copyconstruct.medium.com

Nonblocking I/O - Cindy Sridharan - Medium

Cindy Sridharan

25-31 minutes

What really are descriptors?

The fundamental building block of all I/O in Unix is a sequence of *bytes*. Most programs work with an even simpler abstraction — a *stream of bytes* or an *I/O stream.*

A process references I/O streams with the help of ***descriptors***, also known as ***file descriptors***. Pipes, files, FIFOs, POSIX IPC's (message queues, semaphores, shared memory), event queues are all examples of *I/O streams* referenced by a ***descriptor***.

Creation and Release of Descriptors

Descriptors are either created explicitly by system calls like ***open***, ***pipe***, ***socket*** and so forth, or are inherited from the parent process.

Descriptors are released when:

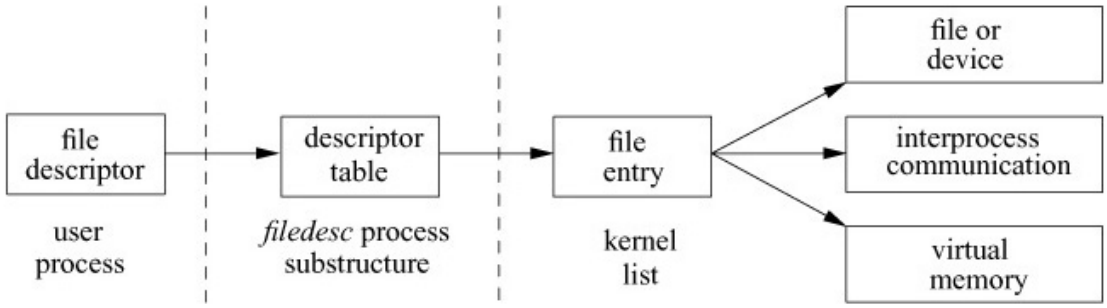
— the process exits
— by calling the ***close*** system call
— implicitly after an ***exec*** when the descriptor is marked as ***close***

on exec.

Close-on-exec

When a process forks, all the descriptors are “duplicated” in the child process. If any of the descriptors are marked **close on exec**, then after the parent **forks** but before the child **execs**, the descriptors in the child marked as “close-on-exec” are closed and will no longer be available to the child process.

Data transfer happens via a *read* or a *write* system call on a descriptor.



Chapter 7. I/O System Overview, from the book Design and Implementation of the FreeBSD Operating System. Page 315

File Entry

Every descriptor points to a data structure called the ***file entry*** in the kernel. The *file entry* maintains a per descriptor *file offset* in bytes from the beginning of the file entry object. An ***open*** system call creates a new **file entry.**

Fork/Dup and File Entries

A *fork* system call results in descriptors being ***shared*** by the parent and child with ***share by reference*** semantics. Both the

parent and the child ***are using the same descriptor*** and reference the ***same offset*** in the file entry. The same semantics apply to a *dup/dup2* system call used to duplicate a file descriptor.

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>int main(char *argv[]) {
   int fd = open("abc.txt", O_WRONLY | O_CREAT | O_TRUNC,
0666);
   fork();
   write(fd, "xyz", 3);
   printf("%ld\n", lseek(fd, 0, SEEK_CUR));
   close(fd);
   return 0;
}
```

which prints:

```
3
6
```

More interesting is what the ***close-on-exec*** flag does, if the descriptors are only being *shared*. My guess is setting the flag removes the descriptor from the child's descriptor table, so that the parent can still continue using the descriptor but the child wouldn't be able to use it once it has **exec**-*ed.*

Offset-per-descriptor

As multiple descriptors can reference the same **file entry**, the **file entry** data structure maintains a *file offset* for ***every descriptor***. Read and write operations begin at this offset and the offset itself

updated after every data transfer. The offset determines the position in the file entry where the next read or write will happen. When a process terminates, the kernel reclaims all descriptors in use by the process. If the process in question was the last to reference the **file entry**, the kernel then deallocates that **file entry**.

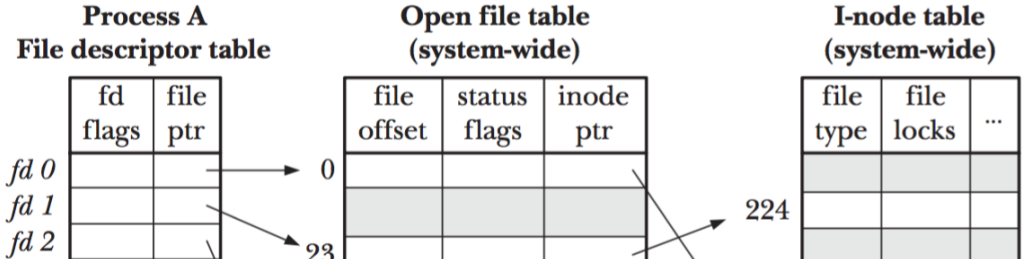
Anatomy of a File Entry

Each ***file entry*** contains:

— the *type*
— an array of function pointers. This array of function pointers translates generic operations on file descriptors to file-type specific implementations.

Disambiguating this a bit further, all file descriptors expose a common generic API that indicates operations (such as *read*, *write*, changing the descriptor mode, truncating the descriptor, *ioctl* operations, polling and so forth) that may be performed on the descriptor.

The actual implementation of these operations *vary* by file type and different file types have their own custom implementation. Reads on sockets aren't quite the same as reads on pipes, even if the higher level API exposed is the same. The *open* call is not a part of this list, since the implementation greatly varies for different file types. However once the file entry is created with an *open* call, the rest of the operations may be called using a generic API.



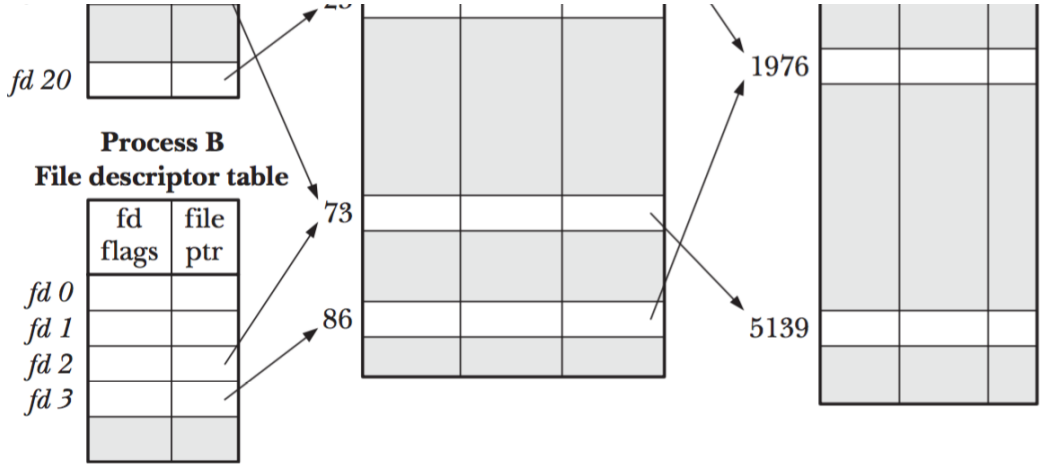


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

From the book the Linux Programming Interface — page 95

Most networking is done using *sockets*. A *socket* is referenced by a descriptor and acts as an endpoint for communication. Two processes can create two sockets each and establish a reliable byte stream by connecting those two end points. Once the connection has been established, the descriptors can be read from or written to using the **file offsets** described above. The kernel can redirect the output of one process to the input of another on another machine. The same *read* and *write* system calls are used for byte-stream type connections, but different system calls handle addressed messages like network datagrams.

Non-Blocking descriptors

By default, ***read*** on any descriptor ***blocks*** if there's no data available. The same applies to ***write*** or ***send***. This applies to operations on most descriptors except disk files, since writes to disk never happen directly but via the kernel buffer cache as a proxy. The only time when writes to disk happen synchronously is when the ***O_SYNC*** flag was specified when opening the disk file.

Any descriptor (pipes, FIFOs, sockets, terminals, pseudo-terminals, and some other types of devices) can be put in the ***nonblocking mode.*** When a descriptor is set in nonblocking mode, an I/O system call on that descriptor will **return immediately,** even if that request can't be immediately completed (and will therefore result in the process being blocked otherwise). The return value can be either of the following:

— **an error**: when the operation cannot be completed at all
— **a partial count**: when the input or output operation can be partially completed
— **the entire result**: when the I/O operation could be fully completed

A descriptor is put in the nonblocking mode by setting the *no-delay* flag ***O_NONBLOCK.*** This is also called an "open-file" status flag (in glibc, "open-file" flags are flags that dictate the behavior of the *open* system call. Generally, these options don't apply after the file is open, but ***O_NONBLOCK*** is an exception, since it is also an I/O operating *mode*).

Readiness of Descriptors

A descriptor is considered *ready* if a process can perform an I/O operation on the descriptor without blocking. For a descriptor to be considered "ready", it doesn't matter if ***the operation would actually transfer any data —*** all that matters is that the I/O operation can be performed without blocking.

A descriptor changes into a *ready* state when an I/O *event* happens, such as the arrival of new input or the completion of a socket connection or when space is available on a previously full

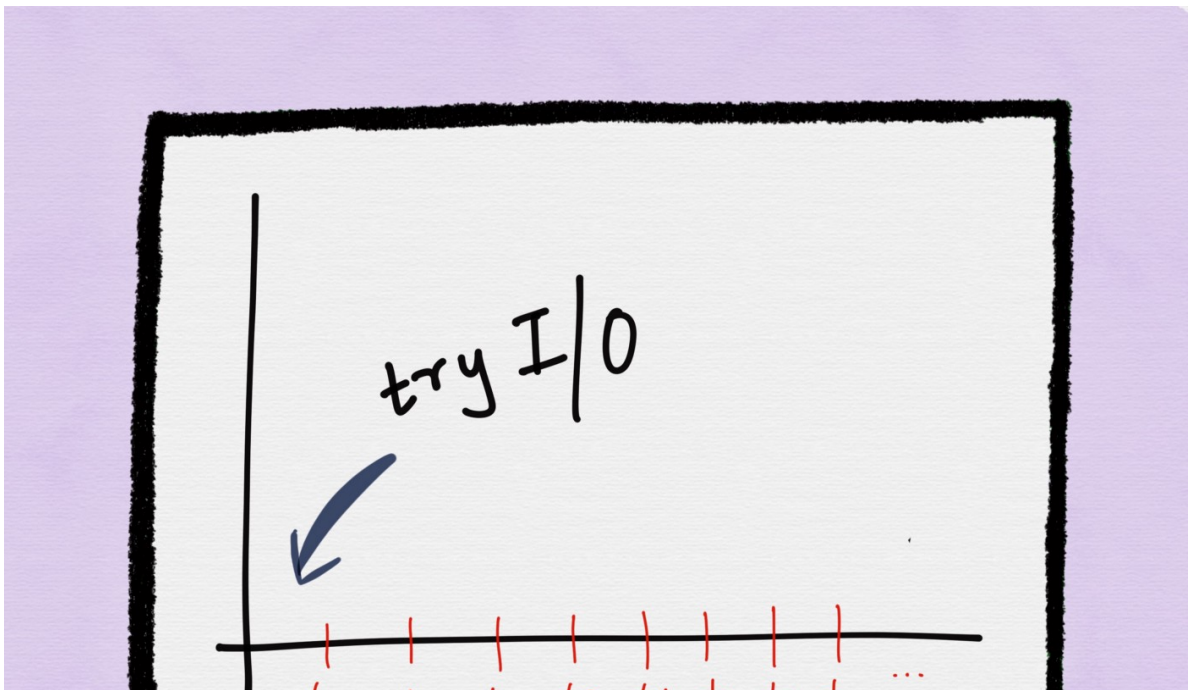
socket send buffer after TCP transmits queued data to the socket peer.

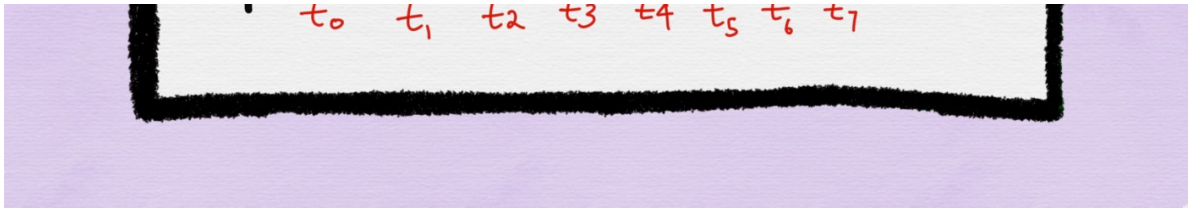
There are two ways to find out about the readiness status of a descriptor — edge triggered and level-triggered.

Level Triggered

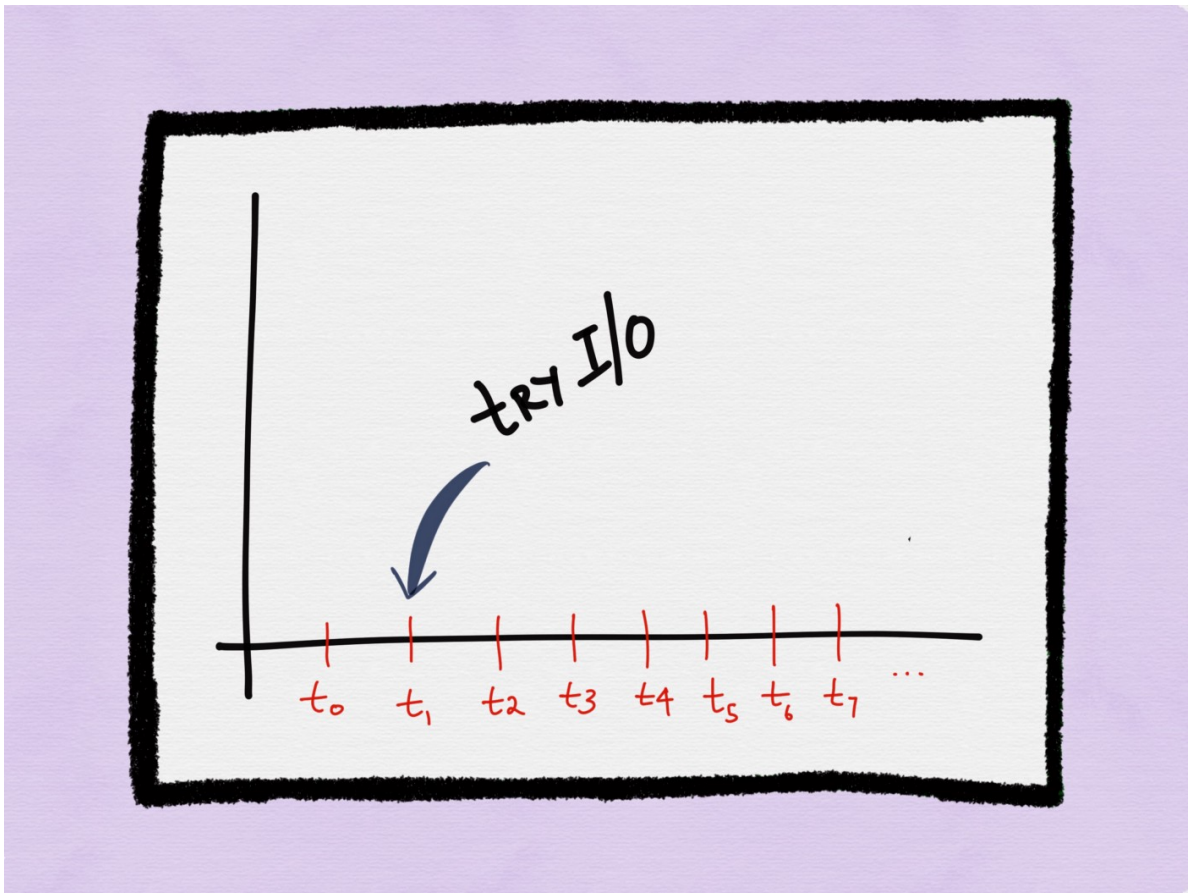
I see this as the “pull” model or the “poll” model. To determine if a descriptor is ready, the process tries to perform a non blocking I/O operation. A process may perform such operations any number of times. This allows for more flexibility with respect to the handling of any subsequent I/O operation —like for instance, if a descriptor is *ready*, a process could choose to either read all the data available or not perform any I/O at all or choose not to read all of the input data available in the buffer. Let’s see how this works with an example.

At time ***t0***, a process could try an I/O operation on a non-blocking descriptor. If the I/O operation *blocks,* the system call returns an error.

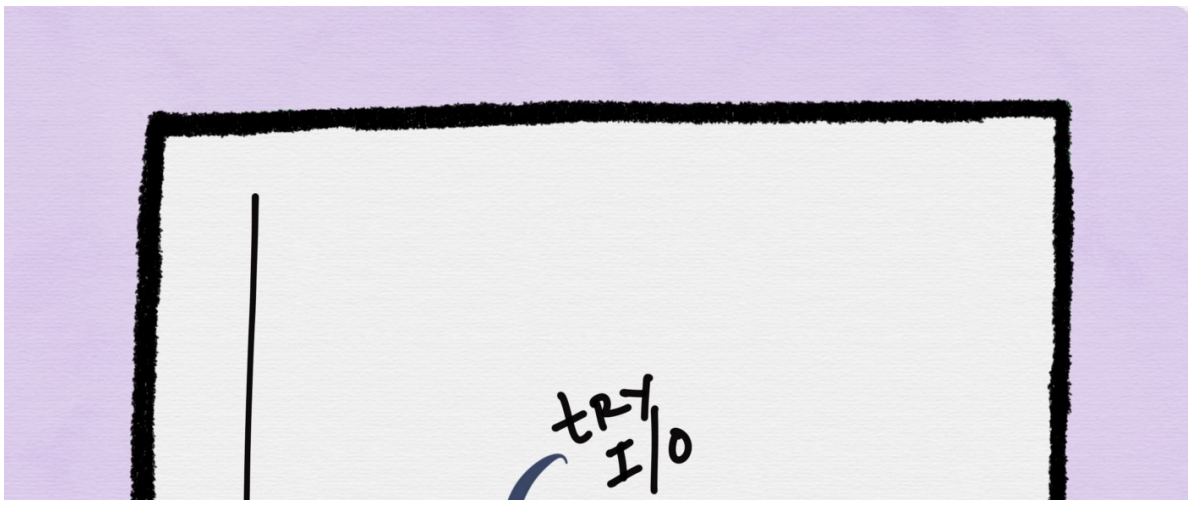


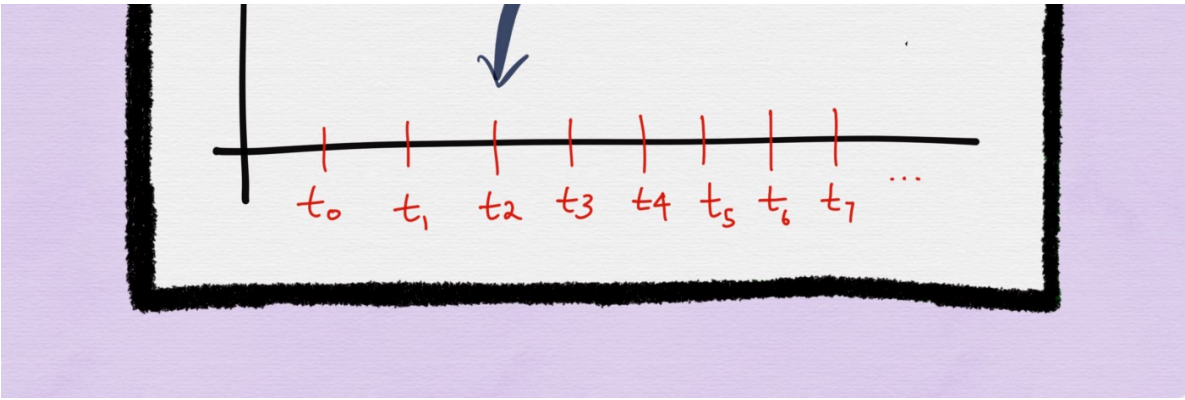


Then at time ***t1***, the process could try I/O on the descriptor again. Let's say the call blocks again and an error is returned.

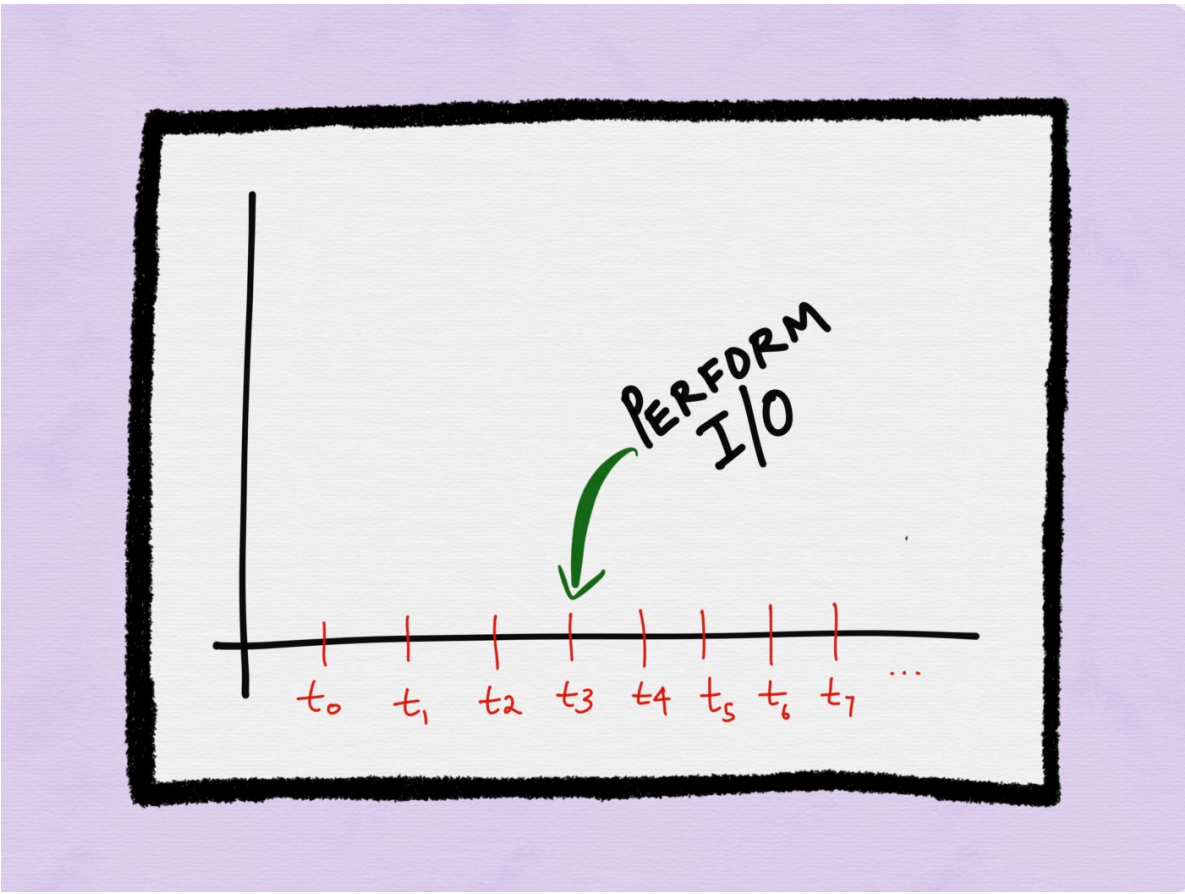


Then at time ***t2***, the process tries I/O on the descriptor again. Let's assume the call blocks yet again and an error is returned.

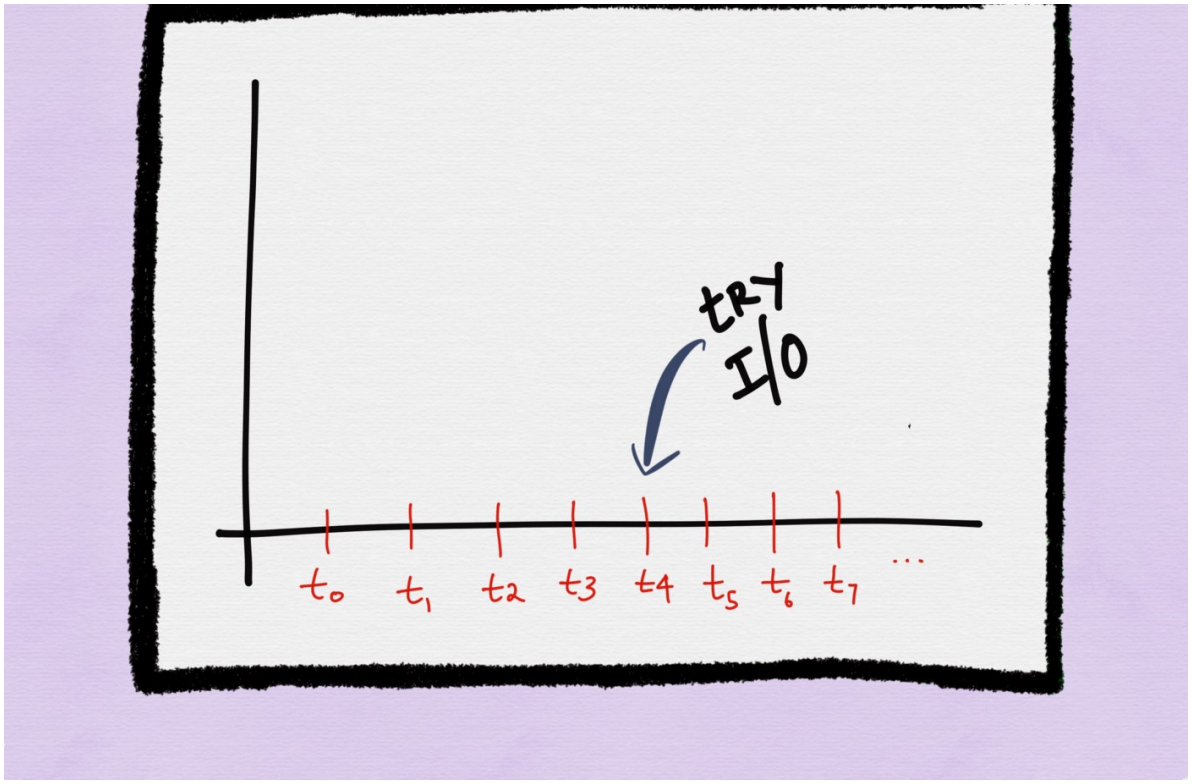




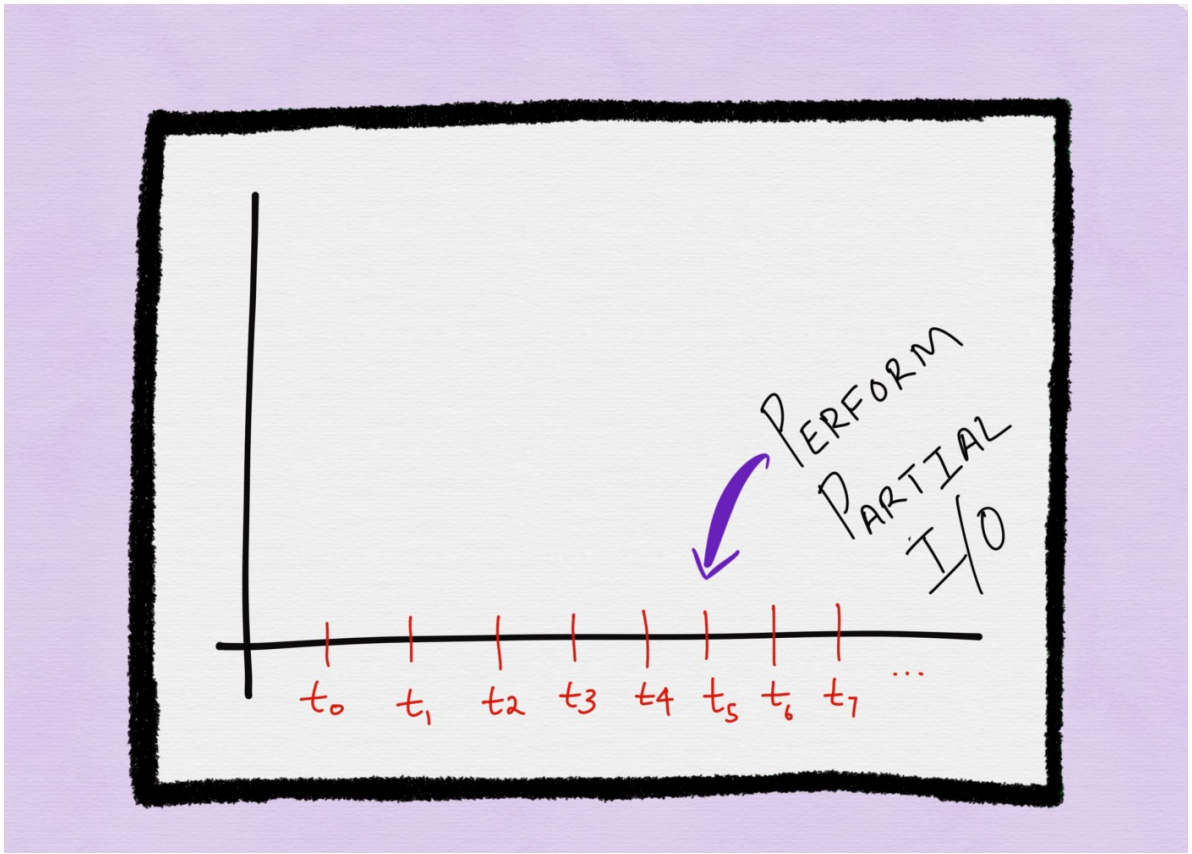
Let's say at time ***t3*** the process polls for the status of a descriptor and the descriptor is *ready*. The process can then chose to actually perform the entire I/O operation (read all the data available on the socket, for instance).



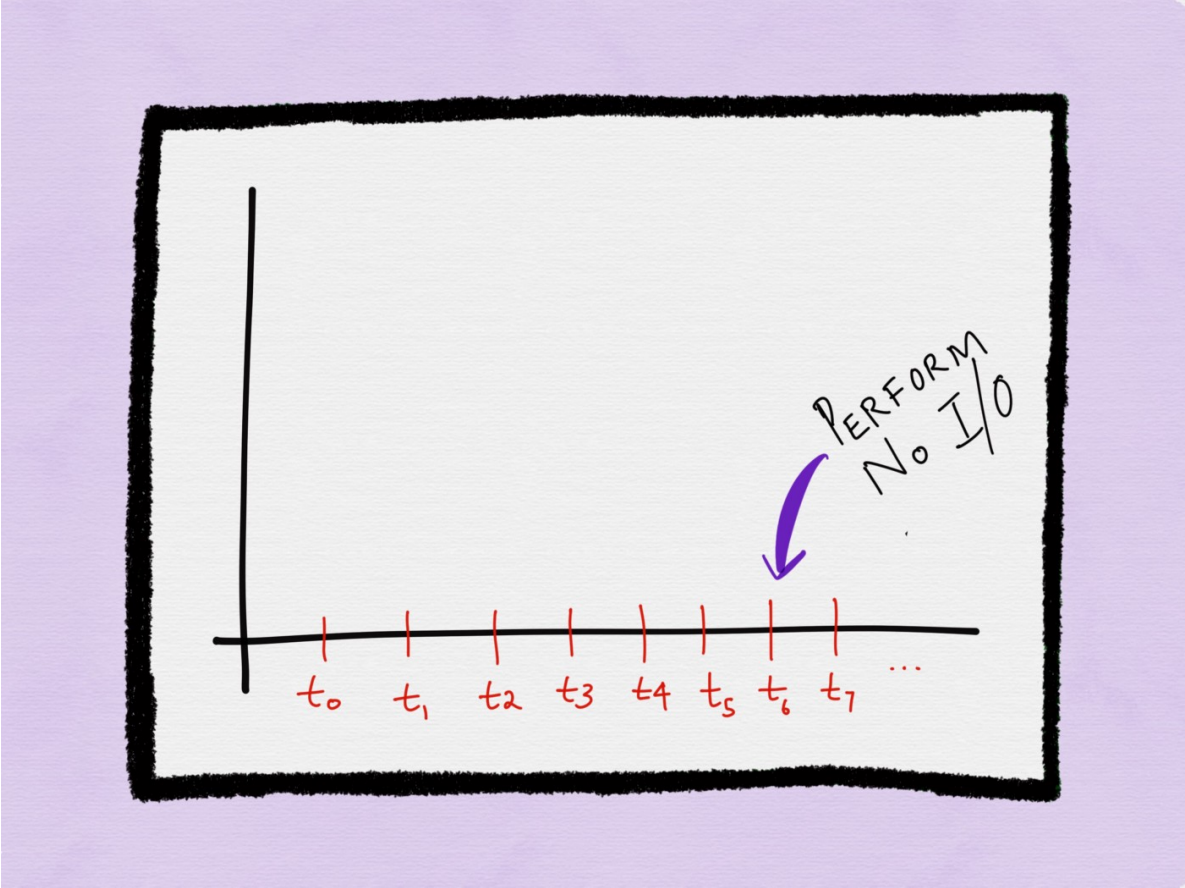
Let's assume at time ***t4*** the process polls for the status of a descriptor and the descriptor is not ready. The call blocks again and the I/O operation returns an error.



Let's assume at time ***t5*** the process polls for the status of a descriptor and the descriptor is ready. The process can subsequently choose to only perform a partial I/O operation (reading only half of all the data available, for instance).



Let's assume at time *t6* the process polls for the status of a descriptor and the descriptor is ready. This time the process may choose to perform no subsequent I/O at all.



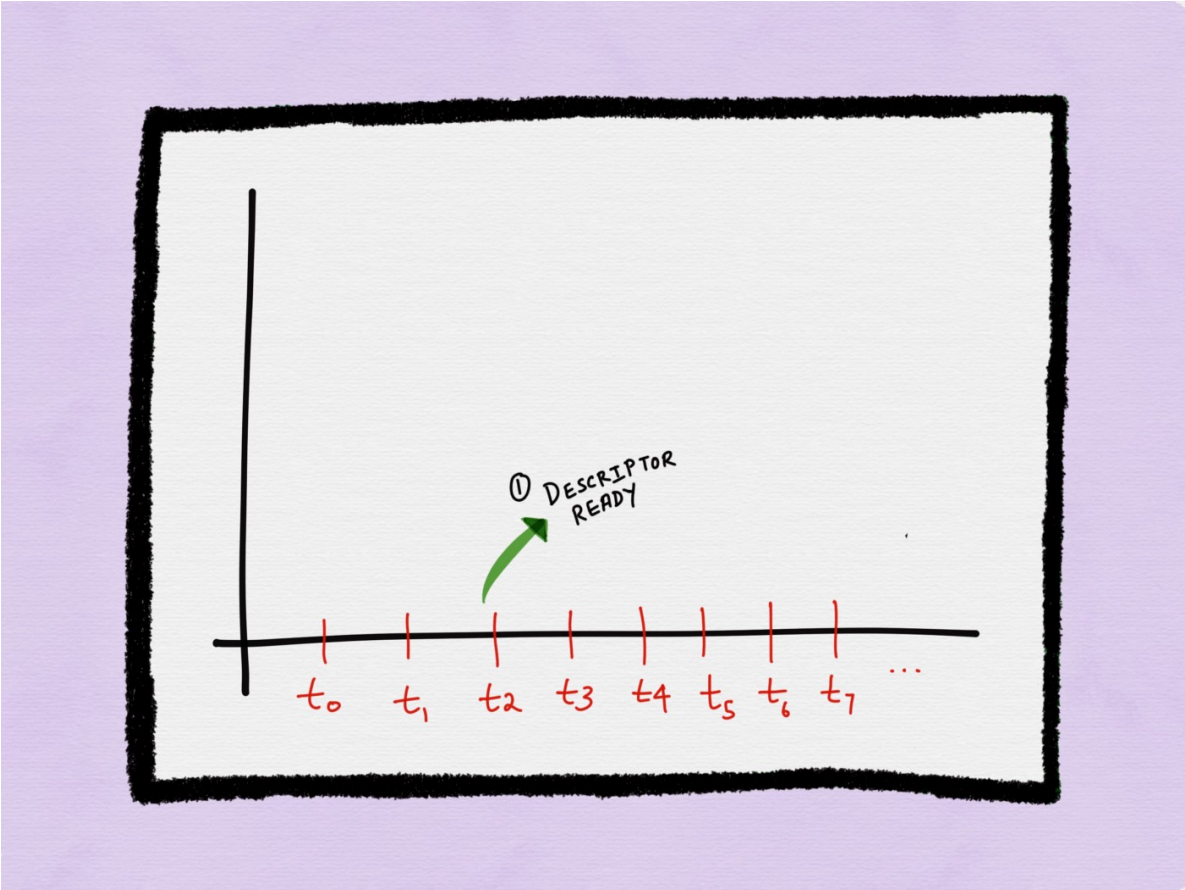
Edge Triggered

The process receives a notification only when the file descriptor is "ready" (usually when there is any new activity on the file descriptor since it was last monitored). I see this as the "push" model, in that a notification is pushed to the process about readiness of a file descriptor. Furthermore, with the push model, the process is only notified that a descriptor is ready for I/O, but not provided additional information like for instance how many bytes arrived on a socket buffer.

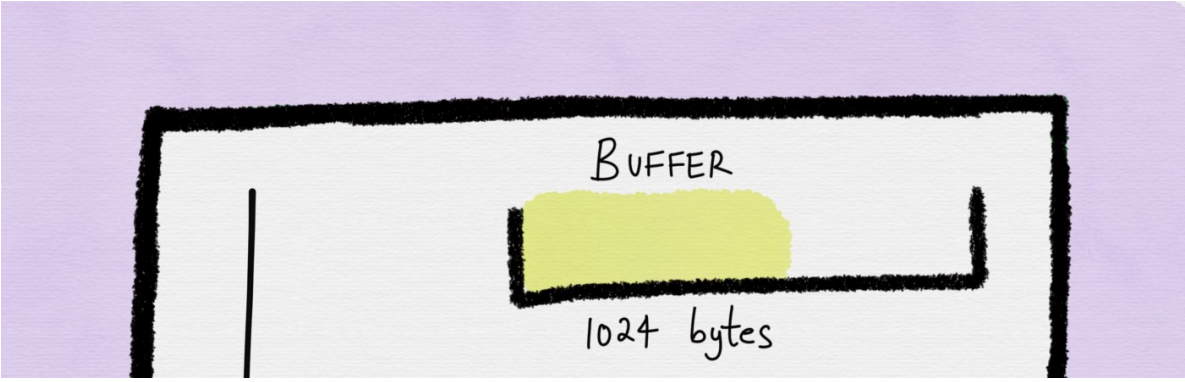
Thus, a process is only armed with incomplete data as it tries to

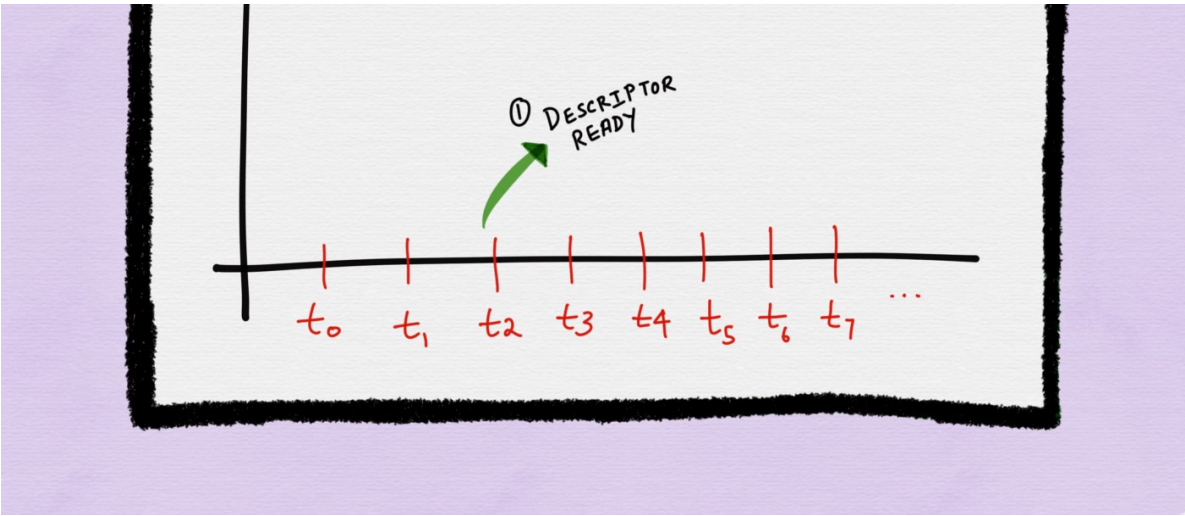
perform any subsequent I/O operation. To work around this, the process can attempt to perform the *maximum* amount of I/O it possibly can every time it gets a descriptor readiness notification, since failing to do this would mean the process would have to wait until the next notification arrives, even if I/O is possible on a descriptor before the arrival of the next notification.

Let's see how this works with the following example.

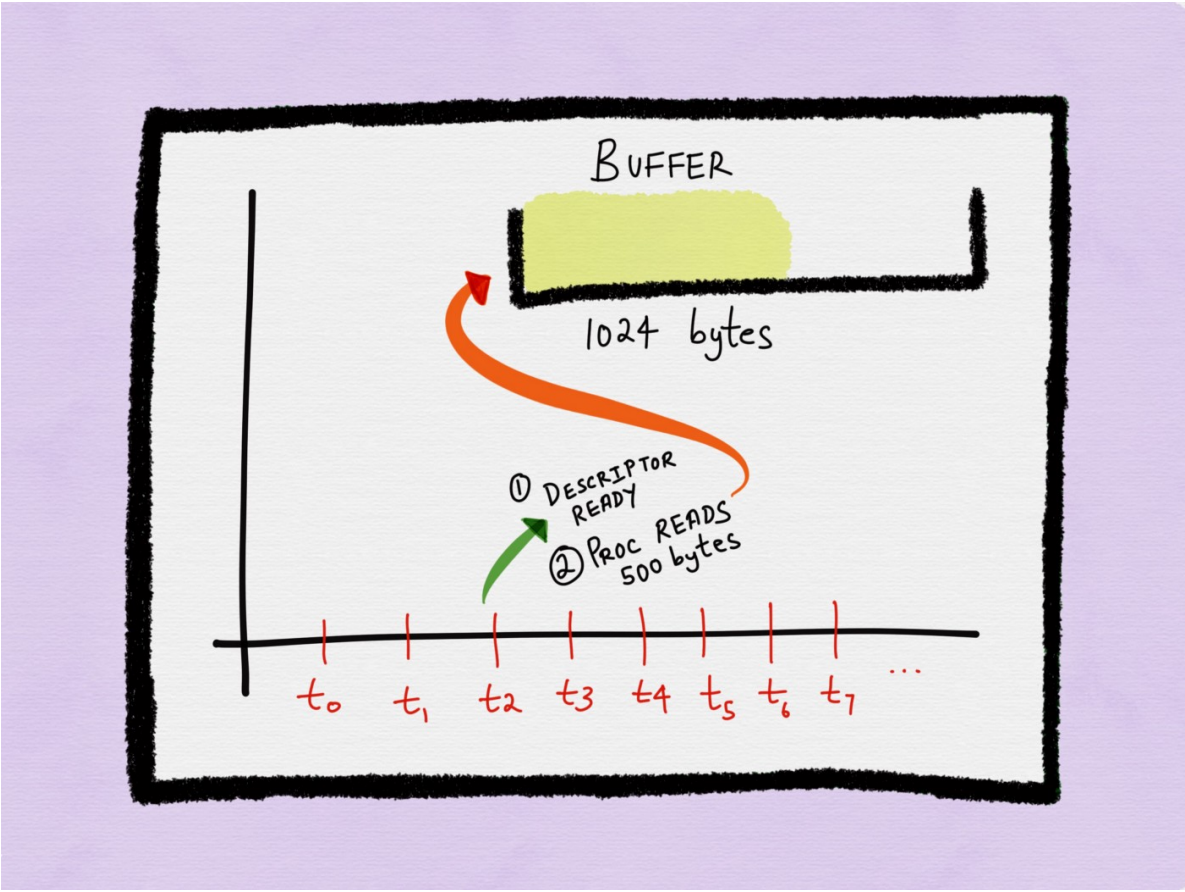


At time ***t2***, the process gets a notification about a descriptor being ready.

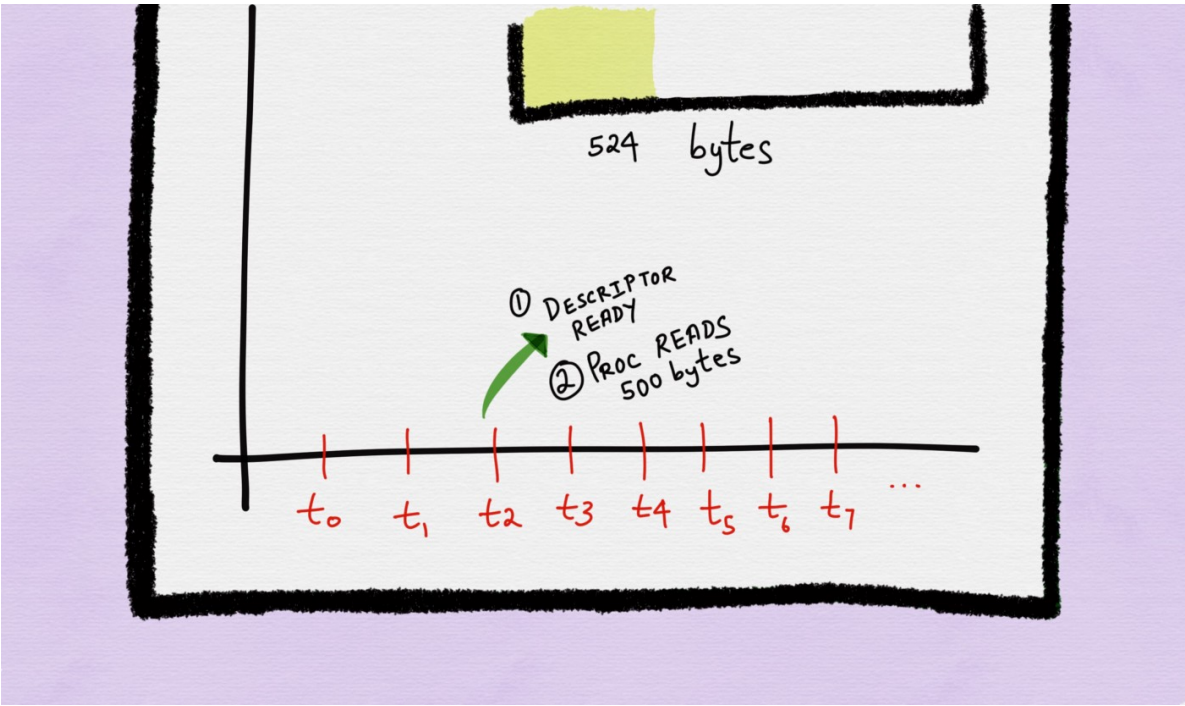




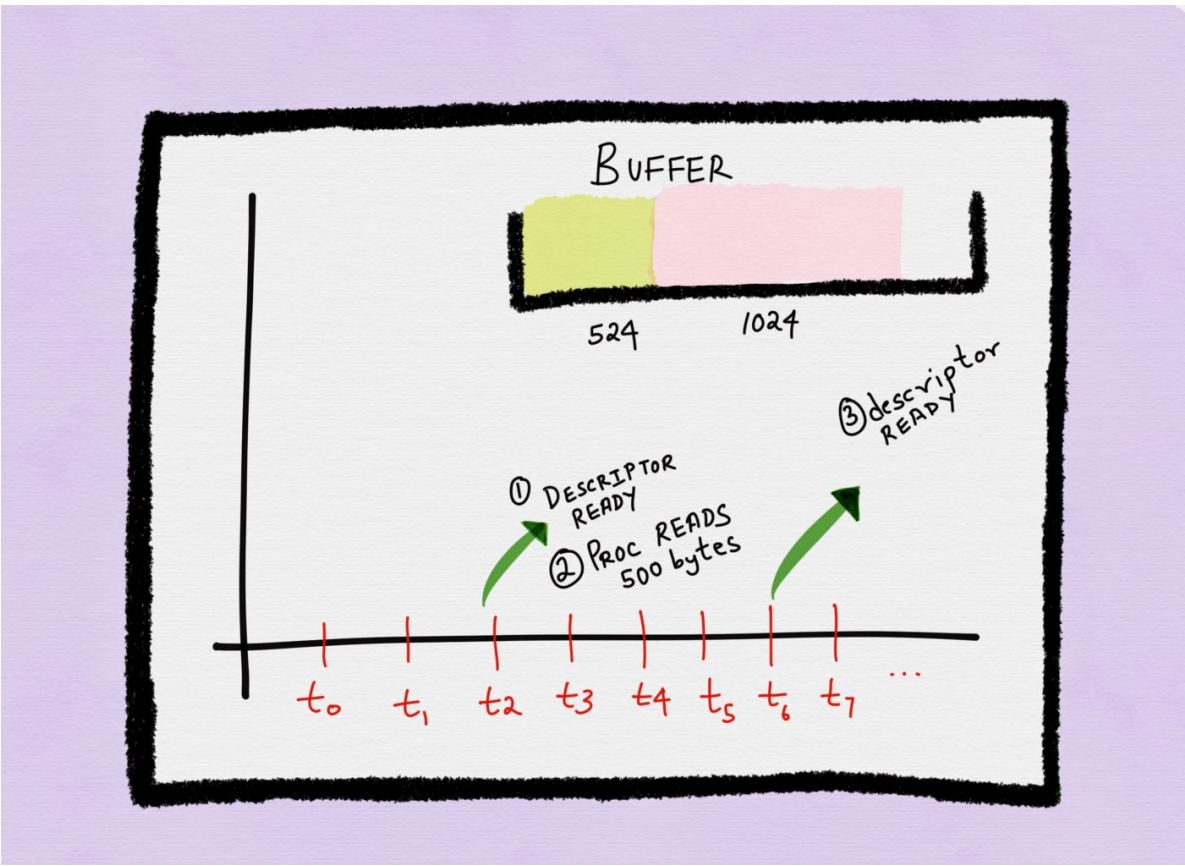
The byte stream available for I/O is stored in a buffer. Let's assume that 1024 bytes are available for reading when the process gets the notification at time ***t2.***



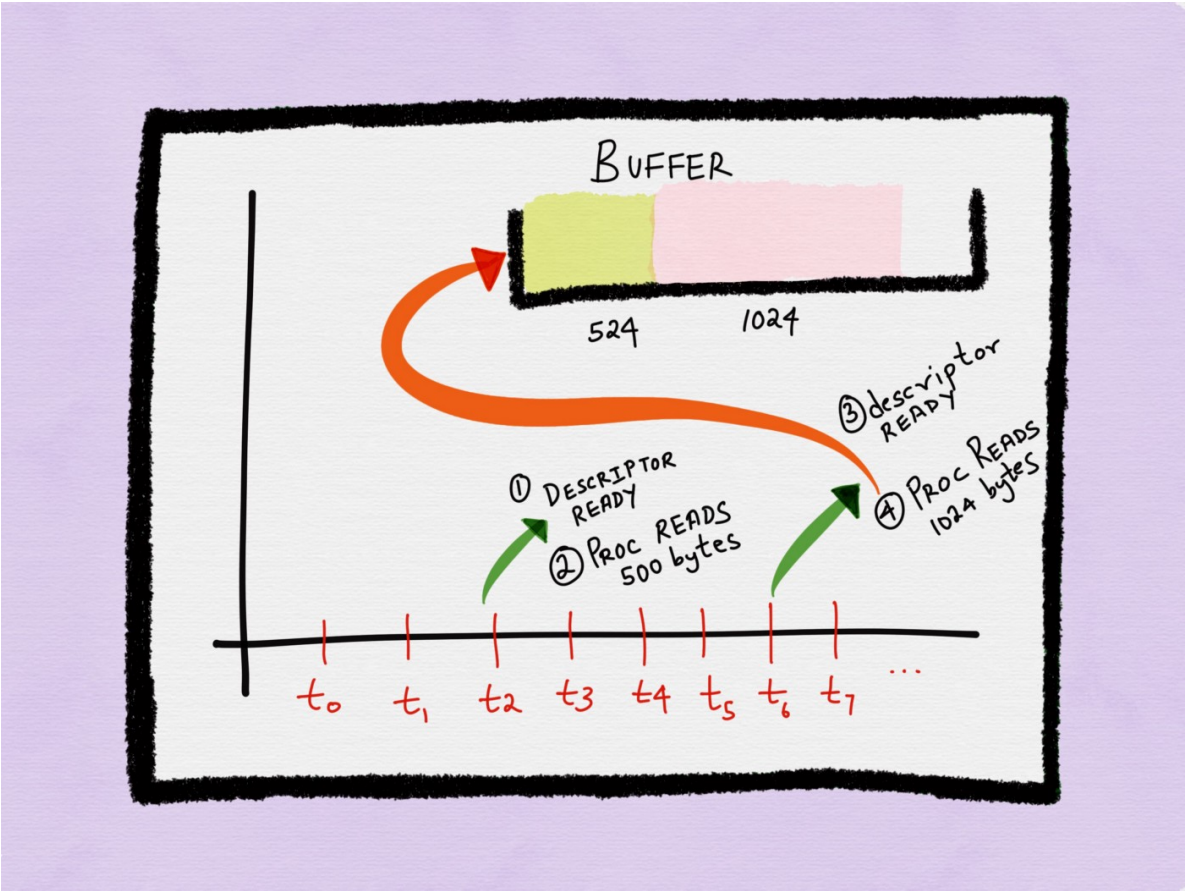
Let's assume the process only reads 500 out of the 1024 bytes.



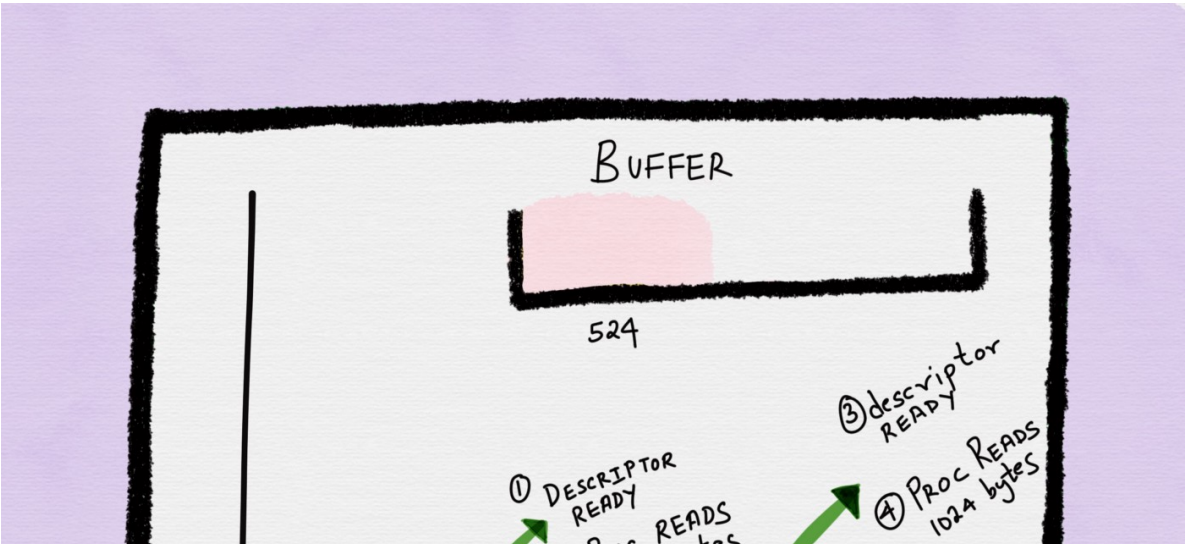
This means that at times ***t3***, ***t4*** *and* ***t5***, there are still 524 bytes available in the buffer that the process can read without blocking. But since the process can only perform I/O once it gets the next notification, these 524 bytes remain sitting in the buffer for that duration.

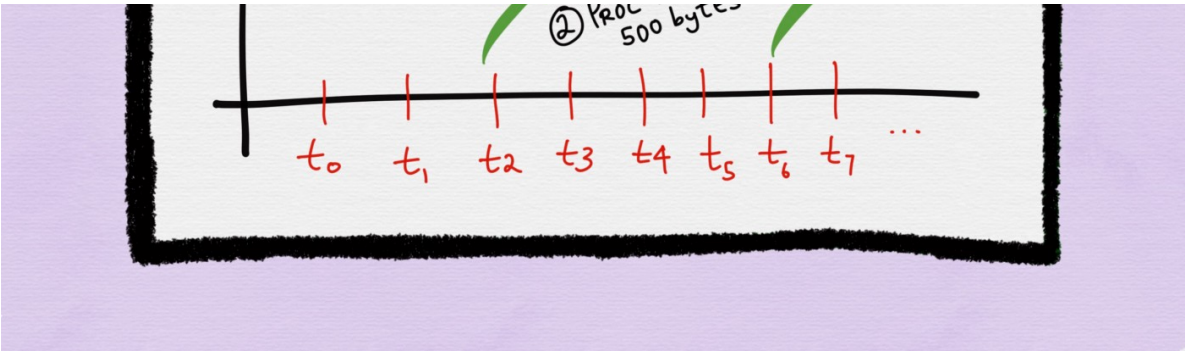


Let's assume the process gets the next notification at time **t6**, when 1024 additional bytes have arrived in the buffer. The total amount of data available on the buffer is now 1548 bytes — 524 bytes that weren't read previously, and 1024 bytes that have newly arrived.



Let's assume the process now reads in 1024 bytes.





This means that at the end of the second I/O operation, 524 bytes still remain in the buffer that the process will be unable to read before the next notification arrives.

While it might be temping to perform all the I/O immediately once a notification arrives, doing so has consequences. A large I/O operation on a single descriptor has the potential to starve other descriptors. **Furthermore, even with the case of level triggered notifications, an extremely large *write* or *send* call has the potential to block.**

Multiplexing I/O on descriptors

In the above section, we only described how a process handles I/O on a *single* descriptor. Often, a process might want to handle I/O on more than one descriptor. An extremely common use case is where a program needs to log to stdout and stderr, while also accept connections on a socket and make outgoing RPC connections to other services.

There are several ways of multiplexing I/O on descriptors:

— non-blocking I/O (the descriptor itself is marked as **non-blocking**, operations may finish partially)
— signal driven I/O (the process owning the descriptor is notified when the I/O state of the descriptor changes)

— polling I/O (with ***select*** or ***poll*** system calls, both of which provide **level triggered** notifications about the *readiness* of descriptors)
— BSD specific kernel event polling (with the ***kevent*** system call).

Multiplexing I/O with Non-Blocking I/O

What happens to the descriptors?

When we have multiple file descriptors, we could put all of them in the non-blocking mode.

What happens in the process?

The process can try to perform the I/O operation on the descriptors to check if any of the I/O operations result in an error.

What happens in the kernel?

The kernel performs the I/O operation on the descriptor and returns an error or a partial output or the result of the I/O operation if it succeeds.

What are the cons?

Frequent checks: If a process tries to perform I/O operations very frequently, the process has to continuously be retrying operations that returned an error to check if any descriptors are ready. Such busy-waiting in a tight loop could lead to burning CPU cycles.

Infrequent checks: If such operations are conducted infrequently, then it might take a process an unacceptably long time to respond to an I/O event that is available.

When it might make sense to use this approach?

Operations on **output descriptors** (*writes* for example) don't generally block. In such cases, it might help to try to perform the I/O operation first, and revert back to polling when the operation returns an error. It might also make sense to use this approach with *edge-triggered* notifications, where the descriptors can be put in nonblocking mode, and once a process gets notified of an I/O event, it can repeatedly try I/O operations until the system calls would block with an `EAGAIN` or `EWOULDBLOCK`.

Multiplexing I/O via Signal Driven I/O

What happens to the descriptors?

The kernel is instructed to send the process a signal when I/O can be performed on any of the descriptors.

What happens in the proceess?

The process will wait for signals to be delivered when any of the descriptor is ready for an I/O operation.

What happens in the kernel?

Tracks a list of descriptors and sends the process a signal every time any of the descriptors become ready for I/O.

What are the cons of this approach?

Signals are expensive to catch, rendering signal driven I/O impractical for cases where a large amount of I/O is performed.

When it might make sense to use this approach?

It is typically used for “exceptional conditions” when the cost of handling the signal is lower than that of polling constantly with *select*/*poll*/*epoll* or *kevent*. An example of an “exceptional case” is the arrival of out-of-band data on a socket or when a state change occurs on a pseudoterminal secondary connected to a master in packet mode.

Multiplexing I/O via Polling I/O

What happens to the descriptors?

The descriptors are put in a non-blocking mode.

What happens in the process?

The process uses the **level triggered mechanism** to ask the kernel by means of a system call (*select* or *poll*) which descriptors are capable of performing I/O. Described below is the implementation of both *select* and *poll.*

Select

The signature of *select* on Darwin is:

```
int
select(
    int nfds,
    fd_set *restrict readfds,
    fd_set *restrict writefds,
    fd_set *restrict errorfds,
    struct timeval *restrict timeout
```

);

while on Linux it is:

```
int
select(
  int nfds,
  fd_set *readfds,
  fd_set *writefds,
  fd_set *exceptfds,
  struct timeval *timeout
);
```

Select monitors three independent sets of descriptors:

— the ***readfds*** descriptors are monitored to see if a *read* will not block (when bytes become available for reading or when encountering an EOF)
— the ***writefds*** descriptors are monitored to for when a *write* will not block.
— the ***exceptfds*** descriptors are monitored for exceptional conditions

When ***select*** returns, the descriptors are modified in place to indicate which file descriptors actually changed status. Any of the three file descriptor sets can be set to NULL if we don't want to monitor that particular category of events.

The final argument is a ***timeout*** value, which specifies for how long the ***select*** system call will block:

— when the ***timeout*** is set to 0, ***select*** does not block but returns immediately after polling the file descriptors
— when ***timeout*** is set to NULL, ***select*** will block "forever". When

select blocks, the kernel can put the process to sleep until ***select*** returns. ***Select*** will block until 1) one or more descriptors specified in the three sets described above are ready or 2) the call is interrupted by a signal handler
— when ***timeout*** is set to a specific value, then ***select*** will block until 1) one or more descriptors specified in the three sets described above are ready or 2) the call is interrupted by a signal handler or 3) the amount of time specified by ***timeout*** has expired

The return values of ***select*** are the following:

— if an error (EBADF or EINTR) occurred, then the return code is -1
— if the call timed out before any descriptor became ready, then the return code is 0
— if one or more file descriptors are ready, then the return code is a positive integer which indicates the total number of file descriptors in all the three sets that are ready. Each set is then individually inspected to find out which I/O event occurred.

Poll

Poll differs from *select* only in terms of how we specify ***which*** descriptors to track.

With ***select***, we pass in three sets of descriptors we want to monitor for reads, writes and exceptional cases.

With ***poll*** we pass in a set of descriptors **each marked with the *events it specifically needs to track*.**

The signature of ***poll*** on Darwin is:

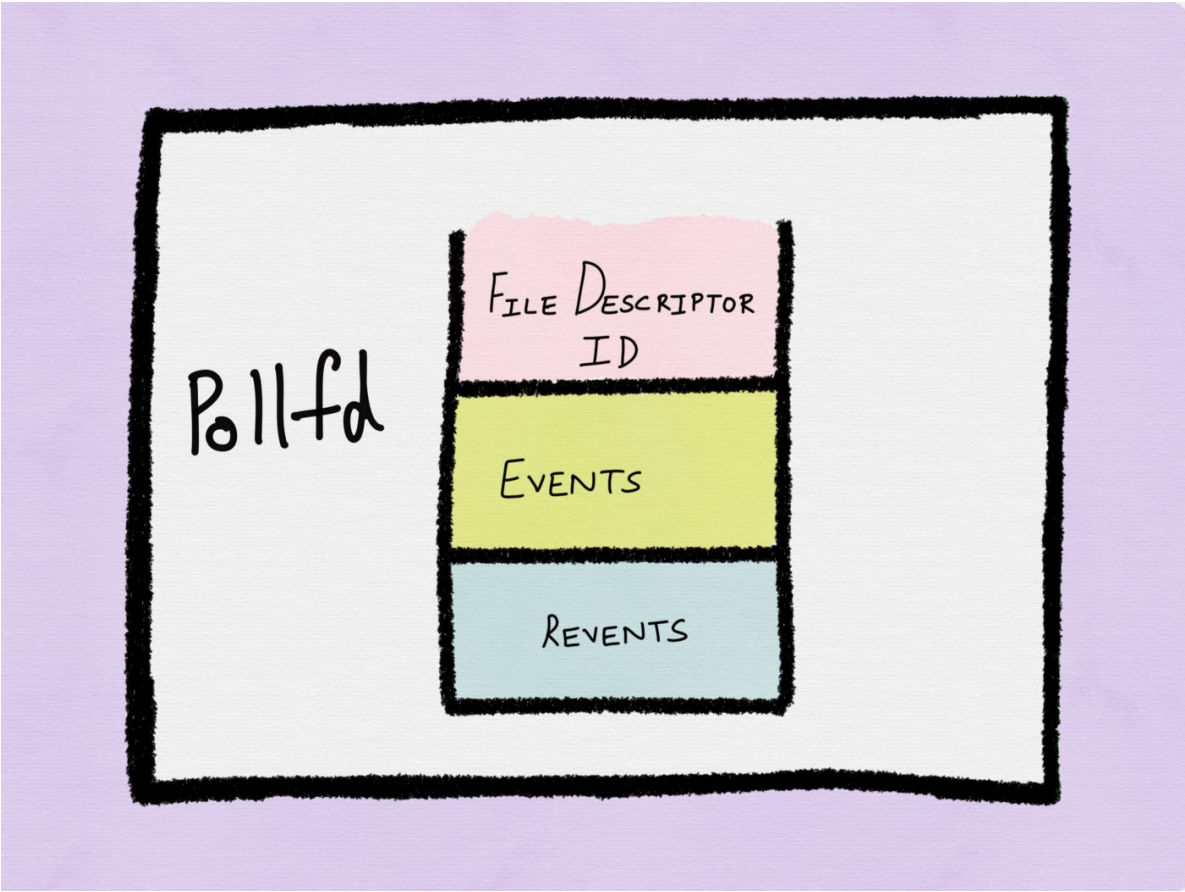
int poll(

```
    struct pollfd fds[],
    nfds_t nfds,
    int timeout
);
```

And on Linux it is:

```
int poll(
    struct pollfd *fds,
    nfds_t nfds,
    int timeout
);
```

The first argument to ***poll*** is an array of all the descriptors we want to monitor.



A *pollfd* structure contains three pieces of information:

— the ID of the descriptor to poll (let's call this descriptor A)

— bit masks indicating what events to monitor for the given descriptor A (***events***)
— bit masks set by the kernel indicating the events that actually occurred on the descriptor A (***revents***)

The second argument is the total number of descriptors we are monitoring (the length of the array of descriptors used as the first argument, in other words).

The third ***timeout*** specifies for how long ***poll*** would block for every time it is invoked.

— when set to -1, *poll* blocks indefinitely until one of the descriptors listed in the first argument is ready or when a signal is caught
— when set to 0, *poll* does not block but returns immediately after polling to check which of the file descriptors, if any, are ready
— when set to a value greater than 0, *poll* blocks until *timeout* milliseconds or until one of the file descriptors becomes ready or until a signal is caught, whichever one is first.

The return values of ***poll*** are the following:

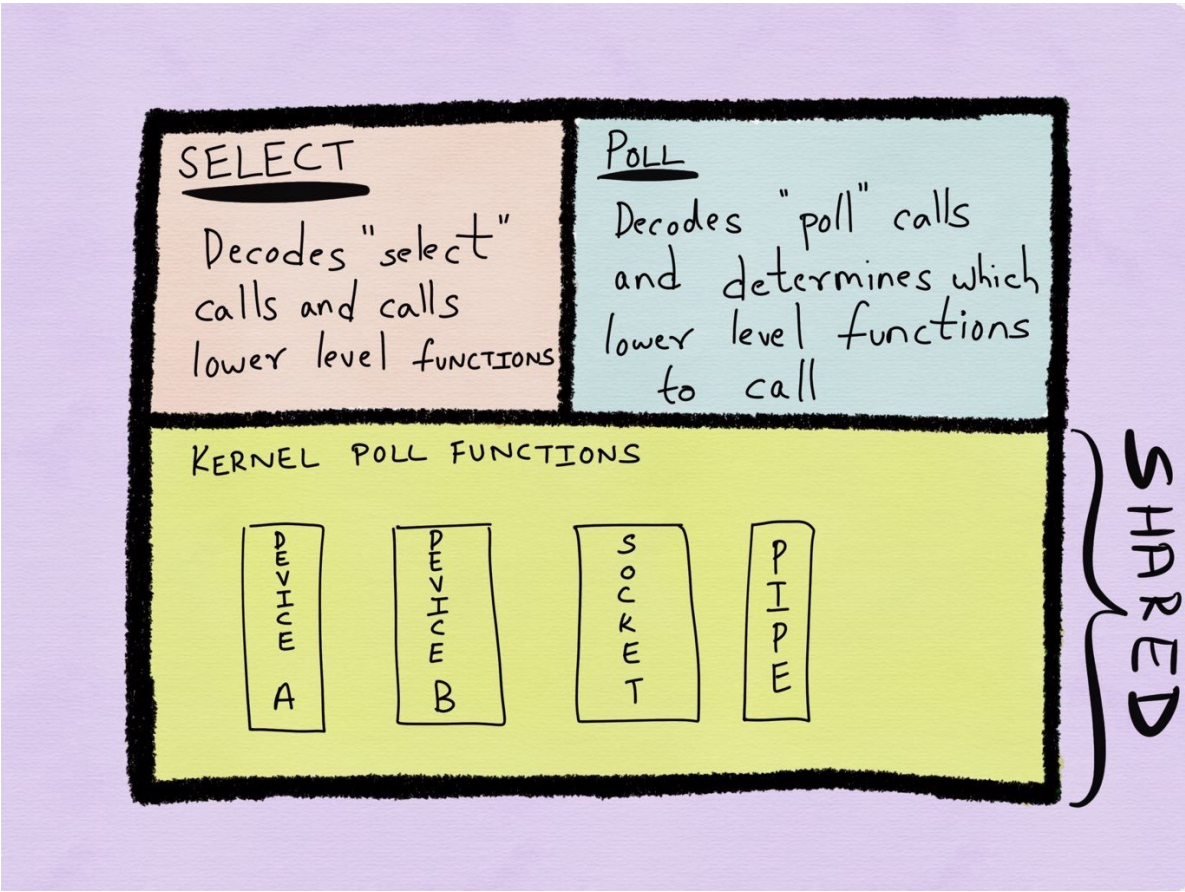
— if an error (EBADF or EINTR) occurred, then the return code is -1
— if the call timed out before any descriptor became ready, then the return code is 0
— if one or more file descriptors are ready, then the return code is a positive integer. This number is the total number of file descriptors in the array on which events have happened. If the number file descriptors in the array is 10, and 4 of them had events that happened, then the return value is 4. The *revents field can be inspected* to find out which of the events actually occurred

for the file descriptor.

What happens in the kernel?

Both **select** and **poll** are *stateless*. Every time a *select* or a *poll* system call is made, the kernel checks *every descriptor* in the input array passed as the first argument for the occurrence an event and return the result to the process. This means that the cost of ***poll*/select** is O(N), where N is the number of descriptors being monitored.

Furthermore, the implementation of both ***select*** and ***poll*** comprises of two tiers — a specific top tier which decodes the incoming request, as well as several device or socket specific bottom layers. The bottom layers compriseof *kernel poll functions* and is used by both ***select*** and ***poll.***



What are the cons of this approach?

The process performs two system calls — ***select***/***poll*** to find out which descriptors are ready to perform I/O and another system call to do the actual operation (***read***/***write***).

Second, both ***select*** and ***poll*** only allow descriptors to be monitored for *three events* — reading, writing and exceptional conditions. A process might be interested in knowing about other events as well, such as signals, filesystem notifications or asynchronous I/O completions.

Furthermore, ***select*** and ***poll*** do not scale very well as the number of descriptors being monitored increases. As stated above, the cost of ***select***/***poll*** is O(N), which means when N is very large (think of a web server handling tens of thousands of mostly sleepy clients), every time ***select***/***poll*** is called, even if there might only be a small number of events that actually occurred (4, in the example above), the kernel still needs to scan *every* descriptor in the list (10 in the example above) and check for all the three conditions on each descriptor, and invoke the appropriate callbacks registered. This also means that once the kernel responds back to the process with the status of every descriptor, the process must then scan the entire list of descriptors in the response to determine which descriptor is ready.

When it might make sense to use this approach?

Perhaps when there the number of descriptors being monitored for I/O alone is small and these descriptors are mostly busy?

Kernel event polling on BSD

What happens to the descriptors?

The descriptors are put in non-blocking mode.

What happens in the process?

The process is provided a generic notification interface known as *kevent* that allows it to monitor a wide variety of events and be notified of these kernel event activities in a scalable manner. For example, the process can track:

— changes to file attributes when a file is renamed or deleted
— posting of signals to it when it *forks*, *execs* or *exits*
— asynchronous I/O completion

The kernel **only returns a list of the events that have occurred**, instead of returning the status of *every event the process has registered.* The kernel builds the event notification structure just once and the process is notified every time one of the events occur. If a process is tracking N events, and only a small number of those events have occurred (M), the cost is O(M) as opposed to O(N). Thus *kevent* scales well when only a small subset of events out of all the events the process is interested in occurs. In other words, this is especially good when there are a large number of sleepy clients or slow clients, since the number of events a process needs to be notified remains small, even if the number of events the process is monitoring is large.

The system call used to create a descriptor to use as a handle on which *kevents* can be registered is *kqueue* on BSD/Darwin. This descriptor is the means by which the process then gets the notification about the occurrence of events that it registered. This

descriptor is also the means by which a process can add or delete events it wishes to be notified about.

What happens in the kernel?

The process subscribes to specific events on descriptors. The kernel provides a system call a process can invoke to find out which events have occurred. If the kernel has a list of events that have occurred, the system call returns. If not, the process is put to sleep until an event occurs.

Async I/O in POSIX

POSIX defines a specification for “parallel I/O”, by allowing a process to instantiate I/O operations without having to block or wait for any to complete. The I/O operation is queued to a file and the process is later notified when the operation is complete, at which point the process can retrieve the results of the I/O.

Linux exposes functions such as `io_setup`, `io_submit`, `io_getevents`, `io_destroy`, which allow for submission of I/O requests from a thread without blocking the thread. This is for disk I/O (especially random IO on SSD). Especially interesting is how the addition of *eventfd* support makes it possible to use it with *epoll.* Here’s a good description of how this works.

On Linux, another way of performing parallel I/O is using the threads based implementation is available within glibc:

> These functions are part of the library with realtime functions named **librt**. They are not actually part of the libc binary. The implementation of these functions can be done using support in the kernel (if available) or using an implementation based on

> threads at user level. In the latter case it might be necessary to link applications with the thread library **libpthread** in addition to **librt**.

This approach, however, is not without its cons:

> This has a number of limitations, most notably that maintaining multiple threads to perform I/O operations is expensive and scales poorly. Work has been in progress for some time on a kernel state-machine-based implementation of asynchronous I/O (see **io_submit**(2), **io_setup**(2), **io_cancel**(2), **io_destroy**(2), **io_getevents**(2)), but this implementation hasn't yet matured to the point where the POSIX AIO implementation can be completely reimplemented using the kernel system calls.

On FreeBSD, POSIX AIO is implemented with the *aio* system call. An async "kernel process" (also called "kernel I/O daemon" or "AIO daemon") performs the queued I/O operations.

AIO daemons are grouped into configurable pools. Each pool will add or remove AIO daemons based on load. One pool of AIO daemons is used to service asynchronous I/O requests for sockets. A second pool of AIO daemons is used to service all other asynchronous I/O requests except for I/O requests to raw disks.

To perform an asynchronous I/O operation:

— the kernel creates an async I/O request structure with all the information needed to perform the operation
— if the request cannot be satisfied by the kernel buffers immediately, this request structured is queued
— If an AIO daemon isn't available at the time of request creation, the request structure is queued for processing and the syscall

returns.
— the next available AIO daemon handles the request using the kernel's sync path
— when the daemon finishes the I/O, the request structure is marked as finished along with a return or error code.
— the process uses the aio_error syscall to poll if the I/O is complete. This call is implemented by inspecting the status of the async I/O request structure created by the kernel
— if a process gets to the point where it cannot proceed until I/O is complete, then it can use the aio_suspend system call to wait until the I/O is complete.
— the process is put to sleep on the AIO request structure and is awakened by the AIO daemon when I/O completes or the process requests a signal be sent when the I/O is done
— once aio_suspend, aio_error or a signal indicating the I/O completion has arrived, using the aio_return system call gets the return value of the I/O operation.

Conclusion

This post only shed light on the forms on I/O possible and didn't touch on *epoll* at all, which is by far the most interesting of all in my opinion. What's often more interesting is buffer and memory management. My next post explores epoll in much greater detail.