Linux Scheduler Workbench

Author: *Tomek Blaszczyk*
e-mail: `tombla@mimuw.edu.pl`

Linux Scheduler Workbench

Contents

1 Introduction

1.1 Document map

This document describes the work I did with Linux scheduler.

Chapter 1 describes principles of the work, a history and Linux Scheduler Workbench (i.e. a kernel patch, scheduling modules, *Visualiser* (`tb-xshow`), scripts that help using the workbench).

Chapter 2 describes changes in the Linux kernel, a scheduling modules interface and an example of a scheduling module.

Chapter 3 is a manual for Linux Scheduler Workbench. It says how to install the patch for the kernel, how to load, activate, deactivate and unload a scheduling module, how to dynamically modify scheduling parameters, how to use `top` and *Visualiser* and how to use different testing environments.

Chapter 4 is for those, who just want to start using better scheduling algorithm without reading 'the whole story'.

1.2 Prerequisites

Reader should be able to patch the kernel, configure it, compile it and boot system using that kernel.

Hardware requirements: Intel based PC, not too fast (I'm not kidding—the slower computer, the easier an observation of scheduling effects), e.g. P166.

Software requirements: Linux system 2.0.35 compatible (I used RedHat 5.1); sources for 2.0.35 kernel; egcs C compiler; X-Windows (for tb-xshow visualiser); QT development library, version 1.33 or higher (for tb-xshow visualiser).

1.3 History

During my studies at Warsaw University, Faculty of Mathematics and Informatics, I decided that my master thesis would concern Linux kernel.

The kernel consists of many, many interesting algorithms so first I decided to describe as many of them as possible. Second thought was that I should try to modify some algorithms, basing on those implemented in other operating systems and described in the literature. Next, I had to narrow down areas I would investigate: I was looking at the algorithms concerning memory management and process management. I worked on sources of kernel 2.0.35. Finally, my master thesis consisted of two parts.

In the first part `vm_area` management algorithm was described. It was famous AVL algorithm—quite complicated, not for the algorithm itself was complicated, but because the implementation was sophisticated. I compared AVL algorithm with linear one (based on linked lists) and with cached versions, where last visited virtual memory area was remembered. The results of my investigations were that it's not worth keeping such a big and hard to modify piece of code in the kernel: timing results were identical for most investigated applications and some differences occurred only for a specially written test program. How (positively!) surprised was I when I found that kernel 2.1.122 (or something around that number) uses... lists with single cache field! My implementation was a little bit simpler: that cache was cleared very cowardly, every time the list was changing, while original development kernel code was better optimized.

The second part of my master thesis concerned scheduling algorithms.

At first I just wanted to hard-code some new scheduling algorithm into the kernel and compare it with Linux algorithm. I did it, but the results were not satisfactory. 'What if I wanted to implement another one?' I thought and started designing a new algorithm and it's implementation. By the time I were almost done with the design I said to myself: 'Hey, there are some general rules that concern every scheduler, why not to make use of it?' That was how Linux Scheduler Workbench was born. I coded an interface for external scheduling algorithm and then I could start implementing scheduling algorithms in separate parts of code: Linux loadable kernel modules. The time necessary to devise a new scheduler shrunk a lot, as I could often unload a bad module, make modifications and run it once again during the same Linux session. *Visualiser*, a tool I wrote in two days, including learning of QT library, was a part of my work. I used it when I tuned up my algorithms. The results of this part of my thesis were very promising—many of the algorithms behaved better than Linux algorithm. It means that response time for interactive processes in loaded system were shorter in my algorithm in comparison with original Linux code.

Well, that's what my master thesis was about. Here, I want present a scheduling part of my work, concentrating on the design and the implementation rather than on a description of the scheduling algorithms.

1.4 What is done and future development

The objective of the work was to create a comprehensive and easy to extend environment for developing scheduling algorithms.

First step, which has been done already, was to implement the environment in 2.0.x kernel, for a single processor architecture. `i386` was selected as a target platform.

Future development tasks:

1. Implementation for a kernel 2.2.x
2. SMP support
3. User/group CPU quota support
4. Smart thread support (special treatment of lightweight processes)

1.5 Linux Scheduler Workbench

Linux Scheduler Workbench (LSW) consists of several parts.

Kernel patch This part mainly implements interface for the external scheduling modules. Another modification was done to provide data for *Visualiser*.

Scheduling modules This part consists of several modules that implement algorithms based on those present in real operating systems and in a literature: classic Unix scheduler, BSD scheduler, SJF (Shortest Job First), SVR4 (System V Release 4). The latter one is highly configurable. You can load your own parameter tables. The package provides tables that make SVR4 scheduler operate like classic SVR4 scheduler, NT scheduler and QNX scheduler.

Testing environments Schedulers operate differently in different environments. For example, SVR4 scheduler was designed to promote I/O-bound tasks over CPU-bound tasks. If you're using a text editor under SVR4 and the system is loaded with many processes that make intensive computations without interacting with users (e.g. renderers, simulators, etc.) you won't feel that the system is loaded. Testing environments run some programs that simulate real world environments. Programs include a CPU-bound task, I/O bound tasks (client/server communicating via *fifo*) and a memory-bound task (program requiring a large amount of a memory and making many page faults).

Visualiser (tb-xshow) A small application for X-Windows that shows an execution profile for selected processes. Can be very useful for parameter tuning and for educational purposes.

2 Scheduling modules

2.1 Design

Linux scheduler code is scattered all over the kernel. For example there are many places where a process yields a processor by simulating an expiration of a time quantum: 0 is assigned to process's `counter` field. On the other hand, the main part of the scheduler code resides in `sched.c` file.

First, I identified points where scheduler code is called. Those points were used afterwards to call the code of the external scheduling modules.

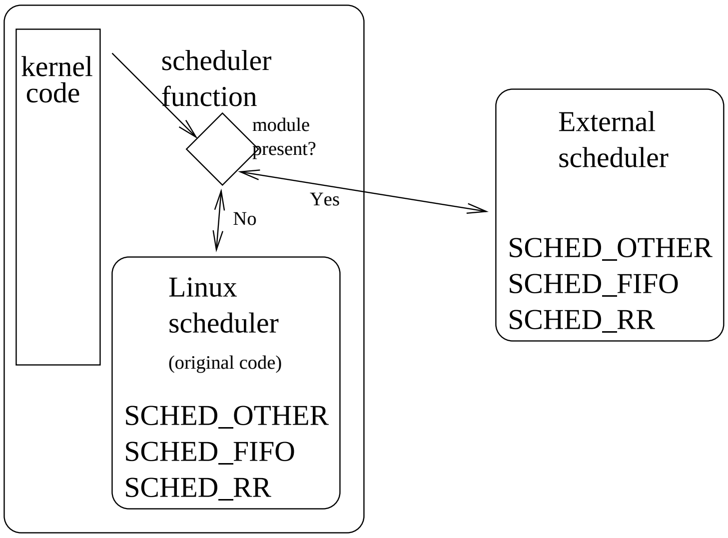
The design of the interface was based on SVR4 (see [5]). In SVR4 each process class (e.g. time-shared, real-time) is handled by a separate scheduling algorithm. I propose a slightly different functionality.

The Linux algorithm takes care of all types (classes) of processes: `SCHED_OTHER`, `SCHED_RR` and `SCHED_FIFO`. Linux scheduler is always present in the kernel code, but some of its actions may be taken over by an alternate, 'external' algorithm (scheduler). The external scheduler must serve all the standard classes of processes, but may also define new classes. For Linux scheduling code is always present in the kernel, it is possible to change external schedulers (some scheduler must take care of processes while changing scheduling modules!). Only one external scheduler can be loaded into the kernel.

The kernel was extended so that the proper code, either the default Linux code or the external module's code is called. See fig. 1.

Next, I had to specify what set of procedures should be implemented in the scheduler. The design of this part is based on the idea of handling events. The events occur mainly when kernel 'wants' to take an action concerning process switching. For example there is an event 'time quantum expired' which yields

Figure 1: Control flow after modifications.



running a code looking for a new candidate to run on the processor. I identified places where 'events' occur. Event `XXX` is handled in the following way:

```
//
// I stated, that this is the place where XXX event occurs
//
// tb_xxx is pointer to the function
if (tb_xxx != NULL ) { // the function is implemented by the module
   (*tb_xxx)(arg1, arg2, ... ) // the function is called
} else { // function is not handled by the module
   ... // original Linux code
}
```

An occurrence of an event `XXX` will be treated from now on like in the code above, i.e. either function `(*tb_xxx)` or original Linux code is called.

There are three process states that are relevant to the scheduling algorithm: a process is running on the processor (*running*), a process is waiting for the processor (*ready*) and a process is sleeping (*sleeping*). We don't bother ourselves with states like *stopped*.

Following events can influent every process in the system:

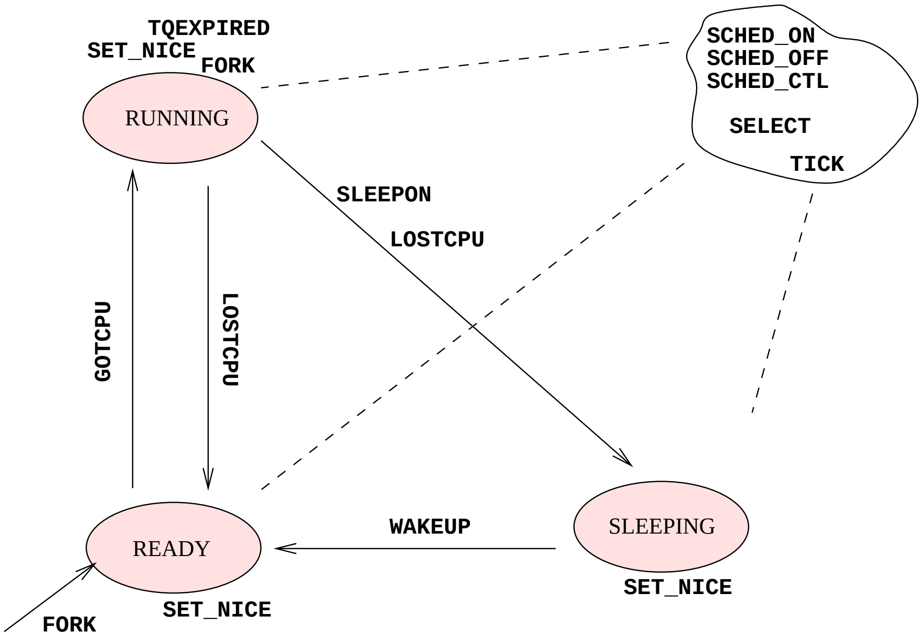
- `TICK`—the event occurs every one second[1].
- `SELECT`—occurs when the scheduler must select a process to run on the processor during scheduling procedure. For example this scheduling procedure is called when a process wakes up.
- `SCHED_ON`—the procedure is called during turning on an external scheduler so that scheduler-specific data can be prepared.
- `SCHED_OFF`—the procedure is called during turning on an external scheduler so that the module can clean-up and prepare processes for Linux scheduler.
- `SCHED_CTL`—the procedure is used to control the scheduler, for example to change running parameters.

Following events affect single processes:

- `FORK`—occurs when a new process is created. Procedure `tb_fork` is called for both parent and child.
- `WAKEUP`—when process wakes up and before is added to the run queue. The scheduler can find out if the process 'really' wakes up, i.e. if was previously removed from the run queue.

[1] I.e. `tb_tick` is called every second

Figure 2: Scheduling events and process states.



- SLEEPON—when a process is going to sleep, after it was removed from the run queue.
- TQEXPIRED—when a process used up an assigned time quantum.
- GOTCPU—when a process is about to get the processor. The procedure is called before a context switch is done.
- LOSTCPU—when a process is to be preempted or yields a processor. The procedure is called for the current process before context switch is done.
- SET_NICE—when static priority is changed. This is the place where Unix values from -20 to 19 can be recalculated to internal ones.

Places of event's occurrences are shown on figure 2. 'Cloudy' events affect all processes.

2.2 Modules interface specification

The module has an access to the following kernel variables:

```
typedef struct task_struct* Process;

int ts_installed;
int ts_on;

int (*ts_sched_on)(void);
int (*ts_sched_off)(void);

int (*ts_sched_ctl)(int command, void* data);

Process (*ts_select)(Process prev, int cpu);

int (*ts_wakeup)(Process tsk);
int (*ts_sleepon)(Process tsk);
int (*ts_tqexpired)(Process tsk);
int (*ts_gotcpu)(Process tsk);
int (*ts_lostcpu)(Process tsk);

int (*ts_tick)(void);
```

```
int (*ts_fork)(Process parent, Process child);

int (*ts_sched_stat)(char* buf);
int (*ts_proc_stat)(Process task, char* buf);
int (*ts_get_pri)(Process task);
int (*ts_get_nice)(Process task);
int (*ts_set_nice)(Process task, int niceval);
```

Installation of the external scheduling module is successful if and only if no other scheduling module is loaded. Examining the flag `ts_installed` checks out a presence of another scheduling module.

The installation procedure assigns module functions to the kernel variables (pointers to functions) and sets `ts_installed` flag.

Unloading the module must reset values of the changed pointers to `NULL` and clear `ts_installed` flag. If the scheduler is active (`ts_on` flag is set) and procedure `ts_sched_off` is implemented then it must be called before unloading the scheduler.

The only event that MUST be handled by an external scheduling module is `SELECT`. All other event handlers are optional.

Here is the specification for the event handlers:

- `ts_sched_on`: The function has no arguments. Returns 0 if scheduler was successfully switched on. Other values mean error.
- `ts_sched_off`: The function switches off the scheduler. The returned value is ignored—the function must always succeed. It's guaranteed that the function is called only if the scheduler was properly switched on.
- `ts_sched_ctl`: The function gets code of the command and a pointer[2]. Function's implementation is module-specific. The function can be called when the scheduler is on or off (but it must be loaded, of course). Function should return 0 if successful, other value to report an error.
- `ts_select`: The only function that must be implemented. Gets a pointer to the control block of a current process and a processor number[3]. Function returns a pointer to the control block of the selected process or `NULL`, when no process can be selected[4].
- `ts_wakeup`: The function should modify dynamic data of the woken process and decide if rescheduling is necessary. The function is called usually before the process is added to the run queue. The exception for that rule is when the process was not removed from the queue[5]. The returned value is ignored.
- `ts_sleepon`: Should handle actions needed when a process is removed from the run queue. The function is called before the process removal. The returned value is ignored.
- `ts_tqexpired`: The function called when a process uses up it's time quantum. The returned value is ignored.
- `ts_gotcpu`: The function is called when a process gets CPU, before a context switch.
- `ts_lostcpu`: The function is called when a process loses CPU, before a context switch.
- `ts_tick`: The function is called every second.
- `ts_fork`: The function called when a new process is created. Function gets pointers to the control blocks of the parent and the child process. On success, the function returns 0, non-zero otherwise.
- `ts_sched_stat`: The function should return a pointer to a string describing scheduler statistics. Not used.
- `ts_sched_proc`: The function should return a pointer to a string describing process parameters. Not used.

[2] E.g. a pointer to the area with arguments.
[3] Ignored, as far.
[4] In this case idle process gets the processor.
[5] It is Linux specific situation.

- ts_get_pri: Gets a pointer to a process control block and returns it's dynamic priority.
- ts_get_nice: Gets a pointer to a process control block and returns it's static priority.
- ts_set_nice: Gets a pointer to a process control block and new nice value. Not used.

Functions ts_tqexpired, ts_wakeup, ts_gotcpu, ts_lostcpu may return value TS_MOVE_LAST. Returning this value forces putting the process (which the procedure is called for) at the end of the run queue. Returning TS_NO_MOVE leaves the process on its place.

Functions ts_get_pri and ts_get_nice are used by *procfs* to show standard parameters for processes.

2.3 Sample implementation

Here, I'll present implementation of Shortest Job First (SJF) algorithm. It is based on a classic Unix algorithm. A dynamic priority is defined as exponential average of time quanta recently used. If a process is waiting too long, it's dynamic priority is temporarily boosted.

The control block for each process has following additional fields:

- counter—like in the Linux algorithm, it's time quantum left to use.
- starttq—a variable used to compute a value of an average time quantum length.
- tq—an exponential average of time quanta used; it's a base for computing the dynamic priority.
- boost—normally 0, but may increase to temporarily boost the dynamic priority of the process; this is to prevent a process starvation

So the code goes here (look at src/mod/sjf2/tb_sjf2.c):

Every control block (task_struct) contains 8 long fields for additional scheduling data (long ts_data[8]). In this case we use fields of indexes 0 (boost field), 1 (starttq field) and 2 (tq field). Macros are defined for convenience only:

```
#define sjf_boost(task)     (task)->ts_data[0]
#define sjf_starttq(task)   (task)->ts_data[1]
#define sjf_tq(task)        (task)->ts_data[2]
```

Turning the scheduler on requires data initialization:

```
int sjf_sched_on(void)
{
   struct task_struct* task;

   for_each_task(task) {
      task->counter = 0;
      sjf_boost(task) = 0;
      sjf_starttq(task) = 0;
      sjf_tq(task) = 0;
   }
   need_resched = 1;
   return 0;
}
```

Turning off yields a control to the standard Linux algorithm. We just clear up:

```
int sjf_sched_off(void)
{
   struct task_struct* p;

   for_each_task(p) {
      p->counter = 0;
   }
   need_resched = 1;
   return 0;
}
```

A process priority is bigger, when the average time quantum is shorter. SCHED_RR and SCHED_FIFO is handled in a standard way:

```
static int sjf_goodness(struct task_struct* task)
{
   int pri;

   if (task->policy != SCHED_OTHER)
      return 1000 + task->rt_priority;

   pri = task->priority + sjf_boost(task) - sjf_tq(task);

   return ( pri<3*DEF_PRIORITY ? pri : 3*DEF_PRIORITY );
}
```

A process with the biggest dynamic priority (goodness value) is chosen. If there is no process to select, NULL must be returned:

```
struct task_struct* sjf_select(struct task_struct* prev, int cpu)
{
   long c;
   struct task_struct *p, *next;

   c = -1000L;
   next = NULL;
   p = init_task.next_run;
   while (p != &init_task) {
      int weight = sjf_goodness(p);
      if (weight > c)
         c = weight, next = p;
      p = p->next_run;
   }

   return next;
}
```

The scheduler boosts temporarily the dynamic priority of a processes that have just woken up by increasing the boost value:

```
int sjf_wakeup(struct task_struct* task)
{
   sjf_boost(task) += 10;

   if ( !need_resched &&
        !task->next_run &&
        sjf_goodness(task) > sjf_goodness(current) )
      need_resched = 1;

   return TS_NO_MOVE;
}
```

When the time quantum expires, the scheduler recalculates the average time quantum and gives a new time quantum:

```
int sjf_tqexpired(struct task_struct* task)
{
   sjf_tq(task) = ( sjf_tq(task) + sjf_starttq(task) - task->counter ) / 2;
   task->counter = sjf_starttq(task) = task->priority;

   return TS_MOVE_LAST;
}
```

The process that got CPU, comes back to it's unboosted dynamic priority—the temporary boost value is cleared:

```
int sjf_gotcpu(struct task_struct* task)
{
   sjf_boost(task) = 0;

   return TS_NO_MOVE;
}
```

If a process have lost CPU and is still is state *running*, it means that it was preempted and in fact uses a longer time quantum. Scheduler doesn't recalculate an average time quantum. In case the process goes to sleep (it's not present on the run queue), it's new average time quantum is recalculated:

```
int sjf_lostcpu(struct task_struct* task)
{
   if ( task->state == TASK_RUNNING )
      return TS_NO_MOVE;

   sjf_tq(task) = ( sjf_tq(task) + sjf_starttq(task) - task->counter ) / 2;
   sjf_starttq(task) = task->counter;

   return TS_NO_MOVE;
}
```

Every 2 seconds each process boosts its priority. The long-sleeping processes, after waking up should get CPU very quickly:

```
int sjf_tick(void)
{
#define BOOST_TICK 2

   struct task_struct* task;
   static ticks = 0;

   if ( ++ticks < BOOST_TICK )
      return 0;
   ticks = 0;

   for_each_task(task) {
      if ( ++sjf_boost(task) > 6*DEF_PRIORITY )
         sjf_boost(task) = 6*DEF_PRIORITY;

   }

   return 0;
}
```

This function returns the dynamic priority value. Handling it in our way provides capability of viewing value of `tq` field in applications like `top`, but may affect applications that make use of that value:

```
int sjf_get_pri ( struct task_struct* task ) {
   return((int) sjf_tq(task));
}

int sjf_get_nice ( struct task_struct* task ) {
   return((int) sjf_boost(task));
}
```

This is Linux module specific functions. We just 'install' handlers for some events and clear up when module is removed from the kernel:

```
int init_module( void )
{
   if ( ts_installed ) {
      printk ( "sjf_install: Alternate scheduler already installed.\n" );
      return -1;
   }

   ts_installed = 1;

   ts_sched_on        = sjf_sched_on;
   ts_sched_off       = sjf_sched_off;
   ts_sched_ctl       = NULL;
   ts_select          = sjf_select;
   // ...and so on...

   printk ( "sjf_install: OK.\n" );
   return 0;
}

void cleanup_module(void)
{
   if ( ts_on && ts_sched_off )
      (*ts_sched_off)();

   ts_on = 0;

   ts_sched_on        = NULL;
   ts_sched_off       = NULL;
   ts_sched_ctl       = NULL;
   // ...and so on...

   ts_installed = 0;
   printk ( "sjf_uninstall: OK.\n" );
}
```

3 Linux Scheduler Workbench How-to

The best way to use this package is to create directory `/home/sched` and unpack an enclosed archive (`tbsched-1.0.tgz`) in that directory. If you're using `bash`, type `source prepare.bash` to prepare the environment (set path and directory information). Please refer to `prepare.bash` file and modify it when you keep the files in a different location than stated above or use other shell.

3.1 How to patch and compile the kernel?

1. Get Linux kernel sources 2.0.35. Install it and go to the source root directory. It's usually `/usr/src/linux-2.0.35`.
2. Type `patch -p1 <FILENAME`, where `FILENAME` is the name of the file with patch (`42_patch_990918.diff`).
3. Configure the kernel and enable kernel modules. You configure the kernel by using `make menuconfig` or `make xconfig`. You can find an example configuration in `config-mod` file in `kernel` directory. You may just load the configuration file into the configuration program. Save the configuration and exit the configuration program.
4. Type `make dep; make clean; make zImage`: this will compile the kernel.
5. Install the kernel (see `lilo` man page for instructions).
6. If directory with new kernel has a name different than `linux`, e.g. `linux-2.0.35`, type `ln -s linux-2.0.35 linux` so that proper header files were visible for compilation of scheduling modules. In short: directory `/usr/src/linux` must contain patched source of the kernel 2.0.35.

3.2 How to compile, install and activate a scheduling module?

To compile a scheduling module, go to a directory with source files, e.g. `cd /home/sched/src/mod/svr4/`. Type `make`.

To install the module type `insmod MODNAME` where `MODNAME` is the name of the module. Module is now installed but not active.

To activate the module use `tb-sched` program (it's located in `/home/sched/src/tb-sched`). Type `tb-sched 1` to activate scheduler, `tb-sched 0` to deactivate it.

Alternatively, you may use `tb-ctrl` script. Type `tb-ctrl MODNAME` to load module. `MODNAME` is written without path and prefix, so to load `/home/sched/src/mod/svr4/tb_svr4.o` module just type `tb-ctrl svr4`. Name `x` unloads currently loaded module. Names `on` and `off` turn sheduler on and off. Type `tb-ctrl` for instructions.

SVR4 scheduler has ability to load different parameter tables. Go to `/home/sched/src/mod/svr4/tab` and type `make`. Use `settbl <TABLE` program to load new parameters, where `TABLE` is the name of the file with parameters. Alternatively, you may use `table` script (see `/home/sched/bin directory`). Use `showtbl` to dump currently used parameters table.

See also `/home/sched/test/sched` directory. It contains scripts that run schedulers and loads various parameters table for SVR4 (e.g. NT, QNX, SV2).

3.3 How to compile and use *Visualiser* (`tb-xshow`)?

To compile `tb-xshow` go to source directory (`/home/sched/src/tb-xshow/`) and type `make`. QT development pack (version 1.33 or higher) must be installed to compile `tb-xshow`.

To use it type `tb-xshow <pid1> <pid2> ...` to show usage scenarios for processes with the specified pids. Green color means *Using CPU*, red means *waiting for CPU*, blue means *sleeping* and black means *no data* (process is running for less than 6 seconds long). Invalid pids (no process of that pid) are shown in red.

The window may be freely resized.

3.4 How to use testing environments?

Go to `/home/sched/test/progs directory` and type `make` to compile programs.

Go to `/home/sched/test/env` directory. There are scripts that set proper environments. The scripts are named `heavy*`, `nice*`, `scrub*` and `www*`. One parameter —on— or `off` is required. *Example:* `heavy10 on` runs 10 CPU-bound processes. You may observe system's behavior under heavy load conditions.

Go to `/home/sched/test/sched` directory. You find convenient scripts that load scheduler, set parameter table (for svr4 scheduler) and turn scheduler on.

4 Kickstart

Prerequisites:

- o Install Linux with 2.0.35 kernel and kernel sources.
- o Install qt-devel-1.33 or higher (to compile and run tb-xshow).

To-do list:

1. Login as root.
2. Type `mkdir /home/sched`.
3. Type `cd /home/sched`.
4. Copy `tbsched.tgz` into that directory.
5. Type `tar xzvf  tbsched.tgz`.
6. Install the patch for the kernel, configure and compile kernel, install new kernel (see section 3.1).
7. Reboot, login as root.

8. Type `cd /home/sched/src/tb-sched; make`.
9. Type `cd /home/sched/src/mod/svr4; make`, and than `cd tab; make`.
10. Type `cd /home/sched/test/progs; make`.
11. *Optional.* Type `cd /home/sched/src/tb-xshow; make`.
12. Start *X-Windows* and run terminal window.
13. Type `cd /home/sched; source prepare.bash`. This will set environment variables.
14. Type `cd test/env; heavy10 on`. This will run 10 heavy processes.
15. Start other apps and observe system behavior.
16. Type `cd /home/sched/test/sched/; p-sv2 on`. This will load and avtivate SVR4 scheduler with *sv2* parameter table.
17. Observe system behavior—the system favors interactive processes (like *X-Windows server*).

References

[1] A. Silberschatz, J. L. Peterson, P. B. Galvin, *Operating systems concenpts*, third edition.

[2] M. J. Bach, *Unix internals (???)*

[3] M. Beck, H. Boehme, *Linux Kernel Internals — Second Edition*, Addison-Wesley, 1998.

[4] U. Vahalia, *Unix Internals — The New Frontiers*, Prentice Hall, 1996.

[5] B. Goodheart, J. Cox, *The Magic Garden Explained — The Internals of System V Release 4*, Prentice Hall, 1994.

[6] D. A. Rusling, *The Linux Kernel*, wyd. HTML, 1999, `http://sunsite.unc.edu/LDP/LDP/tlk/tlk.html`.

[7] M. K. Johnson, A. Rubini, *Linux Kernel Hacker's Guide*, HTML edition, 1997, `http://khg.redhat.com/HyperNews/get/khg.html`.

[8] O. Pomerantz, *The Linux Kernel Module Programming Guide*, HTML edition, 1999, `http://sunsite.unc.edu/LDP/LDP/lkmpg/mpg.html`.