Chapter 7

Scheduling and Kernel Synchronization

In this chapter

The Linux kernel is a multitasking kernel, which means that many processes can run as if they were the only process on the system. The way in which an operating system chooses which process at a given time has access to a system's CPU(s) is controlled by a scheduler.

The scheduler is responsible for swapping CPU access between different processes and for choosing the order in which processes obtain CPU access. Linux, like most operating systems, triggers the scheduler by using a timer interrupt. When this timer goes off, the kernel needs to decide whether to yield the CPU to a process different than the current process and, if a yield occurs, which process gets the CPU next. The amount of time between the timer interrupt is called a **timeslice**.

System processes tend to fall into two types: interactive and non-interactive. Interactive processes are heavily dependent upon I/O and, as a result, do not usually use their entire timeslice and, instead, yield the CPU to another process. Non-interactive processes are heavily dependent on the CPU and typically use most, if not all, of their timeslice. The scheduler has to balance the requirements of these two types of processes and attempt to ensure every process gets enough time to accomplish its task without detrimentally affecting the execution of other processes.

Linux, like some schedulers, distinguishes between one more type of process: a real-time process. Real-time processes must execute in real time. Linux has support for real-time processes, but those exist outside of the scheduler logic. Put simply, the Linux scheduler treats any process marked as real-time as a higher priority than any other process. It is up to the developer of the real-time processes to ensure that these processes do not hog the CPU and eventually yield.

Schedulers typically use some type of process queue to manage the execution of processes on the system. In Linux, this process queue is called the run queue. The run queue is described fully in Chapter 3, "Processes: The Principal Model of Execution,"[1] but let's recap some of the fundamentals here because of the close tie between the scheduler and the run queue.

---

[1] Section 3.6 discusses the run queue.

In Linux, the run queue is composed of two priority arrays:

- **Active**. Stores processes that have not yet used up their timeslice
- **Expired**. Stores processes that have used up their timeslice

From a high level, the scheduler's job in Linux is to take the highest priority active processes, let them use the CPU to execute, and place them in the expired array when they use up their timeslice. With this high-level framework in mind, let's closely look at how the Linux scheduler operates.

7.1 Linux Scheduler

The 2.6 Linux kernel introduces a completely new scheduler that's commonly referred to as the O(1) scheduler. The scheduler can perform the scheduling of a task in constant time.[2] Chapter 3 addressed the basic structure of the scheduler and how a newly created process is initialized for it. This section describes how a task is executed on a single CPU system. There are some mentions of code for scheduling across multiple CPU (SMP) systems but, in general, the same scheduling process applies across CPUs. We then describe how the scheduler switches out the currently running process, performing what is called a context switch, and then we touch on the other significant change in the 2.6 kernel: preemption.

From a high level, the scheduler is simply a grouping of functions that operate on given data structures. Nearly all the code implementing the scheduler can be found in `kernel/sched.c` and `include/linux/sched.h`. One important point to mention early on is how the scheduler code uses the terms "task" and "process" interchangeably. Occasionally, code comments also use "thread" to refer to a task or process. A task, or process, in the scheduler is a collection of data structures and flow of control. The scheduler code also refers to a `task_struct`, which is a data structure the Linux kernel uses to keep track of processes.[3]

---

[2] O(1) is big-oh notation, which means constant time.

[3] Chapter 3 explains the `task_struct` structure in depth.

7.1.1 Choosing the Next Task

After a process has been initialized and placed on a run queue, at some time, it should have access to the CPU to execute. The two functions that are responsible for passing CPU control to different processes are `schedule()` and `scheduler_tick()`. `scheduler_tick()` is a system timer that the kernel periodically calls and marks processes as needing rescheduling. When a timer event occurs, the current process is put on hold and the Linux kernel itself takes control of the CPU. When the timer event finishes, the Linux kernel normally passes control back to the process that was put on hold. However, when the held process has been marked as needing rescheduling, the kernel calls `schedule()` to choose which process to activate instead of the process that was executing before the kernel took control. The process that was executing before the kernel took control is called the current process. To make things slightly more complicated, in certain situations, the kernel can take control from the kernel; this is called kernel preemption. In the following sections, we assume that the scheduler decides which of two user space processes gains CPU control.

Figure 7.1 illustrates how the CPU is passed among different processes as time progresses. We see that *Process A* has control of the CPU and is executing. The system timer `scheduler_tick()` goes off, takes control of the CPU from *A*, and marks *A* as needing rescheduling. The Linux kernel calls `schedule()`, which chooses *Process B* and the control of the CPU is given to *B*.

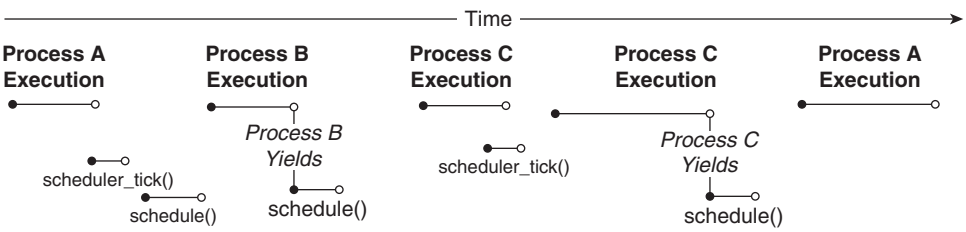

FIGURE 7.1
Scheduling Processes

Process B executes for a while and then voluntarily yields the CPU. This commonly occurs when a process waits on some resource. *B* calls `schedule()`, which chooses *Process C* to execute next.

Process C executes until `scheduler_tick()` occurs, which does not mark *C* as needing rescheduling. This results in `schedule()` not being called and *C* regains control of the CPU.

Process C yields by calling `schedule()`, which determines that *Process A* should gain control of the CPU and *A* starts to execute again.

We first examine `schedule()`, which is how the Linux kernel decides which process to execute next, and then we examine `scheduler_tick()`, which is how the kernel determines which processes need to yield the CPU. The combined effects of these functions demonstrate the flow of control within the scheduler:

```
----------------------------------------------------------------------
kernel/sched.c
2184 asmlinkage void schedule(void)
2185 {
2186   long *switch_count;
2187   task_t *prev, *next;
2188   runqueue_t *rq;
2189   prio_array_t *array;
2190   struct list_head *queue;
2191   unsigned long long now;
2192   unsigned long run_time;
2193   int idx;
2194
2195   /*
2196   * Test if we are atomic. Since do_exit() needs to call into
2197   * schedule() atomically, we ignore that path for now.
2198   * Otherwise, whine if we are scheduling when we should not be.
2199   */
2200   if (likely(!(current->state & (TASK_DEAD | TASK_ZOMBIE)))) {
2201     if (unlikely(in_atomic())) {
2202       printk(KERN_ERR "bad: scheduling while atomic!\n  ");
2203       dump_stack();
2204     }
2205   }
2206
2207 need_resched:
2208   preempt_disable();
2209   prev = current;
2210   rq = this_rq();
2211
2212   release_kernel_lock(prev);
2213   now = sched_clock();
2214   if (likely(now - prev->timestamp < NS_MAX_SLEEP_AVG))
2215     run_time = now - prev->timestamp;
2216   else
2217     run_time = NS_MAX_SLEEP_AVG;
2218
2219   /*
```

```
2220    * Tasks with interactive credits get charged less run_time
2221    * at high sleep_avg to delay them losing their interactive
2222    * status
2223    */
2224    if (HIGH_CREDIT(prev))
2225    run_time /= (CURRENT_BONUS(prev) ? : 1);
```

Lines 2213–2218

We calculate the length of time for which the process on the scheduler has been active. If the process has been active for longer than the average maximum sleep time (`NS_MAX_SLEEP_AVG`), we set its runtime to the average maximum sleep time.

This is what the Linux kernel code calls a timeslice in other sections of the code. A **timeslice** refers to both the amount of time between scheduler interrupts and the length of time a process has spent using the CPU. If a process exhausts its timeslice, the process expires and is no longer active. The **timestamp** is an absolute value that determines for how long a process has used the CPU. The scheduler uses timestamps to decrement the timeslice of processes that have been using the CPU.

For example, suppose Process A has a timeslice of 50 clock cycles. It uses the CPU for 5 clock cycles and then yields the CPU to another process. The kernel uses the timestamp to determine that Process A has 45 cycles left on its timeslice.

Lines 2224–2225

Interactive processes are processes that spend much of their time waiting for input. A good example of an interactive process is the keyboard controller—most of the time the controller is waiting for input, but when it has a task to do, the user expects it to occur at a high priority.

Interactive processes, those that have an interactive credit of more than 100 (default value), get their effective `run_time` divided by (`sleep_avg/max_sleep_avg * MAX_BONUS(10)`):[4]

```
kernel/sched.c
2226
2227    spin_lock_irq(&rq->lock);
2228
2229    /*
2230    * if entering off of a kernel preemption go straight
2231    * to picking the next task.
2232    */
```

[4] Bonuses are scheduling modifiers for high priority.

```
2233   switch_count = &prev->nivcsw;
2234   if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
2235     switch_count = &prev->nvcsw;
2236     if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
2237         unlikely(signal_pending(prev))))
2238       prev->state = TASK_RUNNING;
2239     else
2240       deactivate_task(prev, rq);
2241   }
-------------------------------------------------------------------------
```

Line 2227

The function obtains the run queue lock because we're going to modify it.

Lines 2233–2241

If we have entered `schedule()` with the previous process being a kernel preemption, we leave the previous process running if a signal is pending. This means that the kernel has preempted normal processing in quick succession; thus, the code is contained in two `unlikely()` statements.[5] If there is no further preemption, we remove the preempted process from the run queue and continue to choose the next process to run.

```
-------------------------------------------------------------------------
kernel/sched.c
2243   cpu = smp_processor_id();
2244   if (unlikely(!rq->nr_running)) {
2245     idle_balance(cpu, rq);
2246     if (!rq->nr_running) {
2247       next = rq->idle;
2248       rq->expired_timestamp = 0;
2249       wake_sleeping_dependent(cpu, rq);
2250       goto switch_tasks;
2251     }
2252   }
2253
2254   array = rq->active;
2255   if (unlikely(!array->nr_active)) {
2256     /*
2257     * Switch the active and expired arrays.
2258     */
2259     rq->active = rq->expired;
2260     rq->expired = array;
2261     array = rq->active;
2262     rq->expired_timestamp = 0;
2263     rq->best_expired_prio = MAX_PRIO;
2264   }
-------------------------------------------------------------------------
```

[5] For more information on the unlikely routine, see Chapter 2, "Exploration Toolkit."

Line 2243

We grab the current CPU identifier via `smp_processor_id()`.

Lines 2244–2252

If the run queue has no processes on it, we set the next process to the idle process and reset the run queue's expired timestamp to 0. On a multiprocessor system, we first check if any processes are running on other CPUs that this CPU can take. In effect, we load balance idle processes across all CPUs in the system. Only if no processes can be moved from the other CPUs do we set the run queue's next process to idle and reset the expired timestamp.

Lines 2255–2264

If the run queue's active array is empty, we switch the active and expired array pointers before choosing a new process to run.

```
-----------------------------------------------------------------------
kernel/sched.c
2266    idx = sched_find_first_bit(array->bitmap);
2267    queue = array->queue + idx;
2268    next = list_entry(queue->next, task_t, run_list);
2269
2270    if (dependent_sleeper(cpu, rq, next)) {
2271      next = rq->idle;
2272      goto switch_tasks;
2273    }
2274
2275    if (!rt_task(next) && next->activated > 0) {
2276      unsigned long long delta = now - next->timestamp;
2277
2278      if (next->activated == 1)
2279        delta = delta * (ON_RUNQUEUE_WEIGHT * 128 / 100) / 128;
2280
2281      array = next->array;
2282      dequeue_task(next, array);
2283      recalc_task_prio(next, next->timestamp + delta);
2284      enqueue_task(next, array);
2285    }
next->activated = 0;
-----------------------------------------------------------------------
```

Lines 2266–2268

The scheduler finds the highest priority process to run via `sched_find_first_bit()` and then sets up `queue` to point to the list held in the

priority array at the specified location. `next` is initialized to the first process in `queue`.

Lines 2270–2273

If the process to be activated is dependent on a sibling that is sleeping, we choose a new process to be activated and jump to `switch_tasks` to continue the scheduling function.

Suppose that we have Process A that spawned Process B to read from a device and that Process A was waiting for Process B to finish before continuing. If the scheduler chooses Process A for activation, this section of code, `dependent_sleeper()`, determines that Process A is waiting on Process B and chooses an entirely new process to activate.

Lines 2275–2285

If the process' activated attribute is greater than 0, and the next process is not a real-time task, we remove it from `queue`, recalculate its priority, and enqueue it again.

Line 2286

We set the process' activated attribute to 0, and then run with it.

```
---------------------------------------------------------------------
kernel/sched.c
2287 switch_tasks:
2288   prefetch(next);
2289   clear_tsk_need_resched(prev);
2290   RCU_qsctr(task_cpu(prev))++;
2291
2292   prev->sleep_avg -= run_time;
2293   if ((long)prev->sleep_avg <= 0) {
2294     prev->sleep_avg = 0;
2295     if (!(HIGH_CREDIT(prev) || LOW_CREDIT(prev)))
2296       prev->interactive_credit--;
2297   }
2298   prev->timestamp = now;
2299
2300   if (likely(prev != next)) {
2301     next->timestamp = now;
2302     rq->nr_switches++;
2303     rq->curr = next;
2304     ++*switch_count;
2305
```

```
2306     prepare_arch_switch(rq, next);
2307     prev = context_switch(rq, prev, next);
2308     barrier();
2309
2310     finish_task_switch(prev);
2311   } else
2312     spin_unlock_irq(&rq->lock);
2313
2314   reacquire_kernel_lock(current);
2315   preempt_enable_no_resched();
2316   if (test_thread_flag(TIF_NEED_RESCHED))
2317     goto need_resched;
2318 }
-----------------------------------------------------------------------
```

Line 2288

We attempt to get the memory of the new process' task structure into the CPU's L1 cache. (See `include/linux/prefetch.h` for more information.)

Line 2290

Because we're going through a context switch, we need to inform the current CPU that we're doing so. This allows a multi-CPU device to ensure data that is shared across CPUs is accessed exclusively. This process is called read-copy updating. For more information, see `http://lse.sourceforge.net/locking/rcupdate.html`.

Lines 2292–2298

We decrement the previous process' `sleep_avg` attribute by the amount of time it ran, adjusting for negative values. If the process is neither interactive nor non-interactive, its interactive credit is between high and low, so we decrement its interactive credit because it had a low sleep average. We update its timestamp to the current time. This operation helps the scheduler keep track of how much time a given process has spent using the CPU and estimate how much time it will use the CPU in the future.

Lines 2300–2304

If we haven't chosen the same process, we set the new process' timestamp, increment the run queue counters, and set the current process to the new process.

Lines 2306–2308

These lines describe the assembly language `context_switch()`. Hold on for a few paragraphs as we delve into the explanation of context switching in the next section.

Lines 2314–2318

We reacquire the kernel lock, enable preemption, and see if we need to reschedule immediately; if so, we go back to the top of `schedule()`.

It's possible that after we perform the `context_switch()`, we need to reschedule. Perhaps `scheduler_tick()` has marked the new process as needing rescheduling or, when we enable preemption, it gets marked. We keep rescheduling processes (and context switching them) until one is found that doesn't need rescheduling. The process that leaves `schedule()` becomes the new process executing on this CPU.

7.1.2 Context Switch

Called from `schedule()` in `/kernel/sched.c`, `context_switch()` does the machine-specific work of switching the memory environment and the processor state. In the abstract, `context_switch` swaps the current task with the next task. The function `context_switch()` begins executing the next task and returns a pointer to the task structure of the task that was running before the call:

```
----------------------------------------------------------------------
kernel/sched.c
1048 /*
1049 * context_switch - switch to the new MM and the new
1050 * thread's register state.
1051 */
1052 static inline
1053 task_t * context_switch(runqueue_t *rq, task_t *prev, task_t *next)
1054 {
1055   struct mm_struct *mm = next->mm;
1056   struct mm_struct *oldmm = prev->active_mm;
...
1063     switch_mm(oldmm, mm, next);
...
1072   switch_to(prev, next, prev);
1073
1074   return prev;
1075 }
----------------------------------------------------------------------
```

Here, we describe the two jobs of `context_switch`: one to switch the virtual memory mapping and one to switch the task/thread structure. The first job, which the function `switch_mm()` carries out, uses many of the hardware-dependent memory management structures and registers:

```
-----------------------------------------------------------------------
/include/asm-i386/mmu_context.h
026  static inline void switch_mm(struct mm_struct *prev,
027     struct mm_struct *next,
028     struct task_struct *tsk)
029  {
030   int cpu = smp_processor_id();
031
032   if (likely(prev != next)) {
033    /* stop flush ipis for the previous mm */
034    cpu_clear(cpu, prev->cpu_vm_mask);
035  #ifdef CONFIG_SMP
036    cpu_tlbstate[cpu].state = TLBSTATE_OK;
037    cpu_tlbstate[cpu].active_mm = next;
038  #endif
039    cpu_set(cpu, next->cpu_vm_mask);
040
041    /* Re-load page tables */
042    load_cr3(next->pgd);
043
044    /*
045    * load the LDT, if the LDT is different:
046    */
047   if (unlikely(prev->context.ldt != next->context.ldt))
048     load_LDT_nolock(&next->context, cpu);
049   }
050  #ifdef CONFIG_SMP
051   else {
-----------------------------------------------------------------------
```

Line 39

Bind the new task to the current processor.

Line 42

The code for switching the memory context utilizes the x86 hardware register `cr3`, which holds the base address of all paging operations for a given process. The new page global descriptor is loaded here from `next->pgd`.

Line 47

Most processes share the same LDT. If another LDT is required by this process, it is loaded here from the new `next->context` structure.

The other half of function `context_switch()` in `/kernel/sched.c` then calls the macro `switch_to()`, which calls the C function `__switch_to()`. The delineation of architecture *in*dependence to architecture dependence for both x86 and PPC is the `switch_to()` macro.

7.1.2.1 Following the x86 Trail of switch_to()

The x86 code is more compact than PPC. The following is the architecture-dependent code for `__switch_to()`. `task_struct` (not `thread_struct`) is passed to `__switch_to()`. The code discussed next is inline assembler code for calling the C function `__switch_to()` (line 23) with the proper `task_struct` structures as parameters.

The `context_switch` takes three task pointers: `prev`, `next`, and `last`. In addition, there is the current pointer.

Let us now explain, at a high level, what occurs when `switch_to()` is called and how the task pointers change after a call to `switch_to()`.

Figure 7.2 shows three `switch_to()` calls using three processes: A, B, and C.

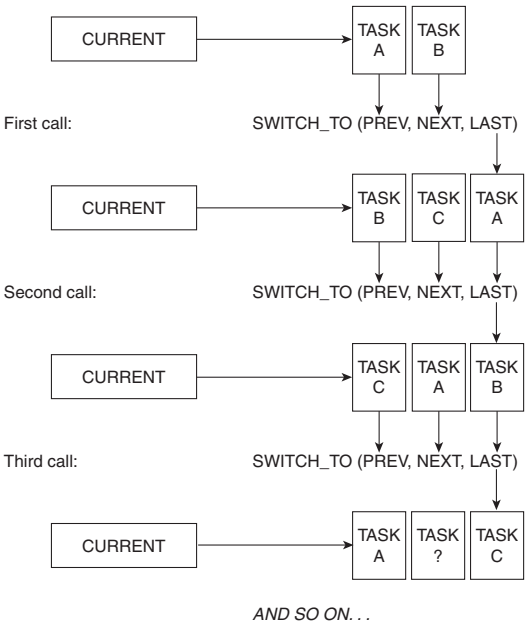

FIGURE 7.2
switch_to Calls

We want to switch A and B. Before, the **first call** we have

- Current → A
- Prev → A, next → B

After the **first call**:

- Current → B
- Last → A

Now, we want to switch B and C. Before the **second call**, we have

- Current → B
- Prev → B, next → C

After the **second call**:

- Current → C
- Last → B

Returning from the **second call**, current now points to task (C) and last points to (B).

The method continues with task (A) being swapped in once again, and so on.

The inline assembly of the `switch_to()` function is an excellent example of assembly magic in the kernel. It is also a good example of the `gcc` C extensions. See Chapter 2, "Exploration Toolkit," for a tutorial featuring this function. Now, we carefully walk through this code block.

```
----------------------------------------------------------------------------
/include/asm-i386/system.h
012  extern struct task_struct * FASTCALL(__switch_to(struct task_struct *prev,
struct task_struct *next));

015  #define switch_to(prev,next,last) do {     \
016   unsigned long esi,edi;       \
017   asm volatile("pushfl\n\t"       \
018   "pushl %%ebp\n\t"        \
019   "movl %%esp,%0\n\t"  /* save ESP */    \
020   "movl %5,%%esp\n\t"  /* restore ESP */    \
021   "movl $1f,%1\n\t"   /* save EIP */   \
022   "pushl %6\n\t"   /* restore EIP */   \
023   "jmp __switch_to\n"        \
```

```
023   "1:\t"          \
024   "popl %%ebp\n\t"        \
025   "popfl"         \
026   :"=m" (prev->thread.esp),"=m" (prev->thread.eip),  \
027   "=a" (last),"=S" (esi),"=D" (edi)     \
028   :"m" (next->thread.esp),"m" (next->thread.eip),   \
029   "2" (prev), "d" (next));       \
030  } while (0)
------------------------------------------------------------------------
```

Line 12

The `FASTCALL` macro resolves to `__attribute__ regparm(3)`, which forces the parameters to be passed in registers rather than stack.

Lines 15–16

The `do {} while (0)` construct allows (among other things) the macro to have local the variables `esi` and `edi`. Remember, these are just local variables with familiar names.

Current and the Task Structure

As we explore the kernel, whenever we need to retrieve or store information on the task (or process) which is *currently* running on a given processor, we use the global variable `current` to reference its task structure. For example, `current->pid` holds the process ID. Linux allows for a quick (and clever) method of referencing the current task structure.

Every process is assigned 8K of contiguous memory when it is created. (With Linux 2.6, there is a compile-time option to use 4K instead of 8K.) This 8K segment is occupied by the task structure and the kernel stack for the given process. Upon process creation, Linux puts the task structure at the low end of the 8K memory and the kernel stack pointer starts at the high end. The kernel stack pointer (especially for x86 and r1 for PPC) decrements as data is pushed onto the stack. Because this 8K memory region is page-aligned, its starting address (in hex notation) always ends in 0x000 (multiples of 4k bytes).

As you might have guessed, the clever method by which Linux references the current task structure is to `AND` the contents of the stack pointer with 0xffff_f000. Recent versions of the PPC Linux kernel have taken this one step further by dedicating General Purpose Register 2 to holding the current pointer.

Lines 17 and 30

The construct `asm volatile ()`[6] encloses the inline assembly block and the volatile keyword assures that the compiler will not change (optimize) the routine in any way.

Lines 17–18

Push the `flags` and `ebp` registers onto the stack. (Note: We are still using the stack associated with the `prev` task.)

Line 19

This line saves the current stack pointer `esp` to the `prev` task structure.

Line 20

Move the stack pointer from the next task structure to the current processor `esp`.

> NOTE By definition, we have just made a context switch.

We are now with a new kernel stack and thus, any reference to current is to the new (`next`) task structure.

Line 21

Save the return address for `prev` into its task structure. This is where the `prev` task resumes when it is restarted.

Line 22

Push the return address (from when we return from `__switch_to()`) onto the stack. This is the `eip` from `next`. The `eip` was saved into its task structure (on line 21) when it was stopped, or preempted the last time.

Line 23

Jump to the C function `__switch_to()` to update the following:

- The next thread structure with the kernel stack pointer
- Thread local storage descriptor for this processor
- `fs` and `gs` for `prev` and `next`, if needed

---

[6] See Chapter 2 for more information on `volatile`.

- Debug registers, if needed
- I/O bitmaps, if needed

`__switch_to()` then returns the updated `prev` task structure.

Lines 24–25

Pop the base pointer and flags registers from the new (next task) kernel stack.

Lines 26–29

These are the output and input parameters to the inline assembly routine. See the "Inline Assembly" section in Chapter 2 for more information on the *constraints* put on these parameters.

Line 29

By way of assembler magic, `prev` is returned in `eax`, which is the third positional parameter. In other words, the input parameter `prev` is passed out of the `switch_to()` macro as the output parameter last.

Because `switch_to()` is a macro, it was executed inline with the code that called it in `context_switch()`. It does not return as functions normally do.

For the sake of clarity, remember that `switch_to()` passes back `prev` in the `eax` register, execution then continues in `context_switch()`, where the next instruction is `return prev` (line 1074 of `kernel/sched.c`). This allows `context_switch()` to pass back a pointer to the last task running.

7.1.2.2 Following the PPC context_switch()

The PPC code for `context_switch()` has slightly more work to do for the same results. Unlike the `cr3` register in x86 architecture, the PPC uses hash functions to point to context environments. The following code for `switch_mm()` touches on these functions, but Chapter 4, "Memory Management," offers a deeper discussion.

Here is the routine for `switch_mm()` which, in turn, calls the routine `set_context()`.

```
----------------------------------------------------------------------
/include/asm-ppc/mmu_context.h
155  static inline void switch_mm(struct mm_struct *prev, struct
mm_struct *next,struct task_struct *tsk)
156  {
157   tsk->thread.pgdir = next->pgd;
```

```
158   get_mmu_context(next);
159   set_context(next->context, next->pgd);
160  }
------------------------------------------------------------------------
```

Line 157

The page global directory (segment register) for the new thread is made to point to the `next->pgd` pointer.

Line 158

The `context` field of the `mm_struct` (`next->context`) passed into `switch_mm()` is updated to the value of the appropriate context. This information comes from a global reference to the variable `context_map[]`, which contains a series of bitmap fields.

Line 159

This is the call to the assembly routine `set_context`. Below is the code and discussion of this routine. Upon execution of the `blr` instruction on line 1468, the code returns to the `switch_mm` routine.

```
------------------------------------------------------------------------
/arch/ppc/kernel/head.S
1437  _GLOBAL(set_context)
1438  mulli  r3,r3,897  /* multiply context by skew factor */
1439  rlwinm  r3,r3,4,8,27  /* VSID = (context & 0xfffff) << 4 */
1440  addis  r3,r3,0x6000  /* Set Ks, Ku bits */
1441  li  r0,NUM_USER_SEGMENTS
1442  mtctr  r0
...
1457  3:  isync
...
1461  mtsrin  r3,r4
1462  addi  r3,r3,0x111  /* next VSID */
1463  rlwinm  r3,r3,0,8,3  /* clear out any overflow from VSID field */
1464  addis  r4,r4,0x1000  /* address of next segment */
1465  bdnz  3b
1466  sync
1467  isync
1468  blr
------------------------------------------------------------------------
```

Lines 1437–1440

The `context` field of the `mm_struct` (`next->context`) passed into `set_context()` by way of `r3`, sets up the hash function for PPC segmentation.

Lines 1461–1465

The pgd field of the mm_struct (next->pgd) passed into set_context() by way of r4, points to the segment registers.

Segmentation is the basis of PPC memory management (refer to Chapter 4). Upon returning from set_context(), the mm_struct next is initialized to the proper memory regions and is returned to switch_mm().

7.1.2.3 Following the PPC Trail of switch_to()

The result of the PPC implementation of switch_to() is necessarily identical to the x86 call; it takes in the current and next task pointers and returns a pointer to the previously running task:

```
-----------------------------------------------------------------------
include/asm-ppc/system.h
88 extern struct task_struct *__switch_to(struct task_struct *,
89   struct task_struct *);
90 #define switch_to(prev, next, last)
((last) = __switch_to((prev), (next)))
91
92 struct thread_struct;
93 extern struct task_struct *_switch(struct thread_struct *prev,
94         struct thread_struct *next);
-----------------------------------------------------------------------
```

On line 88, __switch_to() takes its parameters as task_struct type and, at line 93, _switch() takes its parameters as *thread*_struct. This is because the thread entry within task_struct contains the architecture-dependent processor register information of interest for the given thread. Now, let us examine the implementation of __switch_to():

```
-----------------------------------------------------------------------
/arch/ppc/kernel/process.c
200  struct task_struct *__switch_to(struct task_struct *prev,
   struct task_struct *new)
201  {
202   struct thread_struct *new_thread, *old_thread;
203   unsigned long s;
204   struct task_struct *last;
205   local_irq_save(s);
...
247   new_thread = &new->thread;
248   old_thread = &current->thread;
249   last = _switch(old_thread, new_thread);
250   local_irq_restore(s);
251   return last;
252  }
-----------------------------------------------------------------------
```

Line 205

Disable interrupts before the context switch.

Lines 247–248

Still running under the context of the *old* thread, pass the pointers to the thread structure to the `_switch()` function.

Line 249

`_switch()` is the assembly routine called to do the work of switching the two thread structures (see the following section).

Line 250

Enable interrupts after the context switch.

To better understand what needs to be swapped within a PPC thread, we need to examine the `thread_struct` passed in on line 249.

Recall from the exploration of the x86 context switch that the switch does not officially occur until we are pointing to a new kernel stack. This happens in `_switch()`.

Tracing the PPC Code for _switch()

By convention, the parameters of a PPC C function (from left to right) are held in `r3`, `r4`, `r5`, …`r12`. Upon entry into `switch()`, `r3` points to the `thread_struct` for the current task and `r4` points to the `thread_struct` for the new task:

```
-----------------------------------------------------------------------
/arch/ppc/kernel/entry.S
437  _GLOBAL(_switch)
438   stwu  r1,-INT_FRAME_SIZE(r1)
439   mflr  r0
440   stw  r0,INT_FRAME_SIZE+4(r1)
441   /* r3-r12 are caller saved -- Cort */
442   SAVE_NVGPRS(r1)
443   stw  r0,_NIP(r1)  /* Return to switch caller */
444   mfmsr  r11
...
458  1:  stw  r11,_MSR(r1)
459   mfcr  r10
460   stw  r10,_CCR(r1)
461   stw  r1,KSP(r3)  /* Set old stack pointer */
462
463   tophys(r0,r4)
```

```
464   CLR_TOP32(r0)
465   mtspr  SPRG3,r0/* Update current THREAD phys addr */
466   lwz  r1,KSP(r4)  /* Load new stack pointer */
467   /* save the old current 'last' for return value */
468   mr  r3,r2
469   addi  r2,r4,-THREAD  /* Update current */
...
478   lwz  r0,_CCR(r1)
479   mtcrf  0xFF,r0
480   REST_NVGPRS(r1)
481
482   lwz  r4,_NIP(r1)  /* Return to _switch caller in new task */
483   mtlr  r4
484   addi  r1,r1,INT_FRAME_SIZE
485   blr
--------------------------------------------------------------------------
```

The byte-for-byte mechanics of swapping out the previous `thread_struct` for the new is left as an exercise for you. It is worth noting, however, the use of `r1`, `r2`, `r3`, `SPRG3`, and `r4` in `_switch()` to see the basics of this operation.

Lines 438–460

The environment is saved to the current stack with respect to the current stack pointer, `r1`.

Line 461

The entire environment is then saved into the current `thread_struct` pointer passed in by way of `r3`.

Lines 463–465

`SPRG3` is updated to point to the thread structure of the new task.

Line 466

`KSP` is the offset into the task structure (`r4`) of the new task's kernel stack pointer. The stack pointer `r1` is now updated with this value. (**This is the point of the PPC context switch.**)

Line 468

The current pointer to the previous task is returned from `_switch()` in `r3`. This represents the last task.

Line 469

The current pointer (`r2`) is updated with the pointer to the new task structure (`r4`).

Lines 478–486

Restore the rest of the environment from the new stack and return to the caller with the previous task structure in `r3`.

This concludes the explanation of `context_switch()`. At this point, the processor has swapped the two processes `prev` and `next` as called by `context_switch` in `schedule()`.

```
---------------------------------------------------------------------
kernel/sched.c
1709   prev = context_switch(rq, prev, next);
---------------------------------------------------------------------
```

`prev` now points to the process that we have just switched away from and next points to the current process.

Now that we've discussed how tasks are scheduled in the Linux kernel, we can examine how tasks are told to be scheduled. Namely, what causes `schedule()` to be called and one process to yield the CPU to another process?

7.1.3 Yielding the CPU

Processes can voluntarily yield the CPU by simply calling `schedule()`. This is most commonly used in kernel code and device drivers that want to sleep or wait for a signal to occur.[7] Other tasks want to continually use the CPU and the system timer must tell them to yield. The Linux kernel periodically seizes the CPU, in so doing stopping the active process, and then does a number of timer-based tasks. One of these tasks, `scheduler_tick()`, is how the kernel forces a process to yield. If a process has been running for too long, the kernel does not return control to that process and instead chooses another one. We now examine how `scheduler_tick()` determines if the current process must yield the CPU:

```
---------------------------------------------------------------------
kernel/sched.c
1981 void scheduler_tick(int user_ticks, int sys_ticks)
1982 {
1983   int cpu = smp_processor_id();
1984   struct cpu_usage_stat *cpustat = &kstat_this_cpu.cpustat;
1985   runqueue_t *rq = this_rq();
1986   task_t *p = current;
1987
```

[7] Linux convention specifies that you should *never* call `schedule` while holding a spinlock because this introduces the possibility of system deadlock. This is good advice!

```
1988   rq->timestamp_last_tick = sched_clock();
1989
1990   if (rcu_pending(cpu))
1991    rcu_check_callbacks(cpu, user_ticks);
-----------------------------------------------------------------------
```

Lines 1981–1986

This code block initializes the data structures that the `scheduler_tick()` function needs. `cpu`, `cpu_usage_stat`, and `rq` are set to the processor ID, CPU stats and run queue of the current processor. `p` is a pointer to the current process executing on `cpu`.

Line 1988

The run queue's last tick is set to the current time in nanoseconds.

Lines 1990–1991

On an SMP system, we need to check if there are any outstanding read-copy updates to perform (RCU). If so, we perform them via `rcu_check_callback()`.

```
-----------------------------------------------------------------------
kernel/sched.c
1993   /* note: this timer irq context must be accounted for as well */
1994   if (hardirq_count() - HARDIRQ_OFFSET) {
1995     cpustat->irq += sys_ticks;
1996     sys_ticks = 0;
1997   } else if (softirq_count()) {
1998     cpustat->softirq += sys_ticks;
1999     sys_ticks = 0;
2000   }
2001
2002   if (p == rq->idle) {
2003     if (atomic_read(&rq->nr_iowait) > 0)
2004       cpustat->iowait += sys_ticks;
2005     else
2006       cpustat->idle += sys_ticks;
2007     if (wake_priority_sleeper(rq))
2008       goto out;
2009     rebalance_tick(cpu, rq, IDLE);
2010     return;
2011   }
2012   if (TASK_NICE(p) > 0)
2013     cpustat->nice += user_ticks;
2014   else
2015     cpustat->user += user_ticks;
2016   cpustat->system += sys_ticks;
-----------------------------------------------------------------------
```

Lines 1994–2000

`cpustat` keeps track of kernel statistics, and we update the hardware and software interrupt statistics by the number of system ticks that have occurred.

Lines 2002–2011

If there is no currently running process, we atomically check if any processes are waiting on I/O. If so, the CPU I/O wait statistic is incremented; otherwise, the CPU idle statistic is incremented. In a uniprocessor system, `rebalance_tick()` does nothing, but on a multiple processor system, `rebalance_tick()` attempts to load balance the current CPU because the CPU has nothing to do.

Lines 2012–2016

More CPU statistics are gathered in this code block. If the current process was `niced`, we increment the CPU `nice` counter; otherwise, the user tick counter is incremented. Finally, we increment the CPU's system tick counter.

```
-----------------------------------------------------------------------
kernel/sched.c
2019   if (p->array != rq->active) {
2020     set_tsk_need_resched(p);
2021     goto out;
2022   }
2023   spin_lock(&rq->lock);
-----------------------------------------------------------------------
```

Lines 2019–2022

Here, we see why we store a pointer to a priority array within the `task_struct` of the process. The scheduler checks the current process to see if it is no longer active. If the process has expired, the scheduler sets the process' rescheduling flag and jumps to the end of the `scheduler_tick()` function. At that point (lines 2092–2093), the scheduler attempts to load balance the CPU because there is no active task yet. This case occurs when the scheduler grabbed CPU control before the current process was able to schedule itself or clean up from a successful run.

Line 2023

At this point, we know that the current process was running and not expired or nonexistent. The scheduler now wants to yield CPU control to another process; the first thing it must do is take the run queue lock.

```
kernel/sched.c
2024   /*
2025   * The task was running during this tick - update the
2026   * time slice counter. Note: we do not update a thread's
2027   * priority until it either goes to sleep or uses up its
2028   * timeslice. This makes it possible for interactive tasks
2029   * to use up their timeslices at their highest priority levels.
2030   */
2031   if (unlikely(rt_task(p))) {
2032     /*
2033     * RR tasks need a special form of timeslice management.
2034     * FIFO tasks have no timeslices.
2035     */
2036     if ((p->policy == SCHED_RR) && !--p->time_slice) {
2037       p->time_slice = task_timeslice(p);
2038       p->first_time_slice = 0;
2039       set_tsk_need_resched(p);
2040
2041       /* put it at the end of the queue: */
2042       dequeue_task(p, rq->active);
2043       enqueue_task(p, rq->active);
2044     }
2045     goto out_unlock;
2046   }
```

Lines 2031–2046

The easiest case for the scheduler occurs when the current process is a real-time task. Real-time tasks always have a higher priority than any other tasks. If the task is a FIFO task and was running, it should continue its operation so we jump to the end of the function and release the run queue lock. If the current process is a round-robin real-time task, we decrement its timeslice. If the task has no more timeslice, it's time to schedule another round-robin real-time task. The current task has its new timeslice calculated by `task_timeslice()`. Then the task has its first timeslice reset. The task is then marked as needing rescheduling and, finally, the task is put at the end of the round-robin real-time tasklist by removing it from the run queue's active array and adding it back in. The scheduler then jumps to the end of the function and releases the run queue lock.

```
kernel/sched.c
2047   if (!--p->time_slice) {
2048     dequeue_task(p, rq->active);
2049     set_tsk_need_resched(p);
2050     p->prio = effective_prio(p);
```

```
2051     p->time_slice = task_timeslice(p);
2052     p->first_time_slice = 0;
2053
2054     if (!rq->expired_timestamp)
2055       rq->expired_timestamp = jiffies;
2056     if (!TASK_INTERACTIVE(p) || EXPIRED_STARVING(rq)) {
2057       enqueue_task(p, rq->expired);
2058       if (p->static_prio < rq->best_expired_prio)
2059         rq->best_expired_prio = p->static_prio;
2060     } else
2061       enqueue_task(p, rq->active);
2062   } else {
-----------------------------------------------------------------------
```

Lines 2047–2061

At this point, the scheduler knows that the current process is not a real-time process. It decrements the process' timeslice and, in this section, the process' timeslice has been exhausted and reached 0. The scheduler removes the task from the active array and sets the process' rescheduling flag. The priority of the task is recalculated and its timeslice is reset. Both of these operations take into account prior process activity.[8] If the run queue's expired timestamp is 0, which usually occurs when there are no more processes on the run queue's active array, we set it to jiffies.

Jiffies

Jiffies is a 32-bit variable counting the number of ticks since the system has been booted. This is approximately 497 days before the number wraps around to 0 on a 100HZ system. The macro on line 20 is the suggested method of accessing this value as a u64. There are also macros to help detect wrapping in `include/jiffies.h`.

```
-----------------------------------------------------------------------
include/linux/jiffies.h
017  extern unsigned long volatile jiffies;
020  u64 get_jiffies_64(void);
-----------------------------------------------------------------------
```

We normally favor interactive tasks by replacing them on the active priority array of the run queue; this is the `else` clause on line 2060. However, we don't want to starve expired tasks. To determine if expired tasks have been waiting too long for CPU time, we use `EXPIRED_STARVING()` (see `EXPIRED_STARVING` on line 1968).

[8] See `effective_prio()` and `task_timeslice()`.

The function returns true if the first expired task has been waiting an "unreasonable" amount of time or if the expired array contains a task that has a greater priority than the current process. The unreasonableness of waiting is load-dependent and the swapping of the active and expired arrays decrease with an increasing number of running tasks.

If the task is not interactive or expired tasks are starving, the scheduler takes the current process and enqueues it onto the run queue's expired priority array. If the current process' static priority is higher than the expired run queue's highest priority task, we update the run queue to reflect the fact that the expired array now has a higher priority than before. (Remember that high-priority tasks have low numbers in Linux, thus, the (<) in the code.)

```
-----------------------------------------------------------------------
kernel/sched.c
2062    } else {
2063      /*
2064      * Prevent a too long timeslice allowing a task to monopolize
2065      * the CPU. We do this by splitting up the timeslice into
2066      * smaller pieces.
2067      *
2068      * Note: this does not mean the task's timeslices expire or
2069      * get lost in any way, they just might be preempted by
2070      * another task of equal priority. (one with higher
2071      * priority would have preempted this task already.) We
2072      * requeue this task to the end of the list on this priority
2073      * level, which is in essence a round-robin of tasks with
2074      * equal priority.
2075      *
2076      * This only applies to tasks in the interactive
2077      * delta range with at least TIMESLICE_GRANULARITY to requeue.
2078      */
2079      if (TASK_INTERACTIVE(p) && !((task_timeslice(p) -
2080        p->time_slice) % TIMESLICE_GRANULARITY(p)) &&
2081        (p->time_slice >= TIMESLICE_GRANULARITY(p)) &&
2082        (p->array == rq->active)) {
2083
2084        dequeue_task(p, rq->active);
2085        set_tsk_need_resched(p);
2086        p->prio = effective_prio(p);
2087        enqueue_task(p, rq->active);
2088      }
2089    }
2090 out_unlock:
2091    spin_unlock(&rq->lock);
2092 out:
2093    rebalance_tick(cpu, rq, NOT_IDLE);
2094 }
-----------------------------------------------------------------------
```

Lines 2079–2089

The final case before the scheduler is that the current process was running and still has timeslices left to run. The scheduler needs to ensure that a process with a large timeslice doesn't hog the CPU. If the task is interactive, has more timeslices than `TIMESLICE_GRANULARITY`, and was active, the scheduler removes it from the active queue. The task then has its reschedule flag set, its priority recalculated, and is placed back on the run queue's active array. This ensures that a process at a certain priority with a large timeslice doesn't starve another process of an equal priority.

Lines 2090–2094

The scheduler has finished rearranging the run queue and unlocks it; if executing on an SMP system, it attempts to load balance.

Combining how processes are marked to be rescheduled, via `scheduler_tick()` and how processes are scheduled, `via schedule()` illustrates how the scheduler operates in the 2.6 Linux kernel. We now delve into the details of what the scheduler means by "priority."

7.1.3.1 Dynamic Priority Calculation

In previous sections, we glossed over the specifics of how a task's dynamic priority is calculated. The priority of a task is based on its prior behavior, as well as its user-specified `nice` value. The function that determines a task's new dynamic priority is `recalc_task_prio()`:

```
--------------------------------------------------------------------------
kernel/sched.c
381 static void recalc_task_prio(task_t *p, unsigned long long now)
382 {
383   unsigned long long __sleep_time = now - p->timestamp;
384   unsigned long sleep_time;
385
386   if (__sleep_time > NS_MAX_SLEEP_AVG)
387     sleep_time = NS_MAX_SLEEP_AVG;
388   else
389     sleep_time = (unsigned long)__sleep_time;
390
391   if (likely(sleep_time > 0)) {
392     /*
393     * User tasks that sleep a long time are categorised as
394     * idle and will get just interactive status to stay active &
395     * prevent them suddenly becoming cpu hogs and starving
396     * other processes.
397     */
```

```
398     if (p->mm && p->activated != -1 &&
399       sleep_time > INTERACTIVE_SLEEP(p)) {
400         p->sleep_avg = JIFFIES_TO_NS(MAX_SLEEP_AVG -
401             AVG_TIMESLICE);
402         if (!HIGH_CREDIT(p))
403           p->interactive_credit++;
404     } else {
405       /*
406       * The lower the sleep avg a task has the more
407       * rapidly it will rise with sleep time.
408       */
409       sleep_time *= (MAX_BONUS - CURRENT_BONUS(p)) ? : 1;
410
411       /*
412       * Tasks with low interactive_credit are limited to
413       * one timeslice worth of sleep avg bonus.
414       */
415       if (LOW_CREDIT(p) &&
416        sleep_time > JIFFIES_TO_NS(task_timeslice(p)))
417         sleep_time = JIFFIES_TO_NS(task_timeslice(p));
418
419       /*
420       * Non high_credit tasks waking from uninterruptible
421       * sleep are limited in their sleep_avg rise as they
422       * are likely to be cpu hogs waiting on I/O
423       */
424       if (p->activated == -1 && !HIGH_CREDIT(p) && p->mm) {
425         if (p->sleep_avg >= INTERACTIVE_SLEEP(p))
426           sleep_time = 0;
427         else if (p->sleep_avg + sleep_time >=
428             INTERACTIVE_SLEEP(p)) {
429           p->sleep_avg = INTERACTIVE_SLEEP(p);
430           sleep_time = 0;
431         }
432       }
433
434       /*
435       * This code gives a bonus to interactive tasks.
436       *
437       * The boost works by updating the 'average sleep time'
438       * value here, based on ->timestamp. The more time a
439       * task spends sleeping, the higher the average gets -
440       * and the higher the priority boost gets as well.
441       */
442       p->sleep_avg += sleep_time;
443
444       if (p->sleep_avg > NS_MAX_SLEEP_AVG) {
445         p->sleep_avg = NS_MAX_SLEEP_AVG;
446         if (!HIGH_CREDIT(p))
447           p->interactive_credit++;
448       }
```

```
449     }
450   }
452
452   p->prio = effective_prio(p);
453 }
------------------------------------------------------------------------
```

Lines 386–389

Based on the time `now`, we calculate the length of time the process `p` has slept for and assign it to `sleep_time` with a maximum value of `NS_MAX_SLEEP_AVG`. (`NS_MAX_SLEEP_AVG` defaults to 10 milliseconds.)

Lines 391–404

If process `p` has slept, we first check to see if it has slept enough to be classified as an interactive task. If it has, when `sleep_time > INTERACTIVE_SLEEP(p)`, we adjust the process' sleep average to a set value and, if `p` isn't classified as interactive yet, we increment `p`'s `interactive_credit`.

Lines 405–410

A task with a low sleep average gets a higher sleep time.

Lines 411–418

If the task is CPU intensive, and thus classified as non-interactive, we restrict the process to having, at most, one more timeslice worth of a sleep average bonus.

Lines 419–432

Tasks that are not yet classified as interactive (not `HIGH_CREDIT`) that awake from uninterruptible sleep are restricted to having a sleep average of `INTERACTIVE()`.

Lines 434–450

We add our newly calculated `sleep_time` to the process' sleep average, ensuring it doesn't go over `NS_MAX_SLEEP_AVG`. If the processes are not considered interactive but have slept for the maximum time or longer, we increment its interactive credit.

Line 452

Finally, the priority is set using `effective_prio()`, which takes into account the newly calculated `sleep_avg` field of `p`. It does this by scaling the sleep average

of 0 . . MAX_SLEEP_AVG into the range of -5 to +5. Thus, a process that has a static priority of 70 can have a dynamic priority between 65 and 85, depending on its prior behavior.

One final thing: A process that is not a real-time process has a range between 101 and 140. Processes that are operating at a very high priority, 105 or less, cannot cross the real-time boundary. Thus, a high priority, highly interactive process could never have a dynamic priority of lower than 101. (Real-time processes cover 0..100 in the default configuration.)

7.1.3.2 Deactivation

We already discussed how a task gets inserted into the scheduler by forking and how tasks move from the active to expired priority arrays within the CPU's run queue. But, how does a task ever get removed from a run queue?

A task can be removed from the run queue in two major ways:

- The task is preempted by the kernel and its state is not running, and there is no signal pending for the task (see line 2240 in kernel/sched.c).
- On SMP machines, the task can be removed from a run queue and placed on another run queue (see line 3384 in kernel/sched.c).

The first case normally occurs when schedule() gets called after a process puts itself to sleep on a wait queue. The task marks itself as non-running (TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, TASK_STOPPED, and so on) and the kernel no longer considers it for CPU access by removing it from the run queue.

The case in which the process is moved to another run queue is dealt with in the SMP section of the Linux kernel, which we do not explore here.

We now trace how a process is removed from the run queue via deactivate_task():

```
-----------------------------------------------------------------------
kernel/sched.c
507 static void deactivate_task(struct task_struct *p, runqueue_t *rq)
508 {
509   rq->nr_running--;
510   if (p->state == TASK_UNINTERRUPTIBLE)
511     rq->nr_uninterruptible++;
512   dequeue_task(p, p->array);
513   p->array = NULL;
514 }
-----------------------------------------------------------------------
```

Line 509

The scheduler first decrements its count of running processes because `p` is no longer running.

Lines 510–511

If the task is uninterruptible, we increment the count of uninterruptible tasks on the run queue. The corresponding decrement operation occurs when an uninterruptible process wakes up (see `kernel/sched.c` line 824 in the function `try_to_wake_up()`).

Line 512–513

Our run queue statistics are now updated so we actually remove the process from the run queue. The kernel uses the `p->array` field to test if a process is running and on a run queue. Because it no longer is either, we set it to `NULL`.

There is still some run queue management to be done; let's examine the specifics of `dequeue_task()`:

```
-----------------------------------------------------------------------
kernel/sched.c
303 static void dequeue_task(struct task_struct *p, prio_array_t *array)
304 {
305   array->nr_active--;
306   list_del(&p->run_list);
307   if (list_empty(array->queue + p->prio))
308     __clear_bit(p->prio, array->bitmap);
309 }
-----------------------------------------------------------------------
```

Line 305

We adjust the number of active tasks on the priority array that process `p` is on—either the expired or the active array.

Lines 306–308

We remove the process from the list of processes in the priority array at `p`'s priority. If the resulting list is empty, we need to clear the bit in the priority array's bitmap to show there are no longer any processes at priority `p->prio()`.

`list_del()` does all the removal in one step because `p->run_list` is a `list_head` structure and thus has pointers to the previous and next entries in the list.

We have reached the point where the process is removed from the run queue and has thus been completely deactivated. If this process had a state of `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`, it could be awoken and placed back on a run queue. If the process had a state of `TASK_STOPPED`, `TASK_ZOMBIE`, or `TASK_DEAD`, it has all of its structures removed and discarded.

7.2 Preemption

Preemption is the switching of one task to another. We mentioned how `schedule()` and `scheduler_tick()` decide which task to switch to next, but we haven't described how the Linux kernel decides when to switch. The 2.6 kernel introduces kernel preemption, which means that both user space programs and kernel space programs can be switched at various times. Because kernel preemption is the standard in Linux 2.6, we describe how full kernel and user preemption operates in Linux.

7.2.1 Explicit Kernel Preemption

The easiest preemption to understand is explicit kernel preemption. This occurs in kernel space when kernel code calls `schedule()`. Kernel code can call `schedule()` in two ways, either by directly calling `schedule()` or by blocking.

When the kernel is explicitly preempted, as in a device driver waiting with a `wait_queue`, the control is simply passed to the scheduler and a new task is chosen to run.

7.2.2 Implicit User Preemption

When the kernel has finished processing a kernel space task and is ready to pass control to a user space task, it first checks to see which user space task it should pass control to. This might not be the user space task that passed its control to the kernel. For example, if Task A invokes a system call, after the system call completes, the kernel could pass control of the system to Task B.

Each task on the system has a "rescheduling necessary" flag that is set whenever a task should be rescheduled:

```
-----------------------------------------------------------------------
include/linux/sched.h
988 static inline void set_tsk_need_resched(struct task_struct *tsk)
989 {
990   set_tsk_thread_flag(tsk,TIF_NEED_RESCHED);
991 }
992
993 static inline void clear_tsk_need_resched(struct task_struct *tsk)
994 {
995   clear_tsk_thread_flag(tsk,TIF_NEED_RESCHED);
996 }
...
1003 static inline int need_resched(void)
1004 {
1005   return unlikely(test_thread_flag(TIF_NEED_RESCHED));
1006 }
-----------------------------------------------------------------------
```

Lines 988–996

`set_tsk_need_resched` and `clear_tsk_need_resched` are the interfaces provided to set the architecture-specific flag `TIF_NEED_RESCHED`.

Lines 1003–1006

`need_resched` tests the current thread's flag to see if `TIF_NEED_RESCHED` is set.

When the kernel is returning to user space, it chooses a process to pass control to, as described in `schedule()` and `scheduler_tick()`. Although `scheduler_tick()` can mark a task as needing rescheduling, only `schedule()` operates on that knowledge. `schedule()` repeatedly chooses a new task to execute until the newly chosen task does not need to be rescheduled. After `schedule()` completes, the new task has control of the processor.

Thus, while a process is running, the system timer causes an interrupt that triggers `scheduler_tick()`. `scheduler_tick()` can mark that task as needing rescheduling and move it to the expired array. Upon completion of kernel operations, `scheduler_tick()` could be followed by other interrupts and the kernel would continue to have control of the processor—`schedule()` is invoked to choose the next task to run. So, the `scheduler_tick()` marks processes and rearranges queues, but `schedule()` chooses the next task and passes CPU control.

7.2.3 Implicit Kernel Preemption

New in Linux 2.6 is the implementation of implicit kernel preemption. When a kernel task has control of the CPU, it can only be preempted by another kernel task if it does not currently hold any locks. Each task has a field, `preempt_count`, which marks whether the task is preemptible. The count is incremented every time the task obtains a lock and decremented whenever the task releases a lock. The `schedule()` function disables preemption while it determines which task to run next.

There are two possibilities for implicit kernel preemption: Either the kernel code is emerging from a code block that had preemption disabled or processing is returning to kernel code from an interrupt. If control is returning to kernel space from an interrupt, the interrupt calls `schedule()` and a new task is chosen in the same way as just described.

If the kernel code is emerging from a code block that disabled preemption, the act of enabling preemption can cause the current task to be preempted:

```
-----------------------------------------------------------------------
include/linux/preempt.h
46 #define preempt_enable() \
47 do { \
48   preempt_enable_no_resched(); \
49   preempt_check_resched(); \
50 } while (0)
-----------------------------------------------------------------------
```

Lines 46–50

`preempt_enable()` calls `preempt_enable_no_resched()`, which decrements the `preempt_count` on the current task by one and then calls `preempt_check_resched()`:

```
-----------------------------------------------------------------------
include/linux/preempt.h
40 #define preempt_check_resched() \
41 do { \
42   if (unlikely(test_thread_flag(TIF_NEED_RESCHED))) \
43     preempt_schedule(); \
44 } while (0)
-----------------------------------------------------------------------
```

Lines 40–44

`preempt_check_resched()` sees if the current task has been marked for rescheduling; if so, it calls `preempt_schedule()`.

```
-----------------------------------------------------------------------
kernel/sched.c
2328 asmlinkage void __sched preempt_schedule(void)
2329 {
2330   struct thread_info *ti = current_thread_info();
2331
2332   /*
2333   * If there is a non-zero preempt_count or interrupts are disabled,
2334   * we do not want to preempt the current task. Just return..
2335   */
2336   if (unlikely(ti->preempt_count || irqs_disabled()))
2337     return;
2338
2339 need_resched:
2340   ti->preempt_count = PREEMPT_ACTIVE;
2341   schedule();
2342   ti->preempt_count = 0;
2343
2344  /* we could miss a preemption opportunity between schedule and now */
2345   barrier();
2346   if (unlikely(test_thread_flag(TIF_NEED_RESCHED)))
2347     goto need_resched;
2348 }
-----------------------------------------------------------------------
```

Line 2336–2337

If the current task still has a positive `preempt_count`, likely from nesting `preempt_disable()` commands, or the current task has interrupts disabled, we return control of the processor to the current task.

Line 2340–2347

The current task has no locks because `preempt_count` is 0 and IRQs are enabled. Thus, we set the current tasks `preempt_count` to note it's undergoing preemption, and call `schedule()`, which chooses another task.

If the task emerging from the code block needs rescheduling, the kernel needs to ensure it's safe to yield the processor from the current task. The kernel checks the task's value of `preempt_count`. If `preempt_count` is 0, and thus the current task holds no locks, `schedule()` is called and a new task is chosen for execution. If `preempt_count` is non-zero, it is unsafe to pass control to another task, and control is returned to the current task until it releases all of its locks. When the current task releases locks, a test is made to see if the current task needs rescheduling.

When the current task releases its final lock and `preempt_count` goes to 0, scheduling immediately occurs.

7.3 Spinlocks and Semaphores

When two or more processes require dedicated access to a shared resource, they might need to enforce the condition that they are the sole process to operate in a given section of code. The basic form of locking in the Linux kernel is the spinlock.

Spinlocks take their name from the fact that they continuously loop, or *spin*, waiting to acquire a lock. Because spinlocks operate in this manner, it is imperative not to have any section of code inside a spinlock attempt to acquire a lock twice. This results in deadlock.

Before operating on a spinlock, the `spin_lock_t` structure must be initialized. This is done by calling `spin_lock_init()`:

```
------------------------------------------------------------------------
include/linux/spinlock.h
63 #define spin_lock_init(x) \
64   do { \
65     (x)->magic = SPINLOCK_MAGIC; \
66     (x)->lock = 0; \
67     (x)->babble = 5; \
68     (x)->module = __FILE__; \
69     (x)->owner = NULL; \
70     (x)->oline = 0; \
71   } while (0)
------------------------------------------------------------------------
```

This section of code sets the `spin_lock` to "unlocked," or 0, on line 66 and initializes the other variables in the structure. The `(x)->lock` variable is the one we're concerned about here.

After a `spin_lock` is initialized, it can be acquired by calling `spin_lock()` or `spin_lock_irqsave()`. The `spin_lock_irqsave()` function disables interrupts before locking, whereas `spin_lock()` does not. If you use `spin_lock()`, the process could be interrupted in the locked section of code.

To release a `spin_lock` after executing the critical section of code, you need to call `spin_unlock()` or `spin_unlock_irqrestore()`. The `spin_unlock_irqrestore()` restores the state of the interrupt registers to the state they were in when `spin_lock_irq()` was called.

Let's examine the `spin_lock_irqsave()` and `spin_unlock_irqrestore()` calls:

```
------------------------------------------------------------------------
include/linux/spinlock.h
258 #define spin_lock_irqsave(lock, flags) \
259 do { \
260   local_irq_save(flags); \
261   preempt_disable(); \
262   _raw_spin_lock_flags(lock, flags); \
263 } while (0)
...
321 #define spin_unlock_irqrestore(lock, flags) \
322 do { \
323   _raw_spin_unlock(lock); \
324   local_irq_restore(flags); \
325   preempt_enable(); \
326 } while (0)
------------------------------------------------------------------------
```

Notice how preemption is disabled during the lock. This ensures that any operation in the critical section is not interrupted. The IRQ flags saved on line 260 are restored on line 324.

The drawback of spinlocks is that they busily loop, waiting for the lock to be freed. They are best used for critical sections of code that are fast to complete. For code sections that take time, it is better to use another Linux kernel locking utility: the semaphore.

Semaphores differ from spinlocks because the task sleeps, rather than busy waits, when it attempts to obtain a contested resource. One of the main advantages is that a process holding a semaphore is safe to block; they are SMP and interrupt safe:

```
------------------------------------------------------------------------
include/asm-i386/semaphore.h
44 struct semaphore {
45   atomic_t count;
46   int sleepers;
47   wait_queue_head_t wait;
48 #ifdef WAITQUEUE_DEBUG
49   long __magic;
50 #endif
51 };
------------------------------------------------------------------------

------------------------------------------------------------------------
include/asm-ppc/semaphore.h
24 struct semaphore {
25   /*
```

```
26   * Note that any negative value of count is equivalent to 0,
27   * but additionally indicates that some process(es) might be
28   * sleeping on 'wait'.
29   */
30   atomic_t count;
31   wait_queue_head_t wait;
32 #ifdef WAITQUEUE_DEBUG
33   long __magic;
34 #endif
35 };
-----------------------------------------------------------------------
```

Both architecture implementations provide a pointer to a `wait_queue` and a count. The count is the number of processes that can hold the semaphore at the same time. With semaphores, we could have more than one process entering a critical section of code at the same time. If the count is initialized to 1, only one process can enter the critical section of code; a semaphore with a count of 1 is called a mutex.

Semaphores are initialized using `sema_init()` and are locked and unlocked by calling `down()` and `up()`, respectively. If a process calls `down()` on a locked semaphore, it blocks and ignores all signals sent to it. There also exists `down_interruptible()`, which returns 0 if the semaphore is obtained and `-EINTR` if the process was interrupted while blocking.

When a process calls `down()`, or `down_interruptible()`, the count field in the semaphore is decremented. If that field is less than 0, the process calling `down()` is blocked and added to the semaphore's `wait_queue`. If the field is greater than or equal to 0, the process continues.

After executing the critical section of code, the process should call `up()` to inform the semaphore that it has finished the critical section. By calling `up()`, the process increments the `count` field in the semaphore and, if the count is greater than or equal to 0, wakes a process waiting on the semaphore's `wait_queue`.

7.4 System Clock: Of Time and Timers

For scheduling, the kernel uses the system clock to know how long a task has been running. We already covered the system clock in Chapter 5 by using it as an example for the discussion on interrupts. Here, we explore the Real-Time Clock and its uses and implementation; but first, let's recap clocks in general.

The clock is a periodic signal applied to a processor, which allows it to function in the time domain. The processor depends on the clock signal to know when it can perform its next function, such as adding two integers or fetching data from memory. The speed of this clock signal (1.4GHz, 2GHz, and so on) has historically been used to compare the processing speed of systems at the local electronics store.

At any given moment, your system has several clocks and/or timers running. Simple examples include the time of day displayed in the bottom corner of your screen (otherwise known as wall time), the cursor patiently pulsing on a cluttered desktop, or your laptop screensaver taking over because of inactivity. More complicated examples of timekeeping include audio and video playback, key repeat (holding a key down), how fast communications ports run, and, as previously discussed, how long a task can run.

7.4.1 Real-Time Clock: What Time Is It?

The Linux interface to *wall clock time* is accomplished through the `/dev/rtc` device driver `ioctl()` function. The device for this driver is called a Real-Time Clock (RTC). The RTC[9] provides timekeeping functions with a small 114-byte user NVRAM. The input to this device is a 32.768KHz oscillator and a connection for battery backup. Some discrete models of the RTC have the oscillator and battery built in, while other RTCs are now built in to the peripheral bus controller (for example, the Southbridge) of a processor chipset. The RTC not only reports the time of day, but it is also a programmable timer that is capable of interrupting the system. The frequency of interrupts varies from 2Hz to 8,192Hz. The RTC can also interrupt daily, like an alarm clock. Here, we explore the RTC code:

```
------------------------------------------------------------------------
/include/linux/rtc.h

/*
 * ioctl calls that are permitted to the /dev/rtc interface, if
 * any of the RTC drivers are enabled.
 */

70  #define RTC_AIE_ON   _IO('p', 0x01)  /* Alarm int. enable on */
71  #define RTC_AIE_OFF   _IO('p', 0x02)  /* ... off   */
72  #define RTC_UIE_ON   _IO('p', 0x03)  /* Update int. enable on  */
73  #define RTC_UIE_OFF   _ IO('p', 0x04)  /* ... off    */
74  #define RTC_PIE_ON   _IO('p', 0x05)  /* Periodic int. enable on  */
```

[9] Manufactured by several vendors, most notably Motorola, with the mc146818. (This RTC is no longer in production. The Dallas DS12885 or equivalent is used instead.)

```
75  #define RTC_PIE_OFF   _IO('p', 0x06)  /* ... off    */
76  #define RTC_WIE_ON   _IO('p', 0x0f) /* Watchdog int. enable on  */
77  #define RTC_WIE_OFF   _IO('p', 0x10) /* ... off    */

78  #define RTC_ALM_SET   _IOW('p', 0x07, struct rtc_time) /* Set alarm time */
79  #define RTC_ALM_READ  _IOR('p', 0x08, struct rtc_time) /* Read alarm time*/
80  #define RTC_RD_TIME   _IOR('p', 0x09, struct rtc_time) /* Read RTC time */
81  #define RTC_SET_TIME  _IOW('p', 0x0a, struct rtc_time) /* Set RTC time */
82  #define RTC_IRQP_READ  _IOR('p', 0x0b, unsigned long)  /* Read IRQ rate*/
83  #define RTC_IRQP_SET  _IOW('p', 0x0c, unsigned long)  /* Set IRQ rate */
84  #define RTC_EPOCH_READ  _IOR('p', 0x0d, unsigned long)  /* Read epoch */
85  #define RTC_EPOCH_SET  _IOW('p', 0x0e, unsigned long)  /* Set epoch */
86
87  #define RTC_WKALM_SET  _IOW('p', 0x0f, struct rtc_wkalrm)/*Set wakeupalarm*/
88  #define RTC_WKALM_RD  _IOR('p', 0x10, struct rtc_wkalrm)/*Get wakeupalarm*/
89
90  #define RTC_PLL_GET   _IOR('p', 0x11, struct rtc_pll_info) /* Get PLL
correction */
91  #define RTC_PLL_SET   _IOW('p', 0x12, struct rtc_pll_info) /* Set PLL
correction */
-------------------------------------------------------------------------
```

The `ioctl()` control functions are listed in `include/linux/rtc.h`. At this writing, not all the `ioctl()` calls for the RTC are implemented for the PPC architecture. These control functions each call lower-level hardware-specific functions (if implemented). The example in this section uses the `RTC_RD_TIME` function.

The following is a sample `ioctl()` call to get the time of day. This program simply opens the driver and queries the RTC hardware for the current date and time, and prints the information to `stderr`. Note that only one user can access the RTC driver at a time. The code to enforce this is shown in the driver discussion.

```
-------------------------------------------------------------------------
Documentation/rtc.txt
/*
 *  Trimmed down version of code in /Documentation/rtc.txt
 *
 */


int main(void) {

int fd, retval = 0;
//unsigned long tmp, data;
struct rtc_time rtc_tm;

fd = open ("/dev/rtc", O_RDONLY);


/* Read the RTC time/date */
```

```
retval = ioctl(fd, RTC_RD_TIME, &rtc_tm);

/* print out the time from the rtc_tm variable */

close(fd);
return 0;

} /* end main */
-----------------------------------------------------------------------
```

This code is a segment of a more complete example in `/Documentation/rtc.txt`. The two main lines of code in this program are the `open()` command and the `ioctl()` call. `open()` tells us which driver we will use (`/dev/rtc`) and `ioctl()` indicates a specific path through the code down to the physical RTC interface by way of the `RTC_RD_TIME` command. The driver code for the `open()` command resides in the driver source, but its only significance to this discussion is *which* device driver was opened.

7.4.2 Reading the PPC Real-Time Clock

At kernel compile time, the appropriate code tree (x86, PPC, MIPS, and so on) is inserted. The source branch for PPC is discussed here in the source code file for the generic RTC driver for non-x86 systems:

```
-----------------------------------------------------------------------
/drivers/char/genrtc.c
276  static int gen_rtc_ioctl(struct inode *inode, struct file *file,
277    unsigned int cmd, unsigned long arg)
278  {
279   struct rtc_time wtime;
280   struct rtc_pll_info pll;
281
282   switch (cmd) {
283
284   case RTC_PLL_GET:
...
290   case RTC_PLL_SET:
...
298   case RTC_UIE_OFF:  /* disable ints from RTC updates.  */
...
302   case RTC_UIE_ON:  /* enable ints for RTC updates.  */
...
305   case RTC_RD_TIME:  /* Read the time/date from RTC  */
306
307    memset(&wtime, 0, sizeof(wtime));
308    get_rtc_time(&wtime);
309
310    return copy_to_user((void *)arg,&wtime,sizeof(wtime)) ? -EFAULT:0;
311
```

```
312   case RTC_SET_TIME:  /* Set the RTC */
313    return -EINVAL;
314   }
...
353  static int gen_rtc_open(struct inode *inode, struct file *file)
354  {
355   if (gen_rtc_status & RTC_IS_OPEN)
356    return -EBUSY;
357   gen_rtc_status |= RTC_IS_OPEN;
-------------------------------------------------------------------------
```

This code is the case statement for the `ioctl` command set. Because we made the `ioctl` call from the user space test program with the `RTC_RD_TIME` flag, control is transferred to line 305. The next call is at line 308, `get_rtc_time(&wtime)` in `rtc.h` (see the following code). Before leaving this code segment, note line 353. This allows only one user to access, via `open()`, the driver at a time by setting the status to `RTC_IS_OPEN`:

```
-------------------------------------------------------------------------
include/asm-ppc/rtc.h
045  static inline unsigned int get_rtc_time(struct rtc_time *time)
046  {
047    if (ppc_md.get_rtc_time) {
048     unsigned long nowtime;
049
050    nowtime = (ppc_md.get_rtc_time)();
051
052     to_tm(nowtime, time);
053
054     time->tm_year -= 1900;
055  time->tm_mon -= 1; /* Make sure userland has a 0-based month */
056    }
057   return RTC_24H;
058  }
-------------------------------------------------------------------------
```

The inline function `get_rtc_time()` calls the function that the structure variable pointed at by `ppc_md.get_rtc_time` on line 50. Early in the kernel initialization, this variable is set in `chrp_setup.c`:

```
-------------------------------------------------------------------------
arch/ppc/platforms/chrp_setup.c
447  chrp_init(unsigned long r3, unsigned long r4, unsigned long r5,
448  unsigned long r6, unsigned long r7)
449  {
...
477   ppc_md.time_init  = chrp_time_init;
478   ppc_md.set_rtc_time = chrp_set_rtc_time;
479   ppc_md.get_rtc_time = chrp_get_rtc_time;
480   ppc_md.calibrate_decr = chrp_calibrate_decr;
-------------------------------------------------------------------------
```

The function `chrp_get_rtc_time()` (on line 479) is defined in `chrp_time.c` in the following code segment. Because the time information in CMOS memory is updated on a periodic basis, the block of read code is enclosed in a `for` loop, which rereads the block if the update is in progress:

```
-------------------------------------------------------------------------
arch/ppc/platforms/chrp_time.c
122  unsigned long __chrp chrp_get_rtc_time(void)
123  {
124   unsigned int year, mon, day, hour, min, sec;
125   int uip, i;
...
141   for ( i = 0; i<1000000; i++) {
142    uip = chrp_cmos_clock_read(RTC_FREQ_SELECT);
143    sec = chrp_cmos_clock_read(RTC_SECONDS);
144    min = chrp_cmos_clock_read(RTC_MINUTES);
145    hour = chrp_cmos_clock_read(RTC_HOURS);
146    day = chrp_cmos_clock_read(RTC_DAY_OF_MONTH);
147    mon = chrp_cmos_clock_read(RTC_MONTH);
148    year = chrp_cmos_clock_read(RTC_YEAR);
149    uip |= chrp_cmos_clock_read(RTC_FREQ_SELECT);
150    if ((uip & RTC_UIP)==0) break;
151   }
152   if (!(chrp_cmos_clock_read(RTC_CONTROL)
153   & RTC_DM_BINARY) || RTC_ALWAYS_BCD)
154   {
155    BCD_TO_BIN(sec);
156    BCD_TO_BIN(min);
157    BCD_TO_BIN(hour);
158    BCD_TO_BIN(day);
159    BCD_TO_BIN(mon);
160    BCD_TO_BIN(year);
161   }
...
054  int __chrp chrp_cmos_clock_read(int addr)
055  {   if (nvram_as1 != 0)
056   outb(addr>>8, nvram_as1);
057   outb(addr, nvram_as0);
058   return (inb(nvram_data));
059  }
-------------------------------------------------------------------------
```

Finally, in `chrp_get_rtc_time()`, the values of the individual components of the time structure are read from the RTC device by using the function `chrp_cmos_clock_read`. These values are formatted and returned in the `rtc_tm` structure that was passed into the `ioctl` call back in the userland test program.

7.4.3 Reading the x86 Real-Time Clock

The methodology for reading the RTC on the x86 system is similar to, but somewhat more compact and robust than, the PPC method. Once again, we follow the open driver `/dev/rtc`, but this time, the build has compiled the file `rtc.c` for the x86 architecture. The source branch for x86 is discussed here:

```
------------------------------------------------------------------------
drivers/char/rtc.c
...
352  static int rtc_do_ioctl(unsigned int cmd, unsigned long arg, int kernel)
353  {
...
switch (cmd) {
...
482  case RTC_RD_TIME:  /* Read the time/date from RTC  */
483  {
484   rtc_get_rtc_time(&wtime);
485   break;
486  }
...
1208  void rtc_get_rtc_time(struct rtc_time *rtc_tm)
1209  {
...
1238   spin_lock_irq(&rtc_lock);
1239   rtc_tm->tm_sec = CMOS_READ(RTC_SECONDS);
1240   rtc_tm->tm_min = CMOS_READ(RTC_MINUTES);
1241   rtc_tm->tm_hour = CMOS_READ(RTC_HOURS);
1242   rtc_tm->tm_mday = CMOS_READ(RTC_DAY_OF_MONTH);
1243   rtc_tm->tm_mon = CMOS_READ(RTC_MONTH);
1244   rtc_tm->tm_year = CMOS_READ(RTC_YEAR);
1245   ctrl = CMOS_READ(RTC_CONTROL);
...
1249  spin_unlock_irq(&rtc_lock);
1250
1251  if (!(ctrl & RTC_DM_BINARY) || RTC_ALWAYS_BCD)
1252  {
1253   BCD_TO_BIN(rtc_tm->tm_sec);
1254   BCD_TO_BIN(rtc_tm->tm_min);
1255   BCD_TO_BIN(rtc_tm->tm_hour);
1256   BCD_TO_BIN(rtc_tm->tm_mday);
1257   BCD_TO_BIN(rtc_tm->tm_mon);
1258   BCD_TO_BIN(rtc_tm->tm_year);
1259  }
------------------------------------------------------------------------
```

The test program uses the `ioctl()` flag `RTC_RD_TIME` in its call to the driver `rtc.c`. The `ioctl` switch statement then fills the time structure from the CMOS

memory of the RTC. Here is the x86 implementation of how the RTC hardware is read:

```
------------------------------------------------------------------------
include/asm-i386/mc146818rtc.h
...
018  #define CMOS_READ(addr) ({ \
019   outb_p((addr),RTC_PORT(0)); \
020   inb_p(RTC_PORT(1)); \
021  })
------------------------------------------------------------------------
```

Summary

This chapter covered the Linux scheduler, preemption in Linux, and the Linux system clock and timers.

More specifically, we covered the following topics:

- We introduced the new Linux 2.6 scheduler and outlined its new features.
- We described how the scheduler chooses the next task from among all tasks it can choose and the algorithms the scheduler uses to do so.
- We discussed the context switch that the scheduler uses to actually swap a process and traced the function into the low-level architecture-specific code.
- We covered how processes in Linux can yield the CPU to other processes by calling `schedule()` and how the kernel then marks that process as "to be scheduled."
- We delved into how the Linux kernel calculates dynamic priority based on the previous behavior of an individual process and how a process eventually gets removed from the scheduling queue.
- We then moved on and covered implicit and explicit user- and kernel-level preemption and how each is dealt with in the 2.6 Linux kernel.
- Finally, we explored timers and the system clock and how the system clock is implemented in both x86 and PPC architectures.

Exercises

1. How does Linux notify the scheduler to run periodically?
2. Describe the difference between interactive and non-interactive processes.
3. With respect to the scheduler, what's special about real-time processes?
4. What happens when a process runs out of scheduler ticks?
5. What's the advantage of an O(1) scheduler?
6. What kind of data structure does the scheduler use to manage the priority of the processes running on a system?
7. What happens if you were to call `schedule()` while holding a spinlock?
8. How does the kernel decide whether a kernel task can be implicitly preempted?