Synchronous Message Passing Kernel
Concurrent Programming Project 2

21st October 2002

General Description

In this project you will implement synchronous message passing using shared-memory concurrent programming. A general description of synchronous message passing is found in the textbook in Section 7.5. The details for this project are found in the interface specification below.

You may use Java or C/C++ with pthreads. 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 much easier if you make good use of the libraries available in either of these systems 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.

Oct. 31: a working implementation of synchronous send and receive for channels (stage 1) is due. See the interface specification and implementation notes below.

Nov. 7: a working implementation of synchronous event-based communication (stage 2) is due.

Nov. 19: the full implementation including select() is due.

Interface Specification

Interfaces for synchronous communication. I've left out constructors when they are trivial.

For the first stage implementation:

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

For the second stage implementation. Notice also that the Send and Recv operations of Channel no longer are used.

```
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.) However, you may 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 use each CommEvent and SelectionList only in the thread in which it is created.

Optional

You can make use of SendEvents in SelectionLists more efficient by adding a method to change the value sent. Thus, the whole SelectionList can be used repeatedly to send different values. To do this, just add method

```
void sendValue(Object o)
```

to Class SendEvent.

Implementation notes

Java monitors seem 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 your'e pretty much forced to do with Java's single-lock, single-CV-per-object model).

I suggest doing an initial design with conditional atomic actions, then translating it to use semaphores. Note that you may want to use a semaphore object per thread (or even per operation). I am much less concerned about utmost efficiency than about succesful implementation. Don't succumb to premature optimization disease.

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. Here are notes on the stages.

Stage 1: Channels with Send and Receive

Synchronous communications 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. Channels have two operations, send and receive. Let's consider send. It first checks the recv queue – if non-empty, it matches the current operation with one from the queue, moves the data to the receiver, and releases the queued process to run. If the recv queue is empty, an entry is made in the send queue and the sender waits. Recv is completely analogous.

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

The only problem here is *how* to do the synchronization: Java monitors seem awkward. For example, when a send awakens a waiting receive, we'd like to wake up exactly the process that corresponds to that receive operation. (In this version of the program this isn't such a big deal, but it later becomes more important, so it is worth working it out here.) I suggest that you consider starting with a design using conditional atomic actions and translate it to one using semaphores and passing-the-baton. You'll need to implement semaphores in Java.

If you are using pthreads, I expect that the library facilities will be adequate and that you'll be able to use multiple CVs associated with a single monitor lock to achieve what is needed.

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, an event's sync() method must be called.

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 *block* method, in addition to its sync method. The sync method 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 the block which adds the event to the correct queue of the channel. After that sync waits – when it is awakened another thread's poll will have completed the communication. Observe that every completed communication consists of one thread's sync calling block and then waiting, and another thread's sync calling poll and waking up a waiting process.

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. In doing this we add the SelectionList class which has methods for constructing a list of communication events and for performing a select() operation. Events in a SelectionList can consist of a mixture of SendEvts and RecvEvts. In addition different events may refer to different channels.

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 the parameter of a sync() or it may be a member of a SelectionList passed to another select().

select() is implemented by calling 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 the blockFn for each element of the SelectionList, thus adding each event to the approriate queue of the appropriate channel. After calling all the blockFns 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.)

Alternative Implementation Technique

The techniques described above treat each communicating process as an equal peer. The communicating processes collectively implement the synchronous communication algorithms.

As an alternative, you might implement this project using a clearinghouse process as described in Chapter 10. At the cost of more thread switching you might find that the code is somewhat simpler.

Testing

To test the Stage 1 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 of the Stage 1 test with calls to new SendEvent(...).sync() and new RecvEvent(...).sync();

Testing the third stage is 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 due date.