e-ale-rt-apps

Building Real-Time Applications for Linux

Version 20181023

e-ale

The E-ALE (Embedded Apprentice Linux Engineer) is a series of seminars held at existing conferences covering topics which are fundamental to a Linux professional in the field of Embedded Linux.

This seminar will spend equal time on lecture and hands on labs at the end of each seminar which allow you to practice the material you've learned.

> This material makes the assumption that you have minimal experience with using Linux in general, and a basic understanding of general industry terms. The assumption is also made that you have access to your own computers upon which to practice this material.

More information can be found at **https://e-ale.org**/

Contents

Chapter 1

Preliminaries

e-ale

1.1 Speaker Information

Introduction

- **John Ogness** <john.ogness@linutronix.de>
- Converted to the ”UNIX way” in 1998.
- Using Linux professionally since 2001.
- Working on board support packages (BSPs), real-time systems, and as a trainer at Linutronix GmbH (Germany) since 2008.

John Ogness studied Computer Science at Utah State University (USA) and has been professionally involved with Linux since 2001. He has been working for the company Linutronix GmbH since 2008. There he specializes in Linux-based board support packages, real-time applications, and training. He is also maintainer of the Minicoredumper project. In the past, he developed software for security applications and autonomous robots.

John lives at the Lake of Constance (Bodensee) in Germany with his wife and two sons.

1.2 Real-Time Defined

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

The Linux **PREEMPT_RT** patchset helps to dramatically improve the real-time features of Linux. This patchset is currently being pushed mainline, piece by piece. Many of the pieces are already mainline. For example:

- generic interrupt subsystem
- generic timekeeping
- generic timer handling
- high resolution timers
- consolidation of the locking infrastructure
- tracing infrastructure
- threaded interrupt handlers

Various boot options and strategies will be mentioned later to help improve the real-time features of Linux (with or without the **PREEMPT_RT** patchset).

Visit the Linux Foundation Real-Time Wiki at: https://wiki.linuxfoundation.org/realtime

Priority Inversion

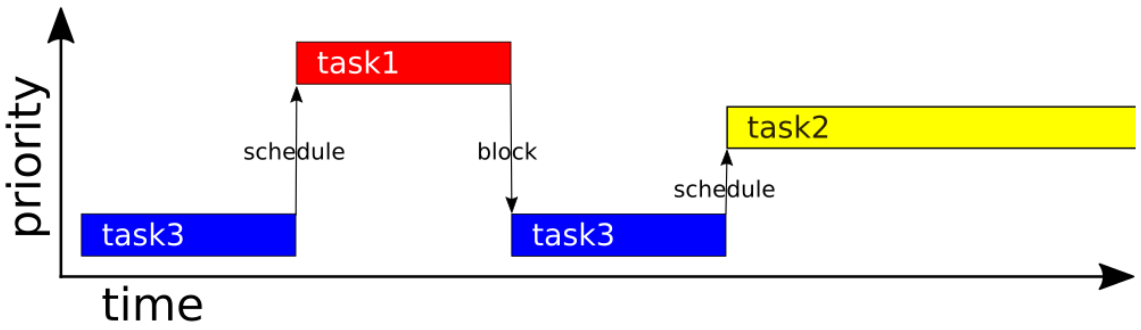


Figure 1.1: **An example of priority inversion.**

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.

The events leading to priority inversion:

- task3 acquires a lock
- task1 later tries to acquire that lock
- task1 blocks and task3 is scheduled so it can release the lock
- task2 interrupts task3 and runs unbounded
- task1 is now (indirectly) waiting on task2... priority inversion

Priority Inheritance

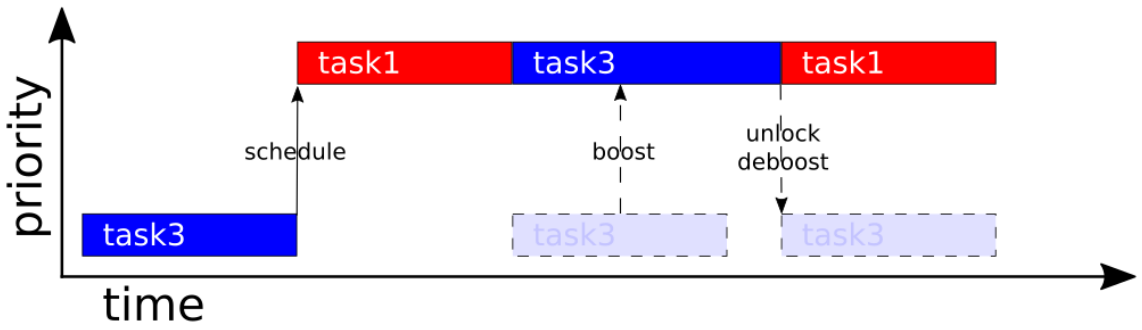


Figure 1.2: **An example showing priority inheritance.**

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.

There are two common methods for avoiding priority inversion:

- **priority inheritance**: When a task of higher priority wants a resource held by a task of lesser priority, the priority of the resource holder is ”boosted” to that of the higher priority task until that resource is made available.
- **priority ceiling**: Instead of assigning priorities to tasks, priorities are assigned to resources. When any task acquires a resource, it is assigned the priority of that resource until it is released.

1.3 Scheduling with Real-Time

Scheduling Policies

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

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

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 to influence their time slices.

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

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);
```

1.4 Limiting/Isolating CPUs

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.

By isolating real-time tasks on CPUs, the real-time performance can be further increased. Less interruptions usually translates to less latency. Just be aware that SMP systems usually have multiple CPUs sharing caches. Unfortunately this means that non-real-time tasks can influence a real-time task, even if that task is completely isolated on its own CPU:

The CPU affinity mask 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*`/smp_affinity`.

1.5 Understanding Memory Management

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

As an example, when a task allocates memory, the kernel will tell the task that the memory has been allocated and return a virtual address pointer to such memory. However, no actual physical memory has been reserved. It is not until the task starts reading/writing to the memory, that the kernel will actually find physical memory to map to it.

When virtual memory is accessed that has no physical memory mapping, this generates a page fault in the memory management unit (MMU). The kernel's exception handler will catch this and scramble to find a free page of physical memory to map. If no physical memory is available, and there are no mapped pages that can be paged out, the kernel will invoke the out of memory (OOM) killer.

In summary, malloc's are cheap and first accesses are expensive.

1.6 Evaluating Real-Time Systems

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
- 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

Loads should be generated for hardware devices that will be in use on the device. If it has USB, generate USB loads. If it has a display, generate graphic loads. Etc...

It is important that **cyclictest** is run at the correct priority for the tests. Otherwise the results may be worse than they should be. If testing the overall real-time performance for the system, it is common to start **cyclictest** with:

```
cyclictest -S -m -p 98 --secaligned
```

Note that real-time priority 99 should never be used by userspace software. This highest priority is used by critical kernel tasks that should never be interrupted by userspace software.

If you are unhappy with the latency results (and your are sure that you performed the test correctly), be aware that there are many kernel options available to tune the real-time performance of Linux.

1.7 Labs

Exercise 1.1: Measure Real-Time Latencies

Use **cyclictest** and various load generation tools and methods to try to find a worst-case latency for your system (your laptop and/or your embedded board).

A good command line for cyclictest:

```
cyclictest -S -m -p 98 --secaligned
```

Some ideas for load generation:

```
while true; do hackbench; done
```

```
ping -f 192.168.7.2
```

```
top -d 0
```

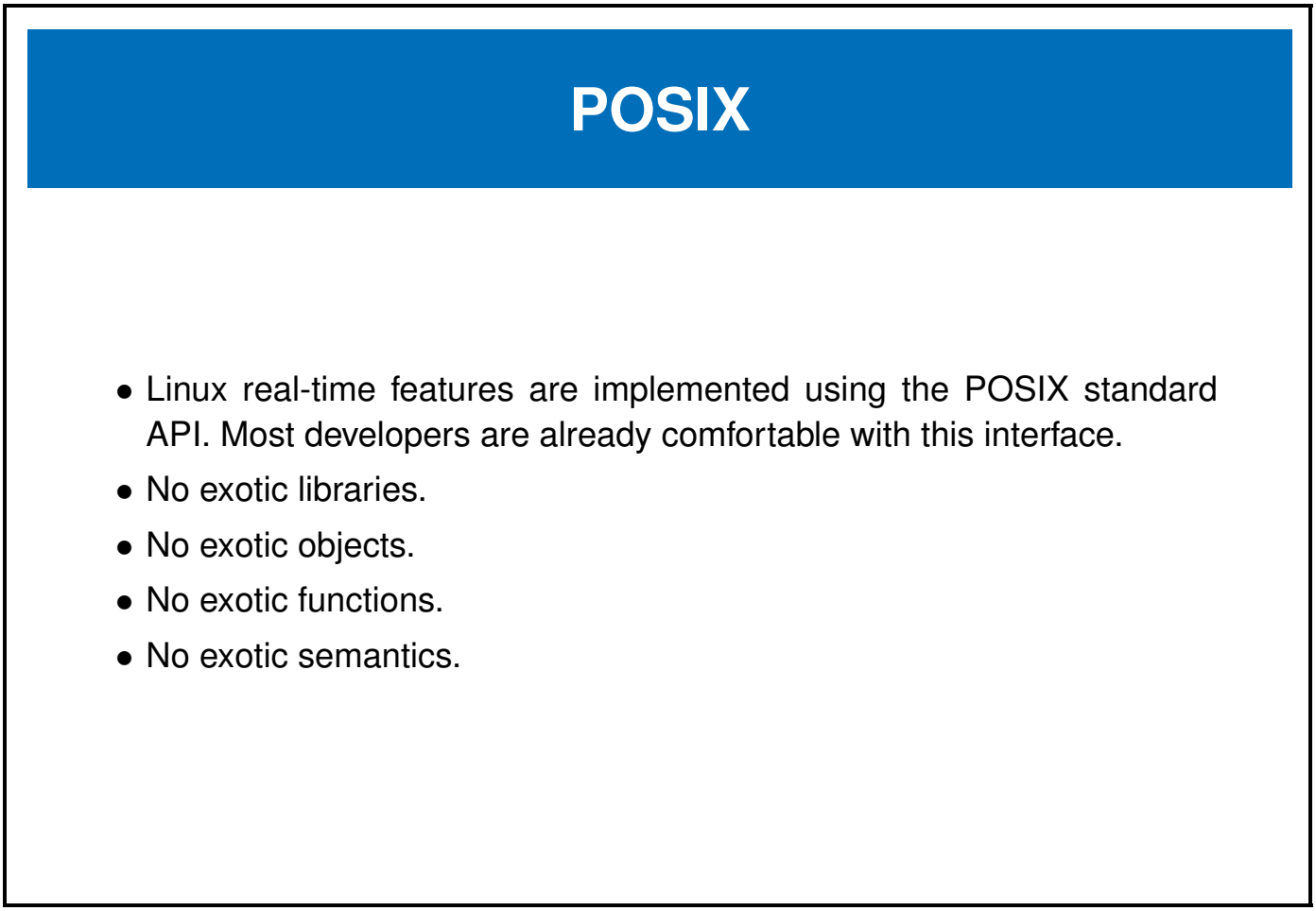
```
while true; do echo -n; done
```


Chapter 2

Application Development

e-ale

2.1 Real-Time API



Because the POSIX API is used, application developers can usually learn to write real-time applications very quickly. Also, since most Linux software is already written for POSIX, there is often little effort in combining existing code bases with real-time development.

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

2.2 Controlling Memory

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 as a form of memory allocation. This is important because glibc cannot reuse freed memory if it has been allocated using **mmap**. Such memory is managed directly by the kernel.

```
#include <malloc.h>

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

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] = i;

    free(dummy);
}
```

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] = i;
}
```

2.3 Using Clocks

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, Tuesday 23 October 2018 17:00:00. This clock can be set by NTP, the user, etc.
- **Use absolute time values**. Calculating relative times is error prone because the calculation 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 cycle_task(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);
    };
}
```

2.4 Locking

Synchronization

- **Use the pthread_mutex as the lock**. 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.

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 because they cannot use the fast userspace mutex (futex) implementation.

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.

2.5 Signalling

Conditional Variables

- **Use pthread_cond objects** for notifying tasks. 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_cond_init(&cond, &cattr);
```

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);
```

There have been many discussions on the usefulness of sending the signal while holding the lock. Indeed, with the glibc and kernel implementations for SMP systems, this procedure does not provide any real advantages.

However, using the POSIX API in this way allows for some (future) implementation where it can make an important difference.

The main concern is what happens when the sender releases the lock. If a high priority task has been waiting, it is only fair that it gets the lock before a lower priority task. By signalling waiters before the lock has been released, the operating system is given the opportunity to prepare for who will get the lock **before** the lock has been released. This can provide the fairness necessary to avoid priority inversion.

2.6 Labs

Exercise 2.1: Examine Page Fault Effects

The **pgflt** program demonstrates the effects of page faulting by performing several actions and tracking the time duration and number of page faults generated by those actions.

The **pgflt** program has implemented the various memory controlling techniques presented here. Each of these techniques can be individually enabled and disabled to explore their effects.

The test actions performed by **pgflt** are:

- allocate, set, and free 10MiB of memory
- call a function recursively to occupy a 7MiB stack

Each of the test actions are performed 4 times.

```
usage: ./pgflt [opts-bitmask]
  opts-bits:
  0x01 = mallopt
  0x02 = mlockall
  0x04 = prefault-stack
  0x08 = prefault-heap
  0x10 = run tests

  0x10 = no rt tweaks + tests
  0x1f = full rt tweaks + tests
```

Examples:

```
./pgflt 0x10
./pgflt 0x1f
```

Exercise 2.2: Run/Investigate LED Master Program

The **ledmaster** program runs a real-time cyclic task that is updating a (different) LED every 50ms. Run it and observe its performance. Things to consider:

- the real-time priority (or non-real-time nice value) of the task
- the load on the system
- the performance of **cyclictest** at different real-time priorities

Priority tools: `chrt`, `nice`, `renice`

Starting **ledmaster** with real-time priority:

```
chrt -f 80 ./ledmaster
```

Modifying the real-time priority of a running **ledmaster**:

```
chrt -f -p 80 $(pidof ledmaster)
```

Starting **ledmaster** with a non-real-time **nice** value:

```
nice -n 19 ./ledmaster
```


Exercise 2.3: Run/Investigate LED Mirror Program

The **ledmaster** does not just set an LED with each cycle, but also stores the LED number and value of the most recently set LED into shared memory. In shared memory there is also a mutex and conditional variable, that is used to synchronize the data and signal any ”listeners”. After setting a value, **ledmaster** signals.

The **ledmirror** program also sets an LED. However, rather than running cyclically, all it has available is the shared memory provided by the **ledmaster**.

Run it and observe its performance. Things to consider:

- the real-time priority (or non-real-time nice value) of the task
- the load on the system
- its effect on the **ledmaster** task

Priority tools: `chrt`, `nice`, `renice`

Starting **ledmirror** with real-time priority:

```
chrt -f 70 ./ledmirror
```

Modifying the real-time priority of a running **ledmirror**:

```
chrt -f -p 70 $(pidof ledmirror)
```

Starting **ledmirror** with a non-real-time **nice** value:

```
nice -n 19 ./ledmirror
```


Exercise 2.4: Run the LED Priority Script

The **ledprio** script makes use of the thumbwheel driver implemented in the ”IIO and Input Drivers” talk. It monitors for input events from the thumbwheel. When these events occur, the thumbwheel value is read and the priority of the **ledmirror** program is adjusted. The range is from `SCHED_OTHER/nice=19` until `SCHED_OTHER/nice=-20` and then finally `SCHED_FIFO/rtprio=1`.

Run all the components and play with the thumbwheel:

```
chrt -f 80 ./ledmaster &
./ledmirror &
./ledprio
```

Chapter 3

Debugging and Verification

e-ale

3.1 Performance Counters and Events

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.

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>
```

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 number of context switches for the LED tools over 5 seconds:

```
perf stat -a -e cpu-cycles -e context-switches -p $(pidof ledmaster),$(pidof ledmirror) 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
```

3.2 Tracing

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 all the scheduling wakeup an switch events on the system over 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
```


kernelshark Output

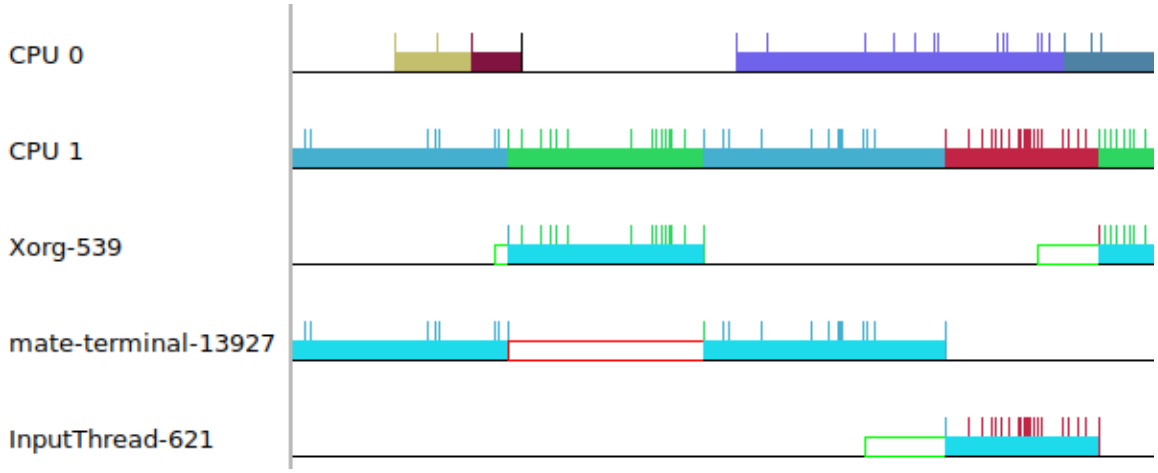


Figure 3.1: **kernelshark: Scheduling overview.**

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!

kernelshark Output

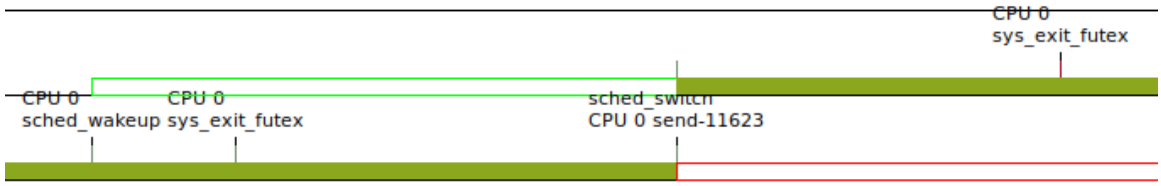


Figure 3.2: **kernelshark: Scheduling details.**

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!

3.3 Labs

Exercise 3.1: Measure Wake Latencies

Install **rt-tests**, **trace-cmd**, and **kernelshark** packages on your laptop.

Record a trace with **cyclictest** running for 2 seconds.

```
sudo trace-cmd record -e irq_vectors:local_timer_entry \
                      -e irq_vectors:local_timer_exit \
                      -e sched:sched_wakeup \
                      -e sched:sched_switch \
                      -e syscalls:sys_exit_clock_nanosleep \
                      cyclictest -S -m -p 98 --secaligned -D 2 -q
```

View the results and measure the components of the wakeup latency.

```
kernelshark
```

Chapter 4

Summary

e-ale

4.1 Checklist

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

Avoid Signals

- such as POSIX timers
- such as kill()

Avoid Priority Inversion

- use pthread_mutex (and set attributes!)
- use pthread_cond (and set attributes!)

Be aware of NMIs

Verify Results

- trace scheduling
- trace page faults
- monitor traces

Although this is not an exhaustive checklist, it should provide a broad set of things to consider. And remember, the presented tools and methods are only as good as the application code. In fact, these tools and methods will require your code to improve because now the real-time performance can be easily measured, analyzed, and verified.