ECE391 Lecture Notes: Abstractions and Interfaces in the Posix Thread and Semaphore APIs

Steven S. Lumetta

December 27, 2009

A number of elective courses open to you after ECE 391 will assume that you have some familiarity and experience with user-level synchronization and threading, primarily in the form of the Posix application programming interfaces (APIs). This set of notes is meant to help you bridge the gap between the synchronization abstractions and interfaces provided by the Linux kernel for use within the kernel and those provided by the Posix standard for application developers. The Posix mechanisms are designed to embrace a wider variety of approaches than those provided by the Linux kernel, but add little conceptually, and should not be difficult for you to pick up and use. These notes are then intended to help you map from the Posix API into concepts that you have learned in ECE 391 and to explain some of the engineering decisions that went into the Posix API and any other implementations that you might use in the future. Some of the material here was developed originally for ECE 498SL in the Spring 2009 semester. As with the other ECE391 notes, in order to make the notes more useful as a reference, definitions are highlighted with boldface, and italicization emphasizes pitfalls.

1 Posix: What and Why

The acronym POSIX stands for Portable Operating System Interface for uniX, and is used to refer to several standards organized and published by the Institute of Electrical and Electronics Engineers (IEEE) to provide a common basis for programmers to leverage operating system functionality. The API of interest here is the Posix threads API, often referred to as **pthreads** (it now occurs to me that the authors may have read Wodehouse, but most people have not, so the "p" is articulated: "P threads").

As you probably recall, threads usually differ from processes in that two threads in the same process share a single virtual address space. Aside from this difference, thread packages and the meaning of the term thread has varied widely. By the mid-90s, use of threads was becoming common, and several commercial implementations were available (*e.g.*, in Sun's Solaris operating system). As more programmers started to use threads for server codes, portability between packages became more important. However, little or no consensus existed as to exactly how thread functionality should be supported, how threads should be scheduled, which synchronization abstractions should be provided and how they should work, *etc.* To address this problem, a committee was created to define a common approach. In 1995, the committee produced the first Posix threads standard, IEEE POSIX 1003.1c, which standardized the abstractions and interfaces to a variety of synchronization and control mechanisms commonly used with parallelism based on shared memory threads.

As mentioned already, APIs such as Solaris threads had begun to see widespread use by the time the first standard appeared, and support for these models had been integrated into the corresponding operating systems. As a result, Posix implementations were for many years little more than glue sitting on top of the native threads for a given operating system. Even today, nearly a decade and a half later, many operating systems are not fully compliant with the standard. Linux, for example, does not support the idea of specifying the scheduling scope for a thread (a concept explained in Section 2.2).

1.1 Incompatibilities with Threads

Although standards are often necessary for mature abstractions, threads posed a particularly pressing issue because integration of threads into most UNIX systems required changes to other common interfaces. The main problems in this regard were two-fold: undefined semantics for many standard C library routines when used by threads, and incompatibility between the C error model and threads. If individual operating systems vendors were required to solve these problems on their own, the resulting solutions would be highly disparate and difficult to reconcile later.

Let's examine the library routine issues first. The behavior of a library routine is typically defined in terms of its interface: given certain inputs, the routine produces certain outputs, or indicates one of several error conditions. In a sequential program, such an input/output description suffices for using the routine. Introduction of threads, however, creates the possibility of several simultaneous (parallel) executions of any given routine. The meaning of routines not designed for this type of execution are not well-defined. Do they operate as though executed in series according to some (unknown) order? In general, no, but even that constraint, which can be achieved by simply wrapping the code implementing each routine with a routine-specific lock, is not always sufficient to make the routines useful.

A routine with semantics and an implementation that extend naturally to use in multiple threads is said to be **thread-safe**, or **reentrant**. The standard C library contains many routines that do not meet this criterion. One common problem is the use of pointers to static data as return values. For example, the `gethostbyname` routine for finding an Internet (IP) address for a given domain (*e.g.*, "www.illinois.edu") returns a pointer to a static block of data in the library. The caller passes in the domain name; the routine performs the translation, fills in a static data block, and returns a pointer to the block; and the caller then makes use of the data *before calling the routine again*. In a sequential system, this technique avoids the overhead associated with dynamic allocation by either the caller or callee. Delaying a subsequent call, of course, cannot be guaranteed easily in a threaded program. If we cast this exchange as a critical section, the critical section must include consumption of the result, which lies outside of the routine and can thus only be implemented by the caller. In other words, obtaining the expected behavior from these routines in a threaded program implies that the application itself must provide synchronization between its threads when using the routine.

Why not just change the interface? Millions of lines of codes used these routines by the time threads became an issue, thus changing their interfaces was not an acceptable solution. For most of these routines, the Posix standards created new versions that require that the caller pass a pointer to a private block of data, thus decoupling the execution of the routines by multiple threads. The names for these new routines are often simply the original routine name with the suffix "`_r`" appended to indicate a reentrant version. In some cases, the standard introduced entirely new routines and names. Note that a caller can (and typically does) use automatic/stack allocation for the data block, then consumes the result before returning. Most Unix manual (`man`) pages now include a section on standards compliance and thread safety as well as notices to the effect that use of the older, non-reentrant routines is deprecated.

The UNIX error model also posed difficulties for threads. Most C library routines return -1 on failure (NULL for routines that return pointers) and set the `errno` variable to indicate which type of error occurred. But this variable is again in static storage in the C library, where another thread can overwrite it before the value stored by a failed invocation is consumed. To address this problem, most pthreads implementations use the C preprocessor to redirect accesses to the `errno` variable to thread-specific storage. The library implementation itself, of course, must also be re-compiled to make use of this trick, but most implementations allow either model, so code that pre-dates pthreads continues to compile and execute as before.

Posix standards also introduced a new error model to avoid perpetuating these types of problems in any new routines. All Posix routines return 0 on success, or a positive error code on failure. Return values such as pointers or integers are delivered as output arguments. In other words, the caller allocates space for the result and passes a pointer to the space. Although this type of interface adds a few instructions to most call sequences, processors were fast enough by the 1990s that most people agreed that the benefit to software engineering outweighed the cost to performance.

1.2 What to Standardize

The first Posix threads standard addressed the deficiencies in previous Unix interfaces, but also introduced new interfaces for creating, managing, and controlling interactions between threads. For threads, the standard addressed how they are created, what they do, and what happens to them when they terminate. It also discussed methods by which one thread can terminate another asynchronously, an action called **cancellation**. A fairly standard mix of synchronization abstractions was also standardized, including mutexes, reader-writer locks, and condition variables. Semaphores had been standardized two years earlier as part of the Posix real-time standard, and the differences between the semaphore API and those of the other synchronization abstractions reflects the maturity of the standardization process. Interfaces for some other abstractions, such as barriers, were also specified by later versions of the pthreads standard, but are not required to be included in an implementation. Finally, the Posix threads standard specified how each of these abstractions interacted with the operating system scheduler, thread priorities, and other abstractions already present in most Unix implementations. For some interfaces, standardization took a little longer to get right: interactions between threads and signals, for example, were somewhat unstable for a couple of years before eventually reaching the fairly clear rules that exist in the latest version of the standard.

One unifying aspect that clearly differentiates the Posix threads standard from the earlier Posix standards such as the real-time standard is the creation of **attributes structures** that allow applications to create named classes of thread and synchronization variable behavior. As the Posix standard aimed to provide a generic thread interface, the abstractions covered—threads, mutexes, reader-writer locks, and condition variables—include controls to specify the behavior of instances. To avoid race conditions, most choices must be made when a given instance is created. Rather than pass each choice as an argument, the threads standard defines attributes structures that specify all options, and require that an appropriate attribute structure be passed to the routine that creates an instance.

Three benefits of the use of attributes structures are worth noting. First, passing a NULL pointer in place of an attributes structure provides the default behavior, which in many cases may be implementation-dependent. In contrast, all arguments passed to a function must be understood by anyone using the function, even if they need only know a constant value used to request default behavior. Second, as noted earlier, since attributes structures are simply variables like any other piece of data, applications can use the names of the attributes structures to refer to behavioral classes of instances of the corresponding abstraction. Obtaining the same benefit with per-choice arguments requires a set of variables for each behavioral class. Finally, future standards can extend the contents of the attributes structure and assign default options to the new choices *without changing the creation interface*, and thus without affecting existing pthreads code. This benefit cannot be obtained easily when arguments specifying the choices are passed separately.

The semaphore API does not use attributes, but instead uses an argument of the instance initialization function to specify a single choice for behavior. Although the benefit of updating the older interface to reflect the introduction of attributes seems obvious, enough code was probably written in the intervening two years using the initial semaphore API that any change would have undermined confidence in Posix standardization.

1.3 Mechanical Aspects

In Linux, making use of pthreads is often as simple as including `pthread.h` in your source files and linking with `-lpthread`. For some features, you will also need to tell `gcc` that you want to use current Posix standards by including "`-D_XOPEN_SOURCE=500`" as a compile flag. Unfortunately, most commercial implementations serve a group of customers with existing code bases that take priority over any new standards for compilers or libraries, thus you may find that you need to read through a few manual pages before your code works on other systems.

```
// thread management functions
   int pthread_create (pthread_t* new_thread, const pthread_attr_t* attr,
                       void* (*thread_main)(void*), void* arg);
   pthread_t pthread_self ();
   int pthread_join (pthread_t thread, void** rval_ptr);
   void pthread_exit (void* ret_val);

// cleanup functions
   void pthread_cleanup_push (void (*routine)(void* arg), void* arg);
   void pthread_cleanup_pop (int execute);

// cancellation functions
   int pthread_cancel (pthread_t thread);
   void pthread_testcancel (void);
   int pthread_setcanceltype (int type, int* old_type);
   int pthread_setcancelstate (int state, int* old_state);
```

Figure 1: Basic Posix threads interfaces.

2 The Basic Model

Posix represents threads using an opaque type, `pthread_t`. The routines for basic thread management appear in Figure 1. Each thread accepts one pointer as an argument, executes a single function, and returns a pointer as its result. Use of a pointer for arguments and return values allows communication of arbitrary amounts of information by simply creating a structure and using the address of the structure as the information passed to or from the thread in question. Use specifically of a void pointer avoids any warnings for implicit pointer conversions; C compilers silently convert any other pointer type to and from the void pointer type.

The function for thread creation is `pthread_create`. Any thread can create (or **spawn**) a new thread, and a thread can obtain its own identifier using `pthread_self`. The `attr` argument points to an attributes structure from which behavioral options for the thread are drawn. A NULL pointer sets all choices to default values. The function returns 0 when successful, in which case a thread identifier is stored in the location referenced by `new_thread`, and the new thread starts executing the function `thread_main` with the single argument `arg`.

Be careful not to make any assumptions about the relative execution order of the thread being created and the thread that calls the `pthread_create` *function*. For example, if the new thread starts by walking over a data structure, be sure that the data structure is initialized before creating the thread. This type of mistake is particularly troublesome because the scheduling dynamics of your test environment may make bugs from the resulting race condition impossible to observe, while someone else's environment might make them impossible to avoid.

As shown by the signature above, the `thread_main` function returns a void pointer. When a thread created with `pthread_create` returns from its `thread_main` function, the thread terminates, but the return value is retained by the operating system until another thread requests it. Asking for a thread's return value is called **joining** the thread, and is a blocking operation in Posix. Any thread can join with any other by calling `pthread_join`. The calling thread does not return successfully from this function until the target thread has terminated. However, the call can fail due to deadlock prevention (cyclic join attempts) or targeting a non-existent or non-joinable thread. When successful, the target thread's return value is stored in the pointer referenced by `rval_ptr`.

The pthreads standard also defines a block-structured exception handling mechanism. Cleanup routines can be pushed onto a stack of such routines using `pthread_cleanup_push` and subsequently popped off using `pthread_cleanup_pop`. When popping a cleanup routine, the `execute` argument specifies whether or not the cleanup routine should be called. If a thread terminates while the stack is non-empty, all routines on the stack are popped off and executed before termination. These interfaces are nearly identical to those used in the ECE391 maze game; see `assert.h` for details.

A thread created with `pthread_create` can also terminate by calling `pthread_exit`, which handles the stack of cleanup routines and then returns with the specified value. Execution of a `return` statement in a thread function makes an implicit call to `pthread_exit`. The natural way to implement this call is to simply start new threads with a stack frame for a routine that handles cleanups, and then to push a stack frame for the thread's main routine on top of the first stack frame. When a thread's main function returns, it effectively "calls" the cleanup execution routine.

The termination behavior discussed here does not extend to the initial program thread—the one that executes `main`. If the initial thread terminates in any manner, all remaining threads are terminated, and the program ends. Calling `exit`, `abort`, or similar functions from any thread also terminates the program.

2.1 Cancellation

A thread can also terminate in a third way: cancellation by another thread using `pthread_cancel`. A joinable thread should be cancelled if at some point the work that it is performing becomes pointless. In a parallel search to find a single solution, for example, one might launch one thread per processor and then, when one thread finds a solution, cancel the others. Clearly one would not want to wait for all threads to find a solution, which is implied by joining. Cancellation creates race conditions and consistency issues because the thread making the cancellation call and the one being cancelled execute asynchronously. To avoid these problems, Posix allows threads that might be cancelled to select between an asynchronous model in which any critical sections must be protected by appropriate calls to prevent cancellation and a synchronous model in which cancellation is only possible at certain points in the code executed by the thread being cancelled. Cancellation is not instantaneous, and a canceled thread executes its stack of cleanup handlers before terminating. *Posix thus expects that another thread will still join with the cancelled thread to confirm that cancellation has completed.*

New threads start in the mode in which cancellation can only occur at points in the code specified by the thread. In other words, cancellation is possible, but is deferred until the thread itself indicates an appropriate time. To do so, the thread calls `pthread_testcancel`.

If a thread can be cancelled at any point in its execution, it can indicate this fact to the operating system by calling `pthread_setcanceltype` with type `PTHREAD_CANCEL_ASYNCHRONOUS` (the initial cancellation type is `PTHREAD_CANCEL_DEFERRED`); the call stores the previous value in the location referenced by `old_type`.

Frequently, a thread that allows asynchronous cancellation must occasionally enter critical sections within which cancellation should be avoided. In this case, the thread can disable cancellation by calling `pthread_setcancelstate` with state `PTHREAD_CANCEL_DISABLE` at the start of the critical section and `PTHREAD_CANCEL_ENABLE` at the end of the critical section. The call again returns the current value in the location referenced by `old_state`, so a programmer can also create nested critical sections by simply restoring the previous state at the end of the critical section.

2.2 Thread Attributes

As mentioned earlier, Posix encapsulates other options for thread creation into an attributes structure—for a thread, a `pthread_attr_t`—that can be passed optionally when creating a thread. The routines for managing thread attributes appear in Figure 2. Choices available with attributes focus primarily on the scheduling policy for the new thread and on whether or not another thread will eventually join it. An attribute can be initialized to default options for all choices using `pthread_attr_init` and destroyed using `pthread_attr_destroy`.

One option encapsulated by the attributes structure specifies whether or not another thread will eventually join the thread. The problem faced by operating systems with threads is identical to that faced with processes: the return value must be retained in case another thread asks for it at some point in the future. The same is true for processes, and occasionally you can see "zombie" processes in an operating system, which have terminated, but remain visible until some other process asks how they terminated. With Posix threads, one can specify in advance that no other thread will ever request a new thread's return value, allowing the operating system to discard all information about the thread when it finishes executing its function.

```
// initialization and destruction
int pthread_attr_init (pthread_attr_t* attr);
int pthread_attr_destroy (pthread_attr_t* attr);

// joinable/detached state; state can be
//      PTHREAD_CREATE_JOINABLE—thread must be joined eventually
//      PTHREAD_CREATE_DETACHED—thread cannot be joined/cancelled
int pthread_attr_getdetachstate (const pthread_attr_t* attr, int* state_ptr);
int pthread_attr_setdetachstate (pthread_attr_t* attr, int state);

// real-time scheduling inheritance; inher can be
//      PTHREAD_INHERIT_SCHED—inherit scheduling parameters from thread calling pthread_create
//      PTHREAD_EXPLICIT_SCHED—use scheduling parameters in attribute
int pthread_attr_getinheritsched (const pthread_attr_t* attr, int* inher_ptr);
int pthread_attr_setinheritsched (pthread_attr_t* attr, int inher);

// scheduling scope; scope can be
//      PTHREAD_SCOPE_SYSTEM—scheduled by OS scheduler
//      PTHREAD_SCOPE_PROCESS—scheduled at user-level (within process); see notes
int pthread_attr_getscope (const pthread_attr_t* attr, int* scope_ptr);
int pthread_attr_setscope (pthread_attr_t* attr, int scope);

// real-time scheduling policy; policy can be
//      SCHED_FIFO—first-in, first out (FIFO)
//      SCHED_RR—round-robin
//      SCHED_OTHER—other (typically OS-specific)
int pthread_attr_getschedpolicy (const pthread_attr_t* attr, int* policy_ptr);
int pthread_attr_setschedpolicy (pthread_attr_t* attr, int policy);

// real-time scheduling parameters (such as priority); see sched.h
int pthread_attr_getschedparam (const pthread_attr_t* attr,
                                struct sched_param* param_ptr);
int pthread_attr_setschedparam (pthread_attr_t* attr,
                                const struct sched_param* policy);
```

Figure 2: Interfaces for managing Posix thread attribute structures.

A thread for which no join call will be made is called a **detached** thread. A detached thread can be created by calling `pthread_attr_setdetachstate` with value `PTHREAD_CREATE_DETACHED` on a thread attribute structure, then using the attribute structure to create the thread. Although detached threads are attractive in many ways, they can introduce subtle race conditions with program termination. Remember that a pthreads program terminates immediately when its initial thread terminates. Thus, *if you create detached threads and want them to finish their work before the program ends, you must ensure that the initial thread does not return from* `main` *until they have done so*. Joining is one method, but other methods of synchronization are also acceptable, and are necessary with detached threads. Detached threads also create problems with cancellation semantics, to the point that it's best to avoid mixing the two at all.

The remaining options for thread attributes focus on the scheduling behavior of the new thread. The `pthread_attr_setinheritsched` call selects between inheriting behavior from the creating thread (`PTHREAD_INHERIT_SCHED`) and using the behavior specified in the attributes structure itself (`PTHREAD_EXPLICIT_SCHED`). Only in the case of explicit specificaiton by the attribute do any of the other attribute aspects matter in thread creation.

Scheduling scope specifies whether thread scheduling is handled by the operating system or left to the threads themselves (user-level scheduling). In a well-written program, threads that have no work to do take themselves out of the run queue until work becomes available, thus avoiding the use of CPU cycles for idling. Deciding and encoding the reasoning and mechanisms through which each thread puts itself to sleep can be tricky, however, and is problematic when such idle threads merely lead to a slower program.

The scheduling scope option in the attribute—specified using `pthread_attr_setscope`—determines whether a new thread competes under the operating system scheduler, essentially as a regular process, or with other threads in its own process, essentially as a user-level thread. The difference lies in the fact that the OS scheduler is generally oblivious to the distinctions between threads in a process and threads across processes, and thus gives a fair share of CPU cycles to each. In contrast, user-level threads implementations may not implicitly yield to other threads unless they have different priorities.

The scheduling scope option can help in identifying threads that have not properly put themselves to sleep by transforming a performance issue (with OS scheduling) into a deadlock issue (with user-level scheduling), forcing the programmer to deal with the idle threads. Linux does not currently (as of 2.6.26) support the `PTHREAD_SCOPE_PROCESS` option, unfortunately, so you will need to pay close attention to idle loops and sleeping when tuning the performance of threaded code.

The other scheduling options available with attributes are useful primarily for real-time threads. Using `pthread_attr_setschedpolicy`, for example, one can choose between first-in, first-out scheduling (`SCHED_FIFO`) and round-robin scheduling (`SCHED_RR`). These two correspond to the Linux kernel policies with the same names. The third option, `SCHED_OTHER`, is typically used to support OS-specific scheduling strategies. In Linux, `SCHED_OTHER` is the normal, non-real-time scheduling priority, which can also be obtained by inheritance from other threads with normal scheduling priority.

The scheduling parameters for real-time threads can be specified using `pthread_attr_setschedparam`. Only the `sched_priority` field of the `sched_param` structure is used for FIFO and round-robin threads.

3 Synchronization Abstractions

The pthreads standard specifies several useful synchronization abstractions and routines to manage them. As mentioned earlier, an attribute structure is also defined for each abstraction, and can be used to specify the behavior of each instance at creation time. A partial list of routines appears in Figure 3. This list includes only dynamic initialization and destruction of attributes along with the routines used to specify process sharing of synchronization variables. Mutex attributes specify many other options, which are discussed later (and shown in Figure 4).

Synchronization variables in Posix can either be specific to a particular process (`PTHREAD_PROCESS_PRIVATE`) or shared by multiple processes (`PTHREAD_PROCESS_SHARED`). Variables that are private to a given process can still be used by all threads within that process, but can not be used safely by threads in other processes.

```
// initialization and destruction
int pthread_mutexattr_init (pthread_mutexattr_t* m_attr);
int pthread_mutexattr_destroy (pthread_mutexattr_t* m_attr);
int pthread_rwlockattr_init (pthread_rwlockattr_t* rw_attr);
int pthread_rwlockattr_destroy (pthread_rwlockattr_t* rw_attr);
int pthread_condattr_init (pthread_condattr_t* cv_attr);
int pthread_condattr_destroy (pthread_condattr_t* cv_attr);

// inter- or intra-process sharing of variable; shared can be
//     PTHREAD_PROCESS_PRIVATE—use only by one process (multiple threads acceptable)
//     PTHREAD_PROCESS_SHARED—can be used by any process
int pthread_mutexattr_getpshared (const pthread_mutexattr_t* m_attr,
                                  int* shared_ptr);
int pthread_mutexattr_setpshared (pthread_mutexattr_t* m_attr, int shared);
int pthread_rwlockattr_getpshared (const pthread_rwlockattr_t* rw_attr,
                                   int* shared_ptr);
int pthread_rwlockattr_setpshared (pthread_rwlockattr_t* rw_attr, int shared);
int pthread_condattr_getpshared (const pthread_condattr_t* cv_attr,
                                 int* shared_ptr);
int pthread_condattr_setpshared (pthread_condattr_t* cv_attr, int shared);
```

Figure 3: Interfaces for managing Posix synchronization variable attribute structures.

How can more than one process access a mutex, given that each process has its own address space? The processes must first map a common area of physical memory into their respective address spaces using interfaces such as the SysV IPC (Unix System 5 Inter-Processor Communication) or SHMEM (shared memory) routines. Setting up such areas lies outside the scope of these notes, but you should realize that the mechanisms exist in most operating systems.

Why does the difference matter? In some operating systems, process-private mutexes are implemented such that the frequently executed path never need enter the kernel, thereby avoiding the overhead of system calls and improving application performance. While atomic access to the shared memory can also be done for processes, conflicting accesses to synchronization variables generally involve the operating system scheduler, and must thus invoke the operating system. Conflicts for private variables can often be handled by user-level scheduling, as discussed in regard to thread scheduling scope in the previous section.

3.1 Mutexes

The most frequently used synchronization abstraction in pthreads is the **mutex**, which enables mutually exclusive execution of critical sections. Posix mutexes are most similar to the Linux kernel's semaphore abstraction in that a thread that tries to acquire a mutex yields its processor if another thread already holds the lock. Like Linux semaphores, Posix mutexes also maintain queues of threads waiting to acquire them, but also make use of scheduling priority to determine the order of acquisition. Priority information is not available inside the Linux kernel. Posix mutexes can also support multiple acquisitions of a mutex by a single thread, which is convenient with certain programming models.

The Posix mutex type is `pthread_mutex_t`, and the associated routines are shown in Figure 4. A mutex can be initialized statically to `PTHREAD_MUTEX_INITIALIZER` or dynamically using `pthread_mutex_init`. The dynamic initialization call takes a pointer to an attribute structure that specifies options for process sharing, locking behavior, and priority inheritance for the mutex. When a mutex is no longer necessary, it can be destroyed with `pthread_mutex_destroy`.

Once a mutex has been initialized, threads can acquire it with `pthread_mutex_lock`, which returns only after the mutex has been acquired, or can try to lock the mutex once with `pthread_mutex_trylock`. The latter call returns 0 if the lock has been acquired and EBUSY if it has not. In order to release a mutex, a thread should call `pthread_mutex_unlock`.

```
// static and dynamic initialization and destruction
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init (pthread_mutex_t* mutex, const pthread_mutexattr_t* m_attr);
pthread_mutex_destroy (pthread_mutex_t* mutex);

// basic usage
pthread_mutex_lock (pthread_mutex_t* mutex);
pthread_mutex_trylock (pthread_mutex_t* mutex);
pthread_mutex_unlock (pthread_mutex_t* mutex);

// mutex type; type can be
//     PTHREAD_MUTEX_DEFAULT—undefined behavior; may map to OS-specific type
//     PTHREAD_MUTEX_NORMAL—non-recursive
//     PTHREAD_MUTEX_RECURSIVE—recursive (one thread can lock repeatedly); checks for some errors
//     PTHREAD_MUTEX_ERRORCHECK—non-recursive; detects recursive locking deadlocks and other errors
int pthread_mutexattr_gettype (const pthread_mutexattr_t* m_attr,
                               int* type_ptr);
int pthread_mutexattr_settype (pthread_mutexattr_t* m_attr, int type);

// priority inheritance protocol; protocol can be
//     PTHREAD_PRIO_NONE—no inheritance
//     PTHREAD_PRIO_INHERIT—thread that owns mutex inherits priority from waiting thread(s)
//     PTHREAD_PRIO_PROTECT—thread that owns mutex inherits priority from mutex' prioceiling
int pthread_mutexattr_getprotocol (const pthread_mutexattr_t* m_attr,
                                   int* protocol_ptr);
int pthread_mutexattr_setprotocol (pthread_mutexattr_t* m_attr, int protocol);

// priority ceiling; minimum priority for critical sections (see PTHREAD_PRIO_PROTECT protocol above)
int pthread_mutexattr_getprioceiling (const pthread_mutexattr_t* m_attr,
                                      int* prioceiling_ptr);
int pthread_mutexattr_setprioceiling (pthread_mutexattr_t* m_attr,;
                                      int prioceiling);

// priority ceiling; minimum priority for critical sections (see PTHREAD_PRIO_PROTECT protocol)
int pthread_mutex_getprioceiling (const pthread_mutex_t* mutex,
                                  int* prioceiling_ptr);
int pthread_mutex_setprioceiling (pthread_mutex_t* mutex, int prioceiling);
```

Figure 4: Interfaces for Posix mutexes and mutex attribute structures. Attribute routines for initialization, destruction, and process sharing appear in Figure 3.

```
// initialization and destruction
int sem_init (sem_t* sem, int shared, unsigned int val);
int sem_destroy (sem_t* sem);

// basic usage
int sem_wait (sem_t* sem);
int sem_trywait (sem_t* sem);
int sem_post (sem_t* sem);

// read current value
int sem_getvalue (sem_t* sem, int* val_ptr);
```

Figure 5: Interfaces for Posix semaphores.

The default locking behavior for Posix mutexes is undefined, and although in most implementations the behavior defaults to one of a few pre-defined types, relying on default behavior is risky. Selection of the mutex type must be done with dynamic initialization using a `pthread_mutexattr_t` configured with `pthread_mutexattr_settype`. The type `PTHREAD_MUTEX_NORMAL` is a non-recursive lock optimized for speed. As you might expect, a thread that attempts to acquire such a mutex more than once deadlocks with itself. A recursive mutex type, `PTHREAD_MUTEX_RECURSIVE`, is also available, and allows a single thread to repeatedly lock the mutex. The number of calls to unlock must match the number of successful calls to lock. Also, if the count of recursive lock calls gets too high, *an attempt to lock the mutex can return EAGAIN*. The recursive lock type also checks for some errors, such as a mutex being unlocked by a different thread than the one that locked it. A third type—also non-recursive—that performs more error-checking is available to help debug mutex usage. This type returns errors when a thread attempts to re-acquire a mutex that it has already locked and when a thread attempts to unlock a mutex that it has not locked. *It does not, at least in Linux 2.6.26, detect deadlocks based on lack of consistent lock acquisition ordering between threads.* Note that no type of pthread mutex allows implicit passing of a locked mutex from one thread to another. If such an action is attempted, the result is undefined for normal mutexes and an error for recursive and error-checking mutexes.

A mutex attribute structure can also be used to configure mutexes so as to avoid **priority inversion**, which occurs when a low priority thread holds a mutex that a high priority thread is waiting to acquire. Priority inversion can lead to further problems; for example, a medium priority thread may preempt the low priority thread that is in turn blocking the high priority thread, further delaying the most important thread in the system. By using `pthread_mutexattr_setprotocol` on a mutex attribute, one can choose no inheritance (`PTHREAD_PRIO_NONE`), inheritance from the highest priority task waiting for a mutex (`PTHREAD_PRIO_INHERIT`), or inheritance from the mutex itself (`PTHREAD_PRIO_PROTECT`). A thread holding a mutex executes at the higher of its own priority and any **inherited priority**. Inheriting priority from a mutex may enable threads to finish their critical sections before high priority threads need the mutex, but can also cause other forms of inversion, such as preemption of a high priority thread due to execution of a critical section at even higher (inherited) priority despite the fact that no thread with the higher priority thread actually needs the mutex. If inheritance from the mutex is chosen, the priority value can be set either at creation time via an attribute and `pthread_mutexattr_setprioceiling` or directly in the mutex with `pthread_mutex_setprioceiling`. Regardless of the protocol chosen for a mutex, threads waiting to acquire the mutex are always sorted in order of priority, so the highest priority currently waiting will obtain a mutex when the current owner calls `pthread_mutex_unlock`.

3.2 Semaphores

Semaphores support a generalization of mutual exclusion in which a specific maximum number of threads can execute critical sections simultaneously. A semaphore with a limit of one thread is thus equivalent to a mutex in theory. In practice, Posix semaphores lack some of the more important functionality provided with Posix mutexes, including support for priority inheritance and cleanly defined signal interactions.

```
// static and dynamic initialization and destruction
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
int pthread_rwlock_init (pthread_rwlock_t* rwlock,
                         const pthread_rwlockattr_t* attr);
int pthread_rwlock_destroy (pthread_rwlock_t* rwlock);

// basic usage
int pthread_rwlock_rdlock (pthread_rwlock_t* rwlock);
int pthread_rwlock_tryrdlock (pthread_rwlock_t* rwlock);
int pthread_rwlock_wrlock (pthread_rwlock_t* rwlock);
int pthread_rwlock_trywrlock (pthread_rwlock_t* rwlock);
int pthread_rwlock_unlock (pthread_rwlock_t* rwlock);
```

Figure 6: Interfaces for Posix reader/writer locks.

The Posix semaphore interface was defined as part of the real-time extensions rather than the pthreads standard. The concept of attributes had not been incorporated at the time, and was probably hard to incorporate later due to the existence of code using the original interfaces. Semaphores thus require that the process sharing option be passed directly as an argument to the dynamic initialization function. *Similarly, the Posix error model had not yet been defined, so semaphore calls return 0 on success and -1 on failure.*

The Posix semaphore type is `sem_t`, and the associated routines appear in Figure 5. A semaphore can be initialized to a specific value (maximum number of concurrent threads) with `sem_init` and destroyed with `sem_destroy`. A semaphore can be decremented (tested, downed, or P/proberen) using `sem_wait` and incremented (posted, upped, or V/verhogen) using `sem_post`. The `sem_wait` call blocks when the semaphore's value is 0. Unlike pthreads synchronization routines, *the* `sem_wait` *call can return EINTR if a signal is delivered to a thread while blocked inside of the routine*. To attempt to decrement a semaphore without blocking, a thread can instead call `sem_trywait`, which returns 0 on success. Whether or not semaphores support prioritization of subsequent acquisitions depends on compile-time constants; see the `sem_post` man page for details. Finally, a thread can read the value of a semaphore with `sem_getvalue`, although of course the result must be assumed to be stale, since nothing prevents other threads from incrementing or decrementing the semaphore between execution of the call and consumption of the return value.

3.3 Reader/Writer Locks

A mutex serializes all accesses to protected data, even those accesses that cannot interfere with each other. Data that are read frequently but modified rarely can be protected with a **reader/writer lock** to enable parallel read accesses. A reader/writer lock allows two types of locking actions: **read locks** and **write locks**. In theory, an infinite number of threads can simultaneously hold read locks, while a write lock guarantees mutual exclusion. In particular, no other thread can hold either form of the lock while any thread holds a write lock.

Reader/writer locks may be prone to **writer starvation**, in which a thread waits indefinitely for a write lock while other threads continuously acquire and release read locks. Since a write lock can only be acquired when no thread holds any type of lock (read nor write), allowing readers to acquire the lock after a writer has started waiting introduces the possibility of starvation. Implementations can easily avoid this problem by disallowing later readers from acquiring read locks until the writer has had a turn with the lock, but doing so reduces parallelism, since a new reader may be able to complete its critical section and release its read lock before a previous reader has done so. *In Posix, whether or not reader/writer locks admit writer starvation is an implementation-dependent decision.*

The Posix reader/writer lock is called a `pthread_rwlock_t`, and the associated routines are shown in Figure 6. A reader/writer lock can be initialized statically to `PTHREAD_RWLOCK_INITIALIZER` or dynamically using `pthread_rwlock_init`. As with all pthreads synchronization types, the dynamic initialization call takes a pointer to an attribute structure, in this case simply to specify whether the reader/writer lock is for use within a single process or may be accessed by multiple processes (see `pthread_rwlockattr_setpshared` in Figure 3). Once a reader/writer lock is no longer necessary, it can be destroyed with `pthread_rwlock_destroy`.

```
// static and dynamic initialization and destruction
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
int pthread_cond_init (pthread_cond_t* cond, const pthread_condattr_t* cv_attr);
int pthread_cond_destroy (pthread_cond_t* cond);

// sleeping on a condition
int pthread_cond_wait (pthread_cond_t* cond, pthread_mutex_t* mutex);
int pthread_cond_timedwait (pthread_cond_t* cond, pthread_mutex_t* mutex,
                            const struct timespec* delay);

// waking threads when a condition becomes true
int pthread_cond_signal (pthread_cond_t* cond);
int pthread_cond_broadcast (pthread_cond_t* cond);
```

Figure 7: Interfaces for Posix condition variables.

After a reader/writer lock has been initialized, threads acquire a read lock with `pthread_rwlock_rdlock` or a write lock with `pthread_rwlock_wrlock`. These calls both block until the lock becomes available. Alternatively, a thread can attempt to obtain the lock by calling `pthread_rwlock_tryrdlock` or `pthread_rwlock_trywrlock`. Each of these calls returns 0 on successful lock acquisition or and gives up and returns EBUSY if the lock cannot be acquired immediately. Real implementations of course allow only a finite number of readers. In current versions of Linux, the number allowed typically exceeds the total number of threads that can exist in the system, but *implementations are allowed to return EAGAIN from the read lock calls to indicate that too many readers are currently holding the lock.* When a thread is done with a reader/writer lock, it must call `pthread_rwlock_unlock` to release the lock. Both lock types use the same unlock call.

3.4 Condition Variables

The synchronization abstractions discussed so far are primarily used to enforce atomicity between execution of threads. Another common requirement for multi-threaded programs is the ability to express dependences between threads and to ensure that these dependences are met. The simplest case is two threads with a producer-consumer relationship: one thread produces objects of some type, and the second thread consumes them, usually by executing some piece of code on each object. If the first thread produces more slowly than the second thread consumes, the second thread must at some point wait for the first, and should relinquish any processor resources while it waits.

Condition variables encapsulate the notion of a thread waiting on a particular condition. They provide a good example of a **race condition**, in which the order in which two threads complete their own actions can mean the difference between a program that works and a program that does not. Consider again a consumer thread that is waiting for a producer thread to finish working on an object for the consumer thread. The consumer thread should simply put itself to sleep to avoid wasting processor cycles, which might be of use to other threads in the system, and may possibly even be used by the producer thread.

The difficulty, however, lies in the race between the consumer putting itself to sleep and the producer waking it up. If, for example, the consumer checks for the object, then puts itself to sleep if the object is not ready, while the producer first finishes creating the object and then wakes the consumer, the following sequence can occur. First, the consumer checks for the object and finds it unavailable. Next, the producer finishes its work and wakes the consumer; as the consumer is already awake, this action has no effect. Finally, the consumer puts itself to sleep. Depending on the schedling scope of the thread, the cost is either a performance hit (operating system scheduling) or a program that never terminates (user-level scheduling).

One way to ensure that this sequence of events cannot occur is to make use of a mutex lock that is held by the consumer during its entire sequence and by the producer when it wants to wake the consumer, thus preventing the wakeup from occurring between the consumer's check and its putting itself to sleep. Posix condition variables make use of this approach.

A condition variable in Posix has type `pthread_cond_t`, and the associated routines appear in Figure 7. A condition variable can be initialized statically to `PTHREAD_COND_INITIALIZER` or dynamically using `pthread_cond_init`, which takes a pointer to an attribute structure specifying whether or not the condition variable may be accessed by multiple processes (see Figure 3). A condition variable can be destroyed with `pthread_cond_destroy`.

The sequence for waiting on a condition is then as follows. First, some mutex must be associated with the condition variable. The mapping need not be one to one; in other words, one mutex may serve for several condition variables. The condition can be checked before acquiring the mutex if desired (and *if it is safe to do so*), but this check does not suffice, and is generally only a performance optimization: if the condition is already true, sleeping is not necessary. The thread then acquires the mutex and, while holding the lock, checks the condition, which can be any function on data structures that can be accessed safely with the mutex. If the condition is false, the thread puts itself to sleep by calling `pthread_cond_wait`. Note that the call is made while holding the mutex, and the mutex is passed to the call. Inside the system call, the operating system releases the mutex to ensure that the thread that will eventually wake the sleeping thread can acquire the mutex without deadlocking. The thread is then put to sleep. When the thread is woken later, the system call reacquires the mutex before returning from `pthread_cond_wait`. The calling thread can then execute with or without the mutex, but must eventually release it.

Conditions should be specified clearly and carefully and documented well so as to simplify the process of identifying all code that might change the condition. *If a thread changes a condition without trying to wake any threads sleeping on the condition, the sleeping threads may never wake.* Inverting the condition so as to identify code that must wake sleeping threads can be challenging, and can sometimes motivate use of different, more easily inverted conditions. On a similar note, while in some cases one might be able to prove that no threads can be sleeping for a specific condition change, keep in mind that the proof must continue to hold true under any future changes to code in order to justify leaving out wakeup code.

The functions that wake sleeping threads do not take mutex pointers as arguments. Since the calling thread often needs to extend the critical section guarded by the mutex beyond the scope of the call, the system call cannot manipulate the mutex. However, the calling thread must acquire the mutex corresponding to the `pthread_cond_wait` call at some point between changing the condition and waking sleeping threads in order to avoid the race condition described at the start of this section. Typically, a thread acquires the appropriate mutex, calls either `pthread_cond_signal` to wake a single sleeping thread or `pthread_cond_broadcast` to wake all sleeping threads, and releases the mutex.

```
// managing thread-specific data
int pthread_key_create (pthread_key_t* key_ptr, void (*destructor)(void* arg));
int pthread_setspecific (pthread_key_t key, const void* value);
void* pthread_getspecific (pthread_key_t key);
int pthread_key_delete (pthread_key_t key);

// execute exactly once semantics
pthread_once_t state = PTHREAD_ONCE_INIT;
int pthread_once (pthread_once_t* once, void (*function)(void));
```

Figure 8: Miscellaneous pthreads interfaces.

4 Miscellaneous Interfaces

Before concluding these notes, two other sets of pthreads routines are worth describing: management of thread-specific data and execute exactly once semantics. The functions are listed in Figure 8.

The **thread-specific data** routines enable each thread to create and use private named variables. Static variables play such a role in sequential programs, but are unique in the program image and must therefore be shared amongst all threads and handled carefully to avoid race conditions, manage contention, and so forth. Each thread does maintain a private stack area to support the illusion of processor virtualization amongst threads, but the stack is only useful for automatic variables. Similarly, threads can use dynamic memory management routines for heap allocation. For static variables, an abstraction that maps variable names to variables on a per-thread basis is useful.

The pthreads thread-specific data abstraction is a set of key-value pairs. The key is the variable "name," and has type `pthread_key_t`. The value is an arbitrary block of data referenced by a void pointer. A key must first be created with `pthread_key_create`. By default, the value associated with any thread for a particular key is NULL. To associate a value for a particular thread, the thread must call `pthread_set_specific`, while `pthread_get_specific` returns the current value. When a thread with a non-NULL value terminates, the destructor function passed to `pthread_key_create` is invoked with argument equal to the thread's value (the value associated with the key in the terminating thread). The destructor is invoked repeatedly so long as the value associated with the thread is not NULL, thus *the destructor must explicitly set the thread's value to NULL*. Finally, a key can be destroyed by calling `pthread_key_delete`. This function is safe to call from destructors, but *does not invoke the destructor function* on remaining non-NULL values.

The second set of miscellaneous support routines allows multiple threads to execute a routine exactly once. This capability is useful for dynamic module initialization, when the initialization routine must be called before other functions, but should only be called once. If the initialization cannot for some reason be performed before creation of threads, or is too expensive to execute unless the module is actually necessary, pthreads' **execute exactly once semantics** provides a solution.

Execution is synchronized using a state variable, using a simple atomic transition coupled with delay of any other executions that overlap the first. The state is a `pthread_once_t`, and is initialized to `PTHREAD_ONCE_INIT`. Each thread then calls `pthread_once` with pointers to the state variable and the routine to be executed exactly once. No thread returns from this call until the execution is complete, but the routine will only be executed one time by one thread.