ЛАБОРАТОРНАЯ РАБОТА N 3

«Создание сетевых приложений в ОС UNIX с использованием интерфейса сокетов»

Содержание

Введение

Интенсивное развитие информационных технологий привело к росту популярности распределенных компьютерных систем и комплексов, важную роль в функционировании которых играет обмен данными между отдельными компонентами. Одним из способов решения этой задачи является использование семейства протоколов TCP/IP в качестве транспортного уровня, поверх которого реализуется прикладной протокол обмена данными между компонентами распределенной компьютерной системы. Повсеместное использование семейства протоколов TCP/IP упрощает интеграцию отдельных компонентов информационных систем, снижая затраты на внедрение и последующее обслуживание.

С другой стороны, нельзя отрицать наблюдаемый в последние годы рост популярности свободных реализаций UNIX-подобных операционных систем, наиболее известной из которых является Linux. Низкие затраты на приобретение и дальнейшее обслуживание, доступность исходных кодов, чрезвычайная гибкость в настройке и многое другое привели к тому, что многие распределенные информационные системы используют Linux в качестве ОС для серверных компонентов. Кроме того, немало популярных сетевых приложений изначально разрабатывались как раз для Linux и только в последствие были перенесены под Windows. Примером такого приложения может являться файлообменная сеть BitTorrent, которая по данным на декабрь 2004 года генерирует около одной трети всего трафика в Internet.

Кроме того, хотелось бы отметить те преимущества, которых можно добиться при внедрении Linux в учебный процесс. Во-первых, не смотря на кажущуюся сложность, основные идеи, заложенные еще разработчиками первых версий UNIX, сохранились до сих пор, удивляя своим изяществом и простотой. Во-вторых, доступность исходных кодов даёт студенту возможность посмотреть, каким образом разрабатываются, отлаживаются и сопровождаются крупные программные продукты, а при желании и наличии достаточной квалификации принять посильное участие. В-третьих, это наличие большого количества качественной документации, как в печатном, так и в электронном виде, что существенно упрощает освоение материала студентом.

Таким образом, можно сделать вывод о том, что тема разработки сетевых приложений под ОС UNIX является весьма актуальной и востребованной.

Программные интерфейсы поддержки сети в UNIX

В связи с тем, что семейство операционных систем UNIX развивалось различными группами разработчиков, существует как минимум два различных интерфейса поддержки сети: интерфейс сокетов и интерфейс XTI (X / Open Transport Interface). В связи с тем, что интерфейс сокетов стал стандартом де-факто для разработки сетевых приложений не только в UNIX, но и в других ОС, рекомендуется использовать именно его.

Описание шаблона последовательного эхо-сервера TCP

Шаблон последовательного эхо-сервера TCP является примером простейшего серверного приложения, использующего интерфейс сокетов. Данный тип сервера полностью обрабатывает запрос каждого клиента, прежде чем перейти к обслуживанию запроса следующего клиента.

```
001 /*
002  * Шаблон последовательного эхо-сервера TCP.
003  *
004  * Компиляция:
005  *      cc -Wall -O2 -o server1 server1.c
006  */
007
008 #include <errno.h>
009 #include <netinet/in.h>
010 #include <stdio.h>
011 #include <stdlib.h>
012 #include <string.h>
013 #include <sys/socket.h>
014 #include <unistd.h>
015
016 /*
017  * Конфигурация сервера.
018  */
019 #define PORT 1027
020 #define BACKLOG 5
021 #define MAXLINE 256
022
023 #define SA struct sockaddr
024
025 /*
026  * Обработчик фатальных ошибок.
027  */
028 void error(const char *s)
029 {
030         perror(s);
031         exit(-1);
032 }
033
034 /*
035  * Функции-обертки.
036  */
037 int Socket(int domain, int type, int protocol)
038 {
039         int rc;
040
041         rc = socket(domain, type, protocol);
042         if(rc == -1) error("socket()");
043
044         return rc;
045 }
046
047 int Bind(int socket, struct sockaddr *addr, socklen_t addrlen)
048 {
049         int rc;
050
051         rc = bind(socket, addr, addrlen);
052         if(rc == -1) error("bind()");
053
```

```
054         return rc;
055 }
056
057 int Listen(int socket, int backlog)
058 {
059         int rc;
060
061         rc = listen(socket, backlog);
062         if(rc == -1) error("listen()");
063
064         return rc;
065 }
066
067 int Accept(int socket, struct sockaddr *addr, socklen_t *addrlen)
068 {
069         int rc;
070
071         for(;;) {
072                 rc = accept(socket, addr, addrlen);
073                 if(rc != -1) break;
074                 if(errno == EINTR || errno == ECONNABORTED) continue;
075                 error("accept()");
076         }
077
078         return rc;
079 }
080
081 void Close(int fd)
082 {
083         int rc;
084
085         for(;;) {
086                 rc = close(fd);
087                 if(!rc) break;
088                 if(errno == EINTR) continue;
089                 error("close()");
090         }
091 }
092
093 size_t Read(int fd, void *buf, size_t count)
094 {
095         ssize_t rc;
096
097         for(;;) {
098                 rc = read(fd, buf, count);
099                 if(rc != -1) break;
100                 if(errno == EINTR) continue;
101                 error("read()");
102         }
103
104         return rc;
105 }
106
107 size_t Write(int fd, const void *buf, size_t count)
108 {
109         ssize_t rc;
110
111         for(;;) {
112                 rc = write(fd, buf, count);
113                 if(rc != -1) break;
114                 if(errno == EINTR) continue;
115                 error("write()");
116         }
117
118         return rc;
119 }
120
121 /*
122  * Чтение строки из сокета.
123  */
124 size_t reads(int socket, char *s, size_t size)
125 {
126         char *p;
127         size_t n, rc;
```

```
128
129         /* Проверить корректность переданных аргументов. */
130         if(s == NULL) {
131                 errno = EFAULT;
132                 error("reads()");
133         }
134         if(!size) return 0;
135
136         p = s;
137         size--;
138         n = 0;
139         while(n < size) {
140                 rc = Read(socket, p, 1);
141                 if(rc == 0) break;
142                 if(*p == '\n') {
143                         p++;
144                         n++;
145                         break;
146                 }
147                 p++;
148                 n++;
149         }
150         *p = 0;
151
152         return n;
153 }
154
155 /*
156  * Запись count байтов в сокет.
157  */
158 size_t writen(int socket, const char *buf, size_t count)
159 {
160         const char *p;
161         size_t n, rc;
162
163         /* Проверить корректность переданных аргументов. */
164         if(buf == NULL) {
165                 errno = EFAULT;
166                 error("writen()");
167         }
168
169         p = buf;
170         n = count;
171         while(n) {
172                 rc = Write(socket, p, n);
173                 n -= rc;
174                 p += rc;
175         }
176
177         return count;
178 }
179
180 void serve_client(int socket)
181 {
182         char s[MAXLINE];
183         ssize_t rc;
184
185         while((rc = reads(socket, s, MAXLINE)) > 0) {
186                 if(writen(socket, s, rc) == -1) break;
187         }
188         Close(socket);
189 }
190
191 int main(void)
192 {
193         int lsocket;    /* Дескриптор прослушиваемого сокета. */
194         int csocket;    /* Дескриптор присоединенного сокета. */
195         struct sockaddr_in servaddr;
196
197         /* Создать сокет. */
198         lsocket = Socket(PF_INET, SOCK_STREAM, 0);
199
200         /* Инициализировать структуру адреса сокета сервера. */
201         memset(&servaddr, 0, sizeof(servaddr));
```

```
202         servaddr.sin_family = AF_INET;
203         servaddr.sin_port = htons(PORT);
204         servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
205
206         /* Связать сокет с локальным адресом протокола. */
207         Bind(lsocket, (SA *) &servaddr, sizeof(servaddr));
208
209         /* Преобразовать неприсоединенный сокет в пассивный. */
210         Listen(lsocket, BACKLOG);
211
212         for(;;) {
213                 csocket = Accept(lsocket, NULL, 0);
214                 serve_client(csocket);
215         }
216
217         return 0;
218 }
```

28–32 Функция обработки фатальных ошибок. Выводит в стандартный поток обработки ошибок заданное сообщение с расшифровкой кода ошибки, находящейся в системной переменной `errno`, и завершает работу программы.

37–119 Функции-обертки для системных вызовов. Предназначены для своевременной диагностики ошибок и их обработки. Возможны следующие ситуации:

- Выполнение системного вызова произошло без ошибок. В этом случае происходит выход из функции-обертки.
- В ходе своего выполнения медленный системный вызов был прерван поступившим сигналом. В этом случае системный вызов *может* возвратить ошибку `EINTR`, следовательно, необходимо перезапустить прерванный системный вызов.
- Системный вызов завершился с ошибкой. В этом случае вызывается функция обработки фатальных ошибок с указанием имени системного вызова.

124–153 Функция чтения строки из сокета. Осуществляет посимвольное чтение строки из сокета. Выход из функции происходит в двух случаях:

- Из сокета был прочитан код начала новой строки.
- Прочитан `(size-1)` байт.

158–178 Функция записи заданного количества байт в сокет. Необходимость в этой функции связана с тем, что потоковые сокеты (в частности, сокеты TCP), демонстрируют с функциями `read()` и `write()` поведение, отличное от файлового ввода-вывода. Функция `read()` или `write()` на потоковом сокете может ввести или вывести меньше байт, чем запрашивалось. Причиной может быть достижение границ буфера для сокета в ядре. Всё, что требуется в этой ситуации, - чтобы процесс повторил вызов функции `read()` или `write()` для ввода или вывода оставшихся байт.

`180–189` Функция обслуживания клиента. Принимает строку от клиента и отсылает ее обратно. В случае завершения соединения производит закрытие присоединенного сокета.

`191–218` Основная функция сервера.

Описание шаблона параллельного эхо-сервера TCP, использующего модель “один клиент – один процесс”

Данный тип сервера способен обрабатывать несколько запросов одновременно, выделяя по одному дочернему процессу для каждого клиента. Единственным ограничением на количество одновременно обрабатываемых клиентских запросов является ограничение на ОС на количество дочерних процессов, допустимых для пользователя, в сеансе которого работает сервер.

```
001 /*
002  * Шаблон параллельного эхо-сервера TCP, работающего по модели
003  * "один клиент - один процесс".
004  *
005  * Компиляция:
006  *      cc -Wall -O2 -o server2 server2.c
007  */
008
009 #include <errno.h>
010 #include <netinet/in.h>
011 #include <signal.h>
012 #include <stdio.h>
013 #include <stdlib.h>
014 #include <string.h>
015 #include <sys/socket.h>
016 #include <sys/wait.h>
017 #include <unistd.h>
018
019 /*
020  * Конфигурация сервера.
021  */
022 #define PORT 1027
023 #define BACKLOG 5
024 #define MAXLINE 256
025
026 #define SA struct sockaddr
027
028 /*
029  * Обработчик фатальных ошибок.
030  */
031 void error(const char *s)
032 {
033         perror(s);
034         exit(-1);
035 }
036
037 /*
038  * Функции-обертки.
039  */
040 int Socket(int domain, int type, int protocol)
041 {
042         int rc;
043
044         rc = socket(domain, type, protocol);
045         if(rc == -1) error("socket()");
046
047         return rc;
048 }
049
050 int Bind(int socket, struct sockaddr *addr, socklen_t addrlen)
051 {
052         int rc;
053
054         rc = bind(socket, addr, addrlen);
055         if(rc == -1) error("bind()");
056
057         return rc;
058 }
059
060 int Listen(int socket, int backlog)
```

```
061 {
062         int rc;
063
064         rc = listen(socket, backlog);
065         if(rc == -1) error("listen()");
066
067         return rc;
068 }
069
070 int Accept(int socket, struct sockaddr *addr, socklen_t *addrlen)
071 {
072         int rc;
073
074         for(;;) {
075                 rc = accept(socket, addr, addrlen);
076                 if(rc != -1) break;
077                 if(errno == EINTR || errno == ECONNABORTED) continue;
078                 error("accept()");
079         }
080
081         return rc;
082 }
083
084 void Close(int fd)
085 {
086         int rc;
087
088         for(;;) {
089                 rc = close(fd);
090                 if(!rc) break;
091                 if(errno == EINTR) continue;
092                 error("close()");
093         }
094 }
095
096 size_t Read(int fd, void *buf, size_t count)
097 {
098         ssize_t rc;
099
100         for(;;) {
101                 rc = read(fd, buf, count);
102                 if(rc != -1) break;
103                 if(errno == EINTR) continue;
104                 error("read()");
105         }
106
107         return rc;
108 }
109
110 size_t Write(int fd, const void *buf, size_t count)
111 {
112         ssize_t rc;
113
114         for(;;) {
115                 rc = write(fd, buf, count);
116                 if(rc != -1) break;
117                 if(errno == EINTR) continue;
118                 error("write()");
119         }
120
121         return rc;
122 }
123
124 void Sigaction(int signum, const struct sigaction *act, struct sigaction *oact)
125 {
126         int rc;
127
128         for(;;) {
129                 rc = sigaction(signum, act, oact);
130                 if(!rc) break;
131                 if(errno == EINTR) continue;
132                 error("close()");
133         }
134 }
```

```
135
136 pid_t Fork()
137 {
138         pid_t rc;
139
140         rc = fork();
141         if(rc == -1) error("fork()");
142
143         return rc;
144 }
145
146 /*
147  * Чтение строки из сокета.
148  */
149 size_t reads(int socket, char *s, size_t size)
150 {
151         char *p;
152         size_t n, rc;
153
154         /* Проверить корректность переданных аргументов. */
155         if(s == NULL) {
156                 errno = EFAULT;
157                 error("reads()");
158         }
159         if(!size) return 0;
160
161         p = s;
162         size--;
163         n = 0;
164         while(n < size) {
165                 rc = Read(socket, p, 1);
166                 if(rc == 0) break;
167                 if(*p == '\n') {
168                         p++;
169                         n++;
170                         break;
171                 }
172                 p++;
173                 n++;
174         }
175         *p = 0;
176
177         return n;
178 }
179
180 /*
181  * Запись count байтов в сокет.
182  */
183 size_t writen(int socket, const char *buf, size_t count)
184 {
185         const char *p;
186         size_t n, rc;
187
188         /* Проверить корректность переданных аргументов. */
189         if(buf == NULL) {
190                 errno = EFAULT;
191                 error("writen()");
192         }
193
194         p = buf;
195         n = count;
196         while(n) {
197                 rc = Write(socket, p, n);
198                 n -= rc;
199                 p += rc;
200         }
201
202         return count;
203 }
204
205 void serve_client(int socket)
206 {
207         char s[MAXLINE];
208         ssize_t rc;
```

```
209
210         while((rc = reads(socket, s, MAXLINE)) > 0) {
211                 if(writen(socket, s, rc) == -1) break;
212         }
213         Close(socket);
214 }
215
216 /*
217  * Обработчик сигнала SIGCHLD.
218  */
219 static void sighandler(int signum)
220 {
221         while(waitpid(-1, NULL, WNOHANG) > 0);
222 }
223
224 /*
225  * Установить обработчик сигнала SIGCHLD.
226  */
227 static void set_sighandler()
228 {
229         struct sigaction act;
230
231         act.sa_handler = sighandler;
232         sigemptyset(&act.sa_mask);
233         act.sa_flags = 0;
234
235         Sigaction(SIGCHLD, &act, NULL);
236 }
237
238 int main()
239 {
240         int lsocket;    /* Дескриптор прослушиваемого сокета. */
241         int csocket;    /* Дескриптор присоединенного сокета. */
242         struct sockaddr_in servaddr;
243         pid_t pid;
244
245         /* Установить обработчик сигнала SIGCHLD. */
246         set_sighandler();
247
248         /* Создать сокет. */
249         lsocket = Socket(PF_INET, SOCK_STREAM, 0);
250
251         /* Инициализировать структуру адреса сокета сервера. */
252         memset(&servaddr, 0, sizeof(servaddr));
253         servaddr.sin_family = AF_INET;
254         servaddr.sin_port = htons(PORT);
255         servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
256
257         /* Связать сокет с локальным адресом протокола. */
258         Bind(lsocket, (SA *) &servaddr, sizeof(servaddr));
259
260         /* Преобразовать неприсоединенный сокет в пассивный. */
261         Listen(lsocket, BACKLOG);
262
263         for(;;) {
264                 csocket = Accept(lsocket, NULL, 0);
265
266                 pid = Fork();
267                 if(pid) {
268                         /* Родительский процесс. */
269                         Close(csocket);
270                 }
271                 else {
272                         /* Дочерний процесс. */
273                         Close(lsocket);
274                         serve_client(csocket);
275                         exit(0);
276                 }
277
278         }
279
280         return 0;
281 }
```

219-222 Обработчик сигнала `SIGCHLD`. Этот сигнал посылается ядром ОС родительскому процессу при завершении одного из его дочерних процессов. Задача родительского процесса провести обработку зомбированных процессов, выполнив для каждого из них системный вызов `wait()` или `waitpid()`.

227-236 Функция, устанавливающая обработчик сигнала `SIGCHLD`.

266-276 Создание дочернего процесса. Дочерний процесс создается с помощью системного вызова `fork()`, при этом дочерний процесс является точной копией процесса родительского. Особенностью функции `fork()` является то, что она вызывается *один* раз, но при этом возвращает *два* значение. Одно значение эта функция возвращает в родительском процессе – этим значением является идентификатор созданного процесса. Второе значение (ноль), она возвращает в дочернем процессе.

Так как дочерний процесс является точной копией процесса родительского, то и все дескрипторы, открытые в родительском процессе до вызова `fork()` становятся доступными дочернему процессу. По этой причине, в родительском процессе производиться закрытие неиспользуемого присоединенного сокета, а в дочернем процессе – сокета прослушиваемого.

Описание шаблона параллельного эхо-сервера TCP, использующего модель “один клиент – один поток”

Этот тип сервера похож на предыдущий, однако в нем для обслуживания каждого клиента выделяется не отдельный процесс, а отдельный поток. Этот подход имеет как свои достоинства, так и свои недостатки.

Достоинства:

- Низкая стоимость создания потока по сравнению с созданием процесса.
- Меньшие накладные расходы на обмен данными между потоками по сравнению с обменом данными между процессами.

Недостатки:

- Все потоки одного процесса разделяют между собой все ресурсы процесса, поэтому сбой одного потока способен привести к сбою все программы.

```
001 /*
002  * Шаблон параллельного эхо-сервера TCP, работающего по модели
003  * "один клиент - один поток".
004  *
005  * Компиляция:
006  *      cc -Wall -O2 -lpthread -o server3 server3.c
007  */
008
009 #include <errno.h>
010 #include <netinet/in.h>
011 #include <pthread.h>
012 #include <stdio.h>
013 #include <stdlib.h>
014 #include <string.h>
015 #include <sys/socket.h>
016 #include <unistd.h>
017
018 /*
019  * Конфигурация сервера.
020  */
021 #define PORT 1027
022 #define BACKLOG 5
023 #define MAXLINE 256
024
025 #define SA struct sockaddr
026
027 /*
028  * Обработчик фатальных ошибок.
029  */
030 void error(const char *s)
031 {
032         perror(s);
033         exit(-1);
034 }
035
036 /*
037  * Функции-обертки.
038  */
039 int Socket(int domain, int type, int protocol)
040 {
041         int rc;
042
043         rc = socket(domain, type, protocol);
044         if(rc == -1) error("socket()");
045
046         return rc;
047 }
048
049 int Bind(int socket, struct sockaddr *addr, socklen_t addrlen)
050 {
051         int rc;
052
```

```
053         rc = bind(socket, addr, addrlen);
054         if(rc == -1) error("bind()");
055
056         return rc;
057 }
058
059 int Listen(int socket, int backlog)
060 {
061         int rc;
062
063         rc = listen(socket, backlog);
064         if(rc == -1) error("listen()");
065
066         return rc;
067 }
068
069 int Accept(int socket, struct sockaddr *addr, socklen_t *addrlen)
070 {
071         int rc;
072
073         for(;;) {
074                 rc = accept(socket, addr, addrlen);
075                 if(rc != -1) break;
076                 if(errno == EINTR || errno == ECONNABORTED) continue;
077                 error("accept()");
078         }
079
080         return rc;
081 }
082
083 void Close(int fd)
084 {
085         int rc;
086
087         for(;;) {
088                 rc = close(fd);
089                 if(!rc) break;
090                 if(errno == EINTR) continue;
091                 error("close()");
092         }
093 }
094
095 size_t Read(int fd, void *buf, size_t count)
096 {
097         ssize_t rc;
098
099         for(;;) {
100                 rc = read(fd, buf, count);
101                 if(rc != -1) break;
102                 if(errno == EINTR) continue;
103                 error("read()");
104         }
105
106         return rc;
107 }
108
109 size_t Write(int fd, const void *buf, size_t count)
110 {
111         ssize_t rc;
112
113         for(;;) {
114                 rc = write(fd, buf, count);
115                 if(rc != -1) break;
116                 if(errno == EINTR) continue;
117                 error("write()");
118         }
119
120         return rc;
121 }
122
123 void *Malloc(size_t size)
124 {
125         void *rc;
126
```

```
127         rc = malloc(size);
128         if(rc == NULL) error("malloc()");
129
130         return rc;
131 }
132
133 void Pthread_create(pthread_t *thread, pthread_attr_t *attr,
134         void *(*start_routine)(void *), void *arg)
135 {
136         int rc;
137
138         rc = pthread_create(thread, attr, start_routine, arg);
139         if(rc) {
140                 errno = rc;
141                 error("pthread_create()");
142         }
143 }
144
145 /*
146  * Чтение строки из сокета.
147  */
148 size_t reads(int socket, char *s, size_t size)
149 {
150         char *p;
151         size_t n, rc;
152
153         /* Проверить корректность переданных аргументов. */
154         if(s == NULL) {
155                 errno = EFAULT;
156                 error("reads()");
157         }
158         if(!size) return 0;
159
160         p = s;
161         size--;
162         n = 0;
163         while(n < size) {
164                 rc = Read(socket, p, 1);
165                 if(rc == 0) break;
166                 if(*p == '\n') {
167                         p++;
168                         n++;
169                         break;
170                 }
171                 p++;
172                 n++;
173         }
174         *p = 0;
175
176         return n;
177 }
178
179 /*
180  * Запись count байтов в сокет.
181  */
182 size_t writen(int socket, const char *buf, size_t count)
183 {
184         const char *p;
185         size_t n, rc;
186
187         /* Проверить корректность переданных аргументов. */
188         if(buf == NULL) {
189                 errno = EFAULT;
190                 error("writen()");
191         }
192
193         p = buf;
194         n = count;
195         while(n) {
196                 rc = Write(socket, p, n);
197                 n -= rc;
198                 p += rc;
199         }
200
```

```
201         return count;
202 }
203
204 void *serve_client(void *arg)
205 {
206         int socket;
207         char s[MAXLINE];
208         ssize_t rc;
209
210         /* Перевести поток в отсоединенное (detached) состояние. */
211         pthread_detach(pthread_self());
212
213         socket = *((int *) arg);
214         free(arg);
215
216         while((rc = reads(socket, s, MAXLINE)) > 0) {
217                 if(writen(socket, s, rc) == -1) break;
218         }
219         Close(socket);
220
221         return NULL;
222 }
223
224 int main(void)
225 {
226         int lsocket;    /* Дескриптор прослушиваемого сокета. */
227         int csocket;    /* Дескриптор присоединенного сокета. */
228         struct sockaddr_in servaddr;
229         int *arg;
230         pthread_t thread;
231
232         /* Создать сокет. */
233         lsocket = Socket(PF_INET, SOCK_STREAM, 0);
234
235         /* Инициализировать структуру адреса сокета сервера. */
236         memset(&servaddr, 0, sizeof(servaddr));
237         servaddr.sin_family = AF_INET;
238         servaddr.sin_port = htons(PORT);
239         servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
240
241         /* Связать сокет с локальным адресом протокола. */
242         Bind(lsocket, (SA *) &servaddr, sizeof(servaddr));
243
244         /* Преобразовать неприсоединенный сокет в пассивный. */
245         Listen(lsocket, BACKLOG);
246
247         for(;;) {
248                 csocket = Accept(lsocket, NULL, 0);
249
250                 arg = Malloc(sizeof(int));
251                 *arg = csocket;
252
253                 Pthread_create(&thread, NULL, serve_client, arg);
254         }
255
256         return 0;
257 }
```

204-222 Функция обслуживания клиента. Отличается от рассмотренных выше способом передачи аргумента – здесь используется универсальный механизм передачи произвольного числа аргументов стартовой функции потока с помощью выделения динамической памяти в родительском потоке и её последующего освобождения в потоке дочернем.

250-253 Выделение необходимого объема динамической памяти для передачи аргумента в стартовую функцию дочернего потока и ее инициализация. Создание дочернего потока.

Описание шаблона клиента

Так как клиент должен обрабатывать поступление данных от двух источников (клавиатуры и удаленного сервера) используется модель мультиплексирования ввода-вывода с помощью функции `select()`.

```
001 /*
002  * Шаблон TCP клиента.
003  *
004  * Компиляция:
005  *      cc -Wall -O2 -o client client.c
006  *
007  * Завершение работы клиента: Ctrl+D.
008  */
009
010 #include <arpa/inet.h>
011 #include <errno.h>
012 #include <netinet/in.h>
013 #include <stdio.h>
014 #include <stdlib.h>
015 #include <string.h>
016 #include <sys/types.h>
017 #include <sys/select.h>
018 #include <sys/socket.h>
019 #include <unistd.h>
020
021 #include <limits.h>
022
023 #define MAX(a, b) ((a) > (b) ? (a) : (b))
024
025 #define PORT 1027
026 #define MAXLINE 256
027
028 #define SA struct sockaddr
029
030 /*
031  * Обработчик фатальных ошибок.
032  */
033 void error(const char *s)
034 {
035         perror(s);
036         exit(-1);
037 }
038
039 /*
040  * Функции-обертки.
041  */
042 int Socket(int domain, int type, int protocol)
043 {
044         int rc;
045
046         rc = socket(domain, type, protocol);
047         if(rc == -1) error("socket()");
048
049         return rc;
050 }
051
052 void Connect(int socket, const struct sockaddr *addr, socklen_t addrlen)
053 {
054         int rc;
055
056         rc = connect(socket, addr, addrlen);
057         if(rc == -1) error("connect()");
058 }
059
060 void Close(int fd)
061 {
062         int rc;
063
```

```
064         for(;;) {
065                 rc = close(fd);
066                 if(!rc) break;
067                 if(errno == EINTR) continue;
068                 error("close()");
069         }
070 }
071
072 void Inet_aton(const char *str, struct in_addr *addr)
073 {
074         int rc;
075
076         rc = inet_aton(str, addr);
077         if(!rc) {
078                 /* Функция inet_aton() не меняет errno в случае ошибки. Чтобы
079                 сообщение, выводимое error(), было более осмысленным,
080                 присваиваем errno значение EINVAL. */
081
082                 errno = EINVAL;
083                 error("inet_aton()");
084         }
085 }
086
087 int Select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
088         struct timeval *timeout)
089 {
090         int rc;
091
092         for(;;) {
093                 rc = select(n, readfds, writefds, exceptfds, timeout);
094                 if(rc != -1) break;
095                 if(errno == EINTR) continue;
096                 error("select()");
097         }
098
099         return rc;
100 }
101
102 size_t Read(int fd, void *buf, size_t count)
103 {
104         ssize_t rc;
105
106         for(;;) {
107                 rc = read(fd, buf, count);
108                 if(rc != -1) break;
109                 if(errno == EINTR) continue;
110                 error("read()");
111         }
112
113         return rc;
114 }
115
116 size_t Write(int fd, const void *buf, size_t count)
117 {
118         ssize_t rc;
119
120         for(;;) {
121                 rc = write(fd, buf, count);
122                 if(rc != -1) break;
123                 if(errno == EINTR) continue;
124                 error("write()");
125         }
126
127         return rc;
128 }
129
130 /*
131  * Запись count байтов в сокет.
132  */
133 size_t writen(int socket, const char *buf, size_t count)
134 {
135         const char *p;
136         size_t n, rc;
137
```

```
138         /* Проверить корректность переданных аргументов. */
139         if(buf == NULL) {
140                 errno = EFAULT;
141                 error("writen()");
142         }
143
144         p = buf;
145         n = count;
146         while(n) {
147                 rc = Write(socket, p, n);
148                 n -= rc;
149                 p += rc;
150         }
151
152         return count;
153 }
154
155 void show_usage()
156 {
157         puts("Usage: client ip_address");
158         exit(-1);
159 }
160
161 void do_work(int socket)
162 {
163         int n;
164         fd_set readfds;
165         char s[MAXLINE];
166         ssize_t rc;
167
168         n = MAX(STDIN_FILENO, socket) + 1;
169
170         for(;;) {
171                 /* Инициализировать набор дескрипторов. */
172                 FD_ZERO(&readfds);
173                 FD_SET(STDIN_FILENO, &readfds);
174                 FD_SET(socket, &readfds);
175
176                 Select(n, &readfds, NULL, NULL, NULL);
177                 if(FD_ISSET(STDIN_FILENO, &readfds)) {
178                         rc = Read(STDIN_FILENO, s, MAXLINE);
179                         if(!rc) break;
180                         writen(socket, s, rc);
181                 }
182                 if(FD_ISSET(socket, &readfds)) {
183                         rc = Read(socket, s, MAXLINE);
184                         if(!rc) break;
185                         Write(STDOUT_FILENO, s, rc);
186                 }
187         }
188 }
189
190 int main(int argc, char **argv)
191 {
192         int socket;
193         struct sockaddr_in servaddr;
194
195         if(argc != 2) show_usage();
196
197         socket = Socket(PF_INET, SOCK_STREAM, 0);
198
199         /* Инициализировать структуру адреса сокета. */
200         memset(&servaddr, 0, sizeof(servaddr));
201         servaddr.sin_family = AF_INET;
202         servaddr.sin_port = htons(PORT);
203         Inet_aton(argv[1], &servaddr.sin_addr);
204
205         Connect(socket, (SA *) &servaddr, sizeof(servaddr));
206         do_work(socket);
207         Close(socket);
208
209         return 0;
210 }
```

170-187 Основной цикл работы клиента. В строках 171–174 инициализируется набор дескрипторов для передачи функции `select()`, в который входит дескриптор стандартного потока ввода и присоединенный сокет. В строке 176 производится вызов функции `select()`, на которой происходит блокировка процесса до момента, когда хотя бы один из переданных дескрипторов будет готов для чтения. В строках 177–181 обрабатывается случай готовности для чтения стандартного потока ввода, а в строках 182–185 – готовность для чтения сокета.

Заключение

В ходе выполнения лабораторной работы студенту необходимо решить следующие задачи:

- Изучить документацию, касающуюся как системного программирования в ОС UNIX, так и непосредственно разработки сетевых приложений.
- Разработаться с шаблонами сетевых приложений: 3 шаблона сервера и 1 шаблон клиента. Шаблоны представляют собой простые гарантированно работающие примеры сетевых приложений, которые дают студенту своеобразную базу для выполнения лабораторной работы.
- Самостоятельно разработать клиент-серверное приложение для предложенного преподавателем задания.

Список литературы

1. У. Р. Стивенс UNIX: разработка сетевых приложений. – СПб.: Питер, 2003. – 1088 с.: ил. – (Серия «Мастер-класс»). ISBN 5-318-00535-7

2. А. М. Робачевский А. М Операционная система UNIX. – СПб.: БХВ-Петербург, 2000. – 528 с.: ил. ISBN 5-8206-0030-4

3. Кейт Хэвиленд, Дайна Грэй, Бен Салама Системное программирование в UNIX. Руководство программиста по разработке ПО. – М., ДМК Пресс, 2000. – 368 с., ил. (Серия «Для программистов»). ISBN 5-94074-008-1