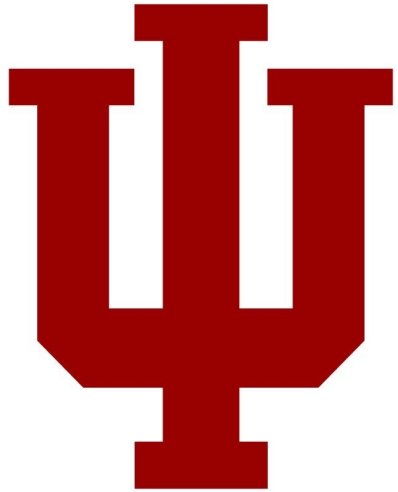
12: Linux MMIO

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University



Some material taken from:
EECS 373 & EECS 370 University of Michigan
https://developer.arm.com/documentation/102202/0300/Transfer-behavior-and-transaction-ordering

Announcements

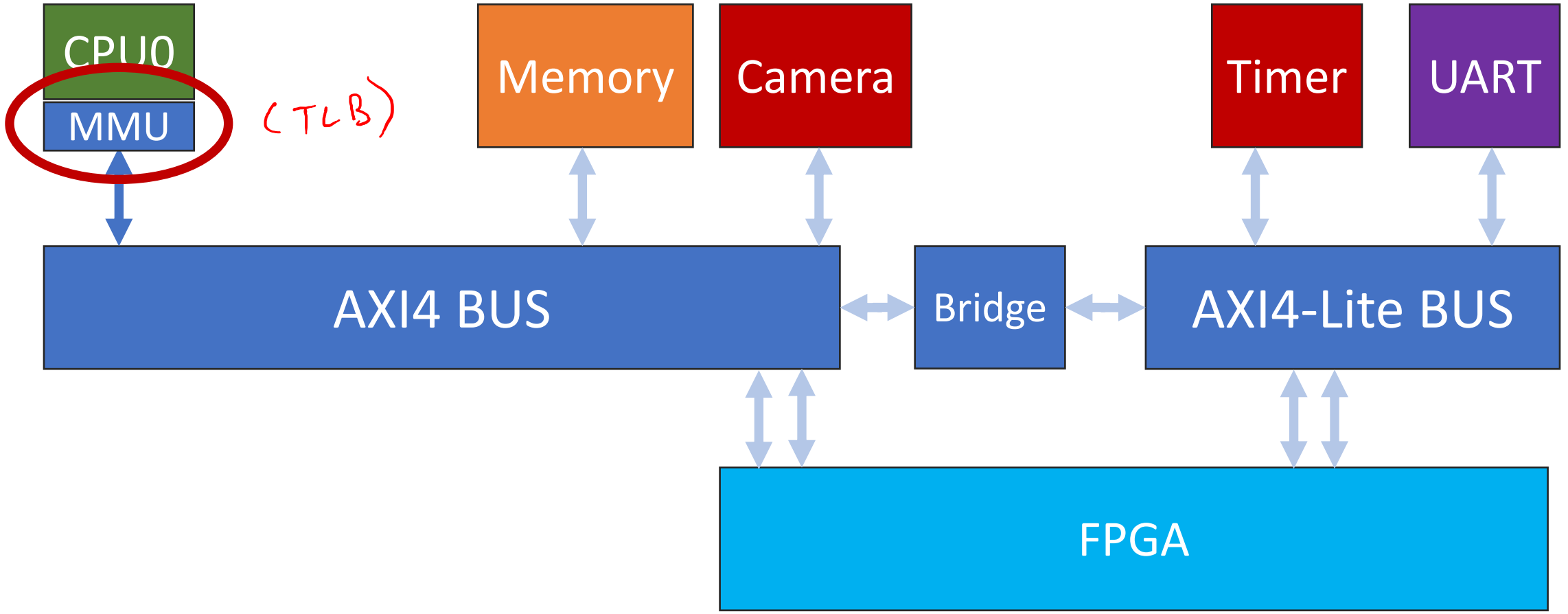
- P4: Due today!
  - verilog/vtests/axi4lite_synth/axi4lite_synth_tb.sv runs on AG.
- P5: Out tonight.

Exam Planning

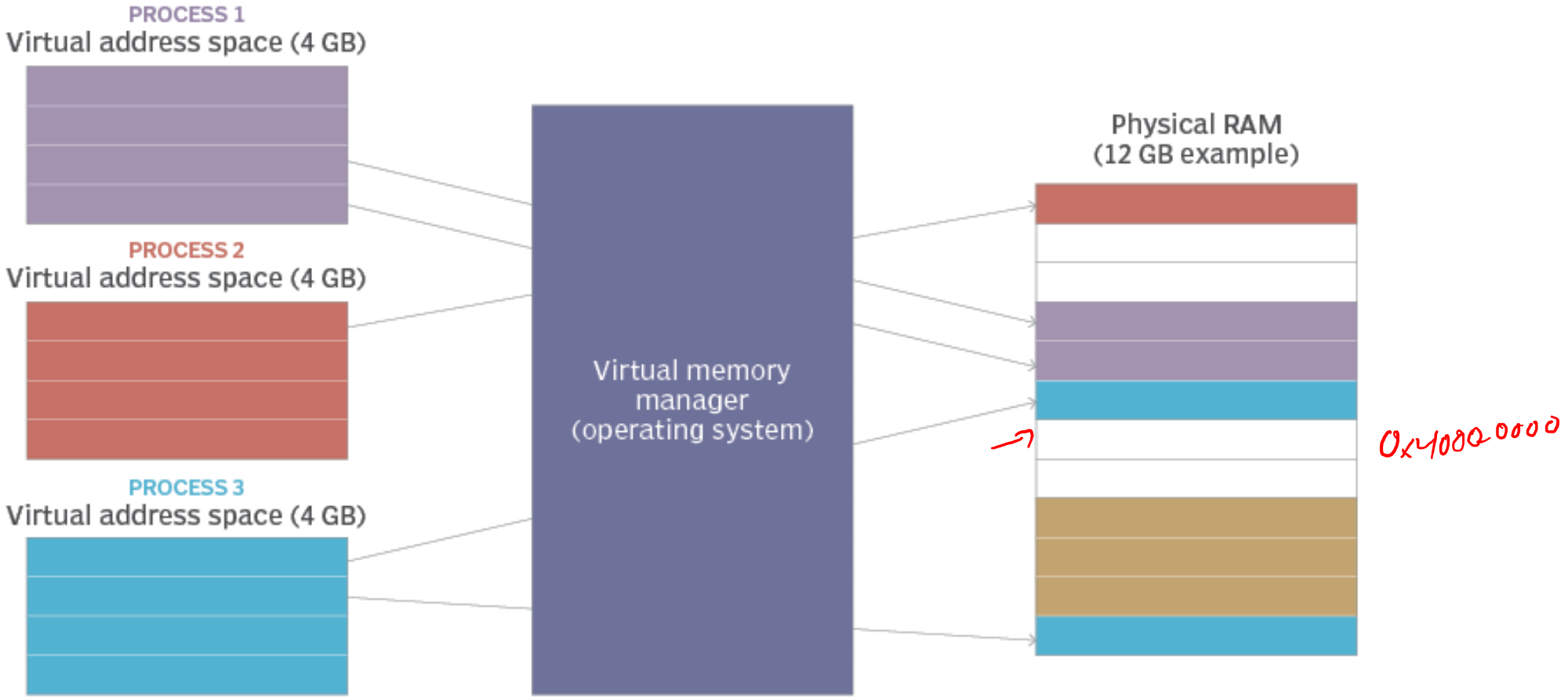
11/01	Wednesday	18	Review	–
11/06	Monday	19	Exam	

https://engr315.github.io

Machine Model, V3: MMUs



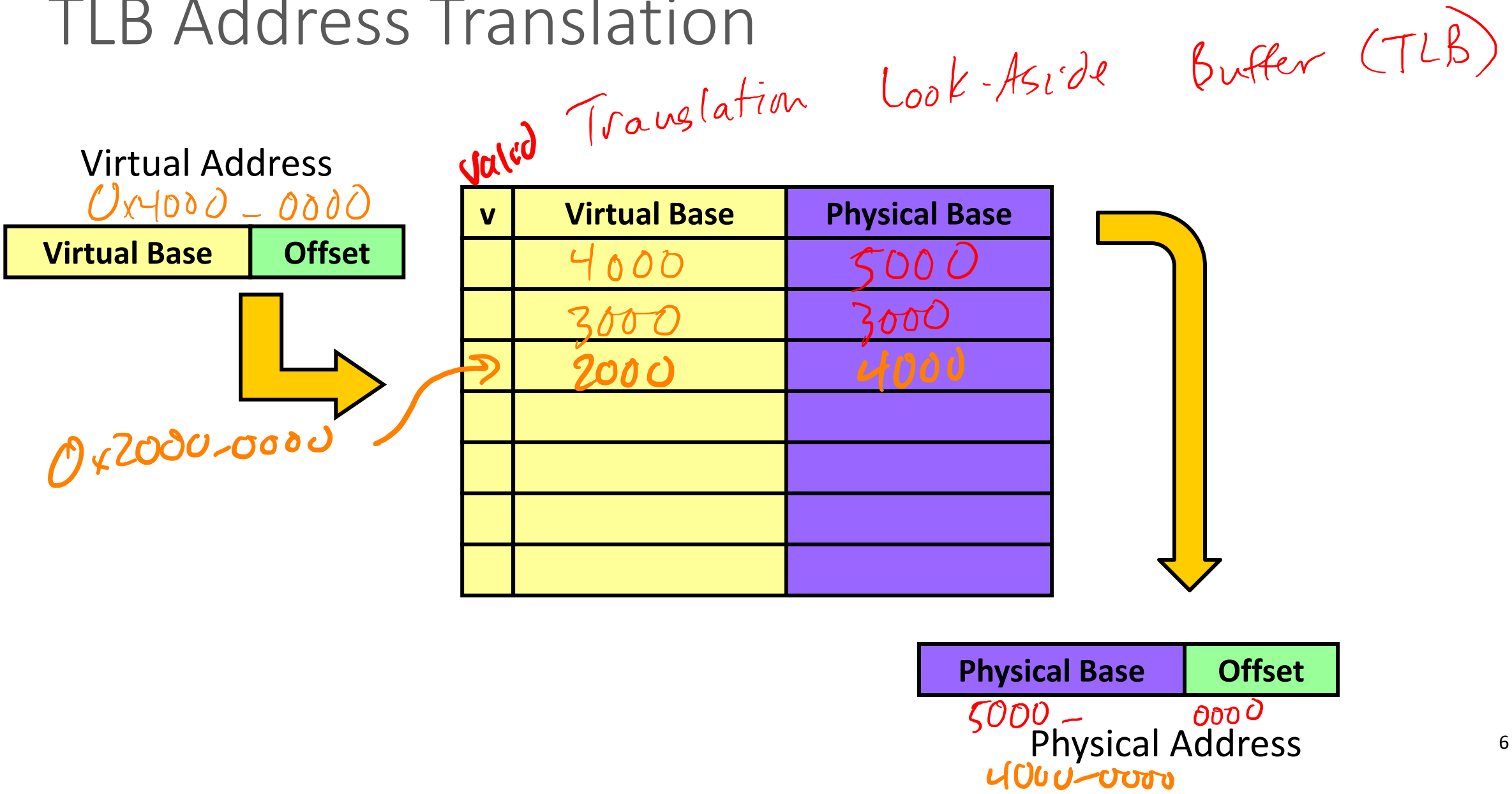
OS (Linux) mains full Virtual->Physical Mappings



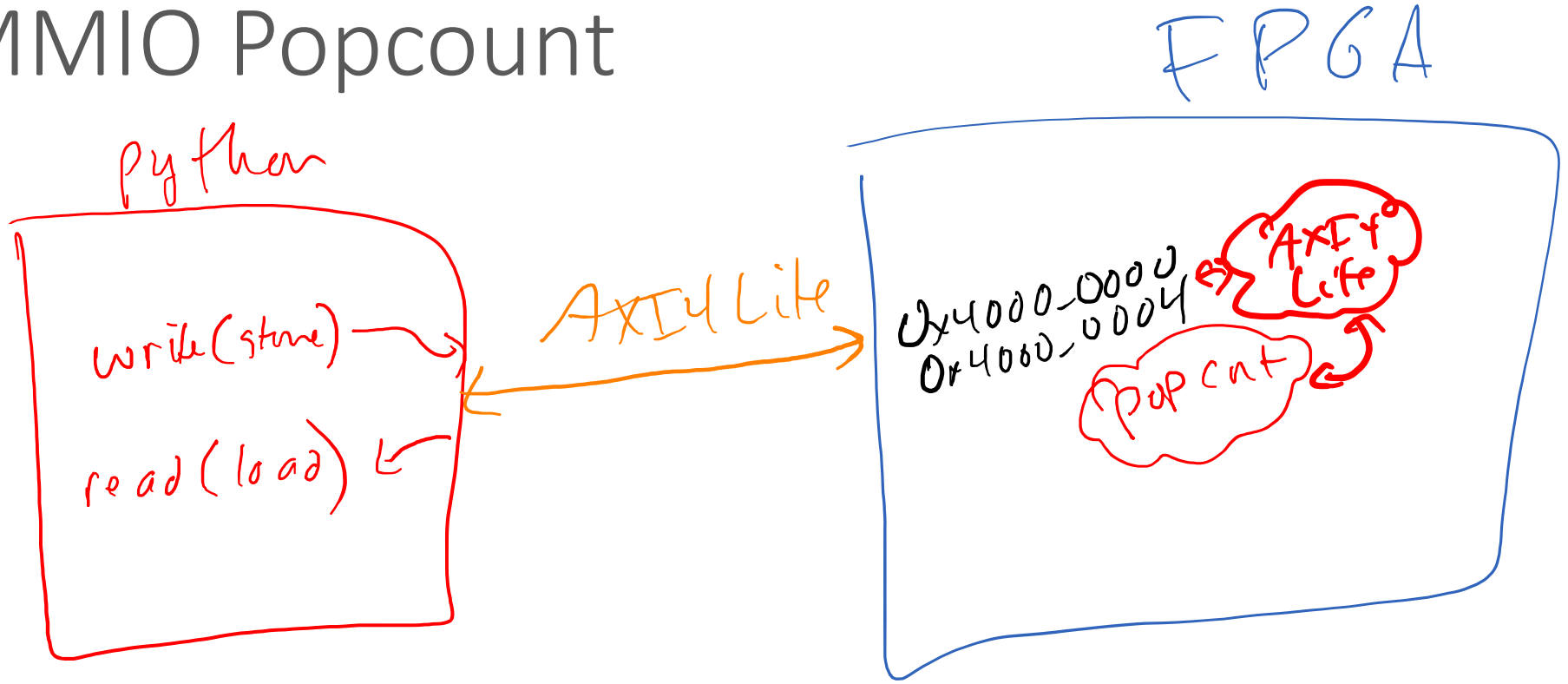
https://www.techtarget.com/searchstorage/definition/virtual-memory

TLB Address Translation

Translation Look-Aside Buffer (TLB)



P4: MMIO Popcount



Vivado MMIO EMA

```
module ema(
        input                   ACLK,
        input                   ARESETN,
        input [31:0]            WRITE_DATA,
        input                   WRITE_VALID,
        output logic [31:0] READ_DATA,
        input                   READ_VALID
    );
    logic [31:0] y_last;
    logic [31:0] y_hist;
    assign READ_DATA = (y_last >> 2) +
            (y_hist>> 1) + (y_hist >> 2);
    always@(posedge ACLK) begin
        if (~ARESETN) begin
            y_last <= 32'h0;
            y_hist <= 32'h0;
        end else if (WRITE_VALID)
        begin
            y_last <= WRITE_DATA;
            y_hist <= READ_DATA;
        end
 end
endmodule
```

Vivado MMIO EMA

```
ema ema0(
        .ACLK(S_AXI_LITE_ACLK),
        .ARESETN(S_AXI_LITE_ARESETN),

        .WRITE_DATA(WRITE_MEM[0]),
        .WRITE_VALID(WRITE_MEM_VALID[0]),

        .READ_DATA(READ_MEM[0]),
        .READ_VALID(READ_MEM_VALID[0])
    );
```

high for 1 cycle to indicate new data

high for 1 cycle to indicate read.

Question:

- How do we access the FPGA from C w/Linux?

- FPGA uses physical address
- CPU (w/Linux) uses virtual address

Answer:

- Linux lets us cheat.

Python Example

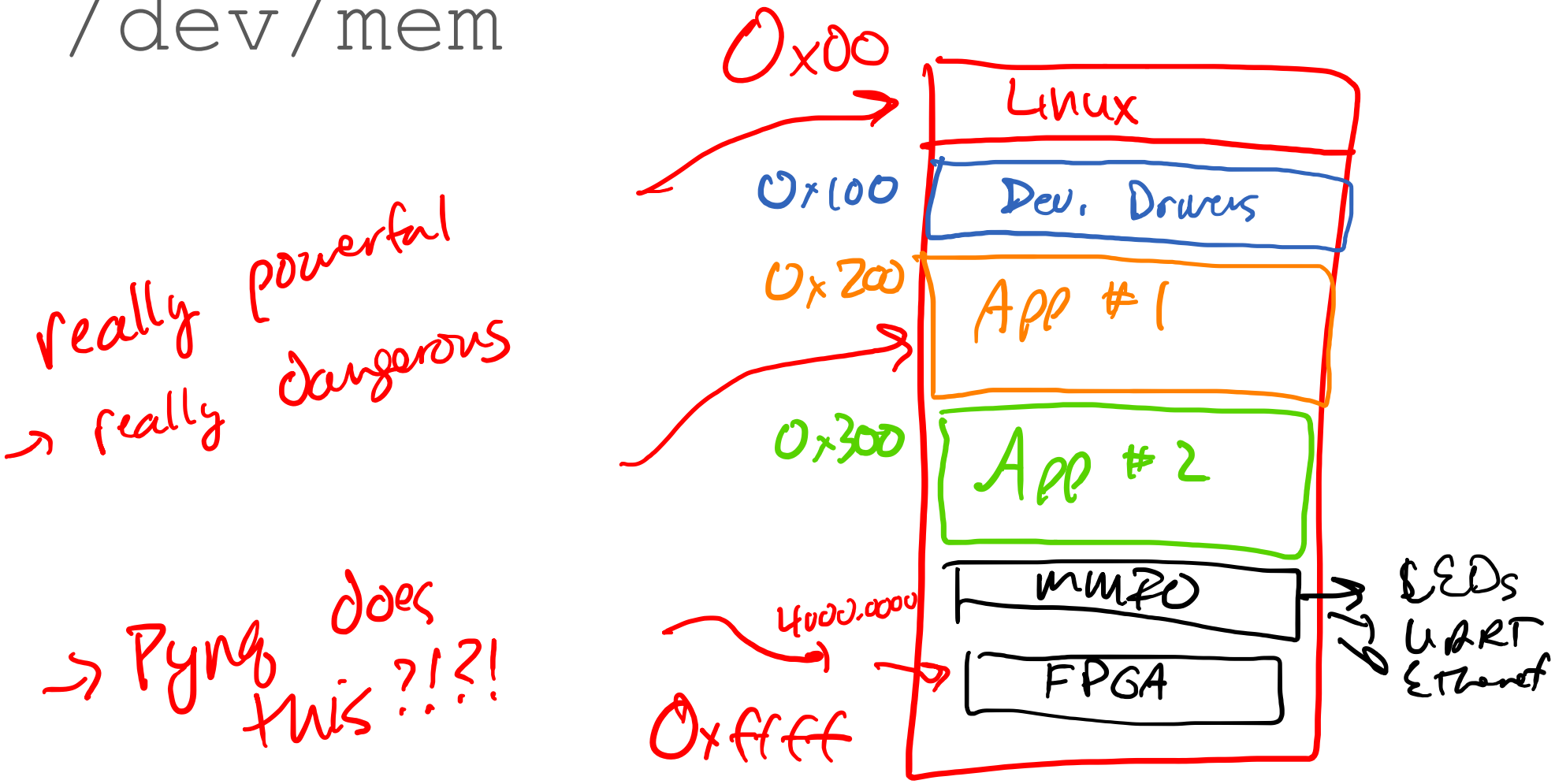
```
from pynq import Overlay
from pynq import MMIO
class hw_ema():
    def __init__(self):
        self.overlay = Overlay('bitstream.bit')
        self.mmio = self.overlay.axi_popcount_0.S_AXI_LITE
    def ema(self, n):
        self.mmio.write(0x0, int(n))
        return self.mmio.read(0x0)
ema = hw_ema()
for i in range(1000,6000,1000):
    x = ema.ema(i)
    print ("In: ", i, " Out: ", x)
```

```
$ sudo python3 mmio_demo.py
In:  1000  Out:  250
In:  2000  Out:  687
In:  3000  Out:  1264
In:  4000  Out:  1948
In:  5000  Out:  2711
```

/dev/mem

- */dev/mem* is a character device file that is an image of the main memory of the computer. It may be used, for example, to examine (and even patch) the system.
- Byte addresses in */dev/mem* are interpreted as physical memory addresses. References to nonexistent locations cause errors to be returned.
- Requires root (sudo) access

/dev/mem

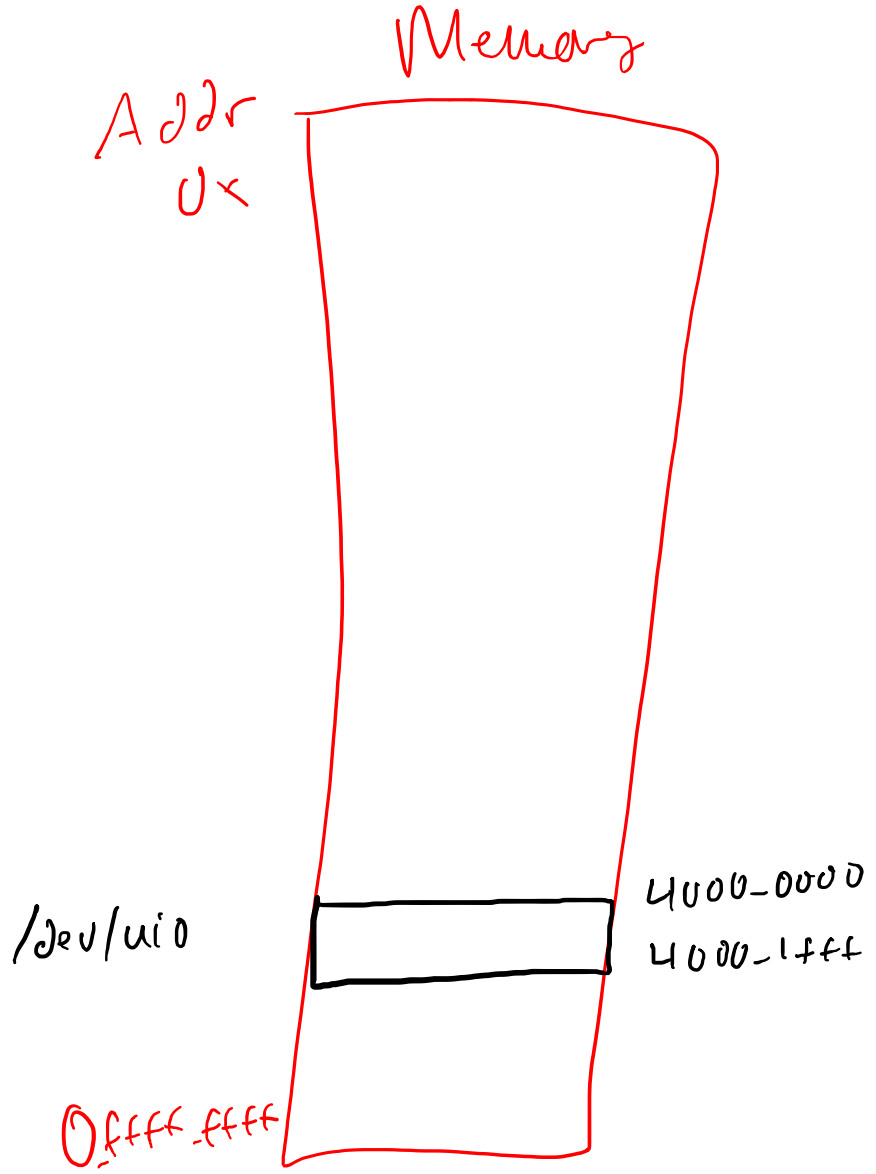

really powerful
→ really dangerous
→ Pyng does this ?!?!
0x00
0x100
0x200
0x300
0x4000.0000
0xffff
Linux
Dev. Drivers
App #1
App #2
MMIO
FPGA
LEDs
UART
Ethernet

Userspace I/O

- Linux provides a more restrictive version of /dev/mem called /dev/uio
- Linux exposes a small portion of I/O memory space to the user through /dev/uioX interfaces.
- Treat it just like /dev/mem, but not the entire memory space

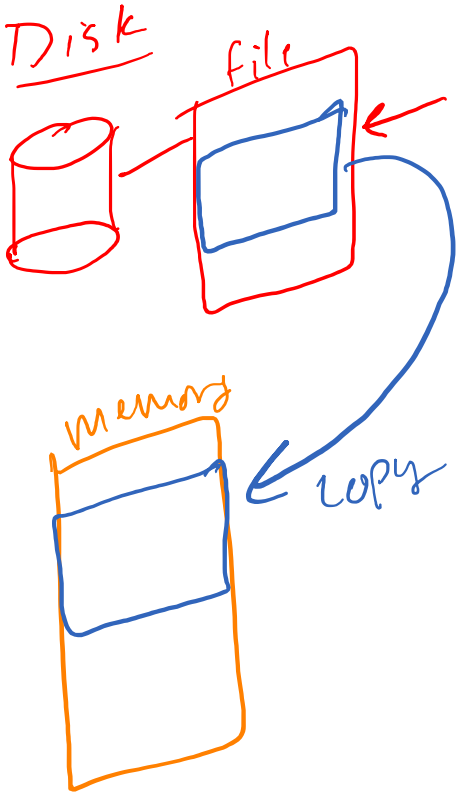
User Accessible 

/dev/uio



/dev/uio

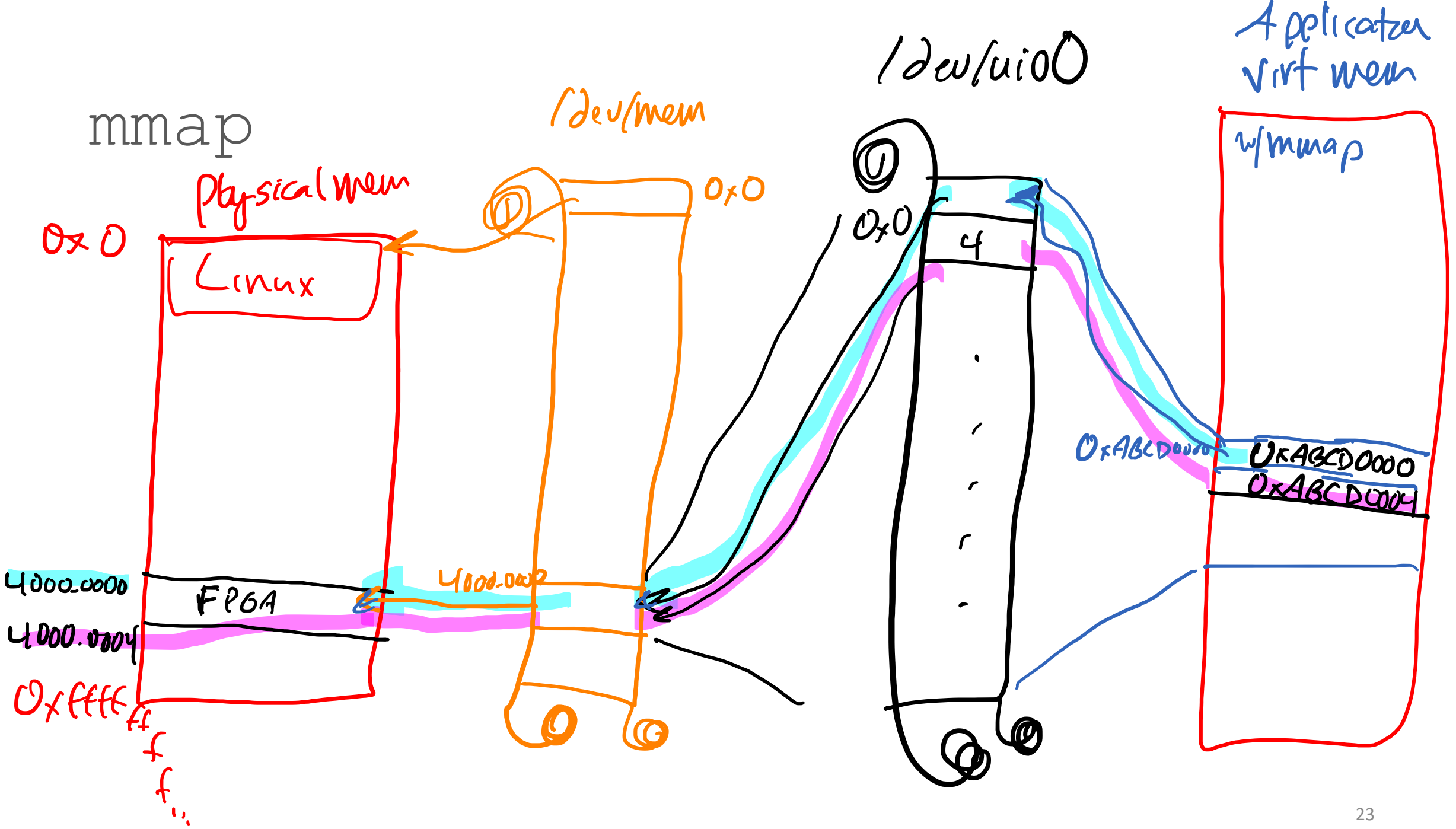
mmap – memory map



- **mmap**() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in *addr*.

- The contents of a file mapping are initialized using *length* bytes starting at offset *offset* in the file (or other object) referred to by the file descriptor *fd*.

mmap


/dev/mem
/dev/uio0
Application virt mem
w/mmap
Physical mem
0x0
Linux
0x0
0x0
4
FPGA
4000_0000
4000_0004
0xffff_ffff
0xABCD0000
0xABCD0004

Userspace EMA w/C

```
int main (){

    int dev_mem_fd = -1;
    void * vaddr_base;

    //Mapping user-space I/O
    dev_mem_fd = open("/dev/uio0", O_RDWR|O_SYNC);
    if (dev_mem_fd < 0) { perror("open() /dev/uio0"); return 1; }

   // Map 1KB of physical memory starting at uio 0x0 (real 0x40000000)
    // to 1KB of virtual memory starting at vaddr_base
    vaddr_base = mmap(0, 1024, PROT_READ|PROT_WRITE,
                MAP_SHARED, dev_mem_fd, 0x0);

    if (vaddr_base == MAP_FAILED) { perror("mmap()"); return 1; }
```

```
    volatile uint32_t * ema_reg = (uint32_t*) vaddr_base;
    uint32_t tmp;

    for (int i = 1000; i < 6000; i +=1000){
        printf ("Sending in:  %d\n", i);
        *ema_reg = i; //mmio store
        tmp = *ema_reg; //mmio load
        printf("Receiving: %d\n", tmp);
    }

    if (munmap(vaddr_base, 1024) != 0) { perror("munmap()"); }
    if (close(dev_mem_fd) != 0) { perror("close()"); }
    dev_mem_fd = -1;

    return 0;
}
```

Userspace EMA w/C

```
int main (){

    int dev_mem_fd = -1;
    void * vaddr_base;

    //Mapping user-space I/O
    dev_mem_fd = open("/dev/uio0", O_RDWR|O_SYNC);
    if (dev_mem_fd < 0) { perror("open() /dev/uio0"); return 1; }

   // Map 1KB of physical memory starting at uio 0x0 (real 0x40000000)
    // to 1KB of virtual memory starting at vaddr_base
    vaddr_base = mmap(0, 1024, PROT_READ|PROT_WRITE,
                MAP_SHARED, dev_mem_fd, 0x0);

    if (vaddr_base == MAP_FAILED) { perror("mmap()"); return 1; }
```

```
    volatile uint32_t * ema_reg = (uint32_t*) vaddr_base;
    uint32_t tmp;

    for (int i = 1000; i < 6000; i +=1000){
        printf ("Sending in:  %d\n", i);
        *ema_reg = i; //mmio store
        tmp = *ema_reg; //mmio load
        printf("Receiving: %d\n", tmp);
    }

    if (munmap(vaddr_base, 1024) != 0) { perror("munmap()"); }
    if (close(dev_mem_fd) != 0) { perror("close()"); }
    dev_mem_fd = -1;

    return 0;
}
```

Userspace EMA w/C

```
int main (){

    int dev_mem_fd = -1;
    void * vaddr_base;

    //Mapping user-space I/O
    dev_mem_fd = open("/dev/uio0", O_RDWR|O_SYNC);
    if (dev_mem_fd < 0) { perror("open() /dev/uio0"); return 1; }

   // Map 1KB of physical memory starting at uio 0x0 (real 0x40000000)
    // to 1KB of virtual memory starting at vaddr_base
    vaddr_base = mmap(0, 1024, PROT_READ|PROT_WRITE,
                MAP_SHARED, dev_mem_fd, 0x0);

    if (vaddr_base == MAP_FAILED) { perror("mmap()"); return 1; }
```

```
    volatile uint32_t * ema_reg = (uint32_t*) vaddr_base;
    uint32_t tmp;

    for (int i = 1000; i < 6000; i +=1000){
        printf ("Sending in:  %d\n", i);
        *ema_reg = i; //mmio store
        tmp = *ema_reg; //mmio load
        printf("Receiving: %d\n", tmp);
    }

    if (munmap(vaddr_base, 1024) != 0) { perror("munmap()"); }
    if (close(dev_mem_fd) != 0) { perror("close()"); }
    dev_mem_fd = -1;

    return 0;
}
```

Userspace EMA w/C

```
int main (){

    int dev_mem_fd = -1;
    void * vaddr_base;

    //Mapping user-space I/O
    dev_mem_fd = open("/dev/uio0", O_RDWR|O_SYNC);
    if (dev_mem_fd < 0) { perror("open() /dev/uio0"); return 1; }

   // Map 1KB of physical memory starting at uio 0x0 (real 0x40000000)
    // to 1KB of virtual memory starting at vaddr_base
    vaddr_base = mmap(0, 1024, PROT_READ|PROT_WRITE,
                MAP_SHARED, dev_mem_fd, 0x0);

    if (vaddr_base == MAP_FAILED) { perror("mmap()"); return 1; }
```

```
    volatile uint32_t * ema_reg = (uint32_t*) vaddr_base;
    uint32_t tmp;

    for (int i = 1000; i < 6000; i +=1000){
        printf ("Sending in:  %d\n", i);
        *ema_reg = i; //mmio store
        tmp = *ema_reg; //mmio load
        printf("Receiving: %d\n", tmp);
    }

    if (munmap(vaddr_base, 1024) != 0) { perror("munmap()"); }
    if (close(dev_mem_fd) != 0) { perror("close()"); }
    dev_mem_fd = -1;

    return 0;
}
```

Userspace EMA w/C

```
int main (){

    int dev_mem_fd = -1;
    void * vaddr_base;

    //Mapping user-space I/O
    dev_mem_fd = open("/dev/uio0", O_RDWR|O_SYNC);
    if (dev_mem_fd < 0) { perror("open() /dev/uio0"); return 1; }

   // Map 1KB of physical memory starting at uio 0x0 (real 0x40000000)
    // to 1KB of virtual memory starting at vaddr_base
    vaddr_base = mmap(0, 1024, PROT_READ|PROT_WRITE,
                MAP_SHARED, dev_mem_fd, 0x0);

    if (vaddr_base == MAP_FAILED) { perror("mmap()"); return 1; }
```

```
    volatile uint32_t * ema_reg = (uint32_t*) vaddr_base;
    uint32_t tmp;

    for (int i = 1000; i < 6000; i +=1000){
        printf ("Sending in:  %d\n", i);
        *ema_reg = i; //mmio store to 0x4000_0000
        tmp = *ema_reg; //mmio load from 0x4000_0000
        printf("Receiving: %d\n", tmp);
    }

    if (munmap(vaddr_base, 1024) != 0) { perror("munmap()"); }
    if (close(dev_mem_fd) != 0) { perror("close()"); }
    dev_mem_fd = -1;

    return 0;
}
```

Complete EMA

```
#include <fcntl.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>

int main (){

    int dev_mem_fd = -1;
    void * vaddr_base;

    //Mapping user-space I/O
    dev_mem_fd = open("/dev/uio0", O_RDWR|O_SYNC);
    if (dev_mem_fd < 0) { perror("open() /dev/uio0"); return 1; }

    // Map 1KB of physical memory starting at uio 0x0 (real 0x40000000)
    // to 1KB of virtual memory starting at vaddr_base
    vaddr_base = mmap(0, 1024, PROT_READ|PROT_WRITE,
                MAP_SHARED, dev_mem_fd, 0x0);

    if (vaddr_base == MAP_FAILED) { perror("mmap()"); return 1; }

    volatile uint32_t * ema_reg = (uint32_t*) vaddr_base;
    uint32_t tmp;

    for (int i = 1000; i < 6000; i +=1000){
        printf ("Sending in:  %d\n", i);
        *ema_reg = i; //mmio store
        tmp = *ema_reg; //mmio load
        printf("Receiving: %d\n", tmp);
    }

    if (munmap(vaddr_base, 1024) != 0) { perror("munmap()"); }
    if (close(dev_mem_fd) != 0) { perror("close()"); }
    dev_mem_fd = -1;

    return 0;
}
```

Demo Time

Next Time:

- DMA

10: Linux MMIO

Engr 315: Hardware / Software Codesign

Andrew Lukefahr
Indiana University