Concurrency

Many applications require concurrency, the ability to handle multiple outstanding tasks with the illusion or reality of simultaneity. Even something as simple as a telnet or ssh client has at least two tasks, responding to the user and responding to the network. A modern word processor can have many tasks: responding to the windowing system, background repagination, background spell and grammar checking, background printing, and running scripts. A high performance web server may have to support hundreds or thousands of connections at a single time.

There are many mechanisms that a developer can use to implement concurrency. This is a brief introduction to several of them and the tradeoffs between them. Your book covers processes and threads in considerable depth also.

Blocking, Starvation, and "Fairness"

A task may block on file system I/O, network I/O, the graphical user interface, waiting for another task, or for many other reasons. A primary goal of a concurrency mechanism is to insure that if one task blocks, another task can run. Another important goal is to allow each task to progress when it is not blocked. Eventually, CPU time must be given to each task that is not blocked. Finally, CPU time should be "fairly" divided among the tasks that are not blocked. What "fair" means depends on the application. For example, in a streaming video player application, it is probably the case that a task that draws frames takes priority over the task that handles the GUI.

It is important to note that the goals of concurrency within an application are the same as those within an operating system.

Communication Between Tasks

Tasks within an application are rarely independent of each other, and thus each concurrency mechanism is typically combined with one or more communication mechanisms. Communication introduces a new way in which task progress can be halted: deadlock. A simple example of deadlock is two tasks that want to exchange messages. Each task may either send its message first or receive its partner's message first. If each task receives first, both will necessarily block forever. If one receives first while the other sends, progress is guaranteed. If each sends first, then the occurrence of deadlock is determined by the details of the implementation and the execution of program up to that point, an example of the nastiest kind of bug, one that only shows up sometimes, a Heisenbug.

Threads

On the face of it, mapping tasks to threads seems to be an ideal solution to concurrency within an application. Blocking, starvation, and fairness are handled by the thread scheduler, and communication is as simple as reading and writing a shared address space. Furthermore, most operating systems provide support for threads at the kernel level, and

threads can also be implemented entirely at the user level through careful use of signals. The details of how threads are implemented and how to program them using pthreads are available in your book.

While mapping application tasks to threads is highly effective, it also has some problems. First, the overhead of starting a new thread is non-trivial. This is not an issue for applications where the tasks are all known a priori or change slowly, but can lead to performance issues in an application that has a stream of arriving tasks. A common way to mitigate this is the thread pool model. Here, the application creates a set of worker threads when it starts up that is as large as the largest number of tasks it expects will be outstanding at any one time. When a task arrives, it is assigned to a free worker thread which runs it to completion and then comes back for the next task.

Another performance issue that arises with threads is the overhead of context switches, especially context switches in kernel level thread implementations. Not only is this overhead much higher than a procedure call (remember, threads are all within a single process), but it is incurred at unexpected times because it is the thread scheduler that decides when to schedule a different thread, not your application.

Thread-based concurrency has proven to be remarkably difficult for programmers to get right when tasks need to communicate in all but the simplest ways. The shared address space makes communication look simple, but, because a context switch can occur at any time, access to shared data must be carefully controlled using synchronization mechanisms such as locks or semaphores. A common approach is to acquire locks or down semaphores in only one order. Because it is possible to get at the shared data whether or not the synchronization is correct or even by ignoring synchronization altogether, it is very easy to introduce Heisenbugs, especially as the code evolves. Even when deadlock is not possible, incorrect synchronization can lead to the corruption of data as it is exchanged between tasks.

Processes

An application can also implement concurrency by mapping each of its tasks to a process and then using interprocess communication (IPC) mechanisms (pipes, fifos, message queues, Unix domain sockets, TCP sockets, shared memory…) to communicate between them. Because each process has a separate address space, communication is usually done explicitly via messages. Generally, it is guaranteed that messages will not be corrupted. However, the developer must still be concerned about deadlock. To avoid it, it is necessary to carefully establish the order in which each process sends and receives messages with respect to all the other processes. It is possible for processes to share parts of their address spaces, in which case difficult synchronization issues similar to those in threads come to into play.

Process creation and process context switch overheads are considerably higher than those of threads, which tends to make process-based concurrency more appropriate for larger tasks. Just as with threads, process creation costs can be mitigated in some cases by maintaining a pool of worker processes.

Processes that communicate using explicit messages tend to be easier to migrate to take advantage of new hardware. For example, on some operating systems, processes can be spread over the CPUs of a multiprocessor machine while threads cannot. An application that consists of processes that communicate using messages can generally easily be adapted to exploit separate machines in a cluster. For example, a web search engine might consist of a front-end process that passes requests to a process pool. As the engine gets more popular, the process pool can both be made larger, and spread over a growing number of machines.

Signal-driven I/O

Some applications have little computation, but need to handle many outstanding I/O connections simultaneously with low overhead. Whenever a file descriptor is ready for reading or writing, they want to respond to it immediately. This maps naturally into a single process using signal-driven I/O. At startup, the application installs a signal handler for SIGIO. Then, for each file descriptor fd, it calls fnctl(fd,F_SETFL,O_NONBLOCK) to set up non-blocking I/O and then calls fcntl(fd,F_SETSIG,0) to indicate that a SIGIO signal should be sent whenever fd can be read or written. The program then executes normally, with abrupt jumps into the signal handler whenever I/O is possible.

Signal-driven I/O can be very efficient because there are no context switches at all, and kernel level exceptions map rather straightforwardly to the application level.

Notice that the signal handler and the normal execution of the program look very much like two separate threads. If they share data, then access must be very carefully controlled for the sake of correctness. However, unlike threads, the normal execution of the program cannot preempt the execution of the signal handler. Multiple signal handlers must similarly synchronize their access to shared data. Also, note that the signal handler must be prepared to handle partial I/Os. If the goal is to read 100 bytes from a file descriptor, and the handler is invoked and can read only 50 before getting an EAGAIN error, it must save state so that it can read further the next time the signal is raised before it returns.

Select-based Programming

In many cases, the primary motivator for concurrency is the need to wait for any one, or several, of a set of events to occur without knowing the order in which they will occur. For example, we may want to read from whichever file descriptor has data available next. If we simply read them in order, we'd block at the first one that had no data, even though there might be others that are ready to be read. Of course, we could set each file descriptor for non-blocking I/O and then repeatedly poll each one. However, if it is often the case that data isn't available on any file descriptor, this could easily burn up lots of CPU cycles. We could use signal-driven I/O, but then we would have to deal with synchronization between the signal handlers and the normal execution of our program.

Instead, Unix (and most other operating systems) provides a system call that will block until one or more of a set of events happens. In Unix, the system call is called "select" and it lets us simultaneously wait on file descriptors becoming ready for reading, writing or having an exceptional condition occur, on a signal arriving (and being processed), and for a specified interval of time. A select-based program is usually based on a loop whose body sets up the events that the program is interested in, a call to select which blocks until one or several of the events are events, and then loop which processes all the events that have happened. In essence, it is a state machine where the processing of the events moves the state forward.

Select-based programs can be quite efficient, although not as efficient as signal-driven I/O programs. They tend to be reasonably easy to think about and debug because they are purely sequential programs in which preemption does not occur. Applications for windowing systems such as X and MS Windows are usually based on a select-based model, although this is often combined with other concurrency schemes.

An important caveat with select-based programs is similar to that of signal-driven I/O programs: it is the programmer's responsibility to ensure that the program makes progress. If handling an event takes too long or results in a system call that could block, the programmer must save the state of the handler and return to it later. Quite simply, select must be called often in order for the program to progress. Of course, the programmer has total control over when and how often this happens.

Combinations

The different concurrency (and communication) mechanisms can be combined, generally without restrictions. However, the semantics often become subtle and can lead to bugs. Especially bug-prone is the combination of signals and threads.