```
131   PRIVATE void init_clock()
132   {
133     /* First of all init the clock system.
134      *
135      * Here the (a) clock is set to produce a interrupt at
136      * every 1/60 second (ea. 60Hz).
137      *
138      * Running right away.
139      */
140     arch_init_clock();    /* architecture-dependent initialization. */
141
142     /* Initialize the CLOCK's interrupt hook. */
143     clock_hook.proc_nr_e = CLOCK;
144
145     put_irq_handler(&clock_hook, CLOCK_IRQ, clock_handler);
146     enable_irq(&clock_hook);              /* ready for clock interrupts */
147
148     /* Set a watchdog timer to periodically balance the scheduling queues. */
149     balance_queues(NULL);                 /* side-effect sets new timer */
150   }
151
152   /*===========================================================================*
153    *                                clock_handler                              *
154    *===========================================================================*/
155   PRIVATE int clock_handler(hook)
156   irq_hook_t *hook;
157   {
158   /* This executes on each clock tick (i.e., every time the timer chip generates
159    * an interrupt). It does a little bit of work so the clock task does not have
160    * to be called on every tick.  The clock task is called when:
161    *
162    *      (1) the scheduling quantum of the running process has expired, or
163    *      (2) a timer has expired and the watchdog function should be run.
164    *
165    * Many global global and static variables are accessed here.  The safety of
166    * this must be justified. All scheduling and message passing code acquires a
167    * lock by temporarily disabling interrupts, so no conflicts with calls from
168    * the task level can occur. Furthermore, interrupts are not reentrant, the
169    * interrupt handler cannot be bothered by other interrupts.
170    *
171    * Variables that are updated in the clock's interrupt handler:
172    *      lost_ticks:
173    *              Clock ticks counted outside the clock task. This for example
174    *              is used when the boot monitor processes a real mode interrupt.
175    *      realtime:
176    *              The current uptime is incremented with all outstanding ticks.
177    *      proc_ptr, bill_ptr:
178    *              These are used for accounting.  It does not matter if proc.c
179    *              is changing them, provided they are always valid pointers,
180    *              since at worst the previous process would be billed.
181    */
182     register unsigned ticks;
183
184     /* Get number of ticks and update realtime. */
185     ticks = lost_ticks + 1;
186     lost_ticks = 0;
187     realtime += ticks;
188
189     /* Update user and system accounting times. Charge the current process for
190      * user time. If the current process is not billable, that is, if a non-user
191      * process is running, charge the billable process for system time as well.
192      * Thus the unbillable process' user time is the billable user's system time.
193      */
194
195     proc_ptr->p_user_time += ticks;
```

```
196     proc_ptr->p_recent_time += ticks;
197     if (priv(proc_ptr)->s_flags & PREEMPTIBLE) {
198         proc_ptr->p_ticks_left -= ticks;
199     }
200     if (! (priv(proc_ptr)->s_flags & BILLABLE)) {
201         bill_ptr->p_sys_time += ticks;
202         bill_ptr->p_ticks_left -= ticks;
203     }
204
205     /* Update load average. */
206     load_update();
207
208     /* Check if do_clocktick() must be called. Done for alarms and scheduling.
209      * Some processes, such as the kernel tasks, cannot be preempted.
210      */
211     if ((next_timeout <= realtime) || (proc_ptr->p_ticks_left <= 0)) {
212         prev_ptr = proc_ptr;                        /* store running process */
213         lock_notify(HARDWARE, CLOCK);               /* send notification */
214     }
215     return(1);                                      /* reenable interrupts */
216   }
217
218   /*===========================================================================*
219    *                                get_uptime                                 *
220    *===========================================================================*/
221   PUBLIC clock_t get_uptime(void)
222   {
223     /* Get and return the current clock uptime in ticks. */
224     return(realtime);
225   }
226
227   /*===========================================================================*
228    *                                set_timer                                  *
229    *===========================================================================*/
230   PUBLIC void set_timer(tp, exp_time, watchdog)
231   struct timer *tp;               /* pointer to timer structure */
232   clock_t exp_time;               /* expiration realtime */
233   tmr_func_t watchdog;            /* watchdog to be called */
234   {
235   /* Insert the new timer in the active timers list. Always update the
236    * next timeout time by setting it to the front of the active list.
237    */
238     tmrs_settimer(&clock_timers, tp, exp_time, watchdog, NULL);
239     next_timeout = clock_timers->tmr_exp_time;
240   }
241
242   /*===========================================================================*
243    *                                reset_timer                                *
244    *===========================================================================*/
245   PUBLIC void reset_timer(tp)
246   struct timer *tp;               /* pointer to timer structure */
247   {
248   /* The timer pointed to by 'tp' is no longer needed. Remove it from both the
249    * active and expired lists. Always update the next timeout time by setting
250    * it to the front of the active list.
251    */
252     tmrs_clrtimer(&clock_timers, tp, NULL);
253     next_timeout = (clock_timers == NULL) ?
254           TMR_NEVER : clock_timers->tmr_exp_time;
255   }
256
257   /*===========================================================================*
258    *                                load_update                                *
259    *===========================================================================*/
260   PRIVATE void load_update(void)
```