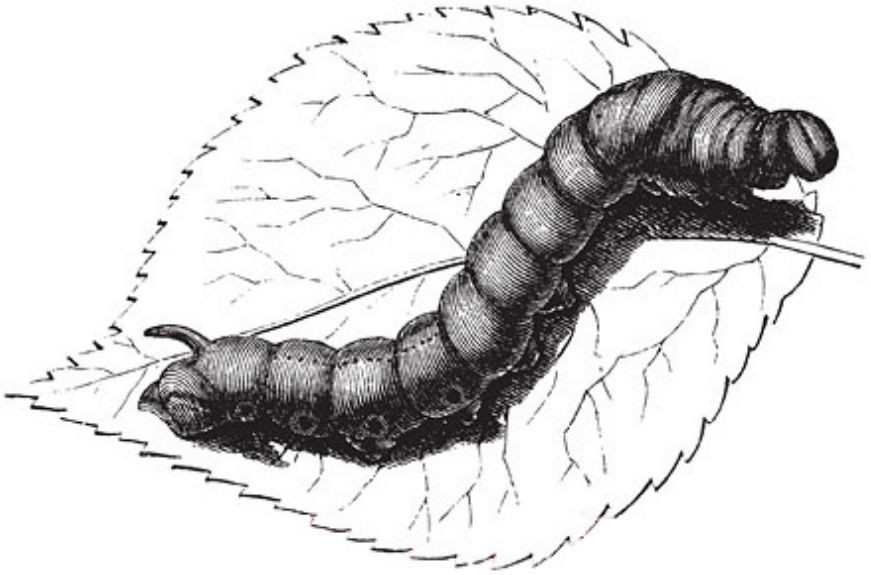
POSIX Threads

Amir Saman Memaripour



CONDITION VARIABLES

If you want one thread to **signal an event** to another thread, you need to use **condition variables**. The idea is that one thread waits until a certain condition is true.

First it tests the condition and, if it is not yet true, calls **pthread_cond_wait()** to block until it is.

At some later time another thread makes the condition true and calls **pthread_cond_signal()** to unblock the first thread.

CONDITION VARIABLES

The **pthread_cond_wait()** and **pthread_cond_timedwait()** functions are used to block on a condition variable. They are called with **mutex** locked by the calling thread or undefined behavior will result.

int pthread_cond_wait(pthread_cond_t **cond*, pthread_mutex_t **mutex*);

int pthread_cond_timedwait(pthread_cond_t **cond*, pthread_mutex_t **mutex*, const struct timespec **abstime*);

CONDITION VARIABLES

The **pthread_cond_signal()** call unblocks at least one of the threads that are blocked on the specified condition variable cond (if any threads are blocked on cond).

The **pthread_cond_broadcast()** call unblocks all threads currently blocked on the specified condition variable cond.

int pthread_cond_signal(pthread_cond_t *_cond_);

int pthread_cond_broadcast(pthread_cond_t *_cond_);

CONDITION VARIABLES

```
#include <pthread.h>
#include <unistd.h>
pthread_cond_t is_zero;
pthread_mutex_t mutex; // Condition variables needs a mutex.
int shared_data = 32767; // Or some other large number.

void * thread_function ( void * arg ) {
        while ( shared_data > 0) {
                pthread_mutex_lock(&mutex ) ;
                --shaed_data ;
                pthread mutex unlock(&mutex ) ;
        }
        pthread_cond_signal (&is_zero ) ;
        return NULL;
}
```

CONDITION VARIABLES

```
int main ( void) {
        pthread_t thread_ID;
        void * exit_status;
        int i;
        pthread_cond_init (&is_zero, NULL);
        pthread_mutex_init (&mutex, NULL);
        pthread_create (&thread_ID, NULL, thread_function, NULL);
        // Wait for the shared data to reach zero.
        pthread_mutex_lock(&mutex );
        while ( shared_data != 0)
                pthread_cond_wait (&is_zero, &mutex ) ;
        pthread_mutex_unlock(&mutex );
        pthread_join ( thread_ID , &exit_status ) ;
        pthread_mutex_destroy(&mutex );
        pthread_cond_destroy (&is_zero );
        return 0 ;
}
```

SEMAPHORES

One of the most important differences between a **pthread mutex** and a **semaphore** is that, unlike a **mutex**, a **semaphore** can be signaled in a different thread than the thread that does the wait operation.

int sem_post(sem_t *sem);

int sem_wait(sem_t *sem);

SEMAPHORES

```
#include <semaphore.h>
int shared ;
sem_t binary_sem ; // Used like a mutex.
void * thread_function ( void * arg ) {
                    sem_wait (&binary_sem ) ; // Decrements count.
                    // Use shared resource.
                    sem_post (&binary_sem ) ; // Increments count.
}
int main ( void) {
                    sem_init (&binary_sem, 0, 1 ) ; // Give semaphore an initial count.
                    // Start threads here.
                    sem_wait (&binary_sem ) ;
                    // Use shared resource.
                    sem_post (&binary_sem ) ;
                    // Join with threads here.
                    sem_destroy(&binary_sem ) ;
                    return 0 ;
}
```

ASSIGNMENT

Write a program in order to model the **Dinning Philosophers** problem. Use **condition variables** and **semaphores** wherever you need.

There should be **at least 5 philosophers** in your program. Once a philosopher starts **waiting**, **dining** or **thinking**, print “Philosopher *i* starts *action*” which *i* indicates philosopher's **unique identifier** and *action* shows **the kind of action** currently is in progress by the philosopher.

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**.

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.