Professor
Edson Pedro FERLIN

Sistemas Operacionais

Comunicação entre Processos

Prof. Edson Pedro Ferlin

Agradecimento ao Prof. Osmar Betazzi Dordal

Professor
Edson Pedro FERLIN

Sistemas Operacionais

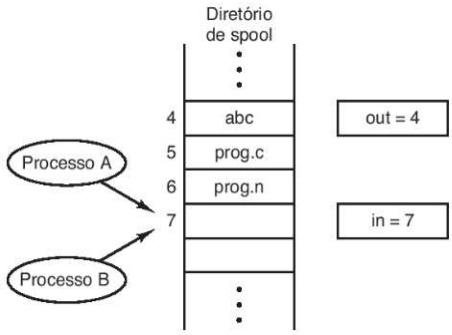
- Objetivos
  - Entender os conceitos de comunicação entre processos e exclusão mútua
- Conteúdos
  - Comunicação entre Processos
  - Exclusão Mútua

Comunicação entre Processos

- Frequentemente processos precisam se comunicar.
- A comunicação entre processos é mais eficiente se for estruturada e não utilizar interrupções.
- Os processos se comunicam por meio de alguma área de armazenamento comum.
- Essa área pode estar na memória principal ou pode ser um arquivo compartilhado.

Condição de Corrida - Problemas de Acessos Simultâneos em Regiões Criticas

- Próximo arquivo a ser impresso será "abc" – (*out = 4).*
- Dois processos, **A** e **B** querem imprimir um arquivo e o próximo local vazio é 7.
- Se o processo **A** for colocar seu arquivo ele colocará em 7, porém, se ele for retirado, o processo **B** fará a mesma coisa.
- Se o processo **B** colocar seu arquivo em 7 e atualizar *in=8* e sair para a entrada novamente do processo **A**. Então, o processo **A** que tinha como *in=7* vai sobrescrever o arquivo do processo **B**.
- Assim, o processo **B** nunca terá seu arquivo impresso.

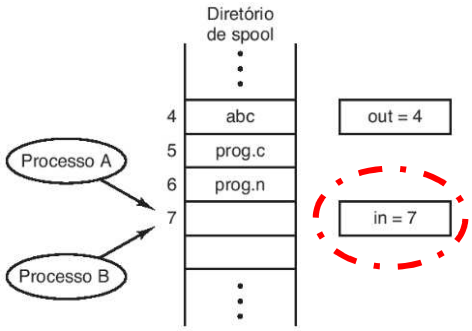


Exemplo : *overbooking* de companhias aéreas

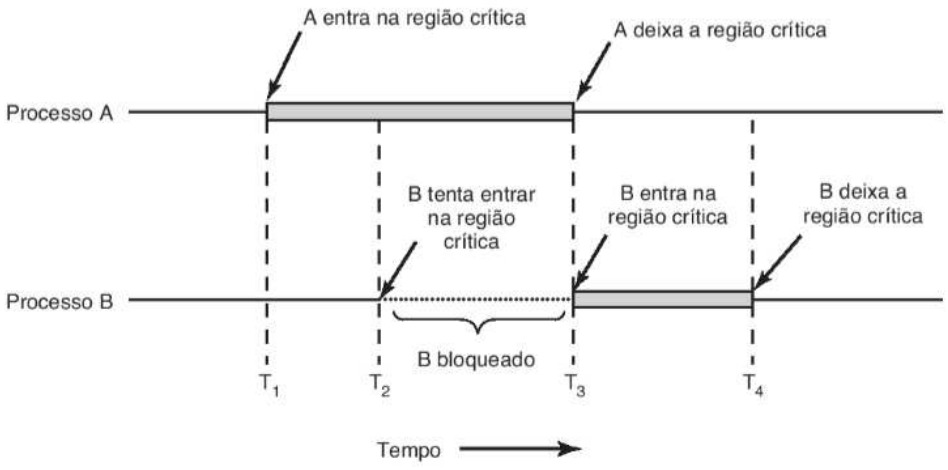
Exclusão Mútua e *Race Condition*

- Impedir que mais de um processo leia e escreva em uma **variável compartilhada** (área de memória) ao mesmo tempo.

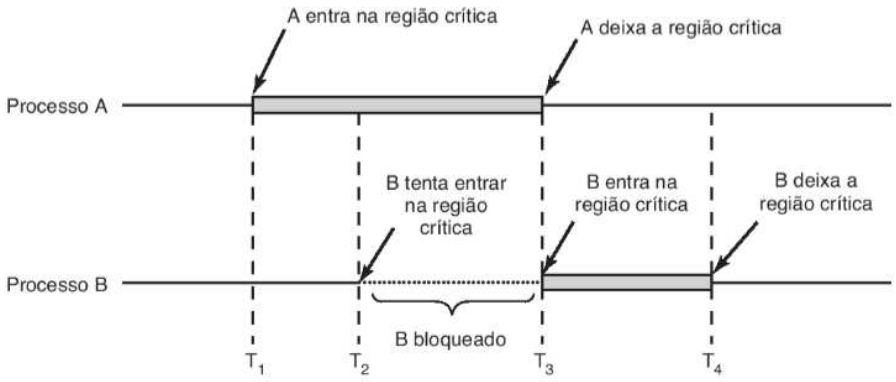
- Essas são chamadas de **regiões críticas**
  - Região crítica do *Spool* de Impressão.
  - Quando o processo entra na **RC**, ele só deveria mudar de contexto quando saísse da **RC**.
  - Processo deve ser atômico
    - Um único processo pode executá-los um de cada vez.



Exclusão Mútua e *Race Condition*



Exclusão Mútua e *Race Condition*



Caso mais crítico com *Threads*, pois compartilham os mesmos recursos dentro de um processo.

Regras para Programação Concorrente

- Dois processos nunca podem estar simultaneamente dentro de suas regiões críticas.
- Não se pode fazer suposições em relação à velocidade e ao número de CPUs.
- Um processo fora da região crítica não deve causar bloqueio a outro processo.
- Um processo não pode esperar eternamente para entrar em sua região crítica.

Soluções para Exclusão Mútua

- Espera Ocupada (*Busy Waiting*)
- Primitivas *Sleep/Wakeup*
- Semáforos
- Monitores
- Troca de Mensagem

Espera Ocupada – *Busy Waiting*

- Consiste na constante checagem por algum valor
  - Se existe um processo na região crítica um outro processo deve consultar se o primeiro saiu ou não.
  - Desperdiça os ciclos de processamento ☹
- Para isso algumas soluções são:
  - Desativar interrupções;
  - Variáveis de travamento – *Lock;*
  - Estrita alternância – *Strict Alternation;*
  - Solução de Peterson e Instrução – TSL.

Primitivas *Sleep/Wakeup* – Dormir e Acordar

- Para solucionar esse problema de espera, **um par de primitivas – *Sleep* e *Wakeup*** – é utilizado.
- São chamadas ao sistema para **bloqueio** e **desbloqueio** de processos.
- *Sleep* bloqueia o processo que a chamou.
  - **Suspende sua execução** até que outro processo o "acorde" (*Wakeup*).
- A primitiva *Wakeup* "**acorda**" **um determinado processo**, evitando espera ocupada.

Problema do Consumidor e Produtor

- **Dois processos compartilham um *buffer* de tamanho fixo**.
- O processo **produtor coloca** dados no *buffer.*
- O processo **consumidor retira** dados do *buffer.*
- **Problema:**
  - O produtor deseja colocar dados e o *buffer* está cheio;
  - Da mesma forma...
  - O consumidor deseja retirar dados quando o *buffer* está vazio.
- Devem ser feitas checagens no *buffer* (variável *count*) fazendo produtor ou consumidor dormir ou acordar.

Semáforos

- **Variável** utilizada para controlar o acesso a recursos compartilhados
- **Sincronizar o uso de recursos em grande quantidade.**
- Nasceu como proposta para contar o número de ***wakeups* armazenados** para uso futuro.
- **Conta o número de recursos ainda disponíveis no sistema**
  - **Semáforo = 0** : não há recurso livre
    - Nenhum *wakeup* está armazenado.
  - **Semáforo > 0** : recurso livre
    - Um ou mais wakeups estão pendentes.
  - *Dijkstra* propôs 2 operações (generalizações de *sleep* e *wakeup*).

Monitores

- Primitiva de alto nível para sincronizar processos e de fácil uso.
- Conjunto de procedimentos, variáveis e estrutura de dados **agrupados em um único modulo ou pacote.**
- **Somente um processo pode estar ativo dentro do monitor em um mesmo instante.**
- Outros processos ficam bloqueados até que possam estar ativos no monitor.
- Todos os recursos compartilhados entre processos devem estar implementados **dentro do monitor.**

```
monitor example
    int i;
    condition c;

    procedure A();
    .
    .
    end;
    procedure B();
    .
    .
    end;
end monitor;
```

Limitações de Monitores e Semáforos

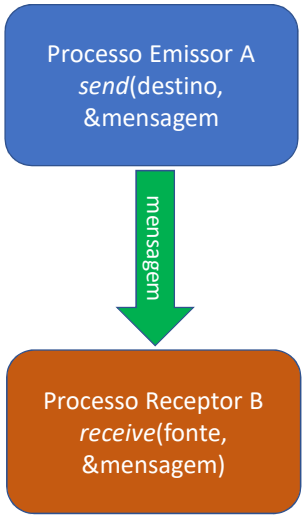
- **Não são soluções adequadas para sistemas distribuídos.**
- Não dão suporte a sincronização entre processos em máquinas diferentes.
  - Somente máquinas locais.
- Monitores dependem de uma linguagem de programação
  - Poucas linguagens suportam monitores.
- É preciso utilizar uma solução a mais (Mensagens).

Troca de Mensagens

- Os processos enviam e recebem mensagens, dessa forma não vão ler e escrever em variáveis compartilhadas.
- Garantida pela restrição de que uma mensagem só poderá ser recebida depois de ter sido enviada.
- A transmissão de dados é realizada após ter ocorrido a sincronização.
- *Send*
- *Receive*

Implementadas como chamadas ao sistema (*kernel*)

  - Se não houver mensagem disponível, o receptor pode:
    - **Bloquear**, até que haja alguma mensagem (*Blocked*);
    - Retornar com **mensagem de erro** (*Unblocked*).



Contato

eferlin@live.com

(BLOG) professorferlin.blogspot.com

(SITE) professorferlin.webnode.com.br

(YOUTUBE) ProfEdsonPedroFerlin