2023 Fall ECE 344: Operating Systems
Lecture 16 1.0.0

Threads

Jon Eyolfson
2023 Fall

Aside: Concurrency and Parallelism Aren't the Same

Concurrency

- Switching between two or more things (can you get interrupted)
  - Goal: make progress on multiple things

Parallelism

- Running two or more things at the same time (are they independent)
  - Goal: run as fast as possible

A Real Life Situation of Concurrency and Parallelism

You're sitting at a table for dinner, and you can:

- Eat
- Drink
- Talk
- Gesture

You're so hungry that if you start eating you won't stop until you finish

Which tasks can and can't be done concurrently, and in parallel?

Choose Any Two Tasks in the Real Life Example

You can't eat and talk (or drink) at the same time, and you can't switch
Not parallel and not concurrent

You could eat and gesture at the same time, but you can't switch
Parallel and not concurrent

You can't drink and talk at the same time, and you could switch
Not parallel and concurrent

You can talk (or drink) and gesture at the same time, and you could switch
Parallel and concurrent

Threads are Like Processes with Shared Memory

The same principle as a process, except by default they share memory
They have their own registers, program counter, and stack

They have the same address space, so changes appear in each thread

You need to explicitly state if any memory is specific to a thread (TLS)

One Process Can have Multiple Threads

By default, a process just executes code in its own address space

Threads allow multiple executions in the same address space

They're lighter weight and less expensive to create than processes
They share code, data, file descriptors, etc.

Assuming One CPU, Threads Can Express Concurrency

A process can appear like it's executing in multiple locations at once
    However, the OS is just context switching within a process

It may be easier to program concurrently
    e.g., handle a web request in a new thread

```
while (true) {
  struct request *req = get_request();
  create_thread(process_request, req);
}
```

Threads are Lighter Weight than Processes

Process	Thread
Independent code / data / heap	Shared code / data / heap
Independent execution	Must live within an executing process
Has its own stack and registers	Has its own stack and registers
Expensive creation and context switching	Cheap creation and context switching
Completely removed from OS on exit	Stack removed from process on exit

When a process dies, all threads within it die as well!

We'll be Using POSIX Threads

For Windows, there's a Win32 thread, but we're going to use *UNIX threads

`#include <pthread.h>` — in your source file

`-pthread` — compile and link the pthread library

All the pthread functions have documentation in the `man` pages

You Create Threads with `pthread_create`

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

thread	creates a handle to a thread at pointer location
attr	thread attributes (NULL for defaults, more details later)
start_routine	function to start execution
arg	value to pass to start_routine

returns 0 on success, error number otherwise (contents of *thread are undefined)

Creating Threads is a Bit Different than Processes

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

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

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

What are some differences? Are we missing anything?

The `wait` Equivalent for Threads — Join

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

thread	wait for this thread to terminate (thread must be joinable)
retval	stores exit status of thread (set by `pthread_exit`) to the location pointed by *retval. If cancelled returns `PTHREAD_CANCELED`. `NULL` is ignored.

returns 0 on success, error number otherwise

Only call this one time per thread!

Multiple calls on the same thread leads to undefined behavior

Previous Example that Waits Properly

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

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

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

Now we joined, the thread's resources are cleaned up

Ending a Thread Early (Think of `exit`)

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

retval return value passed to function that calls `pthread_join`

Note: `start_routine` returning is equivalent of calling `pthread_exit`
Think of the difference between returning from main and `exit`

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

Detached Threads

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

Detached threads release their resources when they terminate

```
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 is undefined behavior

Detached Threads Aren’t Joined

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

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

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

This code just prints “In main”, why?

`pthread_exit` in main Waits for All Detached Threads to Finish

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

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

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

This code now works as expected

You Can Use Attributes To Get/Set Thread Variables

```
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 should show a stack size of 8 MiB (on most Linux systems)

You can also set a thread state to joinable

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

Let's Compare Creating Threads to Processes

See: `lectures/16-threads/multiple-thread-example.c`

Compare this to: `lectures/04-process-creation/multiple-fork-example.c`

Technically, how should we (very slightly) improve the thread example?

Threads Enable Concurrency

We explored threads, and related them to something we already know (processes)

- Threads are lighter weight, and share memory by default
- Each process can have multiple threads (but just one at the start)