Threading – Part 1

"A thread is a logical flow that runs in the context of a process."

Multiple threads can run concurrently within a single process.

"Each thread has its own thread context, including a unique integer thread ID (TID), stack, stack pointer, program counter, general-purpose registers, and condition codes."

"All threads running in a process share the entire virtual address space of that process."

The kernel schedules threads to run.

When a process is created, it is provided with a main thread. It is this main thread that is scheduled to run by the kernel.

Because a thread context is much smaller than a process context, a thread context switch is faster than a process context switch.

(quoted text from Bryant & O'Hallaron, 3rd ed.)

The standard thread library on Unix is known as Posix threads, or `pthreads`.

The `pthreads` library contains approximately 60 functions for manipulating threads.

A `pthread` ID (TID) is stored in a variable of type `pthread_t`.

The thread is created by a call to the `pthread_create` function:

```
#include <pthread.h>
typedef void * (func)(void *);
int pthread_create( pthread_t * tid,
 pthread_attr_t * attr, func * f,
 void * arg );
```

- The `attr` parameter can be used to control certain attributes of the thread.
- The third parameter is a pointer to any function of type `func`.
- The `func` type is defined as a function that has a single parameter of type void * and that returns a void *. The parameter can point to an object of any type, including a struct that contains more objects. Similarly, the function returns a pointer to any type of object. Note that the return value must not point to an object that was local to a function called by the thread.
- The final parameter is a pointer to an object that will be passed to the thread when execution begins.
- `pthread_create` returns 0 if it succeeds; otherwise, it returns a non-zero error code.

The thread calling `pthread_create` can obtain the TID of the newly-created thread by reading the value in the `tid` argument passed.

Once the thread is created, execution begins by calling the function passed as the third argument.

The `arg` object is passed to the call to the function.

The newly-created created thread can learn its TID by calling the function `pthread_self`:

```
#include <pthread.h>
pthread_t pthread_self( void );
```

Example (modified version from Bryant & O'Hallaron, p. 949):

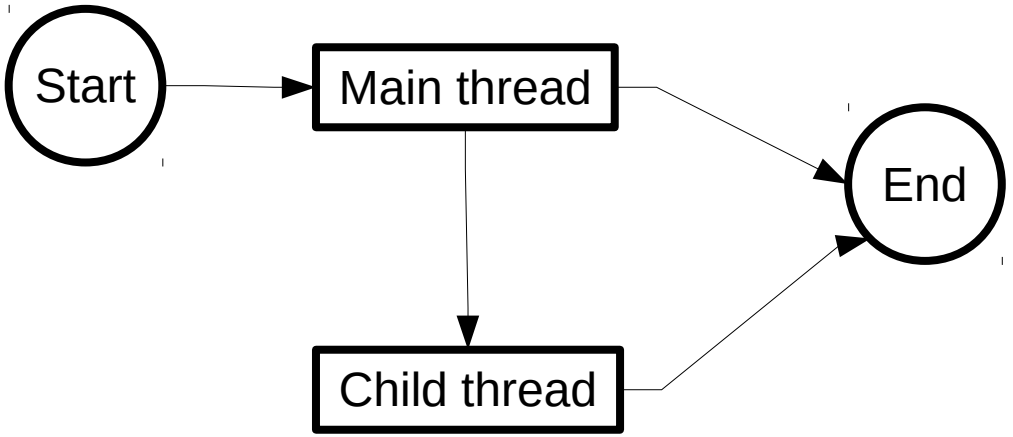
```
void * thread_func(void * vargp);

int main(){
    pthread_t tid;
    void * thread_return;
    pthread_create(&tid, NULL,
     thread_func, NULL);
    printf( "Child Thread ID: %lu\n",
     tid );
    pthread_exit( NULL ); // needs
     // this call or pthread_join to
     // prevent main from exiting the
     // program
    //pthread_join( tid,
     &thread_return );
}

/* Thread routine */
void * thread_func(void * vargp){
    printf("Hello, world!\n");
    return NULL;
}
```

Add “-pthread” to compile

Visualization of threads:



A thread terminates when the initial function called returns.

The thread may also terminate immediately by calling `pthread_exit`:

```
#include <pthread.h>
void pthread_exit(void *
 thread_return);
```

The call terminates the thread immediately.
The `thread_return` value can be obtained by another thread that waits on the calling thread.

If the main thread calls `pthread_exit`, the process waits for all other threads to terminate before returning the value passed by the main thread's call to `pthread_exit`.

If any thread calls `exit`, however, all threads and the process are immediately terminated.

Any thread can terminate any other thread, including the main thread, by calling `pthread_cancel`:

```
#include <pthread.h>
int pthread_cancel( pthread_t tid );
```

A thread can wait for another thread to terminate by calling `pthread_join`:

```
#include <pthread.h>
int pthread_join( pthread_t tid,
 void ** thread_return );
```

The calling thread is suspended until the thread passed as the first argument terminates.

Once `pthread_join` returns, the caller can obtain the value returned by the terminated thread using the second argument passed.

By calling `pthread_join`, the calling thread reaps any system resources used by the terminated thread.

Note that the call to `pthread_join` must specify a particular thread to wait on; it cannot be used to wait for the next thread that happens to terminate.

Not all threads can be joined, though.

A thread can detach another thread.

A detached thread cannot be terminated by another thread.

A thread is detached by calling `pthread_detach`:

```
#include <pthread.h>
int pthread_detach( pthread_t tid );
```

Threads can also detach themselves.

Once a thread is detached, the system reaps it when it terminates.

Because threads share the same memory space of the process under which they are running, they have equal access to functions and certain variables.

There are three classes of variables, two of which are shared by threads:

global variables-- declared outside of the scope of a function

local automatic variables-- declared inside of the scope of a function without any qualifier, i.e., an ordinary stack variable

local static variables-- declared inside the scope of a function with the "static" qualifier

All threads within a process share the global variables and local static variables.

Local automatic variables may be accessed, generally, by only the thread in which they are created. In languages such as C, however, a pointer can be used to provide one thread with access to a local automatic variable that was created by a different thread.