UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
CURSO: CIÊNCIA DA COMPUTAÇÃO

COMUNICAÇÃO ENTRE PROCESSOS (IPC)

SISTEMAS OPERACIONAIS

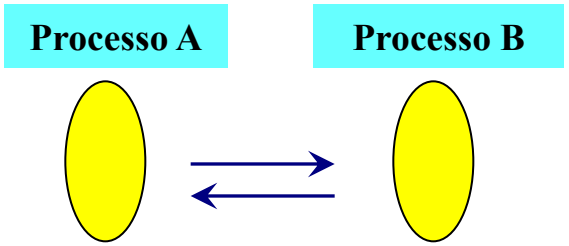
Prof. Rivalino Matias Jr

AGENDA

- ✓ INTRODUÇÃO À IPC
- ✓ COMPARTILHAMENTO DE RECURSOS
- ✓ SINCRONIZAÇÃO
- ✓ REGIÃO CRÍTICA
- ✓ EXCLUSÃO MÚTUA
- ✓ PROBLEMAS E SOLUÇÕES
  - ✓ STARVATION
  - ✓ TEST-AND-SET
  - ✓ SEMÁFORO
  - ✓ MONITORES
  - ✓ DEADLOCKS

INTRODUÇÃO

- Inúmeras aplicações são programadas em forma de vários processos cooperantes.



- Estes processos cooperam entre si compartilhando recursos e informações.
- O kernel do SO deve prover os mecanismos necessários para estas implementações.
- Estes mecanismos são comumente chamados de IPC.

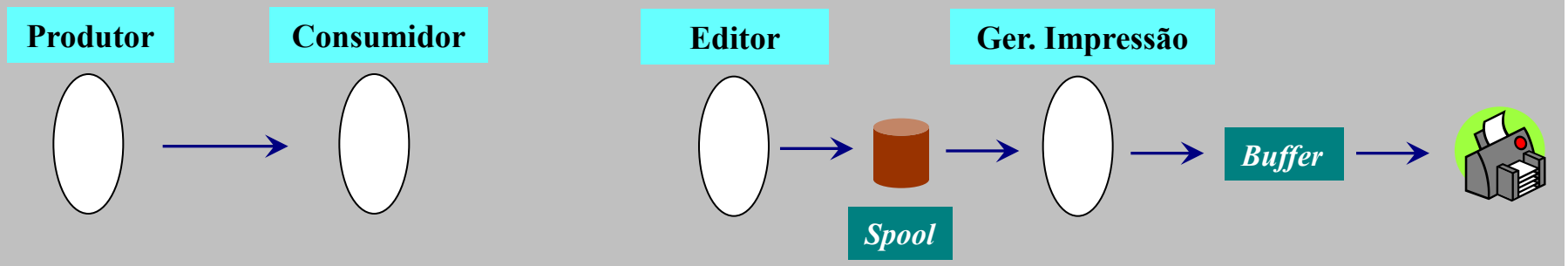
PROPÓSITOS PRINCIPAIS:

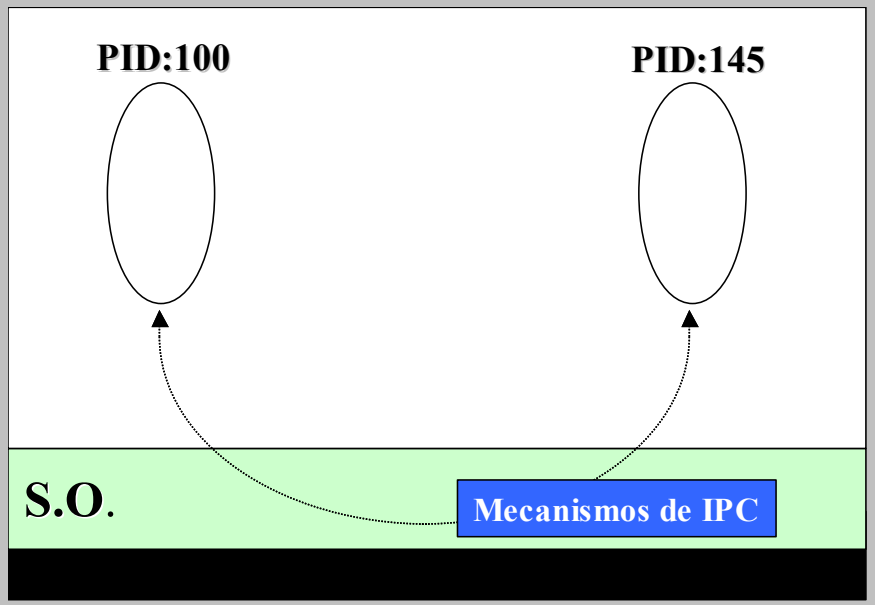
- Transferência de Dados
  - Ex: FTP, ICQ, etc.
- Compartilhamento de Dados
  - Ex: Variáveis de controle entre processos cooperantes.
- Notificações de Eventos
  - Ex: O Processo filho notifica seu processo pai de sua conclusão.
- Compartilhamento de Recursos
  - Ex: Gerenciamento da fila de impressão.
- Controle de Processos
  - Ex: Depuração de um processo em tempo de execução (*debugging*).

APLICAÇÕES

- Sistemas Servidores:
  - WEB;
  - eMAIL;
  - Impressão;
  - Arquivo, etc.
- Sistemas Comerciais:
  - Controle de conta corrente;
  - Controle de estoque;
  - Controle de passagens (avião, ônibus, etc.)
  - etc...

Modelo Produtor-Consumidor





Exemplo de Utilização Prod.-Cons.

- Encadeamento de comandos:
  - MS-DOS:
    - C:\type texto.txt | more
  - UNIX:
    - % cat texto.txt | more
    - % ls –l | grep filename | wc -l

" Pipe é um exemplo de mecanismo, fornecido pelo sistema operacional, para a realização de IPC."

- Outros mecanismos:
  - *Shared memory*
  - *Signals*
  - *Semaphores*
  - *Message Passing / Message Queues*
  - *Ports*

MECANISMOS DE IPC

- Como um processo comunica com outro ?
  - **Signal**: Um processo notifica outro através de uma interrupção de software (Ex. *kill* do unix).
  - **Semaphore**: Processos bloqueados, aguardando sua vez, recebem uma notificação (*signal*).
  - **Pipes**: Fluxo de comunicação unidirecional.
  - **Message passing**: Processos enviam e recebem mensagens (send/receive), podendo a comunicação ser individual (dois processos) ou em grupo.

Exemplos de Aplicação:

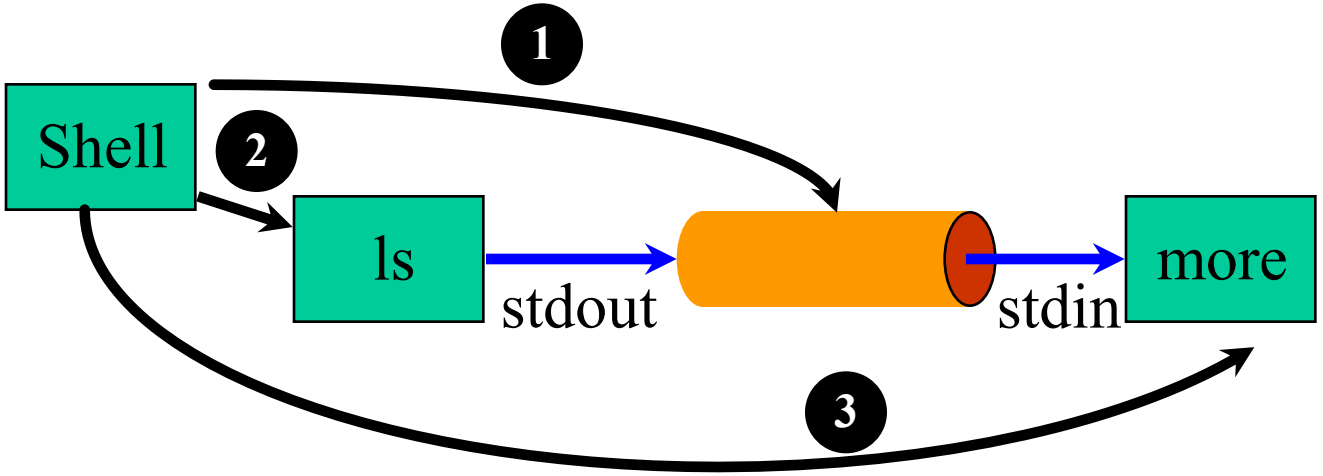
- **SIGNAL**
  - Similar a uma interrupção de hardware;
  - A comunicação se dá de forma assíncrona. O processo que recebe o sinal para sua execução, trata o sinal e continua sua execução.
  - Alarmes (ex. unix *alarm()*) é uma forma de programar a recepção de sinais enviados pelo SO.
  - É comum o SO avisar a aplicação de eventos do sistema, tais como: (disk quota, CPU time, divide by zero, etc.) através de sinais.

Exemplos de Aplicação:

- **SEMAPHORE**
  - Utilizado para sincronização de processos;
  - A interface do semáforo é visível por todos os processos comunicantes;
  - Garante atomicidade das operações;
  - Utilizado para implementar a exclusão mútua de processos;

Exemplos de Aplicação:

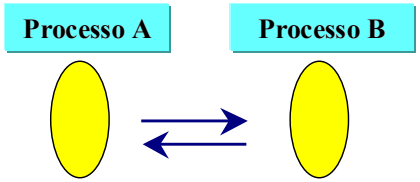
- PIPE
  - Um processo escreve e o segundo processo efetua a leitura (ex. % *ls* | *more*)

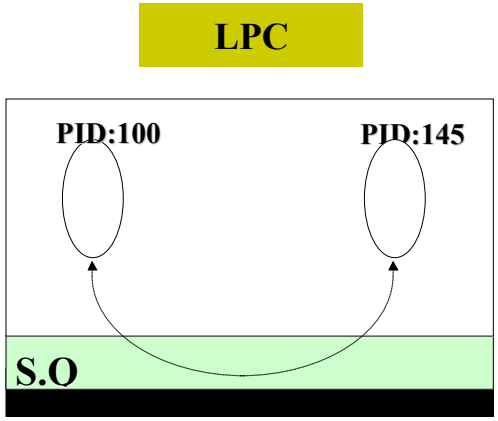


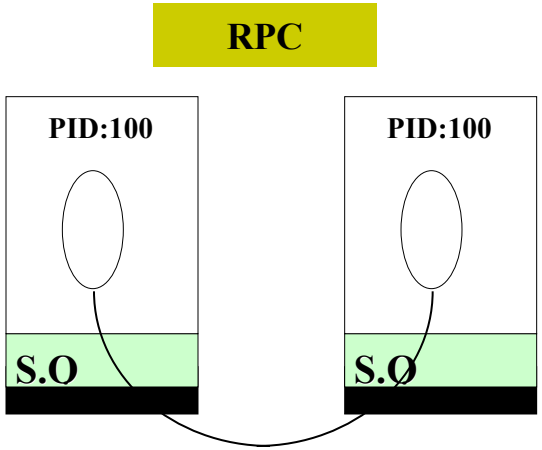
- shell:
  1. Cria um pipe
  2. Cria um processo para o comando (ls), configura a saída (stdout) de ls para escrever no pipe.
  3. Cria um processo para o comando (more), configura a entrada (stdin) de more para ler do pipe.

Exemplos de Aplicação:

- MESSAGING PASSING
  - Processos enviam e recebem mensagens através de primitivas de comunicação (send/receive);
  - A comunicação pode ser síncrona ou assíncrona;
  - A comunicação pode ocorrer entre dois processos ou em grupo;
  - A comunicação pode ser realizada no mesmo sistema (LPC- *local process communication*) ou entre sistemas remotos (RPC – *remote process communication*).







COMPARTILHAMENTO DE RECURSOS

- Ex: Sistema Bancário – controle de conta

.....

READ (Arq_contas, Reg_cliente);

READ(valor_dep_ret);

Reg_cliente.Saldo = Reg_cliente.Saldo + valor_dep_ret;

WRITE(Arq_contas, Reg_cliente); ....

Caixa	Comando	Saldo (Arq)	Valor(Dep/Ret)	Saldo(Memória)
1	READ	1.000	“ “	1.000
1	READ	1.000	-200	1.000
1	=	1.000	-200	800
2	READ	1.000	“ “	1.000
2	READ	1.000	+300	1.000
2	=	1.000	+300	1.300
1	WRITE	800	-200	800
2	WRITE	1.300	+300	1.300

COMPARTILHAMENTO DE RECURSOS

- Solução: *Exclusão mútua*
  - evitar com que mais de um processo execute sua região crítica.

- **Região crítica é a seção de código dos processos que fazem acesso ao(s) recurso(s) compartilhado(s).**

```
...
READ(valor_dep_ret);
```

Reg. Crítica

```
READ(Arq_contas, Reg_cliente);
Reg_cliente.Saldo = Reg_cliente.Saldo + valor_dep_ret;
WRITE(Arq_contas, Reg_cliente);
```

```
.....
```

COMPARTILHAMENTO DE RECURSOS

- Uma boa solução de exclusão mútua deve:
  - Evitar que processos acessem ao mesmo tempo suas RC;
  - Processos fora de sua RC não podem bloquear outros processos;
  - Processos devem ter garantias de entrada na RC, evitando uma espera infinita.
- **A implementação da Exclusão Mútua exige a sincronização entre processos**.
  - Principais Problemas envolvendo sincronização:
    - velocidade de execução dos processos
    - *Starvation*
    - Sincronização condicional

COMPARTILHAMENTO DE RECURSOS

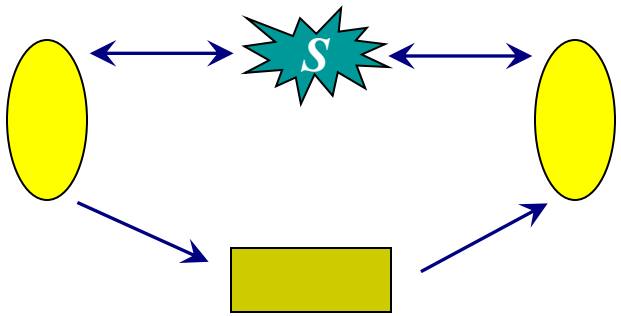
- SOLUÇÃO: *Exclusão mútua*
  - Principais Problemas:
    - **Velocidade de execução dos processos**

```
Processo_1()
{ while ( vez != 1 ){}
  RC();
  vez=2;
  func_xyz(/*lento*/);
}
```

```
Processo_2()
{ while ( vez != 2 ){}
  RC();
  vez=1;
}
```

COMPARTILHAMENTO DE RECURSOS

- SOLUÇÃO: *Exclusão mútua*
  - Principais Problemas:
    - **Starvation**
      - Prioridades/aleatoriedade: Escalonamento justo (FIFO).
    - **Sincronização condicional**
      - ex. produtor/consumidor



COMPARTILHAMENTO DE RECURSOS

- SOLUÇÃO: *Exclusão mútua*
  - Soluções de Hardware:
    - Desabilitar Interrupçoes
    - TSL (*Test and Set Lock*)

COMPARTILHAMENTO DE RECURSOS

- SOLUÇÃO: *Exclusão mútua*
  - Solução de Software:
    - Semáforo (*Dijkstra*)
      - Primitivas Up e Down;
      - Operações atômicas, implementadas como *system calls*;

```
Processo_1()
{  Down(sema);
   RC();
   Up(sema);
   func_xyz(/*lento*/);
}
```

```
Processo_2()
{  down(sema);
   RC();
   Up(sema);
}
```

COMPARTILHAMENTO DE RECURSOS

- SOLUÇÃO: *Exclusão mútua*
  - Semáforo (*Dijkstra*)

```
Down(semaforo s)
{
 if ( s == 0 )
   goto_waitlist();
 else
   s=s-1;
}
```

```
Up(semaforo s)
{
 if ( waitlist() > 0 )
    get_pid_wlist();
 else
    s=s+1;
}
```

sema=1

```
Processo_1()
{ Down(sema);
  RC();
  Up(sema);
  func_xyz(/*lento*/);
}
```

```
Processo_2()
{ Down(sema);
  RC();
  Up(sema);
}
```

COMPARTILHAMENTO DE RECURSOS

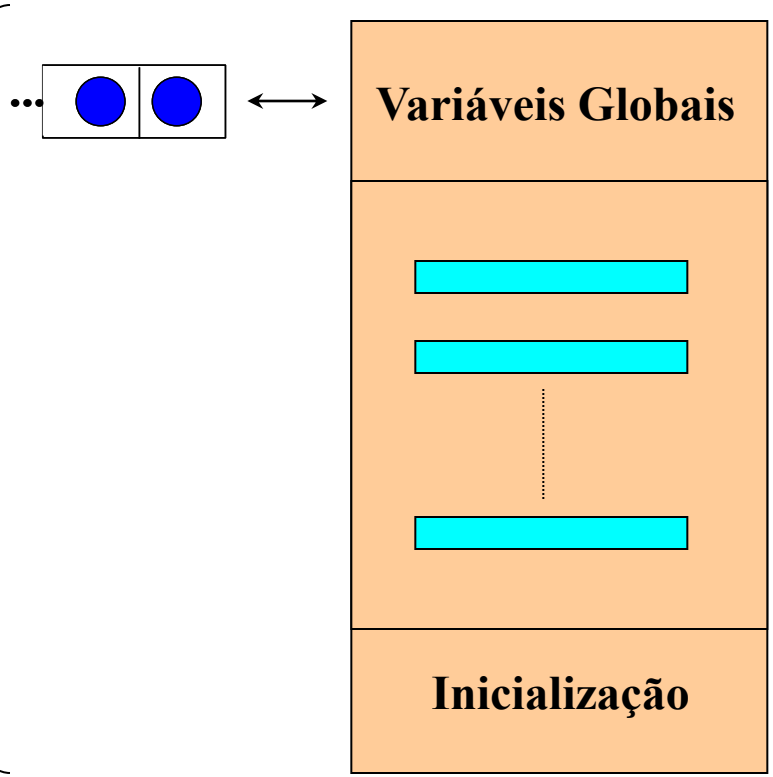
- SOLUÇÃO: *Exclusão mútua*
  - Solução de Software:
    - Monitores (*Hoare*)
      - Remove do programador, parte das responsabilidades da programação com semáforo;
      - Encapsula procedimentos e variáveis utilizadas na programação de semáforos;
      - Implementado como um módulo (classe) que automaticamente realiza o sincronismo entre os procedimentos utilizados;
      - A implementação da exclusão mútua é realizada em tempo de compilação;

COMPARTILHAMENTO DE RECURSOS

- SOLUÇÃO: *Exclusão mútua*
  - Solução de Software:
    - Monitores (*Hoare*)

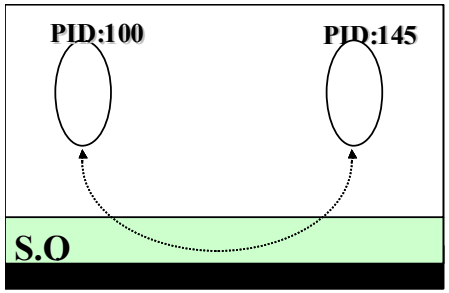
```
MONITOR RC;
  Var X: INTEGER;
 PROCEDURE Adição;
  Begin
    X := X + 1;
  END;
 PROCEDURE Subtração;
    Begin
    X := X - 1;
  END;

BEGIN
     PARBEGIN
        RC.Soma;
        RC.Subtração;
     PAREND;
END.
```

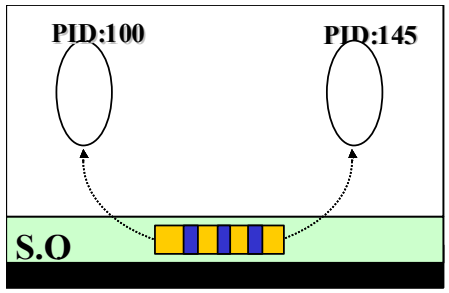


COMPARTILHAMENTO DE RECURSOS

- TROCA DE MENSAGENS
  - Usada para sincronização e comunicação entre processos:
    - SEND(PID,MSG) / RCV(PID,MSG)
  - Endereçamento
    - Direto
    - Indireto
  - Modo
    - Síncrono (*rendezvous*)
    - Assíncrono



Direto



Indireto

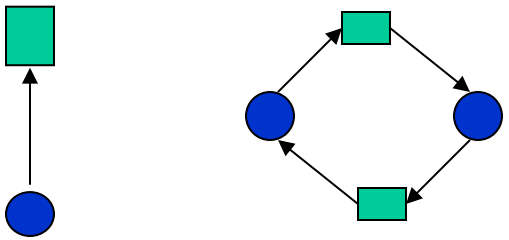
DEADLOCKS

- Quando processos se bloqueiam mutualmente na espera de um recurso indefinidamente.
- Dois processos (*multitask*) querendo imprimir um arquivo na impressora:
  - Processo A bloqueia arquivo;
  - Processo B bloqueia impressora;
  - Processo A tenta usar a impressora;
  - Processo B tenta abrir o arquivo.

  } Exemplo de *Deadlock*
- Recursos preemptíveis e não preemptíveis
  - Ex. memória (preempção) / impressora (ñ preempção)

DEADLOCKS

- Condições para *deadlock* (Coffman et al. 1971):
  - Condição de exclusão mútua;
  - Utilização de dois ou mais recursos exclusivos;
  - Recursos sem preempção, liberados explicitamente pelo processo;
  - Condição de espera circular. Um encadeamento de dois ou mais processos que esperam pelo próximo processo da cadeia.



PREVENÇÃO DE DEADLOCKS

- Garantir que uma ou mais das condições anteriores não ocorra:
  - Condição de exclusão mútua;
    - *Problemas de compartilhamento de recursos.*
  - Utilização de dois ou mais recursos exclusivos;
    - *Alocação antecipada, o que pode causar sub-utilização.*
  - Recursos sem preempção, liberados explicitamente pelo processo;
    - *Garantir preempção. Problemas de starvation e inconsistência podem ocorrer.*
  - Condição de espera circular. Um encadeamento de dois ou mais processos que esperam pelo próximo processo da cadeia.
    - *Excluir a possibilidade de múltiplas alocações de recursos exclusivos ao mesmo tempo.*

PREVENÇÃO DE DEADLOCKS

- Garantir que uma ou mais das condições anteriores não ocorra:
  - Condição de exclusão mútua;
    - *Problemas de compartilhamento de recursos.*
  - Utilização de dois ou mais recursos exclusivos;
    - *Alocação antecipada, o que pode causar sub-utilização.*
  - Recursos sem preempção, liberados explicitamente pelo processo;
    - *Garantir preempção. Problemas de starvation e inconsistência podem ocorrer.*
  - Condição de espera circular. Um encadeamento de dois ou mais processos que esperam pelo próximo processo da cadeia.
    - *Excluir a possibilidade de múltiplas alocações de recursos exclusivos ao mesmo tempo.*

PREVENÇÃO DE DEADLOCKS

- Detecção:
  - Utiliza estruturas internas no kernel do SO;
  - Algoritmos de busca são específicos para cada SO, tais como *time sharing* e de tempo real;
  - Gera grande *overhead* no kernel, por isso, normalmente não se implementa tais funcionalidades.
- Correção:
  - Preempção: Retirar recurso(s) de um processos fornecendo-o para outro processo;
  - *Rollback*: Utilização de *checkpoints;*
  - *Process killing*: Remove um ou mais processos da cadeia de dependências.