Lecture 21
Concurrent Programming

12th November 2002

Asynchronous concurrent programming

I have been pretty negative so far in this class about approaches to concurrency that rely on asynchronous interruption of a process to manage communication. I'm not talking here about asynchronous communication systems that allow senders to proceed without synchronizing with a receiver; rather, I mean the style of programming in which events in the environment (completion of an I/O device operation for example) cause an executing process to stop what it is doing, begin execution in another location, and eventually return to its original task (an interrupt).

My negativity about the asynchronous style stems from the seemingly inherent complexity caused by the style: potentially at any point a program can be interrupted: data structures can be in any state when the interrupt handler is entered, and can be in a different state when it is exited. On uniprocessors, interrupt masking is used to protect critical sections, much as locks protect critical sections in threaded code. As usual, on multiprocessors masking is not sufficient.

As an example of an asynchronous concurrent programming system we will look at the POSIX asynchronous I/O extension. I choose it not because I think it is a good way of doing I/O but merely as standards-based illustration of the asynchronous style. The style sees its greatest use inside operating systems where the primitives are similar but specific to each system)

(My source for this material is *POSIX.4: Programming for the Real World* by Bill O. Gallmeister, O'Reilly and Associates, 1995.)

Functions and data structures

```
int aio_read(struct aiocb *racbp);
int aio_write(struct aiocb *wacbp);
int lio_listio(int wait_or_not,
```

```
                struct aiocb * const lacb[],
                int num_acbs,
                struct sigevent *notification);
int aio_cancel -- omitted;
int aio_suspend(struct aiocb * const lacb[],
                int num_acbs,
                const struct timespec *timeout);
ssize_t aio_return(const struct aiocb *acbp);
int aio_error(const struct aiocb *acbp);

struct aiocb {
   int aio_fildes;
   off_t aio_offset;
   volatile void *aio_buf;
   size_t aio_nbytes;
   struct sigevent aio_sigevent;
   int aio_lio_opcode;
   int aio_reqprio/
}
```

Use of signals

```
struct aiocb a;
a.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
a.aio_sigevent.segev_signo = SIGname;
a.aio_sigevent.sigev_value.sival_ptr = (void *) &a

/* setting up signal handling */
sigset_t completion_signals;
sigemptyset(&completion_signals);
sigaddset(&completion_signals, SIGname);
struct sigaction sa;
sa.sa_flags = SA_SIGACTION;
sigemptyset(&sa.sa_mask);
sa.sa_sigaction = aio_done;
sigaction(SIGname, &sa, NULL); /* 3rd arg gets old action*/

/* blocking signals for critical section */
sigprocmask(SIG_BLOCK, &completion_signals, &prevmask);
... do critical section ...
sigprocmask(SIG_SETMASK, &prevmask, NULL);

/* AIO signal handler */
void aio_done(int signo, siginfo_t *info,
              void *ignored) {
   if ((signo != SIGname) ||
```

```
        (info->si_code != SI_ASYNCIO)) {
      /* something's wrong */
   }
   if (aio_error (...acb... )!= EINPROGRESS){
      if (aio_return(...acb...) != acb->aio_nbytes) {
         /* something's wrong */
      } else {
         /* free buffers, etc. */
         return; /* exit handler */
      }
      return; /* exit handler with IO still in progress -- shouldn't happen */
   }
}
```

Polling as replacement for interrupts under high load

Interrupts are fairly expensive, requiring saving and restoring processor state (i.e. the cost of an interrupt is roughly equivalent to that of a thread switch). In situations where the i/o load is very high (for example, a processor that decompresses video streams as its sole task), it may be preferable to move to polled i/o. If multiple i/o requests are outstanding and a new one is complete each time the processor finishes with the data from the previous i/o request, you can squeeze a few percent more performance out of the processor by turning off interrupts as long as this continues:

```
interrupt handler:
   /* interrupts for this device are disabled here */
   while there is another completed i/o {
      issue new i/o
      process data from this i/o
   }
   return from interrupt /* and enable interrupts */
```

About patterns

Patterns are re-usable approaches to solving problems. They are not necessarily embodied as code libraries. They may exist in the form of narrative description only. Patterns represent collected wisdom about how to solve a certain kind of problem in certain contexts.

Patterns usually begin with an *example* that help sets the *context* and expresses the *problem* leading to a general description of the *solution.* A pattern may provide detailed information on how to *implement* the solution. Pattern descriptions should include information about *variants, known uses,* and *consequences.* Good pattern collections will provide cross-references between the patterns they

describe and other published patterns (e.g. comparing how well different patterns solve similar problems).

Half-async/Half-sync pattern

This example is from *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects,* by Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschman, Wiley and Sons, 2000.

Given the ugliness of programming with asynchronous mechanisms (especially as they appear in POSIX.4 or in the typical OS kernel), an important pattern is one that maps the inherent asynchrony of I/O devices onto synchronous mechanisms. Interestingly enough, this mapping is key to the classical Unix view of I/O – synchronous reads and writes by multiple processes on asynchronous devices. The POSIX.4 mechanisms involve *undoing* the async->sync mapping – which perhaps explains some of their complexity.

Context: a concurrent system that contains both asynchronous and synchronous processing services that must communicate with each other.

Problem: at some point in a system it becomes necessary to face the fact that I/O is asynchronous, and needs to be so in order to allow processing and I/O to proceed simultaneously. Yet, for application developers, synchronous processing is simpler.

Solution approach: decompose the services into an asynchronous layer and a synchronous layer, with a queueing layer between them. In the synchronous layer, long-lasting application processes use multiple threads and synchronous operations to manage concurrency. In the asynchronous layer, short-lived protocol handlers driven by interrupts asynchronously manage short-term state associated with individual I/O operations. The queueing layer provides the communication between the synchronous and asynchronous layers, including any required synchronization.

Known uses: Unix I/O subsystem; CORBA ORBs; Restaurant – host or hostess handles asynchronous interrupts from arriving customers and table cleaners, maintains queues of waiting customers, and hands them over to a more synchronous process associated with order taking, meal preparation, payment, etc.