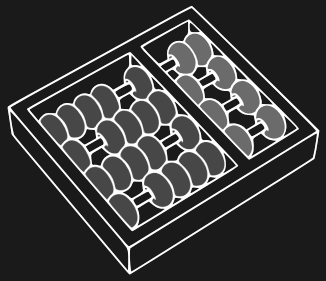

UNICAMP

Sincronização (Parte 1)

Prof. Carlos A. Astudillo

castudillo@ic.unicamp.br

Sistemas Operacionais (MC504A)

2º Semestre 2023

Adaptado de Dave Eckhard

Recapitulando

- Capítulo 5 de OS:P+P
- Os threads são como os processos
- Distinta multiplicidade (programa vs. kernel)
- Perguntas sobre condições de corrida?

Objetivos da aula

- Problemas reais
- Operações fundamentais
- Propriedades necessárias da seção crítica
- Solução a dois processos
- N processos: Algoritmo da padaria (*bakery*)

Problemas com programas multi-threads

- A execução do programa depende dos possíveis intercalamentos de acessos dos threads a objetos compartilhados
- A execução do programa pode ser não deterministico (e.g., *Heisenbugs*)
- Compiladores e HW de processamento podem reordenar instruções

Problemas com threads cooperativos

A execução do programa depende dos possíveis intercalamentos de acessos dos threads a objetos compartilhados

Assumindo que x é uma variável compartilhada no heap

Thread A	Thread B
`x = 1;`	`x = 2;`

Qual é o valor final de x?

Problemas com threads cooperativos

Outro exemplo

Assumindo que x=0 é uma variável compartilhada no heap

Thread A	Thread B
`x = x + 1;`	`x = x + 2;`

Possíveis entrelaçamentos

Uma execução

A	B
`load r1,x`	
`add r2,r1,1`	
`store x,r2`	
	`load r1,x`
	`add r2,r1,2`
	`store x,r2`

x=3

Outra execução

A	B
`load r1,x`	
	`load r1,x`
`add r2,r1,1`	
	`add r2,r1,2`
`store x,r2`	
	`store x,r2`

x=2

Mais uma

A	B
`load r1,x`	
	`load r1,x`
`add r2,r1,1`	
	`add r2,r1,2`
	`store x,r2`
`store x,r2`	

x=1

Problemas com threads cooperativos

Execução de programa pode ser não deterministica

Diferentes execuções do mesmo programa podem gerar resultados differentes

- O escalonador pode tomar diferentes decisões de escalonamento
- O processador pode funcionar a diferentes frequências
- Outro programa pode afetar taxa de acertos da cache
- Técnicas de debug podem afetar o comportamento do programa

Heisenbugs

- Bugs que desaparecem ou mudam quando examinados
- Muito mais dificíl de diagnosticar que *Bohr bugs* (deterministicos)

Quando o compilador é inteligente

- Nossos computadores são “inteligentes”, e isso cria problemas
- Imaginemos o código

```
// Thread 1

p = someComputation();
pInitialized = true;


// Thread 2

while(!pInitialized)
    ;
q = anotherComputation(p);
```

- Podemos garantir que p já foi inicializada quando q vai ser calculado?
  - Embora pareza que p está sempre inicializada antes de anotherComputation(p) ser chamado, esse não é o caso.
- O que aconteceu?

A computadora errou?

- Nossos computadores são modernos: "inteligentes"
  - Maximização de paralelismo no nível de instruções
    - O HW ou compilador podem fazer pInitialized=true antes de que a computação de p seja completada.
  - Comandos redundantes
    - O processador escreve comandos em filas
    - A memória armazena essas filas
    - Mas, comandos redundantes são fusionados
- Por que?
  - Optimização
  - Algumas operações não precisam ser feitas
  - O que acontece com as que sim precisamos?

Soluções

- Barreras de memória
  - Instruções que podem parar o processador
  - Esperar que a lista de escrita esteja vazia
  - Magia (não!)
- Os modelos de memória simples não funcionam na presença de múltiplos processadores
  - `http://www.cs.umd.edu/~pugh/java/memoryModel/`
  - `http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`
- Para nossas explicações um algoritmo de exclusão mutua e o modelo de memória serão simples
  - O que acontecia nos computadores pre-modernos
  - No mundo real, não funciona tão bem
  - Então, ter em conta: compiladores, arquitectura, linguagem, etc.

Fundamentos da sincronização

- Dois operações fundamentais
  - Sequência de instruções atômicas
  - Desescalonamento voluntario (*unscheduling*)
- Múltiplas implementações de cada uma
  - Processador único vs. vários processadores
  - Hardware especial vs. algoritmos especiais
  - Técnicas diferentes nos sistemas operacionais
  - Ajustar o rendimento em alguns casos especiais
- As características em cada um são claras
  - A operação é bem diferentes
  - São opostas, mais similares
- As abstrações nos clientes utilizam as dois operações
- Nossa literatura prefere a seção crítica (semáforos, monitores)
- Relevante
  - Variáveis mutex (mutual exclusion) e de condição (POSIX threads)
  - Java `synchronized` (3 formas)

Operações fundamentais

- Sequência de instruções atômicas
- Desescalonamento voluntario

Sequência de instruções atômicas

- Domínio do problema
  - É uma sequência curta de instruções
  - Ninguém mais pode entrelaçar a mesma sequência
  - Ou uma sequência similar
  - Tipicamente não existe competição

Exemplos de não interferência

- Simulação em multiprocessador (imaginemos Sim City, Civilization, ou outro jogo com "ticks")
  - Grão groso de cada turno (e.g., por hora)
  - Muita atividade em cada turno
  - Imaginemos $M : N$ threads, M objetos, N processadores
- A maioria de objetos não interagem num jogo por turnos
  - Devemos modelar aqueles que sim interagem
  - E.g., uma interseção não pode ser processada por vários automóveis ao mesmo tempo

Comércio

Imaginemos uma loja

Cliente 0	Cliente 1
`cash = store->cash;`	`cash = store->cash;`
`cash += 50;`	`cash += 20;`
`wallet -= 50;`	`wallet -= 20;`
`store->cash = cash;`	`store->cash = cash;`

- O que acontece na execução?
- Pode falhar?
- O que pode fazer a loja?

Observações sobre o problema do comércio

- O conjunto de instruções é pequeno
  - Está bem excluir mutuamente a os competidores
  - Podemos faze-los esperar
- A probabilidade de colisão é baixa
  - Muitas invocações que não colidem (imaginemos muitas lojas utilizando o sistema)
  - Não devemos usar um mecanismo caro para evitar a colisão
  - O caso comum (de não colisão) deve de ser rápido

Desagendamento voluntario

- Domínio do problema
  - Esperar voluntariamente
  - Já chegamos?
- Exemplo: o demônio (*daemon*) de um evento especial num jogo

```
while (date < 2019-11-17)
  cwait(date);
while (hour < 11)
  cwait(hour);
for (i=0; i<max_x; i++)
  for (j=0; j<max_y; j++)
    wreak_havoc(i,j);
```

Propriedades

- Cooperativa
- Anti atômica
  - Queremos estar entrelaçados o mais possível
- Executar-me e fazer que outros esperem é ruim
  - Ruim para eles: não estaremos prontos em muito tempo
  - Ruim para nós: não podemos estar prontos se eles não estão prontos
- Não queremos exclusão
- Queremos que outros se executem, eles nos ajudam
- O desescalonamento da CPU é um serviço do sistema operacional

Padrão de espera

- Nós

```
LOCK WORLD
while ( !(ready = scan_world()) ) {
  UNLOCK WORLD
  WAIT_FOR(progress_event)
  LOCK WORLD
}
```

- Nossos colegas

```
SIGNAL(progress_event)
```

Um pouco mais técnico

```
do {
  entrada à seção
  seção crítica:
    computação no espaço compartilhado
  fim da seção crítica
  resto da seção:
    computação privada
} while(1);
```

Nomenclatura estândar

- O que aprendemos no código anterior?
  - O que está na seção crítica?
  - Uma sequência atômica rápida
  - Precisamos dormir por um longo tempo
- Por enquanto
  - Assumamos que a seção crítica é uma sequência atômica curta
  - Estudemos a entrada e saída da seção crítica

Resumo

- Operações fundamentais:
  - Sequência de instruções atômicas
  - Desagendamento voluntario
- Seção crítica
  - Conjunto de sequências que devemos proteger
  - Precisamos uma forma de poder entrar e sair sem problemas (vamos para lá)
  - Padrão de acesso (mais detalhes a continuação)

Requerimentos da seção crítica

- Exclusão mutua
  - Quando muito, um thread se está executando na seção crítica
- Progresso
  - O protocolo escolhido deve de ter um tempo acotado de trabalho
  - Uma forma de falhar: para escolher o seguinte candidato precisamos que existam candidatos (um exemplo a continuação)
- Espera acotada
  - Não podemos esperar por sempre uma vez que entramos à seção crítica
  - A entrada é acotada por os outros participantes
  - Não necessariamente acotada por um número de instruções
  - Não garante ordem FIFO

Notação para 2 processos

- Suposições
  - Múltiplos threads (1 CPU um timer, ou múltiplos CPU's)
  - Memória compartilhada, mas não temos locks, ou instruções atômicas
- Thread i somos nós
- Thread j outro thread
- i, j são variáveis locais de threads (que também os identificam)

Primeira ideia: revezamos

```
int turn = 0;
while (turn != i)
  continue;
// seção crítica
// faço coisas :)
turn = j;
```

- Exclusão mutua: sim (vem ela?)
- Progresso: não
  - Tomar turnos rigorosos é fatal
  - Se T_i nunca tenta entrar à seção crítica, T_1 espera para sempre
  - Viola a regra de "depender de que não existam participantes"

Outra ideia: mostrar interesse

```
boolean want[2] = {false, false};

want[i] = true;
while (want[j])
  continue;
// seção crítica
// faço (mais) coisas
want[i] = false;
```

- Ao inicio ninguém esta interessado na região crítica
- Se queremos entrar à região crítica, mostramos interesse
- Exclusão mutua: sim
- Progresso: quase (por que?)

Intuição de exclusão mutua

Thread 0	Thread 1
`want[0] = true;`	
`while(want[1]);`	
entramos SC	
	`want[1] = true;`
	`while(want[0]);`
	`while(want[0]);`
	`while(want[0]);`
	`while(want[0]);`
faço coisas	
saímos SC	
`want[0] = false;`	
	`while(want[0]);`
	entramos SC

Outra execução

Thread 0	Thread 1
`want[0] = true;`	
	`want[1] = true;`
`while(want[1]);`	
	`while(want[0]);`

- Falha no progresso da execução
- Funciona para qualquer outra forma de entrelaçar as instruções
- Resultado: sad panda, and waiting panda

Solução de Peterson (1981)

- Tomar turnos quando seja preciso

```
boolean want[2] = {false, false};
int turn = 0;

want[i] = true;
turn = j;
while (want[j] && turn == j)
  continue;
// seção crítica
// faço coisas \o/ (agora sem me travar)
want[i] = false;
```

- Por que `turn = j` e não `turn = i`?

Prova da exclusão

- Assumamos o contrário: dois threads na seção crítica
- Ambos na seção crítica implica que `want[i] == want[j] == true`
- Então ambos ciclos terminaram porque `turn != j`
- Não podemos ter `turn == 0 && turn == 1`
  - Então algum terminou antes
- Sem perder generalidade, T_0 terminou primeiro porque `turn == 1` falhou
  - Então `turn == 0` antes que `turn == 1`
  - Então T_1 tem que estabelecer `turn = 0` antes que T_0 estabeleça `turn = 1`
  - Então T_0 não pode ver `turn == 0`, não pode sair do ciclo antes

Dicas da prova

- `want[i] == want[j] == true`
  - `want[]` não significa nada, então nós focamos em `turn`
- `turn[]` versus quem saiu do ciclo
  - O que acontece se entrelaçamos de distintas maneiras?

Thread 0	Thread 1
`turn = 1;`	`turn = 0;`
`while(turn == 1);`	`while(turn == 0);`

Resumo

- Mostramos interesse sobre a seção crítica
- Tomamos turnos para poder utilizá-la (em caso que ambos a precisemos)
- Cordialidade ante todo!
  - A solução de Peterson depende de entregar voluntariamente o passo ao outro processo
  - — *Depois de você,* — *não depois de você*
- Sincronizar pode ser complicado (e doloroso)
- Estudar as interações para suas implementações

Contexto

- Se há mais de dois processos
  - Generalização baseada no mostrador da padaria (ou qualquer loja basicamente)
  - Obtemos números de tickets que aumentam monotonicamente
  - Esperamos enquanto o número de cliente (que aumenta monotonicamente) seja o nosso
- Versão multiprocesso
  - A diferencia da realidade, dois pessoas (processos) podem obter o mesmo número de ticket
  - Ordenamos pelo número de ticket, alguma forma de quebrar empates
    - Número de ticket, e número de processo

Algoritmo

- Fase 1: escolher um número
  - Obter o número de processos/clientes
  - Agregar 1 ao número
- Fase 2: esperar a que chegue seu turno
  - Não necessariamente certo, vários processos podem ter o mesmo número
  - Utilizamos o número do processo, e o ticket para selecionar
    - (ticket 7, processo 99) $<$ (ticket 7, processo 101)
- Tomamos turno quando temos o ticket e o processo menor

Algoritmo

- Fase 0: contexto

```
boolean choosing[n] = {false, ...};
int number[n] = {0, ...};
```

- Fase 1: escolher um número

```
choosing[i] = true;
number[i] = max(number[0], number[1], ...) + 1;
choosing[i] = false;
```

- No pior dos casos todos escolhem o mesmo número
- Mas na seguinte ronda, todos escolhem um maior

Algoritmo

- Fase 2: escolher o menor número

```
for(j=0; j<n; j++) {
  while(choosing[j])
    continue;
  while( (number[j] != 0) &&
         ( (number[i], i) > (number[j],j) )
       )
    continue;
}
// seção crítica
number[i] = 0;
```

. . .

Continuará!

Pontos importantes

- A memória em condições de corrida pode ser complexa
- Devemos entender dois operações fundamentais:
  - Execução breve para operações atômicas
  - Devemos ceder nossa execução a longo prazo para poder obter resultados
- Três condições necessárias para uma seção crítica
  - Exclusão mutua
  - Progresso na execução
  - Espera deve ser acotada
- Devemos entender os algoritmos básicos de exclusão
  - A solução para dois processos
  - N processos: a padaria

Prática

- Para entender a exclusão mutua há que praticar
- Praticar converte os conceitos em aprendizagem
- Não é o mesmo ler o algoritmo, a entendê-lo e poder escrevê-lo
- Exercício:
  - Dois threads
  - Compartilham uma variável `quantidade`
  - Cada thread recebe um parâmetro `num`
  - Um thread (o pai) toma o `num` e o soma a `quantidade`
  - O outro thread (o filho) toma o `num` e o subtrai de `quantidade`
  - Como se implementa esta interação para que compartilhem `quantidade`?