CSE 333
Section 09

BOOST & p-threads

Logistics:

Due Monday (12/07):

Exercise 17 – pthreads. Out later this afternoon.

Due next Thursday (12/10):

HW4

BOOOOOOOOST

BOOST (HW4)

Boost is a free C++ library that provides support for various tasks in C++

- **Note:** Boost does NOT follow the Google style guide!!!

Boost adds many string algorithms that you may have seen in Java

- Include with `#include <boost/algorithm/string.hpp>`

We are showcasing a few we think could be useful for HW4, but more can be found here:

- https://www.boost.org/doc/libs/1_60_0/doc/html/string_algo.html

Helpful BOOST Functions

```
void boost::trim(string& input);
```

- Removes all leading and trailing whitespace from the string
- `input` is an input *and* output parameter (non-const reference)

```
void boost::replace_all(string& input, const string& search, const string& format);
```

- Replaces all instances of `search` inside `input` with `format`

More Helpful BOOST Functions

```
void boost::split(vector<string>& output,
                  const string& input,
                  boost::PredicateT match_on,
                  boost::token_compress_mode_type compress);
```

- Split the string by the characters in `match_on`

```
boost::PredicateT boost::is_any_of(const string& tokens);
```

- Returns predicate that matches on any of the characters in tokens

p-threads

POSIX threads (pthreads)

- The POSIX standard provides APIs for creating and manipulating threads.
- Part of the standard C/C++ libraries, declared in pthread.h
- Core pthread functions:
  - pthread_create - “Go do this {function}”
  - pthread_exit - “I’m done with my task!”
  - pthread_join - “I’ll wait for you to report back your result”
  - pthread_cancel - “I changed my mind, you can stop now”
  - pthread_detach - “You’re free now, go forth and prosper”

pthread_create

```
#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 and calls start_routine with arg as its parameter.
- `pthread_create` arguments:
  - **thread:** Pointer to a unique identifier for the new thread. (output parameter)
  - **attr:** An attribute object that may be used to set thread attributes. Use NULL for the default values.
  - **start_routine:** The C routine that the thread will execute once it is created.
  - **arg**: A single argument that may be passed to *start_routine*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.
- Compile and link with –pthread.

Terminating Threads

- There are several ways in which a thread may be terminated:
  - The thread returns normally from its starting routine; Its work is done.
  - The thread makes a call to the `pthread_exit` subroutine - whether its work is done or not.
  - The thread is canceled by another thread via the `pthread_cancel` routine.
  - The entire process is terminated due to making a call to either the `exec()` or `exit()`.
  - If `main()` finishes first, without calling `pthread_exit` explicitly itself.

pthread_exit

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

- Allows the user to terminate a thread and to specify an optional termination status parameter, *`retval`*.
- In subroutines that execute to completion normally, you can often dispense with calling `pthread_exit()`.
- Calling pthread_exit() from main():
  - If `main()` finishes before the threads it spawned and does not call `pthread_exit()` explicitly, all the threads it created will terminate.
  - To allow other threads to continue execution, the main thread should terminate by calling `pthread_exit()` rather than `exit()`.

Demo simple_pthreads.cc

pthread_join

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

- Synchronization between threads.
- `pthread_join` blocks the calling thread until the specified thread terminates and then the calling thread joins the terminated thread.
- Only threads that are created as joinable can be joined; a thread created as detached can never be joined. (Refer `pthread_create`)
- The target thread's termination return status can be obtained if it was specified in the target thread's call to `pthread_exit()`.

Demo: *pthreads.cc*

Exercise:
Concurrency Reasoning

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?

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 a few:

g = 6

g = 12

g = 12

g = 6

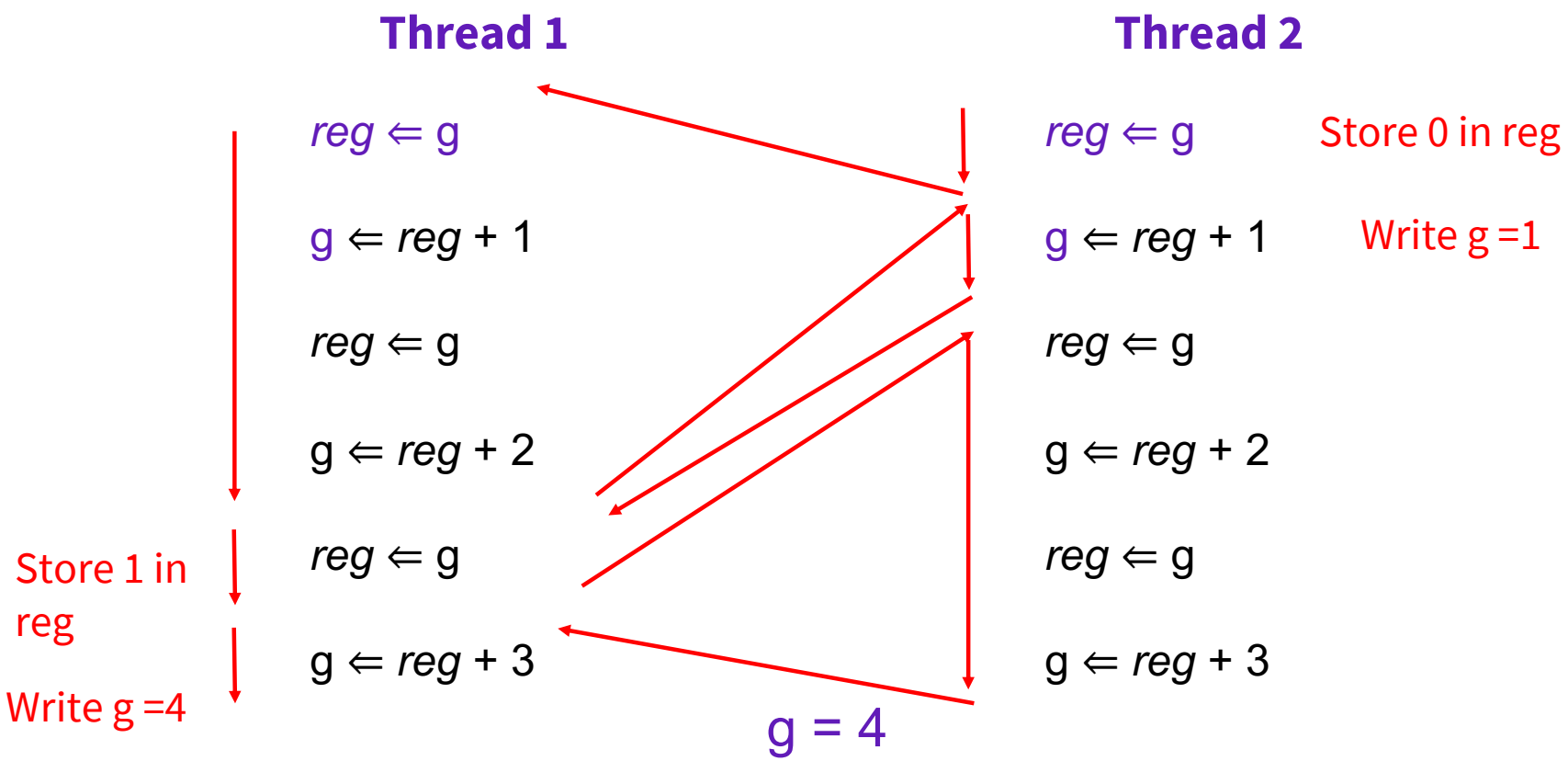
g = 11

g = 7

g = 12

g = 9

How to get 4 from exercise 1



Locking - 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_lock(pthread_mutex_t *mutex);
```

- Grabs the lock

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

- Releases the lock

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

- Destroys the lock

Threads - Locking

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

```
pthread_mutex_lock(&lock);
if (!do_computation(resource)) {
        printf("Error doing computation\n");
        return false;  // !!!
}
pthread_mutex_unlock(&lock);
return true;
```

Demo total.cc & total_locking.cc

Exercise:
Mutex Reasoning

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 =
reinterpret_cast<int*>(arg);

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

  delete TA_index;
  return NULL;
}
```

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

  for (int i = 0; i < NUM_TAS; i++) {
    int *num = new int(i);
    if (pthread_create(&thds[i], NULL, &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?

No its 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.

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?

Because there is a lock over the entire bank account array, so only one thread can increase the value of one account at a time and there is no difference from incrementing each account sequentially. To fix this, we can have one lock per account so that multiple threads can increment the account at the same time. (With the current setup, we could also just not use a lock since we know that no thread will have a conflicting TA_index. For a more generalized program, it would be better to use the first answer.)