Where does the OS live?

- **In its own address space?**
  - Can’t do this on most hardware (e.g., syscall instruction won’t switch address spaces)
  - Also would make it harder to parse syscall arguments passed as pointers

- **So in the same address space as process**
  - Use protection bits to prohibit user code from writing kernel

- **Typically all kernel text, most data at same VA in every address space**
  - On x86, must manually set up page tables for this
  - Usually just map kernel in contiguous virtual memory when boot loader puts kernel into contiguous physical memory
  - Some hardware puts physical memory (kernel-only) somewhere in virtual address space
Hardware/Software Managed TLBs

- **Intel Evolution:**
  - 80286: Segmentation Only
  - 80386: Segmentation + Paging
  - AMD64/x86-64: Paging (segmentation basically removed)

- **Hardware Managed TLB: x86 hardware reloads the TLB**
  - Hardware reloads the TLB as needed from page tables
  - Violations of protection bits lead to page faults

- **Software Managed TLB: MIPS, SPARC, Alpha, POWER**
  - Page fault triggered to ask OS to reload the TLB
  - OS designs its own paging structure
  - Hardware must provide fast exception handling
Very different MMU: MIPS

- Hardware checks TLB on application load/store
  - References to addresses not in TLB trap to kernel

- Each TLB entry has the following fields:
  Virtual page, Pid, Page frame, NC, D, V, Global

- Kernel itself unpaged
  - All of physical memory contiguously mapped in high VM
    (hardwired in CPU)
  - Kernel uses these pseudo-physical addresses

- User TLB fault handler very efficient
  - Two hardware registers reserved for it
  - utlb miss handler can itself fault—allow paged page tables

- OS is free to choose page table format!
### MIPS Memory Layout

**Kernel Memory**
- **kseg2**: Paged Kernel
- **kseg1**: Phys. (Uncached)
- **kseg0**: Physical Memory

**User Memory**
- kuseg
- User Paging

---

Address Ranges:
- FFFF FFFF
- C000 0000
- BFFF FFFF
- A000 0000
- 9FFF FFFF
- 8000 0000
- 7FFF FFFF
- 0000 0000
MIPS Translation Lookaside Buffer (TLB)

- TLB caches mappings from a virtual to a physical address
- Specifically it replaces the upper 20 bits of any address
- MIPS R3000 contains 64 TLB entries

<table>
<thead>
<tr>
<th>Page Frame</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>
MIPS Translation Lookaside Buffer (TLB)

- MIPS co-processor 0 (COP0) provides TLB functionality
- `tlbwr`: TLB write a random slot
- `tlbwi`: TLB write a specific slot
- `tlbr`: TLB read a specific slot
- `tlbp`: TLB probe for the index given an address
- **Registers** `c0_entryhi`, `c0_entrylo`, `c0_index`
- **Entry Hi:**

<table>
<thead>
<tr>
<th>Physical Page Number (PPN)</th>
<th>PID</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

- **Entry Lo:** W (writable), V (valid)

<table>
<thead>
<tr>
<th>Frame Number (VPN)</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
/*
 * tlb_random: use the "tlbwr" instruction to write a TLB entry into a (very pseudo-) random slot in the TLB.
 *
 * Pipeline hazard: must wait between setting entryhi/lo and doing the tlbwr. Use two cycles; some processors may vary.
 */

tlb_random:
    mtc0 a0, c0_entryhi  /* store the passed entry into the */
    mtc0 a1, c0_entrylo  /* tlb entry registers */
    nop                 /* wait for pipeline hazard */
    nop
    tlbwr               /* do it */
    j ra
    nop
tlb_write:

mtc0 a0, c0_entryhi  /* store the passed entry into the */
mtc0 a1, c0_entrylo  /* tlb entry registers */
/* shift the passed index into place */
sll t0, a2, CIN_INDEXSHIFT
/* store the shifted index into the index register */
mtc0 t0, c0_index
nop  /* wait for pipeline hazard */
nop
j ra
nop

/*
* tlb_write: use the "tlbwi" instruction to write a TLB
* entry into a selected slot in the TLB.
*
* Pipeline hazard: must wait between setting entryhi/lo
* and doing the tlbwi. Use two cycles; some processors
* may vary.
*/
/* tlb_read: use the "tlbr" instruction to read a TLB entry from a selected slot in the TLB. */

/* Pipeline hazard: must wait between setting c0_index and doing the tlbr. Use two cycles; some processors may vary. Similarly, three more cycles before reading c0_entryhi/lo. */

tlb_read:
    sll t0, a2, CIN_INDEXSHIFT /* shift the passed index into place */
    mtc0 t0, c0_index /* store the shifted index into the index */
    nop /* wait for pipeline hazard */
    nop
    tlbr /* do it */
    nop /* wait for pipeline hazard */
    nop
    nop
    mfc0 t0, c0_entryhi /* get the tlb entry out of the */
    mfc0 t1, c0_entrylo /* tlb entry registers */
    sw t0, 0(a0) /* store through the passed pointer */
    j ra
    sw t1, 0(a1) /* store (in delay slot) */
tlb_probe:
mtc0 a0, c0_entryhi /* store the passed entry into the */
mtc0 a1, c0_entrylo /* tlb entry registers */
nop /* wait for pipeline hazard */
nop
nop
tlbp /* do it */
nop /* wait for pipeline hazard */
nop
mfc0 t0, c0_index /* fetch the index back in t0 */
/*
  * If the high bit (CIN_P) of c0_index is set, the probe failed.
  * The high bit is not set <---> c0_index (now in t0) >= 0.
  */
bgez t0, 1f /* did probe succeed? if so, skip forward */
nop /* delay slot */
    /* set return value to -1 to indicate failure */
addi v0, z0, -1
j ra /* done */
nop /* delay slot */

1:
/*
  * succeeded: get the index field from the index register value.
  */
andi t1, t0, CIN_INDEX /* mask off the field */
j ra /* done */
sra v0, t1, CIN_INDEXSHIFT /* shift it (in delay slot) */
### Typical Userspace Memory Layout

- **Example layout of user/testbin/sort**
- **Using <2MB of memory, but spread over 2GB VA space**

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FFF FFFF</td>
<td>Stack</td>
</tr>
<tr>
<td>xxxx xxxx</td>
<td></td>
</tr>
<tr>
<td>1012 00B0</td>
<td>Data</td>
</tr>
<tr>
<td>1000 0000</td>
<td></td>
</tr>
<tr>
<td>0040 1A0C</td>
<td>Text (code) + R/O Data</td>
</tr>
<tr>
<td>0040 0000</td>
<td></td>
</tr>
</tbody>
</table>
Managing OS Page Tables

- Example used 1.4MB of memory, over a 2GB VA space
- Simple Map: linear table of 4KB pages
  - Requires 2MBs of memory. More than the program!
  - Lots of unused memory
Managing OS Page Tables

- Example used 1.4MB of memory, over a 2GB VA space

- **Simple Map:** linear table of 4KB pages
  - Requires 2MBs of memory. More than the program!
  - Lots of unused memory

- **Segment Map:** requires very little memory
  - Requires very little memory
  - Requires contiguous segments of physical memory
Managing OS Page Tables

- Example used 1.4MB of memory, over a 2GB VA space
- **Simple Map:** linear table of 4KB pages
  - Requires 2MBs of memory. More than the program!
  - Lots of unused memory
- **Segment Map:** requires very little memory
  - Requires very little memory
  - Requires contiguous segments of physical memory
- **Radix tree:** similar to x86 hardware page tables
  - Requires around 16KB (for sort example)
  - More complex implementation
Managing OS Page Tables

- Example used 1.4MB of memory, over a 2GB VA space
- Simple Map: linear table of 4KB pages
  - Requires 2MBs of memory. More than the program!
  - Lots of unused memory
- Segment Map: requires very little memory
  - Requires very little memory
  - Requires contiguous segments of physical memory
- Radix tree: similar to x86 hardware page tables
  - Requires around 16KB (for sort example)
  - More complex implementation
- VM Objects: hybrid between segments and radix
  - Originally from Mach (used in FreeBSD and other OSes)
  - Memory space consists of objects
  - Each object is represented by a radix tree or similar structure
  - Simplifies sharing memory mapped objects
- Fixed segment map
- You will implement a better version in Lab 3

```c
struct addrspace {
    vaddr_t as_vbase1;
    paddr_t as_pbase1;
    size_t as_npages1;
    vaddr_t as_vbase2;
    paddr_t as_pbase2;
    size_t as_npages2;
    paddr_t as_stackpbase;
};
```
vbase1 = as->as_vbase1;
vtop1 = vbase1 + as->as_npages1 * PAGE_SIZE;
vbase2 = as->as_vbase2;
vtop2 = vbase2 + as->as_npages2 * PAGE_SIZE;
stackbase = USERSTACK - DUMBVM_STACKPAGES * PAGE_SIZE;
stacktop = USERSTACK;

if (faultaddress >= vbase1 && faultaddress < vtop1) {
    paddr = (faultaddress - vbase1) + as->as_pbase1;
}
else if (faultaddress >= vbase2 && faultaddress < vtop2) {
    paddr = (faultaddress - vbase2) + as->as_pbase2;
}
else if (faultaddress >= stackbase && faultaddress < stacktop) {
    paddr = (faultaddress - stackbase) + as->as_stackpbase;
}
else {
    return EFAULT;
}
for (i=0; i<NUM_TLB; i++) {
    tlb_read(&ehi, &elo, i);
    if (elo & TLBLO_VALID) {
        continue;
    }
    ehi = faultaddress;
    elo = paddr | TLBLO_DIRTY | TLBLO_VALID;
    DEBUG(DB_VM, "dumbvm: 0x%x -> 0x%x\n", faultaddress, paddr);
    tlb_write(ehi, elo, i);
    splx(spl);
    return 0;
}
for (i=0; i<NUM_TLB; i++) {
    tlb_write(TLBHI_INVALID(i), TLBLO_INVALID(), i);
}