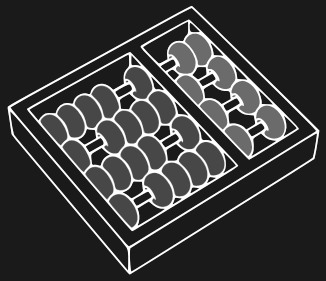


UNICAMP

Sincronização (Parte 2)

Prof. Carlos A. Astudillo

castudillo@ic.unicamp.br

Sistemas Operacionais (MC504A)

2º Semestre 2023

Adaptado de Dave Eckhard

Recapitulando

- Dois formas principais de construir programas com threads
- Três requisitos para seções críticas
- Algoritmos que não se utilizam para seções críticas

Seção crítica

- Protege uma sequência de instruções que devem executar-se atomicamente
  - Devemos fazer algo para proteger a execução das sequências
  - Lembremos, a CPU muda execução entre threads (e processos)
  - Um thread executando-se em outra CPU (multi processador)
- Suposições
  - A sequência de instruções atômicas deve ser pequena
  - Não há outros threads que compitam (dentro dela)

Objetivos da seção crítica

- Caso comum (não há competidores) deve ser rápido
- O caso atípico (não comum)
  - Pode ser lento
  - Mas! não deve desperdiçar os recursos

Sequências que interferem

Cliente	Entrega
`cash = store->cash;`	`cash = store->cash;`
`cash += 50;`	`cash -= 20;`
`wallet -= 50;`	`wallet += 20;`
`store->cash = cash;`	`store->cash = cash;`

- Quais sequências interferem umas nas outras?
- Cliente interfere Entrega
- E viceversa, Entrega interfere Cliente

Objetivos da aula

- Soluções (agora sim) para o problema de seções críticas
- Mutex
- Implementação
- Ambientes de execução

Mutex (Lock ou Latch)

- **Mut**ual **ex**clusion
- Limita (e especifica) o código que interfere através de um objeto (variável)
  - Os dados estão protegidos pelo mutex
- Os métodos dos objetos encapsulam os protocolos de entrada e saída

```
mutex_lock(&store->lock);
cash = store->cash;
cash += 50;
personal_cash -= 50;
store->cash = cash;
mutex_unlock(&store->lock);
```

- O que há dentro do objeto?

Exclusão mutua

Troca atômica

- Nos Intel x86 usamos a instrução `xchg`
- Por exemplo `xchg (%esi), %edi`

```
int32 xchg(int32 *lock, int32 val){
  register int old;
  old = *lock; // bus is locked
  *lock = val; // bus is locked
  return (old);
}
```

Dentro do mutex

- Inicialização

```
int lock_available = 1;
```

- Intentar segurar

```
i_won = xchg(&lock_available, 0);
```

- Spin-wait

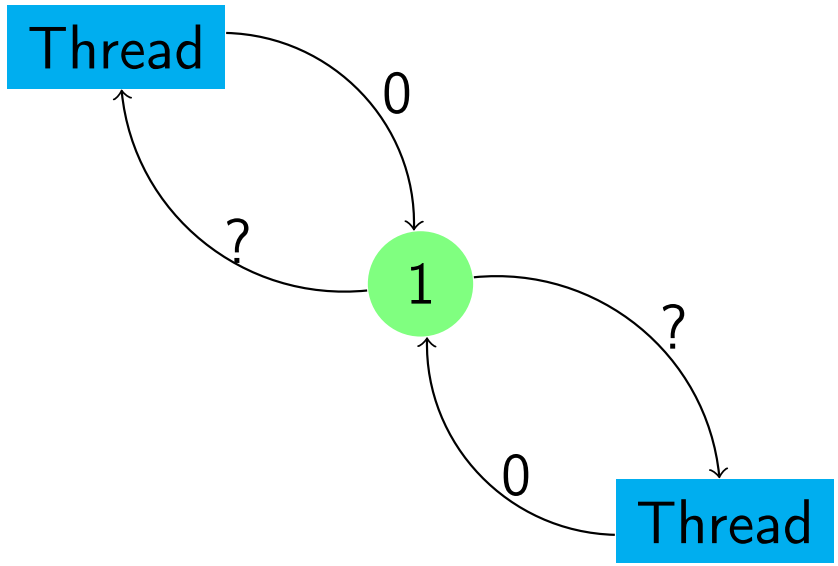
```
while(!xchg(&lock_available, 0))
  continue;
```

- Liberar o seguro (seção crítica)

```
xchg(&lock_available, 1);
```

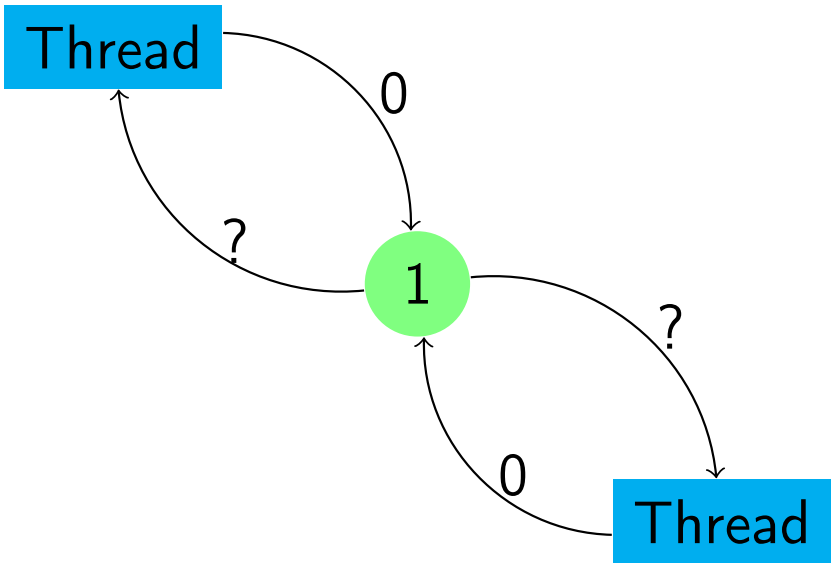
Dois threads

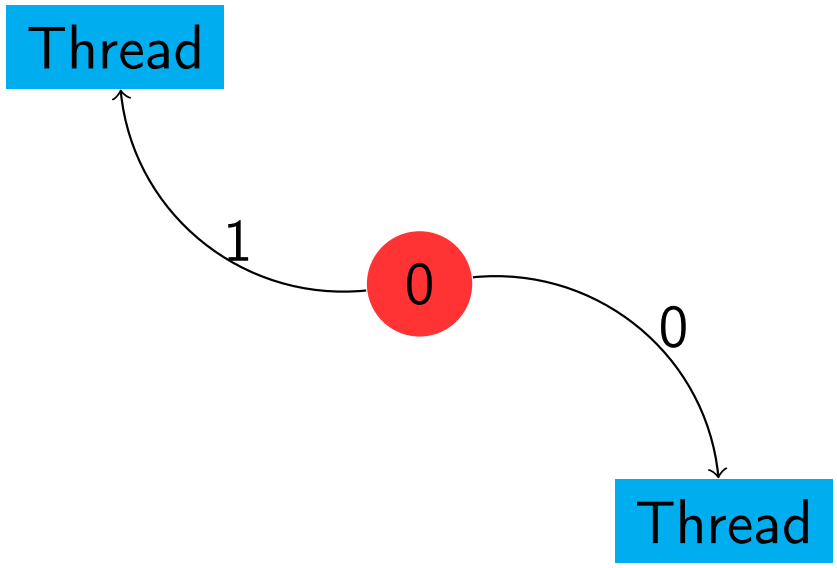
- Imaginemos dois threads tomando um valor (`lock_available`), e deixando outro no mesmo lugar
- Simultaneamente



Dois threads

- Imaginemos dois threads tomando um valor (`lock_available`), e deixando outro no mesmo lugar
- Simultaneamente





Mantemos as características da seção crítica?

- Exclusão mutua
  - Existe só um 1 (os 1 se conservam)
  - Só um thread pode ver o `lock_available == 1`

Mantemos as características da seção crítica?

- Exclusão mutua
  - Existe só um 1 (os 1 se conservam)
  - Só um thread pode ver o `lock_available == 1`
- Progresso
  - Quando `lock_available == 1` algum thread o vai pegar

Mantemos as características da seção crítica?

- Exclusão mutua
  - Existe só um 1 (os 1 se conservam)
  - Só um thread pode ver o `lock_available == 1`
- Progresso
  - Quando `lock_available == 1` algum thread o vai pegar
- Espera acotada
  - Não
  - Um thread pode perder (a toma do valor) arbitrariamente muitas vezes

Garantindo a espera acotada

- Intuição
  - Muitos threads podem `xchg` simultaneamente
  - Precisamos um sistema que permita "tomar turnos" (ou que tenha um comportamento similar)
- Possíveis soluções
  - Garantir que cada adquisição da exclusão em `xchg` tenha uma saída *justa*
    - O como, não é necessariamente obvio
  - E devemos agregar justiça através do procedimento de liberação do lock
    - Alguém é responsável (quem tem a seção crítica)
    - Utilizemos isso a nosso favor

Lock (segurar)

```
waiting[i] = true; // declaremos interesse na seção crítica
got_it = false;

//"spin" enquanto o obtemos
while (waiting[i] && !got_it)
  got_it = xchg(&lock_available, false);

waiting[i] = false;
return; // obtimos a seção crítica, sucesso!
```

Unlock (liberar)

```
j = (j + 1) % n; // lembrem que temos n threads
while ( (j != i) && !waiting[j] ) // ?
  j = (j + 1) % n;
if (j == i) // ?
  xchg(&lock_available, true);
else
  waiting[j] = false;
return;
```

Possíveis variações

- Trocas vs. `TestAndSet`
- O nome das variáveis que protegem a seção crítica podem mudar (e.g., `available` em lugar de `locked`)
- Liberação atômica vs. escrita normal (onde fizemos `xchg` na slide anterior)
  - Alguns fazem uma escrita cega `lock_available = true;`
  - Dependendo da arquitetura, isto pode ser ilegal
  - Quem está liberando deve precisar utilizar acesso especial na memória (e.g., `Exchange`, `TestAndSet`, etc.)

Avaliação

- Um requerimento esquisito
  - Todos devem saber a quantidade de threads
    - Sempre, e instantaneamente
    - Ou utilizar uma cota superior
- Um comportamento desafortunado
  - Esperamos zero competidores (devemos correr rápido em forma normal)
  - O algoritmo deve ser $O(n)$ em caso de ter máximo número de competidores
- Muito dura nossa avaliação?
  - O algoritmo da padaria tinha esses mesmos problemas
  - Por que nos preocupamos?

Vamos ver além do óbvio

- Além da semântica abstrata
  - Exclusão mutua, progresso, e espera acotada
- Consideremos
  - O padrão de acesso típico
  - Os ambientes de execução particulares
- Ambiente
  - Mono processador vs. multiprocessador
    - Quem faz o que quando estamos tentando segurar/liberar?
  - Os threads não estão misteriosamente executando-se ou não
    - A decisão de executá-los é feita através de um algoritmo de escalonamento (*scheduler*) com certas propriedades

Ambiente mono processador

- Segurar (lock)
  - O que acontece se `xchg()` não funcionou a primeira vez?
  - Algum outro processo tem o seguro (lock)
    - Esse processo não se está executando (porque nós estamos usando o processador)
    - Estar no spin (o ciclo de espera) é uma pérda de tempo
    - Nós devemos deixar ao thread que tem a região crítica executar-se, ao invés de nós
- Liberar (unlock)
  - O que acontece com a espera acotada?
  - Quando nós marcamos o mutex como disponível, quem o toma depois?
    - Qualquer que se execute depois, só um por vez (é uma competição falsa)
    - Quão injustos são os *schedulers* reais dos threads do kernel?
    - Se o escalonador é muito injusto, o thread correto nunca vai se executar

Ambiente multiprocessador

- Segurar (lock)
  - O esperar (spin) pode estar justificado
    - Por que?
- Liberar (unlock)
  - O seguinte ganhador de `xchg()` será escolhido pelo hardware de memória
  - Quão injustos são os controladores de memória?

Test and Set

```
boolean testandset(int32 *lock){
  register boolean old;
  old = *lock;
  *lock = true;
  return (old);
}
```

- Precisamos dele em ambientes multiprocessador
  - Por que?
- Conceptualmente é mais simples que xchg?
- Outras instruções de x86
  - xadd, cmpxchg, cmpxchg8b, ...
  - Na documentação da arquitetura se detalham todas as possibilidades

Separemos a implementação

- Para ambientes multiprocessador
  - Fazer um lock no bus é prejudicial
- Solução: dividimos xchg em duas partes
  - `load_linked(addr)` traz o valor antigo da memória
  - `store_conditional(addr, val)` armazena o valor de volta

    - Se ninguém está tentando armazenar (ou o fez) nesse endereço por médio
    - Se alguém o fez, a instrução falha (estabelece um código de erro)

Implementação

```
lock: LA  R1, mutex       ; &mutex in R1
loop: LL  R2, (R1)        ; mutex->avail
      BEQ R2, R0, loop    ; avail == 0 (zero)
      MOV R3, R0          ; prepare 0
      SC  (R1), R3        ; write 0?
      BEQ R3, R0, loop    ; aborted
```

- Nosso cache bisbilhota a memória compartilhada
  - Segurar a escritura da variável não deve apagar todo o trânsito na memória
  - Bisbilhotar permite que o trânsito passe, e observa trânsito conflitivo
  - É correto abortar? Quando é correto?

LA: load address
LL: load linked word
BEQ: branch if equal
SC: system call

Intel `i860 lock bit`

- A instrução põe ao processador em modo segurado (lock mode)
  - Lock ao bus
  - Desativa interrupções
- Não é isso perigoso?
  - O timer emite uma excepção
  - Qualquer excepção (page faults, divisões por cero, etc.) libera o bus
- Por que queremos isto?
  - Implementar `test and set`, `compare and swap`, `semaphore`, sua eleição

Software misterioso para exclusão mutua

- Algoritmo de Lamport "fast mutual exclusion"
  - 5 escritas, 2 leituras (se não há disputas)
  - Não há espera acotada (em teoria, se não há disputas)
    https://www.microsoft.com/en-us/research/publication/fast-mutual-exclusion-algorithm/
    https://cs.stackexchange.com/a/62322/17831
- Algoritmos interessantes
  - Uma solução para computadores "modernos"
  - Revisem o artigo, para outros pontos de vista

E se alguém mais é responsável

- Por que não pedir ao sistema operacional por uma chamada de sistema `mutex_lock()`?
  - Simples no mono processador
    - O kernel automaticamente exclui outros threads
    - O kernel pode desativar interrupções facilmente
    - Não há necessidade de ciclos não acotados, ou esquisitos `xchg`
  - O kernel tem um poder especial num ambiente multiprocessador
    - Pode enviar interrupções remotas a outros CPUs
    - Não há necessidade de um ciclo não acotado
- Então, por que não deixar o trabalho no sistema operacional?
  - É demasiado caro, por que?

Mutex no sistema operacional

Problemas

- Overhead de desempenho: envolve troca de contexto (de modo usuário para kernel) e operações de mutex são frequentes em aplicações multi-threads.
- Escalabilidade: aumenta a contenção, afetando a aplicação.
- Alocação de Recursos: alocação e dealocação de recursos.
- Complexidade: na manutenção do código e no tratamento de erros.
- Portabilidade: diferentes interfaces dependendo do sistema operacional.
- Controle limitado.

Software trapaceador

- Exclusão mutua rápida para mono processadores
  - Bershad, Redell, Ellis, "Fast Mutual Exclusion for Uniprocessors", ASPLOS V (1992)
- Se queremos sequências de instruções ininterruptíveis
  - Pretendemos
  - Mono processador: entrelaçar as instruções requere troca de threads
  - Uma sequência curta, a maioria do tempo, não será interrompida
- Como pode funcionar isto?

Como pode funcionar isto?

- O kernel detecta a mudança de contexto na sequência atômica
  - Talvez um conjunto de instruções pequeño
  - Talvez áreas particulares na memória
  - Talvez uma bandeira `no_interruption_please = 1;`
- O kernel gerencia o caso inusual
  - Entrega mais tempo (está bem?)
  - Simula que as instruções não terminaram
  - Uma sequência idempotente (realizar a mesma sequência para obter o mesmo resultado)

Pontos importantes

- Sequência de instruções atômicas
  - Ninguém pode entrelaçar-se na sequência atômica
- Especificar as sequências que interferem (seção crítica) através de um objeto `mutex`
- Dentro do mutex
  - Condições de corrida
  - Troca atômica, compare and swap, test and set, etc.
  - Divisão em multiprocessadores (load-linked, store-conditional)
  - Os altos e baixos deste tipo de software
- Estrategia dos mutex
  - Como comportar-se segundo o ambiente de execução