[537] Virtual Memory

Tyler Harter
9/15/14
Overview

Review Scheduling

Address Spaces (Chapter 13)

Address Translation (Chapter 15)

Segmentation (Chapter 16)
Review: Schedulers
Scheduling Basics

**Workloads:**
- arrival_time
- run_time

**Schedulers:**
- FIFO
- SJF
- STCF
- RR

**Metrics:**
- turnaround_time
- response_time
Scheduling Basics

**Workloads:**
- arrival_time
- run_time

**Schedulers:**
- FIFO
- SJF
- STCF
- RR

**Metrics:**
- turnaround_time
- response_time

Project grading will be based on turnaround time!
Workload

<table>
<thead>
<tr>
<th>JOB</th>
<th>arrival</th>
<th>run</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>40</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

Schedulers:

- FIFO
- SJF
- STCF
- RR

Timelines

ABCABCABABAAAA

B C A

A B C

A B C
### Workload

<table>
<thead>
<tr>
<th>JOB</th>
<th>arrival</th>
<th>run</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>40</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

### Schedulers:
- FIFO
- SJF
- STCF
- RR

### Timelines

- **RR**
  - Jobs: ABCABCABABAAAAA
  - Timeline:
    - B: 0-20
    - C: 20-40
    - A: 40-80

- **SJF**
  - Jobs: BCA
  - Timeline:
    - B: 0-20
    - C: 20-40
    - A: 40-60

- **STCF**
  - Jobs: BCBA
  - Timeline:
    - B: 0-20
    - C: 20-40
    - A: 40-60

- **FIFO**
  - Jobs: ABC
  - Timeline:
    - A: 0-40
    - B: 40-60
    - C: 60-80
Lottery Scheduler

Job A (1)
Job B (1)
Job C (100)
Job D (200)
Job E (100)

402 total tickets
Job A (1)
Job B (1)
Job C (100)
Job D (200)
Job E (100)

winner = random(402)

402 total tickets
Job A (1)
Job B (1)
Job C (100)
Job D (200)
Job E (100)

402 total tickets

winner = 102
Job A (1)  Job B (1)  Job C (100)  Job D (200)  Job E (100)

is $102 < 1$?

winner = 102

402 total tickets
Job A (1) | Job B (1) | Job C (100) | Job D (200) | Job E (100)

winner = 101

is 101 < 1 ?

402 total tickets
winner = 100

is 100 < 100 ?

Job A (1)  Job B (1)  Job C (100)  Job D (200)  Job E (100)

402 total tickets
winner = 0

is 0 < 200 ?

Job A (1)  Job B (1)  Job C (100)  Job D (200)  Job E (100)

402 total tickets
Run D!

is $0 < 200$ ?

402 total tickets
Address Spaces
More Virtualization

Virtual CPU: *illusion* of private CPU registers
- 2 lectures

Virtual RAM: *illusion* of private memory
- 5 lectures

The 1st “Easy Piece” in OSTEP is virtual CPU+RAM
The Abstraction

A process has a set of addresses that map to bytes.

This set is called an address space.

How can we provide a private address space?

Extend LDE (limited direct execution).

Review: what stuff is in an address space?
int x;
int main(int argc, char *argv[]) {
    int y;
    int *z = malloc(sizeof(int));
}

x code
main data
y heap
z stack
int x;
int main(int argc, char *argv[]) {
    int y;
    int *z = malloc(sizeof(int));
}

x --> code
main --> data
y --> heap
z --> stack
```c
int x;
int main(int argc, char *argv[]) {
    int y;
    int *z = malloc(sizeof(int));
}
```

Match that Segment!

- x
- main
- y
- z

- code
- heap
- stack

in OSTEP
Program Code

Heap

(free)

Stack
where is code?  0x100f2ddd0  (4 GB)
where is data?  0x100f2e020  (4 GB)
where is heap?  0x7ff659403930 (131033 GB)
where is stack? 0x7fff5ecd2a1c (131069 GB)
demo1.c disassemble
```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int x;
    x = x + 3;
}
```

```assembly
_main:
  0000000000000000 pushq %rbp
  0000000000000001 movq %rsp, %rbp
  0000000000000004 movl $0x0, %eax
  0000000000000009 movl %edi, 0xfffffffc(%rbp)
  000000000000000c movq %rsi, 0xffffff0(%rbp)
  0000000000000010 movl 0xffffec(%rbp), %edi
  0000000000000013 addl $0x3, %edi
  0000000000000019 movl %edi, 0xfffffc(%rbp)
  000000000000001c popq %rbp
  000000000000001d ret
```

`otool -tv demo1.o`  
(or `objdump` on Linux)
Memory Accesses

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int x;
    x = x + 3;
}
```

```assembly
_start:
    pushq %rbp
    movq %rsp, %rbp
    movl $0x0, %eax
    movl %edi, 0xfffffc(%rbp)
    movq %rsi, 0xfffff0(%rbp)
    movl 0xffffec(%rbp), %edi
    addl $0x3, %edi
    movl %edi, 0xffffec(%rbp)
    popq %rbp
    ret
```

`otool -tv demo1.o`
(or `objdump` on Linux)
%rip = 0x10
%rbp = 0x200

0x10: movl 0x8(%rbp), %edi
0x13: addl $0x3, %edi
0x19: movl %edi, 0x8(%rbp)

Memory Accesses:
Memory Accesses

%rip = 0x10
%rbp = 0x200

0x10: movl 0x8(%rbp), %edi
0x13: addl $0x3, %edi
0x19: movl %edi, 0x8(%rbp)

Memory Accesses:
Fetch instruction at addr 0x10
%rip = 0x10
%rbp = 0x200

0x10: movl 0x8(%rbp), %edi
0x13: addl $0x3, %edi
0x19: movl %edi, 0x8(%rbp)

Memory Accesses:
Fetch instruction at addr 0x10
Exec, load from addr 0x208
Memory Accesses

%rip = 0x13
%rbp = 0x200

0x10: movl 0x8(%rbp), %edi
0x13: addl $0x3, %edi
0x19: movl %edi, 0x8(%rbp)

Memory Accesses:
Fetch instruction at addr 0x10
Exec, load from addr 0x208
%rip = 0x13
%rbp = 0x200

0x10: movl 0x8(%rbp), %edi
0x13: addl $0x3, %edi
0x19: movl %edi, 0x8(%rbp)

Memory Accesses:
Fetch instruction at addr 0x10
Exec, load from addr 0x208
Fetch instruction at addr 0x13
Memory Accesses

%rip = 0x13
%rbp = 0x200

0x10: movl 0x8(%rbp), %edi
0x13: addl $0x3, %edi
0x19: movl %edi, 0x8(%rbp)

Memory Accesses:
Fetch instruction at addr 0x10
Exec, 

Load from addr 0x208

Fetch instruction at addr 0x13
Exec, no load
Memory Accesses:

%rip = 0x19
%rbp = 0x200

0x10: movl 0x8(%rbp), %edi
0x13: addl $0x3, %edi
0x19: movl %edi, 0x8(%rbp)

Memory Accesses:
Fetch instruction at addr 0x10
Exec, load from addr 0x208
Fetch instruction at addr 0x13
Exec, no load
%rip = 0x19
%rbp = 0x200

0x10: movl 0x8(%rbp), %edi
0x13: addl $0x3, %edi
0x19: movl %edi, 0x8(%rbp)

---

Memory Accesses:
Fetch instruction at addr 0x10
Exec, load from addr 0x208

Fetch instruction at addr 0x13
Exec, no load

Fetch instruction at addr 0x19
%rip = 0x19
%rbp = 0x200

0x10: movl 0x8(%rbp), %edi
0x13: addl $0x3, %edi
0x19: movl %edi, 0x8(%rbp)

Memory Accesses:
Fetch instruction at addr 0x10
Exec, load from addr 0x208
Fetch instruction at addr 0x13
Exec, no load
Fetch instruction at addr 0x19
Exec, store to addr 0x208

%rip = 0x19
%rbp = 0x200
Problem: How to Run Multiple Processes?

Addresses are “hardcoded” into process binaries. How to avoid collisions?

Approaches (covered today):
  Time Sharing
  Static Relocation
  Base
  Base+Bounds
  Segmentation
Time Sharing

We give the illusion of many virtual CPUs by saving **CPU registers** to **memory** when a process isn’t running.

We give the illusion of many virtual memories by saving **memory** to **disk** when a process isn’t running.
Program Memory

create

- code
- data
- heap
- stack

Process 1
Program

- code
- data

Process 1

- code
- data
- heap
- stack

Memory
Program

memory

Process 1

create

Memory

code
data2
heap2
stack2
Process 2
Program

- code
- data

- code
- data2
- heap2
- stack2

Process 2

Memory

- code
- data
- heap
- stack

Process 1
Time Sharing

Problems?

What schedulers would time sharing work well with?

Alternative: space sharing
Problem: How to Run Multiple Processes?

Approaches (covered today):

- Time Sharing
- Static Relocation
- Base
- Base+Bounds
- Segmentation
Static Relocation

Idea: rewrite each program before loading it as a process

Each rewrite uses different addresses and pointers

Change jumps, loads, etc.
Can any addresses be unchanged?
Rewrite for Each New Process

0x10: movl 0x8(%rbp), %edi
0x13: addl $0x3, %edi
0x19: movl %edi, 0x8(%rbp)

0x1010: movl 0x8(%rbp), %edi
0x1013: addl $0x3, %edi
0x1019: movl %edi, 0x8(%rbp)

0x3010: movl 0x8(%rbp), %edi
0x3013: addl $0x3, %edi
0x3019: movl %edi, 0x8(%rbp)
Program Code
Heap
(free)

4 KB

8 KB

12 KB

16 KB

stack
(free)

process 1

process 2

0x1010: movl 0x8(%rbp), %edi
0x1013: addl $0x3, %edi
0x1019: movl %edi, 0x8(%rbp)

0x3010: movl 0x8(%rbp), %edi
0x3013: addl $0x3, %edi
0x3019: movl %edi, 0x8(%rbp)
why didn’t we have to rewrite the stack addr?

0x1010: movl 0x8(%rbp), %edi
0x1013: addl $0x3, %edi
0x1019: movl %edi, 0x8(%rbp)

0x3010: movl 0x8(%rbp), %edi
0x3013: addl $0x3, %edi
0x3019: movl %edi, 0x8(%rbp)
Problem: How to Run Multiple Processes?

Approaches (covered today):
- Time Sharing
- Static Relocation
- Base
- Base+Bounds
- Segmentation
Idea: translate virtual addresses to physical by adding a fixed offset each time.

Store offset in a base register.

Each process has a different value in the base register when running.

This is a “dynamic relocation” technique.
same code
P1 is running

base register
P2 is running

base register
P1: load 100, R1
<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: load 100, R1</td>
<td>load 1124, R1</td>
</tr>
</tbody>
</table>
Virtual
P1: load 100, R1
P2: load 100, R1

Physical
load 1124, R1
<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: load 100, R1</td>
<td>load 1124, R1</td>
</tr>
<tr>
<td>P2: load 100, R1</td>
<td>load 4196, R1</td>
</tr>
<tr>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>P1: load 100, R1</td>
<td>load 1124, R1</td>
</tr>
<tr>
<td>P2: load 100, R1</td>
<td>load 4196, R1</td>
</tr>
<tr>
<td>P2: load 1000, R1</td>
<td></td>
</tr>
<tr>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>P1: load 100, R1</td>
<td>load 1124, R1</td>
</tr>
<tr>
<td>P2: load 100, R1</td>
<td>load 4196, R1</td>
</tr>
<tr>
<td>P2: load 1000, R1</td>
<td>load 5196, R1</td>
</tr>
</tbody>
</table>
Virtual | Physical
--- | ---
P1: load 100, R1 | load 1124, R1
P2: load 100, R1 | load 4196, R1
P2: load 1000, R1 | load 5196, R1
P1: load 100, R1
<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: load 100, R1</td>
<td>load 1124, R1</td>
</tr>
<tr>
<td>P2: load 100, R1</td>
<td>load 4196, R1</td>
</tr>
<tr>
<td>P2: load 1000, R1</td>
<td>load 5196, R1</td>
</tr>
<tr>
<td>P1: load 100, R1</td>
<td>load 2024, R1</td>
</tr>
</tbody>
</table>
Who Controls the Base Register?

Who should **do translation** with base register?
(1) process, (2) OS, or (3) HW

Who should **modify** the base register?
(1) process, (2) OS, or (3) HW
Can P2 hurt P1?
Can P1 hurt P2?
Can P2 hurt P1?
Can P1 hurt P2?

Virtual
P1: load 100, R1
P2: load 100, R1
P2: load 1000, R1
P1: store 3072, R1

Physical
load 1124, R1
load 4196, R1
load 5196, R1
load 2024, R1
store 4096, R1
Problem: How to Run Multiple Processes?

Approaches (covered today):
  - Time Sharing
  - Static Relocation
  - Base
  - Base+Bounds
  - Segmentation
Idea: contain the address space with a bounds register marking the largest physical address

**Base register**: smallest physical addr

**Bounds register**: largest physical addr

What happens if you load/store after bounds?
P1 is running

- base register
- bounds register

P1 is running
P2 is running

- base register
- bounds register
Can P1 hurt P2?
Can P1 hurt P2?

Virtual

P1: load 100, R1
P2: load 100, R1
P2: load 1000, R1
P1: load 100, R1
P1: store 3072, R1

Physical

load 1124, R1
load 4196, R1
load 5196, R1
load 2024, R1
interrupt OS!
Can P1 hurt P2?

Virtual
P1: load 100, R1
P2: load 100, R1
P2: load 1000, R1
P1: load 100, R1
P1: store 3072, R1

Physical
load 1124, R1
load 4196, R1
load 5196, R1
load 2024, R1
interrupt OS!
Base+Bounds Pros/Cons

Pros?

Cons?
Base+Bounds Pros/Cons

Pros?
- fast + simple
- little bookkeeping overhead (2 registers / proc)

Cons?
- not flexible
- wastes memory for large address spaces
Base+Bounds Pros/Cons

Pros?
- fast + simple
- little bookkeeping overhead (2 registers / proc)

Cons?
- not flexible
- wastes memory for large address spaces
Problem: How to Run Multiple Processes?

Approaches (covered today):
  Time Sharing
  Static Relocation
  Base
  Base+Bounds
  Segmentation
Idea: generalize base+bounds

Each base+bound pair is a segment

Use different segments for heap and memory
- how does this help?
- requires more registers!

Resize segments as needed
- how does this help?
Multi-segment translation

One (broken) approach:
- have no gaps in virtual addresses
- map as many low addresses to the first segment as possible, then as many as possible to the second (on so on)
A tricky example

Virtual | Physical
---|---
P1: load 100, R1 | 

- P1: heap
- P1: stack
A tricky example

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: load 100, R1</td>
<td>load 1124, R1</td>
</tr>
</tbody>
</table>

**Diagram:**

- **P1: heap** at 1 KB
- **P1: stack** at 4 KB

**Dimensions:**

- 0 KB
- 1 KB
- 2 KB
- 3 KB
- 4 KB
- 5 KB
- 6 KB
A tricky example

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: load 100, R1</td>
<td>load 1124, R1</td>
</tr>
<tr>
<td>P1: load 1024, R1</td>
<td></td>
</tr>
</tbody>
</table>

Diagram showing:
- 0 KB
- 1 KB: P1: heap
- 2 KB: P1: stack
- 3 KB
- 4 KB
- 5 KB
- 6 KB
A tricky example

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: load 100, R1</td>
<td>load 1124, R1</td>
</tr>
<tr>
<td>P1: load 1024, R1</td>
<td>load 4096, R1</td>
</tr>
</tbody>
</table>

- **P1: heap**
- **P1: stack**
A tricky example

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: load 100, R1</td>
<td>load 1124, R1</td>
</tr>
<tr>
<td>P1: load 1024, R1</td>
<td>load 4096, R1</td>
</tr>
</tbody>
</table>

P1: heap

P1: stack
grow heap
A tricky example

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: load 100, R1</td>
<td>load 1124, R1</td>
</tr>
<tr>
<td>P1: load 1024, R1</td>
<td>load 4096, R1</td>
</tr>
<tr>
<td>P1: load 1024, R1</td>
<td></td>
</tr>
</tbody>
</table>

- **P1: heap**
- **P1: stack**
A tricky example

### Virtual
- P1: load 100, R1
- P1: load 1024, R1
- P1: load 1024, R1

### Physical
- load 1124, R1
- load 4096, R1
- load 2048, R1

#### Diagram:
- 0 KB
- 1 KB
- 2 KB: P1: heap
- 3 KB
- 4 KB: P1: stack
- 5 KB
- 6 KB
Multi-segment translation

One (correct) approach:
- break virtual addresses into two parts
- one part indicates segment
- one part indicates offset within segment
Virtual Address

For example, say addresses are 14 bits. Use 2 bits for segment, 12 bits for offset.

An address might look like 201E.
Virtual Address

For example, say addresses are 14 bits. Use 2 bits for segment, 12 bits for offset.

An address might look like 2 01E

segment 2    offset 30
Virtual Address

For example, say addresses are 14 bits. Use 2 bits for segment, 12 bits for offset.

An address might look like 2 01E

Choose some segment numbering, such as:
0: code+data
1: heap
2: stack
What is the segment/offset?

Segment numbers:
  0: code+data
  1: heap
  2: stack

10 0000 0001 0001  (binary)
110A       (hex)
4096       (decimal)
heap (seg1)

load 0x2010, R1

stack (seg2)
heap (seg1)

stack (seg2)

Virtual
load 0x2010, R1

Physical
4KB + 16
Virtual
load 0x2010, R1
load 0x1010, R1

Physical
4KB + 16
heap (seg1)

stack (seg2)

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x2010, R1</td>
<td>4KB + 16</td>
</tr>
<tr>
<td>load 0x1010, R1</td>
<td>1KB + 16</td>
</tr>
</tbody>
</table>
Heap (seg1)

Stack (seg2)

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x2010, R1</td>
<td>4KB + 16</td>
</tr>
<tr>
<td>load 0x1010, R1</td>
<td>1KB + 16</td>
</tr>
<tr>
<td>load 0x1100, R1</td>
<td></td>
</tr>
</tbody>
</table>
Virtual | Physical
---|---
load 0x2010, R1 | 4KB + 16
load 0x1010, R1 | 1KB + 16
load 0x1100, R1 | 1KB + 256
### Heap (seg1)
- 1 KB
- 2 KB
- 3 KB
- 4 KB

### Stack (seg2)
- 5 KB
- 6 KB

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x2010, R1</td>
<td>4KB + 16</td>
</tr>
<tr>
<td>load 0x1010, R1</td>
<td>1KB + 16</td>
</tr>
<tr>
<td>load 0x1100, R1</td>
<td>1KB + 256</td>
</tr>
<tr>
<td>load 0x1400, R1</td>
<td></td>
</tr>
</tbody>
</table>
heap (seg1)

1 KB

2 KB

3 KB

4 KB

5 KB

6 KB

Virtual
load 0x2010, R1
load 0x1010, R1
load 0x1100, R1
load 0x1400, R1

Physical
4KB + 16
1KB + 16
1KB + 256
interrupt OS!
Stack Growth Problem

Example…
<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x2000, R1</td>
<td></td>
</tr>
</tbody>
</table>

- **Heap (seg1)**: 1 KB to 2 KB
- **Stack (seg2)**: 4 KB to 5 KB
heap (seg1)

stack (seg2)

Virtual
load 0x2000, R1

Physical
4KB
Virtual | Physical
---|---
load 0x2000, R1 | 4KB

heap (seg1)

stack (seg2)

grow stack
<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x2000, R1</td>
<td>4KB</td>
</tr>
<tr>
<td>load 0x2000, R1</td>
<td>4KB</td>
</tr>
</tbody>
</table>

- **heap (seg1)**: 1 KB
- **stack (seg2)**: 5 KB

The diagram shows a memory map with the heap and stack segments. The virtual memory is mapped to physical memory, and the load values are aligned accordingly.
<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x2000, R1</td>
<td>4KB</td>
</tr>
<tr>
<td>load 0x2000, R1</td>
<td>3KB</td>
</tr>
</tbody>
</table>

- Heap (seg1) at 3 KB
- Stack (seg2) at 4 KB
Stack Growth Problem

Example...

Problem: \texttt{phys = virt\_offset + base\_reg}
phys is anchored to \texttt{base\_reg}, which moves
Example...

Problem: \( \text{phys} = \text{virt\_offset} + \text{base\_reg} \)
phys is anchored to \text{base\_reg}, which moves

Solution: anchor heap segment to \text{bounds\_reg}:
\( \text{phys} = \text{bounds\_reg} - (\text{max\_offset} - \text{virt\_offset}) \)
Stack Growth Problem

Example...

Problem: \texttt{phys = virt\_offset + base\_reg}
phys is anchored to \texttt{base\_reg}, which moves

Solution: anchor heap segment to \texttt{bounds\_reg}:
\texttt{phys = bounds\_reg - (max\_offset - virt\_offset)}

Example (with \texttt{max\_offset = FFF})...
stack’s max_offset = FFF

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x2FFE, R1</td>
<td></td>
</tr>
</tbody>
</table>
stack’s max_offset = FFF

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x2FFE, R1</td>
<td>5KB - 1</td>
</tr>
</tbody>
</table>

- heap (seg1)
- stack (seg2)
stack’s max_offset = FFF

Virtual
load 0x2FFE, R1
load 0x2BFF, R1

Physical
5KB - 1
heap (seg1)

stack (seg2)

stack’s max_offset = FFF

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x2FFFE, R1</td>
<td>5KB - 1</td>
</tr>
<tr>
<td>load 0x2BFF, R1</td>
<td>4KB</td>
</tr>
</tbody>
</table>
stack’s max_offset = FFF

Virtual
load 0x2FFE, R1
load 0x2BFF, R1

Physical
5KB - 1
4KB

grow stack

heap (seg1)

stack (seg2)
stack’s max_offset = FFF

heap (seg1)

stack (seg2)

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x2FFFE, R1</td>
<td>5KB - 1</td>
</tr>
<tr>
<td>load 0x2BFF, R1</td>
<td>4KB</td>
</tr>
<tr>
<td>load 0x2BFF, R1</td>
<td></td>
</tr>
</tbody>
</table>
stack’s max_offset = FFF

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x2FFE, R1</td>
<td>5KB - 1</td>
</tr>
<tr>
<td>load 0x2BFF, R1</td>
<td>4KB</td>
</tr>
<tr>
<td>load 0x2BFF, R1</td>
<td>4KB</td>
</tr>
</tbody>
</table>
Translation Summary

Heap: \texttt{phys} = \texttt{base\_reg} + \texttt{virt\_offset}

Stack: \texttt{phys} = \texttt{bounds\_reg} - (\texttt{max\_offset} - \texttt{virt\_offset})

Anchors:
- for heap, anchor \texttt{smallest address to base register}
- for stack, anchor \texttt{biggest address to bounds register}
Code Sharing

Idea: make base/bounds for the code of several processes point to the same physical mem

Careful: need extra protection!
Segmentation Pros/Cons

Pros?
- supports sparse address space
- code sharing
- fine grained protection

Cons?
- external fragmentation
Conclusion

HW+OS work together to trick processes, giving the illusion of private memory

Adding CPU registers for base+bounds extends LDE, so translation is fast (does not always need OS)

Next time: solve fragmentation with paging