Лекция 17.

Архитектура сервера

- Типичный сервер должен одновременно обслуживать несколько (или много) клиентов
  - Клиенты могут быть (относительно) независимые (типичные примеры: http, ssh, smtp)
  - Клиенты могут быть взаимодействующими, либо работающими с общими ресурсами (jabber, ...)

Архитектура сервера

- Процесс на клиента
  - (+) простота реализации
  - (-) «тяжелый» процесс, плохо масштабируется
  - (-) сложности взаимодействия
- Процесс (однопоточный) на много клиентов
  - (+) простое взаимодействие
  - (+) намного лучше масштабируется
  - (-) сложность реализации
- Одна нить на клиента — рассмотрим позже

Работа с многими ф. д.

- Единственный процесс должен работать одновременно с многими файловыми дескрипторами
  - Читать из дескриптора как только появляются данные
  - Записывать в дескриптор как только он готов
- **Процесс не имеет права ожидать готовности на одном файловом дескрипторе в ущерб другим!**

Неблокирующийся ввод-вывод

- Сокеты, пайпы, терминалы и другие «медленные» устройства могут быть переведены в неблокирующий режим (O_NONBLOCK)

```
int cur = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, cur | O_NONBLOCK);
```

Неблокирующийся ввод-вывод

- В неблокирующем режиме:
  - read (и accept) возвращает ошибку EAGAIN, если нет данных, немедленно доступных для чтения, то есть если бы обычный read заблокировал процесс при следующей операции чтения
  - write возвращает EAGAIN, если нет места для добавления данных в буфер вывода, то есть write заблокировался бы

O_NONBLOCK и диски

- O_NONBLOCK не работает при дисковом вводе-выводе, то есть тогда, когда в операции записи/чтения вовлекается подсистема управления памятью и буферный кеш
- В случае отсутствия данных в буферном кеше операция чтения заблокирует процесс в состоянии D (uninterruptible sleep)

Опрос файловых дескрипторов

- Переведя файловые дескрипторы в неблокирующий режим можно опрашивать их, ожидая готовности
- Это неэффективное, немасштабируемое решение, которое требует еще и активного ожидания (busy wait)
- **Требуется поддержка со стороны ядра операционной системы — ядро должно разбудить процесс, когда дескрипторы будут готовы, а иначе процесс должен спать и не потреблять ресурсы**

Отслеживаемые события

- Ф. д. готов к выполнению операции чтения, т. е. операция чтения не заблокирует процесс
  - Поступили данные
  - Поступил запрос на подключение
  - Поступил признак конца файла (закрытие соединения)
- Ф. д. готов к выполнению операции записи
- Поступил сигнал
- Тайм-аут (истекло время ожидания)

Множества ф. д.

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

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

- Функции работают с множеством файловых дескрипторов
- Максимальный номер файлового дескриптора в множестве зависит от ОС (обычно 1024)

Системный вызов pselect

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

int pselect(int nfds,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            const struct timespec *timeout,
            const sigset_t *sigmask);
```

- В более старых системах может использоваться системный вызов select
- Кроме того может использоваться системный вызов poll

Системный вызов pselect

- readfds — множество файловых дескрипторов, проверяемых на готовность к операции чтения (допускается NULL)
- writefds — множество файловых дескрипторов, проверяемых на готовность к операции записи (допускается NULL)
- exceptfds — множество файловых дескрипторов, проверяемых на «срочные» данные (обычно NULL)
- nfds — максимальный номер файлового дескриптора во всех трех множествах + 1

Системный вызов pselect

- timeout задает максимальное время ожидания, NULL — неограниченное ожидание

```
struct timespec {
    long    tv_sec;         /* seconds */
    long    tv_nsec;        /* nanoseconds */
};
```

- sigmask задает маску блокируемых сигналов на время работы pselect (по аналогии с sigsuspend)

Системный вызов pselect

- pselect возвращает
  - -1 при ошибке, например, при поступлении и обработке сигнала
  - 0 при истечении времени ожидания
  - >0 — суммарное число файловых дескрипторов, готовых к выполнению операции
- **Файловые дескрипторы связанные с дисковым вводом-выводом всегда готовы и на чтение, и на запись!**

Использование pselect

- Для каждого клиента хранится информация:
  - Номер файлового дескриптора
  - Состояние обработки данных
  - Буфер данных, ожидающих отправки клиенту
- Когда файловый дескриптор готов, он обрабатывается с помощью **неблокирующих** read/write/accept, пока не будет получена ошибка EAGAIN
- Пример: server.cpp — реализация ping-pong с помощью pselect на сервере

Событийно-ориентированные программы

- Программа, использующая pselect схематично выглядит так:

```
while (1) {
   // ждать поступления события
   // обработать поступившее событие
}
```

- Единственной точкой ожидания процесса является pselect
- Это — пример событийно-ориентированной программы

Событийно-ориентированные программы

- Событийно-ориентированные программы построены по принципу автоматов:
  - Выделяются состояния, в которых может находится автомат
  - Выделяются все типы событий
  - Описываются переходы между состояниями по приходу всех типов событий
  - Требования: действия во время выполнения переходов между состояниями не должны блокировать процесс

Недостатки pselect

- Тип fd_set допускает ограниченное множество дескрипторов (по умолчанию — 1024)
- Системный вызов pselect модифицирует параметры, перед каждым вызовом их нужно готовить заново
- Необходимо просканировать весь fd_set, чтобы найти готовые файловые дескрипторы
- Итог: **pselect плохо масштабируется!**

Улучшенные средства

- Зависят от операционной системы, на Linux — epoll

```
int epoll_create(int size);
```

- Size не используется, должен быть > 0
- Возвращается файловый дескриптор для дальнейшего использования
- Для закрытия используется close

epoll_ctl

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event
*event);
```

- EPOLL_CTL_ADD, EPOLL_CTL_DEL, EPOLL_CTL_MOD — операции
- fd — интересующий нас файловый дескриптор

```
typedef union epoll_data {
    void        *ptr;
    int          fd;
    uint32_t     u32;
    uint64_t     u64;
} epoll_data_t;
struct epoll_event {
    uint32_t     events;      /* Epoll events */
    epoll_data_t data;        /* User data variable */

};
```

epoll_pwait

```
int epoll_pwait(int epfd, struct epoll_event *events,
                int maxevents, int timeout,
                const sigset_t *sigmask);
```

- Возвращаются события (не более maxevents) в массив events
- Timeout задается в миллисекундах
- Системный вызов возвращает
  - число событий (при результате > 0)
  - 0 — признак тайм-аута
  - -1 — признак ошибки

Достоинства epoll

- Нет ограничения на число файловых дескрипторов (до 100000 работает ok)
- Не нужно каждый раз подготавливать множества файловых дескрипторов
- Результат удобнее для обработки

Файловый ввод-вывод

- Операции с регулярными файлами и каталогами могут требовать значительного времени
- Неблокирующие операции, pselect, epoll — не работают с файлами так, как с другими устройствами
  - Путь данных с диска в буферы в процессе затрагивает много подсистем ядра, в частости, управление памятью, корректная реализация крайне сложна

Файловый ввод-вывод

Варианты решения проблемы:

- Выполнять файловые операции в отдельных нитях
- Для чтения отображать файл в память mmap, рекомендовать ядру подгрузить его в память с помощью madvise, проверять наличие файла в памяти с помощью mincore

Другие использования файловых дескрипторов

- Файловый дескриптор удобно использовать в pselect/epoll для одновременного ожидания разнородных событий
- Inotify — отслеживание изменений в файловой системе
- Signalfd — ожидание сигналов с помощью ф. Д.
- Timerfd — таймеры, готовность которых определяется по файловым дескрипторам
- Evenfd — «легкий» механизм wait/notify с помощью файловых дескрипторов