CSSE2310 — 8.1

Threads and Synchronization

Main challenges

- Process management functions are weird.
- Debugging at a distance
- Debugging multiple processes simultaneously
- Being discplined in the way you code and test.

Threads

Not for Ass3

- Threads and synchronization is for Ass4.
- Do not use this material in Ass3!

Working in a process

To run code, we need:

- A process container
  - Address space
  - open files
  - ...
- Code / Instuctions
  - “text segment”
- Global variables / constants
- Heap
  - Dynamic storage
- Stack
  - Function local variables
- CPU and its registers

Multiple workers?

Suppose we want to have multiple workers doing things? `fork()`?

- Open files are shared (see Week 7.1).
- Everything else is separate.
- Needed if you want to have the new worker running a different program.

Different processes can cooperate on tasks (especially since they can start off with the same knowledge).

- But communicating results is more complicated.
  - Pipes?
  - "Network connection"? Similar complexity to pipes.
  - Shared memory?
  - ...
- But processes are safe from each other

Threads?

A thread is a worker in a process[1].
So every process has at least one thread (even if it doesn't explicitly use a thread library).

- Likely you will only interact with these via libraries.
- Different systems may have distinctions between "Threads / Light weight processes / ..."
  - We will avoid talking about these
- Threads can be classified as "native" or "green"
  - We'll get to this later

[1] A thread of execution through the program

Multiple threads

If we want multiple threads operating in the same process, we would want them to be able to share information easily.

Item	Shared?
Process	Yes
Code	Yes
Global variables	Yes
Heap	Yes
Stack	No
CPU + registers	No

Threads and memory

Any thread running in a process can interact with any variable used by any other thread if they know where it is (ie they have a pointer to it).

The kernel won't stop this because, if a page is valid for one thread, it is in the page table and so valid for all of the other threads.

Thread implementations

Two main approaches here:

- "native" or "kernel" threads[2]
  - Threads are known by the kernel.
  - Threads can be scheduled and run independently of each other
    - Scheduling is out of scope for 2310, but gang scheduling is an interesting example.
- "green" threads
  - Kernel just sees a single threaded process
  - User space library switches saves registers and modifies program counter to switch between activities
  - Not actually using multiple cpu cores
  - A kernel call in one means you can't switch until you get back to user mode
    - Blocking I/O?
  - eg Python threading

[2] We are assuming these for 2310

Why threads?

Why do we teach threads? Are they actually needed?
Threads are:

1. Things you should know about[3]
2. A way around blocking I/O
   - I need to listen to 5 network connections[4] and I don't know what order they will return in.
3. A way to use more than one core worth of CPU power.

[3] ie a well trained programmer should know about them.
[4] Blocking reads

2 Blocking I/O

Situation:
Need to interact with multiple I/O `FILE*/fds`

```
while (!done) {
    read from A and process
    read from B and process
    read from C and process
}
```

Problem: If input is available from C but not A, we won't get there.

2 Blocking I/O

Threading approach:

```
... interact(FILE* src) {
    while (...  ) {
        read from src
        process input
    }
}

...
   run interact(A) in thread     // not actual syntax
   run interact(B) in thread
   run interact(C) in thread
```

Each thread blocks on its own input source and doesn't interact with others.

Non-threaded alternative?

Non-blocking I/O calls exist in C, so you could do this:

```
while (!done) {
    if (have_input_on(A)) {
        read from A and process
    }
    if (have_input_on(B)) {
        read from A and process
    }
    if (have_input_on(C)) {
        read from A and process
    }
}
```

Problem: busy waiting. This loop may run hundreds(?) of times before any input arrives.

Event driven programming

```
while (!done) {
    sleep_until_input_available_on(A, B, C)
    process input
}
```

These sorts of calls exist and deal with the busy wait problem.
One core may be plenty — since most of the time is spent waiting.
So why aren't we using them?

- You need to work with `fds` and `read()`. No `FILE*` niceties.
- It is trickier to keep track of where you are up to with multiple tasks (instead of each task having its own thread context).

More power

In applications which do not do much I/O (eg scientific computing), non-blocking operations are not going to help. There you either need multiple processes or multiple threads to access more cores.

pthreads

C has no standard threading library.

- We are using the POSIX-thread API (pthreads).
- Implementations exist for most OS (including windows)
- pthreads wrap OS threads calls.

From now on, any discussion of "threads" means "pthread threads".

pthreads

- Threads have no parent child relationships.
- Any thread can "join" (ie wait on) any other thread in the same process.
- Threads can't interact with threads in other processes.
- Any thread calling `exit()` will end the whole process including all other threads.
- `fork()`?
  - Yes that is an interesting question.
- Signal handlers?
  - Yes ... later

thread functions

- Each thread is created to run a function.
- When that function ends / returns, so does the thread[5]
- The function needs to have the correct signature.

```
void* foo(void*) {
    // do thread things
    return (void*)0;
}
```

So the function can

- Accept any parameter (as long as you can express it as a `void*`)
- Send back anything to whoever `join()`s on it (as long as it's a `void*`)

[5] As far as you know.

Hello

See `thread1.c`

Note: to compile a program using pthreads you need to add `-pthread` to the command line[6]

```
joel@sage:~/csse2310_2020_1/lecture/code$ cat thread1.c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h> // for sleep()

void* hello(void* v) {
    char* s = (char*)v;
    printf("Hello %s\n", s);
    return 0;
}

int main(int argc, char** argv) {
    pthread_t tid;
    pthread_create(&tid, 0, hello, "Larry");
    pthread_create(&tid, 0, hello, "Curly");
    pthread_create(&tid, 0, hello, "Moe");
    sleep(2);
    return 0;
}
```

```
joel@sage:~/csse2310_2020_1/lecture/code$ gcc -pthread thread1.c
joel@sage:~/csse2310_2020_1/lecture/code$ ./a.out
Hello Larry
Hello Curly
Hello Moe
joel@sage:~/csse2310_2020_1/lecture/code$ ./a.out
Hello Curly
Hello Larry
Hello Moe
```

Hello

`man pthread_create`

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

- the id of the new thread is stored at the first argument
    - if the call is sucessful — see return value.
- we'll ignore the second argument for now
- the function we want to call
- the argument we want to pass it

Hello

What is the call to `sleep()` doing there?

See `thread2.c`

We can end an individual thread using `pthread_exit()`;

See `thread3.c`

```
#include <stdio.h>
#include <unistd.h> // for sleep()

void* hello(void* v) {
    char* s = (char*)v;
    printf("Hello %s\n", s);
    return 0;
}

int main(int argc, char** argv) {
    pthread_t tid;
    pthread_create(&tid, 0, hello, "Larry");
    pthread_create(&tid, 0, hello, "Curly");
    pthread_create(&tid, 0, hello, "Moe");
    return 0;
}
joel@sage:~/csse2310_2020_1/lecture/code$ gcc -pthread thread2.c
joel@sage:~/csse2310_2020_1/lecture/code$ ./a.out
joel@sage:~/csse2310_2020_1/lecture/code$ ./a.out
joel@sage:~/csse2310_2020_1/lecture/code$ ./a.out
joel@sage:~/csse2310_2020_1/lecture/code$
```

```
int main(int argc, char** argv) {
    pthread_t tid;
    pthread_create(&tid, 0, hello, "Larry");
    pthread_create(&tid, 0, hello, "Curly");
    pthread_create(&tid, 0, hello, "Moe");
    pthread_exit((void*)0);
}
joel@sage:~/csse2310_2020_1/lecture/code$ gcc -pthread thread3.c
joel@sage:~/csse2310_2020_1/lecture/code$ gcc -pthread -Wall -pedantic -std=g
nu99 thread3.c
joel@sage:~/csse2310_2020_1/lecture/code$ ./a.out
Hello Larry
Hello Curly
Hello Moe
```

Passing values into threads

So far we've passed in strings (`char*`).

- `char*` → `void*` → `char*`

What about `int`?

See `thread4.c`

Why bother with `malloc()`ing?

See `thread5.c`

This is an example of a "race condition".

```
#include <stdio.h>
#include <unistd.h> // for sleep()
#include <stdlib.h> // malloc and free

void* hello(void* v) {
    int value = *(int*)v;
    free(v);
    printf("Hello %d\n", value);
    return (void*)0;
}

int main(int argc, char** argv) {
    pthread_t tid;
    for (int i=0; i<5; ++i) {
        int* val = (int*) malloc(sizeof(int));
        *val = i;
        pthread_create(&tid, 0, hello, val);
    }
    pthread_exit((void*)0);
}
```

```
joel@sage
Hello 2
Hello 3
Hello 1
Hello 4
Hello 0
joel@sage
Hello 2
Hello 1
Hello 0
Hello 4
Hello 3
joel@sage
Hello 1
Hello 0
Hello 2
Hello 3
Hello 4
```

```
Hello 2
Hello 2
Hello 3
Hello 4
Hello 4
joel@sage
Hello 1
Hello 2
Hello 4
Hello 4
Hello 4
```

The Univ

```
void* hello(void* v) {
    int value = *(int*)v;
    printf("Hello %d\n", value);
    return (void*)0;
}

int main(int argc, char** argv) {
    pthread_t tid;
    for (int i=0; i<5; ++i) {
        int val = i;
        pthread_create(&tid, 0, hello, &val);
    }
    pthread_exit((void*)0);
}
```