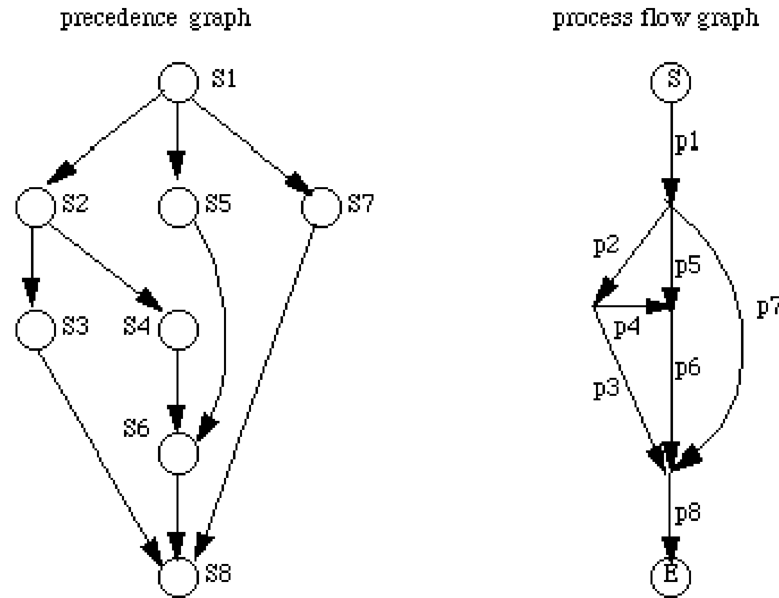
Improper Nesting Example

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 $S(P(p_1,p_2),p_3)$.

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 examples 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 p_i and p_j have at least 1 more process p_k starting or finishing at the node $n_i j$ 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 is not $S(p_i, p_j)$ either. □

Synchronization Problem

This problem is called the Bounded Buffer Problem or the Producer/Consumer Problem. The producer process writes items into a finite buffer, and the consumer process reads them. All the variables are shared.

First, the shared variables:

```
struct item buffer[n];
int in, out, counter;
```

`buffer` is the shared buffer. `counter` is the number of elements currently in the shared buffer. `in` is the index of the element into which the next item is to be placed, and `out` is the index of the element from which the next item is to be removed.

Now, the producer process code; we only list the code that operates on the shared variables.

```
1  loop {
2          p = produce_item();
3          while (counter == n)
4                  /* do nothing */ ;
5          buffer[in] = p;
6          in = (in + 1) mod n;
7          counter++;
8  }
```

The consumer process code is similar.

```
9  loop {
10         while (counter == 0)
11                 /* do nothing */ ;
12         p = buffer[out];
13         out = (out + 1) mod n;
14         counter--;
15         consume_item(p);
16 }
```

If each loop is executed separately, these processes work as expected. But if they are intermingled, the result may be incorrect.

As an example, suppose both processes try to alter count at the same time. Let's say the compiler compiled the statements into the following:

p_1	$r_1 = \texttt{counter};$	c_1	$r_2 = \texttt{counter};$
p_2	$r_1 = r_1 + 1;$	c_2	$r_2 = r_2 - 1;$
p_3	$\texttt{counter} = r_1;$	c_3	$\texttt{counter} = r_2;$

Let `counter` be 3 when these are executed. Then

- $c_1\ c_2\ p_1\ p_2\ p_3\ c_3$ results in `counter` being 2.
- $c_1\ c_2\ c_3\ p_1\ p_2\ p_3$ results in `counter` being 3.
- $c_1\ c_2\ p_1\ p_2\ c_3\ p_3$ results in `counter` being 4.

The problem is that two processes manipulated the variable `counter` simultaneously. Clearly, we need to ensure just one process does.

Classical Synchronization Problems

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:

1. *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.
2. *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. A solution must prevent both.

Solving the Critical Section Problem for 2 Processes

This handout presents several ***proposed*** solutions to the 2 process critical section problem. We will analyze them in class. In these solutions, one process is numbered 0 and the other is numbered 1. The variable `i` holds the number corresponding to the process executing the code, and the variable `j` holds the number corresponding to the other process. All the code shown is shared by both processes, but the variables `i` and `j` hold different values.

First Proposed Solution

Here, `turn` contains the number of the process whose turn it is to execute the critical section.

```
1 shared int turn;
2 loop {
3         while (turn != i) /* do nothing * / ;
4         /* critical section here */
5         turn = j;
6         /* remainder section here */
7 }
```

Second Proposed Solution

Here, `inCS[0]` is **`true`** when process 0 is in the critical section, and **`false`** otherwise. A similar statement holds for `inCS[1]`.

```
1 shared bool inCS[0..1] = { false, false };
2 loop {
3         while (inCS[j]) /* do nothing */ ;
4         inCS[i] = true;
5         /* critical section here */
6         inCS[i] = false;
7         /* remainder section here */
8 }
```

Third Proposed Solution

Here, `interested[0]` is **`true`** when process 0 wants to enter the critical section, and **`false`** otherwise. A similar statement holds for `interested[1]`.

```
1 shared bool interested[0..1] = { false, false };
2 loop {
3         interested[i] = true;
4         while (interested[j]) /* do nothing */ ;
5         /* critical section here */
6         interested[i] = false;
7         /* remainder section here */
8 }
```

Fourth Proposed Solution

This combines the first and third proposed solutions.

```
 1 shared bool interested[0..1]; = { false, false };
 2 shared int turn;
 3 loop{
 4         interested[i] = true;
 5         turn = j;
 6         while (interested[j] && turn == j) /* do nothing */ ;
 7         /* critical section here */
 8         interested[i] = false;
 9         /* remainder section here */
10 }
```

Bakery Algorithm

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.

```
1  shared bool choosing[n-1];
2  shared int number[n-1];
3  loop {
4          choosing[i] = true;
5          number[i] = max(number[0], number[1], ..., number[n-1]) + 1;
6          choosing[i] = false;
7          for(j = 0; j < n; j++) {
8                  while choosing[j]
9                          /* nothing */;
10                 while number[j] != 0 and (number[j], j) < (number[i], i)
11                          /* nothing */;
12         }
13         /* critical section goes here */
14         number[i] = 0;
15         /* remainder section */
16 }
```

lines 1–2: Here, `choosing[i]` is 1 if process i is choosing a number. The number that process i will use to enter the critical section is in `number[i]`; it is 0 if process 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 tries to assign a unique number to the process, process i (line 5); however, that does not always happen. Afterwards, process i indicates it is done (line 6).

lines 7–12: Now we select which process goes into the critical section. Process 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 process i tries to look at it, process i waits until it has done so before looking (line 8).

line 14: Now process i is no longer interested in entering its critical section, so it sets `number[i]` to 0.

Test and Set Solution

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:

```
bool TaS(bool *Lock)
{
        bool tmp = *Lock;
        *Lock = true;
        return(tmp);
}
```

The solution is:

```
 1  shared bool waiting[0..n-1];
 2  shared bool Lock;
 3  int (0..n-1) j;
 4  bool key;
 5  loop {      /* process Pi */
 6          waiting[i] = true;
 7          key = true;
 8          while waiting[i] and key
 9                  key = TaS(Lock);
10          waiting[i] = false;
11          /* critical section here */
12          j = i + 1 mod n;
13          while j != i and not waiting[j]
14                  j = (j + 1) mod n;
15          if (j == i)
16                  Lock = false;
17          else
18                  waiting[j] = false;
19          /* remainder section here */
20  }
```

lines 1–2: These are global to all processes, and are all initialized to `false`.

lines 3–4: These are local to each process*i* and are uninitialized.

lines 5–10: This is the entry section. Basically, waiting[i] is true as long as process *i* is trying to get into its critical section; if any other process is in that section, then `Lock` will also be true, and process *i* will loop in lines 8-9. Once process *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 process *i* 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 process *j* is waiting for entry, process *i* simply changes `waiting[j]` to `false` to allow process *j* to enter the critical section (lines 17–18).

Producer/Consumer Problem with Semaphores

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

```
1  semaphore full = 0, empty = n, mutex = 1;
2  item nextp, nextc;
3  shared item buffer[0..n-1];
4
5  parbegin
6          loop {            /* producer process */
7                  /* produce an item in nextp */
8                  down(empty);
9                  down(mutex);
10                 /* deposit nextp in buffer */
11                 up(mutex);
12                 up(full);
13         }
14         loop {            /* consumer process */
15                 down(full);
16                 down(mutex);
17                 /* extract an item in nextc */
18                 up(mutex);
19                 up(empty);
20                 /* consume the item in nextc */
21         }
22 parend
```

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 (initially 0), `empty` is a semaphore the value of which is the number of emoty slots in the buffer (initially *n*), and `mutex` is a semaphore used to enforce mutual exclusion (so only one process can access the buffer at a time; initially 1). `nextp` and `nextc` are the items produced by the producer and consumed by the consumer, respectively.

line 7 Since the buffer is not accessed while the item is produced, we don't need to put semaphores around this part.

lines 8–10 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 11–12 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`)).

lines 15–17 Extracting an item from the buffer also requires 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 (*down*(`full`)). 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 extract the item.

lines 18–19 As the consumer 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 extracted another item into the buffer (*up*(`empty`)).

line 20: Since the buffer is not accessed while the item is consumed, we do not need to put semaphores around this part.

First Readers Writers Problem with Semaphores

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

```
 1  semaphore wrt = 1, mutex = 1;
 2  shared int readcount = 0;
 3
 4  parbegin
 5          loop {          /* reader process */
 6                  /* do something */
 7                  down(mutex);
 8                  readcount++;
 9                  if (readcount == 1)
10                          down(wrt);
11                  up(mutex);
12                  /* read the file */
13                  down(mutex);
14                  readcount--;
15                  if (readcount == 0)
16                          up(wrt);
17                  up(mutex);
18                  /* do something else */
19          }
20          loop {          /* writer process */
21                  /* do something */
22                  down(wrt);
23                  /* write to the file */
24                  up(wrt);
25                  /* do something else */
26          }
27  parend;
```

lines 1–2 Here, `readcount` (initialized to 0) contains the number of processes reading the file, and `mutex` (initialized to 1) is a semaphore used to provide mutual exclusion when `readcount` is incremented or decremented. The semaphore `wrt` (initialized to 1) 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.

line 6 Since the file is not accessed here, we don't need to put semaphores around this part.

lines 7–11 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, `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` (*up*(`mutex`)) and proceeds to read from the file.

lines 13–17 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 18 Since the file is not accessed here, we don't need to put semaphores around this part.

lines 22–23 The writer process waits (*down*(`wrt`)) until no other process is accessing the file; it then proceeds to write to the file.

line 24 When the writer is done writing to the file, it signals that anyone who wishes to access the file may do so (*up*(`wrt`)).

Producer Consumer Problem with Monitors

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

```
1  monitor buffer {
2          item slots[0..n-1];
3          int count, in, out;
4          condition notempty, notfull;
5
6          entry deposit(item data) {
7                  if (count == n)
8                          notfull.wait;
9                  slots[in] = data;
10                 in = (in + 1) % n;
11                 count++;
12                 notempty.signal;
13         }
14
15         entry extract(item *data) {
16                 if (count == 0)
17                         notempty.wait;
18                 *data = slots[out];
19                 out = (out + 1) % n;
20                 count--;
21                 notfull.signal;
22         }
23
24         count = in = out = 0;
25 }
```

lines 2–4 `slots` is the 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.We care about whether the buffer is not full (represented by the condition variable `notfull`) or not empty (represented by the condition variable `notempty`).

line 6 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 15 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 24 This is the initialization part.

First Readers Writers Problem with Monitors

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

```
1  monitor readerwriter {
2          int readcount;
3          bool writing;
4          condition oktoread, oktowrite;
5
6          entry beginread(void) {
7                  readcount++;
8                  if (writing)
9                          oktoread.wait;
10         }
11
12         entry endread(void) {
13                 readcount--;
14                 if (readcount == 0)
15                         oktowrite.signal;
16         }
17
18         entry beginwrite(void) {
19                 if ((readcount > 0) || writing)
20                         oktowrite.wait;
21                 writing = true;
22         }
23
24         entry endwrite(void) {
25                 int i;
26
27                 writing = false;
28                 if (readcount > 0) {
29                         for(i = 1; i <= readcount; i++)
30                                 oktoread.signal;
31                 }
32                 else
33                         oktowrite.signal;
34         }
35
36         readcount = 0;
37         writing = false;
38 }
```

lines 2–4: `readcount` holds the number of processes reading the file. `writing` is true when a writer is writing to the file. `oktoread` and `oktowrite` correspond to the logical conditions of being able to read and write the file.

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 27–33 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.

lines 36–37 This initializes the variables.

Implementing Monitors with Semaphores

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.

```
1  semaphore mutex, urgent, xcond;
2  int urgentcount, xcondcount;
```

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

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

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

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

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

```
16 urgentcount++;
17 if (xcondcount > 0) {
18         up(xcond);
19         down(urgent);
20 }
21 urgentcount--;
```

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 enforces 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 is used to block processes doing waits on the condition variable x. If there is more than one such condition variable, a corresponding semaphore for each condition variable must be created. 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 so are 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 (*down*(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 (*up*(urgent)). Otherwise, one of the processes trying to enter the monitor may do so (*up*(mutex)).

lines 9–15 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 (*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*(xcond)) before suspending itself (*down*(urgent)). When restarted, the process indicates it is no longer suspended (by subtracting 1 from urgentcount).

Monitors and Priority Waits

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.

```
1   monitor alarmclock {
2           int now;
3           condition wakeup;
4
5           entry wakeme(int n) {
6                   alarmsetting = now + n;
7                   while (now < alarmsetting)
8                           wakeup.wait(alarmsetting);
9                   wakeup.signal;
10          }
11
12          entry tick(void) {
13                  now++;
14                  wakeup.signal;
15          }
16  }
```

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.

lines 13–14 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.

Producer Consumer Problem with IPC

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.

```
1  void producer(void) {
2          item nextp;
3
4          while (1) {
5                  /* produce item in nextp */
6                  send("Consumerprocess", nextp);
7          }
8  }
9
10 void consumer(void) {
11         item nextc;
12
13         while (1) {
14                 receive("Producerprocess", nextc);
15                 /* consume item in nextc */
16         }
17 }
18
19 producer Producerprocess;
20 consumer Consumerprocess;
21
22 parbegin
23         Producerprocess;
24         Consumerprocess;
25 parend
```

line 1 Here, `nextp` is the item the producer produces.

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."

line 1 Here, `nextc` is the item the consumer consumes.

lines 9–15 This code simply receives items from the producer process (named *Producerprocess*) and consumes them. When the *receive* statement is executed, if 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*.