

# Sistemas Distribuídos

## Aula 5

### Roteiro

- 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);  
...
```

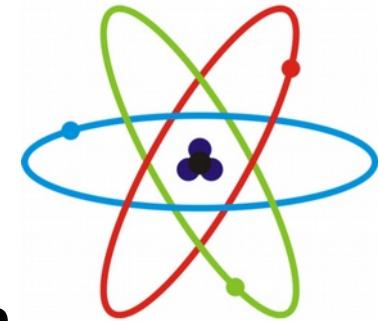
- 
- Duas threads executando este código ao mesmo tempo em *cores* diferentes

## Condição de Corrida!

- Possível violação da exclusão mútua

# Atomicidade

- Também conhecido por *linearizable*, *indivisible*, *uninterruptable*
- Conjunto de instruções que *parecem* ser executadas instantaneamente
  - nenhuma outra instrução com dependência pode ser executada ao mesmo tempo
- *Todas as threads* ficam sabendo do resultado da instrução



# Atomicidade de Instrução

- CPUs modernas tem vários cores, várias técnicas de aceleramento do pipeline (ex. *out-of-order execution*)
- Programas modernos tem várias *threads*, que podem rodar ao mesmo tempo em diferentes cores
- 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 uma variável booleana
- 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?

R: `flag` vale 1

R: o valor anterior

R: `flag = 0`

# 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!

- Apenas uma *thread* vai executar `test_and_set`

# Desabilitando Interrupções

- Desabilitar a troca de contexto, que é a raiz do problema
  - outra alternativa com a ajuda do HW
- Desabilitar todas as interrupções que o processo poderia sofrer (ex. sinais)
- 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
- Processo (*thread*) não pode ser interrompido dentro da região crítica

# Limitações

- Entrega controle da CPU (core) ao processo (thread)
  - região crítica pode ser muito longa (uso do recurso por tempo prolongado)
  - bug dentro da região crítica pode travar o recurso
- Kernel level threads e múltiplos cores
  - duas threads não podem executar ao mesmo tempo na região crítica
  - garantir que temos apenas uma thread em execução quando esta entrar na região crítica
- Muito delicado para uso geral (ex. um programa qualquer)

# Desvantagens

## Spinlocks

- Threads ficam executando em *busy wait*
  - consomem ciclos de CPU
- Quanto maior a rc, mais tempo as threads ficam “girando”
  - thread girando interrompe a 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 são adequadas apenas para rc pequenas e simples
  - mecanismos de sincronização muito primitivos
- Sincronização de alto nível deve
  - bloquear as threads que aguardam acesso (remover de execução, colocar no estado *waiting*)
  - permitir processo receber interrupções dentro da região crítica
- **Ideia:** usar spinlocks e desabilitar interrupções para implementar esta sincronização
- Sincronização oferecida pelo SO
  - através de chamadas ao sistema (*system call*)

# Semáforos

- Estrutura de dados que permite acesso com exclusão mútua a região crítica
  - bloqueia threads colocando em *waiting*, 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” (contador > 0)
  - ***signal***(semaforo): “abre” semáforo (incrementa contador), permitindo a entrada de uma thread em espera

# Bloqueando em Semáforos

- Fila de espera associada a cada processo (FIFO)
- Ao chamar ***wait()***:
  - se “semáforo aberto” (contador > 0), thread continua e decrementa contador
  - se “semáforo fechado” (contador = 0), thread entra no final da fila e fica em espera
- Ao chamar ***signal()***:
  - “abre” semáforo (incrementa contador)
  - se fila não vazia, próxima thread na fila entra na rc, decrementando contador
  - se fila vazia, semáforo continua “aberto”

# Dois Tipos de Semáforos

## ■ Semáforo *mutex*

- uma thread por vez (o que estamos usando)
- garante acesso exclusivo à região crítica

## ■ Semáforo *contador*

- múltiplas threads por vez
  - recursos que podem ser utilizado de forma concorrente
- Número de threads que podem entrar é determinado pelo valor do semáforo
- Mutex: valor=1, Contador: valor=N

# Utilizando Semáforos

```
struct semaphore S = 1;  
  
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 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 este 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 no `wait(w_or_r)` ?
- Quando que uma thread de escrita pode ser desbloqueada?
- O `signal(w_or_r)` de leitura pode encontrar outra thread de leitura bloqueada no `wait(w_or_r)` ?

**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 queremos serializar produção e consumo
  - 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;
    }
}
```

# Perguntas e Observações

- Por que precisamos do *mutex*?
- Onde estão as regiões críticas?
- Podemos ter mais de N threads bloqueadas em *wait(empty)* ?
- Quanto vale *empty* + *full* (valor dos contadores)?
- Podemos trocar a ordem das chamadas de *wait* dos semáforos *mutex* e *empty/full* ?
- *interlock*: padrão de chamadas cruzadas *wait/signal* em *empty/full* (por threads diferentes)
  - frequentemente utilizado

# Observações

- Leitores com Escritor e Produtores-Consumidores
  - instâncias de problemas recorrentes
  - também representativo de outros problemas
- Problemas de sincronização podem sempre ser resolvidos com semáforos

**Sincronização não é trivial!**