Synchronous Message Passing Kernel
Concurrent Programming Project 2

March 17, 2008

General Description

In this project you will implement synchronous message passing using shared-memory concurrent programming. Synchronous communication was discussed in class on March 6. The details for this project are found in the interface specification below.

Use Java for your implementation. If you want to use some other language please see me first (I encourage you to explore other systems, but want to make sure we both know what we're getting into.) Your life will be easier if you make good use of available libraries for manipulating linked lists.

Important Dates

This problem is harder than it looks at first. Therefore, I've broken it down into 3 stages. The first two stages are actually fairly easy, but provide a good foundation for completing the third stage.

March 23: a working implementation of synchronous send and receive for channels (stage 1) is due. See the interface specification and implementation notes below.

March 30: a working implementation of synchronous event-based communication (stage 2) is due.

April 6: the full implementation including select() is due.

Interface Specification

The interface signatures below follow Java syntax, but the intent should be clear regardless of what language you use: you are to provide data structures and operations corresponding to those described here. You may need to provide additional operations or to add fields to the objects in order to implement the required semantics.

For the first stage implementation:

```
class Channel {
   void Send(Object o);
   Object Recv();
}
```

Interfaces for the second stage implementation. Notice that the Send and Recv operations of Channel are no longer required to achieve communication. If you implement them, implement them in terms of SendEvent and RecvEvent.

```
class CommEvent {
   void sync();
}
class SendEvent extends CommEvent {
   SendEvent(Object o, Channel c);
}
class RecvEvent extends CommEvent {
   RecvEvent(Channel c);
   Object GetValue();
}
```

For the final implementation, all of the classes of the second stage plus:

```
class SelectionList {
   void addEvent(CommEvent ce);
   CommEvent select();
}
```

In stages 2 and 3, an event that is used in a sync or select() operation may be used repeatedly. (Each time a SendEvent is sync'd or select'd it sends the same value.) You may (should) assume that an event will not be used twice at the same time: that is it won't appear in two SelectionLists passed simultaneously to select(), nor be passed simultaneously to two sync()s, nor be passed to sync and appear in a SelectionList passed to select() simultaneously. An easy way to make sure that you observe these restrictions in your testing code is to use each CommEvent and SelectionList only in the thread in which it is created.

Implementation notes

Java's intrinsic monitors are ill-suited to the synchronization requirements for this project. You will want to be able to wake up a particular process rather than all waiting processes (which you're pretty much forced to with Java's single-lock, single-CV per object model).

The stages of the project are intended to allow you to develop something fairly easy first, get it working, and gradually enhance it to meet the full specifications. I suggest a more aggressive schedule than the due dates require as the third part is by far the hardest. Here are notes on each of the stages.

Stage 1: Channels with Send and Receive

Synchronous communication on channels is easy to implement: each channel has a send queue and a receive queue. At least one of the queues is always empty (*invariant*, why?). Channels have two operations, send and receive. The *send* operation works as follows. If the recv queue is non-empty a send transfers its object to the first recv from the queue, moving the sent data to the receiver and releasing the queued receving process. On the other hand, if the recv queue is empty, make an entry in the send queue and wait to be woken by a receiver. The *recv* operation is symmetrical to send: if the send queue is non-empty, a receive takes the value from the first send in the send queue, moving the data from the sender to the receiver and wakening the sender.

Mutual exclusion: since only a single channel is involved in any communication, adequate mutual exclusion can be obtained by locking the channel during each operation.

To pass data from the sender to the receiver I suggest using a field in the data structures that are used to represent waiting senders or receivers. Notice that the interface signature says that the value passed is a Java Object.

Stage 2: Events with SendEvt, RecvEvt, sync

This stage introduces the notion of communication *events*, which come in two forms, SendEvt and RecvEvt. An event encapsulates the notion of the *potential to perform a communication action*, without actually performing it. Let me repeat, when an event is created *no communication is performed*. In order to perform its communication, a *sync* operation must be performed on the event.

A SendEvt consists, abstractly, of a value and a channel. A RecvEvt consists of a channel and an operation for extracting the received value. The received value is available after the receive event's sync operation returns.

I suggest the following organization: each event object should have a *poll* method, and a *enqueue* method, in addition to its *sync* method. *sync* first calls *poll* which, if the operation can be performed (the opposite queue is non-empty) performs it and wakes up the partner thread. Otherwise *sync* calls *enqueue* which adds the event to the correct queue of the channel. After that *sync* waits – when it is awakened (by another thread's *poll*) a communication will have been completed. Observe that *every* completed communication consists of one thread's *sync* calling *enqueue* after which it waits until another thread's sync calls *poll,* which wakes up the waiting process. Of course one of threads must be operating on a SendEvt and the other on a RecvEvt, but either can be *enqueue*'s caller while the other calls *poll.* (Don't confuse *poll* here with the poll Unix system call. The two are not related.)

Synchronization: In this implementation it is still sufficient to lock the channel object while performing all operations because only a single channel is involved in all operations.

Stage 3: Full implementation with Select

The final step is to add synchronous selective communication. To do this we add the SelectionList class which has methods for constructing a list of communication events and for performing a select() operation on the list. Events in a SelectionList can consist of a mixture of SendEvts and RecvEvts. In addition different events may refer to different channels. You may assume, however, that a SendEvt and a RecvEvt for the same channel do not ever get put on the same SelectionList.

select() performs one of the events in the SelectionList by matching it with a complementary event on the same channel. The complementary event may be being sync'd directly or it may be itself part of a SelectionList on which *select* is being performed (by a different thread, of course). (Don't confuse *select* here with the select Unix system call. The Unix select call returns data indicating which file descriptor(s) are ready to perform I/O. Our *select* will actually perform one interaction with another thread.

To implement select, call *poll* for each element of the SelectionList. If one of the calls to *poll* finds a matching event and causes the communication to occur we're done. Otherwise, select() calls *enqueue* for each element of the SelectionList, thus adding each event to the approriate queue of the appropriate channel. After calling *enqueue* for all the elements, *select* waits. When awoken, it figures out which event was matched (there must be only one!), removes all the other events of the SelectionList from their channel queues, and returns the successful event.

Synchronization:

Select is manipulating multiple channels: there is a serious risk of deadlock due to acquiring locks in different orders if channel locks are used. At this stage, I suggest using a single, global, lock. When you have successfully implemented all *select* functionality only then should you think about finer-grained locking. (Finer-grained locking is not required to receive full credit for the project, but it is interesting to try to figure out how to do it.)

Testing

To test the first stage implementation, two sender threads and a receiver thread that successfully pass about 10000 values on a single channel should be adequate.

Testing the second stage is not really any harder: just replace the send and recv calls to calls of new SendEvent(...).sync() and new RecvEvent(...).sync();

Testing the third stage *i*s harder. I suggest starting with two senders sending on different channels using sync() on SendEvents, while the receiver selects on two RecvEvents (one for each channel). When that works then make sure that a single sender using a select on two channels is able also to complete the test.

I'll supply a more interesting test a week before the final due date.

References

Synchronous communication in the form here is found in the Concurrent ML language (though threads a much lighter-weight than Java threads and there are other factors that increase its appeal in that context.) Concurrent ML references that are easily available include

Reppy, J.H. "CML: A higher-order concurrent language." In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation,* pp. 293-305.

Reppy, J.H. *Higher Order Concurrency,* Ph.D. Dissertation, Cornell University, 1992. Cornell Computer Science Tech Report TR 92-1285.

There is also a book by John Reppy, *Concurrent Programming in ML,* of which the WSU library has a copy but availability is always iffy.