Real-Time Programming Interfaces

Real Time Operating Systems and Middleware

Luca Abeni

luca.abeni@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: **A**pplication **P**rogramming **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: **P**ortable **O**perating **S**ystem **I**nterface
  - 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
  - Interaction with "Single UNIX Specification" $\Rightarrow$ information available at `http://opengroup.org/onlinepubs/0096953`
- Real-Time POSIX: POSIX.1b, Real-time extensions
  - Priority Scheduling
  - Clocks and Timers, Real-Time Signals
  - ...

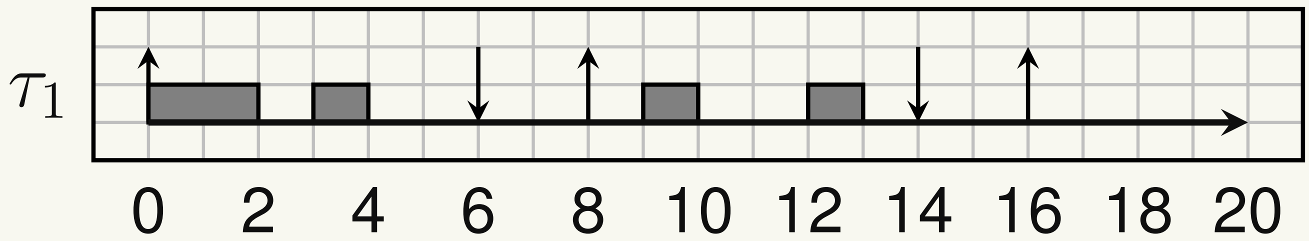
Implementing Periodic Tasks

- Clocks and Timers can be used for implementing peridic tasks

```
void *PeriodicTask(void *arg)
{
  <initialization>;
  <start periodic timer, period = T>;
  while (cond) {
    <job body>;
    <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

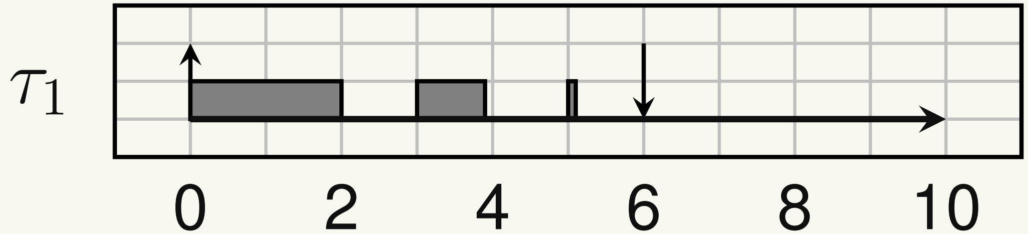


- First idea: on job termination, sleep until the next release time
- <wait next activation>:
  - Read current time
  - δ = next activation time - current time
  - usleep(δ)

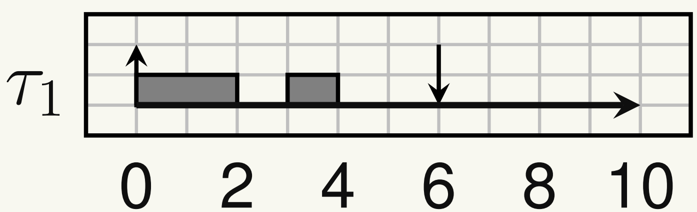
```
void wait_next_activation(void);
{
  gettimeofday(&tv, NULL);
  d = nt - (tv.tv_sec * 1000000 + tv.tv_usec);
  nt += period; usleep(d);
}
```

Problems with Relative Sleeps

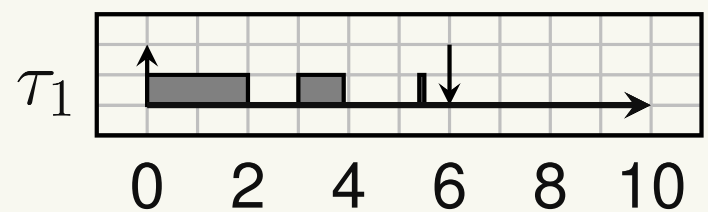
Preemption can happen in `wait_next_activation()`



- Preemption between `gettimeofday()` and `usleep()` $\Rightarrow$
- $\Rightarrow$ The task sleeps 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

```
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/periodic-1.c`

Enhancements

- The previous example uses an empty handler for `SIGALRM`
- This can be avoided by using `sigwait()`

```
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

setitimer() + sigwait()

```
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);
}
```

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 asyncrhonous 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

- Traditional Unix API three interval timers per process, connected to three different clocks
  - Real time
  - Process time
  - Profiling
- $\Rightarrow$ only one real-time timer per process!!!
- POSIX (Portable Operating System Interface):
  - Different clocks (at least `CLOCK_REALTIME`, `CLOCK_MONOTONIC` optional)
  - Multiple timers per process (each process can dynamically allocate and start timers)
  - A timer firing generates an asyncrhonous event which is configurable by the program

POSIX Timers

- POSIX timers are per process
- A process can create a timer with `timer_create()`

```
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()`

```
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.disi.unitn.it/~abeni/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 old distributions, libc does not properly support these “recent” calls $\Rightarrow$ some workaronds can be needed

POSIX Timers & Periodic Tasks

```
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 solved
  - Instead of reading the current time and computing δ 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

```
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

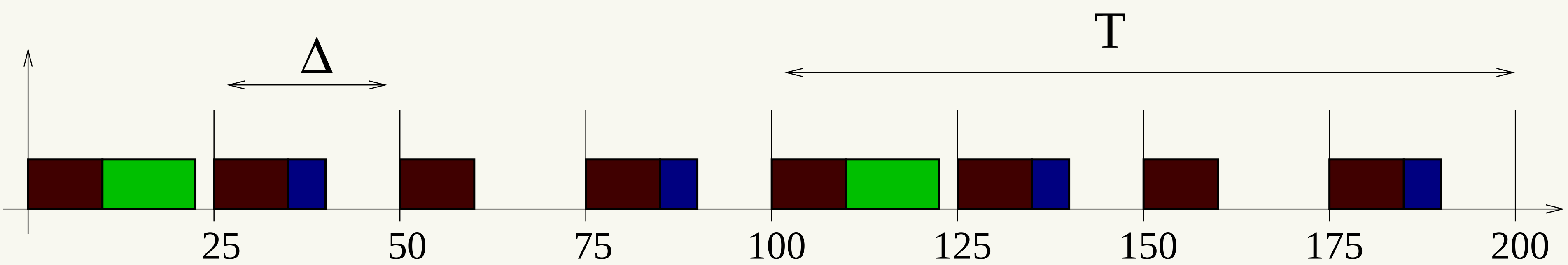
- Usual example; periodic tasks implemented by sleeping fo an absolute time:
  `www.dit.unitn.it/~abeni/periodic-4.c`
  - Exercize: how can we remove global variables?
- Summing up, periodic tasks can be implemented by
  - Using periodic timers
  - Sleeping for an absolute time
- Timers often have a 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
  - 3 tasks: $T_1 = 50ms$, $T_2 = 100ms$, and $T_3 = 150ms$
  - Tasks' bodies are in `www.dit.unitn.it/~abeni/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? Try $T_1 = 60ms$, $T_2 = 80ms$, $T_3 = 120ms$

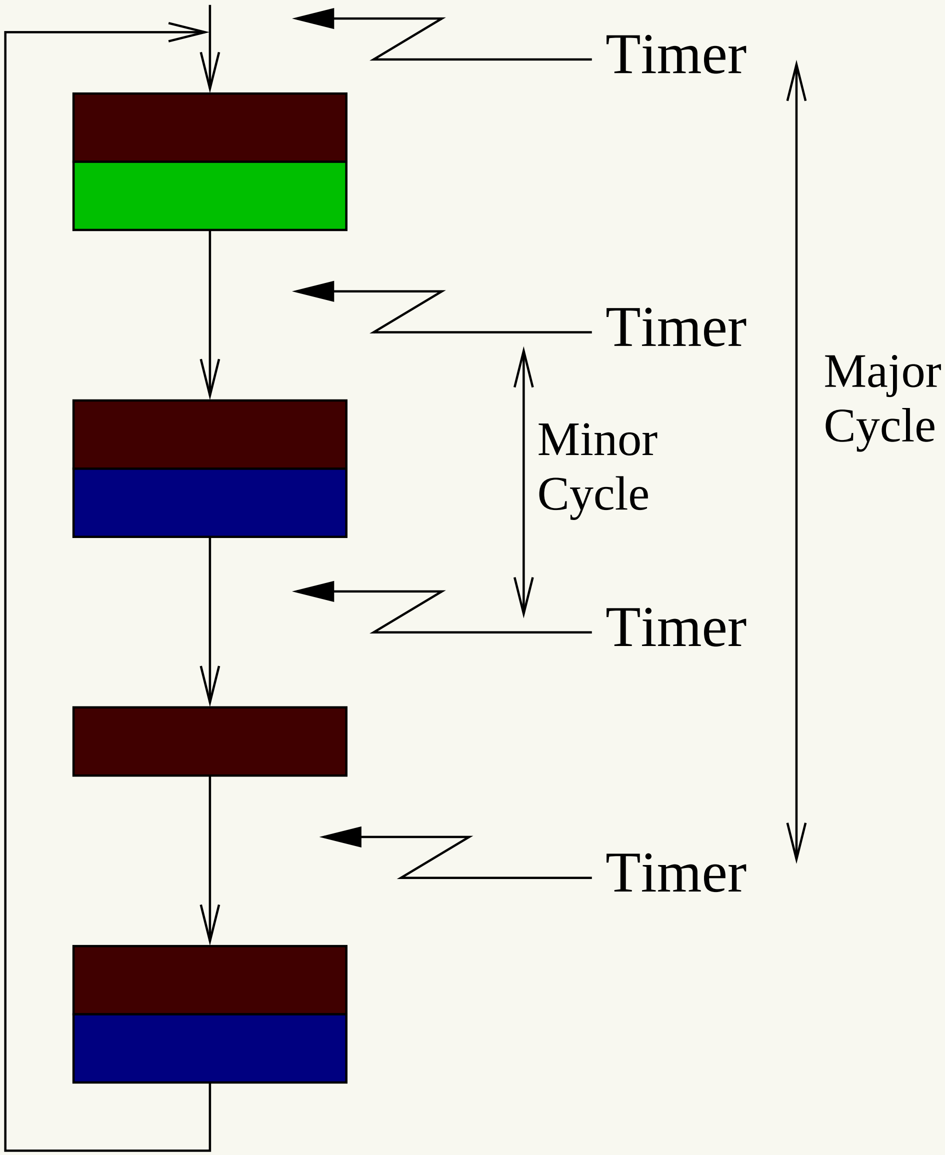
Remember?

- The schedule repeats every 4 minor cycles
  - τ_1 must be scheduled every $25ms \Rightarrow$ scheduled in every minor cycle
  - τ_2 must be scheduled every $50ms \Rightarrow$ scheduled every 2 minor cycles
  - τ_3 must be scheduled every $100ms \Rightarrow$ scheduled every 4 minor cycles



- First minor cycle: $C_1 + C_3 \leq 25ms$
- Second minor cycle: $C_1 + C_2 \leq 25ms$

Implementation



- Periodic timer firing every minor cycle
- Every time the timer fires...
- ...Read the scheduling table and execute the appropriate tasks
- Then, sleep until next minor cycle