```c
/*-------------------------------------------------------------------------*/
/*   Um kernel multitarefa para DOS                                        */
/*                                                                         */
/*    Do livro Born to Code in C                                           */
/*                                                                         */
/*-------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <alloc.h>
#include "mkern0.h"

struct task_struct
{
  unsigned sp;
  unsigned ss;
  enum task_state status;    /*Estado da tarefa*/
  unsigned *pending;     /*semaforo esperando por...*/
  int sleep;             /*tempo para sleep*/
  unsigned char *stck;        /*pilha*/
} tasks[NUM_TASKS];

void interrupt (*old_int8) (void);

unsigned io_out= 0;   /* semaforo para IO, entrada e saida*/
char far *vid_mem;   /*Ponteiro para memoria de video*/


/*-------------------------------------------------------------------------*/
/*  Timer de interrupcão de tarefas dos escalonador                        */
/*-------------------------------------------------------------------------*/
void interrupt int8_task_switch(void)
{
  (*old_int8) ();         /* chama a rotina int8 original*/
  if(single_task)         /* Se uma unica tarefa esta ativa, entao retorne */
    return;          /* sem uma escolha de tarefa */
  tasks[tswitch].ss = _SS;   /* salva o status atual da tarefa/processo na pilha */
  tasks[tswitch].sp = _SP;

  /* Se uma tarefa ou processo estava rodando quando interrompida,
     entao muda o seu status para READY*/
  if(tasks[tswitch].status == RUNNING)
    tasks[tswitch].status = READY;

  /* Verifica se algum sleeper precisa acordar */
  check_sleepers();

  /* Verifica se todos os processos estao mortos. se sim, para */
  if(all_dead())
    tasking = 0;

  /* Parou */
  if(!tasking)
  {
     disable();
        _SS = oldss;
        _SP = oldsp;
       setvect(8, old_int8);
       free_all();
     enable();
     return;
  }

  /* Encontra novo processo */
  tswitch++;
  if(tswitch == NUM_TASKS)
    tswitch = 0;
  while(tasks[tswitch].status != READY)
  {
    tswitch++;
    if(tswitch == NUM_TASKS)
      tswitch = 0;
  }
    _SS = tasks[tswitch].ss;      /* troca o processo para o novo */
    _SP = tasks[tswitch].sp;
  tasks[tswitch].status = RUNNING;       /* status muda para RUNNING*/
}


/*-------------------------------------------------------------------------*/
/*   task_switch()                                                         */
/*  Alternador de tarefas alternativo que o programa pode chamar para forcar
```

```
 a mudança de processo. Não diminui o contador de sleeper, porque nao ocorreu um clock
                                                                                */
/*--------------------------------------------------------------------------*/
void interrupt task_switch(void)
{
  if(single_task)
    return;

  disable();
    tasks[tswitch].ss = _SS; /* salva o status atual do processo na pilha */
    tasks[tswitch].sp = _SP;

    /* Se um processo estava rodando quando interrompido, muda
     o seu status pra READY*/
    if(tasks[tswitch].status == RUNNING)
      tasks[tswitch].status = READY;

    /* Verfica se todos os processos estao mortos. Se assim for, para */
    if(all_dead())
      tasking = 0;

    /* Parou */
    if(!tasking)
    {
      disable();
     _SS = oldss;
     _SP = oldsp;
     setvect(8, old_int8);
     free_all();
      enable();
      return;
    }

    /*encontra um novo processo */
    tswitch++;
    if(tswitch == NUM_TASKS)
      tswitch = 0;
    while(tasks[tswitch].status != READY)
    {
      tswitch++;
      if(tswitch == NUM_TASKS)
    tswitch = 0;
    }
     _SS = tasks[tswitch].ss;     /* Muda para o novo processo */
     _SP = tasks[tswitch].sp;
    tasks[tswitch].status = RUNNING;     /* status muda pra RUNNING */
  enable();
}


/*--------------------------------------------------------------------------*/
/*   multitask()                                                            */
/*                                                                          */
/*   Inicia o kernel multitarefas                                    */
/*--------------------------------------------------------------------------*/
void interrupt multitask(void)
{
  disable();

    /* alterna o timer do escalonador */
    old_int8 = getvect(8);
    setvect(8, int8_task_switch);

    /*
    salva o SP (stack pointer) do programa para que, quando terminar,
    a execução continue de onde parou
    */
    oldss = _SS;
    oldsp = _SP;

    /*  seta a pilha para a primeira tarefa */
     _SS = tasks[tswitch].ss;
     _SP = tasks[tswitch].sp;
  enable();
}


/*--------------------------------------------------------------------------*/
/*   make_task()                                                            */
/*                                                                          */
/*   Retorna false se uma tarefa nao pode ser adicionada na fila.          */
/* De outra maneira, retorna true.                                          */
/*--------------------------------------------------------------------------*/
int make_task(taskptr task,
```

```
169             unsigned stck,
170             unsigned id)
171   {
172     struct int_regs *r;
173
174     if((id>=NUM_TASKS) || (id<0))
175       return 0;
176
177     disable();
178       /* Aloca espaco para a tarefa*/
179       tasks[id].stck = malloc(stck + sizeof(struct int_regs));
180       r = (struct int_regs *) tasks[id].stck + stck -  sizeof(struct int_regs);
181
182       /* inicializa a tareva */
183       tasks[id].sp = FP_OFF((struct int_regs far *) r);
184       tasks[id].ss = FP_SEG((struct int_regs far *) r);
185
186       /* seta os registradores CS e IP da nova tarefa */
187       r->cs = FP_SEG(task);
188       r->ip = FP_OFF(task);
189
190       /* seta os registratores  DS e ES */
191       r->ds = _DS;
192       r->es = _DS;
193
194       /* liga as interrupções */
195       r->flags = 0x200;
196
197       tasks[id].status = READY;
198     enable();
199     return 1;
200   }
201
202
203   /*--------------------------------------------------------------------------*/
204   /*   free_all()                                                             */
205   /*                                                                          */
206   /*   Libera todo espaco da pilha. Esta função nao deve ser chamada          */
207   /* pelo seu programa                                                        */
208   /*                                                                          */
209   /*--------------------------------------------------------------------------*/
210   void free_all(void)
211   {
212     register int i;
213
214     for(i=0; i<NUM_TASKS; i++)
215     {
216       if(tasks[i].stck)
217       {
218         free(tasks[i].stck);
219         tasks[i].stck = NULL;
220       }
221     }
222   }
223
224
225   /*--------------------------------------------------------------------------*/
226   /*  kill_task()                                                             */
227   /*                                                                          */
228   /*  Mata um processo/tarefa, o que muda seu status para DEAD              */
229   /*--------------------------------------------------------------------------*/
230   void kill_task(int id)
231   {
232     disable();
233       tasks[id].status = DEAD;
234       free(tasks[id].stck);
235       tasks[id].stck = NULL;
236     enable();
237     task_switch();
238   }
239
240   /*--------------------------------------------------------------------------*/
241   /*   init_tasks()                                                           */
242   /*                                                                          */
243   /*   Inicializa as estruturas de controle das tarefas                       */
244   /*--------------------------------------------------------------------------*/
245   void init_tasks(void)
246   {
247     register int i;
248
249     for(i=0; i<NUM_TASKS; i++)
250     {
251       tasks[i].status  = DEAD;
252       tasks[i].pending = NULL;
```

```
253       tasks[i].sleep    = 0;
254       tasks[i].stck     = NULL;
255     }
256     set_vid_mem();
257   }
258
259   /*---------------------------------------------------------------------------*/
260   /*   stop_tasking()                                                          */
261   /*                                                                           */
262   /*   Para tarefas                                                            */
263   /*---------------------------------------------------------------------------*/
264   void stop_tasking(void)
265   {
266     tasking = 0;
267     task_switch();
268   }
269
270   /*---------------------------------------------------------------------------*/
271   /*   mono_task()                                                             */
272   /*                                                                           */
273   /*  Executa apenas uma tarefa                                                */
274   /*---------------------------------------------------------------------------*/
275   void mono_task(void)
276   {
277     disable();
278       single_task = 1;
279     enable();
280   }
281
282   /*---------------------------------------------------------------------------*/
283   /*   resume_tasking()                                                        */
284   /*                                                                           */
285   /*   Retorna todas as tarefas em multitask (Usado para retornar uma tarefa */
286   /*   depois de chamar mono_task().                                           */
287   /*---------------------------------------------------------------------------*/
288   void resume_tasking(void)
289   {
290     single_task = 0;
291   }
292
293
294   /*---------------------------------------------------------------------------*/
295   /*   all_dead(void)                                                          */
296   /*                                                                           */
297   /*  retorna 1 se nenhuma tarefa possui status READ para iniciar.             */
298   /*  0, se pelo menos uma tarefa esta pronta para iniciar                     */
299   /*---------------------------------------------------------------------------*/
300   int all_dead(void)
301   {
302     register int i;
303
304     for(i=0; i<NUM_TASKS; i++)
305       if(tasks[i].status == READY)
306         return 0;
307     return 1;
308   }
309
310
311   /*---------------------------------------------------------------------------*/
312   /*   check_sleepers()                                                        */
313   /*                                                                           */
314   /*   Decrementa o sleep count                                                */
315   /*---------------------------------------------------------------------------*/
316   void check_sleepers(void)
317   {
318     register int i;
319
320     for(i=0; i<NUM_TASKS; i++)
321     {
322       if(tasks[i].status == SLEEPING)
323       {
324         tasks[i].sleep--;
325         if(!tasks[i].sleep)
326       tasks[i].status = READY;
327       }
328     }
329   }
330
331
332   /*---------------------------------------------------------------------------*/
333   /*   msleep()                                                                */
334   /*                                                                           */
335   /*   Para a execução de uma tarefa por um tempo determinado de ciclos de clock */
336   /*---------------------------------------------------------------------------*/
```

```
337  void msleep(int ticks)
338  {
339    disable();
340      tasks[tswitch].status = SLEEPING;
341      tasks[tswitch].sleep  = ticks;
342    enable();
343    task_switch();
344  }
345
346
347  /*---------------------------------------------------------------------------*/
348  /*   suspend()                                                               */
349  /*                                                                           */
350  /*   Suspende uma tarefa enquanto nao retomada por outra                     */
351  /*---------------------------------------------------------------------------*/
352  void suspend(int id)
353  {
354    if(id<0 || id>=NUM_TASKS)
355      return;
356    tasks[id].status = SUSPENDED;
357    task_switch();
358  }
359
360
361  /*---------------------------------------------------------------------------*/
362  /*   resume()                                                                */
363  /*                                                                           */
364  /*   Reinicia uma tarefa previamente suspensa                                    */
365  /*---------------------------------------------------------------------------*/
366  void resume(int id)
367  {
368    if(id<0 || id>=NUM_TASKS)
369      return;
370    tasks[id].status = READY;
371  }
372
373
374  /*---------------------------------------------------------------------------*/
375  /*   set_semaphore                                                           */
376  /*                                                                           */
377  /*  Espera por um semaforo                                                */
378  /*---------------------------------------------------------------------------*/
379  void set_semaphore(unsigned *sem)
380  {
381    disable();
382      while(*sem)
383      {
384        semblock(tswitch, sem);
385        task_switch();
386        disable(); /* trocar tarefas vai ligar as interrupções, entao precisam ser
387        desligadas novamente*/
388      }
389      *sem = 1;
390    enable();
391  }
392
393
394  /*---------------------------------------------------------------------------*/
395  /*   clear_semaphore                                                         */
396  /*                                                                           */
397  /*    Libera um semaforo                                                     */
398  /*---------------------------------------------------------------------------*/
399  void clear_semaphore(unsigned *sem)
400  {
401    disable();
402      tasks[tswitch].pending = NULL;
403      *sem = 0;
404      restart(sem);
405      task_switch();
406    enable();
407  }
408
409
410  /*---------------------------------------------------------------------------*/
411  /*   semblock()                                                              */
412  /*                                                                           */
413  /*   Seta uma tarefa para o status BLOCKED. Essa é uma função interna, não */
414  /* chamada pelo seu programa.                                                */
415  /*                                                                           */
416  /*---------------------------------------------------------------------------*/
417  void semblock(int id, unsigned *sem)
418  {
419    tasks[id].status  = BLOCKED;
420    tasks[id].pending = sem;
```

```
421   }
422
423
424   /*--------------------------------------------------------------------------*/
425   /*   restart()                                                              */
426   /*                                                                          */
427   /*   Reinicia uma tarefa que estava esperando por um semaforo especifico.   */
428   /* Essa é uma função interna não chamada pelo seu programa                  */
429   /*--------------------------------------------------------------------------*/
430   void restart(unsigned *sem)
431   {
432     register int i;
433
434     for(i=0; i<NUM_TASKS; i++)
435       if(tasks[i].pending == sem)
436       {
437         tasks[i].pending = NULL;
438         if(tasks[i].status == BLOCKED)
439       tasks[i].status = READY;
440         return;
441       }
442   }
443
444
445   /*--------------------------------------------------------------------------*/
446   /*   task_status()                                                          */
447   /*                                                                          */
448   /*   Mostra os status de todas as tarefas. Essa função NAO deve ser chamada */
449   /*enquanto esta no multitarefas.                                            */
450   /*--------------------------------------------------------------------------*/
451   void task_status(void)
452   {
453     register int i;
454
455     if(tasking)    /* Não pode ser usado em multitarefas */
456       return;
457
458     printf("\n");
459
460     for(i=0; i<NUM_TASKS; i++)
461     {
462       printf("Task %d: ",i);
463
464       switch(tasks[i].status)
465       {
466         case READY:
467       printf("READY\n"); /*Mostra READY */
468       break;
469         case RUNNING:
470       printf("RUNNING\n"); /* Mostra RUNNING*/
471       break;
472         case BLOCKED:
473       printf("BLOCKED\n"); /*mostra  BLOCKED*/
474       break;
475         case SUSPENDED:
476       printf("SUSPENDED\n"); /* mostra SUSPENDED*/
477       break;
478         case SLEEPING:
479       printf("SLEEPING\n"); /*mostra SLEEPING*/
480       break;
481         case DEAD:
482       printf("DEAD\n"); /*mostra DEAD*/
483       break;
484       }
485     }
486   }
487
488
489   /*--------------------------------------------------------------------------*/
490   /*                                                                          */
491   /* Funções de entrada e saída serializadas e reentrant (???) para multitarefas*/
492   /*                                                                          */
493   /*--------------------------------------------------------------------------*/
494
495   /*--------------------------------------------------------------------------*/
496   /* void mputs(char *s)                                                      */
497   /*                                                                          */
498   /*   versão serializada de puts()                                           */
499   /*--------------------------------------------------------------------------*/
500   void mputs(char *s)
501   {
502     set_semaphore(&io_out);
503     puts(s);
504     clear_semaphore(&io_out);
```

```
}


/*-------------------------------------------------------------------------*/
/*  Saida de um numero                                                     */
/*-------------------------------------------------------------------------*/
void mputnum(int num)
{
  set_semaphore(&io_out);
  printf("%d", num);
  clear_semaphore(&io_out);
}


/*-------------------------------------------------------------------------*/
/*  Versao serializada de  getche()                                        */
/*-------------------------------------------------------------------------*/
char mgetche(void)
{
  char ch;

  set_semaphore(&io_out);
  ch= getch();
  clear_semaphore(&io_out);
  return ch;
}


/*-------------------------------------------------------------------------*/
/* void mxyputs()                                                          */
/*                                                                         */
/*   mostra uma string especifica X, Y coordenadas. Essa função é reentrada       */
/* e pode ser chamada por qualquer tarefa em qualquer hora                 */
/*-------------------------------------------------------------------------*/
void mxyputs(int x, int y, char *str)
{
  while(*str)
    moutchar(x++,y,*str++);
}

/*-------------------------------------------------------------------------*/
/* void moutchar()                                                         */
/*                                                                         */
/*   Output a character at specified X, Y coordinates. This function is    */
/* reentrant and may be called by any task at any time.                    */
/*-------------------------------------------------------------------------*/
void moutchar(int x, int y, char ch)
{
  char far *v;

  v = vid_mem;

  v += (y*160) + x*2;    /* calcula o char loc */
  *v++ = ch;         /* escreve o caracter */
  *v = 7;        /* caracter normal */
}

/*-------------------------------------------------------------------------*/
/*  inicializa os sub-sistemas de video                                    */
/*-------------------------------------------------------------------------*/

/*-------------------------------------------------------------------------*/
/* void video_mode()                                                       */
/*                                                                         */
/*   retorna o modo de video atual                                         */
/*-------------------------------------------------------------------------*/
video_mode(void)
{
  union REGS r;

  r.h.ah = 15;  /* get modo de video */
  return int86( 0x10, &r, &r) & 255;
}

/*-------------------------------------------------------------------------*/
/* void set_vid_mem()                                                      */
/*                                                                         */
/* Seta o ponteiro vid_mem para o inicio da memoria de video               */
/*-------------------------------------------------------------------------*/
void set_vid_mem(void)
{
  int vmode;

  vmode= video_mode();
```

```
589     if((vmode!=2) && (vmode!=3) && (vmode!=7))
590     {
591       printf("/n Video must be in 80 column text mode"); /*Video deve estar no modo de texto de
    80 colunas*/
592       exit(1);
593     }
594
595     /* define o enderacamento do video RAM */
596     if(vmode==7)
597       vid_mem = (char far *) MK_FP(0xB000, 0);
598     else
599       vid_mem = (char far *) MK_FP(0xB800, 0);
600   }
601
```