Three "bailing" programmers[1]

Three programmers go out on a boating excursion.

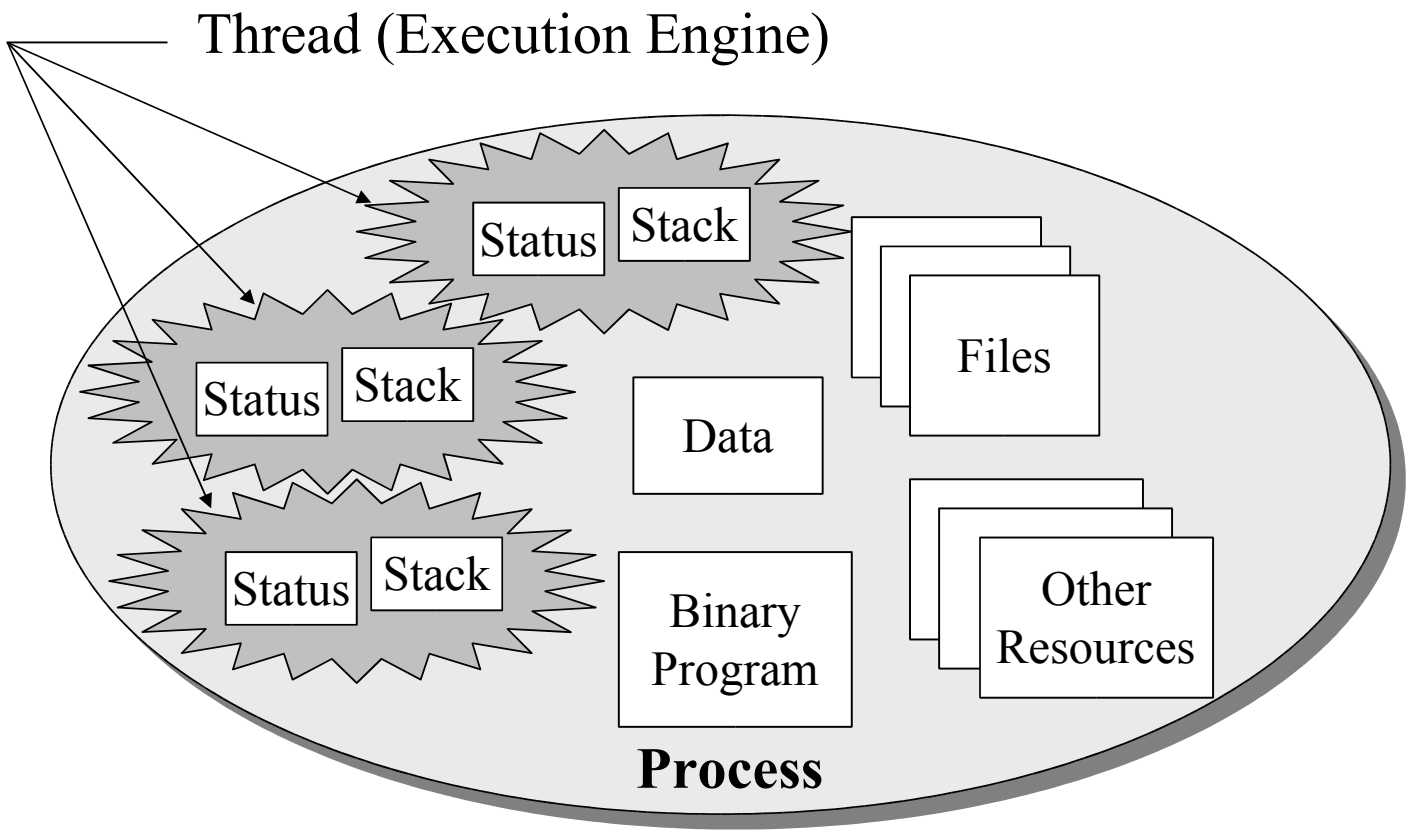
A dark storm, violent waves, broken mast, lost sail and leaking boat.

A boat, a set of oars, a bailing bucket, food and water.

[1] Taken from *Programming with POSIX Threads* by David R. Butenhof.

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.
- Threads share the text, data and heap segments. Each thread has its own stack and status.


Thread (Execution Engine)
Status
Stack
Status
Stack
Status
Stack
Files
Data
Other
Resources
Binary
Program
Process

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

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"

```
#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 ", (char *)message);
   pthread_exit(NULL);
}
```

lab/threads/thread-better-hello-world.c

A Test of Thread Creation

```
/* 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

```
/* 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.
- `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.

```
/* 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_some_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 in Java

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.

Thread example in Java

```
/* files-processes/ThreadExample.java */
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

```
/* 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();
    }
}
```

Thread Test example in Java

```
// 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:

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

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

1. `Thread t1 = new Worker();`
2. `Thread t1 = new Worker().start();`
3. `Thread t1 = new Thread(new BusyBee()).start();`
4. `Thread t1 = 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.

```
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`

Relevant Java Classes/Interfaces

- See documentation for basic classes: `java.lang.Thread`, `java.lang.ThreadGroup` and `java.lang.Runnable` interface.
- 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

Thread Interrupt Example

```
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's Life

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.

What happens if the `run()` method never terminates, and the application that started the thread never calls the `stop()` method? *The thread remains alive even after the application has finished!* (so the Java interpreter has to keep on running...)

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.

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

Thread Synchronization (1)

- Java threads are preemptible. Java threads may or may not be time-sliced. The programmer should not make any timing assumptions.
- Threads have priorities that can be changed (increased or decreased). An application cannot usurp resources from another application since all threads operate within one process.
- This implies that multiple threads will have race conditions (read/write conflicts based on time of access) when they run. The programmer has to resolve these conflicts.
- Example of a race condition: `Account.java`, `TestAccount.java`
- Another example of a race condition: `PingPong.java`

Thread Synchronization (2)

- Java has `synchronized` keyword for guaranteeing mutually exclusive access to a method or a block of code. Only one thread can be active among all synchronized methods and synchronized blocks of code in a class.

```
// Only one thread can execute the update method at a time in the class.
synchronized void update() { //... }

// Access to individual datum can also be synchronized.
// The object buffer can be used in several classes, implying
// synchronization among methods from different classes.

synchronized(buffer) {
    this.value = buffer.getValue();
    this.count = buffer.length();
}
```

- Every Java object has an implicit monitor associated with it to implement the synchronized keyword. Inner class has a separate monitor than the containing outer class.
- Java allows Rentrant Synchronization, that is, a thread can reacquire a lock it already owns. For example, a synchronized method can call another synchronized method.

Synchronization Example 1

- Race conditions: `Account.java`, `TestAccount.java`
- Thread safe version using `synchronized` keyword: `RentrantAccount.java`

Thread Synchronization (3)

- The `wait()` and `notify()` methods (of the `Object` class) allow a thread to give up its hold on a lock at an arbitrary point, and then wait for another thread to give it back before continuing.
- Another thread must call `notify()` for the waiting thread to wakeup. If there are other threads around, then there is no guarantee that the waiting thread gets the lock next. *Starvation* is a possibility. We can use an overloaded version of `wait()` that has a timeout.
- The method `notifyAll()` wakes up all waiting threads instead of just one waiting thread.

Example with `wait()`/`notify()`

```
class MyThing {
  synchronized void waiterMethod() {
    // do something
    // now we need to wait for the notifier to do something
    // this gives up the lock and puts the calling thread to sleep
    wait();
    // continue where we left off
  }

  synchronized void notifierMethod() {
    // do something
    // notifier the waiter that we've done it
    notify();
    //do more things
  }

  synchronized void relatedMethod() {
    // do some related stuff
  }
}
```

In-class Exercise

Trouble in the playground? What happens when we run the following code multiple times? The `ArrayList` class isn't synchronized.

```
public class PlayString implements Runnable  {
    private ArrayList<String> list = new ArrayList();
    private String s = "strrrrring";
    public void run() {
        for (int i = 0; i < 100; i++) {
            list.add(s);
        }
    }
    public static void main (String args[]) {
        PlayString playground = new PlayString();
        Thread [] tid = new Thread[4];
        for (int i = 0; i < tid.length; i++)
            tid[i] = new Thread(playground).start();

        for (int i = 0; i < tid.length; i++)
            tid[i].join();
    }
}
```

1. The list has *four hundred* references to the String object in it every time.
2. The list has *less than or equal to four hundred* references to the String object in it.
3. The list can have *less then, equal to, or greater than four hundred* references to the String object.
4. None of these is correct.

Synchronized Ping Pong using `wait()`/`notify()`

See example `threads/SynchronizedPingPong.java`

MS Windows API for Threads

In MS Windows, the system call interface is not documented. Instead the MS Windows API is documented, which helps with being able to run programs portably across multiple versions of the MS Windows operating systems.

Creating a thread gives you a *handle* that is used to refer to the actual object that represents a thread.

- CreateThread(...). Create a new thread and start running the start function specified in the new thread.
- ExitThread(...), GetExitCodeThread(...), TerminateThread(...), GetCurrentThreadId(), GetCurrentThread().
- WaitForSingleObject(...), WaitForMultipleObjects(...). These can be used to wait for either a process or a thread.

Get detailed information from `http://msdn.microsoft.com/library/`

MS Windows API for Threads

```
HANDLE WINAPI CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
//prototype for a thread start method
DWORD WINAPI ThreadProc(
  LPVOID lpParameter
);

DWORD WINAPI GetCurrentThreadId(void);
HANDLE WINAPI GetCurrentThread(void);

VOID WINAPI ExitThread(
  DWORD dwExitCode
);
BOOL WINAPI TerminateThread(
  HANDLE hThread,
  DWORD dwExitCode
);
```

Compiling Multithreaded Programs in Visual Studio

- The `/MT` or `/MTd` flags to the compiler in Visual Studio enable multi-threaded behavior. These are turned on by default in Visual Studio 2005 onward.
- Go to the project properties. In the *Property Pages* dialog box, click the *C/C++* folder. Select the *Code Generation* page. From the *Runtime Library* drop-down box, select the appropriate Multi-threaded option (it should already be the default). Click *OK*.
- See the following page for details on the various multi-threading and related flags for the C/C++ compiler: `http://msdn.microsoft.com/en-us/library/2kzt1wy3(v=vs.110).aspx`

MS Windows API Examples

- `lab/ms-windows/threads/thread-hello-world.c`
- `lab/ms-windows/threads/thread-scheduling.c`
- `lab/ms-windows/threads/thread-test.c`
- and others in the ms-windows/files-processes examples folder....

Exercises

1. **Thread Dance**. Convert the program in `lab/threads/random` to use multiple threads and measure how much speedup you get relative to the number of CPUs on the system for generating the given number of random values. Also experiment with increasing the number of threads to be higher than the number of CPUs. Allow the user to specify the number of threads via a command line argument as follows:
   `random <numberOfRandoms> <numThreads>`
2. **The fault in our threads**. Modify the `TestList.c` driver program from your linked list project so that it creates multiple threads that run random tests on the same linked list. Also modify the driver program to accept an additional command line argument to specify the number of threads. Since all threads are sharing the same list without any protection, expect many segmentation faults due to race conditions. The purpose of this assignment is to explore race conditions in multi-threaded code.
3. **Safety in the mosh pit!** Study the `lab/threads/safe-bank-balance.c` example to see the use of Pthread mutexes. Try to implement the same idea to protect your linked list from the previous assignment. Can you get it to work safely? Can you prove that it is safe? These topics will be covered in much more depth in the Operating Systems class.
4. **Multithreaded Chat Server**. Write a multithreaded server that can chat with multiple clients simultaneously. Also write a simple client program to test the server. Use named pipes for the communication between clients and servers.