Lecture 13
Concurrent Programming

18th December 2002

Exam Return

Some excellent exams, too many not so good.

I am disappointed that so many have not mastered the basic definitions

- non-interference
- assignment axiom of PL
- how to write a formula for an invariant
- looking at code and seeing that it makes *no sense:* ...P(a); V(a)...

If you don't understand this stuff, why are you not beseiging me in my office? Learning is not a process whereby I pour knowledge into you. *You* must *construct* your own understanding. If the construction is not working, I get a lot less opportunity to see that than you do, so *please* come by during office hours with questions if you are not understanding.

Project due date

I may have caused confusion last time: I was thinking the project was due this Thursday and may have said something to that effect. I apologize for the confusion. We can proceed in several ways

- leave the due date where it is
- extend it to Thursday
- allow individuals to make a case to me that it should be extended on a case by case basis

Thread/Process implementation

A note on terminology:

- *process* is typically (but not always) used in referring to execution contexts managed by the OS kernel. Sometimes you will hear *heavy-weight process* used to refer to an execution context together with an address space.
- *thread* is typically (but not always) used in referring to execution contexts managed in user space. Sometimes you will hear the term *light-weight processes* used to refer to execution contexts that share an address space. The image of *thread* is of a very thin string (but we don't refer to heavy-weight processes as strings so the imagery isn't perfect).

The book in Ch. 6 gives a description of typical approaches to implementing threads/processes in either OS-kernel or user space. Confusingly, a user-space implementation might be called a *threads-kernel*. A OS-kernel space implementation has to deal with addressability issues that don't arise in user space implementations, but otherwise the approach is the same.

Each time a thread (or process) performs a thread operation (fork/join, interrupt, time-slice expiration) or a synchronization operation (monitor entry/exit, cond wait, cond notify, semaphore P/V) three main things have to happen:

- the fate of the current thread must be decided and carried out – it might be placed on a ready queue, or it might go onto the wait queue of a synchronization object (monitor, condition, semaphore). Procedures running on the current thread's call stack usually carry out these operations.
- the next thread to be run must be chosen. In the moment of the decision, this is the role of the *dispatcher*. In making its decision it will be guided by the data structures maintained by the *scheduler.* Again, procedures running on the current thread's call stack usually carry out this decision.
- Finally, the processor must be switched to running the next thread, leaving the current thread in a state where it can be resumed. This involves saving all of the thread state held in processor registers into memory (typically onto the thread stack and into the thread control block) then loading the processor register for the next thread.