Memory Management:
Translation and Protection
Memory Management

- Hello World

  - Why translation?
    - Case study: software TLB in egos
  - Why protection?
    - Case study: physical memory protection (PMP)
  - Combining the two: page table and virtual memory
Recall hello-world from week#2

```c
int str_len = 14;

int main() {
    char* str = malloc(str_len);
    memcpy(str, "Hello World!\n", str_len);
    printf("%s", str);
    return 0;
}
```
Recall memory map from week #6

<table>
<thead>
<tr>
<th>Base</th>
<th>Top</th>
<th>Attr.</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000_0000</td>
<td>0x0000_0000</td>
<td>RA</td>
<td>Debug</td>
<td></td>
</tr>
<tr>
<td>0x0000_1000</td>
<td>0x0000_1000</td>
<td>RA</td>
<td>Mode Select</td>
<td></td>
</tr>
<tr>
<td>0x0000_2000</td>
<td>0x0000_2000</td>
<td>RW</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0x0000_3000</td>
<td>0x0000_3000</td>
<td>RW</td>
<td>Error Device</td>
<td></td>
</tr>
<tr>
<td>0x0000_4000</td>
<td>0x0000_4000</td>
<td>RW</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0x0000_5000</td>
<td>0x0000_5000</td>
<td>RW</td>
<td>Mask ROM (8 KB)</td>
<td></td>
</tr>
<tr>
<td>0x0000_6000</td>
<td>0x0000_6000</td>
<td>RW</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0x0000_7000</td>
<td>0x0000_7000</td>
<td>RW</td>
<td>OTP Memory Region</td>
<td></td>
</tr>
<tr>
<td>0x0000_8000</td>
<td>0x0000_8000</td>
<td>RW</td>
<td>CLINT</td>
<td></td>
</tr>
<tr>
<td>0x0001_0000</td>
<td>0x0001_0000</td>
<td>RW</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0x0001_1000</td>
<td>0x0001_1000</td>
<td>RW</td>
<td>E31 TIM (8 KB)</td>
<td></td>
</tr>
<tr>
<td>0x0001_2000</td>
<td>0x0001_2000</td>
<td>RW</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0x0001_3000</td>
<td>0x0001_3000</td>
<td>RW</td>
<td>PLIC</td>
<td></td>
</tr>
<tr>
<td>0x0001_4000</td>
<td>0x0001_4000</td>
<td>RW</td>
<td>AON</td>
<td></td>
</tr>
<tr>
<td>0x0001_5000</td>
<td>0x0001_5000</td>
<td>RW</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0x0001_6000</td>
<td>0x0001_6000</td>
<td>RW</td>
<td>PRIC</td>
<td></td>
</tr>
<tr>
<td>0x0001_7000</td>
<td>0x0001_7000</td>
<td>RW</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0x0001_8000</td>
<td>0x0001_8000</td>
<td>RW</td>
<td>OTP Control</td>
<td></td>
</tr>
<tr>
<td>0x0001_9000</td>
<td>0x0001_9000</td>
<td>RW</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0x0010_0000</td>
<td>0x0010_0000</td>
<td>RW</td>
<td>GPDO</td>
<td></td>
</tr>
<tr>
<td>0x0010_1000</td>
<td>0x0010_1000</td>
<td>RW</td>
<td>UART 0</td>
<td></td>
</tr>
<tr>
<td>0x0010_2000</td>
<td>0x0010_2000</td>
<td>RW</td>
<td>QSPI 0</td>
<td></td>
</tr>
<tr>
<td>0x0010_3000</td>
<td>0x0010_3000</td>
<td>RW</td>
<td>PWM 0</td>
<td></td>
</tr>
<tr>
<td>0x0010_4000</td>
<td>0x0010_4000</td>
<td>RW</td>
<td>I2C 0</td>
<td></td>
</tr>
<tr>
<td>0x0010_5000</td>
<td>0x0010_5000</td>
<td>RW</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0x0010_6000</td>
<td>0x0010_6000</td>
<td>RW</td>
<td>UART 1</td>
<td></td>
</tr>
<tr>
<td>0x0010_7000</td>
<td>0x0010_7000</td>
<td>RW</td>
<td>SPI 1</td>
<td></td>
</tr>
<tr>
<td>0x0010_8000</td>
<td>0x0010_8000</td>
<td>RW</td>
<td>PWM 1</td>
<td></td>
</tr>
<tr>
<td>0x0010_9000</td>
<td>0x0010_9000</td>
<td>RW</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0x0010_A000</td>
<td>0x0010_A000</td>
<td>RW</td>
<td>SPI 2</td>
<td></td>
</tr>
<tr>
<td>0x0010_B000</td>
<td>0x0010_B000</td>
<td>RW</td>
<td>PWM 2</td>
<td></td>
</tr>
<tr>
<td>0x0010_C000</td>
<td>0x0010_C000</td>
<td>RW</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0x0010_D000</td>
<td>0x0010_D000</td>
<td>RW</td>
<td>QSPI 0 Flash (512 MB)</td>
<td></td>
</tr>
<tr>
<td>0x0010_E000</td>
<td>0x0010_E000</td>
<td>RW</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0x0010_F000</td>
<td>0x0010_F000</td>
<td>RW</td>
<td>E31 TIM (16 KB)</td>
<td></td>
</tr>
<tr>
<td>0x0020_0000</td>
<td>0x0020_0000</td>
<td>RW</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

CPU debug @0000_0000
(ignore this for building an OS)

Device control @0200_0000

Boot ROM @2000_0000

Main memory @0x8000_0000
Before boot up

• **Boot ROM** holds code, read-only data and data.
Boot up, step #1

- **Boot ROM** holds code, read-only data and data.
- The first few instructions setup the **stack pointer**.
Boot up, step #2

- **Boot ROM** holds code, read-only data and data.

- The first few instructions setup the **stack pointer**.

- **Copy data** from ROM to **main memory**.

Hello-world
After the 2-step boot up

- **Boot ROM** holds code, read-only data and data.
- The first few instructions setup the stack pointer.
- Copy data from ROM to main memory.
- The break pointer is saved as 8-byte in data.
Reading the code

• **Boot ROM** holds code, read-only data and data.

• The first few instructions setup the **stack pointer**.

• **Copy data** from ROM to main memory.

• The **break pointer** is saved as 8-byte in data.

  - earth/earth.lds
  - earth/earth.S
  - earth/earth.c
  - & first 2 loops in main()
  - earth/earth.lds
  - & libc/malloc.c
Hello-world → Multi-threading

Boot ROM

Main memory

Code
Read-only data
Data
Heap
Stack
Stack
Stack

Stack pointer of thread #1
Stack pointer of thread #2
Stack pointer of thread #3
Memory Management

• Hello World

➡️ Why translation?
  • Case study: software TLB in egos

• Why protection?
  • Case study: physical memory protection (PMP)

• Combining the two: page table and virtual memory
Why translation?

• In P1, you write the code of every thread yourself.

• In an operating system,
  • Google writes the code of Chrome
  • Adobe writes the code of Photoshop
  • ...

• An operating system provides the standard memory layout specifying where to put code, stack, etc.
Standard memory layout of win32

- code, rodata and data of application
- for dynamic library linking, out of today’s scope
- everything else, including the OS code and data

https://mikeczumak.com/blog/windows-exploit-development-part-1-basics/
Goal #1 of memory translation

Different threads have different stack address, code address, etc.

Different processes have the same stack address, code address, etc.
Goal #2 of memory translation

Different threads share the same code, data, heap regions.

Different processes have separate code, data and heap regions.
Memory Management

- Hello World
- Why translation?
  - Case study: software TLB in egos
- Why protection?
  - Case study: physical memory protection (PMP)
- Combining the two: page table and virtual memory
Case study: Software TLB

• Every memory page is 4KB (0x1000 bytes).

• For every application in egos-2000:
  • 3 pages for code/rodata/data/heap
    • 0x0800_5000 … 0x0800_8000
  • 2 pages for stack
    • 0x8000_0000 … 0x8000_2000

• In addition, egos-2000 maintains a buffer of 256 pages (1 MB)
3 memory regions for Software TLB

- **Page * 3**
  - Application code/data/heap
  - 0x0800_5000

- **Page * 2**
  - Application stack
  - 0x8000_0000

- **Page * 256**
  - Memory buffer
  - 0x8000_4000

Operating system code/data/heap/stack is in other memory regions. For example, consider your `thread_create()` and `thread_yield()` in P1.
RUNNING and RUNNABLE processes

Code/data/heap of the **RUNNING** process

- Page * 3
- 0x0800_5000

Stack of the **RUNNING** process

- Page * 2
- 0x8000_0000

All pages of all **RUNNABLE** processes

- Memory buffer
- 0x8000_4000
Additional step in `create()`

Find 5 free pages in the memory buffer and load the code/data of the new process.

All pages of all RUNNABLE processes
Additional step #1 in `yield()`

Write these 5 pages to the memory buffer.

All pages of all RUNNABLE processes
Additional step #2 in `yield()`

Load the 5 pages of another process from the memory buffer.
Software TLB summary

- Dedicated memory regions for user application.
  - 3 for code/data/heap + 2 for stack
- Comparing to the multi-threading in P1,
  - `create()` allocates and initializes memory pages for the code/data/heap of the process, in addition to stack
  - `yield()` moves memory pages between the dedicated regions (RUNNING) and the memory buffer (RUNNABLE)
Memory Management

• Hello World

• Why translation?
  • Case study: software TLB in egos

Why protection?

  • Case study: physical memory protection (PMP)

  • Combining the two: page table and virtual memory
RUNNING process **should not** access the buffer

**Page * 3**

*RUNNING process can access*

0x0800_5000

**Page * 2**

*RUNNING process can access*

0x8000_0000

**Memory buffer**

*RUNNING process cannot access*

0x8000_4000
Introducing privilege levels

- **Machine mode** can access all memory regions.
- **User mode** can only access regions that are allowed by the machine mode.
  - Machine mode specify these regions and permissions.
  - Permissions are usually readable / writable / executable.
Machine mode specifies user permissions

- Page * 3
  - code: r/-/x
  - rodata: r/-/-
  - data: r/w/-
  - heap: r/w/-
  - 0x0800_5000

- Page * 2
  - stack: r/w/-
  - 0x8000_0000

- Memory buffer
  - -/-/- (no access at all)
  - 0x8000_4000
Memory Management

- Hello World
- Why translation?
  - Case study: software TLB in egos
- Why protection?
  ➡️ Case study: physical memory protection (PMP)
- Combining the two: page table and virtual memory
Physical memory protection (PMP)

- Read section 3.6 of the RISC-V manual
- There are 16 address CSRs and 4 config CSRs
  - pmpaddr0 ... pmpaddr15 + pmpcfg0 ... pmpcfg3
- For example, TOR means “top of region”:

```c
/* Setup PMP TOR region 0x00000000 - 0x08008000 as r/w/x */
asm("csrw pmpaddr0, %0" :: "r" (0x08008000));
asm("csrw pmpcfg0, %0" :: "r" (0xF));
```
Memory Management

• Hello World

• Why translation?
  • Case study: software TLB in egos

• Why protection?
  • Case study: physical memory protection (PMP)

➡️ Combining the two: page table and virtual memory
Page table and virtual memory

• Achieve the same goals as PMP + software TLB.

• In P3, page table translation is left to you as an open-ended hobby project, not graded.
  • Again, the goal of 4411 is to have fun.

• There are 29 lines of code setting up some example page tables in egos-2000. See the handout for details.
Homework

- P3 will be due on Nov 4. You will implement
  - system call, memory protection and exception handling
- Next lecture: I/O bus and device driver