6
DEVICES: *Scheduling of Disk Requests*

6.1 Chapter Objective

The objective of project DEVICES is to teach students about device I/O and, relatedly, certain aspects of device drivers. One main focus will be the scheduling of disk I/O requests. To meet these objectives, students will be asked to implement the three public classes of the DEVICES package: `Device`, `IORB`, and `DiskInterruptHandler`. The class `Device` deals with the scheduling of I/O requests, `IORB` implements the I/O Request Block data structure, and `DiskInterruptHandler` constitutes the interrupt handler for I/O devices.

6.2 Overview of I/O Handling

I/O supervisor and I/O Request Block. When the user thread issues a `read()` or `write()` system call, the OS assembles an **input/output request block** (or **IORB**) and passes the request to the basic I/O supervisor: the portion of the operating system responsible for managing the various I/O devices of the system. The IORB includes information about the thread that issued the call; the buffer page in main memory that contains the data to be written out or into which the data is to be copied from the secondary storage; the disk block to which the buffer data is to be written out or which contains the data to be read in; and the I/O device that is the target of the requested

I/O operation.

The I/O supervisor examines the IORB and places it on the **device queue** of the targeted device. A device queue is nothing more than a queue of waiting-to-be-serviced IORBs, one such queue per I/O device in the system.

Disk interrupt handler. When the device finishes servicing an I/O request, a device interrupt occurs, which is the way by which external devices notify the CPU about completion of an I/O operation. The eventual result of an I/O interrupt is that the appropriate device interrupt handler is called. In *OSP 2* the only external devices are disks, so the only device interrupt handler is the disk interrupt handler.

A disk interrupt handler performs a variety of functions, which we will describe in detail in Section 6.5. One of these is to invoke the I/O scheduler, which chooses the IORB to be serviced next, assuming the device queue is non-empty; i.e. contains at least one IORB. Once an IORB has been selected, it is dequeued from the device queue and the device is instructed to process the request. If the device queue is empty, the device simply idles.

Disk-scheduling algorithms. A variety of disk-scheduling policies have been proposed for use by the I/O scheduler. Many of these policies are concerned with performance and QoS (Quality of Service) issues related to the physical characteristics of a disk device. Such a device is typically configured as a number of platters, each of which has an upper and lower surface on which data can be magnetically encoded. A surface consists of a number of concentric tracks each of which is divided into storage regions known as sectors. For most disk drives, a fixed sector size of 512 bytes is used. The block size of a disk is the number of bytes transferred in a single I/O operation, and is usually a multiple of the sector size. The preceding discussion therefore tells us that a disk address consists of a surface number, track number, and sector number.

Each surface has its own disk arm, at the end of which is a read/write head that must be positioned over the appropriate track for an I/O operation to occur. The arms are attached to the disk-drive boom, which moves the arms in unison back and forth over the tracks of the various surfaces. This gives rise to the concept of the disk cylinder: the collection of tracks carved out of 3-space by virtue of having all read/write heads positioned over the same-numbered track on all surfaces.

Disk I/O can be slow compared with say the time it takes the CPU to access main memory due to the electromechanical aspects of disk operation. In particular, having to position the disk arm over the correct track before an I/O can take place is the biggest culprit. The time taken by this movement is called

the **seek time** and many proposed disk-scheduling strategies seek to minimize this delay. **Rotational delay**, the time spent waiting for the proper sector to circle under the read/write head, is another overhead of disk I/O but of much less concern to us since it is about an order of magnitude smaller than the seek time.

Some of the most well-known disk-scheduling algorithms are:

Shortest Seek Time First (SSTF) Services the IORB that requires the least movement of the disk arm from its current position.

SCAN The arm is moved in one direction only, satisfying all outstanding requests en route until it reaches the last track in that direction. The service direction is then reversed and the scan proceeds in the opposite direction.

LOOK A variant of SCAN where the service direction is reversed when there are no more requests in the current service direction (rather than proceeding to the last track).

C-SCAN A variant of SCAN which restricts scanning to one direction only. When the last track has been visited in that direction, the arm is returned to the opposite end of the disk and the scan begins again.

C-LOOK The variant of LOOK in which scanning is restricted to one direction, just as in C-SCAN.

Priority Unlike the above algorithms, this approach is not intended to optimize disk utilization but rather to meet other system objectives. For example, it may give priority to IORBs coming from interactive processes rather than those from computationally intensive batch jobs, with the goal of providing good interactive response time.

It is important to note that *OSP 2* disks do not support the command that moves the read/write head to a specified cylinder without starting an I/O. The head moves only when the `startIO()` command is issued. It is therefore not possible to implement strategies such as SCAN and LOOK which require the head to be moved to the first and last cylinders even when there are no outstanding I/O requests to these cylinders.

Synchronous versus asynchronous I/O. To conclude our discussion of disk-scheduling strategies, let us consider a relatively new disk-scheduling algorithm that has exhibited superior performance on web-server and file-system benchmarks. *Anticipatory scheduling* is useful in the context of synchronous I/O, where a thread that has issued an I/O operation is blocked until the I/O completes. In contrast, with asynchronous I/O, a thread initiates an I/O operation and then can continue processing while the I/O request is fulfilled.

I/O in *OSP 2* is of the synchronous variety, although it is possible to simulate asynchronous I/O in *OSP 2*; see Section 7.7.

Seek-optimizing algorithms can get confused by synchronous I/O since in this case threads issue one I/O request at a time (after the previous request has finished). Thus the scheduler may incorrectly assume that the thread issuing the last I/O request has momentarily no further I/O requests and therefore selects a request from another thread. This is a bad decision on the part of the scheduler if the current thread is requesting data sequentially positioned on the disk which is often the case in practice. Anticipatory scheduling seeks to mitigate this problem by issuing a short, controlled delay period before selecting the next IORB to be serviced. This allows the thread that issued the last request to issue additional requests before the next scheduling decision is made.

In the rest of this section we provide a detailed description of the classes that comprise the package DEVICES. Figure 6.1 places these classes in the overall context of *OSP 2*.

6.3 Class `IORB`

Before discussing the workings of the I/O supervisor, we need to look more closely at the structure of an IORB. The class for this data structure is defined as follows:

◇ `public class IORB extends IflIORB`

and its only mandatory method has a six-argument constructor:

◇ `public IORB(ThreadCB thread, PageTableEntry page, int blockNumber, int deviceID, int ioType, OpenFile openFile)`
As usual for *OSP 2* class constructors, the first thing this class does is call `super()` with the same set of arguments. The rest depends on your implementation. For instance, if you define additional fields in this class, you can initialize them in the constructor.

As follows from the argument list, an IORB keeps information about the thread which issued the request, the buffer page involved, the device and the device's block that contains the data to be read in or on which the page is to be written out, the type of I/O operation (which can be either `MemoryRead` or `MemoryWrite`, two predefined constants in *OSP 2*), and the open-file handle. The last of these arguments will be defined in more detail in Chapter 7. For now it suffices to know that an open-file handle contains runtime information, such as the file size and the list of blocks allocated to the file, which the OS

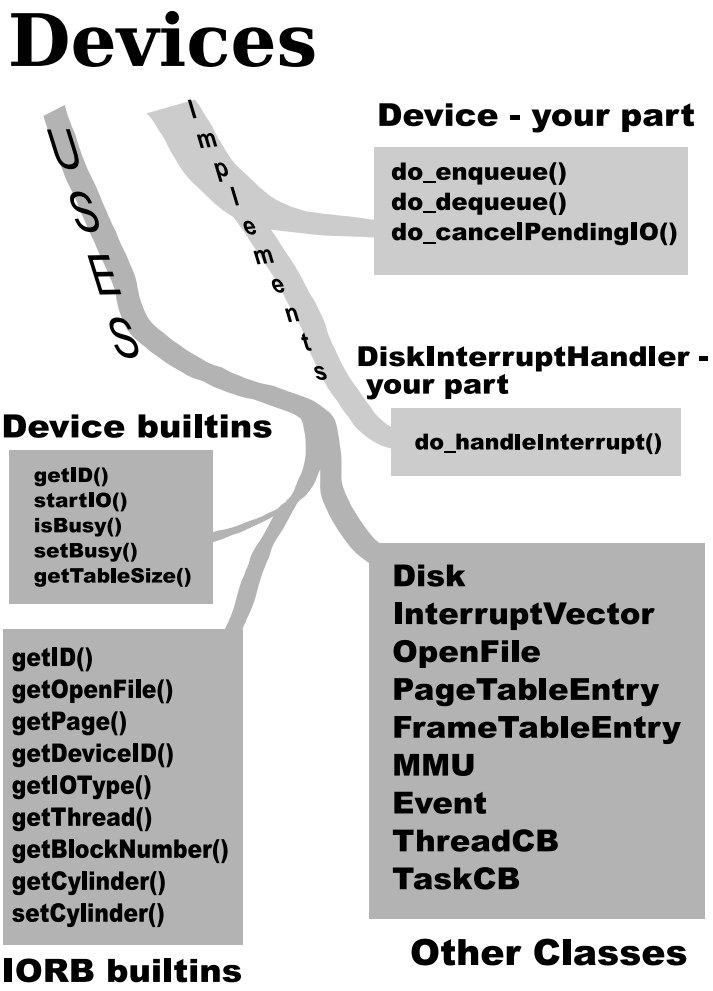

Figure 6.1 An overview of the package DEVICES.

needs in order to process I/O operations on that file. This handle comes from one of the parameters of the `read()` or `write()` system call that created the IORB in question.

It is important to keep in mind that `IORB` is also a subclass of `Event` so threads can wait on it and be notified. See Section 1.6 to refresh your memory about *OSP 2* events.

Relevant methods defined in other classes. Typically, implementation of this class does not use methods provided by other *OSP 2* classes, since the only mandatory method in this class is the constructor and the components of an `IORB` can be queried via the built-ins provided by *OSP 2* itself (see below).

Summary of Class `IORB`

This class defines the `IORB` data structure that is used to maintain the information about all the active I/O operations. The following API can be used to query an IORB. All these methods are built-ins that apply to an IORB object and they return the components of that IORB as described below.

◇ `final public int getID()`
Provides the Id of the IORB.

◇ `final public OpenFile getOpenFile()`
Returns the open-file handle associated with the IORB.

◇ `final public PageTableEntry getPage()`
Returns the buffer page in main memory, which is the source (in the case of `write()`) or the target (in the case of `read()`) of the I/O operation in question.

◇ `final public int getDeviceID()`
Returns the device involved in the I/O operation.

◇ `final public int getIOType()`
Returns the I/O type represented by the IORB. *OSP 2* supports two types: `FileRead` and `FileWrite`.

◇ `final public ThreadCB getThread()`
Returns the thread that requested the I/O.

◇ `final public int getBlockNumber()`
Returns the block number of the device, which is the source (in the case of `read()`) or the target (in the case of `write()`) of the I/O.

◇ `public final void setCylinder(int cylinder)`
Sets the target disk cylinder for this IORB to `cylinder`. This is done in `do_enqueueIORB()` in `Device`. This method is used by *OSP 2* to make sure that both it and the student module calculate the cylinders associated with IORBs the same way.

◇ `public final int getCylinder()`
Returns the target cylinder associated with this IORB. Since the IORB

cylinder is set in `do_enqueueIORB()`, `getCylinder()` can be used only in `do_dequeueIORB()`.

6.4 Class `Device`

This class implements the I/O scheduler and performs other functions, such as starting I/O operations on devices. The following methods are part of the project and must be implemented by the student.

◇ `public static void init()`
This method is called at the very beginning of the simulation and can be used to initialize static variables that might exist in the student program.

◇ `public Device(int id, int numberOfBlocks)`
This is the class constructor. It must call `super(id,numberOfBlocks)` and then initialize the device object. One thing that requires initialization is the variable `iorbQueue` described later in this section.

◇ `public int do_enqueueIORB(IORB iorb)`
This method is executed on a device object and puts `iorb` on the waiting queue of that device. When programming this method, however, you must first perform several tasks before the enqueueing. First, you must lock the page associated with the `iorb` using the `lock()` method of class `PageTableEntry`. This is done in order to ensure that the page will not be swapped out from now till the end of the I/O operation. (If this page is not currently in main memory, `lock()` will cause a pagefault, which will eventually bring the page into main memory.)

Second, you must increment the IORB count of the open-file handle associated with `iorb`. This is accomplished using method `incrementIORBCount()` of class `OpenFile`. Because different threads can issue I/O operations concurrently on the same file, *OSP 2* needs to maintain a count of IORBs that are active for each open-file handle. Knowing the count allows it to ensure that files cannot be closed before all the outstanding I/O operations have finished. (Closing a file deallocates its file handle, which can cause havoc since outstanding IORBs for this file reference the handle.)

Third, you must set the `iorb`'s cylinder, using method `setCylinder()`, to the cylinder that contains the disk block mentioned in the IORB.

You are now ready to enqueue the IORB but not before you check that the thread that requested the I/O is still alive (using the `getStatus()` method

of class `ThreadCB`), i.e., its status is not `ThreadKill`. If the thread has died, method `do_enqueueIORB()` should return `FAILURE`.

If the thread is alive and the device is idle (you can check for idleness by executing the method `isBusy()` on the device), you can start the I/O operation immediately using the method `startIO()` on the device object and passing it the `iorb` as a parameter. The method `do_enqueueIORB()` should then return `SUCCESS` and exit.

If the device is busy, then put the `iorb` on the device queue and exit by returning `SUCCESS`. The device queue is represented by the variable `iorbQueue` that can take any object that implements the type `GenericQueueInterface` (Section 1.5), as described later in this section.

Disk I/O scheduling is typically implemented as part of this method as the different scheduling strategies work best with differently structured device queues. For instance, for the C-SCAN strategy, the IORBs in the queue might need to be ordered according to the cylinder numbers that contain the requested disk blocks. In this case, sorting would be best done when IORBs are enqueued.

◇ `public IORB do_dequeueIORB()`
This method selects an IORB from the device queue according to some scheduling strategy, deletes it from the queue, and returns the selected IORB. If the queue is empty, `null` is returned.

The I/O scheduling strategy (or parts of it) can also be implemented in this method, because ultimately it is this method that chooses the requests to be serviced. *OSP 2* does not mandate any particular way of implementing scheduling.

Note that you should *not* unlock the page used by the dequeued IORB. This is because the device has not finished servicing the IORB, so the page must stay locked. It will eventually be unlocked when the device finishes servicing the request and the device interrupt occurs.

◇ `public void do_cancelPendingIO(ThreadCB thread)`
The purpose of this method is to iterate over the device queue removing all IORBs initiated by `thread`. The need to do this arises when a thread is killed. This prevents the device from servicing requests that nobody wants any more.

For each IORB associated with `thread` found in the queue, you must unlock the buffer page used by that IORB. Indeed, when the IORB was enqueued, the corresponding page was locked. Normally it would be unlocked in the device interrupt handler after the request is serviced. However, since you

are removing the IORB from the device queue, this request will never be serviced, so you must unlock the page here.

In addition, you must decrement the IORB count of the open-file handle associated with the IORB. Again, normally this is done in the device interrupt handler, but because the IORB in question will never be serviced, you must decrement the count here.

Finally, you should *try* to close the open-file handle associated with the IORB. To understand why, let us consider what happens when a thread is trying to issue a `close()` system call on a file handle. If the handle does not have associated IORBs, the file is closed and the handle is deleted. However, if there are outstanding IORBs for the handle, the system sets the `closePending` flag for that handle, but does not close the file in order to allow the outstanding I/O requests to execute.[1] When all such I/O requests have finished, the file is closed. One of the places where the `closePending` flag should be checked is in the `do_cancelPendingIO()` method. Indeed, if the file was not closed due to outstanding I/O requests and now you are canceling all the outstanding IORBs belonging to `thread`, it is possible that the file handle has no remaining IORBs, so it can be closed. In other words, when removing an IORB associated with `thread` you must check the `closePending` flag of the open-file handle of the IORB. If it is set to *true* and the count of IORBs for this handle has become 0, the file handle must be closed with the `close()` method of `OpenFile`. To check the current count of pending IORBs for a file handle use the method `getIORBCount()` of class `OpenFile`.

How to compute a cylinder from a block. Many scheduling strategies require you to compute a cylinder from a given block number. To do this, you first need to compute the number of blocks in a track.

A track consists of a number of blocks, which in turn consists of a number of sectors. To find the block size, you can use the functions `getVirtualAddressBits()` and `getPageAddressBits()`, since the size of a disk block equals the size of a main-memory page. The block size together with the sector size (`getBytesPerSector()`) gives the number of sectors in a block.

The number of blocks per track can be used to compute the track that holds the given block. To compute the cylinder number corresponding to the block you need to know the number of tracks per cylinder. In *OSP 2* we assume

[1] You may have faced this issue while implementing the `kill()` method of `TaskCB`, which destroys a task. One job that this method is tasked with is closing all open files owned by the task. You may have experienced the unexpected effect of the `close()` system call where some open-file handles stayed around after being closed. The reason for this was the presence of outstanding IORBs.

that each disk platter is one sided, so the number of tracks in a cylinder equals the number of platters in the disk. The latter is obtained using the method `getPlatters()`.

Relevant methods defined in other classes. The following methods defined in other modules are used by the methods in class `Device`.

◇ `public final int lock(IORB iorb)` — `PageTableEntry`
When executed on a page object, this methods locks that page in main memory, so it cannot be swapped out.

◇ `public final void unlock()` — `PageTableEntry`
Unlocks the page that was previously locked by the `lock()` method.

◇ `final public void incrementIORBCount()` — `OpenFile`
Increments the count of IORBs active for the given file handle.

◇ `final public void decrementIORBCount()` — `OpenFile`
Decrements the IORB count for the given file handle.

◇ `final public int getIORBCount()` — `OpenFile`
Returns the current IORB count for the open-file handle.

◇ `final public void close()` — `OpenFile`
Closes the open-file handle.

◇ `final public int getStatus()` — `ThreadCB`
Returns the status of a thread. In this case you need to know when a thread is killed. The status of a killed thread is `ThreadKill`.

◇ `static final public int getVirtualAddressBits()` — `MMU`
Returns the number of bits used to specify a virtual address.

◇ `static final public int getPageAddressBits()` — `MMU`
Returns the number of bits used to specify a page address. From this and the number of bits in a virtual address one can compute the size of a memory page (and of a disk block).

◇ `public final void setCylinder(int cylinder)` — `IORB`
Sets the cylinder of the IORB to `cylinder`.

◇ `public final int getCylinder()` — `IORB`
Returns the cylinder previously set by `setCylinder()`. Since the IORB cylinder is set in `do_enqueueIORB()`, `getCylinder()` can be used only in `do_dequeueIORB()`.

In addition, the following methods, implemented in class `Disk`, are available. These methods can be useful in order to implement certain I/O scheduling

strategies. Note that `Disk` is a subclass of `Device`. Since the devices we are dealing with in this project are disks, all these methods are applicable to the `Device` objects that occur in this project.

◇ `final public int getHeadPosition()`
Returns the head position (the cylinder number where the read/write head is parked). Cylinders are counted from 0.

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

◇ `final public int getTracksPerPlatter()`
Tells how many tracks a platter has (or, equivalently, the number of cylinders on the disk).

◇ `final public int getSectorsPerTrack()`
Tells 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 of the disk per tick.

◇ `final public int getSeekTimePerCylinder()`
Tells how long it takes to seek to the next cylinder.

Summary of Class `Device`

The following API provided by class `Device` (implemented in its superclasses) can be used to obtain information about *OSP 2* devices. All the methods and variables listed apply to `Device` objects.

◇ `protected GenericQueueInterface iorbQueue`
This variable holds the device queue. It is manipulated by the methods `do_enqueueIORB()` and `do_dequeueIORB()`. The implementation of the device queue is *up to the student* module. The only requirement is that the class of the queue object must implement the interface `GenericQueue-Interface`. This interface mandates the methods `length()`, `isEmpty()`, and `contains()`, as described at the end of Section 1.5. Note that the interface defines only the methods *OSP 2* itself uses internally. For your purposes, your queue class would need additional methods, such as insertion and deletion of members of the queue. Note that since these methods are not defined in `GenericQueueInterface` you would need to use the *cast* operator to invoke them on `iorbQueue`.

◇ `final public boolean isBusy()`
Tests if the device is busy.

◇ `final public void setBusy(boolean flag)`
Sets the device busy or idle depending on the value of `flag`.

◇ `final static public Device get(int deviceID)`
Returns the device object with the given device Id.

◇ `final public int getID()`
Returns the Id of the device.

◇ `final public void startIO(IORB iorb)`
Starts the device and instructs it to perform the I/O operation specified in `iorb`. As part of this operation the device becomes busy, so you do not need to set it to busy explicitly.

◇ `final public String ospDeviceQueue()`
This method returns a string that contains the OSP version of the waiting queue to the device. You can print it out and use it for debugging.

◇ `final public int getTableSize()`
Returns the size of the device table.

6.5 Class `DiskInterruptHandler`

This class is declared as follows:

```
public class DiskInterruptHandler extends IflDiskInterruptHandler
```

It has only one method, `do_handleInterrupt()`, which implements the device interrupt handler. The method has the following signature:

```
public void do_handleInterrupt()
```

The following actions need to be performed as part of the handler:

1. Obtain information about the interrupt from the interrupt vector, class `InterruptVector`, described in Section 1.4. The main piece of information is the IORB that caused the interrupt. It is obtained using the method `getEvent()` of class `InterruptVector` (since the IORB is the event that "caused" the interrupt). The other necessary pieces of information, the thread, page, open-file handle, etc., are obtained using the API described in Section 6.3.

2. The IORB count of the open-file handle associated with the IORB must be decremented using `decrementIORBCount()` as described earlier.

3. If the open file has the `closePending` flag set and the IORB count is 0, the file might need to be closed. The IORB count of a file handle can be obtained via the method `getIORBCount()`. See the relevant part of the description of the method `do_cancelPendingIO()`.

4. The page associated with the IORB must be unlocked, because the I/O operation (due to which the page was locked) is over.

5. If the I/O operation is *not* a page swap-in or swap-out, then, unless the thread that created the IORB is dead, you need to set the frame associated with the IORB's page as referenced using the method `setReferenced()` of `FrameTableEntry`. In addition if it was a read operation (I/O type `FileRead`) then the frame must be set dirty (using the method `setDirty()` of `FrameTableEntry`). Of course, this can only be done if the task associated with the thread is still alive, as otherwise the memory of the task will be deallocated anyway. The thread's task is obtained using the method `getTask()` and its status is checked using the method `getStatus()`. A live task has status `TaskLive`; otherwise, the status is `TaskTerm`.

   To find out whether an I/O is a swap-in or swap-out from/to the swap device, one should compare the device Id of the IORB (`getDeviceID()`) with `SwapDeviceID`, a constant defined in *OSP 2*.

6. If the I/O was directed to the swap device and the task that owns the thread and the IORB is alive, you should mark the frame as clean (`setDirty(false)`).

7. If the task that owns the IORB is dead (status `TaskTerm`) and the frame associated with the IORB was reserved by that task (verified using `getReserved()`), you must unreserve the frame using `setUnreserved()`.

8. The threads that are waiting on the IORB must be woken up by a call to `notifyThreads()`.

9. The device must be set to idle using the method `setBusy()` with the appropriate flag.

10. The device must be told to service a new I/O request. This IORB is picked up using the method `dequeueIORB()`. If it returns a non-null object, the device should be restarted with that IORB using the method `startIO()`.

11. Finally, a new thread must be dispatched using method `dispatch()` of `ThreadCB`.

Relevant methods defined in other classes. The following methods defined in other modules can be used to implement the disk interrupt handler.

◇ `final static public Event getEvent()` `InterruptVector`
Extracts the event that caused the interrupt (e.g., a page, an IORB).

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

◇ `final public void decrementIORBCount()` `OpenFile`
Decrements the count of active IORBs associated with the open-file handle.

◇ `final public int getIORBCount()` `OpenFile`
Returns the current IORB count for the open-file handle.

◇ `public final void setReferenced(boolean flag)` `FrameTableEntry`
Marks frame as referenced.

◇ `public final void setDirty(boolean flag)` `FrameTableEntry`
Marks frame as dirty.

◇ `public final TaskCB getReserved()` `FrameTableEntry`
Marks frame as reserved.

◇ `public final void setUnreserved(TaskCB t)` `FrameTableEntry`
Unreserves frame.

◇ `final public int getDeviceID()` `IORB`
Returns the device associated with the IORB.

◇ `final public ThreadCB getThread()` `IORB`
Returns the thread that issued the I/O request.

◇ `final public PageTableEntry getPage()` `IORB`
Returns the buffer page in main memory that is the source or the target of the I/O.

◇ `public void notifyThreads()` `Event`
Wakes up threads that are waiting on the event.

◇ `final public void setBusy(boolean flag)` `Device`
If `flag` is *true*, marks the device as busy. Otherwise, marks it as idle.

◇ `final public IORB dequeueIORB()` `Device`
Takes an IORB off the device queue and Returns that IORB object.

◇ `final static public void startIO(IORB iorb)` `Device`
Tells the device to start working on `iorb`.

◇ `final static public void dispatch()` `ThreadCB`
Dispatches a thread to run.

◇ `final public TaskCB getTask()` `ThreadCB`
Returns the task that owns the thread.

◇ `final public int getStatus()` `ThreadCB`
Tells the status of the thread. See `GlobalVariables`, Section 1.5, for the list of legal status codes for a thread.

◇ `final public int getStatus()` `TaskCB`
Tells the status of the task. See `GlobalVariables`, Section 1.5, for the list of legal status codes for a task.

Summary of Class `DiskInterruptHandler`

This class typically does not maintain data structures of its own. However, since it is intended to process device interrupts, it indirectly manipulates other data structures, such as `IORB`'s, threads, and page tables, through the methods provided by these classes.

6.6 Methods Exported by Package DEVICES

The package DEVICES exports the following methods that are used by other classes in *OSP 2*. To the right of each method we indicate the class where the method is defined.

◇ `final public ThreadCB getThread()` `IORB`
Returns the thread that requested the I/O.

◇ `final public int getTableSize()` `IORB`
Returns the size of the device table.

◇ `final static public Device get(int deviceID)` `IORB`
Returns the device object with the given device Id.