

CPSC 457/597 (Winter 2001)
Assignment 1 (7.5 marks)
Process Coordination

Due January 29, 2001

In this assignment, you will experiment with two processes, a consumer process (consumer) and a producer process (producer), the execution of whose instructions is interleaved in time under the control of a Kernel, which you will also write, in order to demonstrate and prevent timing errors due to critical sections.¹

You will write two major methods or functions:

- **Producer:** Produces items (characters read from a fixed structure `source`), one at a time, and places those in a shared `buffer`.
- **Consumer:** Consumes the produced items from the buffer, simply by printing those on the screen.
- **Kernel :** Controls the interleaved execution of the producer and consumer processes.

The Kernel method will, thus, simulate the kernel of a multitasking operating system. It will have a scheduling capability (you will only use Round Robin for this assignment.), and, therefore, it must maintain a Process Control Block (PCB) for each process. When a context-switch takes place, the Kernel saves the state (stored in its PCB) of the switched out process, loads the state of the switched in process, and let the latter run.

PART I (3 marks) Write the three methods Producer, Consumer, and Kernel. Use the producer and consumer code on page 102 of your text (however, these must be written in an assembly-like code as explained below.)

1. In the producer and consumer methods, you write high-level code (C, C++, or Java) that is assembly-like. This means that you will have variables simulating registers R1, R2, R3, and R4. For instance, the high-level statement:

```
x = y + z;
```

would be written as:

```
R1 = y; R2 = z; R3 = R1 + R2; y = R3;
```

Branching, like assembly languages, must be simulated using the `goto` statement for the producer and consumer processes.² However, Kernel can use high-level branching constructs such as `while` and `if then`. So, the code:

```
if (in == out) <body>;
```

would be written (for the producer and consumer) as:

```
R1 = in;
R2 = out
if R1 != R2 goto c5;
<body>;
c5: ...
```

For another example, consider the following while loop:

¹One way to experiment with two or more processes is to use the UNIX `fork()` function that creates a child process (refer to the C example given under supplementary material on the course web page.) When such a child process is created and the two, parent and child, processes are executing concurrently, you will **not** have control over how UNIX schedules these two processes. In this assignment, you should be able to have full control over the way the producer and consumer are scheduled. Therefore, you must not use `fork()`.

²Gotos are considered to be a bad programming practice for regular programs.

```
while (in == out) <body>;
```

In the producer and consumer code, it will be written as:

```
c2: R1 = in;
R2 = out;
if (R1 != R2) goto c10;
<body>;
goto c2;
c10: ...
```

- Every “risky” instruction in the producer or consumer must be prefixed by a label, to allow the switched in process to proceed from where it was interrupted when it was switched out. So, you must also maintain two program counters, one for each producer (PC_p) and consumer (PC_c) process. Each time a producer or consumer executes an assembly-like instruction, the corresponding program counter is incremented. To simulate round robin scheduling, the Kernel gives each process a set of instructions slice, rather than giving them time slices (It is not as easy to go with the latter option.) So, the quantum for each process slice is defined by the variable or constant Q. You should maintain a counter for the number of instructions executed (RRC); when RRC becomes equal to Q, a context-switch should occur. Note that the counters PC_p, PC_c, and RRC are **not** part of the consumer and producer code. Therefore, incrementing those should be done without using registers.

At this point, your producer would look like the following (note that after each instruction PC_p and RRC are incremented and the latter is checked to determine if the producer burst of execution has reached its limit and context-switch is necessary.):

```
...
int in, out;
...
int producer(...)
{ ...
p1: R1 = in;          PCp++; RRC++; if (RRC == Q) return PCp;
p2: R1++;           PCp++; RRC++; if (RRC == Q) return PCp;
p3: R2 = in;        PCp++; RRC++; if (RRC == Q) return PCp;
p4: R3 = out;       PCp++; RRC++; if (RRC == Q) return PCp;
p5: R1 = R1 % R2;   PCp++; RRC++; if (RRC == Q) return PCp;
p6: if (R1 == R3)   PCp = 1; RRC++; if (RRC == Q) return PCp;
    [goto p1;]
p7: R1 = in;        PCp++; RRC++; if (RRC == Q) return PCp;
p8: buffer[R1] = source[i]; j = (j+1) % MAX_SRC_SIZE;
    PCp++; RRC++; if (RRC == Q) return PCp;
p9: R1++;           PCp++; RRC++; if (RRC == Q) return PCp;
p10: R2 = n;        PCp++; RRC++; if (RRC == Q) return PCp;
p11: R1 = R2 % R2;  PCp = 1; RRC++; if (RRC == Q) return PCp;
p12: in = R1;       PCp++; RRC++; if (RRC == Q) return PCp;
    [goto p1;]
...
return ...
```

The array `source[]` contains the characters that are fed to the producer (produced). It is local to the producer and the consumer has no access to it. So, `source[]` cannot be involved in timing errors, and access to it is done without using registers. To the contrary, variables `in`, `out`, and `buffer` are **shared** between the producer and consumer and access to these is done through registers.

The consumer code is similar with different instruction labels, say `c1`, `c2`, ...

Every time, a process is switched in, you have to make sure it starts where it was stopped. A `switch()` statement at the beginning of the process code should do the trick.

One more reminder that the Kernel uses normal high-level code (no registers; no `gotos`.)

3. The idea is that every Q th instruction of the producer or consumer, when the quantum expires, control goes to the Kernel, which switches control to the other process. The Kernel keeps a PCB for each process which contains the program counter and the artificial registers (R1 to R4). When a process exceeds its quantum, it is switched out and its state is saved into its PCB. The artificial registers are loaded from the PCB of the switched in process.
4. The producer and consumer are executed repeatedly, until the consumer is stopped by reading in an appropriate code. Set the max size of `buffer[]` to 8. Fill in `source[]` with: `The only thing that I know is that I do not know anything, thus said Socrates`. In the consumer, after a batch of these lines have been printed (say 20), a request for permission to continue should be displayed. Depending on the entered code, the consumer may proceed or not. Every time the consumer consumes an item (character), it prints that item on the screen. Make sure every sentence is written on a separate line.
Write the program with its three sections, and demonstrate no timing errors. Experiment with values for Q from 0 to the maximum number of producer or consumer instructions.

PART II (1.5 marks) Alter the program so that the producer and consumer reflect the versions on page 156. Demonstrate timing errors.

PART III (1.5 marks) Alter the program again so that the critical section th the consumer and producer versions used in PART II is protected using Peterson's Algorithm (Algorithm 3 on page 161). Peterson's Algorithm must be carried out using artificial registers (it is part of the producer and consumer code); furthermore, branching must be done using `goto`. Demonstrate that timing errors no longer occur.

PART IV (1.5 marks) Repeat PART III using Hyman's Algorithm (given below) instead of Peterson's. What do you notice?

Hyman's Algorithm (1966) for Two Processes

Processes have a 0 and 1 ids. The code shown is for process i ($j = 1 - i$)

shared variables

```
flag[0 .. 1] in {true, false}
turn in {0,1}
```

<entry>

```
flag[i] ← true
while turn ≠ i do
    while flag[j] do nothing end-while
    turn ← i
end-while
```

<critical section>

<exit>

```
flag[i] ← false
```