
Primitivas de bloqueio explícito

Volnys Borges Bernal
volnys@lsi.usp.br

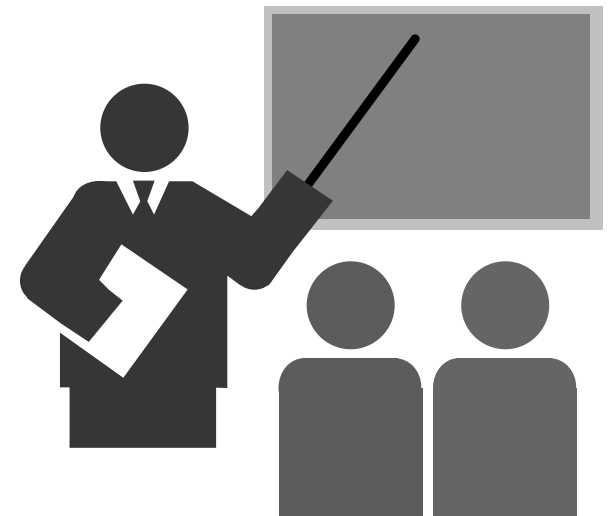
Depto. de Eng. de Sistemas Eletrônicos
Escola Politécnica da USP



Sumário

- **Primitivas de bloqueio explícito:**
 - ❖ **Primitivas Sleep & Wakeup**
 - ❖ **Primitivas Wait & Signal**

Primitivas de bloqueio explícito



Primitivas de bloqueio explícito

- ❑ Também denominadas de
 - ❖ Primitivas de sincronização por variáveis de condição

- ❑ Utilização:
 - ❖ Primitivas voltadas principalmente ao gerenciamento de recursos

- ❑ Duas classes principais:
 - ❖ **Sleep & Wakeup**
 - Utilizadas em ambiente não preemptível
 - Método de sincronização geralmente utilizado no núcleo do sistema operacional (Ex: UNIX)

 - ❖ **Wait & Signal**
 - Utilizado geralmente em processos ou threads de usuário

Primitivas de bloqueio explícito

□ Resumo das primitivas

Primitiva	Pré-condição	Local típico de utilização
Sleep & Wakeup	Ambiente não preemptível	Núcleo do sistema operacional
Wait & Signal	Ambiente preemptível	Aplicações (modo usuário)

- ❖ **Observação: nos ambientes não preemptíveis multiprocessadores existe a possibilidade de ocorrência de condição de disputa.**

Sleep & Wakeup



Sleep & Wakeup

- ❑ **Solução de sincronização**
 - ❖ Bloqueante
 - ❖ Voltada para sincronização por recursos
 - ❖ Necessita de uma variável de condição
 - ❖ Cuidado no uso em ambiente não preemptível e multiprocessadores
- ❑ **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.
- ❑ **Utilização clássica:**
 - ❖ Utilizada no núcleo do sistema operacional UNIX tradicional
- ❑ **Funcionamento**
 - ❖ Bloqueio: `sleep(evento)`
 - ❖ Desbloqueio: `wakeup(evento)`
 - ❖ Nos ambientes não preemptíveis, a troca de contexto sempre é realizada de maneira explícita pela primitiva `yield()` ou quando ocorre o bloqueio na chamada `sleep()`

Sleep & Wakeup

- **Exemplo: Solução do problema produtor-consumidor**
 - ❖ **Ambiente não preemptivo e monoprocessador**
 - ❖ **Solução válida para somente 1 produtor e 1 consumidor**
 - ❖ **Duas variáveis de condição**
 - **CondiçãoFilaCheia** – Para bloquear o produtor no caso de fila cheia
 - **CondiçãoFilaVazia** – 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

Sleep & Wakeup

- ❑ Ambiente não preemptivo e monoprocessador
- ❑ Solução válida para 1 produtor e 1 consumidor

```
#define N 100
```

```
int count = 0; //qde itens na fila
```

```
void producer(void)
```

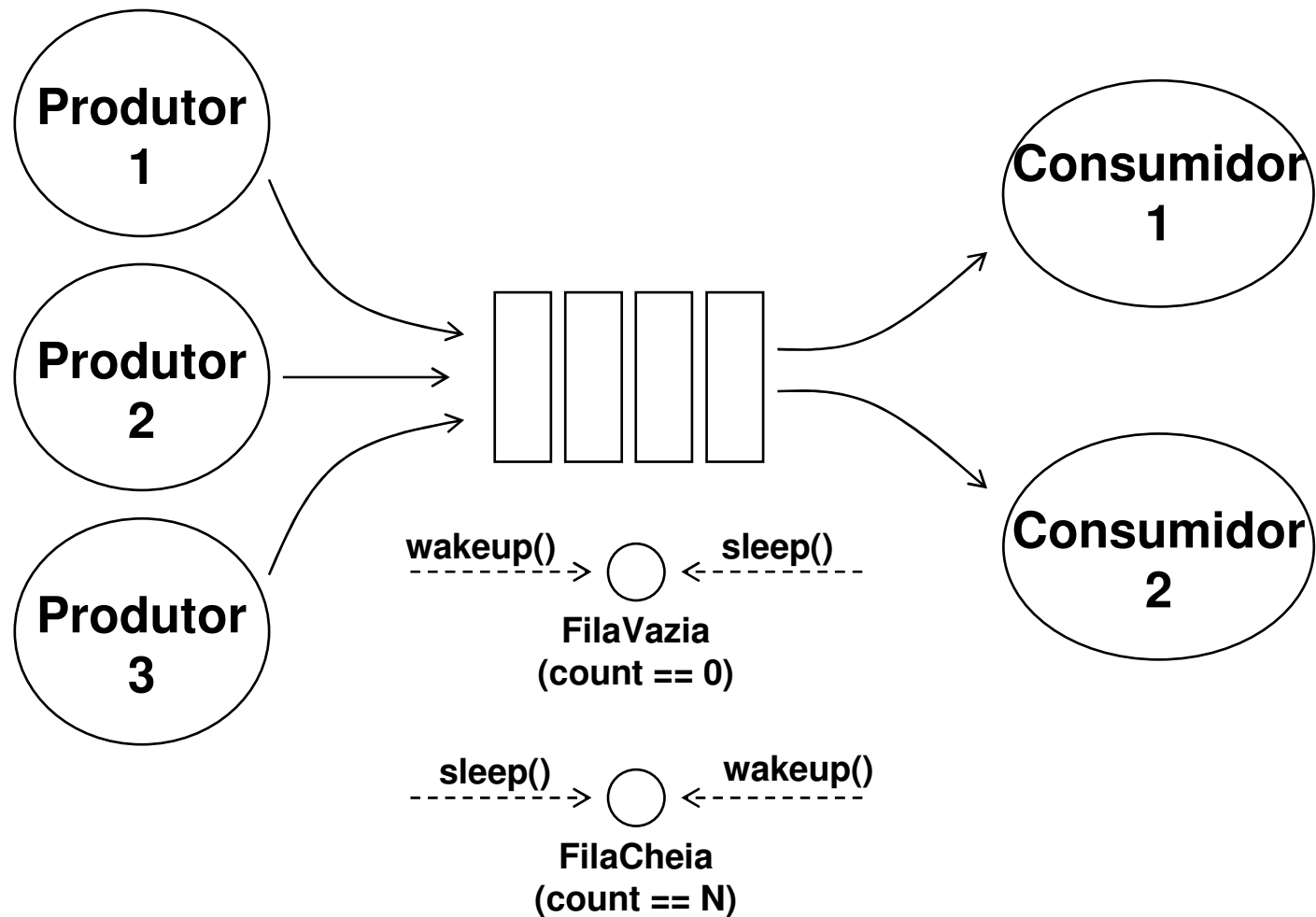
```
{
    int item;
    while (TRUE)
    {
        item = produce_item();
        if (count == N)
            sleep(filacheia);
        insert_item(item);
        count = count + 1;
        if (count == 1)
            wakeup(filavazia);
        yield();
    }
}
```

```
void consumer(void)
```

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

Sleep & Wakeup

- Exemplo: Solução para o problema do produtor-consumidor



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

Exercício

(9) Observe que a variável “count” é compartilhada!

- a) **Em um ambiente monoprocessador, não existiria o problema de condição de disputa?**
- b) **Em um ambiente multiprocessador, não existiria o problema de condição de disputa?**

Exercício

(9) Observe que a variável “count” é compartilhada!

a) Em um ambiente monoprocessador, 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: quando é ativada a primitiva `yield()` ou quando o *thread* é explicitamente bloqueado através da primitiva `sleep()`.**

b) Em um ambiente multiprocessador, não existiria o problema de condição de disputa?

❖ **Resposta: Sim.**

Exercício

- (10) A solução apresentada anteriormente para o problema produtor-consumidor funciona somente para 1 produtor e 1 consumidor. Porque?**
- (11) Modifique o programa de forma a possibilitar o funcionamento com P produtores e C consumidores.**

Sleep & Wakeup

- ❑ Ambiente não preemptivo e monoprocessador.
- ❑ Solução válida para N produtores e M consumidores

```
#define N 100
int count = 0; //qde itens na fila
```

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

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

Wait & Signal



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.

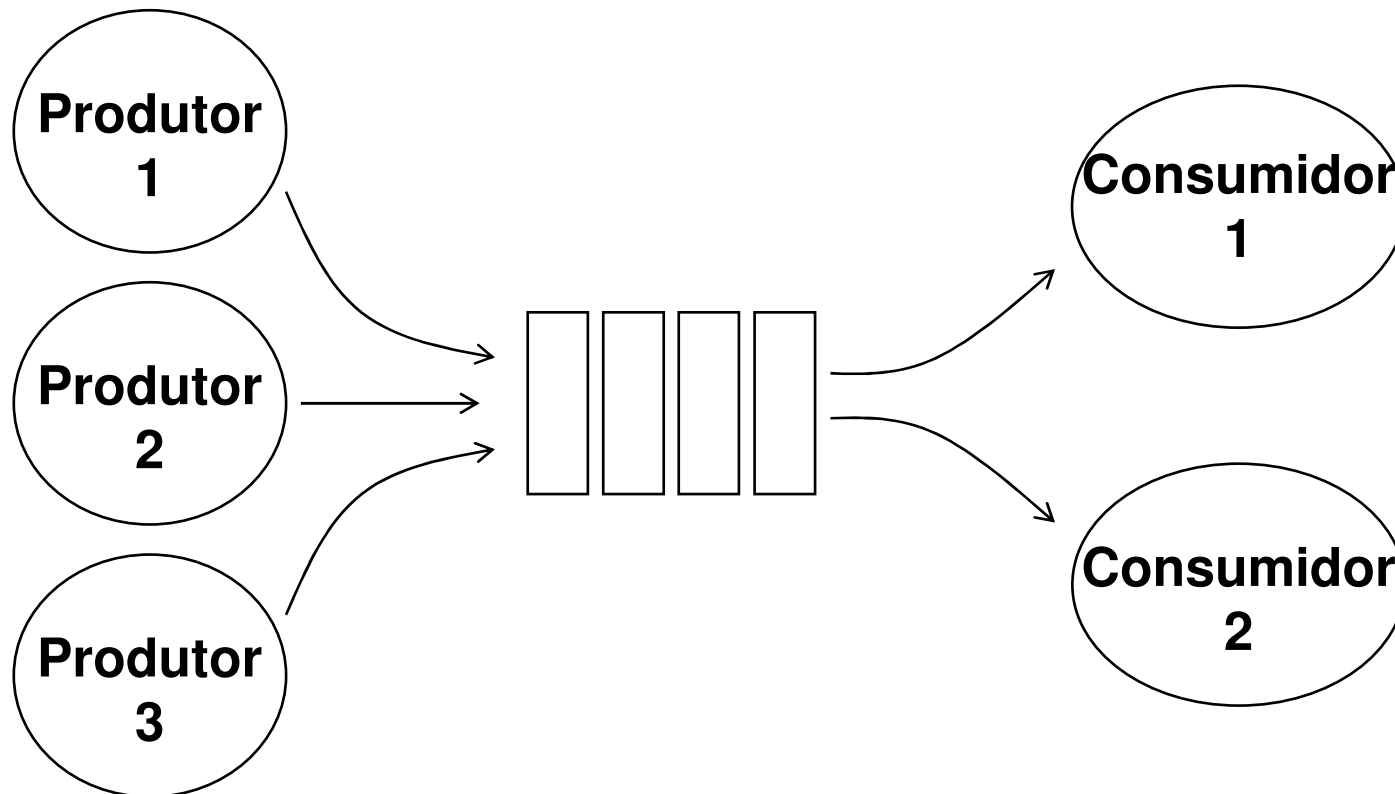
Wait & Signal

□ Integração com mutex

- ❖ É comum o uso de primitivas wait & signal em conjunto com mutex.
- ❖ Quando tais primitivas são usadas em conjunto, existe a possibilidade da entidade ser bloqueada no wait() estando dominando uma região crítica.
- ❖ Esta situação pode causar deadlock. Para evitar este problema, existem implementações que permite a liberação de um mutex no momento caso a entidade seja bloqueada pela ativação da primitiva wait()
- ❖ Primitiva: wait(evento, mutex)
 - Bloqueia a entidade de processamento (processo ou thread) até a ocorrência de um determinado evento.
 - Libera o mutex caso esteja dominando. Quando for desbloqueada, aguarda para obter novamente o mutex..

Wait & Signal

- Exemplo: Solução para o problema do produtor-consumidor



Wait & Signal

□ Problema do produtor-consumidor

Ambiente preemptível !

Produtor ()

```
{
repetir
{
    produzir(E);

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

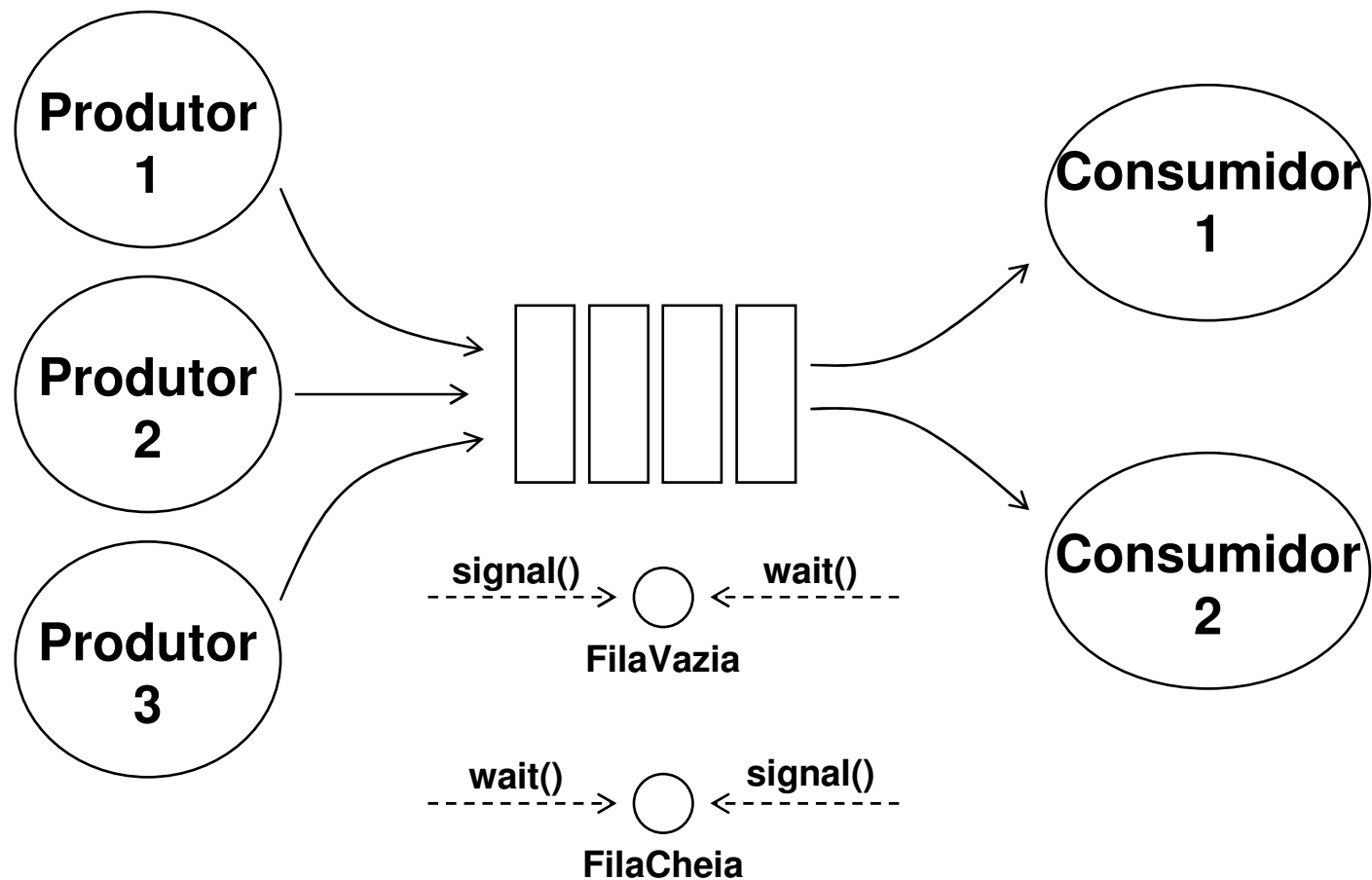
Consumidor ()

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

    Processar(E);
}
}
```

Wait & Signal

- Exemplo: Solução para o problema do produtor-consumidor



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(worker);
    create_thread(extra_worker);
    join_threads();
}
```

Wait & Signal

□ Pthreads

Primitiva	Descrição
<code>pthread_cond_init</code>	Iniciação da variável de condição
<code>pthread_cond_wait</code>	Bloqueia o thread na condição
<code>pthread_cond_signal</code>	Caso existam threads bloqueados pela condição, desbloqueia 1 destes threads
<code>pthread_cond_broadcast</code>	Caso existam threads bloqueados pela condição, desbloqueia todos os estes threads
<code>pthread_cond_destroy</code>	Destrói uma variável de condição

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

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)
```