Zephyr® Project
Developer Summit

Inter-Process Confusion

Navigating Zephyr’s IPC APIs

Andy Ross <andyross@google.com>
Discord: @andyross

THE LINUX FOUNDATION #EmbeddedOSSummit

Inter-Process Communication

Zephyr has a lot of “tools in the box” at this layer of the core kernel APIs

- “Mature” system means “not all choices are the best choices”
- Too much choice leads to bad decisions, missed opportunities
- Zephyr is still an RTOS
  - Almost all apps run with size constraints
  - All code has size costs
  - Choosing the “perfect” tool for every subproblem leads to a needlessly large imperfect app!

What is this talk?

- Intermediate-level, no arch layer, no assembly, no hardware details
- Experienced Zephyr developers wanting a deep dive into things they missed
- Experts from other systems wanting a tour of the API

Why is there an IPC layer anyway?

We need IPC because we have processes (threads) that must communicate

- Threads introduce parallelism and parallelism requires synchronization
- Synchronization is hard
- IPC provides simpler tools with simpler rules
    - Core metaphor: **IPC means "wait until"**
    - Wait until something is ready, or finished, or available

Why so many threads, though?

- Threads allow subproblems to be expressed in sequential code
- Sequential code is easier to reason about vs. e.g. async/callback code
- Logic often imported from elsewhere, already sequential
- Conway's Law makes it hard for large systems to coordinate on algorithms
    - Every team ends up with its own "main loop", etc…
- Usually **not** for performance (v. occasionally SMP scaling)

Why is there an IPC layer anyway? …

What would we do without IPC tools?

- New devs usually try polling
- Naively done produces spinning, power problems, dead/livelocks
- Usually need a k_sleep()
- Race conditions lurk everywhere

```
void main_loop(void)
{
    while (true) {
        if (something_to_do()) {
            do_something();
        }
        /* k_yield() */ // Why won't this work?

        // 20ms supposed to be OK, but sometimes runs long becuase
        // something_to_do() returns false and sleeps twice!?
        k_sleep(K_MSEC(10));
        /* k_sleep(K_MSEC(20)); */
    }
}
```

Must IPC be provided by the kernel?

Well, no, these are general problems

Frameworks have their own abstractions

- ZBus: emerging systemwide high level interconnect
- Network subsystem has a built-in IPC layer (BSD sockets!)
- CMSIS abstraction provides wrappers for low-level primitives
- Alas, frameworks are out of scope for this talk

High performance IPC might even skip the kernel entirely!

- spsc/mpsc lockless messaging utilities
- winstream: lockless byte stream API (cross-bus logging on adsp)
- Lockless algorithms are **way** out of scope for this talk

What if we don’t want all those threads?

IPC is about coordinating threads, so… skip the threads
Works Queues:

- Simple callbacks run in order (with optional timed delay)
- One manager thread per queue, lightweight
- No need to decide on “until”, just insert the item when it’s ready to run
  - Actually stronger: must not spin, wait, sleep or otherwise delay the queue!
- This is async code, analogous to JS frameworks, Python async, etc…
  - “Inside out” logic – you specify what happens after instead of writing a sequence
  - Sometimes messy, and C doesn’t have the tools and syntax help fancy runtimes do!
- No scheduler/priority/ISR interaction, just runs everything in order
  - So, not very “real time”
  - (Consider p4wq, which has a similar API but where workers are preemptible threadlets)

Basic Building Blocks

All these tools are built on the same mechanisms in the kernel

Mutual exclusion

- Basic requirement for any parallel code
- Difficult to reason about
  - Is this data inside the lock?
  - Are these locks properly nested, or will they deadlock?
- At bottom layer implemented with a spinlock
  - Which is just a global interrupt lock in 1-cpu contexts
  - Doesn’t work in userspace
  - Latency problems if critical sections aren’t small
  - Generally not for application use

Basic Building Blocks …

Blocking ("pending" in Zephyr kernelese)

- Threads can't be running all the time, need to be able to wait **until**
- Implemented with a "wait queue" in the kernel
  - Simple list of threads, sorted in priority order
  - Must be called while holding a spinlock, which scheduler will release
- Takes a "timeout" parameter
  - How long to wait until returning with -EAGAIN (or K_FOREVER)
  - K_NOWAIT: enables "nonblocking I/O" and use in ISRs
  - Essentially all Zephyr APIs that can block expose this timeout
  - Semi-unique Zephyr feature, other systems make timeouts API-specific
    - Unix select/poll has it, but read/write don't, setsockopt(SO_RCVTIMEO), etc…

Core IPC primitive: semaphores

Textbook tool

- Conceptually a “queue” w/o data
- Stuff a token in when ready
- Stores count of tokens
- Read to **wait until** token ready

Historical naming is awful

- V/P, up/down, post/wait
- Zephyr: give/take

Great tool

- Fast, small, “fair”
- Used by everything, already present

```
K_SEM_DEFINE(my_sem, 0, INT_MAX);

void my_reader()
{
    while(k_sem_take(&my_sem, K_FOREVER) == 0) {
        printk("Reader awake");
    }
}

void my_writer()
{
    while(true) {
        k_sleep(K_MSEC(500));
        k_sem_give(&my_sem);
    }
}
```

Semaphores as locks

Useful trick:

- Set count to 1
- k_sem_take()
    - First caller returns
    - Everyone else blocks
- k_sem_give()
    - Wakes up highest priority waiter
    - Or remembers the count for next locker
- That just a lock!
    - This trick is used pervasively in the kernel
    - Basically the smallest/fastest way to do blocking mutual exclusion
    - Only slightly slower than a spinlock
    - Use this pattern

```c
/* Start count 1, limit 1 to limit number of "owners" to 1 */
static K_SEM_DEFINE(subsys_lock, 1, 1);

void take_subsys_lock()
{
    int ret = k_sem_take(&my_sem, K_FOREVER);
    __ASSERT(ret == 0); /* Maybe handle better */
}

void release_subsys_lock()
{
    k_sem_give(&my_sem);
}
```

Dedicated mutual exclusion: k_mutex

Semaphores are most common, but not official

k_mutex is the Standard Zephyr Lock

- Detects misuse (multiple unlocks) more cleanly
- Provides “priority inheritance”
  - A high-priority waiter “loans” its scheduler priority

But really… mostly worse

- Bigger, slower
- PI only works if all blocking is on k_mutex
- PI requires kernel data (see below re: userspace)
- Harmless, works fine, but avoid unless required

Queueing as near-universal metaphor

"**Wait until** someone else puts something in the queue"
Semaphores do the wait on read part, but what if the something has data?
"**Wait until** there is space in the queue"

Unix does this natively with just one API (the file descriptor)

- Scales from C10k network server processes down to shell pipelines

```
git log | grep ^Author | sort | uniq -c | sort -n
```

Zephyr's not so elegant, we have a ton

- For good reason though, threads share memory, lots of options for "best"

Queue Abstractions in Zephyr

k_fifo

- (Also k_lifo, both specializations of k_queue)
- Singly-linked list of items
- Insert at either end, remove from front
- Blocks on empty list
- Intrusive: list node is part of app data

Small, simple, pervasive. Use this!

```
struct my_rec {
    int data;
    char buf[16];
    void *queue_node;
};

static K_FIFO_DEFINE(my_queue);

void enqueue_rec(struct my_rec *r)
{
    k_fifo_put(&my_queue, &r->queue_node);
}

struct my_rec *dequeue_rec(void)
{
    void *node = k_fifo_get(&my_queue, K_FOREVER);
    return CONTAINER_OF(node, struct my_rec, queue_node);
}
```

More Queue Abstractions in Zephyr …

Message queues: k_msgq

- Data elements are fixed-size (!) “messages”
- Copied into and out of a object-maintained buffer
- Simple and reasonable if it fits the problem

Pipes: k_pipe

- Ordered stream of bytes
- Can be written or read in arbitrary quantities
- Also fixed-size buffer

Both good choices, but overlap.

- Probably don’t use both. If you need both, just use a pipe

Digression: booby-trap race with “conditional” queuing

Consider an app that needs to make a “choice” about when to read from a queue

- Maybe more than one input, special mode, special circumstance
- What if the condition **changes** after you check it but before you read?
  - Someone else got it first: deadlock
  - Maybe you just missed it: latency bug
- Sounds like a data race: put a lock around it
  - Still doesn’t work, unfixable delay between lock release and k_sem_take()/k_queue_get()/etc…

```
static struct k_fifo *curr_fifo;

// THIS IS A RACE, don't do this
void set_queue(struct k_fifo *new_fifo)
{
    curr_fifo = new_fifo;
}

void enqueue_rec(struct my_rec *r)
{
    k_fifo_put(curr_fifo, &r->queue_node);
}

struct my_rec *dequeue_rec(void)
{
    // RACE HERE: between the load of curr_fifo and the pend call
    // under k_fifo_get(), the fifo can change!  Probably causing a
    // deadlock...
    void *node = k_fifo_get(curr_fifo, K_FOREVER);
    return CONTAINER_OF(node, struct my_rec, queue_node);
}
```

General solution: condition variables

"Most general IPC primitive"

- Can implement other primitives, but not the reverse
- Basic abstraction: wait (**until!**) and signal/wakeup
  - Combines "wait" with "release lock", so decision is atomic
- We've seen it before! (k_spinlock + waitq)
- Uses k_mutex, not k_sem!
- Simple rules:
  - Take lock, inspect state, and either release or k_cond_wait()
  - While holding lock, call k_cond_signal() to wake up a waiter
  - Rules are simple, implementation is not!
- Use this first any time you know you have complicated synchronization
  - But recognize that you have a hard problem

What if you need to wait on more than one thing?

Common problem for “main loops” with input from lots of sources

- Can sometimes solve with more threads
  - I.e. a msgq that multiplexes inputs with a reader for each
  - Threads are expensive

k_poll()

- Analogous to Unix select()/poll(), provide list of inputs to wait on
  - Not the same as the select() in the network layer!
- Works with k_sem, queue, msgq and pipe
  - “Level triggered”, if you don’t handle the input k_poll() will still see it
- k_poll_signal allows for user-defined wakeup sources
  - One shot! App code must reset it for another k_poll() call
- Somewhat heavyweight, lots of code required
  - https://docs.zephyrproject.org/latest/kernel/services/polling.html

k_event, lightweight multi-waiting

Newer tool (2.5 years old)

Think of it like a DIY k_poll:

- All wakeup sources are manual/app-managed
- Stored as a bitmask of 32 possible “events”
- Can wait on any subset of them via mask
- Can signal any subset via mask
  - Level or edge triggered via API parameter (i.e. “reset after signal?”)
- Much smaller than k_poll
- Best choice, unless you have >32 sources to multiplex
  - Or have to interoperate with existing IPC primitives

What about memory protection and userspace?

In theory USERSPACE=y doesn't change the APIs at all

- All IPC primitives are k_obj objects and live in kernel space
- All IPC calls become syscalls and work identically
    - Actually k_queue (intrusive) needs a variant API that does copies: avoid

Userspace radically changes the **performance** characteristics of IPC

- k_sem_down() goes from a dozen instructions to many hundreds!
    - Used to be just decrementing a single variable in the uncontended case!
- No more cheap semaphore locks
- Basically, it starts to look like Unix
    - "Process I/O is expensive"
    - Pipes start to look better: no longer expensive general choice, now just as slow as everything else

Can't we get back fast userspace IPC?

Well, sort of: sys_sem

- Like k_sem, but based on a shared atomic count and a k_futex primitive
  - Must be in same mem_domain
  - k_futex works like a subset of the Linux syscall
- Frustratingly not interoperable: separate code with separate API
  - Slightly slower than k_sem when USERSPACE=n
  - Separate code, can't freeload on kernel use of k_sem
  - Would be nice to have one API that worked in both modes
    - I tried once ("zync"). It **almost** worked, but so much framework code to update

Also sys_mutex

- **Avoid**. It's just a wrapper around k_mutex with no advantages
- Problem isn't solvable (priority inheritance requires kernel involvement, there is no "fast path" possible using only userspace memory)

POSIX subsystem IPC

POSIX defines a lot of these same primitives too

- Some are simple wrappers around matching Zephyr abstractions
  - Mutex and condition variable
- pthread_spinlock is also a wrapper, but avoid without good reason
  - spinlocking in preemptible code is a terrible latency trap! Use native spinlocks if you have a problem with this kind of requirement
- POSIX “mq” is **not** a wrapper around k_msgq
- “rwlock” (read/write lock) is POSIX-only, no Zephyr equivalent
  - Not a ton of value on RTOS machines with only ~4 CPUs, not much contention
- Likewise pthread_barrier (“pistol start” for waiting threads) is unique
- eventfd duplicates the Linux (not actually POSIX) syscall
  - Mostly for interop with network layer poll() call
  - Behaviorally mostly just a semaphore

A few others to mention

Mailboxes: k_mbox

- Like a queue, but a "target" thread can be specified for each message
- Requires O(N) search of the list for each message!
- Doesn't work in ISRs (can't read, because not a thread)
- Avoid. Historical API, hasn't been touched in years. Scales poorly.

Stacks: k_stack

- Not a process stack!
- Like a "msgq" version of k_lifo: last-in/first-out, copies data in/out of buffer
- Works fine, clean, small code size
- Almost never used. Just a weird gadget.

Final Recommendations

- Use subsystem IPC if that's what your code is doing, don't rock boats
    - Network/BT code should be using poll() with eventfd glue
    - POSIX code should stick to POSIX calls
- Lean heavily on the simple abstractions
    - Semaphores for both locks and "queuing"
    - k_queue/fifo/lifo if you have your own records (and won't need userspace)
    - k_pipe if you need to buffer data
    - k_event to multiplex inputs
- **Never predicate your blocking calls**
- Don't get fancy
- If you need to get fancy, **start** with k_condvar
    - If your solution is a slightly imperfect fit for existing primitives, it's probably wrong
- Find us on Discord for support!
    - Most of the inspiration for this talk was an intuition about "Common Questions on Discord"

Zephyr® Project

Developer Summit