POSIX Threads API

Overview

- The POSIX threads API is designed to allow maximum freedom to implementors
- This means the API is not "robust" against mis-use
- This means the semantics of several operations are only very loosely defined, and so may have quite different outcomes, depending on the implementation
- This means the programmer must exercise extreme discipline in order to write code that will execute correctly on all implementation, under all scenarios
- It is very easy to write incorrect code and not be aware of it, because tests produced the expected results for one implemention, all or most of the time.

These notes are intended only to provide a brief introduction to the POSIX threads API. They are not intended to fully duplicate the on-line manual pages. For detailed explanations of signals, handlers, and the operations that generate signals, use the Unix *man* command to view the on-line manual pages, or look in a book about the Unix API or threads. Links to some on-line tutorials on threads programming are given at the end of these notes.

In all the examples here, correct code must check the function return values, to verify that the calls succeeded. Those checks are omitted from the example code fragments, for easier readability. That does not mean you can or should omit the checks in an actual program.

Threads

A POSIX *thread* is a single flow of control within a process.
It shares a virtual address space with other threads in the same process.
The following are per-thread attributes:

- execution stack (within shared address space)
- set of blocked signals (the *signal mask*)
- set of signals pending for the thread

- scheduling policy and priority
- *errno* value
- thread-specific key-to-attribute mapping

Header Files you Need for Thread Programming

```
#define _REENTRANT
#include <pthread.h>
```

```
#include <sched.h>
```

_REENTRANT indicates that the reentrant (*i.e.* thread safe) versions of the standard libraries should be used.

The header file *pthread.h* defines the POSIX thread API.

The header file *sched.h* defines the process and thread scheduling API. Of the functions explained here, only *sched_yield()* requires it.

Reentrancy

- safe for concurrent execution with itself
- stronger than recursive
- cannot share copies of local variables (*i.e.*, no C *static* local declarations)
- non-reentrant code is not safe for use with threads
- standard versions some C library functions may not be reentrant
  (Why?)

A subprogram is *reentrant* if it is safe for it to be called concurrently with another call to itself. It is important that all functions that are called by threads must be reentrant.

One of the things that can make a subprogram non-reentrant is for it to access the same memory that is accessed by another call to the same subprogram. Of course this could happen if the subprogram accesses parameters that are passed by reference, but we usually do not say that alone makes it non-reentrant, since it could be perfectly safe so long as the callers take care not to call it with the same arguments concurrently. On the other hand, if the subprogram accesses global variables or statically allocated local variables without any locking mechanism to protect them from concurrent accesses, it is not reentrant. For example, if a function written in the C language has local variables that are declared *static* chances are it is not reentrant.

For programs written in the C language, if the operating system supports threads, reentrant versions of the functions in the standard libraries are supposed to be provided. However, these functions are likely to to contain locking/unlocking code to protect shared variables, which adds some execution time overhead. So that programs that do not use multithreading do not pay this overhead, the system default may be to link in the non-reentrant libraries. To make sure the system uses the reentrant versions, the application should define the constant *_REENTRANT* before any *#include* directives. This directive will change the effects of the header files, through conditional compilation (*#ifdef*) directives embedded in the files.

Creating a Thread

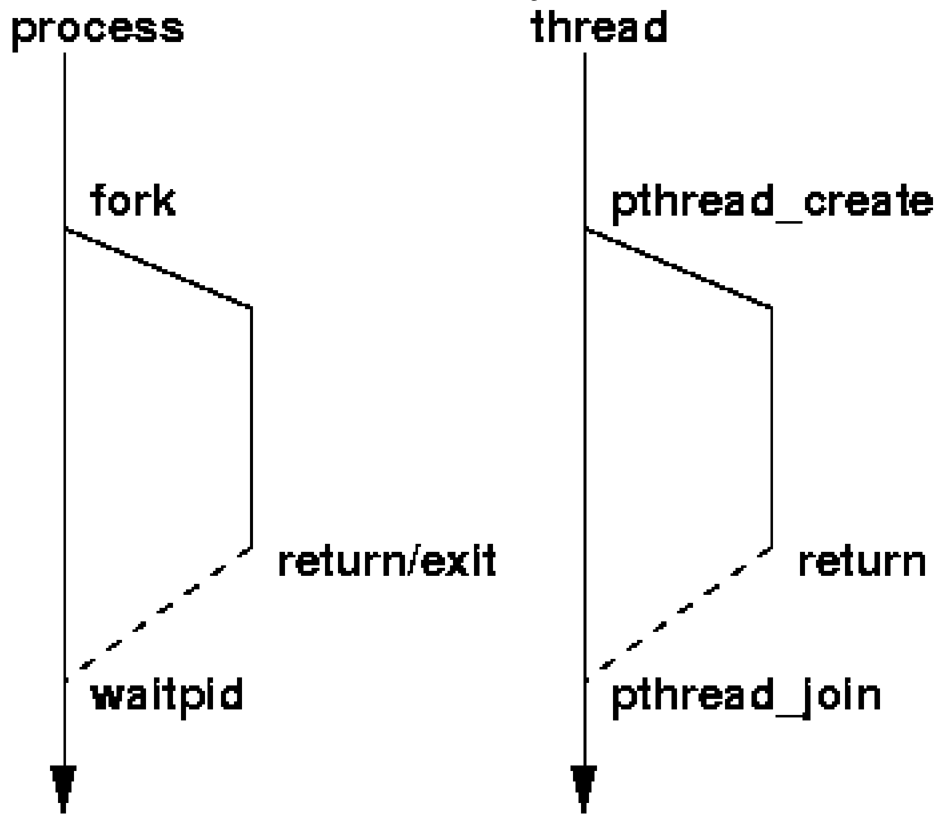
```
#include <pthread.h>
int pthread_create(pthread_t *new_thread_ID,
    const pthread_attr_t *attr,
    void * (*start_func)(void *), void *arg);
```

To create a thread, you need to specify a place to store the ID of the new thread, the procedure that the thread should execute, optionally some thread creation attributes, and optionally an argument for the thread. The call returns a result code.

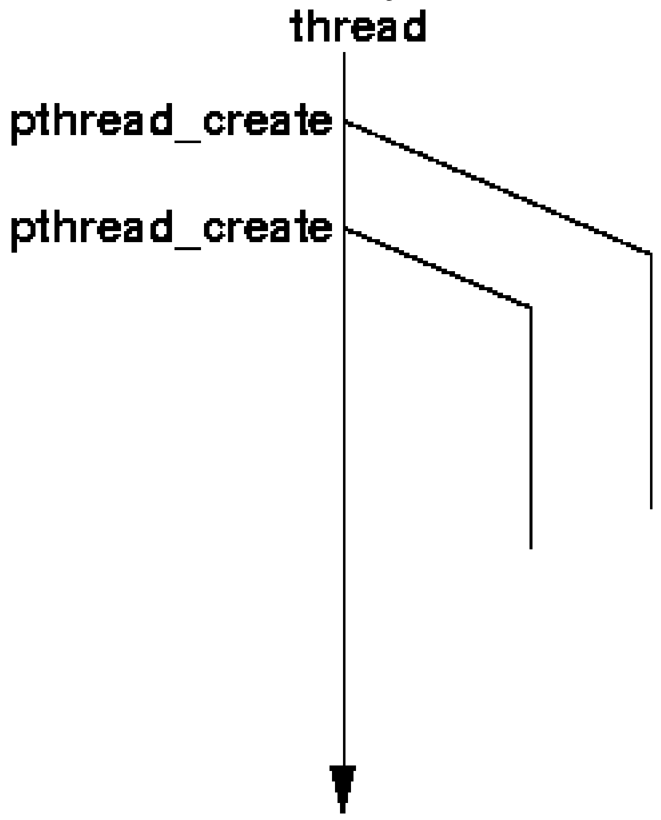
Joining with a Thread

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

By default, threads are created *joinable*. This means that some other thread is required to call *pthread_join* to collect a terminated thread, in a fashion similar to the requirement for a parent process to collect status for terminated child processes.



The use on threads of *pthread_join()* is similar to the use on processes of *waitpid()*. Details differ somewhat, but the main effect is for one thread to wait for (join with) the completion of another thread. The term "join" reflects the joining of control flows, as indicated in the diagram.



If you write a main program that does not use join to wait for its threads, when the main thread exits all the component threads will be aborted. They will just be interrupted and terminated.

Detached Threads

A *detached* thread is not joinable. As soon as the thread terminates its resources may be recovered.

Thread Creation Attributes

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
...
int pthread_attr_setdetachstate(pthread_attr_t *attr,
  int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
  int *detachstate);
int pthread_attr_setstacksize(pthread_attr_t *attr,
  size_t stacksize);
int pthread_attr_getstacksize(const pthread_attr_t *attr,
  size_t *stacksize);
...
```

Certain attributes of a thread, including its priority and whether it is created joinable or detached, are specified via a parameter of type *pthread_attr_t*.

Creating a Detached Thread

```
#include
#include <string.h>
pthread_attr_t attr;
pthread_attr_init (&attr);
pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
result = pthread_create (&server, &attr, thread_body, connfd);
if (result) fprintf (stderr, "pthread_create: %s", strerror (result));
```

The thread operations generally do not use *errno*. Instead, they return the error code as the function value.

Getting the ID of the Current Thread

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

This function can be called to get the ID of the currently executing thread. It is serves for threads the same purpose as *getpid()* serves for processes.

Protecting a Critical Section with a Mutex

```
pthread_mutex_t M;
pthread_mutex_lock (&M);
... critical section ...
pthread_mutex_unlock (&M);
```

This example is simplified. It does not check for error codes.

A mutex is a memory-based data object that is used to implement mutual exclusion. The intent it so provide the kind of protection that is needed to implement a monitor. Mutexes are designed to provide the mutual needed for a monitor. They are designed to work with condition variables, which provide the rest of the monitor support.

On some systems mutexes are likely to only be supported for use between threads within a single process.

What happens if pthread_init() is called more than once on the same mutex?

The man-page, which is based on the POSIX standard, says that this "results in undefined behavior". This means that if you try to initialize a mutex more than once you are to blame for whatever happens. Your program might crash, in some unpredicatable way, or it might just reinitialize the mutex. If you make any assumptions about what happens, your code is certainly not portable.

Mutex Operations

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mp,
   const pthread_mutexattr_t *attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_lock(pthread_mutex_t *mp);
int pthread_mutex_trylock(pthread_mutex_t *mp);
int pthread_mutex_unlock(pthread_mutex_t *mp);
int pthread_mutex_destroy(pthread_mutex_t *mp);
```

Mutex Operation Semantics

- pthread_mutex_lock (&M)
  - blocks the calling thread until the mutex becomes unlocked, if necessary
  - locks the mutex M
  - only one thread can hold the lock on the mutex at any one time
  - may not check for most errors, including the error of locking a mutex whose lock is already held by the calling thread
  - effects of errors are mostly undefined
- pthread_mutex_unlock (&M)
  - unlocks the mutex M
  - may not check for most errors, including the error of unlocking a mutex whose lock is not held by the calling thread
  - effects of errors are mostly undefined

Mutex Creation Attributes

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
  int process-shared);
int  pthread_mutexattr_getpshared(const  pthread_mutexattr_t *attr,
  int *process-shared);
```

For mutex creation, like thread creation, one uses an attribute object to pass attributes of the mutex that is being created.

Creating A Mutex

```
pthread_mutex_t M;
pthread_mutexattr_t attr;
pthread_mutexattr_init (&attr);
pthread_mutex_init (&M, &attr);
```

This example is simplified. It does not check for error codes.

Condition Variable Operation Semantics

- pthread_cond_wait (&CV, &M)
  - should only be called by a thread that holds locked mutex M
  - unlocks the mutex M
  - gives other threads a chance to execute
  - ***may*** suspend the calling thread until CV is signalled
  - relocks the mutex (blocking until the mutex becomes available) before returning
  - must always be used in a loop that polls for a meaningful logical condition involving other variables
  - may not check for most errors
  - effects of errors are mostly undefined
- pthread_cond_signal (&CV)

- does not require that any mutex lock be held
- ensures one or more of the threads waiting on CV (if any) wake up (and begin trying to acquire the associated mutex)
- does not check for most errors, including the error of unlocking a mutex whose lock is not held by the calling thread
- effects of errors are mostly undefined

Condition Variables

- Have ambiguous implementation semantics (see further below)
- Are not variables
- Do not represent conditions
- To be used, require an associated mutex
- Require a polling loop that checks a logical condition

The tricky thing about CV's is that the POSIX standard intentionally leaves a lot of freedom to the implementor, so the CV functions can do different things on different systems. This means that a programmer must write code more carefully than usual, so that the code will work for all legal implementations of CV's.

Some Legal Condition Variable Implementations

The following are pseudo-code descriptions of some permissible implementations.

pthread_cond_wait(&CV,&M)	**pthread_cond_signal(&CV)**

pthread_mutex_unlock(&M);
sched_yield();
pthread_mutex_lock(&M);
	(do nothing)

pthread_mutex_unlock(&M);
CV.enqueue (This_Thread);
This_Thread.sleep;
pthread_mutex_lock(&M);
	T = CV.dequeue_one;
	T.wakeup;

pthread_mutex_unlock(&M);
CV.enqueue (This_thread);
	while (CV.queue_nonempty) {
	T = CV.dequeue_one;

This_thread.sleep;
pthread_mutex_lock(&M);
	T.wakeup;
	}

The main thing about *pthread_cond_wait* is that it must release the mutex, give other threads a chance to lock (and then unlock) the mutex, and then relock the mutex.

Whether the thread goes to sleep during the interval in which the mutex is unlocked is up to the implementation.

The main thing about *pthread_cond_signal* is that if any threads are sleeping on the CV, at least one of them will wake up, lock the mutex, and recheck the logical condition.

Whether it wakes up one thread or all the threads that are waiting on the CV is upt to the implementation.

Synchronizing Threads with a Condition Variable

```
pthread_cond_t C;
pthread_mutex_t M;
int State = 0;
```

Waiter:

```
pthread_mutex_lock (&M);
while (! State) pthread_cond_wait (&C, &M);
pthread_mutex_unlock (&M);
```

Signaler:

```
pthread_mutex_lock (&M);
State = 1;
pthread_cond_signal (&C);
pthread_mutex_unlock (&M);
```

Condition variables are generally only supported for use between threads within a single process.

Condition variables can only be used with a mutex.

Condition Variable Operations

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond,
   const pthread_condattr_t *attr);
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 *abstime);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Yielding to Another Thread

```
#include <sched.h>
int sched_yield (void)
```

Calling this function invokes the scheduler, which may switch execution to another thread.

Initializing Shared Data Just Once

```
     #include <pthread.h>
     pthread_once_t once_control = PTHREAD_ONCE_INIT;

     int    pthread_once(pthread_once_t    *once_control,    void
     (*init_routine, void)););
```

There are situations where a thread may need to initialize some global data, but it cannot tell in advance whether another thread might already have initialized the same data. It may be unsafe to initialize something more than once. For example, the Solaris man-page for *pthread_mutex_init()* says:

> Attempting to initialize an already initialized mutex results in undefined behavior.

That makes sense. For example, what would happen if we reinitialize a mutex that is already locked and may already have some threads blocked on it?

To see what happens, take a look at program examples/threads/init_bad.c, in directory examples/threads. Better, compile it and execute it.

Naive Solution to Initialization Problem

```
void initialize () {
   static int init_flag = 0;  /* is initialized to 0 at process start time */
   if (! init_flag) {
      /* window for race between threads is here
         if two threads fetch the value 0 before one increments init_flag,
         both will execute the initialization code, and if that happens
         there may be a problem
      */
      init_flag++;
      pthread_mutex_init (&M, NULL);a
   }
   pthread_mutex_lock (&M);
   /* now initialize other global data, under protection of the mutex */
   pthread_mutex_unock (&M);
}
```

This solution tries to use a static flag variable to prevent multiple initializations.

As the comments in the code above indicate, this is not 100% safe. It is possible (though unlikely) that two threads might fetch the value of *init_flag* concurrently and both find the value to be zero, in which case *pthread_mutex_init()* would be called twice on the same mutex. Things that can go wrong but usually don't are the worst nightmare of concurrent programmers, because when they do go wrong the error is very hard to track down. For this reason, a good concurrent programmer would not be satisfied with the above "solution".

There is a special thread API function to solve this problem, pthread_once. It can be used as follows, to solve our mutex initialization problem.

The Correct Solution: using `pthread_once_t`

```
void init_routine () {
    pthread_mutex_init (&M, NULL);
}

void initialize () {
    static pthread_once_t init_flag = PTHREAD_ONCE_INIT;
    /* is initialized at process start time */
    pthread_once (&init_flag, init_routine);
    pthread_mutex_lock (&M);
    /* now initialize other global data, under protection of the mutex */
    pthread_mutex_unock (&M);
}
```

Note that the idea is basically the same as our first attempt. The variable *init_flag* is used to indicate whether the initialization has been done yet. The difference is that the function *pthread_once* is guaranteed to ***atomically*** test and modify the flag.

For complete examples of both solutions, see files examples/threads/init_weak.c and examples/threads/init_good.c.

Thread Priorities

- Threads scheduling policies may be specified
- Within policies, priorities may be specified
- Policy and priority may be specfied at creation time, or a run time, by the affected thread or by another

Thread Scheduling Policies

- SCHED_FIFO
  - preemptive priority scheduling
  - highest priority thread runs until it choses to block/sleep
  - when it unblock/wakes it goes to the end of the queue for its priority
- SCHED_RR
  - preemptive priority scheduling
  - highest priority thread runs until it choses to block/sleep, or is suspended because it has used up its time slice (quantum)
  - when it unblock/wakes it goes to the end of the queue for its priority
  - when it uses up its time slice it also goes to the end of the queue for its priority
- SCHED_OTHER
  - an implementation-dependent scheduling policy
  - the default
  - generally lower priority than SCHED_RR and SCHED_FIFO

Setting Thread Scheduling Policies & Priority at Creation

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

pthread_t th;
pthread_attr_t attr;
```

```
struct sched_param parm;
// int scope;

pthread_attr_init (&attr);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
parm.priority = sched_get_priority_max (SCHED_FIFO) - 1;
/* next-to-highest priority */
pthread_attr_setschedparam(&attr, SCHED_FIFO, &parm);
// scope = PTHREAD_SCOPE_SYSTEM;
// pthread_attr_setscope(&attr, &scope);
// pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&th, &attr, some_function, some_argument);
```

The commented-out lines are examples of other thread creation attributes that one might want to set, that do not directly relate to priority.

As in all the examples here, correct code must check the function return values, to verify that the calls succeeded.

If you don't set the scheduling policy and priority at the time of creation, you may set it later, but beware that between times the thread will be scheduled according to the system default policy, or according to the policy inherited from the parent. The latter can be requested using *pthread_attr_setinheritsched()*.

Setting Thread Scheduling Policies Later

- *pthread_setschedparam()* : can set the policy and priority of a thread
- *pthread_setschedprio()* : can set just the priority of a thread

---

There is also a function *sched_setscheduler()*, that allows one to set the scheduling policy and priority of a process. The effect on multi-threaded processes does not appear to be defined by the standards. A reasonable guess at the effect is: (1) if the process has more than one thread, this will probably only affect the scheduling of the main thread; (2) or, if there are threads with PTHREAD_SCOPE_PROCESS, it may affect their scheduling as a group.

Setting Concurrency Level

- A two-level thread implementation may not give every user thread a kernel thread
- To see speed-up from multiple CPU's more kernel threads may be necessary
- Solaris allows user to specified desired concurrency level

```
#include <pthread.h>
int ptread_getconcurrency(void);int
pthread_setconcurrency(int new_level);
```

The Solaris *man* page for *pthread_setconcurrency()* says:

```
Unbound threads in a process may or may not be  required  to
be  simultaneously active. By default, the threads implemen-
tation ensures that  a  sufficient  number  of  threads  are
active  so  that  the process can continue to make progress.
While this conserves system resources, it  may  not  produce
the most effective level of concurrency.

The  pthread_setconcurrency() function allows an application
to  inform  the  threads  implementation of its desired con-
currency level, new_level. The actual level  of  concurrency
provided  by the implementation as a result of this function
call is unspecified.
```

An application should specify a concurrency level higher than one if it expects the threads of a process to make use of more than one physical processor of an SMP system.

Thread-specific Data

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

See the complete example program in perthread.c in examples/threads.

These functions provide a mechanism for a library function to retrieve thread-specific data. The example program cited above shows how they can be used.

This feature amounts a database, in which the identity of the current thread (implicit) and another data value (of type pthread_key_t) can be used as keys to store and retrieve a pointer value. It is intended for the implementors of thread-safe libraries, to keep data that must be different for each thread. (For example, consider the global variable *errno*.) Different libraries may install different keys, and so each have their own private thread-specific data.

- Create a thread attribute key, one time, to be used for all threads:

```
pthread_key_t new_key;
  pthread_key_create (&new_key, my_destructor_function));
```

- Once for each thread, in the thread creator or in the thread-specific initialization code, allocate a record structure to hold the data, initialize it, and "hang" a pointer to it from the thread with the given key, using *pthread_setspecific*:

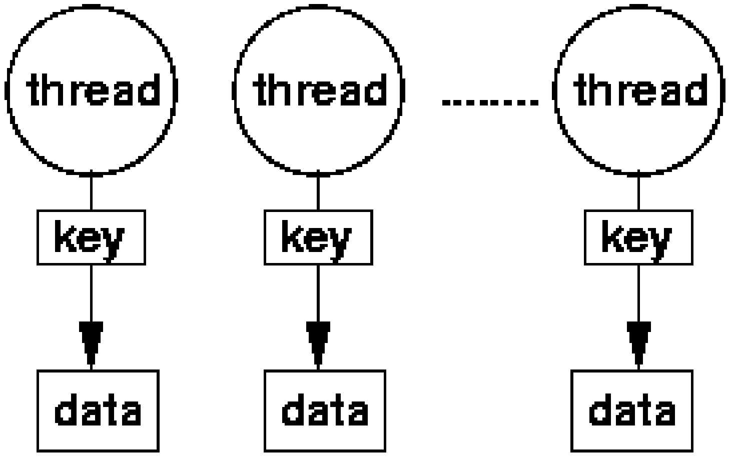
```
p = (perthread_data_ptr) malloc (sizeof (struct perthread_data));
  ... initialize data fields of struct pointed to by p ...
```

pthread_setspecific (my_key, (void *) p));

- Whenever you need to get the pointer the thread-specific record for the current thread:

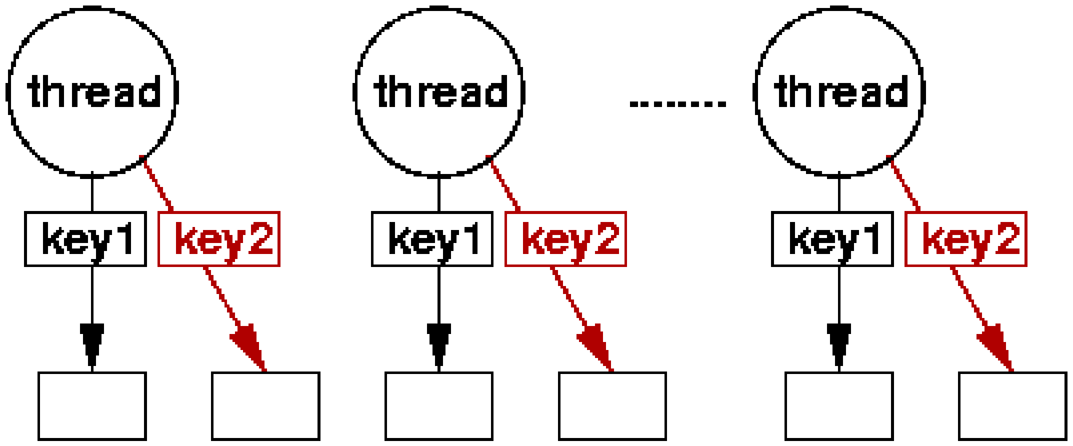
```
perthread_data_ptr mine = pthread_getspecific (my_key);
```

Role of Key in Thread-specific Data



The figure shows how the key is like a tag on a reference from the thread to the user-provided thread-specific object, which is implemented by *pthread_set/getspecific()*.

Use of Multiple Keys in Thread-specific Data



The shows how using different keys allows different libraries to each associate their own thread-specific data with each thread, without interfering with one another.

Other Thread Operations

You can find a full listing of all the pthread operations by using the Unix command "**apropos pthread**". For example, you may get the following output:

```
[baker@websrv processes]$ apropos pthread
pthread_atfork       (3thr)  - register handlers to be called at fork(2) time
pthread_attr_destroy [pthread_attr_init] (3thr)  - thread creation attributes
pthread_attr_getdetachstate [pthread_attr_init] (3thr)  - thread creation attributes
pthread_attr_getinheritsched [pthread_attr_init] (3thr)  - thread creation attributes
pthread_attr_getschedparam [pthread_attr_init] (3thr)  - thread creation attributes
pthread_attr_getschedpolicy [pthread_attr_init] (3thr)  - thread creation attributes
pthread_attr_getscope [pthread_attr_init] (3thr)  - thread creation attributes
pthread_attr_init    (3thr)  - thread creation attributes
pthread_attr_setdetachstate [pthread_attr_init] (3thr)  - thread creation attributes
pthread_attr_setinheritsched [pthread_attr_init] (3thr)  - thread creation attributes
pthread_attr_setschedparam [pthread_attr_init] (3thr)  - thread creation attributes
pthread_attr_setschedpolicy [pthread_attr_init] (3thr)  - thread creation attributes
pthread_attr_setscope [pthread_attr_init] (3thr)  - thread creation attributes
pthread_cancel       (3thr)  - thread cancellation
pthread_cleanup_pop [pthread_cleanup_push] (3thr)  - install and remove cleanup handlers
pthread_cleanup_pop_restore_np [pthread_cleanup_push] (3thr)  - install and remove cleanup ha
```

```
pthread_cleanup_push (3thr)  - install and remove cleanup handlers
pthread_cleanup_push_defer_np [pthread_cleanup_push] (3thr)  - install and remove cleanup han
pthread_cond_broadcast [pthread_cond_init] (3thr)  - operations on conditions
pthread_cond_destroy [pthread_cond_init] (3thr)  - operations on conditions
pthread_cond_init    (3thr)  - operations on conditions
pthread_cond_signal [pthread_cond_init] (3thr)  - operations on conditions
pthread_cond_timedwait [pthread_cond_init] (3thr)  - operations on conditions
pthread_cond_wait [pthread_cond_init] (3thr)  - operations on conditions
pthread_condattr_destroy [pthread_condattr_init] (3thr)  - condition creation attributes
pthread_condattr_init (3thr)  - condition creation attributes
pthread_create       (3thr)  - create a new thread
pthread_detach       (3thr)  - put a running thread in the detached state
pthread_equal        (3thr)  - compare two thread identifiers
pthread_exit         (3thr)  - terminate the calling thread
pthread_getschedparam [pthread_setschedparam] (3thr)  - control thread scheduling parameters
pthread_getspecific [pthread_key_create] (3thr)  - management of thread-specific data
pthread_join         (3thr)  - wait for termination of another thread
pthread_key_create   (3thr)  - management of thread-specific data
pthread_key_delete [pthread_key_create] (3thr)  - management of thread-specific data
pthread_kill [pthread_sigmask] (3thr)  - handling of signals in threads
pthread_kill_other_threads_np (3thr)  - terminate all threads in program except calling threa
pthread_mutex_destroy [pthread_mutex_init] (3thr)  - operations on mutexes
pthread_mutex_init   (3thr)  - operations on mutexes
pthread_mutex_lock [pthread_mutex_init] (3thr)  - operations on mutexes
pthread_mutex_trylock [pthread_mutex_init] (3thr)  - operations on mutexes
pthread_mutex_unlock [pthread_mutex_init] (3thr)  - operations on mutexes
pthread_mutexattr_destroy [pthread_mutexattr_init] (3thr)  - mutex creation attributes
pthread_mutexattr_getkind_np [pthread_mutexattr_init] (3thr)  - mutex creation attributes
pthread_mutexattr_init (3thr)  - mutex creation attributes
pthread_mutexattr_setkind_np [pthread_mutexattr_init] (3thr)  - mutex creation attributes
pthread_once         (3thr)  - once-only initialization
pthread_self         (3thr)  - return identifier of current thread
pthread_setcancelstate [pthread_cancel] (3thr)  - thread cancellation
pthread_setcanceltype [pthread_cancel] (3thr)  - thread cancellation
pthread_setschedparam (3thr)  - control thread scheduling parameters
pthread_setspecific [pthread_key_create] (3thr)  - management of thread-specific data
pthread_sigmask      (3thr)  - handling of signals in threads
```

```
pthread_testcancel [pthread_cancel] (3thr)  - thread cancellation
sigwait [pthread_sigmask] (3thr)  - handling of signals in threads
```

For this course, you do not need to learn to use all of these functions, so they are not explained here. If you want to read more about them you can use the *man* command to read the on-line manual pages about them.

Compiling and Running Programs with Pthreads

To link a program using POSIX threads you may need to specify the POSIX threads library, via the compilation/linkage parameter *-lpthread*.

To link a program using the *sched_yield()* function on Solaris you may need to specify the real-time library, via the compilation/linkage parameter *-lrt*.

To avoid forgetting to include such linkage directives when you compile programs it is helpful to construct a makefile and specify the inclusion of *-lpthread* as part of a default compilation rule.

Review of Thread Pitfalls

- unprotected shared variables
- stack overflow
- waiting on CV without a loop
- forgetting to unlock a mutex
- using a mutex as a CV
- using a CV without the matching mutex
- thread confusion in debugger
- failure to detach or join threads
- mixing signals with threads

When you program with threads, be especially wary of the following:

- Variables that are accessed by more than one thread. If you have no mutual exclusion mechanism in place to protect them you probably have a problem.
- Variables that are accessed by more than one thread and not. If you have no mutual exclusion mechanism in place to protect them you have a problem.

- Thread stack overflow. A process with multiple threads needs multiple execution stacks, one per thread. A bounded area of memory is allocated for each. If this overflows the program will generally crash with a *SIGSEGV* or *SIGBUS*. To reduce the chance of stack overflow, avoid recursion and declarations of large array variables within threads or functions called by threads.
- Calls to *pthread_cond_wait()* that are not in a loop that checks a state variable. Calls outside of such loops are erroneous because of the possibility of spurious wakeups.
- Unmatched *pthread_mutex_lock* and *pthread_mutex_unlock* calls. Failure to unlock a mutex leads to deadlock. To make it easy to match up calls, try to keep your critical sections short. This also will make your code more efficient for concurrent execution.
- Confusion between appropriate uses of a mutex and a condition variable. ***Never*** use a mutex to wait for an event or condition. Use a mutex to protect access to a shared variable. Use a condition variable to wait for an event or condition.
- Calls to *pthread_cond_wait()* outside of critical sections. A thread must be holding a mutex to call *pthread_cond_wait()*. A good way keep this straight is to remember that a condition variable is a shared variable.
- Unmatched calls to *pthread_mutex_lock()* and *pthread_mutex_unlock*. Failure to unlock a mutex leads to deadlock. To make it easy to match up calls, try to keep your critical sections short. This also will make your code more efficient for concurrent execution.
- Problems if you try using a debugger, such as *gdb*. Beware that the interleaving effects of thread execution are likely to confuse you and/or the debugger. (Similar problems arise with signal handlers.)
- Remember to either create your threads as detached, or arrange to collect them after termination with *pthread_join*.
- It is wise to avoid mixing the use of signals with the use of threads in an application. The semantic interactions between the thread operations and the signal operations are subtle. Moreover, the present implementation of signals with threads for Linux does not conform to the Unix/POSIX specification. This is a consequence of the Linux thread implementation treating threads more like processes. In particular, some signal operations that should affect the entire process -- like *sigaction()* and *sigprocmask()* only affect one thread.

Volatile Variables

- "`volatile int X;`" tells compiler not to optimize away load or store operations:
- `X:=1; ... Y:= X; ... X:= 0;` *versus* `... Y:= 1; ... X:= 0;`
- important if variable is accessed by more than one thread

This example deals with a situation that should not arise if you are programming with threads using the mutex and condition variable API for mutual exclusion and synchronization. It illustrates some of the pitfalls that you can expect if you try to "roll your own" solutions to these problems.

Compilers try to "optimize" the code they generate, to make it run faster. Some of these optimizations are not correct if the code contains references to shared variables, which might be accessed concurrently by other threads. For example, suppose the source code of a program contains a loop like the following:

Busy-Wait Loop

```
X = 1; while (X);
```
versus
```
while (1);
```

Observe that this kind of code might be used by a naive programmer to implement a "busy wait" operation, where the current thread is waiting for another thread to change the value of *X* to 0.

Test-compiling this code with no optimizations using the *gcc* compiler with no optimizations we obtain the following Intel x86 assembly language output:

Busy-Wait Loop Code: no optimization

```
        movl    $1, X
        .p2align 2
.L3:
        cmpl    $0, X
        jne     .L3
```

Observe that his code has the desired effect, since the value of *X* is re-fetched every time around the loop.

Test-compiling this code using the *gcc* compiler with optimization *-O2* we obtain the following assembly language output:

Busy-Wait Loop Code: with optimization, no volatile

```
        movl    $1, X
        movl    $1, %eax
        .p2align 2
.L5:
        testl   %eax, %eax
        jne     .L5
```

Observe that in this second version the value of *X* has been loaded into a register *eax*, which cannot change during the loop. Thus, the code does not have the desired effect.

The problem is that the compiler writer reasoned that the variable *X* was "loop invariant". That is, its value would not change during the loop. That would ordinarily be correct in a single-threaded application. It is only wrong because we are accessing the variable *X* from more than one thread.

With the variable *X* declared *volatile* and optimization level `-02` we get the following:

Busy-Wait Loop Code: with optimization & volatile

```
        movl    $1, X
        .p2align 2
.L5:
        movl    X, %eax
        testl   %eax, %eax
        jne     .L5
```

Observe that this code works as originally intended.

When a programmer plans to access a variable from more than one thread it is the programmer's responsibility to tell the compiler about this, using the attribute *volatile*, or by calling synchronization operations like the mutex lock/unlock operations between accesses by different threads. In the latter case, the synchronization operations (besides their main purpose, of providing mutual exclusion between threads) have the side-effect of preventing the compiler from doing optimizations of the kind described here across calls.

For Further Reading

- POSIX Threads Tutorial (Mark Hays)

At the time this was written, if you looked for tutorials on POSIX threads with a Web search engine you would quickly find several good ones, including the two mentioned above.

T. P. Baker (Id)