Sistemas Distribuídos
Aula 5

Aula passada

- Sincronização
- *Race condition*
- Região crítica
- Locks
- Algoritmo de Peterson

Aula de hoje

- Atomicidade
- *test-and-set*
- Locks revisitado
- Semáforos
- Dois problemas

Atomicidade

- Mecanismos para garantir demandas da região crítica necessitam de *atomicidade*
  - algoritmo de Peterson

```
...
   lock->turn = other_thread;
   while(lock->interested[other_thread] &&
         lock->turn == other_thread);
...
```

- O que é atomicidade (aka. *linearizable, indivisible, uninterruptable)*?
- conjunto de instruções que *parece acontecer* instantaneamente
- nada pode aconceter no meio, e *todas as threads* ficam sabendo dos novos valores

Atomicidade de Instrução

- CPUs modernas tem vários cores, várias técnicas de aceleramento do pipeline (ex. o*ut-of-order execution*)
- Programas modernos tem várias *threads*, que podem rodar ao mesmo tempo
- Compiladores modernos fazem muitas otimizações

- Como garantir atomicidade de um conjunto de instruções?
- **Ideia**: instruções de máquina atômicas
  - introduzidas no código pelo compilador, a pedido do programador

Test-and-Set

- Instrução de máquina sobre variável binária
- Semântica: grava valor atual (em anterior), seta valor em 1, retorna valor anterior

```
bool test_and_set(bool *flag) {
    bool anterior = *flag;
    *flag = TRUE;
    retorna anterior
}
```

Executada como única instrução pela CPU!

- Após chamar test_and_set(&flag)
  - qual o valor de flag?
  - qual o valor retornado?
  - como setar flag em zero?

Locks com Test-and-Set

- Como usar Test-and-Set para implementar locks?

- Primeira tentativa

```
struct lock {
   bool held = 0;
}

void acquire(lock) {
   while(lock->held);
   lock->held = 1;
}

void release(lock) {
   lock->held = 0;
}
```

Condição de corrida!

- Com test_and_set

```
struct lock {
   bool held = 0;
}

void acquire(lock) {
   while(test_and_set(
            &lock->held));
}

void release(lock) {
   lock->held = 0;
}
```

- Funciona? Por que?

Desabilitar Interrupções

- Desabilitar a troca de contexto, que é a raiz do problema
  - outra alternativa com a ajuda do HW
- Garantir acesso exclusivo à CPU

```
struct lock {
   bool held = 0;
}

void acquire(lock) {
   disable_interrupts();
}

void release(lock) {
   enable_interrupts();
}
```

- Não precisa manter estado na variável lock
- Funciona? Por que?
- Multicore? Região crítica muito longa?
- Problemas à vista

Desvantagens

Spinlocks

- Threads ficam executando em *busy wait*
  - consomem ciclos de CPU
- Maior rc, mais tempo girando
  - interrompe thread com lock

```
...
   acquire(lock);

// região crítica

   release(lock);
...
```

Desabilitar interrupções

- entrega controle da CPU (ou core)
- atrasa entrega de eventos (ex. sinais)
- apenas o SO deve fazer isto

Sincronização de Alto Nível

- Spinlocks e desabilitar interrupções adequadas apenas para rc pequenas e simples
  - muito primitivas!
- Sincronização de alto nível que permite:
  - bloquear as threads que aguardam acesso
  - permitir interrupções dentro da região crítica
- **Ideia**: usar spinlocks e desabilitar interrupções como primitivas para implementar esta sincronização
- Primitivas de sincronização de alto nível oferecidas pelo SO
  - através de chamadas ao sistema (*system call*)

Semáforos

- Estrutura de dados que permite acesso com exclusão mutua a região crítica
  - bloqueia threads em espera, permite interrupções dentro da rc
  - descritas por Dijkstra em 1968
- Funcionam como “contadores atômicos”
- Duas operações (assim como locks)
  - ***wait***(semaforo): bloqueia até semáforo “estar aberto”
  - ***signal***(semaforo): permite a entrada de uma thread em espera

Bloqueando em Semáforos

- Fila de espera associada a cada processo (FIFO)
- Ao chamar ***wait***():
  - se “semáforo aberto”, thread continua, possivelmente fechando o semáforo
  - se “semáforo fechado”, thread entra no final da fila e bloqueia
- Ao chamar ***signal***():
  - semáforo fica aberto. Próxima thread na fila entra na rc, e fecha semáforo
  - se fila vazia, semáforo continua aberto

Dois Tipos de Semáforos

- Semáforo *mutex*
  - um acesso por vez (o que estamos usando)
  - garante acesso exclusivo à região crítica
- Semáforo *contador*
  - múltiplos acessos por vez, recurso que pode ser utilizado de forma concorrente
  - múltiplas threads podem passar pelo semáforo ao mesmo tempo
  - número de threads determinado pelo valor do semáforo. Mutex: valor=1, Contador: valor=N

Utilizando Semáforos

```
struct semaphore S;

retirada(conta, valor) {
    wait(S);
    saldo = get_saldo(conta);
    saldo = saldo – valor;
    put_saldo(conta, saldo);
    signal(S);
    retorna saldo;
}
```

- Threads são bloqueadas
- Ordem de execução definida pela fila

```
retirada(conta, valor) {
    wait(S);
    saldo = get_saldo(conta);
    saldo = saldo – valor;
```

```
retirada(conta, valor) {
    wait(S);
```

```
retirada(conta, valor) {
    wait(S);
```

```
    put_saldo(conta, saldo);
    signal(S);
    retorna saldo;
}
```

```
...
    signal(S);
    retorna saldo;
}
```

Leitores com Escritor

- Dois exemplos de problemas mais interessantes
- Primeiro: leitores com escritor
  - objeto (variável) compartilhado por várias threads
  - várias threads querendo ler e escrever
  - Ao escrever, apenas uma thread por vez
  - Ao ler, permitir qualquer número de threads

- Como usar semáforos para coordenar o acesso?

Leitores com Escritor

- Três variáveis:
- *readcount*: quantas threads estão lendo
- Semáforo *mutex*: controle de acesso a readcount
- Semáforo *r_or_w*: controle de acesso a escrita

```
// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
}
```

```
reader {
    wait(mutex);       // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex);   // unlock readcount
    Read;
    wait(mutex);       // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex);   // unlock readcount}
}
```

Perguntas e Observações

- O que acontece quando a thread termina a escrita?
- O que acontece quando temos múltiplas threads lendo?
- Quantas threads de escrita e leitura podem estar bloqueadas em *w_or_r* ao mesmo tempo?
- Quando que uma thread de escrita pode ser desbloqueada?

Nada de trivial!

Produtores-Consumidores

- Conjunto de recursos compartilhados (buffer) entre threads produtores e consumidores
- Produtor: insere recurso no buffer limitado
- Consumidor: libera recurso do buffer limitado
- Produtores e consumidores possuem taxas diferentes
  - não podemos serializar um depois do outro
  - recursos produzidos/consumidos de forma independente

- Como usar semáforos para coordenar o acesso?

Produtores-Consumidores

- Três semáforos:

```
Semaphore mutex = 1;   // mutual exclusion to shared set of buffers
Semaphore empty = N;   // count of empty buffers (all empty to start)
Semaphore full = 0;    // count of full buffers (none full to start)
```

```
producer {
  while (1) {
    Produce new resource;
    wait(empty); // wait for empty buffer
    wait(mutex); // lock buffer list
    Add resource to an empty buffer;
    signal(mutex); // unlock buffer list
    signal(full);      // note a full buffer
  }
}
```

```
consumer {
  while (1) {
    wait(full);        // wait for a full buffer
    wait(mutex);    // lock buffer list
    Remove resource from a full buffer;
    signal(mutex); // unlock buffer list
    signal(empty); // note an empty buffer
    Consume resource;
  }
}
```

Observações

- Por que precisamos do *mutex*?
- Onde estão as regiões críticas?
- Podemos trocar a ordem das chamadas de *wait* aos semáforos mutex e empty/full?
- *interlock*: padrão de chamadas cruzadas wait/signal em empty/full (por threads diferentes)
  - frequentemente utilizada

- Leitores com escritor, e Produtores-Consumidores são instâncias de problemas recorrentes
  - sempre podem ser resolvidos com semáforos

Sincronização não é trivial!