CS24 Final 2016 – COVER SHEET

This final requires you to work with several programs, and to answer various questions. As with the homework assignments, a tarball is provided, **`cs24fin.tgz`**. When you start the final, you should download and extract this tarball, and work in the directories specified by the problems.

When you have finished the final exam, you can re-archive this directory into a tarball and submit the resulting file:

> *(From the directory containing **`cs24fin`**; replace **`username`** with your username.)*
> `tar -czvf cs24fin-username.tgz cs24fin`

Important Note!

If you decide to not follow the filenames specified by the final exam problems, or your tarball is really annoying in some way (e.g. the permissions within the tarball have to be overridden by the TAs before they can grade your work!), you may lose some points. If you think you might need help creating this tarball properly, talk to Donnie or a TA; we will be happy to help.

Final Exam Rules

The time limit for this final is 6 hours, in multiple sittings. The main rule is that if you are focusing on the final, the clock should be running. If you want to take the entire final in one sitting, or if you want to take a break (or a nap) after solving a problem, do whatever works best for you. Just make sure to track your time, and only spend 6 hours working on the exam.

The final is a bit long this year. You will likely need the full 6 hours. Don't get bogged down in any specific problem. A suggested approach is to allocate 1 hour for each part, do as much as you can, then go on to the next part. Always solve the easy problems first. Keep explanations concise and to the point; don't be over-verbose. Use left-over time to complete the harder problems.

If you need to go overtime, just indicate where time ran out. The general policy is 50% credit for up to 1 hour after the deadline, and 0% credit thereafter, although this may be waived in special circumstances.

You may use any official course material on the exam. You may refer to the book, the lecture slides, your own assignments, solution sets, associated reference material, and so forth.

You are encouraged to use a computer for this final.

No collaboration. You may not talk to another student about the contents of the final exam until both of you have completed it. You may talk to a TA about the final after you have completed it. You may also talk to a TA if you need help packaging up your final exam files, as mentioned above. You may not search for solutions to problems on the Internet (e.g. Wikipedia, etc.) or any other external source.

Request any needed clarifications directly from Donnie. The TAs weren't responsible for writing the final exam, so they can't answer your questions very well. Feel free to contact Donnie if you have any questions. If you can't make progress while you wait for an answer, stop your timer.

WHEN YOU ARE READY TO TAKE THE FINAL EXAM, YOU MAY GO ON TO THE NEXT PAGE! GOOD LUCK!

Part 1: Prime Number Sieve *(18 points)*

Files for this problem are in the ***cs24fin/primes*** *folder of the archive.*

One of the fastest ways to generate prime numbers is the Sieve of Eratosthenes. You are probably already familiar with this mechanism, but here is an outline of how it works:

1. Start by generating a list of numbers, from 2 up to the maximum value *N*.
2. Next, iterate over the list in increasing order. For each number *P*:
   - If *P* is not marked as composite, *P* is prime. Output *P*, and mark all multiples of *P* as composite. (We can start marking multiples with $P \times P$, since lower multiples of *P* will have already been marked composite.)
   - Advance *P* down the list!
3. Once $P >$ **sqrt**(*N*), the list only contains prime numbers.

For example, if we had this sequence:

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, …

We would start with *P* = 2, which is prime, and then we would cross out all multiples of 2, starting with 2×2:

2, 3, _ , 5, _ , 7, _ , 9, _ , 11, _ , 13, _ , 15, _ , 17, _ , 19, _ , …

Then we advance to *P* = 3, which is also prime, and we cross out all multiples of 3, starting with 3×3 (notice that 3×2 has already been crossed out by the previous step):

2, 3, _ , 5, _ , 7, _ , _ , _ , 11, _ , 13, _ , _ , _ , 17, _ , 19, _ , …

Then we advance to *P* = 4. This value has already been crossed out, so 4 is not prime. And so forth.

(Wikipedia has an excellent animation of a Sieve of Eratosthenes, if you want to check it out.)

You will find a simple implementation of this prime number sieve in the file **psieve.c** in the **primes** directory, which you can build with the provided **Makefile**. This program is capable of finding all primes up to $2^{32} - 1$. When you run it, you can specify the upper bound, or if you don't then it will use 10^9 as the upper bound:

```
donnie$ ./psieve 100
Using maximum value of 100
 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Total primes found:  25
donnie$ ./psieve
Using maximum value of 1000000000
(...lots of output...)
```

This program includes several simple optimizations to improve performance. For example, it doesn't track even values at all, for obvious reasons. Additionally, instead of creating a list of numbers, it uses a *bit vector* to record the primality of each number.

A bit vector is a vector where each element is a Boolean value of either 0 or 1. Since we know that all elements are bits, we can pack multiple elements into a single unsigned integer value. For example, the bit-vector in the prime sieve expects **unsigned int** to be 32-bits in size, so each **unsigned int** can store 32 values. The bit-vector has operations to set and retrieve bit *i* of the

vector; bit *i* is stored in the (*i* **div** 32)nd **`unsigned int`** element of the vector, and the specific bit is the (*i* **mod** 32)nd bit within that **`unsigned int`** value. (See **`bitvector.c`** for details.)

As stated earlier, this prime sieve treats 2 and all even values specially; it doesn't represent them at all. Thus, the bits in the vector only represent the odd values from 3 upward:

Bit Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	…
Number	3	5	7	9	11	13	15	17	19	21	23	25	27	29	…

Thus, given a number *P*, we can map *P* to a corresponding bit *B* in the bit-vector with the expression *B* = (*P* – 3) / 2. (The compiler can compute this very quickly; division or modulo by a power of two can be done with bit-manipulation operations, not actual division.) If the bit stored for *P* is 1 then the number *P* is prime; if the bit for *P* is 0 then the number is composite.

In this problem you will analyze the cache performance characteristics of this prime-number sieve. You can assume typical IA32 cache characteristics, even if this doesn't match your specific machine configuration (it will still be close):

- All cache lines are 64 bytes (16 **`unsigned int`** values) in size
- The L1 data cache is 8-way set associative, and has a total of 32KB of space
- The L2 shared cache is 8-way set associative, and has a total of 1MB of space

Answer the following questions in a file **`primes/problems.txt`**:

1. Determine a maximum value *N* for your machine, where **`psieve`** takes about 1 minute to execute. (You shouldn't use a virtual machine for this problem, if you can avoid it.) You can use this command to perform your timing: **`time ./psieve N > /dev/null`**

   Start with *N* at 500 million, and increase it by 500-million increments until you get a run-time of approximately 1 minute, or you reach $N = 4 \times 10^9$. (If this is still less than a minute, oh well!) Record a brief description of your platform[1], and also this number *N* and your total run-time in your answers.

   Finally, how many bytes will the bit-vector take to represent the values *P* from 3..*N*, for the *N* you have just found? (Don't forget that even numbers are not represented in the bit-vector.)

   (2 points)

2. Study the code in **`psieve.c`**, and characterize the cache performance behavior of the program:

   a. What stride reference pattern does the program use when it accesses the bit-vector? Does it use this reference pattern through the entirety of its execution, or does it vary as the program executes? Explain your answer. *(4 points)*

   b. What portions of the program will demonstrate good locality, and therefore leverage caches heavily, and what portions of the program are likely to suffer cache misses? Describe how the cache performance might change over time, as the program executes. Finally, make sure to include the size of the program's data-set in your discussion, since this will affect what caches can be utilized. *(4 points)*

---

[1] Include each of these things (if you don't know one or more of them, just include the things you know): processor, amount of physical memory, system speed, and operating system.

3. Suggest modifications that can be made to the prime-number sieve that will improve the overall performance of the program. You should focus primarily on improving any issues you previously identified in the program's cache performance, but other changes (e.g. tweaks to the algorithm being used, structure of the program, etc.) are also acceptable. If you wish to add lists or other data structures to the program, this is fine, but you must retain the basic feature that a bit-vector is used to implement the sieve. *(To be clear, we don't expect you to implement your suggestions; you won't have time to do that.)*

   Your suggestions should be specific – at the level that a reasonably proficient programmer could implement the changes you describe. If a change introduces any challenges, e.g. subtle effects on the way the program must process its data, you must identify such issues and how to resolve them in your comments.

   (8 points)

Part 2: Lock-Free Concurrency *(24 points)*

*Files for this problem are in the **`cs24fin/cas`** folder of the archive.*

In Assignment 7 you explored two basic lock-based synchronization mechanisms, a simple binary mutex, and a semaphore. Locks are widely used for synchronizing concurrent access to data structures so that programs don't generate spurious results. However, they impose a certain overhead, both when locks are successfully acquired (lock overhead), and when threads must wait for locks (lock contention).

More recently, systems are being constructed using *lock-free* approaches to thread synchronization. Most such systems rely on the processor to provide an atomic "compare and swap" instruction, which performs these operations atomically:

```
if target == old_value:
      target = new_value
```

In other words, if a target variable (e.g. stored in memory) holds a specific "old value," change the variable to hold the specified "new value." If, however, the target variable's value is no longer the expected "old value," the assignment is not performed and the target variable is left unchanged.

The IA32 ISA provides the **`cmpxchg`** instruction which implements this operation; by specifying a **`lock`** prefix, the instruction will execute atomically even in the context of multiple processors. A program can determine if the **`cmpxchg`** has succeeded or failed by looking at the Zero Flag; ZF will be 1 if the operation succeeds, or 0 if the operation fails.

a) In the file **`cmp_swap.s`**, write an implementation of the **`compare_and_swap()`** function, as specified in **`cmp_swap.h`**. Your implementation must use the "**`lock cmpxchg`**" instruction as specified above. Make sure to follow all good coding practices. Comment your implementation, including a comment header that specifies the location of arguments on the stack. *(8 points)*

b) In the file **`testcas.c`**, write some simple tests to ensure that your **`compare_and_swap()`** function works properly. You really only need to test two major cases – when the compare-and-swap will be successful, and when it will not be successful – but make sure you verify all important behaviors, including that the target variable did/did not change, and that the function's return-value matches the expected results.

Your test program's output must clearly indicate what tests are being performed, whether or not they pass, and if they fail, exactly what failures occurred. Also, your test program should run all tests and identify all failures, even when one or more failures are detected. (Don't just stop at the first failure.)

(4 points)

A **`Makefile`** is provided that will build **`testcas`**, as well as the program described below.

Creating lock-free data structures is incredibly complex, and very easy to get wrong. Therefore, we will stick to a very simple construct, a thread-safe accumulator. You will find two files, **`accum.c`** and **`testaccum.c`**, in the **`cas`** directory, which define an accumulator and then test its thread-safety by adding values from multiple threads. (How many values, and how many threads, are specified at the top of **`testaccum.c`**.) If you run this program, you will find that it frequently generates errors:

```
ERROR:  Accumulator holds 1083943024, but the expected sum is 2100206397
ERROR:  Accumulator holds 1569161772, but the expected sum is 2099981015
```

The accumulator is buggy because it is not thread-safe; you must fix it using your implementation of **`compare_and_swap()`**. (Note that it is possible the program will print "SUCCESS" from time to time due to sheer luck, but the parameters are chosen so that it will almost always report an error.)

c) In the file **`accum.c`** you will find the implementation of **`add_to_accum()`**. Reimplement this function using your **`compare_and_swap()`** primitive, so that the accumulator will work properly when multiple threads add to it concurrently. Explain your implementation in comments in the code. *(8 points)*

Finally, there are multiple categories of lock-free algorithms:

- A *wait-free* implementation provides a very strong guarantee, that every thread performing an operation always has a finite upper bound on the number of steps it will take to complete that operation, regardless of the scenario in which it is being used.

- A *lock-free* implementation provides a weaker guarantee, that the overall system will always make progress, but an individual thread can potentially take an unbounded number of steps to complete a given operation.

d) Would you describe your accumulator implementation as wait-free or lock-free? In other words, imagine that **`testaccum`** was reimplemented so that it runs forever, summing into the accumulator. Is it possible for a specific accumulator-thread to take an unbounded number of steps to complete an **`add_to_accum()`** operation? If so, explain a scenario that would produce this behavior. If not, explain why not. (Assume you have no control over thread-scheduling in the program.) Put your answer in the file **`cas/accum.txt`**. *(4 points)*

Part 3: Process Puzzlers *(12 points)*

Here are some short (but not simple) programs that use the UNIX process API. Answer these questions in the file **`cs24fin/puzzlers.txt`**. In these programs, assume that all system calls always complete successfully. **Full credit requires an explanation of your answers.** You can run these programs if you wish, but unless you explain the reasoning behind your answers, you won't receive full credit.

1. *(4 points)* What does this program print out? Does it print the same result every time it is run?

```
pid_t pid;

void handler1(int sig) {
    printf("zip");
    fflush(stdout);       /* Flushes the printed string to stdout */
    kill(pid, SIGUSR1);
}

void handler2(int sig) {
    printf("zap");
    exit(0);
}

int main() {
    signal(SIGUSR1, handler1);
    pid = fork();
    if (pid == 0) {
        signal(SIGUSR1, handler2);
        kill(getppid(), SIGUSR1);
        while(1) { };
    }
    else {
        pid_t p;
        int status;
        p = wait(&status);
        if (p > 0)
            printf("zoom");
    }
}
```

2. *(8 points)* Here is another program:

```
int main() {
    if (fork() == 0) {
        if (fork() == 0) {
            printf("3");
        }
        else {
            int status;
            pid_t pid = wait(&status);
            if (pid > 0)
                printf("4");
        }
    }
    else {
        if (fork() == 0) {
            printf("1");
            exit(0);
        }
        printf("2");
    }
    printf("0");
    return 0;
}
```

Of the following set of outputs, which ones could be generated by the program?

- 2030401
- 1234000
- 2300140
- 2034012
- 3200410

Part 4: Lottery Scheduling *(20 points)*

An unusual approach to the process-scheduling problem is to grant every process some number of "lottery tickets," and then have the scheduler draw a random ticket to select the next process to run. The number of tickets assigned to each process can be varied dynamically, changing the likelihood that each process will be selected to run next. Because of this, lottery scheduling can approximate

many other scheduling algorithms, simply by carefully controlling how many tickets each process is granted by the scheduler.

Note that only the ready processes participate in the lottery, because only the ready processes can run if they "win" the CPU. Blocked and suspended processes do not participate in the lottery.

As with other scheduling algorithms, lottery schedulers can impose a maximum time-slice on the currently running process, so that no process holds the CPU for too long.

Answer the following questions in a file **`cs24fin/lottery.txt`**. Make sure to clearly indicate which problem each answer corresponds to.

1. In what situations would the lottery scheduler be unfair? What basic conditions must we satisfy to ensure that the lottery scheduler is fair? *(2 points)*

2. The UNIX **`nice`** utility allows the priority of a given process to be raised or lowered. For example, given two processes with identical behaviors, the "nicer" process should receive CPU time less frequently than the "less nice" process. How can this be achieved with the lottery-scheduling mechanism? *(3 points)*

3. To ensure a responsive system, the process scheduler should ensure that interactive processes have a higher priority than compute-intensive processes. Describe how the lottery scheduler can ensure that interactive processes are higher priority than compute-intensive ones, based on e.g. how much of the time-slice the process consumes before blocking/yielding, or is preempted.

   Be specific in your answer. For example, interactive processes also vary in performance, and some interactive processes (e.g. text editors) might use far less of their time-slice than other interactive processes (e.g. graphical drawing tools). Additionally, an interactive process may change over time to being compute-intensive, or vice versa; you should endeavor to handle such scenarios in your solution.

   (Feel free to modify the lottery scheduler as you wish, but the basic approach of assigning tickets and having a random drawing to choose the next ready process must remain the same.)

   (10 points)

4. In light of your answer to the previous question, is it possible that the lottery scheduler might schedule one or more compute-intensive tasks before an interactive task that is also ready to run? If not, explain why not. If so, how might you resolve this issue? (Keep your answer to this question brief; we don't want you to spend a huge amount of time on this question.)

   (5 points)

Part 5: Page Replacement Policies and Scan Resistance *(26 points)*

Answers for the questions in this section should be put in the file ***`cs24fin/policies.txt`***.

Page replacement policies are a very important component of computing systems, since a well-designed policy can reduce the number of expensive page-faults. In general, policies must make a choice between placing more weight on *recency* of access, vs. *frequency* of access.

A good example of this issue is the Least Recently Used (LRU) policy. LRU makes eviction decisions entirely based on recency of access, and it doesn't weight frequency of access at all in its decisions. This makes LRU sensitive to *scans* – a sequence of page accesses where each page is accessed only once, and then not accessed again for a very long time. While this may seem like bad program behavior, it turns out to be very common in programs that must process very large data sets, such as database systems.

Typically, scans occur while other pages are also being accessed. For example, 10-20% of the pages being accessed might be more typical program behavior, displaying good spatial and temporal locality. The other 80-90% of the pages being accessed are part of the scan; each page is accessed once as part of the scan, and then not again.

Since LRU only considers the time of the most recent access, pages that are part of the scan will typically cause other more useful pages to be flushed out of memory because they have access times slightly further in the past. But, they will be needed again shortly, so evicting them increases the page-fault rate. Conversely, the scanned pages will not be accessed again for a long time, but because they have a recent access time, they will not be evicted until enough other pages have been loaded that their access time is far enough in the past. Thus, we say that LRU is not *scan-resistant*.

1. How would you characterize the following page replacement policies, in terms of how they consider recency of access, vs. frequency of access, vs. some combination of the two? Feel free to refer to the descriptions in the Assignment 8 write-up of the various paging policies; it shouldn't be necessary to refer to any external material on these policies.

   Be concise in your answers. *(3 points per policy)*

   a) FIFO

   b) Second Chance / Clock

   c) Aging

2. Is the Aging policy susceptible to scans? How susceptible is it? Explain your answer. *(4 points)*

A simple variant of the LRU policy is LRU-2, in which the time of the *second most recent* access to a page is considered. When a page must be evicted, the page with the oldest "second most recent access time" is evicted. A page that has only been accessed once is considered to have a "second most recent access time" of $-\infty$.

It should be obvious that LRU-2 is quite robust in the face of scans; pages that are accessed as part of the scan will have a "second most recent access time" of $-\infty$, and therefore will be very likely to leave the pool quickly. Any page that is accessed more than once while in memory will immediately be identified as not part of a scan, and will not be flushed out as the scan proceeds.

The problem with LRU-2 is that it is somewhat expensive to implement due to being ordered by the second most recent access time rather than the most recent access time. Thus, an alternative is suggested:

The policy (aptly named 2Q) maintains two queues: A1 (for pages that have been accessed once), and Am (for pages that have been accessed more than once). The queues are updated as follows:

- When a page is accessed, and it does not yet appear in A1 or Am, it is put on the front of A1.
- When a page is accessed, and it is already in A1 or Am, it is moved to the front of Am.

As usual, the overall number of resident pages is fixed; therefore, if pages must be evicted, they are evicted from the back of A1 first; if A1 is empty, pages are evicted from the back of Am.

a) How closely does this policy correspond to LRU-2? Justify your answer. *(4 points)*

b) You will notice that the above discussion of LRU, LRU-2 and 2Q assumes that every single page access can be detected, and that the policy can update its bookkeeping on every page access. This is of course not possible with virtual memory; it would be prohibitively slow.

   Therefore, design a version of the above policy that uses a periodic timer interrupt, and a page's "accessed" bit, to manage the above queues. Make sure to describe what should happen when a page fault occurs, as well as what should happen when the timer interrupt occurs. Finally, make sure to respect the hard limit on the total number of resident pages. *(5 points)*

c) How well or poorly does your modified policy approximate the 2Q algorithm? Can it effectively identify page accesses that are part of a scan? Can it effectively identify pages that are *not* part of a scan? *(4 points)*

You are now finished with CS24! Have a great summer!!!