A New Lightweight Zero Copy Notification Mechanism in Linux

Zijian Zhang, Xiaochun Lu
ByteDance, San Jose, CA, USA
{zijianzhang, xiaochun.lu}@bytedance.com

Abstract

The flag MSG_ZEROCOPY extends zero copy to common socket sendmsg system calls, which is more efficient and flexible. However, copy avoidance is not a free lunch. It replaces the overhead of copy with page management and completion notifications. The reception of notifications incurs extra system call overhead from poll and recvmsg. Assume that applications which use MSG_ZEROCOPY are sensitive to CPU usage, the overhead of extra system calls can be unignorable. Moreover, following the introduction of the flag, page table isolation was added to mitigate the Meltdown vulnerability, which increases system call overhead. Thus, we introduce a new lightweight notification mechanism that embeds the notifications in msg_control. Similar to recvmsg, users can pass msg_control as a placeholder to sendmsg, and upon returning of it, the data of the msg_control will be updated by the kernel so that users can get the completion notifications. For evaluation, we add the new feature in msg_zerocopy selftest, 7%-17% performance gain in TCP and .3%-20% in UDP are observed, the results are dependent on the selftest configuration.

Keywords

MSG_ZEROCOPY, Notificaton Mechanism, Overhead Mitigation

Introduction

Linux has supported various of copy avoidance methods since the early days, such as sendpage and splice[4]. In Linux-4.14, Willem added a flag MSG_ZEROCOPY [1] which extends the copy avoidance mechanism to common socket send calls, users can simply pass the flag to sendmsg system call, and the underlying transmission will be zerocopy. Compared with sendpage and splice, MSG_ZEROCOPY is more efficient and flexible to use. However, copy avoidance is not a free lunch, the implementation of MSG_ZEROCOPY replaces per byte copy cost with page management and completion notification. The pinning of the page changes the semantics of the sendmsg system call. It temporarily shares the buffers between process and user stack. Returning of sendmsg does not mean the completion of transmission but the page pinning, so the process should not reuse or release the buffer at that time. After the users get completion notifications from the kernel, it means it's safe to modify the buffer.

Figure 1 shows the typical process of the usage of MSG_ZEROCOPY. The reception of notifications incurs extra system call overhead from poll and recvmsg, or at least recvmsg. Assume that applications which use MSG_ZEROCOPY are sensitive to CPU usage, the overhead of extra system calls can be unignorable. Especially with the higher syscall overhead due to meltdown mitigations[2] which is introduced after the sendmsg zerocopy. We try to mitigate the notification overhead with a new mechanism based on msg_control.

```
ret = send(fd, buf, sizeof(buf), MSG_ZEROCOPY);
if (ret != sizeof(buf))
    error(1, errno, "send");

pfd.fd = fd;
pfd.events = 0;
if (poll(&pfd, 1, -1) != 1 ||
    pfd.revents & POLLERR == 0)
    error(1, errno, "poll");

ret = recvmsg(fd, &msg, MSG_ERRQUEUE);
if (ret == -1)
    error(1, errno, "recvmsg");

read_notification(msg);
```

Figure 1: Using MSG_ZEROCOPY interface

Design

Stepping away from the existing method of obtaining notifications through MSG_ERRQUEUE, what we need is a mechanism that can pass some information from the kernel back to user space at the time sendmsg returns. It turns out that we already have a similar mechanism in recvmsg.

Users can pass msg_control as a placeholder to recvmsg, allowing the kernel to update the data segment of the control message with the information. This method enables the kernel to send certain information back to the user. In this method, the kernel can send some information back to the user. Figure 2 is the code snippet from Linux net selftest *timestamping.c*. In the function printpacket, the content of msg_control has been updated by the kernel, users can parse it and traverse through each cmsghdr to specifically handle them.

On the other hand, msg_control in sendmsg is used for sending information from the user to the kernel. In Figure

```
struct {
    struct cmsghdr cm;
    char control[512];
} control;

...
msg.msg_control = &control;
msg.msg_controllen = sizeof(control);

res = recvmsg(sock, &msg, recvmsg_flags|
    MSG_DONTWAIT);
if (res >= 0)
    printpacket(&msg, ...);
```

Figure 2: Usage of msg_control in recvmsg

```
msg.msg_control = control;
msg.msg_controllen = CMSG_SPACE(control)

cmsg = CMSG_FIRSTHDR(&msg);
cmsg->cmsg_level = SOL_SOCKET;
cmsg->cmsg_type = SO_TIMESTAMPING;
cmsg->cmsg_len = CMSG_LEN(sizeof(uint32_t));

val = sendmsg(fd, &msg, 0);
...
while (!recv_errmsg(fd)) {}
```

Figure 3: Usage of msg_control in sendmsg

3, the user uses cmsg SO_TIMESTAMPING to ask the kernel to record timestamps of this sendmsg, so that they can call recv_errmsg later to get the information.

If we can support passing msg_control as a placeholder to sendmsg just like recvmsg, then we can get the zerocopy notifications without extra system calls.

One might think of IO_URING when saving system calls is considered. The framework of IO_URING makes it internally suitable for notifications, and it can also save system calls, which serves the same goal as our mechanism. However, IO_URING requires non-trivial modifications to the applications, while our design aims to be compatible with the common socket API, and only needs minor modifications to applications that already use MSG_ZEROCOPY.

Figure 4 shows our paradigm of design. Users pass a cmsg SCM_ZC_NOTIFICATION as a placeholder to sendmsg, and upon retuning of it, users can get notifications by parsing msg_control.

The new feature of msg_control in sendmsg will be completely compatible with the original usage. The users can pass both the placeholder cmsg and the original cmsg to sendmsg, and only the placeholder cmsg will be updated and returned to the caller.

Our design expects information to be returned right after sendmsg finishes. In the case of zerocopy notification, it's possible that after returning of a sendmsg, it can only get the notifications for the previous sendmsgs excluding itself. The returning of the current sendmsg does not mean the transmission is completed. This is acceptable in most cases, because notifications have IDs.

However, assume an application has last several sendmsgs, and will not invoke sendmsg in a relatively long period. These sendmsgs may get empty notifications. How do we get notifications for these trailing sendmsgs? We expect our notification method to be compatible with the original one so that users can always use it as a fallback. Actually, in our implementation, users can even interleavingly use these two methods, or in hybrid mode.

```
cmsg = CMSG_FIRSTHDR(&msg);
cmsg->cmsg_type = SCM_ZC_NOTIFICATION;
...
cmsg = CMSG_FIRSTHDR(&msg);
cmsg->cmsg_type = SO_TIMESTAMPING;
...
ret = sendmsg(fd, &msg, MSG_ZEROCOPY);
if (ret >= 0) {
    print_zc_info(&msg, ...);
    hanlde_timestamping();
}
// Optional, for possible trailing notifications
ret = recvmsg(fd, &msg, MSG_ERRQUEUE);
read_notification(msg);
```

Figure 4: Expected usage of msg_control in sendmsg

```
struct msghdr {
    // msg_control related fields
    union {
        void  *msg_control;
        void __user *msg_control_user;
    };
    bool  msg_control_is_user : 1;
}

static int ____sys_sendmsg(...) {
    unsigned char ctl[sizeof(struct cmsghdr) + 20]
    __aligned(sizeof(__kernel_size_t));
    ...
    if (ctl_len > sizeof(ctl)) {
        ctl_buf = sock_kmalloc(sock->sk, ctl_len,
    GFP_KERNEL);
        if (copy_from_user(ctl_buf, msg_sys->
    msg_control_user, ctl_len))
            goto out_freectl;
        msg_sys->msg_control = ctl_buf;
        msg_sys->msg_control_is_user = false;
    }
    ...
}
```

Figure 5: msg_control in sendmsg

Implementation

Current msg_control in sendmsg

Before diving into the details of implementation, let's review the current msg_control related logic in sendmsg in Figure 5. Struct msghdr has a union that includes either a kernel pointer msg_control or a user pointer msg_control_user. Guided by the bit flag msg_control_is_user, the kernel knows what kind of pointer is stored in the union, so that it can read from or write to the pointer correctly.

In ____sys_sendmsg, the msg_control of msg_sys, which was a user pointer, is overwritten by a kernel pointer. It's either from a kernel stack or sock_kmalloc depending on the ctl_len. The reason for the overwriting is for the convenience of the further reading and writing in the kernel. Any updates will only affect the kernel buffer instead of the user buffer. It is the reason why sendmsg does not support cmsg_control as recvmsg.

```
static int sendmsg_copy_cmsg_to_user(struct msghdr
     *msg_sys,
         struct user_msghdr __user *umsg)
{
    struct msghdr msg_user = *msg_sys;
    ...
    msg_user.msg_control_is_user = true;
    msg_user.msg_control_user = umsg->msg_control;
    cmsg_ptr = (unsigned long)msg_user.msg_control
    ;
    for_each_cmsghdr(cmsg, msg_sys) {
        if (!CMSG_OK(msg_sys, cmsg))
            break;
        if (!cmsg_copy_to_user(cmsg))
            continue;
        put_cmsg(&msg_user, cmsg->cmsg_level, cmsg
    ->cmsg_type, cmsg->cmsg_len - sizeof(*cmsg),
    CMSG_DATA(cmsg));
    }
    // Update of msg_controllen and msg_flags
    return err;
}

static int ____sys_sendmsg(...)
{
    ...
    msg_sys->msg_control_copy_to_user = false;
    err = __sock_sendmsg(sock, msg_sys);
    ...
    if (msg && msg_sys->msg_control_copy_to_user
    && err >= 0) {
      ssize_t len = err;

      err = sendmsg_copy_cmsg_to_user(msg_sys, msg
    );
      if (!err)
        err = len;
    }
out_freectl:
    if (ctl_buf != ctl)
        sock_kfree_s(sock->sk, ctl_buf, ctl_len);
out:
    return err;
}
```

Figure 6: Implementation

Generic cmsg copy back framework in sendmsg

In our implementation Figure 6, we have a generic framework to copy cmsg back to the user as demanded. We introduce a new bit flag msg_control_copy_to_user in msghdr to denote whether some cmsgs need to be copied back. In __sock_cmsg_send, if a cmsg needs to be copied back, its specific handler function can update the kernel buffer directly and set the new bit flag to true.

In sendmsg_copy_cmsg_to_user, we create a msg_user from msg_sys, and update the msg_control of it back to the original user pointer. Then, we have a for loop traversing each cmsg and find out the ones that need to be copied back. put_cmsg will handle the copy logic, which considers compat case so that it works for both 32-bit and 64-bit programs.

SCM_ZC_NOTIFICATION

To make use of the generic copy back framework, we need to first add the cmsg_type in cmsg_copy_to_user. In __sock_cmsg_send, we have the dedicated handler for cmsg_type SCM_ZC_NOTIFICATION in Figure 7. It gets

```
static inline bool cmsg_copy_to_user(struct
    cmsghdr *__cmsg) {
  return __cmsg->cmsg_type == SCM_ZC_NOTIFICATION;
}

int __sock_cmsg_send(struct sock *sk, ...) {
    switch (cmsg->cmsg_type) {
        case SCM_ZC_NOTIFICATION:
        // Get the zc information from
    MSG_ERRQUEUE
        // Populate the msg_control kernel buffer
    with the zc information
        msg->msg_control_copy_to_user = true;
    }
}
```

Figure 7: Code for SCM_ZC_NOTIFICATION

```
struct zc_info_elem {
    __u32 lo;
    __u32 hi;
    __u8 zerocopy; // Whether it is zerocopy or
    reverted back to copy
}

struct zc_info_usr {
    __u64 usr_addr;
    __u64 usr_size;
}

int __sock_cmsg_send(struct sock *sk, ...) {
    switch (cmsg->cmsg_type) {
        case SCM_ZC_NOTIFICATION:
        ...
        zc_info_usr_p = (struct zc_info_usr *)
    CMSG_DATA(cmsg);
        usr_addr = (void *)(uintptr_t)(
    zc_info_usr_p->usr_addr);
        ...
        // Get the zc information and stores it
    into a kernel buffer, then copy to user
        copy_to_user(usr_addr, &zc_info_kern,
    copy_size)
        ...
    }
}
```

Figure 8: Alternative method based on msg_control

the completion notifications from the same data source with the original method, and then populate the msg_control kernel buffer with the zc information. Finally, it sets msg_control_copy_to_user bit flag to true to trigger the copy back function later.

Limitations and Workarounds

The overhead in hotpath of sendmsg

In the hotpath of sendmsg, our method adds a minor cost of initialization and condition check for every other sendmsgs that do not use the new feature. We plan to add a static_branch in ____sys_sendmsg to make the branch a NOOP in the common case, and only enable it on the first setsockopt SO_ZEROCOPY.

The ABI change of sendmsg

This framework incurs the ABI change of sendmsg, which changes the semantics of msg_control. This is the main con-

```
do {
    sends_since_notify++;
    do_sendmsg(fd, &msg, cfg_zerocopy, domain);

    if (sends_since_notify >=
    cfg_notification_limit) {
        do_recv_completions();
        sends_since_notify = 0;
    }
} while (gettimeofday_ms() < tstop);
```

Figure 9: msg_zerocopy selftest

cern of this method, it needs more opinions from the community before it can be merged into the mainline. As a fallback, we have another proposal that requires a change in the handler of SCM_ZC_NOTIFICATION only.

Instead of passing msg_control as a placeholder, we embed struct zc_info_usr, which includes a user address pointing to an array of zc_info_elem and the size of this array, into the cmsg. In the handler of SCM_ZC_NOTIFICATION, the kernel retrieves the zc information from MSG_ERRQUEUE and copies at most some size of elements to the user address.

This trick circumvents having to deal with compat issues and having to figure out copy_to_user in ___sys_sendmsg. However, this is quite hacky, from an API design point of view, and still needs more opinions from the community.

Evaluation

We reuse the net selftest *msg_zerocopy.c* with modification for the new method to do the evaluation. In the selftest, there are two localhost sockets, one for sending, and the other one for receiving. As shown in Figure 9, the sending socket keeps calling sendmsg at a certain period. After some number of sendmsgs, it will receive the remaining completion notifications. At the end of the selftest, it will output the throughput in this period, the higher the throughput, the better the performance is.

When the notification interval is 1, which means the sending socket will get notifications after each one sendmsg. In this case, the overhead of the original method is the highest, for each sendmsg, there is a corresponding recvmsg. In this case, our method has a 16% performance gain in TCP and around 20% in UDP.

	TCP v4	TCP v6	UDP v4	UDP v6
ZC(MB)	7523	7706	7489	7304
New ZC(MB)	8834	8993	9053	9228
Gain	17.42%	16.70%	20.88%	26.34%

When notification the interval is 32, which means the sending socket will get notifications after each 32 of sendmsgs. which means less overhead of poll + recvmsg for the original method. In this case, the new method has around 7% CPU savings in TCP and slightly better CPU usage in UDP. In the context of the selftest, notifications of TCP are more likely to be out of order than UDP, it's easier to coalesce more notifications in UDP. The original method can get one notification with a range of 32 in a recvmsg most of the time. In TCP, most notifications' range is around 2, so the original method needs around 16 recvmsgs to get notified in one round. That's the reason for the "New ZCopy / ZCopy" diff in TCP and UDP here.

In conclusion, when the notification interval is small or notifications are hard to coalesce might because of being out of order, the new mechanism is highly recommended. Otherwise, the performance gain from the new mechanism is very limited.

	TCP v4	TCP v6	UDP v4	UDP v6
ZC(MB)	8842	8735	10072	9380
New ZC(MB)	9366	9477	10108	9385
Gain	6.00%	8.28%	0.31%	0.01%

Possible Use Cases

Besides MSG_ZEROCOPY, any other use cases where users need to obtain information from the kernel via a combination of sendmsg and recv_errmsg could potentially benefit from this method, such as timestamps.

On the other hand, Homa[3], a receiver-driven low latency transport protocol, could also benefit from the mechanism. In homa_send, users need to pass down a homa_sendmsg_args structure as an argument, and upon returning, homa_rpc id in this struct needs to be updated for further use by the user space. While passing msg_control carrying the homa_sendmsg_args to sendmsg, Homa sets msg_controllen to 0 to avoid the msg_control being overwritten to a kernel buffer in sys_sendmsg. Although it works for Homa, it is very hacky, and compromises the semantics of msg_control. Our method could be helpful in this case.

Conclusion

In this paper, we identify the overhead of system calls introduced by the completion notifications in MSG_ZEROCOPY. We try to solve the problem by supporting copying some information back to the user upon returning of sendmsg based on msg_control. By leveraging this new feature, we introduce a new notification mechanism for MSG_ZEROCOPY. In the experiments, compared with the original method, we get 7%-17% performance gain in TCP and .3%-20% in UDP. When the notification interval is small or notifications are hard to be coalesced might because of being out of order, the new mechanism is highly recommended. Otherwise, the performance gain from the new mechanism is very limited. Our implementation method introduces a change in the ABI of sendmsg, a workaround does not need any change to sendmsg but makes the API of msg_control hacky. Both methods need more opinions from the community.

References

[1] de Bruijn, W., and Dumazet, E. 2017. sendmsg copy avoidance with msg_zerocopy. https://netdevconf.info/2.1/papers/debruijn-msgzerocopy-talk.pdf. Accessed: 2023-10-05.

[2] Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Fogh, A.; Horn, J.; Mangard, S.; Kocher, P.; Genkin, D.; Yarom, Y.; and Hamburg, M. 2018. Meltdown: Reading

kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[3] Montazeri, B.; Li, Y.; Alizadeh, M.; and Ousterhout, J. 2018. Homa: a receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM '18: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 221–235. ACM.

[4] Torvalds, L. 2006. Explaining splice() and tee(). LKML email thread. (Re: Linux 2.6.17-rc2).