



Paul's Notes

Experience, Record, Share.

- [RSS](#)

▾

- [Blog](#)
- [Archives](#)

Worker Pool With Eventfd

Nov 9th, 2018 10:41 pm | [Comments](#)

Linux Eventfd Overview

An “eventfd object” can be used as an event wait/notify mechanism by user-space applications, and by the kernel to notify user-space applications of events. It has been added to kernel since Linux 2.6.22. And the object contains an unsigned 64-bit integer (uint64_t) counter that is maintained by the kernel. So it’s extremely fast to access.

```
1 #include <sys/eventfd.h>
2 int eventfd(unsigned int initval, int flags);
```

That’s all we need to create one eventfd file, after that, we can perform normal file operations (like read/write, poll and close) with it.

Once some user-space thread write it with value greater than 0, it will instantly be notified to user-space by kernel. Then, the first thread which read it, will reset it (zero its counter), i.e. consume the event. And all the later read will get Error (Resource Temporarily Unavailable), until it is written again (event triggered). Briefly, it transforms an event to a file descriptor that can be effectively monitored.

There’re several notes of which we should take special account:

Applications can use an eventfd file descriptor instead of a pipe **in all cases where a pipe is used simply to signal events**. The kernel overhead of an eventfd file descriptor is much lower than that of a pipe, and only one file descriptor is required (versus the two required for a pipe).

As with signal events, eventfd is much more light-weight (thus fast) compared to the pipes, it’s just a counter in kernel after all.

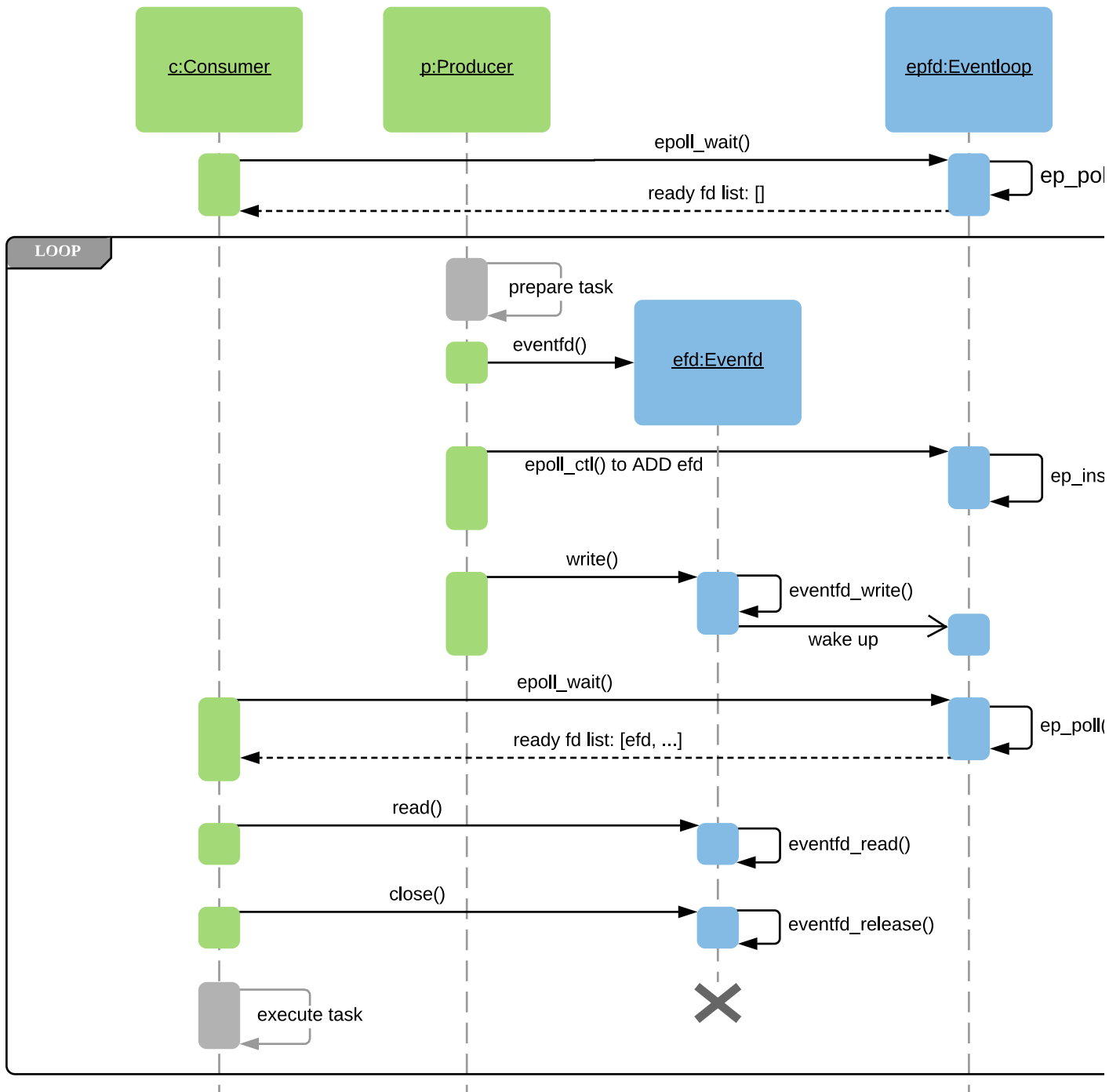
A key point about an eventfd file descriptor is that it can be monitored just like any other file descriptor using select(2), poll(2), or epoll(7). This means that an application can simultaneously monitor the readiness of “traditional” files and the readiness of other kernel mechanisms that support the eventfd interface.

You won’t wield the true power of eventfd, unless you monitor them with epoll (especially EPOLLET).

So, let’s get our hands dirty with an simple worker thread pool!

Worker Pool Design

We adopt Producer/Consumer pattern for our worker thread pool, as it’s the most common style of decoupling, achieving the best scalability. By leveraging the asynchronous notification feature from the eventfd, our inter-thread communication sequence could be described as following:



Implementation

Our per-thread data structure is fairly simple, only contains 3 fields: `thread_id`, `rank` (thread index) and `epfd` which is the epoll file descriptor created by `main` function.

```

1 typedef struct thread_info {
2     pthread_t thread_id;
3     int rank;
4     int epfd;
5 } thread_info_t;

```

Consumer thread routine

```

1 static void *consumer_routine(void *data) {
2     struct thread_info *c = (struct thread_info *)data;
3     struct epoll_event *events;
4     int epfd = c->epfd;
5     int nfds = -1;
6     int i = -1;
7     int ret = -1;
8     uint64_t v;
9     int num_done = 0;
10
11     events = calloc(MAX_EVENTS_SIZE, sizeof(struct epoll_event));
12     if (events == NULL) exit_error("calloc epoll events\n");

```

```

13
14     for (;;) {
15         nfds = epoll_wait(epfd, events, MAX_EVENTS_SIZE, 1000);
16         for (i = 0; i < nfds; i++) {
17             if (events[i].events & EPOLLIN) {
18                 log_debug("[consumer-%d] got event from fd-%d",
19                     c->rank, events[i].data.fd);
20                 ret = read(events[i].data.fd, &v, sizeof(v));
21                 if (ret < 0) {
22                     log_error("[consumer-%d] failed to read eventfd", c->rank);
23                     continue;
24                 }
25                 close(events[i].data.fd);
26                 do_task();
27                 log_debug("[consumer-%d] tasks done: %d", c->rank, ++num_done);
28             }
29         }
30     }
31 }

```

As we can see, the worker thread get the notification by simply polling `epoll_wait()` the epoll-added fd list, and `read()` the eventfd to consume it, then `close()` to clean it. And we can do anything sequential within the `do_task`, although it now does nothing.

In short: poll -> read -> close.

Producer thread routine

```

1 static void *producer_routine(void *data) {
2     struct thread_info *p = (struct thread_info *)data;
3     struct epoll_event event;
4     int epfd = p->epfd;
5     int efd = -1;
6     int ret = -1;
7     int interval = 1;
8
9     log_debug("[producer-%d] issues 1 task per %d second", p->rank, interval);
10    while (1) {
11        efd = eventfd(0, EFD_CLOEXEC | EFD_NONBLOCK);
12        if (efd == -1) exit_error("eventfd create: %s", strerror(errno));
13        event.data.fd = efd;
14        event.events = EPOLLIN | EPOLLET;
15        ret = epoll_ctl(epfd, EPOLL_CTL_ADD, efd, &event);
16        if (ret != 0) exit_error("epoll_ctl");
17        ret = write(efd, &(uint64_t){1}, sizeof(uint64_t));
18        if (ret != 8) log_error("[producer-%d] failed to write eventfd", p->rank);
19        sleep(interval);
20    }
21 }

```

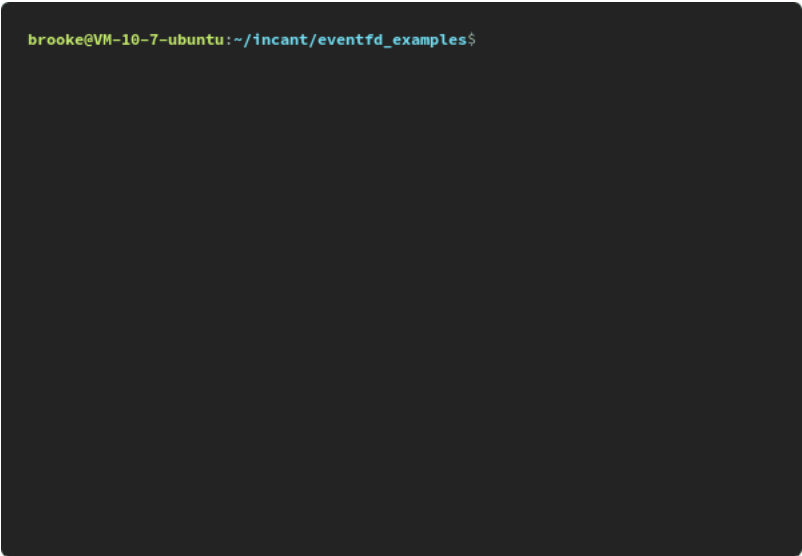
In producer routine, after creating `eventfd`, we register the event with epoll object by `epoll_ctl()`. Note that the event is set for write (EPOLLIN) and Edge-Triggered (EPOLLET). For notification, what we need to do is just write 0x1 (any value you want) to `eventfd`.

In short: create -> register -> write.

Source code repository: [eventfd_examples](#)

Output & Analysis

The expected output is clear as:



```
brooke@VM-10-7-ubuntu:~/incant/eventfd_examples$
```

You can adjust threads number to inspect the detail, and there's a plethora of fun with it.

But now, let's try something hard. We'll `smoke` test our worker by generate a heavy instant load, instead of the former regular one. And we tweak the producer/consumer thread to 1, and watching the performance.

```

1 static void *producer_routine_spike(void *data) {
2     struct thread_info *p = (struct thread_info *)data;
3     struct epoll_event event;
4     int epfd = p->epfd;
5     int efd = -1;
6     int ret = -1;
7     int num_task = 1000000;
8
9     log_debug("[producer-%d] will issue %d tasks", p->rank, num_task);
10    for (int i = 0; i < num_task; i++) {
11        efd = eventfd(0, EFD_CLOEXEC | EFD_NONBLOCK);
12        if (efd == -1) exit_error("eventfd create: %s", strerror(errno));
13        event.data.fd = efd;
14        event.events = EPOLLIN | EPOLLET;
15        ret = epoll_ctl(epfd, EPOLL_CTL_ADD, efd, &event);
16        if (ret != 0) exit_error("epoll_ctl");
17        ret = write(efd, &(uint64_t){1}, sizeof(uint64_t));
18        if (ret != 8) log_error("[producer-%d] failed to write eventfd", p->rank);
19    }
20    return (void *)0;
21 }

```

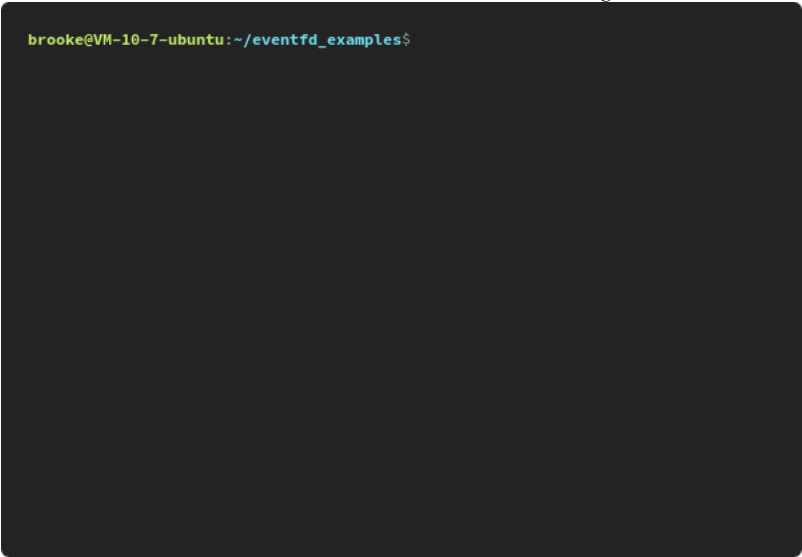
Over 1 million? Indeed! By using the `ulimit` command below, we can increase the open files limit of the current shell, which is usually 1024 by default. Note that you need to be root.

```

1 ulimit -n 1048576
2
3 # 1048576 is the default maximum for open files, as `/proc/sys/fs/nr_open` shows.
4 # To make it larger, you need to tweak kernel settings like this (which is beyond our scope)
5 # sysctl -w fs.nr_open=10485760

```

Since the info of stdout is so much that we redirect the stdout to file *log*.



```

brooke@VM-10-7-ubuntu:~/eventfd_examples$

```

With my test VM (S2.Medium4 type on [TencentCloud](#), which has only 2 vCPU and 4G memory, it takes less than 6.5 seconds to deal with 1 million concurrent (almost) events. And we've seen the kernel-implemented counters and wait queue are quite efficient.

Conclusions

Multi-threaded programming model is prevailing now, while the best way of scheduling (event trigger and dispatching method) is still under discussion and sometimes even opinionated. In this post, we've implemented general-purposed worker thread pool based on an advanced message mechanism, which includes:

1. message notification: asynchronous delivering, extremely low overhead, high performance
2. message dispatching: as a load balancer, highly scalable
3. message buffering: as message queue, with robustness

All the above are fulfilled by using basic Linux kernel feature/syscall, like `epoll` and `eventfd`. Everyone may refers to this approach when he/she designs a single-process performant (especially IO-bound) background service.

To sum up, taking advantage of Linux kernel capability, we are now managed to implement our high-performance message-based worker pool, which is able to deal with large throughput and of high scalability.

References

- [eventfd\(2\) - Linux Man Page](#)
- [eventpoll - Linux Source Code](#)
- [eventfd - Linux Source Code](#)

Posted by Paul (YangYang 杨阳) Nov 9th, 2018 10:41 pm [linux](#)

[« http load testing with wrk2 distributed load testing by locust »](#)

Comments

Does this article solve your problem, and what do you think?

23 Responses


Upvote


Funny


Love


Surprised


Angry


Sad

o Comments

Login ▾

G

Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

 6 **Share**

Best Newest Oldest

Be the first to comment.

[Subscribe](#) [Privacy](#) [Do Not Sell My Data](#)

About Me

I'm Paul, a software developer interested in system programming as well as web service design.

Recent Posts

- [Store and Pin Async Functions in Rust](#)
- [\[Solved\] Shadowsocks: Undefined Symbol](#)
- [Linux Seccomp Filters](#)
- [Distributed Load Testing by Locust](#)
- [Worker Pool With Eventfd](#)
- [Http Load Testing With Wrk2](#)
- [Building Linux Kernel Module](#)
- [Compiling Kernel With Kali Linux](#)
- [\[Solved\] FFMPEG: Libmp3lame Not Found](#)
- [Nodejs Circular Dependencies](#)

Tag Cloud

[blog](#) [cloud](#) [docker](#) [ffmpeg](#) [http](#) [javascript](#) [linux](#) [memcache](#) [network](#) [nginx](#) [node.js](#) [postgresql](#) [ssh](#)

Tire suas ideias do papel do papel

Tudo o que você precisa para vender na internet.

GoDaddy

Comprar >

Tire suas ideias do papel do papel

Tudo o que você precisa para vender na internet.

GoDaddy

Comprar >

Tire suas ideias do papel

Tudo o que você precisa para vender na internet.

GoDaddy

Compre