# *UNIX System Programming*

Roee Leon

DECEMBER 7, 2016

Copyright © 2016 **Roee Leon.** All Rights Reserved.

## Threads - Agenda

- Introduction to Threads
- Thread Concepts
- Thread Identification
- Thread Creation
- Thread Termination
- Thread Synchronization

Copyright © 2016 **Roee Leon.** All Rights Reserved.

## Threads – Introduction to Threads

- We have covered the ways of which processes can be controlled and manipulated
  - *exec/fork/wait/waitpid*
- Each process is a **unique** identity that is composed of a single execution unit
- We have seen that sharing data between processes is very limited as each process has its own separate memory layout
  - Yet, allocated memory can be shared using the *mmap* library call marked as *MAP_SHARED*
- Threads allow us to split a process into multiple execution units that can perform tasks within the environment of a single process
  - All the threads share the same process components (e.g. file descriptors & memory)

Copyright © 2016 **Roee Leon.** All Rights Reserved.

## Threads – Thread Concepts

- As mentioned, a process is composed of a single execution unit
    - Can perform only a single task a time
- Using multiple threads, our programs can be designed to perform multiple-tasks at the same time within a single process
- This approach includes several benefits:
    - Handling of asynchronous events
    - Memory/File-Descriptor sharing
    - Performance (Interleaving multiple tasks)
    - Interactive programs (Threads to handle user input, graphics, etc.)

## Threads – Thread Concepts

- Does multithreaded programming has anything to do with multiprocessor?
  - Not really
  - The benefits of multithreaded programming can be observed even if a program is running on a uniprocessor
    - e.g. when a program needs to block, some threads may continue to run while others are blocked
- A thread consists of all the information necessary to represent an execution context within a process:
  - Thread ID
  - Set of registers values
  - A stack
  - Priority
  - Policy
  - *errno* variable

Copyright © 2016 **Roee Leon.** All Rights Reserved.

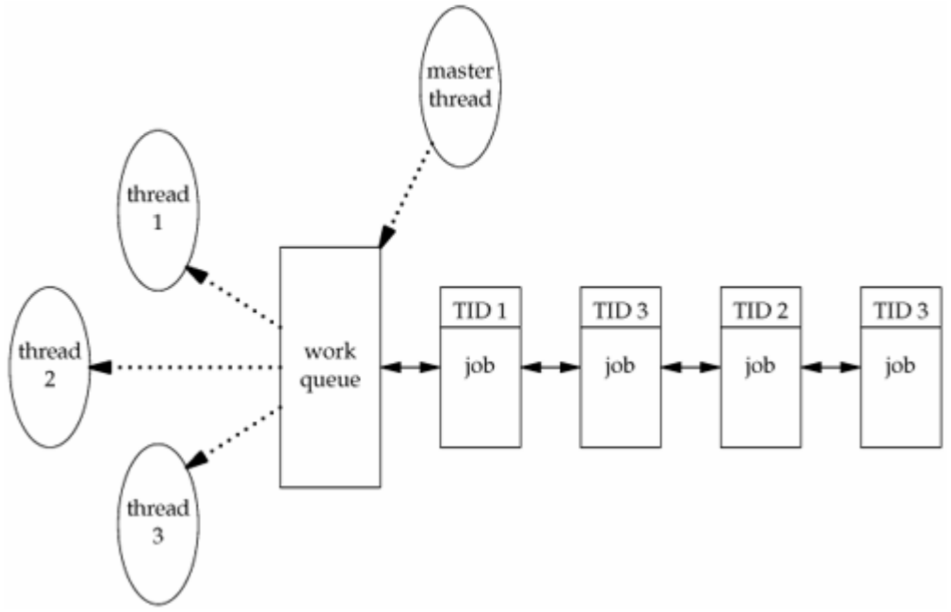
# Threads – Thread Identification

- Just as processes, every thread has a thread ID
  - Unlike the process ID, which is unique to the system, the thread ID is unique only to the process to which it belongs
- A thread ID, other than process ID, is represented as an object of type *pthread_t* which is not guaranteed to be an integer (in contrast to the process ID)
  - Therefore, if desired, comparing two thread IDs can be done using the following library function:
    - int pthread_equal(pthread_t tid1, pthread_t tid2);
      - On Linux, the *pthread* type is defined to be an *unsigned long integer*
      - Returns non-zero if equal and zero otherwise

Copyright © 2016 **Roee Leon.** All Rights Reserved.

## Threads – Thread Identification

- A thread can obtain its own thread ID by calling the library function:
  - pthread_t pthread_self(void);
- For example, this function can be used along with *pthread_equal* in case of a master thread that place tasks within a shared queue. Each task is assigned a specified thread ID
  - A thread can check whether the task belongs to him by comparing the task *pthread_t* with his own



## Threads – Thread Creation

- Initially, each process is made up of only one execution unit
  - i.e. only one thread
- Until a process "decide" to create more execution units (i.e. more threads), its behavior should be exactly the same as the single process model
- A thread can be created by a process using the *pthread_create* library function

## Threads – Thread Creation

- int pthread_create(pthread_t *restrict tidp, const pthread_attr_t *restrict attr, void *(*start_rtn)(void), void *restrict arg);
  - Returns zero on success, non-zero on failure
- The first parameter, *tidp*, is an output argument which will point to the newly created thread's thread-ID (in case of a success)
- The second parameter, *attr*, is used to customize the thread attributes
- The *third* parameter, *start_rtn*, is the address of the start routine of the newly created thread (analogical to the *main* function of a process)
- The *fourth* parameter, *arg*, is a type-less argument to the start routine (*start_rtn*)

Copyright © 2016 **Roee Leon.** All Rights Reserved.

## Threads – Thread Creation

- When a thread is created, there is no guarantee who will run first; The newly created thread or the creating (calling) thread
- The newly created thread has full access to the memory layout of the calling process
- The set of pending-signals to the thread are cleared

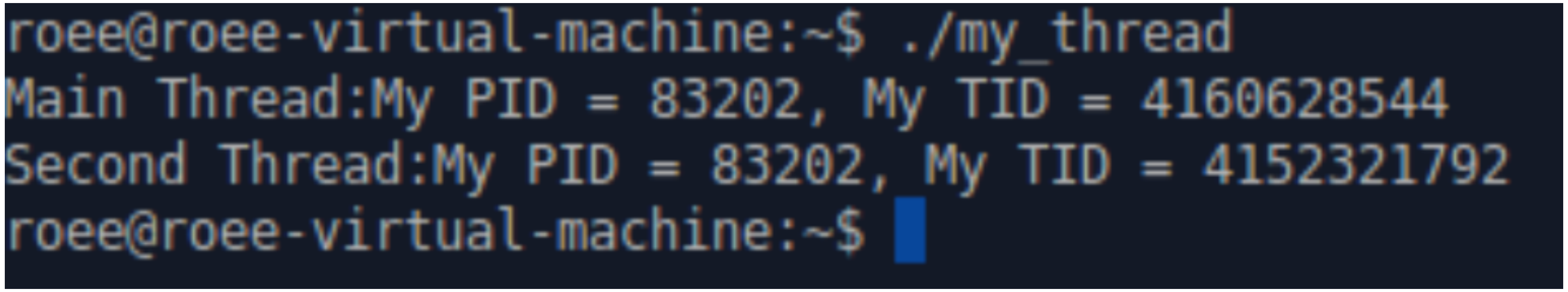
Copyright © 2016 **Roee Leon.** All Rights Reserved.

## Threads – Thread Creation - Example

```
#include <pthread.h>
#include <stdio.h>
pthread_t my_thread;
void print_my_ids(void)
{
        pid_t pid;
        pthread_t tid;
        pid = getpid();
        tid = pthread_self();
        printf("My PID = %u,
        My TID = %u\n", pid,
        (unsigned)tid);
}
```

```
void* foo (void* arg)
{
        printf("Thread:%u\n", my_thread);
        print_my_ids();
        return NULL;
}
int main (int argc, const char* argv[])
{
        if (pthread_create(&my_thread, NULL, foo,
        NULL))
                return 1;
        printf("Main Thread:");
        print_my_ids();
        sleep(2);
        return 0;
}
```

## Threads – Thread Creation - Example



```
roee@roee-virtual-machine:~$ ./my_thread
Main Thread:My PID = 83202, My TID = 4160628544
Second Thread:My PID = 83202, My TID = 4152321792
roee@roee-virtual-machine:~$
```

## Threads – Thread Termination

- Calling the *exit/_Exit* library from any of the process's threads will terminate the entire process
- A signal sent to a thread will also terminate the entire process (e.g. keyboard interrupt)
- A single thread may exit in three different ways:
  - Returning from its start routine
  - Cancelled by another thread in the same process
  - Calling *pthread_exit*
    - void pthread_exit(void *rval_ptr);
      - The *rval_ptr* argument specifies a location in memory that stores the thread's return value

# Threads – Thread Termination

- The return value from *pthread_exit* may be retrieved by another thread in the process by calling *pthread_join*
  - int pthread_join(pthread_t thread, void **rval_ptr);
- The waiting/blocking thread will block until the specified process (given by the *thread* argument) terminates
  - Returns from its start routine, calls *pthread_exit* or is cancelled
  - In case of returning from the start routine, the *rval_ptr* simply sets to the returned value
  - In case of a call to *pthread_exit*, *rval_ptr* is set to the *rval_ptr* argument of *pthread_exit*
  - In case of cancellation, the *rval_ptr* is set to the value *PTHREAD_CANCELED*
- If the given thread has already terminated, the *pthrad_join* function returns immediately

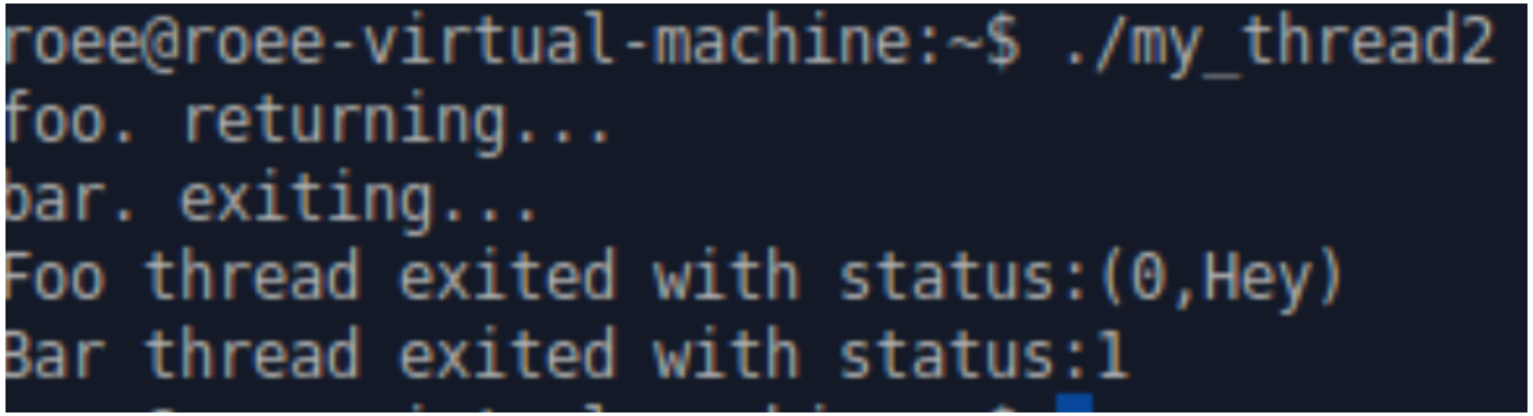
# Threads – Thread Termination - Example

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


struct _foo_struct
{
int some_status;
char* some_string;
}foo_arg;


void* foo(void* arg)
{
printf("foo. returning...\n");
foo_arg.some_status = 0;
foo_arg.some_string = "Hey";
return &foo_arg;
}
```

```
void* bar(void* arg)
{
printf("bar. exiting...\n");
pthread_exit((void*)1);
}


int main (int argc, const char* argv[])
{
pthread_t tid1,tid2;
void* ret;
struct _foo_struct* fs;
if (pthread_create(&tid1, NULL, foo,
NULL))
return 1;
if (pthread_create(&tid2, NULL, bar,
NULL))
return 1;
```

```
if (pthread_join(tid1, &ret))
{
  fprintf(stderr,"Could not join Foo
Thread\n");
  return 1;
}
fs = (struct _foo_struct*)ret;
printf("Foo thread exited with
status:(%d,%s)\n", fs->some_status,
fs->some_string);
if (pthread_join(tid2, &ret))
{
  fprintf(stderr,"Could not join with
Bar Thread\n");
  return 1;
}
printf("Bar thread exited with
status:%d\n", (int)ret);
return 0;
}
```

## Threads – Thread Termination - Example



```
roee@roee-virtual-machine:~$ ./my_thread2
foo. returning...
bar. exiting...
Foo thread exited with status:(0,Hey)
Bar thread exited with status:1
```

## Threads – Thread Termination

- A thread within a process can request another thread within the same process to terminate by calling *pthread_cancel*
  - int pthread_cancel(pthread_t tid);
    - Returns zero in case of a success and non-zero in case of a failure
- All *pthread_cancel* does is requesting the thread specified by *tid* to terminate
  - By default, the requested thread will behave as if it had called *pthread_exit* with an argument of *PTHREAD_CANCELED* (i.e. *pthread_exit((void*)PTHREAD_CANCELED))*
  - *pthread_cancel* does not wait (i.e. does not block) until the requested thread is terminated

## Threads – Thread Termination

- A thread can request for a set of functions to be called when it terminates
    - Sort of Thread Cleanup handlers (i.e. destructors)
- These handlers are stored on the thread's stack and therefore are called in reverse order on which they were pushed
- A thread may push a termination handler by calling *pthread_cleanup_push*
    - void pthread_cleanup_push(void (*rtn)(void *), void *arg);
        - Pushes the address of the function *(rtn)* onto the stack
        - This function will be called when a thread performs one of the following actions:
            - Calls *pthread_exit*
            - Responds to a cancellation request (Sent by another thread)
            - Calls the *pthread_cleanup_pop* library function with a non-zero argument
                - void pthread_cleanup_pop(int execute);
                - Simply pops the last pushed routine and executes it

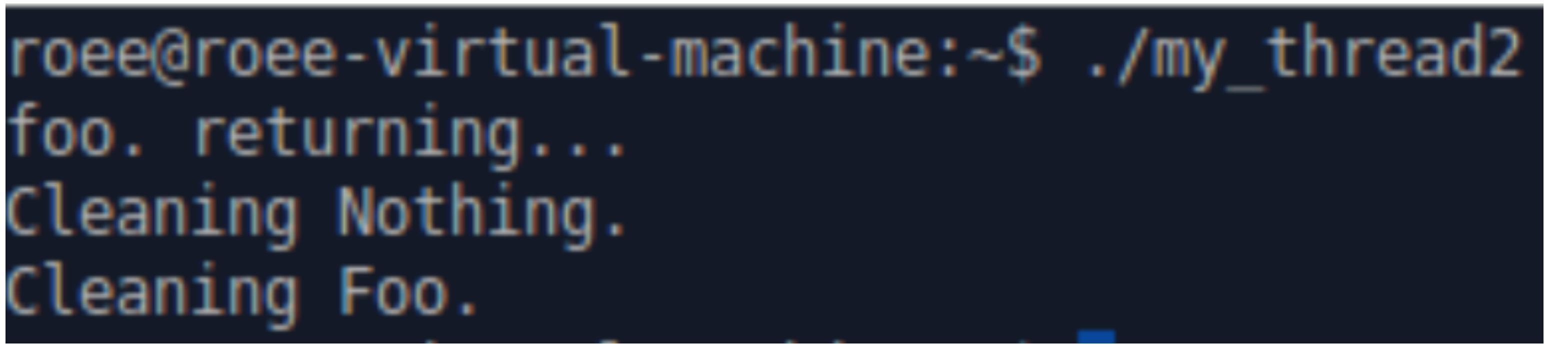
## Threads – Thread Termination - Example

```
#include <pthread.h>
#include <stdio.h>
struct _foo_struct
{
  int some_status;
  char* some_string;
}foo_arg;

void clean_foo(void* arg)
{
  printf("%s\n", (char*)arg);
  foo_arg.some_status = 0;
  foo_arg.some_string = NULL;
}

void clean_nothing(void* arg)
{
  printf("%s\n",(char*)arg);
}
```

```
void* foo(void* arg)
{
  pthread_cleanup_push(clean_foo,
"Cleaning Foo.");
  pthread_cleanup_push(clean_nothing,
"Cleaning Nothing.");
  printf("foo. returning...\n");
  foo_arg.some_status = 10;
  foo_arg.some_string = "Hey";
// ...
  pthread_cleanup_pop(2);
  pthread_cleanup_pop(2);
return NULL;
}
```

```
int main (int argc, const char* argv[])
{
pthread_t tid1,tid2;
void* ret;
struct _foo_struct* fs;
if (pthread_create(&tid1, NULL, foo,
NULL))
return 1;
if (pthread_join(tid1, &ret))
{
   fprintf(stderr,"Could not join Foo
Thread\n");
   return 1;
}
return 0;
}
```

## Threads – Thread Termination - Example



```
roee@roee-virtual-machine:~$ ./my_thread2
foo. returning...
Cleaning Nothing.
Cleaning Foo.
```

# Threads – Thread Synchronization

- As mentioned, threads within a process share the same memory
  - Therefore, we must ensure that the manipulated data remains consistent
  - If the manipulated data is thread specific (i.e. local variable), no consistency problems exist
  - If one thread may read from a data location while another thread may write into it,  the access must be synchronized to ensure only valid data is accessed
- For example, if the target processor's write operation takes more than one cycle, memory read/writes can be interleaved
- Lets see an example …

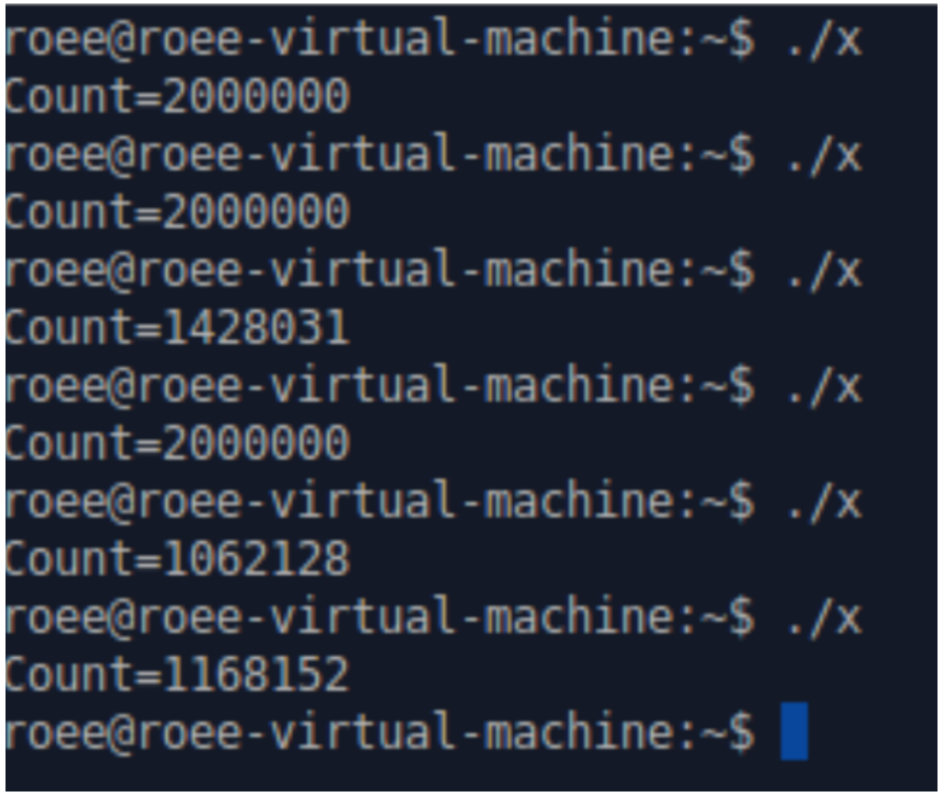
Copyright © 2016 **Roee Leon.** All Rights Reserved.

## Threads – Thread Synchronization - Example

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


void* inc_count_thread(void* f)
{
  int i, *count;
  count = (int*)f;
  for (i = 0; i < 1000000; ++i)
      ++(*count);
  return NULL;
}
```

```
int main (int argc, const char* argv[])
{
    int count = 0;
    pthread_t tid1,tid2;
    if (pthread_create(&tid1, NULL,
inc_count_thread, &count))
    return 1;
   if (pthread_create(&tid2, NULL,
inc_count_thread, &count))
    return 1;


    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("Count=%d\n",count);
    return 0;
}
```

## Threads – Thread Synchronization - Example



```
roee@roee-virtual-machine:~$ ./x
Count=2000000
roee@roee-virtual-machine:~$ ./x
Count=2000000
roee@roee-virtual-machine:~$ ./x
Count=1428031
roee@roee-virtual-machine:~$ ./x
Count=2000000
roee@roee-virtual-machine:~$ ./x
Count=1062128
roee@roee-virtual-machine:~$ ./x
Count=1168152
roee@roee-virtual-machine:~$ 
```

Copyright © 2016 **Roee Leon.** All Rights Reserved.

## Threads – Thread Synchronization - Mutexes

- Multiple threads shared data can be protected using pthreads mutual-exclusion interfaces (Mutexes).
- A mutex is basically a lock that needs to be set (locked) before a shared data is accessed and reset (unlocked) after it is accessed
- While a mutex is set, any other thread that will try to set the mutex will be blocked until it resets (i.e. unlocked)
  - The first thread to run will set (lock) the mutex while the other will go on blocking until it resets again

## Threads – Thread Synchronization - Mutexes

- A mutex variable is represented by the data type *pthread_mutex_t*
- Before a mutex variable can be used, it must be initialized by calling the *pthread_mutex_init* function or by allocating statically with the prefix *PTHREAD_MUTEX_INITIALIZER*
- In case a mutes is allocated dynamically (e.g. using *malloc*), it must be destroyed by calling *pthread_mutex_destroy* before it is freed

  - int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
  - int pthread_mutex_destroy(pthread_mutex_t *mutex);

# Threads – Thread Synchronization - Mutexes

- A thread can lock a mutex by calling the *pthread_mutex_lock* library function
  - int pthread_mutex_lock(pthread_mutex_t *mutex);
  - In case the mutex is locked at the time *pthread_mutex_lock* is called, the thread blocks until it is unlocked
- A thread can unlock a mutex by calling the *pthread_mutex_unlock* library function
  - int pthread_mutex_unlock(pthread_mutex_t *mutex);
- A thread can lock a mutex asynchronously (i.e. without blocking) by calling the *pthread_mutex_trylock* library function
  - int pthread_mutex_trylock(pthread_mutex_t *mutex);
  - If the mutex is unlocked at the time of calling, the mutex becomes locked and the return value is 0
  - If the mutex is locked at the time of calling, the function returns instantly with a failure status of *EBUSY*

## Threads – Thread Synchronization - Mutexes

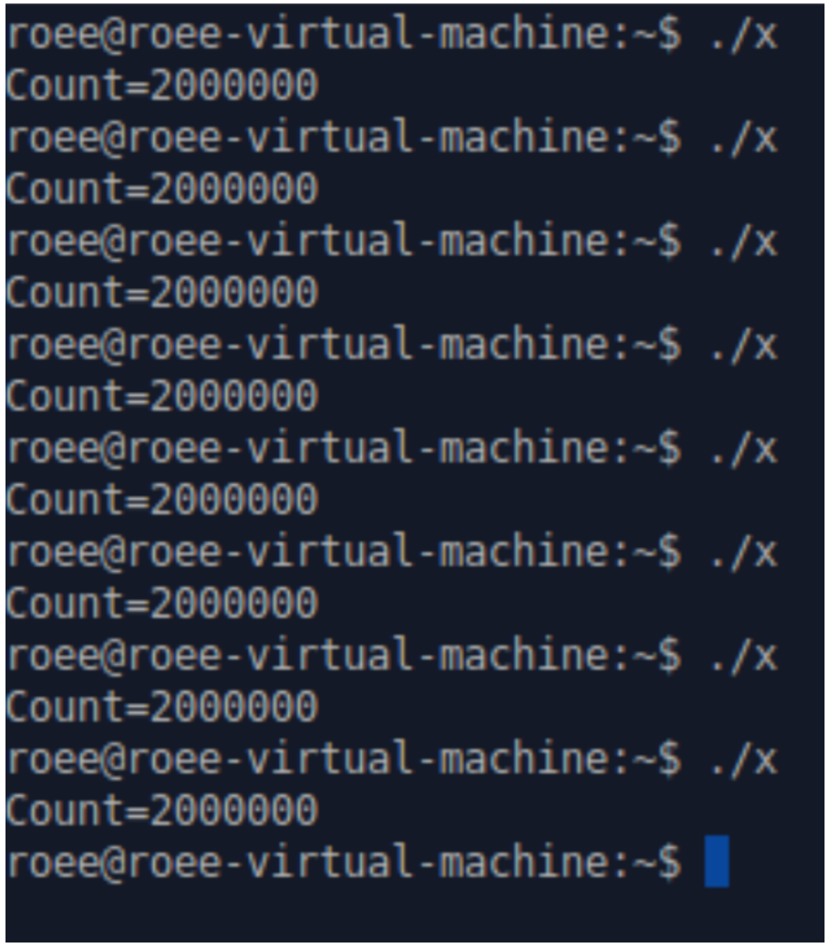
- Back to our previous example..
- This time we will use *mutexes* to provide mutual exclusion
- Lets see it..

# Threads – Thread Synchronization - Example

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

pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER

void* inc_count_thread(void* f)
{
  int i, *count;
  count = (int*)f;
  for (i = 0; i < 1000000; ++i)
   {
     pthread_mutex_lock(&mutex);
     ++(*count);
     pthread_mutex_unlock(&mutex);
   }
   return NULL;
}
```

```
int main (int argc, const char* argv[])
{
    int count = 0;
    pthread_t tid1,tid2;
    if (pthread_create(&tid1, NULL,
inc_count_thread, &count))
    return 1;
   if (pthread_create(&tid2, NULL,
inc_count_thread, &count))
    return 1;


    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("Count=%d\n",count);
    return 0;
}
```

## Threads – Thread Synchronization - Example



```
roee@roee-virtual-machine:~$ ./x
Count=2000000
roee@roee-virtual-machine:~$ ./x
Count=2000000
roee@roee-virtual-machine:~$ ./x
Count=2000000
roee@roee-virtual-machine:~$ ./x
Count=2000000
roee@roee-virtual-machine:~$ ./x
Count=2000000
roee@roee-virtual-machine:~$ ./x
Count=2000000
roee@roee-virtual-machine:~$ ./x
Count=2000000
roee@roee-virtual-machine:~$ ./x
Count=2000000
roee@roee-virtual-machine:~$
```

## Threads – Thread Synchronization - Mutexes

- A thread can lock a mutex by calling the *pthread_mutex_lock* library function
  - int pthread_mutex_lock(pthread_mutex_t *mutex);
  - In case the mutex is locked at the time *pthread_mutex_lock* is called, the thread blocks until it is unlocked
- A thread can unlock a mutex by calling the *pthread_mutex_unlock* library function
  - int pthread_mutex_unlock(pthread_mutex_t *mutex);
- A thread can lock a mutex asynchronously (i.e. without blocking) by calling the *pthread_mutex_trylock* library function
  - int pthread_mutex_trylock(pthread_mutex_t *mutex);
  - If the mutex is unlocked at the time of calling, the mutex becomes locked and the return value is 0
  - If the mutex is locked at the time of calling, the function returns instantly with a failure status of *EBUSY*

Copyright © 2016 **Roee Leon.** All Rights Reserved.

## Threads – Thread Synchronization - Deadlock

- A deadlock is a situation on which a thread enters a waiting state, waiting for some resource to become available, for indefinite amount of time
  - Example for deadlocks:
    - A thread tries to lock the same mutex twice
    - Using two mutexes: one thread holds the first mutex, waiting for the second to become available while the other thread holds the second mutex, waiting for the first to become available. Neither thread can proceed.
      - Solution?

## Threads – Thread Synchronization - Deadlock

- A deadlock is a situation on which a thread enters a waiting state, waiting for some resource to become available, for indefinite amount of time
  - Example for deadlocks:
    - A thread tries to lock the same mutex twice
    - Using two mutexes: one thread holds the first mutex, waiting for the second to become available while the other thread holds the second mutex, waiting for the first to become available. Neither thread can proceed.
      - Solution?
        - Control the order in which the mutexes are locked
        - Always lock at the same time!

## Threads – Thread Synchronization – Synchronized DS Example

- We want to implement a very basic hash table that hold reference-counted pointers to **struct Object**
- The hash table will be used in a multithreaded environment
  - i.e. the provided operations should be synchronized
- The data structure should provide the following functionality:
  - Insert
  - Delete
  - Search
- What do we need to add?

```
#define NHASH 10000
#define HASH(fp) (((unsigned long)fp)%NHASH)


struct Object
{
    int f_count;

    struct Object *f_next;
    int f_id;
    void* data;
};


struct Object *obj_map[NHASH];
```

## Threads – Thread Synchronization – Synchronized DS Example

- We want to implement a very basic hash table that hold reference-counted pointers to **struct Object**
- The hash table will be used in a multithreaded environment
  - i.e. the provided operations should be synchronized
- The data structure should provide the following functionality:
  - Insert
  - Delete
  - Search
- What do we need to add?

```
#define NHASH 5
#define HASH(fp) (((unsigned long)fp)%NHASH)


struct Object
{
    int f_count;
    pthread_mutex_t f_lock;
    struct Object *f_next;
    int f_id;
    void* data;
};


struct Object *obj_map[NHASH];
pthread_mutex_t hashlock =
PTHREAD_MUTEX_INITIALIZER;
```

## Threads – Thread Synchronization – Synchronized DS Example

- Given is a basic object allocation routine
- What does it miss?

```
struct Object* object_alloc(void* data)
{
    struct Object *obj;
    int idx;
    if ((obj = malloc(sizeof(struct Object))) != NULL)
    {
        obj->f_count = 1;
        idx = HASH(obj);
        obj->f_next = obj_map[idx];
        obj_map[idx] = obj;
        obj->data = data;
    }
    return obj;
}
```

Copyright © 2016 Roee Leon. All Rights Reserved.

# Threads – Thread Synchronization – Synchronized DS Example

- Given is a basic object insertion routine
- What does it miss?
  - New object mutex initialization
  - Locking the hash table before inserting the new object
  - Locking the file object before manipulating it

```
struct Object* object_alloc(void* data, int id)
{
    struct Object *obj;
    int idx;
    if ((obj = malloc(sizeof(struct Object))) != NULL)
    {
        obj->f_count = 1;
        obj->f_id = id;
        obj->data = data;
        idx = HASH(obj);
        obj->f_next = obj_map[idx];
        obj_map[idx] = obj;
        // … manipulate obj
    }
    return obj;
}
```

## Threads – Thread Synchronization – Synchronized DS Example

- Given is a basic object insertion routine
- What does it miss?
  - New object mutex initialization
  - Locking the hash table before inserting the new object
  - Locking the file object before manipulating it

```
struct Object* object_alloc(void* data, int id)
{
    struct Object *obj = NULL;
    int idx;
    if ((obj = malloc(sizeof(struct Object))) != NULL)
    {
        obj->f_count = 1;
        obj->f_id = id;
        obj->data = data;
        if (pthread_mutex_init(&obj->f_lock, NULL) != 0)
        {
            free(obj);
            return(NULL);
        }
        idx = HASH(obj);
        pthread_mutex_lock(&hashlock);
        obj->f_next = obj_map[idx];
        obj_map[idx] = obj;
        pthread_mutex_lock(&obj->f_lock);
        pthread_mutex_unlock(&hashlock);
        // … manipulate obj
        pthread_mutex_unlock(&obj->f_lock);
    }
    return obj;
}
```

# Threads – Thread Synchronization – Synchronized DS Example

- Given is a basic object deletion routine
- What does it miss?

```
void object_release(struct Object *obj)
{
    struct Object *tfobj;
    int idx;
    if (obj->f_count == 1)
    {
        idx = HASH(obj);
        tfobj = obj_map[idx];
        if (tfobj == obj)
            obj_map[idx] = obj->f_next;
        else
       {
            while (tfobj->f_next != obj)
                tfobj = tfobj->f_next;
            tfobj->f_next = obj->f_next;
        }
        free(obj);
    }
    else
    {
        obj->f_count--;
    }
}
```

## Threads – Thread Synchronization – Synchronized DS Example

- Given is a basic object deletion routine
- What does it miss?
  - Locking the object before asking for its ref. count
  - If it is the last reference, unlock the object before locking the hashlock
    - Remember that we want to keep the locking order the same!
  - Lock the hashlock
  - Lock the object
  - Recheck whether the condition still holds (last reference count)
    - If so, delete from the hash table
    - Otherwise, decrement the ref. count

```
void object_release(struct Object *obj)
{
    struct Object *tfobj;
    int idx;
    if (obj->f_count == 1)
    {
        idx = HASH(obj);
        tfobj = obj_map[idx];
        if (tfobj == obj)
            obj_map[idx] = obj->f_next;
        else
        {
            while (tfobj->f_next != obj)
                tfobj = tfobj->f_next;
            tfobj->f_next = obj->f_next;
        }
        free(obj);
    }
    else
    {
        obj->f_count--;
    }
}
```

# Threads – Thread Synchronization – Synchronized DS Example

```
void object_release(struct Object *obj)
{
    struct Object *tfobj;
    int idx;
    pthread_mutex_lock(&obj->f_lock);
    if (obj->f_count == 1)
    {
        pthread_mutex_unlock(&obj->f_lock);
        pthread_mutex_lock(&hashlock);
        pthread_mutex_lock(&obj->f_lock);
        if (obj->f_count != 1)
        {
            --obj->f_count;
            pthread_mutex_unlock(&obj->f_lock);
            pthread_mutex_unlock(&hashlock);
            return;
        }
        idx = HASH(obj);
        tfobj = obj_map[idx];
        if (tfobj == obj)
            obj_map[idx] = obj->f_next;
        else
        {
            while (tfobj->f_next != obj)
                tfobj = tfobj->f_next;
            tfobj->f_next = obj->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_unlock(&obj->f_lock);
        pthread_mutex_destroy(&obj->f_lock);
        free(obj);
    }
    else
    {
        obj->f_count--;
        pthread_mutex_unlock(&obj->f_lock);
    }
```

## Threads – Thread Synchronization – Condition Variable

- Another synchronization mechanism
- Provides a place for threads to rendezvous
- Allows threads to wait in a race-free way for an arbitrary condition to hold
    - The condition is protected by a mutex
    - The mutex must be locked before waiting on the condition variable
        - This helps avoiding race conditions on which a thread prepares to wait on a condition and another thread signals the condition just before the first thread actually started waiting

## Threads – Thread Synchronization – Condition Variable

- A condition variable is represented by the data type *pthread_cond_t*
- Before a condition variable can be used, it must be initialized by calling the *pthread_cond_init* function or by allocating statically with the prefix *PTHREAD_COND_INITIALIZER*
- In case a condition variable is allocated dynamically (e.g. using *malloc*), it must be destroyed by calling *pthread_cond_destroy* before it is freed

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

## Threads – Thread Synchronization – Condition Variable

- The functions *pthread_cond_wait* and *pthread_cond_timedwait* are used to wait for a condition to become true

- 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);

- The mutex argument, provided to *pthread_cond_wait* and *pthread_cond_timedwait*, protects the condition
  - The caller must pass it in a locked state to the function
    - The thread is atomically placed on the 'Waiting for condition to hold' list and the mutex is then automatically unlocked – to make sure the thread won't miss a change in the condition
    - When *pthread_cond_wait* returns (i.e. the condition holds) – the mutex is again automatically locked
  - The condition on which the thread waits must be re-evaluated upon awakening because of the possibility of *Spurious Wakeups*

## Threads – Thread Synchronization – Condition Variable

- The functions *pthread_cond_signal* and *pthread_cond_broadcast* are used to notify that a certain condition has been satisfied.
  - The *pthread_cond_signal* function will wake up at least one thread waiting on a condition while *pthread_cond_broadcast* will wake up all threads waiting on a condition – letting them contend for the mutex lock
- We have to make sure that we the condition variable is signaled only after the condition state is changed

Copyright © 2016 **Roee Leon.** All Rights Reserved.

## Threads – Thread Synchronization – Condition Variable - Example

```
struct MsgItem
{
   void* data;
   struct MsgItem* next;
};
struct MsgQueue
{
   struct MsgItem* head;
   struct MsgItem* tail;
};

struct MsgQueue* queue;

pthread_cond_t qcond =
PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock =
PTHREAD_MUTEX_INITIALIZER;
```

```
void pop_message(void)
{
   struct MsgItem* item;
   for (;;)
   {
      pthread_mutex_lock(&qlock);
      while (queue->head == NULL)
         pthread_cond_wait(&qcond,
&qlock);
      item = queue->head;
      queue->head = queue->head-
>next;
      pthread_mutex_unlock(&qlock);
   }
}
```

```
void push_message(struct MsgItem*
item)
{
   pthread_mutex_lock(&qlock);
   queue->tail->next = item;
   queue->tail = item;
   pthread_cond_signal(&qcond);
   pthread_mutex_unlock(&qlock);
  }
```