5. Message-Passing

Threads can communicate and synchronize by sending and receiving messages across channels.

A channel is an abstraction of a communication path between threads:
- If shared memory is available, channels can be implemented as objects that are shared by threads.
- Without shared memory, channels can be implemented using kernel routines that transport messages across a communication network (Chapter 6)

Outline:
- Basic message passing using send and receive commands.
- Rendezvous.
- Testing and debugging message-passing programs.

5.1 Channel Objects

Threads running in the same program (or process) can access channel objects in shared memory.

```java
channel requestChannel = new channel();
channel replyChannel = new channel();
Thread1
requestChannel.send(request); => request = requestChannel.receive();
reply = replyChannel.receive(); <= replyChannel.send(reply);
```

A thread that calls `send()` or `receive()` may be blocked. Thus, send and receive operations are used both for communication and synchronization.

Types of send and receive operations:
- blocking send: the sender is blocked until the message is received (by a `receive()` operation).
- buffer-blocking send: messages are queued in a bounded message buffer, and the sender is blocked only if the buffer is full.
- non-blocking send: messages are queued in an unbounded message buffer, and the sender is never blocked.
- blocking receive: the receiver is blocked until a message is available.
- non-blocking receive: the receiver is never blocked. A receive command returns an indication of whether or not a message was received.

Asynchronous message passing: blocking receive operations are used with either non-blocking or buffer-blocking send operations.

Synchronous message passing:
- The send and receive operations are both blocking.
- Either the sender or the receiver thread will be blocked, whichever one executes its operation first.
- Can be simulated using asynchronous message passing: the sender can issue a buffer-blocking send followed immediately by a blocking receive.
5.1.1 Channel Objects in Java

Threads in the same Java Virtual Machine (JVM) can communicate and synchronize by passing messages through user-defined channels that are implemented as shared objects.

```java
public abstract class channel {
    public abstract void send(Object m); // send a message object
    public abstract void send();     // acts as a signal to the receiver
    public abstract Object receive();   // receive an object.
}
```

Three types of channels:
- **mailbox**: many senders and many receivers may access a mailbox object
- **port**: many senders but only one receiver may access a port object
- **link**: only one sender and one receiver may access a link object

Each type has a synchronous version and an asynchronous version.

A synchronous **mailbox** class is shown in Listing 5.1:

```java
public class mailbox extends channel {
    private Object message = null;
    private final Object sending = new Object();
    private final Object receiving = new Object();
    private final binarySemaphore sent = new binarySemaphore(0);
    private final binarySemaphore received = new binarySemaphore(0);
    public final void send(Object sentMsg) {
        if (sentMsg == null) {throw new
            NullPointerException("Null message passed to send()");
        }
        synchronized (sending) {
            message = sentMsg;
            sent.V(); // signal that the message is available
            received.P(); // wait until the message is received
        }
    }
    public final void send() {
        synchronized (sending) {
            message = new Object(); // send a null message
            sent.V();   // signal that message is available
            received.P();   // wait until the message is received
        }
    }
    public final Object receive() {
        Object receivedMessage = null;
        synchronized (receiving) {
            sent.P();   // wait for message to be sent
            receivedMessage = message;
            received.V(); // signal the sender that the message has
                          //    been received
            return receivedMessage;
        }
    }
}
```

Listing 5.1 A synchronous mailbox class.

Note: A send() operation with no message acts as a signal to the receiver that some event has occurred. Such a send is analogous to a V() operation on a semaphore.
The `send()` methods for classes `port` and `mailbox` are the same.

The `receive()` methods for classes `port` and `link` are also the same.

Only one thread can ever execute a `receive` operation on a `port` or `link` object. In Listing 5.2, an exception is thrown if multiple receivers are detected.

Since a `link` can have only one sender, a similar check is performed in the `send()` method of class `link`.

```java
public final Object receive() {
    synchronized(receiving) {
        if (receiver == null) // save the first thread to call receive
            receiver = Thread.currentThread();
        // if `currentThread()` is not first thread to call receive, throw an exception
        if (Thread.currentThread() != receiver) throw new
            InvalidLinkUsage("Attempted to use link with multiple receivers");
        Object receivedMessage = null;
        sent.P(); // wait for the message to be sent
        receivedMessage = message; // signal the sender that the message has
        received.V(); // been received
        return receivedMessage;
    }
}
```

Listing 5.2 Synchronous `receive` method for the `link` and `port` classes.

An asynchronous `mailbox` class is shown in Listing 5.3:

- The implementation of the buffer-blocking `send()` operation is based on the bounded-buffer solution in Section 3.5.2.
- The `send()` will block if the message buffer is full.
- The messages that a thread sends to a particular mailbox are guaranteed to be received in the order they are sent.
- Also, if Thread1 executes a `send()` operation on a particular mailbox before Thread2 executes a `send()` operation on the same mailbox, then Thread1’s message will be received from that mailbox before Thread2’s message.

Listing 5.4 shows how to use the `link` class.

- Producer and Consumer threads exchange messages with a `Buffer` thread using links `deposit` and `withdraw`.
- The `Buffer` thread implements a 1-slot bounded buffer.
- The Producer builds a `Message` object and sends it to the `Buffer` over the `deposit` link.
- The `Buffer` then sends the `Message` to the Consumer over the `withdraw` link.

```java
public final class asynchMailbox extends channel {
    private final int capacity = 100;
    private final Object messages[] = new Object[capacity]; // message buffer
    private final countingSemaphore messageAvailable = new countingSemaphore(0);
    private final countingSemaphore slotAvailable = new countingSemaphore(capacity);
    private final binarySemaphore senderMutex = new binarySemaphore(1);
    private final binarySemaphore receiverMutex = new binarySemaphore(1);
    private int in = 0, out = 0;

    public final void send(Object sentMessage) {
        if (sentMessage == null) {
            throw new NullPointerException("null message passed to send()");
        }
        slotAvailable.P();
        senderMutex.P();
        messages[in] = sentMessage; in = (in + 1) % capacity;
        senderMutex.V();
        messageAvailable.V();
    }
}
```

Listing 5.3 Asynchronous `asynchMailbox` class.
public final void send() {
    /* same as send(Object sentMessage) above except that the line
    "messages[in] = sentMessage;" becomes "messages[in] = new Object();" */
}

public final Object receive() {
    messageAvailable.P();
    receiverMutex.P();
    Object receivedMessage = messages[out];
    out = (out + 1) % capacity;
    receiverMutex.V();
    slotAvailable.V();
    return receivedMessage;
}

Listing 5.3 (cont.) Asynchronous asynchMailbox class.

final class Message {
    public int number;
    Message(int number) {this.number = number;}
}

final class Producer extends Thread {
    private link deposit;
    public Producer (link deposit) { this.deposit = deposit; }
    public void run () {
        for (int i = 0; i<3; i++) {
            System.out.println("Produced "+i);
            deposit.send(new Message(i));
        }
    }
}

Listing 5.4 Java bounded buffer using channels.

Note: The receiver expects mutually exclusive access to the message. If the sender needs
to access a message after it is sent, or just to be completely safe, the sender should send a
clone (copy) of the message:

final class Message implements Cloneable {
    public int number;
    Message(int number) {this.number = number;}
}

Message m(1);
deposit.send((Message) m.clone());

This prevents the sending and receiving threads from referencing the same message
objects.


5.1.2 Channel Objects in C++/Win32

Listing 5.5 shows a C++ version of the synchronous mailbox class.

Methods send() and receive() operate on “smart pointer” objects of type
message_ptr<T>.

- Smart pointers mimic simple pointers by providing pointer operations like
dereferencing (using operator *) and indirection (using operator ->).
- A message_ptr<T> object contains a pointer to a message object of type T.
- Ownership and memory management are handled by maintaining a count of the
message_ptr objects that point to the same message. Copying a message_ptr object
adds one to the count. The message is deleted when count becomes zero.

Sending and receiving message_ptr<T> objects simulates message passing in Java.

- Messages are shared by the sending and receiving threads
- messages are automatically deleted when they are no longer being referenced.
- virtually any type T of message object can be used.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.

Example: A message containing an integer:

A Message object can be sent over channel deposit using:

A Message object is received using:

The receiver expects mutually exclusive access to the message.

If the sender needs to access a message after it is sent, or just to be completely safe, the
sender should send a copy of the message:

The link and port classes can easily be produced from C++ class mailbox<T> and the
Java link and port counterparts.

Listing 5.6 shows a C++/Win32/Pthreads version of the Java bounded buffer program in
Listing 5.4.
class Consumer : public Thread {
 private:
  mailbox<Message>& withdraw;
 public:
  Consumer (mailbox<Message>& withdraw_) : withdraw(withdraw_) {}  
  virtual void* run () {
    for (int i=0; i<3; i++) {
      message_ptr<Message> m = withdraw.receive();
      std::cout << "Consumed " << m->contents << std::endl;
    }
    return 0;
  }
}

class Buffer : public Thread {
 private:
  mailbox<Message>& deposit; mailbox<Message>& withdraw;
 public:
  Buffer (mailbox<Message>& deposit_, mailbox<Message>& withdraw_) : deposit(deposit_), withdraw(withdraw_) {}  
  virtual void* run () {
    for (int i=0; i<3; i++) {
      message_ptr<Message> m = deposit.receive(); withdraw.send(m);
    }
    return 0;
  }
};

int main () {
  mailbox<Message> deposit; mailbox<Message> withdraw;
  std::auto_ptr<Producer> producer(new Producer(deposit));
  std::auto_ptr<Consumer> consumer(new Consumer(withdraw));
  std::auto_ptr<Buffer> buffer(new Buffer(deposit,withdraw));
  producer->start(); consumer->start(); buffer->start();
  producer->join(); consumer->join(); buffer->join();
  return 0;
}

Listing 5.6 C++ bounded buffer using channels.

5.2 Rendezvous

The following message-passing paradigm is common in a client-server environment:

```
Client
Server
loop {
  request.send(clientRequest);
  clientRequest = request.receive();
  /* process clientRequest and compute result */
  reply.send(result);
}
```

Implementing this with basic message passing:

- one channel that the server can use to receive client requests
- one channel for each client’s reply. That is, each client uses a separate channel to receive a reply from the server.

This paradigm can be implemented instead as follows:

```
Client                  Server
entry E;               loop {
  E.accept(clientRequest, result) {
    E.call(clientRequest,result);
  } // end accept()
} // end loop
```

The server uses a new type of channel called an **entry**.

In the client, the pair of send and receive statements:

```
request.send(clientRequest);
result = reply.receive();
```

is combined into the single entry call statement: `E.call(clientRequest, result);`

This call on entry *E* is very similar to a procedure call:

- Object *clientRequest* holds the message being sent to the server. When the call returns, object *result* will hold the server’s reply.
In the server, the code that handles the client’s request is in the form an accept statement
for entry $E$:

```java
E.accept(clientRequest, result) {
   /* process the clientRequest and compute result*/
   result = ...;
}
```

Only one thread can accept the entry calls made to a given entry. When a server thread
executes an accept statement for entry $E$:

- if no entry call for entry $E$ has arrived, the server waits
- if one or more entry calls for $E$ have arrived, the server accepts one call and executes
  the body of the accept statement. When the execution of the accept statement is
  complete, the entry call returns to the client with the server’s reply, and the client and
  server continue execution.

This interaction is referred to as a *rendezvous*.

- Rendezvous are a form of synchronous communication.
- A client making an entry call is blocked until the call is accepted and a reply is
  returned.
- Executing an accept statement blocks the server until an entry call arrives.

Listing 5.7 shows a Java class named *entry* that simulates a rendezvous. Class *entry*
uses the *link* and *port* channels of section 5.1.

A client issues an entry call to entry $E$ as follows:

```java
reply = E.call(clientRequest);
```

```java
class entry {
   private port requestChannel = new port();
   private callMsg cm;
   public Object call(Object request) throws InterruptedException {
      link replyChannel = new link();
      requestChannel.send(new callMsg(request, replyChannel));
      return replyChannel.receive();
   }
   public Object call() throws InterruptedException {
      link replyChannel = new link();
      requestChannel.send(new callMsg(replyChannel));
      return replyChannel.receive();
   }
   public Object accept() throws InterruptedException {
      // check for multiple callers is not shown
      cm = (callMsg) requestChannel.receive();
      return cm.request;
   }
   public void reply(Object response) throws InterruptedException {
      cm.replyChannel.send(response);
   }
   public void reply() throws InterruptedException {
      cm.replyChannel.send();
   }
   public Object acceptAndReply() throws InterruptedException {
      // the check for multiple callers is not shown
      cm = (callMsg) requestChannel.receive();
      cm.replyChannel.send(new Object()); // send empty reply back to client
      return cm.request;
   }
   private class callMsg {
      Object request;
      link replyChannel;
      callMsg(Object m, link c) { request = m; replyChannel = c; }
      callMsg(link c) { request = new Object(); replyChannel = c; }
   }
}
```

Listing 5.7 Java class *entry*. 
Implementation of method \texttt{call():}
\begin{itemize}
\item a \texttt{send()} operation is issued on a port named \texttt{requestChannel}.
\item The \texttt{send()} operation sends the \texttt{clientRequest}, along with a \texttt{link} named \texttt{replyChannel}, to the server.
\item A new \texttt{replyChannel} is created on each execution of \texttt{call()}, so that each client sends its own \texttt{replyChannel} to the server.
\item The \texttt{call()} operation ends with \texttt{replyChannel.receive()}, allowing the client to wait for the server’s reply.
\end{itemize}

The server accepts entry calls from its client and issues a reply:
\begin{verbatim}
request = E.accept(); // accept client's call to entry E
E.reply(response); // reply to the client
\end{verbatim}

The \texttt{accept()} method in class \textit{Entry} is implemented using a \texttt{receive()} operation on the \texttt{requestChannel} (see Fig. 5.8):
\begin{itemize}
\item \texttt{accept()} receives the \texttt{clientRequest} and the \texttt{replyChannel} that was sent with it.
\item \texttt{accept()} saves the \texttt{replyChannel} and returns the \texttt{clientRequest} to the server thread.
\item method \texttt{reply()} sends the server’s response back to the client using \texttt{replyChannel.send()}.
\item the client waits for the server’s response by executing \texttt{replyChannel.receive()} in method \texttt{call()}.
\end{itemize}

If the server does not need to compute a reply for the client, the server can execute method \texttt{acceptAndReply():}
\begin{itemize}
\item accepts the client’s request and sends an empty reply back to the client so that the client is not delayed.
\item The client simply ignores the reply.
\end{itemize}

Listing 5.9 shows a client and server program using Java \textit{entries} and rendezvous.

![Figure 5.8 Implementing entries.](image_url)
Listing 5.9 Client and server using Java entries and rendezvous.

5.3 Selective Wait

Assume that server thread boundedBuffer has entries deposit and withdraw. Two possible implementations of the run() method of thread boundedBuffer are shown below:

```
    Implementation 1        Implementation 2
    while (true) {           while (true) {
      if (buffer is not full) { if (buffer is not empty) {
        item = deposit.acceptAndReply(); withdraw.accept();
        …                        …
        withdraw.reply(item);    }
      } else if (buffer is not empty) {
        withdraw.accept();       if (buffer is not full) {
        …                        …
        item = deposit.acceptAndReply();
        withdraw.reply(item);    …
      }
    }
```

Both implementations create unnecessary delays for threads calling entries deposit and withdraw:

- Implementation 1: while boundedBuffer is blocked waiting to accept an entry call to deposit, it is possible that a call to withdraw has arrived and is waiting to be accepted.
- Implementation 2: while boundedBuffer is blocked waiting to accept an entry call to withdraw, it is possible that a call to deposit has arrived and is waiting to be accepted.

⇒ allow a thread to wait for a set of entries instead of a single entry:

```
select:
  when (the buffer is not full) => accept a call to entry deposit and deposit the item;
  or
  when (the buffer is not empty) => accept a call to entry withdraw and return item;
end select;
```

When one of the entries is acceptable and the other is not, then the acceptable entry is selected for execution. When both entries are acceptable, then one of them is selected for execution.
The Ada language provides a select statement just like the one above.

A select statement can optionally contain either a delay or else alternative:
- A delay \( t \) alternative is selected if no entry can be accepted within \( t \) seconds.
- An else alternative is executed immediately if no entries are acceptable.

Ada’s selective wait statement can be simulated in Java by a class named `selectiveWait`:

1. Create a `selectiveWait` object
   ```java
   selectiveWait select = new selectiveWait();
   ```
2. Add one or more `selectableEntry` objects to the `selectiveWait`:
   ```java
   selectableEntry deposit = new selectableEntry();
   selectableEntry withdraw = new selectableEntry();
   select.add(deposit);
   select.add(withdraw);
   ```

`selectableEntry` objects are `entry` objects that have been extended so they can be used as alternatives in a `selectiveWait`.

A `selectiveWait` can also contain one `delayAlternative`:
```java
selectiveWait.delayAlternative delayA = select.new delayAlternative(500); // .5 sec.
```
or one `elseAlternative` (but not both):
```java
selectiveWait.elseAlternative elseA = select.new elseAlternative();
```

A delay or else alternative is added to a selective wait in the same way as a `selectableEntry`:
```java
select.add(delayA);
```

Each `selectableEntry` and `delayAlternative` is associated with a condition, called a `guard`, which determines whether the alternative is allowed to be selected.

The guard for each `selectableEntry` and `delayAlternative` must be evaluated before a selection takes place.

Method `guard()` is called with a `boolean` expression that sets the guard to `true` or `false`:
```java
deposit.guard (fullSlots<capacity); // guard set to boolean value (fullSlots<capacity)
withdraw.guard(fullSlots>0);   // guard set to boolean value (fullSlots>0)
delayA.guard(true);     // guard is always set to true
```

Method `choose()` selects one of the alternatives with a true guard:
```java
switch (select.choose()) {
    case 1: deposit.acceptAndReply(); /* alternative 1 */
         break;
    case 2: withdraw.accept();    /* alternative 2 */
         break;
    case 3: delayA.accept();    /* alternative 3 */
        break;
}
```

Method `choose()` returns the alternative number of the selected alternative. Alternative numbers are based on the order in which the alternatives are added to the selective wait.

In the example above,
- `selectableEntry object deposit` was added first, thus its alternative number is 1.
- the alternative number for `withdraw` is 2
- the number for `delayA` is 3.

A switch statement uses the alternative number to execute the appropriate alternative.
When the `selectiveWait` contains no `delay` or `else` alternatives:
- `choose()` will select an open `accept()` alternative (i.e., one with a true guard) that has a waiting entry call.
- if several `accept()` alternatives are open and have waiting entry calls, the one whose entry call arrived first is selected.
- if one or more `accept()` alternatives are open but none have a waiting entry call, `choose()` blocks until an entry call arrives for one of the open `accept()` alternatives.

When the `selectiveWait` has an `else` or `delay` alternative:
- an `else` alternative is executed if all the `accept()` alternatives are closed, or all the open `accept()` alternatives have no waiting entry calls.
- an open `delay` alternative is selected when its expiration time is reached if no open `accept()` alternatives can be selected prior to the expiration time.

When all of the guards of the accept alternatives are false, and there is no delay alternative with a true guard and no else alternative, method `choose()` throws a `SelectException` indicating that a deadlock has been detected.

Listing 5.10 shows a ` boundedBuffer ` server class that uses a selective wait. The `delayAlternative` simply displays a message when it is accepted.

Note that the guards of all the alternatives are evaluated each iteration of the while-loop.
- Changes made to variables `fullSlots` and `emptySlots` in the alternatives of the switch statement may change the values of the guards for entries `deposit` and `withdraw`.
- This requires each guard to be reevaluated before the next selection occurs.

```java
final class boundedBuffer extends Thread {
private selectableEntry deposit, withdraw;
private int fullSlots=0; private int capacity = 0;
private Object[] buffer = null; private int in = 0, out = 0;
public boundedBuffer(selectableEntry deposit, selectableEntry withdraw, int capacity) {
    this.deposit = deposit; this.withdraw = withdraw; this.capacity = capacity;
    buffer = new Object[capacity];
}

public void run() {
    try {
        selectiveWait select = new selectiveWait();
        selectiveWait.delayAlternative delayA = select.new delayAlternative(500);
        select.add(deposit);    // alternative 1
        select.add(withdraw);   // alternative 2
        select.add(delayA);    // alternative 3
        while(true) {
            withdraw.guard(fullSlots>0);
            deposit.guard (fullSlots<capacity);
            delayA.guard(true);
            switch (select.choose()) {
               case 1: Object o = deposit.acceptAndReply();
                       buffer[in] = o;
                       in = (in + 1) % capacity;  ++fullSlots;
                       break;
               case 2: withdraw.accept();
                       Object value = buffer[out];
                       withdraw.reply(value);
                       out = (out + 1) % capacity; --fullSlots;
                       break;
               case 3:  delayA.accept();
                       System.out.println("delay selected");
                       break;
            }
        }
    } catch (InterruptedException e) {
    } catch (SelectException e) {
        System.out.println("deadlock detected"); System.exit(1);
    }
}

Listing 5.10 Java bounded buffer using a `selectiveWait`. 
```
5.4 Message-Based Solutions to Concurrent Programming Problems

5.4.1 Readers and Writers

Listing 5.11 shows a solution to the readers and writers problem for strategy R>W.1 (many readers or one writer, with readers having a higher priority).

Class **Controller** uses a `selectiveWait` with `selectablePort` objects `startRead`, `endRead`, `startWrite`, and `endWrite`.

(A `selectablePort` object is a **synchronous** port that can be used in selective waits.)

- The guard for `startRead` is `(!writerPresent)`, which ensures that no writers are writing when a reader is allowed to start reading.
- The guard for `startWrite` is `(!writerPresent && readerCount == 0 && startRead.count() == 0)`. This allows a writer to start writing only if no other writer is writing, no readers are reading, and no reader is waiting for its call to `startRead` to be accepted.
- Note: the call to `startRead.count()` returns the number of `startRead.send()` operations that are waiting to be received.

The `selectablePorts` are private members of **Controller** and thus cannot be directly accessed by reader and writer threads:
- readers and writers call public methods `read()` and `write()`, respectively.
- public methods `read()` and `write()` their calls on to the private entries, ensuring that the entries are called in the correct order (i.e., `startRead` is called before `endRead`, and `startWrite` is called before `endWrite`).

final class Controller extends Thread {
    //Strategy R>W.1 : Many readers or one writer; readers have a higher priority
    private selectablePort startRead = new selectablePort();
    private selectablePort endRead = new selectablePort();
    private selectablePort startWrite = new selectablePort();
    private selectablePort endWrite = new selectablePort();
    private boolean writerPresent = false;
    private int readerCount = 0;
    private int sharedValue = 0;

    public int read() {
        try {startRead.send();} catch(Exception e) {}
        int value = sharedValue;
        try {endRead.send();} catch(Exception e) {}
        return value;
    }

    public void write(int value) {
        try {startWrite.send();} catch(Exception e) {}
        sharedValue = value;
        try {endWrite.send();} catch(Exception e) {}
    }

    public void run() {
        try {
            selectiveWait select = new selectiveWait();
            select.add(startRead); // alternative 1
            select.add(endRead); // alternative 2
            select.add(startWrite); // alternative 3
            select.add(endWrite); // alternative 4
            while(true) {
                startRead.guard(!writerPresent);
                endRead.guard(true);
                startWrite.guard(!writerPresent && readerCount == 0 &&
                                 startRead.count() == 0);
                endWrite.guard(true);
                switch (select.choose()) {
                    case 1: startRead.receive(); ++readerCount; break;
                    case 2: endRead.receive(); --readerCount; break;
                    case 3: startWrite.receive(); writerPresent = true; break;
                    case 4: endWrite.receive(); writerPresent = false; break;
                }
            } catch (InterruptedException e) {}
        }
    }
}

Listing 5.11 Readers and writers using a `selectiveWait`. 
5.4.2 Resource Allocation

Listing 5.12 shows a solution to the resource allocation problem.

- A resourceServer manages three resources.
- A client calls entry acquire to get a resource and entry release to return the resource.
- Vector resources contains the IDs of the available resources.
- The integer available is used to count the number of available resources.
- A resource can be acquired if the guard (available > 0) is true.
- A resource’s ID is given to the client that acquires the resource. The client gives the resource ID back when it releases the resource.

Listing 5.13 shows an SU monitor solution for this same resource allocation problem.

Monitor resourceMonitor and thread resourceServer demonstrate a mapping between monitors and server threads:

- Thread resourceServer is an active object that executes concurrently with the threads that call it.
- The resourceMonitor is a passive object, not a thread, which does not execute until it is called.
- A monitor cannot prevent a thread from entering one of its methods (although the monitor can force threads to enter one at a time). However, once a thread enters the monitor, the thread may be forced to wait on a condition variable until a condition becomes true.
- Condition synchronization in a server thread works in the opposite way. A server thread will prevent an entry call from being accepted until the condition for accepting the call becomes true.

It has been shown elsewhere that a program that uses shared variables with semaphores or monitors can be transformed into an equivalent program that uses message passing, and vice versa.

final class resourceServer extends Thread {
  private selectableEntry acquire;
  private selectableEntry release;
  private int numResources = 3;
  private int available = numResources;
  Vector resources = new Vector(numResources);
  public resourceServer(selectableEntry acquire, selectableEntry release) {
    this.acquire = acquire;
    this.release = release;
    resources.addElement(new Integer(1));
    resources.addElement(new Integer(2));
    resources.addElement(new Integer(3));
  }
  public void run() {
    int unitID;
    try {
      selectiveWait select = new selectiveWait();
      select.add(acquire); // alternative 1
      select.add(release);  // alternative 2
      while(true) {
        acquire.guard(available > 0);
        release.guard(true);
        switch (select.choose()) {
          case 1: acquire.accept();
            unitID = ((Integer) resources.firstElement()).intValue();
            // Replying early allows the client to proceed as soon as possible
            acquire.reply(new Integer(unitID));
            --available;
            resources.removeElementAt(0);
            break;
          case 2: unitID = ((Integer) release.acceptAndReply()).intValue();
            ++available;
            resources.addElement(new Integer(unitID));
            break;
        }
      }
    } catch (InterruptedException e) {} 
  }
}

Listing 5.12 Resource allocation using a selectiveWait.
final class resourceMonitor extends monitorSU {  
    private conditionVariable freeResource = new conditionVariable();  
    private int available = 3;  
    Vector resources = new Vector(3);  
    public resourceMonitor() {  
        resources.addElement(new Integer(1));  
        resources.addElement(new Integer(2));  
        resources.addElement(new Integer(3));  
    }  
    public int acquire() {  
        int unitID;  
        enterMonitor();  
        if (available == 0)  
            freeResource.waitC();  
        else  
            --available;  
        unitID = ((Integer)resources.firstElement()).intValue();  
        resources.removeElementAt(0);  
        exitMonitor();  
        return unitID;  
    }  
    public void release(int unitID) {  
        enterMonitor();  
        resources.addElement(new Integer(unitID));  
        if (freeResource.empty()) {  
            ++available;  
            exitMonitor();  
        } else  
            freeResource.signalC_and_exitMonitor();  
    }  
}  

Listing 5.13 Resource allocation using a monitor.

5.4.3 Simulating Counting Semaphores

Listing 5.14 shows an implementation of a countingSemaphore that uses a selectiveWait with selectablePorts named P and V.

Clients call public methods P() and V(), which pass the calls on to the (private) ports.

A P() operation can be performed only if the guard (permits>0) is true.

class countingSemaphore extends Thread {  
    private selectablePort V, P;  
    private int permits;  
    public countingSemaphore(int initialPermits) {permits = initialPermits;}  
    public void P() { P.send();}  
    public void V() { V.send();}  
    public void run() {  
        try {  
            selectiveWait select = new selectiveWait();  
            select.add(P); // alternative1  
            select.add(V); // alternative 2  
            while(true) {  
                P.guard(permits>0);  
                V.guard(true);  
                switch (select.choose()) {  
                case 1: P.receive();  
                    --permits;  
                    break;  
                case 2: V.receive();  
                    ++permits;  
                    break;  
                }  
            }  
            } catch (InterruptedException e) {}  
        }  
    }  

Listing 5.14 Using a selectiveWait to simulate a counting semaphore.