

**UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"**  
FACULDADE DE CIÊNCIAS - CAMPUS BAURU  
DEPARTAMENTO DE COMPUTAÇÃO  
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DAVI AUGUSTO NEVES LEITE  
LUIZ FERNANDO MERLI DE OLIVEIRA SEMENTILLE

**DOCUMENTAÇÃO DO NÚCLEO MULTITAREFA COM  
SINCRONIZAÇÃO PARA MS-DOS**

BAURU  
2021

DAVI AUGUSTO NEVES LEITE  
LUIZ FERNANDO MERLI DE OLIVEIRA SEMENTILLE

**DOCUMENTAÇÃO DO NÚCLEO MULTITAREFA COM  
SINCRONIZAÇÃO PARA MS-DOS**

Projeto apresentado ao Curso do Curso de Ciência da Computação da Universidade Estadual Paulista “Júlio de Mesquita Filho”, Faculdade de Ciências, Campus Bauru, como requisito parcial para aprovação na disciplina de Sistemas Operacionais II.

Orientador: Prof. Dr. Antonio Carlos Sementille

# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>3</b>
<b>2</b>	<b>PASTA "EXERCÍCIOS"</b>	<b>6</b>
<b>2.1</b>	<b>TICTAC.C</b>	<b>6</b>
2.1.1	Estrutura de Dados	6
2.1.2	Co-Rotinas e Algoritmos	7
<b>2.2</b>	<b>TICTAC2.C</b>	<b>8</b>
2.2.1	Estrutura de Dados	9
2.2.2	Co-Rotinas e Algoritmos	9
<b>2.3</b>	<b>ESCALA.C</b>	<b>10</b>
2.3.1	Estrutura de Dados	10
2.3.2	Co-Rotinas e Algoritmos	11
<b>3</b>	<b>PASTA "NÚCLEO"</b>	<b>13</b>
<b>3.1</b>	<b>KERNEL.C</b>	<b>14</b>
3.1.1	Estrutura de Dados	14
3.1.2	Co-Rotinas e Algoritmos	17
3.1.3	Kernel.H	22
<b>3.2</b>	<b>TK-01.C</b>	<b>24</b>
3.2.1	Estrutura de Dados	24
3.2.2	Co-Rotinas e Algoritmos	24
<b>3.3</b>	<b>TK-02.C</b>	<b>27</b>
3.3.1	Estrutura de Dados	29
3.3.2	Co-Rotinas e Algoritmos	29

# 1 Introdução

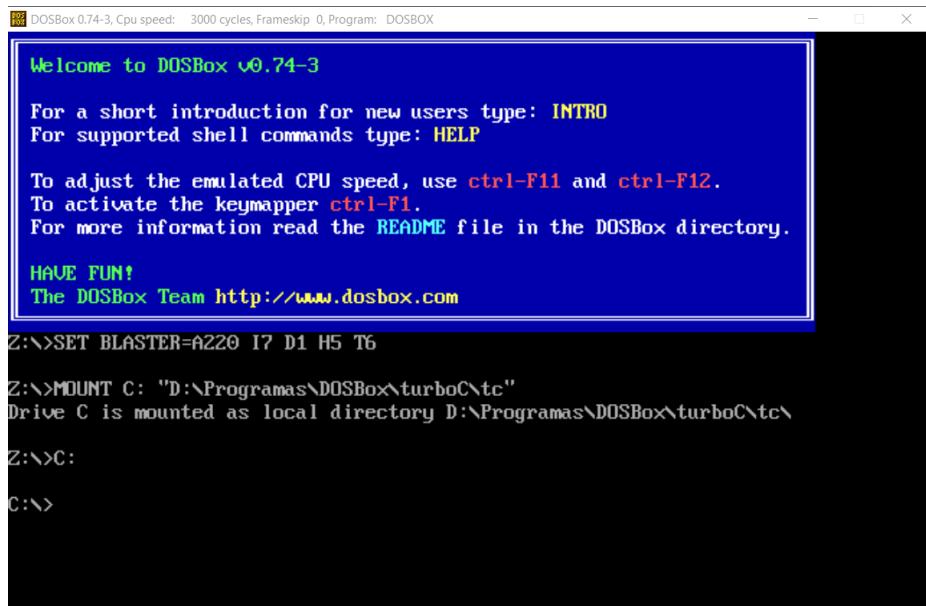
Um núcleo trata-se de um dos pilares para o funcionamento de um Sistema Operacional, haja visto que compõe toda a parte de gerenciamento de hardware e software primordiais para permitir a comunicação entre usuário e sistema. Para tanto, ele deve permitir seguramente que o usuário acesse a um conjunto de funções específicas, como aquelas relacionadas ao gerenciamento de processos, ou seja, tanto a criação e encerramento de processos quanto a comunicação entre processos (por meio de semáforos, por exemplo). Além disso, outras funções cobertas pelo núcleo de um Sistema Operacional incluem: gerenciamento de recursos (memória e dispositivos de entrada e saída, por exemplo) e gerenciamento de UCP (relacionados à manipulação de registradores específicos, como os de interrupção e *flags*).

Desta forma, a construção de um núcleo trata-se de uma das etapas mais importantes e complexas na construção de um Sistema Operacional, haja vista que deve levar em conta diversos aspectos tanto do hardware em que será utilizado quanto do software em que se pretende executar na máquina, de tal forma que possa haver um fluxo de controle viável e otimizado na execução dos processos e que garanta a interatividade do sistema com o usuário (uma das principais funções de um Sistema Operacional).

Em meio a isso, esse trabalho é dividido em duas partes: a primeira está relacionada a exemplos dos funcionamentos do gerenciamento de processos por um Sistema Operacional monoprogramado (aquele em que somente existe um processo em execução), cuja pasta associada foi denominada de “Exercícios”; e a segunda propõe a criação de um núcleo multiprogramado (aquele em que há mais de um processo em execução, sendo utilizado um fator de tempo associado às interrupções e denominado de *quantum*) e comunicável entre processos, por meio de semáforos, de tal maneira a facilitar didaticamente os estudos aplicados relacionados ao tema de Sistemas Operacionais, cuja pasta leva o mesmo nome (“Núcleo”). Ainda, para a segunda parte, foi proposto a criação de dois programas testes: o primeiro contendo cinco rotinas de escrita na tela, a fim de demonstrar a funcionalidade básica do núcleo; e o segundo contendo duas rotinas para demonstrar uma solução para o famoso problema do Produtor/Consumidor, por meio da utilização de semáforos.

Dessa forma, esse núcleo multitarefa possui funções básicas tanto para criação e encerramento de processos quanto para a criação e manipulação de semáforos (segundo as primitivas estabelecidas por Dijkstra), sendo associado um arquivo de cabeçalho para a simples importação e utilização dessas funções pelo usuário. Em meio a isso, o trabalho foi construído a partir do emulador DOSBox v.0.74-3 e do programa Borland Turbo C v2.

O DOSBox, como o próprio nome sugere, é um emulador do Sistema Operacional MS-DOS a partir da arquitetura de um computador com o Intel 8086 (16 bits). A utilização deste Sistema Operacional **monoprogramado** (ou seja, não há a possibilidade de se utilizar um mesmo dispositivo de E/S por diferentes processos de maneira corrente - como na multiprogramação) está associado ao gerenciamento de memória e no enfoque por processo, ou seja, o núcleo proposto no trabalho utiliza o gerenciamento de memória já existente do MS-DOS emulado e dá um enfoque ao gerenciamento de processos de maneira multiprogramada. Mais informações a respeito do emulador, bem como o link de download, podem ser encontrados no site oficial em: <https://www.dosbox.com/download.php?main=1>.



Já o Borland Turbo C, em sua versão 2.x, nada mais é do que um ambiente de desenvolvimento integrado (IDE) para a linguagem de programação ANSI C. Ou seja, toda a parte programável do núcleo e dos exemplos foi realizada na linguagem C e suporta apenas o DOSBox, já que essa linguagem leva em consideração o Sistema Operacional durante a fase de compilação. Em outras palavras, um programa compilado em C no DOSBox (MS-DOS) não pode ser executado em um Linux, já que são Sistemas Operacionais distintos. Portanto, tanto os exemplos quanto o núcleo **somente** podem ser executados e demonstrados no DOSBox com a utilização do Turbo C. Por fim, essa IDE foi modificada para comportar a parte multitarefa do núcleo, ou seja, criou-se códigos-fontes (entre eles, o principal denominado *SYSTEM.C*) que manipulam as funções de sistema do MS-DOS para o gerenciamento de processos do núcleo. Mais informações a respeito dessa IDE, bem como o link de download, podem ser encontrados no seguinte site: <https://winworldpc.com/product/borland-turbo-c/2x>.

DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: TC

**File Edit Run Compile Project Options Debug Break/watch**

**Compile to OBJ C:KERNEL.OBJ**  
**Make EXE file C:KERNEL.EXE**  
**Link EXE file**  
**Build all**  
**Primary C file:**  
**Get info**

C:KERNEL.C

```

Line 1 Col
#include <system.h>
#define MAX_NAME_PROC
#define INTERRUPT_BIT
/* **** */
/* Definicao dos Registradores do Intel_8086 para Regiao Critica */
/* **** */
typedef struct registers_8086{
    unsigned bx1, es1; /* Representa os registradores da flag de servicos do DO
}REG_8086;

/* **** */
/* Definicao da Regiao Critica, com base nos Registradores */
/* **** */

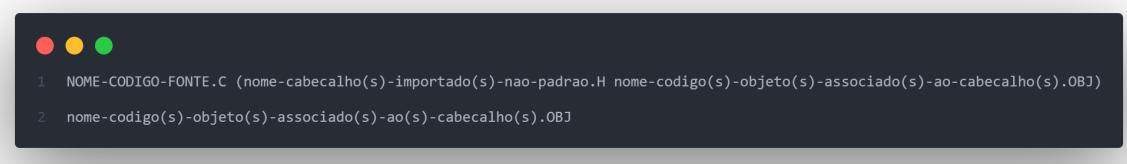
/* Uso de uma "union" para juntar ambas as partes, utilizando um ponteiro para
typedef union critical_region{

```

Message

**F1-Help F5-Zoom F6-Switch F7-Trace F8-Step F9-Make F10-Menu**

Por fim, por conta da utilização do Turbo C como IDE, foi-se necessário a criação de arquivos com extensão **.PRJ** para cada código-fonte presente neste trabalho, de tal forma que permitissem a **linkagem** (momento de criar o executável a partir do código objeto - aquele que é composto por uma sequência de instruções binárias - sendo realizado a junção das bibliotecas importadas em cada código-fonte) dos programas realizados. Em outras palavras, os arquivos com essa extensão permitem ao Turbo C identificar e juntar as bibliotecas **fora da sua predefinição** (como à *SYSTEM.H* que utiliza o *SYSTEM.OBJ* criada especificamente para o núcleo e que não é padrão como a *STDIO.H*) nos códigos-objetos. O escopo de um arquivo **.PRJ** pode ser visualizado na imagem abaixo.



## 2 Pasta "Exercícios"

Este capítulo está relacionado aos exemplos funcionais do gerenciamento de processos do MS-DOS, utilizando as rotinas básicas estabelecidas no *SYSTEM.C*. Para tanto, em cada código-fonte, foi necessário a importação das bibliotecas como demonstra a imagem abaixo:



Existem três programas na pasta associada: *TICTAC*, *TICTAC2* (ou *TICTAC MODIFICADO*) e *ESCALA* (relacionado ao escalonador). Cada qual será explicado em seu tópico específico a seguir.

### 2.1 TICTAC.C

Esse programa trata-se do primeiro exemplificador do gerenciamento básico de processos do MS-DOS, por meio da composição de duas co-rotinas denominadas “tic” e “ tac”, as quais imprimem indefinidamente os textos “tic” e “ tac” na tela, respectivamente. Dessa maneira, o controle é feito diretamente pelo DOSBox, sendo possível ver as funções básicas da criação de processos e de troca de contexto (isto é, a transferência de controle de uma função para outra).

#### 2.1.1 Estrutura de Dados

Para a implementação deste programa foi necessário a utilização de apenas uma estrutura de dados: a do ponteiro para um descriptor de processo (presente no *SYSTEM.C*). Isto nada mais é do que um ponteiro cujo conteúdo está associado ao contexto (co-rotina) do processo. Como pode ser visto na imagem abaixo, existe um ponteiro de descriptor para cada rotina do código-fonte.

```
4  /* Criando ponteiros para corotinas */
5  PTR_DESC dmain, dtic, dtac;
```

### 2.1.2 Co-Rotinas e Algoritmos

A primeira rotina presente no código é a “tic”, sem retorno. Ela está associada a uma impressão na tela, de maneira indefinida (basta ver a presença da estrutura de repetição while com o seu argumento sempre em verdadeiro), do texto de mesmo nome da rotina. Após exibir o texto na tela, o contexto de execução do sistema, isto é, o controle é transferido para a co-rotina “tac” por meio da função auxiliar “*transfer*” (presente em *SYSTEM.C*), sendo possível executá-la. A função “tic” pode ser vista na imagem abaixo.

```
7  void far tic(){
8      while(1){
9          printf("tic");
10         /* Transferindo controle para a corotina "tac" */
11         transfer(dtic, dtac);
12     }
13 }
```

Sendo assim, o semelhante irá acontecer na próxima rotina do código, denominada “tac”. Ou seja, a função irá imprimir na tela um texto com o nome da rotina e transferir o controle para a co-rotina “tic”, de tal maneira que o gerenciamento de processos mostre a “intercalação” de rotinas. Em outras palavras, uma rotina deve ser executada por vez e de maneira indefinida por essa abordagem básica e sem tratamento de problemas relacionados a E/S. A função “tac” pode ser vista na imagem abaixo.

```

15 void far tac(){
16     while(1){
17         printf("tac");
18         /* Transferindo controle para a corotina "tic" */
19         transfer(dtac, dtic);
20     }
21 }
```

E como é de costume na linguagem C, existe a função “main” responsável por iniciar as instruções do programa ao executá-lo na máquina. Desta forma, essa função é responsável por criar os ponteiros dos descritores, com a utilização da rotina “cria\_desc” presente em *SYSTEM.C*, e iniciar os processos nesses descritores, com a utilização da rotina “newprocess” também presente em *SYSTEM.C*. Além disso, para dar uma inicialização na indefinição entre as rotinas “tic” e “tac”, a “main” é responsável por transferir o controle de processo para uma das rotinas. Para maneiras didáticas, a rotina “tic” foi escolhida para ser executada primeiramente. Por fim, a imagem abaixo representa a função “main”, a qual não é necessária a criação de um processo (com o “newprocess”) já que isso é feito pelo próprio MS-DOS ao executar o programa.

```

23 main(){
24     /* Inicializando descritores (ponteiros) */
25     dmain = cria_desc();
26     dtic = cria_desc();
27     dtac = cria_desc();
28     /* Criando processos associados as corotinas. A do main ja eh realizada pelo DOS. */
29     newprocess(tic, dtic);
30     newprocess(tac, dtac);
31     /* Iniciando corotina "tic", ou seja, transferindo o controle para a corotina "tic" */
32     transfer(dmain, dtic);
33 }
```

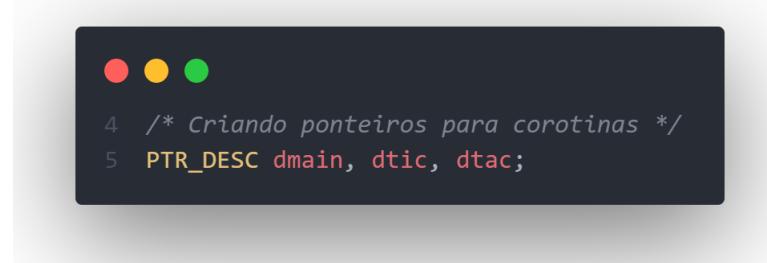
## 2.2 TICTAC2.C

Semelhante ao programa anterior, o “*TICTAC2.C*” (ou *TICTAC.C MODIFICADO*) implementa o mesmo conjunto de co-rotinas “tic”, “tac” e “main”. Contudo, neste caso há a seguinte modificação com relação ao código-fonte *TICTAC.C*: a rotina “tic” é limitada para a realização de cem vezes na impressão do texto de seu nome na tela, tendo o controle transferido novamente para a “main” no fim dela. Com isso, é possível observar as restrições de ocorrência

em um processo e na finalização dos processos (haja visto que o programa anterior somente permitia a execução indefinida dos processos “tic” e “tac”).

### 2.2.1 Estrutura de Dados

Novamente, necessitou-se da mesma estrutura de dados de “ponteiro para descritor de processo” para a implementação deste programa. Isso pode ser visto na imagem abaixo.



```

● ● ●
4 /* Criando ponteiros para corotinas */
5 PTR_DESC dmain, dtic, dtac;

```

### 2.2.2 Co-Rotinas e Algoritmos

As rotinas estabelecidas neste código-fonte são praticamente as mesmas do anterior. Contudo, há pequenas modificações na rotina “tic” e ao fim da rotina “main”. No primeiro caso, existe a declaração de uma variável local inteira, com início em zero, para o controle (limitação) da execução da rotina “tic” e consequentemente da “tac”, já que o controle de rotinas é passado para “tac” dentro da estrutura de repetição limitada. Desta forma, a tela deve mostrar o texto “tictac” cem vezes e transferir o controle para a rotina “main”. As co-rotinas “tic” e “tac” podem ser vistas abaixo.

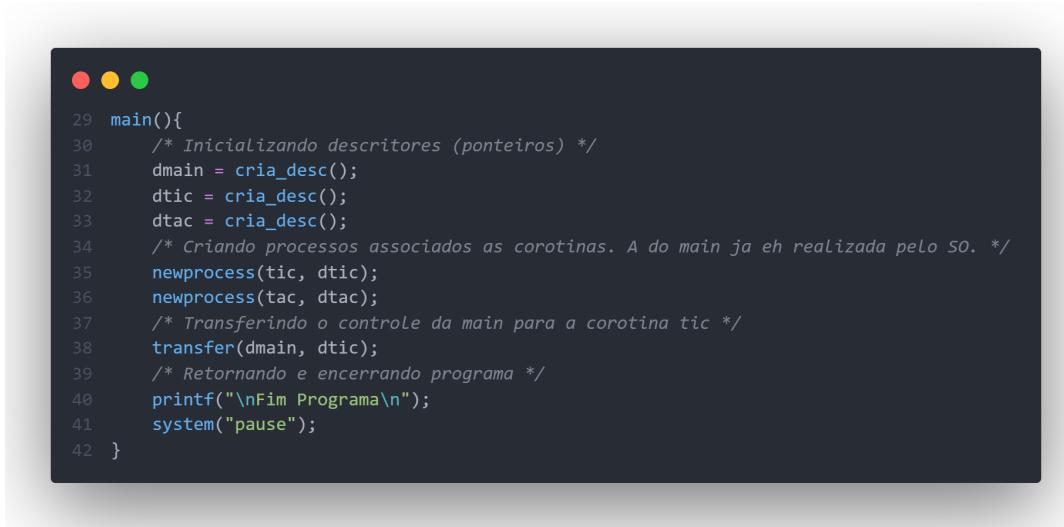


```

● ● ●
7 void far tic(){
8     /* Limitacao em 100 vezes */
9     int i = 0;
10    while(i < 100){
11        printf("tic");
12        /* Transferindo controle para a corotina "tac" */
13        transfer(dtic, dtac);
14        /* Incrementando "i" para Limitacao */
15        i++;
16    }
17    /* Transferindo controle para a corotina MAIN */
18    transfer(dtic, dmain);
19 }
20
21 void far tac(){
22     while(1){
23         printf("tac");
24         /* Transferindo controle para a corotina "tic" */
25         transfer(dtac, dtic);
26     }
27 }

```

Como citado acima, a rotina “main” sofreu uma pequena alteração em seu final: foi inserido uma impressão do texto “Fim Programa”, para indicar o fim da execução das rotinas “tic” e “tac”, e a função “pause” sendo chamada pelo sistema. Essa função está associada a travar o programa corrente no ponto em que foi chamada, sendo possível visualizar os resultados obtidos no terminal até o aperto de uma tecla qualquer. A imagem da nova co-rotina “main” pode ser visualizada abaixo.



```

29 main(){
30     /* Inicializando descritores (ponteiros) */
31     dmain = cria_desc();
32     dtic = cria_desc();
33     dtac = cria_desc();
34     /* Criando processos associados as corotinas. A do main ja eh realizada pelo SO. */
35     newprocess(tic, dtic);
36     newprocess(tac, dtac);
37     /* Transferindo o controle da main para a corotina tic */
38     transfer(dmain, dtic);
39     /* Retornando e encerrando programa */
40     printf("\nFim Programa\n");
41     system("pause");
42 }

```

## 2.3 ESCALA.C

Por fim, o último programa realizado fora denominado de *ESCALA* por conta de sua principal função: apresentar um pequeno escalonador de processos, o qual utiliza funções de transferência de contexto de um processo ao outro com base na interrupção do timer. Ou seja, para cada processo é dada uma fatia de tempo (denominada de *quantum*) e o contexto é trocado para outro processo, pelo escalonador, ao término desse tempo. Para tanto, esse programa possui duas co-rotinas básicas que imprimem o seu nome na tela indefinidamente, tendo uma rotina principal denominada de “escalador” cuja função é trocar o contexto de execução, levando em conta a interrupção do timer do DOSBox.

### 2.3.1 Estrutura de Dados

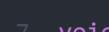
Igualmente aos programas *TICTAC* e *TICTAC2*, o *ESCALA* possui a mesma estrutura de dados de “ponteiro para descriptor de processo” (utilizada para cada rotina) em sua implementação, como pode ser visto na imagem abaixo.



```
4 /* Criacao dos Ponteiros dos Descritores das corotinas associadas */
5 PTR_DESC dmain, dcorA, dcorB, descala;
```

### 2.3.2 Co-Rotinas e Algoritmos

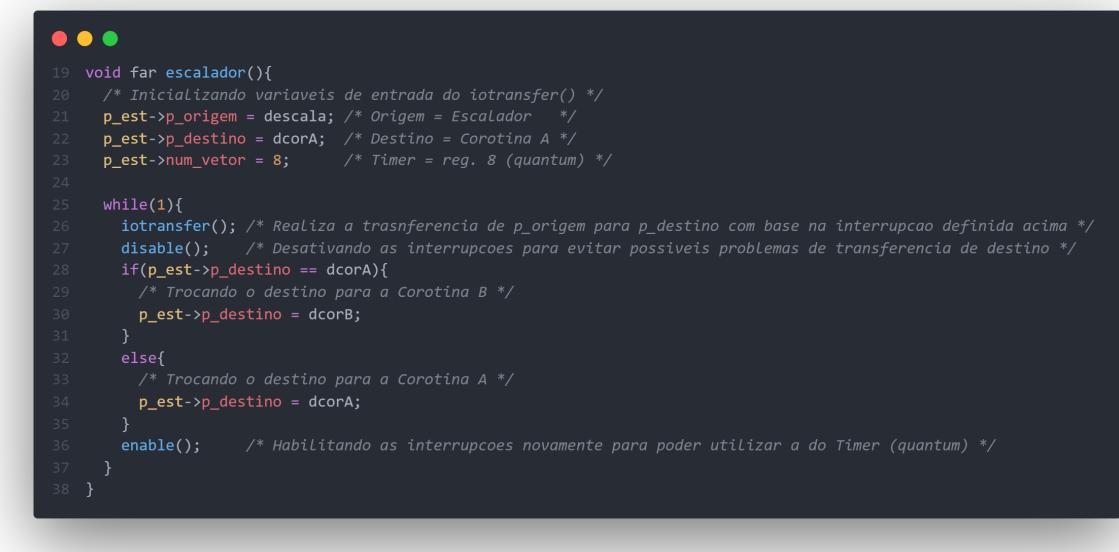
Inicialmente, o código contém duas sub-rotinas bem simples, denominadas de “*corotina\_A*” e “*corotina\_B*” e que possuem a simples função de imprimir os vossos nomes na tela do usuário, de maneira indefinida (basta visualizar as estruturas de repetição com as condições sempre em verdadeiro). Ambas podem ser visualizadas na imagem abaixo.



```
7 void far corotina_A(){
8     while(1){
9         printf("COROTINA-A");
10    }
11 }
12
13 void far corotina_B(){
14     while(1){
15         printf("COROTINA-B");
16    }
17 }
```

Contudo, a próxima rotina é a peça-chave desse programa: o “*escalador*”. Essa importante rotina possui a principal função de controlar o contexto dos processos estabelecidos pelas duas rotinas anteriores. Dessa forma, é possível visualizar a parte da multiprogramação do núcleo, com base em uma quantidade de tempo dada pela interrupção do *timer* do processador Intel 8086. Para tanto, essa rotina utiliza as seguintes funções estabelecidas em *SYSTEM.C*: a primeira delas, denominada de “*iotransfer*”, realiza a transferência de contexto origem de um processo para um contexto destino; a segunda, denominada de “*disable*”, possui a principal função de desabilitar todas as interrupções, de tal forma que não haja problemas na troca de contexto de processos; e, por fim, a “*enable*” ativa todas as interrupções novamente, para que o *timer* possa continuar ser utilizado como limitante dos processos. Além disso, como é possível ver no início da rotina, é necessário inicializar as variáveis de entrada da função “*iotransfer*”, uma vez que ela necessita de um contexto origem para um contexto destino.

Neste caso, a origem é o escalonador que irá iniciar a “*corotina\_A*”. Por fim, é possível ver que há uma estrutura de decisão dentro de uma estrutura de repetição indefinida. Ela nada mais simboliza do que a troca de destino de contexto, indo para a “*corotina\_B*” após o *quantum* da “*corotina\_A*” acabar e vice-versa. A imagem abaixo representa a função final do escalonador de processos deste programa.



```

19 void far escalador(){
20     /* Inicializando variaveis de entrada do iotransfer() */
21     p_est->p_origem = descala; /* Origem = Escalonador */
22     p_est->p_destino = dcorA; /* Destino = Corotina A */
23     p_est->num_vetor = 8;      /* Timer = reg. 8 (quantum) */
24
25     while(1){
26         iotransfer(); /* Realiza a trasnferencia de p_origem para p_destino com base na interrupcao definida acima */
27         disable();      /* Desativando as interrupcoes para evitar possiveis problemas de transferencia de destino */
28         if(p_est->p_destino == dcorA){
29             /* Trocando o destino para a Corotina B */
30             p_est->p_destino = dcorB;
31         }
32         else{
33             /* Trocando o destino para a Corotina A */
34             p_est->p_destino = dcorA;
35         }
36         enable();       /* Habilitando as interrupcoes novamente para poder utilizar a do Timer (quantum) */
37     }
38 }
```

Por fim, há a mesma rotina “*main*” semelhante aos outros programas citados anteriormente, cuja função consiste em inicializar os ponteiros de descritores e aos processos propriamente ditos, trocando o contexto inicial de processo para o processo do escalonador. Desta forma, com o controle sendo transferido para o escalonador, é possível observar o gerenciamento de processos concorrentes, tendo como base uma fatia de tempo do *timer*. A imagem abaixo retrata a função “*main*” final.



```

40 main(){
41     /* Inicializando descritores (ponteiros) */
42     dmain = cria_desc();
43     dcorA = cria_desc();
44     dcorB = cria_desc();
45     descala = cria_desc();
46
47     /* Criando processos associados as corotinas. A do main ja eh realizada pelo SO. */
48     newprocess(corotina_A, dcorA);
49     newprocess(corotina_B, dcorB);
50     newprocess(escalonador, descala);
51
52     /* Transferindo o controle da main para o escalonador */
53     transfer(dmain, descala);
54
55     printf("\nFim Programa\n");
56     system("pause");
57 }
```

### 3 Pasta "Núcleo"

Este capítulo está relacionado a criação de um núcleo multiprogramado e comunicável entre processos. Para tanto, utilizou-se um fator de tempo (*quantum*), associado à interrupção do *timer* (registrador de número 8 do Intel 8086) para limitar a execução de cada processo, bem como a utilização de semáforos para a comunicação entre processos. Para tanto, utilizou-se as seguintes bibliotecas associadas:

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

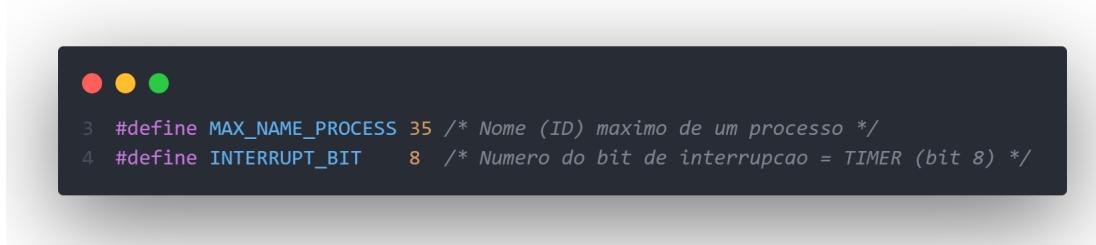
Além disso, de maneira a demonstrar todas as funcionalidades básicas implementadas, criou-se dois programas de testes, denominados *TK-01* e *TK-02*, os quais foram associados para cada objetivo do núcleo. Em outras palavras, o *TK-01* possui rotinas para testes associados à multiprogramação do núcleo e ao fator de tempo; ao passo que o *TK-02* possui rotinas para testes associados à comunicação entre processos, por meio da representação do problema clássico do Produtor/Consumidor. Para tanto, as seguintes bibliotecas foram necessárias em ambos programas de testes:

```
4 #include <kernel.h>
5 #include <stdio.h>
6 #include <stdlib.h>
```

Como visto na imagem acima, houve também a criação de um arquivo de cabeçalho do núcleo, de tal forma que as funções visíveis ao usuário comum estejam disponíveis para acesso. Com esse arquivo, é possível apenas importar o núcleo como uma biblioteca comum da linguagem C e, assim, acessar as funcionalidades implementadas. Mais detalhes a respeito do núcleo, bem como dos programas testes, serão dados em seus respectivos tópicos deste capítulo.

## 3.1 KERNEL.C

Conforme estabelecido na introdução, este programa se trata do núcleo multiprogramado e comunicável entre processos, de tal forma que possa facilitar didaticamente os estudos aplicados relacionados ao tema de Sistemas Operacionais. Para tanto, o *KERNEL.C* é separado em diversas funções, as quais serão descritas e explicadas neste capítulo. Com relação às funções, foi-se necessário a criação de duas definições gerais no topo do código, sendo representadas por *MAX\_NAME\_PROCESS* e *INTERRUPT\_BIT*, as quais simbolizam o tamanho máximo para nome de um processo e o *bit* de interrupção do processador Intel 8086 (*timer*), respectivamente. A imagem abaixo demonstra a criação das definições, bem como seus respectivos valores associados.



```

3 #define MAX_NAME_PROCESS 35 /* Nome (ID) maximo de um processo */
4 #define INTERRUPT_BIT     8 /* Numero do bit de interrupcao = TIMER (bit 8) */

```

### 3.1.1 Estrutura de Dados

No núcleo (*KERNEL.C*), foram utilizadas as seguintes estruturas de dados: *registers\_8086*; *critical\_region*; *pcb* e *semaphore*.

A estrutura ***registers\_8086*** está relacionada aos registradores da *flag* de serviços do DOS e foi utilizada durante o programa para verificar se uma interrupção do *timer* ocorreu enquanto o programa estava executando uma rotina de serviço do DOS, que pode ser potencialmente perigosa caso seja interrompida. Os registradores de *flag* utilizados foram o *BX* e o *ES* cuja sua combinação indica se a pilha do DOS está sendo utilizada, isto é, caso o DOS é quem esteja executando alguma rotina. Ainda nessa questão, a estrutura ***critical\_region*** foi utilizada em combinação com a ***registers\_8086*** para armazenar o resultado dos registradores *BX* e *ES*. A imagem abaixo representa essa estrutura.

```

6  /* ****!*\
7  /* Definicao dos Registradores do Intel_8086 para Regiao Critica */
8  /* ****|^
9  typedef struct registers_8086{
10    unsigned bx1, es1; /* Representa os registradores da flag de servicos do DOS */
11 }REG_8086;
12
13 /* ****|^
14 /* Definicao da Regiao Critica, com base nos Registradores */
15 /* ****|^
16
17 /* Uso de uma "union" para juntar ambas as partes, utilizando um ponteiro para indicar o valor */
18 typedef union critical_region{
19   REG_8086 reg;      /* Representa o resultado dos registradores BX e ES (flag de servicos do DOS) */
20   char far *value_reg; /* Representa um ponteiro para o resultado da flag de servicos do DOS */
21 }PTR_CR;
22
23 /* Variavel para identificacao na Regiao Critica do DOS */
24 PTR_CR crDOS;

```

A estrutura **pcb** é a estrutura que representa o BCP (Bloco de Controle de Processos) e contém os campos relativos ao nome do processo, ao estado corrente do processo, ao contexto do processo, ponteiros para a fila de semáforos e ponteiros para a fila de processos. Essa estrutura foi utilizada para representar os processos existentes no sistema. Além disso, criou-se uma definição de ponteiro dessa estrutura, de maneira a ser a cabeça da fila de processos do sistema (*head\_process\_queue*), a qual será vista logo adiante. Novamente, a imagem abaixo representa essa estrutura.

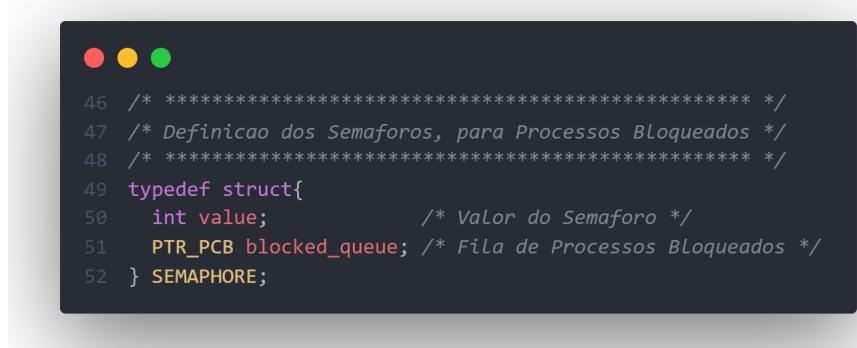
```

26 /* ****|^
27 /* Definicao do BCP = Bloco de Controle de Processo (Descriptor) */
28 /* ****|^
29 typedef struct pcb
30 {
31   char name[MAX_NAME_PROCESS]; /* Armazenar o identificador do processo */
32   enum
33   {
34     activated,
35     blocked,
36     finished
37   } state;                  /* Armazenar o estado corrente do processo */
38   PTR_DESC context;        /* Armazenar o contexto do processo */
39   struct pcb *semaphoreQueue; /* Fila de Processos Bloqueados por um Semaforo */
40   struct pcb *next_pcbs;    /* Fila de Processos em forma de Lista Circular */
41 } PCB;
42
43 /* Ponteiro para o Bloco de Controle de Processo (Alocacao Dinamica) */
44 typedef PCB *PTR_PCB;

```

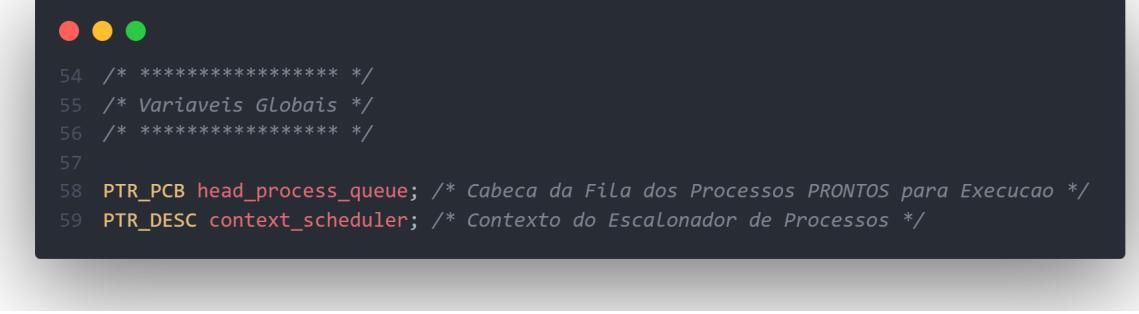
A estrutura **semaphore** está relacionada ao uso de variáveis semáforo para controlar as variáveis compartilhadas entre os processos. Essa estrutura contém os campos relativos ao

valor do semáforo e a fila de processos bloqueados pelo semáforo. Igualmente acima, a imagem abaixo representa essa estrutura.



```
46 /* **** */
47 /* Definicao dos Semaforos, para Processos Bloqueados */
48 /* **** */
49 typedef struct{
50     int value;           /* Valor do Semaforo */
51     PTR_PCB blocked_queue; /* Fila de Processos Bloqueados */
52 } SEMAPHORE;
```

Além das estruturas, necessitou-se da criação de certas variáveis globais. São elas: **head\_process\_queue**, associada à cabeça da fila dos processos presentes no sistema; e o **context\_scheduler**, o qual representa o contexto do processo do escalonador de processos. A imagem abaixo demonstra a criação dessas variáveis extremamente importantes.



```
54 /* **** */
55 /* Variaveis Globais */
56 /* **** */
57
58 PTR_PCB head_process_queue; /* Cabeça da Fila dos Processos PRONTOS para Execucao */
59 PTR_DESC context_scheduler; /* Contexto do Escalonador de Processos */
```

### 3.1.2 Co-Rotinas e Algoritmos

O núcleo possui apenas uma co-rotina, o **scheduler** (escalonador de processos), que é um processo responsável por trocar os processos em execução. Quando um processo tem sua fatia de tempo esgotada e desde que a interrupção do *timer* não tenha ocorrido enquanto um serviço do DOS estava sendo executado, o escalonador retira o processo de execução e coloca o próximo processo da fila de processos para executar. A imagem abaixo ilustra a totalidade dessa função, a qual é separada nas seguintes partes: atribuição de valores iniciais do escalonador (por meio de *p\_est*); configuração e recuperação dos dados advindos dos registradores de *flag* para verificação da região crítica do DOS; e troca de contexto de execução de processos, por meio do *iotransfer*.

```

● ● ●
135 /* Escalonador de Processos */
136 void far scheduler(){
137     /* Atribui valores iniciais nas variáveis globais do Escalonador (p_est) para o iotransfer() */
138     p_est->p_origem = context_scheduler;           /* Origem = Escalonador */
139     p_est->p_destino = head_process_queue->context; /* Destino = head_process_queue->context (Inicio da Fila de Processos) */
140     p_est->num_vetor = INTERRUPT_BIT;             /* Numero do Bit de Interrupcao = Timer (8) */
141
142     /* Iniciando variáveis para identificação na Região Crítica do DOS */
143
144     /* Carrega o registrador AX com 0x34 */
145     _AH=0x34; /* Iniciando o Registrador AH com o End. 0x34 (associado ao End. de um flag de pilha 'servicos' do DOS) */
146     _AL=0x00; /* Iniciando o Registrador AL */
147
148     geninterrupt(0x21); /* Gera uma interrupção associada ao End. 0x21 (aciona serviço inicial do DOS) */
149
150     crDOS.reg.es1 = _ES; /* Salvando registrador ES (associado ao End. do segmento "mais significativo" de 16 bits do flag) */
151     crDOS.reg.bx1 = _BX; /* Salvando registrador BX (associado ao End. do deslocamento "menos significativo" de 16 bits do flag) */
152
153     /* Realizando o controle e mudança de processos */
154     while(1){
155         iotransfer(); /* Realizando controle por interrupção de tempo (dar uma fatia ao processo) */
156         disable(); /* Desabilitando as interrupções para mudança de processo */
157
158         /*
159          Verificando se o processo NÃO está na Região Crítica, para troca.
160          Para tanto, a partir dos registradores ES-BX, o valor do flag de serviços é apontado por "value_reg".
161        */
162
163         if(!(*crDOS.value_reg)){
164             /* Retornando o próximo processo ativo, caso exista */
165             if ((head_process_queue = returnNextActivated()) == NULL){
166                 /* Se não existir, retorna ao DOS */
167                 returnDOS();
168             }
169
170             /* Se existir, o processo é colocado em execução pelo escalonador */
171             p_est->p_destino = head_process_queue->context;
172         }
173
174         enable(); /* Habilitando as interrupções para dar uma fatia de tempo ao novo processo */
175     }
176 }
```

Além disso, a co-rotina do escalonador é auxiliada por outra função chamada “*activateScheduler*” (ativador do escalonador) que inicia o processo do escalonador e transfere o controle para ele, conforme demonstra a imagem abaixo.

```
178 /* Ativar o Escalonador (Processo Principal) */
179 void far activateScheduler(){
180
181     /* Criacao dos Descritores */
182     PTR_DESC aux_activate = cria_desc();
183     context_scheduler = cria_desc();
184
185     /* Inicia o processo do escalador no seu descritor associado */
186     newprocess(scheduler, context_scheduler);
187
188     /* Transfere o controle atual para o escalador */
189     transfer(aux_activate, context_scheduler);
190 }
```

Existem várias outras funções no Kernel, como as funções relacionadas a criação e finalização de processos (*createProcess* e *terminateProcess*). A primeira é responsável por alocar uma estrutura *PCB* na memória (criar o BCP do processo), criar o processo e colocá-lo na fila de processos. A segunda finaliza o processo alterando seu estado para finalizado. Além disso, existe a função “*initiateProcessQueue*” que é responsável por inicializar a fila de processos. As imagens a seguir representam as funções citadas.

```

78 /* Criacao de um Processo (Alocar BCP e inserir no fim da Fila) */
79 void far createProcess(void far (*p_address)(), char p_name[MAX_NAME_PROCESS]){
80     /* Alocacao do BCP = espaco em memoria (malloc) e informacoes */
81     PTR_PCB aloc_process = (PTR_PCB)malloc(sizeof(struct pcb));
82     strcpy(aloc_process->name, p_name);
83     aloc_process->state = activated;
84     aloc_process->context = cria_desc();
85     newprocess(p_address, aloc_process->context);
86     aloc_process->semaphoreQueue = NULL;
87     aloc_process->next_pcb = NULL;
88
89     /* Insercao no fim da Fila de Processos Ativos (prontos para executar) */
90
91     /* Caso a Fila esteja vazia, insere elemento unico */
92     if(head_process_queue == NULL){
93         aloc_process->next_pcb = aloc_process; /* Apontar para o processo */
94         head_process_queue = aloc_process; /* Atribuir o processo na Fila */
95     }
96     /* Do contrario, percorre ate o fim da Fila e insere elemento */
97     else{
98         PTR_PCB aux = head_process_queue; /* Atribuir Fila atual para percorrer */
99         while(aux->next_pcb != head_process_queue){
100             /* Percorre a Fila de Processos Ativos */
101             aux = aux->next_pcb;
102         }
103         /* Insercao do processo no fim da Fila */
104         aux->next_pcb = aloc_process; /* Apontar para o processo */
105         aloc_process->next_pcb = head_process_queue; /* Apontar para o inicio da Fila */
106     }
107 }

```

```

109 /* Encerrar um Processo*/
110 void far terminateProcess(){
111     disable();                                /* Desabilitar as interrupcoes */
112     head_process_queue->state = finished; /* Colocar o processo como terminado */
113     enable();                                 /* Ativar as interrupcoes */
114     while(1);                                /* Gastar o resto de fatia de tempo do processo */
115 }

```

```

73 /* Funcao auxiliar para iniciar a Fila dos Processos PRONTOS para Execucao como vazia */
74 void far initiateProcessQueue(){
75     head_process_queue = NULL;
76 }

```

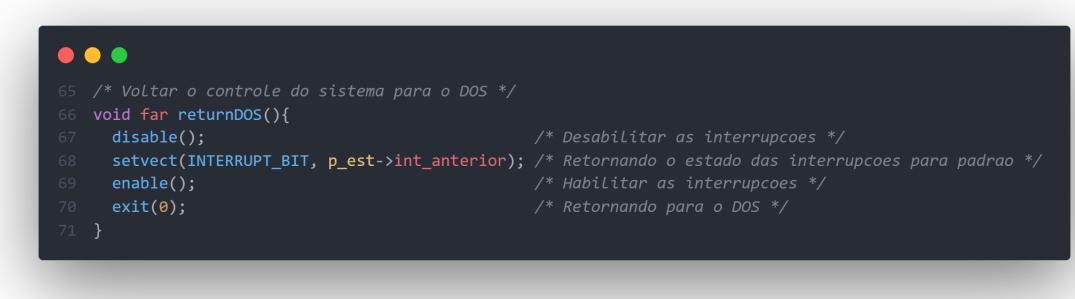
Ainda, existem as funções relacionadas aos semáforos. Existe a função de inicializar o semáforo (*initiateSemaphore*), cujo usuário usará no seu programa e deverá enviar como parâmetros o endereço da variável semáforo e o seu respectivo valor inicial; a função que corresponde à primitiva *P* (ou *Down*) que é responsável por verificar se a região crítica está sendo utilizada e caso esteja, o processo que a utilizou é bloqueado e colocado numa fila de processos bloqueados do semáforo. Caso nenhum outro processo esteja na região crítica, o processo que solicitou a entrada pode acessá-la e o semáforo é decrementado; a função que corresponde à primitiva *V* (ou *Up*) que é responsável por indicar que um processo saiu da região crítica e verificar se há algum processo que foi bloqueado a espera do acesso a ela. Caso haja, o primeiro processo da fila é desbloqueado e pode acessar a região crítica. A imagem em sequência representa as três funções citadas.

```

192 /* ****!*\
193 /* Funcoes para Utilizacao dos Semafornos */
194 /* ****|^
195
196 /* Inicializar o Semaforo */
197 void far initatesSemaphore(SEMAPHORE *user_semaphore, int size_semaphore){
198     user_semaphore->value = size_semaphore; /* Iniciando o valor com o tamanho dado pelo usuario */
199     user_semaphore->blocked_queue = NULL; /* Iniciando a fila de Bloqueados com nulo */
200 }
201
202 /* Primitiva P (Down): decrementar o valor do semaforo. Se for zero, o processo eh colocado na Fila de Bloqueados */
203 void far downSemaphore(SEMAPHORE *user_semaphore){
204     disable(); /* Desabilitando as interrupcoes para manipulacao de variavel compartilhada (semaforos) */
205
206     /* Condicoes da Primitivo P */
207     /* Verifica se a regiao critica esta sendo utilizada */
208     if(user_semaphore->value > 0){
209         /* Decrementa o semaforo e NAO bloqueia o processo */
210         user_semaphore->value--;
211     }
212     else{
213         /* Auxiliar de Ponteiro de BCP */
214         PTR_PCB p_aux;
215
216         /* Mudando o estado do processo ativo (head_process_queue) para bloqueado */
217         head_process_queue->state = blocked;
218
219         /* Bloquear o processo corrente e adiciona-lo a fila do semaforo */
220         /* Verifica se a fila de bloqueados do semaforo esta vazia */
221         if(user_semaphore->blocked_queue == NULL){
222             /* Se estiver, entao insere o processo */
223             user_semaphore->blocked_queue = head_process_queue; /* Insere o primeiro processo na fila */
224         }
225         else{
226             /* Cria BCP auxiliar e utiliza ele para percorrer a fila de processos bloqueados */
227             PTR_PCB aux;
228             aux = user_semaphore->blocked_queue;
229
230             /* Percorre a fila de processos bloqueados ate o ultimo existente */
231             while(aux->semaphoreQueue != NULL){
232                 aux = aux->semaphoreQueue;
233
234                 /* Salva o processo atual no fim da fila de processos bloqueados */
235                 aux->semaphoreQueue = head_process_queue;
236             }
237
238             /* Atribui o semaforo do processo atual com nulo */
239             head_process_queue->semaphoreQueue = NULL;
240         }
241
242         /* Salvar as informacoes da BCP atual (bloqueado) em uma auxiliar */
243         p_aux = head_process_queue;
244
245         /* Procurar o proximo processo ativo, se existir */
246         if(head_process_queue = returnNextActivated() == NULL){
247             /* Retornar o controle para o DOS, ja que o processo atual se bloqueou e nao ha ativos... */
248             /* Situacao de DEADLOCK!! */
249             returnDOS();
250         }
251
252         /* Do contrario, transferir o controle da auxiliar->contexto para o novo processo ativo */
253         transfer(p_aux->context, head_process_queue->context);
254     }
255
256     enable(); /* Habilita as interrupcoes novamente para o Escalonador */
257 }
258
259 /* Primitiva V (Up): colocar o processo como ativo se o semaforo estiver nulo e tiver algum processo na Fila. Do contrario, decrementa o valor do semaforo */
260 void far upSemaphore(SEMAPHORE *user_semaphore){
261
262     disable(); /* Desabilitando as interrupcoes para manipulacao de variavel compartilhada (semaforos) */
263
264     /* Condicoes da Primitivo V */
265
266     /* Verifica se a fila de bloqueados do semaforo esta nula */
267     if(user_semaphore->blocked_queue == NULL){
268         /* Apenas incrementa na variavel do semaforo, para indicacao de uso de uma regiao critica */
269         user_semaphore->value++;
270     }
271     else{
272         /* Auxiliar de Ponteiro de BCP */
273         PTR_PCB p_prox;
274
275         /* Recupera o primeiro elemento da Fila de Bloqueados */
276         p_prox = user_semaphore->blocked_queue;
277
278         /* Avanca a cabeca da fila para o proximo processo bloqueado */
279         user_semaphore->blocked_queue = p_prox->semaphoreQueue;
280
281         /* Remove o processo, novamente ativo, da fila de bloqueados */
282         p_prox->semaphoreQueue = NULL;
283
284         /* Mudar o estado do processo bloqueado para ativo novamente */
285         p_prox->state = activated;
286     }
287
288     enable(); /* Habilita as interrupcoes novamente para o Escalonador */
289 }

```

Por fim, o núcleo possui uma função chamada *returnDOS* que é chamada quando não houverem mais processos ativos para que o escalador coloque para executar. Isso pode ocorrer caso todos os processos terminem ou estejam bloqueados simultaneamente (situação de *deadlock*). Dessa forma, a função *returnDOS* restabelece a rotina padrão para interrupção do *timer* e finaliza a execução do programa, conforme é mostrado abaixo.



```
65 /* Voltar o controle do sistema para o DOS */
66 void far returnDOS(){
67     disable();                                /* Desabilitar as interrupções */
68     setvect(INTERRUPT_BIT, p_est->int_anterior); /* Retornando o estado das interrupções para padrão */
69     enable();                                 /* Habilitar as interrupções */
70     exit(0);                                  /* Retornando para o DOS */
71 }
```

### 3.1.3 Kernel.H

Por fim, para facilitar a utilização do núcleo criado pelos usuários, criou-se um arquivo de cabeçalho de tal forma que somente funções específicas para o usuário fossem visíveis. Em outras palavras, funções como a do escalonador não estão disponíveis para o usuário manipular diretamente, ao passo que as funções de criação e encerramento de processos estão presentes. Dessa forma, é possível importar o núcleo como uma biblioteca comum da linguagem C, facilitando o seu uso. Por fim, o arquivo de cabeçalho foi separado em três partes, sendo elas (com imagens associadas):

- Estruturas de Dados

```

● ● ●

1 #include <system.h>
2
3 #define MAX_NAME_PROCESS 35
4 #define INTERRUPT_BIT 8
5
6 /* **** */
7 /* Definicao do BCP = Bloco de Controle de Processo (Descriptor) */
8 /* **** */
9 typedef struct pcb
10 {
11     char name[MAX_NAME_PROCESS]; /* Armazenar o identificador do processo */
12     enum
13     {
14         activated,
15         blocked,
16         finished
17     } state; /* Armazenar o estado corrente do processo */
18     PTR_DESC context; /* Armazenar o contexto do processo */
19     struct pcb *semaphoreQueue; /* Fila de Processos Bloqueados por um Semaforo */
20     struct pcb *next_pcb; /* Fila de Processos em forma de Lista Circular */
21 } PCB;
22
23 /* Ponteiro para o Bloco de Controle de Processo (Alocacao Dinamica) */
24 typedef PCB *PTR_PCB;
25
26 /* **** */
27 /* Definicao dos Semaforos, para Processos Bloqueados */
28 /* **** */
29 typedef struct{
30     int value; /* Valor do Semaforo */
31     PTR_PCB blocked_queue; /* Fila de Processos Bloqueados */
32 } SEMAPHORE;

```

- Gerenciamento de Processos

```

● ● ●

35 /* **** */
36 /* Funcoes Basicas do Nucleo Multiprogramado VISIVEIS AO USUARIO */
37 /* **** */
38
39 /* Funcao auxiliar para iniciar a Fila dos Processos PRONTOS para Execucao como vazia */
40 void far initiateProcessQueue();
41
42 /* Criacao de um Processo (Alocar BCP e inserir no fim da Fila) */
43 void far createProcess(void far (*p_address)(), char p_name[MAX_NAME_PROCESS]);
44
45 /* Encerrar um Processo*/
46 void far terminateProcess();
47
48 /* Ativar o Escalador (Processo Principal) */
49 void far activateScheduler();

```

- Gerenciamento de Semáforos



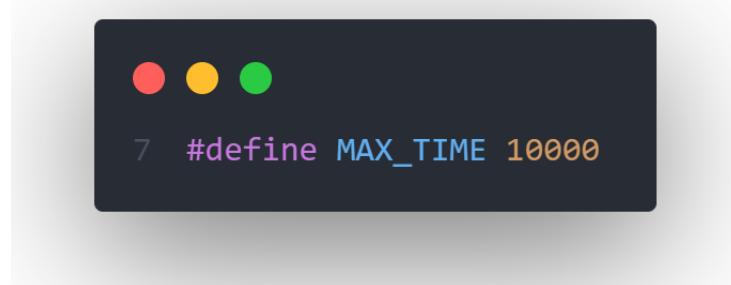
```

52 /* **** */
53 /* Funções para Utilização dos Semáforos */
54 /* **** */
55
56 /* Inicializar o Semaforo */
57 void far initiateSemaphore(SEMAPHORE *user_semaphore, int size_semaphore);
58
59 /* Primitiva P (Down): decrementar o valor do semáforo. Se for zero, o processo é colocado na Fila de Bloqueados */
60 void far downSemaphore(SEMAPHORE *user_semaphore);
61
62 /* Primitiva V (Up): colocar o processo como ativo se o semáforo estiver nulo e tiver algum processo na Fila. Do contrário, decrementa o valor do semáforo */
63 void far upSemaphore(SEMAPHORE *user_semaphore);

```

## 3.2 TK-01.C

O programa TK-01 é um programa que foi feito e usado para testar o funcionamento do escalonador de processos. O programa consiste na existência de cinco processos que executam um laço de repetição que imprime o número do respectivo processo em execução na tela. O escalonador de processos então é ativado e deve escalar esses processos para executar. Para tanto, necessitou-se da seguinte definição no início do código, a fim de limitar o tempo de execução dos processos:



### 3.2.1 Estrutura de Dados

Nesse programa, as estruturas de dados foram utilizadas implicitamente pelo *Kernel*, sendo elas o *PTR\_PCB* e o *crDOS*. A primeira corresponde à estrutura do BCP e a segunda corresponde aos *flags* de serviços do DOS.

### 3.2.2 Co-Rotinas e Algoritmos

As co-rotinas presentes no programa são o *process1*, *process2*, *process3*, *process4*, *process5* que são os processos que executarão nesse exemplo. Cada qual, presente em uma imagem abaixo, possui a simples função de imprimir na tela um dígito associativo. Além disso, a co-rotina do escalonador é utilizada implicitamente dentro do Kernel.

```
9 void far process1(){
10    int i = 0;
11    while(i < MAX_TIME){
12        printf("1");
13        i++;
14    }
15
16    /* Terminando o Processo...*/
17    terminateProcess();
18 }
```

```
20 void far process2(){
21    int i = 0;
22    while(i < MAX_TIME){
23        printf("2");
24        i++;
25    }
26
27    /* Terminando o Processo...*/
28    terminateProcess();
29 }
```

```
31 void far process3(){
32    int i = 0;
33    while(i < MAX_TIME){
34        printf("3");
35        i++;
36    }
37
38    /* Terminando o Processo...*/
39    terminateProcess();
40 }
```

```
42 void far process4(){
43     int i = 0;
44     while(i < MAX_TIME){
45         printf("4");
46         i++;
47     }
48
49     /* Terminando o Processo...*/
50     terminateProcess();
51 }
```

```
53 void far process5(){
54     int i = 0;
55     while(i < MAX_TIME){
56         printf("5");
57         i++;
58     }
59
60     /* Terminando o Processo...*/
61     terminateProcess();
62 }
```

Por fim, a *main* presente tem a função tanto da criação dos processos (função *createProcess*) quanto da fila de processos e na ativação do escalonador, como pode ser visto na imagem abaixo.

```

64 main(){
65     /* Iniciando a Fila de Processos como Vazia */
66     initiateProcessQueue();
67
68     /* Criacao dos Processos */
69     createProcess(process1, "P1");
70     createProcess(process2, "P2");
71     createProcess(process3, "P3");
72     createProcess(process4, "P4");
73     createProcess(process5, "P5");
74
75     /* Transferindo o controle para o Escalonador */
76     activateScheduler();
77 }
```

### 3.3 TK-02.C

O programa TK-02 é um programa que foi feito e usado para testar o funcionamento do mecanismo dos semáforos e do escalonador de processos. O programa é a aplicação do problema do produtor e do consumidor e é composta de dois processos que representam o produtor e consumidor, respectivamente. Para tanto, limitou-se em 1000 iterações com um buffer de valores inteiros com tamanho máximo de 80, conforme mostra a imagem abaixo das definições.

```

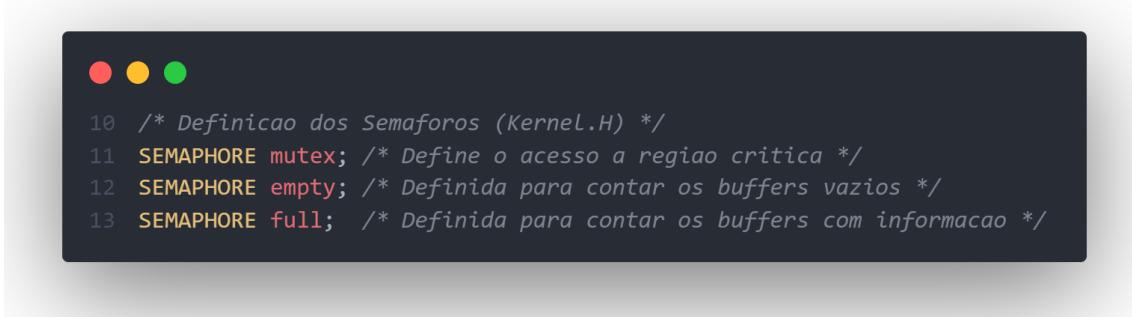
7 #define MAX_ITERATIONS 1000
8 #define MAX_ITEMS 80
```

O problema do produtor e do consumidor consiste na existência de um buffer circular que é compartilhado entre processos, utilizado para a troca de dados. Nesse buffer circular, haverão processos que colocarão dados nos slots vazios do buffer, chamados de produtores, e processos que retirarão esses dados, chamados de consumidores.

O problema surge quando o produtor quer colocar um dado novo nos slots do buffer, mas esse já se encontra cheio. A solução seria o processo produtor se bloquear e ser desbloqueado quando o processo consumidor retirar um ou mais itens do buffer. De forma análoga, o consumidor deve se bloquear quando não houver mais slots contendo dados no buffer e ser acordado quando o produtor tiver colocado um ou mais dados no buffer.

O problema ainda possui algumas restrições. A primeira delas é que o produtor não deve exceder a capacidade finita do buffer, a segunda é que o consumidor não pode retirar os dados mais rápido do que elas são inseridas no buffer pelo produtor, a terceira é que os dados devem ser retiradas do buffer na mesma ordem em que elas foram colocadas e, por fim, deve haver exclusão mútua no acesso ao buffer.

Dessa forma, necessitou-se definir três semáforos: o *mutex*, o qual define o acesso a região crítica pelos processos; o *empty*, o qual é definido para contar os espaços vazios do buffer; e o *full*, cuja função é contar os espaços cheios (com informação) do buffer. A imagem abaixo mostra a definição dos três semáforos no programa.

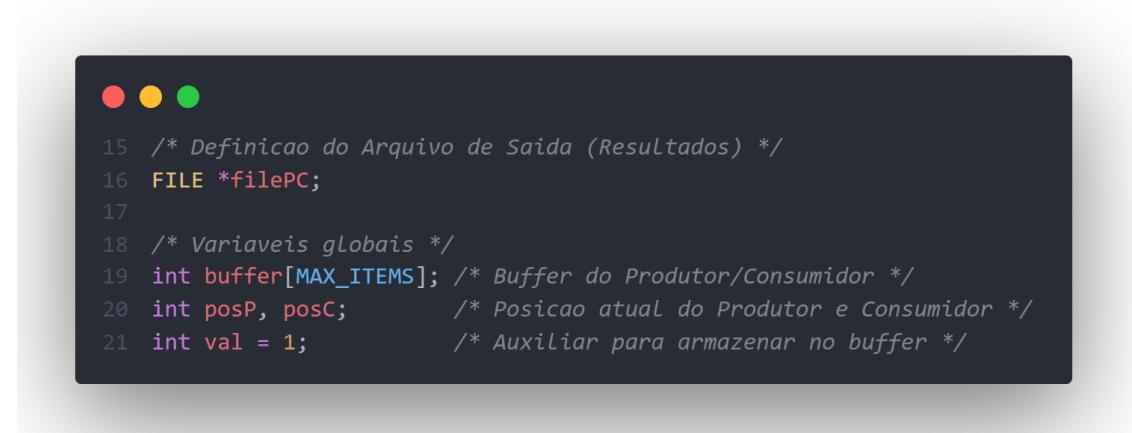


```

10 /* Definicao dos Semaforos (Kernel.H) */
11 SEMAPHORE mutex; /* Define o acesso a regiao critica */
12 SEMAPHORE empty; /* Definida para contar os buffers vazios */
13 SEMAPHORE full; /* Definida para contar os buffers com informacao */

```

Por fim, para facilitar a obtenção dos resultados, definiu-se um arquivo de saída conforme mostra a imagem abaixo. Também são mostradas as variáveis globais presentes nesse programa teste.



```

15 /* Definicao do Arquivo de Saída (Resultados) */
16 FILE *filePC;
17
18 /* Variaveis globais */
19 int buffer[MAX_ITEMS]; /* Buffer do Produtor/Consumidor */
20 int posP, posC; /* Posicao atual do Produtor e Consumidor */
21 int val = 1; /* Auxiliar para armazenar no buffer */

```

### 3.3.1 Estrutura de Dados

As estruturas utilizadas nesse programa foram o **semaphore** e o **pcb**, cuja primeira representa as variáveis semáforo e a segunda representa o BCP dos processos.

### 3.3.2 Co-Rotinas e Algoritmos

As co-rotinas utilizadas foram o **producer** e o **consumer** que são os processos produtor e consumidor, respectivamente.

O processo produtor produz um dado e testa a variável semáforo *empty* utilizando a primitiva *P* (ou *Down*) para saber se existem slots vazios no buffer, em caso positivo o processo em seguida verifica o semáforo *mutex*, para saber se a região crítica do buffer está vazia. Caso o buffer esteja cheio, o processo produtor se bloqueia. Ao testar o semáforo *mutex*, caso seja verificado que não há nenhum processo acessando o buffer, ele insere o item no buffer e invoca a primitiva *V*.

Através da primitiva *V* (ou *Up*), o processo produtor sai da região crítica e checa se há algum processo bloqueado na fila de bloqueados do semáforo à espera de acessá-la, caso haja, o primeiro processo da fila é desbloqueado e então acessa a região crítica. Após, por meio da mesma primitiva, o produtor checa se há algum processo consumidor esperando para ser desbloqueado para consumir os dados do buffer por meio do semáforo *full*. Se houver, o consumidor é desbloqueado.

A implementação do produtor pode ser vista na imagem abaixo, bem como a invocação das primitivas *P* e *V*.

```

● ● ●
78 /* Co-rotina do Produtor, com utilizacao dos semaforos e Limitacao em iteracoes */
79 void far producer(){
80     /* Variaveis auxiliares: valor do item do Produtor e Limitacao de iteracoes */
81     int itemP, i;
82
83     /* Inicializacao de variaveis Locais */
84     posP = 0; /* Posicao atual do Produtor no buffer */
85     i = 0;
86
87     while (i < MAX_ITERATIONS){
88         itemP = produceItem(); /* Realiza a producao do item */
89
90         downSemaphore(&empty); /* Verifica se o buffer possui slots vazios */
91         downSemaphore(&mutex); /* Se sim o produtor entra na regiao critica para inserir o item no slot */
92
93         insertItem(itemP); /* Insere o item no slot vazio do buffer */
94
95         /* Insere no arquivo, de maneira formatada */
96         fprintf(filePC, "\nProdutor depositou no buffer[%d] = %d", posP, itemP);
97
98         /* Mostrando o resultado na tela */
99         /*printf("\n%P-buffer[%d] = %d", posP, itemP);*/
100
101        upSemaphore(&mutex); /* Sair da regiao critica */
102        upSemaphore(&full); /* Verifica se existem slots com item. Se existirem desbloqueia o processo Consumidor caso ele esteja bloqueado.*/
103
104        i++;
105    }
106
107    /* Terminando o Processo...*/
108    terminateProcess();
109 }

```

De forma análoga ocorre com o processo consumidor. O processo consumidor checa o semáforo *full* para saber se existem dados no buffer para serem consumidos, em caso positivo, ele segue para o acesso à região crítica que representa o buffer circular testando o semáforo *mutex*. Quando ele acessa o buffer, ele consome o dado, sai da região crítica e acorda qualquer processo que esteja esperando para acessar o buffer ou que estava bloqueado esperando haver slots vazios para produzir novos dados. A implementação do consumidor pode ser vista na imagem abaixo, bem como a invocação das primitivas *P* e *V*.

```

  ● ○ ●
111 /* Co-rotina do Consumidor, com utilizacao dos semaforos e Limitacao em iteracoes */
112 void far consumer(){
113     /* Variaveis auxiliares: valor do item do Consumidor e limitacao de iteracoes */
114     int itemC, i;
115
116     /* Inicializacao de variaveis locais */
117     posC = 0; /* Posicao atual do Consumidor no buffer */
118     i = 0;
119
120     while (i < MAX_ITERATIONS){
121         downSemaphore(&full); /* Verifica se o buffer possui slots com conteudo */
122         downSemaphore(&mutex); /* Se sim o consumidor entra na regiao critica para consumir o item do slot */
123
124         itemC = consumeItem();
125
126         /* Insere no arquivo, de maneira formatada */
127         fprintf(filePC, "\nConsumidor retirou do buffer[%d] = %d", posC, itemC);
128
129         /* Mostrando o resultado na tela */
130         /*printf("\nC-buffer[%d] = %d", posC, itemC);*/
131
132         upSemaphore(&mutex); /* Sair da regiao critica */
133         upSemaphore(&empty); /* Verifica se existem slots vazios. Se existirem desbloqueia o processo Produtor caso ele esteja bloqueado. */
134
135         i++;
136     }
137
138     /* Terminando o Processo...*/
139     terminateProcess();
140 }
```

Para que tudo ocorra como o esperado, foram criadas ainda as funções *produceItem*, *insertItem* e *consumeItem*. As duas primeiras são executadas pelo processo produtor e tem como função produzir os dados e inserir os dados no buffer, respectivamente. Por fim, a função *consumeItem* é executada pelo consumidor e é responsável por retirar o dado do buffer. As três rotinas podem ser vistas nas imagens a seguir, respectivamente.

```

  ● ○ ●
33 /* Funcao auxiliar do Produtor: produzir um item (aumentar o valor) */
34 int far produceItem(){
35     val += 2;
36     return val;
37 }
```

```

39 /* Funcao auxiliar do Produtor: inserir item produzido no buffer */
40 void far insertItem(int item1){
41     /* Somente insere em uma posicao vazia do buffer */
42     while (buffer[posP] != -1){
43         posP++; /* Avanca a posicao */
44
45         /* Se chegar no indice chegar ao final do buffer, ele volta para o slot inicial do buffer, pois se trata de uma lista circular */
46         if (posP == MAX_ITEMS)
47             posP = 0; /* Posicao do slot inicial do buffer */
48     }
49     buffer[posP++] = item1; /* Insere o item no buffer e avanca uma posicao */
50
51     /* Se chegar no indice chegar ao final do buffer, ele volta para o slot inicial do buffer, pois se trata de uma lista circular */
52     if (posP == MAX_ITEMS)
53         posP = 0; /* Posicao do slot inicial do buffer */
54 }
```

```

56 /* Funcao auxiliar do Consumidor: retirar item do buffer */
57 int far consumeItem(){
58     int item2; /* Auxiliar para resgatar item do buffer */
59
60     while (buffer[posC] == -1){
61         posC++; /* Avanca a posicao */
62
63         /* Se chegar no indice do final do buffer, ele volta para o slot inicial do buffer, pois se trata de uma lista circular */
64         if (posC == MAX_ITEMS)
65             posC = 0; /* Posicao do slot inicial do buffer */
66     }
67     item2 = buffer[posC]; /* Retira item do buffer */
68
69     buffer[posC++] = -1; /* Remove item do buffer e avanca uma posicao */
70
71     /* Se chegar no indice do final do buffer, ele volta para o slot inicial do buffer, pois se trata de uma lista circular */
72     if (posC == MAX_ITEMS)
73         posC = 0; /* Posicao do slot inicial do buffer */
74
75     return item2; /* Retorna o item consumido para mostrar na tela/arquivo */
76 }
```

Ainda, houve a criação de uma função específica para a inicialização do arquivo. Esse arquivo, como já dito, foi teorizado para facilitar a obtenção dos resultados providos pelos processos *produtor* e *consumidor*. Em outras palavras, tanto os itens produzidos quanto os itens consumidos são indicados neste arquivo que é do tipo texto. A imagem abaixo representa essa função de inicialização do arquivo texto.

```

23 /* Funcao auxiliar para criacao e inicializacao do arquivo de saida */
24 void far iniitateFile(){
25     /* Abrindo apenas com Leitura */
26     if (!(filePC = fopen("TK-02.txt", "w"))){
27         printf("Erro ao abrir/criar o arquivo.\n");
28         system("Pause");
29         exit(0);
30     }
31 }
```

Por fim, a função *main* consiste tanto de inicializar os semáforos, buffer e arquivo texto de saída quanto criar os processos associados (*produtor* e *consumidor*), conforme é demonstrado na imagem abaixo.



```
142 main(){
143     /* Iniciando a Fila de Processos como Vazia */
144     initiateProcessQueue();
145
146     /* Inicializacao dos Semaforos */
147     initiateSemaphore(&mutex, 1);
148     initiateSemaphore(&empty, MAX_ITEMS);
149     initiateSemaphore(&full, 0);
150
151     /* Criacao dos Processos */
152     createProcess(producer, "Produtor");
153     createProcess(consumer, "Consumidor");
154
155     /* Inicializacao do buffer com valor -1 (vazio) */
156     memset(buffer, -1, sizeof(buffer));
157
158     /* Realiza a abertura/criacao do arquivo texto para resultados finais */
159     initiateFile();
160
161     /* Transferindo o controle para o Escalonador */
162     activateScheduler();
163 }
```