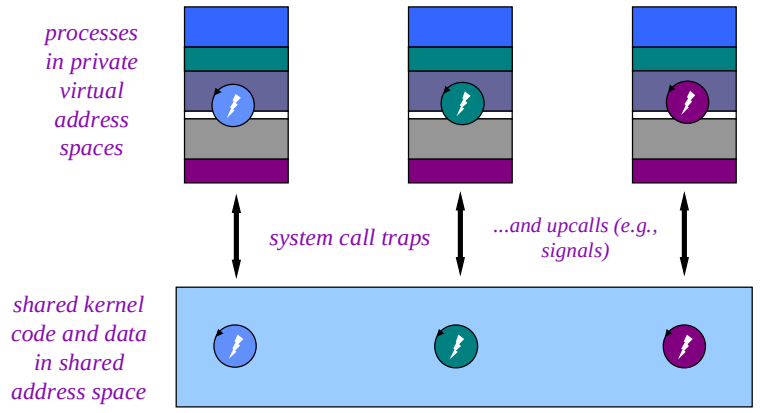
Protection and the Kernel:
Mode, Space, and Context

Challenges for a "Classical" Protected OS

What are the mechanisms that operating systems use to:

- safely allocate physical resources (memory, storage, CPU) to multiple programs?
- securely track and control resource usage?
- isolate programs from the details of running on a shared machine?
- protect one executing program's storage from another?
- prevent rogue programs from taking over the machine or impairing the functions of the operating system?
- allow mutually distrusting programs to interact safely?
- recover cleanly from user program failures

Processes and the Kernel



CPU and devices force entry to the kernel to handle exceptional events.

A First Look at Some Key Concepts

kernel

The software component that controls the hardware directly, and implements the core privileged OS functions.

Modern hardware has features that allow the OS kernel to protect itself from untrusted user code.

thread

An executing stream of instructions and its CPU register context.

virtual address space

An execution context for thread(s) that provides an independent name space for addressing some or all of physical memory.

process

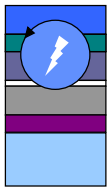
An execution of a program, consisting of a virtual address space, one or more threads, and some OS kernel state.

The Kernel

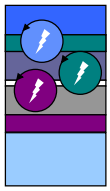
- Today, all "real" operating systems have protected kernels.
  - The kernel resides in a well-known file: the "machine" automatically loads it into memory (*boots*) on power-on/reset.
  - Our "kernel" is called the *executive* in some systems (e.g., NT).
- The kernel is (mostly) a library of service procedures shared by all user programs, *but the kernel is* ***protected***:
  - User code cannot access internal kernel data structures directly, and it can invoke the the kernel only at well-defined entry points (*system calls*).
- *Kernel code is like user code, but the kernel is* ***privileged***:
  - The kernel has direct access to all hardware functions, and defines the machine entry points for *interrupts* and *exceptions*.

Threads vs. Processes

1. The *process* is a *kernel abstraction* for an independent executing program.
   - includes at least one "thread of control"
   - also includes a private address space (VAS)
     - VAS requires OS kernel support
   - often the unit of resource ownership in kernel
     - e.g., memory, open files, CPU usage



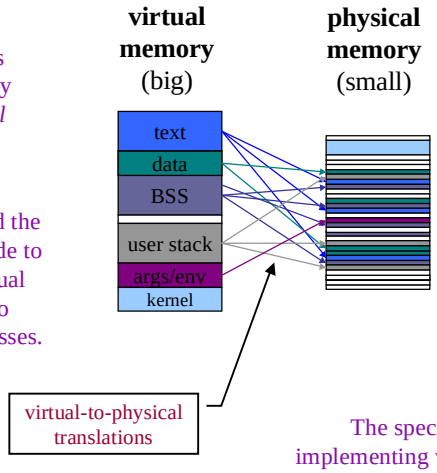
2. Threads may share an address space.
   - Threads have "context" just like vanilla processes.
     - *thread context switch* vs. *process context switch*
   - Every thread must exist within some process VAS.
   - Processes may be "multithreaded" with thread primitives supported by a library or the kernel.



Introduction to Virtual Addressing

User processes address memory through *virtual addresses*.

The kernel and the machine collude to translate virtual addresses to physical addresses.

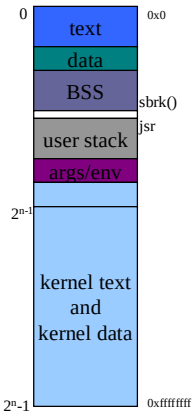


The kernel controls the virtual-physical translations in effect for each space.

The machine does not allow a user process to access memory unless the kernel "says it's OK".

The specific mechanisms for implementing virtual address translation are *machine-dependent*: we will cover them later.

The Virtual Address Space

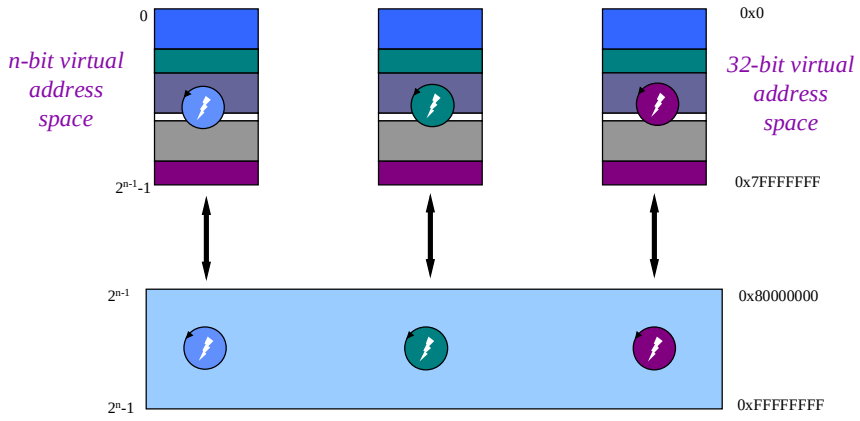


A *typical* process VAS space includes:

- user regions in the lower half
  - V->P mappings specific to each process
  - accessible to user or kernel code
- kernel regions in upper half
  - shared by all processes
  - accessible only to kernel code
- **Nachos:** process virtual address space includes only user portions.
  - mappings change on each process switch

A VAS for a private address space system (e.g., Unix) executing on a typical 32-bit architecture.

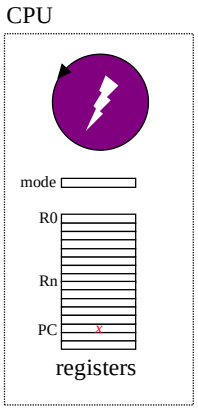
Example: Process and Kernel Address Spaces

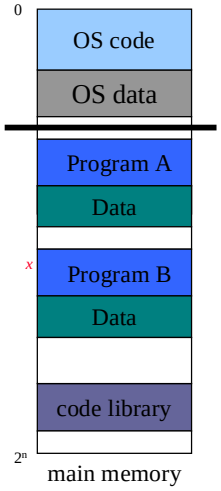


Kernel Mode

CPU *mode* (a field in some status register) indicates whether the CPU is running in a *user* program or in the protected *kernel*.

Some instructions or data accesses are only legal when the CPU is executing in kernel mode.





physical address space

Protecting Entry to the Kernel

Protected events *and* ***kernel mode*** *are the architectural foundations of kernel-based OS (Unix, NT, etc).*

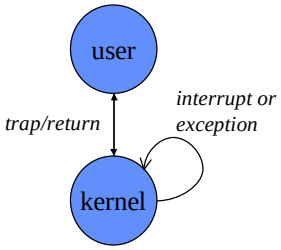
- The *machine* defines a small set of exceptional event types.
- The *machine* defines what conditions raise each event.
- The kernel installs handlers for each event at boot time.

  e.g., a table in kernel memory read by the machine

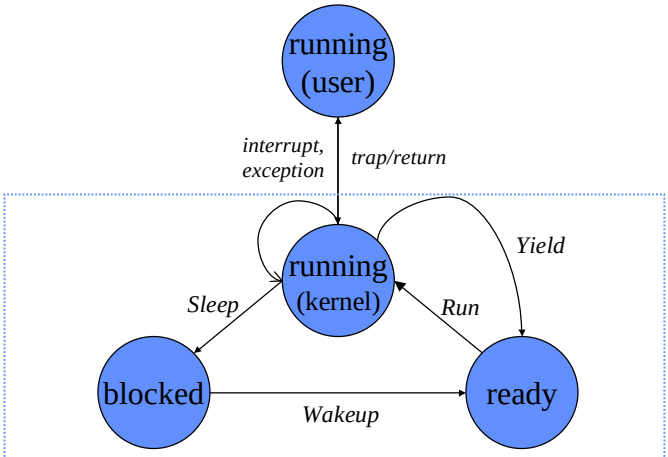
The machine transitions to kernel mode only on an exceptional event.

The kernel defines the event handlers.

Therefore the *kernel* chooses what code will execute in kernel mode, and when.



Thread/Process States and Transitions



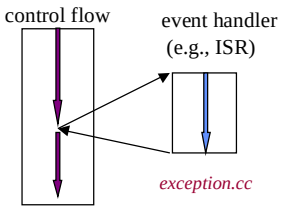
CPU Events: Interrupts and Exceptions

- an "unnatural" change in control flow
- an *interrupt* is caused by an external event

  device requests attention, timer expires, etc.
- an *exception* is caused by an executing instruction

  CPU requires software intervention to handle a *fault* or *trap*.
- the kernel defines a handler routine for each event type

	unplanned	deliberate
sync	*fault*	*syscall trap*
async	*interrupt*	AST



Handling Events, Part I: The Big Picture

1. To deliver the event, the machine saves relevant state in temporary storage, then transfers control to the kernel.

   Set kernel mode and set PC := *handler*.
2. Kernel handler examines registers and saved machine state.

   What happened? What was the machine doing when it happened? How should the kernel respond?
3. Kernel responds to the condition.

   Execute kernel service, device control code, fault handlers, etc., modify machine state as needed.
4. Kernel restores saved context (registers) and resumes activity.
5. Specific events and mechanisms for saving, examining, or restoring context are *machine-dependent*.

Mode, Space, and Context

At any time, the state of each processor is defined by:

1. *mode*: given by the mode bit

   Is the CPU executing in the protected kernel or a user program?

2. *space*: defined by V->P translations currently in effect

   What address space is the CPU running in? Once the system is booted, it always runs in some virtual address space.

3. *context*: given by register state and execution stream

   Is the CPU executing a thread/process, or an interrupt handler?

   Where is the stack?

These are important because the mode/space/context determines the meaning and validity of key operations.

Common Mode/Space/Context Combinations

1. *User code* executes in a process/thread context in a process address space, in user mode.

   Can address only user code/data defined for the process, with no access to privileged instructions.

2. *System services* execute in a process/thread context in a process address space, in kernel mode.

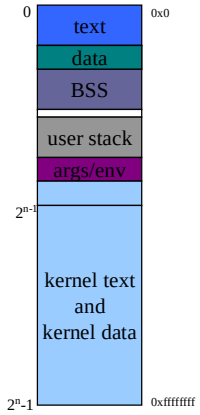
   Can address kernel memory or user process code/data, with access to protected operations: may sleep in the kernel.

3. *Interrupts* execute in a system interrupt context in the address space of the interrupted process, in kernel mode.

   Can access kernel memory and use protected operations.

   no sleeping!

Summary: Mode, Space, and Context



	Process context	System context
User mode	Application	N/A
Kernel mode	Syscall or exception	Interrupt or System task