Now that we have processes... 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?

So, where are we?

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

OS must be able to isolate apps from one another

So, where are we?

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

OS must be able to isolate itself from other processes!

- 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

Fine.

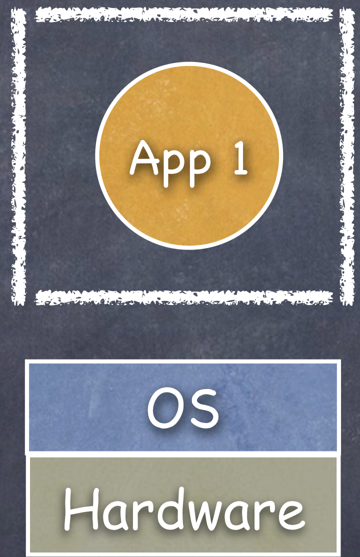
But now that we have successfully isolated each process from everything, how do they get anything done?

I/O?

R & W memory?

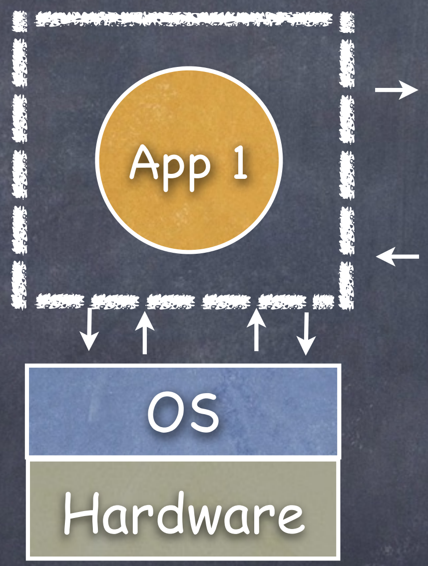
Cooperate/communicate with each other?

The Process, Refined



- A running program with restricted rights
  - trust program with performing harmless, local actions.
  - for the rest, "adult supervision"!
- The mechanism that enforces the restriction must not hinder functionality
  - still efficient use of hardware
  - enable safe communication

The Process, Refined



- A running program with restricted rights
  - trust program with performing harmless, local actions.
  - for the rest, "adult supervision"!
- The mechanism that enforces the restriction must not hinder functionality
  - still efficient use of hardware
  - enable safe communication

Quick aside: Mechanism vs Policy

- Mechanism
  - enables a functionality
- Policy
  - determines how that functionality will be used

Mechanisms should not determine policies!

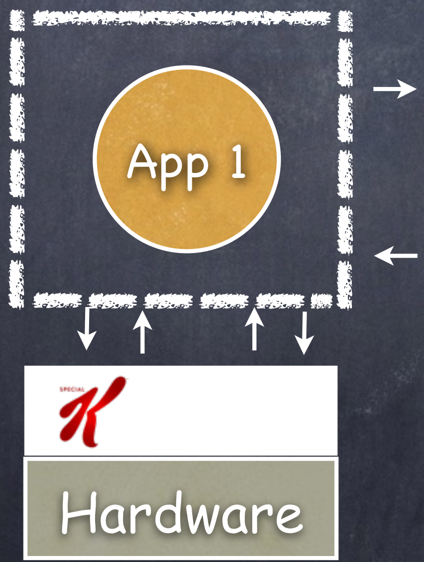
Enters the OS Kernel

- A subset of the OS charged with special rights and responsibilities
- Kernel is trusted with full access to all hardware capability
- All other software (OS or applications) is untrusted

Untrusted	Applications
	Rest of the OS
Trusted	Kernel

How can the OS Enforce Restricted Rights?

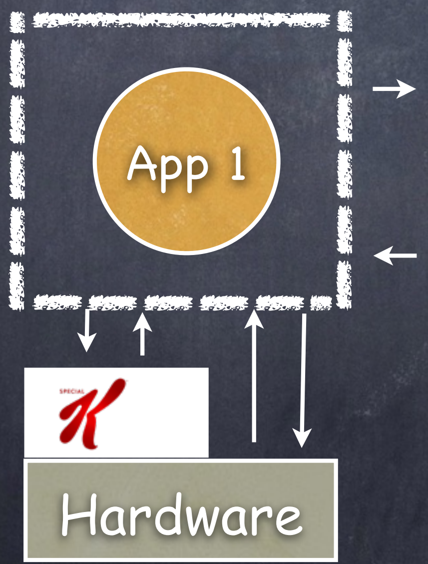
- Easy: kernel interprets and checks each instruction from apps (and untrusted OS)



  - slow
  - many instructions are safe: do we really need to involve the kernel?

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 (checked by processor)
  - in kernel mode, no such restriction
- only OS kernel trusted to run in kernel mode

Amongst our weaponry are such diverse elements as...

- To support dual-mode operation:
  - Privileged instructions
    - in user mode, no way to execute potentially unsafe instructions. HW checks each instruction: if privileged, control is passed to the kernel.
  - 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

*there's more of them!!

I. Privileged instructions

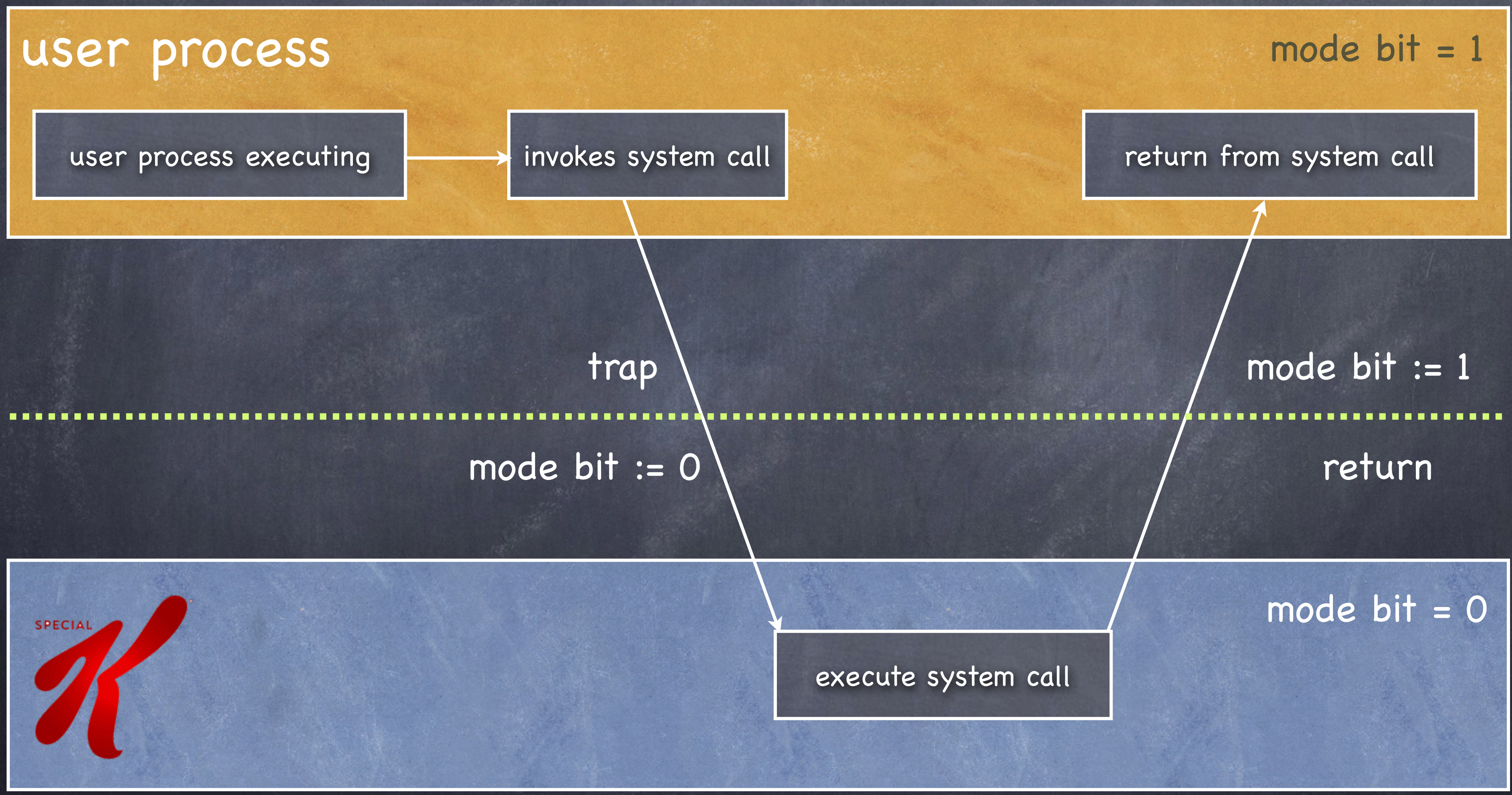
- Set mode bit
- I/O ops
- Memory management ops
- Disable interrupts
- Set timers
- Halt the processor

I. Privileged instructions

- But how can an app do I/O then?
  - it can politely ask the kernel to perform it on its behalf
    - system calls cause the processor to transition from user to kernel mode, from which they execute code specified by the OS (kernel code) and stored at specific memory locations that depend on the system call

Pretty please?

Crossing the line



I. Privileged instructions

- But how can an app do I/O then?
  - it can politely ask the kernel to perform it on its behalf via a system call
  - it can force the issue by executing a privileged instruction while in user mode (naughty naughty...)
  - This causes a processor exception....
  - ...which abruptly passes control to the kernel at specific locations (exception dependent) where appropriate handlers are invoked
    - these locations are specified in a so-called interrupt vector

More aboout this coming up!

I. Privileged instructions

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

Supporting Dual-Mode Operation

- Privileged Instructions
- Memory Isolation
- Timer* Interrupts

Questions?

Supporting Dual-Mode Operation

- Privileged Instructions
- Memory Isolation
- Timer* Interrupts

II. Memory Isolation

Step 1: Virtualize Memory

- Physical address space: set of memory addresses supported by hardware
- Virtual address space: set of memory addresses that process can name
  - CPU works with virtual addresses
  - Kernel is typically mapped in the Virtual address space of every process
  - but that portion of the address space can only be accessed in kernel mode