Lecture 7

Concurrency with Threads and Its API

Previously ..

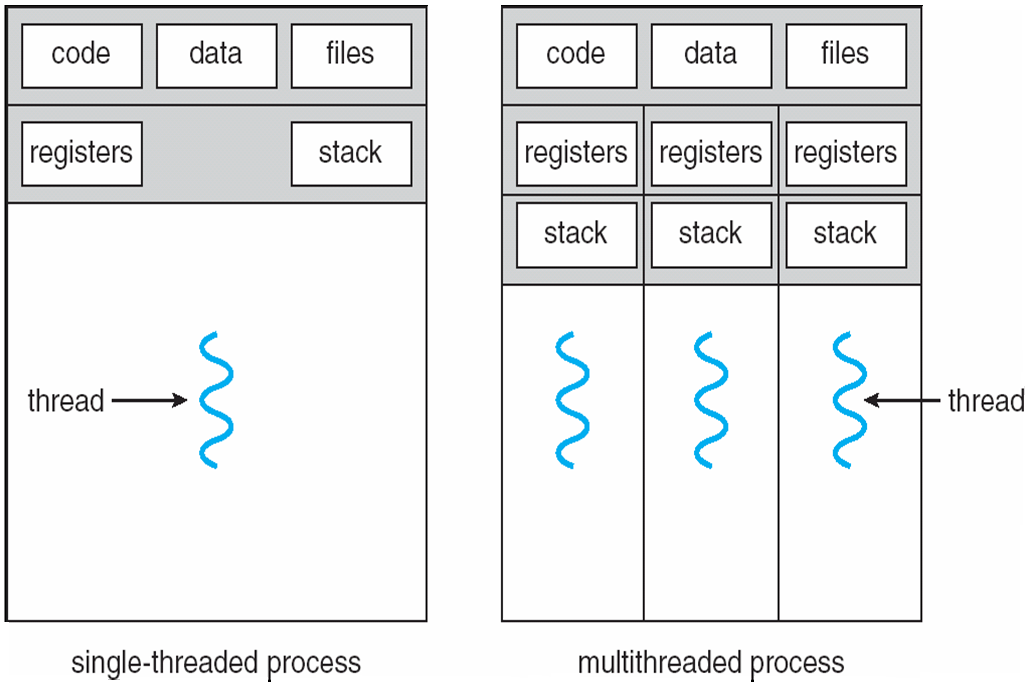
- So far we have covered the first of the main themes of this course – Virtualization
- Virtualization allows many programs or processes to run at the same time by sharing the physical resources of the machine
- We have seen how to take a single physical CPU and turn it into multiple virtual CPUs
- We have also seen how to create the illusion of a large, private virtual memory for each process

Introduction

- We now move on to the second of the main themes of this course – Concurrency
- Concurrency exists at the OS level between different processes executing at the same time
- Concurrency can also exist within a single process
  - Here we introduce a new abstraction for concurrency within a single process: the thread
  - A multi-threaded program or process has more than one point of execution
- Nowadays, with multi-core CPUs, different processes or threads may execute in parallel on different cores

Single- and Multi-threaded Processes

- In a multi-threaded process, each thread runs independently and may call various functions
- Thus, instead of a single stack within the address space, there will be one for each thread

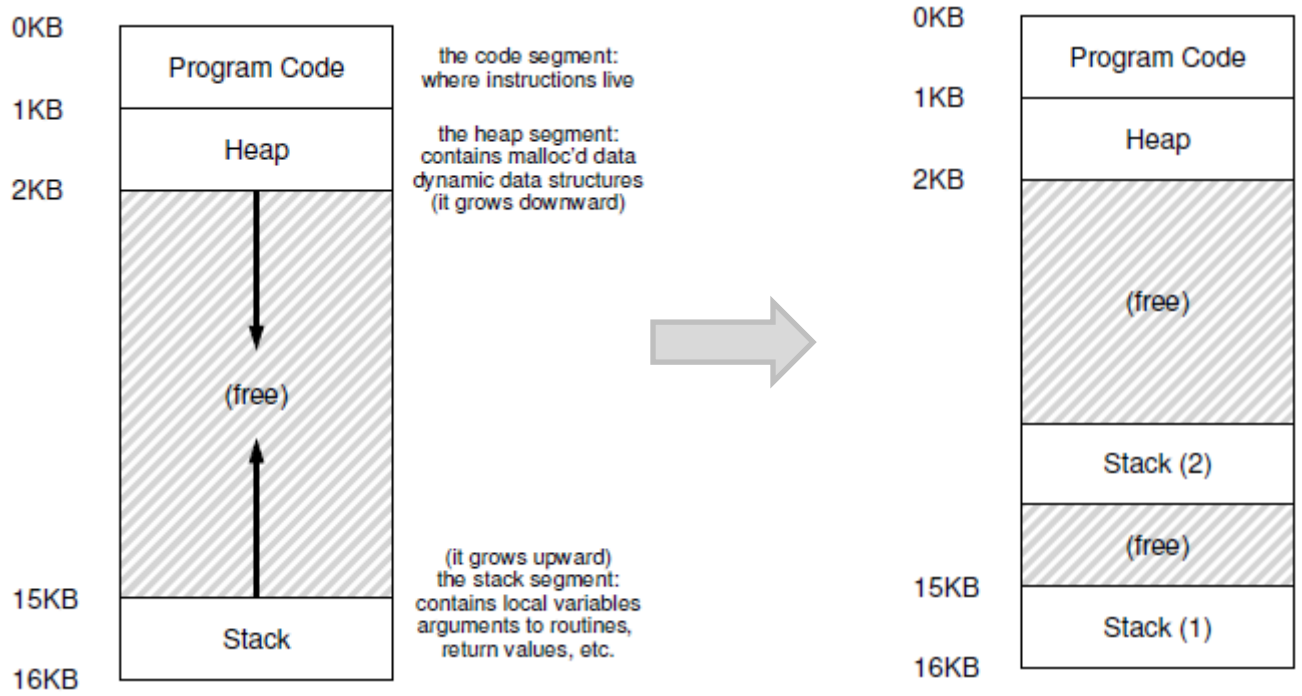


Threads

- Each thread is very much like a separate process, except that they share the same address space and thus can access the same data
- A thread has
  - A program counter (PC) that is used to fetch the next instruction to be executed by that thread
  - A private set of registers for computation
- Thus, if there are two threads on a single processor, a threads' context-switch must take place
- Even on a multi-core CPU, there are generally many more threads than there are cores

Threads' Context-Switch

- The address space remains the same (i.e. there is no need to switch the page table)
- Instead of saving state to a process control block (PCB), we need one or more thread control blocks (TCBs) to store the state of each thread



Example of Thread Creation

```
#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;
}
```

Thread control

Function to run

Argument to pass

Thread Creation

- Thread creation is similar to making a function call
- However, instead of first executing the function and then returning to the caller, the system creates a new thread of execution for the function that is being called
- The newly created threads run independently of the caller and independently of each other
- What if two threads wish to update a global shared variable?

Updating Shared Variables

```
static volatile int counter = 0;
//
// 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;
}
```

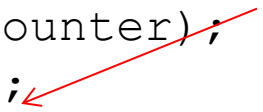
A global variable

A thread independently updates a global variable

Updating Shared Variables

```
//
// 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;
}
```

What if the two threads try to update at the same time?



Possible Output

- When we run this program, we may get the correct result

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

Even on a single processor, each time we run the program, we may get a different result

- However, in some cases, we may not get the desired result

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

Aren't computers supposed to produce **deterministic** results?

Uncontrolled Scheduling

- Accessing shared data needs careful control because of the possible interleaving of threads
  - Two threads each increment a shared variable, **counter**

Thread 1	**Thread 2**
load counter	load counter
add 1	add 1
store counter	store counter

Many possible interleavings, including:

load counter	
add 1	
	load counter
store counter	
	add 1
	store counter

Incorrect answer !!

Uncontrolled Scheduling

- Each thread when running has its own private registers
- The registers are virtualized by the threads' context-switch code that saves and restores them
- What can happen in the previous code:
  - The code to increment counter has been run twice, but counter (which starts at 0), may only be equal to 1
  - The problem lies in the private registers
- What happen here is called a *race condition*

A Race Condition

- A Race Condition occurs when the results depend on the timing execution of the code
- It is caused by context switches that occur at untimely points in the execution
- Wrong results may be produced
- And…the computation is no longer deterministic

Critical Section and Mutual Exclusion

- If multiple threads executing a code section can result in a race condition, we call this code a *critical section*
- A critical section is a piece of code that accesses a shared variable, or more generally a shared resource, and must be executed by only one thread at a time
- This code needs to have a *mutual exclusion* property
- This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so

Atomicity

- One way to guarantee mutual exclusion is to have more powerful instructions that execute in a single step, e.g. a single instruction to add a value to a memory location
- This would remove the possibility of an interrupt between actions
- When an interrupt occurs, either the instruction has not run at all, or it has run to completion – there is no in-between state
- Atomicity means "as a unit", "all or none."
- However … this is not possible for all cases, otherwise the instruction set will be slow and very large
- Synchronization primitives are thus introduced instead

We will be using synchronization primitives to turn short sequences of instructions into atomic blocks of execution

But first we look at some thread APIs

KEY CONCURRENCY TERMS: CRITICAL SECTION, RACE CONDITION, INDETERMINATE, MUTUAL EXCLUSION

These four terms are so central to concurrent code that we thought it worth while to call them out explicitly. See some of Dijkstra's early work [D65,D68] for more details.

- A **critical section** is a piece of code that accesses a *shared* resource, usually a variable or data structure.
- A **race condition** arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.
- An **indeterminate** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not **deterministic**, something we usually expect from computer systems.
- To avoid these problems, threads should use some kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

POSIX THREADS (PTHREADS) APIs

Use of Multi-threaded Model

- Example 1: A parallel program that creates threads to execute a particular task (in parallel)
  - It will likely use join to make sure all such work is completed before either exiting or moving onto the next stage of a computation
- Example 2: a multi-threaded web server might
  - create a number of worker threads
  - then use the main thread to accept requests
  - and pass the requests to the workers, indefinitely

Thread Creation

```
#include <pthread.h>
int
pthread_create(      pthread_t *            thread,
               const pthread_attr_t *       attr,
                     void *                 (*start_routine)(void*),
                     void *                 arg);
```

Function name

Type of argument to pass in

Can return anything

Can pass anything in

- thread: a pointer to a structure of type pthread_t, is used to interact with the thread
- attr: is used to specify any attributes the thread might have – if default, pass NULL
- start_routine: a function pointer that points to the function scheduled to be run by the thread
- arg: the argument to be passed to the function where the thread begins execution

Example of Creation

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

Thread Completion

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

- 1st argument: specifies which thread to wait for
- 2nd argument: a pointer to the return value
- We can pass NULL into pthread join() if we don't care about the return value

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

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = Malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

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

    myarg_t args;
    args.a = 10;
    args.b = 20;
    Pthread_create(&p, NULL, mythread, &args);
    Pthread_join(p, (void **) &m);
    printf("returned %d %d\n", m->x, m->y);
    return 0;
}
```

- Why double pointer ?
- Because the routine can return anything, it is defined to return a pointer to void
- Because the routine can change the value of the passed in argument, need to pass in a pointer to that value, not just the value itself

Locks

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

```
pthread_mutex_t lock;

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

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

Address of the lock

Optional set of lock attribute

Did not check the error code

- If no other thread holds the lock when pthread_mutex_lock() is called, the thread
  - acquires the lock
  - enter the critical section
- If another thread holds the lock, the thread is blocked until it has acquired the lock

Guidelines for Writing Multi-threaded Programs

- Keep the code simple
- Minimize thread interactions
- Initialize locks and condition variables
- Check the return codes
- Be careful with how to pass arguments to, and return values from, threads
- Each thread has its own stack
- Always use condition variables to signal between threads
- Use the manual pages

Note of pthreads Library

- Create and Join

```
pthread_create()
pthread_join()
```

- Locks

```
pthread_mutex_init()
pthread_mutex_lock()
pthread_mutex_unlock()
pthread_mutex_trylock()
pthread_mutex_timedlock()
```

- Condition Variables

```
pthread_cond_init()
pthread_cond_wait()
pthread_cond_signal()
pthread_cond_broadcast()
```

Other Lock Routines

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            struct timespec *abs_timeout);
```

- These two routines are used in lock acquisition
- The trylock version returns a failure if the lock is already held
- The timedlock version returns after acquiring the lock, or after a timeout, whichever happens first
  - The routine returns a failure if a timeout occurs

These two routines are sometimes useful and will be discussed in a later lecture

Condition Variables

- Condition variables are useful when some kind of signaling must take place between threads
  - int pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex);
  - This puts the calling thread to sleep, and waits for some other thread to signal it (wake it)
  - Must have a lock that is associated with the condition (this lock must be held when the wait is called and is released by the call)
- int pthread_cond_signal(pthread_cond_t *cond);
  - Wakes up a thread waiting on the condition variable
  - When signaling , we must also make sure that we hold the lock

Condition Variables will be discussed in more detail later

To Compile and Run

- Must include the header ‘pthread.h’ in the code
- On the link line, must also explicitly link with the pthreads library, by adding the ‘-lpthread’ flag
- gcc -o main main.c -Wall -lpthread