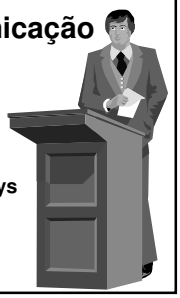
Sincronização e Comunicação entre Processos

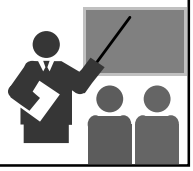
Volnys Borges Bernal
volnys@lsi.usp.br
http://www.lsi.usp.br/~volnys



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**

Escalonamento de entidade de processamento



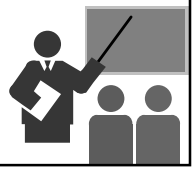
Escalonamento

- **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 baseam-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.

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()

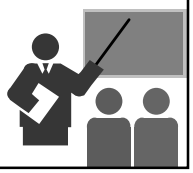
Classificação dos mecanismos de sincronização quanto à espera



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 caso gerais

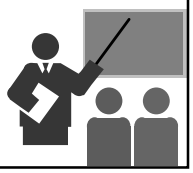
Mecanismos de sincronização e comunicação



Mecanismos de Sincronização e comunicação

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

Exclusão Mútua (Mutex)



Exclusão Mútua (Mutex)

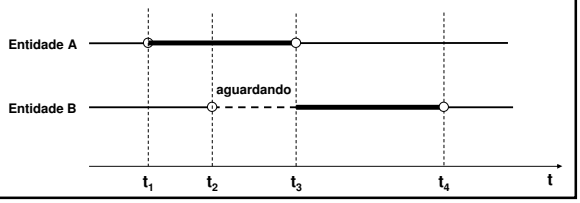
- **Tópicos**
  - **Exclusão Mútua (Mutex)**
    - Objetivo, utilidade, requisitos e primitivas
    - Alternativas para implementação de Exclusão Mútua
      - Implementação em software
        - Alternância obrigatória
        - Solução de Peterson
      - Implementação utilizando recursos de baixo nível
        - Desabilitar interrupção
        - Instrução Test-And-Set (TST)
    - Interface de Mutex em Pthreads

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**
- **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**

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 critica**

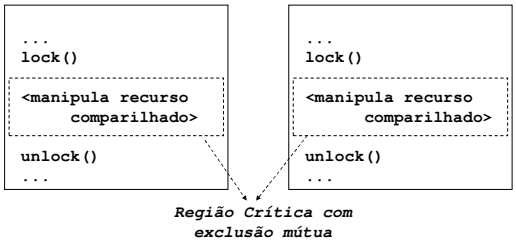


Exclusão Mútua (Mutex)

- **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

Exclusão Mútua (Mutex)

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



Exclusão Mútua (Mutex)

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

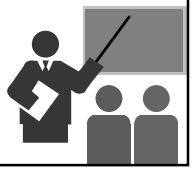
```
Thread1:

...

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

```
Thread2:

...

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


Exclusão Mútua (Mutex)

- **Alternativas para implementação de Exclusão Mútua**
  - **Implementação em software**
    - Alternância obrigatória
    - Solução de Peterson
  - **Implementação utilizando recursos de baixo nível**
    - Desabilitar interrupção
    - Instrução Test-And-Set (TST)

Mutex em software: Alternância Obrigatória



Alternância Obrigatória

- **Objetivo**
  - **Implementação de exclusão mútua**
- **Descrição**
  - **Alterna o acesso à região crítica entre duas entidades**
  - **Totalmente em software**
  - **Utiliza espera ociosa**
- **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**

Alternância Obrigatória

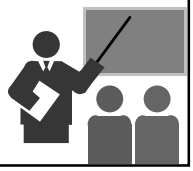
Entidade 1:

```
...
while (TRUE)
    {
    realiza outras atividades

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

Entidade 2:

```
...
while (TRUE)
    {
    realiza outras atividades

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


Mutex em software: Solução de Peterson



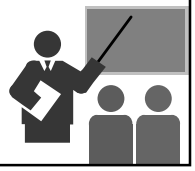
Solução de Peterson

- **Objetivo**
  - **Implementação de exclusão mútua**
- **Autoria**
  - **Publicada por G. L. Peterson em 1981**
  - **Baseada em uma solução do matemático holandês T. Dekker**
- **Descrição**
  - **Totalmente em software**
  - **Utiliza espera ociosa**
- **Desvantagem**
  - **Exemplo mostrado a seguir é válido somente para 2 entidades**

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;
    }
```


Mutex usando recursos de baixo nível: Desabilitar Interrupção



 25

Desabilitar Interrupção

- **Objetivo**
  - **Implementação de exclusão mútua para evitar condição de disputa decorrente de:**
    - Caso 1: concorrência entre threads ou processos
    - Caso 2: concorrência relacionada às rotinas de tratamento de interrupção
- **Método**
  - Caso 1: Desabilitar interrupção de relógio (impede da troca de contexto)
  - Caso 2: Desabilitar a ocorrência da interrupção específica
- **Problemas**
  - **Não aplicável para modo usuário**
  - **Possibilita que um erro do código (ex, loop) faça com que o sistema fique inoperante.**
  - **Para o caso 1, NÃO resolve o problema em sistemas multiprocessadores**
- **Utilidade**
  - **Método utilizado em porções de software executados em modo supervisor, tipicamente no núcleo do sistema operacional, para implementação de exclusão mútua em:**
    1. *Threads* internos ao núcleo do sistema operacional
    2. Drivers de dispositivos que utilizam interrupção

 26

Desabilitar interrupção

- **Exemplo:**

  - **lock()**

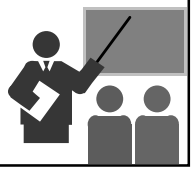
```
{
desabilita_interrupção_relogio;
}
```

  - **unlock()**

```
{
habilita_interrupção_relogio;
}
```

 27

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



 28

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 multirprocessadores é garantido que o acesso á 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**

 29

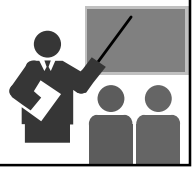
Instrução Test-And-Set-Lock

- **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
```

 30

Interface de Mutex em Pthreads



 31

Interface de Mutex em Pthreads

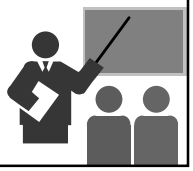
- **Primitivas pthreads**

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

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

// Primitivas
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)
```

-

 32

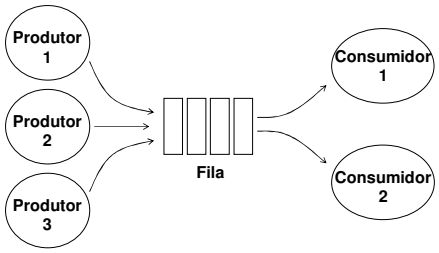
Exercício



 33

Exercício

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



 34

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);
```

 35

Exercício

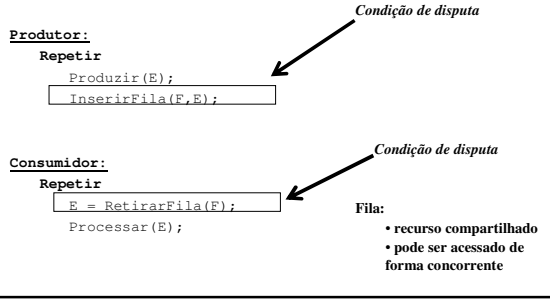
(2) Em relação ao problema do produtor-consumidor, analise o código, identifique as condições de disputa e defina as regiões críticas.

Dica: Inicie pela identificação dos recursos que são compartilhados entre as entidades.

 36

Exercício

- **Identificação das condições de disputa:**



Exercício

(3) Em relação ao problema do produtor-consumidor, identifique necessidades de sincronização de espera por recursos.
Quais são os recursos que podem eventualmente não estar disponível exigindo sua espera pela entidade?

Dica:

- *Identifique em quais situações a entidade deve aguardar por recursos estarem disponíveis. Estes recursos provavelmente são disputados pelas entidades!*
- *Neste caso específico, existem 2 recursos: um importante para os produtores e outro importante para os consumidores*

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

Exercício

(4) Altere o esboço do programa a fim de contornar o problema de espera por recursos (não se preocupe com condição de disputa, por enquanto).

Exercício

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

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

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

(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.

Exercício

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

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

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


Exercício

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

Exercício

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

```
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

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);
```


Exercício

(6) Em relação ao problema do produtor-consumidor, baseado no esboço da solução apresentada responda:

(a) O produtor, quando possui um item produzido e a fila está cheia o que ocorre?

(b) O consumidor, quando deseja retirar um item da fila e a fila está vazia o que ocorre?

(c) Qual é o problema que ocorre nestas situações no qual não existem recursos disponíveis?

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

Exercício

- Segundo esboço:

```
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);
     }
  }
```


Exercício

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

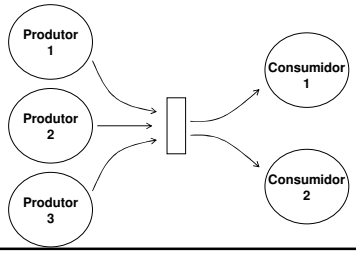
(a) A implementação funciona com múltiplos produtores e múltiplos consumidores?

(b) Suponha que o sistema seja monoprocessador. Qual tipo de primitiva é a mais recomendada: espera ociosa ou bloqueante? Porque?

(c) Suponha que o sistema seja multiprocessador. Qual tipo de primitiva é a mais recomendada: espera ociosa ou bloqueante? Porque?

Exercício

(8) O problema do produtor-consumidor pode ser implementado utilizando-se um único buffer sincronizado por primitivas de exclusão mútua.



Exercício

O programa prodcons_buffer.c mostra uma implementação da solução do problema do produtor-consumidor utilizando-se um único buffer sincronizado por primitivas de exclusão mútua.

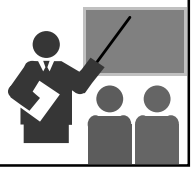
Compile e execute este programa

```
cc -o prodcons_buffer prodcons_buffer.c -lpthread
./prodcons_buffer
```

Analise o programa e responda:

(a) A solução apresentada resolve o problema de condição de disputa?
(b) Qual é a condição de término dos produtores?
(c) Qual é a condição de término dos consumidores?
(d) Proponha uma condição de término para os consumidores.

Problema da Inversão de Prioridade

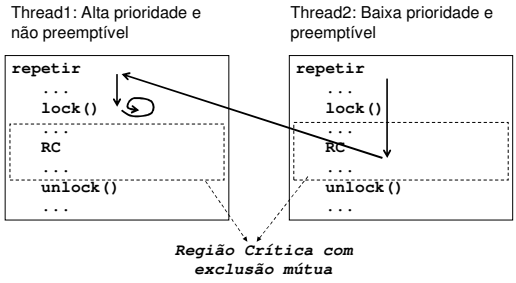


Problema de Inversão de Prioridade

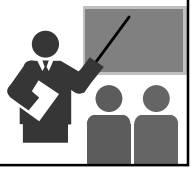
- **Descrição do problema**
  - **Ambiente**
    - Ambiente monoprocessador
    - Sistema com 2 threads:
      - Thread H – Thread de alta prioridade, não preemptivo
      - Thread L – Thread de baixa prioridade, preemptivo
    - Utilização de primitivas de exclusão mútua com espera ociosa
    - Escalonamento:
      - H sempre é executado quando está no estado pronto (ou seja, H tem preferência sobre L)
  - **Situação na qual ocorre o problema**
    - Thread L ganha a região crítica e thread H torna-se pronto
    - Thread H é escalonado e tenta ganhar a região crítica
  - **Resultado**
    - Deadlock

Problema de Inversão de Prioridade

- **Exemplo com possibilidade de *deadlock***



Primitivas Explicitamente Bloqueantes



Primitivas Explicitamente Bloqueantes

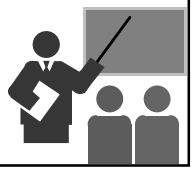
- Também denominadas de
  - Primitivas de sincronização por variáveis de condição
- Utilização:
  - Primitivas voltadas principalmente ao gerenciamento de recursos
- Duas classes principais:
  - Sleep & Wakeup
  - Wait & Signal
- Sleep & Wakeup
  - Utilizadas em ambiente não preemptível e sistemas monoprocessadores

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)

Sleep & Wakeup



Sleep & Wakeup

- Solução de sincronização
  - Bloqueante
  - Voltada para sincronização por recursos
  - Necessita de 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 explicita pela primitiva yield() ou quando ocorre o bloqueio de um thread (por sleep)
- Utilização típica:
  - Utilizada no núcleo do sistema operacional 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.

Sleep & Wakeup

- Exemplo
  - 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

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();
        }
     }
```

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! Não existiria o problema de condição de disputa?

Exercício

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

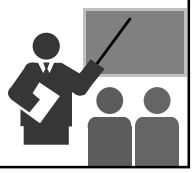
- **Resposta: Não, pois não ocorre a troca de contexto a qualquer momento. O ambiente é não preemptível. Assim, a troca de contexto ocorre somente em duas situações: quanto é ativada a primitiva yield() ou quando o *thread* é explicitamente bloqueado através da primitiva sleep().**

Exercício

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

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

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

- **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`	Destrói 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

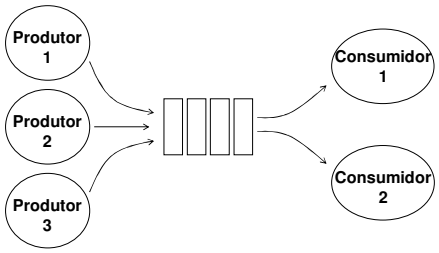
Wait & Signal

- **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 *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)
```


Wait & Signal

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



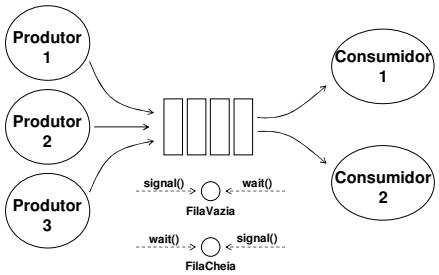
Wait & Signal

- **Problema do produtor-consumidor**

```
Produtor()                          Consumidor()
   {                                   {
   repetir                             repetir
      {                                   {
      produzir(E);                        // Retirar da fila
                                          lock(mutex);
      // Inserir na fila                  enquanto FilaVazia(F)
      lock(mutex);                            wait(FilaVazia,mutex);
      enquanto FilaCheia(F)               E = RetirarFila(F);
          wait(FilaCheia,mutex);          signal(FilaCheia);
      inserirFila(F,E);                   unlock(mutex);
      signal(FilaVazia);
      unlock(mutex);                      Processar(E);
      }                                   }
   }                                   }
```


Wait & Signal

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



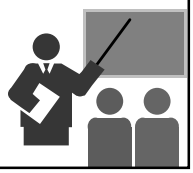
Wait & Signal

- **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();
   }
```

Semáforo



Semáforo

- **Método de sincronização**
- **Primitivas**
  - **Up(semaforo)**
    - Incrementa o contador do semáforo.
    - Se existirem entidades bloqueadas neste semáforo, uma delas é desbloqueada
  - **Down(semaforo)**
    - Decrementa o semáforo
    - Se o resultado for menor que zero, a entidade fica bloqueada neste semáforo.
  - **Init(semaforo,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.**

Semáforo

- **Problema do produtor-consumidor**
  - **Necessita 3 semáforos**
    - 1 para garantir a exclusão mútua
    - 1 para bloquear os produtores
      - quando a fila estiver cheia (sem slots disponíveis)
    - 1 para bloquear os consumidores
      - quando a fila estiver vazia (sem itens na fila)

Semáforo

- **Problema do produtor-consumidor**

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

Produtor

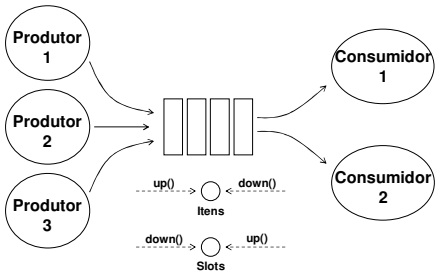
```
Repetir
    Produzir(E);
    Down(slots);
    Down(mutex);
    InserirFila(F,E);
    Up(mutex);
    Up(itens);
```

Consumidor

```
Repetir
    Down(itens);
    Down(mutex);
    E=RetirarFila(F);
    Up(mutex);
    Up(slots);
    Processar(E);
```


Semáforo

- **Problema do produtor-consumidor**



Semáforo

- **Pthreads**
  - **Tipos de dados**

Tipo	Descrição
`sem_t`	Representa o tipo de um semáforo.

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.

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)
```


Semáforo

- **Exemplo de uso**

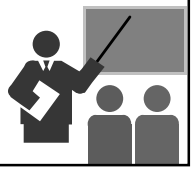
```
#include <semaphore.h>

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


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);
```

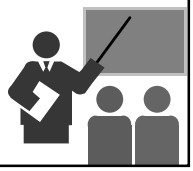

Semáforo Binário



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(semaforo_binário)**
  - **unlock() == up(semaforo_binário)**

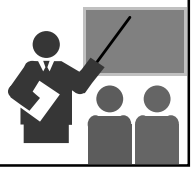
Monitor



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.**

Troca de mensagem



Troca de mensagem

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