Operating Systems
COMS W4118
Reading Notes 5

Alexander Roth

2015 – 02 – 22

Advanced Programming in the Unix Environment Chapter 11: Threads

11.1 Introduction

- All threads within a single process have access to the same process components, such as file descriptors and memory.
- Anytime you try to share a single resource among multiple users, you have to deal with consistency.

11.2 Thread Concepts

- A typical UNIX process can be thought of as having a single thread of control: each process is doing only one thing at a time.
- Multi-threading allows us to design programs that can do more than one thing at a time within a single process, with each thread handling a separate task.
- We can simplify code that deals with asynchronous events by assigning a separate thread to handle each event type.
- Threads automatically have access to the same memory address space and file descriptors.
- The processing of independent tasks can be interleaved by assigning a separate thread per task. Two tasks can be interleaved only if they don't depend on teh processing performed by each other.
- Interactive programs can realize improved response time by using multiple threads to separate the portions of the program that deal with user input and output from the other parts of the program.

- A program can be simplified using threads regardless of the number of processors, because the number of processors doesn't affect the program structure.
- A thread consists of the information necessary to represent an execution context within a process
- This includes:
    1. A *thread ID* that identifies the thread within a process
    2. A signal mask
    3. A set of register values
    4. A stack
    5. A scheduling priority
    6. A policy,
    7. A signal mask
    8. An `errno` variable
    9. And thread-specific data

11.3 Thread Identification

- Every thread has a thread ID, which is only significant within the context of the process to which it belongs.
- A thread ID is represented by the `pthread_t` data type.
- A function must be used to compare two thread IDs.

```
#include <pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
```
Returns: nonzero if equal, 0 otherwise

- A thread can obtain its own thread ID by calling the `pthread_self` function.

```
#include <pthread.h>
pthread_t pthread_self(void);
```
Returns: the thread ID of the calling thread

- `pthread_self` and `pthread_equal` can be used when a thread needs to identify data structures that are tagged with its thread ID.

11.4 Thread Creation

- As the program runs, its behavior should be indistinguishable from the traditional process, until it creates more threads of control.
- Additional threads can be created by calling the `pthread_create` function.

```
#include <pthread.h>
int pthread_create(pthread_t *restrict tidp, const pthread_attr_t
*attr, void *(*start_rtn)(void *), void *restrict arg);
```

Returns: 0 if OK, error number on failure.

- The memory location pointed to by *tidp* is set to the thread ID of the newly created thread when `pthread_create` returns successfully.
- The *attr* argument is used to customize varoius thread attributes.
- The newly created thread starts running at the address of the *start_rtn* function.
- *arg* is a single argument passed to *start_rtn*.
- When a thread is created, there is no guarantee which will run first: the newly created thread or the calling thread.
- The set of pending signals for the thread is cleared for the newly created thread.

11.5 Thread Termination

- A single thread can exit in three ways, thereby stopping its flow of control, without terminating the entire process.
    1. The thread can simply return from the start routine. The return value is the thread's exit code.
    2. The thread can be cancelled by another thread in the same process.
    3. The thread can call `pthread_exit`.

```
#include <pthread.h>
void pthread_exit(void *rval_ptr);
```

- The *rval_ptr* argument is a typeless pointer, similar to the single argument passed to the start routine.
- This pointer is available to other threads in the process by calling the `pthread_join` function.

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **rval_ptr);
```

Returns: 0 if OK, error number on failure.

- The calling thread will block until the specified thread calls `pthread_exit`, returns from its start routinge, or is canceled.
- If the thread simply returnf from its start 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 setting *rval_ptr* to `NULL`, the method does not retrieve the terminated thread's termination status.

11.6 Thread Synchronization

- When multiple threads of control share the same memory, we need to make sure that each thread sees a consistent view of its data.
- WHen one thread can modify a variable that other threads can read or modify, we need to synchronize the threads to ensure that they don't use an invalid value when accessing the variable's memory contents.
- When one thread modifies a variable, other threads can potentially see inconsistencies when reading the value of that variable.
- Thread have to use a lock that will allow only one thread to access the variable at a time.
- If the modification is atomic, then there isn't a race condition. Similarly, if our data always appears to be *sequentially consistent*, then we need no additional synchronization.
- Our operations are sequentially consistent when multiple threads can't observe inconsistencies in our data.
- In a sequentially consistent environment, we can explain modifications to our data as a sequential step of operations taken by the running thread.

11.6.1 Mutexes

- A *mutex* is basically a lock that we set (lock) before accessing a shared resource and release (unlock) when we're done.
- While it is set, any other thread that tires to set it will block until we release it.
- This mutual-exclusion mechanism works only if we design our threads to follow the same data-access rules.
- A mutex variable is represented by the `pthread_mutex_t` data type.

- Before we can use a mutex variable, we must first initialize it by either setting it to the constant `PTHREAD_MUTEX_INITIALIZER` or by calling `pthread_mutex_init`.
- If we allocate the mutex dynamically, then we need to call `pthread_mutex_destroy` before free the memory.

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t
*restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```
Both return: 0 if OK, error number on failure

- To initialize a mutex with the default attributes, we set *attr* to `NULL`.
- To lock a mutex, we call `pthread_mutex_lock`. If the mutex is already locked, the calling thread will block until the mutex is unlocked.
- To unlock a mutex, we call `pthread_mutex_unlock`.

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```
All return: 0 if OK, error number on failure

- If a thread can't afford to block, it can use `pthread_mutex_trylock` to lock the mutex conditionally.
- If a mutex is unlocked at the time trylock is called, then trylock will lock the mutex without blocking and return 0.

11.6.2 Deadlock Avoidance

- A thread will deadlock itself if it tries to lock the same mutex twice.
- Deadlocks can be avoided by carefully controlling the order in which mutexes are locked.
- You'll have the potential for a deadlock only when one thread attempts to lock the mutexes in the opposite order from another thread.

11.6.3 `pthread_mutex_timedlock` Function

- `pthread_mutex_timedlock` is equivalent to `pthread_mutex_lock`, but if the timeout value is reached, `pthread_mutex_timedlock` will return the error code `ETIMEDOUT` without locking the mutex.

```
#include <pthread.h>
#include <time.h>
int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex, const
struct timespec *restrict tsptr);
```
Returns: 0 if OK, error number on failure

- The timeout specifies how long we are willing to wait in terms of absolute time.
- The timeout is represented by the `timespec` structure, which describes time in terms of seconds and nanoseconds.

11.6.4 Reader-Writer Locks

- Three states are possible with a reader-writer lock:
  1. Locked in read mode
  2. Locked in write mode
  3. Unlocked
- 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.
- When a reader-writer lock is write locked, all threads attempting to lock it block until its unlocked.
- When a reader-writer lock is read locked, all threads attempting to lock it in read mode are given access, but any threads attempting to lock it in write mode block until all the threads have released their read locks.
- Reader-writer locks are well suited for situations in which data structures are read more often than they are modified.
- Reader-writer locks are also called shared-exclusive locks.
- Must be initialized before use and destroyed before freeing the underlying memory.

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const
pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```
Both return: 0 if OK, error on failure

- Pass null pointer for *attr* if the reader-writer lock is supposed to have default attributes
- To lock a reader-writer lock in read mode, we call `pthread_rwlock_rdlock`.

- To write lock a reader-writer lock, we call `pthread_rwlock_wrlock`.
- We unlock the lock by calling `pthread_rwlock_unlock`.

```
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```
All return: 0 if OK, error number on failure

- We should always check for errors when we call functions that can potentially fail.
- However, we do not need to check them if we design our locking properly.
- We also have the conidtional versions of the reader-writer locking primitives.

```
#include <pthread.h>
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```
Both return: 0 if OK, error number on failure.

- When the lock can be acquired, these functions return 0.
- Otherwise, they return the error `EBUSY`.

11.6.5 Reader-Writer Locking with Timeouts

- Provides functions to lock reader-writer loks witha timeout to give applications a way to avoid blocking indefinitely.

```
#include <pthread.h>
#include <time.h>
int pthread_rwlock_timedlock(pthread_rwlock_t *restrict rwlock, const
struct timespec *restrict tsptr);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
const struct timespec *restrict tsptr);
```
Both return: 0 if OK, error number on failure

- The *tsptr* argument points to a `timespec` structure specifying the time at which the thread should stop blocking.

11.6.6 Condition Variables

- These synchronization objects provide a place for threads to rendezvous.
- Protected by a mutex.
- Can be initialized statically and dynamically
- Can use `pthread_cond_destroy` function to deinitialize a condition variable before freeing its underlying memory.

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t
*restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```
Both return: 0 if OK, error number on failure

- Again, setting *attr* argument to `NULL` gives default attributes.
- We use `pthread_cond_wait` to wait for a condition to be true.

```
#include <pthread.h>
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 tsptr);
```
Both return: 0 if OK, error number on failure

- The mutex passed to `pthread_cond_wait` protects the condition.
- There are two functions to notify threads that a condition has been satisfied.
- The `pthread_cond_signal` function will wake up at least one thread waiting on a condition.
- `pthread_cond_broadcast` function will wake up all threads waiting on a condition.

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```
Both return: 0 if OK, error number on failure

- When we call these functions, we are said to be *signaling* the thread or condition.
- We should signal the threads only after changing the state of the condition.