CMPSCI 377 Operating Systems **Fall 2008**

Lecture 8: September 25

Lecturer: Prashant Shenoy *Scribe: Shashi Singh*

8.1 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. How to solve it? First, we consider some important concepts and their definitions:

- **Mutex**: prevents things from operating on the same data at the same 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. Also, a program waits if it tries to enter a locked section.
- **Invariant**: something that must always be true when not holding the lock.

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 that someone *does* buy milk when needed (the *progress* property, aka "something good eventually happens"). 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.

8.2 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.

Earlier we discussed the "Too Much Milk" problem, and realized that it's 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*.

8.2.1 Locks/Mutex

Locks (also known as Mutex) 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;
2. locks must be acquired before accessing shared data;
3. locks must use release after use;
4. locks are initially released.

The syntax for using locks in C/C++ is the following:

```
pthread_mutex_init(&l);
...
pthread_mutex_lock(&l);
      update data    // this is the critical section
pthread_mutex_unlock(&l)
```

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:

```
pthread_mutex_lock(&l)
if (no milk)
   buy milk
pthread_mutex_unlock(&l)
```

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.

Now, how could one go about implementing locks? No matter how we choose to implement them, we *must* have some hardware support. One possibility to implement locks is to disable interrupts, since these are the only way that a CPU has to change what it is doing; in other words, if we keep the CPU from switching processes, we can guarantee that only one process (the active one) will have access to the shared data. Another option would be to make use of atomic operations, such as test&set. This operation (which usually

correspond to an assembly instruction), is such that test&set(x) returns 1, if x=1; otherwise, if x=0, it returns 0 and sets x to 1. All this is of course implemented atomically. Having this type of atomic operation, one could implement *thread_lock*(*l*) simply as

```
while test&set(l) do nothing;
```

and *thread_unlock*(*l*) simply as

```
l = 0;
```

Summary:

- Communication between threads is done implicitly, via shared variables;
- Critical sections are regions of code that access shared variables;
- Critical sections must be protected by synchronization;
  - We need primitives that ensure *mutual exclusion*;
  - Writing "personalized" solutions to concurrency is tricky and error-prone;
  - The solution is to introduce general high-level constructs into the language, such as *pthread_lock*() and *pthread_unlock*().