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

Motivação 1/3

- Uma razão para usar programação concorrente é <u>aproveitar processadores com múltiplos núcleos</u>
- Até o início dos anos 2000, a cada cerca de 2 anos, a frequência do clock dos processadores dobrava (lei de Moore)
- Desde então, processadores continuam ficando cada vez mais poderosos, aumentando o número de núcleos de processamento (cores) no mesmo chip

- Para um programa executar mais rapidamente, é necessário torna-lo concorrente (várias tarefas)

Motivação 2/3

- Uma outra razão para usar programação concorrente é
  poder usar entrada e saída síncrona em algumas situações específicas

- Suponha que um programa escrito na linguagem C precise ler comandos do teclado ao mesmo tempo que executa o algoritmo de controle de algum equipamento físico

- O algoritmo de controle não tolera esperas longas

- Programa sequencial não pode ficar bloqueado a espera que algo seja teclado, pois isto traria o risco do equipamento físico ficar descontrolado.
  - Um simples “scanf()” não pode ser usado.

- Programa concorrente pode ter duas tarefas
  - Uma tarefa faz o controle do equipamento físico
  - Outra tarefa fica bloqueada (entrada e saída síncrona) no “scanf()”

Motivação 3/3

- Razão mais importante é
  ser capaz de projetar mais facilmente programas com alto paralelismo intrínseco
- Suponha programa responsável por controlar uma caldeira, como as existentes em hotéis e hospitais
- Programa precisa executar várias funcionalidades simultaneamente
  - Laço de controle de nível
  - Laço de controle de temperatura
  - Apresentar na tela informações de status
  - Aceitar comandos de um operador humano via teclado
  - Manter em arquivo um registro das ocorrências anormais no sistema
  - Disparar alarmes em caso de falhas
  - Etc

- Um programa concorrente, onde cada tarefa cuida de uma das funcionalidades citadas antes, resulta em um design simples
  - Reproduz no código fonte de maneira elegante o que o programa deve fazer
  - É mais facilmente construído como programa concorrente

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 <u>troca de mensagens</u> 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

Métodos 4/7

- Interações entre tarefas podem ser programadas tanto com troca de mensagens como com variáveis compartilhadas
- O poder de expressão dos dois métodos é equivalente e a maioria dos sistemas operacionais oferece suporte para os dois
- Desta forma, a escolha entre um e outro leva em consideração outros aspectos

Métodos 5/7

- Em sistemas distribuídos, quando as tarefas executam em diferentes computadores, o emprego de mensagens é natural
- Porém, mesmo quando as tarefas executam no mesmo computador, mensagens podem ser uma boa escolha
- A necessidade das tarefas trocarem mensagens explícitas cria entre elas um protocolo que precisa ser seguido
- Desvios do protocolo, causados por exemplo por um erro de programação, podem ser mais facilmente detectados e a falha controlada
  - As mensagens criam um isolamento entre as tarefas que pode ser usado na detecção de falhas e na construção de sistemas mais robustos

Métodos 6/7

- Por outro lado, o acesso das tarefas às <u>variáveis compartilhadas</u> em geral é mais rápido, gerando menos custo de implementação

- O lado negativo é que, se uma tarefa corromper os dados compartilhados,

  todas as tarefas que precisam daqueles dados terão provavelmente sua execução corrompida

Métodos 7/7

- Atualmente um grande número de aplicações são distribuídas
  - Requerendo a cooperação de tarefas executando em diferentes computadores
- Desta forma, o uso de troca de mensagens é muito comum

- Ao mesmo tempo,
  - Dado o paralelismo intrínseco de muitas aplicações
  - Disponibilidade de computadores com múltiplos núcleos de processamento (multicore)
- Programação concorrente com variáveis compartilhadas também é muito empregada

- Em resumo,
  os dois métodos são igualmente importantes e necessários

Sincronização e Comunicação com Mensagens 1/7

- Requer que o sistema operacional ou alguma biblioteca ofereça suporte à troca de mensagens
  - Existem diversas opções neste sentido
- Em essência, toda infraestrutura para troca de mensagens precisa oferecer para as tarefas uma forma de enviar mensagens e receber mensagens
- Primitiva chamada SEND que permite enviar mensagens
- Primitiva chamada RECEIVE que permite receber mensagens
- Primitivas podem ser oferecidas pelo sistema operacional como chamadas de sistema
- Podem também ser implementadas como funções de uma biblioteca de comunicação que executa fora do kernel ou microkernel

Sincronização e Comunicação com Mensagens 2/7

- A **primitiva SEND** possui dois parâmetros principais:
  - A mensagem a ser enviada
  - O destinatário da mensagem

- No caso do destinatário, é possível usar endereçamento direto ou indireto

- No endereçamento direto, a mensagem é endereçada a uma tarefa específica
  - Por exemplo, o PID (Process Id) do processo que a implementa no computador destino

- No endereçamento indireto é usado o recurso da caixa-postal (mail-box)
  - Mensagens são enviadas pela tarefa remetente para uma caixa-postal
  - As mensagens serão lidas no futuro por alguma tarefa destinatária cuja identificação específica não é conhecida pela tarefa remetente

Sincronização e Comunicação com Mensagens 3/7

- No SEND o destinatário é normalmente único
  - Uma tarefa ou uma caixa postal apenas
- Alguns sistemas permitem a criação de grupos de destinatários
- O envio de uma mensagem para um grupo pode ter semânticas variadas
- Enviar para um grupo pode significar
  - enviar para todos do grupo
  - enviar para pelo menos um do grupo
  - enviar para no máximo um do grupo
  - enviar para todos ou para nenhum
  - entre outras semânticas possíveis

Sincronização e Comunicação com Mensagens 4/7

- Mensagens podem ter tamanho fixo
  - Isto é mais comum em redes industriais conhecidas como fieldbus
- Na Internet ou entre tarefas no mesmo computador mensagens podem ter tamanho variável
- O conteúdo das mensagens pode ou não ser transformado pelo sistema de troca de mensagens
  - Por exemplo, suponha que no computador remetente caracteres são representados por 1 byte seguindo o código ASCII, enquanto no computador destinatário caracteres são representados por 2 bytes seguindo o código Unicode

Sincronização e Comunicação com Mensagens 5/7

- Os principais valores retornados pela **primitiva RECEIVE** são o conteúdo da mensagem recebida e o endereço do remetente desta mensagem

- Tipicamente a primitiva RECEIVE é usada para receber mensagens de qualquer remetente
- Porém, em muitos sistemas é possível definir um remetente específico e receber mensagem apenas dele

- Também pode ser possível atribuir níveis de prioridade às mensagens
  - Vai definir a ordem de entrega das mensagens
  - O mais comum é entregar as mensagens na ordem de chegada

Sincronização e Comunicação com Mensagens 6/7

- O que acontece quando a primitiva RECEIVE é usada porém nenhuma mensagem ainda foi enviada para aquele destinatário ?

- O RECEIVE é dito síncrono ou bloqueante quando a tarefa que executa o RECEIVE fica bloqueada até que uma mensagem chegue

- O RECEIVE é dito assíncrono quando ocorre retorno imediato, não importando se mensagem foi recebida ou não
  - Caso uma mensagem tenha sido recebida, ela é entregue
  - Caso nenhuma mensagem esteja disponível, o RECEIVE retorna um código

- Abordagem intermediária é bloquear a tarefa até que ou uma mensagem chegue ou passe um certo intervalo de tempo (time-out) sem chegar nenhuma mensagem

Sincronização e Comunicação com Mensagens 7/7

- O que acontece quando um SEND é executado porém o destinatário ainda não executou a respectiva primitiva RECEIVE ?

- Na grande maioria dos sistemas baseados em troca de mensagens, a própria implementação do serviço de mensagens fica responsável por armazenar temporariamente (buffering) as mensagens enviadas porém ainda não entregues

- Elas serão entregues no futuro quando o respectivo RECEIVE for executado
- Obviamente existem limitações
  - Capacidade de bufferização
  - Mecanismo de descarte de mensagens depois de um longo período de espera

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

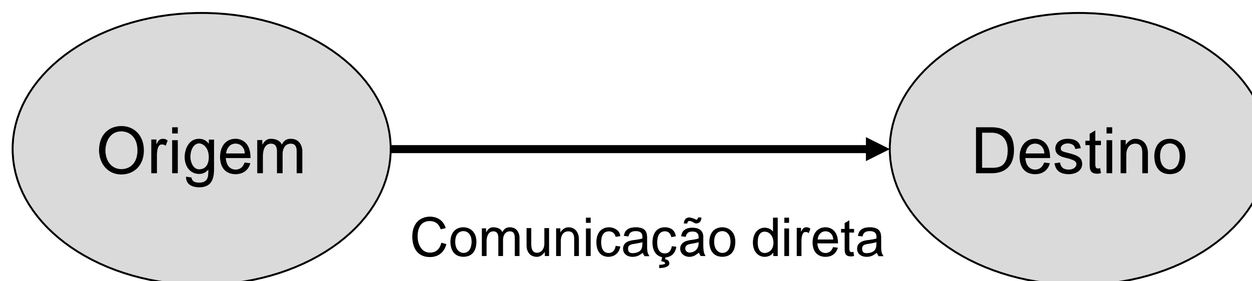
- A literatura descreve um grande número de padrões possíveis

- Depende do tipo de contexto e dos propósitos da aplicação

Padrões de Interação Usando Mensagens 2/4

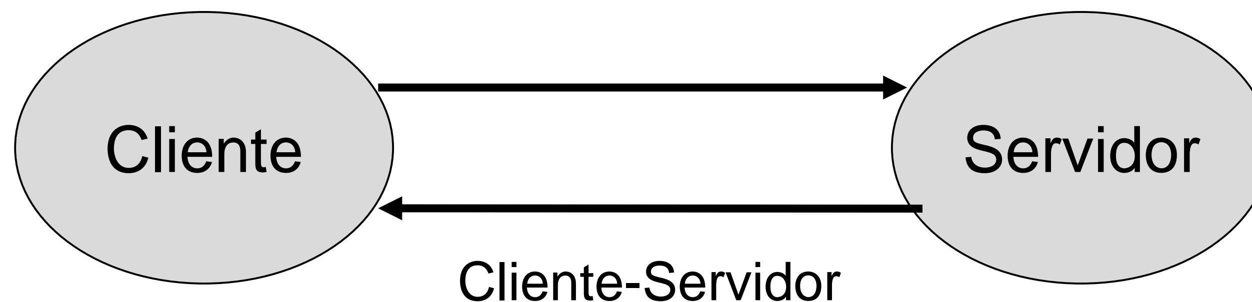
- Em aplicações pequenas muitas vezes uma simples **comunicação direta (*direct communication*)** entre duas tarefas é o bastante

- 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



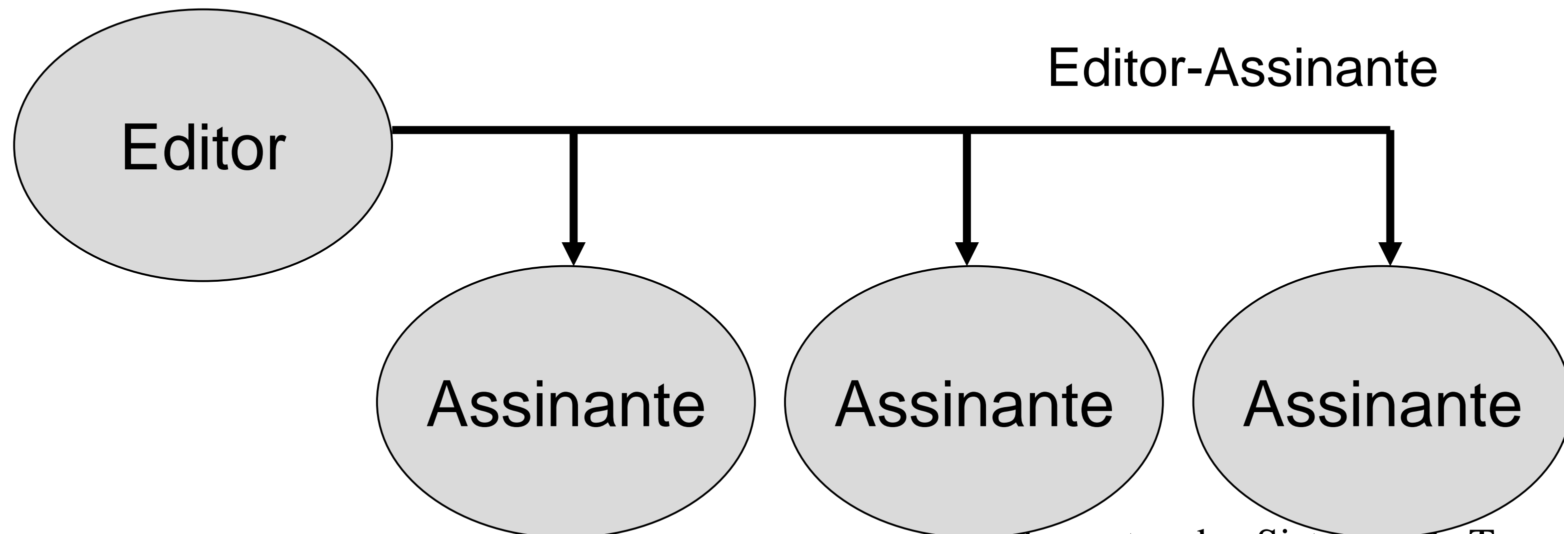
Padrões de Interação Usando Mensagens 3/4

- 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 2/6

- Tarefas podem ser implementadas como processos em sistemas operacionais com proteção de memória entre processos
- Nestes sistemas é possível utilizar chamadas de sistema específicas para criar regiões de memória compartilhadas entre os processos
- Por exemplo, em sistemas que seguem o padrão Posix isto é principalmente feito com as chamadas
  - "shm_open()", a qual cria uma região de memória compartilhada
  - "mmap()", usada para mapear uma região de memória compartilhada na memória acessível pelo processo

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 5/6

- Deve-se incluir as declarações da biblioteca que estão no arquivo "pthread.h"
- O programa inicia normalmente na função "main()" com apenas uma thread
- Cada thread adicional precisa de um identificador do tipo "pthread_t"
- As threads adicionais são criadas através da função "pthread_create()"
- Quatro parâmetros:
  - variável que vai servir como identificador da thread
  - permite alterar propriedades tais como tamanho da pilha e a prioridade da thread
  - endereço da função que será executada pela thread criada
  - permite passar informações para a thread recém criada
- A função "pthread_join()" bloqueia a thread chamadora até que a thread indicada como parâmetro termine

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 1/6

- Suponha que uma lista encadeada de medições de sensor seja mantida como uma estrutura de dados global, acessada por várias threads do programa
- A implementação clássica é manter um apontador para o início da lista e outro para o fim da lista
- No caso da inserção no fim da lista:
  - Caso a lista esteja vazia, tanto "inicio" como "fim" passam a apontar a nova e única medição da lista
  - Caso a lista não esteja vazia, o campo proximo da atual última medição passa a apontar para a nova medição
  - A variável "fim" também passa a apontar a nova medição recém inserida

Sincronização com Variáveis Compartilhadas 2/6

- No caso da retirada, a variável “inicio” passa a apontar para a segunda medição da lista, pois a primeira está sendo retirada
- Caso a lista tenha apenas uma medição, ela ficará agora vazia
  - A variável “inicio” conterá NULL
  - A variável “fim” também recebe NULL

Sincronização com Variáveis Compartilhadas 3/6

- Em um programa concorrente com várias threads, é possível que a thread th1 chame a função “insere()” ao mesmo tempo que a thread th2 chama a função “retira()”

- No caso de um único processador:
  Uma delas pode estar no meio da função chamada quando ocorre uma interrupção e a outra thread passa a executar

- No caso de multiprocessamento:
  As duas threads vão executar as funções em paralelismo real

- Quando várias threads acessam e alteram os mesmos dados concorrentemente, e o resultado da execução será correto ou errado dependendo da ordem de execução temos o que é chamado de **condição de corrida** (**race condition**)

Sincronização com Variáveis Compartilhadas 4/6

- Condições de corrida podem ocorrer mesmo com dados elementares
- Suponha uma variável global do tipo inteiro
  - que é incrementada pelas threads th1 e th2
  - Na linguagem C o incremento aparece como “++xyz”
  - Implementado tipicamente como:

```
MOVE   XYZ,ACC     ; Move o valor de XYZ para o acumulador
INC    ACC         ; Incrementa o conteúdo do acumulador
MOVE   ACC,XYZ     ; Coloca o valor incrementado em XYZ
```

- Suponha que o valor inicial de XYZ é zero
  - A variável é incrementada uma vez por th1 e uma vez por th2
  - Uma variável que inicia com 0 (zero) e é incrementada duas vezes deveria terminar com o valor 2 (dois)

Sincronização com Variáveis Compartilhadas 5/6

- A sequência de instruções abaixo mostra que este resultado não é garantido sempre:

- Thread th1 move XYZ para o seu acumulador, logo acumulador de th1 recebe 0
- Thread th1 incrementa o seu acumulador, logo acumulador de th1 recebe 1
- Thread th2 move XYZ para o seu acumulador, logo acumulador de th2 recebe 0
- Thread th2 incrementa o seu acumulador, logo acumulador de th2 recebe 1
- Thread th2 move o seu acumulador para XYZ, logo XYZ recebe 1
- Thread th1 move o seu acumulador para XYZ, logo XYZ recebe 1

- O erro somente acontece quando as instruções de máquina relativas aos dois incrementos são misturadas

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
  - A solução precisa apresentar:

- Exclusão mútua (mutual exclusion): Quando uma thread está executando sua seção crítica nenhuma outra thread pode executar sua respectiva seção crítica com respeito àquela mesma variável compartilhada

- Progresso (progress): Quando uma thread deseja executar uma seção crítica e nenhuma outra thread está em sua seção crítica, a thread que deseja executar deve poder executar

- Espera limitada (bounded waiting): Nenhuma thread deve ficar para sempre esperando para executar uma seção crítica (ausência de postergação indefinida)

- Independência do escalonador (scheduling independence): A solução não pode depender de algoritmos de escalonamento específicos

Seção Crítica: Mecanismos de Baixo Nível 1/7

- Mecanismo mais básico para atacar o problema da seção crítica é **desabilitar interrupções (*disable interrupts*)**
- Antes de acessar uma variável compartilhada a thread desabilita interrupções
- Chaveamento de contexto sempre acontece em resposta a uma interrupção
  - Interrupções estão desabilitadas
  - A thread em questão poderá executar a seção crítica de forma atômica (indivisível)
  - Volta a habilitar as interrupções na saída da seção crítica
- A execução atômica (atomic execution) da seção crítica garante a propriedade de exclusão mútua

Seção Crítica: Mecanismos de Baixo Nível 2/7

- Não funciona em processadores com múltiplos núcleos (multicore)
  - As várias threads estão executando em paralelismo físico, cada uma no seu núcleo, e podem todas entrar na seção crítica simultaneamente

- É uma instrução privilegiada
  - Proibida para código de usuário em sistemas operacionais que implementam proteção
  - Possível desabilitar interrupções em sistemas operacionais sem proteção entre processos ou como parte do código do kernel

- Reduz a responsividade do sistema como um todo
  - Sistema computacional percebe eventos através das interrupções
  - Uma tecla foi teclada, chegou um pacote de dados pela rede
- As interrupções ficarão pendentes e serão atendidas após a seção crítica
  - Não existe grande prejuízo se a seção crítica for rápida
  - Porém, uma seção crítica longa reduz a qualidade temporal do sistema como um todo

Seção Crítica: Mecanismos de Baixo Nível 3/7

- Outro mecanismo de baixo nível é chamado **spin-lock**
- Esta solução é baseada em uma instrução de máquina chamada muitas vezes de "Test-and-Set"
  - Seu nome pode variar de processador para processador
- Ela faz a **troca atômica (*atomic swap*)** entre dois operandos
  - Dois registradores do processador ou um registrador e uma posição de memória
- A "<u>forma atômica</u>" aqui significa que, se uma interrupção for sinalizada, o tratador da interrupção vai executar
  - antes do XCHG (quando os valores originais ainda estão no lugar)
  - ou depois do XCHG (quando a troca de valores já estiver completa)

Seção Crítica: Mecanismos de Baixo Nível 4/7

- A atomicidade das instruções do tipo “Test-and-Set” também é mantida no caso de vários processadores ou vários núcleos no mesmo processador executarem em paralelismo real

- Caso duas threads em núcleos distintos tentem executar XCHG
  - sobre os mesmos operandos
  - exatamente ao mesmo tempo
  - o hardware vai completar um XCHG antes do outro, ou vice-versa

Seção Crítica: Mecanismos de Baixo Nível 5/7

- O **spin-lock** é construído em torno de uma variável inteira compartilhada chamada usualmente de “lock”
- Ela indica se a seção crítica que ela protege está
  - livre (lock == 0)
  - ou ocupada (lock == 1)

- Usando o exemplo da instrução XCHG, quando uma thread deseja entrar na seção crítica, ela coloca o valor 1 no registrador EAX e em seguida executa:

```
XCHG        EAX, [lock]
```

- Caso a seção crítica estivesse livre antes, “lock” teria o valor 0, mas agora terá 1
- Caso a seção crítica estivesse ocupada antes, “lock” teria o valor 1
  - Continuará com 1
- O valor obtido de lock indica se a thread adquiriu ou não a seção crítica
  - Thread permanece em loop até conseguir
  - Coloca zero em “lock” para liberar a seção crítica

Seção Crítica: Mecanismos de Baixo Nível 6/7

- Quando a seção crítica é liberada acontece uma espécie de "sorteio" entre as threads no spin-lock para determinar qual thread acessará a seção crítica em seguida
  - Uma thread "azarada" poderia ficar em postergação indefinida
  - Somente é uma preocupação quando as seções críticas são muito disputadas
  - Existem soluções mais elaboradas para o spin-lock onde é criada uma fila de threads e as mesmas são atendidas pela ordem de chegada

- A outra limitação do spin-lock é chamada de espera ocupada (busy-waiting)
  - Enquanto espera para entrar na seção crítica, a thread ocupa o processador para executar o laço onde repete constantemente XCHG
  - Spin-lock é usado normalmente em multiprocessadores
  - Enquanto uma thread acessa a seção crítica em um processador (ou núcleo), a thread que espera no spin-lock executa em outro processador, gastando ciclos em busy-waiting
  - Situação tolerável enquanto as seções críticas forem rápidas

Seção Crítica: Mecanismos de Baixo Nível 7/7

- Embora muito usados na implementação do kernel do sistema operacional, estes dois mecanismos de baixo nível são inconvenientes para a programação de aplicações

- Eles obrigam o programador da aplicação a considerar aspectos da arquitetura do computador
  - os quais deveriam ser completamente escondidos pelo compilador e pelo sistema operacional

- A programação das aplicações pode utilizar mecanismos de mais alto nível, mais convenientes
  - Como o “Mutex”

Seção Crítica: Mutex 1/6

- Problema da seção crítica em aplicações:
  Mecanismos implementados pelo sistema operacional
  - Acessados através de chamadas de sistema

- Implementam abstrações de mais alto nível
- Mitigam as limitações associadas com os mecanismos de baixo nível
  - Desabilitar interrupções
  - Spin-lock

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

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);
      ...
}
```

Seção Crítica: Mutex 5/6

- Primitivas LOCK e UNLOCK precisam ser atômicas
- No caso da aplicação, LOCK e UNLOCK são chamadas de sistema
  - Ou funções de biblioteca que mapeiam para chamadas de sistema
- No código da aplicação elas são usadas supondo-se sua atomicidade
- É responsabilidade do kernel do sistema operacional prover esta atomicidade

- No caso de computadores com um único processador: interrupções são desabilitadas durante a implementação de LOCK e UNLOCK
  - Dentro do kernel desabilitar interrupções é possível
  - Interrupções são desabilitadas apenas durante a execução das primitivas
  - Seção crítica da aplicação executará com interrupções habilitadas

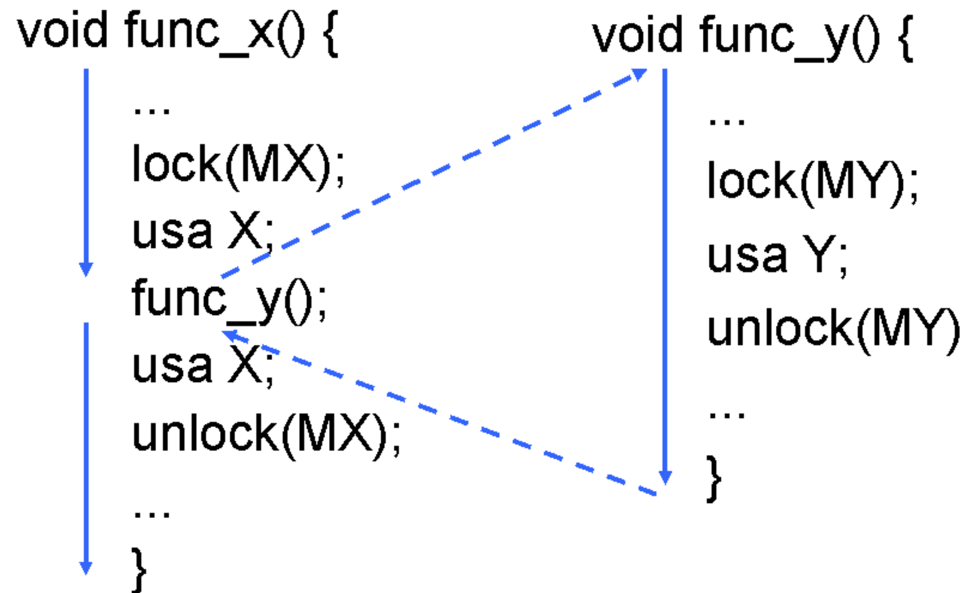
Seção Crítica: Mutex 6/6

- Computadores com vários processadores:
  Além de desabilitar interrupções, é necessário empregar <u>spin-lock</u>
  - Para evitar que threads em diferentes processadores manipulem o mesmo mutex ao mesmo tempo

- Spin-lock é usado para proteger o pequeno código das primitivas LOCK e UNLOCK
  - E não para proteger o código da seção crítica da aplicação
  - Questões como o busy-waiting e a postergação indefinida são minimizadas
  - Acontece apenas durante a execução das primitivas LOCK e UNLOCK

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

Aninhamento de Mutex e Deadlock 3/4

- Para ser possível deadlock são necessárias quatro condições:

  – Existência de recursos que precisam ser acessados de forma exclusiva, o que é inevitável na maioria dos sistemas
  – Possibilidade das tarefas manterem recursos alocados enquanto esperam por recursos adicionais, o que acontece com aninhamento de mutex
  – Necessidade dos recursos serem liberados pelas próprias tarefas que os estão utilizando, o que em geral é necessário para evitar que estruturas de dados sejam corrompidas
  – Possibilidade da formação de uma espera circular onde cada tarefa espera por um recurso que foi obtido por outra tarefa que espera outro recurso e assim por diante, fechando o círculo com a tarefa inicial

Aninhamento de Mutex e Deadlock 4/4

- No caso da espera circular existe uma solução simples
- Ordenar os mutex e exigir que as tarefas somente executem a operação LOCK na ordem crescente deles

- Supondo que os mutex X e Y estejam em ordem alfabetica
  - Fazer LOCK(X) e LOCK(Y) é válido
  - Porém fazer LOCK(Y) e LOCK(X) é proibido
  - Exemplo anterior de deadlock não é mais possível

- Com esta regra de alocação, é impossível deadlock

Resumo

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