# Sistemas Operacionais

- Sincronização de Processos -

# Concorrência

- Tópicos principais no projeto de sistemas operacionais:
  - Multiprogramação (no que vamos focar)
    - gerenciamento de vários processos considerando um processador
  - Multiprocessamento (no que vamos focar)
    - gerenciamento de vários processos considerando vários processadores em uma máquina
  - Processamento distribuído
    - Devido a utilização de cluster de processadores

# Concorrência

- Concorrência é essencial e deve ser tratado pelo SO
- Inclui:
  - Comunicação entre processos
  - Compartilhamento e competição de recursos
    - Memória, arquivos, acesso a E/S
  - Sincronização de atividades por parte de vários processos
  - Alocação de CPU a processos

# Princípios de Concorrência

- Em um ambiente com um processador, podemos ter:
  - Intercalação de execução de trechos de código de dois (ou mais) processos com
    - Tratamento de interrupção
    - Escalonador do SO
- Se existirem mais de um processador, a execução de trechos de código podem ser sobrepostas
  - Mas não elimina a necessidade de interrupção e escalonamento

# Princípios de Concorrência

- Intercalação e sobreposição podem ser vistos como dois exemplos de processamento concorrente
- Os seguintes problemas podem surgir:
  - Compartilhamento de recursos globais (ex, variáveis globais)
  - O escalonamento ótimo de recursos aos processos
    - Processo *A* pede alocação de um canal
    - Processo *A* é interrompido
    - O canal vai ficar alocado para *A*?
  - É difícil prever o tempo de execução, e o que vai ser alocado/ executado antes de executar um programa

# Exemplo

- Seja o seguinte programa

```
void echo(){
    chin = getchar();
    chout = chin;
    putchar( chout );
}
```

- O usuário submete duas aplicações que utilizam o mesmo teclado do computador

# Exemplo

- Seja o seguinte programa

```
void echo(){
    chin = getchar();
    chout = chin;
    putchar( chout );
}
```

- O SO pode usar o mesmo `echo()` compartilhado pelas aplicações em uma área de memória global
- Vantagem
  - economia de espaço de memória
  - Iteração entre processos

# Exemplo

- Seja o seguinte programa

```
void echo(){
    chin = getchar();
    chout = chin;
    putchar( chout );
}
```

- O SO pode usar o mesmo `echo()` compartilhado pelas aplicações em uma área de memória global
- Desvantagem: compartilhamento deve ser cuidadoso.

# Exemplo

- Seja o seguinte programa

```
void echo(){
    chin = getchar();
    chout = chin;
    putchar( chout );
}
```

- P1 chama `echo()`
  - É interrompido depois do `getchar()`, quando `chin` recebe `X`
- P2 chama `echo()`
  - Tudo é executado, e o valor `Y` é mostrado
- P1 será finalizado, mas o valor `X` foi perdido
  - O que vai ser mostrado na tela é o `Y` outra vez

# Exemplo

- Seja o seguinte programa

```
void echo(){
    chin = getchar();
    chout = chin;
    putchar( chout );
}
```

- O problema foi o compartilhamento de uma variável **`chin`**
- Uma solução:
  - P1 chama `echo()`
  - É interrompido depois do `getchar()`, quando `chin` recebe `X`
  - P2 é bloqueado, pois P1 está ainda executando `echo()` e não pode ser liberado devido a variável compartilhada

# Exemplo

```
P1
void echo(){
    .
    chin = getchar();
    .
    chout =chin;
    putchar(chout);
    .
    .
}
```

```
P2
void echo(){
.
.
    chin = getchar();
    chout =chin;
    .
    putchar(chout);
    .
}
```

- O problema pode aparecer tanto na existência de um ou mais processadores
- `P1` também vai ter seu valor `X` perdido

# Interação entre processos

- Processos não estão cientes um do outro
  - Exemplo clássico são processos em ambientes multiprogramáveis
    - Podem ser dependentes, mas podem competir por recursos
    - Tempo de execução de um afeta o outro

*Problemas deste tipo de concorrência*

- Exclusão mutua
- Deadlock
- starvation

# Interação entre processos

- Processos tem uma noção do outro (compartilham objetos)
  - Há uma cooperação devido ao compartilhamento
  - Resultado de um processo pode depender do outro
  - Tempo de execução de um afeta o outro

*Problemas deste tipo de concorrência*

- Exclusão mutua
- Deadlock
- Starvation
- Coerência de dados

# Interação entre processos

- Processos se comunicam (ciente um do outro)
  - Há uma cooperação através de comunicação
  - Resultado de um processo pode depender do outro
  - Tempo de execução de um afeta o outro

*Problemas deste tipo de concorrência*

- Deadlock
- Starvation

# Competição entre processos e recursos

- *N* processos competindo para utilizar os mesmos recursos, por exemplo, um dado de memória
  - Pode ser memória, dispositivos de I/O, etc
- Os processos não estão cientes da existência dos outros e são afetados pela respectivas execuções devido ao recurso compartilhado
  - Ao usar o recurso, o processo deveria deixar o recurso em estado que não afete a execução de outro recurso
  - Os recursos também não se comunicam

# Competição entre processos e recursos

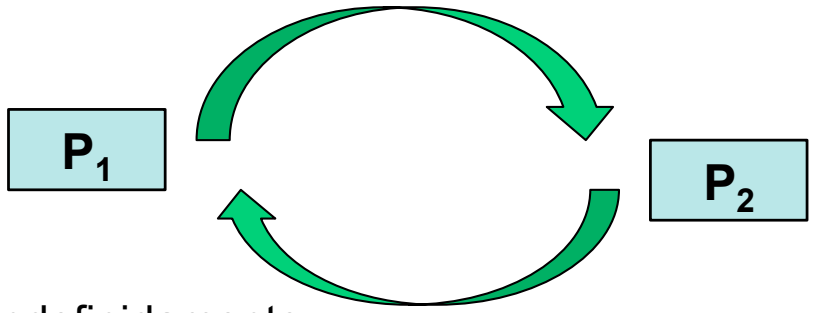
## Problemas que podem surgir

- Necessidade de exclusão mútua
  - Exemplo: uso de impressora – não pode ser compartilhada
- Esse recurso é um recurso crítico e sua utilização é sessão ou região crítica
  - Cada processo tem um segmento de código onde é feito o acesso a este dado ou recurso compartilhado
- O problema é garantir que quando um processo executa a sua região crítica, nenhum outro processo pode acessar a sua região crítica

# Competição entre processos e recursos

## Problemas que podem surgir

- Uma coisa leva a outra: assegurar exclusão mútua pode levar a outros problemas se não for devidamente realizada
- *Deadlock*:
  - **$P_1$** e **$P_2$** precisam cada de **$R_1$** e **$R_2$**
  - **$P_1$** consegue **$R_1$**
  - **$P_2$** consegue **$R_2$**

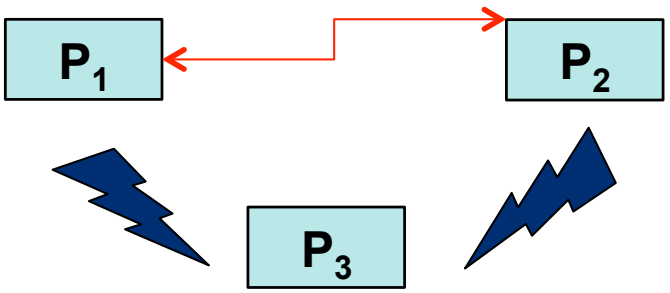



  - Um fica esperando pelo outro indefinidamente

# Competição entre processos e recursos

## Problemas que podem surgir

- Uma coisa leva a outra: assegurar exclusão mútua pode levar a outros problemas se não for devidamente realizada
- *Starvation*
  - **$P_1$** , **$P_2$** e **$P_3$** precisam de **R**
  - SO garante para **$P_1$** e **$P_2$** , mas **$P_3$** nunca consegue



# Competição entre processos e recursos

## Problemas que podem surgir

- O controle pela competição de recursos fatalmente envolve o SO
  - Deve haver mecanismos para assegurar exclusão mútua sem causa de problemas

# Região Crítica

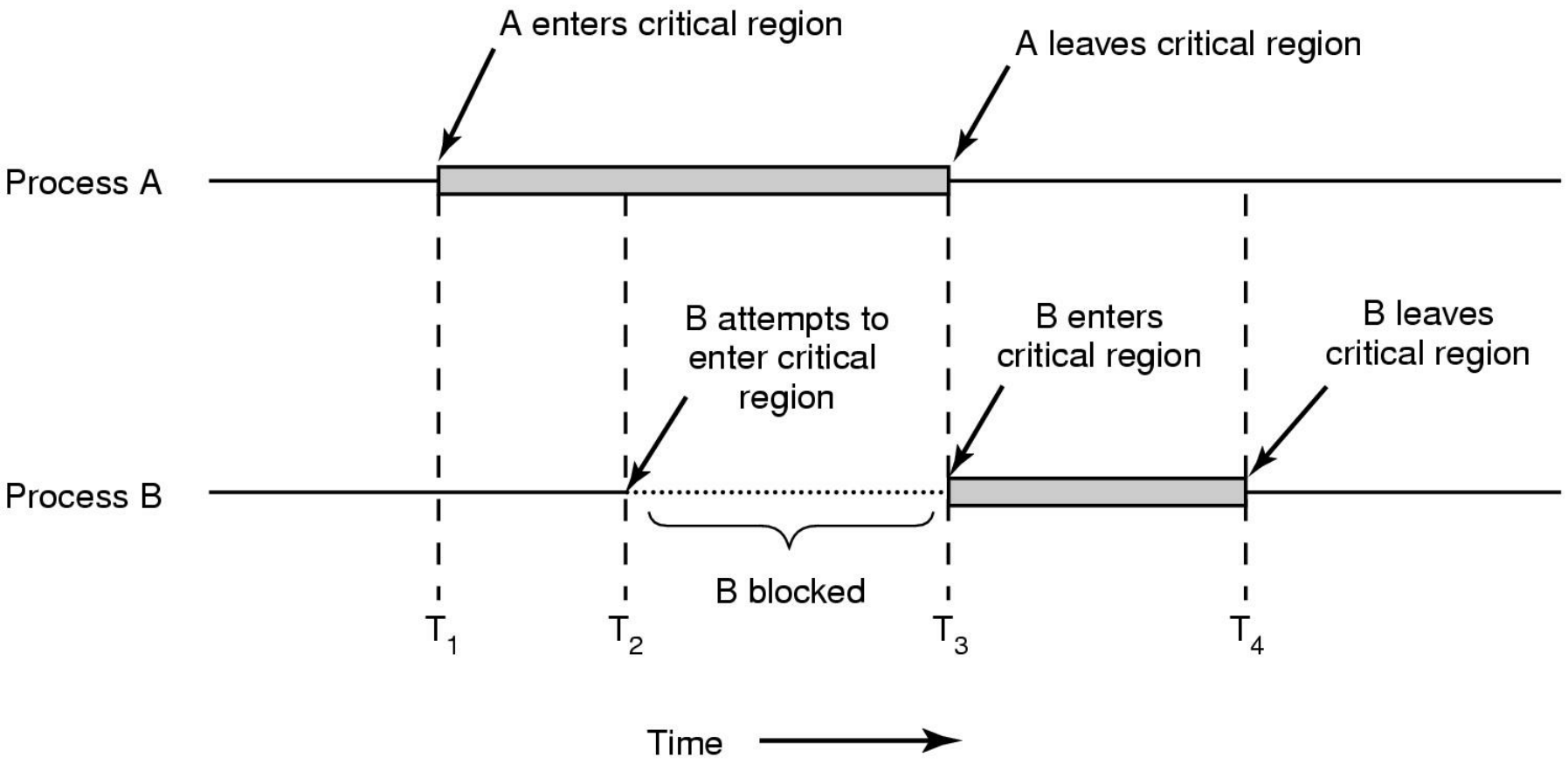
- Dois processos não podem executar em suas regiões críticas ao mesmo tempo
- É necessário um protocolo de cooperação
- Cada processo precisa “solicitar” permissão para entrar em sua região crítica
- Estrutura geral de um processo

```
while (true) {
    /* código que precede seção crítica */
    Seção de entrada
    Seção Crítica
    Seção de Saída
    /* código pós-seção crítica */
}
```

# Região Crítica

- Para solucionar o problema das regiões críticas alguns requisitos precisam ser satisfeitos:
  - Exclusão Mútua: Se um processo $\mathbf{P_i}$ está executando sua região crítica nenhum outro poderá executar a sua região crítica
  - Progresso: Nenhum processo fora de sua região crítica pode bloquear outro processo
  - Espera Limitada: Um processo não pode esperar indefinidamente para entrar em sua região crítica

# Região Crítica



# Cooperação entre processos através de compartilhamento

- Processos podem compartilhar área de dados, arquivos e até base de dados
  - Mecanismos de controle devem manter a integridade dos dados
- Vários processos podem compartilhar variáveis
  - A escrita deve ser feita de modo exclusivo
  - Coerência de dados deve ser mantida – mais um problema a ser controlado
  - Suponha que os dos dados a e b tenham que se manter iguais (a = b)

```
P1:
        a = a +1; b = b + 1;
P2:
    b = b  * 2; a = a * 2;
```

# Cooperação entre processos através de compartilhamento

- Suponha a ordem de execução (entre processos) e a = b = 1 inicialmente

```
a = a +1;    (P1)
b = b  * 2;  (P2)
b = b + 1;   (P1)
a = a * 2;   (P2)
```

*Final*: a = 4 e b = 3

- A condição não se mantém – região crítica deve ser definida e exclusão mútua mantida

# Cooperação entre processos através de comunicação

- Processos podem estar isolados totalmente uns dos outros
  - exclusão mútua não ocorrerá
- No entanto, se trocam mensagens
  - Deadlock e starvation podem ocorrer

# Exclusão mútua - abordagens

- Pode-se deixar a responsabilidade de controle para os processos que executam concorrentemente, sem suporte de linguagem de programação ou SO: suporte por software
  - Tendência a ocorrer alto overhead e erros
- A nível de instrução de máquina – suporte por hardware
  - Overhead mínimo, mas pode não ser uma solução de uso geral
- Suporte do SO e linguagem de programação
  - Maior versatilidade

# Exclusão mútua – suporte de hardware

- Em um ambiente de um processador
  - Processos podem se intercalar, e não se sobrepor
  - Processos só pedem processador
    - quando invocam o SO
    - Quando são interrompidos
  - Então, não devem ser interrompidos durante o acesso a região crítica

# Exclusão mútua – suporte de hardware

- Desabilitar Interrupções

```
Desabilita Interrupções
Região Crítica
Habilita Interrupções
```

  - Perigoso e pode não ser produtivo para multiprogramação deixar processos do usuário desabilitar interrupção
    - Se o processo não as reabilita o funcionamento do sistema está comprometido
    - Sendo as interrupções desabilitadas, o processo executa (no caso, na região crítica) e não vai perder CPU
      - neste caso, exclusão mútua é garantida à região crítica
      - CPU não pode executar outros processos (intercalação)

# Instrução Especiais de Máquina

- Ambiente de multiprocessador: vários processadores e compartilhamento de memória
- O acesso a um espaço compartilhado deve ser exclusivo
- Com isso em mente, projetistas de processador propuseram diferentes instruções que implementam leitura e escrita atômica de um endereço de memória

# Instrução Especiais de Máquina

- exemplo: compare&swap
  - testa o valor corrente de uma posição de memória
    - se é igual a um valor passado, atualiza com novo valor, senão, nada muda

```
while (true){
    while (compare&swap (bolt, 0, 1)
        RC;
    bolt = 0;
    ....
}
```

  - É uma instrução atômica – a instrução não pode ser interrompida

# Instrução Especiais de Máquina

- exemplo: compare&swap
  - testa o valor corrente de uma posição de memória
    - se é igual a um valor passado, atualiza com novo valor, senão, nada muda

```
while (true){
    while (compare&swap (bolt, 0, 1);
    RC;
    bolt = 0;
    ....
}
```

```
if (bolt ==0) bolt = 1;
return bolt;
```

# Instrução Especiais de Máquina

- exemplo: compare&swap
  - testa o valor corrente de uma posição de memória
    - se é igual a um valor passado, atualiza com novo valor, senão, nada muda

```
while (true){
    while (compare&swap (bolt, 0, 1);
    RC;
    bolt = 0;
    ....
}
```

Fica no `while` enquanto `bolt` não tiver valor zero!

# Instrução Especiais de Máquina

- Vantagens da abordagem por hardware
  - Qualquer número de processos em uma máquina uniprocessada ou multiprocessada pode executar a instrução
  - Pode ser usada para dar suporte a várias RCs

# Instrução Especiais de Máquina

- Desvantagens da abordagem por hardware
  - Somente um processo consegue entrar na RC – os outros ficam em busy&wait
  - Starvation pode acontecer – a seleção de um dos vários processos que estão tentando entrar na sessão crítica é arbitrária
  - *Deadlock*: certas situações podem levar

# Região Crítica

- Soluções por hardware
  - Depende da máquina utilizada
  - Tem menor overhead
  - Não são consideradas soluções generalizadas
- Soluções por software
  - Fica a cargo do programador
  - Fatalmente leva a erros

# Como resolver?

- Tanto solução por hardware quanto por software podem levar ao problema de espera ocupada
  - O processo “bloqueado” consome tempo de CPU desnecessariamente
- Solução:
  - Introduzir comandos que permitam que um processo seja colocado em estado de espera quando ele não puder acessar a sua região crítica
    - O processo fica em estado de espera até que outro processo o libere
    - Vamos olhar mecanismos que não consuma CPU

# Semáforos

- Mecanismos de SO e linguagens de programação que serão utilizados para aumentar concorrência
  - tornar a CPU não ociosa

# Semáforos

- Um semáforo é uma variável inteira não negativa, manipulada através:
  - Inicialização
  - Decremento *semWait()* / *acquire*() / *P*() / *Down*() / *wait*() / *lock*()
    - pode bloquear um processo
  - Incremento *segSignal()* / *relase*() / *V*() / *Up*() / *signal*() / *unlock*()
    - pode desbloquear um processo
- As modificações feitas no valor do semáforo são atômicas e não podem ser interrompidas

# Semáforos

- No caso da exclusão mútua as instruções *semWait()* e *smSignal()* funcionam como protocolos de entrada e saída das regiões críticas.
  - *Decremento* é executado quando o processo deseja entrar na região crítica.
    - Decrementa o semáforo de 1
    - bloqueia outros processos
  - *Incremento* é executado quando o processo sai da sua região crítica.
    - Incrementa o semáforo de 1
    - Desbloqueia processos

# Semáforos

- Um semáforo fica associado a um recurso compartilhado, indicando se ele está sendo usado
- Se o valor do semáforo é maior do que zero, então existe recurso compartilhado disponível
- Se o valor do semáforo é zero, então o recurso está sendo usado

```
semWait(s)
    s.count --;
    if (s.count < 0)
        bloqueia processo;
```

```
semSignal(s)
    s.count++;
    if (s.count <= 0)
        desbloqueia processo;
```

# Semáforos – pontos interessantes

- Em geral, não se sabe antes de um decremento, se um processo ficará bloqueado ou não
- Depois de um incremento feito por um processo $P_i$, um outro processo $P_j$ pode se desbloqueado
  - os dois processos $P_i$ e $P_j$ começam a ser executados concorrentemente
  - Não se sabe ao certo qual ganhará uma única CPU
  - Mas note, $P_i$ já passou pela RC
- Ao incrementar um semáforo (sinalizar), não se sabe a priori se há outros processos bloqueados

# Semáforos binários

- Mais restritos com seus valores – 0 e 1
- Se o valor do semáforo é maior do que zero, então existe recurso compartilhado disponível
- Se o valor do semáforo é zero, então o recurso está sendo usado

```
semWaitB(S)
    if (S != 0)
        bloqueia processo
    else
        S = 0;
```

```
semSignalB(S)
    if (S.queue != "empty")
        libera processo
    else
        S = 1;
```

# Semáforos

*mutex*

- É um semáforo binário
- Diferença chave
  - um processo que seta o valor para zero do *mutex*, dá um *lock*, é o processo que deve dar *unlock* no mutex

Semáforo forte X fraco

- Se a fila de processos bloqueados tem ou não tem política FIFO

# Semáforos

- Processos A, B e C dependem do resultado de D

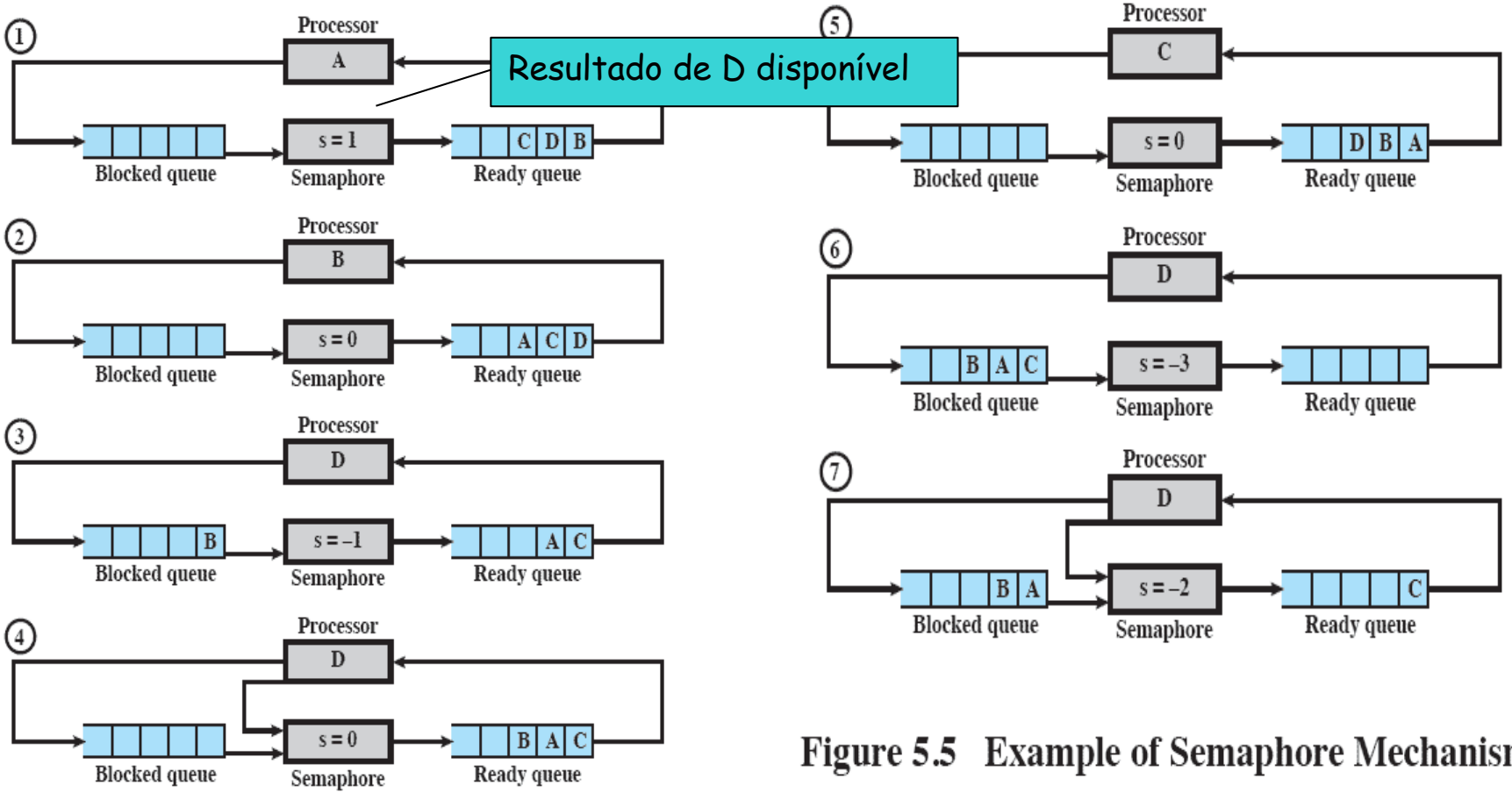



**Figure 5.5 Example of Semaphore Mechanism**

# Semáforos

- Processos A, B e C dependem do resultado de D

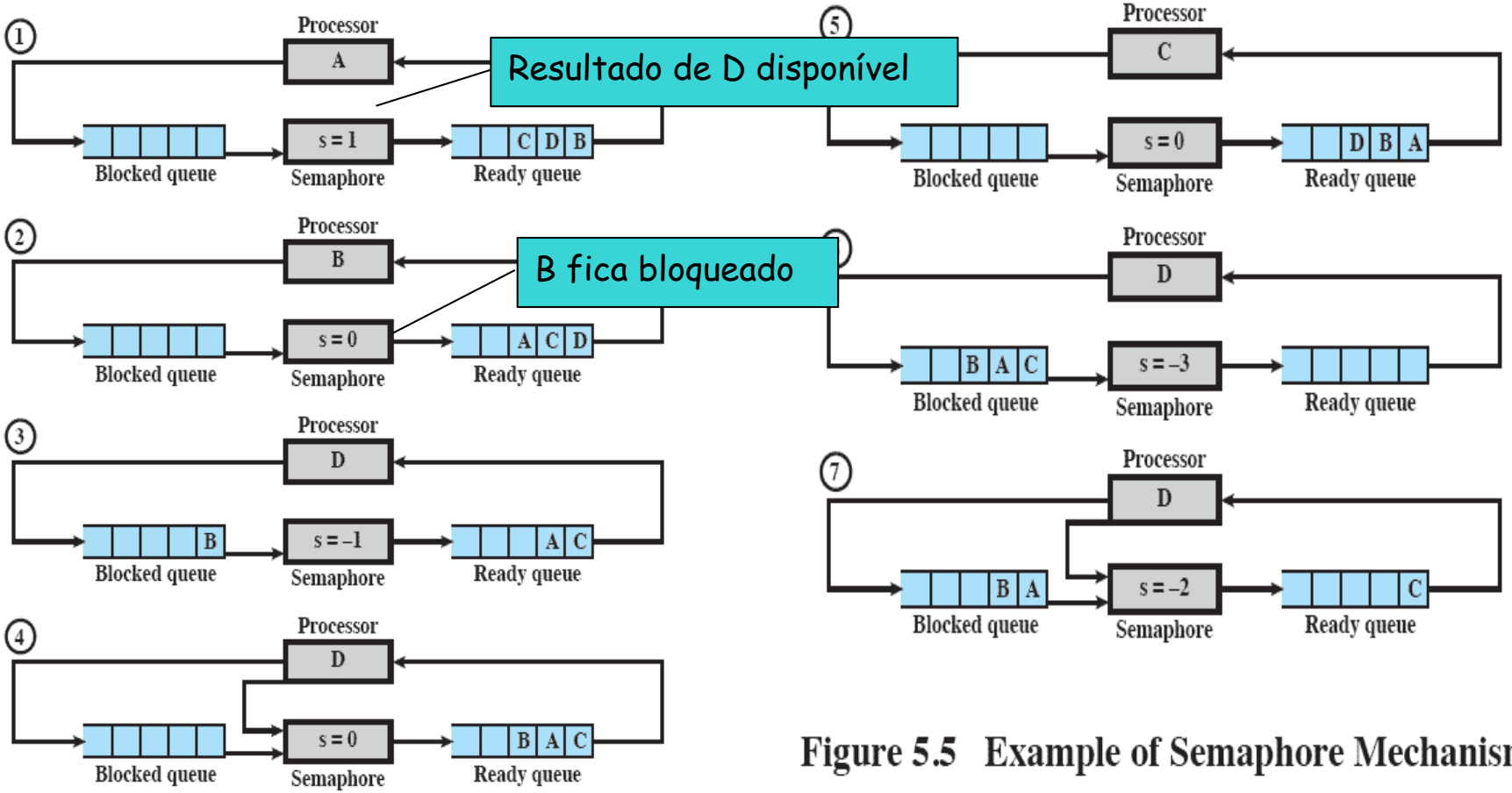



**Figure 5.5 Example of Semaphore Mechanism**

# Semáforos

- Processos A, B e C dependem do resultado de D

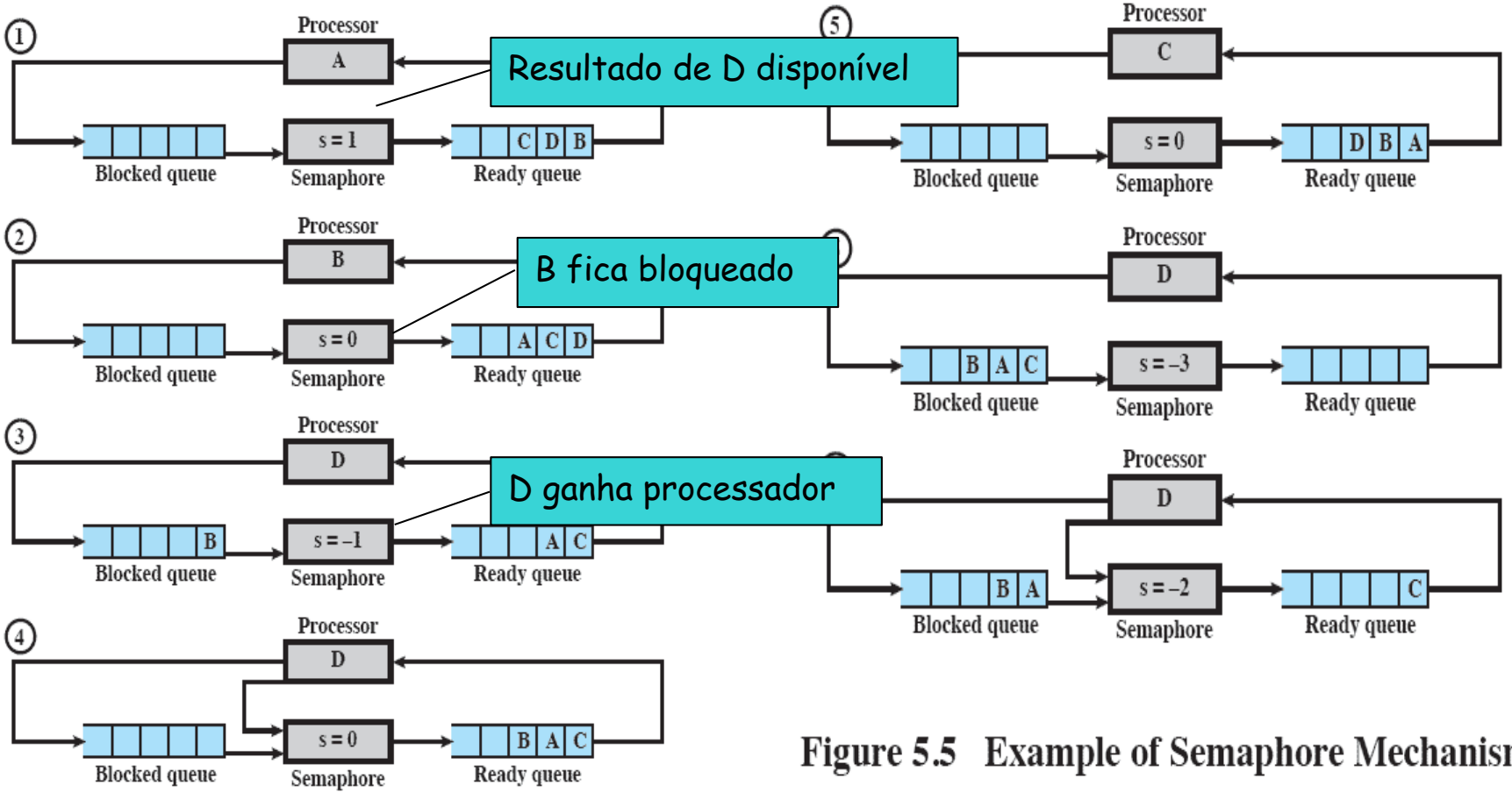



**Figure 5.5 Example of Semaphore Mechanism**

# Semáforos

- Processos A, B e C dependem do resultado de D

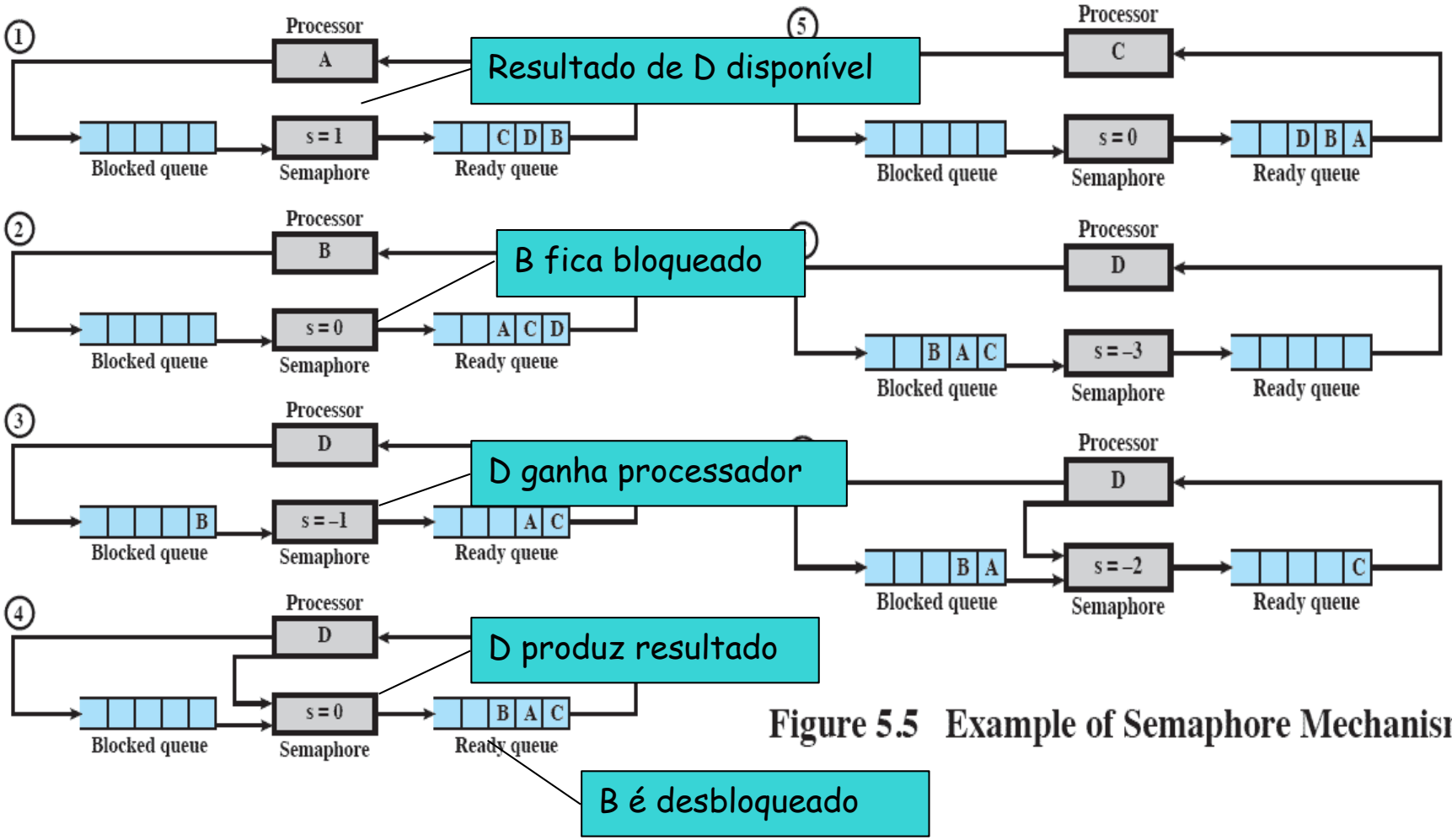



**Figure 5.5 Example of Semaphore Mechanism**

# Semáforos

- Processos A, B e C dependem do resultado de D

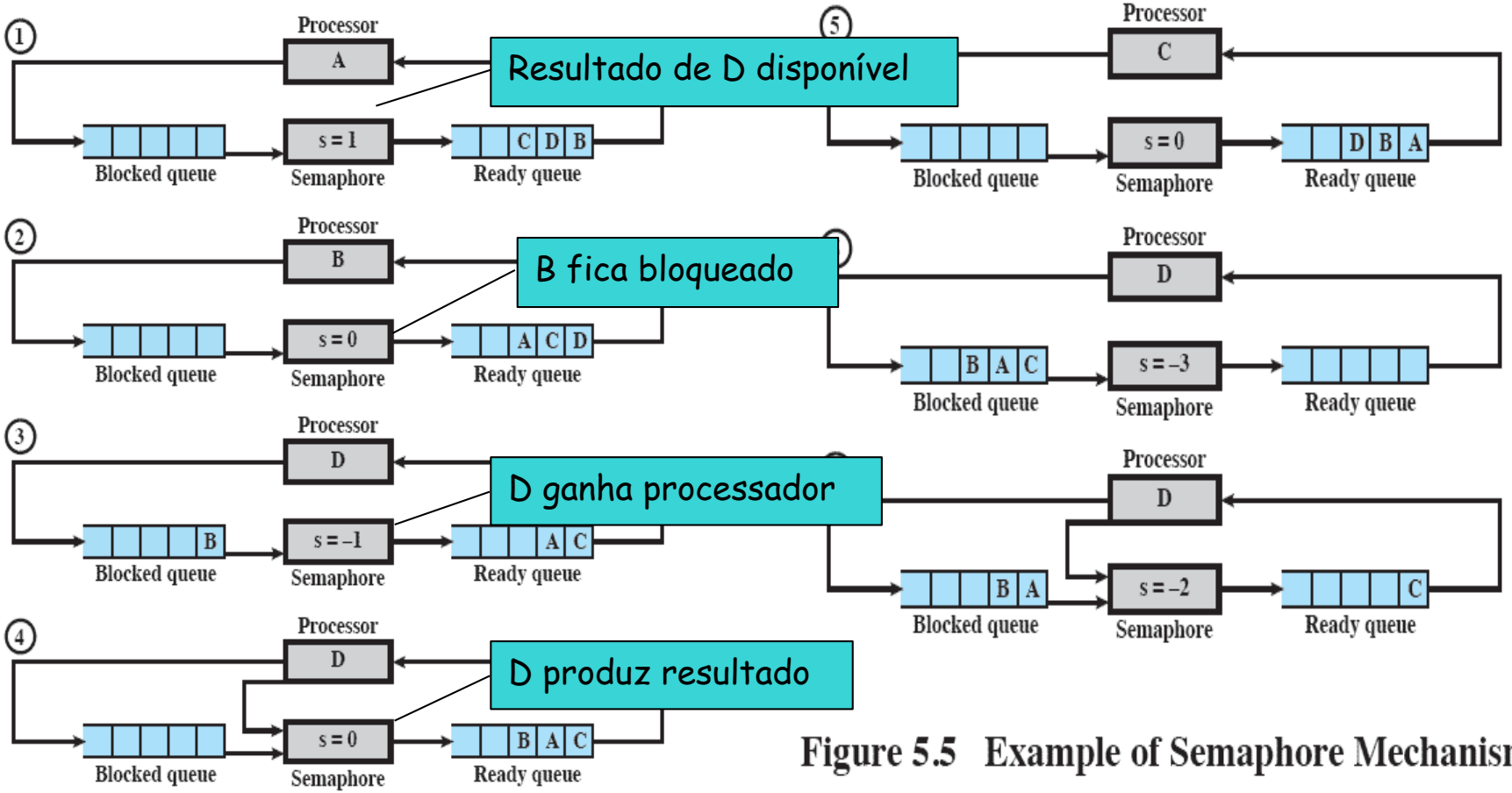



**Figure 5.5 Example of Semaphore Mechanism**

# Semáforos

- Processos A, B e C dependem do resultado de D

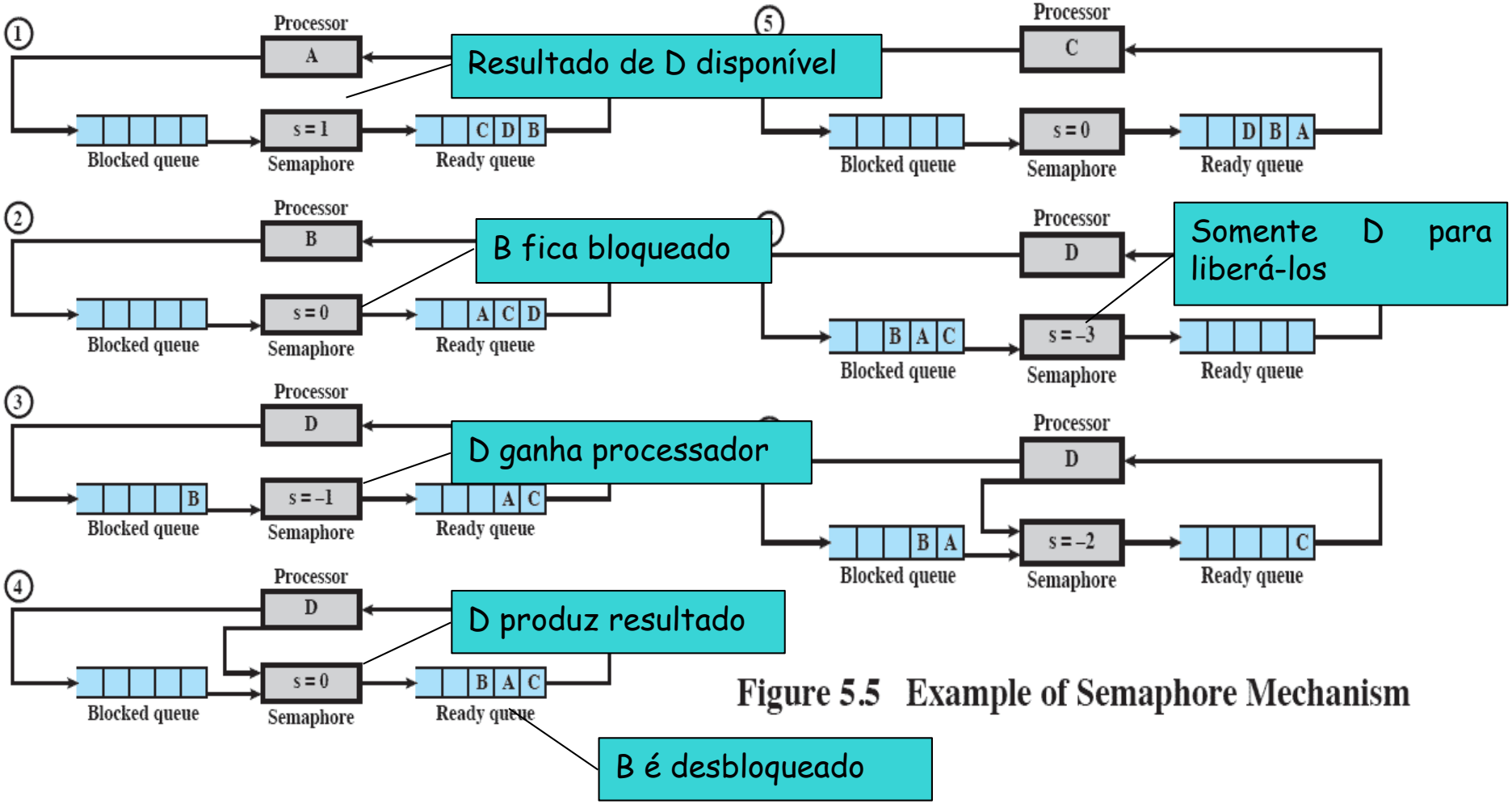



**Figure 5.5 Example of Semaphore Mechanism**

# Semáforos para exclusão mútua

- Para exclusão mútua é usado um semáforo binário

```
        P
Down (mutex);
Região Crítica;
Up (mutex);
```

- Semáforos também são usados para implementar a sincronização entre os processos
- O uso de semáforos exige muito cuidado do programador
  - Os comandos *down* e *up* podem estar espalhados em um programa sendo difícil visualizar o efeito destas operações

# Dados compartilhados protegidos por semáforo

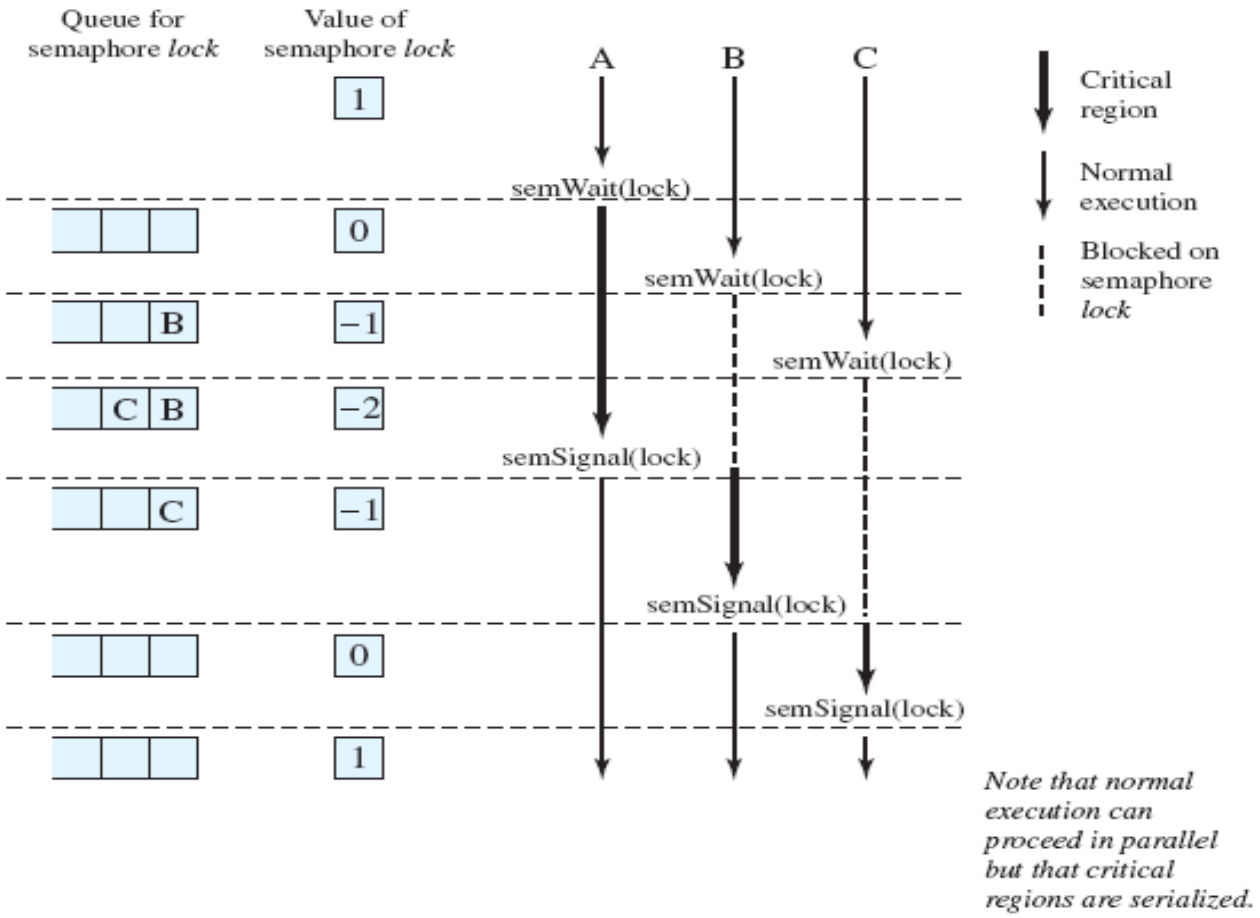



**Figure 5.7** **Processes Accessing Shared Data Protected by a Semaphore**

# Um Problema Clássico de Sincronização

- Produtor/Consumidor

# Produtor/Consumidor

- Processo produz informações que são gravadas em um buffer de tamanho limitado
- As informações são consumidas por um processo consumidor
- O produtor pode produzir um item enquanto o consumidor consome outro
  - Mas somente um consumidor e um produtor podem exclusivamente acessar o buffer compartilhado

# Produtor/Consumidor

- O produtor e o consumidor devem estar sincronizados
  - O produtor não pode escrever no buffer cheio
  - O consumidor não pode consumir informações de um buffer vazio

# Produtor/Consumidor

- Exemplo de uma aplicação "Produtor/Consumidor”:
  - um thread (o produtor) grava dados em um arquivo enquanto outro thread (o consumidor) lê dados do *mesmo* arquivo
- Outro exemplo:
  - enquanto você digita no teclado/mouse, o produtor coloca eventos de mouse em uma fila de eventos e o consumidor lê os eventos da *mesma* fila para tratá-los
- Em ambos os casos, temos threads que compartilham um recurso comum

# Produtor/Consumidor – buffer limitado

- Semáforo binário *s* para exclusão mútua (tipo *mutex*)
  - O *buffer* a ter itens produzidos e consumidos tem tamanho limitado
  - Tratado como um *array* circular

# Produtor/Consumidor – buffer limitado

- Semáforos n(*full)* e *e (empty)*
  - *n* conta os espaços cheios no *buffer*
    - Se *n* igual a zero, então o consumidor deve ser bloqueado
  - *e* conta os espaços vazios no *buffer*
    - Se *e* igual a zero, então o produtor deve ser bloqueado
  - *s* é um *mutex* (semáforo binário)
    - para controle da região crítica, no caso, o *buffer*

# Produtor/Consumidor – buffer limitado

| Block on: | Unblock on: |
|---|---|
| Producer: insert in full buffer | Consumer: item inserted |
| Consumer: remove from empty buffer | Producer: item removed |

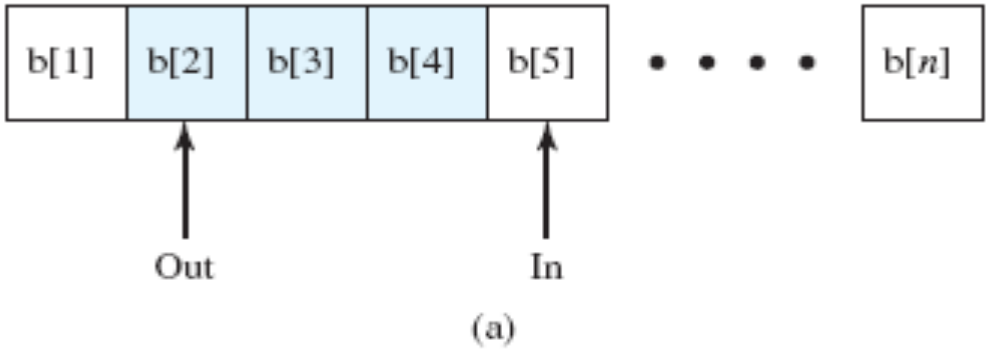



**Figure 5.12** Finite Circular Buffer for the Producer/Consumer Problem

# Produtor/Consumidor

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

| |
|---|
| Gera item |
| Mais uma posição do buffer será usada, logo, decrementa |
| Entra na RC |
| Efetivamente, acessa o buffer |
| Deixa a RC |
| Incrementa o contador de itens no buffer |

| |
|---|
| Decrementa n, pois item será consumido |
| Vai entrar em RC |
| Consome um item |
| Libera RC |
| Incrementa o contador de espaços vazios |
| Faz algo com o item retirado do buffer compartilhado |

# Exercícios

```
/* program boundedbuffer */
const int sizeofbuffer  = /* buffer size */
semaphore s = 1, n = 0, e =sizeofbuffer;
void producer()
{
      while (true){
            produce();
            semWait(e);
            semWait(s);
            append();
            semSignal(s);
            semSignal(n);
      }
}
```

```
void consumer()
{
      while (true){
            semWait(n);
            semWait(s);
            take();
            semSignal(s);
            semSignal(e);
      }

}
```

Would the meaning of the program change if the following were interchanged?

a) semWait(e); semWait(s)
b) semSignal(s); semSignal(n)
c) semWait(n); semWait(s)
d) semSignal(s); semSignal(e)

# Problema Clássico de Sincronização

- Leitores e Escritores – um outro problema clássico

# Leitores e Escritores

- Existem áreas de dados compartilhadas
  - Área pode ser um arquivo ou bloco da memória principal
- Leitores: processos que apenas lêem dados destas áreas
- Escritores: processos que apenas escrevem dados nestas áreas
- Condições:
  - Qualquer número de leitores pode ler ao mesmo tempo
  - Apenas um escritor pode acessar a área por vez
  - Se um escritor está escrevendo, nenhum leitor poderá utilizá-lo

# Leitores e Escritores

- Então exclusão mútua – somente quando escritores estiverem acessando a área
- Diferenças com o produtor/consumidor:
  - Leitores só lêem e escritores só escrevem (a própria operação difere)
- Exemplo: catálogo de biblioteca
  - Vários podem consultar
  - Só uma bibliotecária pode atualizar

# Leitores e Escritores

## Versão 1: prioridade aos leitores

- Semáforo *x* para acesso a *readcount*
- Se é o primeiro leitor a tentar acessar RC, tem prioridade e o escritor não pode acessar
- Se um leitor acessa RC, outros leitores podem acessar
- Para atualizar *readcount*
- Se não tem mais leitor, um escritor pode acessar RC

- Só um escritor pode acessar a RC, quando nenhum leitor estiver acessando

- Por que o semáforo x???

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
      semWait (x);
      readcount++;
      if (readcount == 1) semWait (wsem);
      semSignal (x);
      READUNIT();
      semWait (x);
      readcount--;
      if (readcount == 0) semSignal (wsem);
      semSignal (x);
    }
 }
void writer()
{
    while (true) {
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

# Leitores e Escritores

## Versão 1: prioridade aos leitores

- Semáforo *x* para acesso a *readcount*
- Se é o primeiro leitor a tentar acessar RC, tem prioridade e o escritor não pode acessar
- Se um leitor acessa RC, outros leitores podem acessar
- Para atualizar *readcount*
- Se não tem mais leitor, um escritor pode acessar RC

- Só um escritor pode acessar a RC, quando nenhum leitor estiver acessando

- Por que o semáforo *x*???

```
int readcount;
semaphore x=1; wsem=1;
void reader()
{       while (true){
            semWait (x);
            readcount++;
            if (readcount==1) semWait(wsem);
            semSignal (x);
            READUNIT();
            semWait (x);
            readcount--;
            if(readcount==0) semSignal (wsem);
            semSignal (x);
        }
}
void writer()
{       while (true) {
            semWait (wsem);
            WRITEUNIT();
            semSignal (wsem);
        }
}
void main{
{       readcount=0; parbegin (reader, writer);
}
```

# Leitores e Escritores

## Versão 2: prioridade ao escritor

- Semáforo x para acesso a *readcount*
- Semáforo *rsem*, inibe leitores a acessarem RC se um escritor estiver acessando
  - e inibe escritor, depois que o 1º leitor já tiver garantido RC
- Semáforo *y* para acesso a *writecount* por parte dos escritores
- Semáforo *z* para o controle dos vários leitores
  - Um leitor vai para a fila devido a *rsem*
  - Leitores adicionais vão para a fila devido a *z*
  - Necessário para que escritor não fique esperando uma fila longa de leitores devido somente a rsem, caso muitos leitores apareçam

# Leitores e Escritores

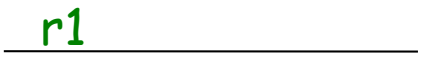
- quando RC são as variáveis
  - x → *readcount* (leitores)
  - *y* → *writecount* (escritores)
- a RC
  - *rsem* → escritor tem prioridade, mas um leitor pode acessar se chegar antes dele
  - *wsem* → outros escritores
  - *z* → se outros leitores

# Leitores e Escritores

writer(w1)

**rsem = 0**
**wsem = 0**

reader(r1) → bloqueado




w1 deixa a RC → fila bloqueado em rsem fica vazia

r1 acessa RC

write(w2) → bloqueado em rsem
reader (r2) → bloqueado em y

# Leitores e Escritores

```
/*program readersandwriters*/
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1) semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0) semSignal (wsem);
     semSignal (x);
    }
}
```

```
void writer ()
{
    while (true) {
     semWait (y);
          writecount++;
          if (writecount == 1) semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

# Leitores e Escritores

## Versão 2: prioridade ao escritor

- Se é o primeiro escritor a acessar, sinaliza para desabilitar outros leitores
- Outros escritores podem agora atualiza *writecount*
- Exclusão mútua para RC
- Escritor libera RC
- Exclusividade para atualizar *writecount*: menos um escritor tentando acesso
- Se não tiver mais escritor tentando acessar RC, então leitores podem acessá-lo
- Libera acesso ao *writecount*

```
void writer ()
{
    while (true) {
     semWait (y);
          writecount++;
          if (writecount == 1) semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

# Leitores e Escritores

## Versão 2: prioridade ao escritor

- Se é o primeiro escritor a acessar, sinaliza para desabilitar outros leitores
- Outros escritores podem agora atualiza *writecount*
- Exclusão mútua para RC
- Escritor libera RC
- Exclusividade para atualizar *writecount*: menos um escritor tentando acesso
- Se não tiver mais escritor tentando acessar RC, então leitores podem acessá-lo
- Libera acesso ao *writecount*

```
void writer()
{
   while (true){
      semWait (y);
      writecount++;
      if(writecount==1)semWait(rsem);
      semSignal (y);
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
      semWait  (y);
      writecount--;
      if(writecount==0)semSignal(rsem);
      semSignal (y);
   }
}
void main()
{
   readcount = writecount = 0;
   parbegin (reader, writer);
}
```

# Leitores e Escritores

```
int readcount, writecount;
semaphore x=1, y=1, z=1, wsem=1 rsem=1;
void reader()
{
   while (true){
      semWait (z);
      semWait(rsem);
      semWait (x);
      readcount++;
      if(readcount==1)semWait(wsem);
      semSignal (x);
      semSignal (rsem);
      semSignal(z)
      READUNIT();
      semWait  (x);
      readcount--;
      if(readcount==0)semSignal(wsem);
      semSignal (x);
   }
}
```

## Versão 2: prioridade ao escritor

- Controla número de leitores – só um leitor ficaria bloqueado em rsem devido a prioridade do escritor.
- Se não tiver um escritor, leitor não precisa estar enfileirado somente
- Exclusividade do acesso a *readcount*
- Se um leitor consegue acessar RC, pede exclusividade: escritor não pode acessar
- Libera acesso a *readcount* (outros leitores podem acessá-lo, mas não o RC)
- Já que um ganhou acesso a RC, outros leitores podem ganhar acesso
- outra vez, tem que atualizar o *readcount*, então precisa pedir acesso através de *x*

# Monitores

- Semáforos são ferramentas importantes para implementar exclusão mútua
  - Mas pode ser difícil utilizá-los corretamente em um programa
    - Vimos alguns exemplos de uso que não garantem exclusão mútua, sendo necessário adicionar outras variáveis de bloqueio
  - Estão sujeitos a erros de programação
    - Principalmente se necessário definir mais de uma RC
  - Não existe um controle formal de sua presença

**Como tornar o uso de semáforos mais fácil/amigável?**

# Monitores

- é um construtor de linguagem de programação que oferece a funcionalidade dos semáforos
  - maior facilidade de utilização por parte do programados
  - não tem em muitas linguagens de programação, mas em Java
- codificar as seções críticas como procedimentos do monitor
- tornar mais modular
  - quando um processo precisa referenciar dados compartilhados
    - chama um procedimento do monitor
  - Módulos de linguagens de programação que fornecem uma funcionalidade equivalente aos semáforo

# Monitores

- O monitor é um conjunto de procedimentos, variáveis e inicialização definidos dentro de um módulo
  - um processo entra no “monitor”, chamando um dos seus procedimentos
  - como se fosse uma classe
- A característica mais importante do monitor é a exclusão mútua automática entre os seus procedimentos
  - Basta codificar as regiões críticas como procedimentos do monitor e o compilador irá garantir a exclusão mútua
  - Como semáforos, só um processo pode estar executando no monitor de cada vez, sendo que outros processos que chamarem o monitor, ficarão bloqueados
    - Oferece exclusão mútua

# Monitores

- Variáveis compartilhadas podem ser protegidas, especificando-as através do monitor
- Um processo pode ser bloqueado quando tentar entrar no monitor, mas este está sendo utilizado por outro
- Quando um processo termina de usar o monitor, tem que liberá-lo

# Monitores

**monitor** ***monitor-name***

```
        {
                declaração de variáveis compartilhadas

                procedure P1 (…) {
                        . . .
                }
                procedure P2 (…) {
                        . . .
                }
                procedure Pn (…) {
                         . . .
                }

                {
                        código de inicialização
                }
        }
```

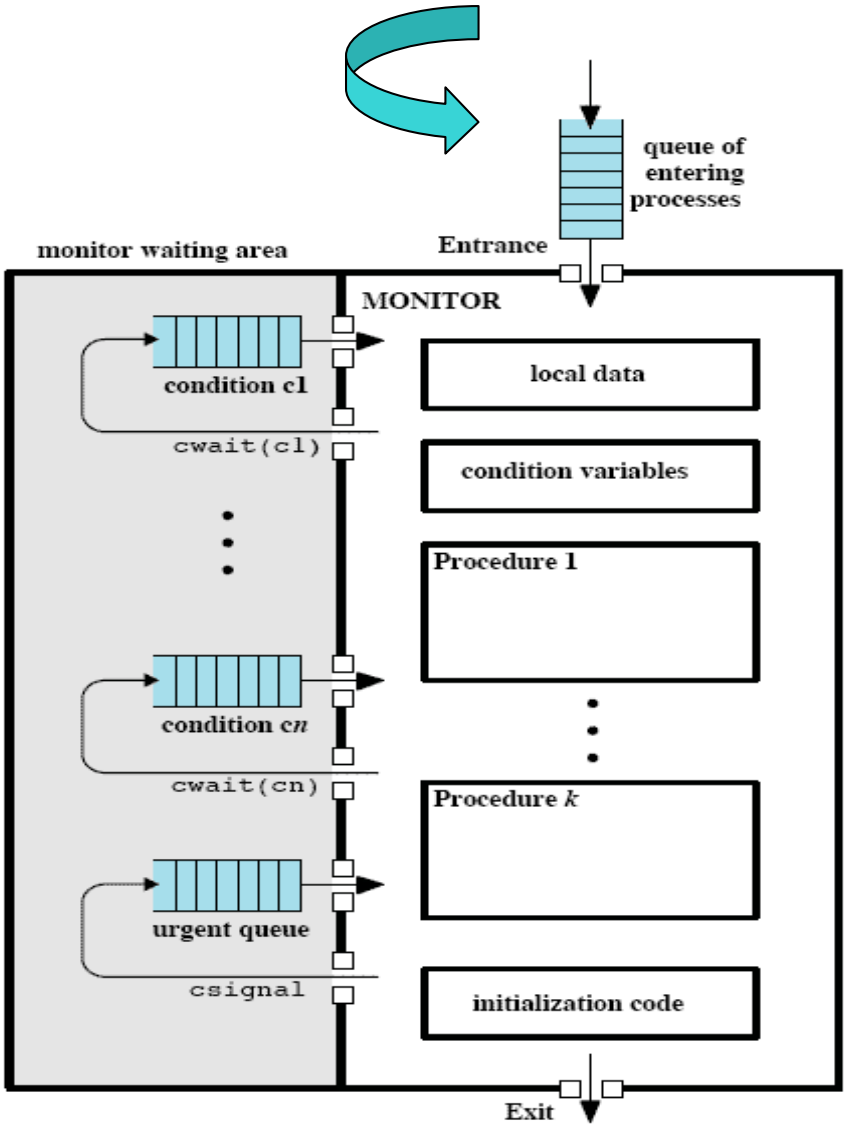
# Monitores

- Para implementar a sincronização é necessário utilizar variáveis de condição, só acessíveis dentro do monitor
- Variáveis de condição
  - são tipos de dados especiais dos monitores
  - são operadas por duas instruções *Wait* e *Signal*
- *Wait(C)*: suspende a execução do processo, colocando-o em estado de espera associado a condição C
- *Signal(C)*: permite que um processo bloqueado por *wait(C)* continue a sua execução.
  - Se existir mais de um processo bloqueado, apenas um é liberado
  - Se não existir nenhum processo bloqueado, não faz nada

# Monitores

- Um processo entra em um monitor, chamando qualquer um de seus procedimentos
- Um processo está em um monitor, quando está pronto ou executando
- Um ponto de entrada somente, para garantir exclusão mútua
  - Outros processos ficam em fila de bloqueados se tentarem entrar no monitor

ponto de único de entrada – garante exclusão mútua



# Monitores

- Produtor/Consumidor utilizando buffer de tamanho limitado
- Duas variáveis de condição:
  - essas variáveis são acessíveis somente dentro do monitor

*notfull* – TRUE se ainda há espaço

*notempty* – TRUE se tem caractere

```
void producer()
{
        char x;
        while (true){
                produce(x);
                append(x);
        }
}
void consumer()
{
        char x;
        while (true){
                take(x);
                consume(x);
        }
}
void main()
{
        parbegin (producer,
consumer);
}
```

# Monitores

```
monitor boundbuffer;                /*  modulo do monitor */
char buffer[N];                     /*  espaço para N itens */
int nextin, nextout;                /*  ponteiros para o buffer */
int count;                          /*  # itens no buffer  */
cond notfull, notempty;             /*  var. de condição p/ sincronização*/
void append (char x)
{
        if (count == N) cwait (notfull); /*  buffer cheio, evitar overflow */
        buffer[nextin] = x;         /* o processo pode executar no monitor */
        nextin = (nextin + 1) mod N;
        count++;                    /*  mais um item no buffer */
        csignal (notempty);         /*  consumidor em espera pode ser finalizado  */
}
void take (char x)
{
        if (count ==0) cwait(notempty); /*  buffer vazio, evite underflow*/
        x = buffer[nextout];
        nextout = (nextout + 1) mod N;
        count --;           /*  menos um item no buffer */
        csignal (notfull); /*  produtor em espera pode ser finalizado  */
}
```

# Monitores

```
/* inicializações no main */

{
        nextin = 0; nextout = 0; count = 0;
}
```

- Um processo entra em um monitor, chamando qualquer um de seus procedimentos
- Um ponto de entrada somente, para garantir exclusão mútua
  - Outros processos ficam em fila de bloqueados se tentarem entrar no monitor

# Monitores - curiosidade

- Uma linguagem atual que oferece o uso de monitores é Java
- cada objeto tem seu próprio monitor
- métodos são colocados em estado de monitor através da palavra chave *synchhronized*
- existe somente uma variável de condição assim, não precisa ser explicitamente especificada
- métodos que testam tais variáveis são: *wait*(), *notify*(), and *notifyAll*()
  - **wait( ) –** o respectivo thread vai para bloqueado (estado sleep) até que algum outro thread que entrou no monitor **notify( )**
  - **notify( )** – acorda o primeiro thread que chamou um **wait( )** no mesmo objeto
  - **notifyAll( )** – acorda todos os threads que chamaram **wait( )** no mesmo objeto, sendo que a thread de maior prioridade executa primeiro.

# Troca de Mensagens

- Quando é necessário trocar informações entre processos que não compartilham memória
- Usado para comunicação e sincronização
- Basicamente usa duas primitivas
  - *send*(destino, mensagem)
  - *receive*(origem, mensagem)
- Estas duas primitivas podem ser facilmente colocadas em bibliotecas
- Uma biblioteca de comunicação que se tornou padrão é MPI

# Troca de Mensagens

- Sincronização
  - Um processo receptor não pode receber uma mensagem até que esta tenha sido enviada
  - Deve se determinar o que acontece com um processo após executar um *send* ou *receive*
  - *Send* – quando um *send* é executado existe a possibilidade de bloquear ou não o processo até que a mensagem seja recebida no destino
  - *Receive* – quando o processo executa um *receive* existem duas possibilidades:
    - se a mensagem já foi enviada o processo a recebe e continua a sua execução
    - se a mensagem ainda não foi enviada (não foi executado um send() por alguém)
      - o processo é bloqueado até que a mensagem chegue ou
      - o processo continua a executar e abandona a tentativa de recebimento

# Troca de Mensagens

- *Send* e *Receive* podem ser bloqueantes ou não bloqueantes
  - O mais comum é *send* não bloqueante e *receive* bloqueante

# Troca de Mensagens

- Endereçamento Direto
  - O processo que envia ou recebe uma mensagem deve especificar a origem e o destino por endereçamento direto
    - Identificação do processo
  - O *sender* sempre especifica a id do recebedor
  - *Receiver*
    - pode especificar a id (comum em aplicações paralelas)
    - ou não: comum quando o recebedor oferece serviços

# Troca de Mensagens

- Endereçamento Direto - exemplo
  - Processo printer-server
    - o cliente que envia mensagem especifica o destinatário
    - o servidor, pode receber de qualquer processo

# Troca de Mensagens

- Endereçamento Indireto
  - As mensagens não são endereçadas diretamente entre processos origem e destino
  - As mensagens são enviadas para caixas postais – são filas/ buffers
    - mailboxes
    - armazenam as mensagens temporariamente

.

# Troca de Mensagens

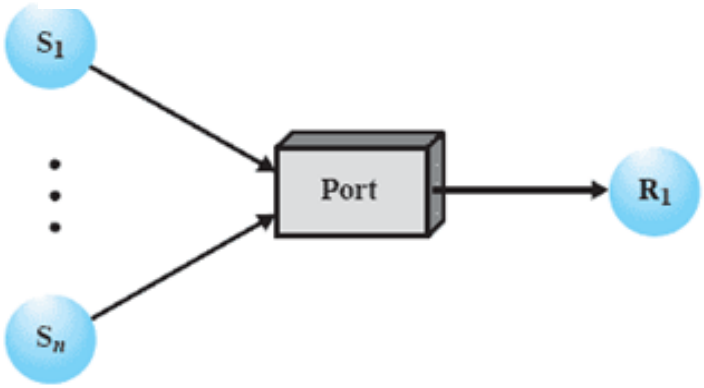
- Endereçamento Indireto
  - Desacopla processo que envia daquele que recebe:
    - Um para um (*one-to-one*)
      - um link privado é definido entre *sender* e *receiver*
    - Vários para um (*many-to-one*)
      - tipo cliente/servidor: um processo oferece um serviço para vários outros processos
      - neste caso o *mailbox* é referenciado como *port*
    - Um para vários (*one-to-many*)
      - um broadcast/difusão
    - Vários para vários (*many-to-many*)

.

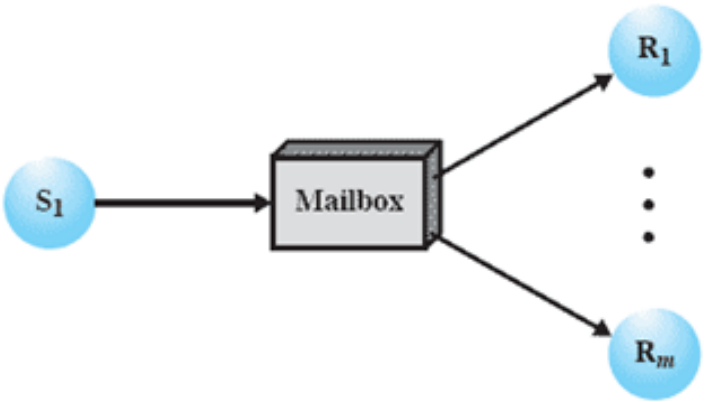
# Troca de Mensagens

- Vários para um (*many-to-one*) pode representar comunicação cliente servidor
- *Broadcast*: é comunicação um-para-vários


(a) One to one




(b) Many to one




(c) One to many

# Troca de Mensagens

- Endereçamento Indireto
  - Associação de processos a mailboxes pode ser
    - Estático
      - portas são normalmente associadas a processos estaticamente
      - Um-para-um geralmente é definido estaticamente
    - Dinâmico
      - Portas são geralmente criadas pelo processo recebedor
      - Mailboxes são geralmente criadas pelo SO

# Troca de Mensagens eExclusão Mútua

## Implementação de Exclusão Mútua com Processos

- processos compartilham um mailbox: *box*
- Inicialmente, *box* contem uma msg com conteúdo NULL
- Para entrar em região crítica:
  - processo tenta receber msg
  - Se box está vazio → processo fica bloqueado
  - Se tem uma msg, processo recebe msg e entra em RC


- Como ficaria um algoritmos?

# Leitores e Escritores

## Troca de mensagens

- A versão a seguir dá prioridade a escritores
- *Controller()* simula a área compartilhada:
  - A mensagem “OK” significa “pode usar a área compartilhada”
  - O controlador gerencia 3 mailboxes (uma para cada tipo de mensagem)
    - *finished*, *readrequest*, *writerequest*

- *count* controla o número máximo de leitores, que é inicializado por exemplo, com 100.
  - *count* > 0: então nenhum escritor fez pedido
  - *count = 0:* o escritor fez pedido – ele irá pedir para finalizar também
  - *Count < 0 escritor espera os leitores acabarem*
- Tipos de mensagens:
  - *finished*, *readrequest*, *writerequest*

# Leitores e Escritores

## Troca de mensagens

- *count:* para controlar exclusão mútua e dar prioridade aos escritores
- *count:* inicializado com um número maior que o número de leitores, por exemplo, com 100;

- *count* > 0
  - nenhum escritor fez pedido
  - Primeiro atende aos "*finished*"
  - Depois aos "*writerequest*"
  - E só depois aos "*readrequest*"
- *count = 0*
  - o escritor já fez pedido – bloqueia outros leitores
- *count < 0*
  - *escritor espera os leitores acabarem*

# Leitores e Escritores – troca de msg

```
void reader(int i)
{
        message rmsg;
        while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
        }
}
void writer(int j)
{
        message rmsg;
        while(true) {
         rmsg = j;
         send (writerequest, rmsg);
         receive (mbox[j], rmsg);
         WRITEUNIT ();
         rmsg = j;
         send (finished, rmsg);
         }
}
```

```
void controller()
{       while (true) {
          if (count > 0) {
                if (!empty (finished)) {
                   receive (finished, msg);
                   count++;
                } else if (!empty
(writerequest)) {
                   receive (writerequest, msg);
                   writer_id = msg.id;
                   count = count – 100;
                          } else if (!
empty (readrequest)) {
                   receive (readrequest, msg);
                   count--;
                   send (msg.id, "OK");
                          }
          }
          if (count == 0) {
                send (writer_id, "OK");
                receive (finished, msg);
                count = 100;
          }
          while (count < 0) {
                   receive (finished, msg);
                   count++;
          }
```

# Leitores e Escritores – troca de msg

```
void reader(int i)
{
        message rmsg;
        while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
        }
}
void writer(int j)
{
        message rmsg;
        while(true) {
         rmsg = j;
         send (writerequest, rmsg);
         receive (mbox[j], rmsg);
         WRITEUNIT ();
         rmsg = j;
         send (finished, rmsg);
         }
}
```

Pede acesso ao buffer

Quando possível, um ack é enviada via mbox[]

Quando possível, um ack é enviada via mbox[]

Para avisar a saida do RC

# Leitores e Escritores – troca de msg

```
void controller()
{       while (true) {
          if (count > 0) {
                if (!empty (finished)) {
                   receive (finished, msg);
                   count++;
                } else if (!empty (writerequest)) {
                   receive (writerequest, msg);
                   writer_id = msg.id;
                   count = count – 100;
                 } else if (!empty (readrequest)) {
                   receive (readrequest, msg);
                   count--;
                   send (msg.id, "OK");
                }
          }
          if (count == 0) {
                send (writer_id, "OK");
                receive (finished, msg);
                count = 100;
          }
          while (count < 0) {
                   receive (finished, msg);
                   count++;
          }
        }
}
```

# Qual o problema com a seguinte implementação de um escritor e vários leitores?

```
int readcount;              // inicializada com 0
Semaphore mutex, wrt;       // inicializada com 1
```

```
// Writer :
semWait(wrt);
/* Writing performed*/
semSignal(wrt);
```

```
// Readers :
semWait(mutex);
readcount := readcount + 1;
if readcount == 1 then semWait(wrt);
semSignal(mutex);
/*reading performed*/
semWait(mutex);
readcount := readcount - 1;
if readcount == 0 then semSignal
(wrt);
semSignal(mutex);
```

# Considere o seguinte programa

```
P1: {
    shared int x;
    x = 10;
    while (1) {
         x = x - 1;
         x = x + 1;
         If (x != 10)
            printf(”x=%d",x);
      }
}
```

```
P2: {
    shared int x;
    x = 10;
    while (1) {
         x = x - 1;
         x = x + 1;
         If (x != 10)
            printf("x=%d",x);
      }
}
```

Suponha que um escalonador de um sistema com um único processador, ao implementar execução pseudoparalela dos dois processos concorrentes, intercale as instruções, sem restrição da ordem da intercalação. Mostre uma sequencia de instruções (trace) tal que o comando “x=10” seja impresso.

# Considere o seguinte programa

```
P1: {
    shared int x;
    x = 10;
    while (1) {
         x = x - 1;
         x = x + 1;
         if (x != 10)
            printf(”x=%d",x);
      }
}
```

```
P2: {
    shared int x;
    x = 10;
    while (1) {
         x = x - 1;
         x = x + 1;
         if (x != 10)
            printf("x=%d",x);
      }
}
```

```
P1: x = x – 1;
P1: x = x + 1;
P2: x = x – 1;
P1: if(x != 10)
P2: x = x + 1;
P1: printf("x is %d", x);
```

```
9
10
9
9
10
10
```

# Considere o seguinte programa

```
P1: {
    shared int x;
    x = 10;
    while (1) {
         x = x - 1;
         x = x + 1;
         if (x != 10)
            printf(”x=%d",x);
      }
}
```

```
P2: {
    shared int x;
    x = 10;
    while (1) {
         x = x - 1;
         x = x + 1;
         if (x != 10)
            printf("x=%d",x);
      }
}
```

Como semáforos devem ser adicionados para assegurar que o printf() não seja executado erroneamente? Sua solução deve permitir concorrência ao máximo.

# Considere o seguinte programa

```
semaphore s;
parbegin
P1: {
    shared int x;
    x = 10;
    for (;;) {
         semWait (s);
         x = x - 1;
         x = x + 1;
         if (x != 10)
            printf("x=%d",x);
         semSignal (s);
      }
}
```

```
P2: {
    shared int x;
    x = 10;
    for (;;) {
    semWait (s);
         x = x - 1;
         x = x + 1;
         if (x != 10)
            printf("x=%d",x);
         semSignal (s);
      }
}

parend
```

# Exclusão mútua por software

- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Comutação alternada
  - Comutação não alternada

# Busy wait

- espera ativa ou espera ocupada
- Para acessar uma região crítica, um processo verifica se pode entrar. Se não puder for, ele espera em um laço que o acesso seja liberado.
  - Conseqüência: desperdício de tempo de CPU.

# Variável de Bloqueio (ou comutação)

- indica se a região crítica está ou não em uso
  - *turn* = 0: livre
  - *turn* = 1: ocupada
- Tentativa para *n* processos:

```
var turn: 0..1
turn := 0
Process Pi:
...
while turn = 1 do {nothing};
turn := 1;
< critical section >
turn := 0;
...
```

# Variável de Bloqueio

- Não confirma exclusão mútua:

- Processos entrar ao mesmo tempo em região crítica, pois dois processos podem testar *turn* antes de setá-la para *true*

```
var turn: 0..1
turn := 0
Process Pi:
...
while turn = 1 do {nothing};
turn := 1;
< critical section >
turn := 0;
...
```

# Alternância Estrita

- Assegura a exclusão mútua entre dois processos alternando a execução entre as regiões críticas
- A variável *turn* indica qual processo está na vez de executar

**$P_A$**

```
while (turn != A);
Região Crítica A;
turn = B;
Processamento .....
```

**$P_B$**

```
while (turn != B);
Região Crítica B;
turn = A;
Processamento ....
```

- Um processo fora da sua região crítica “bloqueia” a execução do outro

# Alternância Estrita

- *turn* é global e binário – para dois processos

```
var turn: 0..1;
P0:
while turn≠0 do {nothing};
< critical section >
turn := 1;
```

```
P1:
while turn≠1 do {nothing};
< critical section >
turn := 0;
```

- garante a exclusão mútua  com alternância na execução dos diferentes processos
- Acesso a região crítica, de acordo com a velocidade do processo

# Comutação não Alternada

- Assegura a exclusão mútua entre dois processos sem precisar alternar a execução entre as regiões críticas
- A variável *turn* indica qual processo está na vez de executar, mas não necessariamente precisa executar
- Então, mais uma informação, em uma variável do tipo *array*
  - *Interested* indica se um processo está interessado e pronto para executar sua região crítica

# Comutação não Alternada

- Um processo entra na sua região crítica se o outro não estiver *interessado*
- Caso os dois processos estejam interessados o valor de *turn* decide qual processo ganha a região crítica

**$P_A$**

```
interested[A] = true;
turn = B;
while (interested[B] && turn==B);

< região crítica >
interested[A] = false;
```

**$P_B$**

```
interested[B] = true;
turn = A;
while (interested[A] && turn==A);

< região crítica >
interested[B] = false;
```