VIRTUALIZATION: CPU TO MEMORY

Shivaram Venkataraman
CS 537, Spring 2023
- Project 1: DONE!?
- How to use slip days? (Piazza)
- Project 2 is out, due next Wednesday

OH from 3pm to 4pm CS 7367
AGENDA / LEARNING OUTCOMES

CPU virtualization
   Recap of scheduling policies (Tue)
   Lottery Scheduling, Multi-CPU

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

A: 50
B: 5
C: 10
Response time = $\text{first\_run\_time} - \text{arrival\_time}$

Pre-emptive scheduling

RR: Round robin with time slice $= 1$

Minimizes response time but could increase turnaround?

A: So, B.5, 2: 10

At any point OS can stop running & replace it.

$A: 50, B: 5, C: 10$
RECAP: MULTI-LEVEL FEEDBACK QUEUE

What if we don’t know how long a job will run?

Support two job types with distinct goals
- “interactive” programs care about response time
- “batch” programs care about turnaround time

Approach:
Multiple levels of round-robin
Each level has higher priority than lower level
Can preempt them
RECAP: MULTI-LEVEL FEEDBACK QUEUE

8-level MLFQ

Round robin scheduling in Q8 → A, B, C

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

Q8 → A
[High Priority]

Q7 → A

Q6

Q5

Q4 → C
[t=50ms]

Q3

Q2

Q1 → D
[Low Priority]

[t=10ms]
INTERACTIVE PROCESS JOINS

Q2

Q1

Q0
Avoid Starvation

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

Priority Boost!

Job A is starved.
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
### Quiz 5

**URL:** [https://tinyurl.com/cs537-sp23-quiz5](https://tinyurl.com/cs537-sp23-quiz5)

#### Question 1

<table>
<thead>
<tr>
<th>Jobs</th>
<th>Runtime</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job A</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>Job B</td>
<td>10</td>
<td>50</td>
</tr>
</tbody>
</table>

#### Question 2

<table>
<thead>
<tr>
<th>Jobs</th>
<th>Runtime</th>
<th>Arrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job A</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>Job B</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>Job C</td>
<td>20</td>
<td>70</td>
</tr>
</tbody>
</table>

#### Diagram

- **High Priority**
  - Job A
  - Job B

- **Low Priority**
  - Q0
  - Q1
  - Q2
  - Q3

- Boost
  - A
  - B
  - C
FAIRNESS IN SCHEDULING

Metrics so far: turn around time, response time.
New metric: Fairness!

3 users; each get 1/3rd of CPU no matter how long they run for

Is MLFQ fair?

Jobs which run at Q0 get more cycles.
LOTTERY SCHEDULING ~ mid-90s

Approach:
1. Give processes lottery tickets
2. Whoever wins the lottery runs
3. Higher priority => more tickets

<table>
<thead>
<tr>
<th>JOB</th>
<th>Tickets</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
</tr>
<tr>
<td>C</td>
<td>30</td>
</tr>
</tbody>
</table>

Total 60

RR

Lottery C ~ 7/15

random

0-10 A

10-30 B

30-60 C
IMPLEMENTING LOTTERY SCHEDULING

Scheduler:

```c
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t *current = head;
while (current) {
    counter += current->tickets;
    if (counter > winner) break;
    current = current->next;
}
// current gets to run
```

Who runs if `winner` is:

<table>
<thead>
<tr>
<th>Winner</th>
<th>Job</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>C</td>
</tr>
<tr>
<td>350</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>E</td>
</tr>
</tbody>
</table>

A 0 - 1
B 1 - 2
C 2 - 102

5 jobs linked list
Mechanism
  - Process abstraction
  - System call for protection
  - Context switch to time-share

Policy
  - Metrics: turnaround time, response time
  - Balance using MLFQ
  - Fairness with Lottery Scheduling
VIRTUALIZING MEMORY
BACK IN THE DAY...

Uniprogramming: One process runs at a time

we want to run more than one process
MULTIPROGRAMMING 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
Virtual address = Process sees

Program Code (free)

Heap (free)

Stack (free)

Process Every

has memory regions

instructions used

static data

dynamic memory allocations

local variables arguments

Operating System (code, data, etc.)

Process C (code, data, etc.)

(Process B (code, data, etc.)

(Process A (code, data, etc.)

(free)

(free)

(free)
WHAT IS IN ADDRESS SPACE?

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

**Static**: Code and some global variables

**Dynamic**: Stack and Heap
Unix-like OS use **fork()**

Fork() - Clones the calling process to create a child process
Make copy of code, data, stack etc.
Add new process to ready list

Exec(char *file): Replace current data and code with file

Advantages: Flexible, clean, simple
Disadvantages: Wasteful to perform copy and overwrite of memory
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!

- Stack
- Heap

the code segment:
where instructions live

the heap segment:
contains malloc'd data
dynamic data structures
(it grows downward)

the stack segment:
contains local variables
arguments to routines,
return values, etc.

- Stack
- Last In first Out
main () {
    int A = 0;
    foo(10);
    printf("A: %d\n", A);
}

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

Allocate 24 bytes
→ No contiguous region
→ Fragmentation

memory allocators
which handle allocation within range

B = malloc(16 bytes)
MEMORY ACCESS

#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)
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
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></td>
</tr>
<tr>
<td>main</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
</tr>
<tr>
<td>*z</td>
<td></td>
</tr>
</tbody>
</table>
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

process 1

4 KB
- (free)
- Program Code
- Heap
- (free)

8 KB
- stack
- (free)

12 KB
- Program Code
- Heap
- (free)

16 KB
- stack
- (free)

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)
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
**IMPLEMENTATION OF DYNAMIC RELOCATION: BASE REG**

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

Virtual

P1: load 100, R1
P2: load 100, R1
P2: load 1000, R1
P1: load 100, R1
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
0 KB
1 KB P1
2 KB
3 KB
4 KB P2
5 KB
6 KB

base register
bounds register
Can P1 hurt P2?

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

Physical
load 1124, R1
load 4196, R1
load 5196, R1
load 2024, R1
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

Advantages
  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
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 2: Out now, due Feb 5th

Next week: Virtual memory segmentation, paging and more!