Sincronização

Sistemas Operacionais

Prof. Pedro Ramos
pramos.costar@gmail.com

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

NA AULA DE HOJE - SINCRONIZAÇÃO

- Recapitulação de escalonamento
  - MLQ e Loteria

- Sincronização
  - Exclusão mútua
  - Seções críticas

- Exemplo: Excesso de Café

- Locks (cadeados)

- Primitivas de sincronização que **permitem apenas 1 thread executar uma seção crítica por vez.**

SINCRONIZAÇÃO

Hora	Você	Seu colega de quarto
15:00	Chega em casa	
15:05	Olha na despensa, sem café	
15:10	Sai para comprar café	Chega em casa
15:15		Olha na despensa, sem café
15:20	Chega no mercado	Sai para comprar café
15:25	Compra café	
15:35	Chega em casa, coloca o café na despensa	
15:45		Compra café
15:50		Chega em casa, guarda o café
15:50		Oh não!

SINCRONIZAÇÃO

Hora	Você	Seu colega de quarto
15:00	Chega em casa	
15:05	Olha na despensa, sem café	
15:10	Sai para comprar café	Chega em casa
15:15		Olha na despensa, sem café
15:20	Chega no mercado	Sai para comprar café
15:25	Compra café	
15:35	Chega em casa, coloca o café na despensa	
15:45		Compra café
15:50		
15:50		

QUAIS MECANISMOS PRECISAMOS PARA ESTABELECER UMA COMUNICAÇÃO ENTRE PROCESSOS INDEPENDENTES E TER UMA VISÃO CONSISTENTE DO MUNDO (ESTADO)?

SINCRONIZAÇÃO - TERMINOLOGIA

- SINCRONIZAÇÃO - Uso de operações atômicas para garantir cooperação entre threads

SINCRONIZAÇÃO - TERMINOLOGIA

- SINCRONIZAÇÃO - Uso de operações atômicas para garantir cooperação entre threads
- EXCLUSÃO MÚTUA - Garantir que apenas uma thread execute uma atividade em um determinado tempo e excluir outras threads de tentarem executar a mesma atividade ao mesmo tempo

SINCRONIZAÇÃO - TERMINOLOGIA

- SINCRONIZAÇÃO - Uso de operações atômicas para garantir cooperação entre threads
- EXCLUSÃO MÚTUA - Garantir que apenas uma thread execute uma atividade em um determinado tempo e excluir outras threads de tentarem executar a mesma atividade ao mesmo tempo
- SEÇÃO CRÍTICA - Pedaço de código que somente uma thread pode executar por vez

SINCRONIZAÇÃO - TERMINOLOGIA

- SINCRONIZAÇÃO - Uso de operações atômicas para garantir cooperação entre threads
- EXCLUSÃO MÚTUA - Garantir que apenas uma thread execute uma atividade em um determinado tempo e excluir outras threads de tentarem executar a mesma atividade ao mesmo tempo
- SEÇÃO CRÍTICA - Pedaço de código que somente uma thread pode executar por vez
- LOCK (Cadeado) - Mecanismo para prevenir outro processo de realizar algo
  - **Tranca** antes de entrar em uma seção crítica ou acessar dados compartilhados
  - **Destranca** após sair da seção crítica ou finalizar acesso à memória compartilhada
  - **Espere** se estiver trancado

SINCRONIZAÇÃO - TERMINOLOGIA

- SINCRONIZAÇÃO - Uso de operações atômicas para garantir cooperação entre threads
- EXCLUSÃO MÚTUA - Garantir que apenas uma thread execute uma atividade em um determinado tempo e excluir outras threads de tentarem executar a mesma atividade ao mesmo tempo
- SEÇÃO CRÍTICA - Pedaço de código que somente uma thread pode executar por vez
- LOCK (Cadeado) - Mecanismo para prevenir outro processo de realizar algo
  - Tranca o cadeado antes de entrar em uma seção crítica ou acessar dados compartilhados
  - Destranca após sair da seção crítica ou finalizar acesso à memória compartilhada
  - Espere se estiver trancado

TODA SINCRONIZAÇÃO ENVOLVE ESPERA

EXCESSO DE CAFÉ - SOLUÇÃO 1

- Quais são as propriedades de corretude deste problema?
  - **Apenas uma pessoa compra café** por vez;
  - **Alguém compra café** se você precisar.

- Plano: restringir a solução a **loads e stores atômicos**
  - Deixar um recado (uma versão de **lock**/trancar)
  - Remover o recado (uma versão de **unlock**/destrancar)
  - Não compre café se existe um recado (**esperar**)

EXCESSO DE CAFÉ - SOLUÇÃO 1

```
THREAD A
if (semCafe && semRecado) {
    deixar Recado;
    comprar Café;
    remover Recado;
}


THREAD B
if (semCafe && semRecado) {
    deixar Recado;
    comprar Café;
    remover Recado;
}
```

ESSA SOLUÇÃO FUNCIONA?

EXCESSO DE CAFÉ - SOLUÇÃO 1

```
THREAD A
if (semCafe && semRecado) {
    deixar Recado;
    comprar Café;
    remover Recado;
}
```

```
THREAD B
if (semCafe && semRecado) {
    deixar Recado;
    comprar Café;
    remover Recado;
}
```

ESSA SOLUÇÃO FUNCIONA? -> *PODE FUNCIONAR ÀS VEZES*

Como vai acontecer o escalonamento? Lembre-se: o SO vai escalonar as threads. Se o tempo de A acabar logo após entrar no IF, e antes de deixar o Recado, ambas as threads comprarão café.

EXCESSO DE CAFÉ - SOLUÇÃO 1

```
THREAD A
if (semCafe && semRecado) {
    deixar Recado;
    comprar Café;
    remover Recado;
}
```

ESSA SOLUÇÃO SEMPRE FUNCIONA? INDEPENDENTE DO ESCALONADOR?

```
THREAD B
if (semCafe && semRecado) {
    deixar Recado;
    comprar Café;
    remover Recado;
}
```

Como vai acontecer o escalonamento? Lembre-se: o SO vai escalonar as threads. Se o tempo de A acabar logo após entrar no IF, e antes de deixar o Recado, ambas as threads comprarão café.

EXCESSO DE CAFÉ - SOLUÇÃO 2

PLANO: Usar recados rotulados (deixar o recado antes de comprar o café)

THREAD A

```
deixar RecadoA
if (semRecadoB) {
    if (semCafe) {
        comprar café;
    }
}
remover Recados
```

THREAD B

```
deixar RecadoB
if (semRecadoA) {
    if (semCafe) {
        comprar café;
    }
}
remover Recados
```

ESSA SOLUÇÃO FUNCIONA?

EXCESSO DE CAFÉ - SOLUÇÃO 2

PLANO: Usar recados rotulados (deixar o recado antes de comprar o café)

THREAD A

```
deixar RecadoA
if (semRecadoB) {
    if (semCafe) {
        comprar café;
    }
}
remover Recados
```

THREAD B

```
deixar RecadoB
if (semRecadoA) {
    if (semCafe) {
        comprar café;
    }
}
remover Recados
```

ESSA SOLUÇÃO FUNCIONA?

THREAD A pode deixar um RecadoA, e o contexto mudar para a THREAD B que deixa o recado B. Nenhuma das duas threads compra café.

EXCESSO DE CAFÉ - SOLUÇÃO 3

THREAD A

```
    deixar RecadoA;
X:  while (RecadoB) {
        faça nada;
    }
    if (semCafe) {
        comprar café;
    }
    remover RecadoA;
```

THREAD B

```
    deixar RecadoB
Y:  if (semRecadoA) {
        if (semCafe) {
            comprar café;
        }
    }
    remover RecadoB
```

ESSA SOLUÇÃO FUNCIONA?

EXCESSO DE CAFÉ - SOLUÇÃO 3

THREAD A

```
    deixar RecadoA;
X:  while (RecadoB) {
        faça nada;
    }
    if (semCafe) {
        comprar café;
    }
    remover RecadoA;
```

THREAD B

```
    deixar RecadoB
Y:  if (semRecadoA) {
        if (semCafe) {
            comprar café;
        }
    }
    remover RecadoB
```

Não é óbvio constatar que esse exemplo funciona.
Thread A está mais “paranóica” em executar sem excesso de café.
Se Thread A enxergar o recado de B, vai “esperar” na porta da despensa.
Thread B eventualmente *sempre* remove Recado B.
Estamos buscando uma forma de trancar (lock) as threads.

EXCESSO DE CAFÉ - SOLUÇÃO 3

THREAD A

```
    deixar RecadoA;
X:  while (RecadoB) {
        faça nada;
    }
    if (semCafe) {
        comprar café;
    }
    remover RecadoA;
```

THREAD B

```
    deixar RecadoB
Y:  if (semRecadoA) {
        if (semCafe) {
            comprar café;
        }
    }
    remover RecadoB
```

FUNCIONA, NÃO É IDEAL POR SER **ASSIMÉTRICO**.
THREADS A E B RODAM CÓDIGOS ***DIFERENTES***.

EXCESSO DE CAFÉ - CORRETUDE DA SOLUÇÃO 3

- **NO PONTO Y, ou há Recado A ou não.**
  1. Se não há Recado A, é seguro para a Thread B checar e comprar café se preciso.
  2. Se há Recado A, então Thread A está checando e comprando café se necessário ou está esperando por B. B termina e remove Recado B.

- **NO PONTO X, ou há Recado B ou não.**
  1. Se não há Recado B, é seguro para A comprar café já que B ainda não começou ou terminou.
  2. Se há Recado B, A está esperando até não haver mais Recado B, e irá encontrar Café ou comprar mais se preciso.

EXCESSO DE CAFÉ - CORRETUDE DA SOLUÇÃO 3

- **NO PONTO Y, ou há Recado A ou não.**
  1. Se não há Recado A, é seguro para a Thread B checar e comprar café se preciso.
  2. Se há Recado A, então Thread A está checando e comprando café se necessário ou está esperando por B. B termina e remove Recado B.

- **NO PONTO X, ou há Recado B ou não.**
  1. Se não há Recado B, é seguro para A comprar café já que B ainda não começou ou terminou.
  2. Se há Recado B, A está esperando até não haver mais Recado B, e irá encontrar Café ou comprar mais se preciso.

THREAD B COMPRA CAFÉ (QUE A THREAD A ENCONTRA) OU NÃO, DE QUALQUER FORMA REMOVE RECADO B. THREAD A ITERA (LOOP), ESPERA POR THREAD B COMPRAR CAFÉ OU NÃO, E SE B NÃO COMPROU (OU O CAFÉ JÁ ACABOU NOVAMENTE), A COMPRA CAFÉ.

EXCESSO DE CAFÉ - SOLUÇÃO 3 É BOA?

- **Complexa** - é difícil convencer de que a solução funciona.
- **Assimétrica** - Threads A e B executam código diferente. Adicionar novas threads envolveria adicionar novos códigos e modificar os já existentes de A e B.
- A está em **espera ocupada** - A consome CPU durante sua fatia de tempo, mas não realiza trabalho útil.

Essa solução confia que **LOADS** e **STORES** (deixar recados em memória compartilhada) **SÃO ATÔMICOS**

SINCRONIZAÇÃO - SUPORTE NA LINGUAGEM

- **Solução** - tenha na linguagem de programação **suporte para sincronização através de rotinas atômicas.**

LOCKS (Cadeados/Trancas): UM PROCESSO “TRANCA” EM UM DETERMINADO TEMPO, FAZ SUA SEÇÃO CRÍTICA E “DESTRANCA”

SEMÁFOROS: VERSÃO GENÉRICA DE LOCKS

MONITORES: CONECTA DADOS COMPARTILHADOS COM AS PRIMITIVAS DE SINCRONIZAÇÃO

→ Todos requerem **suporte no Hardware** e **espera**

LOCKS (CADEADOS)

- Locks: provê exclusão mútua em dados compartilhados através de rotinas “atômicas”:
  - ***Lock.Acquire*** - espera até o Lock estiver livre, e adquire o lock pra si.
  - ***Lock.Release*** - destranca, e acorda qualquer thread que está esperando o *Acquire*.

REGRAS PARA USO DO LOCK:

- Sempre adquirir o lock antes de acessar dados compartilhados.
- Sempre liberar o lock após acessar dados compartilhados.
- Lock é inicialmente *livre*.

EXCESSO DE CAFÉ - SOLUÇÃO COM LOCKS

```
THREAD A

Lock.Acquire();
if (semCafe) {
    comprar Cafe;
}
Lock.Release();

THREAD B

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

- Limpa e simétrica
- ***Como fazer Lock.Acquire() e Lock.Release() atômicos?***

SINCRONIZAÇÃO - SUPORTE NO HARDWARE

- Para implementar primitivas em alto nível, precisamos de primitivas de baixo nível implementadas no Hardware.
- O que temos e o que precisamos:

OPERAÇÕES ATÔMICAS BAIXO NÍVEL (Hardware)	load/store, interrupt, disable, test, set
OPERAÇÕES ATÔMICAS ALTO NÍVEL (Software)	locks, semáforos, monitores, send, receive

SINCRONIZAÇÃO - IMPLEMENTAÇÃO DE LOCKS

No livro texto, há detalhes da implementação de Locks

- **Utilizando mecanismos de interrupção (interrupt/disable)**
  - \+ Complexo, porém não há espera ocupada pois threads dormem quando são interrompidas

- **Utilizando instruções atômicas de acesso à memória (leitura/escrita/alteração)**
  - Test&Set: lê e escreve ‘1’(maioria das arquiteturas)

  - \+ Simples, porém deve lidar com espera ocupada (minimizar a espera ocupada. eliminá-la é impossível)

RESUMO

- Comunicação entre THREADS é tipicamente feita por **variáveis compartilhadas.**

- **Seções críticas** identificam pedaços de código que não podem executar em paralelo por múltiplas threads, tipicamente **código que acessa ou modifica as variáveis compartilhadas.**

- **Primitivas de sincronização** são requeridas para garantir que somente uma thread execute uma seção crítica por vez.
  - Sincronização pura com load/store é complicado e suscetível a erros
  - Solução: usar primitivas de alto nível como **Locks, Semáforos e Monitores.**

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.