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

- 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...
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
    - 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...

App 1

Malicious apps can change the OS
The Process, Refined

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

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

- Mechanism
  enables a functionality

- Policy
  determines how that functionality should be used

Mechanisms should not determine policies!
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!

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

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

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

user process

user process executing ↔ calls system call

trap

execute system call

return from system call

kernel

execute system call

mode bit = 1

mode bit := 0

mode bit := 1

return

mode bit = 0
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 into

Virtual

Advantages:
- isolation
- relocation
- data sharing
- multiplexing

Physical

a486d9

5e3a07
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
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.
Multiplexing

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

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

CPU

Logical addresses

Memory Exception

≤

no

yes

≤

500

Bound Register

1000

Base Register

≤

1500

MAXsys

p's physical address space

≤

1000

0
Contiguous Allocation: contiguous virtual addresses are mapped to contiguous physical addresses

Isolation is easy, but sharing is hard

Say I have 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
III. Timer Interrupts

- Hardware timer
can be set to expire after specified delay
  (time or instructions)
  when it does, control is passed back to the
  kernel

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

Interrupt controllers implement 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
  sometime 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