EECS 482
Introduction to Operating Systems

Winter 2018

Harsha V. Madhyastha
Address Spaces

- Hardware interface:
  - All processes share physical memory

- OS abstraction:
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

```java
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:**
  - No virtual memory
  - External fragmentation
  - Cannot grow parts of address space independently
  - No controlled sharing

- **Cause:** *Address space is indivisible unit*
Segmentation

- Divide address space into segments
- Segment: region of memory contiguous in both physical memory and in virtual address space
Segmentation

virtual memory segment 3

virtual memory segment 1

virtual memory segment 0

physical memory

code

stack

data

code

stack

data

EECS 482 – Lecture 12
Segmentation

<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>n/a</td>
<td>n/a</td>
<td>unused</td>
</tr>
<tr>
<td>3</td>
<td>2000</td>
<td>1000</td>
<td>stack segment</td>
</tr>
</tbody>
</table>

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

- **Pros:**
  - Can grow each segment independently
  - Can share segments across address spaces

- **Cons:**
  - Every segment must be smaller than phys. memory
  - Segment allocation is hard
  - External fragmentation

- **Cause:** Variable amount of contiguous memory
Paging

- Allocate phys. memory in fixed-size units (pages)
  - Any free physical page can store any virtual page
Paging

- 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>

Why no column for bound?

Function to translate VA to PA?

- 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)
Page Lookups

Virtual Address

Page number  Offset

Page Table

Phys page #

Physical Address

Page number  Offset

Physical Memory
Paging

- Translating virtual address to physical address

```cpp
if (virtual page is invalid) {
    trap to OS fault handler
} else {
    physical page # = pageTable[virtual page #].physPageNum
    physical address = {Physical page #}{offset}
}
```

- What must be changed on a context switch?
  - Indirection via Page Table Base Register
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)

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

- Revised page table:

<table>
<thead>
<tr>
<th>Virtual page #</th>
<th>Physical page #</th>
<th>Resident</th>
<th>Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>105</td>
<td>0</td>
<td>RX</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>1</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>283</td>
<td>1</td>
<td>RW</td>
</tr>
<tr>
<td>3</td>
<td>invalid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>invalid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1048575</td>
<td>invalid</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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?
Picking Page Size

- What happens if page size is really small?
- What happens if page size is really big?

- Typically a compromise, e.g., 4 KB or 8 KB
  - Some architectures support multiple page sizes
Growing Address Space

<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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>...</th>
<th>invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1048572</td>
<td>invalid</td>
</tr>
<tr>
<td>1048573</td>
<td>1078</td>
</tr>
<tr>
<td>1048574</td>
<td>48136</td>
</tr>
<tr>
<td>1048575</td>
<td>60</td>
</tr>
</tbody>
</table>

Why less space wastage than base and bounds?
Paging

● Pros
  ♦ Simple memory allocation
  ♦ Flexible sharing
  ♦ Easy to grow address space

● Cons
  ♦ 32-bit virtual address, 4 KB pages, 4 byte PTEs
  ♦ Page table size?
Page table size

- 32-bit address $\rightarrow 2^{32}$ unique addresses
- 4 KB page $\rightarrow (2^{32})/4$ KB = $2^{20}$ virtual pages
- 4 bytes per PTE $\rightarrow$ 4 MB page table
- 25 processes $\rightarrow$ 100 MB for page tables!

How to reduce page table overhead?
Project 2

- Due on Saturday!
  - 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

- Think about why your thread library may cause a program to run for longer than a correct library
Mid-Term

- Next Tuesday!
  - Covers all material up to and including deadlock

- Email me if you would like me to review any material on Wednesday

- Review of sample exams on Sunday (Feb 18th)
  - Important to try questions yourself first!
Multi-level Paging

- Standard page table is a simple array
- Multi-level paging generalizes this into a tree

Example: Two-level page table with 4KB pages
  - Index into level 1 page table: virtual address bits 31-22
  - Index into level 2 page table: virtual address bits 21-12
  - Page offset: bits 11-0
Multi-level Paging

level 1 page table

0 1 2 3

level 2 page tables

virtual address
bits 21-12

0 1 2 3

physical page #

0 10 20 2

1 15 3

2 20

3 3

virtual address
bits 21-12

0 1 2 3

physical page #

0 30

1 4

2 8

3 3

How does this let translation data take less space?
Sparse Address Space

<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>1048572</td>
<td>invalid</td>
</tr>
<tr>
<td>1048573</td>
<td>1078</td>
</tr>
<tr>
<td>1048574</td>
<td>48136</td>
</tr>
<tr>
<td>1048575</td>
<td>60</td>
</tr>
</tbody>
</table>
Sparse Address Space

<table>
<thead>
<tr>
<th>Bits 31-22</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xffffffff389</td>
</tr>
<tr>
<td>1</td>
<td>Invalid</td>
</tr>
<tr>
<td>2</td>
<td>Invalid</td>
</tr>
<tr>
<td>...</td>
<td>Invalid</td>
</tr>
<tr>
<td>1021</td>
<td>Invalid</td>
</tr>
<tr>
<td>1022</td>
<td>Invalid</td>
</tr>
<tr>
<td>1023</td>
<td>0xffffffff7046</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 21-12</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>1021</td>
<td>invalid</td>
</tr>
<tr>
<td>1020</td>
<td>invalid</td>
</tr>
<tr>
<td>1021</td>
<td>1078</td>
</tr>
<tr>
<td>1022</td>
<td>48136</td>
</tr>
<tr>
<td>1023</td>
<td>60</td>
</tr>
</tbody>
</table>
Multi-level paging

- How to share memory between address spaces?
- What must be changed on a context switch?

**Pros**
- Easy memory allocation
- Flexible sharing
- Space efficient for sparse address spaces

**Cons**
- Two or more extra lookups per memory reference
Translation lookaside buffer

- TLB caches virtual page # to PTE mapping
  - Cache hit → Skip all the translation steps
  - Cache miss → Get PTE, store in TLB, restart instruction

- Does TLB change what happens on a context switch?
End-to-end look at paging

- New process \(\rightarrow\) allocate new L1 page table
  - All entries in L1 page table invalid
- As process makes virtual pages valid, allocate new L2 page tables and add entries
- To serve load/store on a virtual page:
  - CPU looks up TLB to find PTE for virtual page #
  - If absent, lookup PTE in memory and load TLB
- When process ends, deallocate L1 and L2 page tables
Page replacement

- Not at all valid pages can be in physical memory
- How to handle loads/stores on non-resident pages?