
Mecanismos de Sincronização com Variáveis Compartilhadas (Semáforos e Monitores)



Fundamentos dos Sistemas de Tempo Real

Rômulo Silva de Oliveira

Edição do Autor, 2018

www.romulosilvadeoliveira.eng.br/livrotemporeal

Outubro/2018

- **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

- 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

- 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

$V(S)$:

$S.\text{valor} = S.\text{valor} + 1;$

Se $S.\text{fila}$ não está vazia

Então libera para execução primeira tarefa de $S.\text{fila}$

- 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

- 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

$V(S);$

- **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

- 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

- 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

- 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

- 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

- 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++

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 exclusivo 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

Monitor em C

Recursos

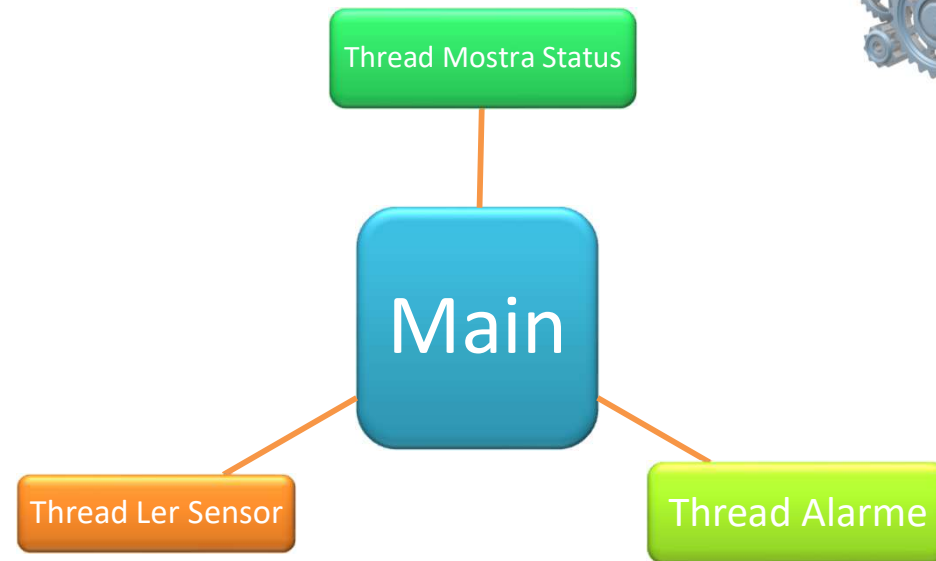


Tela

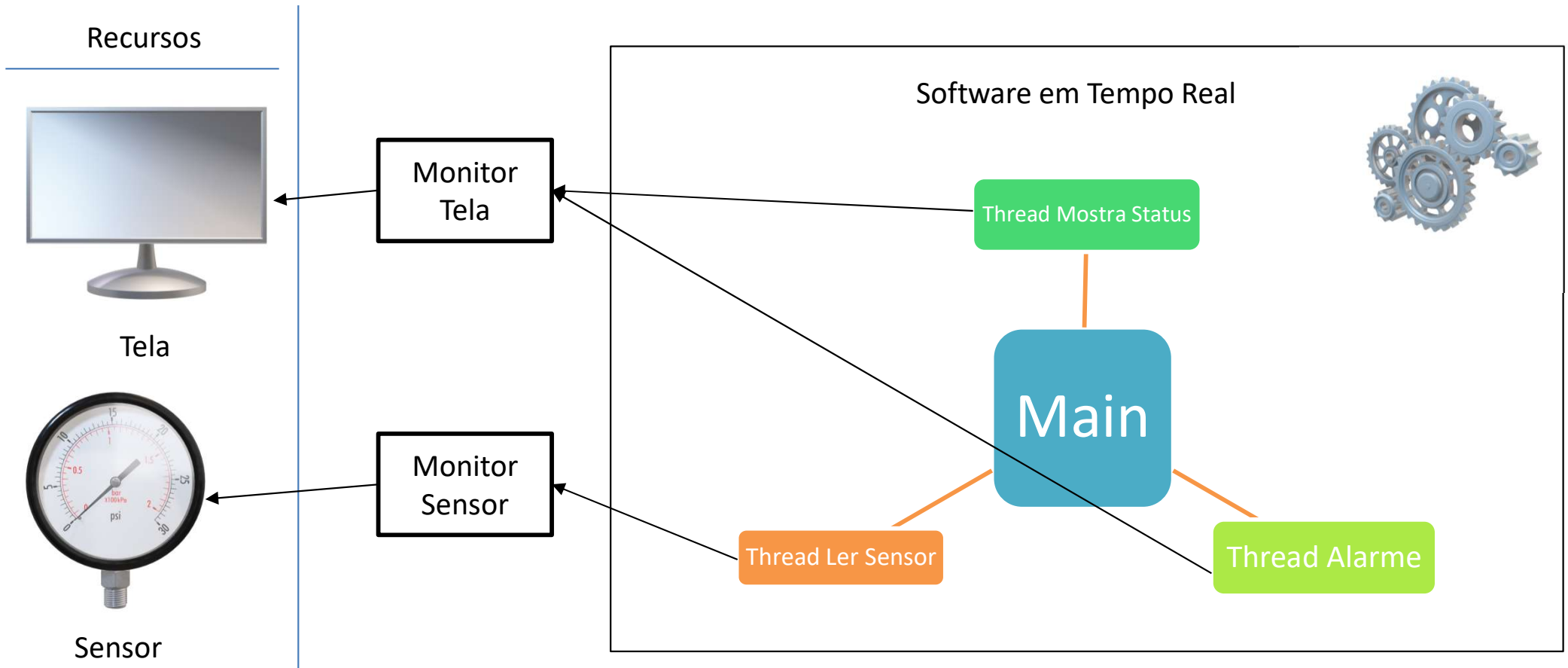


Sensor

Software em Tempo Real



Monitor em C



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);  
}
```


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_inserLeitura( 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;
}
```