UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"

Sistemas Operacionais II

Implementação de Sistema Operacional

Alexandre Salvador Fernandes
Mateus Gonçalez Etto
Wellington Carlos Massola

Bauru
2014

Alexandre Salvador Fernandes
Mateus Gonçalez Etto
Wellington Carlos Massola

Implementação de Sistema Operacional

Monografia apresentada como exigência para obtenção do grau de Bacharelado em Sistemas Operacionais II da UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO".

Orientador: Antonio Carlos Sementille

Bauru
2014

RESUMO

Este trabalho tem como objetivo a compreensão do funcionamento de um sistema operacional com núcleo multitarefas, para isso foi utilizado o DOS como base dentro de um emulador desse sistema, o DOSBox, e então foi criado um programa que funciona como suporte a multitarefas para o sistema original alterando-o e inserindo as funções necessárias como um escalador de processos, um sistema de semáforos, um sistema de troca de mensagens e outras funções.

Palavras-chave: Sistema Operacional, DOS

SUMÁRIO

1 INTRODUÇÃO

Este trabalho visou o entendimento e a implementação de um Núcleo Multitarefas. Para esta finalidade foi utilizado o DOSBox, que emulou uma máquina virtual, e nosso Sistema Operacional foi construído acima do DOS, pois é um sistema que permite ações a nível de máquina.

O trabalho se separou em 5 partes, sendo elas:

- Implementação do Núcleo Básico;
- Implementação de Semáforos no Núcleo Básico;
- Implementação de Troca de Mensagens no Núcleo Básico;
- Implementação de Prioridade Simples no Núcleo Básico;
- Implementação de Filas de Prioridade no Núcleo Básico.

Os detalhes de implementação e como o usuário deve usar o sistema será descrito nos próximos itens.

2 NÚCLEO BÁSICO

O núcleo básico foi construído como um sistema logicamente paralelo, uma vez que o processador emulado no DOSBox não admite um processamento fisicamente paralelo. Para esta finalidade, foi usado um modelo já existente que melhor se aproxima do que se queria: co-rotinas.

De forma resumida, nas co-rotinas é efetuado um "transfer" de uma rotina para outra, e então o contexto da primeira é salva, e então executa-se a segunda. Porém nesse modelo o programador deve chamar espontaneamente a função "transfer", cedendo o controle da UCP. Como a ideia é fazer isso sem que o programador sequer saiba da existência do paralelismo lógico do sistema, algo melhor deveria ser usado.

Então encontrou-se o "iotransfer", um comando como o "transfer", mas que não requer que a rotina declare espontaneamente sua vontade de ceder a UCP. No iotransfer, o transfer de volta é efetuado após uma interrupção, que no nosso caso é a interrupção do timer.

Ou seja, podemos escolher um tempo para o timer, chamar o iotransfer e então saberemos que o controle retornará depois do tempo setado no timer. Entretanto, apesar de essa ser a ideia do nosso Sistema Operacional, não podemos alterar indevidamente o clock do timer, uma vez que no contexto que estamos usando, este timer serve para dar o refresh da memória, e seu uso incorreto pode acarretar no travamento da máquina.

Com isso, temos agora a ideia da temporização do timer, apesar de limitado, e também o iotransfer. Apesar de o iotransfer fazer a troca de processos, ele não salva todo o contexto, então é importante lembrar que é necessário "sobrescrever" o código de interrupção, rodando um código que irá armazenar o contexto e depois chamar a rotina original da interrupção.

Foi em cima deste contexto que foi feita a implementação de nosso núcleo básico.

O código do núcleo básico pode ser visto no Anexo A.

2.1 Detalhes de Implementação

O primeiro passo da implementação foi a criação do tipo BCP, que armazena todas as informações do que chamamos de processo. Tem os campos: nome, estado, contexto e ponteiro para o próximo BCP.
Com este tipo BCP, foi criado uma função Criar_Processo, que será chamado pelo usuário e tem a finalidade de preencher a lista de BCPs. Também foi criado uma função chamada Volta_Dos, que será chamado pelo próprio sistema, e visa retomar a interrupção anterior (que roda apenas o refresh da memória) antes de retornar de fato o comando para o DOS.
Já na rotina Escalador, a ideia foi implementar o Round Robin como sendo o algoritmo que seleciona o próximo processo a rodar. A ideia é a seguinte: seleciona-se o primeiro processo que está disponível para rodar, e faz-se um iotransfer para ele. Desta forma, o processo irá rodar até a próxima interrupção do timer. Havendo a interrupção, o escalonador retoma o controle e procura o próximo ativo (irá percorrer uma lista circular e pegar o primeiro com estado ativo). Durante o processo de procura do próximo as interrupções são desabilitadas, pois não se deseja que haja qualquer "troca de processo" enquanto nem tem processo rodando, mas sim o próprio sistema operacional.
No entanto, existe a possibilidade de um programa do usuário chamar uma rotina do DOS, que use entrada/saída, por exemplo. Isto acontece porque não estamos escrevendo os drivers, e sim usando os drivers que já vem prontos do DOS. E enquanto está em uma rotina do DOS, não deseja-se interromper sua execução (pode estar escrevendo no disco, por exemplo, e se interromper vai dar problema, pois os drivers do DOS não foram feitos prevendo vários acessos a um mesmo driver, uma vez que o DOS é monoprogramado). Logo, quando houver uma interrupção, antes de efetuar a troca verifica-se se o DOS está usando sua pilha (podemos ver isso através de uma variável escondida). Se está usando a pilha, não podemos interrompê-lo. Ou seja, não se troca o processo e se dá outra fatia de tempo a ele.
O Escalador ficará rodando até que todos os programas de usuário terminem, e então chamará a função de retornar ao DOS.
Houve também a implementação do Dispara_Sistema, que tem como finalidade iniciar o sistema, ou seja, chama pela primeira vez o Escalonador, que fará a "troca" do algoritmo de interrupção (agora irá salvar todos os dados do contexto, além de fazer o refresh da memória) e rodar o Round Robin.
Por último, foi implementado o Terminar_Processo, que tem como finalidade alterar logicamente o estado do processo, colocando em terminado, e terminar de gastar a fatia de tempo dada ao processo.

2.2 Guia do Usuário

O usuário deverá seguir os seguintes passos para o uso deste Sistema Operacional:

Em seu código, a primeira coisa a fazer será incluir o nucleo.h, que tem os cabeçalhos das funções que poderá usar.

```
#include <nucleo.h>
```

A partir disso, crie várias funções, sendo que cada uma será o seu processo. Por exemplo, se deseja criar o processo chamado Fibonacci, escreva:

```
void far Fibonacci()
{ /* Seu código aqui. */ }
```

No fim de cada processo, chame a função Terminar_Processo(), uma função do sistema. Esta função providenciará o término do processo e irá parar de dar fatias de tempo para ele rodar.

Terminado de escrever seus processos, escreva em seu main as chamadas de seus processos. Ou seja, avisar o sistema que elas existem. Isto é feito através da chamada da seguinte função:

```
Criar_Processo(nome, processo);
```

No lugar do nome, digite o número que deseja associar a este processo. Este número é da sua escolha. No segundo parâmetro, digite o nome da função que deseja que seja o novo processo. Este nome deve ser de uma das funções que você criou anteriormente. Esta chamada pode ser, neste caso:

```
Criar_Processo(20, Fibonacci);
```

Após adicionar todos os processos desejados, chame a função Dispara_Sistema(). Esta função do sistema irá fazer com que seus processos sejam rodados de forma logicamente paralelos, através do algoritmo do sistema, o Round Robin.

Pronto, são estes os passos que devem ser seguidos para o uso deste Sistema Operacional.

3 SEMÁFOROS

Os semáforos tem como função o controle do acesso a recursos compartilhados. Podendo fazer, por exemplo a função de mutex, quando existe a necessidade de apenas um recurso ser executado de cada vez.

3.1 Detalhes de Implementação

Para a implementação dos semáforos no núcleo, foram necessárias as seguintes modificações:

Declaração de uma *fila_semaforo dentro da struct Desc_Prog, que nada mais é do que a fila de processos bloqueados pelo semáforo.

Criação de uma struct sinaleira que possui um int s, que é o valor do semáforo, atribuído ao inicializar (escolhido pelo usuário), e um DESCRITOR_PROC Q, que irá receber um processo bloqueado.

Inclusão de um novo estado, bloq_P, que simbolizará o estado de “bloqueado por semáforo”.

Criação de uma função inicia_semaforo(semaforo *sem, int n) responsável pela inicialização dos semáforos, acessível ao usuário.

Criação das primitivas P(semaforo *sem) e V(semáforo *sem), sendo a P aquela que irá bloquear o processo atual e coloca-lo na fila caso o valor do semáforo seja zero ou subtrair em uma unidade caso contrário; e a primitiva V aquela que irá desbloquear o primeiro processo na fila de bloqueados caso a mesma não esteja vazia; do contrário irá apenas incrementar o valor do semáforo.

O código do núcleo básico com acréscimo dos semáforos pode ser visto no Anexo B.

3.2 Guia do Usuário

Em adição ao que já foi apresentado no núcleo básico, o usuário poderá utilizar 3

novas funções: inicia_semaforo, P e V. Para que qualquer uma destas três funções possam ser usufruídas, haverá o novo tipo semaforo, que é uma strutura com dois valores: um inteiro que identifica qual a utilidade do semáforo e um ponteiro para o primeiro processo bloqueado pelo semáforo. As três funções são descritas a seguir:

Inicia_semáforo(semáforo sem, int x): deve ser utilizada para qualquer semáforo que o usuário venha a criar, sendo que o valor de x irá determinar a função do semáforo; se o intuito for usar um mutex, x deve ser igual a 1; se for para trabalhar com um buffer, deverá conter o valor máximo deste; no caso de um programa produtor/consumidor, recomenda-se que o usuário utilize um semáforo inicializando com 0.

Primitiva P(semáforo x): esta função irá decrementar o valor da variável do semáforo caso esta seja maior do que zero. Caso contrário, irá bloquear o processo atual e ativar o próximo disponível.

Primitiva V(semáforo x): esta função irá incrementar o valor da variável do semáforo caso a fila de processos bloqueados for nula. Se houver algum processo bloqueado na fila de processos bloqueados, o primeiro processo desta será desbloqueado.

As primitivas P e V devem ser utilizadas antes e depois, respectivamente, do processo concorrente para que o semáforo funcione corretamente.

4 TROCA DE MENSAGENS

A troca de mensagens é um mecanismo de comunicação e sincronização entre processos que independe de uma área de memória compartilhada.

4.1 Detalhes de Implementação

O inicio da implementação da troca de mensagens se deu pela criação do tipo mensagem, o qual armazena 4 campos: *flag*, *nome_emissor*, *mensa* e *prox* que representam, respectivamente, se há conteúdo ou não na mensagem, o nome do emissor da mensagem, uma cadeia de caracteres capaz de armazenar 25 caracteres contando o caractere vazio para representar o fim da "string" e um campo prox que é usado para apontar para a próxima mensagem na lista de mensagens (estrutura inserida no BCP).

A estrutura do BCP também recebeu alterações com a inserção dos campos *ptr_msg*, *tam_fila* e *qtde_msg_fila*, que representam, respectivamente, o ponteiro para uma estrutura do tipo lista com mensagens, quantidade máxima de elementos na lista e quantidade atual de elementos na lista. Além disso foram adicionados os estados "BLOQREC" e "BLOQENV".

Após a criação e alteração das estruturas foram iniciadas as implementações das funções que as utilizarão, iniciando pela função Cria_Fila_Mensagens que recebe um valor *max_fila* indicando a quantidade máxima de elementos e retorna um ponteiro para uma lista encadeada com um número de mensagens igual o indicado por *max_fila*.

A função Cria_Fila_Mensagens foi utilizada pela função Criar_Processo que foi alterada para receber um valor a mais que é o tamanho máximo da fila de mensagens e foi alterada a inicialização dos valores do BCP utilizando a função criada anteriormente.

A próxima função a ser implementada foi a Envia que verifica se o destino da mensagem existe e se a sua fila não está cheia, caso isso seja verdade ele desabilita as interrupções, se o processo receptor estiver como "BLOQREC" ele

recebe "ATIVO", depois procura o primeiro elemento vazio na lista encadeada de mensagens do receptor, marca esse elemento como cheio e copia as informações da mensagem para ele, após isso ele bloqueia o processo que enviou a mensagem com o estado "BLOQENV" retorna o valor 2 que representa sucesso, mas caso as informações verificadas no inicio forem falsas ele retorna 0 ou 1 que significa fracasso no envio.

Por fim a função Recebe foi implementada, ele verifica se a lista de mensagens do processo receptor está vazia, se estiver ele se bloqueia até que um processo envie uma mensagem e o desbloqueie, caso contrário ele decrementa o contador de mensagens pega uma mensagem da fila e retorna na forma de parâmetros que ele possui.

O código do núcleo básico com acréscimo da troca de mensagens pode ser visto no Anexo C.

4.2 Guia do Usuário

Uma modificação em relação ao Núcleo Básico é a função Criar_Processo, que terá 3 parâmetros ao invés de 2. Deve ser escrito da seguinte forma:

Criar_Processo(nome, processo, max_fila);

Onde max_fila representa o tamanho máximo da fila de mensagens que o processo pode receber, ele deve ser um valor positivo.

Para utilização das funções envia e recebe ele deve usá-las da seguinte forma:

Envia(nome_destino, msg);

Recebe(nome_emissor, msg);

nome_destino: é o número referente ao processo ao qual se deseja enviar a mensagem.

nome_emissor: é um parâmetro inteiro passado por referência em Recebe para identificar o número do processo que enviou a mensagem.

msg: é um ponteiro para um vetor de char com 25 elementos que serão enviados ou recebidos.

5 PRIORIDADE SIMPLES

A prioridade simples é a forma mais simples (como o próprio nome diz) de se implementar um sistema de prioridades. A ideia dessa alteração no núcleo básico é, na função de criar processo, também pedir ao usuário um valor que significara a prioridade. Este valor de prioridade irá simbolizar o número de fatias de tempo que o processo irá receber pelo escalonador.

5.1 Detalhes de Implementação

A implementação da Prioridade Simples se deu na adição de 2 campos a mais no BCP: Um para a prioridade e outro como contador de fatias de tempo usadas.

Como agora há mais dados a serem preenchidos, a função Criar Processo agora tem que receber um valor a mais do usuário, que é a prioridade. Inicia-se o valor de prioridade com o dado enviado pelo usuário e o com o contador zerado.

Agora a última alteração feita está no Escalador: toda vez que o processo terminar de rodar sua fatia de tempo, seu contador de fatias é incrementado. Se este valor for igual a sua prioridade, roda-se o algoritmo de troca de processos (o Round Robin). Imediatamente antes de trocar, seu contador é "zerado".

Nesta implementação, o "zerar" não necessariamente deixa o contador com zero. Acontece que existe aquela situação em que o processo é interrompido durante uma chamada ao DOS, e recebe uma fatia de tempo mesmo assim. Nessas situações, o contador é incrementado da mesma maneira, porém não é verificado se ele atingiu o valor da prioridade, pois vai rodar de novo de qualquer maneira. Caso isto aconteça bem no momento que ele devesse sair, o algoritmo até permite rodar de novo e incrementa seu contador, mas depois, ao invés de zerar seu contador, ele apenas decrementa pelo valor da prioridade. Desta forma, o escalonador está apenas dando "créditos" ao processo, ao invés de zerar sua variável cegamente.

Por exemplo, se determinado processo tem prioridade 3, e se na 3ª interrupção foi enquanto estava no DOS, seu contador subirá para 3 e não será verificado se o valor é igual à prioridade. Quando terminar de rodar sua 4ª fatia de tempo, e

considerando não foi interrompido no DOS, seu contador será incrementado para 4 (pois rodou mais uma vez), este valor será comparado com o da prioridade que é 3, o SO verá que já passou do valor, e então decrementa o contador em 3 unidades: 4 - 3 = 1. O valor 1 ficará em seu contador, demonstrando que na próxima vez que chegar nele, a fatia extra de tempo extra que ele recebeu já estará sendo considerado como usado.

O código do núcleo básico com acréscimo de prioridade simples pode ser visto no Anexo D.

5.2 Guia do Usuário

A forma de utilizar o Sistema Operacional implementado aqui é bastante semelhante ao do Núcleo Básico. Para rever como utilizar o núcleo básico, veja o item 2.2.

A única diferença na Prioridade simples é a forma de se criar processos. No núcleo simples deve ser chamado a função da seguinte maneira:

Criar_Processo(nome, processo);

Porém, agora deve ser chamado da seguinte forma:

Criar_Processo(nome, processo, prioridade);

Esta é a única modificação. Prioridade deve ser um valor de 1 a 10, sendo 1 menor prioridade e 10 maior prioridade. A respeito de como usar o SO como um todo, consulte o item 2.2.

6 FILAS DE PRIORIDADE

A ideia de prioridade "real" é o de Filas de Prioridade, no qual os processos que vão rodar primeiro são os que tem maior prioridade.

Ou seja, nesta forma de dar prioridade aos processos, os processos que tiverem maior prioridade vão rodar muito mais vezes e primeiro em relação aos de menor prioridade. Vale afirmar que neste modelo, apesar de processos com menor prioridade rodarem menos, eles vão rodar sim, mesmo que ainda tenha aqueles com maior prioridade, apesar de ser por pouco tempo e em intervalos grandes.

O código do núcleo básico com acréscimo das filas de prioridades pode ser visto no Anexo E.

6.1 Detalhes de Implementação

Em relação ao Núcleo Básico, a alteração no BCP se dá apenas em mais um campo chamado 'prioridade', que pode ser um valor de 0 a 9, sendo 9 a maior prioridade. Não há contador de fatias de tempo.
Ou seja, a primeira alteração se dá na função de Criar_Processo, no qual se deve validar o valor de prioridade enviado pelo usuário e colocá-lo na variável do BCP.
Diferente do Prioridade Simples, aqui não há alteração na função do Escalonador.
A primeira mudança significativa está na adição de um novo procedimento no núcleo: Procura_Primeiro_Ativo(). Como agora não temos como saber quem será o primeiro a rodar pela ordem dos Criar_Processo, deve-se chamar uma função que procura manualmente qual é o processo com maior prioridade adicionado.
Ou seja, quando o usuário chama a função Dispara_Sistema, esta função deve chamar o Procura_Primeiro_Ativo, a fim de procurar o processo que rodará primeiro, e só então chamar o Escalador.
A maior edição no Núcleo Básico se dá na função Procura_Prox_Ativo, que é chamado pelo Escalonador toda vez que ele deve trocar de processo. Aqui foi criado um vetor com 10 filas, uma para cada prioridade. O preenchimento correto dessas filas se dá no procedimento Criar_Processo, que através da prioridade recebida pelo usuário, ele anexa o BCP do processo na fila correspondente. A nova lógica seguida pelo Procura_Prox_Ativo é a seguinte: roda-se a fila de maior prioridade, dando uma fatia de tempo para cada processo. Terminado isso, o próximo passo é rodar a fila de maior prioridade novamente e depois rodar a de segunda maior prioridade. E depois repete o processo considerando o de terceira maior prioridade também. Tudo isso se repete até que todas as 10 prioridades sejam consideradas, e quando terminar começa desde o início novamente.
Este algoritmo considera a maior prioridade como algo que deve rodar sempre antes, e por mais tempo, mas também não desconsidera totalmente o processo com

menor prioridade, dando a ele ao menos uma fatia de tempo de vez em quando.

6.2 Guia do Usuário

Aqui a única modificação em relação ao Núcleo Básico é a função Criar_Processo, que terá 3 parâmetros ao invés de 2. Deve ser escrito da seguinte forma: Criar_Processo(nome, processo, prioridade);

O valor de prioridade deve ser um número de 0 a 9, sendo 9 o de maior prioridade. Os outros passos de como criar seus processos e rodá-los é análogo ao do item 2.2.

7 CONCLUSÃO

Ao término deste trabalho obtivemos um Sistema Operacional simples com suporte a multitarefas e foi possível a compreensão do funcionamento de um núcleo de sistema como o do DOS. Esse era o objetivo inicial e que foi alcançado a partir da implementação de parte da base do sistema multiprogramado e algumas funcionalidades importantes como o sistema de semáforos, a troca de mensagens, sistema de prioridades e as filas de prioridades.

Com todas etapas feitas foi possível concluir que um sistema operacional é composto por suas diversas partes que dependem umas das outras de forma a ter-se uma hierarquia, onde as partes que dependem mais do hardware devem tem uma importância maior e deve ser utilizadas pelo usuário através de partes mais distantes tornando o sistema mais seguro e estável, além de gerar um nível de abstração maior em relação ao que acontece ao hardware.

8 ANEXOS

8.1 ANEXO A – NÚCLEO BÁSICO

```
#include <system.h>
#include <stdio.h>

typedef struct registros
{
    unsigned bx1,es1;
} regis;

typedef union k
{
    regis x;
    char far *y;
} APONTA_REG_CRIT;

APONTA_REG_CRIT a;

typedef struct Desc_Prog *DESCRITOR_PROC;
struct Desc_Prog /* DEFINICAO DO TIPO BCP */
{
    int nome, estado;
    PTR_DESC contexto;
    DESCRITOR_PROC prox_desc;
};

/* VARIAVEIS GLOBAIS */
enum estado {ATIVO, TERMINADO};
PTR_DESC d_esc;
DESCRITOR_PROC PRIM, fim_lista = 0;

/* FUNCOES DO NUCLEO */
void far Criar_Processo(int numero, void far (*end_processo)())
{
    DESCRITOR_PROC p_aux;
    p_aux = (DESCRITOR_PROC) malloc (sizeof (struct Desc_Prog));
    p_aux->nome = numero;
    p_aux->estado = ATIVO;
    p_aux->contexto = cria_desc();
    newprocess(end_processo, p_aux->contexto);
    if (PRIM != 0) /* SE NAO FOR O PRIMEIRO PROCESSO */
    {
        p_aux->prox_desc = fim_lista->prox_desc;
        fim_lista->prox_desc = p_aux;
        fim_lista = p_aux;
    }
    else /* SE FOR O PRIMEIRO PROCESSO */
    {
        PRIM = p_aux;
        p_aux->prox_desc = p_aux;
        fim_lista = p_aux;
    }
}

void far Volta_Dos()
```

```
54  {
55      disable();
56      setvect(8, p_est->int_anterior);
57      enable();
58      exit(0);
59  }
60
61  DESCRITOR_PROC far Procura_Prox_Ativo()
62  {
63      DESCRITOR_PROC aux;
64      aux = PRIM->prox_desc;
65      do
66      {
67          if (aux->estado == ATIVO)
68              return aux;
69          aux = aux->prox_desc;
70      } while (aux != PRIM->prox_desc);
71      return 0;
72  }
73
74  void far Escalador()
75  {
76      p_est->p_origem = d_esc;
77      p_est->p_destino = PRIM->contexto;
78      p_est->num_vetor = 8;
79      /* Iniciando ponteiro para a Pilha do DOS */
80      _AH=0x34;
81      _AL=0x00;
82      geninterrupt(0x21);
83      a.x.bx1=_BX;
84      a.x.es1=_ES;
85      while(1)
86      {
87          iotransfer();
88          disable();
89          /* Houve Interrupcao durante um trecho do DOS? */
90          if (!*a.y)
91          {
92              /* Se entrou aqui, nao estava em um trecho do DOS! */
93              if ((PRIM = Procura_Prox_Ativo()) == NULL)
94                  Volta_Dos();
95              p_est->p_destino = PRIM->contexto;
96          }
97          enable();
98      }
99  }
100
101 void far Dispara_Sistema()
102 {
103     PTR_DESC d_inicio;
104     d_esc = cria_desc();
105     newprocess(Escalador, d_esc);
106     d_inicio = cria_desc();
107     transfer(d_inicio, d_esc);
108 }
109
110 void far Terminar_Processo()
111 {
112     disable();
113     PRIM->estado = TERMINADO;
114     enable();
```

```
115      while(1);
116  }
```

8.2 ANEXO B – NUCLEO BASICO + SEMAFOROS

```
  1  #include <system.h>
  2  #include <stdio.h>
  3
  4  #define max_fila 10
  5
  6  typedef struct registros
  7  {
  8      unsigned bx1,es1;
  9  } regis;
 10
 11  typedef union k
 12  {
 13      regis x;
 14      char far *y;
 15  } APONTA_REG_CRIT;
 16
 17  APONTA_REG_CRIT a;
 18
 19  typedef struct Desc_Prog *DESCRITOR_PROC;
 20  struct Desc_Prog /* DEFINICAO DO TIPO BCP */
 21  {
 22      int nome, estado;
 23      PTR_DESC contexto;
 24      DESCRITOR_PROC prox_desc;
 25
 26      struct Desc_Prog *prox_fila_semaforo; /*fila de processos
bloqueados*/
 27  };
 28
 29  typedef struct sinaleira semaforo;
 30  struct sinaleira
 31  {
 32      int s;
 33      DESCRITOR_PROC Q;
 34  };
 35
 36  /* VARIAVEIS GLOBAIS */
 37  enum estado {ATIVO, bloq_P, TERMINADO};
 38  PTR_DESC d_esc;
 39  DESCRITOR_PROC PRIM, fim_lista=0;
 40
 41  DESCRITOR_PROC far Procura_Prox_Ativo();
 42
 43  void far inicia_semaforo(semaforo *sem, int n)
 44  {
 45      sem->s = n;
 46      sem->Q = NULL;
 47  }
 48  /* FUNCOES DO NUCLEO */
 49  /*BEGINsemaforo*/
 50  void far P(semaforo *sem)
```

```
 51  {
 52      DESCRITOR_PROC p_aux,p1,aux;
 53      disable();
 54
 55      aux = sem->Q; /*VARIAVEL AUXILIAR PEGA O CABEÃ‡A DE FILA*/
 56
 57      if (sem->s > 0)
 58      {
 59          sem->s--;
 60          enable();
 61      }
 62      else
 63      {
 64          while (1)
 65          {
 66              if (sem->Q == NULL) /*SE FOR NULL, ESTÃ• VAZIO. PRIM Ã‰
AGORA O PRIMEIRO DA FILA*/
 67              {
 68                  sem->Q = PRIM;
 69                  break;
 70              }
 71              else if (aux->prox_fila_semaforo == NULL) /*SE O PROXIMO
DA FILA FOR NULL, Ã‰ O FINAL DA FILA*/
 72                              {
 73                  aux->prox_fila_semaforo = PRIM;
 74                  break;
 75              }
 76              aux = aux->prox_fila_semaforo; /*AVANÃ‡A PARA O PRÃ“XIMO
BLOQUEADO*/
 77          }
 78
 79          PRIM->estado = bloq_P; /*COLOCA O ESTADO COMO BLOQUEADO*/
 80          p_aux = PRIM;
 81          p1 = PRIM;
 82          PRIM = Procura_Prox_Ativo();
 83          transfer(p1->contexto, PRIM->contexto);
 84      }
 85  }
 86  void far V(semaforo *sem)
 87  {
 88      DESCRITOR_PROC aux;
 89
 90      disable();
 91      aux = sem->Q; /*VARIAVEL AUXILIAR PEGA O CABEÃ‡A (PRIMEIRO) DA
FILA*/
 92
 93      if (sem->Q == NULL)
 94          sem->s++;
 95      else
 96      {
 97          PRIM = sem->Q; /* RETIRA PRIMEIRO PROCESSO NA FILA SEM->Q*/
 98          sem->Q = aux->prox_fila_semaforo; /*MOVE O SEGUNDO PROCESSO
BLOQUEADO PARA CABEÃ‡A*/
 99          PRIM->estado = ATIVO;
100      }
101      enable();
102
103  }
104  /*ENDsemaforo*/
105  void far Criar_Processo(int numero, void far (*end_processo)())
106  {
```

```
107     DESCRITOR_PROC p_aux;
108     p_aux = (DESCRITOR_PROC) malloc (sizeof (struct Desc_Prog));
109     p_aux->nome = numero;
110     p_aux->estado = ATIVO;
111     p_aux->contexto = cria_desc();
112     newprocess(end_processo, p_aux->contexto);
113     if (PRIM != 0) /* SE NAO FOR O PRIMEIRO PROCESSO */
114     {
115         p_aux->prox_desc = fim_lista->prox_desc;
116         fim_lista->prox_desc = p_aux;
117         fim_lista = p_aux;
118     }
119     else /* SE FOR O PRIMEIRO PROCESSO */
120     {
121         PRIM = p_aux;
122         p_aux->prox_desc = p_aux;
123         fim_lista = p_aux;
124     }
125 }
126
127 void far Volta_Dos()
128 {
129     disable();
130     setvect(8, p_est->int_anterior);
131     enable();
132     exit(0);
133 }
134
135 DESCRITOR_PROC far Procura_Prox_Ativo()
136 {
137     DESCRITOR_PROC aux;
138     aux = PRIM->prox_desc;
139     do
140     {
141         if (aux->estado == ATIVO)
142             return aux;
143         aux = aux->prox_desc;
144     } while (aux != PRIM->prox_desc);
145     return 0;
146 }
147
148 void far Escalador()
149 {
150     p_est->p_origem = d_esc;
151     p_est->p_destino = PRIM->contexto;
152     p_est->num_vetor = 8;
153     /* Iniciando ponteiro para a Pilha do DOS */
154     _AH=0x34;
155     _AL=0x00;
156     geninterrupt(0x21);
157     a.x.bx1=_BX;
158     a.x.es1=_ES;
159     while(1)
160     {
161         iotransfer();
162         disable();
163         /* Houve Interrupcao durante um trecho do DOS? */
164         if (!*a.y)
165         {
166             /* Se entrou aqui, nao estava em um trecho do DOS! */
167             if ((PRIM = Procura_Prox_Ativo()) == NULL)
```

```
168                 Volta_Dos();
169             p_est->p_destino = PRIM->contexto;
170         }
171         enable();
172     }
173 }
174
175 void far Dispara_Sistema()
176 {
177     PTR_DESC d_inicio;
178     d_esc = cria_desc();
179     newprocess(Escalador, d_esc);
180     d_inicio = cria_desc();
181     transfer(d_inicio, d_esc);
182 }
183
184 void far Terminar_Processo()
185 {
186     disable();
187     PRIM->estado = TERMINADO;
188     enable();
189     while(1);
190 }
191
192
```

8.2 ANEXO C – NUCLEO BASICO + TROCA DE MENSAGENS

```
 1 #include <system.h>
 2 #include <stdio.h>
 3
 4 typedef struct registros
 5 {
 6     unsigned bx1,es1;
 7 } regis;
 8
 9 typedef union k
10 {
11     regis x;
12     char far *y;
13 } APONTA_REG_CRIT;
14
15 APONTA_REG_CRIT a;
16
17 typedef struct address mensagem;
18 typedef mensagem *PTR_MENSAGEM;
19 struct address
20 {
21     int flag;
22     int nome_emissor;
23     char mensa[25];
24     struct address *prox;
25 };
26
27 typedef struct Desc_Prog *DESCRITOR_PROC;
28 struct Desc_Prog /* DEFINICAO DO TIPO BCP */
```

```
 29  {
 30      int nome, estado;
 31      PTR_DESC contexto;
 32
 33      /* ------------------------ INICIO ALTERAÃ‡Ã•ES ------------------
------ */
 34      PTR_MENSAGEM ptr_msg;    /*Ponteiro para a lista encadeada de
mensagens*/
 35      int tam_fila;            /*Tamanho da lista de mensagens*/
 36      int qtde_msg_fila;       /*Quantidade de mensagens atualmente na
lisa (deve
 37                               ser inicializado com 0)*/
 38      /* ------------------------ FIM ALTERAÃ‡Ã•ES -------------------
------ */
 39
 40
 41      DESCRITOR_PROC prox_desc;
 42  };
 43
 44  /* VARIAVEIS GLOBAIS */
 45  enum estado {ATIVO, BLOQREC, BLOQENV, TERMINADO};
 46  PTR_DESC d_esc;
 47  DESCRITOR_PROC PRIM, fim_lista = 0;
 48
 49
 50
 51  /* ------------------------ INICIO ALTERAÃ‡Ã•ES ----------------------
-- */
 52  PTR_MENSAGEM far Cria_Fila_Mensagens(int max_fila)
 53  /*  FunÃ§Ã£o para retornar um ponteiro de uma lista simplesmente
encadeada
 54      de "max_fila" elementos do tipo "mensagem", tendo estes o campo
flag
 55      inicializado com o valor 0
 56  */
 57  {
 58      PTR_MENSAGEM aux1,aux2,fila;
 59      int i;
 60
 61      fila = aux1 = (PTR_MENSAGEM) malloc(sizeof(mensagem));
 62      aux1->flag = 0;
 63      aux1->prox = NULL;
 64      aux2 = aux1;
 65      for(i = 1; i < max_fila; i++)
 66      {
 67          aux1 = (PTR_MENSAGEM) malloc(sizeof(mensagem));
 68          aux2->prox = aux1;
 69          aux1->flag = 0;
 70          aux1->prox = NULL;
 71          aux2 = aux1;
 72      }
 73
 74      return fila;
 75  }
 76  /* ------------------------ FIM ALTERAÃ‡Ã•ES ------------------------
-- */
 77
 78
 79
 80
 81  /* FUNCOES DO NUCLEO */
```

```
 82  void far Criar_Processo(int numero, void far (*end_processo)(), int
max_fila)
 83  {
 84      DESCRITOR_PROC p_aux;
 85      p_aux = (DESCRITOR_PROC) malloc (sizeof (struct Desc_Prog));
 86      p_aux->nome = numero;
 87      p_aux->estado = ATIVO;
 88      p_aux->contexto = cria_desc();
 89
 90
 91      /* ------------------------ INICIO ALTERAÃ‡Ã•ES ------------------
------ */
 92      /*inicializaÃ§Ã£o dos campos novos da struct Desc_Prog*/
 93
 94      /*inicializando variÃ¡veis*/
 95      p_aux->tam_fila = max_fila;
 96      p_aux->qtde_msg_fila = 0;
 97
 98      /*funÃ§Ã£o para criar uma lista de mensagens e retornar um
ponteiro para ela*/
 99      p_aux->ptr_msg = Cria_Fila_Mensagens(max_fila);
100      /* ------------------------- FIM ALTERAÃ‡Ã•ES --------------------
------ */
101
102
103      newprocess(end_processo, p_aux->contexto);
104      if (PRIM != 0) /* SE NAO FOR O PRIMEIRO PROCESSO */
105      {
106          p_aux->prox_desc = fim_lista->prox_desc;
107          fim_lista->prox_desc = p_aux;
108          fim_lista = p_aux;
109      }
110      else /* SE FOR O PRIMEIRO PROCESSO */
111      {
112          PRIM = p_aux;
113          p_aux->prox_desc = p_aux;
114          fim_lista = p_aux;
115      }
116  }
117
118  void far Volta_Dos()
119  {
120      disable();
121      setvect(8, p_est->int_anterior);
122      enable();
123      exit(0);
124  }
125
126  DESCRITOR_PROC far Procura_Prox_Ativo()
127  {
128      DESCRITOR_PROC aux;
129      aux = PRIM->prox_desc;
130      do
131      {
132          if (aux->estado == ATIVO)
133              return aux;
134          aux = aux->prox_desc;
135      } while (aux != PRIM->prox_desc);
136      return 0;
137  }
138
```

```
139  void far Escalador()
140  {
141      p_est->p_origem = d_esc;
142      p_est->p_destino = PRIM->contexto;
143      p_est->num_vetor = 8;
144      /* Iniciando ponteiro para a Pilha do DOS */
145      _AH=0x34;
146      _AL=0x00;
147      geninterrupt(0x21);
148      a.x.bx1=_BX;
149      a.x.es1=_ES;
150      while(1)
151      {
152          iotransfer();
153          disable();
154          /* Houve Interrupcao durante um trecho do DOS? */
155          if (!*a.y)
156          {
157              /* Se entrou aqui, nao estava em um trecho do DOS! */
158              if ((PRIM = Procura_Prox_Ativo()) == NULL)
159                  Volta_Dos();
160              p_est->p_destino = PRIM->contexto;
161          }
162          enable();
163      }
164  }
165  /* ------------------------ INICIO ALTERAÃ‡Ã•ES ----------------------
-- */
166  /* TROCA DE MENSAGEMS */
167  int far Envia(int nome_destino,char *p_info)
168  {
169      DESCRITOR_PROC p_aux, p1;
170      PTR_MENSAGEM msg;
171
172      p_aux = PRIM;
173
174      /*procura descritor do destino da mensagem na fila dos prontos; */
175      while(p_aux->nome != nome_destino)
176      {
177          p_aux = p_aux->prox_desc;
178          /* fracasso: nÃ£o achou destino */
179          if(p_aux == PRIM)
180              return 0;
181      }
182
183      if(p_aux->tam_fila < p_aux->qtde_msg_fila)
184          return 1;   /* fracasso: fila cheia*/
185
186      disable();
187
188      /*localiza uma mensagem vazia (flag==0);*/
189      msg = p_aux->ptr_msg;
190      while(msg->flag != 0)
191          msg = msg->prox;
192
193      /*completa a mensagem: */
194      msg->flag = 1;
195      msg->nome_emissor = PRIM->nome;
196      strcpy(msg->mensa,p_info);
197      (p_aux->qtde_msg_fila)++;
198
```

```
199      /*Se o destino estiver como BLOQREC ele Ã© alterado para ATIVO*/
200      if(p_aux->estado == BLOQREC)
201          p_aux->estado = ATIVO;
202
203      /*Bloqueia o processo atual como BLOQENV*/
204      PRIM->estado = BLOQENV;
205
206      /*Passa o controle para o prÃ³ximo processo ativo*/
207      p_aux = Procura_Prox_Ativo();
208      p1 = PRIM;
209      PRIM = p_aux;
210      transfer(p1->contexto,PRIM->contexto);
211
212      return 2;   /*Suceesso no envio*/
213  }
214
215  void far Recebe(int *nome_emissor,char *msg){
216      DESCRITOR_PROC p_aux,p1;
217      PTR_MENSAGEM p_msg;
218      disable();
219
220      /*Se nÃ£o houver mensagens na fila do processo, ele se
221      autobloqueia com BLOQREC atÃ© receber uma mensagem*/
222      if(PRIM->qtde_msg_fila == 0){
223          PRIM->estado = BLOQREC;
224          p_aux = Procura_Prox_Ativo();
225          p1 = PRIM;
226          PRIM = p_aux;
227          transfer(p1->contexto,PRIM->contexto);
228      }
229
230      /*localiza a primeira mensagem cheia (flag==1); */
231      p_msg = PRIM->ptr_msg;
232      while(p_msg->flag != 1)
233          p_msg = p_msg->prox;
234
235      /*salva o nome do emissor*/
236      *nome_emissor = p_msg->nome_emissor;
237
238      /*salva a mensagem*/
239      strcpy(msg,p_msg->mensa);
240
241      /*decrementa o contador de mensagens*/
242      (PRIM->qtde_msg_fila)--;
243
244      /*atualiza a flag para zero, sinalizando um "slot"
245      vazio de mensagens*/
246      p_msg->flag = 0;
247
248      /*localiza descritor do emissor;*/
249      p_aux = PRIM;
250
251      /*Encontra o Emissor e caso ele esteja como BLOQENV
252      ele Ã© setado como ATIVO*/
253      while(p_aux->nome != *nome_emissor)
254          p_aux = p_aux->prox_desc;
255      if(p_aux->estado == BLOQENV)
256          p_aux->estado = ATIVO;
257      enable();
258      return;
259  }
```

```
260
261  /* ------------------------ FIM ALTERAÃ‡Ã•ES ------------------------
*/
262
263  void far Dispara_Sistema()
264  {
265      PTR_DESC d_inicio;
266      d_esc = cria_desc();
267      newprocess(Escalador, d_esc);
268      d_inicio = cria_desc();
269      transfer(d_inicio, d_esc);
270  }
271
272  void far Terminar_Processo()
273  {
274      disable();
275      PRIM->estado = TERMINADO;
276      enable();
277      while(1);
278  }
```

8.2 ANEXO D – NUCLEO BASICO + PRIORIDADE SIMPLES

```
  1  #include <system.h>
  2  #include <stdio.h>
  3
  4  typedef struct registros
  5  {
  6      unsigned bx1,es1;
  7  } regis;
  8
  9  typedef union k
 10  {
 11      regis x;
 12      char far *y;
 13  } APONTA_REG_CRIT;
 14
 15  typedef struct Desc_Prog *DESCRITOR_PROC;
 16  struct Desc_Prog /* DEFINICAO DO TIPO BCP */
 17  {
 18      int nome, estado, prioridade, prior_aux;
 19      PTR_DESC contexto;
 20      DESCRITOR_PROC prox_desc;
 21  };
 22
 23  /* VARIAVEIS GLOBAIS */
 24  enum estado {ATIVO, TERMINADO};
 25  PTR_DESC d_esc;
 26  DESCRITOR_PROC PRIM, fim_lista = 0;
 27  APONTA_REG_CRIT a;
 28
 29  /* FUNCOES DO NUCLEO */
 30  void far Criar_Processo(int numero, void far (*end_processo)(), int
prior)
 31  {
 32      DESCRITOR_PROC p_aux;
```

```
33      p_aux = (DESCRITOR_PROC) malloc (sizeof (struct Desc_Prog));
34      p_aux->nome = numero;
35      p_aux->estado = ATIVO;
36      /* Tratando a prioridade enviada pelo usuario */
37      if (prior > 10)
38          prior = 10;
39      else if (prior < 1)
40          prior = 1;
41      /* Continuando as inicializacoes do BCP */
42      p_aux->prioridade = prior;
43      p_aux->prior_aux = 0;
44      p_aux->contexto = cria_desc();
45      newprocess(end_processo, p_aux->contexto);
46      if (PRIM != 0) /* SE NAO FOR O PRIMEIRO PROCESSO */
47      {
48          p_aux->prox_desc = fim_lista->prox_desc;
49          fim_lista->prox_desc = p_aux;
50          fim_lista = p_aux;
51      }
52      else /* SE FOR O PRIMEIRO PROCESSO */
53      {
54          PRIM = p_aux;
55          p_aux->prox_desc = p_aux;
56          fim_lista = p_aux;
57      }
58  }
59
60  void far Volta_Dos()
61  {
62      disable();
63      setvect(8, p_est->int_anterior);
64      enable();
65      exit(0);
66  }
67
68  DESCRITOR_PROC far Procura_Prox_Ativo()
69  {
70      DESCRITOR_PROC aux;
71      aux = PRIM->prox_desc;
72      do
73      {
74          if (aux->estado == ATIVO)
75              return aux;
76          aux = aux->prox_desc;
77      } while (aux != PRIM->prox_desc);
78      return 0;
79  }
80
81  void far Escalador()
82  {
83      p_est->p_origem = d_esc;
84      p_est->p_destino = PRIM->contexto;
85      p_est->num_vetor = 8;
86      /* Iniciando ponteiro para a Pilha do DOS */
87      _AH=0x34;
88      _AL=0x00;
89      geninterrupt(0x21);
90      a.x.bx1=_BX;
91      a.x.es1=_ES;
92      while(1)
93      {
```

```
 94          iotransfer();
 95          disable();
 96          /* Acabou a fatia de tempo do processo. */
 97          PRIM->prior_aux++; /* Incrementando que ja recebeu + 1 fatia
de tempo, independente de estar num trecho do DOS */
 98          if (!*a.y) /* Houve Interrupcao durante um trecho do DOS? */
 99          {
100              /* Se entrou aqui, nao estava em um trecho do DOS! */
101              if (PRIM->prior_aux >= PRIM->prioridade) /* Verificando de
ja rodou o numero de fatias pedidas */
102              {
103                  PRIM->prior_aux -= PRIM->prioridade; /* Caso tenha
recebido fatia extra, sera considerado aqui. */
104                  if ((PRIM = Procura_Prox_Ativo()) == NULL)
105                      Volta_Dos();
106                  p_est->p_destino = PRIM->contexto;
107              }
108          }
109          enable();
110      }
111  }
112
113  void far Dispara_Sistema()
114  {
115      PTR_DESC d_inicio;
116      d_esc = cria_desc();
117      newprocess(Escalador, d_esc);
118      d_inicio = cria_desc();
119      transfer(d_inicio, d_esc);
120  }
121
122  void far Terminar_Processo()
123  {
124      disable();
125      PRIM->estado = TERMINADO;
126      enable();
127      while(1);
128  }
```

8.2 ANEXO E – NUCLEO BASICO + FILAS DE PRIORIDADE

```
 1  #include <system.h>
 2  #include <stdio.h>
 3
 4  #define tamPrior 10
 5
 6  typedef struct registros
 7  {
 8      unsigned bx1,es1;
 9  } regis;
10
11  typedef union k
12  {
13      regis x;
14      char far *y;
15  } APONTA_REG_CRIT;
```

```
 16
 17  typedef struct Desc_Prog *DESCRITOR_PROC;
 18  struct Desc_Prog /* DEFINICAO DO TIPO BCP */
 19  {
 20      int nome, estado, prioridade;
 21      PTR_DESC contexto;
 22      DESCRITOR_PROC prox_desc;
 23  };
 24
 25  /* VARIAVEIS GLOBAIS */
 26  enum estado {ATIVO, TERMINADO};
 27  PTR_DESC d_esc;
 28  DESCRITOR_PROC lista_prioridade[tamPrior] = {0,0,0,0,0,0,0,0,0,0},
fim_lista[tamPrior] = {0,0,0,0,0,0,0,0,0,0}, processo_atual = 0;
 29  APONTA_REG_CRIT a;
 30  int priorAtual = tamPrior - 1, prior_participantes = tamPrior - 1;
 31
 32  /* FUNCOES DO NUCLEO */
 33  void far Criar_Processo(int numero, void far (*end_processo)(), int
prior)
 34  {
 35      DESCRITOR_PROC p_aux;
 36      p_aux = (DESCRITOR_PROC) malloc (sizeof (struct Desc_Prog));
 37      p_aux->nome = numero;
 38      p_aux->estado = ATIVO;
 39      /* Tratando a prioridade enviada pelo usuario */
 40      if (prior >= tamPrior)
 41          prior = tamPrior - 1;
 42      else if (prior < 0)
 43          prior = 0;
 44      /* Continuando as inicializacoes do BCP */
 45      p_aux->prioridade = prior;
 46      p_aux->contexto = cria_desc();
 47      newprocess(end_processo, p_aux->contexto);
 48      if (lista_prioridade[prior] != 0) /* SE NAO FOR O PRIMEIRO
PROCESSO COM AQUELA PRIORIDADE */
 49      {
 50          fim_lista[prior]->prox_desc = p_aux;
 51          fim_lista[prior] = p_aux;
 52      }
 53      else /* SE FOR O PRIMEIRO PROCESSO COM AQUELA PRIORIDADE */
 54      {
 55          lista_prioridade[prior] = p_aux;
 56          fim_lista[prior] = p_aux;
 57      }
 58      p_aux->prox_desc = 0;
 59  }
 60
 61  void far Volta_Dos()
 62  {
 63      disable();
 64      setvect(8, p_est->int_anterior);
 65      enable();
 66      exit(0);
 67  }
 68
 69  DESCRITOR_PROC far Procura_Prox_Ativo()
 70  {
 71      DESCRITOR_PROC auxiliar = processo_atual;
 72      auxiliar = lista_prioridade[priorAtual]->prox_desc;
 73      /* Caso nao esteja ativo - procura na mesma fila de prioridade */
```

```
 74      while (auxiliar != 0)
 75      {
 76          if (auxiliar->estado == ATIVO)
 77              return auxiliar;
 78          auxiliar = auxiliar->prox_desc;
 79      }
 80      /* Caso tenha que mudar de fila de prioridade */
 81      while (1)
 82      {
 83          /* Passando para outra prioridade - Varios tratamentos devem
ser feitos */
 84          priorAtual--; /* Percorre a lista de processos participantes
*/
 85          if (priorAtual < prior_participantes) /* Se vai aumentar o
numero de participantes */
 86          {
 87              priorAtual = tamPrior - 1; /* Comeca a ver desde o que tem
maior prioridade */
 88              prior_participantes--; /* Aumenta o numero de
participantes a serem escalados */
 89              if (prior_participantes < 0) /* Se ja tinha todos como
participantes */
 90                  prior_participantes = tamPrior - 1; /* Reseta o numero
de participantes para somente os que tem maior prioridade */
 91          }
 92          auxiliar = lista_prioridade[priorAtual]; /* Auxiliar recebe o
primeiro da lista */
 93          /* Percorrendo a fila inteira da prioridade atual */
 94          while (auxiliar != 0)
 95          {
 96              if (auxiliar->estado == ATIVO)
 97                  return auxiliar;
 98              if (auxiliar == processo_atual)
 99                  return 0;
100              auxiliar = auxiliar->prox_desc;
101          }
102      }
103  }
104
105  DESCRITOR_PROC Procura_Primeiro_Ativo()
106  {
107      DESCRITOR_PROC auxiliar;
108      int i;
109      /* Se houver algum elemento com prioridade maxima, ele eh o
primeiro */
110      for (i = tamPrior - 1; i >= 0; i--)
111      {
112          if ((auxiliar = lista_prioridade[i]) != 0)
113          {
114              priorAtual = prior_participantes = i;
115              return auxiliar;
116          }
117      }
118      return 0;
119  }
120
121  void far Escalador()
122  {
123      p_est->p_origem = d_esc;
124      p_est->p_destino = processo_atual->contexto;
125      p_est->num_vetor = 8;
```

```
126     /* Iniciando ponteiro para a Pilha do DOS */
127     _AH=0x34;
128     _AL=0x00;
129     geninterrupt(0x21);
130     a.x.bx1=_BX;
131     a.x.es1=_ES;
132     while(1)
133     {
134         iotransfer();
135         disable();
136         /* Acabou a fatia de tempo do processo. */
137         if (!*a.y) /* Houve Interrupcao durante um trecho do DOS? */
138         {
139             if ((processo_atual = Procura_Prox_Ativo()) == NULL)
140                 Volta_Dos();
141             p_est->p_destino = processo_atual->contexto;
142         }
143         enable();
144     }
145 }
146
147 void far Dispara_Sistema()
148 {
149     PTR_DESC d_inicio;
150     d_esc = cria_desc();
151     newprocess(Escalador, d_esc);
152     d_inicio = cria_desc();
153     processo_atual = Procura_Primeiro_Ativo();
154     transfer(d_inicio, d_esc);
155 }
156
157 void far Terminar_Processo()
158 {
159     disable();
160     processo_atual->estado = TERMINADO;
161     enable();
162     while(1);
163 }
```