Лекция 27
Событийно-ориентированные программы

Signal (2) на Linux

- Системный вызов signal реализует семантику System V
- Библиотечная функция signal реализует либо BSD, либо System V
- Если определен макрос _GNU_SOURCE или _BSD_SOURCE – BSD
- gcc -std=gnu11 prog.c – даст BSD семантику
- gcc -std=c11 prog.c – даст System V семантику

Таймер

- Отмеряют интервалы времени и уведомляют процесс о срабатывании
- Самые простые: alarm(sec), ualarm(usec) – при срабатывании SIGALRM, однократно
- setitimer/getitimer – выбор типа времени, выбор однократно/периодически
  - ITIMER_REAL – реальное время – SIGALRM
  - ITIMER_VIRTUAL – время пользовательского режима – SIGVTALRM
  - ITIMER_PROF – время пользовательского режима и ядра (профилирование) - SIGPROF

POSIX timers

- timer_create, timer_settime, timer_gettime, timer_delete
- Несколько таймеров на один процесс/нить
- При срабатывании: ничего не делать, генерировать сигнал, запускать функцию в нити

Timerfd (Linux)

- Отображает таймеры на файловые дескрипторы
- timerfd_create – создание таймера
- timerfd_settime – установка
- timerfd_gettime – получение состояния
- Read – ждать срабатывания

Другие использования файловых дескрипторов (Linux)

- Inotify — отслеживание изменений в файловой системе
- Fanotify – отслеживание изменений в файловой системе
- Evenfd — «легкий» механизм wait/notify с помощью файловых дескрипторов
- Posix Message Queue (mq) – очереди сообщений
- Memfd – анонимные временные файлы

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)

Системные вызовы

- Select (самый старый, из BSD), pselect – работают со множествами файловых дескрипторов
- Poll, ppoll (SystemV) работают с массивами файловых дескрипторов
- При увеличении числа отслеживаемых файловых дескрипторов растут накладные расходы на передачу параметров в ядро

Epoll (Linux)

-

int epoll_create1(int flags);

- Flags – 0 или FD_CLOEXEC
- Возвращается файловый дескриптор для дальнейшего использования
- Список отслеживаемых файловых дескрипторов хранится в ядре
- 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

Использование select/poll/epoll

- 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
-

Событийно-ориентированные программы

- Программа, использующая epoll схематично выглядит так:

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

- Единственной точкой ожидания процесса является epoll
- 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