CS 111: Operating System Principles

Final Exam Fall '21

Instructor: Jon Eyolfson

December 9, 2021

Duration: 2 hours 50 minutes

_____

Name

_____

Student ID

This is a closed book exam. You are only permitted a pencil or pen.
Answer the questions directly on the exam.

If in doubt, write your assumptions and answer the question as best you can.
There are 12 numbered pages (page 12 is blank if you need extra room).
The pace of the final is approximately one point a minute.
There are 150 total points.

Good luck!

Question 1. Page Replacement (30 points total)

Assume the following accesses to physical page numbers:

1, 2, 3, 2, 4, 2, 5, 4, 1, 2

You have 3 physical pages in memory. Assume that all pages are initially on disk.

1a. (20 points) Use the clock algorithm for page replacement. Recall on a page hit, you'll set the reference bit to 1. For each access write all the pages in memory *after the access* in the boxes below. State the number of page faults after all the accesses.

1	2	3	2	4	2	5	4	1	2
1	1	1	1	4	4	4	4	4	4
	2	2	2	2	2	2	2	1	1
		3	3	3	3	5	5	5	2

7 page faults.

1b. (10 points) Now, use the optimal algorithm for page replacement. All the other constraints are the same as the previous question. For each access write all the pages in memory *after the access* in the boxes below. State the number of page faults after all the accesses.

1	2	3	2	4	2	5	4	1	2
1	1	1	1	1	1	1	1	1	2
	2	2	2	2	2	5	5	5	5
		3	3	4	4	4	4	4	4

6 page faults.

Question 2. Threading (40 points total)

Jon writes an… interesting server that allows remote clients to run the `ls` command on it. I don't know why Jon did it, but he did. Maybe he was tired from running home to get his laptop. For the code example, assume no errors occur and all system calls are successful. As a reminder, the file descriptor returned from `accept` can be used in subsequent `read` and `write` system calls. Consider the following code:

```
void* run(void* arg) {
  int fd = (int) arg;
  sleep(5);
  pid_t pid = fork();
  if (pid == 0) {
    dup2(fd, 1);
    execlp("ls", "ls", NULL);
  }
  return NULL;
}

int main() {
  int socket_fd;
  /* Setup the socket (calls socket, bind, listen) */
  while (true) {
    int fd = accept(socket_fd, NULL, NULL);
    pthread_t thread;
    pthread_create(&thread, NULL, run, (void*) fd);
    pthread_detach(thread);
  }
  return 0;
}
```

Again, assume no errors occur. There's some weird casting, but at the end of the day, all we're doing is passing `fd` to a thread.

2a. (2 points) Assume 4 clients connect and each one makes it to the call to `sleep(5)` and none return from `sleep` yet. How many threads are there *in total* and what are they executing in the code?

There are 5 threads in total. There's 4 in `sleep` and the original main thread is in `accept`.

2b. (10 points) Now, all threads return from `sleep` and complete. What gets sent back over the socket, and why?

Because the file descriptor representing the socket connection gets dup2'ed in the child process to stdout before exec'ing `ls` will send it's output over the socket. So, each client will receieve the output of `ls` on this server machine.

2c. (3 points) How many processes did we create (excluding the original running process)?

We creted 4 processes (this question was mostly a hint that you'd also have an original thread in 2a).

2d. (10 points) What did we forget to do with the newly created porcesses? Why is this especially an issue in this case where our server runs for a very long time and may end up serving thousands of requests?

We didn't call `wait` on them and created a bunch of zombie processes. Since the server is a long running processes, these zombie processes will not also become orphans and get reparented. All the zombie processes will stick around and waste resources. Eventually, given enough requessts, we'll run out of resources and be unable to create new processes.

2e. (5 points) What would happen if we did not `fork` and instead always executed `dup2` followed by `execlp` directly after the `sleep` in every thread?

The first thread that reaches `exec` would transform the current running process (the server) into `ls`. `ls` would still send it's output over the socket to the client, but afterwards no other client would get a response (because the server process no longer exists).

2f. (10 points) What would be the issue if we removed `pthread_detach`?

If we removed `pthread_detach`, similar to creating processes, we would create zombie threads. These zombie threads would waste resources, and eventually we'll have the same issue as in 2d, we'll be unable to create new threads (at minimum, we may also run out of memory).

Question 3. Locks (20 points total)

You were given a `transfer` function to safely transfer funds between two accounts. You decided to refactor it such that there's a separate function for deducting (removing) funds from an account. Assume that multiple threads can call `transfer` simultaneously using different accounts. Consider your initial implementation:

```
struct account {
  pthread_mutex_t mutex;
  char* name;
  int amount;
}

void transfer(struct account* from, struct account* to, int amount) {
  if (from == to) return;
  pthread_mutex_lock(&to->mutex);
  to->amount += deduct(from, amount);
  pthread_mutex_unlock(&to->mutex);
}

/* Returns the amount of money deducted from an account.
   An account cannot go negative. */
int deduct(struct account* account, int amount) {
  pthread_mutex_lock(&account->mutex);
  if (account->amount >= amount) {
    account->amount -= amount;
  }
  else {
    amount = 0;
  }
  pthread_mutex_unlock(&account->mutex);
  return amount;
}
```

Again, assume no errors occur.

3a. (5 points) Does this code have any data races? Come up with an example of a data race if it does, or justify why it does not.

No, it does not have any data races. All accesses to an accounts `amount` are protected by using their single mutex. This includes reading and writing to their `amount` to either add or subtract funds from their account.

3b. (5 points) Does this code have any deadlocks? Come up with an example of a deadlock if it does, or justify why it does not.

Yes, the code can deadlock. Assume A transfers to B and B transfers to A (any amount). One thread (A → B) could acquire A's lock, then it gets context switched out and the other thread (B → A) acquires B's lock. There's now a circular wait and the program deadlocks when it tries to acquire the other lock in `deduct`.

3c. (10 points) Given the issue(s) you found in 3a and 3b, explain how you would fix the code to prevent those issues. You may not significantly refactor the given code. That means `transfer` must add to the `to` account and call `deduct`. Also, `deduct` must safely decrement the account only if it won't have an negative amount, otherwise it must be 0. `deduct` always returns the amount deducted. You may write replacement functions, or explain yourself clearly.

We can keep `deduct` the same, it has no data races if just `deduct` gets called from multiple threads. We can simpily eliminate "hold and wait" by doing the `deduct` call independently from adding the funds to the other account. A valid solution would be:

```
void transfer(struct account* from, struct account* to, int amount) {
  if (from == to) return;
  amount = deduct(from, amount);
  pthread_mutex_lock(&to->mutex);
  to->amount += amount;
  pthread_mutex_unlock(&to->mutex);
}
```

This is safe, because we're only adding funds to the other account. There's no data races, and anything that occurs simultaneously isn't in a defined order.

Question 4. Locking (20 points total)

You're given 4 threads (that get properly created and run) and you want to ensure ordering between them. You decide to use semaphores to accomplish your task. Recall that semaphores, after initialization, use `sem_post(sem_t* sem)` to increment the value and `sem_wait(sem_t* sem)` to decrement the value (waiting until the value is greater than 0). Assume no errors occur, so you never need to check return values. Consider the following code:

```
/* Global variables */ sem_t sem1; sem_t sem2; sem_t sem3;

void initialize() {
  sem_init(&sem1, 0, 0);
  sem_init(&sem2, 0, 0);
  sem_init(&sem3, 0, 0);
}
void* thread1(void*) {
  f1();
  sem_post(&sem1);
  sem_post(&sem1);
  f2();
  sem_post(&sem2);

}
void* thread2(void*) {
  sem_wait(&sem1);

  f3(); /* only runs after f1 completes */
  sem_post(&sem3);

}
void* thread3(void*) {
  sem_wait(&sem1);

  f4() /* only runs after f1 completes */;
  sem_post(&sem3);

}
void* thread4(void*) {
  sem_wait(&sem2);

  f5(); /* only runs after f2 completes */
  sem_wait(&sem3);
  sem_wait(&sem3);
  f6(); /* only runs after f3 and f4 complete */
}
```

4a. (10 points) Fill in the initial values for each semaphore and insert `sem_post` and `sem_wait` calls in a way that ensures the ordering in the comments. You cannot change the ordering of the `f()` calls, and they always execute in the order written. Write your answers on the previous page.

4b. (10 points) For each function state which functions *could* run in parallel with it (not including itself). Write "none" if it can only run by itself.

`f1`: none

`f2`: `f3`, `f4`

`f3`: `f2`, `f4`, `f5`

`f4`: `f2`, `f3`, `f5`

`f5`: `f3`, `f4`

`f6`: none

Question 5. Memory Allocation (10 points total)

You decide to use a buddy allocator to handle a bunch of fixed-sized allocations (it was too early and you slept through slab allocators). Your allocator needs to be able to handle 8 allocations of 10 bytes each.

5a. (5 points) How big of a memory block does your buddy allocator need at minimum to be able to handle every allocation? Justify your answer.

Your memory block needs to be at least 128 bytes. Since the buddy allocator only allocates blocks in powers of 2, each 10 byte allocation is going to get a 16 byte block. Therefore, we need 8 16 byte blocks, or 128 bytes.

5b. (5 points) How many bytes are lost due to fragmentation and what type of fragmentation is this? Justify your answer.

For each allocation there's 6 bytes lost to internal fragmentation. Since we have 8 allocations, there's a total of 48 bytes lost to internal fragmentation. If we used a single 128 byte block for our buddy allocator, there would no external fragmentation.

Question 6. Disks (10 points total)

Inspired by the course, and your data hoarding, you decide you'd like to build yourself a RAID. You buy yourself 8 hard drives that are 4 TB each.

6a. (2 points) What RAID configuration would you use if you really really care about your data? Justify your answer.

If we really really care about our data, we would use RAID 1. RAID 1 mirrors every single disk so they're exact copies of each other. As long as we have one disk working, we'll have all our data. However, we'd only have 4 TB available as usable storage and wouldn't have improved write performance. Read performance however, would be better.

6b. (3 points) What RAID configuration would you use if you care about your data but want more than 4 TB of total capacity? Justify your answer.

We would likely either pick RAID 5 or 6, depending on how many disks we'd like to tolerate losing at once. With RAID 5 we could tolerate a single failure, and RAID 6 we could tolerate two failures. For RAID 5 we'd have 28 TB of usable space, and for RAID 6 we'd have 24 TB of usable space. The performance of RAID 5 would be slightly better than RAID 6. Those are the major tradeoffs, both are good choices.

6c. (5 points) What RAID configuration would you use if you only care about performance? Justify your answer. Also state how much faster your read and write performance would be compared to a single disk.

We'd use RAID 0, and as a benefit we'd get to use all 32 TB of storage as well. Since we're dividing all the data among all 8 disks without any parity or backup, we'd get 8 times the read and write performance compared to a single disk.

Question 7. Filesystems (10 points total)

You'll be using your knowledge of filesystems to explain the difference between `ln` and `cp` in terms of inodes and blocks. Assume you have a file called `my-file.txt` that contains 2290 bytes. The block size of the filesystem is 4096 bytes, so it fits on a single block. As a hint, I ran the `ln` and `cp` commands and you'll find the output of `ls` after the commands below:

```
> ln my-file.txt ln-file.txt
> cp my-file.txt cp-file.txt
> ls
inode   size   name
  101   2290   my-file.txt
  102   2290   cp-file.txt
  101   2290   ln-file.txt
```

7a. (5 points) Explain what happens on the filesystem when you run the `ln` command. Start your explaination with the new directory entry for `ln-file.txt`.

Within the current directory, we'd make a new entry for `ln-file.txt` and it would simply be assigned the same inode as `my-file.txt`. On the inode itself we would increment the link count by one (not needed for this question). The blocks themselves would be unchanged.

7b. (5 points) Explain what happens on the filesystem when you run the `cp` command. Start your explaination with the new directory entry for `cp-file.txt`.

We would create a new entry for `cp-file.txt` in the current directory and create a new inode (it would previously be free). We'd then copy all the blocks in `my-file.txt` to free blocks in the filesystem. In the `cp-file.txt` inode we'd point to these newly copied blocks instead. You could also argue that we could also use a copy-on-write strategy to avoid copying the blocks until we absolutely need to. However, we'd still have a new inode.

Question 8. Virtual Machines (10 points total)

You have a single server that hosts a web server, database, and streams video. Over time, as your server gets used more, you find that it's now overloaded. You decide to buy another server. You could move one of the three applications to the new server, and hope that it's balanced. Given you've taken this course, you think using virtual machines would be a good idea (assume containers do not exist).

8a. (5 points) How would your organize your applications into virtual machines? Explain how many virtual machines you would make and what would go on them.

We would just make a virtual machine for each application. So, we'd have one for a web server, one for a database, and one for streaming video. In the beginning we could still run all of these virtual machines on a single machine if it's not overloaded. Now that we have an overloaded machine, we could just move the virtual machine that's consuming the most resources to the new server. (This question is about how you would organize the virtual machines to applications).

8b. (5 points) What would be the benefits of using your virtual machines in the event one of the two servers is now overloaded?

We could again, just move the virtual machine that's overloaded to the new server. We could even do a live migration and transfer the virtual machine while it's running so there's no interruption whatsoever. Neat!