Threads
Agenda

• Concurrency: An Introduction
• Threads API
• Quiz
Concurrency: An Introduction
Question

• What are threads? How are they useful?

• Name a few POSIX library functions for threads
Threads

• Thread
  – An abstraction for a process
  – A point of execution in the process (Program Counter)
  – Multiple points of execution (each thread has a separate PC)
    ➢ A thread may run over a different part of a program.

• Word Processor
  – A thread for displaying graphics
  – Another thread for responding to keystrokes
  – Another task for grammar checking
Threads

• All threads share the same address space.
  – Share code and heap
  – Global variables: threads communicate together.

• Each thread has a private set of registers.
  – E.g. PC

• Context switch occurs between two threads.
  – The state of each thread is saved/restored in/from thread control block (TCB)
  – A PCB consists of TCBs that are linked together.
Threads

- Each thread has a different stack.
  - Variables, parameters, and return values are stored in a thread-local storage.
Why Use Threads?

• Parallelism
  – Run a program on multiple CPUs
    ➢ One thread per CPU
    ➢ Speed-up computations
  – E.g., increment each element of an array by some amount

• Avoid blocking program progress
  – E.g., Disk I/O, page fault
  – One thread waits for I/O while another thread uses CPU to do some task.
  – Overlap of I/O with other tasks within a program
Example

```c
#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;
}
```

`pthread_create()` creates two threads
Each thread runs `mythread()`
`pthread_join()`: the main thread waits for each thread to complete.
Example

<table>
<thead>
<tr>
<th>main</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>starts running</td>
<td>runs</td>
<td>runs</td>
</tr>
<tr>
<td>prints “main: begin”</td>
<td>prints “A”</td>
<td>prints “B”</td>
</tr>
<tr>
<td>creates Thread 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>creates Thread 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>waits for T1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>runs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>prints “main: end”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>returns</td>
<td></td>
</tr>
</tbody>
</table>

Threads are run after the main thread calls `pthread_join()`
Example

<table>
<thead>
<tr>
<th>main</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>starts running</td>
<td>runs</td>
<td>runs</td>
</tr>
<tr>
<td>prints “main: begin”</td>
<td>prints “A”</td>
<td>prints “B”</td>
</tr>
<tr>
<td>creates Thread 1</td>
<td>returns</td>
<td>returns</td>
</tr>
<tr>
<td>creates Thread 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>waits for T1</td>
<td>returns immediately; T1 is done</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>waits for T2</td>
<td>returns immediately; T2 is done</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Threads are run right after the main thread calls `pthread_create()`
Example

<table>
<thead>
<tr>
<th>main</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>starts running</td>
<td></td>
<td>runs</td>
</tr>
<tr>
<td>prints “main: begin”</td>
<td></td>
<td>prints “B”</td>
</tr>
<tr>
<td>creates Thread 1</td>
<td></td>
<td>returns</td>
</tr>
<tr>
<td>creates Thread 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>waits for T1</td>
<td></td>
<td>runs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>prints “A”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>returns</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>waits for T2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>returns immediately; T2 is done</td>
<td></td>
<td></td>
</tr>
<tr>
<td>prints “main: end”</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Threads do not have to run in the same order that they were created.
Example

- Thread creation is like a function call.
  - Creates a thread of execution for *mythread*
  - Runs independent of caller (pthread_create)
  - Scheduler decides when to run the thread.
Shared Data

#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;
void *
mythread(void *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;
}

Pthread_create() and Pthread_join() wrap pthread_create and pthread_join().

Increment counter 10 million times. The desired result in 20,000.
Shared Data

main: begin (counter = 0)
A: begin
B: begin
A: end
B: end
main: done with both (counter = 20000000)

main: begin (counter = 0)
A: begin
B: begin
A: end
B: end
main: done with both (counter = 19345221)
Shared Data

100 mov 0x8049a1c, %eax
105 add $0x1, %eax
108 mov %eax, 0x8049a1c

Critical section: a piece of code that accesses a shared variable. Race condition: multiple threads enter the critical section at the same time (results are indeterminate). Mutual exclusion: only one thread executes the critical section.
Atomicity

• A super instruction that executes the code sequence in a single step (atomically).
  
  memory-add 0x8049a1c $0x1
  – Three sub instructions are run as an uninterruptable unit.

• Synchronization primitives
  – Access critical section in a controlled (synchronized) way
Threads API
Thread Creation

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

- **thread**: pointer to struct `pthread_t`  
- **attr**: attributes of a thread e.g., stack size or priority  
- **start_routine**: a function that a thread starts running  
- **arg**: argument passed to `start_routine`
```c
#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)

thread: thread to wait for
value_ptr: pointer to the return value of function
Thread Completion

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

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); // this thread has been
    // waiting inside of the
    // pthread_join() routine.
    printf("returned %d %d\n", m->x, m->y);
    return 0;
}
```
Note

Never return a pointer to memory space that is deallocated

```c
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; // ALLOCATED ON STACK: BAD!
5           r.x = 1;
6           r.y = 2;
7       return (void *) &r;
8   }
```
Locks

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

mutex: lock used by a thread to enter a critical section

Only when a thread **acquires** the lock, `pthread_mutex_lock` returns.

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER; or
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0);
```

```c
int rc = pthread_mutex_lock(&lock);
assert( rc == 0);
x = x + 1;
pthread_mutex_unlock(&lock);
```
Condition Variables

- Signal a calling thread when the value of a shared variable changes in a program

```c
int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t * mutex);

int pthread_cond_signal(pthread_cond_t* cond);

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

Waiting thread
Pthread_mutex_lock(&lock);
while(ready == 0) {
    Pthread_cond_wait(&cond, &lock);
}
Pthread_mutex_unlock(&lock);

Signaling thread
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```