CMPSCI 377 Operating Systems **Fall 2008**

Lecture 7: February 9

Lecturer: Prashant Shenoy *Scribe: Vimal Mathew & Tim Wood*

7.1 Lottery Scheduling

Lottery Scheduling is a probabilistic scheduling algorithm for processes in an operating system. Processes are each assigned some number of lottery tickets, and the scheduler draws a random ticket to select the next process. The distribution of tickets need not be uniform; granting a process more tickets provides it a relative higher chance of selection. This technique can be used to approximate other scheduling algorithms, such as Shortest job next and Fair-share scheduling. Allocating more tickets to a process gives it a higher priority since it has a greater chance of being scheduled at each scheduling decision.

Lottery scheduling solves the problem of starvation. Giving each process at least one lottery ticket guarantees that it has a non-zero probability of being selected at each scheduling operation. On average, CPU time is proportional to the number of tickets given to each job. For approximating SJF, most tickets are assigned to short running jobs and fewer to longer runnning jobs. To avoid starvation, every job gets at least one ticket.

7.2 Synchronization

As we already know, threads must ensure consistency; otherwise, race conditions (non-deterministic results) might happen.

Now consider the "too much milk problem": two people share the same fridge and must guarantee that there's always milk, but not too much milk. If the two people do not coordinate, it is possible that both will go to buy milk at once and have too much milk. How can they solve this? First, we consider some important concepts and their definitions:

- **Synchronization**: the use of atomic operations to allow coordination between threads;
- **Mutual Exclusion**: ensuring that only one thread is performing a particular action or accessing a piece of data at a time;
- **Critical section**: a piece of code that only one thread can execute at a time;
- **Lock**: a mechanism for mutual exclusion; the program locks on entering a critical section, acesses the shared data, and then unlocks. Any other programs must wait if they try to enter a locked section.

For the above mentioned problem, we want to ensure some correctness properties. First, we want to guarantee that only one person buys milk when it is neeed (this is the *safety* property, aka "nothing bad happens"). Also, we want to ensure the *liveness* property – that someone *does* buy milk when needed. Now consider that we can use the following atomic operations when writing the code for the problem:

- "leave a note" (equivalent to a lock)

- "remove a note" (equivalent to a unlock)
- "don't buy milk if there's a note" (equivalent to a wait)

Our first try could be to use the following code on both threads:

```
if (no milk and no note)
   leave note
   buy milk
   remove note
```

Unfortunately, this doesn't work because both threads could simultaneously verify that there's no note and no milk, and then both would simultaneously leave a note, and buy more milk. The problem in this case is that we end up with too much milk (safety property not met).

Now consider our solution #2:

Thread A:

```
leave note "A"
if (no note "B")
   if (no milk)
      buy milk
remove note "A"
```

Thread B:

```
leave note "B"
if (no note "A")
   if (no milk)
      buy milk
remove note "B"
```

The problem now is that if both threads leave notes at the same time, neither will ever do anything. Then, we end up with no milk at all, which means that the progress property not met. Solution #3 will be discussed on the next class.

7.3 Concurrency

When programming with threads, processes or with any type of program that has to deal with shared data, we have to take into account all possible interleaving of these processes. In other words, in order to guarantee that concurrent processes are *correct*, we have to somehow guarantee that they generate the correct solution no matter how they are interleaved.

From the "Too Much Milk" problem it is clear that it can be very difficult to come up with an approach that always solves it properly. Let us now consider an approach that *does* work:

Thread A

```
leave note A
while (note B)
   do nothing
if (no milk)
   buy milk
remove note A
```

Thread B

```
leave note B
if (no note A)
   if (no milk)
      buy milk
remove note B
```

This approach, unlike the two examples considered on the previous class, does work. However, it is not easy to be convinced that these two algorithms, when taken together, always produce the desired behavior. Moreover, these pieces of code have some drawbacks: first, notice that Thread A goes into a loop waiting for B to release its note. This is called "busy waiting", and is generally not a good idea because Thread A wastes a lot of CPU, and because it can't execute anything usefull while B is not done. Also, notice that even though both threads try to perform the exact same thing, they do it in very different ways. This is a problem specially when we were to write, say, a third thread. This third thread would probably look very different than both A and B, and this type of assymmetric code does not scale very well. So the question is: how can we guarantee correctness and at the same time avoid all these drawbacks? The answer is that we can augment the programming language with high-level constructs capable of solving synchronization problems. Currently, the best known constructs used in order to deal with concurrency problems are *locks, semaphores, monitors.*

7.3.1 Locks/Mutex

Locks (also known as Mutexes) provide mutual exclusion to shared data inside a critical session. They are implemented by means of two atomic routines: *acquire*, which waits for a lock, and takes it when possible; and *release*, which unlocks the lock and wakes up the waiters. The rules for using locks/mutex are the following:

1. only one person can have the lock at a time;
2. locks must be acquired before accessing shared data;
3. locks must be released after use;
4. locks are initially released.

Let us now try to rewrite the "Too Much Milk" problem in a cleaner and more symmetric way, using locks. In order to do so, the code for Thread A (and also for Thread B) has to be the following:

```
lock.acquire()
if (no milk)
   buy milk
lock.release()
```

This is clearly much easier to understand than the previous solutions; also, it is more scalable, since all threads are implemented in the exact same way.