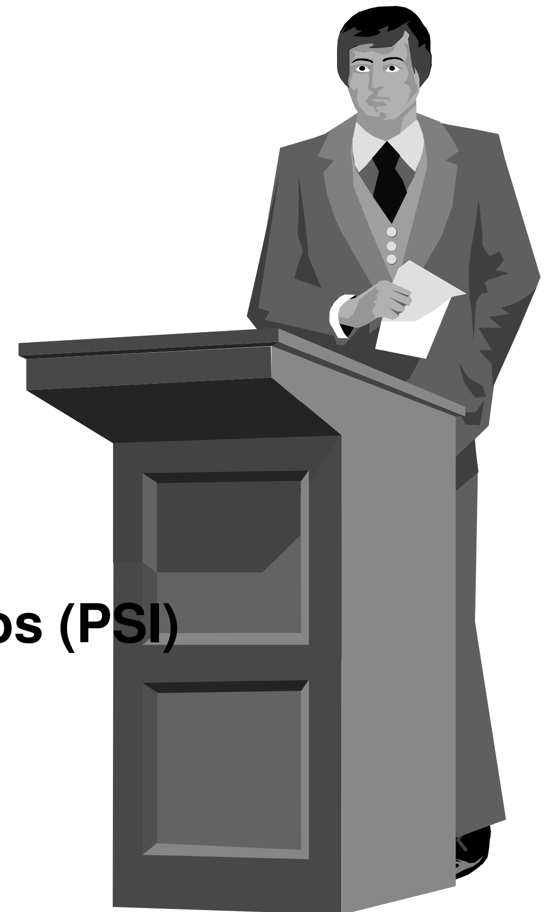
Exclusão Mútua
(*mutex*)



Volnys Borges Bernal
volnys@lsi.usp.br

Depto. de Eng. de Sistemas Eletrônicos (PSI)
Escola Politécnica da USP

Tópicos

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

Exclusão Mútua
(*mutex*)

Exclusão Mútua (Mutex)

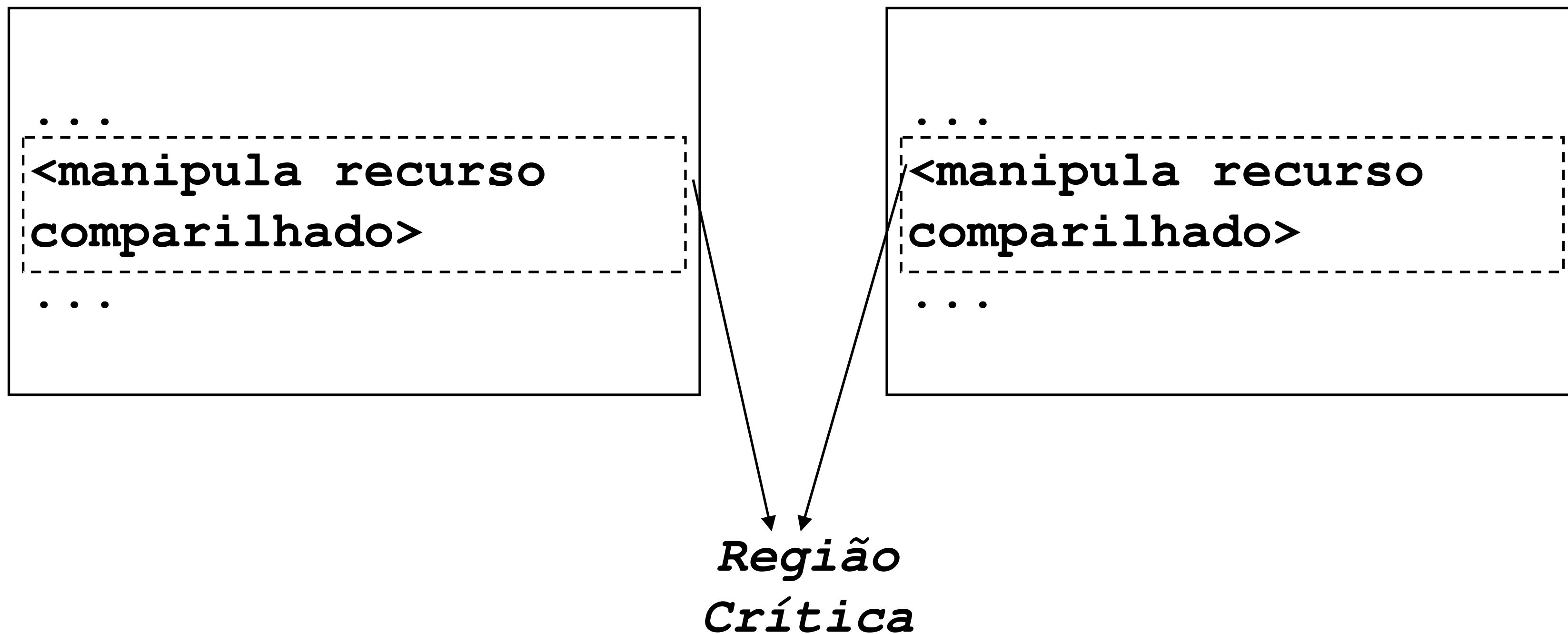
- **Objetivo:**
  - **Técnica de sincronização que possibilita assegurar o acesso exclusivo (leitura e escrita) a um recurso compartilhado por duas ou mais entidades de processamento**
- **Utilidade**
  - **Prevenção de problema de condição de disputa em regiões críticas**
- **Requisitos para a implementação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

- **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

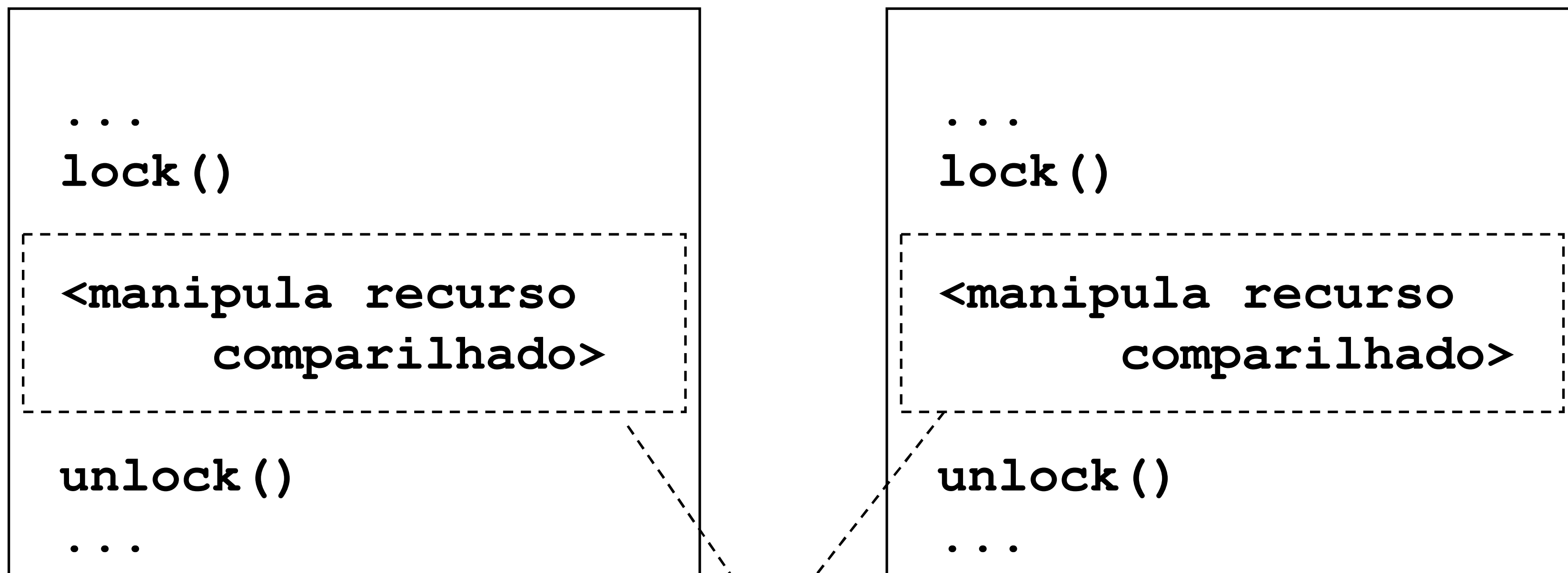
Região Crítica

- **Exemplo:**
  - **Região crítica sem proteção**



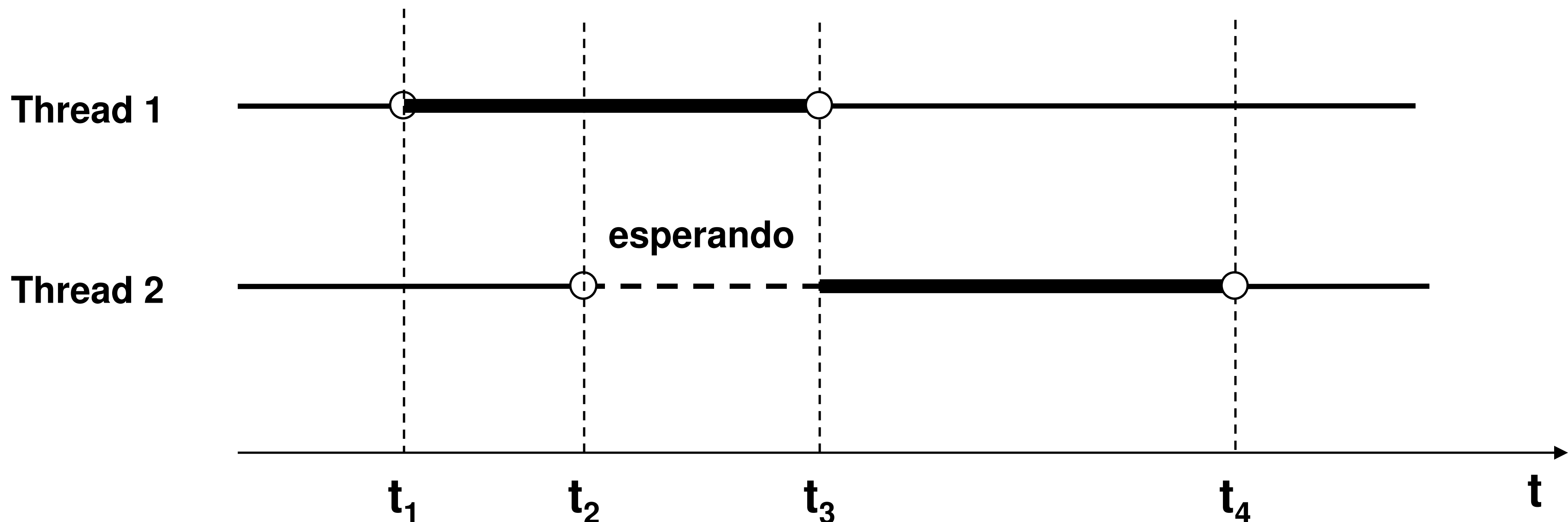
Exclusão Mútua (Mutex)

- **Exemplo:**
  - **Região crítica protegida com uso de mutex:**
    - lock( ) - obtém a exclusão mútua sobre a Região Crítica
    - unlock( ) - libera a exclusão mútua sobre a Região Crítica



Exclusão Mútua (Mutex)

- Exemplo:
  - t1 – Thread 1 entra na região crítica
  - t2 – Thread 2 tenta entrar na região crítica
  - t3 – Thread 1 A sai da região crítica;
    Thread 2 entra na região crítica
  - t4 – Thread 2 sai da região critica



Exclusão Mútua (Mutex)

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

```
Thread1:

...

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

```
Thread2:

...

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

Uso de mutex:

Interface de mutex em Pthreads

Interface de mutex em Pthreads

- Primitivas pthreads

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

// Primitiva de iniciação dinâmica
pthread_mutex_t mymutex;
int  pthread_mutex_init    (pthread_mutex_t *mymutex,
                                 pthread_mutexattr_t *attr);

// Demais primitivas
int  pthread_mutex_lock    (pthread_mutex_t *mymutex)
int  pthread_mutex_unlock  (pthread_mutex_t *mymutex)
int  pthread_mutex_trylock (pthread_mutex_t *mymutex)
```

Exercício

Exercício

(1) Modifique o programa "mythread.c" para proteger a variável "i" contra condição de disputa utilizando primitivas mutex pthreads.

Compile o programa "mythread.c" utilizando a biblioteca libpthread:

```
cc  -o  mythread  mythread.c  –lpthread
```

Execute o programa mythread e verifique o resultado da execução:

```
./mythread
```

```c
#include <pthread.h>
pthread_mutex_t mymutex;
int i=0;

imprimir_msg(char *nome)
    {
   while (i<10)
          {
          pthread_mutex_lock(&mymutex);
          printf("Thread %s - %d\n", nome, i);
          i++;
          pthread_mutex_unlock(&mymutex);
          sleep(2);
          }
   printf("Thread %s terminado \n", nome);
    }

int main()
    {
    pthread_t thread1;
    pthread_t thread2;
    pthread_mutex_init(&mymutex,NULL);
    printf("Programa de teste de pthreads \n");
    printf("Disparando primeiro thread\n");
    pthread_create(&thread1, NULL, (void*) imprimir_msg,"thread_1");
    printf("Disparando segundo thread\n");
    pthread_create(&thread2, NULL, (void*) imprimir_msg,"thread_2");
    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
    printf("Terminando processo");
    }
```

Implementação de Mutex

Implementação de Mutex

- **Alternativas para implementação de exclusão mútua:**
  - **Implementação em software**
    - <u>Não funcionam</u>
    - Ex: Alternância obrigatória, Solução de Peterson
  - **Implementação utilizando recursos de hardware**
    - Desabilitar interrupção
    - Instrução Test-And-Set (TST)

Desabilitar Interrupção

Desabilitar Interrupção

- **Objetivo**

Controlar exclusão mútua em porções de código executados em modo supervisor:

a) Entre threads do núcleo do sistema operacional (modo supervisor):
  - Proteger região crítica desabilitando a interrupção de relógio

b) Entre código do núcleo do sistema operacional (ou sistema embarcado) e rotina de tratamento de interrupção:
  - Proteger região crítica desabilitando a interrupção específica

Desabilitar interrupção

- Exemplo: Para condição de disputa entre rotina de tratamento de interrupção de código do device driver:

```
# lock()
desabilita_interrupcao(controlador);

# unlock()
habilita_interrupcao(controlador);
```

Desabilitar Interrupção

- **Comentários**
  - **O uso desta técnica está limitada às porções de código que executam em modo supervisor (onde existe permissão de acesso ao controlador de interrupção):**
    - Em ambientes operacionais de propósito geral: núcleo do sistema operacional
    - Em sistemas embarcados: maioria dos sistemas embarcados são executados em modo supervisor.

Desabilitar Interrupção

Exemplo de uso: Device driver

- **Exclusão mútua entre rotina de tratamento de interrupção e código do device driver**

- **Método**
  - Desabilitar a ocorrência da interrupção do dispositivo controlado pelo device driver durante a execução da região crítica

- **Problemas e limitações**
  - Em ambientes multiprocessadores deve ser utilizado em conjunto com mecanismos de espera ociosa com TST (spin locks).

- **Comentário**
  - Esta técnica é muito utilizada em device drivers em sistemas monoprocessadores

Desabilitar interrupção

- **Leitura complementar:**
  - **Proteção de região crítica em núcleo de sistema operacional em ambiente monoprocessador e multiprocessador**

Local Interrupt Disabling

Interrupt disabling is one of the key mechanisms used to ensure that a sequence of kernel statements is treated as a critical section. It allows a kernel control path to continue executing even when hardware devices issue IRQ signals, thus providing an effective way to protect data structures that are also accessed by interrupt handlers. By itself, however, local interrupt disabling does not protect against concurrent accesses to data structures by interrupt handlers running on other CPUs, so in multiprocessor systems, local interrupt disabling is often coupled with spin locks (see the later section "Synchronizing Accesses to Kernel Data Structures").

  - Daniel Pierre Bovet; Marco Cesati. **Understanding the Linux Kernel.** O´Reilly

Instrução Test-And-Set-Lock

Instrução Test-And-Set-Lock

- **Objetivo**
  - Primitiva de baixo nível para implementação de sincronização

- **Contexto de uso**
  - Mecanismo básico para implementação de primitivas de sincronização

- **Descrição**
  - Instrução especial de CPU que permite fazer "teste e bloqueio" em uma única instrução atômica.

Instrução Test-And-Set-Lock

- Detalhamento
  - **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

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
```

Problema da Inversão de Prioridade

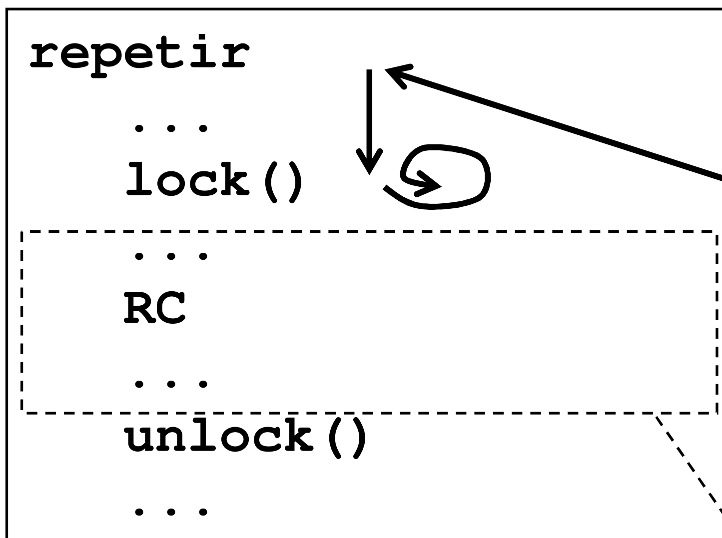
Problema de Inversão de Prioridade

- **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

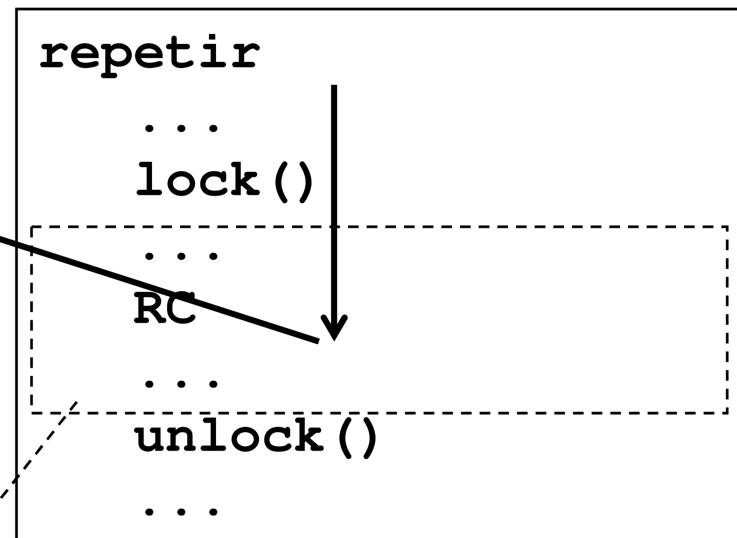
Problema de Inversão de Prioridade

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

Thread1: Alta prioridade e não preemptível



Thread2: Baixa prioridade e preemptível



Região Crítica com exclusão mútua

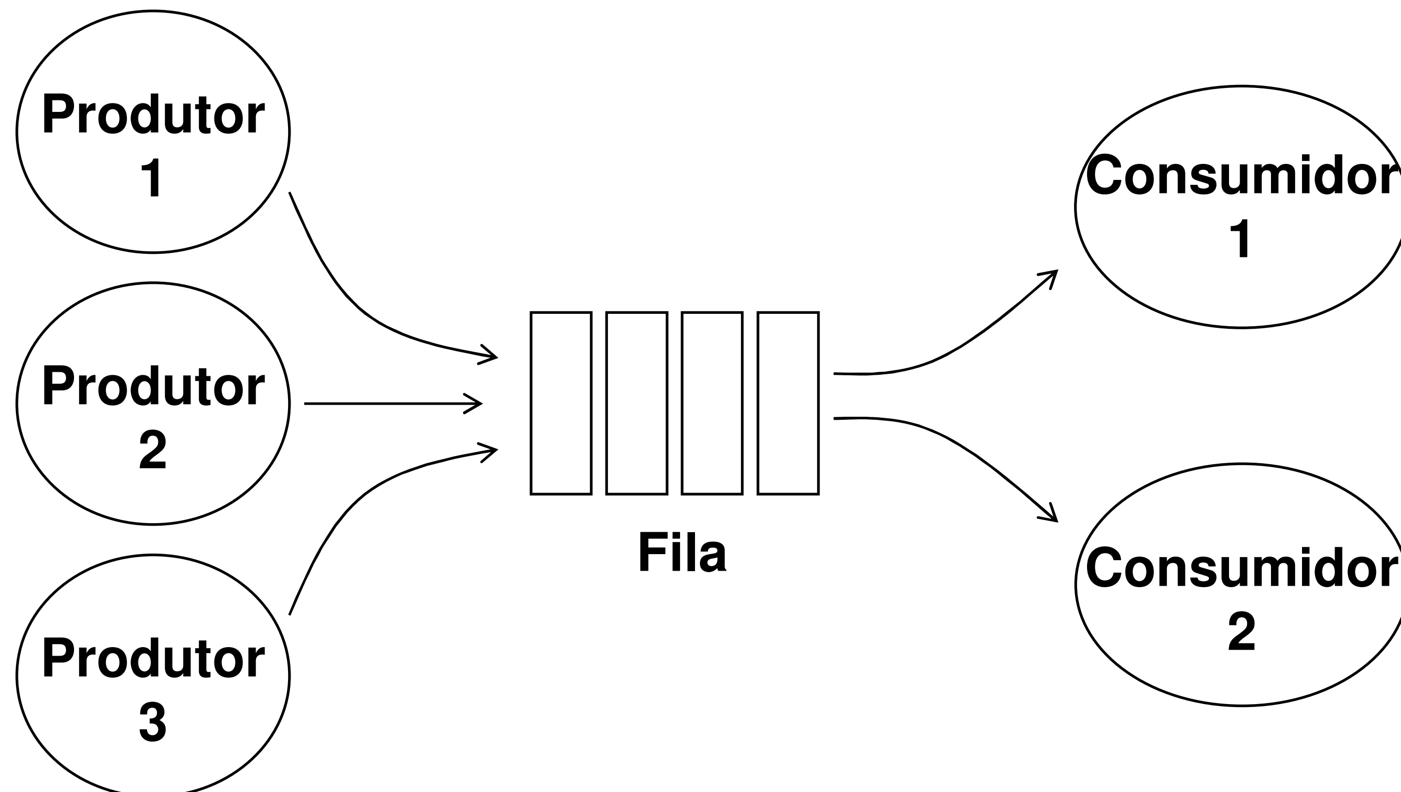
Exercícios

© 2004-2018 Volnys Bernal

Exercício

(4) Em relação ao problema do produtor-consumidor:

(a) Faça um esboço de solução do problema sem levar em consideração as condições de disputa existentes.



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);
```

Exercício

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

Dica: Identifique os recursos que são compartilhados entre as entidades.

Exercício

❖ **Identificação das regiões críticas:**

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

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

Fila:

- **recurso compartilhado**
- **pode ser acessada de forma concorrente**

Exercício

(c) Identifique necessidades de sincronização de espera por recursos.

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

- **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

- Necessidades de sincronização de espera 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.

Exercício

(d) 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

- **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

(e) Apresente uma solução para do acesso compartilhado aos recursos (fila) utilizando primitivas de exclusão mútua.

Exercício

- Alteração do programa para evitar condição 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

- Alteração do programa para evitar condição de disputa.

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

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

Região crítica R

Região crítica R

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

(f) Baseado no esboço da 1ª 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

(g) Baseado no esboço da 2ª solução apresentada responda:

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

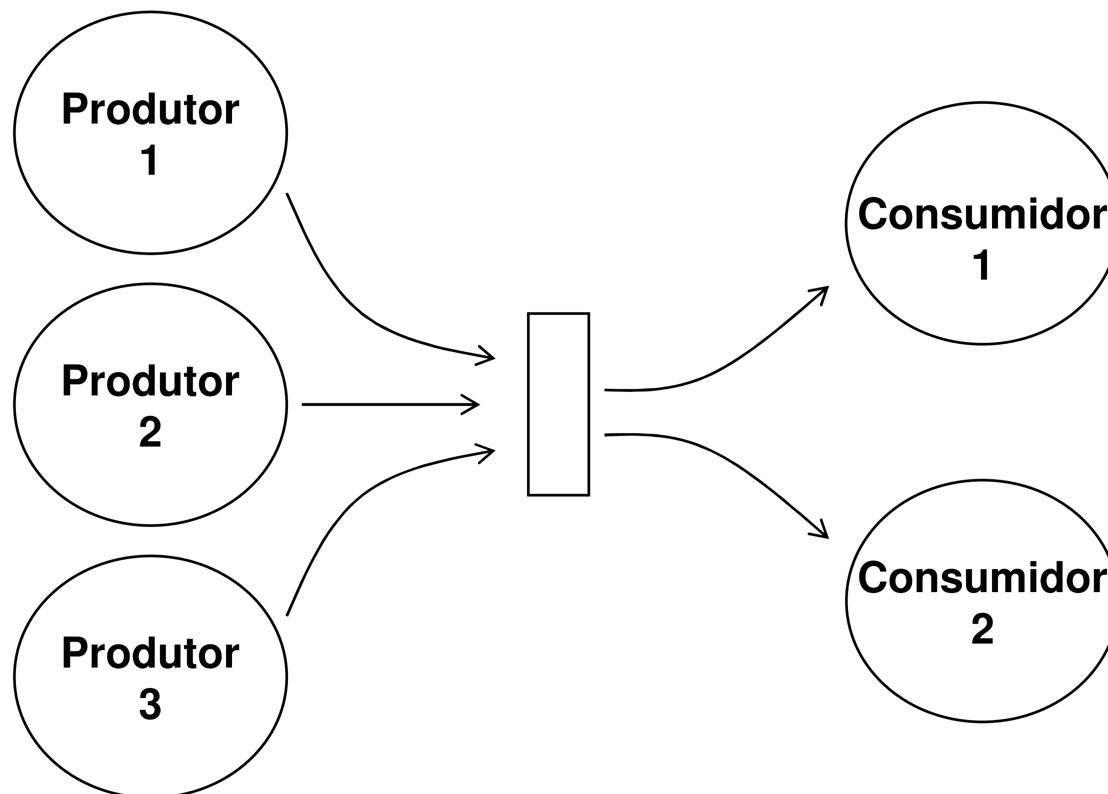
(b) Suponha que o sistema seja monoprocessador. Qual tipo de primitiva é a mais recomendada: espera ociosa ou bloqueante? Porque?

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

© 2004-2018 Volnys Bernal

Exercício

(5) 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.