COMP20200 Unix Programming

Lecture 16

Ravi Reddy Manumachu (ravi.manumachu@ucd.ie)

School of Computer Science, University College Dublin, Ireland

03/04/2022

Lecture overview

- Develop a TCP socket client.
- Develop a parallel TCP socket server using processes.

TCP socket client: System call sequence diagram

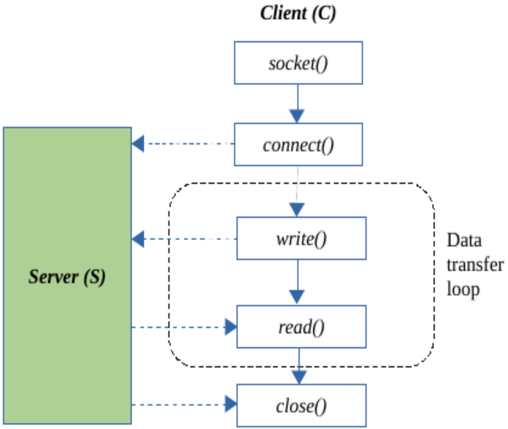

Figure: System calls for TCP stream sockets in client.

Client code - socket creation

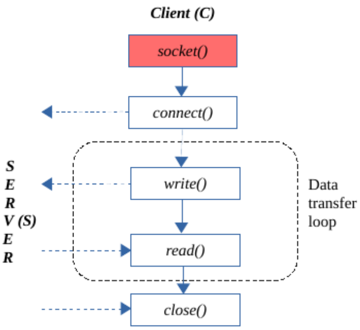

Figure: Client creates a new socket using the socket() call.

```
int cfd = socket(AF_INET,
    SOCK_STREAM, 0);
if (cfd == -1) {
    fprintf(stderr, "socket()
        error.\n");
    exit(-1);
}
```

- On success, *cfd* contains the newly created socket.
- This file descriptor will be used for connection as well as data transfer.

Client code - connect to server

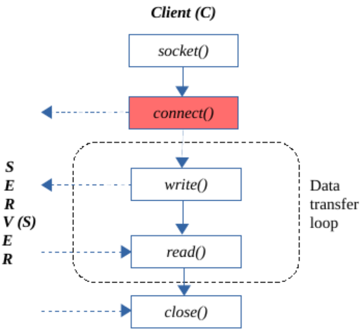

Figure: Client establishes a connection with the server using connect() call.

```
struct sockaddr_in serverAddress;
memset(&serverAddress, 0, sizeof(struct
      sockaddr_in));
serverAddress.sin_family = AF_INET;
serverAddress.sin_addr.s_addr = inet_addr(argv
      [1]);
serverAddress.sin_port = htons(atoi(argv[2]));

int rc = connect(cfd, (struct sockaddr *)&
      serverAddress, sizeof(struct sockaddr));
if (rc == -1) {
    fprintf(stderr, "connect() error, errno %d.\n"
         , errno);
    exit(-1);
}
```

- Since we are creating IPv4 socket address, we fill **struct sockaddr_in**.
- The server address to connect are provided by the first and second arguments.

Client code: data transfer loop

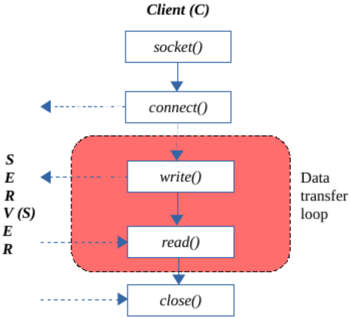

Figure: Client communicating with the server in a data transfer loop.

- Same logic for read and write as discussed for the server.
- Partial transfers can occur when performing I/O on stream sockets.
- The *read()* and *write()* may be interrupted by a signal-handler.

Client code: Sending a message

```
1 size_t totWritten;
2 for (totWritten = 0; totWritten < BUFSIZE; ) {
3     ssize_t numWritten = write(cfd, buf + totWritten, BUFSIZE - totWritten);
4     if (numWritten <= 0) {
5         if (numWritten == -1 && errno == EINTR)
6             continue;
7         else {
8             fprintf(stderr, "Write error.\n");
9             exit(EXIT_FAILURE);
10        }
11    }
12    totWritten += numWritten;
13 }
```

- Line 3: **(buf + totWritten)** advances the buffer **buf** by **totWritten** bytes.
- It places the pointer at **totWritten** position from the start;
- This is pointer arithmetic in C;
- Line 5: **(errno == EINTR)** means the write is interrupted and must be restarted manually.

Client code: Receiving a message

```c
1  size_t totRead;
2  for (totRead = 0; totRead < BUFSIZE; ) {
3      ssize_t numRead = read(cfd, buf + totRead, BUFSIZE - totRead);
4      if (numRead == 0)
5          break;
6      if (numRead == -1) {
7          if (errno == EINTR)
8              continue;
9          else
10             fprintf(stderr, "Read error.\n");
11     }
12     totRead += numRead;
13 }
```

- Line 3: **totRead = 0** for the first call of *read()*.
- *buf* is advanced by *totRead* places in each iteration since *totRead* places have been filled.
- Loop terminates when all the BUFSIZE bytes have been read.

Client code: Termination

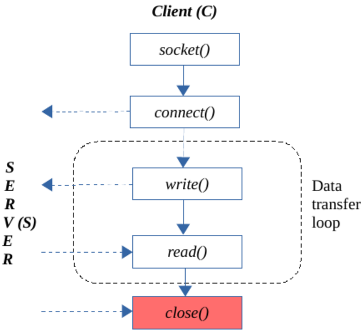

Figure: Client closes the connection to the server.

```
if (close(cfd) == -1) /* Close connection */
{
    fprintf(stderr, "close error.\n");
    exit(EXIT_FAILURE);
}
```

- Close the connection established by the client using the *close()* system call.

Parallel TCP Socket Server using Processes

Parallel TCP socket servers

- Disadvantages of the iterative server:
  - A client request that takes significant processing time can block other clients if the server is not parallel.
  - Poor utilization of the modern multicore computing platforms.
- Parallel servers improve utilization and the server throughput (Number of client requests serviced per second).

Parallel TCP socket servers

- We look at two simple patterns for a parallel server.
  - The server creates a new child process for each new client. The child processes operate independently (and simultaneously handle clients).
  - The server creates a new thread for each new client. The threads run independently (and simultaneously handle clients).
- However, these design patterns are not efficient in real-life production systems.
- Creating a new child or new thread to serve each client can be expensive. Imagine thousands of clients connecting at a time.

Using server pool or thread pool

Common real-life design patterns include:

- Using a server pool:
  - The server creates a fixed number of child processes or threads (pool) on startup.
  - Each child or thread in the pool handles one client request at a time.
- Using a cluster of homogeneous servers:
  - Create a cluster of iterative or multithreaded servers.
  - A single load-balancing server routes the client requests to one of the servers in the cluster.

Parallel server using processes

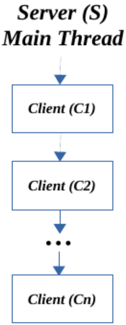

Figure: Iterative server.

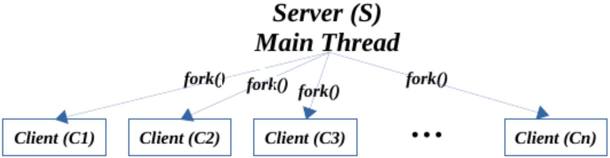

Figure: Parallel server using processes.

Parallel server using processes

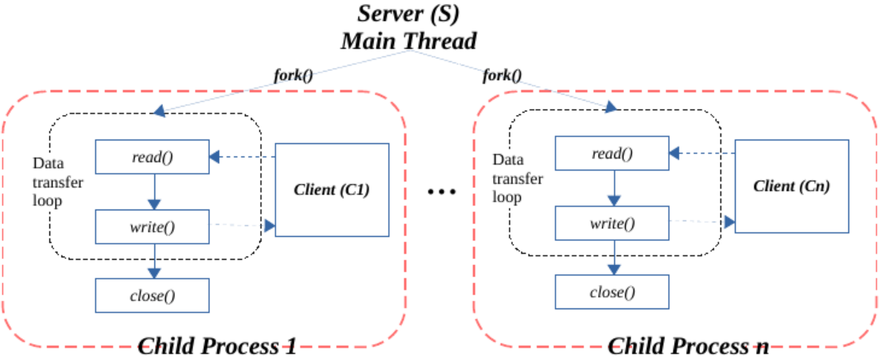

Figure: Parallel server using processes.

- The *for loop* in the server will involve creation of processes to handle clients parallely.

Parallel server: SIGCHLD signal handler

The server code has some essential differences from the iterative case.

- The parent process in the server creates a child process for each client connection.
- Therefore, we must ensure that the parent process in the server deals with the termination of the child processes appropriately.
- We do this using a SIGCHLD signal handler.

Parallel server: Zombies and Orphans

- If SIGCHLD handler is not setup, the children become zombies after they die.
- If the server exits before the children, the children become orphans and are adopted by *init* and removed from the system.
- *init* is the parent of all processes (process ID 1).

Parallel Server: sigaction to handle SIGCHLD

```
int
main(int argc, char *argv[])
{
  struct sigaction sa;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = SA_RESTART;
  sa.sa_handler = reapChild;
  if (sigaction(SIGCHLD, &sa, NULL) == -1)
      exit(EXIT_FAILURE);
/* Cont'd */
```

- **Revisit Lecture 11 on signals how to setup a signal handler.**
- *sigaction()* call sets a handler *reapChild* for SIGCHLD signal.
- SIGCHLD is generated by the kernel for a parent process when one of its children terminates.

Parallel Server: SA_RESTART flag

```c
int
main(int argc, char *argv[])
{
  struct sigaction sa;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = SA_RESTART;
  sa.sa_handler = reapChild;
  if (sigaction(SIGCHLD, &sa, NULL) == -1)
      exit(EXIT_FAILURE);
/* Cont'd */
```

- The flag SA_RESTART allows a blocked system call that is interrupted to be restarted.
- For example, a *read()* call that is blocked and interrupted due to a signal.

Parallel Server: SA_RESTART flag

- When you specify *SA_RESTART*, interrupted system calls are automatically restarted by the kernel.
- What this means is that you do not have to check for EINTR error code.
- You can use this flag for the following calls: *waitpid(), read(), write(), send(), recv()*

Parallel server: Signal Handler

```
static void
reapChild(int sig)
{
    int savedErrno = errno;
    while (waitpid(-1, NULL, WNOHANG) > 0)
        continue;
    errno = savedErrno;
}
```

- *The while loop essentially reaps all the children.*

waitpid() system call

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- *waitpid()* waits for the child process *pid* provided in the first argument to terminate.
- if *pid* is -1, then wait for any child.
- WNOHANG ensures *waitpid()* does not block when the children have not changed state.
- *waitpid()* returns a process ID of child that has terminated.

waitpid() system call

- If a child terminates before the parent calls *waitpid()*, the kernel makes the child a zombie.
- For a zombie child, kernel's process table still has some information about the zombie.
- If a parent creates a child and does not perform a *waitpid()*, then an entry for the zombie child will be maintained indefinitely in the kernel's process table.
- When the parent does perform a *waitpid()*, the kernel removes the zombie information.

Parallel server: Saving errno

```c
static void
reapChild(int sig)
{
    int savedErrno = errno;
    while (waitpid(-1, NULL, WNOHANG) > 0)
        continue;
    errno = savedErrno;
}
```

- A signal handler may overwrite *errno*.
- For example, *waitpid()* call may update *errno*.
- This renders the signal handler *nonreentrant*.

Parallel server: Re-entrant functions

```
static void
reapChild(int sig)
{
  int savedErrno = errno;
  while (waitpid(-1, NULL, WNOHANG) > 0)
        continue;
  errno = savedErrno;
}
```

- A function is said to be *reentrant* if it can safely be simultaneously executed by multiple threads of execution in the same process.
- A function may be *nonreentrant* if it updates global or static variables (for example, errno).
- A workaround is to save the value of *errno* on entry and restore it before leaving the handler.

Parallel Server: Using fork()

When a client request is received, a child process is created.

```
for (;;) {
    int cfd = accept(lfd, NULL
        , NULL);
    if (cfd == -1)
       exit(EXIT_FAILURE);
    switch (fork()) {
      case -1:
         close(cfd);
         break;
      case 0: /* Child */
         close(lfd);
         handleRequest(cfd);
         _exit(EXIT_SUCCESS);
      default: /* Parent */
         close(cfd);
         break;
    }
}
```

- **Revisit Lecture 11 on processes.**
- Parent process uses *fork()* to create a child process that invokes *handleRequest()* function to handle the client.

Parallel Server: Using fork()

When a client request is received, a child process is created.

```
for (;;) {
    int cfd = accept(lfd, NULL
        , NULL);
    if (cfd == -1)
      exit(EXIT_FAILURE);
    switch (fork()) {
      case -1:
        close(cfd);
        break;
      case 0: /* Child */
        close(lfd);
        handleRequest(cfd);
        _exit(EXIT_SUCCESS);
      default: /* Parent */
        close(cfd);
        break;
    }
}
```

- After each *fork()*, the file descriptors for the listening and connected sockets are duplicated in the child.
- Child process closes the duplicate of the file descriptor for the listening socket since it does not accept any new connections.

Parallel Server: Using fork()

```
for (;;) {
    int cfd = accept(lfd, NULL
        , NULL);
    if (cfd == -1)
      exit(EXIT_FAILURE);
    switch (fork()) {
      case -1:
        close(cfd);
        break;
      case 0: /* Child */
        close(lfd);
        handleRequest(cfd);
        _exit(EXIT_SUCCESS);
      default: /* Parent */
        close(cfd);
        break;
    }
}
```

- Parent process therefore closes the file descriptor for the connected socket.
- It must otherwise it will run out of file descriptors.
- Parent process loops to accept the next client connection.

Server code: handleRequest() function

```c
static void
handleRequest(int cfd)
{
    char buf[BUFSIZE];
    size_t totRead;
    char* bufr = buf;
    for (totRead = 0;
         totRead < BUFSIZE;) {
        ssize_t numRead = read(
            cfd, ...);
        if (numRead == 0)
           break;
        if (numRead == -1) {
            /* check EINTR */
        }
        totRead += numRead;
        bufr += numRead;
    }
    printf("Received %s\n", buf);
    /* write to follow */
}
```

- The child process executes *handleRequest()* independently from the parent and other children.
- The read code is the same as the iterative case.

Server code: handleRequest() function

```
static void
handleRequest(int cfd)
{
    /* Continuation... */
    size_t totWritten;
    const char* bufw = buf;
    for (totWritten = 0;
        totWritten < BUFSIZE; ) {
        ssize_t numWritten =
            write(cfd, ...);
        if (numWritten <= 0) {
            /* check EINTR */
        }
        totWritten += numWritten;
        bufw += numWritten;
    }
}
```

- The write code is the same as the iterative case.

Parallel Server Using Processes: Summary

Key differences from the iterative server:

- Setup of SIGCHLD signal handler to deal with child processes appropriately.
- Creation of processes using fork() to handle clients simultaneously.
- Rest of the code is completely reusable.
- The read and write to service a client takes place in the child process's handleRequest() function.

Lecture 17 to follow

In the next lecture, we will develop a multithreaded TCP socket server.

Q & A