Concurrency and Threads

Glenn Bruns
CSUMB

Lecture Objectives

After this lecture, you should be able to:

- Explain why concurrency is in this course
- Explain the benefits of concurrent programming
- Create a simple pthreads program

Concurrency and Operating Systems

Concurrency issues were originally “discovered” in early OS work

Example:

- What if two processes want to write to the same file?
- How does the OS protect the file from getting corrupted?
- A big problem is that interrupts can happen at any time

Benefits of concurrency

Useful programming abstraction

- an IDE can handle editing with one thread and background compilation with another

Responsiveness

- an app can handle user input while waiting for DB to initialize

Leverage multicore machines and GPUs

- run separate code on separate cores

Concurrent programming with processes

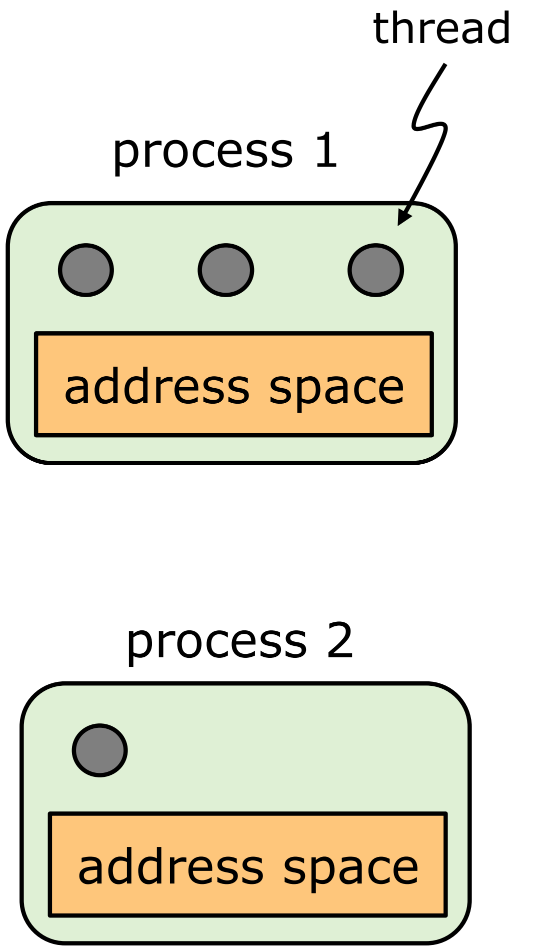
If you write a concurrent application using multiple processes, how do the processes interact?

Threads to the rescue!

Threads are lightweight processes that share virtual memory

Threads vs. processes

- Threads and processes both have state
- Threads are faster to create, destroy, and context switch
- The threads within a process <u>share a virtual address space</u>
- But, each thread has its own stack, and its own registers



C programming with threads

We'll use the C pthreads library

pthreads = "POSIX threads"

(POSIX = "portable OS interface")

Benefits:

- pthreads are widely used
- it's good to be familiar with primitives like locks and condition variables

Drawbacks:

- low level; easy to make mistakes

Some multi-threaded C code

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

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

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Compiling and running

```
$ gcc -o t0 t0.c -pthread
$ ./t0
main: begin
A
B
main: end
$ ./t0
main: begin
A
B
main: end
$ ./t0
main: begin
B
A
main: end
$
```

use –lpthread if -pthread not available

non-determinism!

pthread_create()

create a new thread

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

thread:	pointer to a pthread_t structure
attr:	thread attributes (or NULL, for defaults)
start_routine:	function to start running
arg:	arguments for start_routine (NULL if no args)

`start_routine` has input and output of type void *

```
void *my_thread(void * args) { … }
```

Thread creation example

```
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);
  ...
}
```

<u>cast</u> the argument to the right type

pthread_join()

sleep until the given thread dies (a blocking call)

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

thread: a pthread_t structure (<u>not</u> a pointer to one!)

value_ptr: pointer to the expected return value

Threads that share data

```
static volatile int counter = 0;

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;
}

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;
}
```

shared variable

each thread runs mythread()

Question: what exactly will this program output?

Compiling and running

```
$ gcc -o t1 t1.c -pthread
$ ./t1
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 13505135)
$
$ ./t1
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 11928697)
$
```

Interleaved program execution

```
for (i = 0; i < 1e7; i++) {
    counter = counter + 1;
}
```

```
mov     0x8049d50,%eax      # copy counter value into eax register
add     $0x1,%eax           # increment value in eax register
mov     %eax,0x8049d50      # copy eax register value to counter
```

thread 1	**thread 2**	
load counter into eax increment eax		
		context switch
	load counter into eax increment eax store eax as counter	
		context switch
store eax as counter		

Solution

- Use “synchronization primitives” to control when threads can run
- But OS process scheduling also wants to control the execution of threads!

Summary

- We’re now in the realm of multi-threaded programs
- Pros:
  - can leverage multi-core
  - many programs are naturally concurrent
- Cons:
  - concurrency bugs, non-determinism
- We’ll learn to write multi-threaded C, and to use pthread synchronization primitives