8.1 Chapter Objective

The objective of the Ports project is to teach students about interprocess communication and requires that the student implement two public classes: Message, which describes what OSP 2 messages look like, and Port-CB, which implements the main communication primitives, such as send() and receive().

8.2 Interprocess Communication in OSP 2

Interprocess communication in OSP 2 is based on the abstraction of a port and is modeled after the Mach micro-kernel. In Mach, a process can open a port, and other processes can then send messages to it which can be received by the owning process; a message is basically a block of bytes. Mach manages the ports, and provides guaranteed, in-order delivery, with large messages being handled efficiently by sharing pages between address spaces. There is a sophisticated permission mechanism which restricts the operations that processes can perform on ports.

Thus, a port is like your home mailbox. A task can create a port to serve as a mailbox to which threads from other tasks can send messages. Only threads

1. Note that threads of the same task do not need to communicate this way, since
of the owner task can read from the ports of that task; other threads only write to that port. In OSP 2, reading from a port is done using the `receive()` operation and writing is performed via the `send()` operation.

The OSP 2 model of communication is based on **reliable** message delivery, i.e., correctly formed messages never get lost. When threads communicate, they exchange discrete entities, called **messages**. A message has length and Id. When a thread sends a message to a port, the message is delivered to the destination port and is placed in that port’s **message buffer**. Port buffers are assumed to have finite byte size specified in a global constant `PortBufferLength`. If the message is bigger than this amount, the `send()` operation fails and the message is not delivered. If the message is smaller than `PortBufferLength`, it is considered well-formed and deliverable. However, the destination port might not have enough room due to other messages that might have been delivered to that port but not yet consumed. In this case, the `send()` operation suspends the sender thread until room becomes available.

When a thread wants to receive a message, it invokes the `receive()` method on a port. If a message is available, it is removed from the port message buffer and the operation succeeds. If, however, the port is empty, then the receiver thread is suspended until a message arrives.

It is thus clear that a mechanism is needed for threads to suspend themselves and to be notified. In OSP 2, this is accomplished through the familiar **Event** class. More precisely, `PortCB` is a subclass of `Event`, and threads can suspend themselves on a port when necessary. Likewise, when appropriate conditions arise (e.g., a port buffer gets more room or a message arrives at an empty port), threads that are waiting on the port can be notified. (Note that several threads can be waiting on the same port at the same time.)

The classes comprising the **PORTS** package are described below. The class diagram of Figure 8.1 places these classes in the overall context of the OSP 2 system.

### 8.3 The Message Class

The **Message** class has only one required method, the class constructor, which takes a `length` argument and creates a message with a unique Id.

```
public Message(int length)
```

The message constructor. Must call `super(length)` as its first statement. Your implementation might also add other fields and methods to this class.

they share virtual address space and thus can communicate much more efficiently through shared variables.