ФИЗТЕХ-ШКОЛА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра информатики и вычислительной математики

Н.Н. Ефанов

МУЛЬТИПЛЕКСИРУЕМЫЙ ВВОД-ВЫВОД

Учебно-методическое пособие

МОСКВА
МФТИ
2024

УДК 681.3.06,681.3.07

Рецензент:
Доктор физико-математических наук, доцент *Н. И. Хохлов*

Ефанов, Н. Н.
Мультиплексируемый ввод-вывод: учебно-методическое пособие – М.: МФТИ, 2024. – 24 с.

Учебно-методическое пособие содержит описание основных способов организации мультиплексируемого ввода-вывода в современных Unix-подобных операционных системах, а также содержит указания по организации такого ввода-вывода в пользовательском приложении на основе механизмов poll/select/epoll.

Предназначено для студентов 2 курса, изучающих основы операционных систем в рамках дисциплины «Компьютерные технологии».

УДК 681.3.06,681.3.07

Глава 1. **Обработка событий на множестве файловых дескрипторов**

1.1. Мультиплексирование ввода-вывода

Мультиплексирование ввода-вывода – это техника, позволяющая процессу или потоку отслеживать несколько файловых дескрипторов и получать уведомления, когда системный ресурс, абстрагированный каким-либо из этих дескрипторов, готов к операциям ввода-вывода (например, к чтению или записи).

Данная техника часто используется в операционных системах для эффективного управления несколькими операциями ввода-вывода на сценариях, где один процесс или поток должен обрабатывать множество файловых дескрипторов, абстрагирующих сетевые сокеты, каналы связи, файлы или другие примитивы взаимодействия. Подобный сценарий обработки зачастую реализуется в сетевых приложениях, таких как серверы, которым необходимо обрабатывать несколько клиентских подключений одновременно, в силу того, что организовывать обработку по принципу «один процесс(поток) на соединение» при большом количестве соединений – расточительно [**?**].

1.2. Варианты оповещения о готовности дескрипторов

Прежде чем переходить к рассмотрению реализации обработки множества дескрипторов, опре деленим 2 модели оповещения о готовности дескриптора к обработке[1]:

1) Срабатывание по фронту («edge-triggered») – оповещение происходит по факту события ввода-вывода на дескрипторе

2) Срабатывание по уровню («level-triggered») – оповещение происходит по готовности дескриптора быть обработанным, к примеру, по наличию данных, готовых для чтения без блокировки

[1] Данные модели в целом характерны для систем цифровой обработки сигналов или дискретных событий

Для возможности работать с данными без блокировок нас будут в первую очередь интересовать примитивы, поддерживающие модель срабатывания по уровню. Тем не менее, некоторые из рассматриваемых в данном пособии примитивов поддерживают и срабатывание по фронту, например, epoll [**?**]. Следует также отметить, что рассматриваемые примитивы с моделью срабатывания по уровню не гарантируют отсутствие блокировок, так как не изменяют режим работы с дескриптором, а лишь указывают о возможности такой работы. Блокировка может возникнуть, к примеру, в ходе операции записи большого количества данных [**?**].

1.3. Варианты реализации обработки множества дескрипторов

Рассмотрим основные варианты обработки событий на множестве дескрипторов в рамках одного потока[2] в режиме срабатывания по уровню. В качестве примера, демонстрирующего целесообразность использования мультиплексирования для данного сценария, выберем простейший пример ожидания готовности данных для чтения на фиксированном множестве дескрипторов.

Предположим, существует однопоточное приложение, в котором производится чтение по нескольким дескрипторам, хранящимся в массиве `fds`. Заранее неизвестно, в каком порядке и какого размера данные будут доступны для чтения. Следовательно, для решения задачи вычитывания данных по мере их готовности[3], требуется дожидаться готовности данных для каждого из дескрипторов, чтобы решить задачу корректно. Рассмотрим варианты реализации такой программы, использующие блокирующий, неблокирующий и мультиплексирующий подходы.

1) Блокирующий подход. Подход применяется по умолчанию в большинстве современных операционных систем общего назначения: в случае, если данные не готовы, производится блокировка потока средствами операционной системы до

[2] В данном пособии не рассматриваются как многопоточные, так и асинхронные варианты обработки событий, основанные на вызове со-программ, POSIX AIO, либо на основе сигналов

[3] При этом опустим даже требования соблюдения строгого порядка в вычитывании на дескрипторах

тех пор, пока данные не станут доступны. Далее разблокированный поток может считать порцию готовых данных без блокировки. Следовательно, в случае, если производится проверка готовности данных на дескрипторах с перечислением дескрипторов в массиве `fds` по возрастанию индексов, неготовность данных на дескрипторе `fds[i-1]` заблокирует `read(fds[i-1],...)` до моменда их готовности, а данные на `read(fds[i],...)` все это время не будут считаны, даже если они были готовы ранее, чем на `fds[i-1]`.

2) Неблокирующий подход. В случае, если для файлового дескриптора установлен флаг `O_NONBLOCK`, что можно выполнить при открытии файла либо при настройке посредством `fcntl`, вызов `read` будет обрабатывать дескриптор в неблокирующем режиме. Это означает, что в случае, если данные не готовы, вызов моментально вернет управление, с соответствующим кодом возврата -1 и ошибкой `EAGAIN`(11), не переводя вызвавший поток в состояние ожидания готовности данных. В случае готовности данных, поведение вызова не будет отличаться от поведения при блокирующем подходе. Следовательно, для организации программы в таком неблокирующем режиме требуется завести цикл, производящий попытку прочитать данные на каждом из дескрипторов на каждой итерации. Фактически, функция примитива ожидания выполняется данным циклом, именуемым в литературе активным циклом ожидания. Ожидание называется активным в силу того, что потребляет процессорное время. Несмотря на то, что во многих случаях подобные циклы считаются анти-шаблоном проектирования ПО, в отдельных сценариях, для которых либо априори известно, что количество итераций данного цикла невелико[4], либо требуется обеспечить лучшую интерактивность по сравнению с блокирующим решением[5], такое решение преемлемо. Тем не менее, оно очевидно не является преемлемым для высоконагруженных сетевых приложений, обрабатывающих большое

[4] К примеру, в спин-блокировках в ядре ОС, спроектированных с целью обеспечения эксклюзивного доступа к готовым данным

[5] Заметим, что возврат из вызова в случае неготовности данных происходит условно быстро

количество сетевых соединений, так как за $n >> 1$ итераций цикла будет производиться n вызовов, вносящих дополнительные накладные расходы в деятельность приложения.

3) Мультиплексирующий подход. В данном подходе ожидание происходит на специальном системном вызове, производящем мониторинг некоторых событий на множестве дескрипторов. Возврат управления происходит либо если данные события произошли, либо по другим условиям, например, по истечении некоторого интервала времени. Далее определеляется подмножество дескрипторов, готовых к обработке[6] обычными вызовами чтения/записи.

1.4. Мультиплексирование ввода-вывода в Linux

Мультиплексирование ввода-вывода в Linux представлено тремя семействами механизмов[7]: `select`, `poll`, `epoll`[8]. Семейство определяет «стить», в котором осуществляется мультиплексирование и особенности работы с событиями и дескрипторами. Также существует ряд сторонних библиотек, предоставляющих унифицированный интерфейс для мультиплексирования, к примеру, `libevent`. В данном пособии они не рассматриваются.

Вызов select

Вызов `select` («выделить») стандартизирован по POSIX и является наиболее широко портированным из всех существующих средств мультиплексирования. `select` реализует модель срабатывания по уровню.

Прототип функции для работы с `select` средствами `libc` объявлен в заголовочном файле `sys/select.h` и имеет вид:

```
int select(int nfds, fd_set *readfds,
```

[6] Как будет показано далее, алгоритмическая сложность определения может лежать от $O(1)$ до $O(n)$ для n наблюдаемых дескрипторов, в зависимости от семейства вызовов

[7] Семействами, так как в каждое из семейств входит несколько системных вызовов, к примеру poll и ppoll

[8] Данный механизм часто выделяют в отдельный тип взаимодействия, так как он сочетает в себе элементы и мультиплексирования, и обработки на основе сигналов [?]

```
fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);,
```

Где:

- `nfds` – максимальный номер дескриптора во всех наблюдаемых множествах плюс 1.

- `readfds`, `writefds`, `exceptfds` – указатели на множества дескрипторов, наблюдаемых на готовность чтения, записи, либо на исключительные ситуации соответственно.

- `timeout` – указатель на структуру, содержащую значение времени до тайм-аута ожидания событий. Если тайм-аут задан как 0, вызов завершится без блокировки. В противном случае, либо если тайм-аут не задан (указатель равен NULL), произойдет блокировка вызвавшего потока до истечения тайм-аута, либо до возникновения событий, ассоциированных с дескрипторами из наблюдаемых множеств.

Работа с множествами дескрипторов осуществляется посредством соответсвующих макроопределений из `sys/select.h`, в частности:

- `FD_ZERO(&fds)` – инициализировать пустое множество дескрипторов `fds`.

- `FD_SET(n, &fds)`,`FD_CLR(n, &fds)` – добавить либо соответственно удалить дескриптор $n \leq$ `FD_SETSIZE` из `fds`, где `FD_SETSIZE` – максимально возможный номер дескриптора, определенный как 1024 в заголовочных файлах Linux [?]. Поведение при $n >$ `FD_SETSIZE` не определено. Таким образом, `select` не поддерживает одновременную работу с числом дескрипторов, превышающим `FD_SETSIZE`.

- `FD_ISSET(n, &fds)` – проверить, что произошло событие, ассоциированное с дескриптором n из `fds`.

По истечении интервала времени, заданного `timeout`, без произошедших событий, вызов вернет 0. В случае ошибки, вызов возвращает -1, а код ошибки можно получить из `errno`.

Если во время наблюдения произошли события, вызов вернет число дескрипторов, с которыми данные события ассоциированы. Далее требуется определить, какие именно это дескрипторы, осуществив проверку значения `FD_ISSET` для каждого из дескрипторов в соответствующих множествах. В случае, если оно ненулевое, с данным дескриптором можно совершить работу обыкновенными вызовами ввода-вывода без блокировки. Таким образом, сложность поиска конкретных дескрипторов, на которых произошли события, составляет $O(N)$, где $N = |readfds| + |writefds| + |exceptfds|$. В силу того, что вызов модифицирует саму структуру множеств, повторно использовать их не допускается, и нужно совершать переинициализацию.

С целью возможности изменения маски сигналов на время вызова, а также с целью достичь большей гранулярности по времени, и, как результат, большей точности[9], существует вызов `pselect` [**?**], принимающий в качестве аргумента тайм-аута структуру `struct timespec`, а также 6-й аргумент – маску сигналов типа `sigset_t`, которая выставляется потоку непосредственно перед началом наблюдения и заменяется на исходную непосредственно после.

Вызов poll

Вызов `poll` («опрос»), стандартизированный по XPG4-UNIX, был разработан позже `select` с целью ликвидации ряда недостатков последнего, также реализует срабатывание по уровню. В первую очередь, в `poll` была решена проблема модификации структур, содержащих информацию о наблюдаемых дескрипторах, что достигнуто благодаря использованию специальной структуры `struct pollfd` описания наблюдаемого дескриптора, в которой поля настроек событий и статуса наблюдений – суть разные поля. Также `poll` не имеет ограничения `FD_SETSIZE` по числу дескрипторов, и не требует определять максимальный номер дескриптора.

Прототип функции для работы с `poll` средствами `libc` объявлен в заголовочном файле `poll.h` и имеет вид:

[9] Что зависит от точности программных таймеров, и, вообще говоря, не гарантируется для систем общего назначения

`int poll(struct pollfd fds[], nfds_t nfds,int timeout)`, Где:

- `fds` – массив структур описания дескрипторов.
- `nfds_t` – количество дескрипторов в fds.
- `timeout` – тайм-аут в миллисекундах. В случае указания -1 вызов будет бесконечно ожидать первого события, в случае 0 – завершится без блокировки.

Настройка элементов `fds` производится следующим образом: в поле `fds[i].fd` присваивается номер наблюдаемого дескриптора, в поле `events` – маска наблюдаемых событий, к примеру `POLLIN | POLLOUT`. После возврата `poll` статус события можно проверить, проверив условие взведения соответсвующих бит в `revents` для каждого из дескрипторов, к примеру:

```
if (fds[i].revents & POLLIN) {...
    // можно читать без блокировки }
```

Таким образом, временная сложность определения конкретных дескрипторов составляет $O(N)$, как и в случае с `select`.

По истечении тайм-аута, `poll` возвращает 0. В случае готовности некоторых из наблюдаемых дескрипторов – число готовых к обработке без блокировки, в случае ошибки – -1, с кодом ошибки в `errno`.

Задание `timeout` в миллисекундах вносит ограничение в гранулярность интервалов времени, за которые `poll` наблюдает за дескрипторами. С целью возможности изменения маски сигналов на время вызова, аналогично вызову `pselect`, а также с целью достичь большей гранулярности по времени, и, как результат, большей точности[7], существует вызов `ppoll` [?], принимающий в качестве аргумента тайм-аута структуру `struct timespec`, а также 4-й аргумент – маску сигналов типа `sigset_t`.

Механизм epoll

Linux-специфичный механизм `epoll` («event-poll»–«опрос событий»), добавленный в ядро начиная с версии 2.5.45, обеспечивает лучшую масштабируемость на большое число дескрипторов, нежели `select` и `poll`. Помимо данного ключевого свойства,

`epoll` позволяет получить перечень дескрипторов, на которых произошли события, за $O(1)$.

Интерфейс работы с `epoll` относительно громоздок, и включает в себя несколько вызовов и структур, объявленных в `sys/epoll.h`. Ключевым элементом этого интерфейса является дескриптор `epoll`, используемый для двух целей – сохранение и настройка списка наблюдаемых дескрипторов – «списка интереса», и получение списка дескрипторов, готовых к вводу-выводу – «списка готовности». По выполнении управляющего вызова – `epoll_ctl`, в который передается информация о `epoll`-дескрипторе и о наблюдаемых дескрипторах, ядро осуществляет наблюдение за открытыми файлами, с которыми дескрипторы ассоциированы, и добавляет записи для готовых дескрипторов в «список готовности» в через механизм обратных вызовов. Записи данного списка будут прочитаны вызовом `epoll_wait`.

Создание экземпляра `epoll`-примитива происходит одним из вызовов:

1) `int epoll_create(int size)` – создать экземпляр с ограничением в `size` наблюдаемых дескрипторов. Вызов устарел, на современных версиях ядра `size` игнорируется.

2) `int epoll_create1(int flags)` – создать экземпляр, настраеваемый флагами во `flags` [?].

В случае, если в В случае ошибки, вызовы возвращают -1. В случае успеха – дескриптор, ассоциированный с экземпляром `epoll`.

Добавление, модификация и удаление наблюдаемых дескрипторов выполняется вызовом `epoll_ctl`:

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

Где:

- `epfd` – дескриптор `epoll`
- `op` – операция, задаваемая масками `EPOLL_CTL_ADD`, `EPOLL_CTL_MOD`, `EPOLL_CTL_DEL`, добавления, модификации или удаления дескриптора соответсвенно
- `fd` – наблюдаемый дескриптор

- `event` – указатель на структуру, поле `events` которой является битовой маской настройки наблюдаемых событий, например, `EPOLLIN`. Если в данной маске не выставлен `EPOLLET`, данный дескриптор будет обрабатываться в режиме срабатывания по уровню, иначе – по фронту. Поле `data` – специфицирует данные, которые ядро должно возвращать в вызов опроса –`epoll_wait` – при возникновении событий на дескрипторе.

Вызов опроса – `epoll_wait`:

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

Где:

- `epfd` – дескриптор `epoll`
- `events` – указатель на на начало области памяти, в которой структуры структуры описания детектированных событий. Очевидно, область должна быть доступна на запись, в противном случае возникнет ошибка `EFAULT`.
- `maxevents` – верхнее ограничение количества детектированных событий.
- `timeout` – тайм-аут. Значение -1 означает бесконечное ожидание, значение 0 – возврат без блокировки, иные значения – значения тайм-аута в миллисекундах.

Вызов в случае успеха возвращает количество дескрипторов, готовых к обработке, и -1 в случае ошибки.

Оценки производительности

Особенностью вызовов `select` и `poll` является передача в каждый вызов множеств отслеживаемых дескрипторов, которые в свою очередь передаются в ядро, а по возврату из вызова ядро передает обратно структуру, описывающую факт готовности каждого из дескрипторов. При таком подходе ядро должно проверить все дескрипторы, указанные в вызове, поэтому следует ожидать линейную зависимость времени отклика `select` и `poll` от количества отслеживаемых дескрипторов. В случае

Кол-во операций	poll	select	epoll
10	0.61	0.73	0.41
100	2.9	3.0	0.42
1000	35	35	0.53
10000	990	930	0.66

Таблица 1.1. Временные расходы для select, poll, epoll в секундах для 100000 операций мониторинга различного числа дескрипторов [?]

`epoll`, дескрипторы передаются в `epoll_ctl`, и на момент вызова `epoll_wait` ядру они уже известны, а запись фактов о событиях ввода-вывода на дескрипторах организуется добавлением в список, ассоциированный с `epoll`-дескриптором, элементов для готовых дескрипторов. В дальнейшем вызов `epoll_wait` будет извлекать данные из этого списка, не передавая какой-либо информации ядру. Следовательно, ожидаемая зависимость времени отклика `epoll_wait` от числа отслеживаемых дескрипторов – константная.

Эксперименты с различным количеством наблюдаемых дескрипторов для 100000 операций мониторинга, в ходе выполнения каждой из которой происходит ровно одно событие на случайном дескрипторе, для `select`[10], `poll`, `epoll_wait`, проведенные М. Керриском для ядра версии 2.6 и представленные в книге "Linux API: исчерпывающее руководство" [?] (см. Табл. 1), подтверждают гипотезы о характерных временах работы вызовов.

Сравнение

Как следует из оценок производительности, `epoll` демонстрирует лучшую масштабируемость по количеству наблюдаемых дескрипторов. `select` и `poll`, соответственно, имеют в данном случае недостатки в эффективности, однако в силу стандартизованности, `select` портирован на все POSIX-совместимые, `poll` – на большинство современных UNIX-подобных ОС соответственно, что говорит о преимуществе их использования для кросс-платформенных приложений. Другие аспекты также игра-

[10]Ограничение на максимальный номер дескриптора в данном эксперименте было увеличено

Свойство	poll	select	epoll
Портируемость	UNIX (+/-)	POSIX (+)	Linux (-)
Сложность поиска дескрипторов	$O(N)$	$O(N)$	$O(1)$
Масштабируемость	$O(N)$	$O(N)$	$O(1)$
Отсутствие переинициализации	+	-	+
Предрассчет верхнего № д-ра	+	-	+

Таблица 1.2. Сравнительная таблица для poll, select и epoll

ют роль при выборе конкретного примитива мультиплексирования для конкретного приложения. Приведем в Таб. 2 сравнение ключевых особенностей рассмотренных примитивов.

Глава 2. **Практическая реализация мультиплексирования ввода-вывода**

Приведем простейшие примеры использования `select`, `poll`, `epoll` для мониторинга готовности данных на дескрипторе:

2.1. Примеры программ

Представим модель программы циклом, на каждой итерации которого происходит ожидание готовности чтения в течение тайм-аута, приведем примеры использования `select`, `poll`, `epoll` для наблюдения за стандартным потоком ввода.

Select

Пример 1: select

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#include <sys/time.h>
#include <unistd.h>
#define BUFSIZE 1024

int main() {
    fd_set read_fds;
    struct timeval timeout;
    while(1) {
        // Инициализируем множество на чтение, добавим
дескриптор 0
        FD_ZERO(&read_fds);
        FD_SET(0, &read_fds);
        // Выставим тайм-аут
        timeout.tv_sec = 5;
        timeout.tv_usec = 0;
        int retval = select(1, &read_fds, NULL, NULL, &timeout);
        if (retval == -1) {
            exit(EXIT_FAILURE);
```

```
        } else if (retval == 0) {
            printf("Timeout expired.\n");
        } else {
            // Проверим готовность дескриптора
            if (FD_ISSET(0, &read_fds)) {
                int sz;
                if ((int sz = read(0, buf, BIFSIZE))>0) {
                    buf[sz] = '\0';        //        Нуль-
терминирование вывода
                    printf("%s\n", buf);
                }
            }
        }
    }
    return 0;
}
```

Poll

Пример 2: poll

```
#include <poll.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define TIMEOUT 5*1000
#define BUFSIZE 1024

int main() {
    struct pollfd fds;
    char buf[BUFSIZE];
    fds.fd = 0;
    fds.events = POLLIN;

    int ret = poll(fds, 2, TIMEOUT);

    if (!ret) {
        printf("TIMEOUT!\n");
    } else if (ret > 0) {
```

```
        // Проверим готовность дескриптора
        if (fds.revents & POLLIN) {
            int sz;
            if ((sz = read(fds.fd, buf, BIFSIZE))>0) {
                buf[sz] = '\0';  // Нуль-терминирование
вывода
                printf("%s\n", buf);
            }
        }
    } else {
 perror("poll");
   }
   return 0;
}
```

Epoll

Пример 3: epoll

```
#include <sys/epoll.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#define BUFSIZE 4096

int main() {
    int epoll_fd;
    char buf[BUFSIZE];
    struct epoll_event event;
    struct epoll_event events[5];
    epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        perror("epoll open");
    }
    event.events = EPOLLIN;
    event.data.fd = 0;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, 0, &event)) {
        perror("epoll ctl");
```

```
    }

    int event_count = epoll_wait(epoll_fd, events, 5, 5000);
    if (event_count == -1) {
        perror("epoll wait");
    } else {
        printf("Events count %d\n" , event_count);
        for (int i=0; i < event_count; i++ ) {
            printf("Handling FD %u\n", events[i].data.fd);
            int sz = 0;
            if ((sz = read(events[i].data.fd, buf,
                sizeof(buf))) > 0) {
                buf[sz] = '\0';   // Нуль-терминирование
вывода
                printf("%s\n", buf);
            }
        }
    }
    return 0;
}
```

2.2. Задачи для самостоятельного решения

Оглавление

Учебное издание

Ефанов Николай Николаевич

МУЛЬТИПЛЕКСИРУЕМЫЙ ВВОД-ВЫВОД

Учебно-методическое пособие

Редактор *И. А. Волкова*. Корректор *Н. Е. Кобзева*
Компьютерная верстка *Н. Е. Кобзева*

Подписано в печать 05.08.2024. Формат 60×84 $^{1}/_{16}$.
Усл. печ. л. 5,9. Уч.-изд. л. 5,3. Тираж 50 экз. Заказ № 000.

Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт
(национальный исследовательский университет)»
141700, Московская обл., г. Долгопрудный, Институтский пер., 9
Тел. (495) 408-58-22, e-mail: rio@mail.mipt.ru

Отдел оперативной полиграфии «Физтех-полиграф»
141700, Московская обл., г. Долгопрудный, Институтский пер., 9
Тел. (495) 408-84-30, e-mail: polygraph@mail.mipt.ru