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

1.1 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

Priority Inversion

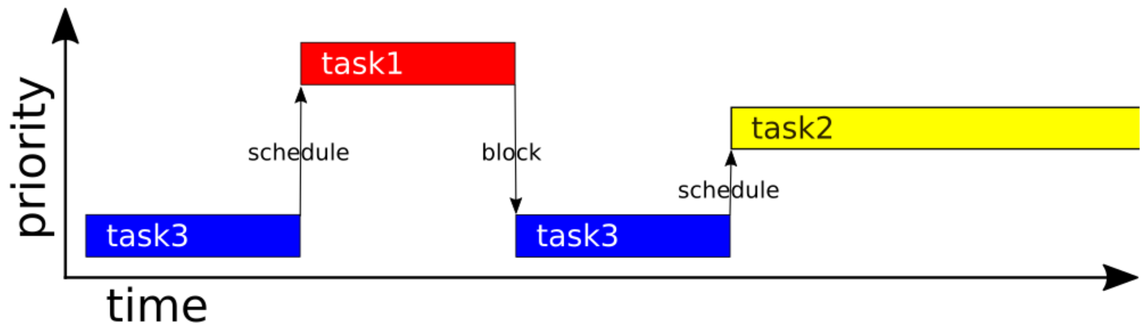


Figure 1.1: **An example of priority inversion.**

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.

Priority Inheritance

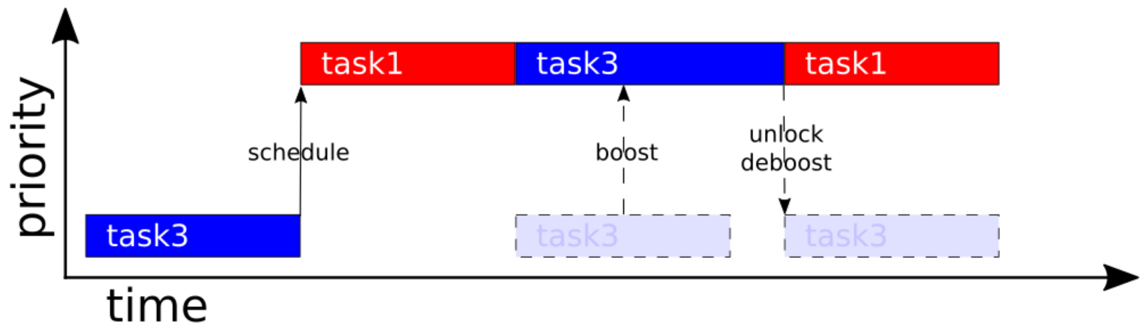


Figure 1.2: **An example showing priority inheritance.**

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.

1.2 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

1.3 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.

1.4 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

1.5 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

1.6 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

2.1 Real-Time API

POSIX

- Linux real-time features are implemented using the POSIX standard API. Most developers are already comfortable with this interface.
- No exotic libraries.
- No exotic objects.
- No exotic functions.
- No exotic semantics.

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).

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.

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.

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.

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);
```

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.

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**.

kernelshark Output

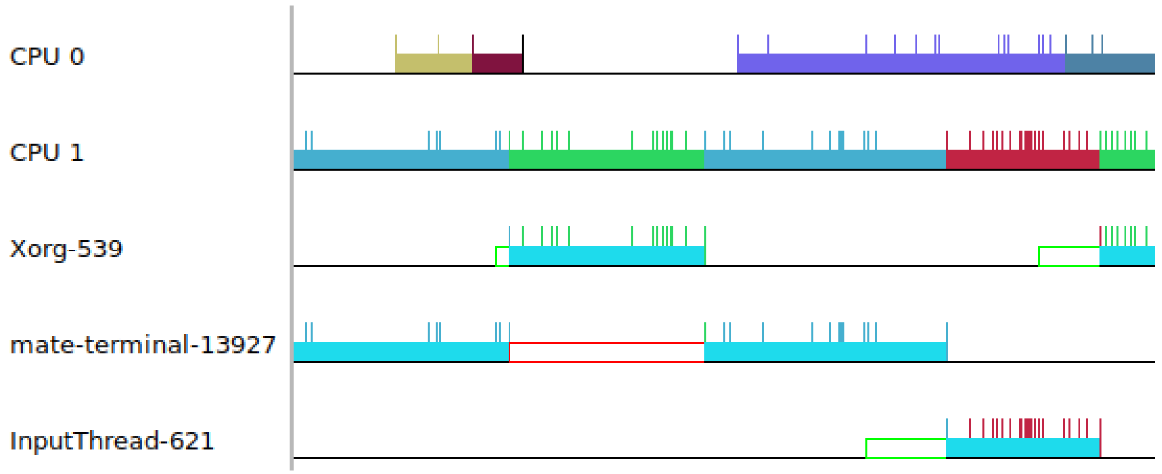


Figure 3.1: **kernelshark: Scheduling overview.**

kernelshark Output

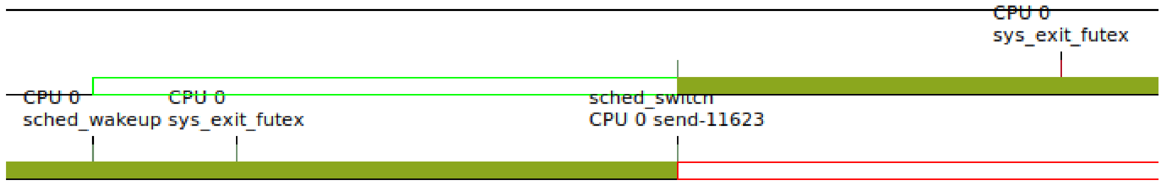


Figure 3.2: **kernelshark: Scheduling details.**

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.

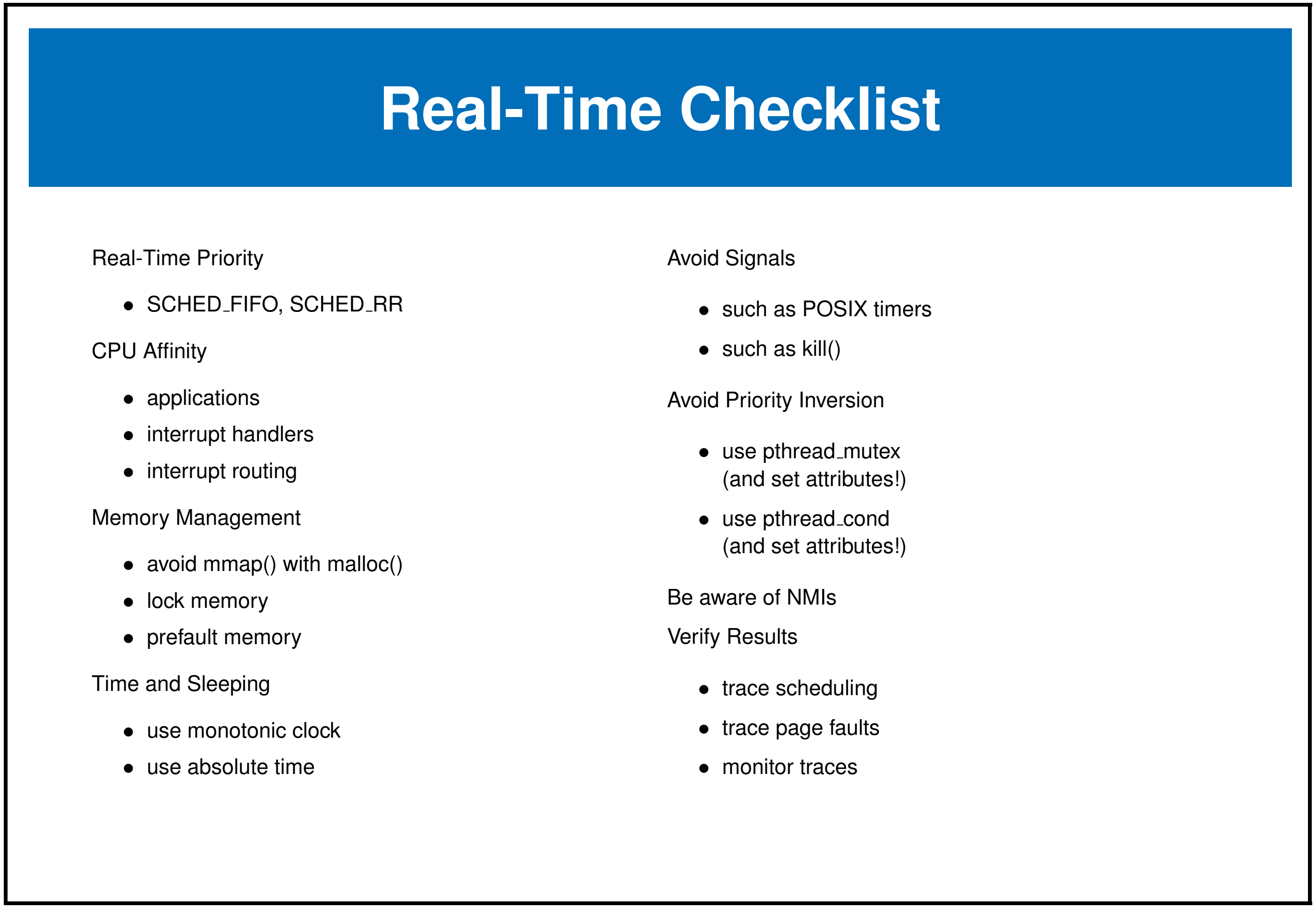
```
kernelshark
```

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