Zircon Kernel Introduction

2018/01/16

Brief Introduction

- The kernel and core platform of Google Fuchsia OS
- Derived from LK (Little Kernel) but quite different nowadays
- 64bit only system (support X86-64 and ARM64)
- Microkernel with capability-based security model
- Target modern phones and modern PC, not small system with very limited hardware resources (LK, FreeRTOS and ThreadX is suitable for small system)
- Reference:
    - https://fuchsia.googlesource.com/zircon/+/HEAD/docs/

Kernel Objects and Syscalls

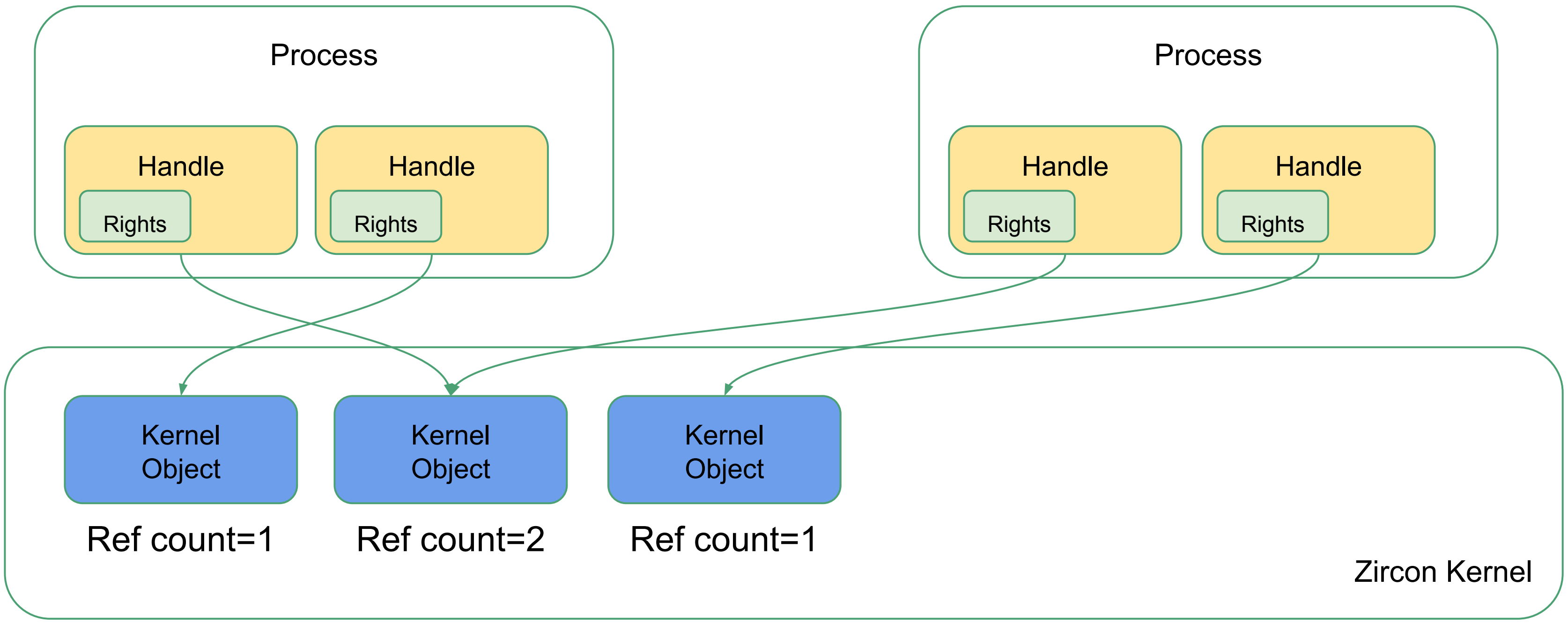
Kernel Objects

- Zircon kernel manages different types of kernel objects
  - **IPC:** Channel, Socket, FIFO
  - **Tasks:** Process, Thread, Job, Task
  - **Signaling:** Event, Event Pair, Futex
  - **Memory:** VMO, VMAR, BTI
  - **Wait:** Port
  - **For drivers:** Interrupt, Resource
- Kernel objects are C++ classes which implement the “Dispatcher” interface
- Userspace code interacts with kernel objects via **system calls** and **handles**
- Dispatcher types
  - **SoloDispatcher:** single endpoint, e.g. Event, Process, Thead, VMO
  - **PeeredDispatcher:** has two endpoints, e.g. Channel, Socket, FIFO, Event Pair

Handles

- Can be thought as a session or connection to a particular kernel object
- Can be bound to **process** or **kernel** (in-transit)
- Each process has its own handle table maintained by kernel
- A kernel object may have multiple handles that refer to it
- When the last open handle that refer to an kernel object is closed
  - The object is destroyed or,
  - The kernel marks the object for garbage collection
- Handle may be transferred to another process by
  - Writing handle into a **Channel** via zx_channel_write()
  - Passing handle as the ‘arg1’ parameter in zx_process_start() to the first thread in a new process
- Handle rights
  - Specify what operations on the kernel object are allowed
  - Different handles to the same kernel object can have different rights

Object, Handle and Rights

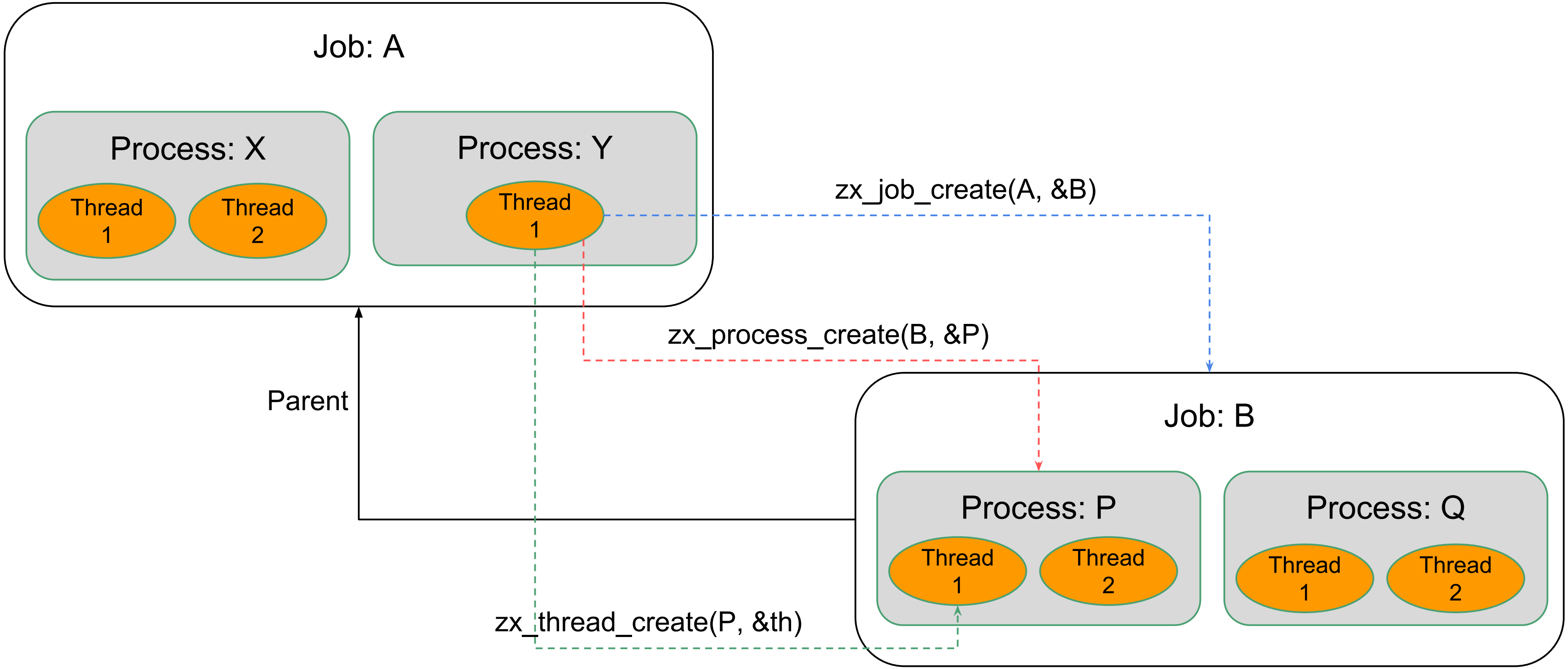


System Calls

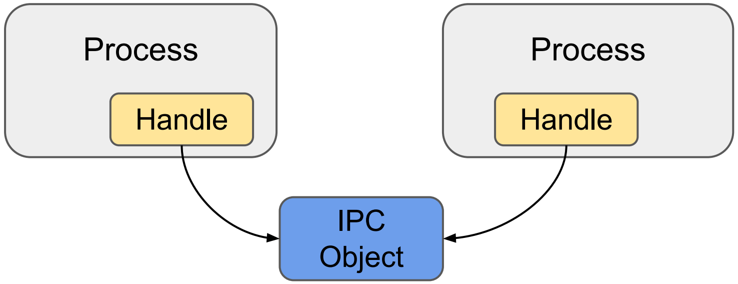
- For userspace code to interact with kernel objects
- System calls fall into three broad categories from an access standpoint:

Access Type	**Permission Check**	**Example**
No limitation	No	zx_clock_get() zx_nanosleep()
Interact with specific object (a handle as first parameter)	• handle is valid • handle is of correct type • handle has required rights	zx_channel_write() zx_port_queue()
Create new object (no handle provided)	Job policy	zx_channel_create() zx_event_create()

Jobs, Processes, and Threads



IPC: Channel, Socket, and FIFO



- Creating an IPC object will return two handles, one referring to each endpoint of the object

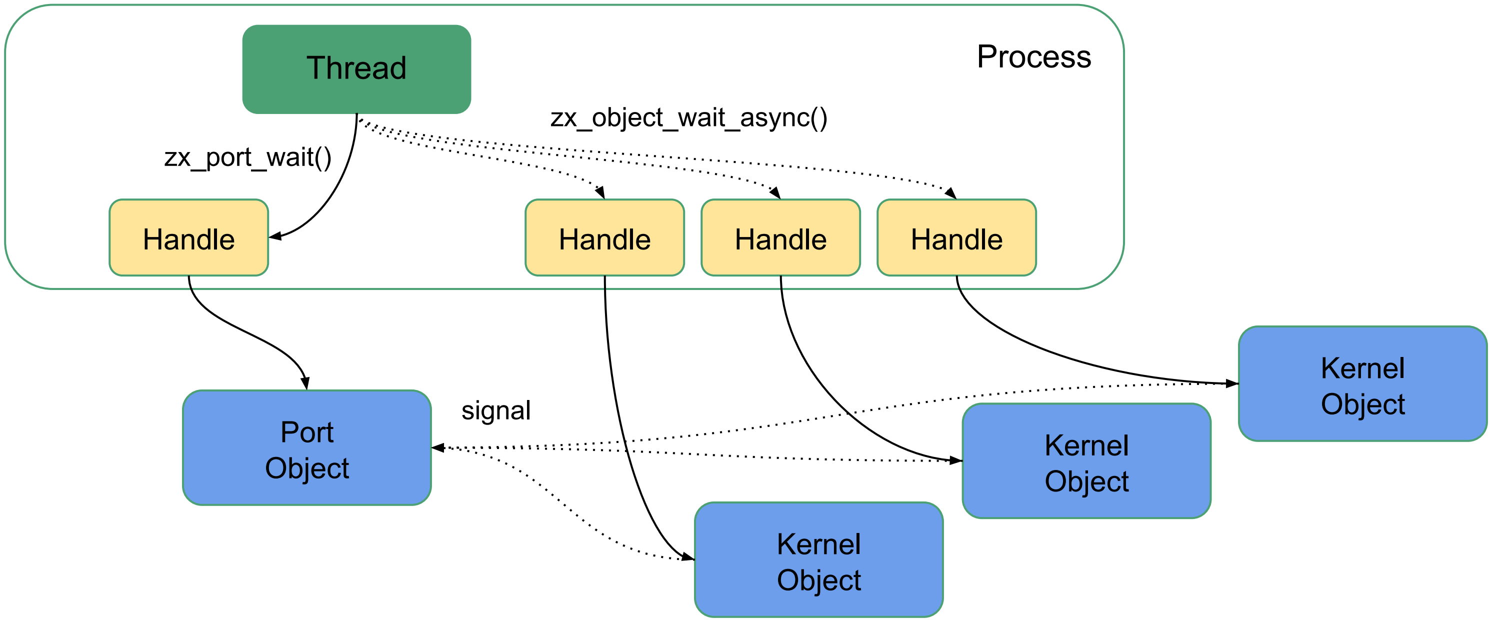
	Transfer Behavior Comparsion	Handle Transfer?	Efficiency	Limitation
Socket	• Stream-oriented • Short read/write are allowed • Data Units: bytes	No	Low	
Channel	• Datagram-oriented • Short read/write are NOT allowed • Data Units: bytes	**Yes**	Mid	Single Write: Max Data Bytes: 64KB Max Handles: 64
FIFO	• Datagram-oriented • Data Units: ***element size*** (assigned when zx_fifo_create())	No	High	Single Write: Max Data Bytes: 4KB

Object Signals and Waiting

- Objects may have up to 32 signals expose to application to represent their current state
  - e.g. ZX_CHANNEL_READABLE, ZX_PROCESS_TERMINATED
- Threads may wait for signals to become active on one or more objects
- Syscalls for waiting signals: (handle should have ZX_RIGHT_WAIT)
  - **zx_object_wait_one():** Blocking wait on a object
  - **zx_object_wait_many():** Blocking wait on one or more objects (up to 8)
  - **zx_object_wait_async() + zx_port_wait():** Blocking wait on one or more objects (> 8)
- User signals
  - **ZX_USER_SIGNAL_0** through **ZX_USER_SIGNAL_7**
  - User signals can be asserted or deasserted by **zx_object_signal()** or **zx_object_signal_peer()**
  - Handle should have ZX_RIGHT_SIGNAL or ZX_RIGHT_SIGNAL_PEER

Port

- If a thread is going to wait on a **large set** of handles, it is more efficient to use a Port
- Multiple signal can be accumulated at port and retrieve at once
- Signal can be queued in port, even no thread is waiting on it



Shared Memory: Virtual Memory Object (VMO)

- Used to represent both paged and physical memory
- The standard method of sharing memory between
  - Processes
  - The kernel and userspace
- **zx_vmo_read()**, **zx_vmo_write():** for userspace to perform basic I/O directly
- **zx_vmo_set_size():** adjust the size of VMO
- **zx_vmar_map():** map VMO in to process address space
- **zx_vmo_op_range():** for some low level operations
  - commit, decommit range of pages
  - cache sync, clean and invalidate
- **zx_vmo_set_cache_policy():** change VMO cache policy

Address Space Management: Virtual Memory Address Region (VMAR)

- Used by the kernel and userspace to represent the allocation of an address space
- Each process starts with a **root VMAR** that spans the entire address space
- Can be logically divided into any number of **non-overlapping** child VMARs
- Have a hierarchical permission model (child ≦ parent)
- All allocations of address space are randomized by default
- Syscalls
  - **zx_vmar_allocate(), zx_vmar_destroy():** create or destroy a VMAR and its child VMARs
  - **zx_vmar_map(), zx_vmar_unmap():** map or unmap a VMO into or from a VMAR
  - **zx_vmar_protect():** adjust memory access permissions

Fast Userspace Mutex (Futex)

- A low level synchronization primitive as a building block for higher level APIs
  - e.g. pthread APIs for mutexes, condition variables
- Zircon futexes are strictly a process local concept and cannot shared across address spaces
- Syscalls
  - **zx_futex_wait():** wait on a futex
  - **zx_futex_wake():** wake some number of threads waiting on a futex
  - **zx_futex_requeue():**
    - wake some number of threads waiting on a futex
    - move more waiters to another wait queue

Resource

- A type of kernel object for controlling access to:
  - Specific range of address space (ZX_RSRC_KIND_MMIO)
  - IRQ numbers (ZX_RSRC_KIND_IRQ)
  - Hypervisor (ZX_RSRC_KIND_HYPERVISOR)
  - SMC (ZX_RSRC_KIND_SMC)
  - ...
- Resource objects are typically private to the DDK and platform bus drivers
- ZX_RSRC_KIND_ROOT resource object is required for creating other kind of resource objects
- Resource allocations can be either shared or exclusive (owned by the holder)

Security Control: Job Policy, Handle Rights and Resources

```c
// Input structure to use with ZX_JOB_POL_BASIC.
typedef struct zx_policy_basic {
    uint32_t condition;
    uint32_t policy;
} zx_policy_basic_t;

// Conditions handled by job policy.
#define ZX_POL_BAD_HANDLE                    0u
#define ZX_POL_WRONG_OBJECT                  1u
#define ZX_POL_VMAR_WX                       2u
#define ZX_POL_NEW_ANY                       3u
#define ZX_POL_NEW_VMO                       4u
#define ZX_POL_NEW_CHANNEL                   5u
#define ZX_POL_NEW_EVENT                     6u
#define ZX_POL_NEW_EVENTPAIR                 7u
#define ZX_POL_NEW_PORT                      8u
#define ZX_POL_NEW_SOCKET                    9u
#define ZX_POL_NEW_FIFO                     10u
#define ZX_POL_NEW_TIMER                    11u
#define ZX_POL_NEW_PROCESS                  12u
#ifdef _KERNEL
#define ZX_POL_MAX                          13u
#endif

// Policy actions.
// ZX_POL_ACTION_ALLOW and ZX_POL_ACTION_DENY can
// ZX_POL_ACTION_KILL implies ZX_POL_ACTION_DENY.
#define ZX_POL_ACTION_ALLOW                 0u
#define ZX_POL_ACTION_DENY                  1u
#define ZX_POL_ACTION_EXCEPTION             2u
#define ZX_POL_ACTION_KILL                  5u
```

```c
#define ZX_RIGHT_NONE               ((zx_rights_t)0u)
#define ZX_RIGHT_DUPLICATE          ((zx_rights_t)1u << 0)
#define ZX_RIGHT_TRANSFER           ((zx_rights_t)1u << 1)
#define ZX_RIGHT_READ               ((zx_rights_t)1u << 2)
#define ZX_RIGHT_WRITE              ((zx_rights_t)1u << 3)
#define ZX_RIGHT_EXECUTE            ((zx_rights_t)1u << 4)
#define ZX_RIGHT_MAP                ((zx_rights_t)1u << 5)
#define ZX_RIGHT_GET_PROPERTY       ((zx_rights_t)1u << 6)
#define ZX_RIGHT_SET_PROPERTY       ((zx_rights_t)1u << 7)
#define ZX_RIGHT_ENUMERATE          ((zx_rights_t)1u << 8)
#define ZX_RIGHT_DESTROY            ((zx_rights_t)1u << 9)
#define ZX_RIGHT_SET_POLICY         ((zx_rights_t)1u << 10)
#define ZX_RIGHT_GET_POLICY         ((zx_rights_t)1u << 11)
#define ZX_RIGHT_SIGNAL             ((zx_rights_t)1u << 12)
#define ZX_RIGHT_SIGNAL_PEER        ((zx_rights_t)1u << 13)
#define ZX_RIGHT_WAIT               ((zx_rights_t)1u << 14)
#define ZX_RIGHT_INSPECT            ((zx_rights_t)1u << 15)
#define ZX_RIGHT_MANAGE_JOB         ((zx_rights_t)1u << 16)
#define ZX_RIGHT_MANAGE_PROCESS     ((zx_rights_t)1u << 17)
#define ZX_RIGHT_MANAGE_THREAD      ((zx_rights_t)1u << 18)
#define ZX_RIGHT_APPLY_PROFILE      ((zx_rights_t)1u << 19)
#define ZX_RIGHT_SAME_RIGHTS        ((zx_rights_t)1u << 31)
```

```c
#define ZX_RSRC_KIND_MMIO           ((zx_rsrc_kind_t)0u)
#define ZX_RSRC_KIND_IRQ            ((zx_rsrc_kind_t)1u)
#define ZX_RSRC_KIND_IOPORT         ((zx_rsrc_kind_t)2u)
#define ZX_RSRC_KIND_HYPERVISOR     ((zx_rsrc_kind_t)3u)
#define ZX_RSRC_KIND_ROOT           ((zx_rsrc_kind_t)4u)
#define ZX_RSRC_KIND_VMEX           ((zx_rsrc_kind_t)5u)
#define ZX_RSRC_KIND_SMC            ((zx_rsrc_kind_t)6u)
#define ZX_RSRC_KIND_COUNT          ((zx_rsrc_kind_t)7u)
```

Virtual Dynamic Shared Object (vDSO)

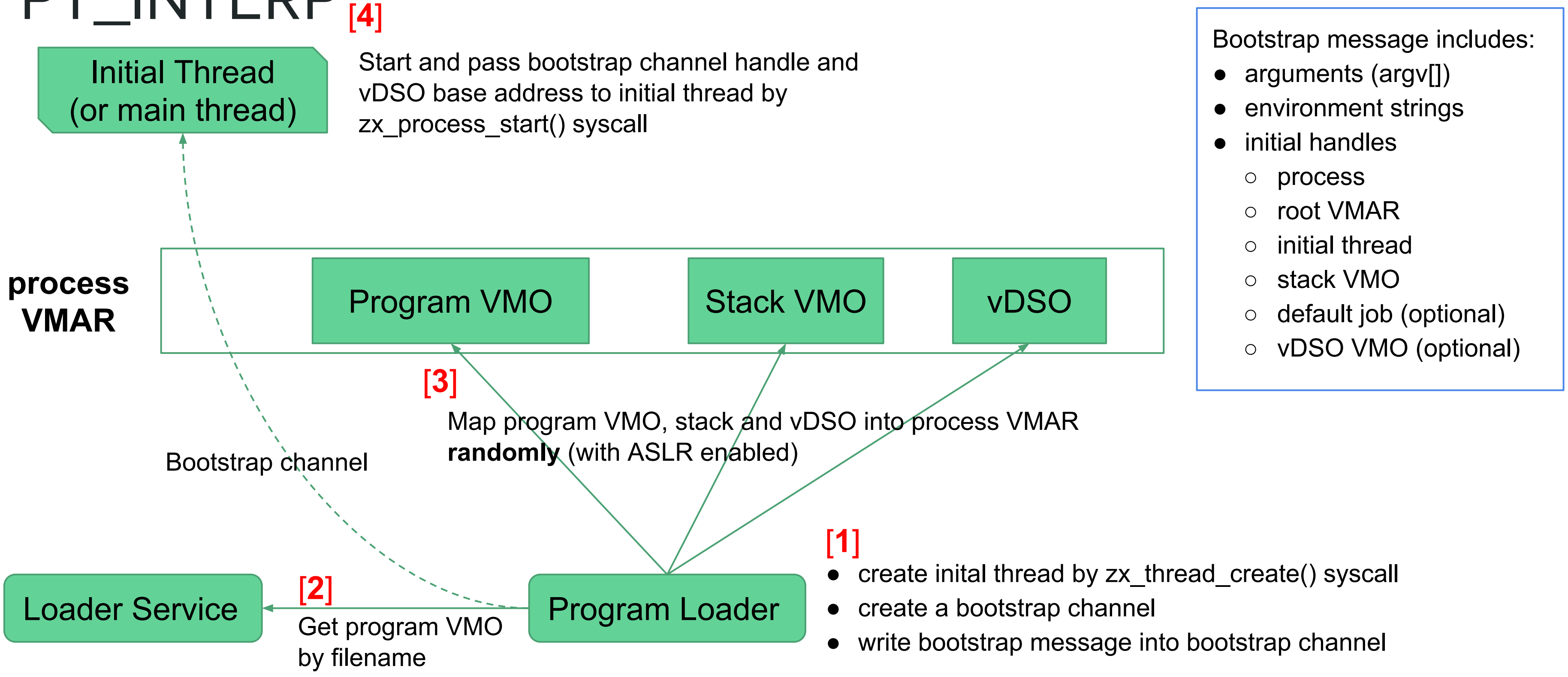
- vDSO is the only way to access to system calls for userspace
- vDSO image (ELF format) is provided directly by the kernel
- The kernel exposes vDSO to userspace as a read-only VMO
- When a new process created by program loader, the program loader:
  - Maps the vDSO into the new process's address space
  - Passes the vDSO base address to the first thread of the new process
  - Passes a vDSO VMO handle to the new process (PA_VMO_VDSO)
- The kernel enforces correct use of the vDSO in two ways:
  - It constrains how the vDSO VMO can be mapped into a process
    - vDSO VMO can only be mapped once in process address space
    - Once the vDSO mapping has been established in a process, it cannot be removed
  - It constrains what PC locations can be used to enter the kernel

Program Loading and Userboot

Program Loading and Dynamic Linking

- The kernel is not directly involved in normal program loading. (Except the userboot process which is the first userspace process booted by the kernel)
- Instead, the kernel merely provides the building blocks (VMO, process, VMAR, thread)
- Userspace environment use ELF format for machine-code executable files and provide a dynamic linker and C/C++ execution environment
- The main implementation of program loading resides in the **launchpad** library
- Program loading is based on VMOs and on IPC protocols used via channels
- Zircon only supports **P**osition-**I**ndependent **E**xecutables (ELF **ET_DYN** files)

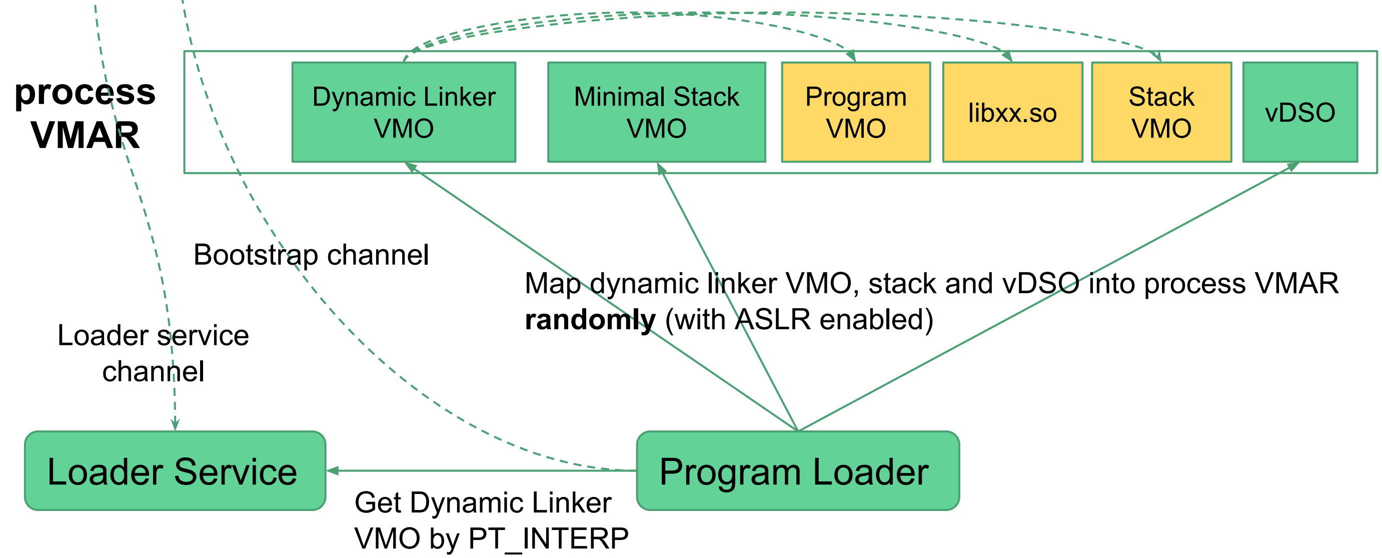
Program Loading: ELF ET_DYN file with no PT_INTERP



Program Loading: ELF ET_DYN file with PT_INTERP

Dynamic linker help to:

- map program VMO
- load and map shared libraries
- prepare stack for main program
- jump to main program (with bootstrap channel and vDSO base address as parameters)



Loader bootstrap message:

- arguments (argv[])
- environment strings
- initial handles
  - main program VMO
  - loader service channel

Bootstrap message:

- arguments (argv[])
- environment strings
- initial handles
  - process
  - root VMAR
  - initial thread
  - stack VMO
  - default job (optional)
  - vDSO VMO (optional)

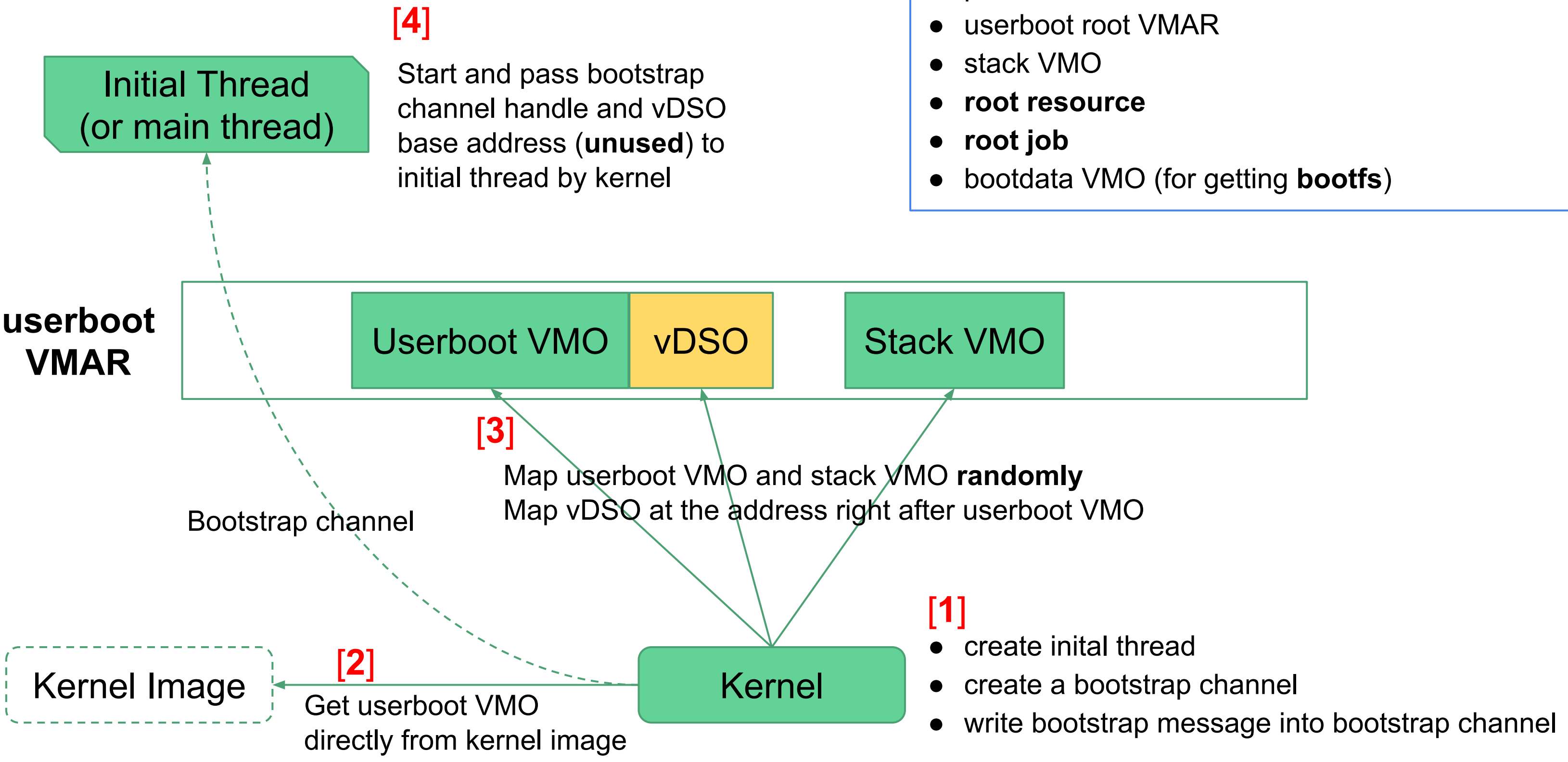
Kernel to Userspace Bootstrapping

- userboot is the first userspace process started by the kernel
- userboot is built as an ELF dynamic shared object which embedded in the kernel image
- Due to the simple layout of userboot image, the kernel does not need to interpret its ELF header.
  - Information for loading userboot image will be extracted in compile time
  - Information including read-only segment size, executable segment size, entry point address
- In conclusion, the kernel starts the first user process **without** implementing:
  - ELF parser
  - Minimal file system
  - Decompression library

Userboot Startup

The kernel transfers handles to userboot including:

- vDSO VMO
- process handle and thread handle of userboot itself
- userboot root VMAR
- stack VMO
- **root resource**
- **root job**
- bootdata VMO (for getting **bootfs**)



Scheduling

Scheduling

- Each CPU has its own set of priority queues levels from 0 to 31 (highest)
- Thread time slice rule:
  - If run out of its time slice, thread will be preempted and reinserted at the **end** of the appropriate priority queue
  - If still has time slice remaining, thread will be inserted at the **head** of the priority queue so it will be able to resume as quickly as possible
- Three different factors used to determine the effective priority of a thread:
  - **Base priority**: set at thread creation time or by thread_set_priority()
  - **Priority boost**: used to offset the base priority
  - **Inherited priority**: inherit from another higher priority thread which is blocked by the thread
- Effective_priority = MAX(base_priority + priority_boost, inherited_priority)
- The intention is to ensure that **interactive threads are serviced quickly**

Realtime and Idle Threads

- Idle threads
  - Runs when no other threads are runnable
  - One on each CPU and lives outside of the priority queues
  - Used to track idle time for platform to implement the low power wait mode
- Realtime threads
  - Marked with THREAD_FLAG_REAL_TIME
  - Allowed to run **without preemption** and will run until they block, yield, or manually reschedule

Backup

Out-Of-Memory System (OOM)

- A kernel OOM thread is responsible for monitoring free memory size periodically
- OOM starts to work if the amount of free memory is lower than ‘redline’
- OOM will pick and kill a job which is marked as ‘kill_on_oom’
- OOM-ranker driver is under development

```
$ k oom info
[01725.664] 01100.01113> OOM info:
[01725.664] 01100.01113>   running: true
[01725.664] 01100.01113>   printing: false
[01725.664] 01100.01113>   simulating lowmem: false
[01725.664] 01100.01113>   sleep duration: 1000ms
[01725.664] 01100.01113>   redline: 50M (52428800 bytes)
```