ESE 605 Homework 3

Stephen Phillips

February 14, 2015

Problem 1

What does the code *incr(i,i)* do on a pass by reference argument? On a pass by copy/restore?

If the pseudo-code is as follows:

```
function incr(reference i, reference j):
  i = i + 1;
  j = j + 1;
end function
```

Then a pass by reference for *incr(i,i)* would increment i twice. On pass by copy/restore it should do the same. The reference given to the function will be the same for both i and j so it will increment twice

Problem 2

The dependency graph of the processes is as follows:

$$\begin{tikzcd}
 & \mathbf{P1} \arrow[dl] \arrow[dr] & \\
\mathbf{P2} \arrow[dr] & & \mathbf{P3} \arrow[dl] \\
 & \mathbf{P4} \arrow[d] & \\
 & \mathbf{P5} &
\end{tikzcd}$$

So we run the following code:

Initialize Semaphores	**Process 1**	**Process 2**

S1.initialize(0)
S2.initialize(0)
S3.initialize(0)
	// Process 1 code ...
	V(S1)
	V(S1)
		P(S1)
		// Process 2 code ...
		V(S2)

Process 3	**Process 4**	**Process 5**

P(S1)
// Process 3 code ...
V(S2)
	P(S2)
	P(S2)
	// Process 4 code ...
	V(S3)
		P(S3)
		// Process 5 code ...

Problem 3

The dependency graph of the processes is as follows:

$$\begin{tikzcd}
 & \mathbf{P1} \arrow[dl] \arrow[dr] & \\
\mathbf{P2} \arrow[d] \arrow[drr] & & \mathbf{P3} \arrow[dll] \arrow[d] \\
\mathbf{P4} & & \mathbf{P5}
\end{tikzcd}$$

Part a

Here we are allowed to only use binary semaphores. So we do the following:

Initialize Semaphores	**Process 1**	**Process 2**

`S1.initialize(0)`
`S2.initialize(0)`
`S3.initialize(0)`
	`// Process 1 code ...`
	`V(S1)`
		`P(S1)`
		`// Process 2 code ...`
		`V(S1) // For P3`
		`V(S2)`

Process 3	**Process 4**	**Process 5**

`P(S1)`
`// Process 3 code ...`
`V(S1) // For P2`
`V(S3)`
	`P(S2)`
	`P(S3)`
	`// Process 4 code ...`
	`V(S1)`
	`V(S3) // For P5`
		`P(S2)`
		`P(S3)`
		`// Process 5 code ...`
		`V(S1)`
		`V(S3) // For P4`

Part b

Here we are allowed to use general counting semaphores. So we do the following:

Initialize Semaphores	**Process 1**	**Process 2**

`S1.initialize(0)`
`S2.initialize(0)`
`S3.initialize(0)`
	`// Process 1 code ...`
	`V(S1)`
	`V(S1)`
		`P(S1)`
		`// Process 2 code ...`
		`V(S2)`
		`V(S2)`

Process 3	**Process 4**	**Process 5**

`P(S1)`
`// Process 3 code ...`
`V(S3)`
`V(S3)`
	`P(S2)`
	`P(S3)`
	`// Process 4 code ...`
		`P(S2)`
		`P(S3)`
		`// Process 5 code ...`

Problem 4

Part a

Why do dead locks occur in the dining philosophers problem? Because we have shared resources that must be used exclusively by 1 party/philosopher. More specifically if we have a situation like this:

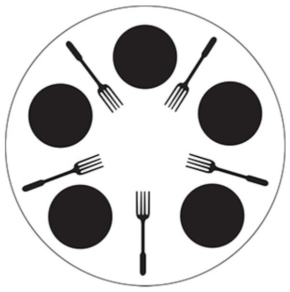

Figure 1: (I'm more partial to the chopsticks myself...)

Then if all the philosophers grab the left for at the same time, then need to grab the right fork, they will have to wait until one of the other ones drop it. If none of the others ever drop it, then they will wait forever and starve. This is the deadlock.

Part b

To fix this problem we will make the philosophers drop the forks after a certain amount of time. We use notation similar to the one used in lecture. Basically we use a simple solution: Give an ordering to the forks.

```
var fork: array[0..4] of semaphores=1
philosopher i
repeat
  first = min( i, i+1 mod 5 )
  next = max( i, i+1 mod 5 )
  wait( first )
  wait( next )
  ...
  eat
  ...
  post( first );
  post( next );
  ...
  think
  ...
forever
```

This prevents deadlock by not allowing everyone to pick up the left fork at the same time. Suppose this did not prevent deadlock, then there must be some fork, say i that is locked forever. Then the philosopher using that must be waiting for a fork, j, which must be great than i. Keep going along the chain and eventually you must, since this is deadlocked, get back to i, which means $i > i$, which is nonsense. So we must have that this prevents deadlock. This is not very efficient, and their are more efficient solutions but this is the simplest (in my mind) that prevents deadlock.