CS 35L- Software Construction Laboratory

Fall 18

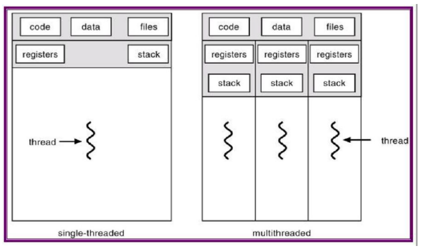
TA: Guangyu Zhou

Quick Review: Thread vs. Process

- A thread is short for 'thread-of-execution'.
  - It represents the sequence of instructions that the CPU has (and will) execute.
- A process can be
  - Single-threaded
  - Multi-threaded
- Threads in a process can run in parallel
- A thread is a lightweight process
- It is a **basic unit** of CPU utilization

Quick Review: Thread vs. Process (cont.)

- Each thread has its own:
  - Stack
  - Registers
  - Thread ID
- Each thread shares the following with other threads belonging to the same process:
  - Code
  - Global Data
  - Heap
  - Files



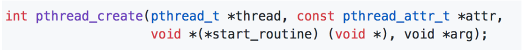
Pthread API

- To use **pthreads** you will need to include <pthread.h> AND you need to compile with **-lpthread** compiler option. This option tells the compiler that your program requires threading support

- There are 5 basic pthread functions:

1. **pthread_create: creates a new thread within a process**
2. **pthread_join: waits for another thread to terminate**
3. <u>pthread_exit: terminates the currently running thread</u>
4. pthread_equal: compares thread ids to see if they refer to the same thread
5. pthread_self: returns the id of the calling thread

pthread_create



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

- Function: creates a new thread and makes it executable
- Can be called any number of times from anywhere within code
- Return value:
  - Success: zero
  - Failure: error number
- **PTHREAD_THREADS_MAX:** Constant for max number of threads that can be created

Parameters

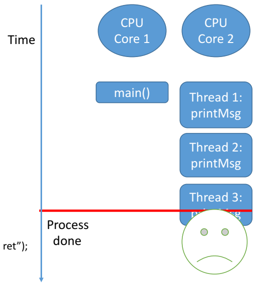
- int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(my_function)(void *), void *arg);
  - tid: unique identifier for newly created thread
  - attr: object that holds thread attributes (priority, stack size, etc.)
    - Pass in NULL for default attributes
  - my_function: function that thread will execute once it is created
  - arg: a single argument that may be passed to my_function
    - Pass in NULL if no arguments

Question

- If I call **pthread_create** twice, at least how many stacks does your process have?
- A. 1
- B. 2
- C. 3
- Your process will contain three stacks - one for each thread. The first thread is created when the process starts, and you created two more. Actually there can be more stacks than this, but let's ignore that complication for now.

pthread_create Example

```
#include <pthread.h> …
void *printMsg(void *thread_num) {
        int t_num = (int) thread_num;
        printf(“It’s me, thread #%d!\n”, t_num); }

int main() {
        pthread_t tids[3];
        int t;
        for(t = 0; t < 3; t++) {
                        ret = pthread_create(&tids[t], NULL, printMsg, (void *) t);
                        if(ret) {
                                        printf(“Error creating thread. Error code is %d\n”, ret”);
                                        exit(-1); }
        }
}
```



Possible problem with this code?

If main thread finishes before all threads finish their job -> incorrect results

pthread_join

- Function: makes originating thread wait for the completion of all its spawned threads’ tasks
- Without join, the originating thread would exit as soon as it completes its job
  - A spawned thread can get aborted even if it is in the middle of its chore
- Return value:
  - Success: zero
  - Failure: error number

Arguments

- int pthread_join(pthread_t tid, void **status);
- tid: thread ID of thread to wait on
- status: the exit status of the target thread is stored in the location pointed to by *status
  - Pass in NULL if no status is needed

What is the purpose of pthread_join?

- Wait for a thread to finish
- Clean up thread resources
- Grabs the return value of the thread

pthread_join Example

```
#include <pthread.h> …
#define NUM_THREADS 5

void *PrintHello(void *thread_ num) {
        printf("\n%d: Hello World!\n", (int) thread_num); }

int main() {
        pthread_t threads[NUM_THREADS];
        int ret, t;
Int threadID[NUM_THREADS]
        for(t = 0; t < NUM_THREADS; t++) {
                printf("Creating thread %d\n", t);
                threadID[t[ = t
                ret = pthread_create(&threads[t], NULL, PrintHello, &threadID[t]);
         }

         for(t = 0; t < NUM_THREADS; t++) {
                ret =  pthread_join(threads[t], NULL);
         }
}
```

pthread_exit

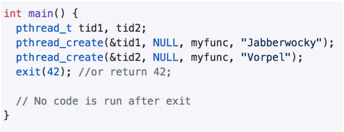
- pthread_exit(void *) only stops the **calling** thread i.e. the thread never returns after calling pthread_exit.
- The pthread library will automatically finish the process if there are no other threads running.

pthread_join vs. pthread_exit

- Both pthread_exit and pthread_join will let the other threads finish on their own (even if called in the main thread).
- However, only pthread_join will return to you when the specified thread finishes. pthread_exit does not wait and will immediately end your thread and give you no chance to continue executing.

How can a thread be terminated?

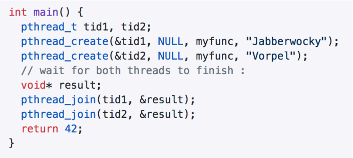
- Terminating the process: exit(); returning from main
- Returning from the thread function
- Calling pthread_exit



```
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, myfunc, "Jabberwocky");
  pthread_create(&tid2, NULL, myfunc, "Vorpel");
  exit(42); //or return 42;

  // No code is run after exit
}
```

How can a thread be terminated?

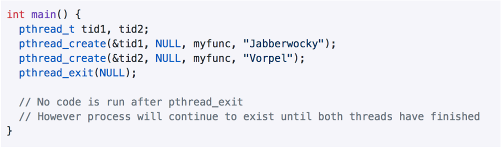
- Terminating the process: exit(); returning from main
- Returning from the thread function
- Calling pthread_exit



```
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, myfunc, "Jabberwocky");
  pthread_create(&tid2, NULL, myfunc, "Vorpel");
  // wait for both threads to finish :
  void* result;
  pthread_join(tid1, &result);
  pthread_join(tid2, &result);
  return 42;
}
```

How can a thread be terminated?

- Terminating the process: exit(); returning from main
- Returning from the thread function
- Calling pthread_exit (will create “Zombie” thread)



```
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, myfunc, "Jabberwocky");
  pthread_create(&tid2, NULL, myfunc, "Vorpel");
  pthread_exit(NULL);

  // No code is run after pthread_exit
  // However process will continue to exist until both threads have finished
}
```

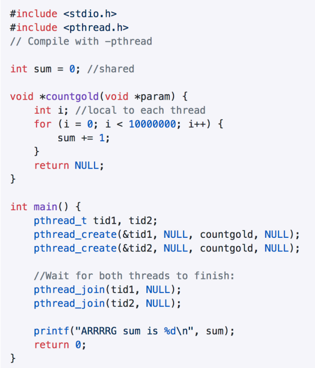
Other basic pthread functions

- int pthread_equal(pthread_t thread_1, pthread_t thread_2);
  - The *pthread_equal()* compares the thread ids *thread_1* and *thread_2* and returns a non 0 value if the ids represent the same thread otherwise 0 is returned.
- pthread_t pthread_self(void);
  - The *pthread_self()* returns the thread id of the calling thread.

Quick Review: Thread safety/synchronization

- **Thread safe function** - safe to be called by multiple threads at the same time. Function is free of 'race conditions' when called by multiple threads simultaneously
- **Race condition** - the output depends on the order of execution
- **Critical section** - a section of code that can only be executed by one thread at a time, if the program is to function correctly.
  - If two threads (or processes) were to execute code inside the critical section at the same time then it is possible that program may no longer have correct behavior.
- Question: Is just incrementing a variable a critical section?
  - Possibly!

Incrementing a variable causes race condition



```
#include <stdio.h>
#include <pthread.h>
// Compile with -pthread

int sum = 0; //shared

void *countgold(void *param) {
    int i; //local to each thread
    for (i = 0; i < 10000000; i++) {
        sum += 1;
    }
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, countgold, NULL);
    pthread_create(&tid2, NULL, countgold, NULL);

    //Wait for both threads to finish:
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("ARRRRG sum is %d\n", sum);
    return 0;
}
```

- Typical output of the this code is ARGGGH sum is 8140268. A different sum is printed each time the program is run because there is a race condition;
  - the code does not stop two threads from reading-writing **sum** at the same time.
- For example both threads copy the current value of sum into CPU that runs each thread (let's pick 123). Both threads increment one to their own copy. Both threads write back the value (124). If the threads had accessed the sum at different times then the count would have been 125.

Quick Review: Thread safety/synchronization

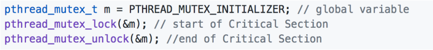
- Order 1
  - balance = 1000
  - T1 - Read balance (1000)
  - T1 - Deduct 50: 950 in temporary result
  - T2 - read balance (1000)
  - T1 - update balance: 950 at this point
  - T2 - add 150 to balance: 1150 in temporary result
  - T2 - update balance: balance is 1150 at this point
- **The final value of balance is 1150**

- Order 2
  - balance = 1000
  - T1 - read balance (1000)
  - T2 - read balance (1000)
  - T2 - add 150 to balance: 1150 in temporary result
  - T1 - Deduct 50: 950 in temporary result
  - T2 - update balance: balance is 1150 at this point
  - T1 - update balance: balance is 950 at this point
- **The final value of balance is 950**

Thread synchronization

- **Mutex (mutual exclusion)**

  Threads start with "Mutex.lock()" and end with "Mutex.unlock()"
  - Thread 1
    - Read balance
    - Deduct 50 from balance
    - Update balance with new value
  - Thread 2
    - Read balance
    - Add 150 to balance
    - Update balance with new value
- Only one thread will get the mutex. Other thread will **block in Mutex.lock()**
- Other thread can start execution only when the thread that holds the mutex calls **Mutex.unlock()**



```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER; // global variable
pthread_mutex_lock(&m); // start of Critical Section
pthread_mutex_unlock(&m); //end of Critical Section
```

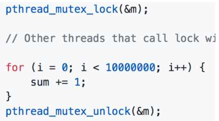
A complete Example with Mutex

```
#include <stdio.h>
#include <pthread.h>

// Create a mutex this ready to be locked!
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int sum = 0;

void *countgold(void *param) {
    int i;

//Same thread that locks the mutex must unlock it
    pthread_mutex_lock(&m);

    for (i = 0; i < 10000000; i++) {
        sum += 1;
    }

// Other threads that call lock will have to wait until we call unlock
    pthread_mutex_unlock(&m);
    return NULL;
}
```

```
int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, countgold, NULL);
    pthread_create(&tid2, NULL, countgold, NULL);

    //Wait for both threads to finish:
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("ARRRRG sum is %d\n", sum);
    return 0;
}
```

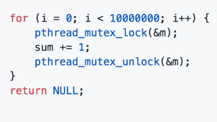
Overhead in calling lock and unlock

- Are they all correct?
- Which one is faster?
  - C > A > B



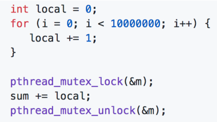
```
pthread_mutex_lock(&m);

// Other threads that call lock wi

for (i = 0; i < 10000000; i++) {
    sum += 1;
}
pthread_mutex_unlock(&m);
```

A



```
for (i = 0; i < 10000000; i++) {
    pthread_mutex_lock(&m);
    sum += 1;
    pthread_mutex_unlock(&m);
}
return NULL;
```

B



```
int local = 0;
for (i = 0; i < 10000000; i++) {
   local += 1;
}

pthread_mutex_lock(&m);
sum += local;
pthread_mutex_unlock(&m);
```

C

Summary of Multi-Thread Programming

- Multithreads is an efficient way to parallelize tasks
- **Thread switches are less expensive** compared to process switches (context switching)
- Inter-thread communication is easy, via **shared global** data
- Need **synchronization** among threads accessing same data
  - e.g. Mutex.lock(), Mutex.unlock()