Sistemas Operacionais

Módulo 6 – Pthreads

Prof. Daniel Sundfeld
daniel.sundfeld@unb.br

Introdução

- Microprocessadores baseados em uma unidade central de processamento (CPU) apresentaram um grande avanço durante mais de duas décadas.
- A evolução começou a diminuir em 2003, devido ao crescente consumo de energia e dissipação de calor que limitaram o crescimento do clock e o número de instruções que podem ser executados em um período de clock em uma única CPU.

Introdução

- Os usuários e programadores estão acostumados a ver seu programa rodando mais rápido a cada nova geração de CPU;
- Isso não é mais verdade nos dias de hoje, um programa sequencial irá rodar em apenas um dos cores de um processador;
- Tradicionalmente, os softwares são escritos como programas sequenciais.

Introdução

- Praticamente, todos os fabricantes mudaram de paradigma de evolução, onde começaram a aumentar o número de núcleos de um processador para poder aumentar a capacidade de processamento.
- Uma forma de aproveitar esses diferentes núcleos é com a utilização de threads.
- Quais funções para criar, destruir e sincronizar threads?

Introdução

- Bem antes dos fabricantes mudarem o paradigma, um padrão já havia sido definido.
- A grande diversidade das possibilidades e implementações motivou a padronização. Por exemplo, alguns pacotes divergem até mesmo como criar uma thread (alguns não usam nome de função).
- Pthreads (“P” de “POSIX”) é uma padronização criada pelo IEEE e define um conjunto de tipos e chamadas na linguagem de programação C.

Criação de uma thread

- Pthread define funções para criar e terminar uma thread:
- *attr e *arg podem ser nulos.

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
void pthread_exit(void *retval);
```

Criação de uma thread

```c
#define NUM_THREADS     5

int main(void)
{
    int ret, i;
    pthread_t threads[NUM_THREADS];

    for (i = 0; i < NUM_THREADS; i++)
    {
        printf("Creating %d\n", i);
        ret = pthread_create(&threads[i], NULL, &thread_func, NULL);
        if (ret)
        {
            printf("Error %d on thread %d\n", ret, i);
            perror("pthread_create");
            exit(-1);
        }
    }

    pthread_exit(NULL);
}
```

```c
void *thread_func(void *arg)
{
    printf("Hello, world!\n");
    pthread_exit(NULL);
}
```

Término de threads

- Uma thread pode terminar nas seguintes situações:
- Quando a thread invoca pthread_exit();
- Quando a função passada por argumento para pthread_create retorna;
- Ela é cancelada;
- Alguma thread chama a função exit() ou a thread principal retorna na função main().

Término de threads

- Exemplos de thread terminando a execução:

```c
void *thread_func(void *arg)
{
    printf("Hello, world!\n");
    return NULL;
}
```

```c
void *thread_func(void *arg)
{
    printf("Hello, world!\n");
    pthread_exit(NULL);
}
```

Argumentos para threads

- Muitas vezes é necessário passar argumentos para threads, assim como é possível passar argumentos para funções.
- O argumento “arg”, da função pthread_create, é utilizado para isso.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

Argumentos para threads

```c
struct thread_arg {
    int num;
};

int main(void)
{
    int ret, i;
    struct thread_arg args[NUM_THREADS];
    pthread_t threads[NUM_THREADS];

    for (i = 0; i < NUM_THREADS; i++)
        args[i].num = i;

    for (i = 0; i < NUM_THREADS; i++)
    {
        printf("Creating %d\n", i);
        ret = pthread_create(&threads[i], NULL, &thread_func, &args[i]);
        if (ret)
        {
            printf("Error %d on thread %d\n", ret, i);
            perror("pthread_create");
            exit(-1);
        }
    }

    pthread_exit(NULL);
}
```

Argumentos para threads

```c
void *thread_func(void *arg)
{
    struct thread_arg *ctx = arg;

    printf("Hello, world %d!\n", ctx->num);
    return NULL;
}
```

```
user@station$ ./args
Creating 0
Creating 1
Creating 2
Hello, world 0!
Creating 3
Hello, world 1!
Hello, world 2!
Creating 4
Hello, world 3!
Hello, world 4!
```

PTHREAD__JOIN

```c
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

- Função que aguarda o término da thread (ou retorna imediatamente se a thread já terminou).
- Se retval != NULL, recebe o valor retornado pela thread.

PTHREAD__JOIN

```c
int main(void)
{
    (...)

    for (i = 0; i < NUM_THREADS; i++)
    {
        printf("Creating %d\n", i);
        ret = pthread_create(&threads[i], NULL, &thread_func, &args[i]);
        (...)
    }

    for (i = 0; i < NUM_THREADS; i++)
    {
        ret = pthread_join(threads[i], NULL);
        if (ret)
        {
            printf("Error waiting thread %d\n", i);
            perror("pthread_join");
            exit(-1);
        }
    }
    return 0;
}
```

PTHREAD__JOIN

- Após a saída com sucesso de um pthread_join, é garantido que a thread terminou.
- Múltiplas threads aguardando a mesma thread possuem comportamento indefinido.
- Qualquer thread pode aguardar qualquer thread.
- Não existe função “aguarde qualquer thread”.

> There is no pthreads analog of waitpid(-1, &status, 0), that is, "join with any terminated thread". If you believe you need this functionality, you probably need to rethink your application design. pthread_join(3)

PTHREAD_MUTEX

```c
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Inicialização
- Lock, unlock, try_lock
- Atributos de mutex: veja pthread_mutexattr(3)

PTHREAD__MUTEX

```c
#define NUM_THREADS     2
#define THREADS_LOOPS   1000 * 1000

static int count = 0;
static pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;

void *thread_func(void *arg)
{
    int i;

    for (i = 0; i < THREADS_LOOPS; i++)
    {
        pthread_mutex_lock(&my_mutex);
        count++;
        pthread_mutex_unlock(&my_mutex);
    }
    return NULL;
}
```

PTHREAD__MUTEX

```c
int main(void)
{

    for (i = 0; i < NUM_THREADS; i++)
    {
        ret = pthread_create(&threads[i], NULL, &thread_func, NULL);
        (...)
    }

    while (count < NUM_THREADS * THREADS_LOOPS)
        printf("count %d/%d\n", count, NUM_THREADS * THREADS_LOOPS);

    return 0;
}
```

```
user@station$ ./mutex
Creating 0
Creating 1
count 0/2000000
count 0/2000000
count 1/2000000
(...)
count 1999986/2000000
count 1999992/2000000
count 1999998/2000000
```

PTHREAD__MUTEX

- Se um sinal chega em uma função que está aguardando um mutex, depois de retornar do sighandler ela volta a bloquear no mutex como se não tivesse sido interrompida.
- Mutex foram criados como para serem primitivas baixo nível, que outras rotinas de sincronização se baseiam. Implementações de mutex tentam ser o mais eficiente possível.

PTHREAD__COND__WAIT

- Função que bloqueia aguardando uma condição;
- É necessário obter o mutex antes de bloquear;
- A thread libera o mutex e aguarda a condição "atomicamente"

```
pthread_mutex_lock(&count_mutex);
while (count > 0)
{
    pthread_cond_wait(&count_cond, &count_mutex);
    printf("%d\n", count);
}
printf("%d!\n", count);
pthread_mutex_unlock(&count_mutex);
```

PTHREAD__COND__WAIT

```c
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
pthread_mutex_t *restrict mutex,
const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond,
pthread_mutex_t *restrict mutex);
```

- Comportamento indefinido se não tiver o lock do mutex;
- Deve-se utilizar apenas um mutex por condição (nunca mais de um mutex);
- Pode-se utilizar mais de uma condição por mutex, comportamento desejado em muitos casos;
- Timed wait comporta-se como o wait, mas retorna erro depois de um certo tempo.

PTHREAD__COND__INIT

```
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- Macro que permite inicialização estática;
- Função para (re)inicialização dinâmica;
- Função que destrói, resultado é que para todos os efeitos, o objeto permanece não-inicializado;
- Comportamento não definido para: destruir condição quando outras threads estão aguardando, utilizar objeto não inicializados, iniciar mais de uma vez ou destruir mais de uma vez.

PTHREAD__COND__SIGNAL

```c
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

- Signal desbloqueia pelo menos uma thread que esteja bloqueada em uma condição.
- Se nenhuma estiver aguardando, nada acontece.
- Broadcast desbloqueia todas as threads que estejam aguardando. Nesse caso, uma thread é desbloqueada e as outras ficam aguardando o mutex ser liberado.

PTHREAD__COND__WAIT

```
#define NUM_THREADS     4
#define THREAD_COUNT    200

pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t count_cond = PTHREAD_COND_INITIALIZER;

int count = NUM_THREADS * THREAD_COUNT;

void *thread_func(void *arg)
{
    int i;

    for (i = 0; i < THREAD_COUNT; i++)
    {
        pthread_mutex_lock(&count_mutex);
        count--;
        pthread_cond_signal(&count_cond);

        pthread_mutex_unlock(&count_mutex);
    }
    return NULL;
}
```

PTHREAD__COND__WAIT

```
int main(void)
{
    (...)
    for (i = 0; i < NUM_THREADS; i++)
    {
        ret = pthread_create(&threads[i], NULL, &thread_func, NULL);
        (...)
    }

    pthread_mutex_lock(&count_mutex);
    printf("main: beginning %d\n", count);
    while (count > 0)
    {
        pthread_cond_wait(&count_cond, &count_mutex);
        printf("main: count is %d\n", count);
    }
    printf("main: bye %d!\n", count);
    pthread_mutex_unlock(&count_mutex);

    pthread_exit(NULL);
}
```

PTHREAD__COND__WAIT

```
user@station$ ./cond_wait
Creating 0
Creating 1
Creating 2
Creating 3
main: beginning 608
main: count is 596
main: count is 580
main: count is 502
main: count is 428
main: count is 0
main: bye 0!
```

Exemplo muito simples... Entre um cond_wait e outro o mutex não foi liberado!

PTHREAD__COND__WAIT

```c
int main(void)
{
    (...)
    for (i = 0; i < NUM_THREADS; i++)
        (...)

    pthread_mutex_lock(&count_mutex);
    printf("main: beginning %d\n", count);
    pthread_mutex_unlock(&count_mutex);

    while (1)
    {
        pthread_mutex_lock(&count_mutex);
        pthread_cond_wait(&count_cond, &count_mutex);
        printf("main: count is %d\n", count);
        aux = count;
        pthread_mutex_unlock(&count_mutex);

        long_func(aux);
    }
    // Never reached...
    printf("main: bye %d!\n", count);

    pthread_exit(NULL);
}
```

PTHREAD__COND__WAIT

```
user@station$ ./cond_wait
Creating 0
Creating 1
Creating 2
Creating 3
main: beginning 233
main: count is 232
main: count is 229
main: count is 228
main: count is 227
main: count is 189
main: count is 0
^C
```

```
user@station$ ./cond_wait
Creating 0
Creating 1
Creating 2
Creating 3
main: beginning 233
main: count is 155
main: count is 99
^C
```

PTHREAD__COND__WAIT

```c
int main(void)
{
    int aux = count;
    (...)
    for (i = 0; i < NUM_THREADS; i++)
        (...)

    pthread_mutex_lock(&count_mutex);
    printf("main: beginning %d\n", count);
    pthread_mutex_unlock(&count_mutex);

    while (aux > 0)
    {
        pthread_mutex_lock(&count_mutex);
        if (aux == count)
            pthread_cond_wait(&count_cond, &count_mutex);
        aux = count;
        printf("main: count is %d\n", count);
        pthread_mutex_unlock(&count_mutex);

        long_func(aux);
    }
    printf("main: bye %d!\n", count);

    pthread_exit(NULL);
}
```

PTHREAD__COND__WAIT

```
user@station$ ./cond_wait
Creating 0
Creating 1
Creating 2
Creating 3
main: beginning 100
main: count is 100
main: count is 99
main: count is 17
main: count is 0
main: bye 0!
```

PTHREAD__COND__WAIT

Esta linha corrige nosso problema antigo, mas não prevê um outro tipo de comportamento.

```
int main(void)
{
    int aux = count;
    (...)
    for (i = 0; i < NUM_THREADS; i++)
        (...)

    pthread_mutex_lock(&count_mutex);
    printf("main: beginning %d\n", count);
    pthread_mutex_unlock(&count_mutex);

    while (aux > 0)
    {
        pthread_mutex_lock(&count_mutex);
        if (aux == count)
            pthread_cond_wait(&count_cond, &count_mutex);
        aux = count;
        printf("main: count is %d\n", count);
        pthread_mutex_unlock(&count_mutex);

        long_func(aux);
    }
    printf("main: bye %d!\n", count);

    pthread_exit(NULL);
}
```

PTHREAD_COND_WAIT

- Spurious wakeups from the pthread_cond_timedwait() or pthread_cond_wait() functions may occur. Since the return from pthread_cond_timedwait() or pthread_cond_wait() does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return. [1];
- Ou seja, o uso correto é um “while” e não um if;
- Boa prática de programação: Sempre duvide de programas com pthread_cond_wait sem um while antes dele.

[1] - The Open Group Base Specifications Issue 6. IEEE Std 1003.1, 2004 Edition

Multiprogramação segura

- Nem toda função pode ser executada em paralelo;
- Tipicamente, variáveis globais, locks de arquivos não são seguras de ser chamadas;
- Funções que são seguras para serem chamadas em paralelo são chamadas de thread-safe.

Multiprogramação segura – dados static

```
int *func(void)
{
  static int i;
  (...)
  return &i;
}
```

- Funções que usam ou retornam ponteiros para dados estáticos não podem ser executadas em paralelo.
- Exemplo disto é a inet_ntoa, vastamente utilizada para converter um inteiro 32 bits em um endereço IP que possa ser lido como “10.0.0.1”.
- Por retornar um ponteiro para um buffer estático, não pode ser usada em múltiplas threads.
- Funções que usam dados estáticos foram recriadas para retornar dados em um buffer passado pelo usuário.

Multiprogramação segura - global

- Às vezes, o uso de variáveis globais sem proteção leva a leitura de dados incorretos que podem levar a segmentation fault.
- Algum mecanismo de sincronia entre as threads deve ser utilizado para a proteção dos dados.
- O uso de variáveis globais de sistema deve ser feito de forma que apenas uma thread precise acessá-las.

Multiprogramação segura - errno

- Chamadas de sistema tradicionais UNIX e POSIX usam a variável errno para indicar o erro das mais diversas funções;
- Comportamento completos de programas podem depender do valor dessa variável;
- O padrão Pthreads reconhece este problema e exige que cada thread receba seu errno.
- errno é definido como uma macro e cada thread tem o seu "errno local" [1].

[1] - Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. Pthreads Programming. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Multiprogramação segura – file locks

- Chamadas de sistema podem sincronizar threads com outros processos do sistema operacional: apenas um processo obtém o lock;
- O sistema não reconhece chamadas de lock de diferentes threads do programa e por isso não pode ser utilizada para sincronizar;
- Locks per-thread foram criados para resolver esse problema (Flockfile, Ftrylockfile, Funlockfile)

Multiprogramação segura – thread safe

- Funções de bibliotecas e manuais de programação vastamente definem as suas funções como “thread safe” e “thread unsafe”.
- Pode parecer óbvio, mas procure sempre utilizar as funções thread safe.
- Por exemplo: função signal() possue comportamento INDEFINIDO para programas multithread.

Conclusão

- Apesar de toda a dificuldade, diversos casos são conhecidos na literatura
- Lembrem-se: a tendência é que o número de cores apenas aumentem! Dominar o bom uso de threads é saber usar melhor os processadores do futuro.