pthread Tutorial

January 18, 2020

Contents

Legal

1 Introduction

This document is intended to be a short but useful tutorial on how to use POSIX threads (pthreads). In this document I do not attempt to give a full description of all pthread features. Instead I hope to give you enough information to use pthreads in a basic, yet effective way. Please refer to a text on pthreads for the more esoteric details of the standard.

In addition to talking about the pthread interface itself, I also spend time in this document discussing issues regarding concurrent programming in general. While such issues are not specific to pthreads, it is a must that you understand them if you are to use pthreads—or any thread library—effectively.

I will assume that you are compiling pthreads programs on a Unix system. However, you should be aware that the pthreads interface is not necessarily specific to Unix. It is a standard application program interface that could potentially be implemented on many different systems. However, pthreads is the usual way multi-threaded support is offered in the Unix world. Although many systems support their own internal method of handling threads, virtually every Unix system that supports threads at all offers the pthreads interface.

The pthreads API can be implemented either in the kernel of the operating system or in a library. It can either be preemptive or it can be non-preemptive. A portable program based on pthreads should not make any assumptions about these matters.

When you compile a program that uses pthreads, you may have to set special options on the compiler's command line to indicate extra (or different) libraries and/or alternative code generating stratagies. Consult your compiler's documentation for more information on this. Often you can indicate your desire to use pthreads by supplying the "-pthread" option at the end of the compiler command line. For example

```
$ gcc -o myprog myprog.c -pthread
```

This single option specifies that the pthreads library should be linked and also causes the compiler to properly handle the multiple threads in the code that it generates.

2 Creating and Destroying Threads

Clearly the first step required in understanding how to build a multi-threaded program is to understand how to create and destroy threads. There are a number of subtle issues associated with this topic. Normally one wants to not only create a thread but also to send that thread one or more parameters. Furthermore when a thread ends, one normally wants to be able to retrieve one or more values that are returned from the thread. In this section I will describe how these things can be done with pthreads.

2.1 Creating Threads

To create a new thread you need to use the `pthread_create()` function. Listing 1 shows a skeleton program that creates a thread that does nothing and then waits for the thread to terminate.

The `pthread_create()` function gives back a thread identifier that can be used in other calls. The second parameter is a pointer to a *thread attribute object* that

Listing 1: Skeleton Thread Program

```c
#include <pthread.h>

/*
 * The function to be executed by the thread should take a
 * void* parameter and return a void* result.
 */
void *thread_function(void *arg)
{
  // Cast the parameter into whatever type is appropriate.
  int *incoming = (int *)arg;

  // Do whatever is necessary using *incoming as the argument.

  // The thread terminates when this function returns.
  return NULL;
}

int main(void)
{
  pthread_t thread_ID;
  void     *thread_result;
  int       value;

  // Put something meaningful into value.
  value = 42;

  // Create the thread, passing &value for the argument.
  pthread_create(&thread_ID, NULL, thread_function, &value);

  // The main program continues while the thread executes.

  // Wait for the thread to terminate.
  pthread_join(thread_ID, &thread_result);

  // Only the main thread is running now.
  return 0;
}
```

you can use to set the thread's attributes. The null pointer means to use default attributes which is suitable for many cases. The third parameter is a pointer to the function the thread is to execute. The final parameter is the argument passed to the thread function. By using pointers to void here, any sort of data could potentially be passed provided proper casts are applied. In the skeleton example I show how a single integer can be used as a thread argument, but in practice one might send a pointer to a structure containing multiple arguments to the thread.

At some point in your program you should wait for each thread to terminate and collect the result it produced by calling `pthread_join()`. Alternatively you can create a detached thread. The results returned by such threads are thrown away. The problem with detached threads is that, unless you make special arrangements, you are never sure when they complete. Usually you want to make sure all your threads have terminated cleanly before you end the process by returning from `main()`. Returning from `main()` will cause any running threads to be abruptly aborted. While this might be appropriate in some cases, it runs the risk of leaving critical work being done by a thread only partially completed.

If you want to kill a thread before its thread function returns normally, you can use `pthread_cancel()`. However, there are difficulties involved in doing that. You must be sure the thread has released any resources that it has obtained before it actually dies. For example if a thread has dynamically allocated memory and you cancel it before it can free that memory, your program will have a memory leak. This is different than when you kill an entire process. The operating system will typically clean up (certain) resources that are left dangling by the process. In particular, the entire address space of a process is recovered. However, the operating system will not do that for a thread since all the threads in a process share resources. For all the operating system knows, the memory allocated by one thread will be used by another thread. This situation makes it difficult to canceling threads properly.

Exercises

1. Write a program that creates 10 threads. Have each thread execute the same function and pass each thread a unique number. Each thread should print "Hello, World (thread n)" five times where *n* is replaced by the thread's number. Use an array of `pthread_t` objects to hold the various thread IDs. Be sure the program doesn't terminate until all the threads are complete. Try running your program on more than one machine. Are there any differences in how it behaves?

2.2 Returning Results from Threads

The example in the last section illustrated how you can pass an argument into your thread function if necessary. In this section I will describe how to return results from thread functions.

Note that the thread functions are declared to return a pointer to `void`. However, there are some pitfalls involved in using that pointer. The code below shows one attempt at returning an integer status code from a thread function.

```
void *thread_function(void *)
{
  int code = DEFAULT_VALUE;

  // Set the value of 'code' as appropriate.

  return (void *)code;
}
```

This method will only work on machines where integers can be converted to a pointer and then back to an integer without loss of information. On some machines such conversions are dangerous. In fact this method will fail in all cases where one attempts to return an object, such as a structure, that is larger than a pointer.

In contrast, the code below doesn't fight the type system. It returns a pointer to an internal buffer where the return value is stored. While the example shows an array of characters for the buffer, one can easily imagine it being an array of any necessary type, or a single object such as an integer status code or a structure with many members.

```
void *thread_function(void *)
{
  char buffer[64];

  // Fill up the buffer with something good.

  return buffer;
}
```

Alas, the code above fails because the internal buffer is automatic and it vanishes as soon as the thread function returns. The pointer given back to the calling thread points at undefined memory. This is another example of the classic dangling pointer error.

In the next attempt the buffer is made static so that it will continue to exist even after the thread function terminates. This gets around the dangling pointer problem.

```
void *thread_function(void *)
{
  static char buffer[64];

  // Fill up the buffer with something good.

  return buffer;
}
```

This method might be satisfactory in some cases, but it doesn't work in the common case of multiple threads running the same thread function. In such a situation the second thread will overwrite the static buffer with its own data and destroy the data left by the first thread. Global data suffers from this same problem since global data always has static duration.

The version below is the most general and most robust.

```
void *thread_function(void *)
{
  char *buffer = (char *)malloc(64);

  // Fill up the buffer with something good.

  return buffer;
}
```

This version allocates buffer space dynamically. This approach will work correctly even if multiple threads execute the thread function. Each will allocate a different array and store the address of that array in a stack variable. Every thread has its own stack so automatic data objects are different for each thread.

In order to receive the return value of a thread the higher level thread must join with the subordinate thread. This is shown in the `main` function of Listing 1. In particular

```
void *thread_result;

// Wait for the thread to terminate.
pthread_join(thread_ID, &thread_result);
```

The `pthread_join()` function blocks until the thread specified by its first argument terminates. It then stores into the pointer pointed at by its second argument the value returned by the thread function. To use this pointer, the higher level thread must cast it into an appropriate type and dereference it. For example

```
char *message;

message = (char *)thread_result;
printf("I got %s back from the thread.\n", message);
free(thread_result);
```

If the thread function allocated the space for the return value dynamically then it is essential for the higher level thread to free that space when it no longer needs the return value. If this isn't done the program will leak memory.

Exercises

1. Write a program that computes the square roots of the integers from 0 to 99 in a separate thread and returns an array of doubles containing the results. In the meantime the main thread should display a short message to the user and then display the results of the computation when they are ready.

2. Imagine that the computations done by the program above were much more time consuming than merely calculating a few square roots. Imagine also that displaying the "short message" was also fairly time consuming. For example, perhaps the message needed to be fetched from a network server as HTML and then rendered. Would you expect the multi-threaded program to perform better than a single threaded program that, for example, did the calculations first and then fetched the message? Explain.

3 Thread Synchronization

In order to effectively work together the threads in a program usually need to share information or coordinate their activity. Many ways to do this have been devised and such techniques usually go under the name of *thread synchronization*. In this section I will outline several common methods of thread synchronization and show how they can be done using POSIX threads.

3.1 Mutual Exclusion

When writing multi-threaded programs it is frequently necessary to enforce mutually exclusive access to a shared data object. This is done with mutex objects. The idea is to associate a mutex with each shared data object and then require every thread that wishes to use the shared data object to first lock the mutex before doing so. Here are the particulars:

1. Declare an object of type `pthread_mutex_t`.

2. Initialize the object by calling `pthread_mutex_init()` or by using the special static initializer `PTHREAD_MUTEX_INITIALIZER`.

3. Call `pthread_mutex_lock()` to gain exclusive access to the shared data object.

4. Call `pthread_mutex_unlock()` to release the exclusive access and allow another thread to use the shared data object.

5. Get rid of the object by calling `pthread_mutex_destroy()`.

The program of Listing 2 demonstrates the basic approach. It is important to understand that if a thread attempts to lock the mutex while some other thread has it locked, the second thread is blocked until the first releases the mutex with `pthread_mutex_unlock()`.

The code above uses dynamic initialization. However, it is also possible to initialize a mutex object statically using the special symbol `PTHREAD_MUTEX_INITIALIZER` as the initializer.

Be sure to observe these points:

1. No thread should attempt to lock or unlock a mutex that has not been initialized.

2. The thread that locks a mutex *must* be the thread that unlocks it.

3. No thread should have the mutex locked when you destroy the mutex.

In practice it is sometimes the case that threads are blocked on mutex objects when the program wishes to terminate. In such a situation it might make sense to `pthread_cancel()` those threads before destroying the mutex objects they are blocked on. Coordinating this properly can be tricky, however.

Notice that it is possible to assign special "mutex attributes" to a mutex object when it is created. This is done by creating a mutex attribute object, assigning attributes to the object, and then passing a pointer to the attribute object into `pthread_mutex_init()`. The program in Listing 2 just calls for default attributes by providing a `NULL` pointer instead. In many cases this is perfectly adequate. The use of mutex attribute objects is beyond the scope of this document.

It is important to understand that mutex locking is *advisory*. This means that no part of the system (the operating system, the runtime system, or any other system) *requires* that you follow the rules. If you forget to lock a mutex before accessing the protected shared data, your thread might interfere with the activity of another thread... even a thread that has played by the rules and locked the mutex first.

Exercises

1. Enter the program in Listing 2 and try it out. Does it behave the way you expected? Try different values for the maximum loop index in the thread

Listing 2: Mutex Example

```c
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t lock;
int shared_data;
  // Usually shared data is more complex than just an int.

void *thread_function(void *arg)
{
  int i;

  for (i = 0; i < 1024*1024; ++i) {
    // Access the shared data here.
    pthread_mutex_lock(&lock);
    shared_data++;
    pthread_mutex_unlock(&lock);
  }
  return NULL;
}

int main(void)
{
  pthread_t thread_ID;
  void     *thread_result;
  int       i;

  // Initialize the mutex before trying to use it.
  pthread_mutex_init(&lock, NULL);

  pthread_create(&thread_ID, NULL, thread_function, NULL);

  // Try to use the shared data.
  for (i = 0; i < 10; ++i) {
    sleep(1);
    pthread_mutex_lock(&lock);
    printf("\rShared integer's value = %d\n", shared_data);
    pthread_mutex_unlock(&lock);
  }
  printf("\n");

  pthread_join(thread_ID, &thread_result);

  // Clean up the mutex when we are finished with it.
  pthread_mutex_destroy(&lock);
  return 0;
}
```

function and different sleep times in the main function. Try removing the call to `sleep()` entirely. Try the program on different machines. Can you explain what is happening?

2. Suppose you are building a C++ string class that you intend to use in a multi-threaded program. You are worried about your string objects possibly getting corrupted if they are updated by more than one thread at a time. You consider adding a mutex as a member of each string and locking that mutex whenever any string method is called. Discuss the implications of this design. Be careful: this question is considerably trickier than it may appear!

3.2 Barriers

A barrier is a synchronization primitive that forces multiple threads to reach the same point in the program before any are allowed to continue. If a thread arrives at the barrier early it will be suspended until the appropriate number of other threads arrive. Once the last thread has reached the barrier, all the waiting threads are released and the threads can proceed concurrently once again.

Barriers tend to be useful when multiple threads are executing different iterations of the same loop. Often it is necessary to be sure all loop iterations are complete before moving on to a task that requires the previous work to be entirely finished.

Listing 3 shows a skeleton that illustrates how barriers can be used and the context in which they might be useful. In this listing all threads execute the same thread function but process different iterations of the `for` loop.

Two barriers are used. The first causes all threads to synchronize after the `for` loop executes so that the preparation for the next cycle can be done knowing that the previous cycle has fully completed. The barrier wait function returns the special value `PTHREAD_BARRIER_SERIAL_THREAD` in exactly one thread (chosen arbitrarily). That thread is thus “elected” to take care of any serial work needed between cycles. In this program there is no attempt to prepare for the next cycle in parallel.

The other threads immediately wait on the next barrier for the serial thread to catch up. Once the preparation is fully completed, they are all released from the second barrier where they loop back to do the next cycle of work.

This style of programming occurs frequently when writing truly parallel programs that expect the threads to be physically executing at the same time. Such programs often use multiple threads to complete different parts of the same task and use a barrier to be sure the complete task is finished before allowing the higher level program logic to continue.

Listing 3: Barrier Example

```
#include <pthread.h>

void *thread_function(void *arg)
{
  int i;
  int done = 0;
  struct TaskArg *thread_arg = (struct TaskArg *)arg;

  while (!done) {
    for (i = thread_arg->start; i < thread_arg->end; ++i) {
      // Work on loop iteration i
      // Each thread gets a separate TaskArg.
      // Different threads do different iterations.
    }

    if (pthread_barrier_wait(&loop_barrier) ==
          PTHREAD_BARRIER_SERIAL_THREAD) {

      // Prepare for next cycle.
      if (nothing_more) done = 1;
    }
    pthread_barrier_wait(&prep_barrier);
  }
  return NULL;
}
```

Barriers are instances of the `pthread_barrier_t` type. They are initialized with `pthread_barrier_init` and cleaned up with `pthread_barrier_destroy`. Listing 4 shows a main function that uses the parallel looping function of Listing 3.

The barrier objects themselves are global so they can be accessed by all threads. Alternatively they could have been made local to the main function and passed into each thread indirectly (i. e., with a pointer) by way of the thread's arguments. In any case, it is necessary for all threads to see the same barrier object.

When the barrier objects are initialized it is necessary to provide the number of threads that can accumulate on the barrier. For example, a barrier initialized with a count of five will cause threads to wait until the fifth thread arrives. This count can't be changed once the barrier object has been initialized.

This listing creates as many threads as there are processors using a previously defined variable `processor_count` (declaration not shown). Since the number of processors is unknown when the program is written space for the thread IDs is allocated dynamically.

Exercises

I need something here!

3.3 Condition Variables

If you want one thread to signal an event to another thread, you need to use *condition variables*. The idea is that one thread waits until a certain condition is true. First it tests the condition and, if it is not yet true, calls `pthread_cond_wait()` to block until it is. At some later time another thread makes the condition true and calls `pthread_cond_signal()` to unblock the first thread.

Every call to `pthread_cond_wait()` should be done as part of a conditional statement. If you aren't doing that, then you are most likely using condition variables incorrectly. For example

```
if (flag == 0) pthread_cond_wait(...);
```

Here I'm waiting until the flag is not zero. You can test conditions of any complexity. For example

```
x = f(a, b);
if (x < 0 || x > 9) pthread_cond_wait(...);
```

Here I'm waiting until `x` is in the range from zero to nine inclusive where `x` is computed in some complex way. Note that `pthread_cond_wait()` is only called if the

Listing 4: Barrier Example Main

```c
pthread_barrier_t loop_barrier;
pthread_barrier_t prep_barrier;

int main(void)
{
  int i;
  pthread_t *thread_IDs;

  pthread_barrier_init(&loop_barrier, NULL, processor_count);
  pthread_barrier_init(&prep_barrier, NULL, processor_count);
  thread_IDs =
    (pthread_t *)malloc(processor_count * sizeof(pthread_t));

  // Create a thread for each CPU.
  for (i = 0; i < processor_count; ++i) {
    struct TaskArg *task =
      (struct TaskArg *)malloc(sizeof(struct TaskArg));
    task->start = i * iterations_per_processor;
    if (i == processor_count - 1)
      task->end = ITERATION_COUNT;
    else
      task->end = (i + 1) * iterations_per_processor;
    pthread_create(&thread_IDs[i],NULL,thread_function,task);
  }

  // Wait for threads to end.
  for (i = 0; i < processor_count; ++i) {
    pthread_join(thread_IDs[i], NULL);
  }

  free(thread_IDs);
  pthread_barrier_destroy(&loop_barrier);
  pthread_barrier_destroy(&prep_barrier);
  return 0;
}
```

condition is not yet true. If the condition is already true, **pthread_cond_wait()** is not called. This is necessary because condition variables do not remember that they have been signaled.

If you look at my examples, you will see that there is a serious race condition in them. Suppose the condition is not true. Then suppose that after the condition is tested but before **pthread_cond_wait()** is called, the condition becomes true. The fact that the condition is signaled (by some other thread) will be missed by **pthread_cond_wait()**. The first thread will end up waiting on a condition that is already true. If the condition is never signaled again the thread will be stuck waiting forever.

To deal with this problem, every time you use a condition variable you must also use a mutex to prevent the race condition. For example:

```
pthread_mutex_lock(&mutex);
x = f(a, b);
if (x < 0 || x > 9) pthread_cond_wait(&condition, &mutex);
pthread_mutex_unlock(&mutex);
```

The thread that signals this condition will use the same mutex to gain exclusive access to the whatever values are involved in computing the condition (which depends on what function `f` does in this example). Thus there is no way that the signaling could occur between the test of the condition and the waiting on the condition.

For the above to work, **pthread_cond_wait()** needs to wait on the condition and unlock the mutex as an atomic action. It does this, but it needs to know which mutex to unlock. Hence the need for the second parameter of **pthread_cond_wait()**. When the condition is signaled, **pthread_cond_wait()** will lock the mutex again before returning so that the **pthread_mutex_unlock()** in the above example is appropriate regardless of which branch of the if is taken.

Here is how the signaling thread might look

```
pthread_mutex_lock(&mutex);
a = ...
b = ...
x = f(a, b);
if (x >= 0 && x <= 9) pthread_cond_signal(&condition);
pthread_mutex_unlock(&mutex);
```

Before doing a computation that might change the condition, the signaling thread locks the mutex to make sure the waiting thread can't get caught in a race condition. For example, in this case it wouldn't do if the *waiting* thread saw a new version of `a` but the old version of `b`. In that case it might calculate an inappropriate value of `f(a, b)` and wait when it shouldn't.

There is a further subtlety regarding the use of condition variables. In certain situations the wait function might return even though the condition variable has not actually been signaled. For example, if the Unix process in general receives an operating system signal, the thread blocked in `pthread_cond_wait()` might be elected to process the signal handling function. If system calls are not restarting (the default in many cases) the `pthread_cond_wait()` call might return with an interrupted system call error code[1]. This has nothing to do with the state of the condition so proceeding as if the condition is true would be inappropriate.

The solution to this problem is to simply retest the condition after `pthread_cond_wait()` returns. This is most easily done using a loop. For example

```
pthread_mutex_lock(&mutex);
while (1) {
  x = f(a, b);
  if (x < 0 || x > 9) pthread_cond_wait(&condition, &mutex);
    else break;
}
pthread_mutex_unlock(&mutex);
```

Of course this assumes you want to ignore any spurious returns from the wait function. In a more complex application you might want to process the error codes in various ways depending on the situation.

The `pthread_cond_signal` function releases only one thread at a time. In some cases it is desirable to release all threads waiting on a condition. This can be accomplished using `pthread_cond_broadcast`. For example

```
pthread_mutex_lock(&mutex);
a = ...
b = ...
x = f(a, b);
if (x >= 0 && x <= 9) pthread_cond_broadcast(&condition);
pthread_mutex_unlock(&mutex);
```

The example in Listing 5 illustrates the use of condition variables in the context of a program. Although contrived, this example is at least complete and compilable.

Notice that in this program the condition variables are also initialized and destroyed by calls to appropriate functions. As with mutex variables you can also initialize condition variables statically using a special symbol: `PTHREAD_COND_INITIALIZER`.

Exercises

1. Modify the program in Listing 5 to print messages and add delays (or wait for user input) at various places so you can verify that the thread is, in

[1] Of course this assumes you are dealing with an actual kernel thread. If the thread is purely a user mode thread such unexpected returns won't occur.

Listing 5: Condition Variable Example

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

pthread_cond_t  is_zero;
pthread_mutex_t mutex;   // Condition variables needs a mutex.
int shared_data = 32767; // Or some other large number.

void *thread_function(void *arg)
{
  // Imagine doing something useful.
  while (shared_data > 0) {
    // The other thread sees the shared data consistently.
    pthread_mutex_lock(&mutex);
    --shared_data;
    pthread_mutex_unlock(&mutex);
  }

  // Signal the condition.
  pthread_cond_signal(&is_zero);
  return NULL;
}

int main(void)
{
  pthread_t thread_ID;
  void     *exit_status;
  int       i;

  pthread_cond_init(&is_zero, NULL);
  pthread_mutex_init(&mutex, NULL);

  pthread_create(&thread_ID, NULL, thread_function, NULL);

  // Wait for the shared data to reach zero.
  pthread_mutex_lock(&mutex);
  while (shared_data != 0)
    pthread_cond_wait(&is_zero, &mutex);
  pthread_mutex_unlock(&mutex);

  pthread_join(thread_ID, &exit_status);

  pthread_mutex_destroy(&mutex);
  pthread_cond_destroy(&is_zero);
  return 0;
}
```

fact, waiting for the condition as appropriate. Verify that the thread does not wait if the condition is already true when it is first tested.

2. In the text above, when a condition is signaled the signaling thread calls `pthread_cond_signal()` before unlocking the mutex. However, it is also possible swap those operations as shown below.

   ```
   pthread_mutex_lock(&mutex);
   a = ...
   b = ...
   x = f(a, b);
   pthread_mutex_unlock(&mutex);
   if (x >= 0 && x <= 9) pthread_cond_signal(&condition);
   ```

   Does this result in the same behavior as before? Are any race conditions introduced (or fixed) by this change? How does this approach impact application performance?

3.4 Semaphores

Semaphore are essentially glorified integer counters. They support two primary operations. One operation, called *down* or *wait*, attempts to decrement the counter. The other operation, called *up*, *signal*, or *post* attempts to increment the counter. What makes semaphores special is that if a thread tries to wait on a zero semaphore it is blocked instead. Later when another thread posts the semaphore the blocked thread is activated while the semaphore remains at zero. In effect, the posting causes the semaphore to be incremented but then the thread that was blocked trying to do a wait is allowed to proceed, causing the semaphore to be immediately decremented again.

If multiple threads are blocked waiting on a semaphore then the system chooses one to unblock. Exactly how this choice is made is generally system dependent. You can not assume that it will be in FIFO order[2]. However, the order in which the threads are unblocked is not normally a concern. If it is, then your program may not be very well designed.

A semaphore with an initial value of one can be used like a mutex. When a thread wishes to enter its critical section and access a shared data structure, it does a wait operation on the semaphore. If no other thread is in its critical section, the semaphore will have its initial value of one and the wait will return immediately. The semaphore will then be zero. If another thread tries to wait on the semaphore during this time it will be blocked. When the first thread is finished executing its critical section it does a post operation on the semaphore. This will unblock one waiting thread or, if there are no waiting

[2] If threads have different priorities, normally the highest priority thread is allowed to go first.

threads, increment the semaphore back to its initial value of one. A semaphore used in this way is called a *binary semaphore* because it has exactly two states.

However, because semaphores are integers they can take on values larger than one. Thus they are often used to count scarce resources. For example a thread might wait on a semaphore to effective reserve one item of a resource. If there are no items left, the semaphore will be zero and the wait operation will block. When a thread is finished using an item of a resource it posts the semaphore to either increment the count of available items or to allow a blocked thread to access the now available item. A semaphore used in this way is called a *counting semaphore*.

The POSIX semaphore API is not really part of the normal pthread API. Instead POSIX standardizes semaphores under a different API. Traditional Unix systems support shared memory, message queues, and semaphores as part of what is called "System V Interprocess Communication" (System V IPC). POSIX also provides shared memory, message queues, and semaphores as a package that competes with, or replaces, the older standard. The functionality of the two systems is similar although the details of the two APIs are different.

Note that POSIX semaphores, like System V IPC semaphores, can be used to synchronize two or more separate processes. This is different than pthread mutexes. A mutex can only be used by threads in the same process. Because POSIX semaphores can be used for interprocess communication, they have the option of being named. One process can create a semaphore under a particular name and other processes can open that semaphore by name. In this tutorial, however, I will focus only on synchronizing threads in the same process.

The skeleton program in Listing 6 shows how to initialize, clean up, and use a POSIX semaphore. For brevity the skeleton program does not show the threads being created or joined nor does it show any error handling. See the manual pages for the various functions for more information on error returns.

Another difference between a pthread mutex and a semaphore is that, unlike a mutex, a semaphore can be posted in a different thread than the thread that does the wait operation. This is necessary when using a semaphore to count instances of a scarce resource. The skeleton program in Listing 6 is using a semaphore like a mutex. I did this to simplify the listing so that the functions used to manipulate the semaphore would be clear.

To see semaphores being used in a more interesting way, consider the classic producer/consumer problem. In this problem one thread is producing items (say, objects of type `void *` that might be pointing at other objects of arbitrary complexity) while another thread is consuming those items. Listing 7 shows an abstract data type that implements a buffer that can be used to hold items as they pass from one thread to another.

The solution actually needs two semaphores. One is used to count the number of free slots in the buffer and the other is used to count the number of used slots.

Listing 6: Semaphore Example

```
#include <semaphore.h>

int    shared;
sem_t  binary_sem;  // Used like a mutex.

void *thread_function(void *arg)
{
  sem_wait(&binary_sem);     // Decrements count.
  // Used shared resource.
  sem_post(&binary_sem);     // Increments count.
}

void main(void)
{
  sem_init(&binary_sem, 0, 1); // Initial count of 1.

  // Start threads here.

  sem_wait(&binary_sem);
  // Use shared resource.
  sem_post(&binary_sem);

  // Join with threads here.

  sem_destroy(&binary_sem);
  return 0;
}
```

Listing 7: Producer/Consumer Abstract Type

```c
#ifndef PCBUFFER_H
#define PCBUFFER_H

#include <pthread.h>
#include <semaphore.h>

#define PCBUFFER_SIZE 8

typedef struct {
  void *buffer[PCBUFFER_SIZE];
  pthread_mutex_t lock;
  sem_t           used;
  sem_t           free;
  int             next_in;   // Next available slot.
  int             next_out;  // Oldest used slot.
} pcbuffer_t;

void  pcbuffer_init(pcbuffer_t *);
void  pcbuffer_destroy(pcbuffer_t *);
void  pcbuffer_push(pcbuffer_t *, void *);
void *pcbuffer_pop(pcbuffer_t *);

#endif
```

This is necessary because we must block the producer when the buffer is full and we must block the consumer when the buffer is empty. However, semaphores only block their caller when one attempts to decremented them below zero; they never block when they are incremented. Listing 8 shows the implementation in detail.

The initialization and clean-up functions are straight forward. In contrast the push and pop functions are surprisingly subtle. In each it is necessary to first reserve a unit of the limited resource. In the case of `pcbuffer_push()` we must reserve a free slot. If no free slots are available, the call to `sem_wait(&p->free)` will block until the consumer posts that semaphore after removing an item.

Once a slot has been reserved we lock the buffer to ensure that no other thread can corrupt it by modifying it at the same time. Finally, after unlocking we post the other semaphore to perhaps unblock the other thread as necessary. For example the call to `sem_signal(&p->used)` will unblock a waiting consumer to handle the item just stored in the buffer.

Exercises

1. Using POSIX mutex and condition variables, implement a semaphore abstract type. For example, consider a header file containing the following.

   ```
   typedef struct {
     // Fill in members as appropriate.
   } semaphore_t;

   void semaphore_init(
     struct semaphore *s, int initial_count);
   void semaphore_destroy(semaphore_t *s);
   void semaphore_up(semaphore_t *s);
   void semaphore_down(semaphore_t *s);
   ```

   Implement the functions declared above. This shows that semaphores are not strictly necessary as part of a low level API.

2. Some semaphore APIs (such as the Win32 API) allow the post operation to advance the value of a semaphore by more than one. This can be implemented by executing a basic post operation multiple times in a loop. However, such an approach is inefficient if the number to be added to the semaphore is large. Extend your solution for the question above so that `semaphore_up` takes an additional integer parameter specifying the how much the semaphore value is to be advanced. Try to use an efficient method of handling large advances. Make sure your solution works properly and does not suffer from any race conditions even when there are multiple threads waiting on the semaphore.

Listing 8: Producer/Consumer Implementation

```
#include "pcbuffer.h"

void pcbuffer_init(pcbuffer_t *p) {
  pthread_mutex_init(&p->lock);
  sem_init(&p->used, 0, 0);
  sem_init(&p->free, 0, PCBUFFER_SIZE);
  p->next_in  = 0;
  p->next_out = 0;
}

void pcbuffer_destroy(pcbuffer_t *p) {
  pthread_mutex_destroy(&p->lock);
  sem_destroy(&p->used);
  sem_destroy(&p->free);
}

void pcbuffer_push(pcbuffer_t *p, void *value) {
  sem_wait(&p->free);
  pthread_mutex_lock(&p->lock);
  p->buffer[p->next_in++] = value;
  if (next_in == PCBUFFER_SIZE) next_in = 0;
  pthread_mutex_unlock(&p->lock);
  sem_post(&p->used);
}

void *pcbuffer_pop(pcbuffer_t *p) {
  void *return_value;

  sem_wait(&p->used);
  pthread_mutex_lock(&p->lock);
  return_value = p->buffer[p->next_out++];
  if (next_out == PCBUFFER_SIZE) next_out = 0;
  pthread_mutex_unlock(&p->lock);
  sem_post(&p->free);
  return return_value;
}
```

3.5 Reader/Writer Locks

Mutex objects provide mutually exclusive access to a shared resource. But sometimes complete mutual exclusion is unnecessarily restrictive. If two threads are only interested in reading a shared resource, it should be possible to allow both to access the resource at the same time. If neither thread tries to modify the resource, the resource will never be in an inconsistent state and simultaneous access is safe. Indeed, it is common for there to be multiple threads trying to read a shared resource where updates to that resource are uncommon. For example a tree data structure might be used many times by multiple threads to look up information and yet updated only rarely by a single thread.

To support this usage POSIX provides *reader/writer locks.* Multiple readers can lock such an object without blocking each other, but when a single writer acquires the lock it has exclusive access to the resource. All following readers or writers will block as long as a writer holds the lock.

The skeleton program in Listing 9 shows the basic structure. By now the pattern of initialization, destruction, and use should look familiar. In a more typical program the thread function where the read lock is acquired might be executed by many threads while the main function where the write lock is needed might be executed by only one thread. Notice in this case that the same function is used to unlock both read locks and write locks.

Depending on the implementation, a steady stream of readers might permanently lock out a writer. This situation is called *writer starvation.* On the other hand if the implementation favors writers in the sense of letting waiting writers obtain the lock as soon as possible, *reader starvation* may occur. The POSIX standard favors writers, depending on specific thread priorities. This behavior is reasonable because writers are presumed to be rare and the updates they want to do are presumed to be important. It is more realistic to expect a steady stream of readers than a steady stream of writers. Thus if readers were favored, writer starvation would be a significant concern.

It is important to note that the system does not (can not) enforce the read-only restriction on reader threads. There is nothing to stop a thread from acquiring a read lock and then go ahead and write to the shared resource anyway. If multiple reader threads do this, data corruption might occur. It is the programmer's responsibility to ensure this does not happen.

Exercises

1. Implement a reader/writer lock abstract type in terms of other POSIX synchronization primitives. Can your implementation cause reader or writer starvation?

Listing 9: Reader/Writer Lock Example

```
#include <pthread.h>

int              shared;
pthread_rwlock_t lock;

void *thread_function(void *arg)
{
  pthread_rwlock_rdlock(&lock);
  // Read from the shared resource.
  pthread_rwlock_unlock(&lock);
}

void main(void)
{
  pthread_rwlock_init(&lock, NULL);

  // Start threads here.

  pthread_rwlock_wrlock(&lock);
  // Write to the shared resource.
  pthread_rwlock_unlock(&lock);

  // Join with threads here.

  pthread_rwlock_destroy(&lock);
  return 0;
}
```

3.6 Monitors

The synchronization primitives discussed so far are all fairly primitive. This makes them flexible but it also makes them difficult to use properly. A synchronization abstraction that is commonly provided by programming languages with direct support for concurrency is the *monitor*. Essentially a monitor is an encapsulation of data and code with the property that only one thread at a time can be inside the monitor. Thus mutual exclusion is provided automatically by the monitor construct.

The POSIX thread API does not support monitors directly but because it is a useful facility there is value in exploring how a monitor could be implemented with the POSIX thread API. Since C is a relatively primitive language, the syntactic features for declaring and using monitors are not directly available. Instead the programmer must adhere to certain programming conventions. This is typical of programming in the C environment.

Listing 10 illustrates the general approach. The monitor maps to a single translation unit (source file) where the internal data and support functions have internal linkage (marked as `static`). A master mutex object is used to control access to the monitor. The monitor operations are ordinary external functions with the property that they lock the mutex on entry and unlock it on exit. Care must be taken by the programmer to ensure that the mutex is properly unlocked on every exit path from those functions. In addition, the external functions can't call each other without risking deadlock on the monitor mutex[3].

Associated with the monitor is a header file that declares any externally visible service functions and the types they require. Multiple threads can call these service functions safely. Only one thread at a time is allowed inside the monitor so the internal data will never experience simultaneous updates.

In the case where some service functions only read the internal data, it may make sense to use a POSIX reader/writer lock as the monitor lock. Functions that only read the data can obtain a read lock on the monitor, allowing several threads to call such functions simultaneously. Of course functions that update the internal data will need to obtain a write lock on the monitor.

Unfortunately mutual exclusion inside the monitor is not enough to make the monitor construct generally useful. In many situations it is necessary for a thread to suspend itself inside the monitor while waiting for a certain condition to be true (for example, for data to be ready, etc). Naturally the monitor must be unlocked when the thread is suspended so that another thread is allowed inside the monitor. If this were not done, the suspended thread would wait forever for a condition that could never arise.

POSIX condition variables are a good fit for these semantics. The suspending thread can call `pthread_cond_wait` on an internal condition variable, passing

[3] Unless the monitor mutex is a recursive mutex.

Listing 10: Basic Monitor

```
#include "service.h"

static pthread_mutex_t monitor_lock =
  PTHREAD_MUTEX_INITIALIZER;

static int internal_data;

static void internal_function( void ) { ... }

void service_function( void )
{
    pthread_mutex_lock(&monitor_lock);
    ...
    // Use internal data and functions freely.
    ...
    pthread_mutex_unlock(&monitor_lock);
}
```

the monitor lock as the second parameter. This will suspend the thread and unlock the monitor in an atomic manner. When another thread signals the condition, the first thread will attempt to reacquire the mutex before continuing. It will be prevented from doing this until the signaling thread leaves the monitor (since the signaling thread will still hold the monitor mutex). Thus the rule that only a single thread executes in the monitor at a time is enforced. Listing 11 illustrates the approach.

In this example the functions `data_ready` and `make_data_ready` are assumed to be internal monitor functions (not shown in Listing 11). Notice that it is important to test the condition in a loop when calling the wait operation. This is to protect against spurious wake-ups (for example due to operating system signals). It also protects against another problem: When the signaling thread leaves the monitor, some other thread waiting to enter the monitor might acquire the mutex before the awakened thread does so. This other thread might then invalidate the condition before the awakened thread is able to return from `pthread_cond_wait`. Thus retesting the condition before continuing from the wait is a must.

Exercises

1. POSIX conditional waits take a pointer to a mutex object. However, if one wants to use a reader/writer lock to control access to the monitor, `pthread_cond_wait` can't be used directly. How can this be handled?

Listing 11: Monitor Example

```
#include "service.h"

static pthread_mutex_t monitor_lock =
  PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t condition =
  PTHREAD_COND_INITIALIZER;

void service_function1( void )
{
    pthread_mutex_lock(&monitor_lock);
    ...
    while (!data_ready( )) {
      pthread_cond_wait(&condition, &monitor_lock);
    }
    ...
    pthread_mutex_unlock(&monitor_lock);
}

void service_function2( void )
{
    pthread_mutex_lock(&monitor_lock);
    ...
    make_data_ready( );
    pthread_cond_signal(&condition);
    ...
    pthread_mutex_unlock(&monitor_lock);
}
```

4 Thread Models

In this section I will describe some ways that threads can be used in real programs. The goal is to give you a feeling for the kind of design ideas that lend themselves to a threaded solution. It is usually possible to build a single threaded program that solves the same problem, but in some cases the single threaded solution is awkward and difficult to manage. Be aware, however, that single threaded solutions are often the most appropriate. There can be a significant amount of overhead associated with synchronizing threads; multiple threads, poorly applied, can easily result in a slower and more confusing program.

4.1 Boss/Worker Model

The idea in the boss/worker model is to have a single *boss* thread that creates work and several *worker* threads that process the work. Typically the boss thread creates a certain number of workers right away—even before any work has arrived. The worker threads form a *thread pool* and are all programmed to block immediately. When the boss thread generates some work, it arranges to have one worker thread unblock to handle it. Should all workers be busy the boss thread might:

1. Queue up the work to be handled later as soon as a worker is free.
2. Create more worker threads.
3. Block until a worker is free to take the new work.

If no work has arrived recently and there are an excessive number of worker threads in the thread pool, the boss thread might terminate a few of them to recover resources. In any case, since creating and terminate threads is relatively expensive (compared to, say, blocking on a mutex) it is generally better to avoid creating a thread for each unit of work produced.

You have already seen this model in action many times. Consider a bank. When you arrive you have work that needs doing. You get in a queue and wait for a free teller (worker thread). When a teller is available that teller handles your work while other tellers are handling other work at the same time. Should someone in line have an unusually complicated transaction, it won't hold up the line. Only one teller will be tied up dealing with the large work item. The other tellers will be available to handle other people's work normally. Thus the response time is reasonable even when some work items are very time consuming.

A web server is another excellent example of where the boss/worker model can be used. The boss thread listens to the network for incoming connections. When a connection is made, the boss thread directs a worker thread to handle that connection. The boss thread then returns to listening to the network again. In this way many connections can be handled at once. If a particularly time consuming connection is active, it won't prevent the server for dealing with other connections as well.

This model works the best when the work items are independent of each other. If the work items depend on each other or have to be processed in a particular sequence the worker threads have to talk to each other and the overall efficiency of this model is much reduced. Also, if you run a boss/worker program on a single processor machine it is important that servicing a work item involves a fair amount of blocking. If the work items are all 100% CPU bound then there won't be any performance enhancement. A single thread servicing all the items in sequence would be just as fast as having multiple threads servicing several items at once. However, if servicing an item requires a lot of blocking, or if multiple CPUs are available, then another thread can use the CPU while the first is blocked and the overall performance is better (often drastically so).

4.2 Pipeline Model

Many programs take some input, transform it in several stages, and then output the result. Instead of having a single thread perform each step in sequence you could have a separate thread handling each stage. The result is much like an assembly line. The data flows from one worker to another and each worker performs their particular operation on the data. By the time the data reaches the end of the line it has been fully transformed into the desired output.

Usually writing a single threaded program to process sequential data in stages is fairly straightforward. However, a multithreaded pipeline has the potential to outperform the single threaded solution. In general, if there are N stages to the pipeline there can be N items being operated on at once by the multithreaded program and the result will be N times faster. In practice it rarely works out this well. To obtain its full efficiency the time required for every stage must be identical and the processing of one stage can't in any way slow down the processing of the others. If the program runs on a single processor the operations being done in each stage must block frequently so that the CPU can execute another stage while the blocked stages are waiting (for example, for I/O).

To balance the load between the stages, the programmer might need to use profiling tools to find out the relative time used by the different stages. Stages that are very short can be combined with the stage on either side while stages that are very long can be split into multiple stages (ideally with blocking operations divided evenly between the stages). Getting this balance just right is difficult yet without it the multithreaded solution to the pipeline model will hardly be any faster than the single threaded solution. In fact, because of locking overhead, it may even be slower[4].

4.3 Background Task Model

Many programs have tasks that they would like to complete "in the background." For example a program might want to backup its data files every 10 minutes or update a special status window every 5 seconds. It is awkward to program such things with only a single thread. The program must remember to check the time regularly and call the background function whenever an appropriate amount of time has elapsed. Since that might happen at any point in the program's execution, the program's logic must be littered with calls to functions that are largely unrelated to the main flow of the program.

With multiple threads, however, this model is quite easy to program. A background thread can be created when the program initializes itself. That thread can sleep for a certain fixed time and then, when it wakes up, perform the necessary background operation. The thread would then just loop back and sleep again until the next time interval has expired. This can happen independently of the main program's logic. The main complication involved with programming this model is in making sure that the background thread is properly synchronized with the main thread when they access shared data.

In this approach the threads are used in a manner similar to the way interrupt service routines are used. They provide background services that the main program does not have to explicity invoke. Many useful tasks can be effectively handled in this way.

4.4 Interface/Implementation Model

Most graphical environments are event driven. Each action taken by the user is a separate event that the program must handle. Examples of events include mouse clicks, menu selections, keystrokes, and so forth. Typically the program contains a single function that is called by the system whenever an event happens. That function must process the event and then return before the system calls the function with the next event. If the event handling function does not return quickly enough events will back up and the user will perceive the program as unresponsive and sluggish. In an extreme case the program will even appear to be dead.

To avoid this, the program can use multiple threads. If handling an event is going to be time consuming and difficult (and involve a lot of blocking), the

[4] The buffers between the stages must be careful to avoid race conditions and overflow/underflow conditions. This involves a significant amount of locking activity.

event handling function can just create a thread to deal with it and then return at once. This gives the event handling function the opportunity to handle additional events while past events are being processed by other threads. The result is that the program's interface remains responsive even if some of the operations requested are very time consuming.

It is not necessary for an entire program to be organized this way. Internal modules in a program can use the same trick. When a function is called in such a module, the function might create a thread to carry out the requested operation and then return at once. The calling program will see the function as very quick and responsive even though the actual work requested hasn't really been completed when the function returns.

The difficulty with this model is that eventually the user of an interface will need to know for sure when certain operations requested in the past have actually completed. Some way of coordinating that information must be provided. Also it is difficult for the program to report errors effectively with this model because an error might occur long after the operation was requested and apparently handled.

Many operating systems themselves use this model extensively. For example, when a program writes to a file, the file is typically not put onto the disk at once. Instead it is just put into a cache (faster) and written to disk latter when the system is less busy. In effect, the operating system writes to disk in a separate thread that is independent of the thread that actually requested the write operation in the first place.

4.5 General Comments

In general, multithreaded programs work best if the threads are as independent as possible. The less the threads have to communicate with each other the better. Whenever threads have to synchronize or share data there will be locking overhead and time spent waiting for other threads. Time blocked while waiting for another thread is time wasted. Such a thread is not doing anything useful. In contrast, if a thread is blocked waiting for I/O it is doing something that the program needs done. Such blocking is good because it allows other threads to get the CPU. But if a thread waits for another thread then it is not accomplishing anything that the program needs. The more threads interact with each other the more time they will spend waiting for each other and the more inefficient the program will be.

It is easy to understand this idea when you think about working with another person on a common project. If you and your partner can do two largely independent activities you can both work without getting in each other's way and you can get twice as much work done. But if you try to work too closely then one of you will simply be waiting for the other and the work won't get done any more quickly than it would by a single person alone. Consider what would happen if you decided to type a paper with your partner but that you and your partner had to alternate keystrokes on the keyboard. To type "Hello" first you type 'H' then your partner types 'e' then you type 'l' and so on. Obviously this would be very inefficient. You would spend more time getting the alternation right than you would actually typing keys. The exact same issues arise in multithreaded programs. An improperly written multithreaded program can be slower—sometimes a lot slower—than its single threaded equivalent.

If two tasks are very independent, they can often be handled by two entirely separate processes. Large software systems are often composed of many executable files, each taking care of a single aspect of the system. At this level the system is "multithreaded" even if the individual programs are not. However, it can be difficult for multiple processes to share information effectively. Putting multiple threads into a single process makes the parallelism more *fine grained*, allows the threads to interact more closely, and share more resources. This can be a good thing. But if taken to extreme it causes inefficiencies as I described above. A good multithreaded program will strike the right balance between sharing and independence. That balance is often difficult to find.

5 Thread Safety

Typically when a complicated object is operated on, it goes through several intermediate, invalid states before the operation is complete. As an analogy consider what a surgeon does when he operates on a person. Although the purpose of the operation is to increase the patient's health, the surgeon performs several steps that would greatly decrease the patient's health if they were left incomplete! Similarly a function that operates on an object will often temporary put that object into an unusable state while it performs the update. Once the update is complete, the function (if written correctly) will leave the object is a fully functional state again.

Should another thread try to use an object while it is in an unusable state (often called an *inconsistent state*) the object will not respond properly and the result will be undefined. Keeping this from happening is the essential problem of thread safety. The problem doesn't come up in a single threaded program because there is no possibility of another thread trying to access the object while the first thread is updating it[5].

[5] Unless exceptions are a possibility. In that case the updating thread might abort the update and then later try to access the incompletely updated object. This causes the same sort of problems to occur.

5.1 Levels of Thread Safety

People often have problems discussing thread safety because there are many different levels of safety one might want to talk about. Just saying that a piece of code is "thread safe" doesn't really say all that much. Yet most people have certain natural expectations about thread safety. Sometimes those expectations are reasonable and valid, but sometimes they are not. Here are some of those expectations.

- Reading an object's value with multiple threads is not normally expected to be a problem. Problems only occur when an object is updated since it is only then that it has to be modified and run the risk of entering inconsistent states.

  However some objects have internal state that gets modified even when its value is read (think about an object that has an internal cache). If two threads try to read such an object there might be problems unless the read operations on that object have been designed to handle the multiple threads properly.

- Updating two independent objects, even of the same type, is not normally expected to be a problem. It is usually assumed that objects that appear to be independent are, in fact, independent and thus the inconsistent states of one such object have no impact on the other.

  However some objects share information behind the scenes (static class data, global data, etc) that causes them to be linked internally even when they do not appear to be linked from a logical point of view. In that case, modifying two "independent" objects might cause a problem anyway. Consider:

  ```
  void f()                      void g()
  {                             {
    std::string x;                std::string y;

    // Modify x.                  // Modify y.
  }                             }
  ```

  If one thread is in function `f()` modifying the string `x` and another is in function `g()` modifying string `y`, will there be a problem? Most of the time you can assume that the two apparently independent objects can be simultaneously modified without synchronization. But it is possible, depending on how `std::string` is implemented, that the two objects share some data internally and that simultaneous modifications *will* cause a problem. In fact, even if one of the functions merely reads the value of the string, there might be a problem if they share internal data that is being updated by the other function.

- Functions that acquire resources, even if from a common pool of resources, are not normally expected to be a problem. Consider:

  ```
  void f()                          void g()
  {                                 {
    char *p = new char[512];          char *p = new char[512];

    // Use the array p.               // Use the array p.
  }                                 }
  ```

  If one thread is in function `f()` and another thread is in function `g()`, both threads might try to allocate memory simultaneously by invoking the `new` operator. In a multi-threaded environment, it is safe to assume that `new` has been written to work correctly in this case even though both invocations of `new` are trying to take memory from the same pool of memory. Internally `new` will synchronize the threads so that each call will return a unique allocation and the internal memory pool will not be corrupted. Similar comments can be made about functions that open files, make network connections, and perform other resource allocation tasks.

  However if the resource allocation functions are not designed with threads in mind then they may well fail if invoked by multiple threads at once.

What people typically expect to cause problems is when a program tries to access (update or read) an object while another thread is updating that same object. Global objects are particularly prone to this problem. Local objects are much less so. For example:

```
std::string x;

void f()
{
  std::string y;

  // Modify x and y.
}
```

If two threads enter function `f()` at the same time, they will get different versions of the string `y`. This is because every thread has its own stack and local objects are allocated on the thread's stack. Thus every thread has its own, independent copy of the local objects. As a result, manipulating `y` inside `f()` will not cause a problem (assuming that manipulating independent objects is safe). However, since there is only one copy of the global `x` that both threads will be touching, there could be a problem caused by those operations.

Local objects are not immune to problems since any function can start a new thread and pass a pointer to a local object as a parameter to that thread. For example

```
void f()
{
  std::string x;

  start_thread(some_function, &x);
  start_thread(some_function, &x);
}
```

Here I assume there is a library function named `start_thread()` that accepts a pointer to a thread function (defined elsewhere) and a pointer to a parameter for that function. In this case I start two threads executing `some_function()`, giving both of them a pointer to the string `x`. If `some_function()` tries to modify that string then two threads will be modifying the same object and problems are likely. Note that this case is particularly insidious because `some_function()` has no particular reason to expect that it will be given the same parameter twice. Thus it is unlikely to have any protection to handle such a case.

5.2 Writing Thread Safe Code

In theory the only way to control the actions of a thread is to use synchronization primitives such as mutexes or semaphores. In languages that provide threads as a part of the language, synchronization primitives of some kind are normally provided by the language itself. In other cases, such as with C, they are library functions, such as the POSIX API described in this tutorial, that interact with the operating system.

Normally you should write code that meets the usual expectations that people have about thread safe code. If you are implementing a C++ class, make sure that multiple simultaneous reads on an object are safe. If you do update internal data behind the caller's back, you will probably have to protect those updates yourself. Also make sure the simultaneous writes to independent objects are safe. If you do make use of shared data, you will probably have to protect updates to that shared data yourself. If you write a function that manages shared resources for multiple threads from a common pool of such resources, you will probably have to protect the resource pool from corruption by multiple, simultaneous requests. However, in general, you probably don't have to bother protecting every single object against simultaneous updates. Let the calling program worry about that case. Such total safety is usually very expensive in terms of runtime efficiency and not normally necessary or even appropriate.

5.3 Exception Safety vs Thread Safety

Both thread and exception safety share a number of common issues. Both are concerned with objects that are in an inconsistent state. Both have to think about resources (although in different ways... exception safety is concerned with resource leaks, thread safety is concerned with resource corruption). Both have several levels of safety that could be defined along with some common expectations about what is and is not safe.

However, there is one important difference between the exception safety and thread safety. Exceptions occur synchronously with the program's execution while threads are asynchronous. In other words, exceptions occur, in theory, only at certain well defined times. Although it is not always clear which operations might throw an exception and which might not, in theory it is possible to precisely define exactly when an exception might happen and when it can't happen. As a result it is often possible to make a function exception safe by just reorganizing it. In contrast there is no way to control when two threads might clash. Reorganizing a function is rarely helpful when it comes to making it thread safe. This difference makes thread related errors difficult to reproduce and difficult to manage.

6 Rules for Multithreaded Programming

In this section I'll try to summarize a few "rules of thumb" that one should keep in mind when building a multithreaded program. Although using multiple threads can provide elegant and natural solutions to some programming problems, they can also introduce race conditions and other subtle, difficult to debug problems. Many of these problems can be avoided by following a few simple rules.

6.1 Shared Data

As I described in Section 3, when two threads try to access the same data object there is a potential for problems. Normally modifying an object requires several steps. While those steps are being carried out the object is typically not in a well formed state. If another thread tries to access the object during that time, it will likely get corrupt information. The entire program might have undefined behavior afterwards. This must be avoided.

6.1.1 What data is shared?

1. Static duration data (data that lasts as long as the program does). This includes global data and static local data. The case of static local data is only significant if two (or more) threads execute the function containing the static local at the same time.

2. Dynamically allocated data that has had its address put into a static variable. For example, if a function uses malloc() or new to allocate an object and then places the address of that object into a variable that is accessible by more than one thread, the dynamically allocated object will then be accessible by more than one thread.

3. The data members of a class object that has two (or more) of its member functions called by different threads at the same time.

6.1.2 What data is not shared?

1. Local variables. Even if two threads call the same function they will have different copies of the local variables in that function. This is because the local variables are kept on the stack and every thread has its own stack.

2. Function parameters. In languages like C the parameters to functions are also put on the stack and thus every thread will have its own copy of those as well.

Since local variables and function parameters can't be shared they are immune to race conditions. Thus you should use local variables and function parameters whenever possible. Avoid using global data. Be aware, however, that *taking the address of a local variable and passing that address to place where another thread can read it amounts to sharing the local variable with that other thread.*

6.1.3 What type of simultaneous access causes a problem?

1. Whenever one thread tries to update an object, no other threads should be allowed to touch the object (for either reading or writing). Mutual exclusion should be enforced with some sort of mutex-like object (or by some other suitable means).

6.1.4 What type of simultaneous access is safe?

1. If multiple threads only read the value of an object, there should be no problem. Be aware, however, that complicated objects often update internal information even when, from the outside, they are only being read. Some objects maintain a cache or keep track of usage statistics internally even for reads. Simultaneous reads on such an object might not be safe.

2. If one thread writes to one object while another thread touches a totally independent object, there should be no problem. Be aware, however, that many functions and objects do share some data internally. What appears to be two separate objects might really be using a shared data structure behind the scenes.

3. Certain types of objects are updated in an uninterruptable way. Thus simultaneous reads and writes to such objects are safe because it is impossible for the object to be in an inconsistent state during the update. Such updates are said to be *atomic*. The bad news is that the types that support atomic updates are usually very simple (for example: int) and there is no good way to know for sure exactly which types they are. The C standard provides the type `sig_atomic_t` for this purpose. It is defined in `<signal.h>` and is a kind of integer. Simultaneous updates to an object declared to be `volatile sig_atomic_t` are safe. Mutexes are not necessary in this case.

6.2 What can I count on?

Unless a function is specifically documented as being thread-safe, you should assume that it is not. Many libraries make extensive use of static data internally and unless those libraries were designed with multiple threads in mind that static data is probably not being properly protected with mutexes.

Similarly you should regard the member functions of a class as unsafe for multiple threads unless it has been specifically documented to be otherwise. It is easy to see that there might be problems if two threads try to manipulate the same object. However, even if two threads try to manipulate different objects there could still be problems. For various reasons, many classes use internal static data or try to share implementation details among objects that appear to be distinct from the outside.

You can count on the following:

1. The API functions of the operating system are thread-safe.
2. The POSIX thread standard requires that most functions in the C standard library be thread-safe. There are a few exceptions which are documented as part of the standard.
3. Under Windows the C standard library is totally thread safe provided you use the correct version of the library and you initialize it properly (if required).
4. C++ 1998 does not discuss threads so The thread safety of the C++ standard library is vague and dependent on the compiler you are using. This has been corrected with the C++ 2011 (or later) standard which supports threads as part of the standard.

If you use a non thread-safe function in one of your functions, your function will be rendered non thread-safe as well. However, you are free to use a non thread-safe function in a multithreaded program provided it is never called by two or more threads at the same time. You can either arrange to use such functions in only one thread or protect calls to such functions with mutexes. Keep in mind that many functions share data internally with other functions. If you try to protect calls to a non thread-safe function with a mutex you must also protect calls to all the other related functions with the same mutex. This is often difficult.

7 Conclusion

Using multiple threads has the potential to improve your program's performance. Even on a single processor system, performance might be increased because one thread can execute while another is blocked waiting on I/O (for example). Also, used appropriately, multiple threads can allow for some elegant designs that can clarify and simplify the architecture of your system.

The pthreads API is a relatively low level API that offers the usual features of such APIs: it is flexible and powerful, but also verbose and hard to use properly. Higher level thread APIs exist, often built on top of the pthreads API, that are easier to use correctly but sacrifice some of the flexibility and generality of pthreads.

This document was prepared to support my classes at Vermont Technical College. However, I offer it to the public for general consumption on the hope that others might find it useful. Please don't hesitate to send me corrections or comments or suggestions for improvements or new sections. I can be reached at pchapin@vtc.edu.

Peter Chapin