CIT 5950 Spring 2024: Midterm

Feb 28, 2024

First Name : ______Bjarne______________________________

Last Name : ______Sutter_______________________________

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 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 {5 pts}

Travis has written a short C++ program that deals with two dimensional vectors of integers. We call these two-dimensional vectors a "matrix". Here is what the main() function of this program looks like:

```
int main() {
  vector<vector<int>> matrix {
    {3, 4, 5},
    {6, 3, 2},
    {1, 2, 7}
  };

  mult_matrix(matrix, 3);
  print_matrix(matrix);
}
```

The functions `mult_matrix()` and `print_matrix()` are written by Travis. `mult_matrix()` intends to modify the input matrix so that each number in it is multiplied by the specified number. `print_matrix()` simply prints the matrix using `cout`.

The program compiles and runs, but the program outputs the wrong thing. We would expect to get:

```
9  12 15
18 9  6
3  6  21
```

However we actually get this as output:

```
3 4 5
6 3 2
1 2 7
```

After some debugging, we narrow down the issue to be in the mult_matrix function.

Problem continues onto the next page

PennID: ______________________

Line num	Code
0	`void mult_matrix(vector<vector<int>> matrix, int n) {`
1	`  for (size_t row_num = 0U; row_num < matrix.size(); row_num++) {`
2	`    vector<int> row = matrix.at(row_num);`
3	`    for (size_t index = 0U; index < row.size(); index++) {`
4	`      row.at(index) *= n;`
5	`    }`
6	`  }`
7	`}`

There are some bugs in this code, and it is your job to fix it. Please list all the lines you would like to change and how you would change them so that the function behaves as expected.

If you want to insert a new line, you may specify by saying something like "I would insert a new line between line 1 and line 2".

Please keep your answer in the box below

On line 0 and on line 2, we should be using a reference to the vector.

For line 0, we pass in a vector<vector<int>> by reference, otherwise the changes we make to its contents will be discarded when we return from the function because we would modify a copy of the matrix.

Similarly on line 2, vector<int> row should be declared as a reference otherwise we make a copy of the row in the matrix and thus changes to the row variable will not affect any data in the matrix.

PennID: ______________________

Question 2 {24 pts}

Part 1 {12 pts}

We want to write the function `count_chars()` that takes in a string as input and returns a map that maps characters to integers. The resulting map contains each character in the string associated with the count of how many times that character shows up in the string.

The following code calls the `count_chars()` function

```
int main() {
  auto res = count_chars("please");
  for (auto& pair : res) {
    cout << pair.first << ": " << pair.second << endl;
  }
}
```

When run, it should print out:

```
a: 1
e: 2
l: 1
p: 1
s: 1
```

It is your job to implement the function count_chars in the box on the next page.

Note that you do not need to worry about the specific ordering of the printed output. map<> will automatically sort the entries when they are stored in the map.

Note: You can use any C++ that was covered in class.

You may use this blank space for scratch work. Your answer goes on the next page.

PennID: ______________________

Note: we do not expect people to use all the space provided, but we are providing it just in case it is needed.

```
map<char, int> count_chars(const string& input) {
  // this is one possible solution
  // others are possible

  map<char, int> result;
  for (auto c : input) {
    result[c] += 1;
  }
  return result;
}
```

Part 2 {12 pts}

We want to write another function `remove_duplicates()` that takes in a vector of strings and returns a new vector that has all elements of the input, but any duplicate entries are removed. If a string shows up more than once in the input vector, we only keep the first instance of it in the output vector.

Given the following main that calls `remove_duplicates()`:

```
int main() {
  vector<string> v {"I", "am", "so", "tired", "I", "i", "am"};
  auto result = remove_duplicates(v);

  for (auto& str : result) {
    cout << str << endl;
  }
}
```

When run, we should get the following printed:

```
I
am
so
tired
i
```

It is your job to implement the function `remove_duplicates()` in the box on the next page.

Note: You can use any C++ that was covered in class.

You may use this blank space for scratch work. Your answer goes on the next page.

Note: we do not expect people to use all the space provided, but we are providing it just in case it is needed.

```
vector<string> remove_duplicates(const vector<string>&
                                                  input) {
  // this is two possible solutions
  // others are possible

  vector<string> result;
  set<string> seen;
  for (const auto& s : input) {
    if (!seen.contains(s)) {
      result.push_back(s);
      seen.insert(s);
    }
  }
  return result;

  /* alternative solution without a set
  for (const auto& s : input) {
    bool found = false;
    for (const auto& str : result) {
      if (str == s) {
        found = true;
        break;
      }
    }

    if (!found) {
      result.push_back(s);
    }
  }

  return result;
  */
}
```

Question 3 {12 pts}

Part 1 {6 pts}

We have a function that is called by a thread. That thread modifies a global integer variable that is shared across threads. As is good practice, we acquire a lock before modifying the variable and release the lock afterwards.

```
pthread_mutex_lock(&count_lock);
count += 1;
pthread_mutex_unlock(&count_lock);
```

Without seeing any other part of our multi-threaded code, do we know that modifying count will always be safe? That there is no data race on count? You can assume that the code compiles, is called by threads, and the variables `count` and `count_lock` are initialized properly.

Please justify your answer

No we do not. Just because this one section of code is properly using a lock does not mean others are. If another piece of code is modifying the variable `count` without acquiring the same count lock, and another thread runs that code, then a data race on count is still possible.

Part 2 {6 pts}

Suppose there is a `const int` global variable that is accessed by many threads frequently across the execution of a program. An example declaration of the variable is below:

const int MAGIC_NUMBER = 13;

You can assume that the entire program (which is not shown) compiles, threads are created properly and that const is never violated.

Without seeing anymore of the program that utilizes the `const int` global variable, what can we determine about the safety of accessing the const int global variable? Is a data race on our `const int` global variable possible?

Please Justify your answer.

A data race can only happen when one or more threads access a shared resource (global variable in our case) and at least one of them is modifying that resource. Since const is respected in this case, that means no thread is modifying the global variable, they are all instead just reading it, so no data race is possible.

Question 4 {20 pts}

Consider the following situation where we have a global linked list of doubles that is very big and shared across threads in a program.

```
struct node {
  node* next;
  double data;
};

node* global_head = nullptr; // list starts "empty"
```

At some point in the program, we want to initialize this list to contain 1,000,000 random doubles. To do this we generate a random double and then call the following function to push onto the front of the global linked list.

```
void push_global_list(double value) {
  node* new_node = new node();
  new_node->next = global_head;
  new_node->data = value;
  global_head = new_node;
}
```

Note: This question is not asking you to implement a linked list or verify the correctness of this function. You can assume the code above compiles and correctly pushes onto the front of the global linked list (at least without considering threads).

This operation is taking a long time to compute, so we explore three different ways to implement this behaviour. Assume for these examples we have multiple processors.

1. We don't create any threads, and instead just have a single for loop that generates 1,000,000 random numbers and pushes them all onto the list.

2. We create 10 threads and have each thread create and push 1/10 of the data we want to eventually store on the list. The first thread calculates creates and pushes 100,00 doubles, The second thread generates and pushes the next 100,000 doubles etc.

3. We do the same as method #2, but we have a mutex on the overall global list. Each thread needs to acquire the mutex of the overall list before it can do generate the number or push onto the list, and releases the mutex when the thread is completely done.

Part 1 {8 pts}

For each of the three implementations described above, list whether or not it "works". By "work", we mean that the program doesn't have any data races and does not have any deadlocks.

Briefly justify your answer for each implementation, a sentence or two should be sufficient. **Please put your answer in the box on the next page.**

Part 1 {8 pts}

See the previous page for the question.

The first one works and there are no data races or deadlocks since there are no threads or mutexes involved.

The second one may not work since if multiple threads access the and try to modify the list at the same time it may not be a straightforward list. One node may try to create a new "head" by creating a node and setting next to the current head, but before making it the new head, another thread does an insertion. Then in this case the node that is inserted is skipped over. The list is no longer just a linear sequence of nodes. Like drawn below:

```
+------------+      +-----------+
| new head   |----> | old head |
+------------+      +-----------+
                            ^
     +-----------------+    |
     | "skipped" node  |----+
     +-----------------+
```

The third one works, Each thread only acquires the lock ones and releases when completely done. There is no possibility for data race or deadlock unless there is a significant programmer error.

Part 2 {12 pts}

Of the three implementations, rank them from fastest to slowest. **<u>Please justify your answer.</u>**

For this question, assume that any faulty implementations end up working correctly, even if it is possible for there to be an error through a data-race. This question is asking purely about speed.

The second one moves the fastest followed by the first and then the third.
The second one moves fastest since it takes advantage of parallel threads.
The first and third one both essentially do their work sequentially, but the first one avoids any overhead of acquiring a lock, creating threads and joining threads.

Question 5 {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, or algorithm analysis to solve this problem. **This problem is about memory, the maps are just the setting for the question.**

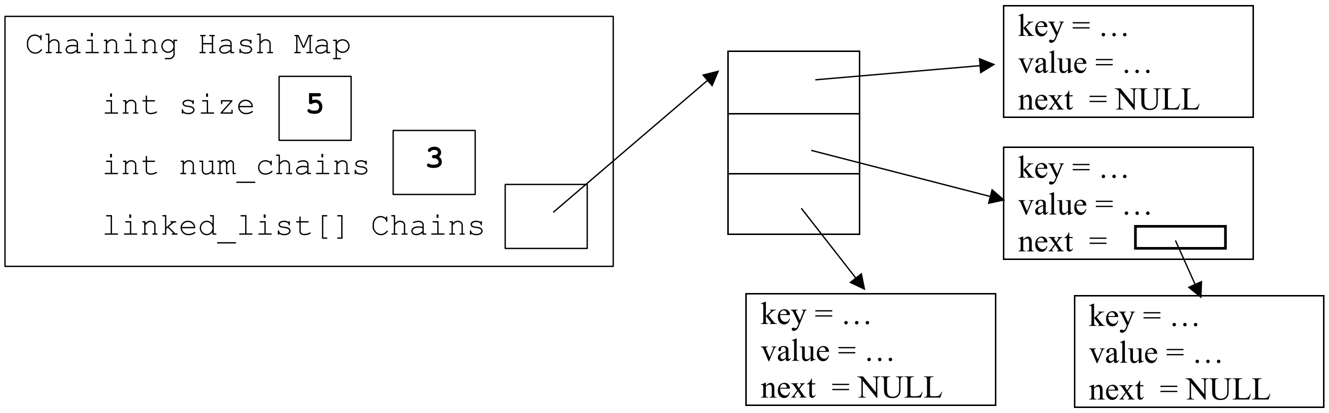
What is a map?

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.

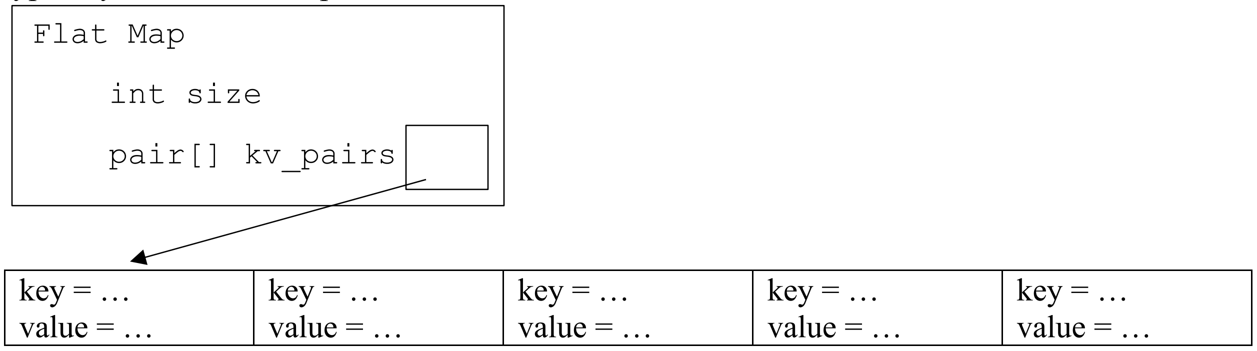
A map is a data structure that has two associated types, a key type and a value type. Users can store keys to be associated with a value. A Map is thus a collection of key-value pairs. Common operations include adding a new pairing, setting an existing pairing to have a new value, finding a specific 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 is faster for this operation due to the key-value pairs being in contiguous memory and thus this benefits from caching. When one KV pair is read from memory, the next few pairs are also pulled into the cache thus making them faster to access than having to go to memory again. The chained hash map does not get as much of this benefit since it's pairs are not stored contiguously .

Part 2 {8 pts}

If the key value pairs were large (let's say that they are 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.

Since the key-value pairs are so big, each access will be a cache miss and we won't get the performance benefit of caching.

PennID: ______________________

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:

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

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											🟥	🟥	🟥

PennID: ______________________

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**?

As it approaches FCFS, it will context switch less often when switching between threads. Thus less time is spent switching threads and more time is spent just running the threads.

Question 7 {5 pts}

In HW1 BufferedFileReader, the reason we maintained a buffer was so that we could "minimize calls to POSIX read()". What characteristic of calling POSIX read() made it so that we wanted to minimize the amount of times we called that function? Be specific.

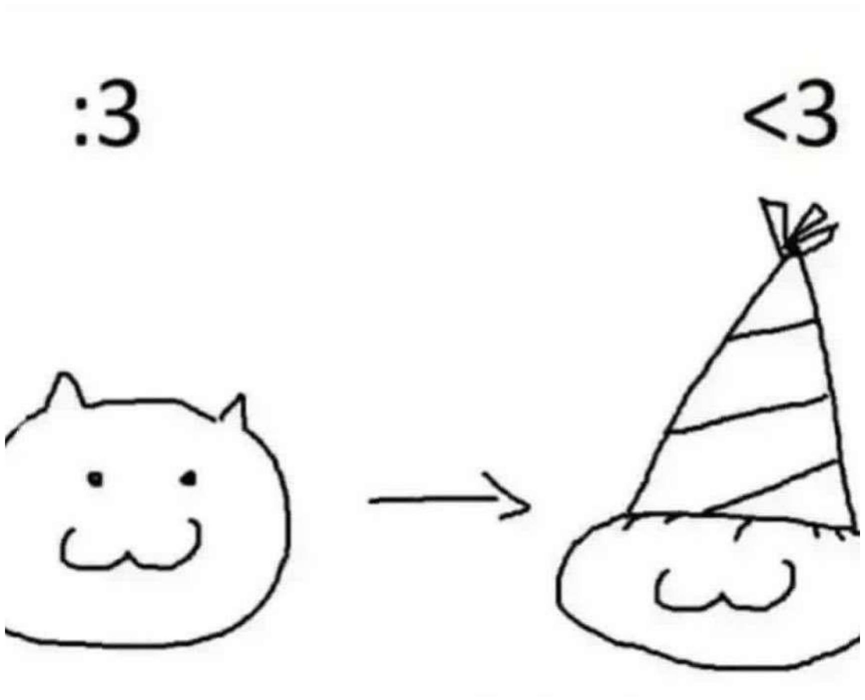
POSIX read is unbuffered on its own, so then each call to the read() function would have to go to the operating system and file system. Going to the operating system and file system is slow, so instead we try to keep data in memory. So if someone reads one character, we instead read enough to fill our buffer (1024 bytes) so that when we read the next character we read it from memory instead of the file system. This minimizes the number of times we go to the file system and saves us time.

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

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.

iterator find(iterator begin, iterator end, T target);

Given a range of values specified by the begin and end iterators, searches with the range starting at begin and ending at (but not including) end for the specified target. If the target value is found, then it returns an iterator to that element. If it is not found, then end is returned.

void vector<T>::push_back(const T& value);

Member function for the vector class. Called on a vector to push the specified value onto the end of the vector, thus extending it to be 1 element larger.

T& vector<T>::at(size_t index);

Member of the vector class to access an element at the specified index of the vector.

V& map<K, V>::operator[](const K& key);

Member of the map class to access the value associated with the specified key. If the key-value pair does not exist already in the map, then it is implicitly inserted using the default value for the value and returning a reference to the value.

bool map<K, V>::contains(const K& key);

Member of the map class to see if the specified key exists in the map. Returns true if it does, false if it does not.

This page is intentionally left blank.