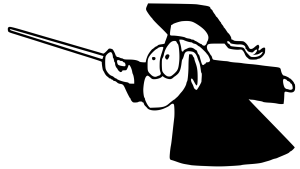
Gerência de Processos - IPC - Implementação

EMB5632 - Sistemas Operacionais
Prof. Dr. Ricardo José Pfitscher
ricardo.pfitscher@ufsc.br

Objetivos de aprendizagem



- Conhecer técnicas para implementar a sincronização de processos

Exercício

- Considere um programa com duas threads, onde, cada uma delas deveria somar 1 NITER vezes ao valor de uma variável global iniciada em zero. O resultado esperado é que a variável global seja igual a NITER*2

- Compile e execute o arquivo “exercicio1.c” disponível no moodle.
  - No Linux:
    - `gcc exercicio1.c -o exercicio1 -lpthread`

- Execute o código e avalie o que pode estar acontecendo

Exercício – Resolvendo o problema

- Para resolver o problema ocorrido, precisamos garantir a exclusão mútua.
- Para tanto, devemos utilizar uma das técnicas estudadas para comunicação entre processos.

- Primeiro, tente resolver utilizando variável de impedimento com espera ocupada.
  - `while (lock == 0 ); lock=0; //código`
- Compile e execute o código algumas vezes, se possível, monitore o consumo de CPU
  - No Linux: digite `top` em outra aba
    - Lembre de colocar o programa em background

Cronograma

- Exclusão mútua sem espera ocupada
- Dormir e acordar
- Semáforos
- Monitores
- Troca de mensagens
- Barreiras

Exclusão mútua sem espera ocupada [1/3]

- O uso de espera ocupada apresenta alguns problemas:
  - Ineficiência:
    - Uso de CPU desnecessária:
      - Um loop vazio ocupa o processador
    - Evita que outros processos executem
  - Inversão de **prioridade**
    - Um processo mais prioritário fica no loop e um menos prioritário não consegue sair de sua seção crítica

Exclusão mútua sem espera ocupada [2/3]

- Exemplo de Inversão de prioridade
  - Dois processos:
    - *H* (mais prioritário) e *L* (menos prioritário)
  - Regra de escalonamento:
    - *H* executa sempre que estiver pronto

H

1. Bloqueando aguardando E/S
2. Recebeu E/S Pronto para executar
3. Foi escalonado e executa espera ocupada
4. Vai executar o *loop* infinitamente

L

1. Executando seção crítica
2. Ainda em seção crítica
3. Ainda em seção crítica
4. Não vai ter oportunidade de terminar a seção crítica

Exclusão mútua sem espera ocupada [3/3]

- Solução ideal:
  - O processo que encontra a seção crítica ocupada deve ficar bloqueado até que a seção crítica seja liberada
- Possíveis soluções
  - Dormir e acordar
  - Semáforos
  - Monitores
  - Troca de mensagens
  - Barreiras

Dormir e acordar [1/4]

- *Sleep* e *Wakeup*
  - Primitivas simples que causam bloqueio e desbloqueio de processos
  - *sleep()*
    - Bloqueia quem chama
  - *wakeup(ID)*
    - Desbloqueia o processo especificado
  - Variação:
    - *var* casa um com o outro
    - *sleep(var) && wakeup(var)*

Dormir e acordar [2/4]

- Problema do produtor–consumidor
  - Dois processos compartilham um *buffer* limitado e de tamanho fixo
  - Produtor:
    - Insere itens no *buffer*
    - Se o *buffer* cheio não pode produzir
  - Consumidor:
    - Retira itens do *buffer*
    - Se o *buffer* vazio não têm o que consumir
  - Muitas situações de IPC se enquadram
    - Ex. Servidor *web multithread*
      - Despachante produz requisições
      - Operário consome requisições

Dormir e acordar [3/4]

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

void producer(void)
{
     int item;

     while (TRUE) {                             /* número de itens no buffer */
          item = produce_item( );               /* gera o próximo item */
          if (count == N) sleep( );             /* se o buffer estiver cheio, vá 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 vazio, vá dormir */
          item = remove_item( );                /* retire o item do buffer */
          count = count – 1;                    /* decresça de um o contador de itens no buffer */
          if (count == N – 1) wakeup(producer); /* o buffer estava cheio? */
          consume_item(item);                   /* imprima o item */
     }
}
```

Dormir e acordar [4/4]

- Pode gerar condição de disputa
  - Buffer vazio e count 0

Consumidor	Produtor
1. Leu que *count* é 0	1. Parado
2. Ação do escalonador	2. Começa a executar
3. .	3. Adiciona item no *buffer* e *count* = *1*
4. . .	4. Achando que o consumidor estava dormindo, chama wakeup para acordar
5. Sabe que *count* é zero e vai dormir .	5. Em algum momento que o buffer estiver cheio também dormirá
6. Dormindo	6. Dormindo

Dormir e acordar [4/4]

- Pode gerar condição de disputa
  - Buffer vazio e count 0

Consumidor	Produtor
1. Leu que *count* é 0	1. Parado
2. Ação do escalonador	2. Começa a executar
3. .	3. Adiciona item no *buffer* e *count* = *1*
4. . .	4. Achando que o consumidor estava dormindo, chama wakeup para acordar
5. Sabe que *count* é zero e vai dormir .	5. Em algum momento que o buffer estiver cheio também dormirá
6. Dormindo	6. Dormindo

Solução: Marcar um wakeup() perdido

Semáforos [1/8]

- Solução proposta por E.W. Dijkstra nos anos 60
  - Pode ser visto como uma variável *S*
    - Representa a seção crítica
    - Conteúdo não acessível ao programador
  - Variável inteira (semáforo) para contar o número de *wakeup()s* pendentes
  - Duas funções: down e up, generalizações de *sleep* e *wakeup*
    - Alterar o valor da variável

Semáforos [2/8]

- Uso de down e sleep:
  - *down(S):*
    - Solicita acesso a seção crítica associada;
      - Se estiver livre, executa, senão fica em espera e é adicionada à fila do semáforo
    - Decrementa S, se o semáforo **for menor que 0**, será bloqueado (dormirá)
  - *up(S):*
    - Libera a seção crítica associada a S
    - Incrementa S; se S<=0, acorda **um** processo que espera por S
      - O primeiro da fila

Isto é um pouco diferente do que diz o Tanenbaum

Semáforos [2/8]

- Uso de down e sleep:
  - *down(S):*
    - Solicita acesso a seção crítica associada;
      - Se estiver livre, executa, senão fica em espera e é adicionada à fila do semáforo
    - Decrementa S, se o semáforo **for menor que 0**, será bloqueado (dormirá)
  - *up(S):*
    - Libera a seção crítica associada a S
    - Incrementa S; se S<=0, acorda **um** pr
      - O primeiro da fila

Se S está negativo, é porque alguém estava esperando pelo semáforo!

Isto é um pouco diferente do que diz o Tanenbaum

Sem

- Uso
  - *do*
    - S
    - [
  - *up*
    - I
    - I

Require: as operações devem executar atomicamente

t: tarefa que invocou a operação
s: semáforo, contendo um contador e uma fila

```
1:  procedure DOWN(t, s)
2:      s.counter ← s.counter − 1
3:      if s.counter < 0 then
4:          append (t, s.queue)          ▷ põe t no final de s.queue
5:          suspend (t)                  ▷ a tarefa t perde o processador
6:      end if
7:  end procedure

8:  procedure UP(s)
9:      s.counter ← s.counter + 1
10:     if s.counter ≤ 0 then
11:         u = first (s.queue)          ▷ retira a primeira tarefa de s.queue
12:         awake(u)                     ▷ devolve u à fila de tarefas prontas
13:     end if
14: end procedure
```

mirá)

Fonte: Maziero (2019)

Semáforos [3/8]

- Implementação de Semáforos [1/2]
  - São implementados no núcleo do SO
  - Semáforo é uma variável inteira
  - As operações *down()* e *up()* executam de forma **atômica**

Semáforos [4/8]

- Implementação de semáforos [2/2]
  - Semáforos binários (mutex)
    - Inicializados em 1
    - Controlam acesso à seção crítica
    - Cada processo faz *down(mutex)* quando entra seção crítica e *up(mutex)* quando sai
  - Semáforos contadores
    - Sincronizam processos
    - Determinam a ordem de execução

Semáforos [5/8]

- Exemplo do produtor-consumidor [1/2]
  - Variáveis Globais

```
#define N 100                          /* número de lugares no buffer */
typedef int semaphore;                 /* semáforos são um tipo especial de int */
semaphore mutex = 1;                   /* controla o acesso à região crítica */
semaphore empty = N;                   /* conta os lugares vazios no buffer */
semaphore full = 0 ;                   /* conta os lugares preenchidos no buffer */
```

Semáforos [6/8]

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

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

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

     while (TRUE) {                    /* laço infinito */
          down(&full);                 /* decresce o contador full */
          down(&mutex);                /* entra na região crítica */
          item = remove_item( );       /* pega o item do buffer */
          up(&mutex);                  /* deixa a região crítica */
          up(&empty);                  /* incrementa o contador de lugares vazios */
          consume_item(item);          /* faz algo com o item */
     }
}
```

Semáforos [7/8]

- Um exemplo mais simples
  - Um estacionamento

```
1  sem_t vagas = 500 ;
2
3  void carro_entra ()
4  {
5     down (vagas) ;        // solicita uma vaga de estacionamento
6     ...                   // demais ações específicas da aplicação
7  }
8
9  void carro_sai ()
10 {
11    up (vagas) ;          // libera uma vaga de estacionamento
12    ...                   // demais ações específicas da aplicação
13 }
```

Fonte: Livro Carlos A Maziero, Ver Bibliografia

Semáforos [8/8]

- Outro exemplo
  - Alterando valores em uma conta

```
1  typedef struct conta_t
2  {
3     int saldo ;              // saldo atual da conta
4     sem_t s = 1;             // semáforo associado à conta, valor inicial 1
5     ...                      // outras informações da conta
6  } conta_t ;
7
8  void depositar (conta_t * conta, int valor)
9  {
10    down (conta->s) ;        // solicita acesso à conta
11    conta->saldo += valor ; // seção crítica
12    up (conta->s) ;          // libera o acesso à conta
13 }
```

Fonte: Livro Carlos A Maziero, Ver Bibliografia

Monitores [1/7]

- O uso de semáforos de forma errada pelo programador poderia causar erros
  - Ex:
    - *down(mutex)* antes de *down(empty)* e o buffer estivesse cheio
      - Produtor bloqueado com *empty* em 0
    - Consumidor também bloqueado, pois ao executar *down(mutex),* o *mutex* já seria 0
    - Isto é um *deadlock*
  - A partir daí surgiu a ideia de se utilizar monitores

Monitores [2/7]

- Sincronização baseada em bibliotecas de linguagem de programação
- O monitor encapsula dados privados e procedimentos que o acessam
- A manipulação dos monitores é feita através de procedimentos criados, nunca através de acessos diretos as suas estruturas
  - Semelhante a uma classe
- Apenas um processo pode ter acesso ao monitor em um dado instante
  - O Controle é feito a nível de linguagem
  - Exclusão mútua entre processos
    - O compilador insere semáforos no código

Monitores [3/7]

- Sincronização é feita usando variáveis de condição
  - Duas operações *wait()* e *signal()* fazem os tratamentos sobre elas
- No exemplo do produtor-consumidor:
  - Produtor: percebendo que o buffer está cheio, pode dar um *wait(full)*, *full* é uma variável condicional (vai aguardar até alguém dar o sinal)
  - Consumidor: Ao retirar os itens faz um *signal(full)*

Monitores [4/7]

- Semântica de *signal()*
  - Evitar que dois processos permaneçam no monitor ao mesmo tempo
  - O que acontece quando um processo executa *signal()?*
    - Primeira proposta (signal-and-exit)
      - O processo atual deve sair do monitor após o signal()
      - O processo sinalizado entra no monitor
    - Segunda proposta (signal-and-continue)
      - O processo atual continua executando
      - O processo sinalizado compete pelo monitor no próximo escalonamento

Monitores [5/7]

```
monitor ProducerConsumer
  condition full, empty;
  integer count := 0;

  procedure enter;
  begin
    if count = N then wait(full);
    enter_item;
    count := count + 1;
    if count = 1 then signal(empty);
  end;

  procedure remove;
  begin
    if count = 0 then wait(empty);
    remove_item;
    count := count - 1;
    if count = N-1 then signal(full);
  end;
end monitor;
```

```
procedure producer;
begin
  while true do
  begin
    produce_item;
    ProducerConsumer.enter;
  end
end;

procedure consumer;
begin
  while true do
  begin
    ProducerConsumer.remove;
    consume_item;
  end
end;
```

Monitores [6/7]

- Monitores em Java
  - Suporte parcial a monitores, usando métodos **synchronized**
  - Java garante exclusão mútua no acesso ao objeto
    - Em métodos synchronized
    - E se não for synchronized?
  - Operações:
    - wait(), notify() e notifyAll()
  - Semântica:
    - signal-and-continue

Monitores [6/7]

- Monitores em Java (Exemplo banco)

```
1  class Conta
2  {
3     private float saldo = 0;
4
5     public synchronized void depositar (float valor)
6     {
7        if (valor >= 0)
8           saldo += valor ;
9        else
10          System.err.println("valor negativo");
11    }
12
13    public synchronized void retirar (float valor)
14    {
15       if (valor >= 0)
16          saldo -= valor ;
17       else
18          System.err.println("valor negativo");
19    }
20 }
```

Monitores [7/7]

- Monitores vs Semáforos
  - Monitores são mais fáceis de programar e reduzem as possibilidades de erro
    - Ordem de *down()* e *up()*
  - Monitores dependem da linguagem de programação enquanto Semáforos são implementados pelo SO
    - Podem ser usados com C, Java, BASIC, ASM, ...
  - Ambos são usados em sistemas centralizados (memória compartilhada)
    - Sistemas distribuídos usam troca de mensagens

Troca de Mensagens [1/2]

- Baseadas em primitivas *send()* e *receive()*
- Usado em sistemas distribuídos
  - Cliente-servidor
- Questões de projeto
  - Confiabilidade (perda ou duplicação de pacotes)
    - Confirmação de recebimento
    - Distinção entre mensagem nova e retransmissão
  - Desempenho em sistemas centralizados
    - Overhead para cópias de mensagens é maior que para semáforos e monitores

Troca de Mensagens [2/2]

```
#define N 100                                   /* número 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 */
     }
}
```

Semáforos

- O consumo excessivo de CPU não é interessante.
- Ainda pode haver condições de corrida
- Podemos utilizar semáforos
  - Bloqueia o processo que se depara com a seção crítica fechada.

Semáforos - Biblioteca

- Vamos utilizar a biblioteca <semaphore.h>
  - http://www.csc.villanova.edu/~mdamian/threads/posixsem.html
- Alternativamente podemos utilizar <pthread.h>
  - https://www.ibm.com/docs/pt-br/aix/7.2.0?topic=programming-using-mutexes
    - Mutexes assumem somente dois valores, aberto ou fechado!

Semáforos - Sintaxe

- Declaração
  - `sem_t sem_name;`
- Inicialização sintaxe:
  - `int sem_init(sem_t *sem, int pshared, unsigned int value);`
    - `sem points` qual objeto semáforo será inicializado
    - `pshared` uma flag que indica se um semáforo pode ser compartilhado por processos fork()ed .
    - `value` valor inicial do semáforo.
- Inicialização exemplo:
  - `sem_init(&sem_name, 0, 10);`
- Down() e up():
  - `sem_wait(&sem_name); //DOWN`
  - `sem_post(&sem_name); //UP`

Semáforos - Sintaxe

- Lendo o valor do semáforo:
  - `int sem_getvalue(sem_t *sem, int *valp);`
  - Captura o valor do semáforo *sem* e armazena na variável *valp*

- Destruindo um semáforo:
  - `sem_destroy(&sem_name);`

Exercício 1 – Usando semáforos

- Resolva o problema utilizando semáforos
- Para compilar usando a biblioteca *semaphore.h* no Linux é necessário usar a flag *–lrt* como parâmetro de configuração
  - `gcc -o filename filename.c -lpthread -lrt`
- Responda: Existe diferença para a implementação com espera ocupada? qual?

Referências

- *MAZIERO, C. Sistemas Operacionais: Conceitos e Mecanismos. Editora da UFPR, 2019. 456 p. ISBN 978-85-7335-340-2*
- Andrew S. Tanenbaum. Sistemas Operacionais Modernos, 3a Edição. Capítulo 2.Pearson Prentice-Hall, 2009.