Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 11: Address Spaces

Prof. Ryan Huang

Administration

Midterm exam next Wednesday (02/26)

- From 5 pm to 7 pm

Format

- In person, closed book
- No electronic devices allowed

Scope

- Lecture materials until the end of Deadlock
- Content in labs, P1, and P2

Project 2: writing good test cases

A good test suite provides comprehensive coverage

Coverage along what dimension?

- Code coverage
- Specification coverage

Example: test if lock() blocks when mutex is held

Thread A	Thread B
`Create thread B`	
`m.lock()`	`m.lock()`
`m.unlock()`	`m.unlock()`

Example: test if lock() blocks when mutex is held

Thread A

```
Create thread B
m.lock()
Yield()
Print A1
m.unlock()
```

Thread B

```

m.lock()
Print B1
m.unlock()
```

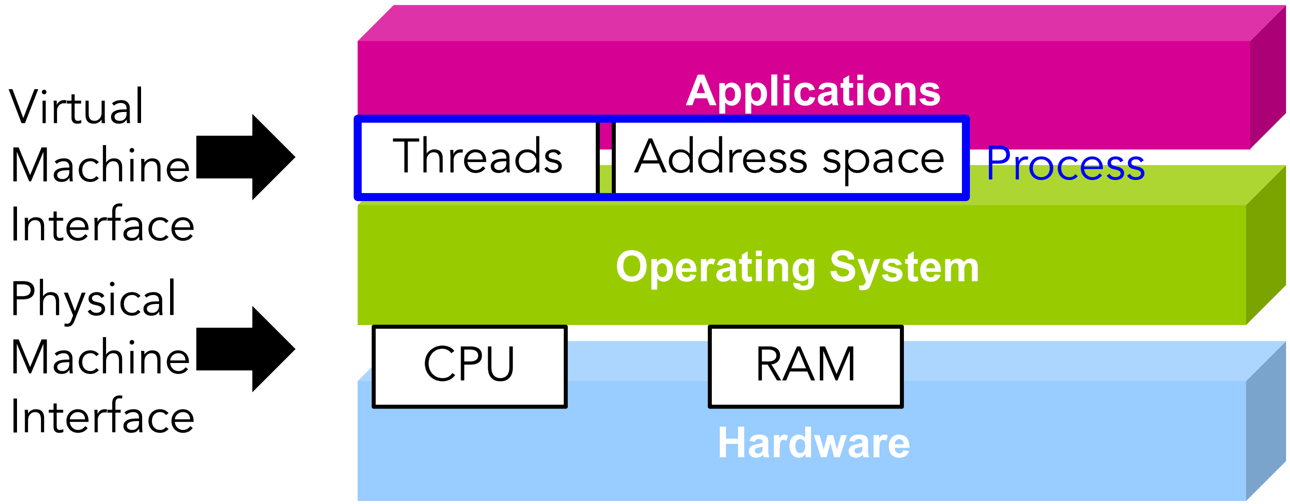
Writing good test cases

Specification coverage

- Targeted test cases (micro test case)
- Stress test (macro test case)

Which is better for debugging?

OS abstractions



Process = one or more threads in an address space

Address space

All the memory space the process can use as it runs

- Code
- Stack
- Data segment

Physical machine interface: single memory shared by all jobs

Virtual machine interface: each process has its own memory

Address space abstractions

Address independence:

- same numeric address can be used in different address spaces, yet refer to distinct data items

Protection (controlled sharing):

- one process can't access data in another process's address space
  - for both writes and reads

Large address space:

- address space for a process can be larger than physical memory

Uni-programming

OS run one job at a time

Only 1 process in physical memory at a time

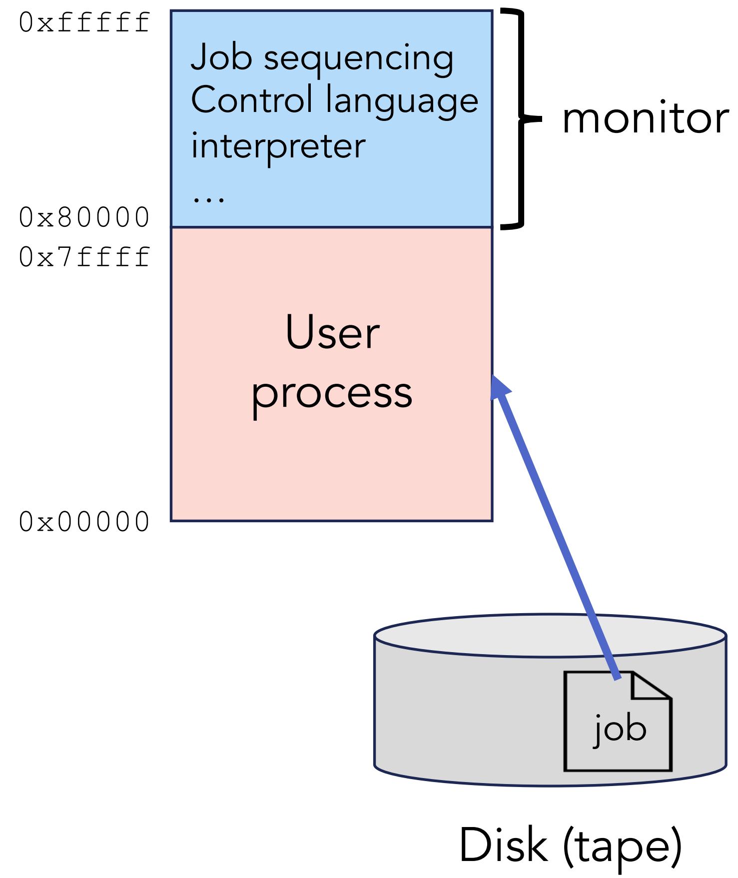
- Loaded to the same address range

OS always in memory in reserved space

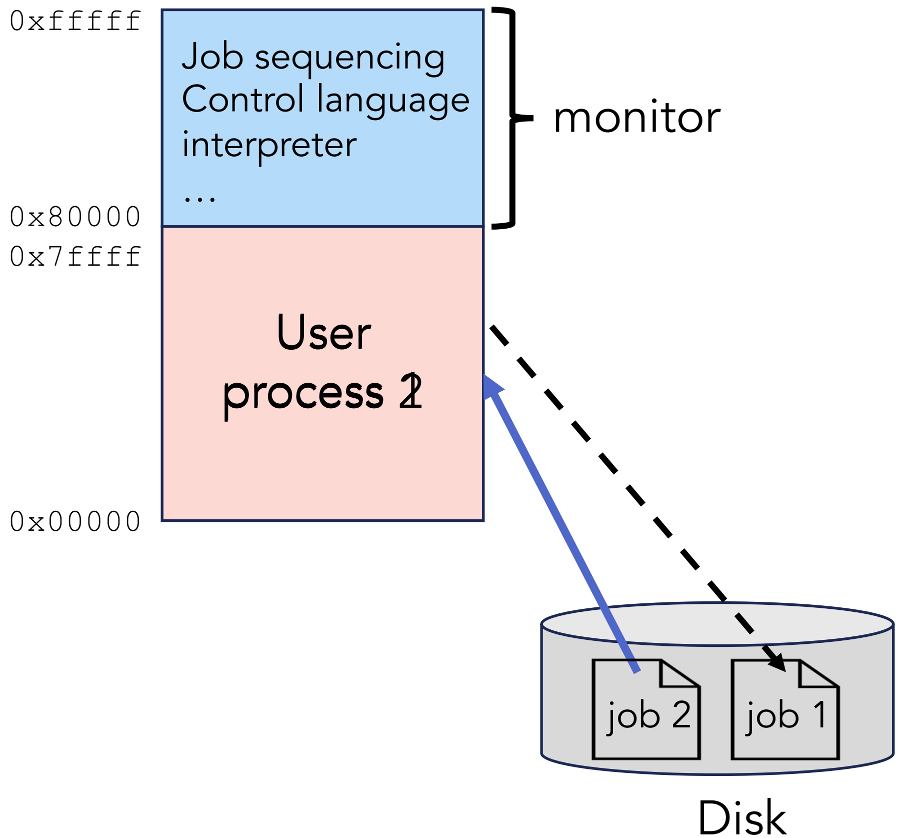
- Also called resident monitor

No virtual memory

- Process' view of memory = hardware view (physical address)



Switching processes with uni-programming



Uni-programming summary

Address space abstractions

- ✓ Address independence?
- ✓ Protection?
- ✘ Large address space?

Simple

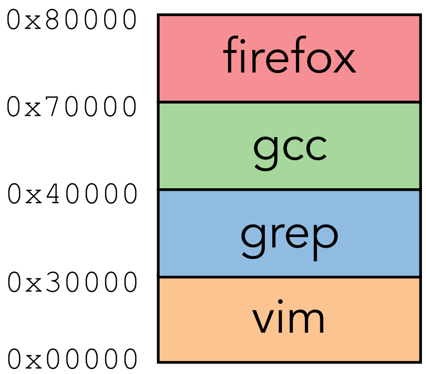
- Early operating systems used this

Inefficient

- (un)loading an entire job each time, expensive to switch jobs

Multi-programming

Allow multiple processes to reside in physical memory at the same time

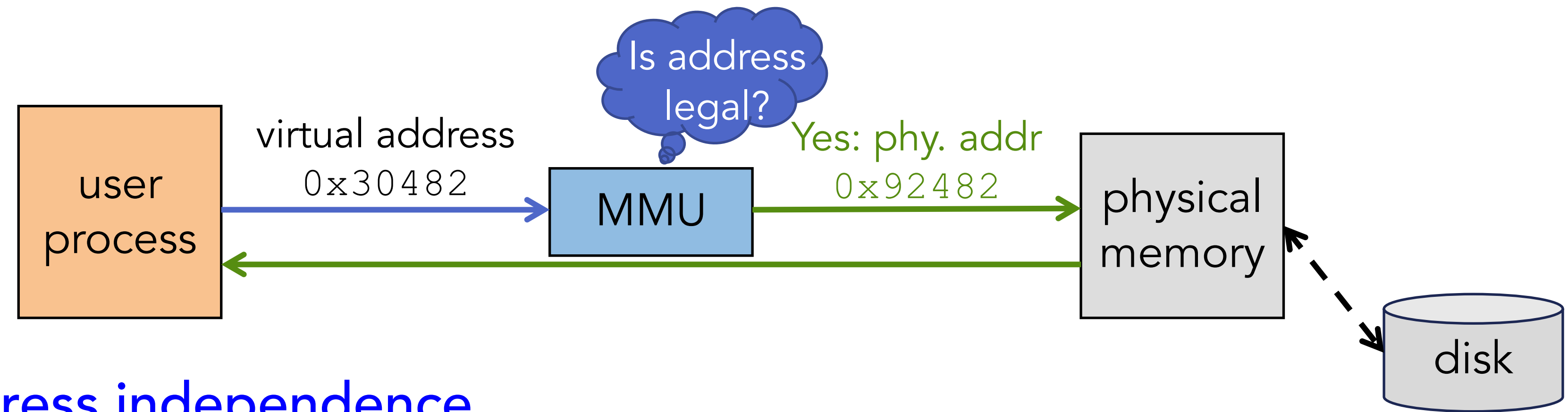


Challenges

- Harder to provide protection
  - E.g., what if `grep` has a bug and writes to `0x7100`?
- Harder to provide address independence
  - `grep` and `firefox` cannot both use address `0x7100`
  - `gcc` will have to know it will run at `0x4000` (when?)

Providing protection and address independence requires system to do some work on each memory access (!)

Dynamic address translation



Address independence

- Give each program its own virtual address space
- At runtime, *Memory-Management Unit* (MMU) relocates each load/store
- Application doesn't see physical memory addresses

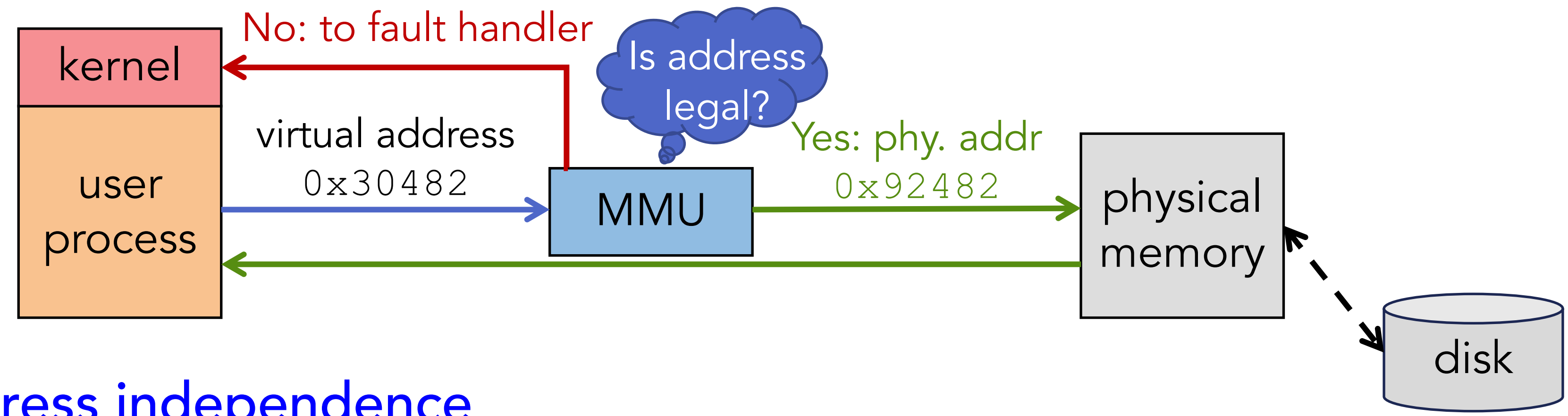
Protection

- MMU ensures that processes don't access each other's data

Large address space

- Somehow relocate some memory accesses to disk

Dynamic address translation



Address independence

- Give each program its own virtual address space
- At runtime, *Memory-Management Unit* (MMU) relocates each load/store
- Application doesn't see physical memory addresses

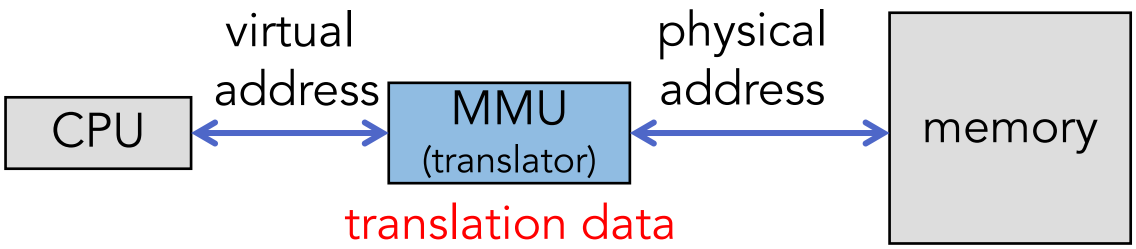
Protection

- MMU ensures that processes don't access each other's data

Large address space

- Somehow relocate some memory accesses to disk

Dynamic address translation



MMU is usually implemented as part of CPU

- Configured through privileged instructions
- Gives per-process view of memory called address space

MMU uses translation data

- Each process has its own translation data
- Changing address spaces → changing translation data

Many ways (data structures) to implement translator

- Speed of translation
- Size of data needed to support translation
- Flexibility (sharing, growth, large address space)

More on large address space

Can re-locate program while running

- Run partially in memory, partially on disk

Most of a process's memory may be idle (80/20 rule)

- Write idle parts to disk until needed
- Let other processes use memory of idle part
- Like CPU virtualization
  - when process not using CPU → switch
  - not using a memory region? → give it to another process

Challenge: VM = extra layer, could be slow