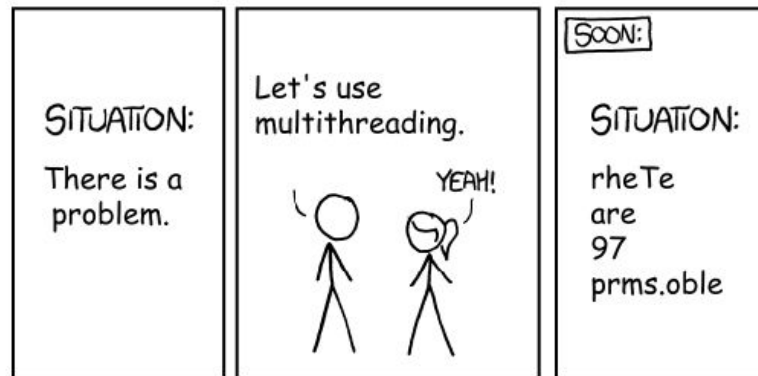
CSE 333
Section 7

Concurrency, pthreads



UNIVERSITY of WASHINGTON

Logistics

- Exercise 17
  - Due next **Monday** (08/12) @ **10:00am**
- Homework 4
  - Due next **Wednesday** (08/14) @ **11:00pm**
- Final
  - Friday next week (08/16) @ **1:10 pm**

pthreads

Concurrency with `pthreads`

- EX17 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)

pthread_create

```c
#include <pthread.h>
int pthread_create( pthread_t *thread,
          const pthread_attr_t *attr,
          void *(*start_routine) (void *),
          void *arg);
```

- `pthread_create` creates a new thread into `*thread`, with attributes `*attr` (`NULL` means default attributes)
- Returns `0` on success and an error number on error (can check against error constants)
- The new thread runs `start_routine(arg)`
- Compile and link with `-pthread`.

Other Ways Threads Terminate

- Let the thread function exit by itself; work is done
- Thread calls `pthread_exit` (thread terminates its own execution)
- Main thread calls `pthread_cancel` to close a child thread.
- The process exits from `main` or calls `exit`

pthread_exit

```
void pthread_exit(void *retval);
```

- Equivalent of `exit(retval);` for a thread instead of a process
  - This means it is called in the thread function (child thread)
  - Will only terminate the thread instead of the entire process (other threads will still run)
- The thread will automatically exit once it returns from `start_routine()`
  - `retval` is an output parameter to indicate success or failure (usually pass the address of a global variable to view).

Synchronizing Threads – Called by Parent Thread

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

- Waits for the thread specified by thread to terminate
- The thread equivalent of `waitpid()`
- The exit status of the terminated thread is placed in `**retval`

```
int pthread_detach(pthread_t thread);
```

- Mark thread specified by thread as detached – it will clean up its resources as soon as it terminates

Exercise 1

Exercise 1

```c
int g = 0;
void *worker(void *ignore) {
  for (int k = 1; k <= 3; k++) {
    g = g + k;
  }
  printf("g = %d\n", g);
  return NULL;
}

int main() {
  pthread_t t1, t2;
  int ignore;
  ignore = pthread_create(&t1, NULL, &worker, NULL);
  ignore = pthread_create(&t2, NULL, &worker, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return EXIT_SUCCESS;
}
```

What are the possible outputs of this program?

What is the range of values that `g` can have at the end of the program?

Exercise 1

```
int g = 0;
void *worker(void *ignore) {
  for (int k = 1; k <= 3; k++) {
    g = g + k;
  }
  printf("g = %d\n", g);
  return NULL;
}

int main() {
  pthread_t t1, t2;
  int ignore;
  ignore = pthread_create(&t1, NULL, &worker, NULL);
  ignore = pthread_create(&t2, NULL, &worker, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return EXIT_SUCCESS;
}
```

What are the possible outputs of this program?

Lots of possible answers, here are four:

```
g = 6   | g = 12 | g = 7 | g = 6
g = 12  | g = 12 | g = 9 | g = 11
```

Exercise 1

```c
int g = 0;
void *worker(void *ignore) {
  for (int k = 1; k <= 3; k++) {
    g = g + k;
  }
  printf("g = %d\n", g);
  return NULL;
}

int main() {
  pthread_t t1, t2;
  int ignore;
  ignore = pthread_create(&t1, NULL, &worker, NULL);
  ignore = pthread_create(&t2, NULL, &worker, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return EXIT_SUCCESS;
}
```

What is the range of values that g can have at the end of the program?

`g = [4 - 12]`

Assembly Instructions

Instructions for g = g + k:

```
mov     0x2ebf(%rip),%edx
mov     -0x4(%rbp),%eax
add     %edx,%eax
mov     %eax,0x2eb4(%rip)
```

Loads global g into local register

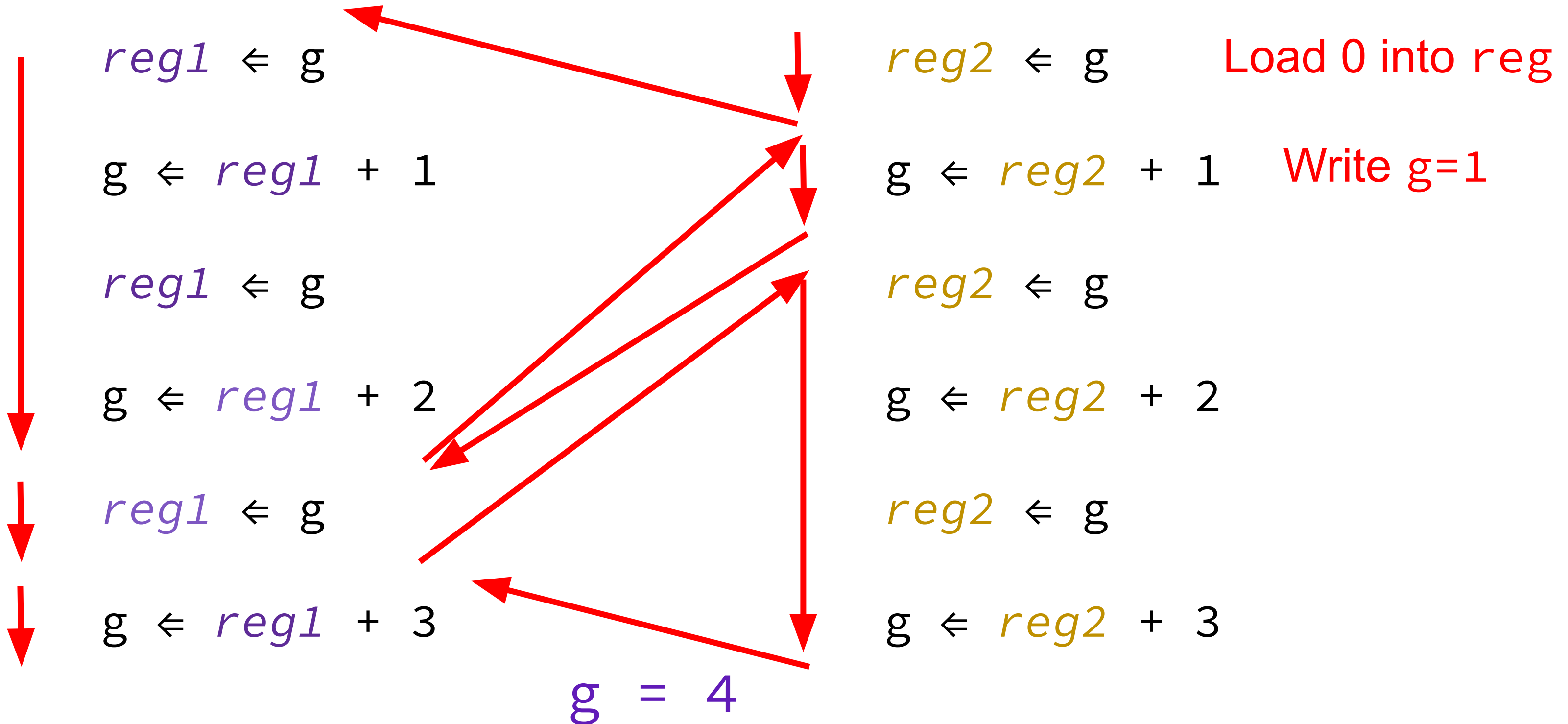
Loads k into %eax register

Adds copy of g in %edx to %eax register

Stores addition result back into global g

The "trick" is that because threads execute concurrently, the processor might be switched to a different thread after executing any instruction. When this sequence of code is executed, it could be interrupted between any two instructions by another thread that reads or writes global variable g.

Getting 4 from Exercise 1

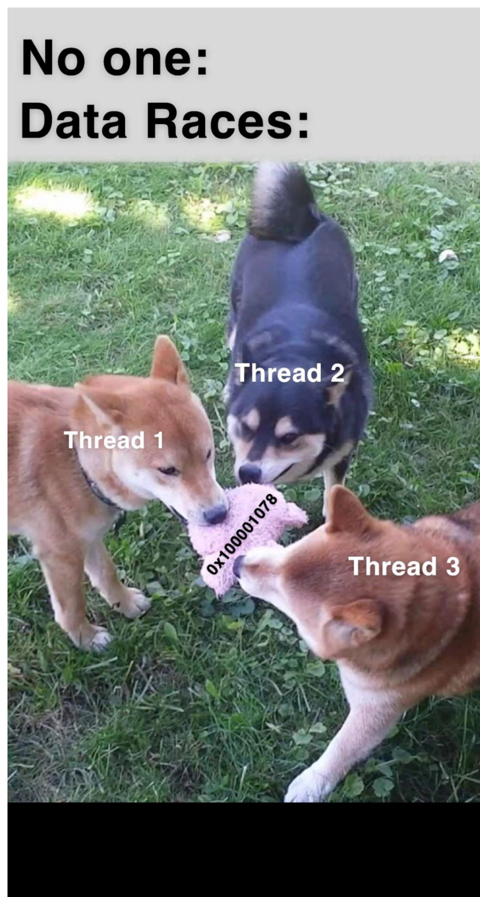


Synchronization

Synchronization

- 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!
  - Make sure nothing gets lost!
  - We'll be using `pthread_mutex`



Locking with `mutex`

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
```

- Initializes the mutex lock pointed to by `mutex` with lock attributes specified by `attr`.
- `attr` can be `NULL`.

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

- Destroys the lock
- Cleans up resource

Locking with `mutex`

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

- Grabs the lock
- If resource is locked, function will be blocked until resource is unlocked

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

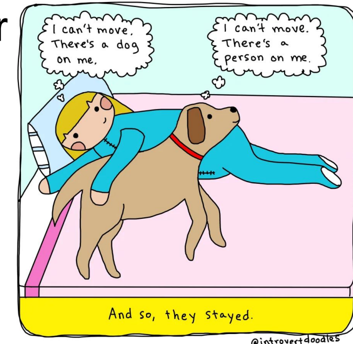
- Releases the lock

Problems with Synchronization

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

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

It's pay day! A CSE333 student wrote this program to pay all CSE TAs, answer the questions on the next slide (or on the worksheet) about this program.

```
// 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 needs to use pthread_join to wait for each thread to finish before exiting the main program.pthread_exit() might not be the best solution here. You want to check the return value of join to make sure the transaction applied rather than just exiting and trusting the threads to finish successfully. Gotta get those TA dolla's.

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)