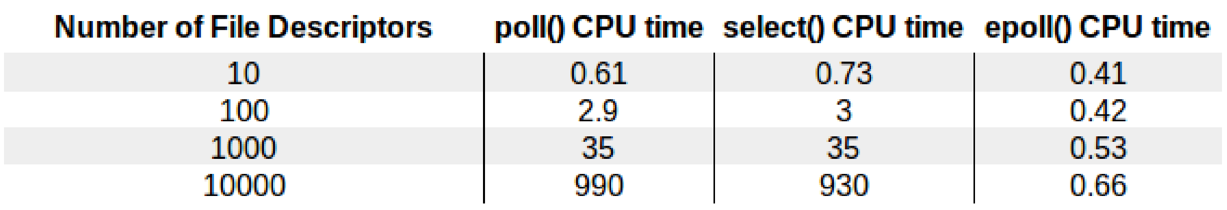
Comparison of Various Asynchronous I/O Calls

- Mainly used for webservers.
- The table shows the time it takes to check different quantities of file descriptors via some of the most common polling methods.



Number of File Descriptors	poll() CPU time	select() CPU time	epoll() CPU time
10	0.61	0.73	0.41
100	2.9	3	0.42
1000	35	35	0.53
10000	990	930	0.66

- As this shows, the performance benefits of epoll are decent enough to have an impact on even as few as 10 descriptors. As the number of descriptors increases, using regular poll() or select() becomes a very unattractive option compared to epoll().

Level triggered and edge triggered event notifications in epoll()

- Level-triggered and edge-triggered are terms borrowed from electrical engineering. When we’re using epoll the difference is important.
- In edge triggered mode we will only receive events when the state of the watched file descriptors change.
- In level triggered mode we will continue to receive events until the underlying file descriptor is no longer in a ready state.
- Generally speaking level triggered is the default and is easier to use.

Data Behaviour for epoll() Level Triggered and Edge Triggered

- If we give a longer input than the read buffer capacity while using epoll, we can see the effect of both this triggering modes.
- If the given input was too long for the read buffer, that is where level triggering is helpful. The events continued to populate until it read all of what was left in the buffer, in edge triggering mode we would have only received 1 notification and the application as-is would not progress until more was written to the file descriptor being watching.

Why is epoll() faster than select() and poll()

- An epoll file descriptor has a private struct eventpoll that keeps track of which fd's are attached to this fd. struct eventpoll also has a wait queue that keeps track of all processes that are currently epoll_waiting on this fd. struct epoll also has a list of all file descriptors that are currently available for reading or writing.
- When you add a file descriptor to an epoll fd using epoll_ctl(), epoll adds the struct eventpoll to that fd's wait queue. It also checks if the fd is currently ready for processing and adds it to the ready list, if so.
- When you wait on an epoll fd using epoll_wait, the kernel first checks the ready list, and returns immediately if any file descriptors are already ready. If not, it adds itself to the single wait queue inside struct eventpoll, and goes to sleep.
- When an event occurs on a socket that is being epoll()ed, it calls the epoll callback, which adds the file descriptor to the ready list, and also wakes up any waiters that are currently waiting on that struct eventpoll.
- Obviously, a lot of careful locking is needed on struct eventpoll and the various lists and wait queues.

Example Scenario

- A typical server might be dealing with, say, 200 connections. It will service every connection that needs to have data written or read and then it will need to wait until there's more work to do. While it's waiting, it needs to be interrupted if data is received on any of those 200 connections.
- With select, the kernel has to add the process to 200 wait lists, one for each connection. To do this, it needs a "thunk" to attach the process to the wait list. When the process finally does wake up, it needs to be removed from all 200 wait lists and all those thunks need to be freed.
- By contrast, with epoll, the epoll socket itself has a wait list. The process needs to be put on only that one wait list using only one thunk. When the process wakes up, it needs to be removed from only one wait list and only one thunk needs to be freed.
- To be clear, with epoll, the epoll socket itself has to be attached to each of those 200 connections. But this is done once, for each connection, when it is accepted in the first place. And this is torn down once, for each connection, when it is removed. By contrast, each call to select that blocks must add the process to every wait queue for every socket being monitored.
- Ironically, with select, the largest cost comes from checking if sockets that have had no activity have had any activity. With epoll, there is no need to check sockets that have had no activity because if they did have activity, they would have informed the epoll socket when that activity happened. In a sense, select polls each socket each time you call select to see if there's any activity while epoll rigs it so that the socket activity itself notifies the process.

References

1. https://suchprogramming.com/epoll-in-3-easy-steps/

2. https://stackoverflow.com/questions/17355593/why-is-epoll-faster-than-select?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa

3. https://jvns.ca/blog/2017/06/03/async-io-on-linux--select--poll--and-epoll/

4. Linux Programming Interface by Michael Kerrisk