


© 2004-2010 Volnys Bernal 1

Sincronização e Comunicação entre Processos

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




© 2004-2010 Volnys Bernal 2

Tópicos

- Escalonamento de entidades de processamento
- Classificação dos mecanismos de sincronização quanto à espera
- Mecanismos de sincronização e comunicação
 - ❖ Exclusão Mútua (Mutex)
 - ❖ Primitivas explicitamente bloqueantes
 - Sleep & Wakeup
 - Wait & Signal
 - ❖ Semáforo
 - ❖ Monitor
 - ❖ Troca de mensagens

© 2004-2010 Volnys Bernal 3

Escalonamento de entidade de processamento



© 2004-2010 Volnys Bernal 4

Escalonamento

- Escalonamento
 - ❖ Termo técnico atribuído à atividade de escolha da entidade de processamento (processo/thread) a ser executada pelo processador.
- Algoritmos de escalonamento
 - ❖ Os algoritmos de escalonamento baseiam-se em propriedades das entidades de processamento (processos/threads):
 - Prioridade (estática ou dinâmica)
 - Tempo de CPU consumido recentemente
 - Entidade de processamento preemptível ou não preemptível
 - Etc.


© 2004-2010 Volnys Bernal 5

Escalonamento

- Classes de entidade de processamento
 - ❖ Preemptível
 - Quando o escalonamento da entidade de processamento (processo/thread) puder ocorrer a qualquer momento
 - ❖ Não preemptível
 - Quando o escalonamento da entidade de processamento (processo/thread) puder ocorrer somente quando a entidade for bloqueada ou quando for ativada a primitiva yield()

© 2004-2010 Volnys Bernal 6

Classificação dos mecanismos de sincronização quanto à espera




© 2004-2010 Volnys Bernal 7

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
 - ❖ **Utilização**
 - Utilizada tipicamente em aplicações paralelas (multiprocessamento) em situações com sincronização rápida
- **Bloqueante**
 - ❖ Não desperdiça tempo de CPU
 - ❖ Quando em modo usuário requer a ativação de uma chamada ao sistema
 - ❖ **Problema**
 - Sobrecarga (custo computacional) da chamada ao sistema e da troca de contexto
 - ❖ **Utilização**
 - Utilizada nos casos gerais

© 2004-2010 Volnys Bernal 8

Mecanismos de sincronização e comunicação




© 2004-2010 Volnys Bernal 9

Mecanismos de Sincronização e comunicação

- **Tópicos**
 - ❖ **Mutex**
 - ❖ **Primitivas explicitamente bloqueantes**
 - Sleep & Wakeup
 - Wait & Signal
 - ❖ **Semáforo**
 - ❖ **Monitor**
 - ❖ **Troca de mensagens**

© 2004-2010 Volnys Bernal 10

Exclusão Mútua (Mutex)



© 2004-2010 Volnys Bernal 11


Exclusão Mútua (Mutex)

- **Objetivo:**
 - ❖ Técnica de sincronização que possibilita assegurar o acesso exclusivo (leitura e escrita) a um recurso compartilhado por duas ou mais entidades
- **Utilidade**
 - ❖ Prevenção de problema de condição de disputa em regiões críticas

© 2004-2010 Volnys Bernal 12

Exclusão Mútua (Mutex)

- **Exemplo:**
 - ❖ T1 – A entra 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



© 2004-2010 Volnys Bernal 13

Exclusão Mútua (Mutex)

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

© 2004-2010 Volnys Bernal 14

Exclusão Mútua (Mutex)

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

Região Crítica com exclusão mútua

© 2004-2010 Volnys Bernal 15

Exclusão Mútua (Mutex)

- **Exemplo:**
 - ❖ Solução do problema do contador

Thread1:

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

Thread2:

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

© 2004-2010 Volnys Bernal 16

Exclusão Mútua (Mutex)

- **Como implementar as primitivas de exclusão mútua**
 - ❖ **Instrução básica para implementação de Exclusão Mútua:**
 - Instrução Test-And-Set (TST)

© 2004-2010 Volnys Bernal 17

Instrução Test-And-Set-Lock

© 2004-2010 Volnys Bernal 18

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**

© 2004-2010 Volnys Bernal 19

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

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

© 2004-2010 Volnys Bernal 20

Interface de Mutex em Pthreads



© 2004-2010 Volnys Bernal 21

Interface de Mutex em Pthreads


❑ Primitivas pthreads

```

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

© 2004-2010 Volnys Bernal 22

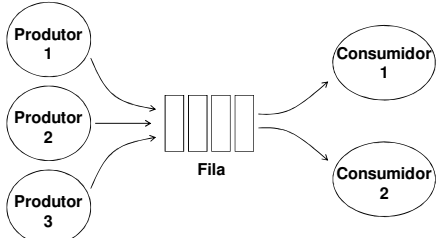
Exercício



© 2004-2010 Volnys Bernal 23

Exercício

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



© 2004-2010 Volnys Bernal 24

Exercício

❑ Primeiro esboço de solução:

```

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

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

© 2004-2010 Volnys Bernal 25

Exercício

□ **Identificação das necessidades de sincronização de espera por recursos:**

Produtor → slots livres
problema: quando não existem slots livres na fila

Consumidor → itens produzidos
problema: quando não existem itens produzidos na fila

© 2004-2010 Volnys Bernal 26

Exercício

□ **Identificação dos pontos de esperar por recursos**

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

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

(1) Fila cheia: o recurso "Fila" é limitado, ou seja, a fila pode tornar-se cheia. Nesta situação (de fila cheia) os produtores devem aguardar a existência de slots livres.

(2) Os consumidores podem ser mais rápidos que os produtores permitindo que em determinados momentos a fila fique vazia. Nesta situação (fila vazia), os consumidores dem aguardar a chegada de itens.

© 2004-2010 Volnys Bernal 27

Exercício

□ **Refinamento ...**

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

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

© 2004-2010 Volnys Bernal 28

Exercício

□ **Alteração do programa (sem levar em consideração eventuais condições de disputa)**

Produtor:
Repetir
 Produzir (E);
 Enquanto FilaCheia (F) ← *Condição de disputa*
 Aguardar;
 InserirFila (F, E); ← *Condição de disputa*

Consumidor:
Repetir
 Enquanto FilaVazia (F) ← *Condição de disputa*
 aguardar;
 E = RetirarFila (F); ← *Condição de disputa*
 Processar (E);

© 2004-2010 Volnys Bernal 29

Exercício

□ **Refinamento ...**

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);

© 2004-2010 Volnys Bernal 30

Exercício

□ **Problemas !!!**

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

© 2004-2010 Volnys Bernal 31

Exercício

□ Refinamento ...

```


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

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

```

© 2004-2010 Volnys Bernal 32

Primitivas Explicitamente Bloqueantes



© 2004-2010 Volnys Bernal 33

Primitivas Explicitamente Bloqueantes

- Também denominadas de
 - ✦ Primitivas de sincronização por variáveis de condição
- Utilização:
 - ✦ Primitivas voltadas principalmente ao gerenciamento de recursos
- Duas classes principais:
 - ✦ Sleep & Wakeup
 - Para ambiente:
 - não preemptível e
 - sistemas monoprocessoadores
 - Método de sincronização geralmente utilizado no núcleo do sistema operacional (Ex: UNIX)
 - ✦ Wait & Signal
 - Utilizado geralmente em processos ou threads de usuário

© 2004-2010 Volnys Bernal 34


Primitivas Explicitamente Bloqueantes

- Resumo das primitivas

Primitiva	Pré-condição	Local típico de utilização
Sleep & Wakeup	Ambiente não preemptível e monoprocessador	No núcleo de sistemas operacionais (modo supervisor)
Wait & Signal		Aplicações (modo usuário)

© 2004-2010 Volnys Bernal 35

Sleep & Wakeup



© 2004-2010 Volnys Bernal 36

Sleep & Wakeup

- Solução de sincronização
 - ✦ Bloqueante
 - ✦ Voltada para sincronização por recursos
 - ✦ Utiliza uma variável de condição
 - ✦ Pressupõe um ambiente não preemptível e monoprocessador.
 - ✦ Obs: Nos ambientes não preemptíveis, a troca de contexto sempre é realizada de maneira explícita pela primitiva yield() ou quando ocorre o bloqueio de um thread (por sleep)
- Utilização típica:
 - ✦ Utilizada no núcleo do sistema operacional (Ex: UNIX tradicional)
- Primitivas
 - ✦ Sleep(evento)
 - Bloqueia a entidade de processamento (processo ou thread) até a ocorrência do evento determinado
 - ✦ Wakeup(evento)
 - Desbloqueia todas as entidades de processamento que aguardam por um determinado evento. Neste momento, todas as entidades de processamento que aguardam por aquele evento são desbloqueadas.

© 2004-2010 Volnys Bernal 37

Sleep & Wakeup

□ **Exemplo – Problema produtor-consumidor**

- ❖ **Solução do problema produtor-consumidor, para somente 1 produtor e 1 consumidor, com primitivas 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

© 2004-2010 Volnys Bernal 38

Sleep & Wakeup

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

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

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

© 2004-2010 Volnys Bernal 39

Sleep & Wakeup

□ **No exemplo anterior observe que existem duas situações importantes:**

- ❖ **Quando a fila está cheia:**
 - O produtor, quando possuir um item para armazenar, é bloqueado (sleep) pois não existe espaço para armazenamento de "itens".
 - Assim, quando o consumidor retirar um item da fila e 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
 - Assim, quando o produtor produzir um item, desbloqueia (wakeup) o consumidor

© 2004-2010 Volnys Bernal 40

Exercício

**(9) Observe que a variável "count" é compartilhada!
Não existiria o problema de condição de disputa?**

© 2004-2010 Volnys Bernal 41

Exercício

**(10) Observe que a variável "count" é compartilhada!
Não existiria o problema de condição de disputa?**

- ❖ **Resposta:** Não, pois não ocorre a troca de contexto a qualquer momento. O ambiente é não preemptível. Assim, a troca de contexto ocorre somente em duas situações: quanto é ativada a primitiva yield() ou quando o *thread* é explicitamente bloqueado através da primitiva sleep().

© 2004-2010 Volnys Bernal 42

Exercício

(11) A solução apresentada anteriormente para o problema produtor-consumidor funciona somente para 1 produtor e 1 consumidor.

Porque?

Modifique o programa de forma a possibilitar o funcionamento com P produtores e C consumidores.

© 2004-2010 Volnys Bernal 43

Sleep & Wakeup


```

#define N 100
int count = 0;

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

© 2004-2010 Volnys Bernal 44

Wait & Signal



© 2004-2010 Volnys Bernal 45

Wait & Signal

- ❑ Solução de sincronização
 - ❖ Bloqueante
 - ❖ Voltada para sincronização por recursos
 - ❖ Necessita de uma variável de condição
 - ❖ Pressupõe um ambiente preemptível (quando existe troca de contexto por interrupção de relógio)
- ❑ Utilização típica
 - ❖ Em processos/threads executados em modo usuário
- ❑ Primitivas
 - ❖ Wait(evento)
 - Bloqueia a entidade de processamento (processo ou thread) até a ocorrência de um determinado evento
 - ❖ Signal(evento)
 - Desbloqueia todas as entidades de processamento que aguardam por um determinado evento. Neste momento, todas as entidades de processamento que aguardam por aquele evento são desbloqueadas.

© 2004-2010 Volnys Bernal 46

Wait & Signal

- ❑ Pthreads
 - ❖ Primitivas

Primitiva	Descrição
pthread_cond_init	Iniciação da variável de condição
pthread_cond_wait	Bloqueia o thread na condição
pthread_cond_signal	Caso existam threads bloqueados pela condição, desbloqueia 1 destes threads
pthread_cond_broadcast	Caso existam threads bloqueados pela condição, desbloqueia todos os estes threads
pthread_cond_destroy	Destroi uma variável de condição
 - ❖ Tipos de dados

Tipo	Descrição
pthread_cond_t	Representa o tipo de uma variável de condição. Para cada condição que possa levar a bloqueio deve ser criada uma variável de condição.

© 2004-2010 Volnys Bernal 47

Wait & Signal

- ❑ Pthreads - sintaxe


```

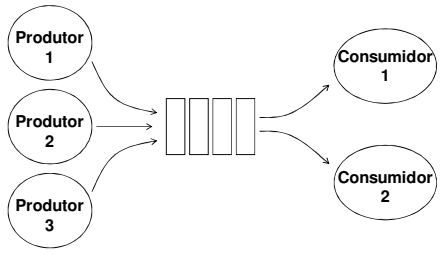
// Declaração da variável de condição "mycondv"
pthread_cond_t  mycondv;

// Declaração da variável de condição "mycondv" pré inicializada
pthread_cond_t  mycondv = PTHREAD_COND_INITIALIZER;

// 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)
int pthread_cond_destroy  (pthread_cond_t *cond)
            
```

© 2004-2010 Volnys Bernal 48

Wait & Signal

- ❑ Exemplo: Solução para o problema do produtor-consumidor
 

© 2004-2010 Volnys Bernal 49

Wait & Signal

❑ Problema do produtor-consumidor

```

Produtor()
{
  repetir
  {
    produzir(E);

    // Inserir na fila
    lock(mutex);
    enquanto FilaCheia(F)
      wait(FilaCheia,mutex);
    inserirFila(F,E);
    signal(FilaVazia);
    unlock(mutex);
  }
}

Consumidor()
{
  repetir
  {
    // Retirar da fila
    lock(mutex);
    enquanto FilaVazia(F)
      wait(FilaVazia,mutex);
    E = RetirarFila(F);
    signal(FilaCheia);
    unlock(mutex);

    Processar(E);
  }
}
    
```

© 2004-2010 Volnys Bernal 50

Wait & Signal

❑ Exemplo: Solução para o problema do produtor-consumidor

© 2004-2010 Volnys Bernal 51

Wait & Signal

❑ Exemplo: Programa worker

```

#define TCOUNT 10
#define COUNT_LIMIT 12

int count = 0;
mutex_t mutex;
cond_t threshold;

void worker()
{
  int i;
  for (i=0; i < TCOUNT; i++)
  {
    worker_processing();
    lock(mutex);
    count++;
    if (count == COUNT_LIMIT)
      signal(threshold);
    unlock(mutex);
  }
}

void extra_worker()
{
  lock(mutex);
  if (count < COUNT_LIMIT)
    wait(threshold,mutex);
  unlock(mutex);
  extra_processing();
}

int main()
{
  mutex_init(&mutex);
  cond_init(&threshold);
  create_thread(worker);
  create_thread(worker);
  create_thread(extra_worker);
  create_thread(extra_worker);
  join_threads();
}
    
```

© 2004-2010 Volnys Bernal 52

Semáforo

© 2004-2010 Volnys Bernal 53

Semáforo

- ❑ Método de sincronização
- ❑ Primitivas
 - ❖ Up(semáforo)
 - Incrementa o contador do semáforo.
 - Se existirem entidades bloqueadas neste semáforo, uma delas é desbloqueada
 - ❖ Down(semáforo)
 - Decrementa o semáforo
 - Se o resultado for menor que zero, a entidade fica bloqueada neste semáforo.
 - ❖ Init(semáforo,valor)
- ❑ Estas primitivas são garantidamente atômicas (indivisíveis)
- ❑ O semáforo deve ser iniciado com um valor inteiro.
- ❑ Normalmente, este valor está associado à quantidade de recursos disponíveis.

© 2004-2010 Volnys Bernal 54

Semáforo

❑ Como alternativa às primitivas de exclusão mútua

```

semáforo mymutex = 1;
    
```

Região Crítica com exclusão mútua

© 2004-2010 Volnys Bernal 55

Semáforo

❑ Problema do produtor-consumidor

❖ Necessita 3 semáforos

- 1 para garantir a exclusão mútua
 - quando a fila estiver cheia (sem slots disponíveis)
- 1 para bloquear os produtores
 - quando a fila estiver vazia (sem itens na fila)

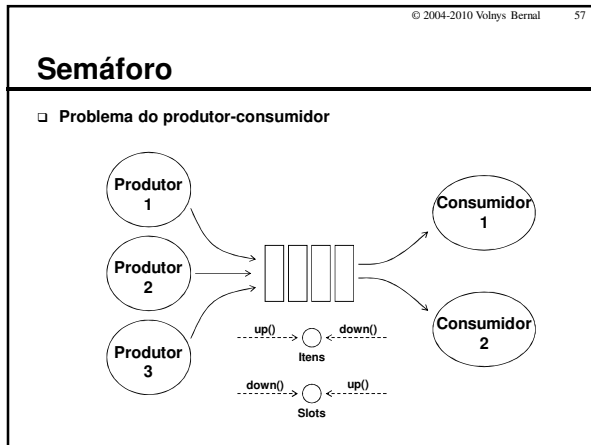
© 2004-2010 Volnys Bernal 56

Semáforo

❑ Problema do produtor-consumidor

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

<p>Produtor</p> <p>Repetir</p> <p> Produzir(E);</p> <p> Down(slots);</p> <p> Down(mutex);</p> <p> Down(mutex);</p> <p> InserirFila(F, E);</p> <p> Up(mutex);</p> <p> Up(itens);</p>	<p>Consumidor</p> <p>Repetir</p> <p> Down(itens);</p> <p> Down(mutex);</p> <p> E=RetirarFila(F);</p> <p> Up(mutex);</p> <p> Up(slots);</p> <p> Processar(E);</p>
--	---



© 2004-2010 Volnys Bernal 58

Semáforo

❑ Pthreads

❖ Tipos de dados

Tipo	Descrição
sem_t	Representa o tipo de um semáforo.

© 2004-2010 Volnys Bernal 59

Semáforo

❑ Pthreads

❖ Primitivas

Primitiva	Descrição
sem_init	Iniciação da variável de um semáforo.
sem_wait	Down. Decrementa o semáforo. Se o valor resultante for menor que zero a entidade de processamento é bloqueada.
sem_trywait	Variante de Down. Caso o valor do semáforo seja igual ou menor que zero, retorna. Caso contrário, decrementa o semáforo.
sem_post	Up. Incrementa o semáforo. Se existirem entidades de processamento bloqueadas neste semáforo, uma delas é desbloqueada.
sem_get_value	Retorna o contador do semáforo.
sem_destroy	Destrói um semáforo.

© 2004-2010 Volnys Bernal 60

Semáforo

❑ Pthreads - sintaxe

```
#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)
```

© 2004-2010 Volnys Bernal 61

Semáforo

❑ Exemplo de uso

```
#include <semaphore.h>
...
sem_t slots;
...
status = sem_init(&slots,0,10);
...
status = sem_wait(&slots);
...
status = sem_post(&slots);
...
```

© 2004-2010 Volnys Bernal 62


Exercício

(13) O que ocorre caso seja invertida a ordem de utilização dos semáforos no exercício anterior, ou seja:

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

© 2004-2010 Volnys Bernal 63

Semáforo Binário



© 2004-2010 Volnys Bernal 64

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)
```

© 2004-2010 Volnys Bernal 65

Monitor



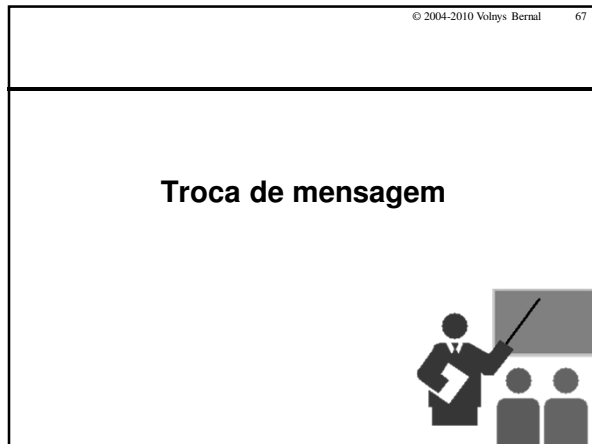
© 2004-2010 Volnys Bernal 66

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.



© 2004-2010 Volnys Bernal 68

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