COSC 407
Intro to Parallel Computing

Topic 4 – POSIX Threads

1

Qutline

Previous pre-recorded lecture (Students' led Q/As):

More on C programming:

- Intro to parallel computing
- Intro to POSIX Threads

Today's topics:

- POSIX Threads – key concepts

Next Lecture:

- Intro to OpenMP

2

POSIX Threads

- Key things
  - Parallel concepts with pthreads
  - Thread management
  - Synchronization
  - Mutexes
  - Condition variables

3

POSIX Threads

- Threads associated with a process share resources
- Each thread has their own
  - Stack (private variables)
  - Program Counter (PC)
  - Registers
  - Thread ID

4

Creating Threads

- To create a thread a `pthread_create` function is used that needs four arguments
  - Thread variable (holds the reference to thread)
  - Thread attribute (specifies the minimum stack size to be used)
  - Function to call when thread starts
  - Arguments to pass to function

```
pthread_t        a_thread;
pthread_attr_t   a_thread_attribute;  //pthread_attr_default
void thread_function(void *argument);
char             *some_argument;

pthread_create( &a_thread,
                a_thread_attribute,
                (void *)&thread_function,
                (void *)&some_argument
                );
```


5

Creating Threads

- Threads begin their execution at the function specified in `pthread_create`
- For each thread, we will need to create a thread variable
- Let's consider an example with two threads that will print out a message
  - One thread prints "**Hello** "
  - One thread prints "**World!**'

6

Example #1

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

void* say_something(void *ptr)
{
printf("%s ", (char*)ptr);
return NULL;
}

int main()
{
pthread_t thread_1, thread_2;

char *msg1 = "Hello ";
char *msg2 = "World!";

pthread_create( &thread_1, NULL, say_something, msg1);
pthread_create( &thread_2, NULL, say_something, msg2);
printf("Done!");
fflush(stdout);
exit(0);
}
```

1 - What does this code do?
2 – What is the issue?
- Run the code multiple times…

7

Challenges with Example #1

- Threads execute concurrently
  - There is no guarantee that the first thread reaches the `printf` function prior to the second thread
  - Output could be
    - "World Hello"
    - "Hello World“
    - "World “ or " Hello"
    - Nothing…. Why?
- Call to exit made by the parent thread in the main block
  - If the parent thread executes the exit call prior to either of the child threads executing `printf`, no output will be generated at all.
    - The exit function exits the process (releases the task) thus terminating all threads
  - Any thread, parent or child, who calls exit can terminate all the other threads along with the process

8

Races

- A **race** condition…
  - Exiting before threads exit
  - Time for threads to complete
- We want each child thread to finish before the parent thread
  - Insert a delay in the parent that will give the children time to reach `printf`
  - Ensure that the first child thread reaches `printf` before the second
  - Insert a delay prior to the `pthread_create` call that creates the second thread

Bad Idea… but let's have a look with example 2

9

Example #2

```
int main()
{
    pthread_t thread_1, thread_2;

    char *msg1 = "Hello ";
    char *msg2 = "World!";

    pthread_create( &thread_1, NULL, say_something, msg1);
    sleep(1);

    pthread_create( &thread_2, NULL, say_something, msg2);
    sleep(1);

    printf("Done!");
    fflush(stdout);
    exit(0);

}
```


10

Issues with Example #2

- This code doesn't really meet the objectives
  - Not safe to relay on timing delays for synchronization
  - Race condition is still present
  - Sleep function impacts entire process
    - Everything is stalled!
    - Our program just takes longer to run….

11

Allowing Threads to Terminate

- The `pthread_join()` function waits for the thread specified by thread to terminate
  - If that thread has already terminated, then `pthread_join()` returns immediately

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

https://man7.org/linux/man-pages/man3/pthread_join.3.html

12

Example #3

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

void* say_something(void *ptr)
{
printf("%s ", (char*)ptr);
return NULL;
};

int main()
{
pthread_t thread_1, thread_2;

char *msg1 = "Hello ";
char *msg2 = "World!";

pthread_create( &thread_1, NULL, say_something, msg1);
pthread_create( &thread_2, NULL, say_something, msg2);

pthread_join(thread_2, NULL);
pthread_join(thread_1, NULL);

printf("Done!");
fflush(stdout);
exit(0);
}
```

- Problems?
  - While this happily waits, there is no control over who finishes first!
  - How can we sync this so that it works properly??

13

The MUTEX aka Mutual Exclusion

- Threads are lacking Synchronization
  - i.e. who gets to run/access things first?
- Thread synchronization can be achieved using a **Mutex (Mutual Exclusion)**
  - Only one thread at a time can have access to a shared resource
- A Mutex is a lock that is set before using a shared resource and release after using it
  - When the lock is set, no other thread can access the locked region of code (critical code section)
  - Ensures synchronized access of shared resources in the code
  - Can be used to protect access to key resources

14

The MUTEX aka Mutual Exclusion

- A mutex is initialized and then a lock is achieved
  - Initializes a mutex

```
pthread_mutex_init(&lock, NULL);
```

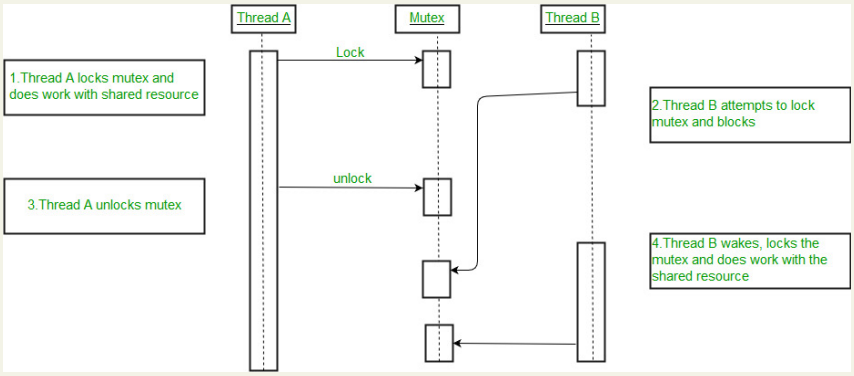
  - Achieve/test lock of the critical code section
  - Mutex needs to be released when done

```
void* say_something(void *ptr)
{
    //this now becomes critical section!
    pthread_mutex_lock(&lock);
    printf("%s ", (char*)ptr);
    pthread_mutex_unlock(&lock);
    pthread_exit(0);
}
```

  - Mutexes need to be deleted (destroyed) when done with them

15

Mutexes



https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/

16

Example #4

```
//get a lock
pthread_mutex_t lock;  initialize globally

void* say_something(void *ptr)
{
     pthread_mutex_lock(&lock);//this now becomes critical section!
     printf("%s ", (char*)ptr);
     pthread_mutex_unlock(&lock);
     pthread_exit(0);
}

int main()
{
     pthread_t thread_1, thread_2;

     char *msg1 = "Hello ";
     char *msg2 = "World!";

     //create the lock -> error checking?
     pthread_mutex_init(&lock, NULL);

     pthread_create( &thread_1, NULL, say_something, msg1);
     pthread_create( &thread_2, NULL, say_something, msg2);

     pthread_join(thread_1, NULL);
     pthread_join(thread_2, NULL);
     printf("Done!");
     fflush(stdout);

     pthread_mutex_destroy(&lock);
     exit(0);
}
```

- Still problems?
  - The thread that gets the lock first, gets to go first

17

Conditions

- There are many cases where a thread wishes to check whether a condition is true before continuing its execution
- Not a variable but are used with an associated mutex
- A condition variable is an explicit queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition);
- When it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue

```
pthread_cond_wait(&cond1, &lock);
```

If the condition is not true, release lock and wait

```
pthread_cond_signal(&cond1);
```

wake up threads waiting for the condition variable.

18

Example 5

```
void* say_something(void *ptr)
{
    pthread_mutex_lock(&lock);//this now becomes critical section!

    //check on some condition - if it is hello, wait for world....
    if (strcmp("World!",(char*)ptr) == 0)
    {
        printf("Waiting on condition variable cond1\n");
        if (done == 0) //only wait in the event that you need to…
                pthread_cond_wait(&cond1, &lock);
    }
    else
    {
        printf("Signaling condition variable cond1\n");
        done == 1;
        pthread_cond_signal(&cond1);
    }
    printf("%s ", (char*)ptr);
    pthread_mutex_unlock(&lock);
    pthread_exit(0);
}
```


19

Key Functions

- **pthread_create**
  - Create a thread
- **pthread_join**
  - Wait for thread to compete
- **pthread_mutex_init**
  - Create a lock
- **pthread_mutex_lock/pthread_mutex_unlock**
  - Lock and unlock a mutex (if available)
- **pthread_cond_wait**
  - Check on condition
- **pthread_cond_signal**
  - Signal threads waiting on condition

20

Conclusion/Up Next

- What we covered today (review key concepts):
  - POSIX Threads – key concepts
  - There is a lot of detail here
    - Gives a basic Idea of challenges
    - Will expand on this with OpenMP
- Next Lecture:
  - OpenMP

21

Homework

- Please review
  - POSIX Threads Programming
    - https://hpc-tutorials.llnl.gov/posix/ (sections 1 – 8)
  - Additional resources on Canvas
  - Have a look at the example code in the course repo (link on Canvas)

22