CSCI 330
The UNIX System

TCP Server Programming

Unit Overview

- TCP server programming
- server fork to process client request
- server fork/exec to run program remotely
- directory I/O

TCP programming

- provides multiple endpoints on a single node: port
- common abstraction: socket

- socket is end-point of communication link
  - identified as IP address + port number
  - can receive data
  - can send data

Socket system calls

server

client

Primitive	Meaning
socket	Create a new communication endpoint
bind	Attach a local address to a socket
listen	Announce willingness to accept connections
accept	Block caller until a connection request arrives
connect	Actively attempt to establish a connection
write	Send(write) some data over the connection
read	Receive(read) some data over the connection
close	Release the connection

TCP Server: basic logic

```
- while (true) {
-     connSock = accept(sock, ...);


-     // process client's request
-      //    via connSock


-     close(connSock);
- }
```

TCP Server fork

- server starts loop
  - waits on accept for connection from client
- after accept, server forks to service client request
  - accept returns dedicated connection socket
- parent process will continue to wait on accept
  - parent closes dedicated connection socket
- child process serves client request
  - communicates with client via dedicated connection socket

TCP Server fork: logic

```
- while (true) {
-     connSock = accept(sock, ...);
-     if (fork()) {     // parent process
-         close(connSock);
-     } else {          // child process
-         // process client's request via connSock
-         close(connSock);
-         exit(0);
-     }
- }
```

TCP server/fork illustration

TCPServerFork.cxx - /home/student/Desktop/Unit 19 Programs - Geany

File Edit Search View Document Project Build Tools Help

TCPServerFork.cxx

```
65      // Run until cancelled
66      while (true) {
67          connSock=accept(sock, (struct sockaddr *) &echoclient, &clientlen);
68          if (connSock < 0) {
69              perror("accept failed");
70              exit(EXIT_FAILURE);
71          }
72          // fork
73          if (fork()) {   // parent process
74              close(connSock);
75          } else {            // child process
76              // read a message from the client
77              if ((received = read(connSock, buffer, 256)) < 0) {
78                  perror("Failed to receive message");
79                  exit(EXIT_FAILURE);
80              }
81              buffer[received] = '\0';
82              cerr << "Client(" << inet_ntoa(echoclient.sin_addr) << ") sent: " << buffer << endl;
83              // write the message back to client
84              if (write(connSock, buffer, received) != received) {
85                  perror("Mismatch in number of echo'd bytes");
86                  exit(EXIT_FAILURE);
87              }
88              close(connSock);
89              exit(EXIT_SUCCESS);
90          }
91      }
92      close(sock);
93      return 0;
```

line: 84 / 95 col: 64 sel: 0 INS TAB mode: Unix (LF) encoding: UTF-8 filetype: C++ scope: main

TCP server/fork detail

```
while (true) {
   connSock=accept(sock, (struct sockaddr *)&echoclient,&clientlen);
   if (connSock < 0) {perror("accept failed"); exit(EXIT_FAILURE);}
   // fork into 2 processes
   if (fork()) {        // parent process
    close(connSock);
   } else {             // child process
    // process the client's request   ...
    close(connSock);
    exit(EXIT_SUCCESS);
   }
}
```

TCP server + multiple clients

TCPServerForkProcess.cxx - /home/student/Desktop/Unit 19 Programs - Geany

File Edit Search View Document Project Build Tools Help

TCPServerForkProcess.cxx

```
24
25  void processClientRequest( int connSock) {
26      int received;
27      char buffer[256];
28
29      while (true) {
30          // read a message from the client
31          if ((received = read(connSock, buffer, 256)) <= 0) {
32              perror("Failed to receive message");
33              exit(EXIT_FAILURE);
34          }
35          buffer[received] = '\0';
36          cerr << "Client sent: " << buffer << endl;
37          if (!strncmp(buffer, "quit", 4)) break;
38          // write the message back to client
39          if (write(connSock, buffer, received) != received) {
40              perror("Mismatch in number of echo'd bytes");
41              exit(EXIT_FAILURE);
42          }
43      }
44      cerr << "done with this client\n";
45      close(connSock);
46      exit(EXIT_SUCCESS);
47  }
```

line: 25 / 104 col: 0 sel: 0 INS TAB mode: Unix (LF) encoding: UTF-8 filetype: C++ sco...

TCP server/fork detail

```
while (true) {
   connSock=accept(sock, (struct sockaddr *)&echoclient,&clientlen);
   if (connSock < 0) {perror("accept failed"); exit(EXIT_FAILURE);}
   // fork into 2 processes
   if (fork()) {        // parent process
    close(connSock);
   } else {             // child process
    // process the client's request   ...
    processClientRequest(connSock);
   }
}
```

Server fork, exec & dup

- after accept, server forks to service client request
  - parent process will continue to wait on accept
- child process duplicates socket descriptor(s) into standard I/O
- child process uses exec to run requested program

Server fork/exec: child dups output

- child process dups socket descriptor into standard out
- child process uses exec to run requested program

TCPServerForkExecOut.cxx - /home/student/Desktop/Unit 19 Programs - Geany

File Edit Search View Document Project Build Tools Help

TCPServerForkExecOut.cxx

```
50      // duplicate socket descriptor into standard output
51      if (dup2(connSock, 1) < 0) {
52          perror("socket dup2");
53          exit(EXIT_FAILURE);
54      }
55      // prepare argv array of strings
56      char *argv[] = { (char*)0, (char*)0, (char*)0, (char*)0, (char*)0, (char*)0, (char*)0 };
57      // parse command into words
58      for (char *p = strtok(command, " "); p; p = strtok(NULL, " ")) {
59          argv[count++] = p;
60      }
61      // run command and its arguments via execvp
62      if (execvp(argv[0], argv) < 0) {
63          perror("exec in child after fork");
64          exit(EXIT_FAILURE);
65      }
```

line: 63 / 123 col: 43 sel: 0 INS TAB mode: Unix (LF) encoding: UTF-8 filetype: C++ scope: processClientRequest

TCP server: dup2/exec detail

```
int received, count=0;
char command[256];

// read a message from the client
if ((received = read(connSock, command, 256)) <= 0) {
    perror("Failed to receive message");
    exit(EXIT_FAILURE);
}
command[received] = '\0';    // ensure string is terminated
chomp(command);          // remove trailing \r and \n
```

TCP server: chomp detail

```
void chomp(char *s) {


 for (char *p = s + strlen(s)-1;   // start at the end of string
     *p == '\r' || *p == '\n';     // while there is a trailing \r or \n
      p--)                         // check next character from back
   *p = '\0';                      // change \r or \n to \0
}
```

- removes trailing newline or carriage return from string s

TCP server: dup2 detail

```
// duplicate socket descriptor into standard output
if (dup2(connSock, 1) < 0) {
   perror("socket dup2");
   exit(EXIT_FAILURE);
}
```

TCP server: execvp detail

```
// prepare argv array of strings

char *argv[] = { (char*)0, (char*)0, (char*)0, (char*)0, (char*)0,
    (char*)0, (char*)0 };

// parse command into words

for (char *p = strtok(command, " "); p; p = strtok(NULL, " "))

    argv[count++] = p;

// run command and its arguments via execvp

if (execvp(argv[0], argv) < 0) {

    perror("exec in child after fork");

    exit(EXIT_FAILURE);

}
```

Server fork/exec: child dups input

- child process dups socket descriptor into standard in
- child process uses exec to run requested program

TCPServerForkExecIn.cxx - /home/student/Desktop/Unit 19 Programs - Geany

File Edit Search View Document Project Build Tools Help

TCPServerForkExecIn.cxx

```
50      // duplicate socket descriptor into standard input
51      if (dup2(connSock, 0) < 0) {
52          perror("socket dup2");
53          exit(EXIT_FAILURE);
54      }
55      // prepare argv array of strings
56      char *argv[] = { (char*)0, (char*)0, (char*)0, (char*)0, (char*)0, (char*)0, (char*)0 };
57      // parse command into words
58      for (char *p = strtok(command, " "); p; p = strtok(NULL, " "))
59          argv[count++] = p;
60      // run command and its arguments via execvp
61      if (execvp(argv[0], argv) < 0) {
62          perror("exec in child after fork");
63          exit(EXIT_FAILURE);
64      }
```

line: 54 / 122 col: 5 sel: 0 INS TAB mode: Unix (LF) encoding: UTF-8 filetype: C++ scope: processClientRequest

Server fork/exec: child dups I/O

- child process dups socket descriptor into standard in & out

*TCPServerForkExecInOut.cxx - /home/student/Desktop/Unit 19 Programs - Geany

File Edit Search View Document Project Build Tools Help

TCPServerForkExecInOut.cxx

```
50      // duplicate socket descriptor into standard input
51      if (dup2(connSock, 0) < 0) {
52          perror("socket in dup2");
53          exit(EXIT_FAILURE);
54      }
55      // duplicate socket descriptor into standard output
56      if (dup2(connSock, 1) < 0) {
57          perror("socket out dup2");
58          exit(EXIT_FAILURE);
59      }
60      close(connSock);
61      // prepare argv array of strings
62      char *argv[] = { (char*)0, (char*)0, (char*)0, (char*)0, (char*)0, (char*)0, (char*)0 };
63      // parse command into words
64      for (char *p = strtok(command, " "); p; p = strtok(NULL, " "))
65          argv[count++] = p;
66      // run command and its arguments via execvp
67      if (execvp(argv[0], argv) < 0) {
68          perror("exec in child after fork");
69          exit(EXIT_FAILURE);
70      }
```

line: 57 / 128 col: 34 sel: 0 INS TAB MOD mode: Unix (LF) encoding: UTF-8 filetype: C++ scope: processClientReq...

Directory Input/Output

- current directory: chdir, getcwd
- directory I/O functions: opendir, readdir
  - directory I/O types: DIR, struct dirent
- directory I/O system call: mkdir

Directory I/O function: opendir

Terminal

File Edit View Search Terminal Help

```
OPENDIR(3)                Linux Programmer's Manual                OPENDIR(3)

NAME
       opendir, fdopendir - open a directory

SYNOPSIS
       #include <sys/types.h>
       #include <dirent.h>

       DIR *opendir(const char *name);
       DIR *fdopendir(int fd);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       fdopendir():
           Since glibc 2.10:
               _XOPEN_SOURCE >= 700 || _POSIX_C_SOURCE >= 200809L
           Before glibc 2.10:
               _GNU_SOURCE

DESCRIPTION
       The  opendir()  function opens a directory stream corresponding to the
       directory name, and returns a pointer to the  directory  stream.   The
 Manual page opendir(3) line 1 (press h for help or q to quit)
```

Directory I/O function: opendir

```
DIR *opendir(const char *name)
```

- opens directory **`name`** as a stream
- returns **`DIR`** pointer for **`readdir`** function
- returns NULL on error, and **`errno`** is:

  `ENOENT` directory does not exist

  `ENOTDIR` name is not a directory

Directory I/O function: readdir

```
Terminal
File  Edit  View  Search  Terminal  Help
READDIR(3)                Linux Programmer's Manual                READDIR(3)

NAME
       readdir, readdir_r - read a directory

SYNOPSIS
       #include <dirent.h>

       struct dirent *readdir(DIR *dirp);

       int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       readdir_r():
           _POSIX_C_SOURCE >= 1 || _XOPEN_SOURCE || _BSD_SOURCE ||
           _SVID_SOURCE || _POSIX_SOURCE

DESCRIPTION
       The readdir() function returns a pointer to a dirent structure  repre-
       senting the next directory entry in the directory stream pointed to by
       dirp.  It returns NULL on reaching the end of the directory stream  or
       if an error occurred.
 Manual page readdir(3) line 1 (press h for help or q to quit)
```

Directory I/O function: readdir

```
struct dirent *readdir(DIR *dirp)
```

- returns a pointer to a **`dirent`** structure representing the next directory entry in directory **`dirp`**
- returns NULL on reaching end of directory

  or if an error occurred

dirent structure

```
struct dirent {
  ino_t          d_ino;       /* inode number */
  off_t          d_off;       /* offset to the next dirent */
  unsigned short d_reclen;    /* length of this record */
  unsigned char  d_type;      /* type of file */
  char           d_name[256]; /* filename */
};
```

Illustration: readDir.cxx

readDir.cxx - /home/student/Desktop/Unit 19 Programs - Geany

File Edit Search View Document Project Build Tools Help

readDir.cxx

```
17  int main(int argc, char* argv[]) {
18
19      if (argc != 2) {
20          cerr << "USAGE: readDir pathname\n";
21          exit(EXIT_FAILURE);
22      }
23
24      DIR *dirp;
25      struct dirent *dirEntry;
26
27      // open directory
28      dirp = opendir(argv[1]);
29      if (dirp == 0) {
30          perror(argv[1]);
31          exit(EXIT_FAILURE);
32      }
33
34      while ((dirEntry = readdir(dirp)) != NULL) {
35
36          cout << dirEntry->d_name << endl;
37
38      }
39
40      closedir(dirp);
41      return 0;
42  }
```

line: 12 / 43 col: 18 sel: 0 INS TAB mode: Win (CRLF) encoding: UTF-8 filetype: C++ scope: unknown

Directory I/O detail

```
// open directory
dirp = opendir(argv[1]);
if (dirp == 0) {
   perror(argv[1]);
   exit(EXIT_FAILURE);
}
while ((dirEntry = readdir(dirp)) != NULL) {
   cout << dirEntry->d_name << endl;
}
closedir(dirp);
```

Directory System call: mkdir

```
Terminal
File  Edit  View  Search  Terminal  Help
MKDIR(2)                Linux Programmer's Manual                MKDIR(2)

NAME
       mkdir - create a directory

SYNOPSIS
       #include <sys/stat.h>
       #include <sys/types.h>

       int mkdir(const char *pathname, mode_t mode);

DESCRIPTION
       mkdir() attempts to create a directory named pathname.

       The argument mode specifies the permissions to use.  It is modified by
       the process's umask in the usual way: the permissions of  the  created
       directory  are (mode & ~umask & 0777).  Other mode bits of the created
       directory depend on the operating system.  For Linux, see below.

       The newly created directory will be owned by the effective user ID  of
       the  process.  If the directory containing the file has the set-group-
       ID bit set, or if the file system is mounted with BSD group  semantics
       (mount  -o  bsdgroups or, synonymously mount -o grpid), the new direc-
 Manual page mkdir(2) line 1 (press h for help or q to quit)
```

Illustration: makeDir.cxx

makeDir.cxx - /home/student/Desktop/Unit 19 Programs - Geany

File Edit Search View Document Project Build Tools Help

makeDir.cxx readDir.cxx

```
19  int main(int argc, char* argv[]) {
20      const char *dirName = "newDir";
21      if (argc == 2)
22          dirName = argv[1];
23
24      // make directory
25      if (mkdir(dirName, 0777) < 0) {
26          perror(dirName);
27          exit(EXIT_FAILURE);
28      }
29
30      DIR *dirp;
31      struct dirent *dirEntry;
32
33      // open directory
34      dirp = opendir(argv[1]);
35      if (dirp == 0) {
36          perror(argv[1]);
37          exit(EXIT_FAILURE);
38      }
39
40      while ((dirEntry = readdir(dirp)) != NULL)
41          cout << dirEntry->d_name << endl;
42
43      closedir(dirp);
44      return 0;
45  }
```

line: 26 / 46 col: 24 sel: 0 INS TAB mode: Win (CRLF) encoding: UTF-8 filetype: C++ scope: main

Summary

- TCP server programming
- server fork to process client request
- server fork/exec to run program remotely
- directory I/O