University of Glasgow

Systems Programming – Part 2
Concurrent Systems Programming

Dr Lauritz Thamsen
lauritz.thamsen@glasgow.ac.uk
https://lauritzthamsen.org

University of Glasgow

Topics of Part 2 of SP(GA)

- **Intro to Concurrency (with Processes and Threads)**
  - **Intro to POSIX Threads**
- Process/Thread Synchronisation
- More on Process Management (from an OS Perspective)
- Concurrency Beyond Threads & Limits of Scalability
- Virtual Memory & Levels of Storage

Intro to POSIX Threads

University of Glasgow

Thread Implementations

- There are thread implementations in almost all programming languages; we will look at the C **pthread** library

- Many threading implementations are conceptually quite similar, and realise a particular thread lifecycle:
  - A thread is **created**: starts executing a specified function, with some arguments, and is given an identifier
  - A thread can **wait** for another thread to terminate
  - A thread can **cancel** another thread
  - A thread **terminates** either by calling exit or when its main function ends
  - **Communication** between threads happens through modifying the state of **shared variables**

University of Glasgow

POSIX Threads

- Part of the **POSIX specification** collection, defining an API for thread creation and management (***pthread.h***)
- Implementations for all **Unix-alike operating systems** available
  - Utilization of kernel- or user-mode threads depends on implementation
- Groups of functionality (pthread_ function prefix)
  - **Thread management**: Start, wait for termination, ...
  - **Synchronization means**: Mutexes, condition variables, read/write locks, barriers
- Semaphore API is a separate POSIX specification (sem_ prefix)

University of Glasgow

POSIX Threads

- POSIX Threads is the most used threading implementation for C
- To use it we need to *#include <pthread.h>* and specify a compiler flag (*–lpthread)*

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

void * PrintHelloWorld(void*) {
  printf("Hello World from a thread!\n");
  return NULL;
}
```

```c
int main() {
  pthread_t thread;
  int error = pthread_create(&thread,
    NULL, PrintHelloWorld, NULL);
  assert(error == 0);
  printf("Created thread\n");
  error = pthread_join(thread, NULL);
  assert(error == 0);
}

clang -Wall -Werror program.c -lpthread -o program
```

University of Glasgow

POSIX Threads

- *pthread_create()*
  - Create a new thread in the same process, with a given routine and argument

- *pthread_exit(), pthread_cancel()*
  - Terminate a thread from inside (→ exit) or outside (→ cancel) of the thread

University of Glasgow

Creating POSIX Threads

- Threads are created with the *pthread_create* function

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

- It takes four arguments:
  1. **A thread** identifier, which is a pointer to a memory location of type *pthread_t*
  2. Thread attributes which set properties such as scheduling policies or stack size (and passing *NULL* results in default attributes)
  3. A **function pointer** to the ***start_routine***
     This function takes a single argument of type *void** and returns a value of type *void**
  4. The **argument** that is passed **to *start_routine*** *(a pointer)*
- It returns 0 if the thread is created successfully or, else, a non-zero error code
- **Passing pointers to and from start_routine allows the passing of arbitrary data**
- It requires care to ensure that the memory locations pointed to have appropriate lifetimes

Passing an argument to a thread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
  long tid;
  tid = (long)threadid;
  printf("Hello World! It's me, thread #%ld!\n", tid);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  for(t=0;t<NUM_THREADS;t++){
    printf("In main: creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

University of Glasgow

Waiting for Other Threads

- *pthread_join()*
  - Blocks the caller until the specified thread terminates
  - Only one joining thread per target is allowed
  - If the thread gave an exit code to *pthread_exit()*, it can be determined here

University of Glasgow

Waiting for Other Threads

- To wait for another thread to terminate we use *pthread_join*

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

- It takes two arguments:
  1. A thread identifier (the thread we want to wait for)
  2. A **pointer** to a **memory location** of type void*

     The return value of the *start_routine* (== a generic pointer), as passed to *pthread_create,* will be copied to this location

- It returns 0 on success and, otherwise, a non-zero error code

University of Glasgow

Waiting for Another Thread

- Example of returning a single int value from a thread:

```
int* return_value;

int error = pthread_join(thread, (void**)&return_value);

assert(error == 0);

if (return_value) {
  printf("return_value: \n", *return_value);
}

// maybe: free(return_value);
```

University of Glasgow

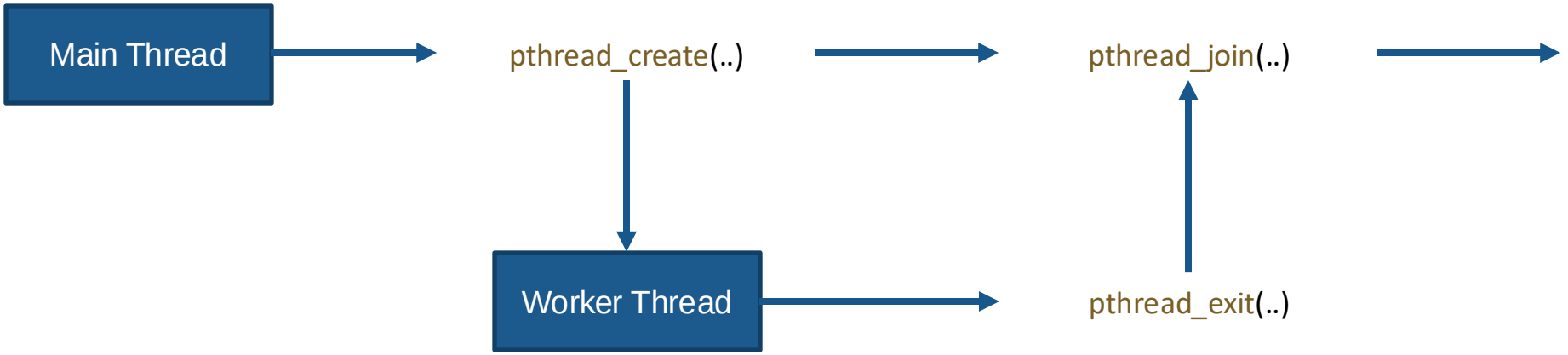
Waiting for Another Thread

```
void *thread_func(void *args) {
  /* ... */

  /* Allocate heap space for this thread's results */
  struct results *results = calloc(sizeof (struct results), 1);

  /* ... populate results struct ... */

  /* Return the pointer to results */
  pthread_exit(results);
}
```

```
struct results *results;

int error = pthread_join(thread, (void**)&results);

assert(error == 0);

/* make use of results, but then also free(results)
at some point */
```

University of Glasgow

Forking and Joining Threads



University of Glasgow

To Do Now

Completing this activity will support your understanding of the material today (before we have real lab time tomorrow!)

In the "Lecture 2.2 files" folder on Moodle

- Review the code for *pthread_hello_world.c*
- Compile it and run it – don't forget the "-lpthread" flag for clang (see Slide 5 for an example)
- Add another print statement "Last thread about to die" just before the main thread terminates

(Optional) Recommended Reading

- Blaise Barney, Lawrence Livermore National Laboratory, UCRL-MI-133316, POSIX Threads Programming, https://hpc-tutorials.llnl.gov/posix/, Section 5 for today and later Section 7

- Videos
  - Programming with Threads, Jacob Sorber, https://www.youtube.com/playlist?list=PL9IEJIKnBJjFZxuqyJ9JqVYmuFZHr7CFM
  - Unix Threads in C, CodeVault, https://www.youtube.com/playlist?list=PLfqABt5AS4FmuQf70psXrsMLEDQXNkLq2