**CIS 657: Principles of Operating Systems**

**Lab3 PortCB**

- Bharath Darapu

- Avinash Peddi

In this lab project we are assigned to finish the following files:  
1. PortCB.java

2. Message.java

And finish the following method implementations:

1. PortCB constructor
2. Init()
3. Do\_create(…)
4. Do\_destroy()
5. Do\_send()
6. Do\_receive()
7. Message constructor

The main idea in this module is dealing with the creation, destroy, send and finally receive messages for ports of current threads.

**Approach used:**

The approach used is pretty simple and was as described in the book (OSP2 approach by Michael Kifer).

Let me explain the implementation in the order of occurrence in the program.

To begin with, two integer variables for maintaining a buffer count per port are maintained.

*private int bufferIn;*

*private int bufferOut;*

Next, we shall be implementing the do\_create() method as shown below:

*System.out.println("PortCB do\_create called");*

*// A new port is being created here*

*PortCB newPort = new PortCB();*

*TaskCB currentTask = null;*

*try {*

*//get the requesting thread*

*currentTask = MMU.getPTBR().getTask();*

*} catch (NullPointerException e){*

*System.out.println("Null pointer exception while getching the current task"+e);*

*}*

*//get the port count of the current task*

*int currentPortNum = currentTask.getPortCount();*

*//Check if adding a new port will exceed the allowed maximum*

*if(currentPortNum == MaxPortsPerTask)*

*{*

*System.out.println("Maximum ports exceeded");*

*return null;*

*}*

*//Check if adding the port to the task failed return null*

*if( currentTask.addPort(newPort) == FAILURE){*

*System.out.println("Adding port failed");*

*return null;*

*}*

*//set the port to the current task*

*newPort.setTask(currentTask);*

*//set the port status as live*

*newPort.setStatus(PortLive);*

*//initialize the new port buffers*

*newPort.bufferIn = 0;*

*newPort.bufferOut = 0;*

*//return the newly created port*

*return newPort;*

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 Section 4 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.

Next, the **do\_destroy()** method implementation is as follows:  
  
*System.out.println("PortCB do\_destroy called");*

*//set status of the port as destroyed*

*this.setStatus(PortDestroyed);*

*//notify the threads*

*this.notifyThreads();*

*//remove the port from the task*

*this.getTask().removePort(this);*

*//port's owner task is set to null*

*this.setTask(null);*

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.

The **do\_send()** implementation is as follows:  
  
*System.out.println("PortCB do\_send called");*

*// your code goes here*

*//Here we check if the message is well-formed or not*

*if( msg == null || (msg.getLength() > PortBufferLength)){*

*System.out.println("message is not well formed");*

*return FAILURE;*

*}*

*//SystemEvent constructor is used to create a new system event*

*SystemEvent newEvent = new SystemEvent("send\_msg\_suspension");*

*TaskCB currentTask = null; //get the requesting thread*

*ThreadCB currentThread = null;*

*//getting the current task*

*try {*

*currentTask = MMU.getPTBR().getTask();*

*currentThread = currentTask.getCurrentThread();*

*}catch (NullPointerException e){*

*System.out.println("Null pointer error in getting the current task");*

*}*

*//current thread is suspened here*

*currentThread.suspend(newEvent);*

*int bufferRoom;*

*//set suspendMsg flag here*

*boolean suspendMsg = true;*

*while(suspendMsg) {*

*//if the destination thread does not have enough room in the //message buffer, the thread has to be suspended on that port*

*if( this.bufferIn < this.bufferOut){*

*bufferRoom = this.bufferOut - this.bufferIn;*

*}*

*else if( this.bufferIn == this.bufferOut){*

*if(this.isEmpty()){*

*bufferRoom = PortBufferLength;*

*}*

*else{*

*bufferRoom = 0;*

*}*

*}*

*else{*

*bufferRoom = PortBufferLength + this.bufferOut - this.bufferIn;*

*}*

*//suspendmsg flag is false*

*if( msg.getLength() <= bufferRoom){*

*suspendMsg = false;*

*}*

*else{*

*//suspend the current thread*

*currentThread.suspend(this);*

*}*

*//if status is kill then remove thread*

*if( currentThread.getStatus() == ThreadKill){*

*System.out.println("Current thread status is kill");*

*this.removeThread(currentThread);*

*return FAILURE;*

*}*

*if( this.getStatus() != PortLive){*

*newEvent.notifyThreads();*

*return FAILURE;*

*}*

*}*

*//update the message buffer of the port*

*this.appendMessage(msg);*

*//notify the threads that are waiting on the*

*this.notifyThreads();*

*//if buffer was previously empty then notify the threads that are waiting on the port in receive mode*

*this.bufferIn = (this.bufferIn + msg.getLength()) % PortBufferLength;*

*newEvent.notifyThreads();*

*System.out.println("message successfully sent");*

*//return success if everything is done correctly*

*return SUCCESS;*

Prior to sending a message, we 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 wellformed, 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. We 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 to refresh your memory and read about thread suspension and resumption in Section 3.2. 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 we 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 waken up when the port gets more room in it 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 our port before T had a chance. In this case, T has to be suspended again (on the same port). Another possibility is that the waken up 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 waken up because the owner task has 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, we 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, we know that send should succeed. Thus, we 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, we should execute notifyThreads() on the previously created SystemEvent object and return SUCCESS.

The do\_receive() method implementation:

First, we 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, we must create a SystemEvent object and suspend T on that event. As explained earlier, this corresponds to T changing the status from being a user thread to a system thread. Note that the receiving thread, T, is the currently executing thread, which can be obtained using PTBR. Next, recall that the receiving thread must be suspended, if the message buffer of the port contains no messages. This thread can be waken up when some other thread sends a message to that port. However, keep in mind that altough a port can have several threads suspended in the receive mode, only one waken up thread will succeed at getting a message. All other threads would have to be suspended again. There is a possibility that a waken up thread was killed or that the port was destroyed. In both cases, do receive must return a 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 we must execute notifyThreads() on the previously created SystemEvent object for this receive operation.

After all the implementation we navigated to the project path and

* Compiled the .java file using ‘javac –classpath .;OSP.jar \*.java’ and
* Later used ‘java -classpath .;Demo.jar osp.OSP’ command to run the simulation.

OSP.log file which was generated during the execution is also attached.