

CURSO DE ENGENHARIA DE SOFTWARE

Disciplina: Sistemas Operacionais

Comunicação e Sincronização entre Processos

Prof. M.e Alexandre Tannus

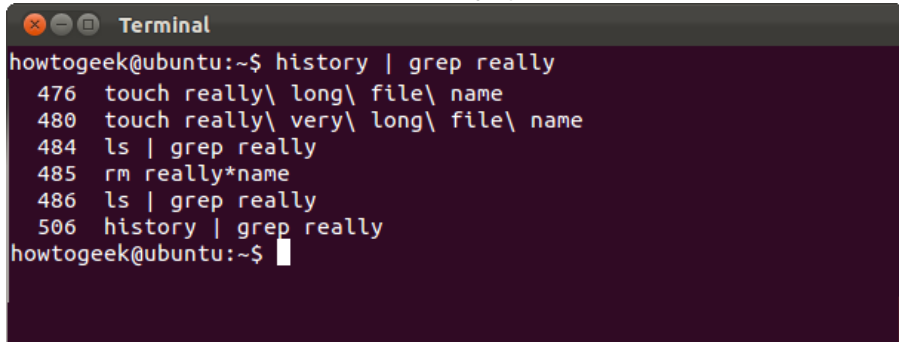
Introdução

Condições de corrida

Soluções de Exclusão Mútua

- ▶ Como implementar concorrência dentro de uma aplicação?
- ▶ E se duas ou mais *threads*/processos quiserem o mesmo recurso?
- ▶ Quais regras se aplicam para considerar um sistema concorrente como bem projetado?

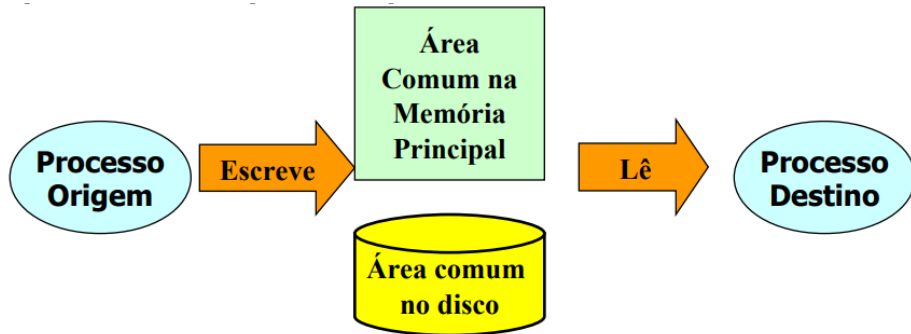
- ▶ Diferentes processos necessitam se comunicar e podem compartilhar diretamente um espaço de endereçamento

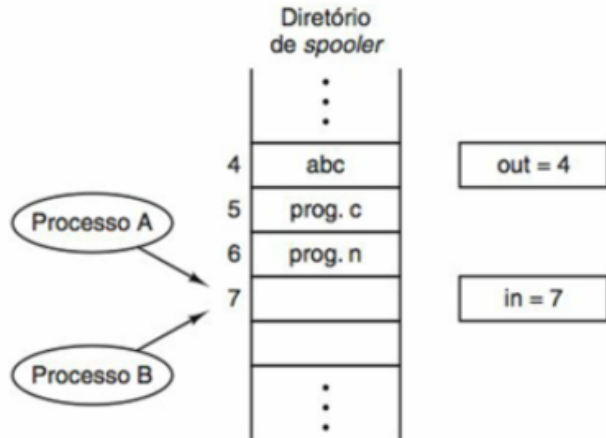


```
Terminal
howtogeek@ubuntu:~$ history | grep really
476 touch really\ long\ file\ name
480 touch really\ very\ long\ file\ name
484 ls | grep really
485 rm really*name
486 ls | grep really
506 history | grep really
howtogeek@ubuntu:~$
```

- ▶ Passagem de informações
- ▶ Interferência entre processos em situações críticas
- ▶ Sequenciamento adequado entre as dependências

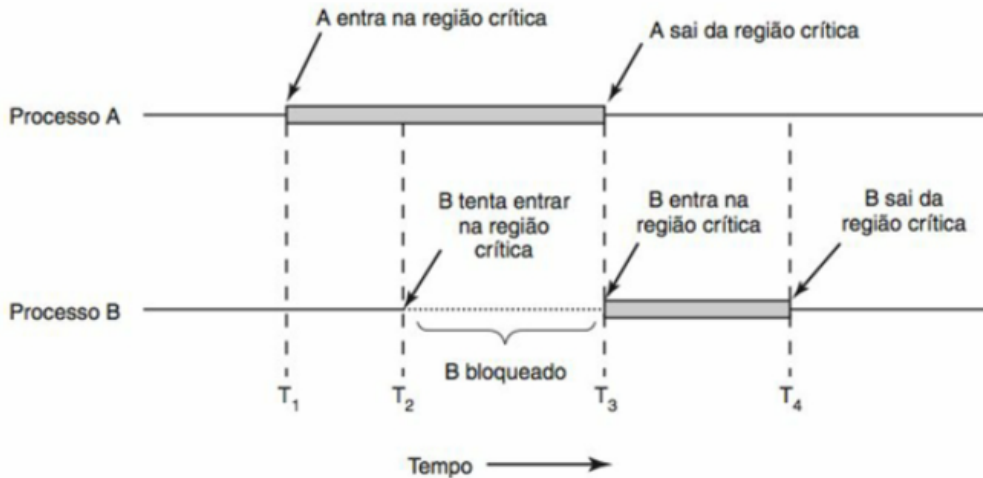
- Situações onde dois ou mais processos estão lendo ou escrevendo algum dado compartilhado e o resultado depende de quem processa no momento propício.





- ▶ Proibição de ocorrência de múltiplos processos façam uso simultâneo de dados compartilhados (seção crítica).

EXCLUSÃO MÚTUA



- ▶ Dois processos não podem estar simultaneamente dentro de uma região crítica.
- ▶ Nenhuma suposição pode ser feita sobre as velocidades ou sobre o número de CPUs.
- ▶ Nenhum processo executando fora de sua região crítica pode bloquear outros processos.
- ▶ Nenhum processo deve ter que esperar eternamente para entrar em sua região crítica. (*starvation*).

- ▶ Espera ocupada
- ▶ Primitivas *Sleep/Wakeup*
- ▶ Semáforos
- ▶ Monitores
- ▶ Passagem de mensagem

- ▶ Constante checagem por algum valor
- ▶ Soluções
 - ▶ Desabilitação de interrupções
 - ▶ Variáveis de travamento (*Lock*)
 - ▶ Estrita Alternância
 - ▶ Solução de Peterson

- ▶ Solução de *hardware*
- ▶ Desabilitação das interrupções quando o processo entra na região crítica e reabilitação quando sai.
- ▶ Vantagens
 - ▶ Simplicidade de implementação
- ▶ Desvantagens
 - ▶ Processo pode esquecer de reabilitar interrupções
 - ▶ Problemas em sistemas *multicore*

- ▶ Solução de *software*
- ▶ Criação de uma variável *lock*
 - ▶ Valor 0: Nenhum processo na região crítica
 - ▶ Valor 1: Existe processo na região crítica

```
while(true) {  
    while(lock!=0); //loop  
    lock=1;  
    critical_region();  
    lock=0;  
    non-critical_region();  
}
```

Processo A

```
while(true) {  
    while(lock!=0); //loop  
    lock=1;  
    critical_region();  
    lock=0;  
    non-critical_region();  
}
```

Processo B

32

- Solução de *software*

- Criação de uma variável *turn*

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

```
while (T  
    while  
    critica  
    turn =  
    nonc
```


- ▶ Solução mista de *hardware/software*
- ▶ Variável compartilhada para bloqueio da entrada de um processo na região crítica quando ela estiver ocupada

- ▶ Verificação constante de possibilidade de entrada na região crítica
- ▶ Desperdício de tempo da CPU
- ▶ Pode provocar espera infinita (*deadlock*) em determinados sistemas

- ▶ Bloqueio e desbloqueio de processos
 - ▶ *Sleep* - Bloqueia o processo (suspensão da execução)
 - ▶ *Wakeup* - Desbloqueio do processo
- ▶ Exemplo de uso
 - ▶ Problema do produtor-consumidor

Problema do produtor-consumidor

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

void producer(void)
{
    int item;

    while (TRUE) {                            /* repete para sempre */
        item = produce_item();               /* gera o próximo item */
        if (count == N) sleep();             /* se o buffer estiver cheio, bloqueia */
        insert_item(item);                   /* coloca item no buffer */
        count = count + 1;                   /* incrementa a contagem de itens no buffer */
        if (count == 1) wakeup(consumer);   /* o buffer estava vazio? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                            /* repete para sempre */
        if (count == 0) sleep();             /* se o buffer estiver vazio, bloqueia */
        item = remove_item();                /* retira item do buffer */
        count = count - 1;                   /* decrementa a contagem de itens no buffer */
        if (count == N - 1) wakeup(producer); /* o buffer estava cheio? */
        consume_item(item);                  /* imprime o item */
    }
}
```

- ▶ SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G.. **Fundamentos de sistemas operacionais: princípios básicos.** Rio de Janeiro: LTC – Livros Técnicos e Científicos, 2013.
- ▶ TANENBAUM, A.S., WOODHULL, A.S. **Sistemas Operacionais.** Porto Alegre: Grupo A, 2008.



UniEVANGÉLICA

CENTRO UNIVERSITÁRIO