Linux API с точки зрения разработчика высокопроизводительного веб-сервера

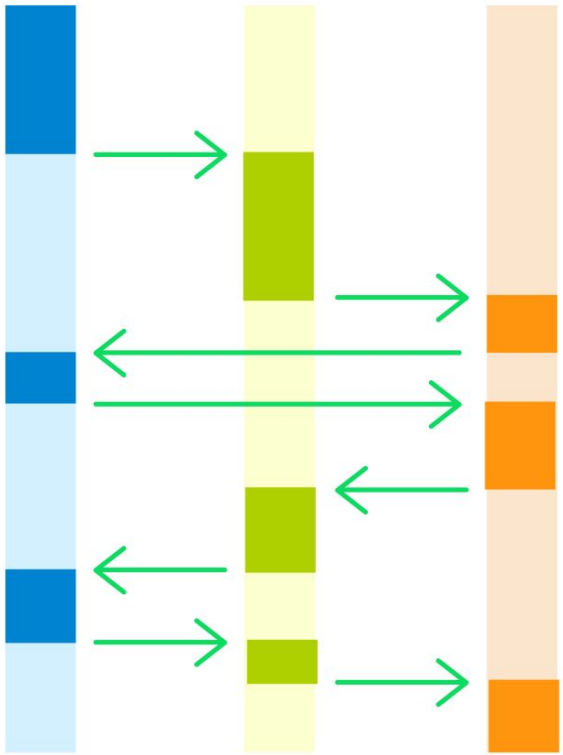
Валентин Бартенев
NGINX, Inc.

(Сверх)краткое введение

- Linux API - набор системных вызовов ядра и библиотек
- Системный вызов - обращение приложения к ядру
- Ядро
  - Взаимодействие с оборудование
  - Управление памятью
  - Управление процессами
  - Работа с файловыми системами
  - Сетевые протоколы
- POSIX - Portable Operating System Interface

Обработка соединений

Традиционный подход



Мультиплексирование I/O

Режимы работы

- Блокирующий
  - Приложение ожидает завершения операции
  - Исполнение будет продолжено только после получения/отправки данных или при ошибке
- Неблокирующий
  - Ядро сразу сообщает нам если данные для чтения отсутствуют или буфер отправки заполнен
  - Исполнение продолжается даже при невозможности выполнить операцию на данный момент

Цикл обработки соединений

`connections[]` - массив соединений

`handle_connection()` - обработчик соединения

```
loop {
    for (i = 0; i < count; i = i + 1) {
        handle_connection(connections[i]);
    }
}
```

Цикл с задержкой

```
loop {
    for (i = 0; i < count; i = i + 1) {
        handle_connection(connections[i]);
    }

    sleep(200ms)
}
```

Эффективный цикл

```
loop {
    count = wait_for_events(connections);

    for (i = 0; i < count; i = i + 1) {
        handle_connection(connections[i]);
    }
}
```

Существующие механизмы

- POSIX: select(), poll()
  - Неэффективны при большом количестве соединений
- Эффективные механизмы:
  - Linux: epoll
  - FreeBSD: kqueue
  - Solaris: /dev/poll, event ports

Базовый интерфейс epoll

Создание экземпляра epoll:

```
int epoll_create();
```

Регистрация дескрипторов:

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

Ожидание событий:

```
int epoll_wait(int epfd, struct epoll_event *events,
               int maxevents, int timeout);
```

Параметры epoll_ctl()

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

`int epfd` - экземпляр epoll
`int op` - тип операции
`int fd` - отслеживаемый дескриптор

```
struct epoll_event {
    uint32_t     events;  - битовая маска типов событий
    epoll_data_t data;    - пользовательские данные
};
```

События epoll

```
struct epoll_event {
    uint32_t     events;  - битовая маска типов событий
    epoll_data_t data;    - пользовательские данные
};
```

Основные флаги событий:

- `EPOLLIN` - чтение
- `EPOLLOUT` - запись
- `EPOLLERR` - ошибка
- `EPOLLRDHUP` - закрытие соединения (Linux 2.6.17+)

Детектирование закрытия соединения

- В Linux 2.6.17 появился новый флаг - EPOLLRDHUP
  - На ядрах старше 2.6.17 необходимо вычитывать все данные и делать на один системный вызов больше
- EPOLLRDHUP тихо игнорируется старыми ядрами
  - Невозможно отличить отсутствие события от неработоспособности флага
  - Приходится тестировать работоспособность

Избыточный тип операции в epoll_ctl()

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

Операции (`int op`):

- `EPOLL_CTL_ADD` - добавление
- `EPOLL_CTL_MOD` - изменение
- `EPOLL_CTL_DEL` - удаление

Можно ли было обойтись двумя? А одной?

```
int epoll_ctl(epfd, fd, event);
```

Статистика системных вызовов на примере nginx в режиме проксирования:

System Call	Count	Total ns	Avg ns	Min ns	Max ns
epoll_ctl	300296	611292755	2035	825	190073
close	200198	2146993913	10724	3690	1028030
recvfrom	200167	802459251	4008	1443	396482
accept	102186	266163074	2604	1363	104853
socket	100098	507851535	5073	2310	122225
ioctl	100098	131223227	1310	613	1003929
connect	100098	838910174	8380	4437	993180
setsockopt	100067	236083311	2359	835	130495
sendto	100067	595770368	5953	2560	160524
writev	100067	1084452444	10837	4316	198168
epoll_wait	4034	65422061	16217	1610	2163480
brk	1899	23225586	12230	1673	137208
write	33	294372	8920	3913	64126
Total:	1.409.308	7.310.142.071			

Хитрости epoll и close()

- После close(fd) вызов epoll_ctl(fd, EPOLL_CTL_DEL) не нужен, удаление происходит автоматически.

- На ядрах до версии Linux 3.3 и после 3.13, вызов epoll_ctl(fd, EPOLL_CTL_DEL) перед вызовом close(fd) - повышает производительность!

Интерфейс epoll против kqueue

```
epoll_ctl()
epoll_wait()
epoll_pwait()
epoll_create()
epoll_create1()
eventfd()
eventfd2()
timerfd_create()
timerfd_settime()
timerfd_gettime()
signalfd()
signalfd4()
inotify_init()
inotify_init1()
inotify_add_watch()
inotify_rm_watch()
```

~=

```
kqueue()
kevent()
```

Линус о kqueue

« I've actually read the BSD kevent stuff, and I think it's classic over-design. It's not easy to see what it's all about, and the whole <kq, ident, filter> tuple crap is just silly. Looks much too complicated. »

@ http://yarchive.net/comp/linux/event_queues.html

- man epoll и связанные: 12 517 слов
- man kqueue: 2 479 слов

Асинхронная работа с файлами

- POSIX AIO (aio_read(), aio_write() и т. д.) в Linux
  - Обертка от glibc в пользовательском пространстве
  - Плохо масштабируется, достаточно высокие накладные расходы
- Kernel AIO (io_submit(), io_setup() и т. д.)
  - Требования к выравниванию
  - Работает только в режиме O_DIRECT

Собственный пул потоков в NGINX

- «Пулы потоков: ускоряем NGINX в 9 и более раз»
  - https://habrahabr.ru/post/260669/
- Лишние накладные расходы если данные в памяти
- Предложенные решения (так и не включены в ядро):
  - pwritev2()/preadv2() с флагом RWF_NONBLOCK
  - fincore()

Отправка файлов

Стандартный способ:

```
loop {
    read(file -> buf);
    write(buf -> connection);
}
```

Эффективный способ:

```
sendfile(file -> connection);
```

Интерфейс sendfile()

Linux:

```
ssize_t sendfile(int out_fd, int in_fd,
                 off_t *offset, size_t count);
```

FreeBSD:

```
int sendfile(int fd, int s, off_t offset, size_t nbytes,
             struct sf_hdtr *hdtr, off_t *sbytes,
             int flags);
```

Solaris:

```
ssize_t sendfilev(int fildes, struct sendfilevec *vec,
                  int sfvcnt, size_t *xferred);
```

Неприятности с sendfile() в свежих ядрах

```
ret = sendfile(out_fd, in_fd, offset, count);
```

- Linux до 4.3:
  - EINTR только в самом начале
  - ret < count эквивалентно EAGAIN
- Linux после 4.3:
  - EINTR в произвольный момент
  - Не отличить EAGAIN от EINTR при ret < count
  - Требуется на один вызов больше

Linux и опция SO_REUSEPORT

- В Linux и DragonFly BSD работает иначе, чем в других unix-подобных системах и позволяет:
  - Избежать борьбы за лок на listen-сокете
  - Равномерно распределять соединения
  - Складывать пакеты от одного отправителя в один и тот же процесс
- Теряет соединения при закрытии
  - В DragonFly BSD при реализации SO_REUSEPORT консультировались с разработчиками nginx и проблему решили


КТО ТАМ ПЕРВЫЙ, А?
НЕЛЬЗЯ ЛИ РЫТЬ ТРОПУ ПРЯМО?
ХЗ, МОЖЕТ КТО-ТО КОСОНОГИЙ
ИЛИ КОСОГЛАЗЫЙ, ЛОЛ.
РЕАЛЬНО ЗАПАРИЛСЯ УЖЕ
ТУДА-СЮДА ШАРАХАТЬСЯ
ПО КАНАВЕ.
ОЙ, НЕ ПОХЕРУ ЛИ?
ИДЁМ - И НА ТОМ СПАСИБО.
Я ШЕРСТЯНОЙ ВОЛЧАРА,
БОЖЕ, КАК Я ХОРОШ,
КАК МОЩНЫ МОИ ЛАПИЩИ!

Спасибо за внимание.

Валентин Бартенев
vbart@nginx.com

Вакансия разработчика (Си, Москва):
https://www.nginx.com/jobs/cunix-developer/