Introduction to Concurrency and Pthreads

CMPU 224 – Computer Organization
Jason Waterman

Thread

- A new abstraction for **a single running process**

- Multi-threaded program
  - A multi-threaded program has **more than one point of execution**
    - Multiple program counters, one for each thread
    - Also known as **multiple threads of execution**
  - Threads in a process **share the same address space (memory)**
  - Each thread has its own stack
  - Each thread has its own private set of registers

Context switch between threads

- Each thread has its own program counter and set of registers
  - One or more **thread control blocks (TCBs)** are needed to store the state of each thread

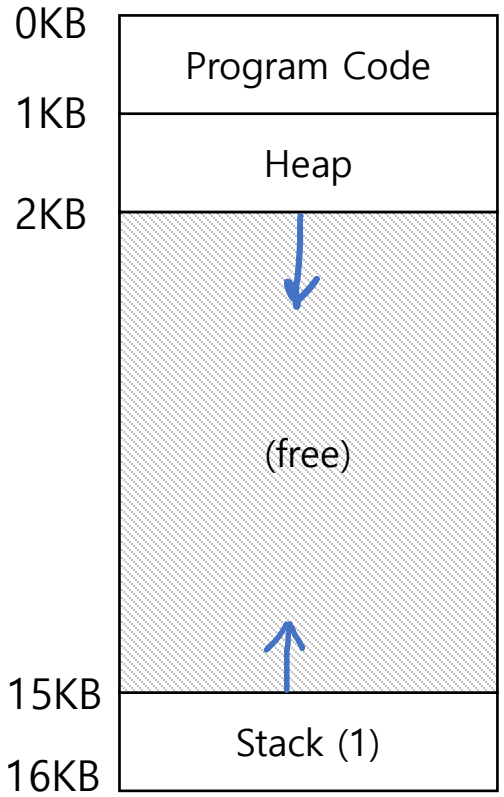
- When switching from running one thread (T1) to running the another (T2):
  - The register state of T1 be saved
  - The register state of T2 restored
  - The **address space remains** (memory) the same

Why Use Threads?

- Two major reasons
  1. Parallelism (multi-core)
     - Divide a task among several threads
     - On a system with multiple processors threads can work in parallel with each other
  2. Overlap of I/O with other activities **within** a single program (single-core)
     - When one thread requests I/O (e.g., disk access) from the system, switch to another thread ready to work

The stack of the relevant thread

- There is **one stack per thread** of execution

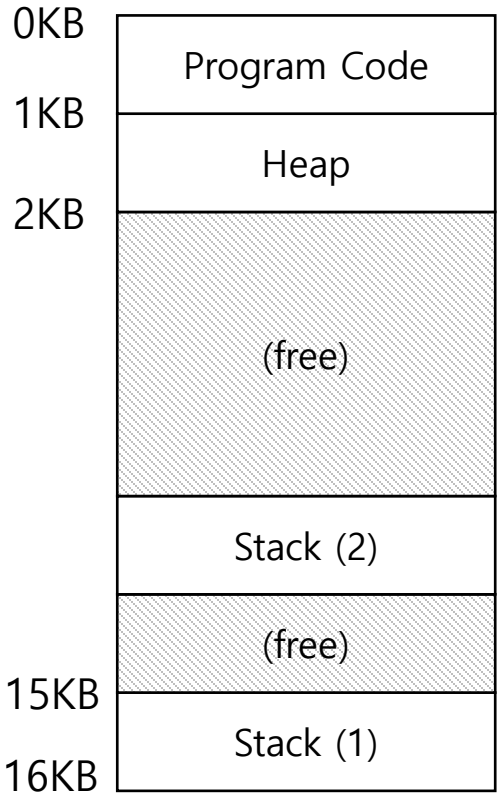


A Single-Threaded Address Space

The code segment: where instructions live

The heap segment: contains malloc'd data dynamic data structures. It grows towards increasing memory addresses

The stack segment: contains local variables arguments to routines, return values, etc. It grows towards decreasing memory addresses



Two threaded Address Space

Example: Creating a Thread

```
#include <pthread.h>
void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}


int main(int argc, char *argv[]) {
    if (argc != 1) {
        fprintf(stderr, "usage: main\n");
    exit(1);
    }

    pthread_t p1, p2;
    printf("main: begin\n");
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Thread Creation

- How to create and control threads?

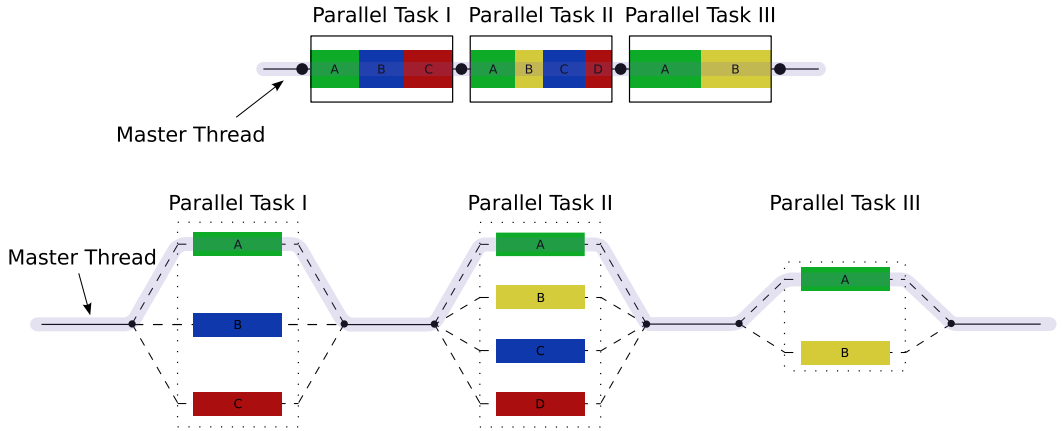
```
#include <pthread.h>

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

  - **`thread`**: Thread object used to interact with this thread
  - **`attr`**: Used to specify any attributes this thread might have
    - Stack size, Scheduling priority, etc. – pass in `NULL` if not changing any attributes
  - **`start_routine`**: the function the thread executes when first started
  - **`arg`**: the argument to be passed to the function (`start_routine`)
    - *a void pointer* allows us to pass in *any type of* argument
  - **return value**: on success, returns 0; on error, it returns an error number, and the contents of `thread` are undefined

Wait for a thread to complete

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

- **`thread`**: Specify which thread **to wait for**
- **`value_ptr`**: A pointer to the **return value**
  - Because `pthread_join()` routine changes the value, you need to **pass in a pointer** to the return value (which is of type void *)
  - If you don't care about the return value of the thread, pass in NULL
- The name join comes from the fork-join model of executing parallel programs



By Wikipedia user A1 - w:en:File:Fork_join.svg, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=32004077

Example: Creating a Thread with Arguments

```
#include <pthread.h>

typedef struct __myarg_t {
        int a;
        int b;
} myarg_t;

void *mythread(void *arg) {
        myarg_t *m = (myarg_t *) arg;
        printf("%d %d\n", m->a, m->b);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p;
        int rc;

        myarg_t args;
        args.a = 10;
        args.b = 20;
        rc = pthread_create(&p, NULL, mythread, &args);
        …
}
```

Example: Returning Data from a Thread

```
1   #include <stdio.h>
2   #include <pthread.h>
3   #include <assert.h>
4   #include <stdlib.h>
5
6   typedef struct __myarg_t {
7       int a;
8       int b;
9   } myarg_t;
10
11  typedef struct __myret_t {
12      int x;
13      int y;
14  } myret_t;
15
16  void *mythread(void *arg) {
17      myarg_t *m = (myarg_t *) arg;
18      printf("%d %d\n", m->a, m->b);
19      myret_t *r = malloc(sizeof(myret_t));
20      r->x = 1;
21      r->y = 2;
22      return (void *) r;
23  }
24
```

```
25  int main(int argc, char *argv[]) {
26      int rc;
27      pthread_t p;
28      myret_t *m;
29
30      myarg_t args;
31      args.a = 10;
32      args.b = 20;
33      pthread_create(&p, NULL, mythread, &args);
34      pthread_join(p, (void **) &m);
35      // this thread has been
36      // waiting inside of the
37      // pthread_join() routine.
38      printf("returned %d %d\n", m->x, m->y);
39      return 0;
40  }
```

Example: Dangerous code

- Be careful with **how values are returned** from a thread

```
1   void *mythread(void *arg) {
2       myarg_t *m = (myarg_t *) arg;
3       printf("%d %d\n", m->a, m->b);
4       myret_t r; // Danger!!!!! Why is this bad?
5       r.x = 1;
6       r.y = 2;
7       return (void *) &r;
8   }
```

  - Variable `r` is allocated on the stack
  - When the thread returns, `r` is automatically **de-allocated**

Example: Simpler Argument Passing to a Thread

- Passing in a single integer value

```
1   void *mythread(void *arg) {
2       int m = (int) arg;
3       printf("%d\n", m);
4       return (void *) (arg + 1);
5   }
6
7   int main(int argc, char *argv[]) {
8       pthread_t p;
9       int rc, m;
10      pthread_create(&p, NULL, mythread, (void *) 100);
11      pthread_join(p, (void **) &m);
12      printf("returned %d\n", m);
13      return 0;
14  }
```

Threading: Shared Variables

```
static volatile int counter = 0; // shared variable global to the .c file

// mythread()
// Simply adds 1 to counter repeatedly, in a loop. No, this is not how you would add 10,000,000 to a counter,
// but it shows the problem nicely.
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);

    return NULL;

// main()
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
int main(int argc, char *argv[]) {

    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    Pthread_join(p1, NULL); // join waits for the threads to finish
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Race condition

- Example with two threads and counter = 50
  - counter = counter + 1 (runs twice, once for each thread)
  - We expect the result to be 52

```
100 lw   a0,-608(gp)#<counter>
104 addi a0,a0,1
108 sw   a0,-608(gp)#<counter>
```

OS	Thread1	Thread2	PC (after instruction)	a0	Counter (0x13788)
			100	0	50
	lw a0,-608(gp)		104	50	50
	addi a0,a0,1		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		lw a0,-608(gp)	104	50	50
		addi a0,a0,1	108	51	50
		sw a0,-608(gp)	112	51	51
interrupt					
save T2's state					
restore T1's state			108	51	50
	sw a0,-608(gp)		112	51	**51**

Critical section

- In the previous example, there are three actions that take place to increase the counter
  - **Read** the current value from memory
  - **Modify** the value
  - **Write** the new value to memory
- These three operations must happen **atomically**
  - Once the read takes place, the thread should not be interrupted by another thread that uses the shared variable until after the modify and write occur
- A piece of code that **accesses a shared variable** and must not be concurrently executed by more than one thread is called a **critical section**
  - Multiple threads executing critical section can result in a **race condition**
  - Need to support **atomicity** for critical sections (**mutual exclusion**)

Locks

- Locks ensure that any such critical section executes as if it were a single atomic instruction (**execute a series of instructions atomically**)

```
1     lock_t mutex;
2     . . .
3     lock(&mutex);
4     counter = counter + 1;
5     unlock(&mutex);
```

Critical section

Locks

- Provide **mutual exclusion** to a critical section
  - Interface – two operations **lock** and **unlock**

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

  - Usage (without lock initialization and error checking)

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

    - If no other thread holds the lock → the thread will acquire the lock and **enter the critical section**
    - If another thread holds the lock → the thread will **not return from the call** until it has acquired the lock

Lock Initialization

- All locks must be **properly initialized**
  - One way: using `PTHREAD_MUTEX_INITIALIZER`

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

  - The dynamic way: using `pthread_mutex_init()`

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Lock Error Checking

- Always check the return value for errors when calling lock and unlock
  - An example wrapper

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

Compiling and Running

- To compile them, you must include the header `pthread.h`
  - Explicitly link with the **pthreads library**, by adding the `-pthread` flag

```
prompt> gcc -o main main.c -Wall -pthread
```

  - For more information

```
man -k pthread
```

Thread API Guidelines

- **Keep it simple**
  - Tricky thread interactions lead to bugs
- **Minimize thread interactions**
  - Each interaction should be carefully thought out and constructed with known design patterns (which we will learn about)
- **Check your return codes**
  - Failure to do so will lead to bizarre and hard to understand behavior
- **Be careful how you pass arguments and return values**
  - If you returning a reference to a variable on the stack, you are going to have a bad time
- **Each thread has its own stack**
  - To share data between threads, the values must be in the heap or a global variable
- **Use the man pages**
  - Highly informative with good examples