Advanced Programming in the UNIX Environment — *Threads*

Hop Lee
`hoplee@bupt.edu.cn`

Contents

1 Introduction

- From the former study, we saw that a limited amount of sharing can occur between related processes
- The technology named **thread of control/thread** can solve this problem.
- All threads within a single process have access to the same process components, such as file descriptors and memory.
- In this chapter, we'll introduce the synchronization mechanisms which can prevent multiple threads from viewing inconsistencies in their shared resources.

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.

- With multiple threads of control, we can design our programs to do more than one thing at a time within a single process.
- This approach can have several benefits.
  - 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.
  - Some problems can be partitioned so that overall program throughput can be improved.
  - Interactive programs can realize improved response time by using multiple threads.
- A thread consists of the information necessary to represent an execution context within a process.
- It includes a **thread ID** that identifies the thread within a process, a set of register values, a stack, a scheduling priority and policy, a signal mask, an `errno` variable, and thread-specific data.
- Everything within a process is sharable among the threads in a process.
- In this chapter, we talk about **pthread**(POSIX.1-2001 thread interface).

3 Thread Identification

- Every thread has a thread ID, which is significant only within the context of the process to which it belongs.
- A thread ID is represented by the `pthread_t` data type. Which can be a structure or an integer number.
- So portable implementations use a function to compare two thread IDs.

```
#include <pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

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

```
#include <pthread.h>
pthread_t pthread_self(void);
```

4 Thread Creation

- Additional threads can be created by calling the `pthread_create` function within a process.

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

- 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 various thread attributes.
- The newly created thread starts running at the address of the *start_rtn* function. This function takes a single argument, *arg*, which is a typeless pointer.

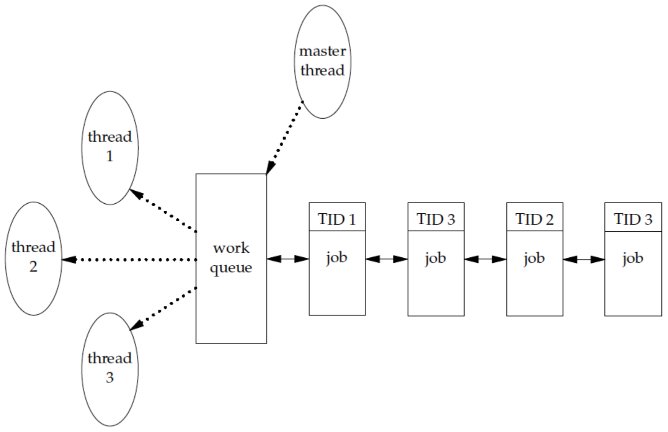

Figure 1: Work queue example

- If you need to pass more than one argument to the `start_rtn` function, then you need to store them in a structure and pass the address of the structure in *arg*.
- When a thread is created, there is no guarantee which runs first: the newly created thread or the calling thread.
- Example (Figure 11.2, `threads/threadid.c`) prints the thread IDs.
- Notes: *We can't make any assumptions about how threads will be scheduled.*

5 Thread Termination

- If any thread within a process calls `exit`, `_Exit`, or `_exit`, then the entire process terminates.
- When the default action is to terminate the process, a signal sent to a thread will terminate the entire process.
- 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 canceled 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* 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);
```

- The calling thread will block until the specified thread calls `pthread_exit`, returns from its start routine, or is canceled.
- If the thread simply returned 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 calling `pthread_join`, we automatically place a thread in the detached state so that its resources can be recovered.
- If the thread was already in the detached state, calling `pthread_join` fails, returning `EINVAL`.
- If we're not interested in a thread's return value, we can set *rval_ptr* to `NULL`. In this case, calling `pthread_join` allows us to wait for the specified thread, but does not retrieve the thread's termination status.
- Example (Figure 11.3, `threads/exitstatus.c`) shows how to fetch the exit code from a thread that has terminated.
- Example (Figure 11.4, `threads/badexit2.c`) shows the problem with using an automatic variable as the argument to `pthread_exit`.
- One thread can request that another in the same process be canceled by calling the `pthread_cancel` function.

```
#include <pthread.h>
int pthread_cancel(pthread_t tid);
```

- In the default circumstances, `pthread_cancel` will cause the thread specified by *tid* to behave as if it had called `pthread_exit` with an argument of `PTHREAD_CANCELED`.
- However, a thread can elect to ignore or otherwise control how it is canceled.
- Note that `pthread_cancel` doesn't wait for the thread to terminate. It merely makes the request.
- A thread can arrange for functions to be called when it exits, similar to the way that the `atexit` function can be used by a process to arrange that functions can be called when the process exits.
- The functions are known as **thread cleanup handlers**.
- More than one cleanup handler can be established for a thread.
- The handlers are recorded in a stack, which means that they are executed in the reverse order from that with which they were registered.

```
#include <pthread.h>
void pthread_cleanup_push(void (*rtn)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

- The `pthread_cleanup_push` function schedules the cleanup function, *rtn*, to be called with the single argument, *arg*, when the thread performs one of the following actions:
  - Makes a call to `pthread_exit`;
  - Responds to a cancelation request;
  - Makes a call to `pthread_cleanup_pop` with a nonzero execute argument
- If the *execute* argument is set to zero, the cleanup function is not called. In either case, `pthread_cleanup_pop` removes the cleanup handler established by the last call to `pthread_cleanup_push`.
- A restriction with these functions is that, because they can be implemented as macros, they must be used in matched pairs within the same scope in a thread.
- If the thread terminates by returning from its start routine, its cleanup handlers are not called.

- Example (Figure 11.5, threads/cleanup.c) shows how to use thread cleanup handlers.
- We can detach a thread by calling pthread_detach.

```c
#include <pthread.h>
int pthread_detach(pthread_t tid);
```

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 modifies a variable, other threads can potentially see inconsistencies when reading the value of the variable.
- Left figure shows a hypothetical example of two threads reading and writing the same variable.
- To solve the inconsistent problem, the threads have to use a lock that will allow only one thread to access the variable at a time. Right figure shows this synchronization.

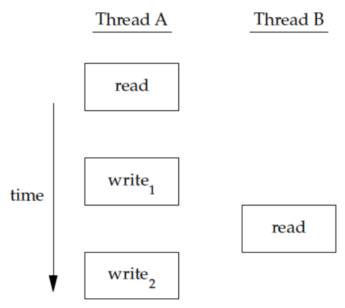

Figure 2: Inconsistent of data

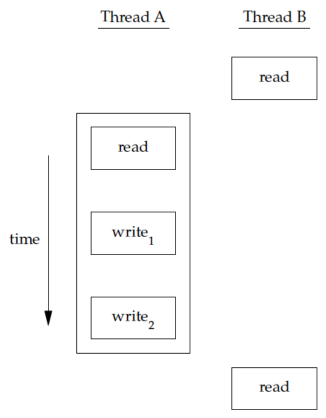

Figure 3: Synchronization of data

- You also need to synchronize two or more threads that might try to modify the same variable at the same time. The increment operation is usually broken down into three steps.

1. Read the memory location into a register.
2. Increment the value in the register.
3. Write the new value back to the memory location

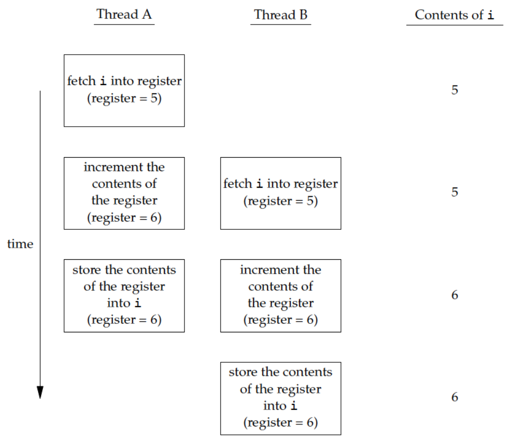
Thread A
Thread B
Contents of i
fetch i into register (register = 5)
5
increment the contents of the register (register = 6)
fetch i into register (register = 5)
5
time
store the contents of the register into i (register = 6)
increment the contents of the register (register = 6)
6
store the contents of the register into i (register = 6)
6

Figure 4: Two unsynchronized threads incrementing the same variable

- If two threads try to increment the same variable at almost the same time without synchronizing with each other, the results can be inconsistent.
- If the modification is atomic, then there isn't a race.
- If our data always appears to be *sequentially consistent*, then we need no additional synchronization.
- In a sequentially consistent environment, we can explain modifications to our data as a sequential step of operations taken by the running threads.

6.1 Mutexes

- We can protect our data and ensure access by only one thread at a time by using the pthreads mutual-exclusion interfaces.
- 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 tries to set it will block until we release it.
- If more than one thread is blocked when we unlock the mutex, then all threads blocked on the lock will be made runnable, and the first one to run will be able to set the lock. The others will see that the mutex is still locked and go back to waiting for it to become available again.
- In this way, only one thread will proceed at a time.
- This mutual-exclusion mechanism works only if we design our threads to follow the same data-access rules. So, mutex mechanism is an advisory type lock basically.
- 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` (for statically-allocated mutexes only) or calling `pthread_mutex_init`.

- If we allocate the mutex dynamically (by calling malloc, for example), then we need to call pthread_mutex_destroy before freeing 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);
```

- 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);
```

- Example (Figure 11.10, threads/mutex1.c) illustrates a mutex used to protect a data structure.

6.2 Deadlock Avoidance

- A thread will deadlock itself if it tries to lock the same mutex twice.
- When we use more than one mutex in our programs, a deadlock can occur if we allow one thread to hold a mutex and block while trying to lock a second mutex at the same time that another thread holding the second mutex tries to lock the first mutex.
- Deadlocks can be avoided by carefully controlling the order in which mutexes are locked.
- Sometimes, an application's architecture makes it difficult to apply a lock ordering.
- If your locking granularity is too coarse, you end up with too many threads blocking behind the same locks, with little improvement possible from concurrency. If your locking granularity is too fine, then you suffer bad performance from excess locking overhead, and you end up with complex code.
- Example (Figure 11.11, threads/mutex2.c), and example (Figure 11.12, threads/mutex3.c).

6.3 pthread_mutex_timedlock Function

- One additional mutex primitive allows us to bound the time that a thread blocks when a mutex it is trying to acquire is already locked:

```
#include <pthread.h>
#include <time.h>
int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
                            const struct timespec *restrict tsptr);
```

- Example in Figure 11.13 shows how to use pthread_mutex_timedlock to avoid blocking indefinitely.

6.4 Reader-Writer Locks

- Readerwriter locks are similar to mutexes, except that they allow for higher degrees of parallelism.
- Three states are possible with a readerwriter lock:
  1. locked in read mode,
  2. locked in write mode,
  3. and unlocked.
- Only one thread at a time can hold a readerwriter lock in write mode, but multiple threads can hold a readerwriter lock in read mode at the same time.
- Although implementations vary, readerwriter locks usually block additional readers if a lock is already held in read mode and a thread is blocked trying to acquire the lock in write mode. This prevents a constant stream of readers from starving waiting writers.
- Readerwriter locks are well suited for situations in which data structures are read more often than they are modified.
- Readerwriter locks are also called sharedexclusive locks.
- As with mutexes, readerwriter locks must be initialized before use and destroyed before freeing their 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);
```

- To lock a readerwriter lock in read mode or write mode and unlock a readerwriter lock through these three functions:

```
#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);
```

- The Single UNIX Specification also defines conditional versions of the readerwriter locking primitives.

```
#include <pthread.h>
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

- When the lock can be acquired, these functions return 0. Otherwise, they return the error `EBUSY`.
- Example (Figure 11.14, `thresds/rwlock.c`).

6.5 Reader-Writer Locking with Timeout

- Just as with mutexes, the SUS provides functions to lock reader-writer locks with a timeout to give applications a way to avoid blocking indefinitely while trying to acquire a reader-writer lock. These functions are:

```c
#include <pthread.h>
#include <time.h>
int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
                               const struct timespec *restrict tsptr);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
                               const struct timespec *restrict tsptr);
```

- These functions behave like their "untimed" counterparts. The *tsptr* argument points to a `timespec` structure specifying the time at which the thread should stop blocking. If they can't acquire the lock, these functions return the `ETIMEDOUT` error when the timeout expires. Like the `pthread_mutex_timedlock` function, the timeout specifies an absolute time, not a relative one.

6.6 Condition Variables

- Condition variables are another synchronization mechanism available to threads.
- Condition variables provide a place for threads to rendezvous. 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 mutex to change the condition state.
- Before a condition variable is used, it must first be initialized with `pthread_cond_init`. We can use the `pthread_cond_destroy` function to de-initialize a condition variable before freeing its underlying memory.

```c
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *restrict cond,
                      pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

- We use these functions to wait for a condition to be true.

```c
#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 timeout);
```

- Using this structure, we need to specify how long we are willing to wait as an absolute time instead of a relative time.
- If the timeout expires without the condition occurring, `pthread_cond_timedwait` will reacquire the mutex and return the error `ETIMEDOUT`.
- When it returns from a successful call to `pthread_cond_wait` or `pthread_cond_timedwait`, a thread needs to reevaluate the condition, since another thread might have run and already changed the condition.
- There are two functions to notify threads that a condition has been satisfied.

```c
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- When we call these functions, we are said to be signaling the thread or condition.
- Example (Figure 11.15, `threads/condvar.c`).

6.7 Spin Locks

- A **spin lock** is like a mutex, except that instead of blocking a process by sleeping, the process is blocked by *busy-waiting* (spinning) until the lock can be acquired.
- A spin lock could/should be used in situations where locks are held for short periods of times and threads don't want to incur the cost of being descheduled.
- Spin locks are often used as low-level primitives to implement other types of locks.
- The interfaces for spin locks are similar to those for mutexes, making it relatively easy to replace one with the other.

```
#include <pthread.h>
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

- We can use these functions to lock/unlock the spin lock:

```
#include <pthread.h>
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

- If we try to lock a spin lock is locked or unlock a spin lock that is not locked, the results are undefined. The behavior depends on the implementation.

6.8 Barriers

- **Barriers** are a synchronization mechanism that can be used to coordinate multiple threads working in parallel. A barrier allows each thread to wait until all cooperating threads have reached the same point, and then continue executing from there.
- We can use the functions shown bellow to initialize/deinitialize a barrier:

```
#include <pthread.h>
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
                         const pthread_barrierattr_t *restrict attr,
                         unsigned int count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

- We use the `pthread_barrier_wait` function to indicate that a thread is done with its work and is ready to wait for all the other threads to catch up:

```
#include <pthread.h>
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

- The thread calling `pthread_barrier_wait` is put to sleep if the barrier count is not yet satisfied. If the thread is the last one to call `pthread_barrier_wait`, thereby satisfying the barrier count, all of the threads are awakened.
- To one arbitrary thread, it will appear as if the `pthread_barrier_wait` function returned a value of `PTHREAD_BARRIER_SERIAL_`
The remaining threads see a return value of 0.
- Example (Figure 11.16, `threads/barrier.c`).

The End of Chapter 13.