

Pthreads

Mario João Junior

O que são threads?

- ◆ Uma thread é um fluxo de execução de instruções que pode ser escalonado pelo sistema operacional.

O que são threads?

◆ No UNIX, uma thread:

- Existe no interior de um processo e se utiliza dos recursos alocados para ele
- Compartilha os recursos do processo ao qual pertence com outras threads do mesmo
- Morre quando o processo a que pertence morre

O que são threads?

◆ Do ponto de vista do programador:

- Uma thread é uma função que é executada de forma independente do seu programa principal

O que são threads?

◆ Cada processo possui atributos que são compartilhados por suas threads. São eles:

- PID, PPID, GID e UID
- Variáveis de ambiente
- Diretório corrente
- Código fonte
- Alguns registradores especiais
- Pilha
- Área de dados (Globais + Heap)
- Descritores de arquivos

O que são threads?

◆ Cada processo possui atributos que são compartilhados por suas threads. São eles:

- Rotinas para tratamento de sinais
- Bibliotecas compartilhadas
- Ferramentas para comunicação entre processos (como filas de mensagens, pipes, semáforos e memória compartilhada).

O que são threads?

◆ Cada thread possui seu(s) próprio(s):

- SP (Cadeia Dinâmica)
- Registradores (inclusive PC, PSW...)
- Prioridade de escalonamento
- Sinais pendentes
- Dados específicos da thread

O que são threads?

◆ Todas as threads de um mesmo processo executam no mesmo espaço de endereçamento

◆ Num processo multi-threaded, existe mais de um ponto de execução simultaneamente

O que são threads?

- ◆ Algumas situações propiciadas pelo compartilhamento:
 - Uma thread pode fechar um arquivo que outra está utilizando
 - Dois ponteiros de threads diferentes podem apontar para o mesmo dado no Heap
 - Dados globais são compartilhados e os acessos a eles devem ser sincronizados

O que é Pthreads?

- ◆ É uma interface de manipulação de threads padronizada em 1995 pelo IEEE (IEEE POSIX 1003.1c)
- ◆ POSIX threads \Rightarrow Pthreads
- ◆ Pthreads foi definido como um conjunto de tipos e procedimentos em C, definidos em `pthread.h`

Threads x Processos

- ◆ A criação e a sincronização das threads são mais rápidas
- ◆ A comunicação entre threads é mais eficiente, por causa do espaço de endereçamento compartilhado
- ◆ Melhor eficiência em arquiteturas SMP

Threads de Kernel x Threads de Usuário

- ◆ Threads de usuário ($N - 1$)
 - O Kernel não tem conhecimento das threads
 - A criação, o escalonamento, e o gerenciamento das threads são controlados por uma biblioteca de funções
 - Muito leves para criar
 - Pthreads é uma biblioteca que implementa esse tipo de threads

Threads de Kernel x Threads de Usuário

◆ Threads de Kernel (1 – 1)

- O Kernel tem conhecimento das threads
- A criação, o escalonamento, e o gerenciamento das threads são controlados pelo Kernel
- Utilizam multi-processadores de forma mais eficiente
- Chamadas ao sistema não bloqueiam outras threads

Threads de Kernel x Threads de Usuário

◆ Modelo Híbrido (M – N)

- Utiliza M threads de usuário mapeadas em N threads de kernel.
- Possuem implementação mais complexa

Cuidado com threads

- ◆ Nem todas as funções das bibliotecas foram projetadas para trabalhar com threads (*thread-safe*)
- ◆ Na dúvida, assuma que a função NÃO é *thread-safe*

Modelos Mais Comuns de Programação com Threads

- ◆ Mestre-escravo: uma thread atribui as tarefas de todas as outras, podendo ou não participar da computação
- ◆ *Pipeline*: cada estágio do *pipeline* é atribuído a uma thread

Pthreads API

- ◆ Contem mais de 60 funções
- ◆ Incluir sempre `pthread.h`
- ◆ O padrão é definido apenas para a linguagem C

Pthreads API

- ◆ É dividida em 3 grandes categorias:
 - Gerenciamento de threads:
 - ◆ Criação, configuração, escalonamento...
 - Mutexes:
 - ◆ Exclusão mútua
 - Variáveis condicionais
 - ◆ Comunicação entre threads que compartilham mutexes

Criação de Threads

- ◆ Um processo começa sua execução com apenas uma thread, a thread mãe
- ◆ O número máximo de threads que um processo suporta é dependente de implementação

Criação de Threads

- ◆ Para criar threads:
 - `int pthread_create (pthread_t * thread, pthread_attr_t *attr, void * (*start_routine)(void *), void * arg);`
 - Valor de retorno: 0, se funcionar, ou um valor indicando erro, caso contrário

Criação de Threads

- ◆ **thread** é passado por referência e retorna o thread ID da nova thread
- ◆ **attr** são atributos para a criação da nova thread. Para os atributos default, basta passar NULL

Criação de Threads

- ◆ **start_routine** é a função em C onde será iniciada a nova thread. Seu protótipo segue o formato:
 - `void *start_routine (void *arg);`
- ◆ Apenas o parâmetro **arg** é passado para a nova thread. Caso seja necessário passar mais de um parâmetro, criar um novo tipo e agrupar os parâmetros necessários

Término de Threads

◆ Uma thread termina quando:

- A thread retorna da função que a originou (strat_routine)
- A thread chama **pthread_exit**
- A thread é cancelada por outra thread através da função **pthread_cancel**
- O processo inteiro termina

Término de Threads

◆ Para terminar a thread corrente:

- `void pthread_exit(void *retval);`

◆ **retval** é o valor de retorno. Pode ser utilizado por **pthread_join**.

Término de Threads

- ◆ Se a thread mãe termina retornando da sua função principal, suas filhas morrem
- ◆ Se a thread mãe termina com `pthread_exit`, suas filhas NÃO morrem

Término de Threads

- ◆ Exercício:
 - Criar 5 threads que escrevam seus thread IDs

Término de Threads

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void *ola (void *tid)
{
    printf ("\nSou a thread %d\n", tid);
    pthread_exit (NULL);
}
int main (int argc, char *argv[])
{
    pthread_t      thread;
    int            err,
                  t;

    for (t = 0; t < NUM_THREADS; t++)
    {
        printf ("Criando thread %d\n", t);
        if (err = pthread_create (&thread, NULL, ola, (void *)t))
        {
            printf ("Erro numero %d\n", err);
            exit (-1);
        }
    }
    pthread_exit (NULL);
}
```

Joining Threads

- ◆ "Joining" é uma maneira de se obter sincronização entre threads.
- ◆ Para que uma thread fique bloqueada até que uma outra termine:
 - `int pthread_join (pthread_t thread, void **status);`
- ◆ **thread** indica a thread a ser aguardada
- ◆ **status** é o mesmo determinado por **pthread_exit**

Identificação de Threads

- ◆ Para saber o thread ID da thread corrente:
 - `pthread_t pthread_self (void);`
- ◆ Para comparar dois thread IDs:
 - `int pthread_equal (pthread_t t1, pthread_t t2);`
 - Retorna 0 se forem diferentes, ou diferente de 0 caso contrário

Detached Threads

- ◆ Uma thread detached é uma thread que não pode ser sincronizada com **pthread_join**
- ◆ “Ser detached” é um atributo da thread, que pode ser utilizado no momento da criação da thread

Detached Threads

- ◆ Para criar uma thread detached:
 - Declarar uma variável do tipo `pthread_attr_t`
 - Inicializar essa variável através da função **`pthread_attr_init`**
 - Colocar o atributo detached nessa variável através da função **`pthread_attr_setdetachstate`**
 - Criar as threads com `pthread_create` usando essa variável como parâmetro

Detached Threads

- ◆ Para inicializar uma variável do tipo `pthread_attr_t`:
 - `int pthread_attr_init (pthread_attr_t *attr)`

Detached Threads

- ◆ Para colocar o atributo detached numa variável do tipo `pthread_attr_t`:
 - `int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate)`
- ◆ O parâmetro **detachstate** pode ser:
 - **PTHREAD_CREATE_DETACHED** ⇒ criada como detached
 - **PTHREAD_CREATE_JOINABLE** ⇒ pode ser sincronizada com **join** (default)

Detached Threads

- ◆ Para fazer com que uma thread fique detached:
 - `int pthread_detach (pthread_t t)`

Detached Threads

◆ Exercício:

- Criar 4 threads que achem multipliquem todos os elementos de um vetor por um escalar e depois achem o maior elemento
- Primeiro fazem a multiplicação
- Depois acham o maior

Mutex

- ◆ Mutex é uma abreviação de "mutual exclusion" (exclusão mútua)
- ◆ Variáveis do tipo Mutex são a principal forma que o Pthreads apresenta para a proteção de regiões críticas
- ◆ Uma variável do tipo mutex é um "lock" que protege dados compartilhados pelas threads.

Mutex

- ◆ O princípio básico é que apenas uma thread pode ter efetuado um lock em uma variável do tipo mutex em um dado instante.
- ◆ Mesmo que diversas threads tentem efetuar o lock, apenas uma delas será bem sucedida.
- ◆ Nenhuma outra thread poderá efetuar o lock antes que a primeira thread o libere.

Mutex

- ◆ Um uso muito comum para um mutex é a sincronização da atualização de uma variável global.
- ◆ Com isso, garante-se que esta variável será atualizada de forma correta, como na versão sequencial do mesmo programa.
- ◆ Neste caso, esta variável global passa a ser uma região crítica

Mutex

- ◆ A seqüência de uso de uma variável do tipo mutex normalmente é:
 - Criar e inicializar uma variável do tipo mutex
 - Diversas threads tentam efetuar o lock no mutex
 - Apenas uma thread consegue
 - Essa thread realiza alguma computação
 - E depois libera o mutex
 - Outra thread efetua o lock e repete o processo
 - No final, o mutex é destruído

Mutex

- ◆ Quando diversas threads competem por um mutex, aquelas que não conseguiram efetuar o lock ficam bloqueadas na chamada da função de lock
- ◆ Uma forma não bloqueante seria usar um trylock ao invés de um lock

Mutex

- ◆ Variáveis do tipo mutex são do tipo `pthread_mutex_t` em C
- ◆ Mutexes inicialmente estão “unlocked”

Mutex

- ◆ Existem 4 tipos de mutexes
 - Fast
 - Error checking
 - Recursive
 - Timed (default)
- ◆ O tipo do mutex afeta a forma como ele é tratado quando uma thread que já efetuou o lock no mutex tenta fazê-lo novamente.

Mutex

- ◆ Fast: a thread fica bloqueada para sempre
- ◆ Recursive: a thread retorna da função de lock imediatamente como se tivesse conseguido efetuar o lock.
 - O número de vezes que a função de lock foi chamada para o mutex fica armazenado na estrutura do mesmo

Mutex

- ◆ Error checking: a thread retorna da função de lock imediatamente com o código de erro EDEADLK
- ◆ Timed: A thread fica bloqueada por um determinado tempo.
 - Para utilizar o mutex dessa forma, a função `pthread_mutex_timedlock` deve ser chamada no lugar de `pthread_mutex_lock`

Mutex

◆ Para inicializar uma variável do tipo mutex existem duas formas:

- Estática
- Dinâmica

Mutex

◆ Forma estática:

- Através de uma atribuição a uma variável do tipo `pthread_mutex_t` de uma das constantes:
 - ◆ `PTHREAD_MUTEX_INITIALIZER` (Timed)
 - ◆ `PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP` (Recursive)
 - ◆ `PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP` (Error Check)
 - ◆ `PTHREAD_MUTEX_ADAPTIVE_NP` (Fast)

Mutex

◆ Forma dinâmica:

■ Inicializar os atributos de um mutex com:

- ◆ `int pthread_mutexattr_init`
`(pthread_mutexattr_t *attr)`

■ Inicializar o mutex com esses atributos:

- ◆ `int pthread_mutex_init` (`pthread_mutex_t`
`*mx, const pthread_mutexattr_t *mxattr`)

Mutex

◆ Para alterar o valor da variável que vai alterar os atributos do mutex:

- `int pthread_mutexattr_settype`
`(pthread_mutexattr_t *attr, int kind)`

◆ Onde kind pode ser:

- `PTHREAD_MUTEX_TIMED_NP`
- `PTHREAD_MUTEX_RECURSIVE_NP`
- `PTHREAD_MUTEX_ERRORCHECK_NP`
- `PTHREAD_MUTEX_ADAPTIVE_NP`

Mutex

◆ Para liberar um mutex:

- `int pthread_mutex_destroy`
`(pthread_mutex_t *mx)`

Mutex

◆ Para efetuar o lock num mutex:

- `int pthread_mutex_lock(pthread_mutex_t`
`*mx);`

◆ Se o lock já tiver sido feito por outra thread, a thread corrente fica bloqueada até que o mutex seja liberado

Mutex

- ◆ Para liberar um mutex:

- `int pthread_mutex_unlock(pthread_mutex_t *mx);`

- ◆ Faz com que a thread deixe a região crítica

- ◆ Retorna erro se:

- O mutex já estiver liberado
 - Outra thread tiver feito lock no mutex estiver

Mutex

- ◆ Uma forma não bloqueante de entrar numa região crítica é através de:

- `int pthread_mutex_trylock(pthread_mutex_t *mx);`

- ◆ Se outra thread já tiver feito o lock, a chamada da função retorna erro

- ◆ É interessante para evitar deadlocks

Mutex

◆ Exercícios:

- Criar 4 threads que realizem o produto escalar de dois vetores
- Inserções em uma estrutura ortogonal

Variáveis Condicionais

- ◆ São uma outra forma de sincronização
- ◆ Provêm sincronização parcial
- ◆ Estão sempre associadas a mutexes

Variáveis Condicionais

◆ Thread Principal:

- Inicializa as variáveis globais que requerem a sincronização (Ex.: contadores)
- Inicializa a variável condicional
- Inicializa um mutex associado a variável condicional
- Cria as threads A e B para realizar a computação

Variáveis Condicionais

◆ Thread A

- Executa até o ponto que uma sincronização parcial é necessária
- Efetua o lock do mutex associado e verifica o valor da variável que requer a sincronização
- Realiza uma espera condicional, que automaticamente libera o mutex
- É acordada com o mutex alocado
- Libera o mutex e continua a execução

◆ Thread B

- Executa até o ponto que uma sincronização parcial é necessária
- Efetua o lock do mutex associado
- Altera o valor da variável que requer a sincronização
- Sinaliza a thread que está aguardando
- Libera o mutex associado



Variáveis Condicionais

◆ Duas maneiras de inicializar uma variável condicional:

■ Estática:

- ◆ `pthread_cond_t cond =
PTHREAD_COND_INITIALIZER;`

■ Dinâmica:

- ◆ `int pthread_cond_init (pthread_cond_t *cond,
pthread_condattr_t *cond_attr);`

Variáveis Condicionais

- ◆ Onde **attr** representa o conjunto de atributos de uma variável condicional

- ◆ Para os atributos default, `attr = NULL`

- ◆ No Linux, os atributos NÃO são implementados. Só existem por compatibilidade.

- ◆ Em outros UNIXs, poderiam ser utilizados para compartilhar variáveis condicionais com outros processos

Variáveis Condicionais

◆ Para bloquear uma thread:

- `int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mx);`

◆ Para sinalizar uma thread bloqueada:

- `int pthread_cond_signal (pthread_cond_t *cond);`
- Deve ser chamada depois de ter efetuado o lock no mutex associado

Variáveis Condicionais

◆ Para sinalizar todas as thread bloqueadas:

- `int pthread_cond_broadcast (pthread_cond_t *cond);`
- Também deve ser chamada depois de ter efetuado o lock no mutex associado

Variáveis Condicionais

◆ Exercício:

- Utilizando um buffer circular de 10 posições, implemente um algoritmo de produtor/consumidor (2 produtores e 2 consumidores)
- Produza números sequenciais
- Use as funções *random* e *sleep* para forçar a execução das *threads* de forma não sincronizada
- Imprima os elementos produzidos e consumidos