Real Time Operating Systems and Middleware

POSIX Threads Synchronization

Luca Abeni

abeni@dit.unitn.it

Threads Synchronisation

- All the threads running in a process share the private resources of the process
- So, the natural way to synchronise threads is by using the *shared resources* paradigm
- In particular, there can be two kind of interactions between threads belonging to a process:
  - Cooperation, when different threads need to synchronise for providing a service (examples: mailbox, pipeline, etc...)
  - Competition, when different threads need a shared resource for their execution, and the shared resource cannot be accessed by more than 1 thread at time (example: video output)

Competition

- Two threads need a *shared resource* to perform some action
- The resource must be accessed in *mutual exclusion* (simultaneous accesses from different threads are not allowed)
  - Example: the two threads need to print a file $\rightarrow$ if mutual exclusion is not enforced, the two printings are interleaved
- Code accessing the shared resource: *critical section*
  - Two threads cannot execute in critical section (for the same resource) simultaneously
- Mutual exclusion must be enforced by some kind of synchronisation mechanism

Cooperation

- A complex algorithm can be *parallelised*, by splitting it in a set of parallel activities
  - A parallel algorithm can take advantage of SMP
  - A parallel algorithm can be simpler
- Each one of such activities is executed in a thread
- Each thread:
  - Works on the data produced by another thread
  - Or produces data for another thread
- When the data needed by a thread is not ready, the thread must block
- When a thread τ_1 finishes producing data for a blocked thread τ_2, τ_2 must woken up

Enforcing Mutual Exclusion: Mutexes

- Mutexes: synchronisation objects used to enforce *mutual exclusion* in critical sections
  - Each critical section **must** be protected by a mutex
  - $1 \to 1$ mapping between mutexes and critical sections
- A mutex is similar to a binary semaphore
  - Mutex == mutual exclusion semaphore
  - Has two states: *locked* and *unlocked*
  - Internal *binary* counter, can be 0 (locked) or 1 (unlocked)
  - Two possible operations
    - `lock()`: enters the critical section
    - `unlock()`: exits the critical section

Mutex Operations

- `lock(m)`:
  - If mutex `m` is unlocked, lock it (decrease the internal counter) and continue
  - If mutex `m` is locked (the counter is 0), block until `m` is unlocked
- `unlock(m)`:
  - If mutex `m` is unlocked (the counter is 1), error
  - If mutex `m` is locked (the counter is 0), unlock it (increase the counter) and wake up blocked threads
- A mutex must be locked to acquire a shared resource (entering the critical section), before accessing it, and must be unlocked when the access to the shared resource is terminated

Mutexes and Semaphores

- A semaphore provides *generic* synchronization
  - The semaphore counter can be initialized to a generic value
- A mutex explicitly provides the concept of critical section (can be only used for mutual exclusion)
  - The mutex counter is always automatically initialized to 1
  - A mutex can be unlocked only by the thread that locked it
- $\Rightarrow$ Mutexes are less powerful, but can help preventing programming errors
- Mutexes can support real-time resource sharing protocols

POSIX Mutexes

- In POSIX, a mutex is identified by a descriptor, of type `pthread_mutex_t`
- A mutex must be initialized before using it
- The `pthread_mutex_init()` function can be used to initialize a mutex
- When initializing a mutex, a structure of type `phtread_mutexattr_t` can be used to describe the mutex attributes
  - Real-time resource protocol eventually used by the mutex
  - Priority of the highest priority thread that can try to lock the mutex (for HLP-like protocols)

POSIX Mutex Initialisation / Destruction

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

- Returns 0 in case of success, $\neq 0$ in case of error
- The mutex descriptor is returned in `mutex`
- If standard attributes are used, `attr` can be `NULL`
- An initialised mutex can be destroyed by calling `pthread_mutex_destroy()`

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

Other POSIX Mutex Operations

- POSIX provides the usual `lock` and `unlock` operations, but adds a *non blocking lock* operation
- Non blocking lock (called `trylock`) works as follows:
  - If the mutex is unlocked, lock it (decreasing the counter to 0) and continue
  - If the mutex is already locked, fail without blocking (but returning an error)

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

- Note that `pthread_mutex_lock()` is **not** a cancellation point

Cooperation Between Threads

- Mutexes solve the competition problem (provide mutual exclusion for competing threads)...
- ...But are not generic synchronisation objects
  - Mutexes cannot be used for synchronising cooperating threads
- A different synchronization object (with different primitives) is needed
  - Think about *monitors*
  - They guarantee mutual exclusion between methods...
  - ...But they also provide a way to *wait for some kind of condition to be verified*
- Condition Variables!!!

Condition Variables

- A condition variable is a synchronisation object on which a thread can sleep waiting for a condition to be true
- A condition variable is always associated to a mutex
  - It is possible to sleep on a condition variable only inside a critical section
  - Before blocking on a condition variable, a thread must acquire (lock) the associated mutex
- When a thread blocks on a condition variable, the associated mutex is released (unlocked)
- When a thread blocked on a condition variable is woken up, some different options are possible

Waking up from a Condition

- To wake up a thread τ_1 blocked on a condition, a thread τ_2 must lock the associated mutex first
- Some unblocking semantics are possible:
  - τ_2 unlocks the mutex, and τ_1 acquires it immediately
  - The mutex locking is “transferred” from τ_2 to τ_1, and τ_2 blocks on the mutex
  - τ_1 is unblocked and inserted in the mutex queue. When τ_2 will unlock the mutex, τ_1 will eventually compete for it with other threads
  - ...
- POSIX implements the last solution
- Note that when τ_1 is woken up and locks the mutex again, the condition might be false again...

Waking up – 2

```
    thread1()                           thread2()                  thread3()
/*...*/
<lock mutex>
<Is C true?>
<NO: block on cond var>
/*mutex is released)*/
                            /* ... */
                            <lock mutex>
                            <C is now true>
                            <Wake up thread1>
/*contending for mutex*/    /* ... */
                            <unlock mutex>
                                                        /* ... */
                                                        <lock mutex>
                                                        <Make C false>
                                                        <unlock mutex>
/* ... */
<lock mutex>
/* BUT C IS FALSE AGAIN!!! */
```

Solution: `thread1` has to test the condition again

POSIX Condition Variables

```
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *cond_attr)
int pthread_cond_destroy(pthread_cond_t *cond)
```

- Identified by a descriptor of type `pthread_cond_t`
- Initialized by calling `pthread_condition_init()`
- Destroyed by calling `pthread_condition_destroy()`
- As usual, attributes can be used in the `_init()` function
  - To create a default condition variable, you can set `cond_attr` to `NULL`

Blocking on a Condition Variable

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

- A thread can block on a condition by calling `pthread_cond_wait()`
  - Note that it must first lock the associated mutex
- Remember: after waking up, the condition must be checked again!!!
- We cannot check the condition with `if()`: a `while()` cycle is needed

```
pthread_mutex_lock(&m);
/* ... */
while (!c) {
  pthread_cond_wait(&cond_var, &m);
}
/* ... */
pthread_mutex_unlock(&m);
```

Waking up from a Condition Variable

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

- A thread can wake up:
  - One thread blocked on a condition, by calling `pthread_cond_signal()`
  - All the threads blocked on a condition, by calling `pthread_cond_broadcast()`
  - Note that it must first lock the associated mutex `mutex`
- If no thread is blocked on `cond`, nothing happens
  - A condition variable is not a semaphore!!!

Cancellation Problems

- As usual, things are more complex than expected...
  - As said, `pthread_mutex_lock()` is **not** a cancellation point...
  - ...But `pthread_cond_wait()` **is**!!!
- If a thread is killed while blocked on a condition variable, the associated mutex **is locked again** before dying...
  - The thread dies, the mutex is locked, and noone can lock it anymore!!!
- A cleanup handler *must* be used to protect a thread sleeping on a condition variable