# Lecture 6 — Working with Threads

Patrick Lam & Jeff Zarnett
patrick.lam@uwaterloo.ca jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 20, 2018

# POSIX Threads

- Available on most systems
- Windows has pthreads Win32, but I wouldn't use it; use Linux for this course
- API available by `#include <pthread.h>`
- Compile with pthread flag (`gcc -pthread prog.c -o prog`)

# Pthread Functions

Need a refresher? See the pthreads.pdf document in the course repository!

```
pthread_create( pthread_t *thread, const pthread_attr_t *attributes,
                void *(*start_routine)( void * ), void *argument )
pthread_join( pthread_t thread, void **return_value )
pthread_detach( pthread_t thread )
pthread_cancel( pthread_t thread )
pthread_testcancel( ) /* If the thread is cancelled, this function does not
    return (thread terminated) */
pthread_exit( void *value )
```

# C++ 11 Threads

- Now part of the C++ standard (library)
- API available with `#include <thread>`
- Compile with flags:
  (`g++ -std=c++11 -pthread prog.c -o prog`)

# Creating Threads—C++11 Example

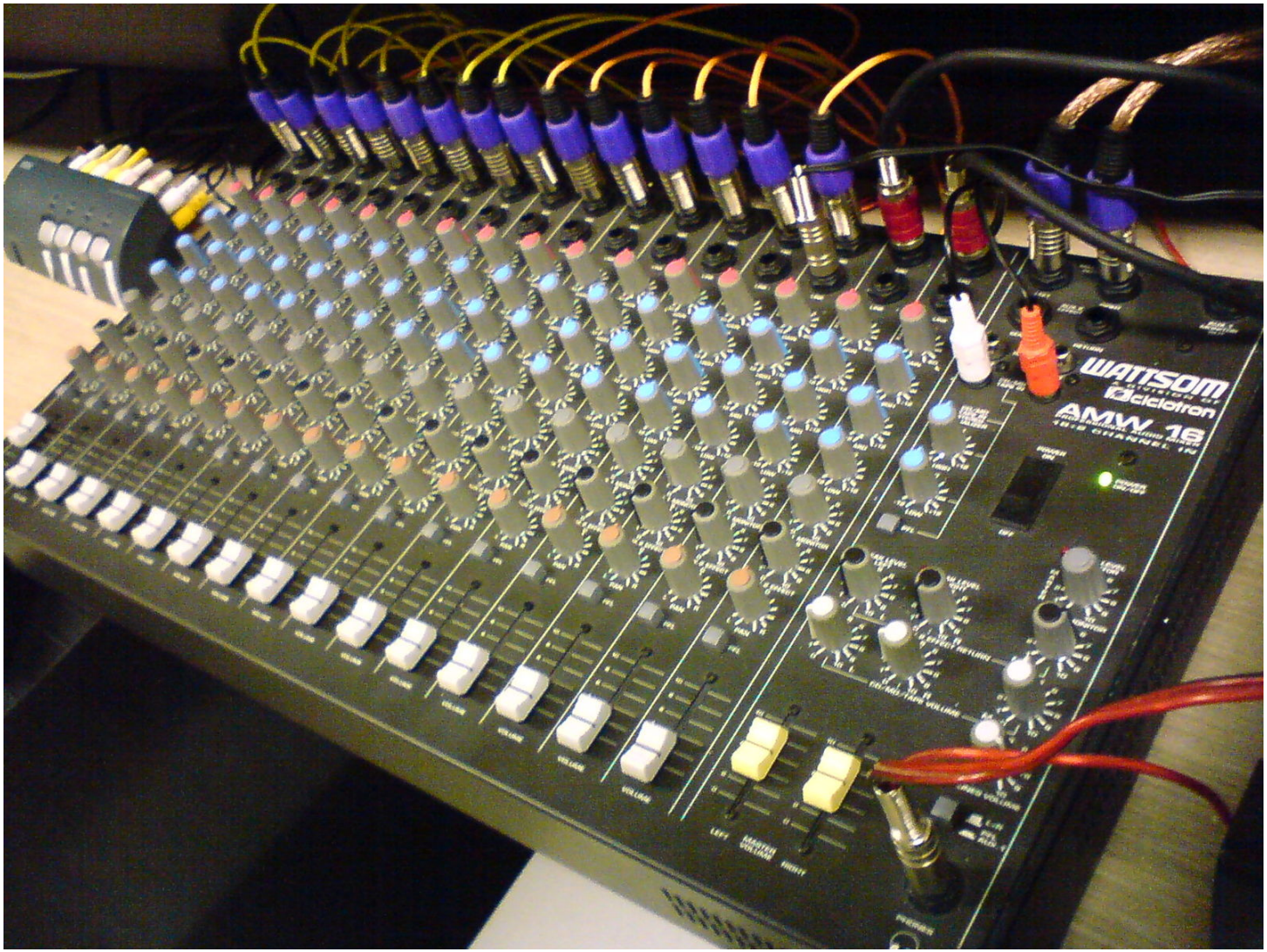
```
#include <thread>
#include <iostream>

void run() {
  std::cout << "In run\n";
}

int main() {
  std::thread t1(run);
  std::cout << "In main\n";
  t1.join(); // hang in there...
}
```

# Attributes

In previous courses, the default attributes were fine...
But now we should know about them!



# Attributes

By default, threads are *joinable* on Linux, but a more portable way to know what you're getting is to set thread attributes. You can change:

- Detached or joinable state
- Scheduling inheritance
- Scheduling policy
- Scheduling parameters
- Scheduling contention scope
- Stack size
- Stack address
- Stack guard (overflow) size

# Attributes—Example

```
size_t stacksize;
pthread_attr_t attributes;
pthread_attr_init(&attributes);
pthread_attr_getstacksize(&attributes, &stacksize);
printf("Stack size = %i\n", stacksize);
pthread_attr_destroy(&attributes);
```

Running this on a laptop produces:

```
jon@riker examples master % ./stack_size
Stack size = 8388608
```

Setting a thread state to joinable:

```
pthread_attr_setdetachstate(&attributes,
                            PTHREAD_CREATE_JOINABLE);
```

# Passing Data to Pthreads threads...Wrongly

Consider this snippet:

```
int i;
for (i = 0; i < 10; ++i)
  pthread_create(&thread[i], NULL, run, (void*)&i);
```

This is a terrible idea. Why?

# Passing Data to Pthreads threads...Wrongly

Consider this snippet:

```
int i;
for (i = 0; i < 10; ++i)
  pthread_create(&thread[i], NULL, run, (void*)&i);
```

This is a terrible idea. Why?

1. The value of `i` will probably change before the thread executes
2. The memory for `i` may be out of scope, and therefore invalid by the time the thread executes

# Passing Data to Pthreads threads

Correct:

```
int i;
int*
for (i = 0; i < 10; ++i) {
  arg = malloc( sizeof( int ) );
  *arg = i;
  pthread_create(&thread[i], NULL, run, arg);
}
```

# `int*` and `int` are always the same size, right guys?

What about:

```
int i;
for (i = 0; i < 10; ++i)
  pthread_create(&thread[i], NULL, run, (void*)i);

...

void* run(void* arg) {
  int id = (int)arg;
```

This is suggested in the book, but should carry a warning:

# `int*` and `int` are always the same size, right guys?

What about:

```
int i;
for (i = 0; i < 10; ++i)
  pthread_create(&thread[i], NULL, run, (void*)i);

...

void* run(void* arg) {
  int id = (int)arg;
```

This is suggested in the book, but should carry a warning:

- Beware size mismatches between arguments: no guarantee that a pointer is the same size as an int, so your data may overflow.
- Sizes of data types change between systems. For maximum portability, just use pointers you got `from malloc`.

# Detached Threads

*Joinable* threads (the default) wait for someone to call `pthread_join` before they release their resources.

*Detached* threads release their resources when they terminate, without being joined.

```
int pthread_detach(pthread_t thread);
```

**thread**: marks the thread as detached

returns 0 on success, error number otherwise.

Calling `pthread_detach` on an already detached thread results in undefined behaviour.

# Passing Data to C++11 threads

It's easier to get data to threads in C++11:

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

void run(int i) {
  std::cout << "In␣run␣" << i << "\n";
}

int main() {
  for (int i = 0; i < 10; ++i) {
    std::thread t1(run, i);
    t1.detach();.
  }
}
```

# Thread Safety

Make sure the libraries you use are thread-safe.

That means it protects its shared data (more detail later).

“How do I know?”

Well, you could... Read the documentation...?

# READ THE WHAT?!

glibc reentrant functions are also safe.

A program can have more than one thread calling these functions concurrently.

**Example:** `rand_r` versus `rand`.

# Joinable Threads and Detached Threads

Joinable threads hang around until someone joins them.

Detached threads clean up as soon as execution is finished.

It is good practice to detach threads if they are never joined.

And undefined behaviour to try to join a detached thread.

# Getting Data from C++11 threads

In C++ it's harder to get data back.
Use `async` and `future` abstractions:

```
#include <thread>
#include <iostream>
#include <future>

int run() {
  return 42;
}

int main() {
  std::future<int> t1_retval =
                   std::async(std::launch::async, run);
  std::cout << t1_retval.get();
}
```

# Detached Threads: Warning!

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

void* run(void*) {
  printf("In run\n");
}

int main() {
  pthread_t thread;
  pthread_create(&thread, NULL, run, NULL);
  pthread_detach(thread);
  printf("In main\n");
}
```

When I run it, it just prints “In main”, why?

# Detached Threads: Solution to Problem

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

void* run(void*) {
  printf("In run\n");
}

int main() {
  pthread_t thread;
  pthread_create(&thread, NULL, run, NULL);
  pthread_detach(thread);
  printf("In main\n");
  pthread_exit(NULL); // This waits for all detached
                      // threads to terminate
}
```

Make the final call `pthread_exit` if you have any detached threads. (There is no C++11 equivalent.)

# Thread Termination

```
void pthread_exit(void *retval);
```

**retval**: return value passed to function that calls `pthread_join`

start_routine returning is equivalent to calling `pthread_exit` with that return value;

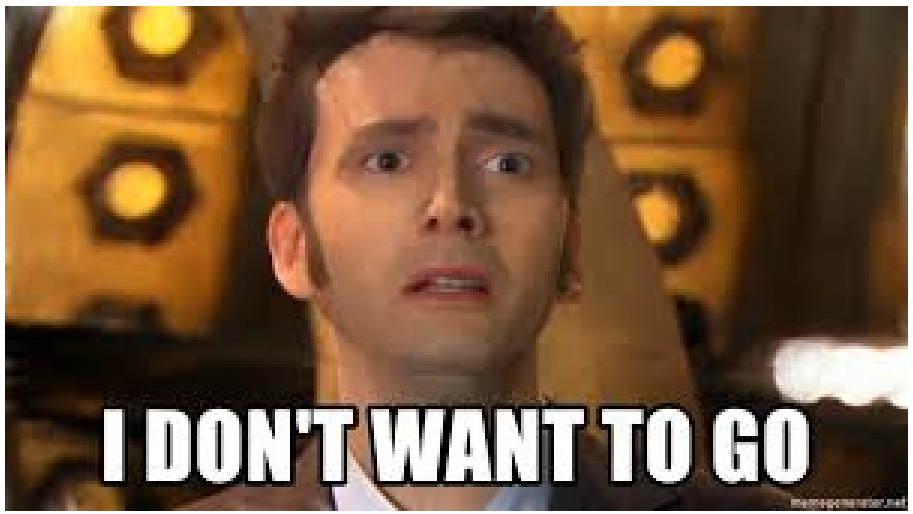
`pthread_exit` is called implicitly when the `start_routine` of a thread returns.

There is no C++11 equivalent.

# Not Now!

Remember cancellation? Asynchronous and Deferred.

Sometimes a thread could die before it has cleaned up.



# Cleanup Handler

The functions for cleaning up are:

```
pthread_cleanup_push( void (*routine)(void*), void *argument ); /* Register
    cleanup handler, with argument */
pthread_cleanup_pop( int execute ); /* Run if execute is non-zero */
```

The push function always needs to be paired with the pop function at the same level in your program (where level is defined by the curly braces).

# Don’t You Forget About Me

Consider the following code:

```
void* do_work( void* argument ) {
  struct job * j = malloc( sizeof( struct job ) );
  /* Do something useful with this structure */
  /* Actual work to do not shown */
  free( j );
  pthread_exit( NULL );
```

# Don’t You Forget About me

```
void cleanup( void* mem ) {
  free( mem );
}

void* do_work( void* argument ) {
  struct job * j = malloc( sizeof( struct job ) );
  pthread_cleanup_push( cleanup, j );
  /* Do something useful with this structure */
  /* Actual work to do not shown */
  free( j );
  pthread_cleanup_pop( 0 ); /* Don't run */
  pthread_exit( NULL );
```

# More fun with pthreads

There are some additional pthread functions we can take a look at:

```
pthread_t pthread_self( void );
int pthread_equal( pthread_t t1, pthread_t t2 );
int pthread_once(pthread_once_t* once_control, void (*init_routine)(void));
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```