Process Synchronization

Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE451

---

Producer/Consumer Implementation

Producer process

```plaintext
var c : char
begin
loop
produce a character "c"
while nextIn+1 mod n = nextOut do
NOOP
end while
buf[nextIn] := c
nextIn := nextIn+1 mod n
end loop
end
```

Consumer process

```plaintext
var c : char
begin
loop
while nextIn = nextOut do
NOOP
end while
c := buf[nextOut]
nextOut := nextOut+1 mod n
consume a character "c"
end loop
end
```

globals buf : array [0..n-1] of char;
nextIn, nextOut : 0..n-1 := 0 n-1

...
Producer/Consumer Implementation with a shared counter

```
process Producer
  var c : char
begin
  loop
    <produce a character “c”>
    while count = n do
      NOOP
    end while
    buf[nextIn] := c
    nextIn := nextIn + 1 mod n
    count := count + 1
  end loop
end
```

```
process Consumer
  var c : char
begin
  loop
    while count = 0 do
      NOOP
    end while
    c := buf[nextOut]
    nextOut := nextOut + 1 mod n
    count := count - 1
    <consume a character “c”>
  end loop
end
```

```
globals
  buf : array [0..n-1] of char;
  nextIn, nextOut : 0..n-1 := 0
  count : integer := 0
```

The Critical Section Problem

- One implementation of the shared counter

```
process Producer
  begin :
    <count := count + 1>
    MOV R1, @count
    ADD R1, 1
    MOV @count, R1
  end
```

```
process Consumer
  begin :
    <count := count - 1>
    MOV R2, @count
    SUB R2, 1
    MOV @count, R2
  end
```

```
Algorithms for Mutual Exclusion

- General algorithm structure
  ```
  process P:
  
  begin
  loop
    Entry_Protocol
    <critical_section>
    Exit_Protocol
  end loop
  
  end
  ```

- Correctness conditions
  - Does it guarantee mutual exclusion?
  - Is it expedient?
  - Does it provide bounded waiting?

2-Process Mutual Exclusion

Algorithm 1

- Turn taking/strict alternation
  ```
  global var turn : int := 2

  process P1:
  begin
    loop
      while turn = 2 do
        NOOP
      end while
      <critical_section>
      turn := 2
      end loop
      end

  process P2:
  begin
    loop
      while turn = 1 do
        NOOP
      end while
      <critical_section>
      turn := 1
      end loop
      end
  ```
2-Process Mutual Exclusion

Algorithm 2

- Use status flags

```plaintext
global var inCS : array[1..2] of boolean := (FALSE,FALSE)

process P1
begin
loop
    inCS[1] := TRUE
<critical section>
    inCS[1] := FALSE
end loop
end P1

process P2
begin
loop
<critical section>
end loop
end P2
```

Algorithm 2a

- Move the setting of the status flags

```plaintext
global var inCS : array[1..2] of boolean := (FALSE,FALSE)

process P1
begin
loop
    inCS[1] := TRUE
    while inCS[2] do
        NOOP
    end while
<critical section>
    inCS[1] := FALSE
end loop
end P1

process P2
begin
loop
    while inCS[1] do
        NOOP
    end while
<critical section>
end loop
end P2
```
2-Process Mutual Exclusion
Algorithm 3

- New and improved use of status flags

```
global var inCS : array[1..2] of boolean := (FALSE, FALSE)

process P1
begin
loop
inCS[1] := TRUE
if NOT inCS[2] then
<critical section>
end if
inCS[1] := FALSE
end loop
end
```

```
global var inCS : array[1..2] of boolean := (FALSE, FALSE)

turn : integer := 2

process P2
begin
loop
turn := 2
while turn = 2 AND inCS[1] do
<critical section>
end while
end loop
end
```

2-Process Mutual Exclusion
Algorithm 4 (Peterson’s algorithm)

- Careful combination of alternation and status flags

```
global var inCS : array[1..2] of boolean := (FALSE, FALSE)
turn : integer := 2

process P1
begin
loop
inCS[1] := TRUE
turn := 1
while turn = 1 AND inCS[2] do
<critical section>
inCS[1] := FALSE
end while
end loop
end
```

```
global var inCS : array[1..2] of boolean := (FALSE, FALSE)
turn : integer := 2

process P2
begin
loop
turn := 2
while turn = 2 AND inCS[1] do
<critical section>
end while
end loop
end
```