Multithreaded Programming (POSIX pthreads Tutorial)

Table of Contents:

1. Introduction
2. What is a Thread?
3. Thread Design Patterns
4. Protecting Shared Resources
5. Thread Synchronization Primitives
6. POSIX pthreads
7. Performance Considerations
8. Other Approaches
9. Resources

If the document URL does not begin with https://randu.org/tutorials/threads/ then you are viewing a copy.

Introduction

Code is often written in a *serialized* (or sequential) fashion. What is meant by the term serialized? Ignoring instruction level parallelism (ILP), code is executed sequentially, one after the next in a monolithic fashion, without regard to possibly more available processors the program could exploit. Often, there are potential parts of a program where performance can be improved through the use of threads.

With increasing popularity of machines with symmetric multiprocessing (largely due in part to the rise of multicore processors), programming with threads is a valuable skill set worth learning.

Why is it that most programs are sequential? One guess would be that students are not taught how to program in a parallel fashion until later or in a difficult-to-follow manner. To make matters worse, multithreading non-trivial code is difficult. Careful analysis of the problem, and then a good design is not an option for multithreaded programming; it is an absolute must.

We will dive into the world of threads with a little bit of background first. We will examine thread synchronization primitives and then a tutorial on how to use POSIX pthreads will be presented.

What is a Thread?

Analogy

Isn't that something you put through an eye of a sewing needle?

> Yes.

How does it relate to programming then?

> Think of sewing needles as the processors and the threads in a program as the thread fiber. If you had two needles but only one thread, it would take longer to finish the job (as one needle is idle) than if you split the thread into two and used both needles at the same time.

Taking this analogy a little further, if one needle had to sew on a button (blocking I/O), the other needle could continue doing other useful work even if the other needle took 1 hour to sew on a single button. If you only used one needle, you would be ~1 hour behind!

Definition

In order to define a thread formally, we must first understand the boundaries of where a thread operates.

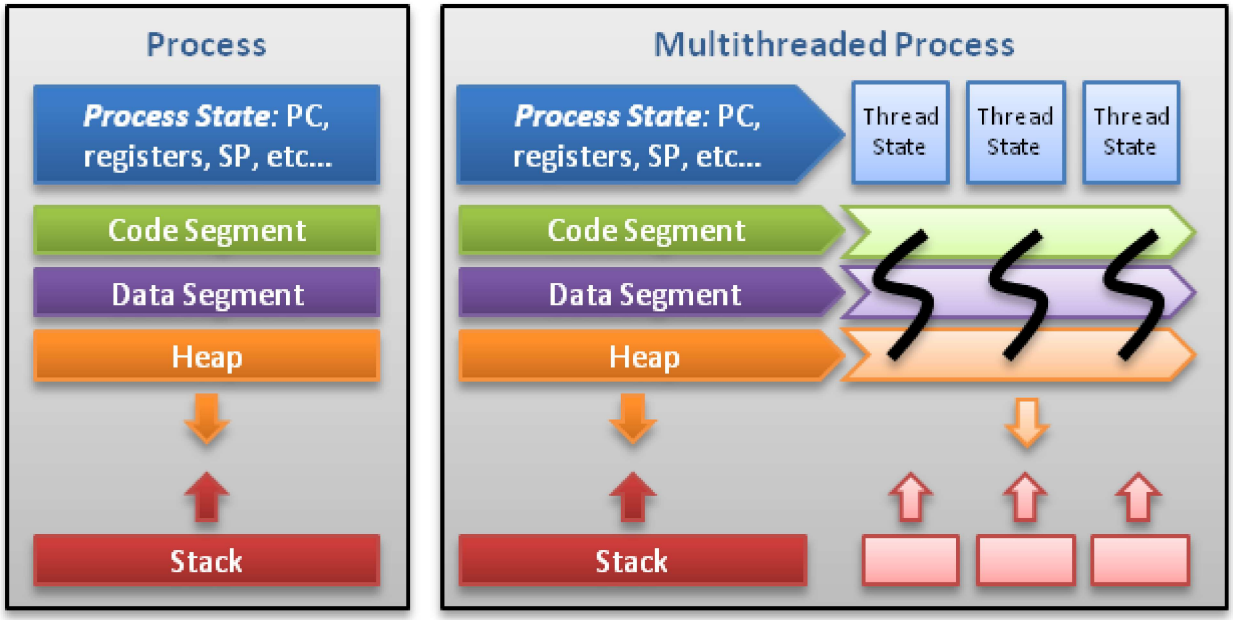
A computer program becomes a **process** when it is loaded from some store into the computer's memory and begins execution. A process can be executed by a processor or a set of processors. A process description in memory contains vital information such as the program counter which keeps track of the current position in the program (i.e. which instruction is currently being executed), registers, variable stores, file handles, signals, and so forth.

A **thread** is a sequence of such instructions within a program that can be executed independently of other code.

The figure to the right conceptually shows that threads are within the *same process address space*, thus, much of the information present in the memory description of the process can be shared across threads.

Some information cannot be replicated, such as the stack (stack pointer to a different memory area per thread), registers and thread-specific data. This information sufficies to allow threads to be scheduled independently of the program's main thread and possibly one or more other threads within the program.

Explicit operating system support is required to run multithreaded programs. Fortunately, most modern operating systems support threads such as Linux (via NPTL), BSD variants, Mac OS X, Windows, Solaris, AIX, HP-UX, etc. Operating



Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

© Alfred Park, http://randu.org/tutorials/threads

systems may use different mechanisms to implement multithreading support.

Terminology

Before we can dive in depth into threading concepts, we need to get familiarized with a few terms related to threads, parallelism and concurrency.

- **Lightweight Process** (LWP) can be thought of as a virtual CPU where the number of LWPs is usually greater than the number of CPUs in the system. Thread libraries communicate with LWPs to schedule threads. LWPs are also sometimes referred to as *kernel threads*.
- **X-to-Y** model. The mapping between LWPs and Threads. Depending upon the operating system implementation and/or user-level thread library in use, this can vary from 1:1, X:1, or X:Y. Linux, some BSD kernels, and some Windows versions use the 1:1 model. User-level threading libraries are commonly in the X:1 class as the underlying kernel does not have any knowledge of the user-level threads. The X:Y model is used in Windows 7.
- **Contention Scope** is how threads compete for system resources (i.e. scheduling).
- **Bound threads** have system-wide contention scope, in other words, these threads contend with other processes on the entire system.
- **Unbound threads** have process contention scope.
- **Thread-safe** means that the program protects shared data, possibly through the use of mutual exclusion.
- **Reentrant** code means that a program can have more than one thread executing concurrently.
- **Async-safe** means that a function is reentrant while handling a signal (i.e. can be called from a signal handler).
- **Concurrency vs. Parallelism -** They are not the same! Parallelism implies simultaneous running of code (which is not possible, in the strict sense, on uniprocessor machines) while concurrency implies that many tasks can run in any order and possibly in parallel.

Amdahl's Law and the Pareto Principle

Threads can provide benefits... *for the right applications!* Don't waste your time multithreading a portion of code or an entire program that isn't worth multithreading.

Gene Amdahl argued the theoretical maximum improvement that is possible for a computer program that is parallelized, under the premise that the program is strongly scaled (i.e. the program operates on a fixed problem size). His claim is a well known assertion known as Amdahl's Law. Essentially, Amdahl's law states that the speedup of a program due to parallelization can be no larger than the inverse of the portion of the program that is immutably sequential. For example, if 50% of your program is not parallelizable, then you can only expect a maximum speedup of 2x, regardless the number of processors you throw at the problem. Of course many problems and data sets that parallel programs process are not of fixed size or the serial portion can be very close to zero. What is important to the reader here, is to understand that most interesting problems that are solved by computer programs tend to have some limitations in the amount of parallelism that can be effectively expressed (or introduced by the very mechanism to parallelize) and exploited as threads or some other parallel construct.

It must be underscored how important it is to understand the problem the computer program is trying to solve first, before simply jumping in head first. Careful planning and consideration of not only what the program must attack in a parallel fashion and the means to do so by way of the algorithms employed and the vehicle for which they are delivered must be performed.

There is a common saying: "90% of processor cycles are spent in 10% of the code." This is more formally known as the Pareto Principle. Carefully analyze your code or your design plan; don't spend all of your time optimizing/parallelizing the 90% of the code that doesn't matter much! Code profiling and analysis is outside of the scope of this document, but it is recommended reading left to those unfamiliar with the subject.

Thread Design Patterns

There are different ways to use threads within a program. Here, three common thread design patterns are presented. There is no hard and fast rule on which is the best. It depends on what the program is intended to tackle and in what context. It is up to you to decide which best pattern or patterns fit your needs.

Thread Pool (Boss/Worker)

One thread dispatches other threads to do useful work which are usually part of a *worker thread pool*. This thread pool is usually pre-allocated before the boss (or master) begins dispatching threads to work. Although threads are lightweight, they still incur overhead when they are created.

Peer (Workcrew)

The peer model is similar to the boss/worker model except once the worker pool has been created, the boss becomes the another thread in the thread pool, and is thus, a peer to the other threads.

Pipeline

Similar to how pipelining works in a processor, each thread is part of a long chain in a processing factory. Each thread works on data processed by the previous thread and hands it off to the next thread. You must be careful to equally distribute work and take extra steps to ensure non-blocking behavior in this thread model or you could experience pipeline "stalls."

Protecting Shared Resources

Threads may operate on disparate data, but often threads may have to touch the same data. It is unsafe to allow concurrent access to such data or resources without some *mechanism that defines a protocol for safe access*! Threads must be explicitly instructed to block when other threads may be potentially accessing the same resources.

Mutual Exclusion

Mutual exclusion is the method of *serializing access* to shared resources. You do not want a thread to be modifying a variable that is already in the process of being modified by another thread! Another scenario is a dirty read where the value is in the process of being updated and another thread reads an old value.

Mutual exclusion allows the programmer to create a defined protocol for serializing access to shared data or resources. Logically, a **mutex** is a lock that one can virtually attach to some resource. If a thread wishes to modify or read a value from a shared resource, the thread must first gain the lock. Once it has the lock it may do what it wants with the shared resource without concerns of other threads accessing the shared resource because other threads will have to wait. Once the thread finishes using the shared resource, it unlocks the mutex, which allows other threads to access the resource. This is a protocol that serializes access to the shared resource. Note that such a protocol must be enforced for the data or resource a mutex is protecting across all threads that may touch the resource being protected. If the protocol is violated (e.g., a thread modifies a shared resource without first requesting a mutex lock), then the protocol defined by the programmer has failed. There is nothing preventing a thread programmer, whether unintentionally (most often the case, i.e., a bug -- see race conditions below) or intentionally from implementing a flawed serialization protocol.

As an analogy, you can think of a mutex as a safe with only one key (for a standard mutex case), and the resource it is protecting lies within the safe. Only one person can have the key to the chest at any time, therefore, is the only person allowed to look or modify the contents of the chest at the time it holds the key.

The code between the lock and unlock calls to the mutex, is referred to as a **critical section**. Minimizing time spent in the critical section allows for greater concurrency because it potentially reduces the amount of time other threads must wait to gain the lock. Therefore, it is important for a thread programmer to minimize critical sections if possible.

Mutex Types

There are different types of locks other than the standard simple blocking kind.

- **Recursive**: allows a thread holding the lock to acquire the same lock again which may be necessary for recursive algorithms.
- **Queuing**: allows for *fairness* in lock acquisition by providing FIFO ordering to the arrival of lock requests. Such mutexes may be slower due to increased overhead and the possibility of having to wake threads next in line that may be sleeping.
- **Reader/Writer**: allows for multiple readers to acquire the lock simultaneously. If existing readers have the lock, a writer request on the lock will block until all readers have given up the lock. This can lead to writer starvation.
- **Scoped**: RAII-style semantics regarding lock acquisition and unlocking.

Depending upon the thread library or interface being used, only a subset of the additional types of locks may be available. POSIX pthreads allows recursive and reader/writer style locks.

Potential Traps with Mutexes

An important problem associated with mutexes is the possibility of **deadlock**. A program can deadlock if two (or more) threads have stopped execution or are spinning permanently. For example, a simple deadlock situation: thread 1 locks lock A, thread 2 locks lock B, thread 1 wants lock B and thread 2 wants lock A. Instant deadlock. You can prevent this from happening by making sure threads acquire locks in an agreed order (i.e. preservation of **lock ordering**). Deadlock can also happen if threads do not unlock mutexes properly.

A **race condition** is when non-deterministic behavior results from threads accessing shared data or resources without following a defined synchronization protocol for serializing such access. This can result in erroneous outcomes that cause failure or inconsistent behavior making race conditions particularly difficult to debug. In addition to incorrectly synchronized access to shared resources, library calls outside of your program's control are common culprits. Make sure you take steps within your program to enforce serial access to shared file descriptors and other external resources. Most man pages will contain information about thread safety of a particular function, and if it is not thread-safe, if any alternatives exist (e.g., `gethostbyname()` and `gethostbyname_r()`).

Another problem with mutexes is that contention for a mutex can lead to **priority inversion**. A higher priority thread can wait behind a lower priority thread if the lower priority thread holds a lock for which the higher priority thread is waiting. This can be eliminated/reduced by limiting the number of shared mutexes between different priority threads. A famous case of priority inversion occurred on the Mars Pathfinder.

Atomic Operations

Atomic operations allow for concurrent algorithms and access to certain shared data types without the use of mutexes. For example, if there is sufficient compiler and system support, one can modify some variable (e.g., a 64-bit integer) within a multithreaded context without having to go through a locking protocol. Many atomic calls are non-portable and specific to the compiler and system. Intel Threading Building Blocks (see below), contains semi-portable atomic support under C++. The C++1x and C1x standards will also include atomic operations support. For gcc-specific atomic support, please see this and this.

Lock-free algorithms can provide highly concurrent and scalable operations. However, lock-free algorithms may be more complex than their lock-based counterparts, potentially incurring additional overhead that may induce negative cache effects and other problems. Careful analysis and performance testing is required for the problem under consideration.

Thread Synchronization Primitives

As we have just discussed, mutexes are one way of synchronizing access to shared resources. There are other mechanisms available for not only coordinating access to resources but synchronizing threads.

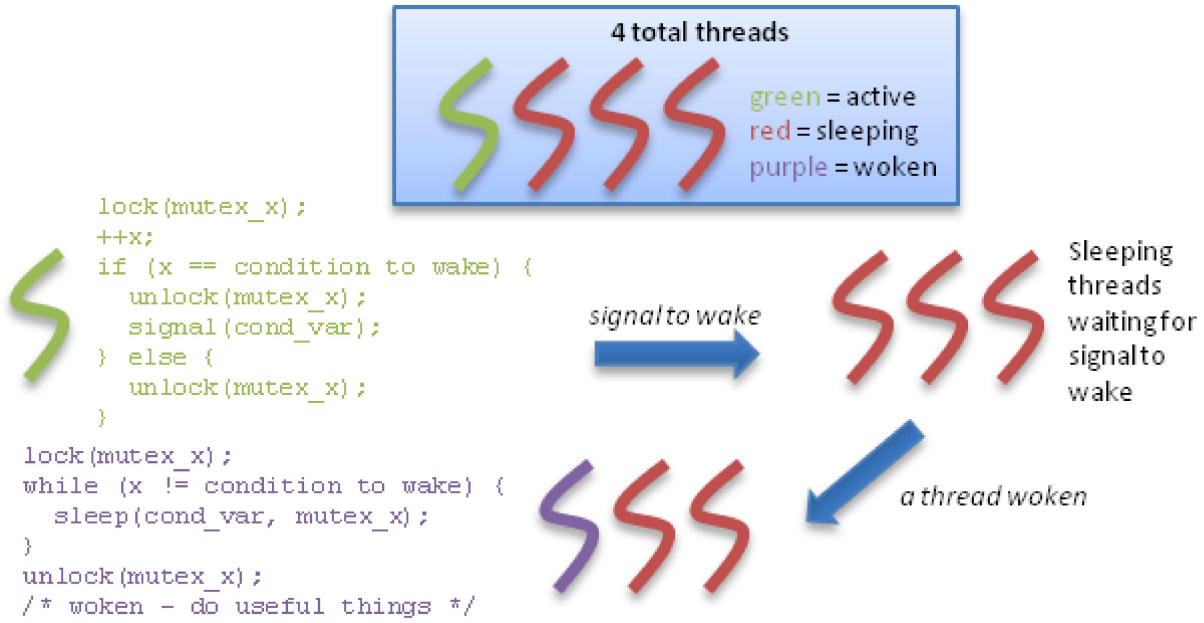
Join

A thread join is a protocol to allow the programmer to *collect* all relevant threads at a logical synchronization point. For example, in fork-join parallelism, threads are spawned to tackle parallel tasks and then join back up to the main thread after completing their respective tasks (thus performing an implicit barrier at the join point). Note that a thread that executes a join has terminated execution of their respective thread function.

Condition Variables

Condition variables allow threads to synchronize to a value of a shared resource. Typically, condition variables are used as a notification system between threads.

For example, you could have a counter that once reaching a certain count, you would like for a thread to activate. The thread (or threads) that activates once the counter reaches the limit would *wait* on the condition variable. Active threads *signal* on this condition variable to notify other threads waiting/sleeping on this condition variable; thus causing a waiting thread to wake. You can also use a *broadcast* mechanism if you want to signal *all* threads waiting on the condition variable to wakeup. Conceptually, this is modeled by the figure on the right with pseudocode.



© Alfred Park, http://randu.org/tutorials/threads

When waiting on condition variables, the wait should be inside a loop, not in a simple if

statement because of **spurious wakeups**. You are not guaranteed that if a thread wakes up, it is the result of a signal or a broadcast call.

Barriers

Barriers are a method to synchronize a set of threads at some point in time by having all participating threads in the barrier wait until all threads have called the said barrier function. This, in essence, blocks all threads participating in the barrier until the slowest participating thread reaches the barrier call.

Spinlocks

Spinlocks are locks which *spin* on mutexes. Spinning refers to continuously polling until a condition has been met. In the case of spinlocks, if a thread cannot obtain the mutex, it will keep polling the lock until it is free. The advantage of a spinlock is that the thread is kept active and does not enter a sleep-wait for a mutex to become available, thus can perform better in certain cases than typical blocking-sleep-wait style mutexes. Mutexes which are heavily contended are poor candidates for spinlocks.

Spinlocks should be avoided in uniprocessor contexts. Why is this?

Semaphores

Semaphores are another type of synchronization primitive that come in two flavors: binary and counting. Binary semaphores act much like simple mutexes, while counting semaphores can behave as *recursive mutexes*. Counting semaphores can be initialized to any arbitrary value which should depend on how many resources you have available for that particular shared data. Many threads can obtain the lock simultaneously until the limit is reached. This is referred to as *lock depth*.

Semaphores are more common in multiprocess programming (i.e. it's usually used as a synch primitive between processes).

POSIX pthreads

Now that we have a good foundation of thread concepts, lets talk about a particular threading implementation, POSIX pthreads. The pthread library can be found on almost any modern POSIX-compliant OS (and even under Windows, see pthreads-win32).

Note that it is not possible to cover more than an introduction on pthreads within the context of this short overview and tutorial. pthreads concepts such as thread scheduling classes, thread-specific data, thread canceling, handling signals and reader/writer locks are not covered here. Please see the Resources section for more information.

If you are programming in C++, I highly recommend evaluating the Boost C++ Libraries. One of the libraries is the Thread library which provides a common interface for portable multithreading.

It is assumed that you have a good understanding of the C programming language. If you do not or need to brush up, please review basic C (especially pointers and arrays). Here are some resources.

Preliminaries

Before we begin, there are a few required steps you need to take before starting any pthreads coding:

1. Add `#include <pthread.h>` to your source file(s).
2. If you are using gcc, you can simply specify `-pthread` which will set all proper defines and link-time libraries. On other compilers, you may have to define `_REENTRANT` and link against `-lpthread`.
3. *Optional:* some compilers may require defining `_POSIX_PTHREAD_SEMANTICS` for certain function calls like `sigwait()`.

Creating pthreads

A pthread is represented by the type `pthread_t`. To create a thread, the following function is available:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

Let's digest the arguments required for `pthread_create()`:

1. `pthread_t *thread`: the actual thread object that contains pthread id
2. `pthread_attr_t *attr`: attributes to apply to this thread
3. `void *(*start_routine)(void *)`: the function this thread executes
4. `void *arg`: arguments to pass to thread function above

Before we dive into an example, let's first look at two other important thread functions:

```
void pthread_exit(void *value_ptr);
int pthread_join(pthread_t thread, void **value_ptr);

/* ignore me: needed so underscores above do not get clipped off */
```

`pthread_exit()` terminates the thread and provides the pointer `*value_ptr` available to any `pthread_join()` call.

`pthread_join()` suspends the calling thread to wait for successful termination of the thread specified as the first argument `pthread_t thread` with an optional `*value_ptr` data passed from the terminating thread's call to `pthread_exit()`.

Let's look at an example program exercising the above pthread functions:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 2

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
```

```
  int tid;
  double stuff;
} thread_data_t;

/* thread function */
void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);

  pthread_exit(NULL);
}

int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS];
  int i, rc;
  /* create a thread_data_t argument array */
  thread_data_t thr_data[NUM_THREADS];

  /* create threads */
  for (i = 0; i < NUM_THREADS; ++i) {
    thr_data[i].tid = i;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }
  /* block until all threads complete */
  for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thr[i], NULL);
  }

  return EXIT_SUCCESS;
}
```

This program creates `NUM_THREADS` threads and prints their respective user-assigned thread id. The first thing to notice is the call to `pthread_create()` in the main function. The syntax of the third and fourth argument are particularly important. Notice that the `thr_func` is the name of the thread function, while the fourth argument is the argument passed to said function. Here we are passing a thread function argument that we created as a `thread_data_t` struct. Of course, you can pass simple data types as pointers if that is all that is needed, or `NULL` if no arguments are required. However, it is good practice to be able to pass arguments of arbitrary type and size, and is thus illustrated for this purpose.

A few things to mention:

- Make sure you check the return values for all important functions.
- The second argument to `pthread_create()` is `NULL` indicating to create a thread with default attributes. The defaults vary depend upon the system and pthread implementation.

- Notice that we have broken apart the `pthread_join()` from the `pthread_create()`. Why is it that you should not integrate the `pthread_join()` in to the thread creation loop?
- Although not explicitly required to call `pthread_exit()` at the end of the thread function, it is good practice to do so, as you may have the need to return some arbitrary data back to the caller via `pthread_join()`.

pthread Attributes

Threads can be assigned various thread attributes at the time of thread creation. This is controlled through the second argument to `pthread_create()`. You must first pass the `pthread_attr_t` variable through:

```
int pthread_attr_init(pthread_attr_t *attr);

/* ignore me: needed so underscores above do not get clipped off */
```

Some attributes that can be set are:

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

/* ignore me: needed so underscores above do not get clipped off */
```

Attributes can be retrieved via complimentary `get` functions. Consult the man pages for the effect of each of these attributes.

pthread Mutexes

pthread mutexes are created through the following function:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);

/* ignore me: needed so underscores above do not get clipped off */
```

The `pthread_mutex_init()` function requires a `pthread_mutex_t` variable to operate on as the first argument. Attributes for the mutex can be given through the second parameter. To specify default attributes, pass `NULL` as the second parameter. Alternatively, mutexes can be initialized to default values through a convenient macro rather than a function call:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Here a mutex object named `lock` is initialized to the default pthread mutex values.

To perform mutex locking and unlocking, the pthreads provides the following functions:

```c
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

/* ignore me: needed so underscores above do not get clipped off */
```

Each of these calls requires a reference to the mutex object. The difference between the lock and trylock calls is that lock is blocking and trylock is non-blocking and will return immediately even if gaining the mutex lock has failed due to it already being held/locked. It is absolutely essential to check the return value of the trylock call to determine if the mutex has been successfully acquired or not. If it has not, then the error code `EBUSY` will be returned.

Let's expand the previous example with code that uses mutexes:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 5

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;

/* shared data between threads */
double shared_x;
pthread_mutex_t lock_x;

void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);
  /* get mutex before modifying and printing shared_x */
  pthread_mutex_lock(&lock_x);
    shared_x += data->stuff;
    printf("x = %f\n", shared_x);
  pthread_mutex_unlock(&lock_x);

  pthread_exit(NULL);
}

int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS];
  int i, rc;
  /* create a thread_data_t argument array */
  thread_data_t thr_data[NUM_THREADS];

  /* initialize shared data */
```

```
  shared_x = 0;

  /* initialize pthread mutex protecting "shared_x" */
  pthread_mutex_init(&lock_x, NULL);

  /* create threads */
  for (i = 0; i < NUM_THREADS; ++i) {
    thr_data[i].tid = i;
    thr_data[i].stuff = (i + 1) * NUM_THREADS;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }
  /* block until all threads complete */
  for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thr[i], NULL);
  }

  return EXIT_SUCCESS;
}
```

In the above example code, we add some shared data called `shared_x` and ensure serialized access to this variable through a mutex named `lock_x`. Within the `thr_func()` we call `pthread_mutex_lock()` before reading or modifying the shared data. Note that we continue to maintain the lock even through the `printf()` function call as releasing the lock before this and printing can lead to inconsistent results in the output. Recall that the code in-between the lock and unlock calls is called a critical section. Critical sections should be minimized for increased concurrency.

pthread Condition Variables

pthread condition variables are created through the following function call or initializer macro similar to mutexes:

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

/* ignore me: needed so underscores above do not get clipped off */
```

Similar to the mutex initialization call, condition variables can be given non-default attributes through the second parameter. To specify defaults, either use the initializer macro or specify `NULL` in the second parameter to the call to `pthread_cond_init()`.

Threads can act on condition variables in three ways: wait, signal or broadcast:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);

/* ignore me: needed so underscores above do not get clipped off */
```

`pthread_cond_wait()` puts the current thread to sleep. It requires a mutex of the associated shared resource value it is waiting on. `pthread_cond_signal()` signals *one* thread out of the possibly many sleeping threads to wakeup. `pthread_cond_broadcast()` signals *all* threads waiting on the `cond` condition variable to wakeup. Here is an example on using pthread condition variables:

```
void *thr_func1(void *arg) {
  /* thread code blocks here until MAX_COUNT is reached */
  pthread_mutex_lock(&count_lock);
    while (count < MAX_COUNT) {
      pthread_cond_wait(&count_cond, &count_lock);
    }
  pthread_mutex_unlock(&count_lock);
  /* proceed with thread execution */

  pthread_exit(NULL);
}

/* some other thread code that signals a waiting thread that MAX_COUNT has been reached */
void *thr_func2(void *arg) {
  pthread_mutex_lock(&count_lock);

  /* some code here that does interesting stuff and modifies count */

  if (count == MAX_COUNT) {
    pthread_mutex_unlock(&count_lock);
    pthread_cond_signal(&count_cond);
  } else {
    pthread_mutex_unlock(&count_lock);
  }

  pthread_exit(NULL);
}
```

In `thr_func1()`, we are locking the `count_lock` mutex so we can read the value of count without entering a potential race condition. The subsequent `pthread_cond_wait()` also requires a locked mutex as the second parameter to avoid a race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it (as explained from the man page on `pthread_cond_wait`). Notice how a `while` loop is used instead of an `if` statement for the `pthread_cond_wait()` call. This is because of spurious wakeups problem mentioned previously. If a thread has been woken, it does not mean it was due to a `pthread_cond_signal()` or `pthread_cond_broadcast()` call. `pthread_cond_wait()` if awoken, automatically tries to re-acquire the mutex, and will block if it cannot. Locks that other threads could be waiting on should be released *before* you signal or broadcast.

pthread Barrier

pthreads can participate in a barrier to synchronize to some point in time. Before a barrier can be called, a pthread barrier object must be initialized first:

```
int pthread_barrier_init(pthread_barrier_t *barrier, pthread_barrierattr_t *barrier_attr, unsigned int count);
```

```

pthread_barrier_t barrier = PTHREAD_BARRIER_INITIALIZER(count);

/* ignore me: needed so underscores above do not get clipped off */
```

Barrier objects are initialized like mutexes or condition variables, except there is one additional parameter, `count`. The count variable defines the number threads that must join the barrier for the barrier to reach completion and unblock all threads waiting at the barrier. If default barrier attributes are used (i.e. `NULL` for the second parameter), one can use the initializer macro with the specified `count`.

The actual barrier call follows:

```
int pthread_barrier_wait(pthread_barrier_t *barrier);

/* ignore me: needed so underscores above do not get clipped off */
```

This function would be inside thread code where the barrier is to take place. Once `count` number of threads have called `pthread_barrier_wait()` then the barrier condition is met and all threads are unblocked and progress continues.

Miscellaneous

Here are some suggestions and issues you should consider when using pthreads:

- You should check all return values for important pthread function calls!
- Sometimes it is desirable for a thread not to terminate (e.g., a server with a worker thread pool). This can be solved by placing the thread code in an infinite loop and using condition variables. Of course, there needs to be some terminating condition(s) to the infinite loop (i.e., `break` when it is deemed necessary).

Additional useful pthread calls:

- `pthread_kill()` can be used to deliver signals to specific threads.
- `pthread_self()` returns a handle on the calling thread.
- `pthread_equal()` compares for equality between two pthread ids
- `pthread_once()` can be used to ensure that an initializing function within a thread is only run once.
- There are many more useful functions in the pthread library. Consult pthreads man pages or the Nichols text (Appendix C).

Performance Considerations

The performance gains from using threads can be substantial when done properly and in the right problem context, but can it be even better? You should consider the following when analyzing your program for potential bottlenecks:

- **Lock granularity -** How "big" (coarse) or "small" (fine) are your mutexes? Do they lock your whole structure or fields of a structure? The more fine-grained you make your locks, the more concurrency you can gain, but at the cost of more overhead and potential deadlocks.
- **Lock ordering -** Make sure your locks are always locked in an agreed order (if they are not, make sure you take steps to rectify situations where locks are obtained in an out-of-order fashion, e.g. by using trylock/unlock calls).

- **Lock frequency -** Are you locking too often? Locking at unnecessary times? Reduce such occurences to fully exploit concurrency and reduce synchronization overhead.
- **Critical sections -** This has been mentioned before, but you should take extra steps to minimize critical sections which can be potentially large bottlenecks.
- **Worker thread pool -** If you are using a Boss/Worker thread model, make sure you pre-allocate your threads instead of creating threads on demand. It doesn't matter to the user how long it took your server to initialize, it only matters how fast it processes his or her request!
- **Contention scope -** Do your threads perform better when they are in contention with all of the system's processes? Or do they perform better when individually scheduled by the thread library itself? Only experimentation can give you the answers.
- **Scheduling class -** We have not touched on this topic, but changing the thread scheduling class from FIFO to RR can give better response times. But is this what you really want? Refer to Nichols or Lewis book for more information on thread scheduling classes.
- **Too many threads? -** At what point are there too many threads? Can it serverely impact and degrade performance? Again, only experimentation will give you the real answers to this question.

Other Approaches

C++ Template Libraries

There are various template libraries available that ease implementation of multithreading in a (semi-)portable fashion. For those programming in C++, you may want to look at Boost, Intel Threading Building Blocks (TBB) and POCO.

Multiprocess and Shared Memory

This tutorial has explored the very basics of multithreaded programming. What about multiprocess programming?

These topics are beyond the scope of this document, but to perform cross-process synchronization, one would use some form of IPC: pipes, semaphores, message queues, or shared memory. Of all of the forms of IPC, shared memory is usually the fastest (excluding doors). You can use `mmap()`, POSIX (e.g., `shm_open()`) or SysV (e.g., `shmget()`) semantics when dealing with cross-process resource management, IPC and synchronization. For those interested in shared memory programming in C++, I recommend looking at Boost.Interprocess first.

OpenMP

OpenMP is a portable interface for implementing fork-join parallelism on shared memory multi-processor machines. It is available for C/C++ and Fortran. For a quick introduction, please see the slides here.

MPI

The Message Passing Interface (MPI) is the de-facto standard for distributed memory parallel processing. Data can be sent/received from distinct computing machines with support for vectored I/O (scatter/gather), synchronization and collectives.

It is not uncommon to see programs that are both multithreaded and contain MPI calls to take advantage of shared memory within a node and MPI to perform processing across nodes.

Resources

It is difficult to cover more than an introduction to threads with this short tutorial and overview. For more in-depth coverage on threads (like thread scheduling classes, thread-specific data (thread local storage), thread canceling, handling signals and reader/writer locks) and pthreads programming, I recommend these books:

- Lewis, Bill and Daniel J. Berg. Multithreaded Programming with Pthreads. California: Prentice Hall, 1998.
- Nichols, Bradford, et. al. Pthreads Programming. Beijing: O'Reilly & Associates, Inc., 1998.

There are many excellent online resources regarding pthreads on the web. Use your favorite search engine to find these.

Notice: Please do not replicate or copy these pages and host them elsewhere. This is to ensure that the latest version can always be found here.

Disclaimer: The document author has published these pages with the hope that it may be useful to others. However, the document author does not guarantee that all information contained on these webpages are correct or accurate. There is no warranty, expressed or implied, of merchantability or fitness for any purpose. The author does not assume any liability or responsibility for the use of the information contained on these webpages.

If you see an error, please send an email to the address below indicating the error. Your feedback is greatly appreciated and will help to continually improve these pages.

© 1999-2021 Alfred Park (fred [ANTISPAM-REMOVE-THIS] AT randu.org)