Mecanismos de Sincronização com Variáveis Compartilhadas

Fundamentos dos Sistemas de Tempo Real

2ª Edição

Rômulo Silva de Oliveira

Edição do Autor, 2020

www.romulosilvadeoliveira.eng.br/livrotemporeal

Como descrever os padrões de interação mais frequentes na programação concorrente com variáveis compartilhadas ?

Introdução

- Problema da seção crítica é o mais frequente,
  mas não é o único

- Existem outras situações onde é necessário sincronizar tarefas que colaboram através de variáveis compartilhadas
  - Cenários mais complexos que a simples seção crítica

- Mecanismos adicionais além do mutex são necessários
- Por exemplo
  - Semáforos
  - Monitores

Problemas Clássicos de Sincronização 1/8

- Tarefas podem apresentar outros padrões de interação mais complexos do que a simples necessidade de exclusão mútua
- Na prática existem alguns padrões de interação entre tarefas que são encontrados com relativa frequência
  - E não podem ser resolvidos apenas com mutex

- São padrões de interação os quais podemos considerar como **padrões de projeto (*design patterns*)** da programação concorrente com variáveis compartilhadas

Problemas Clássicos de Sincronização 2/8

- Em geral, tais padrões incluem
  - A sempre presente necessidade de exclusão mútua
  - Relações de precedência entre tarefas

- Uma tarefa precisa esperar até que outra tarefa conclua alguma operação, para então prosseguir
  - Esta espera é uma relação de sincronização entre essas tarefas

- Na literatura de programação concorrente os padrões de interação mais frequentes são descritos na forma de cenários exemplo que capturam a essência da situação

- Tais cenários são chamados de **problemas clássicos de sincronização** (***classic problems of synchronization***)

Problemas Clássicos de Sincronização 3/8

- **Problema dos produtores e consumidores (*bounded–buffer problem*)**
- Existe um conjunto de tarefas repetidamente produzindo dados
- e um conjunto de tarefas repetidamente consumindo estes dados
- Dados podem ser de qualquer tipo (inteiro, string, struct, ...)
- Qualquer tarefa consumidora pode receber o dado gerado por qualquer tarefa produtora, desde que esteja livre

- A solução consiste em implementar um buffer (área de armazenamento temporário) onde produtores depositam dados e consumidores retiram dados
- O buffer consiste de um array, usado de maneira circular

- Necessário bloquear produtores quando o array fica lotado
- E bloquear consumidores quando o array fica vazio

Problemas Clássicos de Sincronização 4/8

- **Problema dos leitores e escritores (*readers–writers problem*)**
- Existe um conjunto de dados que é compartilhado entre várias tarefas
- A grande maioria das tarefas deseja apenas ler estes dados
  - O que pode ser feito concorrentemente
  - Sem a necessidade de exclusão mútua entre elas
- Existem algumas tarefas que atualizam estes dados
  - Neste caso, é necessária a exclusão mútua
  - Tarefas escritoras precisam de acesso exclusivo durante a alteração dos dados
- O problema pode ser enunciado requerendo
  - Prioridade para os leitores
  - Prioridade para os escritores
  - A ordem de chegada das tarefas é respeitada

Problemas Clássicos de Sincronização 5/8

- **Problema dos filósofos jantadores (*dining-philosophers problem*)**
- Cinco filósofos (tarefas) estão sentados em torno de uma mesa circular
- Cada um deles tem um prato para seu uso exclusivo (recurso privativo)
- Existe um garfo entre cada dois filósofos (recurso compartilhado)
- Existe uma panela com macarrão no centro da mesa
- Para comer, cada filósofo usa seu prato privativo
  - Porém precisa simultaneamente dos dois garfos ao seu lado
  - Os quais são compartilhados com os filósofos vizinhos

- O problema consiste em usar mecanismos de sincronização entre tarefas
  - Para impor o uso correto dos garfos
  - Sem que nenhum filósofo morra de fome (postergação indefinida)
  - Sem que o sistema entre em um impasse global (deadlock)

Problemas Clássicos de Sincronização 6/8

- **Problema do barbeiro dorminhoco (*sleeping barber problem*)**
- Existe uma barbearia onde trabalha apenas um barbeiro
- Existe uma cadeira de barbeiro usada no momento do corte da barba
- Existem algumas cadeiras de espera
  - Para os clientes usarem enquanto esperam sua vez de serem atendidos
- Quando não existem clientes, o barbeiro senta na cadeira de barbeiro e dorme
- Quando um cliente chega
  - Ele precisa acordar o barbeiro dorminhoco
  - Sentar na cadeira do barbeiro para ser atendido
- Se um cliente chega, porém o barbeiro está ocupado
  - O cliente senta em uma das cadeiras de espera
  - No caso de todas as cadeiras de espera estarem ocupadas, o cliente vai embora
- Uma solução deve respeitar as restrições do enunciado
  - Impedir postergação indefinida
  - Evitar um impasse que trave o funcionamento da barbearia

Problemas Clássicos de Sincronização 7/8

- **Problema da barreira (*barrier problem*)**
- Existem várias tarefas que cooperam dividindo o trabalho a ser feito
- Existe um ponto do algoritmo (a barreira) no qual todas as tarefas precisam chegar antes que qualquer uma delas possa prosseguir
- A barreira é um ponto de encontro de todas as tarefas no código

- Situação prática que surge quando algoritmos paralelizados são usados em computadores com múltiplos processadores (multicore)

Problemas Clássicos de Sincronização 8/8

- O **Mutex** é um excelente mecanismo para lidar com o problema da seção crítica
- Porém ele não é suficiente para lidar de forma conveniente com os todos esses problemas clássicos de sincronização
- **Outros mecanismos** são necessários

Resumo

- Introdução
- Problemas Clássicos de Sincronização
- Semáforos
- Monitores
- Monitores com a Linguagem C e Pthreads

Mecanismos de Sincronização com Variáveis Compartilhadas

Fundamentos dos Sistemas de Tempo Real

2ª Edição

Rômulo Silva de Oliveira

Edição do Autor, 2020

www.romulosilvadeoliveira.eng.br/livrotemporeal

Mecanismos de Sincronização com Variáveis Compartilhadas

Parte II:
O que são os semáforos ?

Semáforos 1/8

- **Semáforo (*semaphore*)**
- Criado pelo matemático holandês E. W. Dijkstra nos anos 1960
- Semáforo é um tipo abstrato de dado que possui como atributos
  - Um valor inteiro, cujo valor inicial pode variar
  - Uma fila de tarefas bloqueadas no semáforo
- Duas primitivas são realmente essenciais
- **P** (do holandês proberen, testar)
- **V** (do holandês verhogen, incrementar)
- Alguns autores utilizam
  - DOWN no lugar do P
  - UP no lugar de V

Semáforos 2/8

- Operação P(S) serve para bloquear a tarefa que a executa
  - Quando o valor do semáforo S for menor ou igual a zero

P(S):

S.valor = S.valor - 1;

Se S.valor < 0

Então bloqueia a tarefa, insere em S.fila

Semáforos 3/8

- Operação V(S) serve para liberar uma tarefa previamente bloqueada
- Primeiramente o valor inteiro de S é incrementado
- Caso a fila de tarefas bloqueadas em S não esteja vazia
  - Primeira tarefa é liberada para execução

V(S):

S.valor = S.valor + 1;

Se S.fila não está vazia

Então libera para execução primeira tarefa de S.fila

Semáforos 4/8

- A implementação de P(S) e V(S) deve ser atômica
- Caso duas tarefas tentem realizar P ou V sobre um mesmo semáforo
  - Uma das operações será completamente executada antes da outra
  - ou vice-versa
- Semáforos são implementados pelo kernel do sistema operacional
  - Ou em uma biblioteca de programação que se vale de algum mecanismo equivalente oferecido pelo kernel
- Atomicidade de P e V pode ser obtida, dentro do kernel, pelos mecanismos de baixo nível
  - Desabilitação de interrupções
  - Spin-lock, no caso de multiprocessamento

Semáforos 5/8

- O problema da exclusão mútua pode ser facilmente resolvido com apenas um semáforo S
- Basta iniciar seu valor inteiro com 1

P(S);

Seção crítica

V(S);

Semáforos 6/8

- Semáforos são amplamente disponíveis para uso
  - Existem implementações de semáforos para um grande número de sistemas operacionais e linguagens de programação
- Uma versão simplificada de semáforo é quando ele pode assumir apenas os valores 0 e 1
  - São os **semáforos binários (*binary semaphore*)**
  - Funcionam de maneira semelhante ao mutex
- O semáforo completo, como descrito nesta seção, é as vezes chamado de **semáforo contador (*counting semaphore*)**

Semáforos 7/8

- O **semáforo contador** é mais poderoso do que o mutex
- O fato dele possuir um valor inteiro lhe concede uma memória
- Por exemplo, suponha que sejam executadas 1000 operações V(S)
  - Levando o valor inteiro do semáforo S até 1000
  - Serão necessárias 1000 operações P(S) até que uma tarefa seja bloqueada
  - O semáforo S tem “memória” das operações V executadas sobre ele
- Considere agora situação semelhante com o mutex
  - Após 1000 operações “unlock()” o mutex estará livre
  - Uma operação “lock()” deixará o mutex ocupado
  - Uma segunda operação “lock()” bloqueará a tarefa em questão
  - O mutex não tem memória dos 1000 unlocks executados antes

Semáforos 8/8

- Semáforos tem um poder de expressão tão grande quanto qualquer outro mecanismo de sincronização
- Porém seu emprego deixa o código difícil de entender
- Alguns problemas de sincronização exigem uso complexo de semáforos
  - Reduz a legibilidade do código
  - Aumenta o custo de desenvolvimento e manutenção

- Monitores tem o mesmo poder de expressão dos semáforos
  - Porém tendem a gerar um código mais legível e fácil de entender

Resumo

- Introdução
- Problemas Clássicos de Sincronização
- Semáforos
- Monitores
- Monitores com a Linguagem C e Pthreads

Mecanismos de Sincronização com Variáveis Compartilhadas

Fundamentos dos Sistemas de Tempo Real

2ª Edição

Rômulo Silva de Oliveira

Edição do Autor, 2020

www.romulosilvadeoliveira.eng.br/livrotemporeal

Mecanismos de Sincronização com Variáveis Compartilhadas

Parte III:
O que são os monitores ?

Monitores 1/7

- **Monitores** foram criados por C. A. R. Hoare em 1974
- Seu propósito foi definir um mecanismo de sincronização entre tarefas
  - Com o mesmo poder de expressão do semáforo
  - Mas que resultasse em programas mais legíveis
- Na maioria dos programas, a maior parte do código é sequencial
- Em algumas poucas situações, as tarefas precisam interagir entre elas
  - Dando origem a situações de exclusão mútua e outras mais complexas
- Monitores são módulos onde as interações entre tarefas acontecem
  - Código fora dos monitores é sequencial
  - Todos os problemas relacionados com o acesso a variáveis compartilhadas acontecem apenas dentro dos monitores

Monitores 2/7

- O monitor é um módulo que encapsula variáveis compartilhadas e operações sobre elas
- Variáveis compartilhadas não podem ser acessadas diretamente de fora do monitor
- A única forma de acessá-las de fora do monitor é chamar as funções públicas do monitor, as quais formam a sua interface
- Como em qualquer módulo, também é possível a existência de funções internas
- O monitor é passivo
  - Um conjunto de funções chamadas por tarefas
  - Que foram criadas fora do monitor
  - E executam fora do monitor a maior parte do tempo

Monitores 3/7

- O monitor possui algumas características adicionais àquelas normalmente associadas com módulos
- Por definição, existe exclusão mútua automática dentro do monitor
  - Somente uma tarefa pode executar dentro do monitor a cada momento
  - Se uma primeira tarefa estiver executando uma função do monitor e outra tarefa chamar aquela mesma ou qualquer outra função do monitor a segunda tarefa ficará bloqueada até que a primeira tarefa saia do monitor
- Pode ser interessante criar vários monitores no mesmo programa
  - Tarefas sem nenhuma relação entre si podem acessar monitores diferentes
  - Não são gerados bloqueios desnecessários
  - Uma aplicação pode ter tantos monitores quantos forem desejados

Monitores 4/7

- Outra característica própria do monitor é a existência de **variáveis condição (*condition variables*)**
- A variável condição é um tipo abstrato de dado cujo único atributo é uma fila de tarefas bloqueadas esperando por determinada condição
- Variáveis condição oferecem duas operações fundamentais
- Tarefa que executa a operação WAIT(CV) fica imediatamente bloqueada
  - É retirada da fila do processador e inserida na fila da variável condição CV
  - Ela para de executar dentro do monitor, permitindo que outras tarefas possam entrar
- Tarefa que executa a operação SIGNAL(CV) libera tarefas bloqueadas em CV
  - Uma delas é liberada para execução
  - Caso a fila da variável condição CV esteja vazia, SIGNAL(CV) é inócuo

Monitores 5/7

- É preciso compatibilizar a operação SIGNAL(CV) com a regra básica do monitor:
  - Somente uma tarefa pode estar ativa dentro do monitor a cada momento
- Após o signal teríamos duas tarefas executando código do monitor
  - A tarefa que executou o SIGNAL(CV)
  - E a tarefa liberada da fila de CV
- Esta compatibilização pode ocorrer de formas diversas
- Na implementação mais frequente a tarefa que executou o SIGNAL(CV) continua sua execução
  - Somente quando ela deixa o monitor é que a tarefa liberada pode voltar a executar
- Outra possibilidade é, na ocorrência de um SIGNAL(CV) que libera uma tarefa, colocar a tarefa liberada imediatamente para executar
  - Implica em bloquear a tarefa que executou o SIGNAL(CV) até o monitor ficar livre
- Uma terceira possibilidade é fazer com que a tarefa que executa a operação SIGNAL(CV) seja imediatamente expulsa do monitor
  - Como um “return” automático para fora da função do monitor

Monitores 6/7

- A idéia por trás das variáveis condição é que
  - As vezes uma tarefa entra no monitor
  - Mas descobre que precisará esperar por algum evento futuro
- A operação WAIT permite que ela fique bloqueada esperando
  - Sem impedir que outras tarefas acessem as funções do monitor
- Quando, no futuro, o evento esperado acontecer
  - Outra tarefa sinalizará esta ocorrência através da execução de SIGNAL
  - Liberará a tarefa que estava esperando
- Caso existam tarefas esperando por diferentes eventos futuros
  - Para cada evento deve ser criada uma variável condição correspondente

Monitores 7/7

- A descrição de monitores é abstrata e genérica
- Para ser usado, o conceito de monitor precisa ser implementado em alguma linguagem de programação ou biblioteca
- Algumas linguagens de programação incorporam totalmente a idéia
  - Euclid Concorrente
- Em algumas outras linguagens, podem ser facilmente implementados
  - Ada
  - Java
- Mesmo em linguagens de programação sequenciais, podem ser implementados com a ajuda de uma biblioteca
  - C
  - C++

Resumo

- Introdução
- Problemas Clássicos de Sincronização
- Semáforos
- Monitores
- Monitores com a Linguagem C e Pthreads

Mecanismos de Sincronização com Variáveis Compartilhadas

Fundamentos dos Sistemas de Tempo Real

2ª Edição

Rômulo Silva de Oliveira

Edição do Autor, 2020

www.romulosilvadeoliveira.eng.br/livrotemporeal

Mecanismos de Sincronização com Variáveis Compartilhadas

Parte IV:
Como implementar Monitores com a linguagem C e a biblioteca Pthreads ?

Monitores com a Linguagem C e Pthreads 1/14

- O monitor é um módulo
- Na linguagem C, um programa organizado em módulos é um programa composto por vários arquivos do tipo ".c", onde cada arquivo é um módulo
- A palavra reservada "static" na declaração de uma função indica que a mesma somente pode ser diretamente chamada dentro do módulo onde foi definida
- A palavra reservada "static" na frente de uma variável global também indica que a mesma somente pode ser diretamente usada dentro do módulo
- Um arquivo do tipo ".h" pode ser criado para indicar as coisas públicas (exportadas) pelo módulo

Monitores com a Linguagem C e Pthreads 2/14

- Uma propriedade fundamental dos monitores é permitir que apenas uma tarefa esteja ativa dentro do monitor a cada momento
- No caso das Pthreads,
  exclusão mútua é obtida com o emprego de **mutex**
- Basta criar um mutex geral para o monitor
- Colocar uma operação LOCK no início de cada função pública do monitor e colocar uma operação UNLOCK ao final de cada função pública
- Não é necessário usar mutex nas funções privadas do monitor, as quais são chamadas somente de dentro do próprio monitor
- No caso de uma função que termine com “return”, é necessário colocar o UNLOCK imediatamente antes do “return”
  - Usar no “return” apenas variáveis locais da função

Monitores com a Linguagem C e Pthreads 3/14

```
/* Monitor sensor, no arquivo sensor.c */

static pthread_mutex_t exclusao_mutua = PTHREAD_MUTEX_INITIALIZER;
static double sensor_lido = 0;

/* Função pública, pode ser chamada de fora do monitor */
void sensor_put( double lido)
{  pthread_mutex_lock( &exclusao_mutua);
   sensor_lido = lido;
   pthread_mutex_unlock( &exclusao_mutua);
}

/* Função pública, pode ser chamada de fora do monitor */
double sensor_get( void)
{  double aux;
   pthread_mutex_lock( &exclusao_mutua);
   aux = sensor_lido;
   pthread_mutex_unlock( &exclusao_mutua);
   return aux;
}
```

Monitores com a Linguagem C e Pthreads 4/14

- A biblioteca das Pthreads inclui recursos para a criação de **variáveis condição**
  - Implementa as operações WAIT e SIGNAL
  - Também implementa uma versão especial de SIGNAL, o BROADCAST
  - BROADCAST libera todas as threads bloqueadas na variável condição naquele instante
- Quando uma thread fica bloqueada ao executar a operação WAIT, ela precisa liberar o acesso ao monitor
  - Executar um unlock sobre o mutex que controla o acesso esclusivo ao monitor
- A implementação de WAIT nas Pthreads inclui um unlock implícito quando a thread fica bloqueada
  - A operação WAIT recebe dois parâmetros
  - A variável condição em questão
  - E também o mutex que deve ser liberado quando acontece o bloqueio no WAIT

Monitores com a Linguagem C e Pthreads 5/14

- Nas Pthreads, a thread que executa o SIGNAL continua a executar
  - A thread liberada da variável condição fica esperando pelo mutex
  - Existe um LOCK implícito no WAIT, quando a thread é liberada
- Quando a thread que executou o SIGNAL e continuou a executar finalmente liberar o monitor (UNLOCK na variável mutex)
  - A thread liberada da variável condição poderá obter implicitamente este mutex e retomar sua execução dentro do monitor
- Caso várias threads sejam liberadas por um BROADCAST
  - Uma a uma terá sucesso no LOCK do mutex e executará com exclusividade

Monitores com a Linguagem C e Pthreads 6/14

- Uma variável condição “vc” pode ser criada e receber uma inicialização default da seguinte forma:

```
pthread_cond_t vc = PTHREAD_COND_INITIALIZER;
```

- E os protótipos das funções “wait”, “signal” e “broadcast” são:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond).
```

Monitores com a Linguagem C e Pthreads 7/14

- EXEMPLO 1
- Existe um **sensor de temperatura** o qual é lido por uma thread
- O valor lido é posteriormente usado por várias outras threads
- A função sensor_put() é chamada pela thread que leu o sensor
  - Para informar o novo valor
- A função sensor_get() permite que qualquer thread obtenha o último valor lido do sensor
- Ao chamar a função sensor_alarme() a thread ficará bloqueada até que o valor lido do sensor ultrapasse o limite indicado como parâmetro
  - Caso o valor atual já seja superior ao limite indicado, a thread chamadora retornará imediatamente
  - Caso contrário, a thread executará um WAIT e ficará bloqueada

Monitores com a Linguagem C e Pthreads 8/14

```
/* Monitor sensor, no arquivo sensor.c */
#include <math.h>

static pthread_mutex_t exclusao_mutua =
    PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t alarme = PTHREAD_COND_INITIALIZER;
static double sensor_lido = 0;
static double limite_atual = HUGE_VAL;
```

Monitores com a Linguagem C e Pthreads 9/14

```
/* Chamado pela thread que le o sensor e disponibiliza aqui o valor lido */
void sensor_put( double lido)
{   pthread_mutex_lock( &exclusao_mutua);
    sensor_lido = lido;
    if( sensor_lido >= limite_atual )
            pthread_cond_signal( &alarme);
    pthread_mutex_unlock( &exclusao_mutua);
}

/* Chamado por qualquer thread que precisa do valor lido do sensor */
double sensor_get( void)
{   double aux;
    pthread_mutex_lock( &exclusao_mutua);
    aux = sensor_lido;
    pthread_mutex_unlock( &exclusao_mutua);
    return aux;
}
```

Monitores com a Linguagem C e Pthreads 10/14

```
/* Thread fica bloqueada até o valor do sensor chegar em limite */
void sensor_alarme( double limite)
{   pthread_mutex_lock( &exclusao_mutua);
    limite_atual = limite;
    while( sensor_lido < limite_atual )
            pthread_cond_wait( &alarme, &exclusao_mutua);
    limite_atual = HUGE_VAL;
    pthread_mutex_unlock( &exclusao_mutua);
}
```

Resumo

- Introdução
- Problemas Clássicos de Sincronização
- Semáforos
- Monitores
- Monitores com a Linguagem C e Pthreads

Mecanismos de Sincronização com Variáveis Compartilhadas

Fundamentos dos Sistemas de Tempo Real

2ª Edição

Rômulo Silva de Oliveira

Edição do Autor, 2020

www.romulosilvadeoliveira.eng.br/livrotemporeal

Mecanismos de Sincronização com Variáveis Compartilhadas

Parte V:
Exemplo de monitores:
Buffer duplo (*double buffering*)

Monitores com a Linguagem C e Pthreads 11/14

- EXEMPLO 2
- **Buffer duplo (*double buffering*)**
  - Buffer dividido em duas metades, digamos buffer_0 e buffer_1
- Inicialmente a thread escreve os dados no buffer_0
- Quando ele ficar cheio, a thread escritora passa a usar o buffer_1 para escrever
  - Enquanto outra thread esvazia de uma só vez o buffer_0
- É essencial que o buffer_0 seja esvaziado antes do buffer_1 lotar
  - Pois quando o buffer_1 lotar
  - A thread escritora passará novamente a usar o buffer_0
  - E encaminhará o buffer_1 para a thread que consome os dados
- O tamanho do buffer deve ser dimensionado de tal forma que um buffer possa ser sempre esvaziado mais rapidamente do que um buffer demora para ser preenchido

Monitores com a Linguagem C e Pthreads 12/14

```
/* Monitor buffer duplo, no arquivo bufduplo.c */
#define TAMBUF 100

static double buffer_0[TAMBUF];
static double buffer_1[TAMBUF];

static int emuso = 0;
static int prox_insercao = 0;
static int gravar = -1;

static pthread_mutex_t exclusao_mutua = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t buffer_cheio = PTHREAD_COND_INITIALIZER;
```

Monitores com a Linguagem C e Pthreads 13/14

```
void bufduplo_insereLeitura( double leitura)
{    pthread_mutex_lock( &exclusao_mutua);

     if( emuso == 0 )
          buffer_0[prox_insercao] = leitura;
     else
          buffer_1[prox_insercao] = leitura;

     ++prox_insercao;

     if( prox_insercao == TAMBUF ) {
          gravar = emuso;
          emuso = (emuso + 1) % 2;
          prox_insercao = 0;
          pthread_cond_signal( &buffer_cheio);
     }
     pthread_mutex_unlock( &exclusao_mutua);
}
```

Monitores com a Linguagem C e Pthreads 14/14

```
double *bufduplo_esperaBufferCheio( void)
{   double *buffer;
    pthread_mutex_lock( &exclusao_mutua);
    while( gravar == -1 )
        pthread_cond_wait( &buffer_cheio, &exclusao_mutua)
    if( gravar==0 )
        buffer = buffer_0;
    else buffer = buffer_1;
    gravar = -1;
    pthread_mutex_unlock( &exclusao_mutua);
    return buffer;
}
```

Resumo

- Introdução
- Problemas Clássicos de Sincronização
- Semáforos
- Monitores
- Monitores com a Linguagem C e Pthreads