Sistemas Distribuídos
Aula 6

Aula passada

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

Aula de hoje

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

Sincronização de Alto Nível

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

- Como conseguimos isto?

Semáforos!

  - *wait*(), *signal*(), implementadas pelo sistema operacional
  - utilizando instruções atômicas de *test-and-set* e desabilitando interrupções

Desvantagens Semáforos

- Quais desvantagens ou limitações de semáforos?

- Não são muito elegantes ou intuitivas
- São variáveis globais do processo
- Não há conexão entre variável e código sendo protegido
- Usada para região crítica (exclusão mútua) e coordenação
- Uso muito livre, mais difíceis de usar e susceptível 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
- Facilitar uso, evitar bugs
- Disponível em C++, Java, Python, Ruby, etc

Monitores

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

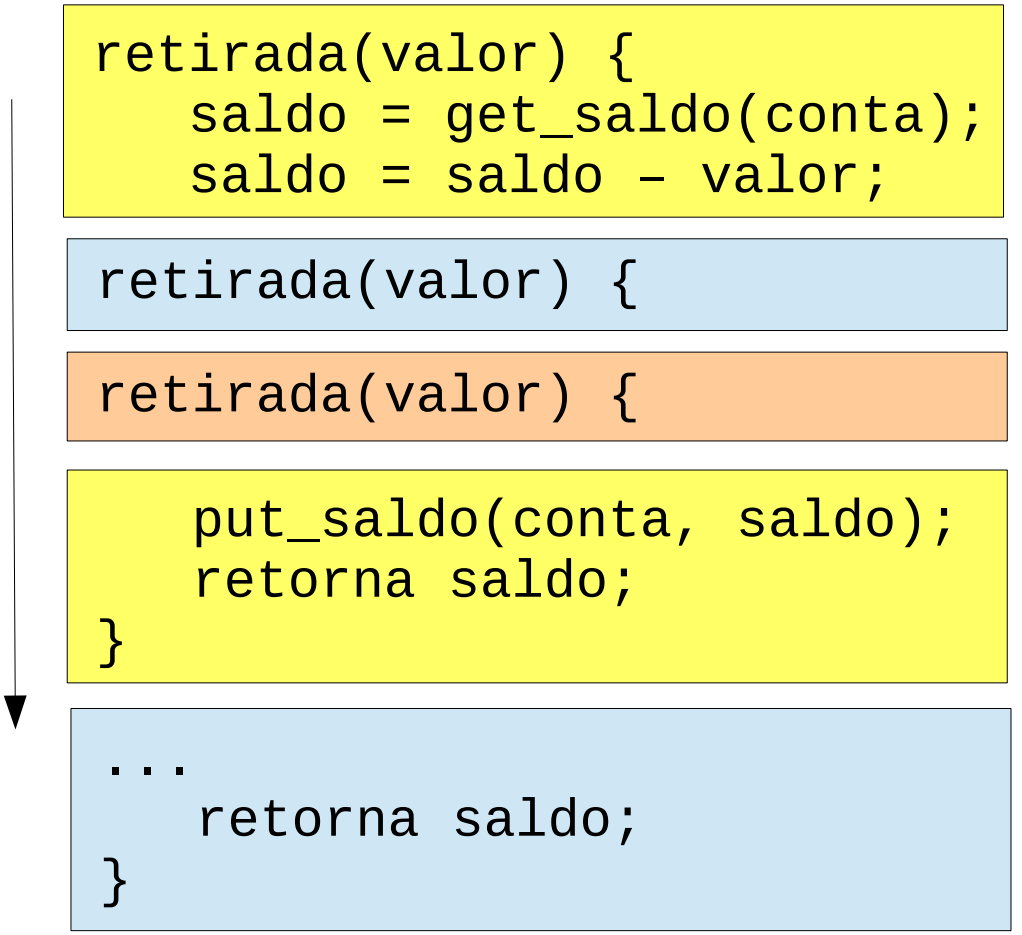
Semântica de Monitores

- Considere um objeto/módulo encapsulado por um monitor
  - também chamado de *thread-safe*
- Exclusão mútua
  - apenas uma função/método em execução por vez (thread está no monitor)
  - outras threads bloqueiam aguardando o retorno da thread em execução
  - se a thread em execução bloquear (em uma *condição*), outra thread pode entrar

Exemplo

```
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!

Variáveis de Condição

- Como esperar dentro do monitor?
  - ex. produtor-consumidor

- Variáveis de condição, forma de coordenação
- Suportam três operações
  - *wait()*: libera acesso a RC, aguarda até ser sinalizada
  - *signal()*: acorda uma thread aguardando em *wait*()
  - *broadcast()*: acorda todas as threads
- 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*(); o nome é somente um nome...
  - podem ser implementadas uma com a outra
- Acesso às funções do monitor controlada por mutex
  - wait() bloqueia a thread e libera o mutex
  - para chamar wait() thread tem que estar na rc
  - semáforo::*wait*() bloqueia a thread
- *signal*() acorda uma thread para entrar na rc
  - se não há threads em *wait*(), sinal é perdido
  - semáforo::*signal*() incrementa contador
  - semáforo tem memória, 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 podemos serializar um depois do outro
  - recursos produzidos/consumidos de forma independente

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;
  }
}
```

Com Monitores

- Podemos focar na memória compartilhada e nas funções de acesso a memória *put*() e *get*()

```
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
```

- O que acontece ao final de um *put()* ou get*()*?
- *put()* chamado com buffer cheio? Múltiplas threads?

Semântica do *Signal*

- Considere thread em execução e 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 thread em wait
  - condição garantida de ser verdadeira na thread que entra em execução
- Mesa monitors, *non-blocking cv* (Java, Mesa)
  - signal() coloca thread em wait em ready, mas continua execução
  - condição não necessariamante verdadeira quando thread entra em execução

Usando Hoare x Mesa

- Hoare

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

- "Menos sujeito" à erros
- Mais fácil de acompanhar o código

- Mesa

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

- Necessário verificar condição depois de sair do *wait*()
- "Mais fácil" de usar, mais eficiente para implementar

Importante saber qual semântica!

- depende da linguagem de programação