

EA872: Laboratório de Programação de Software Básico

Atividade 10 - Programação com Threads

Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas

Eleri Cardozo
Matheus Fernandes de Oliveira
Marco A. Amaral Henriques

Abstract

Esta atividade tem por objetivo prover familiaridade com a programação concorrente baseada em threads (ou processos leves). As chamadas de sistema referentes a threads do padrão POSIX (pthreads) serão estudadas visando dotar o servidor Web de mecanismos de concorrência mais eficientes.

1 Introdução

O modelo convencional de processo no UNIX é limitado a uma única atividade no processo de cada vez, ou seja, em um determinado instante existe uma única linha de execução (thread ou fluxo de execução) no processo: trata-se de um modelo mono-threaded. Diferentemente deste modelo, sistemas operacionais modernos oferecem um modelo no qual muitas atividades avançam concorrentemente dentro de um único processo, isto é, um processo nestes sistemas pode ser multi-threaded.

Programação com múltiplas threads é uma técnica poderosa para tratar problemas naturalmente concorrentes como aqueles encontrados, por exemplo, em aplicações de tempo real. Isto porque o programador não necessita tratar múltiplas atividades, todas relacionadas a um mesmo contexto, do ponto de vista de um único fluxo de execução. Neste caso uma thread pode ser criada para tratar cada atividade lógica.

Nesse contexto é importante esclarecer a distinção entre processo e thread. Podemos considerar que o processo não é nada mais do que um hospedeiro de threads. Isto nos permite pensar da seguinte forma: não existem mais processos, existem somente threads. O que antes era considerado para nós um processo (como por exemplo a execução de *ls*) é, na realidade, uma única thread executando em um certo espaço de endereçamento. O processo se torna, então, um conceito conveniente para se caracterizar um espaço de endereçamento, um conjunto de descritores de arquivo e outros parâmetros. A Figura 1 ilustra este novo conceito de processo.

As threads são as unidades de escalonamento dentro de um processo. Em outras palavras, processos na realidade não são executados, mas sim suas threads. Cada processo no sistema possui pelo menos uma thread. Do ponto de vista do usuário o contexto de uma thread consiste de um conjunto de registros, uma pilha e uma prioridade de execução (Figura 2).

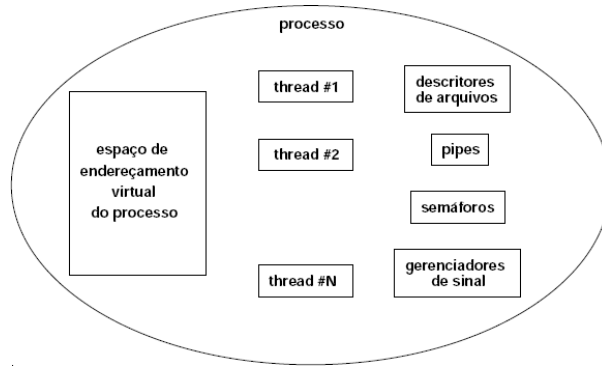


Figure 1: Estrutura de um processo

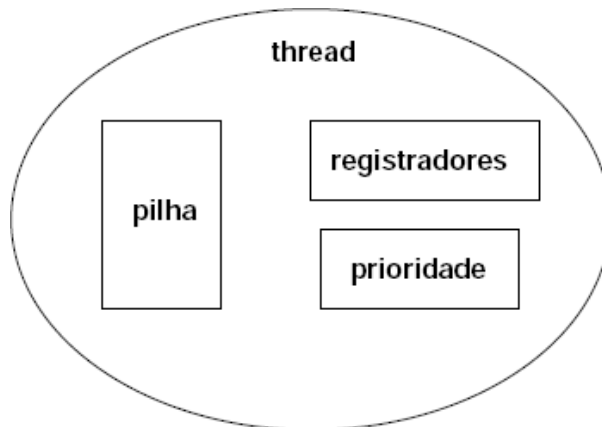


Figure 2: Estrutura de uma thread

Como comentado anteriormente, as threads compartilham todos os recursos possuídos pelo processo que as criou. Todas as threads dentro de um processo compartilham o mesmo espaço de endereço virtual, arquivos abertos, semáforos e filas. Cada thread encontra-se em um destes três estados: executando, pronta para executar ou bloqueada. Somente uma única thread no sistema encontra-se no estado execução quando se trata de uma plataforma de uma única CPU com um único núcleo (core). A thread que se encontra em execução é aquela que foi selecionada pelo sistema operacional para ocupar a CPU de acordo com a política de escalonamento. Uma thread no estado “pronta para executar” está apenas aguardando sua vez de ocupar a CPU, não havendo nenhuma outra pendência que a impeça de ser executada. Threads que se encontram no estado de bloqueio estão esperando a ocorrência de um evento como, por exemplo, a chegada de um dado solicitado ao disco.

Na sequência, são discutidos alguns aspectos relativos à manipulação das threads utilizando-se como referência o padrão POSIX para as chamadas de sistema referentes a threads. Infelizmente, algumas versões do UNIX e do Linux implementam threads de forma não padronizada e precisamos descobrir e nos adaptar a eventuais diferenças.

2 POSIX Threads

2.1 Criação de Threads

Threads são criadas através da chamada `pthread_create`. A chamada retorna um inteiro indicando sucesso ou falha na criação da thread. Os parâmetros da chamada são:

1. um ponteiro para uma variável do tipo `pthread_t` que será atribuída pela chamada (esta variável é utilizada para referenciar a thread em chamadas subsequentes);
2. um ponteiro para uma variável do tipo `pthread_attr_t` contendo atributos para a nova thread (prioridade, política de escalonamento, etc). O valor `NULL` para este parâmetro força a utilização de atributos default;
3. um ponteiro para a rotina C que implementa o código da thread. Esta rotina deve aceitar um único parâmetro de tipo `void*` e retornar `void`;
4. um ponteiro do tipo `void*` para o parâmetro a ser passado para a rotina acima.

2.2 Terminação de Threads

Threads são terminadas através da chamada `pthread_exit`. Esta chamada tem como parâmetro um ponteiro para uma área de dados que poderá ser inspecionada por outra thread através da chamada `pthread_join`.

2.3 Sincronização de Threads

Threads de um mesmo processo podem ser sincronizadas de três maneiras:

1. sequenciamento: uma thread bloqueia até que outra termine;
2. mútua exclusão: threads operam regiões críticas protegidas por variáveis mutex (semáforos binários);
3. variáveis de condição: uma thread de posse de um mutex pode suspender sua execução e liberar o mutex em benefício de outra thread bloqueada no mesmo mutex.

O sequenciamento entre threads se dá com a chamada `pthread_join`. Esta chamada possui dois parâmetros:

1. um ponteiro para um identificador de thread (`pthread_t*`) indicando a thread alvo da sincronização;
2. um duplo ponteiro para coletar os dados passados pela thread alvo quando esta invocar `pthread_exit`.

A thread que executa `pthread_join` é suspensa até que a thread alvo termine ou seja cancelada. Uma restrição importante quanto ao uso de `pthread_join` é que apenas duas threads podem sincronizar suas execuções por meio desta chamada.

Mútua exclusão é implementada com o auxílio de mutex (semáforos binários). A chamada `pthread_mutex_init` inicializa um mutex, que fica no estado desbloqueado.

A chamada `pthread_mutex_lock` bloqueia o mutex caso este esteja desbloqueado. Caso contrário (mutex já bloqueado) a thread é bloqueada até o desbloqueio do mutex.

A chamada `pthread_mutex_unlock` desbloqueia o mutex caso o mesmo tenha sido bloqueado pela thread que invocou a chamada. As chamadas `pthread_mutex_lock` e `pthread_mutex_unlock` são delimitadores de regiões críticas para threads.

Variáveis de condição são implementadas com as chamadas de sistema `pthread_cond_wait` e `pthread_cond_signal`. Ambas as chamadas ocorrem dentro de uma região crítica.

A chamada `pthread_cond_wait` suspende a thread e libera o mutex para outra thread bloqueada na mesma região crítica. Esta segunda thread, imediatamente antes de encerrar a região crítica, deve sinalizar a primeira thread para que a mesma assuma novamente a posse do mutex e continue sua execução. Esta sinalização se dá com a chamada `pthread_cond_signal`.

2.4 Atributos de Threads

Uma thread possui um conjunto de atributos a ela associados, tais como:

- estado: indica se a thread é passível de join (estado `PTHREAD_CREATE_JOINABLE`) ou não (estado `PTHREAD_CREATE_DETACHED`);
- política de escalonamento: round robin (`SCHED_RR`) ou fifo (`SCHED_FIFO`);
- prioridade: prioridade da thread;
- escopo: define o escopo de escalonamento que pode ser referente ao processo (`PTHREAD_SCOPE_PROCESS`) ou ao sistema (`PTHREAD_SCOPE_SYSTEM`).

Estes atributos são alterados através da chamada `pthread_attr_t`, onde `xxx` define uma família de chamadas do tipo set e get. Estas chamadas operam sobre uma estrutura do tipo `pthread_attr_t`, a mesma utilizada em `pthread_create`.

3 Atividades

3.1 Atividades em sala de aula

Durante o período do laboratório, o aluno deverá responder diretamente no Moodle as questões lá formuladas sobre o conteúdo deste roteiro. O valor total das questões é de 3,0 pontos.

3.2 Atividades a serem entregues até a próxima aula

As atividades a seguir têm um valor total de 7 pontos e devem ser entregues em um relatório de acordo com o calendário divulgado.

Para compilar os programas com threads, utilize a biblioteca `libpthread`.

```
gcc progX.c -o progX -lpthread
```

Detalhes sobre cada uma das chamadas de sistema referentes a threads podem ser obtidos por meio de `man pthread_XXX` (POSIX Threads), onde `XXX` deve ser substituído por uma das diversas funções disponíveis e/ou usadas nos códigos abaixo.

1. Exemplo simples de criação da threads

Compile e em seguida execute o programa `prog1.c` passando como argumento na linha de comando o número de threads a serem criadas: `./prog1 <num_threads>`.

1.1 (0,5) Verifique e explique no relatório o que ocorre com a ordem da execução do programa para diferentes valores de `<num_threads>` abaixo e acima de 7.

1.2 (1,0) Explique o funcionamento do programa, detalhando as ações realizadas e os motivos das saídas produzidas.

1.3 (0,5) Qual o papel da função `void * PrintHello(void * data)`?

1.4 (0,5) Explique o efeito causado no funcionamento do programa pela remoção da linha destacada.

```
#include <pthread.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#define MAX_THREADS 50

void * PrintHello(void * data)
{
    printf("Thread %ld - Criada na iteracao %ld.\n", pthread_self(), (intptr_t)data);
    pthread_exit(NULL);
}

int main(int argc, char * argv[])
{
    int rc, i, n;
    pthread_t thread_id[MAX_THREADS];

    if(argc != 2){
printf("Uso: %s <num_threads>\n", argv[0]);
exit(1);
    }
    n = atoi(argv[1]);
    if(n > MAX_THREADS || n <= 0) n = MAX_THREADS;
    for(i = 0; i < n; i++)
    {
        rc = pthread_create(&thread_id[i], NULL, PrintHello, (void*)(intptr_t)i);
        if(rc)
        {
            printf("\n ERRO: codigo de retorno de pthread_create eh %d \n", rc);
            exit(1);
        }
        printf("\n Thread %ld. Criei a thread %ld na iteracao %d ... \n",
            pthread_self(), thread_id[i], i);
        //////////////////////////////////////
        if(i % 3 == 0) sleep(2); //<-- linha destacada
        //////////////////////////////////////
    }
}
```

```
    pthread_exit(NULL);
}
```

2. Sincronização de threads

Compile e em seguida execute o programa prog2.c passando como argumento na linha de comando o número de threads a serem criadas: `./prog2 <num_threads>`.

2.1 (0,5) Verifique o que ocorre para diferentes valores de `<num_threads>`. Explique no relatório os motivos do comportamento observado.

2.2 (1,0) Explique a influência das funções `pthread_join`, `pthread_attr_init`, `pthread_attr_setdetachstate` e também `pthread_attr_destroy` na execução do programa.

2.3 (0,5) Explique a razão das threads terminarem em uma ordem diferente da ordem em que foram criadas e como a função `main` trata essa questão.

```
#include <pthread.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#define MAX_THREADS      10

void *WorkerThread(void *t)
{
    int i, tid;
    double result=0.0;
    tid = (intptr_t)t;

    printf("Comecando thread %d ...\n",tid);

    for (i=0; i<1000000; i++)
    {
        result = result + (double)random();
    }

    printf("Thread %d terminada. Resultado = %e\n",tid, result);
    pthread_exit((void*)(intptr_t)t);
}

int main(int argc, char *argv[])
{
    pthread_t thread[MAX_THREADS];
    pthread_attr_t attr;
    int n, rc, t, k;
    void *status;

    if(argc != 2){
printf("Uso: %s <num_threads>\n", argv[0]);
exit(1);
    }

    n = atoi(argv[1]);
    if(n > MAX_THREADS || n <= 0) n = MAX_THREADS;
```

```

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

for(t = 0; t < n; t++) {
    printf("Main: criando thread %d\n", t);
    rc = pthread_create(&thread[t], &attr, WorkerThread, (void *)(&t));
    if (rc) {
        printf("\n ERRO: codigo de retorno de pthread_create eh %d \n", rc);
        exit(1);
    }
}

pthread_attr_destroy(&attr);
for(t = 0; t < n ; t++) {
    rc = pthread_join(thread[t], &status);
    if (rc) {
        printf("\n ERRO: codigo de retorno de pthread_join eh %d \n", rc);
        exit(1);
    }
    k = (&t);
    printf("Main: completou join com thread %d com status %d\n", t, k);
}

printf("Programa %s terminado\n", argv[0]);
pthread_exit(NULL);
}

```

3. Modelo produtor-consumidor

O programa a seguir ilustra o modelo produtor-consumidor implementado com threads. Observe o uso de mutex para definir uma região crítica para a manipulação do buffer e de variáveis de condição para ceder o mutex em caso de buffer cheio (produtor) ou vazio (consumidor).

Estude o código do modelo produtor/consumidor com processos e threads e responda:

3.1 (0,5) O que acontece se retirarmos a chamada `pthread_join` da função principal?

3.2 (1,0) Programas multi-threaded são passíveis de inconsistências. Elimine do programa as delimitações das regiões críticas e a sincronização entre produtor e consumidor (variáveis de condição). O que foi observado? Como você explica este comportamento?

3.3 (1,0) Programas multi-threaded são passíveis de deadlock. Altere o mecanismo de sincronização entre produtor e consumidor para provocar um deadlock. O que foi alterado e porque a sua alteração causou deadlock?

```

#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#define MAXITENS 20

/* variaveis globais */
int buffer[MAXITENS];
int pointer = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond1 = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond2 = PTHREAD_COND_INITIALIZER;

```

```

/* gera um num. aleatorio de itens para produzir / consumir */
int nitens() {
    sleep(1);
/*  return(rand() % MAXITENS); <= esta linha deve ser corrigida
    como mostrado abaixo. */
    return(rand() % MAXITENS + 1); /* correcao feita para evitar
                                   que seja retornado o valor 0. */
}

/* insere item */
void insere_item(int item) {
    buffer[pointer] = item;
    printf("\nitem %d inserido (valor=%d)", pointer, item);
    fflush(stdout);
    pointer++;
    sleep(1);
}

/* remove item */
int remove_item() {
    pointer--;
    printf("\nitem %d removido (valor=%d)", pointer, buffer[pointer]);
    fflush(stdout);
    sleep(1);
}

/* produtor */
void* produtor(void* in) {
    int i, k;
    printf("\nProdutor iniciando...");
    fflush(stdout);
    while(1) {
        pthread_mutex_lock(&mutex);
        /* produz */
        k = nitens();
        printf("\nNum. de itens a tratar no produtor =%d\n",k);
        for(i = 0; i < k; i++) {
            if(pointer == MAXITENS) { /* buffer cheio -- aguarda */
                printf("\nBuffer cheio -- produtor aguardando..."); fflush(stdout);
                pthread_cond_wait(&cond1, &mutex);
                printf("\nProdutor reiniciando..."); fflush(stdout);
            }
            insere_item(rand() % 100);
        }
        pthread_cond_signal(&cond2);
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
}

/* consumidor */
void* consumidor(void* in) {
    int i, k;
    printf("\nConsumidor iniciando...");
    fflush(stdout);

```



```

while(1) {
    pthread_mutex_lock(&mutex);
    /* consome */
    k = nitens();
    printf("\nNum. de itens a tratar no consumidor =%d\n",k);
    for(i = 0; i < k; i++) {
        if(pointer == 0) { /* buffer vazio */
            printf("\nBuffer vazio -- consumidor aguardando..."); fflush(stdout);
            pthread_cond_wait(&cond2, &mutex);
            printf("\nConsumidor reiniciando..."); fflush(stdout);
        }
        remove_item();
    }
    pthread_cond_signal(&cond1);
    pthread_mutex_unlock(&mutex);
    sleep(1);
}

int main() {
    pthread_t thread1, thread2;
    pthread_create(&thread2, NULL, &consumidor, NULL);
    pthread_create(&thread1, NULL, &produtor, NULL);
    pthread_join( thread1, NULL);
    exit(0);
}

```