# 8
# PORTS: *Interprocess Communication*

## 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 `PortCB`, 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.[1] 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.

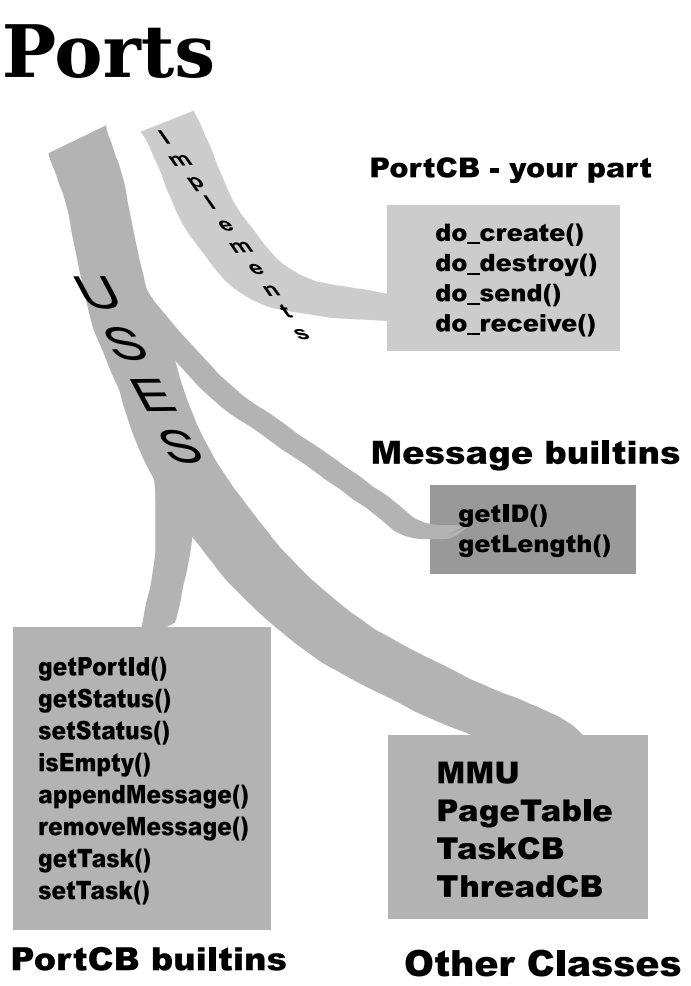


**Figure 8.1** A diagram summarizing the package PORTS.

In addition, your implementation of class `PortCB` can use a number of methods defined in class `Message` provided by *OSP 2*:

◇ `public int getID()`
Returns the Id of the message.

◇ `public int getLength()`
Returns the length of the message in bytes.

**Built-ins and relevant methods defined in other classes.** The method constructor for class `Message` does not use any methods provided by other *OSP 2* classes.

### Summary of Class `Message`

A message in *OSP 2* is a simplified abstraction of messages used in real communication protocols, such as TCP/IP: it includes only these two parameters:

ID: The ID of a message. The value of an ID can be retrieved using the method `getID()`.

length: The length of a message. This parameter can be queried using the method `getLength()`.

## 8.4 The `PortCB` Class

The methods of `PortCB` to be implemented as part of the student project include the class constructor, the initialization method, the methods for creating/destroying ports and for sending/receiving messages.

A port has an Id, the owner task, a status (`PortLive` or `PortDestroyed`), and a message buffer. *OSP 2* provides methods for manipulating the message buffer of a port (`appendMessage()`, `removeMessage()`, `isEmpty()`), but the student implementation must keep track of the free space left in the buffer in order to be able to correctly decide when a message can be sent to the port.

◇ `public PortCB()`
This is a class constructor whose only required statement is `super()`, the usual call to the corresponding constructor in the superclass.

◇ `public static void init()`
This is the usual initialization method, which is called at the very beginning of the simulation run. It is a place where your implementation can initialize static variables.

◇ `public static PortCB do_create()`
This method creates and returns a new port. After a new `PortCB` object is created, it needs to be assigned to the current task, i.e., the task that owns the currently running thread. Recall from Chapter 5 that `PTBR`, the page table base register, always points to the page table of the current

task. Thus, the current task can be retrieved using the following idiom: `MMU.getPTBR().getTask()`.

To assign the port to the task, use the method `addPort()` of `TaskCB`. However, keep in mind that there is a limit of how many ports a task can have, which is defined by the global constant `MaxPortsPerTask`. If the task already has that many ports, `addPort()` will return `FAILURE` and `do_create()` should then return the `null` object.

If all is well, the owner task of the port should be set (using `setTask()`), and the status set to `PortLive` using the method `setStatus()` of class `PortCB`, which is provided by *OSP 2*. In addition, you have to initialize the variables that you might have introduced to keep track of the state of the message buffer. Finally, the newly created `PortCB` object is returned.

◇ `public void do_destroy()`
Ports are destroyed by the owner task when they are no longer needed for the task's operation or when the task itself is killed. To destroy a port, the port's status should be set to `PortDestroyed`, and the port should be removed from the task's table of active ports. The latter is accomplished using the method `removePort()` of `TaskCB`. Next, the port's owner task should be set to null using the method `setTask()` of `PortCB`.

You must also notify the threads that might be waiting for an event associated with this port. As usual, this is accomplished using the method `notifyThreads()` applied to the appropriate event.

◇ `public int do_send(Message msg)`
Prior to sending a message, you must first check that the message is well-formed. In *OSP 2*, this means that the parameter `msg` is not `null` and that the message length is not greater than the length of the port message buffer. If the message is not well-formed, `FAILURE` should be returned.

In the next step, a new system event must be created using the constructor `SystemEvent()` and the current thread must be suspended on that event. You already saw how to find the current task from the page table base register. The current thread is obtained using the method `getCurrentThread()` of that task.

At this point it is recommended that you refresh your memory about thread suspension and resumption as described in Section 4.3. A thread that is suspended on a system event is not really blocked, but instead can be thought of as having changed status from user thread to system thread. When the send operation is complete, the event will "happen" and the thread will be resumed. To be able to resume the thread before leaving `do_send()`, you

should save the `SystemEvent` object in a variable.

Now you are ready to attempt to send the message. Recall that if the destination port (i.e., the port on which the `send()` method is executed) does not have enough room in the message buffer, the sender thread must be suspended on that port. (Recall that you have saved the information about that thread before suspending it on a `SystemEvent`.) A thread $T$ suspended on a port can be woken up when the port gets more room in its buffer. This happens when one of the threads that owns the port executes a `receive()` operation on that port. However, the sending thread $T$ might discover that the port still does not have enough room for the message because either too little space was freed up or because some other thread managed to send a message to the port before $T$ had a chance. In this case, $T$ has to be suspended again (on the same port).

Another possibility is that the newly awakened thread was killed while waiting to send the message. `FAILURE` should be returned in this case. The third possibility is that the thread might have been awakened because the owner task decided to destroy the port on which the thread was suspended (or, maybe, the task itself was killed). Again, `FAILURE` should be returned. In addition, you should notify the threads that were suspended on the `SystemEvent` associated with the current send operation. (Recall that the current thread was suspended on this event at the beginning of the `do_send()` method.)

If none of the above problems are detected, you know that send should succeed. Thus, you should update the message buffer of the port (using `appendMessage()`) and, if the buffer was previously empty, notify the threads that may be waiting on that port in the receive mode.[2] Finally, you should execute `notifyThreads()` on the previously created `SystemEvent` object and return `SUCCESS`.

◇ `public Message do_receive()`

First, you must check that the receive operation is permitted, i.e., that the receiving thread's task owns the port on which `do_receive()` has been invoked. If this is not the case, `null` should be returned. Second, when a thread $T$ executes a `receive()` operation on a port `P`, you must create a `SystemEvent` object and suspend $T$ on that event. As explained earlier, this corresponds to $T$ changing its status from user thread to system thread. Note that the receiving thread $T$ is the currently executing thread, which can be obtained using the PTBR.

[2] Note that other threads may have been waiting to receive a message from this port *only* if its message buffer was empty.

Next, recall that the receiving thread must be suspended if the message buffer of the port contains no messages. This thread can be woken up when some other thread sends a message to that port. However, keep in mind that although a port can have several threads suspended in receive mode, only one of them will be awakened and thereby succeed in getting a message. All other threads would have to be suspended again.

There is a possibility that a woken-up thread was killed or that the port was destroyed. In both cases, `do_receive` must return the `null` object. If none of the above bad things happen, the `do_receive()` method succeeds. In this case, the method should "consume" a message from the port message buffer using `removeMessage()` and notify threads waiting on the port. (This is needed because consuming a message will probably free up space in the message buffer of the port and, as a result, some previously suspended send operation might be able to proceed.) Finally, the message consumed by this receive operation should be returned.

In all cases (whether the receive operation ended successfully or not), prior to exiting you must execute `notifyThreads()` on the previously created `SystemEvent` object for this receive operation.

**Built-ins and relevant methods from other classes.** A typical implementation of the methods in class `PortCB` uses the following methods defined in other classes or methods of `PortCB` provided by *OSP 2*:

◇ `final public int addPort(PortCB newPort)` `TaskCB`
Adds a new port to the task.

◇ `public int removePort(PortCB oldPort)`
Removes `oldPort` from the task.

◇ `public ThreadCB getCurrentThread()` `TaskCB`
Returns the currently running thread of the task. Null, if the task itself is not current.

◇ `static public PageTable getPTBR()` `MMU`
Returns the value of PTBR.

◇ `public final TaskCB getTask()` `PageTable`
Returns the owner task for the page table.

◇ `final public int getStatus()` `ThreadCB`
Tells the status of the thread.

◇ `final public void suspend(Event event)` `ThreadCB`
Suspends the thread on `event`.

◇ `final public int getStatus()` PortCB
Tells the status of the port.

◇ `final public void setStatus()` PortCB
Sets the status of the port.

◇ `final public void setTask(TaskCB owner)` PortCB
Sets the port owner.

◇ `final public TaskCB getTask()` PortCB
Tells who owns the port.

◇ `final public Message removeMessage()` PortCB
Removes a message from the port's message buffer.

◇ `final public void appendMessage(Message msg)` PortCB
Appends a new message to the port's message buffer.

◇ `final public boolean isEmpty()` PortCB
Checks if the port's message buffer is empty.

## Summary of the `PortCB` class

The `PortCB` class maintains information about the open ports attached to the various processes. The following list describes the main attributes of a port and the methods that are used to query these attributes.

Port ID: *OSP 2* assigns an ID to each port at creation time. This ID can be retrieved using the method `getPortID()` of the `PortCB` class.

Owner: This is the task that owns the port. This attribute is manipulated using the methods `getTask()` and `setTask()`.

Status: `PortLive` or `PortDestroyed`. This attribute is manipulated using the methods `getStatus()` and `setStatus()`.

Message buffer: This buffer is manipulated using the methods `appendMessage()`, `removeMessage()`, and `isEmpty()` of class `PortCB`, and are provided by *OSP 2*. However, your implementation must keep track of the free space left in the message buffer.

## 8.5 Methods Exported by Package PORTS

The PORTS package exports the following methods that are used by other packages in the system:

◇ `final static public void create()`
Creates a new port.

◇ `final public void destroy()`
Destroys an existing port.

◇ `final public void send(Message msg)`
Sends a message, `msg`, to the port on which this method is invoked.

◇ `final public Message receive()`
Receives a message from the port on which this method is invoked.