This post originally appeared on the SIGBED Blog in September 2020.

Liu and Layland and Linux: A Blueprint for "Proper" Real-Time Tasks

by Björn Brandenburg (MPI-SWS)

TLDR: a template for implementing periodic tasks on Linux — jump straight to the code.

Hi there! I hope you are enjoying the new SIGBED Blog as much as I am. One of the aspects that I love about the embedded and real-time systems landscape is its vast diversity in topics — there are few areas of computer science that do not feature in the embedded context in one way or another, with new papers routinely touching on just about anything from complexity theory to low-level nitty-gritty implementation details, and everything in between.

I think it's fun to see this diversity also reflected in the SIGBED Blog. So for a change of pace, today I wanted to have a look at something a bit more technical and hands-on — specifically, a primer on how to connect the models studied in the real-time systems literature to actual running code.

Scheduling Theory vs. Actual Code

If one is exploring the area of embedded real-time systems for the first time, or even if one has plenty of practical experience with time-critical code but is diving into the literature for the first time, the models typically studied in real-time papers can be a bit bewildering initially. In particular, the classic and widely studied *periodic task model* (due to Liu and Layland) and *sporadic task model* (due to Al Mok) can appear a wee bit "theoretical" at first — how can such simple models tell us anything about real software with all its complexities?

I know it certainly took me a while until it really "clicked" and I understood just how practical these models are. Of course, not every pile of "spaghetti code" can be captured with "nice" models, but it is certainly possible, even easy, to write code that matches the common assumptions in the scheduling literature well. That is, with just a little discipline on behalf of the programmers and software architects — which arguably they should be striving for anyway, clean code always wins in the long run — it is trivial to unlock the benefits of decades of research into time-critical computing. So there really isn't any reason not to do it: to obtain *reliable*, *predictable* systems, it's surely better to build on top of the "shoulders of giants" than to grab a machete and explore the wilderness left and right of the well-trodden paths…

So how to get started? I'm glad you asked — in this blog, I would like to share a blueprint for writing software in a Linux environment that can be analyzed with methods rooted in the classic periodic task model. But let's start at the beginning.

The Liu and Layland Model of Recurrent Time-Critical Processes

Many (if not most) embedded real-time systems can be seen as a bunch of activities that are carried out over and over at various rates. For example, an autonomous robot like a Roomba usually has a high-level navigation activity that, say, once a second plans the "big picture" — where is the robot located, where should it be going next, how should it get there? In addition, there are lower-level sensing, control, and steering activities that are carried out a couple of times per second to check whether any obstacles have suddenly appeared in front of the robot, whether the robot is pointed in the right direction to reach the next way point, whether a wheel is slipping due to a change in traction, etc. The distinguishing characteristics of such workloads are:

1. The activities are carried out repeatedly as long as the system runs (i.e., they are *recurrent*).
2. They are more or less *time-critical* — each activity must not be delayed "too much," where the acceptable magnitude of delay is of course highly specific to each application and setting.

There are countless ways in which systems that match this rough characterization can be implemented. For instance:

- In the case of robotics, it's common to use publish-subscribe frameworks such as ROS.
- Erlang's message-passing paradigm is popular for realizing latency-sensitive "soft" real-time applications.
- Arduino encourages a style of programming in which a "big main loop" drives all activity.
- Avionics software commonly relies on rigid time partitions.
- Automotive software based on AUTOSAR implements repeating activities as "runnables," which are then aggregated into larger tasks.

This list is of course not exhaustive, and as anyone who's been in the industry for a while can attest to, "in the wild" one can also find many ad-hoc solutions that mix and match techniques and approaches in creative ways.

However, not all implementation approaches are equally well-suited to understanding a system's temporal behavior, in particular its *worst-case* timing. Case in point, from personal experience I can attest to the fact that it can be frustratingly difficult to predict the timing of ROS applications even in the best of circumstances. Similarly, actor-based languages and frameworks such as Erlang or Akka, while certainly a powerful and fun way to program concurrent reactive systems, don't lend themselves easily to temporal analysis. It's one thing to build a system that reacts in a timely manner, it's another thing entirely to build it such that one can rigorously ascertain that it always does so.

Which brings us to Liu and Layland, who famously introduced a simple model to capture the essence (and *only* the essence) of recurrent time-critical workloads in an elegant way that enables powerful mathematical analyses. The rest, as they say, is history: in the roughly five decades since Liu and Layland's seminal work appeared, tens of thousands of papers have adopted, studied, and build on their *periodic task model* (or on one of its many generalized descendants). To benefit from this rich literature, we need to make sure our software conforms as well as possible to the model, so let's see what that takes.

Implementing a Periodic Task on Linux

You probably know this already, but just to be sure, let's quickly recap: the periodic task model describes each recurrent activity as a *task* that generates an infinite sequence of *jobs*, and the whole system as a set of tasks. Each job corresponds to one activation of a task (as illustrated in Figure 1 below). A system comprised of a number of recurrent activities is hence modeled simply as a set of tasks, where each task represents one repeating activity, and each instance (or *activation*) of an activity is a job to be executed.

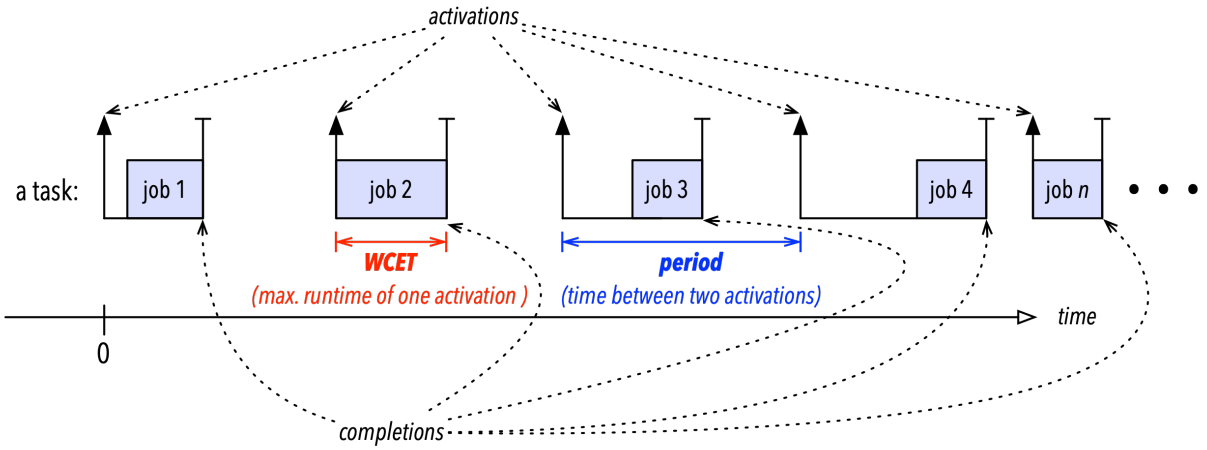

Figure 1: Illustration of a periodic task

The first question we need to answer is how to map the model entities "tasks" and "jobs" to actual system concepts and resources. Now, there are many plausible ways to do this, but the most simple and most common approach is to realize each task with a separate *thread* (or OS process) and to implement the corresponding sequence of jobs as *loop iterations*, which results in the following basic sketch:

```
int main(int argc, char** argv) {
    // do whatever one-time initialization, argument parsing, etc. is necessary
    process_activations();
}

// the "periodic task"
void process_activations(void) {
    // an infinite sequence of jobs <-> an infinite loop
    while (1) {
        // wait until "release of next job"
        // (i.e., wait until next activation)
        sleep_until_next_activation();
        // call the actual application logic
        process_one_activation();
    }
}

// wait until it's time for the next "job" of the task
void sleep_until_next_activation(void) {
    // TBD: still need to implement this, see next section
}

// one "job" of the task
void process_one_activation(void) {
    // application logic goes here
    printf("Hello real-time world!\n");
}
```

There are other ways in which "jobs" and "tasks" can be realized in an OS (e.g., each job could be a separate thread, there could be a thread pool shared among jobs, or jobs could be realized as signal handlers, etc.), but the above sketch is about as simple as it gets and works most of the time, so we'll focus on it here.

In its most basic form, the periodic task model describes the temporal behavior with two per-task parameters:

- The task's *period*, which is the separation in time between two consecutive activations of the task (i.e., the inverse of the frequency at which the corresponding activity should be carried out). In the above sketch, this corresponds to how much time must pass between consecutive calls to `process_one_activation()`).
- The task's *worst-case execution cost* (WCET), which is the (analytically derived, assumed, measured, or "guesstimated") maximum number of processor cycles required to complete one activation (or job) of the task. In the above examples, the WCET corresponds to how many cycles are needed to execute one iteration of the infinite loop in `process_activations()`, which is usually dominated by the cost of calling, executing, and returning from `process_one_activation()`.

The WCET is determined by the implementation of the application logic and the underlying processing platform (processor core, any caches, memory arbitration, etc.). To perform a meaningful *analysis*, the WCET needs to be somehow extracted from the implementation or estimated with reasonable confidence. This is actually not so trivial[1], but fortunately for us, when *implementing* a periodic task, we can largely ignore this parameter.

Most significant for us is the period parameter, as this needs to be enforced correctly to ensure a correct pattern of activations. In the above sketch, this ought to be done in the `sleep_until_next_activation()` function, which however we still need to implement.

Sleeping Until the Next Activation

The word "sleep" in the name of the function `sleep_until_next_activation()` gives it away that we will have to ask the OS to suspend the execution of the thread until the time of the next activation. It is hence very tempting to try to use the well-known `nanosleep(2)` system call, which sounds like it was made just for this purpose. However, that approach is fundamentally flawed — there's actually *no* way to use `nanosleep()` correctly (for the purpose of implementing periodic tasks). Since there's code in the wild that gets this wrong, let's briefly understand why using `nanosleep()` doesn't work reliably.

The root of the problem is that `nanosleep()` implements a *relative* sleep — the OS suspends the thread for a given number of nanoseconds relative to the time at which the OS gets around to processing the system call. Concretely, suppose that we are implementing a task that is supposed to be activated every 100ms (i.e., an activation frequency of 10Hz). Suppose the task was correctly activated at time 1000ms, ran until time 1021ms, and now needs to sleep until time 1100ms, the time of the next activation. To use `nanosleep()`, we'd have to calculate the sleep time as $\Delta = 1100\text{ms} - 1021\text{ms} = 79\text{ms}$, and then call `nanosleep()` with $\Delta = 79\text{ms}$ as the argument. However, this opens a race window: if the thread is preempted by the kernel *after* calculating Δ but *before* calling `nanosleep()` (as illustrated in Figure 2 below), then the sleep will overshoot the desired wake-up time, because it is relative to when `nanosleep()` is

executed in the kernel, and *not* relative to when Δ was calculated. To get back to the example, suppose our thread was preempted for 13ms just before it could call `nanosleep()`: as a result, it would call `nanosleep()` only at time 1034ms, and consequently be activated again only at time 1034ms $+\Delta = 1113$ms, which is too late.

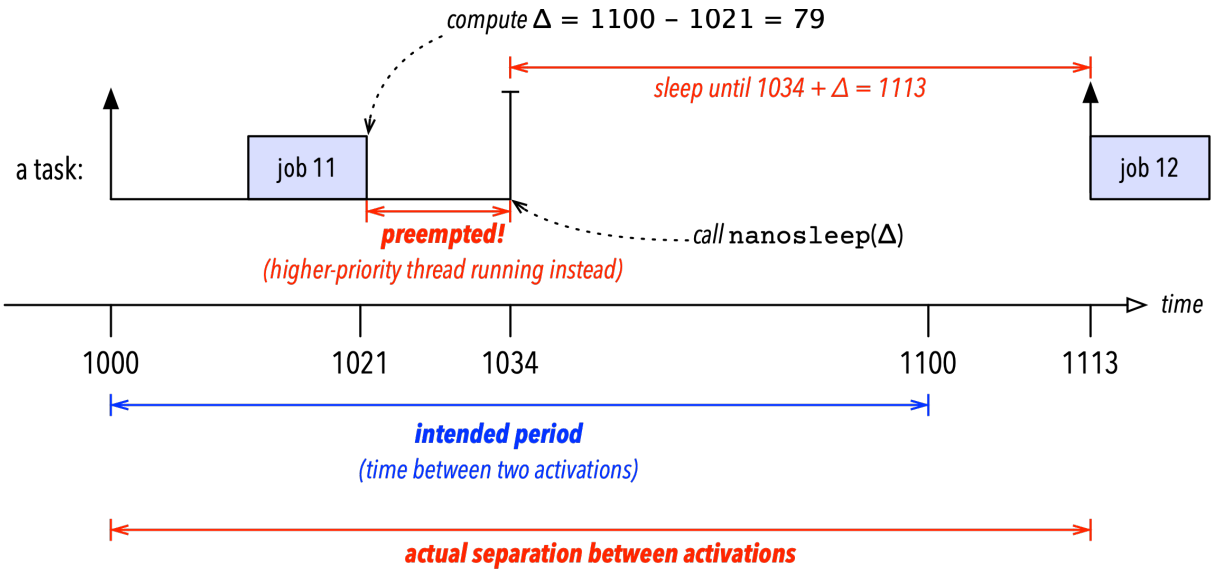

Figure 2: Illustration of a delayed wakeup due to a preempted relative sleep

This problem is fundamental to relative sleeps in general and a shortcoming in API design — there is no way to correctly use APIs like `nanosleep()` in systems require precision timing and allow threads to be preempted.

The correct alternative is to use timer APIs that support *absolute* wake-up times. Instead of communicating to the kernel for *how long* a thread should sleep (as in in the case of `nanosleep()`), which is meaningless without the context of when that request is made, an API to realize precision sleeps needs to communicate *until when* a thread should sleep, as the semantics of this is invariant w.r.t. when the system call is executed.

Linux offers the ability for a thread to sleep until an absolute time via the newer `clock_nanosleep()` system call. In fact, by default `clock_nanosleep()` also results in relative sleeps, but with the `TIMER_ABSTIME` flag, an absolutely timed sleep is possible (refer to the manual page for details).

Additionally, `clock_nanosleep()` also allows specifying a *clock* (or timeline) w.r.t. which the given time instant should be interpreted. There is another pitfall lurking here: based on the name alone, it is tempting to use CLOCK_REALTIME when implementing a real-time task. That however is exactly the wrong choice since the "REAL" here refers to calendar time, which is settable by the user, affected by daylight saving time, etc., and hence not suitable for driving real-time tasks. Rather, a better choice is `CLOCK_MONOTONIC`, which as the name suggests is monotonic and usually counts the time since boot.

Finally, another potential pitfall is that `clock_nanosleep()` may return early. For example, in case of a signal or for other obscure reasons, the kernel may choose to resume the thread earlier than the requested wake-up time. To allow the thread to react to such spurious wakeups, `clock_nanosleep()` returns a nonzero error code if it did not sleep until the desired time (refer to the manual page for details). To implement a proper precision sleep, a thread must hence check for premature activation and call `clock_nanosleep()` repeatedly until the desired point in time is reached.

A Periodic Task Template

Putting everything together, we arrive at the following skeleton for a proper periodic real-time task on Linux. Note that `clock_nanosleep()` expects to be given a point in time with nanosecond resolution using the type `struct timespec`, which consists of an integral number of seconds (`tv_sec`) plus a number of nanoseconds (`tv_nsec`) to express the sub-second part. A value of type `struct timespec` can express either a duration or a point in time (as a duration since some "time zero").

```
int main(int argc, char** argv) {
    // do whatever one-time initialization, argument parsing, etc. is necessary
    process_activations();
}

// the state that we need to keep track of to ensure
// periodic activations
struct periodic_task {
    // just for convenience, we keep track of the
    // sequence number of the current job
    unsigned long current_job_id;

    // desired separation of consecutive activations
    struct timespec period;

    // time at which the task became fist operational
    struct timespec first_activation;

    // time at which the current instance was (supposed
    // to be) activated
    struct timespec current_activation;

    // flag to let applications terminate themselves
    int terminated;
};

// for example, 100ms
#define PERIOD_IN_NANOS (100UL * 1000000UL)

// the "periodic task"
void process_activations(void) {
    int err;
    struct periodic_task tsk;

    // to match the real-time theory, the job count starts at "1"
    tsk.current_job_id = 1;

    // run until application logic tells us to shut down
    tsk.terminated = 0;

    // note the desired period
    tsk.period.tv_sec = 0;
    tsk.period.tv_nsec = PERIOD_IN_NANOS;

    // record time of first job
    err = clock_gettime(CLOCK_MONOTONIC, &tsk.first_activation);
    assert(err == 0);
    tsk.current_activation = tsk.first_activation;

    // execute a sequence of jobs until app shuts down (if ever)
    while (!tsk.terminated) {
```

```
            // wait until release of next job
            sleep_until_next_activation(&tsk);
            // call the actual application logic
            process_one_activation(&tsk);
            // advance the job count in preparation of the next job
            tsk.current_job_id++;
            // compute the next activation time
            timespec_add(&tsk.current_activation, &tsk.period);
        }
    }

    // wait until it's time for the next "job" of the task
    void sleep_until_next_activation(struct periodic_task *tsk) {
        int err;
        do {
          // perform an absolute sleep until tsk->current_activation
            err = clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &tsk->current_activation, NUL
    L);
            // if err is nonzero, we might have woken up too early
        } while (err != 0 && errno == EINTR);
        assert(err == 0);
    }

    // one "job" of the task
    void process_one_activation(struct periodic_task *tsk) {
        // application logic goes here
        printf("Hello real-time world! This is job #%lu.\n",
               tsk->current_job_id);
    }
```

[download complete example]

That's it! As long as the application logic in `process_one_activation()` does not call any blocking system calls (which we arguably are violating here by calling `printf()`), the above template results in an activation and execution pattern that perfectly matches Liu and Layland's classic periodic task model. Not difficult at all, isn't it?

Let me add two quick comments on likely extensions of the above template. First, the given periodic task skeleton is implicitly robust w.r.t. overruns. If for any reason the periodic task becomes backlogged (i.e., if the thread doesn't run for a while because the OS is overloaded with higher-priority threads), then it will *not* skip any jobs. Rather, `clock_nanosleep()` returns immediately without suspending the thread if asked to carry out an absolute sleep until a point in time already in the past. So if several activations of a task are delayed, it will simply executed the delayed jobs in a back-to-back manner until it has caught up. This is a reasonable default behavior (no job is ever skipped, which can make the application logic simpler), but if such queuing of late jobs is not desired (e.g., to more quickly recover from transient overload, it helps to "drop" tardy jobs), then it is easy to modify `process_activations()` and `sleep_until_next_activation()` to skip jobs with activation times already in the past.

Second, note how the time of the first activation is just taken as the current time when the process is launched, which is effectively nondeterministic. As the time of the next activation is always computed by advancing the time of the last activation by `tsk.period` time units, this implicitly fixes the timeline for all future activations. As a result, if a task *set* comprised of multiple tasks is launched, they are unlikely to have aligned timelines due to unpredictable differences in their initial activation times. This can be undesirable, especially if tasks communicate via shared memory. The alternative is to coordinate a shared `tsk.first_activation` value for all periodic tasks (rather than reading it from `clock_gettime()`), so that all

tasks release their first job at the exactly the same time. Such a shared start time, which is called a *synchronous release* in the real-time scheduling literature, can be easily realized either via shared memory, a named pipe, simply as a command-line argument or environmental variable set by a startup script (to name a few possibilities).

Conclusion

From following the above simple template for periodic tasks, we already get some nice properties. For instance, if all tasks look like this and are scheduled under Linux's `SCHED_FIFO` scheduler on a single core with rate-monotonic priorities, and if no task has a WCET exceeding its period, then as a rule of thumb all jobs complete within one period after their activation if the tasks in total cause no more than roughly 69% utilization (i.e., Liu and Layland's famous utilization bound for rate-monotonic scheduling of periodic tasks). Furthermore, given the rich literature on scheduling and the analysis of periodic tasks, much strong guarantees (and higher utilizations) can be accomplished.

To conclude, there's actually a rather straightforward connection between code that runs on Linux and the models assumed in large parts of the real-time systems literature. In any case, I hope you find this template to be useful — please let me know what you think! In particular, are there any other practical topics you'd like see covered, such as how to implement theory-compliant event-driven sporadic tasks, precedence constraints, or synchronization of shared resources? Please leave your suggestions in the comments or send me an email.

---

1. How to estimate WCETs properly would be a topic for another whole (series of) blog post(s) — let the editors know if you'd like to see more on this question. ↩