

Специфика сетевых программ для Windows

С идейной точки зрения то же самое, но есть ряд технических отличий.

1. Библиотечные функции объявлены в файле `winsoc2.h`, некоторые сервисные функции объявлены в `ws2tcpip.h` (для использования этих функций требуется, например, библиотека `Ws2_32.lib`). Также современные версии Visual Studio, по умолчанию не пропускают многие “устревшие” и небезопасные функции, которые мы использовали в Linux вариантах кода. Поэтому для сохранения единообразия можно включить в код следующие строки, чтобы отключить предупреждения и подключить сетевые библиотеки

```
#pragma warning(suppress : 4996)
#pragma comment(lib, "Ws2_32.lib")
```

2. Необходимо выполнить инициализацию сетевых интерфейсов при помощи вызова `WSAStartup(...)`, а в конце работы освободить ресурсы при помощи вызова `WSACleanup()`.

3. Для обмена используются функции `send` и `recv` (`read` и `write` с сокетами не работают). Для закрытия используется специальная функция `closesocket()`

4. Некоторые функции заменены новыми альтернативными версиями. Например, `gethostbyname` имеет более продвинутый аналог `GetAddrInfo()`. Старые версии подобных функций можно использовать, если определить имя

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
```

5. Структура `fd_set` устроена по-другому. Она представляет собой не битовое множество, а просто массив `fd_array` дескрипторов сокетов максимальной длиной `FD_SETSIZE` (по умолчанию = 64), и текущим количеством зарегистрированных сокетов `fd_count`.

6. Функция `select` работает так же, только первый параметр (длина множеств) не учитывается. Вместо `poll` в современных выпусках windows используется функция `WSAPoll()`.

В остальном примерно то же самое, хотя есть и другие системные функции.

Пример. tcp клиент и сервер с использованием `select`.

TCP сервер на базе fork

Этот вариант сервера очень естественно решает проблему распределения внимания сервера между несколькими клиентами, поскольку сводит ее к системной поддержке 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 identifier`), 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
...
pid_t p = fork();
...
некий код
...
```

<- здесь выполнялся один исходный процесс
<- вызов в старом процессе, возврат в старом и новом
<- а здесь уже два - исходный и порожденный

Но нам надо после развилки выполнять разные операции в разных процессах. Функция `fork` в исходном процессе возвращает `pid` порожденного процесса, а в порожденном процессе возвращает 0. Таким образом мы можем выяснить в каком процессе мы находимся 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, родительский 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 заблокированном состоянии 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 применением 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 в 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 этого 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` и т.п.). Поэтому в наших примерах родительский процесс сервера предусматривает выполнение таких вызовов после выполнения `fork`.

В конспективном виде эти две схемы работы можно представить в следующем виде.

Схема 1 — отдельный процесс на обслуживание каждого клиента

```

int main()
{
    инициализация адресных структур
    sock = socket (PF_INET, SOCK_STREAM, 0);
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (char*)&opt, sizeof(opt));
    bind(sock, ...);
    listen(sock, ...);

    // разветвление - отдельный процесс для каждого клиента
    pid_t pid;
    while (1) {
        // блокирующий режим
        new_sock = accept(sock, ...);
        pid = fork();
        if (pid == 0) {
            // порожденный процесс
            close(sock);
            while ( ProcessClientRequest(new_sock) );
            return 0;
        }
    }
}

```

```

    } else {
        // родительский процесс
        while ( waitpid(-1, nullptr, WNOHANG) > 0 );
    }
}
}

```

```

bool ProcessClientRequest (int cli_socket)
{
    читаем и пишем через cli_socket
    при ошибке и т.п. закрываем cli_socket
    и возвращаем false
    return true;
}

```

Здесь есть небольшая погрешность с wait — вызывается только при возникновении нового запроса на соединение.

Схема 2 — несколько предварительно запущенных процессов

```

int main()
{
    инициализация адресных структур
    sock = socket (PF_INET, SOCK_STREAM, 0);
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (char*)&opt, sizeof(opt));
    bind(sock, ...);
    err = listen(sock, ...);
    // запускаем несколько независимых процессов обслуживания клиентов
    pid_t pid;
    for (int ip = 0; ip < NUM_PROC && (pid = fork()) > 0; ip++);
    if (pid == 0) {
        // порожденный процесс - обслуживание клиентов
        WorkWithClients(sock);
        return 0;
    } else {
        // родительский процесс - ожидание завершения порожденных процессов
        // здесь wait блокируется до завершения порожденного процесса
        while(wait(0)>0);
    }
    return 0;
}

void WorkWithClients (int sock)
{
    ...
    while (true) {
        new_sock = accept(sock, ...);
        обмен данными с клиентом через new_sock
        при завершении соединения - закрываем new_sock
    }
}

```

Общая проблема — fork дорогая операция поскольку копирует всю память процесса.