UNIVERSIDADE ESTADUAL DE CAMPINAS

LABORATÓRIO DE REDES - MC823

Tarefa 4

Miguel Francisco Alves de Mattos Gaiowski

RA 076116

Guillaume Massé

RA 107888

Prof. Paulo Lício de Geus

Campinas, 8 de novembro de 2010

Sumário

1 Introdução

Esta tarefa pede que o cliente da tarefa 2 seja modificado para não usar mais fork() e sim select().

Além disso, o servidor deve ser transformado em um daemon.

2 Objetivos

Verificar que um cliente multiplexado usando select() é tão bom quanto um que use vários processos para cuidar de cada tarefa.

Transformar o servidor em um daemon.

3 Desenvolvimento

Usando o tutorial do Beej [1], aprendemos sobre o funcionamento do método select(). Fizemos as modificações necessárias para que o cliente deixasse de usar fork() e fosse só um processo que multiplexava os canais de comunicação.

No servidor, usando a documentação encontrada na internet, conseguimos fazer com que o servidor fosse um daemon. Ou seja, continuasse rodando mesmo que o usuário fizesse logout. Para isso vários passos devem ser seguidos. Entre eles, desacoplar de terminais, escrever em log, fazer fork e matar o processo pai, etc.

4 Dificuldades

A principal dificuldade desta tarefa foi aprender o funcionamento do método select(). Graças ao tutorial do Beej [1] conseguimos entender o funcionamento e implementar o que foi pedido.

Além disso, outro problema que acontece é quanto a bufferização. É necessário que usemos a função setvbuf() para modificar o modo de buffer.

5 Experimentos

Os programas foram testados com uma máquina em casa e outra do IC-3. O RTT medido entre as máquinas foi de 150 milisegundos.

Primeiramente repetimos os testes da tarefa 2 neste ambiente, para garantir que o RTT fosse o mesmo.

Usando o cliente e o servidor da tarefa 2, com um processo que envia e outro que recebe os dados, obtivemos um resultado de 7.519926 segundos para enviar o arquivo etcservices.

Repetimos o experimento usando os programas da tarefa 4. Com o servidor daemon e o cliente usando select obtivemos um tempo de 7.377163 segundos.

Com isso verificamos que usando select o desempenho é tão bom quanto usando multiplos processos. Na verdade o resultado foi até um pouco melhor.

6 Conclusões

Notamos que o desempenho de um cliente usando select chega a ser mais rápido que um que usa multiplos processos. A pequena diferença de tempo pode ser devida ao servidor ser um daemon, e por isso estar desaclopado de terminais.

Aprendemos que select() é uma solução viável para multiplexar comunicação de um servidor com vários clientes ou até mesmo para um simples cliente de echo.

7 Bibliografia

Referências

[1] Brian “Beej Jorgensen” Hall. *Beej’s Guide to Network Programming*. 2009.

8 Anexos

Os códigos dos programas seguem anexos.

```c
/* Lab 4 - client.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define PORT 40001    /* the port client will be connecting to */
#define MAXDATASIZE 1000 /* max number of bytes we can get at once */

int main(int argc, char *argv[]) {
  int sockfd;
  char inbuf[MAXDATASIZE], outbuf[MAXDATASIZE];
  struct hostent *he;
  struct sockaddr_in their_addr; /* connector's address information */
  int numInLines = 0, numOutLines = 0, maxlinesize = 1, totalInChars = 0, totalOutChars = 0;
  FILE *rsock, *wsock;

  /* Variáveis para o select() */
  fd_set master;
  fd_set temp;

  int done = 0;
  int fdmax;

  if (argc != 2) {
    fprintf(stderr,"usage: client hostname\n");
    exit(1);
  }

  /* Limpa os conjuntos de file descriptors master e read_fds */
  FD_ZERO(&master);
  FD_ZERO(&temp);

  if ((he=gethostbyname(argv[1])) == NULL) {  /* get the host info */
    perror("gethostbyname");
    exit(1);
  }
  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
  }

  their_addr.sin_family = AF_INET;         /* host byte order */
  their_addr.sin_port = htons(PORT);     /* short, network byte order */
  their_addr.sin_addr = *((struct in_addr *)he->h_addr);
  bzero(&(their_addr.sin_zero), 8);        /* zero the rest of the struct */

  if (connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct sockaddr)) == -1) {
    perror("connect");
    exit(1);
  }
  if ((rsock = fdopen(sockfd, "r")) == NULL) {
    perror("fdopen");
    exit(1);
  }
  if ((wsock = fdopen(sockfd, "w")) == NULL) {
    perror("fdopen");
    exit(1);
  }

  /* Seta modo de buffer */
  setvbuf(wsock, NULL, _IOLBF, 0);
```

```
  setlinebuf(rsock);
  setlinebuf(stdout);
  setlinebuf(stdin);

  FD_SET(sockfd, &master);
  FD_SET(STDIN_FILENO, &master);
  fdmax = sockfd;

  if (fgets(inbuf, MAXDATASIZE, rsock) == NULL) {
    perror("fgets");
    exit(1);
  }
  fflush(rsock);
  fprintf(stderr, "Received: %s", inbuf);
  struct timeval start;
  gettimeofday( &start, NULL );

  while (1) {
    temp = master;              /* Salvando grupo de fds. */
    /* Checando se há algo no stdin ou no sockfd */
    if (select(fdmax+1, &temp, NULL, NULL, NULL) < 0) {
      perror("select");
      exit(1);
    }
    if (FD_ISSET(sockfd, &temp)) { /* Temos algo pra ler do servidor */
      if (fgets(inbuf, MAXDATASIZE, rsock) == NULL) { /* Já recebemos tudo */
        break;
      }
      fflush(rsock);
      numInLines++;
      totalInChars += strlen(inbuf);
      printf("%s", inbuf);
    }
    if (FD_ISSET(STDIN_FILENO, &temp)) { /* Lendo da entrada padrão e mandar pro servidor */
      if (fgets(outbuf, MAXDATASIZE, stdin) != NULL) {
        if (fputs(outbuf, wsock) == EOF) {
          perror("send");
          exit(1);
        }
        numOutLines++;
        maxlinesize = maxlinesize > strlen(outbuf) ? maxlinesize : strlen(outbuf);
        totalOutChars += strlen(outbuf);
      }
      else {
        if (!done) {
          shutdown(sockfd, SHUT_WR); /* Já mandamos tudo. */
          fprintf(stderr, "Numero de linhas enviadas: %d\n", numOutLines);
          fprintf(stderr, "Numero de caracteres na maior linha: %d\n", maxlinesize - 1);
          fprintf(stderr, "Total de caracteres enviados: %d\n", totalOutChars);
          done = 1;
        }
      }
    }
  }

  struct timeval end;
  gettimeofday( &end, NULL );
  double time_elapsed = ( (double)( end.tv_usec - start.tv_usec ) ) /
    1.0e+6 + ( (double)( end.tv_sec - start.tv_sec ) );
  fprintf( stderr, "Tempo total de transferencia: %lf s\n", time_elapsed );
  fprintf(stderr, "Numero de linhas recebidas: %d\n", numInLines);
  fprintf(stderr, "Total de caracteres recebidos: %d\n", totalInChars);

  fclose(rsock);
  close(sockfd);
  return 0;

}
```

```c
/* Servidor daemon de echo */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>

#define RUNNING_DIR     "/tmp"
#define LOCK_FILE       "EchoDaemon.lock"
#define LOG_FILE        "EchoDaemon.log"

void log_message( char* message )
{
  FILE* logfile = fopen( LOG_FILE, "a");

  if( !logfile ) return;

  fprintf( logfile, "%s\n", message );

  fclose( logfile );
}

void log_message_ip( char* message, char* ip )
{
  FILE* logfile = fopen( LOG_FILE, "a");

  if( !logfile ) return;

  fprintf( logfile, "[%s]: %s\n", ip, message );

  fclose( logfile );
}

void log_message_stats( int lines, int chars )
{
  FILE* logfile = fopen( LOG_FILE, "a");

  if( !logfile ) return;

  fprintf( logfile, "Stats >> Lines: %d  Chars: %d\n", lines, chars );

  fclose( logfile );
}


void signal_handler( int sig )
{
  switch(sig)
    {

    case SIGHUP:
      log_message( "hangup signal catched" );
      break;

    case SIGTERM:
      log_message( "terminate signal catched" );
      exit(0);
      break;

    }
}
```

```
void daemonize( int method )
{

  int i,lfp;
  char str[10];

  /* already a daemon */
  if( getppid() == 1 ) return;

  i = fork();
  if ( i < 0 ) exit( 1 ); // fork error
  if ( i > 0 ) exit( 0 ); // parent exits

  /* child (daemon) continues */

  /* obtain a new process group and be it's leader */
  setsid( );

  /* Avoid that deamon open a terminal device automaticly */
  if( method == 1 )
    {
      // refork
      i = fork();
      if ( i < 0 ) exit( 1 ); // fork error
      if ( i > 0 ) exit( 0 ); // parent exits
    }
  else if( method == 2 )
    {

      // Method 2
      /*  close all descriptors */
      for ( i = getdtablesize( ); i >= 0; --i )
        close( i );

      /*  handle standart I/O */
      i = open( "/dev/null", O_RDWR );
      dup( i );
      dup( i );
    }

  /* change running directory to one not mounted by the system */
  chdir( RUNNING_DIR );

  /* set newly created file permissions */
  umask( 027 );


  /* Create a lock file to make sute only one instance is running */
  lfp = open( LOCK_FILE, O_RDWR | O_CREAT, 0640 );
  if ( lfp < 0 ) exit( 1 ); // can not open
  if ( lockf( lfp, F_TLOCK, 0 ) < 0 ) exit( 0 ); // can not lock

  /* first instance continues */

  /* record pid to lockfile */
  sprintf( str, "%d\n", getpid( ) );
  write( lfp, str, strlen( str ) );

  /* ignore child */
  signal( SIGCHLD, SIG_IGN );

  /* ignore tty signals */
  signal( SIGTSTP, SIG_IGN );
  signal( SIGTTOU, SIG_IGN );
  signal( SIGTTIN, SIG_IGN );

  /* catch hangup signal */
  signal( SIGHUP, signal_handler );

  /* catch kill signal */
  signal( SIGTERM, signal_handler );
}
```

```
/*
  UNIX Daemon Server Programming Sample Program
  Levent Karakas <levent at mektup dot at> May 2001
  Taken from: http://www.enderunix.org/documents/eng/daemon.php
*/

/* End Deamonize */


/* server_echo.c - Servidor simples */


#define MYPORT 40001    /* the port users will be connecting to */
#define BACKLOG 10     /* how many pending connections queue will hold */
#define BUFF_SIZE 1000

int main( int argc, char** argv ) {

  if( argc != 2 )
    {
      printf("Method 1 or Method 2 ?\n");
      exit(1);
    }

  daemonize( atoi(argv[1]) );

  int sockfd, new_fd;  /* listen on sock_fd, new connection on new_fd */
  struct sockaddr_in my_addr;    /* my address information */
  struct sockaddr_in their_addr; /* connector's address information */
  int sin_size;
  char buffer[BUFF_SIZE];
  int numInLines = 0, totalInChars = 0;
  FILE *rsock, *wsock;

  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("socket");
    exit(1);
  }

  my_addr.sin_family = AF_INET;         /* host byte order */
  my_addr.sin_port = htons(MYPORT);     /* short, network byte order */
  my_addr.sin_addr.s_addr = INADDR_ANY; /* automatically fill with my IP */
  bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */

  if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1) {
    perror("bind");
    exit(1);
  }
  if (listen(sockfd, BACKLOG) == -1) {
    perror("listen");
    exit(1);
  }

  log_message("Deamon ready");

  while(1) {  /* main accept() loop */
    sin_size = sizeof(struct sockaddr_in);
    if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, (socklen_t *)&sin_size)) == -1)
{
      perror("accept");
      continue;
    }
    numInLines = 0; totalInChars = 0;

    log_message_ip( "Client Connect", inet_ntoa( their_addr.sin_addr ) );

    if ((rsock = fdopen(new_fd, "r")) == NULL) {
      perror("fdopen");
      exit(1);
    }
    if ((wsock = fdopen(new_fd, "w")) == NULL) {
```

```
      perror("fdopen");
      exit(1);
    }

    char SendText[] = "Conectado, envie uma mensagem que eu devolvo.\n";
    if (fputs(SendText, wsock) == EOF) {
      perror("send");
      exit(1);
    }
    fflush(wsock);
    while ( fgets(buffer, BUFF_SIZE, rsock) != NULL ) { // recebe msg do cliente
      fflush(rsock);
      numInLines++;
      totalInChars += strlen(buffer);
      fputs(buffer, wsock); // devolve a mesma coisa
      fflush(wsock);
    }

    log_message_stats( numInLines, totalInChars );
    log_message_ip( "Client Disconnect", inet_ntoa( their_addr.sin_addr ) );

    fclose(rsock);
    fclose(wsock);
    close(new_fd);
  }
  return 0;
}
```