# Concurrency in C / C++

https://youtu.be/zHDcAwhJQIc

# Concurrency in C / C++

There are several ways to achieve concurrent execution in C/C++

Multiple Threads – Threads sharing the same process memory.

Multiple Processes – Completely separate in their own memory.

These are essentially conceptually the same as the options in Python.  One difference: there is no interpreter or GIL in C / C++

Accordingly, we must also be careful to use synchronization and communication where appropriate.

# Concurrency in C / C++

**Multiple Threads:**

To spawn and manage multiple threads there are a couple of options:

**pthreads** – POSIX Threads Library (the old way)

**<thread>** - Starting with C++11 threading was added to the C standard language facilities. (the new way)

Let’s start with pthreads.

# Concurrency in C / C++

**Pthreads:**

Prior to C++11, there was no threading support directly built into the language itself.

So threading had to be supported through the use of a library.

Pthreads is a POSIX library for C / C++ that adds threading support in a standardized way.

To use pthreads, you just need to include pthread.h.

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

To use pthreads you just need to use it’s functions...

# Concurrency in C / C++

**Create a thread:**

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

**Description:** The pthread_create() function is used to create a new thread, with attributes specified by attr, within a process. If attr is NULL, the default attributes are used. Upon successful completion, pthread_create() stores the ID of the created thread in the location referenced by thread.

The thread is created executing start_routine with arg as its sole argument. If the start_routine returns, the effect is as if there was an implicit call to pthread_exit() using the return value of start_routine as the exit status.

**Return**: If successful, the pthread_create() function returns zero. Otherwise, an error number is returned to indicate the error.

# Concurrency in C / C++

**Join a thread:**

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

**Description:** The pthread_join() function suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated.

On return from a successful pthread_join() call with a non-NULL value_ptr argument, the value passed to pthread_exit() by the terminating thread is made available in the location referenced by value_ptr.

When a pthread_join() returns successfully, the target thread has been terminated.

**Return:** If successful, the pthread_join() function returns zero. Otherwise, an error number is returned to indicate the error.

# Concurrency in C / C++

**Exit a thread:**

```
void pthread_exit(void *value_ptr);
```

**Description:** The pthread_exit() function terminates the calling thread and makes the value value_ptr available to any successful join with the terminating thread.

An implicit call to pthread_exit() is made when a thread other than the thread in which main() was first invoked returns from the start routine that was used to create it. The function's return value serves as the thread's exit status.

**Return:** The pthread_exit() function cannot return to its caller.

# Concurrency in C / C++

## Pthreads example:

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

void *thread_func1(void *);
void *thread_func2(void *);

int main(void)
   {
      pthread_t t1;
      pthread_t t2;
      int *arg;
      char *ret;

      // start the threads
      printf("Starting Threads\n");
      pthread_create(&t1, NULL, thread_func1, (void *) NULL);
      pthread_create(&t2, NULL, thread_func2, (void *) NULL);


// continued...
```

# Concurrency in C / C++

```
// continued…
      // Wait for each thread to return
      pthread_join(t2, (void *)&ret);
      free(ret);
      pthread_join(t1, (void *)&ret);
      free(ret);
      printf("\nBoth threads finished!\n");
      return(0);
   }

   void *thread_func1(void *crap)
   {
       for(i=0;i<100;i++)
          {
               printf("1"); fflush(stdout);
          }
          pthread_exit( NULL );
   }

   void *thread_func2(void *crap)
   {
          for(i=0;i<100;i++)
          {
               printf("2"); fflush(stdout);
          }
          pthread_exit( NULL );
   }
```

# Concurrency in C / C++

**Example Output:**

```
Starting Threads
1122212122121212121211212112121112121212112121211212122121121121121121121122121121212121
1211221112121212121122121212212121211121212112122121121122121121121212121221211212121212
1212222222222222222
Both threads finished!
```

**Other runs will give different output:**

```
Starting Threads
1221212112211212211221212121121212111212122121212121121112212112121121121221121121212121122112111
2122121211212211212121122121121122121121112121222112112112122112112121121121121121121121121121121212
2222222222222222222
Both threads finished!
```

Since both threads are executing concurrently their execution will be non-deterministic.

# Concurrency in C / C++

**Pthreads Synchronization:**

Since concurrent execution is non-deterministic, we need mechanisms to control that execution.

Pthreads offers three mechanisms for thread synchronization:

pthread_join() - Wait until another thread completes. We've already used this!

mutexes - Mutual exclusion lock: Can be locked and unlocked to control access to a critical section. This enforces mutual exclusion.

condition variables - data type pthread_cond_t that supports wait(), signal(), and broadcast() functions.

These are analogous to the synchronization mechanisms we've seen before.

# Concurrency in C / C++

**Pthreads mutexes:**

Mutexes are identical to Lock() objects we've been using in Python.

```
/* global mutex can be accessed by multiple threads */
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

/* some thread function */
void *some_thread_func(void *args)
{
  pthread_mutex_lock( &mutex1 );
  // some critical section protected by the mutex.
  pthread_mutex_unlock( &mutex1 );
}
```

We can have as many mutexes as necessary.
We can have as many threads share a set of locks as necessary.

# Concurrency in C / C++

**Pthreads condition variables:**

The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true.

A condition variable must always be associated with a mutex to avoid a race condition created by one thread preparing to wait and another thread which may signal the condition before the first thread actually waits on it resulting in a deadlock.  If this happens, the thread will be waiting for a signal that will never be sent.

Let’s look at an example...

# Concurrency in C / C++

**Pthreads condition variables (wait):**

```
pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_cond  = PTHREAD_COND_INITIALIZER;

void *thread_function1(void*args)
{
   //some code here
   pthread_mutex_lock( &condition_mutex );
   while( /* some condition that we want to wait upon */ )
      {
         pthread_cond_wait( &condition_cond, &condition_mutex );
      }
    //perform the condition protected action.
    pthread_mutex_unlock( &condition_mutex );
    //some code here
}
```

# Concurrency in C / C++

**Pthreads condition variables (signal):**

```
void *thread_function2(void*args)
{
        //some code here
        pthread_mutex_lock( &condition_mutex );
        if( /*some condition that releases the hold*/ )
        {
           pthread_cond_signal( &condition_cond );
           //could also use:
           // pthread_cond_broadcast( &condition_cond );  which is like signal_all.
        }
        pthread_mutex_unlock( &condition_mutex );
        //some code here
}
```

# Concurrency in C / C++

**Concurrency in C++11.  (The NEW way!)**

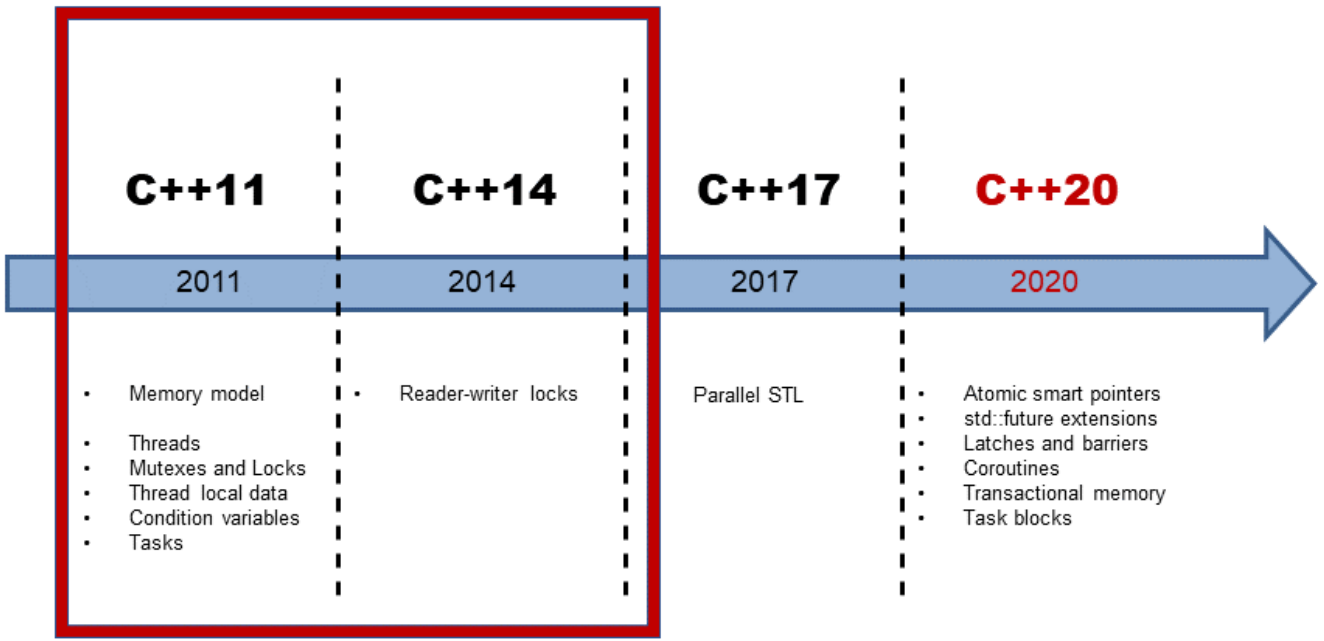
Starting in C++11 threads and other concurrency support mechanisms were added to the standard language facilities.

This means that we don’t necessarily have to use a library like the POSIX pthreads library to write concurrent code.

Let’s take a quick look at this...

# Concurrency in C / C++

**Concurrency in C++11.  (The NEW way!)**



# Concurrency in C / C++

**Simple Threading in C++11:**

```
#include <iostream>
#include <thread>

void thread_func1(void);
void thread_func2(void);

using namespace std;

int main()
{
    cout << "Starting threads" << endl;
    thread t1(thread_func1);
    thread t2(thread_func2);
    t1.join();
    t2.join();
    cout << endl << "Both threads done!" << endl;
    return 0;
}

//continued...
```

# Concurrency in C / C++

```
//...continued

void thread_func1(void)
{
  int i;
  for(i=0;i<100;i++)
  {
     printf("1"); fflush(stdout);
  }
}

void thread_func2(void)
{
  int i;
  for(i=0;i<100;i++)
  {
     printf("2"); fflush(stdout);
  }
}
```

# Concurrency in C / C++

**Synchronization in C++11:**

C++ mutexes are just like pthreads mutexes.

```
/* global mutex can be accessed by multiple threads */
std::mutex mutex1;

/* some thread function */
void some_thread_func(void)
{
  mutex1.lock();
  // some critical section protected by the mutex.
  mutex1.unlock();
}
```

Again, we can have as many mutexes as necessary.
And, we can have many threads share mutexes as necessary.

# Concurrency in C / C++

**Synchronization in C++11:**

C++11 also supports other synchronization primitives:

**mutex** (C++11) - provides basic mutual exclusion facility

**timed_mutex** (C++11) - provides mutual exclusion facility which implements locking with a timeout

**recursive_mutex** (C++11) - provides mutual exclusion facility which can be locked recursively by the same thread.

**recursive_timed_mutex** (C++11) - provides mutual exclusion facility which can be locked recursively by the same thread and implements locking with a timeout.

**lock_guard** (C++11) - implements a strictly scope-based mutex ownership wrapper.

**unique_lock** (C++11) - implements movable mutex ownership wrapper.

**condition_variable** (C++11) - provides a condition variable associated with a std::unique_lock.

**condition_variable_any** (C++11) - provides a condition variable associated with any lock type.

# Concurrency in C / C++

**Other Concurrency Features in C++11 and beyond...**

C++11 and later versions provide many features related to concurrent execution.

- Locks (other types)
- Condition Variables
- Futures and Async
- Latches and Barriers
- Semaphores
- Parallel algorithms

For more details consult the C++ documentation:
https://en.cppreference.com/w/cpp/thread

There are also C interfaces to threading support that have been added in C11:
https://en.cppreference.com/w/c/thread

# Concurrency in C / C++

That’s all for today!

Stay Safe!