CS 423
Operating System Design: Virtualization: CPU to Memory
02/05

Ram Alagappan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

Logistics

MP0: Due on 02/12

This lecture:

Finish CPU virtualization

- Recap on MLFQ
- Proportional share and fair scheduling

Start memory virtualization - memory layout and translation

AGENDA / LEARNING OUTCOMES

CPU virtualization

- Recap scheduling policies
- Fair and proportional share scheduling

Memory virtualization

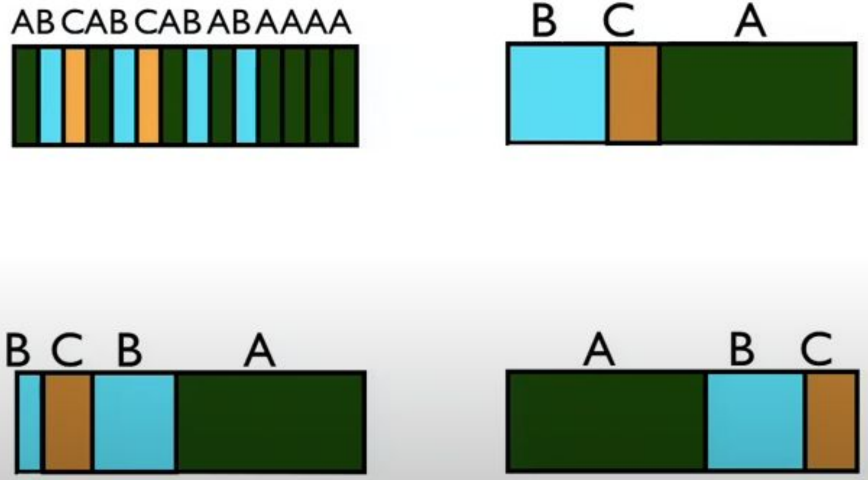
- What is the need for memory virtualization?
- How to virtualize memory?

RECAP: Scheduling Policies

Workload

JOB	arrival	run
A	0	40
B	0	20
C	5	10

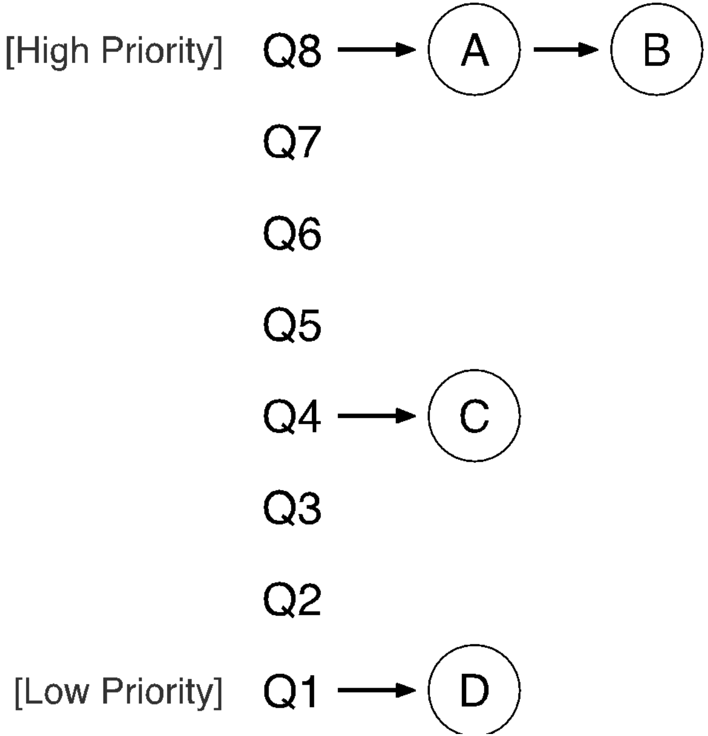
Timelines



Schedulers:

FIFO
SJF
STCF
RR

RECAP: MLFQ



Rules for MLFQ

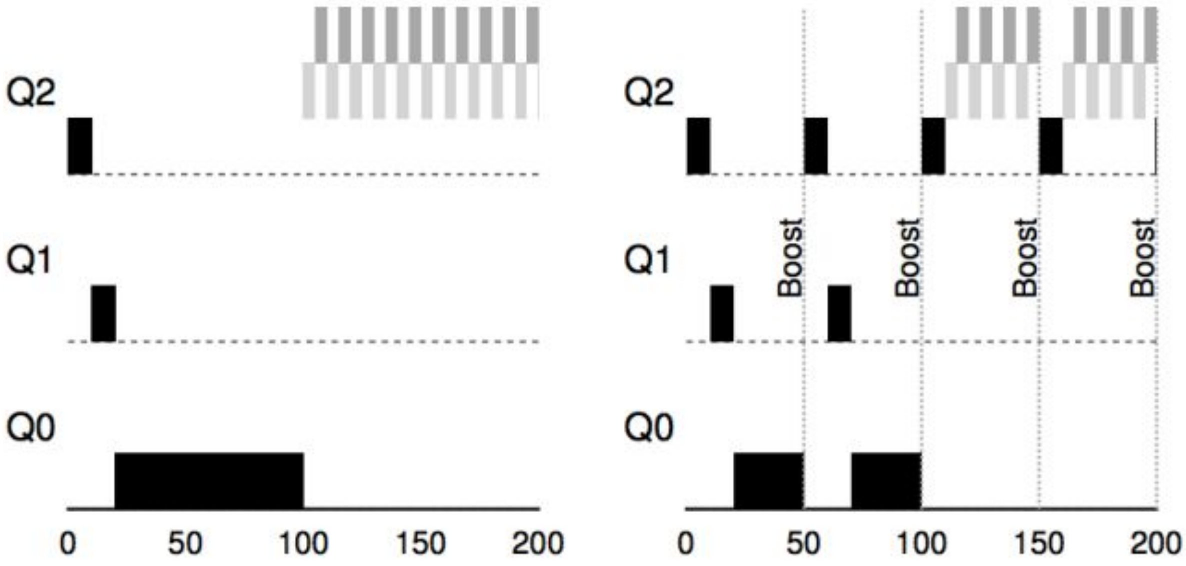
Rule 1: If priority(A) > Priority(B)
A runs

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

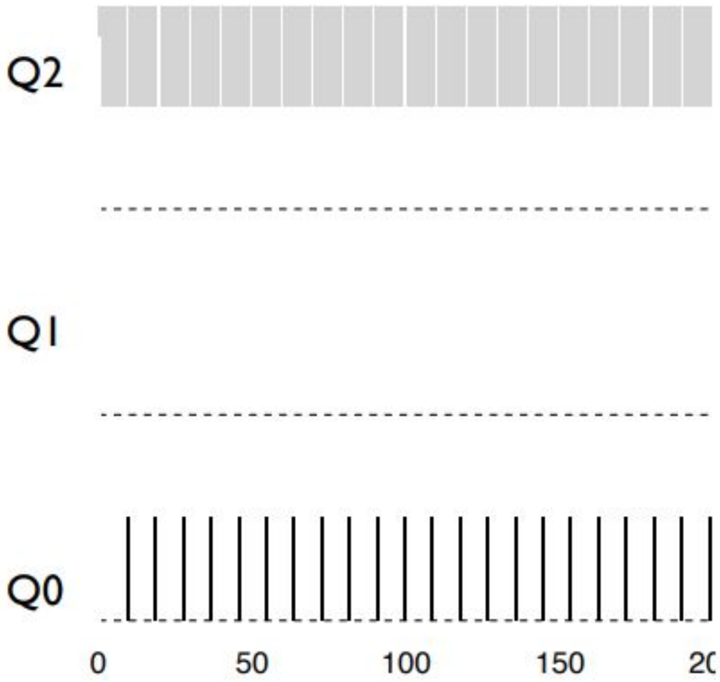
RECAP: MLFQ - Solving Starvation



Priority Boost!

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

RECAP: MLFQ 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

Proportional Share in Scheduling

Metrics so far: turnaround time, response time

New metrics: proportional share and fairness

E.g., if Job A has paid 5x more price than Job B, then A is 5 times more likely to get the CPU than B

If both paid equally, A and B are equally likely to get the CPU

Lottery Scheduling

A	10 tickets
B	20 tickets
C	30 tickets

Give each job tickets

Conduct a lottery periodically to see who is winner

Winner gets scheduled on CPU

Higher priority job is given more tickets

Any number between 0 and 9 – A gets to run

Any number between 10 and 29 – B

Any number between 30 and 59 – C

If this workload mix were running for 1 hr, how much CPU time roughly C would have gotten?

Implementing Lottery Scheduling

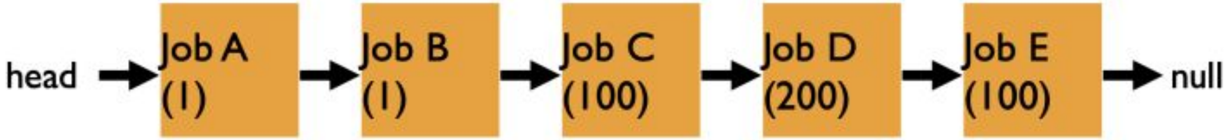
```
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t *current = head;
while (current) {
    counter += current->tickets;
    if (counter > winner) break;
    current = current->next;
}
// current gets to run
```

Who runs if winner is:

50

350

0



More about Lottery Scheduling

Ticket transfer – temporarily transfer tickets from one process to another

When is this useful?

How to assign tickets?

Lottery vs. RR

With equal tickets, all jobs are equally likely to picked up for running in lottery

RR also behaves the same way…

What is the difference then? (chat with neighbors for 2 mins and find out!)

Linux CFS

Completely fair scheduler - used in Linux

Scheduling efficiency is a key goal – Google data center studies show that 5% CPU time spent in making scheduling decisions!

CFS high-level: fairly divides CPU evenly among all processes

Vruntime to track how much CPU time a process has gotten so far, pick the one with least vruntime

Tension:

If need to decide too often → more fair but lots of overhead

If decide too infrequently → little overhead but less fair

Linux CFS - control parameters

Sched_lat: 48 ms typical value

Time slice = 48/Num process (dynamically decides quantum)

Example: with 4 processes, each gets 12 ms. If two exist, each of the rest two will get 24 ms

If too many processes, then too much scheduling overhead, use a min_granularity control parameter (which is usually 6 ms)

Example: with 10 processes, each will get 6 ms

More details: takes care of Nice values and uses fast data structures to reduce overhead

CPU SUMMARY

Mechanism

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

Policy

- Metrics: turnaround time, response time
- Balance using MLFQ
- Fairness and proportional share

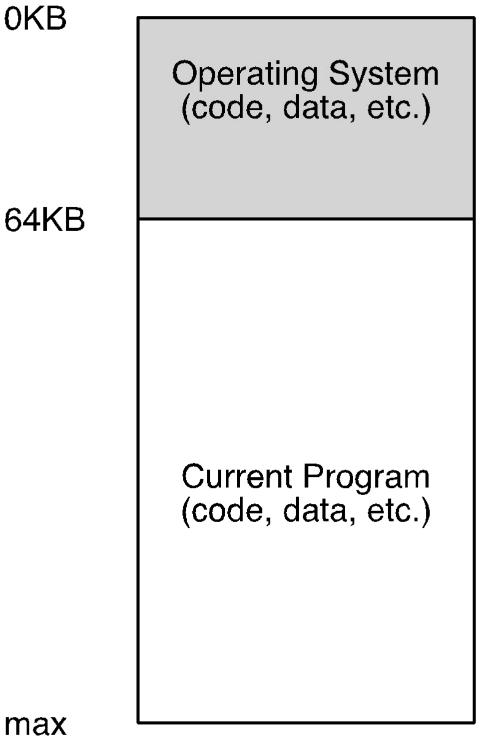
Virtualization

1st part of this course: Virtualization

First, OS virtualizes the CPU: illusion of private CPU for each process

Now, how OS virtualizes the memory: illusion of private memory for each process

BACK IN THE DAY…



Early systems did not virtualize memory

Uniprogramming: One process runs at a time

Disadvantages?

- Only one process ready at a time
- Process can destroy OS

MULTIPROGRAMMING GOALS

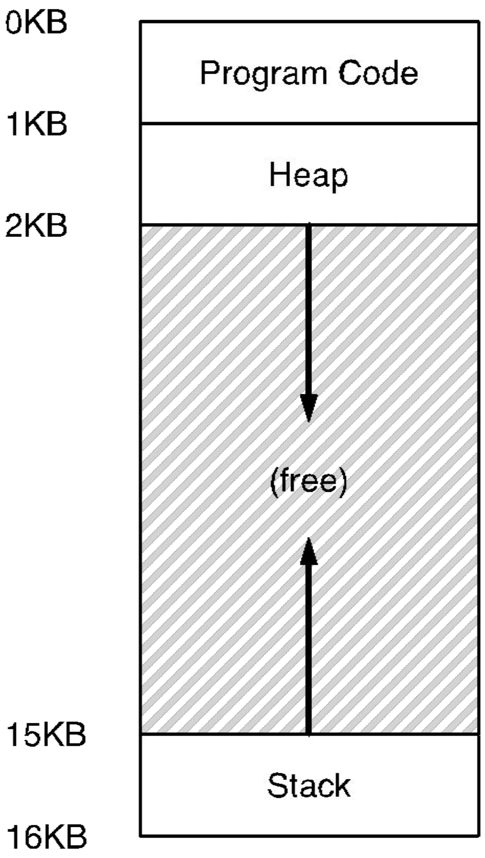
Transparency: Process is unaware of sharing

Protection: Cannot corrupt or read OS’s or other process’ memory

Efficiency: Do not waste memory (no fragmentation) or slow down processes

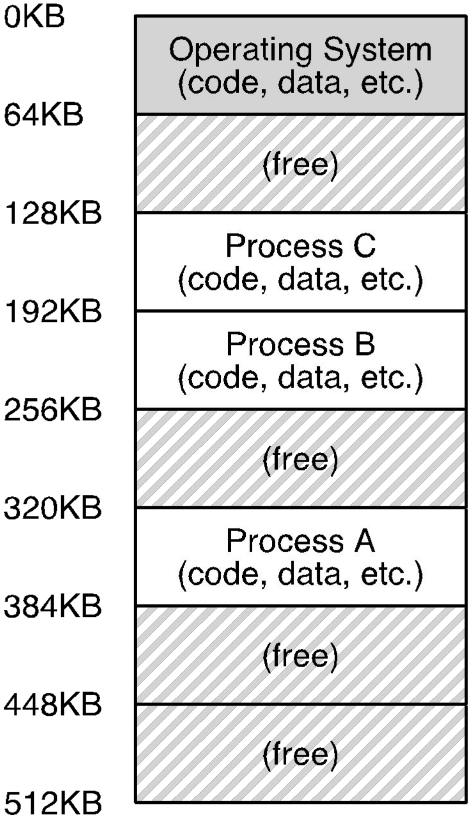
Sharing: Enable sharing between cooperating processes

ABSTRACTION: ADDRESS SPACE

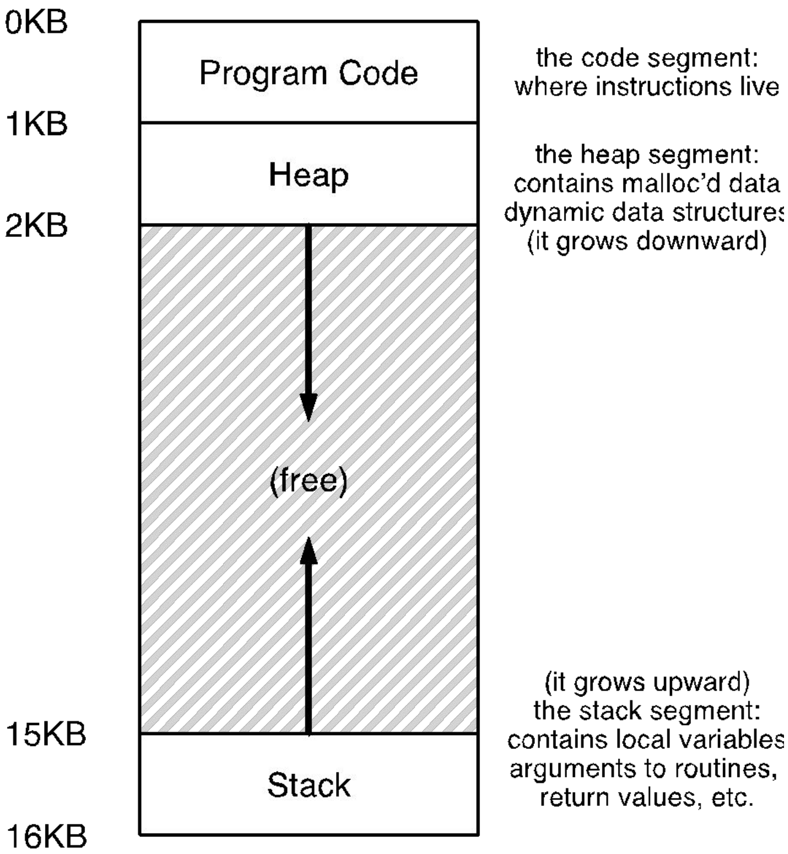


Address Space:
Each process has its own set of addresses

OS aims to provide Illusion of private memory



WHAT IS IN ADDRESS SPACE?



Static: Code and some global variables

Dynamic: Stack and Heap

Why do we put stack and heap at the opposite ends?

What happens with multiple threads?

Why need dynamic memory?

Why do processes need dynamic allocation memory?

Do not know how much memory needed at compile time

Must do worst-case assumption and allocate more

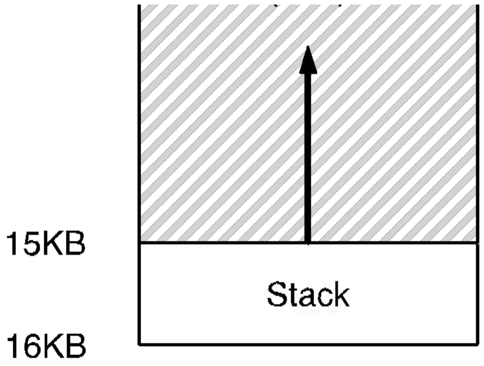
Complex data structures: trees, graphs, lists – cannot say how much will be allocated: malloc(size(struct my_struct)

Recursive procedures: cannot say how many times procedure will be nested

Two types of dynamic allocation: heap and stack

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!

HEAP ORGANIZATION

Allocate from any random location: malloc(), new()

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

Adv: works for all data structures

Disadv:

Can be fragmented

How to allocate 20 bytes?

What is OS's role in managing the heap?

OS gives a big chunk of free memory process, library manages individual allocations

Possible locations:
static data/code, stack, heap

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

Address	Location
x	
main	
y	
z	
*z	

MEMORY ACCESS

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

int main(int argc, char *argv[]) {
  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

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)

How many memory accesses?

To what addresses?

Chat with neighbors for 2 mins.

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

HOW TO VIRTUALIZE MEMORY

Problem: How to run multiple processes simultaneously?

Addresses are “hardcoded” into process binaries

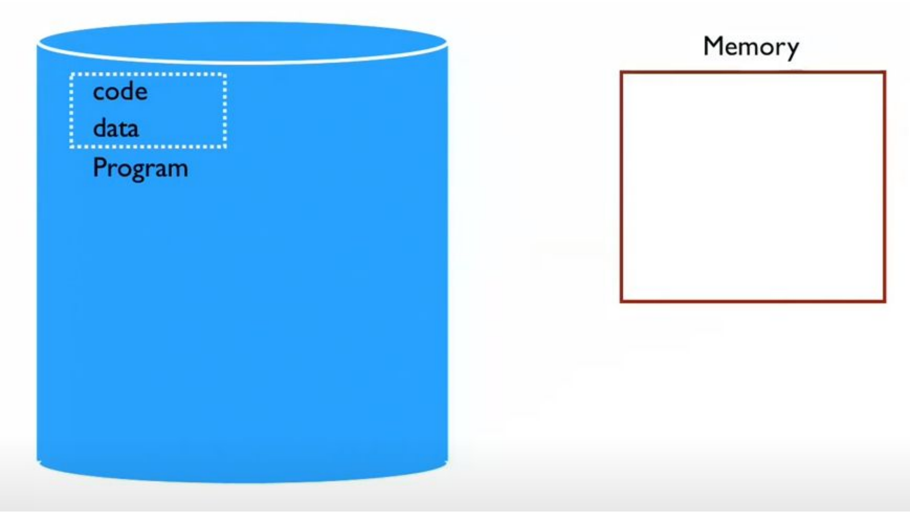
How to avoid collisions?

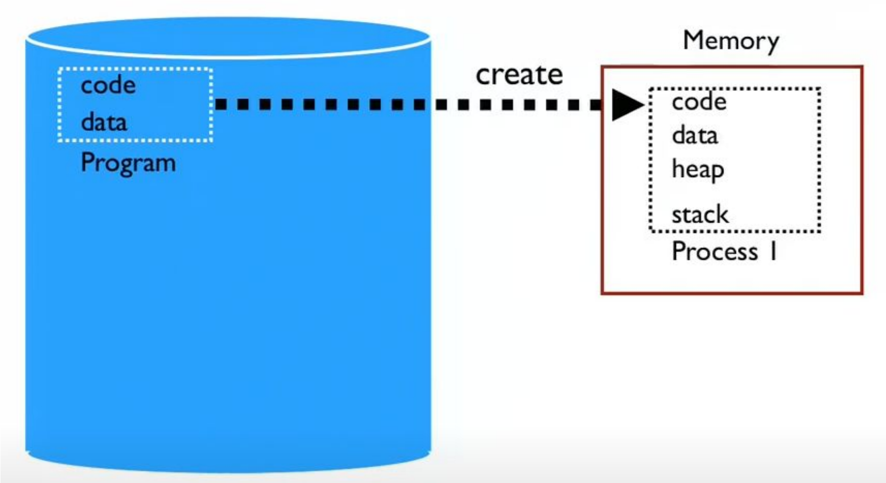
Possible Solutions for Mechanisms (covered today):

1. Time Sharing
2. Static Relocation
3. Base
4. Base+Bounds

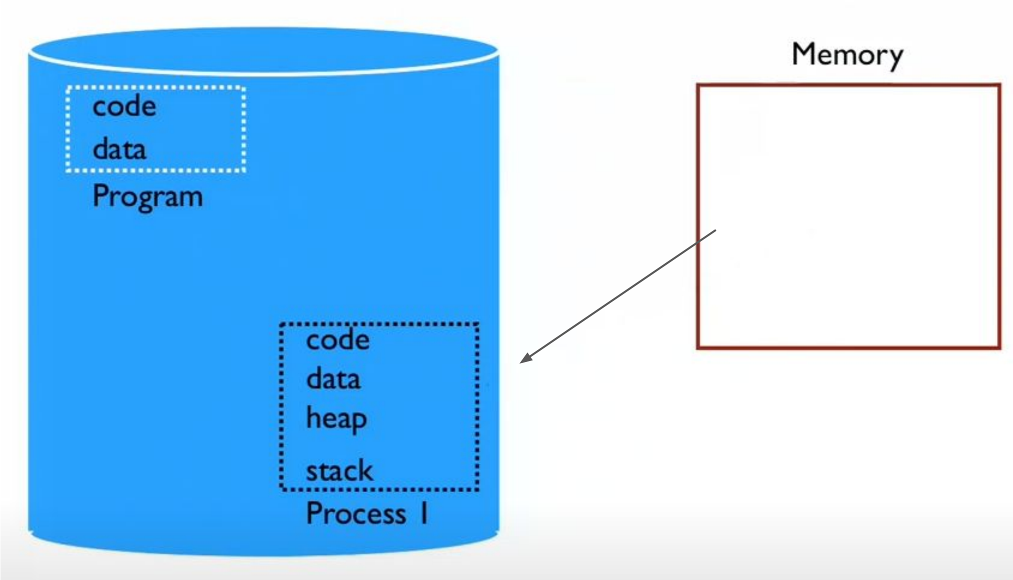
Assumptions: each AS is the same size, AS is smaller than physical memory, AS can be placed contiguously in physical memory (not realistic…)

1. TIME SHARE MEMORY





Time Sharing Memory



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

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

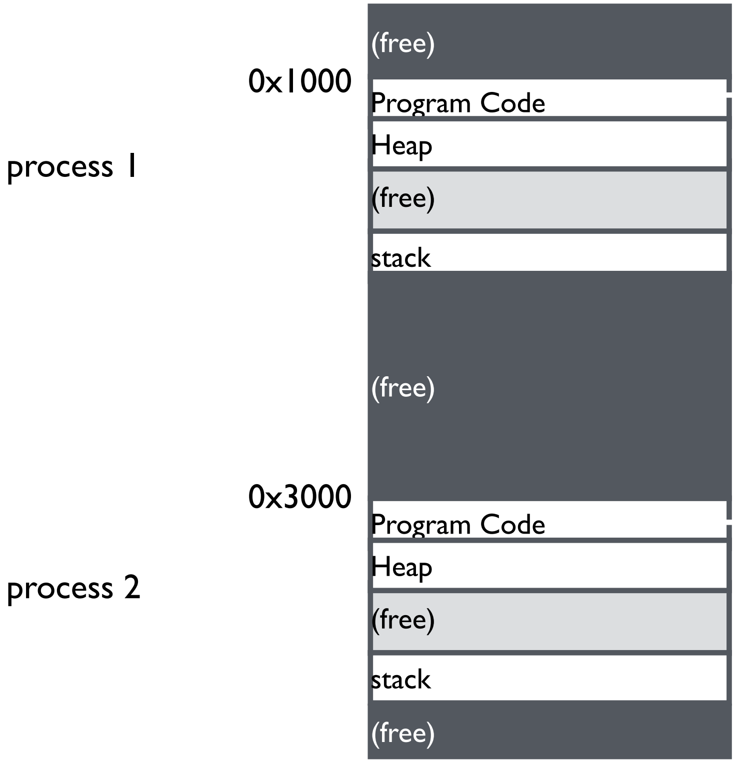
rewrite

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

rewrite

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

Static: Layout in Memory




```
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

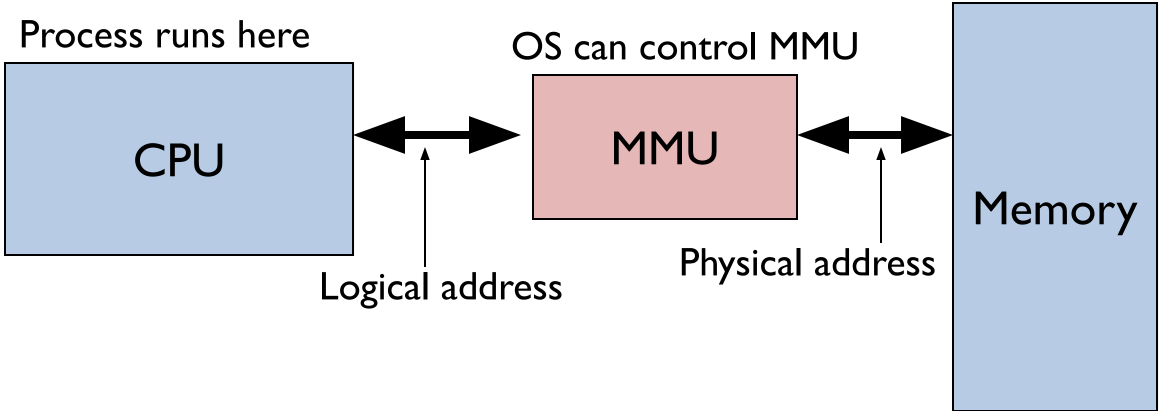
Goal: Protect processes from one another

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



HW 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

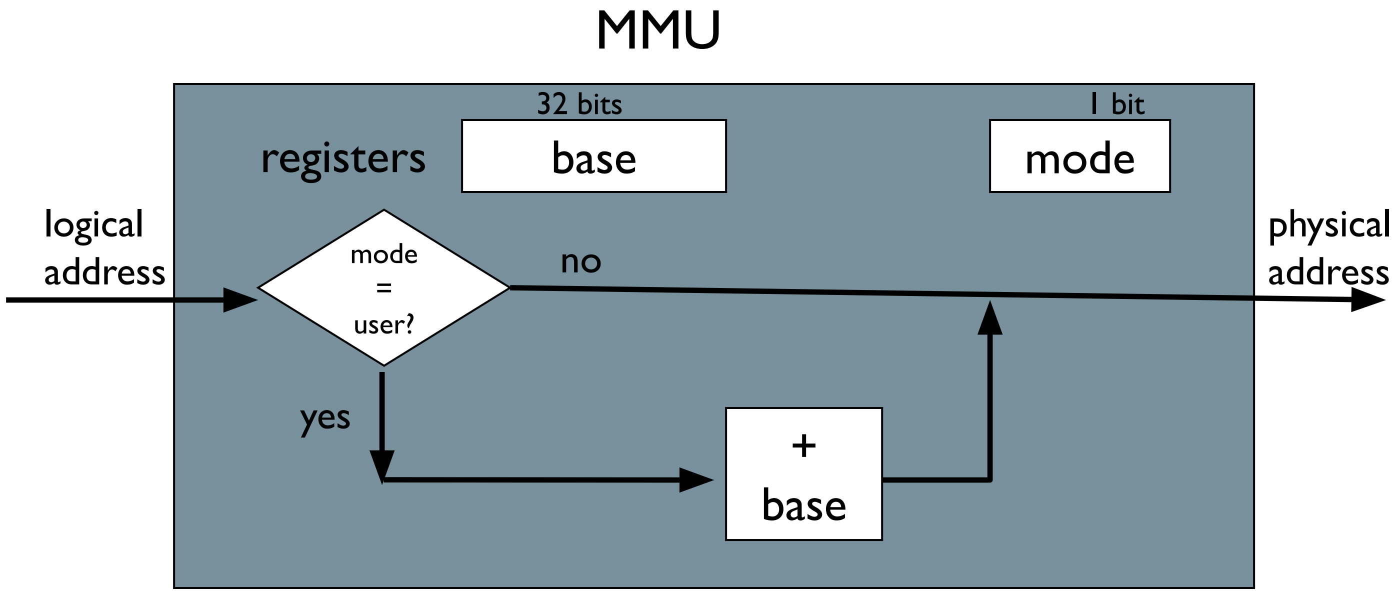
User mode: User processes run

- 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



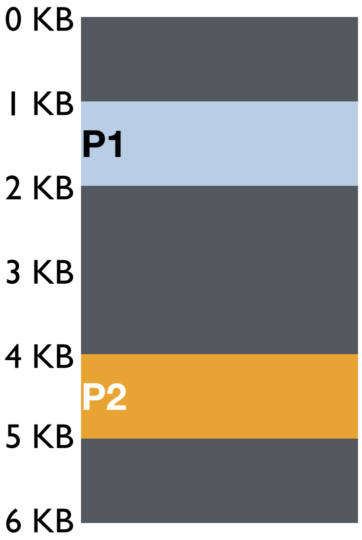
Dynamic Relocation with Base Register

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!



Virtual

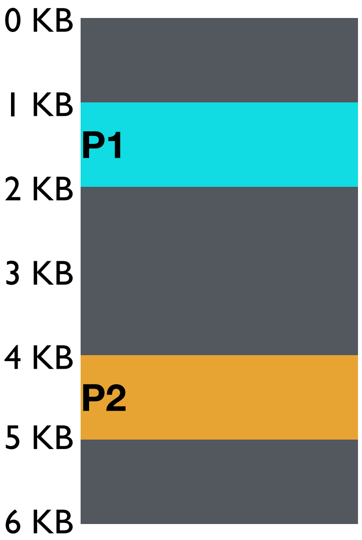
P1: load 100, R1

P2: load 100, R1

P2: load 1000, R1

P1: load 1000, R1

VISUAL Example of DYNAMIC RELOCATION: BASE REGISTER



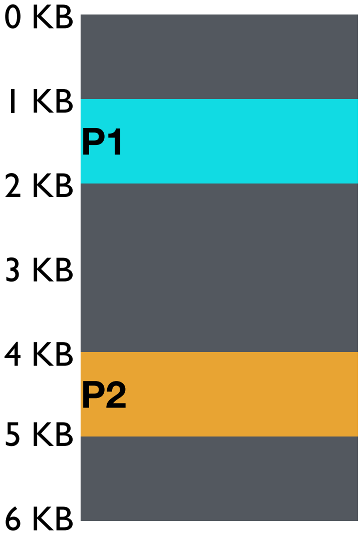
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5096, R1
P1: load 1000, R1	load 2024, R1

Does the base register contain physical or virtual address?

Who converts VA to PA based on base register? Process? OS? HW?

Who modified contents of base register?

What happens on a context switch to the base register?



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5096, R1
P1: load 1000, R1	load 2024, R1

Can P2 hurt P1?

Can P1 hurt P2?

How well does dynamic relocation do with base register for protection?

Virtual	Physical	
P1: load 100, R1	load 1124, R1	
P2: load 100, R1	load 4196, R1	
P2: load 1000, R1	load 5096, R1	
P1: load 100, R1	load 2024, R1	
P1: store 3072, R1	store 4096, R1	(3072 + 1024)

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

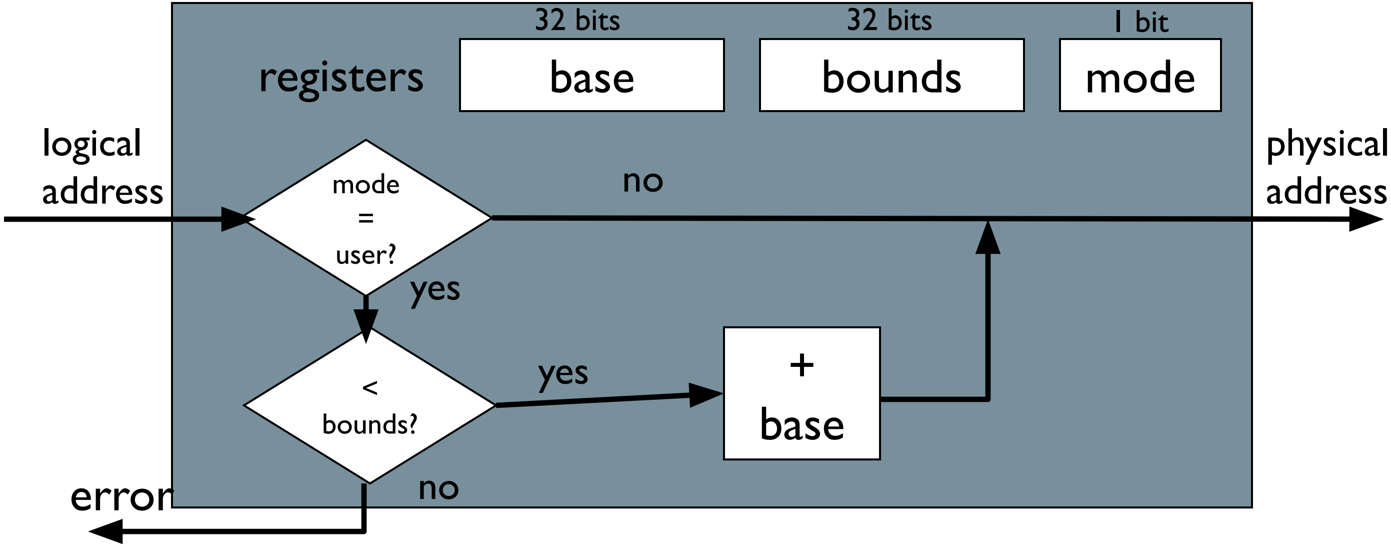
- Sometimes defined as largest physical address (base + size)

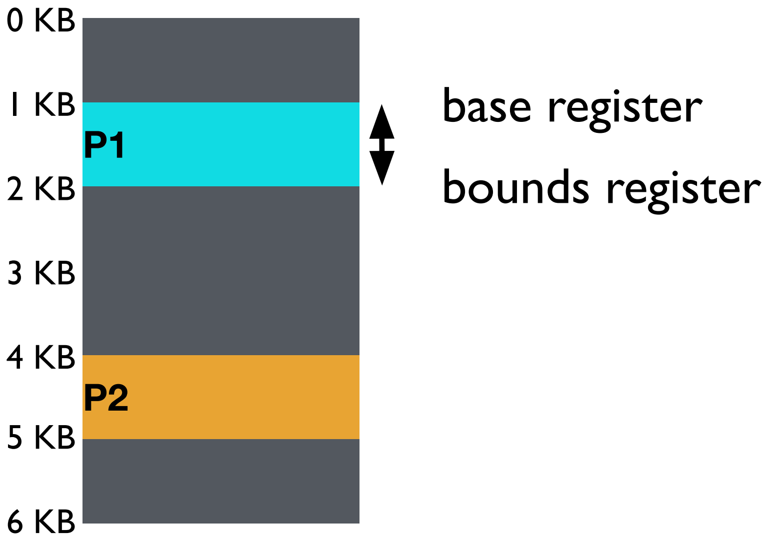
OS kills process if process loads/stores beyond bounds

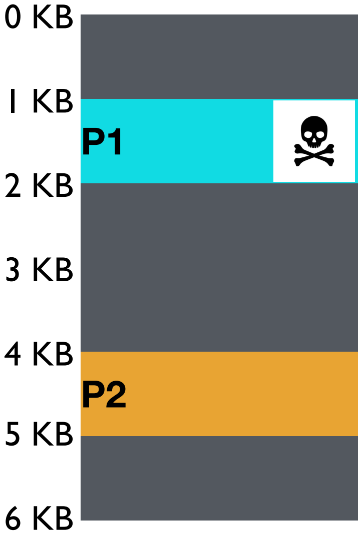
Implementation of BASE+BOUNDS

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



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	load 2024, R1
P1: store 3072, R1	Interrupt OS!

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

Are there cases where you won’t change Base and Bounds register during context switches?

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

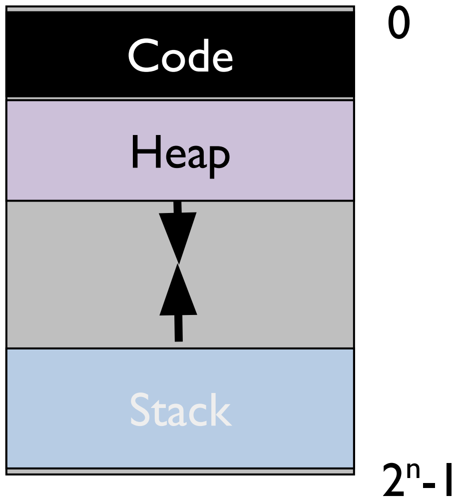
Simple, inexpensive implementation: Few registers, little logic in MMU
Fast: Add and compare in parallel

Base and Bounds DISADVANTAGES

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 LECTURES…

Remove these disadvantages

- Segmentation
- Paging
- Segmentation + Paging
- Multi-level Page Tables
- TLBs