Operating System Design and Implementation

Process Management – Part II

Shiao-Li Tsao
CPU runs Process A
System call
Program, process, thread
Fork

Execute
Schedule
Context switch
CPU runs Process B
Process schedule and context switching in Linux

• Scheduling
  – Find the next suitable process to run

• Context switch
  – Store the context of the current process, restore the context of the next process
Scheduler in Details

OS
- scheduler
- context switch

Process #B

Process #A

OS
- scheduler
- context switch

Process #B

Process #A
Process schedule and context switching in Linux

• When is the scheduler be invoked
  – Direct invocation vs. Lazy invocation
  – When returning to user-space from a system call
  – When returning to user-space from an interrupt handler
  – When an interrupt handler exits, before returning to kernel-space
  – If a task in the kernel explicitly calls schedule()
  – If a task in the kernel blocks (which results in a call to schedule())
Interrupt Basics

Source: Qing Li “real-time concepts for embedded systems”
# X86 Interrupts

<table>
<thead>
<tr>
<th>IRQ</th>
<th>Standard Function</th>
<th>Bus Slot</th>
<th>Card Type</th>
<th>Recommended Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>System Timer</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>Keyboard Controller</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>2nd IRQ Controller Cascade</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>Real-Time Clock</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>Avail. (as IRQ2 or IRQ9)</td>
<td>Yes</td>
<td>8/16-bit</td>
<td>Network Card</td>
</tr>
<tr>
<td>10</td>
<td>Available</td>
<td>Yes</td>
<td>16-bit</td>
<td>USB</td>
</tr>
<tr>
<td>11</td>
<td>Available</td>
<td>Yes</td>
<td>16-bit</td>
<td>SCSI Host Adapter</td>
</tr>
<tr>
<td>12</td>
<td>Mouse Port/Available</td>
<td>Yes</td>
<td>16-bit</td>
<td>Mouse Port</td>
</tr>
<tr>
<td>13</td>
<td>Math Coprocessor</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>Primary IDE</td>
<td>Yes</td>
<td>16-bit</td>
<td>Primary IDE (hard disks)</td>
</tr>
<tr>
<td>15</td>
<td>Secondary IDE</td>
<td>Yes</td>
<td>16-bit</td>
<td>2nd IDE (CD-ROM/Tape)</td>
</tr>
<tr>
<td>3</td>
<td>Serial 2 (COM2:)</td>
<td>Yes</td>
<td>8/16-bit</td>
<td>COM2:/Internal Modem</td>
</tr>
<tr>
<td>4</td>
<td>Serial 1 (COM1:)</td>
<td>Yes</td>
<td>8/16-bit</td>
<td>COM1:</td>
</tr>
<tr>
<td>5</td>
<td>Sound/Parallel 2 (LPT2:)</td>
<td>Yes</td>
<td>8/16-bit</td>
<td>Sound Card</td>
</tr>
<tr>
<td>6</td>
<td>Floppy Controller</td>
<td>Yes</td>
<td>8/16-bit</td>
<td>Floppy Controller</td>
</tr>
<tr>
<td>7</td>
<td>Parallel 1 (LPT1:)</td>
<td>Yes</td>
<td>8/16-bit</td>
<td>LPT1:</td>
</tr>
</tbody>
</table>

## Vector range

<table>
<thead>
<tr>
<th>Range</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–19 (0x0–0x13)</td>
<td>Nonmaskable interrupts and exceptions</td>
</tr>
<tr>
<td>20–31 (0x14–0x1f)</td>
<td>Intel-reserved</td>
</tr>
<tr>
<td>32–127 (0x20–0x7f)</td>
<td>External interrupts (IRQs)</td>
</tr>
<tr>
<td>128 (0x80)</td>
<td>Programmed exception for system calls</td>
</tr>
<tr>
<td>129–238 (0x81–0xee)</td>
<td>External interrupts (IRQs)</td>
</tr>
</tbody>
</table>
Time Interrupt Basics

1. Process #A
2. Vector table
3. timer_ISR
4. context switch
5. scheduler

(1) (2) (3) (4) (5) (5')
**Process Context**

- **CPU runs user codes**
  - User space
  - Kernel space
  - System calls
    - int $0x80/Exception vector 128/system_call()
      - eax: system no.
  - System call complete

- **CPU runs kernel on behalf of user process**
  - Kernel is in process context
  - User stack
  - Kernel stack
  - PCW
  - PC
Interrupt Context

CPU runs user codes

Interrupt occurs

User space

Kernel space

Interrupt handler returns

CPU runs interrupt handler (or called ISR) in the kernel space

Kernel is in interrupt context

User stack

Interrupt stack

PCW

PC
Interrupt Context

CPU runs user codes

System calls

User space

Kernel space

CPU runs interrupt handler (or called ISR) in the kernel space

Kernel is in interrupt context
When returning to user-space from a system call
If a task in the kernel blocks (which results in a call to schedule())
When returning to user-space from an interrupt handler

Diagram showing the flow of operations from user-space to kernel-space and back. The diagram includes labels for AP1, AP2, AP3, Lib1, Lib2, Syscall, Klib1, Klib2, Schr, Drv1, Drv2, and HW layers.
When returning to user-space from an interrupt handler
When an interrupt handler exits, before returning to kernel-space...
If a task in the kernel explicitly calls schedule()
Process context + interrupt context

1. AP calls read()
2. read() lib
3. read() syscall
4. syscall handler
5. VFS/buffer cache
6. FS
7. Device driver
8. Set to sleep
9. Schedule()
10. Signal process #1

User space

Kernel space

Process #1 context
Process #2 context
Interrupt #1 context
User preemption

- User preemption occurs when the kernel is in a safe state and about to return to user-space.
User preemption

- User preemption occurs when the kernel is in a safe state and about to return to user-space.
Kernel preemption

- Linux kernel is possible to preempt a task at any point, so long as the kernel does not hold a lock.
Preemptive Kernel

- Non-preemptive kernel supports user preemption
- Preemptive kernel supports kernel/user preemption

- Kernel can be interrupted $\neq$ kernel is preemptive
  - Non-preemptive kernel, interrupt returns to interrupted process
  - Preemptive kernel, interrupt returns to any schedulable process
Preemptive Kernel

• 2.4 is a non-preemptive kernel
• 2.6 is a preemptive kernel
• 2.6 could disable CONFIG_PREEMPT
Preemptive Kernel

User process #1

<table>
<thead>
<tr>
<th>User space</th>
<th>Kernel space</th>
</tr>
</thead>
<tbody>
<tr>
<td>System call</td>
<td>Interrupt handler</td>
</tr>
<tr>
<td>lock</td>
<td>lock</td>
</tr>
</tbody>
</table>

Check need_resched
Call schedule()
Check preempt_count
About to return to user space

User process #2
Preemptive Kernel

• How difficult to implement a preemptive kernel?

```
User process #1

User space

Kernel space

System call #1

Interrupt handler

Call schedule()
Check need_resched
About to return to user space

System call #1

User process #2

Kernel code must be reentrant
```
Single Core vs. Multi-core

- **Single processor/single core**
- **Multiple processor/single core**
- **Single processor/single core/hyper threading**
- **Single processor/multi core/separated cache**
- **Single processor/multi core/shared cache**
- **Single processor/multi core/hyper threading**
Single Core vs. Multi-core

Main Memory

Linux Kernel

shared resources for all cores (critical section)

shared resources per core (critical section)
Schedule Algorithms

- Think about yourself (your homework schedule)
  - Given that you have a lot of homework to do, each with a deadline
  - Profs. continue assigning new homework

- What is the next homework to do? (next task to schedule)
- Why should we stop a homework? (time to schedule)
- How long can we concentrate on a homework? (scheduling period)
- How long do we spend to determine the next homework? (scheduling algorithm overhead)
- How much effort do we spend to switch homework? (context switch overhead)
- What is the importance of a homework? (priority of a job)
- How long does a homework need? (job length)
How Linux Scheduler Works

Priority = static priority (nice) +
dynamic priority (heuristic)

Time slice = Func(static priority)

Nice (-20 to +19): normal tasks OR
Real-time priority (0 to 99) : real-time tasks

From parent’s
# Timeslice

- **Timeslice function**

<table>
<thead>
<tr>
<th>Type of Task</th>
<th>Nice Value</th>
<th>Timeslice Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially created</td>
<td>parent's</td>
<td>half of parent's</td>
</tr>
<tr>
<td>Minimum Priority</td>
<td>+19</td>
<td>5ms (MIN_TIMESLICE)</td>
</tr>
<tr>
<td>Default Priority</td>
<td>0</td>
<td>100ms (DEF_TIMESLICE)</td>
</tr>
<tr>
<td>Maximum Priority</td>
<td>-20</td>
<td>800ms (MAX_TIMESLICE)</td>
</tr>
</tbody>
</table>

\[
\text{base time quantum} = \begin{cases} 
(140 - \text{static priority}) \times 20 & \text{if } \text{static priority} < 120 \\
(140 - \text{static priority}) \times 5 & \text{if } \text{static priority} \geq 120 
\end{cases} 
\]
Process schedule and context switching in Linux

• Priority-based scheduler

• Dynamic priority-based scheduling
  – Dynamic priority
    • Normal process
      – nice value: -20 to +19 (larger nice values imply you are being nice to others)
  – Static priority
    • Real-time process
      – 0 to 99
  – Total priority: 140
Linux O(1) scheduler

140-bit priority array

- bit 0 (priority 0)
- bit 7 (priority 7)
- bit 139 (priority 139)

- schedule()
  - sched_find_first_set()

- run the first process in the list

- lists of all runnable tasks, by priority
- list of runnable tasks for priority 7
Recalculating timeslice

Timeslice = Func(static priority)
Calculating Priority

- static_prio = nice
- Prio = nice – bonus + 5
  
  dynamic priority = max (100, min (static priority - bonus + 5, 139))

- Heuristic
  - sleep_avg: (0 to MAX_SLEEP_AVG(10ms))
  - sleep_avg+=sleep (becomes runnable)
  - Sleep_avg-=run (every time tick when task runs)
Sleeping and waking up

```
add_wait_queue(q, &wait);
while (!condition) {
    set_current_state(TASK_INTERRUPTIBLE); /* or TASK_UNINTERRUPTIBLE */
    if (signal_pending(current))
        /* handle signal */
        schedule();
}
set_current_state(TASK_RUNNING);
remove_wait_queue(q, &wait);
```
## System calls related to scheduling

<table>
<thead>
<tr>
<th>System call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nice( )</td>
<td>Change the static priority of a conventional process</td>
</tr>
<tr>
<td>getpriority( )</td>
<td>Get the maximum static priority of a group of conventional processes</td>
</tr>
<tr>
<td>setpriority( )</td>
<td>Set the static priority of a group of conventional processes</td>
</tr>
<tr>
<td>sched_getscheduler( )</td>
<td>Get the scheduling policy of a process</td>
</tr>
<tr>
<td>sched_setscheduler( )</td>
<td>Set the scheduling policy and the real-time priority of a process</td>
</tr>
<tr>
<td>sched_getparam()</td>
<td>Get the real-time priority of a process</td>
</tr>
<tr>
<td>sched_setparam()</td>
<td>Set the real-time priority of a process</td>
</tr>
<tr>
<td>sched_yield( )</td>
<td>Relinquish the processor voluntarily without blocking</td>
</tr>
<tr>
<td>sched_get_priority_min( )</td>
<td>Get the minimum real-time priority value for a policy</td>
</tr>
<tr>
<td>sched_get_priority_max( )</td>
<td>Get the maximum real-time priority value for a policy</td>
</tr>
<tr>
<td>sched_rr_get_interval( )</td>
<td>Get the time quantum value for the Round Robin policy</td>
</tr>
<tr>
<td>sched_setaffinity( )</td>
<td>Set the CPU affinity mask of a process</td>
</tr>
<tr>
<td>sched_getaffinity( )</td>
<td>Get the CPU affinity mask of a process</td>
</tr>
</tbody>
</table>
How Linux Scheduler Works

SCHED_FIFO or SCHED_RR
Priority = static priority (nice)
Time slice = N/A (FIFO)
Time slice = config

PD #2
Nice (-20 to +19): normal tasks OR
Real-time priority (0 to 99): real-time tasks

Process list
Run queue
Process schedule and context switching in Linux

• **Context switch**
  – Hardware context switch
    • Task State Segment Descriptor (Old Linux)
  – Step by step context switch
    • Better control and optimize

• **Context switch**
  – switch_mm()
    • Switch virtual memory mapping
  – switch_to()
    • Switch processor state

• Process switching occurs only in kernel mode
• The contents of all registers used by a process in User Mode have already been saved
Scheduler in Details

- CPU
  - IP
  - SP
  - EAX
  - EBX

Task state segment

- OS
  - scheduler
  - context switch

- Process #B
- Process #A

- TSS (A)
- TSS (B)
Scheduler in Details

CPU

Task state segment

Process #A

TSS (A)  TSS (B)

OS

scheduler

context switch

Process #B

I/O bit map

Loaded but not stored

IP
SP
EAX
EBX
How x86 helps in Context Switch

- Kernel code seg.
- Kernel data seg.
- GDT
- LDT #1
- TSS #1
- LDT #2
- Task 2 code seg.
- Task 2 data seg.
- Kernel code seg.
- Kernel data seg.
- GDT
- LDT #1
- TSS #1
- LDT #2
- Task 1 code seg.
- Task 1 data seg.

LDT:
- NULL
- Task k text
- Task k data

GDT:
- LDT k
- TSS k
- ...
- LDT 1
- TSS 1
- ...
- Kernel Data
- Kernel Text
Process related terms

CPU

Storage

program

Memory

process

Code segment

Data segment

Stack segment

Is that good enough?
If not, why?

Physical memory might be discontinuous
Process related terms (Cont.)

- Storage
- Program
- Process
- Related info
- Thread
- Dedicated pages
- Code segment
- Data segment
- Stack segment
- Registers
- CPU
Process related terms (Cont.)

- Depending on OS designs
Process related terms (Cont.)

- Linux lightweight process
Discussion on 3/22

```c
#include <stdio.h>
void main()
{
    int x = 38;
    printf("Address of x = %p\n", &x);
}
```

/a.out
Address of x = 0×00010000

Cache, TLB, virtual address, logical address, physical address, ...

0×00010000???