Advanced Programming in the UNIX Environment — *Thread Control*

Hop Lee
hoplee@bupt.edu.cn

Contents

1 Introduction

- In Chapter 13, we learned the basics about threads and thread synchronization. In this chapter, we will learn the details of controlling thread behavior.
- We will look at thread attributes and synchronization primitive attributes, which we ignored in the previous chapter in favor of the default behaviors.
- We will follow this with a look at how threads can keep data private from other threads in the same process.
- Then we will wrap up the chapter with a look at how some process-based system calls interact with threads.

2 Thread Limits

- The SUS also defines several limits associated with the operation of threads.
- As with other system limits, the thread limits can be queried using `sysconf`. Figure 12.1 on the textbook summarizes these limits.
- As with the other limits reported by `sysconf`, use of these limits is intended to promote application portability among different operating system implementations.

3 Thread Attributes

- The pthread interface allows us to fine-tune the behavior of threads and synchronization objects by setting various attributes associated with each object.
- Generally, the functions for managing these attributes follow the same pattern:
  1. Each object is associated with its own type of attribute object. An attribute object can represent multiple attributes. The attribute object is opaque to applications.
  2. An initialization function exists to set the attributes to their default values.
  3. Another function exists to destroy the attributes object.
  4. Each attribute has a function to get the value of the attribute from the attribute object.
  5. Each attribute has a function to set the value of the attribute.
- We can use the `pthread_attr_t` structure to modify the default attributes, and associate these attributes with threads that we create.
- We use the `pthread_attr_init` function to initialize the `pthread_attr_t` structure.
- After calling `pthread_attr_init`, the `pthread_attr_t` structure contains the default values for all the thread attributes supported by the implementation.

```
1 #include <pthread.h>
2 int pthread_attr_init(pthread_attr_t *attr);
3 int pthread_attr_destroy(pthread_attr_t *attr);
```

- To deinitialize a `pthread_attr_t` structure, we call `pthread_attr_destroy`.
- If an implementation of `pthread_attr_init` allocated any dynamic memory for the attribute object, `pthread_attr_destroy` will free that memory.
- In addition, `pthread_attr_destroy` will initialize the attribute object with *invalid* values, so if it is used by mistake, `pthread_create` will return an error code.
- The thread attributes defined by POSIX.1 are summarized in Figure 12.3 of the textbook.
- If we are no longer interested in an existing thread's termination status, we can use `pthread_detach` to allow the operating system to reclaim the thread's resources when the thread exits.
- If we know that we don't need the thread's termination status at the time we create the thread, we can arrange for the thread to start out in the *detached state* by modifying the **detachstate** thread attribute in the `pthread_attr_t` structure.

- We can use the `pthread_attr_setdetachstate` function to set the detachstate thread attribute to one of two legal values: `PTHREAD_CREATE_DETACHED` to start the thread in the detached state or `PTHREAD_CREATE_JOINABLE` to start the thread normally, so its termination status can be retrieved by the application.
- We can call `pthread_attr_getdetachstate` to obtain the current *detachstate* attribute.

```
#include <pthread.h>
int pthread_attr_getdetachstate(const pthread_attr_t *restrict attr,
                                int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

- Example (Figure 12.4, `threadctl/detach.c`).
- Support for thread stack attributes is optional for a POSIX-conforming operating system, but is required if the system is to conform to the XSI.
- The preferred way to query and modify a thread's stack attributes is to use the newer functions `pthread_attr_getstack` and `pthread_attr_setstack`.

```
#include <pthread.h>
int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                          void **restrict stackaddr,
                          size_t *restrict stacksize);
int pthread_attr_setstack(const pthread_attr_t *attr,
                          void *stackaddr, size_t *stacksize);
```

- An application can also get and set the **stacksize** thread attribute using the `pthread_attr_getstacksize` and `pthread_attr_setstacksize` functions.

```
#include <pthread.h>
int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                              size_t *restrict stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr,
                              size_t stacksize);
```

- The **guardsize** thread attribute controls the size of the memory extent after the end of the thread's stack to protect against stack overflow.
- By default, this is set to `PAGESIZE` bytes.
- We can set the **guardsize** thread attribute to 0 to disable this feature: no guard buffer will be provided in this case.

```
#include <pthread.h>
int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                              size_t *restrict guardsize);
int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
```

4 Synchronization Attributes

- Just as threads have attributes, so do their synchronization objects.
- In this section, we discuss the attributes of mutexes, reader-writer locks, and condition variables.

4.1 Mutex Attributes

- We use `pthread_mutexattr_init` to initialize a `pthread_mutexattr_t` structure and `pthread_mutexattr_destroy` to deinitialize one.

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

- The `pthread_mutexattr_init` function will initialize the `pthread_mutexattr_t` structure with the default mutex attributes.
- Two attributes of interest are the **process-shared** attribute and the **type** attribute.
- Within a process, multiple threads can access the same synchronization object. In this case, the process-shared mutex attribute is set to `PTHREAD_PROCESS_PRIVATE`.
- The process-shared mutex attribute allows the pthread library to provide more efficient mutex implementations when the attribute is set to `PTHREAD_PROCESS_PRIVATE`, which is the default case with multithreaded applications.
- As we saw in Chapters 14 and 15, mechanisms exist that allow independent processes to map the same extent of memory into their independent address spaces. If the process-shared mutex attribute is set to `PTHREAD_PROCESS_SHARED`, a mutex allocated from a memory extent shared between multiple processes may be used for synchronization by those processes.
- We can use the `pthread_mutexattr_getpshared` function to query a `pthread_mutexattr_t` structure for its process-shared attribute. We can change the process-shared attribute with the `pthread_mutexattr_setpshared` function.

```
#include <pthread.h>
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict attr,
                                 int *restrict pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```

- The `robust` mutex attribute is meant to address the problem of mutex state recovery when a process terminates while holding a mutex.
- We can use the `pthread_mutexattr_getrobust` function to get the value of the `robust` mutex attribute. To set the value of the `robust` mutex attribute, we can call the `pthread_mutexattr_setrobust` function.

```
#include <pthread.h>
int pthread_mutexattr_getrobust(const pthread_mutexattr_t *
                                restrict attr,
                                int *restrict robust);
int pthread_mutexattr_setrobust(pthread_mutexattr_t *attr,
                                int robust);
```

- If the application state can't be recovered, the mutex will be in a permanently unusable state after the thread unlocks the mutex. To prevent this problem, the thread can call the `pthread_mutex_consistent` function to indicate that the state associated with the mutex is consistent before unlocking the mutex.

```
#include <pthread.h>
int pthread_mutex_consistent(pthread_mutex_t * mutex);
```

- The `type` mutex attribute controls the characteristics of the mutex. POSIX.1 defines four types.

  `PTHREAD_MUTEX_NORMAL` A standard mutex that doesn't do any special error checking or deadlock detection.

  `PTHREAD_MUTEX_ERRORCHECK` A mutex type that provides error checking.

  `PTHREAD_MUTEX_RECURSIVE` A mutex type that allows the same thread to lock it multiple times without first unlocking it. A recursive mutex maintains a lock count and isn't released until it is unlocked the same number of times it is locked.

  `PTHREAD_MUTEX_DEFAULT` A mutex type providing default characteristics and behavior. Implementations are free to map this to one of the other mutex types.

- The behavior of the four types is shown in Figure 12.5 in the textbook, p433.

- We can use `pthread_mutexattr_gettype` to get the mutex type attribute and `pthread_mutexattr_settype` to change it.

```
1 #include <pthread.h>
2 int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
3                               int *restrict type);
4 int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

- Recursive mutexes are useful when you need to adapt existing single-threaded interfaces to a multithreaded environment, but can't change the interfaces to your functions because of compatibility constraints.

- However, using recursive locks can be tricky, and they should be used only when no other solution is possible.

- Example (Figure 12.8, `threadctl/timeout.c`) illustrates another situation in which a recursive mutex is necessary.

4.2 Reader-Writer Lock Attributes

- Readerwriter locks also have attributes, similar to mutexes. We use `pthread_rwlockattr_init` to initialize a `pthread_rwlockattr_t` structure and `pthread_rwlockattr_destroy` to deinitialize the structure.

```
1 #include <pthread.h>
2 int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
3 int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

- The only attribute supported for readerwriter locks is the `process-shared` attribute.

- It is identical to the mutex `process-shared` attribute.

- Just as with the mutex `process-shared` attributes, a pair of functions is provided to get and set the `process-shared` attributes of reader-writer locks.

```
1 #include <pthread.h>
2 int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict attr,
3                                   int *restrict pshared);
4 int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

4.3 Condition Variable Attributes

- The SUS currently defines two attributes for condition variables: the `process-shared` attribute and the `clock` attribute.
- As with the other attribute objects, a pair of functions initialize and deinitialize condition variable attribute objects.

```
#include <pthread.h>
int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);
```

- The `process-shared` attribute is the same as with the other synchronization attributes. It controls whether condition variables can be used by threads within a single process only or from within multiple processes.

```
#include <pthread.h>
int pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,
                                int *restrict pshared);
int pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared);
```

- The `clock` attribute controls which clock is used when evaluating the timeout argument (*tsptr*) of the `pthread_cond_timedwait` function.

```
#include <pthread.h>
int pthread_condattr_getclock(const pthread_condattr_t *
                              restrict attr,
                              clockid_t *restrict clock_id);
int pthread_condattr_setclock(pthread_condattr_t *attr,
                              clockid_t clock_id);
```

4.4 Barrier Attributes

- Barriers have attributes, too.

```
#include <pthread.h>
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```

- The only barrier attribute currently defined is the `process-shared` attribute, which controls whether a barrier can be used by threads from multiple processes or only from within the process that initialized the barrier.

```
#include <pthread.h>
int pthread_barrierattr_getpshared(const pthread_barrierattr_t *
                                   restrict attr,
                                   int *restrict pshared);
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);
```

- The value of the `process-shared` attribute can be either `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

5 Reentrancy

- If a function can be safely called by multiple threads at the same time, we say that the function is **thread-safe**.
- All functions defined in the SUS are guaranteed to be thread-safe, except those listed in Figure 12.9 of the textbook, p442.
- With thread-safe functions, implementations provide alternative, thread-safe versions of some of the POSIX.1 functions that aren't thread-safe.
- Functions are made thread-safe by changing their interfaces to require that the caller provide its own buffer.
- If a function is reentrant with respect to multiple threads, we say that it is *thread-safe*. This doesn't tell us, however, whether the function is reentrant with respect to signal handlers.
- We say that a function that is safe to be reentered from an asynchronous signal handler is **async-signal safe**.
- POSIX.1 provides a way to manage `FILE` objects in a thread-safe way.
- You can use `flockfile` and `ftrylockfile` to obtain a lock associated with a given `FILE` object.
- This lock is recursive: you can acquire it again, while you already hold it, without deadlocking.

```
1 #include <stdio.h>
2 int ftrylockfile(FILE *fp);
3 void flockfile(FILE *fp);
4 void funlockfile(FILE *fp);
```

- If the standard I/O routines acquire their own locks, then we can run into serious performance degradation when doing character-at-a-time I/O. In this situation, we end up acquiring and releasing a lock for every character read or written. To avoid this overhead, unlocked versions of the character-based standard I/O routines are available.

```
1 #include <stdio.h>
2 int getchar_unlocked(void);
3 int getc_unlocked(FILE *fp);
4 int putchar_unlocked(int c);
5 int putc_unlocked(int c, FILE *fp);
```

- These four functions should not be called unless surrounded by calls to `flockfile` (or `ftrylockfile`) and `funlockfile`. Otherwise, unpredictable results can occur.
- Once you lock the `FILE` object, you can make multiple calls to these functions before releasing the lock. This amortizes the locking overhead across the amount of data read or written.
- Example (Figure 12.11, `threadctl/getenv.c`) shows a possible implementation of `getenv`. This version is not reentrant.
- Example (Figure 12.12, `threadctl/getenv2.c`) shows a reentrant version.

6 Thread-Specific Data

- **Thread-specific data**, also known as **thread-private data**, is a mechanism for storing and finding data associated with a particular thread.
- The reason we call the data thread-specific, or thread-private, is that we'd like each thread to access its own separate copy of the data, without worrying about synchronizing access with other threads.

- There are two reasons why would anyone want to promote interfaces that prevent sharing in thread model:
  1. First, sometimes we need to maintain data on a per thread basis.
  2. The second reason for thread-private data is to provide a mechanism for adapting process-based interfaces to a multithreaded environment.
- Even though the underlying implementation doesn't prevent access, the functions provided to manage thread-specific data promote data separation among threads.
- Before allocating thread-specific data, we need to create a *key* to associate with the data. The key will be used to gain access to the thread-specific data.

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *keyp,
                       void (*destructor)(void *));
```

- Threads usually use `malloc` to allocate memory for their thread-specific data. The destructor function usually frees the memory that was allocated.
- A thread can allocate multiple keys for thread-specific data. Each key can have a destructor associated with it.
- We can break the association of a key with the thread-specific data values for all threads by calling `pthread_key_delete`.

```
#include <pthread.h>
int pthread_key_delete(pthread_key_t *key);
```

- We need to ensure that a key we allocate doesn't change because of a race during initialization. Depending on how the system schedules threads, some threads might see one key value, whereas other threads might see a different value. The way to solve this race is to use `pthread_once`.

```
#include <pthread.h>
pthread_once_t initflag = PTHREAD_ONCE_INIT;
int pthread_once(pthread_once_t *initflag,
                 void (*initfn)(void));
```

- The *initflag* must be a nonlocal variable (i.e., global or static) and initialized to `PTHREAD_ONCE_INIT`.
- If each thread calls `pthread_once`, the system guarantees that the initialization routine, *initfn*, will be called only once, on the first call to `pthread_once`.
- Once a key is created, we can associate thread-specific data with the key by calling `pthread_setspecific`. We can obtain the address of the thread-specific data with `pthread_getspecific`.

```
#include <pthread.h>
void *pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key,
                        const void *value);
```

- If no thread-specific data has been associated with a key, `pthread_getspecific` will return a null pointer. We can use this to determine whether we need to call `pthread_setspecific`.
- Example (Figure 12.13, `threadctl/getenv3.c`).

7 Cancel Options

- Two thread attributes that are not included in the `pthread_attr_t` structure are the **cancelability state** and the **cancelability type**. These attributes affect the behavior of a thread in response to a call to `pthread_cancel`.
- The **cancelability state** attribute can be either `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`. A thread can change its **cancelability state** by calling `pthread_setcancelstate`.

```
1 #include <pthread.h>
2 int pthread_setcancelstate(int state,
3                            int *oldstate);
```

- In the default case, a thread will continue to execute after a cancellation request is made, until the thread reaches a **cancellation point**.
- A cancellation point is a place where the thread checks to see whether it has been canceled, and then acts on the request.
- A thread starts with a default cancelability state of `PTHREAD_CANCEL_ENABLE`. When the state is set to `PTHREAD_CANCEL_DISABLE`, a call to `pthread_cancel` will not kill the thread. Instead, the cancellation request remains pending for the thread. When the state is enabled again, the thread will act on any pending cancellation requests at the next cancellation point.
- If your application doesn't call one of the functions in Figure 12.14 or Figure 12.15 on the textbook for a long period of time, then you can call `pthread_testcancel` to add your own cancellation points to the program.

```
1 #include <pthread.h>
2 void pthread_testcancel(void);
```

- When you call `pthread_testcancel`, if a cancellation request is pending and if cancellation has not been disabled, the thread will be canceled. When cancellation is disabled, however, calls to `pthread_testcancel` have no effect.
- The default cancellation type we have been describing is known as deferred cancellation. After a call to `pthread_cancel`, the actual cancellation doesn't occur until the thread hits a cancellation point. We can change the cancellation type by calling `pthread_setcanceltype`.

```
1 #include <pthread.h>
2 int pthread_setcanceltype(int type, int *oldtype);
```

- The type parameter can be either `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`.
- Asynchronous cancellation differs from deferred cancellation in that the thread can be canceled at any time. The thread doesn't necessarily need to hit a cancellation point for it to be canceled.

8 Threads and Signals

- Dealing with signals can be complicated even with a process-based paradigm. Introducing threads into the picture makes things even more complicated.
- Each thread has its own signal mask, but the signal disposition is shared by all threads in the process.
- The behavior of `sigprocmask` is undefined in a multithreaded process. Threads have to use the `pthread_sigmask` function instead.

```c
#include <signal.h>
int pthread_sigmask(int how, const sigset_t *restrict set,
                    sigset_t *restrict oset);
```

- The `pthread_sigmask` function is identical to `sigprocmask`, except that `pthread_sigmask` works with threads and returns an error code on failure instead of setting `errno` and returning -1.
- A thread can wait for one or more signals to occur by calling `sigwait`.

```c
#include <signal.h>
int sigwait(const sigset_t *restrict set,
            int *restrict signop);
```

- If one of the signals specified in the set is pending at the time sigwait is called, then `sigwait` will return without blocking. Before returning, `sigwait` removes the signal from the set of signals pending for the process. If the implementation supports queued signals, and multiple instances of a signal are pending, sigwait will remove only one instance of the signal; the other instances will remain queued.
- To avoid erroneous behavior, a thread must block the signals it is waiting for before calling `sigwait`. The `sigwait` function will atomically unblock the signals and wait until one is delivered. Before returning, `sigwait` will restore the thread's signal mask. If the signals are not blocked at the time that `sigwait` is called, then a timing window is opened up where one of the signals can be delivered to the thread before it completes its call to `sigwait`.
- The advantage to using `sigwait` is that it can simplify signal handling by allowing us to treat asynchronously generated signals in a synchronous manner. We can prevent the signals from interrupting the threads by adding them to each thread's signal mask.
- If multiple threads are blocked in calls to `sigwait` for the same signal, only one of the threads will return from `sigwait` when the signal is delivered.
- If a signal is being caught and a thread is waiting for the same signal in a call to `sigwait`, it is left up to the implementation to decide which way to deliver the signal. The implementation could either allow `sigwait` to return or invoke the signal handler, but not both.
- To send a signal to a process, we call `kill`. To send a signal to a thread, we call `pthread_kill`.

```c
#include <signal.h>
int pthread_kill(pthread_t thread, int signo);
```

- Example (Figure 12.16, `threadctl/suspend.c`).

9 Threads and `fork`

- When a thread calls `fork`, a copy of the entire process address space is made for the child.
- By inheriting a copy of the address space, the child also inherits the state of every mutex, reader-writer lock, and condition variable from the parent process. If the parent consists of more than one thread, the child will need to clean up the lock state if it isn't going to call `exec` immediately after `fork` returns.
- Inside the child process, only one thread exists. It is made from a copy of the thread that called `fork` in the parent. If the threads in the parent process hold any locks, the same locks will also be held in the child process. The problem is that the child process doesn't contain copies of the threads holding the locks, so there is no way for the child to know which locks are held and need to be unlocked.

- This problem can be avoided if the child calls one of the `exec` functions directly after returning from `fork`. This is not always possible, however, so if the child needs to continue processing, we need to use a different strategy.
- To avoid problems with inconsistent state in a multithreaded process, POSIX.1 states that only async-signal safe functions should be called by a child process between the time that `fork` returns and the time that the child calls one of the `exec` functions.
- To clean up the lock state, we can establish **fork handlers** by calling the function `pthread_atfork`.

```
1 #include <pthread.h>
2 int pthread_atfork(void (*prepare)(void), void (*parent)(void),
3                    void (*child)(void));
```

- The parent and the child end up unlocking duplicate locks stored in different memory locations, as if the following sequence of events occurred:
  1. The parent acquired all its locks.
  2. The child acquired all its locks.
  3. The parent released its locks.
  4. The child released its locks.
- We can call `pthread_atfork` multiple times to install more than one set of fork handlers.
- If we don't have a need to use one of the handlers, we can pass a null pointer for the particular handler argument, and it will have no effect.
- Example (Figure 12.17,) illustrates the use of `pthread_atfork` and fork handlers.
- Although the `pthread_atfork` mechanism is intended to make locking state consistent after a `fork`, it has several drawbacks that make it usable in only limited circumstances:
  - There is no good way to reinitialize the state for more complex synchronization objects such as condition variables and barriers.
  - Some implementations of error-checking mutexes will generate errors when the child fork handler tries to unlock a mutex that was locked by the parent.
  - Recursive mutexes can't be cleaned up in the child fork handler, because there is no way to determine the number of times one has been locked.
  - If child processes are allowed to call only async-signal safe functions, then the child fork handler shouldn't even be able to clean up synchronization objects, because none of the functions that are used to manipulate them are async-signal safe. The practical problem is that a synchronization object might be in an intermediate state when one thread calls `fork`, but the synchronization object can't be cleaned up unless it is in a consistent state.
  - If an application calls `fork` in a signal handler (which is legal, because `fork` is async-signal safe), then the fork handlers registered by `pthread_atfork` can call only async-signal safe functions, or else the results are undefined.

10 Threads and I/O

- We introduced the `pread` and `pwrite` functions in Section 3.11. These functions are helpful in a multithreaded environment, because all threads in a process share the same file descriptors.
- Consider two threads reading from or writing to the same file descriptor at the same time.

```
1       Thread A                        Thread B
2 lseek(fd, 300, SEEK_SET);      lseek(fd, 700, SEEK_SET);
3 read(fd, buf1, 100);           read(fd, buf2, 100);
```

- If thread A executes the call to `lseek` and then thread B calls `lseek` before thread A calls `read`, then both threads will end up reading the same record. Clearly, this isn't what was intended.
- To solve this problem, we can use `pread` to make the setting of the offset and the reading of the data one atomic operation.

```
1       Thread A                        Thread B
2 pread(fd, buf1, 100, 300);    pread(fd, buf2, 100, 700);
```

- Using `pread`, we can ensure that thread A reads the record at offset 300, whereas thread B reads the record at offset 700. We can use `pwrite` to solve the problem of concurrent threads writing to the same file.

The End of Chapter 15.