Embedded Linux Device Drivers

OS Booting

- Power ON
- Firmware program --> RAM
- POST -- check all peripherals
- Bootstrap loader -- find bootable device/partition
- start bootloader -- get OS to boot from user (if multiple)
- start bootstrap program
- load kernel

Linux Booting (on PC -- GrUB)

- Power ON.
- Firmware programs loaded into RAM.
- Run POST -- check all peripherals.
- Run Bootstrap Loader -- Find the bootable device (Linux GrUB loader).
- Run GrUB stage 1.
    - First 512 bytes of the device.
    - Transfer control to GrUB stage 1.5.
- Run GrUB stage 1.5.
    - Between boot sector of the device and the First partition of the disk.
    - Have Linux FS driver (to read files from Linux partition).
    - Transfer control to GrUB stage 2.
- Run GrUB stage 2.
    - On Linux boot partition.
    - Reads /boot/grub/grub.cfg file.

```
menuentry 'Ubuntu' --class ubuntu --class os {
  linux /boot/vmlinuz-5.15.0-76-generic root=/dev/sda1 ro
    initrd  /boot/initrd.img-5.15.0-76-generic
}
```

  - Present options to end user.
- End user selects Linux kernel and Initial RAM fs to boot from and GrUB loads them into RAM.
- The kernel get self-extracted and execution of the kernel begins.
- In initrd (initial ram disk -- loaded into the RAM by GrUB) serves as temporary filesystem for Linux kernel until root filesystem from disk is available. It contains device drivers for minimal required devices like disk, ...
- Kernel use initrd drivers to access "root filesystem" and then access all files/drivers there. Now kernel starts first user space process i.e. init/systemd.
- The init/systemd process start the kernel booting sequence. Kernel booting is logically divided into runlevels/init levels/systemd targets.
  - 1 (rescue.target) - Single user (root login)
  - 2 (multi-user.target) - Multi user (no network)
  - 3 (multi-user.target) - Network (no GUI)
  - 4 - Reserved
  - 5 (graphical.target) - GUI

Embedded Linux

- Prime components of Embedded Linux
  - Embedded Hardware
  - Bootloader
  - Kernel
  - Root Filesystem/Busybox
  - Application/Drivers
- Steps to build Embedded Linux 0. Host system (PC) with Ubuntu/Debian
  1. Install toolchain and necessary tools
  2. Download the source code of Linux Kernel, Bootloader (u-Boot), BusyBox
  3. Build Bootloader
  4. Build Linux Kernel

5. Build RootFS with BusyBox
6. Prepare Boot media (SD-Card BOOT and ROOT partition)
7. Boot the Embedded system (board)

Yocto

- Refer yocto notes

Proc File System

- "procfs" is a pseudo file system i.e. no persistent storage is associated with the file system.
- It is considered as "window" to the kernel i.e. it is used to monior activities and data structures inside the kernel.
- Typically they are used to make kernel space information available to user space application in most simplified way. The device drivers/kernel components use them as debugging feature.
- Few important proc files:
  - /proc/cpuinfo
  - /proc/meminfo, /proc/buddyinfo, /proc/slabinfo
  - /proc/interrupts
  - /proc/devices
  - /proc/timer_list
  - /proc//maps, /proc//pagemap, ...
- Some of the proc files are also used to set kernel config.
  - /proc/sys/kernel/sched_child_runs_first
  - /proc/sys/vm/overcommit_memory
  - /proc/sys/kernel/sched_rr_timeslice_ms
- In older kernels, proc files are implemented as kernel buffer; however programmer was responsible to keep track of page sized buffers. In other words, the output of proc should be restricted within a single page (4 kb) or programmer should keep track of page buffers in order to generate larger output.
- Newer kernel versions, has simplified access to the proc files using concept of "sequence files". The sequence files present a particular sequence of data to the user. This mechanism frees programmer from tracking page buffer size. Now programmer is only responsible for iterating some data structure and displaying its contents.
- Sequence files are represented with "struct seq_file". Any data written into this file (using helper functions seq_printf(), seq_putc() and seq_puts()) will be visible to the end user as proc file contents.

Implementing Sequence files

- step 1: Create entry in proc file system using kernel api proc_create(). This takes filename as an arg & also file_operations/proc_ops struct as another arg. The file_operations/proc_ops struct stores predefined kernel functions i.e. seq_read(), seq_lseek() & seq_release(). However, open operation is replaced by an user-defined function.
- step 2: The user-defined open() operation, simply calls pre-defined seq_open() with sequence file operations struct seq_operations
- step 3. Programmer should define these seq_operations i.e. .start, .stop, .next, .show.
- step 4: Implement .start operation:
  - Called when seq file is opened.
  - args: struct seq_file *s, loff_t *pos
    - 1st arg: is rarely needed into this operation.
    - 2nd arg: "pos" is programmer defined position (not byte based position). For just opened file, it is 0.
  - Programmer is expected to return the first object to be displayed from this method (return type is void*).
- step 5: Implement .next operation:
  - Called when progressing seq file reading.
  - args: struct seq_file *s, void *v, loff_t *pos
    - 1st arg: is rarely needed into this operation.
    - 2nd arg: current displayed object. Programmer can use it to go to next object in list.
    - 3rd arg: "pos" must be incremenet in (programmer-defined way).
  - Programmer is expected to return the object to be displayed from this method. If end of data list is reached, return NULL.
- step 6. Implement .stop operation:
  - Called when sequence ends i.e. when next() operation returns NULL.
  - It is used to release resources, if any resource is allocated in start operation.
  - Immediately after .stop(), once again .start operation is invoked (with current pos i.e. pos!=0). In that case start operation is expected to return NULL.
- step 7. Implement .show operation:
  - Called to display current object after non-NULL return from start/next operation.
  - args: struct seq_file *s, void *v
    - arg1: seq_file -> to display, contents are added in this file using seq_printf(), etc.
    - arg2: object returned from start/next operation. Use this object to get contents to be displayed.
- step 8. At the end (typically in module_exit()), remove proc file entry using kernel api remove_proc_entry().

Debugging Techniques

Kernel panics

- When kernel panics, system hangs and reboots.
- The panic log is added into /var/crash directory, if configured in sysctl.
- step 1: add following lines in /etc/sysctl.conf

```
kernel.core_pattern = /var/crash/core.%t.%p
kernel.panic=10
kernel.unknown_nmi_panic=1
kernel.sysrq=1
```

- step 2: Execute following command after changes done in sysctl.conf.

> sysctl -p

- step 3: Generate manual dump using SysRq key (Alt + SysRq + C). On laptops, SysRq key might be PrtSc.
- step 4: After reboot analyze dump from /var/crash.

> apport-unpack /var/crash/_usr_libexec_fwupd_fwupd.0.crash panic

> cd panic

> ls panic

  - Now refer the files from this directory to know more about kernel panic.
- Reference: https://unix.stackexchange.com/questions/194171/kernel-panic-dumps-no-log-files
- For user application core dumps (segmentation fault), in Ubuntu it is sent to apport program. By default, it doesn't log it if it is unpackaged program.

- To enable core dump and analyze it refer below link.
- Reference: https://askubuntu.com/questions/1349047/where-do-i-find-core-dump-files-and-how-do-i-view-and-analyze-the-backtrace-st/1442665#1442665

Memory management

User space memory management

- Memory mangement schemes
  - Contiguous allocation
  - Segmentation
  - Paging -- Followed in Linux
- Linux Paging internals
  - Each process (32-bit) has virtual address space -- 4 GB.
  - struct task_struct
    - struct mm_struct
      - *pgd --> pointer to page directory (primary page table)
      - *mmap --> pointer to VAD (vm_area_struct) list.

Kernel Memory management

Buddy Allocator

- Physical RAM is divided into Zones like ZONE_DMA, ZONE_NORMAL, ZONE_HIGHMEM (depends on architecture).
- Each Zone free memory is tracked and allocated by a Buddy allocator.
- Buddy allocator is used to allocate large contiguous memory blocks. It allocates pages in order of 2 only.
- It maintains info about free blocks on free list for each order of 2.
- While allocation if free block is available (on corresponding free list), it is allocated from there.
- If not available, it will search higher order block and split into 2 parts -- called buddies. The process continues until desired block is found and that block is allocated (i.e. removed from the free list).
- When any block is released, it's info again added into respective free list. If it's buddy is available on the list, they will be merged together to build bigger block. This is called as coalescing.

- However, if immediately next request for smaller block arrives, then need split higher order blocks again. This is time consuming.
- To improve performance of small block allocations, released blocks are not immediately merged with their buddies. The merging is done only if required contiguous memory is not available while allocating higher order block. This is called as lazy coalescing.
- API (LKD):
  - ptr = (type-cast) __get_free_pages(gfp_mask, order);
  - free_pages(ptr);
- gfp_mask -- zone modifier | action modifier
  - GFP_KERNEL = **GFP_IO | **GFP_FS | __GFP_WAIT

Slab allocator

- To allocate smaller objects buddy allocator should not be used. Because it allocates minimum memory of 1 page (i.e. 4 KB). If used for smaller objects, rest of allocated memory will be wasted. This is "internal fragmentation".
- For smaller objects memory allocation, Linux kernel have slab allocator (or slub allocator).
- Slab allocator maintains caches of pre-defined objects. When kmalloc() is called, depending on size, the object is allocated from the corresponding cache.
  - sudo cat /proc/slabinfo
- Slab caches are made up of slabs. Few slabs can be full, partial, or empty.
- Each slab is made up of number of pages in consecutive memory. Slab allocator internally calls buddy allocator to allocate each slab.
- All objects in the cache are in pre-initialized state (when cache is created).
- When a new object is requested, pre-allocated but unused object address is returned.
- When the object is released, that object is marked free/unused into the slab.
- If all slabs are full, then new slabs will be allocated and added into the cache.
- In case of memory shortage, empty slabs are released.
- APIs:
  - ptr = (type-cast)kmalloc(size, gfp_mask);
  - kfree(ptr);

vmalloc()

- Buddy allocator allocates physical pages and kmalloc() allocates smaller objects.
- To allocate virtually contigous memory block use vmalloc().
- vmalloc() allocate contigous range of virtual addresses for the current/calling process. It will also create necessary page table entries.

- It doesn't allocate the physical pages. The physical pages will be allocated when pages are accessed (during page fault).
- vmalloc() is internally called when user space process calls mmap().
- These addresses must be released using vfree().