

Παράλληλη Επεξεργασία

Εαρινό Εξάμηνο 2023-24
«Νήματα Προτύπου POSIX»

Παναγιώτης Χατζηδούκας, Ευστράτιος Γαλλόπουλος

Outline

- POSIX Threads API
 - Thread management
 - Synchronization with mutexes
 - Deadlock and thread safety
 - Barriers
 - Condition variables
 - **Exercises – Examples**
 - Special topics (quick overview)
 - Attribute objects
 - Once-only initialization
 - Thread-private data

Processes vs POSIX Threads

	Processes	POSIX Threads
Identifier	pid_t getpid()	pthread_t pthread_self()
Creation	fork()	pthread_create()
Start	exec()	
Wait	wait()	pthread_join()
Exit	exit()	pthread_exit()

POSIX Threads (Pthreads)

- Standardized C language threads programming interface
 - <http://pubs.opengroup.org/onlinepubs/9699919799/>
- Header file:
`#include <pthread.h>`
- Compilation
`$ gcc -pthread -o hello hello.c`
- Execution
`$./hello`

Spawning and Joining Threads

```
void *func(void *arg)
{
    sleep(1);
    return NULL;
}

int main(int argc, char * argv[])
{
    pthread_t id[4];

    for (long i = 0; i < 4; i++)    {
        pthread_create(&id[i], NULL, func, NULL);
    }

    for (long i = 0; i < 4; i++)    {
        pthread_join(id[i], NULL);
    }

    return 0;
}
```

Creating and Joining Threads

```
long x=5;

void *func(void * arg)
{
    long sec = *((long *)arg);
    sleep(x+sec);
    return NULL;
}

int main(int argc, char * argv[])
{
    pthread_t id[4];
    long a[4];

    for (long i = 0; i < 4; i++) {
        a[i] = i;
        pthread_create(&id[i], NULL, func, (void *) &a[i]);
    }

    for (long i = 0; i < 4; i++) {
        pthread_join(id[i], NULL);
    }

    return 0;
}
```

Synchronization

- Consider the following code
 - `next_ticket` is a global variable initialized to 0
 - `ticket` is a local variable, private to each thread

```
ticket = next_ticket++;          /* 0 ⇒ 1 */
```

- In the general case, this is equivalent to the following:

```
ticket = temp = next_ticket;    /* 0 */
++temp;                          /* 1 */
next_ticket = temp;             /* 1 */
```

```
R1 ← next_ticket (load)
ticket ← R1      (store)
R1 ← R1+1       (inc)
next_ticket ← R1 (store)
```

Execution with 2 Threads

Thread 0

```
tkt = tmp = n_tkt; (0)
++tmp; (1)
n_tkt = tmp; (1)
```

Thread 1

```
tkt = tmp = n_tkt; (1)
++tmp; (2)
n_tkt = tmp; (2)
```

time ↓

Another Possible Case

Thread 0

`tkt = tmp = n_tkt; (0)`

`++tmp; (1)`

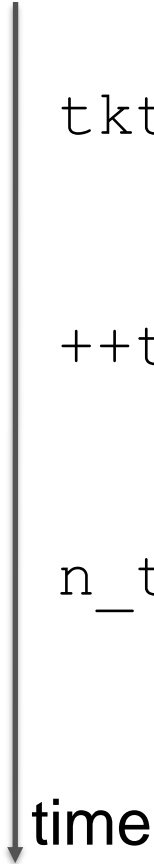
`n_tkt = tmp; (1)`

Thread 1

`tkt = tmp = n_tkt; (0)`

`++tmp; (1)`

`n_tkt = tmp; (1)`



What we observe here is a race condition

The update of the shared variable is a critical section and must be protected

Synchronization - Mutexes

- POSIX Threads include several synchronization mechanisms
 - Mutexes, Condition variables, Barriers, Reader-Writer locks, Spinlocks, Semaphores
- A Mutex (*Mutual Exclusion*) is a mechanism that allows multiple threads to synchronize their access to shared resources (e.g. variables)
 - A mutex has two states: *locked* and *unlocked*
 - Only one thread can lock the mutex
 - Once a mutex is locked, any other thread that tries to lock that mutex will suspend its execution, until the thread unlocks the mutex
 - At that point, one of the waiting threads acquires the mutex and continues its execution

Mutex Management

- Declaration and static initialization of a mutex:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

- Declaration and dynamic initialization of a mutex:

```
pthread_mutex_t mymutex;  
pthread_mutex_init(&mymutex, NULL);
```

- Locking (acquiring) the mutex:

```
pthread_mutex_lock(&mymutex);
```

- Unlocking (releasing) the mutex after the critical section:

```
pthread_mutex_unlock(&mymutex);
```

Dynamic Allocation of Mutex

- **Pointer to mutex**

```
pthread_mutex_t *mymutex;
```

- **Memory allocation:**

```
mymutex = (pthread_mutex_t *)  
          malloc(sizeof(pthread_mutex_t));
```

- **Mutex initialization:**

```
pthread_mutex_init(mymutex, NULL);
```

- **Locking and unlocking as before**

- **Destroying mutex:**

```
pthread_mutex_destroy(mymutex);
```

Usage

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
double global_sum = 0.0;

void *work(void *arg)
{
    ...
    pthread_mutex_lock(&mutex);
    global_sum += local_sum;
    pthread_mutex_unlock(&mutex);
    ...
}
```

Check and Lock

```
int pthread_mutex_trylock(pthread_mutex_t *m);
```

- Allows a thread to try to lock a mutex
- If the mutex is available then the thread locks the mutex
- If the mutex is locked then the function informs the user by returning a special value (EBUSY)
- This approach allows for implementations of spinlocks

```
while (pthread_mutex_trylock(&mutex) == EBUSY)  
    /*sched_yield()*/;
```

Mutexes - Recap

```
#include <pthread.h>
```

```
#include <stdlib.h>
```

```
pthread_mutex_t static_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_mutex_t *dynamic_mutex;
```

```
dynamic_mutex = (pthread_mutex_t *)  
                malloc(sizeof(pthread_mutex_t));
```

```
pthread_mutex_init(&dynamic_mutex, NULL);
```

```
pthread_mutex_destroy(&dynamic_mutex);
```

```
free(dynamic_mutex);
```

Example: Thread Number

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int temp_id = 0;

void *work(void *arg)
{
    int id;
    ...
    pthread_mutex_lock(&mutex);
    id = temp_id;
    temp_id = temp_id+1;
    pthread_mutex_unlock(&mutex);
    ...
}
```


Example: try lock

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

void * func(void *arg) {
    int res;
    res = pthread_mutex_trylock(&mutex1); /* EBUSY */;
    return 0;
}

int main() {
    pthread_t t;
    pthread_mutex_lock(&mutex1);
    pthread_create(&t, NULL, func, NULL);
    sleep(2);
    pthread_mutex_unlock(&mutex1);
    pthread_join(t, NULL);
    return 0;
}
```

Compute Pi - Sequential Version

```
long num_steps = 100000;
double step;

int main()
{
    double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for (int i=0; i <num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }

    pi = step * sum;

    return 0;
}
```

POSIX Threads Version

```
#define NUM_THREADS 2
pthread_t thread[NUM_THREADS];
pthread_mutex_t Mutex;
long num_steps = 100000;
double step;
double global_sum = 0.0;

void *Pi (void *arg) {
    long start;
    double x, sum = 0.0;

    start = (long) (*(int *) arg);
    step = 1.0/(double) num_steps;

    for(long i=start; i<num_steps;
        i+=NUM_THREADS) {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pthread_mutex_lock (&Mutex);
    global_sum += sum;
    pthread_mutex_unlock (&Mutex);

    return 0;
}
```

```
int main()
{
    double pi;

    int Arg[NUM_THREADS];

    for(int i=0; i<NUM_THREADS; i++)
        Arg[i] = i;

    pthread_mutex_init(&Mutex, NULL);

    for (int i=0; i<NUM_THREADS; i++)
        pthread_create(&thread[i], NULL,
            Pi, &Arg[i]);

    for (int i=0; i<NUM_THREADS; i++)
        pthread_join(thread[i], NULL);

    pi = global_sum * step;

    return 0;
}
```

Deadlocks

- Deadlock can occur when multiple mutexes are not locked in the same order.
- The threads cannot continue their execution:

Thread 0	Thread 1
<code>pthread_mutex_lock(&mut1);</code>	<code>pthread_mutex_lock(&mut2);</code>
<code>pthread_mutex_lock(&mut2);</code>	<code>pthread_mutex_lock(&mut1);</code>

- Deadlock can also occur if a mutex is locked twice (recursively) by the same thread

```
pthread_mutex_lock(&mut1);  
pthread_mutex_lock(&mut1);
```

Thread Safety

- A function is thread-safe if it can be called safely at the same time by multiple threads
- Reentrant function: can have multiple concurrent active calls, at different points, without side-effects
- Thread-safe function: reentrant or protected with mutual exclusion
- Functions that use static variables are not thread-safe, e.g.: `rand()`, `drand48()`

Solutions

1. Use a mutex inside the function
 - provides an easy but inefficient solution due to the serialization of the code
 - this is the case for `rand()`, `drand48()`
2. Use static (per thread) data as function arguments (e.g. `strtok_r`)
 - not convenient, exposes the internal implementation of the function, requires code rewriting
3. Use thread private data (`pthread_key_t`)
 - Every thread has its own private data that other threads cannot access

Example 1

- `rand_r`: thread-safe version of `rand()`
 - `randp` is assigned a number from 0 and `RAND_MAX`
 - returns 0 on success

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

int rand_r(int *ranp){
    static pthread_mutex_t = PTHREAD_MUTEX_INITIALIZER;
    int error;
    if (error = pthread_mutex_lock(&lock))
        return error;
    *ranp = rand();
    return pthread_mutex_unlock(&lock);
}
```

Example 2

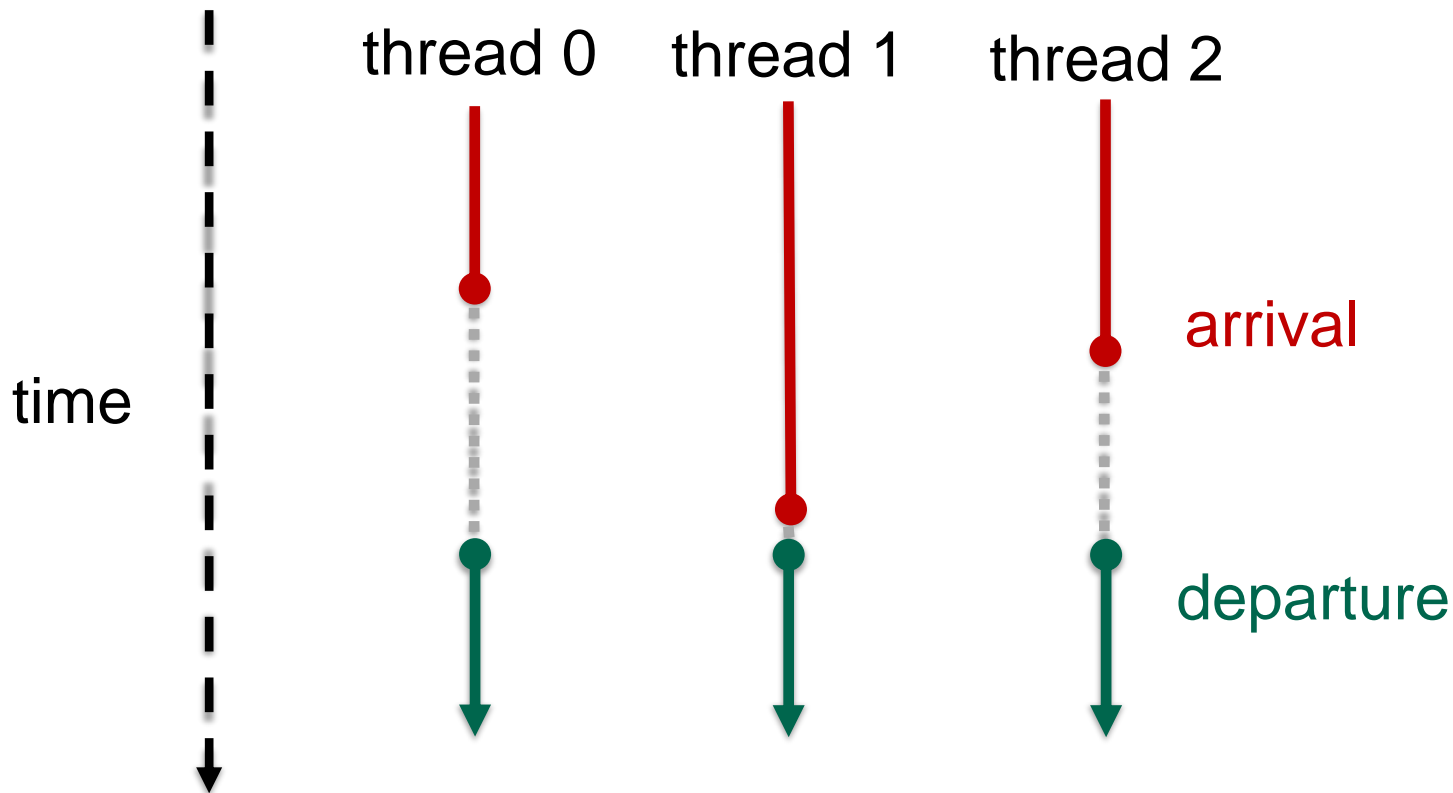
- `drand48()` vs `erand48()`
 - "return non-negative, double-precision, floating-point values, uniformly distributed over the interval [0.0 , 1.0]"
 - <http://pubs.opengroup.org/onlinepubs/7908799/xsh/drand48.html>

```
srand48(10); // initialization
double xi = drand48();
double yi = drand48();
```

```
unsigned short buf[3]; // random stream
buf[0] = 0; buf[1] = 0; buf[2] = 10; // initialization
double xi = erand48(buf);
double yi = erand48(buf);
```


Barriers

- Barrier: synchronization mechanism that makes sure all calling threads in a program
- No thread can cross the barrier until all the threads have reached it



Barriers

```
int pthread_barrier_init(pthread_barrier_t * bar,  
                        const pthread_barrierattr_t*attr,  
                        unsigned int count);  
  
int pthread_barrier_wait(pthread_barrier_t *bar);  
  
int pthread_barrier_destroy(pthread_barrier_t *bar);
```

- `pthread_barrier_t`: data type
- `count`: number of threads that must call the barrier

Barrier Example

```
pthread_barrier_t bar;

void *func(void *arg)
{
    int id = *((int *)arg);
    printf("Hello world from thread %d!\n", id);

    pthread_barrier_wait(&bar);
    sleep(id);

    printf("Goodbye world from thread %d!\n", id);

    return NULL;
}

int main(int argc, char * argv[])
{
    ...
    pthread_barrier_init(&bar, NULL, 4);
    ...
}
```

Condition Variables

- Synchronization mechanism that allows multiple threads to wait on a condition and resume their execution after some time
- Mutexes protect critical code but condition variable protect a more general operation
- A thread waits on a condition variable until it is informed (by the variable) that it can continue
- Another (running) thread signals the condition variable, allowing other threads to continue
- Every condition variable, as shared data, is associated with a specific mutex

Condition Variables

- No need for continuous checking of data
- A thread that modifies the values of data also informs the other threads through the condition variable
- Example: producer-consumer scheme using data queue
 - The consumer thread extracts and processes elements of the queue while this is not empty, otherwise the thread suspends its execution
 - The producer thread adds elements to the queue and informs, through the condition variable, the consumer thread
 - The interaction is based on the number of elements in the queue

Declaration and Initialization

```
int pthread_cond_init(pthread_cond_t * cond,  
                      const pthread_condattr_t *attr);
```

- `pthread_cond_t`: data type

- **Static initialization:**

```
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
```

- **Dynamic initialization:**

```
pthread_cond_init(&condition, NULL);
```

- There must be always an associated mutex

Waiting

```
int pthread_cond_wait(pthread_cond_t *condition,  
                      pthread_mutex_t *mutex);
```

- The thread suspends its execution until the condition variable is signaled
- The thread must have locked the associated mutex
- The thread is suspended and the mutex is unlocked automatically in order to be used by other threads
- When the condition variable is signaled and the suspended thread wakes up, it automatically locks the mutex

Signaling

```
int pthread_cond_signal(pthread_cond_t *cond);  
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- `pthread_cond_signal`: a thread informs a single thread that waits on the condition variable
- `pthread_cond_broadcast`: a thread informs all the threads that wait on the condition variable
- The signaling thread has usually locked the associated mutex. In that case, it must release the mutex in order to allow `pthread_cond_wait` to return

Usage

1. A thread locks the associated mutex `m`
2. The thread checks the condition and decides if it can continue or not
3. If it can continue (the condition is true):
 1. The thread executes its operation and
 2. Unlocks the mutex
4. If it cannot continue (the condition is false):
 1. The thread suspends itself by calling `pthread_cond_wait(&c,&m)`, and the mutex is automatically unlocked (by the system)
 2. Another thread sets the condition to true and calls `pthread_cond_signal(&c)`
 3. The thread wakes up with the mutex locked by itself and executes its operation
 4. The thread unlocked the mutex when it completes its operation

Example 1

Thread 0

```
T1: pthread_mutex_lock(&m);  
T2: while(!condition_ok)  
T3:   while(pthread_cond_wait(&c, &m));  
T4:  
T5:  
T6:  
T7:  
T8: go_ahead_and_do_it();  
T9: pthread_mutex_unlock(&m);
```

Thread 1

```
pthread_mutex_lock(&m);  
condition_ok = TRUE;  
pthread_cond_signal(&c);  
pthread_mutex_unlock(&m);
```

Example 2

Thread 0

Thread 1

```
for (...)
{
    -----
    pthread_mutex_lock(&mut);
    count++;
    if (count== N) {
        /* Wake-up thread 1 */
        pthread_cond_signal(&cond);
    }
    pthread_mutex_unlock(&mut);
    -----
    //Do work
}
```

```
-----
pthread_mutex_lock(&mut);
while(count<N) {
    pthread_cond_wait(&cond, &mut);
}
pthread_mutex_unlock(&mut);
-----

//Do work
```

Example 3

Threads 0,...,N-1

Thread N

```
for (...)
{
    -----
    pthread_mutex_lock(&mut);
    count++;
    if (count== N) {
        /* Wake-up thread N */
        pthread_cond_signal(&cond);
    }
    pthread_mutex_unlock(&mut);
    -----
    //Do work
}
```

```
-----
pthread_mutex_lock(&mut);
while(count<N) {
    pthread_cond_wait(&cond, &mut);
}
pthread_mutex_unlock(&mut);
-----

//Do work
```

Example 4

Thread 0

```
for (...)
{
    -----
    pthread_mutex_lock(&mut);
    count++;
    if (count== N){
        /* Wake-up threads 1..N */
        pthread_cond_broadcast(&cond);
    }
    pthread_mutex_unlock(&mut);
    -----
    //Do work
}
```

Thread 1,2,...,N

```
-----
pthread_mutex_lock(&mut);
while(count<N) {
    pthread_cond_wait(&cond,&mut);
}
pthread_mutex_unlock(&mut);
-----
//Do work
```

Comments

- Condition variables are used for signaling and not mutual exclusion
- The condition must be always checked if it is true or not
 - Interrupted wake-ups: Before `pthread_cond_wait` returns with the mutex locked, another thread locks the mutex and changes the condition
 - Loose conditions: an approximation of the real condition (e.g. “there might be available work”)
 - False wakeups: `pthread_cond_wait` returns but the condition variable has not been signaled (e.g. in multicore systems)

Condition Variables

- Condition variables can be used without a condition
- The mutex must be locked and released before and after `wait()`, respectively
- Example:

```
pthread_mutex_lock(&mut);  
pthread_cond_wait(&cond, &mut);  
pthread_mutex_unlock(&mut);
```

Condition Variables - Extension

```
int pthread_cond_timedwait(&condition, &mutex,  
                           (struct timespec *)expiration);
```

- Waits on a condition variable but with time constraint. If timeout occurs, the function returns ETIMEDOUT.

```
=====
```

```
struct timespec timeout;
```

```
/* Wait on the condition variable for 2 seconds */
```

```
timeout.tv_sec = time(NULL) + 2;
```

```
timeout.tv_nsec = 0;
```

```
pthread_cond_timedwait(&condition, &mutex, &timeout);
```


Fibonacci (1/3)

```
/* version 1*/  
int fib(int n)  
{  
    if (n <= 1) return n;  
    else return fib(n-1) + fib(n-2);  
}
```

```
/* version 2*/  
void fib(int n, int *res)  
{  
    int r1, r2;  
    if (n <= 1) *res = n;  
    else {  
        fib(n-1, &r1);  
        fib(n-2, &r2);  
        *res = r1 + r2;  
    }  
}
```

Fibonacci (2/3)

```
/* version 3*/
struct fib_arg {
    int n;
    int res;
};

void fib(struct arg *a)
{
    struct fib_arg a1, a2;
    int r1, r2;
    if (a->n <= 1) a->res = a->n;
    else {
        a1.n = a->n-1;
        a2.n = a->n-2;
        fib(&a1);
        fib(&a2);
        a->res = a1.res + a2.res;
    }
}
```

Fibonacci (3/3)

```
/* version 4*/
void *fib(void *arg) {
    struct fib_arg *a = (struct fib_arg *) arg;
    struct fib_arg a1, a2;
    pthread_t t1, t2;

    pthread_mutex_lock(&nthreads_mutex);
    nthreads++;
    pthread_mutex_unlock(&nthreads_mutex);

    if (a->n <= 1) {
        a->res = a->n;
    }
    else {
        a1.n = a->n-1;
        a2.n = a->n-2;
        pthread_create(&t1, NULL, fib, (void *) &a1);
        pthread_create(&t2, NULL, fib, (void *) &a2);
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
        a->res = a1.res + a2.res;
    }
    return NULL;
}
```

Execution Order (1/2)

```
#define N 10

void *routine(void *arg) {
    int id = (int) arg;
    printf("Hello from thread:%d\n", id);

    return 0;
}

int main() {
    int i;
    pthread_t t[N];
    int a[N];

    for (i = 0; i < N; i++) {
        a[i] = i;
        pthread_create(&t[i], NULL, routine, (void *)&a[i]);
    }

    for (i = 0; i < N; i++)
        pthread_join(t[i], NULL);

    return 0;
}
```

Execution Order (2/2)

```
pthread_mutex_t m[N]; /* initialized in main */
pthread_cond_t c[N]; /* initialized in main */

void *routine(void *arg)
{
    int id = *((int *) arg);

    if (id > 0) {
        pthread_mutex_lock(&m[id]);
        pthread_cond_wait(&c[id], &m[id]);
    }

    printf("Hello from thread:%d\n", id);

    if (id > 0) pthread_mutex_unlock(&m[id]);
    if (id < N) pthread_cond_signal(&c[id+1]);

    return 0;
}
```

Recursive Locks (1/4)

```
int  othr_init_nest_lock(othr_nest_lock_t *lock);
int  othr_destroy_nest_lock(othr_nest_lock_t *lock);
int  othr_set_nest_lock(othr_nest_lock_t *lock);
int  othr_unset_nest_lock(othr_nest_lock_t *lock);
```

```
typedef struct {
    pthread_mutex_t lock;          /* real lock */
    pthread_mutex_t ilock;        /* data lock */
    pthread_cond_t cond;
    int count;
    othr_t owner;
} othr_nest_lock_t;
```

Recursive Locks (2/4)

```
int othr_init_nest_lock(othr_nest_lock_t *lock)
{
    pthread_mutex_init(&lock->iunlock, NULL);
    pthread_mutex_init(&lock->lock, NULL);
    lock->count = 0;
    pthread_cond_init(&lock->cond, 0);
    return (0);
}
```

```
int othr_destroy_nest_lock(othr_nest_lock_t *lock)
{
    pthread_mutex_destroy(&lock->lock);
    pthread_cond_destroy(&lock->cond);
    pthread_mutex_destroy(&lock->iunlock);
    return (0);
}
```

Recursive Locks (3/4)

```
int othr_set_nest_lock(othr_nest_lock_t *lock)
{
    pthread_mutex_lock(&lock->iunlock);
    if (pthread_mutex_trylock(&lock->lock) == 0)    /* lock it */
    {
        lock->owner = pthread_self();                /* Get ownership */
        lock->count++;
    }
    else
        if (pthread_equal(lock->owner, pthread_self())) /* mine */
            lock->count++;
        else /* someone else */
        {
            while ( pthread_mutex_trylock(&lock->lock) )
                pthread_cond_wait(&lock->cond, &lock->iunlock);
            lock->owner = pthread_self();
            lock->count++;
        }
    pthread_mutex_unlock(&lock->iunlock);
    return (0);
}
```


Recursive Locks (4/4)

```
int othr_unset_nest_lock(othr_nest_lock_t *lock)
{
    pthread_mutex_lock(&lock->iunlock);
    if (pthread_equal(lock->owner, pthread_self()) && lock->count > 0)
    {
        lock->count--;
        if (lock->count == 0)
        {
            pthread_mutex_unlock(&lock->lock);
            pthread_cond_signal(&lock->cond);
        }
    }
    pthread_mutex_unlock(&lock->iunlock);
    return (0);
}
```

Special topics

- Attributes objects
 - Thread attributes
 - Mutex attributes
- One-time initialization
- Thread private data

Attribute objects

- Thread creation and mutex, barrier and condition variable initialization routines had the second argument equal to NULL (so far)
- This argument is a pointer to attribute object
- NULL corresponds to the default attribute value
- Attribute object: some kind of private data structure (like C++)
- The members of the object can be accessed through appropriate routines, e.g.:

```
pthread_attr_getstacksize  
pthread_attr_setstacksize
```

Thread Attributes

- Declaration, initialization and destruction of a thread attribute object

```
pthread_attr_t attr;  
int pthread_attr_init(&attr);  
int pthread_attr_destroy(&attr);
```

- Detach state of thread

```
int pthread_attr_getdetachstate(&attr, (int *) &ds);  
int pthread_attr_setdetachstate(&attr, (int) ds);
```

```
PTHREAD_CREATE_JOINABLE: joinable thread  
PTHREAD_CREATE_DETACHED: daemon thread
```

Thread Attributes

- **Stack size of the thread, minimum is PTHREAD_STACK_MIN**

```
int pthread_attr_getstacksize(&attr, (size_t *) &ss);  
int pthread_attr_setstacksize(&attr, (size_t) ss);
```
- **Stack of a thread, allocated explicitly (e.g. malloc)**

```
int pthread_attr_getstackaddr(&attr, (void **) &sa);  
int pthread_attr_setstacksize(&attr, (void *) sa);
```

Mutex Attributes

- Declaration, initialization and destruction of a mutex attribute object

```
pthread_mutexattr_t attr;  
int pthread_mutexattr_init(&attr);  
int pthread_mutexattr_destroy(&attr);
```

- **Mutex types**

```
int pthread_mutexattr_gettype(&attr, (int *) &type);  
int pthread_mutexattr_settype(&attr, (int ) type);
```

- PTHREAD_MUTEX_NORMAL: **faster implementation**
- PTHREAD_MUTEX_RECURSIVE: **can be locked several times by the same thread**
- PTHREAD_MUTEX_ERRORCHECK: **for debugging**
- PTHREAD_MUTEX_DEFAULT: **usually the first one above**

Once-Only Initialization

- Data initialization (e.g., mutexes) within main()
- Software libraries do not have this option
- A boolean variable that is set when the initialization has been performed is enough for single-threaded applications
- Static initialization of mutexes does not cause any issues
- Dynamic initialization of mutexes requires `pthread_once`

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;  
int pthread_once(pthread_once_t *once_control,  
                void (*init_routine)(void));
```

- Any thread can call the `pthread_once` function
- The associated variable is first checked, and the initialization routine is then called
- Any other thread waits for the completion of the initialization routine

Once-Only Initialization

```
pthread_once_t once_block = PTHREAD_ONCE_INIT;
pthread_mutex_t mutex;

/* Initialization routine */
void once_init_routine(void) {
    pthread_mutex_init(&mutex, NULL);
}

/* Thread start routine that calls pthread_once */
void *thread_routine(void *arg) {
    pthread_once(&once_block, once_init_routine);
    ...
}

int main() {
    pthread_create(&t, NULL, thread_routine, NULL);
    pthread_once(&once_block, once_init_routine);
    ...
}
```


Thread private data

- Static variables, declared in a file or function, can be accessed by all threads
- Only the register values are private to a thread. Even its stack can be accessed by other threads
- Goal: maintain the value of a variable between calls of different routines on a particular thread
- Creation and destruction of key for management of thread private data:

```
pthread_key_t key;  
int pthread_key_create(pthread_key_t *key,  
                      void (*destructor)(void *));  
int pthread_key_delete(pthread_key_t key);
```

- Access of key values (set, get):

```
int pthread_setspecific(pthread_key_t key, const void *value);  
void *pthread_getspecific(pthread_key_t key);
```

Thread private data

```
pthread_key_t key;
pthread_once_t key_once = PTHREAD_ONCE_INIT;

void once_routine(void) {
    pthread_key_create(&key, NULL);
}

void routine() {
    int *value = pthread_getspecific(key);
}

void *func(void *arg) {
    int id = *((int *)arg);
    int *value;
    pthread_once(&key_once, once_routine);

    value = malloc(sizeof(int));
    *value = id;
    pthread_setspecific(key, value);
    routine();
    return NULL;
}
```

References

- Advanced Programming in the Unix Environment, W. Richard Stevens
- Programming with POSIX Threads, David R. Butenhof
 - www.openmp.org
- POSIX threads tutorial at LLNL, Blaise Barney
 - <https://computing.llnl.gov/tutorials/pthreads/>