

Programming RT systems with pthreads

Giuseppe Lipari

<http://www.lifl.fr/~lipari>

CRISAL - University de Lille 1

October 4, 2015

Outline

- 1 Timing utilities
- 2 Periodic threads
- 3 Scheduler selection
- 4 Resource Contention
 - Critical Sections
- 5 Mutex and Conditions
- 6 Priority Inheritance and Ceiling
- 7 Exercises

Timing handling in POSIX

- A time value is handled with different data structures and variable times, depending on the use and scope
- The “most standard” way to store time values for real-time processing is through the timespec structure

```
// defined in <time.h>

struct timespec {
    time_t    tv_sec;    // seconds
    long     tv_nsec;   // nanoseconds
}
```

- time_t is usually an integer (32 bits) that stores the time in seconds
- this data type can store both absolute and relative time values

Operations with timespec

- It is very common to perform operation on timespec values. Unfortunately, the standard library does not provide any helper function to do such kind of operations.
- An example of two common operation follows (see file time_utils.h and time_utils.c)

Example

```
void timespec_add_us(struct timespec *t, long us)
{
    t->tv_nsec += us*1000;
    if (t->tv_nsec > 1000000000) {
        t->tv_nsec = t->tv_nsec - 1000000000; // + ms*1000000;
        t->tv_sec += 1;
    }
}

int timespec_cmp(struct timespec *a, struct timespec *b)
{
    if (a->tv_sec > b->tv_sec) return 1;
    else if (a->tv_sec < b->tv_sec) return -1;
    else if (a->tv_sec == b->tv_sec) {
        if (a->tv_nsec > b->tv_nsec) return 1;
        else if (a->tv_nsec == b->tv_nsec) return 0;
        else return -1;
    }
}
```

Getting the time

- To get/set the current time, the following functions are available:

```
#include <time.h>

int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

- These functions are part of the Real-Time profile of the standard
- (in Linux these functions are part of a separate RT library)
- `clockid_t` is a data type that represents the type of real-time clock that we want to use

Clocks

- `clock_id` can be:
 - `CLOCK_REALTIME` represent the system real-time clock, it is supported by all implementations. The value of this clock can be changed with a call to `clock_settime()`
 - `CLOCK_MONOTONIC` represents the system real-time since startup, but cannot be changed. Not every implementation supports it
 - if `_POSIX_THREAD_CPUTIME` is defined, then `clock_id` can have a value of `CLOCK_THREAD_CPUTIME_ID`, which represents a special clock that measures execution time of the calling thread (i.e. it is increased only when a thread executes)
 - if `_POSIX_THREAD_CPUTIME` it is possible to get a special `clock_id` for a specific thread by calling `pthread_getcpuclockid()`

```
#include <pthread.h>
#include <time.h>

int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);
```

Sleep functions

- To suspend a thread, we can call the following functions

```
#include <unistd.h>

unsigned sleep(unsigned seconds);
```

```
#include <time.h>

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

- The first one only accepts seconds;
- The second one is part of the POSIX real-time profile and has a high precision (depends on the OS)
- `rqtp` represents the **interval of time** during which the thread is suspended
- if the thread is woke up before the interval has elapsed (for example, because of the reception of a signal), the `clock_nanosleep` will return -1 and the second parameter will contain the remaining time

Example of usage - I

examples/nanosleepexample.c

```
void *thread(void *arg)
{
    struct timespec interval;

    interval.tv_sec = 0;
    interval.tv_nsec = 500 * 1000000; // 500 msec
    while(1) {
        // perform computation
        nanosleep(&interval, 0);
    }
}
```

Example of usage - II

- The previous example does not work!

examples/nanosleepexample2.c

```
void *thread(void *arg)
{
    struct timespec interval;
    struct timespec next;
    struct timespec rem;
    struct timespec now;

    interval.tv_sec = 0;
    interval.tv_nsec = 500 * 1000000; // 500 msec
    clock_gettime(&next);
    while(1) {
        // perform computation
        timespec_add(&next, &interval); // compute next arrival
        clock_gettime(&now); // get time
        timespec_sub(&rem, &next, &now); // compute sleep interval
        nanosleep(&rem, 0); // sleep
    }
}
```

Problems

- Once again, it does not work!
 - It could happen that the thread is preempted between calls to `clock_gettime` and `nanosleep`,
 - in this case the interval is not correctly computed
- The only “clean” solution is to use a system call that performs the above operations atomically

Correct implementation

```
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
                   const struct timespec *rqtp, struct timespec *rmtp);
```

- This is the most flexible and complete function for suspending a thread (only available in the POSIX RT profile)
- `clock_id` is the clock id, usually `CLOCK_REALTIME`
- `flags` is used to specify if we want to suspend for a relative amount of time, or until an absolute point in time. It can be `TIMER_ABSTIME`, or 0 to mean relative interval
- `rqtp` is a pointer to a `timespec` value that contains either the interval of time or the absolute point in time until which the thread is suspended (depending on the flag value)
- `rmtp` only makes sense if the flag is 0. If the function is interrupted by a signal, this parameter will contain the remaining interval of sleeping time

Example

examples/periodicslides.c

```
struct periodic_data {
    int index;
    long period_us;
    int wcet_sim;
};

void *thread_code(void *arg) {
    struct periodic_data *ps = (struct periodic_data *) arg;
    int j; int a = 13, b = 17;
    struct timespec next;
    struct timespec now;

    clock_gettime(CLOCK_REALTIME, &next);
    while (1) {
        timespec_add_us(&next, ps->period_us);

        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                        &next, NULL);

        for (j=0; j<ps->wcet_sim; j++) a *= b;
    }
    return NULL;
}
```

Deadline miss detection

- The following code is used to detect a deadline miss (in this case, the behaviour is to abort the thread)

examples/periodicslides2.c

```
void *thread_code(void *arg) {
    struct periodic_data *ps = (struct periodic_data *) arg;
    int j;
    int a = 13, b = 17;
    struct timespec next, now;

    clock_gettime(CLOCK_REALTIME, &next);
    while (1) {
        clock_gettime(CLOCK_REALTIME, &now);
        timespec_add_us(&next, ps->period_us);

        if (timespec_cmp(&now, &next) > 0) {
            fprintf(stderr, "Deadline miss for thread %d\n", ps->index);
            fprintf(stderr, "now: %d sec %ld nsec      next: %d sec %ldnsec \n",
                    now.tv_sec, now.tv_nsec, next.tv_sec, next.tv_nsec);
            exit(-1);
        }

        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                        &next, NULL);

        for (j=0; j<ps->wcet_sim; j++) a *= b;
    }
    return NULL;
}
```

Scheduling policy

- It is possible to specify the policy and the parameters by using the thread attributes before creating the thread

```
#include <pthread.h>

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

Input arguments:

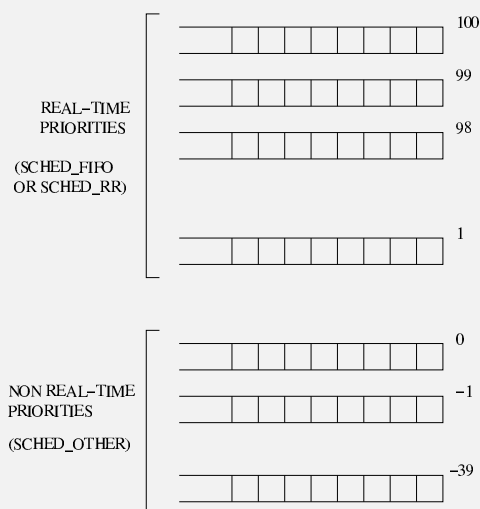
a attributes

policy can be SCHED_RR, SCHED_FIFO (fixed priority scheduling with or without round-robin) or SCHED_OTHER (standard Linux scheduler).

- **IMPORTANT:** to use the real-time scheduling policies, the user id of the process must be root.

Scheduling in POSIX

- The scheduling policies in POSIX:



Example

```
pthread_t th1, th2, th3;
pthread_attr_t my_attr;
struct sched_param param1, param2, param3;

pthread_attr_init(&my_attr);
pthread_attr_setschedpolicy(&my_attr, SCHED_FIFO);

param1.sched_priority = 1;
param1.sched_priority = 2;
param1.sched_priority = 3;

pthread_attr_setschedparam(&my_attr, &param1);
pthread_create(&th1, &my_attr, body1, 0);

pthread_attr_setschedparam(&my_attr, &param2);
pthread_create(&th2, &my_attr, body2, 0);

pthread_attr_setschedparam(&my_attr, &param3);
pthread_create(&th3, &my_attr, body3, 0);

pthread_attr_destroy(&my_attr);
```

Warning

- It is important to underline that only the superuser (root) can assign real-time scheduling parameters to a thread, for security reasons.
- if a thread with SCHED_FIFO policy executes forever in a loop, no other thread with lower priority can execute on the same processor

Setting scheduling priority

- To dynamically set thread scheduling and priority, use the following functions:

```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy,
                      const struct sched_param *param);
int sched_setparam(pid_t pid, const struct sched_param *param);
```

Input arguments:

- `pid` id of the process (or thread) on which we want to act
- `policy` the new scheduling policy
- `param` the new scheduling parameters (priority)

Resources

- A resource can be
 - A HW resource like a I/O device
 - A SW resource, i.e. a data structure
 - In both cases, access to a resource must be regulated to avoid interference
- example 1
 - If two processes want to print on the same printer, their access must be sequentialised, otherwise the two printing could be intermingled!
- example 2
 - If two threads access the same data structure, the operation on the data must be sequentialized otherwise the data could be inconsistent!

Mutual Exclusion Problem

- We do not know in advance the relative speed of the processes
 - hence, we do not know the order of execution of the hardware instructions
- Recall the example of incrementing variable x
 - incrementing x is not an atomic operation
 - atomic behaviour can be obtained using interrupt disabling or special atomic instructions

Example 1

```
/* Shared memory */  
int x;
```

```
void *threadA(void *)  
{  
    ...;  
    x = x + 1;  
    ...;  
}
```

```
void *threadB(void *)  
{  
    ...;  
    x = x + 1;  
    ...;  
}
```

- Bad Interleaving:

```
...  
LD   R0, x      (TA)  x = 0  
LD   R0, x      (TB)  x = 0  
INC  R0         (TB)  x = 0  
ST   x, R0      (TB)  x = 1  
INC  R0         (TA)  x = 1  
ST   x, R0      (TA)  x = 1  
...
```

Example 2

```
// Shared object (sw resource)
class A {
    int a;
    int b;
public:
    A() : a(1), b(1) {};
    void inc() {
        a = a + 1; b = b + 1;
    }
    void mult() {
        b = b * 2; a = a * 2;
    }
} obj;
```

```
void * threadA(void *)
{
    ...
    obj.inc();
    ...
}
```

```
void * threadB(void *)
{
    ...
    obj.mult();
    ...
}
```

Consistency:
After each operation, $a == b$

Resource in a non-consistent state!!

```
a = a + 1;    TA    a = 2
b = b * 2;    TB    b = 2
b = b + 1;    TA    b = 3
a = a * 2;    TB    a = 4
```

Consistency

- For any resource, we can state a set of **consistency properties**
 - A consistency property C_i is a boolean expression on the values of the **internal variables**
 - A consistency property must hold before and after each operation
 - It does not need to hold during an operation
 - If the operations are properly sequentialized, the consistency properties will always hold
- Formal verification
 - Let R be a resource, and let $C(R)$ be a set of consistency properties on the resource
 - $C(R) = \{C_i\}$
 - A concurrent program is correct if, for every possible interleaving of the operations on the resource, $\forall C_i \in C(R), C_i$ holds.

Critical sections

- the shared object where the conflict may happen is a **resource**
- the parts of the code where the problem may happen are called **critical sections**
- a critical section is a sequence of operations that cannot be interleaved with other operations on the same resource
- two critical sections on the same resource must be properly sequentialized
- we say that two critical sections on the same resource must execute in **MUTUAL EXCLUSION**
- there are three ways to obtain mutual exclusion
 - implementing the critical section as an atomic operation
 - disabling the preemption (system-wide)
 - selectively disabling the preemption (using semaphores and mutex)

Implementing atomic operations

- In single processor systems
 - disable interrupts during a critical section
 - non-voluntary context switch is disabled!
- Limitations:
 - if the critical section is long, no interrupt can arrive during the critical section
 - consider a timer interrupt that arrives every 1 msec.
 - if a critical section lasts for more than 1 msec, a timer interrupt could be lost
 - It must be done only for very short critical section;
 - Non voluntary context switch is disabled during the critical section
 - Disabling interrupts is a very low level solution: it is not possible in user space.

```
CLI;  
<critical section>  
STI;
```

Atomic operations on multiprocessors

- Disabling interrupts is not sufficient
 - disabling interrupts on one processor lets a thread on another processor free to access the resource
- Solution: use `lock()` and `unlock()` operations
 - define a flag `s` for each resource, and then surround a critical section with `lock(s)` and `unlock(s)`;

```
int s;  
...  
lock(s);  
<critical section>  
unlock(s);  
...
```

Disabling preemption

- On single processor systems
 - in some scheduler, it is possible to disable preemption for a limited interval of time
- problems:
 - if a high priority critical thread needs to execute, it cannot make preemption and it is delayed
 - even if the high priority task does not access the resource!

```
disable_preemption();  
<critical section>  
enable_preemption();
```

no context switch may happen during the critical section, but interrupts are enabled

Producer / Consumer model

- Mutual exclusion is not the only problem
 - we need a way of synchronise two or more threads
- example: producer/consumer
 - suppose we have two threads,
 - one produces some integers and sends them to another thread (PRODUCER)
 - another one takes the integer and elaborates it (CONSUMER)



A more general approach

- We need to provide a general mechanism for synchronisation and mutual exclusion
- Requirements
 - Provide mutual exclusion between critical sections
 - Avoid two interleaved insert operations
 - (semaphores, mutexes)
 - Synchronise two threads on one condition
 - for example, block the producer when the queue is full
 - (semaphores, condition variables)

Mutex generalities

- A mutex is a special kind of binary semaphore, with several restrictions:
 - It can only be used for mutual exclusion (and not for synchronization)
 - If a thread locks the mutex, only the same thread can unlock it!
- Advantages:
 - It is possible to define RT protocols for scheduling, priority inheritance, and blocking time reduction
 - Less possibility for errors

Mutex creation and usage

```
#include <pthread.h>

pthread_mutex_t m;

int pthread_mutex_init(pthread_mutex_t *m,
                      const pthread_mutex_attr_t *attr);

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

- lock corresponds to a wait on a binary semaphore
- unlock corresponds to a post on a binary semaphore
- a mutex can be initialized with attributes regarding the resource access protocol

Example with mutexes

examples/mutex.c

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

pthread_mutex_t mymutex;

void *body(void *arg)
{
    int i, j;

    for (j=0; j<40; j++) {

        pthread_mutex_lock(&mymutex);
        for (i=0; i<1000000; i++);
        for (i=0; i<5; i++) fprintf(stderr, "%s", (char *)arg);
        pthread_mutex_unlock(&mymutex);

    }

    return NULL;
}
```

Example continued

examples/mutex.c

```
int main()
{
    pthread_t t1,t2,t3;
    pthread_attr_t myattr;
    int err;

    pthread_mutexattr_t mymutexattr;

    pthread_mutexattr_init(&mymutexattr);
    pthread_mutex_init(&mymutex, &mymutexattr);
    pthread_mutexattr_destroy(&mymutexattr);

    pthread_attr_init(&myattr);
    err = pthread_create(&t1, &myattr, body, (void *)".");
    err = pthread_create(&t2, &myattr, body, (void *)"#");
    err = pthread_create(&t3, &myattr, body, (void *)"o");
    pthread_attr_destroy(&myattr);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);

    printf("\n");

    return 0;
}
```

Condition variables

- To simplify the implementation of critical section with mutex, it is possible to use **condition variables**
- A condition variable is a special kind of synchronization primitive that can only be used together with a mutex

```
#include <pthread.h>

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

- A call to `pthread_cond_wait()` is equivalent to:
 - release the mutex
 - block on the condition
 - when unblock from condition, lock the mutex again

Condition variables

- To unblock a thread on a condition

```
#include <pthread.h>

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

- The first one unblocks one thread blocked on the condition
- The second one unblocks all threads blocked in the conditions

More on conditions

- A condition variable **is not** a semaphore
 - internally, there is a queue of blocked threads
 - however, unlike the semaphore there is no counter
 - hence, if a thread calls `pthread_cond_signal` and there is no blocked thread on the condition, *nothing happens*
 - Vice-versa, a call to `pthread_cond_wait` is always a blocking call

Example with conditions

- Let's implement a synchronization barrier with mutex and condition variables
 - A synch barrier can synchronize up to N thread on one point
 - it has only one method, `synch()`
 - the first N-1 threads that call `synch()` will block, the N-th will unblock all previous threads

Example with conditions

examples/synch.cpp

```
class SynchObj {
    pthread_mutex_t m;
    pthread_cond_t c;
    int nblocked;
    int nthreads;
public:
    SynchObj(int n);

    void synch();
};

SynchObj::SynchObj(int n)
{
    nthreads = n;
    nblocked = 0;
    pthread_mutex_init(&m, 0);
    pthread_cond_init(&c, 0);
}
```

Example continued

examples/synch.cpp

```
void SynchObj::synch()
{
    pthread_mutex_lock(&m);

    nblocked++;

    if (nblocked < nthreads)
        pthread_cond_wait(&c, &m);
    else {
        nblocked = 0;
        pthread_cond_broadcast(&c);
    }

    pthread_mutex_unlock(&m);
}
```

Exercise

- Suppose we want to guarantee that a set of N periodic threads are activated at the same time (i.e. their first instance all arrive at the same time)
- When calling `pthread_create`, the thread is immediately active, so we cannot guarantee synchronicity
- We must implement this behavior manually
 - Every thread, will initially block on a condition
 - when the manager (the `main()`) calls a function, all threads are waken up at the same time, and get the same value of the arrival time

Design the data structure

examples/synchperiodic.h

```
#ifndef __SYNCHPERIODIC_H__
#define __SYNCHPERIODIC_H__

#include <time.h>
#include <pthread.h>

class PeriodicBarrier {
public:
    // constructor, initialize the object
    PeriodicBarrier(int n);

    // called by the threads for initial synch,
    // returns the same arrival time for all threads
    void wait(struct timespec *a);

    // called by the manager thread
    void start();

private:
    struct timespec arrival;

    int nthreads;
    int blocked;

    pthread_mutex_t m;
    pthread_cond_t c_threads;
    pthread_cond_t c_manager;
};

#endif
```

Implementation

examples/synchperiodic.cpp

```
#include "synchperiodic.h"

PeriodicBarrier::PeriodicBarrier(int n) :
    nthreads(n), blocked(0)
{
    pthread_mutex_init(&m, 0);
    pthread_cond_init(&c_threads, 0);
    pthread_cond_init(&c_manager, 0);
}

void PeriodicBarrier::wait(struct timespec *a)
{
    pthread_mutex_lock(&m);
    blocked++;
    if (blocked == nthreads)
        pthread_cond_signal(&c_manager);
    pthread_cond_wait(&c_threads, &m);
    *a = arrival;
    pthread_mutex_unlock(&m);
}

void PeriodicBarrier::start()
{
    pthread_mutex_lock(&m);
    if (blocked < nthreads)
        pthread_cond_wait(&c_manager, &m);

    pthread_cond_broadcast(&c_threads);
    clock_gettime(CLOCK_REALTIME, &arrival);
    pthread_mutex_unlock(&m);
}
```

Thread code

examples/exsynchper.cpp

```
PeriodicBarrier pb(NTHREADS);

void *thread_code(void *arg) {
    struct periodic_data *ps = (struct periodic_data *) arg;
    struct timespec next;

    fprintf(stdout, "TH %d waiting for start\n", ps->index);

    pb.wait(&next);

    while (1) {
        fprintf(stdout, "TH %d activated at time %ld\n", ps->index,
            next.tv_nsec/1000);

        waste(ps->wacet_sim);

        timespec_add_us(&next, ps->period_us);
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
            &next, NULL);
    }
    return NULL;
}
```

Exercise

- Modify the previous code to add an offset to the periodic threads
- Modify the previous code to add a “stop” mechanism (i.e. the manager thread can stop all periodic threads by pressing a key on the keyboard)
 - Hint: modify the data structure such that the wait() is called every instance, and add a stop() function

Setting protocol attributes

- With mutexes it is possible to set the priority inheritance or priority ceiling protocol
- This can be done on each semaphore separately by using the pthread_mutexattr_t attributes

```
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *
                                restrict attr, int *restrict protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
                                int protocol);
```

- where the protocol can be PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT, for no protocol, priority inheritance or priority ceiling, respectively

Priority Ceiling

- when specifying `PTHREAD_PRIO_PROTECT`, it is necessary to specify the priority ceiling of the mutex with the following function

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

- where `prioceiling` is the ceiling of the semaphore

Example with priority inheritance

- In this example, we create 2 mutex semaphores with priority inheritance

```
pthread_mutexattr_t mymutexattr;  
  
pthread_mutexattr_init(&mymutexattr);  
pthread_mutexattr_setprotocol(&mymutexattr, PTHREAD_PRIO_INHERIT);  
pthread_mutex_init(&mymutex1, &mymutexattr);  
pthread_mutex_init(&mymutex2, &mymutexattr);  
pthread_mutexattr_destroy(&mymutexattr);
```

- Notice that we can reuse the same attributes for the 2 semaphores
- Of course, the usage of the mutex remains the same (i.e. `lock()` and `unlock()` where appropriate)

Example with priority ceiling

- In this example, we create 2 mutex semaphores with priority ceiling

```
pthread_mutexattr_t mymutexattr;

pthread_mutexattr_init(&mymutexattr);
pthread_mutexattr_setprotocol(&mymutexattr, PTHREAD_PRIO_PROTECT);
pthread_mutexattr_setprioceiling(&mymutexattr, 10);
pthread_mutex_init(&mymutex1, &mymutexattr);
pthread_mutexattr_setprioceiling(&mymutexattr, 15);
pthread_mutex_init(&mymutex(2, &mymutexattr);
pthread_mutexattr_destroy(&mymutexattr);
```

- In this case, the first mutex (mymutex1) has priority ceiling equal to 10 (i.e. the highest priority task that accesses this semaphore has priority 10)
- the second mutex (mymutex2) has priority 15

Some exercise

- 1 Modify the periodic thread example so that a periodic thread can tolerate up to N consecutive deadline misses. Write an example that demonstrate the functionality
- 2 Modify the periodic thread example so that the period can be modified by an external manager thread. Write an example that demonstrates the functionality
- 3 (Dual priority) Modify the periodic thread example so that each thread is assigned 2 priorities and:
 - The first part of the code runs at “low” priority
 - The last part of the code executes at “high” priority
- 4 Write a “chain” of threads, so that each thread can start executing only when the previous one has completed its job
- 5 Which solution is better for the dual priority scheme? the chain of two tasks of modifying the priority on the fly?