Scribe Notes 2/28

CS390 Principles of Concurrency and Parallelism

Benjamin Sterrett

Peterson's Algorithm

As a quick reference, here is the algorithm as presented in the lecture slides:

```
public void lock() {
	flag[i] = true;
	victim = i;
	while (flag[j] && victim == i) {};
}
public void unlock() {
	flag[i] = false;
}
```

Peterson's algorithm is essentially a combination of the "lock one" and "lock two" designs which were discussed in the previous lecture. Peterson's algorithm works because it combines the aggressive check from lock one with the passive check from lock two. If only lock one is used, it is possible for deadlock to occur when both threads set their flag before either thread reaches the while loop. With both checks, even if the aforementioned situation arises, the thread that became the victim first would be allowed to continue. If only lock two is used, execution will halt when only one thread needs the lock. With Peterson's algorithm, a single thread can continue past the while loop.

A significant drawback to Peterson's algorithm as discussed in class is that it is only viable for up to two threads. To carry the ideas behind Peterson's algorithm on into a program with n threads, the bakery algorithm can be used.

Bakery Algorithm

As a quick reference, here is part of the algorithm as presented in the lecture slides:

```
class Bakery implements Lock {
	…
	public void lock() {
		flag[i] = true;
		label[i] = max(label[0], …,label[n-1])+1;
		while ( k flag[k]
			&& (label[i],i) > (label[k],k));
	}
}
```

As a quick refresher, the bakery algorithm works by assigning each incoming thread two things: a boolean flag, and a label consisting of the thread's ID and the next available number. The idea is that the threads will enter the locked section in order of the number they received. The basics are described quite in-depth in the class slides, so this document will instead focus on some concerns raised during class discussion.

One concern arose when first contemplating the prospect of preemption during the locking code section. Given threads A and B, consider the following scenario:

Thread A has a thread ID of 1, thread B has a thread ID of 2.

1) Thread B enters the lock first, sets flag[B] to true and enters the max function. It is then preempted by thread A just after max reports the value of 0, but before the value is stored into label[B].

2) Thread A enters the lock method and sets flag[A] to true. Thread A also enters the max function and since label[B] is not yet set, it retrieves the value 0, as well.

At this point, both threads will end up with a label[i] of 1. Because the label is a pair of (thread ID, number), the algorithm should be able to handle this situation by allowing the lower ID to continue first. But consider the following continuation to our above scenario:

3) Thread B runs and sets label[B] to 1. Thread B continues on to enter the while loop first, while thread A still has not set its label.

4*) Since thread B has number 1 and sees no other labels, it obtains the lock.

5) Thread A runs and sets label[A] to 1. Thread A continues on to enter the while loop.

6) Thread A sees thread B and the algorithm notices that label[A] == label[B] == 1. The algorithm then checks the thread IDs and sees that thread A has a lower ID, so it allows thread A to have the lock, as well.

As presented in the slides, this behavior is possible. However, the bakery algorithm as presented in its original paper (linked at the bottom of this document) includes an addition line of imperative code. In the bakery algorithm, entries in the flag array are not used to represent that a thread currently holds a lock. Instead, it is used to show that a thread is currently obtaining a label. The original pseudo-code from the paper is presented below:

```
begin integer j;
       L1 : choosing [i] : = 1 ;
       number[i] := 1 + maximum (number[l],..., number[N]);
       choosing[i] := 0;

       for j = 1 step l until N do
         begin
              L2: if choosing[j] != 0 then goto L2;
              L3: if number[j] != 0 and (number [j], j) < (number[i],
                     i) then goto L3;
         end;

       critical section;
       number[i] := O;
       noncritical section;

       goto L1 ;
end
```

Now, it is impossible for the above scenario to take place. Examining the choosing[] array, which replaces the flag[] array, prevents threads from entering the critical section while any other thread is attempting to obtain a label(in this code, a "number"). In our scenario, step 4) would not be able to occur if we had used the original bakery algorithm. Additionally, when the lock is returned, number[i] (or in our original code, label[i]) is set to 0, rather than our implementation which set flag[i] = false.

Another issue raised in class was the question of what would happen when the values for max rolled over under special circumstances. In the paper, a "Further Remarks" section is included which admits that there are problems with number[i] becoming large. In essence, the algorithm relies on "practical considerations," which would limit the value given by the maximum function. The bakery algorithm is not designed to accommodate a rollover of the maximum function.

A proof of correctness is also provided in the paper. There is not much reason to re-write the author's work here. The paper is not even two full pages and would be a valuable read for anyone in the class, especially anyone who still has doubts about the bakery algorithm. An even more interesting read are the comments on the paper by the author himself, Leslie Lamport, presented below the PDF link.

References

Lamport's Bakery algorithm:
http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#bakery