Introduction to Scheduling Concepts

Michael E. Locasto
# Agenda

<table>
<thead>
<tr>
<th>Scheduling Concepts</th>
<th>Scheduling Machinery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduling motivation / considerations</td>
<td>sched.h and task_struct support</td>
</tr>
<tr>
<td>Types of processes and scheduling metrics</td>
<td>General scheduling algorithms</td>
</tr>
<tr>
<td>Recall: task state graph</td>
<td>Scheduling in Linux 2.4</td>
</tr>
<tr>
<td></td>
<td>Scheduling in Linux 2.6</td>
</tr>
</tbody>
</table>
Reminder: System Startup Leads to...

We've seen how the OS can transition from sequential code execution to concurrent execution; it:

• initializes the scheduler
• creates two processes (via do_fork and via sys_execve)
• enables interrupts;
• and then allows the newly created scheduler to begin choosing between these two processes (0/swapper and 1/init).

• In the meanwhile, one of the processes (the **init** process) begins creating children via do_fork() and sys_execve()
...the “problem” of scheduling

This procedure naturally brings up the question: as more processes are created, and some hundreds or thousands of processes come to exist, how does the OS choose which process should be given the CPU?

The OS asks itself: to whom do I give the CPU?
Which process gets the CPU?

The CPU(s) is/are a critical system resource.

When/how often does the OS ask itself this question?

How does it determine which process gets it next?

Why? (i.e., what are the justifications for the choice?)
SCHEDULING IS A MECHANISM THAT CREATES THE ILLUSION OF SIMULTANEOUS EXECUTION OF MULTIPLE PROCESS.
NB: Process List vs. Run Queue

The list of all processes is logically distinct from the data structure used to organize processes into groups ready to be scheduled.

The concepts are different, but implementations may share a data structure or be two completely separate lists.
Considerations for Scheduling

### Scheduling Metrics?

- **TIME**
- **WORK TYPE**
- **RESPONSIVE** (delay)
- **CONSISTENT** (delay variance)

### Priority

### Fairness

### Efficiency

### Progress

### Quantum

### Preemption
Critical Concept: Preemption

The kernel can interrupt a running process at any point in time or when its quantum (see next slide) expires to do something else more useful.

Contrast: preemption vs. cooperative multitasking (why is OS involved in scheduling at all?)
Quantum

The default amount of time given to processes to accomplish progress

If too short, context switch overhead is high

If too long, concurrency disappears

\[ \text{my\_stopped\_time} = N \times \text{quantum\_length} \]
## Types of Processes

<table>
<thead>
<tr>
<th>Traditional Classification</th>
<th>Alternative Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-bound (many CPU and memory operations: scientific computing, financial software, rendering)</td>
<td>Interactive</td>
</tr>
<tr>
<td>I/O-bound (much use of devices; may spend time waiting for input)</td>
<td>Batch</td>
</tr>
<tr>
<td></td>
<td>Real-time</td>
</tr>
</tbody>
</table>
These Types Are Somewhat Orthogonal

A batch process can be I/O bound or CPU bound

database
rendering

Real-time programs are explicitly labeled as such
SCHEDULING POLICY AND ALGORITHMS
Conflicting Objectives

Fast response time (processes seem responsive)

Good throughput for background processes

Avoiding “starvation”

*Scheduling policy*: rules for balancing these needs and helping decide what task gets CPU
NICE(1) User Commands NICE(1)

NAME

nice - run a program with modified scheduling priority

SYNOPSIS

nice [OPTION] [COMMAND [ARG]...]

DESCRIPTION

Run COMMAND with an adjusted niceness, which affects process scheduling. With no COMMAND, print the current niceness. Nicenesses range from -20 (most favorable scheduling) to 19 (least favorable).

-n, --adjustment=N
   add integer N to the niceness (default 10)

--help display this help and exit

--version
   output version information and exit

NOTE: your shell may have its own version of nice, which usually supersedes the version described here. Please refer to your shell’s documentation for details about the options it supports.
NAME

nice - change process priority

SYNOPSIS

#include <unistd.h>

int nice(int inc);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

nice(): _BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE

DESCRIPTION

nice() adds inc to the nice value for the calling process. (A higher nice value means a low priority.) Only the superuser may specify a negative increment, or priority increase. The range for nice values is described in getpriority(2).

RETURN VALUE

On success, the new nice value is returned (but see NOTES below). On error, -1 is returned, and errno is set appropriately.
[michael@xorenduex ~]$ !360
ssh -i ~/.ssh/cpsc457svnkey.pem ec2-user@ec2-50-19-41-44.compute-1.amazonaws.com
Last login: Thu Jan 26 12:43:26 2012 from s010600112514707c.cg.shawcable.net

    _\_\_/       Amazon Linux AMI
   (   )

See /usr/share/doc/system-release/ for latest release notes.
There are 20 security update(s) out of 28 total update(s) available
[ec2-user@ip-10-243-89-228 ~]$ sudo su
[root@ip-10-243-89-228 ec2-user]# nice -n -20 yes "no" > out.dat
^C
[root@ip-10-243-89-228 ec2-user]# head out.dat
no
no
no
no
no
no
no
no
no
no

[root@ip-10-243-89-228 ec2-user]#
[ec2-user@ip-10-243-89-228 ~]$ nice -n 20 yes "yes" > out.dat
^C
[ec2-user@ip-10-243-89-228 ~]$ ll -h
total 286M
drwxrwxr-x 4 ec2-user ec2-user 4.0K Jan 14 23:45 courseadmin
-rw-rw-r-- 1 ec2-user ec2-user 286M Jan 26 13:14 out.dat
[ec2-user@ip-10-243-89-228 ~]$ cat out.dat | uniq -c
^C
[ec2-user@ip-10-243-89-228 ~]$ cat out.dat | sort -r | uniq -c
 13844479 yes
    1 nyes
 81387520 no
[ec2-user@ip-10-243-89-228 ~]$
Scheduling Algorithms

First Come First Served

Shortest Job Next

Round-robin

Some simple examples: A, B, C
LINUX SUPPORT FOR SCHEDULING
Flags for Scheduling Policy

```c
/*
 * Scheduling policies
 */
#define SCHED_NORMAL 0
#define SCHED_FIFO 1
#define SCHED_RR 2
#define SCHED_BATCH 3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE 5

#ifndef __KERNEL__

struct sched_param {
  int sched_priority;
};
```
/*
 * Task state bitmask. NOTE! These bits are also encoded in fs/proc/array.c: get_task_state().

 * We have two separate sets of flags: task->state is about runnability, while task->exit_state are about the task exiting. Confusing, but this way modifying one set can't modify the other one by mistake.
 */

#define TASK_RUNNING 0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_STOPPED 4
#define TASK_TRACED 8

/* in tsk->exit_state */
#define EXIT_ZOMBIE 16
#define EXIT_DEAD 32

/* in tsk->state again */
#define TASK_DEAD 64
#define TASK_WAKEKILL 128

/* Convenience macros for the sake of set_task_state */
#define TASK_KILLABLE (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED (TASK_WAKEKILL | TASK_STOPPED)
#define TASK_TRACED (TASK_WAKEKILL | TASK_TRACED)
struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
}

/* task state */
struct linux_binfmt *binfmt;
int exit_state;
int exit_code, exit_signal;
int pdeath_signal; /* The signal sent when the parent dies */
/* ??? */
unsigned int personality;
unsigned did_exec:1;
pid_t pid;
pid_t tgid;

#ifndef CONFIG_CC_STACKPROTECTOR
    /* Canary value for the -fstack-protector gcc feature */
    unsigned long stack_canary;
#endif
The 2.6 runqueue

All processes in TASK_RUNNING state
1 runqueue per-CPU

Each runqueue maintains 2 arrays (active, expired)
Each array has 140 doubly linked lists (1 for each possible priority value)
2.4 vs. 2.6 scheduler

Runqueue structure
Selecting the next process:
  2.4: O(n): iterate over tasks, calculate, switch to
  2.6: O(1): main idea: update priority, grab off front

2.6: actively update dynamic priority based on load and running time of each process (tries heuristics to guess @ batch vs. interactive)

This migrates a process through the runqueue array levels
/* 
 * schedule() is the main scheduler function.
 */ 
asmlinkage void __sched schedule(void) 
{ 
    struct task_struct *prev, *next; 
    unsigned long *switch_count; 
    struct rq *rq; 
    int cpu; 

    need_resched: 
    preempt_disable(); 
    cpu = smp_processor_id(); 
    rq = cpu_rq(cpu); 
    rcu_qsctr_inc(cpu); 
    prev = rq->curr; 
    switch_count = &prev->nivcsw; 
    release_kernel_lock(prev); 

    need_resched_nonpreemptible: 
    schedule_debug(prev); 

    if (sched_feat(HRTICK)) 
        hrtick_clear(rq);
/*
 * 'schedule()' is the scheduler function. It's a very simple and nice
 * scheduler: it's not perfect, but certainly works for most things.
 *
 * The goto is "interesting".
 *
 * NOTE!! Task 0 is the 'idle' task, which gets called when no other
 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
 * information in task[0] is never used.
 */

asmlinkage void schedule(void)
{

    struct schedule_data * sched_data;
    struct task_struct *prev, *next, *p;
    struct list_head *tmp;
    int this_cpu, c;

    spin_lock_prefetch(&runqueue_lock);
}
Scheduling Metrics: Events

Submission Time / Arrival Time

Completion Time

Time of first response (first given CPU)
Scheduling Metrics

Avg. Turnaround Time:
\[
\text{average( time( [Arrival .. Completion] ) )}
\]

Response Time:
\[
\text{time( [Arrival..FirstResponse] )}
\]

Avg. Wait Time:
\[
\text{average( time ( [Runnable && !Running ] ) )}
\]
REFERENCE SLIDES
Reference Material

ULK: Chapter 7 “Process Scheduling”

MOS: Chapter 2
Quantum (cont.)

But does a long quantum make interactive applications unresponsive?

No...these have relatively high priority, so they can preempt other (batch) processes