Operating Systems

Second part of Operating Systems and Systems Programming Module

University of Birmingham

Eike Ritter

Overview

Operating Systems

Second part of the course: Operating Systems
Outline of lecture:

- What is an operating system?
- How to interact with the operating system
- Possible operating systems architectures
- How to write programs in the kernel
- Virtual machines

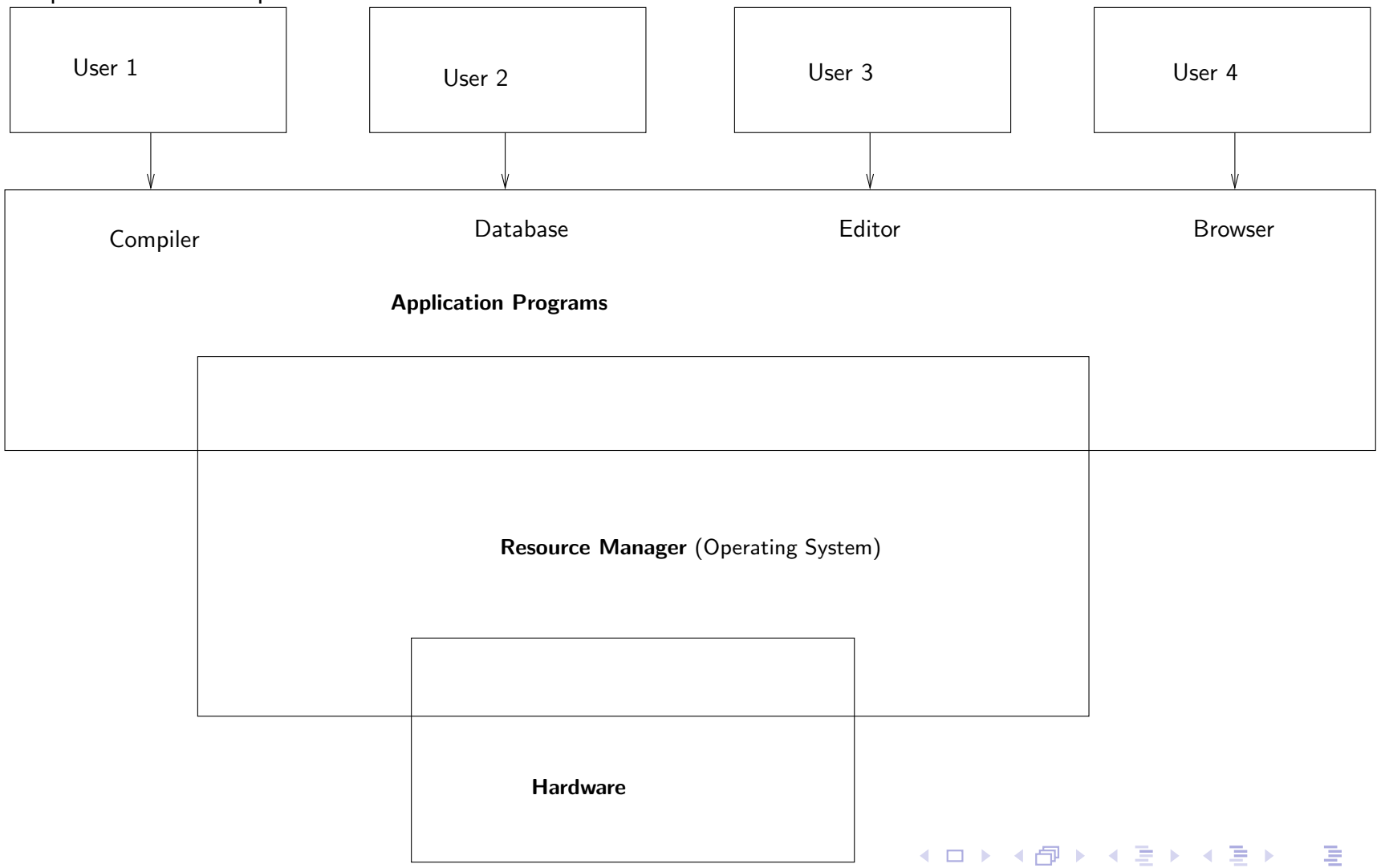
Recommended books

Recommended books for this part of the course:

- Operating Systems Concepts Silberschatz *et al.*
- Linux Kernel Development. Robert Love.
- Linux Programming Interface, Michael Kerrisk
- The C Programming Language (2nd Edition) Kernighan and Ritchie

What is an Operating System?

A program that acts as an intermediary between a user of a computer and the computer hardware



Main functions of an Operating Systems:

OS as a resource allocator:

Manages all resources (eg hardware) and decides between conflicting requests for efficient and fair resource use (*e.g.* accessing disk or other devices)

OS as a control system:

Controls execution of programs to prevent errors and improper use of the computer (*e.g.* protects one user process from crashing another)

Examples of Operating Systems

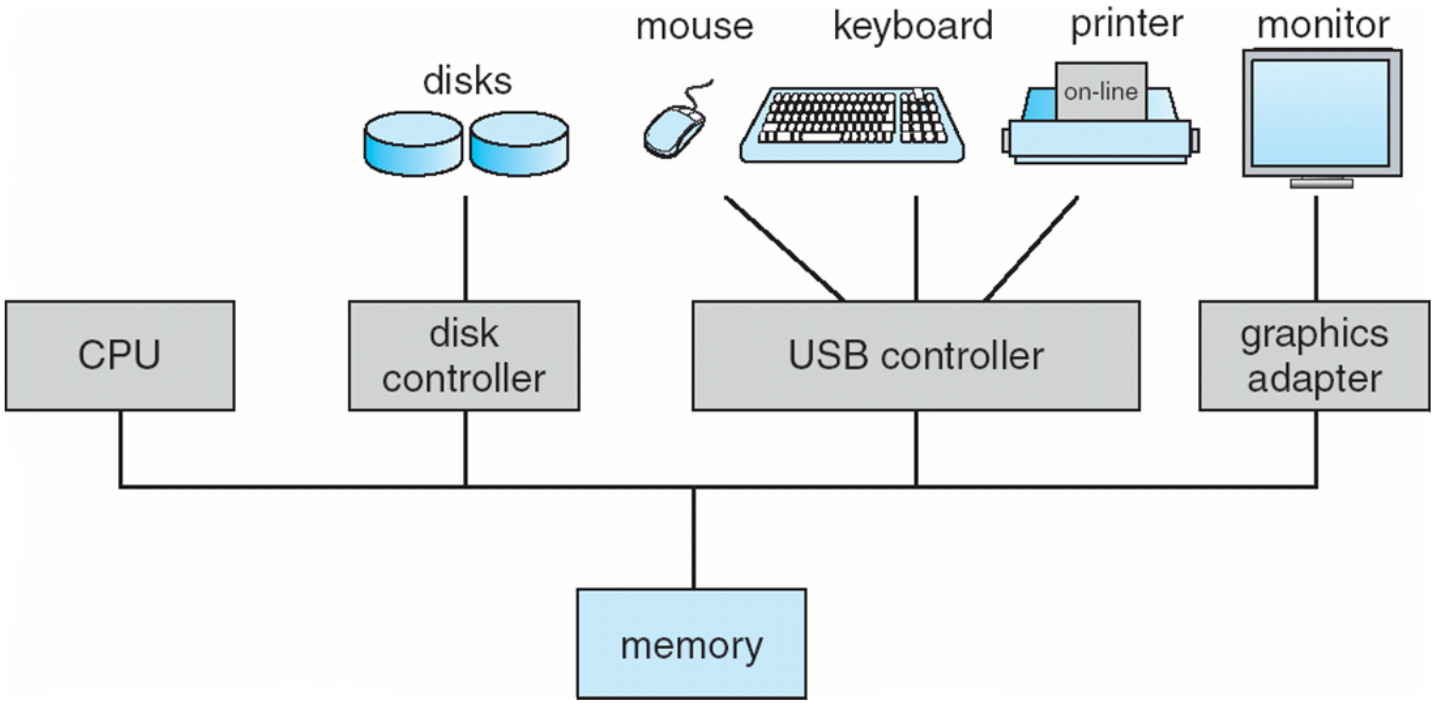
- Windows: current Windows OS based on Windows NT
- Mac OS: based on BSD Unix
- Linux: Based on Unix. Used also for Android.
- Newer operating systems for realtime applications and small reesource usage (eg for Internet of Things)

Operating System Topics

Bootstrapping of the OS

- Small bootstrap program is loaded at power-up or reboot
  - Typically stored in ROM or EPROM, generally known as firmware (*e.g.* BIOS)
- Initializes all aspects of the system (*e.g.* detects connected devices, checks memory for errors, *etc.*)
- Loads operating system kernel and starts its execution

Computer System Organisation



- One or more CPUs, device controllers connect through common bus providing access to shared memory
- CPU(s) and devices compete for memory cycles (*i.e.* to read and write memory addresses)

Computer System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller (*e.g.* controller chip) is in charge of a particular device type
- Each device controller has a local buffer (*i.e.* memory store for general data and/or control registers)
- CPU moves data from/to main memory to/from controller buffers (*e.g.* write this data to the screen, read coordinates from the mouse, *etc.*)
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an *interrupt*

Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction so original processing may be resumed
- Incoming interrupts are disabled while another interrupt is being processed to prevent a lost interrupt
- A trap is a software-generated interrupt caused either by an error or a user request

Storage Structure

- Main memory - only large storage media that the CPU can access directly
- Secondary storage - provides large non-volatile storage capacity
- Important example: magnetic disks - rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into tracks, which are subdivided into sectors
  - The disk controller determines the logical interaction between the device and the computer
- Today also often flash memory
  However, same logical division intro tracks and sectors still used

OS Services

How do Users and Processes interact with the Operating System?

- **Users** interact indirectly through a collection of system programs that make up the operating system interface. The interface could be:
  - A GUI, with icons and windows, *etc.*
  - A command-line interface for running processes and scripts, browsing files in directories, *etc.*
  - Or, back in the olden days, a non-interactive batch system that takes a collection of jobs, which it proceeds to churn through (*e.g.* payroll calculations, market predictions, *etc.*)
- **Processes** interact by making *system calls* into the operating system proper (*i.e.* the *kernel*).
  - Though we will see that, for stability, such calls are not direct calls to kernel functions.

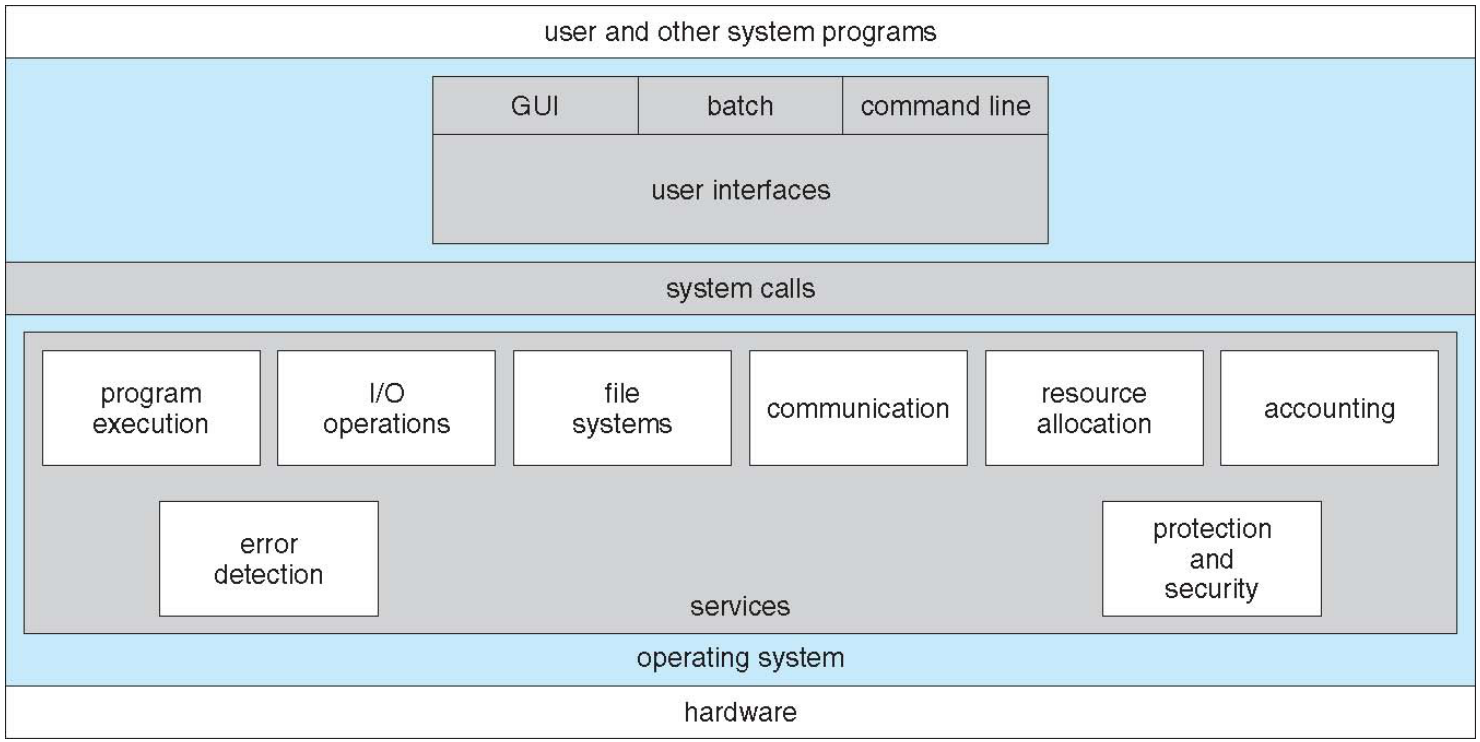
Services for Processes

- Typically, operating systems will offer the following services to processes:
  - **Program execution**: The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations**: A running program may require I/O, which may involve a file or an I/O device
  - **File-system manipulation**: Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
  - **Interprocess Communication (IPC)**: Allowing processes to share data through message passing or shared memory

Services for the OS Itself

- Typically, operating systems will offer the following internal services:
  - **Error handling**: what if our process attempts a divide by zero or tries to access a protected region of memory, or if a device fails?
  - **Resource allocation**: Processes may compete for resources such as the CPU, memory, and I/O devices.
  - **Accounting**: *e.g.* how much disk space is this or that user using? how much network bandwidth are we using?
  - **Protection and Security**: The owners of information stored in a multi-user or networked computer system may want to control use of that information, and concurrent processes should not interfere with each other

OS Structure with Services



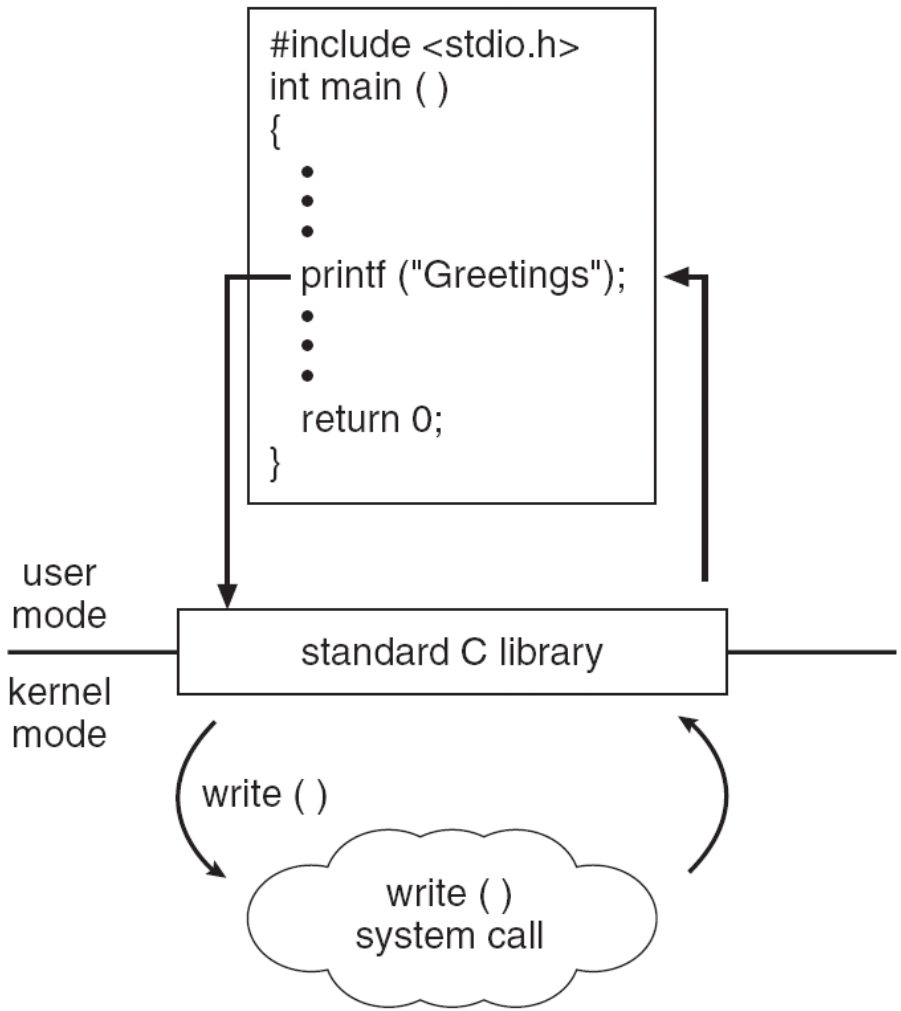
System Calls

- Programming interface to the services provided by the OS (*e.g.* open file, read file, *etc.*)
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call.
- Three most common APIs are Win32 API for Windows, POSIX API for UNIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
- So why use APIs in user processes rather than system calls directly?
  - Since system calls result in execution of privileged kernel code, and since it would be crazy to let the user process switch the CPU to privileged mode, we must make use of the low-level hardware trap instruction, which is cumbersome for user-land programmers.
  - The user process runs the trap instruction, which will switch CPU to privileged mode and jump to a kernel pre-defined address of a generic system call function, hence the transition is controlled by the kernel.
  - Also, APIs can allow for backward compatibility if system calls change with the release of a OSS

System calls provided by Windows and Linux

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

An example of a System Call



System calls for file operations

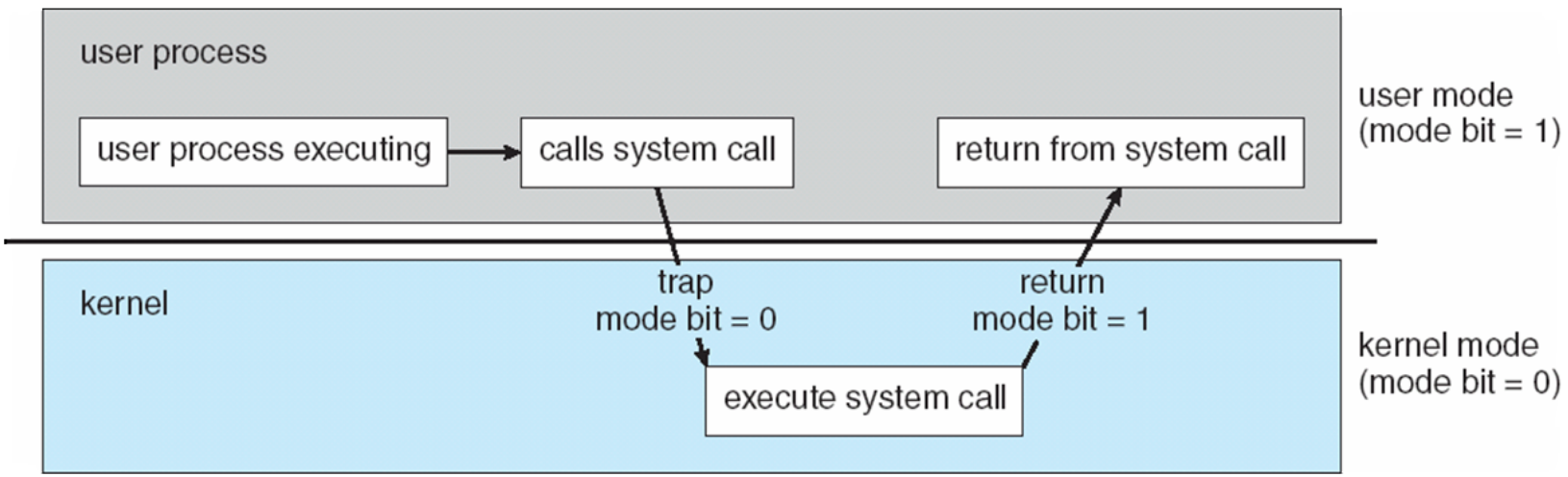
Have the following operations for files:

- **open** Register the file with the operating system. Must be called before any operation on the file. Returns an integer called the file descriptor - an index into the list of open files maintained by the OS.
- **read** Read data from the file. Returns number of bytes read or 0 for end of file.
- **write** Write data to a file. Returns number of bytes written.
- **close** De-registers the file with the operating system. No further operations on the file are possible.

These system calls return a negative number on error. Can use `perror`-function to display error.

Trapping to the Kernel

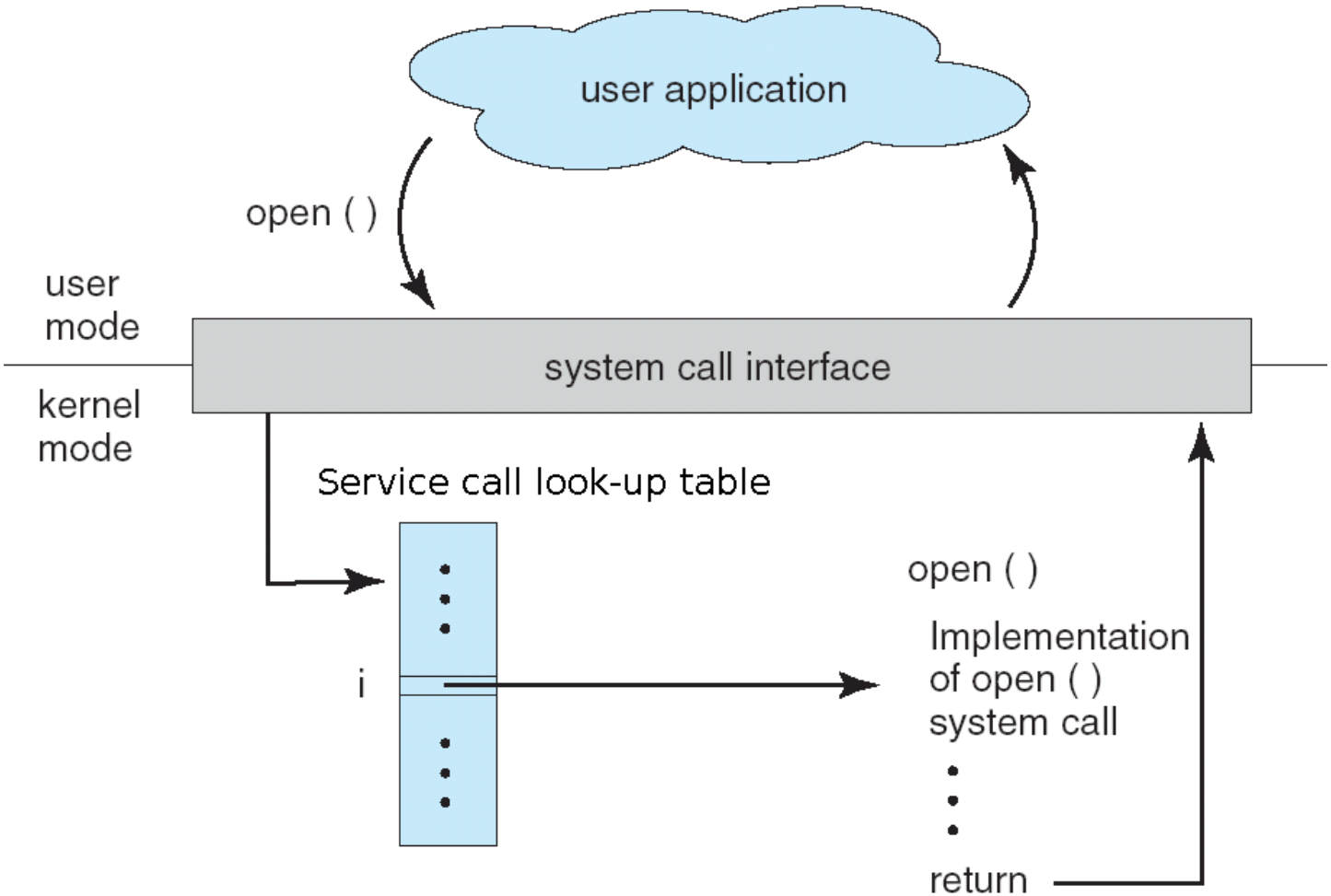
- The user process calls the system call wrapper function from the standard C library
- The wrapper function issues a low-level *trap* instruction (in assembly) to switch from user mode to kernel mode



Trapping to the Kernel

- To get around the problem that no call can directly be made from user space to a specific function in kernel space:
  - Before issuing the trap instruction, an index is stored in a well known location (*e.g.* CPU register, the stack, *etc.*).
  - Then, once switched into kernel space, the index is used to look up the desired kernel service function, which is then called.
- Some function calls may take arguments, which may be passed as pointers to structures via registers.

Trapping to the Kernel



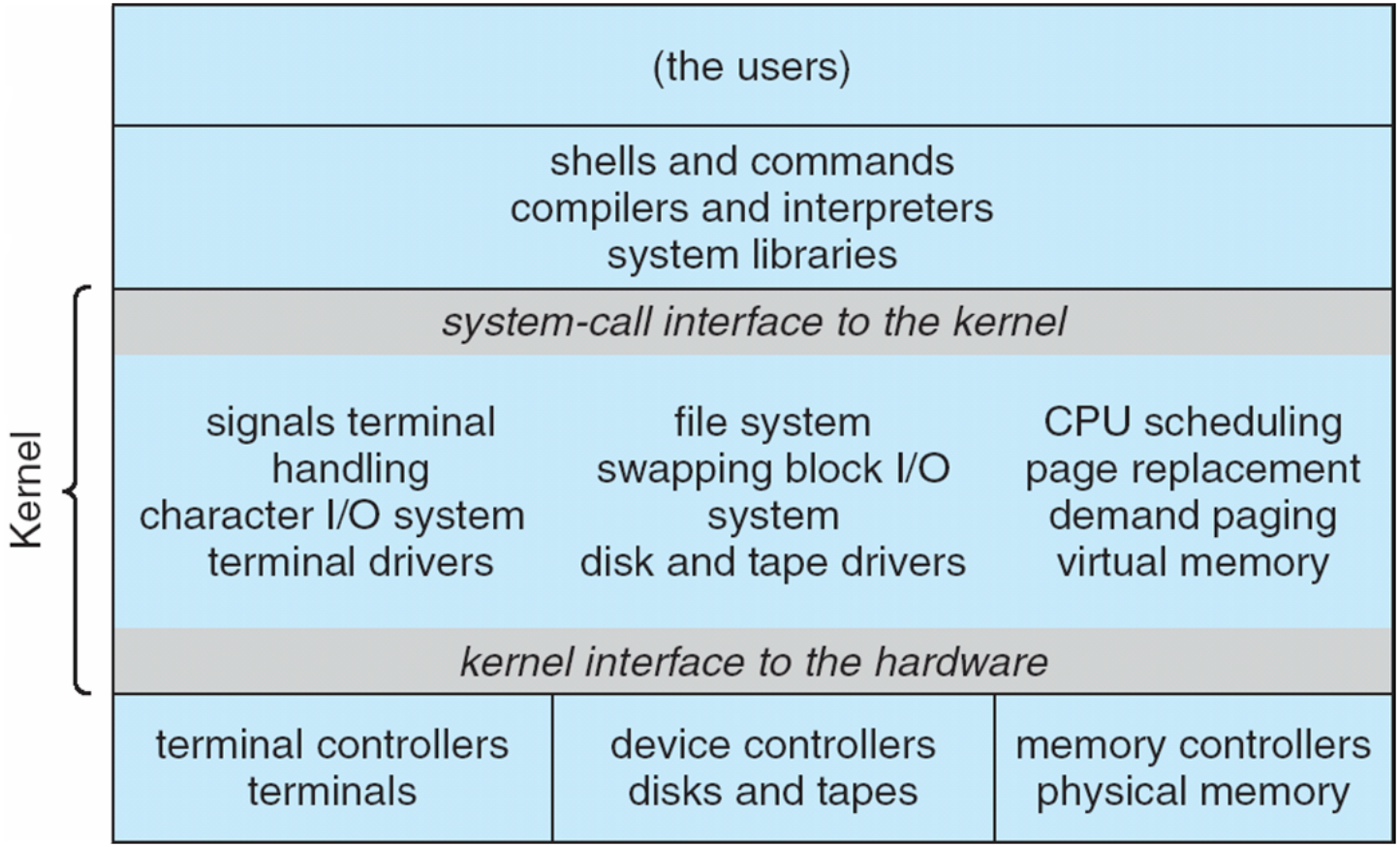
OS Architecture

Traditional UNIX

UNIX - one big kernel

- Consists of everything below the system-call interface and above the physical hardware
- Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level
- Limited to hardware support compiled into the kernel.

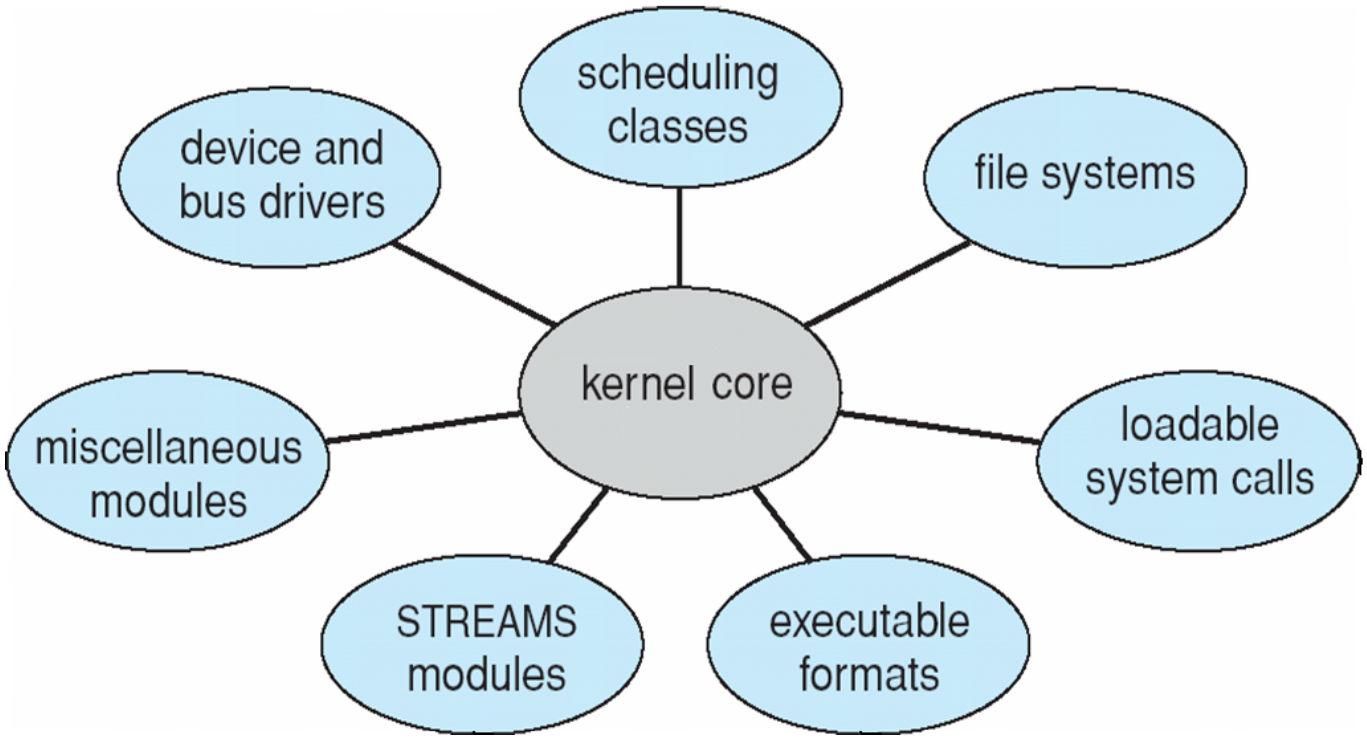
Traditional UNIX



Modular Kernel

- Most modern operating systems implement kernel modules
  - Uses object-oriented–like approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel, so you could download a new device driver for your OS and load it at run-time, or perhaps when a device is plugged in
- Overall, similar to layered architecture but with more flexibility, since all require drivers or kernel functionality need not be compiled into the kernel binary.
- Note that the separation of the modules is still only logical, since all kernel code (including dynamically loaded modules) runs in the same privileged address space (a design now referred to as monolithic), so I could write a module that wipes out the operating system no problem.
  - This leads to the benefits of micro-kernel architecture, which we will look at soon

Modular Kernel



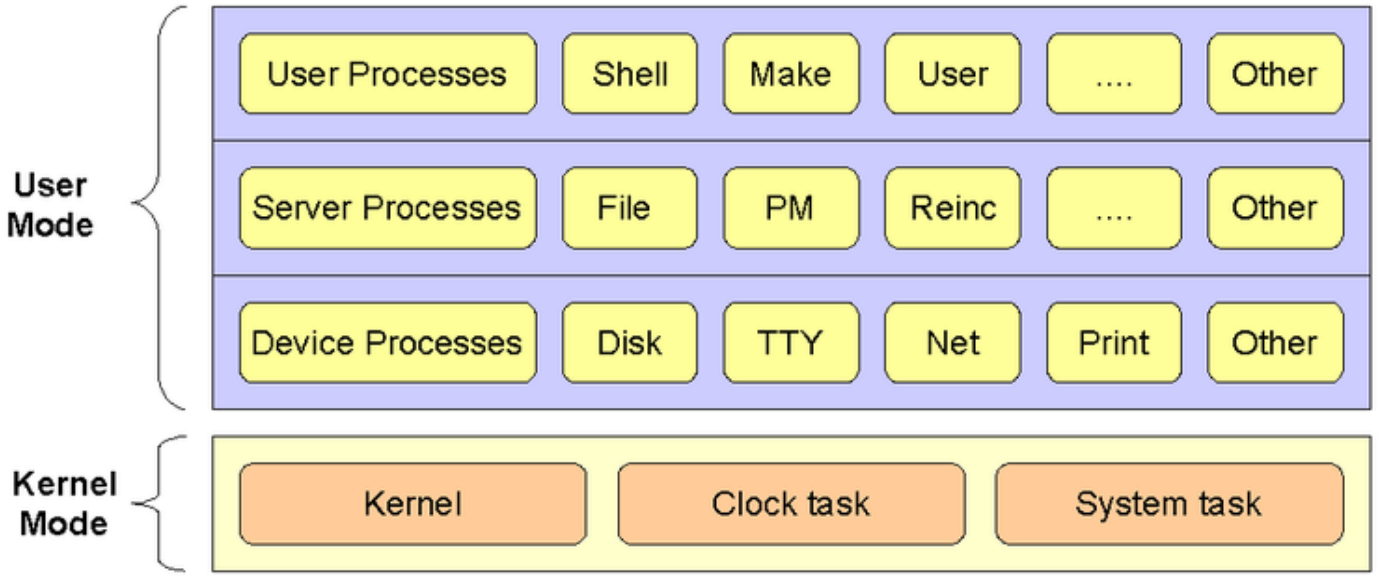
Microkernel

- Moves as much as possible from the kernel into less privileged "user" space (*e.g.* file system, device drivers, *etc.*)
- Communication takes place between user modules using message passing
  - The device driver for, say, a hard disk device can run all logic in user space (*e.g.* decided when to switch on and off the motor, queuing which sectors to read next, *etc.*)
  - But when it needs to talk directly to hardware using privileged I/O port instructions, it must pass a message requesting such to the kernel.

Microkernel

- Benefits:
  - Easier to develop microkernel extensions
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode) - if a device driver fails, it can be re-loaded
  - More secure, since kernel is less-complex and therefore less likely to have security holes.
  - The system can recover from a failed device driver, which would usually cause “a blue screen of death” in Windows or a “kernel panic” in linux.
- Drawbacks:
  - Performance overhead of user space to kernel space communication
- The Minix OS and L3/L4 are examples of microkernel architecture

Microkernel: MINIX



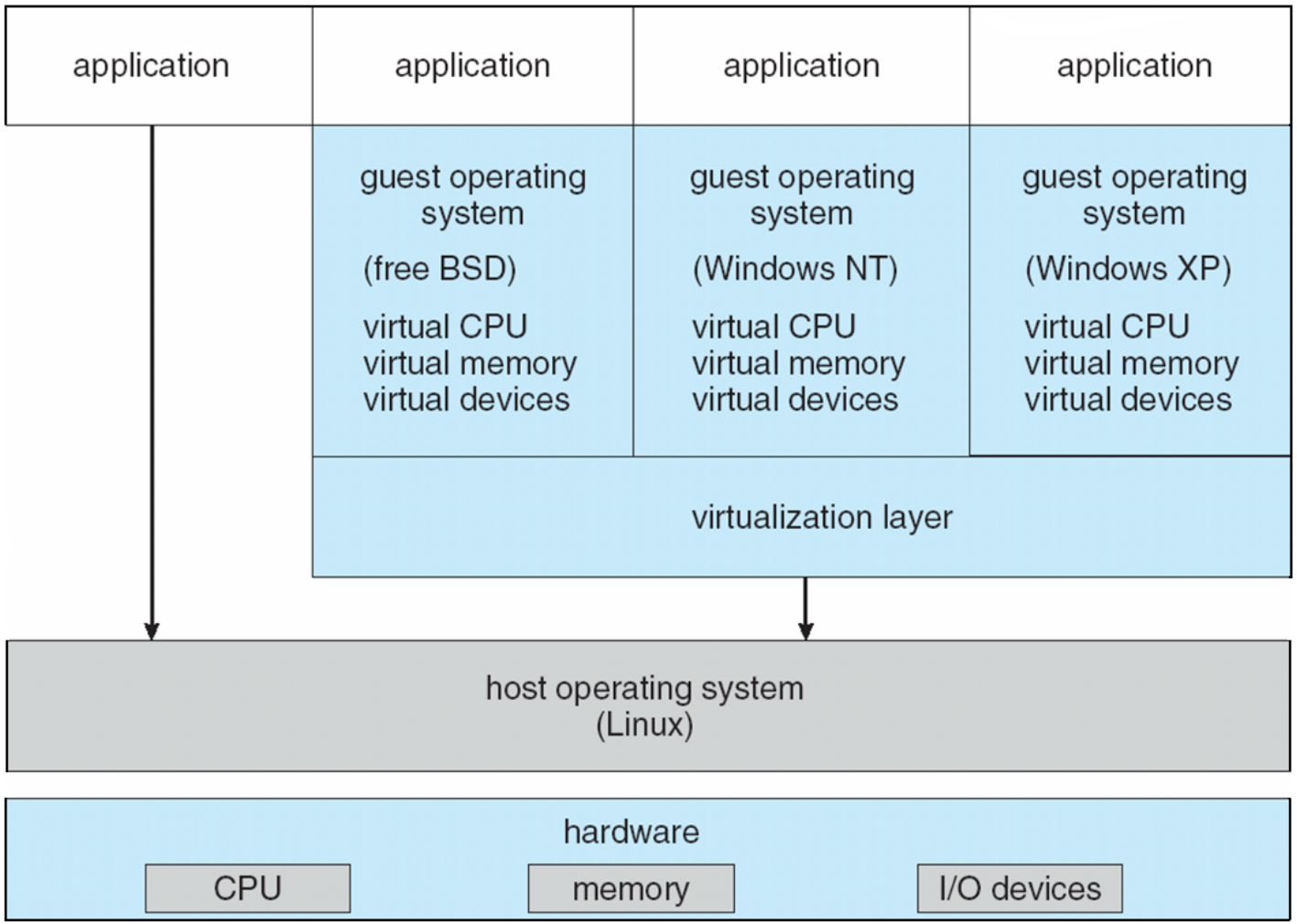
The MINIX 3 Microkernel Architecture

Virtual Machines

Virtual Machines

- A virtual machine allows to run one operating system (the guest) on another operating system (the host)
- A virtual machine provides an interface identical to the underlying bare hardware
- The operating system host creates the illusion that a process has its own processor and (virtual memory)
- Each guest is provided with a (virtual) copy of underlying computer, so it is possible to install, say, Windows 10 as a guest operating system on Linux.

VM Architecture



Virtual Machines: History and Benefits

- First appeared commercially in IBM mainframes in 1972
- Fundamentally, multiple execution environments (different operating systems) can share the same hardware
- Protected from one another, so no interference
  - Some sharing of files can be permitted, controlled
  - Communicate with one another other and with other physical systems via networking
- Useful for development, testing, especially OS development, where it is trivial to revert an accidentally destroyed OS back to a previous stable snapshot.

Virtual Machines: History and Benefits

- Consolidation of many low-resource use systems onto fewer busier systems
- "Open Virtualization Format" (OVF): standard format of virtual machines, allows a VM to run within many different virtual machine (host) platforms.
- Not to be confused with emulation, where guest instructions are run within a process that pretends to be the CPU (*e.g.* Bochs and QEMU). In virtualisation, the goal is to run guest instructions directly on the host CPU, meaning that the guest OS must run on the CPU architecture of the host.

Para-virtualisation

- Presents guest with system similar but not identical to hardware (*e.g.* Xen Hypervisor)
- Guest OS must be modified to run on paravirtualized 'hardware'
  - For example, the kernel is recompiled with all code that uses privileged instructions replaced by hooks into the virtualisation layer
  - After an OS has been successfully modified, para-virtualisation is very efficient, and is often used for providing low-cost rented Internet servers (*e.g.* Amazon EC2, Rackspace)

VMWare Architecture

- VMWare implements full virtualisation, such that guest operating systems do not require modification to run upon the virtualised machine.
- The virtual machine and guest operating system run as a user-mode process on the host operating system

VMWare Architecture

- As such, the virtual machine must get around some tricky problems to convince the guest operating system that it is running in privileged CPU mode when in fact it is not.
  - Consider a scenario where a process of the guest operating system raises a divide-by-zero error.
  - Without special intervention, this would cause the host operating system immediately to halt the virtual machine process rather than the just offending process of the guest OS.
  - So VMWare must look out for troublesome instructions and replace them at run-time with alternatives that achieve the same effect within user space, albeit with less efficiency
  - But since usually these instructions occur only occasionally, many instructions of the guest operating system can run unmodified on the host CPU.

Linux kernel programming

Structure of kernel

Simplified structure of kernel:

```
initialise data structures at boot time;
while (true) {
    while (timer not gone off) {
        assign CPU to suitable process;
        execute process;
    }
    select next suitable process;
}
```

Kernel programming

Kernel has access to all resources
Kernel programs not subject to any constraints for memory acces or hardware access
$\Rightarrow$ faulty kernel programs can cause system crash

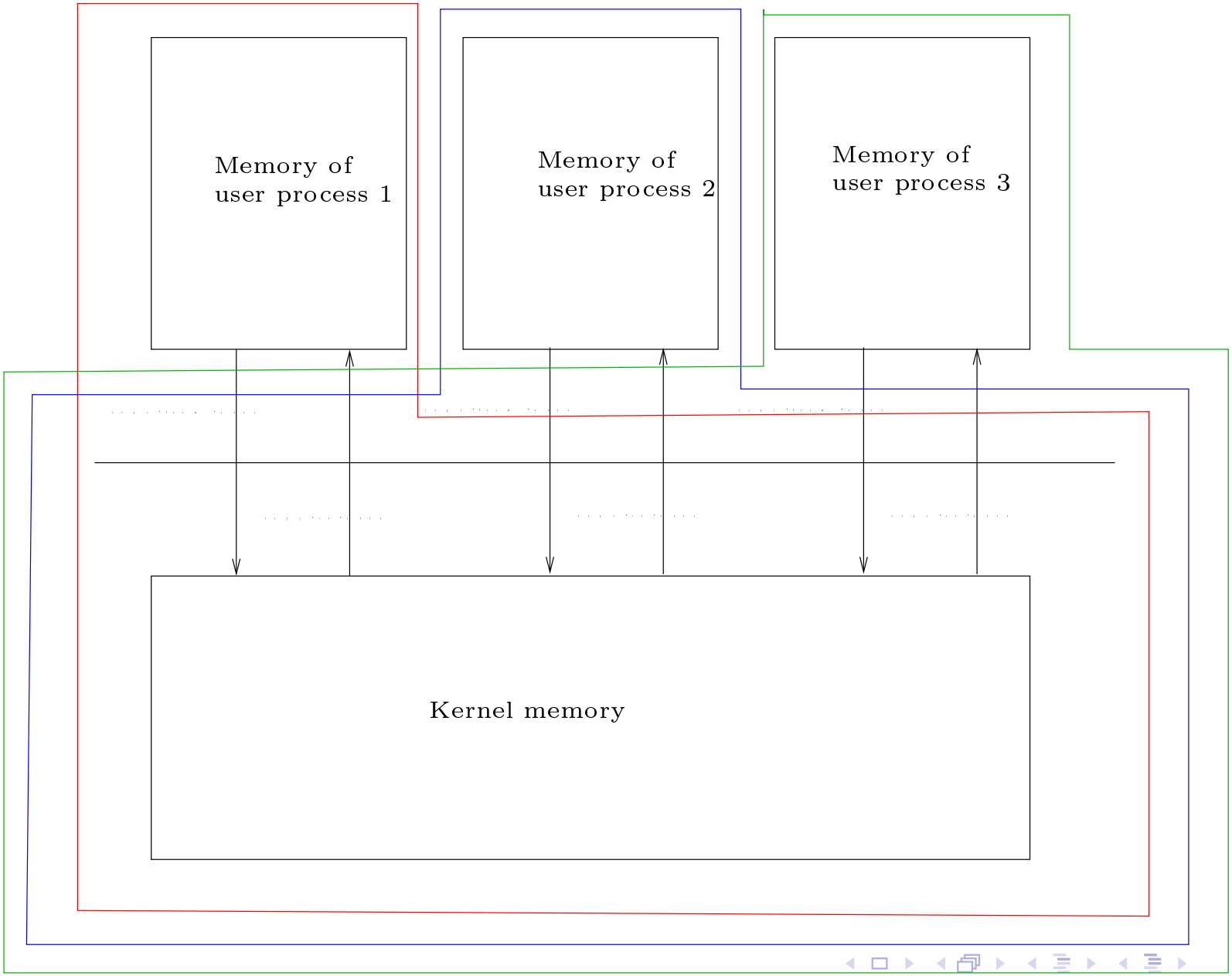
Interaction between kernel and user programs

Kernel provides its functions only via special functions, called system calls

standard C-library provides them

Have strict separation of kernel data and data for user programs

$\Rightarrow$ need explicit copying between user program and kernel (`copy_to_user()`, `copy_from_user()`)


Memory of
user process 1
Memory of
user process 2
Memory of
user process 3
Kernel memory

In addition, have interrupts:
kernel asks HW to perform certain action
HW sends interrupt to kernel which performs desired action
interrupts must be processed quickly
$\Rightarrow$ any code called from interrupts must not sleep

Linux kernel modes

Structure of kernel gives rise to two main modes for kernel code:

- process context: kernel code working for user programs by executing a system call
- interrupt context: kernel code handling an interrupt (eg by a device)

have access to user data only in process context
Any code running in process context may be pre-empted at any time by an interrupt
Interrupts have priority levels
Interrupt of lower priority are pre-empted by interrupts of higher priority

Kernel modules

can add code to running kernel
useful for providing device drivers which are required only if hardware present
`modprobe` inserts module into running kernel
`rmmod` removes module from running kernel (if unused)
`lsmod` lists currently running modules

Concurrency issues in the kernel

Correct handling concurrency in the kernel important:
Manipulation of data structures which are shared between

- code running in process mode and code running in interrupt mode
- code running in interrupt mode

must happen only within critical regions
In multi-processor system even manipulation of data structures shared between code running in process context must happen only within critical sections

Achieving mutual exclusion

Two ways:

- Semaphores/Mutex: when entering critical section fails, current process is put to sleep until critical region is available
  $\Rightarrow$ only usable if all critical regions are in process context
  Functions: `DEFINE_MUTEX()`, `mutex_lock()`, `mutex_unlock()`
- Spinlocks: processor tries repeatedly to enter critical section
  Usable anywhere
  Disadvantage: Have busy waiting
  Functions: `spin_lock_init()`, `spin_lock()`, `spin_unlock()`

Use of semaphores

Have two kinds of semaphores:

- Normal semaphores
- Read-Write semaphores: useful if some critical regions only read shared data structures, and this happens often

Programming data transfer between userspace and kernel

Linux maintains a directory called `proc` as interface between user space and kernel
Files in this directory do not exist on disk
Read-and write-operations on these files translated into kernel operations, together with data transfer between user space and kernel
Useful mechanism for information exchange between kernel and user space

A tour of the Linux kernel

Major parts of the kernel:

- Device drivers: in the subdirectory `drivers`, sorted according to category
- file systems: in the subdirectory `fs`
- scheduling and process management: in the subdirectory `kernel`
- memory management: in the subdirectory `mm`
- networking code: in the subdirectory `net`
- architecture specific low-level code (including assembly code): in the subdirectory `arch`
- include-files: in the subdirectory `include`

Summary

- Operating System as Resource manager, in particular for hardware
- OS has access to all resources
- Have API to interact with OS from programs (system calls)
- Hve monolithic kernel (one big program) (eg Unix, Linux) or microkernels (only direct interaction with HW part of kernel)
- In Linux, can modify kernel via kernel modules - separate programs which are still part of the kernel
- Virtual machines make it possible to run one operating system on top of another one. Need to be careful with privileged OS instructions.