

# The Process

A running program

(Chapters 2-6)

# From Program to Process

- ☉ To make the program's code and data come alive
  - need a CPU
  - need memory — the process' address space
    - for data, code, stack, heap
  - need registers
    - PC, SP, regular registers
  - need access to I/O
    - list of open files



# A First Cut at the API

- ① Create

  - causes the OS to create a new process

- ② Destroy

  - forcefully terminates a process

- ③ Wait (for the process to end)

- ④ Other controls

  - e.g. to suspend or resume the process

- ⑤ Status

  - running? suspended? blocked? for how long?

# How the OS Keeps Track of a Process

- 1 A process has code

OS must track program counter

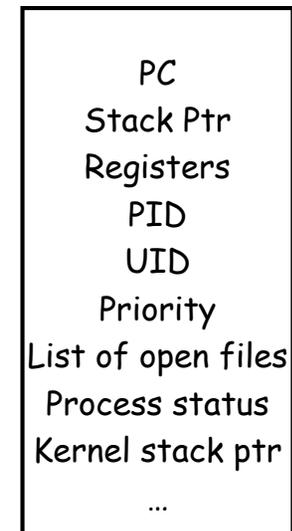
- 2 A process has a stack

OS must track stack pointer

- 3 OS stores state of process in Process Control Block (PCB)

Data (program instructions, stack & heap) resides in memory, metadata is in PCB

Process Control Block



# You'll Never Walk Alone

- ① Machines run (and thus OS must manage) multiple processes

how should the machine's resources be mapped to these processes?

- ① OS as a referee...



# You'll Never Walk Alone

- ☉ Machines run (and thus OS must manage) multiple processes

how should the machine's resources be mapped to these processes?

- ☉ Enter the illusionist!



give every process the illusion of running on a private CPU

Virtualize  
the CPU

which appears slower than the machine's

give every process the illusion of running on a private memory

Virtualize  
memory

which may appear larger (??) than the machine's

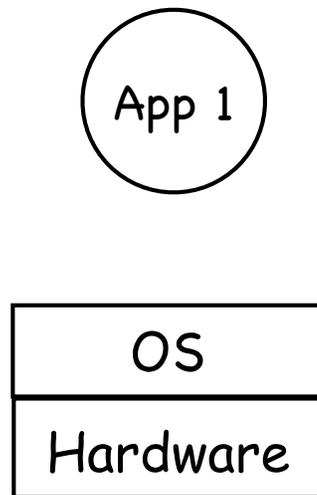
# Isolating Applications



Reading and writing memory,  
managing resources, accessing I/O...

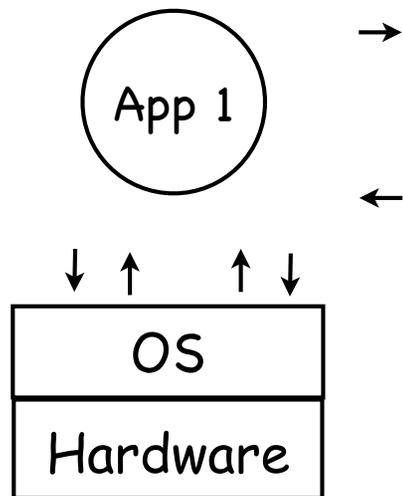
- 🕒 Buggy apps can crash other apps
- 🕒 Buggy apps can crash OS
- 🕒 Buggy apps can hog all resources
- 🕒 Malicious apps can violate privacy of other apps
- 🕒 Malicious apps can change the OS

# The Process, Refined



- ④ A running program with restricted rights
- ④ The enforcing mechanism must not hinder functionality
  - still efficient use of hardware
  - enable safe communication

# The Process, Refined



- ④ A running program with restricted rights
- ④ The enforcing mechanism must not hinder functionality
  - still efficient use of hardware
  - enable safe communication

# Mechanism and Policy

- ① Mechanism

  - enables a functionality

- ② Policy

  - determines how that functionality should be used

Mechanisms should not determine policies!

# Special

- ③ The process abstraction is enforced by the kernel

a part of the OS entrusted with special powers

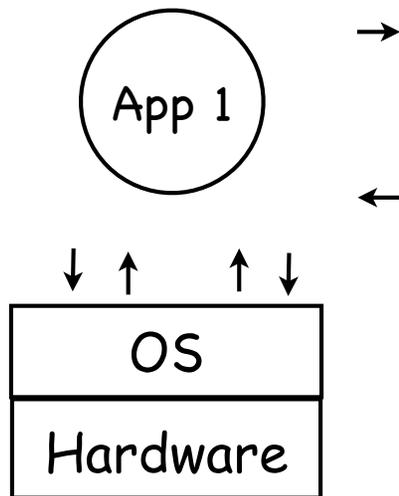
not all the OS is in the kernel

e.g., widgets libraries, window managers etc

why not? robustness

# How can the OS Enforce Restricted Rights?

- Easy: kernel interprets each instruction!



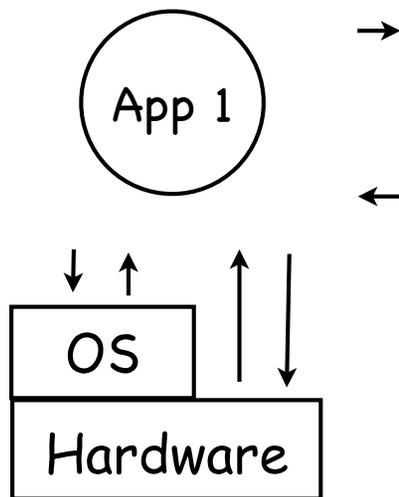
slow

many instructions are safe:  
do we really need to  
involve the OS?

# How can the OS Enforce Restricted Rights?

## Mechanism: Dual Mode Operation

hardware to the rescue: use a bit to enable two modes of execution:



→  
in user mode, processor only executes a limited (safe) set of instructions

←  
in kernel mode, no such restriction  
only OS kernel trusted to run in kernel mode

Think of Kernel as a “library with privileges”

# Amongst our weaponry are such diverse elements as...

## Privileged instructions

in user mode, no way to execute potentially unsafe instructions

## Memory isolation

in user mode, memory accesses outside a process' memory region are prohibited

## Timer interrupts

ensure kernel will periodically regain control from running process

# I. Privileged instructions

- ④ Set mode bit
- ④ I/O ops
- ④ Memory management ops
- ④ Disable interrupts
- ④ Set timers
- ④ Halt the processor

# I. Privileged instructions

☉ But how can an app do I/O then?

system calls achieve access to kernel mode  
only at specific locations specified by OS

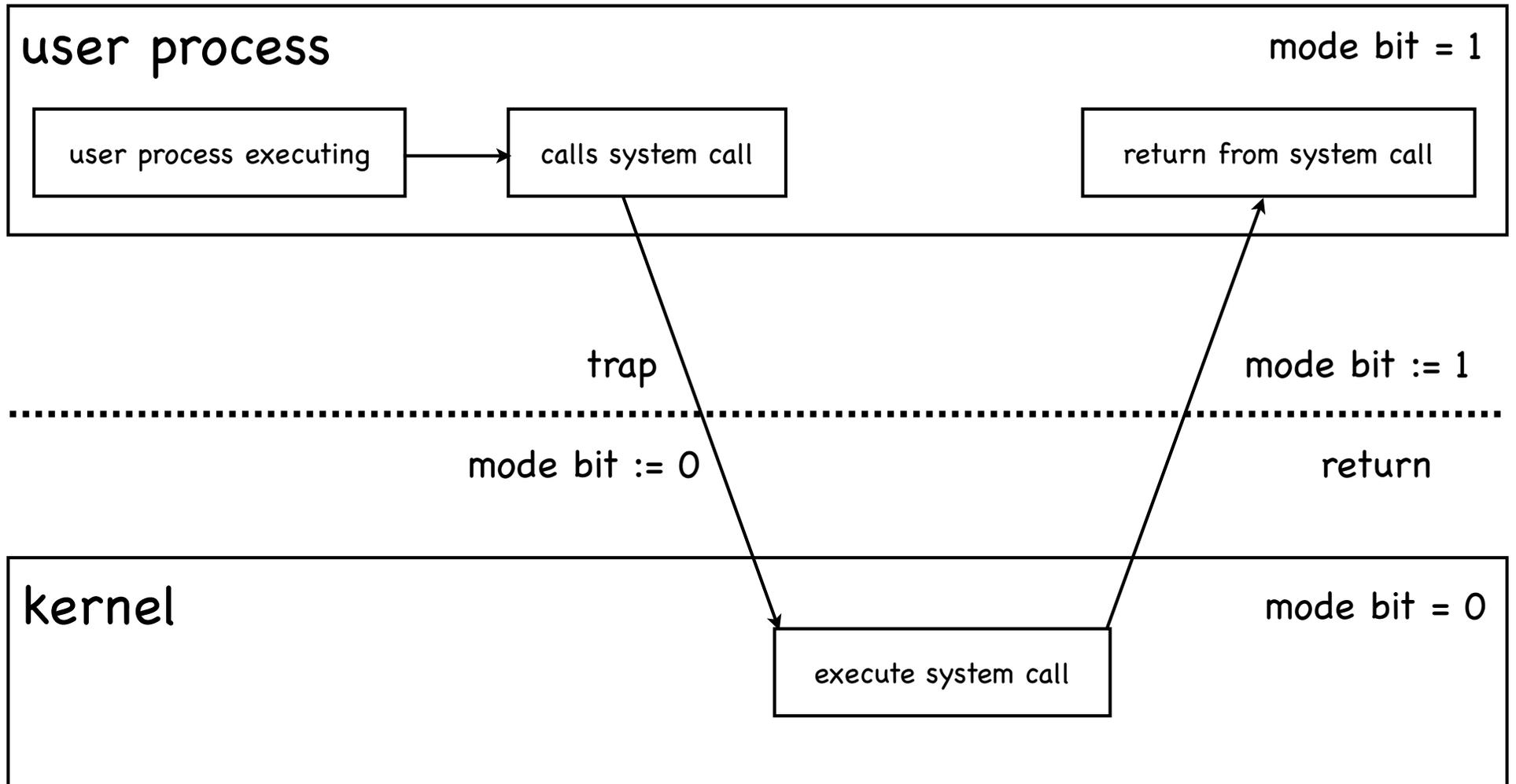
☉ Executing a privileged instruction while in  
user mode (naughty naughty...) causes a  
processor exception....

...which passes control to the kernel

# I. Privileged instructions

- ④ Set mode bit
- ④ I/O ops
- ④ Memory management ops
- ④ Disable interrupts
- ④ Set timers
- ④ Halt the processor
- ④ Set location of interrupt vector

# Crossing the line

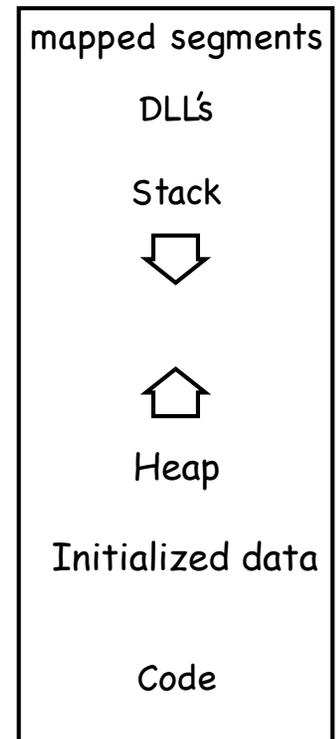


# II. Memory Protection

## Step 1: Virtualize Memory

- ⑥ Virtual address space: set of memory addresses that process can “touch”  
CPU works with virtual addresses
- ⑥ Physical address space: set of memory addresses supported by hardware

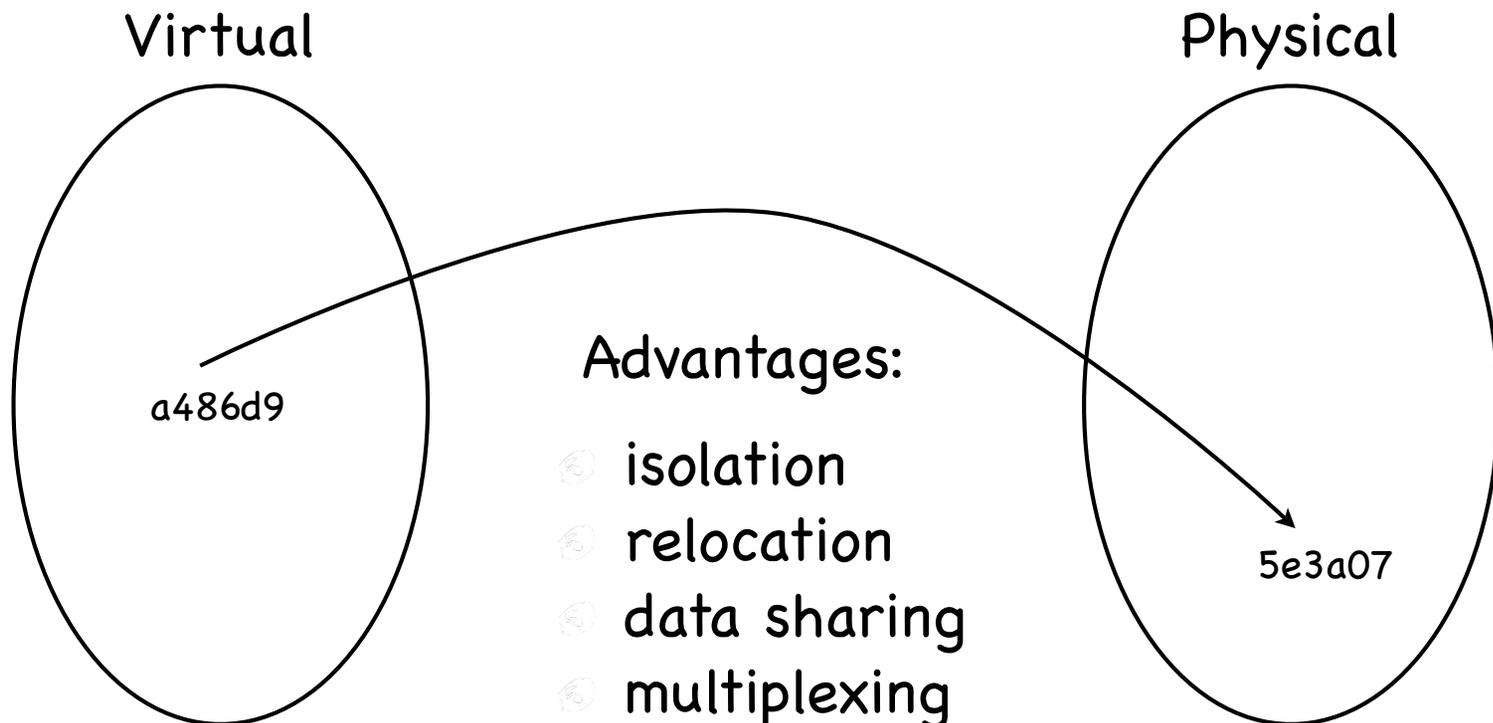
Virtual  
address  
space



# II. Memory Isolation

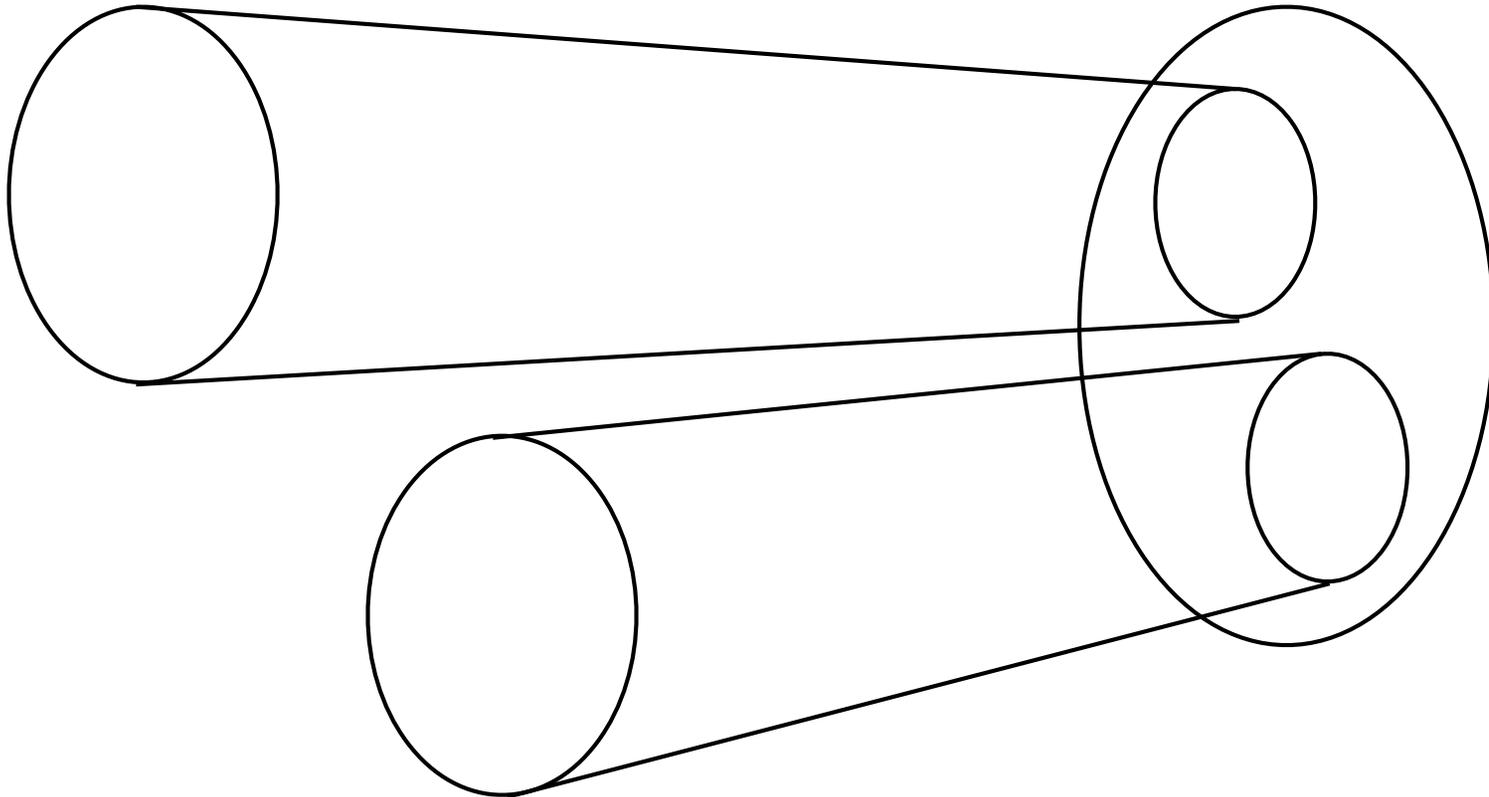
## Step 2: Address Translation

- 🕒 Implement a function mapping  
into



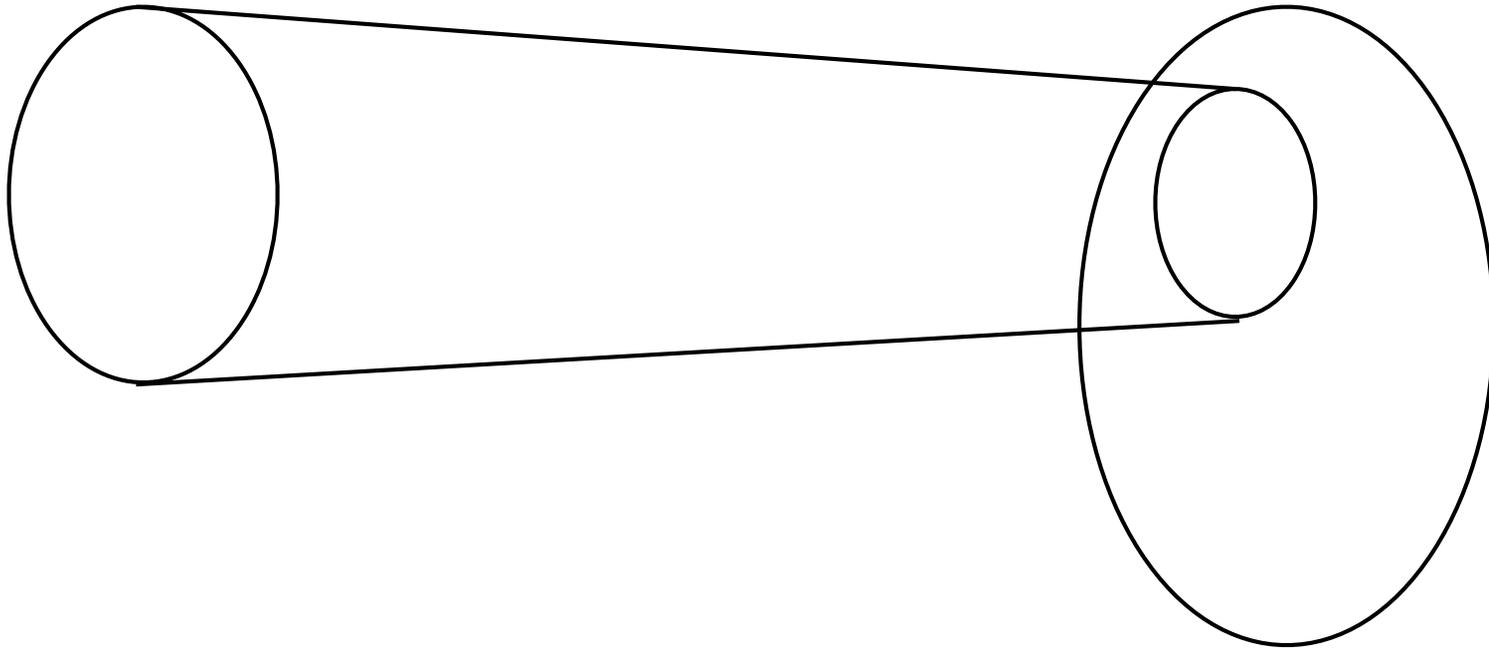
# Isolation

- At all times, functions used by different processes map to disjoint ranges – aka “Stay in your room!”



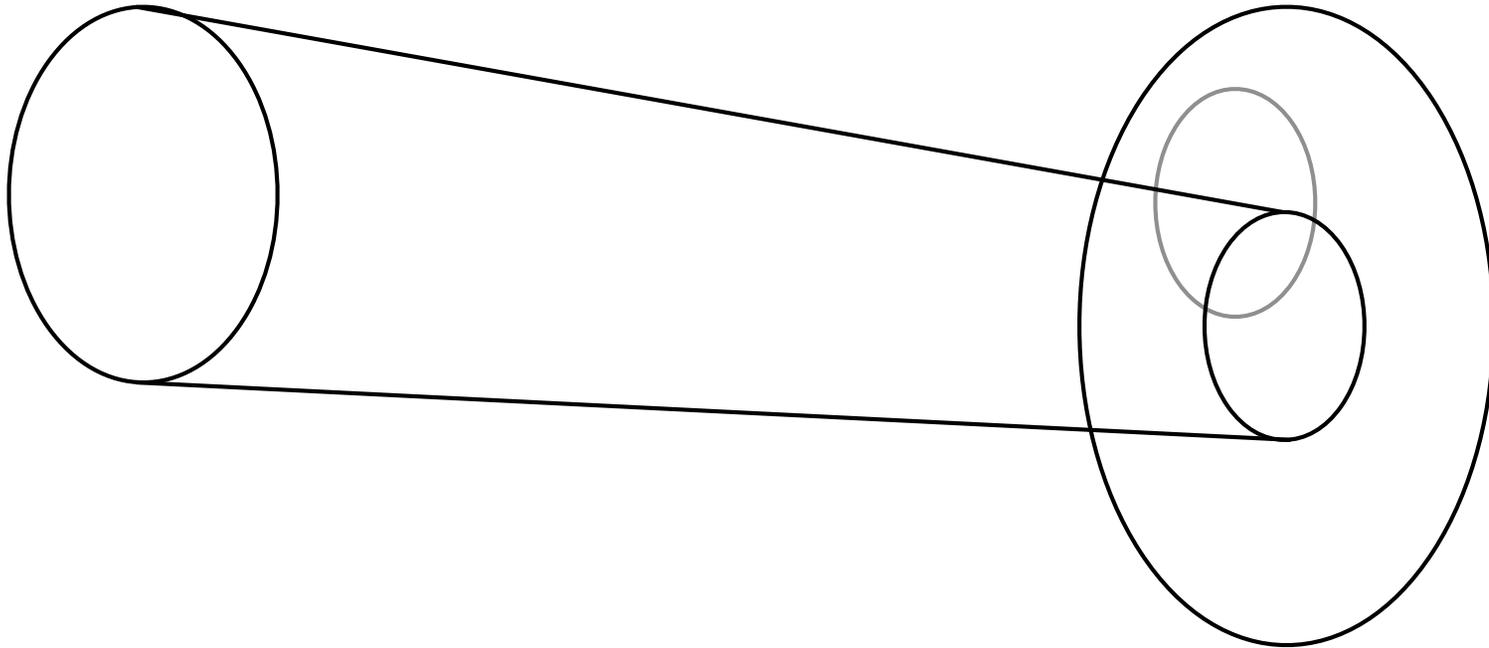
# Relocation

- ④ The range of the function used by a process can change over time



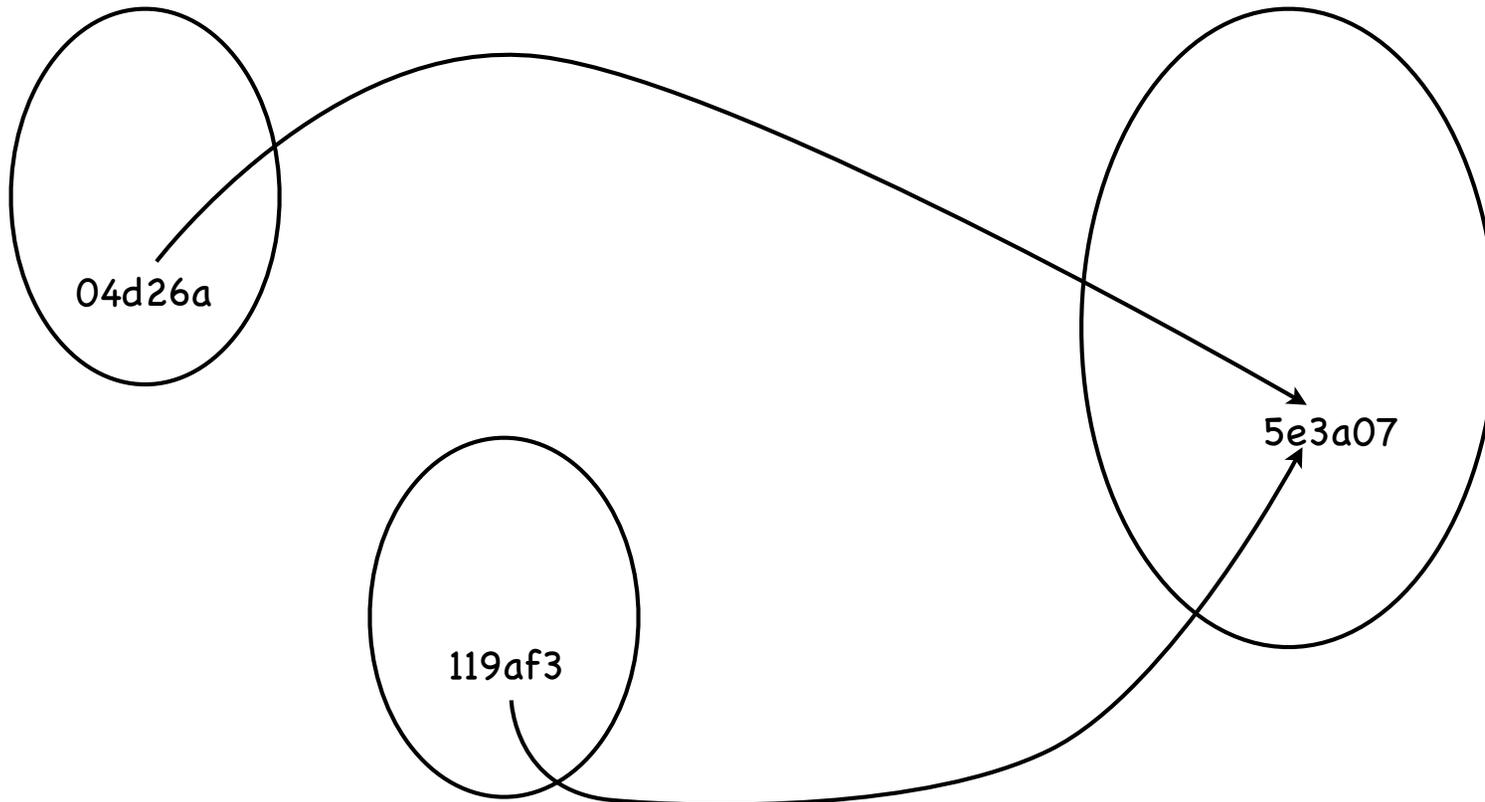
# Relocation

- ⑤ The range of the function used by a process can change over time – “Move to a new room!”



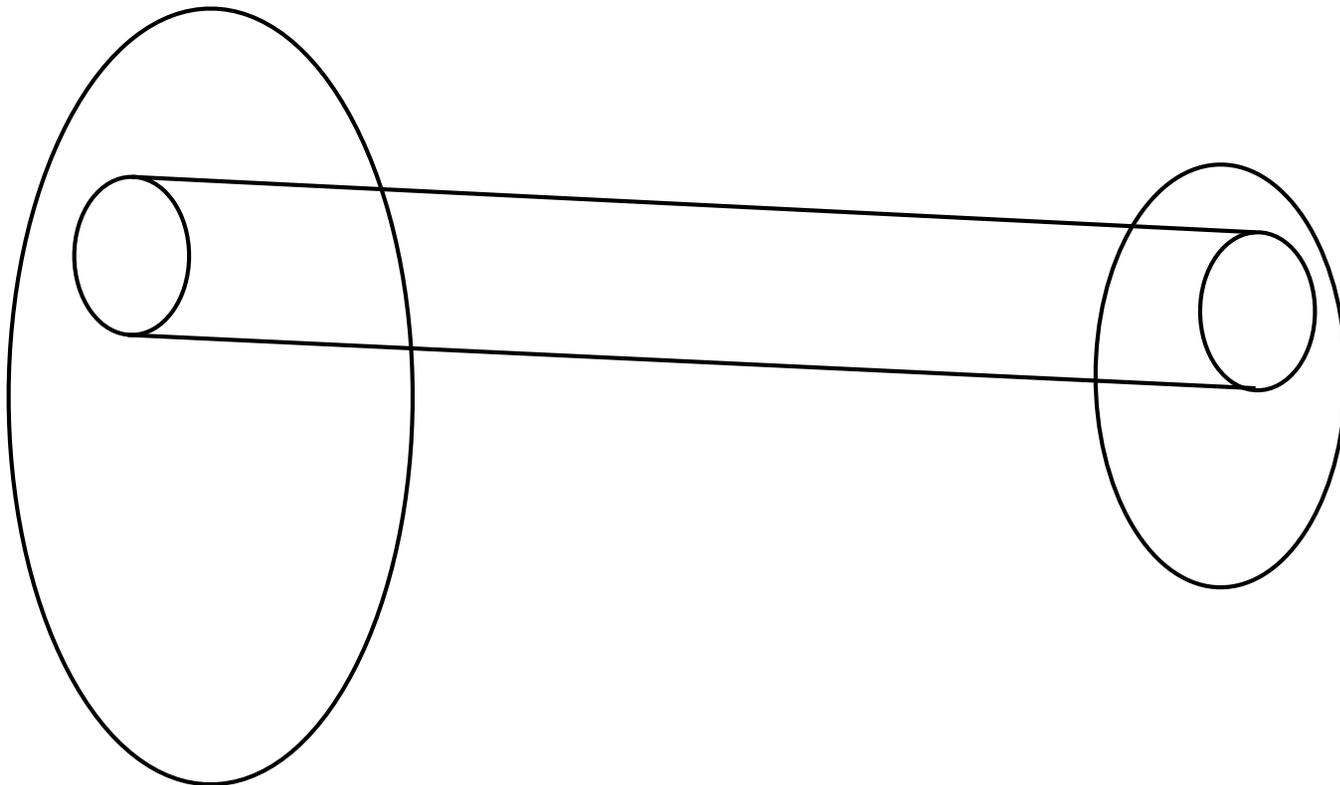
# Data Sharing

- Map different virtual addresses of distinct processes to the same physical address — “Share the kitchen!”



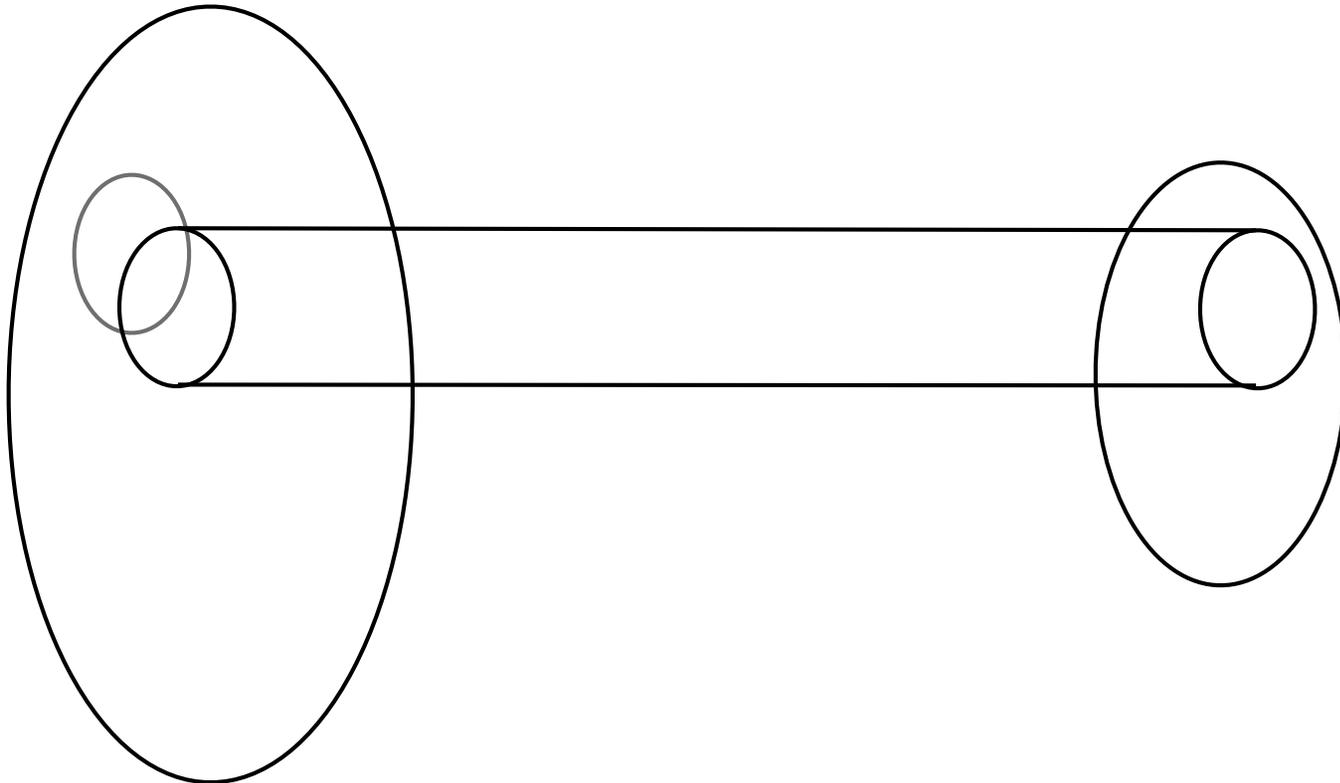
# Multiplexing

- ① Create illusion of almost infinite memory by changing domain (set of virtual addresses) that maps to a given range of physical addresses — ever lived in a studio?



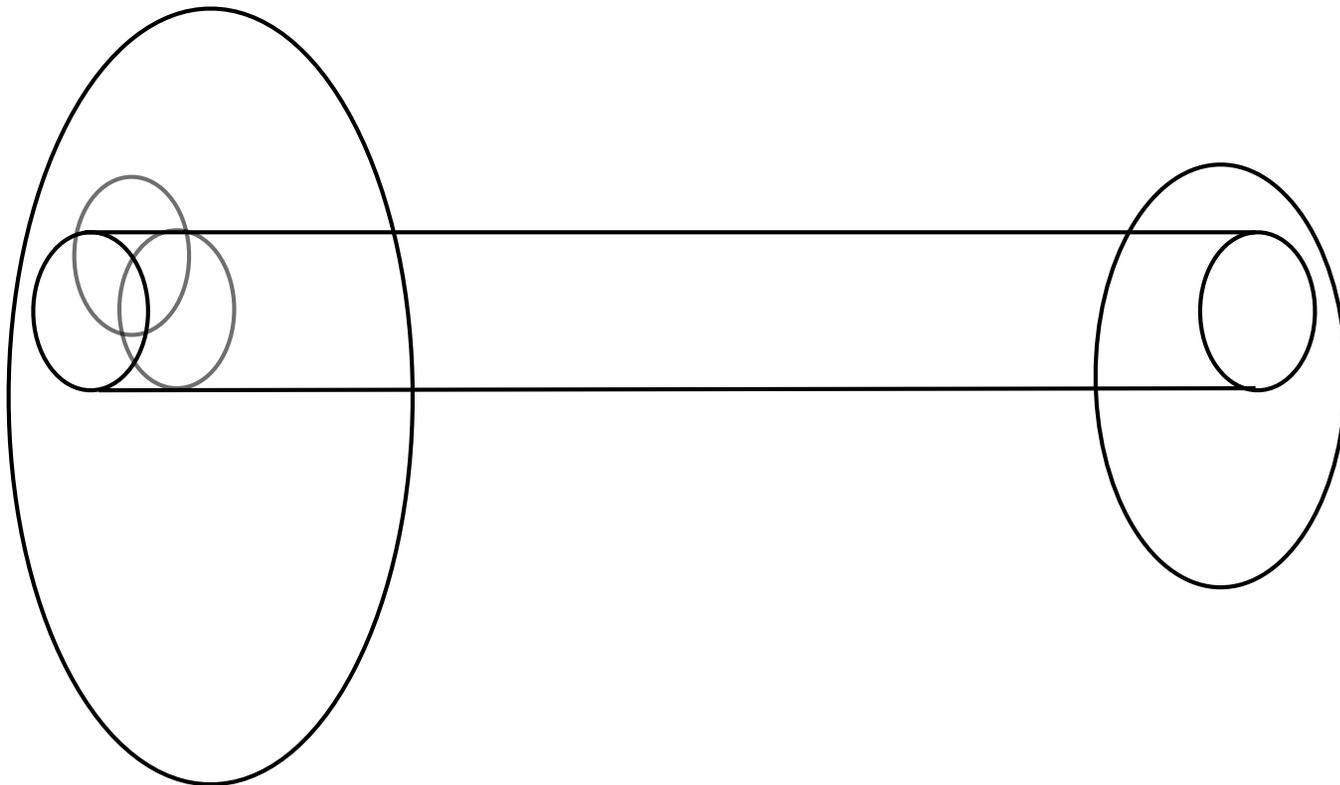
# Multiplexing

- ④ The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



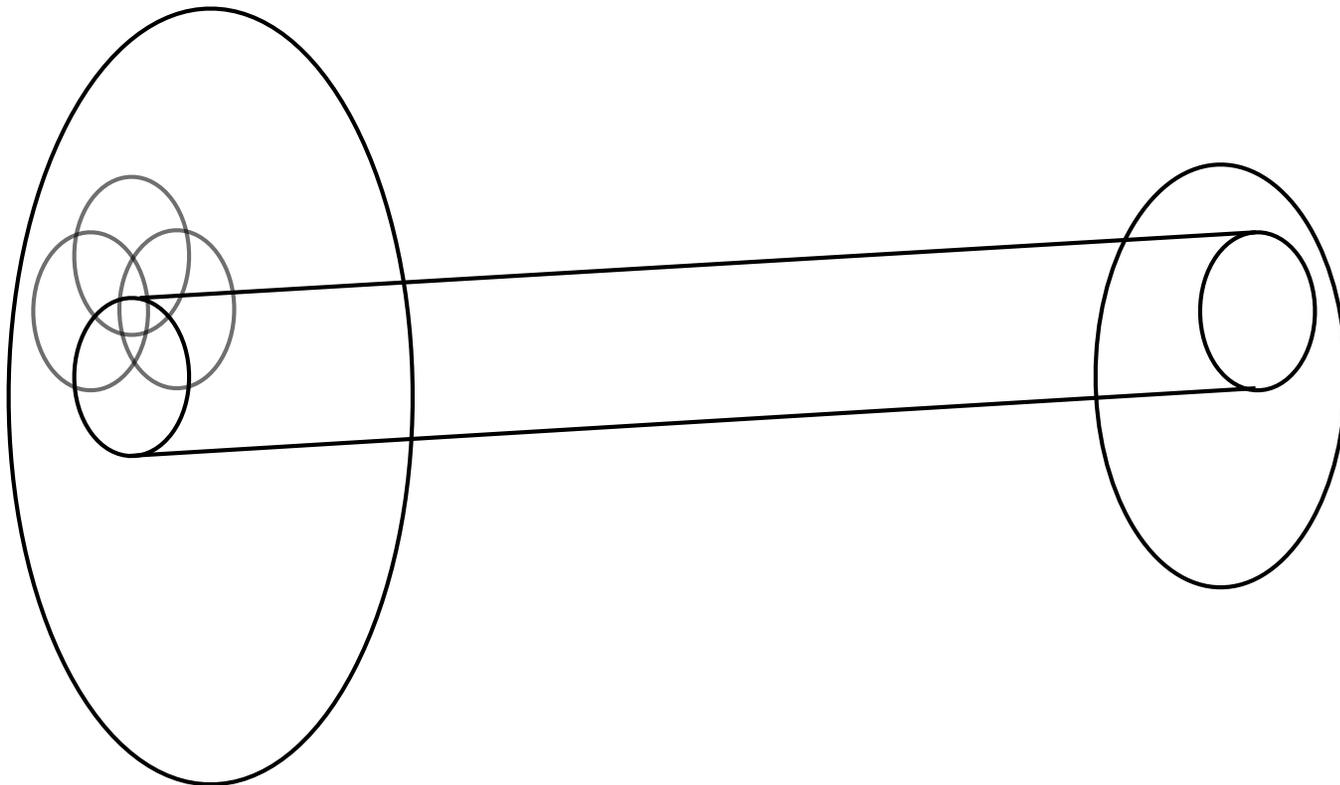
# Multiplexing

- ④ The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



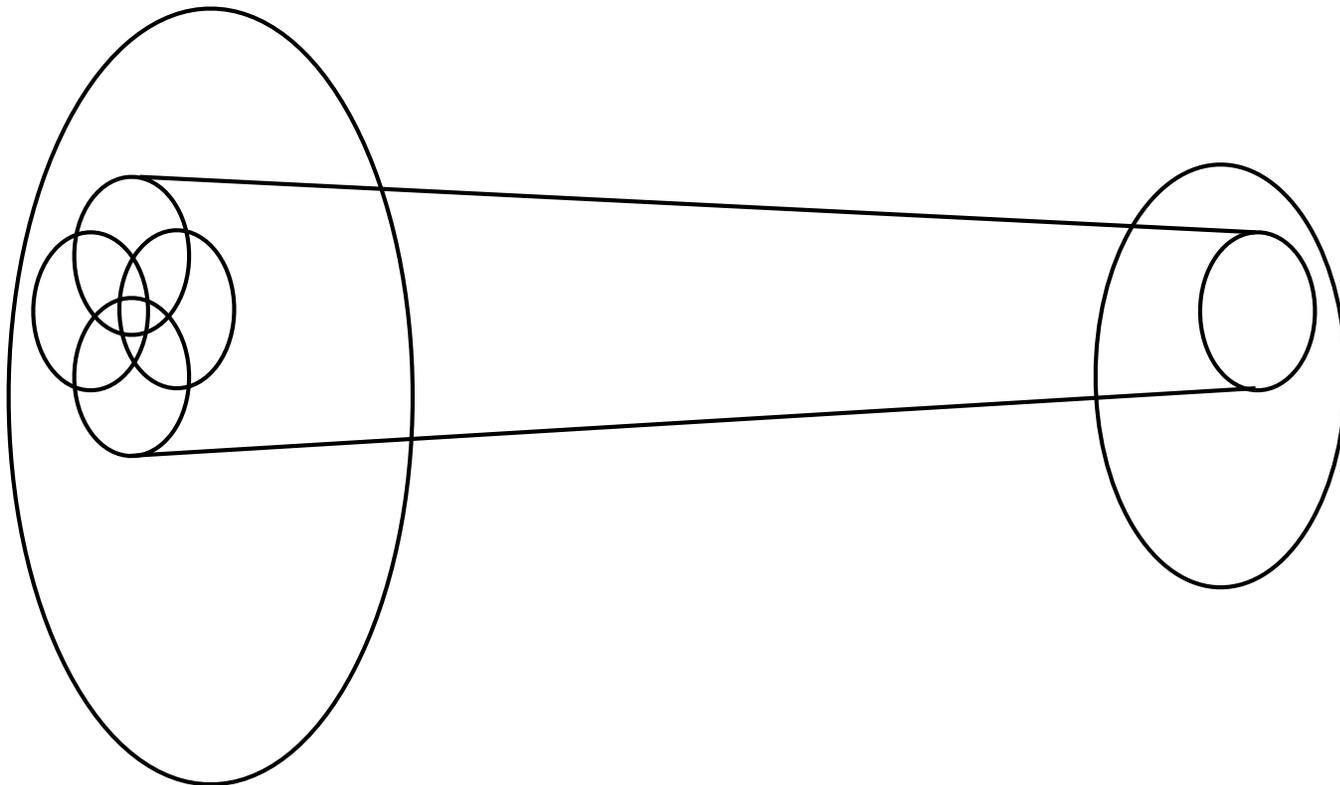
# Multiplexing

- ④ The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



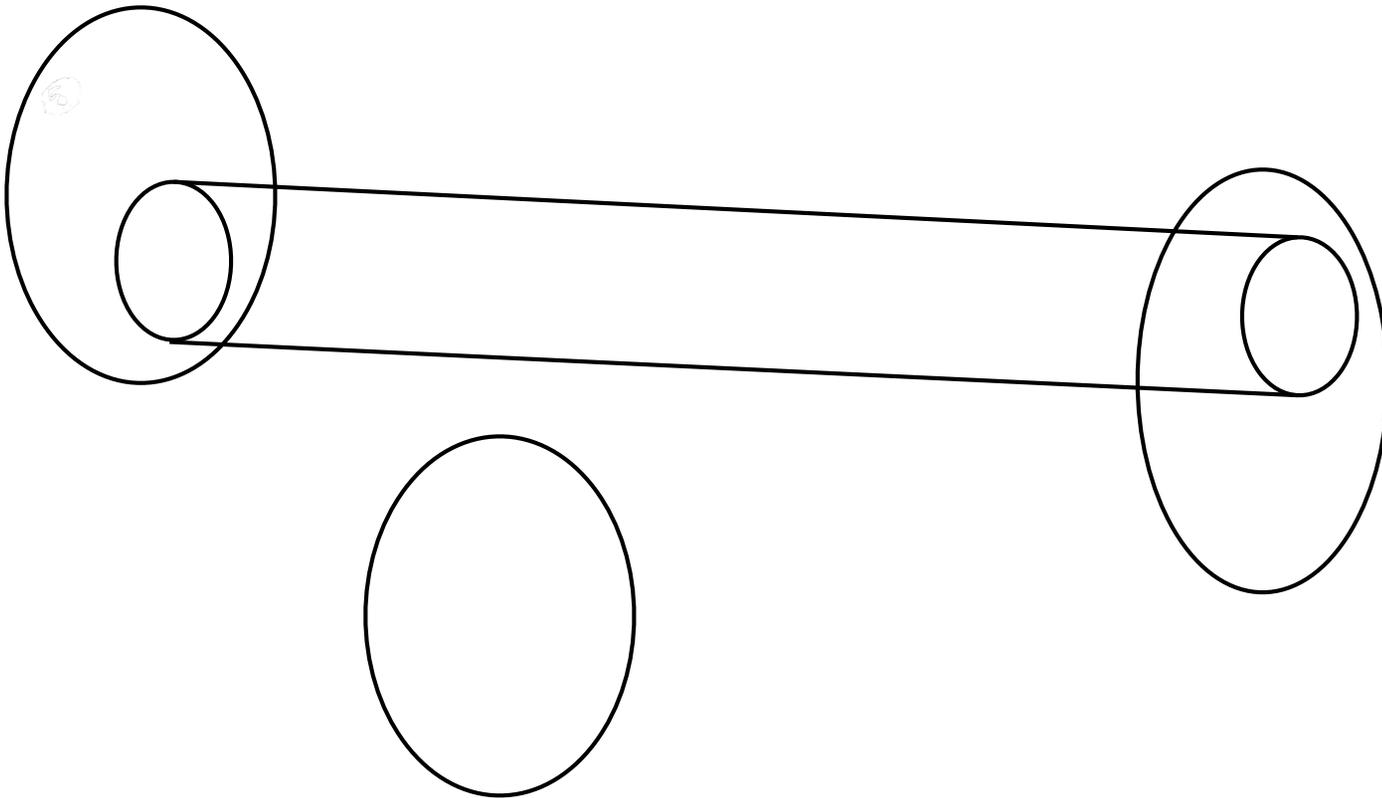
# Multiplexing

- ④ The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



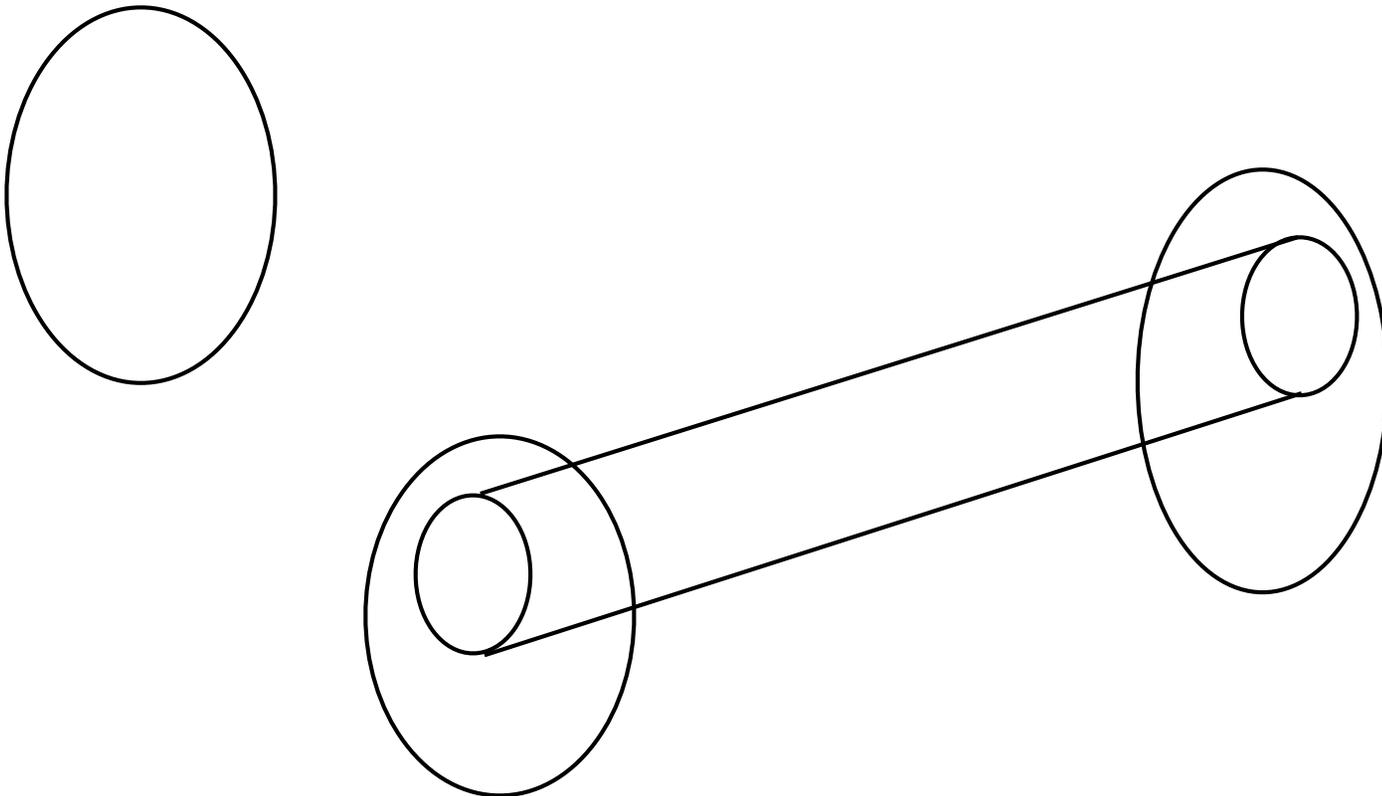
# More Multiplexing

- At different times, different processes can map part of their virtual address space into the same physical memory – change tenants!



# More Multiplexing

- ⑤ At different times, different processes can map part of their virtual address space into the same physical memory – change tenants!





# On Base & Limit

- ① Contiguous Allocation: contiguous virtual addresses are mapped to contiguous physical addresses
- ① Isolation is easy, but sharing is hard
  - ① Say I have two copies of Emacs: want to share code, but have heap and stack distinct...
- ① And there is more...
  - Hard to relocate
  - ① Hard to account for dynamic changes in both heap and stack

# III. Timer Interrupts

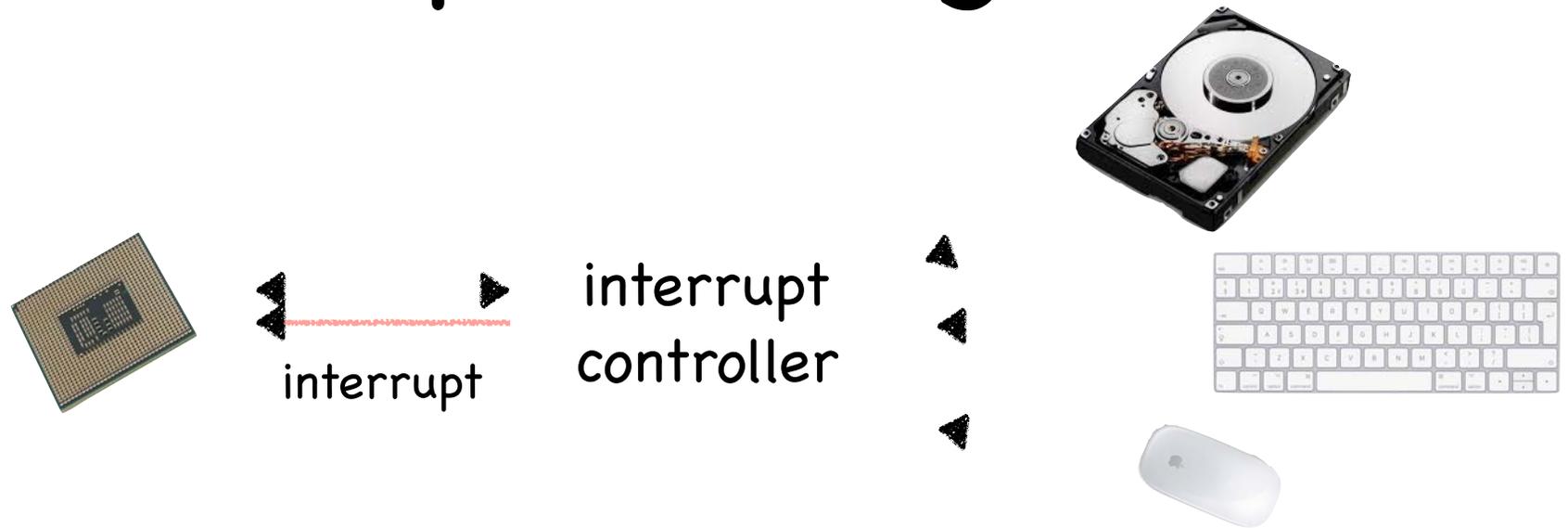
- ① Hardware timer

  - can be set to expire after specified delay (time or instructions)

  - when it does, control is passed back to the kernel

- ② Other interrupts (e.g. I/O completion) also give control to kernel

# Interrupt Management



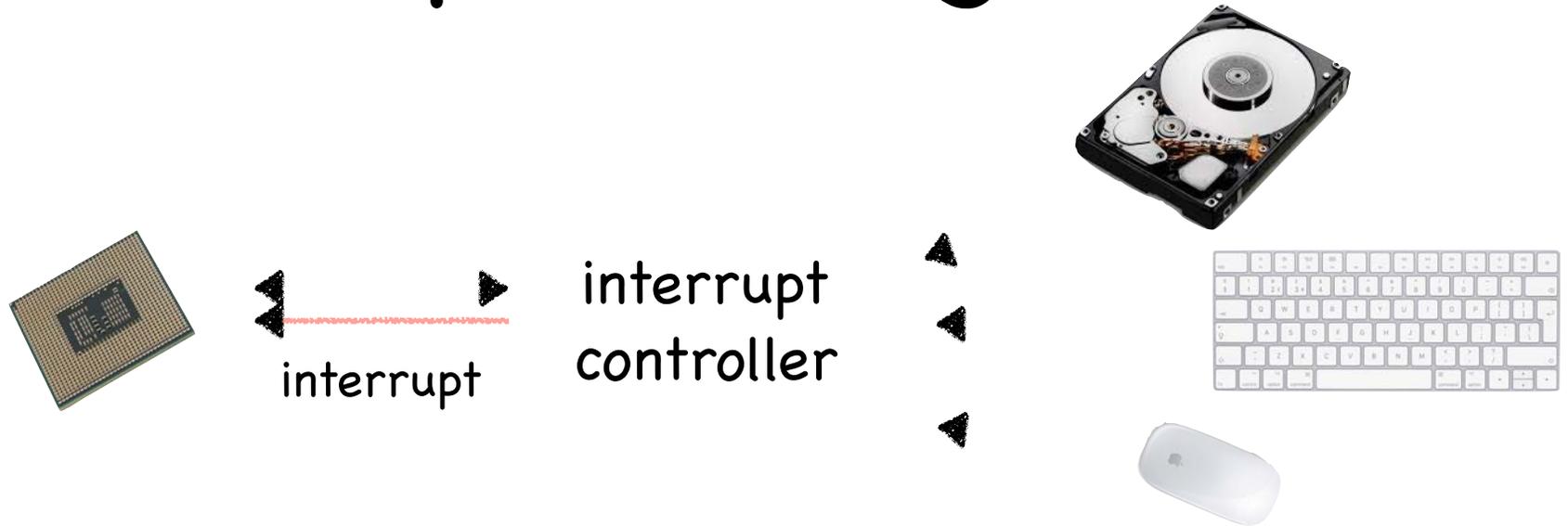
Interrupt controllers implements interrupt priorities:

Interrupts include descriptor of interrupting device

Priority selector circuit examines all interrupting devices, reports highest level to the CPU

Controller can also buffer interrupts coming from different devices

# Interrupt Management



## Maskable interrupts

can be turned off by the CPU for critical processing

## Nonmaskable interrupts

indicate serious errors (power out warning, unrecoverable memory error, etc.)

# Types of Interrupts

## Exceptions

- process missteps (e.g. division by zero)
- attempt to perform a privileged instruction  
sometimes on purpose! (breakpoints)
- synchronous/non-maskable

## Interrupts

- HW device requires OS service  
timer, I/O device, interprocessor
- asynchronous/maskable

## System calls/traps

- user program requests OS service
- synchronous/non-maskable