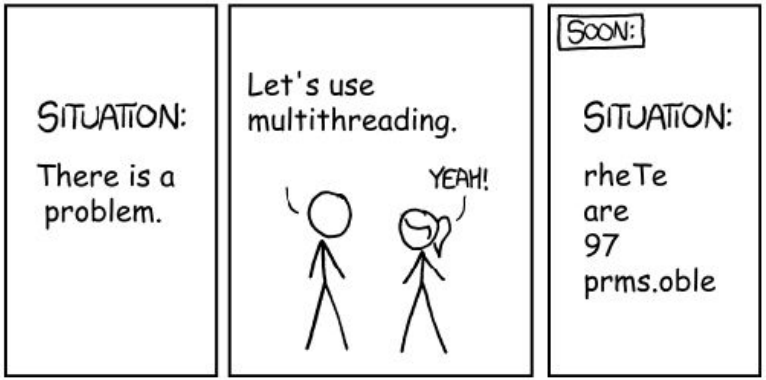
CSE 333 Section 10

Concurrency/HW Wrap-up



UNIVERSITY of WASHINGTON

Logistics

- **HW4**
  - Due **Thursday** @ 11:59 pm (Tonight!!)
  - You can use still use late days
- **Course Evaluations** - Please fill these out!
  - Closes **Sunday** (3/13) @ 11:59 pm
- **Final Exam**
  - Opens **Sunday** (3/13) @ 12:00 am
  - Closes **Wednesday** (3/16) @ 11:59 pm

Concurrency

Concurrency

- **Concurrency:** Managing and progressing multiple tasks over the same period of time.
  - Able to switch between different tasks when it wants to
  - Particularly helpful for functions that often get blocked (Networking, File I/O)
  - Different to **Parallelism**: Concurrency processes only one task at a time whereas parallelism will execute multiple tasks at the same time.

- **Mediums** for Concurrency?
  - Multiple **Threads**
  - Multiple **Processes**

Exercise 1

Threads vs Processes

	Multiple Threads	Multiple Processes
Memory / Address Space	Shared / Separate	Shared / Separate
Stack	Shared / Separate	Shared / Separate
Heap	Shared / Separate	Shared / Separate
Communication	Easy / Difficult	Easy / Difficult
Synchronization	Easy / Difficult / Not Applicable	Easy / Difficult / Not Applicable
Context-switch “Weight”	Light / Heavy	Light / Heavy
Crash Tolerance (“what happens to the others when one dies?”)	Tolerant / Not Tolerant	Tolerant / Not Tolerant

Threads vs Processes

	Multiple Threads	Multiple Processes
Memory / Address Space	Shared / Separate	Shared / Separate
Stack	Shared / Separate	Shared / Separate
Heap	Shared / Separate	Shared / Separate
Communication	Easy / Difficult	Easy / Difficult
Synchronization	Easy / Difficult / Not Applicable	Easy / Difficult / Not Applicable
Context-switch "Weight"	Light / Heavy	Light / Heavy
Crash Tolerance ("what happens to the others when one dies?")	Tolerant / Not Tolerant	Tolerant / Not Tolerant

Think about overhead and switching between them

Threads vs Processes

- **Multiple Processes:**
  - Created through `fork()`
  - Each process works independently from one another with their own address space.
  - Has additional overhead to switch between processes

- **Multiple Threads:**
  - Multiple executions on the same address space and tend to share resources with one another.
  - Easier to switch between different threads
  - All threads fail when one thread fails.

Concurrency with `pthreads`

Concurrency with `pthreads`

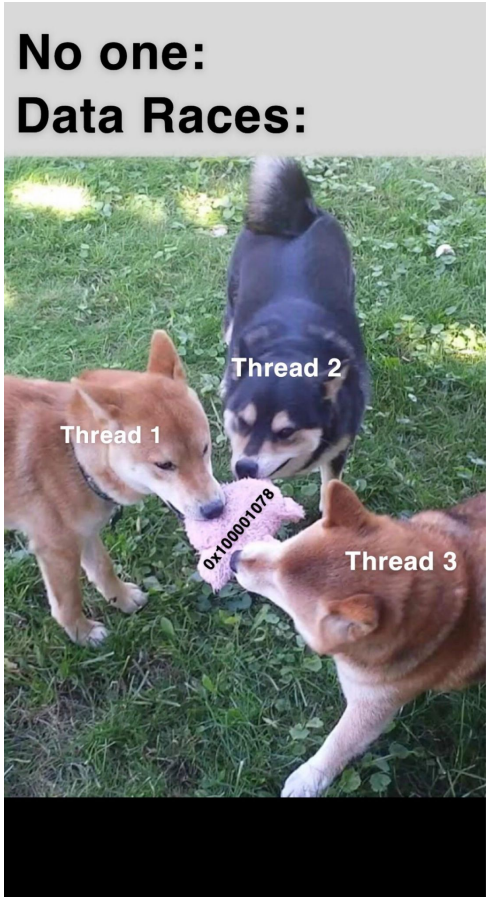
- EX12 and HW4 both use `pthreads` to create thread concurrency

Creation	`pthread_`**`create`**	**Parent: "Go do this {function}"**
Termination	`pthread_`**`exit`** `start_routine` returns	**Child: "I'm done with my task!"**
	`pthread_`**`cancel`**	**Parent: "I changed my mind, you can stop now"**
Resource Clean-up	`pthread_`**`join`**	**Parent: "I'll wait for you to finish and report back your result"** (resource persists until joined)
	`pthread_`**`detach`**	**Parent: "You're free now, go forth and prosper"** (automatically cleans up on termination)

Synchronization with Locking

- Remember, threads share an address space and system resources
- This makes it easy to communicate, but how do you avoid a total free-for-all?

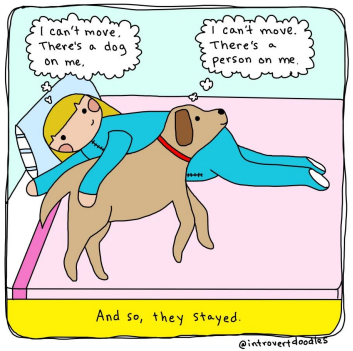
- Protect your critical sections with locks / mutexes!
  - Make sure nothing gets lost!



Problems with Synchronization

- Sharing Resources
  - Must be allocated / deallocated **exactly once**
  - Don’t use deallocated resources from other threads

- Locking is hard!
  - Too much, and performance is **worse than sequential**
  - Too little, and threads clash - **often unexpected results (unwanted interleaving)**
  - Not careful, and **deadlock** freezes your program forever!



Exercise 2

Exercise 2

```
// Assume all necessary libraries
and header files are included
const int NUM_TAS = 10;

static int bank_accounts[NUM_TAS];
static pthread_mutex_t sum_lock;

void *thread_main(void *arg) {
  int *TA_index = static_cast<int
*>(arg);

  pthread_mutex_lock(&sum_lock);
  bank_accounts[*TA_index] += 1000;
  pthread_mutex_unlock(&sum_lock);

  delete TA_index;
  return nullptr;
}
```

```
int main(int argc, char **argv) {
  pthread_t thds[NUM_TAS];
  pthread_mutex_init(&sum_lock, nullptr);

  for (int i = 0; i < NUM_TAS; i++) {
    int *num = new int(i);
    if (pthread_create(&thds[i], nullptr, &thread_main, num)!= 0){
      /*report error*/
    }
  }


  for (int i = 0; i < NUM_TAS; i++) {
    cout << bank_accounts[i] << endl;

  }

  pthread_mutex_destroy(&sum_lock);
  return 0;
}
```

Exercise 2

a. Does the program increase the TAs' bank accounts correctly? Why or why not?

b. Could we implement this program using processes instead of threads? Why would or why wouldn't we want to do this?

c. Assume that all the problems, if any, are now fixed. The student discovers that the program they wrote is kinda slow even though its a multithreaded program. Why might it be the case? And how would you fix that?

Exercise 2

a) Does the program increase the TAs' bank accounts correctly? Why or why not?

No, it's not correct. It requires main to call `pthread_join` to wait for each thread to finish before exiting the main program.

`pthread_exit()` will let a child thread finish leave to its parent, but it needs to be used in conjunction with `pthread_join` in order to check the results of the child thread.

Exercise 2

b) Could we implement this program using processes instead of threads? Why would or why wouldn't we want to do this?

We could, but doing so would require some way for the processes to communicate with each other so that the data structure can be "shared" (remember that inter-process communication can be difficult and time consuming).

It is much easier to just use threads since each thread could directly access the data structure.

Exercise 2

c) Assume that all the problems, if any, are now fixed. The student discovers that the program they wrote is kinda slow even though its a multithreaded program. Why might it be the case? And how would you fix that?

```
thread_mutex_lock(&sum_lock);
bank_accounts[*TA_index] += 1000;
pthread_mutex_unlock(&sum_lock);
```

→

```
thread_mutex_lock(&acct_lcks[*TA_index]);
bank_accounts[*TA_index] += 1000;
pthread_mutex_unlock(&acct_lks[*TA_index]);
```

Only one thread can increase the value of one account at a time and there is no difference from incrementing each account sequentially because we only have a single lock on this line for every single thread to share.

To fix this, we can have one lock per account so that multiple threads can increment the account at the same time. (An alternative solution is to just not use locks as well since the threads made will not conflict with each other, but we should aim for safe options for the bank accounts)

Homework Summary

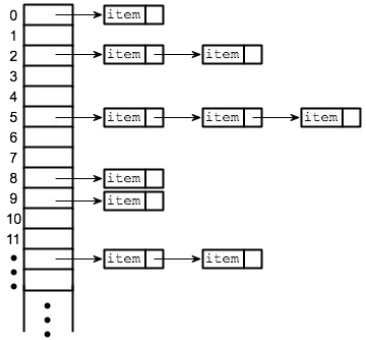
Homework Summary

Homework 1: Implemented a Doubly-Linked List and a Chained Hash Table

Homework 2: Created a local search engine (like `grep`)

Homework 3: Modified HW2 to save and read index files from disk

Homework 4: Creating an internet-accessible version of your search engine!




```
[attu1 hw2]$ ./searchshell ./test_tree
Indexing './test_tree'
enter query:
ulysses
  ./test_tree/books/ulysses.txt (10)
  ./test_tree/books/countofmontecristo.txt (2)
  ./test_tree/books/leavesofgrass.txt (1)
  ./test_tree/books/lesmiserables.txt (1)
  ./test_tree/books/paradiselost.txt (1)
enter query:
```

Exercise 3

Journaling Activity

Take about 5 minutes with yourself or your neighbor to journal about your experiences with homework and exercises this quarter.

Guiding Questions:

- What were a couple main themes or takeaways from each homework?
- What lessons (e.g., concepts, tools, habits, etc.) did you learn from each homework?
- If you could change something about a certain homework or exercise what would it be?
    - Let us know in the course evals :)

Outcomes of Doing the Homework

- Building a sample system based on the design decisions the staff has made for the homeworks
  - You can consider why these decisions were made

- Getting accustomed to larger-scale programming projects
  - Learning about the code surrounding what you are to implement

- Practicing programming idioms – error checking, style guides, etc.
- Reading Documentation!

Main Course Themes

- Debugging: gdb, valgrind, and… rubber duckies
  - Solving Seg. Faults, Memory Leaks, incorrect output
  - C/C++ programming hopefully made you more cognizant of where bugs may occur at a low level
- Program Layout: Modularity, Helper functions, Inheritance
  - Improve readability and ease of maintenance
- Documentation: Reading API’s and incorporating into programs
  - C++ Reference Guide, man pages
- Google Style Guide: linter, Lots of reading…
  - clint.py and cpplint.py are some automated tools to help with catching style issues
  - A lot of it was learning to adapt to a system for writing and reading code in the same format

Adding it to your Resume

- Why consider adding it in?
  - You had to work with this code base for an entire quarter
  - It demonstrates your ability to work with a large code base and system
  - It’s pretty cool to add “mini-Google” in

- What might be on there?
  - **System Design** – Creating data structures to store information. Saving it into a file. Following a protocol to create a web application
  - Programming in C and C++ (handling C/C++ idioms)
  - Working with people

Before We Go...

This quarter has been rough! We were swapping between virtual/in-person a lot!

This section wouldn't have been the same without all of you!

We were glad to have you! You earned your place here, you belong here (on this campus, in this program, in this course)

You know the content, take it slow, you can do it!

You've Learned A Lot! Good Luck!

Thank You for a Great Quarter!

Bonus Slides

TA-ing

- You are all well enough equipped to TA CSE333, CSE351, CSE374 and others.
- You do NOT have to 4.0 a class to TA it
- You do NOT have to be a super social person to TA
  (Some of us are very introverted)
- TAing will reinforce your understanding of any material
- **TAs are human too. It is ok to start off imperfect and make mistakes**
- **If you think you would be interested, I would highly recommend reaching out and giving it a try. Please feel free to talk to us if you are interested!**

Shoutout: Other Classes

Like C and the “mysterious” kernel?	451 OS
How about that “mysterious” compiler?	401 Compilers
Want to write a bunch of C++?	457 Graphics
Liked html (for some reason)?	154 Web Dev
Like doing a bunch of concurrency?	452 Distributed
Want to do C on limited systems?	474 Embedded Sys
Learn about more low-level stuff?	369 & 371 Digital Design
Want more 351-esque concepts?	469 & 470 Comp Arch
Want to understand the networking code you wrote?	461 Networks
Want to change some colors with C?	455 Vision
Liked patching up your 333gle? ssh-keys?	484 Security

Cool Fact: Segmentation Faults

- You've probably run afoul of SIGSEGV (a.k.a. "Seg fault")
  - What is it?
- UNIX processes can communicate with each other!
- signals are notifications sent between processes
  - They all have default handlers, such as "crash the program"
- You can use `signal()` or `sigaction()` to handle them yourself!

Cool Fact: Process Communication

- To send “real” messages between processes, you already have what you need in your toolkit!
  - Set up a socket connection from a process to itself
    - You’ll have to use nonblocking calls for this
  - `fork()`
  - Each process closes one end, and uses the other to communicate
- You can do this with TCP sockets, but there are better options available
- `socketpair()` does all of this for you!

Shoutout: The Rust Language

- No memory errors.
- No race conditions.
  - Whaaaaat? Yes.
- Performance close to C/C++ level
- Good abstractions like iterators & closures
  - Optimized down, so it’s as fast as if you wrote it by hand