Tutorial 4 - Kernel level threads

Gidon Rosalki

2025-04-23

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

1 Introduction to multithreading

Modern computers have multiple cores. Each resides within the CPU (some computers have multiple CPUs). Cores run commands simultaneously (truly in parallel). This is useful when using blocking operations, since we may use other processors while one is blocking.
We shall define the concept of a ”task”. These are user defined, and are sections of code that are processed simultaneously, such that the program will benefit from the multiple resources of the computer, and can thus run faster. For example, one mught need to perform a task on every object in an array, but if they are independent of each other, these can be split up to different cores. It is important to remember the difference between user level threads, and kernel level threads (previous tutorial).

We will be using the C++ thread class to represent this, from the std::thread library.

2 Creating threads

A thread runs a function (which may of course call other functions). Creating this thread is done by constructing an object. After construction, the thread enters the scheduling queue immediately. The default empty constructor runs nothing.
The constructor should refer to the thread’s function, and arguments to pass to it, if any. For example:

```cpp
#include <iostream>
#include <thread>

void thread_func(int x, int y)
{
    std::cout << "x = " << x << " and y = " << y << std::endl;
}

int main()
{
    std::cout << "Creating threads" << std::endl;
    std::thread t1(thread_func, 1, -1);
    std::thread t1(thread_func, 2, -2);
}
```

However, this code block will result in an error ”Terminate called without an active exception”. What happened is that this process has 3 threads, `main`, `t1`, and `t2`. In execution, `main` continues to run after creating the others, reaches the end, and then calls their destructors. This is illegal since no thread may be destructed before it is **joined** (or cancelled).

2.1 Join

Joining threads means that the one thread waits until the thread on which we called join finishes. This in essence blocks the thread that called join on the second (so here main is blocked until the others finish).

```cpp
#include <iostream>
#include <thread>

void thread_func(int x, int y)
{
    std::cout << "x = " << x << " and y = " << y << std::endl;
}

int main()
{
    std::cout << "Creating threads" << std::endl;
    std::thread t1(thread_func, 1, -1);
    std::thread t1(thread_func, 2, -2);

    t1.join();
    t2.join();
}
```

So here the output will be the result of the printing, in any order, since this happened asynchronously.

2.2 jthreads

C++ has another library (since C++20) called jthreads, which ensures that the thread is automatically destroyed in destruction, and you therefore have no need to call join. There is also std::this_thread, which adds the function yield, which hints to the scheduler to run another thread instead of the calling thread (much like yield in python, allows continued execution from elsewhere). Additionally there is sleepfor(), which sleeps for a given amount of time (from std::chrono). We also have lambdas, allowing initialisation of threads using anonymous functions:

```
int x = 0;
std::thread t ([] (const int &wait) { sleep (wait); }, 5);
t.join ();
std::cout << "Out "! << std::endl;
```

2.3 Variable accessibility

Each thread running a function has its own stack (in a similar manner to user level threads). However, it is import to note that threads **can** access other threads variables if given their address, because threads are still living in the same process address space. However, when the thread terminates, the variable will be destroyed. It is typical to have shared variables allocated dynamically in the heap, or put into the data segment (global).

2.3.1 Mutual exclusion

It is better if threads do not share resources, consider for example the following function:

```
int x = 0;
void
thread_function ()
{
    for (int i = 0; i < 1000000; i++)
        {
            x++;
        }
}
```

Should this code be run with more than one thread, we will get many possible options for the value of x. This is because `x++` is not an atomic operation, it involves 3 asm instructions. Therefore, if there are two operations implementing it at almost the same time, they will both take the old value, add one to it, and then update to their new value, resulting in a final value of `x + 1` instead of `x + 2`. This is called a **race condition**.

2.3.2 Atomic variables

C++ offers a wrapper std::atomic, which makes a variable ”atomic”, in that it provides atomic operations such as load & store, compare and swap (CAS), and fetch & add. So in short, the wrapper overrides the shorthand arithmetic and bitwise operations (ie +=), but does **not** override RHS (such as `x = x + 5`). So, if we correct the above code:

```
std::atomic<int> x = 0;
void
thread_function ()
{
    for (int i = 0; i < 1000000; i++)
        {
            x++;
        }
}
```

this is correct.

2.3.3 Mutex

The c++ standard library fuctions and classes are **not** thread safe by default. Thread safety is when multiple threads accessing the same container is well defined, with no problems. For example:

```
#include <iostream>
#include <thread>
#include <vector>
#define RANGE 1000000
void
thread_function (std::vector<int> &shared, int range)
{
    for (int i = 0; i < range; i++)
        {
            shared.push_back (i);
        }
}
int
main ()
{
    std::vector<int> vec;
    {
        std::jthread threads[8];
        for (int i = 0; i < 8; i++)
            {
                threads[i]
                    = std::jthread (thread_function, std::ref (vec), RANGE);
            }
    }
    std::cout << "vec has " << vec.size () << " elements" << std::endl;
    return 0;
}
```

Here, instead of the expected output of vec containing some ordering of numbers, because when the vector runs out of space, then there is a reallocation, to create a larger array, and the memory can be moved within the heap, resulting in other threads that are possibly in the middle of operations, accessing unallocated areas of memory, resulting in `munmap` errors. The solution to this is a `mutex`. This is the simplest primitive synchronisation mechanism, which is also called a lock. It is used to protect critical sections. When lock / acquire locks the mutex, any other thread calling lock will be blocked, or slept. Unlock / release releases the mutex, and wakes up waiting threads (if they exist).
C++ implements a mutex in std::mutex, which is a class supplying an abstraction of mutex. This **must** be shared between the threads. To make an example of using them:

```
#include <iostream>
#include <mutex>
#include <thread>
void
thread_func (std::mutex &m, int thread_num)
{
   m.lock ();
   std::cout << "I am thread " << thread_num << std::endl;
   m.unlock ();
}
int
main ()
{
   std::mutex m;
   std::jthread threads[10];
   for (int i = 0; i < 10; i++)
      {
         threads[i] = std::jthread (thread_func, std::ref (m), i);
      }
}
```

It is important to note that mutexes **must be** passed by reference, or by pointer, since otherwise it is copied, and thus two threads can lock at the same time, and it has given you nothing.

However, we do not trust the user to remember to always lock and unlock their mutexes. So instead we have std::unique_lock. This is a wrapper helper class for using mutexes, which allows calling lock() again without blocking (recursive locking), ownership transfer and lock swaps, and use for condition variables (later). This uses RAII (resource acquisition is initialisation). Calling to unique_lock's constructor **automatically locks the thread**, and similarly its destructor automatically releases the mutex.

```
#include <iostream>
#include <mutex>
#include <thread>
void
thread_func (std::mutex &m, int thread_num)
{
   std::unique_lock<std::mutex> lock (m);
   std::cout << "I am thread " << thread_num << std::endl;
}
int
main ()
{
   std::mutex m;
   std::jthread threads[10];
   for (int i = 0; i < 10; i++)
      {
         threads[i] = std::jthread (thread_func, std::ref (m), i);
      }
}
```

2.3.4 Semaphores

A semaphore is a generalised version of a mutex. This allows an entrance of **at most** n threads to a critical section, however unlike a mutex, its value **can** be modified in runtime. We have a few types:

- std::counting_semaphore: The constructor takes the initial value, release() releases the semaphore (up / post), and acquire() acquires the semaphore (down / wait).
- std::binary_semaphore: This is equivalent to std::counting_semaphore<1> (ie, only 1 thread allowed). It is different from a mutex, since in a mutex the thread that locked the mutex is the only thread that can allow another into the critical zone, but in a binary_semaphore, all other threads can release the semaphore.

2.3.5 Conditional variables

A conditional variable (also called monitor), is a synchronisation object used to control conditional entrance to a critical section. They may send signals, and broadcasts, on signal it notifies a single thread that it is permitted to enter, and broadcast notifies all threads they are permitted to continue. They are used as a sleep - wake up mechanism **inside a critical section**. Before waiting, we acquire a mutex and enter the first part of the critical section. Now we wait, we don't want other threads waiting to enter a critical section to be blocked, so we release the mutex. When another thread wakes us up, we acquire the mutex again, and continue at the critical section, with the mutex locked, and release it at the end. The three important functions here are wait, notify_one, and notify_all.

According to the standard, a wait call may be awakened spuriously, as in even if there was not a call to notify, or there was a call that has already woken up another thread. Although these are rare, it is important to check for the condition completion, and calling wait again. (ie, use while instead of if for this condition). There is also a "wait predicate", so add the requirement inside the wait call as follows:

```
cv.wait(lock, [&x](){x == 4});
```

This calls the predicate (returns true only if x is 4), and goes to sleep while it is false.

2.3.6 Barrier

This is a location of code all threads (or some of them) are required to reach, before all can continue to the next instruction. Barrier is **not** a primitive synchronisation mechanism. Instead, we shall implement it ourselves. The idea is that a shared variable counts the number of arrived threads, with a CV wait for the release event, and if the shared variable reaches the required number of threads, then it broadcasts.

Listing 1: barrier.h

```
#include <condition_variable>
#include <mutex>
class Barrier
{
  public:
    Barrier (int num_threads); // constructor
    void wait ();          // wait for all threads to arrive
  private:
    int arrived;
    int num_threads;
    std::condition_variable cv;
    std::mutex mtx;
};
```

Listing 2: barrier.cpp

```
void
Barrier::wait ()
{
    // acquire mutex to increment `arrived`
    std::unique_lock<std::mutex> lock (this->mtx);
    // now the mutex is locked
    int currently_arrived = ++this->arrived;
    if (currently_arrived == this->num_threads)
        { // if all threads have arrived, notify all (broadcast)
            this->arrived = 0; // reset this->arrived
            this->cv.notify_all ();
        }
    else
        { // wait for broadcast, release the lock
            while (this->arrived != 0)
                {
                    this->cv.wait (lock);
                }
        } // reaching to this point, the mutex is locked.
    // the mutex is automatically released since we used
    std::unique_lock
}
```

2.3.7 Thread local cache (TLS)

Thread local cache is a memory management method, allowing the data segment variables to have a separate copy on each thread. Recall, the data segment saves function static variables, and global variables. Unlike the normal behaviour of shared static and global variables, the variables are **not** shared among the threads, each thread uses its own copy.

In C++ we have the keyword thread_local:

```cpp
#include <iostream>
#include <thread>
thread_local int x;
void
thread_func (int tid)
{
   x = tid;
   std::this_thread::sleep_for (std::chrono::microseconds (200));
   // t2 probably already updated x
   // we would expect x to be 1
   std::cout << "x = " << x << std::endl;
}

int
main ()
{
   std::jthread t1 (thread_func, 0);
   // t2 delays and updates x probably after t1
   std::this_thread::sleep_for (std::chrono::microseconds (100));
   std::jthread t2 (thread_func, 1);
   return 0;
}
```

It is difficult to debug multithreaded programs, since they are non-deterministic. Be wary of print debugs, since print does not necessarily print to the screen at the same moment, and print also causes delays, which can affect race conditions that you want to debug. Additionally breakpoints are not individual for each thread, and each time a thread reaches a breakpoint, the whole process stops. You can alternate between threads in a debugger, and see their individual stacks.