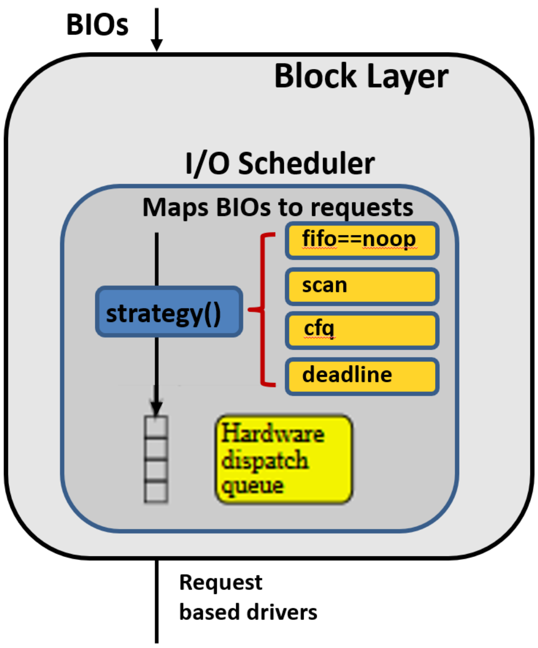
Programming Assignment #4 (Lab 4): IO Scheduling **Professor Hubertus Franke**
CSCI-GA.2250-001 Summer 2025

In this lab you will implement and simulate the scheduling and optimization of I/O operations for a hard disk. Applications submit their block IO requests (bio) to the IO subsystem [Block Layer] (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 the figure below. 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.



The schedulers that need to be implemented are FIFO (N), SSTF (S), LOOK (L), 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, **Makefile**, **make.log and grade.log** as a single *.zip, *.tar or *.tar.Z, which we will compile and run. The same deductions for missing files or non-requested files (input/outputs) are assessed as in previous labs.

Invocation is as follows:

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

Only the "-s" option is required. The default scheduler is fifo is "-s" is not supplied. Options as usual can be in any order.
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
1 328
129 401
:
```

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. Once a request is active on the disk it cannot be interrupted by any other incoming request. Hence these requests 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", iop, req->arr_time, r->start_time, r->end_time);
```

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

Please remember " %5d" is not "%6d" !!! For C++ formatting refer back to lab2 and lab3 where similar outputs were created.

For the statistics of the simulation provide a SUM line (note variables printed as "%lf" are double floats).

Created by:
```
printf("SUM: %d %d %.4lf %.2lf %.2lf %d\n",
        total_time, tot_movement, io_utilization,
        avg_turnaround, avg_waittime, max_waittime);
```

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
io_utilization:	ratio of time_io_was_busy / total_time
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.

10 sample inputs and outputs and runit/gradeit scripts are provided with the assignment on NYU brightspace.
Please look at the sum results and identify what different characteristics the schedulers exhibit.

You can make the following assumptions (enforced and caught by the reference program).
- 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 for this lab. You can write a simple 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 (there are some edge conditions you have to consider, such as the next or more than the next I/O is for the track the head currently is at, etc.):

```
while (true)
   if a new I/O arrived at 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 the 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 change the direction to up, until the queue is empty. Then you switch direction to down until empty and then switch the queues again going up. 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/lab4/iosched on the cims machine implements three additional 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 1151..1307 from submission to completion.

```
1151:   18 add 221          // 18 is the IO-op # (starting with 0) and 221 is the track# requested
1239:   18 issue 221 289    // 18 is the IO-op #, 221 is the track# requested, 289 is the current track#
1307:   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/lab4/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/lab4/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 a poisson distribution, default is 1.0 (consider ranges from 0.01 .. 10.0)
interarrival_factor is time factor how rapidly IOs 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 10 inputs files provided in the assignment so you don't pick the same.

```
 1.  iomake -v -t  128 -i  10   -L0.11 -f 0.4
 2.  iomake -v -t  512 -i  20   -L0.51
 3.  iomake -v -t  128 -i  50   -L0.51
 4.  iomake -v -t  512 -i 100   -L0.01
 5.  iomake -v -t  256 -i  50   -L1.1
 6.  iomake -v -t  256 -i  20   -L0.3
 7.  iomake -v -t  512 -i 100   -L0.9
 8.  iomake -v -t  300 -i  80   -L3.4  -f 0.6
 9.  iomake -v -t 1000 -i  80   -L3.4  -f 0.6
 10. iomake -v -t  512 -i 500   -L2.4  -f 0.6
```