◇ Semaphores ◇

Semaphores are a new datatype introduced into languages such as Pascal-FC to support mutual exclusion and synchronisation between processes.

A semaphore stores a value (a non-negative integer) and there are two operations which can be applied to it: `wait` and `signal`.

If `s` is a semaphore then:

`signal(s)` increases the value of `s` by 1

`wait(s)` decreases the value of `s` by 1 as soon as the result would be non-negative.

i.e. `wait(s)` waits until the value of `s` is at least 1, then subtracts 1 from the value.

The idea is that semaphores can be used safely by many processes running in parallel, i.e. there should be no problems of mutual exclusion when manipulating a semaphore. This means that:

`signal` must be implemented as a single indivisible action;

the conditional statement

```
if s > 0 then s := s - 1
```

required to implement `wait(s)` must also be an indivisible action.

`wait` could be implemented by using busy waiting, but usually a waiting process would be *blocked* — unable to be executed — until the semaphore becomes non-zero.

Semaphores can be used to implement mutual exclusion; this might seem circular because implementing semaphores requires mutual exclusion, but the point is that the mutual exclusion required by semaphores only involves a very small critical section and it is much easier to handle it at the machine or operating system level.

◇ Mutual Exclusion with Semaphores ◇

Declare a semaphore `s` globally. In each process with a critical section, the entry protocol is `wait(s)` and the exit protocol is `signal(s)`.

The semaphore needs to be initialised to the value 1. The `initial(s,v)` function initialises `s` with the value `v`. Semaphores must be initialised *before* any processes which use them are activated.

```
program gardens2;
(*
semaphore solution to
Ornamental Gardens problem

File is
/CS/ftp/pub/CS375/mutex/gardens2.pfc
*)

var
  count: integer;
  s: semaphore;

process turnstile1;
var
  loop: integer;
begin
  for loop := 1 to 20 do
    begin
    wait(s);
    count := count + 1;
    signal(s)
    end
end;  (* turnstile1 *)

process turnstile2;
(* The same definition as turnstile1 *)
```

```
begin
  count := 0;
  initial(s,1);
  cobegin
    turnstile1;
    turnstile2
  coend;
  writeln('Total admitted: ',count)
end.
```

It is straightforward to prove that this program obeys mutual exclusion, so we will do it.

First of all, at all times

$$s \geqslant 0$$

(this is guaranteed by the implementation of semaphores).

Also, if we write s_0 for the initial value of s, $\#signals$ for the number of `signal(s)` operations carried out so far, and $\#waits$ for the number of *completed* `wait(s)` operations, then

$$s = s_0 + \#signals - \#waits$$

Writing $\#CS$ for the number of processes in their critical sections at any given time, we have

$$\#CS = \#waits - \#signals$$

because each `wait` corresponds to a process entering its critical section and each `signal` corresponds to a process leaving its critical section.

From

$$\#CS = \#waits - \#signals$$

and

$$s = s_0 + \#signals - \#waits$$

we deduce

$$s = 1 - \#CS$$

because $s_0 = 1$ and therefore

$$\#CS + s = 1.$$

Because $\#CS$ and s are both non-negative, this means that

$$\#CS \leqslant 1$$

(which is what we wanted to prove), and also

$$s \leqslant 1$$

which means that the semaphore s only ever has values 0 or 1.

Notice that if we initialised s to 0 then we would get $\#CS = 0$ always, i.e. neither process would ever be allowed into its critical section. If we initialised s to any value larger than 1, then mutual exclusion would not be guaranteed.

◇ Producer-Consumer Problems ◇

Consider a situation in which one process *produces* data while another process *consumes* it. Examples might be:

- ◇ A computer generating documents to be printed, and a printer printing them;
- ◇ A tokeniser (lexical analyser) producing tokens (in the syntax of some programming language) and a parser receiving them;
- ◇ A mailserver sending messages to a mail router.

To allow for differences in processing speeds of the producer and consumer, there is a buffer between them, so that if the consumer is temporarily slower than the producer, the producer is not held up. However, the buffer has a finite capacity, so we would like to producer to be blocked whenever the buffer is full; this gives the connsumer a chance to catch up.

In the following example the producer produces the letters from 'a' to 'z', and the consumer simply receives them. The buffer is implemented by means of an array. Three semaphores are used:

- `mutex`, a binary semaphore used to ensure that the buffer is accessed under mutual exclusion
- `spacesleft`, a general semaphore indicating the number of free spaces in the buffer
- `itemsready`, a general semaphore indicating the number of items in the buffer.

Before adding an item to the buffer, the producer must wait on `spacesleft`. Before removing an item from the buffer, the consumer must wait on `itemsready`. Adding and removing items are critical sections, protected by `wait`s on `mutex`.

```
program pcsem;
(*
semaphore solution to
producer-consumer problem
*)

const
  buffmax = 4;
var
  buffer: array[0..buffmax] of char;
  nextin, nextout: integer;
  spacesleft, itemsready: semaphore;
  mutex: semaphore;
```

```
procedure put(ch: char);
begin
  buffer[nextin] := ch;
  nextin := (nextin + 1) mod (buffmax + 1)
end;  (* put *)

procedure take(var ch: char);
begin
  ch := buffer[nextout];
  nextout := (nextout + 1) mod (buffmax + 1)
end;  (* take *)

process producer;
var
  local: char;
begin
  for local := ’a’ to ’z’ do
    begin
    wait(spacesleft);
    wait(mutex);
    put(local);
    signal(mutex);
    signal(itemsready)
    end
end;  (* producer *)
```

```
process consumer;
var
  local: char;
begin
  repeat
    begin
    wait(itemsready);
    wait(mutex);
    take(local);
    signal(mutex);
    signal(spacesleft);
    write(local);
    end
  until local = 'z';
end;  (* consumer *)

begin
  initial(spacesleft,buffmax + 1);
  initial(itemsready,0);
  initial(mutex,1);
  nextin := 0;
  nextout := 0;
  cobegin
    producer;
    consumer
  coend
end.
```

◇ Binary Semaphores ◇

A semaphore which only takes values 0 or 1 is called a *binary* semaphore. A semaphore which can take any value is called a *general* semaphore. Pascal-FC does not distinguish between the two, but in our example programs we can see which are which.

Binary semaphores are seemingly less general then general semaphores, but it turns out that binary semaphores can be used to implement general semaphores.

Suppose that there is a type `BinSemaphore` of binary semaphores. A type `GenSemaphore` of general semaphores can be defined as follows.

```
type GenSemaphore =
  record
    mutex : BinSemaphore;
    delay : BinSemaphore;
    count : integer
  end;
```

The component `mutex` is used to provide mutual exclusion over the operations on `GenSemaphore`, and must be initialised to 1. The component `count` holds the integer value of the semaphore. The `delay` component is used to block any process which calls a `wait` operation when the value of `count` is 0.

```
procedure GenWait(var s:GenSemaphore);
begin
  wait(s.delay);
  wait(s.mutex);
  s.count := s.count - 1;
  if s.count > 0 then
    signal(s.delay);
  signal(s.mutex)
end;

procedure GenSignal(var s:GenSemaphore);
begin
  wait(s.mutex);
  s.count := s.count + 1;
  if s.count = 1 then
    signal(s.delay);
  signal(s.mutex)
end;
```

◇ Dining Philosophers with Semaphores ◇

The Dining Philosophers can be implemented by representing the forks as binary semaphores. In order to pick up a fork, a philosopher executes `wait` on the corresponding semaphore; to put it down, he executes `signal`.

The butler can be represented by a general semaphore, initialised to 1 less than the number of chairs. Before sitting down, a philosopher `wait`s on this semaphore; when getting up, he executes `signal`.

◇ Semaphore Philosophers ◇

```
program philsem1;

(* Dining Philosophers - semaphore version 1
File is /CS/ftp/pub/CS375/sems/philsem1.pfc
*)

const
   N = 5;
var
   fork : array [1..N] of semaphore;
     (* binary *)
   i : integer;

process type philosophers(name : integer);
begin
   repeat
      sleep(random(3));   (* THINKING *)
      wait(fork[name]);
      wait(fork[(name mod N) + 1]);
      sleep(random(3));   (* EATING *)
      writeln(name);
      signal(fork[name]);
      signal(fork[(name mod N) + 1]);
   forever
end;  (* philosophers *)
```

```
var
   phils: array[1..N] of philosophers;

begin
   for i := 1 to N do
      initial(fork[i],1);
   cobegin
      for i := 1 to N do
         phils[i](i);
   coend
end.
```

◇ Semaphore Butler ◇

```
program philsem2;
(* Dining Philosophers - semaphore version 2
File is /CS/ftp/pub/CS375/sems/philsem2.pfc *)
const
   N = 5;
var
   fork : array [1..N] of semaphore;
     (* binary *)
   freechairs : semaphore;  (* general *)
   i : integer;

process type philosophers(name : integer);
begin
   repeat
      sleep(random(3));   (* THINKING *)
      wait(freechairs);
      wait(fork[name]);
      wait(fork[(name mod N) + 1]);
      sleep(random(3));   (* EATING *)
      writeln(name);
      signal(fork[name]);
      signal(fork[(name mod N) + 1]);
      signal(freechairs)
   forever
end;  (* philosophers *)
```

◇ Monitors ◇

Semaphores enable mutually exclusive access to data to be programmed, and also support synchronisation between processes. However, they suffer from a number of problems:

- they are low-level
- it's easy to make mistakes, e.g. `waiting` at the wrong time
- code relating to mutual exclusion is distributed throughout the program.

Monitors provide a higher-level, more structured solution.

A *monitor* consists of

- some data
- some procedures or functions which manipulate the data.

The implementation guarantees that code within a monitor is executed under mutual exclusion, i.e. if one process is executing a monitor function then other processes are prevented from executing *any* monitor function.

When a monitor is used, all the operations manipulating a shared data structure are defined in the same place, and the programmer does not have to worry about using semaphores to ensure mutually exclusive access to the data.

Here is the Ornamental Gardens program, implemented with a monitor.

```
program gardens4;
const
  nprocs = 2;
var
  procloop: integer;

process type turnstype;
var
  loop: integer;
begin
  for loop := 1 to 20 do
    counter.inc
end;  (* turnstype *)
```

```
monitor counter;
export
  inc, print;
var
  count: integer;

procedure inc;
begin
  count := count + 1
end;  (* inc *)

procedure print;
begin
  writeln('Total admitted - ',count:1)
end;  (* print *)

begin  (* body *)
  count := 0
end;  (* monitor counter *)
```

```
var
  turnstile: array[1..nprocs] of turnstype;

begin
  cobegin
    for procloop := 1 to nprocs do
      turnstile[procloop]
  coend;
  counter.print
end.
```

Points to note:

- only the `exported` operations are visible outside the monitor
- the body of the monitor (`count := 0`) is executed just once, before the monitor is used
- the `print` procedure is part of the monitor, even though mutual exclusion is not required when executing it

Because monitors incorporate data and functions, they look rather like objects. Indeed, the development of monitors (by Per Brinch Hansen and Tony Hoare in the early 1970s) was partly inspired by Smalltalk, an early object oriented language.

In C++ terms, the exported functions are `public`; all other functions, and all the data, are `private`.

◇ Monitors in Java ◇

Java provides concurrency via lightweight processes (called *threads*). To support mutual exclusion between threads, Java has the concept of a *synchronized method*. Synchronized methods (designated as synchronized by the programmer) behave like the exported functions of a monitor: if one thread (process) calls a synchronized method of an object, then no other thread can call *any* synchronized method of the same object until the original call has finished.

A class can define both synchronized and non-synchronized methods, so in the ornamental gardens program there would be no need to make the `print` procedure synchronized. It is up to the programmer to decide which methods need to be synchronized.

Amusingly, one Java book states that "Java provides unique language-level support for [mutual exclusion]".

◇ Producer-Consumer with Monitors ◇

Now we can attempt to implement the producer-consumer program using a monitor. The monitor takes care of the necessary mutually exclusive access to the buffer, but we are also using the semaphores `itemsready` and `spacesleft` as before.

```
program pcmon1;
(* producer-consumer problem  -
first attempt at
monitor solution (incorrect)

file is
/CS/ftp/pub/CS375/mutex/pcmon1.pfc *)

monitor buffer;
export
  put, take;
const
  buffmax = 4;
var
  store: array[0..buffmax] of char;
  count: integer;
  spacesleft, itemsready: semaphore;
  nextin, nextout: integer;
```

```
procedure put(ch: char);
begin
  wait(spacesleft);
  store[nextin] := ch;
  nextin := (nextin + 1) mod (buffmax + 1);
  signal(itemsready)
end;  (* put *)

procedure take(var ch: char);
begin
  wait(itemsready);
  ch := store[nextout];
  nextout := (nextout + 1) mod (buffmax + 1);
  signal(spacesleft)
end;  (* take *)

begin  (* body of buffer *)
  initial(itemsready,0);
  initial(spacesleft,buffmax+1);
  nextin := 0;
  nextout := 0
end;  (* buffer *)
```

```
process producer;
var
  local: char;
begin
  for local := ’a’ to ’z’ do
    buffer.put(local);
end;  (* producer *)

process consumer;
var
  ch: char;
begin
  repeat
    buffer.take(ch);
    write(ch)
  until ch = ’z’;
  writeln
end;  (* consumer *)

begin  (* main *)
  cobegin
    producer;
    consumer
  coend
end.
```

◇ Condition Variables ◇

The following sequence of events shows that there is a problem with this program.

1. Initially the buffer is empty, and `itemsready` = 0.
2. The producer calls `buffer.take`.
3. Inside `buffer.take`, the producer does `wait` on `itemsready`.
4. The consumer cannot call `buffer.put` because the producer is in the monitor.

The result is either a livelock, if the consumer is busy waiting on `itemsready`, or a form of deadlock with no process able to be executed.

Semaphores are not designed to work with monitors in this way. Instead, we need to use *condition variables*. A condition variable is like a binary semaphore in that a process can be blocked by it. It contains a queue of blocked processes (either a FIFO queue or possibly a priority queue, depending on whether the operating system/language supports different priorities for processes).

The crucial difference is that if a process enters a monitor, then finds itself blocked by a condition variable, it leaves the monitor until it becomes unblocked. This allows other processes to enter the monitor in the meantime.

The operations on a condition variable are `delay`, which causes a process to become blocked immediately (and join the queue of blocked processes on that condition variable), and `resume`, which allows the first blocked process to execute.

The previous program does not compile, because Pascal-FC does not allow semaphores to be declared within a monitor. Instead, we can replace the semaphores by condition variables, as follows.

```
var
  store: array[0..buffmax] of char;
  count: integer;
  notfull, notempty: condition;
  nextin, nextout: integer;
```

```
procedure put(ch: char);
begin
  if count > buffmax then
    delay(notfull);
  store[nextin] := ch;
  count := count + 1;
  nextin := (nextin + 1) mod (buffmax + 1);
  resume(notempty)
end;  (* put *)

procedure take(var ch: char);
begin
  if count = 0 then
    delay(notempty);
  ch := store[nextout];
  count := count - 1;
  nextout := (nextout + 1) mod (buffmax + 1);
  resume(notfull)
end;  (* take *)
```

Java has one condition variable implicitly associated with each monitor, and uses methods `wait` and `notify` to control blocking.