Introduction: OS Function And Structure

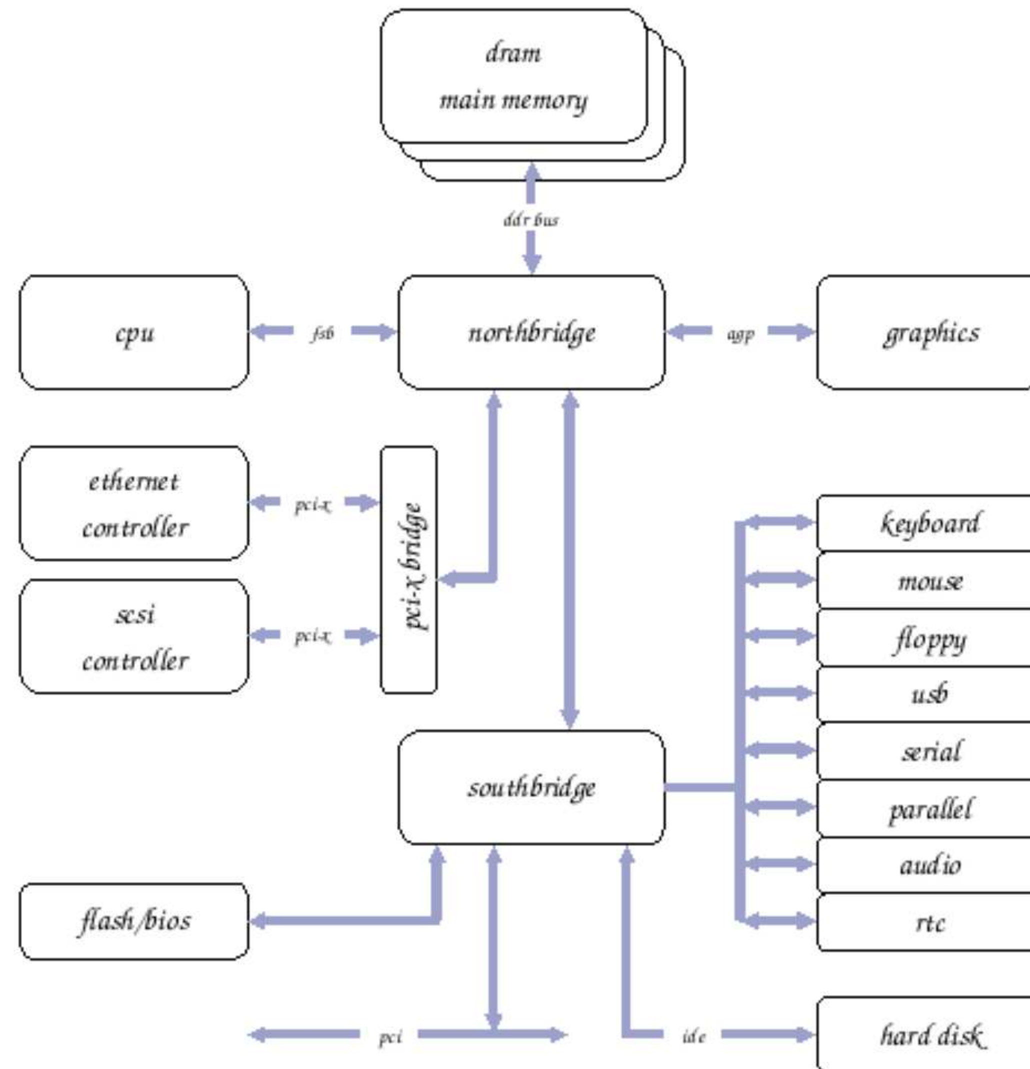
Operating Systems

Oleg Goldshmidt

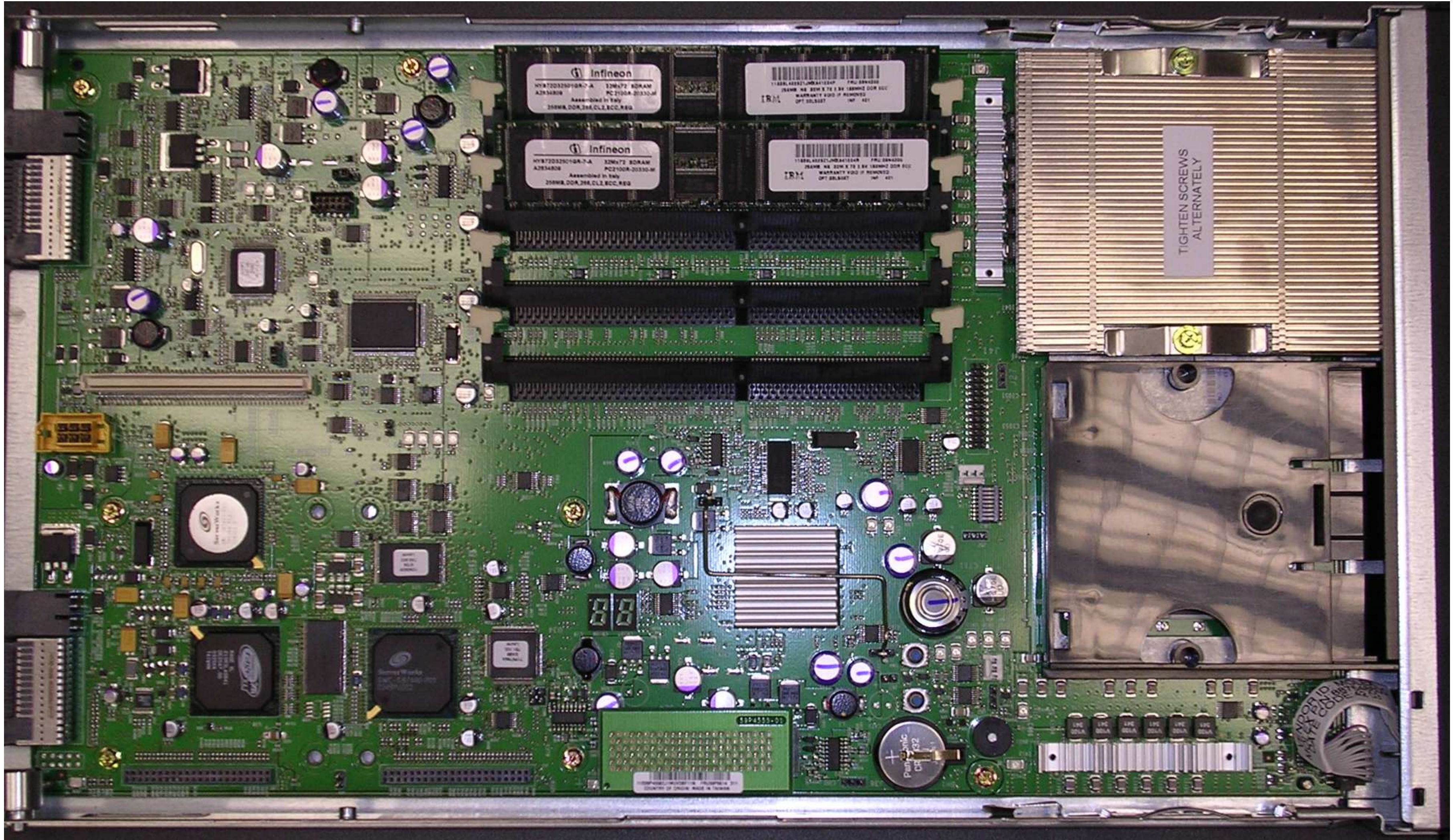
ogoldshmidt@computer.org

Lecture 1

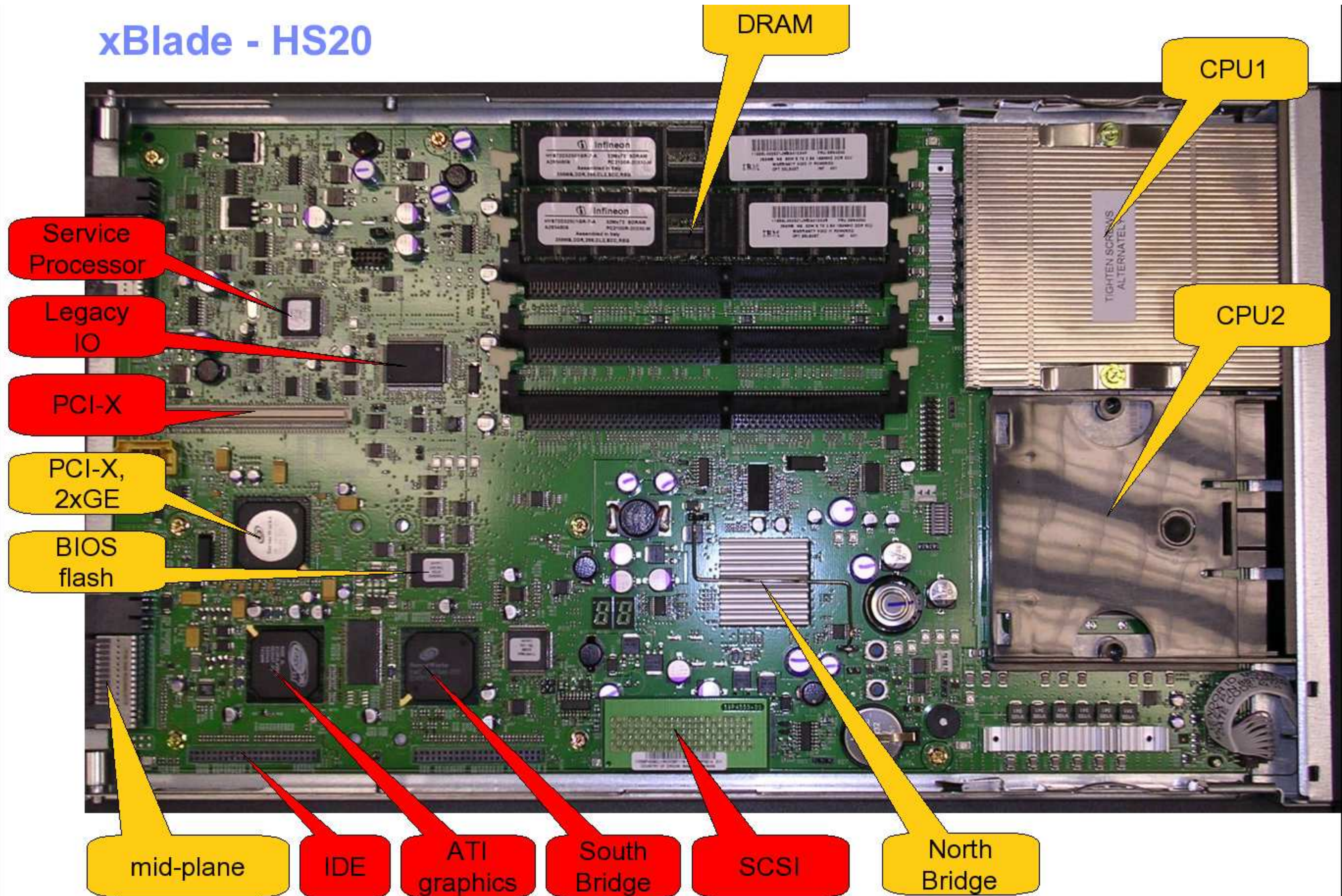
What Is An (Intel) Computer?



An Example of a Computer



Components of a Computer



How Computers Work: Execution

- Von Neumann's "Stored Program" model
  - both programs and data are stored in memory
  - `fetch` — `decode` — `execute` — `store`
  - enables programmability, viruses, etc.
- pipelining
  - while some circuits are executing an instruction, others may be decoding the next one, and yet others may be fetching another one
  - a low-level manifestation of parallelism
  - need to be very careful — much of the course will deal with synchronization problems
- are there other models?

How Computers Work: Peripherals

- functions of peripheral devices
  - communicate with users (displays, consoles, keyboards, mice, audio, video, printers, etc.)
  - serve as external storage devices (disks, tapes, CD, DVD, etc.)
  - enable communications with other computers or peripherals (network, serial, etc.)
- controlled by controllers/adapters
- data flows between peripherals and main memory (and back):
  - programmed I/O — to/from I/O registers
  - DMA — direct memory access
- peripheral data: commands, parameters, data proper

How Computers Work: Interrupts (I)

- what happens in the keyboard when you press the `a` key?
  - electric current flows in a wire
  - the keyboard controller polls the wires to detect pressed (and released) keys
  - the character corresponding to the key pressed is stored for further retrieval
  - the keyboard controller “taps the shoulder” of the main CPU — raises an interrupt

How Computers Work: Interrupts (II)

- what happens in the main CPU upon an interrupt?
  - the current state (registers, program counter, etc. — details later) is stored (on a stack)
  - an `interrupt handler` is invoked and executed (NB: software!)
    - retrieves the character `a` from where it is stored
    - checks whether `Shift` has been pressed etc.
    - drives the console, the display, etc. to show `a` to the user (other interrupts may be involved)
  - the prior state is restored from the stack, and execution resumes from the point where it was interrupted

How Computers Work: Interrupts (III)

- interrupts signal events
- interrupts may be caused by external events or by a step in the program operation — hardware vs. software interrupts
  - software interrupts: traps, system calls
  - hardware interrupts: human action, the end of an I/O operation, exceptions and errors, timers
- both types cause the computer to perform the following steps:
  - save the current state (program counter, registers, etc.)
  - jump to a location dependent on the specific interrupt
  - NB: hardware saves only the hardware state!

Direct Memory Access

- interrupt-driven operation is fine for slow devices such as terminals (humans are slow)
- not suitable for fast devices, e.g. modern storage devices
  - the CPU would have to save state, execute interrupt handler, restore state far too often
- solution: set up buffers, pointers, counters once, transfer an entire block of data to/from memory using a separate “DMA controller”, without involving the main CPU
  - inform the main CPU (via an interrupt!) when the transfer is done
  - 1 interrupt/operation rather than 1 interrupt/byte

Memory Hierarchy and Storage

- von Neumann: both programs and data are stored in memory — the most important resource of the computer (together with the CPU, of course)
- fast memory is expensive, hence scarce, slow memory is cheap, and thus abundant
- memory hierarchy:
  - registers
  - cache
  - RAM
  - storage (disks, tapes, etc.)
- non-volatile memory — ROM, flash
  - ROM is often used for boot
  - flash is also used as disk replacement

Multiprogramming and Protection

- many programs may be executed at once (through time-sharing)
- computer resources are shared between concurrent programs
- an incorrect (or malicious) program may not cause other programs to execute incorrectly
- hardware support for (at least) 2 execution modes
  - user mode
  - kernel (a.k.a. system, privileged) mode
- memory, CPU, I/O protection — all need to be protected

Does This Look Simple?

- not really, especially if we start thinking about performance and price considerations
- early computers were expensive, readers and printers were slow compared to the CPU
  - allow card reading and printing to run concurrently
- some programs read and print intermittently — CPU sits idle
  - load other jobs while the original one does I/O, let it run for a while, then restart the original one
  - time-share different users
- personal computers: unsophisticated users, diverse interfaces, a multitude of commodity devices
- multiprocessor computers, integrated controllers, etc.

Bare Metal Is Too Complex (I)

- the drive to make computers faster and cheaper makes them more complex
- hardware and architectural complexity must be hidden from the users and from application programmers
  - recall how complex interrupt handling is
  - it will help if someone will move data and programs arpound the complicated memory hierarchy in a smart way
  - your data are usually not stored as contiguous blobs on disks, but the directory and file structure can be as neat as we make it
  - the Net is very messy behind your browsers, but we do navigate it rather easily

Bare Metal Is Too Complex (II)

- users and programmers must see a more abstract picture of the machine than the machine designers/builders
- the various resources must be managed well and allocated to competing programs in a controlled manner, independently of the users' qualifications and without undue burden on the application writers
- various operations need to be automated

What Is An Operating System?

- a program that acts as an intermediary between a user program and the computer hardware
- OS functions:
  - makes programming applications easier — the programmer is using an ”extended machine” implementing high level abstractions quite remote from hardware
  - makes operation easier — the machine user is presented with tools enabling automation of most of the machine operating procedures an mostly unmanned operation
  - manages the machine resources and ensures efficient usage of hardware

The Need for Operating Systems

- can we do without operating systems?
  - yes, sure — just let the applications talk to the hardware directly
  - sometimes (often, actually) it is the case — in special-purpose embedded systems
- not practical in a general purpose computer system
  - applications will be tied to a particular set of HW devices, need to be ported for each new model of each type of device
  - it will be practically impossible to run multiple applications on a single machine ("multiprogramming")
    - each application will behave as if it owns the whole system, and chaos will ensue

High Level Abstraction

- the machine and system architecture at machine language level are primitive and hard to program
  - to do a simple operation like printing a line of text or reading a disk sector you have to: (a) set up some controller registers in a precisely defined sequence, (b) prepare the data in a form suitable for the device, (c) start some mechanical operation and wait for some conditions, (d) do the data transmission and wait for the operation to end (an interrupt)
- to be productive programmers must be shielded from all those details (similar to what high level languages do)
- application programmers need a convenient interface
- hiding the low level details is very important for portability

Resource Management

- a system owner wants his equipment to be utilized
- a system is composed of many elements — processors, memories, peripherals etc.
- as in factory with many machines we would like the system to “produce” as much as possible
- as an application is not using all the system resources (e.g., when waiting for I/O the processor is not used) we could “schedule” several applications to run on the system
- coordinating resource usage by different programs is done by the operating system
- coordination is necessary both for conflicting as well as for cooperating programs (sharing)

Automated Operation

- a function often overlooked
- a system owner/user would like to have unmanned system operation
- in large organizations efficient operation requires exception handling and system supervision to be done by a small number of people for a large number of machines
- system management an essential ingredient in every component including OS
- setup, tuning, boot, reconfiguration — automatic
- scheduled tasks (e.g., backup) — automatic
- OS has to be designed for management

Computer System Architecture

"userspace"

applications

(office, email, web, the normal stuff we use and develop)

operating system

process management, memory management,
file systems, I/O, and other interesting stuff

device drivers

hardware

(CPU, memory, controllers, disks, network, KVM, etc.)

Computer System Interfaces

"userspace"

applications

(office, email, web, the normal stuff we use and develop)

ABI (system calls etc.)

operating system

process management, memory management, file systems, I/O, and other interesting stuff

device drivers

ISA and other hardware interfaces

hardware

(CPU, memory, controllers, disks, network, KVM, etc.)

OS Services

- program services
  - program execution
  - data I/O
  - persistent storage functions (files)
  - communications (with other programs)
  - exception handling (faults, errors, etc.)
- operating services
  - resource allocation
  - activity logging (accounting, auditing, tracing, quota enforcement, etc.)
  - protection and access control

How OS Services Are Provided

- system calls — used for for most of the program services
  - process control
  - file manipulation
  - device operation
  - information maintenance
  - communications
- system programs (system utilities)
  - file, status, communication, and other utilities
  - editors
  - programming language environments and libraries
  - program manipulation (compilers, linkers, loaders)
  - command interpreters (shells)

OS Parts and Structure

- major functions (components)
  - process management
  - memory management
  - storage management
  - I/O management
  - file system
  - protection and access control
  - networking
  - command interpreter
- structure
  - monolithic
  - microkernel
  - virtual

Process Management

- a program in execution is a process
- a process is a “dynamic” entity — it is the program execution, not the program instructions
- a process can create other processes
- a process has associated resources (memory, files, devices etc.) that are given to him either when it is created or upon request
- OS in charge of:
  - process creation and deletion (kill)
  - suspension/resumption, scheduling
  - IPC and synchronization
- threads

Memory Management

- to execute a program the machine must have it loaded in memory and must have memory for its data
- except in very early systems we keep several programs in memory
- allocating an freeing main memory is done by the OS
- OS has to keep track of allocated memory
- programs and data are located by the CPU through absolute addresses
- allocating main memory is a hard problem
- virtual memory removes much of the complexity of memory allocation

Storage Management

- Main storage is not sufficient to hold all programs and data: expensive and volatile
- disks are used used to hold what is “spilled” over from main memory (temporary) or what has to be persistent (files, databases)
- programs are stored on disk and use disk as source and destination for data
- OS does:
  - free space management
  - disk and space allocation
  - content organizations in structures with fast access and resiliency to power failures
  - disk caching and swapping

Device Operation

- one of the purposes of OS is to hide diverse hardware peculiarities — it is important that even the OS sees the devices in a uniform way
- this is accomplished by having the I/O accessed through a common set of interfaces (APIs) even within the system (I/O subsystem in UNIX, I/O manager in NT)
- devices are manipulated by `device drivers` that "look the same" to the rest of the OS and act differently on hardware — numerous, expensive
- OS has to offer (common) services to the driver builder, such as buffer management, memory mapping (user space/system space/I/O space)
- layered driver structure (block/SCSI/specific disks)

File Systems

- a file is viewed as an indiscriminate sequence of bytes (some systems add some structure — e.g., records on OS390 and AS/400)
- hierarchical organization (directories)
- mask the actual data organization on media
- OS must provide:
  - creation and deletion of files and directories
  - file access (open, close, read, write, seek)
  - file access control and coordination
  - file mapping to media
  - utilities for backup/restore, export/import
- general abstraction in some OS (UNIX): “everything is a file” — e.g., special “device files”

Protection

- if a system holds several programs in memory they must be protected against accidental or malicious misuse
- memory protection ensures that a process can access only the memory areas allocated to it
- I/O devices allocated to one process should be protected against use by other processes; in general users don't do I/O by themselves
- a process should be prevented to gain control of the CPU at the expense of others
- access to files (data, programs etc.) is controlled
- extension: correct use of interface between subsystems

Networking

- with very few exceptions today machines are interconnected through various communication networks (LAN, WAN) and various protocols
- OS should provide different levels of hiding the network complexity
- present users with simple paradigms like remote file access, remote program execution, etc.
- provide programmers with a communication mechanism suited to the application developed (protocol, types of connection etc.)
- provide administrators with means to monitor and actively manage networks (connections, routers etc.)

Command Interpreters

- an essential component in a system
- in most modern systems it is not part of the OS (the "kernel") but an application through which the user communicates with the system
- command interpreter accepts commands:
  - as interactive control statements or scripts
  - as a graphical interface (GUI) set of actions (e.g., mouse movements, "clicks", etc.)
- command languages are powerful programming languages by themselves
- allow direct invocation of system utilities for process management, file system manipulation, protection, networking etc.

System Calls

- the programming interface to the OS services
- historically in assembly, in most modern OS they are available directly from HL programming languages, often wrapped into library calls
- system calls are frequently invoked through software interrupts and parameters are passed through registers and/or through stack
- system calls are used for:
  - process control
  - file manipulation
  - device control
  - information and configuration manipulation
  - communication

How To Use System Calls (I)

On UNIX (and Linux) systems:

```
#include <sys/types.h>
#include <unistd.h>
```

will give you the necessary (in ANSI C) data types and function prototypes, such as

```
int close (int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf,
              size_t count);
pid_t getpid(void);
pid_t getppid(void);
pid_t fork(void);
```

When in doubt, use Section 2 of the “man pages”, e.g.,

```
man 2 read
```

How To Use System Calls (II)

Often it is better to use library facilities rather than “naked” system calls, e.g., after

```
#include <stdio.h>
```

standard C I/O library facilities such as

```
FILE *fopen(const char *path,
            const char *mode);
size_t fread(void *ptr, size_t size,
             size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
int fflush(FILE *stream);
int fclose(FILE *stream);
```

become available. Cf. `man 3 fopen` etc.

System Calls vs. Library Calls (I)

- what is the difference?
- from the user's point of view, not much:
  - both syscalls and library calls are normal C functions
  - both can be used anywhere in your code
  - both exist to provide services to application programmers
- realize that you can replace library functions as desired, but not system calls

System Calls vs. Library Calls (II)

- example: consider memory allocation — no single technique is optimal for every program.
  - behind the scenes the system call `sbrk(2)` is normally used
    - actually, `sbrk` may be a C library wrapper around `brk(2)`, but that's an implementation detail
  - however, we usually call `malloc(3)` from `libc` (that calls `sbrk(2)` as necessary)
  - if necessary, we can write a custom allocator that will likely use `sbrk(2)`
- key distinction: mechanism vs. policy — (general purpose) OS usually implement mechanism, not policy

System Calls vs. Library Calls (III)

- system calls usually provide a minimal interface, while libraries provide more elaborate functionality
- process control system calls (`fork(2)`, `exec(2)`, `wait(2)`) are usually used directly
- library routines often simplify certain common cases, e.g., `system(3)`, `popen(3)`
- throughout the course we will use the word `function` for both system and library calls, except when the distinction is necessary

What Syscalls Does This Program Use?

```
# strace cat foo
execve("/bin/cat", ["cat", "foo"],
       [/* 64 vars */])                      = 0
...
open("/lib/tls/libc.so.6", O_RDONLY)         = 3
read(3, "\177ELF\1\1\1\0\0\0"..., 512)   = 512
...
close(3)
...
open("foo", O_RDONLY|O_LARGEFILE)            = 3
...
read(3, "foo\n", 4096)                       = 4
write(1, "foo\n", 4096)                      = 4
read(3, "", 4096)                            = 0
close(3)                                     = 0
```

Relevant Standards: ANSI C

- a practically necessary condition to make your code portable between "platforms" (including different OS)
- the OS kernels are exceptions (sometimes), e.g., Linux is `gcc`-dependent
- but any normal OS "distribution" will provide the user with ANSI C facilities (libraries, headers, etc.)
- compilers are crucial — most are approximations to the standard

Relevant Standards: POSIX

- defines the services an OS must provide
- obviously important to assist application portability
- defines interface, not implementation — no distinction between system calls and library calls, both called functions
- OS are usually some approximations of POSIX (as compilers are for ANSI C), e.g. Linux is “mostly compliant”, Windows is not compliant at all
- there are other standards, and often “implementation families” (such as System V and BSD in the UNIX world) are regarded as de facto standards
- the UNIX man pages normally have a “CONFORMING TO” section