Recap of Address Spaces

- Must offer three properties:
  - Address independence
  - Protection
  - Virtual memory

- Need dynamic instead of static translation

Base and bounds

- Load each process into contiguous region of physical memory
- Prevent process from accessing data outside its region

```
if (virtual address > bound) {
    trap to kernel; kill process (core dump)
} else {
    physical address = virtual address + base
}
```

Base and bounds

- Pros?
  - Fast
  - Simple hardware support

- Cons?
  - Virtual address space limited by physical memory
  - No controlled sharing
  - External fragmentation

External fragmentation

- Processes come and go, leaving a mishmash of available memory regions
- Wasted memory between allocated regions

Growing address space

- How can stack and heap grow independently?
**Segmentation**

- Divide address space into segments

- Segment: region of memory contiguous in both physical memory and in virtual address space

---

- Virtual address is of the form: (segment #, offset)
  - Physical address = base for segment + offset

- Many ways to specify the segment #
  - High bits of address
  - Special register
  - Implicit to instruction opcode

---

- Not all virtual addresses are valid
  - Valid → region is part of process’s address space
  - Invalid → virtual address is illegal to access
    - Accessing an invalid virtual address causes a trap to OS (usually resulting in core dump)

- Reasons for virtual address being invalid?
  - Invalid segment number
  - Offset within valid segment beyond bound

---

- How to grow a segment?

- Different segments can have different protection
  - E.g., code is usually read only (allows instruction fetch, load)
  - E.g., data is usually read/write (allows instruction fetch, load, store)
  - Protection in base and bounds?

- What must be changed on a context switch?
**Benefits of Segmentation**

- Multiple areas of address space can grow separately
- Easy to share part of address space

<table>
<thead>
<tr>
<th>Segment #</th>
<th>Base</th>
<th>Bounds</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4000</td>
<td>700</td>
<td>code segment</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>500</td>
<td>data segment</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
<td>1000</td>
<td>stack segment</td>
</tr>
</tbody>
</table>

**Drawbacks of Segmentation**

- Have we eliminated external fragmentation?
- Can an address space be larger than physical memory?
- How can we:
  - Make memory allocation easy
  - Not have to worry about external fragmentation
  - Allow address space size to be > physical memory

**Project 2**

- Due in a week
  - Check calendar on web page for extra office hours
- For every thread, think about “where is the current context?”
- Think about memory leaks and how to test if they exist

**Mid-Term**

- In two weeks
  - Covers intro + threads/concurrency
- Sample exams posted on web page
  - GSIs will go over solutions during discussion section next Friday and a review session
  - Important to try questions yourself first!

**Paging**

- Allocate physical memory in fixed-size units (pages)
  - Any free physical page can store any virtual page
- Virtual address is split into
  - Virtual page # (high bits of address, e.g., bits 31-12)
  - Offset (low bits of address, e.g., bits 11-0, for 4 KB page size)
- Translation data is the page table

<table>
<thead>
<tr>
<th>Virtual page #</th>
<th>Physical page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>105</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>283</td>
</tr>
<tr>
<td>3</td>
<td>Invalid</td>
</tr>
<tr>
<td>...</td>
<td>Invalid</td>
</tr>
<tr>
<td>1048575</td>
<td>Invalid</td>
</tr>
</tbody>
</table>
Paging

- Each virtual page can be in physical memory or "paged out" to disk
- How does processor know that a virtual page is not in physical memory?
- Like segments, pages can have different protections (e.g., read, write, execute)

```c
if (virtual page is invalid or non-resident or protected) {
    trap to OS fault handler
} else {
    physical page # = pageTable[virtual page #].physPageNum
}
```

Valid versus resident

- **Valid** → virtual page is legal for process to access
- **Resident** → virtual page is in physical memory
- Error to access invalid page, but not to access non-resident page

- Who makes a virtual page resident/non-resident?
- Who makes a virtual page valid/invalid?
- Why would a process want one of its virtual pages to be invalid?