

Sistemas Distribuídos

Aula 6

Roteiro

- Limitação dos semáforos
- Monitores
- Variáveis de condição
- Semântica do *signal*

Sincronização de Alto Nível

- Sincronização de alto nível permite
 - bloquear as threads que aguardam acesso
 - permitir interrupções dentro da região crítica

Semáforos!

- *wait()*, *signal()*, implementadas pelo sistema operacional
- SO utiliza instruções atômicas de *test-and-set* e instruções para desabilitar interrupções

Desvantagens Semáforos



- Quais as desvantagens ou limitações de semáforos?
- São variáveis globais do processo
- Não há conexão entre variável e código sendo protegido
 - não são muito elegantes ou intuitivas
- Usadas para definir região crítica (exclusão mútua) e coordenar execução das threads
- Uso arbitrário e por isto difíceis de usar e propensa a *bugs*



Buscando Ajuda

- Quem pode ajudar a tornar sincronização mais elegante, menos sujeita a bugs?

Linguagem de Programação!

- Encapsular funcionalidade de sincronização e coordenação como parte da linguagem
- Facilita sincronização, evita bugs
- Disponível em C++, Java, Python, Ruby, etc

Monitores

- Funcionalidade da linguagem de programação para acessar dados compartilhados
 - código de sincronização adicionado pelo compilador (ex. usando semáforos)
- Monitor encapsula:
 - estrutura de dados compartilhada entre threads
 - funções que acessam os dados compartilhados
 - coordenação das threads que chamam as funções
- Garante que acesso *multi-threaded* aos dados será feito de forma segura
 - *se o programador não atrapalhar!*

Semântica de Monitores

- Considere um objeto/módulo encapsulado por um monitor
 - também chamado de *thread-safe*
- Exclusão mútua
 - apenas uma thread executando cada função/método por vez
 - outras threads bloqueiam aguardando o retorno de função da thread em execução
 - se thread em execução bloquear em uma *condição*, outras threads ficam aguardando

Exemplo

Encapsulamento da
linguagem de programação

```
Monitor Conta {  
    int conta;  
    double saldo;  
  
    retirada(valor) {  
        saldo = get_saldo(conta);  
        saldo = saldo - valor;  
        put_saldo(conta, saldo);  
        retorna saldo;  
    }  
}
```

- Threads bloqueadas na chamada de função
- Desbloqueadas no retorno
- Bem mais fácil de usar!

```
retirada(valor) {  
    saldo = get_saldo(conta);  
    saldo = saldo - valor;  
}
```

```
retirada(valor) {
```

```
retirada(valor) {
```

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

```
    ...  
    retorna saldo;  
}
```



Variáveis de Condição

- Como esperar dentro do monitor?
 - ex. produtor-consumidor
- Variáveis de condição oferecem coordenação
- Suportam duas operações
 - *wait()*: bloqueia a thread e aguarda até ser sinalizada
 - *signal()*: acorda uma thread bloqueada em *wait()*
- Variável de condição (vc) não é booleana
 - “if (vc) then ...” não faz sentido
 - “if (a <= 0) then wait(vc) ...” faz sentido

Variáveis de Condição

- Semântica totalmente diferente de semáforos
 - apesar dos mesmos nomes de *wait()* e *signal()*
- Acesso às funções do monitor controlada por mutex
 - apenas uma thread em cada função
- *wait()* bloqueia a thread
 - thread em execução é bloqueada até receber um signal
 - nenhuma outra thread entra na função
- *signal()* acorda uma thread para continuar execução
 - se não há threads em *wait()*, signal é perdido
- Variáveis de condição não tem memória
 - funcionam como sinais

Comparação com Semáforos

- VC podem ser implementadas com semáforos
 - apesar da semântica bem distinta
- semáforo::*wait()* decrementa contador ou bloqueia a thread (caso contador = 0)
- vc::*wait()* bloqueia thread em execução
- semáforo::*signal()* incrementa contador, e pode desbloquear uma thread
- vc::*signal()* acorda uma thread bloqueada no *wait()* para continuar execução
- semáforo tem memória (contadores), vc não

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

Sincronização com Semáforos

■ 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;
  }
}
```

Com Monitores

- Podemos focar na memória compartilhada e nas funções de acesso a memória *put_res()* e *get_res()*

```
Monitor bounded_buffer {  
    Resource buffer[N];  
    // Variables for indexing buffer  
    Condition not_full, not_empty;  
  
    void put_resource (Resource R) {  
        while (buffer array is full)  
            wait(not_full);  
        Add R to buffer array;  
        signal(not_empty);  
    }  
}
```

```
Resource get_resource() {  
    while (buffer array is empty)  
        wait(not_empty);  
    Get resource R from buffer array;  
    signal(not_full);  
    return R;  
}  
} // end monitor
```

Perguntas

- Quantas threads podem estar executando o *while* do *put_res()* ou *get_res()*
- *put_res()* chamado com buffer cheio?
- Múltiplas threads chamando *put_res()* e *get_res()*?
- O que acontece com o *signal(not_empty)* quando há mais de um recurso no buffer?
- O que garante que a execução simultânea de *put_res()* e *get_res()* não vai causar problemas no acesso ao buffer?

Semântica do *Signal*



- Considere thread em execução e a chamada *signal(vc)*
- O que acontece no instante seguinte?
Quem executa?

- Hoare monitors, *blocking cv* (proposta original)
 - *signal()* troca o contexto imediatamente, colocando em execução a thread em wait (caso exista)
 - condição garantida de ser verdadeira na thread que entra em execução
- Mesa monitors, *non-blocking cv* (Java, Mesa)
 - *signal()* coloca thread em wait para *ready*, mas continua execução
 - condição não necessariamente verdadeira quando thread entrar em execução

Usando Hoare x Mesa

■ Hoare

```
if (a < 0) {  
    wait(positive)  
}  
return sqrt(a);
```

- Mais intuitivo, mais controlado

■ Mesa

```
while (a < 0)  
    wait(positive)  
return sqrt(a);
```

- Necessário verificar condição depois de sair do *wait()*
- Mais fácil de implementar

Importante saber qual semântica!

- depende da linguagem de programação