# Machines and Virtualization

Systems and Networks

Jeff Chase

Spring 2006

# Memory Protection

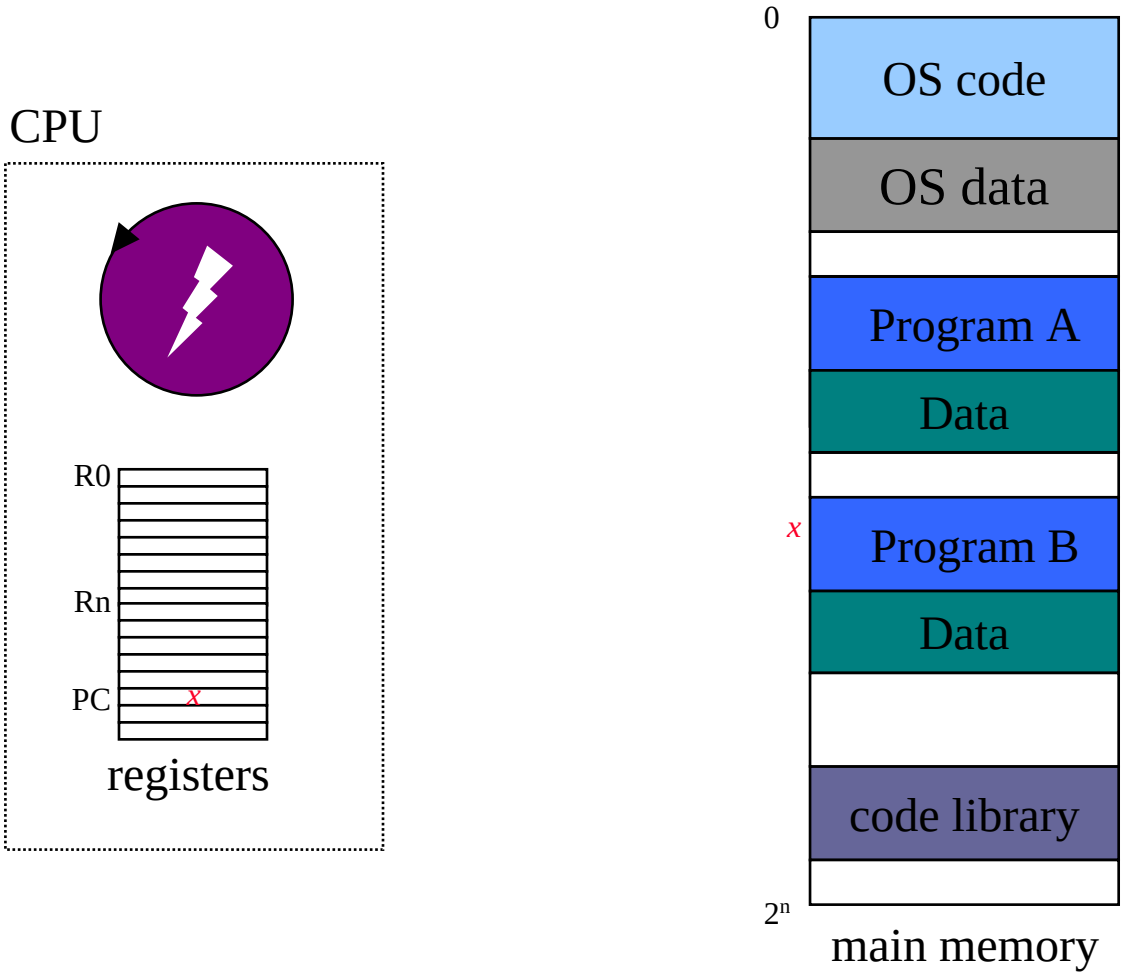
Paging Virtual memory provides protection by:

- Each process (user or OS) has different virtual memory space.
- The OS maintain the page tables for all processes.
- A reference outside the process allocated space cause an exception that lets the OS decide what to do.
- Memory sharing between processes is done via different Virtual spaces but common physical frames.

# Architectural Foundations of OS Kernels

- One or more privileged execution modes (e.g., *kernel mode*)
  - protected device control registers
  - privileged instructions to control basic machine functions
- System call *trap* instruction and protected fault handling
  - User processes safely enter the kernel to access shared OS services.
- Virtual memory mapping
  - OS controls virtual-physical translations for each address space.
- Device interrupts to notify the kernel of I/O completion etc.
  - Includes timer hardware and clock interrupts to periodically return control to the kernel as user code executes.
- Atomic instructions for coordination on multiprocessors

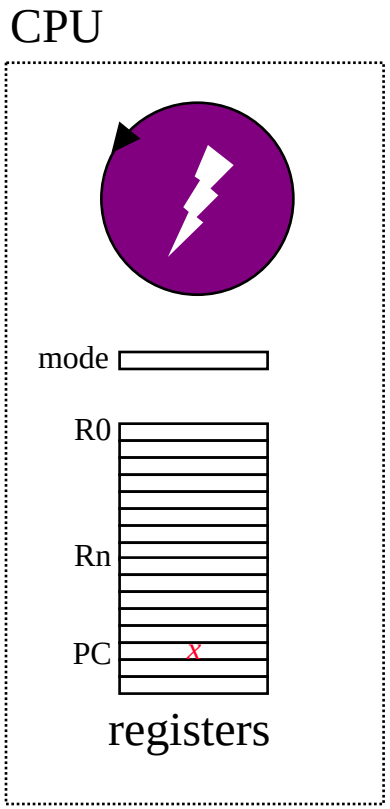
# Memory and the CPU

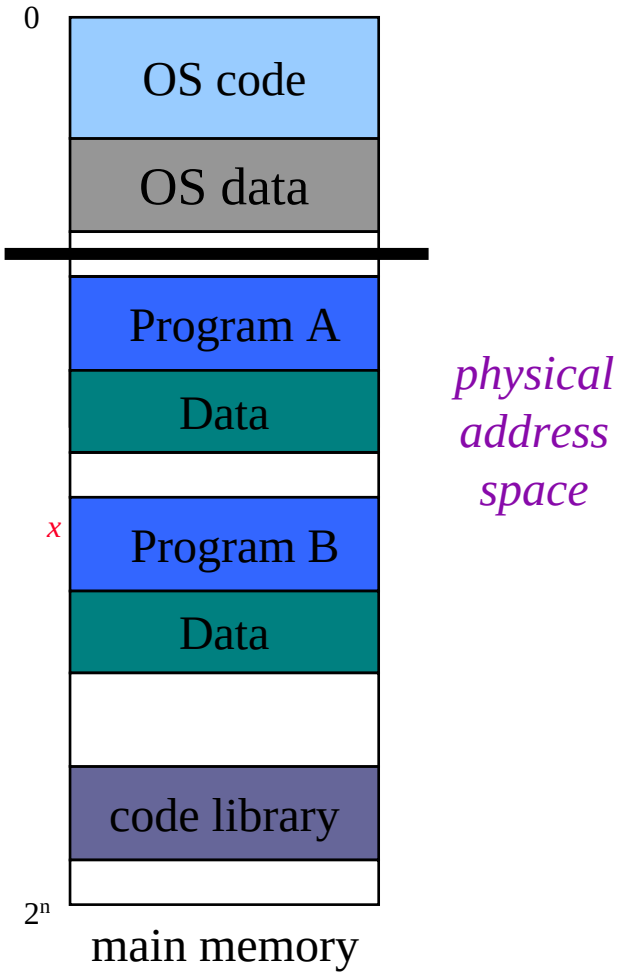


# Kernel Mode

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 register accesses are only legal when the CPU is executing in kernel mode.

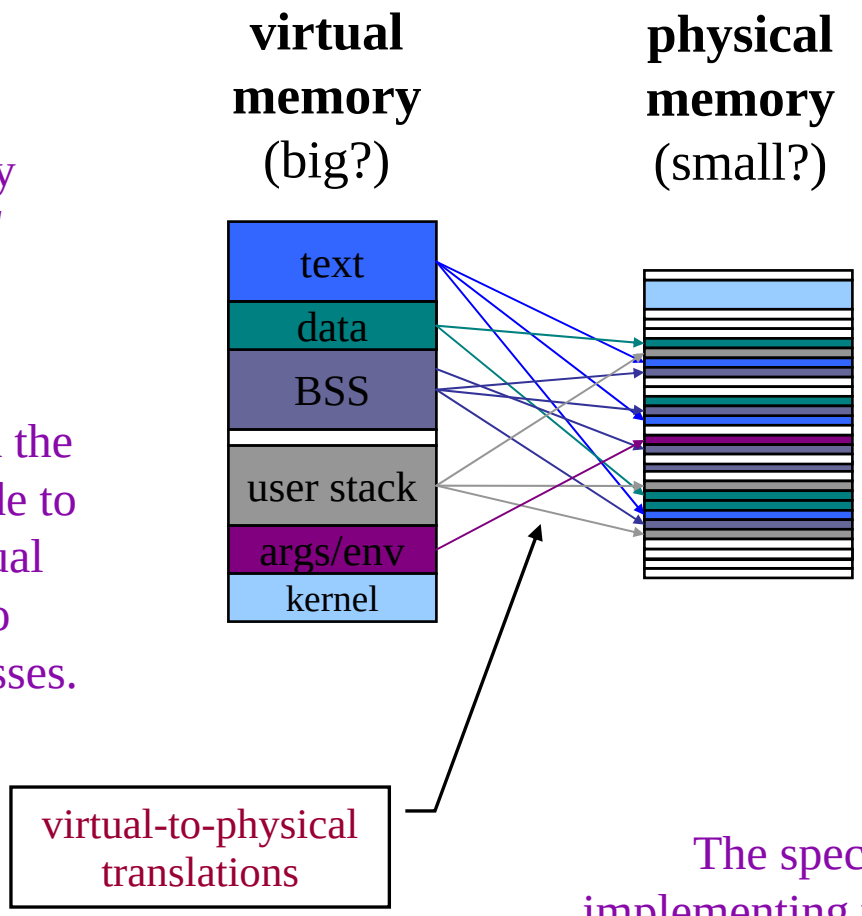


# Introduction to Virtual Addressing

User processes address memory through *virtual addresses*.

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



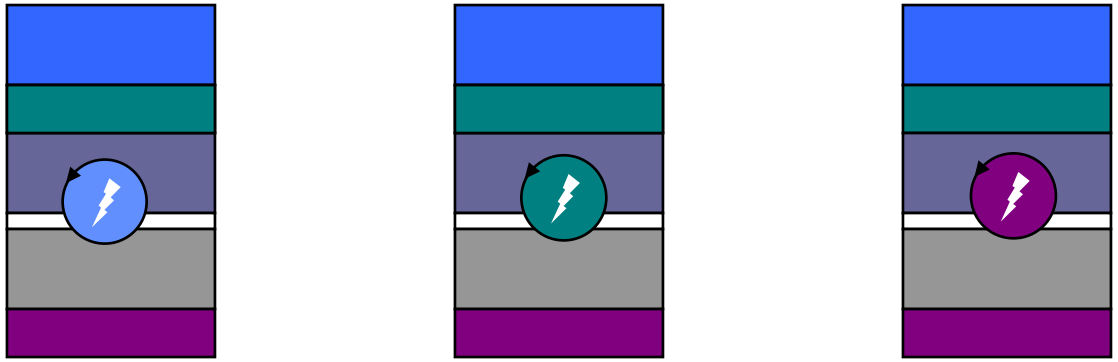

The kernel controls the virtual-physical translations in effect for each space.

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*.

# Processes and the Kernel

*processes in private virtual address spaces*

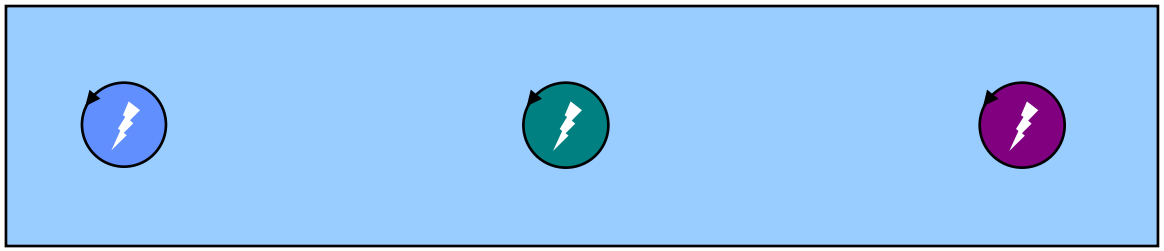


The kernel sets up process execution contexts to "virtualize" the machine.

*system call traps*

*...and upcalls (e.g., signals)*

*shared kernel code and data in shared address space*



Threads or processes enter the kernel for services.

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

# 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., XP).

- 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 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*.

# Protecting Entry to the Kernel

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

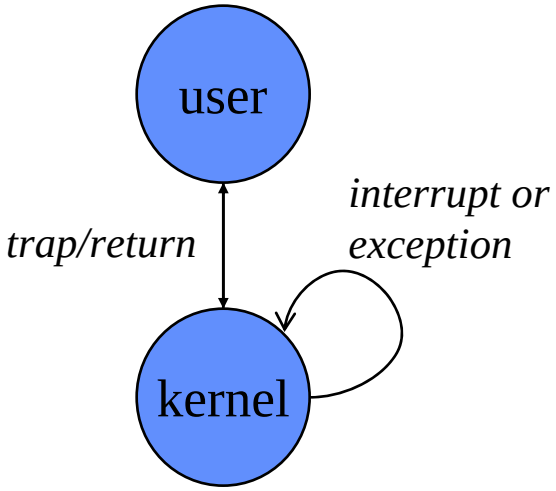
- The *machine* defines a small set of exceptional event types.
- The *machine* defines what conditions raise each event.
- The kernel installs handlers for each event at boot time.

  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.



# Example: System Call Traps

User code invokes kernel services by initiating *system call* traps.

- Programs in C, C++, etc. invoke system calls by linking to a *standard library* of procedures written in assembly language.

  the library defines a *stub* or *wrapper* routine for each syscall

  stub executes a special **trap** instruction (e.g., **chmk** or **callsys** or **int**)

  syscall arguments/results passed in registers or user stack

Alpha CPU architecture

read() in Unix libc.a library (executes in user mode):

```
#define SYSCALL_READ 27      # code for a read system call

move arg0…argn, a0…an        # syscall args in registers A0..AN
move SYSCALL_READ, v0        # syscall dispatch code in V0
callsys                      # kernel trap
move r1, _errno              # errno = return status
return
```

# Faults

Faults are similar to system calls in some respects:

- Faults occur as a result of a process executing an instruction.

  Fault handlers execute on the process kernel stack; the fault handler may block (sleep) in the kernel.

- The completed fault handler may return to the faulted context.

But faults are different from syscall traps in other respects:

- Syscalls are deliberate, but faults are “accidents”.

  divide-by-zero, dereference invalid pointer, memory page fault

- Not every execution of the faulting instruction results in a fault.

  may depend on memory state or register contents

# The Role of Events

A CPU *event* is an "unnatural" change in control flow.

- Like a procedure call, an event changes the PC.
- Also changes mode or context (current stack), or both.
- Events do *not* change the current space!

The kernel defines a *handler* routine for each event type.

- Event handlers always execute in kernel mode.
- The specific types of events are defined by the machine.

Once the system is booted, *every entry to the kernel occurs as a result of an event.*

- In some sense, the whole kernel is a big event handler.

# CPU Events: Interrupts and Exceptions

An *interrupt* is caused by an external event.

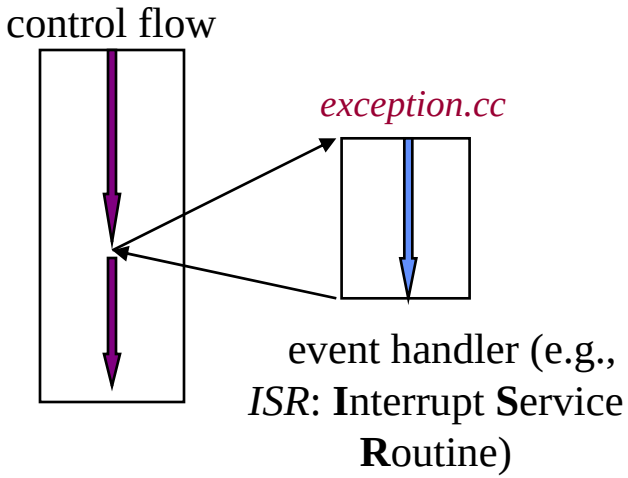
device requests attention, timer expires, etc.

An *exception* is caused by an executing instruction.

CPU requires software intervention to handle a *fault* or *trap*.

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

AST: **A**synchronous **S**ystem **T**rap
Also called a *software interrupt* or an Asynchronous or Deferred Procedure Call (APC or DPC)




*Note*: different "cultures" may use some of these terms (e.g., trap, fault, exception, event, interrupt) slightly differently.

# 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.

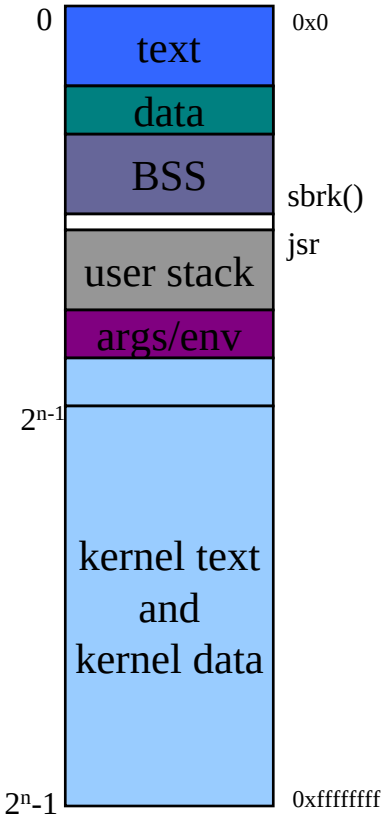
3. *context*: given by register state and execution stream

   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.

# The Virtual Address Space

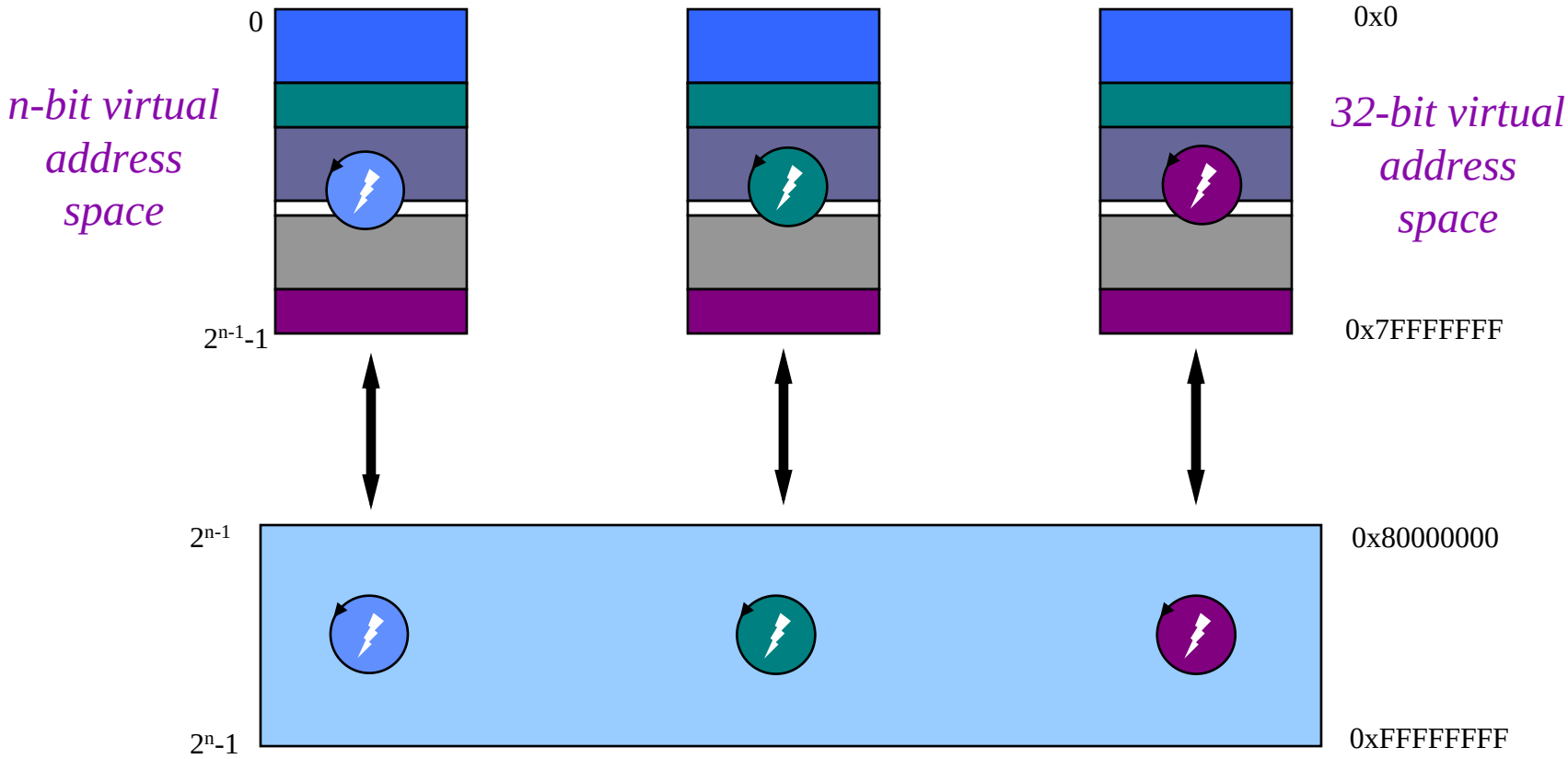



A *typical* process VAS space includes:

- 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, but accessible only to kernel code
- NT (XP?) on x86 subdivides kernel region into an unpaged half and a (mostly) paged upper half at 0xC0000000 for page tables and I/O cache.
- Win95/98 uses the lower half of system space as a system-wide shared region.

*A VAS for a private address space system (e.g., Unix, NT/XP) executing on a typical 32-bit system (e.g., x86).*

# Process and Kernel Address Spaces



# The OS Directs the MMU

The OS controls the operation of the MMU to select:

(1) the subset of possible virtual addresses that are valid for each process (the process *virtual address space*);

(2) the physical translations for those virtual addresses;

(3) the modes of permissible access to those virtual addresses;

read/write/execute

(4) the specific set of translations in effect at any instant.

need rapid context switch from one address space to another

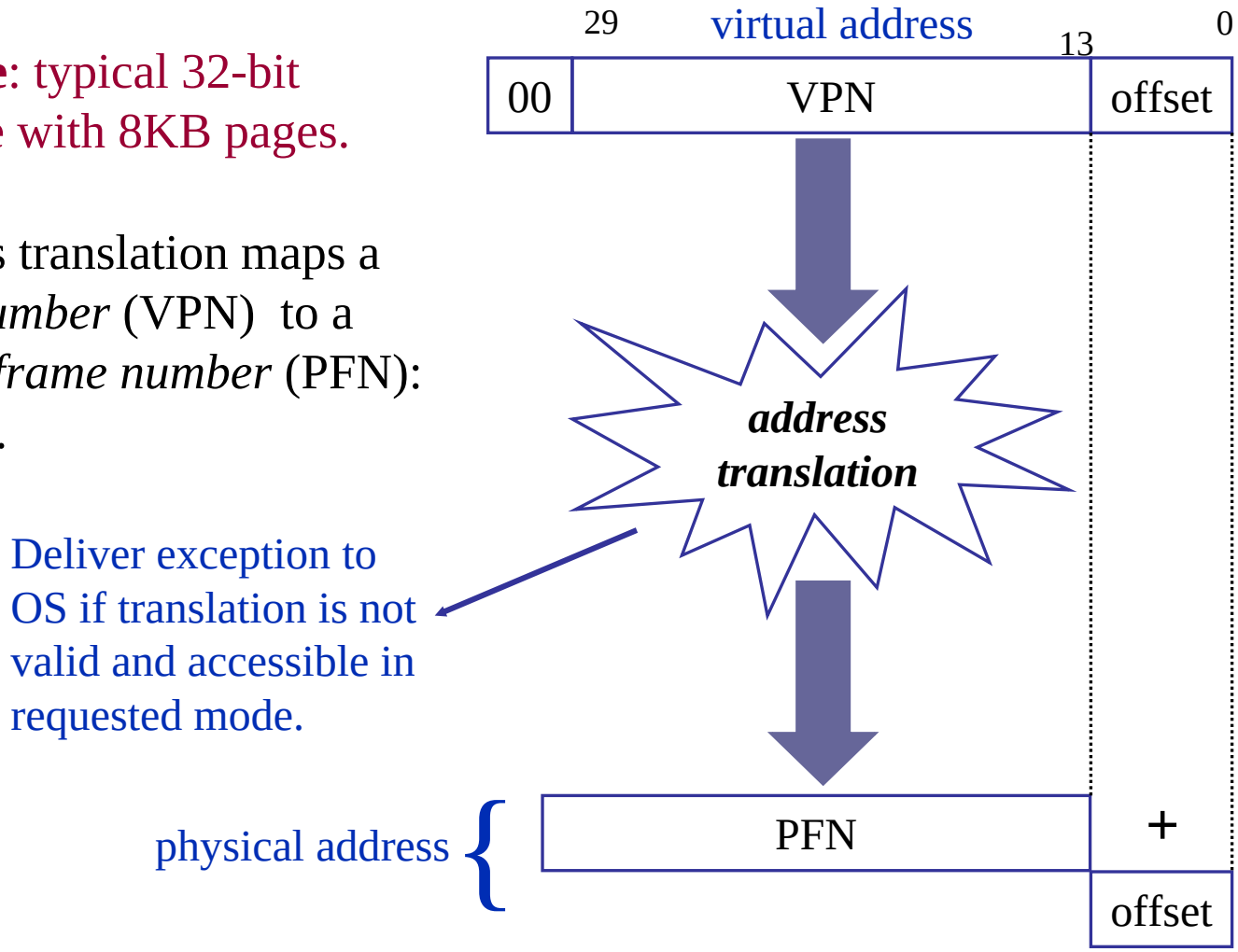
MMU completes a reference only if the OS "says it's OK".

MMU raises an exception if the reference is "not OK".

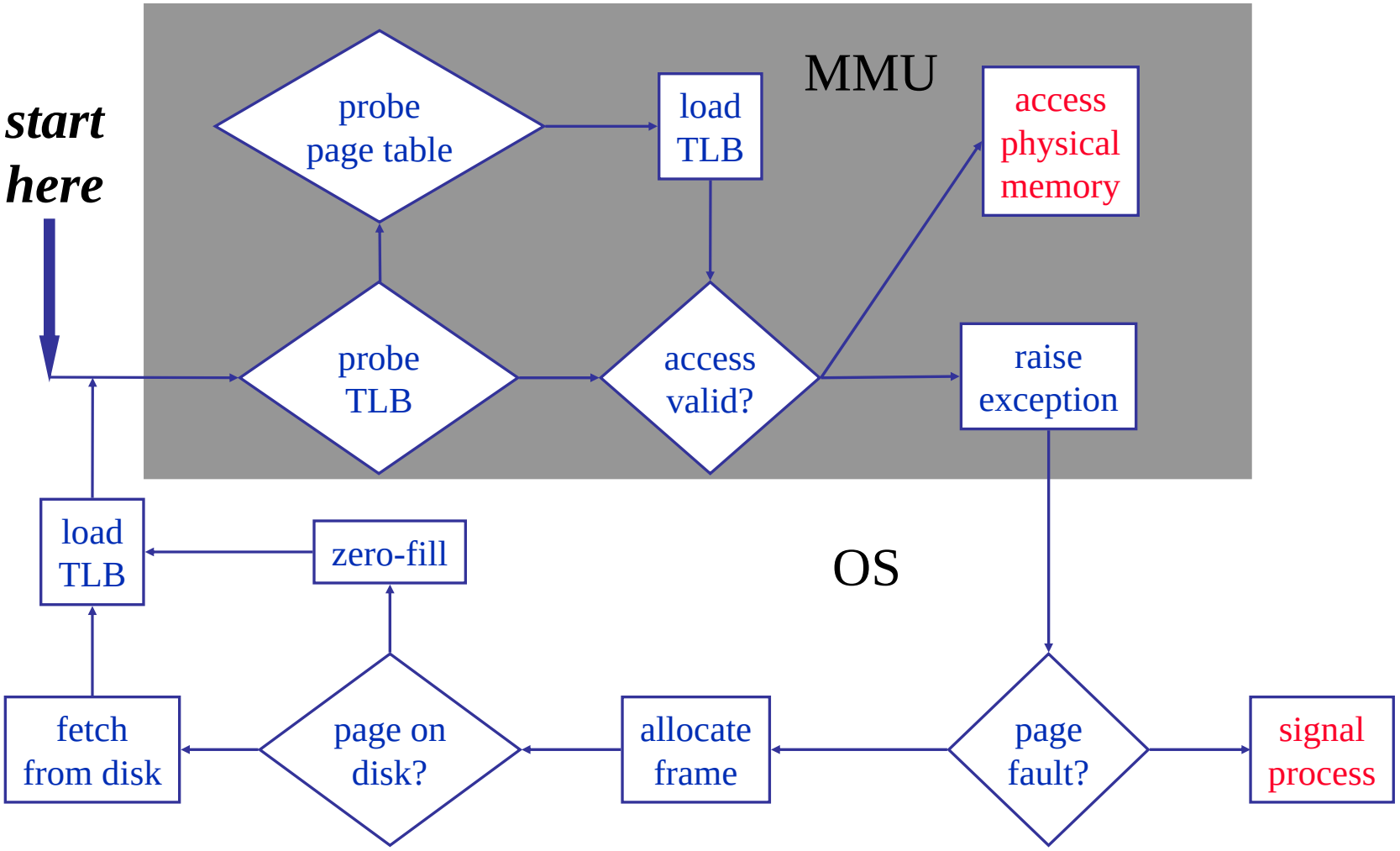
# Virtual Address Translation

**Example**: typical 32-bit architecture with 8KB pages.

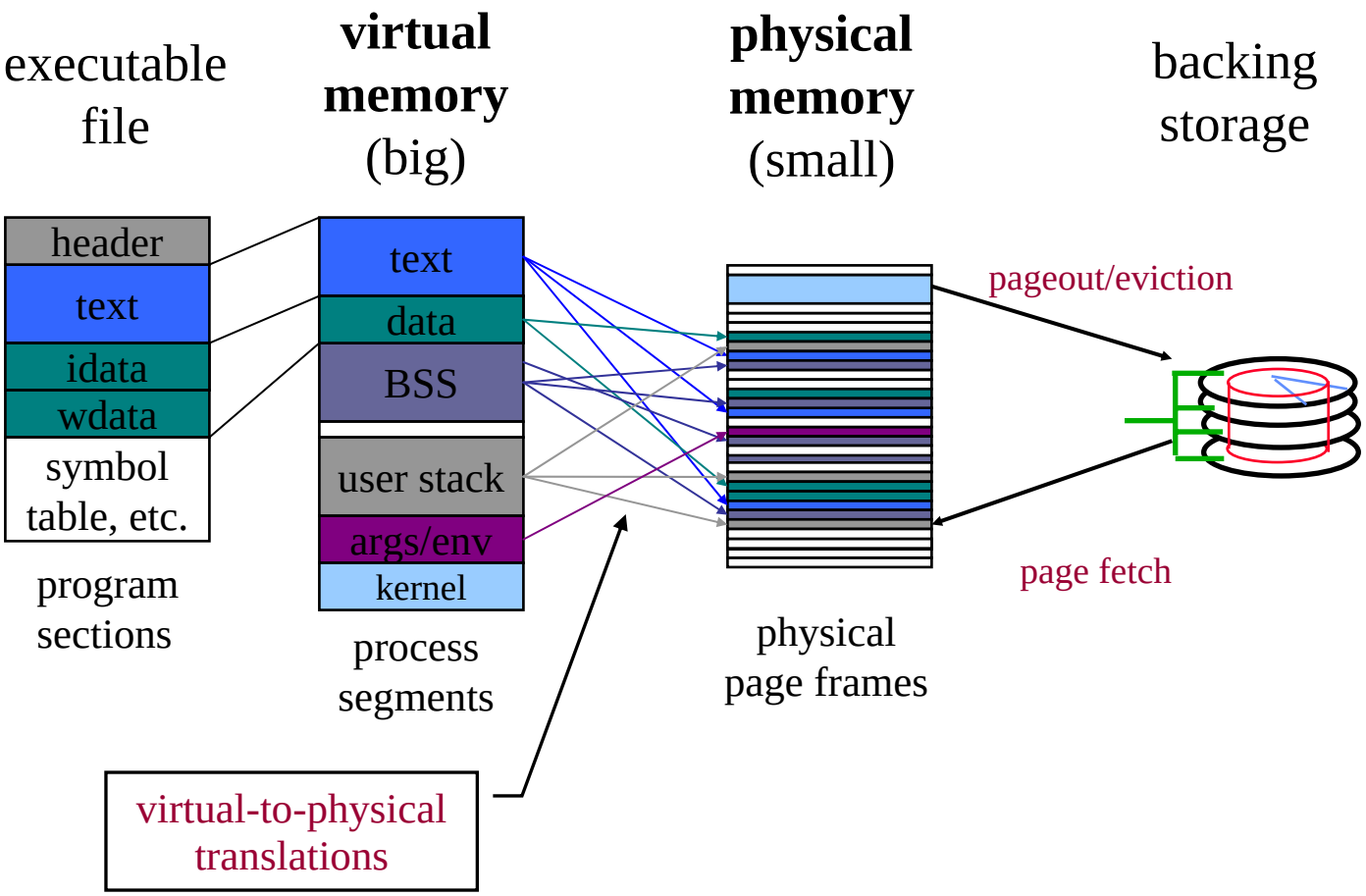
Virtual address translation maps a *virtual page number* (VPN) to a physical *page frame number* (PFN): the rest is easy.



# Completing a VM Reference



# Virtual Memory as a Cache



# Wrapping Up

There is lots more to say about address translation, but we don't want to spend too much time on it now.

- On NT/x86, each address space has a *page directory*
- One page: 4K bytes, 1024 4-byte entries (PTEs)
- Each PDIR entry points to a "page table"
- Each "page table" is one page with 1024 PTEs
- each PTE maps one 4K page of the address space
- Each page table maps 4MB of memory: 1024*4K
- One PDIR for a 4GB address space, max 4MB of tables
- Load PDIR base address into a register to activate the VAS

# What did we just do?

We used special machine features to “virtualize” a core resource: memory.

- Each process/space only gets some of the memory.
- The OS decides how much you get.
- The OS decides what parts of the program and its data are in memory, and what parts you will have to wait for.
- You can’t tell exactly what you have.
- The OS isolates each process from its competitors.

*Virtualization involves a clean abstract interface with a level of indirection that enables the system to interpose on important actions, securely and transparently, in order to cover up ugly details of the environment.*

# Sharing the CPU

We have seen how an operating system can share and "virtualize" one hardware resource: memory.

How can does an OS share the CPU among multiple running programs (processes)?

- Safely
- Fairly (?)
- Efficiently

# Sharing Disks

How should the OS mediate/virtualize/share the disk(s) among multiple users or programs?

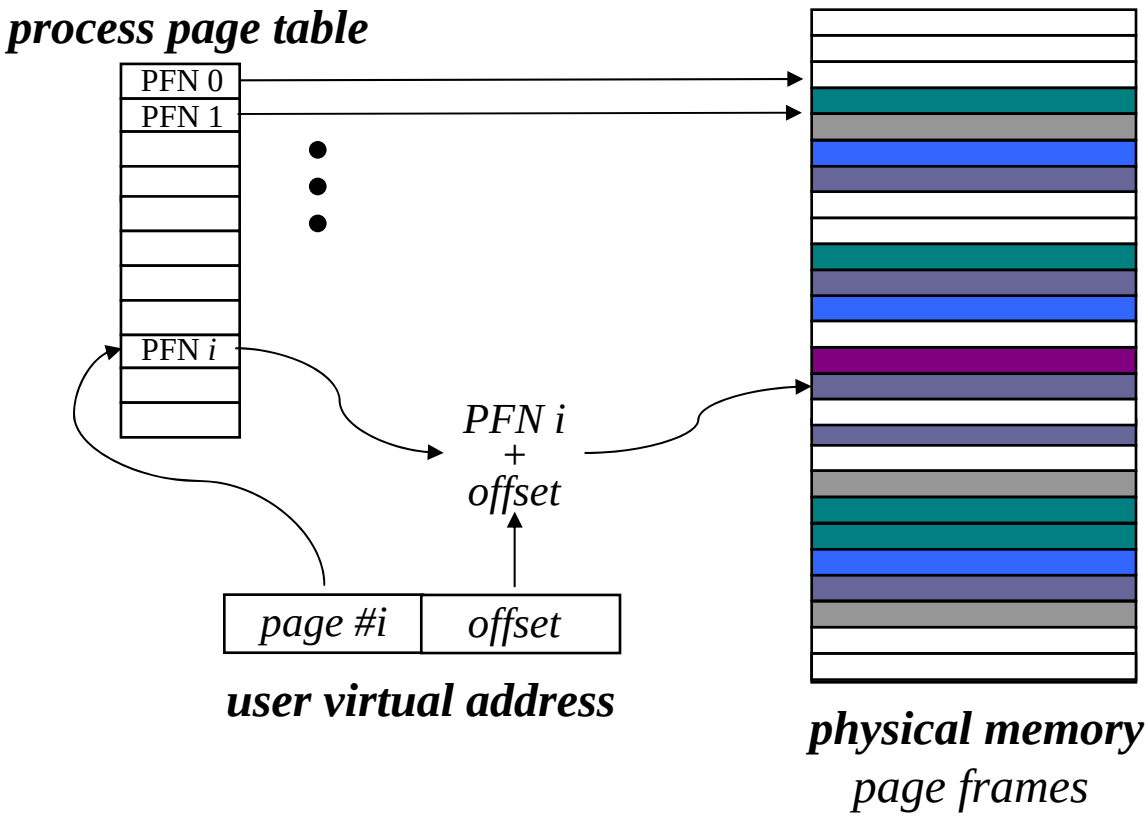
- Safely
- Fairly
- Securely
- Efficiently
- Effectively
- Robustly

# Classical View: The Questions

The basic issues/questions in this course are *how to*:

- allocate memory and storage to multiple programs?
- share the CPU among concurrently executing programs?
- *suspend* and *resume* programs?
- share data safely among concurrent activities?
- protect one executing program's storage from another?
- protect the code that implements the protection, and mediates access to resources?
- prevent rogue programs from taking over the machine?
- allow programs to interact safely?

# A Simple Page Table

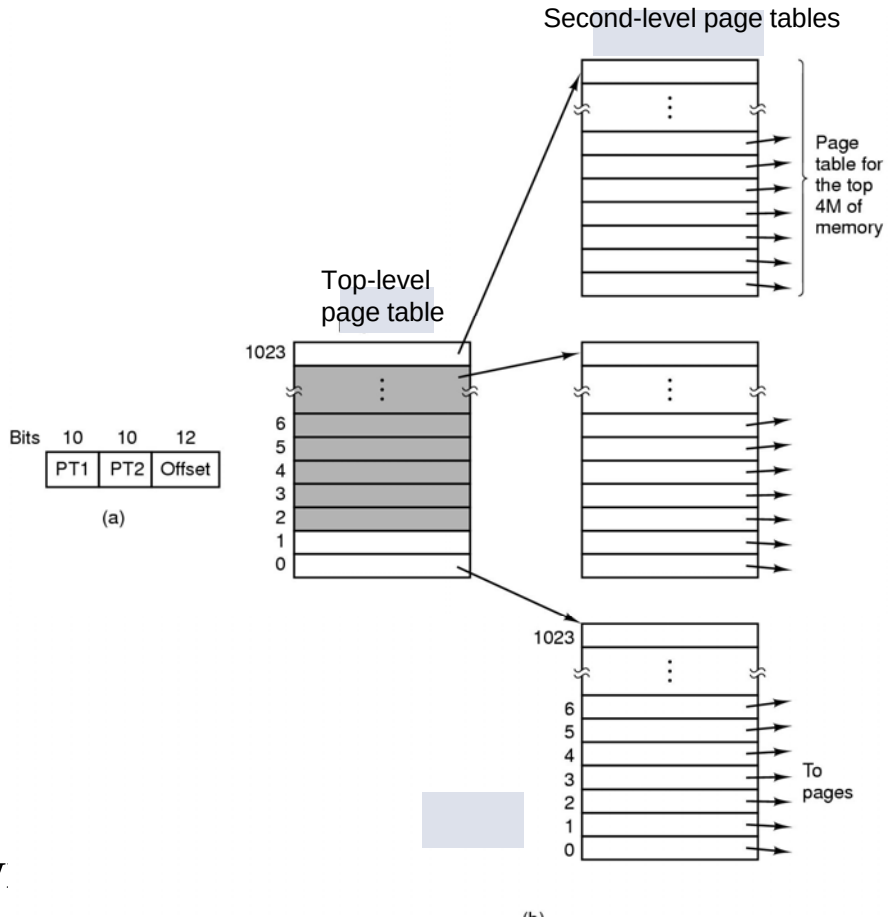



Each process/VAS has its own page table. Virtual addresses are translated relative to the current page table.

In this example, each VPN *j* maps to PFN *j*, but in practice any physical frame may be used for any virtual page.

The page tables are themselves stored in memory; a protected register holds a pointer to the current page table.

# Page Tables (2)




32 bit address w

Two-level page tables

[from Tanenbaum]