Computer Science 32
Object-Oriented Design and Implementation (in C++ on Linux)

- Pre-requisite: CS 24
  - So already know much C++ including object-based fundamentals: classes and ADTs
  - Also familiar with at least some Linux usage
- Designed for 2nd year CS majors
  - Others welcome if pre-req. met and space permits
- Primary goal: ready for CS 48 & upper div. CS

Course structure

- Cover all of Reader + key chapters of Textbook
- *Mixture* of OOP/C++ and OS topics (not sequentially)

OOP/C++	**OS/Linux**
Intro. OOP and OO design	Intro. OS and Unix
Classes – basics	Processes
Classes – advanced	Tools and pgm. building
Inheritance, polymorphism	Memory concepts
Templates and STL	Libraries

Plus four exams: one after every six lectures (or so)

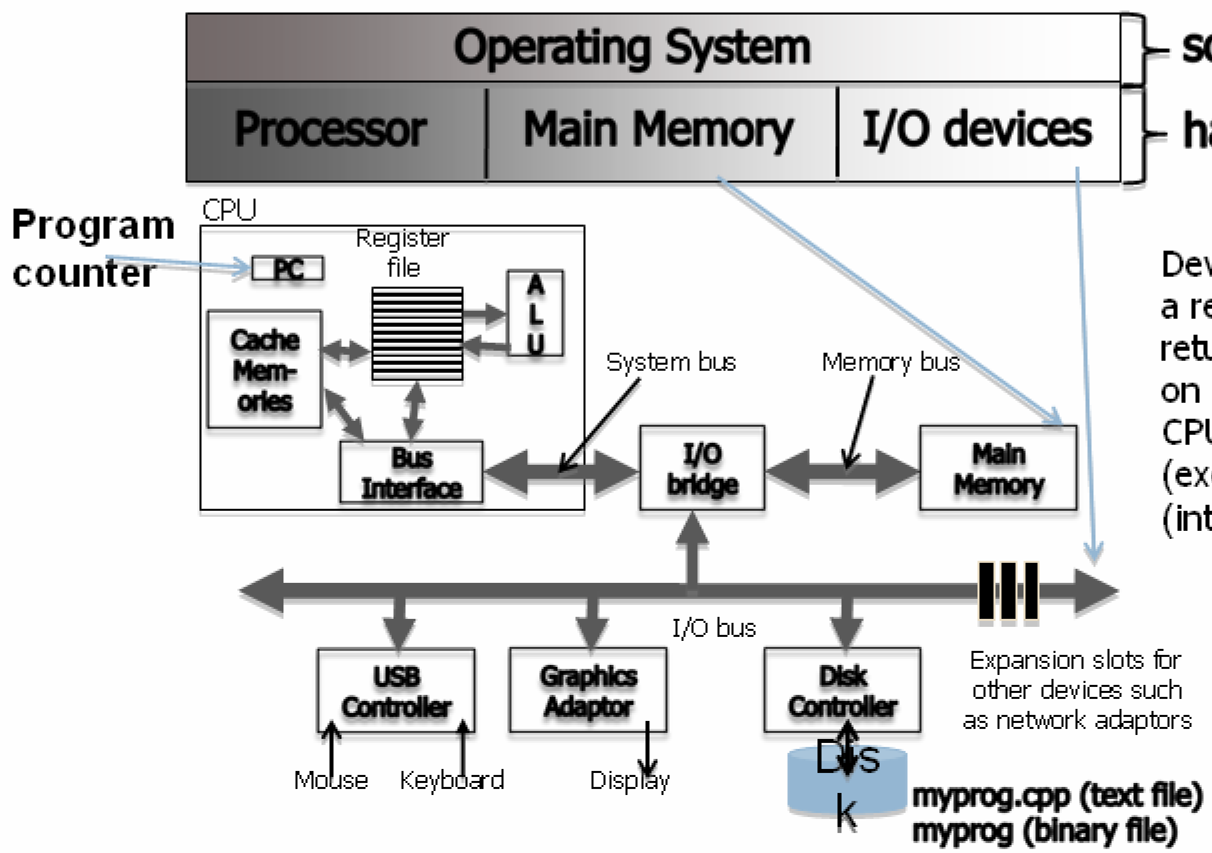
Requirements

- 84 percent of grade: *best three out of four* exams (each counts 28 percent)
  - Probably Wednesday, April 17
  - Probably Friday, May 3
  - Probably Monday, May 20
  - Probably Friday, June 7 (No final exam this quarter)
- 16 percent of grade: labs and related work
- Students are responsible for monitoring changes to course web pages too
- Questions?

To Do – first *week*

- Readings #1 and #2 (from Reader)
  - In general, read ahead of lectures
- Attend *your* assigned lab section *next* week
  - First week's labs were cancelled
- Verify CSIL access *well before* next Wednesday
  - You need a user account @engineering.ucsb.edu (@cs is an alias) – apply online if you don't already have one
  - Change password as required – sign on and play a bit with Linux commands (see Reading #1)

Underlying computer system = hardware + software



Devices alert the CPU that they have a request (keyboard input, disk read return) via an ***interrupt*** (signal sent on the bus). This interrupts what the CPU is doing so that it can respond to (execute code for) the request (interrupt handler).

Thanks to Chandra Krintz and Kevin Sanft, for this figure and some other parts of these lecture notes.

Processing data & instructions

- Program instructions and data are in memory
  - CPU tracks which instruction it’s on using a dedicated register (PC) which holds the address of the instruction
- CPU stores the next few instructions in a cache – much faster to access than memory
  - Similarly stores data used by the instructions in a data cache
  - For even faster access, the CPU stores some data values and addresses in registers (fewer in number than cache entries and even faster to access than cache)
- CPU components (hardware registers, ALU, bus) all use same data width (e.g., 32 bit or 64 bit)

Processing (continued)

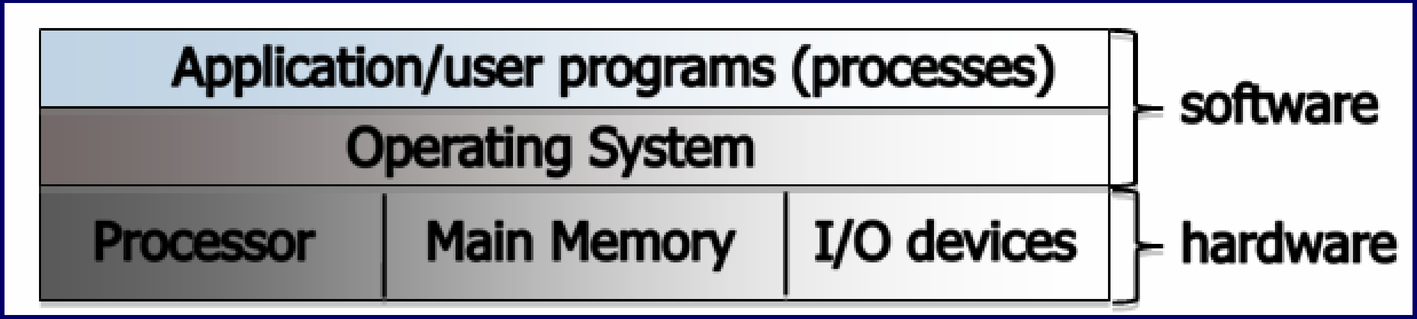
- System bus = address bus + data bus + other signals (wires)
  - CPU requests the next instruction address by putting it on the address bus (wires connected to pins)
  - CPU requests data used by the instruction (operands) by putting the addresses on the data bus
- CPU toggles other pins to identify which devices (memory, IO) it wishes to access – and whether it wants to read or write
- Devices use special wires/pins to alert the CPU that the data that the CPU requested are ready
  - The CPU doesn’t block after a request, it goes onto another task until the device “interrupts” it with the data.

Things to ponder

- How are all of these computer operations managed effectively?
  - After all, the CPU just responds to the next instruction. So how are all the instructions managed, especially when there are many clients (users, processes)?
- How are we – and our simple programs – able to deal with such a complex system?
  - Don’t we need an intermediary?

Operating systems: two views

- Top-down view: an OS is software that isolates us from the complications of hardware resources
  - In other words, an OS is an application programmer's and a user's interface to computer operations



- Bottom-up view: an OS is software that allocates and de-allocates computer resources – efficiently, fairly, orderly and securely

Types of operating systems

- Single-user, single-process – i.e., one customer, and one job at a time
- Single-user, multi-process – one workstation, but lots of stuff running
  - Actually the CPU handles just one process at any moment – jobs are swapped in/out in "time slices"
- Multi-user, multi-process – e.g., Unix/Linux
  - Same idea, but much more swapping to do
  - And added fairness, efficiency and security concerns

Unix history (Linux prequel)

- AT&T Bell Labs – System V standard
  - 1969-70: Ken Thompson wrote Unix in “B”
  - 1972: Dennis Ritchie developed C – a better B
  - Unix rewritten in C, 1973
  - … eventually System V, 1983
- UC Berkeley – BSD standard
  - Started with a copy of System IV, late 1970s
  - Lots of changes/additions in 1980s
  - Now FreeBSD
- Open source – Linux, since early 1990s

Unix philosophy (same as C)

- Small is beautiful
  - Each program does just one thing
  - Pipe commands (or use successive functions in C) to accomplish more complicated things
  - Less typing is best (using 1970s computers)
    - That’s why so many commands are short (ls, cp, mv, …)
- Users/programmers know what they are doing
  - That’s what makes the brevity sufficient
  - Means very few restrictions (or safety nets) apply

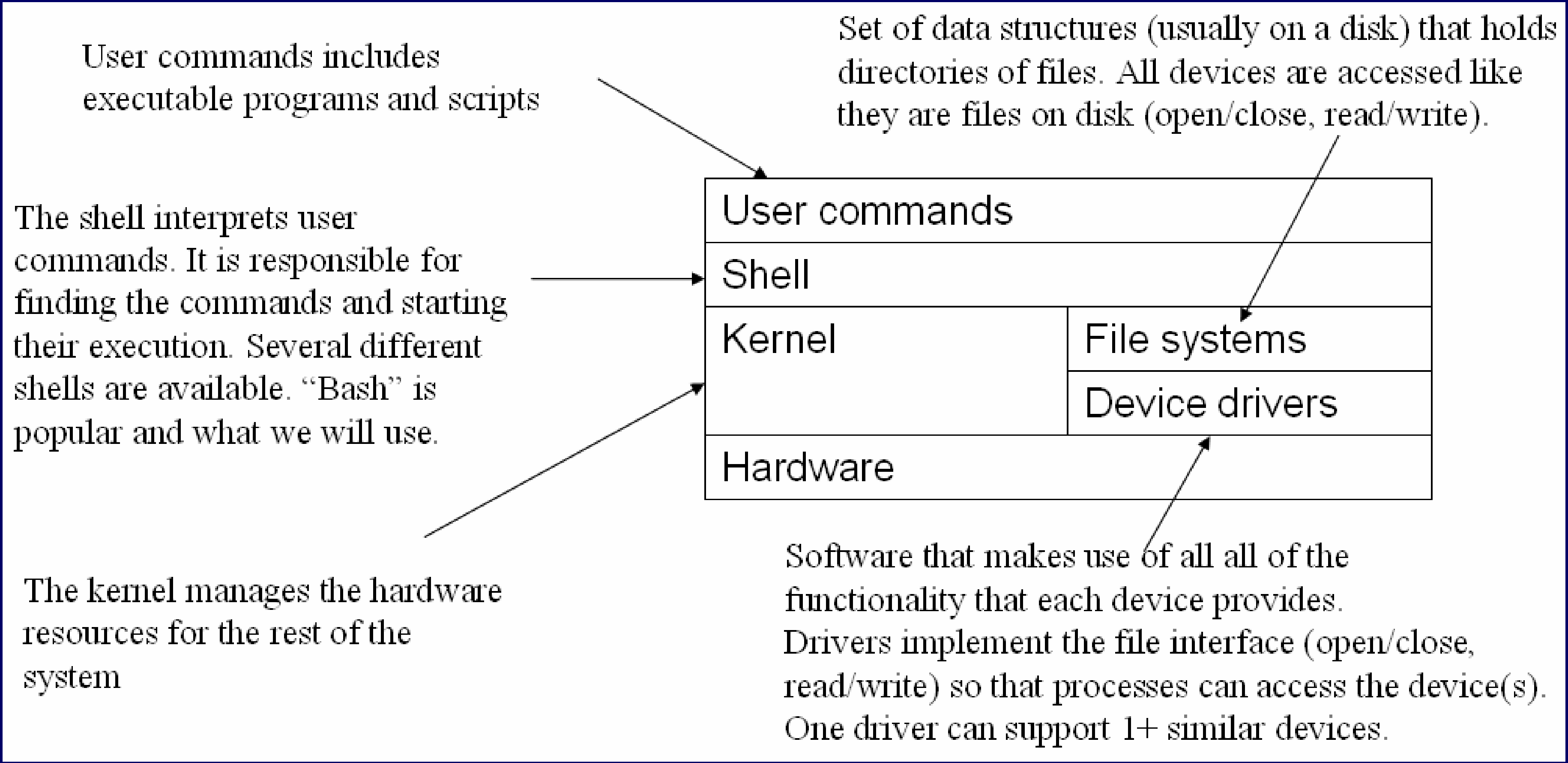
Linux

- Linus Torvalds created it as a Finnish *undergraduate* student
- Posted on Internet in 1991
  - Open source – licensed under GPL
  - Version 1.0 in 1994; version 2.2 in 1999
  - 1000's of programmers working to enhance it
- When programmers worldwide can read, modify, and redistribute a program's source code, *it evolves.*
  - People improve it, adapt it, fix bugs, …

What is Linux?

- A fully-networked Unix-like operating system
- Multi-user, multitasking, multiprocessor system
  - Fundamental in the system’s design and implementation
- Has both command-line and graphical interfaces
- Coexists with other operating systems
- Runs on multiple platforms
- Distribution includes the source code
- Can download it free from the Internet!

The Linux System


User commands includes executable programs and scripts

Set of data structures (usually on a disk) that holds directories of files. All devices are accessed like they are files on disk (open/close, read/write).

The shell interprets user commands. It is responsible for finding the commands and starting their execution. Several different shells are available. "Bash" is popular and what we will use.

The kernel manages the hardware resources for the rest of the system

Software that makes use of all all of the functionality that each device provides. Drivers implement the file interface (open/close, read/write) so that processes can access the device(s). One driver can support 1+ similar devices.

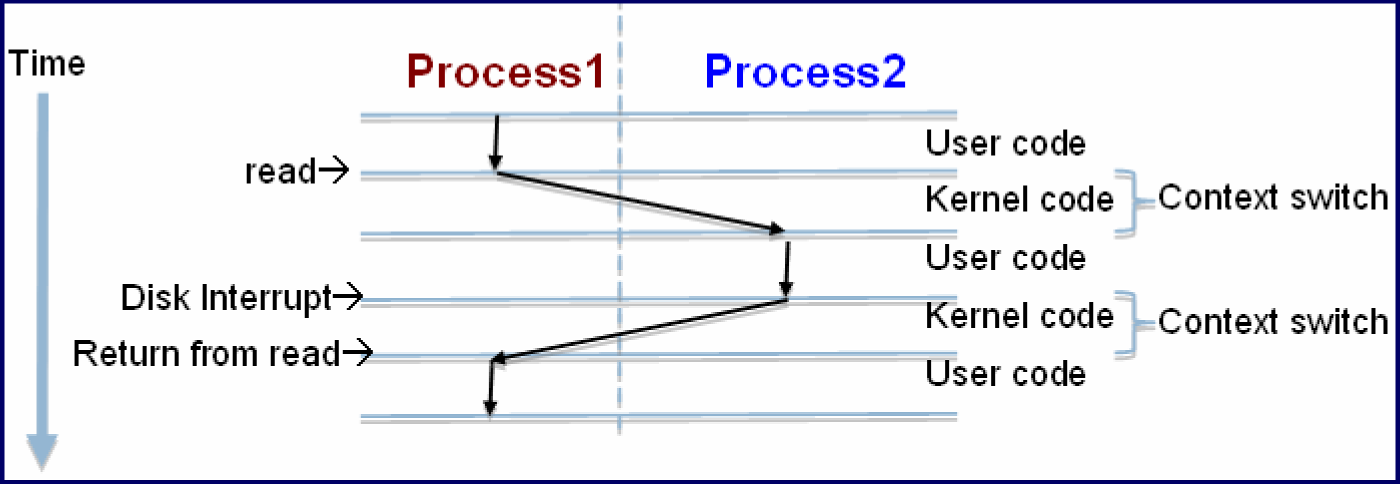
Thanks again to Chandra Krintz and Kevin Sanft.

Linux kernel – the actual OS

- Manages processes
  - Starts, stops, suspends, swaps, manages inter-process communication, …
  - Maintains their state
- Manages files (and directories)
- Manages main memory
- Manages disk operations
- Delegates to CPU(s), printers, other I/O devices

CPU scheduling

- Kernel sends interrupt to a process to give another process a turn to use the CPU
- Processes can give up CPU when they don’t need it (e.g. waiting on I/O device)

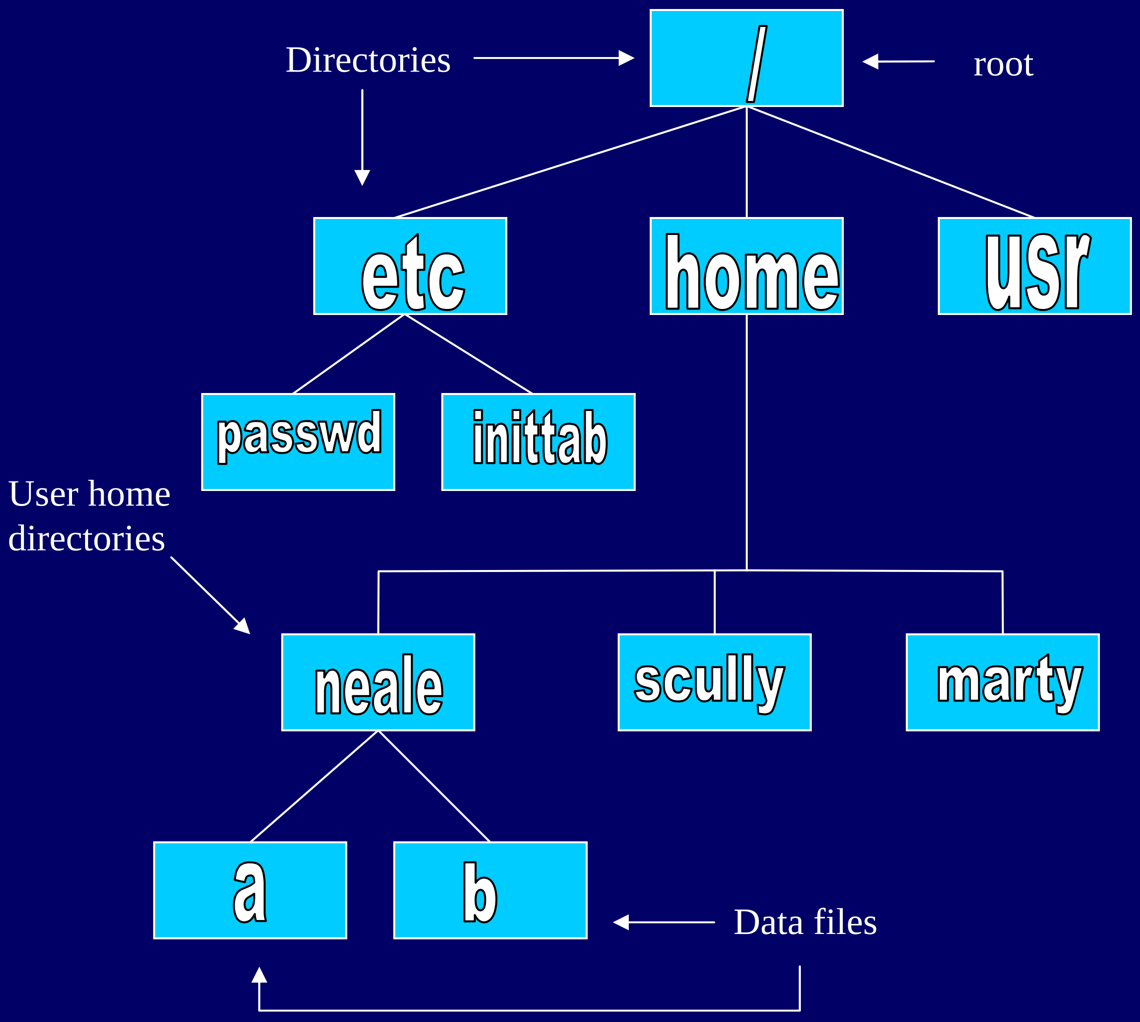


Processes *request* kernel services

- Using system calls (read, write, fork, …)
  - OOP idea: these are the kernel’s interface
  - Processes access devices just like files – that’s how they are represented by the kernel, and they occupy places in the file system
    - Use open, close, read, write, release, seek, …
- Or indirectly, by using shell commands or libraries/programs that use system calls

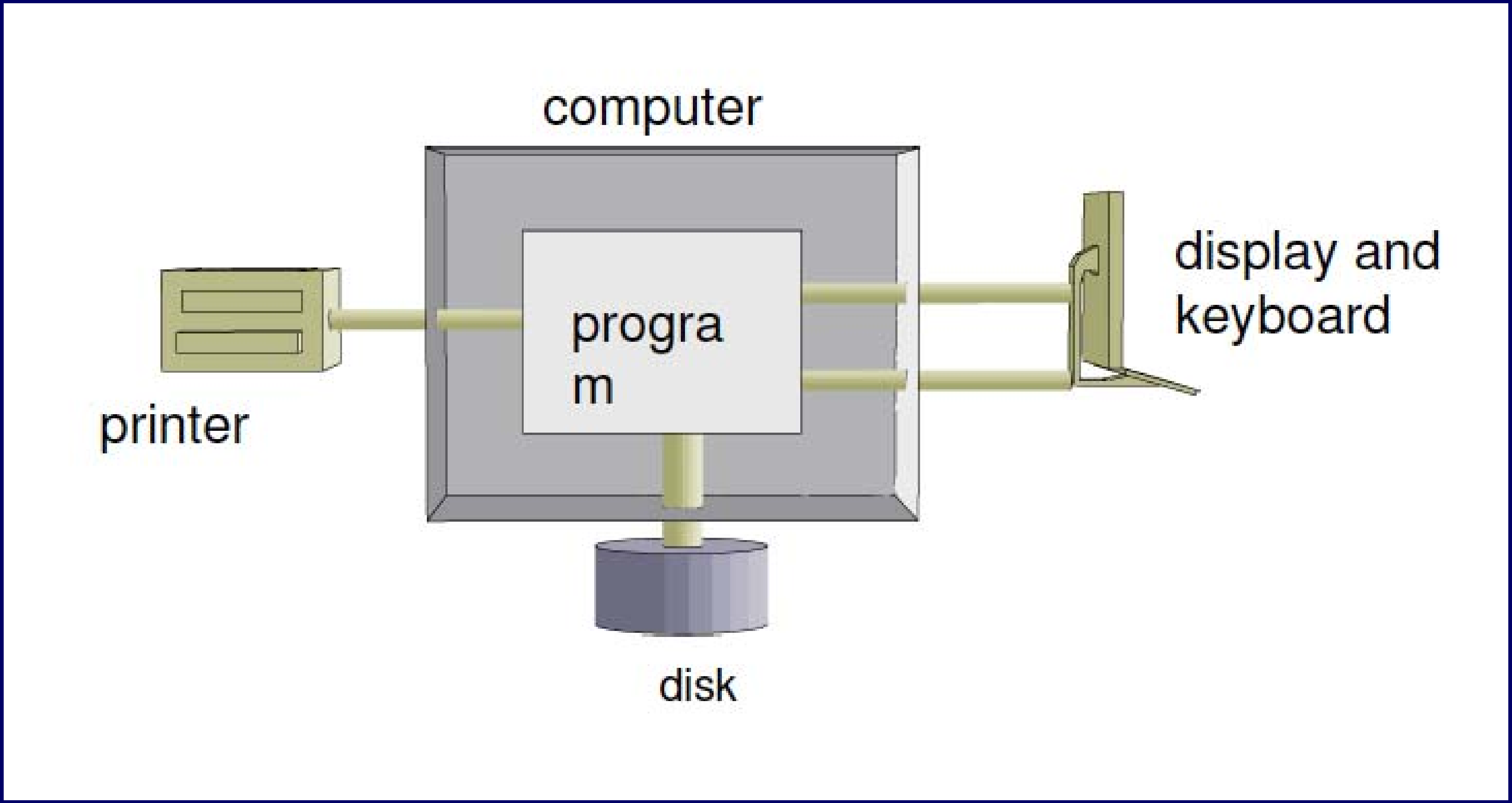
Linux file system

- Rooted, hierarchical
  - Data files are stored in *directories*
- A file's (full) *pathname* starts at the root
  - /etc/passwd
  - /home/neale/b



Some "big picture" ideas

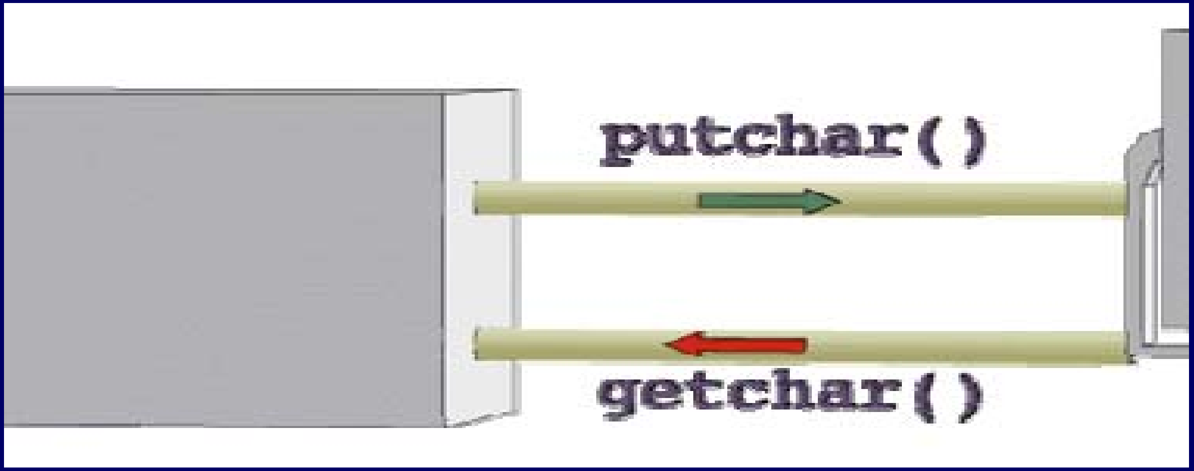
A *simple* computer model



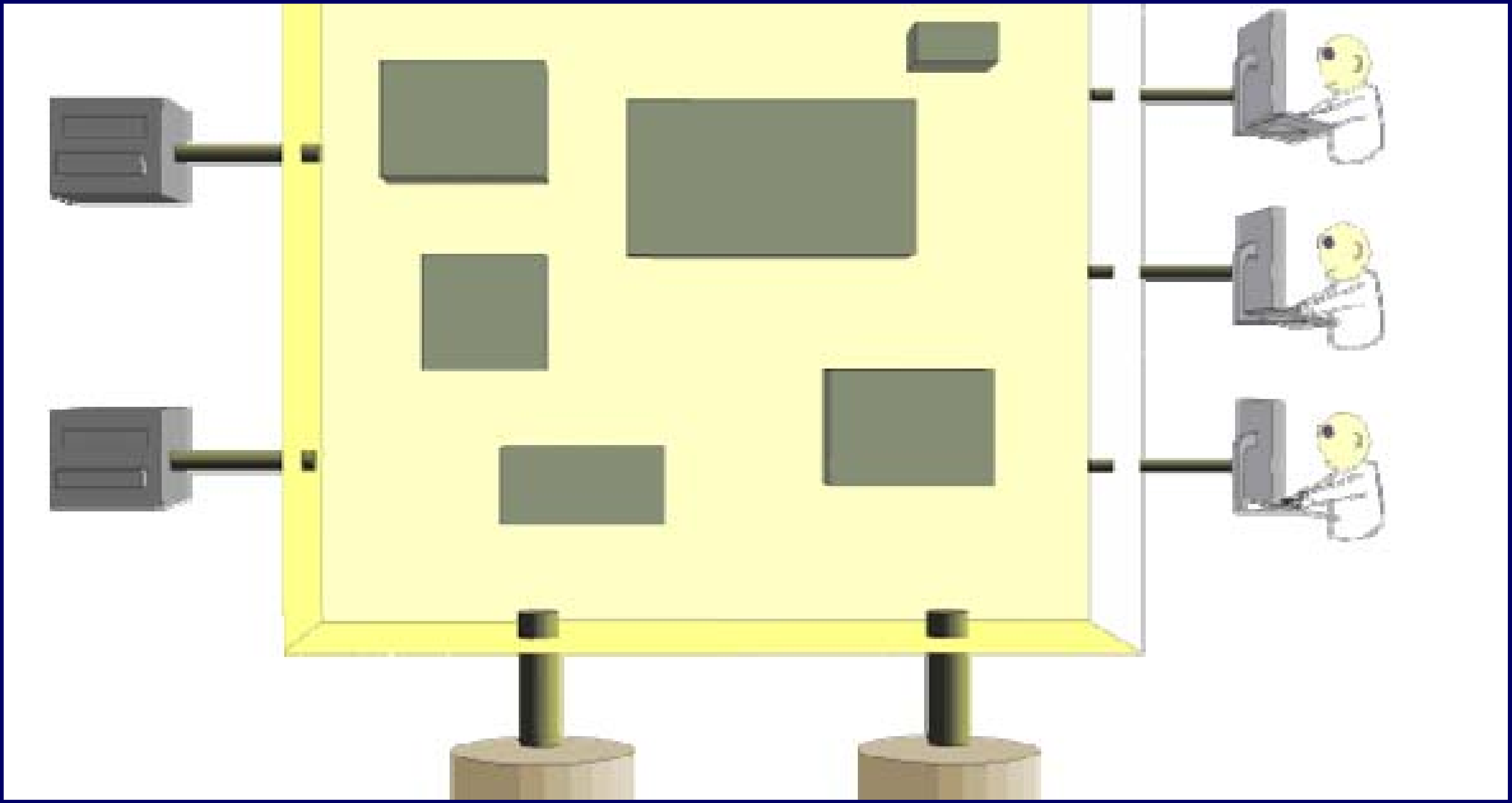
This and the next six figures derived from B. Molay's *Understanding Unix/Linux Programming*, Pearson 2003.

An example program

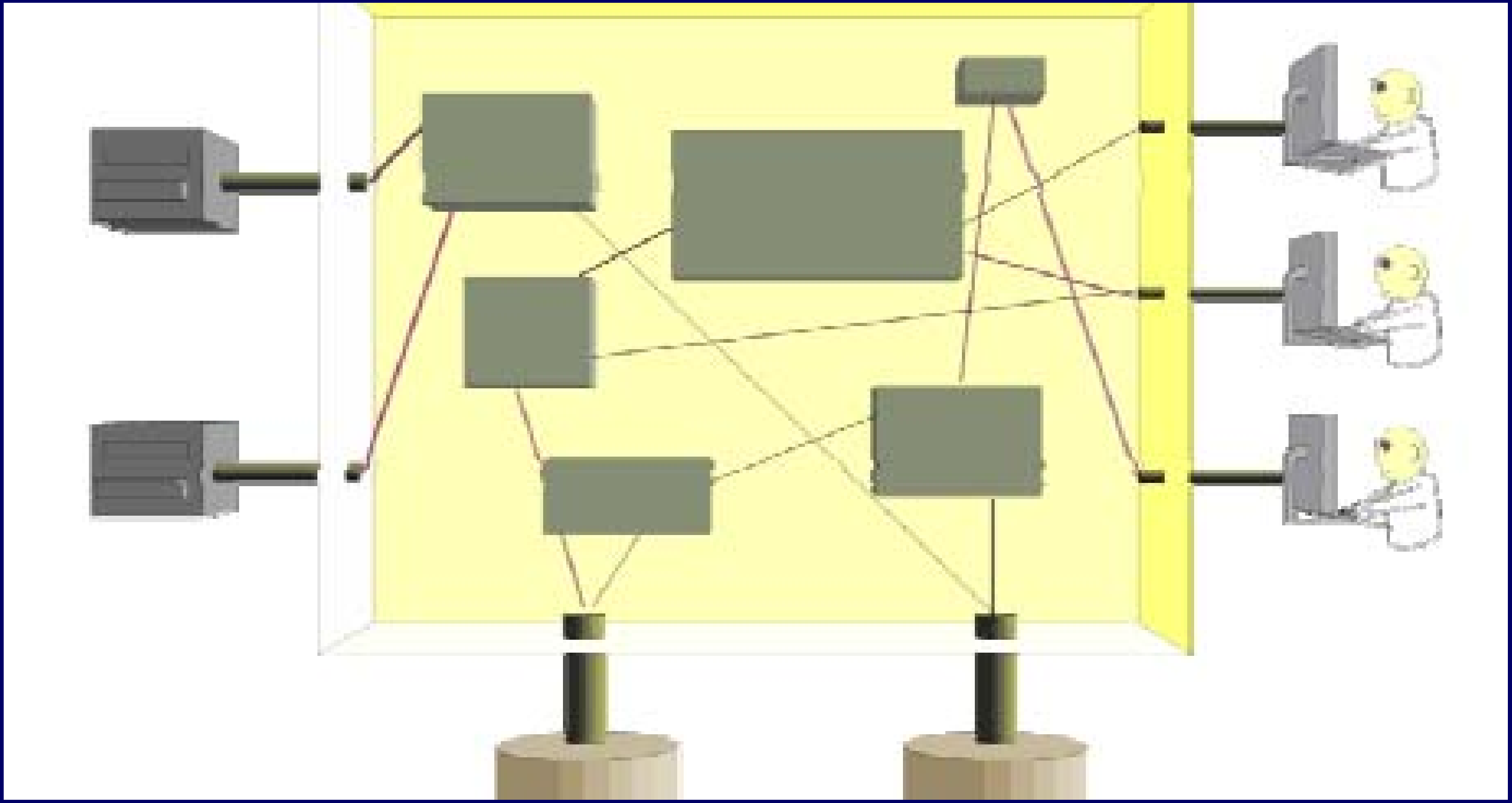
```
#include <stdio.h>
int main(void) {
    int c;
    while ( (c = getchar()) != EOF )
        putchar(c);
}
```



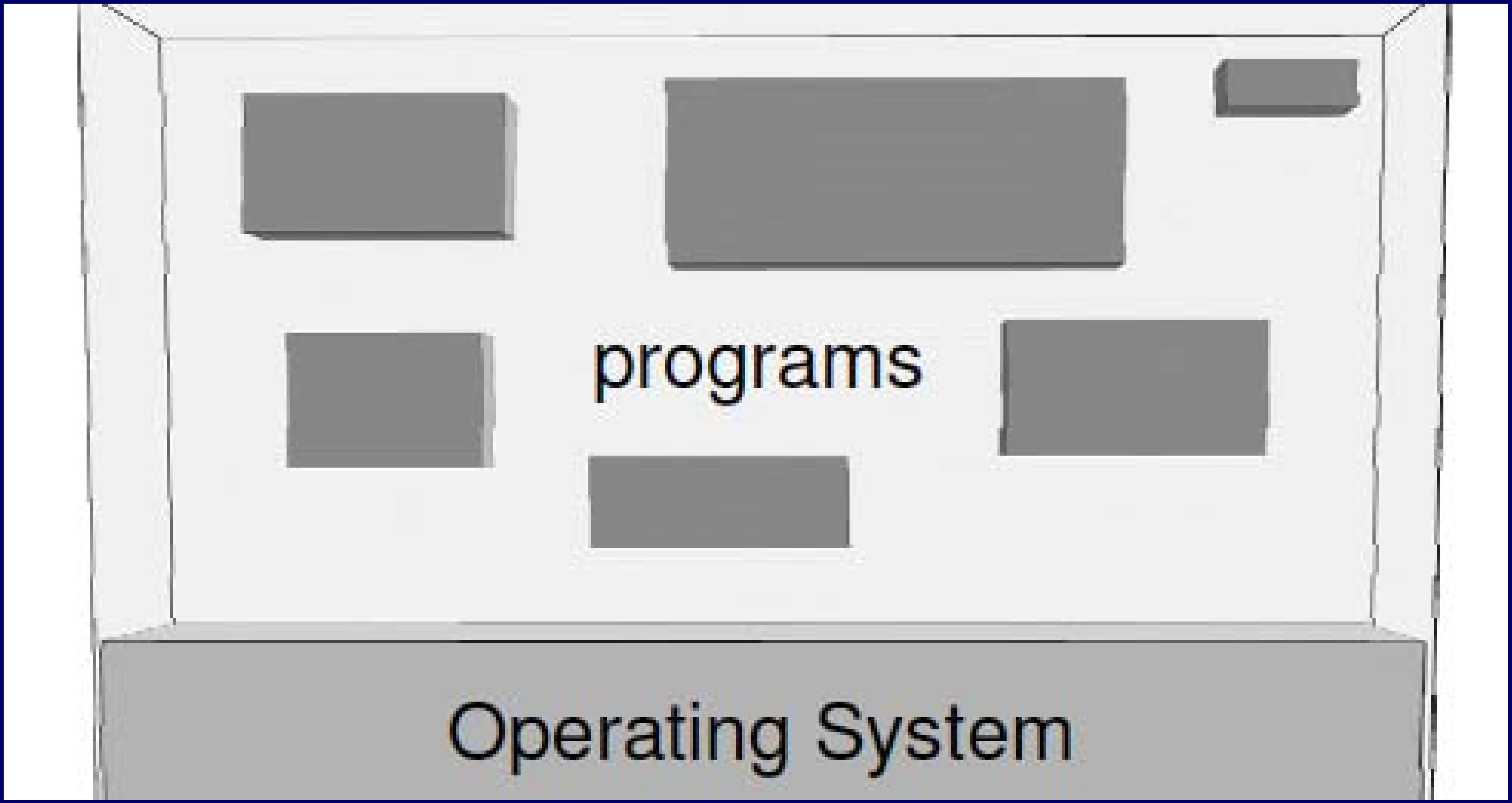
More realistic computer model



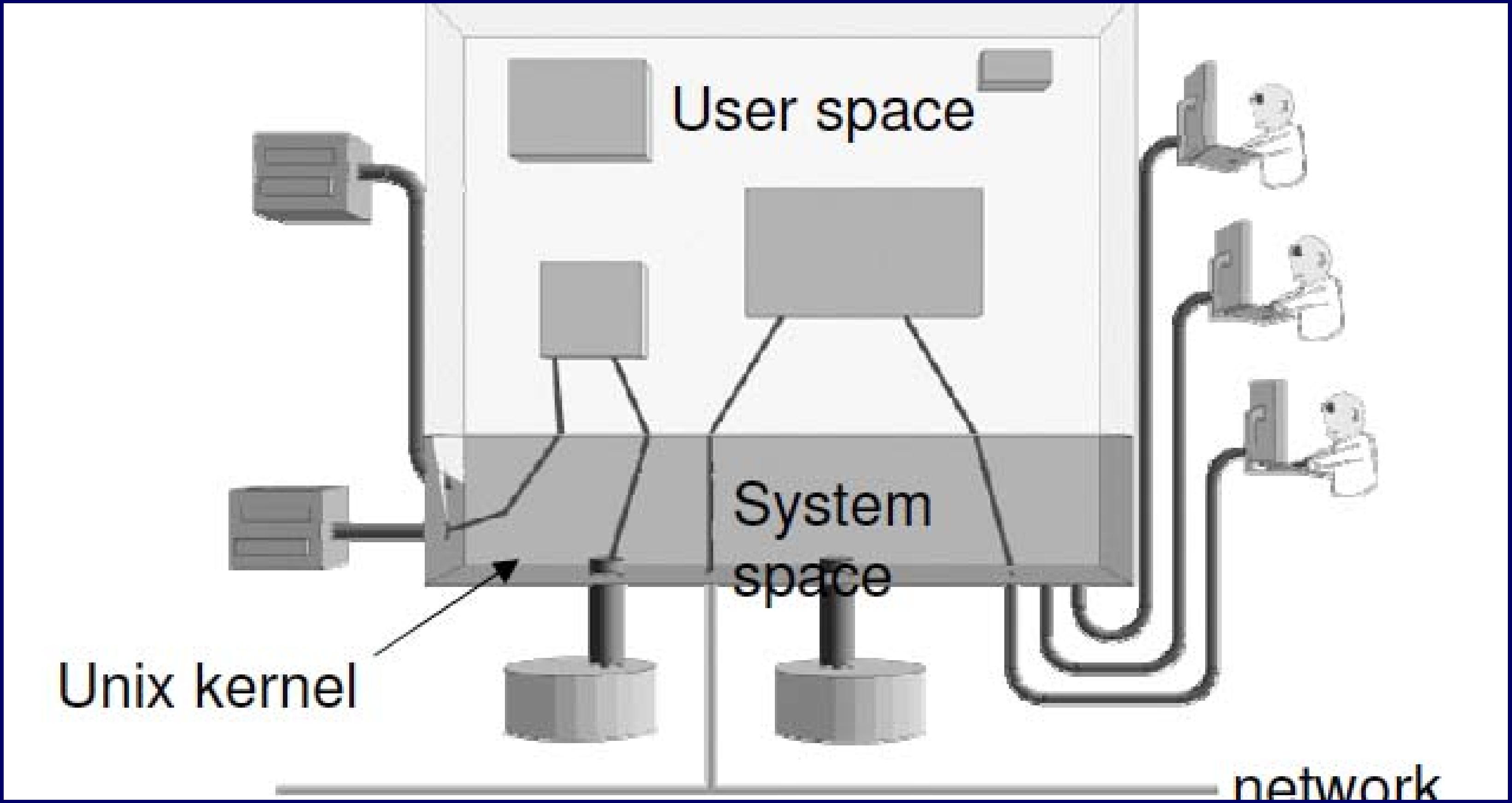
How connected? Not like this!



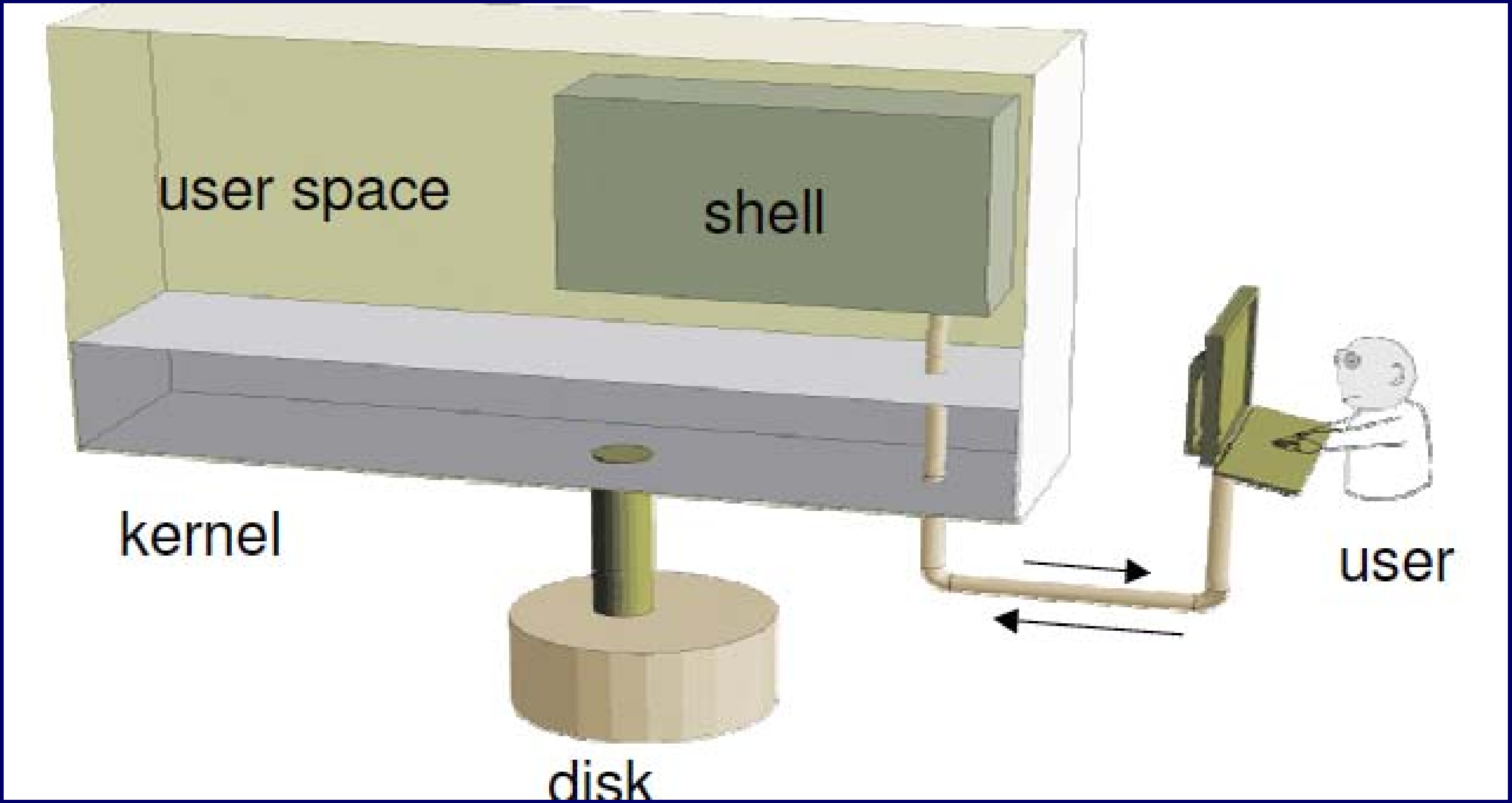
OS manages everything!



OOP idea: OS provides *services*



User interface is the shell



Shell

- A program that runs in a terminal and provides a command-line interface for user
- An interpreter that executes user commands
- Also a powerful programming language
  - Shell script – a sequence of commands in a file
- Lots of different shells to choose from
  - sh, csh, tcsh, bash …
  - We’ll focus on bash (and sh scripts) in this course

Special file names

- `.` (by itself) The current directory
  - `./a` is the same as a
- `..` The parent (toward root) directory
  - `../jane/x` go up one level then look in directory named jane for x
- `~` Your home directory
  - `~harvey` Username harvey’s home directory
- Have to “escape” spaces with a backslash
  - `my\ file\ name\ with\ spaces`
  - Moral: don’t use spaces in file or directory names!

Starting Reader #2

Object-oriented perspective

Operating system = computer interface

Shell/libraries/system calls = OS interface

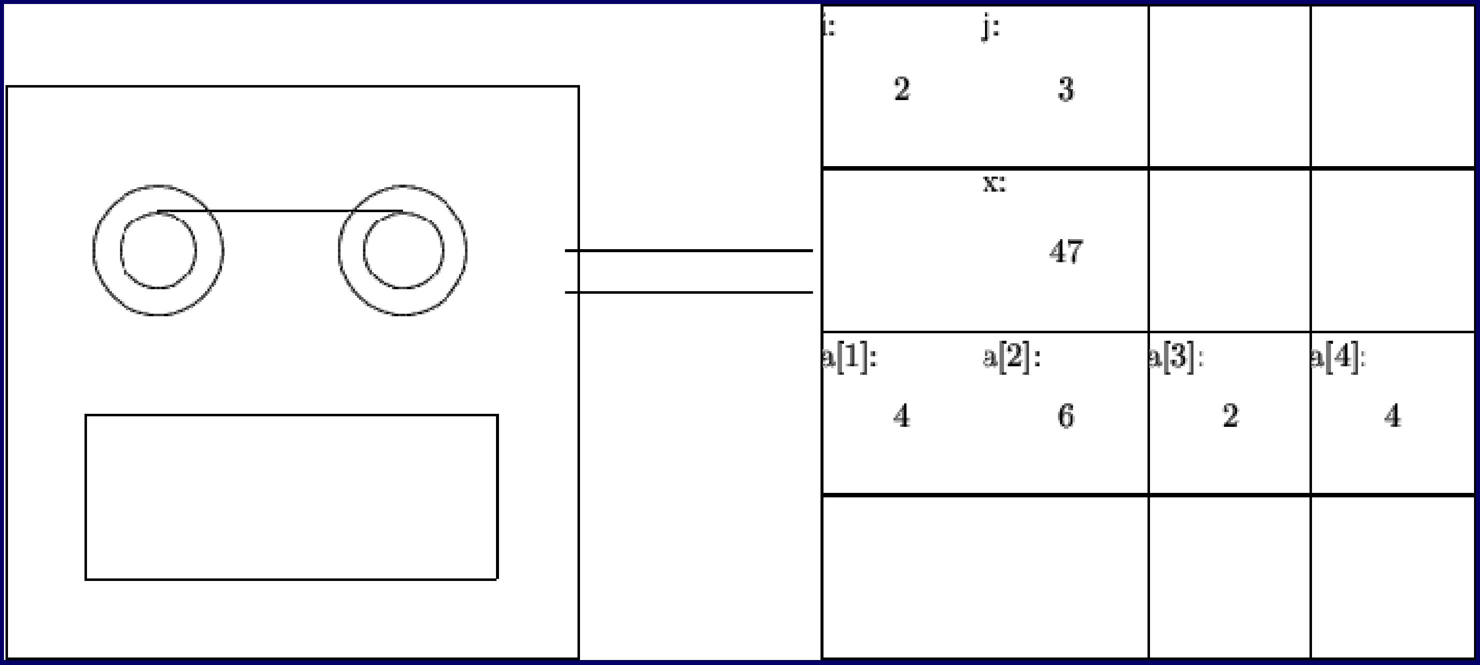
Will return to OS topics (processes, …) in upcoming lectures. Now: OO intro.

Objects

- Include *things*
  - Stack, queue, list, …
  - Window, spaceship, recipe, …
- Also include *concepts*
  - Power, trajectory, mood, …
- Can represent people, places, roles, …
- In programming: an object is a software entity encapsulating data and/or methods

Imperative programming (not OOP)

- Data, and the operations that manage the data are separate entities (physically and *logically*)



- What are implications of this programming style?

Kay's Description of OOP

1. Everything is an object.
2. Objects perform computations by making requests of each other through the passing of messages.
3. Every object has its own memory, which consists of other objects.
4. Every object is an instance of a class. A class groups similar objects.
5. The class is the repository for behavior associated with an object.
6. Classes are organized into a *singly-rooted* tree structure, called an inheritance hierarchy.

Alan Kay: "Simple things should be simple, complex things should be possible."

Solving problems *with* objects

- First decide what objects are needed
  - Instead of what functions are required
  - And instead of how specifically to handle data
- Then give each object responsibilities
  - Which probably include storing some data and performing some functions
- Finally, have objects interact by sending messages (usually method calls) to one another
  - i.e., they collaborate to fulfill responsibilities

Budd’s “real life” example

- Budd decides to send flowers to his grandmother
- First he selects an *agent*: Flo, a capable florist
  - Then he sends a message to Flo – not unlike:

    ```
    flo.sendBouquet(1, &grandma);
    ```
- The next step is Flo’s responsibility
  - Budd does not participate in this part of the process
  - Likely that *many other agents* do participate though!
- Finally Flo probably sends a message to Budd:

  ```
  budd.pay(bouquetPrice, this);
  ```

Elements of OOP - Objects

- 1. Everything is an object
  - Actions in OOP are performed by agents, called *instances* or *objects*.
- Several agents in the example scenario, including Budd, Grandma, Flo, the florist in Grandma’s city, driver, flower arranger, grower
  - Each agent has a part to play, and the result is produced when all work together in the solution of a problem.

Elements of OOP - Messages

- 2. Objects perform computations by making requests of each other through the passing of messages.
  - Actions in OOP are produced in response to requests for actions, called *messages*. An instance may accept a message, and in return will perform an action and return a value.
- To begin the process of sending the flowers, Budd gives a message to Flo. She in turn gives a message to the florist in Grandma’s city, who gives another message to the driver, and so on.