Laboratório de Pesquisa em Redes e Multimídia

Sistemas Operacionais

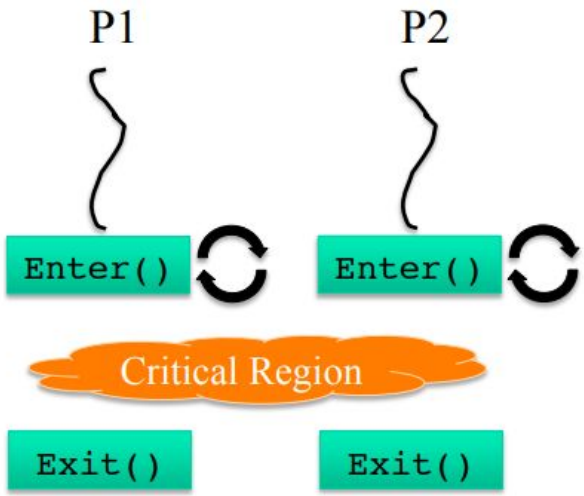
Sincronização de Processos: Semáforos

Universidade Federal do Espírito Santo
Departamento de Informática

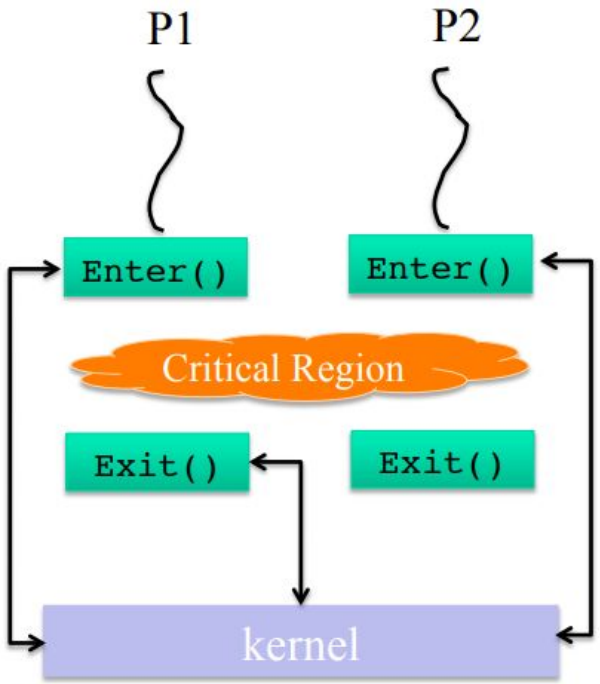
Tipos de Soluções (cont.)

- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Dekker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup, Semáforos, Monitores

Espera Ocupada x Bloqueio



Espera Ocupada: consome ciclos de processamento. Pode ser usado em multi-core



Bloqueio:.o núcleo garante atomicidade

Tipos de Soluções (cont.)

- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Dekker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Semáforos, Monitores

Semáforos (1)

- Mecanismo criado pelo matemático holandês E.W. Dijkstra, em 1965.
- O semáforo é uma **variável inteira** que pode ser mudada por apenas duas operações primitivas (atômicas): ***P*** e ***V***.
  - *P* = *proberen* (testar)
  - V = *verhogen* (incrementar).
- Quando um processo executa uma operação ***P***, o valor do semáforo é **decrementado** (se o semáforo for maior que 0). O processo pode ser eventualmente bloqueado (semáforo for igual a 0) e inserido na **fila de espera** do semáforo.
- Numa operação *V,* o semáforo é **incrementado** e, eventualmente, um processo que aguarda na **fila de espera** deste semáforo é acordado.

Semáforos (2)

- A operação ***P*** também é comumente referenciada como:
  - ***down*** ou ***wait***
- ***V*** também é comumente referenciada
  - ***up*** ou ***signal***

- Semáforos que assumem somente os valores *0* e *1* são denominados ***semáforos binários*** ou ***mutex***. Neste caso, P e V são também chamadas de ***LOCK*** e ***UNLOCK***, respectivamente.

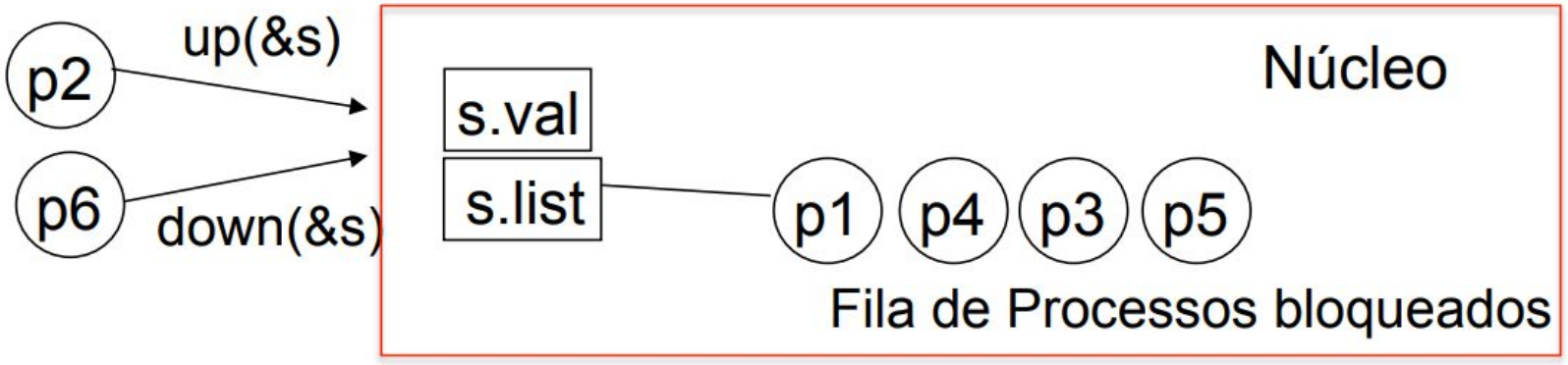
Semáforos (3)

```
P(S):   //down(S)
  If S > 0
   Then S := S - 1
  Else bloqueia processo (coloca-o na fila de S)


V(S):   //up(S)
 If algum processo dorme na fila de S
   Then acorda processo
 Else S := S + 1
```

As operações V(S) e P(S) são atômicas!

Semáforos (4)



- Operações down() e up() geralmente são implementadas como chamadas do núcleo (system call), e durante a sua execução o núcleo desabilita temporariamente as interrupções (para garantir a atomicidade).

- Se há mais de uma CPU, torna-se necessário bloquear o barramento, impedindo que 2 CPUs acessem o mesmo endereço na RAM.

Uso de Semáforos (1)

Exclusão mútua (semáforos binários/mutexes):

```
...
Semaphore s = 1;     /*var.semáforo,
                       iniciado com 1*/
```

Processo P_1	Processo P_2	...Processo P_n
...	...	...
P(mutex)	P(mutex)	P(mutex)
// R.C.	// R.C.	// R.C.
V(mutex)	V(mutex)	V(mutex)
...	...	...

Uso de Semáforos (2)

Alocação de Recursos (semáforos contadores)

```
...
Semaphore S := 3; /*var. semáforo, iniciado com
                    qualquer valor inteiro */

Processo P1          Processo P2          Processo P3
   ...                  ...                  ...

//usa recurso        //usa recurso        //usa recurso

   ...                  ...                  ...
```

Uso de Semáforos (3)

Alocação de Recursos (semáforos contadores)

```
...
Semaphore S := 3; /*var. semáforo, iniciado com
                    qualquer valor inteiro */
```

Processo P_1	Processo P_2	Processo P_3
...	...	...
P(S)	P(S)	P(S)
//usa recurso	//usa recurso	//usa recurso
V(S)	V(S)	V(S)
...	...	...

p0_rot1 -------> p1_rot2

Uso de Semáforos (4)

Sincronização (relação de precedência)

(Ex: executar *p1_rot2* somente depois de *p0_rot1*)

```
semaphore S := 0    ;
parbegin
   begin                /* processo P0*/
      p0_rot1()         ***

      p0_rot2()
   end
   begin                /* processo P1*/
      p1_rot1()

      p1_rot2()  ***
   end
parend
```

PARBEGIN – Inicia uma lista de programas que serão executados paralela e aleatoriamente (dependendo do escalonamento).
PAREND – Especifica o ponto de sincronização.

p0_rot1 -------> p1_rot2

Uso de Semáforos (5)

Sincronização (relação de precedência)

(Ex: executar *p1_rot2* somente depois de *p0_rot1*)

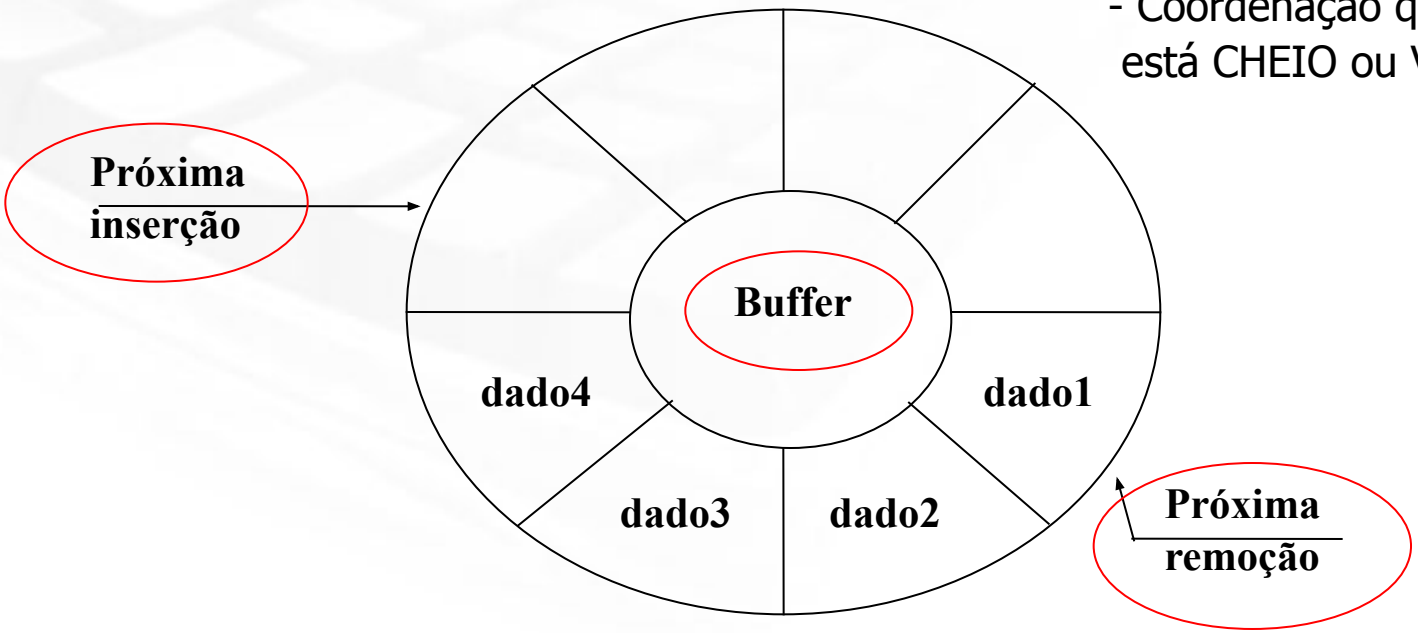
```
semaphore S := 0 ;
parbegin
   begin              /* processo P0*/
      p0_rot1()
      Up(S)
      p0_rot2()
   end
   begin              /* processo P1*/
      p1_rot1()
      Down(S)
      p1_rot2()
   end
parend
```

Problema do Produtor-Consumidor

- Na computação, o problema do produtor-consumidor (também conhecido como problema do buffer limitado) é um exemplo clássico de problema de sincronização multi-processo.
- Na sua versão tradicional, o problema descreve dois processos, o **produtor** e o **consumidor**, que compartilham um **buffer de tamanho fixo**. O trabalho do produtor é gerar um dado, colocá-lo no buffer e repetir essa operação indefinidamente. Ao mesmo tempo, a tarefa do consumidor é consumir tais dados (i.e. removendo do buffer), um de cada vez.
- O problema consiste em assegurar que o produtor não irá tentar adicionar dados no buffer quando este estiver cheio, e que o consumidor não tentará remover dados quando o buffer estiver vazio.

Produtor - Consumidor c/ *Buffer* Circular (1)

Dois problemas p/ resolver:
- Variáveis compartilhadaS
- Coordenação quando o buffer está CHEIO ou VAZIO



Produtor Consumidor c/ *Buffer* Circular (2)

- *Buffer* com capacidade N (vetor de N elementos).
- Variáveis *proxima_insercao* e *proxima_remocao* indicam onde deve ser feita a próxima inserção e remoção no *buffer*.
- Efeito de *buffer* circular é obtido através da forma como essas variáveis são incrementadas. Após o valor *N-1* elas voltam a apontar para a entrada zero do vetor
  - % representa a operação "resto da divisão"
- Três semáforos, duas funcionalidades diferentes:
  - exclusão mútua
    - ***mutex***: garante a exclusão mútua. Deve ser iniciado com "1".
  - sincronização.
    - ***espera_dado***: bloqueia o consumidor se o *buffer* está vazio. Iniciado com "0".
    - ***espera_vaga***: bloqueia produtor se o *buffer* está cheio. Iniciado com "N".

Produtor - Consumidor c/ *Buffer* Circular (3)

```
struct tipo_dado buffer[N];
int proxima_insercao := 0;
int proxima_remocao := 0;
...
semaphore mutex := 1;
semaphore espera_vaga := N;
semaphore espera_dado := 0;

----------------------------------------
void produtor(void){
...
down(espera_vaga)

buffer[proxima_insercao] := dado_produzido;
proxima_insercao := (proxima_insercao + 1) % N;

up(espera_dado)
... }

----------------------------------------
void consumidor(void){
...
down(espera_dado)

dado_a_consumir := buffer[proxima_remocao];
proxima_remocao := (proxima_remocao + 1) % N;

up(espera_vaga)
... }
```

```
down(S):
  SE S > 0 ENTÃO S := S - 1
  SENÃO bloqueia processo


up(S):
  SE algum processo dorme na fila de S
  ENTÃO acorda processo
  SENÃO S := S + 1
```

```
struct tipo_dado buffer[N];
int proxima_insercao := 0;
int proxima_remocao := 0;
...
semaphore mutex := 1;
semaphore espera_vaga := N;
semaphore espera_dado := 0;

----------------------------------------
void produtor(void){
...
down(espera_vaga);
down(mutex);
buffer[proxima_insercao] := dado_produzido;
proxima_insercao := (proxima_insercao + 1) % N;
up(mutex);
up(espera_dado);
... }

----------------------------------------
void consumidor(void){
...
down(espera_dado);
down(mutex);
dado_a_consumir := buffer[proxima_remocao];
proxima_remocao := (proxima_remocao + 1) % N;
up(mutex);
up(espera_vaga);
... }
```

```
down(S):
  SE S > 0 ENTÃO S := S - 1
  SENÃO bloqueia processo


up(S):
  SE algum processo dorme na fila de S
  ENTÃO acorda processo
  SENÃO S := S + 1
```

Produtor - Consumidor c/ *Buffer* Circular (3)

Deficiência dos Semáforos (1)

- Exemplo: suponha que os dois *down* do código do produtor estivessem invertidos. Neste caso, *mutex* seria diminuído antes de *empty*. Se o *buffer* estivesse completamente cheio, o produtor bloquearia com *mutex = 0*. Portanto, da próxima vez que o consumidor tentasse acessar o *buffer* ele faria um *down* em *mutex*, agora zero, e também bloquearia. Os dois processos ficariam bloqueados eternamente.
- Conclusão: erros de programação com semáforos podem levar a resultados imprevisíveis.

```
struct tipo_dado buffer[N];
int proxima_insercao := 0;
int proxima_remocao := 0;
...
semaphore mutex := 1;
semaphore espera_vaga := N;
semaphore espera_dado := 0;

----------------------------------------
void produtor(void){
...
down(espera_vaga);
down(mutex);
buffer[proxima_insercao] := dado_produzido;
proxima_insercao := (proxima_insercao + 1) % N;
up(mutex);
up(espera_dado);
... }

----------------------------------------
void consumidor(void){
...
down(espera_dado);
down(mutex);
dado_a_consumir := buffer[proxima_remocao];
proxima_remocao := (proxima_remocao + 1) % N;
up(mutex);
up(espera_vaga);
... }
```

EXERCÍCIO:
Havendo apenas 1 consumidor e 1 produtor, se retirarmos as chamadas ao semáforo "mutex" (em vermelho), isso causaria algum erro/condição de corrida? O que aconteceria?
Dica: Observem que nesta implementação do buffer, dois processos podem, simultaneamente, acessar posições diferentes do buffer sem causar condição de corrida.

Produtor - Consumidor c/ *Buffer* Circular

Deficiência dos Semáforos (2)

- Embora semáforos forneçam uma abstração flexível o bastante para tratar diferentes tipos de problemas de sincronização, ele é inadequado em algumas situações.
- Semáforos são uma abstração de alto nível baseada em primitivas de baixo nível, que provêm atomicidade e mecanismo de bloqueio, com manipulação de filas de espera e de escalonamento. Tudo isso contribui para que a operação seja lenta.
- Para alguns recursos, isso pode ser tolerado; para outros esse tempo mais longo é inaceitável.
  - Ex: (Unix) Se um bloco *x* do disco é achado no *buffer cache*, *getblk()* tenta reservá-lo com P(). Se o *buffer* já estiver reservado, não há nenhuma garantia que ele conterá o mesmo bloco que ele tinha originalmente.