Linux Process Scheduling

Andreas Schlapbach (schlpbch@iamexwi.unibe.ch) Version 0.1 , Thu Mai 2 21:10:27 CEST 2000

This is an attempt to describe the scheduling concepts used in Linux. Scheduling is a core part of every OS. The first section tries to show the dependenies between scheduling and other parts of the system like memory managment or the network subsystem. The second part gives an overview of the algorithms and the data structures used while scheduling. Linux uses a **simple priority based scheduling algorithm** to choose between the current processes in the system. There are two types of processes in Linux, **normal** and **real time**. Real time processes will always run before normal processes and they may have either of two types of policy: **round robin** or **first in first out**. As Linux uses **preemptive scheduling**, every process is given a fixed time slice of 200ms to run. The last section tries to locate these mechanisms in the actual code from the upcoming 2.4 kernel.

Contents

1 Introduction **1**

1.1 Remark . . . 1

1.2 Purpose of the Kernel . . . 2

1.3 Overview of the Kernel Structure . . . 2

2 Scheduling Concepts used by Linux **3**

2.1 Scheduling code - Overview . . . 3

2.1.1 Interlude: **task_struct** . . . 3

2.2 Pseudo - Code . . . 7

3 Linux Scheduling - the Code **8**

3.1 Goodness() . . . 8

3.2 Schedule() . . . 9

4 Time Accounting **14**

4.1 What are bottom halfs? . . . 14

4.2 Time Accounting using bottom halfs . . . 14

5 Resources **16**

1 Introduction

1.1 Remark

Some of the text presented here is compiled together from existing sources. IMHO it would be pointless to try to formulate ideas/concepts in my own words that other more gifted authors have done clear and precise. The sections 1.2 (Purpose of the Kernel) and 1.3 (Overview of the Kernel Structure) are taken from 5 (Conceptual Architecture of the Linux Kernel) by Iwan T. Bowman. The sections 2.1 (Scheduling code -

Overview) and 4.1 (What are bottom halfs?) are taken from 5 (The Linux Kernel) by David A. Rushling. See 5 (resources) for details. I would also like to thank *Rik van Riel* for having answered my questions on 5 (IRC).

All the code examples are taken from the recent development kernel *Linux-2.3-99-pre6* released on April, 13. 2000. This kernel is from the prerelease to the next stable 2.4 kernel. Fundamental changes in the kernel and especially in the scheduling concepts used are *very* unlikely.

1.2 Purpose of the Kernel

The Linux kernel presents a virtual machine interface to user **processes**. Processes are written without needing any knowledge of what physical hardware is installed on a computer – the Linux kernel abstracts all hardware into a consistent virtual interface. In addition, Linux supports **multi-tasking** in a manner that is transparent to user processes: each process can act as though it is the only process on the computer, with exclusive use of main memory and other hardware resources. The kernel actually runs several processes concurrently, and is responsible for mediating access to hardware resources so that each process has fair access while inter-process security is maintained.

1.3 Overview of the Kernel Structure

The Linux kernel is composed of five main subsystems:

- The **Process Scheduler** (SCHED) is responsible for controlling process access to the CPU. The scheduler enforces a policy that ensures that processes will have fair access to the CPU, while ensuring that necessary hardware actions are performed by the kernel on time
- The **Memory Manager** (MM) permits multiple process to securely share the machine's main memory system. In addition, the memory manager supports virtual memory that allows Linux to support processes that use more memory than is available in the system. Unused memory is swapped out to persistent storage using the file system then swapped back in when it is needed.
- The **Virtual File System** (VFS) abstracts the details of the variety of hardware devices by presenting a common file interface to all devices. In addition, the VFS supports several file system formats that are compatible with other operating systems.
- The **Network Interface** (NET) provides access to several networking standards and a variety of network hardware.
- The **Inter-Process Communication** (IPC) subsystem supports several mechanisms for process-to-process communication on a single Linux system.

The figure *Kernel Subsystem Overview* shows a high-level decomposition of the Linux kernel, where lines are drawn from dependent subsystems to the subsystems they depend on.

This diagram emphasizes that the most central subsystem is the **process scheduler**: all other subsystems depend on the process scheduler since all subsystems need to suspend and resume processes. Usually a subsystem will suspend a process that is waiting for a hardware operation to complete, and resume the process when the operation is finished. For example, when a process attempts to send a message across the network, the network interface may need to suspend the process until the hardware has completed sending the message successfully. After the message has been sent (or the hardware returns a failure), the network interface then resumes the process with a return code indicating the success or failure of the operation. The other subsystems (memory manager, virtual file system, and inter-process communication) all depend on the process scheduler for similar reasons.

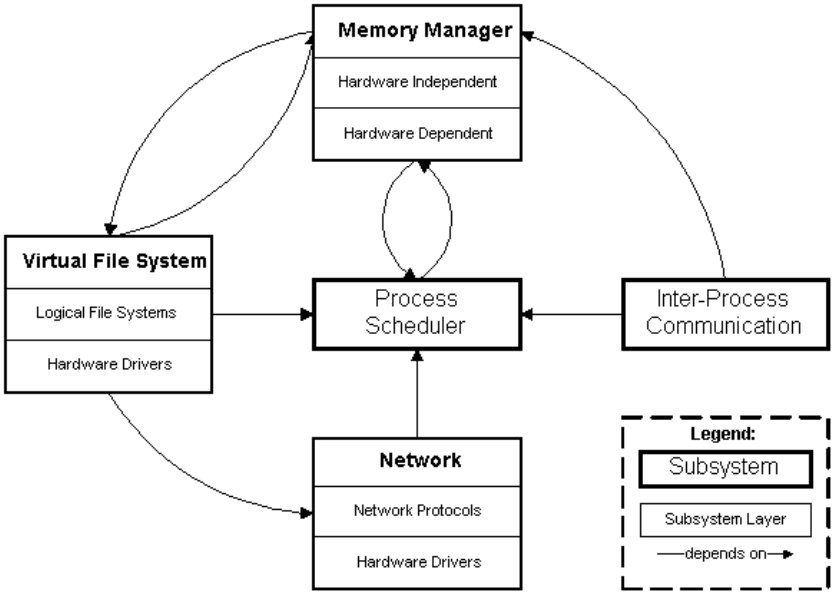

Figure 1: Kernel Subsystem Overview

The other dependencies are somewhat less obvious, but equally important:

- The process-scheduler subsystem uses the memory manager to adjust the hardware memory map for a specific process when that process is resumed.
- The inter-process communication subsystem depends on the memory manager to support a shared-memory communication mechanism. This mechanism allows two processes to access an area of common memory in addition to their usual private memory.
- The virtual file system uses the network interface to support a network file system (*NFS*), and also uses the memory manager to provide a *ramdisk* device.
- The memory manager uses the virtual file system to support *swapping*; this is the only reason that the memory manager depends on the process scheduler. When a process accesses memory that is currently swapped out, the memory manager makes a request to the file system to fetch the memory from persistent storage, and suspends the process.

2 Scheduling Concepts used by Linux

2.1 Scheduling code - Overview

It is the **scheduler** that must select the most deserving process to run out of all of the runnable processes in the system. A runnable process is one which is waiting only for a CPU to run on. Linux uses a reasonably **simple priority based scheduling algorithm** to choose between the current processes in the system. When it has chosen a new process to run it saves the state of the current process, the processor specific registers and other context being saved in the processes **task_struct** data structure.

2.1.1 Interlude: task_struct

The most important structure for the scheduling (and may be the whole system) is the **task_struct**. This structure represents the states of all tasks running in the systems. All executing processes have an entry in

the process table. The first entry in the process table is the special **init process**, which is the first process started at boot time.

There is a field that represents the process state, a field that indicates the processes **priority**, and a field which holds the number of clock ticks (**counter**) which the process can continue executing without forced rescheduling. It also contains the **schedule policy** (SCHED_OTHER, SCHED_FIFO, SCHED_RR) to determine how to schedule the process.

In order to keep track of all executing processes, a doubly linked list is maintained, (through two fields: **next_task** and **prev_task**). Since every process is related to some other process, there are fields which describe a processes: original parent, parent, youngest child, younger sibling, and finally older sibling.

There is a nested structure, **mm_struct**, which contains a process's memory management information, (such as start and end address of the code segment). This information is especially crucial when changing processes.

Process ID information is also kept within the task_struct. The process and group id are stored. File specific process data is located in a **fs_struct** substructure. Finally, there are fields that hold timing information; for example, the amount of time the process has spent in user mode and other information less crucial to scheduling.

From *include/linux/sched.h*

```
250 struct task_struct {
251 /* these are hardcoded - don't touch */
252         volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
253         unsigned long flags;    /* per process flags, defined below */
254         int sigpending;
255         mm_segment_t addr_limit;        /* thread address space:
256                                                 0-0xBFFFFFFF for user-thead
257                                                 0-0xFFFFFFFF for kernel-thread
258                                          */
259         struct exec_domain *exec_domain;
260         volatile long need_resched;
261
262         cycles_t avg_slice;
263         int lock_depth;         /* Lock depth. We can context switch in and out
                                       of holding a syscall kernel lock... */
264 /* begin intel cache line */
265         long counter;
266         long priority;
267         unsigned long policy;
268 /* memory management info */
269         struct mm_struct *mm, *active_mm;
270         int has_cpu;
271         int processor;
272         struct list_head run_list;
273         struct task_struct *next_task, *prev_task;
274         int last_processor;
275
276 /* task state */
277         struct linux_binfmt *binfmt;
278         int exit_code, exit_signal;
279         int pdeath_signal;  /*  The signal sent when the parent dies  */
280         /* ??? */
281         unsigned long personality;
282         int dumpable:1;
```

```
283         int did_exec:1;
284         pid_t pid;
285         pid_t pgrp;
286         pid_t tty_old_pgrp;
287         pid_t session;
288         /* boolean value for session group leader */
289         int leader;
290         /*
291          * pointers to (original) parent process, youngest child, younger sibling,
292          * older sibling, respectively.  (p->father can be replaced with
293          * p->p_pptr->pid)
294          */
295         struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
296
297         /* PID hash table linkage. */
298         struct task_struct *pidhash_next;
299         struct task_struct **pidhash_pprev;
300
301         wait_queue_head_t wait_chldexit;        /* for wait4() */
302         struct semaphore *vfork_sem;            /* for vfork() */
303         unsigned long rt_priority;
304         unsigned long it_real_value, it_prof_value, it_virt_value;
305         unsigned long it_real_incr, it_prof_incr, it_virt_incr;
306         struct timer_list real_timer;
307         struct tms times;
308         unsigned long start_time;
309         long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
310 /* mm fault and swap info: this can arguably be seen as either mm-specific
       or thread-specific */
311         unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
312         int swappable:1;
313         int hog:1;
314 /* process credentials */
315         uid_t uid,euid,suid,fsuid;
316         gid_t gid,egid,sgid,fsgid;
317         int ngroups;
318         gid_t   groups[NGROUPS];
319         kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
320         int keep_capabilities:1;
321         struct user_struct *user;

..

351 };
```

It is this information saved in the **task_struct** that is used by the scheduler to restore the state of the new process (this is processor specific) to run and then gives control of the system to that process. For the scheduler to fairly allocate CPU time between the runnable processes in the system it keeps information in the **task_struct** for each process:

Main variables:

policy

This is the scheduling policy that will be applied to this process. There are two types of Linux process, **normal** and **real time**. Real time processes have a higher priority than all of the other processes. If

there is a real time process ready to run, it will always run first. Real time processes may have two types of policy, **round robin** and **first in first out**. In **round robin** scheduling, each runnable real time process is run in turn and in **first in, first out** scheduling each runnable process is run in the order that it is in on the run queue and that order is never changed.

priority

This is the priority that the scheduler will give to this process. It is the value used for recalculation when all runnable processes have a **counter** value of 0. You can alter the priority of a process by means of system calls and the renice command (see *man nice* for details).

rt_priority

Linux supports real time processes and these are scheduled to have a higher priority than all of the other non-real time processes in system. This field allows the scheduler to give each real time process a relative priority. The priority of a real time processes can be altered using system calls.

counter

This is the amount of time (in jiffies) that this process is allowed to run for. It is set to **priority** when the process is first run and is decremented each clock tick.

The scheduler is run from several places within the kernel, especially to achieve good granularity on SMP-machines (in 2.3.99-pre6 **schedule()** is referenced in 239 files). It is run after putting the current process onto a wait queue and it may also be run at the end of a system call, just before a process is returned to process mode from system mode. One reason that it might need to run is because the system timer has just set the current processes counter to zero. Each time the scheduler is run it does the following:

Kernel work

The scheduler runs the bottom half handlers and processes the scheduler task queue.

Current process

The current process must be processed before another process can be selected to run.

If the scheduling policy of the current processes is **round robin** then it is put onto the back of the run queue.

If the task is INTERRUPTIBLE and it has received a signal since the last time it was scheduled then its state becomes RUNNING.

If the current process has timed out, then its state becomes RUNNING.

If the current process is RUNNING then it will remain in that state.

Processes that were neither RUNNING nor INTERRUPTIBLE are removed from the run queue. This means that they will not be considered for running when the scheduler looks for the most deserving process to run.

Process selection

The scheduler looks through the processes on the run queue looking for the most deserving process to run. If there are any real time processes (those with a real time scheduling policy) then those will get a higher weighting than ordinary processes. The weight for a normal process is its counter but for a **real time process** it is **counter plus 1000**. This means that if there are any runnable real time processes in the system then these will always be run before any normal runnable processes. The current process, which has consumed some of its time-slice (its counter has been decremented) is at a disadvantage if there are other processes with equal priority in the system; that is as it should be. If several processes have the same priority, the one nearest the front of the run queue is chosen. The

current process will get put onto the back of the run queue. In a balanced system with many processes of the same priority, each one will run in turn. This is known as **Round Robin** scheduling. However, as processes wait for resources, their run order tends to get moved around.

Swap processes

If the most deserving process to run is not the current process, then the current process must be suspended and the new one made to run. When a process is running it is using the registers and physical memory of the CPU and of the system. Each time it calls a routine it passes its arguments in registers and may stack saved values such as the address to return to in the calling routine. So, when the scheduler is running it is running in the context of the current process. It will be in a privileged mode, kernel mode, but it is still the current process that is running. When that process comes to be suspended, all of its machine state, including the program counter (PC) and all of the processor's registers, must be saved in the processes **task_struct** data structure. Then, all of the machine state for the new process must be loaded. This is a system dependent operation, no CPUs do this in quite the same way but there is usually some hardware assistance for this act.

This swapping of process context takes place at the end of the scheduler. The saved context for the previous process is, therefore, a snapshot of the hardware context of the system as it was for this process at the end of the scheduler. Equally, when the context of the new process is loaded, it too will be a snapshot of the way things were at the end of the scheduler, including this processes program counter and register contents.

If the previous process or the new current process uses virtual memory then the system's page table entries may need to be updated. Again, this action is architecture specific.

2.2 Pseudo - Code

- do kernel work
  - run bottom halfs
  - do soft IRQ's
- treat current process
  - if current process policy == ROUND_ROBIN: put process at the back of run queue.
  - if process id INTERRUPTIBLE and received a signal: current process state := RUNNING
  - if current process state == RUNNING: NOP
  - else remove process from run queue
- select process
  - calculate goodness
    - if process is a real time process: weight := counter + 1000
    - weight := weight + priority
  - select the process with the heighest weight
  - put the current process at the end of run queue
- swap process
  - if (previous process /= next process)
    - save context of previous process
    - load context of next process

3 Linux Scheduling - the Code

3.1 Goodness()

In order to select the process to run the function **goodness()** get's called. Its easy to see, that real time processes get a large priority and are always run before any other processes. Second, every process gets a goodness approximation according to the time it has left to run. As Linux uses a preemptive scheduling mechanism that gives every process a fixed time slice (set in *include/linux/sched.h* 200 ms time slices) the more time slices a process has left, the better its chances are to be selected.

PROC_CHANGE_PENALTY is a magic constant (set to 15), that tries to keep a running process on the current CPU. Evidently, this only makes sense on multi-processor machines running an SMP-Kernel (SMP stands for Synchronous Multii Processing). It's important to keep one process on the same processor, because switching one process from one processor to another will loose the benefit of L1/L2 cache provoquing cache misses, and thus slowing the system down.

From now on due to the highly complex matter, I will only consider scheduling on systems with one processor. (Thinking about concurrency gets me regularly stuck in a deadlock, and rebooting myself is a pain.)

mm_struct is a data structure that is used to describe the virtual memory of a task or process. So if two proccess have the same priority up to now and for one of them the MM (memory mapping) equals the current MM, then this one gets a higher priority.

From *kernel/sched.c*

```
 98 /*
 99  * This is the function that decides how desirable a process is..
100  * You can weigh different processes against each other depending
101  * on what CPU they've run on lately etc to try to handle cache
102  * and TLB miss penalties.
103  *
104  * Return values:
105  *       -1000: never select this
106  *           0: out of time, recalculate counters (but it might still be
107  *              selected)
108  *         +ve: "goodness" value (the larger, the better)
109  *       +1000: realtime process, select this.
110  */
111
112 static inline int goodness(struct task_struct * p, int this_cpu,
                               struct mm_struct *this_mm)
113 {
114         int weight;
115
116         /*
117          * Realtime process, select the first one on the
118          * runqueue (taking priorities within processes
119          * into account).
120          */
121         if (p->policy != SCHED_OTHER) {
122                 weight = 1000 + p->rt_priority;
123                 goto out;
124         }
125
126         /*
127          * Give the process a first-approximation goodness value
```

```
128          * according to the number of clock-ticks it has left.
129          *
130          * Don't do any other calculations if the time slice is
131          * over..
132          */
133         weight = p->counter;
134         if (!weight)
135                 goto out;
136
137 #ifdef CONFIG_SMP
138         /* Give a largish advantage to the same processor...   */
139         /* (this is equivalent to penalizing other processors) */
140         if (p->processor == this_cpu)
141                 weight += PROC_CHANGE_PENALTY;
142 #endif
143
144         /* .. and a slight advantage to the current MM */
145         if (p->mm == this_mm || !p->mm)
146                 weight += 1;
147         weight += p->priority;
148
149 out:
150         return weight;
151 }
```

3.2 Schedule()

Astonishingly, most of the scheduler code has not much to do with scheduling per se, but with dealing with interrupts, acquiring and releasing locks on important data structures and optionally dealing with SMP-Architectures.

All the code snippets in this section are from *kernel/sched.c*

```
438 asmlinkage void schedule(void)
439 {
440         struct schedule_data * sched_data;
441         struct task_struct *prev, *next, *p;
442         struct list_head *tmp;
443         int this_cpu, c;
444
445         if (!current->active_mm) BUG();
```

The following lines deal with the handling of 4.1 (bottom halfs), generated by interrupts. It's important to see (and it took me a long time), that the time accouting is done here. See 4 (time accounting) for more details.

```
446         if (tq_scheduler)
447                 goto handle_tq_scheduler;
448 tq_scheduler_back:
449
450         prev = current;
451         this_cpu = prev->processor;
452
```

If we are in an interrupt we must oops (crash) because an interrupt is not run in a process context and you cannot schedule away from servicing the interrupt.

```
453         if (in_interrupt())
454                 goto scheduling_in_interrupt;
455
456         release_kernel_lock(prev, this_cpu);
457
```

Handling softirqs:

```
458         /* Do "administrative" work here while we don't hold any locks */
459         if (softirq_state[this_cpu].active & softirq_state[this_cpu].mask)
460                 goto handle_softirq;
461 handle_softirq_back:
462
463         /*
464          * 'sched_data' is protected by the fact that we can run
465          * only one process per CPU.
466          */
467         sched_data = &aligned_data[this_cpu].schedule_data;
```

Acquiring a lock on the runqueue.

```
468
469         spin_lock_irq(&runqueue_lock);
470
```

If this is a real time process running on a round robin strategy, it will get moved to the end.

```
471         /* move an exhausted RR process to be last.. */
472         if (prev->policy == SCHED_RR)
473                 goto move_rr_last;
```

But the default behaviour is to delete it from the runqueue.

```
474 move_rr_back:
475
476         switch (prev->state & ~TASK_EXCLUSIVE) {
477                 case TASK_INTERRUPTIBLE:
478                         if (signal_pending(prev)) {
479                                 prev->state = TASK_RUNNING;
480                                 break;
481                         }
482                 default:
483                         del_from_runqueue(prev);
484                 case TASK_RUNNING:
485         }
486         prev->need_resched = 0;
487
```

Here the scheduling as described above is done:

```
488         /*
489          * this is the scheduler proper:
490          */
491
492 repeat_schedule:
493         /*
494          * Default process to select..
495          */
496         next = idle_task(this_cpu);
497         c = -1000;
498         if (prev->state == TASK_RUNNING)
499                 goto still_running;
500
501 still_running_back:
502         list_for_each(tmp, &runqueue_head) {
503                 p = list_entry(tmp, struct task_struct, run_list);
504                 if (can_schedule(p)) {
505                         int weight = goodness(p, this_cpu, prev->active_mm);
506                         if (weight > c)
507                                 c = weight, next = p;
508                 }
509         }
510
511         /* Do we need to re-calculate counters? */
512         if (!c)
513                 goto recalculate;
```

The algoritm has found the process with the highest priority. If it was lucky it's the same process already running, so not much has to be done.

```
514         /*
515          * from this point on nothing can prevent us from
516          * switching to the next task, save this fact in
517          * sched_data.
518          */
519         sched_data->curr = next;
520 #ifdef CONFIG_SMP
521         next->has_cpu = 1;
522         next->processor = this_cpu;
523 #endif
524         spin_unlock_irq(&runqueue_lock);
525
526         if (prev == next)
527                 goto same_process;
```

The next 33 lines are SMP and hardware specific and not within the scope of this paper (and definitely not within the scope of my knowledge), so I skip them.

Doing some statistics:

```
560         kstat.context_swtch++;
```

If we switch processes, the schedule algorithm must prepare the system to switch them. This is done with the function **prepare_to_switch()**. Switching is hardware-specific, on an Intel processor nothing happens

;-). (This took me some time to figure out: **prepare_to_switch()** is a macro that gets expanded to *do* { } *while(0)* which on the other hand gets optimised away by the compiler.)

```
561         /*
562          * there are 3 processes which are affected by a context switch:
563          *
564          * prev == .... ==> (last => next)
565          *
566          * It's the 'much more previous' 'prev' that is on next's stack,
567          * but prev is set to (the just run) 'last' process by switch_to().
568          * This might sound slightly confusing but makes tons of sense.
569          */
570         prepare_to_switch();
```

Now there has to be done some memory mapping, probably reloading some page tables and LDT's

```
571         {
572                 struct mm_struct *mm = next->mm;
573                 struct mm_struct *oldmm = prev->active_mm;
574                 if (!mm) {
575                         if (next->active_mm) BUG();
576                         next->active_mm = oldmm;
577                         atomic_inc(&oldmm->mm_count);
578                         enter_lazy_tlb(oldmm, next, this_cpu);
579                 } else {
580                         if (next->active_mm != mm) BUG();
581                         switch_mm(oldmm, mm, next, this_cpu);
582                 }
583
584                 if (!prev->mm) {
585                         prev->active_mm = NULL;
586                         mmdrop(oldmm);
587                 }
588         }
589
590         /*
591          * This just switches the register state and the
592          * stack.
593          */
```

Now the switching of the two processes occurs (**switch_to(prev, next, prev)**). This is again hardware specific, on an Intel machine this means saving the **Stack Pointer**, and the **Base Pointer** data and then reestablishing the state the new to run process was in the last time it was running, using the data saved in the **task_struct**.

```
594         switch_to(prev, next, prev);
```

The next line is SMP-specific, nothing happens on a one processor machine.

```
595         __schedule_tail(prev);
596
597 same_process:
698         reacquire_kernel_lock(current);
699         return;
600
```

The end of the algorithm, what follows are the labels of the goto jumps.

A recalulating most be done for all the processes. This is done by halfing the counter (*p->counter >> 1 == p->counter / 2*) and adding the processes priority. This formula takes into account the process's history and the process's priority. If a process is running often, its credits will exhaust rapidly, while processes that seldom run will not use up their credits this fast and thus get a better chance to run. This scheme has a tendency to prioritize processes, which deserve a rapid response time. (See 5 (S&G), page 716 for more details.)

```
601 recalculate:
602         {
603                 struct task_struct *p;
604                 spin_unlock_irq(&runqueue_lock);
605                 read_lock(&tasklist_lock);
606                 for_each_task(p)
607                         p->counter = (p->counter >> 1) + p->priority;
608                 read_unlock(&tasklist_lock);
609                 spin_lock_irq(&runqueue_lock);
610         }
611         goto repeat_schedule;
612
613 still_running:
614         c = prev_goodness(prev, this_cpu, prev->active_mm);
615         next = prev;
616         goto still_running_back;
617
618 handle_softirq:
619         do_softirq();
620         goto handle_softirq_back;
621
622 handle_tq_scheduler:
623         /*
624          * do not run the task queue with disabled interrupts,
625          * cli() wouldn't work on SMP
626          */
627         sti();
628         run_task_queue(&tq_scheduler);
629         goto tq_scheduler_back;
630
```

As described earlier, if this process is scheduled according to *round robin* policy we set its **counter** variable to its priority and move the current process to the end of the run_queue, and thus reducing its chances to run.

```
631 move_rr_last:
632         if (!prev->counter) {
633                 prev->counter = prev->priority;
634                 move_last_runqueue(prev);
635         }
636         goto move_rr_back;
637
```

We crash, provoqued by the BUG macro.

```
638 scheduling_in_interrupt:
```

```
639         printk("Scheduling in interrupt\n");
640         BUG();
641         return;
642 }
```

4 Time Accounting

In order to be able to implement the scheduling policies described above, we must keep track of how long a process has run to be able to do a fair selectioning between the processes waiting to be processed. And if a process has used up its credit to run, we must signal this to system so another process can be choosen to run.

In Linux this time accounting is done using bottom halfs, a concept unique to Linux(?).

4.1 What are bottom halfs?

There are often times in a kernel when you do not want to do work at this moment. A good example of this is during interrupt processing. When the interrupt was asserted, the processor stopped what it was doing and the operating system delivered the interrupt to the appropriate device driver. Device drivers should not spend too much time handling interrupts as, during this time, nothing else in the system can run. There is often some work that could just as well be done later on. Linux's bottom half handlers were invented so that device drivers and other parts of the Linux kernel could queue work to

Whenever a device driver, or some other part of the kernel, needs to schedule work to be done later, it adds work to the appropriate system queue, for example the timer queue, and then signals the kernel that some bottom half handling needs to be done. It does this by setting the appropriate bit in bh_active. Bit 8 is set if the driver has queued something on the immediate queue and wishes the immediate bottom half handler to run and process it. The bh_active bitmask is checked at the end of each system call, just before control is returned to the calling process. If it has any bits set, the bottom half handler routines that are active are called. Bit 0 is checked first, then 1 and so on until bit 31.

The bit in bh_active is cleared as each bottom half handling routine is called. bh_active is transient; it only has meaning between calls to the scheduler and is a way of not calling bottom half handling routines when there is no work for them to do.

4.2 Time Accounting using bottom halfs

Very early in the boot process when the system gets setup (paging, traps and IRQ get intialized) the scheduler too gets initialized (see *sched_ init()* in *init/main.c*). It's here where the infrastructure for time accouting is set up by setting a function pointer to the time accouting code which is run whenever the bottom halfs are processed, ergo every clock tick.

From *kernel/sched.c*:

```
1160 void __init sched_init(void)
1161 {

..

1174         init_bh(TIMER_BH, timer_bh);
1175         init_bh(TQUEUE_BH, tqueue_bh);
1176         init_bh(IMMEDIATE_BH, immediate_bh);
```

```
1177
1178         /*
1179          * The boot idle thread does lazy MMU switching as well:
1180          */
1181         atomic_inc(&init_mm.mm_count);
1182         enter_lazy_tlb(&init_mm, current, cpu);
```

update_timers is the interesting funciton call here. (*run_old_timers* and *immediate_bh* will trigger the timer task queue and the immediate task queue to be run.)

From *kernel/timer.c*:

```
664 void timer_bh(void)
665 {
666         update_times();
667         run_old_timers();
668         run_timer_list();
669 }
```

update_times() calls **update_process_times(ticks, system)** where the updating of the time left for a process is done. Therefore the **counter** variable gets decreased by **ticks**, a magical value (at least in my eyes) that describes the time past between the last call. It's important to see, that if the currently running process has used up its credit (**counter**<0) a flag is set (**need_resched**$=1$) that will force the scheduler to reschedule as soon as possible by selecting a process to run.

From *kernel/timer.c*:

```
563 static void update_process_times(unsigned long ticks, unsigned long system)
564 {
565 /*
566  * SMP does this on a per-CPU basis elsewhere
567  */
568 #ifndef  CONFIG_SMP
569         struct task_struct * p = current;
570         unsigned long user = ticks - system;
571         if (p->pid) {
572                 p->counter -= ticks;
573                 if (p->counter <= 0) {
574                         p->counter = 0;
575                         p->need_resched = 1;
576                 }
```

Doing some statistics, again.

```
577                 if (p->priority < DEF_PRIORITY)
578                         kstat.cpu_nice += user;
579                 else
580                         kstat.cpu_user += user;
581                 kstat.cpu_system += system;
582         }
583         update_one_process(p, ticks, user, system, 0);
584 #endif
585 }
```

5 Resources

- *The Linux Kernel* <http://sunsite.unc.edu/linux/LDP/tlk/tlk.html> by David A. Rusling.
- *Conceptual Architecture of the Linux Kernel* <http://plg.uwaterloo.ca/~itbowman/papers/CS746G-a1.html> by Ivan T. Bowman.
- *Concrete Architecture of the Linux Kernel* <http://plg.uwaterloo.ca/~itbowman/papers/CS746G-a2.html> by Ivan T. Bowman, Saheem Siddiqi, and Meyer C. Tanuan.
- *Linux as a Case Study: Its Extracted Software Architecture* <http://plg.uwaterloo.ca/~itbowman/papers/linuxcase.html> by Ivan T. Bowman, Richard C. Holt and Neil V. Brewster.
- *The source code, cross referenced using LXR* <http://lxr.linux.no>
- Using IRC #kernelnewbies at irc.openprojects.net. See <http://www.surriel.com/kernelnewbies.shtml> for more details.
- Silberschatz & Galvin: *Operating System Concepts*. Fifth Edition. John Wiley & Sons , Inc. (1999).

This text is also available as a *PDF document* <LinuxScheduling.pdf>.