CSE2431 – Lecture Topic 7
Concurrency (part 1)

Concurrency (Part 1) Overview

Instructor: Luan Duong, Ph.D.

CSE 2431: Introduction to Operating Systems

Reading: **Chap. 26 [OSTEP]**

Remarks

- This is a programming-heavy topic
  - There will be three labs related to this topic
  - Writing a correct concurrent program is hard
- You will find what you learn in this topic will be frequently used in your future career (perhaps not now)

Outline: Concurrency (part 1)

- Revision: Threads
- Multi-threading and Multi-processing
- Reason why we need this topic: Sharing makes life more difficult

Why concurrent program?

- Moore's law: the number of transistors in a dense integrated circuit has doubled approximately every two years

- Before 2005, people use more transistors to build faster CPUs
  - The speed of a program can automatically increase with a faster CPU.

- Today people use more transistors to build more CPUs
  - The speed of a program does not automatically increase with more CPUs.
  - We have to parallelize our program.

Support concurrent program

- We have Thread (Thread Control Blocks – TCBs) → Multi-Threading
  - Multiple pieces of code (threads) in parallel
  - Multiple threads share the same address space
  - One PC per thread
- Sharing address space (single-threaded vs. multi-threaded address space)

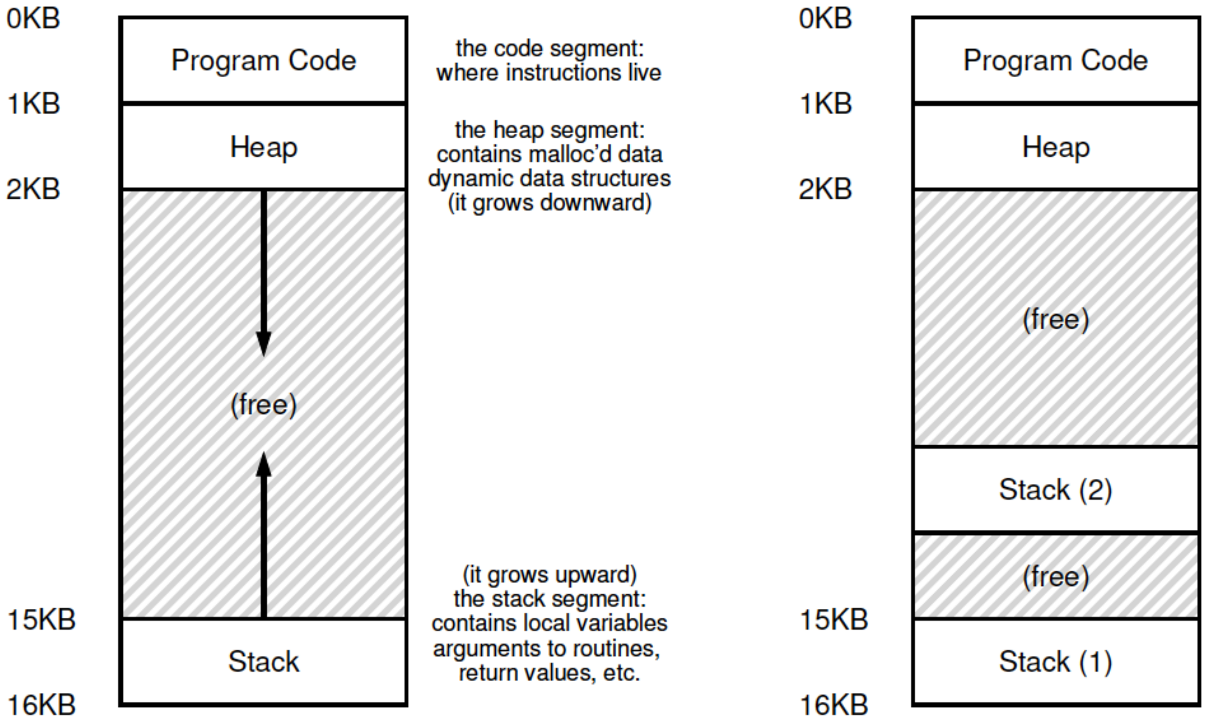

Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

Multi-threading vs., Multi-processing

- First, there is nothing multi-threading can do but multi-processing cannot do.
- Fundamental difference: threads share the memory address space, but processes do not
- Trade-offs:
  - Communication among processes are harder and less efficient, although possible (sockets, pipes, etc)
  - Processes provide better protection: if one process is faulty, others are not affected. If one thread has a bug, all threads can be affected.
  - If you need to parallelize a program on multiple machines (distributed systems), then multi-processing is necessary at this moment.

Revision: Thread Creation Workflow (1)

- First, define the function the thread is going to run:
  - **`void *fun_name(void *args)`** (function must have this format)
  - **`args`** contains all arguments necessary for this code
- If you want different threads to do different things:
  - Define several different functions
  - Or define one function, pass different arguments to it

```
void *mythread(void *arg) {
  printf(“%s\n”, (char *) arg);
  return NULL;
}
```

Revision: Thread Creation Workflow (2)

- **Next, create the threads**
- `int pthread_create(pthread_t *pid, const pthread_attr_t *attr, void *(*routine)(void *), void *args);`
  - **Return value:** whether creation succeeds
  - **pid**: Pthread ID, used to control the thread
  - **attr**: thread attributes (NULL works fine for now)
  - **routine:** the function the thread is going to run (see previous slide)
  - **args**: the arguments you want to pass to routine
- How do we pass multiple arguments? Define a struct.

```
#include<assert.h>
#include<pthread.h>
/* ...... */
rc = pthread_create(&p1, NULL, mythread "A"); assert(rc == 0);
rc = pthread_create(&p1, NULL, mythread "A"); assert(rc == 0);
```

Revision: Thread Creation Workflow (3)

- We can now **run** the threads
  - `int` **`pthread_join`**`(pthread_t pid, void **value_ptr)`
    - Wait until a thread completes
    - `return value`: success or not
    - `pid`: pthread_id you got from `pthread_create()`
    - `value_ptr`: assume `NULL` for now

Thread: Caution!

- `pthread_create()` only **creates** a thread; **does not run it.**
  - Usually, it returns quickly
  - There's **no guarantee** when a thread runs
- Threads can run in any order
  - Threads created earlier may start later.
  - They might run awhile, the OS may schedule them out awhile, and then continue to run (like processes)
- There is no function to **cleanly** terminate a thread (unlike processes, which we can kill easily)
  - If you need such functionality, you need to design the thread to handle certain signals (software interrupts)
  - More info: Blaise Barney, "POSIX Threads Programming," Lawrence Livermore National Laboratory.

Let’s do an exercise

- We want to create 10 threads
  - They are given unique IDs from 0 to 9.
  - Each of them outputs its ID.
- How would you write the program?

Let's do an exercise

```
#define THREAD_NO 10
void *mythread(void *arg) {
    int *id = (int *)arg;
    printf("my id is %d\n", *id);
}

int main(){
    pthread_t p[THREAD_NO];
    int i = 0;
    for(i=0; i<THREAD_NO; i++){
        pthread_create(&p[i], NULL,
mythread, &i);
    }

    for(i=0; i<THREAD_NO; i++){
        pthread_join(p[i], NULL);
    }
    return 0;
}
```

Is it correct?

What is wrong with it?

"mythread" may not run immediately

If one "mythread" does not start until the next iteration, "i" will be changed

Let’s do an exercise

```c
#define THREAD_NO 10
void *mythread(void *arg) {
    int *id = (int *)arg;
    printf("my id is %d\n", *id);
}

int main(){
    pthread_t p[THREAD_NO];
    int i = 0;
    for(i=0; i<THREAD_NO; i++){
        pthread_create(&p[i], NULL, 
mythread, &i);
    }

    for(i=0; i<THREAD_NO; i++){
        pthread_join(p[i], NULL);
    }
    return 0;
}
```

Solution 1

Create an array of 10 “i”s. This is fine if there are ”join” after “create”. Otherwise, the local array may be deallocated when the function returns

Defining “i” as a global array will be fine, although this is usually considered a bad practice

Let’s do an exercise

```
#define THREAD_NO 10
void *mythread(void *arg) {
    int *id = (int *)arg;
    printf("my id is %d\n", *id);
}

int main(){
    pthread_t p[THREAD_NO];
    int i = 0;
    for(i=0; i<THREAD_NO; i++){
        pthread_create(&p[i], NULL,
mythread, &i);
    }

    for(i=0; i<THREAD_NO; i++){
        pthread_join(p[i], NULL);
    }
    return 0;
}
```

Solution 2

`malloc` each “i”

Be careful where to call free.

Multi-threading programming is hard

- Reasoning is harder:
  - In a single-threaded program, when you call a function, you know the next statement will be executed after the function call finishes.
  - In a multi-threaded program, when you start a thread, the thread may not even start when the program executes the next statement.
- Testing is harder: bug is usually not deterministic.
  - How can I test it? Unfortunately, it is still an open problem. The only thing you can do is to run it multiple times.
- Multi-threading + memory allocation is even harder.

Sharing introduces new problems

- Suppose you have the following:
- Assume your program has **one global variable counter**
- Assume the program has **two threads**, each doing num=num+1
- What is your expected value of num after two threads terminate?
  - Obviously 2 right?
- But if you run the following program, you may get surprised.

```c
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");

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

Sharing introduces new problems

- Reason:
  - `counter = counter + 1` is not a single instruction when compiled

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

  - When running in one thread, no problem

```
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");

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

Sharing introduces new problems

- This run may have no problem. But can you imagine how a problem can happen? Remember that a context switch can happen at any time.

Thread 1

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Thread 2

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Sharing introduces new problems

- This run may have no problem. But can you imagine how a problem can happen? Remember that a context switch can happen at any time.

Thread 1

value = 0

```
mov 0x8049a1c, %eax
add $0x1, %eax
```

value = 1

```
mov %eax, 0x8049a1c
```

Thread 2

value = 0

```
mov 0x8049a1c, %eax
add $0x1, %eax
```

value = 1

```
mov %eax, 0x8049a1c
```

Atomicity and Synchronization

- In many cases, we hope our piece of code can act like a single atomic instruction that is never interrupted in the middle

- We will learn synchronization mechanisms to achieve atomicity

- If multiple threads access shared data without synchronization, it is called a data race
  - It is usually bad. You should avoid that.

Another problem

- Sometimes one thread needs to wait for another thread to complete something first

- This is related to but not the same as synchronization

- We will learn how to do that with semaphores and condition variables.

Things to keep in mind

- Whenever you do multi-threaded programming, you should ask yourself “do I need to do synchronization?”
  - The answer is usually YES.
  - Two exceptions: threads don’t share data; shared-data are read-only.

- Even simple statement like “a=1” may not be atomic
  - So do not assume anything is atomic
  - Always rely on proper synchronization for atomicity