Região Crítica

Prof. Paulo Roberto O. Valim
Universidade do Vale do Itajaí

Introdução

- Programas podem ser desenvolvidos para executar tarefas de forma cooperativa.
- Para que estas tarefas possam cooperar elas precisam trocar dados. Esta troca pode ser, por exemplo, através de memória compartilhada.
- Dependendo do tipo de operação que estas tarefas estão realizando sobre a memória compartilhada, problemas podem ocorrer.

Introdução

- **Programação concorrente**
  - Composta por um conjunto de processos sequenciais que se executam concorrentemente
  - Processos disputam recursos comuns
  - Um processo é dito cooperante quando ele é capaz de afetar, ou ser afetado, pela execução de outro processo.

Introdução

- Motivação da programação concorrente:
  - Aumento de desempenho
    - Permite a exploração de paralelismo real em máquinas multiprocessadas;
    - Sobreposição de operações de E/S com processamento
  - Facilidade de desenvolvimento de aplicações que possuem um paralelismo intrínseco

Introdução

- Desvantagens da programação concorrente:
  - Programação complexa
  - Aspecto não determinístico
    - Difícil depuração

A taxa na qual um processo executa sua computação não irá ser uniforme, e provavelmente nem mesmo reprodutível se o mesmo processo executar novamente

Processos cooperativos

- Processo independente: quando não pode afetar nem se afetado por outros processos em execução no sistema.
- Processo cooperativo: quando pode afetar ou ser afetado por outros processos em execução.
- Razões para oferecer ambiente que permita cooperação:
  - Compartilhamento de informação;
  - Processamento mais rápido;
  - Modularidade;
  - Comodidade.
- Requer algum mecanismo de comunicação entre processos, como por exemplo o compartilhamento de dados.
- O acesso concorrente a dados compartilhados podem resultar em inconsistência de dados.

O Problema do compartilhamento de recursos

- A programação concorrente implica em um compartilhamento de recursos
  - Variáveis compartilhadas são recursos essenciais para a programação concorrente
- Acessos a recursos compartilhados devem ser feitos de forma a manter um estado coerente e correto do sistema

Exemplo 1

Saída esperada

```
int s = 0; /* Variável compartilhada */
```

Thread 0		Thread 1	
(i)	`s = 0;`	(iii)	`s = 1;`
(ii)	`print ("Thr 0:  ", s);`	(iv)	`print ("Thr 1:  ", s);`

Saída: Thr 0: 0

Thr 1: 1

Exemplo 1

Saída esperada II

```
int s = 0; /* Variável compartilhada */
```

Thread 0		Thread 1	
(iii)	`s = 0;`	(i)	`s = 1;`
(iv)	`print ("Thr 0:  ", s);`	(ii)	`print ("Thr 1:  ", s);`

Saída: Thr 1: 1

Thr 0: 0

Exemplo 1

Saída inesperada

```
int s = 0; /* Variável compartilhada */
```

Thread 0		Thread 1	
(i)	`s = 0;`	(ii)	`s = 1;`
(iii)	`print ("Thr 0:  ", s);`	(iv)	`print ("Thr 1:  ", s);`

Saída: Thr 0: 1

Thr 1: 1

Exemplo 2 - Produtor - consumidor

- Exemplo clássico que para demonstrar o uso de memória compartilhada entre duas tarefas. Uma delas produz dados e os coloca em um buffer circular localizado em memória compartilhada. A outra tarefa consome os dados do buffer. Uma variável compartilhada controla o número de elementos no buffer (counter).

Produtor

```
while (true) {
/* produce an item in next produced */
        while (counter == BUFFER SIZE)
        ; /* do nothing */
        buffer[in] = next produced;
        in = (in + 1) % BUFFER SIZE;
        counter++;
}
```

Consumidor

```
while (true) {
        while (counter == 0)
        ; /* do nothing */
        next consumed = buffer[out];
        out = (out + 1) % BUFFER SIZE;
        counter--;
/* consume the item in next consumed */
}
```

Será que funciona?

Exemplo 2 - Produtor - Consumidor

- Vamos considerar que o valor da variável *counter* é 5 e que o produtor e o consumidor concorrentemente executam a linha *"counter++;"* e *"counter--;"*, respectivamente.
- O resultado esperado é que ao final o valor de *counter* continue sendo 5. Porém, quando estas duas linhas são executadas, o resultado em *counter* pode ser 4, 5 ou 6. Como?
- Primeiro devemos lembrar que a instrução "*counter++;"* vai ser traduzida por um código de máquina pelo compilador que poderia ser como mostrado a seguir:

```
register1 = counter
register1 = register1 + 1
Counter = register1
```

- E o "*counter--;"* como:

```
register2 = counter
register2 = register2 - 1
Counter = register2
```

Exemplo 2 - Produtor - consumidor

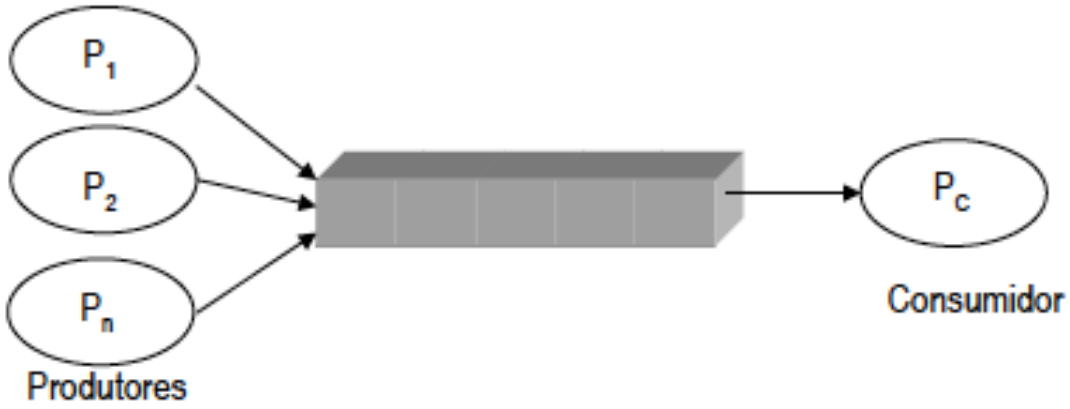
- Considerando que uma tarefa pode perder a posse do processador a qualquer momento, poderíamos ter a seguinte sequência de execução:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

Exemplo 2 - Produtor - consumidor

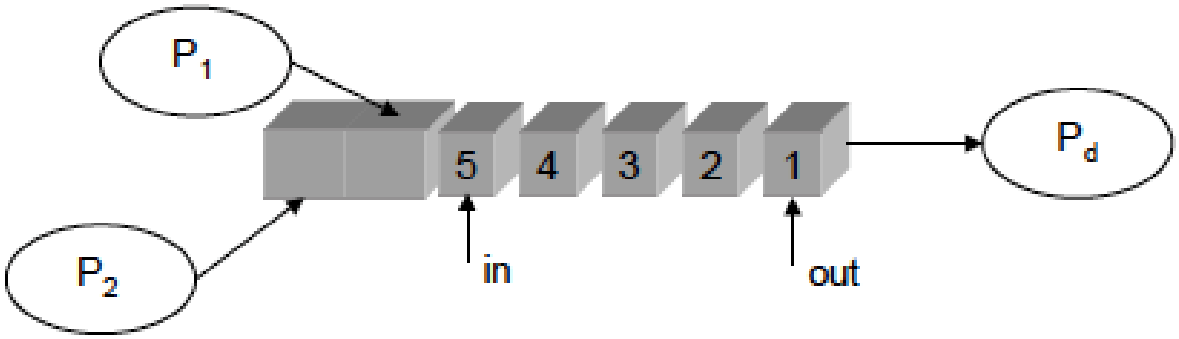
Exemplo 3: Produtor/Consumidor em uma fila de impressão

- **A relação produtor/consumidor é uma situação bastante comum em sistemas operacionais**
- **Ex.: um servidor de impressão**
  - Processos usuários produzem “impressões”
  - Impressões são organizadas em uma fila a partir da qual o um processo (consumidor) os lê e envia para a impresssora



Exemplo 3: Produtor/Consumidor em uma fila de impressão

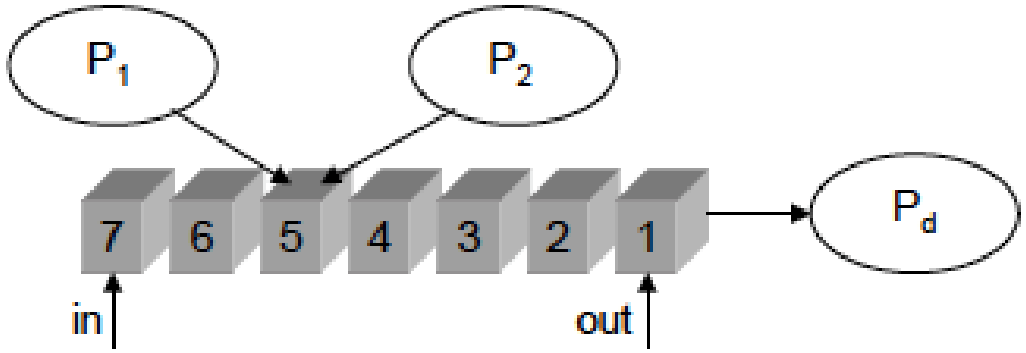
- **Suposições:**
  - Fila de impressão é um buffer circular
  - Existência de um ponteiro (in) que aponta para uma posição onde a impressão é inserida para aguardar o momento de ser efetivamente impressa
  - Existência de um ponteiro (out) que aponta para a impressão que está sendo realizada



Exemplo 3: Produtor/Consumidor em uma fila de impressão

- **Sequência de operações:**
  - P1 vai imprimir, lê o valor de in(5); perde o processador
  - P2 ganha o processador, lê o valor de in(5); insere arquivo; atualiza in(6); perde o processador
  - P1 ganha o processador; insere arquivo(5); atualiza in(7)

- **Estado incorreto:**
  - Impressão P2 é perdida;
  - Na posição 6 não há uma solicitação válida de impressão



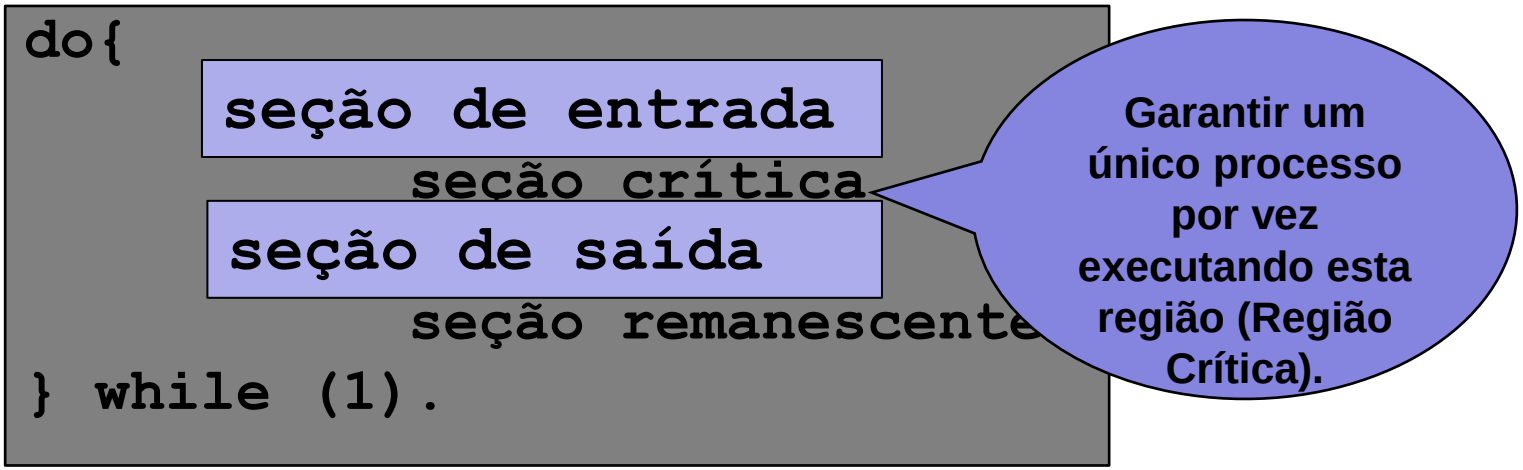
O Problema da seção crítica

- Uma situação como essa, em que vários processos acessam e manipulam os mesmos dados simultaneamente e o resultado da execução depende da ordem específica em que o acesso ocorre, é chamada de **Condição de Disputa** ou **Condição de Corrida (Race Condition).**

- **Seção crítica:**
  - Segmento de código no qual um processo realiza a alteração de um recurso compartilhado.

- **Para resolver o problema da seção crítica é necessário projetar um protocolo que os processos podem usar para cooperar.**

O problema da seção crítica



Requisitos necessários para resolver o problema da seção crítica

- **Regra 1: Exclusão mútua**
  - Dois ou mais processos não podem estar simultaneamente em uma seção crítica
- **Regra 2: Progressão**
  - Nenhum processo fora da seção crítica pode bloquear a execução de um outro processo
- **Regra 3: Espera limitada**
  - Nenhum processo deve esperar infinitamente para entrar em uma seção crítica
- **Regra 4:**
  - Não fazer considerações sobre o número de processadores, nem de suas velocidades relativas

Soluções não recomendadas: Desabilitar as interrupções

- **Não há troca de processos sem a ocorrência de interrupções de tempo (quantum) ou de eventos externos**
- **Desvantagens**
  - Poder demais para um usuário
  - Não funciona em máquina multiprocessadas (SMP) pois apenas a CPU que executa a instrução é afetada (violação da regra 4)

Regra 4:

Não fazer considerações sobre o número de processadores, nem de suas velocidades relativas

Soluções não recomendadas: Variáveis do tipo *lock*

- **Criação de uma variável especial compartilhada que armazena dois estados (boolean, por exemplo):**
  - Zero: livre
  - Um: ocupado
- **Desvantagem:**
  - Apresenta ***race condition***

Regra 1: Exclusão mútua

Dois ou mais processos não podem estar simultaneamente em uma seção crítica

Alternância

- **Desvantagem:**
  - Teste contínuo do valor da variável compartilhada provoca o desperdício do tempo do processador (busy waiting)
  - Viola a regra 2 se a parte não crítica de um processo for muito maior que a do outro

Regra 2: progressão

Nenhum processo fora da seção crítica pode bloquear a execução de um outro processo

}

}

S3: Peterson´s solution (1981)

```
boolean flag[2];
int turn;
```

```
do{
        flag[i]= true;
        turn = j;
        while (flag[j] && turn == j);
                secão crítica
        flag[i] = false;
                seção remanescente
} while (1).
```

```
boolean flag[2];
int turn;
```

```
do{
        flag[0]= true;
        turn = 1;
        while (flag[1] && turn == 1);
                seção crítica
        flag[0] = false;
                seção remanescente
} while (1).
```

P1

```
do{
        flag[1]= true;
        turn = 0;
        while (flag[0] && turn == 0);
                seção crítica
        flag[1] = false;
                seção remanescente
} while (1).
```

P2

Uma ajuda do hardware

- Muitos processadores oferecem uma instrução que permite tanto testar quanto modificar o conteúdo de uma palavra de forma atômica, ou seja, sem ser interrompida.

- **TEST AND SET LOCK (TSL)**

```
Boolean TestAndSet (boolean &target) {
        boolean rv= target;
        target = true;
        return rv;
}
```

Resolução do problema de seção crítica: com o uso da TSL

```
do{
        while (TestAndSet(lock));
                seção crítica
        lock= false;
                seção remanescente
} while (1).
```

Outro exemplo existente em algumas arquiteturas de microprocessadores

- Outros processadores oferecem uma instrução que efetua a troca atômica de conteúdo (swapping) entre dois registradores, ou entre um registrador e uma posição de memória.

- Ex Intel i386 e seus sucessores:

$$XCHG\ op_1, op_2 :\ op_1 \rightleftarrows op_2$$

Solução usando XCHG

```
1  int lock ;                  // variável de trava
2
3  enter (int *lock)
4  {
5     int key = 1 ;            // variável auxiliar (local)
6     while (key)              // espera ocupada
7       XCHG (lock, &key) ;    // alterna valores de lock e key
8  }
9
10 leave (int *lock)
11 {
12    (*lock) = 0 ;            // libera a seção crítica
13 }
```

Spinlocks

- Os mecanismos de exclusão mútua usando instruções atômicas no estilo TSL são amplamente usados no interior do sistema operacional, para controlar o acesso a seções críticas internas do núcleo, como descritores de tarefas, buffers de arquivos ou de conexões de rede, etc. Nesse contexto, eles são muitas vezes denominados **spinlocks**.

Considerações

Apesar das soluções para o problema de acesso à seção crítica usando espera ocupada garantirem a exclusão mútua, elas sofrem de alguns problemas que impedem seu uso em larga escala nas aplicações de usuário:

- **Ineficiência** : as tarefas que aguardam o acesso a uma seção crítica ficam testando continuamente uma condição, consumindo tempo de processador sem necessidade. O procedimento adequado seria suspender essas tarefas até que a seção crítica solicitada seja liberada.

- **Injustiça** : não há garantia de ordem no acesso à seção crítica; dependendo da duração de quantum e da política de escalonamento, uma tarefa pode entrar e sair da seção crítica várias vezes, antes que outras tarefas consigam acessá-la.

Semáforos

- Semáforo foi inventado por Edgser Dijkstra em 1965. É um mecanismo oferecido pelo maioria de Núcleos de SO multitarefas. São usados para:
  - Controlar o acesso a um recurso compartilhado (exclusão mútua);
  - Sinalizar a ocorrência de um evento;
  - Permitir que duas tarefas sincronizem suas atividades.

Um semáforo pode ser visto como uma variável *s*, que representa uma seção crítica e cujo conteúdo não é diretamente acessível ao programador. Internamente, cada semáforo contém um contador inteiro *s:counter* e uma fila de tarefas *s:queue*, inicialmente vazia. Sobre essa variável podem ser aplicadas duas operações atômicas, descritas a seguir:

- *Down*(*s*) : usado para solicitar acesso à seção crítica associada a *s*. Caso a seção esteja livre, a operação retorna imediatamente e a tarefa pode continuar sua execução; caso contrário, a tarefa solicitante é suspensa e adicionada à fila do semáforo; o contador associado ao semáforo é decrementado3. Dijkstra denominou essa operação P(s) (do holandês probeer, que significa tentar).

- *Up*(*s*) : invocado para liberar a seção crítica associada a *s*; o contador associado ao semáforo é incrementado. Caso a fila do semáforo não esteja vazia, a primeira tarefa da fila é acordada, sai da fila do semáforo e volta à fila de tarefas prontas para retomar sua execução. Essa operação foi inicialmente denominada *V*(*s*) (do holandês *verhoog*, que significa *incrementar*). Deve-se observar que esta chamada não é bloqueante: a tarefa não precisa ser suspensa ao executá-la.

Pseudocódigo das operações *Down(s) e Up(s)*

Down(*s*): // a executar de forma atômica
s.counter ← *s.counter* − 1
if *s.counter* < 0 **then**
    põe a tarefa corrente no final de *s.queue*
    suspende a tarefa corrente
end if

Up(*s*): // a executar de forma atômica
s.counter ← *s.counter* + 1
if *s.counter* ≤ 0 **then**
    retira a primeira tarefa *t* de *s.queue*
    devolve *t* à fila de tarefas prontas (ou seja, acorda *t*)
end if

Semáforos

- As operações de acesso aos semáforos são geralmente implementadas pelo núcleo do sistema operacional, na forma de chamadas de sistema. É importante observar que a execução das operações Down(s) e Up(s) deve ser atômica, ou seja, não devem ocorrer acessos concorrentes às variáveis internas do semáforo, para evitar condições de disputa sobre as mesmas.

- A suspensão das tarefas que aguardam o acesso à seção crítica elimina a espera ocupada, o que torna esse mecanismo mais eficiente no uso do processador que os anteriores.

Exemplo hipotético de uso de um semáforo para controlar o acesso a um estacionamento

- O valor inicial do semáforo vagas representa o número de vagas inicialmente livres no estacionamento (500). Quando um carro deseja entrar no estacionamento ele solicita uma vaga; enquanto o semáforo for positivo não haverá bloqueios, pois há vagas livres. Caso não existam mais vagas livres, a chamada carro_entra() ficará bloqueada até que alguma vaga seja liberada, o que ocorre quando outro carro acionar a chamada carro_sai().

```
1  sem_t vagas = 500 ;
2
3  void carro_entra ()
4  {
5     down (vagas) ;        // solicita uma vaga de estacionamento
6     ...                   // demais ações específicas da aplicação
7  }
8
9  void carro_sai ()
10 {
11    up (vagas) ;          // libera uma vaga de estacionamento
12    ...                   // demais ações específicas da aplicação
13 }
```

API POSIX

- A API POSIX define várias chamadas para a criação e manipulação de semáforos. As chamadas mais frequentemente utilizadas estão indicadas a seguir:

```
1  #include <semaphore.h>
2
3  // inicializa um semáforo apontado por "sem", com valor inicial "value"
4  int sem_init(sem_t *sem, int pshared, unsigned int value);
5
6  // Operação Up(s)
7  int sem_post(sem_t *sem);
8
9  // Operação Down(s)
10 int sem_wait(sem_t *sem);
11
12 // Operação TryDown(s), retorna erro se o semáforo estiver ocupado
13 int sem_trywait(sem_t *sem);
```

Mutexes (semáforos binários)

- uma versão simplificada de semáforos, na qual o contador só assume dois valores possíveis: livre (1) ou ocupado (0).

- Exemplo de funções definidas pelo padrão POSIX

```
1  #include <pthread.h>
2
3  // inicializa uma variável do tipo mutex, usando um struct de atributos
4  int pthread_mutex_init (pthread_mutex_t *restrict mutex,
5                          const pthread_mutexattr_t *restrict attr);
6
7  // destrói uma variável do tipo mutex
8  int pthread_mutex_destroy (pthread_mutex_t *mutex);
9
10 // solicita acesso à seção crítica protegida pelo mutex;
11 // se a seção estiver ocupada, bloqueia a tarefa
12 int pthread_mutex_lock (pthread_mutex_t *mutex);
13
14 // solicita acesso à seção crítica protegida pelo mutex;
15 // se a seção estiver ocupada, retorna com status de erro
16 int pthread_mutex_trylock (pthread_mutex_t *mutex);
17
18 // libera o acesso à seção crítica protegida pelo mutex
19 int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Variáveis de condição

- Além dos semáforos, outro mecanismo de sincronização de uso frequente são as variáveis de condição, ou variáveis condicionais. Uma variável de condição não representa um valor, mas uma condição, que pode ser aguardada por uma tarefa. Quando uma tarefa aguarda uma condição, ela é colocada para dormir até que a condição seja verdadeira. Assim, a tarefa não precisa testar continuamente aquela condição, evitando uma espera ocupada.

- Uma tarefa aguardando uma condição representada pela variável de condição c pode ficar suspensa através do operador wait(c), para ser notificada mais tarde, quando a condição se tornar verdadeira. Essa notificação ocorre quando outra tarefa chamar o operador notify(c) (também chamado signal(c)). Por definição, uma variável de condição c está sempre associada a um semáforo binário c.mutex e a uma fila c.queue. O mutex garante a exclusão mútua sobre a condição representada pela variável de condição, enquanto a fila serve para armazenar em ordem as tarefas que aguardam aquela condição.

Exemplo implementação do wait, notify e broadcast

```
1  wait (c):
2     c.queue ← t                  // coloca a tarefa t no fim de c.queue
3     unlock (c.mutex)             // libera o mutex
4     suspend (t)                  // põe a tarefa atual para dormir
5     lock (c.mutex)               // quando acordar, obtém o mutex imediatamente
6
7  notify (c):
8     awake (first (c.queue)) // acorda a primeira tarefa da fila c.queue
9
10 broadcast (c):
11    awake (c.queue)              // acorda todas as tarefas da fila c.queue
```

Exemplo de uso da variável de condição

```
1  Task A ()
2  {
3     ...
4     lock (c.mutex)
5     while (not condition)
6        wait (c) ;
7     ...
8     unlock (c.mutex)
9     ...
10 }
11
12 Task B ()
13 {
14    ...
15    lock (c.mutex)
16    condition = true
17    notify (c)
18    unlock (c.mutex)
19    ...
20 }
```

Variáveis de condição

- As variáveis de condição estão presentes no padrão POSIX, através de operadores *como pthread_cond_wait*, *pthread_cond_signal* e *pthread_cond_broadcast*. O padrão POSIX adota a semântica *Mesa*[1].

1 - a operação *notify(c)* apenas “acorda” as tarefas que esperam pela condição, sem suspender a execução da tarefa corrente. Cabe ao programador garantir que a tarefa corrente vai liberar o mutex e não vai alterar o estado associado à variável de condição.

Problemas Clássicos de Sincronização

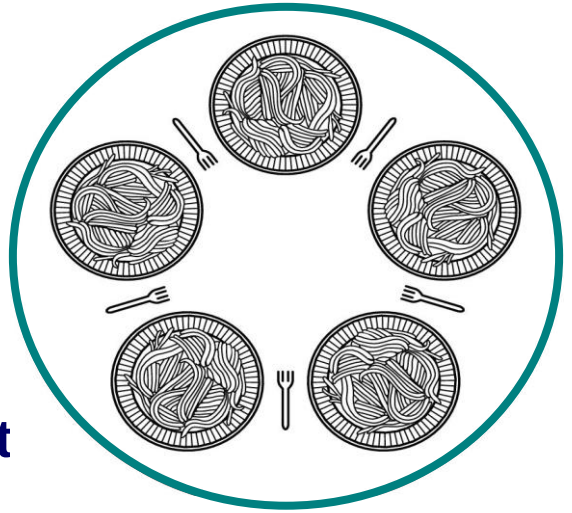
- Produtor/Consumidor
- Jantar dos Filósofos
- Leitores e Escritores
- Barbeiro Dorminhoco

Produtor/Consumidor

- **Um processo produz informações que são gravadas em um buffer limitado**
- **As informações são consumidas por um processo consumidor**
- **O produtor pode produzir um item enquanto o consumidor consome outro**
- **O produtor e o consumidor devem estar sincronizados**
  - O produtor não pode escrever no buffer cheio
  - O consumidor não pode consumir informações de um buffer vazio

Jantar dos Filósofos

- **Cada filósofo possui um prato de espaguete**
- **Para comer o espaguete o filósofo precisa de dois garfos**
- **Existe um garfo entre cada par de prat**
- **Um filósofo come ou medita**
  - Quando medita não interage com seus colegas
  - Quando está com fome ele tenta pegar dois garfos um de cada vez. Ele não pode pegar um garfo que já esteja com outro filósofo
- **Os garfos são os recursos compartilhados**



Leitores e Escritores

- **Existem áreas de dados compartilhadas**
- **Existem processos que apenas lêem dados destas áreas → Leitores**
- **Existem processos que apenas escrevem dados nestas áreas → Escritores**
- **Condições:**
  - Qualquer número de leitores pode ler o arquivo ao mesmo tempo
  - Apenas um escritor pode acessar o arquivo por vez
  - Se um escritor está escrevendo no arquivo, nenhum leitor poderá utilizá-lo

Barbeiro Dorminhoco

- **Neste problema existe:**
  - 1 barbeiro
  - 1 cadeira de barbeiro
  - *N* cadeiras de espera
- **Se não houver clientes o barbeiro senta em sua cadeira e dorme**
- **Quando o cliente chega:**
  - Ele acorda o barbeiro, caso ele esteja dormindo
  - Se o barbeiro estiver trabalhando, o cliente senta para esperar. Caso não existam cadeiras vazias, o cliente vai embora

Região Crítica

Referências:

- Tanenbaum, A. – “Sistemas Operacionais – Projeto de Implementaçãoi” – terceira edição – cap 2 tópico 2; (disponível em formato digital na biblioteca a).
- Maziero, C – “Sistemas Operacionais: Conceitos e Mecanismos” – cap 10; (disponível na internet - Versão compilada em 22 de setembro de 2023).
- Silberschatz, A. – “Operating System Concepts”; 9ª. Edição; editora Wiley – cap 5.