LINUTRONIX
LINUX FOR INDUSTRY

A Checklist for Writing Linux Real-Time Applications

Embedded Linux Conference Europe
26-29 October 2020

John Ogness <john.ogness@linutronix.de>

LINUTRONIX
LINUX FOR INDUSTRY

What is Real-Time?

- Correctness is not just about writing bug-free, efficient code.
- It also means **executing at the correct time**.
- And failing to meet timing restrictions leads to an error.
- This requires:
  - deterministic runtime/scheduling behavior
  - interruptibility
  - priority inversion avoidance

Priority Inversion

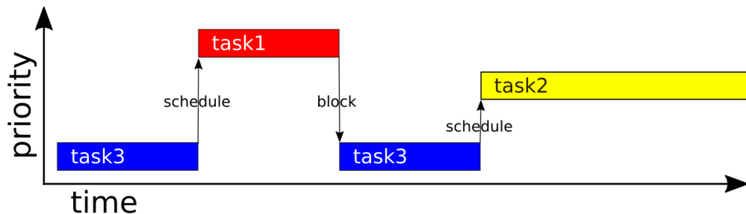


Figure: An example of priority inversion.

In this example, task3 is holding a lock that task1 wants. However, task3 never gets a chance to release that lock because it was interrupted by task2.

Priority Inheritance

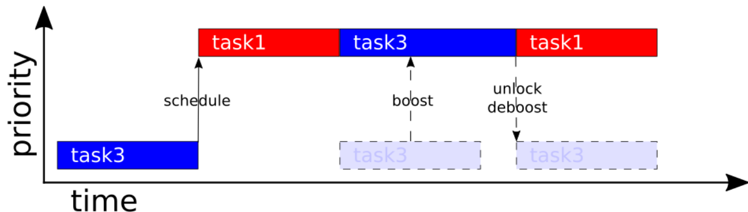


Figure: An example showing priority inheritance.

Linux supports **priority inheritance** to temporarily boost the priority of task3 once task1 tries to acquire the lock. This results in task1 acquiring the lock as soon as possible.

LINUTRONIX
LINUX FOR INDUSTRY

POSIX

Linux real-time features are implemented using the POSIX standard API. Most developers are already comfortable with this interface.

The real-time API for Linux is essentially contained in the `sched.h`, `time.h`, and `pthread.h` header files.

Scheduling Policies

Non-Real-Time Policies:

- **SCHED_OTHER**: dynamic time slices based on **nice** value
- **SCHED_BATCH**: a disfavored SCHED_OTHER
- **SCHED_IDLE**: run only when otherwise idle

Real-Time Policies:

- **SCHED_FIFO**: static priority (1-99), can only lose the CPU to higher priority tasks or hardware interrupts
- **SCHED_RR**: like SCHED_FIFO but with round robin scheduling for tasks of the same priority
- **SCHED_DEADLINE**: dynamic priority based on deadlines

Be aware of `/proc/sys/kernel/sched_rt_runtime_us`**, which can artificially limit real-time tasks. Set to** `-1` **to disable.**

Scheduling with Real-Time

When dealing with real-time systems, generally only SCHED_FIFO and SCHED_RR are of interest. However, these policies should only be used on the real-time tasks. All other tasks of the system will generally use SCHED_OTHER, with varying nice values and/or cgroups to control their CPU time.

The scheduling policy can be set using the chrt command:

```
Set policy:
  chrt [opts] <policy> <prio> <pid>
  chrt [opts] <policy> <prio> <cmd> [<arg> ...]

Scheduling policies:
-f, --fifo        set policy to SCHED_FIFO
-o, --other       set policy to SCHED_OTHER
-r, --rr          set policy to SCHED_RR (default)
```

... or in code:

```
#include <sched.h>

struct sched_param param;

param.sched_priority = 80;
sched_setscheduler(0, SCHED_FIFO, &param);
```

CPU Affinity

- Each task has its own CPU affinity mask, specifying which CPUs it may be scheduled on.
- Boot parameters are available to set default masks for all tasks (including the kernel's own tasks).
- A CPU affinity mask for routing individual hardware interrupt handling is also available.

Isolating activity on CPUs can increase determinism by controlling how tasks are interrupted. Just be aware that SMP systems usually have multiple CPUs sharing caches.

The CPU affinity mask for tasks can be set using the **taskset** command:

```
taskset [options] mask command [arg]...
taskset [options] -p [mask] pid
```

... or in code:

```
#define _GNU_SOURCE
#include <sched.h>

cpu_set_t set;

CPU_ZERO(&set);
CPU_SET(0, &set);
CPU_SET(1, &set);
sched_setaffinity(pid, CPU_SETSIZE, &set);
```

Kernel parameters related to CPU isolation:

- maxcpus=*n*: limits the kernel to bring up *n* CPUs
- isolcpus=*cpulist*: specify CPUs to isolate from disturbances

The default CPU affinity for routing hardware interrupts when new interrupt handlers are registered can be viewed/set in:

```
/proc/irq/default_smp_affinity
```

The CPU affinity for routing hardware interrupts for already registered interrupt handlers can be viewed/set in:

/proc/irq/***irq-number***/effective_affinity
/proc/irq/***irq-number***/smp_affinity

LINUTRONIX
LINUX FOR INDUSTRY

Page Faulting

By default, physical memory pages are mapped to the virtual address space **on demand**. This allows features such as over-commitment and it affects **all** virtual memory of a process:

- text segment
- initialized data segment
- uninitialized data segment
- stack(s)
- heap

Avoid Page Faults

- **Tune glibc's malloc** to avoid memory mapping as a form of memory allocation. mmap'd memory cannot be reused after being freed.
- **Lock down allocated pages** so that they cannot be returned to the kernel. Hold on to what you've been given.
- **Prefault** the heap and the stack(s).

The **malloc** function of glibc can be tuned using the **mallopt** function. Below are the calls to disable memory mapping for memory allocation and disable heap trimming.

```
#include <malloc.h>

mallopt(M_MMAP_MAX, 0);
mallopt(M_TRIM_THRESHOLD, -1);
```

Memory pages can be locked down with the **mlockall** function:

```
#include <sys/mman.h>

mlockall(MCL_CURRENT | MCL_FUTURE);
```

An example of heap prefaulting:

```
#include <stdlib.h>
#include <unistd.h>

void prefault_heap(int size)
{
    char *dummy;
    int i;

    dummy = malloc(size);
    if (!dummy)
        return;

    for (i = 0; i < size; i += sysconf(_SC_PAGESIZE))
        dummy[i] = 1;

    free(dummy);
}
```

LINUTRONIX
LINUX FOR INDUSTRY

An example of stack prefaulting:

```
#include <unistd.h>

#define MAX_SAFE_STACK (512 * 1024)

void prefault_stack(void)
{
    unsigned char dummy[MAX_SAFE_STACK];
    int i;

    for (i = 0; i < MAX_SAFE_STACK; i += sysconf(_SC_PAGESIZE))
        dummy[i] = 1;
}
```

LINUTRONIX
LINUX FOR INDUSTRY

Synchronization

- **Use pthread_mutex for locking**. These objects have owners (unlike semaphores) so the kernel can more intelligently choose which processes to schedule.
- **Activate priority inheritance**. Unfortunately this is not the default.
- Activate shared and robustness features **if** the lock is accessed by multiple processes in shared memory.

LINUTRONIX
LINUX FOR INDUSTRY

Here is a trivial example showing the syntax and semantics of lock initialization and usage.

```
#include <pthread.h>

pthread_mutex_t lock;
pthread_mutexattr_t mattr;

pthread_mutexattr_init(&mattr);
pthread_mutexattr_setprotocol(&mattr, PTHREAD_PRIO_INHERIT);
pthread_mutex_init(&lock, &mattr);

pthread_mutex_lock(&lock);
/* do critical work */
pthread_mutex_unlock(&lock);

pthread_mutex_destroy(&lock);
```

The feature **PTHREAD_PROCESS_SHARED** should only be set if the lock resides in shared memory. Locks marked with this feature have additional overhead when contended.

Be aware that activating **PTHREAD_MUTEX_ROBUST** introduces new semantics for `phread_mutex_lock()`. Make sure you fully understand its usage, otherwise you may have broken synchronization. See the man page of `pthread_mutexattr_setrobust(3)` for details.

Conditional Variables

- **Use pthread_cond objects** for notifying tasks if shared resources are involved. These can be associated with pthread_mutex objects to provide synchronized notification.
- **Do not use signals** (such as POSIX timers or the kill() function). They involve unclear and limited contexts, do not provide any synchronization, and are difficult to program correctly.
- Activate the shared feature **if** the conditional variable is accessed by multiple processes in shared memory.
- The sender should notify the receiver **before** releasing the lock associated with the conditional variable.

Here is an example showing the syntax and semantics of conditional variable initialization.

```
#include <pthread.h>

pthread_condattr_t cattr;
pthread_cond_t cond;

pthread_condattr_init(&cattr);
/* pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED); */
pthread_cond_init(&cond, &cattr);
```

Like with pthread_mutex, the feature PTHREAD_PROCESS_SHARED should only be set if the conditional variable resides in shared memory.

Signalling (code snippet)

```
#include <pthread.h>

pthread_mutex_t lock;
pthread_cond_t cond;
```

Code of receiver:

```
pthread_mutex_lock(&lock);
pthread_cond_wait(&cond, &lock);
/* we have been signaled */
pthread_mutex_unlock(&lock);
```

Code of sender:

```
pthread_mutex_lock(&lock);
/* do the work */
pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&lock);
```

The Monotonic Clock

- Use the POSIX functions that allow clock specification. These begin with **clock_**.
- **Choose CLOCK_MONOTONIC**. This is a clock that cannot be set and represents monotonic time since some unspecified starting point.
- **Do not use CLOCK_REALTIME**. This is a clock that represents the ”real” time. For example, Monday 26 October 2020 13:00:00. This clock can be set by NTP, the user, etc.
- **Use absolute time values**. Calculating relative times is error prone because the execution itself takes time.

When using CLOCK_MONOTONIC and absolute times, a cyclical task becomes trivial to implement.

```
#include <time.h>

#define CYCLE_TIME_NS (100 * 1000 * 1000)
#define NSEC_PER_SEC (1000 * 1000 * 1000)

static void norm_ts(struct timespec *tv)
{
    while (tv->tv_nsec >= NSEC_PER_SEC) {
        tv->tv_sec++;
        tv->tv_nsec -= NSEC_PER_SEC;
    }
}

void cyclic_task_main(void)
{
    struct timespec tv;

    clock_gettime(CLOCK_MONOTONIC, &tv);

    while (1) {
        /* do the work */

        /* wait for next cycle */
        tv.tv_nsec += CYCLE_TIME_NS;
        norm_ts(&tv);
        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &tv, NULL);
    };
}
```

LINUTRONIX
LINUX FOR INDUSTRY

How to Evaluate a Real-Time System?

- Use the **cyclictest** tool. (Part of the **rt-tests** package.)
  - measures/tracks latencies from hardware interrupt to userspace
  - run at the priority level to evaluate
  - **example:** `cyclictest -S -m -p <prio> --secaligned`
- Generate worst case system loads.
  - scheduling load: the **hackbench** tool
  - interrupt load: flood pinging with "`ping -f`"
  - serial/network load: "`top -d 0`" via console and network shells
  - memory loads: OOM killer invocations
  - various load scenarios: the **stress-ng** tool
  - stress controller/peripheral devices
  - idle system

perf

- **perf** is a tool that can count various types of hardware and software events.
- Some examples: CPU cycles, page faults, cache misses, context switches, scheduling events, ...
- It can help to identify performance issues with real-time tasks by showing if certain types of latency causing events are occurring.

```
usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]
```

Versions of **perf** may depend on certain kernel versions/features. For this reason it is important that the correct **perf** version is used. **perf** is part of the Linux kernel source, in `tools/perf`. It is best to use the version that comes from the same source as the running kernel.

perf has many features and the help is spread across many man pages. However, **perf** can automatically open the correct man pages:

```
perf --help
perf --help <command>
```

LINUTRONIX
LINUX FOR INDUSTRY

List all supported events:

```
perf list
```

Count the number of CPU cycles and page faults on a system over 5 seconds:

```
perf stat -a -e cpu-cycles -e page-faults sleep 5
```

Count the combined number of context switches only for the tasks 123 and 124 over 5 seconds:

```
perf stat -e cpu-cycles -e context-switches \
     -p 123,124 sleep 5
```

View the top symbols causing cache misses on the first CPU (with perf pinned to the second CPU):

```
taskset 2 perf top -C 0 -e cache-misses
```

LINUTRONIX
LINUX FOR INDUSTRY

The Linux Tracing Infrastructure

- Log not only **what** happened but also **when** it happened.
- Provides a rich set of **software events** (points of code) in the kernel.
- Custom kernel events can be added to a live system.
- Custom userspace events can be added to a live system. (Userspace tasks must be started after the events are added, but the programs do not need to be modified.)
- Tools are available to simplify usage, such as **trace-cmd** and **perf**.
- Graphical tools are available to view and analyze trace data, such as **kernelshark** and **Trace Compass**.

The Linux tracing infrastructure has many more features than presented here. With it, the possibilities for debugging and analyzing real-time software are nearly limitless. If you are serious about debugging real-time issues, it is worth it to look deeper into these Linux features.

Keep in mind that this is not just for debugging. With the tracing infrastructure developers and testers can **verify** real-time performance.

Simple Examples:

List all supported events:

```
trace-cmd list
```

Record scheduling wakeup and switch events system-wide for 5 seconds:

```
trace-cmd record -e sched:sched_wakeup -e sched:sched_switch sleep 5
```

View the recorded events (recorded in `trace.dat`):

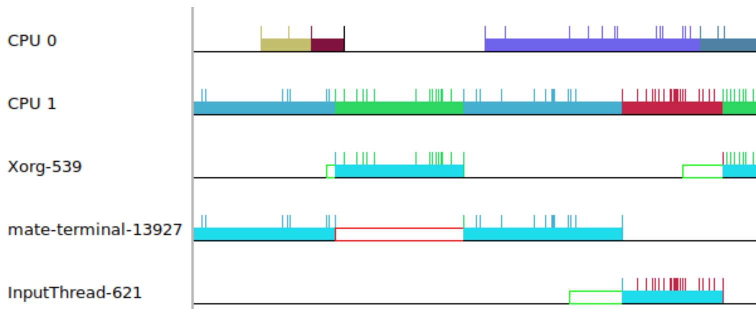
```
trace-cmd report
```

Graphically view the recorded events:

```
kernelshark
```

Tracing (kernelshark output)

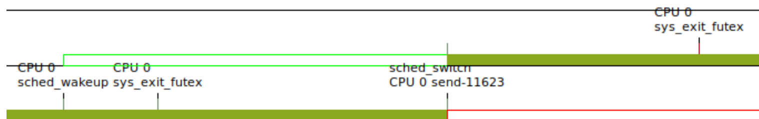
LINUTRONIX LINUX FOR INDUSTRY



In this figure we see that mate-terminal had work to do (is in the RUNNABLE state) but Xorg has the CPU. And CPU0 is idle!

Is this a problem? Why do we have this situation? From the trace we do not know the answers to these questions. But at least we know it is happening!

Tracing (kernelshark output)



From a detailed (zoomed in) view, we can measure the latences between when the task is set `RUNNABLE`, when it started running on the CPU, and when it returned to userspace.

In most cases there is no need for expensive hardware solutions. The kernel comes with all these features built-in!

Optimal Real-time Configuration

Certain kernel options can make a big difference in real-time performance:

- preemption mode
- watchdog/debug features
- tickless, dynamic, or periodic ticks
- CPU idle features
- frequency scaling features

Kernel Configuration

- `CONFIG_PREEMPT_*`
  To activate all real-time capabilities, set to `CONFIG_PREEMPT_RT_FULL` (currently only available with the PREEMPT_RT patch, but will be mainline soon).
- `CONFIG_LOCKUP_DETECTOR` `CONFIG_DETECT_HUNG_TASK`
  These tasks run with priority 99. Unless explicitly used/needed, disable them.
- `CONFIG_NO_HZ` `CONFIG_HZ_*`
  Eliminating or delaying the kernel tick can reduce power consumption, but will result in larger latencies.
- `CONFIG_CPU_FREQ_GOV_*` `CONFIG_CPU_FREQ_DEFAULT_*`
  Make sure CPU frequency is enabled, but use the performance governor for minimal latency.
- `CONFIG_DEBUG_*`
  Some debugging features are very expensive. Make sure only those are enabled that are explicitly wanted.
- `CONFIG_FTRACE` `CONFIG_KPROBES` `CONFIG_UPROBES`
  Not performance related, but important so that tracing is possible.

Real-Time Checklist

Real-Time Priority
- SCHED_FIFO, SCHED_RR

CPU Affinity
- applications
- interrupt handlers
- interrupt routing

Memory Management
- avoid mmap() with malloc()
- lock memory
- prefault memory

Time and Sleeping
- use monotonic clock
- use absolute time

Kernel Configuration
- be aware of options affecting latency

Avoid Signals
- such as POSIX timers
- such as kill()

Avoid Priority Inversion
- use pthread_mutex (and set attributes!)
- use pthread_cond (and set attributes!)

Verify Results
- trace scheduling
- trace page faults
- monitor traces

NMIs
- know what NMIs exist and how they can be triggered/avoided

Questions?

https://wiki.linuxfoundation.org/realtime/