CIS 3800 Fall 2023: Midterm 0

Oct 19, 2023

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 7 questions made of 12 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 one 8.5 x 11 inch sheet 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 7 questions made up of a total of 12 parts (and a short bonus 8th 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. A bad grade on this exam is not the end of the world. This grade also can be overwritten by a better grade with the Midterm “Clobber” Policy (details in the course syllabus)

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.

PennID: _____

Question 1 {24 pts}

One of the nice things about processes (and threads but this question is about processes!) is that we can run processes in parallel across multiple CPUs, allowing for the operation to be completed faster.

We want to take advantage of this parallelism in our code by forking some number of processes and splitting up the work across each of the processes.

For each of the problems we will identify a problem we want to solve and how we would split it across multiple processes.

We will also list some system calls related to processes, and you will have to tell us roughly how many times we would need that system call to complete the desired task. You can either choose either 0 or 1 or N, each are explained here:

- 0: to indicate that the system call is not needed
- 1: to indicate that the system call only needs to be invoked once or a constant number of times (roughly the same number of invocations regardless of number of processes)
- N: to indicate that the system call would need to be invoked a number of times proportional to N, where N is the number of processes we intend to use.

You will have to briefly (3 sentences max) justify each answer that is not 0

Note: when deciding what functionality is needed, focus on what is strictly needed for the task to be completed. Do not worry about functionality that is not described.

Note: we are asking how many times the system call gets invoked when the program runs. Not how many times it shows up in your code file.

You may use this blank space for scratch work. The questions begin on the next page.

PennID: _____

Part 1 {12 pts}

We have a large array of integers stored in program memory, but we want to find the sum of all numbers in the array.

We fork N processes, and have each process calculate the sum for a different part of the array. We would make sure that the whole array is divided evenly between processes and there is no overlap.

The parent process could then get the results from all processes, add them together and get the final result.

System call	0 / 1 / N
fork()	**N**
waitpid()	**N**
pipe()	**N**
execvp()	**0**
kill()	**0**
signal()	**0**

For each system call you gave a non-zero answer to, please briefly (3 sentences max) justify why it is needed and why that many times.

Reminder: we have an appendix at the end of the exam containing part of the man page for most of these functions

Note: This is just one possible answer

Fork: must fork N times to create N processes.

waitpid: must call waitpid N times so that we can reap the zombie children, know when they are done, and thus, that they have sent us their result.

Pipe: must call pipe N times so that we can get the results from each child that we forked. We must use pipe as a form of inter process communication.

Execvp: there is no reason to call execvp, the child can run their own code to calculate the sum of its portion of the array.

Kill: there is no signal processing required for this question

Signal: there is no signal processing required for this question

PennID: _____

Part 2 {12 pts}

We have a list of files, and we want to convert every letter in every file to lower case.

We fork N processes, and evenly split the files across the processes, so that each process will handle different files. Each process would then go through its assigned files and set every uppercase letter to lowercase.

System call	0 / 1 / N
fork()	**N**
waitpid()	**N**
pipe()	**0**
execvp()	**0**
kill()	**0**
signal()	**0**

For each system call you gave a non-zero answer to, please briefly (3 sentences max) justify why it is needed and why that many times.

Reminder: we have an appendix at the end of the exam containing part of the man page for most of these functions

Note: This is just one possible answer. I included answers for why 0 was chosen, but this was not required.

Fork: must fork N times to create N processes.

waitpid: must call waitpid N times so that we can reap the zombie children, know when they are done, and thus, that they have sent us their result.

Pipe: there is no need to send data to/from users The results of each processes operations is just modifying the files and each process will have a copy of the parent's address space, and thus a copy of the list of files to modify.

Execvp: there is no reason to call execvp, the child can run their own code to read/write the desired files.

Kill: there is no signal processing required for this question

Signal: there is no signal processing required for this question

PennID: _____

Question 2 {9 pts}

Consider the following block of code that waits for a child process to complete/terminate:

```
pid_t pid = …  ; // assume initialized to the pid of the
                 // process we want to wait for
int status;

pid_t ret = waitpid(pid, &status, WNOHANG);
if (ret == -1) {
   perror("waitpid errored");
   exit(EXIT_FAILURE);
}

while(ret == 0 && !WIFSIGNALED(status) && !WIFEXITED(status)) {
  ret = waitpid(pid, &status, WNOHANG);
  if (ret == -1) {
     perror("waitpid errored");
     exit(EXIT_FAILURE);
  }
}
```

We could also have the same code but modify **both** of our calls to waitpid to instead be:

```
waitpid(pid, &status, 0);
```

One of these implementations is considered better than the other. Which example and why?
Limit your answer to 3 sentences at maximum.

Note: you should be familiar with waitpid, but we've included part of the man page for it in the appendix if you need it.

Remember what we say on the front page about requirements for an explanation.

The implementation that uses `waitpid(pid, &status, 0);` would be preferred. This is because when those arguments are passed to `waitpid`, it will block the calling thread till there is an update in the status of a child process.

If `WNOHANG` was passed in instead, the calling thread would not block and instead loop repeatedly calling `waitpid`, which would consume CPU cycles without achieving anything (e.g., the thread would "busy wait").

Question 3 {16 pts}

Consider that we were working with a 32-bit system with byte addressability and a page size of 4096 (2^{12}) bytes. You can assume that we have 1 GiB (2^{30}) bytes of physical memory.

- 32-bit system means that each virtual address is 32 bits. (4 bytes)
- Byte Addressability means each byte in memory has a unique address.

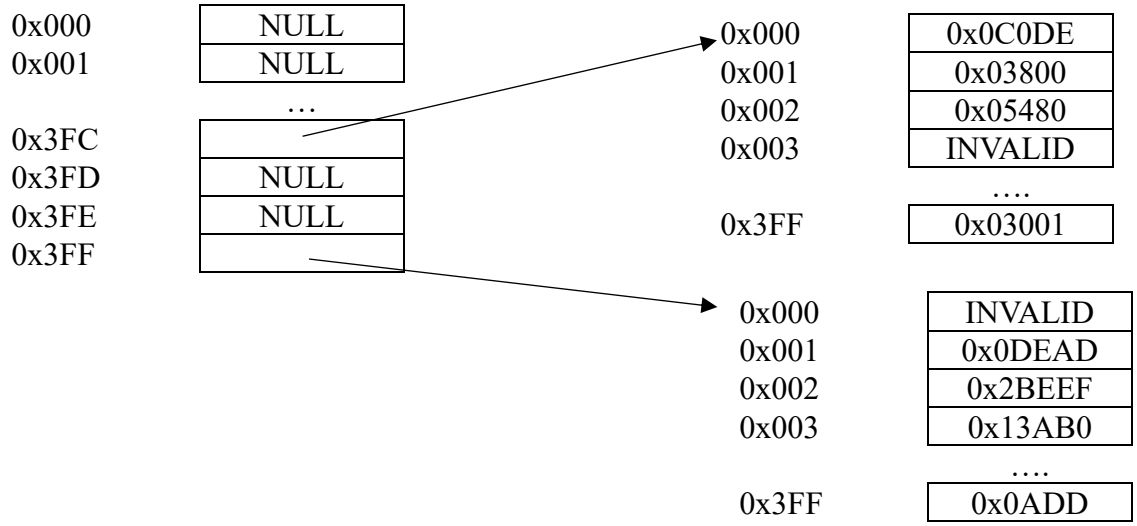
Part 1{4 pts}

On such a system, how many bits are needed for each of the following? No explanation is needed, feel free to leave some answers in terms of powers of 2 if that would be easier.

Item	Number of Bits Needed
Virtual Page Number	20
Page Offset	12
Physical Page Number	18
Physical Address	30

Part 2 {6 pts}

In this 32-bit system, if we were to use a multi-level page table, we could have a two-level table with 10-bits to index into each level of the table. Consider the following page diagram of what the multi-level page table looks like:



QUESTION CONTINUES ONTO THE NEXT PAGE

Part 2 continued

Using the page table described and displayed on the previous page, what are the indexes into each level of the table and the physical address translation for the virtual address 0xFFC03001. If the conversion cannot be completed since the page we want to access is not in physical memory, state that there is a page fault for the physical address. We've provided a hexadecimal-to-binary conversion table and rewritten the address in binary for your convenience.

Hex	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

```
Hexadecimal: 0xFFC03001

Binary: 1111 1111 1100 0000 0011 0000 0000 0001
        |_____10 bits_____|_____10 bits_____|
```

	Value (answers in this column)
First Level Index	**0x3FF**
Second Level Index	**0x003**
Physical Address	**0x13AB0001**

Part 3 {6 pts}

In a 64-bit machine (with the same addressability and page size) we would have instead used 9-bits to index into each level in our page table. 10 bits for 32-bit machines and 9 bits 64-bit machines are carefully chosen for those architectures, what is significant about using those numbers for these machines? Explain in 2-3 sentences at maximum.

Hint: this has to do with the size of something.

This answer is longer than 3 sentences to try and make the explanation as clear as possible

On both machines, a page is 4096 bytes, it would be convenient to store each node of the page table inside a singular page, making memory access of the page table itself more elegant.

In a 32-bit machine, each pointer (and each page table entry) is 32-bits, which is 4 bytes. This means we can fit 2^{10} pointers/entries into a single block. ($4 * 2^{10} = 2^{12}$).

64-bit machines have 64-bit pointers, which are 8-bytes. 2^{12} (4096) bytes per page divided by 2^3 (8) bytes per pointer gets us 2^9 pointers per page.

PennID: _____

Question 4 {6 pts}

Consider a new page replacement policy called MRU (Most Recently Used). MRU can be thought of as the "opposite" of LRU. Where the page that was most recently used is evicted from physical memory if another page needs to be brought in. (e.g., if page A was most recently accessed, and we then wanted to load in a page not in the physical memory, page A would be evicted to make space for the new page)

Assume we have virtual pages A B C D, start with empty physical memory and physical memory can only hold 2 physical pages.

What is a sequence of page accesses that would result in MRU having less page faults than LRU? **Please give a sequence of accesses that is exactly 6 page accesses long in the boxes below.**

This is just one possible solution; other answers were accepted.

Access #	0	1	2	3	4	5
Page accessed	**A**	**B**	**C**	**D**	**A**	**B**

Question 5 {8 pts}

The buddy allocator has an issue with internal fragmentation, forcing allocations to allocate a number of pages that is a power of 2. As a result of this, if I wanted to allocate many items that are of size 1024, I would waste 3072 bytes per allocation (assuming pages are 4096 bytes).

Instead, we could instead make a big allocation from buddy and use that allocation as a slab for the slab allocator.

How is this different from just allocating some space directly from buddy? What happened to all the memory that would have been internal fragmentation from a buddy algorithm allocation? Please explain your answer briefly in at most 3 sentences.

The slab allocator takes the block of memory allocated by the buddy algorithm and, instead of using it for a single 1024-byte object, can instead split the block into multiple objects. The 3072-bytes that would have been internal fragmentation for the allocation in the question is now used for other allocations of other objects

PennID: _____

Question 6 {18 pts}

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

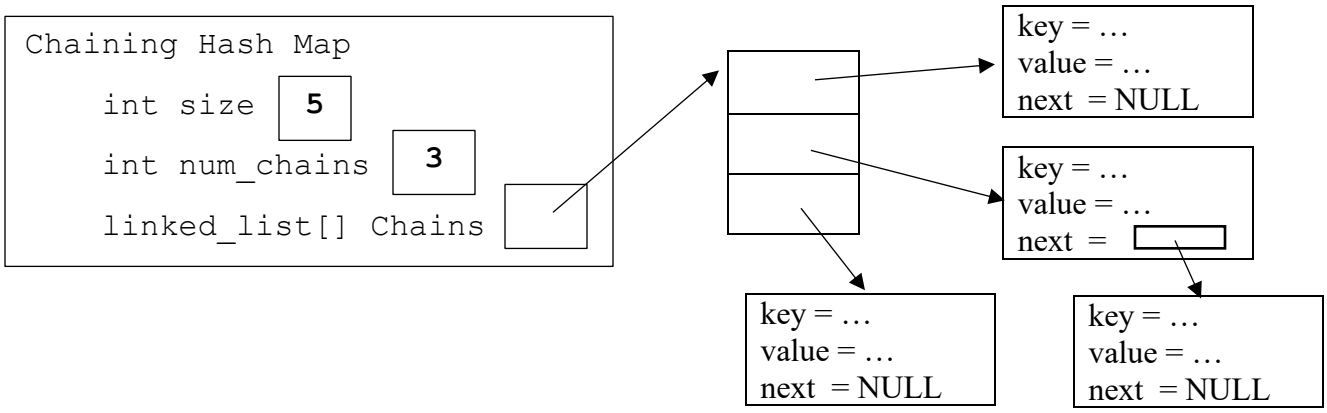
What is a map?

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.

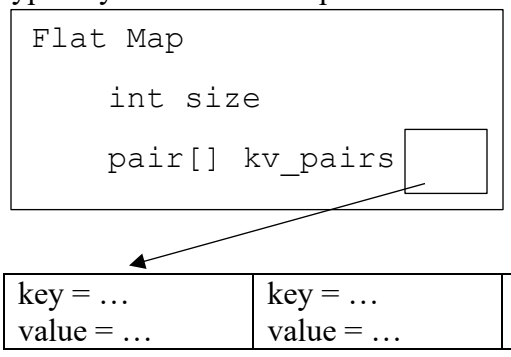
A map is a data structure that has two associated types, a key type and a value type. Users can store keys to be associated with a value. A Map is thus a collection of key-value pairs. Common operations include adding a new pairing, setting an existing pairing to have a new value, finding a specific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

Memory Diagrams

One common way to store key-value pairs is to use a chaining hash map. In memory, we can think of it as being represented like this:



Another way key-value pairs can be stored is by storing them in an array. Structures like this are typically called a flat map:



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 to iterate through all the key value pairs in the structure. 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

The flat map would be faster in this case.

Since the key-value pairs are 8 bytes and in contigious memory, then the cache would allow for much faster lookup speed. If we were to assume that a cache line is 64 bytes, then when we access a key-value pair, the next 7 key-value pairs will also be loaded into the cache and be much quicker to access than going to memory.

The chaining hash map cannot assume that the key-value pairs are next to each other in memory and would not get the same speedup.

Part 2 {8 pts}

If the key value pairs were large (let's say that they are the size of a page, 4096 bytes) we don't get the same performance boost we got before when iterating over the entries and the two map implementations seem much more comparable at run-time.

Why might this be the case? Please explain why. Limit your answers to 3 sentences at maximum.

If the key value pairs are 4096 bytes, then when we load a key-value pair in memory, we would only load that key-value pair into the cache. The next key-value pairs would not be loaded into the cache since they would not fit into the cache-line of the accessed pair, so the flat map would not get the same benefit from the cache.

Question 7 {18 pts}

Part 1 {8 pts}

This question will refer to some of the same situations encountered in question 1, but that question does not have to be answered correctly (or even at all) to answer this question.

Let's say we are trying to solve the problem in question 1 part 1 with threads. Namely:

> We have a large array of integers stored in program memory, but we want to find the sum of all numbers in the array.

Instead of forking N processes, we could instead create N threads and split the work up across threads. Then the parent thread could get the result from all the child threads and calculate the final sum.

Threads are considered "lightweight" processes and thus switching between threads would be faster than switching between processes. Even if we ignore speed to context switch, using threads would allow for a faster running program that solves this problem. Why is this the case? Please justify your answer. Your answer should be 3 sentences at maximum.

One possible answer:

Since threads share memory, threads can share their answers with each other by storing them at a specific point in memory. If we used processes, then we would have to share answers through the file system which takes a lot longer to do.

Part 2 {10 pts}

Note: it is okay if you are a bit rusty on your hw assignments, you should be able to answer this question even if those assignments did not go well.

Forking processes works pretty well for some problems. One example would be programs like penn-shredder or penn-shell. In those assignments, you wrote some code that looks like this:

```
// this is pseudo code
// feel free to raise your hand if you don't understand what
// it means. You don't need the exact details though.
while (!eof) {
  input = getline(stdin);
  cmd = parse_cmd(input);

  for each program in cmd {
     pid = fork();
     if (pid == 0) {
       // child
       execvp(program);
     }
  }

  for each pid we forked {
     waitpid(pid);
  }
}
```

We may be tempted to do something like this with threads, but it is rather difficult, if not impossible, to support this to work for every legal command one could input into the shell.

What is **one** thing that could go wrong if we tried to implement penn-shell and/or penn-shredder by creating threads instead of fork processes? Please answer in at most 3 sentences.

Reminder: we have some man pages in the appendix at the back of the exam

Clarification: it is not just about taking the code and replacing fork with pthread_create. If we wanted to recreate the functionality of penn_shell with threads and could change the program however needed, there would be at least one issue that we cannot fix without calling `fork()`. What is one of those issues?

One possible answer:

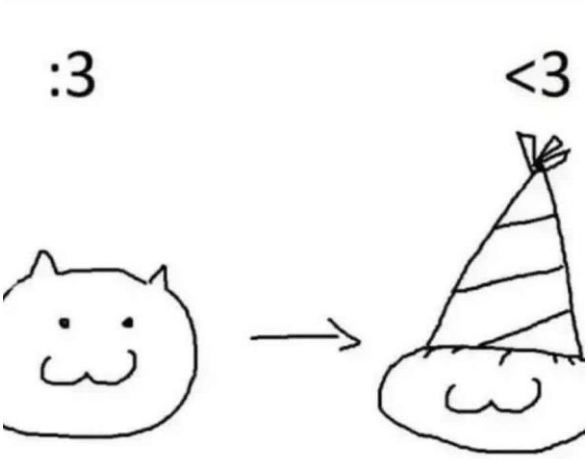
In penn-shell, we could execvp any arbitrary program the user wants. Execvp does not work with threads though, if we were to call execvp in a thread, it would replace the entire running process with a process running specified function . The other threads would be affected by this and the shell would not be able to run after calling execvp.

PennID: _____

Question 8 {1 pt} <u>all submissions will get this point</u>

Select one member of the course staff. Create a piece of art (e.g. drawing, poem, anything you like) about that person.

If you don't want to do that, then put anything here! What's your favourite thing about C programming? 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.

The value of options is an OR of zero or more of the following constants:

WNOHANG

return immediately if no child has exited.

WUNTRACED

also return if a child has stopped.

If wstatus is not NULL, waitpid() stores status information in the int to which it points. This integer can be inspected with the following macros

WIFEXITED(wstatus)

returns true if the child terminated normally, that is, by calling exit() or by returning from main().

WIFSIGNALED(wstatus)

returns true if the child process was terminated by a signal

RETURN VALUE

on success, returns the process ID of the child whose state has changed; if WNOHANG was specified and one or more child(ren) specified by pid exist, but have not yet changed state, then 0 is returned. On error, -1 is returned.

execvp man page

SYNOPSIS

```
int execvp(const char *file, char *const argv[]);
```

DESCIRPTION

replaces the current process image with a new process image. This causes the program that is currently being run by the calling process to be replaced with a new program specified by the argument file and, that program will have the arguments specified by argv. The process will have a newly initialized stack, heap, and data segments.

RETURN VALUE

does not return on success, and the text, initialized data, uninitialized data (bss), and stack of the calling process are overwritten according to the contents of the newly.

Returns -1 on error

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.

kill man page

SYNOPSIS

```
int kill(pid_t pid, int sig);
```

DESCRIPTION

The kill() system call can be used to send any signal to any process group or process. In normal usage, signal sig is sent to the process with the ID specified by pid.

signal man page

SYNOPSIS

```
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

DESCRIPTION

signal() sets the disposition of the signal signum to handler, which is either SIG_IGN, SIG_DFL, or the address of a programmer-defined function (a "signal handler").

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.

This page is intentionally left blank.