Sincronização e Comunicação entre Tarefas

Fundamentos dos Sistemas de Tempo Real

Rômulo Silva de Oliveira

Edição do Autor, 2018

www.romulosilvadeoliveira.eng.br/livrotemporeal

Outubro/2018

Introdução 1/2

- **Programas sequenciais** (sequential programs)
  - São aqueles executados por uma tarefa sozinha
  - Utiliza serviços do sistema operacional através de chamadas de sistema
  - Mas é a única tarefa no programa

- **Programas concorrentes** (concurrent programs)
  - São aqueles executados simultaneamente por várias tarefas, as quais cooperam entre si
  - Cooperar significa trocar dados
    - uma tarefa produz um dado que é usado por outra tarefa
  - Ou sincronizar a execução
    - uma tarefa precisa esperar outra tarefa fazer algo para ela poder prosseguir)

Introdução 2/2

- Programação paralela refere-se a paralelismo físico na execução de instruções, como em um processador multicore
- Programação Distribuída refere-se a existência de vários computadores conectados através de uma rede de comunicação
- Programação concorrente pode ser implementada em processadores multicore e pode ser implementada também em uma rede de computadores
- Programação concorrente pode também ser implementada em um processador único, onde processos e threads são criados a partir do compartilhamento deste processador único

Métodos 1/7

- Métodos de Sincronização e Comunicação entre Tarefas

- A sincronização e comunicação entre tarefas é feita, na maioria dos programas concorrentes, através de dois métodos:

- **Troca de mensagens (*message passing*)**
- **Acesso a variáveis compartilhadas (*shared memory*)**

Métodos 2/7

- Quando a troca de mensagens é empregada,
- as tarefas enviam mensagens umas para as outras

- Dados podem ser trocados através do conteúdo das mensagens enviadas

- E o fato de enviar ou receber uma mensagem também permite que as tarefas sincronizem suas ações

Métodos 3/7

- Quando variáveis compartilhadas são empregadas
- Existem variáveis que podem ser acessadas por várias tarefas
- As variáveis compartilhadas permitem diretamente a troca de dados
- Para a sincronização é usual a disponibilização pelo sistema operacional (kernel ou microkernel) de mecanismos criados especialmente para isto

Padrões de Interação Usando Mensagens 1/4

- Um aspecto importante quando mensagens são usadas na construção de programas concorrentes é definir como será o **padrão de interação (*messaging pattern*)** entre as tarefas

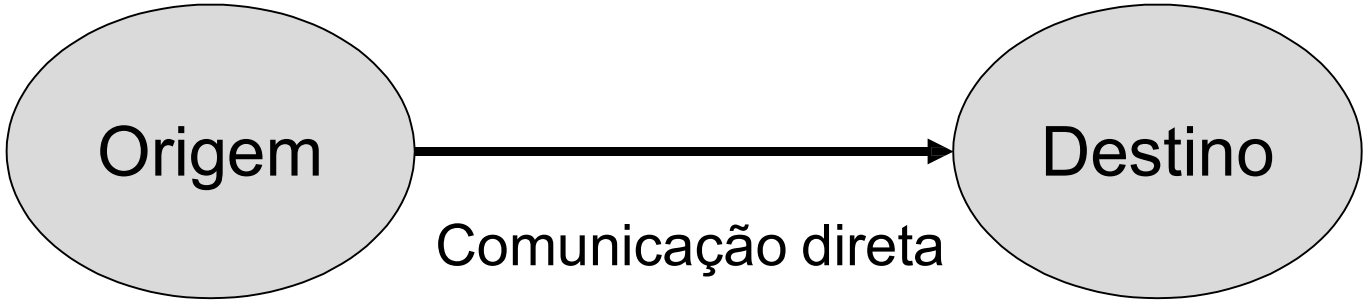
- A literatura descreve um grande número de padrões possíveis

- Depende do tipo de contexto e dos propósitos da aplicação

Padrões de Interação Usando Mensagens 2/4

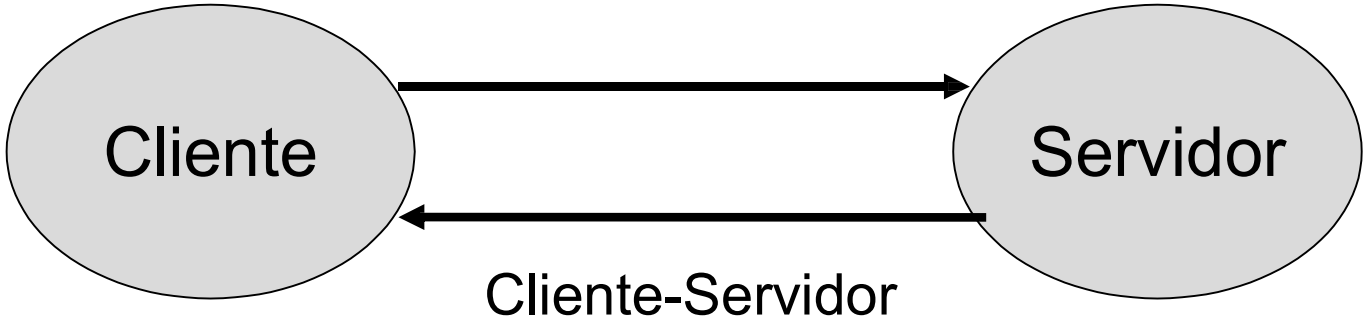
- Em aplicações pequenas muitas vezes uma simples **comunicação direta (*direct communication*)** entre duas tarefas é o bastante

- Por exemplo, uma tarefa periodicamente lê um sensor de temperatura e envia o valor lido através de uma mensagem para a tarefa responsável pela execução da estratégia de controle



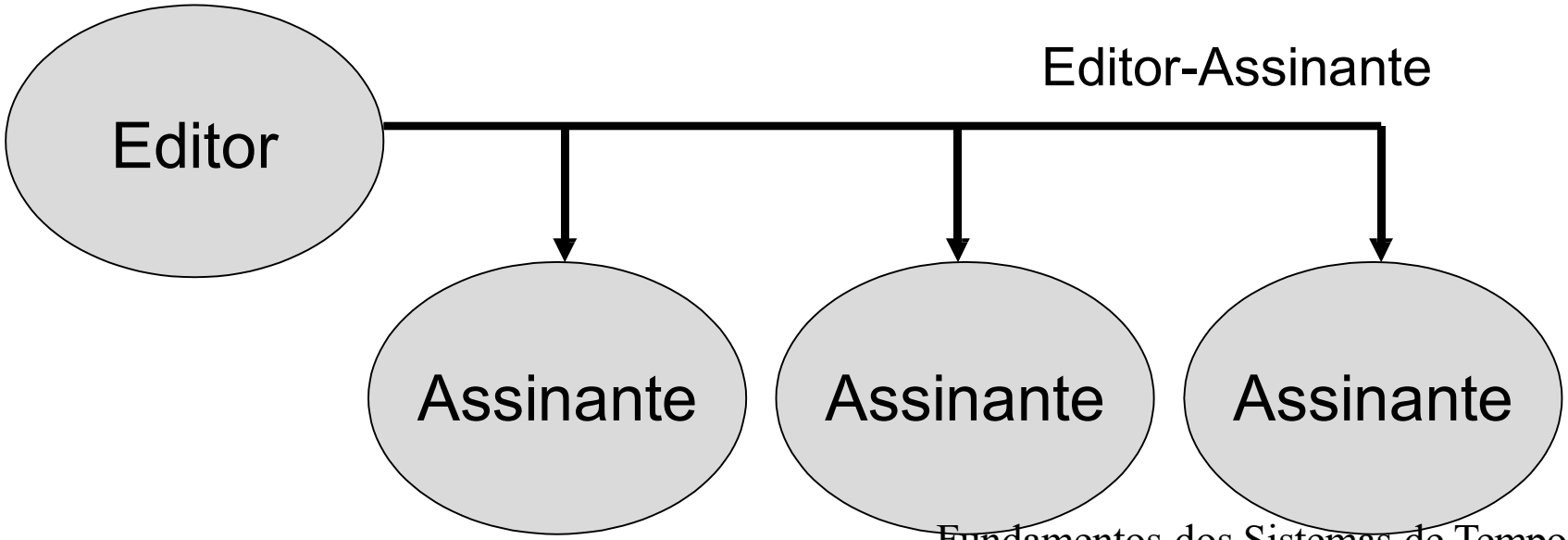
Padrões de Interação Usando Mensagens 3/4

- O padrão **cliente-servidor (*client-server*)** é o padrão dominante na Internet e é também muito usado em aplicações de tempo real
- A tarefa cliente sempre tem a iniciativa de enviar uma mensagem contendo uma requisição (request) para a tarefa servidora
- A tarefa servidora nunca toma a iniciativa, ela sempre espera receber uma requisição e então executa o que foi solicitado e envia uma mensagem de resposta (response) de volta para o remetente da requisição



Padrões de Interação Usando Mensagens 4/4

- O padrão **editor-assinante (*publisher-subscriber*)** é muito usado em sistemas de tempo real
- Permite que uma tarefa que possua uma informação relevante (a leitura de um sensor por exemplo) possa comunicar esta informação (editor) para todas as tarefas (assinantes) que possuam interesse nela
- Cada tarefa pode publicar uma ou mais informações e também ser assinante de outras informações



Comunicação com Variáveis Compartilhadas 1/6

- Primeiro requisito:
  Sistema operacional permite o acesso das tarefas a uma mesma memória física
- A forma como a memória compartilhada é implementada depende da gerência de memória empregada

- No caso mais simples, o sistema operacional consiste de um microkernel que não implementa proteção de memória
  - Todas as threads do sistema podem acessar toda a memória física
  - Toda a memória é automaticamente compartilhada entre todas as threads

Comunicação com Variáveis Compartilhadas 3/6

- Em programas concorrentes com variáveis compartilhadas as tarefas são normalmente implementadas como **threads** de um mesmo processo

- Todas as threads automaticamente compartilham as variáveis globais do programa
  - Náo é necessário nenhum esquema especial para isto

- Variáveis locais são tipicamente alocadas na pilha de cada thread e, por isto, são privativas de cada thread

- Uma vantagem adicional é a rapidez no chaveamento de contexto entre threads de um mesmo processo

Comunicação com Variáveis Compartilhadas 4/6

- O emprego de threads predomina na construção de programas concorrentes com variáveis compartilhadas

- A biblioteca mais usada para isto é a **Posix Threads** ou **Pthreads**
  - Parte do padrão POSIX de sistemas operacionais

- O Posix foi criado pela IEEE para padronizar sistemas operacionais estilo Unix nos anos 80 e tornou-se extremamente popular ao longo dos anos
- A biblioteca Pthread é rica em funções, com muitas variações e parametrizações possíveis

Comunicação com Variáveis Compartilhadas 6/6

```
#include <pthread.h>
pthread_t th1, th2;
void main(void)
{    ...
     pthread_create(&th1, NULL,(void *)codigo_th1, NULL);
     pthread_create(&th2, NULL,(void *)codigo_th2, NULL);
     ...
     pthread_join( &th1);
     pthread_join( &th2);
     ...
}
...
void codigo_th1( void)
{    ...
}
...
void codigo_th2( void)
{    ...
}
```

Sincronização com Variáveis Compartilhadas 6/6

- O código que acessa variável compartilhada é chamado de **seção crítica** (**critical section**)
- O **problema da seção crítica** (**critical section problem**):
  - Garantir que todas as threads possam executar suas seções críticas sem interferir com as seções críticas das outras threads

Seção Crítica: Mutex 2/6

- Provavelmente o mecanismo mais usado é o **mutex**
- Mutex é uma contração de “mutual exclusion” (exclusão mútua)

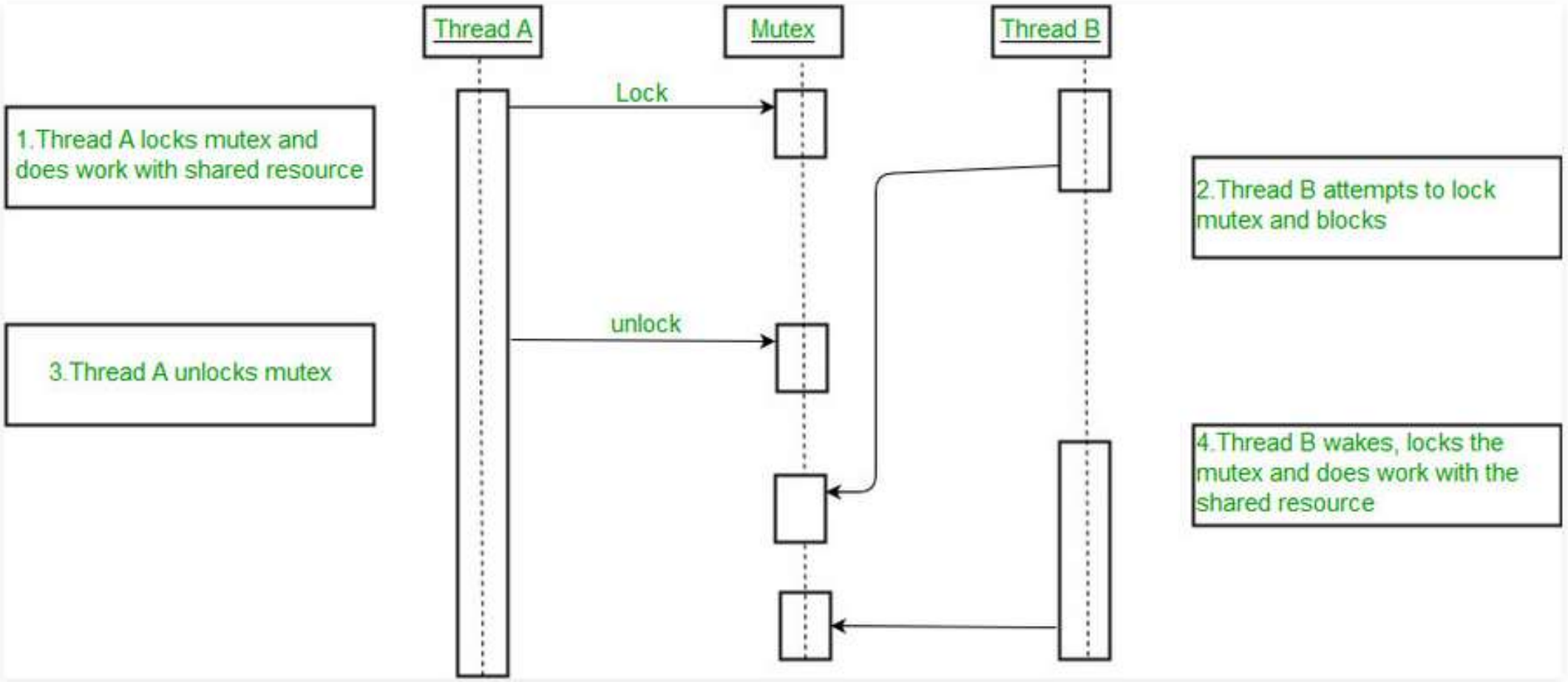
- O mutex é um tipo abstrato de dado
  - Possui um valor lógico
  - e uma fila de threads bloqueadas

- O valor lógico indica estado livre ou ocupado
- Fila de threads contém as threads bloqueadas esperando a liberação deste mutex

Seção Crítica: Mutex 3/6

- LOCK sobre um mutex livre:
  - Passa para ocupado e a thread segue sua execução
- LOCK sobre um mutex ocupado:
  - Thread fica bloqueada
  - Sai da fila do processador
  - Inserida na fila do mutex

- UNLOCK sobre um mutex livre:
  - Nada acontece
- UNLOCK sobre um mutex ocupado e a sua fila de threads está vazia:
  - Mutex passa para o estado livre
- UNLOCK sobre um mutex ocupado cuja fila de threads não está vazia:
  - Mutex permanece ocupado
  - Primeira thread bloqueada em sua fila é liberada
  - Novamente inserida na fila processador


Thread A
Mutex
Thread B
Lock
1.Thread A locks mutex and does work with shared resource
2.Thread B attempts to lock mutex and blocks
unlock
3.Thread A unlocks mutex
4.Thread B wakes, locks the mutex and does work with the shared resource

Seção Crítica: Mutex 4/6

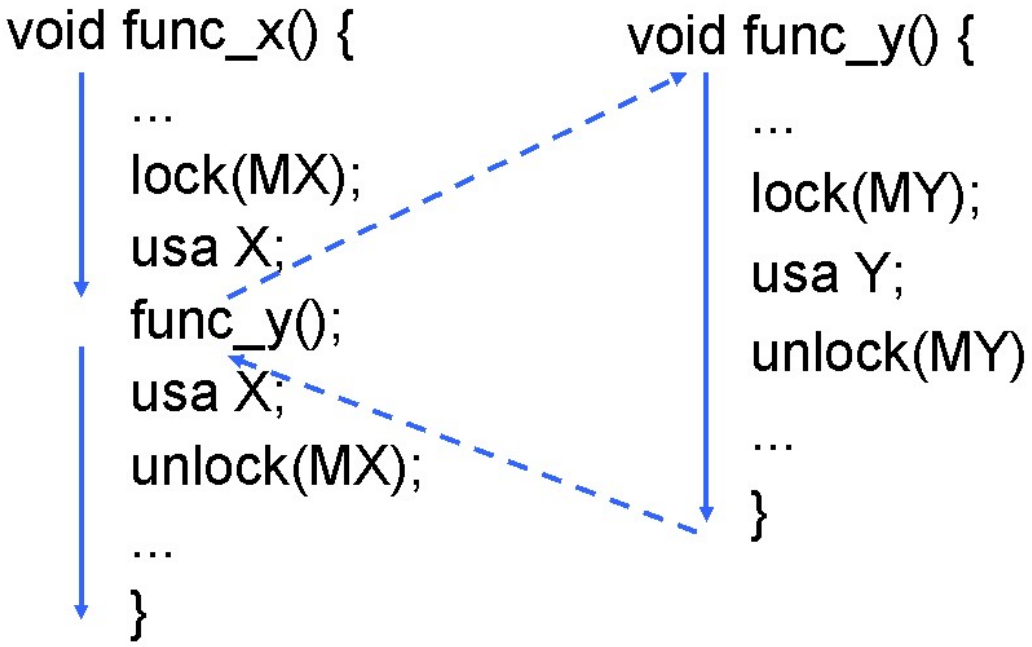
```
#include <pthread.h>
pthread_mutex_t meuMutex = PTHREAD_MUTEX_INITIALIZER;
...
int soma = 0;

void codigo_th1( void)
{    ...
     pthread_mutex_lock( &meuMutex );
     ++soma;
     pthread_mutex_unlock( &meuMutex );
     ...
}

void codigo_th2( void)
{    ...
     pthread_mutex_lock( &meuMutex);
     ++soma;
     pthread_mutex_unlock( &meuMutex);
     ...
}
```

Aninhamento de Mutex e Deadlock 1/4

- Seções críticas podem ser aninhadas
- **Aninhamento perfeito (*perfect nesting*)**
  - UNLOCK acontece exatamente na ordem reversa das operações de LOCK



Aninhamento de Mutex e Deadlock 2/4

- Quando é possível o aninhamento de mutex, surge também a possibilidade de ocorre **deadlock (*impasse*)**
- Um conjunto de N tarefas está em deadlock quando
  - Cada uma das N tarefas está bloqueada à espera de um evento
  - Que somente pode ser causado por uma das N tarefas do conjunto

- Exemplo:
  - Tarefa $\tau 2$ faz LOCK(Y)
  - Tarefa $\tau 1$ chega, preempta o processador, executa LOCK(X)
  - Tarefa $\tau 1$ faz LOCK(Y) e fica bloqueada
  - Tarefa $\tau 2$ volta a executar, faz LOCK(X) e fica bloqueada

Resumo

- Introdução
- Métodos
- Sincronização e Comunicação com Mensagens
- Padrões de Interação Usando Mensagens
- Comunicação com Variáveis Compartilhadas
- Sincronização com Variáveis Compartilhadas
- Seção Crítica: Mutex
- Aninhamento de Mutex e Deadlock