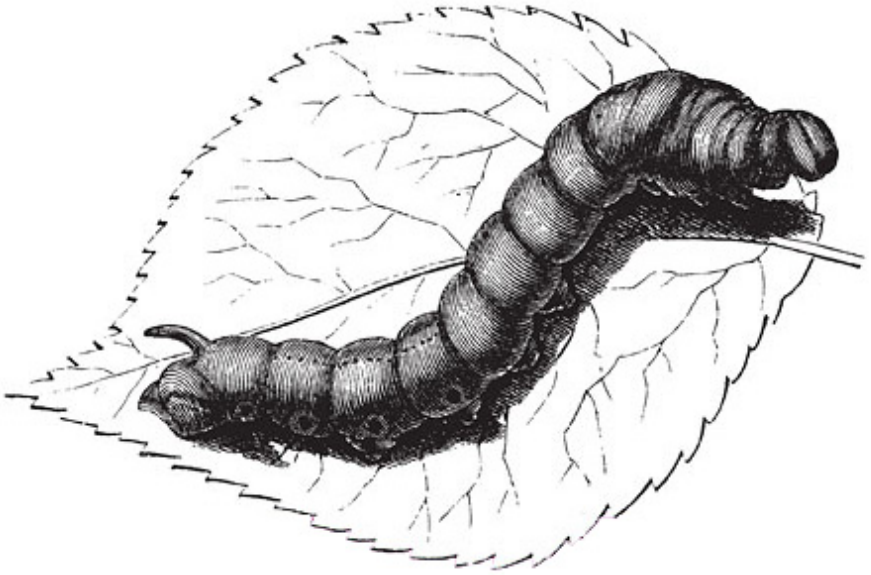
POSIX Threads

Amir Saman Memaripour



CREATING AND DESTROYING THREADS

The **pthread_create()** function is used to create a new thread. If successful, the **pthread_create()** function returns zero. Otherwise, an error number is returned to indicate the error.

int pthread_create(pthread_t **thread*, const pthread_attr_t **attr*, void *(**start_routine*)(void*), void **arg*);

CREATING AND DESTROYING THREADS

The **pthread_cancel()** function requests that thread be canceled. The target threads cancelability state and type determines when the cancellation takes effect.

int pthread_cancel(pthread_t thread);

CREATING AND DESTROYING THREADS

You should plan to collect the exit status of all the threads you create by calling **pthread_join()** on each thread eventually. The **pthread_join()** function suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

CREATING AND DESTROYING THREADS

```
#include <pthread.h>
void * thread_function ( void * arg ) {
        int * incoming = ( int *) arg;
        // Do whatever is necessary using * incoming as the argument.
        return NULL;
}
int main ( void ) {
        pthread_t thread_ID;
        void * exit_status;
        int value = 42;
        pthread_create (&thread_ID , NULL, thread_function , &value ) ;
        pthread_join ( thread_ID , &exit_status ) ;
        return 0;
}
```

RETURNING RESULTS FROM THREADS

```
void * thread_function ( void ) {
        char * buffer = ( char *) malloc ( 64 );
        // Fill up the buffer with something good.
        return buffer;
}

void  * exit_status;
pthread_join ( thread_ID, &exit_status );
char * thread_result;
thread_result = ( char * ) exit_status;
printf ( "I got %s back from the thread .\n" , thread_result );
free ( exit_status );
```

THREAD SYNCHRONIZATION

The idea is to associate a **mutex** with each shared data object and then require every thread that wishes to use the shared data object to first lock the **mutex** before doing so.

1. Declare an object of type **pthread_mutex_t**.
2. Initialize the object by calling **pthread_mutex_init()**.
3. Call **pthread_mutex_lock()** to gain exclusive access to the shared data object.
4. Call **pthread_mutex_unlock()** to release the exclusive access and allow another thread to use the shared data object.
5. Get rid of the object by calling **pthread_mutex_destroy()**.

THREAD SYNCHRONIZATION

```
#include <pthread.h>
#include <unistd.h>
pthread_mutex_t lock;
int shared_data = 0;

void * thread_function ( void * arg ) {
        int i;
        for ( i = 0 ; i < 10241024; ++i ) {
                // Access the shared data here.
                pthread_mutex_lock(&lock);
                shared_data++;
                pthread_mutex_unlock(&lock);
        }
        return NULL;
}
```

THREAD SYNCHRONIZATION

```
int main ( void ) {
        pthread_t thread_ID;
        void * exit_status;
        int i;
        // Initialize the mutex before trying to use it.
        pthread_mutex_init (&lock, NULL);
        pthread_create (&thread_ID, NULL, thread_function, NULL);
        // Try to use the shared data.
        for ( i = 0 ; i < 1 0 ; ++i ) {
                sleep ( 1 );
                pthread_mutex_lock(&lock );
                printf ( "\rShared integer's value = %d\n" , shared_data );
                pthread_mutex_unlock(&lock );
        }
        pthread_join ( thread_ID, &exit_status );
        // Clean up the mutex when we are finished with it.
        pthread_mutex_destroy(&lock );
        return 0 ;
}
```

THREAD SYNCHRONIZATION

1. No thread should attempt to lock or unlock a **mutex** that has not been initialized.
2. The thread that locks a **mutex** must be the thread that unlocks it.
3. No thread should have the **mutex** locked when you destroy the **mutex**.
4. Any **mutex** that is initialized should eventually be destroyed, but only after any thread that uses it has either terminated or is no longer interesting in using it.

ASSIGNMENT

Write a program that creates **10 threads**. Have each thread execute the same function and **pass each thread a unique number**.

Each thread should print “Hello, World (thread n)” five times where ‘n’ is replaced by the **thread's number**.

Use an array of **pthread_t** objects to hold the various **thread IDs**. Be sure the program doesn't terminate **until all the threads are complete**.

Try running your program on more than one machine. Are there any differences in how it behaves?

REFERENCES

- D. Buttlar, J. Farrell, B. Nichols, *PThreads Programming: A POSIX Standard for Better Multiprocessing*, O'Reilly Media, September 1996.
- Open Group's Online Manual for PThreads, available at http://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html
- S. King, *pthread Examples*, University of Illinois, Lecture Notes.
- P. C. Chapin, *pthread Tutorial*, August 2008.