Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências

Enzo Campanholo Paschoalini - RA 221026215
Lucca Comerão Stecca Almeida- RA 221026282
Thiago Bigotte Gullo - RA 221026241

Profº Drº Antônio Carlos Sementille

Documentação do Núcleo Multiprogramado

Bauru, 21 de Julho de 2024

Introdução

Este documento detalha as estruturas e funções utilizadas, além dos testes realizados para validar o código. Estruturas repetidas não são descritas novamente; cada seção detalha apenas as partes novas acrescentadas em cada tipo de código, como no caso do Núcleo simples e Núcleo com semáforo. Os códigos completos estão disponíveis no arquivo .rar enviado no Classroom.

Programas

1. Programa Tic-Tac

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

PTR_DESC c01, c02, prin;
```

O programa se inicia com a inclusão das bibliotecas, das quais a system.h contém as estruturas de dados e as funções necessárias para a implementação das co-rotinas. Depois, declaram-se os ponteiros de descritores de contexto. As variáveis c01, c02 e prin representam, respectivamente, a co-rotina 1, a co-rotina 2 e a rotina main do programa.

```
void far corotina1()
{
 while(1)
 {
  printf("tic-");
  transfer(c01, c02);
 }
}
```

A co-rotina 1 representa a parte do programa que escreverá os “tic” na saída. Inicia-se um loop infinito, na qual o programa imprime “tic-” e transfere o controle para a próxima co-rotina (a co-rotina 2, que representa o “tac”) utilizando a função transfer().

```


void far corotina2()
{
 while(1)
 {
  printf("tac\n");
  transfer(c02,c01);
 }
}
```

A co-rotina 2 representa a parte do programa que escreverá os “tac” na saída. Inicia-se um loop infinito, na qual o programa imprime “tac.”, quebra a linha e transfere o controle para a próxima co-rotina (a co-rotina 1, que representa o “tic”) utilizando a função transfer().

```
main()
{
 clrscr();

 /* cria descritores de co-rotinas */
 c01 = cria_desc();
 c02 = cria_desc();
 prin = cria_desc();

 /* associa descritores com as co-rotinas */
 newprocess(corotina1, c01);
 newprocess(corotina2, c02);

 /* transfere o controle para a co-rotina 1 */
 transfer(prin, c01);
 getch();
}
```

A rotina main é iniciada com um clrscr() para que informações de outros programas não ocupem espaço na tela de saída. Depois, criam-se descritores de contexto que são associados aos seus respectivos ponteiros. Os processos das co-rotinas 1 e 2 são iniciadas associando as co-rotinas aos seus respectivos ponteiros usando a função newprocess(). Por fim, o controle do programa é transferido para a co-rotina 1 utilizando a função transfer(). Para conseguir visualizar a saída do programa, é utilizado o getch().

2. Programa Tic-Tac Modificado

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

/* ponteiros de descritores */
PTR_DESC c01, c02, prin;
```

O programa se inicia de modo semelhante ao Tic-Tac original, com a inclusão das bibliotecas e do system.h. Depois, declaram-se os ponteiros de descritores de contexto. As variáveis c01, c02 e prin representam, respectivamente, a co-rotina 1, a co-rotina 2 e a rotina main do programa.

```
/* co-rotina tic */
void far corotina1()
{
 int i = 0;
```

```
 while(i < 100)
 {
  printf("tic-");
  transfer(c01, c02);
  i++;
 }
}
```

A co-rotina 1 representa a parte do programa que escreverá os “tic” na saída. Inicia-se um laço while. Desta vez, diferente do Tic-Tac original, utiliza-se um contador inteiro para permitir que se escreva 100 vezes o “tic-tac”. Assim, o while() é iniciado com a condição de que executará o laço enquanto o contador for menor ou igual a 100. O programa imprime “tic-” e transfere o controle para a próxima co-rotina (a co-rotina 2, que representa o “tac”) utilizando a função transfer().

```
/* co-rotina tac */
void far corotina2()
{
 while(1)
 {
  printf("tac\n");
  transfer(c02,c01);
 }
}
```

A co-rotina 2 representa a parte do programa que escreverá os “tac” na saída, e é idêntica à co-rotina original. Inicia-se um loop infinito, na qual o programa imprime “tac.”, quebra a linha e transfere o controle para a próxima co-rotina (a co-rotina 1, que representa o “tic”) utilizando a função transfer(). Na última iteração do programa, ele retorna o controle para a co-rotina 1 e, como a co-rotina não executa o laço, o programa é encerrado após 100 iterações.

```
/* Programa Principal */

main()
{
 clrscr();

 /* cria descritores de co-rotinas */
 c01 = cria_desc();
 c02 = cria_desc();
 prin = cria_desc();

 /* associa descritores com as co-rotinas */
 newprocess(corotina1, c01);
 newprocess(corotina2, c02);
```

```
 /* transfere o controle para a co-rotina 1 */
 transfer(prin, c01);
 getch();
}
```

A rotina main é iniciada com um clrscr() para que informações de outros programas não ocupem espaço na tela de saída. Depois, seguindo o algoritmo do Tic-Tac original, criam-se descritores de contexto que são associados aos seus respectivos ponteiros. Os processos das co-rotinas 1 e 2 são iniciadas associando as co-rotinas aos seus respectivos ponteiros usando a função newprocess(). Por fim, o controle do programa é transferido para a co-rotina 1 utilizando a função transfer(). Para conseguir visualizar a saída do programa, é utilizado o getch().

3. Escalonador de Processos

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

PTR_DESC dmain, desc, dX, dY;
/*descritores de contexto */
```

O escalonador de processos se inicia da mesma maneira que os demais programas aqui apresentados. Ponteiros de descritores de contexto são iniciados para o programa principal (dmain), para o escalador (desc), e para co-rotinas X e Y.

```
/* função escalador */
void far escalador() {
    p_est->p_origem = desc;
    p_est->p_destino = dX;
    p_est->num_vetor = 8;

    while(1) {
        iotransfer();
        /* transfere contexto */
        disable();
        if (p_est->p_destino == dX)
            p_est->p_destino = dY;
        /* alterna entre corotinas */
        else
            p_est->p_destino = dX;
        enable();
    }
}
```

A função do escalonador se inicia com a definição do ponteiro de estados, onde a origem é atribuída como a rotina do escalonador, o destino é o processo/co-rotina X (o que sempre será o ponteiro que representa o processo destino - no caso do núcleo, prim->contexto) , e num_vetor, estrutura que serve como valor que controla o timer das interrupções para troca de processos, é iniciado com valor inteiro 8. Depois, é iniciado um laço infinito onde serão alternados os processos. Inicialmente, é chamado a função iotransfer(). Da 1ª. vez que é chamada, instala a co-rotina chamadora como rotina de interrupção associada ao vetor de interrupção passado como parâmetro. Em seguida, transfere o controle para a co-rotina especificada como destino. Ao ocorrer uma interrupção do tipo escolhido, a co-rotina destino é interrompida, seu estado é armazenado no descritor correspondente, a rotina anterior do DOS (antiga do vetor de interrupção) é executada, e o controle retorna para a co-rotina chamadora, após a chamada ao iotransfer().

Depois, no momento em que o programa for entrar em sua região crítica, são desativadas as interrupções. Caso o ponteiro destino for uma co-rotina X, troca-se pela co-rotina Y e vice-versa. Finalmente, são reativadas as interrupções.

```
/* função corotina X */
void far CorotinaX() {
    while(1) {
        printf("corotinaX ");
    }
}

/* função corotina Y */
void far CorotinaY() {
    while(1) {
        printf("corotinaY ");
    }
}

/* função principal */
main() {
    dX = cria_desc();
    /* cria descritor para corotina X */

    dY = cria_desc();
    /* cria descritor para corotina Y */

    desc = cria_desc();
    /* cria descritor para escalador */

    dmain = cria_desc();
    /* cria descritor para main */
```

```
    newprocess(CorotinaX, dX);
    /* cria processo para corotina X */

    newprocess(CorotinaY, dY);
    /* cria processo para corotina Y */

    newprocess(escalador, desc);
   /* cria processo para escalador */

    transfer(dmain, desc);
   /* transfere controle para o escalador */
}
```

Por fim, são criadas duas co-rotinas exemplo para mostrar o funcionamento do escalador. Ambas têm laços infinitos que identificam as co-rotinas através de printf(). Na rotina principal, são criados descritores de contexto para as co-rotinas main, escalador, corotinaX e corotinaY. São associadas às suas respectivas funções através da função newprocess(). O controle é transferido da rotina principal para o escalador.

4. Núcleo Simples

O arquivo nucleo.c implementa um sistema de gerenciamento de processos simples em C. Ele utiliza descritores de processos e um escalador de processos para simular a execução concorrente de múltiplos processos em um ambiente cooperativo.

Estruturas de Dados

regis: Estrutura que armazena dois registros, bx1 e es1;

```
/* estrutura para regiao critica */
typedef struct registros {
    unsigned bx1, es1;
} regis;
```

APONTA_REG_CRIT: União que permite acessar a região crítica como registros ou como um ponteiro far.

```
/* aponta para a regiao critica */
typedef union k {
    regis x;
    char far *y;
} APONTA_REG_CRIT;
```

DESCRITOR_PROC: Estrutura que representa o descritor de um processo, incluindo seu nome, estado, contexto e ponteiros para os próximos descritores de processos.

Funções Principais

insereFila: Insere um novo processo na fila de processos prontos.

```
/* insere um novo processo na fila */
void far insereFila(PTR_DESC_PROC novo) {
    PTR_DESC_PROC prox, ant;
    prox = prim;
    ant = prim;
    if (prox == NULL) {
        prim = novo;
        prim->prox_desc = novo;
    } else {
        prox = prox->prox_desc;
        while (prox != prim) {
            ant = prox;
            prox = prox->prox_desc;
        }
        novo->prox_desc = prim;                    /* insere no final da fila */
        ant->prox_desc = novo;
    }
}
```

criaProcesso: Cria um novo processo e o adiciona à fila de processos prontos.

```
/* cria um novo processo */
void far criaProcesso(char nome_processo[35], void far (*final_processo)()) {
    PTR_DESC_PROC auxiliar;
    auxiliar = (DESCRITOR_PROC *)malloc(sizeof(DESCRITOR_PROC)); /* inicia campos do descritor */
    strcpy(auxiliar->nome, nome_processo);                       /* copia o nome */
    auxiliar->estado = ativo;                                    /* muda o estado para ativo */
    auxiliar->contexto = cria_desc();                            /* inicia o descritor de contexto */
    newprocess(final_processo, auxiliar->contexto);
    insereFila(auxiliar);                                        /* coloca no final da fila de processos */
}
```

procuraProxAtivo: Responsável por percorrer a lista de descritores ativos e retornar o próximo processo ativo, caso não o encontre, retorna NULL.

```
/* procura o proximo processo ativo */
PTR_DESC_PROC far procuraProxAtivo() {
    PTR_DESC_PROC aux;
    if (prim == NULL) return NULL;                  /* retorna nulo */
    aux = prim->prox_desc;                          /* percorre a lista dos descritores ativos */
    while (aux != prim && aux->estado != ativo)
        aux = aux->prox_desc;
    if (aux == prim) {
        if (aux->estado == ativo) return aux;
        else return NULL;                           /* se nao achou retorna nulo */
    }
    return aux;
}
```

voltaDOS: Tem a responsabilidade de restaurar o sistema para o estado original do DOS e encerrar a execução do programa.

```
/* volta para o DOS */
void far voltaDOS() {
    disable();
    setvect(8, p_est->int_anterior);                    /* restaura a interrupcao original */
    enable();
    printf("\t\nFIM");
    getch();
    exit(0);
}
```

escalador: Escalador de processos que alterna entre os processos prontos.

```
/* escalador de processos */
void far escalador() {
    p_est->p_origem = desc;
    p_est->p_destino = prim->contexto;
    p_est->num_vetor = 8;                               /* ponteiro ja na regiao critica do DOS */
    _AH = 0x34;                                         /* procedimento padrao de registradores */
    _AL = 0X00;
    geninterrupt(0x21);
    a.x.bx1 = _BX;
    a.x.es1 = _ES;
    while (1) {
        iotransfer();
        disable();
        if (!*a.y) {

            if ((prim = procuraProxAtivo()) == NULL)    /* se nao esta na regiao critica troca o processo */
                voltaDOS();
            p_est->p_destino = prim->contexto;
        }
        enable();
    }
}
```

startSistema: Inicializa o sistema e dispara o escalador de processos.

```
/* dispara o sistema */
void far startSistema() {
    PTR_DESC daux;
    daux = cria_desc();
    desc = cria_desc();
    newprocess(escalador, desc);
    transfer(daux, desc);
}
```

terminaProcesso: Termina o processo atual, marcando-o como terminado.

```
/* termina o processo atual */
void far terminaProcesso() {
    disable();                                          /* desabilita interrupcoes */
    prim->estado = terminado;                           /* marca o processo como terminado */
    enable();                                           /* habilita interrupcoes */
    while (1);                                          /* loop eterno */
}
```

Programa teste executável:

O programa teste para o algoritmo de núcleo do núcleo foi construído junto com sua lógica de funcionamento. Neste caso foram propostos dois processos distintos: Processo X e processo Y. Na main() o programa é iniciado criando esses dois processos e disparando o sistema.
Logo, Inicializa descritores de processo e cria um processo para o escalador também transferindo o controle de execução para o escalador.

Por sua vez, o escalador alterna entre os processos X e Y, transferindo o contexto de execução. As funções processoX e processoY representam as funções de funcionalidades de um processo qualquer. Para melhor abstração, foi inserido uma ação de impressão de caracteres X e Y respectivamente para os processos:

```
/* representacao de um processo X */
void far processoX() {
    int i = 0;
    while (i < 10000) {
        printf("x");
        i++;
    }
    terminaProcesso();
}

/* representacao de um processo Y */
void far processoY() {
    int i = 0;
    while (i < 10000) {
        printf("y");
        i++;
    }
    terminaProcesso();
}
```

Quando esses processos terminam suas interações, a função terminaProcesso() é chamada, encerrando, assim, seu funcionamento marcando o processo em questão como terminado e chamando o próximo processo ativo para execução.

5. Núcleo Multiprogramado c/ implementação do problema do Produtor/Consumidor usando Semáforos

Semelhante ao Núcleo Simples, o arquivo `nucleose.c` implementa um gerenciador de processos em C. No entanto, além das estruturas utilizadas no Núcleo Simples, este arquivo também incorpora mecanismos de semáforos para gerenciamento de processos. Abaixo serão descritas as estruturas adicionais em relação ao Núcleo Simples.

Estrutura de Dados Acrescentadas

semaforo: Estrutura de dados que armazena o inteiro "s" e um ponteiro de descritor de processos

```c
/* estrutura semáforo */
typedef struct {
    int s;
    PTR_DESC_PROC Q;
} semaforo;
```

Funções Principais Acrescentadas

startSemaforo: Inicia os campos do semáforo com a quantidade que um processo suporta pela passagem de parâmetro "n" e a lista como nula

```c
/* instruções iniciais dos semáforo */
void far startSemaforo(semaforo *sem, int n) {
    sem->s = n;
    sem->Q = NULL;
}
```

removeSemaforo: Remove os processos bloqueados da fila de bloqueados e muda o estado para "ativo"

```c
/* remove processo do semáforo */
void far removeSemaforo(semaforo *sem) {
    PTR_DESC_PROC p_aux;
    p_aux = sem->Q;
    p_aux->estado = ativo;
    sem->Q = p_aux->fila_sem;                    /* remove da fila do semáforo */
}
```

insereSemaforo: Insere um processo bloqueado no final da fila de Bloqueados do semáforo

```c
/* insere processo no semáforo */
void far insereSemaforo(semaforo *sem) {
    PTR_DESC_PROC p_aux;
    p_aux = sem->Q;
    if (p_aux == NULL) {
        sem->Q = prim;
    } else {
        while (p_aux->fila_sem != NULL) {
            p_aux = p_aux->fila_sem;
        }
        p_aux->fila_sem = prim;                  /* insere no final da fila */
        prim->fila_sem = NULL;
    }
}
```

P: Gerencia a entrada de processos em uma seção crítica, garantindo que apenas um número limitado de processos possam acessar um recurso compartilhado ao mesmo tempo, bloqueando aqueles que precisam esperar e insere-os na fila do semáforo.

```
/* P (wait) de espera do semáforo */
void far P(semaforo *sem) {
    PTR_DESC_PROC p_aux;
    disable();

    if (sem->s > 0) {
        sem->s--;
    } else {
        insereSemaforo(sem);                        /* bloqueia processo e insere na fila do semáforo */
        prim->estado = bloq_P;
        p_aux = prim;
        if ((prim = procuraProxAtivo()) == NULL) {
            voltaDOS();
        }
        transfer(p_aux->contexto, prim->contexto);  /* transfere o contexto */
    }
    enable();
}
```

V: Usado para liberar um recurso em um semáforo, o que pode desbloquear processos. Se há processos na fila de espera, V remove um deles, permitindo sua execução.

```
/* V (signal) de largada do semáforo */
void far V(semaforo *sem) {
    disable();
    if (sem->Q == NULL) {
        sem->s++;
    } else {
        removeSemaforo(sem);                        /* remove processo da fila do semáforo */
    }
    enable();
}
```

Programa teste executável:

O programa de teste para o algoritmo do núcleo com semáforos implementa o Problema do Produtor/Consumidor utilizando um buffer de tamanho N=10. Os semáforos usados para sincronização são: “mutex” controla o acesso à região crítica, “empty” rastreia o espaço disponível no buffer e “full” monitora os itens presentes no buffer. Produtores adicionam itens ao buffer, enquanto consumidores removem e processam esses itens.

Uma variável global inteira é usada para imprimir o item consumido do buffer, enquanto outra variável global inteira é utilizada para imprimir o item produzido no buffer, facilitando o rastreamento das operações de consumo e produção. Este problema clássico demonstra a utilização de semáforos para gerenciar a concorrência entre processos produtores e consumidores, garantindo acesso ordenado e seguro ao buffer compartilhado.

```c
/* buffer circular */
#define N 10
int buffer[N];
int in = 0;
int out = 0;

/* semáforos */
semaforo mutex;
semaforo empty;
semaforo full;

/* funções produtor e consumidor */
int produto = 0;
void far produtor() {
    int item;
    while (1) {
        item = rand() % 100;                                /* produz um item */
        P(&empty);                                          /* decrementa empty */
        P(&mutex);                                          /* entra na seção crítica */
        buffer[in] = item;                                  /* coloca o item no buffer */
        in = (in + 1) % N;
        V(&mutex);                                          /* sai da seção crítica */
        V(&full);                                           /* incrementa full */
        printf("Produziu: %d\t qtd: %d \n", item, produto); /* imprime item produzido */
        produto++;
    }
}
int consumo = 0;
void far consumidor() {
    int item;
    while (1) {
        P(&full);                                           /* decrementa full */
        P(&mutex);                                          /* entra na seção crítica */
        item = buffer[out];                                 /* retira o item do buffer */
        out = (out + 1) % N;
        V(&mutex);                                          /* sai da seção crítica */
        V(&empty);                                          /* incrementa empty */
        printf("Consumiu: %d\t qtd: %d \n", item, consumo); /* imprime item consumido */
        consumo++;
    }
}
```

A inicialização dos semáforos com seus respectivos valores é realizada na função main() conforme mostrado abaixo:

```c
/* função principal */
void main() {
    startProntos();                                 /* inicializa a fila de prontos */
    startSemaforo(&mutex, 1);                       /* inicia semáforo mutex */
    startSemaforo(&empty, N);                       /* inicia semáforo empty */
    startSemaforo(&full, 0);                        /* inicia semáforo full */
    criaProcesso("Produtor", produtor);             /* cria processo produtor */
    criaProcesso("Consumidor", consumidor);         /* cria processo consumidor */
    startSistema();                                 /* dispara o sistema */
}
```