Sistemas Operacionais

Comunicação entre processos

Profª Drª Thaína Aparecida Azevedo Tosta

tosta.thaina@unifesp.br

UNIFESP
Universidade Federal de São Paulo

São José dos Campos
2024/01

Aula passada

- Utilização de threads
- O modelo de thread clássico
- Implementando threads no espaço do usuário
- Implementando threads no núcleo
- Implementações híbridas
- Threads pop-up
- Convertendo código de um thread em código multithread

Sumário

- Condições de corrida
- Regiões críticas
- Exclusão mútua com espera ocupada
- Dormir e acordar
- Mutexes e semáforos
- Monitores
- Troca de mensagens
- Barreiras

Objetivo: base teórica para execução de códigos.

Comunicação entre processos

Processos quase sempre precisam comunicar-se com outros processos (*interprocess communication* - IPC), o que envolve:

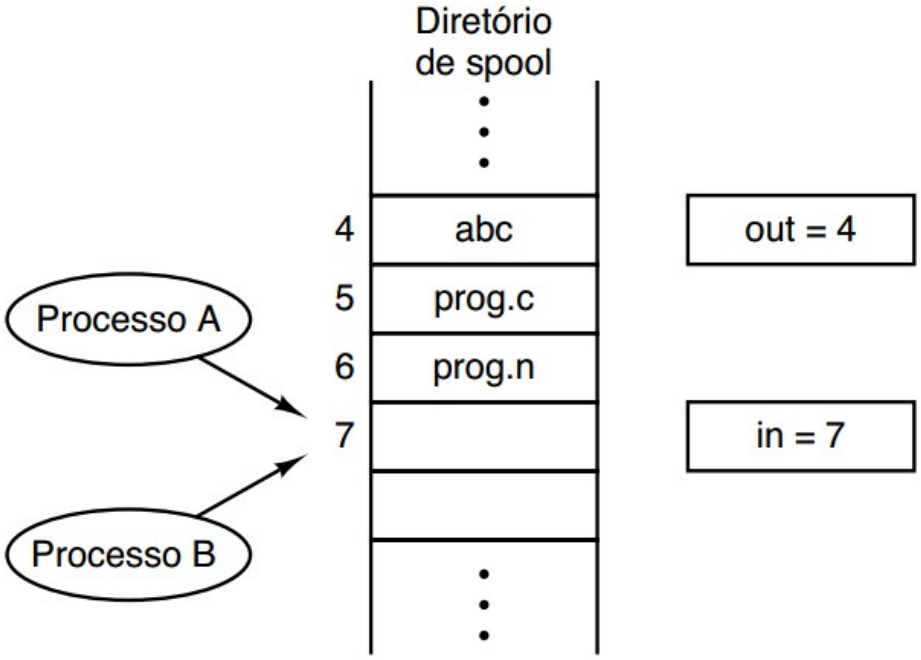
1. Passar informações: maneira bem estrutura e sem interrupções;
2. Certificar-se que dois ou mais processos não se atrapalhem;
3. Sequenciamento adequado quando há dependências.

Esses problemas (e soluções) são também aplicáveis a threads.

Condições de corrida

De maneira mais ou menos simultânea, os processos A e B decidem que querem colocar um arquivo na fila para impressão;

- O processo A lê in e armazena o valor, 7, em uma variável local chamada next_free_slot;
- O processo A tem sua execução interrompida pela CPU;
- O processo B também lê in e recebe um 7 e o armazena em next_free_slot;
- O processo B armazena o nome do seu arquivo na vaga 7 e atualiza in para ser um 8;
- O processo B segue suas demais tarefas;
- O processo A volta e encontra um 7 em next_free_slot e escreve seu nome de arquivo na vaga 7.



Condições de corrida

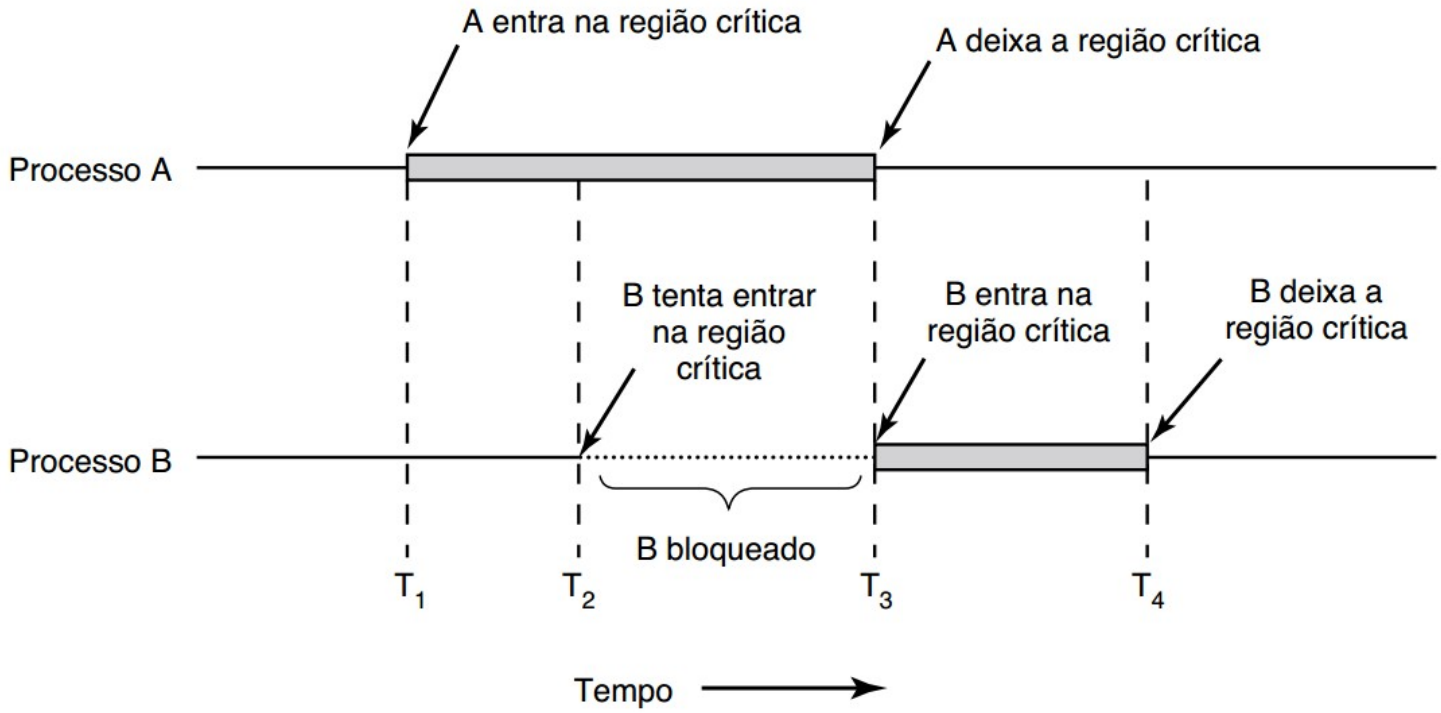
- Situações em que dois ou mais processos estão lendo ou escrevendo alguns dados compartilhados e o resultado final depende de quem executa precisamente e quando, são chamadas de **condições de corrida**;

- Com o incremento do paralelismo pelo maior número de núcleos, as condições de corrida estão se tornando mais comuns.

Regiões críticas

- A chave para evitar problemas com dados compartilhados é proibir mais de um processo de ler e escrever ao mesmo tempo (**exclusão mútua**) questão de projeto do sistema operacional;
- Partes do programa que acessam dados compartilhados são chamadas de **região crítica ou seção crítica**;
- Precisamos fazer com que dois processos jamais estejam em suas regiões críticas ao mesmo tempo.

Regiões críticas



Regiões críticas

Precisamos que quatro condições se mantenham para chegar a uma boa solução:

1. Dois processos jamais podem estar simultaneamente dentro de suas regiões críticas;
2. Nenhuma suposição pode ser feita a respeito de velocidades ou do número de CPUs;
3. Nenhum processo executando fora de sua região crítica pode bloquear qualquer processo;
4. Nenhum processo deve ser obrigado a esperar eternamente para entrar em sua região crítica.

Exclusão mútua com espera ocupada

Alternância explícita

A variável do tipo inteiro turn serve para controlar de quem é a vez de entrar na região crítica e examinar ou atualizar a memória compartilhada:

- O processo 0 reconhece turn=0 e entra na sua região crítica;
- O processo 1 espera em um laço fechado testando continuamente turn (espera ocupada com desperdício de CPU → *spin lock*);
- O processo 0 deixa a região crítica e configura turn para 1, com execução de ambos fora da região crítica;
- O processo 0 volta e reconhe turn=1...

```
while (TRUE) {
        while (turn !=0)            /* laco */;
        critical_region( );
        turn = 1;
        noncritical_region( );
}
```

(a) **Processo 0**

```
while (TRUE) {
        while (turn !=1)            /* laco */;
        critical_region( );
        turn = 0;
        noncritical_region( );
}
```

(b) **Processo 1**

Exclusão mútua com espera ocupada

Alternância explícita

Precisamos que quatro condições se mantenham para chegar a uma boa solução:

1. Dois processos jamais podem estar simultaneamente dentro de suas regiões críticas;
2. Nenhuma suposição pode ser feita a respeito de velocidades ou do número de CPUs;
3. ~~Nenhum processo executando fora de sua região crítica pode bloquear qualquer processo;~~
4. Nenhum processo deve ser obrigado a esperar eternamente para entrar em sua região crítica.

Exclusão mútua com espera ocupada

```c
#include <stdio.h>
#include <pthread.h>

int turn=1;

void critical_region(int pid){
        printf("Regiao critica\t pid=%d turn=%d \n", pid, turn);
        fflush(stdout);
}

void noncritical_region(int pid){
        printf("Fora da regiao critica\t pid=%d turn=%d \n", pid, turn);
        fflush(stdout);
}

#define ALTERNA(turn) (((turn) == 0)?(1):(0))

void *proc(void *args) {
        int pid = *(int*)args;
        while(1){
                while(turn != pid);
                critical_region(pid);
                turn = ALTERNA(turn);
                noncritical_region(pid);
        }
}

int main(int argc, char **argv){
        pthread_t pt0, pt1;
        int pid0 = 0, pid1 = 1;
        pthread_create(&pt0, NULL, proc, &pid0);
        pthread_create(&pt1, NULL, proc, &pid1);
        pthread_join(pt0, NULL);
        pthread_join(pt1, NULL);
        return 0;
}
```

```
tostathaina@tostathaina-Predator-PH315...
Fora da regiao critica   pid=1 turn=0
Regiao critica   pid=0 turn=0
Fora da regiao critica   pid=0 turn=1
Regiao critica   pid=1 turn=1
Fora da regiao critica   pid=1 turn=0
Regiao critica   pid=0 turn=0
Fora da regiao critica   pid=0 turn=1
Regiao critica   pid=1 turn=1
Fora da regiao critica   pid=1 turn=0
Regiao critica   pid=0 turn=0
Fora da regiao critica   pid=0 turn=1
Regiao critica   pid=1 turn=1
Fora da regiao critica   pid=1 turn=0
Regiao critica   pid=0 turn=0
Fora da regiao critica   pid=0 turn=1
Regiao critica   pid=1 turn=1
Fora da regiao critica   pid=1 turn=0
Regiao critica   pid=0 turn=0
Fora da regiao critica   pid=0 turn=1
```

```
tostathaina@tostathaina-Predator-PH315-54: ~/Documentos
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ ./busy-wait | grep "Fora da
 regiao critica" | grep "pid=0" | grep "turn=0"
Fora da regiao critica   pid=0 turn=0
^C
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ ./busy-wait | grep "Fora da
 regiao critica" | grep "pid=1" | grep "turn=1"
Fora da regiao critica   pid=1 turn=1
Fora da regiao critica   pid=1 turn=1
Fora da regiao critica   pid=1 turn=1
^C
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$
```

Quais as condições de inconsistência e por que elas acontecem?

Exclusão mútua com espera ocupada

Solução de Peterson

Variável compartilhada →
Posições não compartilhadas →

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* numero de processos */

int turn;                               /* de quem e a vez? */
int interested[N];                      /* todos os valores 0 (FALSE) */

void enter_region(int process);         /* processo e 0 ou 1 */
{
    int other;                          /* numero do outro processo */

    other = 1 – process;                /* o oposto do processo */
    interested[process] = TRUE;         /* mostra que voce esta interessado */
    turn = process;                     /* altera o valor de turn */
    while (turn == process && interested[other] == TRUE) /* comando nulo */ ;
}

void leave_region(int process)          /* processo: quem esta saindo */
{
    interested[process] = FALSE;        /* indica a saida da regiao critica */
}
```

Exclusão mútua com espera ocupada

A instrução TSL (*Test and Set Lock*, em hardware)

Lê o conteúdo da palavra `lock` da memória para o registrador `RX` e então armazena um valor diferente de zero no endereço de memória `lock`.

```
enter_region:
        TSL REGISTER,LOCK       | copia lock para o registrador e poe lock em 1
        CMP REGISTER,#0         | lock valia zero?
        JNE enter_region        | se fosse diferente de zero, lock estaria ligado; portanto,
                                  continue no laco de repeticao
        RET                     | retorna a quem chamou; entrou na regiao critica


leave_region:
        MOVE LOCK,#0            | coloque 0 em lock
        RET                     | retorna a quem chamou
```

Dormir e acordar

- A espera ocupada desperdiça tempo de CPU e pode gerar o problema de inversão de prioridade:
  - Processo H de alta prioridade é executado sempre que estiver pronto (pelo escalonamento);
  - Processo L de baixa prioridade fica na região crítica de H e não recebe a chance de deixá-la.

- Alternativa: chamadas de sistema `sleep` e `wakeup` (comunicação entre processos que bloqueiam quando não são autorizados a entrar nas suas regiões críticas).

Dormir e acordar

O problema do produtor-consumidor

- Também conhecido como problema do buffer limitado;
- Problemas:
  - Buffer cheio: produtor não consegue colocar novo item e vai dormir, sendo desperto quando o consumidor remover um ou mais itens;
  - BUffer vazio: consumidor não tem item para remover e vai dormir até o produtor colocar algo no buffer.

Dormir e acordar

O problema do produtor-consumidor

Se o sinal de despertar é perdido? Bit de espera pelo sinal de acordar.

```
#define N 100                                   /* numero de lugares no buffer */
int count = 0;                                  /* numero de itens no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repita para sempre */
          item = produce_item( );               /* gera o proximo item */
          if (count == N) sleep( );             /* se o buffer estiver cheio, va dormir */
          insert_item(item);                    /* ponha um item no buffer */
          count = count + 1;                    /* incremente o contador de itens no buffer */
          if (count == 1) wakeup(consumer);     /* o buffer estava vazio? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repita para sempre */
          if (count == 0) sleep( );             /* se o buffer estiver cheio, va dormir */
          item = remove_item( );                /* retire o item do buffer */
          count = count – 1;                    /* descresca de um contador de itens no buffer */
          if (count == N – 1) wakeup(producer); /* o buffer estava cheio? */
          consume_item(item);                   /* imprima o item */
     }
}
```

Mutexes e semáforos

- **Mutexes** são bons somente para gerenciar a exclusão mútua de algum recurso ou trecho de código compartilhados (variável count);
- São fáceis e eficientes de implementar, e úteis em threads inteiramente no espaço do usuário;
- É uma variável compartilhada binária (0 - destravado, 1 - travado).

Mutexes e semáforos

- Qual a diferença entre `mutex_lock` e `enter_region`?
- Como os threads ficam na espera ocupada?

```
mutex_lock:
    TSL REGISTER,MUTEX      | copia mutex para o registrador e atribui a ele o valor 1
    CMP REGISTER,#0         | o mutex era zero?
    JZE ok                  | se era zero, o mutex estava desimpedido, portanto retorne
    CALL thread_yield       | o mutex esta ocupado; escalone um outro thread
    JMP mutex_lock          | tente novamente
ok: RET                     | retorna a quem chamou; entrou na regiao critica

mutex_unlock:
    MOVE MUTEX,#0           | coloca 0 em mutex
    RET                     | retorna a quem chamou
```

Mutexes e semáforos

O **semáforo** mutex recebe 0 quando o processo entra na região crítica, e o semáforo bloqueia quem o chama nessa condição.

```
#define N 100                                   /* numero de lugares no buffer */
typedef int semaphore;                          /* semaforos sao um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso a regiao critica */
semaphore empty = N;                            /* conta os lugares vazios no buffer */
semaphore full = 0;                             /* conta os lugares preenchidos no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE e a constante 1 */
          item = produce_item( );                /* gera algo para por no buffer */
          down(&empty);                          /* decresce o contador empty */
          down(&mutex);                          /* entra na regiao critica */
          insert_item(item);                     /* poe novo item no buffer */
          up(&mutex);                            /* sai da regiao critica */
          up(&full);                             /* incrementa o contador de lugares preenchidos */
     }
}
```

```
void consumer(void)
{
     int item;

     while (TRUE) {                             /* laco infinito */
          down(&full);                           /* decresce o contador full */
          down(&mutex);                          /* entra na regiao critica */
          item = remove_item( );                 /* pega item do buffer */
          up(&mutex);                            /* sai da regiao critica */
          up(&empty);                            /* incrementa o contador de lugares vazios */
          consume_item(item);                    /* faz algo com o item */
     }
}
```

Monitores

- Suponha que os dois downs no código do produtor fossem invertidos ou mutex inicializado com 0 → impasse (*deadlock*) → cuidado com semáforos;
- O monitor pode ser chamado por qualquer processo, mas eles não podem acessar diretamente as estruturas de dados internos do monitor;
- Apenas um processo pode estar ativo em um monitor em qualquer dado instante, com implementação pelo compilador.

```
monitor example
    integer i;
    condition c;

    procedure producer ( );
    .
    .
    .
    end;

    procedure consumer ( );
    .       .       .
    end;
end monitor;
```

Monitores

```
monitor ProducerConsumer
    condition full, empty;
    integer count,

    procedure insert(item: integer);
    begin
        if count= N then wait (full);
        insert_item(item);
        count: = count + 1;
        if count = 1 then signal (empty)
    end ;

    function remove:integer;
    begin
        if count = 0 then wait (empty);
        remove = remove_item;
        count: = count– 1;
        if count  = N – 1 then signal (full)
    end;

    count:= 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
Fora da região crítica →  item = produce_item;
Dentro da região crítica →  ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
    while  true do
    begin
Dentro da região crítica →  item = ProducerConsumer.remove;
Fora da região crítica →  consume_item(item)
    end
end ;
```

Troca de mensagens

- Para um sistema distribuído consistindo em múltiplas CPUs, cada uma com sua própria memória privada e conectada por uma rede de área local, semáforos e monitores não são utilizáveis;
- Assim, a comunicação entre processos acontece por troca de mensagens (chamadas de sistema facilmente colocadas em bibliotecas);
- Questão importante: sincronização.

Troca de mensagens

```
#define N 100                                   /* numero de lugares no buffer */

void producer(void)
{
     int item;
     message m;                                 /* buffer de mensagens */

     while (TRUE) {
          item = produce_item( );               /* gera alguma coisa para colocar no buffer */
          receive(consumer, &m);                /* espera que uma mensagem vazia chegue */
          build_message(&m, item);              /* monta uma mensagem para enviar */
          send(consumer, &m);                   /* envia item para consumidor */
     }
}
```

```
void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* envia N mensagens vazias */
     while (TRUE) {
          receive(producer, &m);                  /* pega mensagem contendo item */
          item = extract_item(&m);                /* extrai o item da mensagem */
          send(producer, &m);                     /* envia a mensagem vazia como resposta */
          consume_item(item);                     /* faz alguma coisa com o item */
     }
}
```

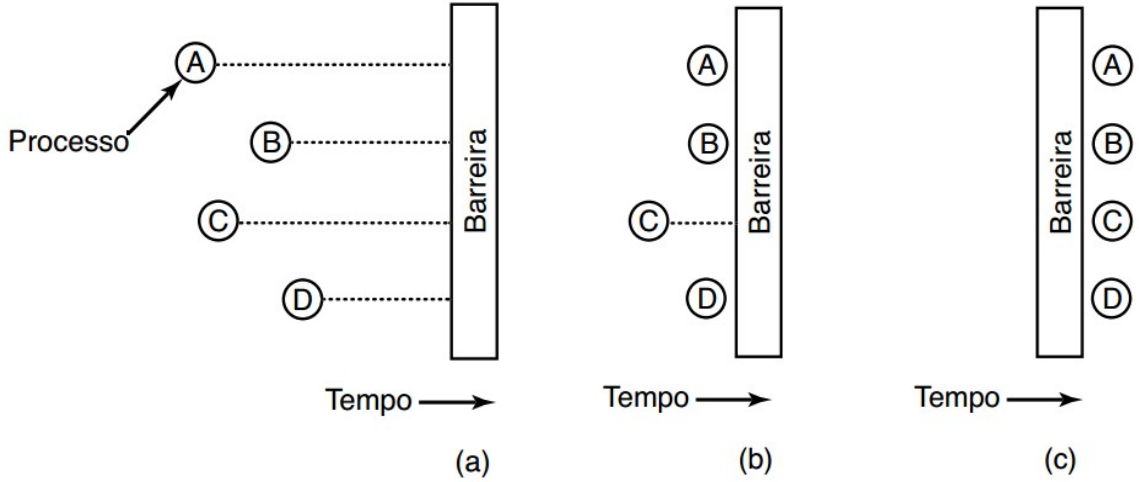
Barreiras

- Algumas aplicações são divididas em fases e nenhum processo deve prosseguir até que todos estejam prontos para isso → barreira;
- Quando um processo atinge a barreira, ele é bloqueado até que todos os processos a tenham atingido → sincronização.

(a) Todos ainda não chegaram ao fim da fase;

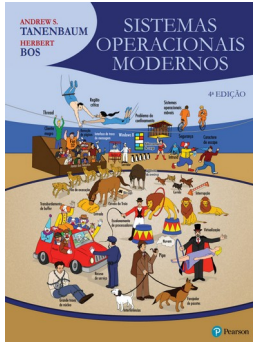
(b) Quem termina, executa barrier (geralmente por procedimento de biblioteca) e é suspenso;

(c) Todos os processos são liberados.



Objetivo: base teórica para execução de códigos.

Referências



TANENBAUM, Andrew S.; BOS, Herbert. Sistemas operacionais modernos. 4. edição. São Paulo: Pearson, 2016. xviii, 758 p. ISBN 9788543005676.