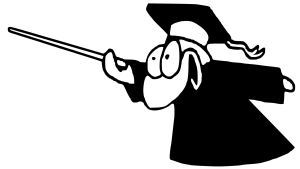
Gerência de Processos - IPC

EMB5632 - Sistemas Operacionais

Prof. Dr. Ricardo José Pfitscher
ricardo.pfitscher@ufsc.br

Objetivos de aprendizagem



- Conhecer os principais métodos de comunicação entre processos;
- Introduzir os métodos de sincronização de processos

UFSC

Cronograma

- Introdução
  - Processos
  - Comunicação
- Desafios
- Técnicas de Comunicação
- Técnicas de Sincronização
  - Exclusão mútua
  - Ordem de eventos

Revisão – última aula

- Processos são programas em execução

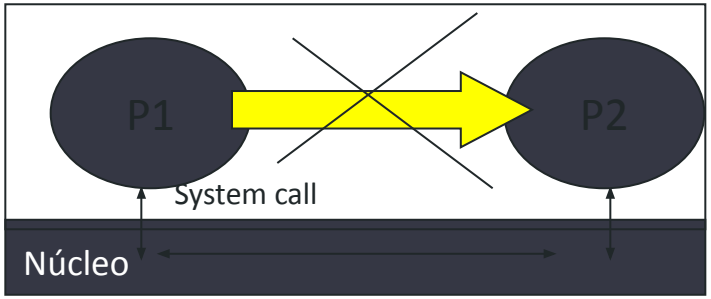
- Possuem um espaço de endereçamento próprio
  - Local da memória de acesso restrito ao processo para armazenamento de variáveis

- Em termos de SO são armazenados em uma tabela de processos
  - Entre outras informações contém o estado de execução
    - Pronto, Bloqueado, Executando

- Os processos de um sistema precisam interagir
  - Enviar informações, Executar outro programa, Aguardar dados de outro programa, Aguardar eventos de outro programa, etc...
  - Exemplos?
    - Imprimir NF em PDF, Monitorar uma variável global (sistema de sensores)

Introdução - Comunicação

- Os processos de um sistema executam em “cápsulas-autônomas”
  - Variáveis internas são de acesso restrito



- A interação entre os processos ocorre por meio de mecanismos de IPC (*Inter Process Comunication*)
  - Memória compartilhada
    - Espaço de endereçamento compartilhado
  - Mecanismos de S.O para transportar dados de um processo para o outro

Desafios

- Existem alguns desafios quanto a comunicação entre processos
  1. **Como a comunicação pode ser feita?**
  2. Como garantir que dois processos não invadam um ao outro, quando estiverem acessando uma **região crítica.**
     - Ex.: Processo A altera o valor de uma variável utilizada por B.
  3. Como garantir a execução dos processos em uma sequência adequada.
     - Ex.: Processo A é responsável por imprimir arquivos e Processo B coloca os arquivos para imprimir, A não pode executar antes de B

UFSC

Comunicação [1/7]

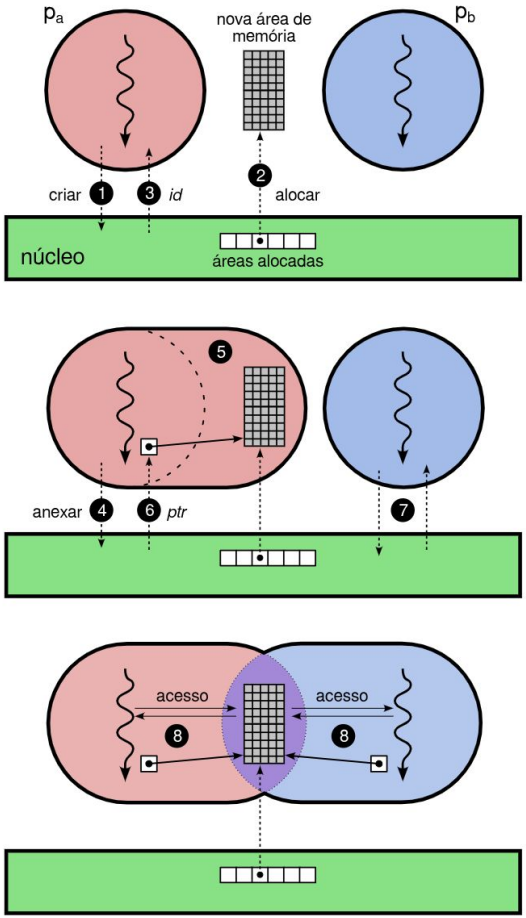
- Quais são as formas de realizar comunicação entre processos?
  - Memória compartilhada
  - Pipe
  - Chamada com passagem de argumentos
  - Troca de mensagens

Comunicação [2/7]

- Memória compartilhada
  - O espaço de endereçamento de um processo é compartilhado a outro processo
  - O SO é responsável por oferecer o espaço compartilhado e gerenciar o acesso

Comunicação [3/7]

- Memória compartilhada
  - Funcionamento:
    - Dois processos A e B
    - **A** solicita ao núcleo do SO a criação de uma área compartilhada, informando o tamanho necessário e as permissões de acesso
    - Após criada, **A** solicita ao núcleo que anexe esta área compartilhada a seu espaço de endereçamento
    - O processo **B** recebe um ponteiro para a área de memória criada por **A**, também solicita ao SO que anexe esta área a seu espaço de endereçamento
    - Ambos tem acesso a mesma área de memória e trocam informações



Comunicação [4/8]

- Memória compartilhada
  - https://www.geeksforgeeks.org/ipc-shared-memory/
  - *ftok():* cria uma chave única para acesso à memória compartilhada
  - *shmget():* int shmget(key_t,size_tsize,intshmflg); se executar com sucesso, shmget() retorna o identificador do segmento de memória compartilhado
  - *shmat():* O processo precise se conectar ao segmento de memória. void *shmat(int shmid ,void *shmaddr ,int shmflg); *shmid* é o ID do espaço de memória compartilhado. *shmaddr* define o endereço de memória específico, quando zero o SO escolhe o endereço.
  - *shmdt():* utilizado para desconectar da região de memória compartilhada. int shmdt(void *shmaddr);
  - *shmctl()*: utilizado para destruir o espaço reservado. shmctl(int shmid,IPC_RMID,NULL);

Comunicação [4/8]

- Memória compartilhada (exemplo) - Moodle

exemplo-mem-shared.c > main(int, char * [])

```c
// Arquivo exemplo-mem-shared.c: cria e usa uma área de memória compartilhada POSIX.
// Em Linux, compile usando: cc -Wall exemplo-mem-shared.c -o shm -lrt

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#define produtor 1 // 1 para produtor, 0 para consumidor


int main (int argc, char *argv[]){
    int fd, value, *ptr;

    fd = shm_open ("/sharedmem", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    if (fd == -1){
        perror("shm_open");
        exit(1);
    }

    if (ftruncate (fd, sizeof(value)) == -1){
        perror("ftruncate");
        exit(1);
    }

    ptr = mmap( NULL, sizeof(value), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if(ptr == MAP_FAILED){
        perror ("mmap");
        exit(1);
    }

    for (;;){
        //PRODUTOR:
        if (produtor){
            value = random() %1000;
            (*ptr) = value;
            printf("Valor escrito: %d\n", value);
            sleep(1);
        }
        //CONSUMIDOR:
        else{
            value = (*ptr) ;
            printf("Valor lido %d\n", value);
            sleep(1);
        }
    }

}
```

Comunicação [5/8]

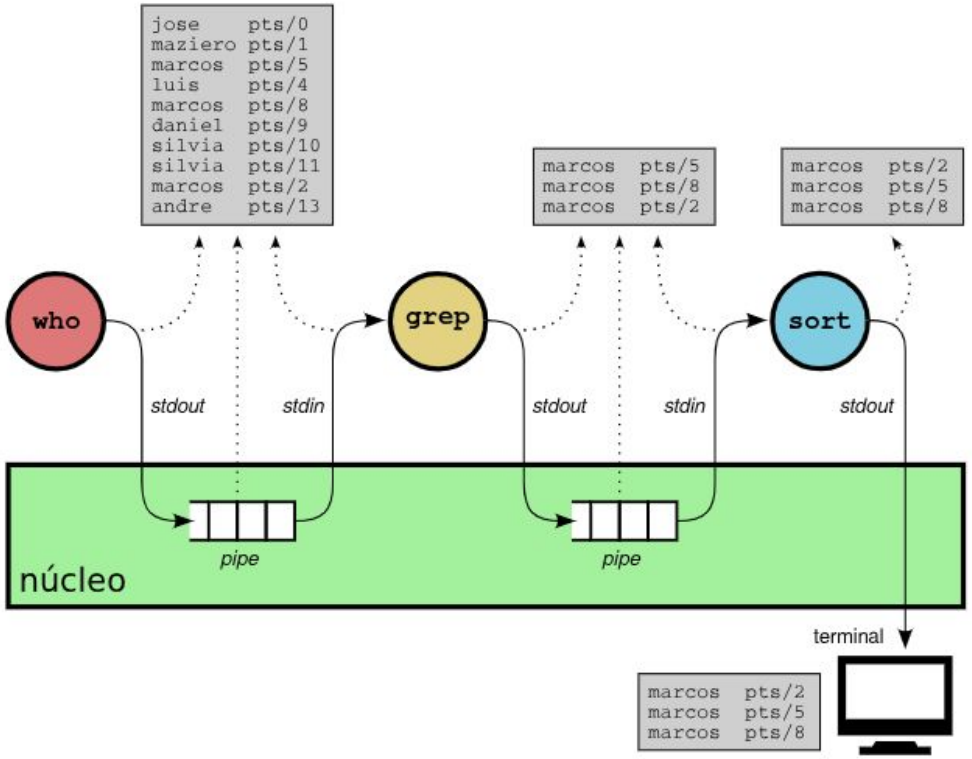
- Pipe
  - É um “tubo” de comunicação, um canal.
  - A comunicação é feita de forma *unilateral*
    - Processo A envia e Processo B recebe, os dados são considerados como um argumento.
  - O tamanho das mensagens é limitado, normalmente pequeno;
  - Muito utilizado em prompt de comando.

Comunicação [6/8]

- Pipe (exemplo)

```
#who|grep marcos|sort> login-marcos.tx
```

  - *who* verifica quais os usuários conectados ao computador
    - a saída deste comando é enviada par o próximo
  - *grep* filtra e gera uma saída com linhas contendo a string “marcos”
    - a saída também é enviada ao próximc através do pipe
  - *sort* ordena a lista recebida e insere no arquivo



Ver no terminal!

Comunicação - Pipe

Exemplo em C

C pipe.c > main(int, char * [])

```c
// Arquivo pipe.c: cria e usa uma área de memória compartilhada POSIX.
// Em Linux, compile usando: cc -Wall pipe.c -o pipes -lrt

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>


int main (int argc, char *argv[]){
    int fd[2]; //pos 0 de fd para ler um valor, pos 1 para escrever
    int value;
    if (pipe(fd) == -1){
        perror("PIPE!");
        exit(1);
    }
    int id = fork();
    if (id == -1){
        perror("FORK!");
        exit(1);
    }
    if (id ==0){ // Filho - consumidor
        close(fd[1]); //não vai escrever nada
        for (;;){
            read(fd[0],&value,sizeof(value));
            printf("Valor lido %d\n", value);
            sleep(1);
        }
        close(fd[0]);
    }
    else{ // Pai - produtor
        close(fd[0]);
        for(;;){
            value = random()%1000;
            //escreve o valor em uma ponta do tubo (pipe):
            write(fd[1], &value, sizeof(value));
            printf("Valor escrito: %d\n", value);
            sleep(1);
        }
        close(fd[1]);
    }
}
```

Comunicação

- Pipe nomeado (FIFO)
  - Um pipe só existe durante a execução da linha de comando ou processo que criou
  - FIFOs permanecem até serem destruídos
    - *Pipe* independente do processo que os criou

```
# cria um pipe nomeado, cujo nome é /tmp/pipe
$ mkfifo /tmp/pipe

#mostra o nome do pipe no diretório
$ ls -l /tmp/pipe

#envia dados para o pipe
$ date > /tmp/pipe

#em outro terminal, recebe os dados do pipe
$ cat < /tmp/pipe

#remove o pipe nomeado
$ rm /tmp/pipe
```

Ver no terminal!

Comunicação - Pipe nomeado

Exemplo em C

```c
int main (int argc, char *argv[]){
    int fd, value;

    char * myfifo = "/tmp/myfifo";
    mkfifo(myfifo, 0666); //leitura e escrita

    for(;;){
        if(PRODUTOR){
            fd = open(myfifo,O_WRONLY);
            if(fd == -1){
                perror("PRODUTOR NÃO ABRIU!");
                exit(1);
            }
            value = random() %1000;
            write(fd,&value,sizeof(value));
            close(fd);
            printf("Valor escrito: %d\n", value);
            sleep(1);
        }
        else{
            fd = open(myfifo,O_RDONLY);
            if(fd == -1){
                perror("CONSUMIDOR NÃO ABRIU!");
                exit(1);
            }
            read(fd,&value,sizeof(value));
            close(fd);
            printf("Valor lido: %d\n", value);
            sleep(1);
        }
    }
}
```

UFSC

Comunicação [7/8]

- Chamada com passagem de argumentos
  - A interação ocorre quando um processo “chama” outro processo informando parâmetros para sua execução
    - Inicia a execução de outro processo
  - Ex.: Dois programas em C, A e B, **B** necessita que sejam informadas 3 *strings* (valores para variáveis), **A** durante sua execução chama o programa **B** informando os valores das 3 variáveis.
  - Outro Exemplo?
    - Impressão de PDF
      - Um ERP chama o programa PDFCreator (ou qualquer outro) informando os parâmetros para criação do PDF
  - Semelhante a passagem de argumentos para uma função interna do programa

Comunicação [8/8]

- Troca de mensagens
  - Os processos “abrem” uma porta para comunicação
    - Função, método em execução que espera uma *string*
    - As ações sequenciais do programa dependem do recebimento ou envio da mensagem monothread
    - Outras ações podem ser executadas enquanto a mensagem for enviada ou espera para ser recebida Multithread
      - O resultado da mensagem é normalmente armazenado em variáveis globais
      - Difícil sincronismo
  - Utiliza os comandos *send* e *receive*
    - *send (destino, &mensagem)*
    - *receive (origem, &mensagem)*
- Qual a diferença para o *pipe* de comunicação?
  - A comunicação é bilateral
  - O tamanho das mensagens é maior.

Desafios

- Existem alguns desafios quanto a comunicação entre processos
  1. Como a comunicação pode ser feita?

  2. **Como garantir que dois processos não invadam um ao outro, quando estiverem acessando uma região crítica.**
     - Ex.: Processo A altera o valor de uma variável utilizada por B.

  3. **Como garantir a execução dos processos em uma sequência adequada.**
     - Ex.: Processo A é responsável por imprimir arquivos e Processo B coloca os arquivos para imprimir, A não pode executar antes de B

Condição de corrida ou disputa [1/5]

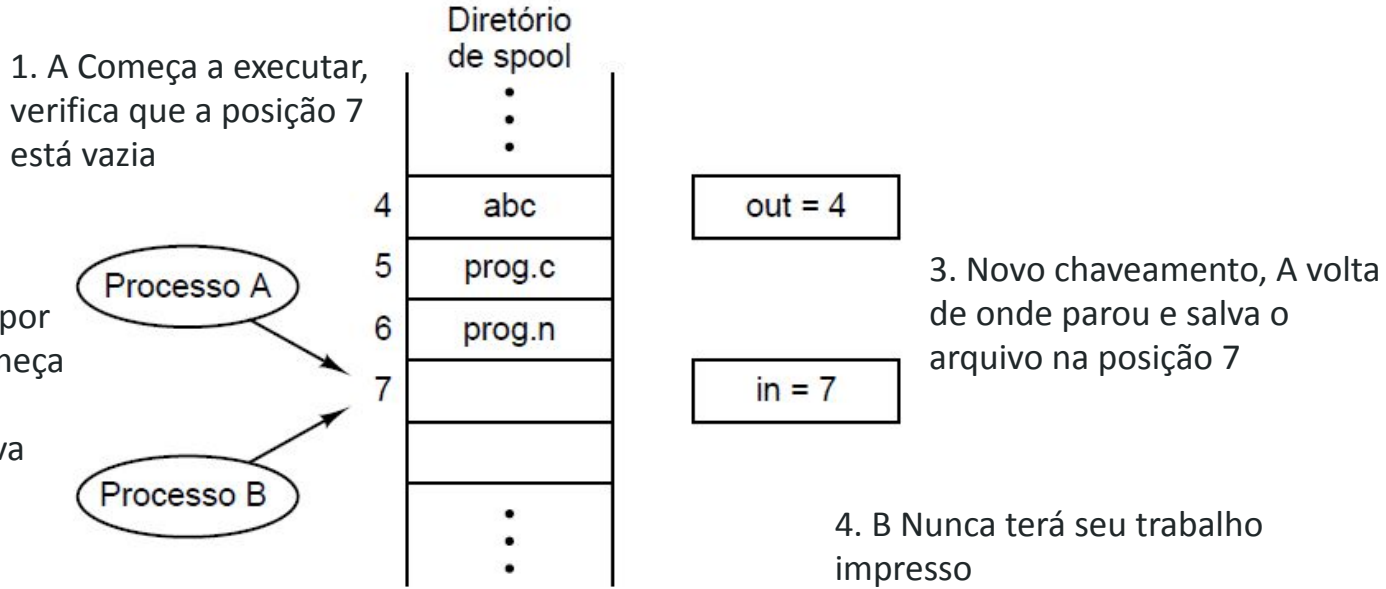
- Quando dois ou mais processos manipulam simultaneamente dados compartilhados, o resultado das operações depende da ordem em que eles executam.
- Exemplo
  - Funcionamento do *Spool* de impressão
  - Processos colocam os seus trabalhos em uma fila
  - O servidor de impressão (outro processo) retira os trabalhos e envia para a impressora
  - A fila de impressão é uma área de armazenamento compartilhada
  - Utiliza um diretório de Spool para organizar a fila
    - Os espaços de memória neste diretório são numerados e devem ser atualizados *pelos* processos a cada inclusão

Condição de corrida ou disputa [2/5]

- *Spool* de impressão
  - Controle dos espaços é feito por duas variáveis no processo Spool
    - **In** □ próximo espaço vazio onde podem entrar mais arquivos
    - **Out** □ próximo arquivo a ser impresso
  - Imagine uma situação onde dois processos A e B decidam imprimir os seus arquivos no mesmo instante de tempo

Condição de corrida ou disputa [3/5]

- *Spool* de impressão



Condição de corrida ou disputa [4/5]

- Outro exemplo, uma analogia em forma de algoritmo
- Administração do estoque de leite
  1. Vê se tem leite na geladeira
  2. Se acabou
     1. Vai para o mercado
     2. Chega no mercado
     3. Compra leite
     4. Vai para casa
     5. Põe leite na geladeira

Condição de corrida ou disputa [5/5]

- **Alice**
  1. Vê que acabou o leite
     1. Vai para o mercado
     2. Chega no mercado
     3. Compra leite
     4. Volta pra casa
     5. Põe leite na geladeira

- **Bob**
  1. Vê que acabou o leite
     1. Vai para o mercado
     2. Chega no mercado
     3. Compra leite
     4. Volta pra casa
     5. Põe leite na geladeira

Leite vai azedar

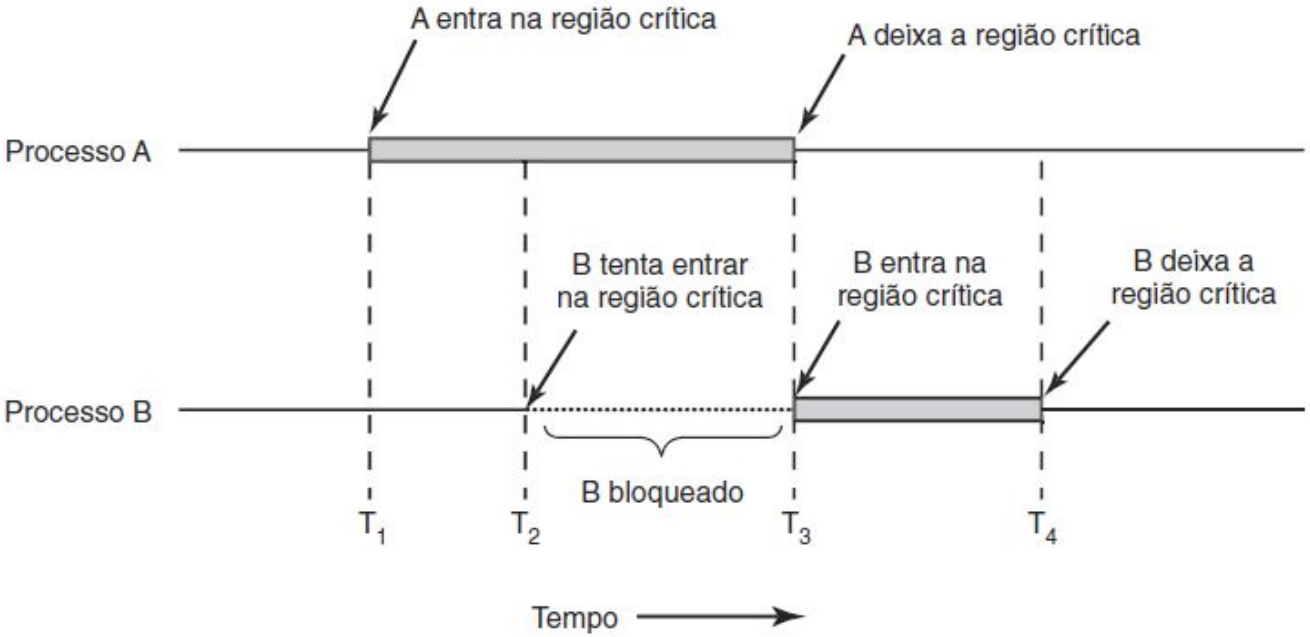
O estoque de leite é um estado (dado) compartilhado

Região Crítica

- Para evitar as situações que ocorrem as condições de disputa são necessários alguns mecanismos
  - Sincronização e exclusão
- A solução é empregar a **exclusão mútua**
  - Garantia de que no momento em que um processo acessa a área de memória compartilhada nenhum outro processo o fará
- O trecho de código onde um processo lê e escreve em uma região de memória compartilhada é a **região crítica**

Conceitos

- Região Crítica



Exemplo - transação

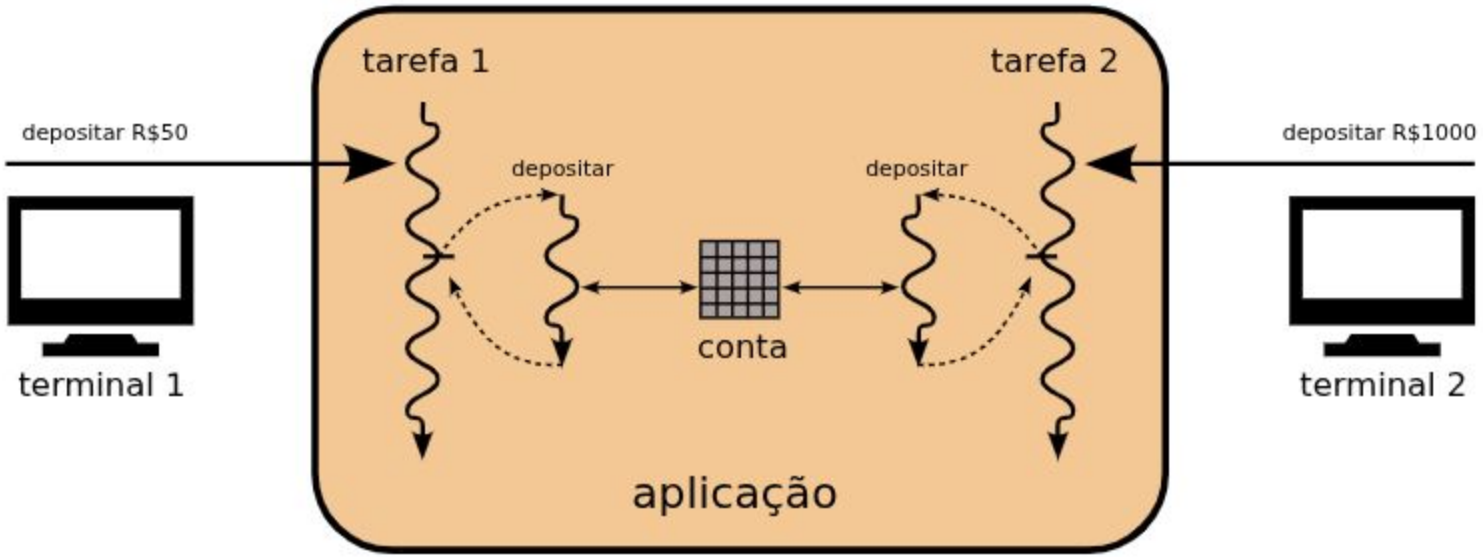
```c
void depositar (long * saldo, long valor)
{
   (*saldo) += valor ;
}
```

```
0000000000000000 <depositar>:
       ; inicializa a função
       push  %rbp
       mov   %rsp,%rbp
       mov   %rdi,-0x8(%rbp)
       mov   %esi,-0xc(%rbp)

       ; carrega o conteúdo da memória apontada por "saldo" em EDX
       mov   -0x8(%rbp),%rax       ; saldo → rax (endereço do saldo)
       mov   (%rax),%edx           ; mem[rax] → edx

       ; carrega o conteúdo de "valor" no registrador EAX
       mov   -0xc(%rbp),%eax       ; valor → eax

       ; soma EAX ao valor em EDX
       add   %eax,%edx             ; eax + edx → edx

       ; escreve o resultado em EDX na memória apontada por "saldo"
       mov   -0x8(%rbp),%rax       ; saldo → rax
       mov   %edx,(%rax)           ; edx → mem[rax]

       ; finaliza a função
       nop
       pop   %rbp
       retq
```

UFSC

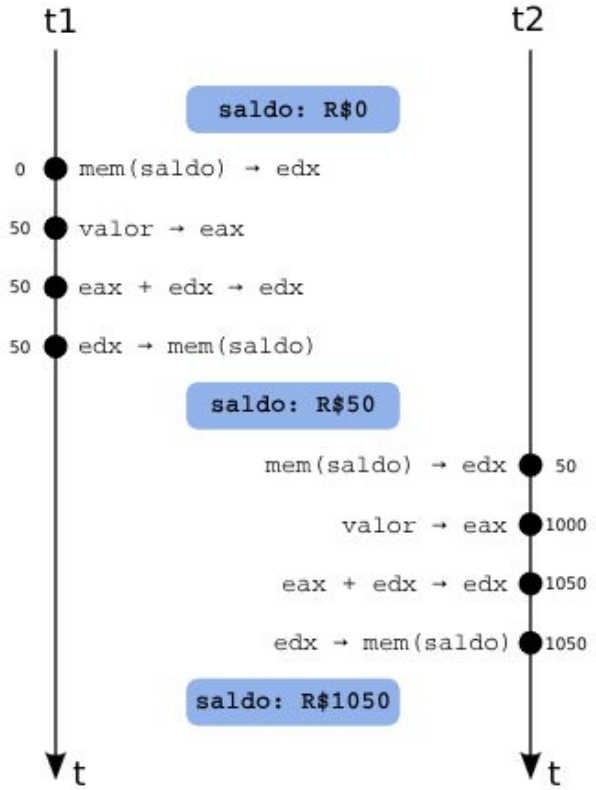
Exemplo - transação

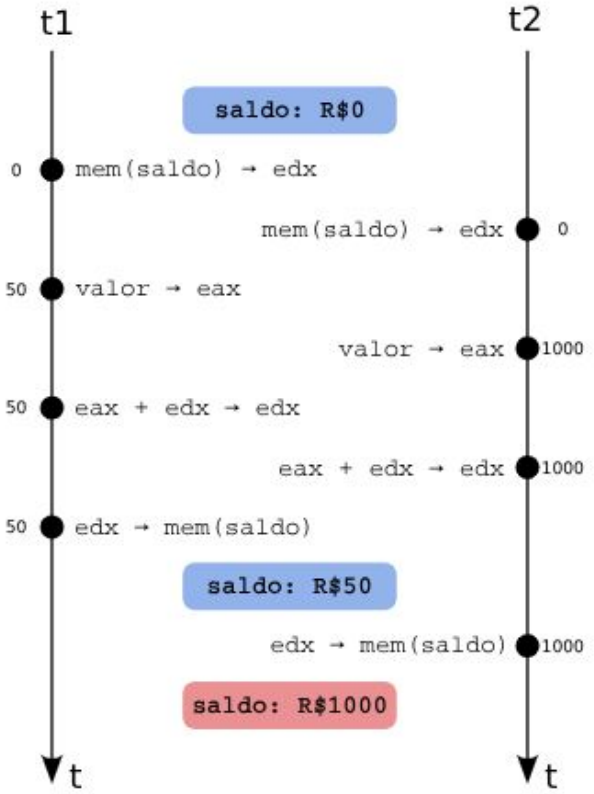
```
1  void depositar (long * saldo, long valor)
2  {
3     (*saldo) += valor ;
4  }
```



Exemplo - transação

```
1 void depositar (long * saldo, long valor)
2 {
3    (*saldo) += valor ;
4 }
```





Como garantir a exclusão mútua

- Na **analogia**: Como impedir que os dois alterem o estoque de leite?
- Podemos dividir as soluções existentes em duas formas:
  - Exclusão mútua **com** espera ocupada
    - O processo fica em *loop* até conseguir acessar a região crítica
  - Exclusão mútua **sem** espera ocupa
    - O processo fica bloqueado, ou aguarda sua vez de acessar a região crítica

Exemplo - transação

```
1  void depositar (long conta, long *saldo, long valor)
2  {
3     enter (conta) ;          // entra na seção crítica "conta"
4     (*saldo) += valor ;      // usa as variáveis compartilhadas
5     leave (conta) ;          // sai da seção crítica
6  }
```

Exclusão mútua com espera ocupada [1/2]

- Desabilitar interrupções
  - Quando o processo entra na região crítica ele desabilita as interrupções de CPU, assim, ele executa até o fim do seu trecho de código, não será retirado
    - Se esquecer de habilitar as interrupções a máquina trava
- Variáveis de impedimento (lock)
  - Quando um processo vai entrar na região crítica ele testa se alguém já está utilizando, se sim, fica em um loop infinito até que a região seja liberada
- Alternância obrigatória
  - Cada processo tem sua vez de entrar na região crítica, utiliza uma variável *turn* que indica de quem é a vez (id) de entrar na região crítica
- Instrução TSL (**T**est and **S**et **L**ock)
  - Instrução atômica em hardware, similar às variáveis de impedimento, o hardware testa a variável antes de entrar na seção crítica, se está ocupada fica testando até estar livre.

Exclusão mútua com espera ocupada [2/2]

- Exemplo – Variável de impedimento

```
1: while (lock == 1)
2:     ;     /* loop vazio */
3: lock = 1;
4: /* seção crítica */
5: lock = 0;
6: /* seção não crítica */
```

- Problema
  - O estado da variável *lock* pode ser lido ao mesmo tempo, pelos dois processos (ou threads)
  - Pouco eficiente Consumo de recurso

Exclusão mútua sem espera ocupada [1/3]

- Dormir e Acordar
  - Primitivas simples para controlar as ações dos processos
  - Dormir(faz com que o processo aguarde para ser acordado)
  - Wakeup(acorda um processo)
- Semáforos
  - Variação do dormir e acordar
  - Uma variável compartilhada (semáforo) é utilizada para controlar o acesso a região crítica (semelhante a variável trava)
  - As ações UP() e Down() são **atômicas** e **implementadas pelo SO**.

Exclusão mútua sem espera ocupada [2/3]

- Monitores
  - A sincronização é feita em nível de linguagem de programação
  - Elementos de linguagem que controlam o acesso a região crítica
  - Dois processos não podem acessar o monitor em dado instante
  - Primitivas *wait()* e *signal(),* quem espera aguarda um sinal de liberação
- Troca de mensagens
  - Utiliza as primitivas *send()* e *receive()*
  - Um processo que dispara *receive()* vai aguardar até que alguém lhe envie uma mensagem para dar continuidade no código
  - Antes de entrar em uma região crítica envia uma mensagem teste e aguarda resposta para dar continuidade

Exclusão mútua sem espera ocupada [3/3]

- Exemplo – Semáforos

```
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 }
```

Sincronização [1/2]

- Técnicas de garantia de exclusão mútua podem ser usadas para determinar a ordem de execução;
- Exemplo:
  - O que será impresso na execução simultânea das funções abaixo:

```
void consoante(){
     while(TRUE){
          printf(“C”);
          printf(“R”);
          printf(“T”);
     }
```

```
void vogal(){
     while(TRUE){
          printf(“A”);
          printf(“E”);
          printf(“O”);
     }
}
```

UFSC

Sincronização [2/2]

- Exemplo:
  - E agora?

```
semaphore vog=1, cons=0;
```

```
void consoante(){
     while(TRUE){
          down(&cons);
          printf(“C”);
          up(&vog);
          down(&cons);
          printf(“R”);
          printf(“T”);
          up(&vog);
     }
}
```

```
void vogal(){
     while(TRUE){
          down(&vog);
          printf(“A”);
          up(&cons);
          down(&vog);
          printf(“E”);
          up(&cons);
          down(&vog);
          printf(“O”);
     }
}
```

Próxima aula

- Implementação das técnicas de garantia de exclusão mútua.
- Cap. 2 Tanembaum

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.

UFSC