OHJ-4100 Käyttöjärjestelmät

Tentti 18.3.2008

Tentissä ei saa käyttää ylimääräistä kirjallista materiaalia, laskimia, tietokoneita tai muita lunttausvälineitä.

Muutama sana tenttivastauksen kirjoittamisesta:

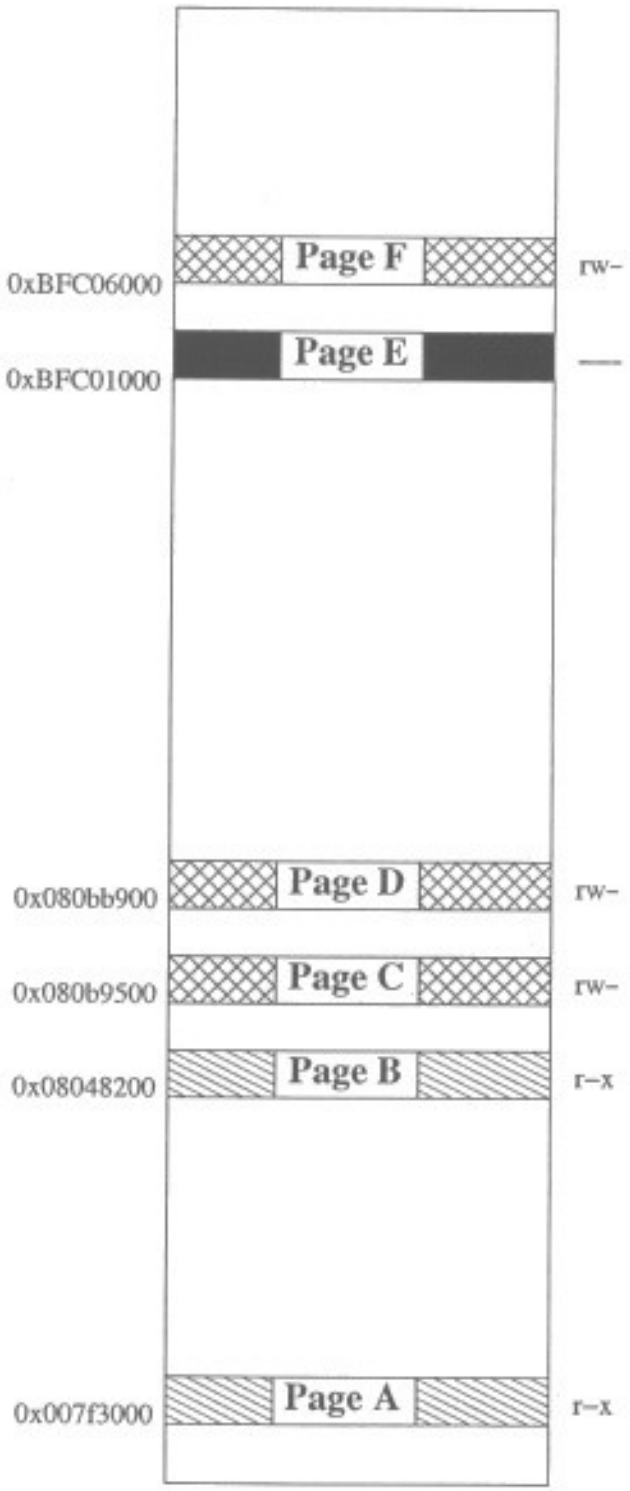
1. Vastauksessa oleta vastaavasi sellaisen ihmisen esittämään kysymykseen, joka tuntee kohtalaisen hyvin ohjelmistotekniikan aihealuetta muutoin paitsi juuri tämän kysymyksen osalta.
2. Mieti etukäteen vastauksesi pääkohdat ja lajittele ne johdonmukaiseen järjestykseen — älä kirjoita yhteen pötköön kaikkea mieleen tulevaa.
3. Muista vastata kaikkiin tehtävän kysymyslauseisiin, sillä täysiä pisteitä ei voi saada jos kaikkiin kysyttyihin asioihin ei ole vastattu.
4. Jos vastaus vaatii ohjelmakoodin kirjoitamista, sen ei tarvitse olla pilkulleen syntaksiltaan oikein. Mikä tahansa johdonmukaisesti käytetty ja yleisessä käytössä olevia ohjelmointirakenteita sisältävä koodin esitysmuoto käy.
5. Järjen käyttö on sallittua, jopa toivottavaa ☺

1. Esittele lyhyesti kurssin aihealuetta tuntemattomalle mitä seuraavat asiat ovat:
   a) Mitkä ovat käyttöjärjestelmän päätehtävät?
   b) Etuoikeutettu käsky (priviledged instruction).
   c) Sivuttava virtuaalimuisti (paging virtual memory)
   d) Ympäristön vaihto (context switch)
   e) *mount*-operaatio UFS-tiedostojärjestelmissä (kurssikirjan mallitiedostojärjestelmä).
2. Prosessilla voi olla tilat: RUN, READY, WAIT, SWAPPED WAIT ja SWAPPED READY. Selosta lyhyesti:
   - Mitä kukin tila tarkoittaa?
   - Mitkä tilasiirtymät ovat mahdollisia tilojen välillä, milloin siirtymä tapahtuu ja mikä osa käyttöjärjestelmästä tyypillisesti aiheuttaa siirtymän?
3. Juuri käynnistynyt Linux-prosessi omistaa kuvassa 1 näkyvät muistisivut. Kaaviossa virtuaaliosoitteet kasvavat alhaalta ylöspäin ja suojausbitit tarkoittavat:

r	prosessi pystyy lukemaan muistisivua
w	prosessi pystyy kirjoittamaan sivulle
x	prosessi pystyy suorittamaan ohjelmakoodia sivulta

   Selosta mitkä ajossa olevan prosessin osat sijaitsevat milläkin muistisivulla. Kerro myös missä muistisivun sisältö sijaitsee silloin kun se ei ole keskusmuistissa?
4. Käyttäjän ohjelma haluaa kirjoittaa tiedostoon `/home/kj/log/a.txt` Mitä operaatioita joudutaan tekemään käyttöjärjestelmän sisällä jotta voidaan varmistua, että operaatio voidaan sallia suojausten puolesta? Mitä (montako) levyoperaatioita (I/O) voidaan joutua tekemään kirjan esimerkkitiedostojärjestelmässä (Unix File System) jotta kaikki tarkistukset saadaan tehtyä? Kuinka nämä tarkistukset yleensä optimoidaan siten, että niitä ei tarvitse tehdä joka kerta kun ohjelma haluaa kirjoittaa samaan tiedostoon?
5. Oheisessa listauksessa on osa Minix 1.1 käyttöjärjestelmän ytimen lähdekoodia. Tutki ja kerro jokaisesta kuudesta funktiosta:
   - Mikä on funktion käyttötarkoitus? (Minkä käyttöjärjestelmien perusoperaation se toteuttaa?)
   - Mistä funktiota kutsutaan? (Laitteiston operaation seurauksena? Jostain toisesta osasta ydintä tai käyttäjän prosessista?)

   Ohjelmakoodi myös toteuttaa yhden vuoronnusalgoritmin. Miten se toimii?



Kuva 1: Prosessin käyttämät muistisivut

```
/*===========================================================================*
 *                                interrupt                                  *
 *===========================================================================*/
PUBLIC interrupt(task, m_ptr)
int task;                       /* number of task to be started */
message *m_ptr;                 /* interrupt message to send to the task */
{
/* An interrupt has occurred.  Schedule the task that handles it. */

  int i, n, old_map, this_bit;

  /* Try to send the interrupt message to the indicated task. */
  this_bit = 1 << (-task);
  if (mini_send(HARDWARE, task, m_ptr) != OK) {
        /* The message could not be sent to the task; it was not waiting. */
        old_map = busy_map;     /* save original map of busy tasks */
        if (task == CLOCK) {
                lost_ticks++;
        } else {
                busy_map |= this_bit;           /* mark task as busy */
                task_mess[-task] = m_ptr;       /* record message pointer */
        }
  } else {
        /* Hardware interrupt was successfully sent as a message. */
        busy_map &= ~this_bit;  /* turn off the bit in case it was on */
        old_map = busy_map;
  }

  /* See if any tasks that were previously busy are now listening for msgs. */
  if (old_map != 0) {
        for (i = 2; i <= NR_TASKS; i++) {
                /* Check each task looking for one with a pending interrupt. */
                if ( (old_map>>i) & 1) {
                        /* Task 'i' has a pending interrupt. */
                        n = mini_send(HARDWARE, -i, task_mess[i]);
                        if (n == OK) busy_map &= ~(1 << i);
                }
        }
  }

  /* If a task has just been readied and a user is running, run the task. */
  if (rdy_head[TASK_Q] != NIL_PROC && (cur_proc >= 0 || cur_proc == IDLE))
        pick_proc();
}


/*===========================================================================*
 *                                sys_call                                   *
 *===========================================================================*/
PUBLIC sys_call(function, caller, src_dest, m_ptr)
int function;                   /* SEND, RECEIVE, or BOTH */
int caller;                     /* who is making this call */
int src_dest;                   /* source to receive from or dest to send to */
message *m_ptr;                 /* pointer to message */
{
/* The only system calls that exist in MINIX are sending and receiving
 * messages.  These are done by trapping to the kernel with an INT instruction.
 * The trap is caught and sys_call() is called to send or receive a message (or
 * both).
 */

  register struct proc *rp;
  int n;

  /* Check for bad system call parameters. */
  rp = proc_addr(caller);
  if (src_dest < -NR_TASKS || (src_dest >= NR_PROCS && src_dest != ANY) ) {
        rp->p_reg[RET_REG] = E_BAD_SRC;
        return;
  }
  if (function != BOTH && caller >= LOW_USER) {
        rp->p_reg[RET_REG] = E_NO_PERM; /* users only do BOTH */
        return;
  }

  /* The parameters are ok. Do the call. */
  if (function & SEND) {
        n = mini_send(caller, src_dest, m_ptr); /* func = SEND or BOTH */
        if (function == SEND || n != OK) rp->p_reg[RET_REG] = n;
        if (n != OK) return;    /* SEND failed */
  }

  if (function & RECEIVE) {
        n = mini_rec(caller, src_dest, m_ptr);      /* func = RECEIVE or BOTH */
        rp->p_reg[RET_REG] = n;
  }
}

/*===========================================================================*
 *                                pick_proc                                  *
 *===========================================================================*/
PUBLIC pick_proc()
{
/* Decide who to run now. */

  register int q;               /* which queue to use */

  if (rdy_head[TASK_Q] != NIL_PROC) q = TASK_Q;
  else if (rdy_head[SERVER_Q] != NIL_PROC) q = SERVER_Q;
  else q = USER_Q;

  /* Set 'cur_proc' and 'proc_ptr'. If system is idle, set 'cur_proc' to a
   * special value (IDLE), and set 'proc_ptr' to point to an unused proc table
   * slot, namely, that of task -1 (HARDWARE), so save() will have somewhere to
   * deposit the registers when a interrupt occurs on an idle machine.
   * Record previous process so that when clock tick happens, the clock task
   * can find out who was running just before it began to run.  (While the
   * clock task is running, 'cur_proc' = CLOCKTASK. In addition, set 'bill_ptr'
   * to always point to the process to be billed for CPU time.
   */
  prev_proc = cur_proc;
  if (rdy_head[q] != NIL_PROC) {
        /* Someone is runnable. */
        cur_proc = rdy_head[q] - proc - NR_TASKS;
        proc_ptr = rdy_head[q];
        if (cur_proc >= LOW_USER) bill_ptr = proc_ptr;
  } else {
        /* No one is runnable. */
        cur_proc = IDLE;
        proc_ptr = proc_addr(HARDWARE);
        bill_ptr = proc_ptr;
  }
}
```

```
125 /*=========================================================================*
126  *                                ready                                    *
127  *=========================================================================*/
128 PUBLIC ready(rp)
129 register struct proc *rp;       /* this process is now runnable */
130 {
131 /* Add 'rp' to the end of one of the queues of runnable processes.  Three
132  * queues are maintained:
133  *   TASK_Q   - (highest priority) for runnable tasks
134  *   SERVER_Q - (middle priority) for MM and FS only
135  *   USER_Q   - (lowest priority) for user processes
136  */
137
138   register int q;               /* TASK_Q, SERVER_Q, or USER_Q */
139   int r;
140
141   lock();                       /* disable interrupts */
142   r = (rp - proc) - NR_TASKS;   /* task or proc number */
143   q = (r < 0 ? TASK_Q : r < LOW_USER ? SERVER_Q : USER_Q);
144
145   /* See if the relevant queue is empty. */
146   if (rdy_head[q] == NIL_PROC)
147         rdy_head[q] = rp;       /* add to empty queue */
148   else
149         rdy_tail[q]->p_nextready = rp;  /* add to tail of nonempty queue */
150   rdy_tail[q] = rp;             /* new entry has no successor */
151   rp->p_nextready = NIL_PROC;
152   restore();                    /* restore interrupts to previous state */
153 }
154
155
156 /*=========================================================================*
157  *                                unready                                  *
158  *=========================================================================*/
159 PUBLIC unready(rp)
160 register struct proc *rp;       /* this process is no longer runnable */
161 {
162 /* A process has blocked. */
163
164   register struct proc *xp;
165   int r, q;
166
167   lock();                       /* disable interrupts */
168   r = rp - proc - NR_TASKS;
169   q = (r < 0 ? TASK_Q : r < LOW_USER ? SERVER_Q : USER_Q);
170   if ( (xp = rdy_head[q]) == NIL_PROC) return;
171   if (xp == rp) {
172         /* Remove head of queue */
173         rdy_head[q] = xp->p_nextready;
174         pick_proc();
175   } else {
176         /* Search body of queue.  A process can be made unready even if it is
177          * not running by being sent a signal that kills it.
178          */
179         while (xp->p_nextready != rp)
180                 if ( (xp = xp->p_nextready) == NIL_PROC) return;
181         xp->p_nextready = xp->p_nextready->p_nextready;
182         while (xp->p_nextready != NIL_PROC) xp = xp->p_nextready;
183         rdy_tail[q] = xp;
184   }
185   restore();                    /* restore interrupts to previous state */
186 }
```

```
187
188
189 /*=========================================================================*
190  *                                sched                                    *
191  *=========================================================================*/
192 PUBLIC sched()
193 {
194 /* The current process has run too long.  If another low priority (user)
195  * process is runnable, put the current process on the end of the user queue,
196  * possibly promoting another user to head of the queue.
197  */
198
199   lock();                       /* disable interrupts */
200   if (rdy_head[USER_Q] == NIL_PROC) {
201         restore();              /* restore interrupts to previous state */
202         return;
203   }
204
205   /* One or more user processes queued. */
206   rdy_tail[USER_Q]->p_nextready = rdy_head[USER_Q];
207   rdy_tail[USER_Q] = rdy_head[USER_Q];
208   rdy_head[USER_Q] = rdy_head[USER_Q]->p_nextready;
209   rdy_tail[USER_Q]->p_nextready = NIL_PROC;
210   pick_proc();
211   restore();                    /* restore interrupts to previous state */
212 }
213
```