CS162
Operating Systems and
Systems Programming
Lecture 14

Memory 1: Virtual Memory,
Segments and Page Tables

March 7th, 2023
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu
Recall: Four requirements for occurrence of Deadlock

• Mutual exclusion
  – Only one thread at a time can use a resource.

• Hold and wait
  – Thread holding at least one resource is waiting to acquire additional resources held by other threads

• No preemption
  – Resources are released only voluntarily by the thread holding the resource, after thread is finished with it

• Circular wait
  – There exists a set \( \{T_1, \ldots, T_n\} \) of waiting threads
    » \( T_1 \) is waiting for a resource that is held by \( T_2 \)
    » \( T_2 \) is waiting for a resource that is held by \( T_3 \)
    » …
    » \( T_n \) is waiting for a resource that is held by \( T_1 \)
Virtualizing Resources

• Physical Reality:
  Different Processes/Threads share the same hardware
  – Need to multiplex CPU (Just finished: scheduling)
  – Need to multiplex use of Memory (starting today)
  – Need to multiplex disk and devices (later in term)

• Why worry about memory sharing?
  – The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  – Consequently, cannot just let different threads of control use the same memory
    » Physics: two different pieces of data cannot occupy the same locations in memory
  – Probably don’t want different threads to even have access to each other’s memory if in different processes (protection)
Important Aspects of Memory Multiplexing

• Protection:
  – Prevent access to private memory of other processes
    » Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
    » Kernel data protected from User programs
    » Programs protected from themselves

• Translation:
  – Ability to translate accesses from one address space (virtual) to a different one (physical)
  – When translation exists, processor uses virtual addresses, physical memory uses physical addresses
  – Side effects:
    » Can be used to avoid overlap
    » Can be used to give uniform view of memory to programs

• Controlled overlap:
  – Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
  – Conversely, would like the ability to overlap when desired (for communication)
Alternative View: Interposing on Process Behavior

- OS interposes on process’ I/O operations
  - How? All I/O happens via syscalls.

- OS interposes on process’ CPU usage
  - How? Interrupt lets OS preempt current thread

**Question:** How can the OS interpose on process’ memory accesses?
  - Too slow for the OS to interpose every memory access
  - Translation: hardware support to accelerate the common case
  - Page fault: uncommon cases trap to the OS to handle
Recall: Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Fully describes program state
  - Program Counter, Registers, Execution Flags, Stack
- **Address space (with or w/o translation)**
  - Set of memory addresses accessible to program (for read or write)
  - May be distinct from memory space of the physical machine (in which case programs operate in a virtual address space)
- **Process: an instance of a running program**
  - Protected Address Space + One or more Threads
- **Dual mode operation / Protection**
  - Only the “system” has the ability to access certain resources
  - Combined with translation, isolates programs from each other and the OS from programs
THE BASICS: Address/Address Space

- What is $2^{10}$ bytes (where a byte is abbreviated as “B”)?
  - $2^{10} B = 1024 B = 1 \text{ KB}$ (for memory, $1K = 1024$, not 1000)
- How many bits to address each byte of 4KB page?
  - $4\text{KB} = 4 \times 1\text{KB} = 4 \times 2^{10} = 2^{12} \Rightarrow 12 \text{ bits}$
- How much memory can be addressed with 20 bits? 32 bits? 64 bits?
  - Use $2^k$
Address Space, Process Virtual Address Space

• Definition: **Set of accessible addresses and the state associated with them**
  – $2^{32} \approx 4$ billion **bytes** on a 32-bit machine

• How many 32-bit numbers fit in this address space?
  – 32-bits = 4 bytes, so $2^{32}/4 = 2^{30} \approx$ 1 billion

• What happens when processor reads or writes to an address?
  – Perhaps acts like regular memory
  – Perhaps causes I/O operation
    » (Memory-mapped I/O)
  – Causes program to abort (segfault)?
  – Communicate with another program
  – …
Recall: Process Address Space: typical structure
Recall: Uniprogramming

- Uniprogramming (no Translation or Protection)
  - Application always runs at the same place in physical memory since only one application at a time
  - Application can access any physical address
  - Application given illusion of dedicated machine by giving it reality of a dedicated machine

![Diagram showing valid 32-bit addresses ranging from 0x00000000 to 0xFFFFFFFF]
Primitive Multiprogramming

- Multiprogramming without Translation or Protection
  - Must somehow prevent address overlap between threads

- Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
  - Everything adjusted to memory location of program
  - Translation done by a linker-loader (relocation)
  - Common in early days (… till Windows 3.x, 95?)

- With this solution, no protection: bugs in any program can cause other programs to crash or even the OS
Binding of Instructions and Data to Memory

Process view of memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
<td>0000</td>
</tr>
<tr>
<td>0x0900</td>
<td>8C2000C0</td>
</tr>
<tr>
<td>0x0904</td>
<td>0C000280</td>
</tr>
<tr>
<td>0x0908</td>
<td>2021FFFF</td>
</tr>
<tr>
<td>0x090C</td>
<td>14200242</td>
</tr>
<tr>
<td>0x0910</td>
<td>...</td>
</tr>
<tr>
<td>0x0A00</td>
<td>...</td>
</tr>
</tbody>
</table>

Physical addresses

Assume 4byte words

- $0x300 = 4 \times 0xC0$
- $0x0C0 = 0000\ 1100\ 0000$
- $0x300 = 0011\ 0000\ 0000$
Binding of Instructions and Data to Memory

Process view of memory

```
data1:  dw  32
start: lw  r1,0(data1)
jal  checkit
loop:  addi r1, r1, -1
bnz  r1, loop
...
checkit: ...
```

Physical addresses

```
0x0300  0000020
0x0900  8C2000C0
0x0906  0C000280
0x0908  2021FFFF
0x90C   14200242
0xA00   ...
```

Physical Memory

```
0x0000  00000020
0x0300  0C2000C0
0x0900  0C000340
0x900   2021FFFF
0x90C   14200242
0xFFFF  ...
```
Second copy of program from previous example

Process view of memory

- `data1: dw 32`
- `start: lw r1,0(data1)`
- `jal checkit`
- `loop: addi r1, r1, -1`
- `bnz r1, loop`
- `checkit: ...`

Physical addresses

```
0x0300 00000020
0x0900 8C2000C0
0x0904 0C000280
0x0908 2021FFFF
0x090C 14200242
0x0A00
0xFFFF
```

Need address translation!
Second copy of program from previous example

Process view of memory

```
data1:    dw    32
start:    lw    r1,0(data1)
        jal   checkit
loop:     addi  r1, r1, -1
        bnz   r1, loop
...
checkit:  ...
```

Physical addresses

```
0x1300  00000020
0x1900  8C2004C0
0x1904  0C000680
0x1908  2021FFFF
0x190C  14200642
...
0x1A00
```

• One of many possible translations!
• Where does translation take place?

Compile time, Link/Load time, or Execution time?
From Program to Process

• Preparation of a program for execution involves components at:
  – Compile time (i.e., “gcc”)
  – Link/Load time (UNIX “ld” does link)
  – Execution time (e.g., dynamic libs)

• Addresses can be bound to final values anywhere in this path
  – Depends on hardware support
  – Also depends on operating system

• Dynamic Libraries
  – Linking postponed until execution
  – Small piece of code (i.e. the stub), locates appropriate memory-resident library routine
  – Stub replaces itself with the address of the routine, and executes routine
Administrivia

• Midterm 2: Wednesday 3/15 from 8-10PM
  – A week from tomorrow!!!
  – All material up to Lecture 16 technically in bounds
• Homework 4 coming out
  – Released tomorrow, Wednesday 3/08
• Project 2 design document due this Friday!
Administrivia (Con’t)

- You need to know your units as CS/Engineering students!

- Units of Time: “s”: Second, “min”: 60s, “h”: 3600s, (of course)
  - Millisecond: \(1\text{ms} \Rightarrow 10^{-3}\text{s}\)
  - Microsecond: \(1\mu\text{s} \Rightarrow 10^{-6}\text{s}\)
  - Nanosecond: \(1\text{ns} \Rightarrow 10^{-9}\text{s}\)
  - Picosecond: \(1\text{ps} \Rightarrow 10^{-12}\text{s}\)

- Integer Sizes: “b” \(\Rightarrow\) ”bit”, “B” \(\Rightarrow\) “byte” == 8 bits, “W” \(\Rightarrow\)”word”==? (depends. Could be 16b, 32b, 64b)

- Units of Space (memory), sometimes called the “binary system”
  - Kilo: \(1\text{KB} \equiv 1\text{KiB} \Rightarrow 1024\text{ bytes} \Rightarrow \approx 1024 \approx 1.0 \times 10^3\)
  - Mega: \(1\text{MB} \equiv 1\text{MiB} \Rightarrow (1024)^2\text{ bytes} \Rightarrow \approx 1,048,576 \approx 1.0 \times 10^6\)
  - Giga: \(1\text{GB} \equiv 1\text{GiB} \Rightarrow (1024)^3\text{ bytes} \Rightarrow \approx 1,073,741,824 \approx 1.1 \times 10^9\)
  - Tera: \(1\text{TB} \equiv 1\text{TiB} \Rightarrow (1024)^4\text{ bytes} \Rightarrow \approx 1,099,511,627,776 \approx 1.1 \times 10^{12}\)
  - Peta: \(1\text{PB} \equiv 1\text{PiB} \Rightarrow (1024)^5\text{ bytes} \Rightarrow \approx 1,125,899,906,842,624 \approx 1.1 \times 10^{15}\)
  - Exa: \(1\text{EB} \equiv 1\text{EiB} \Rightarrow (1024)^6\text{ bytes} \Rightarrow \approx 1,152,921,504,606,846,976 \approx 1.2 \times 10^{18}\)

- Units of Bandwidth, Space on disk/etc, Everything else…., sometimes called the “decimal system”
  - Kilo: \(1\text{KB/s} \Rightarrow 10^3\text{ bytes/s}, \quad 1\text{KB} \Rightarrow 10^3\text{ bytes}\)
  - Mega: \(1\text{MB/s} \Rightarrow 10^6\text{ bytes/s}, \quad 1\text{MB} \Rightarrow 10^6\text{ bytes}\)
  - Giga: \(1\text{GB/s} \Rightarrow 10^9\text{ bytes/s}, \quad 1\text{GB} \Rightarrow 10^9\text{ bytes}\)
  - Tera: \(1\text{TB/s} \Rightarrow 10^{12}\text{ bytes/s}, \quad 1\text{TB} \Rightarrow 10^{12}\text{ bytes}\)
  - Peta: \(1\text{PB/s} \Rightarrow 10^{15}\text{ bytes/s}, \quad 1\text{PB} \Rightarrow 10^{15}\text{ bytes}\)
  - Exa: \(1\text{EB/s} \Rightarrow 10^{18}\text{ bytes/s}, \quad 1\text{EB} \Rightarrow 10^{18}\text{ bytes}\)
Multiprogramming with Protection

- Can we protect programs from each other without translation?
  - Yes: **Base and Bound!**
  - Used by, e.g., **Cray-1 supercomputer**
Recall: Base and Bound (No Translation)

- Still protects OS and isolates program
- Requires relocating loader
- No addition on address path
Recall: General Address translation

- Consequently, two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (Memory Management Unit or MMU) converts between two views
- Translation ⇒ much easier to implement protection!
  - If task A cannot even gain access to task B’s data, no way for A to adversely affect B
- With translation, every program can be linked/loaded into same region of user address space
Recall: Base and Bound (with Translation)

- Hardware relocation
- Can the program touch OS?
- Can it touch other programs?
Issues with Simple B&B Method

- Fragmentation problem over time
  - Not every process is same size \(\Rightarrow\) memory becomes fragmented over time

- Missing support for sparse address space
  - Would like to have multiple chunks/program (Code, Data, Stack, Heap, etc)

- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by providing multiple segments per process
More Flexible Segmentation

- Logical View: multiple separate segments
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc
- Each segment is given region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory
Implementation of Multi-Segment Model

- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    » x86 Example: mov [es:bx],ax.
- What is “V/N” (valid / not valid)?
  - Can mark segments as invalid; requires check as well
Intel x86 Special Registers

• Typical Segment Register
  – Current Priority is RPL of Code Segment (CS)
• Segmentation can’t be just “turned off”
  – What if we just want to use paging?
  – Set base and bound to all of memory, in all segments
Example: Four Segments (16 bit addresses)

<table>
<thead>
<tr>
<th>Seg ID #</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Virtual Address Format

![Virtual Address Space](image)

Physical Address Space

![Physical Address Space](image)
Example: Four Segments (16 bit addresses)

<table>
<thead>
<tr>
<th>Seg ID #</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Virtual Address Format

Virtual Address Space

Physical Address Space

SegID = 0
Example: Four Segments (16 bit addresses)

<table>
<thead>
<tr>
<th>Seg ID #</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Virtual Address Format

Virtual Address Space

Physical Address Space
Example: Four Segments (16 bit addresses)

<table>
<thead>
<tr>
<th>Seg ID #</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Virtual Address Format

Virtual Address Space

Physical Address Space

Might be shared

Space for Other Apps

Shared with Other Apps
Let's simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get “la $a0, varx”
   Move 0x4050 $a0, Move PC+4→PC

---

Example of Segment Translation (16bit address)

<table>
<thead>
<tr>
<th>Seg ID #</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>
Let’s simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get “la $a0, varx”
   Move 0x4050 \rightarrow $a0, Move PC+4 \rightarrow PC

2. Fetch 0x0244. Translated to Physical=0x4244. Get “jal strlen”
   Move 0x0248 \rightarrow $ra (return address!), Move 0x0360 \rightarrow PC
Let’s simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get “la $a0, varx”
   Move 0x4050 → $a0, Move PC+4→PC

2. Fetch 0x0244. Translated to Physical=0x4244. Get “jal strlen”
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC

3. Fetch 0x0360. Translated to Physical=0x4360. Get “li $v0, 0”
   Move 0x0000 → $v0, Move PC+4→PC
Let's simulate a bit of this code to see what happens (PC=0x0240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get “la $a0, varx”
   Move 0x4050 → $a0, Move PC+4→PC

2. Fetch 0x0244. Translated to Physical=0x4244. Get “jal strlen”
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC

3. Fetch 0x0360. Translated to Physical=0x4360. Get “li $v0, 0”
   Move 0x0000 → $v0, Move PC+4→PC

4. Fetch 0x0364. Translated to Physical=0x4364. Get “lb $t0, ($a0)”
   Since $a0 is 0x4050, try to load byte from 0x4050
   Translate 0x4050 (0100 0000 0101 0000). Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
   Load Byte from 0x4850→$t0, Move PC+4→PC
Observations about Segmentation

• Translation on every instruction fetch, load or store
• Virtual address space has holes
  – Segmentation efficient for sparse address spaces
• When it is OK to address outside valid range?
  – This is how the stack (and heap?) allowed to grow
  – For instance, stack takes fault, system automatically increases size of stack
• Need protection mode in segment table
  – For example, code segment would be read-only
  – Data and stack would be read-write (stores allowed)
• What must be saved/restored on context switch?
  – Segment table stored in CPU, not in memory (small)
  – Might store all of processes memory onto disk when switched (called “swapping”)
What if not all segments fit in memory?

- Extreme form of Context Switch: **Swapping**
  - To make room for next process, some or all of the previous process is moved to disk
    » Likely need to send out complete segments
  - This greatly increases the cost of context-switching
- What might be a desirable alternative?
  - Some way to keep only active portions of a process in memory at any one time
  - Need finer granularity control over physical memory
Problems with Segmentation

• Must fit variable-sized chunks into physical memory

• May move processes multiple times to fit everything

• Limited options for swapping to disk

• Fragmentation: wasted space
  – External: free gaps between allocated chunks
  – Internal: don’t need all memory within allocated chunks
Recall: General Address Translation

Translation Map 1

Translation Map 2

Physical Address Space
Paging: Physical Memory in Fixed Size Chunks

• Solution to fragmentation from segments?
  – Allocate physical memory in **fixed size** chunks (“pages”)
  – Every chunk of physical memory is equivalent
    » Can use simple vector of bits to handle allocation:
      00110001110001101 ... 110010
    » Each bit represents page of physical memory
      1 $\Rightarrow$ allocated, 0 $\Rightarrow$ free

• Should pages be as big as our previous segments?
  – No: Can lead to lots of internal fragmentation
    » Typically have small pages (1K-16K)
  – Consequently: need multiple pages/segment
How to Implement Simple Paging?

- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page (e.g. Valid bits, Read, Write, etc)
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    » Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions
Simple Page Table Example

Virtual Memory

Physical Memory

Example (4 byte pages)

Page Table

Virtual Memory

Physical Memory

Page Table

0x00

0x04

0x08

0x09?

0x05!

0x0E!
What about Sharing?

Virtual Address (Process A):

<table>
<thead>
<tr>
<th>Page Table Pointer A</th>
<th>Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>page #0</td>
<td>V,R</td>
<td></td>
</tr>
<tr>
<td>page #1</td>
<td>V,R</td>
<td></td>
</tr>
<tr>
<td>page #2</td>
<td>V,R,W</td>
<td></td>
</tr>
<tr>
<td>page #3</td>
<td>V,R,W</td>
<td></td>
</tr>
<tr>
<td>page #4</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>page #5</td>
<td>V,R,W</td>
<td></td>
</tr>
</tbody>
</table>

Virtual Address (Process B):

<table>
<thead>
<tr>
<th>Page Table Pointer B</th>
<th>Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>page #0</td>
<td>V,R</td>
<td></td>
</tr>
<tr>
<td>page #1</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>page #2</td>
<td>V,R,W</td>
<td></td>
</tr>
<tr>
<td>page #3</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>page #4</td>
<td>V,R</td>
<td></td>
</tr>
<tr>
<td>page #5</td>
<td>V,R,W</td>
<td></td>
</tr>
</tbody>
</table>

Shared Page

This physical page appears in address space of both processes.
Where is page sharing used?

• The “kernel region” of every process has the same page table entries
  – The process cannot access it at user level
  – But on U->K switch, kernel code can access it AS WELL AS the region for THIS user
    » What does the kernel need to do to access other user processes?
• Different processes running same binary!
  – Execute-only, but do not need to duplicate code segments
• User-level system libraries (execute only)
• Shared-memory segments between different processes
  – Can actually share objects directly between processes
    » Must map page into same place in address space!
  – This is a limited form of the sharing that threads have within a single process
Memory Layout for Linux 32-bit (Pre-Meltdown patch!)

http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png
Some simple security measures

• Address Space Randomization
  – Position-Independent Code ⇒ can place user code anywhere in address space
    » Random start address makes much harder for attacker to cause jump to code that it
      seeks to take over
  – Stack & Heap can start anywhere, so randomize placement

• Kernel address space isolation
  – Don’t map whole kernel space into each process, switch to kernel page table
  – Meltdown⇒map none of kernel into user mode!

![Kernel page-table isolation diagram]
Summary: Paging

Virtual memory view:
- Stack: 1111 111
- Heap: 1111 0000
- Data: 1000 0000
- Code: 0100 0000

Page Table:
- Page 1111: Stack
- Page 1110: Heap
- Page 1100: Code
- Page 1000: Data

Physical memory view:
- Stack: Page Table: 1110 0000
- Heap: Page Table: 0111 0000
- Data: Page Table: 0101 0000
- Code: Page Table: 0001 0000

Page # and Offset:
- Stack: 0000 0000
- Heap: 0001 0000
- Data: 0010 0000
- Code: 0011 0000
Summary: Paging

What happens if stack grows to 1110 0000?
Summary: Paging

Virtual memory view

<table>
<thead>
<tr>
<th>Page #</th>
<th>Offset</th>
<th>Stack</th>
<th>Heap</th>
<th>Data</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>1111</td>
<td>1110</td>
<td>1100</td>
<td>1000</td>
<td>0100</td>
</tr>
<tr>
<td>1110</td>
<td>0000</td>
<td>1110</td>
<td>1100</td>
<td>1000</td>
<td>0100</td>
</tr>
<tr>
<td>1100</td>
<td>0000</td>
<td>1110</td>
<td>1100</td>
<td>1000</td>
<td>0100</td>
</tr>
<tr>
<td>1000</td>
<td>0000</td>
<td>1110</td>
<td>1100</td>
<td>1000</td>
<td>0100</td>
</tr>
</tbody>
</table>

Page Table

<table>
<thead>
<tr>
<th>Page #</th>
<th>Offset</th>
<th>Stack</th>
<th>Heap</th>
<th>Data</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>1111</td>
<td>1110</td>
<td>1100</td>
<td>1000</td>
<td>0100</td>
</tr>
<tr>
<td>1110</td>
<td>0000</td>
<td>1110</td>
<td>1100</td>
<td>1000</td>
<td>0100</td>
</tr>
<tr>
<td>1100</td>
<td>0000</td>
<td>1110</td>
<td>1100</td>
<td>1000</td>
<td>0100</td>
</tr>
<tr>
<td>1000</td>
<td>0000</td>
<td>1110</td>
<td>1100</td>
<td>1000</td>
<td>0100</td>
</tr>
</tbody>
</table>

Physical memory view

<table>
<thead>
<tr>
<th>Page #</th>
<th>Offset</th>
<th>Stack</th>
<th>Heap</th>
<th>Data</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>1111</td>
<td>1110</td>
<td>1100</td>
<td>1000</td>
<td>0100</td>
</tr>
<tr>
<td>1110</td>
<td>0000</td>
<td>1110</td>
<td>1100</td>
<td>1000</td>
<td>0100</td>
</tr>
<tr>
<td>1100</td>
<td>0000</td>
<td>1110</td>
<td>1100</td>
<td>1000</td>
<td>0100</td>
</tr>
<tr>
<td>1000</td>
<td>0000</td>
<td>1110</td>
<td>1100</td>
<td>1000</td>
<td>0100</td>
</tr>
</tbody>
</table>

Allocate new pages where room!
Conclusion

- **Segment Mapping**
  - Segment registers within processor
  - Segment ID associated with each access
    - Often comes from portion of virtual address
    - Can come from bits in instruction instead (x86)
  - Each segment contains base and limit information
    - Offset (rest of address) adjusted by adding base

- **Page Tables**
  - Memory divided into fixed-sized chunks of memory
  - Virtual page number from virtual address mapped through page table to physical page number
  - Offset of virtual address same as physical address
  - Large page tables can be placed into virtual memory

- **Next Time: Multi-Level Tables**
  - Virtual address mapped to series of tables
  - Permit sparse population of address space