

— PROJETO SISTEMAS OPERACIONAIS —

- *Tema:* núcleo multiprogramado
 - *Alunos:*
 - Dante Martini Chiarelli Ramacciotti | RA 221023259
 - Manuele Sabatini Christófaló | RA 221026291
 - Paulo Henrique Dionysio | RA 221026169
-

SUMÁRIO

EXERCÍCIOS	2
TicTac	2
TicTac Modificado	3
Escalonador de Corotinas	4
 NÚCLEO	 7
Detalhamento do algoritmo.....	7
Estrutura de Dados.....	7
Casos de Teste.....	10

– EXERCÍCIOS –

1. TicTac

a. Detalhamento do algoritmo

O algoritmo TicTac foi feito como exemplo para aprendermos algumas novidades usando o TurboC. No início do algoritmo, demos um include para chamar as funções referentes ao “system.h”, essa biblioteca disponibiliza algumas funções que usaremos no futuro. Também foi dado include no “stdio.h”, mas essa é a biblioteca referente à própria linguagem C.

A seguir foi feita a declaração dos descritores, “PTR_DESC” é um tipo definido que é usado para criar esses descritores. Nesse caso, está sendo usado como um ponteiro de contexto para um tipo descritor.

Função TIC: Colocada em um laço de repetição infinito “while(1)”. Dentro desse laço de repetição rodará um “printf(“TIC-”);” para realizar o print da palavra TIC. Logo em seguida, a função “transfer(d_tic, d_tac)”. Essa função corresponde à transferência de uma função para outra, passando por parâmetro seus respectivos descritores, a fim de transferir o controle da co-rotina origem para a co-rotina destino. Vale lembrar que a função “transfer(origem, destino)” envia como primeiro parâmetro o descritor de origem, ou seja, aquele que está sendo usado atualmente e guarda o contexto da co-rotina origem. Como segundo parâmetro, o descritor destino, com a finalidade de mapear o estado da co-rotina destino.

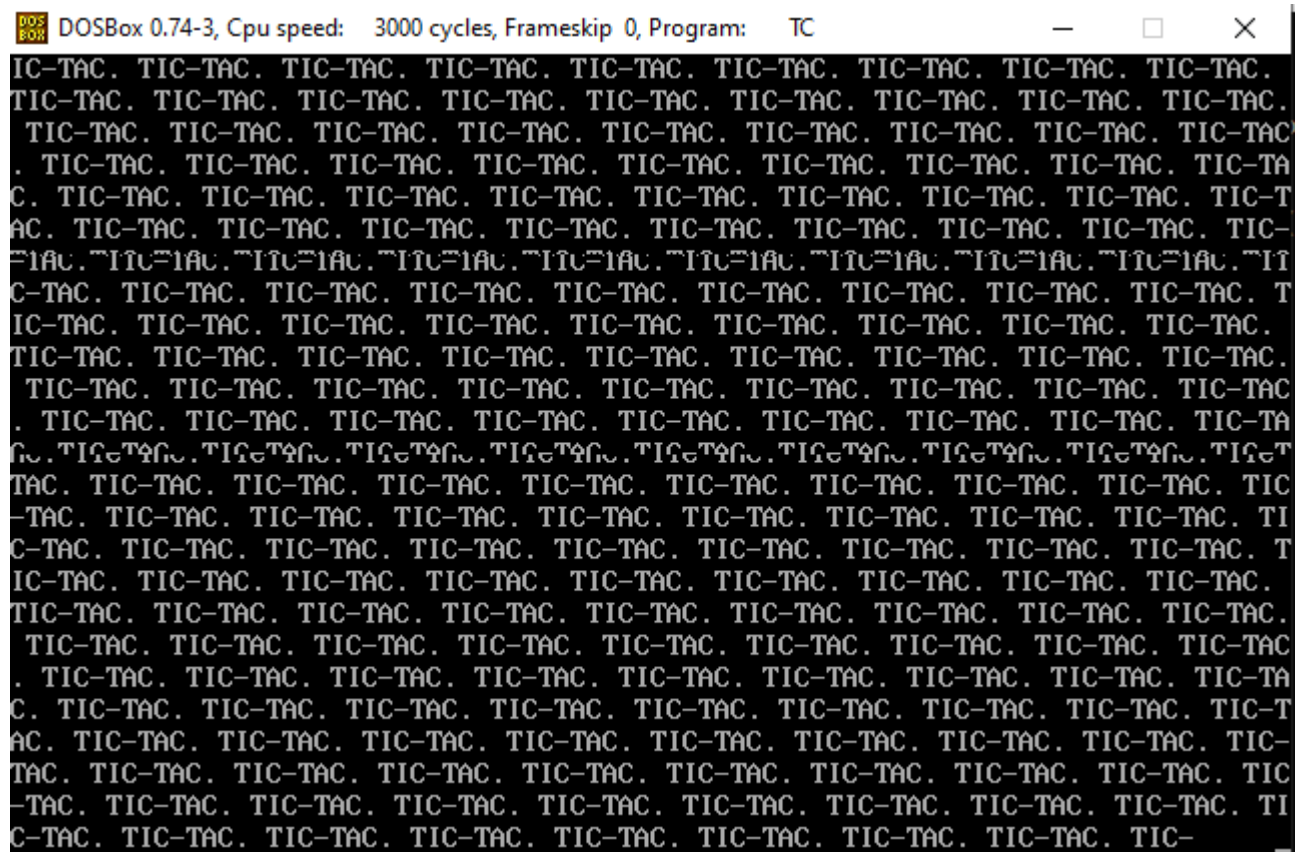
Função TAC: Esta função desempenhará o mesmo que a função TIC. As duas únicas diferenças são o “printf(“TAC. ”);” que printará a palavra TAC e a função “transfer(d_tac, d_tic);” que guardará o contexto da co-rotina origem e desviará o controle para a co-rotina destino, mapeando o seu estado.

Função MAIN: Para finalizar o algoritmo, a função principal está criando 3 descritores, são eles: “d_main = cria_desc(), d_tic = cria_desc(), d_tac = cria_desc()”, todos recebendo a função “cria_desc()”, que cria dinamicamente um descritor de contexto e retorna o endereço desse descritor.

A função “newprocess(tic, d_tic)” tem o objetivo de iniciar a estrutura de contexto (descritor) da função TIC. Seguindo dessa forma, também foi criada a estrutura de contexto da função TAC com “newprocess(tac, d_tac)”.

A finalização do algoritmo se dá com o “transfer(d_main, d_tic)”, salvando o contexto da main e mapeando o contexto do TIC.

b. Caso de teste



2. TicTac Modificado

a. Detalhamento do algoritmo

O algoritmo TicTac Modificado é uma cópia do algoritmo TicTac, com uma diferença: um limite foi imposto no laço de repetição “while(count < 100)”. Uma variável global foi iniciada após a criação dos descritores (PTR_DESC). Essa variável é um inteiro de valor igual a 0. Quando a compilação chega até a função TIC, então ele é incrementado em 1 toda vez que é rodado, até chegar ao valor 99, quando serão repetidos 100 vezes os respectivos prints: “TIC-” e “TAC.”.

b. Caso de teste

```

DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: TC
C:\>TC>TC.EXE
TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC.
TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC
. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TA
C. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-T
AC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-
TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC
-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TI
C-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. T
IC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC.
TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC.
TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC
. TIC-TAC. TIC-TAC. Divide error
TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC.
TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC
. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TA
C. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-T
AC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-
TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC
-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TI
C-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. T
IC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC.
TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC.
TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC. TIC-TAC
. TIC-TAC. TIC-TAC.

```

3. Escalonador de Corotinas

a. Detalhamento do algoritmo

O algoritmo Escalonador é uma reformulação do TicTac e do TicTac Modificado. No início do código é verificado um novo descritor sendo declarado: "desc". Esse novo descritor declarado se refere ao descritor do escalonador, que será explicado ao longo do texto.

Função Tic e Tac: Agora essas funções irão desempenhar apenas dois comandos: o laço de repetição infinito (while(1)) e o print na tela (printf("TIC-"); e printf("TAC. ");). A função transfer() foi retirada para a inserção de uma nova função, o iotransfer().

Função Escalonador: Ao iniciar essa função, é visto um ponteiro para uma struct que armazena informações relacionadas à transferência de contexto ou manipulação de interrupções. O p_est está armazenando os seguintes campos:

- p_origem: Sua funcionalidade é armazenar o descritor de origem (co-rotina que está atualmente em execução).
- p_destino: Sua funcionalidade é armazenar o descritor de destino (co-rotina que deverá ser executada em seguida).

- `num_vetor`: Este campo irá armazenar o número do vetor de interrupção, nesse caso, o valor 8, que será o valor referente ao Timer (interrupção que usaremos ao implementar o núcleo).

Após as declarações, é iniciado um looping infinito com a função `while(1)`, em seguida, a declaração da função `iotransfer()`, cujos parâmetros são os três tópicos citados acima. Essa função desempenha um papel crucial na manipulação das interrupções. Em sua primeira chamada, ela instala a co-rotina chamadora como a rotina de interrupção associada ao vetor de interrupção especificado (`num_vetor`). Em seguida, ocorre uma transferência de controle, na qual o `iotransfer()` transfere o controle para a co-rotina especificada como destino (`p_est->p_destino`, inicialmente `dtic`). Após a ocorrência de transferência, é efetuado o salvamento de estado. Isso acontece quando ocorre uma interrupção; `iotransfer()` salva o estado da co-rotina interrompida (aquela que estava em execução antes da interrupção). Por fim, executa a rotina anterior do DOS (se aplicável) e então retorna o controle à co-rotina chamadora. É importante frisar que a função `iotransfer()` é chamada repetidamente para alternar entre as funções TIC e TAC.

A função `disable()` é usada para desativar interrupções. Quando as interrupções são desativadas, o processador não responderá a interrupções externas até que elas sejam reativadas. Isso garante que o código que segue a chamada `disable()` seja executado de forma atômica, sem ser interrompido por outras tarefas ou interrupções. Essa função garante que a verificação e atualização do descritor de destino (`p_destino`) ocorram sem interrupções, evitando condições de corrida ou outros problemas de sincronização.

Já a função `enable()` é usada para reativar (desmascarar) interrupções que foram desativadas pela função `disable()`. Isso permite que o processador volte a responder a interrupções externas. Essa função permite que as interrupções sejam tratadas novamente após a atualização do descritor de destino.

Função MAIN: Para finalizar a análise do algoritmo, na função `main` é visto `desc = cria_desc();` e `newprocess(escalonador, desc)`, no qual, respectivamente, um está criando um descritor e o outro está iniciando a estrutura de contexto da função `Escalonador()`.

– NÚCLEO –

1. Detalhamento do algoritmo

O código inicia tratando da região crítica do dos. No caso, o escalador deve identificar se o processo interrompido já havia chamado algum serviço do sistema operacional – se sim, o escalador retorna o controle ao DOS para que este termine o serviço, concedendo-o mais uma fatia de tempo. Para isso, ele define os registradores responsáveis por identificar a entrada na região crítica.

Em seguida, ele detalha a criação do descritor de processos (BCP). Cada processo deve ter um BCP associado e que contenha os campos: nome, estado, ponteiro para o descritor de contexto, e ponteiro para o próximo descritor de processo. Para isso, uma struct é ideal.

As funções essenciais são então definidas, sendo que:

- `cria_processo()`: associa um descritor de processo ao código do processo e coloca-o na fila dos prontos;
- `dispara_sistema()`: transfere o controle do programa principal (main) para o escalador;
- `escalador()` : co-rotina do escalador;
- `termina_processo()`: marca o processo chamador como “terminado”;
- `volta_dos()`: desinstala o escalador e retorna o controle para o DOS;
- `procura_prox_ativo()`: retorna o endereço do descritor do próximo processo ativo da fila dos prontos (a partir da posição atual do cabeça da fila —“prim”).

Para o semáforo, é criada uma variável compartilhada “sem”, com duas operações primitivas indivisíveis que atuam sobre ela. No caso, P (Down), que decrementa o valor do semáforo quando este é diferente de 0; se for 0, ele bloqueia o processo. V (Up), incrementa o valor do semáforo se não há processos bloqueados – caso haja, o processo cabeça é acordado.

2. Estrutura de Dados

```
// Registradores da regioao critica -----
typedef struct registros{
    unsigned bx1, es1;    //bx = registrador base; es = registrador de segmento extra
}regis;
typedef union k{
    regis x;              //Registradores bx + es
    char far *y;          //Valor da flag de servicos
}APONTA_REG_CRIT;
APONTA_REG_CRIT a;
```

Estruturas de dados usadas para manejar a região crítica do DOS. A ED registros é usada na union k para auxiliar o escalonador a definir se a execução do processo está na RC ou não

antes de realizar a troca dele por outro. Isso é necessário para prevenir condições de corrida e trazer uma maior segurança e consistência para o SO.

Quando, no programa, se acessa `a.x.*`, o programa está acessando os valores dos registradores base e do segmento extra. Quando se acessa `a.y`, o programa está manipulando a flag que indica se a RC está em uso.

Em suma, no primeiro trecho, ela é usada para definir a região delimitada pela RC, e, no segundo, para checar se o processo está em sua RC.

```
// Descritor de processos (BCP) -----
typedef struct desc_p{
    char nome[35];           //Nome do processo
    enum{ativo, bloqueado, terminado} estado; //Estado do processo
    PTR_DESC contexto;       //Ponteiro para descritor de contexto
    struct desc_p *fila_sem;  //Fila de processos bloqueados por um semaforo
    struct desc_p *prox_desc; //Ponteiro para o proximo descritor
} DESCRITOR_PROC;
typedef DESCRITOR_PROC *PTR_DESC_PROC; //Ponteiro para o descritor
```

O BCP é a estrutura de dados usada para armazenar o contexto de um processo, permitindo que o escalonador cumpra sua função sem perder o estado de execução do processo sendo substituído. Nesta implementação, o BCP armazena o nome do processo, seu estado, seu contexto, a fila de processos bloqueados por semáforo e um ponteiro para o próximo BCP.

Vale ressaltar que o BCP não é declarado diretamente no programa. Ele é acessado por meio de um ponteiro para um `desc_p`, estrutura definida logo abaixo do próprio BCP no `typedef DESCRITOR_PROC *PTR_DESC_PROC;`

```
// Tipo semaforo -----
typedef struct{
    int s;           //Parte inteira
    PTR_DESC_PROC Q; //Fila bloqueados
} semaforo;
```

Essa é a estrutura que define a implementação dos semáforos neste programa. O `int s` representa o contador do semáforo, que controla o número de recursos disponíveis. A diretiva `P` é invocada quando um processo solicita um recurso, e o valor de `s` é decrementado. A diretiva `V` é invocada quando um processo libera um recurso, incrementando o valor de `s`. Enquanto `s` for maior que 0, recursos estão disponíveis para serem utilizados. Caso contrário, os recursos não estão disponíveis e os processos que os solicitarem podem ser bloqueados.

Q é um ponteiro que aponta para a cabeça da fila de processos bloqueados que estão aguardando a liberação do recurso. Quando um processo libera o recurso, o primeiro processo da fila Q é marcado como ativo e removido da fila, podendo então ser escalonado. Essa fila é uma fila encadeada simples.

```
void far cria_processo(void far (*p_address)(), char nome_p[16]){
    /* Definicoes base */
    PTR_DESC_PROC descritor = (PTR_DESC_PROC)malloc(sizeof(struct desc_p)); /* Criacao
dinamica do descritor */
    strcpy(descritor->nome, nome_p); /* Copia nome */
    descritor->estado = ativo; /* Marca o estado como "ativo" */
    descritor->contexto = cria_desc(); /* Cria descritor de contexto */
    newprocess(p_address, descritor->contexto); /* Inicia descritor de contexto
*/
    descritor->fila_sem = NULL; /* Nenhum processo bloqueado */

    /* Insercao na fila de processos prontos */
    descritor->prox_desc = NULL;

    if(prim == NULL){ /* -> Fila vazia */
        descritor->prox_desc = descritor; /* Como a fila eh circular, aponta para ele mesmo */
        prim = descritor; /* Ele eh o cabeca de fila */
    }

    else{ /* -> Fila povoada */
        PTR_DESC_PROC aux = prim; /* Criacao de um auxiliar para percorrer a
fila */
        while(aux->prox_desc != prim) aux = aux->prox_desc; /* Enquanto a fila nao acabar,
percorre a fila */

        aux->prox_desc = descritor; /* Auxiliar aponta para o processo */
        descritor->prox_desc = prim; /* Fecha a LCSE apontando para o cabeca de
fila */
    }
}
```

Nesse trecho de código, fica evidente a utilização de uma lista circular simplesmente encadeada (LCSE) para representar a fila de processos ativos. A escolha dessa estrutura se dá pois a circularidade da fila simula o ciclo infinito de processos, em que o sistema retorna ao início sem necessidade de manipulações adicionais, permitindo que o escalador lide com os

processos de forma contínua. Além disso, a LCSE também é ótima para a inserção e remoção de elementos em termos de complexidade e escalonamento.

3. Casos de Teste

Considerando que o código já fora construído para suportar o caso de teste base – como explicitado na seção IV do arquivo “Nucleo.C”, a saída produzida é:

