This homework covers OSTEP chapters 32 and 37. It is due in class Monday, April 27.

While you may discuss ideas and strategies for problems with other students, you should always make the first attempt on a problem yourself and **you must write up your own solutions in your own words**. *You may not collaboratively write solutions or copy a solution that one person in the group writes up. You also may not look for, copy, or use solutions from other sources, including from generative AI like ChatGPT, even if you make changes. There's no such thing as using someone else's solution for a problem "as an example" for writing your own.*

Preliminaries

- Copy the directories `threads-bugs` and `file-disks` (and their contents) from `/classes/cs331` to your `~/cs331` directory. (You won't be writing any code so these don't need to go into your `workspace` directory.)

`thread-bugs` contains C programs along with a `Makefile`. `file-disks` contains programs `disk.py` and `disk-precise.py`. You will only be using `disk.py`. Both directories contain `README`s with instructions. You can also find the text of the `README`s in the appendix sections A and B at the end of this document.

Exercises

Chapter 32 homework problems (at the very end of chapter 32) —

Run `make` in the `threads-bugs` directory to compile all of the programs.

1. Do #1–3. First focus on understanding `vector-deadlock.c`, the various flags, and the output — use the `-v` flag and keep the number of threads and loops fairly small so the output is manageable. When you consider the "does it deadlock?" questions, run without `-v` — the program runs very quickly even up to 100,000 loops or more (depending on the number of threads) so if it doesn't finish almost immediately, it is probably deadlocked. (Use ctrl-C to kill it.)

   In both cases keep in mind that just because deadlock *can* occur doesn't mean it will — run the program many times and/or use large numbers of loops to provide opportunities for less-common behaviors to occur.

2. Do #4–10. For the "observe the performance" parts of the questions, you should gather data for the same combinations of flags for each of the eight variations (four programs, both with and without `-p`) so that you can more easily compare them. Include a table of the data gathered in your writeup.

3. Based on your observations: What strategies would you turn to first for deadlock prevention, and why? Does your answer depend on how frequent contention is expected to be? Is deadlock detection rather than prevention a viable alternative? Why or why not?

Chapter 37 homework problems (at the very end of chapter 37) —

The `README` does not cover all of the flags supported by `disk.py`. Most of the ones you need are mentioned in the exercises but you may find it useful to run `disk.py -h` to see what all is available.

4. Do #1–3 to gain an understanding of how seek, rotation, and transfer times contribute to the overall time to satisfy a series of requests and the impact of different seek vs rotational speeds on random and sequential request streams. Run the simulator with the `-G` option to observe what is going on. Answer the questions about the effect of different seek and rotational speeds but you do not need to compute times by hand — just observe the simulator. (Though you should understand how those values are computed — see the `README`.)

5. Do #4–5. Run the simulator with `-G` to see what is going on. Explain your answers.

6. Do #6. Run the simulator with `-G` to see what is going on. Explain your answers.

7. Do #8. Review the mini introduction to `bash` at the end of homework 3 for a reminder on how to automate running the simulator for many different window sizes. Two additional tips: {*low*..*high*..*step*} yields a sequence of values from *low* to *high* incremented by *step* rather than 1, and `echo -n` *string* prints output without a trailing newline. (Put *string* in double quotes if it contains spaces.) This is handy to print out the window size and the simulator's timing output on the same line.

8. Do #9. Create a series of 10-15 requests to demonstrate how SATF can starve a particular request. Address both how BSATF performs compared to SATF on that particular sequence and more generally with a random series of requests. For the latter, repeat #8 using BSATF and compare the results.

A threads-bugs

This homework lets you play around with a number of ways to implement a small, deadlock-free vector object in C. The vector object is quite limited (e.g., it only has `add()` and `init()` functions) but is just used to illustrate different approaches to avoiding deadlock.

Some files that you should pay attention to are as follows. They, in particular, are used by all the variants in this homework.

- `common_threads.h`: The usual wrappers around many different pthread (and other) library calls, so as to ensure they are not failing silently
- `vector-header.h`: A simple header for the vector routines, mostly defining a fixed vector size and then a struct that is used per vector (`vector_t`)
- `main-header.h`: A number of global variables common to each different program
- `main-common.c`: Contains the main() routine (with arg parsing) that initializes two vectors, starts some threads to access them (via a worker() routine), and then waits for the many vector_add()'s to complete

The variants of this homework are found in the following files. Each takes a different approach to dealing with concurrency inside a "vector addition" routine called `vector_add()`; examine the code in these files to get a sense of what is going on. They all use the files above to make a complete runnable program.

The relevant files:

- `vector-deadlock.c`: This version blithely grabs the locks in a particular order (dst then src). By doing so, it creates an "invitation to deadlock", as one thread might call `vector_add(v1, v2)` while another concurrently calls `vector_add(v2, v1)`.
- `vector-global-order.c`: This version of `vector_add()` grabs the locks in a total order, based on address of the vector.
- `vector-try-wait.c`: This version of `vector_add()` uses `pthread_mutex_trylock()` to attempt to grab locks; when the try fails, the code releases any locks it may hold and goes back to the top and tries it all over again.
- `vector-avoid-hold-and-wait.c`: This version ensures it can't get stuck in a hold and wait pattern by using a single lock around lock acquisition.
- `vector-nolock.c`: This version doesn't even use locks; rather, it uses an atomic fetch-and-add to implement the `vector_add()` routine. Its semantics (as a result) are slightly different.

Type `make` (and read the `Makefile`) to build each of five executables.

```
prompt> make
```

Then you can run a program by simply typing its name:

```
prompt> ./vector-deadlock
```

Each program takes the same set of arguments (see main-common.c for details):

- `-d`: This flag turns on the ability for threads to deadlock. When you pass `-d` to the program, every other thread calls `vector_add()` with the vectors in a different order, e.g., with two threads, and `-d` enabled, Thread 0 calls `vector_add(v1, v2)` and Thread 1 calls `vector_add(v2, v1)`
- `-p`: This flag gives each thread a different set of vectors to call add upon, instead of just two vectors. Use this to see how things perform when there isn't contention for the same set of vectors.
- `-n num_threads`: Creates some number of threads; you need more than one to deadlock.
- `-l loops`: How many times should each thread call add?
- `-v`: Verbose flag: prints out a little more about what is going on.
- `-t`: Turns on timing and shows how long everything took.

B file-disks

This homework uses `disk.py` to familiarize you with how a modern hard drive works. It has a lot of different options, and unlike most of the other simulations, has a graphical animator to show you exactly what happens when the disk is in action.

Note: there is also an experimental program, `disk-precise.py`. This version of the simulator uses the python Decimal package for precise floating point computation, thus giving slightly better answers in some corner cases than `disk.py`. However, it has not been very carefully tested, so use at your own caution.

Let's do a simple example first. To run the simulator and compute some basic seek, rotation, and transfer times, you first have to give a list of requests to the simulator. This can either be done by specifying the exact requests, or by having the simulator generate some randomly.

We'll start by specifying a list of requests ourselves. Let's do a single request first:

```
prompt> disk.py -a 10
```

At this point you'll see:

```
...
REQUESTS [br '10']

For the requests above, compute the seek, rotate, and transfer times.
Use -c or the graphical mode (-G) to see the answers.
```

To be able to compute the seek, rotation, and transfer times for this request, you'll have to know a little more information about the layout of sectors, the starting position of the disk head, and so forth. To see much of this information, run the simulator in graphical mode (-G):

```
prompt> ./disk.py -a 10 -G
```

At this point, a window should appear with our simple disk on it. The disk head is positioned on the outside track, halfway through sector 6. As you can see, sector 10 (our example sector) is on the same track, about a third of the way around. The direction of rotation is counter-clockwise. To run the simulation, press the "s" key while the simulator window is highlighted.

When the simulation completes, you should be able to see that the disk spent 105 time units in rotation and 30 in transfer in order to access sector 10, with no seek time. Press "q" to close the simulator window.

To calculate this (instead of just running the simulation), you would need to know a few details about the disk. First, the rotational speed is by default set to 1 degree per time unit. Thus, to make a complete revolution, it takes 360 time units. Second, transfer begins and ends at the halfway point between sectors. Thus, to read sector 10,

the transfer begins halfway between 9 and 10, and ends halfway between 10 and 11. Finally, in the default disk, there are 12 sectors per track, meaning that each sector takes up 30 degrees of the rotational space. Thus, to read a sector, it takes 30 time units (given our default speed of rotation).

With this information in hand, you now should be able to compute the seek, rotation, and transfer times for accessing sector 10. Because the head starts on the same track as 10, there is no seek time. Because the disk rotates at 1 degree / time unit, it takes 105 time units to get to the beginning of sector 10, halfway between 9 and 10 (note that it is exactly 90 degrees to the middle of sector 9, and another 15 to the halfway point). Finally, to transfer the sector takes 30 time units.

Now let's do a slightly more complex example:

```
prompt> ./disk.py -a 10,11 -G
```

In this case, we're transferring two sectors, 10 and 11. How long will it take? Try guessing before running the simulation!

As you probably guessed, this simulation takes just 30 time units longer, to transfer the next sector 11. Thus, the seek and rotate times remain the same, but the transfer time for the requests is doubled. You can in fact see these sums across the top of the simulator window; they also get printed out to the console as follows:

```
...
Sector:  10  Seek:  0  Rotate:105  Transfer: 30  Total: 135
Sector:  11  Seek:  0  Rotate:  0  Transfer: 30  Total:  30
TOTALS       Seek:  0  Rotate:105  Transfer: 60  Total: 165
```

Now let's do an example with a seek. Try the following set of requests:

```
prompt> disk.py -a 10,18 -G
```

To compute how long this will take, you need to know how long a seek will take. The distance between each track is by default 40 distance units, and the default rate of seeking is 1 distance unit per unit time. Thus, a seek from the outer track to the middle track takes 40 time units.

You'd also have to know the scheduling policy. The default is FIFO, though, so for now you can just compute the request times assuming the processing order matches the list specified via the `-a` flag.

To compute how long it will take the disk to service these requests, we first compute how long it takes to access sector 10, which we know from above to be 135 time units (105 rotating, 30 transferring). Once this request is complete, the disk begins to seek to the middle track where sector 18 lies, taking 40 time units. Then the disk rotates to sector 18, and transfers it for 30 time units, thus completing the simulation. But how long does this final rotation take?

To compute the rotational delay for 18, first figure out how long the disk would take to rotate from the end of the access to sector 10 to the beginning of the access to sector 18, assuming a zero-cost seek. As you can see from the simulator, sector 10 on the outer track is lined up with sector 22 on the middle track, and there are 7 sectors separating 22 from 18 (23, 12, 13, 14, 15, 16, and 17, as the disk spins counter-clockwise). Rotating through 7 sectors takes 210 time units (30 per sector). However, the first part of this rotation is actually spent seeking to the middle track, for 40 time units. Thus, the actual rotational delay for accessing sector 18 is 210 minus 40, or 170 time units. Run the simulator to see this for yourself; note that you can run without graphics and with the "-c" flag to just see the results without seeing the graphics.

```
prompt> ./disk.py -a 10,18 -c
...
Sector:  10  Seek:  0  Rotate:105  Transfer: 30  Total: 135
Sector:  18  Seek: 40  Rotate:170  Transfer: 30  Total: 240
TOTALS       Seek: 40  Rotate:275  Transfer: 60  Total: 375
```

You should now have a basic idea of how the simulator works. The questions below will explore some of the different options, to better help you build a model of how a disk really works.