CIS 3800 Fall 2023: Midterm 1

Dec 7, 2023

First Name : ______**SAMPLE**______________________________

Last Name : ______**Solution**______________________________

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 13 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 13 parts (and a short bonus 9th 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.

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

Please keep all answers in the designated boxes.

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 {8 pts}

One thing we hoped you learned in this course was how the CPU can be pre-empted at almost any point to stop running code/instructions in the continuous sequence we are used to. For example, if we have the code:

```
int x = 0;
x += 3;
x *= 2;
```

Then after we execute `x += 3`, the processor may start executing a line of code that is not the next line of code: `x *= 2;`.

This is true even if we have a single threaded process and only 1 CPU core. What is something that would make it so that after executing the line `x += 3;`, the next line of code run by our CPU in the same process is something other than `x *=2;` but still within the same process?

Briefly (~3 sentences max) explain your answer in the box below

One case for this is that a signal handler interrupts the normal flow of our program and thus we we start executing the code in our signal handler.

Another explanation (not expected or required, we would only look at the first explanation you give, but since this is the sample solution we will include it)

It could be due to out of order execution done by the processor or the compiler. The processor may run code in a different order than we write because it thinks it may result in a performance benefit, and may be doing it here. The processor is allowed to do this as long as the code runs *as if* it were the code compiled.

Question 2 {15 pts}

When we print in a program via `printf()` or `System.out.println()`, many students often thought that this means "send this data to the terminal output immediately". After this course, we know now that these print operations are not so straightforward.

Part 1{9 pts}

When we call `printf("hello");` that data will probably not go directly to the terminal output (or whatever destination is intended). Instead, this data gets buffered somewhere in the standard library and will get sent to its destination eventually. Why does the C standard library (and most other programming language's standard library) do this?

This is done to minimize the number of times we go to through the file system and invoke the write system call. Doing the write system call and going to the file system takes time, so we would rather write a bunch of stuff at once to minimize this cost.

Part 2{6 pts}

Ignoring the fact that data we want to print may not go to the terminal immediately, it may not go to the terminal at all and instead end up elsewhere while still being a successful operation (e.g. it was not an error to go somewhere else)! How can this happen?

We may have redirected the output so that it goes to a pipe or to a file!

Question 3 {18 pts}

A cool optimization that is done in C++ is called Short String Optimization (SSO).

In C++ there is a string object type. We can think of the string object as pretty much being the following struct equivalent in C:

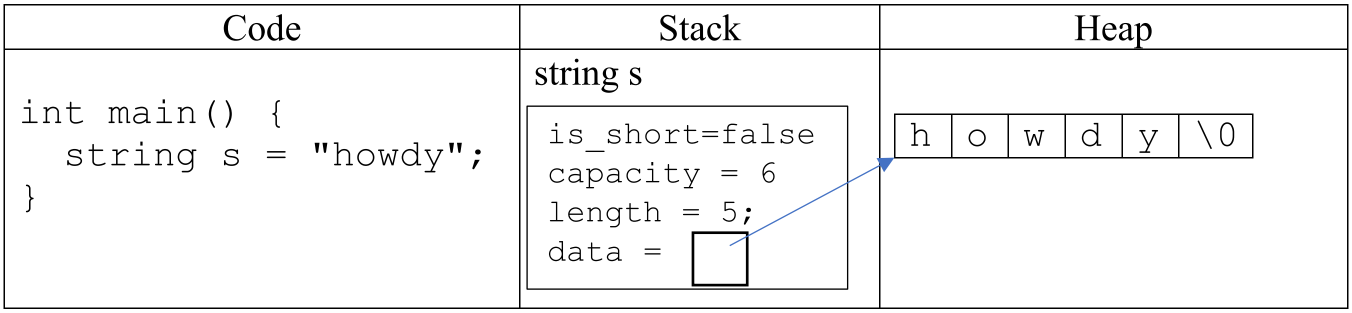
```
typedef struct string_st {
  bool is_short; // used to determine if we are using
                 // the optimization

  int capacity;  // how long the string can be before
                 // we would have to resize (and thus
                 // re-allocate the data)

  int length;    // the current length of the string

  char* data;    // a pointer to the characters the
                 // string represents, on the heap
} string;
```

So, for the following code, we can reason about the string having the associated memory layout:



Code	Stack	Heap

`int main() {`
`  string s = "howdy";`
`}`
	string s
	`is_short=false`
	`capacity = 6`
	`length = 5;`
	`data = □`
		`h` `o` `w` `d` `y` `\0`

For short string optimization, some people realized that if the string is short, we could reinterpret the memory inside the struct to store the characters in the struct directly (instead of being a pointer to the heap). This means we could instead represent the string with the following struct layout:

Continued onto the next page.

```
typedef struct string_st {
  bool is_short; // used to determine if we are using
                 // the optimization

  int length;    // the current length of the string

  char data[12]; // the characters in the string
} string;
```

For a short enough string (like “howdy”) we now have the following memory layout, but longer strings would follow the first layout (the struct that contains a pointer to the heap).

Code	Stack	Heap

`int main() {`
`  string s = "howdy";`
`}`
	string s
	`is_short=false`
	`length = 5;`
	`data =`
	h, o, w, d, y, \0, …
		// nothing!

This code is space efficient, but arguably more importantly, it makes programs using short enough strings to run faster.

Part 1{9 pts}

One case this optimization helps is the case where we have an array of contiguous string objects that we want to do some operation on (such as printing all the strings in the array). How does the short string optimization allow this case to run faster?

Since the string data is stored in the string struct itself and the structs are in contiguous memory, then all the string data is also contiguous with other strings. This helps cache performance and there are less cache misses from having to go to the heap. If string data was on the heap then different strings likely have their data in different parts of the heap and so we don't get the same memory locality.

Part 2{9 pts}

One way this makes strings faster is when creating a new string object. Consider the following C pseudocode that creates a string "object" by copying the data from a passed in `char*`.

```
// takes in a "C string" and creates a string "object"
string create_string(char* raw_data) {
  if (strlen(raw_data) > 11) {
    long_string result;
    result.is_short = false;
    result.length = strlen(raw_data);
    result.capacity = result.length + 1;
    result.data = malloc(strlen(data) + 1);

    // copies the raw characters into the allocated string.
    strcpy(result.data, raw_data);
    return result;
  } else {
    short_string result;
    result.is_short = true;
    result.length = strlen(raw_data);

    // copies the raw characters into our array.
    strcpy(result.data, raw_data);
    return result;
  }
}
```

You may have noticed that the code for each of the two cases are very similar. Despite this similarity, the short string allocation is much quicker than the long string allocation. **Why is short string allocation faster? Please justify your answer.**

(*Note*: it is not just that there are less lines of code in the short string case in this function. We also expect this to be at least slightly different than your answer to part 1).

Please put your answer to part 2 on the next page.

Part 2 continued {9 pts}

Correct: Since we don't have to call malloc, short string optimization is faster. Malloc has overhead to managing the heap that we don't have from just storing thing in a statically sized struct

Question 4 {8 pts}

When we context switch to a thread inside of a different process, it is a more costly operation than switching to another thread within the same process. In some high-performance situations, a CPU may be configured to only run threads within the same process to avoid this cost. Why is it more costly for run time if a CPU switches between threads that are in different processes?

When you switch between threads in different processes you need to load in a new memory space which costs time. Staying in the same address space doesn't have this cost. It also clears the cache and possibly part of the TLB, which could also affect performance.

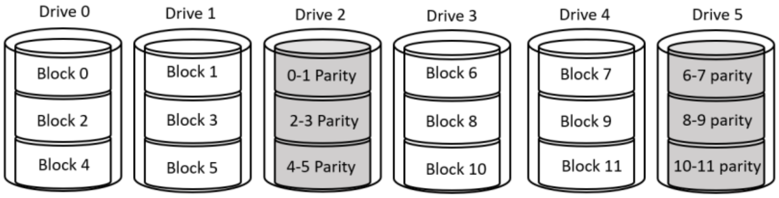
Question 5 {8 pts}

Linked List Allocation via FAT and I-nodes would load some amount of meta data about the file system into memory (by either keeping I-nodes in memory or the FAT in memory). Why do we want to store this information in memory when that same information is a duplicate of what is already stored in the disk drive we are managing?

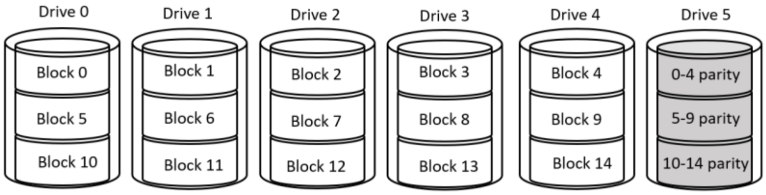
We want to cache the inodes and FAT in memory so that we don't have to access disk as much. Accessing disk is much more expensive than accessing memory.

Question 6 {10 pts}

Consider this RAID level 4 configuration with 6 disk drives. We have a 2:1 data parity ratio so that the first two disks use the third disk as parity, and the 4th and 5th disk use the 6th disk as parity.



Suppose we change this configuration so that instead of having two parity disks, we instead have the first 5 disks dedicated to storing data and leave only 1 disk to store the parity for those 5 other disks.



Continued onto the next page

Part 1{4 pts}

What is one reason why this change is an improvement on the system? Pleas justify your answer

Two possible answers:

1. **There is more data that can be stored since now less blocks are dedicated to parity**
2. **Higher read parallelism possible after the change since there are more drives holding data, goes from 4 to 5 possible parallel reads**

Part 2 {6 pts}

How does this change affect the system's tolerance of disk failures? Please justify your answer. You can provide an example if you wish.

Some things that would not be faults before are now faults. Consider the case where disk 1 and disk 4 fail. In the first configuration, we could recover, now we can't.

Question 7 {8 pts}

Let's say you are working on a parallel algorithm and have gotten a good amount of the algorithm to run in parallel. A non-negligible amount of the algorithm is not done in parallel and is instead done sequentially (pretend 20% is sequential, the exact number is not relevant to the problem), but most of it is done in parallel.

A friend looks at your code and makes the claim that running the algorithm with 2 threads means it should finish in 1/2 of the time it takes to run with one thread. Is this the case? Please explain why.
Note: Your answer should ignore the overhead time it takes to create and destroy threads. You can also assume that your computer has at least 2 CPUs that can be dedicated to running your code.

No, this is not a 2x speedup since part of it cannot be done in parallel. If it were fully parallel only then would there be a 2x speedup (and thus done in half the time).

Could optionally show the Amdahl's calculation:

Amdahl's law gives speedup = $\frac{1}{(1-P)+\frac{P}{2}}$, where P is the percentage of the algorithm we can parallelize. If P=0.8, the speedup is $\frac{1}{0.2+0.4} \approx 1.667$.

It is only when P=1 that the speedup will be 2; if P<1, speedup <2.

Question 8 {24 pts}

Consider the case where we have N numbered blocks of data and some number of threads that may want to do some operation on those shared blocks. Only one thread can access a block at a time, so to support this situation we do something like this:

```
void operate_on_blocks(list<int> block_numbers) {
  sort(&block_numbers);
  for (every block_num in block_numbers) {
    acquire_lock(block_num)
  }

  // I have all the blocks I requested.
  // now I can do some operation on the blocks
  do_operation(block_numbers);

  for (every block_num in block_numbers) {
    release_lock(block_num);
  }
}
```

The call to sort the list of block numbers is to ensure that locks are acquired in a strict ordering across all threads, and guarantees that no deadlock is possible.

Assume that each thread calls this function to access the shared blocks and that no thread tries to acquire the same block twice.

Part 1 {8 pts}

Some of these blocks are only used for part of the operation. Because of this, someone suggests modifying the code so that during a transaction a thread delays acquiring the lock for a specific block until "absolutely necessary". In this example, we will say that it is "absolutely necessary" we acquire the lock for a block with number “B”:

- right before we use that block B.
- If we need to acquire the lock for some block that has a number greater than B and we currently do not have block B locked, we must acquire the lock to block B first. (This is to make sure we still acquire the locks in a strict ordering).

Part 1 continues onto the next page

Example: let's say a thread wants to operate on blocks 1, 2 and 5. It starts the operation and notes that the operation starts by accessing block 1, so the thread acquires the lock for block 1. The thread continues the operation and notices it needs block 5, so it acquires the lock for block 2 and then the lock for block 5. Later in the operation it needs block 2, but it already has the lock acquired for that block, so it doesn't have to acquire any new lock. Once the thread is done, it can release the locks to all blocks it used.

Part 1 continued {8 pts}

With this modification, is deadlock possible? Please briefly explain why. If you think deadlock is possible, please also provide an example as to how it can happen.

There is no deadlock since the order of lock acquisition is still maintained

Part 2 {8 pts}

Instead of the change proposed in part 1 of this question, someone else proposes that we instead have the thread release the lock to a shared resource as soon as the thread has no more use for it (as opposed to waiting till the end of the operation). Is deadlock possible in this code? Please briefly explain why. If you think deadlock is possible, please also provide an example.

A deadlock cannot happen since the order of acquiring locks is still enforced. When we release a lock generally has no possibility of causing a deadlock

Part 3 {8 pts}

In both adjustments to the algorithm described in part 1 and part 2, the goal is to minimize the amount of time a thread spends holding a lock. This is a beneficial optimization (assuming that no deadlocks occur), but why is this an optimization? Why would we want to minimize the amount of time a thread holds a lock?

Please put your answer to part 3 on the next page

Part 3 continued {N pts}

We want to minimize the amount of time threads hold a lock since that prevents other threads from running code that needs the lock, thus lessening how parallelizable the code is

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

Now that you are an operating systems expert, you've got to show off your knowledge to your friends, loved ones or just random people you are meeting. Write a pick-up line related to operating systems that shows off your knowledge of the field 😊

If you don't want to do that, then put anything here! What's the most important thing you learned this semester? It doesn't have to be from this or any course.

Can't share the good pickup lines, they get a little spicy 😋

This page is intentionally left black for scratch space.

This page is intentionally left black for scratch space.