MEMORY MANAGEMENT

Kai Mast

CS 537

Fall 2022

Announcements

- P1B is out
- My office hours are only on Tuesdays (not Thu) from now on
  - You can always schedule one for another time!
- I will try to upload slides the day before the lecture
  - But there might be minor changes and fixes later

Recap: Scheduling

Scheduler: Decides which job to run

Job: A CPU burst of a process

Turnaround Time: Time from arrival to completion

Response Time: Time from arrival to first time running

Starvation: Job never gets scheduled

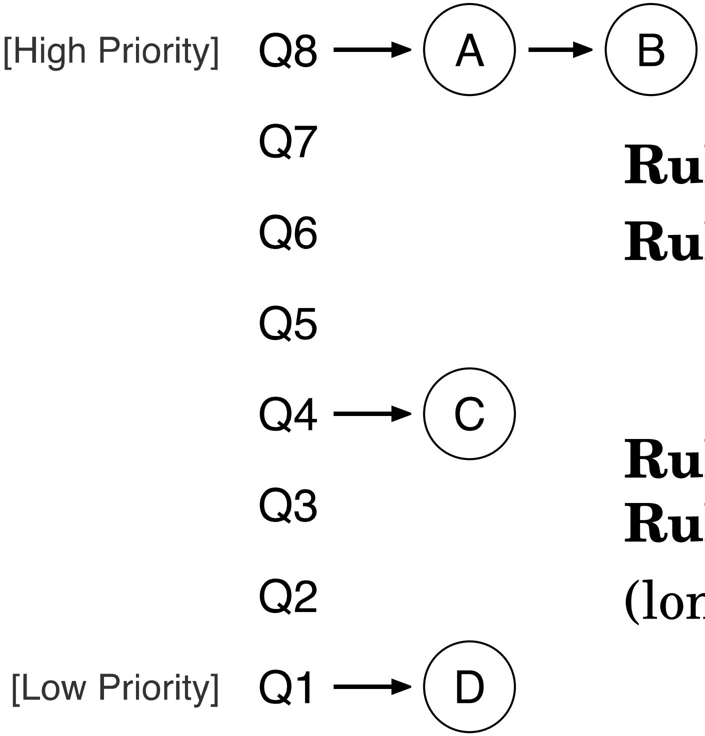
SCHEDULING POLICIES

FIFO:
- The job that arrives first, runs first

Shortest-Job First:
- Run the shortest ready job
- Best for minimizing average turnaround time

Round-Robin:
- At the end of every time slice, preempt the current process and switch to the next one
- Improves response time

MULTI-LEVEL FEEDBACK QUEUE (MLFQ)



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

Rule 2: If priority(A) == Priority(B),
A & B run in Round-Robin

Rule 3: Processes start at top priority

Rule 4: If job uses whole slice, demote process

(longer time slices at lower priorities)

MLFQ INTUITION

Detect interactive processes

- Will frequently wait for user (and other) I/O

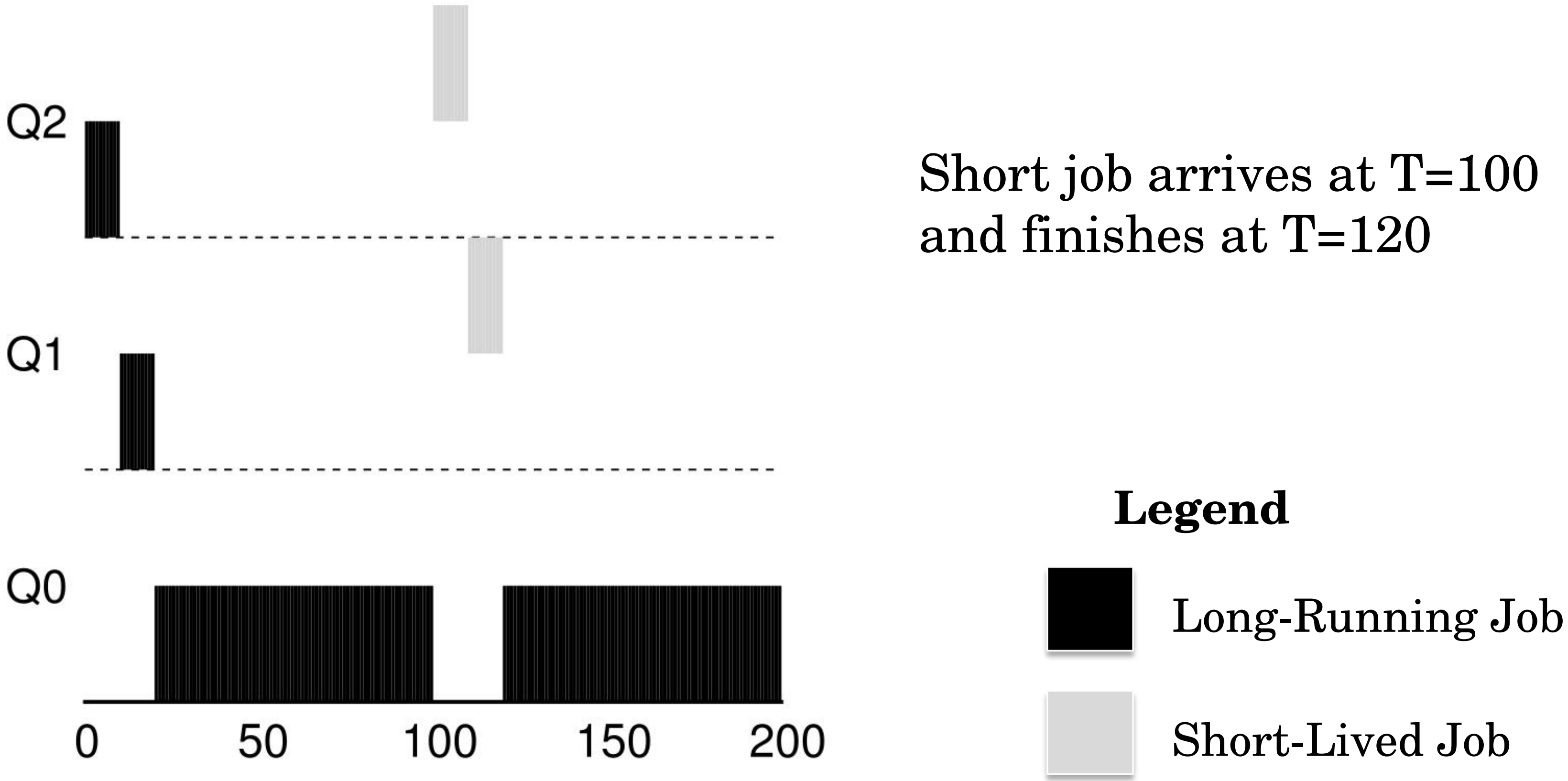
Schedule batch processes less frequently

- Usually not time sensitive

Reset levels regularly by boosting all processes to the highest level

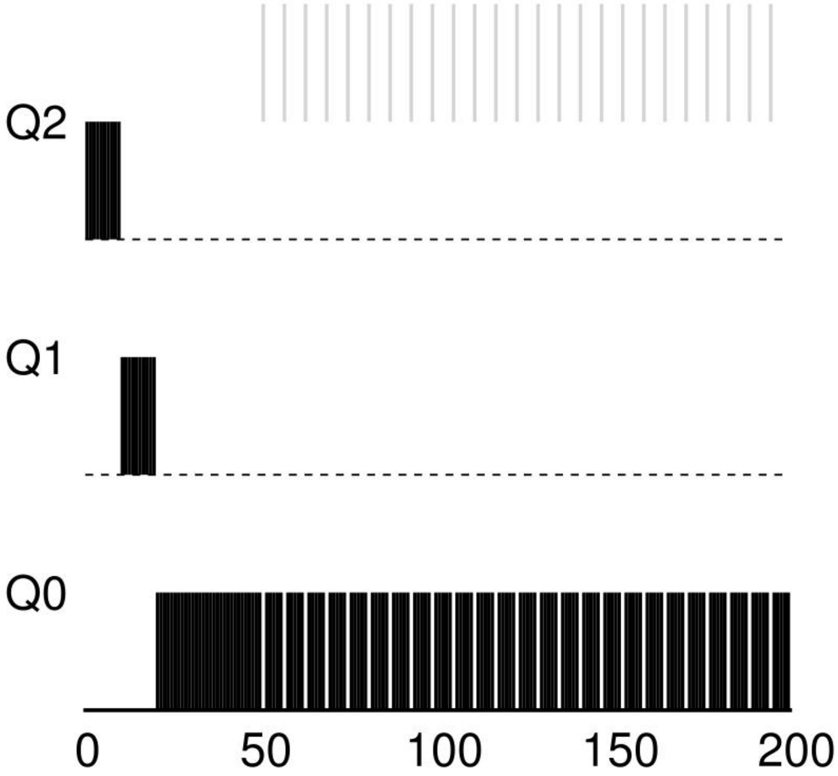
- Allows adapting to workload changes

MLFQ EXAMPLE (NO I/O)

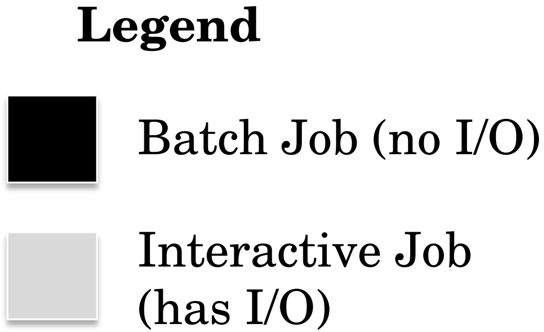


Short job arrives at T=100 and finishes at T=120

MLFQ EXAMPLE (WITH I/O)



Interactive job does not use the full slice and stays at highest level



INTERLUDE: PROCESS MANAGEMENT

PROCESS CREATION

Two ways to create a process

Option 1: Build a new process from scratch

Option 2: Copy an existing process and change it appropriately

OPTION 1: NEW PROCESS

Create new process with specified executable and state

- Load specified code and data into memory; Create empty call stack
- Create and initialize PCB (make look like context-switch)
- Put process on ready list

Advantages: No wasted work

Disadvantage: Difficult to setup process and to express all possible options

- Process permissions, where to write I/O, environment variables
- Example: Windows NT has call with 10 arguments

Option 2: Copy and Change

Copy existing process (fork) and change as needed (exec)

- `fork()`
  - Calling process (parent) creates a child process
  - Make copy of code, data, stack, and PCB of parent
  - Add new PCB to ready list

- `exec(const char *file)`
  - Replace current data and code segments with those in specified executable file

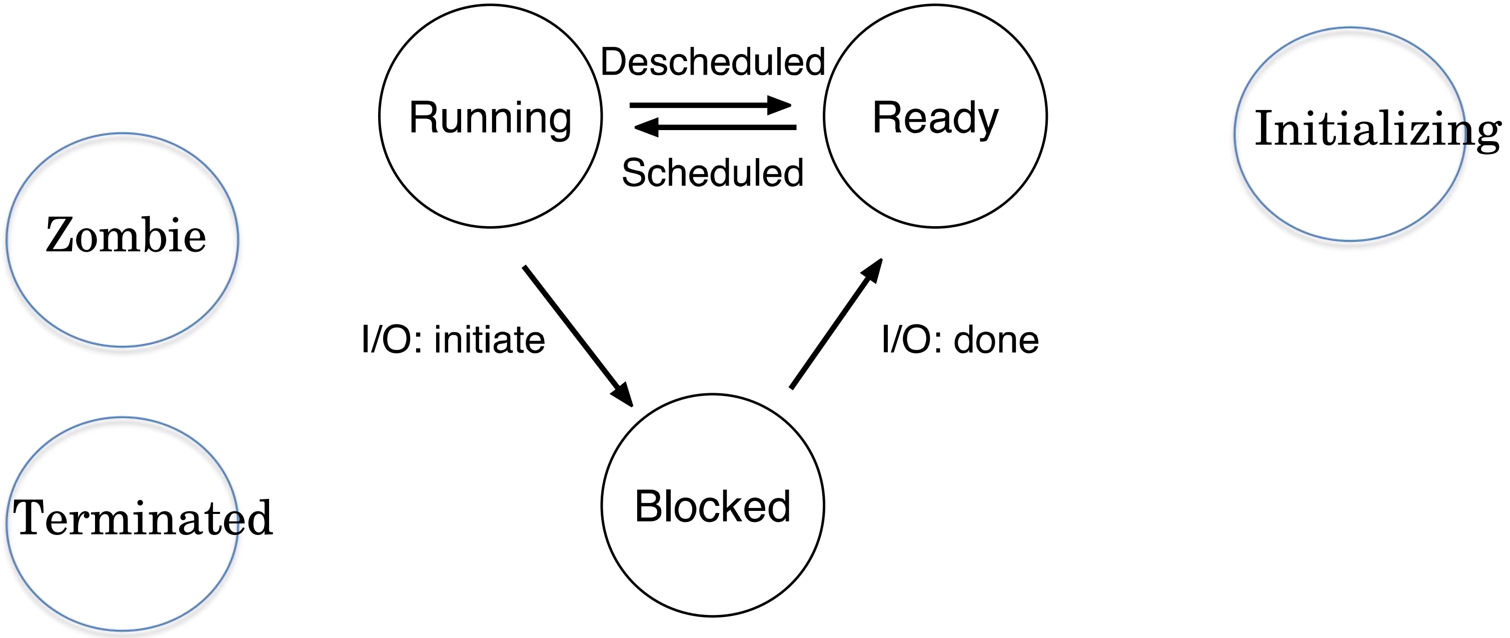
Advantages: Flexible, clean, simple

Disadvantages: Wasteful to perform copy and then overwrite of memory

UNIX SHELLS

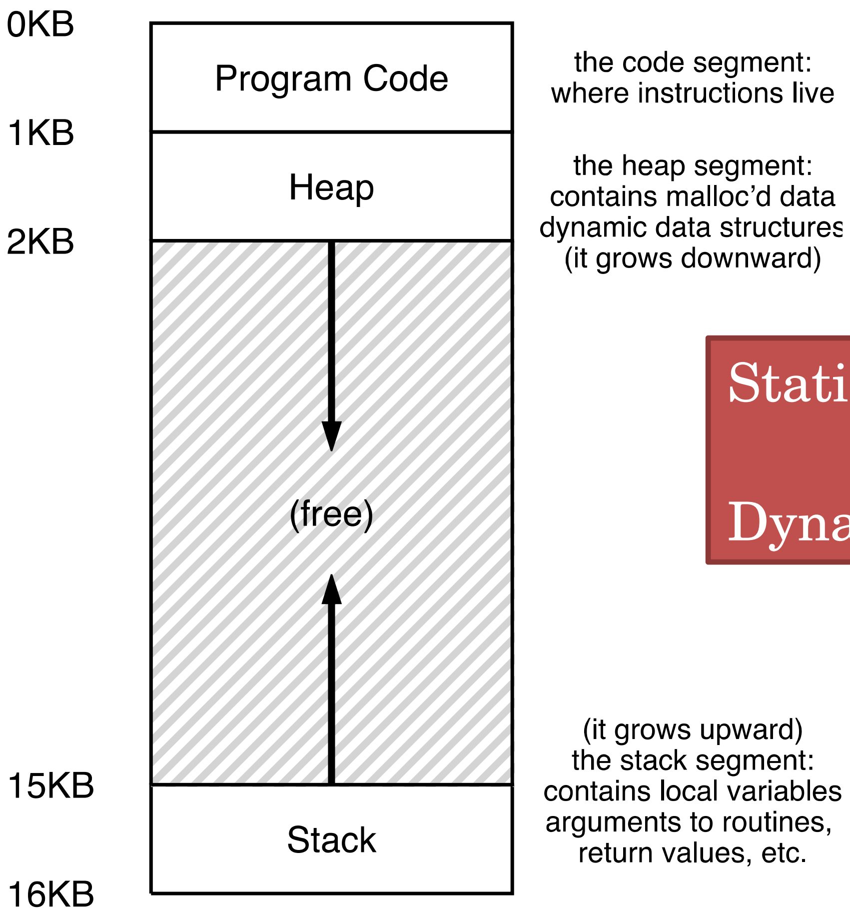
```
while (true) {
  char *cmd = getcmd();
  int retval = fork();

  if (retval == 0) {
     // This is the child process
     // Setup the child’s process environment here
     // E.g., where is standard I/O, how to handle signals?
     exec(cmd);
   } else if (retval > 0) {
     // This is the parent process; Wait for child to finish
     int pid = retval;
     wait(pid);
   } else {
     // Handle errors here
   }
}
```

STATE TRANSITIONS EXTENDED



MEMORY MANAGEMENT

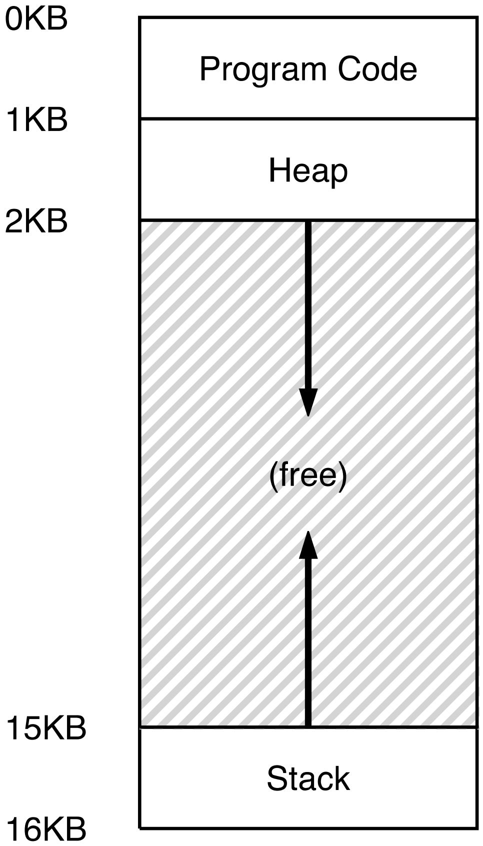
BACKGROUND: WHAT IS AN ADDRESS SPACE?



Static: Code and some global variables

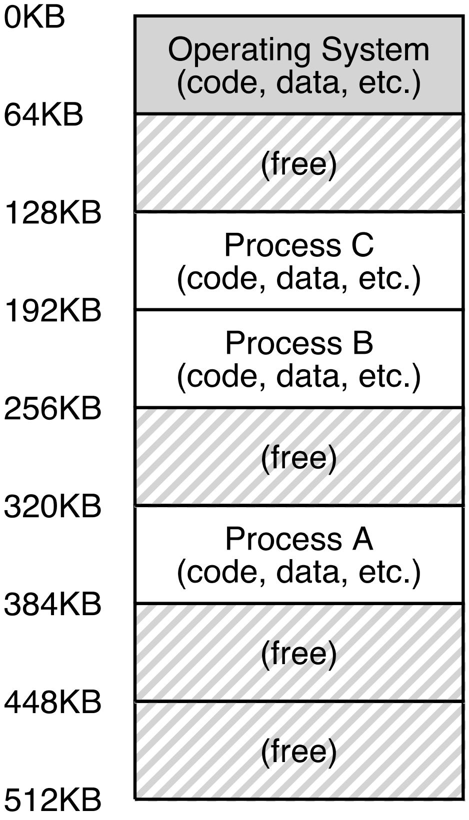
Dynamic: Stack and Heap

ABSTRACTION: ADDRESS SPACE



Each process has own set of addresses

How can the OS provide illusion of private (virtual) address space to each process?



REVIEW: MEMORY ACCESS (LOGICAL)

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)

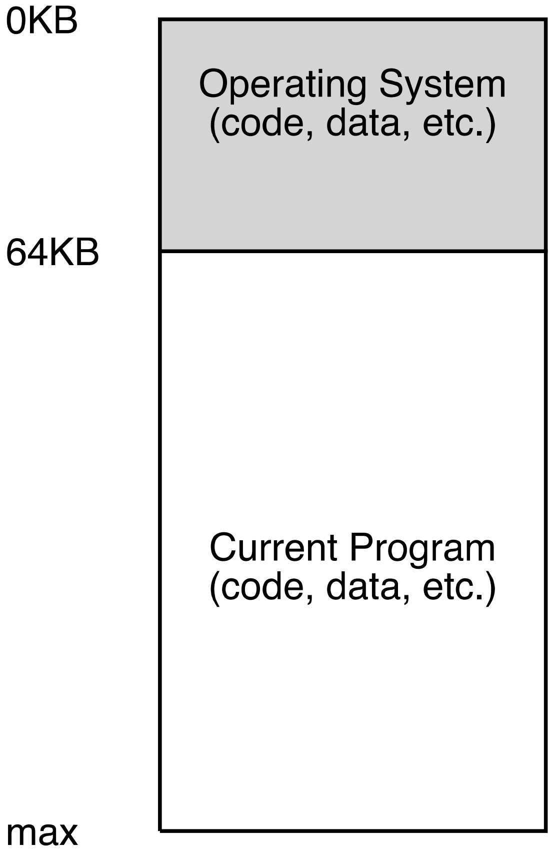
Fetch instruction at addr 0x10
Exec:
load from addr 0x208

Fetch instruction at addr 0x13
Exec:
addition (no memory access)

Fetch instruction at addr 0x19
Exec:
store to addr 0x208

5 total memory accesses

MOTIVATION FOR VIRTUALIZING MEMORY



First systems did not virtualize memory

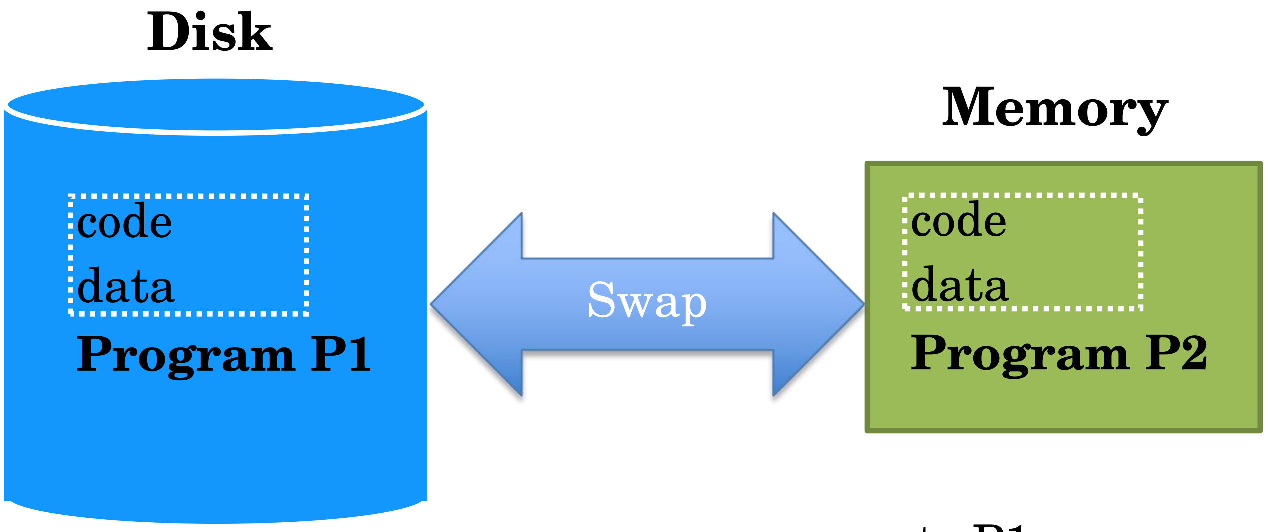
Uniprogramming: One process runs at a time

Disadvantages

- Only one process be ready at a time
- Process can modify OS

Solutions?

TIME-SHARE MEMORY



- create P1
- swap out P1
- create P2
- swap out P2
- swap in P1
- ...

Problems with Time Sharing

Ridiculously poor performance (disk are slow!)

Better Alternative:

- Space sharing of physical memory
- At same point in time, space of memory is divided across processes
- Remainder of solutions all use space sharing

Static Relocation

Rewrite each code segment before loading it in memory

- Pick static physical location for each process when started
- Each rewrite for different process uses different addresses and pointers
- Change jumps, loads of static data

STATIC RELOCATION

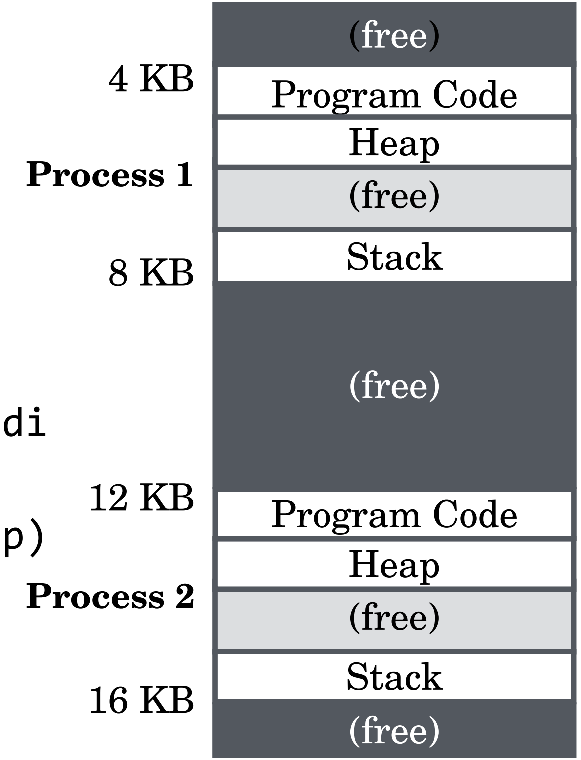
```
0x10:movl 0x8(%rbp), %edi
0x13:addl $0x3, %edi
0x19:movl %edi, 0x8(%rbp)
```

Rewrite for P1

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

Rewrite for P2

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



Why did the OS not rewrite the stack address?

STATIC RELOCATION: DISADVANTAGES

No Protection

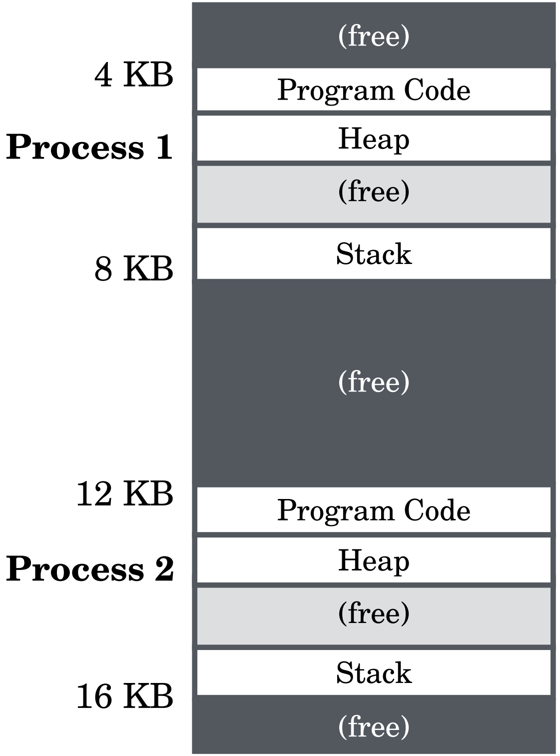
- Any process can modify memory of the OS or other processes

No Security

- All memory is visible to all processes

No Dynamic Allocation

- Cannot move address space after it has been placed
- May not have free space to allocate new process



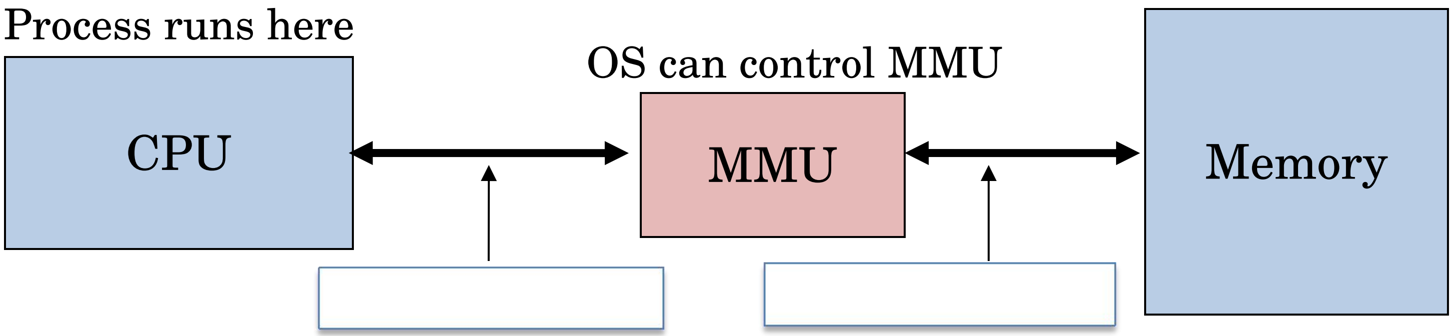
DYNAMIC RELOCATION

Goal: Protect processes from one another (and the OS from processes)

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

Leverage privilege levels (introduced with system calls)

Non-privileged (user) mode: Processes run

- Perform translation of logical address to physical address

Privileged (protected, kernel) mode: OS runs

- When enter OS (trap, system calls, interrupts, exceptions)
- Allows privileged instructions to be executed
  - Can manipulate contents of MMU
- Allows OS to access all of physical memory

DYNAMIC RELOCATION USING BASE REGISTERS

Base Registers

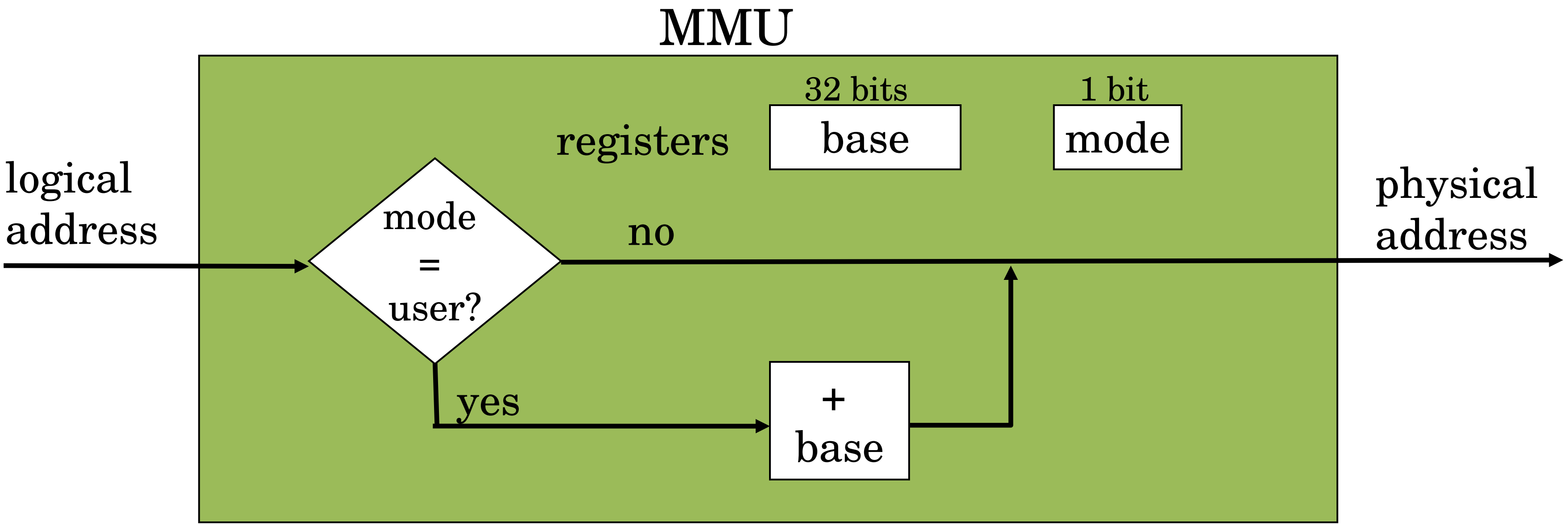
- Stored in the MMU
- Each process has different value in base register
- Set by the OS on context switch

Address Translation using Base Registers

- Add base register to virtual address on every memory access
- Dynamic relocation by changing value of base register!

BASE REGISTERS: IMPLEMENTATION

- Translation on every memory access of user process
- MMU adds base register to logical address to form physical address



PROTECTION WITH BASE REGISTERS?

Process	Base
P1	1024
P2	4096

Possible Execution

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
P1: store 3072, R1	**store 4096, R1**

P1 can modify P2’s address space!