Concurrent Programming
Lecture 25

4th December 2003

Exceptions

What if something goes wrong in a computation?

Return-code checking on each operation: tiresome, error-prone.

Exceptions and exception handling introduced into programming languages to make it easier to handle erroneous/unusual situations in a more structured fashion. (Java, C++, Modula-3, ...)

Interaction of exceptions and synchronization

Review the semantics of exceptions and synchronization locks.

The basic situation in Java is that the synchronization lock is released when the synchronized block is exited. So the following will not leave a stray lock held:

```
synchronized m(...) throws E {
   ... some code that might raise exception E ...
}
```

But depending on the state of the object's data when the exception is raised, the above may well leave the object in an inconsistent state (monitor invariant does not hold). For example:

```
{invariant: i>=0}
synchronized m(...) throws E {
   i = -1;
   ... do some work that might raise E ...
   i = 1;
}
```

So, it is much more common to explicitly handle exceptions within synchronized blocks than it is to just let them propagate. The question is how should a program be structured to restore the monitor invariant? There are several possibilities:

- abruptly terminate - make sure that the program as a whole terminates; this is surprisingly difficult to get right because often you want the program to terminate cleanly with respect to its effect on the external environment – files, network connections, etc. That means that its data structures must be made sufficiently consistent to allow clean-up actions to occur.
- continue - both the examples above are examples of the *continue* appropach. The program does the same thing to restore the invariant whether or not the exception was raised. You can recognize this approach by the use of *finally* clauses. If you've programmed in C++ you may recognize this approach as being that which is embodied in the "destructors for local variables are called at block exit regardless of its cause" rule. This approach is applicable only when the interactions between the code and invariants are quite simple.

If *do some work that might raise E* is has a complicated effect on the state of the object the *continue* strategy is unlikely to be sufficient and techniques such as we discussed last time for transactions become relevant.

- roll-back: restore the state as it was at the beginning of execution of the method. Two possibilities: save a copy of the state so as to be able to restore it, or work in a new copy of the state. If the state is very large, it is worth considering a *copy-on-write* approach to saving parts of the state that are modified.
- roll-forward or recovery: these techniques inspect the state as of the time of the exception and push it ahead to a consistent state. Examples include closing files that were opened, taking compensating actions with the environment, etc. Note that roll-back and roll-forward may have to be combined in some instances where a point-of-no-return is reached in executing the method body.
- retry: some exceptions occur due to temporary conditions in the system or the environment. In such cases it may be worth retrying an operation that raised an exception as part of handling the exception.

Notice that these approaches to exception handling are not so different from approaches used in sequential programs, but that we have to consider the effect of exception handling on other threads as well as the current thread. In my opinion this consideration makes the use of abrupt termination in particular less viable for concurrent programs. In a sequential program exception handling up the stack will eventually have an opportunity to clean up everything that the program is doing. In a concurrent program this is not the case.

Exception handling design note: the Java exception facility was inspired by that of Modula-3 which was a simplification and improvement on the the exception facility

of the Xerox Mesa language. These later languages use a so-called termination model for exceptions: once an exception is raised from a particular point of execution, the stack will be unwound to that point. The Mesa language offered so-called continuation semantics for exceptions: after an exception handler processed the exception it had the ability to indicate that the source statement raising the exception could be retried, or that execution could continue immediately after the statement causing the exception. Consider that the handler could be many procedure calls away from and in an entirely separate module from the point where the exception was raised. RETRY and CONTINUE introduce a strong dependence between the handler and the point of the exception which is not desirable from the software engineering perspective of providing independence and isolation. Providing support for these possibilities also contributed a great deal of complexity to the implementation of exception handling.

Notice the difference between the Mesa RETRY and the retry strategy above. Mesa RETRY allows an exception handler anywhere on the stack to direct that the statement that caused the exception (at the very hot end of the stack) should be executed again (notice the implication that the stack is still intact when the handler is running). The retry strategy, on the other hand, says that as part of the handler we will re-attempt the operation at the handler's level of the stack. There is no implication that the stack beyond the handler is still intact.

Another point worth consideration: automatic storage management simplifies exception handling; in C++ automatic destructor invocation is helpful, but cannot deal with all the situations that arise when storage is explicitly allocated.

Synchronous and asynchronous exceptions

Synchronous exceptions are ones raised as a result of action by the thread, either an explicit throw or an run-time exception due to a mistake in the code. Synchronous exceptions are in some sense predictable and this is reflected in Java's rule that a method that calls another method declared to throw an exception must either handle that exception or itself be declared as throwing it.

Asynchronous exceptions are ones that may be raised at arbitrary points in the execution of a thread. Asynchronous exceptions are about as difficult to deal with as signals() in a Unix process (though the handling mechanism is a lot cleaner in my opinion).

Early versions of Java promoted use of an asynchronous exception Thread.ThreadDeath caused by invoking Thread.stop(). Its use is now deprecated (as in "don't use it"). Thread.interrupt() and Thread.InterruptedException are considerably cleaner: InterruptedException is thrown only by methods that wait on conditions (or explicitly by user code). Therefore, places where it needs to be caught are predictable, and the action to be taken in the handler can be determined by the surrounding context of the handler.

Exceptions and constructed synchronization mechanisms

Although locks are released automatically for synchronized methods and statements the same is not true of constructed synchronization mechanisms such as semaphores that you might build in Java. The problem, of course, is that the point of such objects is that they can be used in an unstructured way – so the compiler has no hope of automatically generating code to automatically release the lock.

```
sem.P();
... code that might raise E ...
sem.V();
```

Thus, yet another complication in using such mechanisms is that you as the programmer have to worry, at each point in your code, about which locks might be affected by the arrival of which exceptions. If the use is structured you could write

```
sem.P();
try {
   ... code that might raise E ...
}
finally { sem.V; }
```

Indeed this is essentially how lock releases are implemented by the compiler for synchronized methods and statements. But consider now the difficulty if you are using split binary semaphores, or a more complicated constructed synchronization mechanism such as reader/writer locks.

The best advice I can offer about this is: use structured mechanisms (synchronized in Java) whenever possible. If that is not possible, use constructed mechanisms in a structured way. If neither of those apply, expect to spend extra time with your design to work out how exceptions are to be dealt with. In the first case remember that datastructure invariants must hold when outside of code protected by synchronized, so the default behavior of releasing locks and allowing the exception to propagate is often not likely to be what you want.