Programação paralela com bibliotecas básicas

Aleardo Manacero Jr.

Paralelismo com bibliotecas básicas

Pode ser feito tanto para ambientes de memória compartilhada quanto de troca de mensagens

Mecanismos de memória compartilhada

THREADS

São “processos” que executam sem a carga de gerenciamento normal

Seu uso permite a construção de programas paralelos, mesmo usando apenas uma única máquina (aplicações *multithreaded*)

Com o surgimento dos processadores multicore tornou-se ainda mais importante

Padronização de threads

Padrão POSIX define um conjunto de funções C, que são agrupadas em uma biblioteca destinada a programação de múltiplos threads

Essa biblioteca é denominada *pthreads*, sendo padrão em todos sistemas UNIX-like e, até mesmo, no Windows NT (e seus sucessores)

Uso de *threads*

Para o uso de um thread é preciso antes cria-lo. Para isso se faz:

PASSO 1 – inclusão da biblioteca

```
#include <pthread.h>
```

PASSO 2 – declarações

```
pthread_attr_t atrib;//atributos do thread
pthread_t ident; //descritor do thread
```

Uso de threads

PASSO 3 – atribuição de valores aos threads

```
pthread_attr_init(atrib);
/* atribui valores padrão */
pthread_attr_setscope(&atrib,
PTHREAD_SCOPE_SYSTEM);
/* usado para modificar o escopo de
  escalonamento do thread, sendo:
SYSTEM – disputa com todos os threads do
  sistema
PROCESS – disputa apenas com threads do
  mesmo processo */
```

Uso de threads

PASSO 4 – criação efetiva do thread

```
pthread_create(&ident, &atrib,
  start_func, arg);
  /*start_func é o nome da função
  inicial de um thread, com
  parâmetro igual a arg */
  /* Essa função retorna 0 se a
  criação for bem sucedida */
```

Uso de threads

PASSO 5 – uso efetivo do thread

Um thread pode ser visto como uma função, diferenciando-se, é claro, pelo tipo de tratamento recebido do sistema operacional.

Entram em atividade pela execução de *pthread_create* e podem se comunicar e controlar sua execução por primitivas próprias.

Uso de threads

PASSO 6 – encerrando um thread

```
pthread_exit(valor); /* forma amigável e sinalizada. A variável valor é um ponteiro para o retorno de algum valor ao processo pai */
```

Pode ser feita também apenas pelo encerramento da função inicialmente chamada

Uso de threads

PASSO 6 – encerrando um thread *(cont.)*

Do lado do processo que criou o thread (seu pai), após chamar *pthread_create* o processo continua a execução.

Para faze-lo esperar pelo(s) filho(s) se deve executar

```
pthread_join(ident,ptr_valor);
/*  ident é o descritor do filho e
ptr valor aponta para o ponteiro de
retorno usado por pthread_exit()*/
```

Sincronizando *threads*

O disparo de vários *threads,* que colaboram na execução de uma dada tarefa (paralelismo), exige um trabalho de sincronização para que o resultado seja o esperado.

A biblioteca *pthreads* faz isso usando semáforos ou bloqueios e variáveis condicionais

Sincronismo e Regiões Críticas

Região crítica é um trecho de um programa em que é necessário acesso exclusivo a algum recurso (exclusão mútua)

Sincronismo identifica a necessidade de coordenação de ações entre os processos

Semáforos foram criados para resolver os problemas de regiões críticas e de sincronismo

Semáforos (Dijkstra)

O uso dos semáforos, na forma definida por Dijkstra (primitivas P e V), é concretizado pela inclusão da biblioteca semaphore.h

Seu uso demanda a declaração e atribuição de valores iniciais ao semáforo, o que é feito pela função que ativa os threads

Semáforos (Dijkstra)

Apenas lembrando:

P(s) → bloqueia o processo se s=0 ou
libera o processo e faz s=s-1 caso s>0

V(s) → faz s=s+1 e libera algum processo
que esteja bloqueado pelo semáforo s

Semáforos

Declaração de um semáforo:

```
sem_t mutex;// declara o semáforo mutex
```

Atribuição de valor inicial ao semáforo:

```
sem_init(&mutex, SHARED, 1);   /* faz
mutex=1, identificando-o como
globalmente compartilhado se SHARED=1
ou localmente (apenas threads do mesmo
processo) se SHARED=0 */
```

Semáforos

Controle dos valores do semáforo:

```
sem_wait(&mutex); /* equivale à primitiva P(mutex), ou seja, o thread é bloqueado se mutex=0 */

sem_post(&mutex); /* equivale ao V(mutex), ou seja, libera processos esperando por mutex */
```

Exemplo de uso de semáforos

O problema do produtor/consumidor envolve o sincronismo entre dois processos

Um que produz algo, em uma variável compartilhada, que será consumido pelo outro

O compartilhamento deve ser feito de forma a que um não produza/consuma enquanto o outro estiver acessando a variável compartilhada, demandando semáforos

Exemplo de uso de semáforos

Produtor/Consumidor

```
#include <stdio.h>
#include <pthread.h>      /* linhas padrão */
#include <semaphore.h>
#define SHARED 1
/* protótipos dos dois threads */
void *Producer(void*);
void *Consumer(void*);
```

Exemplo de uso de semáforos


```
sem_t empty, full;  /* semáforos globais */
int data;           /* buffer compartilhado */
int numIters;

/* main lê linha de comando criando os threads*/
int main(int argc, char *argv[]) {

  pthread_t pid, cid; /* id dos threads */
  pthread_attr_t attr; /* atributos comuns */
```

Exemplo de uso de semáforos


```
pthread_attr_init(&attr);
pthread_attr_setscope(&attr,PTHREAD_SCOPE_SYSTEM);

sem_init(&empty, SHARED, 1); /* sem empty = 1 */
sem_init(&full, SHARED, 0);  /* sem full = 0  */

 numIters = atoi(argv[1]);
 pthread_create(&pid, &attr, Producer, NULL);
 pthread_create(&cid, &attr, Consumer, NULL);
 pthread_join(pid, NULL);
 pthread_join(cid, NULL) ;
}  /* Fim da função main() */
```

Exemplo de uso de semáforos


```
/* deposita 1, ..., numIters no buffer de dados */
void *Producer(void *arg) {
int produced;

 for (produced = 1; produced <= numIters; produced++)
 {
    sem_wait(&empty);
    data = produced;
    sem_post(&full);
 }
} * fim do thread Producer */
```

Exemplo de uso de semáforos


```
/* busca numIters items no buffer e os soma */
void *Consumer(void *arg) {
int total=0, consumed;

  for (consumed=1; consumed <= numIters; consumed++)
  {  sem_wait(&full);
     total = total + data;
     sem_post(&empty);
  }
  printf("the total is %d\n", total);
} * fim do thread Consumer */
```

Bloqueios e variáveis condicionais

A biblioteca *pthreads* possui outras formas (além dos semáforos) para tratar regiões críticas e sincronismo de processos

Regiões críticas podem ser tratadas com bloqueios (*locks*)

Sincronismo com variáveis condicionais

Bloqueios

Seguem a mesma estrutura de um *thread* normal, isto é, devem ser criados seus atributos, descritores, etc.

DECLARAÇÃO:

```
pthread_mutex_t mutex; /* define um
bloqueio de exclusão mútua */
pthread_mutex_init (&mutex,NULL); /*
valor inicial do bloqueio */
```

Bloqueios

USO:

```
pthread_mutex_lock(&mutex);
/* verifica se o bloqueio está
liberado ou não, bloqueando-o se
estava liberado */

pthread_mutex_unlock(&mutex);
/* libera o bloqueio da região
crítica cercada por mutex */
```

Variáveis condicionais

DECLARAÇÃO

```
pthread_cond_t  cond;
/* declaração da variável condicional
cond */
pthread_cond_init (&cond,NULL);
/* atribui um valor inicial (NULL)
para a variável condicional */
```

Variáveis condicionais

USO

```
pthread_cond_wait(&cond,&mutex);
/* libera o uso do bloqueio mutex e
espera sinalização em cond */

pthread_cond_signal(&cond);

pthread_cond_broadcast(&cond);
/* sinalizam a liberação da condição
para o(s) thread(s) que havia(m)
liberado o bloqueio mutex */
```

Variáveis condicionais

Vale explicitar que `pthread_cond_wait` libera o uso do bloqueio (*mutex*) **mas** mantém o processo que a chamou bloqueado

Enquanto isso `pthread_cond_broadcast` faz a liberação de fato dos processos que esperavam por essa sinalização (*cond*)

Exemplo de barreiras (globais)


```
#include <pthread.h>
#include <stdio.h>
#define SHARED 1
#define MAXSIZE 2000 /* tamanho máximo da matriz */
#define MAXWORKERS 4 /* número máximo de workers */

pthread_mutex_t barrier; /* lock para a barreira */
pthread_cond_t go;       /* variável condicional */
int numWorkers;     /* número de worker threads  */
int numArrived = 0; /* quantos estão na barreira */
```

Exemplo de barreiras (globais)


```
void *Worker(void *);
   /* protótipo do thread */

int size, stripSize;
   /* size == stripSize*numWorkers */

int sums[MAXWORKERS];
   /* somas calculadas por cada worker */

int matrix[MAXSIZE][MAXSIZE];
```

Exemplo de barreiras (função main)


```
/* lê a linha de comando, iniciliza e cria
   os threads */
int main(int argc, char *argv[]) {
 int i, j;
 pthread_attr_t attr;
 pthread_t workerid[MAXWORKERS];
```

Exemplo de barreiras (função main)


```
/* define atributos globais dos threads */
 pthread_attr_init(&attr);
 pthread_attr_setscope(&attr,PTHREAD_SCOPE_SYSTEM);

/*inicializa mutex e variável condicional*/
 pthread_mutex_init(&barrier, NULL) ;
 pthread_cond_init(&go, NULL) ;
```

Exemplo de barreiras (função main)


```
/* Usa valores da linha de comando */
 size = atoi(argv[1]) ;
 numWorkers = atoi(argv[2]);
 stripSize = size/numWorkers;

/* inicializa a matriz */
 for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
      matrix[i][j] = 1;
```

Exemplo de barreiras (função main)


```
/* cria os workers e termina o thread principal */
for (i = 0; i < numWorkers; i++)
    pthread_create(&workerid[i], &attr,
                   Worker, (void *) i);
 pthread_exit(NULL);
}
```

Exemplo de barreiras (thread worker)


```
// THREAD WORKER
/* Cada worker soma os valores em uma faixa.
Depois da barreira worker[0] imprime o total */
void *Worker(void *arg) {
 int myid = (int) arg;
 int total, i, j, first, last;

/* calculo 1a e última linha de minha faixa */
 first = myid * stripSize;
 last = first + stripSize -1;
```

Exemplo de barreiras (thread worker)


```
/* somo os valores na minha faixa */
 total = 0;
 for (i = first; i <= last; i++)
    for (j = 0; j < size; j++)
       total += matrix[i][j];
  sums [myid] = total ;
Barrier(); /* espero que todos terminem */
```

Exemplo de barreiras (thread worker)


```
if (myid == 0)
    {/* apenas worker 0 calcula o total */
      total = 0;
      for (i = 0; i < numWorkers; i++)
          total += sums [i];
      printf("the total is %d\n", total);
    }
} /* final do thread Worker */
```

Exemplo de barreiras (função barrier)


```
/* A barreira contadora reutilizável */
void Barrier () {
  pthread_mutex_lock (&barrier);
  numArrived++;
  if (numArrived < numWorkers)
     pthread_cond_wait (&go, &barrier);
    else {
     numArrived =0;/* último a chegar acorda demais*/
     pthread_cond_broadcast(&go);
    }
  pthread_mutex_unlock(&barrier);
}
```

Troca de mensagens

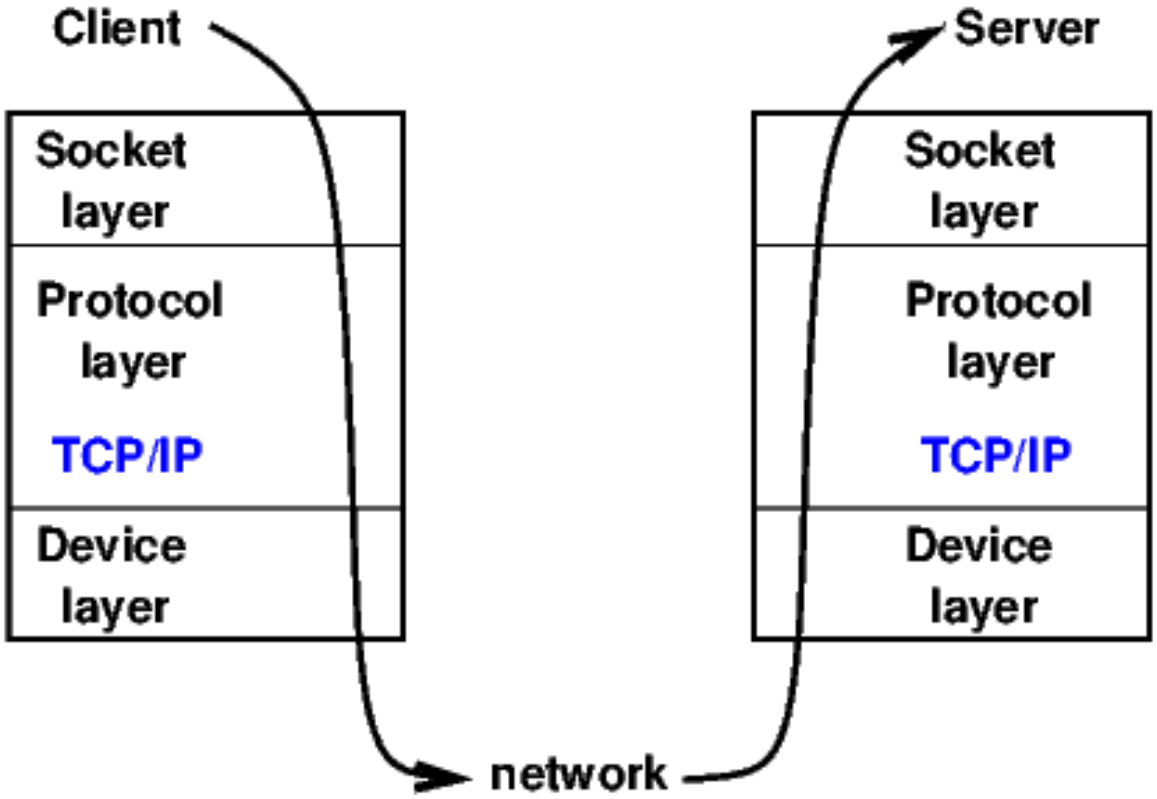
Troca de mensagens

O uso de *threads* não é viável para interação entre processos quando o paralelismo ocorrer usando sistemas multicomputadores

Nesses casos se faz uso de outra forma de interação, baseada no modelo cliente-servidor

O modelo cliente-servidor é importante para a implementação de ambientes de troca de mensagens

Modelo de troca de mensagens



Troca de mensagens

É feita em cima dos protocolos TCP/UDP, usando bibliotecas padrão, como *SOCKETS* e *NSL*

A comunicação ocorre por meio da criação de *sockets*, condicionada à existência de um processo servidor!

Criação de um *socket*

sd = socket (format, type, protocol);

em que:

format – system/internet

type – circuito virtual/datagrama

protocol – TCP/UDP

host = gethostbyname (host_identification)

Associa endereço de um host

Criação de um *socket*

A estrutura de dados de um socket inclui:

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
    uint32_t     s_addr;   /* address in network byte order */
};
```

Criação de um *socket*

Do lado do servidor:

```
bind (sd, address, length);
```

Associa o endereço do servidor ao descritor sd

```
listen (sd, qlength);
```

Aguarda uma requisição em sd (máximo de qlength)

```
nsd = accept (sd, address, addrlen);
```

Aceita uma conexão que será tratada no socket nsd

Criação de um *socket*

Do lado do cliente:

```
connect (sd, address, length);
```

Requisita a conexão com o servidor em address

Comunicação via *sockets*

Trocando mensagens:

```
count = send (sd, msg, length, flags);

count = recv (sd, buf, length, flags);
```

Comunicação via *sockets*

Outras alternativas envolvem:

Uso de datagramas com *sendto()* e *recvfrom()*

Uso de *read()* e *write()* depois que uma conexão já está estabelecida

Outras funções com *sockets*


```
shutdown (sd, node); /* fecha uma conexão */

close(); /* fecha (destrói) um socket */

getsockname (sd, name, length); /* recupera o
nome de um socket que tenha sido ligado pelo
bind */
```

Outras funções com *sockets*


```
getsockopt(sock, level, opt, opval, len); /*
  recupera os valores de opções de comunicação
  (opval e len) de sock operando no nível dado*/


setsockopt(sock, level, opt, opval, len); /*
atribui valores (opval e len) para as opções de
comunicação de um socket */
```

Exemplo cliente/servidor

Com o uso de *sockets* um sistema cliente/servidor deve ser composto por um par de programas: um com o código do cliente e outro com o servidor

O servidor é disparado e entra em *loop* a espera de conexões dos clientes

Os clientes devem conhecer o endereço do servidor na rede para poder requisitar uma conexão

Exemplo (processo servidor)


```
#include <stdio.h>
#include<stdlib.h>
#include <string.h>
#include <sys/socket.h> // define operações nos sockets
#include <netinet/in.h> // define constantes para internet
#include <netdb.h> // conversão de endereços IP
#include <arpa/inet.h> // define operações sobre internet

int main(int argc, char *argv[])
{int sock, connected, bytes_recv, true = 1, i, j;
 char send_data[1024] , recv_data[1024];
 struct sockaddr_in server_addr,  client_addr;
 int sin_size;
```

Exemplo (processo servidor)


```
// Cria socket e verifica se foi bem sucedido na operação
 if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
       perror("Erro na criação do Socket");
       exit(1);
    }
// Ajusta opções para o socket criado
  if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &true,sizeof(int))
== -1)
   {   perror("Erro em Setsockopt");
       exit(1);
    }
// Inicializa parâmetros para o socket. Note o endereço da porta (5000)
  server_addr.sin_family = AF_INET;
  server_addr.sin_port = htons(5000);  // host-endian to network-endian
  server_addr.sin_addr.s_addr = INADDR_ANY;
  bzero(&(server_addr.sin_zero),8);
```

Exemplo (processo servidor)


```
// Liga a estrutura do socket criado ao socket físico real
  if (bind(sock, (struct sockaddr *)&server_addr, sizeof(struct
sockaddr)) == -1) {
        perror("binding não foi possível");
        exit(1);
   }

// Define o tamanho da fila de entrada (5)
  if (listen(sock, 5) == -1) {
        perror("Erro na definição do tamanho da fila de entrada");
        exit(1);
   }

  printf("\nServidor TCP esperando por cliente na porta 5000\n");
  fflush(stdout);
```

Exemplo (processo servidor)


```
// Laço infinito de existência do servidor
  while(true)
  { sin_size = sizeof(struct sockaddr_in);

    connected = accept(sock, (struct sockaddr *)&client_addr,
    &sin_size);

    printf("\nConexão recebida (Cliente: %s , Porta: %d)\n",
    inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));

    fflush(stdout);
```

Exemplo (processo servidor)


```
// O laço a seguir é executado para todas conexões. O servidor
// envia texto para o cliente até que alguém resolva sair
    while(still_talking)
    { // Recebe a solicitação
      if ((bytes_recv = recv(connected, recv_data,1024,0)) <= 0)
          { printf("Conexao perdida\n");
            break;
          }
      recv_data[bytes_recv] = '\0';
// Copia a entrada invertida para ser a futura saída de dados
      j = strlen(recv_data)-2;
      for (i=0; i<=j; i++)
          send_data[i] = recv_data[j-i];
      send_data[i] = '\0';
```

Exemplo (processo servidor)


```
// Verifica a resposta, fechando o socket se receber sinal de saída (q/Q)
      if (strcmp(recv_data,"q")==0 || strcmp(recv_data,"Q")==0)
          { strcpy(send_data,"Encerrando conexao\n");
            send(connected, send_data, strlen(send_data), 0);
            printf("Conexao encerrada pelo cliente\n");
            close(connected);
            break;
          }
        else    //Senão envie o texto recebido invertido
          send(connected, send_data, strlen(send_data), 0);
      fflush(stdout);
    }   // final do laço da conexão (still_talking)
  }   // final do laço infinito do servidor
  close(sock);
  return 0;
}
```

Exemplo (processo cliente)


```
#include <stdio.h>
#include<stdlib.h>
#include <string.h>
#include <sys/socket.h> // define operações nos sockets
#include <netinet/in.h> // define constantes para internet
#include <netdb.h> // conversão de endereços IP

int main(int argc, char *argv[])
{int sock, bytes_recv;
 char send_data[1024],recv_data[1024];
 struct hostent *host;
 struct sockaddr_in server_addr;
```

Exemplo (processo cliente)


```
// Le linha de comando para ter o nome (IP) do servidor
  host = gethostbyname(argv[1]);
// Cria um socket
  if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
     perror("Erro na criação do Socket");
     exit(1);
  }
  fflush(stdout);
// Define informações sobre o socket do servidor
  server_addr.sin_family = AF_INET;
  server_addr.sin_port = htons(5000);
  server_addr.sin_addr = *((struct in_addr *)host->h_addr);
  bzero(&(server_addr.sin_zero),8);
```

Exemplo (processo cliente)


```
// Conecta com o servidor definido
  if (connect(sock, (struct sockaddr *)&server_addr,
                                    sizeof(struct sockaddr)) == -1)
    { perror("Erro na conexão");
      exit(1);
    }
// Repete enquanto a conexão for mantida
  while(1)  {   // Lê entrada do usuário
    printf ("\n TEXTO (q or Q to quit) : ");
    fgets(send_data, sizeof(send_data), stdin);
// Envia texto ou pedido de desconexão ao servidor
    send(sock, send_data, strlen(send_data), 0);
```

Exemplo (processo cliente)


```
// Recebe resposta do servidor
    bytes_recv=recv(sock,recv_data,1024,0);
    recv_data[bytes_recv] = '\0';
// Imprime a resposta ou fecha o socket se havia pedido
    if (strcmp(recv_data,"Encerrando conexao\n")!=0)
          printf("\n DADO RECEBIDO = %s " , recv_data);
      else
        { printf("\n %s " , recv_data);
          close(sock);
          break;
        }
  }   // Final de while(1)
  return 0;
}
```

Exemplo cliente/servidor

Para compilar esses programas:

```
$ gcc -o client client.c
$ gcc -o server server.c
```

Para executar:

```
$ ./server                          (no servidor)
$./client nome_host_servidor        (no cliente)
```

Conclusões

A programação de processos paralelos usando bibliotecas básicas é possível e não muito difícil de se obter

O problema com threads é que funcionam quando restritos a uma máquina

O problema com sockets é a razoável complexidade de sua programação

Conclusões

Para contornar em parte tais problemas surgem as chamadas bibliotecas para programação paralela, como PVM, MPI, OpenMP

Veremos OpenMP e MPI em breve

A desvantagem dessas bibliotecas é que os procedimentos de comunicação não são otimizados para aplicações específicas


GUESS WHAT
I QUIT THIS STUPID JOB
HAHA IDIOTS DID YA
SERIOUSLY THINK
I NEEDED
THIS MANY SIGNS
TODAY?? DOH
THERE'S LIKE
3 TOWNS
AROUND HERE
LOL MORONS
HEY LOOK I BROUGHT
SYMBOLS TOO
HEY MR. JENKINS
YOU CAN SHOVE
A
IN YOUR
LOL PWNED
OK FUCKERS
GOOD LUCK
WITH
THIS STUPID JOB
OH BTW PPL
LITTLETON IS
THIS WAY
I THINK
PMSLweb.com