Grad School Event

Connect with graduate student and learn about graduate school options + experiences!

Nov 16 6–7:30PM @ Gates 114

P.S. Come for desserts!

RSVP
ANNOUNCEMENTS

- We have an alternate time for the prelim: 5:30pm. See Ed post #652 and your email for details.

- We will have a review session before the prelim on Tuesday, 11/15, from 6:00 pm to 8:00 pm, tentatively in Statler 196. Will confirm on Ed.

- Practice exams were made available. See Ed post #659

- The prelim will be comprehensive. See Ed post #409, #421, and #638

- Exam for students with SDS accommodations: November 17 at 6:45pm in Statler 198. Email to follow
The Identity Mapping

- Map each virtual address onto the identical physical address
  - Virtual and physical address spaces have the same size
  - Run a single program at a time
    - OS can be a simple library
    - very early computers

Friendly amendment: leave some of the physical address space for the OS

- Use loader to relocate process
  - early PCs
More sophisticated address translation

How to perform the mapping efficiently?

- So that it can be represented concisely?
- So that it can be computed quickly?
- So that it makes efficient use of the limited physical memory?
- So that multiple processes coexist in physical memory while guaranteeing isolation?
- So that it decouples the size of the virtual and physical addresses?

Ask hardware for help!
Base & Bound

- Goal: let multiple processes coexist in memory while guaranteeing isolation

- Needed hardware
  - two registers: Base and Bound (a.k.a. Limit)
  - Stored in the PCB

- Mapping
  - $pa = va + \text{Base}$
    - as long as $0 \leq va \leq \text{Bound}$
  - On context switch, change B&B (privileged instruction)
**Base & Bound**

- \( P_1 \): Base = 1000; Bound = 300
- \( P_2 \): Base = 500; Bound = 400
**Base & Bound**

- $P_1$: Base = 1000; Bound = 300
- $P_2$: Base = 500; Bound = 400

**Diagram:**
- **CPU**
- **Virtual address**
- **Bound Register**: 300
- **Base Register**: 1000
- **Memory Exception**
- **Physical address**
- **MAX$_{sys}$**
- **P1**
- **P2**: Base = 500; Bound = 400
- **150**
- **no**
- **yes**
**Base & Bound**

- **$P_1$**: Base = 1000; Bound = 300
- **$P_2$**: Base = 500; Bound = 400
**Base & Bound**

- $P_1$: Base = 1000; Bound = 300
- $P_2$: Base = 500; Bound = 400

---

**Memory Exception**

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>1000</td>
</tr>
<tr>
<td>1150</td>
<td>1300</td>
</tr>
</tbody>
</table>

**Context Switch**

- Base & Bound saved in $P_1$'s PCB
Base & Bound

- $P_1$: Base = 1000; Bound = 300
- $P_2$: Base = 500; Bound = 400

Context Switch

Virtual address

Memory Exception

Physical address

Bound Register

Base Register

1000

1300

MAX_{sys}
On Base & Bound

Contiguous Allocation

- contiguous virtual addresses are mapped to contiguous physical addresses

But mapping entire address space to physical memory

- is wasteful
  - lots of free space between heap and stack...
  - makes sharing hard
- does not work if the address space is larger than physical memory
  - think 64-bit registers...
An address space comprises multiple **segments**

- contiguous sets of virtual addresses, logically connected
  - heap, code, stack, (and also globals, libraries...)

- each segment can be of a different size

![Diagram of address space with segments]

**E Pluribus Unum**
Segmentation: Generalizing Base & Bound

- Base & Bound registers to each segment
- Each segment independently mapped to a set of contiguous addresses in physical memory
- No need to map unused virtual addresses

<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>10K</td>
<td>2K</td>
</tr>
<tr>
<td>Stack</td>
<td>28</td>
<td>2K</td>
</tr>
<tr>
<td>Heap</td>
<td>35K</td>
<td>3K</td>
</tr>
</tbody>
</table>
Segmentation

Goal: Supporting large address spaces (while allowing multiple processes to coexist in memory)

Needed hardware

- two registers (Base and Bound) per segment
  - values stored in the PCB
- if many segments, a segment table, stored in memory, at an address pointed to by a Segment Table Register (STBR)
  - process’ STBR value stored in the PCB
How do we map a virtual address to the appropriate segment?

- Read VA as having two components
  - $s$ most significant bits identify the segment
    - at most $2^s$ segments
  - $o$ remaining bits identify offset within segment
    - each segment's size can be at most $2^o$ bytes

$k = s + o$ bits
Segment Table

- Use s bits to index to the appropriate row of the segment table

<table>
<thead>
<tr>
<th>Code</th>
<th>Base</th>
<th>Bound (Max 4K)</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>32K</td>
<td>2K</td>
<td>Read/Execute</td>
</tr>
<tr>
<td>01</td>
<td>34K</td>
<td>3K</td>
<td>Read/Write</td>
</tr>
<tr>
<td>10</td>
<td>28K</td>
<td>3K</td>
<td>Read/Write</td>
</tr>
</tbody>
</table>

- Segments can be shared by different processes
  - use protection bits to determine if shared Read only (maintaining isolation) or Read/Write (if shared, no isolation)
  - processes can share code segment while keeping data private
Implementing Segmentation

Segment table generalizes Base & Bound

CPU

STBR

Segment Table Base Register

Logical addresses

Memory exception

no

yes

Bound

512

Base

40K

Physical addresses

Segment table

MAXsys

40K

0

28

R/X

512

40K

Bound

Base

Access

S

O

s

STBR

Segment Table Base Register

Logical addresses

Memory exception

no

yes

Bound

512

Base

40K

Physical addresses

Segment table

MAXsys

40K

0

28

R/X

512

40K
Revisiting `fork()`
Revisiting fork()

- Copying an entire address space can be costly...
  - especially if you proceed to obliterate it right away with exec()!
Revisiting fork():
Segments to the Rescue

Instead of copying entire address space, copy just segment table (the VA->PA mapping)

<table>
<thead>
<tr>
<th>Code</th>
<th>Base</th>
<th>Bound</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>32K</td>
<td>2K</td>
<td>RX</td>
<td></td>
</tr>
<tr>
<td>34K</td>
<td>3K</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>28K</td>
<td>3K</td>
<td>RW</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>32K</td>
</tr>
<tr>
<td>34K</td>
</tr>
<tr>
<td>28K</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>2K</td>
</tr>
<tr>
<td>3K</td>
</tr>
<tr>
<td>3K</td>
</tr>
</tbody>
</table>

Parent

Child

but change all writeable segments to Read only
Revisiting fork():
Segments to the Rescue

Instead of copying entire address space, copy just segment table (the VA->PA mapping)

but change all writeable segments to Read only

Segments in VA spaces of parent and child point to same locations in physical memory
Copy on Write (COW)

When trying to modify an address in a COW segment:

- exception!
  - exception handler copies just the affected segment, and changes both the old and new segment back to writeable

If `exec()` is immediately called, only stack segment is copied!

- it stores the return value of the `fork()` call, which is different for parent and child
Managing Free space

Many segments, different processes, different sizes

OS tracks free memory blocks ("holes")
- Initially, one big hole

Many strategies to fit segment into free memory (think "assigning classrooms to courses")
- First Fit: first big-enough hole
- Next Fit: Like First Fit, but starting from where you left off
- Best Fit: smallest big-enough hole
- Worst Fit: largest big-enough hole
External Fragmentation

- Over time, memory can become full of small holes
  - Hard to fit more segments
  - Hard to expand existing ones

Compaction

- Relocate segments to coalesce holes
External Fragmentation

- Over time, memory can become full of small holes
  - Hard to fit more segments
  - Hard to expand existing ones
- Compaction
  - Relocate segments to coalesce holes
External Fragmentation

Over time, memory can become full of small holes
- Hard to fit more segments
- Hard to expand existing ones

Compaction
- Relocate segments to coalesce holes
  - Copying eats up a lot of CPU time!
    - if 4 bytes in 10ns, 8 GB in 20s!

But what if a segment wants to grow?
Eliminating External Fragmentation: Swapping

- Preempt processes and reclaim their memory
- Move images of suspended processes to backing store