Real-Time Programming
Lecture8-Multithread programming

Rola El Osta

CCNE Department
Lebanese University

I. Thread Priority and Scheduling

Sometimes in a multithreading application, the threads must execute according to their importance and a specific order to execute them must be chosen: it is the problematic of assigning the priorities to the threads and the type of their scheduling. This part is a problem generally treated in the field of real-time systems. To assign a priority and scheduling type to a thread, you must change its attribute *attr* to type *pthread_attr_t*. For this we will need the following:

1. The declaration of the Posix data structure used to assign a priority to the thread to be created:

struct sched_param param

2. The function that forces the operating system to take into account the different parameters (priority and type of scheduling) that the thread to be created will acquire:

int pthread_attr_setinheritsched(pthread_attr_t *attr, **int** inheritsched)

The different values that the *inheritsched* variable can take, are:

• **PTHREAD_EXPLICIT_SCHED**: The thread to be created will be forced to use the scheduling parameters contained in its ***attr*** property;

• **PTHREAD_INHERIT_SCHED**: The thread to be created will inherit the scheduling properties of its creator process. Regardless of the scheduling parameters specified in ***attr***, it will ignore them if this option is used.

The default value of the inheritsched attribute is **PTHREAD_INHERIT_SCHED**.

3. The function that assigns to the ***attr*** property of the thread to be created, its priority (which is an integer) contained in the ***param*** variable:

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

4. The function that assigns to the ***attr*** property of the thread to be created, its scheduling type noted ***policy***:

int pthread_attr_setschedpolicy(pthread_attr_t *attr, **int** policy)

There are several types of corresponding scheduling under Posix at the following values:

SCHED_FIFO: First-in first-out (FIFO) scheduling. Each thread has a fixed priority; when multiple threads have the same priority level, they run to completion in FIFO order (that is, in the order of their activations).

SCHED_RR: Round-robin (RR) scheduling. Each thread has a fixed priority; a thread uses a quantum of time and then moves to the tail of the priority queue.

When multiple threads have the same priority level, they run for a fixed time slice in FIFO order.

SCHED_OTHER: Each thread has a initial priority that is dynamically modified by the scheduler, according to the thread's activity; thread execution is time-sliced (time-shared). On other systems, this scheduling policy may be different.

5. The function to assign scheduling parameters to a running job. It is used to assign the scheduling parameters of the parent process that creates all other threads, or directly in the body of the thread:

int pthread_setschedparam(pthread_t thread, **int** policy, **const struct** sched_param *param)

The following source code presents a small program with 2 threads that takes into account the scheduling parameters (priority and type of scheduling).

See fifo.c

II. Problem of Priority Inversion

II.A Description of problem

Priority inversion is the problem that shows the fact that a thread prevents the execution of another thread of higher priority than it. It is usually a consequence of the data-sharing problem between threads presented lately (mutex solution). This part is a problem generally treated in the field of real-time systems. To better illustrate this problem, consider the following scenario:

There are three threads *th1*, *th2*, *th3* and a mutex called *lock*. Suppose these threads are such that priority (th3)> priority (th2)> priority (th1). *Th1* and *th3* wanted to access the same resource at different times. Initially, the threads *th3* and *th2* are asleep. Thread *th1* starts its execution first and seizes the lock mutex. Then, the higher priority thread th3 wakes up and preempts *th1*, executes and then tries to take the lock mutex. But it crashes (blocks) and goes to sleep again, because *th1* has already taken the lock.

Following this, thread th1 wakes up and continues executing. Shortly thereafter, thread *th2* (which never uses the lock) has a higher priority than *th1*, wakes up too and preempts *th1*. The result is the following:

- The thread th2 executes;

- The thread th1 is preempted, but holds (keeps) the lock mutex;

- The thread th3 is blocked and asleep waiting for the lock mutex.

The fact that the thread th3 is blocked while the thread th2 (with less priority than it) is running, is called the problem of priority inversion.

II.B Problem solution

To solve the problem of the priority inversion, a priority corresponding to that of the highest priority thread (*th3*) must be assigned to the lock mutex. You must also associate the threads with an inheritance protocol for the lock priority. Thus, the following scenario will be the one that avoids the inversion of priority:

- thread th1 starts its execution, obtains the lock and inherits the priority of the latter;

- thread th3 wakes up, preempts th1 (which inherits a priority equal to its own) and starts its execution, but blocks and goes back to the lock already taken by th1;

- thread th1 takes again the hand and continues its execution in its critical section;

- thread th2 wakes up in its turn but can not begin its execution, because the thread th1 running is the highest priority (th1 has inherited the priority of the lock which is that of the highest priority thread of the system) ;

- When th1 releases the lock, it loses the priority of this lock and resumes its initial priority (the lowest priority in our example);

- The th2 thread already ready can not run because the th3 thread that was stuck (blocked) on the lock has higher priority than it. It is th3 who takes the hand and finishes its execution;

- Finally, th2 runs and ends. If th1 had not finished its execution by releasing the lock, it takes again the hand and ends also.

- In addition to the functions already presented in mutex, here are the elements you need to know to solve this problem:

1. The declaration of the property to assign to the mutual exclusion mutex (the lock):

```
pthread_mutexattr_t m_attr;
```

2. The initialization function of the mutex property. Mandatory before any manipulation of the mutex attribute:

```
int pthread_mutexattr_init(pthread_mutexattr_t *m_attr)
```

3. The function that allows a mutex to be shared by threads belonging to any process. For this, the *pshared* variable must be equal to **PTHREAD_PROCESS_SHARED**. If the share is internal to the process (creator thread of the mutex), then *pshared* must be set to **PTHREAD_PROCESS_PRIVATE**:

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *m_attr, int pshared)
```

4. The function allowing to put in the *m_attr* property that will be assigned to the mutex, its ceiling priority *prioceiling* :

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *m_attr, **int** prioceiling)

pthread_mutex_setprioceiling() changes the priority ceiling, *prioceiling*, of a mutex, *mutex*. **pthread_mutex_setprioceiling()** locks a mutex if unlocked, or blocks until **pthread_mutex_setprioceiling()** successfully locks the mutex, changes the priority ceiling of the mutex and releases the mutex. The process of locking the mutex need not adhere to the priority protect protocol.

5. The function to put in the m_attr property that will be assigned to the mutex, its inheritance protocol *protocol*. In our case, since any thread taking hold of the mutex must inherit its priority, *protocol* will be worth **PTHREAD_PRIO_PROTECT**. There are also other values for the protocol variable **PTHREAD_PRIO_NONE** and **PTHREAD_PRIO_INHERIT**:

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *m_attr, **int** protocol)

6. The function that allows a thread with the propriety *attr* to inherit a priority. To do this, the *scope* variable will be worth **PTHREAD_SCOPE_SYSTEM**, if the thread competes (in concurrency) with any other thread in any system process. However, the scope variable will be worth **PTHREAD_SCOPE_PROCESS**, if the thread is only competing with internal threads in the same process:

int pthread_attr_setscope(pthread_attr_t *attr, **int** scope)

- The source code below shows the implementation of the problem solving scenario of the priority reversal problem for the three previous threads.
- See PI.c

III. Manage the periodicity of a thread

- In a multithreading application, some threads may run periodically. That is, a thread th1 with a period of p time units must always start a new execution after the lapse of this time. In some applications, this periodicity must be very strict and rigorous. In this case, the application must run directly using the time provided by the system clock.
- It is therefore interesting to see how this can be handled under Posix. In the previous examples, you may have noticed that the usleep (int p) function was used to wait for a thread for a period of time p. But for reasons of clock drift that we do not need to detail here, this technique is not rigorous in the management of the periodicity of the thread. In this part, without considering the proprietary solutions Posix not generally portable, we present a technique to manage the strict periodicity of the threads. The following elements are to know:

1. The declaration of the time management data structure (it uses the declaration of the time.h library):

struct timespec time;

Structure holding an interval broken down into seconds and nanoseconds.

Member objects

time_t tv_sec	whole seconds (valid values are >= 0)
long tv_nsec	nanoseconds (valid values are [0, 999999999])

2. The function to retrieve the time of the system clock (specified clock *clk_id)*:

int clock_gettime(clockid_t clk_id, **struct** timespec *time)

The variable clk_id can take the following values: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID. Be careful, many of these primitives do not work on Windows.

CLOCK_REALTIME
System-wide realtime clock. Setting this clock requires appropriate privileges.
CLOCK_MONOTONIC
Clock that cannot be set and represents monotonic time since some unspecified starting point.
CLOCK_PROCESS_CPUTIME_ID
High-resolution per-process timer from the CPU.
CLOCK_THREAD_CPUTIME_ID
Thread-specific CPU-time clock.

Return Value

clock_gettime(), **clock_settime**() and **clock_getres**() return 0 for success, or -1 for failure (in which case *errno* is set appropriately).

3. The mutex manipulation functions and condition variable that we presented previously plus the following function:

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *verrou, **struct** timespec *time)

This function uses a timeout on the time *time* to wake up the sleeping thread on waiting for the *cond* condition variable that will never be signaled.

Reminder:

- The *pthread_cond_timedwait*() and *pthread_cond_wait*() functions shall block on a condition variable. They shall be called with *mutex* locked by the calling thread or undefined behavior results.
- These functions atomically release *mutex* and cause the calling thread to block on the condition variable *cond*; atomically here means "atomically with respect to access by another thread to the mutex and then the condition variable". That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to *pthread_cond_broadcast*() or *pthread_cond_signal*() in that thread shall behave as if it were issued after the about-to-block thread has blocked.
- Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.

- The code below shows an example illustrating the setting up of a periodic task.
- See Periodictask.c

Go to Practice exercises!