EA872
Laboratório de Programação de Software Básico

Atividade 11 - Programação com Threads

Eleri Cardozo
Matheus Fernandes de Oliveira
Marco A. A. Henriques
Ricardo R. Gudwin

Departamento de Engenharia de Computação
e Automação Industrial
Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas

2s/2013

Conteúdo

Resumo

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

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 como Solaris, Linux e Windows 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.

Neste 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.

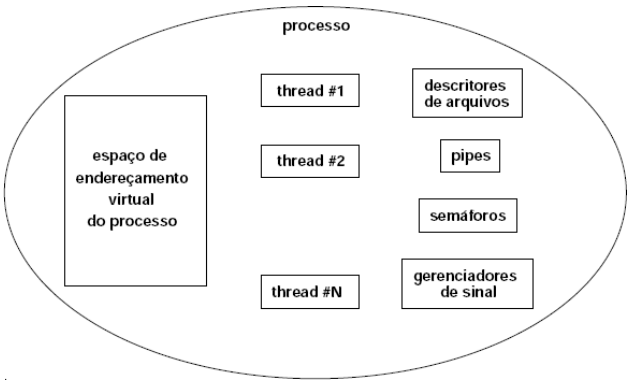

Figura 1: Estrutura de um processo.

As threads são as unidades de escalonamento dentro de um processo. Em outras palavras, processos na realidade não executam, 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, conforme mostrado na Figura 2.

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

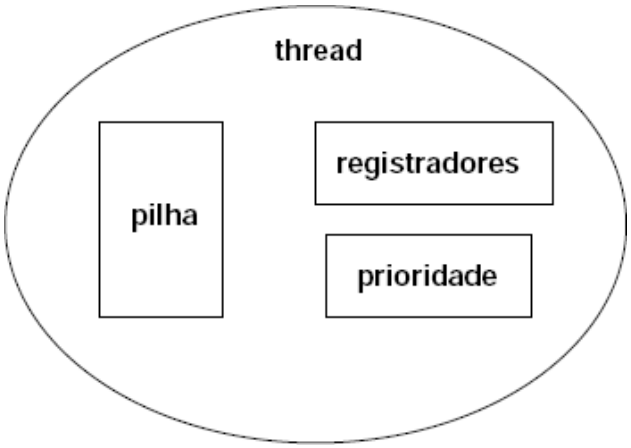

Figura 2: Estrutura de uma thread.

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. 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. Threads no estado "pronta para executar"estão 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, por exemplo, a chegada de um dado solicitado ao disco.

Na seqüência, são discutidos alguns aspectos relativos à manipulação das threads utilizando-se como referência o padrão POSIX para as chamadas de sistema referentes a threads. Algumas versões do UNIX implementarem threads de forma não padronizada.

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 subseqüentes);

2. um ponteiro para uma variável do tipo `pthread_att_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 implenta o código da thread. Esta rotina deve aceitar um único parâmetro de tipo `void*` e retornar `void`;

4. um ponteiro `(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. seqüenciamento: uma thread bloqueia até que outra termine;
2. mútua exclusão: threads operam regiões críticas protegidas por mutex;
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 seqüenciamento 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 critica. 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_xxx`, 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 Exemplo de Utilização de Threads

3.1 Criação da threads

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

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

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

    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*)i);
        if(rc)
        {
             printf("\n ERRO: codigo de retorno de pthread_create eh %d \n", rc);
             exit(1);
        }
        printf("\n Thread %d. Criei a thread %d na iteracao %d ...\n",
                pthread_self(), thread_id[i], i);
        if(i % 5 == 0) sleep(1); //<-- linha 36
    }

    pthread_exit(NULL);
```

```
}
```

3.2 Sequenciamento de threads

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

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

int main(int argc, char *argv[])
{
   pthread_t thread[MAX_THREADS];
   pthread_attr_t attr;
   int n, rc, t;
   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);
      }
      printf("Main: completou join com thread %d com status %d\n", t, (int)status);
   }

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

3.3 Sincronização de threads

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).

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#define MAXITENS 8

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

/* gera um numero aleatorio de itens para produzir / consumir */
int nitens() {
  sleep(1);
  return(rand() % MAXITENS);
}

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

```
}

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

/* produtor */
void* produtor(void* in) {
  int i, k;
  printf("\nProdutor iniciando...");
  fflush(stdout);
  while(1) {
    pthread_mutex_lock(&mutex);
    /* produz */
    k = nitens();
    for(i = 0; i < k; i++) {
      if(pointer == MAXITENS) { /* buffer cheio -- aguarda */
        printf("\nBuffer cheio -- produtor aguardando..."); fflush(stdout);
        pthread_cond_wait(&cond, &mutex);
        printf("\nProdutor reiniciando..."); fflush(stdout);
      }
      insere_item(rand() % 100);
    }
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
  }
}

/* consumidor */
void* consumidor(void* in) {
  int i, k;
  printf("\nConsumidor iniciando...");
  fflush(stdout);
  while(1) {
    pthread_mutex_lock(&mutex);
    /* consome */
    k = nitens();
    for(i = 0; i < k; i++) {
      if(pointer == 0) { /* buffer vazio */
        printf("\nBuffer vazio -- consumidor aguardando..."); fflush(stdout);
        pthread_cond_wait(&cond, &mutex);
        printf("\nConsumidor reiniciando..."); fflush(stdout);
      }
      remove_item();
```

```
    }
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
  }
}

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

4 Projeto do Servidor Web

Na atividade anterior você utilizou processos para tornar seu servidor concorrente. Nesta atividade, você deverá implementar a concorrência por meio de threads.

O trecho referente a execução no processo filho deve ser isolado em uma função que executará como thread. A seguir, a chamada `fork` deverá ser substituída pela chamada `pthread_create` e o servidor deverá criar uma thread por requisição. Como as threads acessarão apenas o sistema de arquivos, nenhum mecanismo de sincronização se faz necessário (o sistema operacional cuida do acesso concorrente ao sistema de arquivos).

Nesta atividade você deve elaborar um fluxograma completo de seu servidor e documentar em detalhes o novo código desenvolvido (partes novas apenas).

Teste comparativo das versões do servidor

1. Utilize o programa http-load e faça medidas de desempenho para três versões de servidor web: a versão com só um processo (lab9), a versão com vários processos de uma só thread (lab10) e esta nova versão com um processo e várias threads (lab11). Procure estressar o servidor simulando várias requisições simultâneas e busque o melhor desempenho.

2. Documente os testes feitos (parâmetros utilizados etc.).

3. Explique os resultados obtidos.

4. Apresente sugestões de melhorias em cada uma das três versões que aumentariam o desempenho máximo obtido nestes testes iniciais.

5. Usando os programas do lab10 e lab11, varie o número máximo de processos (lab10) e de threads (lab11) de acordo com a seguinte lista: 1, 2, 4, 8, 16, 32 e 64 processos (ou threads). Para cada número de processos (ou threads) listado, determine com ajuda do http-load qual é a melhor velocidade que pode ser obtida (em requisições atendidas por segundo (por exemplo) e plote os resultados em um gráfico "núm. de processos (ou threads) X velocidade". Avalie e discuta seus resultados, justificando-os.