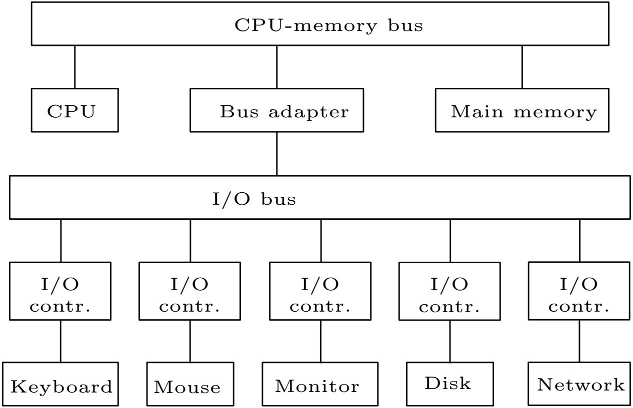
Computer Labs
The Minix 3 Operating System
2º MIEIC

Pedro F. Souto (pfs@fe.up.pt)

September 11, 2012

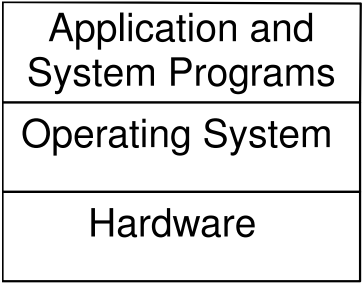
LCOM Labs

- One of the goals of LCOM is that you learn to use the HW-level interface of the most common PC I/O devices



Operating System

- In most modern computer systems, access to the HW is mediated by the operating system (OS)



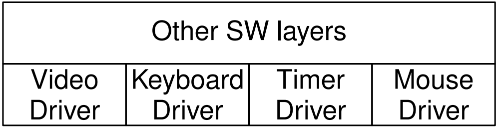
- I.e. user programs are not able to access directly the HW

Parenthesis: Layered Structure

- Structure typically used to address complex problems
  - It allows us to think about the **what** without worrying about the **how** (this is usually called **abstraction**)
- This has several advantages

  Decomposition An “intractable” problem is decomposed in smaller problems that can be solved

  Modularity Facilitates adding new functionality or changing the implementation, as long as the **interfaces are preserved**
- Your project will be a somewhat complex piece of code
  - To structure it in several layers may be very important for your success

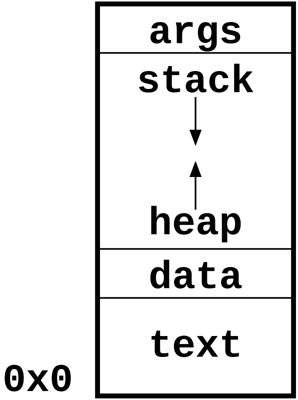


Parenthesis: Program vs. Process

Program Piece of code, i.e. a set of instructions, that can be executed by a processor

Process OS abstraction of a program in execution

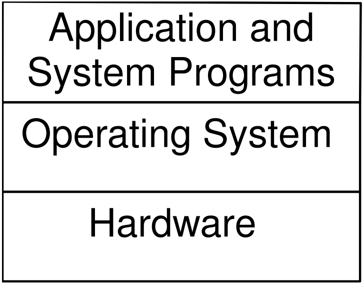
```
int main(int argc, char *argv[], char* envp[])}
```



- `args` Arguments passed in the command line and environment variables
- `stack` *Activation records* for invoked functions
- `heap` Memory region allocated dynamically with `malloc`.
- `data` Memory region allocated statically by the compiller (e.g., a *string* “`Hello, World!`”)
- `text` Program instructions

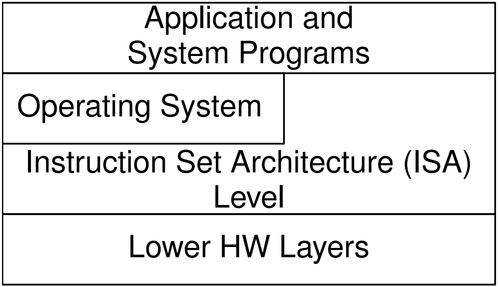
Operating System (repeated)

- In most modern computer systems, access to the HW is mediated by the operating system (OS)



- I.e. user processes are not able to access directly the HW

Access to the HW-level Interface



- Most of the HW interface, actually the processor instruction set, is still available to user processes
- A few instructions however are not directly accessible to user processes
  - Thus preventing user processes from interfering with:

    Other user processes most OSs are multi-process

    The OS which manages the HW resources
- Instead, the operating system offers its own “instructions”, which are known as **system calls.**

OS API: Its System Calls

Hides some ISA instructions

Extends the ISA instructions with a set of “instructions” that support concepts at a higher abstraction level

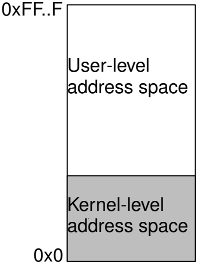
- **Process** A program in execution
- **User** Typically a person, but it can also be a role
- **File** A data source/sink
- Offering an interface that is more convenient to use

Processor Privilege Levels

- Modern computer architectures provide mechanisms to ensure that applications do not bypass the OS interface
  - Most OS support multiple processes
    - Many of them associated to different users
  - Applications should not be allowed to access directly OS code and data
- These mechanisms are usually:
  - At least two execution modes
    - Privileged (kernel) vs. non-privileged (user)
  - A mechanism to change in a controlled way between the execution modes
- The execution mode (privilege level) determines
  - The set of instructions that the processor can execute
  - The range of memory addresses that can be accessed

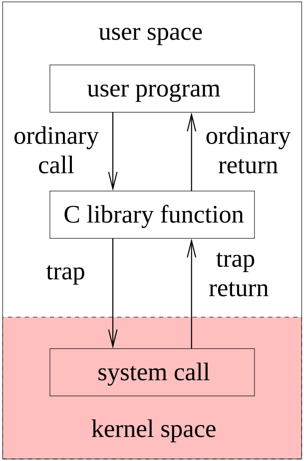
Kernel-level vs. User-level space

- This partitions a process address space in user-level and kernel-level spaces
  - The kernel level address space can be accessed only when the processor executes in privileged mode
    - I.e. kernel code
  - The kernel level address space is shared among all processes



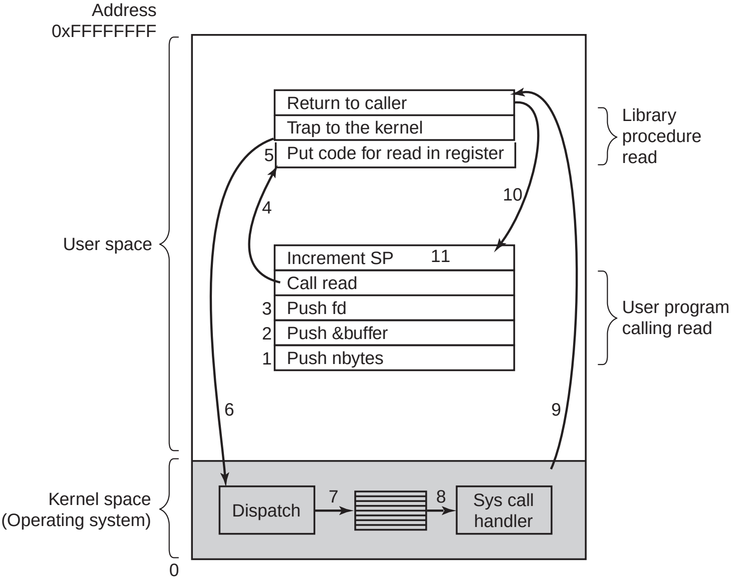
- To support the implementation of system calls, modern processor architectures provide instructions that
  - Switch to privileged execution mode;
  - Transfer execution control (jump) to specific locations in the kernel address space
- An example is the software interrupt instruction `INT` of the IA-32 architecture.

System Call Implementation



- Uses special processor instructions (*call gates* ou *sw interrupts*, in the case of IA32 architecture) that switch automatically the execution mode
- But this is hidden from the programmer
  - Programs call a C library function, which in turn executes the special instruction

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

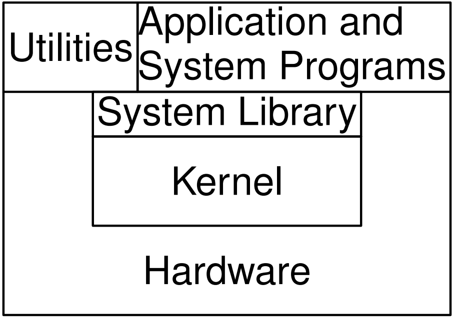


Steps in the execution of `read()`

1, 2, 3 `push` arguments to the *stack*;
4 call C library function `read`;
5 setup of register with the system call #;
6 switch the processor execution mode;
7 dispatch to the appropriate *handler*;
8 execution of the *handler*;
9 **possible** return to the C library function;
10 return from the C library function `read`;
11 *stack* adjustment

OS vs. Kernel

- Usually, when we mention the OS we mean the kernel
- An OS has several components



Kernel Which implements the OS services

Library Which provides an API so that programs can use the OS services

Utilities A set of “basic” programs, that allows a “user” to use the OS services

How is an OS/Kernel implemented?

Monolithic All OS services are implemented at kernel level by the kernel

- Usually, the kernel is developed in a modular fashion
- However, there are no mechanisms that prevent one module from accessing the code, or even the data, of another module

Micro-kernel Most OS services are implemented as modules that execute in their own address spaces

- A module cannot access directly data or even code of another module
- There is however the need for some functionality to be implemented at kernel level, but this is minimal (hence the name)

Monolithic Implementation

- Virtually all “main stream” OSs use this architecture
- It has lower overheads, and is faster

Minix 3: Micro-kernel Based

- It has a very small size kernel (about 6 K lines of code, most of it C)
- Most of the OS functionality is provided by a set of privileged user level processes:

  Services E.g. file system, process manager, VM server, Internet server, and the ressurection server.

  Device Drivers All of them are user-level processes

Issue OS services and device drivers need to execute instructions that are allowed only in kernel mode

- But now, they are executed at user-level

Kernel Calls

Solution The (micro-)kernel provides a set of kernel calls

- These calls allow privileged processes to execute operations that:
  - Can be executed only when running in privileged/kernel mode;
  - That are needed for them to carry out their tasks

Examples from Labs 1 and 2

- `vm_map_phys()`
- `sys_int86()`

Note Kernel calls are (conceptually) different from system calls

- Any process can execute a system call
- Only privileged processes are allowed to execute a kernel call

However, they use the same basic mechanism:

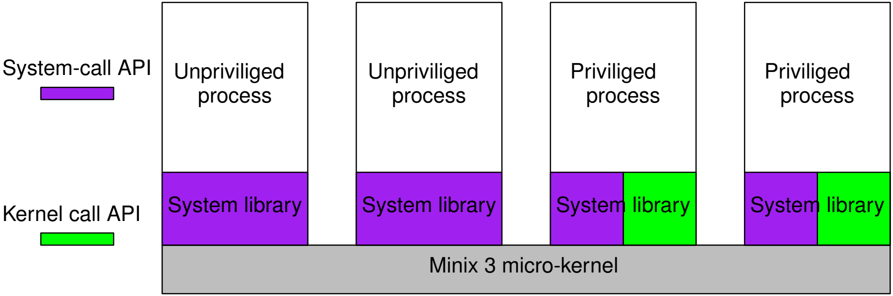
- An instruction that switches to privileged execution mode

Minix 3 Privileged Processes and the Service Utility

- A process must be initiated by the **service** utility in order to become privileged
- **service** reads the privileges of a privileged process from
  - either the `/etc/system.conf`
  - or a file in `/etc/system.conf.d/`

```
service at_wini {
        io
                1f0:8           # Controller 0
                3f6             # Also controller 0
                170:8           # Controller 1
                376             # Also controller 1
                ;
        irq
                14              # Controller 0
                15              # Controller 1
                ;
        system
                UMAP            # 14
                IRQCTL          # 19
                DEVIO           # 21
                SDEVIO          # 22
                VDEVIO          # 23
                READBIOS        # 35
                ;
        pci class
                1/1             # Mass storage / IDE
                1/80            # Mass storage / Other (80 hex)
                1/4             # Mass storage / RAID
                ;
};
```

Minix 3: Non-Privileged vs. Privileged User Processes



LCOM Lab Programs

- In LCOM, you'll use Minix 3 and its kernel-API to develop privileged programs:
  - Akin to device-drivers
    - They will access/control I/O devices
  - Different from device drivers. Your programs:
    - Will be self-contained

    Whereas each device driver:
    - Manages a class of I/O devices
    - Provides an interface so that other processes can access I/O devices of that class
- The use of Minix 3 simplifies the development
  - These processes do not belong to the kernel
  - Their actions can be controlled

  Thus, bugs are much less harmful