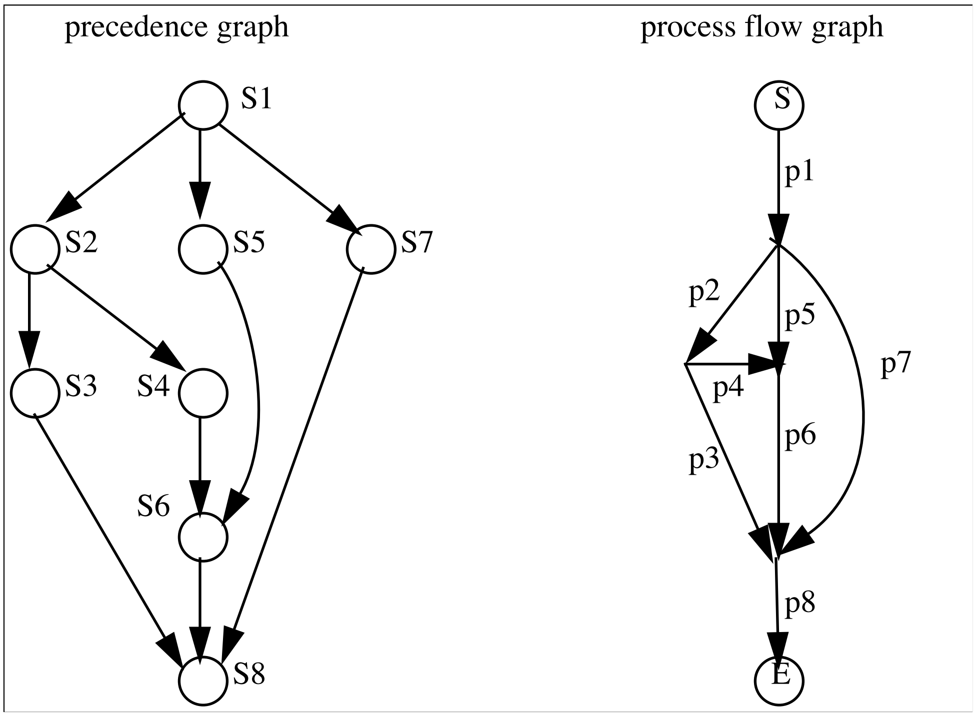
c. Invariant expressions
d. CSP
e. RPC
f. ADA™

Improper Nesting Example

Introduction

One of the limits on the use of parbegin/parend, and any related constructs, is that the program involved must be properly nested. Not all programs are. For example, consider the program represented by the following graphs.

The Program as Graphs



Using *fork/join* Primitives

The program equivalent to these precedence and process flow graphs is:

```
        t6 := 2;
        t8 := 3;
        S1; fork p2; fork p5; fork p7; quit;
p2:     S2; fork p3: fork p4; quit;
p5:     S5; join t6, p6; quit;
p7:     S7; join t8, p8; quit;
p3:     S3; join t8, p8; quit;
p4:     S4; join t6, p6; quit;
p6:     S6; join t8, p8; quit;
p8:     S8; quit
```

where S*i* is the program for p*i*.

Using *parbegin/parend* Primitives

To see if this is possible, we must determine if the above program is properly nested. If not, we clearly cannot represent it using *parbegin* and *parend*, which require a block structure, and hence proper nesting.
Let $S(a,b)$ represent the serial execution of processes a and b, and $P(a,b)$ the parallel execution of processes a and b. Then a process flow graph is properly nested if it can be described by P, S, and functional composition. For example, the program

```
parbegin
      p1:    a := b + 1;
      p2:    c := d + 1;
parend
p3:          e := a + c;
```

would be written as

$$S(P(\text{p1},\text{p2}),\text{p3})$$

We now prove:

Claim. The example is not properly nested.

Proof: For something to be properly nested, it must be of the form *S*(p*i*,p*j*) or *P*(p*i*,p*j*) at the most interior level. Clearly the example's most interior level is not *P*(p*i*,p*j*) as there are no constructs of that form in the graph.

In the graph, all serially connected processes pi and pj have at least 1 more process p*k* starting or finishing at the node between p*i* and p*j*; but if *S*(p*i*,p*j*) is in the innermost level, there can be no such p*k* (else a more interior *P* or *S* is needed, contradiction). Hence, it's not *S*(p*i*,p*j*) either.

Maximally Parallel Systems

Introduction

A *maximally parallel system* is a determinate system for which the removal of any pair from the precedence relation $\rightarrow$ makes the two processes in the pair interfering processes.

Example

The system $S = (\prod, \rightarrow)$ is composed of the set of processes $\prod = \{ p_1, .., p_9 \}$ and the precedence relation

$$\rightarrow = \{ (p_1,p_2), (p_1,p_3), (p_1,p_4), (p_2,p_5), (p_3,p_5), (p_4,p_6), (p_4,p_7), (p_4,p_8), (p_5,p_8), (p_6,p_8), (p_7,p_9), (p_8,p_9) \}.$$

The processes have the following domains and ranges:

process	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
domain	1	4	3	1	3	6	5	1,3	1,4,6
range	2,3	4	2,3	1	3	6	5	4	2,3

Transitive closure of $\rightarrow$

In the following, a bullet is placed whenever the process in the row precedes the process in the column under $\rightarrow$.

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
p_1		•	•	•					
p_2					•				
p_3					•				
p_4						•	•	•	
p_5								•	
p_6								•	
p_7									•
p_8									•

For p_1, we have $p_1 \rightarrow p_2$ and $p_2 \rightarrow p_5$, so $p_1 \rightarrow p_5$. As $p_5 \rightarrow p_8$, $p_1 \rightarrow p_8$. As $p_8 \rightarrow p_9$, $p_1 \rightarrow p_9$. The table becomes:

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
p_1		•	•	•	•	•	•	•	•
p_2					•				
p_3					•				
p_4						•	•	•	
p_5								•	
p_6								•	
p_7									•
p_8									•

Continuing on in this fashion, the table finally becomes:

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
p_1		•	•	•	•	•	•	•	•
p_2					•			•	•
p_3					•			•	•
p_4						•	•	•	•
p_5								•	•
p_6								•	•
p_7									•
p_8									•

giving the transitive closure of $\rightarrow$ to be:

$$\rightarrow^* = \{ (p_1,p_2), (p_1,p_3), (p_1,p_4), (p_1,p_5), (p_1,p_6), (p_1,p_7), (p_1,p_8), (p_1,p_9), (p_2,p_5), (p_2,p_8), (p_2,p_9), (p_3,p_5), (p_3,p_8), (p_3,p_9), (p_4,p_6), (p_4,p_7), (p_4,p_8), (p_4,p_9), (p_5,p_8), (p_5,p_9), (p_6,p_8), (p_6,p_9), (p_7,p_9), (p_8,p_9) \}$$

Bernstein Conditions

For the system to be determinate, the Bernstein conditions must hold. This means that two processes which write into the same memory location cannot be executed concurrently. Also, if a process reads from a location that another process writes to, those two processes cannot be concurrent. So we first list those processes which cannot be concurrent by computing the elements of the three sets listed below. (We use $\rightarrow^*$ for this, because the original precedence relation may omit pairs that follow from the transitivity of $\rightarrow$.) Note that the range of p_i is the set of memory locations that p_i writes to, and the domain of p_i is the set of memory locations that p_i reads from.

$$range(p_i) \cap range(p_j) = \{ (p_1,p_3), (p_1,p_5), (p_1,p_9), (p_2,p_8), (p_3,p_5), (p_3,p_9), (p_5,p_9) \}$$
$$domain(p_i) \cap range(p_j) = \{ (p_1,p_4), (p_2,p_8), (p_3,p_5), (p_3,p_9), (p_5,p_9), (p_8,p_9) \}$$
$$range(p_i) \cap domain(p_j) = \{ (p_1,p_3), (p_1,p_5), (p_1,p_8), (p_2,p_9), (p_3,p_5), (p_3,p_8), (p_4,p_8), (p_4,p_9), (p_5,p_8), (p_6,p_9) \}$$

The Equivalent Maximally Parallel System

The only precedences that are actually *required* by the system are those that enforce the Bernstein conditions. The complete set of precedences that exist in the system is given by the set $\rightarrow^*$, so taking those elements of $\rightarrow^*$ in the three sets above gives us the precedence relation $\rightarrow^+$ for the maximally parallel system equivalent to the original system:

$$\rightarrow^+ = \{ (p_1,p_3), (p_1,p_4), (p_1,p_5), (p_1,p_8), (p_1,p_9), (p_2,p_8), (p_2,p_9), (p_3,p_5), (p_3,p_8), (p_3,p_9), (p_4,p_8), (p_4,p_9), (p_5,p_8), (p_5,p_9), (p_6,p_9), (p_8,p_9) \}$$

Now, note that several of these elements are implied by others, since precedence is transitive; for example, (p_1,p_4) and (p_4,p_8) means (p_1,p_8) holds. Eliminating these redundent precedences, this set becomes:

$$\{ (p_1,p_3), (p_1,p_4), (p_2,p_8), (p_3,p_5), (p_4,p_8), (p_5,p_8), (p_6,p_9), (p_8,p_9) \}$$

Producer/Consumer Problem

Introduction

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

Algorithm

```
 1   var         buffer: array [0..n-1] of item;
 2               full, empty, mutex: semaphore;
 3               nextp, nextc: item;
 4   begin
 5         full := 0;
 6         empty := n;
 7         mutex := 1;
 8         parbegin
 9               repeat                  (* producer process *)
10                     (* produce an item in nextp *)
11                     P(empty);
12                     P(mutex);
13                     (* deposit nextp in buffer *)
14                     V(mutex);
15                     V(full);
16               until false;
17               repeat                  (* consumer process *)
18                     P(full);
19                     P(mutex);
20                     (* extract an item in nextc *)
21                     V(mutex);
22                     V(empty);
23                     (* consume the item in nextc *)
24               until false;
25         parend;
26   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 emoty 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 (`P(empty)`). When there is, it waits until it can obtain exclusive access to the buffer (`P(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 (`V(mutex)`). It then indicates it has put another item into the buffer (`V(full)`).

lines 18-20 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 (`P(full)`). When there is, it waits until it can obtain exclusive access to the buffer (`P(mutex)`). Once both these conditions are met, it can safely extract the item.

lines 21-22 As the consumer is done with the buffer, it signals that any other process needing to access the buffer may do so (`V(mutex)`). It then indicates it has extracted another item into the buffer (`V(empty)`).

line 23 Since the buffer is not accessed while the item is consumed, we don't need to put semaphores around this part.

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              P(mutex);
11              readcount := readcount + 1;
12              if readcount = 1 then
13                    P(wrt);
14              V(mutex);
15              (* read the file *)
16              P(mutex);
17              readcount := readcount - 1;
18              if readcount = 0 then
19                    V(wrt);
20              V(mutex);
21       (* do something else *)
22        until false;
23        repeat            (* writer process *)
24              (* do something *)
25              P(wrt);
26              (* write to the file *)
27              V(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 (`P(mutex)`). Since this process will read from the file, `readcount` is incremented by 1; if this is the only reader that will access the file, it waits until any writers have finished (`P(wrt)`). It then indicates other processes may access `readcount` (`P(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 (`P(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 (`V(wrt)`). Finally, it indicates it is done with `readcount` (`V(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 (`P(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 (`V(wrt)`).

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.

send	*blocking*	*non-blocking*
explicit naming	send message to receiver; wait until message accepted	send message to receiver
implicit naming	broadcast message; wait until all processes accept message	broadcast message

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

receive	*blocking*	*non-blocking*
explicit naming	wait for message from named sender	if there is a message from the named sender, get it; otherwise, proceed
implicit naming	wait for message from any sender	if there is a message from any sender, get it; otherwise, proceed

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  begin
17        parbegin
18              Consumerprocess: consumer;
19              Producerprocess: producer;
20        parend
21  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 ias 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`.

Producer Consumer Problem

Introduction

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

Algorithm

```
 1  buffer: monitor;
 2  var   slots: array [0..n-1] of item;
 3              count, in, out: integer;
 4              notempty, notfull: condition;
 5  procedure entry deposit(data: item);
 6  begin
 7        if count = n then
 8              notfull.wait;
 9        slots[in] := data;
10        in := in + 1 mod n;
11        count := count + 1;
12        notempty.signal;
13  end;
14  procedure entry extract(var data: item);
15  begin
16        if count = 0 then
17              notempty.wait;
18        data := slots[out];
19        out := out + 1 mod n;
20        count := count – 1;
21        notfull.signal;
22  end;
23  begin
24        count := 0; in := 0; out := 0;
25  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

	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.

First Readers Writers Problem

Introduction

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

Algorithm

```
 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

	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 priority, 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.

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  P(xmutex);
 4  (* procedure body*)
 5  if urgentcount > 0 then
 6        V(urgent)
 7  else
 8        V(mutex);
```

Each **`x.wait`** within the procedure is replaced by:

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

Each **`x.signal`** within the procedure is replaced by:

```
16  urgentcount := urgentcount + 1;
17  if xcondcount > 0 then begin
18        V(xcond);
19        P(urgent);
20  end;
21  urgentcount := urgentcount - 1;
```

Comments

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

line 2: The integer `urgentcount` indicates how many processes are suspended as a result of a **`signal`** operation (and are therefore waiting on the semaphore `urgent`); the counter `xcondcount` is associated with the condition variable `x`, and indicates how many processes are suspended on that condition (*i.e.*, suspended on the semaphore `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 (`P(mutex)`). On exit, the process signals others that they may attempt entry, using the following order: if any other process has issues a signal and been suspended (*i.e.*, `urgentcount` ≠ `0`), the exiting process indicates that one of those is to be continued (`V(urgent)`). Otherwise, one of the processes trying to enter the monitor may do so (`V(mutex)`).

lines 9-15: First, the process indicates it will be executing an **`x.wait`** by adding `1` to `xcondcount`. It then signals some other process that that process may proceed (using the same priority as above). It suspends on the semaphore `xcond`. When restarted, it indicates it is done with the **`x.wait`** by subtracting `1` from `xcondcount`, and proceeds. Note that the `P(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 (`V(xcondsem)`) before suspending itself (`P(urgent)`). When restarted, the process indicates it is no longer suspended (by subtracting `1` from `urgentcount`).

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

```
 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 **`signal`**s 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 **`signal`**s 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.

First Readers Writers Problem

Introduction

This uses crowd monitors to solve the first readers/writers problem.

Algorithm

```
readerwriter: crowd monitor
var   Readers: crowd read;
      Writers: crowd read, write;
      readcount: integer;
      writing: boolean;
      oktoread, oktowrite: condition;
guard procedure entry beginread;
begin
      readcount := readcount + 1;
      if writing then
            oktoread.wait;
      enter Readers;
end;
guard procedure entry endread;
begin
      leave Readers;
      readcount := readcount - 1;
      if readcount = 0 then
            oktowrite.signal;
end;
guard procedure entry beginwrite;
begin
      if readcount > 0 or writing then
            oktowrite.wait;
      writing := true;
      enter Writers;
end;
guard procedure entry endwrite;
var   i: integer;
begin
      leave Writers;
      writing := false;
      if readcount > 0 then
            for i := 1 to readcount
                  oktoread.signal;
      else
            oktowrite.signal;
end;
procedure entry read;
            … read from shared data …
end;
procedure entry write;
            … write to shared data …
end;
begin
      readcount := 0; writing := false;
```

```
47          end.
```

Comments

lines 2-3 These lines define which procedures can be called by members of the crowd; here, members of the *Readers* crowd can call *read*, and members of the *Writers* crowd can call either *read* or *write*. Only processes in those crowds can call *read* or *write*; should any other process do so, it will cause a run-time error.

line 7 The keyword **guard** means this procedure is mutually exclusive (so only one process at a time may be in the guarded procedures). Note this relaxes the definition of Hoare's monitor, in that multiple proceses may now access the monitor simultaneously.

line 12 This puts the calling process into the *Readers* crowd; it may now call the procedure *read*.

line 16 This removes the calling process from the *Readers* crowd, so it may not call *read* until after it calls *beginread* and executes line 12 again.

line 26 This puts the calling process into the *Writers* crowd; it may now call the procedures *read* and *write*.

line 31 This removes the calling process from the *Readers* crowd, so it may not call *read* or *write* until after it calls *beginread* or *beginwrite* and executes lines 12 or 26 again.

line 39 Now any number of processes may access the *read* procedure simultaneously.

line 42 Although it may appear that any number of processes may access the *write* procedure simultaneously, note that all callers must first have invoked *beginwrite* — and only one such process will be active at a time. So at most one process will call *write*.

Producer Consumer Problem

Introduction

This uses invariant expressions to solve the producer consumer problem.

Algorithm

```
 1  buffer: invariant module;
 2  const n = 1024;
 3  var   slots: array [0..n-1] of item;
 4        in, out: 0..n-1;
 5  invariant deposit
 6        StartCount(deposit) - FinishCount(extract) < n;
 7        CurrentCount(deposit) = 0;
 8  invariant extract
 9        StartCount(extract) - FinishCount(deposit) < 0
10        CurrentCount(extract) = 0;
11  procedure entry deposit(data: item);
12  begin
13        slots[in] := data;
14        in := in + 1 mod n;
15  end;
16  procedure entry extract(var data: item);
17  begin
18        data := slots[out];
19        out := out + 1 mod n;
20  end;
21  begin
22        in := 0; out := 0;
23  end.
```

Comments

lines 3-4	Here, *slots* is the actual 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.
line 5	The next constraints apply to the procedure *deposit*.
line 6	This invariant checks that there is at least one slot in the buffer that is empty. If false, then *deposit* must have been started at least *n* times more than *extract* finished.
line 7	This ensures at most one process can be in *deposit* at a time (mutual exclusion).
line 8	The next constraints apply to the procedure *extract*.
line 6	This invariant checks that there is at least one slot in the buffer that is full. If so, then *deposit* finished more times than *extract* started.
line 7	This ensures at most one process can be in *extract* at a time (mutual exclusion).
line 11	As with the previous procedure, this is called from outside the monitor by *buffer.extract*(…).
lines 12-15	This code actually extracts the item from the buffer. Note that the invariant guarantees mutual exclusion.
lines 23-25	This is the initialization part.

First Readers Writers Problem

Introduction

This uses invariant expressions to solve the first readers writers problem.

Algorithm

```
1   readerwriter: invariant module
2   invariant read
3         CurrentCount(write) = 0;
4   invariant write
5         CurrentCount(write) + CurrentCount(read) = 0;
6   procedure entry read;
7         … read from shared data …
8   end;
9   procedure entry write;
10        … write to shared data …
11  end;
12  begin
13  end.
```

Comments

lines 2-3	This states the condition that must hold whenever the procedure *read* is executed; it requires that no processes be executing write. Note this means readers will have priority over writers when a reader is presently reading; it says nothing about what happens if a reader and a writer call the module at the same time.
lines 4-5	This states the condition that must hold whenever the procedure *write* is executed; it requires that no processes be executing either *read* or *write*.
lines 6-11	Here, the routines simply do the reading and writing.
lines 12-13	The initialization part of the module; as there are no variables in it, this part is empty.

Producer Consumer Process

Introduction

This uses Hoare's CSP language to solve the producer consumer problem.

Algorithm

This process manages the buffer; call it *boundedbuffer*.

```
 1   buffer: (0..9) item;
 2   in, out: integer;
 3   in := 0;
 4   out := 0;
 5   *[in < out + n; producer ? buffer(in mod n)
 6          ■ in := in + 1
 7          → out < in; consumer ? more()
 8                 ■  consumer ! buffer(out mod n);
 9                 out := out + 1
10    ]
```

Comments

lines 1-2: Here, *buffer* is the buffer, *in* the number of items put into the buffer, and *out* the number of items extracted. The producer process outputs an item *nextp* to this process by:

bounded-buffer ! *nextp*;

and the consumer process outputs an item *nextc* to this process by:

bounded-buffer ! *more*(); *bounded-buffer* ? *nextc*;

(*more*() is there because CSP does not allow output commands in guards.)

lines 3-4: These just initialize *in* and *out*.

lines 5-6: If there is room for another item in the buffer (*in* < *out* + *n*), wait for the producer to produce something and deposit it in an empty buffer slot (*producer* ? *buffer*(*in* **mod** *n*)) and indicate that slot is now used (*in* := *in* + 1).

lines 7-9: If the buffer is full (*out* < *in*), wait until the consumer asks for something (*consumer* ? *more*()), then output the next element of the buffer (*consumer* ! *buffer*(*out* **mod** *n*)), and indicate it has been extracted (*out* := *out* + 1).

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.

```
 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                          end;
16                          in := (in + 1) mod n;
17                          count := count + 1;
18                    or when count > 0 =>
19                          accept extract(data: out item) do
20                                data := buffer[out];
21                          end;
22                          out := (out + 1) mod n;
23                          count := count - 1;
24              end select;
25        end loop;
26  end.
```

The producer deposits an item into the buffer with

```
27        boundedbuffer.deposit(nextp);
```

and the consumer extracts an item from the buffer with

```
28        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 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.)