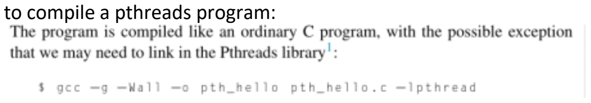
Hello, world

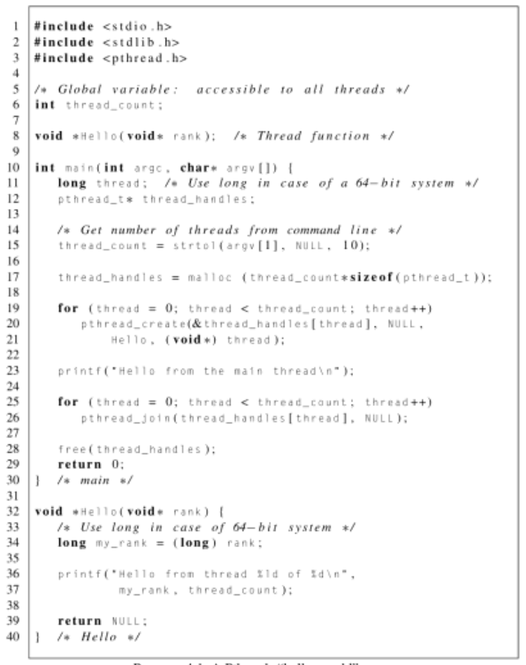
We will review a simple Hello World program using the C language and pthreads library

How to execute a pthreads program:

- How to compile a pthreads program:
  - 
    The program is compiled like an ordinary C program, with the possible exception that we may need to link in the Pthreads library[1]:
    ```
    $ gcc -g -Wall -o pth_hello pth_hello.c -lpthread
    ```
    - The -l pthread links the pthreads library

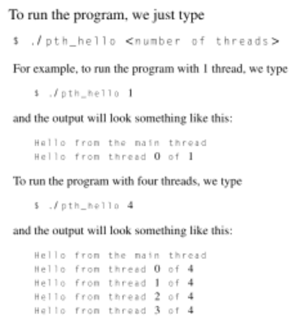
- How to run/execute a pthreads program once compiled:
  - ./[program name][number of threads you want]
    - p.s. it is compiled like a regular c file only difference is you need to include number of threads you want

Hello World Pthreads Program:



```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <pthread.h>
 4
 5 /* Global variable:  accessible to all threads */
 6 int thread_count;
 7
 8 void *Hello(void* rank);  /* Thread function */
 9
10 int main(int argc, char* argv[]) {
11    long thread;  /* Use long in case of a 64-bit system */
12    pthread_t* thread_handles;
13
14    /* Get number of threads from command line */
15    thread_count = strtol(argv[1], NULL, 10);
16
17    thread_handles = malloc (thread_count*sizeof(pthread_t));
18
19    for (thread = 0; thread < thread_count; thread++)
20       pthread_create(&thread_handles[thread], NULL,
21           Hello, (void*) thread);
22
23    printf("Hello from the main thread\n");
24
25    for (thread = 0; thread < thread_count; thread++)
26       pthread_join(thread_handles[thread], NULL);
27
28    free(thread_handles);
29    return 0;
30 }  /* main */
31
32 void *Hello(void* rank) {
33    /* Use long in case of 64-bit system */
34    long my_rank = (long) rank;
35
36    printf("Hello from thread %ld of %d\n",
37           my_rank, thread_count);
38
39    return NULL;
40 }  /* Hello */
```

Program 4.1: A Pthreads "hello, world" program.



To run the program, we just type

```
$ ./pth_hello <number of threads>
```

For example, to run the program with 1 thread, we type

```
$ ./pth_hello 1
```

and the output will look something like this:

```
Hello from the main thread
Hello from thread 0 of 1
```

To run the program with four threads, we type

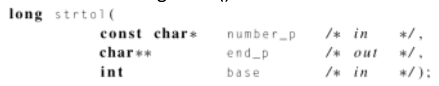
```
$ ./pth_hello 4
```

and the output will look something like this:

```
Hello from the main thread
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

Note: that we usually don't have direct control of the order in which threads execute and thus the output isnt always in the order you expect it to be in

- **Preliminaries** of the hello world pthreads code
  - First notice that this is just a C program with a main function and one other function.
  - In Line 3 we include pthread.h, the Pthreads header file, which declares the various Pthreads functions, constants, types, and so on.
  - In Line 6 we define a global variable thread_count.
    - In pthreads prgs, global variables are shared by all threads
    - Local variables and function arguments—that is, variables declared in functions—are (ordinarily) private to the thread executing the function.
      - If several threads are executing the same function, each thread will have its own private copies of the local variables and function arguments.
        - This makes sense if you recall that each thread has its own stack.
    - Also note that global variables can cause bugs in pthread prgs. Try and minimize using them.
  - In Line 15 the program gets the number of threads it should start from the command line.
    - Unlike MPI programs, Pthreads programs are typically compiled and run just like serial programs, and one relatively simple way to specify the number of threads that should be started is to use a command-line argument.
  - What does the long strtol() function do?

    ```
    long strtol(
              const char*   number_p   /* in   */,
              char**        end_p      /* out  */,
              int           base       /* in   */);
    ```

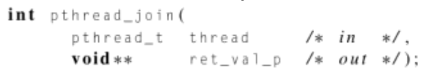
    - The purpose of the strtol() is that it returns a long.
    - Strtol() has three params.
      - Number_p which is a pointer to the first character in the string that is a number
      - End_p is a param which can be null but it tells you if the string has an invalid (non numerical char)
      - Base which can be anywhere from base 2 to base 32, base 10 is the system we use like 0,1,2,..
- **Starting the threads**: As we already noted, unlike MPI programs, in which the processes are usually started by a script, in Pthreads the threads are started by the program executable. This introduces a bit of additional complexity, as we need to include code in our program to explicitly start the threads, and we need data

structures to store information on the threads.
- In Line 17 we allocate storage for one pthread_t object for each thread.
    - The pthread_t data structure is used for storing thread-specific information.
    - It's declared in pthread.h.

    -
      ```
      pthread_t* thread_handles;
      ```
        - Of type pthread_t pointer
    - The pthread_t objects are examples of opaque objects. The actual data that they store is system specific, and their data members aren't directly accessible to user code.
        - However, the Pthreads standard guarantees that a pthread_t object does store enough information to uniquely identify the thread with which it's associated.
- In Lines 19–21,we use the pthread_create function to start the threads.

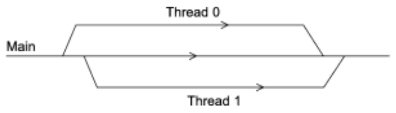
    -
      ```
      int pthread_create(
          pthread_t*              thread_p                   /* out */,
          const pthread_attr_t*   attr_p                     /* in  */,
          void*                   (*start_routine)(void*)   /* in  */,
          void*                   args_p                     /* in  */);
      ```
        - The first argument is a pointer to the appropriate pthread_t object.
            - Note that the object is not allocated by the call to pthread_create; it must be allocated before the call.
            - In this case we allocated for the object on line 17
        - We won't be using the second argument, so we just pass NULL in our function call.
        - The third argument is the function that the thread is to run
            - The function that's started by pthread_create should have a prototype that looks something like this:
              ```
              void* thread_function(void* args_p);
              ```
                - Recall that the type void∗ can be cast to any pointer type in C, so args_p can point to a list containing one or more values needed by thread_function. Similarly, the return value of thread_function can point to a list of one or more values.
        - last argument is a pointer to the argument that should be passed to the function start_routine.
            - we're effectively assigning each thread a unique integer rank.
            - if when we start the threads, we assign the first thread rank 0, and the second thread rank 1, we can easily determine which thread
            - Since the thread function takes a **void**∗ argument, we could allocate one **int** in main for each thread and assign each allocated **int** a unique value. When we start a thread, we could then pass a pointer to the appropriate **int** in the call to pthread_create. However, most programmers resort to some trickery with casts. Instead of creating an **int** in main for the "rank," we cast the loop variable thread to have type **void**∗. Then in the thread function, hello, we cast the argument back to a **long** (Line 34).
            - Note that our method of assigning thread ranks and, indeed, the thread ranks themselves are just a convenient convention that we'll use. There is no requirement that a thread rank be passed in the call to pthread_create, nor a requirement that a thread be assigned a rank. The following thread procedure expects a pointer to a **struct** to be passed in for args_p. The **struct** contains both a rank and the name of the task. (Imagine distinguishing between different requests in a web server, for instance.)
              When we create the thread, a pointer to the appropriate **struct** is passed to pthread_create. We can add the logic to do this at Line 19 (in this case, each thread has the same "task name"):
              Also note that there is no technical reason for each thread to run the same function; we could have one thread run hello, another run goodbye, and so on. However, as with the MPI programs, we'll typically use "single program, multiple data" style parallelism with our Pthreads programs. That is, each thread will run the same thread function, but we'll obtain the effect of different thread functions by branching within a thread.
              ```
              struct thread_args {
                  long my_rank;
                  char *task_name;
              };

              void *Hello(void *args) {
                  struct thread_args* t_args
                     = (struct thread_args *) args;
                  printf("Thread %ld is working on task '%s'\n",
                     t_args->my_rank, t_args->task_name);
                  return NULL;
              }
              ```
              <- it creates its own struct to house the values
                - in the thread function, each thread deconstructs the struct to access its values.
              ```
              struct thread_args *t_args
                 = malloc(sizeof(struct thread_args));

              t_args->my_rank = thread;
              t_args->task_name = "Hello task";

              pthread_create(&thread_handles[thread],
                 NULL,
                 Hello,
                 (void *) t_args);
              ```
              <- creating space in heap for storing the args that will go into each function
              <- changing the pthreads_create() function to now include the struct of args
                - All of this is saying that we can pass in using a struct any number of arguments into the function that the threads will be using. We would allocate space in the heap of type stuct-we-created multiplied by the number of threads we will be running
        - The return value for most Pthreads functions indicates if there's been an error in the function call.
- Also note that there is no technical reason for each thread to run the same function; we could have one thread run hello, another run goodbye, and so on.
    - However, as with the MPI programs, we'll typically use "single program, multiple data" style parallelism with our Pthreads programs.
    - That is, each thread will run the same thread function, but we'll obtain the effect of different thread functions by branching within a thread.

- **Running the Threads**
    - The thread running the main function is sometimes called the main thread
        - Hence why it prints out "Hello from the main thread"
        - In the meantime the threads started by the calls to pthread_create() are also running
            - They get their rank and then they print their rank
        - The function running the thread has a return value but in this case we don't need it to return anything so we return NULL
        - There's no argument in pthread_create saying which core should run which thread; thread placement is controlled by the operating system.
            - Indeed, on a heavily loaded system, the threads may all be run on the same core.
            - In fact, if a program starts more threads than cores, we should expect multiple threads to be run on a single core.

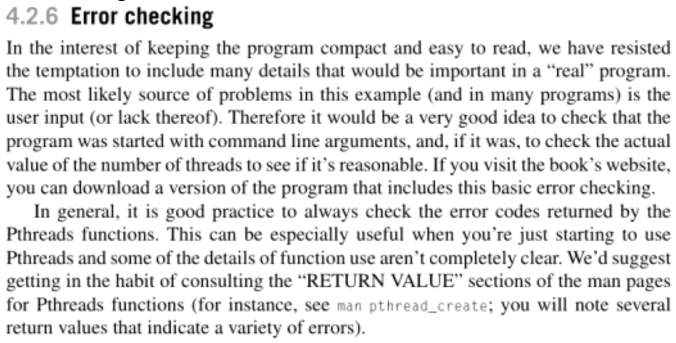
            - However, if there is a core that isn't being used, operating systems will typically place a new thread on such a core.
- Stopping the threads
    - In Lines 25 and 26, we call the function pthread_join once for each thread.
        - A single call to pthread_join will wait for the thread associated with the pthread_t object to complete.
            - Pthread_t object essentially holds the thread, almost like a casing, so when we talk about a thread we talk about the object
            - 

```
int pthread_join(
        pthread_t  thread      /* in  */,
        void**     ret_val_p   /* out */);
```

                - The first param is the thread object
                - The second param is the return value(s)(could be a struct) of that thread
            - eventually the main thread will call pthread_join on that thread to complete its termination.
            - This function is called pthread_join because of a diagramming style that is often used to describe the threads in a multithreaded process.



                - If we think of the main thread as a single line in our diagram, then, when we call pthread_create, we can create a branch or fork off the main thread.
                - Multiple calls to pthread_create will result in multiple branches or forks.
                - Then, when the threads started by pthread_create terminate, the diagram shows the branches joining the main thread.
            - As noted previously, every thread requires a variety of resources to be allocated, including stacks and local variables.
                - The pthread_join function not only allows us to wait for a particular thread to finish its execution but also frees the resources associated with the thread.
                - In fact, not joining threads that have finished execution produces zombie threads that waste resources and may even prevent the creation of new threads if left unchecked.
                    - If your program does not need to wait for a particular thread to finish, it can be detached with the pthread_detach function to indicate that its resources should be freed automatically upon termination.
- **Error Checking**



4.2.6 Error checking

In the interest of keeping the program compact and easy to read, we have resisted the temptation to include many details that would be important in a "real" program. The most likely source of problems in this example (and in many programs) is the user input (or lack thereof). Therefore it would be a very good idea to check that the program was started with command line arguments, and, if it was, to check the actual value of the number of threads to see if it's reasonable. If you visit the book's website, you can download a version of the program that includes this basic error checking.

In general, it is good practice to always check the error codes returned by the Pthreads functions. This can be especially useful when you're just starting to use Pthreads and some of the details of function use aren't completely clear. We'd suggest getting in the habit of consulting the "RETURN VALUE" sections of the man pages for Pthreads functions (for instance, see `man pthread_create`; you will note several return values that indicate a variety of errors).

- Other approaches to thread startup
    - In our example, the user specifies the number of threads to start by typing in a command-line argument. The main thread then creates all of the "subsidiary" threads. While the threads are running, the main thread prints a message, and then waits for the other threads to terminate. This approach to threaded programming is very similar to our approach to MPI programming, in which the MPI system starts a collection of processes and waits for them to complete.
    - There is however another way to go about creating a pthread program which creates all the threads at the start and they sit idle until they are needed
        - This approach saves on resources because it doesn't need to continuously make overhead calls to make and delete threads.
    - Here is a more in depth look at this:

There is, however, a very different approach to the design of multithreaded programs. In this approach, subsidiary threads are only started as the need arises. As an example, imagine a Web server that handles requests for information about highway traffic in the San Francisco Bay Area. Suppose that the main thread receives the requests and subsidiary threads fulfill the requests. At 1 o'clock on a typical Tuesday morning, there will probably be very few requests, while at 5 o'clock on a typical Tuesday evening, there will probably be thousands. Thus a natural approach to the design of this Web server is to have the main thread start subsidiary threads when it receives requests.

Intuitively, thread startup involves some overhead. The time required to start a thread will be much greater than, for instance, a floating point arithmetic operation, so in applications that need maximum performance the "start threads as needed" approach may not be ideal. In such a case, it is usually more performant to employ a scheme that leverages the strengths of both approaches: our main thread will start all the threads it anticipates needing at the beginning of the program, but the threads will sit idle instead of terminating when they finish their work. Once another request arrives, an idle thread can fulfill it without incurring thread creation overhead. This approach is called a *thread pool*, which we'll cover in Programming Assignment 4.5.