Лекция 4. Клиент-сервер на базе TCP

TCP работает в терминах соединения.

Клиент посылает запрос на установление соединения (connect) — сервер подтверждает и выделяет у себя отдельный канал (сокет) для работы с этим клиентом, клиент же работает со своим изначальным сокетом. Потом клиент или сервер могут послать запрос на закрытие соединения (close)

Пока соединение действует, клиент и сервер могут обмениваться сообщениями по сокетам, связанным с этим соединением.

Иллюстрация: схема работы сервера с несколькими клиентами.

Проблемы:

1. сервер должен как-то отслеживать с кем у него установлено соединение, регистрировать новые соединения и удалять закрытые

2. сервер должен как-то понимать кто из клиентов к нему обращается, т.е. обнаруживать сам факт входящего сообщения от клиента и выстраивать дисциплину обслуживания, если обращается несколько клиентов одновременно

3. обслуживание одного клиента не должно существенно влиять на ожидание обслуживания других клиентов (справедливое разделение времени).

Можно решать разными средствами. Мы рассмотрим системные вызовы select, poll, fork.

TCP сервер на базе select.

По пунктам.

1. Множество рабочих каналов ввода/вывода задается множеством типа fd_set. По сути оно работает как битовое множество:

```
fd_set  iod;
FD_SETSIZE              максимальное количество элементов в множестве
FD_ZERO( &iod );        обнулить множество
FD_SET( k, &iod );      добавить дескриптор k в множество (k-й элемент = 1)
FD_CLR( k, &iod );      удалить дескриптор k из множества (k-й элемент = 0)
FD_ISSET( k, &iod );    проверить установлен ли элемент k в 1
```

Функция select

```
fd_set  read_set, write_set, exc_set;
struct timeval tv;    // секунды и микросекунды
int ret, size = FD_SETSIZE;

ret = select ( size, &read_set, &write_set, &exc_set, &tv );
```

Все структурные параметры функции select могут быть изменены в результате вызова. Поэтому перед следующим вызовом их надо обновлять.

Идея реализации сервера.

Все сокеты, обслуживающие активные соединения от клиентов, и начальный сокет, обслуживающий запросы на установление соединения, регистрируются в множестве типа fd_set.

Копия этого множества отправляется как параметр в вызов функции select, которая оставляет в этом множесте только те дескрипторы каналов, в которых есть данные для чтения.

Далее в этом полученном множестве ищутся ненулевые элементы (т.е. каналы, в который есть что читать) и обрабатываются данные этих каналов (либо accept, либо read|recv).

При закрытии какого-либо соединения соответствующий дескриптор сокета исключается из множества активных дескрипторов и более в проверке не участвует.

Пример. Предположим, что сервер способен ответить на запрос клиента сразу одним сообщением. Но это ответное сообщение может иметь разную длину. Для клиента есть несколько вариантов работы:

— сервер может первым делом сообщить клиенту длину своего сообщения. Тогда клиент создает буфер под эту длину и спокойно прочитает сообщение сервера.

— сервер не сообщает длину, а обозначает конец сообщения некоторым согласованным с клиентом условным кодом. Здесь клиент не может с гарантией предоставить для чтения буфер достаточного размера, но может читать сообщения порциями и проверять наличие специального кода конца.

В обоих случаях схема сервера будет выглядеть примерно так.

```
иициализация сетевых интерфейсов
sock = socket(...);
bind(sock ...);
listen(sock ...);
fd_set active;
fd_set ready;
бесконечный цикл
   ready = active;
   select (..., ready, ...);
   проверка ready на наличие элементов
   если ready содержит sock, то
      newsock = accept(sock ...)
      добавить  newsock в active
   если ready содержит другой дескриптор fd, то
      прочитать запрос клиента из fd
      обработать запрос клиента
      ответить клиенту через fd
      (а если клиент просит завершить соединение?)
      (тогла закрыть fd и исключить его из active)
конец бесконечного цикла
```

Возможны случаи, когда обработка запроса конкретного клиента может занять существенное время. Во время этой обработки сервер не сможет реагировать на запросы других клиентов. В такой ситуации можно разбить ответ одному клиенту на несколько шагов, и между этими шагами пытаться обслуживать других клиентов, если они требуют внимания. Такая тактика потребует введения разных очередей, списков и т.п. по отслеживанию состояния ответов на запросы клиентов и может оказаться нетривиальной. Поэтому в рамках использования select мы не будем это рассматривать, так как подоные проблемы просто и естественно решаются в рамках использования fork.

TCP сервер на базе poll.

Системный вызов poll работает идейно так же как и select, но в некоторых аспектах устроен более удобно. Основное отличие в том, что состояния сетевых каналов по каждому клиенту представлены отдельной структурой pollfd

```
struct pollfd {
   int fd;         // дескриптор файла
   short events;   // запрошенные события
   short revents;  // возникшие события
};
```

и для анализа используется массив таких структур. Мы можем назначить какие события мы хотим проверять для кждого отдельного клиента и также нам не нужно отслеживать максимальный номер сокета для просмотра множества fd_set.

```
pollfd  act_set[100];
act_set[0].fd = sock;
act_set[0].events = POLLIN;   // requested event
act_set[0].revents = 0;       // returned event
```

Здесь канал sock будет проверяться на событие POLLIN, соответствующее появлению данных для чтения.

В плане реализации сценария работы с клиентами схема сервера здесь очень похода на предыдущую. Поэтому просто повторим код предыдущех примеров с соответствующими изменениями.

TCP сервер на базе fork.

Этот вариант сервера очень естественно решает проблему распределения внимания с0ервера между несколькими клиентами, поскольку сводит ее к системной поддерже многозадачности. Однако при этом так же естественно возникают некоторые подводные камни программирования многозадачных приложений, в частности, взаимодействие родительских и порожденных процессов.

Прежде всего рассмотрим смысл самой функции fork. Каждый процесс в системе имеет свой уникальный идентификатор — pid (process idenfificator), который в программе представлен типом pid_t.

Вызов функции fork создает и запускает в системе второй процесс в точности идентичный тому, из которого эта функций была вызвана

```
...
некий код
...                    <-  здесь выполнялся один исходный процесс
pid_t p = fork();      <- вызов в старом процессе, возврат в старом и новом
...                    <-  а здесь уже два - исходный и порожденный
некий код
...
```

Но нам надо после развилки выполнять разные операции в разных процессах. Функция fork в исходном процессе возвращяет pid порожденного процесса, а в порожденном процессе возвращает 0. Таким образом мы можем выяснить в каком процессе мы назодимся и соответственно реализовывать алгоритм.

```
код исходного процесса до развилки
pid_t p = fork();
if (p == 0)  {
   код порожденного процесса
} else {
   код родительского процесса
}
```

Можно предложить разные тактики разветвления процессов.

1. Родительский процесс запускает бесконечный цикл и в этом цикле блокируется на вызове accept, ожидающем запросы на соединения. При срабатывании accept и создании соединения, родительской процесс выполняет fork и порожденный процесс обслуживает соединение с данным клиентом и при закрытии соединения завершает свою работу.

2. Родительский процесс сразу запускает несколько порожденных процессов, которые выполняют у себя бесконечный цикл с блокировкой на accept. Если запросов на соединение нет, то все эти порожденные процессы находятся в блокированном состоии и практически не потребляют ресурсов вычислительной системы. При появлении запроса соединения этот запрос обрабатывается некоторым порожденным процессом, и этот процесс начинает работать с данным клиентом до завершения соединения. Остальные процессы все так же ждут на accept появления новых запросов, и при их появлении подключаются к обслуживанию клиентов.

Обе схемы имеют свои недостатки, и в реальной работе должны быть устроены более хитро с применение разнообразных системных механизмов взаимодействия параллельных процессов.

Проблема зомби. Если порожденный процесс завершается раньше родительского, то он может попасть в состояние “зомби”, когда он остается зарегистрированным с разных системных таблицах, т.е. не удаляется из них, хотя не выполняет никаких действий. Системные процедуры управления и мониторинга тратят некоторые ресурсы на обслуживание этих процессов, хотя это уже не требуется. Чтобы порожденный процесс не превратился в зомби при своем завершении, родительский процесс должен “отцепить его” от себя, выполнив системный вызов wait (или waitpid или т.п.). Поэтому в наших примерах родительскиий процесс сервера предусматривает выполнение таких вызовов после выполнения fork.