CIS 3800 Spring 2024: Final Exam

May 7, 2024

First Name : ______________________________________

Last Name : ______________________________________

Penn ID : ____

Please fill in your information above, read the following pledge, and sign in the space below:

I neither cheated myself nor helped anyone cheat on this exam. All answers on this exam are my own. Violation of this pledge can result in a failing grade.

Sign Here : ______________________________________

Exam Details & Instructions:

- There are 8 questions made of 14 parts (and a short bonus) worth a total of 100 points.
- You have 120 minutes to complete this exam.
- The exam is closed book. This includes textbooks, phones, laptops, wearable devices, other electronics, and any notes outside of what is mentioned below.
- You are allowed two 8.5 x 11 inch sheets of paper (double sided) for notes.
- Any electronic or noise-making devices you do have should be turned off and put away.
- Remove all hats, headphones, and watches.
- **Your explanations should be more than just stating a topic name. Don't just say something like (for example) "because of threads" or just state some facts like "threads are parallel and lightweight processes". State how the topic(s) relate to the exam problem and answer the question being asked.**

Advice:

- Remember that there are 8 questions made up of a total of 14 parts (and a short bonus question). Please budget your time so you can get to every question.
- Do not be alarmed if there seems to be more space than needed for an answer, we try to include a lot of space just in case it is needed.
- Try to relax and take a deep breath. Remember that we also want you to learn from this.

Please put your PennID at the top of each page in case the pages become separated.

If you need extra space, the last page of this exam is blank for you as scratch space and to write answers. If you use it, please clearly indicate on that page and under the corresponding question prompt that you are using the extra page to answer that question. Please also write your full name and PennID at the top of the sheet.

Question 1 {9 pts}

One of the things we learned in this class is that different data storage has different access times.

Consider the following cases:

1	We tried to look up a virtual page number translation in the TLB, but the translation is not present. So, we have to go to the multi-level page table, and we find the translation in there.
2	We access data that is currently stored in a register to do some computation.
3	We try to look up data that is stored in the cache and the data is present in the cache, so we fetch it and use it.

List each of those cases from fastest to slowest. Briefly justify your answer. (you should not need much more than 1 sentence each for each case).

You probably don't need this full space, but we have provided it just in case.

Question 2 {12 pts}

One of the big advantages to using a FAT is that we keep some data in memory instead of storing it in disk, thus we do not have to go to disk as often.

This works fine if we read data, but when that data is modified, it is always immediately written back to disk. Adam supposes that this is rather inefficient and proposes that when we update the FAT in memory, we do not always write it back to disk, instead we update the FAT in disk every 10 times that the FAT in memory is modified (or when the user specifically requests that it is written to disk).

Part 1 {5 pts}

Would this actually make the code that modifies the FAT run faster? Please briefly justify your answer.

You probably don't need this full space, but we have provided it just in case.

Part 2 {7 pts}

Adam's proposal is something that is very rarely done. There is a good reason why if the FAT is updated in memory we update it on disk as well. Please explain what that reason is and an example of something that could go wrong if we were to implement this change.

You probably don't need this full space, but we have provided it just in case.

Question 3 {5 pts}

One of the notable ways we interact with various operating system features is through items we would consider "handles" or "descriptors". For example, when we open a file, we are given a file descriptor, which is just an integer value. Notably we are not given access to any "real" file data structure directly.

Fork() does a similar thing by returning a process ID instead of direct access to a process control block. A similar thing happens when we create a pthread, the "returned" pthread_t is usually nothing more than an integer that identifies the thread.

Why do you think system calls are designed in this way? Designed to give us these identifiers and don't give us direct access to their underlying structure. There are more than one reason, but please give us only one (we will only grade the first reason you list).

You probably don't need this full space, but we have provided it just in case.

Question 4 {15 pts}

One of the most common data structures in computer sciences is a map structure.
Note: you do not need to be familiar with maps, or algorithm analysis to solve this problem. **The maps are just the setting for the question.**

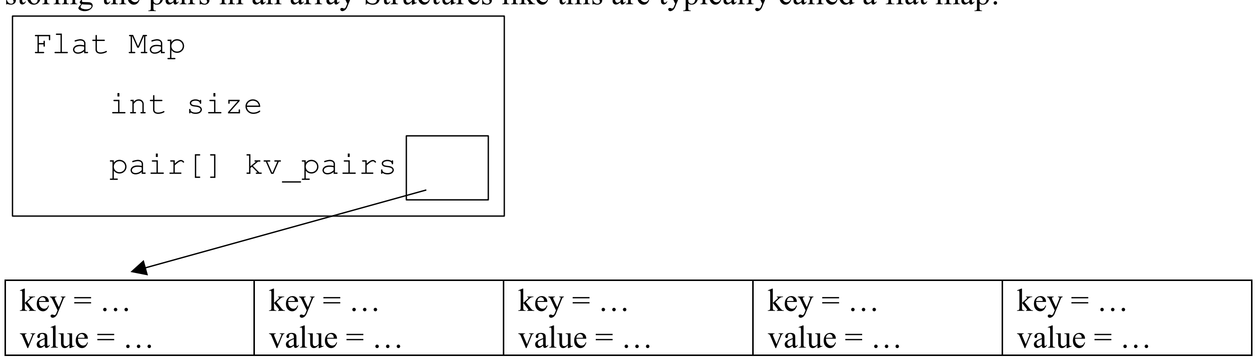
What is a map?

I believe you should be familiar with what a map is from taking the pre-requisite course, but I've included a brief refresher on a map here. Feel free to skip to the memory diagrams if you think you are already familiar. You do not need to be familiar with hashing to answer this question.

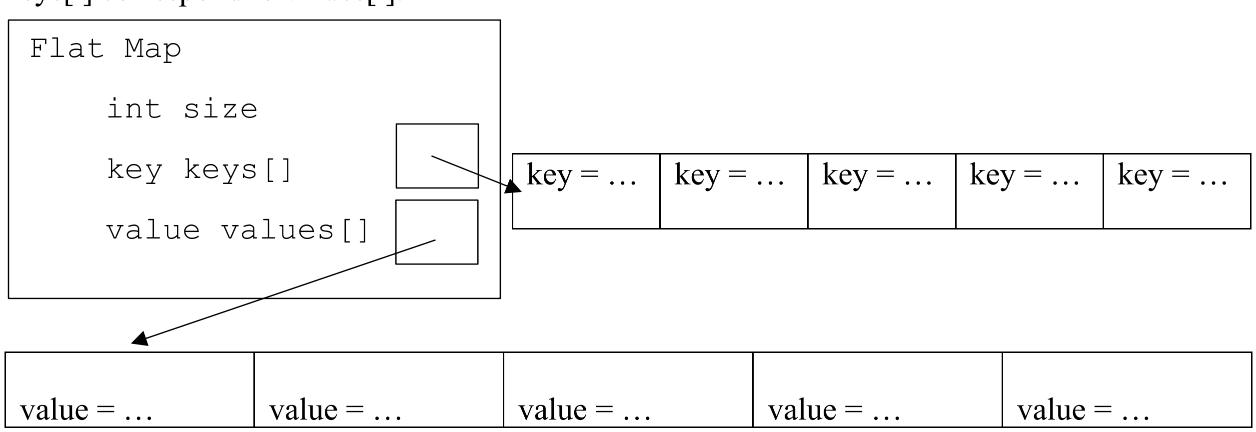
A map is a data structure that has two associated types, a key type and a value type. Users can store keys to be associated with a value. A Map is thus a collection of key-value pairs. Common operations include adding a new pairing, setting an existing pairing to have a new value, finding a specific pair from just the key, and iterating over all elements in the map.

Memory Diagrams

One way we can store key-value pairs is by storing each key with is value as a pair and then storing the pairs in an array Structures like this are typically called a flat map:



A modification we can make on the flat map is that instead of one array, we have two arrays. One array has the keys and another has the values. The key and values are associated by index, so that keys[i] correspond to values[i].



Part 1 {10 pts}

Let's say we write code that has a huge map containing many elements. The map uses 4-byte integers as keys and 4-byte floats as values (8 bytes together). We analyze our code and notice that by far the most common operation performed on this structure is the "get" operation:

```
// given a map and a key, looks through the map for the
// presences of the key and returns the corresponding value
value get(map m, key k);
```

If we wanted to maximize performance for that operation, which structure would be better? Why? **Your answer should be 2-3 sentences.**

Hint: if you are thinking about algorithm analysis like O(n) stuff or counting the number instructions executed, you are doing it wrong

Part 2 {5 pts}

If the keys and value pairs were larger (let's say that the keys are 64 bit integers (8 bytes) and that the value is the size of a page, 4096 bytes). How does this change your answer from part 1? Please explain why. Limit your answers to 3 sentences at maximum. Note: We will try and treat your answer to part 1 as "correct" for the sake of this question.

Question 5 {15 pts}

We usually uses `pipe()` and think about file descriptors in the context of using `fork()` to create a new process. However, we are not restricted to using a `pipe()` with `fork()`. Jerry decides to experiment using `pipe()` in combination with threads.

Jerry starts by looking at a program that compiles and works as expected but uses processes and `fork()`. (This example code is optional, mostly here to refresh you on pipe and processes incase you are a bit rusty)

```
void* child_code(void* fds) {
   // cast the void* arg to access it as an int array
   int* pipe_fds = (int*) fds;

   close(pipe_fds[1]);  // close write end
   char buf[1024];
   ssize_t res = read(pipe_fds[0], buf, 1024);
   // readS up to 1024
   // does not repeatedly read till EOF, reads ONCE

   write(STDOUT_FILENO, buf, res);
   // write what we read to stdout
   return NULL;
}

int main() {

  int pipe_fds[2];
  pipe(pipe_fds);

  pid_t pid = fork();

  if (pid == 0) {
    child_code(pipe_fds);
    exit(EXIT_SUCCESS); // child process exits
  }

  close(pipe_fds[0]); // close read end
  write(pipe_fds[1], "Hi!\n", 4);

  waitpid(pid, NULL, 0);
  return EXIT_SUCCESS;
}
```

Part 1 {7 pts}

Jerry starts by rewriting `main()` to use threads instead of processes. When he does, he gets this:

```
int main() {
  int pipe_fds[2];
  pipe(pipe_fds);

  pthread_t thd;
  pthread_create(&thd, NULL, child_code, pipe_fds);

  close(pipe_fds[0]); // close read end
  write(pipe_fds[1], "Hi!\n", 4);

  pthread_join(thd, NULL);
  return EXIT_SUCCESS;
}
```

This code compiles without errors, but it does not do the expected behaviour the initial program did. (The parent should write “Hi!\n” to the pipe, the child reads and prints, and everything exits gracefully). Why does this code not do what is expected? You can assume that pthread_create, pthread_join, and pipe() do not error and that they are called with the proper arguments.

Hint: remember that there are two threads here (main and the one created by pthread_create) that share the same process. :)

Part 2 {8 pts}

What could Jerry do to make the code he wrote have the expected behavior? Please be specific about what lines need to be **added**, **modified** **or** **deleted**. Your changes must stay in the spirit of the problem (e.g. still use threads and the pipe with each thread interacting with the pipe).

Hint: This is accomplishable with only the functions and lines used in the previous snippets.

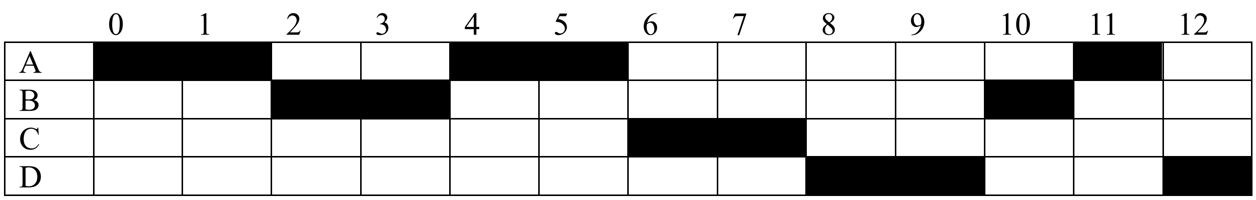
Note: You probably don't need this full space, but we have provided it just in case. (and so that the next question starts on a new page)

Question 6 {15 pts}

Suppose we have a scheduling using round robin with a time quantum of 2. Assume our machine is a single processor/core machine. If we have processes described in the table below

Process Name	Arrival Time	Job Length
A	0	5
B	1	3
C	3	2
D	4	3

Then the processes will be scheduled like this:



In this algorithm, if there are multiple processes to add to the "ready queue" at the same time, assume they are put into the queue in this order:

1. any arriving processes are put into the queue first
2. any process that just finished its time slice is put into the queue last

Part 1 {10 pts}

If we were to instead schedule them with a round robin time quantum of 3, what would the scheduling look like? Please fill in the diagram below

	0	1	2	3	4	5	6	7	8	9	10	11	12
A													
B													
C													
D													

You may use the blank space here as scratch space. Part 2 begins on the next page.

Part 2 {5 pts}

If we increase the quantum size, then round robin starts approaching behaviour more similar to First Come First Serve (FCFS). What is one way this may be **good**?

Question 7 {16 pts}

Suppose we have a new data structure called a `super_vector` that is thread safe to up to two threads. In other words, if more than two threads are accessing the `super_vector`, it is unsafe. If only one or two threads are accessing the `super_vector`, it is safe.

To help make sure accesses to this data structure is safe, Zhiyan creates a version of a lock that is built using `pthread_mutex_t` (similar to how we implemented a rw_lock in lecture).

Our lock structure ends up looking like this:

```
typedef struct two_mutex_st {
  pthread_mutex_t meta_lock;
  int num_holding_threads;
} two_mutex;

// initializes a two_mutex
void two_mutex_init(two_mutex* m) {
  pthread_mutex_init(&m->meta_lock, NULL);
  m->num_holding_threads = 0;
}
```

This problem continues onto the next page.

The lock and unlock functions look like this:

```
// acquire the specified two_mutex
// does not return from the function until the mutex is
// successfully acquired.
void two_mutex_lock(two_mutex* m) {
  pthread_mutex_lock(&m->meta_lock);
  while (m->num_holding_threads >= 2) {
    pthread_mutex_unlock(&m->meta_lock);
    // gap
    pthread_mutex_lock(&m->meta_lock);
  }
  m->num_holding_threads += 1;
  pthread_mutex_unlock(&m->meta_lock);
}

// release the two_mutex
void two_mutex_unlock(two_mutex* m) {
  pthread_mutex_lock(&m->meta_lock);

  m->num_holding_threads -= 1;

  pthread_mutex_unlock(&m->meta_lock);
}
```

So, whenever a thread wants to use a `super_vector`, it runs code that looks something like:

```
super_vector sv;
two_mutex sv_lock;

void* thread_fn(void* arg) {
  // assume sv_lock is initialized at the beginning
  // of the program with `two_mutex_init(&sv_lock);`

  two_mutex_lock(&sv_lock);

  // access the super vector here

  two_mutex_unlock(&sv_lock);
}
```

Part 1 {8 pts}

Assuming that the program is well formed (it compiles, the `sv_lock` is properly initialized, each thread acquires the `sv_lock` before accessing the super vector and release appropriately, etc.) does the `sv_lock` actually work?

In other words, does our `two_mutex` and functions ensure that only two threads can "lock" or "acquire" the `two_mutex` at a time? Does it do this without deadlocking? Briefly justify your answer.

Note: You probably don't need this full space, but we have provided it just in case.

Part 2 {8 pts}

For this question, assume that the code written worked properly without deadlocking.

Zhiyan notices that when a thread returns from `two_mutex_lock()` that we aren't actually holding the `meta_mutex` anymore. She proposes that we remove the `meta_lock` from our code since it may be unnecessary.

If we were to remove the `meta_lock` (and the associated calls to `pthread_mutex_lock` and unlock around it), would our lock still work in a safe manner? If the `meta_lock` is necessary, please explain why it is needed.

Please justify your answer.

Note: You probably don't need this full space, but we have provided it just in case.

Question 8 {12 pts}

One of the allocators we mentioned in class is the slab allocator, which is fast but restricted to allocations of a fixed size.

Seungmin has the idea for a new allocator to replace the C standard library `malloc()`. Seungmin builds his allocator out of several slab allocators that each have a different allocation size of various powers of 2 (8, 16, 32, 64, 128, 512, 1024, 4096, ...).

When someone wants to allocate memory, he has a slab allocator handle it, and uses the smallest slab allocator that will still satisfy the request. For example, if someone allocates 48 bytes, he goes to the 64-byte slab allocator and uses that to perform the allocation.

Part 1 {6 pts}

What is a possible upside to using this allocation scheme? Please briefly explain your answer

PennID: ______________________

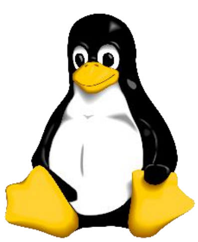
Part 2 {6 pts}

How does both the internal and external fragmentation look for this scheme? Please be clear about what fragmentation is internal and which is external. Briefly justify your answer.

Question 9 {1 pt} <u>all submissions will get this point</u>

Tux the penguin (see below) is the mascot for Linux! Design a mascot for PennOS 😊

If you don't want to do that, then put anything here! What's your favourite thing you learned in this course? Anything you want to show us or want us to know?



Appendix

Waitpid man page

SYNOPSIS

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Description

This system call is used to wait for state changes in a child of the calling process and obtains information about the child whose state has changed.

If a child has already changed state, then these calls return immediately. Otherwise, they block until either a child changes state or a signal handler interrupts the call.

pipe man page

SYNOPSIS

```
int pipe(int pipefd[2]);
```

DESCRIPTION

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe.

dup2 man page

SYNOPSIS

```
int dup2(int oldfd, int newfd);
```

DESCRIPTION

The dup2() system call creates a copy of the file descriptor oldfd, using the file descriptor number specified in newfd. If the file descriptor newfd was previously open, it is silently closed before being reused.

pthread_create

SYNOPSIS

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

DESCRIPTION

The pthread_create() function starts a new thread in the calling process. The new thread starts execution by invoking start_routine(); arg is passed as the sole argument of start_routine().

pthread_join

SYNOPSIS

```
int pthread_join(pthread_t thread, void **retval);
```

DESCRIPTION

The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately.

If retval is not NULL, then pthread_join() copies the return value of the target thread into the location pointed to by retval.

pthread_mutex_lock

SYNOPSIS

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

DESCRIPTION

The mutex object referenced by mutex shall be locked by calling pthread_mutex_lock(). If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

pthread_mutex_unlock

SYNOPSIS

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

The pthread_mutex_unlock() function shall release the mutex object referenced by mutex.

This page is intentionally left blank.