The method to epoll's madness

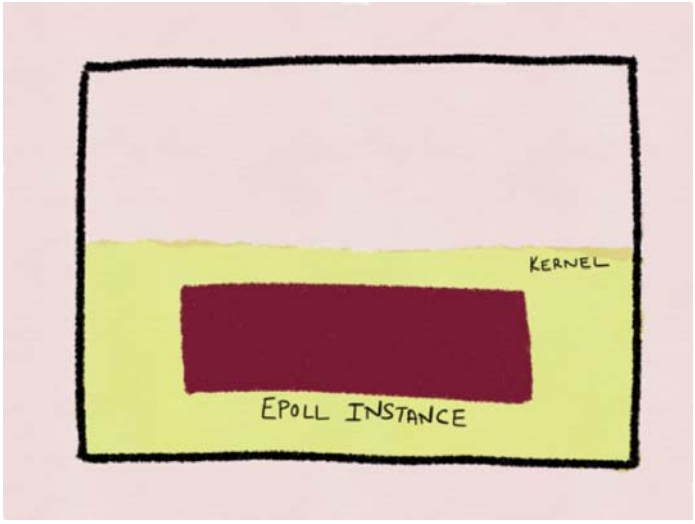
Cindy Sridharan Oct 30, 2017 · 16 min read

My previous post covered the fundamentals of file descriptors as well as some of the most commonly used forms on non-blocking I/O operations on Linux and BSD. I had some people wonder why it didn't cover **epoll** at all, but I'd mentioned in the conclusion of that post that **epoll** is by far the most interesting of all and as such warranted a separate post in its own right.

epoll stands for *event poll* and is a Linux specific construct. It allows for a process to monitor multiple file descriptors and get notifications when I/O is possible on them. It allows for both *edge-triggered* as well as *level-triggered* notifications. Before we look into the bowels of *epoll*, first let's explore the syntax.

The syntax of epoll

Unlike *poll*, *epoll* itself is not a system call. It's a kernel data structure that allows a process to multiplex I/O on multiple file descriptors.



This data structure can be created, modified and deleted by three system calls.

1) epoll_create

The *epoll* instance is created by means of the `epoll_create` system call, which returns a file descriptor to the *epoll* instance. The signature of

epoll_create is as follows:

```
#include <sys/epoll.h>
int epoll_create(int size);
```

The ***size*** argument is an indication to the kernel about the number of file descriptors a process wants to monitor, which helps the kernel to decide the size of the *epoll* instance. Since Linux 2.6.8, this argument is ignored because the *epoll* data structure dynamically resizes as file descriptors are added or removed from it.

The ***epoll_create*** system call returns a file descriptor to the newly created *epoll* kernel data structure. The calling process can then use this file descriptor to add, remove or modify *other* file descriptors it wants to monitor for I/O to the *epoll* instance.

There is another system call ***epoll_create1*** which is defined as follows:

```
int epoll_create1(int flags);
```

The ***flags*** argument can either be 0 or **EPOLL_CLOEXEC.**

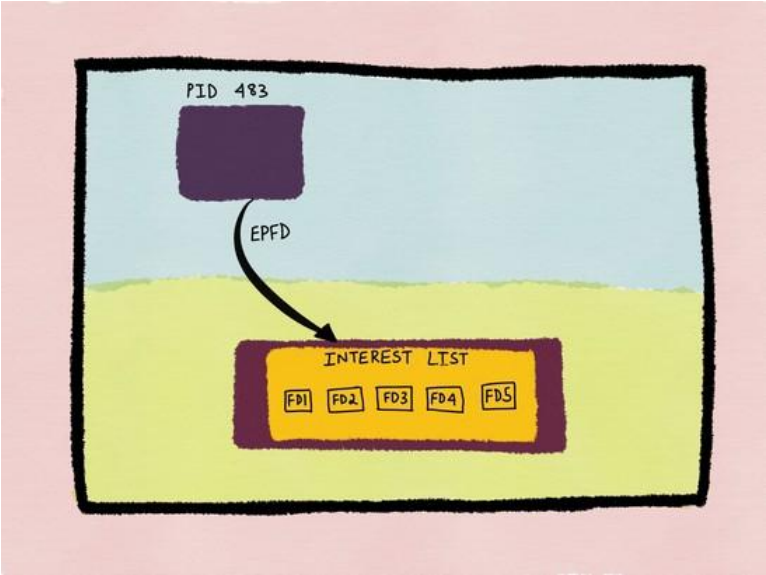
When set to 0, *epoll_create1* behaves the same way as *epoll_create*.

When the **EPOLL_CLOEXEC** flag is set, any child process forked by the current process will close the *epoll* descriptor before it *execs,* so the child process won't have access to the *epoll* instance anymore.

It's important to note that the file descriptor associated with the *epoll* instance needs to be released with a *close()* system call. Multiple processes might hold a descriptor to the same *epoll* instance, since, for example, a *fork* without the **EPOLL_CLOEXEC** flag will duplicate the descriptor to the *epoll* instance in the child process). When all of these processes have relinquished their descriptor to the *epoll* instance (by either calling *close()* or by exiting), the kernel destroys the *epoll* instance.

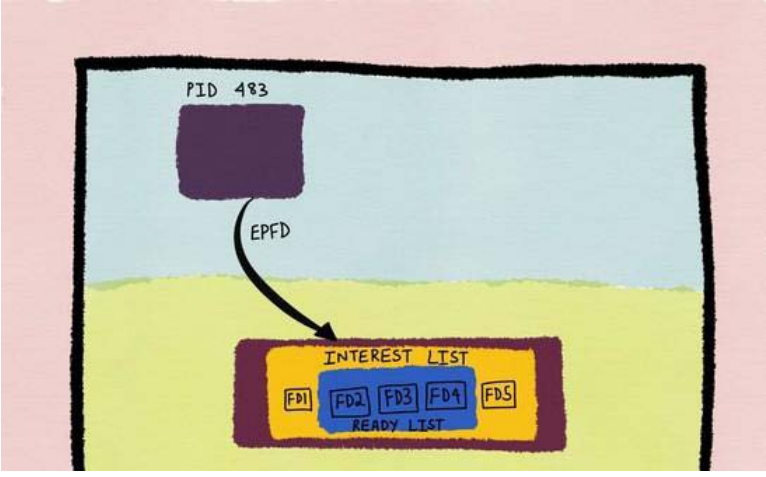
2) epoll_ctl

A process can add file descriptors it wants monitored to the *epoll* instance by calling `epoll_ctl`. All the file descriptors registered with an *epoll* instance are collectively called an ***epoll set*** or the ***interest list***.



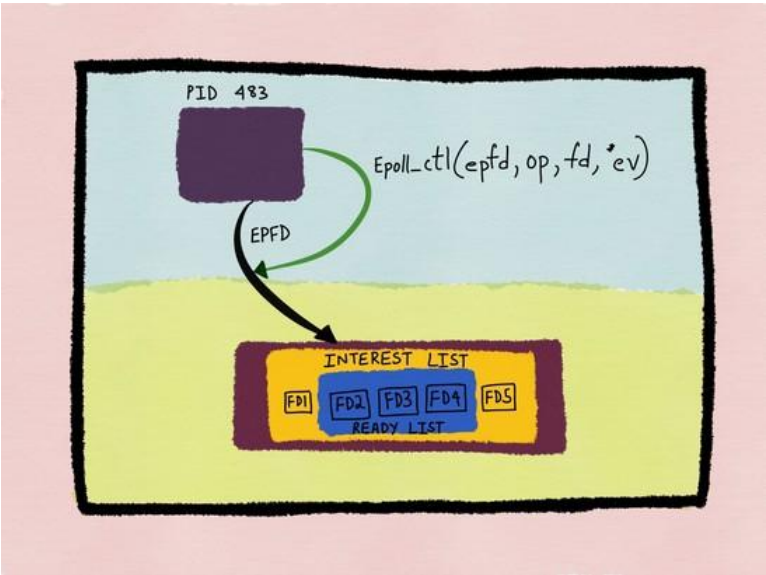
In the above diagram, process 483 has registered file descriptors ***fd1***, ***fd2***, ***fd3***, ***fd4*** and ***fd5*** with the *epoll* instance. This is the ***interest list*** or the ***epoll set*** of that particular *epoll* instance. Subsequently, when any of the file descriptors registered become ready for I/O, then they are considered to be in the ***ready list***.

The ***ready list*** is a subset of the ***interest list***.



The signature of the `epoll_ctl` syscall is as follows:

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

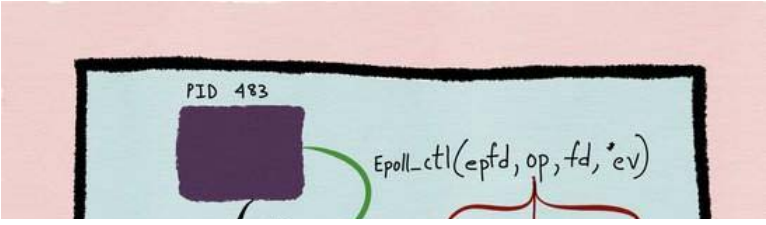


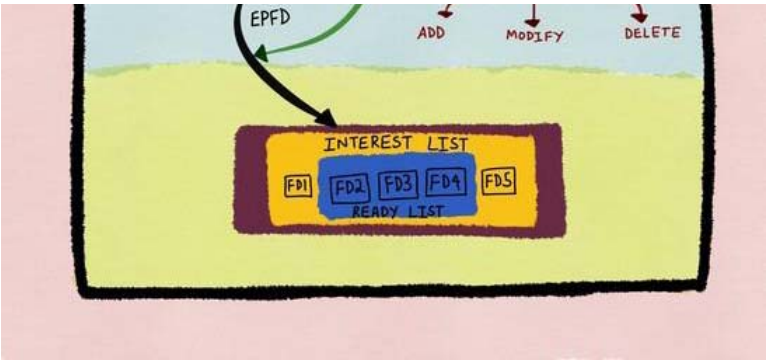
epfd **—** is the file descriptor returned by `epoll_create` which identifies the *epoll* instance in the kernel.

fd **—** is the file descriptor we want to add to the ***epoll list/interest list***.

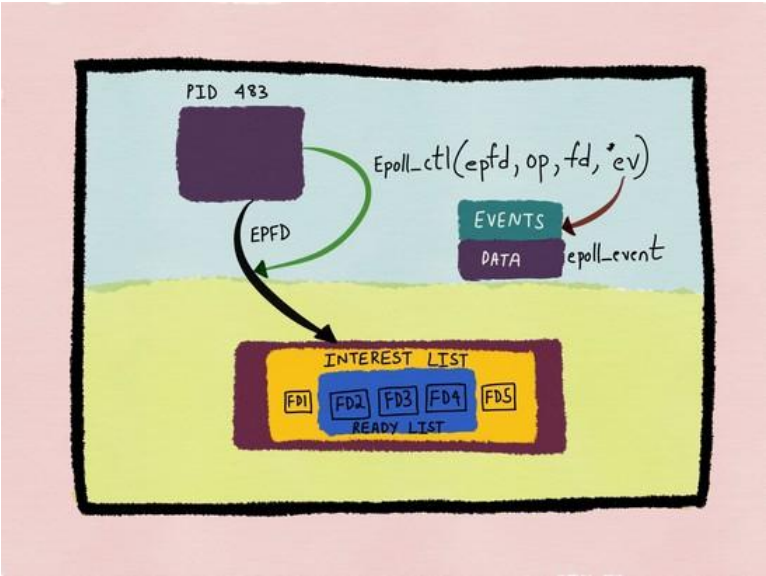
op **—** refers to the operation to be performed on the file descriptor *fd*. In general, three operations are supported:

— **Register** ***fd*** with the *epoll* instance (**EPOLL_CTL_ADD**) and get notified about events that occur on ***fd***
— **Delete**/deregister ***fd*** from the *epoll* instance. This would mean that the process would no longer get any notifications about events on that file descriptor (**EPOLL_CTL_DEL**). If a file descriptor has been added to multiple *epoll* instances, then closing it will remove it from all of the *epoll* interest lists to which it was added.
— **Modify** the events ***fd*** is monitoring (**EPOLL_CTL_MOD**)





event **—** is a pointer to a structure called ***epoll_event*** which stores the *event* we actually want to monitor ***fd*** for.



The first field ***events*** of the *epoll_event* structure is a *bitmask* that indicates which events ***fd*** is being monitored for.

Like so, if ***fd*** is a socket, we might want to monitor it for the arrival of new data on the socket buffer (**EPOLLIN).** We might also want to monitor ***fd*** for edge-triggered notifications which is done by OR-ing **EPOLLET** with **EPOLLIN.** We might also want to monitor ***fd*** for the occurrence of a registered event but only *once* and stop monitoring ***fd*** for subsequent occurrences of that event. This can be accomplished by OR-ing the other flags (**EPOLLET, EPOLLIN)** we want to set for descriptor ***fd*** with the flag for only-once notification delivery **EPOLLONESHOT.** All possible flags can be found in the man page.

The second field of the *epoll_event* struct is a union field.

3) epoll_wait

A thread can be notified of events that happened on the ***epoll set/interest set*** of an *epoll* instance by calling the `epoll_wait` system call, which blocks until any of the descriptors being monitored becomes *ready* for I/O.

The signature of `epoll_wait` is as follows:

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist, int maxevents,
int timeout);
```

epfd — is the file descriptor returned by `epoll_create` which identifies the *epoll* instance in the kernel.

evlist — is an array of ***epoll_event*** structures. ***evlist*** is allocated by the calling process and when ***epoll_wait*** returns, this array is modified to indicate information about the subset of file descriptors in the interest list that are in the *ready* state (this is called the ***ready list***)

maxevents — is the length of the evlist array

timeout — this argument behaves the same way as it does for ***poll*** *or* ***select***. This value specifies for how long the ***epoll_wait*** system call will block:

— when the ***timeout*** is set to 0, ***epoll_wait*** does not block but returns immediately after checking which file descriptors in the interest list for *epfd* are *ready*
— when ***timeout*** is set to -1, ***epoll_wait*** will block “forever”. When ***epoll_wait*** blocks, the kernel can put the process to sleep until ***epoll_wait*** returns. ***epoll_wait*** will block until 1) one or more descriptors specified in the interest list for *epfd* become ready or 2) the call is interrupted by a signal handler
— when ***timeout*** is set to a non negative and non zero value, then ***epoll_wait*** will block until 1) one or more descriptors specified in the interest list for *epfd* becomes ready or 2) the call is interrupted by a signal handler or 3) the amount of time specified by ***timeout*** milliseconds have expired

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

— if an error (EBADF or EINTR or EFAULT or EINVAL) occurred, then the return code is -1
— if the call timed out before any file descriptor in the interest list became ready, then the return code is 0
— if one or more file descriptors in the interest list became *ready*, then the return code is a positive integer which indicates the total number of file descriptors in the *evlist* array. The *evlist* is then examined to determine which events occurred on which file descriptors.

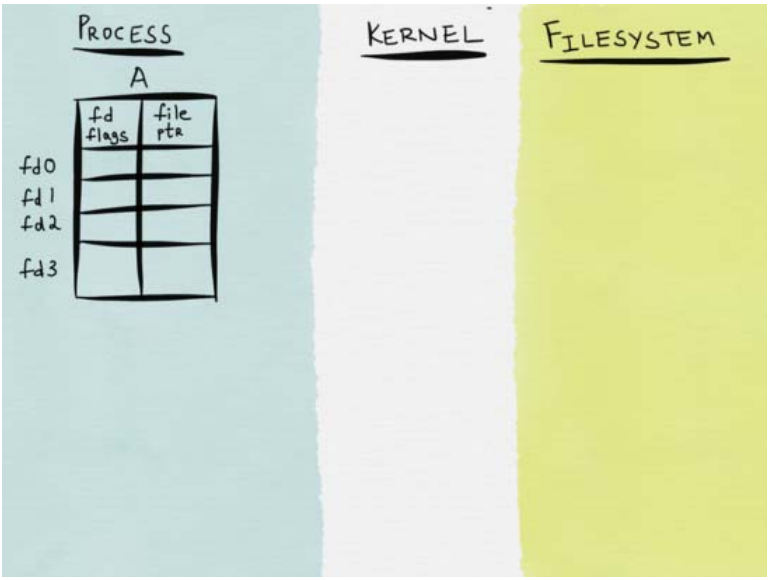
The gotchas of epoll

To fully understand the nuance behind ***epoll***, it's important to understand how ***file descriptors*** really work. This was explored in my previous post, but it's worth restating again.

A process references I/O streams with the help of ***descriptors.*** Every process maintains a table of file descriptors which it has access to. Every entry in this table has two fields:

— flags controlling the operation of the file descriptor (the only such flag is the *close on exec* flag)
— a pointer to an underlying kernel data structure we'll explore in a bit

Descriptors are either created explicitly by system calls like *open, pipe, socket* and so forth or are ***inherited from the parent process*** during a *fork*. Descriptors are also "duplicated" with a *dup/dup2* system call.

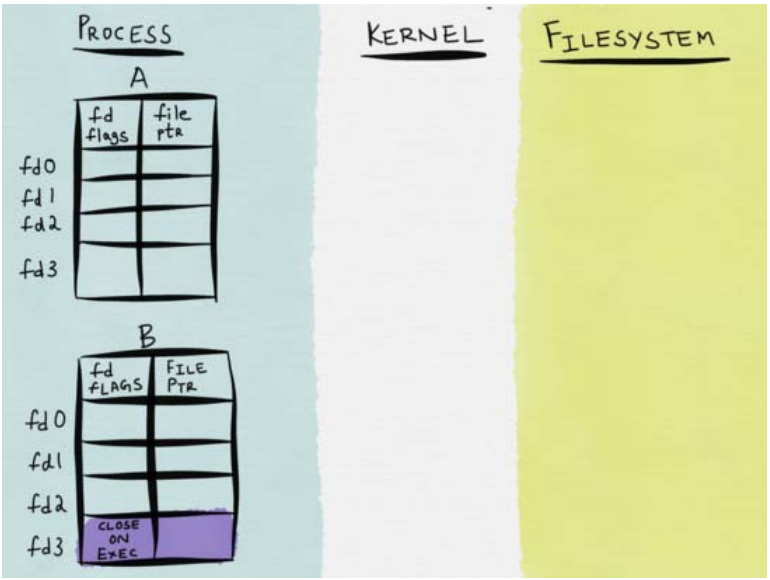


Descriptors are released when:

— the process exits
— by calling the ***close*** system call
— 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. The parent can still continue using the descriptor but the child wouldn't be able to use it once it has **exec**-*ed.*

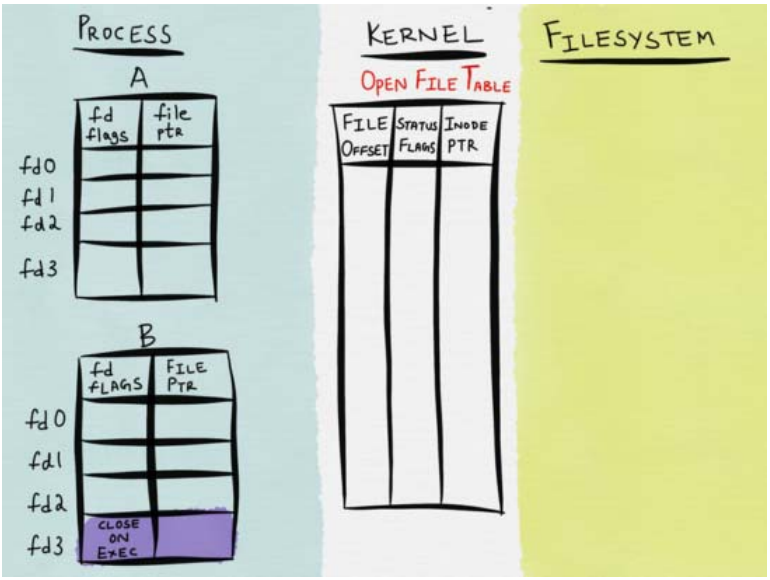
Let us assume in the above example process A has descriptor 3 marked with the ***close-on-exec*** flag. If process A forks process B, then immediately after the fork, process A and process B are identical, and as such process B will have "access" to file descriptors 0, 1, 2 and 3.

But since descriptor 3 is marked as *close-on-exec,* before process B **execs**, this descriptor will be marked as "inactive", and process B won't be able to access it anymore.



To really understand what this means, it becomes important to understand that a descriptor really is just a *per process* pointer to an underlying kernel data structure called (confusingly) the ***file description***.

The kernel maintains a table of all open ***file descriptions*** called the ***open file table***.

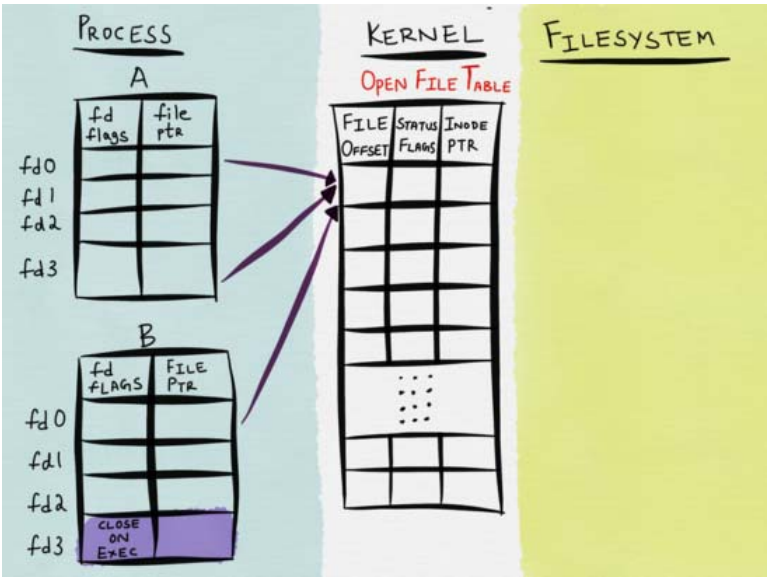


Let's assume ***fd3*** of process A was created as a result of a *dup* or an *fcntl* system call on descriptor ***fd0***. Both the original descriptor ***fd0*** and the "*duplicated*" descriptor ***fd3*** point to the same ***file description*** in the kernel.

If process A then forks process B and ***fd3*** is marked with the ***close-on-exec*** flag, then the child process B will inherit all of the parent process A's

descriptors *but cannot use* ***fd3***.

It's important to note that ***fd0*** in the child process B will also ***point to the same open file description*** in the kernel's open file table.



We have three descriptors — ***fd0*** and ***fd3*** in Process A and ***fd0*** in Process B — that all point to the same underlying kernel open file description. Hold this thought, because this has some important implications for *epoll*. All other file descriptors in both processes A and B also point to an entry in the open file table, but have been omitted from the diagram.

Note - File descriptions aren't just shared by two processes when one forks the other. If one process passed a file descriptor to another process over a Unix Domain Socket socket, then the descriptors of both processes again point to the same underlying kernel open file description.

Finally, it becomes important to understand what the *inode pointer* field of a *file description* is. But prior to that, it's important to understand what an *inode* is.

An inode is file system data structure that contains information about a filesystem object like a file or a directory. This information includes:

— the location of the *blocks* on disk where the file or directory data is stored
— the *attributes* of the file or directory
— additional *metadata* about the file or directory, such as access time, owner, permissions and so forth.

Every file (and directory) in the file system has an ***inode*** entry, which is a number that refers to the file. This number is also called the ***inode number***. On many file systems, the maximum number of *inodes* is capped to a certain

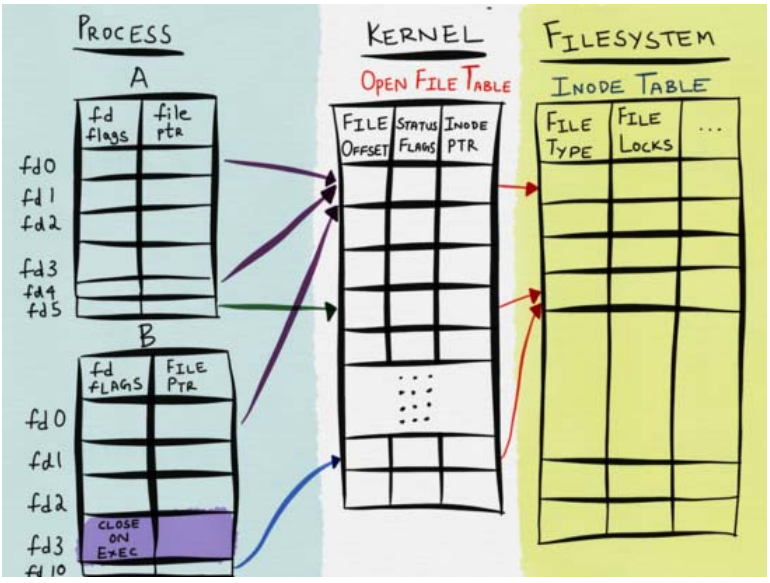
value, meaning the total number of files that can be stored on the system is capped too.

There's an ***inode table*** entry on disk that maintains a map of the ***inode number*** to the actual *inode* data structure on disk. Most file systems are accessed via the kernel's file system driver. This driver uses the ***inode number*** to access the information stored in the *inode*. Thus in order to know the location of a file or any metadata pertaining to the file, the kernel's file system driver needs to access the ***inode table***.

Let's assume *after* process A forks process B, process A has created two more file descriptors ***fd4*** and ***fd5***. These aren't duplicated in process B.

Let's assume ***fd5*** is created as a result of process A calling `open` on file `abc.txt` for *reading*. Let us assume process B also calls `open` on `abc.txt` but for *writing* and the file descriptor the `open` call returns to process B is ***fd10***.

Then process A's ***fd5*** and process B's ***fd10*** point to different open file descriptions in the open file table, *but they point to the same* ***inode table*** *entry* (or in other words, the same file).



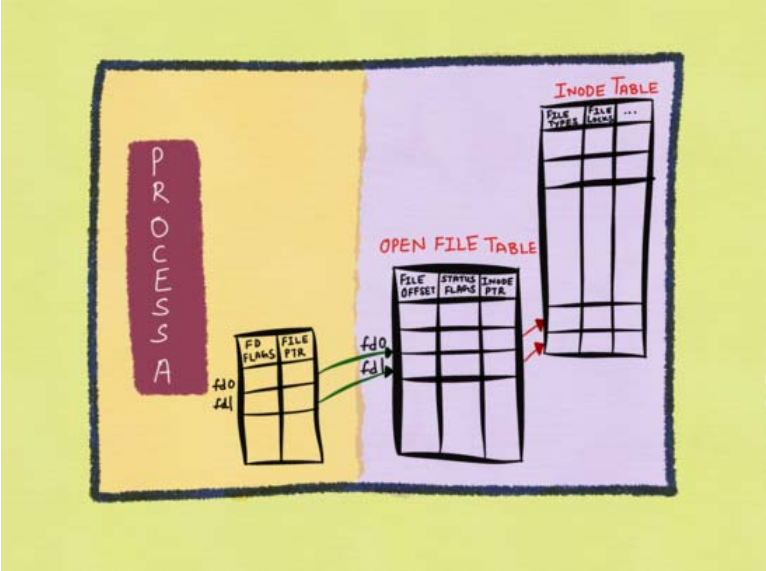
This has two very important implications:

— Since ***fd0*** in both process A and process B refer to the same open file description, they share the ***file offset***. This means that if process A advances the file offset (by calling `read()` or `write()` or `lseek()`), then the offset changes for process B as well. This is also applicable to ***fd3*** belonging to process A, since ***fd3*** refers to the same open file description as ***fd0***.
— This is also applicable to modifications made by a file descriptor in one process to an open file status flag (O_ASYNC, O_NONBLOCK, O_APPEND). So if process B sets ***fd0*** to the non blocking mode by setting

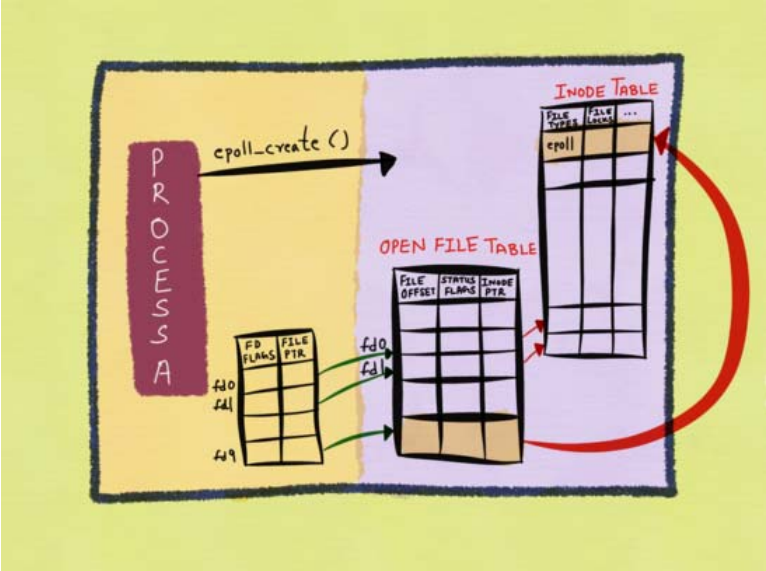
the `O_NONBLOCK` flag via the `fcntl` system call, then descriptors ***fd0*** and ***fd3*** belonging to process A will also start observing non-blocking behavior.

The bowels of epoll

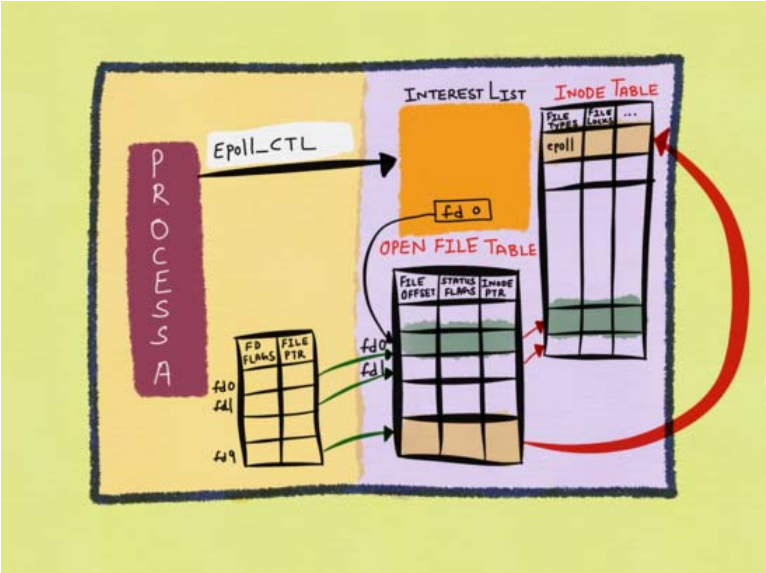
Let us assume a process A has two open file descriptors ***fd0*** and ***fd1***, that have two open *file descriptions* in the open file table. Let is assume both these file descriptions point to different *inodes*.



`epoll_create` creates a new *inode entry* (the *epoll* instance) as well as an open *file description* for it in the kernel, and returns to the calling process a file descriptor (***fd9***) to this open file description.



When we use `epoll_ctl` to add a file descriptor (say ***fd0***) to the epoll instance's *interest list*, we're ***actually fd0's underlying file description*** to the *epoll* instance's interest list.

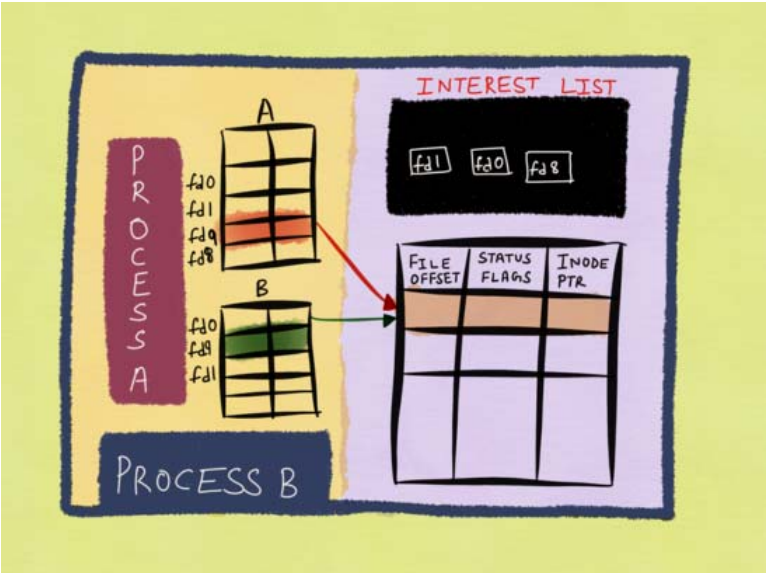


Thus the *epoll* instance actually monitors the *underlying file description*, and ***not the per process file descriptor***. This has some interesting implications.

— If process A forks a child process B, then B inherits all of A's descriptors, including ***fd9***, the epoll descriptor. However, process B's descriptors ***fd0***, ***fd1*** and ***fd9*** still refer to the same underlying kernel data structures. Process B's *epoll* descriptor (***fd9***) *shares the same interest list* with process A.

If after the fork, if process A creates creates a new descriptor **fd8** (non-duplicated in process B) to its *epoll* interest list via `epoll_ctl`, then it's not just process A that gets notifications about events on **fd8** when calling `epoll_wait()`

If process B calls `epoll_wait()`, then process B gets the notification about **fd8** (which belongs to process A and wasn't duplicated during the fork) as well. This is also applicable when the *epoll* file descriptor is duplicated by means if a call to `dup/dup2` or if the *epoll* file descriptor is passed to another process over a Unix Domain Socket.



Let's assume process B opens the file pointed to by ***fd8*** with a new `open` call, and gets a new file descriptor (***fd15***) as a result. Let's now assume process A closes ***fd8***. One would assume that since process A has closed ***fd8***, it will no longer get notifications about events on ***fd8*** when calling `epoll_wait`. This, however, isn't the case, since the interest list monitors the *open file description.* Since ***fd15*** points to the same description as ***fd8*** (since they are both the same underlying file), process A gets notifications about events on ***fd15***. It's safe to say that once a file descriptor has been registered by a process with the epoll instance, then the process will continue getting notifications about events on the descriptor even if it closes the descriptor, so long as the underlying open file description is still referenced by at least one other descriptor (belonging to the same or a different process).

Why epoll is more performant that select and poll

As stated in the previous post, 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, the kernel still needs to scan every descriptor in the list.

Since epoll *monitors the underlying file description*, every time the open file description becomes ready for I/O, the kernel adds it to the ready list without waiting for a process to call `epoll_wait` to do this. ***When*** a process does call `epoll_wait`, then at that time the kernel doesn't have to do any additional work to respond to the call, but instead returns all the information about the *ready list* it's been maintaining all along.

Furthermore, with every call to *select/poll* requires passing the kernel the information about the descriptors we want to monitored. This is obvious from the signature to both calls. The kernel returns the information about all the file descriptors passed in which the process again needs to examine (by scanning all the descriptors) to find out which ones are *ready* for I/O.

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

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

With *epoll*, once we add the file descriptors to the epoll instance's ***interest list*** using the `epoll_ctl` call, then when we call `epoll_wait` in the future, we don't need to subsequently pass the file descriptors whose *readiness* information we wish to find out. The kernel again only returns back information about those descriptors which are ready for I/O, as opposed to

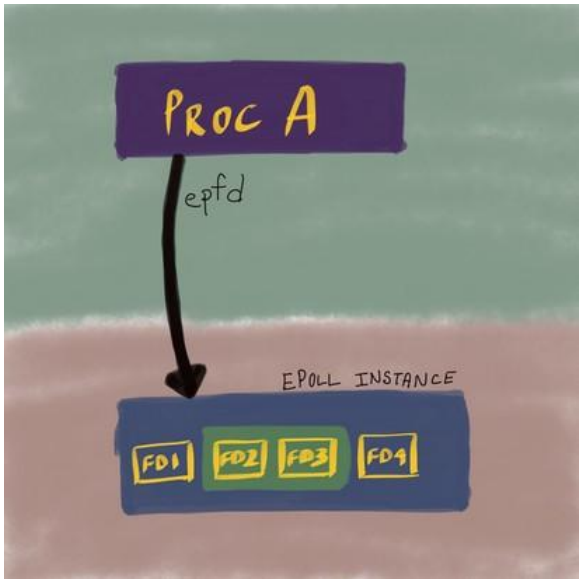
the *select/poll* model where the kernel returns information about ***every descriptor passed in***.

As a result, the cost of epoll is **O(number of events that have occurred)** and not **O(number of descriptors being monitored)** as was the case with *select/poll*.

Edge triggered epoll

By default, epoll provides *level-triggered* notifications. Every call to `epoll_wait` only returns the subset of file descriptors belonging to the interest list that are *ready*.

So if we have four file descriptors (***fd1***, ***fd2***, ***fd3*** and ***fd4***) registered, and only two (***fd2*** and ***fd3***) are *ready* at the time of calling `epoll_wait`, then only information about these two descriptors are returned.



It's also interesting to note that in the default *level-triggered* case, the nature of the descriptors (*blocking* versus *non-blocking*) in *epoll's* interest won't really affect the result of an `epoll_wait` call, since *epoll* only ever updates its *ready list* when the underlying open file description becomes *ready*.

Sometimes we might just want to find the status of *any descriptor* (say ***fd1***, for example) in the interest list, irrespective of whether it's ready or not. *epoll* allows us to find out whether I/O is possible on any particular file descriptor (***even if it's not ready*** at the time of calling `epoll_wait`) by means of supporting ***edge-triggered*** notifications. If we want information about whether there has been any I/O activity on a file descriptor since the previous call to `epoll_wait` (or since the descriptor was opened, if there was no previous `epoll_wait` call made by the process), we can get *edge-*

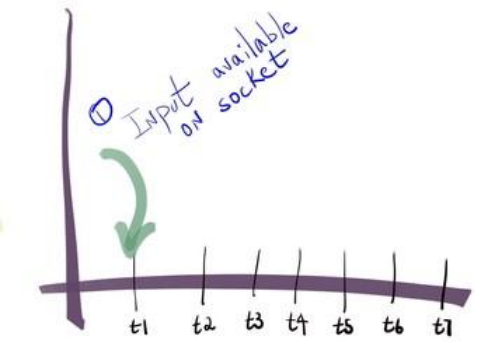
triggered notifications by ORing the **EPOLLET** flag while calling `epoll_ctl` while registering a file descriptor with the *epoll* instance.

Perhaps it becomes more helpful to see this in action in code from a real project where a file descriptor is being registered with an *epoll* instance with `epoll_ctl` where the **EPOLLET** flag is ***OR***ed along with some other flags.

```
function Poller:register(fd, r, w)
        local ev = self.ev[0]
        ev.events = bit.bor(C.EPOLLET, C.EPOLLERR, C.EPOLLHUP)
        if r then
                ev.events = bit.bor(ev.events, C.EPOLLIN)
        end
        if w then
                ev.events = bit.bor(ev.events, C.EPOLLOUT)
        end
        ev.data.u64 = fd
        local rc = C.epoll_ctl(self.fd, C.EPOLL_CTL_ADD, fd, ev)
        if rc < 0 then errors.get(rc):abort() end
end
```

Perhaps an illustrative example can better help understand how edge-triggered notifications work with *epoll*. Let's use the example used previously, where a process has registered four descriptors with the *epoll* instance. Let's assume that **fd3** is a socket.

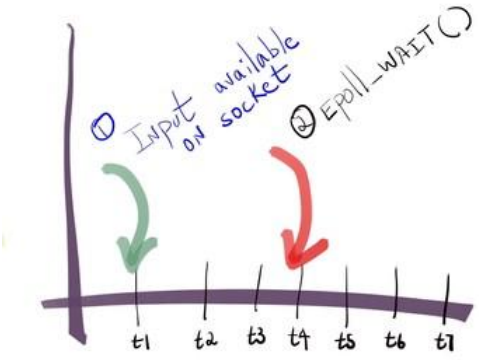
Let's assume that at time **t1**, an input byte stream arrives on the socket referenced by **fd3**.



At **t0**, input arrives on the socket.

Let's now assume that at time ***t4***, the process calls `epoll_wait()` .

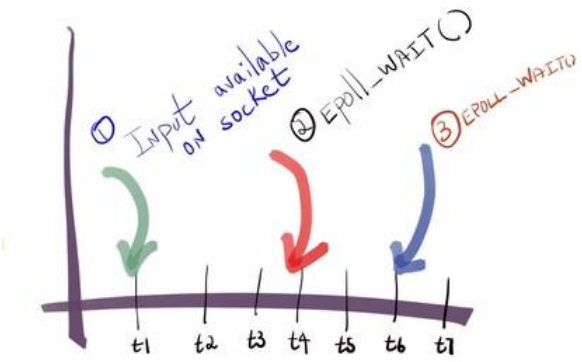
If at time ***t4***, file descriptors **fd2** and **fd3** are ready, then the `epoll_wait` call reports **fd2** and **fd3** as *ready*.



At time **t4**, the process calls epoll_wait

Let's assume that the process calls `epoll_wait` again at time ***t6***. Let's assume **fd1** is ready. Let's also assume that *no* input arrived on the socket referenced by ***fd3*** between times ***t4*** and ***t6***.

In the *level-triggered* case, a call to `epoll_wait` will return ***fd1*** to the process, since ***fd1*** is the only descriptor that is *ready*. However in the *edge-triggered case*, this call will *block*, since no new data has arrived on the socket referenced by ***fd3*** between times **t4** and **t6**.



At time **t6**, the process calls **epoll_wait** again

Conclusion

This post aimed to capture the "method" part. In order to understand the "madness" wrecked by these semantics of *epoll*, a good reference would be the following two blog posts:

epoll is fundamentally broken — parts 1 and 2.