

**EECS 338 Assignment #4 Solutions**  
*Spring'07*

(10 pts) **(1)** Give a semaphore-based implementation of the conditional critical region construct:

**region v do begin S1; await(B); S2; end;**

Answer:

```

wait(x-mutex);
S1;
if not B then begin
    x-count:=x-count + 1;
    signal(x-mutex);
    wait(x-wait);
    while not B do begin
        x-temp:=x-temp + 1;
        if x-temp < x-count then signal(x-wait) else signal(x-mutex);
        wait(x-wait);
    endwhile;
    x-count:=x-count - 1;
endif;
S2;
if x-count > 0 then begin x-temp:=0; signal(x-wait) end else signal(x-mutex);

```

Initially, x-mutex semaphore is 1; x-wait semaphore is 0, and integers x-count and x-temp are zeroes.

(10 pts) **(2)** Consider the following monitor implementation with a single condition variable x.

Initialization:

```

semaphore mutex := 1; semaphore next := 0; int next-count := 0;
semaphore x-sem := 0; int x-count := 0;

```

*For mutual exclusion, each procedure P is implemented as:*

```

wait (mutex);
    Body of P;
if next-count > 0 then signal(next) else signal (mutex);

```

*Each x.wait in a monitor procedure is implemented as:*

```

x-count := x-count +1;
if next-count > 0 then signal (next) else signal (mutex);
wait (x-sem);
x-count := x-count - 1;

```

*Each x.signal in a monitor procedure is implemented as:*

```

if x-count > 0 then begin
    next-count :=next-count + 1;
    signal (x-sem);
    wait (next); -----A
next-count := next-count -1;

```

**end;**

**(a)** Assume that process  $P_i$  is waiting on  $x$  inside a monitor procedure, and process  $P_j$  issues  $x.signal$ . Then what happens: Does  $P_i$  wait until  $P_j$  leaves the monitor, or vice versa? Explain your answer.

*$P_j$  waits at  $A$  until  $P_i$  leaves the monitor or blocks on another wait statement. At this time,  $P_j$  is released.*

**(b)** Assume that our monitor has two condition variables, namely  $x$  and  $y$ . Revise the above monitor implementation with semaphores (i.e., give implementations for the mutual exclusion,  $x.wait$ ,  $x.signal$ ,  $y.wait$ , and  $y.signal$ ).

**Answer:**

Initialization:

```
semaphore mutex := 1; semaphore next := 0; int next-count := 0;  
semaphore x-sem := 0; int x-count := 0;
```

*Each monitor procedure P:*

```
wait (mutex);  
    Body of P;  
if next-count > 0 then signal(next)  
    else signal (mutex);
```

*x.wait implementation:*

```
x-count := x-count + 1;  
if next-count > 0 then signal (next) else  
signal (mutex);  
wait (x-sem);  
x-count := x-count - 1;
```

*x.signal implementation:*

```
if x-count > 0 then begin  
    next-count := next-count + 1;  
    signal (x-sem);  
wait (next);  
next-count := next-count - 1;  
end;
```

*y.wait implementation:*

```
y-count := y-count + 1;  
if next-count > 0 then signal (next) else  
signal (mutex);  
wait (y-sem);  
y-count := y-count - 1;
```

*y.signal implementation:*

```
if y-count > 0 then begin  
    next-count := next-count + 1;  
    signal (y-sem);  
wait (next);  
next-count := next-count - 1;  
end;
```

### Question 3.a. Railroad Crossing Problem: Conditional Critical Region-based solution

#### Shared region variable:

```
var crossing: shared record begin
    enum light{green, red, yellow} := green;  int PassCarCount := 0;
    int TrainCount := 0;    Boolean TrainInProgress := False end
```

#### procedure Car

```
begin
```

```
    Do-Something;
```

```
region crossing when (light = green and PassCarCount < 4) do PassCarCount++; //Entry code.
```

```
    Pass-Crossing;
```

```
region crossing do PassCarCount--; // Housekeeping work. Exit code.
```

```
    Do-Something;
```

```
end
```

#### procedure Train

```
begin
```

```
    Do-Something;
```

```
region crossing do
```

```
    { TrainCount++; // Maintain the count of trains arriving.
```

```
    await (not TrainInProgress);
```

```
    do TrainInProgress := True; light := yellow} // Let the first train start moving. Block others.
```

```
    Approach-Crossing; // Enough time passes, and all crossing cars, if any, clear the tracks.
```

```
region crossing when PassCarCount=0 do light := red; // Given our problem specs, the condition
```

```
// "PassCarCount=0" is redundant.
```

```
    Pass-Crossing;
```

```
region crossing do // Housekeeping work. Exit code.
```

```
    { TrainCount--;
```

```
    TrainInProgress := False;
```

```
    if TrainCount = 0 then light := green
```

```
    else {light:=green; light:=yellow} // "light:=green" (superfluous really) is there to satisfy
```

```
    } // light rotation order: green → yellow → red → green
```

```
    Do-Something;
```

```
end
```

### Question 3.b. Railroad Crossing Problem: Monitor-Based Solution

#### monitor Railroad

```
procedure car-enter()
```

```
begin
```

```
if (PassCarCount ≥ 4 or light ≠ green)
```

```
    then car.wait;
```

```
PassCarCount++
```

```
end
```

```
procedure car-exit()
```

```
begin
```

```
    PassCarCount--;
```

```
    if ( light = green) then car.signal;
```

```
end
```

```
procedure train-approach()
```

```
begin
```

```
    TrainCount++;
```

```
    light := yellow;
```

```
    if TrainCount > 1 then train.wait;
```

```
end
```

```
procedure train-enter()
```

```
light := red;
```

```
procedure train-exit()
```

```
begin
```

```
    TrainCount--;
```

```
    if TrainCount > 0
```

```
        then {light := yellow; train.signal}
```

```
        else { light := green; car.signal; car.signal;
```

```
              car.signal; car.signal}
```

```
end
```

```
begin
```

```
    // Shared and condition variable initialization
```

```
int PassCarCount := 0;    TrainCount := 0;    enum light{red, green, yellow} := green;
```

```
condition car; train;
```

```
end.
```

#### USE:

##### Train:

```
RR of-type Railroad;
```

```
    Do-Something;
```

```
train-approach();
```

```
    Approach-crossing;
```

```
train-enter();
```

```
    Pass-crossing;
```

```
train-exit();
```

##### Car:

```
RR of-type Railroad;
```

```
    Do-Something;
```

```
car-enter();
```

```
    Cross-tracks;
```

```
car-exit();
```

```
    Do-Something;
```

**Signaling/signaled process policy:** Signaling process continues to execute; and, signaled process is blocked until signaled process is blocked or leaves the monitor.