Programming Real-Time Periodic Tasks in Linux

Real-Time Industrial Systems

Marcello Cinque

Roadmap

- Time abstractions
- Programming periodic tasks (and pthreads)
- Real-time scheduling in Linux
  - Fixed Priority (RM) Scheduling (SCHED_FIFO)
  - Resource management with priority inheritance and priority ceiling
  - EDF Scheduling (SCHED_DEADLINE)

- References:
  - L. Abeni. “Periodic Timers in modern OS”
  - https://gitlab.retis.santannapisa.it/l.abeni/ExampleCode/
  - G. Lipari. “Programming RT systems with pthreads”
  - https://pubs.opengroup.org/onlinepubs/009695399/idx/realtime.html
  - https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html
  - Linux manpages

Time Abstractions

Clocks and timers

- **Clock**: abstraction modelling and entity which provides the current time
  - Clock: what time is it?
  - Counts the time passed from a given past reference called *epoch* (e.g., microseconds since 1st January 1970)
- **Timer**: abstraction modelling an entity which can generate events at a given time (interrupts, signals, …)
  - Timer: wake me up at time *t*

Types of timers

- **One-shot**:
  - A one-shot timer is a timer that is armed with an initial expiration time, either relative to the current time or at an absolute time (based on some timing base, such as time in seconds and nanoseconds since the Epoch). The timer expires once and then is disarmed.
- **Periodic**:
  - A periodic timer is a timer that is armed with an initial expiration time, again either relative or absolute, and a repetition interval. When the initial expiration occurs, the timer is reloaded with the repetition interval and continues counting.

Getting the time in UNIX

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

- The *tv* argument is a *struct timeval*

```
struct timeval {
        time_t tv_sec; /* seconds */
        suseconds_t tv_usec; /* microseconds */
};
```

1 secondo = 1000000 microsecondi (1e+6)
1 secondo = 1000000000 nanosecondi (1e+9

- and gives the number of seconds and microseconds since the Epoch

- Both the function and the structures are defined in <sys/time.h>

Timers in UNIX

- Traditional Unix API has three interval timers per process, connected to three different clocks:
  - **Real-time** (`ITIMER_REAL`): system real-time (wall) clock L'ora vera e propria
  - **Virtual** (`ITIMER_VIRTUAL`) : passage of virtual time, incremented only when the process is executing in user mode
  - **Profiling** (`ITIMER_PROF`): passage of virtual time plus the time the kernel is executing on behalf of the process virtual time + utilizzo del kernel
- → only 1 real-time timer per process!

Setting a timer

```
int setitimer(int which, const struct itimerval *new_value,
      struct itimerval *old_value);
```

- Sets a timer to fire at the values specified into the `new_value` field, of type `struct itimerval`: (all defined in <sys/time.h>)

```
struct itimerval {
        struct timeval it_interval; /* interval for periodic timer */
        struct timeval it_value; /* time until next expiration*/
};
```

Se voglio un timer che fire solo una volta metto in it_interval 0

- The `which` parameter specifies the clock type (e.g., `ITIMER_REAL`: in this case the `SIGALARM` signal is delivered to the process when the timer fires)

Time handling in RT-POSIX

- The library to adopt is <time.h> Non ctime.h ma time.h
- Time values are handled using the timespec structure, similar to `timeval`, but adopting nanoseconds

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

- The functions to get/set the time and to set timers are part of the RT-POSIX standard
- `itimerspec` to be use instead of `itimerval` to set timers
- Utility functions are not available and must be implemented by the user

An example of utility function

- Adding a certain amount `d` of microseconds to a timespec `t`

```
#define NSEC_PER_SEC 1000000000ULL   ULL= Unsigned Long Long
void timespec_add_us(struct timespec *t, uint64_t d) {
        d *= 1000;  Dobbiamo moltiplicare per avere macrosecond e noi
                    abbiamo nanosecond
        t->tv_nsec += d;
        t->tv_sec += t->tv_nsec / NSEC_PER_SEC;
        t->tv_nsec %= NSEC_PER_SEC;
}
```

- Similar functions can be done to subtract, to compare two timespecs, etc…

Getting/setting the time with POSIX

invece di gettimeofday()

```
#include <time.h>

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

`clock_id` can be:

- `CLOCK_REALTIME` represent the system real-time clock, 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 supported in all implementations
- 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)

RT-POSIX Timers

aggiungere il -lrt linking flag al gcc comand nella shel se non funzionano automaticamente questi comandi

- A process can create and start multiple timers
- A timer firing generates a signal event, configurable by the program

```
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 (CLOCK_REALTIME or CLOCK_MONOTONIC)
  - `e` describes the asynchronous notification
  - On success, ID of the created timer in `t_id`

- A timer can be armed (started) with:

```
int timer_settime(timer_t timerid, int flags, const struct itimerspec *v, struct
itimerspec *ov)
```

- flags: TIMER_ABSTIME sets the timer at the absolute time of v, otherwise it is relative to the call time
- Being part strictly of RT-POSIX, the –lrt linking flag needs to be used

Sleep functions

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
int usleep(useconds_t usec);
```

- Standard Unix functions, accepting seconds and microseconds

```
#include <time.h>

int nanosleep(const struct timespec *req, struct timespec *rem);
```

- RT-POSIX function, accepting nanoseconds
- `*req` is the amount of time required to sleep
- If the thread wakes up before time is elapsed (e.g., due to a signal) the `nanosleep` returns -1 and `*rem` will contain the remaining time

Sleep functions

```
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
                    const struct timespec *req, struct timespec *rem);
```

- Similar to `nanosleep`, but
  - It allows to specify the clock, e.g. `CLOCK_REALTIME`
  - It allows to sleep until an absolute future time, if `flags = TIMER_ABSTIME`
  - `*req` contains either the interval of time or the absolute point in time until which the thread is suspended (depending on the flag)
  - `*rem` makes sense only if the sleep time is relative (`flags = 0`)

Programming periodic tasks

Structure of a periodic task

Per il momento iniziamo dall'ipotesi di un solo process con un solo task

```
int main() {
        start_periodic_timer(2000000, 5000);
        while(1) {
                wait_next_activation();
                job_body();
        }
        return 0;
}
```

La durata meggiore di job_body() è il nostro WCT - Worst Case Time

- A periodic task starting after 2 seconds and cycling every 5 ms
- How can we implement `start_periodic_timer` and `wait_next_activation`?

Solution 1: sleep for the remaining time

- On instance termination, sleep for the remaining time until the next release
  - Read current time
  - Calculate delay = next activation time – current time
  - usleep(delay)

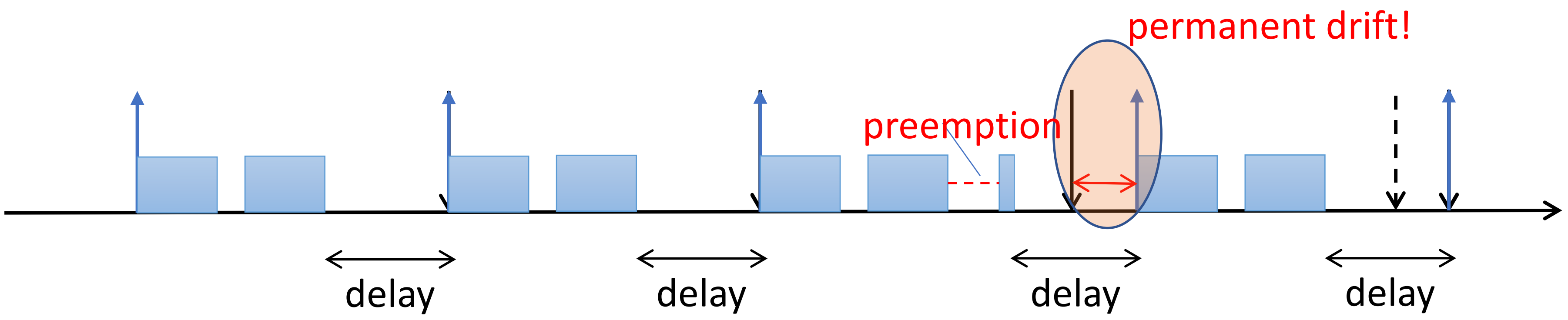
```
static long next_period;
static int period;

void start_periodic_timer(uint64_t offs, int t){
        struct timeval t1;
        gettimeofday(&t1,NULL);
        long now = t1.tv_sec*1000000+t1.tv_usec;
        next_period = now + offs;
        period = t;
}
```

```
void wait_next_activation(void) {
        struct timeval t1;
        gettimeofday(&t1,NULL);
        long now = t1.tv_sec*1000000+t1.tv_usec;
        long delay = next_period - now;
        next_period += period;
        usleep(delay);
}
```

Solution 1: not a good idea…

- Preemption can happen in `wait_next_activation()` between `gettimeofday()` and `usleep()`, causing a dangerous time drift in the task!!



Solution 2: using timers

- The «relative sleep» problem can be solved using periodic timers

```
#define wait_next_activation pause
static void sighand(int s){ }

void 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);

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

Non ci piace il SIGHAND vuoto perchè potremmo ricevere un signal alarm dall'esterno e svegliarci, in pratica non abbiamo controllo sul signal alarm

Solution 2: using timers

- wait_next_activation just pauses the task
- The task periodically receives a SIGALARM every period, after the first offs, that resumes it trough an empty handler (sighand)
- The empty handler can be avoided using sigwait in wait_next_activation, suspending the caller for the pending signal

- PROBLEMS:
  - Only 1 timer per process!
    - Can be solved using RT-POSIX timers
  - Overhead (and latency) introduced by the signal generation/handling
    - Need for a different strategy
  - Timers might have a limited resolution (multiple of a system tick, not a real problem in modern systems)

Libreria time.h

In questa soluzione complichiamo il codice, ma se avessi diversi processi e diversi timer settati così (signal and event) potrei incappare nell'overlap tra di essi, vedremo la solzuione 3 con il riutilizzo delle Sleep non relative ma assoluti

Solution 2: with RT-POSIX timers

Abbiamo ristretto la nostra WAIT a quel particolare signal implementato da noi

Ma abbiamo ancora il problema di un solo timer, se avessimo due thread avremo dei problemi

```
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);
                                                 Creamiamo l'evento
                                                 (sigev) e lo
        memset(&sigev, 0, sizeof(struct sigevent)); attacchiamo al timer
        sigev.sigev_notify = SIGEV_SIGNAL;
        sigev.sigev_signo = signal;
        res = timer_create(CLOCK_MONOTONIC, &sigev, &timer);
        if (res < 0) {perror("Timer Create"); exit(-1);}
        return timer_settime(timer, 0 /*TIMER_ABSTIME*/, &t, NULL);
}             Equivalente al setitemer() di slide 18
```

→ Events attached to the timer

Solution 3: clock_nanosleep

- The «relative sleep» problem can be solved by making the task directly wait for the *absolute* arrival time of the next job

Utilizziamo posix e quindi time.h, invece di gettimeofday() usiamo clock_gettime()

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

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

- `gettime` used to initialize the arrival time
- `timespec r` initialized to arrival time + `offs`

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

Sleep fino al prossimo r, in tempo assoluto, aggiungendo di volta in volta il periodo.

- `clock_nanosleep` makes the task sleep until the time specified in `r`
- `r` postponed on a `period` each cycle

The solution uses global variables for `r` and `period`. To be avoided in real code!

global variables not the best

Real-time scheduling

It is possible to schedule tasks with Rate Monotonic in Linux?

- All we need is a preemptive fixed priority (PFP) scheduler!
  - In such a case, it is sufficient to assign fixed priorities to tasks proportional to their frequencies

- The default Linux scheduler is not PFP
  - The default policy, SCHED_OTHER, is based on the CFS (Completely Fair Scheduler) algorithm
  - CFS is dynamic,and based on a time horizon (the *targeted latency*) within which every task is executed, starting from the task having the lowest *virtual runtime*

- *But*, Linux also includes other scheduling classes: SCHED_FIFO and SCHED_RR!

POSIX Scheduling policies

- 140 priority levels, the higher ones for RT
- SCHED_FIFO: task run to completion, can be preemted only by higher priority tasks
  - If a SCHED_FIFO task eecute forever in a loop all other low priority tasks will starve!
- SCHED_RR: task run to completion or till their timeslice ends
- SCHED_OTHER: also indicated as TS (Time Sharing) is the default scheduler (CFS in current Linux kernels)

- Only the superuser (root) can assign real-time priorities to a task, for security reasons
- Priorities can be remapped in different implemen-tations, e.g., from 0 to 140, RT starting from 41.

So, SCHED_FIFO can be used as the PFP scheduler we need!

How to assign real-time priorities to tasks

- Three possible approaches
  - Use the functions available in <sched.h>
  - Use the chrt command
  - Use pthread functions, to change scheduling class and priorities to threads setting the proper attributes before creating the thread

Non è una libreria Standard ma solo di Linux

- In all the three cases, the user must be root

Using set scheduler function

```
#include <sched.h>

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

struct sched_param {
               ...
               int sched_priority;
               ...
            };
```

- `pid`: the identifier of the task, if 0 it is the calling task
- `policy`: can be SCHED_FIFO, SCHED_RR or SCHED_OTHER
- `param`: used to set the priority from 0 to 99 (must be 0 for SCHED_OTHER)

- To check the new scheduling policy and priority, the `ps -eLfc` command can be used

Example

Guarda il codice periodic-task-3-FIFO.c che è uguale alla soluzione ma solo con l'aggiunta dello scheduler

Attenzione: deve essere eseguito con il comando sudo altrimenti non assegniamo la priorità, scriviamo:
sudo ./task-fifo

Se eseguiamo il comando ps -eLfc vediamo che passiamo dalla classe TS a quella FF e la priorità è aumetanta, diventa 51 perchè per i RT parte da 40+11 dato da noi

```
int main()
{
    struct sched_param sp;
    sp.sched_priority = 11;

    sched_setscheduler(0, SCHED_FIFO, &sp);

    start_periodic_timer(2000000, 5000);

    while(1) {
        wait_next_activation();
        job_body();
    }

    return 0;
}
```

Assegniamo priorità 11, sheduler FIFO, a questo process.
Per i thread sarebbe meglio utilizzare lo standard pthread

Using `chrt` command

```
chrt [options] priority command
chrt [options] -p priority <pid>
```

- `chrt` sets or retrieves the real-time scheudling attributes of a task (a new *command* or an existing pid)
- Among the options:
  - `-o, --other`: set the policy to SCHED_OTHER
  - `-r, --rr`: set the policy to SCHED_RR
  - `-f, --fifo`: set the policy to SCHED_FIFO
  - `-d, --deadline`: set the policy to SCHED_DEADLINE (see next)
  - `-p <pid>`: to retrieve (or set) the scheduling info of a task
- For instance:
  - `sudo chrt -f 11 ./task` launches «task» with SCHED_FIFO and priority 11
  - `sudo chrt -f -p 11 2677` sets SCHED_FIFO and priority 11 to the running task with pid 2677

Ad esempio se lanciamo il programma senza lo scheduler nel main possiamo assengnare la priorità direttamente dal lancio:
sudo chr -f 11 ./task
ottenendo lo steso risultato

Using pthreads functions

- Real-time scheduling and priority can be set by setting thread attributes in a variable `attr` of type `pthread_attr_t`
- Three settings to be performed on `attr`:
  1. setting the scheduling policy
  2. setting the thread priority
  3. forcing the explicit scheduling
- `attr` is then passed to `pthread_create` when starting the new thread

Abbiamo bisogno di modificare il codice della soluzione 3 in modo da non utilizzare più le variabili globali e passare all'uso dei pthread
guarda periodic-thread.c

Nell'esempio abbiamo spostato il codice del body all'interno della funzione run() del thread perchè si usano variabili statiche locali al thread stesso: aggiugni -pthread linker al comando gcc

pthreads functions: setting the policy

```
#include <pthread.h>

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

- Adds the scheduling policy to the thread attribute `attr`
- `policy`: can be `SCHED_FIFO`, `SCHED_RR` or `SCHED_OTHER`
- Returns 0 if no errors occur

pthreads functions: setting the priority

```
struct sched_param myparam;
myparam.sched_priority = <value>;
pthread_attr_setschedparam(&attr, &myparam);
```

- The `sched_param` structure is used to set the priority value (from 0 to 99) to be added to the thread attribute `attr` via the `pthread_attr_setschedparam` function

Facendo solo questo non risolveremmo il problema dell'assegnazione della priorità ai thread perché di defaul essi ereditano la priorità dal padre

pthread functions: setting explicit scheduling

Dobbiamo fare quest' ulteriore passaggio

- By default, the child thread inherits the scheduling attributes of the parent thread
- Explicit scheduling has to be set on the `attr` variable with:

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

- `inheritsched` can be:
  - `PTHREAD_INHERIT_SCHED` (default): threads that are created using `attr` inherit scheduling attributes from the creating thread; the scheduling attributes in `attr` are ignored.
  - `PTHREAD_EXPLICIT_SCHED`: threads that are created using `attr` take their scheduling attributes from the values specified by the attributes object.

Example

periodic-thread-FIFO.c ovviamente eseguito tramite sudo ./

Attenzione: se guardiamo tramite ps-eLfc nella prima colonna con i numeri vediamo due process con lo stesso PID, quello del padre che è il main e quello del thread figlio che effettivamente ha una priorità diversa ed è FF, l' indivudal PID del thread è dato dalla terza colonna con i numeri

- Spawning a FIFO thread with priority 11

```
pthread_t th;
pthread_attr_t myattr;
struct sched_param myparam;
pthread_attr_init(&myattr);

pthread_attr_setschedpolicy(&myattr, SCHED_FIFO);
myparam.sched_priority = 11;
pthread_attr_setschedparam(&myattr, &myparam);
pthread_attr_setinheritsched(&myattr, PTHREAD_EXPLICIT_SCHED);
pthread_create(&th, &myattr, thread_code, &thread_params);
pthread_attr_destroy(&myattr);
```

Excercise: write a `pthread_create_fifo` function to simplify the initialization

Real-time resource management with RT-POSIX

- How to access mutual exclusive resources with pthreads avoiding priority inversion?
- Use mutexes!

- Pthreads mutexes are compliant to RT-POSIX and can be set to work with priority inhertitance and with priority ceiling protocols

Setting mutex protocols

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

- Protocol can be:
  - `PTHREAD_PRIO_NONE` for no protocol
  - `PTHREAD_PRIO_INHERIT` for priority inheritance
  - `PTHREAD_PRIO_PROTECT` for priority ceiling
- In case of priority ceiling, the `ceiling` of the mutex has to be specified with:

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

Example

Esempio: thread-mutex-PI.c

- Setting priority inheritance to a mutex
  - The mutexattr can be used to initialize more than one mutex, if necessary

```
pthread_mutex_t mylock;
pthread_mutexattr_t mymutexattr;

pthread_mutexattr_init(&mymutexattr);
pthread_mutexattr_setprotocol(&mymutexattr,PTHREAD_PRIO_INHERIT);
pthread_mutex_init(&mylock, &mymutexattr);
pthread_mutexattr_destroy(&mymutexattr);
```

It is possible to schedule tasks with EDF in Linux?

- Since the Linux Kernel version 3.14, the SCHED_DEADLINE policy is available to schedule tasks with EDF
- It is a Linux-specific policy, not yet included in RT-POSIX
  - So, not yet available in pthreads functions
- The policy can be set to tasks using
  - `chrt` from the command line
  - or scheduling system calls in the code

Non possiamo utilizzare i pthread per farlo

SCHED_DEADLINE: rationale

- Regular tasks (`SCHED_OTHER`) are scheduled in background with respect to real-time ones
- Real-time tasks can starve other applications in case of bugs or bad programming
- Example: this task if scheduled as `SCHED_FIFO` with high priority can make a CPU core unstable

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

SCHED_DEADLINE: rationale

- Linux impelments a rt-trhottling protection mechanism that throttles bad tasks (removes them from the CPU)
  - Not clear how this interferes with real-time guarantees…
  - Can we use something more theoretically founded?

- What about aperiodic servers?

SCHED_DEADLINE: key idea

- Augment the basic EDF algorithm with a bandwidth reservation mechanism (specifically, the Constant Bandwidth Server – CBS) to isolate the behavior of real-time tasks between each other.
- The CBS coupled with EDF solves two problems:
  - Enables more appropriate "rt throttling" performed when the task finishes its budget, allowing feasibility analysis and removing interference in case of bad programming
  - Assigns dynamic deadlines to tasks, even when having periodic behavior
    - The kernel should assign deadlines to the individual jobs, but it is actually not aware of jobs! It only sees the task that voluntary "sleeps" until next period!

SCHED_DEADLINE: parameters

- Each task is bound to a CBS characterized by three parameters:
  - ***runtime***, ***period***, and ***deadline***
- Each SCHED_DEADLINE task should receive «*runtime*» microseconds of execution time every «*period*» and the allotted «*runtime*» is available within «*deadline*» microseconds from the start of the period
- The state of each task is described by a ***scheduling deadline*** (the current deadline d_s of its CBS) and a ***remaining runtime*** (the current budget c_s of its CBS)

SCHED_DEADLINE: rules

- When a SCHED_DEADLINE task wakes up (becomes ready for execution), the scheduler checks if:

$$\underbrace{\frac{remaining\ runtime}{scheduling\ deadline\ - current\ time}}_{\text{Remaining Budget}} > \underbrace{\frac{runtime}{period}}_{\text{Bandwith}}$$

- if the scheduling deadline is smaller than the current time, or above condition is verified, the scheduling deadline and the remaining runtime are re-initialized as
  - $scheduling\ deadline = current\ time + deadline$
  - $remaining\ runtime = runtime$
- otherwise, the scheduling deadline and the remaining runtime are left unchanged

SCHED_DEADLINE: rules

- When a task executes for an amount of time t, its *remaining runtime* is decreased as:
  - *remaining runtime = remaining runtime - t*

  (technically, the runtime is decreased at every tick, or when the task is preempted)

- When the remaining runtime becomes 0, the task is “throttled” and cannot be scheduled until its scheduling deadline. The “replenishment time” for this task is set to be equal to the current *scheduling deadline* (implementing a sort of hard CBS behavior)

- When the current time is equal to the replenishment time of a throttled task, the parameters are updated as:
  - *scheduling deadline = scheduling deadline + period*
  - *remaining runtime = remaining runtime + runtime*

SCHED_DEADLINE: how to set the parameters?

- Depends on the task model
- For classical Liu and Layland periodic (C_i, T_i) tasks, it has to be:

$$runtime \geq C_i \quad \text{and} \quad period \leq T_i$$

- For sporadic tasks with (C_i, D_i, T_i), where T_i is the minimum interarrival time and $D_i < T_i$, it has to be:

$$runtime \geq C_i \quad \text{and} \quad deadline = D_i \quad \text{and} \quad D_i < period \leq T_i$$

- SCHED_DEADLINE can be used also for aperiodic tasks and self-suspending tasks (so, basically arbitrary tasks but with a guaranteed *runtime/period* bandwith, assuring isolation)

Setting SCHED_DEADLINE with `chrt`

- Using `chrt` with `-d` or `--deadline` flag it is possible to assign the SCHED_DEADLINE scheduling class to a new or an existing task
- Scheduling parameters can be set with the following flags
  - `-T, --sched-runtime nanoseconds` to set the *runtime*
  - `-P, --sched-period nanoseconds` to set the *period*
  - `-D, --sched-deadline nanoseconds` to set the *deadline*
  - priority must be set to 0 (meaningless for SCHED_DEADLINE)
- Example:
  - `sudo chrt -d -T 1000000 -P 10000000 -D 5000000 0 ./task` launches «task» with SCHED_DEADLINE and the specified parameters

Questo cambierà lo Policy del thread specificato

Verrà schedulato il task con politica DLN e priorità 140 che è quella massima di Linux (SCHED_FIFO arriva massimo a priorità 139) ma che comunque, se non lo specifichiamo noi, non consumeremo il 100% della Banda della CPU

Setting SCHED_DEADLINE in the code

Linux primitive

- Using `sched_gettattr` and `sched_settattr` primitives:

il primo parametro è il PID del task, 0 consideriamo il chiamante. L'ultimo 0 sono i flag che non ci interessano

```
struct sched_attr attr;
sched_getattr(0, &attr, sizeof(attr), 0);
attr.sched_policy = SCHED_DEADLINE;
attr.sched_runtime = 500 * 1000;           //ns
attr.sched_deadline = 5 * 1000 * 1000;     //ns
attr.sched_period = 5 * 1000 * 1000;       //ns
sched_setattr(0, &attr, 0);
```

Note: these primitives are still not available in sched.h on some distributions; they need a wrapper (see `sched_attributes.h` in the examples)

Setting SCHED_DEADLINE in the code

```
struct sched_attr {
        u32 size;              /* Size of this structure */
        u32 sched_policy;      /* Policy (SCHED_*) */
        u64 sched_flags;       /* Flags */
        s32 sched_nice;        /* Nice value (SCHED_OTHER, SCHED_BATCH) */
        u32 sched_priority;    /* Static priority (SCHED_FIFO, SCHED_RR) */
        /* Remaining fields are for SCHED_DEADLINE */
        u64 sched_runtime;    /* nanoseconds */
        u64 sched_deadline;    /* nanoseconds */
        u64 sched_period;      /* nanoseconds */
};
```

Se vogliamo SCHED_DEADLINE per i thread e non solo per il main non possiamo usare i pthread_attr_setschedpolicy come il caso FIFO ma dobbiamo usare lo stesso file .h e settare i Parametri direttamente nella funzione del task