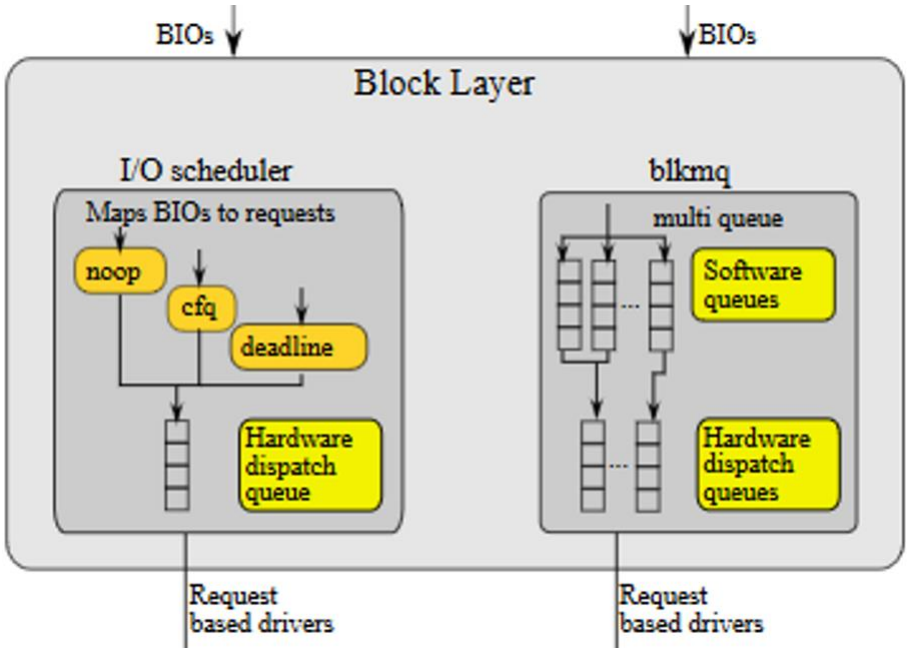
Programming Assignment #4 (Lab 4): IO Scheduling **Professor Hubertus Franke**
Class CSCI-GA.2250-001/003 Fall2021

In this lab you shall implement and simulate the scheduling and optimization of I/O operations. Applications submit their IO requests to the IO subsystem (potentially via the filesystem), where they are maintained in an IO-queue until the disk device is ready for servicing another request. The IO-scheduler then selects a request from the IO-queue and submits it to the disk device. This selection is commonly known as the `strategy()` routine in operating systems and shown in below figure. On completion, another request can be taken from the IO-queue and submitted to the disk. The scheduling policies will allow for some optimization as to reduce disk head movement or overall wait time in the system. For this lab we only provide the left hand side of the figure (I/O Scheduler)



The schedulers that need to be implemented are FIFO (i), SSTF (j), LOOK (s), CLOOK (c), and FLOOK (f) (the letters in bracket define which parameter must be given in the –s program flag shown below).

You are to implement these different IO-schedulers in C or C++ and submit the **source** code, which we will compile and run. Your submission must contain a Makefile so we can run on **linserv*.cims.nyu.edu** (and please at least test there as well).

Invocation is as follows:

```
./iosched [ -s<schedalgo> | -v | -q | -f ] <inputfile>
```

The input file is structured as follows: Lines starting with '#' are comment lines and should be ignored.
Any other line describes an IO operation where the 1st integer is the time step at which the IO operation is issued and the 2nd integer is the track that is accesses. Since IO operation latencies are largely dictated by seek delay (i.e. moving the head to the correct track), we ignore rotational and transfer delays for simplicity. The inputs are well formed.

```
#io generator
#numio=32 maxtracks=512 lambda=10.000000
1 339
131 401
:
```

We assume that moving the head by one track will cost one time unit. As a result, your simulation can/should be done using integers. The disk can only consume/process one IO request at a time. Everything else must be maintained in an IO queue and managed according to the scheduling policy. The initial direction of the LOOK algorithms is from 0-tracks to higher tracks. The head is initially positioned at track=0 at time=0. Note that you do not have to know the maxtrack (think SCAN vs. LOOK).

Each simulation should print information on individual IO requests followed by a SUM line that has computed some statistics of the overall run. (see reference outputs).

For each IO request create an info line (5 requests shown) in the order of appearance in the input file.

```
    0:     1     1   431
    1:    87   467   533
    2:   280   431   467
    3:   321   533   762
    4:   505   762   791
```

Created by

```
     printf("%5d: %5d %5d %5d\n",i, req->arrival_time, r->start_time, r->end_time);
```

- IO-op#,
- its arrival to the system (same as inputfile)
- its disk service start time
- its disk service end time

Please remember " %5d" is not "%6d" !!!

and for the complete run a SUM line:

total_time:	total simulated time, i.e. until the last I/O request has completed.
tot_movement:	total number of tracks the head had to be moved
avg_turnaround:	average turnaround time per operation from time of submission to time of completion
avg_waittime:	average wait time per operation (time from submission to issue of IO request to start disk operation)
max_waittime:	maximum wait time for any IO operation.

Created by:

```
printf("SUM: %d %d %.2lf %.2lf %d\n",
        total_time, tot_movement, avg_turnaround, avg_waittime, max_waittime);
```

Various sample inputs and outputs are provided with the assignments on NYU brightspace.
Please look at the sum results and identify what different characteristics the schedulers exhibit.

You can make the following assumptions:
- at most 10000 IO operations will be tested, so its OK (recommended) to first read all requests from file before processing.
- all io-requests are provided in increasing time order (no sort needed)
- you never have two IO requests arrive at the same time (so input is monotonically increasing)

I strongly suggest, you do not use discrete event simulation this time. You can write a loop that increments simulation time by one and checks whether any action is to be taken. In that case you have to check in the following order.
The code structure should look *something* like this.

```
while (true)
   if a new I/O arrived to the system at this current time
      → add request to IO-queue
   if an IO is active and completed at this time
      → Compute relevant info and store in IO request for final summary
   if no IO request active now
      if requests are pending
         → Fetch the next request from IO-queue and start the new IO.
      else if all IO from input file processed
         → exit simulation
   if an IO is active
      → Move the head by one unit in the direction its going (to simulate seek)
   Increment time by 1
```

When switching queues in FLOOK you always continue in the direction you were going from the current position, until the queue is empty. Then you switch direction until empty and then switch the queues continuing into that direction and so forth. While other variants are possible, I simply chose this one this time though other variants make also perfect sense.

Additional Information:

As usual, I provide some more detailed tracing information to help you overcome problems. Note your code only needs to provide the result line per IO request and the 'SUM line'.

The reference program under ~frankeh/Public/iosched on the cims machine implements three options –v, -q, -f to debug deeper into IO tracing and IO queues.

The **–v** execution trace contains 3 different operations (***add*** a request to the IO-queue, ***issue*** an operation to the disk and ***finish*** a disk operation). Following is an example of tracking IO-op 18 through the times 1139..1295 from submission to completion.

```
1139:  18 add 211          //  18 is the IO-op # (starting with 0) and 211 is the track# requested
1127:  18 issue 211 279    //  18 is the IO-op #, 211 is the track# requested, 279 is the current track#
1195:  18 finish 68        //  18 is the IO-op #, 68 is total length/time of the io from request to completion
```

-q shows the details of the IO queue and direction of movement (1==up , -1==down) and
–f shows additional queue information during the FLOOK.

> Here Queue entries are tuples during add [ior# : #io-track] or triplets during get [ior# : io-track# : distance], where distance is negative if it goes into the opposite direction (where applicable).

Please use these debug flags and the reference program to get more insights on debugging the ins and outs (no punt intended) of this assignment and answering certain "why" questions.

Generating your own input for further testing:

A generator program is available under ~frankeh/Public/iomake and can be used to create additional inputs if you like to expand your testing. You will have to run this against the reference program ~frankeh/Public/iosched yourself.

Usage: iomake [-v] [-t maxtracks] [-i num_ios] [-L lambda] [-f interarrival_factor]

maxtracks is the tracks the disks will have, default is 512
num_ios is the number of ios to generate, default is 32
lambda is parameter to create poisson distribution, default is 1.0 (consider ranges from 0.1 .. 10.0)
interarrival_factor is time factor how rapidly io will arrive, default is 1.0 (consider values 0.5 .. 1.5), too small and the system will be overloaded and too large it will be underloaded and scheduling is mute as often only one i/o is outstanding.

Below are the parameters for the 15 inputs files provided in the assignment:

```
#numio=10   maxtracks=128  lambda=0.100000
#numio=20   maxtracks=512  lambda=0.500000
#numio=50   maxtracks=128  lambda=0.500000
#numio=100  maxtracks=512  lambda=0.500000
#numio=50   maxtracks=256  lambda=5.000000
#numio=20   maxtracks=256  lambda=2.000000
#numio=100  maxtracks=512  lambda=9.000000
#numio=80   maxtracks=300  lambda=3.500000
#numio=80   maxtracks=1000 lambda=3.500000
#numio=500  maxtracks=512  lambda=2.500000
#numio=51   maxtracks=300  lambda=4.000000
#numio=99   maxtracks=312  lambda=2.200000
#numio=30   maxtracks=10   lambda=0.300000
#numio=1000 maxtracks=648  lambda=2.500000
#numio=999  maxtracks=999  lambda=2.500000
#numio=200  maxtracks=512  lambda=2.500000
```