Функция выбора - select

Одна из самых замечательных функций имеющейся в UNIX системах является функция выбора - select. Что же она выбирает, что позволяет нам применять её в сетевом программировании? И главное: как она нам поможет правильно обработать приём и отправку данных, а также подключение к удалённому узлу с нужными режимами ожидания?

Данная функция позволяет выбрать один или несколько файловых дескрипторов, которые изменили некоторое своё состояние на "готовое" для выполнения операции ввода или вывода. Казалось бы, что в этом такого, но всё по порядку. Во-первых, напомним, что сокет является файловым дескриптором согласно идеологии UNIX. Поэтому если речь идёт о дескрипторах, то сокеты здесь к месту. Вся суть работы этой функции в том, что она позволяет узнать произошло ли что-нибудь с нашим дексриптором в отведённый промежуток времени. Функция блокируется на определённый промежуток времени до момента, когда произойдёт: либо смена состояния одного из дескрипторов, либо выйдёт отведённое время, в зависимости что наступит раньше.

Дескрипторов можно задать несколько, причём делятся они на три группы:

- дескрипторы для ожидания чтения
- дескрипторы для ожидания записи
- дескрипторы для ожидания наступления исключения

Для задания каждой группы нам необходимо будет заполнить объект "множество", которое мы заполним. Переменная, которая хранит множества, обозначается как fd_set. В библиотеке предоставлены функции для выполнения стандартных математических операций над множествами:

- FD_ZERO(fd_set * set) - обнулить множество
- FD_SET(int fd, fd_set * set) - добавить в множество идентификатор fd
- FD_ISSET(int fd, fd_set * set) - проверить, установлен ли в множестве идентификатор fd
- FD_CLR(int fd, fd_set * set) - удалить из множества идентификатор fd

С помощью этих операций можно легко настроить нужное нам множество сокетов на анализ функцией select. Отметим, что размеры множеств ограничены и не могут быть более FD_SETSIZE, в Линуксе эта константа равна 1024.

Функция select получает на вход, как уже было упомянуто выше, три множества в виде указателей, которые надо анализировать. Причём можно передать NULL вместо одно из множеств, если такое множество нам не требуется обрабатывать. Также в эту функцию надо передать указатель на структуру timeval, которую мы уже использовали ранее. Если в качестве указателя на структуру timeval передать NULL, то select будет ожидать без ограничений по времени. А вот первым аргументом функции надо передать максимальный дескриптор плюс 1. Заголовок функции select:

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

Для примера мы можем воспользоваться функцией select, чтобы получить от пользователя данные, при этом не зависая в вечном ожидании:

Код 1.15: пример запуска select для консоли

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <memory.h>
#include <unistd.h> // for close

int run(void) {
  char buf[100];
  struct timeval tv;
  fd_set fds;
  int ret;
  tv.tv_sec = 5;
  tv.tv_usec = 0;
  FD_ZERO(&fds);
  FD_SET(0, &fds);

  ret = select(1, &fds, NULL, NULL, &tv);
  if ( ret < 0 ) {
    printf("[-] Select error\n");
    return ret;
  }
  if(ret == 0) {
    return 0;
  }
  ret = read(0, buf, sizeof(buf)-1);
  if(ret < 0) {
    printf("[-] Error reading data\n");
    return -1;
  }
  buf[ret] = '\0';
  printf("You entered: %s\n", buf);
  return 1;
}

int main(void) {
  do {
    printf("[?] Please enter anything: \n");
  } while(run() == 0);
  return 0;
}
```

Данная программа, будет ждать ввода от пользователя в консоль, при этом каждые 5 секунд напоминать о том, что надо ввести данные. Чтобы воспользоваться функцией select, мы создаём множество fd_set. И обнуляем его с помощью вызова FD_ZERO, т.е. делаем множество пустым. Затем добавляем в это множество 0, с помощью функции FD_SET. Согласно стандартам 0 - это файловый дескриптор, отвечающий за ввод данных с клавиатуры в UNIX системах. Соответственно, 1 - отвечает за канал вывода данных, 2 - за канал вывода ошибок. После того, как мы заполнили множество и структуру timeval. Вызываем select, который может вернуть три разных значения:

- отрицательное число в случае ошибки
- 0 в случае наступления конца ожидаемого времени
- и положительное число в случае, если что-то на канал чтения

Если мы достигли времени ожидания функция run вызывается ещё раз для повторного ожидания, если же что-то пришло - читаем и выводим на экран.

Применение select для ожидания прихода данных

Давайте воспользуемся новыми знаниями для решения одной из проблем, с которой столкнулись выше: как настроить работу функции recv, чтобы она ожидала определённое время?

Возьмём наш пример Код 1.6-b, и в качестве сервера запустим утилиту NetCat. Клиент успешно подключится, отправить 4 байта и потом начнёт ожидать приёма ответа. Поскольку утилита NetCat в качестве ответа отсылает то, что мы вводим в консоль, то клиент будет ждать нашего ввода, без каких-либо ограничений по времени. Нашей задачей будет через определённое время остановиться ожидать ответа, написать пользователю, что время истекло и завершить программу.

Оформим наше решение в виде отдельной функции, для дальнейшего использования. Идея заключается в то, что перед чтением данных необходимо узнать, а пришли ли туда данные.

Код 1.16: пример запуска select для сокета

```
int hx_recv(int sock, char * buf, size_t len, int sec = -1, int usec = 0, int flags = 0 ) {
  struct timeval tv;
  int ret;
  fd_set fds;

  if(sec < 0) {
    return recv(sock, buf, len,0, flags);
  }
  tv.tv_sec = sec;
  tv.tv_usec = usec;
  FD_ZERO(&fds);
  FD_SET(sock, &fds);

  ret = select(sock+1, &fds, NULL, NULL, &tv);
  if ( ret < 0 ) {
    return ret;
  }
  if(ret == 0) {
    return -2;
  }
  return recv(sock, buf, len, flags);
}

int run(void) {

  // подлючение и отправка PING

  res = hx_recv(sock, buf, sizeof(buf)-1, 5, 0);
  if(res == -1) {
    printf("[-] Failed to receive data from remote\n");
    close(sock);
    return -5;
  }
  if(res == -2) {
    printf("[-] Time out of 5 seconds expired, no data from remote\n");
    close(sock);
    return -5;
  }
  buf[res] = '\0';
  printf("[+] Successfully received %d bytes: %s\n", res, buf);

  // завершение
```

```
}
```

Новая функция принимает на вход теперь 5 аргументов, последние два из них отвечают за ожидание как и раньше - секунду и микросекунды. Если секунды отрицательные, то поведение будет совпадать с обычным вызовом recv, иначе будет работать с ожиданием.

Чтобы воспользоваться функцией, мы создаём множество fd_set на чтение и записываем туда, наш сокет, соответственно. Вызываем select с ожиданием и нашим множеством с сокетом. И как и ранее получаем три возможных ситуации:

- отрицательное число - произошла ошибка
- 0 - наступил конец ожидаемого времени
- положительное число в случае прихода данных на сокет

В последнем случае мы можем спокойно вызвать функцию recv, поскольку теперь мы уверены, что блокировки на ней не возникнет. Собственно на этом и всё, hx_recv полностью справляется с поставленной задачей. Функция возвращает -2 в случае наступления времени ожидания, во всех остальных случаях она возвращает значение, которая возвращает библиотечная функция recv.

! Замечание ! Иногда из некоторых соображений можно заранее знать, что ответ от сервера будет более долгим чем при подключении. Например, Вы запросили поискать в базе данных какие-нибудь записи, и знаете, что это может занять больше времени чем обычно. Поэтому применение разных времён ожидания в разные моменты работы программы оправданы. Более того иногда канал связи может становится то быстрее то медленнее в рамках одного и того же соединения, например, машина заехала в тоннель и качества связи резко ухудшилось.

Ожидание сразу двух

Усложним задачу: пусть требуется одновременно обрабатывать и сокет, и данные от пользователя, вводимые с клавиатуры. То что ввёл пользователь надо отправить по сокету, то что пришло по сокета - напечатать на экран. Можно сказать интерактивная программа общалка. Как быть в этом случае?

Можно, конечно, создать два процесса и каждый из них будет заниматься своей задачей, но при этом возникнет трудность с обменом данными между двумя этими процессами. Однако, для решения поставленной задачи не требуется таких усложнений как работы с двумя процессами, здесь вполне справится уже нам знакомая функция select. Тем более в данной задаче нет особо никаких задач требующих параллельных вычислений, а всего лишь ожидания. Ожидать же можно одновременно.

Итак, суть проста: создадим множество на чтение, но в него положим сразу два дескриптора - сокет и чтение с клавиатуры. Далее воспользуемся select'ом для выбора по какому каналу пришли данные и выполним соответствующее действие. Завершение программы будет в том случае если пользователь прекратит ввод (в консоли нажмёт Ctrl+D).

Код 1.17: интерактивный обмен между пользователем и удалённым узлом

```
int run(void) {
  int sock, res;
```

```
  char buf[100];
  fd_set fds;

  // connect to 127.0.0.1 and port 1234
  sock = hx_connect_v4("127.0.0.1", 1234);
  if (sock < 0 ) {
    printf("[-] Connection error\n");
    return -1;
  }
  printf("[+] Connection successfull\n");

  while(1) {
    FD_ZERO(&fds);
    FD_SET(sock, &fds);
    FD_SET(0, &fds);

    res = select(sock+1, &fds, NULL, NULL, NULL);
    if ( res < 0 ) {
      printf("[-] Select failed\n");
      close(sock);
      return -5;
    }
    if(res == 0) {
      continue;
    }
    // data arrived to stdin
    if(FD_ISSET(0, &fds)) {
      res = read(0, buf, sizeof(buf));
      if(res < 0) {
        printf("[-] Read failed on stdin\n");
        close(sock);
        return -6;
      }
      if(res == 0) {
        printf("[-] User sent end of input\n");
        close(sock);
        return -7;
      }
      if(send(sock, buf, res, 0) < 0) {
        printf("[-] Failed to send to remote\n");
        close(sock);
        return -8;
      }
    }
    // data arrived to socket
    if(FD_ISSET(sock, &fds)) {
      res = recv(sock, buf, sizeof(buf), 0);
      if(res < 0) {
        printf("[-] Read failed on socket\n");
        close(sock);
        return -9;
      }
      write(1, buf, res);
    }
  }

  close(sock);
  return 0;
}
```

В данном примере мы задаём сразу два дескриптора в множество, и поскольку сокет всегда больше 0, то максимальным дескриптором будет он - первый аргумент в функцию select. Также стоит обратить внимание на то, что мы передаём NULL пятым аргументом, что означает, что select будет работать без какого либо предельного времени ожидания.

Основная идея в том, что select разблокируется в тот момент, когда на любой из дескриптором что-то придёт. Поэтому при выходе из ожидания, надо понять какой же дескриптор готов, или даже оба. Поэтому при успешном завершении select, множество fds будет в себе содержать только готовые дескрипторы и нам потребуется лишь проверить лежит ли определенное значение внутри. Библиотечная функция FD_ISSET как раз и выполняет требуемую проверку.

Если что-то пришло с клавиатуры, мы читаем и отправляем по сокету, если что-то пришло по сокету - мы принимаем и печатаем на экран. После чего мгновенно заново становимся в режим ожидания.

!Замечание! Данный код имеет размер буфера 100 байт, и может сложиться впечатление, что программа ограничена максимальным размером пересылки в 100 байт. Однако это не так. Потому что если пришло 200 байт, а мы прочитали 100, то оставшиеся 100 останутся в очереди и на них сработать ещё раз select. Поэтому можно считать, что данный код достаточно надёжен для организации такой интерактивной программы.

Обратите внимание, что код 1.17 есть ни что иное, как программная реализация Команды 1.3

Использование пулов вместо выбора

Функция select является незаменимой при программировании сетевых операций, однако, есть ряд моментов связанные с работой функции select , которые могут привезти к медленной работе программы или вообще сбоям. В современных UNIX подобных системах, Linux, FreeBSD, Solaris, а также и Windows есть несколько механизмов обработки событий, которые можно применять для наших задач связанных с сетевыми соединениями. Функции select является самой старой и самой надёжной из всех, поэтому примеры выше использовали именно её. Можно считать, что на любой ОС где есть сокеты, есть и механизм select (даже в очень старых ОС). И при программировании всяких аппаратных средств также можно быть уверенным, что вот select уж точно есть. Однако, у этого механизма есть ряд недостатков, которые не были учтены на заре разработки сокет библиотеки и ядра ОС. Для совместимости со старыми программами работу функции селект менять не стали, а разработали новые подходы, которые на данный момент рекомендуются в замену старой парадигме select.

Какими же недостатками обладает функция select? Начнём с тех, которые читатель мог уже сам обнаружить при первом же применении этой функции:

- количество максимально ожидаемых дескрипторов ограничено константой FD_SETSIZE, которая равна 1024 в Линуксе, конечно, кажется, что 1024 вполне большое число, но для некоторых протоколов или при высоконагруженных серверах, это довольно мало;
- более того максимальный номер сокета должен быть меньше ограничения FD_SETSIZE, т.е. даже ожидание одного сокета но с номером, скажем 1027, уже не сработает;
- необходимо посчитать максимальный номер дескриптора и передать в функции, не так критично, но не удобно при разработке;
- более критичным является то, что после того, как select сработал, выяснить какой из дескрипторов изменился можно только полным перебор множества, которые мы задали. Когда элементов в множестве много это может замедлить работу программы;
- также множества которые мы наблюдаем, надо каждый раз переинициализировать при новом вызове select, поскольку они модифицируются;
- ожидания происходит только два типа событий (третий вариант "исключений" почти нигде не используются): чтения и запись, и нет возможности ожидать другие события.

В менее очевидные недостатки, которые можно увидеть экспериментально или проанализировав работу функции, входят:

- большая нагрузка на ЦП при ожидании большого набора дескрипторов, в результате чего низкая скорость выполнения этой функции;
- плохая совместимость с многопоточностью. К примеру поведение функции select неопределено, если ожидаемый сокет был закрыт в другом потоке или по сокету было что-то отправлено пока он висит на select;

Для решения части этих проблем был придуман более новый механизм пулов: poll. Эта функциональность существует достаточно давно (где-то с 2000х годов), поэтому можно смело ей пользоваться. Принцип работы пула простой: задаём массив ожидающих дескрипторов и запускаем ожидание изменения состояния. Интерфейс более понятен, чем у select'а.

Для примера перепишем Код 1.17 с помощью пула.

Код 1.47: пример работы пула

```
#include <poll.h> // for poll

...

int run(void) {
  int sock, res;
  char buf[100];
  struct pollfd fds[2];

  // connect to 127.0.0.1 and port 1234

  fds[0].fd = 0;
  fds[0].events = POLLIN;

  fds[1].fd = sock;
  fds[1].events = POLLIN;

  while(1) {

    res = poll(fds, 2, -1);
    if ( res < 0 ) {
      printf("[-] poll failed\n");
      close(sock);
      return -5;
    }
    if(res == 0) {
      continue;
    }
    // data arrived to stdin
    if(fds[0].revents & POLLIN) {
      res = read(0, buf, sizeof(buf));
      if(res < 0) {
        printf("[-] Read failed on stdin\n");
        close(sock);
        return -6;
      }
      if(res == 0) {
        printf("[-] User sent end of input\n");
        close(sock);
        return -7;
      }
      if(send(sock, buf, res, 0) < 0) {
        printf("[-] Failed to send to remote\n");
        close(sock);
        return -8;
      }
    }
    // data arrived to socket
    if(fds[1].revents & POLLIN) {
      res = recv(sock, buf, sizeof(buf), 0);
      if(res < 0) {
        printf("[-] Read failed on socket\n");
        close(sock);
        return -9;
      }
      write(1, buf, res);
    }
  }

  // ...
```

```
}
```

Сперва создаём массив из наблюдаемых дескрипторов, это структура типа `struct pollfd`. Которая, содержит три поля: `fd` - дескриптор, `events` - битовое поле событий, которые хотим ожидать, `revents` - битовое поле произошедших событий, которое будет заполнено после вызова. В данном случае оба дескритора должны ожидать событий прихода данных (чтение), поэтому будем ожидать события POLLIN. Обратите внимание, что массив дескрипторов заполняется до бесконечного цикла опроса. Его не надо переинициализировать каждый раз перед вызовом опроса событий в отличие от select. Далее идёт вызов функции poll, которая принимает массив, зармер массива и третий аргумент время ожидания. Отрицательное время ожидания - это ожидание без ограничения по времени. Как только по одному из дескриптров произойдёт событие чтения (или иное), то функция poll завершится. Далее возникает как и с select три случая: ошибка, ничего не произошло, и наступило событие. Первые два случая обрабатываются как и ранее, а вот проверка по какому каналу наступило событие нескольку другая. Необходимо проверить поле `revents` выставлен ли нужный бит события чтения в нём или нет. После чего можно обрабатывать канал.

Отметим, что в функции poll время ожидания задаётся одним числом - целым числом, и измеряется в милисекундах. Конечно, это менее приятно, поскольку в select можно передать время с точностью до микросекунд и даже наносекнуд. Но на самом деле, время в таких вызовов всё равно округляется с точностью до гранулированности времени в ядре ОС. Т.е. даже если мы зададим очень мало время, ОС не сможет обработать события с такой точностью, ввиду устройства ядра операционной системы. Такая точность может быть достигнута только в специальном классе операционных систем - операционные системы реального времени. Более подробно о таких системах можно узнать в [Богачёв]. Поэтому с практической точки зрения на обычной ОС, задачние точности по времени в милисекнудах вполне достаточно. Но всё же стоит отметить преимущество в более точном задании времени ожидания в функции select над poll.

Преимущества пула

Пример Кода 1.47 показывает, как можно ожидать два дескриптора на чтение. Для ожидания события на запись необходимо выставить бит POLLOUT в поле events. Помимо этого можно для одного дескриптора выставлить одновременное ожидание сразу нескольких событий:

```
fds[0].events = POLLIN | POLLOUT;
```

И при обработке сразу смотреть какие события произошли.

Если разбираться более детально, то можно обнаружить и некоторые более приятные особенности при использовании пула. Когда пул сообщил об изменении события,то в результате на одном из дескрипторов может произойти событие, которое мы не заказывали. К числу таких событий относятся: POLLHUP, POLLERR и POLLNVAL. Эти события бессмысленно выставлять в поле events, они выставляются автоматически только в выходном поле revents. Служат они для оповещения о специальных состояниях:

- POLLHUP - означает, что соединение по сокету было закрыто удалённым узлом
- POLLERR - возникла ошибка, к примеру мы проводили запись данных по соекту, но удалённый узел закрыл приём данных

- POLLNVAL - неверный дескриптор: не откыт файл или сокет

Наиболее часто используемым можно считать первое событие POLLHUP. С помощью него можно определить, что соединение было закрыто. В случае использования селекта, для определения такого что удалённый узел закрыл соединение пришлось бы:

- ожидать прихода в множестве на чтение;
- сделать попытку чтения и получить 0 байт.

Это менее удобно поскольку надо делать дполнительный системный вызов на чтение, а если ещё и при этом мы на этом сокете только пишем данные, но не читаем, нам придётся заполнять отдельное множество ожидания на чтение, и также обрабатывать эти события в цикле.

Помимо этого можно сравнить скорость работы select и poll. Для этого достаточно в цикле ожидать наступления некоторого события и обработки его. Генерировать эти события можно даже не прибегая к сокету, а просто читая файл, который постоянно будет поставлять данные. Для таких целей к примеру можно взять файл /dev/zero.

> Кстати, существует ситуация, когда select и poll может сообщить о приходе данных на сокет, но последующее чтение будет всё равно заблокриовано. Такая ситуация может возникнуть в случае, когда пакет с данными пришёл на сокет, но после его обработки он был отброшен, например, по причине неправильной контрольной суммы.
>
> Чтобы учесть данный случай необходимо использовать, неблокируемый сокет при чтении.

Как видим, poll имеет некоторые преимущества перед select'ом, однако, полностью решить все недостатки не может. А именно:

- нам всё также приходится опрашивать в цикле на каких дескрипторах наступуло то или иное событие;
- сохраняются проблемы с многопоточностью, т.е. нельзя динамически менять заданные события, когда программа уже запустила функцию poll.

Для борьбы с этими проблемами существуют более совершенные механизмы, такие как: kqueue, epoll, libevent, iocp. Но об этих механизмах мы поговорим позже, когда они потребуются. Более того эти механизмы менее универсальны и поэтому они зависят от ОС, на которой будет запущена программа. Если стоит задача писать кросс-платформенную программу, то обычно сначала пишется версия программы с обычным select или poll, а затем уже более сложные механизмы для увеличения производительности. Но при написании обычных(не высоконагруженных программ и не программ, работающих с одноранговыми сетями и подобными) описанных ранее select и poll вполне достаточно.

Глава 2

О сервере

При работе с передачей данных мы считаем, что контакт между двумя узлами уже налажен. Однако, процесс наладки данного канала строится на основе "клиент-сервер" подхода, в котором клиент подключается к серверу. В данном случае клиент инициирует соединение, поэтому можно считать, что инициатива исходит именно от того кто запускает клиентское приложение. А серверная часть оказывает услугу для данного запроса. Как правило, данная услуга может оказываться сразу нескольким клиентам, даже одновременно. Мы говорим тут "как правило", потому что создание "серверного" или прослушивающего сокета может быть только для одного клиента. К примеру, такой подход сделан для построения канала данных в протоколе FTP. Однако, в большинстве случаев прослушивающий сокет нужен для организации сервисов, к которым могут подключаться множество клиентов. И тут возникает, задача корректной обработки входящих клиентов.

Некоторые моменты, на которые стоит обратить внимание, при проектировании серверного приложения:

- одновременность обработки клиентов - новые клиенты не должны долго ждать пока сервер обработает текущего клиента.
- если сервис не настроен на взаимодействие между клиентами, то каждый клиент должен считать себя изолированным. Проще говоря, клиент должен ощущать себя единственным на сервере, как будто остальных нет. Кстати, при программировании сервера это тоже имеет место быть, поскольку позволяет написать процедуру обработки клиентов отдельно.
- отказоустойчивость от "злоумышленника" - зависание одного клиента не должно приводить к отказу обрабатывать других.
- сходбищеустойчивость - при массированном подключении большого числа "злоумышленных" клиентов, продолжать обработку добросовестных клиентов.

Для решения поставленной задачи необходимо учитывать вышеупомянутые пункты/требования. Чем больше пунктов мы удовлетворяем, тем более сложен программных код сервера. Более того, в современном мире программирование хорошего отказоустойчивого сервиса может потребовать очень много усилий и программных решений, которых в рамках данного пособия мы не в состоянии описать. Такими программными продуктами, которые разрабатываются уже десятки лет, к примеру являются Вебсервер nginx или Apache, DNS сервер bind, БД Mysql/Oracle.

Все проблемы обсуждаемые в данном разделе будут крутиться вокруг функции accept, которая принимает нового клиента на обработку сервером. Будем считать, что проблемы связанные с пересылкой данных между подключённым клиентом и сервером нас не будут интересовать.

Для анализа рассмотренных решений будем применять различные подходы в том числе и строить специального клиента "злоумышленника", который будет намеренно выводить работу сервиса из строя, тем самым выявлять слабые места в программном коде.

Сервер A: последовательная обработка

Это самый простой серверный обработчик клиентов - его суть заключается в том, что:

- мы принимаем клиента, его обрабатываем и только после этого готовы принять следующего.

Код 2.1 как раз и является примером такого сервера. Для более универсального вида мы функцию, которая будет обрабатывать клиента, можем передать в качестве аргумента основной функции. В результате чего получим вполне универсальное решение для сервера:

Код 2.2: универсальный "эхо" сервер - последовательная обработка

Далее мы функцию `hxp_echo_server` трогать не будем. Интерфейс `hx_server_v1` функции достаточно прост - мы указываем порт на котором ожидать соединения и функцию-обработчик клиента. Вся сложность обработки будет спрятана внутри функций вида hx_server_*.

Как видим, пока функция `client_processor` не завершится новый клиент не будет принят. С одной стороны код сервера прост и прозрачен, но при этом имеет серьёзные недостатки. Последовательное выполнение обработки клиентов исключает одновременность работы с несколькими, что для серверных приложение критично. Даже самое быстрое железо может оказаться бесполезным в ускорении работы такой программы. Несмотря на то, что код обработки клиента выполняется очень быстро и заметить "торможение" данного кода при подключении многих "хороших" клиентов будет непросто, достаточно одного злоумышленника, чтобы вывести всю систему из строя. А именно, в "эхо" коде первой операцией является приём данных, а как мы знаем, ожидание прихода данных в функции recv выполняется "бесконечно". Следовательно злоумышленнику достаточно подключиться и ничего не отсылать при этом не разрывать соединение. В этом случае сервер повиснет на ожидании и не будет обрабатывать новых клиентов. Конечно, можно добавить максимальное время ожидания на recv, но даже 5 секундная задержка существенно замедлит обработку "хороших" клиентов.

Последовательная обработка клиентов полностью непригодна в случае "сессионных" клиентов.

"Сессионный" клиент

> Во многих протоколах обработка клиентов осуществляется в режиме сессий. Т.е. Клиент подключается, если надо проходит авторизацию, и начинает общение с сервером. Такими протоколами к примеру являются FTP, POP3, IMAP.
>
> При таком протоколе клиент может быть подключён к серверу достаточно долго пока клиент не решит завершить сессию.

Более того последовательная обработка выполняется только на одном ядре процессора и не использует серверное оборудование (которое как правило многоядерное) на все 100%. И тем не менее существуют случаи "серверов", когда такое решение вполне пригодно и более того является предпочтительным.

Такими случаями являются ситуации, когда:

- клиент не влияет на решения сервера;

- код обработчика клиента выполняется быстрее, чем какая-либо процедура по постановке в очередь или обработке клиента. Т.е. вызов `client_processor` сопоставим по времени с вызовом к примеру функции fork/create_thread (см. ниже).

В качестве примера такого "быстрого" сервера можно привести программу, которая информирует клиентов о некотором состоянии. Причем поскольку приём данных (вызов recv) от клиента, как видим ранее, невозможен, то состояние является одним для всех клиентов, иными словами выдаёт универсальный ответ. Например, "сервер" выдающий точное текущее время, или информацию о погоде.

Сервер B: последовательная обработка с выбором обрабатываемого клиента

Попробуем устранить недостаток, создаваемый "злоумышленником" для сервера A. Если злоумышленник подключился и ничего не высылает, то система повиснет на приёме данных, когда в это время может придти новый клиент. Тут нам приходит на помощь механизмы выбора нужного дескриптора при изменении его состояния: функция select или poll. Поскольку пул более современный и более удобный механизм, воспользуемся им.

Код 2.3: универсальный "эхо" сервер - последовательная обработка с выбором

Поскольку при работе сервер A при подключении зависает на функции recv, то для решения этой проблемы достаточно предварительно узнать - появились ли от клиента какие-нибудь данные. На помощь приходил пул, однако, поскольку количество клиентов заранее неизвестно. Мы используем "список" клиентов в данном случае массив. Для удобства мы заведём массив на фиксированное количество клиентов - 100, это ограничение можно убрать запрограммировав динамический массив. Оставим, это в качестве упражнения читателю. Также отметим, что если пользоваться функцией select, то там есть физическое ограничение количество сокетов, поэтому пул лучше.

Поскольку нам необходимо ожидать поступления нового клиента и приход данных от уже подключённых, то в массиве для пула на нулевом месте будет серверный сокет. Все остальные будут находится после. Как только происходит изменение на серверном сокете - т.е. приходит новый клиент. Мы его добавляет в наш список для пула, на первое свободное место. И если надо увеличиваем размер списка. Далее, если произошли изменения на одном из клиентских сокетов, то мы запускаем процедуру обработки, после чего удаляем сокет из списка, путём выставления в качестве дескриптора отрицательного числа. В документации пула сказано, что если файловый дескриптор отрицательный, то это значение игнорируется. Что довольно удобно в нашем случае.

Таким образом, данный код прекрасно справляется с проблемой злоумышленника, поскольку в этом случае, сокет будет в ожидании и пул не будет срабатывать на него. Конечно, если придут сразу 100 злоумышленников, то данная система также будет поломана. В этом случае можно отслеживать времена ожидания таких злоумышленников и насильно отключать их.

Большим недостатком такого сервера, является то, что обработка клиентов тут не "изолирована". Т.е. мы предполагаем, что первой процедурой в обработчике клиента будет именно recv. А что

если это не так? Добавления клиентского сокета в пул фактически отменяет принцип чёрного ящика для функции client_processor. Ситуация становится ещё сложнее, если протокол общения с клиентом предполагает несколько recv. В этом случае, необходимо также действовать через пул и при этом помнить и передавать текущее состояние протокола клиента. Здесь поможет модель вычислений под названием конченый автомат или машина состояний. В общем случае решение использовать сервер типа B, сильно зависит от конкретного протокола. Код получается сильно нагромождённым.

Преимуществом является то, что здесь используется только одно ядро процессора, и не создаётся никаких лишних процессов в ОС, что снижает нагрузку (и не требует много ресурсов ОС) на ядро ОС при достаточно многом количестве приходящих и обрабатываемых клиентов.

Сервер C: параллельная обработка с помощью fork

Альтернативным методом борьбы со злоумышленником из сервера A является создание отдельного процесса для его обработки. Таким образом, в этом случае обработка клиента на стороне сервера будет "идеально" изолированной, поскольку фактически каждый клиент будет обработан "отдельной" программой никак не связанной с другой.

В UNIX подобных операционных системах для этого есть механизм создания клонированного процесса с помощью функции fork(). В ОС Windows такого механизма нет, там надо усложнять обработку и использовать CreateProcess с механизмом передачи сокета. Более подробно про создание процессов в UNIX можно прочитать в [Богачёв].

Код 2.4: универсальный "эхо" сервер - обработка через fork

Функция fork создаёт клон текущего процесса. И фактически после строки с вызовом fork у нас есть два процесса, которые отличаются только значением переменной pid. В дочернем процессе она равна нулю, а в родительском содержит идентификатор дочернего. Поэтому в случае родительского мы закрываем клиентский сокет и опять садимся на ожидание нового клиента. В то время как дочерний процесс выполняет функцию обработки клиента и по завершению её выполняет завершается. Таким образом, каждый приходящий клиент будет обработан в отдельном системном процессе, что мы и хотели. Отметим, что для корректного завершения дочерних процессов, чтобы они не повисали в памяти, необходимо на родительском игнорировать сигнал от них вызовом: `signal(SIGCHLD,SIG_IGN)`.

Данная схема параллельной обработки клиентов является классической в том смысле, что за счёт простого решения получается довольно легко вести параллельную обработку клиентов. И достоинством такой схемы, является то, что она может использовать все доступные процессорные ресурсы для успешной обработки клиентов. Клиенты-злоумышленники будут висеть в ожидании и никак не влиять на обработку других клиентов.

Однако, у такой схемы есть и недостатки одним из которых является то, что на каждого клиента тратится довольно дефицитный системный ресурс процесс. Поскольку в ядре ОС есть таблица процессов и она ограничена по размеру, то при массовом нападении она может переполнится и функция fork перестанет работать. Если в случае с сервером A или B, злоумышленник мог испортить работу приложения. То переполнив таблицу процессов, начнёт зависать вся ОС,

поскольку другие приложения невозможно будет запустить. Поэтому по-хорошему необходимо вводить ограничения на количество созданных дочерних процессов, чтобы не забить всю систему.

Также стоит отметить, что функция fork требует некоторого времени и ресурсов для выполнения. Поэтому в случае, когда код обработчика достаточно мал и не требует действий от клиента, сервер A может быть более успешен для решения задачи.

Сервер D: параллельная обработка с помощью threads

В предыдущем варианте для обработки клиента создавался клон текущего процесса, на практике это не всегда удобно поскольку на это может тратится много ресурсов. Помимо этого если потребуется взаимодействие между клиентами, необходимо будет наладить некоторый механизм. Более интересным решением будет использовать технологию нитей или потоков (threads). Поток создаётся в рамках текущего процесса и имеет с ним общую память, которую можно использовать для передачи данных между клиентами, или влиять на сервер. Например, клиент может подать команду на завершение сервера. В этом случае это можно осуществить к примеру через глобальную переменную. Помимо этого потом не клонирует всю память серверного процесса, и тем самым бережёт ресурсы. Более подробно о потоках можно также посмотреть в [Богачёв].

Код 2.5: универсальный "эхо" сервер - обработка через threads

Для создания потока нам необходимо указать стартовую функцию, через аргумент этой функции мы передаём структуру с сокетом и указателем на нашу рабочую функцию. Поскольку потоки могут создаваться в произвольном порядке, даже если поток был создан позже он может начать выполнения раньше, то нам необходимо для каждого потока выделить память под параметры и передать. Таким образом, у каждого потока будет своя ячейка памяти откуда он возьмёт информацию, и потом её удалит. Важно отметить, что при создании потока необходимо его создать в отсоединённом режиме (detached), чтобы после того как он завершится вся информация о нём была очищена. Аналогично при fork мы игнорировали сигнал `SIGCHLD`.