Real Time Operating Systems and Middleware

Real-Time Programming Interfaces

Luca Abeni
abeni@dit.unitn.it
Needs for a Real-Time Interface

Real-Time applications might need to:
- Implement a periodic / sporadic behaviour
- Schedule themselves with fixed priorities (RM, DM, etc...)
- Disable paging for their memory (or disable mechanisms that introduce unpredictabilities)

Which Application Programming Interface (API) is needed?
- Which are the requirements for real-time applications?
- For example: is the standard Unix API enough?
- How should we extend it to support real-time applications?
A Real-Time API

- API: Application Programming Interface
- Source code interface
- Provides functions, data structures, macros, ...
- Specified in a *programming language*
  - We use C

Of course, we want to use a *standard* API

- A program written by using a standard API can be easily ported to new architectures (often, a simple recompilation is needed)

Refrasing our previous question: is any standard API capable to support real-time applications?
POSIX

- **POSIX:** Portable Operating System Interface
  - Family of IEEE / ISO / IEC standards defining the API, services, and standard applications provided by a *unix like* OS
  - Original standard: IEEE 1003.1-1988; today, more than 15 standards

- **Real-Time POSIX:** POSIX.1b, Real-time extensions
  - Priority Scheduling
  - Clocks and Timers, Real-Time Signals
  - ...
Implementing Periodic Tasks

- Clocks and Timers can be used for implementing periodic tasks

```c
void *PeriodicTask(void *arg)
{
    <initialization>;
    <start periodic timer, period = T>;
    while (cond) {
        <read sensors>;
        <update outputs>;
        <update state variables>;
        <wait next activation>;
    }
}
```

- How can it be implemented using the C language?
- Which kind of API is needed to fill the following blocks:
  - <start periodic timer>
  - <wait next activation>
Sleeping for the Next Job

First idea: on job termination, sleep until the next release time

<wait next activation>:

- Read current time
- $\delta = \text{next activation time} - \text{current time}$
- `usleep(\delta)`

```c
void wait_next_activation(void)
{
    gettimeofday(&tv, NULL);
    d = nt - (tv.tv_sec * 1000000 + tv.tv_usec);
    nt += period; usleep(d);
}
```
Problems with Relative Sleeps

Preemption can happen in `wait_next_activation()`.

- If preemption happens between `gettimeofday()` and `usleep()`...
- ...The task ends up sleeping for the wrong amount of time!!

- Correctly sleeps for 2ms
- Sleeps for 2ms; should sleep for 0.5ms
Using Periodic Signals

The “relative sleep” problem can be solved by a call implementing a periodic behaviour.

Unix systems provide a system call for setting up a periodic timer:

```c
setitimer(int which, const struct itimerval *value, struct itimerval *ovalue)
```

- **ITIMER_REAL**: timer fires after a specified real time. **SIGALRM** is sent to the process.
- **ITIMER_VIRTUAL**: timer fires after the process consumes a specified amount of time.
- **ITIMER_PROF**: process time + system calls.

<start periodic timer> can use setitimer()
Using Periodic Signals - setitimer()

```c
#define wait_next_activation pause

static void sighand(int s)
{
}

int start_periodic_timer(uint64_t offs, int period)
{
    struct itimerval t;

    t.it_value.tv_sec = offs / 1000000;
    t.it_value.tv_usec = offs % 1000000;
    t.it_interval.tv_sec = period / 1000000;
    t.it_interval.tv_usec = period % 1000000;

    signal(SIGALRM, sighand);

    return setitimer(ITIMER_REAL, &t, NULL);
}

Try [www.dit.unitn.it/~abeni/RTOS/periodic-1.c](http://www.dit.unitn.it/~abeni/RTOS/periodic-1.c)
```
Enhancements

The previous example uses an empty handler for SIGALRM.

This can be avoided by using `sigwait()`:

```c
int sigwait(const sigset_t *set, int *sig)
```

- Select a pending signal from `set`
- Clear it
- Return the signal number in `sig`
- If no signal in `set` is pending, the thread is suspended
```c
void wait_next_activation(void) {
    int dummy;
    sigwait(&sigset, &dummy);
}

int start_periodic_timer(uint64_t offs, int period) {
    struct itimerval t;
    t.it_value.tv_sec = offs / 1000000;
    t.it_value.tv_usec = offs % 1000000;
    t.it_interval.tv_sec = period / 1000000;
    t.it_interval.tv_usec = period % 1000000;
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGALRM);
    sigprocmask(SIG_BLOCK, &sigset, NULL);
    return setitimer(ITIMER_REAL, &t, NULL);
}
```
Enhancements

- Periodic timers have a **big** problem:
  - "Timers will never expire before the requested time, instead expiring some short, constant time afterwards, dependent on the system timer resolution"

- Try
  - www.dit.unitn.it/~abeni/RTOS/periodic-2.c
  - The period is **6ms** instead of **5ms**!!!
  - $HZ = 1000 \Rightarrow$ up to **1ms** error in itimer (accumulates)

- Solution: decrease **period** by half jiffy

```c
int start_periodic_timer(uint64_t offs, int period) {
  struct itimerval t;
  period -= 500;
  t.it_value.tv_sec = offs / 1000000;
  ...
```

Real Time Operating Systems and Middleware – p. 12
Clocks & Timers

Let’s look at the first `setitimer()` parameter:

- `ITIMER_REAL`
- `ITIMER_VIRTUAL`
- `ITIMER_PROF`

It selects the `timer`: every process has 3 interval timers

`timer`: abstraction modelling an entity which can generate events (interrupts, or signal, or asynchronous calls, or...)

`clock`: abstraction modelling an entity which provides the current time

- Clock: “what time is it?”
- Timer: “wake me up at time $t$”
POSIX Clocks & Timers

- The traditional Unix API provides each process with three interval timers, connected to three different clocks
  - Real time
  - Process time
  - Profiling

  ⇒ only one real-time timer per process!!

- POSIX (Portable Operating System Interface):
  - Different clocks (must provide at least CLOCK_REALTIME, can provide CLOCK_MONOTONIC)
  - Multiple timers per process (each process can dynamically allocate and start timers)
  - A timer firing generates an asynchronous event which is configurable by the program
POSIX Timers

- POSIX timers are per process
- A process can create a timer with `timer_create()`
  
  ```c
  int timer_create(clockid_t c_id, struct sigevent *e, timer_t *t_id)
  ```

  - `c_id` specifies the clock to use as a timing base
  - `e` describes the asynchronous notification to occur when the timer fires
  - On success, the ID of the created timer is returned in `t_id`

- A timer can be armed (started) with `timer_settime()`
  
  ```c
  int timer_settime(timer_t timerid, int flags,
                    const struct itimerspec *v, struct itimerspec *ov)
  ```

  - `flags`: TIMER_ABSTIME
POSIX Timers

- POSIX Clocks and POSIX Timers are part of RT-POSIX
- To use them in real programs, `librt` has to be linked
  1. Get
     
     www.dit.unitn.it/~abeni/RTOS/periodic-3.c
  2. `gcc -Wall periodic-3.c -lrt -o ptest`
  3. The `-lrt` option links `librt`, that provides
     `timer_create()`, `timer_settime()`, etc...

- On some distributions, `libc` does not properly support these “recent” calls ⇒ we can work around this problem by providing missing prototypes, etc... (see periodic-3.c)
```c
int start_periodic_timer(uint64_t offs, int period) {
    struct itimerspec t;
    struct sigevent sigev;
    timer_t timer;
    const int signal = SIGALRM;
    int res;

    t.it_value.tv_sec = offs / 1000000;
    t.it_value.tv_nsec = (offs % 1000000) * 1000;
    t.it_interval.tv_sec = period / 1000000;
    t.it_interval.tv_nsec = (period % 1000000) * 1000;

    sigemptyset(&sigset); sigaddset(&sigset, signal);
    sigprocmask(SIG_BLOCK, &sigset, NULL);

    memset(&sigev, 0, sizeof(struct sigevent));
    sigev.sigev_notify = SIGEV_SIGNAL; sigev.sigev_signo = signal;
    res = timer_create(CLOCK_MONOTONIC, &sigev, &timer);
    if (res < 0) {
        return res;
    }
    return timer_settime(timer, 0, &t, NULL);
}
```
Using Absolute Time

- POSIX clocks and timers provide *Absolute Time*
  - The “relative sleeping problem” can be easily solved
  - Instead of reading the current time and computing $\delta$ based on it, `wait_next_activation()` can directly wait for the *absolute* arrival time of the next job

- The `clock_nanosleep()` function must be used

```c
int clock_nanosleep(clockid_t c_id, int flags,
                    const struct timespec *rqtp,
                    struct timespec *rmtp)
```

- The `TIMER_ABSTIME` flag must be set
- The next activation time must be explicitly computed and set in `rqtp`
- In this case, the `rmtp` parameter is not important
Implementation with clock_nanosleep

```c
static struct timespec r;
static int period;

static void wait_next_activation(void)
{
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &r, NULL);
    timespec_add_us(&r, period);
}

int start_periodic_timer(uint64_t offs, int t)
{
    clock_gettime(CLOCK_REALTIME, &r);
    timespec_add_us(&r, offs);
    period = t;
    return 0;
}
```

- `clock_gettime` is used to initialize the arrival time.
- The example code uses global variables `r` (next arrival time) and `period`. Do not do it in real code!
Some Final Notes

- Usual example; periodic tasks implemented by sleeping for an absolute time:
  
  www.dit.unitn.it/~abeni/RTOS/periodic-4.c

- Exercize: how can we remove global variables?

- Summing up, periodic tasks can be implemented by
  
  - Using periodic timers
  
  - Sleeping for an absolute time

- Timers often have a limited resolution (generally multiple of a system tick)
  
  - In system’s periodic timers (itimer(), etc...) the error often sums up

- In modern systems, clock resolution is generally not a problem
Exercize: Cyclic Executive

- Implement a simple cyclic executive
  - Three tasks, with periods $T_1 = 50ms$, $T_2 = 100ms$, and $T_3 = 150ms$
  - Tasks’ bodies are in [www.dit.unitn.it/~abeni/RTOS/cyclic_test.c](http://www.dit.unitn.it/~abeni/RTOS/cyclic_test.c)

- Use the mechanism you prefer for implementing the periodic event (minor cycle)

Some hints:

- Compute the minor cycle
- Compute the major cycle
- So, we need a periodic event every ... $ms$
- What should be done when this timer fires?

Done? So, try $T_1 = 60ms$, $T_2 = 80ms$, and $T_3 = 120ms$
Remember?

<table>
<thead>
<tr>
<th>task</th>
<th>f</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>40 Hz</td>
<td>25 ms</td>
</tr>
<tr>
<td>B</td>
<td>20 Hz</td>
<td>50 ms</td>
</tr>
<tr>
<td>C</td>
<td>10 Hz</td>
<td>100 ms</td>
</tr>
</tbody>
</table>

\[ \Delta = \text{gcd} \quad \text{(minor cycle)} \]
\[ T = \text{lcm} \quad \text{(major cycle)} \]

**guarantee:**
\[
\left\{ \begin{array}{l}
C_A + C_B \leq \Delta \\
C_A + C_C \leq \Delta 
\end{array} \right.
\]