Disk Queue

Introduction

One of the functions of an operating system is to implement device drivers. This project will have you write an operating system simulator which will simulate two versions of a disk device driver. The difference in the two versions will be the way scheduling is implemented. In the process of writing this simulation we will be working with various data structures including a normal first-come-first-serve (FCFS or FIFO) *queue* and a *priority queue*. We will be writing many different classes which will be used by the program. We will also be utilizing random number generators to assist us in implementing the simulation. Please read this whole document and make sure that you understand what is going on. Then you may start designing and implementing the program.

What is a simulation?

A simulation program tries to simulate the behavior of a real world system. They are used when it is impractical to build a prototype or economically unfeasible to do so. Since we don't have the hardware available nor the access into an operating system we will be simulating the operating system and the disk device driver. We won't really be reading or writing any data from or to the disk. Our concern will be the performance of the two different scheduling algorithms. To help us to do this simulation we will simulate requests coming into the operating system at random times for disk blocks to be read/written from/to the disk. The blocks will be randomly distributed across the disk. We will be using an *event queue* to keep track of the events that the operating system will be simulating.

Disk drive organization/configuration

A modern disk drive consists of multiple magnetic platters (in our example we will consider nine (9) platters). Each platter has two surfaces (on top and on bottom) (the top and bottom platters only use the inside surfaces, thus the topmost and bottommost surfaces are not used to store information). The platters are stacked on a spindle in the center of the platters. There is a read/write head arm mechanism which contains all of the read/write heads for all of the accessible platter surfaces (these heads all move in and out of the disk together). With nine platters we would, therefore, have 16 accessible surfaces (remember that the top and bottom surfaces of the stack are not used) and thus 16 read/write heads. The head arm can be moved such that it can access data anywhere from the outer rim of the platter to the inner hub of the platter. Each platter is divided into concentric (one inside the other) circles, called **tracks**, which contain the information stored on the disk. We generally consider all of the tracks at a given distance from the center of the stack (on all of the platters) a **cylinder**. Finally, each track is subdivided into independent **sectors** of 512 bytes each. Therefore, we can describe the disk configuration as containing a specified number of *cylinders* (the concentric rings on **all** surfaces), *tracks* (the ring on a particular surface), and *sectors* (the 512 byte blocks of data that together make up a single *track*).

In our example we will simulate a Western Digital Caviar 30 GB (WD300BB) disk drive. The configuration of the drive is:

Number of cylinders	58,168
Number of tracks/cylinder (heads)	16

Number of sectors/track	63
Number of bytes/sector	512

If we multiply these numbers together we come out with approximately 30.020 GB (here GB = $1{,}000^3$ and not $1{,}024^3$) (before formatting).

Drives are designed with different performance characteristics. One of these characteristics is how long it takes for the head arm to move between cylinders. To describe this manufacturers refer to the average track to track time (how much time it takes to move from one cylinder to the next cylinder) and the end to end time (how much time it takes to move from the innermost cylinder to the outermost cylinder (or the other way)). Taking these two pieces of information into account and realizing that to move the head arm we need to ramp up from a velocity of 0 to some traveling velocity, move the head arm and then ramp back down to 0 velocity. We come up with the following performance numbers:

Ramp time (up/down)	1,000 µsec
Track to track time (once moving)	0.326651 µsec

Therefore, to move from one cylinder to the next takes 1,000 + 0.326651 + 1,000 µsec which equals approximately 2 msec. To move from the first to last cylinder takes 1,000 + 0.326651 × 58,168 + 1,000 µsec which equals approximately 21 msec. These two statistics are the quoted statistics for the Western Digital drive in question.

Other configuration parameters deal with the rotational speed of the drive. This particular drives rotates at 7,200 revolutions per minute (RPM).

Note that we need to consider the time it takes to move the head arm between cylinders and the time it takes for the platters to rotate so that the specified sector is under the read/write head (we will discuss this later). We do not need to worry about the time it takes to change heads (i.e., to select which head (or platter surface) is to be read from/written to. This can be done very quickly (at electronic speeds rather than mechanical speeds (as is the case with cylinder movement and drive rotation)).

We will need to use a class which will maintain this information. I have included the class `DiskConfig` for that purpose. It obtains the necessary symbolic constants from an interface which the class *implements* called `WD300BB` (also included) and includes the accessor methods needed to retrieve them. Also note that some data is derived from the fundamental constants. The accessor methods will perform the calculations (see the code).

Random number generation

The simulation requires requests to come in at random times and the request to be for data at a random block location. To generate these random numbers we need to consider what the probability statistics are for these values. Let's consider the arrival time of requests. It makes sense to consider that these times follow a normal distribution around some mean value. Let us say that requests come in approximately every 10 msec with a standard deviation of 2 msec (i.e., 67% of all requests arrive between 8 and 12 msec apart) (10 ± 2 msec). The standard Java class, `java.util.Random`, contains a method (`nextGaussian()`) which returns the next *normally* distributed (according to the *Gaussian* distribution) value with a mean of 0.0 and a standard deviation of 1.0. To convert this value to 10 ± 2 we need to multiply the return value from the method by 2.0, to double the standard deviation, and then add 10.0 to offset the mean from 0.0 to 10.0. (Our clock uses µsec timing so we really multiply by 2,000 and add 10,000.)

The disk request will need to specify which disk block (512 byte sector) we want to access. To do this we can call the method (`nextInt(maxSect)`) from class `java.util.Random`. This method will produce a value between 0 and `maxSect` (the maximum block number on the disk (see the `DiskConfig` class code)). These values should be *uniformly* distributed across all of the block numbers and the above method will do just that.

The use of the random numbers will only appear in the main simulator code. This will be provided for you in the class definition for `DiskSimulator`.

Note: to properly test our assumptions regarding scheduling policy we really should make sure that the two simulation runs are based on the same data (i.e., the requests arrive at the same time for each simulation run and the disk blocks requested each time should also be the same). To guarantee this we set the random number generator's *seed* value to the same constant before each of the two runs. This will then produce the same random sequences and therefore will simulate the same environment but just use different scheduling policies, thus a fair test of the policies and not the random number generators is obtained.

DiskDriver class

The actual work of the simulation will be done in the `DiskDriver` class. We will discuss this class here so that you understand what it is doing, but I will provide the code for this class.

The class is an *abstract* class. It defines many methods but does not define the request scheduling algorithms. There are two derived classes which handle the two different algorithms (described later).

The purpose of the disk driver is to accept requests to access specified disk blocks, schedule these requests, issue commands to move the read/write head arm to the proper cylinder on the disk (and select the proper read/write head on the head arm), when the cylinder has been accessed the driver must wait until the disk revolves so that the required sector is under the read/write head. The drive would normally then read or write the sector under the read/write head. When the read/write has completed, the driver posts a notice that the request has been satisfied and upon acknowledgement from the operating system (simulator) selects the next scheduled request to process.

Our code needs to link the driver to the simulator so that the constructor for this class has a reference to the simulator class object as its parameter. The simulator contains a reference to the configuration class object and an access method to return the reference. The driver obtains the reference from the simulator and uses the numbers stored in the configuration object to define the behavior of the disk drive being simulated. The driver also keeps track of the current disk request object (containing the information regarding the request and performance data) (see below).

The simulator's time is needed to keep track of disk rotation. When the simulation is begun we assume that the disk head is over sector 0. As the disk rotates (and it does so completely independently of any other activity) the sector under the head keeps changing. Basically, we can use the simulated time to tell us which sector is currently under the read/write head. The private method, `setCurrSector()`, determines the current sector based on the simulated time and stores it in the driver's data member which keeps track of the `sector` (there are also data members which keep track of the `cylinder` and `head` that is currently active).

There are two more private methods, `calcSeekTime()` and `calcRotTime()`, which are used to determine (based on configuration data) the time it will take to satisfy a seek (move to the appropriate `cylinder`) and to satisfy a rotation wait (wait for the appropriate sector to be under the read/write `head`).

The `procRequest()` method will:

1. Determine the current simulator time (method `getTimer()` from the simulator object).
2. Store the current simulator time in the request object instance to mark the start of processing for this request.
3. Calculate the time it will take to seek to the proper cylinder based on the current location of the read/write head.
4. Add a simulator event which represents the satisfaction of the seek activity. This will, in effect, invoke a seek interrupt from the operating system into the disk driver code.

The `seekInterrupt()` method will:

1. Determine the current simulator time (method `getTimer()` from the simulator object).
2. Set the disk driver's `cylinder` and `head` data members to record the fact the these have changed because of the interrupt.
3. Since the simulator time has changed, it follows that the `sector` under the read/write had has also changed, so we call `setCurrSector()` to adjust the driver's knowledge of which sector is under its head.
4. Calculate the time it will take to wait for the proper sector to rotate under the read/write head.
5. Add a simulator event which represents the satisfaction of the rotation wait activity. This will, in effect, invoke a rotation wait interrupt from the operating system into the disk driver code.

The `rotInterrupt()` method will:

1. Determine the current simulator time (method `getTimer()` from the simulator object).
2. Since the simulator time has changed, it follows that the `sector` under the read/write had has also changed, so we call `setCurrSector()` to adjust the driver's knowledge of which sector is under its head.
3. Verify that the sector we are looking for is, in fact, the sector under the read/write head. In the case of the simulation this is done by checking simulator time. In reality, when the disk is originally formatted each sector is labeled with a small ID block. This ID block is read and verified before the read/write action is started. If the verification fails, the disk waits for one more rotation and tries again. We will not implement this retry effort, but we will report a validation failure.
4. The disk request object is modified to include the request satisfaction time. This will be used by the simulator to determine timing statistics.
5. Add a simulator event which represents the satisfaction of the complete disk request. The time of this event is the same time as the rotation interrupt event. The event will, therefore, be the next event processed. The simulator will report the performance of this request to the user.

Event model

In any simulation we need to define the event model. There are a few ways that event modeling can be done. In one model, we keep track of time by incrementing a timer by a unit value (in our simulation the unit value is the μsec (microsecond, millionth of a second, 10^{-6} second)). We set up a loop and increment the timer by one (1), check an event queue to see if an event is scheduled to happen at this time (if so, process it), and then continue the loop.

A second approach is to realize that there will be many cycles of this loop for which no events are scheduled. Instead of looping and performing no activity for each loop cycle, why not just request the next event and set the timer to that value. This is what our simulator will do.

What are the events that our simulated operating system will need to deal with? There are four (4) that are relevant to the disk driver simulation.

1. A `NEW_REQUEST` for a disk read/write is entered from a user process.
2. The disk driver has requested the read/write head arm to move to another cylinder (seek) and the cylinder has been reached (`SEEK_SATISFIED`).
3. The disk driver is waiting for the disk to rotate such that the proper sector is under the read/write head and the rotation has reached its destination (`ROT_SATISFIED`).
4. The complete user request (started off by the first event) has been satisfied (`REQUEST_SATISFIED`).

You will be writing the class, `Event`, which will represent event objects that are added to the event queue. The class contains public constants representing the four event types described above. The class contains private data members representing the `type` of the event (one of the four constants, above) (`int`) and the `time` (in simulated time (µsec)) (`long`) in the future that the event should take place. The complete constructor takes a type and time and sets the data members. There are appropriate accessor methods for the type and time. Because the event object will be added to a priority queue (since we want to select the next event to happen from the front of the queue) we will need to be able to compare event objects. Therefore, the `Event` class will implement the `Comparable<Event>` interface and implement the `compareTo()` method. It should also implement the `equals()` method, for completeness. Finally, the class will implement the `toString()` method. This method will display the `type` of the request (as a reasonable string) followed by the "`@`" symbol (surrounded by spaces) and then the `time` in fractional milliseconds. To assist in formatting times I have provided the `Fmt` class which contains the `static` method, `time(long timeValue)`. Be aware that the `type` value may not be valid and so a check should be made that if the `type` is not one of the known constants, the `toString()` method will display "`undefined:` *type* `@` *time*".

You will also need to write the class, `EventQueue`. This class will depend on the priority queue model and will be discussed below. The purpose of the event queue will be to store the `Event` objects in time priority. As events are added to the queue they are placed appropriately according to their time priority (highest priority given to smallest time value). Events are removed from the front of the queue and thus the next event will always be the one to be removed. Note: when the event queue is empty the simulation run has completed.

Disk requests

The scheduling of disk requests requires that we encapsulate the concept of a disk request. You will write the class, `DiskRequest`, which will serve to encapsulate not only the information regarding the requested disk block but also information regarding the times of various events associated with the request. These times will be used by the simulator to generate performance statistics. The statistics will be reported by the simulator and will represent the quality of the two scheduling algorithms.

Each disk request will need (as private member data):

1. The request number (`reqNum`) (`int`). The request number is a sequential number used to keep track of requests (for reporting purposes). It is implemented by using a class (`static`) variable for this class called `seq`. Each time a new request is constructed the value of this class variable is incremented by the constructor and its value stored in the instance variable `reqNum`.
2. The `cylinder`, `head`, and `sector` of the data block to be read or written (all `int`s).
3. The `requestTime`, `startTime`, and `satisfyTime` for this request (all `long`s in units of milliseconds). The `requestTime` is when the request is first entered in to the device driver's queue of waiting requests. The `startTime` is when the service of the request is actually started. The `satisfyTime` is when the request has been completed. We can compute the waiting time for this request by subtracting the `requestTime` from the `startTime`. We can compute the processing time for this request by subtracting the `startTime` from the `satisfyTime`.

The default constructor will increment the class variable, `seq`, and store its value in the `reqNum` instance data member. All other data members are set to zero.

The second constructor will increment the class variable, `seq`, and store its value in the `reqNum` instance data member. This constructor will be passed the requested `cylinder`, `head`, and `sector`, as well as the request time for this request. The parameters will be stored in the appropriate instance variables. Since we have not yet started processing of the request the start and satisfy times are set to zero (0).

The third constructor will behave like the other two with respect to the `seq` and `reqNum` variables. It will be passed a `block` number and a reference to the `DiskConfig` object from the simulator. The block number will be converted into a `cylinder`, `head`, and `sector` using the information available from the `DiskConfig` object.

Accessor methods should be written for all instance data members.

Mutator methods should be written to set the start and satisfy times. Also, you should write a method, `resetSeq`, that restarts the sequence counter (so that we can run another simulation using the same sequence numbers).

Write the method `toString()`. This method will use the instance variables to call the pre-written class method `Fmt.chs(int cylinder, int head, int sector)`. This method will format a string to look like: "`[`*cyl*`/`*head*`/`*sect*`]`".

Write the methods `equals()` and `compareTo()` (make sure the class implements `Comparable<DiskRequest>`). Both methods compare the cylinder/head/sector address of the requested disk block but then go on to compare the request time. Thus, if two requests ask for the same block the one entered first will be processed first.

Queuing methods (scheduling algorithms)

We will need to deal with two scheduling algorithms and thus two queuing methods in this simulation. The first is the standard first-come-first-served (FCFS) queuing method. The second is the priority queuing method. To implement these two queuing methods you will write two generic classes: `Queue<T>` and `PriorityQueue<T extends Comparable<T>>`.

The `Queue<T>` class will be implemented using the standard Java library `List<T>` interface (and the `LinkedList<T>`) implementation of the interface. It will contain the following public methods:

- `int getSize()`
- `boolean isEmpty()`
- `void clear()`
- `void add(T data)`
- `T get()`
- `T remove()`

The `PriorityQueue<T extends Comparable<T>>` class will be an `abstract` class. There will be two descendent classes from it: `PriorityQueueAsc<T extends Comparable<T>>` (ascending) and `PriorityQueueDesc<T extends Comparable<T>>` (descending). We need two classes because they implement one of two different priority calculation schemes (ascending or descending) and the result of the `compareTo()` method must compare to 0 in the opposite sense. In the declaration of the derived classes the `extends` clauses should read "`PriorityQueue<T>`" since the type parameter `T` was declared just before to be "`<T extends Comparable<T>>`" and should not be redeclared but just referenced. We will need to implement this class completely (not taking advantage of any predefined class as with `Queue<T>`). The class will need a `Node` inner class (`protected` so that the derived classes can access it). It will also need two `Node` references, one for the `head` or front of the queue and one for the `tail` or back of the queue. It will also need a `count` of the number of elements in the queue. The default (and only) constructor will `clear()` the queue (thus initializing the data member references). It will contain the following public methods:

- `int getSize()`
- `boolean isEmpty()`
- `void clear()`
- `abstract void add(T data)`
- `T get()`
- `T remove()`

The two derived classes will each implement a constructor that just invokes the parent's constructor (`super()`) and will implement the abstract method `add`. Note that you must pay careful attention to the relational operator when evaluating the `compareTo()` result to make sure that the queue is actually going in the proper direction.

Disk driver concrete classes

These classes implement the scheduling algorithms of the `DiskDriver` class (that were abstract and not defined, only declared). They will implement their constructors which invoke their parent's constructor (`super()`) and instantiate the `requests` queue(s). They will implement (actually, override) the methods: `schedRequest()`, `nextRequest()`, and `removeRequest()`.

DiskDriver_fcfs

This class implements scheduling via a first-come-first-served queuing algorithm. It uses the `Queue<T>` class (see above) to store requests. The simulator will create requests and set the requesting time in the request. It will pass the request to the disk driver for scheduling (`schedRequest()`). This driver's method will add the request to the back of the queue. If the driver is not processing any requests at present (signified by the instance variable `request` having a value `null`) the method will get the next request (by calling `nextRequest()`, see below) and start processing it (by calling `procRequest()`). The `nextRequest()` method will get the next request form the queue (without removing the request from the queue, *yet*) and set the disk driver's `request` instance variable. Note that the driver's `procRequest()` method will process whatever request is being referenced by the `request` instance variable. After the request has be processed, this driver's `removeRequest()` method will remove the request at the front of the queue.

DiskDriver_elevator

This class implements scheduling via the elevator queuing algorithm. Consider an elevator. There are buttons inside the elevator for each floor. Also, on each floor there are buttons to call the elevator. At any point in time the elevator may need to stop at many floors (either because users inside the elevator have requested floors or user on the floors have requested access to the elevator ("called" the elevator). The elevator does not respond to the floor requests in a first-come-first-served manner. Instead it optimizes its performance by moving up until it has reached the highest requested/requesting floor. It then goes down until it has reached the lowest requested/requesting floor. It continues to make loops, first going up, then going down, stopping at floors, as needed, along the way. The order of the floors at which the elevator stops is *not* the same as the order at which the requests came in. That would not be optimal. The elevator reorders the requests to improve performance.

Let us think about how this is done. If we consider two queues of floor requests (either requested by users in the elevator or users calling the elevator from outside) one for going up and one for going down. If the elevator is at floor *n* and is going *up*, and a request comes in for floor *m* which is above floor *n*, then the request gets added to the *up* queue. If, however, floor *m* is below floor *n*, then the request gets added to the *down* queue. When the elevator reaches its highest floor (and the *up* queue is empty) it changes direction to *down* and starts looking at the *down* queue. At this point, any new requests, *k*, are evaluated. If floor *k* is below floor *n* (still the current floor of the elevator) then we add *k* to the *down* queue, otherwise we add *k* to the *up* queue. When we reach the lowest floor (and the *down* queue is empty) the elevator changes direction to *up* and starts looking at the *up* queue again.

The same logic holds for accessing disk blocks (seeking cylinders, actually). Rather than access the request in the order in which they come, we reorder the requests so that they are processed in cylinder number order. We will need two *priority queues* to store the requests (unlike above which only needed a single FCFS queue). One queue to handle upward (outward on the platter) going requests and one queue to handle downward (inward on the platter) going requests. In the first case the priority of disk access is going to be ascending with cylinder number. In the second case the priority of disk access is going to be descending with cylinder number. This is most easily implemented by deriving two types of priority queues, one ascending and one descending. The methods are mostly the same except that the relational operator that compares two values will use the opposite relation (≤ versus ≥). Within our disk driver we will have an array of two priority queues, one constructed from an ascending priority queue and the other constructed from a descending priority queue.

```
private PriorityQueue<DiskRequest>[]requests;
```
defined as a data member in the class

```
requests = new PriorityQueue[2]
requests[UP] = new PriorityQueueAsc<DiskRequest> ();
requests[DN] = new PriorityQueueDesc<DiskRequest> ();
dir = UP;
```
defined in the instance constructor

Note: the compiler will produce a warning for the array construction. This warning can be removed by adding the *annotation*: "`@SuppressWarnings("unchecked")`" on the line preceding the constructor declaration (note that this annotation is like a modifier (e.g., `public` or `static`) and does not terminate with a semicolon (`;`).

`UP` and `DN` are defined as:

```
private static final int UP = 0;
private static final int DN = 1;
```

Note that we can easily swap directions by:

```
dir = 1 - dir;
```

If `dir` has the value 0 (`UP`), then `1 - dir` is 1 – 0 →1 (`DN`).
If `dir` has the value 1 (`DN`), then `1 - dir` is 1 – 1 → 0 (`UP`).

You can also use:

```
dir ^= 1;
```

The caret (`^`) operator is the exclusive OR (XOR) operator. A single bit XOR 1 will toggle the state of the bit. Thus, 0 → 1 and 1 → 0.

Review of class provided and those needed to be written

The following classes will be provided to you to use:

- `DiskSimulator`, includes `main()`.
- `WD300BB`, drive configuration interface.
- `DiskConfig`, implements above interface.
- `DiskDriver`, abstract.
- `Fmt`, formatting utilities.

The following classes will need to be written by you and must fit together with the supplied classes:

- `Event`, object type to be inserted into event queue.
- `DiskRequest`, object type to be inserted into disk driver queues.
- `DiskDriver_fcfs`, first-come-first-served driver derived class.

- `DiskDriver_elevator`, elevator algorithm driver derived class.
- `Queue`, generic fcfs queue.
- `PriorityQueue`, abstract priority queue whose `add()` method is also abstract and will be implemented by the derived classes.
- `PriorityQueueAsc`, derived class for ascending priority queue.
- `PriorityQueueDesc`, derived class for descending priority queue.
- `EventQueue`, an ascending priority queue for holding events.

Also provided is a sample run of the program. Your output should be the same if your part of the project is implemented properly. You will notice the following statistical results for processing 50 requests:

Model	Total Time	Avg. Wait Time	Avg. Run Time
FCFS	633.891	83.102	12.677
Elevator	517.227	21.265	10.344
Perf. Improvement	18%	74%	18%

Total time and average run time improved by 18%. Average wait time improved by a whopping 74%. You should observe that in the fcfs model the wait time per request kept increasing as the run continued while in the elevator model the wait time remained fairly stable throughout the run.