Mutual Exclusion and Condition Variables in Linux

GWU CSCI 3411 - Fall 2019 - Lab 7
Prepared by James Taylor (Fall 2017)

September 29, 2019

Topics

1 Introduction

In Lab 5, we practiced mutithreading with shared memory and mutual exclusion in Linux using `pthreads`. Mutual exclusion protects access to shared memory, but does not guarantee optimal usage of processing resources. Using mutual exclusion alone produces a form of *busy waiting*, so threads end up spinning and consume processing resources when other threads or processes could use the processor instead. Condition variables allow for threads to use *block waiting* instead.

In this lab, we will study the difference between busy waiting and block waiting. We will implement a simple project twice, using mutual exclusion and using mutual exclusion with condition variables.

Note: Each Exercise is followed by a set of Questions. Read through the Exercise and the Questions and discuss the questions with your labmates before attempting to implement a solution. In your solution, try to find a way to examine the questions empirically.

2 Exercises

π can be approximated using a monte carlo approximation by randomly sampling a set of points within a square and then computing the ratio between points inside the unit circle and points inside the square. If random points are generated inside the upper left cartesian quadrant, where x and y are in the range $[0, 1]$, π is approximated by four times the ratio between the number of points inside the unit circle and the number of points in the overall set of samples. Note that as the number of samples increases, the accuracy of the π estimation increases.

In the repository for this lab, `circle.h` provides all of the structures and functions necessary to compute π using the Monte Carlo approach and `circle.c` provides an example of using `circle.h`. The repository also contains a `Makefile` and two skeleton files, `ex1.c` and `ex2.c`.

2.1 Exercise 1

Use multithreading with mutual exclusion only to compute the monte carlo approximation of π. Implement this exercise in `ex1.c`

Each worker thread will sample a point and determine whether or not the sample is inside the unit circle. The main thread will accumulate a total count of samples and a count of samples that are inside the unit circle. *Note:* The main thread does not need to know the point itself as the main thread only needs to know whether a sample is inside or outside the unit circle.

To communicate between the worker threads and the main thread, shared memory is required. The information shared will be encoded using a very simple scheme, and all threads will communicate through a single integer variable. The variable may have the values of -1, 0, and 1. If the value of the variable is -1, the main thread is waiting for a sample. If the value of the variable is 0, a worker thread has determined a sample to be outside the unit circle. If the value of the variable is 1, a worker thread has determined a sample to be inside the unit circle.

Note: You will need to use some form of busy waiting to successfully implement this exercise.

Your code should support the ability to change the number of worker threads easily and the ability to change the total number of samples easily. Using a `#define` is a simple way to control these parameters and your thread creation code should use a loop.

When testing your code, use two threads and a sample size of 10. Once your program works correctly, run your program to compute π using 1 million samples with 2 threads, 4 threads, and 40 threads.

What performance differences do you note between 2, 4, and 40 threads? Why do you suppose these differences exist?

2.1.1 Questions

If the mutex is held on this core, we *want* to block letting it progress. If it is on another core, context switches aren't free, so spinning might be better. If we get the critical section, what is the likelihood that we will be able to change the shared structure (i.e. it is set to -1)? If we can't then, we'll retry (or switch to another thread that was contending the critical section). So what is this likelihood, as that has 100% impact on the overheads of "how much spinning is there?"

What is the likelihood if we have N threads on 1 core?

What is the likelihood with a 1 to 1 ratio between number of threads and number of cores?

What is the likelihood with N threads on M cores where $N = M * x$?

2.2 Exercise 2

Use multithreading with mutual exclusion and condition variables to compute π using the monte carlo approximation. Implement this exercise in `ex2.c`

In this exercise, you will extend the program developed in Exercise 1 to use condition variables. *Note*: Condition variables will allow you to implement block waiting. Refer to Section 4.3 for information on `pthread` condition variable functions.

Condition variables allow you to block corresponding threads until their inputs are ready. Consider that there is only one main thread doing accumulation while there are multiple worker threads sampling data. Because there are multiple worker threads, you will probably want to use `pthread_cond_broadcast` to wake up all workers threads; because there is only one main thread, you will probably want to use `pthread_cond_signal` to wake up the main thread.

What is the ideal count for the number of threads that gives the best calculation time for 1 million samples?

2.2.1 Questions

By blocking with condition variables, we are *targeting* which process to take the mutex next. This should remove overhead from spinning. How much overhead from spinning is removed? Can you reduce overhead down to O(1)?

3 Discussion

Thread pools are collections of threads that take an argument from a main coordinator thread and compute on it. Arguments might be webpage requests, simulation requests, etc.

What is the cost of starting a thread on demand? What is the cost of spinning threads? How can we pool threads such that we avoid both of these costs?

4 Linux Multithreading and Mutual Exclusion Functions

This section documents the functions that you will use for multithreading and mutual exclusion in Linux. The documentation provided is most relevant excerpts from `man` pages for each of these functions; however, you are encouraged to look at the `man` pages if you require more information.

4.1 Linux multithreading

These functions enable the basic functionality of thread creation and thread joining of pthreads in Linux. There are additional functions associated with pthreads, such as creating, setting, and destroying thread attributes; however, we will not need these other functions for this lab. To gain a comprehensive knowledge of pthreads on Linux, you should study the rest of the pthreads API.

4.1.1 `pthread_create`

```
#include <pthread.h>

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

The `pthread_create()` function starts a new thread in the calling process. The new thread starts execution by invoking `start_routine()`; `arg` is passed as the sole argument of `start_routine()`.

The new thread terminates in one of the following ways:

- It calls `pthread_exit(3)`, specifying an exit status value that is available to another thread in the same process that calls `pthread_join(3)`.
- It returns from `start_routine()`. This is equivalent to calling `pthread_exit(3)` with the value supplied in the return statement.
- It is canceled (see `pthread_cancel(3)`).
- Any of the threads in the process calls `exit(3)`, or the main thread performs a return from `main()`. This causes the termination of all threads in the process.

The `attr` argument points to a `pthread_attr_t` structure whose contents are used at thread creation time to determine attributes for the new thread; this structure is initialized using `pthread_attr_init(3)` and related functions. If `attr` is `NULL`, then the thread is created with default attributes.

On success, `pthread_create()` returns 0; on error, it returns an error number, and the contents of `*thread` are undefined.

4.1.2 `pthread_join`

```
#include <pthread.h>

int pthread_join( pthread_t thread,
                  void **retval );
```

The `pthread_join()` function waits for the thread specified by `thread` to terminate. If that thread has already terminated, then `pthread_join()` returns immediately. The thread specified by `thread` must be joinable.

On success, `pthread_join()` returns 0; on error, it returns an error number.

4.2 pthreads Mutual Exclusion

These functions enable creation, deletion, locking, and unlocking of pthread mutexes in Linux. There are additional functions associated with mutexes, such as creating, setting, and destroying mutex attributes; however, we will not need these other functions for this lab.

4.2.1 `pthread_mutex_init` and `pthread_mutex_destroy`

```
#include <pthread.h>

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

The `pthread_mutex_destroy()` function shall destroy the mutex object referenced by `mutex`; the mutex object becomes, in effect, uninitialized. An implementation may cause `pthread_mutex_destroy()` to set the object referenced by `mutex` to an invalid value. A destroyed mutex object can be reinitialized using `pthread_mutex_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

The `pthread_mutex_init()` function shall initialize the mutex referenced by `mutex` with attributes specified by `attr`. If `attr` is `NULL`, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

If successful, the `pthread_mutex_destroy()` and `pthread_mutex_init()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

4.2.2 `pthread_mutex_lock` and `pthread_mutex_unlock`

```
#include <pthread.h>

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

The mutex object referenced by `mutex` shall be locked by calling `pthread_mutex_lock()`. If the mutex is already locked, the calling thread shall *block until the mutex becomes available*. This operation shall return with the mutex object referenced by `mutex` in the locked state with the calling thread as its owner.

The `pthread_mutex_unlock()` function shall release the mutex object referenced by `mutex`. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by `mutex` when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

If successful, the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

4.3 pthreads Condition Variables

4.3.1 `pthread_cond_init` and `pthread_cond_destroy`

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

The `pthread_cond_destroy()` function shall destroy the given condition variable specified by `cond`; the object becomes, in effect, uninitialized. An implementation may cause `pthread_cond_destroy()` to set the object referenced by `cond` to an invalid value. A destroyed condition variable object can be reinitialized using `pthread_cond_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

The `pthread_cond_init()` function shall initialize the condition variable referenced by `cond` with attributes referenced by `attr`. If `attr` is `NULL`, the default condition variable attributes shall be used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable shall become initialized.

Only cond itself may be used for performing synchronization. The result of referring to copies of `cond` in calls to `pthread_cond_wait()`, `pthread_cond_timedwait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`, and `pthread_cond_destroy()` is undefined.

Attempting to initialize an already initialized condition variable results in undefined behavior.

In cases where default condition variable attributes are appropriate, the macro `PTHREAD_COND_INITIALIZER` can be used to initialize condition variables that are statically allocated. The effect shall be equivalent to dynamic initialization by a call to `pthread_cond_init()` with parameter `attr` specified as `NULL`, except that no error checks are performed.

If successful, the `pthread_cond_destroy()` and `pthread_cond_init()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

4.3.2 **`pthread_cond_signal`** and **`pthread_cond_broadcast`**

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

These functions shall unblock threads blocked on a condition variable.

The `pthread_cond_broadcast()` function shall unblock all threads currently blocked on the specified condition variable `cond`.

The `pthread_cond_signal()` function shall unblock at least one of the threads that are blocked on the specified condition variable `cond` (if any threads are blocked on `cond`).

If more than one thread is blocked on a condition variable, the scheduling policy shall determine the order in which threads are unblocked. When each thread unblocked as a result of a `pthread_cond_broadcast()` or `pthread_cond_signal()` returns from its call to `pthread_cond_wait()` or `pthread_cond_timedwait()`, the thread shall own the mutex with which it called `pthread_cond_wait()` or `pthread_cond_timedwait()`. The thread(s) that are unblocked shall contend for the mutex according to the scheduling policy (if applicable), and as if each had called `pthread_mutex_lock()`.

The `pthread_cond_broadcast()` or `pthread_cond_signal()` functions may be called by a thread whether or not it currently owns the mutex that threads calling `pthread_cond_wait()` or `pthread_cond_timedwait()` have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex shall be locked by the thread calling `pthread_cond_broadcast()` or `pthread_cond_signal()`.

The `pthread_cond_broadcast()` and `pthread_cond_signal()` functions shall have no effect if there are no threads currently blocked on `cond`.

If successful, the `pthread_cond_broadcast()` and `pthread_cond_signal()` functions shall return zero; otherwise, an error number shall be returned to indicate the error.

4.3.3 **`pthread_cond_wait`**

```
#include <pthread.h>

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

The `pthread_cond_wait()` function shall block on a condition variable. They shall be called with mutex locked by the calling thread or undefined behavior results.

These functions atomically release mutex and cause the calling thread to block on the condition variable `cond`; atomically here means ”atomically with respect to access by another thread to the mutex and then the condition variable”. That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to `pthread_cond_broadcast()` or `pthread_cond_signal()` in that thread shall behave as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.

When using condition variables there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the `pthread_cond_wait()` function may occur. Since the return from `pthread_cond_wait()` does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return.

The effect of using more than one mutex for concurrent `pthread_cond_wait()` operations on the same condition variable is undefined; that is, a condition variable becomes bound to a unique mutex when a thread waits on the condition variable, and this (dynamic) binding shall end when the wait returns.

A condition wait (whether timed or not) is a cancellation point. When the cancelability enable state of a thread is set to `PTHREAD_CANCEL_DEFERRED`, a side effect of acting upon a cancellation request while in a condition wait is that the mutex is (in effect) re-acquired before calling the first cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to `pthread_cond_wait()`, but at that point notices the cancellation request and instead of returning to the caller of `pthread_cond_wait()`, starts the thread cancellation activities, which includes calling cancellation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to `pthread_cond_wait()` shall not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it shall return zero due to spurious wakeup.

Upon successful completion, a value of zero shall be returned; otherwise, an error number shall be returned to indicate the error.