Raspberry Pi

OS Port Project

梁远志 15231099

Lab1 Summary

- UART Character Output
- Link File Configuration
- Firmware

UART

- A poll driver implementation

- Implement two interface:
- `u32 uart_recv();`
- `void uart_send(u32 c);`

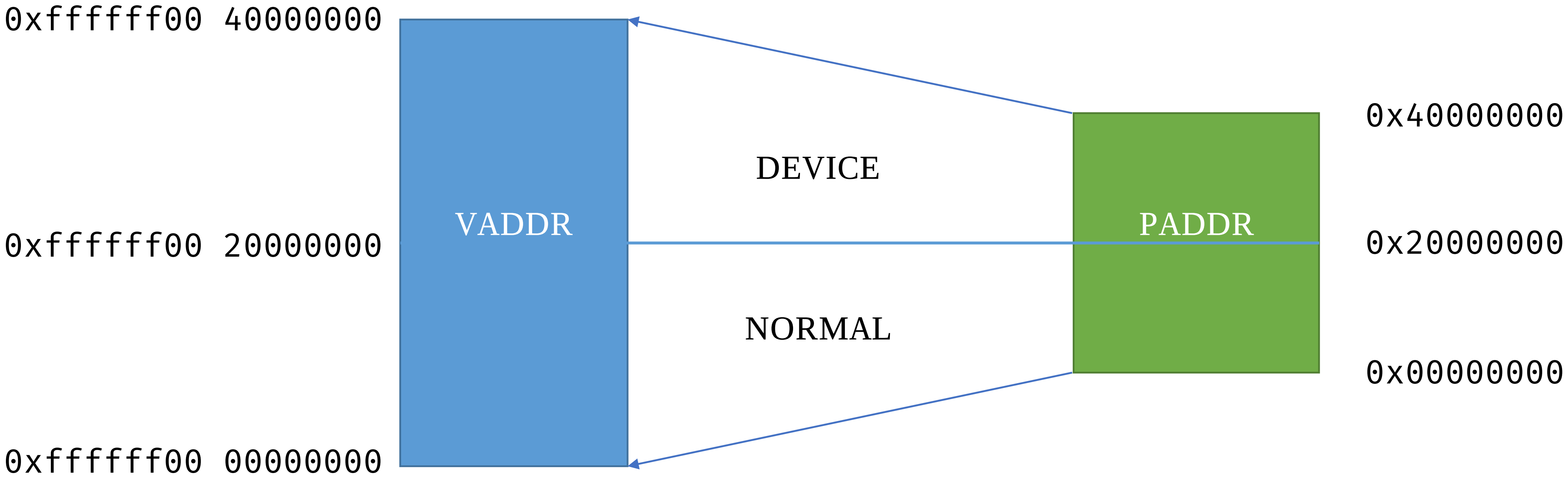
Link File & Firmware

- Kernel start at physical address:
  - `0x00080000`
- Configure the kernel to run at Aarch64 mode:
  - `arm_control=0x200`
- Dump the ELF kernel image to a
  - `kernel8.img`

Lab2 Summary

- Initialize MMU
- Initialize Page Management

MMU



- Reference: Yradex/RaspberryPi3_OS

MMU

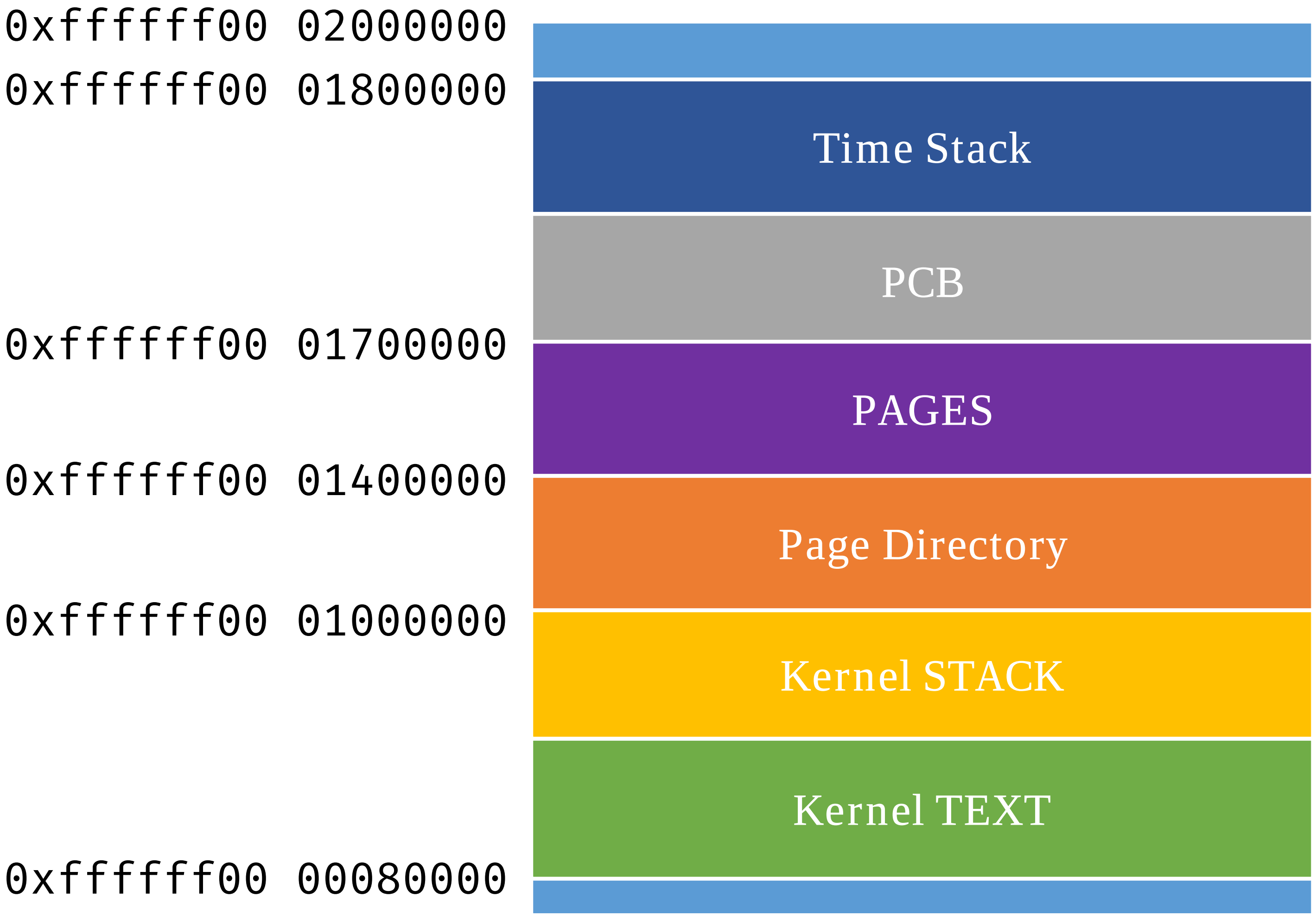
- `// Use 40bit virtual address (4kB page and 4 levels page table)`
- `    // [   2   |   9   |   9   |   9   |   12   ]`
- `    //    Pge     Pde     Pme     Pte     Page`

MMU

1. Initialize exception level (EL1)
2. Initialize the page directory to map those address
3. Turn on MMU
4. Jump to high address kernel

Page Management

- Reserve space for kernel



0xffffff00 02000000

0xffffff00 01800000

Time Stack

PCB

0xffffff00 01700000

PAGES

0xffffff00 01400000

Page Directory

0xffffff00 01000000

Kernel STACK

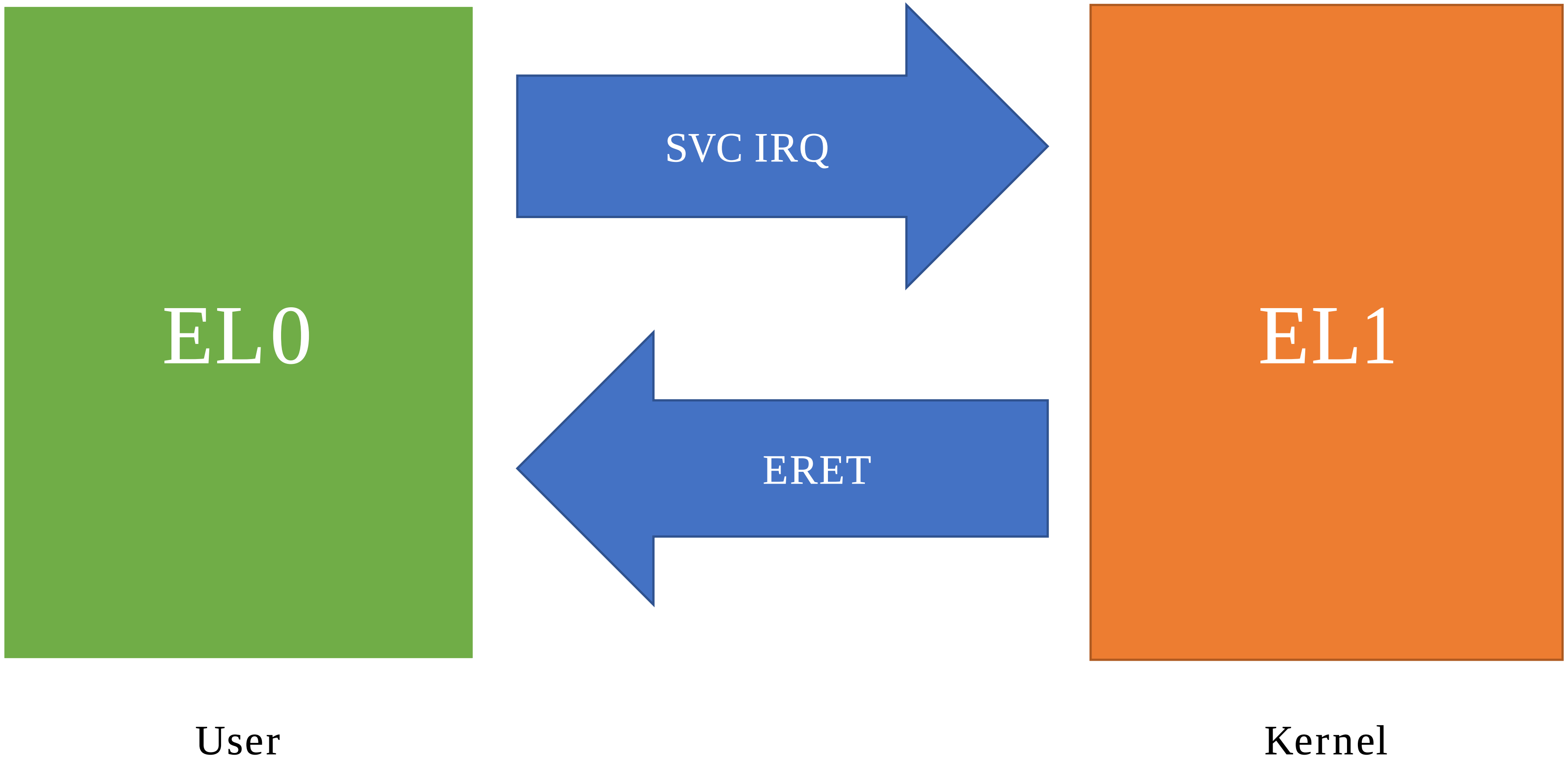
Kernel TEXT

0xffffff00 00080000

Lab3 Summary

- Exception Model
- Clock Interrupt
- System Call (put character)
- Context Switch
- EMMC Driver

Exception Model



EL0

SVC IRQ

ERET

EL1

User

Kernel

Clock Interrupt

- Interrupt driven driver

- Implement interface:
- `int usleep(useconds_t usec);`
- `void setup_clock_int(u32);`
- `void clear_clock_int();`

System Call

- User

- svc #0
- ret

- Kernel

- Push Time Stack
- Resume Stack
- bl uart_send
- Pop Time Stack
- eret

Context Switch

- Backup trap frame
- Switch low address page directory register `TTBR0`
- Flush TLB (all?)
- Restore trap frame

EMMC Driver

- To load user program (ELF image):
  - Convert to a huge array
  - Read from block device (EMMC)

EMMC Driver

- A poll implemented driver
- Read Only (Write at risk)
- Implemented for future(Lab5)

- Provides interface:
- `int emmc_read_sector(u32 secno, void *buf);`

EMMC Driver

- Write image to SD card:
- `dd if=[elf image] of=/dev/sd[x] seek=[sector] bs=512`
- Load image in kernel:
  - `load_program(u32 sector)`
  - `env_create(void *buf, u32 size)`

Lab4 Summary

- All System Call
- A fork implementation
- IPC

All System Call

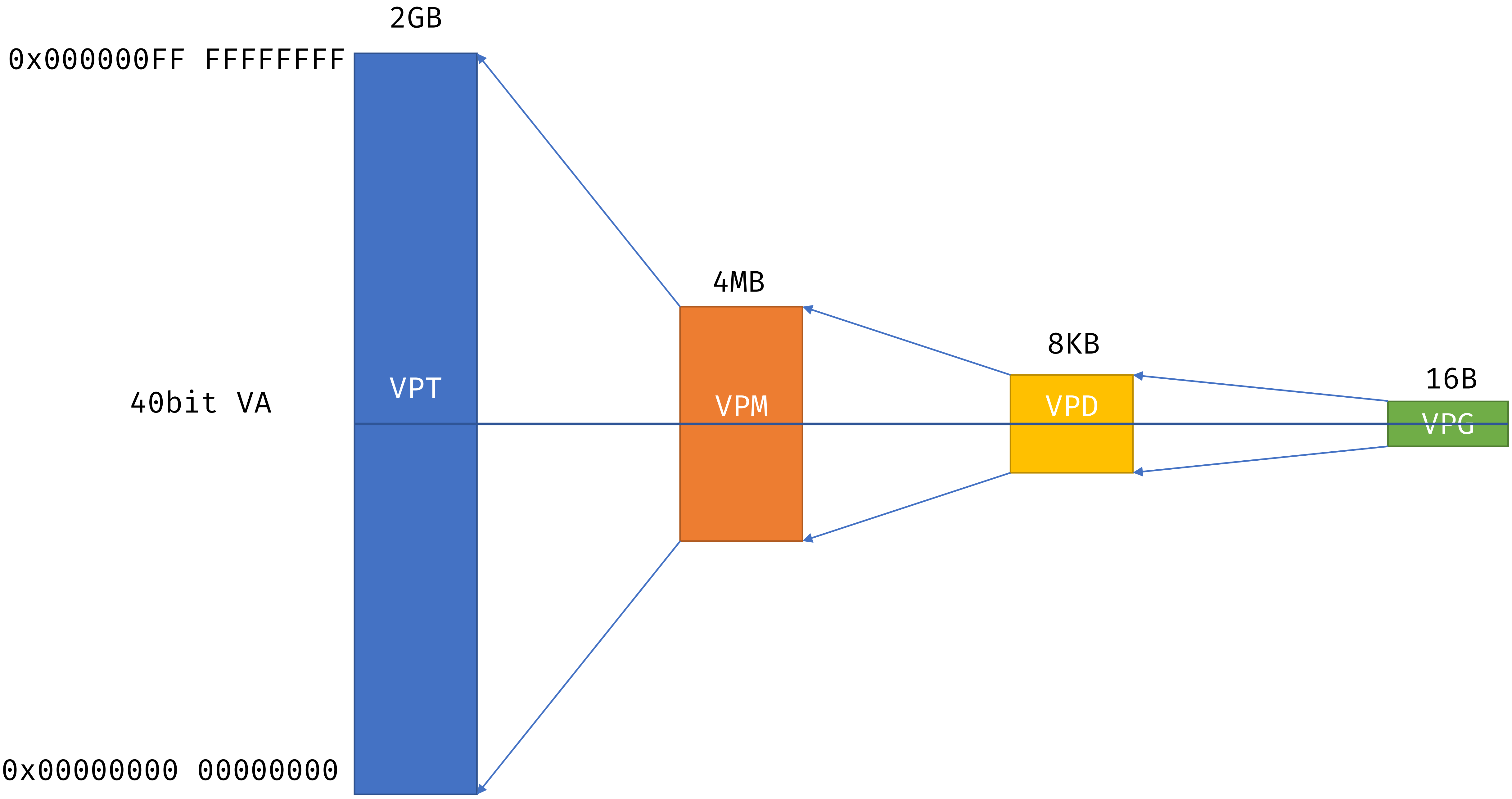
- The form of system call remained
- Use ARMv8 ABI

A fork implementation

- User Fork with Copy on Write?
- Emmmmm, I can’t implement it.

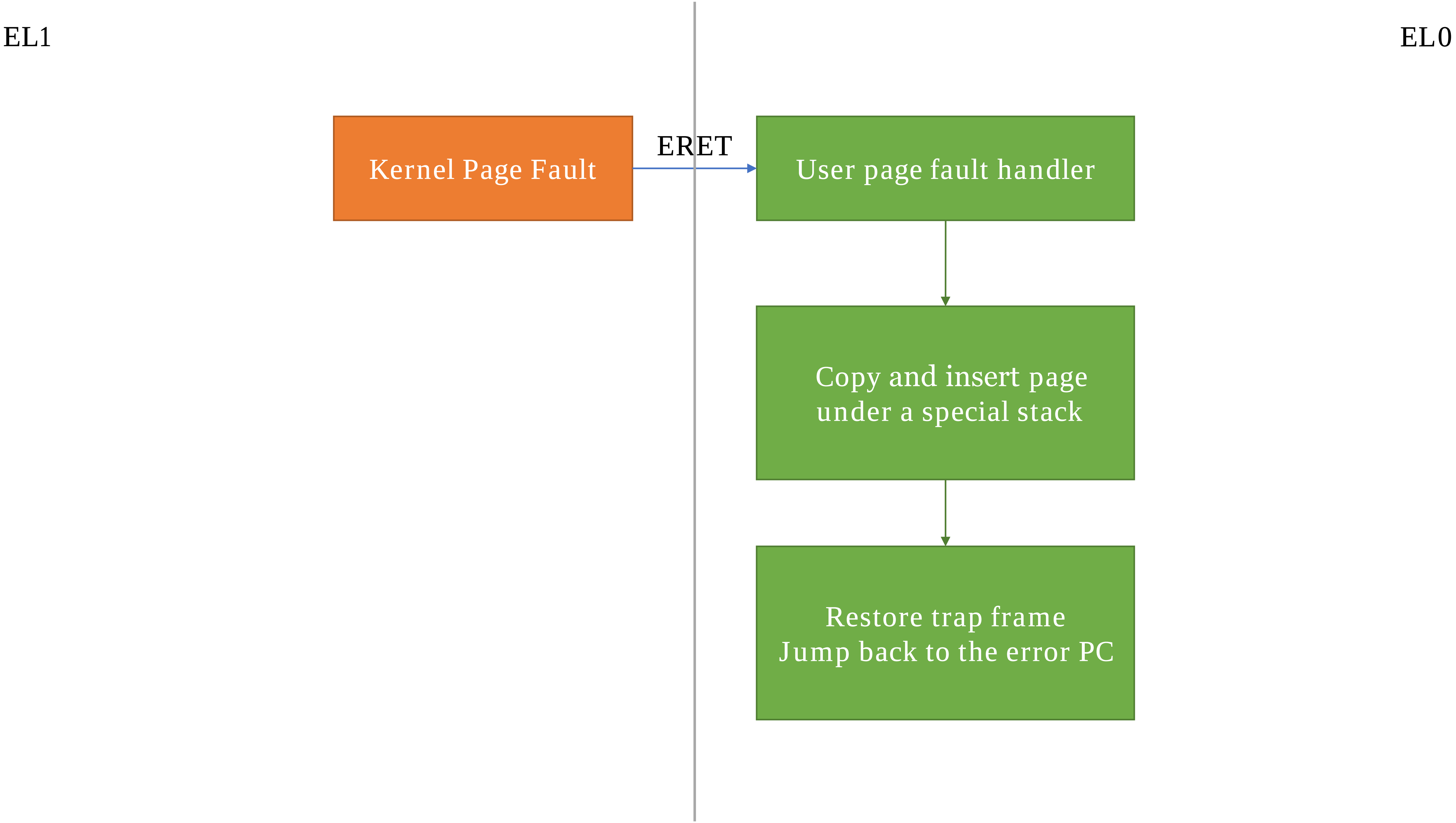
A fork implementation

- A self-mapped page table?


2GB
0x000000FF FFFFFFFF
40bit VA
VPT
0x00000000 00000000
4MB
VPM
8KB
VPD
16B
VPG

A fork implementation

- Deal with page fault (COW) under user mode?



EL1

EL0

Kernel Page Fault

ERET

User page fault handler

Copy and insert page
under a special stack

Restore trap frame
Jump back to the error PC

A fork implementation

- For those difficulties~~ and limited time~~,
- I just implement a kernel fork with no COW
- Add a system call `syscall_pgtable_entry(u64 va)`

IPC

- `syscall_yield` cause error
- Stack behavior of System call
- Do context switching under kernel stack

Lab5 Summary

- Use EMMC Driver
- Create a Disk Image
- Some bugs

Use EMMC Driver

- Implement a system call to read a block (8 sectors)
  - `syscall_emmc_read(u32 sector, u64 va)`
- Disable dirty write back
- Use system call instead of `ide_read` in function `read_block`

Create a Disk Image

- Aarch64 is BIG ENDIAN by default
- Remove the conversation in `fsformat` to generate a disk image
- Write disk image to SD card (with a specified sector)
  - `dd if=fs.img of=/dev/sd[x] seek=[sector] bs=512`

Some bugs

- A system call with context switch (IPC and etc.) my cause the stack behavior abnormal
- I just add some dummy (a system call with no effect) to deal with it

Thanks