Virtualizing Memory

CS 537: Introduction to Operating Systems

Louis Oliphant

University of Wisconsin - Madison

Fall 2023

Administrivia

- Project 1 Due Sep 19th, 11:59pm (TONIGHT!)
- Project 2 Released Today, Due Sep 26th @ 11:59pm

Agenda

- Goals of Memory Virtualization
- Understand Address Space
- Memory API (`malloc()` and `free()`)
- Address Translation (Base & Bounds)

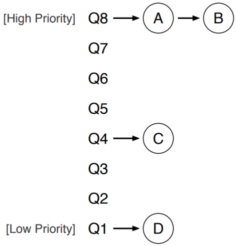
- CPU Virtualization (2 lectures: mechanism + policy)
- Memory Virtualization (6 lectures)

Review: CPU Scheduling

- Design scheduling policy:
  - Understanding Workload (**interactive** vs. **batch** programs)
  - Using metrics to optimize type of performance (**turnaround time**, **response time**)
  - Incorporating non-preemptive or **preemptive** concepts
- Scheduler Types & Issues:
  - FIFO/FCFS, SJF, STCF, RR
  - **MLFQ**
    - Using past behavior to predict future behavior
    - Handling mix of IO vs CPU bound jobs
    - Handling tricky processes
    - Tuning length of **time slice**, number of queues, **boosting** length
- Other Goals/Metrics (**fairness**) and Policies (**Lottery**)

MLFQ Review

Quiz 3 MLFQ: https://tinyurl.com/cs537-fa23-q3



RULES:

Rule 1: If Priority(A) > Priority(B) then A runs

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

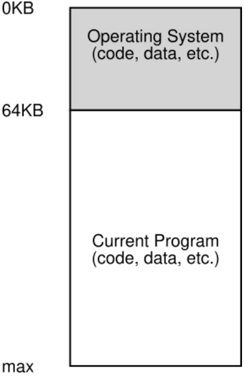
Rule 3: Processes start at top priority

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

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

Memory Early Days

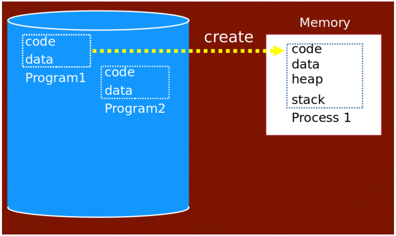
Uniprogramming: One process runs at a time

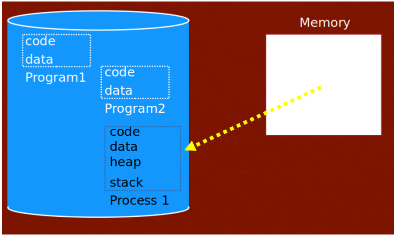


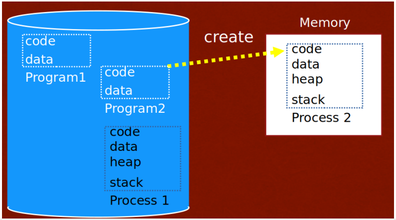
Multiprogramming Goals

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

Alternative 1: Time Sharing

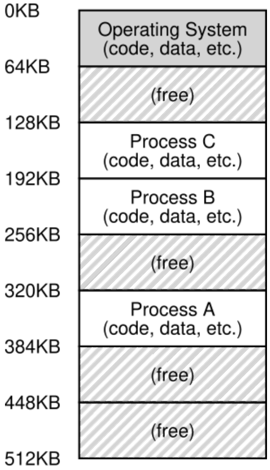






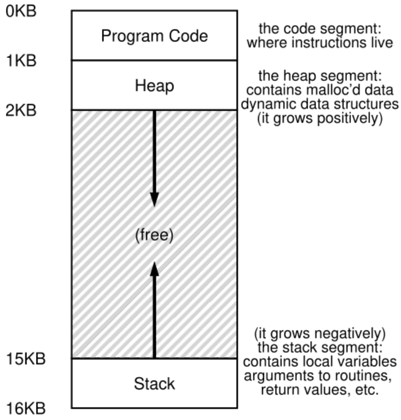
- Storing process and loading another is extremely slow!
- Better Alternative: Space Sharing!

Alternative 2: Space Sharing



Protection becomes extremely important, don't want a process to be able to read or write some other process's memory.

Abstraction: Address Space



View of memory from program's perspective.

- Heap can become fragmented
- Stack does not

Demo

`vm-intro/va.c`

What Variables Go Where (Stack, Heap, Code/Static)?

```
int J;

int* foo(int Y, int *Z) {
        int *A = malloc(sizeof(int));
        *A = 2;
        Y = 3;
        *Z = 4;
        return A;
}
void main() {
        J = 10;
        int A = 0;
        int *B;
        B = malloc(sizeof(int));
        *B = 5;
        int *C = foo(A,B);
        printf("A:%d, B:%d,",A,*B);
        printf("C:%d, J:%d\n",*C,J);
        free(B);
        free(C);
}
```

Memory Access

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

int main() {
  int x;
  x = x + 3;
}
```

```
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

Memory Access (cont.)

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 (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**

Space Sharing Attempt 1 (Static Relocation)

Idea: OS rewrites each program as it is loaded and placed in memory

Change jumps, loads of static data, etc.

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

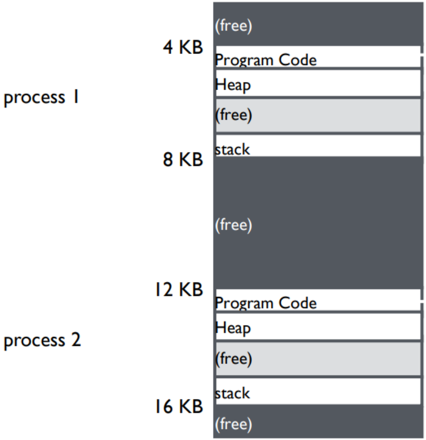
rewrite

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

rewrite

```
0x3010:movl  0x8(%rbp), %edi
0x3013:addl  $0x3, %edi
0x3019:movl  %edi, 0x8(%rbp)
```

Static Relocation Memory Layout




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

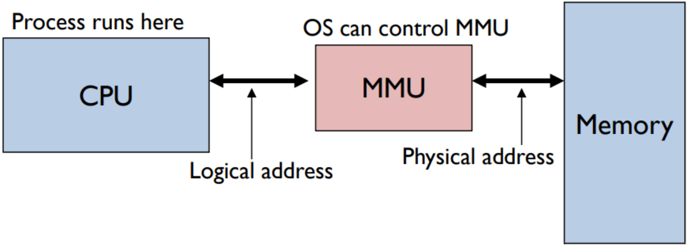
```
0x3010:movl  0x8(%rbp), %edi
0x3013:addl  $0x3, %edi
0x3019:movl  %edi, 0x8(%rbp)
```

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

Space Sharing Attempt 2 (Dynamic Relocation)

- 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



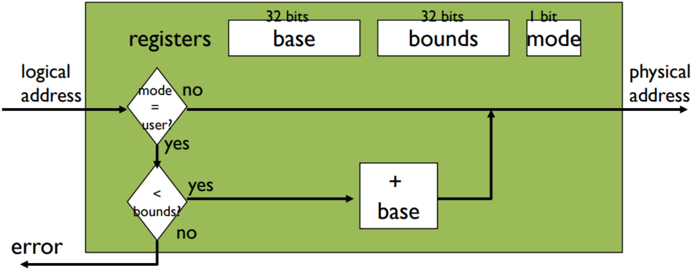
Dynamic Relocation Hardware Support

- Kernel Mode: OS runs
  - Allows instructions for manipulating MMU
  - OS access to all of physical memory
- User mode: process runs
  - Perform translation of logical address to physical address

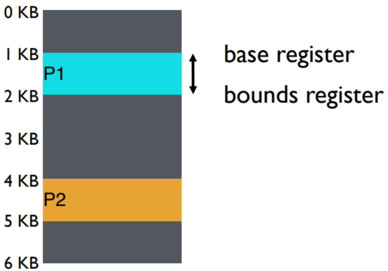
Dynamic Relocation with **Base+Bounds**

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



Base+Bounds Example



Every process has its own set of base and bounds register values

OS sets registers when loading process

Process can be moved, just need to update its base register

Process is restricted to its address space

Hardware Requirements

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

Figure 15.3: **Dynamic Relocation: Hardware Requirements**

OS Requirements

OS Requirements	Notes
Memory management	*Need to allocate memory for new processes;* *Reclaim memory from terminated processes;* *Generally manage memory via* ***free list***
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise;* *likely action is to terminate offending process*

Limited Direct Execution (Dynamic Relocation) @ Boot

OS @ boot (kernel mode)	**Hardware**	**(No Program Yet)**
initialize trap table		
	remember addresses of... system call handler timer handler illegal mem-access handler illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table **initialize free list**		

Figure 15.5: **Limited Direct Execution (Dynamic Relocation) @ Boot**

Good Running Process

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
To start process A:		
allocate entry in process table		
alloc memory for process		
set base/bound registers		
return-from-trap (into A)		
	restore registers of A	
	move to **user mode**	
	jump to A's (initial) PC	
		Process A runs
		Fetch instruction
	translate virtual address	
	perform fetch	
		Execute instruction
	if explicit load/store: ensure address is legal	
	translate virtual address	
	perform load/store	
		(A runs...)

Context Switch

(A runs...)

Timer interrupt
move to **kernel mode**
jump to handler

Handle timer
decide: stop A, run B
call `switch()` routine
save regs(A)
to proc-struct(A)
(including base/bounds)
restore regs(B)
from proc-struct(B)
(including base/bounds)
return-from-trap (into B)

restore registers of B
move to **user mode**
jump to B's PC

Process B runs

Bad Process

Process B runs
Execute bad load

Load is out-of-bounds;
move to **kernel mode**
jump to trap handler

Handle the trap
decide to kill process B
deallocate B's memory
free B's entry
in process table

Figure 15.6: **Limited Direct Execution (Dynamic Relocation) @ Runtime**

Advantages and Disadvantages

Advantages

- Provides protection across address spaces
- Supports Dynamic relocation – Can place process at different locations initially and move address spaces later
- 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
- Must allocate memory that may not be used by process
- No partial sharing: Cannot share parts of address space