



Managing Concurrency in POSIX POSIX Threads

Luca Abeni
`luca.abeni@unitn.it`

November 5, 2014



The POSIX Standard

- Is an IEEE standard that specifies an operating system interface similar to most unix systems
 - ◆ The POSIX standard is not “free as gratis” (you have to pay for having it)
 - ◆ You can refer to the opengroup standard (www.opengroup.org) instead
 - ◆ <http://www.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html> for pthreads...
- The standard defines a C API to handle concurrent activities
 - ◆ POSIX makes a distinction between *processes* and *threads*

Threads

- A thread is a schedulable entity (a flow of execution)
- A process is composed by one or more threads + some private resources (address space, file table, etc...)
 - ◆ So, a thread is a single flow of control within a process
 - ◆ Every process has at least 1 thread, the *main* thread
- All the threads in a process share the same address space, file table, program body, etc...
- Each thread has its own context, and its own stack
- In each process, there is a “special” thread:
 - ◆ Terminating the main thread of a process terminates the process

PThreads

- The POSIX standard defines its own threading library: the `pthread` library
- All the primitives operating (creation, termination, synchronization, etc...) on threads are implemented in the `pthread` library
- POSIX threading primitives and data structures are declared in:
 - ◆ `pthread.h`
 - ◆ `semaphore.h` (if using semaphores!)
- Use the `man` command to access on-line documentation
- When compiling with `gcc` (generally under GNU/Linux), use the `-pthread` option!!!

Thread Body

- The code executed by a thread is defined by a C function, the thread body:

```
1 void *my_thread(void *arg)
2 {
3     ...
4 }
```

- When created, a thread starts executing the first instruction of its body
- The thread ends when exiting the body (at the end of the C function)
 - ◆ But a thread can terminate also in other ways (by explicitly calling a termination function, when killed by another thread, etc...)

Thread Creation

- Threads can be created by using the `pthread_create()` call:

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

- The `attr` parameter (of type `pthread_attr_t`) describes some thread's attributes
- `body` is a pointer to the thread body
- `arg` is the argument passed to the thread body on start
- The identifier of the created thread is returned in `id` (of type `pthread_t`)
- The return value is 0 if no error occurred, $\neq 0$ in case of error

Thread Attributes

- The thread attributes specified in `attr` permits to control some of the characteristics of the created threads
 - ◆ Stack size (and address)
 - ◆ Detach state (joinable or detached)
 - ◆ Some scheduling parameters (priority, etc...)

- Thread attributes must be initialized and destroyed:

```
int pthread_attr_init(pthread_attr_t *attr)
```

```
int pthread_attr_destroy(pthread_attr_t *attr)
```

Thread Termination

- A thread can terminate by using the `pthread_exit()` call:

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

- A thread also terminates when its execution arrives at the end of the thread body
 - ◆ `pthread_exit()` is automatically called when returning from the thread body
- When the main thread exits, `exit()` is called (and not `pthread_exit()`), and the process terminates

Thread IDs

- Each thread is identified by an unique ID (returned by `pthread_create()`)
- The ID of the current thread can be obtained by using `pthread_self()`
`pthread_t pthread_self(void)`
- Two IDs can be compared by using `pthread_equal()`
`int pthread_equal(pthread_t id1,
pthread_t id2)`

Thread Synchronization - Join

- A thread can wait for the termination of another thread by calling `pthread_join()`

```
int pthread_join(pthread_t id, void **result)
```

- The return value of the thread (or `PTHREAD_CANCELED` if the thread has been killed) is returned in `result`
- By default, every thread must be joined
 - ◆ The private resources of a terminate thread are not released until a join occurs
 - ◆ Similar to what happens with processes and `wait()` (think about zombies)

Detached Threads

- A thread that will not be joined has to be declared as `detached`
 - ◆ When a detached thread terminates, its resources are immediately released
- There are two ways to detach a thread:
 - ◆ The “detached” state is set on thread creation (through the `attr` parameter), by using `pthread_attr_setdetachstate()`
 - ◆ The thread becomes detached by calling `pthread_detach()`
- Joining a detached thread results in an error

Example 1

- File: `http://dit.unitn.it/~abeni/RTOS/ex_create.c`
- The example shows how to create threads
 - ◆ The main thread (having body `main()`) creates a second thread, with body `body()`
 - ◆ The second thread check the thread IDs, by using `pthread_equal()`, and then exits
 - ◆ The main thread waits for the other thread termination by joining it
- Also: `http://dit.unitn.it/~abeni/RTOS/thread.c`
 - ◆ Compare with the `fork.c` example

Scheduling Algorithms - 1

- The POSIX standard supports *fixed priority* scheduling through the SCHED_FIFO and SCHED_RR scheduling policies
- The functions and data types needed to set the scheduling policy are declared in the sched.h header
 - ◆ The sporadic server has been recently added to the standard
- The system can define additional scheduling policy
 - ◆ In particular, traditional unix scheduling is often supported with the name SCHED_OTHER or SCHED_NORMAL

Scheduling Algorithms - 2

- The POSIX standard mandates a fixed priority scheduler with at least 32 priority levels (from 0 to 31)
 - ◆ As usual, the highest priority ready thread is scheduled
 - ◆ What happens if two threads have the same priority?
- There is one queue per priority level, containing all the ready threads for such priority
 - ◆ The highest priority ready thread is in the highest level non-empty queue
 - ◆ The first thread from the highest non-empty queue is selected for scheduling and becomes the **running thread**
 - ◆ So, the question is: how are the priority queues handled?

FIFO vs Round Robin

- The priority queues can be handled according to a FIFO or to a Round-Robin strategy
 - ◆ SCHED_FIFO: First In First Out queueing. This means that the highest priority thread is scheduled until it ends (or it is canceled), it blocks, or it is preempted by an higher priority thread.
 - ◆ SCHED_RR: Round Robin queueing. The highest priority thread is also descheduled when its *scheduling quantum* expires
- The SCHED_OTHER policy is also often provided as implementation dependent.
 - ◆ It often is a UNIX scheduler with aging
 - Quantum expiration → priority decreases
 - Task blocks → priority increases

Resource Sharing

- Some real-time resource sharing protocols are also supported, by using mutexes to protect the shared resource
 - ◆ Priority Ceiling
 - ◆ Priority Inheritance
 - ◆ **Warning**: not all the implementations support them
- POSIX leaves unspecified the scheduling order between threads belonging to different processes
- Example:
 - ◆ There can be “global” thread scheduling...
 - ◆ ...Or threads can be scheduled “per process”

Setting the Scheduling Policy

- When creating a thread, the scheduling policy and parameters can be set through the `attr` parameter
 - ◆ To do this, the “scheduler inheritance” attribute should be set to `PTHREAD_EXPLICIT_SCHED`
 - ◆ The default value of such attribute is implementation dependent

```
int pthread_attr_setschedpolicy(pthread_attr_t *a,  
                               int policy)  
int pthread_attr_setschedparam(pthread_attr_t *a,  
                               const struct sched_param *param)  
int pthread_attr_setinheritsched(pthread_attr_t *attr,  
                                 int inheritsched)
```

- The only important field in `sched_param` is `sched_priority`

Real-Time Priorities - 1

- General purpose systems such as Windows or Unix schedule “regular” threads in background respect to real-time ones
- Real-Time threads can be scheduled in foreground respect to all the other threads / processes
- A real-time thread can preempt / starve other applications
 - ◆ This can be implementation-dependent (the scheduling order between threads belonging to different processes is unspecified)...
 - ◆ ...But most systems use “global” scheduling → we can have problems!

- Example: the following thread scheduled at high priority can make the system unusable

```
1 void *bad_bad_thread(void *arg)
2 {
3     while(1);
4 }
```

- So, we learned that:

- ◆ Real-time computation have to be limited (use real-time priorities only when **really needed!**)
- ◆ On sane systems, running applications with real-time priorities requires root privileges (or part of them!)