Sincronização: Semáforos e Monitores

Sistemas Operacionais

Prof. Pedro Ramos
pramos.costar@gmail.com

Pontifícia Universidade Católica de Minas Gerais
ICEI - Departamento de Ciência da Computação

ANTERIORMENTE: SINCRONIZAÇÃO

- Exclusão mútua

- Seções críticas

    - Exemplo: Excesso de Café

    - Locks (Travamentos)

    - Primitivas de sincronização são necessárias para garantir que apenas uma thread execute na seção crítica por vez: Lock.Acquire() e Lock.Release();

HOJE - SEMÁFOROS

O que são semáforos?

– Semáforos são basicamente locks generalizados.

– Assim como os locks, semáforos são um tipo especial de variável que suporta duas operações atômicas e oferece soluções elegantes para problemas de sincronização.

– Foram inventados por Dijkstra em 1965.

SEMÁFOROS

Semáforo: **uma variável inteira** que só pode ser atualizada usando duas instruções atômicas especiais.

- **Semáforo Binário (MUTEX): (exatamente igual a um lock)**
  - Garante acesso mutuamente exclusivo a um recurso (**apenas um processo na seção crítica por vez**).
  - Pode variar de 0 a 1
  - É inicializado como livre (valor = 1)

- **Semáforo de Contagem:**
  - Útil quando múltiplas unidades de um recurso estão disponíveis
  - O valor inicial = número de recursos.
  - Um processo pode adquirir acesso desde que pelo menos uma unidade do recurso esteja disponível.

SEMÁFOROS - CONCEITOS BÁSICOS

Assim como locks, o semáforo suporta duas operações atômicas, Semaphore.Wait() e Semaphore.Signal().

```
S.Wait()    // aguarda até que o semáforo S esteja disponível
<seção crítica>
S.Signal()  // sinaliza para outros processos que o semáforo S está livre
```

- **Cada semáforo TEM UMA FILA DE PROCESSOS AGUARDANDO** para acessar a seção crítica (*por exemplo, para comprar café*).
- Se um processo executa S.Wait()
  1. semáforo S está livre (não-zero): continua executando.
  2. semáforo S não está livre: o processo entra na fila de espera do semáforo S.
- Um ***S.Signal() desbloqueia um processo da fila de espera do*** semáforo S.

SEMÁFOROS BINÁRIOS

```
THREAD A                THREAD B

Lock.Acquire();         Lock.Acquire();
if (semCafe) {          if (semCafe) {
    comprar Cafe;           comprar Cafe;          ⇒ COM LOCKS
}                       }
Lock.Release();         Lock.Release();
```

```
THREAD A                THREAD B

S.Wait();               S.Wait();
if (semCafe) {          if (semCafe) {
    comprar Cafe;           comprar Cafe;          ⇒ COM SEMÁFOROS
}                       }
S.Signal();             S.Signal();
```

SIGNAL() E WAIT()

```
class Semaphore {
    public:
        void Wait(Process P);
        void Signal();
    private:
        int valor;
        Queue Q; // fila de processos
}

Semaphore(int val) {
    valor = val;
    Q = vazio;
}
```

```
Wait(Process P) {
    valor = valor - 1;
    if (valor < 0) {
        adicionar P a Q;
        P->bloquear();
    }
}

Signal() {
    valor = valor + 1;
    if (valor <= 0) {
        remover P de Q;
        acordar(P);
    }
}
```

Signal e Wait devem ser atômicos!

SIGNAL() E WAIT()

```
class Semaphore {
    public:
        void Wait(Process P);
        void Signal();
    private:
        int valor;
        Queue Q; // fila de processos
}

Semaphore(int val) {
    valor = val;
    Q = vazio;
}
```

```
Wait(Process P) {
    valor = valor - 1;
    if (valor < 0) {
        adicionar P a Q;
        P->bloquear();
    }
}

Signal() {
    valor = valor + 1;
    if (valor <= 0) {
        remover P de Q;
        acordar(P);
    }
}
```

não há espaço para acessar

Pra que serve o valor?
O que significa um valor negativo?
E um valor positivo?
Aplicações? Porquê usar um semáforo ao invés de locks?

SIGNAL() e WAIT()

```
P1: S.Wait();
    S.Wait();
    S.Signal();
    S.Signal();
```

```
P2: S.Wait();
    S.Signal();
```

```
P1: S->Wait();
P2: S->Wait();
P1: S->Wait();
P2: S->Signal();
P2: S->Signal();
P2: S->Signal();
```

		ESTADO: EXECUTANDO / BLOCK	
VALOR	FILA	P1	P2
2	VAZIA	EXECUTANDO	EXECUTANDO

UTILIZANDO SEMÁFOROS - PRA QUÊ?

Exclusão Mútua: usada para proteger seções críticas
- O semáforo tem valor inicial == 1.
- **S->Wait() é chamado antes da seção crítica**, e **S->Signal() é chamado após a seção crítica.**

Restrições de Escalonamento: usadas para expressar restrições gerais de escalonamento, em que threads devem aguardar por algum estado.
- O valor inicial do semáforo == 0
- Exemplo: implementar Thread.Join (ou a chamada de sistema Unix waitpid(PID)) com semáforos:

```
Semaphore S;
S.value = 0;  // inicialização do semáforo
Thread.Join      Thread.Finish
  S.Wait();       S.Signal();
```

UTILIZANDO SEMÁFOROS - PRA QUÊ?

Exclusão Mútua: usada para proteger seções críticas
- O semáforo tem valor inicial **é**
- **S->Wait() é chamado antes da chamado após a seção crítica.**

mesma utilidade do lock

Restrições de Escalonamento: usadas para expressar restrições gerais de escalonamento, em que threads devem aguardar por algum estado.
- O valor inicial do semáforo == 0
- Exemplo: implementar Thread.Join (ou a chamada de sistema Unix waitpid(PID)) com semáforos:

```
Semaphore S;
S.value = 0;  // inicialização do semáforo
Thread.Join      Thread.Finish
  S.Wait();       S.Signal();
```

real utilidade do semáforo

UTILIZANDO SEMÁFOROS - PRA QUÊ?

```
Semaphore S;
S.value = 0;
Thread.Join          Thread.Finish
  S.Wait();           S.Signal();
```

E SE TIVESSE UMA TERCEIRA THREAD?

UTILIZANDO SEMÁFOROS - PRA QUÊ?

```
Semaphore S;
S.value = 0;
Thread.Join      Thread.Finish
  S.Wait();       S.Signal();
```

E SE TIVESSE UMA TERCEIRA THREAD?

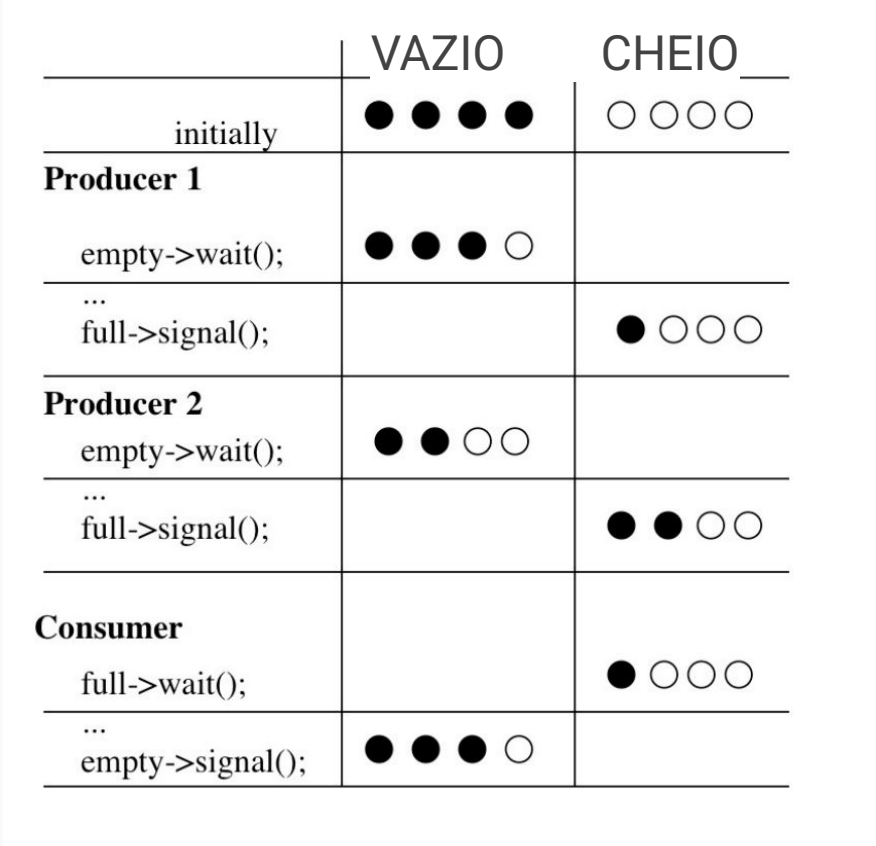
Comprar Café
Comprar Xícara
Preparar o café

SEMÁFOROS - MÚLTIPLOS CONSUMIDORES E PRODUTORES

```
class BufferComLimite {
 public:
    void Produtor();
    void Consumidor();

 private:
    Items buffer;
    // controla o acesso aos buffers
    Semaphore mutex;
    // conta os slots livres
    Semaphore vazio;
    // conta os slots usados
    Semaphore cheio;
};

BufferComLimite::BufferComLimite(int N) {
    mutex.value = 1;
    vazio.value = N;
    cheio.value = 0;
    buffer = new Items[N];
}
```

```
void BufferComLimite::Produtor() {
    <produzir item>;
    vazio.Wait();  // um slot a menos, ou aguarda
    mutex.Wait();  // obtém acesso aos buffers
    <adicionar item ao buffer>;
    mutex.Signal();  // libera os buffers
    cheio.Signal();   // um slot a mais usado
}

void BufferComLimite::Consumidor() {
    cheio.Wait();   // aguarda até que haja um item
    mutex.Wait();   // obtém acesso aos buffers
    <remover item do buffer>;
    mutex.Signal();  // libera os buffers
    vazio.Signal();  // um slot a mais livre
    <usar item>;
}
```

SEMÁFOROS - MÚLTIPLOS CONSUMIDORES E PRODUTORES



DETALHES DE IMPLEMENTAÇÃO DE SEMÁFOROS

Em Resumo…

- Locks podem ser implementados ativando/desativando interrupções ou usando espera ocupada.

- Semáforos são uma generalização dos locks.

- Semáforos podem ser usados para **três finalidades:**

1. **Garantir execução mutuamente exclusiva de uma seção crítica** (como os locks fazem).
2. **Controlar o acesso a um *pool* compartilhado de recursos** (usando um **semáforo de contagem**).
3. Fazer com que uma **thread espere por uma ação específica sinalizada por outra thread.**

A SEGUIR: MONITORES E VARIÁVEIS DE CONDIÇÃO

- O que há de errado com semáforos?

- Monitores: **Arcabouço + sofisticado de sincronização**
  - O que são?
  - Como implementamos monitores?
  - Dois tipos de monitores: Mesa e Hoare

- Comparação entre semáforos e monitores

O QUE HÁ DE ERRADO COM SEMÁFOROS?

Desvantagens:

– São essencialmente variáveis globais compartilhadas.

– Não há uma conexão linguística entre o semáforo e os dados aos quais o semáforo controla o acesso.

– O acesso aos semáforos **pode vir de qualquer lugar em um programa.**

– Eles servem para múltiplos propósitos: exclusão mútua e restrições de escalonamento.

– Não há controle ou garantia de uso adequado.

– Fácil de cometer erros e não conseguir rastreá-los

• Solução: usar um primitivo de nível superior chamado **monitores**.

O QUE É UM MONITOR?

É uma classe que provê sincronização para o usuário.

Um monitor é semelhante a uma **classe que une os dados, operações e, em particular, as operações de sincronização.**

- Ao contrário das classes:
  - Os monitores garantem exclusão mútua, ou seja, apenas uma thread pode executar um dado método do monitor por vez.
  - Os **monitores exigem que todos os dados sejam privados.**

MONITORES: DEFINIÇÃO FORMAL

Um Monitor define **um lock** e **zero ou mais variáveis de condição** para gerenciar o acesso concorrente a dados compartilhados.

– O monitor usa o lock para **garantir que apenas 1 única thread esteja ativa no monitor a qualquer momento.**

– O lock garante exclusão mútua para dados compartilhados.

– **As variáveis de condição permitem que as threads adormeçam dentro de seções críticas, liberando seu lock ao mesmo tempo em que coloca a thread para dormir**.

MONITORES: DEFINIÇÃO FORMAL

Operações do Monitor:

- **Encapsula** os dados compartilhados que você deseja proteger.
- **Adquire** o mutex no início.
- **Opera** sobre os dados compartilhados.
- **Libera** temporariamente o mutex se não conseguir concluir.
- **Re-adquire** o mutex quando pode continuar.
- **Libera** o mutex no final.

MONITORES EM JAVA

É simples transformar uma classe Java em um monitor:
- Torne todos os dados privados
- Torne todos os métodos sincronizados *(ou pelo menos os não privados)*

```
class Queue {
    private ...; // dados da fila

    public synchronized void Add(Object item) {
        coloca o item na fila
    }

    public synchronized Object Remove() {
        if (fila não está vazia) {
            remove o item
            return item;
        }
    }
}
```

VARIÁVEIS DE CONDIÇÃO

Como podemos mudar o método remove() para esperar até que haja algo na fila?

– Queremos adormecer dentro da seção crítica.

– Mas se **segurarmos o lock e adormecermos**, outras threads não poderão *acessar a fila compartilhada, adicionar um item a ela e acordar a thread adormecida.*

=> **A thread pode adormecer para sempre.**

• Solução: usar variáveis de condição.

– As **variáveis de condição permitem que uma thread durma dentro de uma seção crítica.**

– *Qualquer lock mantido pela thread é liberado atomicamente quando a thread é colocada para dormir.*

VARIÁVEIS DE CONDIÇÃO

Variável de condição: é uma **fila de threads esperando por algo dentro de uma seção crítica.**

suportam três operações:
1. *Wait(Lock lock)*: **atômica** (**libera o lock, vai dormir**); quando o processo acorda, ele re-adquire o lock.
2. *Signal()*: acorda a thread que está esperando, se houver uma. Caso contrário, não faz nada.
3. *Broadcast()*: acorda todas as threads que estão esperando.

• Regra: **a thread deve manter o lock ao realizar operações com variáveis de condição.**

VARIÁVEIS DE CONDIÇÃO NO JAVA

_ Use wait() para abrir mão do lock.
_ Use **notify**() para sinalizar que a **condição** pela qual uma thread está esperando foi **satisfeita**.
_ Use notifyAll() para acordar todas as threads que estão esperando.
_ *1 variável de condição por objeto.*

```
class Queue {
    private ...; // dados da fila

    public synchronized void Add(Object item) {
        coloca o item na fila
        notify(); // acorda a thread dentro da seção crítica
    }

    public synchronized Object Remove() {
        while (fila está vazia) {
            wait(); // abre mão do lock e vai dormir
        }
        remove e retorna o item
    }
}
```

COMO MONITORES SÃO IMPLEMENTADOS?

O que acontece quando signal() é chamado?

– Sem threads esperando => o **sinalizador continua e o sinal é efetivamente perdido** (diferente do que acontece com semáforos).

– Se há uma thread esperando, uma das threads começa a executar, enquanto as outras devem esperar.

- **Estilo Mesa: (Nachos, Java e a maioria dos sistemas operacionais reais)**
  – A thread que sinaliza mantém o lock (e, portanto, o processador).
  – A thread esperando aguarda o lock.

- **Estilo Hoare: (a maioria dos livros didáticos)**
  – A thread que sinaliza libera o lock e a thread esperando obtém o lock.
  – Quando a thread que estava esperando e agora está executando sai ou espera novamente, ela libera o lock de volta para a thread que sinalizou.

VARIÁVEIS DE CONDIÇÃO NO JAVA

Hoare-style

```
class Queue {
    private ...; // dados da fila

    public synchronized void Add(Object item) {
        coloca o item na fila
        notify();
    }

    public synchronized Object Remove() {
        if (fila está vazia) {
            wait(); // abre mão do lock e vai dormir
        }
        remove e retorna o item
    }
}
```

RESUMO

- Monitor encapsula operações com um MUTEX (Lock)
- Variáveis de condição “liberam” o mutex temporariamente
- Java tem monitores built-in na linguagem, C++ não, mas pode ser implementado usando Lock.Acquire() e Lock.Release()
- Monitores podem ser implementados com semáforos também.

EXERCÍCIO EM SALA - ENTREGAR

Considere dois Processos, P_A e P_B. Suponha que os processos compartilham as variáveis y e z. Considere que o sistema operacional utiliza um escalonador *round robin* com compartilhamento de tempo (*time sharing*).

Processo A:	Processo B:
// inicializações inteiro x; $x = y + z$; //... *Processo A continua*	// inicializações $z = 2$; $y = 1$; //... *Processo B continua*

a) Quais os possíveis valores finais para x?

b) Mostre como seria possível resolver o problema utilizando semáforos.

PERGUNTAS?

REFERÊNCIAS

- **TANENBAUM, Andrew.** Sistemas operacionais modernos.
- **SILBERSCHATZ, Abraham et al.** Fundamentos de sistemas operacionais: princípios básicos.
- **MACHADO, Francis; MAIA, Luiz Paulo.** Arquitetura de Sistemas Operacionais.
- **CARISSIMI, Alexandre et al.** Sistemas operacionais.