9
RESOURCES: *Resource Management*

9.1 Chapter Objective

The objective of the RESOURCES project is to expose students to the concept of shared resources in a concurrent system, and to provide an environment in which they can implement various deadlock-handling techniques. *OSP 2* simulation supports two approaches to handling deadlock in an operating system: **deadlock avoidance** and **deadlock detection**, discussed further below. To this end, students will be asked to implement the three public classes of the RESOURCES package: `ResourceCB`, the resource control block; `RRB`, the resource request block; and `ResourceTable`.

9.2 Overview of Resource Management

The RESOURCES project focuses on techniques for managing shared resources in a concurrent system. Examples of such resources include files, printer, disks, and interprocess-communication messaging buffer space. When processes compete for access to shared resources, especially when such access is exclusive, deadlock becomes an issue. Simply put, deadlock arises when there exists a closed chain of processes such that each process holds at least one resource needed by the next process in the chain. This phenomenon is known as **circular wait**.

For circular wait to exist it must be the case that processes require:

Mutual Exclusion. Mutually exclusive access to resources;

Hold and Wait. A process may hold allocated resources while awaiting assignment of others; and

No Preemption. No resource can be forcibly removed from a process holding it.

Clearly deadlock is an undesirable situation since if it is not dealt with properly the processes involved in the deadlock will wait forever, without furthering their execution. There are three main techniques for dealing with deadlock in an operating system:

Deadlock Prevention. Design the system in such a way that the possibility of deadlock is excluded. This can be accomplished by constraining resource requests to prevent one of the four conditions of deadlock. For example, the hold-and-wait condition can be prevented by requiring that a process request all of its resources at one time and blocking the process until all requests can be granted simultaneously.

Deadlock Avoidance. With this technique, a decision is made dynamically whether the current resource request will, if granted, potentially lead to a deadlock; if so, the request is denied. Deadlock avoidance thus requires knowledge of future process resource requests. A primary approach to deadlock avoidance utilizes the **Banker's algorithm**. The idea here is to determine if the current allocation of resources to processes represents a *safe state*: one in which there is at least one sequence of process resource requests that does not result in a deadlock; i.e. all of the processes can be run to completion.

Deadlock Detection. Resource request are granted to processes whenever possible. Periodically, the operating system executes an algorithm that checks if deadlock (circular wait) exists. If so, a *recovery strategy* is undertaken, namely one of the following.

- ◇ Abort all processes.
- ◇ Back up each deadlocked process to some previously defined checkpoint and restart all processes.
- ◇ Successively abort deadlocked processes until deadlock no longer exists.
- ◇ Successively preempt resources until deadlock no longer exists.

9.3 Overview of Resource Management in *OSP 2*

OSP 2 provides simulation support for deadlock avoidance and deadlock detection. This means that it understands the semantics of each of these two types of deadlock handling and provides appropriate error-checking facilities. For instance, in deadlock avoidance, a deadlock created after granting a resource allocation request to a process is considered an error, while in deadlock detection it is not.

The class `ResourceCB` does the bulk of the work. It represents the **resource control block**, where much of the information about the available resources is maintained. Resources are divided into **resource types**, where each resource type can have several **resource instances**. Each resource type is represented by a distinct resource control block.

A thread might issue a request to **acquire** a given number of instances of a particular resource type, but it does not care which particular resource instances are given to it as long as the instances are of the requested type. When such a request arrives, the operating system (which is part of the student code in class `ResourceCB`) must decide whether to grant the request, abort (kill) the requesting thread, or block the thread until its request is granted at some future time. This decision depends on the current state of resource allocation and on the deadlock-handling method (detection or avoidance) in use.

The class `RRB` represents **resource request blocks**. An RRB contains information about one outstanding request for one particular resource type issued by a particular thread. An `RRB` object is also an `Event` object (Section 1.6). When a thread issues a request that cannot be granted, the thread is suspended on the RRB associated with this request. Subsequently, when the needed resources become available, a `notifyThreads()` operation issued on that RRB will eventually wake up the thread.

The **resource table** is represented by the class `ResourceTable`; it is represented as an array of `ResourceCB` objects and lists all resource types available in the system. In *OSP 2*, all resource types are created at the beginning of simulation and no new resources are added or deleted afterwards. The total number of instances of each resource type remains constant as well. However, the number of *available* resource instances changes as processes acquire and release them.

Resource types are identified by a resource ID, a number between 0 and the resource table size, which is determined using the static method `getSize()` of class `ResourceTable`.

We will now describe the classes of package RESOURCES in detail. Figure 9.1

depicts the relationship these classes have with the other classes in the *OSP 2* system.

9.4 Class `ResourceTable`

This class is the simplest of them all: only a constructor is required. You can add other methods and variables to support your implementation of the project, but these would be specific to your particular design.

◇ `public ResourceTable()`
Calls `super()` and might do additional initialization, if the student implementation defines additional fields in this class.

OSP 2 provides the following built-ins that you will use to implement other classes in this project:

◇ `public static final ResourceCB getResourceCB(int resourceID)`
Since resource types are identified using their numeric IDs, this method lets you visit, in a loop, the resource control block of every resource type in the system.

◇ `public static final void getSize(int size)`
Returns the size of the resource table, which is also the number of resource types available in the system.

Built-ins and relevant methods defined in other classes. Since this class has only its constructor, your implementation will not use any methods provided by other *OSP 2* classes.

Summary of Class `ResourceTable`

This class is intended to maintain the resource table of the system. A resource table is simply a fixed-size array of resource objects. This size can be queried using the method `getSize()`. In addition, resource objects can be retrieved from the table using the `getResourceCB()` method, as described earlier.

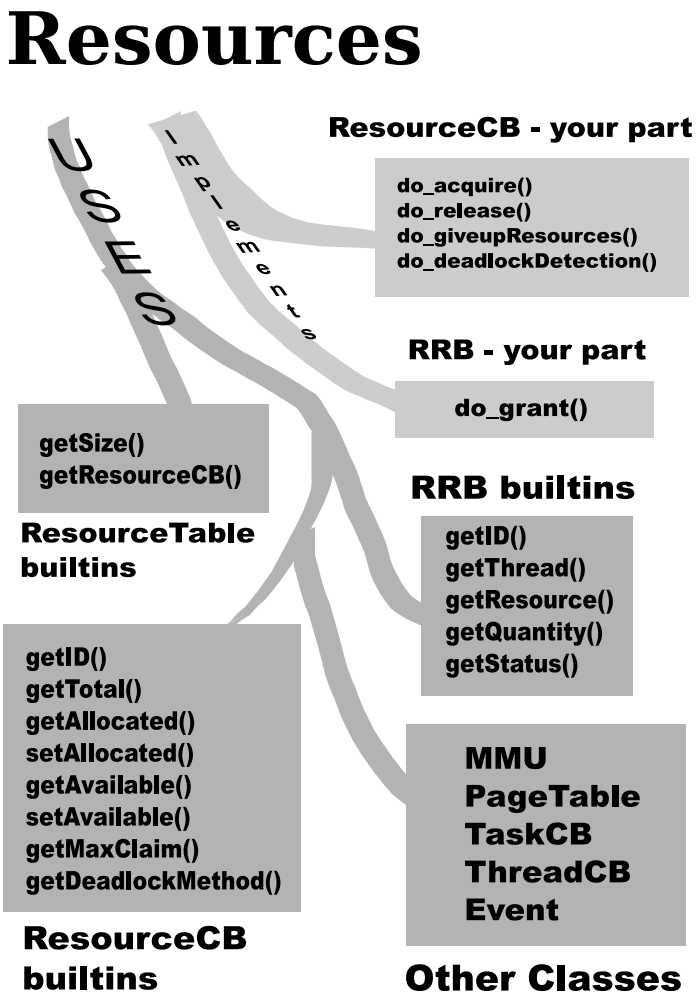

Figure 9.1 A diagram summarizing the package RESOURCES.

9.5 Class RRB

This class represents the resource request block, which threads use to specify their requests to the system. It is declared as follows:

◇ `public class RRB extends IflRRB`

Note that `IflRRB` extends class `Event`, which makes it possible to treat RRB objects as events. In particular, threads can be suspended on an RRB object and later resumed.

An `RRB` object includes the following information:

- ◇ The *ID*, which can be obtained with the help of the method `getID()`.
- ◇ The *thread* that issued the request; it can be obtained using the method `getThread()`.
- ◇ The *resource type* involved in the request. Its control block can be obtained using the method `getResource()`. Only one resource type can be requested using an RRB.
- ◇ The *quantity* of the requested resource. It is obtained by calling the method `getQuantity()`.
- ◇ The *status* of the RRB. The status can be one of these constants defined by *OSP 2*: `Denied`, `Suspended`, `Granted`. The status is `Denied` when the system denies the request (because, for instance, the thread wants more resource instances than the total that the system has); it is `Suspended` if the system decides that the resource request cannot or should not be granted now, but can be in the future; when the request is granted, the status is set to `Granted`. Two methods are used to manipulate the status of an RRB: `getStatus()` and `setStatus()`.

The class `RRB` contains only two methods that need to be implemented by the student:

- ◇ `public RRB(ThreadCB thread,ResourceCB resource,int quantity)`
  This is the class constructor. The first statement in this constructor must be `super(thread, resource, quantity)`, but the rest depends on your program design.
- ◇ `public void do_grant()`
  This method is used to grant the RRB on which it is invoked. Note that `do_grant()` does not make any *decision* on whether to grant or not. This decision is made elsewhere, as described later in this chapter. Thus, this method does bookkeeping only. In particular, it decrements the number of available instances of the requested resource by the requested quantity and increments the number of allocated instances of this resource by that same quantity. The current number of available instances of a resource is given by the method `getAvailable()` and is set by the method `setAvailable()`. Similarly, the number of allocated resources is obtained and changed using the methods `getAllocated()` and `setAllocated()`, respectively.

To finish granting the request, the status of the RRB must be set to `Granted` and the thread that was waiting on this RRB should be resumed. The latter is done by invoking the method `notifyThreads()` of class `Event` (recall that a RRB is also an `Event` object).

Built-ins and relevant methods defined in other classes. The implementation of the methods in the `RRB` class relies on the following methods provided by other classes (or inherited from the superclasses of `RRB`):

◇ `final public int getStatus()` `RRB`
Returns the status of the RRB: `Denied`, `Suspended`, or `Granted`.

◇ `final public void setStatus(int value)` `RRB`
Sets the status of the RRB to `Denied`, `Suspended`, or `Granted`.

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

◇ `final public int getQuantity()` `RRB`
Returns the quantity of the resource requested by the thread that issued the request.

◇ `final public ThreadCB getThread()` `RRB`
The thread that issued the request.

◇ `final public ResourceCB getResource()` `RRB`
The resource for which the request was issued.

◇ `public final int getAvailable()` `ResourceCB`
Returns the number of free instances of this resource type.

◇ `public final void setAvailable(int value)` `ResourceCB`
Sets the number of free instances of this resource type.

◇ `public final int getAllocated(ThreadCB thread)` `ResourceCB`
Returns the number of allocated instances of this resource type.

◇ `public final void setAllocated(ThreadCB thread,int value)` `ResourceCB`
Sets the number of allocated instances of this resource type.

Summary of Class RRB

The class `RRB` is intended to maintain the information about requests that were issued by the various threads for the non-shareable resources that are provided

by the system. As mentioned earlier, an `RRB` object has the following attributes: *ID*, *thread*, *resource type*, the *quantity* of the requested resource type, and the *status* of the request. These attributes can be queried and manipulated using the methods described earlier in the section.

9.6 Class `ResourceCB`

This class does most of the work. In particular, this is where the deadlock detection and avoidance algorithms are implemented. The deadlock-avoidance algorithm is invoked by the `do_acquire()` method, while deadlock detection is the responsibility of the method `deadlockDetection()`, which is invoked periodically by *OSP 2*. The `ResourceCB` class is declared as follows:

◇ `public class ResourceCB extends IflResourceCB`

Most textbooks describe deadlock avoidance and detection algorithms in terms of the various resource allocation and resource request matrices, which are used for keeping track of the current state of system resources. This all looks simple enough, except for one important point: textbook algorithms all assume that all the threads and resource types are known in advance, so they represent the matrices as two-dimensional arrays. In a real system, neither resources, nor threads are static: they come and go and their total number cannot be assumed to be bounded by a known constant. Therefore, matrices used by the *real-life* deadlock-handling algorithms cannot be represented as two-dimensional arrays.

In *OSP 2*, the number of resource types is fixed, which simplifies things a bit. However, the number of threads that can potentially request resources is not known and cannot be estimated. Thus, using two-dimensional arrays for representing resource allocation and request matrices is also out of the question: you must come up with another suitable data structure. Since most operations in deadlock-detection and -avoidance algorithms reference the matrix elements via a specific resource and/or thread, your data structure must provide efficient access to the matrix elements using either of these keys. For instance, if you have to scan arrays and compare their entries to a particular thread ID or resource, it is a sure sign that you have chosen a bad data structure.

One good data structure in this case would be an array of hash tables, where each hash table represents all requests made by the various threads for a particular resource type. Since Java hash tables are dynamic, they provide exactly what the doctor ordered for this particular problem.

◇ `public ResourceCB(int qty)`

This is the required class constructor. It must have `super(qty)` as its first

statement, but the rest depends on your program design.

◇ `public static void init()`
As in other student modules, this method is called by the simulator at the beginning of simulation. It can be used to initialize the static variables and structures that you might use in your implementation.

◇ `public RRB do_acquire(int quantity)`
This method is typically invoked by an *OSP 2* thread on a given resource type (represented by a `ResourceCB` object) in order to obtain `quantity` instances of that resource type. To determine which *OSP 2* thread has issued the request, the following method can be used. First, the current task can be found from the *page table base register*, or PTBR; see Section 5.2 for more information on this subject. The value of the PTBR is the page table of the currently running task. In *OSP 2*, the value of the PTBR is obtained using the static method `getPTBR()` of class `MMU`, and the current task can be obtained from a page table via the method `getTask()`.

Next, you have to create an RRB that describes the request. What follows depends on whether the simulator is in deadlock-avoidance or deadlock-detection mode (which is determined by an input simulation parameter that you might have spotted in the GUI window). To find out which mode is in effect, use the method `getDeadlockMethod()`.

If the deadlock-handling method is `Detection`, there are three possibilities. If the system has enough available instances of the requested resource, the request is granted immediately by executing the method `grant()` on the RRB. If the requested number of instances cannot be granted under *any* circumstances (e.g., because the total number of instances of the requested resource type that are either held or requested by the given thread exceeds what the system has), then `null` is returned. If the requested number of instances cannot be granted immediately (but might be in the future, if all other threads release their resources) then the requesting thread must be suspended on the RRB and the RRB's status should be set to `Suspended`. The RRB status is set using the method `setStatus()`, while threads are suspended using the `suspend()` method of class `ThreadCB`. Recall that an RRB is an `Event` object as well, so in order to suspend a thread on an RRB, the RRB must be passed as a parameter to `suspend()`. Read more about thread suspension and resumption in Section 4.3.

If the deadlock-handling method is `Avoidance`, then you must use a deadlock-avoidance algorithm, such as the Banker's algorithm. If this algorithm says that it is safe to grant the request, the RRB is granted. Otherwise, the thread is suspended and the RRB status is set to `Suspended` as well.

When a thread is suspended inside `do_acquire()`, its execution is paused until the request is granted (possibly as a result of a `release()` operation on the same resource or of `giveupResources()` operation, which is invoked when a thread is killed), and the thread is resumed. Whether the RRB is granted immediately or the thread is suspended, `do_acquire()` returns the RRB that was created earlier in order to represent the request.

◇ `public void do_release(int quantity)`
This method might be invoked by an *OSP 2* thread on a given resource type (represented by a `ResourceCB` object) in order to release `quantity` instances of that resource type.

As with `do_acquire()`, you first must find the thread that issued the `release()` request. Then the state of the resource allocation should be updated appropriately in order to reflect the new number of free resources and the new allocation of the given resource to the thread. Note that the thread might release some, but not all, instances held for this resource type. The exact details depend on your representation of the resource-allocation state, but this would typically involve the methods `setAllocated()`, `setAvailable()`, `getAvailable()`, etc.

This is not all, however. Since new resources became available after the release operation, it is possible that some of the previously suspended requests can now be granted. In order to be able to determine whether this is the case, one needs to keep track of the RRBs that were previously suspended in `do_acquire()`. Once a grantable RRB is found, it should be granted (using the `grant()` method) and the thread waiting on that RRB is resumed (resumption is done by method `grant()`).

◇ `public static Vector do_deadlockDetection()`
If the simulation method is `Detection`, this method will be periodically called by *OSP 2* in order to test your implementation of the deadlock-detection algorithm. This method should first check if a deadlock exists and, if so, remove it. Your instructor might have imposed specific requirements on your implementation of deadlock detection and recovery, and *OSP 2* adds its own.

First, there should be no deadlocks left after `do_deadlockDetection()` returns. The result returned by this method should be a vector of `ThreadCB` objects that were found to be involved in a deadlock. *OSP 2* will compare this list with its own and will issue an error if the two lists differ. If no deadlock exists, `null` should be returned.

You can use any textbook deadlock-detection algorithm that can detect deadlocks in the presence of multiple instances per resource type. (For instance,

cycle detection in a wait-for graph would *not* be a suitable algorithm for this purpose.)

Deadlock recovery is done by killing some or all of the threads involved in the deadlock. However, *OSP 2* insists that threads must not be killed unnecessarily. This means that no thread should be killed unless it is deadlocked and, in addition, if the deadlock is gone after killing of *some* deadlocked threads, then no further thread destruction should occur.[1]

Threads are killed using the `kill()` method of class `ThreadCB`. Note that when a thread is killed, it releases its resources by calling `do_giveupResources()` (described next). As in the case of the `do_release()` method, this creates an opportunity for granting a previously suspended RRB and resuming the associated thread. See the description of `do_release()` to learn how to do this.

◇ `public static void do_giveupResources(ThreadCB thread)`
This method is called in order to release all resources previously allocated to `thread`, and it happens when `thread` is terminated. You will never need to call this method in this project. Instead, your implementation of this method is made available to *other OSP 2* modules, which will call `do_giveupResources()` when necessary. This method should go over the resources allocated to the given thread and update the number of the available instances of such resources accordingly. The number of resources allocated to the thread should also be adjusted (to 0).

Since the thread releases its resources, the system might have enough free resources to unblock some suspended RRBs. Therefore, as in the case of `do_release()`, it is necessary to check the suspended RRBs and grant those that are grantable.

Built-ins and relevant methods defined in other classes. The following methods and fields, which are defined in other classes or are provided by the superclasses of `ResourceCB`, might be used in the implementation of the class `ResourceCB`.

◇ `public final int getID()` `ResourceCB`
Returns the ID of the resource.

◇ `public final int getTotal()` `ResourceCB`
Returns the total number of instances (free plus allocated) for this resource type.

[1] Note that if N threads are involved in the deadlock, then killing any $N-1$ of them will eliminate the deadlock. But often the deadlock can be eliminated by killing fewer than $N-1$ threads.

◇ `public final int getAllocated(ThreadCB thread)` `ResourceCB`
Returns the number of allocated instances of this resource type.

◇ `public final void setAllocated(ThreadCB thread,int value)` `ResourceCB`
Sets the number of allocated instances for this resource type.

◇ `public final int getAvailable()` `ResourceCB`
Returns the number of free instances of this resource type.

◇ `public final void setAvailable(int value)` `ResourceCB`
Sets the number of free instances for this resource type.

◇ `public final int getMaxClaim(ThreadCB thread)` `ResourceCB`
Returns the maximal number of instances of this resource type that can ever be acquired by the given thread. Used for deadlock avoidance only.

◇ `public final static int getDeadlockMethod()` `ResourceCB`
Returns the deadlock-handling method currently in effect: `Avoidance` or `Detection`.

◇ `public final static int getSize()` `ResourceTable`
Returns the size of the resource table. This value is also equal to the number of different resource types in *OSP 2*.

◇ `public static final ResourceCB getResourceCB(int resourceID)`
Given an index into the resource table, returns the `ResourceCB` object in that table cell. This method makes it possible to visit the resource control block of each resource type in a loop.

◇ `static public PageTable getPTBR()` `MMU`
Returns the value of the page table base register, which is either `null` or the page table of the currently running task.

◇ `public final TaskCB getTask()` `PageTable`
Indicates which task owns the given page table. In RESOURCES, this method is used to determine the thread that issued the request.

◇ `public ThreadCB getCurrentThread()` `TaskCB`
Returns the running thread of the currently running task.

◇ `public RRB(ThreadCB thread, ResourceCB resource, int quantity)` `RRB`
A constructor for creating resource request blocks with the given parameters.

◇ `public final void grant()` `RRB`
Grants the request represented by this RRB.

◇ `final public void setStatus(int value)` `RRB`
Sets the status of the RRB to `Denied`, `Suspended`, or `Granted`.

◇ `final public ThreadCB getThread()` RRB
The thread that issued the request represented by this RRB.

◇ `final public ResourceCB getResource()` RRB
The resource for which the request was issued.

◇ `final public int getQuantity()` RRB
Returns the quantity of the resource requested by the thread that issued the request.

◇ `final public void suspend(Event event)` ThreadCB
Suspends the thread on which this method is called and puts the thread on the waiting queue of `event`.

◇ `final public void kill()` ThreadCB
Kills this thread. Note that this will cause the thread to release its resources, which in turn might make some previously suspended RRBs grantable.

◇ `final public int getStatus()` ThreadCB
Returns the status of the thread. See Section 4.3 for more information on the different states of a thread. In this project you might need to know that killed threads have status `ThreadKill`. If such a thread shows up in a resource-allocation matrix or elsewhere, you might want to delete or skip it in your algorithms.

◇ `public void notifyThreads()` Event
Resumes all threads that might be waiting on this event. In the case of package RESOURCES, the event would be an RRB and the single resumed thread would be the thread that issued the corresponding request.

Summary of Class `ResourceCB`

Instances of this class are used to represent individual non-shareable resources in the system. An individual resource has the following attributes:

ID: The identity of the resource. This parameter can be retrieved using the built-in `getID()`.

Total number of instances: This attribute describes the total number of instances of the resource that exist in the system. It can be obtained using the built-in method `getTotal()`.

Number of allocated instances: The number of instances of the resource that are currently allocated to a given thread. This parameter can be retrieved using the method `getAllocated()` and changed using the method `setAllocated()`.

Number of free instances: This parameter represents the number of free instances of the resource. It can be obtained by calling the built-in method `getAvailable()` and changed using the method `setAvailable()`.

Maximum number of claimable instances: This parameter represents the maximal number of instances of a resource that can possibly be acquired by a single thread.

9.7 Methods Exported by the RESOURCES Package

Only one method defined in this package is used by other modules:

◇ `public static void giveupResources(ThreadCB thread)` `ResourceCB`
Called by terminating threads in order to release the abstract shared resources held by that thread.