Parallel and Concurrent Programming CS3S666

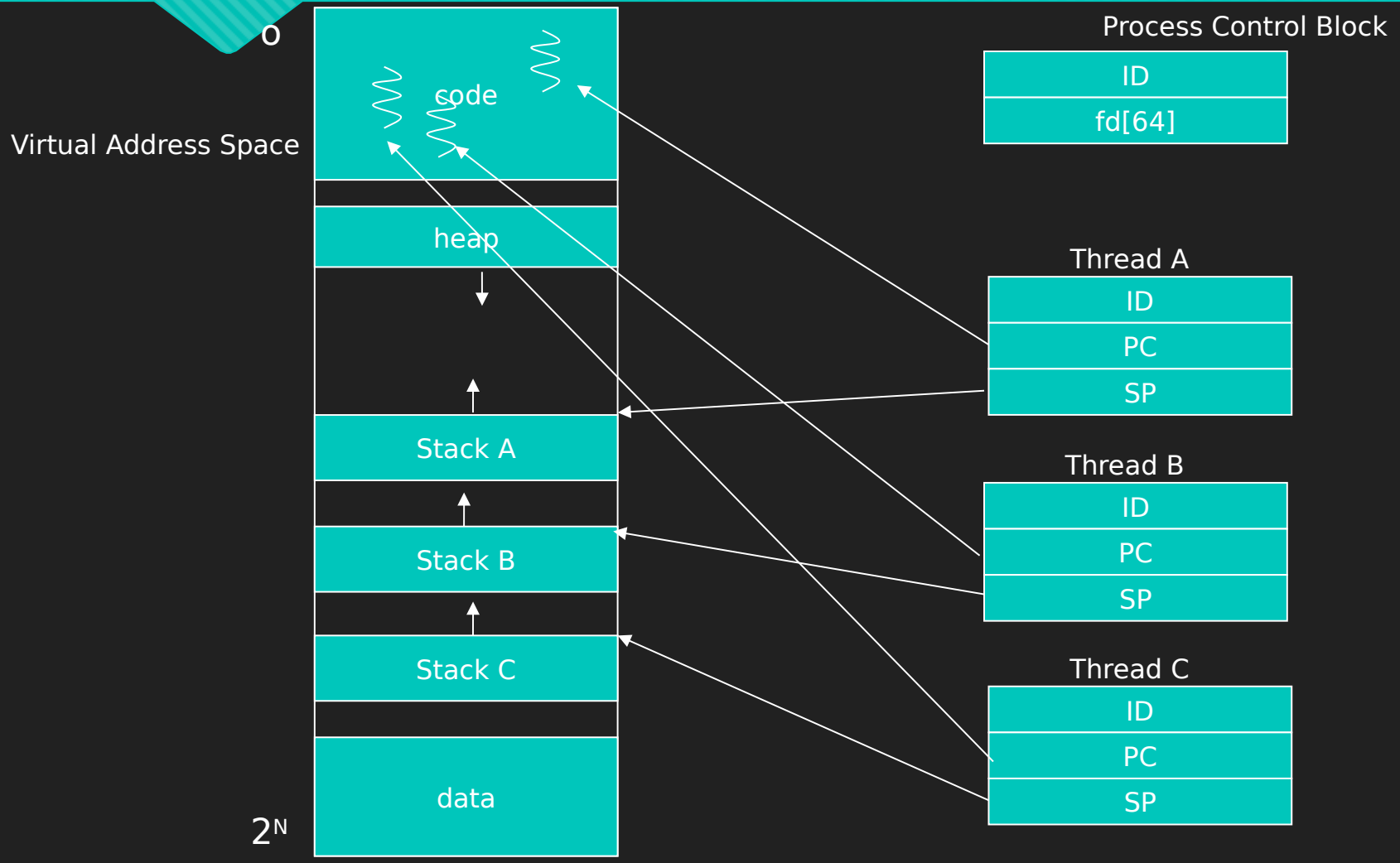
Mutex and Conditional Variables

Threads

- A thread is similar to a process (conceptually at least), except it shares a common address space.
  - All memory is shared memory*!!
- The downside is it’s much easier to corrupt your own memory.

*in that it’s accessible from all threads

Thread Memory Space



There’s nothing stopping thread A accessing thread B’s stack data.

Thread Benefits

- Advantages:
  - Light creation
  - Light context switch
  - Suitable for parallel computing
  - Natural form of resource sharing

Threads

- The initial program will contain one thread.
- A thread can spawn other threads.
- Threads are created equal!
- There is no hierarchy or dependency.

POSIX Threads (Pthreads)

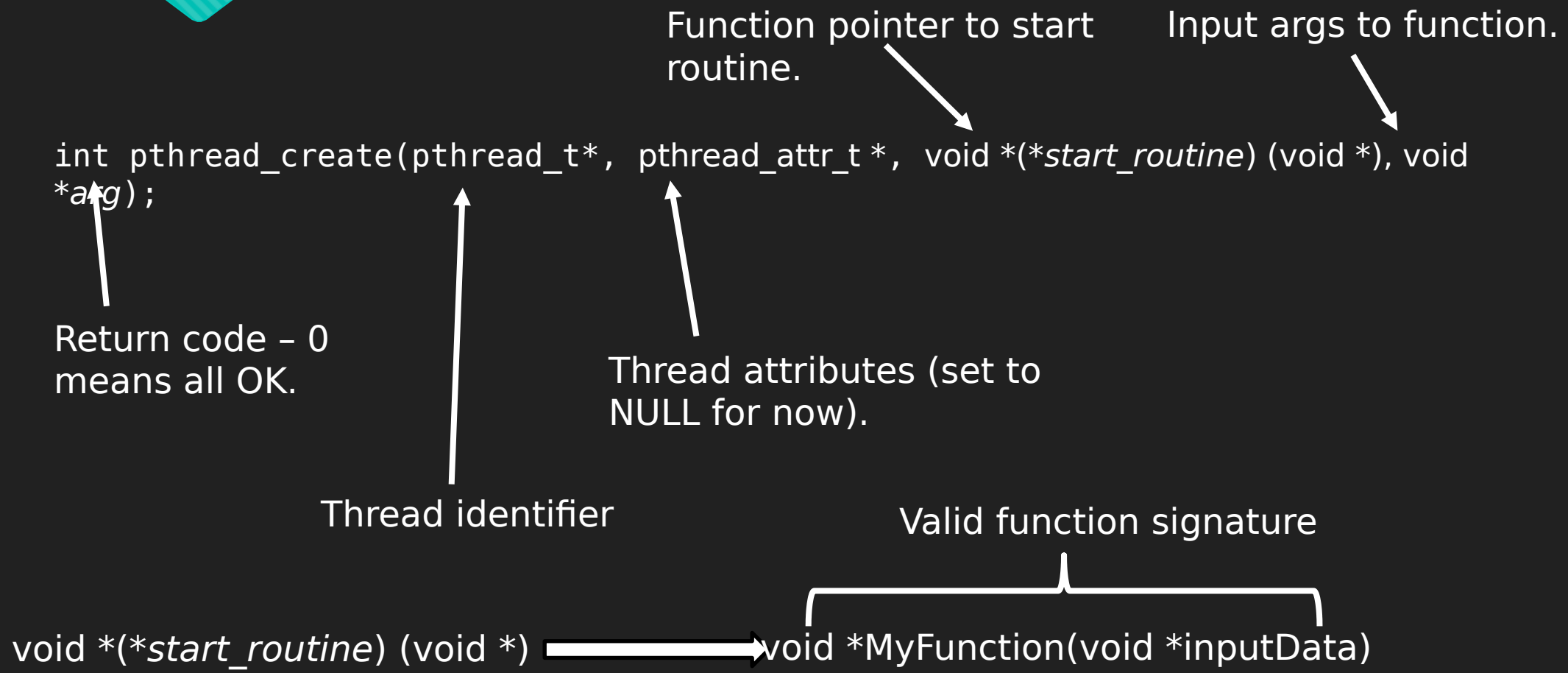
- UNIX system specific threading interface.
- Required header pthead.h.
- To compile

  g++ -pthread –o objectname cppname.cpp

Exiting a Thread

- Several options:
  - Start routine returns.
  - Thread calls pthread_exit to terminate itself.
  - Another thread calls pthread_cancel.
  - Exit() is called.
  - Main terminates.
- Good news – no zombie threads!

Getting Started



Join()

- Equivalent to processes Join (wait for thread to finish).

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

0 means no error

Thread to wait for.

Gets status of thread (use NULL for now).

This Week

- How do we prevent concurrent access to a resource?
  - Mutex
- How to prevent threads accessing a resource until that resource is in an appropriate state?
  - Conditional Variables

Sounds Familiar

- Didn’t we do this with Semaphores?
  - Yes, this can be achieved with Semaphores (more or less) However:
    - Semaphores relate to processes not threads, therefore, they are heavier to use.
    - The exact requirements (and purpose) of a mutex and semaphore are different.

Mutex (Mutual Exclusion Object)

- A mutex is a mechanism that can be used to prevent simultaneous access to a shared resource by multiple threads.

- Similar to semaphore, however:
  - At thread level so more lightweight.
  - Is either locked or not.
  - Cannot be locked or unlocked multiple times.
  - Must be locked then unlocked on same thread.

Mutex

- A mutex guarantees:
  - Atomicity - Locking a mutex is an atomic operation, meaning that the operating system (or threads library) assures you that if you locked a mutex, no other thread can lock this mutex at the same time.
  - Singularity - If a thread managed to lock a mutex, it is assured that no other thread will be able to lock the mutex until the original thread releases the lock.
  - Non-Busy Wait - If a thread attempts to lock a thread that was locked by a second thread, the first thread will be suspended (and will not consume any CPU resources).

Example

```
1  int COUNTER_A = 0;
2  int COUNTER_B = 0;
3
4
5  void *ChangeCounter(void *pVal)
6  {
7      int myVal = *(int *)pVal;
8
9      for (int i = 0; i< 10000; i++)
10     {
11         COUNTER_A = myVal;
12         sleep(0.001);
13         COUNTER_B = myVal;
14         if (COUNTER_A != COUNTER_B)
15             cout << "Counter's different!" << endl;
16     }
17
18     return NULL;
19 }
20
21
22 int main()
23 {
24     pthread_t threadA;
25     pthread_t threadB;
26
27     int valA = 0;
28     int valB = 1;
29
30     pthread_create(&threadA, NULL,     ChangeCounter, (void*)&valA);
31     pthread_create(&threadB, NULL, ChangeCounter, (void*)&valB);
32
33     pthread_join(threadA, NULL);
34     pthread_join(threadB, NULL);
35
36     return 0;
37 }
38
```

```
Counter's different!
Counter's different!
Counter's different!
Counter's different!
Counter's different!
Counter's different!
Counter's different!
Counter's different!
Counter's different!
Counter's different!
Counter's different!
simon@simon-VirtualBox:~$
```

- Create and initialise a Mutex.

```
pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
```

- Get a Lock:

```
int pthread_mutex_lock( pthread_mutex_t *);
```

- Unlock:

```
int pthread_mutex_unlock(pthread_mutex_t *);
```

- Destroy:

```
int pthread_mutex_destroy(pthread_mutex_t *);
```

```cpp
int COUNTER_A = 0;
int COUNTER_B = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *ChangeCounter(void *pVal)
{
    int myVal = *(int *)pVal;

    for (int i = 0; i< 10000; i++)
    {
        pthread_mutex_lock(&mutex);
        COUNTER_A = myVal;
        sleep(0.001);
        COUNTER_B = myVal;
        if (COUNTER_A != COUNTER_B)
            cout << "Counter's different!" << endl;
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}


int main()
{
    pthread_t threadA;
    pthread_t threadB;

    int valA = 0;
    int valB = 1;

    pthread_create(&threadA, NULL,    ChangeCounter, (void*)&valA);
    pthread_create(&threadB, NULL, ChangeCounter, (void*)&valB);

    pthread_join(threadA, NULL);
    pthread_join(threadB, NULL);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

```
simon@simon-VirtualBox:~$ g++ -pthread -o Example Example.cpp
simon@simon-VirtualBox:~$ ./Example
simon@simon-VirtualBox:~$
```

Conditional Variable

- Mechanism that allows threads to wait (without consuming CPU cycles) for some event to occur.
- Several threads can wait on a conditional variable.
- Either one thread can be signalled to start or all waiting threads.

Conditional Variable

- Warning:
  - A conditional Variable does not provide locking, hence must be used with Mutex.

Conditional Variable

- Creating a Conditional Variable:

```
pthread_cond_t condVar = PTHREAD_COND_INITIALIZER;
```

- Signalling a conditional Variable:

```
pthread_cond_signal(pthread_cond_t *)
```

Signals a single waiting thread.

```
pthread_cond_broadacast(pthread_cond_t *)
```

Signals all waiting threads.

Signalling Typical Usage

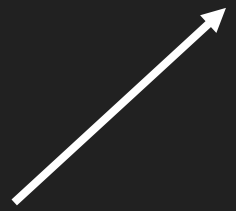
Lock prevents call on Wait() at same time.

```
pthread_mutex_lock(&mutex);
//do something which requires lock.
pthread_cond_signal(&CondVar);
pthread_mutex_unlock(&mutex);
```

Waiting

- Waiting is a *little* complicated:

```
pthread_cond_wait(pthread_cond_t *, p_thread_mutex_t
*);
```



Pointer to
Conditional Variable

Pointer to Mutex
which must be
locked!?

Waiting typical Usage

```
pthread_mutex_lock(&mutex);
pthread_cond_wait(&conditionalVar, &mutex);
// Do something that requires lock
pthread_mutex_unlock(&mutex);
```

1. Locks Mutex

2. Unlocks Mutex whilst waiting for signal (why?).

3. Locks Mutex once signal is received (Why? What does this mean?).

4. Unlocks Mutex.

Example

- See condVar.cpp (on BlackBoard) for example using Conditional Variables.

Summary

- We have learnt about Mutexes and Conditional Variables.
- Next week a quick look at Processes in Windows.
- Almost the end!
- Revise this stuff and practice it will all be assessed. And will also make you better programmers