Matheus Polachini e Gabriel Vieira

Sistemas Operacionais II

Brasil

21 de Junho de 2016

Matheus Polachini e Gabriel Vieira

Sistemas Operacionais II

Monografia apresentada como parte de avaliação da disciplina Sistemas Operacionais II, curso de Bacharel em Ciência da Computação, Universidade Estadual Paulista (UNESP).

Prof. Dr. Antônio Carlos Sementille

Universidade Estadual Paulista Júlio de Mesquita Filho

Ciência da Computação

Brasil

21 de Junho de 2016

Resumo

Este documento tem por objetivo demonstrar e explicar o algoritmo de contrução de um núcleo de um sistema operacional DOS incluindo o uso de semáforos e troca de mesagens simulando comunicação via rede. Para tal, foi usada uma máquina virtual DOSBox 7.4 e linguagem de programação C. A realização desse projeto resultou em um aprendizado prático sobre o funcionamento de um núcleo.

Palavras-chaves: sistema operacional. nucleo. DOS.

Sumário

1 Núcleo Básico

O núcleo básico contém as estruturas mínimas para o funcionamento dos processos gerados pelo usuário. Isso inclui o escalonamento desses processos, bem como todas as operações para tal. As bibliotecas utilizadas foram stdio.h e system.h. Esta última foi fornecida pelo professor Prof. Dr. Antônio Carlos Sementille da Univeridade Estadual Paulista (UNESP) com controles específicos para uso do TIMER e outras operações do sistemas operacional DOS.

O conjunto de funções usadas no núcleo básico pode ser observadas nos tópicos a seguir.

1.1 Estruturas utilizadas

Foram criadas três estruturas (tipo struct em linguagem C) para manipulação e operação das funções.

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

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

APONTA_REG_CRIT a;
```

As estruturas acima são utilizadas pelo escalador para acessar o endereço de um flag após a chamada ao serviço 34H do DOS.

```
typedef struct desc_p{
        char nomep[35];
        enum{ativo, terminado} estado;
        PTR_DESC contexto;
        struct desc_p *prox_desc;
}DESCRITOR_PROC;
```

A estrutura DESCRITOR_PROC foi criada para repesentar o BCP de todos processos a serem criados pelo usuário, ou seja, será o descritor do processo e armazenará todas as informações de um determinado processo. Possuirá as seguintes variáveis:

- nomep: nome do processo;
- estado: o estado atual de execução que no caso no núcleo básico possuirá somente dois estados (ativo e terminado);
- contexto: possui todas as informações do processo, como valores do registradores, onde no código fonte a execução do programa para quando ocorre interrupção, entre outras informações.
- *prox_desc: ponteiro apontando para o próximo processo existente na fila de processos criados pelo usuário.

1.2 Variáveis das estruturas

Foram criadas as seguintes variáveis derivadas das estruturas visto anteriormente:

```
1 APONTA_REG_CRIT a;
2 typedef DESCRITOR_PROC *PTR_DESC_PROC;
3 PTR_DESC_PROC prim = NULL;
4 PTR_DESC d_esc;
```

- a: union representando o endereço de um flag que indica se o DOS está em uma região crítica
- *PTR_DESC_PROC: ponteiro derivado pra representar a estrutura DESCRITOR_PROC ao longo do código;
- prim: criada para representar o processo corrente, ou seja, que estará sendo executado no momento, iniciada com o valor NULL;
- d_esc: criada para representar a corrotina do escalador.

1.3 Função criar_processo

Esta função recebe como parâmetros o endereço e nome do processo do usuário para ser associado ao descritor da corrotina correspondente.

Primeiramente a função cria dinâmicamente um descritor do processo através do malloc e atribui este processo a variável p_aux, se p_aux for igual a NULL, o núcleo encerra sua execução, pois isso simboliza que não houve como criar um novo processo. Esta situação pode acontecer por diversos fatores, sendo um deles a alocação de memória do DOS não ser suficiente para o processo.

Logo após a função atribui os dados da nova variável a seu BCP, ou seja, o nome, estado, contexto e chama a função newprocess cujo objetivo é associar esses dados com o descritor da corrotina correspondente.

A seguir é feita várias verificações para montar a fila de processos de maneira correta, ajustando os ponteiros para o caso do processo ser o primeiro da fila (if), ser o segundo (else if) ou demais posições (else). Para o caso das demais posições é necessário usar uma variável auxiliar para encontrar o final da fila sem perder o ponteiro que aponta para o primeiro da fila (prim), isso é feito através do while.

```
1  void far criar_processo (void far (*end_proc)(), char nome_proc
       []){
2          PTR_DESC_PROC p_aux, aux;
3          p_aux = (PTR_DESC_PROC) malloc (sizeof(DESCRITOR_PROC));
4          if (p_aux == NULL)
5             exit(1);
6          strcpy (p_aux->nomep, nome_proc);
7          p_aux->estado = ativo;
8          p_aux->contexto = cria_desc();
9          newprocess (end_proc , p_aux->contexto);
10         if (prim == NULL){
11                 prim = p_aux;
12                 prim->prox_desc = NULL;
13         } else if(prim->prox_desc == NULL){
14                 prim->prox_desc = p_aux;
15                 p_aux->prox_desc = prim;
16         } else{
17                 aux = prim;
18                 while(aux->prox_desc != prim)
19                         aux = aux->prox_desc;
20                 p_aux->prox_desc = prim;
21                 aux->prox_desc = p_aux;
22         }
23 }
```

1.4 Função procura_prox_ativo

Esta função tem por objetivo procurar o próximo processo a ser executado, após o atual ser boqueado por algum motivo ou finalizado. Utilizando uma variável auxiliar com o mesmo valor do ponteiro do processo corrente, é feito uma busca na fila de processos existentes a procura de um processo com estado igual a ativo. O primeiro processo a ser

encontrado é retornado para tornar-se o novo processo corrente. Caso não seja encontrado nenhum processo com estado igual a ativo, a função simboliza o ato retornando NULL.

```
PTR_DESC_PROC far procura_prox_ativo(){
        PTR_DESC_PROC aux;
        aux = prim->prox_desc;
        while(aux->prox_desc != prim){
                if(aux->estado == ativo)
                        return aux;
                else
                        aux = aux->prox_desc;
        }
        return NULL;
}
```

1.5 Função escalador

O escalador do núcleo funcionará utilizando a função iotransfer() para transferir a execução de um processo para outro quando ocorrer uma interrupção do timer. Para isso, os campos da estrutura apontada por p_est devem ser inicializados adequadamente. Como a trasferência será sempre do escalador para um processo, p_origem sempre será o descritor da co-rotina do escalador. O destino, no início, é inicializado com a co-rotina primeiro processo da lista de processos. Como será utilizada a interrupção do timer para trocar os processos, num_vetor sempre possuirá o valor 8, que indica a interrupção desejada.

No momento em que ocorre uma interrupção, o processo só poderá ser trocado caso o processo atual não esteja em uma região crítica do DOS (por exemplo, não esteja no meio da execução do código de um driver). Para detectar se esse é o caso, é necessário fazer uma chamada ao serviço 34H do DOS. Para isso, é necessário armazenar o valor 34H no registrador AH e o valor 0H no registrador AL, gerando, a seguir, uma interrupção associada ao endereço 21H.

Tal serviço do DOS retorna o endereço de um flag, e é esse flag que possui a informação necessária para o escalador. Após a chamada ao serviço, o endereço de retorno encontra-se armazenado nos registradores ES e BX, na forma ES:BX. Para acessar esse endereço através da linguagem C, é necessário armazenar os valores dos registradores em posições consecutivas na memória, através de uma estrutura, e utilizar um union para interpretar os valores como um ponteiro para um endereço.

Por fim, ao obter o ponteiro, representado por *y, o flag pode ser acessado através do endereço apontado por ele. Caso esse flag tenha valor 0, o processo atual não está em

uma região crítica e, portanto, pode ser trocado.

```
void far escalador(){
        p_est->p_origem = d_esc;
        p_est->p_destino = prim->contexto;
        p_est->num_vetor = 8;
        _AH=0x34;
        _AL=0x00;
        geninterrupt(0x21);
        a.x.bx1=_BX;
        a.x.es1=_ES;
        while(1){
                iotransfer();
                disable();
                if(!*a.y){
                        prim->contexto = p_est->p_destino;
                        prim=procura_prox_ativo();
                        p_est->p_destino = prim->contexto;
                }
                enable();
        }
}
```

1.6 Função volta_dos

Esta função tem por objetivo restaurar as configurações originais do emulador DOS utilizado para sua execução sem interferência do núcleo. Como essa função deve ser atômica, são desabilitadas as interrupções através da função disable. Logo após, a função setvect restaura o endereço original da rotina do timer (8), que havia sido modificado por funções da biblioteca system.h. Ou seja, toda essa modificação de endereços foi feita oculto ao código fonte do núcleo.

Por fim, a função enable habilita novamente as interrupções e o programa é finalizado através da função exit.

```
void far volta_dos(){
        disable();
        setvect(8, p_est->int_anterior);
        enable();
        exit(0);
}
```

1.7 Função dispara_sistema

Para iniciar o funcionamento do núcleo, no programa teste do usuário será chamada essa função. Primeiramente, a função cria um descritor de co-rotina, representado por d_aux, que será usado apenas para transferir o controle para outra co-rotina.

Após isso, a função cria outro descritor de co-rotina, indicado por d_esc, e o associa à função escalador através da função newprocess. Por fim, o controle é transferido de d_aux para d_esc, iniciando o funcionamento do escalador.

```
1 void far dispara_sistema(){
2         PTR_DESC d_aux;
3         d_aux = cria_desc();
4         d_esc = cria_desc();
5         newprocess(escalador, d_esc);
6         transfer (d_aux, d_esc);
7 }
```

1.8 Função termina_processo1

Esta função tem por objetivo encerrar o processo que terminou toda sua execução, para isso são desabilitadas as interrupções através da função disable e alterado seu estado para terminado. Logo após a função procura_prox_ativo encontra o próximo processo a ser colocado em execução. Se não houver nenhum processo, o programa do núcleo é finalizado, caso houver a execução do processo corrente é passada para o processo encontrado como ativo. Este último realizado pela função transfer.

```
1 void far termina_processo1(){
2         PTR_DESC_PROC p_aux;
3         disable();
4         prim->estado = terminado;
5         p_aux = prim;
6         if((prim = procura_prox_ativo()) == NULL)
7             volta_dos();
8         transfer(p_aux->contexto, prim->contexto);
9 }
```

2 Núcleo com semáforos

O núcleo com uso de semáforos foi derivado do núcleo básico, portanto neste tópico será apresentado somente as modificações realizadas a partir do núcleo básico.

2.1 Estruturas utilizadas

Houve duas alterações em relação às estrutura desc_p do núcleo básico, sendo elas:

- bloq_p: um novo estado para sinalizar o processo bloqueado pela primitiva P do semáforo;
- *fila_sem: ponteiro para sinalizar a fila dos bloqueados por um determinado semáforo.

```
1 typedef struct desc_p{
2         char nomep[35];
3         enum{ativo, bloq_p, terminado} estado;
4         PTR_DESC contexto;
5         struct desc_p *fila_sem;
6         struct desc_p *prox_desc;
7 }DESCRITOR_PROC;
```

Também houve a criação de uma nova estrutura demonimada semaforo que conterá os seguintes campos:

- s: contador para o semáforo.
- Q: uma representação da estrutura desc_p para ser usado como cabeça de fila para aquele determinado semádoro.

```
1 typedef struct {
2         int s;
3         PTR_DESC_PROC Q;
4 }semaforo;
```

As demais estruturas não foram alteradas.

2.2 Função inicia_semaforo

Esta função possui dois parâmetros: um valor inteiro n que iniciará o contador do semáforo (s) com o valor atribuido a n e um ponteiro representando o semáforo a ser manipulado. A função também inicia a fila do semáforo que a está chamando com NULL, já que no inicio nenhum processo está bloqueado pelo semáforo.

```
1  void far inicia_semaforo(semaforo *sem, int n) {
2          sem->s = n;
3          sem->Q = NULL;
4  }
```

2.3 Função p

A função p possui um enable e disable para habilitar e desabilitar as interrupções, pois se trata de uma região crítica.

Esta função representa a primitiva p do semáforo e possui um parâmetro: o semáforo a ser manipulado. A função primeiramente testa se o semáforo é maior que zero, se sim significa que nenhum outro processo solicitou os mesmos recursos, por isso apenas decrementa o semáforo. Se não, significa que outro processo já solicitou e está usando determinado recurso, por isso o processo muda seu estado para bloqueado pelo semáforo (bloq_p) e é inserido na fila dos bloqueados. O primeiro if é para o caso dele ser o primeiro da fila, o else é para o caso dele ser o segundo em diante, pois é necessario um ponteiro auxliar (aux) para varrer toda fila até encontrar o final, assim o processo poderá ser pendurado no final da fila sem haver perda de dados pela manipulação do ponteiro.

Caso o processo seja bloqueado pela primitiva p, o escalador deverá encontrar o próximo processo para a execução através da função procura próximo_ativo. Se não for encontrado nenhum processo, o programa volta a execução para o DOS.

```
1  void far p(semaforo *sem){
2          PTR_DESC_PROC aux;
3          disable();
4          if(sem->s > 0)
5                  sem->s--;
6          else {
7                  prim->estado = bloq_p;
8                  if(sem->Q == NULL){
9                          sem->Q = prim;
10                         prim->fila_sem = NULL;
11                 }
12                 else{
```

```
13                          aux = sem->Q;
14                          while(aux->fila_sem != NULL)
15                                  aux = aux->fila_sem;
16                          aux->fila_sem = prim;
17                          prim->fila_sem = NULL;
18                  }
19                  aux = prim;
20                  if((prim = procura_prox_ativo()) == NULL)
21                          volta_dos();
22                  transfer(aux->contexto, prim->contexto);
23                  }
24          enable();
25  }
```

2.4 Função v

A função v possui um enable e disable para habilitar e desabilitar as interrupções, pois se trata de uma região crítica.

Esta função representa a primitiva v do semáforo e possui um parâmetro: o semáforo a ser manipulado. A função primeramente testa se o semáforo é diferente de NULL, se sim significa há pelo menos um processo bloqueado na fila desse semáforo, o estado do primeiro processo na fila é mudado para ativo e removido da fila. Se não, significa que não há nenhum processo bloqueado, por isso é só incrementado o semáforo.

```
1   void far v(semaforo *sem){
2           PTR_DESC_PROC aux;
3           disable();
4           if(sem->Q != NULL){
5                   sem->Q->estado = ativo;
6                   aux = sem->Q->fila_sem;
7                   sem->Q = aux;
8           } else
9                   sem->s++;
10          enable();
11  }
```

3 Núcleo com troca de mensagens

O núcleo com uso de troca de mensagens foi derivado do núcleo com semáforos, portanto neste tópico será apresentado somente as modificações realizadas apartir do núcleo com semáforos.

3.1 Estruturas utilizadas

Houve a criação de uma nova estrutura responsável por representar as mensagens enviadas.

```
1 typedef struct address {
2   int flag;
3   char nome_emissor[35];
4   char mensa[25];
5   struct address *ptr_msg;
6 } mensagem;
7
8 typedef mensagem *PTR_MENSAGEM;
```

Seus campos são descritos da seguinte forma:

- flag: indica se a estrutura está preenchida com uma mensagem que ainda não foi lida (valor 1) ou se a mensagem da estrutura já foi lida (valor 0).
- nome_emissor: armazena o nome do processo que enviou a mensagem.
- mensa: armazena o texto da mensagem
- *ptr_msg: ponteiro para a próxima mensagem da lista de mensagens de um processo

Houve também alterações no descritor de processo.

```
1 typedef struct desc_p {
2   char nome[35];
3   enum {ativo, terminado, bloqrec, bloqrecSeletivo, bloqenv,
       bloq_P} estado;
4   PTR_DESC contexto;
5   struct desc_p *prox_desc;
6   struct desc_p *fila_sem;
7
8   PTR_MENSAGEM ptr_msg;
```

```
9   int tam_fila;
10  int qtde_msg_fila;
11
12  char nomeRecebeSeletivo[35];
13 } DESCRITOR_PROC;
```

Três novos estados foram adicionados:

- bloqrec: representa o bloqueio devido à chamada da primitiva recebe();
- bloqenv: representa o bloqueio devido à chamada da primitiva envia();
- bloqrecSeletivo: representa o bloqueio devido à chamada da primitiva recebe_seletivo();

Também foram adicionados quatro novos campos que lidam com o gerenciamento das mensagens:

- ptr_msg: ponteiro para a lista de mensagens recebidas pelo processo;
- tam_fila: número máximo de mensagens que podem existir na lista de mensagens do processo;
- qtde_msg_fila: quantidade de mensagens não lidas existentes na fila de mensagens do processo;
- nomeRecebeSeletivo: armazena o nome do processo emissor do qual se deseja receber uma mensagem após uma chamada à primitiva recebe_seletivo();

3.2 Função cria_fila_mensa

Essa função é responsável por criar e retornar uma fila de mensagens do tamanho especificado pelo parâmetro max_fila. Todas as mensagens inseridas na fila são inicializadas com o valor de flag 0.

```
1  PTR_MENSAGEM cria_fila_mensa(int max_fila) {
2    PTR_MENSAGEM fila = NULL, novo_elemento;
3    int i = 0;
4
5    while(i < max_fila) {
6      novo_elemento = (PTR_MENSAGEM) malloc(sizeof(mensagem));
7      novo_elemento->flag = 0;
8      novo_elemento->ptr_msg = fila;
9      fila = novo_elemento;
10     i++;
```

```
  }

  return fila;
};
```

3.3 Função criar_Processo

Essa função foi modificada de modo a receber o tamanho da fila de mensagens do processo como parâmetro, que será armazenado no campo tam_fila do seu descritor. O campo ptr_msg é inicializado com o retorno de uma chamada à função cria_fila_mensa, passando o tamanho especificado como parâmetro. Como, no início, não há mensagens novas a serem lidas, o campo qtde_msg_fila é inicializado com o valor 0.

```
void far criar_Processo(char nome_proc[35], void far (*end_proc)
   (), int max_fila) {
  PTR_DESC_PROC p_aux;

  p_aux = (PTR_DESC_PROC) malloc(sizeof(DESCRITOR_PROC));
  if(p_aux == NULL)
    exit(1);

  strcpy(p_aux->nome, nome_proc);
  p_aux->estado = ativo;
  p_aux->contexto = cria_desc();
  p_aux->fila_sem = NULL;
  p_aux->tam_fila = max_fila;
  p_aux->qtde_msg_fila = 0;
  p_aux->ptr_msg = cria_fila_mensa(max_fila);
  newprocess(end_proc, p_aux->contexto);

  if (prim == NULL) {
    prim = p_aux;
    prim->prox_desc = NULL;
  } else {
    aux = prim;
    while(aux->prox_desc != prim)
      aux = aux->prox_desc;

    p_aux->prox_desc = prim;
    aux->prox_desc = p_aux;
  }
}
```

3.4 Função procuraProcesso

Função auxiliar responsável por retornar um ponteiro para o descritor de processo que possui o nome especificado como parâmetro. Caso não exista nenhum descritor com tal nome, o valor retornado é NULL.

```
1  PTR_DESC_PROC procuraProcesso(char *nome) {
2    PTR_DESC_PROC d_aux;
3
4    if(strcmp(nome, PRIM->nome) == 0) {
5      return PRIM;
6    }
7
8    d_aux = PRIM->prox_desc;
9    while(d_aux != PRIM) {
10     if(strcmp(nome, d_aux->nome) == 0) {
11       return d_aux;
12     }
13
14     d_aux = d_aux->prox_desc;
15   }
16
17   return NULL;
18 }
```

3.5 Função localizaMensagem

Função auxiliar responsável por retornar um ponteiro para uma mensagem que possui a flag especificada como parâmetro, dentro da lista de mensagens informada.

```
1  PTR_MENSAGEM localizaMensagem(PTR_MENSAGEM fila, int flagDesejada
       ) {
2    PTR_MENSAGEM aux = fila;
3    while(aux->flag != flagDesejada) {
4      aux = aux->ptr_msg;
5    }
6
7    return aux;
8  }
```

3.6 Função localizaMensagemSeletivo

Função auxiliar semelhante à função localizaMensagem, porém retornando uma mensagem que possui o nome do emissor especificado além da flag passada como parâmetro.

```
1  PTR_MENSAGEM localizaMensagemSeletivo(PTR_MENSAGEM fila, char *
       nome_emissor, int flagDesejada) {
2    PTR_MENSAGEM aux = fila;
3    while(aux != NULL) {
4      if((aux->flag == flagDesejada) && (strcmp(aux->nome_emissor,
           nome_emissor) == 0)) {
5        return aux;
6      }
7      aux = aux->ptr_msg;
8    }
9
10   return NULL;
11 }
```

3.7 Função envia

Função responsável por enviar uma mensagem a outro processo, bloqueando o processo emissor até que a mensagem seja recebida pelo receptor.

Como o processo não pode ser trocado durante a sua execução, a função disable() é utilizada no começo, desativando as interrupções, e a função enable() é utilizada nos pontos de saída da função, de modo a restaurar as interrupções.

Uma chamada é feita à função procuraProcesso, que retorna o descritor do processo destino da mensagem. Caso o valor retornado seja NULL, a função retorna o código de erro 1, indicando que o destino da mensagem não existe.

Após isso, é verificado se o número de mensagens na fila de mensagens do processo destino é igual ao número máximo de mensagens do processo, retornando o código de erro 2 caso seja, indicando que a fila de mensagens está cheia.

Com isso, há a certeza de que existe uma mensagem vazia (com flag igual a 0) na fila de mensagens do processo destino. Uma chamada à função localizaMensagem é feita para localizá-la. Essa mensagem, então, tem os campos mensa e nome_emissor inicializados com valores correspondentes passados à função envia por parâmetro.

Como o processo atual deve ser bloqueado devido a uma chamada à função envia(), seu estado é alterado para bloqenv. Caso o estado do processo receptor seja bloqrec, ele

é desbloqueado. Caso o estado do receptor seja bloqrecSeletivo e o seu campo nomeRecebeSeletivo tenha o valor do nome do processo atual, ele também é desbloqueado.

Após isso, como o processo atual está bloqueado e não pode continuar, o processo é trocado para o próximo processo ativo. Caso não exista tal processo, é feita uma chamada à função volta_dos().

```
1  int far envia(char *nome_destino, char *msg) {
2    PTR_DESC_PROC d_aux, p_aux;
3    PTR_MENSAGEM slot;
4
5    disable();
6    d_aux = procuraProcesso(nome_destino);
7
8    if(d_aux == NULL) {
9      enable();
10     return 1;
11   }
12
13   if(d_aux->tam_fila == d_aux->qtde_msg_fila) {
14     enable();
15     return 2;
16   }
17
18   slot = localizaMensagem(d_aux->ptr_msg, 0);
19   slot->flag = 1;
20   strcpy(slot->nome_emissor, PRIM->nome);
21   strcpy(slot->mensa, msg);
22   (d_aux->qtde_msg_fila)++;
23
24   PRIM->estado = bloqenv;
25
26   if(d_aux->estado == bloqrec) {
27     d_aux->estado = ativo;
28   }
29
30   if(d_aux->estado == bloqrecSeletivo) {
31     if(strcmp(d_aux->nomeRecebeSeletivo, PRIM->nome) == 0) {
32       d_aux->estado = ativo;
33     }
34   }
35
36   p_aux = PRIM;
```

```
37    if((PRIM = procura_prox_ativo()) == NULL) {
38      volta_dos();
39    }
40
41    transfer(p_aux->contexto, PRIM->contexto);
42    return 0;
43  }
```

3.8 Função recebe

Função responsável por receber uma mensagem enviada por outro processo, bloqueando o receptor até que exista uma mensagem que possa ser recebida.

De forma semelhante à função envia, as funções disable() e enable() são utilizadas para tornar a função recebe atômica.

Caso a lista de mensagens do processo que chamou o recebe (ou seja, o processo atual) esteja vazia, o processo é bloqueado e a execução passa para o próximo processo ativo, retornando quando existir uma mensagem que possa ser lida. Devido à utilização do transfer, que possui um enable() interno, é necessário chamar novamente a função disable().

Após isso, é feita uma chamada à função localizaMensagem, de forma a localizar uma mensagem nova (com flag igual a 1). A flag da mensagem é, então, alterada para 0, indicando que ela foi lida, e o seu texto e nome do emissor são copiados para os parâmetros de saída correspondentes na função recebe. Como uma mensagem foi lida, o campo qtde_msg_fila do processo atual é decrementado.

Por fim, o processo emissor é localizado através de uma chamada à função procuraProcesso e é desbloqueado caso seu estado seja bloqenv.

```
1   void far recebe(char *msg, char *p_emissor) {
2     PTR_DESC_PROC p_aux, d_aux;
3     PTR_MENSAGEM slot;
4
5     disable();
6     if(PRIM->qtde_msg_fila == 0) {
7       PRIM->estado = bloqrec;
8       p_aux = PRIM;
9       if((PRIM = procura_prox_ativo()) == NULL) {
10        volta_dos();
11      }
12      transfer(p_aux->contexto, PRIM->contexto);
13    }
```

```
14    disable();
15
16    slot = localizaMensagem(PRIM->ptr_msg, 1);
17    slot->flag = 0;
18
19    strcpy(p_emissor, slot->nome_emissor);
20    strcpy(msg, slot->mensa);
21
22    (PRIM->qtde_msg_fila)--;
23
24    d_aux = procuraProcesso(slot->nome_emissor);
25
26    if(d_aux->estado == bloqenv) {
27      d_aux->estado = ativo;
28    }
29
30    enable();
31  }
```

3.9 Função recebe_seletivo

Função responsável por receber uma mensagem enviada por um processo específico, bloqueando o receptor até que exista uma mensagem desse processo que possa ser recebida.

Seu funcionamento é semelhante ao funcionamento da função recebe, com a diferença de que utiliza-se a função localizaMensagemSeletivo para verificar se já existe uma mensagem do emissor desejado e para localizá-la após o processo ser desbloqueado.

No momento em que o processo receptor é bloqueado, o nome do processo do qual se deseja receber uma mensagem é armazenado no campo nomeRecebeSeletivo do receptor.

```
1  void far recebe_seletivo(char *nome_processo, char *msg, char *
       p_emissor) {
2    PTR_DESC_PROC p_aux, d_aux;
3    PTR_MENSAGEM slot;
4
5    disable();
6
7    slot = localizaMensagemSeletivo(PRIM->ptr_msg, nome_processo,
         1);
8
9    if(slot == NULL) {
```

```
10      PRIM->estado = bloqrecSeletivo;
11
12      strcpy(PRIM->nomeRecebeSeletivo, nome_processo);
13
14      p_aux = PRIM;
15      if((PRIM = procura_prox_ativo()) == NULL) {
16        volta_dos();
17      }
18      transfer(p_aux->contexto, PRIM->contexto);
19    }
20
21    disable();
22
23    slot = localizaMensagemSeletivo(PRIM->ptr_msg, nome_processo,
          1);
24    slot->flag = 0;
25
26    strcpy(p_emissor, slot->nome_emissor);
27    strcpy(msg, slot->mensa);
28
29    (PRIM->qtde_msg_fila)--;
30
31    d_aux = procuraProcesso(slot->nome_emissor);
32    if(d_aux->estado == bloqenv) {
33      d_aux->estado = ativo;
34    }
35
36    enable();
37  }
```

4 Núcleo com prioridades

O núcleo com prioridades foi derivado do núcleo básico, portanto neste tópico será apresentado somente as modificações realizadas apartir do núcleo básico.

4.1 Estruturas utilizadas

Houve a adição de um novo campo à estrutura desc_p que indica a prioridade do processo, variando de 0 (menor prioridade) até 4 (maior prioridade).

```
typedef struct desc_p {
    char nome[35];
    enum {ativo, terminado} estado;
    PTR_DESC contexto;
    struct desc_p *prox_desc;
    int prioridade;
} DESCRITOR_PROC;
```

As demais estruturas não foram alteradas.

4.2 Variáveis das estruturas

Foram criadas cinco listas de descritores de processo, cada uma associada a um dos cinco níveis de prioridade existentes. Todas elas são inicializadas com o valor NULL.

```
PTR_DESC_PROC lista_processo[5] = {NULL, NULL, NULL, NULL, NULL}
```

4.3 Função insere_Descritor

Função responsável por inserir o descritor do processo na lista associada à sua prioridade.

```
void insere_Descritor(PTR_DESC_PROC descritor) {
  PTR_DESC_PROC fila = lista_processo[descritor->prioridade];
  descritor->prox_desc = NULL;
  if(fila == NULL) {
    lista_processo[descritor->prioridade] = descritor;
    return;
  }

```

```
9   while(fila->prox_desc != NULL) {
10    fila = fila->prox_desc;
11  }
12
13  fila->prox_desc = descritor;
14 }
```

4.4 Função junta_listas

Essa função recebe como parâmetro dois níveis de prioridade e adiciona a lista de processos referente ao segundo parâmetro no final da lista de processos referente ao primeiro parâmetro.

```
1  void far junta_listas(int indice1, int indice2) {
2    PTR_DESC_PROC ultimo;
3
4    if(lista_processo[indice1] == NULL) {
5      lista_processo[indice1] = lista_processo[indice2];
6      return;
7    }
8
9    ultimo = lista_processo[indice1];
10   while(ultimo->prox_desc != NULL) {
11     ultimo = ultimo->prox_desc;
12   }
13
14   ultimo->prox_desc = lista_processo[indice2];
15 }
```

4.5 Função criar_Processo

Essa função foi modificada de modo a incluir o parâmetro que especifica a prioridade do processo. O valor recebido é validado de forma a garantir que esteja dentro dos limites inferior e superior de prioridade. Além disso, como a lista de processos não é mais circular, o campo prox_desc do novo processo é inicializado com NULL. Por fim, a inserção do processo na lista foi substituída por uma chamada à função insere_Descritor.

```
1  void far criar_Processo(char nome_proc[35], void far (*end_proc)
     (), int prioridade) {
2    PTR_DESC_PROC p_aux;
3
```

```
4   p_aux = (PTR_DESC_PROC) malloc(sizeof(DESCRITOR_PROC));
5   if(p_aux == NULL)
6     exit(1);
7
8   if(prioridade < 0) {
9     prioridade = 0;
10  } else if(prioridade > 4) {
11    prioridade = 4;
12  }
13
14  strcpy(p_aux->nome, nome_proc);
15  p_aux->estado = ativo;
16  p_aux->contexto = cria_desc();
17  p_aux->prioridade = prioridade;
18
19  p_aux->prox_desc = NULL;
20  newprocess(end_proc, p_aux->contexto);
21
22  insere_Descritor(p_aux);
23 }
```

4.6 Função procura_prox_ativo

Essa função foi modificada de modo a receber como parâmetro o ponteiro para um descritor de processo, a partir do qual deve ser iniciada a busca por um processo ativo.

```
1  PTR_DESC_PROC procura_prox_ativo(PTR_DESC_PROC inicio) {
2    PTR_DESC_PROC elemento = inicio;
3
4    while(elemento != NULL) {
5      if(elemento->estado == ativo) {
6        return elemento;
7      } else {
8        elemento = elemento->prox_desc;
9      }
10   }
11
12   return NULL;
13 }
```

4.7 Função escalador

A função escalador foi modificada de modo a considerar as listas de cada prioridade. O primeiro processo é escolhido a partir de uma busca por um processo com a maior prioridade disponível.

Quando ocorre a interrupção, é verificado se existe outro processo ativo na lista de prioridade atual após a posição do processo que está rodando, escolhendo este para rodar caso exista. Caso a busca tenha atingido o final da lista, a lista atual é adicionada no final da lista de prioridade imediamente inferior, reiniciando a busca a partir da primeira posição dessa lista.

Caso a busca atinja o fim da última lista, todas as listas são esvaziadas e cada processo é reinserido na lista correspondente à sua prioridade. O processo escolhido para rodar é feito por uma busca por um processo ativo com maior prioridade disponível. Caso não exista nenhum processo ativo, o escalador termina chamando a função volta_dos().

```
1  void far escalador() {
2    int i, indice = 4;
3    int prioridadeAtual;
4    PTR_DESC_PROC aux, inicio, proximo;
5    p_est->p_origem = d_esc;
6    p_est->num_vetor = 8;
7
8    while(lista_processo[indice] == NULL) {
9      indice--;
10   }
11   PRIM = lista_processo[indice];
12   prioridadeAtual = indice;
13
14   p_est->p_destino = PRIM->contexto;
15
16   _AH = 0x34;
17   _AL = 0x00;
18   geninterrupt(0x21);
19   a.x.bx1 = _BX;
20   a.x.es1= _ES;
21
22   while(1) {
23     iotransfer();
24     disable();
25
26     if(!*a.y) {
```

```
27      aux = procura_prox_ativo(PRIM->prox_desc);
28      if(aux != NULL) {
29        PRIM = aux;
30        p_est->p_destino = PRIM->contexto;
31      } else {
32        do {
33          if(prioridadeAtual != 0) {
34            junta_listas(prioridadeAtual - 1, prioridadeAtual);
35            lista_processo[prioridadeAtual] = NULL;
36            prioridadeAtual--;
37          } else {
38            aux = NULL;
39            break;
40          }
41        } while((aux = procura_prox_ativo(lista_processo[
             prioridadeAtual])) == NULL);
42
43        if(aux != NULL) {
44          PRIM = aux;
45          p_est->p_destino = PRIM->contexto;
46        } else {
47
48          inicio = lista_processo[0];
49
50          i = 0;
51          while(i < 5) {
52            lista_processo[i] = NULL;
53            i++;
54          }
55
56          while(inicio != NULL) {
57            proximo = inicio->prox_desc;
58            inicio->prox_desc = NULL;
59            insere_Descritor(inicio);
60
61            inicio = proximo;
62          }
63
64          prioridadeAtual = 4;
65
66          while( ((aux = procura_prox_ativo(lista_processo[
               prioridadeAtual])) == NULL) && (prioridadeAtual != 0)
```

```
                ) {
67              prioridadeAtual--;
68            }
69
70            if(aux != NULL) {
71              PRIM = aux;
72              p_est->p_destino = PRIM->contexto;
73            } else {
74              volta_dos();
75            }
76          }
77        }
78      }
79
80      enable();
81    }
82  }
```

5 Testes

Todos os testes são realizados pelo usuário, portanto os códigos fontes seriam escrito por eles, tendo acesso somente as funções presentes na interface da biblioteca núcleo.h

5.1 Teste núcleo básico

A biblioteca núcleo.h terá os escopos das seguintes funções para o usuário:

```
#include<system.h>
#include<stdio.h>
typedef struct desc_p{
        char nomep[35];
        enum{ativo, terminado} estado;
        PTR_DESC contexto;
        struct desc_p *prox_desc;
}DESCRITOR_PROC;
typedef DESCRITOR_PROC *PTR_DESC_PROC;
extern void far volta_dos();
extern PTR_DESC_PROC far procura_prox_ativo();
extern void far criar_processo(void far (*end_proc)(), char
   nome_proc[]);
extern void far escalador();
extern void far dispara_sistema();
extern void far termina_processo();
extern void far termina_processo1();
```

No código fonte teste para núcleo básico possuirá quatro funções, cada uma representando um processo distinto. Esses processos serão executados por um intervalo de tempo até que sofram preempção, o escalador do núcleo encontrará o próximo processo e assim por diante. Ou seja, os quatros processos irão ficar executando em intercalação até que todos terminem suas tarefas. Sendo que a tarefa de cada um é printar na tela o número de seu processo mil vezes.

```
#include<nucleo.h>
#include<stdio.h>
void far processo1(){
        int i=0;
        while (i<10000){
                printf("1");
```

```
7                   i++;
8           }
9           termina_processo1();
10  }
11  void far processo2(){
12          int i=0;
13          while (i<10000){
14                  printf("2");
15                  i++;
16          }
17          termina_processo1();
18  }
19  void far processo3(){
20          int i=0;
21          while (i<10000){
22                  printf("3");
23                  i++;
24          }
25          termina_processo1();
26  }
27  void far processo4(){
28          int i=0;
29          while (i<10000){
30                  printf("4");
31                  i++;
32          }
33          termina_processo1();
34  }
35  main(){
36          criar_processo(processo1, "p1");
37          criar_processo(processo2, "p2");
38          criar_processo(processo3, "p3");
39          criar_processo(processo4, "p4");
40          dispara_sistema();
41  }
```

5.2 Teste núcleo com semáforos

A biblioteca núcleo.h terá os escopos das seguintes funções para o usuário:

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

```
3  typedef struct desc_p{
4          char nomep[35];
5          enum{ativo, terminado} estado;
6          PTR_DESC contexto;
7          struct desc_p *prox_desc;
8  }DESCRITOR_PROC;
9  typedef DESCRITOR_PROC *PTR_DESC_PROC;
10 typedef struct {
11         int s;
12         PTR_DESC_PROC Q;
13 }semaforo;
14 extern PTR_DESC_PROC far procura_prox_ativo();
15 extern void far criar_processo(void far (*end_proc)(), char
     nome_proc[]);
16 extern void far escalador();
17 extern void far dispara_sistema();
18 extern void far termina_processo();
19 extern void far termina_processo1();
20 extern void far p(semaforo *sem);
21 extern void far v(semaforo *sem);
22 extern void far inicia_semaforo(semaforo *sem, int n);
```

No código fonte teste para núcleo com semáforos possuirá quatro funções, duas representando dois produtores e duas prepresentando dois consumidores. Todos os processos farão a condição de corrida em um buffer circular representado por um vetor. Para isso usarçao quatro semáforos: mutex representa se o buffer está em uso ou não; vazio representa se o buffer está vazio ou não; e cheio representa se o buffer está cheio ou não.

Esses processos serão executados por um intervalo de tempo até que sofram preempção, o escalador do núcleo encontrará o próximo processo e assim por diante. Ou seja, os produtores pruduzirão uma mensagem e colocarão do buffer até que sofram preempção, se o buffer estiver lotado eles continuam tentando inserir a mensagem do buffer, porém não consguirão. Enquanto os consumidores retirarão uma mensagem do buffer até que sofram preempção, se o buffer estiver vazio eles continuam tentando retirar a mensagem do buffer, porém não consguirão.

```
1  #include<nucleo.h>
2  #include<stdio.h>
3  FILE *arq;
4  int buffer[6], mensagem = 0, pont_prod = 0, pont_consum = 0, max
     = 6;
5  semaforo mutex, cheio, vazio;
6  void far deposita(int msg, int prod){
```

```
7         if((&vazio)->s < 0){
8                 fprintf(arq, "%s%d%s%d%s", "BUFFER CHEIO:
                     produtor",prod," nao inseriu mensagem ",msg,"\n
                     ");
9                 printf("BUFFER CHEIO: produtor%d nao inseriu
                     mensagem %d", prod, msg);
10        } else {
11                fprintf(arq, "%s%d%s%d%s", "produtor",prod,"
                     inseriu mesagem ",msg,"\n");
12                printf("produtor%d inseriu mesagem %d", prod, msg
                     );
13                mensagem++;
14                buffer[pont_prod++] = msg;
15                if(pont_prod > max)
16                        pont_prod = 0;
17        }
18}
19void far retira(int prod){
20        if((&cheio)->s > max){
21                fprintf(arq, "%s%d%s", "BUFFER VAZIO: consumidor"
                     , prod, "nao retirou mensagem\n");
22                printf("BUFFER VAZIO: consumidor%d nao retirou
                     mensagem", prod);
23        } else {
24                fprintf(arq, "%s%d%s%d%s", "consumidor",prod,"
                     retirou mesagem ",buffer[pont_consum],"\n");
25                printf("consumidor%d retirou mesagem %d", prod,
                     buffer[pont_consum]);
26                pont_consum++;
27                if(pont_consum > max)
28                        pont_consum = 0;
29        }
30}
31void far produtor1(){
32        int i = 0;
33        while(i < 10){
34                p(&vazio);
35                p(&mutex);
36                deposita(mensagem, 1);
37                v(&mutex);
38                v(&cheio);
39                i++;
```

```
40          }
41          termina_processo1();
42  }
43  void far produtor2(){
44          int i = 0;
45          while(i < 10){
46                  p(&vazio);
47                  p(&mutex);
48                  deposita(mensagem, 2);
49                  v(&mutex);
50                  v(&cheio);
51                  i++;
52          }
53          termina_processo1();
54  }
55  void far consumidor1(){
56          int i = 0;
57          while(i < 10){
58                  p(&cheio);
59                  p(&mutex);
60                  retira(1);
61                  v(&mutex);
62                  v(&vazio);
63                  i++;
64          }
65          termina_processo1();
66  }
67  void far consumidor2(){
68          int i = 0;
69          while(i < 10){
70                  p(&cheio);
71                  p(&mutex);
72                  retira(2);
73                  v(&mutex);
74                  v(&vazio);
75                  i++;
76          }
77          termina_processo1();
78  }
79  main(){
80          arq = fopen("C:RESUL.txt","w");
81          criar_processo(produtor1, "p1");
```

```
82          criar_processo(produtor2, "p2");
83          criar_processo(consumidor1, "c1");
84          criar_processo(consumidor2, "c2");
85          inicia_semaforo(&cheio, 0);
86          inicia_semaforo(&vazio, max);
87          inicia_semaforo(&mutex, 1);
88          dispara_sistema();
89          fclose(arq);
90  }
```

5.3 Teste núcleo com troca de mensagens

A biblioteca núcleo.h terá os escopos das seguintes funções para o usuário:

```
1  #include <system.h>
2
3  typedef struct desc_p {
4    char nome[35];
5    enum {ativo, terminado} estado;
6    PTR_DESC contexto;
7    struct desc_p *prox_desc;
8  } DESCRITOR_PROC;
9
10 typedef DESCRITOR_PROC *PTR_DESC_PROC;
11
12 typedef struct {
13   int s;
14   PTR_DESC_PROC Q;
15 } semaforo;
16
17 extern void far criar_Processo(char nome_proc[35], void far(*
     end_proc)(), int max_fila);
18 extern void far dispara_sistema();
19 extern void far terminar_Processo();
20 extern void far inicia_semaforo(semaforo *sem, int n);
21 extern void far P(semaforo *sem);
22 extern void far V(semaforo *sem);
23 extern int far envia(char *nome_destino, char *msg);
24 extern void far recebe(char *msg, char *p_emissor);
25 extern void far recebe_seletivo(char *nome_processo, char *msg,
     char *p_emissor);
```

No código dos testes, são criados 9 processos. Os processos 1, 2 e 3 formam um ciclo de envio e recebimento:

- O processo 1 envia uma mensagem ao processo 2 e recebe uma mensagem do processo 3
- O processo 2 recebe uma mensagem do processo 1 e a envia ao processo 3
- O processo 3 recebe uma mensagem do processo 2 e a envia ao processo 1

O processo 1 também envia uma mensagem ao processo 4, que receberá, seletivamente, apenas as mensagens enviadas pelo processo 1. Também é feita uma simulação de erro no início do processo 1 ao tentar enviar uma mensagem para um processo inexistente.

O processo 5 tenta enviar mensagens ao processo 4. Como o processo 4 recebe apenas mensagens do processo 1, espera-se que o processo 5 permaneça bloqueado após chamar a função envia().

Os processos 6, 7 e 8 enviam mensagens ao processo 9, que irá recebê-las. Tais processos são utilizados para testar o armazenamento das mensagens na lista correspondente. Os processos emissores ficam em um laço tentando enviar a mensagem até que exista espaço na fila.

Todos os processos escrevem uma linha em um arquivo especificado sempre que enviam ou recebem alguma mensagem. Tal linha contém o nome dos processos envolvidos e a mensagem enviada ou recebida.

```
1  #include <stdio.h>
2  #include <nucleo.h>
3
4  FILE *arquivo;
5
6  void far processo1() {
7    int i = 0, codigo_erro;
8    char msg[30], emissor[30];
9
10   codigo_erro = envia("P10", "mensagem enviada a um processo
       inexistente");
11   if(codigo_erro != 1) {
12     fprintf(arquivo, "Erro esperado mas nao obtido ao tentar
         enviar uma mensagem a um processo inexistente\n");
13   }
14
15   while(i < 100) {
```

```
16      sprintf(msg, "%d", i);
17      fprintf(arquivo, "Processo 1 enviando para P2:  %s\n", msg);
18      envia("P2", msg);
19
20      sprintf(msg, "%d", i * 10);
21      fprintf(arquivo, "Processo 1 enviando para P4:  %s\n", msg);
22      envia("P4", msg);
23
24      recebe(msg, emissor);
25      fprintf(arquivo, "Processo 1 recebeu de %s: %s\n", emissor,
            msg);
26      i++;
27    }
28
29    terminar_Processo();
30  }
31
32  void far processo2() {
33    int i = 0;
34    char msg[30], emissor[30];
35
36    while(i < 100) {
37      recebe(msg, emissor);
38      fprintf(arquivo,"Processo 2 recebeu de %s: %s\n", emissor,
            msg);
39
40      fprintf(arquivo, "Processo 2 enviando para P3: %s\n", msg);
41      envia("P3", msg);
42      i++;
43    }
44
45    terminar_Processo();
46  }
47
48  void far processo3() {
49    int i = 0;
50    char msg[30], emissor[30];
51
52    while(i < 100) {
53      recebe(msg, emissor);
54      fprintf(arquivo, "Processo 3 recebeu de %s: %s\n", emissor,
            msg);
```

```
55
56      fprintf(arquivo, "Processo 3 enviando para P1: %s\n", msg);
57      envia("P1", msg);
58      i++;
59    }
60
61    terminar_Processo();
62  }
63
64  void far processo4() {
65    int i = 0;
66    char msg[30], emissor[30];
67
68    while(i < 100) {
69      recebe_seletivo("P1", msg, emissor);
70      fprintf(arquivo, "Processo 4 recebeu de %s: %s\n", emissor,
          msg);
71      i++;
72    }
73
74    terminar_Processo();
75  }
76
77  void far processo5() {
78    int i = 0;
79    char msg[30], emissor[30];
80
81    while(i < 100) {
82      envia("P4", "essa mensagem nao sera recebida");
83      i++;
84    }
85
86    terminar_Processo();
87  }
88
89  void far processo6() {
90    int i = 0, codigo_erro = -1;
91    char msg[30], emissor[30];
92
93    while(i < 100) {
94      sprintf(msg, "%d", i + 500);
95      fprintf(arquivo, "Processo 6 enviando para P9:  %s\n", msg);
```

```
96
97      codigo_erro = envia("P9", msg);
98
99      while(codigo_erro != 0) {
100       codigo_erro = envia("P9", msg);
101     }
102
103     i++;
104   }
105
106   terminar_Processo();
107 }
108
109 void far processo7() {
110   int i = 0, codigo_erro = -1;
111   char msg[30], emissor[30];
112
113   while(i < 100) {
114     sprintf(msg, "%d", i + 500);
115     fprintf(arquivo, "Processo 7 enviando para P9:  %s\n", msg);
116
117     codigo_erro = envia("P9", msg);
118
119     while(codigo_erro != 0) {
120       codigo_erro = envia("P9", msg);
121     }
122
123     i++;
124   }
125
126   terminar_Processo();
127 }
128
129 void far processo8() {
130   int i = 0, codigo_erro = -1;
131   char msg[30], emissor[30];
132
133   while(i < 100) {
134     sprintf(msg, "%d", i + 500);
135     fprintf(arquivo, "Processo 8 enviando para P9:  %s\n", msg);
136
137     codigo_erro = envia("P9", msg);
```

```
138
139     while(codigo_erro != 0) {
140       codigo_erro = envia("P9", msg);
141     }
142
143     i++;
144   }
145
146   terminar_Processo();
147 }
148
149 void far processo9() {
150   int i = 0;
151   char msg[30], emissor[30];
152
153   while(i < 300) {
154     recebe( msg, emissor);
155     fprintf(arquivo, "Processo 9 recebeu de %s: %s\n", emissor,
          msg);
156     i++;
157   }
158
159   terminar_Processo();
160 }
161
162 main() {
163   criar_Processo("P1", processo1, 10);
164   criar_Processo("P2", processo2, 10);
165   criar_Processo("P3", processo3, 10);
166   criar_Processo("P4", processo4, 10);
167   criar_Processo("P5", processo5, 10);
168   criar_Processo("P6", processo6, 10);
169   criar_Processo("P7", processo7, 10);
170   criar_Processo("P8", processo8, 10);
171   criar_Processo("P9", processo9, 10);
172
173   arquivo = fopen("saida.txt", "w");
174
175   dispara_sistema();
176 }
```

5.4 Teste núcleo com prioridades

A biblioteca nucleo.h terá os escopos das seguintes funções para o usuário:

```
#include <system.h>

typedef struct desc_p {
  char nome[35];
  enum {ativo, terminado} estado;
  PTR_DESC contexto;
  struct desc_p *prox_desc;
} DESCRITOR_PROC;

typedef DESCRITOR_PROC *PTR_DESC_PROC;

extern void far criar_Processo(char nome_proc[35], void far(*
   end_proc)(), int prioridade);
extern void far dispara_sistema();
extern void far terminar_Processo();
```

No código dos testes são criados um processo com prioridade 4, dois processos com prioridade 2 e um processo com prioridade 0. Cada processo escreve um número único em um arquivo especificado 10000 vezes e, ao seu término, uma mensagem única indicando seu fim.

É esperado que o processo com prioridade 4 seja o primeiro a terminar, que os dois processos com prioridade 2 terminem em tempos próximos e que o processo com prioridade 0 seja o último a terminar. Além disso, espera-se que a ordem das execuções esteja de acordo com as manipulações das listas feitas pelo escalador.

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

FILE *arquivo;

void far processo1() {
  int i = 0;
  while(i < 10000) {
    fprintf(arquivo, "1");
    i++;
  }

  fprintf(arquivo, " Processo 1 terminando ");
  terminar_Processo();
```

```
15 }
16
17 void far processo2() {
18   int i = 0;
19   while(i < 10000) {
20     fprintf(arquivo, "2");
21     i++;
22   }
23
24   fprintf(arquivo, " Processo 2 terminando ");
25   terminar_Processo();
26 }
27
28 void far processo3() {
29   int i = 0;
30   while(i < 10000) {
31     fprintf(arquivo, "3");
32     i++;
33   }
34
35   fprintf(arquivo, " Processo 3 terminando ");
36   terminar_Processo();
37 }
38
39 void far processo4() {
40   int i = 0;
41   while(i < 10000) {
42     fprintf(arquivo, "4");
43     i++;
44   }
45
46   fprintf(arquivo, " Processo 4 terminando ");
47   terminar_Processo();
48 }
49
50 main() {
51   criar_Processo("P1", processo1, 4);
52   criar_Processo("P2", processo2, 2);
53   criar_Processo("P3", processo3, 2);
54   criar_Processo("P4", processo4, 0);
55
56   arquivo = fopen("saida.txt", "w");
```

```
57
58    dispara_sistema();
59 }
```