Virtualizing Resources

- Physical Reality: Different Processes/Threads share the same hardware
  - Need to multiplex CPU (just finished: scheduling)
  - Need to multiplex use of Memory (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, two different threads cannot use the same memory
    » Physics: two different data cannot occupy same locations in memory
  - May not want different threads to have access to each other’s memory

Next Objective

- Dive deeper into the concepts and mechanisms of memory sharing and address translation
- Enabler of many key aspects of operating systems
  - Protection
  - Multi-programming
  - Isolation
  - Memory resource management
  - I/O efficiency
  - Sharing
  - Inter-process communication
  - Demand paging
- Today: Linking, Segmentation

Recall: Single and Multithreaded Processes

- Threads encapsulate concurrency
  - “Active” component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
  - “Passive” component of a process
Important Aspects of Memory Multiplexing (1/2)

• 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)

• 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

Important Aspects of Memory Multiplexing (2/2)

• 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

Recall: Loading


Binding of Instructions and Data to Memory

Process view of memory

Assume 4-byte words

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
<td>0x0C0</td>
</tr>
<tr>
<td>0x0C0</td>
<td>0x0000</td>
</tr>
<tr>
<td>0x0900</td>
<td>0x8C20</td>
</tr>
<tr>
<td>0x0904</td>
<td>0x02B0</td>
</tr>
<tr>
<td>0x0908</td>
<td>2021FFF0</td>
</tr>
<tr>
<td>0x090C</td>
<td>14280242</td>
</tr>
<tr>
<td>0x0A00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>data1: dw 32</td>
<td></td>
</tr>
<tr>
<td>start: lw r1,0(data1)</td>
<td></td>
</tr>
<tr>
<td>jal checkit</td>
<td></td>
</tr>
<tr>
<td>loop: addi r1, r1, -1</td>
<td></td>
</tr>
<tr>
<td>bnz r1, loop</td>
<td></td>
</tr>
<tr>
<td>checkit: ...</td>
<td></td>
</tr>
</tbody>
</table>
**Binding of Instructions and Data to Memory**

Process view of memory

<table>
<thead>
<tr>
<th>Physical addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>0x0900</code></td>
</tr>
<tr>
<td><code>0x0904</code></td>
</tr>
<tr>
<td><code>0x0908</code></td>
</tr>
<tr>
<td><code>0x090C</code></td>
</tr>
<tr>
<td><code>0x0910</code></td>
</tr>
<tr>
<td><code>0x0914</code></td>
</tr>
</tbody>
</table>

Data1: `dw 32`

Start: `lw $r1,0(data1)`
Jal: `checkit`

Loop: `addi $r1, $r1, -1`
Bnz: `$r1, loop`

Checkit: `_`

**Second copy of program from previous example**

Process view of memory

<table>
<thead>
<tr>
<th>Physical addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>0x0900</code></td>
</tr>
<tr>
<td><code>0x0904</code></td>
</tr>
<tr>
<td><code>0x0908</code></td>
</tr>
<tr>
<td><code>0x090C</code></td>
</tr>
<tr>
<td><code>0x0910</code></td>
</tr>
<tr>
<td><code>0x0914</code></td>
</tr>
</tbody>
</table>

Data1: `dw 32`

Start: `lw $r1,0(data1)`
Jal: `checkit`

Loop: `addi $r1, $r1, -1`
Bnz: `$r1, r0, loop`

Checkit: `_`

Need address translation!

**Multi-step Processing of a Program for Execution**

- 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, stub, used to locate appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine
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

Multiprogramming (primitive stage)

- 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

Multiprogramming (Version with Protection)

- Can we protect programs from each other without translation?
  - Yes: use two special registers BaseAddr and LimitAddr to prevent user from straying outside designated area
    - If user tries to access an illegal address, cause an error
  - During switch, kernel loads new base/limit from PCB (Process Control Block)
    - User not allowed to change base/limit registers

Recall: General Address translation

- Recall: Address Space:
  - All the addresses and state a process can touch
  - Each process and kernel has different address space
- Consequently, two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (MMU) converts between the two views
- Translation makes it 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
Simple Example: Base and Bounds (CRAY-1)

- Could use base/bounds for dynamic address translation – translation happens at execution:
  - Alter address of every load/store by adding "base"
  - Generate error if address bigger than limit
- This gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
  - Program gets continuous region of memory
  - Addresses within program do not have to be relocated when program placed in different region of DRAM

Issues with Simple B&B Method

- Fragmentation problem over time
  - Not every process is same size → memory becomes fragmented
- Missing support for sparse address space
  - Would like to have multiple chunks/program (Code, Data, Stack)
- 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 [edx+],ax
- What is “V/N” (valid / not valid)?
  - Can mark segments as invalid; requires check as well
Typical Segment Register
Current Priority is RPL
Of Code Segment (CS)

Intel x86 Special Registers

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>
**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>

**Example of Segment Translation (16b address)**

```assembly
0x240
main: la $a0, varx
0x244
jal strlen
0x368
strlen: li $v0, 0 ;count
0x364
loop: lb $t0, ($a0)
0x368
beq $r0, $t0, done
0x4050 varx dw 0x314159
```

**Administrivia**

- Midterm #1 regrades open until Thursday 3/9 11:59PM
- Final report due today!
1. Let’s simulate a bit of this code to see what happens (PC=0x240):
   - Fetch 0x240. 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
   - Fetch 0x244. Translated to Physical=0x4244. Get “jal strlcn”
     Move 0x0248 → $ra (return address!), Move 0x0360 → PC

2. Fetch 0x240. 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 0x244. Translated to Physical=0x4244. Get “jal strlcn”
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC

3. Fetch 0x360. 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=0x240):
1. Fetch 0x240. 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 0x244. Translated to Physical=0x4244. Get “jal strlcn”
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC
3. Fetch 0x360. 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=0x240):
1. Fetch 0x240. 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 0x244. Translated to Physical=0x4244. Get “jal strlcn”
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC
3. Fetch 0x360. Translated to Physical=0x4360. Get “li $v0, 0”
   Move 0x0000 → $v0, Move PC+4→PC

Observations about Segmentation

- Virtual address space has holes
  - Segmentation efficient for sparse address spaces
  - A correct program should never address gaps (except as mentioned in moment)
    » If it does, trap to kernel and dump core
- When it is OK to address outside valid range:
  - This is how the stack and heap are 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)
  - Shared segment could be read-only or read-write
- 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”)
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

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 ⇒ allocated, 0 ⇒ 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
Simple Page Table Example

Virtual Memory 0x00 ----> 0x001110
Physical Memory

Virtual memory view
Stack (stack) 1111 1111
Memory Mapping Support
File mappings including memory protection
Program text
Data
Stack
Heap
Shared page

Virtual Address (Process A):

Virtual Page # Offset
Page TablePtrA

This physical page appears in address space of both processes

Virtual Address (Process B):

Virtual Page # Offset
Page TablePtrB

What about Sharing?

Offset Virtual Page # Virtual Address
(Process A):
Page TablePtrA

Page TablePtrB

Offset Virtual Page # Virtual Address
(Process B):

Summary: Paging

Virtual memory view
Page Table
Physical memory view

0000 0000
0001 0000
0100 0000
0101 0000
0110 0000
0111 0000
1000 0000
1001 0000
1010 0000
1011 0000
1100 0000
1101 0000
1110 0000
1111 0000

0000 0000
0001 0000
0010 0000
0011 0000
0100 0000
0101 0000
0110 0000
0111 0000
1000 0000
1001 0000
1010 0000
1011 0000
1100 0000
1101 0000
1110 0000
1111 0000

0000 0000
0001 0000
0010 0000
0011 0000
0100 0000
0101 0000
0110 0000
0111 0000
1000 0000
1001 0000
1010 0000
1011 0000
1100 0000
1101 0000
1110 0000
1111 0000

Memory Layout for Linux 32-bit

http://static.duarte.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png
Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit

- Analysis
  - Pros
    » Simple memory allocation
    » Easy to share
  - Con: What if address space is sparse?
    » E.g., on UNIX, code starts at 0, stack starts at \((2^{31}-1)\)
    » With 1K pages, need 2 million page table entries!
  - Con: What if table really big?
    » Not all pages used all the time ⇒ would be nice to have working set of page table in memory

- How about combining paging and segmentation?
Multi-level Translation: Segments + Pages

- What about a tree of tables?
  - Lowest level page table ⇒ memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):

  - What must be saved/restored on context switch?
    - Contents of top-level segment registers (for this example)
    - Pointer to top-level table (page table)
### Multi-level Translation Analysis

**Pros:**
- Only need to allocate as many page table entries as we need for application
  - In other words, sparse address spaces are easy
- Easy memory allocation
- Easy Sharing
  - Share at segment or page level (need additional reference counting)

**Cons:**
- One pointer per page (typically 4K – 16K pages today)
- Page tables need to be contiguous
  - However, previous example keeps tables to exactly one page in size
- Two (or more, if >2 levels) lookups per reference
  - Seems very expensive!

### Summary

**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

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