real-time operating systems course

4

introduction to POSIX pthread programming

introduction – thread creation, join, end - thread scheduling - thread cancellation – semaphores - thread mutexes and condition variables

introduction to POSIX pthread programming

the POSIX standard

- is an IEEE standard that specifies an operating system interface similar to most UNIX systems
- the standard 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 in that address space
- a **thread** is a single flow of control within a process
  - every process has at least one thread, the "`main()`" thread; its termination ends the process
  - all the threads share the same address space, and have a separate stack

the pthread library

- the pthread primitives are usually implemented into a pthread library
- all the declarations of the primitives cited in these slides can be found into `sched.h`, `pthread.h` and `semaphore.h`
- use `man` to get on-line documentation
- when compiling under gcc & GNU/Linux, remember the `-lpthread` option!

thread creation, join, end

thread body

- a thread is identified by a C function, called body:

```
void *my_thread(void *arg)
{
   ...
}
```

- 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 finish

thread creation

- thread can be created using the primitive

```
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
  - stack size and address
  - detach state (joinable or detached)
  - scheduling parameters (priority, …)
- 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 itself by calling

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

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

- two thread IDs can be compared using

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

joining a thread

- a thread can wait the termination of another thread using

```
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 task **must** be joined
- the join frees all the internal resources (stack, registers, and so on)

joining a thread (2)

- a thread which does not need to be joined must be declared as detached.
- 2 ways:
  - the thread is created as detached using `pthread_attr_setdetachstate()`
  - the thread becomes detached by calling `pthread_detach()` from its body
- joining a detached thread returns an error

example 1

- filename: `ex_create.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

pthread scheduling

scheduling algorithms

- the POSIX standard specifies in `sched.h` *at least* two scheduling strategies, identified by the symbols `SCHED_FIFO` and `SCHED_RR`
  - also, the sporadic server has been added recently to the standard
- other scheduling policies may be supported by each particular implementation, under the symbol `SCHED_OTHER`

scheduling algorithms (2)

- 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**

scheduling algorithms (3)

- the running thread is scheduled following its policy
  - `SCHED_FIFO` means the thread is scheduled until it ends, it blocks or it is canceled
  - `SCHED_RR` means the thread is scheduled until it ends, it blocks, it is canceled or it consumes its quantum
    - the quantum sizeis implementation defined
  - `SCHED_OTHER` is implementation defined
    - usually it is a UNIX scheduler with aging

scheduling algorithms (4)

- real time protocols are supported using mutexes
  - Priority Ceiling
  - Priority Inheritance
  - not all the implementations support them
- POSIX leaves unspecified the scheduling order between threads belonging to different processes

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 preempt 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 2

- filename: `ex_rr.c`
- the demo explains the behavior of the RT priorities and of the other policies
- the `main()` thread creates an 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`
- the low priority thread is always scheduled in background

pthread cancellation

killing a thread

- a thread can be killed by 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*

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 canc. point
  - a complete list can be found into the POSIX Std

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 an 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

example 3

- filename: `ex_cancellation.c`
- highlights the behavior of:
  - asynchronous cancellation
  - deferred cancellation
- explains the cleanup handlers usage

semaphores

semaphores

- a semaphore is a counter managed with a set of primitives
- it is used for
  - synchronization
  - mutual exclusion
- POSIX Semaphores can be
  - unnamed (local to a process)
  - named (shared between processed through a file descriptor)

unnamed semaphores

- mainly used with multithread applications
- 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);
```

- if the counter is greater than 0 the thread does not block
  - `sem_trywait` never blocks
- `sem_wait` is a cancellation point

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 4

- filename: `ex_sem.c`
- in this example, semaphores are used to implement mutual exclusion in the output of a character in the console.

pthread mutexes

what is a POSIX mutex?

- think at a mutex as a binary semaphore used for **mutual exclusion**
  - with the restriction that 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

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 attributes (2)

```
int pthread_mutexattr_setprotocol
  (pthread_mutexattr_t *attr, int protocol);
```

- `protocol` can be `PTHREAD_PRIO_NONE`, `PTHREAD_PRIO_INHERIT`, `PTHREAD_PRIO_PROTECT`

```
int pthread_mutexattr_setprioceiling
  (pthread_mutexattr_t *attr, int pceiling);
```

- set the priority ceiling of a `PTHREAD_PRIO_PROTECT` mutex

mutex initialization

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

- initializes a mutex with a given mutex attribute

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

- destroys a mutex

mutex lock and unlock

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

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

example 5

- filename: `ex_mutex.c`
- this is example 4 written using mutexes instead of semaphores.

pthread condition variables

what is a POSIX condition variable?

- condition variables are used to enforce synchronization between threads
  - a thread into a mutex critical section can **wait on a condition variable**
  - when waiting, the mutex is automatically released and locked again at wake up
  - the synchronization point must be checked into a **loop**!

cancellation and mutexes

- mutexes are **not** cancellation points
- the condition wait **is** a cancellation point
- when a thread is killed while blocked on a condition variable, the mutex **is locked again** before dieing
  - a cleanup function must be used to protect the thread from a cancellation
  - if they are not used, the mutex is left locked, and no thread can use it anymore!

condition variable attribute

- attributes are used to initialize a condition variable

```
int pthread_condattr_init (pthread_condattr_t *attr);
int pthread_condattr_destroy (pthread_condattr_t
  *attr);
```

  - these functions initialize and destroy a condition variable

initializing and destroying a condition variable

- to be used, a condition variable must be initialized

```
int pthread_cond_init (pthread_cond_t *cond, const
  pthread_condattr_t *attr)
```

- …and destroyed when it is no more needed

```
int pthread_cond_destroy(pthread_cond_t *cond)
```

waiting for a condition

```
int pthread_cond_wait (pthread_cond_t *cond,
  pthread_mutex_t *mutex);
```

- every condition variable is implicitly linked to a mutex
  - given a condition variable, the mutex parameter must always be the same
- **note:** the condition wait must always be called into a loop protected by a cleanup handler!!!

signaling a condition

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- these functions wakes up at least one (signal) or all (broadcast) the thread blocked on the condition variable
- the thread should lock the associated mutex when calling these functions
- nothing happens if no thread is blocked on the condition variable

example 6

- filename: `ex_cond.c`
- this is Example 4 written using simulated semaphores obtained using mutexes and condition variables
- a simulated semaphore is composed by a counter, a mutex and a condition variable
- the functions lock the mutex to work with the counter and use the condition variable to block