1
Organization of OSP 2

1.1 Chapter Objective

The objective of this chapter is to provide the essential general information about *OSP 2*, which is necessary in order to begin working with the system. This includes a description of the organization (breakdown into modules), instructions on how to compile, run and submit *OSP 2* projects, and general guidelines about programming *OSP 2*. Because of its introductory nature, this chapter should be read/reviewed before taking on any of the *OSP 2* projects your instructor may assign to you.

1.2 Operating System Basics

As explained in the Preface to this book, *OSP 2* is organized as a collection of modules, each corresponding to a class of resource that *OSP 2* is intended to manage. For your *OSP 2* programming assignments, your instructor will assign you one or more of these modules to implement, plug back into the rest of the system, and run via a simulation to ensure that your code is working correctly and efficiently. This chapter describes in some detail this division of *OSP 2* into modules and also provides you with other helpful information you will need to carry out your assignments. First, though, we shall step back and ask ourselves the questions: What is an operating system, and what kind of operating system is *OSP 2*?

What is an Operating System? In order to understand exactly what *OSP 2* is and how it is organized, it is useful to first consider the basic question: What is an operating system? Two generally held views are that an OS is an *extended machine*, and an OS is a *resource manager*. According to the first view, the function of an operating system is to present the user with the equivalent of an "extended machine" or "virtual machine" that is easier to program than the underlying hardware. This is accomplished through the operating system's *system call interface*: the collection of system calls that application programs may invoke to obtain one kind of service or another. For example, there are system calls to read and write files and to set the value of timers. Moreover, it is much easier to invoke these system calls to obtain system service as opposed to mucking around with hardware-specific instructions and machine registers, which one would be forced to do if there was no OS present.

Two well-known examples of system-call interfaces are the Win32 API (application programming interface) for various flavors of Microsoft Windows (Windows 2000/XP/Vista), and POSIX for the Unix flavor of operating systems, such as System V, BSD, and Linux. *OSP 2* has its own system call interface, and you will be introduced to the system calls (Java methods) that constitute this interface in the subsequent chapters of this book.

According to the second view, an operating system is responsible for efficiently and fairly managing the resources of a computer system. These include processors (CPUs); memory (physical and virtual); devices such as disks; files and directories; and network connections (ports). By efficient, we mean that the OS should aim to maximize resource utilization whenever possible. By fair, we mean that users programs should be granted equitable allocation of resources during their execution. Note that most of the example resources we have listed are physical ones. One exception is files and directories. The part of the OS responsible for these "logical resources" is often called the file system.

As we will make clear later in this chapter, the view of an operating system as a resource manager is well suited to *OSP 2*, as *OSP 2*'s system call interface is organized in terms of the various resources *OSP 2* is intended to manage. More specifically, *OSP 2* is organized into a number of modules—Java packages to be precise—and there is one such module for each type of resource *OSP 2* is asked to manage. For example, there is an *OSP 2* module for each of memory, devices, ports, etc., and each module exports (defines) a number of Java methods relevant to that module. Collectively, these methods make up *OSP 2*'s system call interface.

Different Flavors of Operating Systems. To better understand *OSP 2*, it is also useful to realize that there are different flavors of operating systems available for the choosing. Some of those that immediately come to mind, and which you have probably heard of, are Unix, Linux, Windows, and MacOS. These systems differ mainly in the way they are structured and, of course, in their system call interfaces. Systems like Windows XP/Vista, Solaris (a version of Unix from SUN Microsystems), and Mach (an OS developed at Carnegie Mellon University in the 1980s and which later influenced a number of commercial operating systems, e.g., MacOS X) can be viewed as object-oriented in the following sense: basic system resources are represented as objects and there exist well-defined message-passing interfaces between objects.

Although *OSP 2* is not modeled after any particular OS, a bias towards Unix and Mach can be seen in some parts of its architecture. The Unix bias is most evident in the FileSys package, where i-nodes are used to represent files stored on disk and directories map file names to i-numbers (inode indices). The Mach influence can be detected in the Ports package where Mach-like ports are used for interprocess communication. Mach also uses ports for exception handling (each process has an exception port), a topic not treated by *OSP 2*.

OSP 2 is an object-oriented operating system in the truest sense of the term. It is written in the object-oriented programming language Java. System resources and data structures are represented by classes, thereby providing well-defined method-call interfaces between objects à la Windows XP/Vista. And subclassing is used to specialize objects; for example, the I/O Request Block (IORB) is a subclass of `Event` so that threads can wait on it and be notified of its occurrence.

Another way in which operating systems differ, and which in some sense distinguishes older operating systems from newer ones, is whether or not they support *threads*. In older systems like Unix, executing programs are organized as processes: the OS is responsible for scheduling processes on the CPU and switching the CPU from one process to another for the purposes of *multiprogramming*. Multiprogramming is a technique aimed at increasing resource utilization. The basic idea is to have more than one process memory-resident at a time, and to switch the CPU from a process that has become blocked waiting for some event, say, the completion of an I/O operation, to a process that is ready to execute. In this way, the CPU is kept busy doing useful work most of the time, just the kind of thing a resource manager should strive for.

To conclude our brief look at multiprogramming, we should consider a little more carefully what it means to switch the CPU from one process to another, an operation commonly referred to as a *context switch*. Several steps are involved. First, the currently executing process must be removed from the CPU and placed on a queue associated with the event on which it is waiting. Then the process the OS has decided to schedule next for execution must be *dispatched* on to the CPU. This involves resetting a number of machine registers (such as the program counter, general-purpose registers, memory-management registers, etc.) to values associated with the newly dispatched process when it was last running. The execution of this process can now resume. This is an admittedly simplified view of what's behind a context switch; the subject is treated more thoroughly in Chapter 4.

In newer systems like Mac, Solaris, and Windows 2000/XP/Vista, the schedulable and dispatchable units of execution are no longer processes but rather threads; a process simply serves as a container for one or more threads. Processes of this kind are usually referred to as *tasks*, and that shall be the convention adopted in this book. So what does it mean for a task to be a "container" for threads? It means that the constituent threads of a task share the resources allocated to the task, including memory, files, and communication ports. As a result, switching the CPU from one thread to another is a lot simpler than switching the CPU from one process to another process as required in an OS that does not support threads. As we shall see, *OSP 2* supports tasks and threads.

Operating Systems are Event-Driven. Operating systems are a perfect example of so-called *event-driven* systems. As the name applies, an event-driven system goes into action in response to the occurrence of some event that it is familiar with. For example, a GUI (graphical user interface) program is an event-driven system that responds to clicks of the mouse made by the user; the precise piece of code that gets executed depends on what widget (tool-bar item, button, radio dial, etc.) gets clicked. In the case of operating systems, the events that an OS responds to include system calls made by user (or even system) programs, hardware interrupts, and machine errors. Event-driven systems are typically structured as one large case-statement contained in a while-loop that "catches" the various events the system is intended to respond to. When an event is caught, the case in the case-statement corresponding to that event is executed.

This kind of event-loop structure is indeed present in operating systems. Consider, for example, how a system call gets executed in a typical OS.The calling program first pushes the parameters of the system call on the system stack. The system call number is placed in a register and a trap instruction is executed to switch from user mode to kernel mode. The kernel examines the system call number and branches to the correct system call handler, usually via a table of pointers to system call handlers indexed on the system call number. At that point, the system call handler runs and, when finished, control may be returned to the calling procedure at the instruction following the trap instruction.

Hardware interrupts are handled in a similar event-driven way by an OS. In this case, a portion of system memory is set aside for the *interrupt vector*. Using the device number of the device that caused the interrupt, the interrupt vector may be indexed into to find the address of the interrupt handler for this device.

OSP 2 is also event-driven, not surprising given that, after all, it is an operating system. However, *OSP 2* responds to *simulated* events. That is, at the core of *OSP 2* is a simulator called the *event engine* (see Figure 1.1) that semi-randomly generates events of the kinds discussed above (system calls, hardware interrupts, etc.). In response to such an event, the appropriate Java method is called. For example, suppose the event engine generates an event corresponding to an instance of the system call for opening a file. Then the method `open()` in class FileSys will be called. Moreover, if your instructor has assigned module FileSys to you as a project, then it is the code that you wrote for method `open()` that will be executed in response to the event. This is actually a somewhat simplified view of how things work in *OSP 2*. Section 1.9 explains *OSP 2* event handling in greater detail.

What this all means is that in *OSP 2*, there are no user programs per se that are being executed; all such programs are simulated by the event engine in the form of a stream of events that *OSP 2* responds to. There are several advantages to this simulation-based approach. First, events are passed through a so-called *interface layer* (IFL) of *OSP 2* that sits between the event engine and the various *OSP 2* modules in which the code for the system calls resides (see, again, Figure 1.1). The IFL therefore has the opportunity to monitor the execution of system call methods, making sure that the actions taken by these methods are semantically correct. Should an error be detected in a student implementation of a system call method, the IFL can return a meaningful error message to the student. These messages can be a great help to you in debugging your code.

The IFL performs another useful role as far as students (and instructors!) are concerned: it gathers statistics about the system's performance as the event stream is processed. Example statistics collected by the IFL include cpu utilization, number of page faults, and disk-arm movement measured in number of tracks. These statistics are very helpful in gauging the performance of your cpu scheduling algorithm, page replacement scheme, disk scheduling algorithm, etc.

Another advantage of the simulation-based approach is that to debug the OS modules that the student writes there is no need to write and run user-level test programs (as would be the case if you were working with a real OS): the simulator provides the event stream for testing. Moreover, the make-up and intensity of this event stream generated by the event engine can be adjusted dynamically by manipulating the *simulation parameters*. For example, if the instructor has assigned module FileSys as a project, he can set the simulation parameters so that the event stream will contain a high percentage of file-system related events. This yields a simple and effective way of testing the quality of student programs.

User programs are not the only thing simulated in *OSP 2*. The underlying hardware is simulated as well and includes a CPU, disk, system clock, hardware timer, and interrupt vector. The simulated hardware of *OSP 2* is described fully in Section 1.4.

***OSP 2*'s Microkernel Architecture.** An interesting topic in operating-system design is the *monolithic kernel* versus *microkernel* architecture distinction. Here the term "kernel" is used to refer to that portion of the operating system that runs in *kernel mode*: the more privileged mode of execution, as compared to *user mode*, where executing code has access to system data structures and services. Getting back to the monolithic-versus-microkernel distinction, a monolithic kernel groups together all operating-system functionality into a single process while a microkernel assigns only a few essential functions to the kernel, including address spaces, interprocess communication, and basic scheduling. The microkernel approach is also typified by well-encapsulated module boundaries for basic services with well-defined interfaces.

As depicted in Figure 1.1 and described above, *OSP 2* is hierarchically structured, consisting of three main layers: the event engine, the IFL, and the student modules. Once control enters a student module, the system can be considered to be in kernel mode. Since there are no actual user programs in *OSP 2*, replaced instead by a stochastic simulation of user threads in the form of an event stream, there is no user mode in *OSP 2* to speak of. Therefore, the file system, task-management subsystem, virtual memory-management subsystem, etc. run in kernel mode.

Because of *OSP 2*'s pure object-oriented design, all OS subsystems, including the primitive ones dealing with activities such as thread scheduling and interprocess communication, are encapsulated in modules (classes) with well-defined method interfaces. The kind of architecture adopted by *OSP 2* is sometimes referred to as modified microkernel architecture.

1.3 *OSP 2* Organization

OSP 2 comprises a number of projects that may be assigned to students as programming assignments. Each project involves the implementation of a separate Java package consisting of one or more Java classes and their associated methods. Because of their role as potential programming assignments, we shall often refer to these packages as **student packages** or **student projects**. It should be understood, however, that reference implementations of these packages are part of the standard *OSP 2* distribution and must be in place for the system to function normally (unless the reference implementation of a package has been replaced by a student implementation). Each student package is responsible for managing its own class of system resources, as described in the following:

Devices: Handles I/O requests for secondary storage devices such as disk drives.

FileSys: Implements the file system including basic file operations and directory structures.

Memory: Manages physical and virtual memory using techniques such as paging and segmentation.

Resources: Manages abstract resources of the system using deadlock detection and deadlock avoidance algorithms.

Tasks: Controls the creation and deletion of tasks, each of which is a container for a set of threads and their associated resources.

Threads: Responsible for creating, killing, dispatching, suspending, and resuming threads, the fundamental units of execution in *OSP 2*.

Ports: Implements an interprocess communication facility that allows threads to send messages to each other.

To illustrate how student projects are organized, consider the Memory module of *OSP 2*. This module corresponds to the Java package `osp.Memory` and contains the classes `PageFaultHandler`, `PageTableEntry`, and `FrameTableEntry`, among others. Each of these classes is kept in its own `.java` file: `PageFaultHandler.java`, `PageTableEntry.java`, `FrameTableEntry.java`, etc. For the Memory project, students are expected to implement the various classes associated with these files.

At the heart of *OSP 2* is the **Event Engine**, the event-based simulator that drives the execution of the student packages. The events generated by the event engine are calls to methods in student packages, representing system calls (e.g. create a task, write a file) or hardware interrupts (e.g. disk interrupt, page fault). Collectively, they simulate the behavior of a stream of executing programs in a multiprogramming operating-system environment.

There is also a layer that sits between the event engine and the student layer, the so-called **Interface Layer** or *IFL*. The IFL monitors the execution of the student packages for the purpose of catching semantic errors in student code (and subsequently producing intelligible error or warning messages), and for gathering performance statistics. Thus, the IFL can be viewed as a protective "wrapper" around the student packages. The logical structure of *OSP 2* is depicted in Figure 1.1.

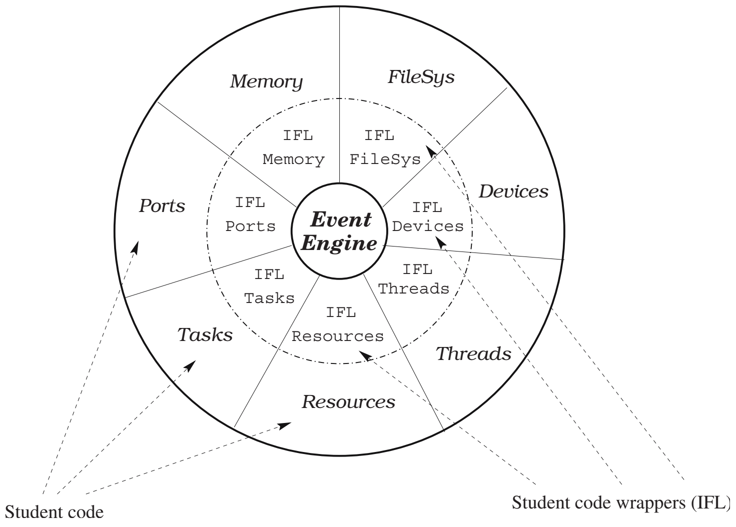


Figure 1.1 The logical structure of *OSP 2*.

1.4 Simulated Hardware in *OSP 2*

The `Hardware` and the `Interrupts` packages of *OSP 2* model the hardware-oriented aspects of the simulated multiprogramming operating system. `Hardware` consists of four Java classes, which we now describe.[1]

`CPU`: This class models the CPU of the simulated machine. It defines one method, `interrupt()`, which is used to generate an interrupt with the given type (e.g. disk interrupt, page fault). The interrupt vector supported by the `Interrupts` package is described later in this section.

[1] Note that all the methods of the `Hardware` and `Interrupts` packages are declared as `final`, meaning that they cannot be subclassed. This is done for object-oriented design reasons: you should think of these classes as "perfect" or that, conceptually, they should have no subclasses. Many of the methods contained in other *OSP 2* packages are also declared to be `final` for the same reason.

`Disk`: This class represents a hard disk attached to the system and is declared as follows:

```
public class Disk extends Device;
```

It implements methods that provide access to the physical characteristics of the disk and its current state of operation. The methods in this class are:

◇ `final public int getPlatters()`
Returns the number of platters.

◇ `final public int getTracksPerPlatter()`
Returns the number of tracks per platter.

◇ `final public int getSectorsPerTrack()`
Returns the number of sectors per track.

◇ `final public int getBytesPerSector()`
Returns the number of bytes per sector.

◇ `final public int getRevsPerTick()`
Returns the number of revolutions per tick.

◇ `final public int getSeekTimePerTrack()`
Returns the average time it takes to move the head to the adjacent track.

◇ `final public int getHeadPosition(int track)`
Returns the position of the disk head, i.e., the cylinder where the head is parked.

These methods might be used for implementing I/O schedulers; see Scheduling of Disk Requests, Chapter 6, for more information about *OSP 2* devices.

`HClock:` This class represents the hardware clock. It can be used to access the current simulation time using the following method:

◇ `public final static long get()`
Returns current simulation time.

`HTimer:` This class represents the hardware timer. If set to a positive integer, a timer interrupt will occur after that many (simulated) clock ticks. This class provides the following methods:

◇ `public final static void set(int time)`
Sets timer. Time is relative to the current time. If `time` is zero or negative, timer interrupts are disabled.

◇ `public final static long get()`
Returns time left until the timer interrupt. Returns a negative number if timer interrupts are disabled.

The `Interrupts` package of *OSP 2* consists of one Java class, which is important for several student projects.

`InterruptVector`: This class represents the hardware register called the **interrupt vector**. It contains information about the interrupt that just occurred. Interrupt handlers check the interrupt vector for the information about the interrupt so that they can properly handle the interrupt. Not all parts of the interrupt vector are relevant to every kind of interrupt. For instance, for timer interrupts, only the type of the interrupt (i.e., that it came from the timer device) is important. On the other hand, for a disk interrupt, the relevant information also includes the IORB (I/O Request Block; see Section 1.6) that caused the interrupt. For a page fault, the relevant information includes the thread and the page that caused the interrupt, etc. The student is supposed to set and query the appropriate parameters of the interrupt vector depending on the type of interrupt. The methods provided by this class are:

◇ `final static public void setInterruptType(int newInterruptType)`
Sets the type of the interrupt: `PageFault`, `DiskInterrupt`, or `Timer-Interrupt`; see `GlobalVariables` for more details.

◇ `final static public int getInterruptType()`
Returns the type of the interrupt.

◇ `final static public ThreadCB getThread()`
Returns the thread that caused the interrupt.

◇ `final static public void setThread(ThreadCB thread)`
Sets the thread that is about to cause the interrupt. In this way, other modules can query the interrupt vector to find out which thread caused the interrupt.

◇ `final static public PageTableEntry getPage()`
Returns the page that caused the interrupt (pagefault).

◇ `final static public void setPage(PageTableEntry newPage)`
Sets the page that caused the interrupt. In this way, other modules can query the interrupt vector to find out which page has cause the page fault.

◇ `final static public void setReferenceType(int referenceType)`
Sets the reference type of a memory interrupt, i.e., `MemoryRead`, `MemoryWrite`, or `MemoryLock`; see `GlobalVariables`.

- ◇ `final static public int getReferenceType()`
  Returns the type of memory reference that caused the interrupt.
- ◇ `final static public Event getEvent()`
  Returns the event that caused the interrupt.
- ◇ `final static public void setEvent(Event newEvent)`
  Sets the event that is about to cause the interrupt.

The hardware components listed above are *provided* by the *OSP 2* system and are not to be implemented by the student. In contrast, *OSP 2* also has hardware, notably the **memory management unit** (or **MMU**), that is part of a student package, module Memory. *OSP 2* memory management is discussed in Chapter 5.

1.5 Utilities

The utilities package contains a number of classes that are needed purely for simulation support. It also provides a class, `GlobalVariables`, that is required by the student packages, and several other "utility" classes that assist students in implementing their projects.

The class `GlobalVariables` comprises a number of variables that define the nature of a memory reference (e.g. `MemoryWrite`), interrupt types (e.g. `TimerInterrupt`), and method return status (e.g. `SUCCESS` and `FAILURE`). It also defines constants such as `NONE` and `SwapDeviceID`. The former represents a common return value used for integer objects (e.g. the value returned when a free frame is not found) and the latter is the device number of the swap device.

All of these constants are integers and *must* be referred to using their symbolic names. For debugging, however, it is often useful to know what the corresponding numeric values are. This is accomplished with the help of the following methods:

◇ `final static public String printableStatus(int status)`
Returns the printable representation of the following constants:

- `ThreadReady` – status of a ready-to-run thread.
- `ThreadRunning` – status of a running thread.
- `ThreadWaiting` – status of a waiting thread. (There are multiple levels of waiting, so this status is printed as `ThreadWaitingX`, where `X` is the waiting level. See Chapter 4 for details.)

– `ThreadKill` – status of a killed thread.

– `TaskLive` – status of a live task.

– `TaskTerm` – status of a killed task.

– `PortLive` – status of a live communication port.

– `PortDestroyed` – status of a destroyed communication port.

This method is useful for debugging. For instance, if you need to find out the status of a thread, you might want to display that status on the screen. But status is an integer, which does not hold much information for a human reader. The method `printableStatus()` will convert such an integer into, say, `ThreadReady` (a string).

◇ `final static public String printableRequest(int request)`
Returns human-readable representations of request constants, which are:

– `MemoryRead` – Memory read request (in `refer()`).

– `MemoryWrite` – Memory write request (in `refer()`).

– `MemoryLock` – Memory lock request (in `lock()`).

– `FileRead` – File read request (in `read()`).

– `FileWrite` – File write request (in `write()`).

◇ `final static public String printableDevice(int device)`
Returns human-readable representations for devices, which are:

– `SwapDeviceID` – the number of the swap device.

– `Disk1`, `Disk2`, `Disk3`, `Disk4` – the disk devices.

◇ `final static public String printableInterrupt(int interrupt)`
Returns human-readable representations for interrupts, which are:

– `PageFault` – Pagefault interrupt.

– `DiskInterrupt` – Disk interrupt.

– `TimerInterrupt` – Timer interrupt.

◇ `final static public String printableRetCode(int retcode)`
Returns human-readable representations of method return-codes. The supported return-codes are:

– `SUCCESS` – successful completion.

– `FAILURE` – unsuccessful completion.

– `NotEnoughMemory` – returned by the page-fault handler when it cannot find a frame to satisfy a page fault.

◇ `static public String userOption`
This variable is set using the command line option `-userOption`. It can be used to pass a parameter to the student program when *OSP 2* is invoked from command line. This variable is not used internally by the simulator and its use is solely up to the student's discretion.

Other useful classes in the `Utilities` package include:

`MyOut`: The methods in this class can be used to insert messages into the *OSP 2* **system log** for debugging purposes. The system log tracks system events as they occur and messages inserted into the log by students are inserted in chronological order with other system events. The following methods are provided:

◇ `final public synchronized static void print(Object where, String msg)`
Prints a message to the system log. The argument `where` must be an object from which the package and the class from where `print` is called can be derived. If `print()` is called from a non-static method, then the `where` argument should be *this* (the Java keyword that denotes the context object); otherwise, if `print()` is invoked from within a static method, then the `where` argument should be a string-object of the form `"osp.packageName.className"`. For instance,

```
MyOut.print("osp.Tasks.TaskCB", "Hello World!");
```

◇ `final public synchronized static void error(Object where, String msg)`
Prints an error message to the system log and terminates *OSP 2*. The format of the `where` argument is the same as before. This method can be used to halt execution of *OSP 2* when a bug is discovered; further execution of *OSP 2* under these circumstance is probably not useful under the circumstances. The `error()` method also causes a stack trace and the current *OSP 2* snapshot to be included in the log for debugging purposes.

◇ `final public synchronized static void checkCondition(boolean condition, Object where, String msg)`
Similar to `error()` except that the error message is printed and *OSP 2* is terminated only if the boolean `condition` is `false`.

◇ `final public synchronized static void warning(Object where, String msg)`

Similar to `print()` except that a *warning* message is printed to the log. Unlike `error()` and `checkCondition()` (but like `print()`), the execution of *OSP 2* can proceed after this method is called. Like method `error()`, a snapshot and a stack trace are included in the system log. This method can be used by the student to check conditions that are not necessarily fatal to the execution, but are still undesirable and must be fixed.

◇ `final public synchronized static void snapshot()`
Although `error()`, `warning()`, and `checkCondition()` can be used to obtain the current *OSP 2* snapshot, the `snapshot()` method can be used to insert a snapshot into the system log at any time, not necessarily when a warning or an error condition is detected.

`GenericList:` This class provides the following methods for maintaining doubly linked lists of objects:

◇ `public GenericList() implements GenericQueueInterface`
A constructor that creates an empty list.

◇ `public GenericList(Object obj)`
A constructor that creates a list and initializes it with a given object.

◇ `public final int length()`
Returns the length of the list.

◇ `public final boolean isEmpty()`
Returns true if the list is empty, false otherwise.

◇ `public final synchronized void insert(Object obj)`
Inserts an object at the beginning of the list.

◇ `public final synchronized void append(Object obj)`
Appends an object to the end of the list.

◇ `public final synchronized Object remove(Object obj)`
Removes the specified object from the list and returns the object. Null, if the object is not found.

◇ `public final synchronized Object appendToCurrent(Object obj)`
Inserts the object `obj` into the list after the current item in the list. The current item is set by the enumerators (see below) as they traverse the list (after each call to `nextElement()`).

◇ `public final synchronized Object prependAtCurrent(Object obj)`
Inserts the object `obj` into the list before the current item in the list. The current item is set by the enumerators (see below) as they traverse the list (after each call to `nextElement()`).

◇ `public final synchronized boolean contains(Object obj)`
Returns true if the specified object is in the list, false otherwise.

◇ `public final synchronized Object removeHead()`
Removes the object at the head of the list and returns the object. Null, if the list is empty.

◇ `public final synchronized Object removeTail()`
Removes the object at the tail of the list and returns the object. Null, if the list is empty.

◇ `public final synchronized Object getHead()`
Returns the object at the head of the list without removing the object.

◇ `public final synchronized Object getTail()`
Returns the object at the tail of the list without removing the object.

◇ `public final synchronized Enumeration forwardIterator()`
An iterator is a general Java mechanism for dealing with collections such as sets and lists. A forward iterator returns an object of class `Enumeration` (a standard Java class), which can then be used to conveniently traverse the list. For instance,

```
GenericList list;
.....
Enumeration enum = list.forwardIterator();
while(enum.hasMoreElements()) {
   Object obj = enum.nextElement();
}
```

Each call to `nextElement()` advances the current pointer in the list. The current pointer is the point of insertion for the previously described methods `appendToCurrent()` and `prependAtCurrent()`.

◇ `public final synchronized Enumeration forwardIterator(Object first)`
Works like `forwardIterator()` but starts the iteration from the first occurrence of the specified object in the list.

◇ `public final synchronized Enumeration backwardIterator()`
Similar to `forwardIterator()` but traverses the list backwards.

◇ `public final synchronized Enumeration backwardIterator(Object first)`
Like `forwardIterator(Object first)` but traverses the list backwards.

`GenericQueueInterface`: The `GenericQueueInterface` that `GenericList` implements contains the following methods:

◇ `public int length();`
Returns the number of elements in the queue.

◇ `public boolean isEmpty();`
Returns true if the queue is empty, false otherwise.

◇ `public boolean contains(Object obj);`
Returns true if the queue contains object `obj`, false otherwise.

This interface mandates only the methods that *OSP 2* itself uses internally. For classes that use this interface you might need to define additional methods, such as insertion into the queue and deletion of queue members.

1.6 *OSP 2* Events

Like any other operating system, *OSP 2* is *event-driven.* When a thread executes an I/O operation, it blocks until the I/O completes. When one threads needs to communicate with another, it sends a message and might decide to block itself until a response arrives. When a thread blocks, we say that it is waiting for an **event** to occur (like the completion of an I/O operation or message delivery) so that the thread may continue its execution.

In a typical operating system, events are represented by some kind of **event** data structure. A thread that wishes to block itself, or, more generally, to be notified about the completion of an event, executes a `suspend()` operation on that event, which places the thread on the event's **waiting queue**. The event "happens" when some other thread (a user or a system thread, depending on the type of the event) announces that the event has taken place. For example, in the case of an I/O operation, a disk interrupt will cause the disk-interrupt handler to execute and the handler eventually will announce the completion of the I/O event. In *OSP 2*, an event is an object and such an announcement is made by executing the `notifyThreads()` method associated with the event. As a result, threads waiting on the event are unblocked by the operating system and can continue their execution.

In *OSP 2*, events are represented by the `Event` class. A basic event has an id, which serves to distinguish this event from other events and a **waiting queue**. Thus, an event provides the means for suspending threads when they have to wait, and subsequently locating them when they are to be resumed.

In practice, the `Event` class is almost always subclassed before it is used. This is because threads are usually interested in very specific kinds of events rather than just generic events. For example, a thread is suspended because it has to wait for an I/O operation to complete or a page to be swapped in, or because it is suspended on a communication port until a message arrives. Thus, *OSP 2* treats memory pages, I/O request blocks (IORBs), and communication ports as events in the sense that all these classes extend the class `Event`.

The `Event` class provides the methods necessary for maintaining the waiting queue, and these methods can be used on pages, ports, and IORBs when these are used in their capacity as events. The methods provided by class `Event` are as follows:

◇ `public void addThread(ThreadCB thread)`
Add the specified thread to the waiting queue of the event. No checks are performed to ensure that the thread is not already on the queue.

◇ `public void removeThread(ThreadCB thread)`
Remove the specified thread from the queue. If the thread is not found, return silently.

◇ `public boolean contains(ThreadCB thread)`
Return true if the thread is on the waiting queue for this event, false otherwise.

◇ `public int getNumberOfThreadsWaiting()`
Returns the length of the waiting queue.

◇ `public GenericList getThreadList()`
Returns the waiting queue itself.

◇ `public ThreadCB getHead()`
Returns the thread at the head of the waiting queue or the null object.

◇ `public void notifyThreads()`
Resumes all threads on the waiting queue (i.e., executes `resume()` on each one of them) and empties the queue. It is quite possible that some threads on the waiting queue have been destroyed while waiting. In this case, `notifyThreads()` simply removes the destroyed threads from the queue as executing `resume()` on such a thread would be an error.

Several projects in *OSP 2* make extensive use of events and we will refer back to this section when necessary.

1.7 *OSP 2* Daemons

The implementation of certain functions of an OS can be facilitated through the use of daemons: special system threads that run periodically and perform "work" specified by the user. In *OSP 2*, such work might include proactive swapping out of dirty memory pages, as required by some memory-management algorithms, and deadlock detection.

Daemon support in *OSP 2* is provided by the `Daemon` class and the interface `DaemonInterface`. To use a daemon, one creates an object in a class that implements `DaemonInterface` and then registers this object with the system. The following statements declare a class of daemons whose only job is to insert a notice in the system log:

```
class MyDaemon implements DaemonInterface
{
    public void unleash(ThreadCB thread)
    {
        MyOut.print(this, "My daemon executed at time: "
                              + HClock.get());
    }
}
```

The only mandatory method in this class is `unleash`, which should contain the code you want the daemon to execute. For instance, in case of a deadlock-detection daemon, a method should be provided that executes the appropriate deadlock-detection algorithm. This method is called by *OSP 2* when it wakes up the daemon.

Defining a daemon is your responsibility. You also need to register it with the system and provide three things: the name of the daemon (for easy identification of the daemon in a system trace), a concrete daemon object to call, and the amount of time that should pass between invocations of the daemon. This is typically done when *OSP 2* begins executing, inside the `init()` method that exists in the main class of each student package. Here is an example of registering a daemon:

```
Daemon.create("My own daemon", new MyDaemon(), 20000);
```

The first argument can be an arbitrary string. The second is an object of the daemon class defined earlier. The third argument tells *OSP 2* that the daemon should be periodically woken up after every 20,000 ticks.[2] You can create several

[2] *OSP 2* does not guarantee that it will wake up the daemon exactly after the specified number of ticks, but it will try to wake it up as soon as possible after the specified interval.

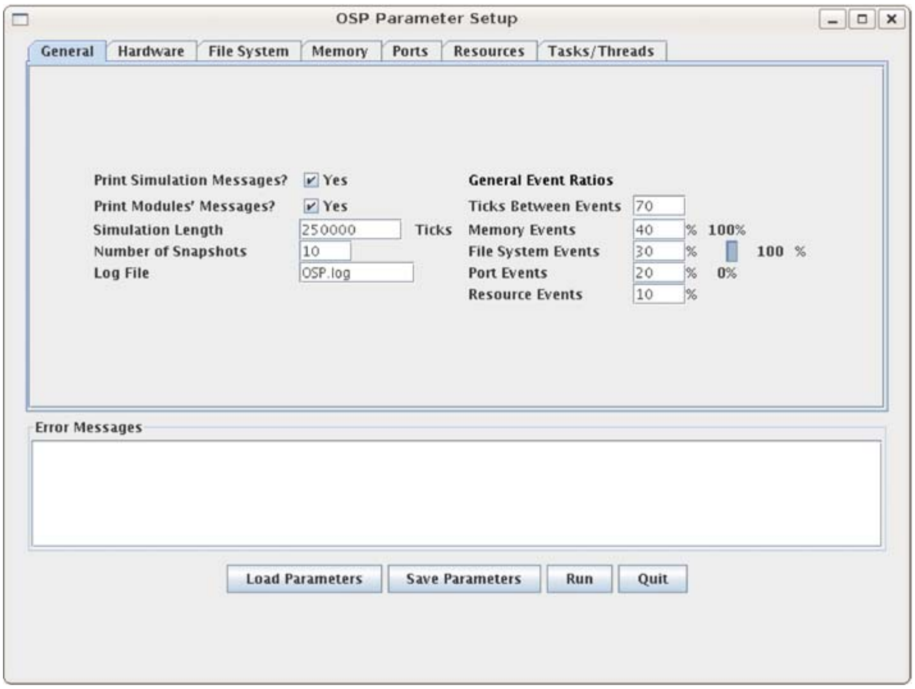
daemons if several periodic jobs need to be performed by the module that you are implementing. Typically the requirement to use daemons would be part of the assignment given out by your instructor, but you might also decide to use them on your own, based on your understanding of the problem.

1.8 Compiling and Running Projects

A student project assignment consists of several files:

1. `Demo.jar`, which contains a demo version of *OSP 2*. It can be used to get a general idea of how *OSP 2* works, to familiarize yourself with the graphical interface and command-line options of the system, and to create configuration files for running *OSP 2* with different parameters.
2. Template files, each of which contains the necessary import statements, the class header of the public class to be implemented, and the headers of the public methods that must be implemented by the student. For instance, for the Threads project, the template files would be
   a) `ThreadCB.java`
   b) `TimerInterruptHandler.java`
3. `OSP.jar`, which contains the compiled classes of the *OSP 2* simulator that drive the execution of the classes in the student project. When your implementation of the classes in the project is complete, they should be compiled and linked with the `OSP.jar` file.
4. A `Makefile` that simplifies the compilation process under Unix-based systems (Solaris, Linux, Free BSD, etc.).
5. The `Misc` subdirectory, which includes two files:
   a) `params.osp`
   b) `wgui.rdl`

   The first file contains the parameters that will drive the simulation and the second file is a configuration file for the GUI. You should not edit either of these files manually. In fact, there is no reason to touch `wgui.rdl` at all, unless you are not satisfied with the overall look of the graphical interface `:-)`. However, you might want to run *OSP 2* with different parameters and create a new configuration file derived from `params.osp`. The only recommended way of doing this is to change the parameters through the GUI of the demo version of *OSP 2* and then save the new parameters in a new file. A GUI panel that lets the user change the simulation parameters is shown in Figure 1.2.



Print Simulation Messages? ✓ Yes
Print Modules' Messages? ✓ Yes
Simulation Length 250000 Ticks
Number of Snapshots 10
Log File OSP.log

General Event Ratios
Ticks Between Events 70
Memory Events 40 % 100%
File System Events 30 % 100 %
Port Events 20 % 0%
Resource Events 10 %

Figure 1.2 Panel for changing $\mathcal{OSP}$ 2 simulation parameters.

Java settings. $\mathcal{OSP}$ 2 requires JDK 1.5 or a later version. To run and compile $\mathcal{OSP}$ 2 you must first make sure that Java is properly set up on your machine and that your personal configuration files are set appropriately. This simply means that the environment variable `PATH` is set appropriately. For Windows, this variable should be set in the `autoexec.bat` file or through the control panel as follows:

```
set PATH=%PATH%;C:\jdk\bin
```

The second component in this setting should, of course, point to the place where the Java executables are installed and our choice of `C:\jdk\bin` is merely an example.

For Unix-based systems, the setting depends on the type of the shell used. We show the settings for the two most popular shells: `bash` and `csh`. Settings for other shells (such as `ksh`, `sh`, `tcsh`) would be similar to either `bash` or `csh`, the only difference being the name of the configuration file.

To set the `PATH` variable for `bash`, place the following in the `.bashrc` file in your home directory:

```
PATH=/usr/local/bin/jdk:$PATHexport PATH
```

As before, `/usr/local/jdk/bin` is just an example. The actual location of the Java executables can vary.

For `csh`, the `PATH` variable should be set in the file `.cshrc` in your home directory:

```
setenv PATH /usr/local/bin/jdk:$PATH
```

Running the demo program. To run the demo version of *OSP 2*, simply type:

```
java -classpath .:Demo.jar osp.OSP
```

(use `.;Demo.jar` on Windows).

Some installations of JDK might require that you set the `CLASSPATH` environment variable (this requirement would then be part of the Java installation instructions). In this case, you might need to run *OSP 2* as follows:

```
java -classpath .:Demo.jar:${CLASSPATH} osp.OSP
```

for Unix-based systems and

```
java -classpath .;Demo.jar;%CLASSPATH% osp.OSP
```

for Windows.

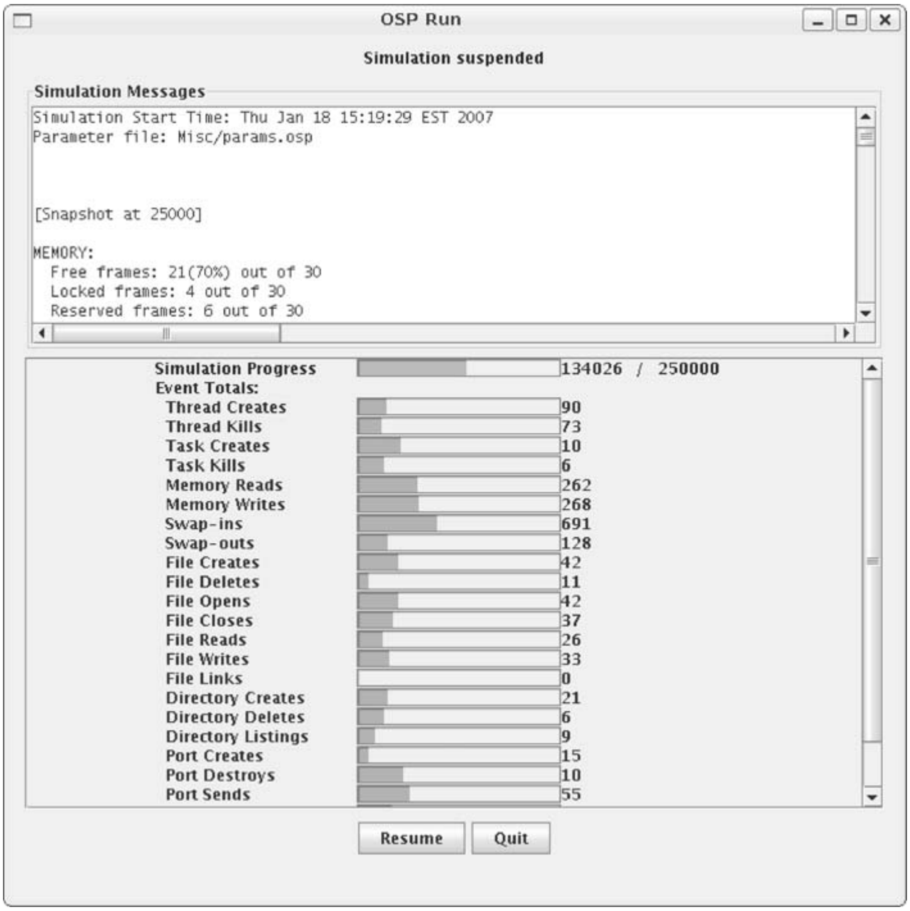
Compiling and running the project. Once your implementation of the project is finished, you are ready to compile and run the system. Here is how to do this.

On Unix-based systems, simply type `make`, and the project will be compiled. To run it without the GUI, type `make run`; with the GUI, type `make gui`; and to run with the debugger type `make debug`. Sometimes `make clean; make` can be helpful if you need to get rid of stale `.class` files and force recompilation of the entire project. That's all! The only caveat is that this must be a version of *GNU make*, which is available on most Unix-based systems, albeit sometimes under different names, such as `gnumake` or `gmake`. To find out it your make-program is a GNU make, type

```
make --version
```

If it does not say that this is GNU make or if it does not understand the `--version` argument, then it is *not* GNU make, and you should ask the system administrator if this version of the make-program is installed (and under which name). If you cannot locate the appropriate make-program, read on.

Figure 1.3 shows what you can expect when running *OSP 2* with a GUI and Figure 1.4 shows a run without the GUI.



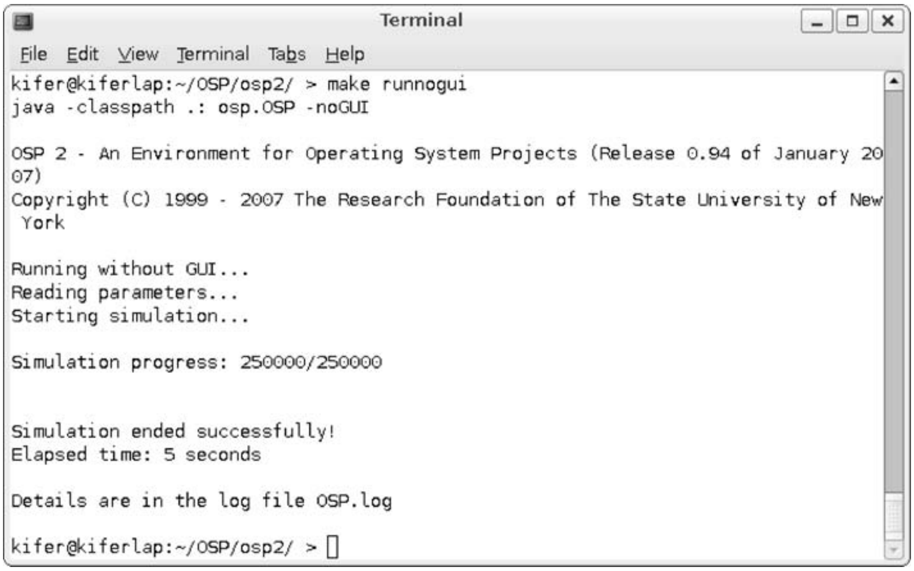
```
Simulation Start Time: Thu Jan 18 15:19:29 EST 2007
Parameter file: Misc/params.osp



[Snapshot at 25000]

MEMORY:
  Free frames: 21(70%) out of 30
  Locked frames: 4 out of 30
  Reserved frames: 6 out of 30
```

Simulation Progress	134026 / 250000
Event Totals:	
Thread Creates	90
Thread Kills	73
Task Creates	10
Task Kills	6
Memory Reads	262
Memory Writes	268
Swap-ins	691
Swap-outs	128
File Creates	42
File Deletes	11
File Opens	42
File Closes	37
File Reads	26
File Writes	33
File Links	0
Directory Creates	21
Directory Deletes	6
Directory Listings	9
Port Creates	15
Port Destroys	10
Port Sends	55

Figure 1.3 An *OSP 2* run with a graphical interface.

The following commands can be used to compile and run *OSP 2* on a Unix-based system:

```
javac -g -classpath .:OSP.jar: -d . *.java
java -classpath .:OSP.jar:. osp.OSP
jdb -classpath .:OSP.jar:. osp.OSP
```



```
kifer@kiferlap:~/OSP/osp2/ > make runnogui
java -classpath .: osp.OSP -noGUI

OSP 2 - An Environment for Operating System Projects (Release 0.94 of January 20
07)
Copyright (C) 1999 - 2007 The Research Foundation of The State University of New
 York

Running without GUI...
Reading parameters...
Starting simulation...

Simulation progress: 250000/250000


Simulation ended successfully!
Elapsed time: 5 seconds

Details are in the log file OSP.log

kifer@kiferlap:~/OSP/osp2/ > 
```

Figure 1.4 An *OSP 2* run without the GUI.

The only difference under Windows is that one has to replace ":" with ";". The first command compiles the project, the second runs it, and the third runs it under the Java debugger.[3] Running *OSP 2* with the Java debugger can be excruciatingly slow, so you should try this only if you need to trace the execution of your program or examine it in some other way that the debugger provides.

Again, some installations of JDK might insists that you set the `CLASSPATH` environment variable and attach it to the `-classpath` argument as explained earlier.

***OSP 2* command-line options.** You can run *OSP 2* with certain command-line options. Here is the full list of options:

```
      -help - lists all command-line options
     -noGUI - runs the simulator without the GUI
 -paramFile - use the next argument as the parameter file
   -guiFile - use the next argument as GUI configuration file
```

[3] Some Java distributions for Linux have problems with running the debugger due to broken shell scripts. When run, the debugger will complain that it cannot load certain libraries. To fix this, you must set the environment variable `LD_LIBRARY_PATH` to something like `/usr/local/jdk/lib/i386:$LD_LIBRARY_PATH`. You might have to do some experimentation to find out the exact path.

```
  -userOption - use the next argument to set the global
                variable userOption
     -debugOn - includes debugging messages in the OSP system log
```

Among these, only `-userOption`, `-noGUI`, and `-paramFile` are useful for student projects. The first option, `-userOption`, allows you to pass a string argument to the student program from the command line. As a result, the string argument specified on the command line becomes the value of the global variable `userOption`. This can be used, for example, when experimenting with different project implementations, based, perhaps, on different algorithms, and a command-line option is needed to indicate which algorithm to execute. This option can also be used to invoke debugging code that is normally hidden. The second option, `-noGUI`, runs *OSP 2* without the GUI, which saves time. *OSP 2*'s GUI is very useful as a tool for setting the simulation parameters, but apart from that it is just a very fancy progress bar and, as such, is intended to distract serious people from doing work.

The second useful option, `-paramFile`, can be used to run *OSP 2* with alternative parameter files, which can be helpful for debugging. The use of `-debugOn` option is not recommended for student projects. It is mainly a tool for debugging *OSP 2* itself, and the messages it produces can be confusing to someone who is not familiar with the source code of the system. Apart from that, with this option turned on, the OSP system log can be in excess of 30M, which might be a problem on shared file systems.

Here is an example of how to specify command-line arguments to the `make` command under Unix:

```
make run OPTS="-paramFile my-other-param-file.osp -noGUI"
```

For Windows and for those Unix users who do not trust makefiles, the same effect can be achieved as follows:

```
java -classpath .:OSP.jar osp.OSP
     -paramFile my-other-param-file -noGUI
```

(This command should be typed on one line.)

1.9 General Rules of Engagement

This section describes important general conventions about writing code for student projects.

1.9.1 A Day in the Life of an *OSP 2* Thread

A key concept in *OSP 2* is the *thread*, the schedulable and dispatchable unit of execution in *OSP 2*. Threads are simulated in *OSP 2* by the event engine. That is, the event engine, using a certain "stochastic model", semi-randomly decides how many threads to create, how long each of them will live, and what services they will request of the OS while they are alive. A request-of-service by a thread is represented in the event engine by an event corresponding to a call to a method in one of the *OSP 2* modules. For example, if *OSP 2* wants to simulate a request by a thread to read a certain file, an event is created that will eventually result in a call to method `do_read()` of class `OpenFile` of package FileSys.

A key point that we will emphasize numerous times in this book is that the threads that *OSP 2* simulates represent the behavior of user programs or applications. In a typical computing environment, an application program performs some useful work for a user. To do so, the application requests services from the operating system, such as memory allocation, the use of the CPU, management of files, etc. The user sees the results of the work performed by the application, but the details of how the services are implemented by the operating system are normally hidden from the user.

In *OSP 2* you have to take the opposite view: your concern is the operating system itself, and the user applications are faceless programs that you know nothing about. The only time you become aware of these programs is when they—or more precisely the simulated *OSP 2* threads representing the behavior of these programs—request services from you, the operating system. The aim of each of student project is to implement a well-defined service, such as memory or thread management, that might be requested by a typical application. When a simulated *OSP 2* thread requests a service from the OS, it suddenly becomes "real": a call is made to one of the methods in your project and the simulated computation becomes live computation of one of the methods that you implemented.

OSP 2 has a modular, object-oriented design with clear interfaces. Every student project implements a particular service. The implementation of the classes needed to complete each project is under the student's control. For each class, the student is required to implement certain methods and in doing so can augment the class with any number of auxiliary methods or variables. The student is also provided with a set of methods to operate on the "built-in" data structures of the class (which are represented as private fields in the IFL layer). In some cases, it becomes necessary to obtain services from other parts of the system, which is also done through the published interfaces.

It is important to keep in mind that if you are assigned, say, the memory-management project, MEMORY, then you are responsible for implementing all the necessary functionality as defined by the project description. *OSP 2* will not attempt to provide any memory-related service, leaving everything to you. However, like a Big Brother, it is watching and is very keen on reporting errors.

When implementing a project, only the interfaces described in that project's description can be used. Method calls and classes that you might find in the description of other projects will not work and are likely to result in a compilation error. This is the result of the method-name obfuscation described in Section 1.9.4, which is performed to prevent corruption of the internal system state.

1.9.2 Convention for Calling Student Methods

One of the most important tasks of the *OSP 2* simulator is to verify the actions performed by student code for semantic correctness and to provide meaningful error messages and warnings. This error checking is performed by the **interface layer** of *OSP 2* (or IFL). The IFL contains wrapper methods that validate the state of the system before and after student code is executed. Because of these wrappers, a special convention for naming and invoking methods must be followed when implementing an *OSP 2* project. To make the discussion concrete, consider the THREADS package, which is responsible for thread-management tasks such as thread creation. There is both a Java class for threads in the IFL, called `IflThreadCB` (the `CB` stands for "control block"), and a Java class for threads in the student package, simply named `ThreadCB` (i.e. without the `Ifl` prefix). Moreover, `ThreadCB` is a *subclass* of `IflThreadCB` and both of these classes implement methods for thread creation (among others), with the IFL method serving as a protective "wrapper" for the student-layer method.

To distinguish these thread-creation methods, the one defined in the superclass is simply called `create()`, while the one in the subclass is called `do_create()`, i.e. the corresponding method name in the student package is prepended with the prefix `do_`. In general, we have the following naming convention.

> *Methods in the OSP 2 API that are to be implemented by the student have the naming schema* `do_name`*, where* `name` *is the name of the wrapper in the IFL.*

There is an exception to this rule, namely the methods `atError()` and `atWarning()`, which are introduced below.

This convention has several ramifications that the student must be aware of when implementing a project. These are best understood by considering the flow of execution in *OSP 2* when an event is generated by the event engine and subsequently "handled" by the appropriate classes in the IFL layer and student package. Five main points of control can be identified within this execution flow; see also Figure 1.5.

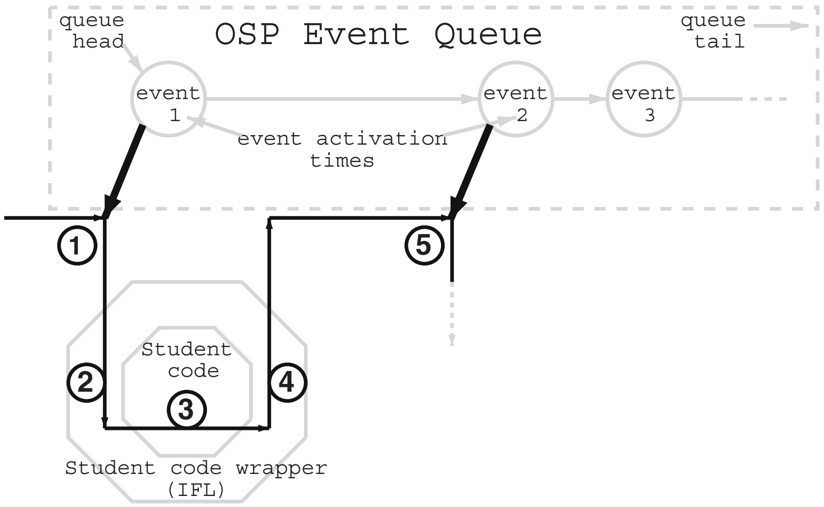


Figure 1.5 Execution flow for handling an event.

1. The event engine selects the event at the head of the "event queue" for execution. The event is actually a call to one of the student methods although control must go through the IFL. Assume, for the sake of discussion, that the selected event is a call to the `create()` thread method. In this case, the event engine calls `create()` in the IFL.
2. The IFL performs some bookkeeping for the purpose of detecting possible errors in, and for monitoring the performance of, the student's implementation of `create()` and then calls `do_create()` in the student layer.
3. The student implementation of `do_create()` performs the requested action.
4. Control returns to `create()`, which verifies that the actions taken by the student code were correct. If the student code executed incorrectly, an error message is written to the simulation log and simulation halts.
5. Assuming the student code executed correctly, simulation proceeds to the next scheduled event on the event queue.

Students should therefore adhere to the following additional naming convention:

> *When* calling *a method named* `name` *in this or another package, call the method* `name`*, i.e. without the "*`do_`*" prefix.*

In contrast, as noted above, when *implementing* a method named `name`, students will actually implement the method `do_name`.

Note also that *the student implementation should never directly refer to the classes defined in the IFL layer.* For instance, even though the method `dispatch()` is defined in the class `IflThreadCB`, it is inherited by `ThreadCB` and it should be called as `ThreadCB.dispatch()` rather than `IflThreadCB.dispatch()`.

1.9.3 Static vs. Instance Methods

When you receive a project assignment that contains the templates of the methods to be implemented, you will notice that some methods are static (i.e., they apply to class objects) and some methods (those that do not have the `static` keyword attached) work on instance objects.

This division of the project methods into static and instance methods comes from the differences in their function. For example, the method `do_dispatch()` is static in class `ThreadCB`, because it makes no sense to call it on any particular thread: the thread to be dispatched is not known at the time of the call and, in fact, it is the job of the `do_dispatch()` method to find such a thread and give it control of the CPU, as described in Chapter 4.

On the other hand, methods `do_resume()` and `do_suspend()` in `ThreadCB` are *not* static: they are called on specific thread objects, because the job of these methods is to resume or suspend the threads on which the methods are called. As usual in Java, the context object of a non-static method is accessible through the variable `this`.

Therefore, when reading the description of each method in the project, it is important to be aware of whether this method is static or an instance method.

1.9.4 Obfuscation of Method and Class Names

Chapters 3 through 8 describe the classes and methods that comprise the various student projects. For each project, the student implementation may require services implemented in other parts of *OSP 2* and must call the appropriate methods to obtain these services. Methods needed for one project, however, are not necessarily needed for another. In some cases, incorrect use of methods that belong to other packages might even corrupt the internal state of the system.

To prevent the student implementation from incorrectly using public methods that are not required for the given project, the names of these methods are changed in that project by a special “obfuscater” program. For example, the method `isFree()` of class `FrameTableEntry` is available in project MEMORY, but it is obfuscated away and will cause a compilation error if it is used by methods in project THREADS.

1.9.5 Possible Hanging After Errors

When *OSP 2* detects an error in a student program, it prints information about the error and then tries to terminate. Graceful termination, however, is not always possible because *OSP 2* is a multi-threaded application and termination of some of the active threads might depend on student code (whose behavior cannot be predicted). It is therefore possible that, after printing an error message, *OSP 2* may hang; in this case the system must be terminated by the user.

1.9.6 Possible Exceptions After the End of Execution

On very rare occasions you might see exceptions that occur after the end of a run of the simulator. This is nothing to worry about, however, as it does *not* indicate a problem with your program. The reason for these exceptions is that when the designated simulation time runs out, *OSP 2* tries hard to stop all the currently active Java threads. Unfortunately, it is not possible to terminate threads immediately, so a thread may continue to run for a short while even though some of the vital system objects may have already been destroyed. In such situations, `NullPointerException` and other exceptions can occur.

1.9.7 General Advice: How to Figure it Out

When you begin an *OSP 2* project, it is important to understand the specifications of the various student projects contained in the following chapters, and how your implementation fits into the big picture. Perhaps, it is best to state what this manual is *not*:

1. It is *not* intended to replace the textbook.
2. It is *not* intended to teach you the basic concepts in operating systems.
3. It is *not* intended to guide you every step of the way to the completion of

your project.

Instead, the description of a student project provides a complete description of the API that you can use to implement the project and a description of the functionality of each method in the project. The manual does not explain how to put the pieces of the puzzle together—this is for you to figure out based on your understanding of the subject.

The best advice is: think logically. In these projects *you are implementing parts of an operating system*, which is probably very different from the kind of programming you have done in the past. If you are in doubt about whether or not it is appropriate for your implementation to take a certain action, consider whether you would like it if the OS on your desktop behaved this way. For example, suppose you are implementing a thread scheduler and at certain point in the program you have to deal with the situation where no threads are left to schedule. Should you leave the CPU idle or create and run a dummy thread, thereby wasting computing resources? The answer should become obvious if you just ask yourself the simple question: "Would I want my home computer to behave this way?"

1.10 System Log, Snapshots, and Statistics

During a run, $\mathcal{OSP}$ 2 prints messages in the system log. Each message describes a specific event that occurred during execution. Messages that come from the simulator are prefixed with `Sim:`; those that come from student packages other than the project-assignment module(s) are prefixed with `Mod:`; and those that come from the project you are currently implementing are prefixed with `My:`.

Periodically $\mathcal{OSP}$ 2 dumps **snapshots** of the system state into the log file. These snapshots are primarily intended for performance checking and debugging. A snapshot contains a complete dump of main memory, the status of all page tables, the status of all threads, including the queues they are in, and the status of all communication ports.

In addition, the snapshot provides statistics such as CPU utilization, **average service time** (also known as **average turnaround time**) of an I/O request and a thread, the average number of tracks swept on each device per I/O request, and the **average normalized service time**. The last of these describes the average relative delay suffered by each thread, and is determined by the following formula:

$$\frac{\Sigma_i \dfrac{\text{CPU time used(thread}_i\text{)}}{\text{turnaround time(thread}_i\text{)}}}{\text{total number of threads}}$$

where the sum is over all threads (dead or alive). This is a better measure of performance than the average turnaround time, and this statistic should be kept as high as possible (but, of course, it cannot exceed 1).

It should be noted that some entries in the system log can have fairly long lines, so to view the log it may be necessary to use a viewer with horizontal scroll capability. For example, if you are running *OSP 2* with a parameter file that specifies long page tables (say, more than 64 pages), then you can expect to need to use a scrollable viewer. Most text editors provide this capability.

1.11 Debugging

There is no special-purpose debugger for *OSP 2*, but there are a few things that can help. Generally, errors in student code can be divided in two categories:

1. Errors that cause Java exceptions.
2. Semantic errors, such as an incorrect action taken in response to a simulator request. Examples include the failure to maintain the correct status of a thread (e.g., `ThreadWaiting` instead of `ThreadRunning`) or replacing a dirty page without first swapping it out to the swap device.

Errors of the first kind are much easier to fix since they can be identified with the help of a Java debugger, such as `jdb`. For example, a Java debugger can be used to determine where the exception `NullPointerException` has occurred. In all likelihood, Java exceptions are due to errors in student code. If an exception takes place in *OSP 2* code, it does not necessarily mean that the student code is correct; rather, it likely means that *OSP 2* has failed to catch the problem early enough to generate a meaningful error message to guide you to the real problem.

Apart from tracking down exceptions, Java debuggers are not very useful for debugging *OSP 2* projects, especially for finding semantic mistakes in student code. This is because such an error might be detected by *OSP 2* thousands of instructions after the erroneous action was performed by the student program and using the debugger trace facility to track down the source of the error might wear you down before the first signs of a problem begin to show up. Therefore, the following procedure is recommended for finding and fixing semantic problems.

System log. When *OSP 2* detects a semantic error, it tries to come up with as clear an explanation as possible. When an error or a warning is issued, the log file (`OSP.log`, unless specified differently in the configuration file) will contain a message of the form `<<Error>>`, `<<Assertion>>`, or `<<Warning>>`, which are easy to find with an editor. When *OSP 2* terminates, it tells you if one of these conditions was encountered or if it terminated successfully.

In case of a problem, the best way to understand what might have happened is to trace back the messages in the system log. For instance, if an error message says that you are trying to dispatch a thread that is waiting on some event that has not occurred yet, you should trace back and see when the thread was suspended on that event and what was the sequence of events that happened since. You might discover, for example, that your program is placing threads on the ready queue that, in reality, are not ready to execute. Likewise, if *OSP 2* complains that there is a discrepancy between what it perceives to be the dirty/clean status of a page and the value of the dirty bit set for this page by the student program, tracing the system log might reveal that, say, this page has just been swapped in but your program did not reset the dirty bit to *false*.

OSP 2 generates a log by default, unless tracing is turned off. However, the log messages thus generated are typically not sufficient by themselves to debug errors in your code. This is because *OSP 2* cannot know what is actually happening inside student code and it is therefore necessary to put the execution of your program in the context of the overall execution of *OSP 2*. This can be achieved with the help of the methods in the class `MyOut`, which were discussed earlier. Moreover, it is useful to keep in mind that the `toString()` method of all major classes in *OSP 2* is set up in a printer-friendly manner. For example, executing

```
MyOut.print(this,
            "The " + thread + " is suspended on " + event);
```

where `thread` is an object of class `ThreadCB` and `event` is an object of class `Event` will produce output that looks like this:

```
My:  2534.5533 [Threads.ThreadCB] The Thread(15:32/RU)
     is suspended on Event(3)
```

Thus, no special care is needed to produce a readable representation of *OSP 2* objects. The header of the *OSP 2* system log provides a brief explanation of the printable representation of various objects. For instance, in the above representation for a thread, `Thread(15:32/RU)`, the first number (15) is the thread id, the second (32) is the Id of the task the thread belongs to, and `RU` is the code that represents the current status of the thread (`ThreadRunning` in this case).

Error and warning hooks. In addition to `MyOut`, the main class of every student project has the following pair of methods:

◇ `public static void atError()`

◇ `public static void atWarning()`

The first method is called when an error or a condition violation is detected by *OSP 2*, and the second is called right after *OSP 2* issues a warning message. Normally, the bodies of these methods are empty, *and this is how you should leave them when you submit your program.* However, during debugging you can put arbitrary code there. Most useful would be code that prints the status of the relevant variables in your program. Note that whenever a condition violation, error, or warning occurs, *OSP 2* prints the full stack trace that indicates the sequence of method calls that led to the problem.

System snapshot. *OSP 2* also produces a **system snapshot** when a condition violation or error occurs. The snapshot conveys the status of memory allocation, the status of each task and thread in the system, etc. This information can be compared with the status of the system per your implementation and the system log can be consulted to determine where the discrepancy arises. When *OSP 2* prints out a warning, no snapshot is added to the log by default. This is because warnings tend to come in large numbers and this can lead to an unmanageably large number of snapshots in the log. However, you can include the `snapshot()` method of class `MyOut` in the body of the `atWarning()` method of the main class of your project and produce a snapshot in this way. (It is recommended to print a snapshot only on the first warning, since subsequent snapshots are not likely to shed any more light on the problem.)

Execution stack trace. Another important resource for debugging *OSP 2* projects is the execution stack trace provided by the Java virtual machine when a Java exception occurs. Here is an example of such a trace:

```
java.lang.NullPointerException
  at osp.Threads.ThreadCB.do_kill(ThreadCB.java:195)
  at osp.IFLModules.IflThreadCB.kill(IflThreadCB.java:634)
  at osp.IFLModules.IflThreadCB.killOldThreads(IflThreadCB.java,
     Compiled Code)
  at osp.IFLModules.CallbackThreadKill.voidCallback(IflThreadCB.java,
     Compiled Code)
  at osp.EventEngine.EventCallback.Activate(EventCallback.java,
     Compiled Code)
  at osp.EventEngine.EventEngObj.ActivateChildren(EventEngObj.java,
     Compiled Code)
  at osp.EventEngine.EventEngObj.Activate(EventEngObj.java,
     Compiled Code)
  at osp.EventEngine.EventDriver.go(EventDriver.java:114)
  at osp.EventEngine.EngineThread.run(EngineThread.java:61)
```

The trace says that a `NullPointerException` has occurred in method do_kill() of class ThreadCB at source code line 195. Going down the trace, we can see the sequence of method calls that led to the error: do_kill() was called by kill() of IflThreadCB, etc. The most important information here is the line number where the error occurred.[4]

OSP 2 prints a similar trace in the system log when an error or a warning is issued. For instance,

```
Sys: 36360 <<Warning!>> [Threads.ThreadCB]
  After do_kill(Thread(36:1/KL)): CPU is idle,
  but there are ready threads
  ready queue = (89:3,115:2,130:2,141:3,142:5)
  at osp.IFLModules.IflThreadCB.idleCPUwarning(IflThreadCB.java,
     Compiled Code)
  at osp.IFLModules.IflThreadCB.kill(IflThreadCB.java, Compiled Code)
  at osp.Tasks.TaskCB.do_kill(TaskCB.java, Compiled Code)
  at osp.IFLModules.IflTaskCB.kill(IflTaskCB.java, Compiled Code)
  at osp.IFLModules.IflTaskCB.killOldTasks(IflTaskCB.java, Compiled Code)
  at osp.IFLModules.CallbackTaskKill.voidCallback(IflTaskCB.java,
     Compiled Code)
  at osp.EventEngine.EventCallback.Activate(EventCallback.java,
     Compiled Code)
  at osp.EventEngine.EventEngObj.ActivateChildren(EventEngObj.java,
     Compiled Code)
  at osp.EventEngine.EventEngObj.Activate(EventEngObj.java, Compiled Code)
  at osp.EventEngine.EventDriver.go(EventDriver.java:114)
  at osp.EventEngine.EngineThread.run(EngineThread.java:61)
```

The trace appears after the warning message. In this case, we must look deeper in the trace to find out what happened. The top line of the trace says that the warning was issued by method idleCPUwarning() of class IflThreadCB, which was called by kill(), the system wrapper for the do_kill() method, which is part of a student project (refer back to Section 1.9.2 for the information about the naming conventions for methods that are implemented as part of student projects). The trace further says that the method IflThreadCB.kill() was in turn called by the method do_kill() of class TaskCB, which was called by IflTaskCB.kill(). It takes a little bit of analysis and understanding of the functionality of the different system calls to realize what happened: the task Task(1/L) has been destroyed by the system call TaskCB.kill(), which caused the destruction of all the threads that belong to that task. In particular, just after thread Thread(36:1/KL) was killed, the system detected that the CPU was idle even though some ready-to-run threads were present in the system. Thus, the cause of the warning is most probably the failure of the student implementation to call the dispatch() method at the end of do_kill().

[4] Line-number information is not always provided, unless you run the system using the debugger.

Obfuscation and stack traces. Unfortunately, the obfuscation that *OSP 2* employs to prevent inappropriate calls to certain methods diminishes the value of execution stack traces, because the names of some method calls listed in a trace might be unintelligible. However, even with name obfuscation, the trace often contains enough information to be useful. Here is an example of an obfuscated trace:

```
java.lang.NullPointerException
   at osp.IFLModules.IflDevice.enqueueIORB(IflDevice.java:283)
   at osp.FileSys.OpenFile.do_read(OpenFile.java:421)
   at osp.IFLModules.IflOpenFile.read(IflOpenFile.java:415)
   at osp.Memory.PageFaultHandler.a(PageFaultHandler.java:395)
   ...
```

In the last line, the real name of the method `osp.Memory.PageFaultHandler.a` was obfuscated and became unrecognizable (there is no method called `a` in the source code). However, it is still clear that the error occurred while a call to the method `enqueueIORB` was made from within the `read` method, which was executed on a file.

1.12 Project Submission

The manner by which you submit your *OSP 2* projects is determined by the instructor. The following instructions apply if your instructor chooses to use the automatic project submission system of *OSP 2*.

First, you will have to supply your email address to the instructor, who will prepare an account for you. The email address identifies you to the system. You must use the same address in all your interactions with the submission system.

The submission system provides three functions, which are available as links from the project submission page. The URL of this page will be supplied to you by your instructor. The functions are as follows:

1. *Change of password.*
   Clicking on this link will let you change your password. Your initial password will be mailed to you when the instructor sets up your account.

   After you change your password and then try to submit a project, you might see the “authorization has failed” dialog box. This happens when the browser tries to use your old password. It is not a problem, however, because clicking “OK” in the dialog box lets you re-enter the correct password.

2. *Password reminder.*

If you forget your password or if you did not receive the initial password for some reason, you should click on this link. First, you will get email with a link to a servlet. Clicking on this link will have the following effect:

◇ Your password will be changed to some random string.

◇ You will get your new password by email.

If the new password is hard to remember, you can use the "Change of password" function to change your password.

If you do not click on the aforesaid servlet link, your password will not be changed. It should be noted that the password-reminder function can be used only within intervals of at least four hours.

3. *Project submission.*
When you are ready to submit your project assignment, click on the "Submit assignment" link on the project submission page. After authentication, you will be presented with a form where you will be asked to enter the project name and the `*.java` files that comprise your program. The system then copies the sources over to the server and compiles them. If successful, the sources stay on the server (so that they can be checked by the instructor and his or her teaching assistants) and the compiled class files are sent back to your browser as an applet. Next, you will have to run this applet (by clicking on appropriate buttons). If you are happy with the results, click on the submission button. The simulation run will then be sent to the server (again, so that the instructor can check it for errors).

   Note that some browsers do not give a warning when a non-existing file is being sent to the server. In some cases (e.g. when the file is actually a directory name) the browser might even hang. Therefore, it is important to make sure that you send the correct `*.java` files to the server.

   You should keep in mind that the instructor might set up the submission process in such a way that your project would have to be run with several parameter files. When the first run is finished, you should press the *Submit* button and then the *Next* button. If there are more parameter files to be considered, a new applet will start. When this is finished, submit the output and hit *Next* again. When your project has been run with all the parameter files, you will receive a confirmation and the main project-submission page will be displayed.

   Finally, some browsers might issue a security exception when you try to run the submission applet. You will see this exception in the Java console of the browser (we recommend that you always run the submission applet with the Java console open). If this happens, you should place the file

.java.policy in your home directory. This file should contain the entry

```
grant {
     permission java.security.AllPermission;
};
```