◇ Ornamental Gardens Output ◇

Recall that the program `gardens1` has a global variable and two concurrent processes. Each process increments the variable 20 times. When both processes have finished, the final value of the variable is displayed.

We might have expected that the displayed value would be 40, and indeed this was our intention — the program is supposed to be counting the visitors entering the gardens through both gates.

However, running the program produced unexpected results. The final total was different nearly every time. The expected output of 40 might have appeared a few times, but usually the result was less than 40, and in some cases it was even less than 20.

What is going on?

◇ The Multiple Update Problem ◇

The Ornamental Gardens illustrates the *multiple update problem*. To understand it, we need to consider how the instruction

```
count := count + 1
```

is executed. If the hardware could execute this instruction as a single, indivisible (or *atomic*) action, the program would work correctly, because

- ◇ if running on a single processor, as in Pascal-FC, only one instruction can be executed at a time
- ◇ if running on multiple processors, the hardware would normally ensure that simultaneous accesses to the same memory location take place in one order or another.

However, executing `count := count + 1` means executing the following separate steps:

1. load the value of `count` from memory into a register
2. increment the value in the register
3. return the new value to memory, updating `count`

Suppose that `count` has the value 5, and `turnstile1` and `turnstile2` are both ready to do `count := count + 1`. One possible sequence of steps is as follows.

1. $T1$ loads `count` from memory, getting 5
2. $T2$ loads `count` from memory, getting 5
3. $T2$ increments this value to 6
4. $T1$ increments this value to 6
5. $T2$ stores 6 in `count`
6. $T1$ stores 6 in `count`

so the overall effect is that one increment has been lost. Any number of increments can be lost in this way — for example, between step 1 and step 4, $T2$ could do `count := count + 1` any number of times, and the effect of all of them would be lost when $T1$ reached step 6.

We need some way of making `count := count + 1` (or in general, an arbitrary block of code) appear to be an atomic action, so that it is not subject to interference. A piece of code which needs to appear atomic is called a *critical section* or *critical region*.

One example of a critical region is a piece of code which modifies a shared variable.

◇ Mutual Exclusion ◇

If two (or more) processes contain critical sections which would interfere if executed in an overlapping way, then we need to ensure *mutual exclusion* — it must be impossible for two processes to be in their critical sections simultaneously.

In the early days of concurrent programming (the early 1960s) it was not known whether it was possible to implement mutual exclusion purely in software, by suitable use of shared variables, or whether special-purpose hardware and additional programming language features would be required.

Eventually, however, mutual exclusion algorithms were developed. The first was discovered by Dekker in 1968. We will now look at a later algorithm, developed by Peterson in 1981.

Some of you might have come across Peterson's algorithm in the Operating Systems course last year.

Dekker's and Peterson's algorithms are for mutual exclusion between *two* processes.

◇ Shared Variables ◇

We will make the following assumptions about how shared variables work.

- ◇ If two processes simultaneously read a shared variable, there is no interference and both processes get the current value.
- ◇ If two processes simultaneously update a shared variable, then the variable gets one value or the other, not a mixture of the two (but we don't know which value it will be).
- ◇ If a shared variable is updated and read simultaneously, then the process reading the variable gets either the old value or the new value, not a mixture of the two (but we don't know which value it will be).

◇ Mutual Exclusion Algorithms ◇

The general form of a mutual exclusion algorithm is that each process will protect access to its critical section by executing an *entry protocol* before entering it, and a corresponding *exit protocol* when leaving. Thus a skeleton process definition is

```
process P;
begin
  repeat
    entry protocol;
    critical section;
    exit protocol;
    non-critical section;
  forever
end;
```

Typically, the non-critical section is much longer than the critical section. This means that the processes can execute independently for most of the time, and only occasionally need to coordinate their actions. Such processes are said to be *loosely coupled*. In our examples, however, we will ignore the non-critical sections. We will also assume that a process cannot fail in its critical section or in its entry and exit protocols.

◇ Requirements ◇

We can state some natural requirements for a mutual exclusion algorithm, in order to rule out unacceptable solutions. (For example, one way to guarantee mutual exclusion is to never execute one of the processes at all, but this is obviously unsatisfactory!)

1. At any given time, at most one process should be in its critical section.
2. If both processes are competing for entry into their critical section, the decision as to which should succeed cannot be postponed indefinitely.
3. If one process is in its non-critical section and the other process requests entry into its critical section, the request should succeed.

◇ Peterson's Algorithm ◇

```
program peterson;
(* Peterson's two-process mutual exclusion algorithm *)

var
  count, turn : integer;
  flag1, flag2: boolean;

process turnstile1;
var loop: integer;
begin
 for loop := 1 to 20 do
  begin
   (* entry protocol *)
   flag1:= true;    (* announce intent to enter *)
   turn:= 2;      (* give priority to other process *)
   while flag2 and (turn = 2) do
    null;
   (* end of entry protocol *)
   (* critical section *)
   count := count + 1;
   (* end of critical section *)
   (* exit protocol *)
   flag1:= false
   (* end of exit protocol *)
  end
end;

process turnstile2; (* similar *)
```

```
begin (* program *)
 count := 0;
 (* initialise variables for Peterson's algorithm *)
 turn := 1;
 flag1 := false;
 flag2 := false;
 (* start the processes *)
 cobegin
  turnstile1; turnstile2
 coend;
 writeln('Total admitted:  ',count)
end.
```

The complete program is

`/CS/ftp/pub/CS375/peterson.pfc`

You can try it out (remember to copy it into your own directory first) and see that the result is 40 every time.

◇ Do we trust it? ◇

If `peterson` is executed several times, the result always seems to be 40. But how can we be sure that it really will be 40 every time, given that we have already seen that the result of a concurrent program might vary from run to run? There are three possible ways of convincing ourselves that Peterson's algorithm is correct.

1. Execute the program many more times, until we are satisfied that we have seen a representative sample of executions (whatever that means!).
2. Produce a mathematical proof that our 3 requirements are satisfied.
3. Execute the program in some non-standard way, which guarantees to test all possible execution sequences.

Approach 1 is obviously not completely satisfactory, because we can never be sure that we have seen enough executions.

Approach 2 is ideal in principle, but it is possible to make mistakes in mathematical proofs (and indeed the original published proof of correctness of Lamport's "bakery algorithm", another mutual exclusion algorithm, contained errors, although the algorithm was in fact correct and the proof was later corrected).

We will look at approach 3 later in the course, by constructing a model of Peterson's algorithm in CSP and analysing it with the FDR tool. We can be very confident in the result, but as we will see, this approach might not work for large systems.