Lecture 12
Concurrent Programming

10th October 2002

Program Handback

Designs are settling down. Many people have got something working. A general concern is the use of sleep() to do the signal timings, especially sleep(constant). While a thread is sleeping it cannot react to other inputs, so sleep(constant) is not going to work for handling sensor input while a signal is green. In the absence of the train part of the problem you would get away with it if you calculated the next wakeup time each time, but you will not be able to respond instantly to the train with this strategy. You will have to use wait().

Structures observed repeatedly:

1. Input thread
   controller thread (finite state machine)
2. Input thread
   per signal thread, with finite-state machine control distributed amongst the signals. The currently green signal determines the next state.
3. Input thread
   controller thread
   per signal thread

Also, at least one person is using a structure involving

- Input thread
  per signal thread with states determined using dining-philosophers-style competition for locks: concern - won't be able to easily implement predictable cycle EWR, N, S, EWL.

Some people are done or almost done. Some don't really have working code yet.

Make sure that your code will accept input on standard input and produce output on standard output in the specified formats.

Indent your code: the body of every {} pair should be indented (unless it is all on one line).

Code printing: the easiest-to-read printout of code was obtained with

```
enscript -1 -j -G -fCourier9 -Ejava -MLetter -H -T3 -C -r -ooutput.ps *.java
```

If you have your files on a Linux/Unix machine see if you can use this command to prepare your printed output. It is very helpful to me in reading the code.

Common mistakes in thread programming to watch out for

Reference - Choi, Sung-Eun and Lewis, E. Christopher. *A Study of Common Pitfalls in Simple Multi-threaded Programs,* ACM SIGCSE, March 2000, Austin, TX, pp. 325-329

Thread mistakes

- not having a lock for shared data
- not *using* a lock around access to shared data
- prematurely releasing a lock
- accidentally making data shared (using global variable where a local one was intended)
- using two different locks at different times around shared data
- deadlocks – locking order – establish a lock order and stick to it
- deadlock – failure to notify a CV

Another viewpoint on threads

A quite famous talk by John Ousterhout, creator of TCL and TK, at the 1995 Usenix symposium "Why threads are a bad idea, most of the time". There is no paper for this talk (it was an invited presentation), but the slides are available and make the author's point quite well.

I will talk through Ousterhout's slides, to set up a discussion on event-driven programming as an alternative approach to programming with threads. So you

know my biases, I think Ousterhout overstates the case against thread programming and understates the difficulty of solving some problems using event-driven programs. Although we disagree about where the lines should be drawn, we agree that the lines exist.

URL for Ousterhout's slides: http://home.pacbell.net/ouster/threads.ppt

Event driven programs

Two "styles":

- event loop in the client program (MS Windows, Mac?),
- event loop in framework (X11, Tcl/Tk, Java).

The event loop is in the client program typically looks like

```
main () {
   while true {
      evt = getEvent();
      if (evt.type==c1) {...
      } else if {evt.type==c2) {...
      } else if {evt.type==c3) {...
      }
   }
}
```

I think it is really hard to program in this style in a way that is clear to subsequent readers. The while loop necessarily gets very large when lots of event types have to be processed, and there is a great temptation to reproduce code fragments in different arms of the event demultiplexer.

The type 1 solutions (above) to our traffic signal problem typically have a strong flavor of this style).

The other common approach puts the event loop and dispatcher in the framework. In this style, the client program registers handlers for different event types and the loop in the framework calls them as appropriate. This code will look something like:

```
int ev1proc(void* data, args for ev1) {
}
int ev2proc(void* data, args for ev2) {
}
...
```

```
main () {
   fwk_register_ev1_handler(ev1proc, ev1data);
   fwk_register_ev2_handler(ev2proc, ev2data);
   ...
   fwk_mainloop();
}
```

In a framework like java's swing registration takes on a whole collection of rules concerning objects as fields of other objects with fairly simple though very numerous rules concerning what gets called, when and by whom. My understanding is that the java swing framework is mainly single threaded but I'm not an expert.

Issues: lack of isolation – if one event handler takes a long time no other events can be processed. Early versions of Windows suffered tremendously from this sort of problem. Newer versions isolate more activities in their own threads or processes so this is less of a problem at the system level (fewer system hangs), but at the application level it can still be a problem: example dialog box requiring use of the application that generated it to answer its question. These days, however, you can never be sure whether this is an event handling issue or a shared data structure locking issue.

For next time

Read Chapter 6

Read "Using Threads in Interactive Systems: A Case Study". http://doi.acm.org/10.1145/168619.168627

We will talk about what the code looks like for some of the patterns described there.

Plan – patterns for using threads (the Using threads.. paper), thread implementation issues (Chapter 6 + Using threads...)

We will then move on to concurrent programming with message passing.