ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

КУРСОВОЙ ПРОЕКТ

по дисциплине “Операционные системы”

на тему

LiteSh

Выполнили студенты Крюков Илья, Русманов Владислав, Сысоева Анастасия

Ф.И.О.

Группы ИВ-822

Работу принял _____ Ассистент Б.В. Бочкарёв

подпись

Защищена _____ Оценка _____

Новосибирск – 2020

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данном проекте LiteSh реализован функционал по работе с файловой системой. Функционал с помощью которого можно порождать процессы и переводить их в фоновый режим, получать и обрабатывать сигналы от внешних программ и операционной системы.Также в проекте реализован клиент с возможностью подключения к сетевому интерфейсу, отправкой команд на сервер и получение результата выполнения. Сервер с возможностью предоставления сетевого интерфейса для подключения, с обработкой приходящих команд и отправкой результата. Помимо прочего в проекте присутствует функционал управления демонами. Также имеются динамические библиотеки для работы с сервером.

ОСНОВНАЯ ТЕОРИЯ

Процессы

В самом первом приближении можно считать, что процесс — это программа, выполняющаяся в оперативной памяти компьютера. Но такая формулировка как бы подразумевает, что речь идет только о наборе машинных инструкций, последовательно выполняемых процессором. Фактически же в многозадачных системах понятие процесса является значительно более сложным.

В любой многозадачной системе одновременно может быть запущено много программ, то есть много процессов. Впрочем, слово "одновременно" здесь применено не совсем корректно, поскольку на самом деле в каждый момент времени выполняется только один процесс. Ядро (точнее, особый процесс ядра – планировщик) выделяет каждому процессу небольшой квант времени и по истечении этого кванта передает управление следующему процессу. Кванты времени, выделяемые каждому процессу, так малы, что у пользователя создается иллюзия одновременного выполнения многих процессов. Но, чтобы организовать переключение между процессами по истечении кванта времени, приходится делать как бы «мгновенный снимок» состояния программы и сохранять этот снимок где-то в памяти. Этот «снимок» содержит информацию о состоянии регистров центрального процессора на момент прерывания программы, указание на то, с какой команды возобновить исполнение программы (состояние счетчика команд), содержимое стека и тому подобные данные. Когда процесс снова получает в свое распоряжение ЦП, состояние регистров ЦП и стека восстанавливается из сделанного «снимка» и выполнение программы возобновляется в точности с того места, где она была остановлена. Примерно такие же действия выполняются в тех случаях, когда какому-то процессу необходимо вызвать некоторую системную функцию (вызов ядра).

Новые процессы создаются в Linux методом «клонирования» какого-то уже существующего процесса, путем вызова системных функций clone(2) и

fork(2). Процедура порождения нового процесса выполняется в режиме ядра и происходит следующим образом.

Создается новая структура task_struct в таблице процессов ядра и содержание такой же структуры старого (или текущего) процесса копируется в новую структуру.

Назначается идентификатор (PID) нового процесса. PID – это уникальное положительное число, которое присваивается каждому процессу при его рождении. Именно по этим идентификаторам система различает процессы.

Увеличиваются счетчики открытия файлов (порожденный процесс наследует все открытые файлы родительского процесса).

После того, как процесс создан, запускается выполняемая им программа с помощью одного из вариантов системного вызова exec. Параметрами функции exec является имя выполняемого файла и, если нужно, параметры, которые будут переданы этой программе. Программа из указанного файла загружается в адресное пространство процесса, порожденного с помощью fork(), счетчик команд устанавливается в начальное значение и вновь созданный процесс переходит в режим ожидания того момента, когда планировщик выделит ему время центрального процессора.

В том процессе, откуда вызывались функции fork() и exec, управление передается в точку возврата из системного вызова и выполнение этого процесса продолжается. Родительский процесс может дожидаться окончания выполнения всех своих процессов-потомков с помощью системного вызова wait.

Демоны

Демоны отличаются от обычных процессов только тем, что они работают в неинтерактивном режиме. Если с обычным процессом всегда ассоциирован какой-то терминал или псевдотерминал, через который осуществляется взаимодействие процесса с пользователем, то демон такого

терминала не имеет. Демоны обычно используются для выполнения сервисных функций, обслуживания запросов от других процессов, причем не обязательно выполняющихся на данном компьютере. Пользователь не может непосредственно управлять демонами, он может влиять на их работу, только посылая им какие-то задания, например, отправляя документ на печать.

Одним из главных, если можно так выразиться, демонов в системе является демон init. Как уже говорилось, init является прародителем всех процессов в системе и имеет идентификатор 1. Выполнив задачи, поставленные в ему в файле inittab, демон init не завершает свою работу – он постоянно находится в памяти и отслеживает выполнение других процессов.

Сокеты

Сокет (от англ. “socket” – разъем, гнездо) – это унифицированный интерфейс взаимодействия процессов с использованием стека сетевых протоколов в ядре. Сокет представляет собой виртуальный объект, аналогичный в некотором смысле сетевому интерфейсу. Такой объект характеризуется семейством протоколов и типом сокета, который представляет собой не что иное, как тип передаваемого через сокет потока данных. В настоящее время сокеты в Linux допускают использование примерно тридцати семейств протоколов и 6 типов сокетов (все они описываются в файле /usr/include/bits/socket.h)

Сокеты имеют соответствующий интерфейс доступа в файловой системе (имя, подобное имени файла или, точнее, устройства) и обращение к ним, так же как к обычным файлам, осуществляется через дескрипторы. Однако, в отличие от обычных файлов, сокеты представляют собой виртуальный объект, который существует, пока на него ссылается хотя бы один из процессов.

Чтобы использовать сокет, какой-то процесс должен его создать, а другие процессы – установить соединение с данным сокетом. Для создания сокетов используется системный вызов soket(2). Затем сокету присваивается имя

путем вызова системного вызова bind(2). Другие процессы для установления соединения с сокетом используют системный вызов connect(2). По всем этим вызовам имеются man-странички, из которых вы можете получить дополнительную информацию о сокетах.

Сокетное программирование — это способ соединения двух узлов в сети для связи друг с другом. Один сокет (узел) прослушивает определенный порт с IP-адреса, а другой сокет обращается к другому, чтобы сформировать соединение. Сервер формирует сокет слушателя, в то время как клиент обращается к серверу.

Динамические библиотеки

Библиотеки были придуманы для объединения схожей функциональности в отдельные модули, которые могли использоваться совместно несколькими разработчиками. Такой подход соответствует модульному программированию, при котором программы строятся на основе модулей. В Linux доступно два вида библиотек, и каждый из них имеет свои преимущества и недостатки. При использовании **с**татических библиотек их функциональность внедряется в программный код на этапе компиляции. Напротив, динамические библиотек загружаются после запуска приложения, а связывание происходит на этапе выполнения. Существует два способа использования совместно используемых библиотек: динамическая компоновка в момент загрузки и динамическая загрузка с подключением программным путем

В простых программах с минимальной функциональностью статические библиотеки могут быть предпочтительнее. В программах же, использующих несколько библиотек, применение совместно используемых библиотек позволяет снизить потребление оперативной и дисковой памяти во время работы приложения. Это достигается за счет того, что одна совместно используемая библиотека может использоваться одновременно несколькими приложениями, при этом она присутствует в памяти в единственном

экземпляре. В случае со статическими библиотеками каждая программа загружает свою собственную копию библиотечных функций.

В GNU/Linux доступно два метода работы с совместно используемыми библиотеками. Первый способ – это динамическая компоновка вашего приложения с совместно используемой библиотекой. При этом загрузку библиотеки при запуске программы возьмет на себя Linux (если, конечно, она не была загружена в память раньше). Второй способ подразумевает явный вызов функций библиотеки в процессе т. н.динамической загрузки. В этом случае программа явно загружает нужную библиотеку, а затем вызывает определенную библиотечную функцию. На этом методе обычно основан механизм загрузки подключаемых программных модулей – плагинов.

БЛОК-СХЕМЫ ПРОЕКТА

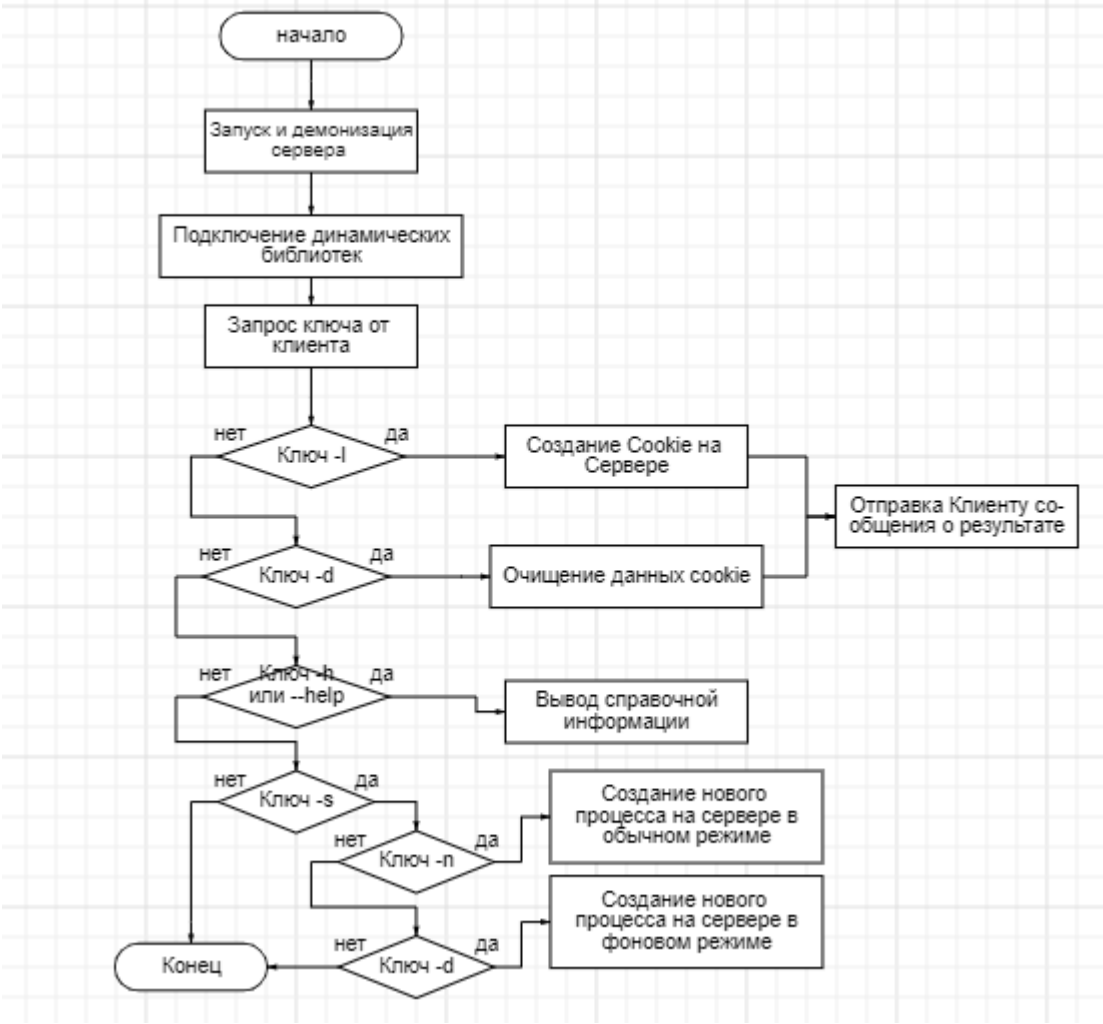

Рисунок 1. Блок-схема подключения сервера и клиента

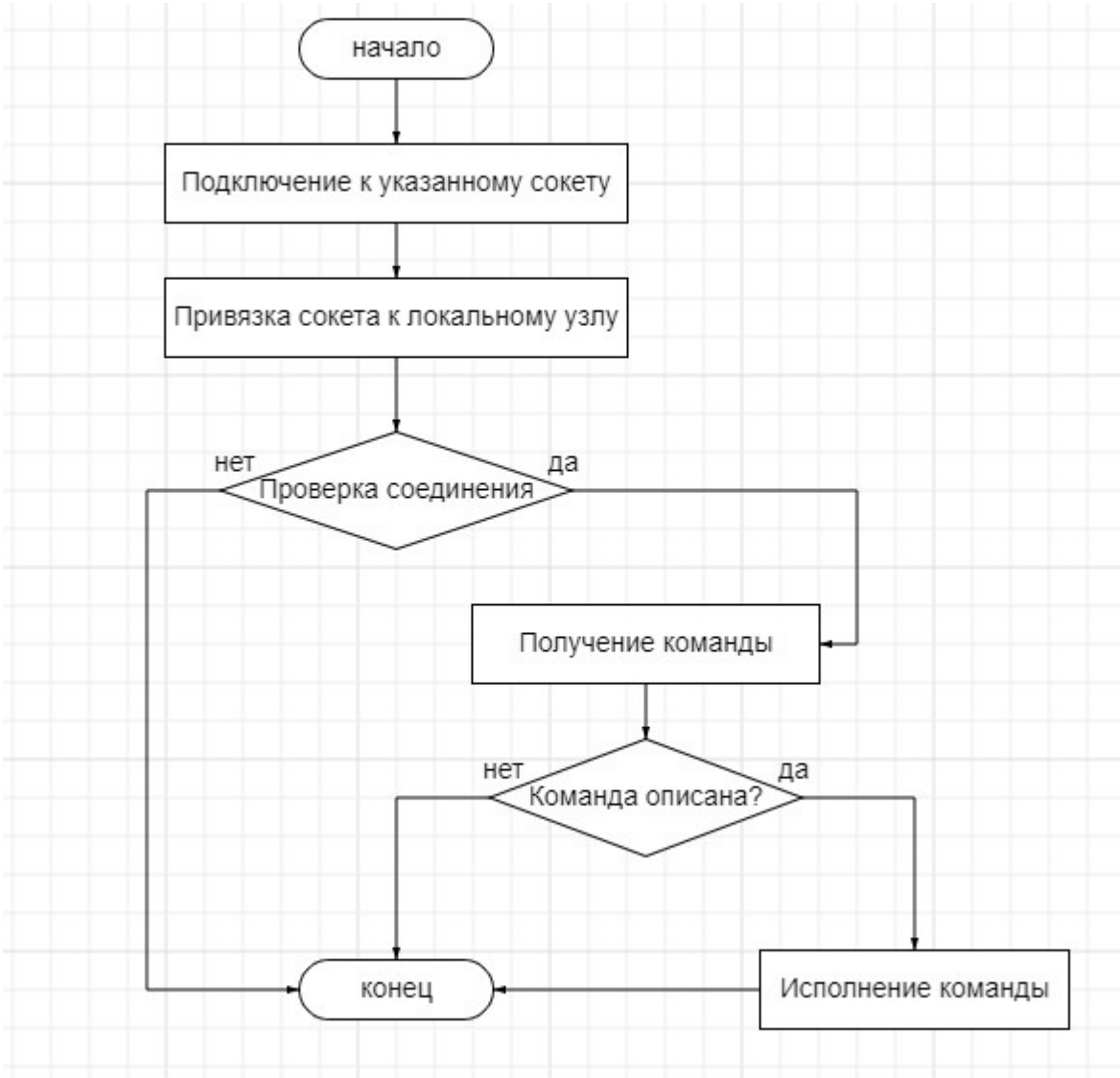

Рисунок 2. Блок-схема подключения сервера и клиента

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Виртуальная энциклопедия Linux. URL: http://rus-linux.net/kos.php?name=/papers/proc/proc_lin.html (дата обращения 25.11.2020)
2. Портал информатиков для гиков. URL: http://espressocode.top/socket-programming-cc/ (дата обращения 25.11.2020)
3. “IBM” (портал посвящённый IT сфере). URL: https://www.ibm.com/developerworks/ru/library/l-dynamic-libraries/ (дата обращения 25.11.2020)

Исходный код

cookie.h

```
#ifndef COOKIE_H
#define COOKIE_H

extern "C" int cookie();
#endif
```

clearcookie.h

```
#ifndef COOKIE_H
#define COOKIE_H

extern "C" int clearcookie();
#endif
```

clearcookie.cpp

```
#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>
#include <unistd.h>
#include <stdio.h>
#include <memory.h>
#include <string.h>
#include <errno.h>
#include <iostream>
#include <fstream>
#include <ctime>

using namespace std;

#include "clearcookie.h"

extern "C" int clearcookie()
{
                  ofstream outf;

        outf.open("Cookie");
        outf.close();
   return 0;
}
```

cookie.cpp

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdio.h>
#include <memory.h>
#include <string.h>
#include <errno.h>
#include <iostream>
#include <fstream>
#include <ctime>

using namespace std;

#include "cookie.h"

extern "C" int cookie()
{
 char * langPtr = getenv("USERNAME");
                  ofstream outf;

        outf.open("Cookie", ios::app);
        time_t rawtime;
  struct tm * timeinfo;

  time( &rawtime );
  timeinfo = localtime ( &rawtime );

        outf<<langPtr<<": "<<asctime (timeinfo)<<endl;
        return 0;
}
```

clearcookie.cpp

```
#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>
#include <unistd.h>
#include <stdio.h>
#include <memory.h>
#include <string.h>
#include <errno.h>
#include <iostream>
#include <fstream>
#include <ctime>

using namespace std;
```

```
#include "clearcookie.h"

extern "C" int clearcookie()
{
                  ofstream outf;

        outf.open("Cookie");
        outf.close();
   return 0;
}
```

lab4c.cpp

```
 #include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <memory.h>
#include <stdio.h>
#include <cmath>
#include <iostream>
#include <sys/time.h>
#include <stdlib.h>
#include <cstdlib>
#include <ctime>
#include <stdio.h>
#include <fstream>
#include <string.h>
#include <dirent.h>
#include <fcntl.h>
#include <pwd.h>
#include <stddef.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#include <malloc.h>
#include <sys/wait.h>
#include <unistd.h>
#include <csignal>
#include <signal.h>

using namespace std;

void spravka()
{
cout«"Авторы: Русманов Владислав ИВ-822, Крюков Илья ИВ-822,
Сысоева Анастасия ИВ-822"«endl;
cout«"Краткое описание проекта:"«endl;
cout«"Данная программа позволяет выполнять простые действия по
работе с процессами Linux"«endl;
cout«" Чтобы получить функционал по работе с файловой системой,
```

```
нужно запустить программу с ключом -l"«endl;
cout«" Чтобы запустить любую операцию сначала необходимо указать
ключ данной операции, а затем режим фона "«endl;
cout«" -d Запускает операцию в фоновом режиме" «endl;
cout«" -n Запускает операцию в не фоновом режиме" «endl;
cout«" -s Порождает новый процесс" «endl;
cout«" -t Режим получения и обработки сигналов"«endl;
cout«"Все доступные аргументы:"«endl;
cout«" -l"«endl;
cout«" -d"«endl;
cout«" -n"«endl;
cout«" -s"«endl;
cout«" -t"«endl;
cout«"Примеры запуска:"«endl;
cout«" ./lab3 —l -h"«endl;
cout«" ./lab3 -s -n proc"«endl;
cout«" ./lab3 -s -d proc"«endl;
cout«" ./lab3 -t"«endl;
}




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

   if (strcmp(argv[1],"-h")==0) spravka();

   if (strcmp(argv[1],"--help")==0) spravka();



      int s = socket( AF_INET, SOCK_STREAM, 0 );
      if(s < 0)
      {
              perror( "Error calling socket" );
              return 0;
      }


      struct sockaddr_in peer;
      peer.sin_family = AF_INET;
      peer.sin_port = htons( 18666 );
      peer.sin_addr.s_addr = inet_addr( "127.0.0.1" );
      int result = connect( s, ( struct sockaddr * )&peer,
sizeof( peer ) );
      if( result )
      {
              perror( "Error calling connect" );
              return 0;
      }
```

```
      result = send( s, argv[1], 2, 0);

      if (strcmp(argv[1],"-s")==0)
      {
        send( s, argv[2], 29, 0);
        send( s, argv[3], 29, 0);
      }

      if (strcmp(argv[1],"-o")==0)
      {
        send( s, argv[2], 29, 0);
        send( s, argv[3], 29, 0);
        send( s, argv[4], 29, 0);
      }
      if( result <= 0 )
      {
              perror( "Error calling send" );
              return 0;
      }

      if( shutdown(s, 1) < 0)
      {
              perror("Error calling shutdown");
              return 0;
      }


      fd_set readmask;
      fd_set allreads;
      FD_ZERO( &allreads );
      FD_SET( 0, &allreads );
      FD_SET( s, &allreads );
      for(;;)
      {
              readmask = allreads;
              if( select(s + 1, &readmask, NULL, NULL, NULL ) <=
0 )
              {
                      perror("Error calling select");
                      return 0;
              }
              if( FD_ISSET( s, &readmask ) )
              {
                      char buffer[20];
                      memset(buffer, 0, 20*sizeof(char));
                      int result = recv( s, buffer,
sizeof(buffer) - 1, 0 );
                      if( result < 0 )
                      {
                              perror("Error calling recv");
                              return 0;
                      }
```

```
                        if( result == 0 )
                        {
                                printf("Server disconnected\n");
                                return 0;
                        }
                        if(strncmp(buffer, "1", 1) == 0)
                                printf("Got answer. Success.\n");
                        else
                                perror("Wrong answer!");
                }
                if( FD_ISSET( 0, &readmask ) )
                {
                        printf( "No server response" );
                        return 0;
                }
        }
        return 0;
}
```

lab4s.cpp

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdio.h>
#include <memory.h>
#include <string.h>
#include <errno.h>
#include <iostream>
#include <fstream>
#include <dlfcn.h>
 #include <cmath>
#include <iostream>
#include <sys/time.h>
#include <stdlib.h>
#include <cstdlib>
#include <ctime>
#include <stdio.h>
#include <fstream>
#include <string.h>
#include <dirent.h>
#include <fcntl.h>
#include <pwd.h>
#include <stddef.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#include <malloc.h>
#include <sys/wait.h>
#include <unistd.h>
#include <csignal>
#include <signal.h>
```

```
using namespace std;

int slt(char *arv1, char *arv2, char *arv3) {
    const char *path = "lab2";
    int status;
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        execl(path, path, arv1,arv2,arv3, NULL);
        _exit (EXIT_FAILURE);
    } else if (pid < 0)
        status = -1;
    else if (waitpid (pid, &status, 0) != pid)
        status = -1;
    return status;
}

int sp(char *d ,char *path) {
    int status;
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        if(strcmp(d,"-d")==0) daemon (1, 0);
        execl(path, path, NULL);
        while (1) {
            if (waitpid(pid, &status, 0) == pid)
                _exit (EXIT_FAILURE);
        }
    } else if (pid < 0)
        status = -1;
    else if (waitpid(pid, &status, 0) != pid)
        status = -1;

    return status;
}



int main(int argc, char **argv)
{
    pid_t pid;
    pid = fork();
    daemon (1, 0);
      int s = socket(AF_INET, SOCK_STREAM, 0);
      if(s < 0)
      {
              perror("Error calling socket");
              return 0;
      }
```

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(18666);
addr.sin_addr.s_addr = htonl(INADDR_ANY);
if( bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0 )
{
        return 0;
}

if( listen(s, 5) )
{
        perror("Error calling listen");
        return 0;
}

int s1 = accept(s, NULL, NULL);
if( s1 < 0 )
{
        perror("Error calling accept");
        return 0;
}

char buffer[31];
char buffer1[31];
char buffer2[31];
char buffer3[31];
int counter = 0;
for(;;)
{
        memset(buffer, 0, sizeof(char)*31);
        int rc = recv(s1, buffer, 30, 0);
        if (strcmp(buffer,"-s")==0)
        {
            recv(s1, buffer1, 30, 0);
            recv(s1, buffer2, 30, 0);
        }
        if (strcmp(buffer,"-o")==0)
        {
            recv(s1, buffer1, 30, 0);
            recv(s1, buffer2, 30, 0);
            recv(s1, buffer3, 30, 0);
        }
        if( rc < 0 )
        {
                if( errno == EINTR )
                        continue;
                perror("Can't receive data.");
                return 0;
        }
        if( rc == 0 )
                break;
        printf("%s\n", buffer);
```

```
               if (strcmp(buffer,"-l")==0)
               {
                        void *handle =
dlopen("./libcookie.so",RTLD_LAZY);
                       if (!handle)
                        {
                            fputs(dlerror(), stderr);
                         //return 5;
                    }
              int (*fun)();
              fun = (int (*)())dlsym(handle,"cookie");
              int x = (*fun)();
              dlclose(handle);
               }

               if (strcmp(buffer,"-d")==0)
               {
                          void *handle =
dlopen("./libclearcookie.so",RTLD_LAZY);
                         if (!handle)
                   {
                   fputs(dlerror(), stderr);
                   //return 5;
              }
              int (*fun)();
              fun = (int (*)())dlsym(handle,"clearcookie");
              int x = (*fun)();
              dlclose(handle);
               }

               if (strcmp(buffer,"-o")==0) slt(buffer1, buffer2,
buffer3);

               if ((strcmp(buffer,"-s")==0) &&
((strcmp(buffer1,"-d")==0) || (strcmp(buffer1,"-n")==0)))
sp(buffer1, buffer2);

         if (strcmp(buffer,"-h")==0) printf("Client want to see
our names...");

         if (strcmp(buffer,"--help")==0) printf("Client want to
see our names...");
      }
      char response[] = "1";
      if( sendto( s1, response, sizeof(response), 0, (struct
sockaddr *)&addr, sizeof(addr) ) < 0 )
               perror("Error sending response");
      return 0;
}
```