3
TASKS: *Management of Tasks (a.k.a. Processes)*

3.1 Chapter Objective

The objective of the TASKS project is to teach students about task management in a modern-day operating system and to provide them with a well-structured programming environment in which to implement task-management techniques. To this end, students will be asked to implement the *OSP 2* class `TaskCB`, the only class of package TASKS. `TaskCB` stands for *Task Control Block*, the *OSP 2* object used to represent tasks.

3.2 Conceptual Background

Like other modern operating systems, *OSP 2* distinguishes between program execution and resource ownership. The former is captured through the concept of a **thread**, which represents a running program, and the latter is captured using the concept of a **task**. In older operating systems, like traditional Unix, the **process** filled both of these roles; actually, we sometimes use the term "process" as a synonym for task. In *OSP 2*, a task serves as a "container" for one or more threads, all executing the same code and sharing the same memory address space. Also associated with a task is a swap file containing an image of

the task's address space, other files opened by the task's constituent threads, and the communication ports created by these threads. We say that these resources (memory, ports, files, etc.) are *owned* by the task and *shared* by the task's threads; this explains how the issue of resource ownership is organized around the concept of a task.

Threads are the schedulable and dispatchable units of execution in *OSP 2*. They are sometimes referred to as "lightweight processes" for it is much easier in a multiprogramming OS to switch the CPU from one thread to another than from one process to another, due to above-explained separation of program execution and resource ownership in an OS supporting the task/thread doctrine. We will have more to say about threads in the next chapter.

A task can be created or destroyed, newly created threads can be added to a task, and threads are deleted from the owner task's thread list after they are destroyed. There is also a system-wide notion of the **current task**, which is the task that owns the currently running thread. This thread is known as the **current thread** of the task.

In the rest of this chapter we describe `TaskCB`, the only class in the TASKS package. The class diagram of Figure 3.1 puts `TaskCB` in context with related classes.

3.3 Class `TaskCB`

Tasks are represented by the class `TaskCB`, which is the only class to be implemented in the TASKS project. It is defined as follows:

◇ `public class TaskCB extends IflTaskCB`

The following methods are to be implemented as part of this project:

◇ `public static void init()`
This method is called at the very beginning of simulation and can be used to initialize static variables of the class, if necessary.

◇ `public static TaskCB do_create()`
This method creates a new task object and then initializes it properly.

In *OSP 2*, creation of a task involves the creation of a task object, allocation of resources to the task, and various initializations. The task object is created using the default task constructor `TaskCB()`. First, a page table must be created using the `PageTable()` constructor, and associated with the task using the method `setPageTable()`. Second, a task must keep track of its threads (objects of type `ThreadCB`), communication ports (objects of type `PortCB`),

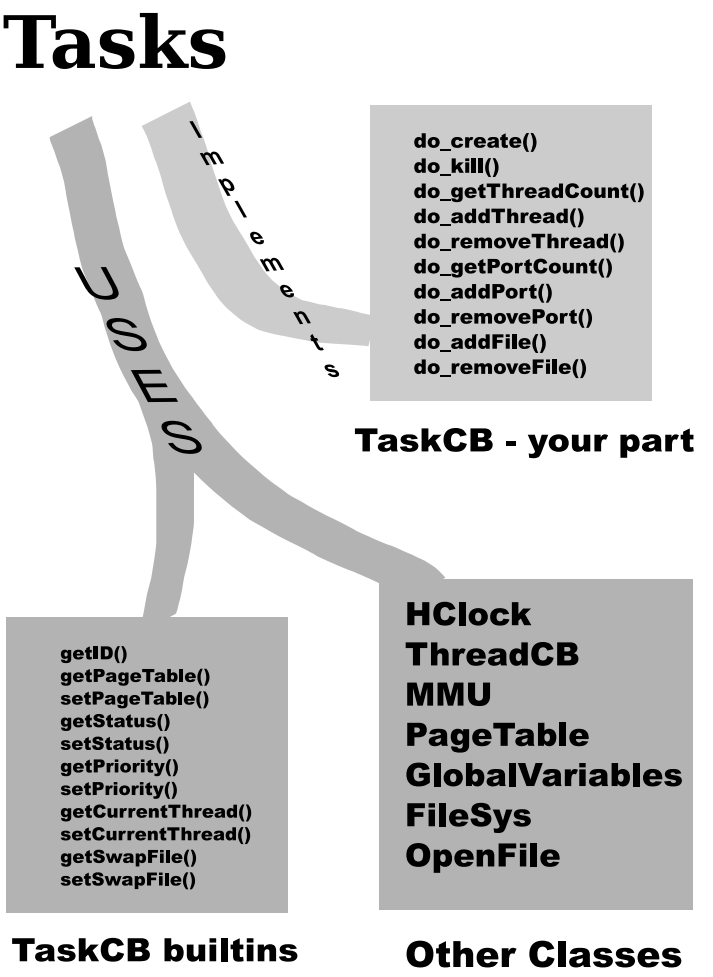

Figure 3.1 Overview of the package TASKS.

and files (objects of type `OpenFile`), which means that the appropriate structures have to be created. *OSP 2* does not have any specific requirements for these data structures, except that they must correctly maintain the inventory of threads, ports, and files attached to the task. Lists or variable-size arrays are good candidates.

Next, the task-creation time should be set equal to the current

simulation time (available through the class `HClock`), the status should be set to `TaskLive`, and the task priority should be set to some integer value. *OSP 2* does not prescribe what this value should be; it is determined by the requirements of the project and might be specified by the instructor (if, for example, the scheduling strategy implemented in the THREADS project uses task priorities).

The next important step is the creation of the swap file for the task. A swap file contains the image of the task's virtual memory space and thus is equal to the maximal number of bytes in the virtual address space of the task. In *OSP 2* this number is determined by the number of bits needed to specify an address in the virtual address space of a task, and is obtained using the following method: `MMU.getVirtualAddressBits()`. The name of the swap file is, by convention, the same as the task ID number, and the file itself resides in the directory specified by the global constant `SwapDeviceMountPoint`. To create the swap file, you should use the static method `create()` of class `FileSys`. Then the file has to be opened using the static method `open()` of `OpenFile`. The `open()` method takes a string that represents a full path name of a file and returns a run-time file handle that is used in the read, write, and close file operations. The resulting open-file handle should be saved in the task data structure using the method `setSwapFile()`.

An `open()` operation can fail due to lack of space on the swap device. In this case the `do_create()` method of `TaskCB` should dispatch a new thread and return `null`.

A task in *OSP 2* must have at least one live thread, so you need to create the first thread for the task using the static method `create()` of class `ThreadCB`. Finally, the `TaskCB` object created and initialized by your `do_create()` method should be returned.[1]

◇ `public void do_kill()`
This method is called to destroy a task. First, it should iterate through the list of all live threads of the task and `kill()` them. (Recall that maintenance of this list is entirely the responsibility of your implementation.) Each time a thread is killed, the `do_removeThread()` method is called by the THREADS package. The `do_kill()` method should then iterate over the ports attached to the task and `destroy()` them as well. Each request to destroy a port will eventually result in a call to your `do_removePort()` method. The status

[1] There is no need to invoke the `dispatch()` method of `ThreadCB` in order to schedule a thread to run after the `do_create()` system call is complete. Since a new thread is created as part of the process of task creation, `dispatch()` will be called by the `create()` method of `ThreadCB`. However, calling `dispatch()` before leaving `do_create()` is harmless.

of the task should be set to `TaskTerm` (terminated task) and the memory previously allocated to the task should be released. The latter is accomplished by invoking the method `deallocateMemory()` of class `PageTable` on the page table of the task.

The last resource left to be released by the task is the set of files opened by the various threads of the task and the swap file of the task. The **open files table** of a task is a data structure that should be maintained as part of the implementation of class `TaskCB` and should include all files opened by the threads of the task (which are objects of class `OpenFile`); *OSP 2* does not prescribe how this should be done. To free up this resource, you must `close()` every file in the open files table.

You should keep in mind that each call to `close()` eventually results in a call to your method `do_removeFile()`. However, this might not happen immediately. When you close a file that is the target of an active I/O operation, i.e., an operation that is currently being processed by an external device such as a disk, the file is not closed immediately. Rather, the system will remember that the file needs to be closed and will re-issue the `close()` command when the I/O operation completes. Because of this possible delay, some files of the task can remain open for a period of time even after you perform the `close()` operation on every open file. This means, of course, that calls to your method `do_removeFile()` might be similarly delayed.

Finally, the swap file of the task must be destroyed using method `delete()` of `FileSys`.[2] The argument to this method is the name of the swap file (see the discussion of `do_create()`).

◇ `public int do_getThreadCount()`
This method must return a correct thread count, which must be maintained as part of the implementation of the `do_create()` and `do_kill()` methods.

◇ `public int do_addThread(ThreadCB thread)`
This method is called by other parts of *OSP 2* whenever a new thread is created. The purpose of these calls is to notify `TaskCB` of the creation of a new thread so that the inventory of threads owned by the task can be properly updated. `SUCCESS` is to be returned unless the maximum number of threads for this task has been reached, in which case, `FAILURE` should be returned.

◇ `public int do_removeThread(ThreadCB thread)`
This method is called when a thread is destroyed. The thread should be

[2] Closing a file does not deallocate the space; it merely removes the file handle and flushes the data on disk. Deleting a file removes a hard link to the file, and when the number of such links becomes zero, the file space is freed.

removed from the list of threads owned by the task. `SUCCESS` should be returned if the thread belongs to the task and `FAILURE` otherwise.

◇ `public int do_getPortCount()`
Returns the number of ports owned by the task.

◇ `public int do_addPort(PortCB newPort)`
This method is called when a new communication port is created by one of the task's constituent threads. It enables `TaskCB` to maintain the inventory of ports that belong to the task. If the maximum number of ports for this task has been reached, `FAILURE` should be returned. Otherwise, `SUCCESS` is returned.

◇ `public int do_removePort(PortCB oldPort)`
This method is called when one of the task's communication ports is destroyed. The method should remove the port from the list of ports maintained by `TaskCB`. `SUCCESS` is to be returned if the port belongs to the task; `FAILURE` otherwise.

◇ `public void do_addFile(OpenFile file)`
Adds `file` to the table of open files of the task. The implementation of the table is entirely up to the student. This method is typically called by the method `open()` of class `OpenFile` (indirectly, through the wrapper `addFile()`).

◇ `public int do_removeFile(OpenFile file)`
Removes `file` from the table of open files of the task. This method is typically called by the method `close()` of class `OpenFile`. It returns `SUCCESS` if the file belongs to the task; `FAILURE` otherwise.

Relevant methods and fields defined in this and other packages.
The following public methods and fields of other classes are useful for implementing the methods of the TASKS project.

◇ `public final static float get()` — `HClock`
Returns the current simulation time.

◇ `static public int MaxThreadsPerTask` — `ThreadCB`
Maximum allowed number of threads per task.

◇ `final static public void dispatch()` — `ThreadCB`
Dispatches a new thread.

◇ `public static int MaxPortsPerTask` — `PortCB`
Maximum allowed number of ports per task.

◇ `final public int destroy()` PortCB
Destroys the port on which it is called.

◇ `static public int getVirtualAddressBits()` MMU
Returns the number of bits needed to specify a virtual address. Can be used to determine the size of the swap file.

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

◇ `final public void deallocateMemory()` PageTable
Deallocates (frees) the memory used by the task. Called when a task is terminated. Is invoked on the task's page table.

◇ `public PageTable(TaskCB ownerTask)` PageTable
Page table constructor (should be used with the `new` operator). Used to create a page table object for a newly created task. This object must then be associated with the task using the `setPageTable()` method.

◇ `public final static String SwapDeviceMountPoint GlobalVariables`
The mount point for the swap device in the file system. It is the name of the directory where all swap files live, and is terminated with a slash or a backslash. The name of the task's swap file is `SwapDeviceMountPoint` concatenated with the task ID.

◇ `final public static int create(String name, int size)` FileSys
Here `name` is the *full path name* of the file and `size` is the desired initial size in bytes. The size of a file is assumed to always be a multiple of the disk block size (which is identical to the virtual memory page/frame size). This method returns `SUCCESS` if the file is successfully created and `FAILURE` otherwise. A `create()` operation can fail if, for example, the device does not have enough space.

◇ `final public static void delete(String name)` FileSys
Deletes the file. (See the description of class `FileSys` for more details about this method.)

◇ `final public static OpenFile open(String name,TaskCB task)` OpenFile
Opens the file `name` and returns a file handle for use at run time to read and write the file.

◇ `final public int close()` OpenFile
When invoked on an open file handle, closes the file. Returns `SUCCESS` if the file is successfully closed and `FAILURE` otherwise. A `close()` operation might fail, for example, if the file has outstanding I/O operations.

◇ `final static public ThreadCB create(TaskCB task)` ThreadCB
Creates an active thread for the task supplied as an argument. Returns the created thread.

◇ `final public void kill()` `ThreadCB`
Destroys the thread. Notice that this method calls your implementation of `do_removeThread()` to disassociate the thread from the task.

Summary of Class `TaskCB`

The following table summarizes the attributes of class `TaskCB` and the methods for manipulating them. These attributes and methods are provided by the class `IflTaskCB` and are inherited. The methods appearing in the table are more fully described in Section 3.4.

Identity: The identity of a task is set by the system, but it can be queried with the method `getID()`.

Page table: The page table of a task is set with the method `setPageTable()` and can be retrieved using `getPageTable()`.

Status: The status of a task is handled using the methods `setStatus()` and `getStatus()`.

Priority: The status of a task is handled using the methods `setPriority()` and `getPriority()`.

Current thread: Indicates which thread of a task is currently running. The methods to query and modify this attribute are `getCurrentThread()` and `setCurrentThread()`.

Creation time: The creation time of a task is handled using the methods `getCreationTime()` and `setCreationTime()`.

Swap file: A task's swap file is set and retrieved using the methods `getSwapFile()` and `setSwapFile()`.

Table of open files: Keeps track of all of the open files of a task, which are instances of class `OpenFile`. *OSP 2* does not impose any requirements to how this table is maintained as long as it properly keeps inventory of a task's open files. Two methods are used in conjunction with this table: `addFile()` and `removeFile()`. Calls to these methods by other packages are intended to notify a task as to which files it owns. In addition, when a task is destroyed, all its files must be closed. This is performed as part of the `do_kill()` method, which must iterate through this table and close all the files in it. The `do_`-versions of the `addFile()` and `removeFile()` methods are part of the TASKS project. Note that `TaskCB` never calls these methods—it *implements* them.

Table of ports: Keeps track of all of the communication ports owned by a task. *OSP 2* does not define a specific variable by which to refer to this table, and the internal data structure used to implement it is entirely up to the student. However, the following methods are defined to manipulate this table: `getPortCount()`, `addPort()`, and `removePort()`. The first indicates how many open ports the task has; the second is used to attach a new port to the task; and the last is used to remove destroyed ports. The `do_`-versions of these methods are part of the TASKS project. `TaskCB` *implements* these methods—it never calls them.

Table of live threads: As with ports, *OSP 2* does not prescribe how this table is to be implemented. However, the following methods are defined to manipulate this table: `getThreadCount()`, `addThread()`, and `removeThread()` . The first method counts the number of live threads owned by the task, the second adds newly created threads to tasks, and the third method removes killed threads. The `do_`-versions of these methods are implemented by the student. These methods are *implemented* by `TaskCB`— they are never called by this class.

3.4 Methods Exported by the TASKS Package

The following is a summary of the public methods defined in the classes of the TASKS package or in its superclasses. These methods can be used in the implementation of this or 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 TASKS package, all exported methods belong to a single class, `TaskCB`, which inherits them from the superclass `IflTaskCB`. In general, the public methods exported by a student package may belong to more than one class; see, for example, package MEMORY (Section 5.8).

◇ `final public void setPageTable(PageTable table)` `TaskCB`
Sets the page table of the task.

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

◇ `final public int getStatus()` `TaskCB`
Returns the status of the task. Allowed values are `TaskLive`, for live tasks, and `TaskTerm`, for terminated tasks.

◇ `final public void setStatus(int s)` `TaskCB`
Sets the status of the task.

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

◇ `final public void setPriority(int p)` TaskCB
Sets the priority of the task.

◇ `public ThreadCB getCurrentThread()` TaskCB
Returns the current thread of the task. The current thread is the thread that will run when the task is made current by the dispatcher.

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

◇ `final public int getID()` TaskCB
Returns the ID of the task.

◇ `final public double getCreationTime()` TaskCB
Returns the task creation time.

◇ `final public void setCreationTime(double time)` TaskCB
Sets the task creation time to `time`.

◇ `public final OpenFile getSwapFile()` TaskCB
Returns the swap file of the task.

◇ `public final void setSwapFile(OpenFile file)` TaskCB
Sets the swap file of task to `file`.

◇ `final public int addThread(ThreadCB thread)` TaskCB
Adds the specified thread to the list of threads of the given task.

◇ `final public int removeThread(ThreadCB thread)` TaskCB
Removes the specified thread from the list of threads of the given task.

◇ `final public int getThreadCount()` TaskCB
Returns the number of threads in the task.

◇ `public final void addFile(OpenFile file)` TaskCB
Adds `file` to the table of open files of the task. The implementation of the table is entirely up to the student.

◇ `public final void removeFile(OpenFile file)` TaskCB
Removes `file` from the table of open files of the task.

◇ `final public int addPort(PortCB newPort)` TaskCB
Adds `newPort` to the list of ports associated with the task.

◇ `final public int removePort(PortCB oldPort)` TaskCB
Removes `oldPort` from the list of ports owned by the task.

◇ `final public int getPortCount()` `TaskCB`
Returns the number of ports owned by the task.