Threads

- In a multithreaded program, **all threads execute the same piece of code**. They **share the heap**, but each thread **has its own stack frame**.
- **main() function** comprise a **single, default thread**. Threads other than the default one can be **created by programmers**.

Threads in C

- To create a multithreaded C program, you need to know these basic:
  - First, include the header file **<pthread.h>**
  - Second, a **worker function** should be defined. A worker function is **a C routine that the thread will execute once it is created**.

```c
#include <pthread.h>

void *foo (void *args) ()
{
}

pthread_attr_t attr;

pthread_attr_init (attr);

pthread_t thread;

pthread_create (&thread, &attr, foo, arg);

pthread_join (thread, status); //suspend execution of the calling thread until the target thread terminates

pthread_exit (status); // terminates the calling thread

Compiling: using -pthread
```

- **pthread** (POSIX thread): threads that use the POSIX standard programming interface
  - #include <pthread.h>
  - Define a worker function: a C routine that the thread will execute once it is created
    - void *foo (void *args) () {}
  - Initialize pthread_attr_t: you can use NULL for the default values
    - pthread_attr_t attr;
    - pthread_attr_init (attr);
  - Create a thread
    - pthread_t thread;
    - pthread_create (&thread, &attr, foo, arg);
  - Thread management
    - pthread_join (thread, status); //suspend execution of the calling thread until the target thread terminates
    - pthread_exit (status); // terminates the calling thread
  - Compiling: using -pthread

- Third, the thread attribute should be initialized. You can use **NULL for the default values**.(attr: scope, joinable, size and etc.)
- Now you can create a thread by using the pthread_create function.
- After create threads, you can use pthread_join to suspend execution of the calling thread until the target thread terminates.
- Or you can use pthread_exit to terminates the calling thread.
- When compiling, remember to include the pthread library. You may no need this on Mac. If you use other machines, double check if the library path has been set already.
Show the `helloThreads.c`, and run the code. Comment out the join for loop, run the program again, and ask students why the results are different. [pthread_join will suspends the calling thread (main thread) until the target thread terminates.]

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

#define NUM_THREADS 5

typedef struct {
    int id;
} threadInfo;

void *hello_thread(void *threadinfo) {
    threadInfo *ti = (threadInfo *) threadinfo;
    printf("Thread %d saying Hello!\n", ti->id);
    pthread_exit(NULL);
}

int main () {
    int i;
    threadInfo ti[NUM_THREADS];
    pthread_t thread[NUM_THREADS];

    // Set up the parameters for each thread
    for (i = 0; i < NUM_THREADS; i++)
        ti[i].id = i;

    // Get the threads going
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&(thread[i]), NULL, hello_thread, &(ti[i]));

    // Join up with them. This will wait until they are done.
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(thread[i], NULL);

    return 0;
}
```

- with join for loop, the output is several “saying Hello!”
- without join for loop, the output is empty.
Synchronization

- Show students the `incrementb.c`, go through the code, and ask them the result? [The result may not equal to the number of threads]
  - Code REF: http://randu.org/tutorials/threads/
  - The code expects each thread increments the counter by one, and the final result should be the number of threads.
  - counter is a global variable, all threads share the same piece of memory.

```c
/**
 * incrementb.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#define NUM_THREADS 10
/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
} thread_data_t;
/* shared data between threads */
int shared_x;
void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;
  printf("hello from thr_func, thread id: %d\n", data->tid);
  sleep(1);
  shared_x++;
  printf("x = %d\n", shared_x);
  pthread_exit(NULL);
}
int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS];
  int i, rc;
  /* create a thread_data_t argument array */
  thread_data_t thr_data[NUM_THREADS];
  /* initialize shared data */
  shared_x = 0;
  /* create threads */
  for (i = 0; i < NUM_THREADS; ++i) {
    thr_data[i].tid = i;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }
  /* block until all threads complete */
  for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thr[i], NULL);
  }
  return EXIT_SUCCESS;
}
```
- Why we get incorrect results?
  - Race condition happens. Multi-threads try to read and write to the shared memory in an unsynchronized way.

- How do we address the races?
  - If a thread has to execute multiple atomic instructions on a shared variable, it must lock out other thread until it is done with its critical section.
  - The section of program where a thread read or write a shared variable are called critical sections.
  - What is the critical section in the sample code? [counter++, printf]
  - We call the solution, synchronization.