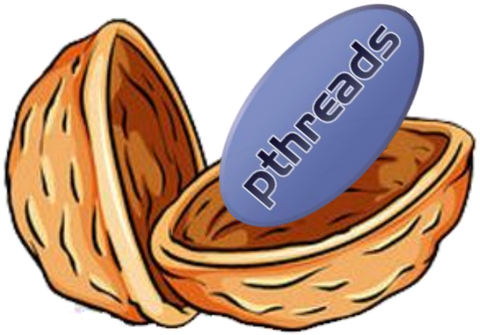
POSIX Threads in a nutshell

Paolo Burgio
paolo.burgio@unimore.it



HiPeRT Lab

What will we see

- A mix of theory…
- ..and practice / exercise
  - Don't miss it
- Please, interrupt me

The POSIX IEEE standard

eng.wikipedia.org

POSIX Threads, usually referred to as Pthreads, is an execution model that exists independently from a language, as well as a parallel execution model. It allows a program to control multiple different flows of work that overlap in time.

- Threading API
- Single process
- Shared memory space

The POSIX IEEE standard

- Specifies an operating system interface similar to most UNIX systems
  - It extends the C language with primitives that allows the specification of the concurrency

- POSIX distinguishes between the terms process and thread
  - "A process is an address space with one or more threads executing"
  - "A thread is a single flow of control within a process (a unit of execution)"

- Every process has at least one thread
  - the "`main()`" (aka "master") thread; its termination ends the process
  - All the threads share the same address space, and have a private stack

Thread body

- A (P)thread is identified by a C function, called body:

```
void *my_pthread_fn(void *arg)
{
  // Thread body
}
```

- A thread starts with the first instruction of its body
- The threads ends when the body function ends
  - It's not the only way a thread can die

Thread creation

- Thread can be created using the primitive

pthread.h

```
typedef unsigned int pthread_t;

int pthread_create ( pthread_t *ID,
                     pthread_attr_t *attr,
                     void *(*body)(void *),
                     void * arg
                       );
```

- `pthread_t` is the type that contains the thread ID
- `pthread_attr_t` is the type that contains the parameters of the thread
- `arg` is the argument passed to the thread `body` when it starts

Thread attributes

- Thread attributes specifies the characteristics of a thread
  - We won't see this; leave empty

- Attributes must be initialized and destroyed - always

pthread.h

```
int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_destroy(pthread_attr_t *attr);
```

Thread termination

- A thread can terminate itself calling

pthread.h

```
void pthread_exit(void *retval);
```

- When the thread body ends after the last “}”, pthread_exit() is called implicitly
- Exception: when main() terminates, exit() is called implicitly

Thread IDs

- Each thread has a unique ID

pthread.h

```
pthread_t pthread_self(void);
```

- The thread ID of the current thread can be obtained using

pthread.h

```
int pthread_equal( pthread_t thread1,
                   pthread_t thread2 );
```

- Two thread IDs can be compared using

Joining a thread

- A thread can wait the termination of another thread using

pthread.h

```
int pthread_join ( pthread_t th,
                   void **thread_return);
```

- It gets the return value of the thread or `PTHREAD_CANCELED` if the thread has been killed

- By default, every thread must be joined
  - The join frees all the internal resources
  - Stack, registers, and so on

Example

Let's code!

- Filename: `hello_pthreads_world.c`
- The demo explains how to create a thread
  - the `main()` thread creates another thread (called `body()`)
  - the `body()` thread checks the thread Ids using `pthread_equal()` and then ends
  - the `main()` thread joins the `body()` thread
- When compiling under gcc & GNU/Linux, remember
  - the `-lpthread` option!
  - to add `#include "pthread.h"`

- Credits to PJ

Detached threads

- A thread which does not need to be joined has to be declared as detached

- 2 ways to have it:
  - While creating (in father thread) using `pthread_attr_setdetachstate()`
  - The thread itself can become detached calling in its body `pthread_detach()`

- Joining a detached thread returns an error

Killing a thread

- A thread can be killed calling

```
int pthread_cancel(pthread_t thread);
```

- When a thread dies its data structures will be released
  - By the join primitive if the thread is joinable
  - Immediately if the thread is *detached*
  - Why?

PThread cancellation

- Specifies how to react to a kill request
- There are two different behaviors:
  - deferred cancellation

    when a kill request arrives to a thread, the thread does not die. The thread will die only when it will execute a primitive that is a cancellation point. This is the default behavior of a thread.
  - asynchronous cancellation

    when a kill request arrives to a thread, the thread dies. The programmer must ensure that all the application data structures are coherent.

Cancellation states and cleanups

- The user can set the cancellation state of a thread using:

```
int pthread_setcancelstate(int state,int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

- The user can protect some regions providing destructors to be executed in case of cancellation

```
int pthread_cleanup_push(void (*routine)(void *),
                         void *arg);
int pthread_cleanup_pop(int execute);
```

Cancellation points

- The cancellation points are primitives that can potentially block a thread

- When called, if there is a kill request pending the thread will die
  - `void pthread_testcancel(void);`
  - `sem_wait`, `pthread_cond_wait`, `printf` and all the I/O primitives are cancellation points
  - `pthread_mutex_lock`, is NOT a cancellation point
    - Why?

- A complete list can be found into the POSIX Standard

Cleanup handlers

- The user must guarantee that when a thread is killed, the application data remains coherent.

- The user can protect the application code using cleanup handlers
  - A cleanup handler is a user function that *cleans up* the application data
  - They are called when the thread ends and when it is killed

Cleanup handlers (2)

```
void pthread_cleanup_push (void (*routine)(void *),
                                  void *arg);
void pthread_cleanup_pop (int execute);
```

- They are pushed and popped as in a stack
- If `execute!=0` the cleanup handler is called when popped
- The cleanup handlers are called in LIFO order

How to run the examples

Let's
code!

- Download the `Code/` folder from the course website

- Compile

```
$ gcc code.c -o code -lpthread
```

- Run (Unix/Linux)

```
$ ./code
```

- Run (Win/Cygwin)

```
$ ./code.exe
```

Useful links

- Course webpage
  - https://hipert.unimore.it/people/paolob/pub/Calcolo_Parallelo/
- Course GitHub
  - https://github.com/HiPeRT/cp19/
- My contacts
  - paolo.burgio@unimore.it
  - http://hipert.mat.unimore.it/people/paolob/
- PThreads
  - https://computing.llnl.gov/tutorials/pthreads/
  - http://man7.org/linux/man-pages/man7/pthreads.7.html
- A "small blog"
  - http://www.google.com