333 Section 9 - Concurrency and pthreads

Welcome back to section! We're happy you're here ʕ•ᴥ•ʔ⊹₊✧⋆

Boost Library

- Very useful for dealing with strings (HW4!), such as trimming, pattern matching, splitting, replacing, etc.
- `#include <boost/algorithm/string.hpp>`.
- Boost is template madness, so we have some handy reference information here :)

```
// Trim the whitespace off the left and right of to_modify
// to_modify is input and output parameter
void boost::trim(string& to_modify);

// Replace all instances of to_find with to_replace.
// to_modify is input and output parameter
void boost::replace_all(string& to_modify,
                        const string& to_find,
                        const string& to_replace);

// Returns a predicate that matches on any of the characters in tokens
boost::PredicateT boost::is_any_of(const string& tokens);

// Split the string by the characters in match_on
void boost::split(vector<string>& output,
                  const string& input,
                  boost::PredicateT match_on,
                  boost::token_compress_mode_type compress);
```

Examples:

TRIM:
```
string s("  \t  HI  \n  ");
boost::algorithm::trim(s);  // s == "HI"
```

REPLACE_ALL:
```
string s1("ynrnrt");
boost::algorithm::replace_all(s1, "nr", "e");  // s1 == "yeet"

string s2("queue?");
boost::algorithm::replace_all(s2, "que", "q"); // s2 == "que?"
```

SPLIT:
```
string str1("hello abc-*-ABC-*-aBc goodbye");
vector<string> SplitVec;  // #2: Search for tokens
split( SplitVec, str1, is_any_of("-*"), token_compress_on );
  // SplitVec == { "hello abc", "ABC", "aBc goodbye" }
```

POSIX threads (pthreads) API

- Part of the standard C/C++ libraries and declared in `pthread.h`.
- **Must compile and link with** `-pthread`.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

➔ `thread`: Output parameter for thread identifier

➔ `attr`: Used to set thread attributes. Use `NULL` for defaults.

➔ `start_routine`: Pointer to a function that the thread will execute upon creation.

➔ `arg`: A single argument that may be passed to `start_routine`. `NULL` may be used if no argument is to be passed.

★ Creates a new thread and calls `start_routine(arg)`.

★ Returns 0 if successful and an error number otherwise.

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

★ Called by parent thread to wait for the termination of the thread specified by `thread`. If `retval` is non-`NULL`, then `retval` acts an output parameter and the address passed to `pthread_exit` by the finished thread is stored in it.

★ Returns 0 if successful and an error number otherwise.

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

★ Terminates the calling thread with an optional termination status parameter, `retval`, which can just be set to `NULL`.

POSIX mutual exclusion (mutex) API

- Restrict access to sections of code in order to protect shared data from being simultaneously accessed by multiple threads.

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

★ Initializes the mutex referenced by `mutex` with attributes specified by `attr` (use `NULL` for adefault ttributes).

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

★ Destroys (*i.e.* uninitializes) the mutex object referenced by `mutex`.

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

★ Attempts to acquire the mutex object referenced by `mutex` and blocks if it's currently held by another thread. Should be placed at the start of your critical section of code.

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

★ Releases the mutex object referenced by `mutex`. Should be placed at the end of your critical section of code.

Exercise 1) Consider the following multithreaded C program:

```
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;
}
```

Give three different possible outputs (there are many)

What are the possible final values of the global variable ‘g’? (circle all possible)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15+

Exercise 2) It's the payday! It's time for UW to pay each of the 333 TAs their monthly salary. Each of the TA's bank accounts is inside the `bank_accounts[]` array and the person who is in charge of paying the TAs is a 333 student and decided to use pthreads to pay the TAs by adding 1000 into each bank account. Here is the program the student wrote:

```
// 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;
}
```

(see next page for discussion questions)

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 it's a multithreaded program. Why might it be the case? And how would you fix that?

Exercise 3 (Bonus!)

Write a function called `ExtractRequestLine` that takes in a well-formatted HTTP request as a `string` and returns a `map` with the keys as `method`, `uri`, `version` and the values from the corresponding request. For example,

Example Input:

```
"GET /index.html HTTP/1.1\r\nHost: www.mywebsite.com\r\nConnect ion:
keep-alive\r\nUpgrade-Insecure-Requests: 1\r\n\r\n"
```

Map Returned:

```
{
    "method"  : "GET"
    "uri"     : "/index.html"
    "version" : "HTTP/1.1"
}
```

```
map<string,string> ExtractRequestLine(const string& request) {









}
```