Programming Assignment #4 (Lab 4): IO Scheduling **Professor Hubertus Franke**
Class CSCI-GA.2250-001/002 Spring2021

In this lab you implement/simulate the scheduling and optimization of I/O operations. Applications submit their IO requests to the IO subsystem (potentially via the filesystem and page-cache), 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. 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 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).

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 a follows:

```
./iosched –s<schedalgo>  [options] <inputfile>    // options are -v -q -f (see below)
```

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
1 430
129 400
:
```

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).

```
    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 classes.
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.

1) If a new I/O arrived to the system at this current time → add request to IO-queue ?
2) If an IO is active and completed at this time → Compute relevant info and store in IO request for final summary
3) If an IO is active but did not yet complete → Move the head by one sector/track/unit in the direction it is going (to simulate seek)
4) If no IO request active now (after (2)) but IO requests are pending → Fetch the next request and start the new IO.
5) If no IO request is active now and no IO requests pending → exit simulation
6) 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), to small and the system will be overloaded and to 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 provide 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
```

Spring 2021
Programming Languages
Homework 4

- This homework is a combination programming and "paper & pencil" assignment.
- Due via NYU Classes on Sunday, May 2, 2021 11:55 PM Eastern Time. Due to timing considerations relating to the end of the semester, late submissions will not be accepted. **No exceptions.**
- For the Prolog questions, you should use SWI Prolog. A link is available on the course page. For the Prolog Adventure question, please see the attachment `adv.pl`. You will start with the code in this file and modify it as you answer the questions.
- You may collaborate and use any reference materials necessary to the extent required to understand the material. All solutions must be your own, except that you may rely upon and use Prolog code from the lecture if you wish. Homework submissions containing any answers or code copied from any other source, in part or in whole, will receive a zero score.
- Please submit your homework as a zip file, `hw4-<netid>.zip`. The zip file should contain:
  - For Q1, `hw4-<netid>-Q1.pl` which should contain the reordered facts for questionable 1 and the 2 new rules for sub-question 4.
  - For the Prolog Rules question (Q2), `hw4-<netid>-prules.pl` which should contain your implementation of all of the rules. You do not need to submit queries, although we will run our queries on your solutions.
  - For Q3, and Q5, a PDF file `hw4-sols.pdf` containing the answers to all of the "paper & pencil" questions.
  - For the Twitter question (Q4), `hw4-<netid>-twitter.pl` which should contain all *rules* necessary to make the solution work. Use comments to mark your modifications.
  - For the Twitter question (Q4), `hw4-<netid>-twitter-queries.txt` containing the queries that the question asks for.
- Make sure your Prolog code compiles before submitting. *If your code does not compile for any reason, it may not be graded.*

1. [10 points] **Investigating Prolog**

   Consider the following:

```
male(brian).
male(kevin).
male(zhane).
male(fred).
male(jake).
male(bob).
male(stephen).
male(tom).
male(paul).

parent(melissa,brian).
parent(mary,sarah).
parent(stephen,jennifer).
parent(bob,jane).
parent(paul,kevin).
parent(tom,mary).
parent(jake,bob).
parent(zhane,melissa).
parent(tom,stephen).
parent(stephen,paul).
parent(emily,bob).
parent(zhane,mary).

grandfather(X,Y) :- male(X), parent(X,Z),parent(Z,Y).
```

   1. Reorder the facts above to provide faster execution time when querying `grandfather(tom,jennifer).` List the re-ordered facts and briefly explain what you changed.
   2. Explain in your own words why the change above affects total execution time. Show evidence of the faster execution time (provide a trace for each).
   3. Can we define a new rule `grandmother` that calls into the goals presented above to correctly arrive at an answer? Why or why not?
   4. Define new rules `aunt` and `uncle` that work with the rules above. If you need to define other rules (including facts) in order to correctly define these, go ahead. Assume that the universe of facts is **not** complete.

2. [20 points] **Prolog Rules**

Write the Prolog rules described below in a file `hw4-<netid>-prules.pl`. All rules must be written using the subset of the Prolog language discussed in class. You may not, for example, call built-in library rules unless specifically covered in the slides. You may call your own rules while formulating other rules. You do not need to turn in queries, although you should certainly test your rules using your own queries.

1. Write a rule `remove_item(I,L,O)` in which O is the output list obtained by removing every occurrence of an item I from list L. The items in O should appear in the same order as L, only without I. Optional hint: try first defining `remove_item/4` and then define `remove_item/3` in terms of that.
2. Write a rule `remove_items(I,L,O)` which is the same as `remove_item/3` above except that I is a *list* of items to be removed from L instead of just a single item.
3. Write a rule `intersection2(L1,L2,F)`[1] in which F is a set containing only items that appear in both L1 and L2. You should not assume that L1 or L2 are sets[2] and, therefore, may contain duplicate items. Optional hint: first defining `intersection/4` and then define `intersection/3` in terms of that.
4. Write a rule `is_set(L)` which is true if L is a set.
5. Write a rule `disjunct_union(L1,L2,U)` in which U is the disjunctive union of the items in L1 and L2. That is, items in L1 or L2 that are not in the intersection of L1 and L2. You should not assume that L1 or L2 are sets.
6. Write a rule `remove_dups(L1,L2)` in which L2 is L1 with duplicates removed. The order of the output list does not matter. This can also be viewed as a rule that creates a set from a list.
7. Write a rule `union(L1,L2,U)` in which U is the union (i.e., a set) of the items in L1 and L2. Do not assume that L1 or L2 are sets.

[1] The rule `intersection` is already defined by the Prolog standard library.
[2] A *set* is a collection of unique, unordered items.

3. [10 points] **Unification**

For each expression below that unifies, show the bindings. For any expression that doesn't unify, explain why it doesn't. Assume that the unification operation is left-associative.

1. d(15) & c(15)

2. 4 & X & 76

3. a(X, b(3, 1, Y)) & a(4, Y)

4. b(1,X) & b(X,Y) & b(Y,1)

5. a(1,X) & b(X,Y) & a(Y,3)

6. a(X, c(2, B, D)) & a(4, c(A, 7, C))

7. e(c(2, D)) & e(c(8, D))

8. X & e(f(6, 2), g(8, 1))

9. b(X, g(8, X)) & b(f(6, 2), g(8, f(6, 2)))

10. a(1, b(X, Y)) & a(Y, b(2, c(6, Z), 10))

11. d(c(1, 2, 1)) & d(c(X, Y, X))

4. [40 points] **Twitter on Prolog**

Consider the micro-blogging site, Twitter—first conceived at NYU by an undergraduate student and later launched in around 2006. It was originally written in Ruby on Rails, then later ported for reasons of scalability to the cross-paradigm language, Scala. Although Twitter was never implemented in Prolog, we shall see that implementing the majority of Twitter's decision engine in Prolog is surprisingly simple. Twitter has evolved over the years into a fairly complex system. However, we will consider Twitter as it existed in its earlier, simpler, days. First, some background.

Users may broadcast short messages called *tweets*. Although Twitter restricts the length of tweets, we shall ignore this restriction here and instead consider tweets of any arbitrary length. We shall assume two modes of tweet broadcast we shall consider:

1. Public, in which the tweet is visible to everybody.
2. Protected, in which the tweet is only visible to *followers* of the author, defined below.

If user x is generally interested in what user y has to say, user x may *follow* user y, causing all of y's tweets to appear in x's Twitter *feed*. The feed of user x is a running list of tweets generated by anybody that x follows.

If a user x finds a public tweet of another user y interesting, she may *retweet* it to her followers. This has the effect of the tweet being visible to the followers of x (in addition to y, as before). Note that a "follows" relationship is not a requirement for retweeting public tweets.

If user y is protected then y's tweet is visible only to direct followers of y. Therefore, if x retweets protected user y's tweet, then only those users who follow x *and* y will see the retweet.

Modern Twitter also allows a retweet to be optionally accompanied by text (in addition to the retweeted tweet), but we shall disregard the optional text feature here.

Another way tweets can gain visibility is through searching. Users can search the entire universe of tweets for certain keywords (including, but not limited to, hash tags). This is typically how users develop a following by other users.

In early versions of Twitter, a tweet which is directed to a particular user conventionally begins with the recipient's Twitter handle in the message itself, followed by the content[3]. Even if a tweet directed to a particular user, the tweet is still viewable by everybody[4].

Note that you may write more rules than defined below (e.g. "helper" rules), but you must write *at least* the rules defined below. Now let's get started:

1. Create a number of Twitter users by stating several `user(U,P)` facts, where U is the user's name and P represents that status of the user's account (i.e., public or protected). You may assume that both values are fixed to whatever is stated as a fact at the time you write your program. Note that all Twitter user names begin with the character @. For example, @tony.
2. The relation `follows(X,Y)` establishes that user "X *follows* user Y." Note that this relation should not be symmetric (i.e., user x following y does not imply that y follows x). Relate the above users by creating several `follows` facts.
3. The relation `tweet(U,I,M)` represents a tweet broadcast by user U with unique identifier I and message M. The identifier is needed because the same user could tweet the same message twice, but this is considered 2 distinct tweets. We can represent the message M as an array of atoms. We will not concern ourselves with the 140 character limit, nor the length of the array, for this assignment. Create several tweet facts.
4. Create a Prolog relation `retweet(U,I)` which represents user U retweeting the tweet identified by I.

[3]These days, the handle of the recipient is no longer considered part of the tweet.

[4]Twitter has a direct message feature that permits private messages (not the same as Tweets), but we shall disregard this here.

5. Create a Prolog relation `feedhelper(U,F,M,I)` which establishes user U's Twitter feed. F is any user who U follows, M is any message tweeted or retweeted by F and I is a unique tweet identifier. The intended use is to supply a binding for U in queries and F, M, I will serve as outputs. When doing so, note that Prolog may output identical instantiations (i.e., bind M, I to the same values more than once), effectively causing duplicate tweets to appear in the feed. This could happen for several reasons, such as if two users $f_1, f_2 \in F$ both retweet the same tweet. We can remedy this situation by defining the Prolog relation `feed(U,M)` as follows:

```
feed(U,M) :- uniquefeed(U,O),remove_ident(O,M).

uniquefeed(U,R) :- setof([I,F|M],feedhelper(U,F,M,I),R).

remove_ident([],[]).
remove_ident([[_|Y]|T1],[H2|T2]) :- Y=H2,remove_ident(T1,T2).
```

6. Create a Prolog relation `search(K,U,M)` which searches the universe of tweets for the keyword K. Variables U and M, when uninstantiated, will be bound to each tweet sent by user U whose message is M. For this homework, we will limit searches to single atoms so that finding a particular keyword (i.e., atom) within a tweet amounts to searching the tweet's array of atoms for a match.
7. A tweet identified by j is considered *viral* (as far as this homework is concerned) if there exists some $n \geq 2$ and there also exists a sequence of users $u_1, \ldots, u_n$ such that:
   - u_1 follows u_2.
   - for all $1 < i < n : u_i$ follows u_{i+1} *and* u_i retweets j.
   - u_n tweets j.

   The spirit of "virality" is that the tweet will have a high degree of visibility among the user base due to the combination of both following and retweeting.

   Create a Prolog relation `isviral(S,I,R)`, where I is the unique identifier of a tweet, $S = u_n$ and $R = u_1$. Note that if `isviral` holds for some n, then by definition it also holds for all i such that $1 < i < n$.
8. Create a Prolog relation `isviral(S,I,R,M)`, where I is viral and at least M transitive `follows` relations (beginning from R and terminating at S) exist. When the above relation holds, we say that a tweet is viral with at least M *levels of indirection*.

With these in place, please write the following queries:

1. Write a query that shows who is following a specified user. (For this question and all questions below where more than one correct answer to the query may exist, we assume that one would press semicolon after each variable binding to produce additional permissible answers until the list is exhausted).
2. Write a query that shows all tweets posted by a specified user.
3. Write a query that shows the users who retweeted a specified tweet.
4. Write queries that shows a particular user's feed. Ensure that tweets of any users that the user follows are visible in the feed.
   (a) Show at least one example where the user's feed contains at least one public tweet.
   (b) Show at least one example where the user's feed contains at least one protected tweet (that the user is authorized to see). For example, user x may follow y and y may retweet a tweet authored by protected user z where x follows z and therefore should see z's tweet.
   (c) Show at least one example where the user's feed **does not** contain at least one protected tweet. For example, user x may follow y and y may retweet a tweet authored by protected user z where x does not follow z and therefore should not see z's tweet.
5. Write a query that searches for a keyword in the universe of tweets.

6. Write a query that shows if a particular tweet is viral between the sender and a specified receiver.
7. Write a query that shows if a particular tweet is viral between the sender and a specified receiver in no less than 3 levels of indirection.

5. [10 points] **Virtual Functions**

I recommend waiting until the November 23 lecture before completing this problem. Consider the following C++ code:

```
#include<iostream>
using namespace std;

class Base
{
public:
    virtual void output ()
    { cout<< "output base class" <<endl; }

    void show ()
    { cout<< "show base class" <<endl; }
};

class Derived:public Base
{
public:
    void output ()
    { cout<< "output derived class" <<endl; }

    virtual void show ()
    { cout<< "show derived class" <<endl; }
};

int main()
{
  Base b;
  Derived d;
  Base* bp = new Derived();
  Base* bp2 = &b;

  b.output();
  b.show();
  d.output();
  d.show();
  bp->output();
  bp->show();
  bp2->output();
  bp2->show();

  return 0;
}
```

Complete the following:

1. Show the output of the main function above.
2. For which of the statements in `main` will the virtual function table be used to determine the specific method to call?

3. Draw the virtual function tables for `Base` and `Derived`. Show all of the entries in each vtable and the version of each method each entry will point to.

6. [10 points] **Prototype OOLs**

I recommend waiting until the November 23 lecture before completing this problem. Consider the following code below, written in a prototype OOL:

```
var obj1;
obj1.x = 20;
var obj2 = clone(obj1);
var obj3 = clone(obj2);
var obj4 = clone(obj1);
obj2.y = 5;
obj4.x = 10;
obj3.z = 30;
```

Assume that the program fragment above has executed. Answer the following:

1. What fields are contained locally to `obj1`?
2. What fields are contained locally to `obj2`?
3. What fields are contained locally to `obj3`?
4. What fields are contained locally to `obj4`?
5. To what value would `obj1.x` evaluate, if any?
6. To what value would `obj2.x` evaluate, if any?
7. To what value would `obj3.x` evaluate, if any?
8. To what value would `obj4.x` evaluate, if any?
9. To what value would `obj4.y` evaluate, if any?
10. To what value would `obj2.y` evaluate, if any?
11. To what value would `obj3.y` evaluate, if any?
12. To what value would `obj3.z` evaluate, if any?