Boost I/O Strategy In Web Server - Dive into Libev and Libeio

Published: December 01, 2018 by Zhenghui Lee

Categories: io-strategy 2

Tags: I/O 2 c/c++ 3 event-loop 1 linux 3 perf 2

This article is **Part 2** in a **2-Part** Series.

- Part 1 - Boost I/O Strategy In Web Server - Motivation
- Part 2 - This Article

Table of Contents

It has been a little bit long time since my last post for this series. My apologize for the late update.

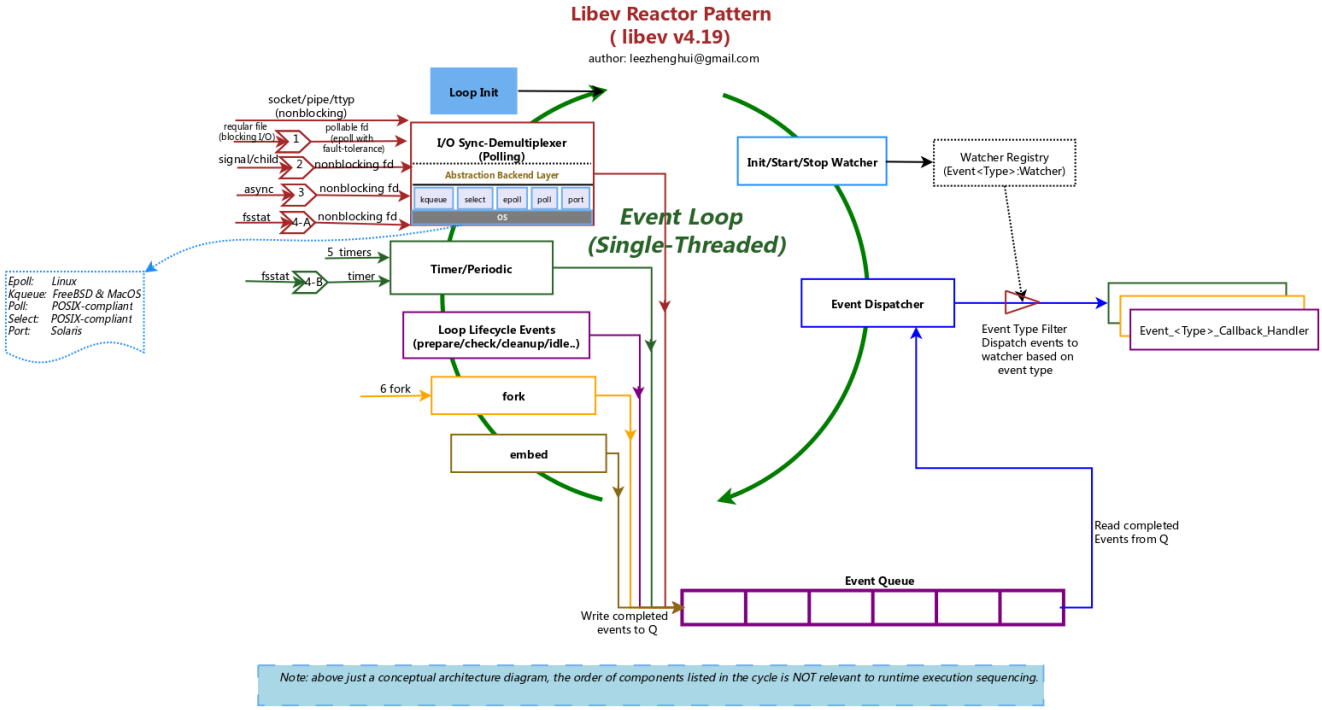
Following my original plan, the 2nd post in this series should be an introduction of I/O models followed by recap of c10k problem, and after that we will explore some of well-used frameworks/products based on different programming languages and take a closer look at the details from implementation perspective. Recently, I was invited to give a presentation on the topic of libev based event-loop in our team, the survey result seems many programers know about c10k and I/O models. This change my mind on the posts order, so we see this topic comes here. The content in this post is more suitable to the audiences who already have the backgroud knowledge of Linux system I/O models(Blocking, Non-Blocking, Sync and Async), reactor pattern and know about libev and libeio. If you have system programming experiences, that would be better.

Libev

libev - a high performance full-featured event-loop written in C.

> From man
>
> Libev supports "select", "poll", the Linux-specific "epoll", the BSD-specific "kqueue" and the Solaris-specific event port mechanisms for file descriptor events ("ev_io"), the Linux "inotify" interface (for "ev_stat"), Linux eventfd/signalfd (for faster and cleaner inter-thread wakeup ("ev_async")/signal handling ("ev_signal")) relative timers ("ev_timer"), absolute timers with customised rescheduling ("ev_periodic"), synchronous signals ("ev_signal"), process status change events ("ev_child"), and event watchers dealing with the event-loop mechanism itself ("ev_idle", "ev_embed", "ev_prepare" and "ev_check" watchers) as well as file watchers ("ev_stat") and even limited support for fork events ("ev_fork").

Components architecture



- convertor

  The convertor in above diagram is used to represent the mechanisms which transform a non-pollable resource to a pollable operation fashion, such that be able to adopte to the event-loop processing. The non-pollable resoruces would be:

- The target file fd is not supported by the underlaying demultiplexer(in libev code,we call it `backend`), e.g: the regular disk file is not supported by epoll in linux.

> 
>
> For the fd of regular disk file, it can't work with epoll, a EPERM error will returned if you do that. BTW, poll and select based backend can support disk file fd, but the read/write operation actually running in a blocking I/O mode, that is definitely we need to be aware of, and in a single thread event-loop application, we need to avoid this kind of usage, as it may potentially introduce "world-stop" due to the I/O blocking operation(espcially for a disk file read call, I guess write operation may be better than read operation in this case, as it just put the data to memory buffer and return), and slow down the whole iteration of event processing.

- The resources run in an ultimate async manner, adopt to the pollable mode. e.g: signal.

- 

For regular disk file fd, which does not work in a real non-blocking manner, it is not supported by epoll directly. To unify the programming interface shape for all of fd, libev make some additional efforts to adopt it on epoll based demultiplexer, including provide fault-tolerance logic for EPERM" error and handle these I/O fds in a separate data model and put them to event-loop queue.(please refer to code line 120 and line 222 for more details). Please keep in mind, run disk file fd in this way will get a poor performance. AFAIK, the best practice on blocking I/O operation is, convert it to an async fashion operation(either via multi-thread or signal) and adopt to event-loop processing via a `eventfd` or `pipe` file.

- 

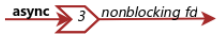
There are two kinds of underlying techologies can be used in the convertor to adopt signal to the event-loop processing.

- [A] signalfd
- [B] self-pipe pattern, using either an eventfd or built-in pipe.

With self-pipe pattern, one side of pipe is written by by an async resource handler, an other side is polled by the backend in the loop.

- 

  Unix signals child process termination with `SIGCHLD`, the ev_child actually initail a ev_signal and listen on SIGCHLD, generate/consume child event accordingly. In libev implementation (L2466, childcb method), try to avoid in-lined loop in the callback method, instead puting a sig_child event to event pending queue to make sure the childcb are called again in the same tick until all children have been reaped. Refer to the workflow for the details in later of this post.

- 

  ev_async provide a generic way to enable an naturally async source(e.g: the source is triggerred by signal or running in an other thread) adopt to the event_loop processing. Under the hood, the convertor actually leverage self-pipe pattern(either eventfd or built-in pipe) to achieve the transformation.

- 

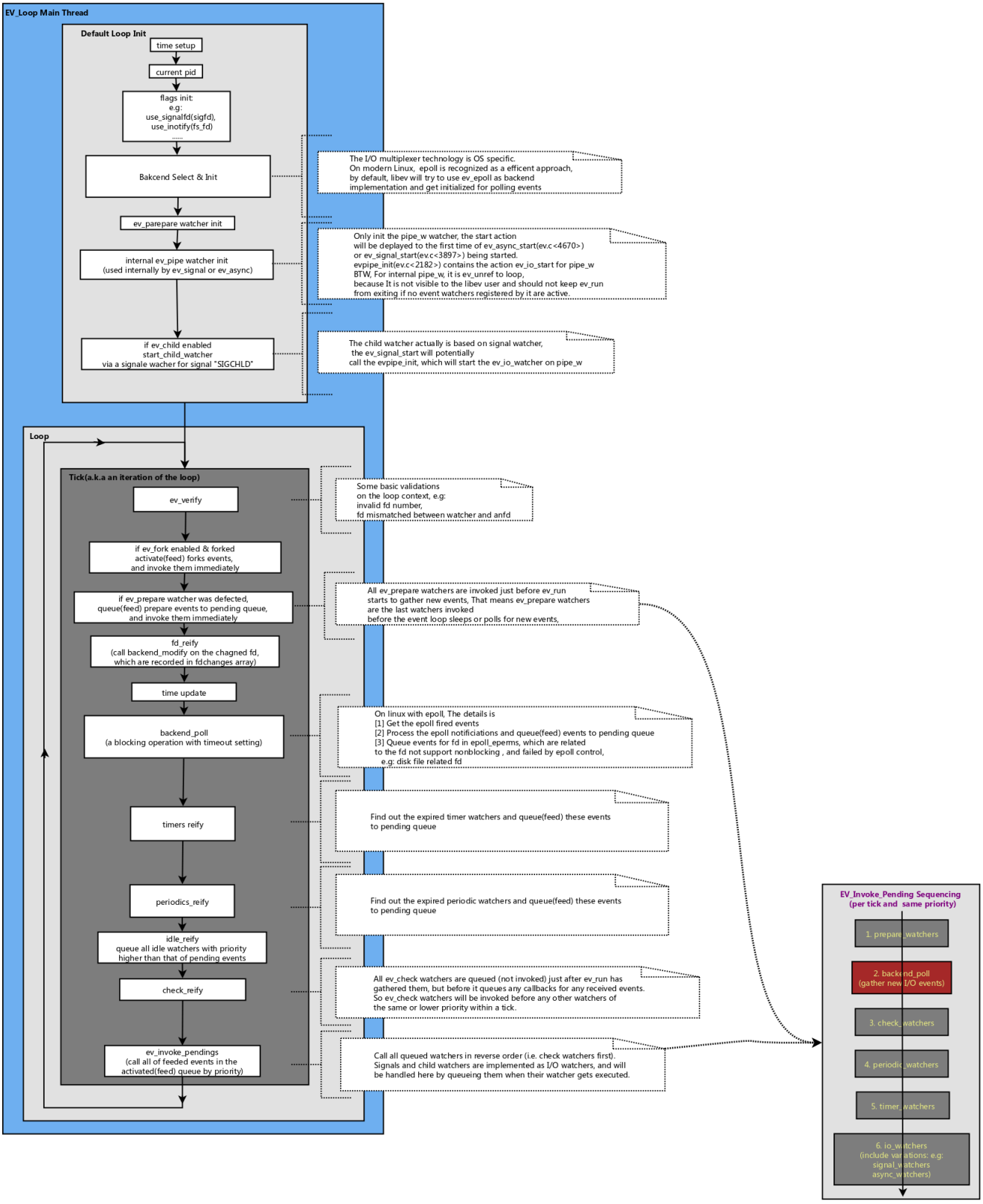
  There two approaches to get stat of a file:

  - [A] inotify, it is suitable to local file system, and linux kernel version >= 2.6.25
  - [B] timer based, it is used by remote file system or linux kernel version < 2.6.25

Framework Workflow

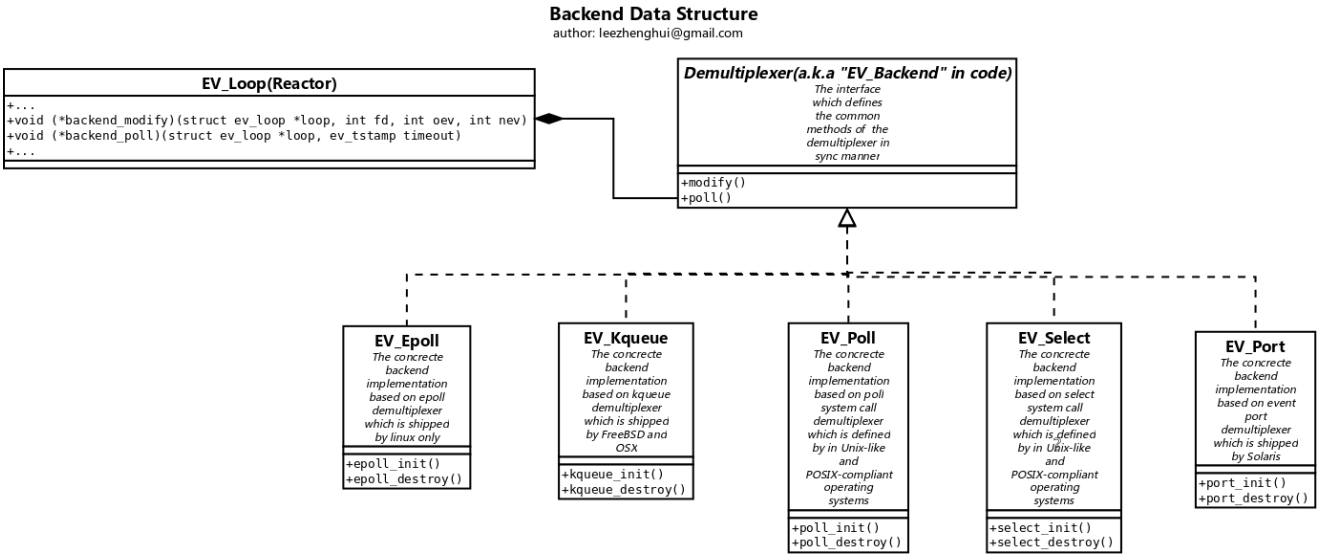
Workflow of Libev Bootstrap and Loop handling

author: leezhenghui@gmail.com



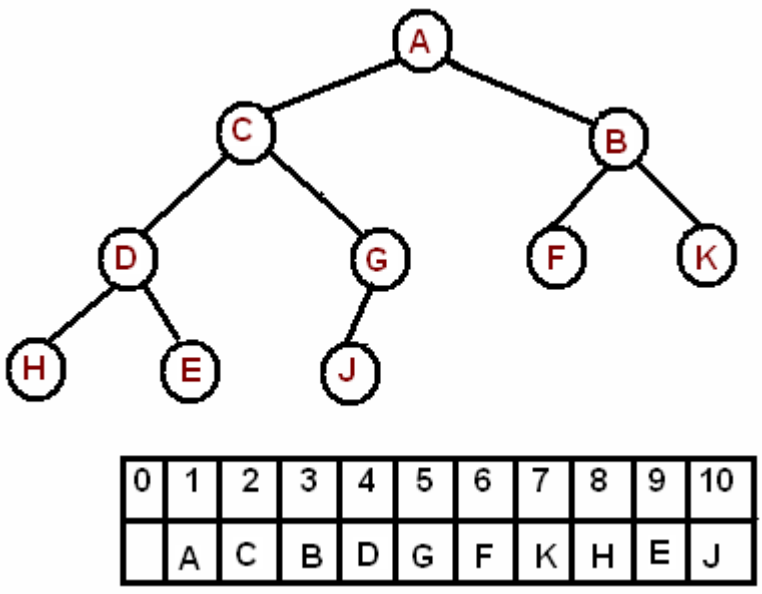
Data sturcture

Backend

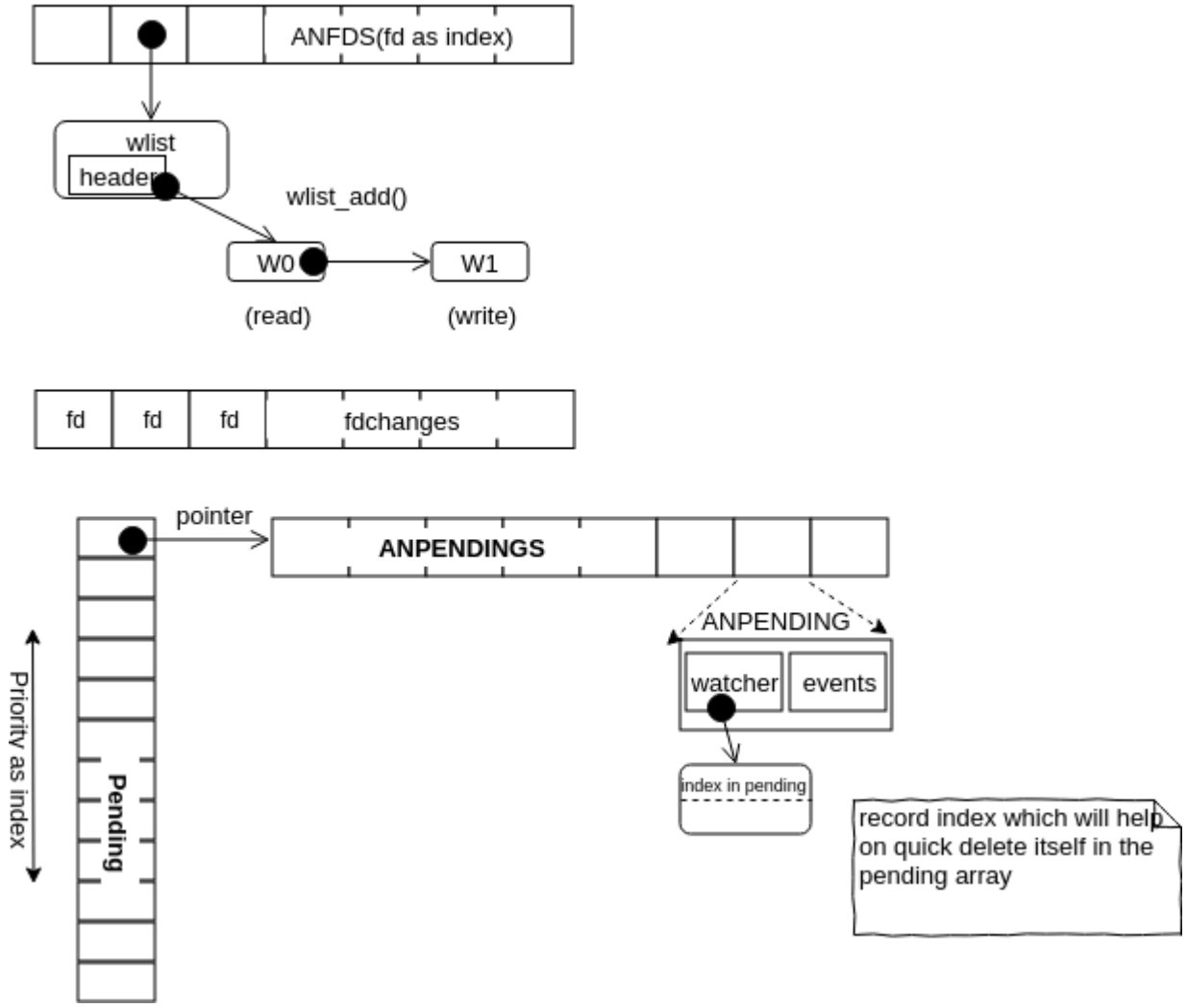


Timer

Libev is using heap data structure to achieve the efficient query/store on ordered elements.



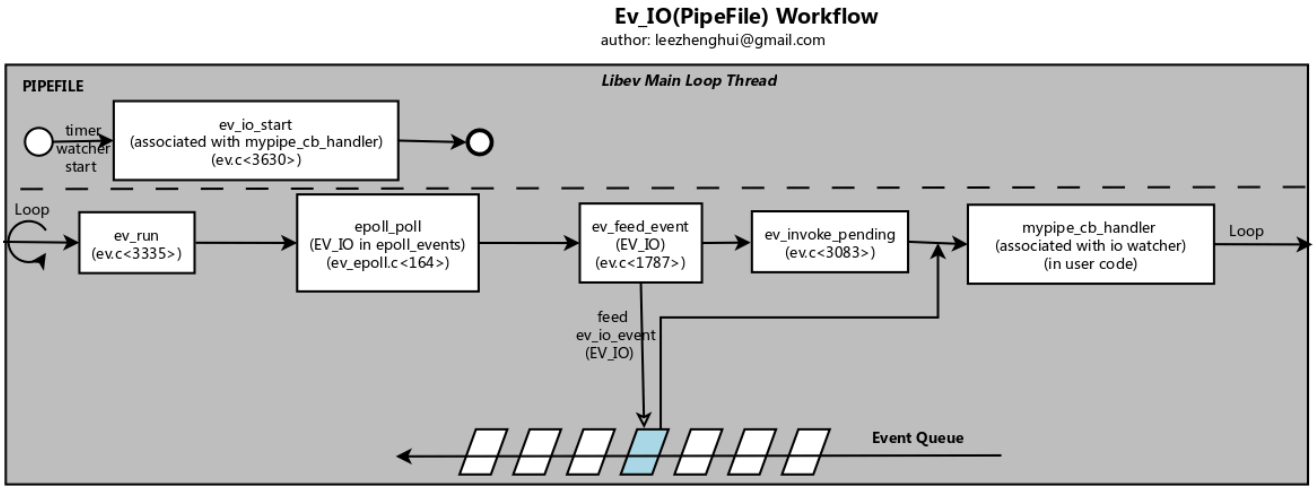
Watcher and Pending Queue



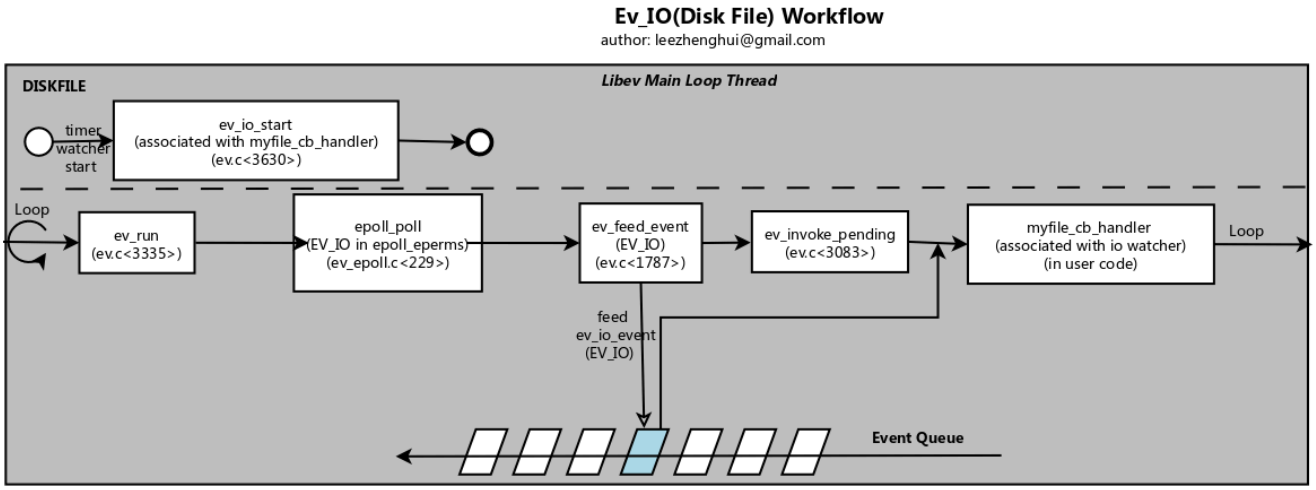
Event Workflow

EV_IO

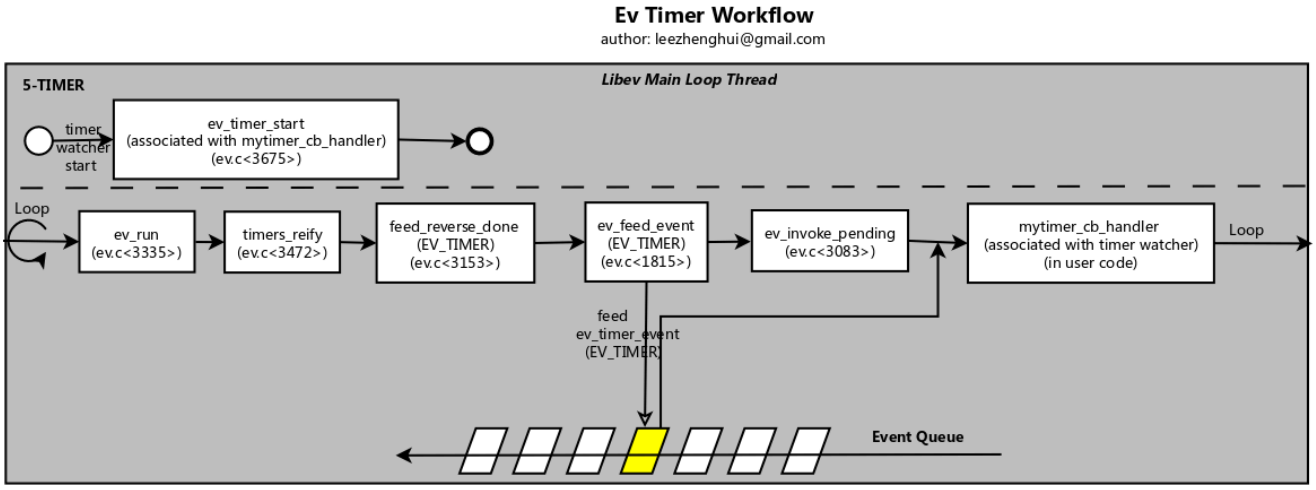
- Non-Blocking I/O



- Blocking I/O

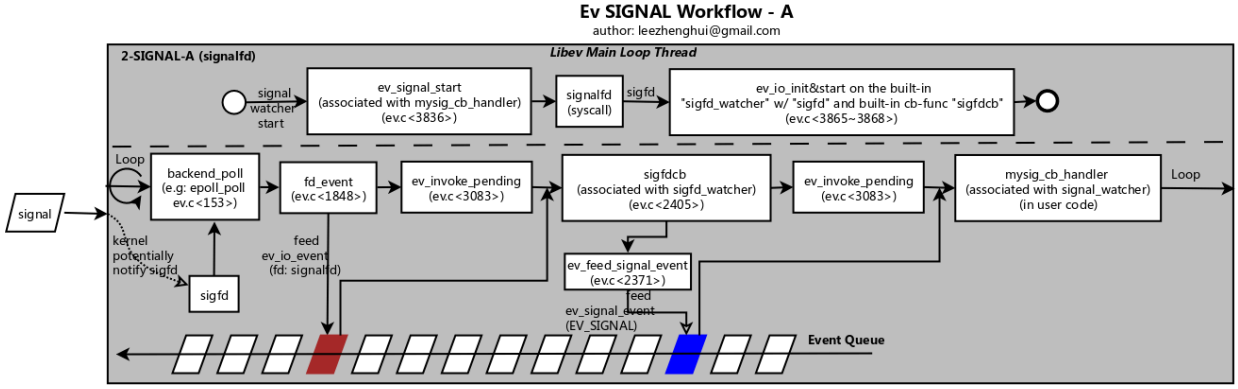


EV_Timer

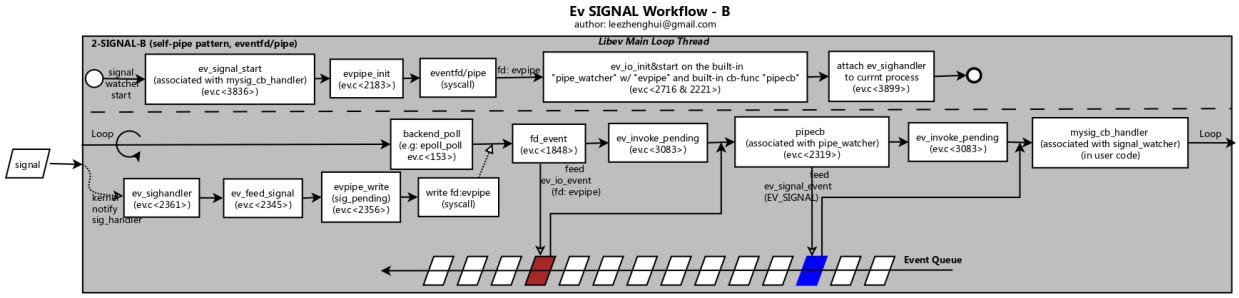


EV_SIGNAL

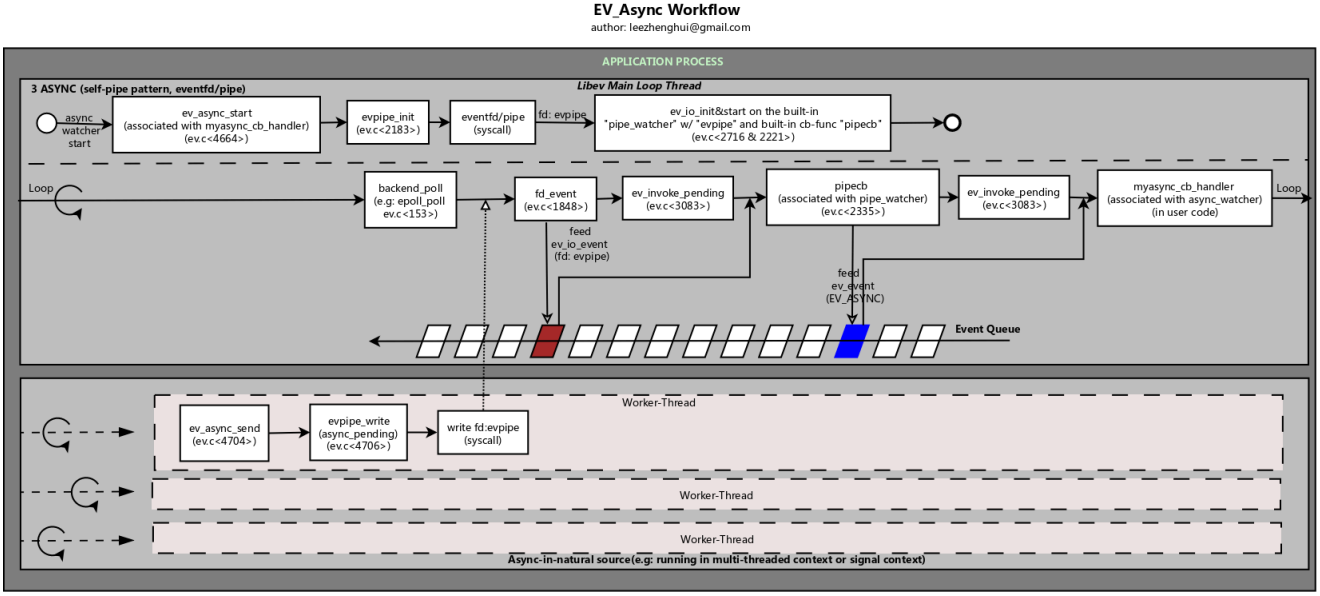
- Approach-A(signalfd)



- Approach-B(built-in pipe pattern)

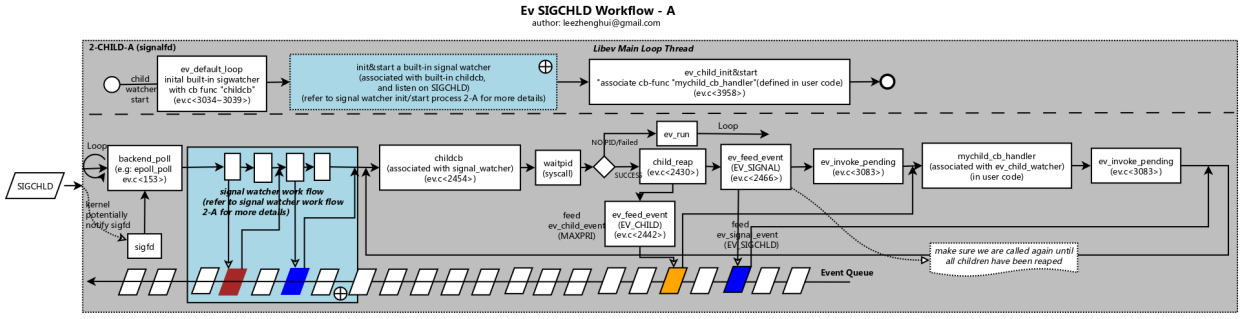


EV_Async

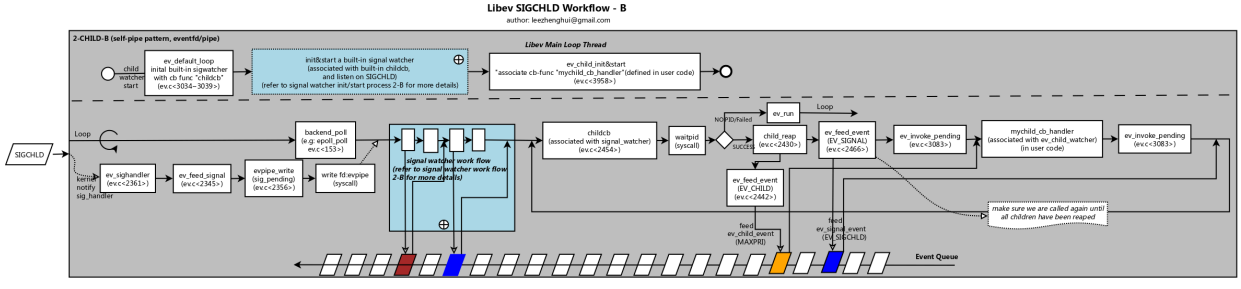


EV_CHILD

- Approach-A(signalfd)

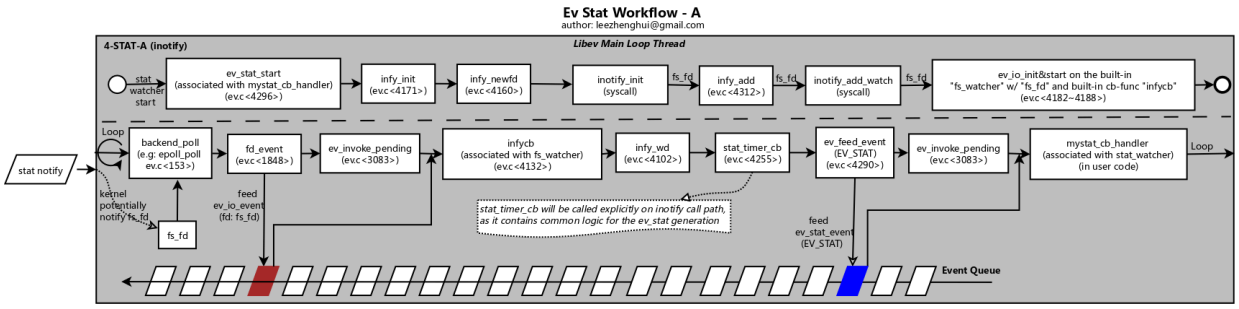


- Approach-B(built-in pipe pattern)

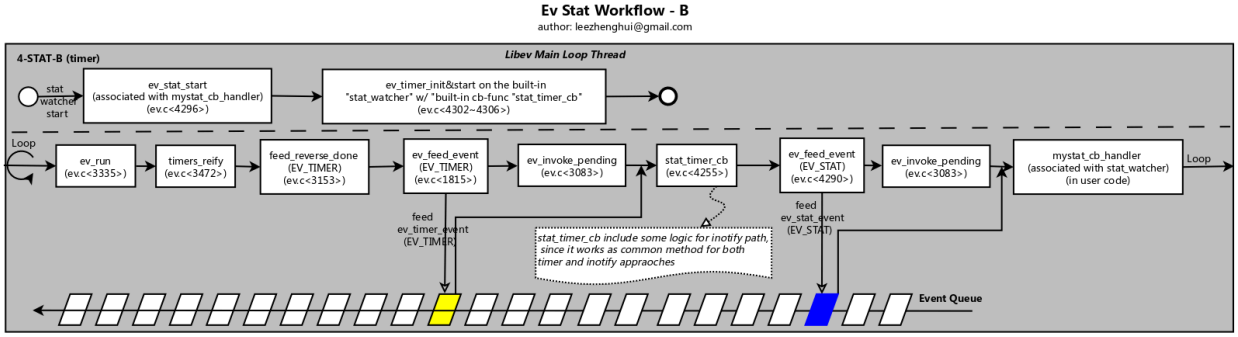


EV_Stat

- Approach-A(inotify)



- Approach-B(ride on timer)

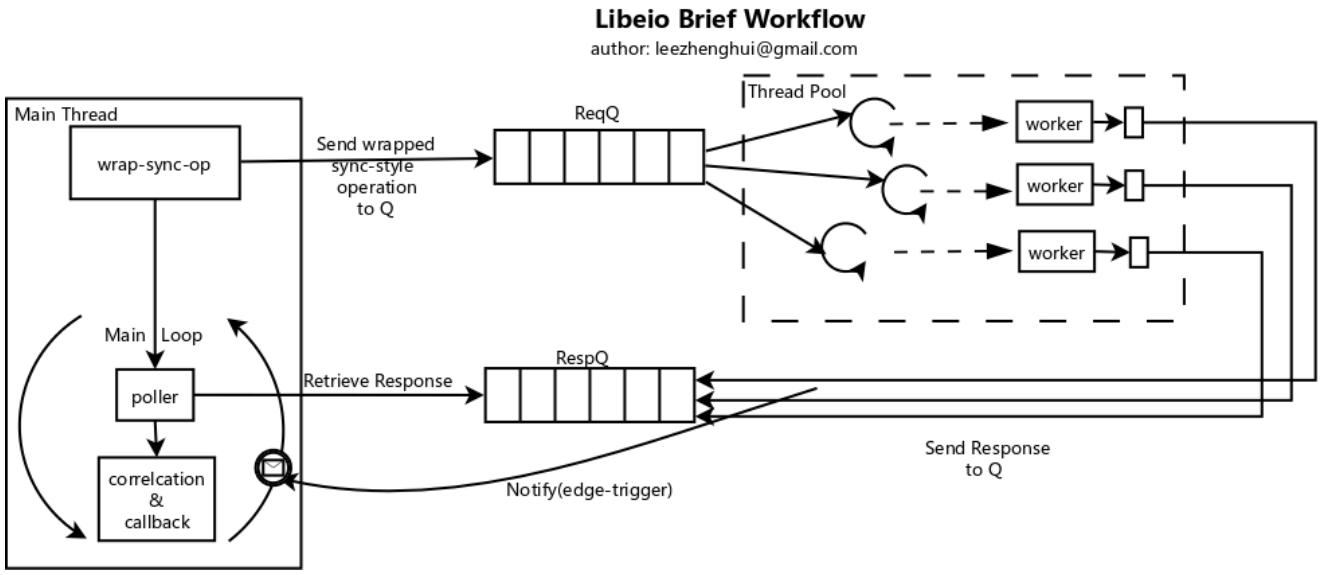


Libeio

Libeio is event-based fully asynchronous I/O library for C

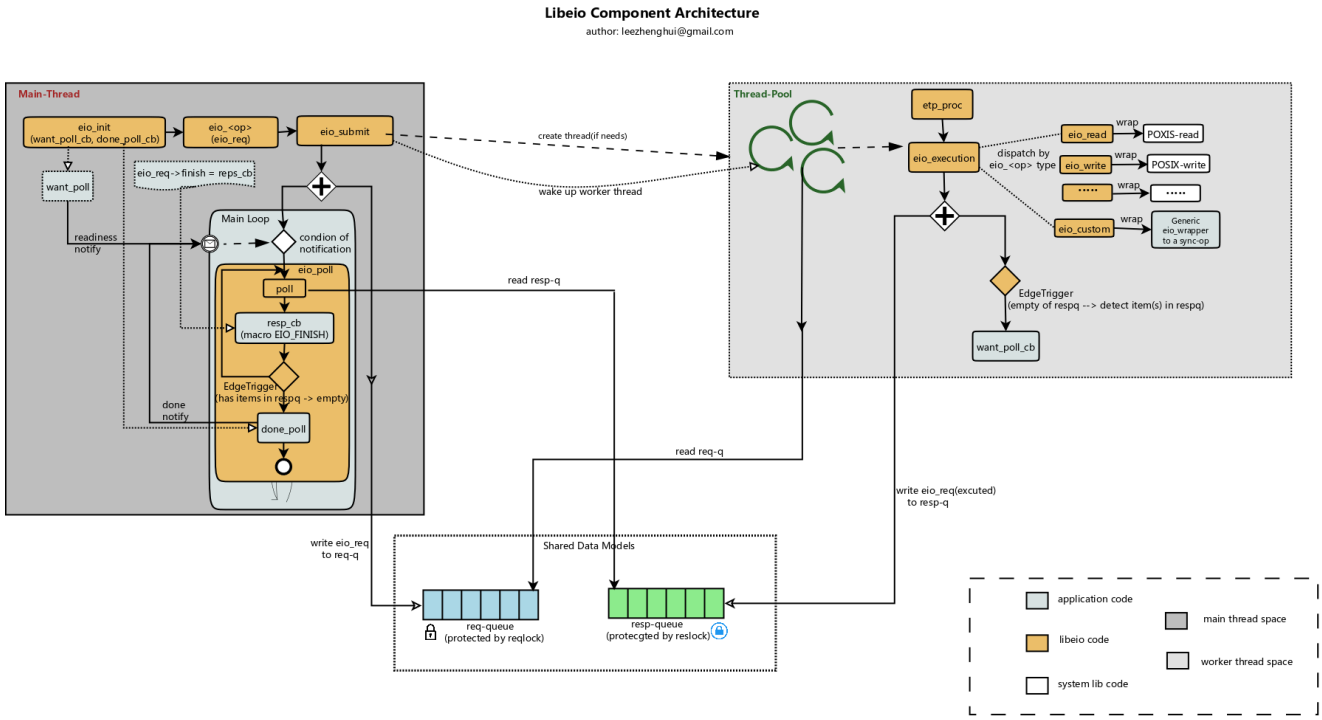
> From libeio website
>
> Libeio is a full-featured asynchronous I/O library for C, modelled in similar style and spirit as libev. Features include: asynchronous read, write, open, close, stat, unlink, fdatasync, mknod, readdir etc. (basically the full POSIX API). sendfile (native on solaris, linux, hp-ux, freebsd, emulated everywehere else), readahead (emulated where not available).

Breif workflow



Briefly, libeio will wrap a sync-style operation and leverage userland thread pool to simulate an async invocation manner.

Components Architecture



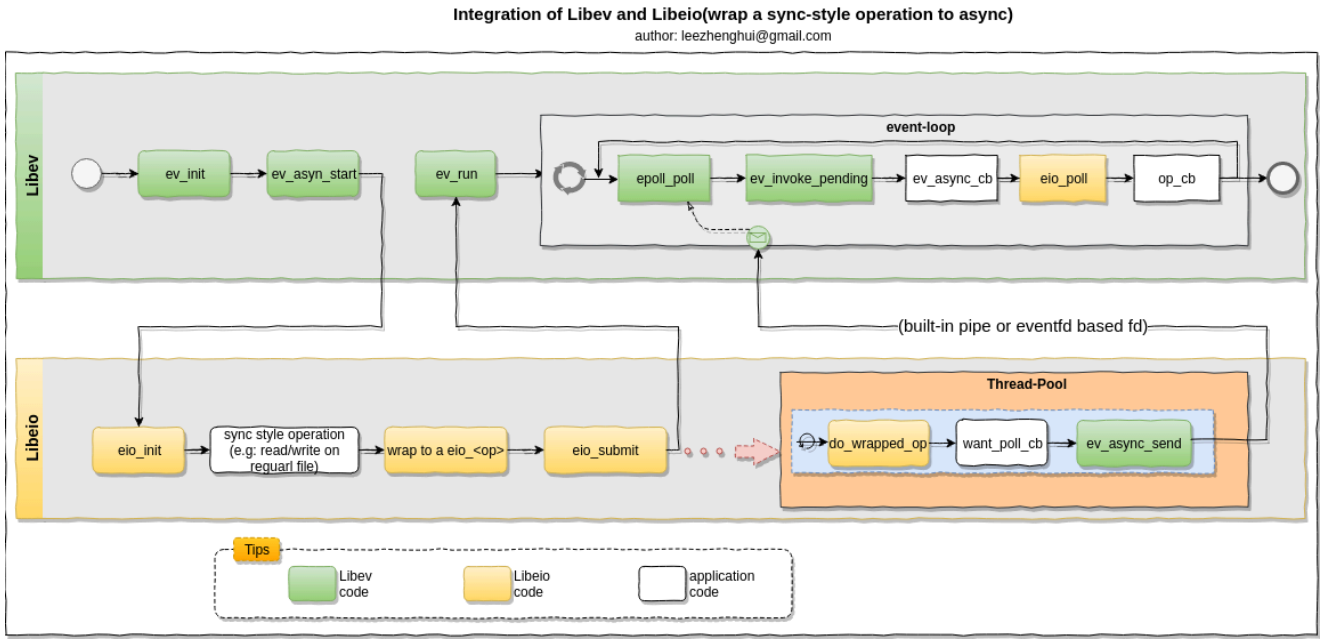
POSIX.AIO

POSIX.AIO provides an asynchronous I/O interface implemented as part of glibc. It achieves this via using userland thread pool internally. But seems it is not good enough. If using callback function as completion notifiction, POSIX.AIO will spawn a new thread to do the callback. Imaging in a high

concurrency case, each callback function contains a long running logic, this design will draw us back to request-per-thread mode. On the contrary, Libeio is implemented in the polling natural design. As you can see above, the callback function is executed in the original thread, instead of the thread spwaned by libeio, in such way, libeio can fully control the thread pool size and lifecycle.

BTW, besides callback notification, POSIX.AIO also supports signal notificaiton, but that also have some problems, if you are interested in this topic, please refer to link for more details.

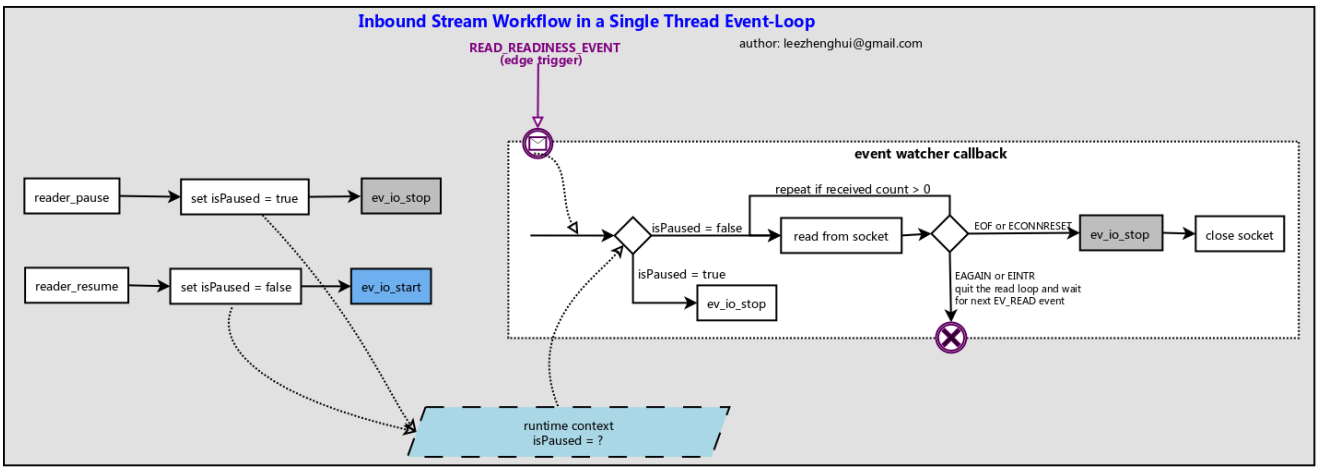
Integration of Libev and Libeio



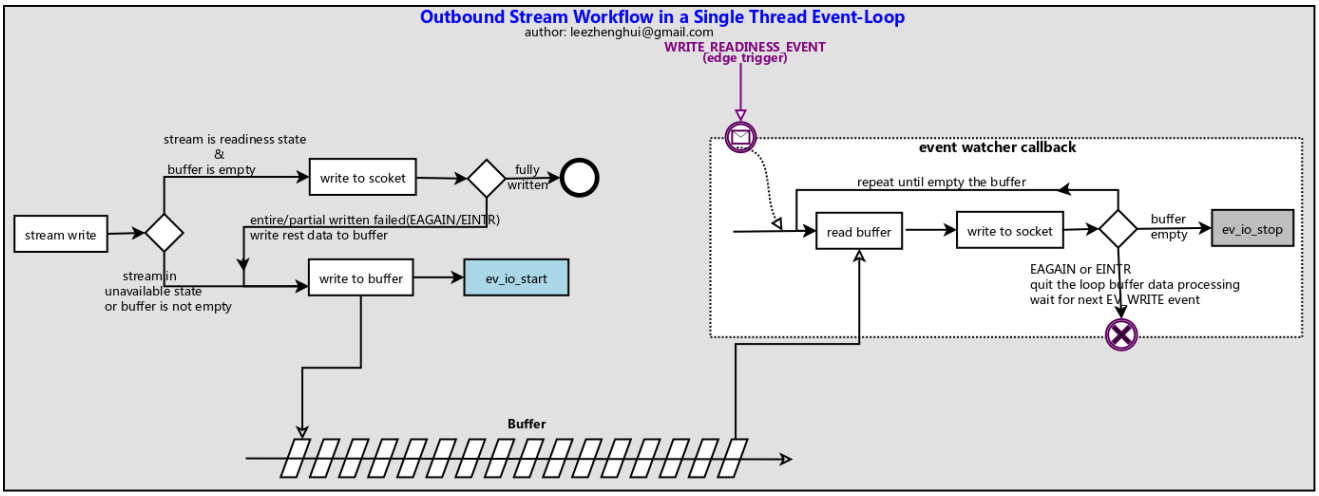
High level design of network stream abstraction layer based on libev

> Notable, Libev is running on epoll backend in level-triggered mode. But the workflows described below actually are intented to adopt to epoll technology with edge-triggered mode.

Inbound



Outbound

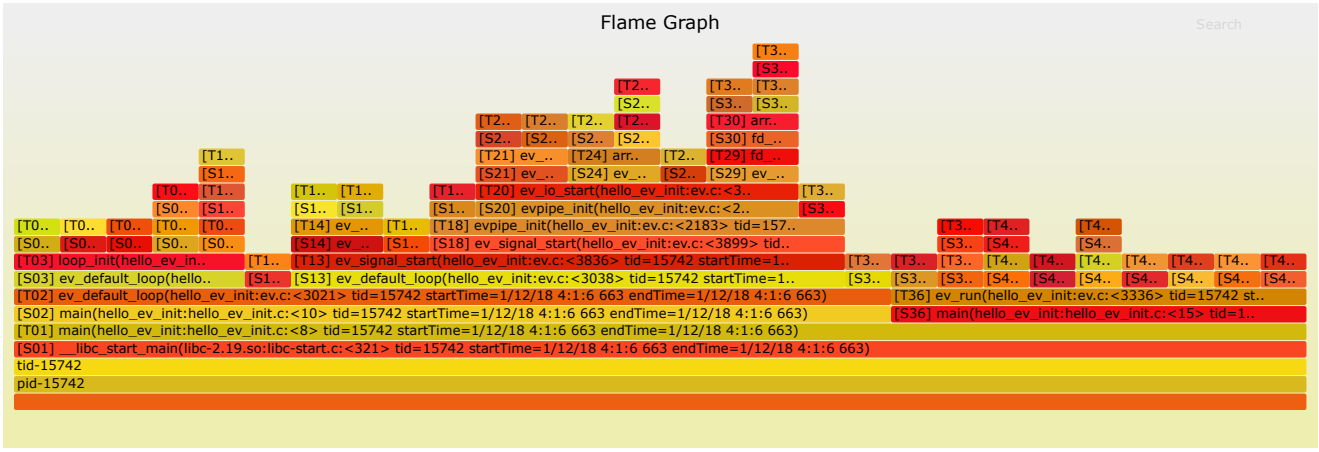


Appendix

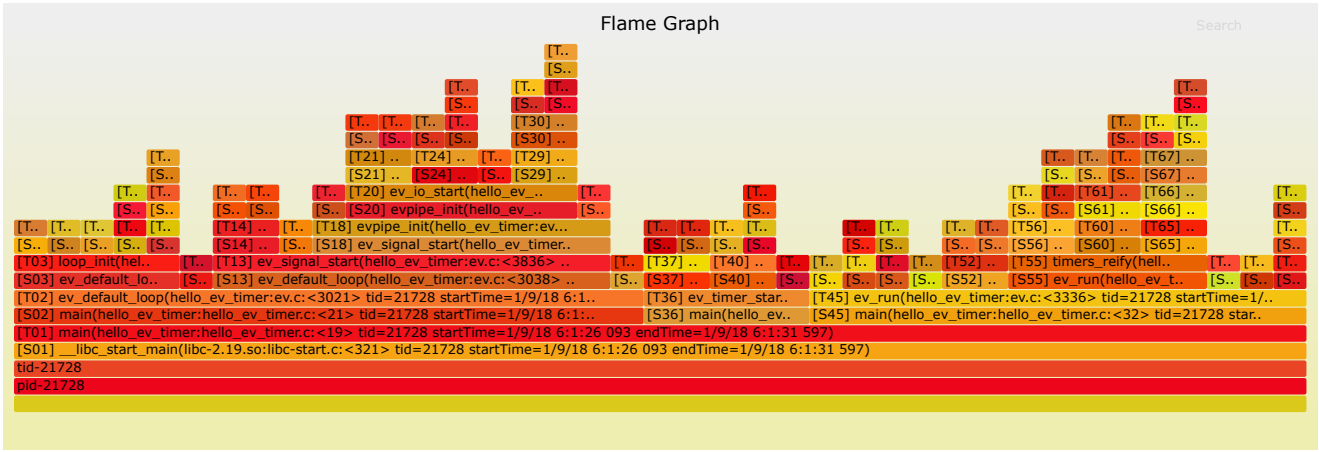
Detailed execution stack

The flamegraphs are generated by `CallTracer`

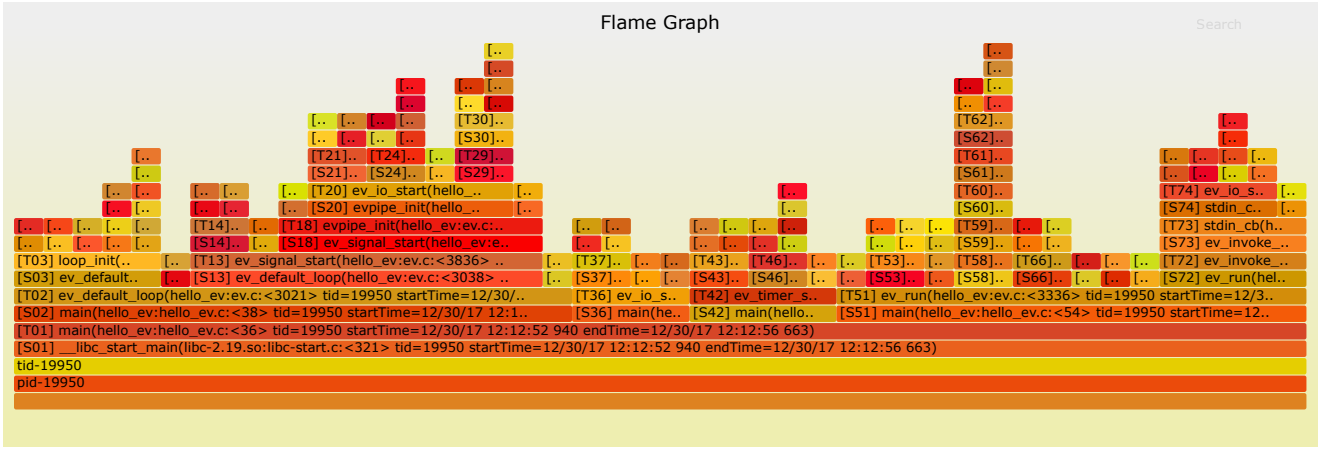
init



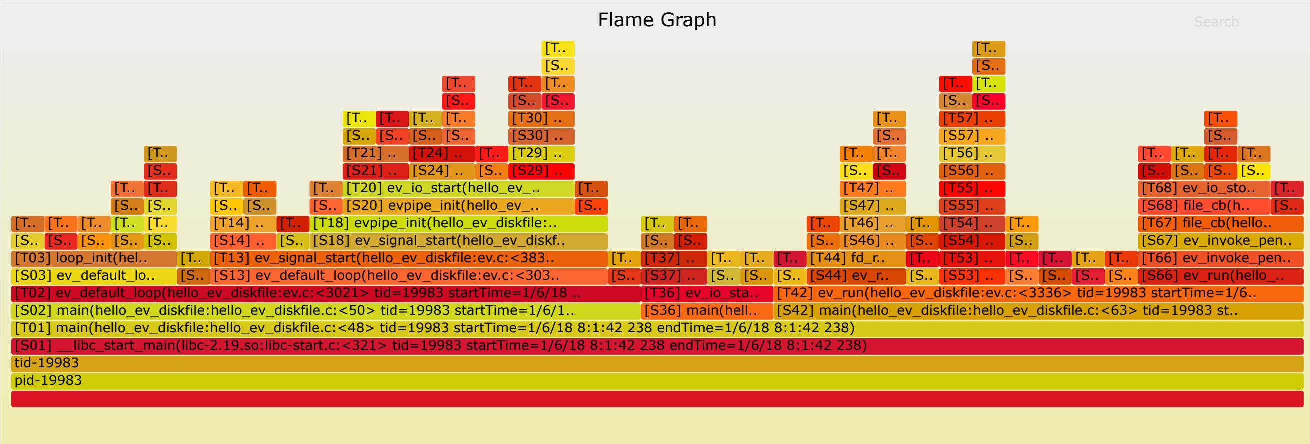
Timer



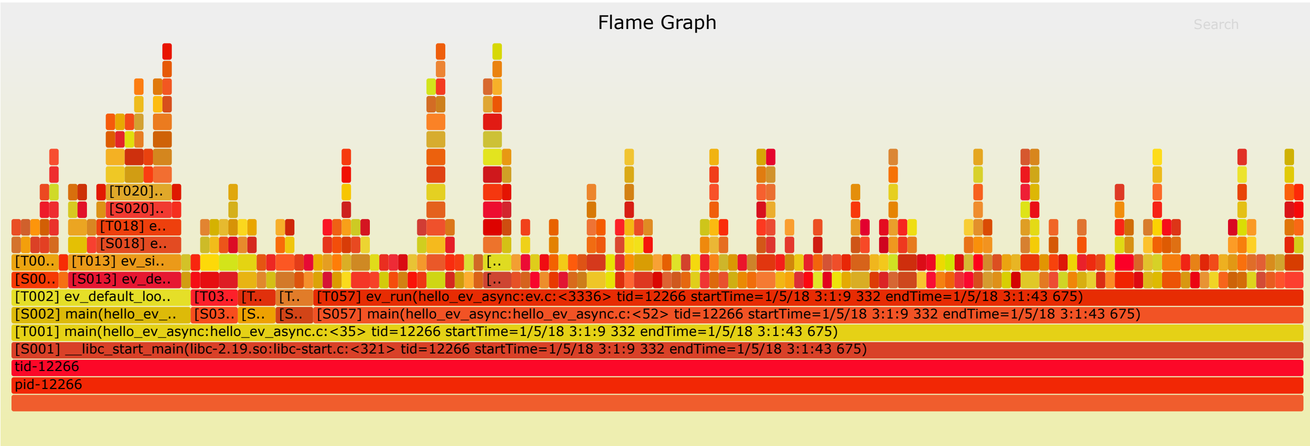
Stdio



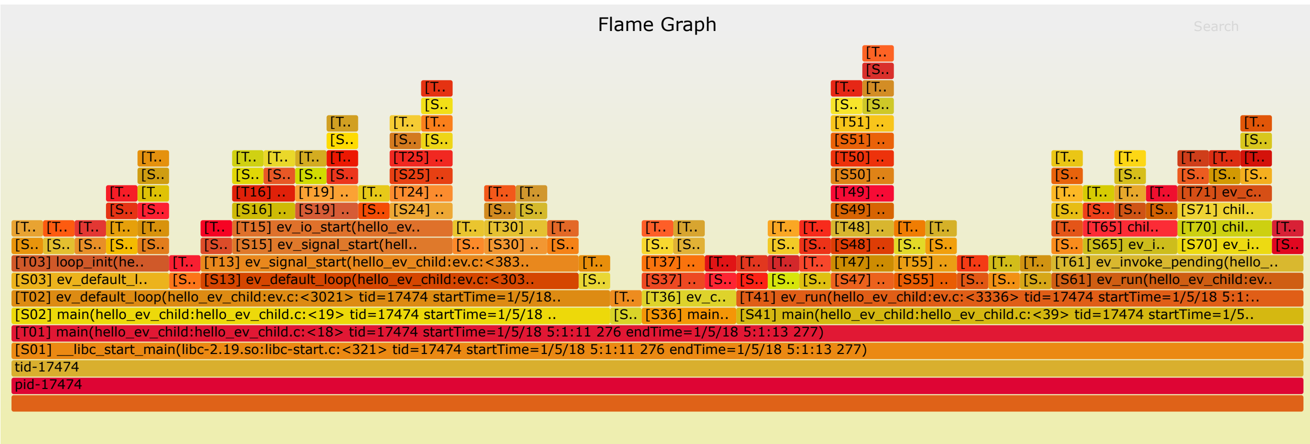
Regular File(a.k.a disk file)



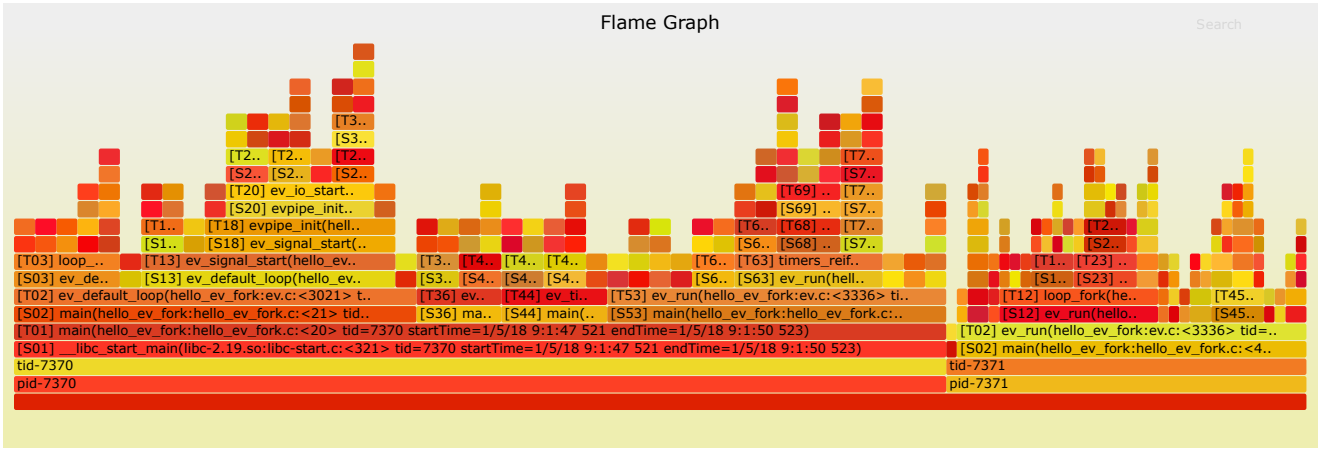
Async



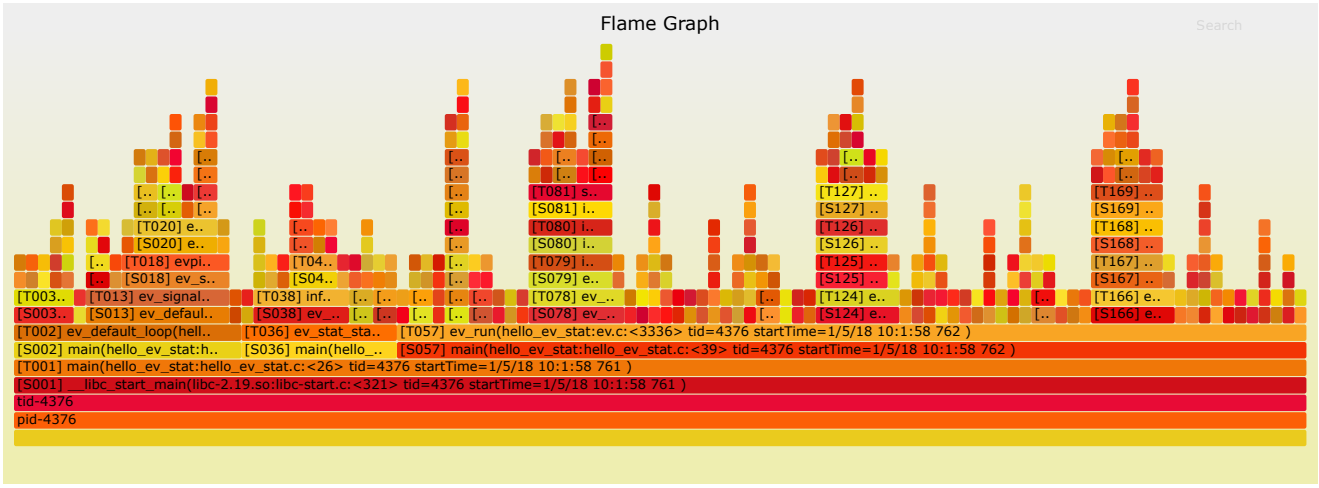
Child



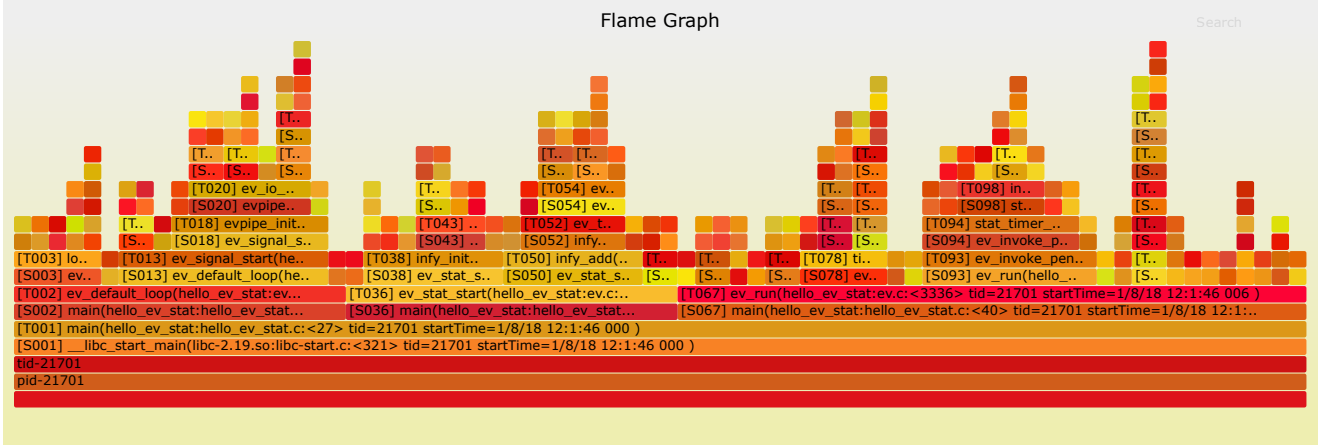
Fork



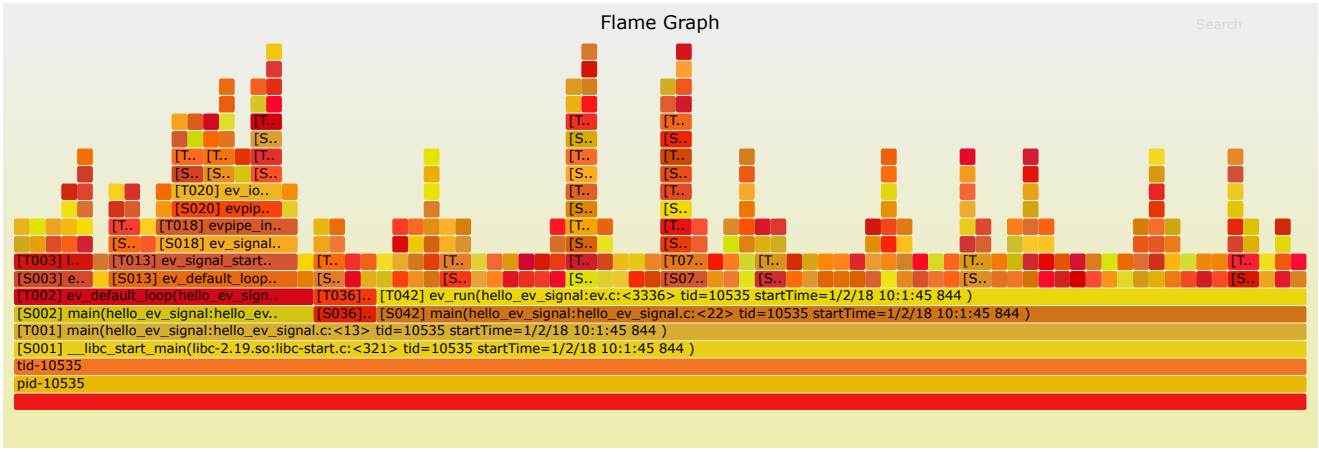
Local file stat(inotify)



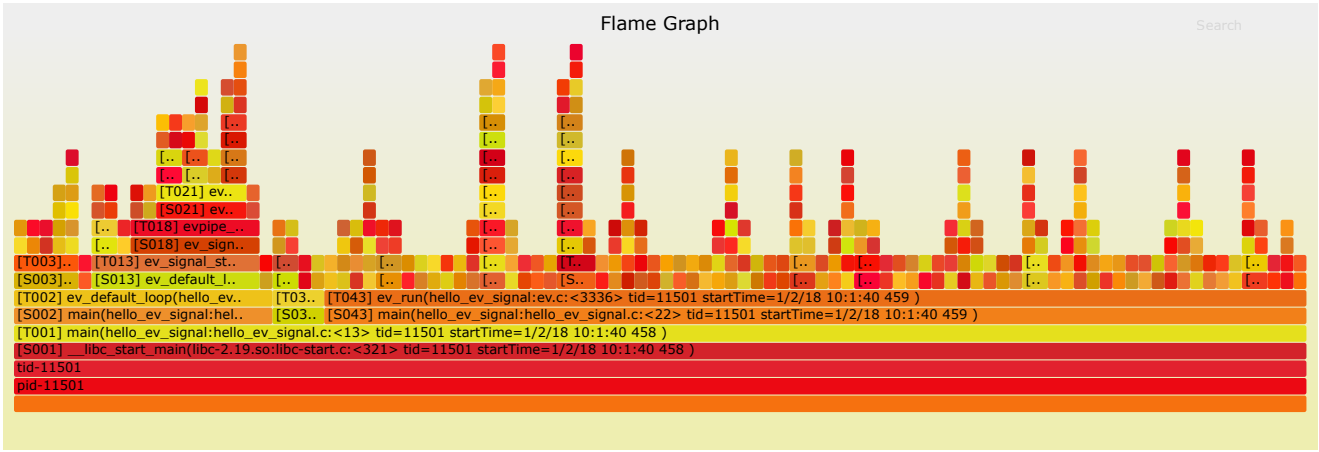
Remote file stat



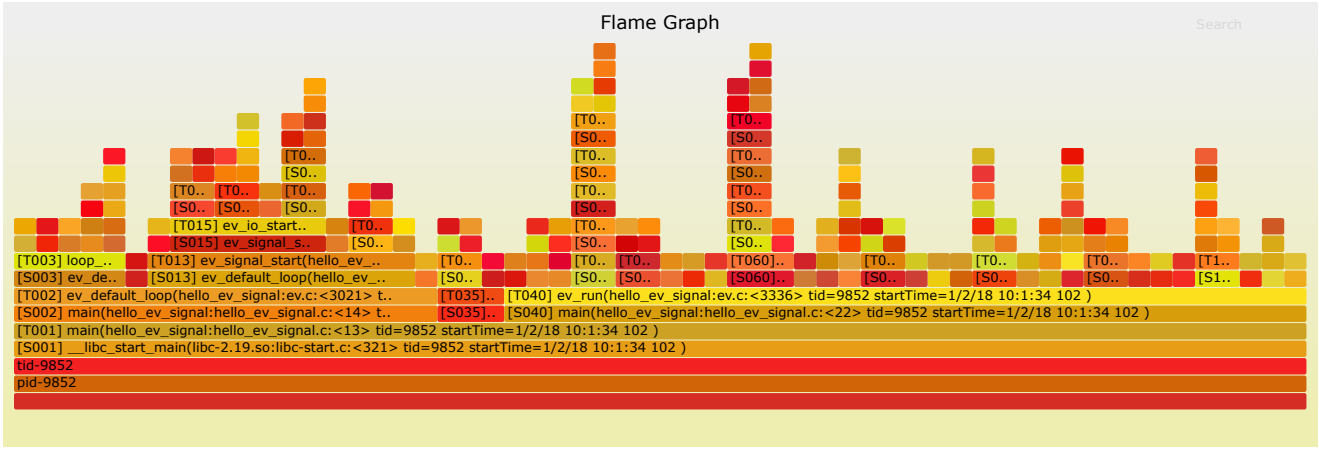
Signal(w/ eventfd)



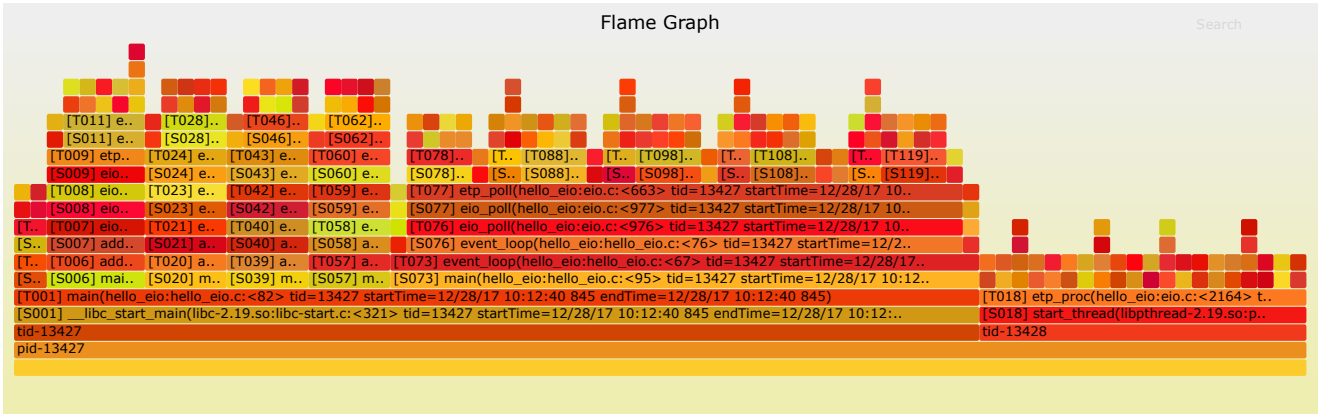
Signal(w/ pipe)



Signal(w/ signalfd)



Libeio(custom-handler)



This article is **Part 2** in a **2-Part** Series.

- Part 1 - Boost I/O Strategy In Web Server - Motivation
- Part 2 - This Article

Back to Top

← Build a Modern Scalable System - Practice on Service Mesh Mode with Consul, Nomad and Envoy

Exploring USDT Probes on Linux →

Show Comments (0)