Programação Concorrente

Paradigmas de Linguagens de Programação

Profa. Heloisa

Programação concorrente

- Programação concorrente consiste em construir programas contendo múltiplas atividades que progridem em paralelo.

- Existem duas **categorias de controle** de unidades concorrentes:

- Concorrência Física – unidades do mesmo programa executam simultaneamente, supondo que mais de um processador esteja disponível

- Concorrência lógica – programador e aplicativo supõem a existência de múltiplos processadores, quando de fato a execução real está se desenvolvendo intercaladamente em um único processador

- Linguagens de programação oferecem mecanismos para facilitar a construção de sistemas computacionais concorrentes.

- Do ponto de vista do programador e do projetista de linguagem, **concorrência lógica** é o mesmo que **concorrência física**.

- Cabe ao implementador da linguagem fazer a correspondência da lógica com o hardware subjacente.

- Tanto a concorrência lógica quanto a física permitem que o conceito de **concorrência** seja usado como uma metodologia de projeto de programas.

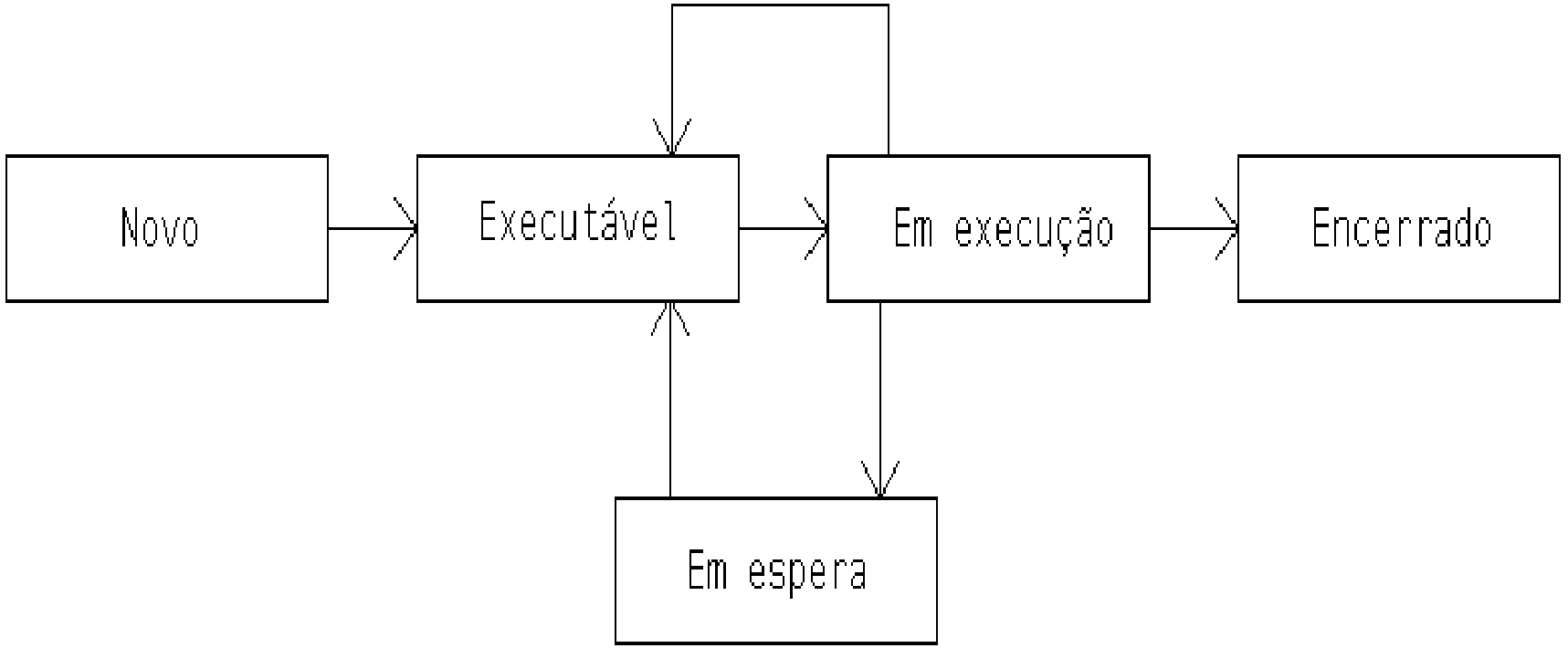
Processos

- Processos são programas em execução;
- Vários processos podem estar associados ao mesmo programa, mas são tratados independentemente;
- Cada processo possui um contexto: espaço de endereçamento, informações de controle, sua identificação, suas variáveis de ambiente, etc;
- Um programa pode ser utilizado por mais de um usuário simultaneamente, sem que a execução de um processo sofra interferência do outro.

Estados de um processo – processador ocupado por apenas um processo

- Novo – foi criado mas sua execução não começou.
- Executável ou pronto – está pronto para ser executado mas não está executando, ou porque o processador está ocupado por outro processo ou estava executando e foi bloqueado. Ficam na fila de programas prontos;
- Rodando (em execução) – estado em que ocupa o processador; poderá passar direto para o estado executável se acabar sua fatia de tempo de posse do processador;
- Em espera – estava rodando, fez uma chamada de sistema (impressora) e a execução é interrompida; quando ocorre o evento esperado (chamada atendida) e estado passa a ser executável;
- Morto (encerrado) – não está mais ativo, terminou sua execução.

Transição de Estados

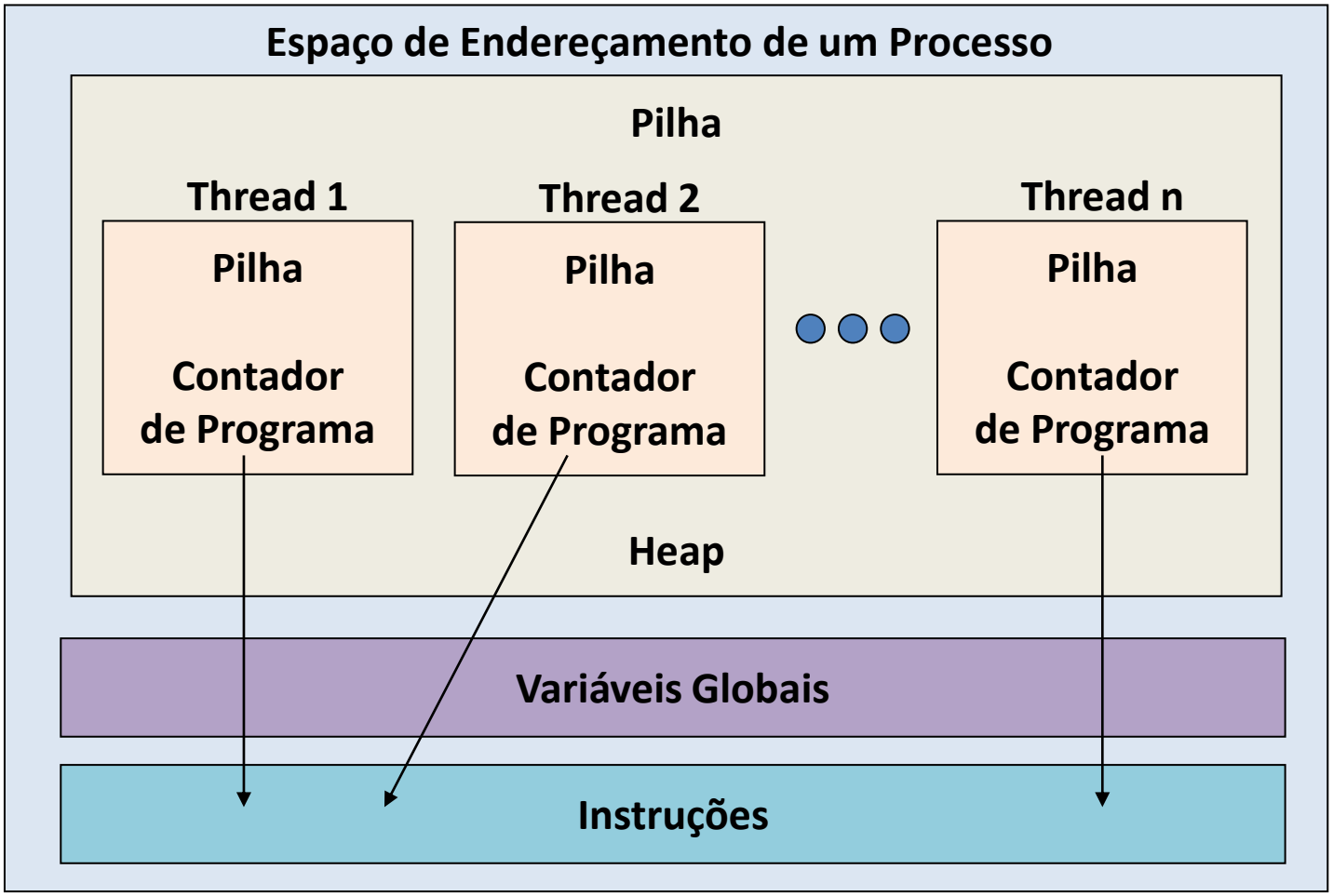


Características de um programa concorrente:

- Vivacidade (liveness): continua a executar
- Deadlock: perda da vivacidade – tarefas esperam por recursos que estão indisponíveis e só podem ser gerados por tarefas que também estão bloqueadas;
- Lockout – duas ou mais tarefas ficam esperando por um evento que nunca acontecerá;
- Starvation – uma tarefa tem a aquisição de um determinado recurso postergada indefinidamente.

Threads

- Um mesmo processo pode ter vários fluxos de execução;
- “Threads” são fluxos de execução (linhas de execução) concorrentes que compartilham recursos do processo do qual são originários;
- “Threads” também tem estados, características próprias e precisam ser gerenciados.
- Threads oferecem vantagens para a programação concorrente pois são mais leves que o processo – menor custo de criação, utilizam recursos de um processo criado anteriormente, compartilham memória com o processo e outros threads;


Espaço de Endereçamento de um Processo
Pilha
Thread 1
Pilha
Contador de Programa
Thread 2
Pilha
Contador de Programa
Thread n
Pilha
Contador de Programa
Heap
Variáveis Globais
Instruções

Distinção:
tarefa X subprograma

- Tarefa – unidade de um programa que pode estar em execução concorrente com outras unidades do mesmo programa. Cada tarefa pode originar um “thread”.

- Uma tarefa pode ser iniciada implicitamente
  - (subprograma deve ser chamado explicitamente)

- Quando uma unidade invoca uma tarefa, não precisa esperar sua conclusão para prosseguir
  - (subprogramas devem esperar a conclusão das unidades chamadas para prosseguir)

- Quando a execução de uma tarefa é concluída o controle pode ou não retornar à unidade que iniciou essa execução
  - (Quando a execução de um subprograma é concluída o controle volta à unidade que o chamou)

Controle de execução de tarefas

- É possível existir mais tarefas do que processadores, logo a execução de tarefas deve ser controlada.

- **Scheduler (escalonador)** – gerencia o compartilhamento dos processadores pelas tarefas.

- Implementa um algoritmo que decide qual tarefa será executada em qual processador

Comunicação entre tarefas

- A comunicação entre tarefas é o foco principal da programação concorrente

- Em geral, tarefas devem usar alguma forma de comunicação para:
  - sincronizar suas execuções
  - compartilhar dados (ou ambos)

- Comunicação pode ser por meio de:
  - **Memória compartilhada**
  - Passagem de mensagem

- Tarefa disjunta – é aquela que não se comunica ou não afeta a execução de nenhuma outra tarefa no programa

Passagem de mensagem
(troca de mensagem)

- mecanismo que permite comunicação entre processo ou tarefas executados simultaneamente

- usado para sincronizar unidades em um sistema distribuído, no qual cada processador tem sua própria memória

- geralmente implementado pelo sistema operacional, disponibilizando duas chamadas básicas: send (envio) e receive (recepção)

Comunicação entre tarefas que compartilham dados

- Sincronização: Mecanismo que controla a ordem de execução das tarefas.

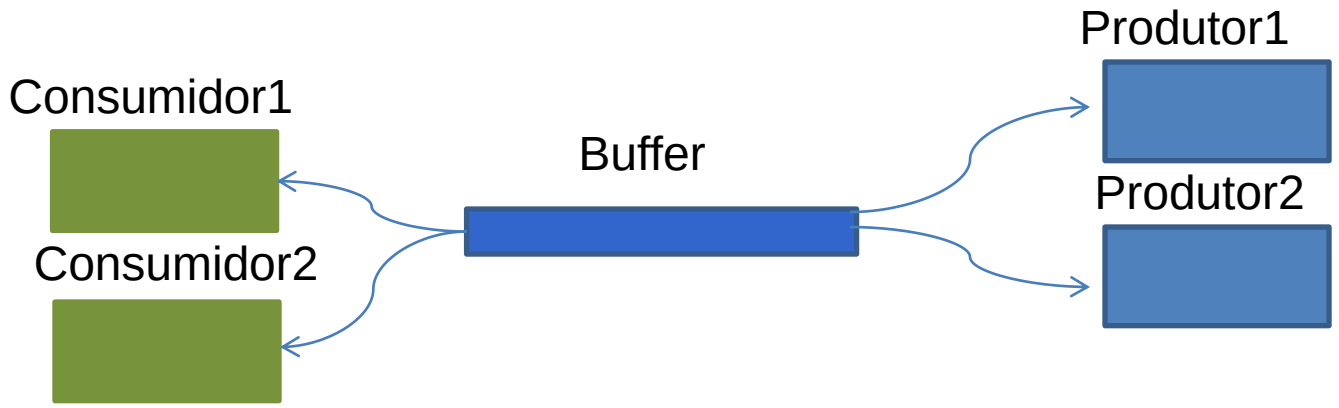
- Tipos de sincronização:

- **Sincronização de cooperação** entre tarefas A e B: quando a tarefa A precisa aguardar que B conclua alguma atividade específica antes de prosseguir (sincronização por condição).

- **Sincronização de competição** entre duas tarefas: quando ambas requerem o uso de algum recurso que não pode ser usado simultaneamente (sincronização por exclusão mútua)

Exemplo - Sincronização de cooperação
Problema produtor-consumidor

- Uma unidade de programa produz um dado ou recurso usado por outra unidade.
- Dados produzidos são depositados em um buffer pelo produtor e removidos pelo consumidor
- A seqüência de retiradas e depósitos deve ser sincronizada.
- Os usuários do dado compartilhado devem cooperar para que o buffer seja usado corretamente.



Sincronização de competição – acesso à região crítica

- **Exclusão mútua: impedir que dois ou mais processos** acessem um mesmo recurso ao mesmo tempo.

- **Região Crítica: parte do código do programa onde é feito o** acesso à memória compartilhada (ou ao recurso compartilhado), ou seja, é a parte do programa cujo processamento pode levar à ocorrência de condições de corrida
  - Há uma condição de corrida quando dois ou mais processos estão acessando dados compartilhados e o resultado depende de qual processo roda quando.

Exemplo - Sincronização de competição
Acesso à região crítica

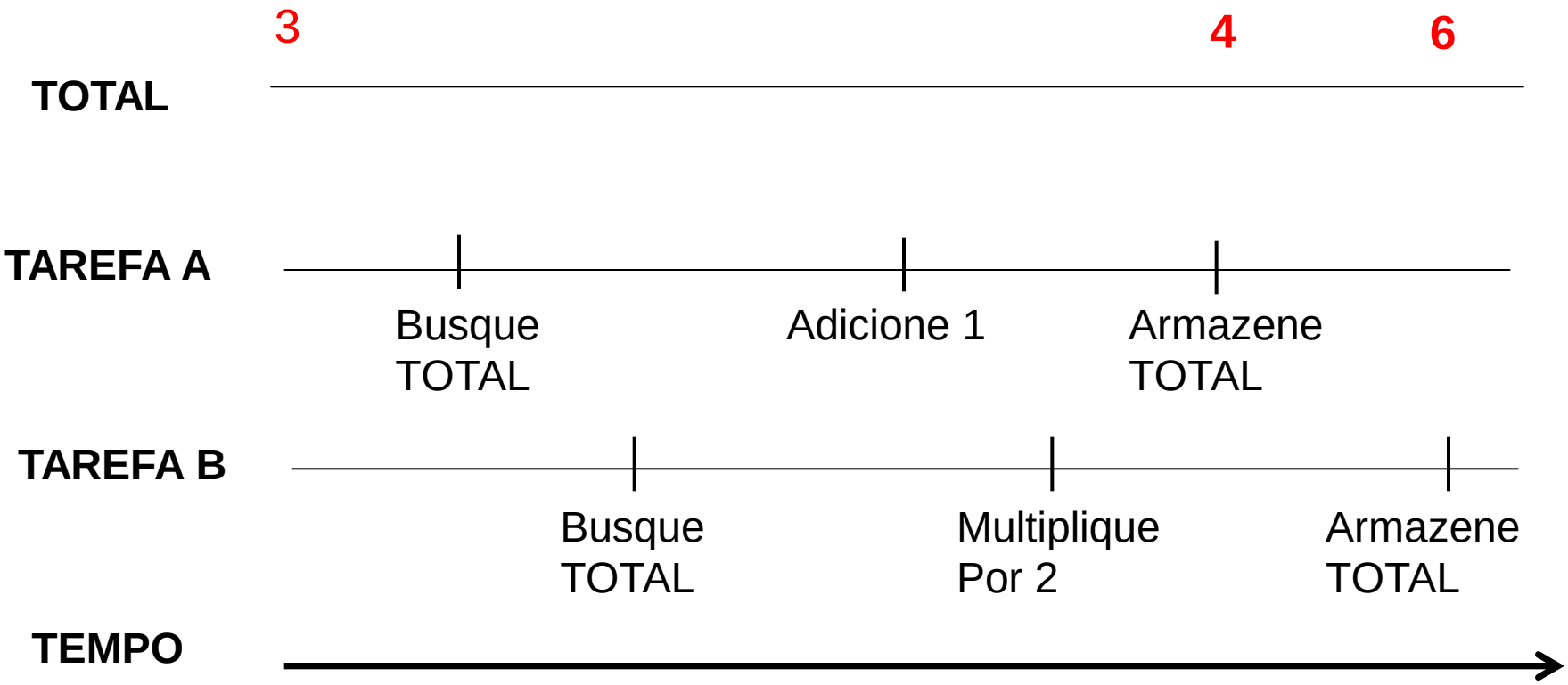
Impede que duas tarefas acessem uma estrutura de dados compartilhada ao mesmo tempo

Cenário:

- Tarefa A: adicionar 1 a TOTAL
- Tarefa B: multiplicar TOTAL por 2

- Cada tarefa realiza sua operação por:
  - buscando o valor de TOTAL,
  - realizando uma operação aritmética e
  - colocando o novo valor de volta em TOTAL

- Sem sincronização de competição, 3 resultados são possíveis:


3
4
6
TOTAL
TAREFA A
Busque
TOTAL
Adicione 1
Armazene
TOTAL
TAREFA B
Busque
TOTAL
Multiplique
Por 2
Armazene
TOTAL
TEMPO

Resultados

- Se tarefa A completa sua operação antes que a tarefa B comece

  TOTAL = 8 (correto)

- Se A e B buscam o valor de TOTAL antes que um novo valor seja depositado de volta:
  - Se A devolve o novo valor primeiro

    TOTAL = 6
  - Se B devolve o novo valor primeiro

    TOTAL = 4

Outro exemplo – depósitos em uma conta bancária

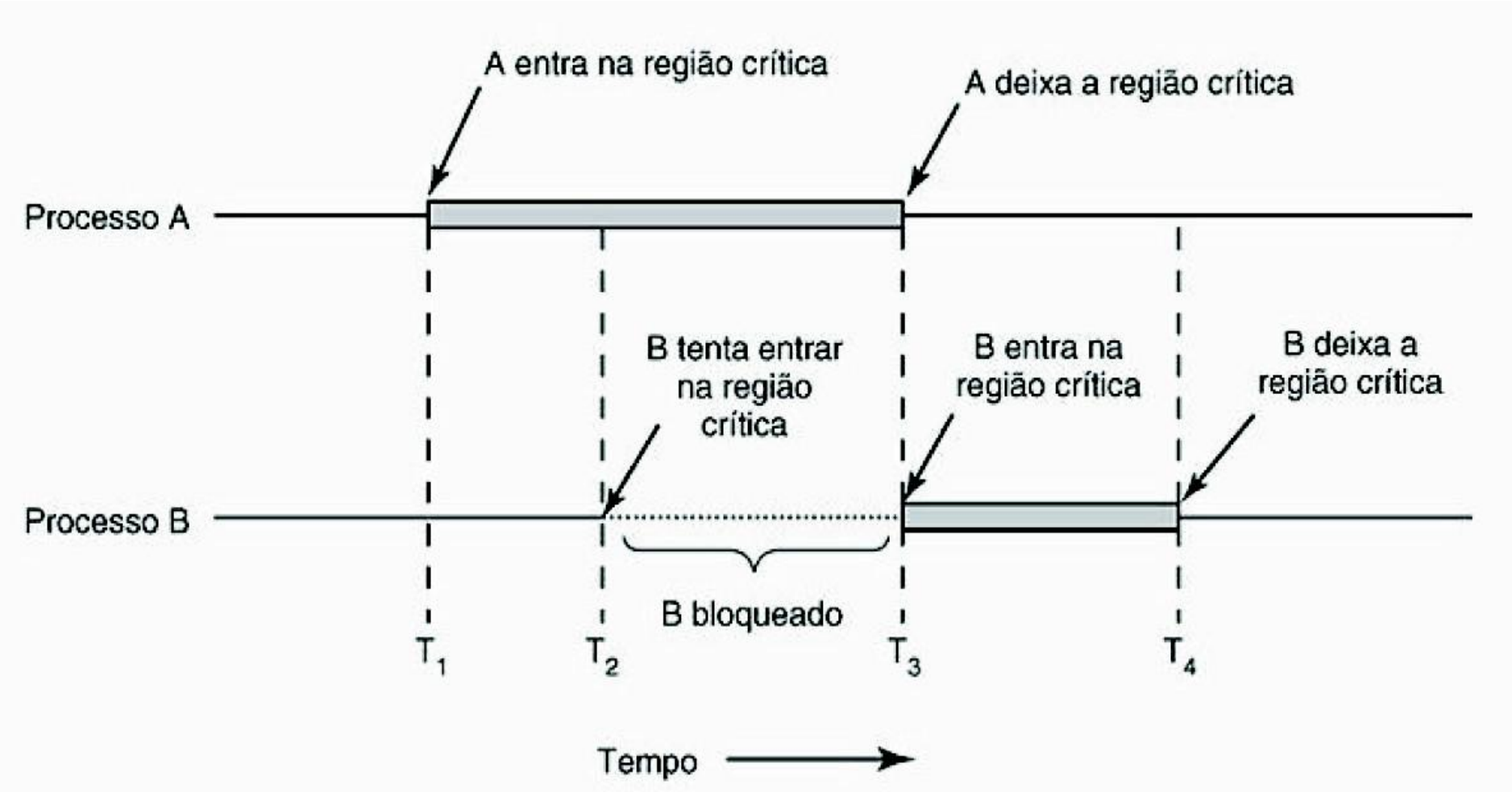
- Duas Threads vão fazer depósitos na mesma conta.
- Se o acesso não for mutuamente exclusivo, o resultado pode não ser o esperado
- Um dos depósitos não foi registrado

Thread 1	Thread 2	Saldo
Lê saldo: R$ 1.000		R$ 1.000
	Lê saldo: R$ 1.000	R$ 1.000
	Deposita R$ 300	R$ 1.000
Deposita R$ 200		R$ 1.000
Atualiza saldo R$ 1.000 + R$ 200		R$ 1.200
	Atualiza saldo R$ 1.000 + R$ 300	R$ 1.300

Sincronização de competição

Fluxo de execução de um processo que implementa a sincronização por exclusão mútua:

1. Executa região não crítica
2. Executa ''procedimento'' para entrar na região crítica
3. Executa sua REGIÃO CRÍTICA
4. Executa ''procedimento'' para deixar a região crítica
5. Executa região não crítica

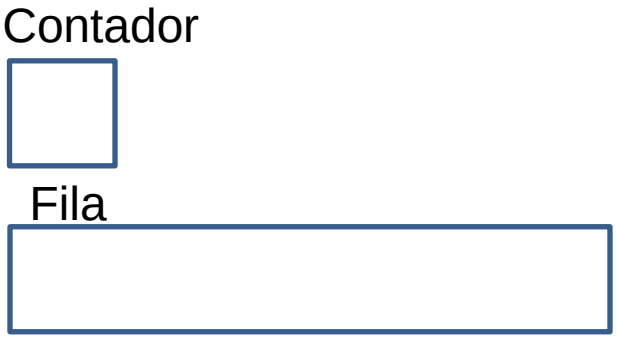

A entra na região crítica
A deixa a região crítica
Processo A
B tenta entrar na região crítica
B entra na região crítica
B deixa a região crítica
Processo B
B bloqueado
T1
T2
T3
T4
Tempo

Métodos para acesso mutuamente exclusivo a um recurso compartilhado:

- Semáforos
- Monitores

Semáforos

- Mecanismo usado para sincronização por competição ou por cooperação.



- É uma estrutura de dados:
  1 inteiro + fila de descritores de tarefas
- Conceito: colocar “guardas” ou guias ao redor do código que acessa a estrutura compartilhada (região crítica) e só permitir sua execução em determinadas condições.
- Deve usar uma técnica que garanta que todas as tentativas de execução ocorram armazenando as requisições por acesso na fila de descritores de tarefas.

Operações básicas:

- As duas únicas operações associadas com os semáforos são:

- P (passeren (holandes)- testar): wait
  - A operação P sobre um semáforo S testa se o processo que executou P(S) pode ou não entrar na região crítica. Se não puder, será colocado na fila de espera de S.
- V (vrygeren - liberar): release
  - A operação V sobre um semáforo S sinaliza ao semáforo que o processo não está mais na região crítica e retira outro processo da fila de espera, colocando-o em execução

Subprogramas wait e release

- As operações de semáforos são feitas por meio de chamadas aos subprogramas wait e release.
- Wait e release devem ter acesso a fila de tarefas prontas.

Wait:

- testa o contador de um semáforo.
- Se o contador for maior que zero, o chamador pode realizar sua operação. O contador é decrementado, para indicar que existe um a menos do que o semáforo está contando.
- Se o contador for zero, o chamador deve ser colocado na fila de espera do semáforo e o processador deve ser dado a outra tarefa pronta.

Subprogramas wait e release

```
wait(aSemaforo)

  if contador do aSemaforo > 0
  then
      Decrementar o contador
  else

      Colocar o chamador de aSemaforo na fila do aSemaforo
     Transferir controle para outra tarefa pronta
         (se a fila de tarefas prontas está vazia, ocorre deadlock)

end
```

Subprogramas wait e release

- Subprograma release e usado por uma tarefa para permitir que outra tarefa utilize o que o semáforo estiver contando.

Release:

- Se a fila do semáforo estiver vazia, (nenhuma tarefa está esperando) incrementa o contador para indicar que existe mais uma unidade daquilo que o semáforo estiver contando.

- Se uma ou mais tarefas estiverem esperando, remove uma das tarefas da fila do semáforo para a fila de tarefas prontas.

release (aSemaforo)

if fila do aSemaforo está vazia (nenhuma tarefa está esperando)
then
incrementar o contador
else
Colocar o chamador de aSemaforo na fila de tarefas prontas
Transferir controle para outra tarefa da fila do aSemaforo
end

Sincronização de cooperação usando semáforos

- Exemplo: buffer compartilhado
- O buffer deve ter meios de manter o número de posições vazias e o número de posições ocupadas
- As filas de tarefas contém tarefas que foram bloqueadas

Variáveis semáforo:

- emptyspots :
  - Contador:armazena o número de localizações vazias.
  - Fila: armazena as tarefas que esperam por posições vazias no buffer
- fullspots :
  - Contador: armazena o número de localizações preenchidas.
  - Fila: armazena as tarefas que esperam que valores sejam colocados no buffer

Sincronização de cooperação usando semáforos

- Implementação de Buffer:
- Tipo dado abstrato com as seguintes operações:

  - DEPOSIT : insere dados no buffer
  - FETCH : retira dados do buffer

- As operações DEPOSIT e FETCH são solicitadas por tarefas que querem escrever ou retirar dados do buffer

Para executar a operação DEPOSIT, deve ser feito:

- Verifica emptyspots
  - Se emptyspots.count >0 armazena dado no buffer e decrementa emptyspots
  - Se emptyspots.count = 0, o chamador de DEPOSIT espera na fila de emptyspots até que uma posição vazia esteja disponível.
- Após a execução de DEPOSIT, incrementa o contador de fullspots.

Para executar a operação FETCH, deve ser feito:

- Verifica fullspots
  - Se fullspots.count > 0 remove dado e decrementa fullspots
  - Se fullspots.count = 0 o chamador de FETCH espera na fila de fullspots
- Após finalizar a operação, incrementa contador de emptyspots

```
semaphore fullspots, emptyspots;
fullspots.count := 0;
emptyspots.count := BUFLEN;

task producer;
  loop
  /* produzir VALOR  */
  wait(emptyspots);   /* espere por espaço */
  DEPOSIT (VALOR);
  release (fullspots);    */ incremente espaços preenchidos*/
  end loop;
end producer;
```

```
task consumer;
  loop
  wait(fullspots);   /* certifique-se – não vazio */
  FETCH (VALOR);
  release (emptyspots);    */ incremente espaços vazios*/
  /* consumir VALOR  */
  end loop;
end consumer;
```

Sincronização de competição

- O acesso a estruturas compartilhadas pode ser controlado por um semáforo adicional para implementar a sincronização de competição.
- Esse semáforo não precisa contar recursos, mas deve apenas indicar se a estrutura está sendo usada ou não
- O método wait permite acesso apenas se o contador do semáforo tem valor 1, que indica que a estrutura compartilhada não está sendo usada.
- Se o contador tiver valor 0, significa que a estrutura está sendo usada e a tarefa é colocada na fila do semáforo
- Este tipo de semáforo é chamado de semáforo binário.

Sincronização de competição

- Exemplo:
  - Buffer compartilhado com um semáforo extra que controla os acessos indicando se o buffer está sendo usado ou não (Semáforo Binário)

- Wait :
  - Se contador do semáforo = 1 permite acesso ao buffer (buffer não está sendo acessado)
  - Se contador do semáforo = 0 não permite acesso e coloca a tarefa na fila do semáforo (buffer está sendo acessado)

- Inicializações:
  - Contador = 1
  - Filas dos semáforos com “vazio”

Exemplo com sincronização de cooperação e de competição

```
semaphore access, fullspots, emptyspots;
access.count := 1;
fullspots.count := 0;
emptyspots.count := BUFLEN;

task producer;
  loop
  /* produzir VALOR */
  wait (emptyspots);   /* esperar por espaço vazio
  wait( access);         /* esperar por acesso
  DEPOSIT (VALOR);
  release (access);     /* liberar acesso
  release (fullspots);   /* incrementar espaços vazios
  end loop;
end producer;
```

```
task consumer;
  loop
  wait (fullspots);       /* confirmar que não está vazio
  wait( access);          /* esperar por acesso
  FETCH (VALOR);
  release (access);     /* liberar acesso
  release (emptyspots);   /* incrementar espaços vazios
  /* consumir VALOR */
end loop;
end consumer;
```

Outra implementação para semáforos

O contador do semáforo pode assumir valores negativos. Esse valor dá uma indicação do tamanho da fila de tarefas na lista de espera:

```
wait(Semaphore s) {
  s=s-1;
  if (s<0)
  { // add process to queue block(); }
}
release(Semaphore s){
  s=s+1;
  if (s<=0)
  { // remove process p from queue wakeup(p); }
 }
Init(Semaphore s , Int v){
 s=v;
 }
```

Monitores

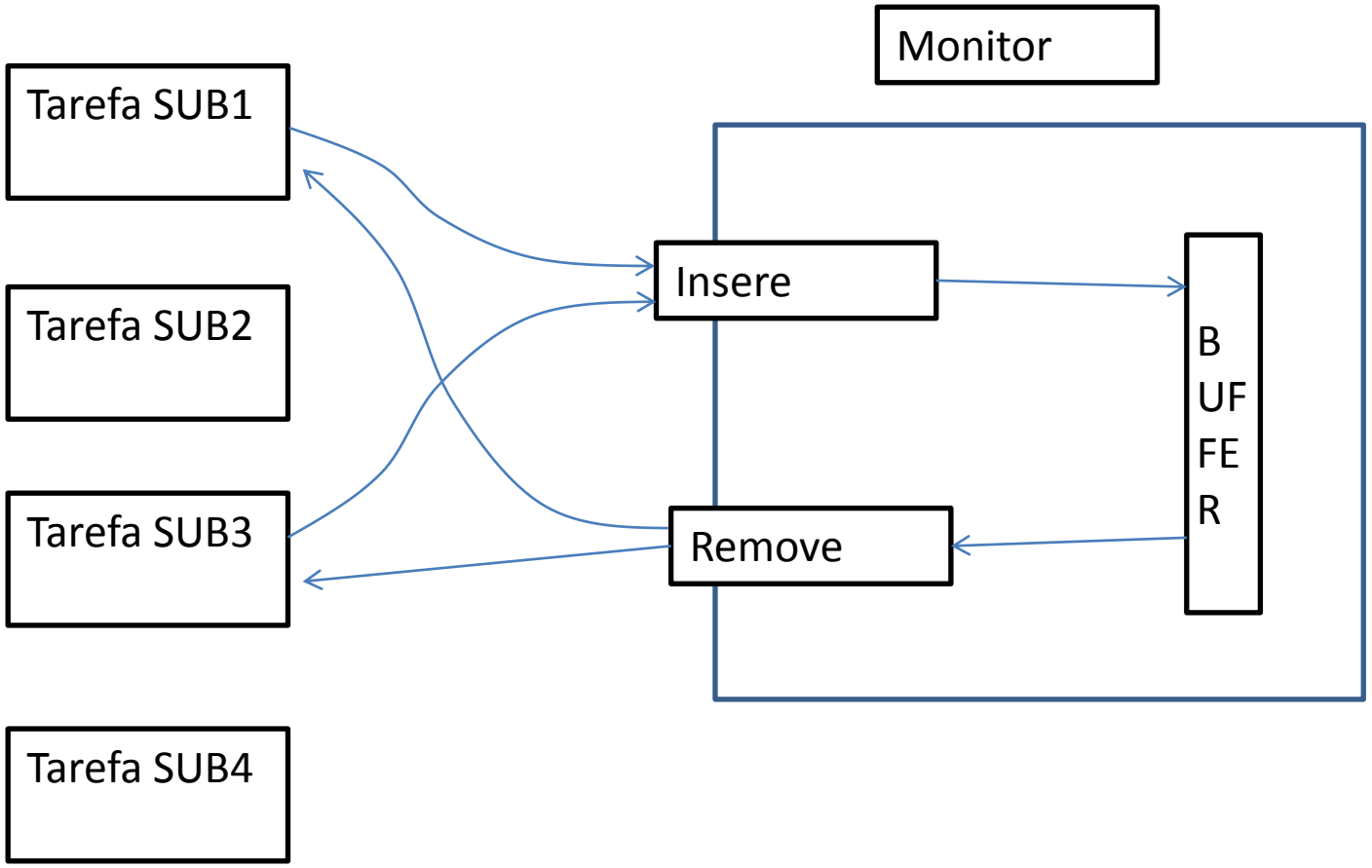
- Mecanismo que encapsula estruturas de dados compartilhadas e suas operações
- Estruturas de dados compartilhadas são definidas como tipos abstratos de dados
- Pode implementar sincronização de competição sem semáforos, transferindo a responsabilidade de sincronização para o sistema de execução
- Procedimentos dentro do monitor não podem ser chamados simultaneamente por mais de uma unidade

Monitores

```
Monitor <nome-do-monitor> {
        // declarações de variável
        p1(...) {
        ...
        }
        p2(...) {
        ...
        }
}
```

- No exemplo é mostrada a definição de dois procedimentos p1 e p2
  - Eles possuem acesso exclusivo

- Sincronização em monitores:
- Dado compartilhado fica no monitor
- Acesso mutuamente exclusivo não precisa ser controlado pelo programador
- Chamadas aos procedimentos do monitor são implicitamente colocadas em fila se o monitor estiver ocupado no momento da chamada


Monitor
Tarefa SUB1
Tarefa SUB2
Tarefa SUB3
Tarefa SUB4
Insere
Remove
B
UF
FE
R