Архитектура ЭВМ. Операционные системы. Асинхронный I/O

Власов Е.Е.

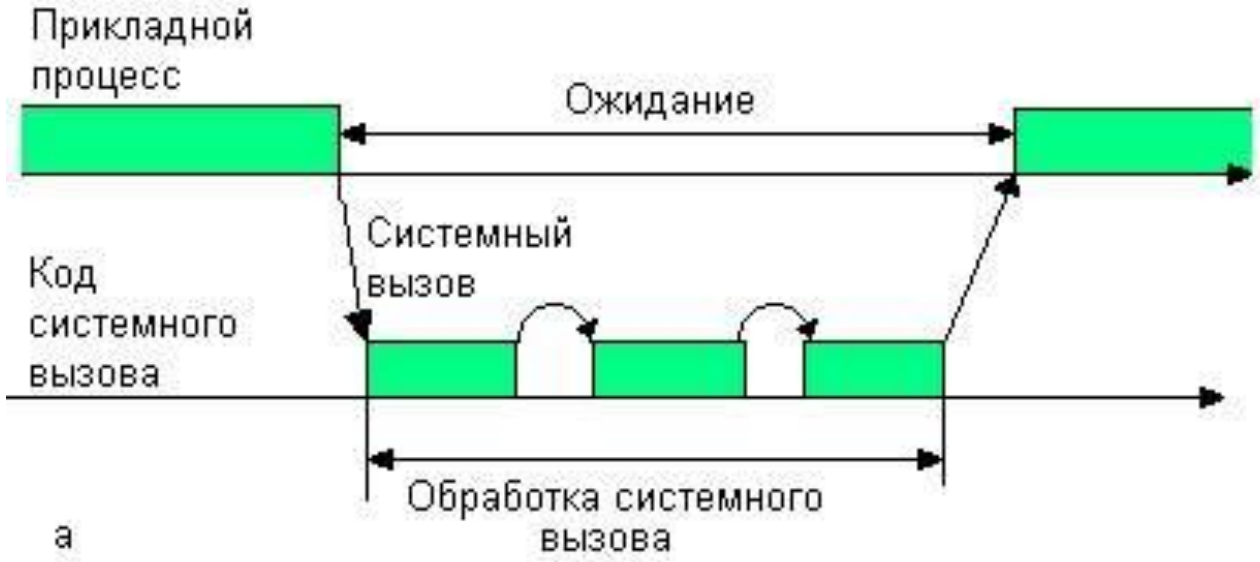
Асинхронный режим работы

Асинхронный ввод/вывод, или неблокирующий ввод/вывод является формой обработки ввода/вывода, который позволяет другим процессам продолжить выполнение до того, как передача будет завершена.

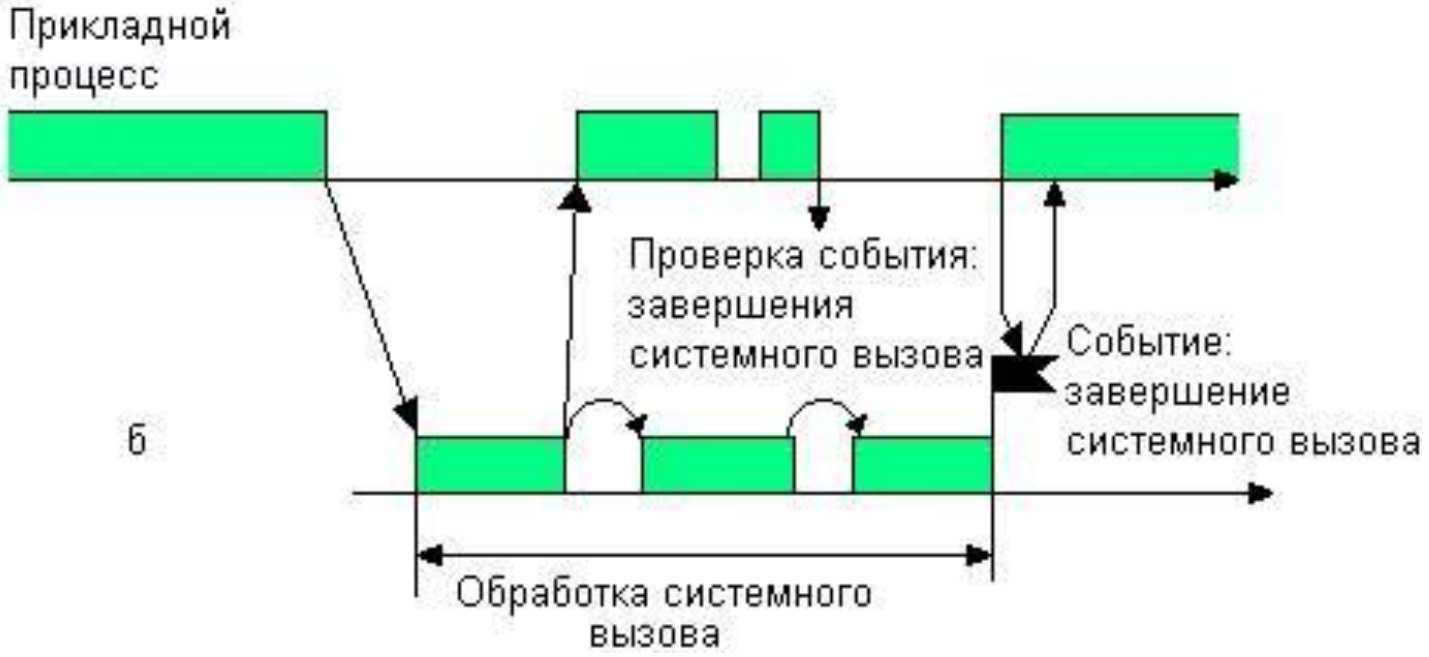
Операции ввода/вывода (I/O) в большинстве случаев значительно медленнее, по сравнению с обработкой данных в памяти. Устройство ввода/вывода может включать в себя механические части, которые должны физически двигаться (например, жесткий диск, который ищет дорожку для чтения или записи), это часто на несколько порядков медленнее, чем переключения электрического тока. Например, во время дисковой операции, которой требуется десять миллисекунд для выполнения, процессор, который работает на частоте один гигагерц, может выполнить десять миллионов циклов команд обработки.

Простым подходом к вводу/выводу было бы запустить процесс доступа, а затем ждать его завершения. Но такой подход (так называемого синхронного ввода/вывода или блокирующего ввода/вывода) будет блокировать выполнение программы, в то время, как коммуникация в процессе выполнения, оставив системные ресурсы простаивать на холостом ходу. Когда программа делает много операций ввода/вывода, это означает, что процессор может проводить почти все своё время простаивая в ожидании завершения операций ввода/вывода.

В качестве альтернативы, можно инициировать операцию ввода/вывода, а затем выполнять обработку, которая не требует завершения операции ввода/вывода. Такой подход называется асинхронный ввод/вывод. Любая задача, которая зависит от завершения ввода/вывода, (в том числе как использование значений ввода, так и критических операций, которые подтверждают, что операция вывода была завершена) по-прежнему вынуждена ожидать завершения операции ввода/вывода, и, таким образом, прежнему быть заблокированной, но другая обработка, которая не имеет зависимости от операции ввода/вывода может быть продолжена.


Прикладной процесс
Ожидание
Системный вызов
Код системного вызова
Обработка системного вызова

а


Прикладной процесс
Проверка события: завершения системного вызова
Событие: завершение системного вызова
Обработка системного вызова

б

Виды асинхронного ввода/вывода

- Процесс
- Опрашивание
- Select(/poll) цикл
- Callback
- Очереди завершения/порты
- Флаги событий
- Каналы ввода/вывода
- Зарегистрированный ввод/вывод

Процесс

Для выполнения длительной операции ввода/вывода создается новый процесс. После завершения операции процесс уведомляет процесс родитель (например, с помощью сигнала) об этом.

Преимущества:

- Простота реализации

Недостатки

- Создание нового процесса является «тяжелой» операцией
- Необходим обмен данными с дочерним процессом

Опрашивание

Варианты:

- Ошибка, если ещё не завершено (повторное обращение позднее)

- Оповещение, когда завершено, и может быть выполнено без блокировки (тогда, соответственно, пользуемся)

Доступно в традиционном Unix и Windows. Главная проблема опрашивания в том, что приходится тратить процессорное время на опрос неоднократно, во время когда нет других заданий, при этом уменьшается время, доступное для других процессов. Кроме того, поскольку опрос является по существу однопоточным, это не позволяет в полной мере использовать параллелизм ввода/вывода, на который способны аппаратные средства.

Перевод файлового дескриптора в неблокирующий режим

```
int fd_set_blocking(int fd, int blocking) {
    /* Сохраняем флаги*/
   int flags = fcntl(fd, F_GETFL, 0);
   if (flags == -1)
      return 0;
   if (blocking)
      flags &= ~O_NONBLOCK;
   else flags |= O_NONBLOCK;
   return fcntl(fd, F_SETFL, flags) != -1;
}
```

Пример чтения с опросом

```
int count=0, full_count=1024; char buffer[1024];
while (full_count> 0) {
   count = read(fd, buffer, full_count);

   if(count < 0 && errno == EAGAIN) {
          // Нет данных для чтения}
   else if(count >= 0) {
      //прочитали данные
      full_count -= count;
   } else {
      // Ошибка чтения.
   }
}
```

select/pselect

```
int select(int n, fd_set *readfds, fd_set *writefds,fd_set *exceptfds,
struct timeval *utimeout);
int pselect(int n, fd_set *readfds, fd_set *writefds,fd_set *exceptfds,
const struct timespec *ntimeout, sigset_t *sigmask);
```

Функция `select` (или pselect) является основной функцией большинства программ на языке C, эффективно обрабатывающих одновременно более одного файловового дескриптора (илисокета). Ее аргументами являются три массива файловых дескрипторов: readfds, writefds и exceptfds. Как правило, при использовании `select` программа ожидает "изменения состояния" одного или более файловых дескрипторов. Под "изменением состояния" понимается появление новых символов в потоке, с которым связан файловый дескриптор, или появление во внутренних буферах ядра места для записи в поток, или возникновение ошибки, связанной с файловым дескриптором (в случае сокета или канала это происходит, когда другая сторона закрывает соединение).

Таким образом, `select` просто следит за несколькими файловыми дескрипторами и является стандартным вызовом Unix для этих целей.

Аргументы `select()`

`readfds`	Этот набор служит для слежения за операциями чтения. После возврата из select readfds очищается от всех дескрипторов файлов, за исключением тех, для которых возможно немедленное чтение функциями recv() (для сокетов) или read() (для каналов, файлов и сокетов).
`writefds`	Этот набор служит для слежения за появлением места для записи данных в любой из файловых дескрипторов набора. После возврата из select writefds очищатся от всех файловых дескрипторов, за исключением тех, для которых возможна немедленная запись функциями send() (для сокетов) или write() (для каналов, файлов и сокетов).
`exceptfds`	Этот набор служит для слежения за исключениями или ошибками, связанными с любым из файловых дескрипторов набора. На самом деле слежение производится за появлением внепоточных (Out of Bounds - OOB) данных. Внепоточные данные посылаются через сокет с помощью флага MSG_OOB и, в действительности, exceptfds работает только для сокетов. После возврата из `select`() exceptfds очищается от всех файловых дескрипторов, кроме тех, для которых доступны внепоточные данные. Прочитать можно лишь один байт внепоточных данных (это делается с помощью recv()). Записать внепоточные данные можно в любой момент. Эта операция является неблокируемой. Поэтому нет необходимости в четвертом наборе, который мог бы служить для слежения за возможностью записи внепоточных данных в сокет.

Аргументы `select()`

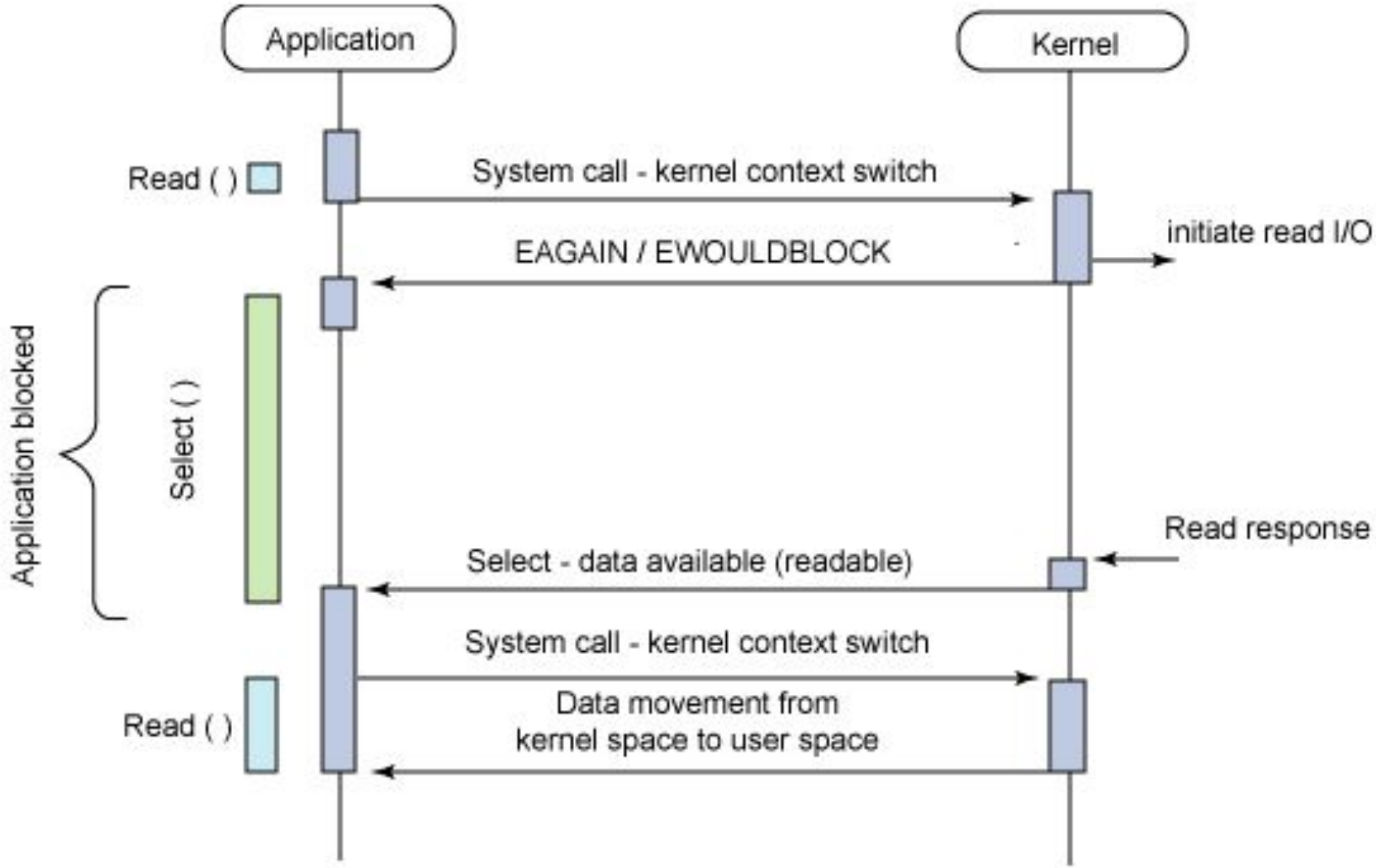
`n`	Это целое число содержит значение, на единицу большеее максимального файлового дескриптора любого из наборов. Другими словами, при добавлении файловых дескрипторов в наборы необходимо подсчитывать максимальное целое значение любого из них, затем увеличить это значение на единицу и передать как аргумент n функции select.
`select - utimeout` `pselect - ntimeout`	Этот аргумент задает наибольшее время, которое функция select будет ожидать изменения состояния дескрипторов. Если за это время ничего не произойдет, то функция возвратит управление вызвавшей программе. Если значение этого аргумента равно NULL, то select будет ожидать бесконечно. utimeout может быть установлен в ноль секунд; в этом случае select возвратит управление немедленно.utimout задает время ожидания с точностью до мксек, а ntimeout с точностью до наносекунд
`pselect- sigmask`	Этот аргумент содержит набор сигналов, которые разрешены во время вызова pselect. В качестве аргумента может быть передан NULL; в этом случае при входе в функцию и выходе из нее набор разрешенных сигналов не меняется. В этом случае функция ведет себя как select.

```
fd_set fd_in, fd_out;
struct timeval tv;


// обнуляем наборы
FD_ZERO( &fd_in );
FD_ZERO( &fd_out );
// добалыем дескриптор в для записи
FD_SET( sock1, &fd_in );
// добавляем дексриптор для чтения
FD_SET( sock2, &fd_out );
// считаем какой и дескрипторо больше
int largest_sock = sock1 > sock2 ? sock1 :
sock2;
// таймаут в 10 секунд
tv.tv_sec = 10;
tv.tv_usec = 0;
```

```
int ret = select( largest_sock + 1, &fd_in,
&fd_out, NULL, &tv );
// проверям что вернул select
if ( ret == -1 )
    // ошибка
else if ( ret == 0 )
    // timeout; за это время ничего не было
else
{
    if ( FD_ISSET( sock1, &fd_in ) )
        // событие записи

    if ( FD_ISSET( sock2, &fd_out ) )
        // событие чтения
}
```


Application
Kernel
Read ( )
System call - kernel context switch
initiate read I/O
EAGAIN / EWOULDBLOCK
Application blocked
Select ( )
Read response
Select - data available (readable)
System call - kernel context switch
Read ( )
Data movement from
kernel space to user space

Особенности использования `select`

1. Всегда старайтесь использовать `select` без указания времени ожидания. Ваша программа не должна ничего делать, если нет данных. Код, зависимый от времени ожидания, обычно плохо переносим и сложен для отладки.
2. Для повышения эффективности значение `n` должно быть правильно вычислено как указано выше.
3. Файловые дескрипторы не должны добавляться в наборы, если вы не планируете после вызова `select` проверить результат и соответствующим образом отреагировать.
4. После возврата из `select` должны быть проверены все файловые дескрипторы во всех наборах. В каждый дескриптор, готовый к записи, должны быть записаны данные, и из каждого дескриптора, готового к чтению, данные должны быть прочитаны, и т.д.

Особенности использования `select`

5. Функции read(), recv(), write() и send() необязательно считывают/записывают данные в полном объеме. Такое, конечно, возможно при низком траффике или быстром потоке, однако происходит далеко не всегда. Вы должны рассчитывать, что ваши функции получают/отправляют только один байт зараз.
6. Никогда не считывайте/записывайте побайтно, если только вы не абсолютно уверены в том, что нужно обработать небольшой объем данных. Крайне не эффективно считывать/записывать меньшее количество байт, чем вы можете поместить в буфер за один раз..

Особенности использования `select`

7. Функции read(), recv(), write() и send(), также как и `select`() могут возвратить -1 с errno установленным в EINTR или EAGAIN (EWOULDBLOCK), что не является ошибкой. Такие ситуации должны быть правильно обработаны (в вышеприведенной программе этого не сделано). Если ваша программа не собирается принимать сигналы, то маловероятно, что вы получите EINTR. Если ваша программа не использует неблокирующий ввод-вывод, то вы не получите EAGAIN. В любом случае, вы должны обрабатывать эти ошибки для полноты.

Особенности использования `select`

8. Кроме случаев, описанных в 7., функции read(), recv(), write() и send() никогда не возвращают значение меньшее единицы, если не произошла ошибка. Например, read() при работе с каналом, на котором противоположная сторона завершила работу, возвращает ноль, но возвращает ноль только один раз. Если хотя бы одна из этих функций вернула 0или -1, то вы не должны больше использовать этот дескриптор. В примере выше я немедленно закрываю дескриптор и устанавливаю его в -1 для предотвращения его включения в набор.

Особенности использования `select`

9. Значение времени ожидания должно быть инициализировано при каждом новом вызове `select`, так как некоторые операционные системы изменяют структуру.
10. Есть мнение, что сокетный уровень в Windows не обрабатывает правильно вне поточные данные. Кроме того, он неправильно работает с `select` при отсутствии файловых дескрипторов. Отсутствие файловых дескрипторов - это полезный способ перевести процесс в режим ожидания на период меньше секунды.