CORNELL UNDERGRADUATE RESEARCH BOARD PRESENTS

7TH ANNUAL CURBx

APRIL 7TH AT 6:00 PM ET

Featuring distinguished virtual research presentations from Cornell Undergraduates

Join the webinar at www.cornellcurb.com/curbx!
Password: CURBx

Contact Daniel Volshteyn at dlv44@cornell.edu with any questions!

cornellcurb.com  facebook.com/cornellcurb  curb_cornell
Dante Alighieri
1265-1321
700 Anniversary

John Neumeister, 1472
Cornell Library
Memory Management
(Ch. 12-17)
Abstraction is our Business

- What I have
  - A single (or a finite number) of CPUs
  - Many programs I would like to run

- What I want: a Thread
  - Each program has full control of one or more CPUs
Abstraction is our Business

What I have
- A certain amount of physical memory
- Multiple programs I would like to run
  - together, they may need more than the available physical memory

What I want: an Address Space
- Each program has as much memory as the machine’s architecture will allow to name
- All for itself
Address Space

- Set of all names used to identify and manipulate unique instances of a given resource
  - memory locations (determined by the size of the machine's word)
    - for 32-bit-register machine, the address space goes from 0x00000000 to 0xFFFFFFFF
  - memory locations (determined by the number of memory banks mounted on the machine)
  - phone numbers (XXX) (YYY-YYYY)
  - colors: R (8 bits) + G (8 bits) + B (8 bits)
Virtual Address Space: An Abstraction for Memory

- Virtual addresses start at 0
- Heap and stack can be placed far away from each other, so they can nicely grow
- Addresses are all contiguous
- Size is independent of physical memory on the machine
Physical Address Space: How memory actually looks

- Processes loaded in memory at some memory location
  - virtual address 0 is not loaded at physical address 0

- Multiple processes may be loaded in memory at the same time, and yet...

- ...physical memory may be too small to hold even a single virtual address space in its entirety
  - 64-bit, anyone?
Address Translation

A function that maps \( \langle \text{pid, virtual address} \rangle \) into a corresponding physical address

Virtual

\[ \begin{align*}
\text{P}_i \\
a486d9
\end{align*} \]

function implemented through a combination of hw and sw

Advantages:

- protection
- relocation
- data sharing
- multiplexing

Physical

\[ \begin{align*}
5e3a07
\end{align*} \]
Address Translation, Conceptually

Who does this?
Memory Management Unit (MMU)

- **Hardware device**
  - Maps virtual addresses to physical addresses

- **User process**
  - deals with *virtual* addresses
  - never sees the physical address

- **Physical memory**
  - deals with *physical* addresses
  - never sees the virtual address
Protection

The functions used by different processes map their virtual addresses to disjoint ranges of physical addresses.
Relocation

The range of the function used by a process can change over time
Relocation

The range of the function used by a process can change over time.

The same physical address can map over time to different physical addresses.

- or the mapping can be (temporarily) undefined.
Multiplexing

The set of virtual addresses that map to a given range of physical addresses can change over time.
Multiplexing

The set of virtual addresses that map to a given range of physical addresses can change over time.
Multiplexing

The set of virtual addresses that map to a given range of physical addresses can change over time.
Multiplexing

The set of virtual addresses that map to a given range of physical addresses can change over time.

\[ \mathcal{P}_i \]
Multiplexing

The set of virtual addresses that map to a given range of physical addresses can change over time.
Data Sharing

Map different virtual addresses of different processes to the same physical address
Contiguity

Contiguous virtual addresses need not map to contiguous physical addresses
Contiguity

Contiguous virtual addresses need not map to contiguous physical addresses
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

Memory Exception

Virtual address

Bound Register

Base Register

CPU

Physical address

MAX$_{sys}$
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

Physical address

MAXsys

0

1300

1000

Virtual address

P1

≤ yes

300

Bound Register

150

P2

≥ no

1000

Base Register
Base & Bound

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

Memory Exception

Virtual address

Bound Register

Base Register

Context Switch

Base & Bound saved in $P_1$'s PCB

Physical address

MAX_{sys}

1300

1000

0
Base & Bound

\[ P_1 : Base = 1000; \text{ Bound} = 300 \]
\[ P_2 : Base = 500; \text{ Bound} = 400 \]
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
Segmentation: Generalizing Base & Bound

- Base & Bound registers to each segment
- Each segment is 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
  - Stored in the PCB
- a segment table, stored in memory, at an address point 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
Segmentation: Mapping

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

$$k = s + o \text{ bits}$$
Segmentation: Mapping

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