POSIX Threads

Contents of Lecture 5

- POSIX Threads architecture — threads, synchronization, and scheduling.
- Functions and types for starting and stopping threads
- Functions and types for synchronization
- The memory consistency model for POSIX Threads
- Thread cancellation
- Thread private data in POSIX Threads, GNU C, and C11
- POSIX Threads and OS kernel threads
- POSIX Threads realtime scheduling

POSIX Threads Overview

- POSIX stands for Portable Operating System Interface for UNIX.
- POSIX is thus not an implementation of UNIX but rather specifications of the APIs available for UNIX implementations.
- The standards in POSIX are specified by the IEEE.
- POSIX Threads are described by three parts:
  - Functions for runnning threads
  - Functions for synchronizing threads
  - Functions for scheduling threads

The pthread_t Type

- All types in POSIX Threads are opaque, which means client code should regard them as secret and not rely on any particular knowledge about them.
- FILE is another example of an opaque type.
- The most important type is: pthread_t.
- It can be regarded as a "key" to identify a thread between the client and the POSIX Threads implementation.
- When a thread is created, the client code passes a pointer to a variable of this type which is filled in by the library code.

Creating and Starting a POSIX Thread

```
int pthread_create(
        pthread_t*                thread,          // output param.
        const pthread_attr_t*     attr,            // input param.
        void*                     (*work)(void*), // a function.
        void*                     arg);            // param to work.

arg_struct_t arg_struct = { n, a, b, c }; // args to work.

status = pthread_create(&thread, NULL, work, &arg_struct);
```

- The thread identifier is filled in by the call and attributes are optional.
- The created thread runs the `work` function and then terminates.
- Typically multiple arguments are passed in a struct as above.

Return Value from `pthread_create()`

- A zero return value from `pthread_create()` indicates success, while a nonzero describes an error printable with `perror`.
- The `work` function can return a void pointer.
- Calling `pthread_join` waits for the termination of another thread and also gives access to the returned void pointer.
- A thread can only be joined once.

Terminating a Thread

- There are three ways to terminate a thread:

1. Return from the work function.
2. The thread can call the function `void pthread_exit(void* value)`. The Standard C Library function `void exit(int status)` should normally not be used since it terminates the entire program.
3. Calling the function `int pthread_cancel(pthread_t thread)` makes a request to terminate the specified thread. See cancellation below.

- The first two are used by the thread itself and the third is used to stop another thread.
- Stopping another thread can be useful e.g. when a user has hit a "Cancel" button or another thread already has found a winning chess move.

Thread Cancellation Overview

- Simply terminating a thread can be disastrous if for example it has locked a mutex and is modifying shared data.
- Therefore the `pthread_cancel` is asynchronous and simply requests that the thread should terminate. To actually know that the thread has terminated, it must be joined with.
- A thread that received a cancellation request is informed about this fact at certain points in the program, called **cancellation points**.
- The termination of the thread is started when it comes to such a cancellation point, if it has a **pending** cancellation request.
- A thread can install a function that is executed before the thread actually terminates.

Cancellation State and Type

- For cancellation, a thread has two variables each with two possible values.
- The variables cannot be accessed directly but only through function calls.
- They are:
  - **State** — cancellation is either enabled or disabled.
  - **Type** — cancellation is either asynchronous or deferred.
- These result in three different cancellation modes:
  1. **Disabled** — any cancellation request received is saved until cancellation is enabled in the future.
  2. **Deferred** — cancellation is started at a cancellation point if there is a pending cancellation request.
  3. **Asynchronous** — cancellation can start at any time.

Modifying the Cancellation Mode

- The functions to modify the cancellation mode returns the thread's old value of the respective variable.
- `int pthread_setcancelstate(int state, int* old);` The state must be one of:
  - `PTHREAD_CANCEL_ENABLE`
  - `PTHREAD_CANCEL_DISABLE`
- `int pthread_setcanceltype(int type, int* old);` The type must be one of:
  - `PTHREAD_CANCEL_DEFERRED`
  - `PTHREAD_CANCEL_ASYNCHRONOUS`
- The default mode for new threads is **deferred**.

Cancellation Points

- A number of functions are cancellation points, including

```
pthread_cond_wait          pthread_cond_timedwait
pthread_testcancel         pthread_join
close                      creat
open                       read
system                     wait
waitpid                    write
```

- POSIX guarantees that the above (and some others) are cancellation points.
- Another list contains possible cancellation points, including

```
printf                     scanf
fopen                      fclose
```

- ANSI C and POSIX functions not on any of those lists are guaranteed not to be cancellation points.

Receiving a Cancellation Request

- Thus if a cancellation request is received while cancellation is disabled, the request is simply blocked until it is enabled again.
- A pending request is delivered when the thread comes to a function which is a cancellation point.
- Changing the cancellation mode is **not** a cancellation point.
- At a cancellation point the thread first executes any installed cleanup handler (see below) and then terminates the thread.
- The return value from a cancelled thread is `PTHREAD_CANCELED` which thus is a valid value of a void pointer.

Installing Cleanup Handlers

- Each thread has a stack of cleanup handlers.
- They are installed with the function:
  `void pthread_cleanup_push(void (*func)(void*), void* arg);`
- The argument `arg` will be passed to `func` when it is executed.
- To remove a cleanup handler, use the function:
  `void pthread_cleanup_pop(int execute);`
- If the argument `execute` is nonzero, the cleanup handler will first be executed and then popped.
- When a thread is about to be terminated, all cleanup handlers on the stack are executed, starting with what is on the top of the stack.

Execution of Cleanup Handlers

- There are three situations when one or all cleanup handlers are executed:
  1. When a thread is being terminated due to a cancellation.
  2. When a thread is being terminated due to it has called `pthread_exit`.
  3. When it has called `pthread_cleanup_pop` with a nonzero parameter.
- In the last case, only one cleanup handler is executed, as we just saw.

Synchronization

- POSIX Threads has three main primitives for synchronization:
  - Mutex
  - Condition variable
  - Barriers

Avoid Synchronization!

- Ideally a parallel program needs no synchronization.
- Synchronization and therefore data communication between threads/caches take time.
- Some problems can be divided into suitable tasks statically.
- However, a common problem if T tasks are statically assigned to P threads is that some tasks take more time and therefore there becomes an inbalance in the work load, i.e. some threads take much longer time than the others.
- We will next look at a program with dynamic task assignment.

Mutex

- A POSIX Threads **mutex** is a lock with a sleep queue — and not a spin lock.
- The type is `pthread_mutex_t` and the most important functions related to it are:
- `pthread_mutex_init`
- `pthread_mutex_destroy`
- `pthread_mutex_lock`
- `pthread_mutex_trylock`
- `pthread_mutex_unlock`
- All five take a pointer to a `pthread_mutex_t`, and `pthread_mutex_init` also takes a pointer to attributes, which may be `NULL`. We will look at the attributes below.

A Worklist

- Assume we have a number of tasks to be processed.
- We put the tasks in a list and create threads which take tasks from the list and process them.
- Concurrently adding or removing of items in the list means the list must be protected, otherwise the program suffers data races and a likely disaster.
- Two alternatives:
  1. Put a mutex lock in each list head, i.e. protect the data.
  2. Use a common mutex for all lists, i.e. protect the code.
- Which is best depends on the application. There can be more concurrency if each list head has its own lock, at the cost of memory...

Condition Variables

- A condition variable lets a thread wait for something to happen in the future, and another thread to inform it that it has happened.
- For example: a worker thread can wait for a task being inserted in the list and the master can signal any waiting worker thread that it just has inserted a new task.
- The POSIX Thread condition variable type is: `pthread_cond_t`.
- In addition to initialization and destruction functions the main functions are:
  - `pthread_cond_wait` — causes calling thread to wait
  - `pthread_cond_signal` — wakes up one waiting thread
  - `pthread_cond_broadcast` — wakes up all waiting threads

Pthread Mutex and Condition Variable

- Normally a thread has taken a mutex and then inspects the state of a data structure.
- If the thread must wait for a change in the state by some other thread it needs to call `pthread_cond_wait`.
- It must also unlock the mutex, otherwise no other thread can change the state.
- Suppose it wrongly would first unlock the mutex and then wait. Then by Murphy's Law (if something bad can happen, it will) the other thread signals the condition just before our thread calls `pthread_cond_wait` and it will therefore never wake up.
- On the other hand, it cannot first go to sleep and unlock the mutex...
- The solution is to do both operations in the call to `pthread_cond_wait` and therefore a pointer to the mutex is also passed as an argument to `pthread_cond_wait`.
- When the thread wakes up again, it will have the mutex locked.

More Details

- One mutex may be used for multiple condition variables, such as a mutex for protecting a buffer with the condition variables to signal to a consumer thread that a buffer is no longer empty or to a producer that it is no longer full.
- Two threads wanting to wait for the same condition variable **must** use the same mutex.
- It is legal to signal a condition variable without having locked the corresponding mutex, but not so common.

Predicate, Condition Variable, and Mutex

- The logic expression in the C code which decides whether a thread should wait on the condition variable is called the **predicate** associated with the condition variable.
- The predicate is computed from shared data which different threads can modify, and therefore that data must be protected using a mutex.
- For example, the predicate may be computed by a boolean function:

```
bool empty(buffer_t* buffer);
```

- The predicate should **always** be tested in a loop and **not** in an if-statement.
- Strictly speaking, some programs for example some with only two threads, might work correctly without a loop but you will enjoy life more if you use loops. See next slide.

Why You Need A Loop

- You should write your code like this.

```
pthread_mutex_lock(&mutex);
while (!predicate())
        pthread_cond_wait(&cond, &mutex);
/* do something... */
pthread_mutex_unlock(&mutex);
```

- There are at least three reasons for doing so:
  1. **Intercepted wakeups:** Another thread might have locked the mutex before yours.
  2. **Loose predicates:** This is a kind of predicate that says "the predicate may be true (but check before relying on it)."
  3. **Spurious wakeups:** This is very uncommon and is essentially an error that a thread was woke up without any good reason.

Intercepted Wakeups

- Should you signal a condition variable before or after you unlock its associated mutex (in case you have it locked) ???
- If you signal first, then the woke up thread will immediately try to lock the mutex and find it locked and wait again, now instead on a mutex, causing unnecessary context switching and synchronization overhead, both in the form of instructions and cache misses.
- If you unlock first, another thread may take the lock before the thread you wake up. That is not wrong, of course.
- It means the predicate might not longer be true after the other thread has unlocked the lock and it is your turn.
- This is called an intercepted wakeup.
- Due to intercepted wakeups, you must check the condition in a loop.

Loose Predicates

- It may be more convenient and/or efficient to say "you might have something interesting to check out" rather promising something.
- The woke up thread then must itself determine if there really was something for it, or whether it should continue waiting.
- This is called a loose predicate.

Spurious Wakeups

- When you hit CTRL-C to terminate a program you send a so called UNIX signal to it. This use of the word signal has nothing to do with the signal function of condition variables.
- When a thread receives a UNIX signal and was in the UNIX kernel waiting for a system call to complete, that system call is terminated and returns with the error code `EINTR`.
- Some UNIX signals are sent to all threads of a running program (called a **process**) and a thread waiting on a condition variable, i.e. in a system call on UNIX will thus be interrupted to handle the signal.
- An interrupted system call is not resumed but the application proceeds after it has returned.
- In principle a system call used by `pthread_cond_wait` could be interrupted and result in a spurious wakeup, but that is not the behaviour on Linux which uses the **futex** system call described below.

Implementation of Linux Native Pthreads Library

- Should a mutex lock involve the Linux kernel?
- Preferably not!
- On Linux there is a low-level synchronization primitive called **futex** which is used to implement the Pthreads library.
- We will next look at the implementation of `pthread_mutex_lock`.

User Level Locking with Futex in Linux

- Originates from IBM Reseach and the IBM Linux Technology Center.
- Implemented in the GNU C Library and in the Linux kernel, since version 2.5.7.
- The lock variable is in user space in shared memory and there is a corresponding wait queue for a lock in the kernel.
- The fast case is when there is no contention for the lock and therefore the kernel needs not be involved.
- The lock is manipulated in user space with atomic instructions.

Initialization of a Pthread Mutex

```
pthread_mutex_t        mutex = PTHREAD_MUTEX_INITIALIZER;
```

- This initialises the mutex with default attributes.
- `PTHREAD_MUTEX_INITIALIZER` is a constant expression, meaning we can initialise a mutex like this at file scope (static storage, ie a `static` or global variable).
- The mutex starts out as unlocked (of course).
- If allocated by eg malloc, then `pthread_mutex_init()` should be called.
- After usage, `pthread_mutex_destroy` should be called for a mutex, and then its memory should be deallocated using `free`, if appropriate.

pthread_cond_timedwait

- To wait on a condition variable with a time out, use `pthread_cond_timedwait`.

```
int pthread_cond_timedwait(
        pthread_cond_t*,
        pthread_mutex_t*,
        struct time_spec*);
```

- The time is absolute time and to wait eg for at most 3 seconds, one can use:

```
timeout.tv_sec = time(NULL) + 3;
timeout.tv_nsec = 0;
```

- If there is a time out, the return value is `ETIMEDOUT`.

Initialisation in Sequential Programs

- The usual sequential way to initialise is to have code like this:

```
#include <stdbool.h>
void f(void)
{
        static bool initialised = false;
        if (!initialised) {
                init();
                initialised = true;
        }
        ...
}
```

- Might not work in a multithreaded program!

Initialisation in multithreadthreaded programs

```
#include <stdbool.h>
pthread_mutex_t  init_lock;
void f(void)
{
        static bool initialised = false;
        pthread_mutex_lock(&init_lock);
        if (!initialised) {
                init();
                initialised = true;
        }
        pthread_mutex_unlock(&init_lock);
}
```

Initialisation in multithreadthreaded programs

- Now we just need to initialise the mutex.... :-)
- If it can be done in main before any other threads are created, then fine.
- In the current (and future) Pthreads standard, as we have seen, we can do

```
pthread_mutex_t          init_lock = PTHREAD_MUTEX_INITIALIZER;
```

- In earlier versions of the standard, it was not guaranteed that the right hand side of above, `PTHREAD_MUTEX_INITIALIZER`, was a constant expression, and hence could not be an initialiser of data with "static storage duration" (ISO C terminology for static or global variable) before the program started.

pthread_once

- One can write `pthread_once_t once = PTHREAD_ONE_INIT;`.
- The `once` variable can have static storage duration.
- The function `int pthread_once(pthread_once_t*, void (*)(void));` is used to call a function once. It takes a pointer to a "once" variable and a function to execute for the initialisation.
- If another thread executes the same call after the first is done, nothing will happen.
- If it instead calls it during the first call, the second thread will wait until the first call is done.

Thread attributes

- Examples of attributes which can be set:
  - Whether another thread can join with a particular thread (portable).
  - Stack size (not portable)
  - Stack address (even less portable)
- A thread which is not joinable is "detached" which means the resource used by the thread are recycled immediately when the thread terminates.
- The joinable attribute can be set to one of
  - `PTHREAD_CREATE_JOINABLE`, or
  - `PTHREAD_CREATE_DETACHED`.
- An initially joinable thread can make itself detached but not vice versa.

Setting the detached and the stack size attributes

```
pthread_attr_t          attr;
pthread_t               thread;
void*                   work(void*);

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_attr_setstacksize(&attr, 2 * PTHREAD_STACK_MIN);
pthead_create(&thread, &attr, work, arg);
```