


© 2004-2006 Volnys Bernal 1

Sincronização e Comunicação entre Processos

Volnys Borges Bernal
volnys@lsi.usp.br
<http://www.lsi.usp.br/~volnys>




© 2004-2006 Volnys Bernal 2

Agenda

- ❑ Condições de disputa
- ❑ Região Crítica
- ❑ Classificação dos mecanismos de sincronização quanto à espera
- ❑ Primitivas de sincronização e comunicação
 - ❖ Exclusão Mútua
 - Implementação em software
 - Alternância obrigatória
 - Solução de Peterson
 - Implementação utilizando recursos de baixo nível
 - Desabilitar interrupção
 - Intrução Test-And-Set (TST)
 - ❖ Primitivas explicitamente bloqueantes
 - Sleep & Wakeup
 - Wait & Signal
 - ❖ Semáforo
 - ❖ Monitor
 - ❖ Troca de mensagens

© 2004-2006 Volnys Bernal 3

Condições de disputa (Race Conditions)



© 2004-2006 Volnys Bernal 4

Condições de disputa

- ❑ Condição de disputa é
 - ❖ Uma situação de conflito ...
 - ❖ No acesso a um determinado recurso (variável, arquivo, ...)
 - ❖ Por duas ou mais entidades (processos, *threads*, ...)
 - ❖ Que pode causar situações não desejáveis e resultados não esperados
- ❑ Importante:
 - ❖ *Threads* de um mesmo processo possuem diversos recursos compartilhados
 - Área de dados
 - Arquivos abertos
 - etc
 - ❖ Quando existem acessos de escrita a estes recursos compartilhados podem ocorrer potenciais situações de condição de disputa

© 2004-2006 Volnys Bernal 5

Condições de disputa

- ❑ Exemplo 1: Contador de tarefas
 - ❖ Dois *threads* realizam determinadas tarefas.
 - ❖ Após cada tarefa incrementam um contador *c*.

Thread1:

```
...
while (1)
  <Realiza tarefa>
  c = c + 1
...
```

Thread2:

```
...
while (1)
  <Realiza tarefa>
  c = c + 1
...
```

© 2004-2006 Volnys Bernal 6

Condições de disputa

- ❑ Exemplo 1: Contador de tarefas (cont.)
 - ❖ Versão do programa em assembler

Thread1:

```
...
repete: ...
        realiza tarefa
        ...
        LOAD    AC,(c)
        ADD     AC,1
        STORE   (c),AC
        JUMP    repete
        ...
```

Thread2:

```
...
repete: ...
        realiza tarefa
        ...
        LOAD    AC,(c)
        ADD     AC,1
        STORE   (c),AC
        JUMP    repete
        ...
```

© 2004-2006 Volnys Bernal 7

Condições de disputa

❑ Exemplo 1: Contador de tarefas (cont.)

❖ Problemas: condição de disputa sobre o contador "c"

- Sistemas monoprocessadores
 - Concorrência:
 - troca de contexto durante a atualização do contador "c"
- Sistemas multiprocessadores
 - Concorrência:
 - troca de contexto durante a atualização do contador "c"
 - Paralelismo:
 - incremento simultâneo do contador "c"

© 2004-2006 Volnys Bernal 8

Condições de disputa

❑ Exemplo 1: Contador de tarefas (cont.)

❖ Condição de disputa na concorrência:

Thread1:

```

repete: ...
          realiza tarefa
          ...
          LOAD AC, (c)
          ADD AC, 1
          STORE (c), AC
          JUMP repete
          ...
        
```

Thread2:

```

repete: ...
          realiza tarefa
          ...
          LOAD AC, (c)
          ADD AC, 1
          STORE (c), AC
          JUMP repete
          ...
        
```

© 2004-2006 Volnys Bernal 9

Condições de disputa

❑ Exemplo 1: Contador de tarefas (cont.)

❖ Condição de disputa no paralelismo

Thread1:

```

...
repete: ...
        realiza tarefa
        ...
        LOAD AC, (c)
        ADD AC, 1
        STORE (c), AC
        JUMP repete
        ...
      
```

Thread2:

```

...
repete: ...
        realiza tarefa
        ...
        LOAD AC, (c)
        ADD AC, 1
        STORE (c), AC
        JUMP repete
        ...
      
```

© 2004-2006 Volnys Bernal 10

Condições de disputa

❑ Exemplo 2: Manipulação de lista ligada

❖ Dois threads manipulam (leitura ou modificação) uma lista ligada

Thread1:

```

...
<manipula lista ligada>
...

```

Thread2:

```

...
<manipula lista ligada>
...

```

© 2004-2006 Volnys Bernal 11

Condições de disputa

❑ Exemplo 2: Manipulação de lista ligada (cont.)

❖ Problema: condição de disputa durante a manipulação da lista ligada:

- Sistemas monoprocessadores
 - Concorrência: Troca de contexto durante a atualização da lista deixa-a em um estado inconsistente
- Sistemas multiprocessadores
 - Concorrência: Troca de contexto durante a atualização da lista deixa-a em um estado inconsistente
 - Paralelismo: Dois threads utilizando a lista simultaneamente e, pelo menos 1 deles alterando-a

© 2004-2006 Volnys Bernal 12

Condições de disputa

❑ Exemplo 3: Variável de proteção

❖ Dois threads definem uma variável compartilhada para controle do uso do recurso.

❖ Se a variável for 1 significa que o recurso está ocupado, se for zero está livre.

Thread1:

```

...
while (ocupado == 1);
ocupado = 1;
<usa recurso>
ocupado = 0;
...

```

Thread2:

```

...
while (ocupado == 1);
ocupado = 1;
<usa recurso>
ocupado = 0;
...

```

© 2004-2006 Volnys Bernal 13

Condições de disputa

❑ Exemplo 3: Variável de proteção (cont.)

❖ Versão em assembler

```

Thread1:
...
repete: LOAD    AC, (ocupado)
        JUMP_EQ AC, 1, repete
        STORE   1, (ocupado)
...
        usa_recurso
...
        STORE   0, (ocupado)
        ...

```

```

Thread1:
...
repete: LOAD    AC, (ocupado)
        JUMP_EQ AC, 1, repete
        STORE   1, (ocupado)
...
        usa_recurso
...
        STORE   0, (ocupado)
        ...

```

© 2004-2006 Volnys Bernal 14

Condições de disputa

❑ Exemplo 3: Variável de proteção (cont.)

❖ Problema: Condição de disputa sobre a variável "ocupado"

- Sistemas monoprocessoadores
 - Concorrência: Troca de contexto durante a alteração da variável "ocupado" para 1
- Sistemas multiprocessoadores
 - Concorrência: Troca de contexto durante a alteração da variável "ocupado" para 1
 - Paralelismo: Dois *threads* alterando simultaneamente a variável "ocupado" para 1

© 2004-2006 Volnys Bernal 15

Condições de disputa

❑ Exemplo 3: Variável de proteção (cont.)

❖ Condição de disputa quando existe concorrência

```

Thread1:
...
repete: LOAD    AC, (ocupado)
        JUMP_EQ AC, 1, repete
        STORE   1, (ocupado)
...
        usa_recurso
...
        STORE   0, (ocupado)
        ...

```


```

Thread1:
...
repete: LOAD    AC, (ocupado)
        JUMP_EQ AC, 1, repete
        STORE   1, (ocupado)
...
        usa_recurso
...
        STORE   0, (ocupado)
        ...

```

© 2004-2006 Volnys Bernal 16

Região Crítica



© 2004-2006 Volnys Bernal 17

Região Crítica

❑ Região crítica é ...

- ❖ Uma região de código ...
- ❖ Na qual existe acesso a recursos compartilhados ...
- ❖ Que pode causar problema de condição de disputa

❑ Objetivo da região crítica

- ❖ Identificar a região de código na qual existe potencialmente ocorrência de condição de disputa entre duas ou mais entidades
- ❖ Possibilitar a utilização de soluções de sincronização para evitar condição de disputa na região crítica

❑ Obs:

- ❖ Entidades
 - Processos, threads, ...

© 2004-2006 Volnys Bernal 18

Região Crítica

❑ Exemplo de região de código na qual existe acesso a recursos compartilhados que pode causar problema de condição de disputa

```

...
<manipula recurso
compartilhado>
...

```

```

...
<manipula recurso
compartilhado>
...

```

Região Crítica

© 2004-2006 Volnys Bernal 19

Região Crítica

❑ Exemplo 1:
❖ Contador de tarefas

Thread1:

```
...
while (1)
...
<Realiza tarefa>
...
c = c + 1
...
```

Thread2:

```
...
while (1)
...
<Realiza tarefa>
...
c = c + 1
...
```

RC_1

© 2004-2006 Volnys Bernal 20

Região Crítica

❑ Exemplo 2:
❖ Manipulação de lista ligada

Thread1:

```
...
<manipula lista ligada>
...
```

Thread2:

```
...
<manipula lista ligada>
...
```

RC_1

© 2004-2006 Volnys Bernal 21

Região Crítica

❑ Exemplo 3:
❖ Variável de proteção

Thread1:

```
...
while (ocupado == 1);
ocupado = 1;
...
<usa recurso>
...
ocupado = 0;
...
```


Thread2:

```
...
while (ocupado == 1);
ocupado = 1;
...
<usa recurso>
...
ocupado = 0;
...
```

RC_1

© 2004-2006 Volnys Bernal 22

Classificação dos mecanismos de sincronização quanto à espera



© 2004-2006 Volnys Bernal 23

Classificação quanto à espera

Espera ociosa (*busy waiting*)


- ❖ A entidade (processo ou thread) testa repetidamente a condição de sincronização. Geralmente é utilizada uma variável de impedimento, que é chamada de "spin lock"
- ❖ Problema: Desperdício de tempo de CPU quando a espera é longa
- ❖ Utilizada tipicamente em aplicações paralelas (multiprocessamento) em situações com sincronização rápida

❑ Bloqueante

- ❖ Não desperdiça tempo de CPU
- ❖ Acrescenta a sobrecarga da troca de contexto em sistemas multiprocessadores
- ❖ Utilizada nos caso gerais
- ❖ Quando em modo usuário requer uma chamada ao sistema

© 2004-2006 Volnys Bernal 24

Mecanismos de sincronização e comunicação




© 2004-2006 Volnys Bernal 25

Mecanismos de Sincronização e comunicação

- ❑ **Exclusão Mútua**
 - ❖ Implementação em software
 - Alternância obrigatória
 - Solução de Peterson
 - ❖ Implementação utilizando recursos de baixo nível
 - Desabilitar interrupção
 - Intrução Test-And-Set (TST)
- ❑ **Primitivas explicitamente bloqueantes**
 - ❖ Sleep & Wakeup
 - ❖ Wait & Signal
- ❑ **Semáforo**
- ❑ **Monitor**
- ❑ **Troca de mensagens**

© 2004-2006 Volnys Bernal 26

Exclusão Mútua (Mutex)



© 2004-2006 Volnys Bernal 27


Exclusão Mútua (Mutex)

- ❑ **Objetivo:**
 - ❖ Assegurar o acesso exclusivo (leitura e escrita) a um recurso compartilhado por duas ou mais entidades
- ❑ **Útil para**
 - ❖ Prevenção de problema de condição de disputa em regiões críticas
- ❑ **Requisitos para a implementação de exclusão mútua**
 - 1- Nunca duas entidades podem estar simultaneamente em suas regiões críticas
 - 2- Deve ser independente da quantidade e desempenho dos processadores
 - 3- Nenhuma entidade fora da região crítica pode ter a exclusividade desta
 - 4- Nenhuma entidade deve esperar eternamente para entrar em sua região crítica

© 2004-2006 Volnys Bernal 28

Exclusão Mútua (Mutex)

- ❑ **Exemplo:**
 - ❖ T1 – A tenta entrar na região crítica
 - ❖ T2 – B tenta entrar na região crítica
 - ❖ T3 – A sai da região crítica; B entra na região crítica
 - ❖ T4 – B sai da região crítica



© 2004-2006 Volnys Bernal 29

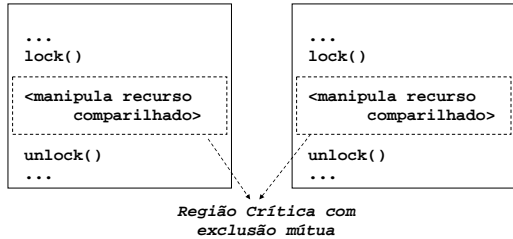
Exclusão Mútua (Mutex)

- ❑ **Pode ser implementada com duas primitivas básicas:**
 - ❖ lock() [ou enter_region()]
 - Garante a exclusividade da região crítica no ponto de entrada da região
 - ❖ unlock() [ou leave_region()]
 - Libera a exclusividade da região crítica no ponto de saída da região

© 2004-2006 Volnys Bernal 30

Exclusão Mútua (Mutex)

- ❑ **Exemplo:**
 - ❖ lock() - para obter a exclusão mútua sobre a RC
 - ❖ unlock() - para liberar a exclusão mútua sobre a RC



© 2004-2006 Volnys Bernal 31

Exclusão Mútua (Mutex)

- Primitivas pthreads


```
// Iniciação estática
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;

// Iniciação dinâmica
pthread_mutex_t mymutex;
int pthread_mutex_init (pthread_mutex_t *mutex,
                        pthread_mutexattr_t *attr);

// Primitivas
int pthread_mutex_init (pthread_mutex_t *mutex,
                        pthread_mutexattr_t *attr)
int pthread_mutex_lock (pthread_mutex_t *mutex)
int pthread_mutex_unlock (pthread_mutex_t *mutex)
int pthread_mutex_trylock (pthread_mutex_t *mutex)
```

© 2004-2006 Volnys Bernal 32

Mutex em software: Alternância Obrigatória



© 2004-2006 Volnys Bernal 33

Alternância Obrigatória

- Objetivo**
 - Implementação de exclusão mútua
- Descrição**
 - Altera o acesso à região crítica entre duas entidades
- Desvantagem:**
 - Viola requisito #3 (Nenhuma entidade fora da região crítica pode ter a exclusividade desta)
 - Válida para somente duas entidades (processos/threads)
 - Utiliza espera ociosa

© 2004-2006 Volnys Bernal 34

Alternância Obrigatória

Entidade 1:


```
...
while (TRUE)
{
    // lock()
    while (turn!=0);
    ...
    região_critica
    ...
    // unlock()
    turn=1;
    ...
}
...
```

Entidade 2:

```
...
while (TRUE)
{
    // lock()
    while (turn!=1);
    ...
    região_critica
    ...
    // unlock()
    turn=0;
    ...
}
...
```

© 2004-2006 Volnys Bernal 35

Mutex em software: Solução de Peterson



© 2004-2006 Volnys Bernal 36

Solução de Peterson

- Publicada por G. L. Peterson em 1981
- Baseada em uma solução do matemático holandês T. Dekker
- Utiliza espera ociosa
- Válida somente para 2 entidades

© 2004-2006 Volnys Bernal 37

Solução de Peterson

```

int turn;                // Duas entidades:
int interested[2];       // Entidade 0
                        // Entidade 1

void lock(int me)
{
    int me;
    int other;


    other = (me + 1) mod 2;
    interested[me] = TRUE;
    turn = me;
    while (turn != me && interested[other] == TRUE);
}

void unlock(int me)
{
    interested[me] = FALSE;
}

```

© 2004-2006 Volnys Bernal 38

Mutex usando recursos de baixo nível: Desabilitar Interrupção



© 2004-2006 Volnys Bernal 39

Desabilitar Interrupção

- ❑ **Objetivo**
 - ❖ Implementação de exclusão mútua
- ❑ **Descrição**
 - ❖ Desabilita interrupção de relógio (que impede a ocorrência de troca de contexto) para garantir a exclusão mútua em uma região crítica
- ❑ **Problemas**
 - ❖ Não aplicável para modo usuário
 - Processos e *threads* executados em modo usuário
 - Impede a troca de contexto com outros processos/threads não relacionados à região crítica.
 - ❖ Possibilita que um erro do código (ex, loop) faça com que o sistema fique inoperante.
 - ❖ NÃO resolve o problema em sistemas multiprocessadores
- ❑ **Onde é útil**
 - ❖ Internamente ao sistema operacional
 - Em *threads* internos ao núcleo do sistema operacional
 - Na implementação de drivers de dispositivos que utilizam interrupção, em regiões críticas presentes em rotinas de tratamento de interrupção

© 2004-2006 Volnys Bernal 40

Desabilitar interrupção

- ❑ **Exemplo:**
 - ❖ lock()

```

{
    desabilita_interrupção_relogio;
}

```
 - ❖ unlock()


```

{
    habilita_interrupção_relogio;
}

```

© 2004-2006 Volnys Bernal 41

Mutex usando recursos de baixo nível: Instrução Test And Set Lock



© 2004-2006 Volnys Bernal 42

Instrução Test And Set Lock

- ❑ **Objetivo**
 - ❖ Primitiva de baixo nível para implementação de sincronização
- ❑ **Descrição**
 - ❖ Instrução especial da CPU
 - ❖ Operação
 - (variável_memória) → registrador (leitura)
 - 1 → (variável_memória) (escrita)
 - ❖ Instrução atômica (indivisível)
 - As operações de leitura da variável e alteração (escrita) do valor ocorrem em uma única instrução. Não existe possibilidade de ocorrer interrupção entre estas operações.
 - ❖ Acesso atômico à memória
 - Em sistemas multiprocessadores é garantido que o acesso à memória (leitura/escrita) seja atômico, ou seja, não seja interrompido entre as operações de leitura e escrita
- ❑ **Primitiva básica para construção de primitivas de exclusão mútua**

© 2004-2006 Volnys Bernal 43

Instrução Test And Set Lock

❑ Exemplo:


- ❖ Implementação de exclusão mútua utilizando TST
- ❖ “var” é uma variável alocada na memória

```
lock:    TST  register,(var)  # register ← var; var ← 1
        CMP  register,#0    # register == 0?
        JNE  lock           # se register != 0, loop
        RET                  # retorna

unlock:  MOV  (var),#0       # var ← 0 (libera lock)
        RET                  # retorna
```

© 2004-2006 Volnys Bernal 44

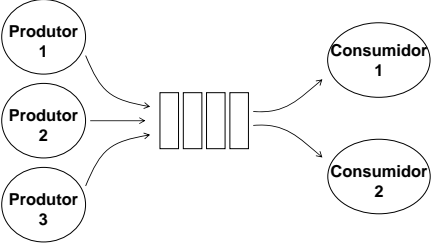
Exercício



© 2004-2006 Volnys Bernal 45

Exercício

(1) Em relação ao problema do produtor-consumidor, faça um esboço de solução sem levar em consideração as condições de disputa.



© 2004-2006 Volnys Bernal 46

Exercício

❑ Primeiro esboço de solução sem levar em consideração as condições de disputa

```
Produtor
Repetir
    Produzir(E);
    InserirFila(F,E);

Consumidor
Repetir
    E = RetirarFila(F);
    Processar(E);
```

© 2004-2006 Volnys Bernal 47

Exercício

(2) Em relação ao problema do produtor-consumidor, analise o código, identifique as condições de disputa e defina as regiões críticas.

© 2004-2006 Volnys Bernal 48

Exercício

❑ Identificação das condições de disputa

```
Produtor
Repetir
    Produzir(E);
    InserirFila(F,E);

Consumidor
Repetir
    E = RetirarFila(F);
    Processar(E);
```

Condição de disputa

Condição de disputa

Fila:

- recurso compartilhado
- pode ser acessado de forma concorrente

© 2004-2006 Volnys Bernal 49

Exercício

(3) Em relação ao problema do produtor-consumidor, identifique outras necessidades de sincronização como, por exemplo, sincronização por disputa de recurso. Quais são os recursos disputados?

© 2004-2006 Volnys Bernal 50

Exercício

□ Identificação de outras necessidades de sincronização

Produtor
Repetir
 Produzir(E);
 InserirFila(F,E);

Consumidor
Repetir
 E = RetirarFila(F);
 Processar(E);

(1) Não leva em consideração que o recurso "Fila" é limitado, ou seja, a fila pode tornar-se cheia. Nesta situação (de fila cheia) os produtores não podem inserir itens na fila.

(2) Os consumidores podem ser mais rápidos que os produtores permitindo que em determinados momentos a fila fique vazia. Neste momento, não é possível retirar da fila.

© 2004-2006 Volnys Bernal 51

Exercício

□ Identificação de outras necessidades de sincronização

Produtor
Repetir
 Produzir(E);
 Enquanto FilaCheia(F)
 Aguardar;
 InserirFila(F,E);

Consumidor
Repetir
 Enquanto FilaVazia(F)
 aguardar;
 E = RetirarFila(F);
 Processar(E);

Condição de disputa

Condição de disputa

© 2004-2006 Volnys Bernal 52

Exercício

(4) Em relação ao problema do produtor-consumidor, apresente uma solução para o problema utilizando primitivas de exclusão mútua.

© 2004-2006 Volnys Bernal 53

Exercício

□ Primeiro esboço: proteção das regiões críticas

Produtor
Repetir
 Produzir(E);
 lock();
 Enquanto FilaCheia(F)
 Aguardar;
 InserirFila(F,E);
 unlock();

Consumidor
Repetir
 lock();
 Enquanto FilaVazia(F)
 aguardar;
 E = RetirarFila(F);
 unlock();
 Processar(E);

© 2004-2006 Volnys Bernal 54

Exercício

□ Primeiro esboço: proteção das regiões críticas

Produtor
Repetir
 Produzir(E);
 lock();
 Enquanto FilaCheia(F)
 Aguardar;
 InserirFila(F,E);
 unlock();

Consumidor
Repetir
 lock();
 Enquanto FilaVazia(F)
 aguardar;
 E = RetirarFila(F);
 unlock();
 Processar(E);

□ Problemas:

(1) Deadlock quando produtor encontra fila cheia

(2) Deadlock quando consumidor encontra fila vazia

© 2004-2006 Volnys Bernal 55

Exercício

□ Segundo esboço:

```

Produtor()
{
    repetir
    {
        Produzir(E);
        lock();
        enquanto FileCheia(F)
        {
            unlock();
            lock();
        }
        InserirFila(F,E);
        unlock();
    }
}

Consumidor()
{
    repetir
    {
        lock();
        enquanto FilaVazia(F)
        {
            unlock();
            lock();
        }
        E = RetirarFila(F);
        unlock();
        Processar(E);
    }
}

```

© 2004-2006 Volnys Bernal 56

Exercício

(5) Em relação ao problema do produtor-consumidor, baseado na solução apresentada responda:

- O produtor, quando possui um item produzido e a fila está cheia o que ocorre?
- O consumidor, quando deseja retirar um item da fila e a fila está vazia o que ocorre?
- Qual é o problema que ocorre nestas situações no qual não existem recursos disponíveis?

© 2004-2006 Volnys Bernal 57

Exercício

- Produtor desperdiça tempo de CPU quando a fila está cheia.
- Consumidor desperdiça tempo de CPU quando a fila está vazia.
- Gasto de tempo de CPU de maneira desnecessária para teste da condição de disponibilidade de recurso.

Primitivas de exclusão mútua não são adequadas para implementação do bloqueio em situações de sincronização a espera de recursos.

© 2004-2006 Volnys Bernal 58


Exercício

(6) Em relação ao problema do produtor-consumidor, analise sua solução e responda:

- A implementação funciona com múltiplos produtores e múltiplos consumidores?
- Suponha que o sistema seja monoprocessador. Qual tipo de primitiva é a mais recomendada: espera ociosa ou bloqueante? Porque?
- Suponha que o sistema seja multiprocessador. Qual tipo de primitiva é a mais recomendada: espera ociosa ou bloqueante? Porque?

© 2004-2006 Volnys Bernal 59

Problema da Inversão de Prioridade



© 2004-2006 Volnys Bernal 60

Problema de Inversão de Prioridade

- Ocorre quando
 - ❖ Entidade possui prioridade alta, não preemptiva e primitiva do tipo espera ociosa
- Preemptivo
 - ❖ Quando a entidade (thread/processo) é escalonada por utilizar por muito tempo a CPU
- Exemplo:
 - ❖ Em um sistema monoprocessado, considere um sistema com 2 threads:
 - H – Thread de alta prioridade
 - L – Thread de baixa prioridade
 - ❖ Escalonamento: H sempre é executado quando está no estado pronto (ou seja, tem preferência sobre L)
 - ❖ L ganha a região crítica e H torna-se pronto
 - ❖ H é escalonado e tenta ganhar a região crítica

© 2004-2006 Volnys Bernal 61

Problema de Inversão de Prioridade

❑ Exemplo com possibilidade de *deadlock*

Thread1: Alta prioridade e não preemptível

```

repetir
...
lock()
...
RC
...
unlock()
...

```

Thread2: Baixa prioridade e preemptível

```

repetir
...
lock()
...
RC
...
unlock()
...

```

Região Crítica com exclusão mútua

© 2004-2006 Volnys Bernal 62

Primitivas Explicitamente Bloqueantes

© 2004-2006 Volnys Bernal 63

Primitivas Explicitamente Bloqueantes

❑ Também chamadas de

- ❖ Sincronização por variáveis de condição

❑ Duas classes principais:

- ❖ Sleep & Wakeup
- ❖ Wait & Signal

❑ Sleep & Wakeup

- ❖ Utilizadas em ambiente não preemptível (quando não há troca de contexto entre os threads por término da fatia de tempo) e sistemas monoprocessadores
- ❖ Método de sincronização geralmente utilizado no núcleo do sistema operacional (Ex: UNIX)

❑ Wait & Signal

- ❖ Utilizadas em ambiente preemptível (quando há troca de contexto entre os threads por término da fatia de tempo)
- ❖ Utilizado geralmente em processos de usuário

© 2004-2006 Volnys Bernal 64

Primitivas Explicitamente Bloqueantes

❑ Resumo das primitivas

Primitiva	Tipo de ambiente	Local típico de utilização
Sleep & Wakeup	Não preemptível e monoprocessador	Internamente em sistemas operacionais (modo supervisor)
Wait & Signal	Preemptível	Em aplicações (modo usuário)

© 2004-2006 Volnys Bernal 65

Sleep & Wakeup

© 2004-2006 Volnys Bernal 66

Sleep & Wakeup

❑ Solução de sincronização

- ❖ Bloqueante
- ❖ Necessita de uma variável de condição
- ❖ Pressupõe um ambiente não preemptível (quando não existe troca de contexto por interrupção de relógio) e monoprocessador. A troca de contexto sempre é realizada de maneira explícita.

❑ Exemplo:

- ❖ Utilizada internamente ao núcleo do sistema operacional UNIX tradicional

❑ Primitivas

- ❖ Sleep
 - Bloqueia a entidade (processo ou *thread*) até a ocorrência de um determinado evento
- ❖ Wakeup
 - Gera evento de desbloqueio (para que entidades que estejam eventualmente esperando por um determinado evento sejam desbloqueadas)

© 2004-2006 Volnys Bernal 67

Sleep & Wakeup

- ❑ **Problema produtor-consumidor com Sleep & Wakeup**
 - ❖ **Duas variáveis de condição**
 - FilaCheia – Para bloquear o produtor no caso de fila cheia
 - FilaVazia – Para bloquear o consumidor no caso de fila vazia
 - ❖ **Sleep**
 - Para bloquear o produtor no caso de fila cheia
 - Para bloquear o consumidor no caso de fila vazia
 - ❖ **Wakeup**
 - Utilizada pelo consumidor para desbloquear o produtor quando a fila estiver cheia
 - Utilizada pelo produtor para desbloquear o consumidor quando a fila estiver vazia

© 2004-2006 Volnys Bernal 68

Sleep & Wakeup

```
#define N 100
int count = 0;

void producer(void)
{
    int item;
    while (TRUE) {
        item = produce_item();
        if (count == N)
            sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1)
            wakeup(consumer);
    }
}

void consumer(void)
{
    int item;
    while (TRUE) {
        if (count == 0)
            sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1)
            wakeup(producer);
        consume_item(item);
    }
}
```


© 2004-2006 Volnys Bernal 69

Sleep & Wakeup

- ❑ **No exemplo anterior observe que:**
 - ❖ **A variável count é compartilhada! Não existiria o problema de condição de disputa?**
 - Resposta: não pois não ocorre interrupção de relógio!
 - ❖ **Duas situações importantes:**
 - Quando a fila está cheia:
 - O produtor quando tiver um item para armazenar é bloqueado (sleep) pois não existe espaço para armazenamento de "itens".
 - Quando o consumidor liberar espaço, desbloqueia (wakeup) o produtor
 - Quando a fila está vazia:
 - Se o consumidor for consumir um item ele é bloqueado (sleep) pois não existem itens disponíveis
 - Quando o produtor produzir um item, desbloqueia (wakeup) o consumidor

© 2004-2006 Volnys Bernal 70

Wait & Signal



© 2004-2006 Volnys Bernal 71

Wait & Signal

- ❑ **Solução de sincronização**
 - ❖ Bloqueante
 - ❖ Necessita de uma variável de condição
 - ❖ Pressupõe um ambiente preemptível (quando existe troca de contexto por interrupção de relógio)
- ❑ **Exemplo**
 - ❖ Utilizada em processos/threads executados em modo usuário
- ❑ **Primitivas**
 - ❖ **Wait**
 - Bloqueia a entidade (processo ou thread) até a ocorrência de um determinado evento
 - ❖ **Signal**
 - Possibilita gerar eventos (ou seja, possibilita que entidades que estejam eventualmente esperando por um determinado evento sejam desbloqueadas)

© 2004-2006 Volnys Bernal 72

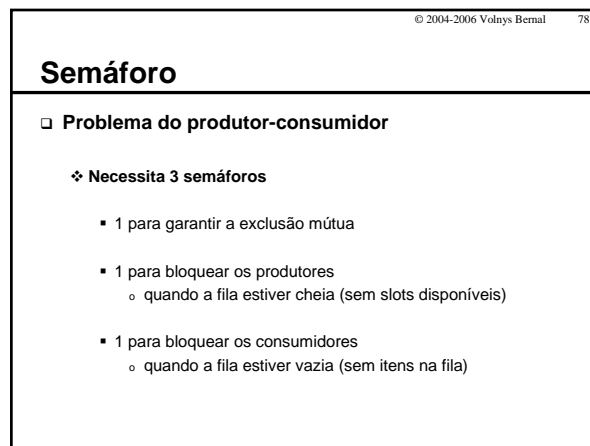
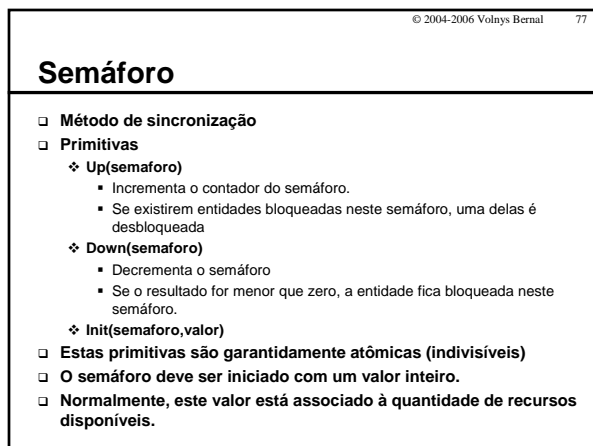
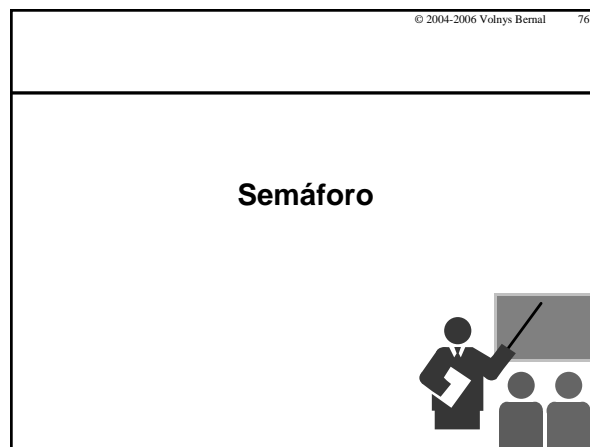
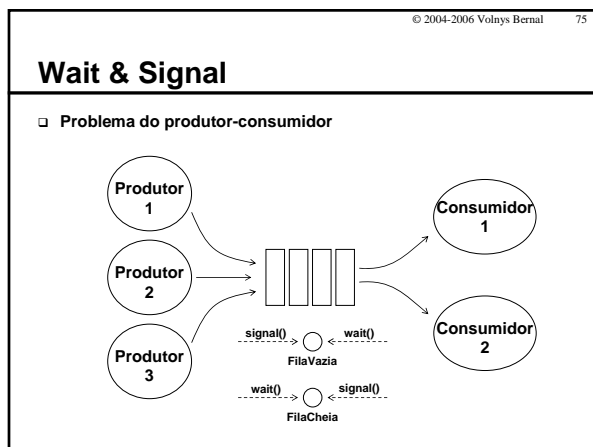
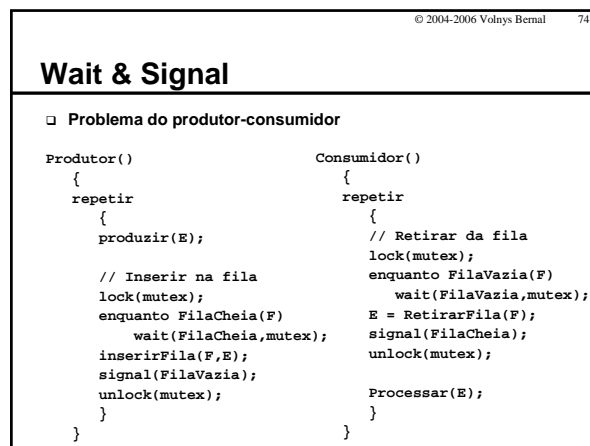
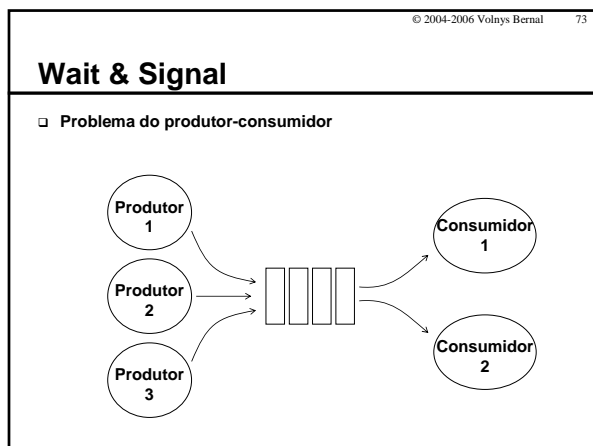
Wait & Signal

- ❑ **Primitivas pthread**
 - ❖ Primitivas denominadas de "variável de condição"

```
// Iniciação estática
pthread_cond_t mycondv = PTHREAD_COND_INITIALIZER;

// Iniciação dinâmica
pthread_cond_t mycondv;
int pthread_cond_init (pthread_cond_t *cond, NULL);

// Primitivas
int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *attr)
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)
int pthread_cond_signal (pthread_cond_t *cond)
int pthread_cond_broadcast(pthread_cond_t *cond)
```



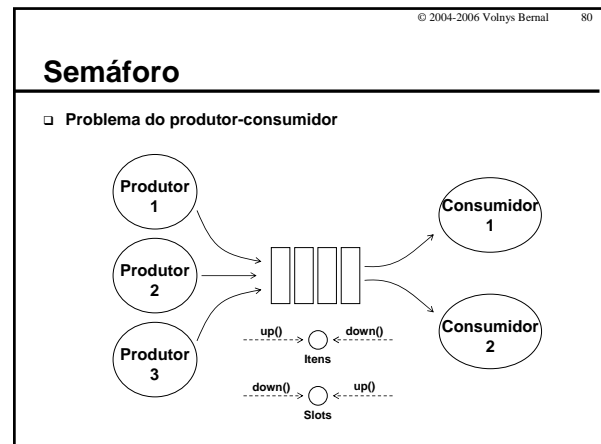
© 2004-2006 Volnys Bernal 79

Semáforo

❑ Problema do produtor-consumidor

```
semaforo mutex = 1;
semaforo slots = N;
semaforo itens = 0;
```

Produtor	Consumidor
Repetir	Repetir
Produzir(E);	Down(itens);
Down(slots);	Down(mutex);
Down(mutex);	E=RetirarFila(F);
InserirFila(F,E);	Up(mutex);
Up(mutex);	Up(slots);
Up(itens);	Consumir(E);



© 2004-2006 Volnys Bernal 81

Semáforo

❑ Primitivas pthreads

```
#include <semaphore.h>

int sem_init (sem_t *sem, int pshared, unsigned int value)
int sem_wait (sem_t * sem)
int sem_trywait (sem_t * sem)
int sem_post (sem_t * sem)
int sem_getvalue(sem_t * sem, int * sval)
int sem_destroy (sem_t * sem)
```

© 2004-2006 Volnys Bernal 82

Semáforo

❑ Exemplo de uso

```
#include <semaphore.h>

...
sem_t slots;
...
status = sem_init(&slots,0,10);
...
status = sem_wait(&slots);
...
status = sem_post(&slots);
...
```

© 2004-2006 Volnys Bernal 83

Semáforo Binário

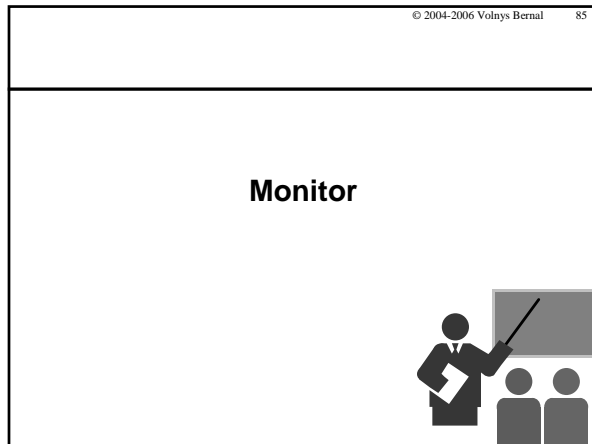
© 2004-2006 Volnys Bernal 84

Semáforo Binário

❑ Caso particular de semáforo no qual é iniciado com valor 1 e cujo valor nunca ultrapassa 1

❑ Pode ser utilizado para implementação de exclusão mútua:

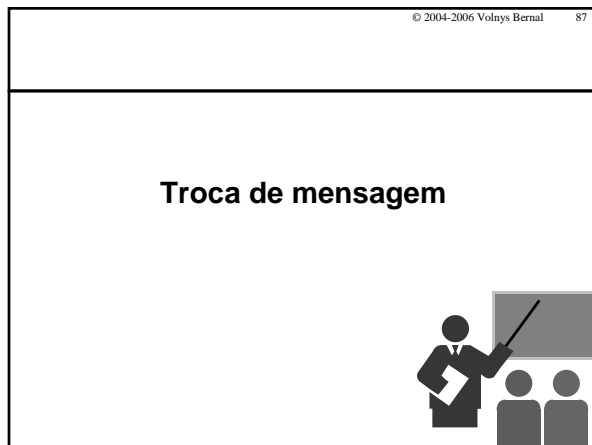
```
❖ lock() == down(semáforo_binário)
❖ unlock() == up(semáforo_binário)
```



© 2004-2006 Volnys Bernal 86

Monitor

- ❑ Conjunto de funções agrupadas em um pacote especial, como uma caixa preta.
- ❑ As estruturas internas manipuladas pela funções do monitor não são visíveis.
- ❑ Existe um mecanismo de sincronização de alto nível que garante que somente uma entidade (processo ou thread) pode adentrar no monitor em um determinado momento.



© 2004-2006 Volnys Bernal 88

Troca de mensagem

- ❑ Mecanismo muito utilizado para sincronização e comunicação entre processos
- ❑ Primitivas
 - ❖ Send(destination, message)
 - ❖ Receive(source, mensagem)
- ❑ Tipos de primitivas
 - ❖ Sincrona
 - Send() fica bloqueado até a entidade parceira ativar o Receive()
 - Receive fica bloqueado até a entidade parceira ativar Send()
 - Não necessita de buffers temporários
 - ❖ Assíncrona
 - Send() não é bloqueante
 - Receive é bloqueante
 - Necessita de gerenciamento de buffers temporários