This homework covers OSTEP chapters 21 and 22. It is due in class Friday, March 27. Hand in a hardcopy of your answers in class.

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 `vm-beyondphys` and `vm-beyondphys-policy` (and their contents) from `/classes/cs331` to your `~/cs331` directory. (You won't be writing any code, so the copied directories don't need to go into your `workspace` directory.)
- The `vm-beyondphys` directory contains a C program `mem.c`, a `Makefile`, and a README with instructions for using the program. Compile `mem.c` by changing to the `vm-beyondphys` directory and running `make`. You can also find the text of the `README` in the appendix section A at the end of this document.
- The `vm-beyondphys-policy` directory contain a python program (`paging-policy.py`) and a `README` with instructions for using the program. You can also find the text of the `README` in the appendix section B at the end of this document.

Exercises

Chapter 21 homework problems (at the very end of chapter 21) —

1. Do #1. You don't need to hand in anything for this, just make sure you understand what you are looking at with `vmstat`.
2. Do #2–4: Answer the questions posed and explain why what you see happening is happening. Also identify which VDI pool you are using (GPU or non-GPU) or if you are using the dual-boot computers in Demarest 002.

   Eventually you will find that `mem` terminates with the message "Killed". About when does that occur? (in terms of the memory you are asking `mem` to allocate) (This happens because Linux has an "out of memory killer" which terminates memory hog processes in order to protect the rest of the system from thrashing.)

   Notes:

- To find out the available memory on the system, use `cat /proc/meminfo` as directed and look for the "MemTotal" (it will probably be the first line).
- Increase the memory usage slowly — start with telling `mem` to use about half the available memory, then increase by 1000 or so at a time.
- `mem` is very resource-intensive. Let a run go 10-15 iterations, but don't leave `mem` running longer than that in order to minimize the impact on the system. Also make sure to ctrl-C it before logging out or disconnecting from the VDI.
- You may notice that once allocated, swap space usage doesn't go back to 0 when `mem` terminates. Work up from smaller amounts of memory to larger ones as you observe the behavior, and disconnect and reconnect to the VDI as you move from #2 to #3 to #4 so that you are starting each task with a fresh environment.

3. Do #5: Rather than picking sizes that fit into memory and exceed memory as the problem directs, pick several sizes covering the range between "comfortably fits in memory" and "just barely runs without getting killed". You don't need to plot your data, but do note the bandwidth values and behavior you are seeing in your writeup. Can you explain what is going on?

Chapter 22 homework problems (at the very end of chapter 22) —

4. Do #1 for practice — make sure you understand the concepts (how each of the page replacement policies work, how to compute the hit rate) but you don't need to hand in your solutions. (Use `-c` to check your answers.)

5. Repeat #1 with the CLOCK policy using 1 clock bit, a cache size of 5, and the following arguments: `-s 0 -n 20`. Using 1 clock bit means that the algorithm works as discussed in class — there is a single "use" or "reference" bit that is set when the page is accessed (read or written) and cleared when the clock hand sweeps past the page. Sweep the clock hand through the physical frames in the cache rather than maintaining a separate list of pages; start the hand at 0 and advance it to the next slot when a page is replaced. Use `-c` to check your answers; you do not need to hand in your solutions.

6. Do #2, with an addition: describe the worst-case address reference streams for the three policies listed plus CLOCK and explain what about those sequences make them the worst cases. (MRU is *most-recently-used* — the last (most recent) page accessed is the first page evicted.) Then, how much bigger a cache is needed to improve performance, and why?

7. MRU replaces the most recently used page first, which is the complete opposite of LRU and seems counterintuitive. Does MRU *ever* perform well? What would be a best-case scenario for it? Describe the best-case address reference stream for MRU and explain what about that sequence makes it the best case.

8. Do #3, comparing the algorithms OPT, FIFO, LRU, CLOCK (with 1 clock bit). Note that by default, `paging-policy.py` generates random addresses so you do *not* need to write a python script to do this as the problem says — just run the simulator without specifying a particular address sequence. Use a cache size of 5 and generate a large number of addresses (100000). Use `-c` to have it print the resulting hit rate — you'll want to run with `-N` (for "no trace") to turn off the detailed output and just get the final stats.

9. The provided file `ls-trace` contains page references (one per line) generated from running the command `ls` as described in #5.

   (a) Take a look at the file by running the following command line from the directory containing `ls-trace`:

   ```
   less ls-trace
   ```

   `less` is a pager — it lets you view text files one screenful (page) at a time. Press the space bar to move forward one page, 'b' to move backward one page, and 'q' to quit. Look up `less` in the system man pages to find out more.

   (There's nothing to hand in for this part, just look at the file so you know what you are working with.)

   (b) How many page references are there in `ls-trace`? Use the following command line to find out:

   ```
   wc ls-trace
   ```

   Look up `wc` in the system man pages to understand what this is doing and how to interpret the output.

   (c) How many unique pages are referenced? Use the following command line to find out:

   ```
   sort -n ls-trace | uniq -c | wc
   ```

   To understand what this is doing, try the following command lines to examine the parts of the whole command line. Also look up `sort` and `uniq` in the system man pages.

   ```
   sort -n ls-trace | less
   sort -n ls-trace | uniq -c | less
   ```

   (d) Does this trace exhibit locality? Explain why or why not. Use the following command line:

   ```
   uniq -c ls-trace | less
   ```

(e) Do #4, using the addresses in `ls-trace` instead of generating your own as the problem says. Use a cache size of 5 and compare OPT, RAND, FIFO, LRU, and CLOCK with 1 clock bit.

Then try CLOCK with different numbers of clock bits — 1, 2, 3, etc. How does it compare to LRU now? Explain what you observe about the hit rate as you increase the number of clock bits. (What happens and why?)

About clock bits: The clock algorithm discussed in class can be generalized to allow finer-grained tracking of page usage — instead of just setting or clearing a single use bit, increment a reference counter when a page is referenced and decrement the counter each time the clock hand sweeps past the page. Pages are eligible replacement when the reference count is 0. In the simulator, the number of clock bits sets the upper limit on the reference count — 1 clock bit means the algorithm works as discussed in class, while 3 clock bits means that the reference count is incremented to a max of 3.

A mem

In this homework, you'll be investigating swap performance with a simple program found in `mem.c`. The program is really simple: it just allocates an array of integers of a certain size, and then proceeds to loop through it (repeatedly), incrementing each value in the array.

Type `make` to build it (and look at the file `Makefile` for details about how the build works).

Then, type `./mem` followed by a number to run it. The number is the size (in MB) of the array. Thus, to run with a small array (size 1 MB):

```
prompt> ./mem 1
```

and to run with a larger array (size 1 GB):

```
prompt> ./mem 1024
```

The program prints out the time it takes to go through each loop as well as the bandwidth (in MB/s). Bandwidth is particularly interesting to know as it gives you a sense of how fast the system you're using can move through data; on modern systems, this is likely in the GB/s range.

Here is what the output looks like for a typical run:

```
prompt> ./mem 1000
allocating 1048576000 bytes (1000.00 MB)
  number of integers in array: 262144000
loop 0 in 448.11 ms (bandwidth: 2231.61 MB/s)
loop 1 in 345.38 ms (bandwidth: 2895.38 MB/s)
loop 2 in 345.18 ms (bandwidth: 2897.07 MB/s)
loop 3 in 345.23 ms (bandwidth: 2896.61 MB/s)
^C
prompt>
```

The program first tells you how much memory it allocated (in bytes, MB, and in the number of integers), and then starts looping through the array. The first loop (in the example above) took 448 milliseconds; because the program accessed the 1000 MB in just under half a second, the computed bandwidth is (not surprisingly) just over 2000 MB/s.

The program continues by doing the same thing over and over, for loops 1, 2, etc.

Important: to stop the program, you must kill it. This task is accomplished on Linux (and all Unix-based systems) by typing control-C (^C) as shown above.

Note that when you run with small array sizes, each loop's performance numbers won't be printed. For example:

```
prompt>  ./mem 1
allocating 1048576 bytes (1.00 MB)
  number of integers in array: 262144
loop 0 in 0.71 ms (bandwidth: 1414.61 MB/s)
loop 607 in 0.33 ms (bandwidth: 3039.35 MB/s)
loop 1215 in 0.33 ms (bandwidth: 3030.57 MB/s)
loop 1823 in 0.33 ms (bandwidth: 3039.35 MB/s)
^C
prompt>
```

In this case, the program only prints out a sample of outputs, so as not to flood the screen with too much output.

The code itself is simple to understand. The first important part is a memory allocation:

```
// the big memory allocation happens here
int *x = malloc(size_in_bytes);
```

Then, the main loop begins:

```
while (1) {
x[i++] += 1; // main work of loop done here.
```

The rest is just timing and printing out information. See `mem.c` for details.

Much of the homework revolves around using the tool vmstat to monitor what is happening with the system. Read the vmstat man page (type `man vmstat`) for details on how it works, and what each column of output means.

B paging-policy.py

This simulator, paging-policy.py, allows you to play around with different page-replacement policies. For example, let's examine how LRU performs with a series of page references with a cache of size 3:

```
0 1 2 0 1 3 0 3 1 2 1
```

To do so, run the simulator as follows:

```
prompt> ./paging-policy.py --addresses=0,1,2,0,1,3,0,3,1,2,1
                           --policy=LRU --cachesize=3 -c

And what you would see is:

ARG addresses 0,1,2,0,1,3,0,3,1,2,1
ARG numaddrs 10
ARG policy LRU
ARG cachesize 3
ARG maxpage 10
ARG seed 0

Solving...

Access: 0 MISS LRU->      [br 0]<-MRU Replace:- [br Hits:0 Misses:1]
Access: 1 MISS LRU->   [br 0, 1]<-MRU Replace:- [br Hits:0 Misses:2]
Access: 2 MISS LRU->[br 0, 1, 2]<-MRU Replace:- [br Hits:0 Misses:3]
Access: 0 HIT  LRU->[br 1, 2, 0]<-MRU Replace:- [br Hits:1 Misses:3]
Access: 1 HIT  LRU->[br 2, 0, 1]<-MRU Replace:- [br Hits:2 Misses:3]
Access: 3 MISS LRU->[br 0, 1, 3]<-MRU Replace:2 [br Hits:2 Misses:4]
Access: 0 HIT  LRU->[br 1, 3, 0]<-MRU Replace:2 [br Hits:3 Misses:4]
Access: 3 HIT  LRU->[br 1, 0, 3]<-MRU Replace:2 [br Hits:4 Misses:4]
Access: 1 HIT  LRU->[br 0, 3, 1]<-MRU Replace:2 [br Hits:5 Misses:4]
Access: 2 MISS LRU->[br 3, 1, 2]<-MRU Replace:0 [br Hits:5 Misses:5]
Access: 1 HIT  LRU->[br 3, 2, 1]<-MRU Replace:0 [br Hits:6 Misses:5]
```

The complete set of possible arguments for paging-policy is listed on the following page, and includes a number of options for varying the policy, how addresses are specified/generated, and other important parameters such as the size of the cache.

```
prompt> ./paging-policy.py --help
Usage: paging-policy.py [options]

Options:
```

```
-h, --help      show this help message and exit
-a ADDRESSES, --addresses=ADDRESSES
                a set of comma-separated pages to access;
                -1 means randomly generate
-f ADDRESSFILE, --addressfile=ADDRESSFILE
                a file with a bunch of addresses in it
-n NUMADDRS, --numaddrs=NUMADDRS
                if -a (--addresses) is -1, this is the
                number of addrs to generate
-p POLICY, --policy=POLICY
                replacement policy: FIFO, LRU, MRU, OPT,
                                    UNOPT, RAND, CLOCK
-b CLOCKBITS, --clockbits=CLOCKBITS
                for CLOCK policy, how many clock bits to use
-C CACHESIZE, --cachesize=CACHESIZE
                size of the page cache, in pages
-m MAXPAGE, --maxpage=MAXPAGE
                if randomly generating page accesses,
                this is the max page number
-s SEED, --seed=SEED  random number seed
-N, --notrace   do not print out a detailed trace
-c, --compute   compute answers for me
```

Note that "clock bits" is actually referring to the maximum reference count stored for a page, not the number of bits that would be used to store that count. "-b 2" means a max reference count of 2, not 4.

As usual, "-c" is used to solve a particular problem, whereas without it, the accesses are just listed (and the program does not tell you whether or not a particular access is a hit or miss).

To generate a random problem, instead of using "-a/–addresses" to pass in some page references, you can instead pass in "-n/–numaddrs" as the number of addresses the program should randomly generate, with "-s/–seed" used to specify a different random seed. For example:

```
prompt> ./paging-policy.py -s 10 -n 3
.. .
Assuming a replacement policy of FIFO, and a cache of size 3 pages,
figure out whether each of the following page references hit or miss
in the page cache.

Access: 5  Hit/Miss?  State of Memory?
Access: 4  Hit/Miss?  State of Memory?
Access: 5  Hit/Miss?  State of Memory?
```

As you can see, in this example, we specify "-n 3" which means the program should generate 3 random page references, which it does: 5, 7, and 5. The random seed is also specified (10), which is what gets us those particular numbers. After working this out yourself, have the program solve the problem for you by passing in the same arguments but with "-c" (showing just the relevant part here):

```
prompt> ./paging-policy.py -s 10 -n 3 -c
...
Solving...

Access: 5 MISS FirstIn->   [br 5] <-Lastin Replace:- [br Hits:0 Misses:1]
Access: 4 MISS FirstIn->[br 5, 4] <-Lastin Replace:- [br Hits:0 Misses:2]
Access: 5 HIT  FirstIn->[br 5, 4] <-Lastin Replace:- [br Hits:1 Misses:2]
```

The default policy is FIFO, though others are available, including LRU, MRU, OPT (the optimal replacement policy, which peeks into the future to see what is best to replace), UNOPT (which is the pessimal replacement), RAND (which does random replacement), and CLOCK (which does the clock algorithm). The CLOCK algorithm also takes another argument (-b), which states how many bits should be kept per page; the more clock bits there are, the better the algorithm should be at determining which pages to keep in memory. Note that "clock bits" is actually referring to the maximum reference count stored for a page, not the number of bits that would be used to store that count. "-b 2" means a max reference count of 2, not 4.

Other options include: "-C/–cachesize" which changes the size of the page cache; "-m/–maxpage" which is the largest page number that will be used if the simulator is generating references for you; and "-f/–addressfile" which lets you specify a file with addresses in them, in case you wish to get traces from a real application or otherwise use a long trace as input.

One last piece of fun: why are these two examples interesting?

```
./paging-policy.py -C 3 -a 1,2,3,4,1,2,5,1,2,3,4,5
```

and

```
./paging-policy.py -C 4 -a 1,2,3,4,1,2,5,1,2,3,4,5
```