ESE 605 Homework 3

Stephen Phillips

February 18, 2015

Problem 1

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

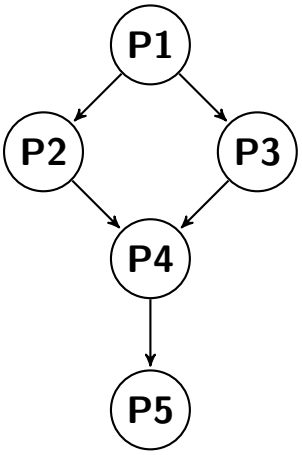
If the pseudo-code is as follows:

```
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 giving it a final value of $i + 2$. On pass by copy/restore it will not do the same. The reference to i will be aliased, so it will set the value of i to $i + 1$ twice, giving it the final value of $i + 1$. In other words, pass by copy/restore has aliasing issues.

Problem 2

The dependency graph of the processes is as follows:



So we run the following code:

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

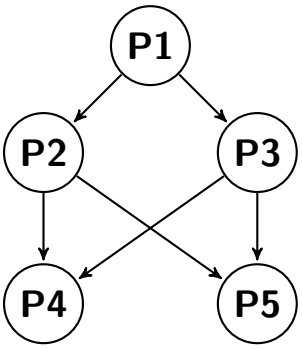
S1.initialize(0)
S2.initialize(0)
S3.initialize(0)
	// Process 1 code ...
	V(S1)
	V(S1)
		P(S1)
		// Process 2 code ...
		V(S2)

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:



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(S2)
	V(S3) // For P5
		P(S2)
		P(S3)
		// Process 5 code ...
		V(S2)
		V(S3) // For P4

This does not have any deadlocks. There are some race conditions, meaning the order in which P2 and P3 run or P4 and P5 run is dependent on which happens to get the lock first which can vary from run to run. This is fine though, since we did not want to enforce any order on P2/P3 or P4/P5, but they still must execute after P1 and P2/P3, respectively. The idea of the code is that first everything is locked (initialized to zero), and then P1 unlocks the first semaphore. P2 or P3 then get the lock, execute their code, relinquish the lock, then each of P2 and P3 relinquish locks they own. P4 and P5 need to grab both P2 and P3's locks before executing, after which they relinquish those locks so the other can run. As the order in which P2/P3 and P4/P5 get and release their locks is fixed, they cannot deadlock.

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 ...

The idea is the same as the previous, except now each process can have in some sense multiple locks using one semaphore. Therefore this code can run more concurrently than the last one since P2/P3 and P4/P5 do not have to go one after the other but can run at the same time. This still does not allow deadlock, as there are no competing resources now that we have counting semaphores. If the main resource is not time but number of semaphores, there is a way to use less semaphores:

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

S1.initialize(0)
S2.initialize(0)
S3.initialize(0)
	// Process 1 code ...
	V(S1)
	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(S1)
	P(S2)
	P(S2)
	// Process 4 code ...
	V(S1)
	V(S2) // For P5
	V(S2)
		P(S1)
		P(S2)
		P(S2)
		// Process 5 code ...
		V(S1)
		V(S2) // For P4
		V(S2)

Though this loses the concurrency benefit.

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:

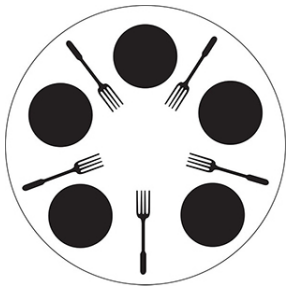


Figure 1: (I'm more partial to the chopsticks myself...)

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 there are more efficient solutions but this is the simplest (in my mind) that prevents deadlock. This is equivalent to something mentioned in class where you make one of the philosophers a 'righty' when the rest are 'leftys'.