SISTEMAS OPERACIONAIS

Comunicação e Sincronização entre Processos - EAD

Análise e Desenvolvimento de Sistemas

Prof. Ms. Renato Leite

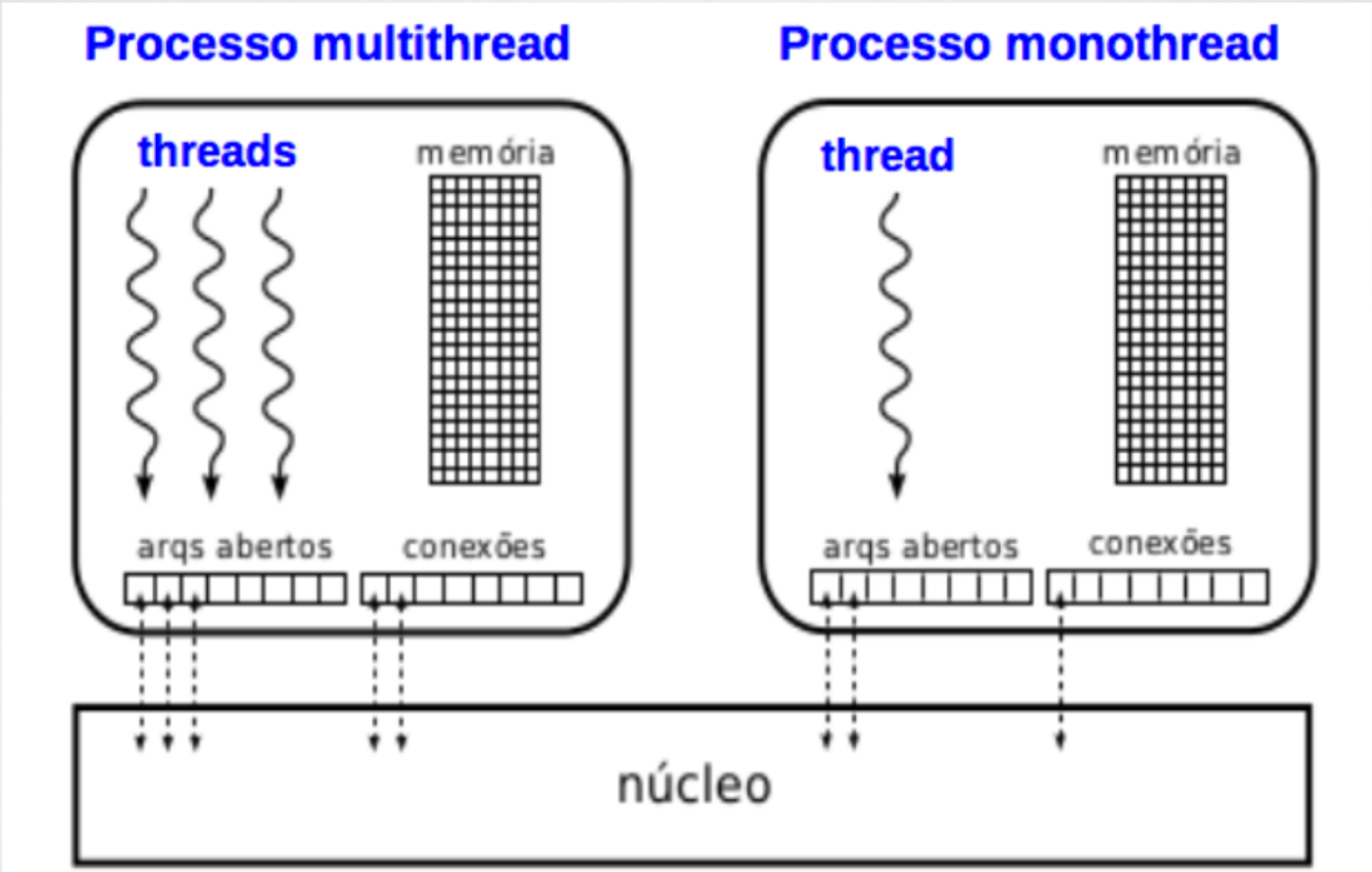
renato@sectras.edu.br

Naturalmente, sistemas multi-programáveis permitem soluções que se utilizem do poder do paralelismo de atividades para tornar a execução de tarefas mais rápida e eficiente. Como em atividades cotidianas, precisa-se de um gerente ou um fluxo de dados entre os responsáveis por cada parte da tarefa. Como fazer isso? Comunicação entre processos através de áreas compartilhadas e canais (fluxos)! E como evitar problemas no uso de recursos? Sincronização entre processos!

OBJETIVOS

- Conhecer os mecanismos de comunicação entre processos e threads;
- Entender quais problemas podem ocorrer mediante acesso de recursos compartilhados;
- Estudar as soluções propostas através de algoritmos de sincronização.

RELEMBRANDO



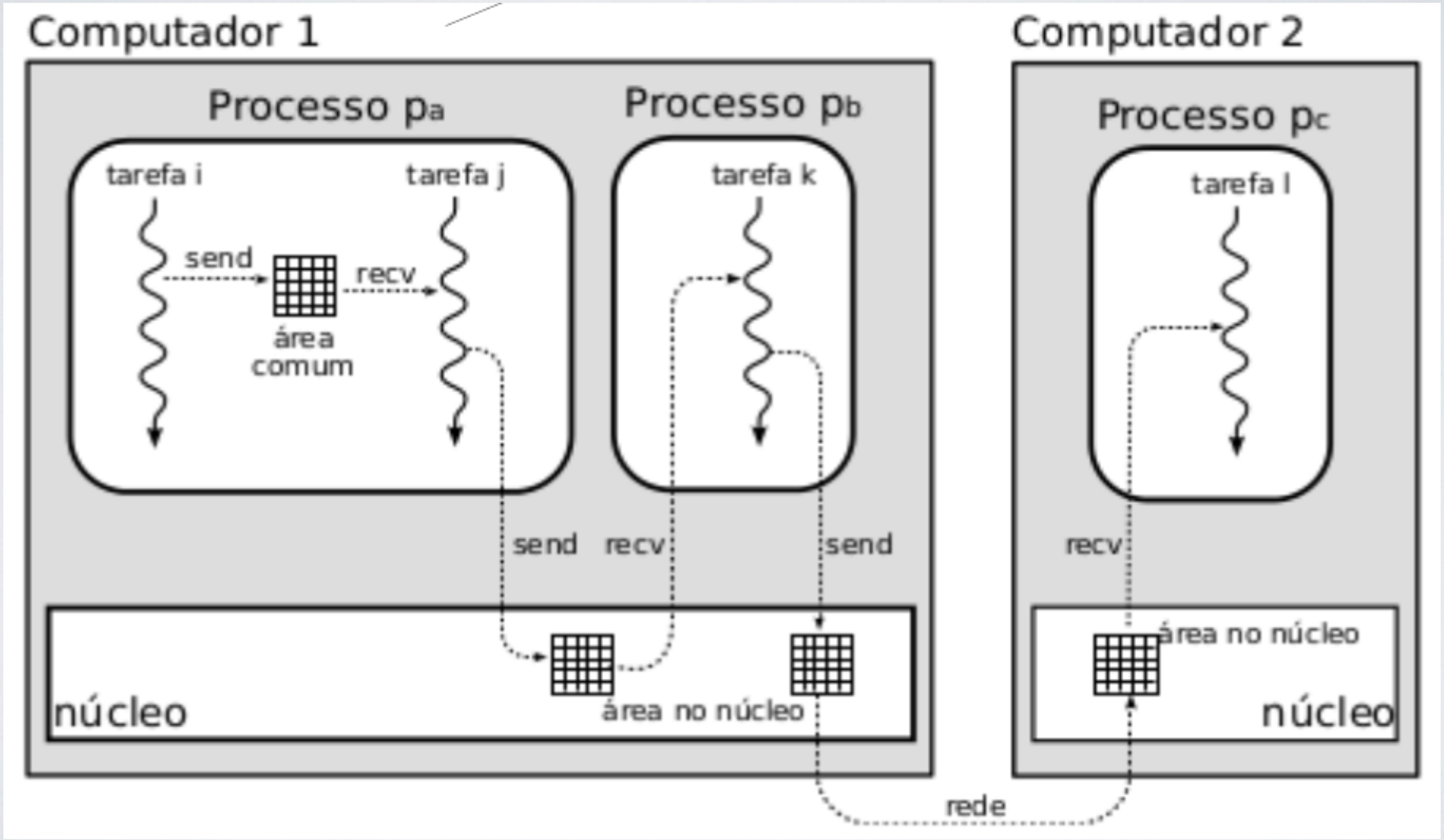
COMUNICAÇÃO ENTRE TAREFAS

- Arquivos;
- Sockets;
- Pipes;
- Área de Memória Compartilhada

COMUNICAÇÃO VIA ARQUIVOS E SOCKETS

- Processos podem trocar dados através de arquivos;
  - Um mesmo arquivo pode ser alimentado por um processo e consumido por outro;
- Processos podem enviar e receber dados através de sockets
  - Toda comunicação é feita pela pilha TCP-IP, orientado ou não a conexão (UDP);
  - Pode ser utilizado para soluções distribuídas.

SOCKETS


Computador 1
Processo pa
tarefa i
tarefa j
send
recv
área
comum
Processo pb
tarefa k
send
recv
send
núcleo
área no núcleo
Computador 2
Processo pc
tarefa l
recv
área no núcleo
núcleo
rede

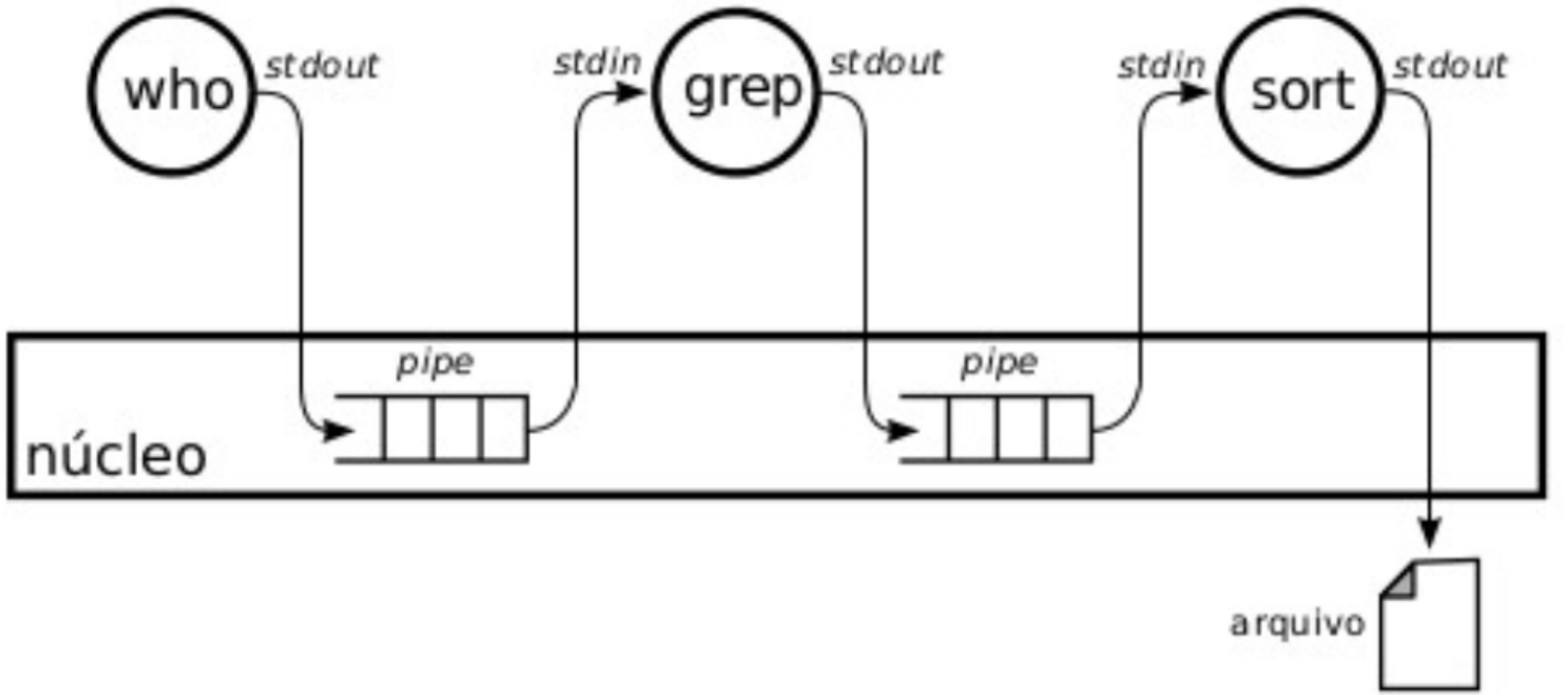
COMUNICAÇÃO VIA PIPES

- Um dos mecanismos de comunicação entre processos mais simples de usar no ambiente UNIX é o pipe, ou tubo. Na interface de linha de comandos, o pipe é frequentemente usado para conectar a saída padrão (stdout) de um comando à entrada padrão (stdin) de outro comando, permitindo assim a comunicação entre eles. A linha de comando a seguir traz um exemplo do uso de pipes:

- # who | grep marcos | sort > login-marcos.txt

PIPES

who | grep marcos | sort > login-marcos.txt



COMUNICAÇÃO ENTRE PROCESSOS VIA MEMÓRIA

- A comunicação entre **threads do mesmo processo** pode ser realizada utilizando variáveis globais;

- Caso se deseje comunicar-se com thread de outro processo, é necessário a intervenção do Kernel do S.O. o mesmo ocorre caso o processo esteja em outro computador;

- Essa comunicação deve ocorrer com controle de sincronização ou poderão haver inconsistências nos dados processados pelas aplicações.

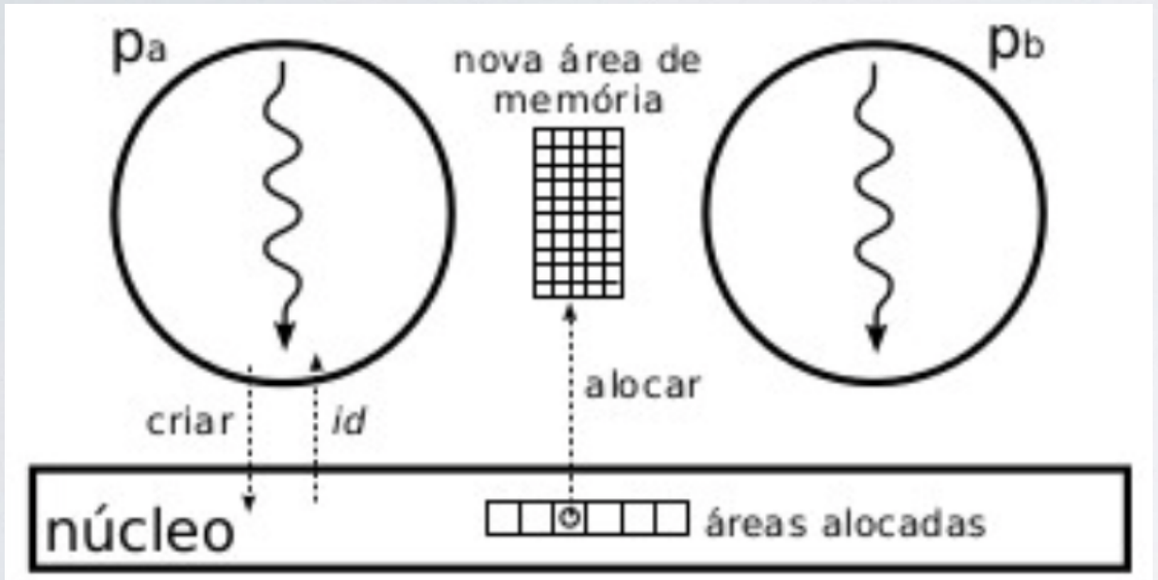
MEMÓRIA COMPARTILHADA

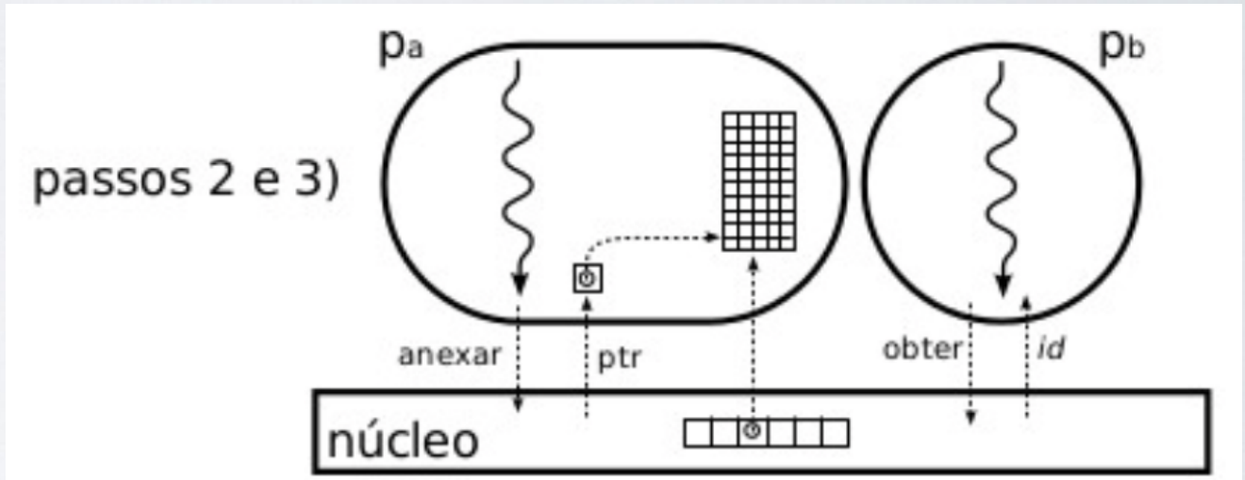
- A maioria dos sistemas operacionais atuais oferece mecanismos para o compartilhamento de áreas de memória entre processos (shared memory areas);
- As áreas de memória compartilhadas e os processos que as utilizam são gerenciados pelo núcleo, mas o acesso ao conteúdo de cada área é feito diretamente pelos processos, sem intermediação ou coordenação do núcleo;

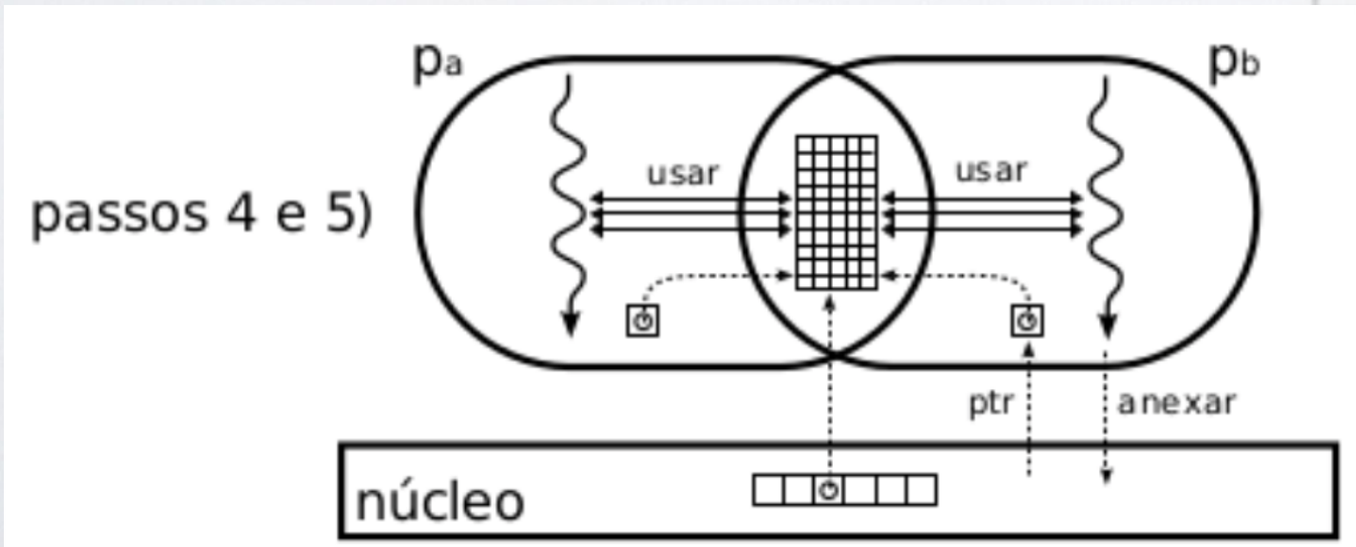
MEMÓRIA COMPARTILHADA

- Passo a passo:
  - 1 - Processo A solicita criação de área de memória compartilhada, kernel retorna id da área de memória;
  - 2 - Processo A solicita que espaço de endereçamento criado seja anexado ao seu contexto de endereçamento;
  - 3 - Processo B recebe ID da área criada;
  - 4 - Processo B solicita que espaço de endereçamento criado seja anexado ao seu contexto de endereçamento;
  - 5 - PA e PB podem agora acessar a mesma área de memória através dos ponteiros no seu espaço de endereçamento.

MEMÓRIA COMPARTILHADA

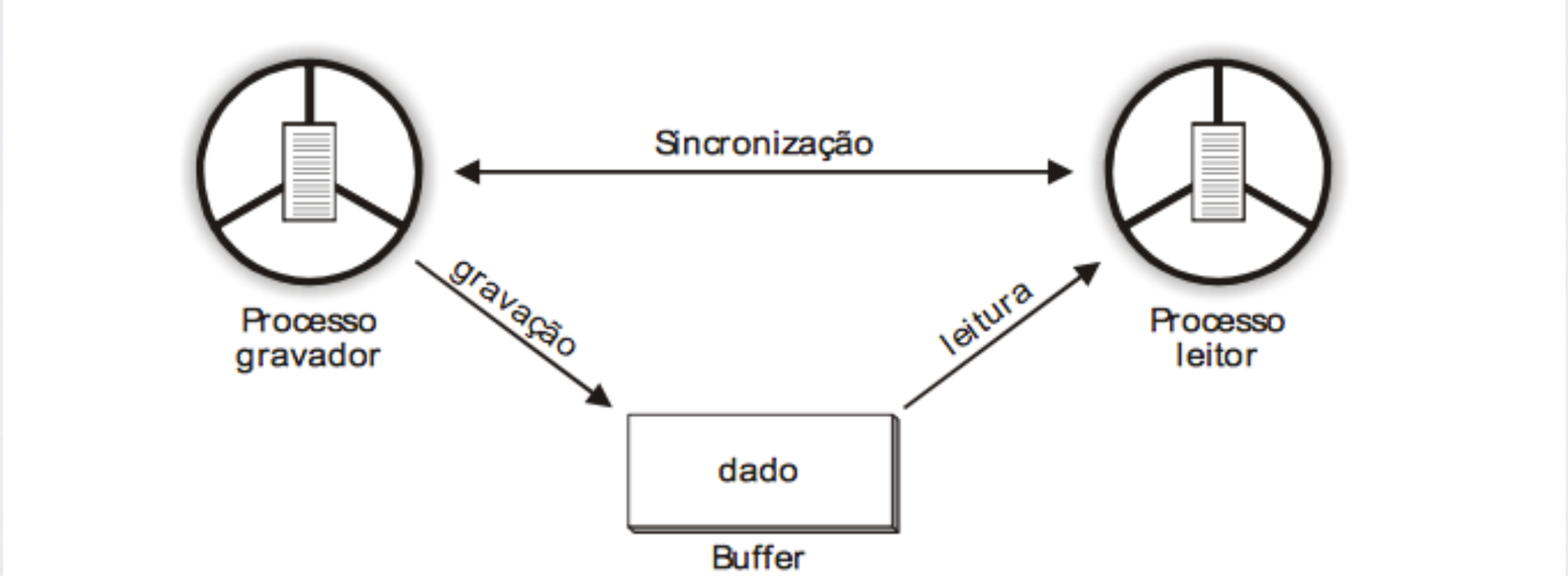




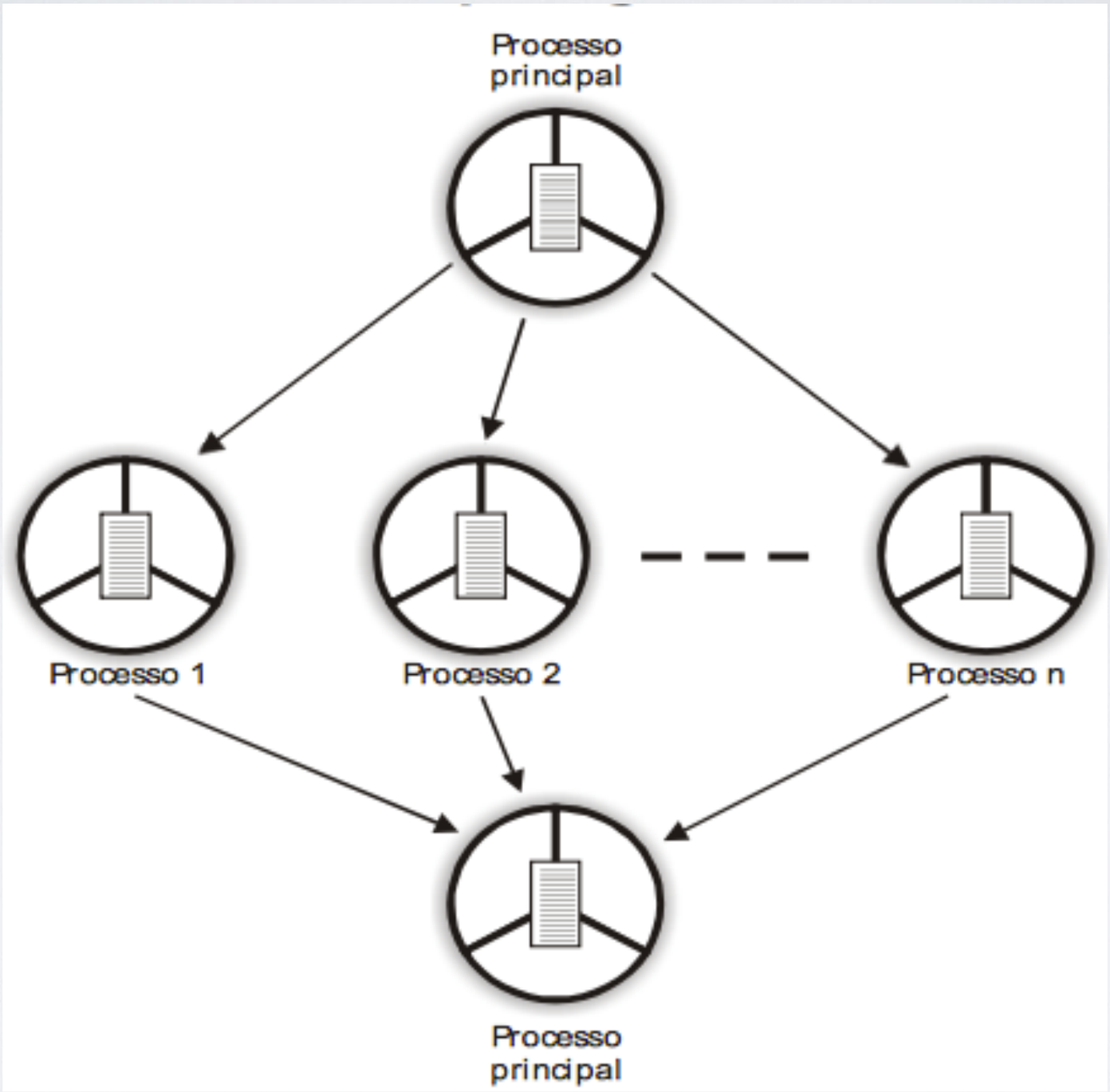


SINCRONIZAÇÃO ENTRE PROCESSOS

- Memória compartilhada ou variáveis globais são áreas disponíveis para diversos processos e necessitam de controle de acesso através de **sincronização** para concorrência.



CONCORRÊNCIA



CONCORRÊNCIA

```
X := SQRT (1024) + (35.4 * 0.23) - (302 / 7)

PROGRAM Expressao;
      VAR X, Temp1, Temp2, Temp3 : REAL;
BEGIN
   PARBEGIN                                   (ou COBEGIN)
      Temp1 := SQRT (1024);
      Temp2 := 35.4 * 0.23;
      Temp3 := 302 / 7;
   PAREND;                                    (ou COEND)
   X := Temp1 + Temp2 - Temp3;
   WRITELN ('x = ', X);
END.
```

Há concorrência?

Podemos ter para cada uma das partes do código uma thread, compartilhando variáveis globais.

Cada variável: Temp1, Temp2, Temp3 poderia ser calculada por uma thread separada.

PROBLEMAS?

- Quanto existe compartilhamento de recursos de forma concorrente, em contextos de multiprogramação, precisa-se ter prudência e desenvolver métodos de não existir inconsistências entre os dados compartilhados devido a condições de corrida;

- Enquanto uma **região crítica**, com recurso compartilhado sendo utilizado, deve-se utilizar um mecanismo de acesso exclusivo;

- Aos mecanismos que venham a propor solução a esse problema damos o nome de mecanismos de **exclusão mútua**; Saldo inicial = 100

PROBLEMAS?

- Thread A lê saldo (100)
- Thread B lê saldo (100)
- Thread A adiciona 50 → 150
- Thread B subtrai 30 → 70

Qual resultado esperado? 120 ou 170?

EXCLUSÃO MÚTUA

- Exclusão mútua nada mais é que garantir acesso exclusivo a uma região crítica, ou seja, aquela que acessa um recurso compartilhado, evitando-se condições de corrida (dois acessos simultâneos desordenados) e em casos extremos, travamento das tarefas concorrentes:

```
BEGIN
   Entra_Regiao_Critica; (protocolo de entrada)
   Regiao_Critica;
   Sai_Regiao_Critica; (protocolo de saída)
END.
```

SOLUÇÃO

```
entra_região_crítica
  Thread A lê saldo (100)
  Thread A adiciona 50 → 150
sai_região_crítica
```

```
entra_região_crítica
  Thread B lê saldo (150)
  Thread B subtrai 30 → 120
sai_região_crítica
```

Resultado: 120

Ao entrar numa região crítica, é sinalizado ao sistema operacional que aquela área compartilhada está momentaneamente reservada a uma thread. Dessa forma, as demais threads devem aguardar para acessa-la.

SOLUÇÃO VIA HARDWARE

- Pode-se implementar via hardware a solução de exclusão mútua, utilizando instruções especiais ou desabilitando interrupções;

- Ambas soluções possuem problemas graves e normalmente não são utilizadas, causando starvation e prejudicando enormemente a multi-programação.

SOLUÇÃO VIA HARDWARE - ATÔMICA

- Test and set:
  - Nessa solução, o processador possui uma instrução própria, que só permite que se libere uma região crítica entre tarefas uma de cada vez;
  - A instrução copia de uma área de memória compartilhada um valor que sinalizará a liberação da região crítica. Como garantir que apenas 1 acessará de cada vez? Essa é uma instrução que não pode ser interrompida!
- Uma variável de bloqueio controlada por uma que função utilizadora da premissa de primitiva atômica:
  - atomic_flag_test_and_set(&lock)
  - atomic_exchange() (C11),
  - pthread_spin_lock() (POSIX),

SOLUÇÃO VIA HARDWARE - ATÔMICA

- Abaixo algoritmo que exemplifica o comportamento das ações executadas em hardware

```
thread_SAQUE:
      bloqueio = falso
      enquanto (test-and-set(bloqueio) == verdadeiro):
        #aguardar
      #região crítica saque;
      bloqueio := falso;
```

```
test_and_set(bloqueio):
    status_anterior := bloqueio
    bloqueio := true
    retorna status_anterior
```

```
thread_DEPOSITO:
      bloqueio = false
      enquanto (test-and-set(bloqueio) == verdadeiro):
        #aguardar
      #região crítica deposito;
      bloqueio := falso;
```

SOLUÇÃO VIA HARDWARE - ATÔMICA

- Test and set: PROBLEM?

SOLUÇÃO VIA HARDWARE

- Desabilitar interrupções:
  - Nessa solução a interrupções são desabilitadas pela tarefa que irá acessar o recurso compartilhado;
  - Desabilitando as interrupções não há condições de outra tarefa que utilize o recurso compartilhado entrar em execução e causar condição de corrida;
  - Infelizmente ela não permite que QUALQUER outra tarefa acesse, prejudicando seriamente a multi-programação. Além disso, pode haver loop ou esquecer de habilitar as instruções;
  - Normalmente é utilizado apenas por tarefas de núcleo que não podem ser interrompidas!

SOLUÇÃO DE SOFTWARE: ESPERA OCUPADA

```
PROGRAM Algoritmo_1;
    VAR Vez : CHAR;
PROCEDURE Processo_A;
BEGIN
    REPEAT
        WHILE (Vez = 'B') DO (* Nao faz nada *);
        Regiao_Critica_A;
        Vez := 'B';
        Processamento_A;
    UNTIL False;
END;
PROCEDURE Processo_B;
BEGIN
    REPEAT
      WHILE (Vez = 'A') DO (* Nao faz nada *);
      Regiao_Critica_B;
      Vez := 'A';
      Processamento_B;
    UNTIL False;
END;
```

ESPERA OCUPADA

- As soluções inicialmente implementadas para resolver o problema de exclusão mútua sofriam da síndrome do "busy waiting", ou seja, o processo permanecia em loop checando a liberação do acesso à região crítica e, portanto consumindo ciclos de UCP;

- As soluções mais atuais adotam primitivas que permitem que um processo seja colocado em estado de espera e reativado apenas quando o recurso estiver disponível. Nesta linha foram introduzidos os semáforos e os monitores.

PROBLEMA DO PRODUTOR CONSUMIDOR

```
PROCEDURE Produtor;
BEGIN
    REPEAT
        Produz_Dado (Dado_1);
        WHILE (Cont = TamBuf) DO (* Nao faz nada *);
        Grava_Buffer (Dado_1, Buffer);
        Cont := Cont + 1;
    UNTIL False;
END;

PROCEDURE Consumidor;
BEGIN
    REPEAT
        WHILE (Cont = 0) DO (* Nao faz nada *);
        Le_Buffer (Dado_2, Buffer);
        Consome_Dado (Dado_2);
        Cont := Cont - 1;
    UNTIL False;
END;
```

Ainda temos o problema da espera ocupada!

SEMÁFOROS

- Um holandês, chamado Dijkstra, em 1965, vendo que não era bom utilizar ciclos do processador com instruções vazias e consequentemente gerar desperdício de tempo de CPU com a espera ocupada, propôs um mecanismo que une exclusão mútua via software com instruções de hardware;

- Nessa solução não há possibilidade de todos os processos da máquina serem bloqueados como nos primeiros exemplos de hardware. Além disso, não há espera ocupada!

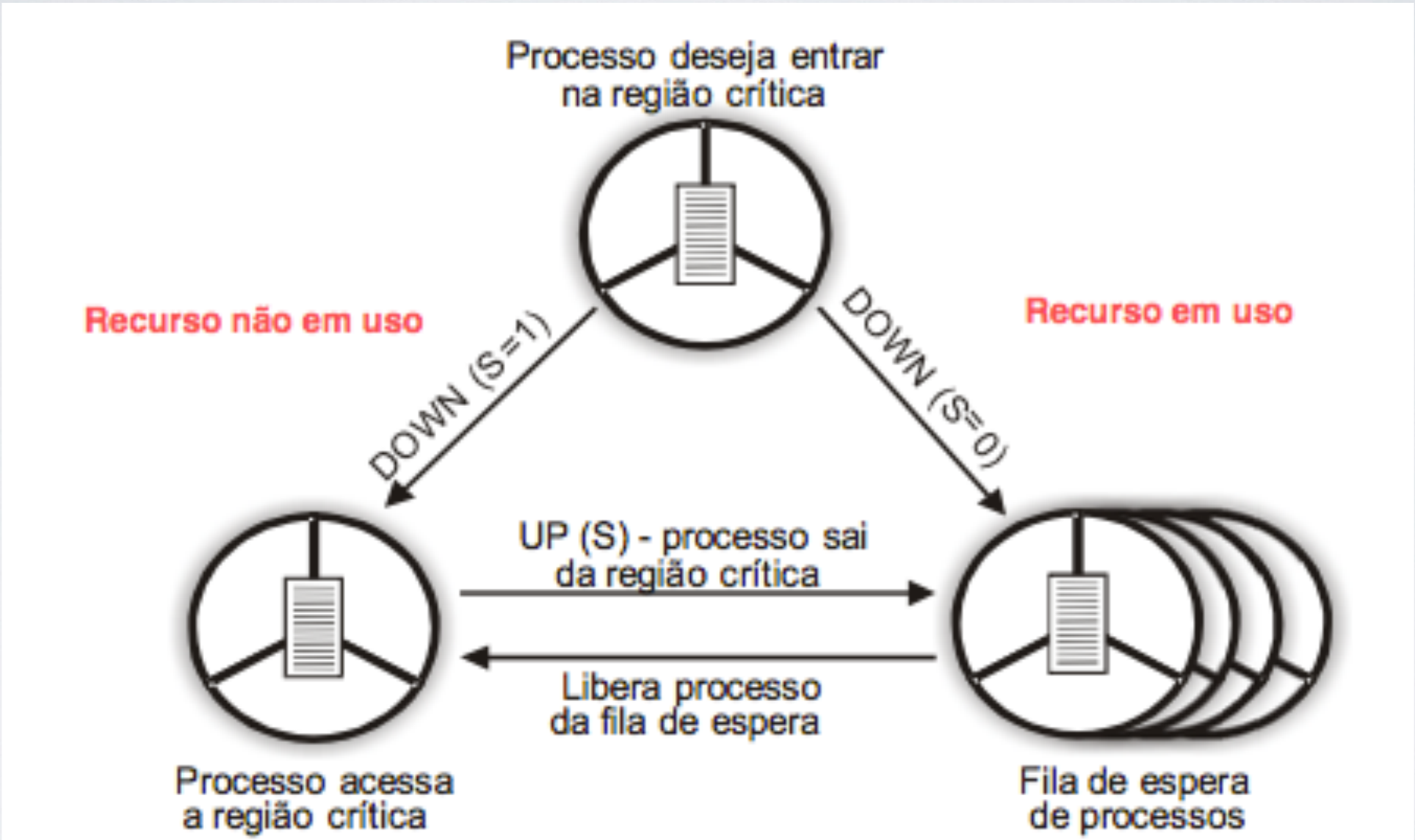
SEMÁFOROS

- Semáforo nada mais é que uma variável inteira, sempre não-negativa, que só pode ser manipulada por duas instruções de máquina DOWN e UP;
- Cada semáforo é associado a um único recurso compartilhado e seu significado é:
  - 0 → inexistência de recurso disponível; >0 → dispolibilidade de recurso.
- Um processo para entrar em sua região crítica precisa antes decrementar o conteúdo do semáforo (down), que só é permitido se ele for ">0" e ao sair incrementa-o (up), liberando o recurso para os demais processos. Up e Down são atômicas e geralmente implementadas como system calls. Quando o processo executa um down em um semáforo vazio, ele automaticamente é colocado em estado de espera (sleeping) e é automaticamente reativado quando algum outro processo liberar (up) aquele recurso.

SEMÁFOROS

- Semáforo nada mais é que uma variável inteira, sempre não-negativa, que só pode ser manipulada por duas instruções de máquina DOWN e UP;
- Cada semáforo é associado a um único recurso compartilhado e seu significado é:
  - 0 → inexistência de recurso disponível;
  - > 0 → disponibilidade de recurso.
- Um processo para entrar em sua região crítica precisa antes decrementar o conteúdo do semáforo (down), que só é permitido se ele for ">0" e ao sair incrementa-o (up), liberando o recurso para os demais processos. Up e Down são atômicas e geralmente implementadas como system calls. Quando o processo executa um down em um semáforo vazio, ele automaticamente é colocado em estado de espera (sleeping) e é automaticamente reativado quando algum outro processo liberar (up) aquele recurso.

SEMÁFOROS


Processo deseja entrar
na região crítica
Recurso não em uso
Recurso em uso
DOWN (S=1)
DOWN (S=0)
UP (S) - processo sai
da região crítica
Libera processo
da fila de espera
Processo acessa
a região crítica
Fila de espera
de processos

SEMÁFOROS

- Semáforos se dividem poder assumir dois comportamentos:
  - Mutex: semáforos binários, assumindo o valor 0 ou 1;
  - Valores maiores que 1 para caracterizar comportamento de regras de negócio. Nessa situação, cada acesso a uma área de recurso compartilhado é precedida de um DOWN que irá indicar que foi utilizado uma parte do recurso, do mesmo modo que liberação de mais recursos será seguido de UP.

SEMÁFOROS

```
PROCEDURE Consumidor;
BEGIN
    REPEAT
        DOWN (Cheio);
        DOWN (Mutex);
        Le_Buffer (Dado_2, Buffer);
        UP (Mutex);
        UP (Vazio);
        Consome_Dado (Dado_2);
    UNTIL False;
END;
```

```
PROCEDURE Produtor;
BEGIN
    REPEAT
        Produz_Dado (Dado_1);
        DOWN (Vazio);
        DOWN (Mutex);
        Grava_Buffer (Dado_1, Buffer);
        UP (Mutex);
        UP (Cheio);
    UNTIL False;
END;
```

MONITORES

- O uso de semáforos exige muito cuidado do programador pois qualquer engano pode levar a problemas de sincronização imprevisíveis e difíceis de reproduzir, face à execução concorrente dos processos;

- Monitores são mecanismos de alto nível implementados pelo próprio compilador. Para isto basta especificar todas as regiões críticas em forma de procedimentos do monitor, e o compilador se encarregará de implementar a exclusão mútua;

- Esse mecanismo foi proposto na década de 1970 por Hansen e Hoare;

MONITORES

```
MONITOR Regiao_Critica;
    VAR X : INTEGER;

    PROCEDURE Soma;
    BEGIN
        X := X + 1;
    END;

    PROCEDURE Diminui;
    BEGIN
        X := X - 1;
    END;
```

Através da indicação de área de código com acesso a regiões críticas, o compilador gera protocolos de saída e entrada específicos.

```
PROCEDURE Produz;
BEGIN
    IF (Cont = TamBuf) THEN WAIT (Cheio);
    .
    .
    IF (Cont = 1) THEN SIGNAL (Vazio);
END;

PROCEDURE Consome;
BEGIN
    IF (Cont = 0) THEN WAIT (Vazio);
    .
    .
    IF (Cont = TamBuf - 1) THEN SIGNAL (Cheio);
END;
```

EXERCÍCIO

- Estude a condição inadequada de implementação em software de concorrência que gera o chamado dead-lock e explique os dois problemas seguintes:
  - Jantar dos filósofos;
  - Problema do barbeiro;

REFERÊNCIAS

- MACHADO, Francis B.; MAIA, Luiz Paulo. Arquitetura de Sistemas Operacionais. 3a ed. Rio de Janeiro : LTC, 2002;

- Baseado em partes nas notas de aula do professor Pedro Filho, IFPB, Campus Patos;

DÚVIDAS?