The Process

A running program

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

- Create
  - causes the OS to create a new process
- Destroy
  - forcefully terminates a process
- Wait (for the process to end)
- Other controls
  - e.g. to suspend or resume the process
- Status
  - running? suspended? blocked? for how long?

How the OS Keeps Track of a Process

- A process has code
  - OS must track program counter
- A process has a stack
  - OS must track stack pointer
- OS stores state of process in Process Control Block (PCB)
  - Data (program instructions, stack & heap) resides in memory, metadata is in PCB
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?
- OS as a referee...

Enter the illusionist!
- give every process the illusion of running on a private CPU
  - which appears slower than the machine’s
- give every process the illusion of running on a private memory
  - which may appear larger(??) than the machine’s

Isolating Applications

- 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

Operating System

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

Mechanism and Policy

- Mechanism
  - what the system can do
- Policy
  - what the system should do

Mechanisms should not determine policies!
The Process, Refined

An abstraction for isolation
the execution of an application program with restricted rights

The enforcing mechanism must not hinder functionality
still efficient use of hardware
enable safe communication

Special

The process abstraction is enforced by the kernel

- all kernel is in the OS
- not all the OS is in the kernel
  - (why not? robustness)
  - widgets libraries, window managers etc

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 mode bit
  - in user mode, processor checks every instruction
  - in kernel mode, unrestricted rights
- hardware to the rescue (again) to make checks efficient

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**
  - kernel must be able to periodically regain control from running process

I. Privileged instructions

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

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
Crossing the line

II. Memory Protection

Step 1: Virtualize Memory
- **Virtual address space**: set of memory addresses that process can “touch”
  - CPU works with virtual addresses
- **Physical address space**: set of memory addresses supported by hardware

II. Memory Isolation

Step 2: Address Translation
- Implement a function mapping \( \langle \text{pid}, \text{virtual address} \rangle \) into physical address

Advantages:
- isolation
- relocation
- data sharing
- multiplexing

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

- 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

- 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

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!

A simple mapping mechanism: Base & Bound

On Base & Limit

Contiguous Allocation: contiguous virtual addresses are mapped to contiguous physical addresses

Isolation is easy, but sharing is hard

Two copies of emacs: want to share code, but have heap and stack distinct...

And there is more...

- Hard to relocate
- Hard to account for dynamic changes in both heap and stack