CSSE2310

...

Concurrency and Threads

Week 11, Sem 1, 2020

Week 11

- Assignment 4 is being prepared for release
  - Involves threads, mutexes and semaphores -> this tute!
  - Involved network programming
- Useful resources
  - Lectures (concurrency and network programming)
  - Modularity PDFs on Blackboard
  - Piazza (see if your question has been asked already)
- Today’s content
  - Thread diagrams and concurrency material
  - Running through an example concurrent program and making it thread-safe

Threads: A Refresher

- You can create a thread using pthread_create:

```
int pthread_create(pthread_t* tid, const pthread_attr_t* attr,
                   void* (*start)(void*), void* arg)
```

  - First argument is a pointer to store the ID of the created thread
  - Second argument is a pointer to a struct containing thread attribute settings (usually NULL)
  - Third argument is a pointer to the function which you want the thread to start executing at
  - Last argument is the void* to the data you want to pass as parameters to the thread function

Exercise: Thread Diagram Question 1

- What diagram is produced by the following code? What is the execution time?

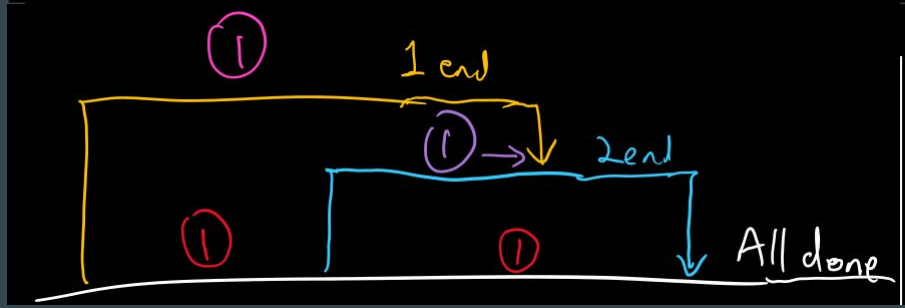
```
int main(int argc, char** argv) {

    pthread_t tId[2];
    pthread_create(&tId[0], NULL, thread1,
        NULL);
    sleep(1);
    pthread_create(&tId[1], NULL, thread2,
        (void*) tId[0]);
    sleep(1);
    pthread_join(tId[1], NULL);
    printf("All done\n");
    return 0;
}
```

```
void* thread1(void* arg) {
    sleep(1);
    printf("Thread 1 ending\n");
    return NULL;

}

void* thread2(void* arg) {
    sleep(1);
    pthread_join((pthread_t)  arg, NULL);
    printf("Thread 2 ending\n");
    return NULL;
}
```

Exercise: Thread Diagram Question 1 Solution

```c
int main() {

    pthread_t tId[2];
    pthread_create(&tId[0], NULL,
thread1,
        NULL);
    sleep(1);
    pthread_create(&tId[1], NULL,
thread2,
        (void*) tId[0]);

    sleep(1);
    pthread_join(tId[1], NULL);
    printf("All done\n");
    return 0;
}
```

```c
void* thread1(void* arg) {
    sleep(1);
    printf("Thread 1 ending\n");
    return NULL;

}

void* thread2(void* arg) {
    sleep(1);
    pthread_join((pthread_t)  arg, NULL);
    printf("Thread 2 ending\n");
    return NULL;

}
```



The pink sleep and first main sleep will run in parallel, so is expected to take 1 second in total.

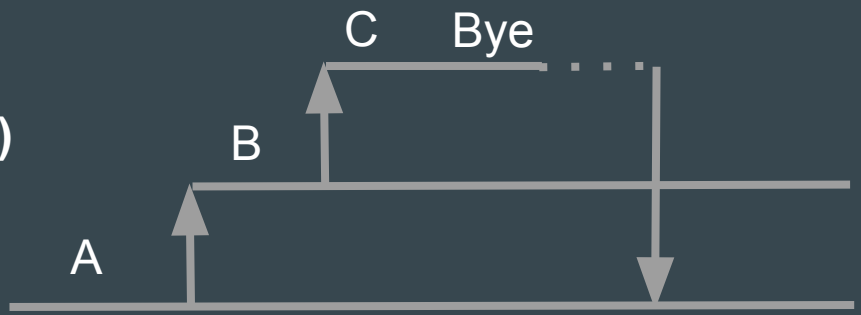
The purple sleep and the second main sleep will also run in parallel, so is expected to take 1 second in total.

Therefore we expect the program to take 2s.

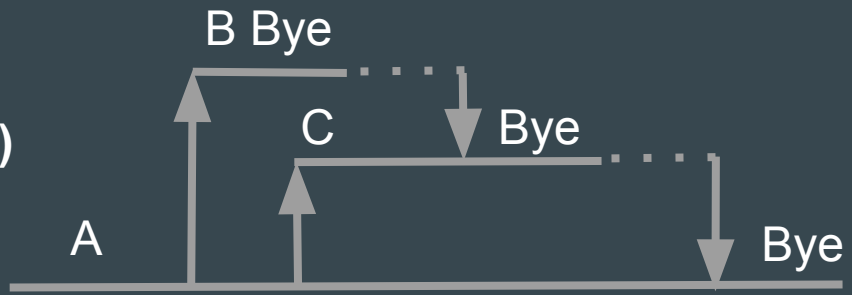
Exercise: Thread Diagram Question 2

- Which of the following are possible with threads?

a)



b)

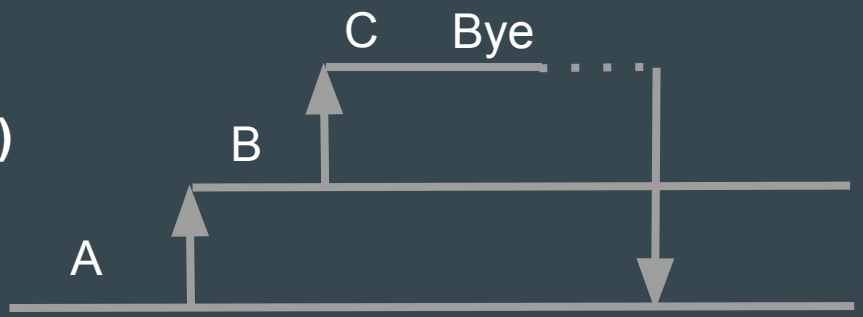


c)

Exercise: Thread Diagram Question 2 Solution

- Which of the following are possible with threads?

a)

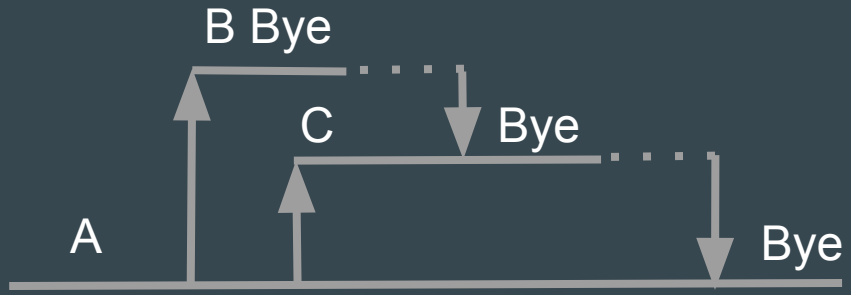


This is possible with threads, however we need a global variable for A to know C's thread id so it can join with it.

Exercise: Thread Diagram Question 2 Solution

- Which of the following are possible with threads?

b)



This is also possible with threads, see the thread diagram example.

Exercise: Thread Diagram Question 2 Solution

- Which of the following are possible with threads?

c)

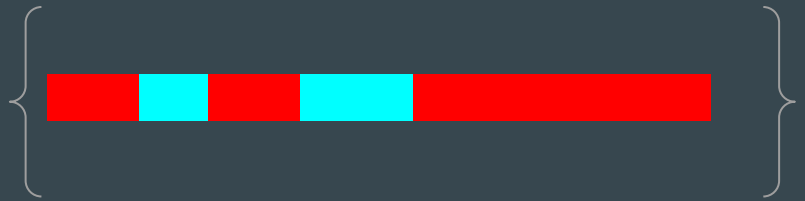
This is also possible with threads, this was covered in the concurrency lectures and uses pthread_exit.

Concurrency

- Two actors* are concurrent if the could run at the same time
  - This doesn't necessarily mean that they run in parallel
  - It just means that there isn't a guaranteed ordering in how they run

- Example: Concurrent actors which aren't necessarily run in parallel

2 concurrent actors using 2 CPUs



2 concurrent actors using 1 CPU

*processes, threads (in other environments these could be tasks etc.)

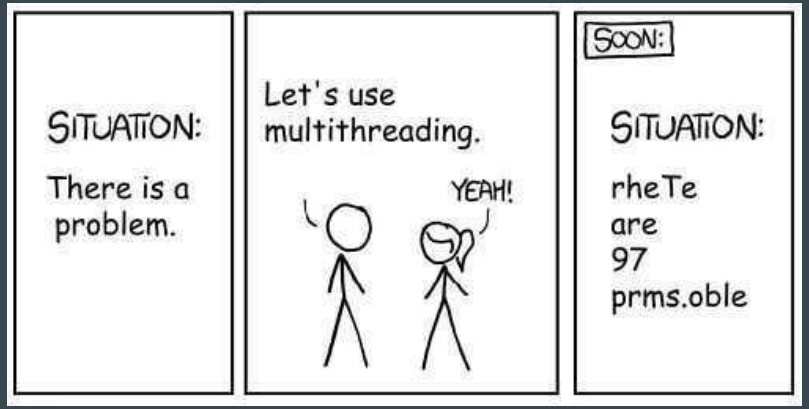
Concurrency: Pros and Cons

Pros

- Can be faster
- More efficient CPU utilisation
- Lets you do more than one thing at a time. For example:
  - More than one visitor can view your website at once
  - More than one person can be logged into moss at once
  - An automated car can look at traffic lights while staying in its lane

Cons

- Race conditions
- Memory corruption… but worse and harder to debug than normal
- Non-deterministic bugs
- Debuggers hiding bugs



Concurrency: Race Conditions

- When A should happen before B, but the order gets confused

- Fixed by synchronising actors
  - Ensure that B doesn't start until A has finished
  - A must tell B when to start
  - Read / write from a pipe
  - Lock / unlock a mutex
  - Wait / post a semaphore
  - etc.

Concurrency: Memory Corruption

- One thread writes to memory while the other one tries to access it (read or write)
- This was avoided in assignment 3 due to **memory isolation**
  - You can't write to the same memory if that memory isn't shared
- Can be prevented by:
  - Using read-only memory (note that calling `free(3)` is a type of write)
  - Giving memory to another thread and no longer accessing it
  - Only accessing memory with a single thread at a time (via mutexes and semaphores)

Concurrency: Primitives

- Mutexes
  - Can only be “locked” by one thread at a time
  - If attached to a block of code, ensures only one thread runs that code at any one time (i.e. provides mutual exclusion)

- Semaphores
  - Based off the concept of a counter
  - Once the counter reaches zero, threads trying to decrement (wait) it are put to sleep
  - When a process increments the counter (post), one of the waiting processes will wake up

Concurrency: Deadlocks

- Deadlocks occur when different actors are waiting on eachother in a **cycle**
  - A waiting for B, B waiting for C, C waiting for A
  - A waiting for B, B waiting for A
- Fix one: don't have cycles
  - Draw a diagram of which actor waits for which and ensure it is cycle free
  - Not always applicable, but reducing loops can simplify things
- Fix two: ensure at least one actor writes (and flushes) before they read
  - Draw some diagrams of communication over time, perform some tests
- When thinking about deadlocks, please remember
  - Reading is a form of waiting if there is something on the other end
  - Other processes, as well as threads, can be actors and can deadlock

Exercise: Making an I/O Program Thread Safe

- We've got a program that does I/O using multiple threads
- There are two "writing" threads:
  - One reads from stdin and sends the read data to the channel
  - One reads from a file and sends the read data to the channel
- And one "reading" thread:
  - Reads from the channel, and prints to stdout

- The channel in this case is a queue which is used as a means of communication between threads (i.e. it is essentially a fancy pipe - threads can write to it, and other threads can read from it)

Exercise: Making an I/O Program Thread Safe

- The code is available on Blackboard under Learning Resources -> Prac Slides
- The code contains:
  - A makefile (notice -g and -pthread)
  - main.c (logic for each of the threads)
  - queue.c/h (a FIFO queue implementation)
  - channel.c/h (wrappers around the queue to make a thread-safe channel)

- Running ./channel seems to work correctly
- Try to stress test it: `cat /dev/urandom | base64 | fold -w 60 | ./channel`
- It will break eventually if you leave it running
  - Memory corruption is occurring due to the channel not being thread-safe

Exercise: Making an I/O Program Thread Safe

- Step 1: Add a mutex to the channel
- Mutexes are built for this scenario. There is a piece of data (the queue) that you want to make sure no two threads write to at once. You fix this by wrapping each access to that data in a lock/unlock statement

```
read_channel:

    pthread_mutex_lock(lock);
    output = read_queue();
    pthread_mutex_unlock(lock);
```

```
write_channel:

    pthread_mutex_lock(lock);
    output = write_queue();
    pthread_mutex_unlock(lock);
```

Exercise: Making an I/O Program Thread Safe

- We should now be passing the stress test
- In a separate terminal, try:
  - `ps -L -o pid,uid,cmd,pcpu  # Prints pid, uid, cmd (normal output) AND %CPU used for each thread`

- Why is one of the threads taking up 100% CPU?
- Which thread is it?

Exercise: Making an I/O Program Thread Safe

- The writer thread is causing the issue
  - It is doing what it known as “busy polling”
  - If nothing is available to be read, it will keep looping and burning CPU cycles
    - Our program currently calls read_queue more than 10 million times a second on moss
- We need a way to put this thread to sleep until information is ready to be output

Exercise: Making an I/O Program Thread Safe

- Enter semaphores
  - We use a semaphore in this case to keep track of how many elements are in the queue
  - The semaphore starts at 0
  - We increment it (post) each time we write to the channel
  - We decrement it (wait) each time we read from the channel

```
read_channel:

    sem_wait(signal);

    pthread_mutex_lock(lock);
    output = read_queue();
    pthread_mutex_unlock(lock);
```

```
write_channel:

    pthread_mutex_lock(lock);
    output = write_queue();
    pthread_mutex_unlock(lock);

    sem_post(signal);
```

Things to Remember

- No sleeping (or blocking) with a locked mutex
- If one thread is writing, no other thread is nearby
- Just because your code works once doesn't mean it will work every time