CprE 308 Laboratory 5-b Homework: Analysis of a Round Robin Scheduler

Department of Electrical and Computer Engineering
Iowa State University

Spring 2014

1 Introduction

In this lab, you will be studying Hewlett-Packard's Plug-in Scheduler Policies for Linux. Linux already employs modules to pull optional device drivers into your kernel without rebooting or recompiling; this patch alters the Linux kernel to support kernel modules that can change the scheduler policy on the fly. The patch is made to apply to the 2.2 and 2.4 versions of the linux kernel.

In this lab, you will analyze two kernel modules that implement schedulers. This will involve reading a great deal of C, searching through source code, and a rudimentary understanding of several structures and algorithms provided by the Linux kernel.

The two sample schedulers that are provided with this patch are:

- const_sched.c – A simple constant time round robin scheduler
- pset.c – Implements processor sets. In other words, a process can only run on a specific group of processors. This is used to guarantee that mission critical applications have the CPU time they need and/or to manage the usage of large systems.

We're going to go on a tour of both modules and touch on some of the highlights. The pluggable schedulers use several structures that you need to be aware of. Every scheduler module initializes and registers an instance of `sched_policy`, and sets up an interface to `/dev/sched`. In addition, modules have to register themselves when they're loaded into the kernel and unregister when they're unloaded. Most importantly, we will look at the functions that implement the scheduling policy (the code that picks which task to run next).

2 const_sched.c

Skim over the const_sched.c source file, and then answer the following questions (enable line numbers in your editor or use `nl const_sched.c > const_sched_numbered.c` to number the lines).

2.1 Preprocessor directives

C compilers use a preprocessor to alter text before compilation. You should already be familiar with `#include` – it literally inserts the text of another file into your source file. If you're curious, you can see your source code after the preprocessing phase by compiling using the `-E` option.

You should also be familiar with `#define`. The line `#define X Y` tells the preprocessor to replace every X in the code with Y before compiling.

2 pts What does the `#define` on line 26 do?

`#if` is another preprocessor directive. In the following code, if X is true, then the preprocessor includes `CODE`, otherwise it includes `MORECODE`.

```
#if X
CODE
#else
MORECODE
#endif
```

When compiling the kernel, `LINUX_VERSION_CODE` will be replaced with a number representing the version of the Linux kernel. For example, the number will be 0x020406 for kernel 2.4.06.

2pts Which lines are included and which lines are excluded due to the `#if` on line 43, assuming we are using kernel 2.4.06?

`struct file_operations ctrr_fops` is related to `/dev/sched`. It isn't used for these labs, but you will see some code for the interface in pset.c.

Lines 50 and 51 have declarations with sched_policy in them. `NR_CPUS` is defined as the number of CPUs the Linux kernel will use.

2pts What is the difference between the two? (Hint: describe the type of each declaration)

2.2 The sched_policy structure

`sched_policy` contains the basic information needed by a scheduler policy. Note the function pointers stored in this struct:

```
struct sched_policy {
    int sp_runnable; /* number of members on the run q */
    int (*sp_preemptability)(struct task_struct *, struct task_struct *, int);
    void (*sp_choose_cpu)(struct task_struct *);
    void (*sp_handle_ticks)(struct task_struct *, int);
    struct task_struct * (*sp_choose_task)(struct task_struct *, int);
    char sp_private[0]; /* per-policy private data. MUST go last */
};
```

The module must define the function pointers `sp_preemtability` and `sp_choose_task`. `sp_choose_task` is called every tick in order to determine the next task to run on a particular cpu. This is where the core of the scheduler lies. `sp_preemtability` is called in order to determine if a process may be preempted by another.

By now you should be getting the feeling that the scheduling module is just providing bits of code to some other part of the kernel. The mainline scheduler in Linux is actually somewhere else, and it simply calls the functions in this module when it need to decide which task to run next, or whether a particular task should be allowed to preempt the current task.

2.3 Initialization

Skip down to the code that registers the module (line 154). The code in `init_module` is run when the module is loaded into the kernel.

2 pts What does the following code do (line 158)?

```
struct sched_policy *new = &round_robin;
```

Line 160 zeros out the memory. This is a good way to initialize memory in case the structure changes.

4 pts What is the purpose of lines 161 and 162?

When registering a scheduler module, an array of sched_policy is required that contains a pointer to a policy for each CPU. In this case the policy is the same for all CPUs.

2 pts Find the lines that initialize this array.

Lines 168-170 register the scheduler and handle any errors. Lines 173-178 set all of the tasks in the system to use the new policy.

2 pts Which lines handle the unregistration of the module?

2.4 Scheduling

Go back to line 57.

3 pts Based on line 161, what must this function do? `ctrr_choose_task` takes two parameters: the currently running task, and the number of the CPU running the scheduler.

All tasks are represented by a `task_struct`. An abbreviated version is provided here:

```
struct task_struct{
    volatile long state;
    long counter;
    long nice;
    int has_cpu, processor;
    struct task_struct *next_task,*prev_task;
    struct mm_struct *active_mm;
    struct sched_policy *alt_policy;
    pid_t pid;
};
```

The fields in `task_struct` store various properties of a task:

- state – Can be one of: `TASK_RUNNING`, `TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE`, `TASK_ZOMBIE`, or `TASK_STOPPED`.
- counter – Number of ticks left in this process, the number of ticks is linearly proportional to (20-nice).
- nice – How willing a process is to let other processes run. (see the renice command)
- has_cpu – true if this process is currently assigned to a cpu.
- processor – CPU this process is assigned to; used to determine the processor switch penalty.
- next_task, prev_task – Used by macros to loop through all the processes, which are stored in doubly-linked lists. You should never read or write these; use the macros instead.
- active_mm – Used to determine the memory mapping switch penalty. Threads within a process will have the same value for this pointer.

- alt_policy – NULL if there is no policy module loaded. Otherwise it points to a sched_policy object.
- pid – Process id.

Have a look at lines 67-71. `idle_task()` returns the idle task for a particular CPU, and `SCHED_YIELD` is defined as follows:

```
/*
 * This is an additional bit set when we want to
 * yield the CPU for one re-schedule.
 */
#define SCHED_YIELD 0x10
```

4 pts Describe lines 67-70 in words. Translate the syntax into plain English one line at a time, but keep it brief.

Line 74 is an example of a Linux macro. Several macros are provided in the Linux Kernel header files to traverse and manage lists. In this lab the macros of interest are `list_for_each` and `list_entry`. Here is an example where the macros are used to traverse all running processes:

```
struct task_struct *p;
struct list_head *tmp;
list_for_each(tmp, &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    /* insert code to run for each list element */
}
```

Note that the use of "continue" causes execution to resume at the start of a loop, and `list_for_each` is defined as a type of loop.

On line 76 you can find another preprocessor directive, `CONFIG_SMP`, which is defined when Linux is being compiled for multiple processors. The configuration we will be using has only one processor, so assume this is not defined (false). You can assume the same about `__SMP__`.

Lines 90 and 91 should be fairly self-explanatory. All you need to know is what the global variables are:

- runqueue_head – Top of running process queue, used in list macros
- run_list – List of currently running processes, used in list macros

A process's tick counter is decremented outside of the module every tick it's run. It is the responsibility of the module to replenish the counter, however. This can be done using the `NICE_TO_TICKS` macro to set `p->counter`, which converts a nice value into ticks. The nice value measures how nice this process is. In other words, niceness is how willing it is to yield to other processes; the meaner processes are more likely to run. You can see the nice value under the NI column when you run the top command.

3 pts On what line is the counter replenished? Does this happen for all tasks every time this algorithm is invoked?

2 pts Study lines 97-139. In what case is the idle task returned?

3 pts On line 142, what is the purpose of the function `ctrr_preemptability` (in words)?

3 pset.c

One of the biggest differences between this and the previous scheduler is the use of goodness. Goodness is the measure of a process's right to run. The higher the goodness, the more right a process has to run. In the source code for Linux, `kernel/sched.c` defines goodness' numerical value as:

```
/*
 * This is the function that decides how desirable a process is..
 * You can weigh different processes against each other depending
 * on what CPU they've run on lately etc to try to handle cache
 * and TLB miss penalties.
 *
 * Return values:
 * -1000: never select this
 * 0: out of time, recalculate counters (but it might still be selected)
 * +ve: "goodness" value (the larger, the better)
 * +1000: realtime process, select this.
 */
```

You can find this in `include/linux/sched.h` in the version patched with the HP scheduler. The goodness value is calculated by the following formula:

$$g = c + p + m + (20 - n)$$

Where:
g = goodness
c = ticks left in the counter
p = the processor switch penalty
m = the memory mapping penalty
n = the nice value

The kernel already defines a function to calculate goodness:
`goodness(struct task_struct * p, int this_cpu, struct mm_struct *this_mm)` A process's tick counter is decremented every tick it gets to run. Therefore, if two processes start with the same number of ticks, the one currently running will be less likely to continue running next time. If a process is running on the current CPU, it is rewarded with a higher goodness since switching processors is expensive. The same goes for having to re-map memory.

3 pts Find all of the lines in `pset.c` where goodness is used. Note any differences in the parameters passed.

2 pts What is `IDLE_WEIGHT` defined as? Why? You will need to look in the include files to answer the first part, and the definition of goodness for the second part. (Refer to the patched sched.h file we provide.)

3 pts Where is `can_choose` defined? Where is `is_visible` defined? When answering, pick the correct lines based on the values of `CONFIG_SMP` we gave you. These macros are more useful for multi-processor systems. They're used to check if a task is allowed to execute on a particular processor. You can assume these always return true (otherwise we couldn't use our single processor).

In your own words, answer the following:

4 pts What are the initial values of `high` and `choice`?

3 pts Under what conditions is choice set to `curr_task`? (lines 130-135)

5 pts In one sentence, which task does the code on lines 137-148 pick?

2 pts If there are tasks that are runnable but have no ticks left, then what is the value of `high`? (see line 151)

2 pts Where does line 158 goto?

5 pts Will lines 150-159 ever run twice in a row? Why or why not? In what two cases will it not run, and what task does the algorithm return in those two cases? Look again at lines 130-135.

2 pts extra If this "if" is removed (the whole thing: lines 130-135), in what case will the task chosen be different? Note that the list of tasks is not sorted in any particular manner. Think of cases where some of the tasks have the same goodness, and the current task is closer to the end of the list.

2 pts extra Why include this "if"? Think in terms of penalties incorporated into goodness.

4 Further Reading

If you are interested in how Vista or the 2.6 series of Linux kernels handles scheduling, sections 10.3.3-10.3.4 and 11.4 of your textbook have information on this. Since version 2.6.23 of Linux (released in October 2007), a new scheduler called the Completely Fair Scheduler is used.