4
THREADS: *Management and Scheduling of Threads*

4.1 Chapter Objective

Threads are the schedulable and dispatchable units of execution in *OSP 2*. The objective of the THREADS project is to teach students about thread management and scheduling in a modern-day operating system and to provide them with a well-structured programming environment in which to implement thread-management and scheduling techniques. To this end, students will be asked to implement the two public classes of the THREADS package: `ThreadCB` and `TimerInterruptHandler`. The former implements the most common operations on a thread, while the latter is a timer interrupt handler that can be used to implement time-quantum-based scheduling algorithms for threads. We begin this chapter with an overview of thread basics.

4.2 Overview of Threads

Multi-threading refers to the ability of an OS to support multiple threads of execution within a single task. There are at least four reasons why it is desirable to structure applications as multi-threaded ones:

Parallel Processing: A multi-threaded application can process one batch of data while another is being input from a device. On a multiprocessor architecture, threads may be able to execute in parallel, leading to more work getting done in less time.

Program Structuring: Threads represent a modular means of structuring an application that needs to perform multiple, independent activities.

Interactive Applications: In an interactive application, one thread can be used to carry out the current command while, at the same time, another thread prompts the user for the next command. This pipelining effect can lead to a perceived increase in the speed of the application.

Asynchronous Activity: A thread can be created whose sole job is to schedule itself to perform periodic backups in support of the main thread of control in a given application.

Concurrency: Threads can execute concurrently. Thus, for example, a server process can service a number of clients concurrently: each client request triggers the creation of a new thread within the server.

We thus see that there is considerable incentive from an application programming perspective for an OS to support multi-threading.

Threads as Independent Entities. As explained in Chapter 3, the resources available to a thread, such as memory, open files and communication ports, are those belonging to the task to which the thread is affiliated. That is, a task is a container for one or more threads and each of these threads has shared access to the resources owned by the task. There is, however, certain information associated with a thread that allows it to execute as a more or less independent entity:

- ◇ A thread execution state (Running, Ready, Blocked, etc.).
- ◇ A saved thread context when not running. This context includes the contents of the machine registers when it was last running; in particular, every thread has its own, independent program counter.
- ◇ An execution stack.
- ◇ A certain amount of per-thread static[1] storage for local variables.
- ◇ Access to the memory and resources of its container task; it shares these resources with the other threads in that task.

[1] Not to be confused with the Java keyword `static` used to define a variable as a class variable or a method as a class method.

It is worth taking time to emphasize the implications of this last item. All the threads of a given task reside in the same address space and have access to the same data. Consequently, when one thread modifies a piece of data, the effect of this change is visible to the other threads should they subsequently decide to read this data item. If one thread opens a file with read access, the other threads in the same task will also be able to read from this file. It is thus imperative that when programming a multi-threaded application, the actions of the threads be carefully coordinated; otherwise conflicts could easily arise that could hinder the threads from performing their desired computation.

Scheduling Algorithms for Threads. As previously noted, threads are the schedulable units of execution in *OSP 2* and any other OS that supports threads. This represents a shift from older operating systems like traditional Unix in which processes played this role.[2] Thread scheduling is an integral part of multiprogramming: when the currently executing thread becomes blocked waiting for some event to occur, this represents a golden opportunity for the OS to perform a context switch so that a ready-to-run thread can be given control of the CPU. In this way, the CPU is kept busy most of the time, thereby increasing its utilization.

So what are the kinds of events that threads may block on? These include I/O interrupts and software signals. It should be noted, however, that an OS can decide to perform a context switch any time it is convenient, again for the purpose of improving system performance. Convenient in this case means any time control resides within the OS, and include occasions such as timer interrupts and system call invocations.

The question you must now ask yourself is which thread should the OS schedule next when a context switch is to take place? The decision taken here is critical; it can significantly impact a variety of performance-related measures, such as:

CPU utilization: the percentage of time the CPU is kept busy (not idle).

Throughput: the number of jobs or tasks processed per unit of time.

Response time: the amount of time needed to process an interactive command. Typically one is interested in the average response time over all commands.

Turnaround time: The amount of time needed to process a given task. Includes actual execution time plus time spent waiting for resources, including the CPU.

[2] Modern Unix implementations, like SUN's Solaris, IBM's AIX, and Linux, do, of course, support threads.

The answer to the question as to which thread to schedule next lies in the **CPU scheduling algorithm** the OS implements. A variety of scheduling algorithms have been proposed in the literature and they can be classified along the following lines:

Emphasis on response time vs. CPU utilization. Algorithms of the former kind can be thought of as user-oriented and those of the latter kind as system-oriented.

Preemptive vs. nonpreemptive. A preemptive algorithm may interrupt a thread and move it to the ready-to-run queue, while in the nonpreemptive case, a thread continues to execute until it terminates or blocks on some event. Several preemptive algorithms preempt a thread after it has finished up its "slice" or quantum of CPU time.

Fair vs. unfair. In a fair algorithm, every thread that requires access to the CPU eventually gets time on the CPU. In the absence of fairness, **starvation** is possible and the algorithm is said to be unfair in this case.

Choice of selection function. The selection function determines which thread, among the ready-to-run threads, is selected next for execution. The choice can be based on priority, resource requirements, or execution characteristics of the thread such as the amount of elapsed time since the thread last got to execute on the CPU.

We now briefly describe some of the more common scheduling algorithms that have been proposed. In describing these algorithms, we assume the existence of a **ready queue** where ready-to-run threads lie in wait for the CPU.

First-Come-First-Served (FCFS) As the name indicates, threads are serviced in the order they entered the ready queue. This is probably the simplest scheduling algorithm that has been proposed and has the tendency to favor long, CPU-intensive threads over short, I/O-bound threads.

Round Robin. Like FCFS but each thread gets to execute for a length of time known as the **time slice** or **time quantum** before it is preempted and placed back on the ready queue. Time slicing can be used to allow short-lived threads, corresponding to interactive commands, to get through the system quickly, thereby improving the system's response time.

Shortest Thread Next (STN). This is a nonpreemptive policy in which the thread with the shortest expected processing time is selected next. Like round robin, it tends to favor I/O-bound threads. The scheduler must have an estimate of processing time to perform the selection function.

Shortest Remaining Time (SRT). This is a preemptive version of STN in which the thread with the shortest expected remaining processing time is selected next. SRT tends to yield superior turnaround time performance compared with STN.

Highest Response Ratio Next (HRRN). A nonpreemptive algorithm that chooses the thread with the highest value of the ratio of $R = \frac{w+s}{s}$, where R is called the response ratio, w is the time spent waiting for the CPU, and s is the expected service time. Favors short threads but also gives priority to aging threads with high values for w.

Feedback. This algorithm, sometimes referred to as "multi-level round robin" utilizes a series of queues, each with their own time quantum. Threads enter the system at the top-level queue. If a thread gains control of the CPU and exhausts its time quantum, it is demoted to the next lower queue. The lowest queue implements pure round robin. The selection function chooses the thread at the head of the highest non-empty queue. Thus this algorithm penalizes long-running threads since each time they use up their time quantum, they are demoted to the next lower queue.

Priority-Driven Preemptive Scheduling. The basic idea of this scheme is that when a thread becomes ready to execute whose priority is higher than the currently executing thread, the lower-priority thread is preempted and the processor is given to the higher-priority thread. Thread priorities may be computed statically (threads have a fixed priority that never changes) or adjusted dynamically (a thread's priority begins at some initial assigned value and then may change, up or down, during the thread's lifetime). The priority-driven preemptive approach to thread scheduling is especially important in operating systems that support **real-time** threads or processes, such as Linux, Unix SVR4, and Windows 2000/XP/Vista.

The rest of this chapter describes each class in the package THREADS in detail. These classes are placed in a larger context in the class diagram given in Figure 4.1.

4.3 The Class `ThreadCB`

`ThreadCB` stands for **thread control block**; it is a class that contains all the structures necessary for maintaining the information about each particular thread. This class is defined as follows:

◇ `public class ThreadCB extends IflThreadCB`

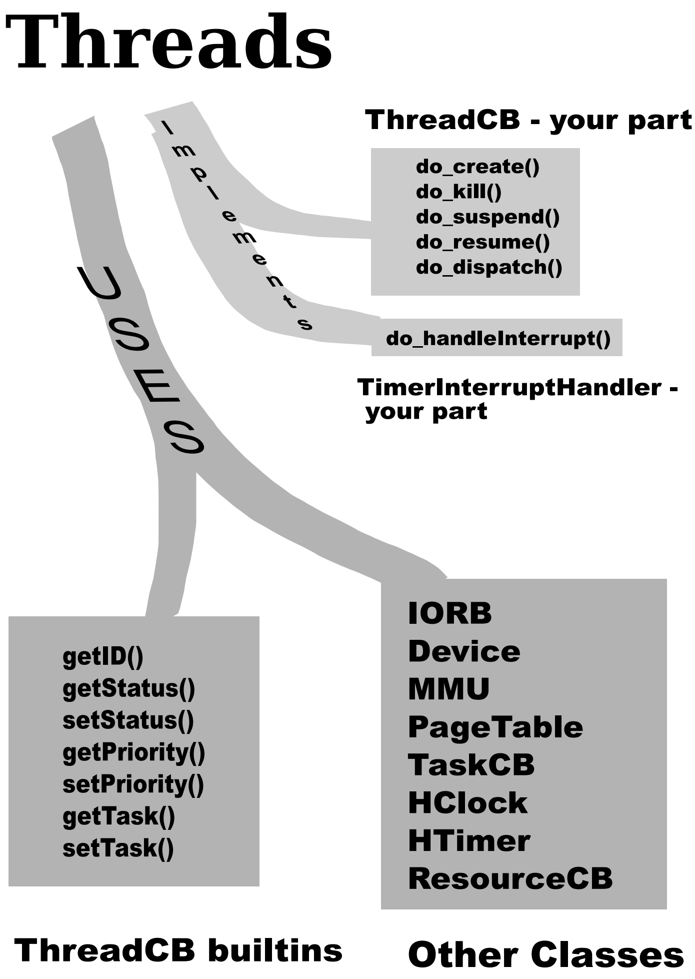

Figure 4.1 Overview of the package THREADS.

Like other classes that belong to student projects, this class defines methods that start with `do_` and that are wrapped with similarly named methods in class `IflThreadCB`. Before discussing the required functionality of the methods in `ThreadCB` we need to look deeper into the nature of *OSP 2* threads.

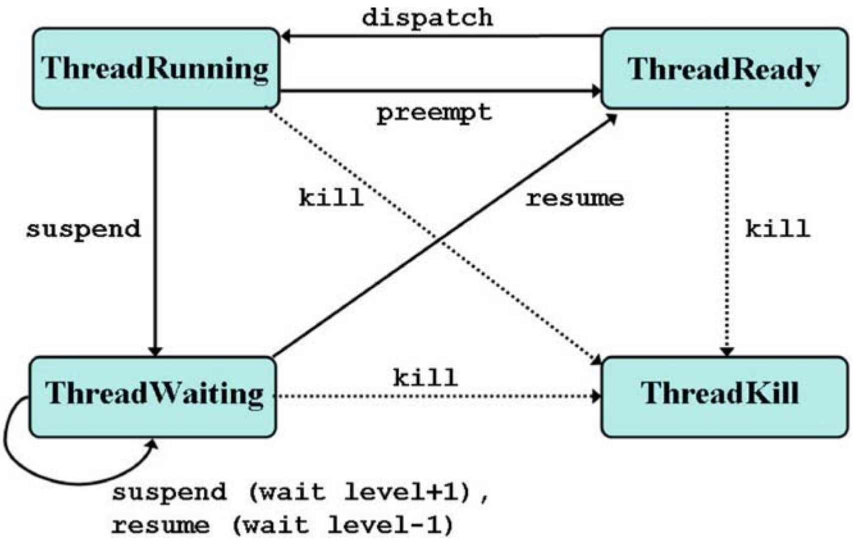

Figure 4.2 The state transition diagram for *OSP 2* threads.

State transitions. Thread management is concerned with two main issues: the life cycle of a thread (i.e., **creation** and **destruction** of threads) and maintaining thread status and moving threads between different queues and CPU (**suspension**, **resumption**, and **dispatching**). Therefore, to understand thread management in *OSP 2* it is important to understand the different states a thread can be in and how state transitions take place. Figure 4.2 illustrates this issue.

When a thread is first created, it enters the ready state (`ThreadReady`), which means it must be placed on the queue of ready-to-run threads. *OSP 2* does not prescribe how this queue is supposed to be organized and it is entirely up to the student implementation, unless the instructor has specific requirements.

From then on, two things can happen: a ready-to-run thread can be **scheduled to run** (and **dispatched**) and gain control of the CPU (and thus change its status to `ThreadRunning`), or it can be **destroyed** (or **killed**) and change its status to `ThreadKill`.

A thread can be dispatched only if it has the status `ThreadReady`, but a live thread (i.e., one that has status *other than* `ThreadKill`) can be killed in any state, not only in the ready state. One sad thing about *OSP 2* threads is that they never die of natural causes: they either get destroyed by somebody else or self-destroy. In other words, there is no separate system call to terminate a thread normally and there is no special state to denote normal thread termination.

A running thread can be preempted and placed back into the **ready queue** or it can be suspended to the **waiting state**. The latter can happen due to a pagefault or when the thread executes a blocking system call, such as an I/O operation or a communication (sending or receiving a message). *OSP 2* does not place any restrictions on the way the ready queue is implemented, so you should use your own design. However, your instructor may have specific requirements to how scheduling is to be done. In this case, some designs might be much better than others.

An *OSP 2* thread can be at several levels of waiting. When a running thread enters the pagefault handler or when it executes a blocking system call (e.g., `write()`), it enters the level 0 waiting state represented by the integer constant `ThreadWaiting`. Level 1 waiting state is represented by the constant `ThreadWaiting+1`, etc.

A thread is not always blocked when it enters a waiting state. For instance, when a thread causes a pagefault or executes a `write()` operation on a file, its waiting state signifies that in order to continue execution of the user program the thread needs to wait until the pagefault or the system call is finished. In other words, the thread switches hats: it leaves the user program and becomes a system thread. A system thread might do some work needed to process the request and then it might execute another system call. At this point, it would enter the waiting state at level 1, which signifies that the original thread has to wait for two system calls to complete. If the second system call is blocking (e.g., involves I/O), the execution of the thread will block until the appropriate event happens (e.g., the I/O completes).

To illustrate this process, consider processing of a pagefault (Chapter 5). When a pagefault occurs, the thread enters the level 0 waiting state, executes a page replacement algorithm and then makes a system call to `write()`. When the `write()` call starts execution, the thread's waiting level is bumped up to 1. After assembling a proper I/O request to the swap device, the thread will suspend itself on a blocking event, to wait for the I/O. At this point, the thread will be in state `ThreadWaiting+2`. When the I/O is finished, the `resume()` method is executed on the thread and it drops into the level 1 waiting state. When the `write()` system call is about to exit, another `resume()` is executed and the thread's wait level drops to 0 (i.e., its state becomes `ThreadWaiting` again). Next, while still in the pagefault handler, the thread would execute the `read()` system call and go into the waiting state at levels 1 and 2, similar to the `write()` call. When the `read()` operation is finished, the ensuing `resume()` operations will drop the thread to level 0 again. At this point, the pagefault handler performs some record-keeping operations (see Chapter 5), executes a `resume()` operation and exits. This causes the thread to change its status from `ThreadWaiting` to `ThreadReady`.

In sum, an *OSP 2* thread can be suspended to several levels of depth by executing a sequence of nested `suspend()` operations. When all the corresponding events happen, the `resume()` method is called on the thread, which decreases the wait level by 1. When all the events on which the thread is suspended occur, the thread goes back into the `ThreadReady` state.

Context switching. Passing control of the CPU from one thread to another is called **context switching**. This has two distinct phases: **preempting** the currently running thread and **dispatching** another thread. Preempting a thread involves the following steps:

1. Changing of the state of the currently running thread from `ThreadRunning` to whatever is appropriate in the particular case. For instance, if a thread loses control of the CPU because it has to wait for I/O, then its status might become `ThreadWaiting`. If the thread has used up its time quantum, then the new status should become `ThreadReady`. Changing the status is done using the method `setStatus()` described later.

   This step requires knowing the currently running thread. The call `MMU.getPTBR()` (described below) lets you find the page table of the currently scheduled task. The task itself can be obtained by applying the method `getTask()` to this page table. The currently running thread is then determined using the method `getCurrentThread()`.

2. Setting the **page table base register** (PTBR) to null. PTBR is a register of the memory management unit (a piece of hardware that controls memory access), or MMU, which always points to the page table of the running thread. This is how MMU knows which page table to use for address translation. In *OSP 2*, PTBR can be accessed using the static methods `getPTBR()` and `setPTBR()` of class `MMU`.

3. Changing the current thread of the previously running task to `null`. The current thread of a task can be set using the method `setCurrentThread()`.

When a thread, t, is selected to run, it must be given control of the CPU. This is called **dispatching** a thread and involves a sequence of steps similar to the steps for preempting threads:

1. The status of t is changed from `ThreadReady` to `ThreadRunning`.

2. PTBR is set to point to the page table of the task that owns t. The page table of a task can be obtained via the method `getPageTable()`, and the PTBR is set using the method `setPTBR()` of class `MMU`.

3. The current thread of the above task must be set to t using the method `setCurrentThread()`.

In practice, context switch is performed as part of the `dispatch()` operation, and steps 2 and 3 in the first list above can be combined with steps 2 and 3 of the second list.

In the degenerate case, when the running thread t is suspended and no other thread takes control of the CPU, consider it as a context switch from t to the imaginary "`null` thread". Likewise, if no process is running and the dispatcher chooses some ready-to-run thread for execution, you can view it as a context switch from the `null` thread to t.

Events. Before going on you must revisit Section 1.6, which describes the `Event` class.

The state transition diagram shows that to a large extent thread management is driven by two operations: `suspend()` and `resume()`. The suspend operation places a thread into a waiting queue of the event passed as an argument (and increases the wait level) and the resume operation decreases the wait level and, if appropriate, places it into the queue of ready-to-run threads (in which all threads are in the `ThreadReady` state). All this is accomplished using the `Event` class discussed in Section 1.6. Note that, as described earlier, a thread can execute several suspend operations on different events, so it might find itself in different waiting queues. The thread will be notified about the completion of these events in the order opposite to that in which the `suspend()` operations were performed. After all the relevant events have occurred, the thread is free to execute again and is placed on the ready queue.

Only the first method in class `Event`, `addThread()`, is really necessary for the THREADS project, but other methods might be useful for debugging (and, of course, they are necessary for other *OSP 2* projects).

Methods of class `ThreadCB`. These are the methods that have to be implemented as part of the project. Their implementation requires support from other parts of OSP in the form of the methods that can be called from within `ThreadCB` to accomplish a specific objective. We discuss these methods as part of the required functionality and then give a summary of these methods in a separate section.

◇ `public static void init()`
This method is called once at the beginning of the simulation. You can use it to set up static variables that are used in your implementation, if necessary. If you find no use for this feature, leave the body of the method empty.

◇ `public static ThreadCB do_create(TaskCB task)`
The job of this method is to create a thread object using the default constructor `ThreadCB()` and associate this newly created thread with a task (provided as an argument to the `do_create()` method). To link a thread to its task, the method `addThread()` of class `IflTaskCB` should be used and the thread's task must be set using the method `setTask()` of `IflThreadCB`.

There is a global constant (in `IflThreadCB`), called `MaxThreadsPerTask`. If this number of threads per task is exceeded, no new thread should be created for that task, and `null` should be returned. `null` should also be returned if `addThread()` returns `FAILURE`. You can find out the number of threads a task currently has by calling the method `getThreadCount()` on that task.

If priority scheduling needs to be implemented, the `do_create()` method must correctly assign the thread's initial priority. The actual value of the priority depends on the particular scheduling policy used. *OSP 2* provides methods for setting and querying the priority of both tasks and threads. The methods are `setPriority()` and `getPriority()` in classes `TaskCB` and `ThreadCB`, respectively.

Finally, the status of the new thread should be set to `ThreadReady` and it should be placed in the ready queue.

If all is well, the thread object created by this method should be returned.

It is important to keep in mind that each time control is transferred to the operating system, it is seen as an opportunity to schedule a thread to run. Therefore, regardless of whether the new thread was created successfully, the dispatcher must be called (or else a warning will be issued).

◇ `public void do_kill()`
This method destroys threads. To destroy a thread, its status must be set to `ThreadKill` and a number of other actions must be performed depending on the current status of the thread. (The status of a thread can be obtained via the method `getStatus()`.)

If the thread is ready, then it must be removed from the ready queue. If a running thread is being destroyed, then it must be removed from controlling the CPU, as described earlier.

There is nothing special to do if the killed thread has status `ThreadWaiting` (at any level). However, you are not done yet. First, the thread being destroyed might have initiated an I/O operation and thus is suspended on the corresponding IORB. The I/O request might have been enqueued to some device and has not been processed because the device may be busy with other work. What should now happen to the IORB? Should you just let the device work on a request that came from a dead thread?

The answer is that you should cancel the I/O request by removing the corresponding IORB from its device queue. This can be done by scanning all devices in the device table and executing the method `cancelPendingIO()` on each device. The device table is an array of size `Device.getTableSize()` (starting with device 0), where device `i` can be obtained with a call to `Device.get()`.

During the run, threads may acquire and release shared resources that are needed for the execution. Therefore, when a thread is killed, those resources must be released into the common pool so that other threads could use them. This is done using the static method `giveupResources()` of class `ResourceCB`, which accepts the thread be killed as a parameter.

Two things remain to be done now. First, you must dispatch a new thread, since you should use every interrupt or a system call as an opportunity to optimize CPU usage. Second, since you have just killed a thread, you must check if the corresponding task still has any threads left. A task with no threads is considered dead and must be destroyed with the `kill()` method of class `TaskCB`. To find out the number of threads a task has, use the method `getThreadCount()` of `TaskCB`.

◇ `public void do_suspend(Event event)`
To suspend a thread, you must first figure out which state to suspend it to. As can be seen from Figure 4.2, there are two candidates: If the thread is running, then it is suspended to `ThreadWaiting`. If it is already waiting, then the status is incremented by 1. For instance, if the current status of the thread is `ThreadWaiting` then it should become `ThreadWaiting+1`. You now must set the new thread status using the method `setStatus()` and place it on the waiting queue to the event.

If `suspend()` is called to suspend the running thread, then the thread must lose control of the CPU. Switching control of the CPU can also be done in the dispatcher (as part of the context switch), but it has to be done somewhere to avert an error.

Finally, a new thread must be dispatched using a call to `dispatch()`.

◇ `public void do_resume()`
A waiting thread can be resumed to a waiting state at a lower level (e.g., `ThreadWaiting+2` to `ThreadWaiting+1` to `ThreadWaiting` or from `ThreadWaiting` to the status `ThreadReady`). If the thread becomes ready, it should be placed on the ready queue for future scheduling. Finally, a new thread should be dispatched.

Note that there is no need to take the resumed thread out of the waiting queue to any event. A typical sequence of actions that leads to a call to `resume()` is as follows: When an event happens, the method `notifyThreads()` is invoked on the appropriate `Event` object. This method examines the waiting queue of the event, removes the threads blocked on this event one by one, and calls `resume()` on each such thread. So, by the time `do_resume()` is called, the corresponding thread is no longer on the waiting queue of the event.

◇ `public static int do_dispatch()`
This method is where thread scheduling takes place. Scheduling can be as simple as plain round-robin or as complex as multi-queue scheduling with feedback. *OSP 2* does not impose any restrictions on how scheduling is to be done, provided that the following conventions are followed.

First, some thread should be chosen from the ready queue (or the currently running thread can be allowed to continue). If a new thread is chosen, *context switch* must be performed, as described earlier, and `SUCCESS` returned. If no ready-to-run thread can be found, `FAILURE` must be returned.

Relevant methods defined in other packages. Apart from the methods of the `Event` class listed above, the following methods of other classes should or can be used to implement the methods in class `ThreadCB` as described above:

◇ `final public int getDeviceID()` `IORB`
Returns the device Id number that this I/O request is for.

◇ `final static public Device getDevice(int deviceID)` `Device`
Returns the device object corresponding to the given Id number.

◇ `final static public int getTableSize()` `Device`
Tells how many devices there are. The number is specified in the parameter file and can vary from one simulation run to another.

◇ `final static public Device get(int deviceID)` `Device`
Returns the device object with the given Id. In conjunction with `getTableSize()`, this method can be used in a loop to examine each device in turn. Note that all devices are mounted by *OSP 2* at the beginning of the simulation and no devices are added or removed during a simulation run. Therefore the number of devices remains constant and the device table has no "holes".

◇ `public void cancelPendingIO(ThreadCB th)` `Device`
The context for this method is a device object, and the method cancels pending IORBs of the thread *th* on that device. This is done when *th* is killed to prevent the servicing of pending I/O's requested by killed threads. However, this method does not cancel the IORB that is currently being serviced by the device. The device is just allowed to finish.

◇ `final static public PageTable getPTBR()` `MMU`
This method returns the value of the page table base register (PTBR) of the MMU. PTBR holds a reference to the page table of the currently running task.

◇ `static public void setPTBR(PageTable table)` `MMU`
This method allows one to set the value of PTBR. When no thread is running, the value should be null; otherwise, it must be the page table of the task that owns the currently running thread.

◇ `public final TaskCB getTask()` `PageTable`
Returns the task that owns the page table.

◇ `public void kill()` `TaskCB`
Kills the task on which this method is invoked.

◇ `public int getThreadCount()` `TaskCB`
Tells how many threads the task has.

◇ `public int addThread(ThreadCB thread)` `TaskCB`
Attaches a newly created thread to task. Returns `SUCCESS` or `FAILURE`.

◇ `public int removeThread(ThreadCB thread)` `TaskCB`
Removes killed thread to task.

◇ `public ThreadCB getCurrentThread()` `TaskCB`
Returns the current thread object of the task.

◇ `public void setCurrentThread(ThreadCB t)` `TaskCB`
Sets the current thread of the task to the given thread.

◇ `final public int getPriority()` `TaskCB`
Tells the priority of the task.

◇ `final public void setPriority(int p)` `TaskCB`
Sets the priority of the task. The `setPriority()/getPriority()` methods are provided for convenience, in case priority scheduling is used and dispatching takes into account the priority of both the task and the thread.

◇ `final public PageTable getPageTable()` `TaskCB`
Returns the page table of the task.

◇ `final public int getStatus()` `TaskCB`
Returns the task's status.

◇ `set()` and `get()` `HTimer`
These classes can be used to set and query the hardware timer. See Section 1.4 for details.

◇ `get()` `HClock`
This method is described in Section 1.4; it is used to query the hardware clock of the simulated machine.

◇ `public static void giveupResources(ThreadCB thread)` `ResourceCB`
Releases all abstract shared resources held by the thread. Note: these resources do not include concrete resources such as memory or CPU.

Summary of Class `ThreadCB`

The following table summarizes the attributes of class `ThreadCB` and the methods for manipulating them. These attributes and methods are provided by the class `IflThreadCB` and are inherited. The methods appearing in the table are more fully described in Section 4.5.

Task: The task that owns the thread. This property can be set and queried via the methods `setTask()` and `getTask()`.

Identity: The identity of a thread can be obtained using the method `getID()`. This property is set by the system.

Status: The status of the thread. The relevant methods are `setStatus()` and `getStatus()`.

Priority: The priority of the thread. The methods to query and change thread's priority are `setPriority()` and `getPriority()`.

Creation time: The value of this property can be obtained using the method `getCreationTime()`.

CPU time used: The total CPU time used by the thread can be obtained via the method `getTimeOnCPU()`.

4.4 The Class `TimerInterruptHandler`

This class is much simpler than `ThreadCB`. It is defined as `public class TimerInterruptHandler extends IflTimerInterruptHandler`

and contains only one method:

- ◇ `public void do_handleInterrupt()`
  This method is called by the general interrupt handler when the system timer expires. The timer interrupt handler is the simplest of all interrupt handlers in *OSP 2*. Its main purpose is to schedule the next thread to run and, possibly, to set the timer to cause an interrupt again after a certain time interval. Resetting the times can also be done in the `dispatch()` method of `ThreadCB` instead, because the dispatcher might want to have full control over CPU time slices allocated to threads.

Relevant methods defined in other packages. The following is a list of methods that belong to other classes and might be useful for implementing `do_handleInterrupt()`:

◇ `final static public void set(int time)` `HTimer`
Sets the hardware timer to `time` ticks from now. Cancels the previously set timer, if any.

◇ `final static public int get()` `HTimer`
Returns the time left to the next timer interrupt.

4.5 Methods Exported by the THREADS Package

The following is a summary of the public methods defined in the classes of the THREADS package or in the corresponding superclasses, which can be used to implement this and other student packages. To the right of each method we list the class of the objects to which the method applies. In the case of the THREADS package, all exported methods belong to the class `TaskCB`, which inherits them from the superclass `IflTaskCB`.

◇ `final public static ThreadCB create()` `ThreadCB`
This method is a wrapper around the method `do_create()` described in this chapter. It is provided by `IflThreadCB` and is inherited by `ThreadCB`. Returns the created thread.

◇ `final public static void dispatch()` `ThreadCB`
This is a wrapper around the method `do_dispatch()` described in this chapter. This method is provided by `IflThreadCB` and is inherited by `ThreadCB`.

◇ `final public void suspend(Event event)` `ThreadCB`
This is a wrapper around the method `do_suspend()` described in this chapter. This method is provided by `IflThreadCB` and is inherited by `ThreadCB`.

◇ `final public void resume()` ThreadCB
This is a wrapper around the method `do_resume()` described in this chapter. This method is defined in `IflThreadCB`, but it is inherited by `ThreadCB`.

◇ `final public void kill()` ThreadCB
This is a wrapper around the method `do_kill()` described in this chapter. This method is defined in `IflThreadCB`, but it is inherited by `ThreadCB`.

◇ `final public TaskCB getTask()` ThreadCB
Returns the task this thread belongs to.

◇ `final public void setTask(TaskCB t)` ThreadCB
Sets the task of the thread.

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

◇ `final public void setStatus(int s)` ThreadCB
Sets the status of this thread.

◇ `public double getTimeOnCPU()` ThreadCB
Tells the total time the thread has been using CPU.

◇ `final public long getCreationTime()` ThreadCB
Returns the creation time of the thread.

◇ `final public int getPriority()` ThreadCB
Tells the priority of this thread.

◇ `final public void setPriority(int p)` ThreadCB
Sets the priority. The `setPriority` and `getPriority` methods are provided for convenience, in case the assignment calls for priority scheduling. *OSP 2* does not actually care how priority is used, if at all.