

# POSIX Threads in a nutshell

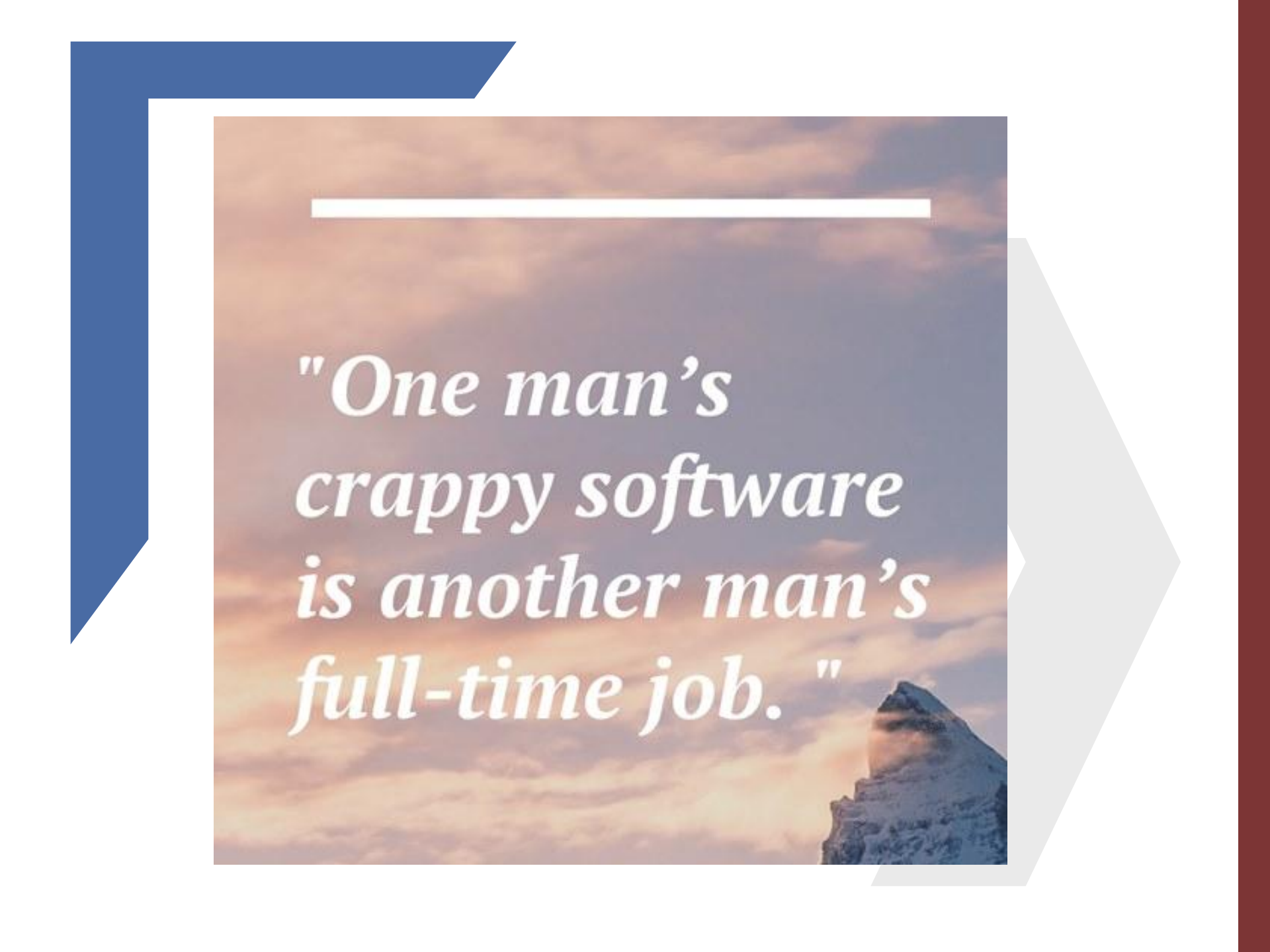
---

Paolo Burgio  
paolo.burgio@unimore.it



**UNIMORE**  
UNIVERSITÀ DEGLI STUDI DI  
MODENA E REGGIO EMILIA

High Performance  
Real Time **Lab**



---

*"One man's  
crappy software  
is another man's  
full-time job. "*



# The POSIX IEEE standard

---

[eng.wikipedia.org](http://en.cppreference.com/w/cpp/thread)

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_workd.c[pp]`
- › 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





# Semaphores



# Semaphores

---

A semaphore is a counter managed with a set of primitives

It is used for

- › Synchronization
- › Mutual exclusion (critical sections)

POSIX Semaphores can be

- › Unnamed (local to a process)
- › Named (shared between processes through a file descriptor – we won't see them)



# Unnamed semaphores

---

Operations permitted:

- › initialization /destruction
- › blocking wait / nonblocking wait
  - counter decrement
- › post
  - counter increment
- › counter reading
  - simply returns the counter



# Initializing a semaphore

---

- › The `sem_t` type contains all the semaphore data structures

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- `pshared` is 0 if `sem` is not shared between processes

```
int sem_destroy(sem_t *sem)
```

- It destroys the `sem` semaphore



# Semaphore waits

---

```
int sem_wait(sem_t *sem);
```

```
int sem_trywait(sem_t *sem);
```

- › Under the hood..
- › If the counter is greater than 0 the thread does not block
  - `sem_trywait` never blocks





# Other semaphore primitives

---

```
int sem_post(sem_t *sem);
```

- It increments the semaphore counter
- It unblocks a waiting thread

```
int sem_getvalue(sem_t *sem, int *val);
```

- It simply returns the semaphore counter



# Example(s)

---

Let's  
code!

Filename: `critical-section.c`

- › In this example, semaphores are used to implement mutual exclusion in the output of a character in the console

Filename: `producer-consumer.c`

- › In this example, semaphores are used to implement producer-consumer synchronization



Mutexes



# What is a (POSIX) mutex?

---

- › Like a **binary semaphore**, used for **mutual exclusion**
  - But.. a mutex can be unlocked **only** by the thread that locked it
- › Mutexes also support some RT protocols
  - Priority inheritance, priority ceiling...
  - They are not implemented under a lot of UNIX OS
  - Out of scope for this course



# Mutex attributes

---

- › Mutex attributes are used to initialize a mutex

```
int pthread_mutexattr_init (pthread_mutexattr_t *attr);  
int pthread_mutexattr_destroy (pthread_mutexattr_t  
    *attr);
```

- › Initialization and destruction of a mutex attribute



# Mutex initialization

---

- › Initialize a mutex with a given mutex attribute

```
int pthread_mutex_init (pthread_mutex_t *mutex,  
                        const pthread_mutexattr_t *attr);
```

- › Destroys a mutex

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



# Mutex lock and unlock

---

- › This primitives implement the blocking lock, the non-blocking lock and the unlock of a mutex
- › The mutex lock is **NOT** a cancellation point

```
int pthread_mutex_lock(pthread_mutex_t *m);  
int pthread_mutex_trylock(pthread_mutex_t *m);  
int pthread_mutex_unlock(pthread_mutex_t *m);
```



# Example

---

Let's  
code!

- › Filename: `ex_mutex.c`
- › This is prev. example written using mutexes instead of semaphores.





---

---

# PThreads scheduling





# Scheduling algorithms

---

- › The POSIX standard specifies in `sched.h` *at least* two scheduling strategies which can be used, identified by the symbols `SCHED_FIFO` and `SCHED_RR`
  - Other scheduling policies may be supported by each particular implementation, under the symbol `SCHED_OTHER`

POSIX specifies a Fixed Priority scheduler with at least 32 priorities (0 to 31)

- › Every priority corresponds to a queue, where all the threads with the same priority are inserted
- › The first ready thread in the highest non-empty priority queue is selected for scheduling and becomes the running thread



# POSIX and priorities

---

thread priorities can be specified at creation time into the thread attributes

```
int pthread_attr_setschedpolicy  
    (pthread_attr_t *a, int policy);
```

› policy can be SCHED\_RR, SCHED\_FIFO or SCHED\_OTHER

```
int pthread_attr_setschedparam  
    (pthread_attr_t *attr,  
     const struct sched_param *param);
```

› The priority field is param.sched\_priority



# Real-Time and UNIX

---

- › UNIX systems usually schedule all its threads at low priorities
- › When a RT thread is created, it always preempts all the other applications (i.e. the X server, and all the other demons)
- › For that reason,
  - real-time computations have to be limited
  - **only root** can use the real-time priorities



# Example

Let's  
code!

- › Filename: `ex_rr.c`
- › The demo explains the behavior of the RT priorities and of the other policies
- › The `main()` thread creates a high priority thread that activates a low priority thread and two medium priority threads
- › The medium priority threads are scheduled with policies `SCHED_RR` and `SCHED_FIFO`
- › When compiling under gcc & GNU/Linux, remember
  - the `-lpthread` option!
  - to add `#include "pthread.h"`

› Credits to PJ





# 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
```



# References

---



## Course website

- › [http://hipert.unimore.it/people/paolob/pub/Industrial\\_Informatics/index.html](http://hipert.unimore.it/people/paolob/pub/Industrial_Informatics/index.html)

## My contacts

- › [paolo.burgio@unimore.it](mailto:paolo.burgio@unimore.it)
- › <http://hipert.mat.unimore.it/people/paolob/>

## Resources

- › <https://computing.llnl.gov/tutorials/pthreads/>
- › <http://man7.org/linux/man-pages/man7/pthreads.7.html>
- › A "small blog"
  - <http://www.google.com>