Multi-Threaded Programming

- You have a set of 6 tasks you have to do:
  - do your homework
  - clean up your room
  - do the shopping
  - pick the apples in the garden
  - wash the dishes
  - sweep the porch
- Each task takes 1 hour to do.
- If you have to do all the work by yourself, it will take you 6 hours (If you start at 5:00, all tasks will be completed at 11:00).
Multi-Threaded Programming

- If you find two friends and divide the work and you all start working at the same time, you will finish all the tasks sooner.
  - All 3 of you will do 2 tasks each, which will take 2 hours (If you all start at 5:00, all tasks will be completed at 7:00! Plenty of time to do even more homework!).

![Diagram]

5:00pm to 7:00pm

- Homework to Clean Room
- Shop to Pick Apples
- Do Dishes to Sweep Porch
Threads

- **Threads** (an abbreviation of threads of control) are how we can get more than one thing to happen at once in a program. A thread has a minimum of internal state and a minimum of allocated resources.

- A thread (a.k.a. lightweight process) is associated with a particular process (a.k.a. heavyweight process, a retronym). A heavyweight process may have several threads and a thread scheduler. The thread scheduler may be in the user or in the system domain.

- First, we need some background on the Linux/UNIX process model.
The Linux/UNIX Process model

- **Text**
  - Program Binary
  - 0x00000000

- **Data**
  - Global/Static variables
  - Dynamically allocated variables

- **Heap**
  - Local variables, function/method arguments
  - 0xFFFFFFFF

- **Stack**
  - 0xFFFFFFFF

Where is Neo? Where is Morpheus? Somewhere in the Matrix. Based on the following code, determine in which segment are the specified variables (on the next slide) allocated.

```c
#define BODY_BIT_SIZE 1000000
int A[BODY_BIT_SIZE];
extern void transfer();

void booth(char *xyz)
{
    int i;
    static int neo[BODY_BIT_SIZE];
    int *morpheus = (int *) malloc(sizeof(int)*BODY_BIT_SIZE);
    for (i=0; i<BODY_BIT_SIZE; i++)
        morpheus[i] = neo[i];
    morpheus[0] = xyz;
    transfer();
}

int main(int argc, char *argv[])
{
    char *xyz = (char *) malloc(sizeof(char)*BODY_BIT_SIZE);
    printf("Hello?\n"); scanf("%s", xyz)
    booth(xyz);
}
```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Data</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>A[100]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>morpheus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>morpheus[0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>neo[10]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>argc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>xyz (in booth(...))</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thread model

- Threads exist within a process and use the processes resources.
- Threads share the text, data and heap segments. Each thread has its own stack and status.
- Allows threads to be scheduled by the OS and run as independent entities.
Why do we need threads?

*Life is asynchronous.*

Asynchronous means things can happen independently unless there’s some enforced dependency.

Some examples of where threads are useful:

- Windowing systems
- GUI applications
- Web browsers
- Database servers
- Web servers
Why do we need threads? (2)

- Light weight (compared to processes)
- Efficient data exchange (shared resources)
- Overlapping CPU work with I/O
  - For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
- Priority/real-time scheduling. Tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
- Asynchronous event handling. Tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.
Threaded Programs

In general, in order for a program to take advantage of threads, it must be able to be organized into discrete, independent tasks which can execute concurrently. For example, if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.
Pthreads: POSIX Threads Library

A standardized threads library that is the native threads library under Linux. Contains around a hundred functions. We will examine a few core functions.

- **pthread_create()**: Creates and starts running a new thread from the specified start function.
- **pthread_exit()**: Terminates the calling thread.
- **pthread_join()**: Wait for the specified thread to finish.
- **pthread_self()**: Prints the thread id of the calling thread.
A Multithreaded "Hello World"

```
#include <pthread.h>
void print_message_function(void *ptr);
int main(int argc, char **argv)
{
    pthread_t thread1, thread2;
    char *message1 = "Goodbye";
    char *message2 = "World";

    pthread_create(&thread1, NULL, print_message_function, (void*) message1);
    pthread_create(&thread2, NULL, print_message_function, (void*) message2);
    exit(0);
}
void *print_message_function(void *ptr)
{
    printf("%s ", (char *)message);
}
```

lab/threads/thread-hello-world.c
Better Multithreaded "Hello World"

```c
#include <pthread.h>
void print_message_function(void *ptr);
int main(int argc, char **argv) {
  pthread_t thread1, thread2;
  char *message1 = "Goodbye";
  char *message2 = "World";

  pthread_create(&thread1, NULL, print_message_function, (void*) message1);
  pthread_create(&thread2, NULL, print_message_function, (void*) message2);

  pthread_join(thread2, NULL); /* wait for thread2 to finish */
  pthread_join(thread1, NULL); /* wait for thread1 to finish */

  exit(0);
}
void *print_message_function(void *ptr) {
  printf("%s \n", (char *)message);
  pthread_exit(NULL);
}
lab/threads/thread-better-hello-world.c
```
A Test of Thread Creation

#include "appropriate header files"

void *run(void *);
#define MAX 10000

int main(int argc, char *argv[])
{
    pthread_t tids[MAX];
    int i, status, count=0;

    for (i=0; i<MAX; i++) {
        status = pthread_create(&tids[i], NULL, run, (void*) NULL);
        if (status != 0) {
            perror(thread-test);
            break;
        }
        printf("Created thread number %d \n",i);
        count++;
    }
    for(;;) sleep(40);
    exit(0);
}

void *run(void *arg)
{
    printf("This is thread id = %d\n",pthread_self());
    sleep(30);
    pthread_exit(0);
}

lab/threads/thread-test.c

Use the command ps xm to see all threads and processes. Or use the KDE system guard ksysguard to monitor processes and number of threads.
Thread Ids

```c
/* lab/threads/thread-ids.c */
/* appropriate header files */
void *run(void *ptr);

int main(int argc, char **argv)
{
    int i, n;
    int *value;
    pthread_t *tid;

    n = atoi(argv[1]);
    tid = (pthread_t *) malloc(sizeof(pthread_t) * n);
    for (i=0; i<n; i++) {
        value = (int *) malloc(sizeof(int));
        *value = i;
        pthread_create(&tid[i], NULL, run, (void *) value);
    }

    for (i=0; i<n; i++)
        pthread_join(tid[i], NULL);

    exit(EXIT_SUCCESS);
}

void *run(void *ptr)
{
    printf("I am thread %d with thread id %X\n", *(int *)ptr, pthread_self());
    pthread_exit(NULL);
}
```
Other Multithreaded Examples

- **lab/threads/thread-sum.c** A multithreaded parallel sum program.

- **lab/threads/bad-bank-balance.c** Illustrates race conditions when multiple threads access the same global variable.

<table>
<thead>
<tr>
<th></th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read balance: $1000</td>
<td>$1000</td>
<td>Read balance: $1000</td>
<td>$1000</td>
</tr>
<tr>
<td>Deposit $200</td>
<td>$1000</td>
<td>Deposit $200</td>
<td>$1000</td>
</tr>
<tr>
<td>Update balance $1000+$200</td>
<td>$1200</td>
<td>Update balance $1000+$200</td>
<td>$1200</td>
</tr>
</tbody>
</table>

- **lab/threads/safe-bank-balance.c** Shows a solution to the race condition using a mutex. Mutexes will be studied in depth in the Operating Systems class.
In-class exercise

**Evil in the Garden of Threads?** The following code shows the usage of two threads to sum up a large array of integers in parallel.

```c
/* appropriate header files */
void *partial_sum(void *ptr);
int *values, n;
int result[2]; /* partial sums arrays */

int main(int argc, char **argv) {
  int i;
pthread_t thread1, thread2;
  if (argc != 2) { fprintf(stderr, "Usage: %s <n> \n", argv[0]); exit(1); }
  n = atoi(argv[1]);
  values = (int *) malloc(sizeof(int)*n);
  for (i=0; i<n; i++)
    values[i] = 1;
  pthread_create(&thread1, NULL, partial_sum, (void *) "1");
  pthread_create(&thread2, NULL, partial_sum, (void *) "2");
  do_s om_e other _computing_for_a_while();
  printf("Total sum = %d \n", result[0] + result[1]);
  exit(0);
}

void *partial_sum(void *ptr) {
  /* same as the example earlier */
}
```

Choose the statements that best explains how the code runs.

1. The code always adds the values array correctly.
2. The code never adds the values array correctly.
3. The code sometimes adds the values array correctly.
4. The code will corrupt the values array because of a race condition.
Multithreading support in GDB

The gdb debugger provides these facilities for debugging multi-thread programs:

- Automatic notification of new threads.
- `thread threadno`, a command to switch among threads
- `info threads`, a command to inquire about existing threads
- `thread apply [threadno] [all] args`, a command to apply a command to a list of threads
- Thread-specific breakpoints

See GDB manual for more details.
Threads are part of the Java language. There are two ways to create a new thread of execution.

- Declare a class to be a subclass of `Thread`. This subclass should override the run method of class `Thread`. An instance of the subclass can then be allocated and started.
- The other way to create a thread is to declare a class that implements the `Runnable` interface. That class then implements the `run` method. An instance of the class can then be allocated, passed as an argument when creating `Thread`, and started.
class Grape extends Thread {
    Grape(String s) {super(s);} //constructor

    public void run() {
        for (int i=0; i<10; i++) {
            System.out.println("This is the " +
                             this.getName() + " thread.");
            this.yield();
        }
    }
}

public class threads1 {
    public static void main (String args[]) {
        new Grape("merlot").start();
        new Grape("pinot").start();
        new Grape("cabernet").start();
    }
}
Another Thread example in Java

```java
/* threads/RunnableExample.java */
class Grape implements Runnable {
    private String name;
    Grape(String s) {name = s;}
    public String getName() {return name;}
    public void run() {
        for (int i=0; i<10000; i++) {
            System.out.println("This is the " + this.getName() + " thread.");
        }
    }
}

public class RunnableExample {
    public static void main (String args[]) {
        Grape g1 = new Grape("merlot");
        Grape g2 = new Grape("pinot");
        Grape g3 = new Grape("cabernet");
        new Thread(g1).start();
        new Thread(g2).start();
        new Thread(g3).start();
    }
}
```
// See how many threads can be created,
// or, how big a quagmire we can create.
// threads/MaxThreads.java

public class MaxThreads {
    final static int MAX = 50;
    public static void main (String args[])
        throws InterruptedException
    {
        for(int i=0; i<MAX; i++) {
            Integer I = new Integer(i);
            new nuts(I.toString()).start();
        }
        Thread.sleep(20000);
    }
}

class nuts extends Thread {
    nuts(String s) {super(s);} //constructor
    public void run() {
        System.out.println("Thread number "+this.getName());
        try {
            Thread.sleep(20000); //in millisecs
        } catch (InterruptedException e) {
            System.err.println(e);
        }
    }
}
In-class Exercise

Given the following classes:

```java
class Worker extends Thread {...}
class BusyBee implements Runnable {...}
```

Which of the following statements (one or more) correctly creates and starts running a thread?

1. `new Worker();`
2. `new Worker().start();`
3. `new Thread(new BusyBee()).start();`
4. `new Thread(new BusyBee());`
In-class Exercise

Let’s get together and compute? Consider the following code and choose the statement that best explains how the code runs.

```java
public class ComputeALot implements Runnable {
    public void run() {
        /* ... */
    }
    public static void main (String args[]) {
        ComputeALot playground = new ComputeALot();
        Thread [] tid = new Thread[4];
        for (int i=0; i < tid.length; i++)
            tid[i] = new Thread(playground);
    }
}
```

1. The code creates and runs four threads that all run the method `run`
2. The code creates four threads but they don’t do anything.
3. The code creates three threads but they don’t do anything.
4. The code creates and runs three threads that all run the method named `run`

See the `Object` class for synchronization methods.

A collection of threads that work together is known as a thread pool. For automatic management of thread pools, see: `Executor` interface from `java.util.concurrent`.

Controlling Threads

- start()
- stop(), suspend() and resume() Note: These have been deprecated in the current version of java
- sleep().
- interrupt(): wake up a thread that is sleeping or blocked on a long I/O operation
- join(): causes the caller to block until the thread dies or with an argument (in millisecs) causes a caller to wait to see if a thread has died
public class InterruptTest implements Runnable {

public static void main( String[] args ) throws Exception {
    Thread sleepyThread = new Thread( new InterruptTest() );
    sleepyThread.setName("SleepyThread");
    sleepyThread.start();
    // now we two threads running, the main thread and the sleepy thread,
    // which goes to sleep after printing a message.
    Thread.sleep(500);// put main thread to sleep for a while
    sleepyThread.interrupt();// interrupt sleepyThread's beauty sleep
    Thread.sleep(500);// put main thread to sleep for a while
    sleepyThread.interrupt();// interrupt sleepyThread's beauty sleep
}

public void run() {
    Thread me = Thread.currentThread();
    while (true) {
        try {
            System.out.println(me.getName() + " : sleeping...");
            Thread.sleep(5*1000); // in millisecs
        } catch (InterruptedException e) {
            System.out.println(me.getName() +" : argh! let me sleep #$@!";
        }
    }
}
}
A thread continues to execute until one of the following thing happens.

- it returns from its target `run()` method.
- it’s interrupted by an uncaught exception.
- it’s `stop()` method is called.
Daemon Threads

- Useful for simple, periodic tasks in an application.
- The `setDaemon()` method marks a thread as a daemon thread that should be killed and discarded when no other application threads remain.

```java
class Devil extends Thread {
    Devil() {
        setDaemon(true);
        start();
    }
    public void run() {
        //perform evil tasks
        ...
    }
}
```