Real-Time Programming

Luca Abeni
`luca.abeni@santannapisa.it`

March 15, 2021

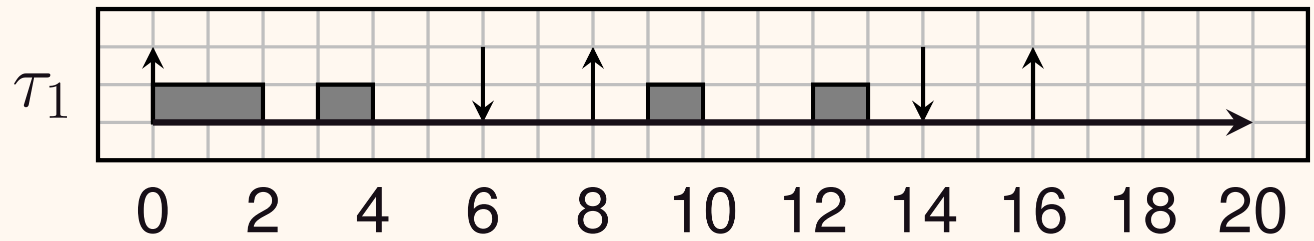
Implementing Periodic Tasks

- 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

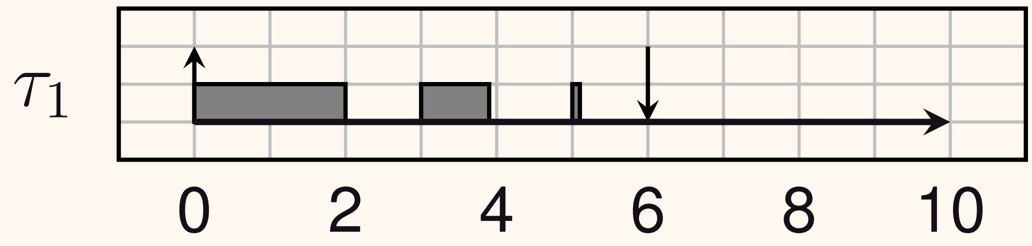


- On job termination, sleep until the next release time
- <wait next activation>:
  - Read current time
  - δ = next activation time - current time
  - usleep(δ)

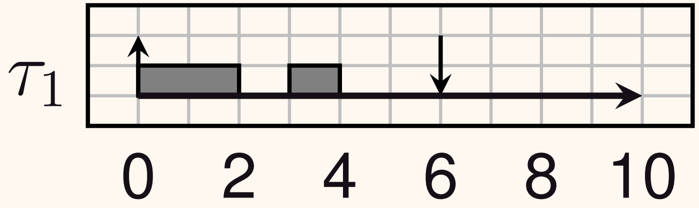
```
void wait_next_activation(void);
{
  gettimeofday(&tv, NULL);
  d = nt - (tv.tv_sec * 1000000 + tv.tv_usec);
  nt += period; usleep(d);
}
```

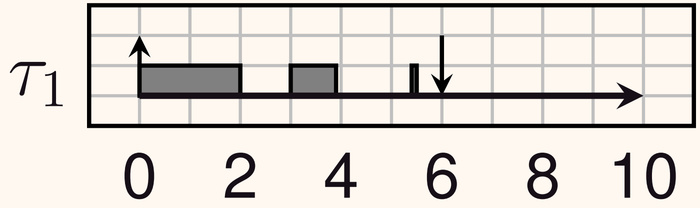
Problems with Relative Sleeps

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 *ovalu
```

  - 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()

```
#define wait_next_activation pause

static void sighand(int s)
{
}

int start_periodic_timer(uint64_t offs, int perio
{
  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);
}
```

Example Code

- Example code at
  - Various examples for all the code explained in these slides

`https://gitlab.retis.santannapisa.it/l.abeni/ExampleCode`

- For a `setitimer()` example, try `periodic-1.c`
  - Simple program creating a timer with period $5ms$
  - `start_periodic_timer()` and `wait_next_activation()` from previous slide

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

- Code: `periodic-2.c`

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
  - On success, ID of the created timer 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 itimersp
```

  - `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 `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 ⇒ 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) * 100
    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, &time
    if (res < 0) {
        return res;
```

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

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

static void wait_next_activation(void)
{
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &r
    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: `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

Real-Time Scheduling in POSIX

- POSIX provides support for Real-Time scheduling
- Priority scheduling
  - Multiple priority levels
  - A task queue per priority level
  - The first task from the highest-priority, non empty, queue is scheduled
- POSIX provides multiple scheduling policies
  - A scheduling policy describes how tasks are moved between the priority queues
  - Fixed priority: a task is always in the same priority queue

Real-Time Scheduling in POSIX

- POSIX specifically requires four scheduling policies:
  - `SCHED_FIFO`
  - `SCHED_RR`
  - `SCHED_SPORADIC`
  - `SCHED_OTHER`
- `SCHED_FIFO` and `SCHED_RR` have fixed priorities
- `SCHED_SPORADIC` is a *Sporadic Server* → decreases the response time for aperiodic real-time tasks
- `SCHED_OTHER` is the “traditional” Unix scheduler
  - Dynamic priorities
  - Scheduled in background respect to fixed priorities

Fixed Priorities - 1

- SCHED_FIFO and SCHED_RR use fixed priorities
  - They can be used for real-time tasks, to implement RM and DM
  - Remember: the application developer is in charge of assigning priorities to tasks!
  - Real-time tasks have priority over non real-time (SCHED_OTHER) tasks
- So... What is the difference between these two policies?
  - Only visible when more tasks have the same priority

Fixed Priorities - 2

- `SCHED_FIFO`: priority queues handled in FIFO order
  - When a task start executing, only higher priority tasks can preempt it
- `SCHED_RR`: time is divided in intervals
  - After executing for one interval, a task is removed by the head of the queue, and inserted at the end
- So, there is a difference only if multiple tasks have the same priority
  - Never do this!

SCHED_FIFO vs SCHED_RR

- Only one task per priority level $\rightarrow$ `SCHED_FIFO` and `SCHED_RR` behave the same way
- More tasks with the same priority
  - With `SCHED_FIFO`, the first task of a priority queue can starve other tasks having the same priority
  - `SCHED_RR` tries serve tasks having the same priority in a more fair way
- The round-robin interval (scheduling quantum) is implementation dependent
- RR and FIFO priorities are comparable. Minimum and maximum priority values can be obtained with `sched_get_priority_min()` and `sched_get_priority_max()`

Setting the Scheduling Policy

```
int sched_get_priority_max(int policy)
int sched_get_priority_min(int policy)

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

- If `pid == 0`, then the parameters of the running task are changed
- The only meaningful field of `struct sched_param` is `sched_priority`

Problems with Real-Time Priorities

- In general, "regular" (`SCHED_OTHER`) tasks are scheduled in background respect to real-time ones
- A real-time task can preempt / starve other applications
- Example: the following task scheduled at high priority can make the system unusable

```
void bad_bad_task()
{
    while(1);
}
```

  - Real-time computation have to be limited (use real-time priorities only when **really needed**!)
  - Running applications with real-time priorities requires root privileges (or part of them!)

Memory Swapping and Real-Time

- The *virtual memory* mechanism can swap part of the process address space to disk
  - Memory swapping can increase execution times unpredictabilities
  - Not good for real-time applications
- A real-time task can lock part of its address space in main memory
  - Locked memory cannot be swapped out of the physical memory
  - This can result in a DoS (physical memory exhausted!!!)
- Memory locking can be performed only by applications having (parts of) the root privileges!

Memory Locking Primitives

- `mlock()`: lock some pages from the process address space into main memory
  - Makes sure this region is always loaded in RAM
- `munlock()`: unlock previously locked pages
- `mlockall()`: lock the whole address space into main memory
  - Can lock the *current* address space only, or all the future allocated memory too
  - Can be used to disable "lazy allocation" techniques
- These functions are defined in `sys/mman.h`
  - Please check the manpages for details