Chapter 11 Notes

Threads

Advanced Programming in the Unix Environment 3e

MCIT 5950

Tyler Ryan

February 15, 2023

Contents

1 Before Reading

Prerequisite Knowledge

You should know a little about the following subjects:

- Generic Types using Void Pointers
- Single Linked List
- Double Linked List
- Queues
- Hash Tables (Using Linked Lists)

Installing Missing Man Pages

Run ***dnf install man-pages-posix*** to install all **pthread.h** manual pages.

2 High Level Overview

- See how threads of control (i.e. Threads) can be used to perform multiple tasks within the environment of a single process.
- All threads within a single process have access to (share) the **same** process components, such as file descriptors and memory.
  - Compare this to this to multiples processes.
- Threads and POSIX.1 primitives available to create and destroy them.
- Thread synchronization problem.

- 3 fundamental synchronization mechanisms:
  - Mutexes
  - Reader-Writer Locks
  - Conditional Variables

3 Thread Concepts

- With multiple threads (of control), a program can be designed to do more than on thing at a time **within a single process**, with each thread handling separate tasks.
- Benefits of Threads:
  - Code that deals with asynchronous events can be simplified by assigning a separate thread to handle each event type.
    - Each thread can then handle its event using a synchronous programming model.
    - A synchronous programming model is much simpler than an asynchronous one.
  - Unlike multiple processes which have to use complex mechanisms provided by the operating system to share memory and file descriptors, **Threads automatically have access to the same memory address space and file descriptors.**
  - Unlike processes which implicitly serializes multiple tasks, threads can interleave indepentent tasks by assigning a separate thread per task.
    - Two tasks can be interleaved only if they don't depend on the processing performed by each other.
  - Interactive programs can realize **improved response time** by using multiple threads to separate the porions of the program that deal with user input and output from the other parts of the program.

- The benefits of a multithreaded programming model can be realized even if your program is running on a uniprocessor.
- A thread consists of the information necessary to represent an **execution context** within a process.
  - Thread ID
  - Set of Registers
  - A Stack
  - A Scheduling priority and policy
  - A Signal Mask
  - An Errno Variable
  - Thread-Specific Data
- Everything within a process is sharable among the threads in that process including:
  - The text of the executable program
  - The program's global and heap memory
  - The Stacks
  - The File Descriptors

4 Basic Function Prototypes

```
#include <pthread.h>
// Data Type
pthread_t

// Compare the value of 2 threads
int pthread_equal(pthread_t tid1, pthread_t tid2);

// Get current thread's ID
pthread_t pthread_self(void);

// Create a thread
int pthread_create(pthread_t * restrict tidp,
                   const pthread_attr_t * restrict attr,
                   void * (*start_rtn)(void *),
                   void * restrict arg);

// Returns from a thread
void pthread_exit(void * rval_ptr);

// Like wait() for threads
int pthread_join(pthread_t thread, void **rval_ptr);

// Like sending pthread_exit() to another thread
int pthread_cancel(pthread_t tid);

// Cleanup functions just before threads exit
void pthread_cleanup_push(void (*rtn)(void *), void * arg);
// "int execute" is more like "bool execute"
void pthread_cleanup_pop(int execute);
```

4.1 Comparing Process Functions and Thread Functions

Table 1: Process Functions vs Thread Functions

Process Primitive	Thread Primitive	Description
fork	pthread_create	Create a new flow of control
exit	pthread_exit	Exit from an existing flow control
waitpid	pthread_join	Get exit status from flow control
atexit	pthread_cleanup_push	Register function to be called at exit from flow of control
getpid	pthread_self	Get ID for flow of control
abort	pthread_cancel	Request abnormal termination of flow of control
kill	pthread_kill	Send signals to other flows of control

5 Thread Identification

- Every thread has a Thread ID
- These Thread IDs are only unique inside each process.
- Each thread shares the same Process ID (PID)

6 Thread Creation

- All programs start as signle threaded processes.
- When a thread is created, just like with processes, there is no guarantee which runs first: The newly created thread or the calling thread (program).
  - Has access to the process address space.
  - Inherits the calling thread's floating-point environment and signal mask.
  - However, the set of *pending signals* for the thread is cleared.
- If you need to pass more than one argument to the **start_rtn** function, then you need to store them in a struct and pass the *address* of that struct to the **arg** parameter.
- pthread functions usually return an error code when they fail. they don't set **errno** like the other POSIX functions.

6.1 Code: Printing Thread IDs

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <string.h>

/* Write a function that prints the current thread id
 * */

pthread_t ntid;

/* Notice how similar the Hex numbers (address ranges) are
 * This means that they are near each other in memory.
 *
 * Running on Linux (Fedora 37), these are the results.
 *
 * PIDs are the same.
 *
```

```
 * TID addresses are different,
 *  First 4 positions are the same.
 *  Last 5 positions are the same.
 */
void
print_tid(const char * s)
{
    pthread_t tid  = pthread_self();
    printf("%s (TID): %u, (HEX) %08x%x, (PID) %u\n",
           s,
           (unsigned int) tid,
           (unsigned int) tid,
           (unsigned int) getpid());
}

/* When you create a new thread, you must pass in a function.
 * In our case, our function just calls our print_tid() function.
 */
void *
thread_fn(void * arg)
{
    print_tid("new thread: ");

    // Note: In Linux, returning just 0 is allowable.
    return (void *) 0;
}

int
main(void)
{
    int     err;

    // This will call a thread that will then print it's TID.
    err = pthread_create(&ntid, NULL, thread_fn, NULL);

    if (err != 0)
    {
        printf("error: can't create thread: %d\n", strerror(err));
        exit(EXIT_FAILURE);
    }

    // Print the main thread's TID
    print_tid("main thread: ");
    sleep(1);
    exit(EXIT_SUCCESS);
}
```

Output

```
main thread:  (TID): 1975252800, (HEX) 75bbf7401fef9, (PID) 206267015
new thread:  (TID): 1975248576, (HEX) 75bbe6c01fef9, (PID) 206267015
```

7 Thread Termination

- A single thread can exit in 3 ways, stopping its flow of control without terminating the entire process.
  1. The thread can return from the start routine.
     - The return value is the thread's exit code
  2. The thread can be canceled by another thread in the **same process**.
  3. The thread can call **pthread_exit()**
- The thread that calls **pthread_join()** will block until the specified thread calls **pthread_exit()** returns from its start routing or is canceled.
  - If the thread returns from its routine, **rval_ptr** will contain the return code.
  - If the thread was canceled, the memory location specified by **rval_ptr** is set to PTHREAD_CANCELED.
- By calling **pthread_join()**, the joining thread is automatically placed in a detached state so that its resources can be recovered.

7.1 Code: Getting Exit Codes From Threads

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <string.h>

/* A simple program that shows how to
 * obtain the exit status and the
 * return value of a thread using either
 * "return" or "pthread_exit()"
 */

void *
thread_func1(void * arg)
{
    printf("thread 1 returning\n");
    return ((void*)1);
}

// See if you can return a char *
// I'd prefer this one
void *
thread_func2(void * arg)
{
    printf("thread 2 exiting\n");
    // To return a char *
    //pthread_exit("HELLER\n");
    pthread_exit((void*)2);
}
```

```
int
main(void)
{
    pthread_t  thread1, thread2;
    int        err;
    void     * thread_return;

    // Create thread 1, which returns 1
    err = pthread_create(&thread1, NULL, thread_func1, NULL);
    if (err !=0)
    {
        printf("error: can't create thread 1: %s\n", strerror(err));
        exit(EXIT_FAILURE);
    }

    // Create thread 2, which returns 2 using pthread_exit()
    err = pthread_create(&thread2, NULL, thread_func2, NULL);
    if (err != 0)
    {
        printf("error: can't create thread 2: %s\n", strerror(err));
        exit(EXIT_FAILURE);
    }

    /* Waits for thread 1 and returns the exit status to err,
     * and your whatever thread_func1() returned to thread_return.
     */
    err = pthread_join(thread1, &thread_return);
    // err == 0
    // thread_return == 1

    if (err !=0)
    {
        printf("error: can't join with thread 1: %s\n",
                                            strerror(err));
        exit(EXIT_FAILURE);
    }

    printf("thread 1 exit code: %d\n", (int*) thread_return);

    /* Waits for thread 2 and returns the exit status to err,
     * and your whatever thread_func2() returned to thread_return.
     */
    err = pthread_join(thread2, &thread_return);
    // err == 0
    // thread_return == 2

    if (err !=0)
    {
        printf("error: can't join with thread 2: %s\n",
                                          strerror(err));
```

```
        exit(EXIT_FAILURE);
    }

    printf("thread 2 exit code: %d\n", (int*) thread_return);
    // To return a char *
    //printf("thread 2 exit code: %s\n", (char*) thread_return);

    exit(EXIT_SUCCESS);
}
```

Output

```
thread 2 exiting
thread 1 returning
thread 1 exit code: 1
thread 2 exit code: 2
```

- Be careful when passing values from thread to thread.
- For example, if you create a struct on one thread's stack and try to pass it to another, it will return random values as that struct was destroyed when that thread was destroyed.
- In essence, when values create within threads die with their threads unless made static or put on the heap.

7.2 Code: Incorrectly Passing Values Between Threads

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>


struct foo { int a, b, c, d; } ;

void
print_foo(const char *s, const struct foo *fp)
{
    printf(s);
    printf(" structure at 0x%x\n", *(unsigned int*)fp);
    printf(" foo.a = %d\n", fp->a);
    printf(" foo.b = %d\n", fp->b);
    printf(" foo.c = %d\n", fp->c);
    printf(" foo.d = %d\n", fp->d);
}

void *
thread_func1(void * arg)
{
    // Creating a foo struct on this thread's stack
    struct foo foo = {1,2,3,4};
    print_foo("thread 1:\n", &foo);
```

```
    /* Because foo was not made static or put on the
     * heap, it will die when we exit this function.
     */
    pthread_exit((void*)&foo);
}

void *
thread_func2(void * arg)
{
    printf("thread 2: ID is %ld\n", pthread_self());
    pthread_exit((void*)0);
}

int
main(void)
{
    /* For the sake of simplicity, I'm not handling
     * any errors
     */
    pthread_t tid1, tid2;
    struct    foo *fp;
    int       err;

    // CREATES a foo struct WITHIN this thread
    err = pthread_create(&tid1, NULL, thread_func1, NULL);

    /* Returns a pointer to the foo struct here.
     * However, that thread has already been terminated.
     * If you tried to access the value of fp which points
     * to memory that has been terminated, you will get
     * junk.
     */
    err = pthread_join(tid1, (void*)&fp);

    sleep(1);

    printf("parent starting second thread\n");

    err = pthread_create(&tid2, NULL, thread_func2, NULL);

    sleep(1);
    print_foo("parent:\n", fp);

    exit(EXIT_SUCCESS);
}
```

Output

```
thread 1:
 structure at 0x1
 foo.a = 1
 foo.b = 2
 foo.c = 3
 foo.d = 4
parent starting second thread
thread 2: ID is 140514740258496
parent:
 structure at 0x232dc6c0
 foo.a = 590202560
 foo.b = 32716
 foo.c = 0
 foo.d = 0
```

The parent's output *should* be the same as the first thread, however it isn't because its data was destroyed when the first thread exited.

7.2.1 Code Challenge 1: Returning Multiple Values From Thread

Try to get this to work. Hint: Use global/heap space.

Expected Output

```
thread 1:
 structure at 0x1
 foo.a = 1
 foo.b = 2
 foo.c = 3
 foo.d = 4
parent starting second thread
thread 2: ID is 140436664235712
parent:
 structure at 0x1
 foo.a = 1
 foo.b = 2
 foo.c = 3
 foo.d = 4
```

See Listing 2 for a **solution**.

8 Thread Synchronization

8.1 Function Prototypes

```
#include <pthread.h>

// ---------------------- MUTEX Functions ---------------------------
// Mutex Data Type
pthread_mutex_t

// Use for statically-allocated Mutexes
PTHREAD_MUTEX_INITIALIZER


int pthread_mutex_int(pthread_mutex_t * restrict mutex
                      const pthread_mutexattr_t * restrict attr);
int pthread_mutex_destroy(pthread_mutex_t * mutex);

int pthread_mutex_lock(pthread_mutex_t * mutex);
int pthread_mutex_trylock(pthread_mutex_t * mutex);
int pthread_mutex_unlock(pthread_mutex_t * mutex);

                    Return: 0 if OK, error number on failure
// ------------------------------------------------------------------
```

8.2 Mutexes

A Mutex variable is represented by **pthread_mutex_t**

Use **PTHREAD_MUTEX_INITIALIZER** for statically-allocated mutexes.

8.2.1 Code: Basic Mutex Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

/* Using a signle thread, get this
 * code to work.
 */

struct foo
{
    int             f_count;
    pthread_mutex_t  f_lock;
};

struct foo *
foo_alloc(void)
{
```

```
    struct foo * fp;
    fp = malloc(sizeof(struct foo));
    // Keep track of f_lock
    int res = pthread_mutex_init(&fp->f_lock, NULL);

    if (res != 0)
    {
        free(fp);
        return (NULL);
    }

    return fp;
};

void
foo_hold(struct foo * fp)
{
    // Lock f_lock
    pthread_mutex_lock(&fp->f_lock);
    // Do something
    fp->f_count++;
    // Unlock f_lock
    pthread_mutex_unlock(&fp->f_lock);
}

void
foo_release(struct foo * fp)
{
    /* "--fp->f_count" is interesting code.
     *
     * What it does is pre-decrement the actual
     * f_count amount by 1.
     *
     * if fp->f_count was 1, then --fp->f_count
     * would == 0.
     *
     * If that is the case, then we can go ahead and
     * do the following:
     * 1. Unlock it
     * 2. Destroy it (make it uninitialized)
     * 3. Free it's memory
     */

    if (--fp->f_count == 0)
    {
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
        fp == NULL;
    }
```

```
        /* However, if there are multiple foo objects,
         * then just unlock this particular one.
         */
        else
        {
            pthread_mutex_unlock(&fp->f_lock);
        }
    }

    int
    main(void)
    {
        /* Note: For the sake of simplicity, this example
         * is only showing how to use these functions with
         * 1 thread.
         */
        struct foo * fpointer = foo_alloc();

        // Get 200
        for (int i = 0; i < 200; i++)
        {
            foo_hold(fpointer);
        }

        printf("200: %d\n", fpointner->f_count); // 200

        // Release all but 1
        for (int i = 0; i < 199; i++)
        {
            foo_release(fpointer);
        }

        printf("1: %d\n", fpointer->f_count); // 1

        // Release the last one
        foo_release(fpointer);
        // Invalid memory produces junk information
        printf("Destroyed: %d\n", fpointer->f_count);

        exit(EXIT_SUCCESS);
    }
```

Output

```
200: 200
1: 1
Destroyed: 6846
```

This example shows how to use a mutex with only a single thread.

8.2.2 Code Challenge 2: Dual Threaded Mutex

Try running 2 threads. Each thread should increment fpointer by 1 up to 100. So the total should still end up being 200 (i.e. the same output as above).

See Listing 3 for a **solution**.

8.3 Thread Deadlocking

- A thread will deadlock itself if it tries to lock the same mutex twice.
- If more than one mutex is used in a program a deadlock can occur if the thread is allowed to hold a mutex and block while trying to lock a second mutext at the same time the other thread holding the second mutex tries to lock the first mutex.
- In this case, neither thread can proceed, because each needs a resource that is held by the other, thus the deadlock.
- To avoid deadlocks, ensure that when there is a need to acquire two mutexes at the same time, to always lock them in the same order.

8.3.1 Code: Locking Multiple Mutexes

Note:

- At this point the textbook gets pretty wild with its examples.
- The code in the book wasn't a working example. I've added a few things and made it work using a single thread for simplicity.

In this example, deadlocks are avoided by ensuring that when two mutexes need to be acquired at the same time, they are a locked in the same order.

- The second mutex protects a hash list that is used to keep track of the **foo** data structures.
- The **hashlock** mutex protects both the **fh** hash table and the **f_next** hash link field in the **foo** structure.
- The **f_lock** mutex in the **foo** structure protects the acess to the remainder of the **foo** structure's fields.

Listing 1: Multiple Mutex Locking with Hash Table

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
/* Please note, the book did not give a
 * working example of this program.
 */

// Number of slots
#define NHASH 12
/* HASH(fp) basically reduces our hash number to fit
 * within our NHASH slots
 *
 * Ex.
 * int hash = 5000;
```

```
 * int smaller_hash = HASH(hash); // smaller_hash = 8;
*/
#define HASH(fp) (((unsigned long)fp) % NHASH)

// Struct we are trying to protect
struct foo
{
    int             f_count;
    pthread_mutex_t f_lock;
    struct foo      * f_next;
    int             f_id;
    /* ... more stuff here ... */
};

// Functions
struct foo * foo_alloc(int);
void   foo_hold(struct foo * fp);
struct foo * foo_find(int id);
struct foo * my_find(struct foo * entry);
void   foo_release(struct foo * fp);
void   print_ht_values(void);

// Global variable
// An array of foo
struct foo * fh[NHASH];
// Our global lock/unlock variable
pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;


int
main(void)
{
    /* TODO: Mess around with some stuff you
     * are unsure about here.
     */
    struct foo * a = foo_alloc(150);
    struct foo * b = foo_alloc(200);
    struct foo * c = foo_alloc(300);

    // Note that these both will stay the same thoughout the
    // same execution, but can/will change when reran.
    printf("Actual Value of fp: %ld\n", a);
    printf("Hash value: %ld\n", HASH(a));

    print_ht_values();

    int a_val = foo_find(150)->f_id;
    printf("Just found %d by searching\n", a_val);

    // Uses the hash feature of the hash table to find values
```

```
    int my_val = my_find(a)->f_id;
    printf("I just found %d\n", my_val);

    // This doesn't seem to work
    foo_release(a);
    printf("Removed 150\n");
    printf("New Table\n");
    print_ht_values();
}

void print_ht_values(void)
{
    // Loop through the array
    for (int i = 0; i < NHASH; i++)
    {
        long int fp_value = (long int) fh[i];
        int hash_value = HASH(fp_value);
        // Just print the ones that were allocated space
        if (fp_value != 0)
        {
            printf("index %d, fp: 0x%x, val: %d\n",
                    hash_value,
                    (unsigned int) fp_value,
                    fh[i]->f_id
                    );
        }
    }
}

struct foo *
foo_alloc(int id)
{
    struct foo * fp;
    int    ht_index;

    fp = malloc(sizeof(struct foo));
    // If malloc worked
    if (fp != NULL)
    {
        // Set our count to 1
        fp->f_count = 1;

        // If this fails, free fp
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0)
        {
            free(fp);
            return NULL;
        }
        /* fp is just a number we're trying to sqeeze down
         * below our NHASH number.
```

```
             * This will create the index where we will store
             * our values
             */
            ht_index = HASH(fp);
            // Lock down the struct so we can mess with it safely
            pthread_mutex_lock(&hashlock);
            // point next node to this index
            fp->f_next = fh[ht_index];
            // Set value of that index to this fp
            fh[ht_index] = fp;
            fp->f_id = id;

            // Never allowed to unlock this node?
            pthread_mutex_lock(&fp->f_lock);
            // Unlock struct
            pthread_mutex_unlock(&hashlock);
            /* ... continue initializing ...*/
        }
        return fp;
    }

    void
    foo_hold(struct foo * fp)
    {
        pthread_mutex_lock(&hashlock);
        fp->f_count++;
        pthread_mutex_unlock(&hashlock);
    }

    // this doesn't utilize the hash feature when finding
    // values in the hash table lol
    // It just loops through everything.
    struct foo *
    foo_find(int id)
    {
        struct foo * fp;
        int          ht_index;

        pthread_mutex_lock(&hashlock);
        // For each index of the array
        for (ht_index = 0; ht_index < NHASH; ht_index++)
        {
            // for each node at each index
            for (fp = fh[ht_index]; fp != NULL; fp = fp->f_next)
            {
                // If the node's id matches our id
                if (fp->f_id == id)
                {
                    // increase fp's memory location by 1?
                    fp->f_count++;
```

```
                pthread_mutex_unlock(&hashlock);
                return fp;
            }
        }
    }

    // ID couldn't be found
    pthread_mutex_unlock(&hashlock);
    return NULL;
}

/* This way uses the hash feature.
 * Notice how I've reduced the nested for loop
 * to just a single for loop.
 *
 * If you were to assert that all keys were unique,
 * then you could create a hash function where you
 * could look up the actual value (instead of the object)
 * like you can with the book's function.
 */
struct foo *
my_find(struct foo * entry)
{
    struct foo * fp;
    int        ht_index;

    pthread_mutex_lock(&hashlock);

    unsigned int hash_code = HASH(entry);
    printf("Hash Code %d\n", hash_code);

    for (fp = fh[hash_code]; fp != NULL; fp = fp->f_next)
    {
        // If the node's id matches our id
        if (fp->f_id == entry->f_id)
        {
            // increase fp's memory location by 1?
            fp->f_count++;
            pthread_mutex_unlock(&hashlock);
            return fp;
        }
    }
    // ID couldn't be found
    pthread_mutex_unlock(&hashlock);
    return NULL;
}

// This function DOES utilize the hash feature
void
foo_release(struct foo * fp)
```

```
{
    struct foo  *temp_foo_ptr;
    int         ht_index;

    pthread_mutex_lock(&hashlock);

    // If this is the last "instance"
    if (--fp->f_count == 0)
    {
        ht_index     = HASH(fp);
        temp_foo_ptr = fh[ht_index];

        // If temp_fp happens to be the first node
        if (temp_foo_ptr == fp)
        {
            // Hashtable[index] = this node
            fh[ht_index] = fp->f_next;
        }
        // else loop through the nodes, until
        // the correct node is found.
        else
        {
            while (temp_foo_ptr->f_next != fp)
            {
                temp_foo_ptr = temp_foo_ptr->f_next;
            }
            // At this point, if it's there
            // this will set us there.
            temp_foo_ptr->f_next = fp->f_next;
        }

        // Remove that one from the table
        // 1. Unlock the table
        // 2. Destroy the node's lock
        // 3. Free from memory
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    }
    // If it wasn't found, unlock table and do nothing.
    else
    {
        pthread_mutex_unlock(&hashlock);
    }
}
```

8.4 Reader-Writer Locks

8.4.1 Function Prototypes

```
#include <pthread.h>

// Initialize
int pthread_rwlock_init(pthread_rwlock_t * restrict rwlock,
                        const pthread_rwlockattr_t * restrict attr);
// Destroy/Clean Up
int pthread_rwlock_destroy(pthread_rwlock_t * rwlock);


// Read Lock
int pthread_rwlock_rdlock(pthread_rwlock_t * rwlock);
// Write Lock
int pthread_rwlock_wrlock(pthread_rwlock_t * rwlock);
// Unlock
int pthread_rwlock_unlock(pthread_rwlock_t * rwlock);


// Try Read
int pthread_rwlock_tryrdlock(pthread_rwlock_t * rwlock);
// Try Write
int pthread_rwlock_trywrlock(pthread_rwlock_t * rwlock);
```

- Also called **shared-exclusive locks**.
  - Read mode == shared mode.
  - Write mode == exclusive mode.
- Reader-writer locks are similar to mutexes, except that they allow for higher degrees of parallelism.
- With a mutex, the state is either locked or unlocked, and only one thread can lock it at a time.
- Only one thread at a time can hold a reader-writer lock in write mode, but multiple threads can hold a reader-writer lock in read mode at the same time.
- Reader-writer locks are well suited for situations in which data structures are read more often than they are modified.
- When a reader-writer lock is write-locked, all threads attempting to lock it block until it is unlocked.
- When a reader-writer lok is read-locked, all threads attempting to lock it in read modeare give-naccess, but any threads attempting t olock it in write mode block until all the threads have relinquished their read locks.
  - Implementations vary, but reader-writer locks usually block additional readers if a lock is already held in read mode and a thread is blocked trying to acuquire the lock in write mode.
- When a read-writer lock is held in write mode, the data structure it protects can be modified safely, since only one thread at a time can hold the lock in write mode.
  - Prevents constant stream of readers from starving waiting writers.

8.5 Condition Variables

8.5.1 Function Prototypes

```
#include <pthread.h>

// Data type
pthread_cond_t

// For statically-allocated condition variables.
PTHREAD_COND_INITIALIZER


int pthread_cond_init(pthread_cond_t * restrict cond,
                      pthread_condattr_t * restrict attr);
int pthread_cond_destroy(pthread_cond_t * cond);


int pthread_cond_wait(pthread_cond_t * restrict cond,
                      pthread_mutex_t * restrict mutex);
int pthread_cond_timedwait(pthread_cond_t * restrict cond,
                           pthread_mutex_t * restrict mutex,
                           const struct timespec * restrict timeout);
// timespec structure used in ...timedwait()
struct timespec
{
    time_t tv_sec;  // Seconds
    long   tv_nsec; // Nanoseconds
}


// Wakes up 1 thread waiting on a condition
int pthread_cond_signal(pthread_cond_t * cond);
// Wakes up all threads waiting on a condition
int pthread_cond_broadcast(pthread_cond_t * cond):
```

- Condition variables are another synchronization mechanism available to threads.
- When used with mutexes, condition variables allow threads to wait in a race-free way for arbitrary conditions to occur.
- The condition itself is protected by a mutex.
- A thread must first lock the mutext to change the condition state.
  - Other threads will not notice the change until they acquire the mutex.
  - As the mutext must be locked to be able to evaluate the condition.

9 Code Challenge Solutions

These are just some problems I came up with.

They might or might not:

- Help
- Be correct
- Be worth your time

9.1 Challenge 1

Here I used **malloc** to solve this problem, but you could've also used **static**.

Listing 2: Solution: Returning Multiple Values From A Thread

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <string.h>

/* Write a program that returns a struct
 * (i.e. multiple values) from a thread
 * using malloc
 */

struct foo { int a, b, c, d; } ;

void
print_foo(const char *s, const struct foo *fp)
{
    printf(s);
    printf(" structure at 0x%x\n", *(unsigned int*)fp);
    printf(" foo.a = %d\n", fp->a);
    printf(" foo.b = %d\n", fp->b);
    printf(" foo.c = %d\n", fp->c);
    printf(" foo.d = %d\n", fp->d);
}

// Create a function struct within the thread and returns it
void *
thread_func1(void * arg)
{
    // If you don't put foo on the heap, it will die
    // when this thread dies.
    struct foo * foo = (struct foo*) malloc(sizeof(struct foo));
    foo-> a = 1;
    foo-> b = 2;
    foo-> c = 3;
    foo-> d = 4;
```

```
    // Proof foo exists within the thread
    print_foo("thread 1:\n", foo);
    pthread_exit((void *)foo);
}

void *
thread_func2(void * arg)
{
    printf("thread 2: ID is %ld\n", pthread_self());
    pthread_exit((void*)1);
}


void print_error(char * msg, int err)
{
    printf("%s\n", strerror(err));
    exit(EXIT_FAILURE);
}

int
main(void)
{
    int        err;
    struct     foo *fp;
    pthread_t tid1, tid2;

    // Creates a foo struct within this thread
    err = pthread_create(&tid1, NULL, thread_func1, NULL);
    if (err != 0) print_error("",err);

    // Returns a pointer to the foo struct here
    err = pthread_join(tid1, (void*)&fp);
    if (err != 0) print_error("",err);

    sleep(1);

    printf("parent starting second thread\n");

    err = pthread_create(&tid2, NULL, thread_func2, NULL);
    if (err != 0) print_error("",err);

    sleep(1);
    print_foo("parent:\n", fp);

    // You must free when you use malloc
    free(fp);

    exit(EXIT_SUCCESS);
}
```

9.2 Challenge 2

Listing 3: Solution: Dual Threaded Mutex

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

/* Using multiple threads,
 * get fp->f_count to equal to 200.
 */

struct foo
{
    int             f_count;
    pthread_mutex_t  f_lock;
};

struct foo *
foo_alloc(void)
{
    struct foo * fp;
    fp = malloc(sizeof(struct foo));
    // Keep track of f_lock
    int res = pthread_mutex_init(&fp->f_lock, NULL);

    if (res != 0)
    {
        free(fp);
        return (NULL);
    }

    return fp;
};

void
foo_hold(struct foo * fp)
{
    // Lock f_lock
    pthread_mutex_lock(&fp->f_lock);
    // Do something
    fp->f_count++;
    // Unlock f_lock
    pthread_mutex_unlock(&fp->f_lock);
}

void
foo_release(struct foo * fp)
{
    /* "--fp->f_count" is interesting code.
     *
```

```
         * What it does is pre-decrement the actual
         * f_count amount by 1.
         *
         * if fp->f_count was 1, then --fp->f_count
         * would == 0
         *
         * If that is the case, then we can go ahead and
         * do the following:
         * 1. Unlock it
         * 2. Destroy it (make it uninitialized)
         * 3. Free it's memory
         */
        if (--fp->f_count == 0)
        {
            pthread_mutex_unlock(&fp->f_lock);
            pthread_mutex_destroy(&fp->f_lock);
            free(fp);
        }

        else
        {
            pthread_mutex_unlock(&fp->f_lock);
        }
    }

    void
    empty_foo(struct foo * fp)
    {
        int total = fp->f_count;

        for (int i = 0; i < total; i++)
        {
            foo_release(fp);
        }
    }

    void *
    thread_func(void * arg)
    {
        for (int i = 0; i < 100; i++)
        {
            foo_hold(arg);
        }
    }

    int
    main(void)
    {
        struct foo * fp = foo_alloc();
        int       status;
```

```
    pthread_t tid1, tid2;

    // returns 0 upon success. I'm just not catching
    // these errors for simplicity's sake.
    status = pthread_create(&tid1, NULL, thread_func, fp);
    status = pthread_create(&tid2, NULL, thread_func, fp);

    if (pthread_join(tid1, NULL) != 0)
    {
        printf("error: thread 1\n");
        exit(EXIT_FAILURE);
    }

    // Same thing as above with different syntax
    if (pthread_join(tid2, NULL))
    {
        printf("error: thread 2\n");
        exit(EXIT_FAILURE);
    }

    printf("200: %d\n", fp->f_count);

    // Unlock, Destroy, and Free structure memory
    empty_foo(fp);
    // Now has junk data
    printf("Destroyed: %d\n", fp->f_count);

    exit(EXIT_SUCCESS);
}
```