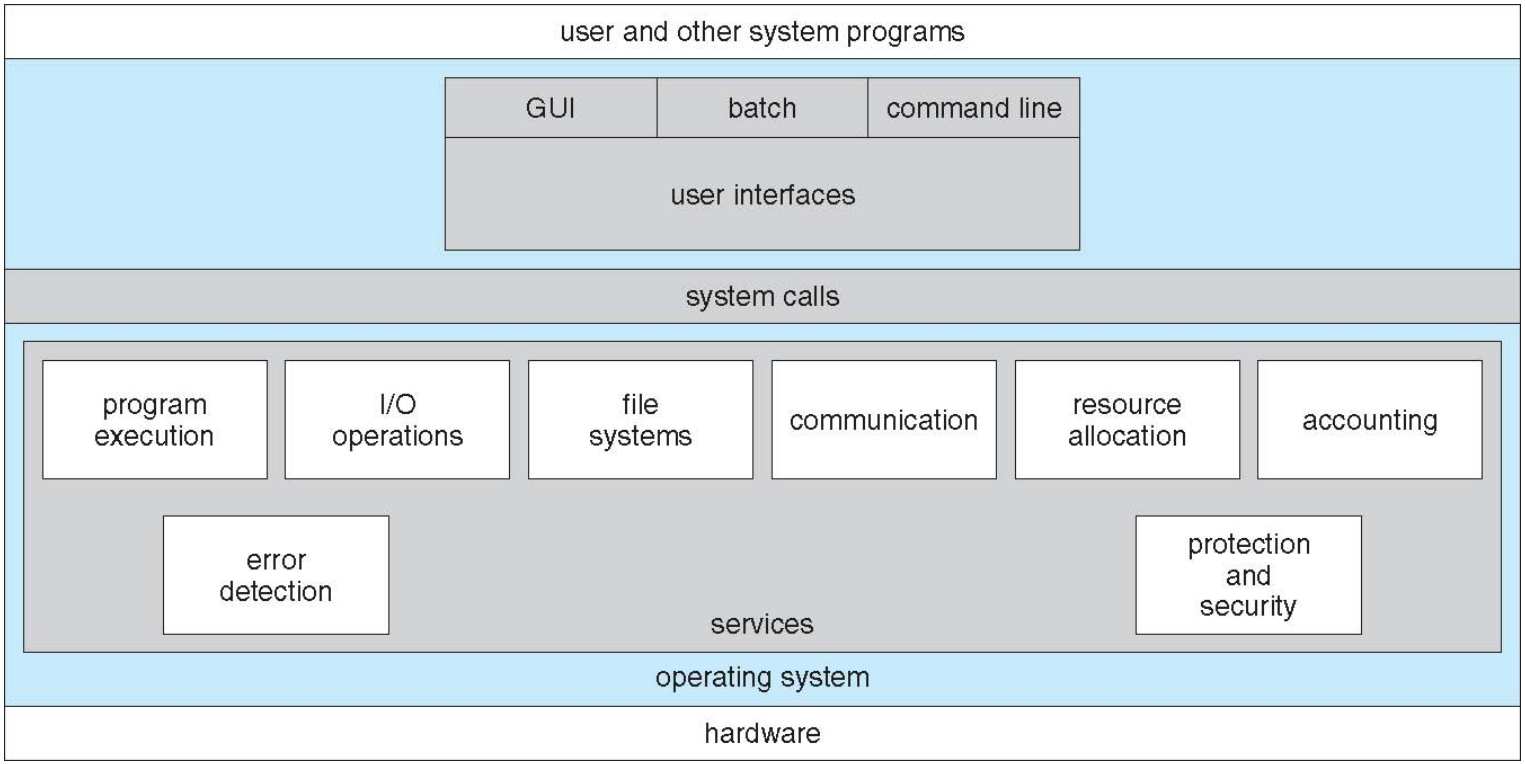
Anthony Ventresque
anthony.ventresque@ucd.ie

Protection in OS

School of Computer Science, UCD

Scoil na Ríomheolaíochta, UCD

Previously…

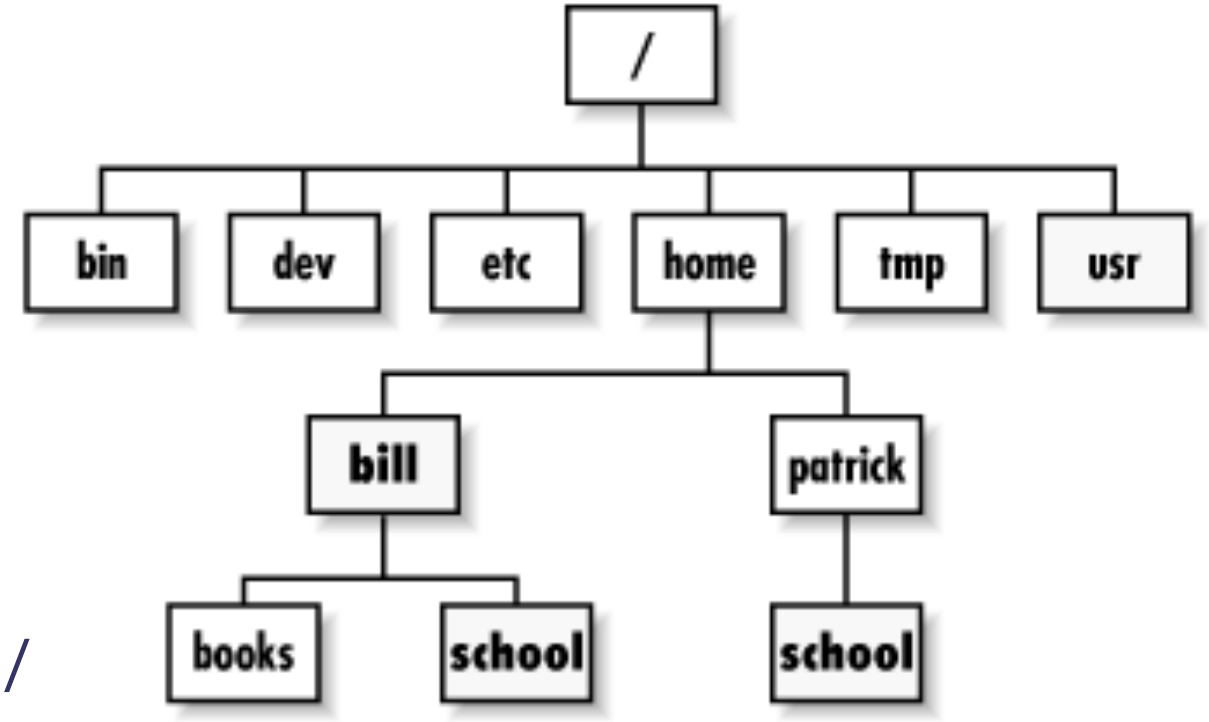

user and other system programs
GUI
batch
command line
user interfaces
system calls
program execution
I/O operations
file systems
communication
resource allocation
accounting
error detection
protection and security
services
operating system
hardware

Previously…

- General-purpose OS is very large program
- Various ways to structure OS
  - ***Simple*** Structure: Only one or two levels of code (e.g., MS-DOS)
  - ***Monolithic*** Architectures: More complex (e.g., UNIX)
  - ***Layered***: Lower levels independent of upper levels (an abstraction)
  - ***Microkernel***: OS built from many user-level processes (Mach)
  - ***Modular***: Core kernel with Dynamically loadable modules

LINUX DIRECTORY TREE

Linux Directory Tree



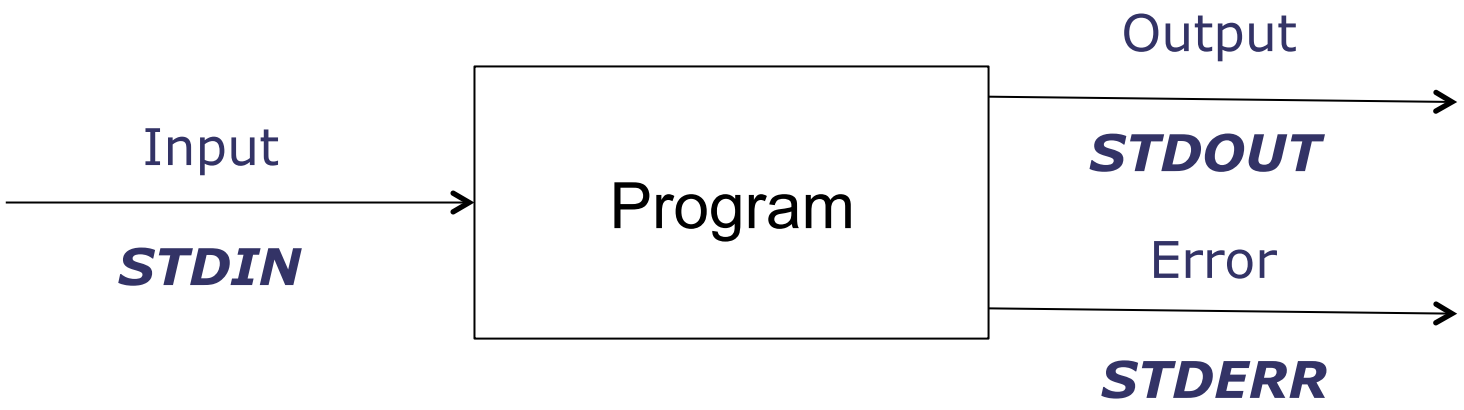
- /
- /home
- /home/bill
- /home/bill/books

PROGRAM I/O, REDIRECTION, PIPES

Program Input and Output

- Each program when executed has 3 files preset for it:
  - One input file, ***STDIN*** (usually the keyboard)
  - and 2 seperate output files, ***STDOUT*** and ***STDERR*** (usually the console, terminal or shell window)
- (In Unix/Linux "everything is a file" - even the keyboard/ screen)
- The program can of course completely ignore these and get its inputs, and set its outputs to anything it wants. But most of the standard Unix/Linux command line tools follow a convention in the use of these preset files.
  - If there is no alternative input file(s) specified, then the program takes its input from STDIN
  - It directs its normal output to STDOUT
  - It directs any error messages to STDERR

Program Input and Output



Redirection

- When executing programs, the SHELL can change what "file" is connected to STDIN, STDOUT and STDERR.

- Redirecting STDOUT
  - Use the '>' or '>>' symbols to change where the normal output goes. '>>' causes the output to be appended to any previous contents of the file.
  - $> ls –l > ls_list
  - $> ls –l /etc >> ls_list

Redirection

- When executing programs, the SHELL can change what "file" is connected to STDIN, STDOUT and STDERR.

- Redirecting STDIN
  - Use the '<' symbol to change where the normal input comes from
  - $> wc –l < ls_list

Redirection

- When executing programs, the SHELL can change what "file" is connected to STDIN, STDOUT and STDERR.

- Redirecting STDERR
  - Use '2>' to change where the error output goes, 2 because STDERR is file 2. To append use '2>>’
  - $> ls –l /home/brangelina 2> ls_list
  - $> ls -l Z* 2> /dev/null
  - (note /dev/null is like a permanent bin)

Joining Programs Together: Pipes

- The shell can use a ***PIPE*** (see man pipe) to feed the output of one program into the input of another. This corresponds to joining STDOUT of one command with STDIN of a second command. Use the '**|**' symbol between the 2 commands.
  - $> ls -l /etc | less

Outline

- Protection
  - Dual Mode operation
  - System Calls
  - Memory, I/O and CPU Protection

Take home message:

OS and hardware need protection from user programs, which is ensured by Dual Mode and isolation of programs from one another.

The Need for Protection: Multiple Apps

- OS needs to provide Full Coordination and Protection
  - Manage interactions between different users
  - Multiple programs running simultaneously
  - Multiplex and protect Hardware Resources
    - CPU, Memory, I/O devices like disks, printers, etc.

Example: Protecting Processes from Each Other

- Problem: Run multiple applications in such a way that they are protected from one another
- Goal:
  - Keep User Programs from Crashing OS
  - Keep User Programs from Crashing each other
  - [Keep Parts of OS from crashing other parts?]
- (Some of the required) Mechanisms:
  - Dual Mode Operation
  - Address Translation
- Simple Policy:
  - Programs are not allowed to read/write memory of other Programs or of Operating System

Protected Instructions

- In modern systems some instructions within the whole set of instructions of the CPU are typically restricted to the OS (***protected or privileged instructions***)
  - otherwise a faulty or malicious user program may affect the whole system
- Typically, users are not allowed
  - to directly access I/O (disks, printers, displays, etc.)
  - to directly manage memory
  - to execute CPU halt instructions
  - to directly manipulate the system's clock
- These operations are always handled through special instructions called ***system calls*** or memory mapping

Hardware Support: Dual-mode Operation

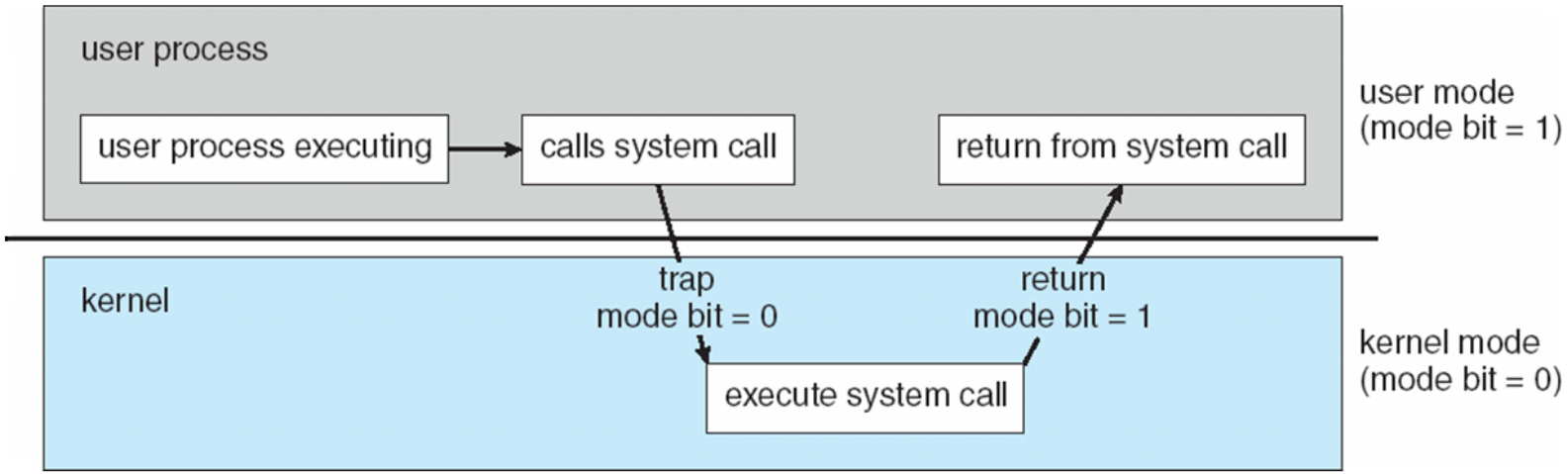
- The implementation of protected instructions requires a hardware mechanism: modes of operation
- ***Kernel mode*** (or "supervisor" or "protected")
  - Execution with the full access to the hardware
  - Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet
- ***User mode*** (Normal programs executed)
  - Limited access restricted to a subset of the instruction set
  - Only those granted by the operating system kernel

Hardware Support: Dual-mode Operation

- What is needed in hardware to support "dual mode" operation?
  - The mode is indicated by a status bit (mode bit) in a special processor register (which can only be altered in kernel mode)
  - OS programs & protected instructions executed in kernel mode
  - User programs executed in user mode
  - Certain operations / actions only permitted in system/ kernel mode
    - In user mode they fail or trap

Hardware Support: Dual-mode Operation

- User->Kernel transition sets system mode AND saves the user PC (Program Counter)
  - Operating system code carefully puts aside user state then performs the necessary operations
- Kernel->User transition clears system mode AND restores appropriate user PC
  - return-from-interrupt

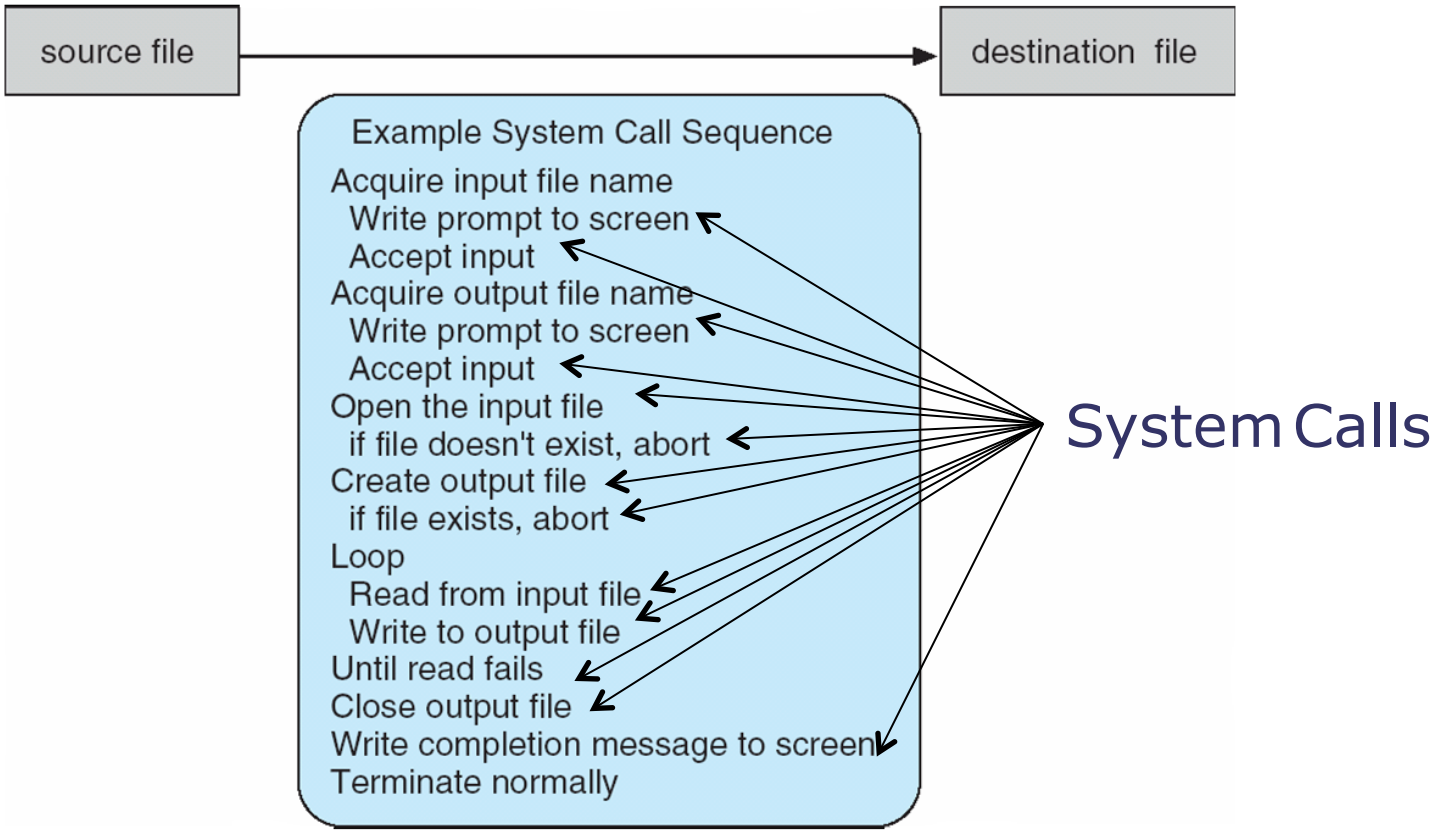


Crossing Protection Boundaries

- User-mode programs cannot execute privileged instructions, but they still need kernel-mode services (I/O operations, memory management, etc.)

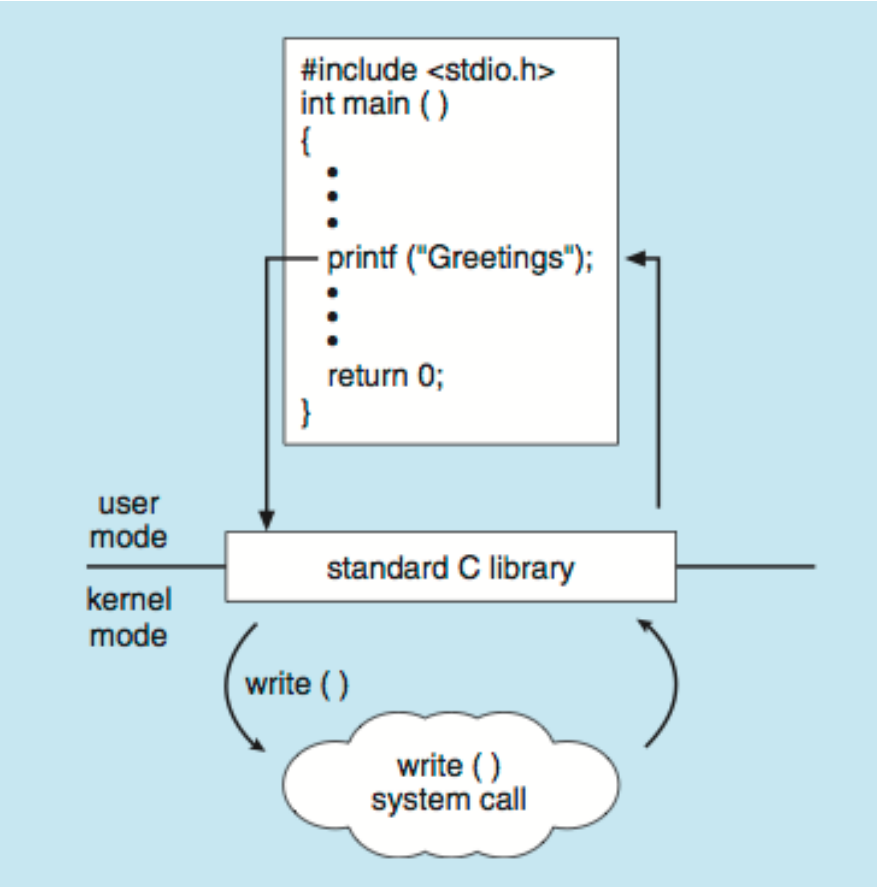
- To execute a privileged instruction, a user must call an OS procedure: ***system call***
  - System calls are the interface between processes and the OS – they are commonly available as an ***application programming interface*** (API) in some high level language (typically C)
  - a system call causes a trap, which is a jump from the code being executed to the trap handler in the kernel
  - a trap is treated by the hardware as a software-initiated ***interrupt***

Example of System Calls Involved in one User Program



Standard C Library Example

- C program invoking printf() library call, which calls write() system call



Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

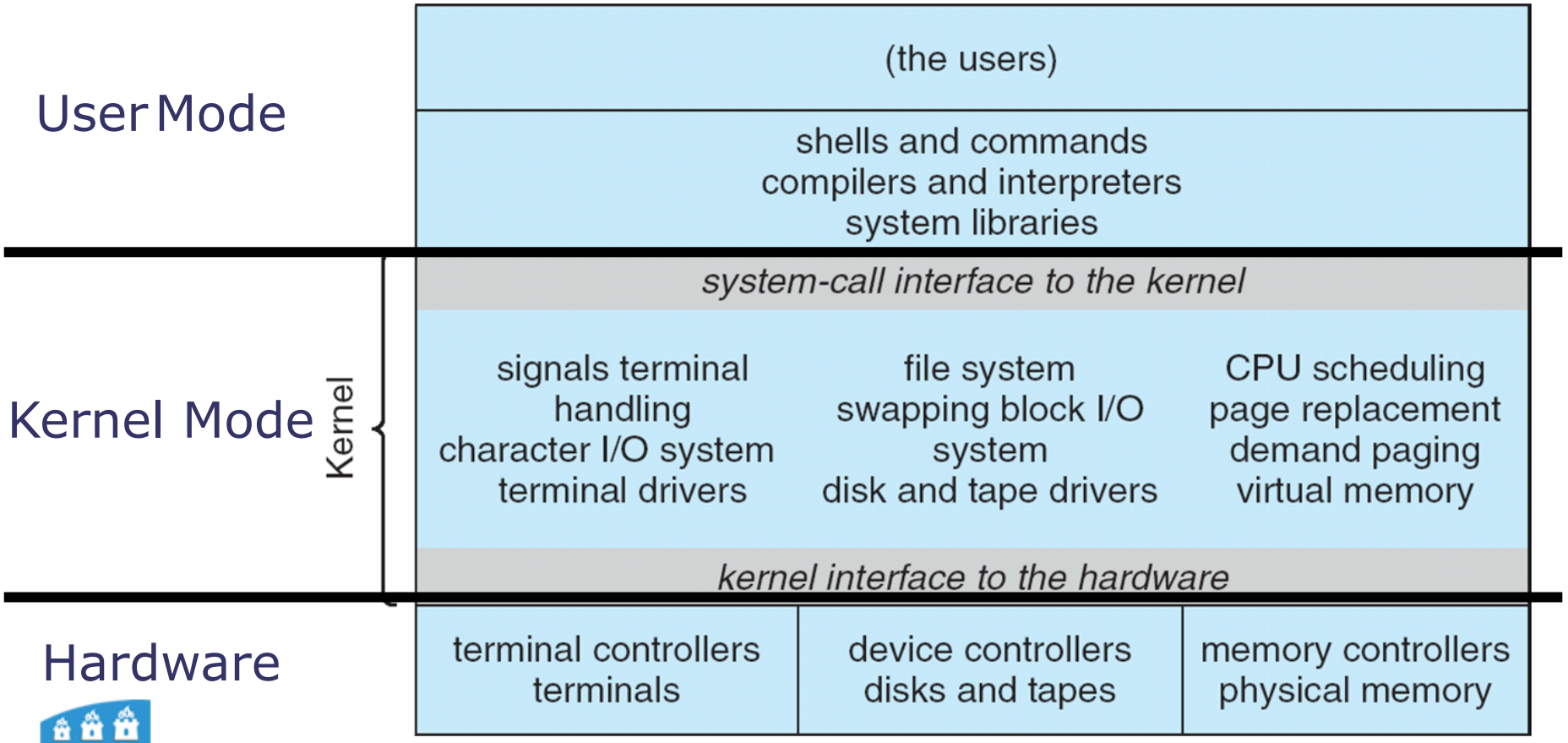
return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

For Example: UNIX System Structure



3 Types of Mode Transfer

- From user mode to kernel mode
  - ***System calls (aka protected procedure call)***
    - Request by program for kernel to do some operation on its behalf
    - Only limited # of very carefully coded entry points
  - ***Interrupts***
    - Triggered by timer, I/O devices
  - ***Exceptions***
    - Triggered by unexpected program behavior
    - Or malicious behavior!

Exceptions

- ***Exception*** (hardware-initiated interrupt):basically the same as a trap, but automatically triggered by an error or a particular situation rather than on purpose
- Typical exceptions that hardware must detect:
  - arithmetic errors: overflow, underflow, division by zero
  - illegal use of privileged instructions
  - memory access out of user space or outside boundaries
  - virtual memory (paging): page faults, write to read-only page – trace traps (debugging)
- Exceptions also transfer control to a handler within the OS
  - system status can be saved on exceptions (memory dump), so that faulty processes can be later debugged

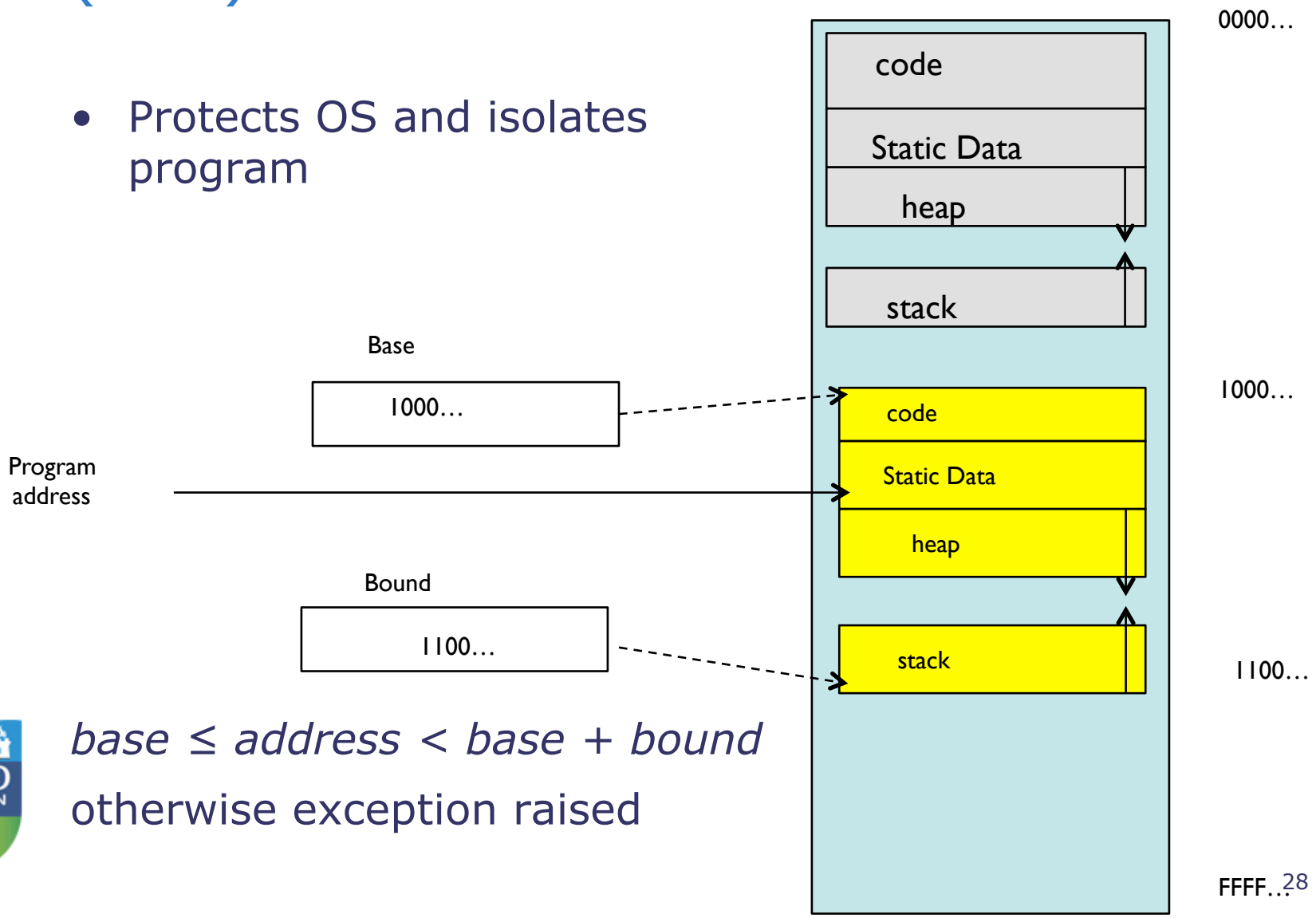
Memory Protection

- OS and user processes exist in the same physical memory
  - Some mechanism must protect the kernel (primary) memory from being accessed by user processes (and also the memory allocated to a process from being accessed by other processes)
- Simplest scheme is to use base and bound registers
  - these registers are loaded by the OS before starting the execution of any program in user mode

  base ≤ address < base + bound; otherwise exception raised
- Modern memory protection is more complex than this
  - However, the scheme above is the basis of modern virtual memory (when a technique called "segmentation" is used)

Simple Protection: Base and Bound (B&B)

- Protects OS and isolates program



base ≤ address < base + bound

otherwise exception raised

I/O Control

- ***All I/O instructions are privileged***: A program could disrupt the whole system by issuing illegal I/O instructions
- Two situations must be contemplated:
  - I/O start: handled by system calls
  - I/O completion and I/O events: handled by interrupts
- Interrupts are the basis for ***asynchronous I/O***
  - I/O devices have small processors that allow them to run autonomously (i.e. asynchronously with respect to the CPU)
  - I/O devices send interrupt signals when done with an operation; CPU switches to address corresponding to interrupt
  - An interrupt vector table contains the list of kernel routine addresses that handle different events
- Sometimes I/O operations are handled by means of memory-mapping rather than by specialised system calls

CPU Protection

- Apart from protecting memory and I/O, we must ensure that the OS always maintains control
  - A user program might get stuck into an infinite loop and never return control to the OS
- ***Timer***: Generates an interrupt after a fixed or variable amount of execution time
  - OS may choose to treat the interrupt as a fatal error (and stop program execution) or allocate more execution time
  - note: with time-sharing a timer interrupt is periodically generated in order to schedule a new process

Conclusion

- The OS needs to provide coordination and protection between multiple applications, and between the OS and applications
- A key protection mechanism is Dual Mode operation. Syscalls, exceptions and interrupts can be used to transition between modes
- A simple scheme for memory protection is to use base and bound registers