Lecture 12 — Threads and Concurrency

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 16, 2019

Thread Cancellation

Thread cancellation is exactly what it sounds like: a running thread will be terminated before it has finished its work.

The thread that we are going to cancel is called the *target*.

Cancellation Type

1. **Asynchronous Cancellation**
2. **Deferred Cancellation**

thread can declare its own cancellation type through the use of the function:

```
pthread_setcanceltype( int type, int *oldtype )
```

`type`: `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`

`oldtype`: previous state, if we care.

Send Cancellation

The pthread command to cancel a thread is `pthread_cancel` and it takes one parameter (the thread identifier).

To check if the current thread has been cancelled, the function call is `pthread_testcancel` which takes no parameters.

It's polite to check this, if it's a risk.

Cancellation Points

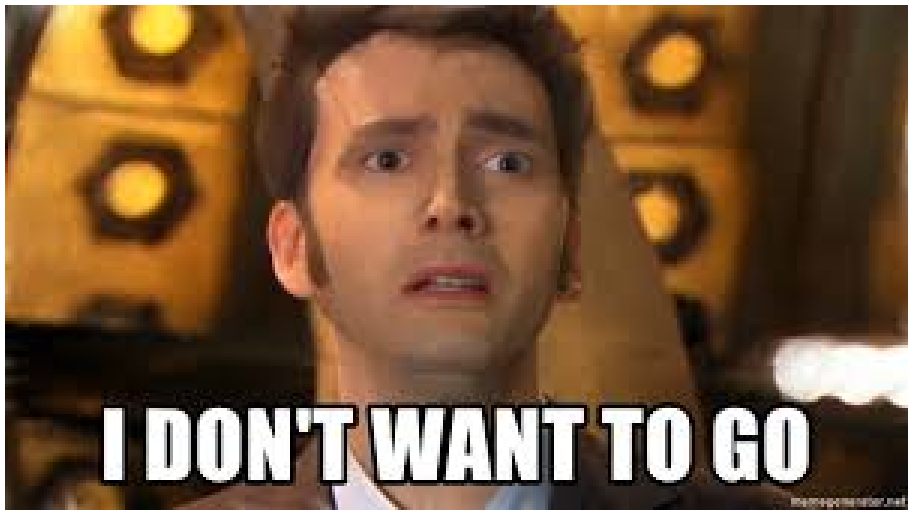
A large number of functions are cancellation points.

That is, the POSIX specification requires there is an implicit check for cancellation when calling one of those functions.

Even more are “potential cancellation points” – maybe, maybe not?

Not Now!

Sometimes a thread could die before it has cleaned up.



This can leave memory allocated, things locked…

Cleanup Handler

We can prevent this with cancellation handlers.

The functions for cleaning up are:

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

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 );
```

Ah! Ah! Ah!

Next, we'l; do an example where we don't use the return value of a thread, but do use attributes.

For the sake of simplicity: we are just going to count!

Slightly Larger Example

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

int sum; /* Shared Data */

void *runner(void *param);

int main( int argc, char **argv ) {
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if ( argc != 2 ) {
    fprintf(stderr,"usage: %s <integer value>\n", argv[0]);
    return -1;
  }
  if ( atoi( argv[1] ) < 0 ) {
    fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
    return -1;
  }

  pthread_attr_init( &attr ); /* set the default attributes */
  pthread_create( &tid, &attr, runner, argv[1] ); /* create the thread */
  pthread_join( tid, NULL );
  printf( "sum = %d\n", sum );
  pthread_exit( NULL );
}
```

Slightly Larger Example

```
void *runner( void *param ) {
  int upper = atoi( param );
  sum = 0;
  for ( int i = 1; i <= upper; i++ ) {
    sum += i;
  }
  pthread_exit( 0 );
}
```

In this example, both threads are sharing the global variable `sum`.

Do we have coordination?

Yes! The parent thread will join the newly-spawned thread (i.e., wait until it is finished) before it tries to print out the value.

If it did not, the parent would print the sum early.

Let's do a different take on that program.

Slightly Larger Example 2

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

int sum = 0;

void* runner( void *param ) {
    int upper = atoi( param );
    for (int i = 1; i <= upper; i++ ) {
        sum += i;
    }
    pthread_exit( 0 );
}
```

Slightly Larger Example 2

```
int main( int argc, char** argv ) {

    pthread_t tid[3];

    if ( argc != 2 ) {
        printf("An integer value is required as an argument.\n");
        return -1;
    }
    if ( atoi( argv[1]) < 0 ) {
        printf( "%d must be >= 0.\n", atoi(argv[1]) );
    }

    for ( int i = 0; i < 3; ++i ) {
        pthread_create( &tid[i], NULL, runner, argv[1] );
    }
    for ( int j = 0; j < 3; ++j ) {
        pthread_join( tid[j], NULL );
    }
    printf( "sum = %d.\n", sum );

    pthread_exit( 0 );
}
```

What happens when we run this program?

Sometimes...

For very small values of the argument, nothing goes wrong.

For a large number we get some strange and inconsistent results. Why?

There are three threads that are modifying `sum`.

But what does “at the same time” mean?

Multiprocessing

Not that long ago, a typical computer had one processor with one core.

It could accordingly do exactly one thing at a time.

1 processor: 1 general purpose processor that executes user processes.

There may be special-purpose processors in the system (RAID controller).

Only one general purpose processor so we call it a uniprocessor system.

Multiprocessing

Now, desktops, laptops, and even cell phones are using multi-core processors.

A quad-core processor may be executing four different instructions from four different threads at the same time.

In theory, multiple processors may mean that we can get more work done in the same amount of (wall clock) time, but this is not a guarantee.

Terminology Note

Terminology note: we often refer to a logical processing unit as a core.

CPU may refer to a physical chip that contains 1+ logical processing units.

As far as the operating system is concerned, it does not much matter if a system has four cores in four physical chips or four cores in one chip.

Either way, there are four units that can execute instructions.

Execution

1 process, 1 thread: it does not matter how many cores are available.
At most one core will be used to execute this task.

If there are multiple processes, each process can execute on a different core.

But what if there are more processes and threads than available cores?

We can hope that the processes get blocked frequently enough and long enough?

Chest Day Best Day

The line at my gym is actually longer 💪

"Can I work in with you?"

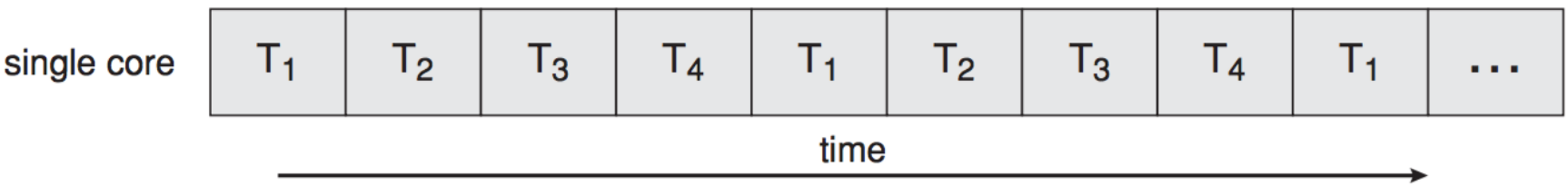
Execution

Switch between the different tasks via a procedure we call time slicing.

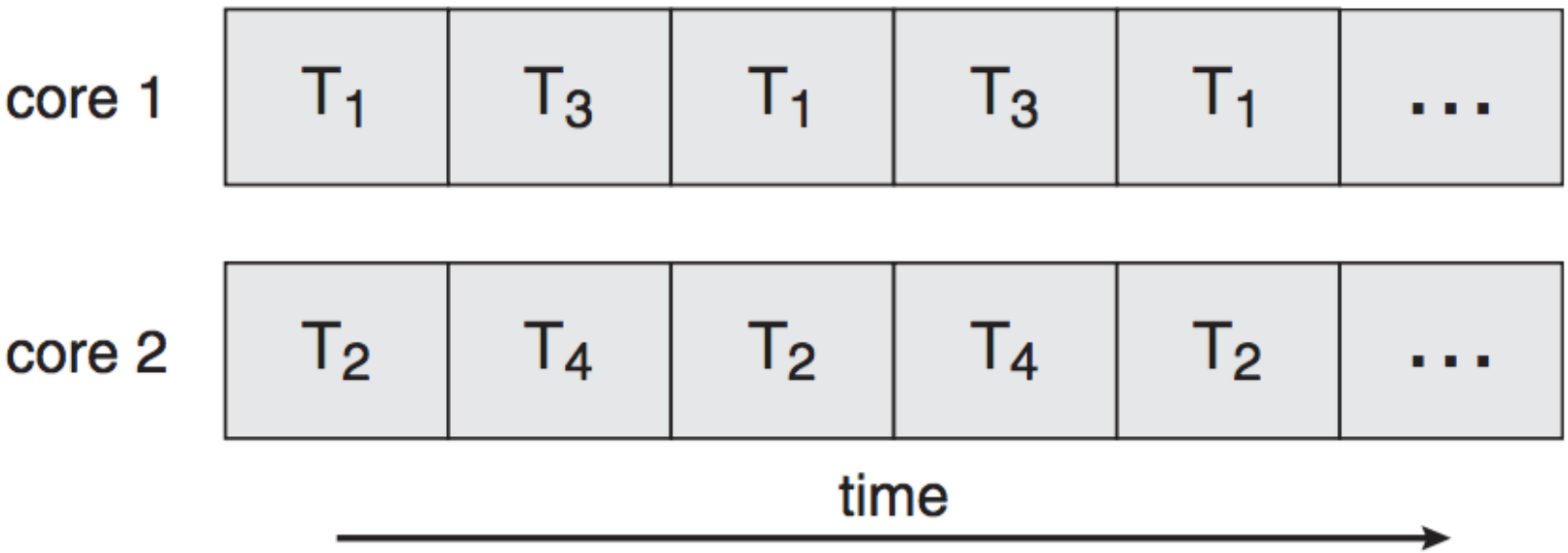
So thread 1 would execute for a designated period, such as 20 ms, then thread 2 for 20 ms, then thread 3 for 20 ms, then back to thread 1 for 20 ms.

To the user, it seems like threads 1, 2, and 3 are being executed in parallel.
20 ms is fast enough that the user does not notice the difference.

Single Core Execution



Time slicing will still occur, if necessary:



Parallelism

Multiple threads at the same time = tasks completed faster?

Parallelism and Speedup

Depends on the nature of the task!

Fully parallelized: $2 \times$ Threads = $2 \times$ Speed

Partially parallelized: $2 \times$ Threads = $(1 < n < 2) \times$ Speed

Cannot be parallelized: $2 \times$ Threads = $1 \times$ Speed

Heard, Chef!

Speedup Example

Suppose: a task that can be executed in 5 s, containing a parallelizable loop.

Initialization and recombination code in this routine requires 400 ms.

So with one processor executing, it would take about 4.6 s to execute the loop.

Split it up and execute on two processors: about 2.3 s to execute the loop.

Add to that the setup and cleanup time of 0.4 s and we get a total time of 2.7 s.

Completing the task in 2.7 s rather than 5 s represents a speedup of about 46%.

Amdahl's Law

Gene Amdahl came up with a formula for the general case of how much faster a task can be completed based on how many processors we have available.

Let us define S as the portion of the application that must be performed serially and N as the number of processing cores available.

Amdahl's Law:

$$\text{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$$

Amdahl's Law

Take the limit as $N \rightarrow$ infinity and you will find the speedup converges to $\frac{1}{S}$.

The limiting factor on how much additional processors help is the size of S.

Matches our intuition of how it should work.

Amdahl's Law on the 5 s Task

Applying this formula to the example from earlier:

Processors	Run Time (s)
1	5
2	2.7
4	1.55
8	0.975
16	0.6875
32	0.54375
64	0.471875
128	0.4359375

Observations on the 5 s Task

1. Diminishing returns as we add more processors.

2. Converges on 0.4 s.

The most we could speed up this code is by a factor of $\frac{5}{0.4} \approx 12.5$.

But that would require infinite processors (and therefore infinite money).

Merge Sort Example

Recall from data structures and algorithms the concept of merge sort.

This is a divide-and-conquer algorithm like binary search.

Split the array of values up into smaller pieces, sort those, and then merge the smaller pieces together to have sorted data.

Merge Sort Example