CS24 Final 2015 – 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.

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.

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 aren't responsible for writing the final exam, and often don't have the foggiest idea what Donnie was thinking when he wrote that poorly worded problem. Feel free to contact Donnie if you have any questions. If you can't make any progress while you wait for an answer, stop your timer.

You may use a computer for this final. However, programming isn't as central a part of this exam, so you really won't need to compile and test your work.

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

Sorting Large Data Sets (22 points)

(Answer these questions in the ***`cs24fin/sort.txt`*** *file.)*

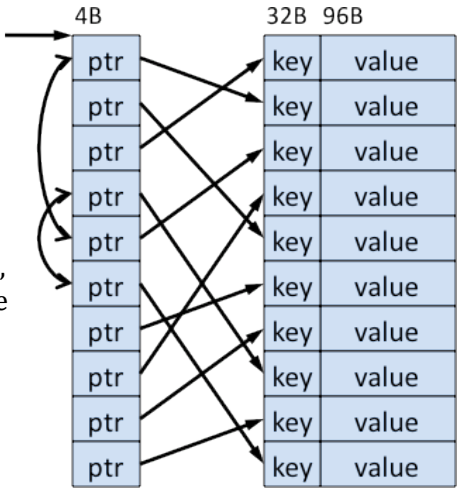
A project you are working on requires the sorting of a large number of data records, typically on the order of 25 million records at a time. Individual records are 128 bytes in size, and the first 32 bytes are the key for each record. The 32-byte keys are comprised of uniformly distributed sequences of the characters '**`A`**' through '**`Z`**'; we must order these keys by increasing key value, lexicographically.

Fortunately, the data set you must sort will always fit within 4GB, so a 32-bit address space is sufficient for your needs. However, this is still a lot of data to sort, and it will need to be done often, so you need to find an efficient way to do this.

We will assume that the records are already in memory in some unsorted order, and our primary responsibility is to generate a list of records sorted on the record key. We will also assume that the sorting algorithm we choose has an $O(N \times \mathbf{log}(N))$ time complexity, although we really don't want your analysis to get into the level of detail where the algorithm itself matters.

Approaches

The sorting algorithm must somehow present records in a sorted order, and a naïve approach would be to move the actual records themselves within memory as we sort them. Of course, this would maximize the amount of data that has to be moved around, so this approach can be discarded immediately.



Another approach would be to have an array of *pointers* (shown right), each of which points to one of the records in the data to sort. Then, the sorting algorithm simply needs to reorder the pointers, a much less copying-intensive operation.

1. Assume two records are compared with a function like this:

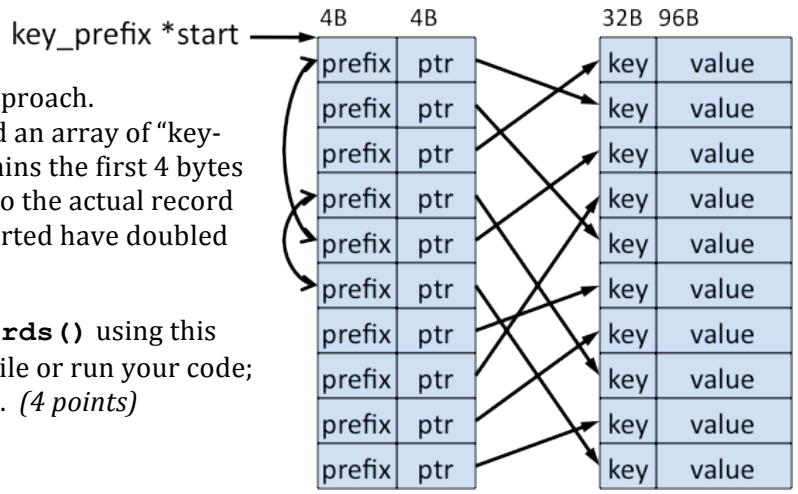
```
typedef struct record {
    char key[KEY_SIZE];
    char value[VALUE_SIZE];
} record;

int cmp_records(const record **rec_ptrs, int i, int j) {
     /* Elements are of type record *. */
     return memcmp(rec_ptrs[i]->key, rec_ptrs[j]->key, KEY_SIZE);
}
```

   (The **`memcmp()`** function simply iterates through the two memory regions, comparing bytes until it finds two that are different, or if it reaches the end of the regions. It returns a value < 0 if the first argument is "less than" the second argument, > 0 if the first argument is "greater than" the second argument, or 0 if the two byte-arrays are identical.)

   Given the size of the data being sorted, and how sorting algorithms generally operate, what do you think will be the typical case for how many cache-misses this function will generate when it is called? (Only consider data accesses, not stack or code accesses.) Describe your reasoning. *(5 points)*

2. How much do you think this approach will benefit from the hardware caches? Will it perform at closer to main-memory speeds, or closer to L1 cache speeds? Explain your answers. *(4 points)*



We can make a small tweak to the above approach. Instead of an array of record-pointers, build an array of "key-prefix" elements, where each element contains the first 4 bytes of the key (the prefix), as well as a pointer to the actual record in the data set. Thus, the elements being sorted have doubled in size from 4 to 8 bytes.

3. Write an implementation of **`cmp_records()`** using this new approach. You don't need to compile or run your code; minor syntax errors won't be penalized. *(4 points)*

```
typedef key_prefix {
    char prefix[4];
    record *ptr;
} key_prefix;

int cmp_records(const key_prefix *rec_ptrs, int i, int j) {
    /* TODO:  Implement */
}
```

4. We stated before that the keys are comprised of uniformly distributed sequences of the characters '**A**' through '**Z**'; given this detail, what benefit would we expect to see from the modified approach? How will this affect the number of cache-misses we should see when comparing records? *(5 points)*

5. If our keys are all very similar, will the modified approach be better or worse than the original approach? Explain your answer. *(4 points)*

Matrix Transpose (13 points)

*(Answer these questions in the **`cs24fin/matrix.txt`** file.)*

Consider a simple function to transpose a matrix:

```
#define SIZE ...  /* TODO:  SEE BELOW */

typedef int matrix_t[SIZE][SIZE];

void transpose(const matrix_t src, matrix_t dst) {
    int r, c;

    for (r = 0; r < SIZE; r++)
        for (c = 0; c < SIZE; c++)
            dst[c][r] = src[r][c];
}
```

Assume we are running this code on an IA32 processor with a single L1 cache that is 8-way set-associative with 64 cache sets, and 64-byte blocks (thus, 32KiB total L1 cache size). All values are 32-bit dword-aligned, and the processor accesses memory using 32-bit dwords. The cache is write-allocate/write-back, and assume the replacement policy is Least Recently Used (LRU).

1. What is the stride for memory accesses to **`src`**? What about for **`dst`**? (Give this answer in terms of dwords, i.e. array elements, not bytes.) *(2 pts)*

2. How far apart must two addresses be to refer to different cache blocks, but still map to the same cache set? (Give the minimum distance, in bytes.) Explain how you reached your answer. *(4 points)*

3. What is the smallest **`SIZE`** value that will result in the *worst-case* cache utilization of the above code, given that we only have one level of caching between the CPU and main memory? Make sure to include an explanation of how you reach your answer. *(7 points)*

Process Puzzlers *(20 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, we don't check for errors due to space; 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 the bulk of the points depend on correct explanation of the behaviors.

1. Consider this UNIX program:

```
int counter = 0;

int main() {
    int i;

    for (i = 0; i < 2; i++) {
        fork();
        counter++;
        printf("counter = %d\n", counter);
    }
    printf("counter = %d\n", counter);
    return 0;
}
```

   a) How many times would the value of **`counter`** be printed? Explain your answer. *(2 points)*

   b) What is the value of **`counter`** printed in the first line? Explain your answer. (If this cannot be determined, explain why.) *(2 points)*

   c) What is the value of **`counter`** printed in the last line? Explain your answer. (If this cannot be determined, explain why.) *(2 points)*

2. Here is another UNIX program that uses signals. **`SIGUSR1`** is a signal that can be used by user applications for any purpose; it must be manually sent via the **`kill()`** system call.

```
pid_t pid;
int counter = 0;

void handler1(int sig) {
    counter++;
    printf("counter = %d\n", counter);
    fflush(stdout); /* Flushes the printed string to stdout */
    kill(pid, SIGUSR1);
}

void handler2(int sig) {
    counter += 3;
    printf("counter = %d\n", counter);
```

```
    exit(0);
}

main() {
    signal(SIGUSR1, handler1);
    if ((pid = fork()) == 0) {
        signal(SIGUSR1, handler2);
        kill(getppid(), SIGUSR1);
        while(1) {};
    }
    else {
        pid_t p;
        int status;
        if ((p = wait(&status)) > 0) {
            counter += 2;
            printf("counter = %d\n", counter);
        }
    }
}
```

What is the output of this program? Is it the same every single time, or does it vary? Explain your answers. *(7 points)*

3. Here is one more UNIX program:

```
int main() {
    int val = 2;

    printf("%d", 0);
    fflush(stdout);

    if (fork() == 0) {
        val++;
        printf("%d", val);
        fflush(stdout);
    }
    else {
        val--;
        printf("%d", val);
        fflush(stdout);
        wait(NULL);
    }
    val++;
    printf("%d", val);
    fflush(stdout);
    exit(0);
}
```

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

- 01432
- 01342
- 03142
- 01234
- 03412

Make sure to explain your rationale for determining which outputs could be generated. *(7 points)*

Priority Scheduling and Priority Inversion (25 points)

Put answers to these questions in the file **`cs24fin/priority.txt`**

Priority scheduling is a very simple scheduling mechanism that is frequently used in event-driven systems to handle repeating tasks and/or events that might occur. Each process is assigned a fixed priority, and the scheduler always runs the highest priority process that is ready to run. The scheduler will only run a lower priority process if no higher priority processes are available to run; the premise is that the highest-priority process should never be preempted by a lower-priority process.

Note that the highest-priority process is never preempted; when it decides to stop running, that's when lower-priority processes get the CPU. It should be obvious that such a scheduler is unfair; if a high-priority process decides to run forever, no lower priority process will ever receive the CPU. Therefore, proper operation of such a system relies on careful construction, and on cooperative multitasking; processes must be designed to relinquish the processor when they have completed their work, or when they can no longer make progress.

You might think this would make this scheduling algorithm useless, but the ease of implementing such a scheduler makes it very attractive, especially for event-driven embedded systems. Like, for example, the Mars Pathfinder rover.

Such systems can also suffer from a special kind of issue called a **priority inversion**. For example, assume there are three processes running on the system, a high-priority process *H*, a low-priority process *L*, and a long-running medium-priority process *M*. Furthermore, assume that there is a shared resource *R* that both *H* and *L* must use, but they must lock *R* before they may use it.

Scoring: a = 5 pts, b = 2 pts, c = 5 pts, d = 8 pts, e = 5 pts.

a) Assume that *L* is running, and that it locks the shared resource *R*. While it is still holding the lock, both *M* and *H* become ready to run, but *M* will take a very long time to complete. The first thing that *H* tries to do is to acquire the lock on *R*. What happens after this? (Be detailed in your answer.) At what point will *H* be allowed to run? Why is this called a "priority inversion"?

b) Unfortunately, the Mars Pathfinder rover had exactly this scenario in its firmware.

   On the Pathfinder, the high-priority process was responsible for resetting a *watchdog timer*, among other things. Embedded systems often have a special hardware timer that only counts up; when it overflows, the entire system resets. During normal operation, the system software periodically clears this timer (called "petting the dog" or "kicking the dog," depending on how nice or mean the engineer is), avoiding the reset. However, if the system hangs, the watchdog timer will not be cleared; it will overflow, causing the entire system to reset. Ideally, after the reset, the system will return to normal operation. Thus, the watchdog timer is a simple mechanism to keep a system from becoming permanently hung.

   The medium priority process was responsible for weather measurements, communications with Earth, and so forth – all quite long-running tasks.

   How do you think this priority inversion issue affected the Mars Pathfinder rover?

c) A simple solution to the priority-inversion problem is called **priority donation**. If a process *P*1 attempts to lock a resource that another process *P*2 already holds, and *P*1's priority is higher

than P2's, the system can temporarily donate *P*1's priority to *P*2, until *P*2 releases the resource that *P*1 requires. At that point, *P*2's priority is lowered back to its original priority.

Explain how this approach resolves the priority-inversion problem outlined in part a.

Luckily for the Mars Pathfinder, there was a single configuration parameter in the software that could enable or disable priority donation; once this was enabled, the rover began to run properly! A welcome outcome, when the hardware is 140 million miles away from the engineers…

d) Assume now that there are four processes, *H* (high priority), *MH* (medium-high priority), *ML* (medium-low priority), and *L* (low priority), and two shared resources, *R*1 and *R*2. Also, assume that priority donation is *not* being used. Construct a scenario in which priority inversion can again occur. In your scenario, no process should hold more than one lock, although it may be blocked while waiting for another lock.

   Given this kind of scenario, what general approach must the system follow to ensure that the priority inversion issue cannot occur? What information must be recorded for each process to implement your approach, and how is this information utilized by the operating system?

   Clarification:

   Imagine that priority donation is a facility that can be enabled or disabled. Further, imagine the scenario in part d is playing out on a system in which priority donation is disabled. So then you need to spell out the details of how such a scenario could lead to priority inversion.

   Then, in the last half of the question, examine the scenario you define, and extrapolate details about how priority donation needs to work in the general case, as well as the details specified in the last part of the problem.

e) In a simple deadlock scenario, there are two processes *P*1 and *P*2, and two resources *R*1 and *R*2. Process *P*1 acquires a lock on *R*1, and *P*2 acquires a lock on *R*2. Then *P*1 requests a lock on *R*2 and *P*2 requests a lock on *R*1. Of course, now *P*1 and *P*2 are deadlocked.

   What issue could occur with the priority donation mechanism, in the context of such deadlocks? Describe how this issue could be resolved.

Virtual Memory Page-Replacement Policies (20 points)

Put answers to these questions in a file **`cs24fin/vmem.txt`**

The virtual memory manager (VMM) must regularly make an important decision – which page to evict from DRAM when a new page must be loaded. There are many different strategies to choose from. In this problem, you will get to explore two different strategies.

For this problem, assume that each physical page frame will be referenced by exactly one page-table entry; in other words, assume there is no shared memory. (Otherwise, it's too complicated.)

1. Recall that every page-table entry has two important bits: the Accessed bit and the Dirty bit. These bits are set by the MMU, but the kernel is responsible for clearing these bits. A very simple page replacement policy can be designed solely around these two bits.

In this policy, the kernel maintains a periodic timer interrupt that traverses all page-table entries, clearing the Accessed bit of each entry. (This interrupt must traverse all page-table entries across all process; it should not be limited to a specific process.) This means that page-table entries could fall into one of four different classes:

- Accessed and Dirty
- Accessed and Not Dirty
- Not Accessed and Dirty (e.g. it was written sometime before the most recent timer-tick)
- Not Accessed and Not Dirty

When a new page must be loaded from disk, the virtual memory manager must choose exactly one page to evict, from the pages that are currently in memory. Assume that the VMM will choose a page based on the above classifications: it will examine each class of pages in some order, and if there are pages in a given class, it will randomly choose a page from that class to evict.

In what order should the VMM consider these four classes? In other words, what classification should the VMM prefer to evict from first, and then second, and so forth? Explain your rationale for the ordering your choose, making sure to enumerate all reasons for the relative position of every classification in your ordering. *(5 points)*

2. Another replacement policy also relies on a periodic timer interrupt that traverses all page-table entries in all processes. The VMM maintains a *k*-bit *history* value for every active page (e.g. *k* = 32 on IA32). Every time the timer-interrupt triggers, each page's history value is shifted right by one bit, and then the topmost bit is set to the page-table entry's Accessed bit. Each page's Accessed bit is also cleared after its history value is updated.

   a) When the VMM must evict a page, describe how it should use this value to choose what page to evict. Make sure to explain the rationale behind your design. *(5 points)*

   b) Assume that the kernel's timer interrupt uses a function **`update_history()`** to update *history* values based on page-table entries, with a signature like this:

```
void update_history(pte_t pte, unsigned int *p_history)
```

   The **`pte`** value is an IA32 page-table entry. Recall that on IA32, bit 5 of page-table entries is the Accessed bit (see Lecture 24, slide 19, for details). The **`p_history`** argument is a pointer to the *history* value being updated; it is an in-out parameter.

   Provide an implementation of the **`update_history()`** function in C. You do not have to compile or test your function; just define it in your submission. (We will be lenient on minor syntax errors.) *(5 points)*

   c) Compare this policy to the Least-Recently Used (LRU) page-replacement policy. Recall that in the LRU policy, we always choose to evict the page that was accessed furthest in the past. How is this policy similar? How is it different? Explain your answers. *(5 points)*

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