```
586     check_runqueues("dequeue2");
587   #endif
588   }
589
590   /*===========================================================================*
591    *                                sched                                      *
592    *===========================================================================*/
593   PRIVATE void sched(rp, queue, front)
594   register struct proc *rp;                       /* process to be scheduled */
595   int *queue;                                     /* return: queue to use */
596   int *front;                                     /* return: front or back */
597   {
598   /* This function determines the scheduling policy.  It is called whenever a
599    * process must be added to one of the scheduling queues to decide where to
600    * insert it.  As a side-effect the process' priority may be updated.
601    */
602     int time_left = (rp->p_ticks_left > 0);       /* quantum fully consumed */
603
604     /* Check whether the process has time left. Otherwise give a new quantum
605      * and lower the process' priority, unless the process already is in the
606      * lowest queue.
607      */
608     if (! time_left) {                            /* quantum consumed ? */
609         rp->p_ticks_left = rp->p_quantum_size;    /* give new quantum */
610         if (rp->p_priority < (IDLE_Q-1)) {
611             rp->p_priority += 1;                  /* lower priority */
612         }
613     }
614
615     /* If there is time left, the process is added to the front of its queue,
616      * so that it can immediately run. The queue to use simply is always the
617      * process' current priority.
618      */
619     *queue = rp->p_priority;
620     *front = time_left;
621   }
622
623   /*===========================================================================*
624    *                                pick_proc                                  *
625    *===========================================================================*/
626   PRIVATE void pick_proc()
627   {
628   /* Decide who to run now.  A new process is selected by setting 'next_ptr'.
629    * When a billable process is selected, record it in 'bill_ptr', so that the
630    * clock task can tell who to bill for system time.
631    */
632     register struct proc *rp;                     /* process to run */
633     int q;                                        /* iterate over queues */
634     register struct proc *low_rp = NIL_PROC;      /* lowest time user process */
635     clock_t low_time = LONG_MAX;                  /* lowest time */
636
637     /* Check each of the scheduling queues except for the idle queue for ready
638      * processes. The number of queues is defined in proc.h, and priorities are
639      * set in the task table.
640      */
641     for (q = 0; q < NR_SCHED_QUEUES - 1; q++) {
642           for (rp = rdy_head[q]; rp != NIL_PROC; rp = rp->p_nextready) {
643                   /* System process. */
644                   if (priv(rp)->s_flags & SYS_PROC) {
645                           next_ptr = rp;          /* run process 'rp' next */
646                           return;
647                   }
648                   /* User process. */
649                   if (rp->p_recent_time < low_time) {
650                           low_time = rp->p_recent_time;   /* record lowest */
```

```
651                             low_rp = rp;                    /* time and proc */
652                     }
653             }
654     }
655     /* The function has scanned through the scheduling queues and determined
656      * that no system tasks wish to run while also recording the user process
657      * (if any) with the lowest recent CPU time. Now test to see if there ARE
658      * any user processes that wish to run.
659      */
660     if (low_rp != NIL_PROC) {
661             next_ptr = low_rp;                      /* run process 'low_rp' next */
662             bill_ptr = low_rp;                      /* bill for system time */
663             return;
664     }
665     /* No other process wants to run, so attempt to run the idle process. */
666     if ( (rp = rdy_head[q]) != NIL_PROC) {
667             next_ptr = rp;                          /* run process 'rp' next */
668             bill_ptr = rp;                          /* bill for system time */
669             return;
670     }
671     panic("no ready process", NO_NUM);
672   }
673
674   /*==========================================================================*
675    *                              balance_queues                              *
676    *==========================================================================*/
677   #define Q_BALANCE_TICKS  100
678   PUBLIC void balance_queues(tp)
679   timer_t *tp;                                      /* watchdog timer pointer */
680   {
681   /* Check entire process table and give all process a higher priority. This
682    * effectively means giving a new quantum. If a process already is at its
683    * maximum priority, its quantum will be renewed.
684    */
685     static timer_t queue_timer;                     /* timer structure to use */
686     register struct proc* rp;                       /* process table pointer  */
687     clock_t next_period;                            /* time of next period  */
688     int ticks_added = 0;                            /* total time added */
689
690     for (rp=BEG_PROC_ADDR; rp<END_PROC_ADDR; rp++) {
691         if (! isemptyp(rp)) {                               /* check slot use */
692             rp->p_recent_time >>= 1;                        /* decay recent cpu */
693             lock(5,"balance_queues");
694             if (rp->p_priority > rp->p_max_priority) {      /* update priority? */
695                 if (rp->p_rts_flags == 0) dequeue(rp);      /* take off queue */
696                 ticks_added += rp->p_quantum_size;          /* do accounting */
697                 rp->p_priority -= 1;                        /* raise priority */
698                 if (rp->p_rts_flags == 0) enqueue(rp);      /* put on queue */
699             }
700             else {
701                 ticks_added += rp->p_quantum_size - rp->p_ticks_left;
702                 rp->p_ticks_left = rp->p_quantum_size;      /* give new quantum */
703             }
704             unlock(5);
705         }
706     }
707   #if DEBUG
708     kprintf("ticks_added: %d\n", ticks_added);
709   #endif
710
711     /* Now schedule a new watchdog timer to balance the queues again.  The
712      * period depends on the total amount of quantum ticks added.
713      */
714     next_period = MAX(Q_BALANCE_TICKS, ticks_added);        /* calculate next */
715     set_timer(&queue_timer, get_uptime() + next_period, balance_queues);
```