2
Putting it All Together: An Example Session with *OSP 2*

2.1 Chapter Objective

Your instructor has assigned you the THREADS project to implement; see Chapter 4. You are new to *OSP 2*. What do you do? In this chapter, we present an example session with *OSP 2* that is intended to give you the guidance and confidence you need to successfully complete your assignment.

2.2 Overview of Thread Management in *OSP 2*

The THREADS project, as the name implies, deals with thread management and scheduling, where threads are the executable and dispatchable units in *OSP 2*. Our example will focus on thread management, in particular, the resumption of a thread from a waiting state. This activity is the responsibility of the method `do_resume()`, one of the methods you are to implement as part of your implementation of the class `ThreadCB`.

Thread management involves the notions of thread creation, destruction, suspension, resumption and dispatching; maintaining thread status; and moving threads between different (ready and waiting) queues. Underlying all of this is the notion of a thread *state*, which can be one of `ThreadReady`, `ThreadWaiting`, `ThreadKill`, etc.

An *OSP 2* thread assumes the `ThreadWaiting` state when it enters the pagefault handler or when it executes a blocking system call (e.g., `write()`). The `ThreadWaiting` state is also known as the "level-0 waiting state". While in this state, a thread can again enter the pagefault handler or execute a blocking system call, causing it to enter the level-1 waiting state, represented by the constant `ThreadWaiting+1`. This process can continue indefinitely, leading to arbitrarily nested depths of waiting.

When a thread completes the execution of the pagefault handler or blocking system call, it should be moved up to the next highest waiting level by decrementing its waiting status; in the case of level 0 (`ThreadWaiting`, it should transit to the `ThreadReady` state.

2.3 The Student Method `do_resume()`

As mentioned previously, we will focus our attention during this example session on the method `do_resume()` of class `ThreadCB`. Its code is given in Figure 2.1. Notice the use of the `MyOut` utility to insert student output in the file `OSP.log`. For example, the statement

```
MyOut.print(this, "Resuming " + this);
```

will result in output such as

```
Mod: 63 [Threads.ThreadCB]
        Resuming Thread(0:1/W2)
```

appearing in the log file, indicating that at simulation time 63, thread 0 of task 1 is at waiting-level 2 (`W2`). The tag "`Mod:`" identifies this output as being from a student module, making it easy for you to distinguish your output from *OSP 2*'s in the log file.

`Do_resume()` is one of the simplest methods in *OSP 2*. All it needs to do is decrement the thread's waiting-level, place it on the ready queue if its new status is `ThreadReady`, and call `dispatch()` so that some thread can be dispatched onto the CPU for execution.

Assuming that you have completed your design and coding of the THREADS project, let us proceed in a step-by-step fashion with the example session.

```
/** Resumes the thread.

Only a thread with status ThreadWaiting or higher can
be resumed.  The status must be set to ThreadReady or
decremented, respectively.  A ready thread should be
    placed on the ready queue.

@OSPProject Threads
*/
public void do_resume()
{
    if(getStatus() < ThreadWaiting) {
    MyOut.print(this,
        "Attempt to resume "
        + this + ", which wasn't waiting");
    return;
}

    MyOut.print(this, "Resuming " + this);

    // Set thread's status.
if (getStatus() == ThreadWaiting) {
    setStatus(ThreadReady);
} else if (getStatus() > ThreadWaiting)
    setStatus(getStatus()-1);

    // Put the thread on the ready queue, if appropriate
if (getStatus() == ThreadReady)
    readyQueue.append(this);

dispatch();
}
```

Figure 2.1 Code for student method `do_resume()`.

2.4 Step 1: Compiling and Running the Project

◇ You have a directory with all the necessary files in it for the THREADS project: `ThreadCB.java`, `TimerInterruptHandler.java`, `OSP.jar`, `Makefile`, etc.

◇ You have set the environment variable `PATH` appropriately so that the proper version of JDK (1.5 or newer) will be invoked.

◇ On Unix-based systems, you can use the `make` command to compile the project. For this example session, we will compile *OSP 2* to run without the GUI by issuing the command:

```
make runnogui
```

◇ Problems in compiling? If you think this could be due to stale `.class` files,

type `make clean` and then `make` to force recompilation of the entire project.

◇ To now run the project, type:

```
java -classpath .: osp.OSP -noGUI
```

2.5 Step 2: Examining the `OSP.log` File

Assuming for the moment that you have correctly implemented the THREADS project and *OSP 2* ran successfully to completion without errors, let us now take a look at a relevant snippet from the `OSP.log` file:

```
Sim: 63 [Memory.PageTableEntry]
        Unlocking Page(12:1/0). New lock count: 0
Sim: 63 [Threads.ThreadCB]
        Entering resume(Thread(0:1/W2))
Mod: 63 [Threads.ThreadCB]
        Resuming Thread(0:1/W2)
Sim: 63 [Threads.ThreadCB]
        Leaving resume(Thread(0:1/W1))
Mod: 63 [Hardware.Disk]
        Device(0) has no pending IORBs to dequeue
```

At simulation time 63, thread 0 of task 1 has exited the pagefault handler and is "resumed" by the student method `do_resume()`. In this case, this means the thread moves from waiting-level 2 to waiting-level 1.

The log file also contains statistics about tasks and threads generated during our successful run of the THREADS project. It is a good idea to have a look at these too, both to see how well your implementation is performing and to simply get a better understanding of how threads behave in *OSP 2*.

```
TASKS and THREADS:
  CPU Utilization: 61.382%
  Average service time per thread: 36180.812
  Average normalized service time per thread: 0.047044374
  Total number of tasks: 4
  Running thread(s): none
  Threads summary: 18 alive
  Among live threads: 0 running
                      6 suspended
                      0 ready
```

```
ready queue = ()
running thread(s) = ()
waiting thread(s) = (97:12,107:13,110:15,111:15,112:15,113:15)
thread(s) in pagefault = (110:15,115:13,124:13)
killed thread(s) = (7:1,15:1,13:1,12:1,10:1,9:1)
```

2.6 Step 3: Introducing an Error into do_resume()

Unfortunately, not all of your runs of *OSP 2* will be as successful as the one above: we all make programming mistakes, whether they be logical errors or simply typographical errors. Let us consider what happens when the latter occurs. In particular, suppose that in `do_resume()`, instead of typing:

```
} else if (getStatus() > ThreadWaiting)
    setStatus(getStatus()-1);
```

you type:

```
} else if (getStatus() > ThreadWaiting)
    setStatus(getStatus()+1);
```

This is not an uncommon mistake: typing a plus sign when indeed you meant to type a minus sign. What are the consequences of this typo? Well, for one, *OSP 2* will terminate unsuccessfully at simulation time 63 and place the following output in the log file:

```
 Sim: 63
[Threads.ThreadCB]
        Entering resume(Thread(0:1/W2))
Mod: 63 [Threads.ThreadCB]
        Resuming Thread(0:1/W2)
Sim: 63 <<Error!>> [Threads.ThreadCB]
        After do_resume(Thread(0:1/W3)): Thread status is
        ThreadWaiting3; should be ThreadWaiting1

        at osp.IFLModules.IflThreadCB.resume(IflThreadCB.java:1101)
        at osp.IFLModules.Event.notifyThreads(Event.java:130)
        at osp.Devices.DiskInterruptHandler.do_handleInterrupt
           (DiskInterruptHandler.java:114)
        at osp.IFLModules.IflDiskInterruptHandler.handleInterrupt
           (IflDiskInterruptHandler.java:107)
```

```
        at osp.Interrupts.Interrupts.interrupt(Interrupts.java:48)
        at osp.Hardware.CPU.interrupt(CPU.java:54)
        at osp.IFLModules.IflIORB.voidCallback(IflIORB.java:238)
        at osp.IFLModules.CallbackDiskInterrupt.voidCallback
           (IflDevice.java:604)
        at osp.EventEngine.EventCallback.Activate(EventCallback.
           java:48)
        at osp.EventEngine.EventDriver.go(EventDriver.java:119)
        at osp.EventEngine.EngineThread.run(EngineThread.java:60)
```

As you can see, the simulator has detected our error! What follows the error message is a dump of the system-call stack which indicates the sequence of method calls that led to the problem. Not surprisingly, $\mathcal{OSP}$ 2's IFL version of `do_resume` is at the top of the stack, as it was in this "wrapper method" where the error was detected. In an actual debugging situation, you would use this information to isolate and repair the problem in your implementation of the `do_resume()` method.

To complete our example session, here are the statistics for tasks and threads that can be found in the log file at the end of our unsuccessful run.

```
TASKS and THREADS:
  CPU Utilization: 28.57143%
  Average service time per thread: 63.0
  Average normalized service time per thread: 0.28125
  Total number of tasks: 1
  Running thread(s): none
  Threads summary: 1 alive
  Among live threads: 0 running
                      1 suspended
                      0 ready
  ready queue = ()
  running thread(s) = ()
  waiting thread(s) = (0:1)
  thread(s) in pagefault = (0:1)
  killed thread(s) = ()
```