VIRTUALIZATION: CPU TO MEMORY

Shivaram Venkataraman
CS 537, Spring 2020
- Project 1a: DONE!?
- How to use slip days? (Piazza)

- Project 1b is out, due Feb 5th (Next Wednesday) at 10 pm!
- Discussion section → 5:30pm Thursday
  - xv6 code walk through
  - How to use gdb
AGENDA / LEARNING OUTCOMES

CPU virtualization
- Recap of scheduling policies
- Work through problems

Memory virtualization
- What is the need for memory virtualization?
- How to virtualize memory?
RECAP: CPU VIRTUALIZATION
RECAP: SCHEDULING MECHANISM

Process: Abstraction to virtualize CPU
Use time-sharing in OS to switch between processes

Limited Direct Execution
Use system calls to run access devices etc. from user mode
Context-switch using interrupts for multi-tasking
Turnaround time = completion_time - arrival_time

FIFO: First come, first served
SJF: Shortest job first
SCTF: Shortest completion time first
Response time = \textit{first\_run\_time} - \textit{arrival\_time}

RR: Round robin with time slice
Minimizes response time but could increase turnaround?
MULTI-LEVEL FEEDBACK QUEUE
MLFQ EXAMPLE

Rules for MLFQ

Rule 1: If priority(A) > Priority(B), A runs

Rule 2: If priority(A) == Priority(B), A & B run in RR

Rule 3: Processes start at top priority

Rule 4: If job uses whole slice, demote process. If not stay at level
ONE LONG JOB

10 ms  Q2

10 ms  Q1

Q0

0  50  100  150  200
At $t=90\text{ms}$

$Q_2 \rightarrow B$

$Q_1$

$Q_0 \rightarrow A$

At $Q_0$

Queue $TS = \infty$

RR $TS = 1: \infty$

At $t=100$

$Q_2 \rightarrow$

$Q_1 \rightarrow B$

$Q_0 \rightarrow A$

Queue $TS$: denote

RR $TS$: within queue

$Q_{TS}: RR_{TS}$
At \( t = 10 \text{ ms} \)

A has used

10 ms

< \( Q_{TS} \)

Keep A at Q1
What is the problem with this schedule?

MLFQ PROBLEMS?

A suffers from starvation.

A, B2, ..., B100
AVOIDING STARVATION

Rule 5: After some time period $S$, move all the jobs in the system to the topmost queue.

$S: 50 \text{ ms}$
GAMING THE SCHEDULER?

Job could trick scheduler by doing I/O just before time-slice end

$p_a$ finishes I/O
rejoins at $Q_2$
GAMING THE SCHEDULER?

Job could trick scheduler by doing I/O just before time-slice end

Rule 4*: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced
CPU SUMMARY

Mechanism

   Process abstraction
   System call for protection
   Context switch to time-share

Policy

   Metrics: turnaround time, response time
   Balance using MLFQ
VIRTUALIZING MEMORY
BACK IN THE DAY...

Uniprogramming: One process runs at a time

One program that exists in the system
MULTI PROGRAMMING GOALS

- **Transparency**: Process is unaware of sharing
- **Protection**: Cannot corrupt OS or other process memory
- **Efficiency**: Do not waste memory or slow down processes
- **Sharing**: Enable sharing between cooperating processes
ABSTRACTION: ADDRESS SPACE

- Stack: contains local variables, arguments to routines, return values, etc.
- Heap: contains malloc'd data, dynamic data structures (it grows downward)
- Program Code: where instructions live

In the process view, the stack grows upward and the heap grows downward. The physical memory includes space for the operating system, processes, and free memory.
WHAT IS IN ADDRESS SPACE?

<table>
<thead>
<tr>
<th>0KB</th>
<th>1KB</th>
<th>2KB</th>
<th>15KB</th>
<th>16KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Code</td>
<td>Heap</td>
<td>Stack</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Program Code**: Where instructions live
- **Heap**: Contains malloc'd data, dynamic data structures
  - (it grows downward)
- **Stack**: Contains local variables, arguments to routines, return values, etc.
  - (it grows upward)

**Static**: Code and some global variables

**Dynamic**: Stack and Heap

- `mov $3 %eax`
- `call add`
- `malloc()`, new
- `local variables`, `return`
STACK ORGANIZATION

alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);

Pointer between allocated and free space
Allocate: Increment pointer
Free: Decrement pointer

→ No fragmentation!

LIFO behavior

15KB
Stack
16KB

Heap

Program Code: where instructions live
Contains malloc'd data
dynamic data structures (it grows downward)

Stack segment:
Contains local variables
Arguments to routines, return values, etc.

No empty regions which are used
WHAT GOES ON STACK?

main () {
    int A = 0;
    foo(A);
    printf("A: %d\n", A);
}

void foo (int Z) {
    int A = 2;
    Z = 5;
    printf("A: %d Z: %d\n", A, Z);
}

Calling convention
x86

local variable
arguments

return value
also goes on stack
HEAP ORGANIZATION

Allocate from any random location: `malloc()`, `new()` etc.

- Heap memory consists of allocated and free areas (holes)
- Order of allocation and free is unpredictable

<table>
<thead>
<tr>
<th></th>
<th>12 bytes</th>
<th>16 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Free</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Alloc</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Free</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`c = malloc(12)`
`a = malloc(16)`
`b = malloc(24)`
`free(a)`
`free(c)`

Fragmentation
#include <stdio.h>
#include <stdlib.h>

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

%rbp is the base pointer:
points to base of current stack frame
MEMORY ACCESS

Initial %rip = 0x10
%rbp = 0x200

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

%rbp is the base pointer:
points to base of current stack frame

%rip is instruction pointer (or program counter)

Fetch Inst at addr 0x10
Exec: fetch data at 0x208
Fetch Inst at 0x13
Exec:
Fetch inst at 0x19
Exec: Store data at 0x208
MEMORY ACCESS

Initial %rip = 0x10
%rbp = 0x200

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

%rbp is the base pointer:
points to base of current stack frame

%rip is instruction pointer (or program counter)

Fetch instruction at addr 0x10
Exec:
  load from addr 0x208

Fetch instruction at addr 0x13
Exec:
  no memory access

Fetch instruction at addr 0x19
Exec:
  store to addr 0x208
```c
int x;
int main(int argc, char *argv[]) {
    int y;
    int* z = malloc(sizeof(int));
}
```

Possible locations:
- static data/code, stack, heap

<table>
<thead>
<tr>
<th>Address</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>static data/code</td>
</tr>
<tr>
<td>main</td>
<td>code</td>
</tr>
<tr>
<td>y</td>
<td>stack</td>
</tr>
<tr>
<td>z</td>
<td>stack</td>
</tr>
<tr>
<td>*z</td>
<td>heap</td>
</tr>
</tbody>
</table>
HOW TO VIRTUALIZE MEMORY

Problem: How to run multiple processes simultaneously? Addresses are “hardcoded” into process binaries How to avoid collisions?

Possible Solutions for Mechanisms (covered today):

1. Time Sharing
2. Static Relocation
3. Base
4. Base+Bounds
TIME SHARE MEMORY: EXAMPLE
PROBLEMS WITH TIME SHARING?

Ridiculously poor performance

Better Alternative: space sharing!
   At same time, space of memory is divided across processes
   Remainder of solutions all use space sharing
2) STATIC RELOCATION

Idea: OS rewrites each program before loading it as a process in memory.
Each rewrite for different process uses different addresses and pointers.
Change jumps, loads of static data.

```assembly
0x10: movl 0x8(%rbp), %edi
0x13: addl $0x3, %edi
0x19: movl %edi, 0x8(%rbp)
0x3010: movl 0x8(%rbp), %edi
0x3013: addl $0x3, %edi
0x3019: movl %edi, 0x8(%rbp)
```
STATIC: LAYOUT IN MEMORY

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 KB</td>
<td>0x1010: movl 0x8(%rbp), %edi</td>
</tr>
<tr>
<td></td>
<td>0x1013: addl $0x3, %edi</td>
</tr>
<tr>
<td></td>
<td>0x1019: movl %edi, 0x8(%rbp)</td>
</tr>
<tr>
<td>8 KB</td>
<td>0x3010: movl 0x8(%rbp), %edi</td>
</tr>
<tr>
<td></td>
<td>0x3013: addl $0x3, %edi</td>
</tr>
<tr>
<td></td>
<td>0x3019: movl %edi, 0x8(%rbp)</td>
</tr>
<tr>
<td>12 KB</td>
<td>Program Code</td>
</tr>
<tr>
<td></td>
<td>Heap</td>
</tr>
<tr>
<td></td>
<td>Stack</td>
</tr>
<tr>
<td>16 KB</td>
<td>(free)</td>
</tr>
</tbody>
</table>

Program Code
Heap
(free)
Stack
(free)
STATIC RELOCATION: DISADVANTAGES

No protection
   - Process can destroy OS or other processes
   - No privacy

Cannot move address space after it has been placed
   - May not be able to allocate new process
3) DYNAMIC RELOCATION

Goal: Protect processes from one another
Requires hardware support
  – Memory Management Unit (MMU)
MMU dynamically changes process address at every memory reference
  – Process generates logical or virtual addresses (in their address space)
  – Memory hardware uses physical or real addresses
HARDWARE SUPPORT FOR DYNAMIC RELOCATION

Privileged (protected, kernel) mode: OS runs
  – When enter OS (trap, system calls, interrupts, exceptions)
  – Allows certain instructions to be executed
    (Can manipulate contents of MMU)
  – Allows OS to access all of physical memory

User mode: User processes run
  – Perform translation of logical address to physical address
Translation on every memory access of user process
MMU adds base register to logical address to form physical address
DYNAMIC RELOCATION WITH BASE REGISTER

Translate virtual addresses to physical by adding a fixed offset each time.
  Store offset in base register

Each process has different value in base register
  Dynamic relocation by changing value of base register!
VISUAL EXAMPLE OF DYNAMIC RELOCATION: BASE REGISTER
Can P2 hurt P1?
Can P1 hurt P2?

How well does dynamic relocation do with base register for protection?
How well does dynamic relocation do with base register for protection?
4) DYNAMIC WITH BASE+BOUNDS

Idea: limit the address space with a bounds register

Base register: smallest physical addr (or starting location)
Bounds register: size of this process’s virtual address space
   – Sometimes defined as largest physical address (base + size)

OS kills process if process loads/stores beyond bounds
Implementation of BASE+BOUNDS

Translation on every memory access of user process

- MMU compares logical address to bounds register
  if logical address is greater, then generate error
- MMU adds base register to logical address to form physical address

![Diagram](image-url)

**registers**

- base: 32 bits
- bounds: 32 bits
- mode: 1 bit

**Flowchart**

- Logical address
- Mode = user?
  - Yes: < bounds?
    - Yes: base + base
    - No: error
  - No: error
base register

bounds register
Can P1 hurt P2?
MANAGING PROCESSES WITH BASE AND BOUNDS

Context-switch: Add base and bounds registers to PCB

Steps
- Change to privileged mode
- Save base and bounds registers of old process
- Load base and bounds registers of new process
- Change to user mode and jump to new process

Protection requirement
- User process cannot change base and bounds registers
- User process cannot change to privileged mode
Base and Bounds Advantages

Provides protection (both read and write) across address spaces
Supports dynamic relocation
  Can place process at different locations initially and also move address spaces

Simple, inexpensive implementation: Few registers, little logic in MMU
Fast: Add and compare in parallel
Disadvantages

– Each process must be allocated contiguously in physical memory
  Must allocate memory that may not be used by process

– No partial sharing: Cannot share parts of address space
NEXT STEPS

Project 1b: Out now, due Feb 5\textsuperscript{th}

Thursday discussion
  xv6 introduction, walk through
  Project 1b tips

Next week: Virtual memory segmentation, paging and more!