Politecnico di Milano

Soft Real-Time in the Linux Kernel

Improving predictability

Lecturer:
Ing. Roberto Farina
farina@cefriel.it

Overview

- Real-Time in a standard Linux System
- User space capability
- Kernel space capability

Linux and Real-Time

- Linux is not a Real-Time Operating System
- RTOS need
  - Fast interrupts management
  - Fast context switching
  - Granted task schedulability
- Linux could grant
  - Good predictability
- Linux could not grant
  - Task schedulability

Improve predictability

- Users can improve Linux predictability in two ways
  - Change some user space issues
    - Change schedule policies
    - Act on paging algorithm
    - Manage time within processes
  - Change some kernel space issues
    - Direct management of hardware interrupts
      - Low latency
    - Define start time for some tasks
    - Increase system time resolution
    - …

User space – schedule policy

- Linux manage processes in a time-sharing fashion
  - Scheduler dispatches ready process
  - Parallel execution illusion
  - Dynamic process priority
    - Execution time based
  - It’s impossible predict next process
- Two different scheduling algorithms to improve predictability
  - FIFO scheduler
  - FIFO-RR scheduler

User space – FIFO scheduler

- FIFO scheduler
  - Each task has own static priority
  - Each priority level has its own queue
  - If a higher priority task is ready, lower priority task is suspended and CPU is assigned to the higher one
  - The suspended task is moved to the head of its priority queue
  - When a process releases the CPU, it is inserted at the end of the queue
- It’s possible to know the exact execution order!

User space – FIFO-RR scheduler

- FIFO-RR scheduler
  - Soft version of FIFO
  - Each priority level has its own queue
  - Round-Robin algorithm within each priority level
  - It’s defined a *quantum* after that the running process releases the CPU, it is assigned to another process with the same priority
- This scheduler realize a sort of time-sharing within the same priority level
- Fight starvation problem

User space – change scheduler

- Syscall to change scheduler

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

- Policy indicates wanted scheduling algorithm
  - `SCHED_FIFO`
    - FIFO scheduling with fixed priority
  - `SCHED_RR`
    - FIFO-RR scheduling
  - `SCHED_OTHER`
    - Time-sharing scheduling (normal)

User space – change scheduler

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

- Indicates static process priority

User space – lock memory page

- Processes may need a swapped page
  - Introduce big latency
  - Not predictable
- User can lock
  - Current pages
  - Current and future pages

User space – lock memory page

- Linux provide some useful system calls

  `int mlock(const void *addr, size_t len)`

  `int mlockall(int flags)`
- You can lock a memory zone with `mlock()`
- You can lock all process memory with `mlockall()`
  - `MCL_FUTURE`
    - Lock current pages and future ones
  - `MCL_CURRENT`
    - Lock only current pages; future pages are not locked

User space – example 0

- The program says if a number is prime
  - Long computation
- The program tries to change scheduler
  - If owner is a simply user, the program run in time-slice environment
  - If owner is root, the program freezes the machine until its end
- How does execution time change?
- Why?
- What’s the main advantage?

User space – example 1

- Program more complex than the first one
- This program read the rtc device
  - Real-time clock is a char device

```
The /dev/rtc interface will block on
reads until an interrupt has been
received. If a RTC interrupt has already
happened, it will output an unsigned
long and than block. The output value
contains the interrupt status in the low
byte and the number of interrupts since
the last read in the remaining high
bytes.
```

  - From Linux sources

User space – example 1

- Program set up rtc device to generate an interrupt series with a certain frequency
- A read block the program (suspend state)
- When an interrupt arise, the system activate the program (ready state)
  - With *time-slice* scheduler, we don't know what process gains CPU
    - A high latency is experienced
  - With FIFO scheduler, we exactly know what process gains the CPU
    - A limited latency is experienced

User space – example 1

- The program show how many interrupts occurs between read() syscall and program execution
- Differences between schedulers are significant in high load systems
  - Load doesn’t only mean CPU!!
  - This measure is affect by many issues
    - Disk load
    - CPU load
    - …
  - Typically, FIFO schedulers may significantly reduce variance

Kernel space

- To improve predictability of latency, we must move in kernel space
- Kernel could interrupt user application, gaining CPU control
    - Kernel preemption (2.6 feature)
- Direct management of hardware interrupt drastically reduces latency
- Using kernel timer to define periodic tasks
- Improve system time granularity

Kernel space – kernel timer

- This timer execute a certain function at a certain expires time
  - If we want a periodic behaviour, this function could reprogram the timer
- Kernel timer is programmed with system ticks
  - System ticks is the finest time grain
- The number of ticks generated from boot time is stored in the `jiffies` variable
- `HZ` symbol contain the number of ticks needed to build 1 second

Kernel space – kernel timer

- A kernel timer is represented with a

```
struct timer_list kt;
```

- The field `function` points to the function to execute
- The field `expires` contain the time to run the function

Kernel space – kernel tick granularity (2.4)

- Tick resolution is defined with HZ
- Typically we have (`include/asm/param.h`)

```
#define HZ 100
```

- Low value
  - Coarse time granularity
  - Low overhead to serve interrupts
- High value
  - Fine time granularity
  - High overhead to serve interrupts

Kernel space – kernel tick granularity (2.6)

- Tick resolution is configured during kernel configuration
- In `include/asm/param.h`

```
#ifdef __KERNEL__
# define HZ         1000
  /* Internal kernel timer frequency */
# define USER_HZ    100
  /* .. some user interfaces are in "ticks" */
# define CLOCKS_PER_SEC (USER_HZ)
  /* like times() */
#endif
```

Bibliography

- Giometti R. “Il Soft “real-time” del pinguino”, Linux&C, Numero 40, Giugno 2004