Introduction

Week 03 – Lecture 1

OS Structure, The Shell, System Calls

Team

Instructors

Giancarlo Succi

Joseph Brown

Teaching Assistants

Vladimir Ivanov

Stanislav Litvinov

Alexey Reznik

Munir Makhmutov

Hamna Aslam

Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013
    Prentice-Hall, Inc.

  and customised for the needs of this course.
- Additional input for the slides are detailed later

Operating System Structure

- Six different structures have been tried, in order to get some idea of the spectrum of possibilities
- These are not exhaustive, but they give an idea of some designs that have been tried in practice
- The six most common designs are:
  - monolithic systems
  - layered systems
  - microkernels
  - client-server systems
  - virtual machines
  - exokernels (read it at home)

Monolithic Systems (1)

- By far the most common organization
- The OS is written as a collection of procedures, linked together into a single large executable binary program that runs in kernel mode
- Each procedure in the system is free to call any other one
- However, such a structure may lead to a system that is unwieldy and difficult to understand
- A crash in any of these procedures will take down the entire operating system

Monolithic Systems (2)

- To construct the actual object program of the OS, wee need to compile all the individual procedures (or the files containing the procedures)
- Then to bind them all together into a single executable file using the system linker
- Every procedure is visible to every other procedure

Monolithic Systems (3)

- Basic structure of OS (Fig. 1-24):
  - A main program that invokes the requested service procedure
  - A set of service procedures that carry out the system calls
  - A set of utility procedures that help the service procedures
- In addition to the core OS that is loaded when the computer is booted, many OSs support loadable extensions, such as I/O device drivers and file systems which are loaded on demand. In UNIX they are called **shared libraries**. In Windows they are called **DLLs (Dynamic-Link Libraries)**

Monolithic Systems (4)

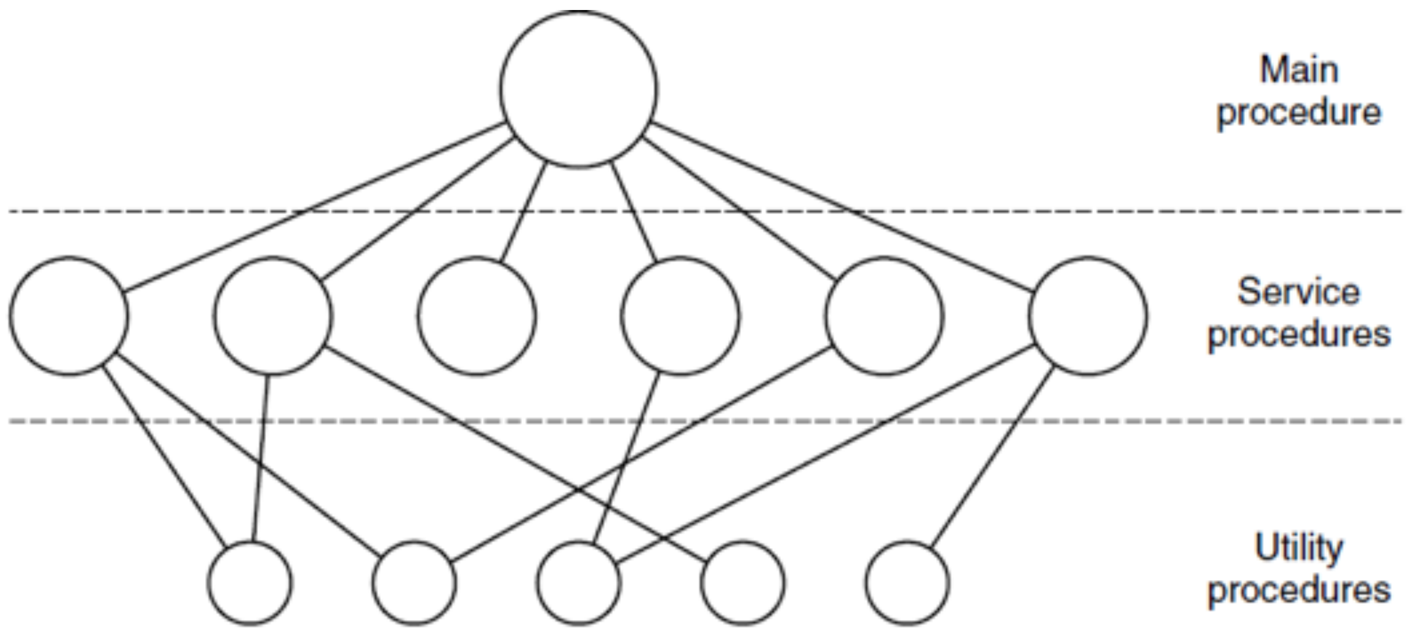


Figure 1-24. A simple structuring model
for a monolithic system.

Layered Systems (1)

- A generalization of the structuring approach of the monolithic system is to organize the OS as a hierarchy of layers
- The first layered system was the THE system built by Edsger W. Dijkstra and his students in 1968 (Fig. 1-25)
- A further generalization of the layering concept was present in the MULTICS system, where instead of layers the OS had a series of concentric rings, with the inner ones being more privileged than the outer ones

Layered Systems (2)

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Figure 1-25. Structure of the THE operating system.

Layered Systems (3)

- The entire OS was part of the address space of each user process in MULTICS
- The hardware made it possible to designate individual procedures (memory segments, actually) as protected against reading, writing, or executing
- All the parts of the THE system were linked together into a single executable program. Layering scheme was really only a design aid
- In MULTICS, the ring mechanism was very much present at run time and enforced by the hardware
- It can easily be extended to structure user subsystems providing means of security to different user groups

Microkernels (1)

- Bugs in the kernel can bring down the system instantly
- Industrial systems may contain 2 to 10 bugs per thousand lines of code (KLOC)
- Thus, monolithic operating system of 5 MLOC is likely to contain between 10,000 and 50,000 kernel bugs
- The basic idea behind the microkernel design is to achieve high reliability by splitting the operating system up into small, well-defined modules

Microkernels (2)

- One of the modules – the microkernel – runs in kernel mode and the rest run as relatively powerless ordinary user processes
- Such systems are dominant in real-time, industrial, avionics, and military applications that are mission critical and have very high reliability requirements
- For example, The MINIX 3 microkernel (Fig. 1-26) is only about 12,000 lines of C and some 1400 lines of assembler for very low-level functions such as catching interrupts and switching processes

Microkernels (3)

- Microkernel provides a set of about 40 kernel calls
- These calls perform functions like:
  - hooking handlers to interrupts
  - moving data between address spaces
  - installing memory maps for new processes
  - managing the clock
- Outside the kernel there are three layers of processes all running in user mode

Microkernels (4)

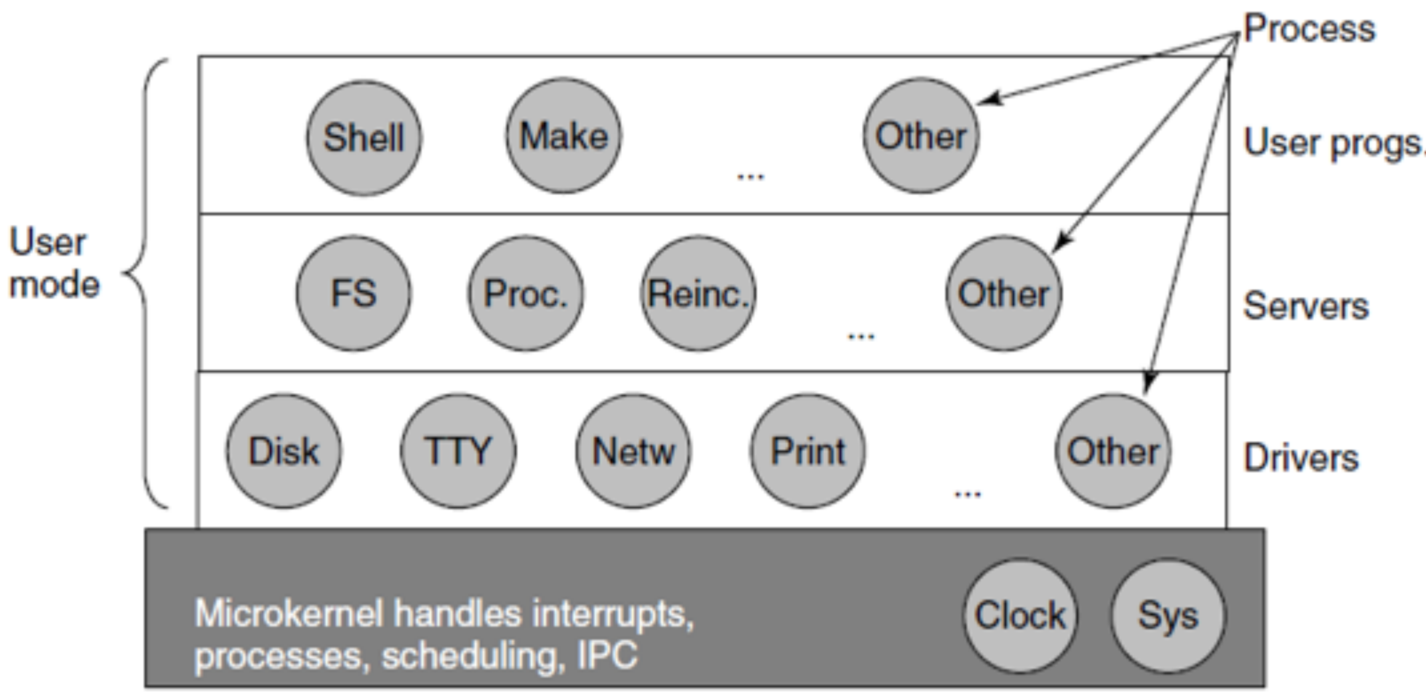


Figure 1-26. Simplified structure of the MINIX 3 system.

Client-Server Model (1)

- In client-server model two classes of processes exist:
  - the **servers**, each of which provides some service
  - the **clients**, which use these services
- To obtain a service, a client process constructs a message saying what it wants and sends it to the appropriate service
- The service does the work and sends back the answer
- The client-server model is an abstraction that can be used for a single machine or for a network of machines (Fig. 1-27)

Client-Server Model (2)

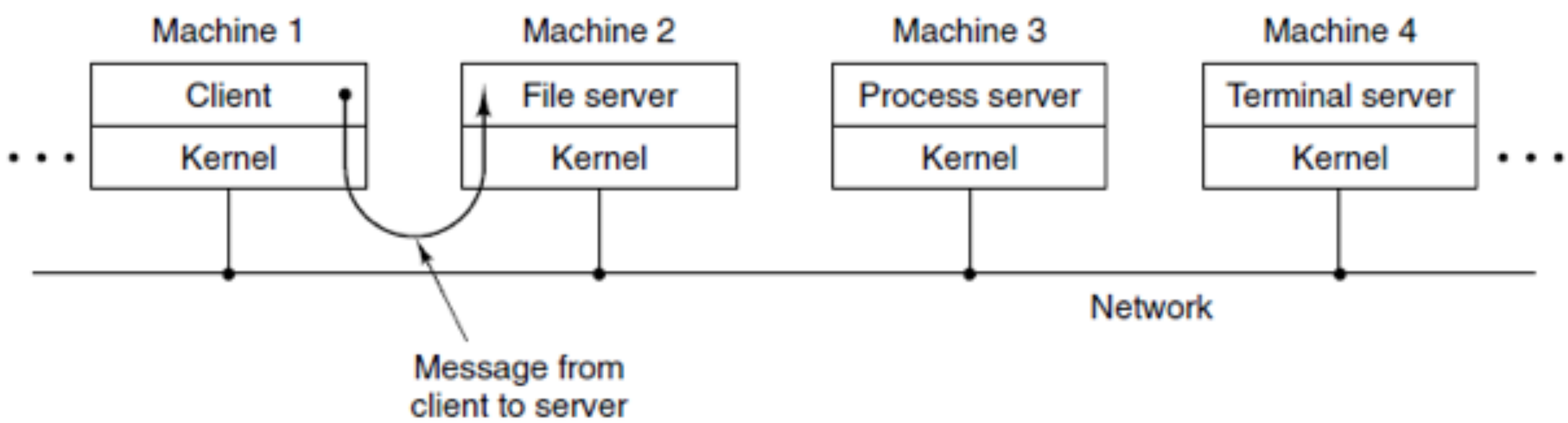


Figure 1-27. The client-server model over a network.

Virtual Machines (1)

- Many 360 users wanted to be able to work interactively at a terminal
- Various groups, both inside and outside IBM, decided to write timesharing systems for it
- They developed their own system TSS/360 that was too big and too slow and was later replaced by CP/CMS (later renamed VM/370)
- The **virtual machine monitor** runs on the bare hardware and does the multiprogramming, providing several virtual machines to the next layer up (Fig. 1-28)

Virtual Machines (2)

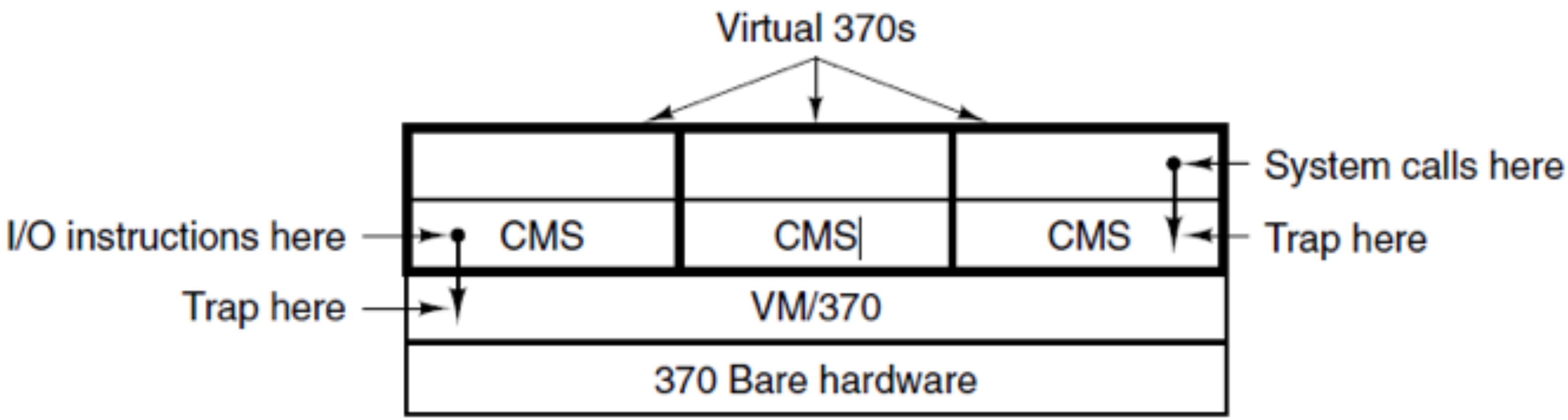


Figure 1-28. The structure of VM/370 with CMS.

Virtual Machines Rediscovered (1)

- Initially, not many companies were interested in the idea of virtualization
- New needs, software and technologies made it possible to grow:
  - Companies could run different virtual machines (DB server, web-server, etc.) on one server to avoid crashing the whole system if one of them crashes
  - Renting a virtual machine is cheaper than renting the whole server which is good for small companies
  - Users are able to run two or more different OSs at the same time (Fig. 1-29a)

Virtual Machines Rediscovered (2)

- Problem: in order to run virtual machine software on a computer, its CPU must be virtualizable:
  - an OS running on a virtual machine in user mode
  - when it executes a privileged instruction (modifying the PSW or doing I/O), it is essential that the hardware trap to the virtual-machine monitor so the instruction can be emulated in software
  - on some CPUs attempts to execute privileged instructions in user mode are ignored
  - this made virtualization impossible because of hardware limitation

Virtual Machines Rediscovered (3)

- Though it was possible to create an interpreter (for example, Bochs), the performance loss was too high
- The solution was to translate blocks of code on the fly, storing them in an internal cache, and then reusing them if they were executed again
- This technique is called **binary translation** and it led to appearance of **machine simulators** (Fig 1-29b)
- It was still not fast enough for commercial use

Virtual Machines Rediscovered (4)

- The next step was to include a kernel module which led to appearance of **type 2 hypervisors** (Fig. 1-29c)
- The difference between type 1 and type 2 hypervisors is that **type 1** runs on bare metal, whereas **type 2** uses **host OS** and its file system to create processes, store files, and so on
- Type 2 hypervisor creates virtual disk, which is just a large file stored in host OS's file system. This disk is used to install a guest OS

Virtual Machines Rediscovered (5)

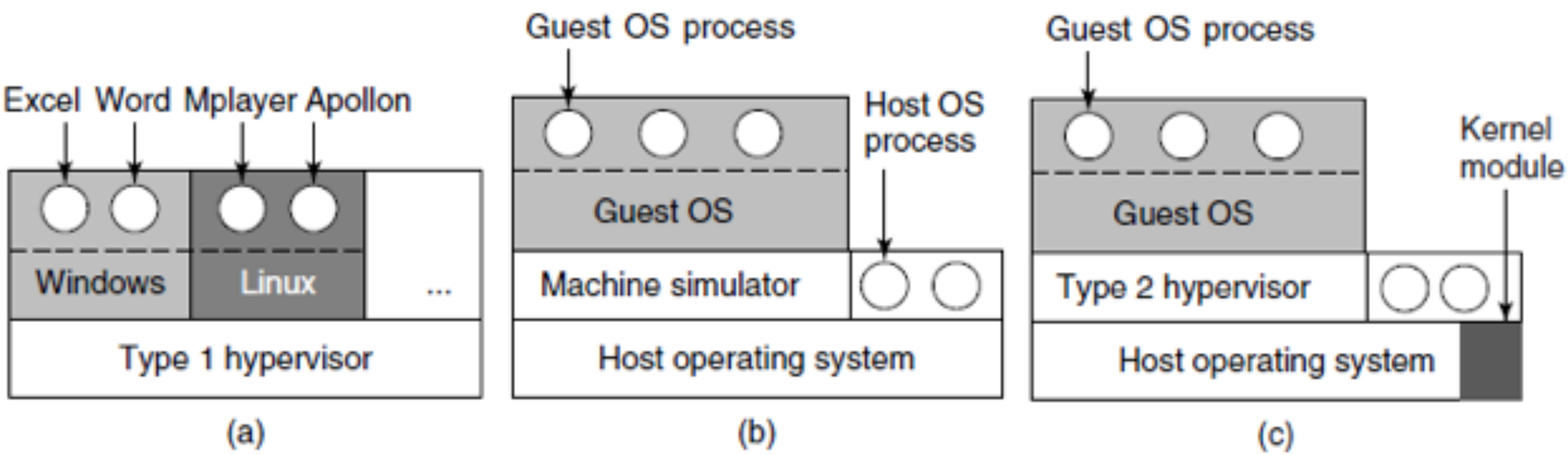


Figure 1-29. (a) A type 1 hypervisor. (b) A pure type 2 hypervisor. (c) A practical type 2 hypervisor.

The Java Virtual Machine

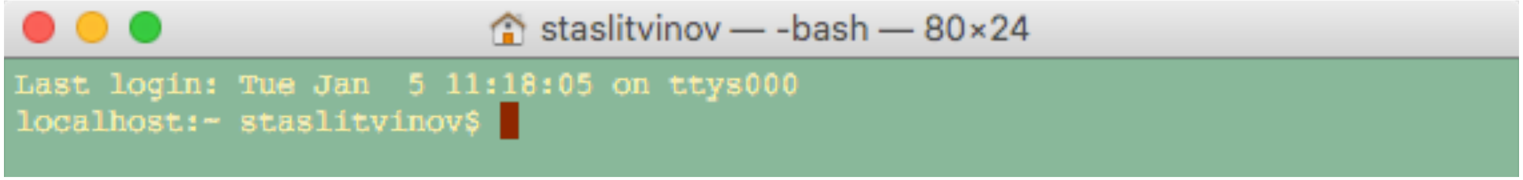
- Virtual machines are also used for running Java programs
- The Java compiler produces code for JVM (Java Virtual Machine), which then typically is executed by a software JVM interpreter
- JVM code can run by any computer that has a JVM interpreter. If the compiler had produced binary programs, they could not have been shipped and run anywhere as easily
- If the interpreter is implemented properly, incoming JVM programs can be checked for safety and then executed in a protected environment so they cannot steal data or do any damage

The Shell (1)

- The OS is the code that carries out the system calls
- Editors, compilers, assemblers, linkers, utility programs, and command interpreters **are not part of the OS**
- The shell is a command interpreter that acts as the main interface between a user the OS
- Another alternative (or on the top of the shell) there is a GUI (Graphical User Interface)
- There are many shells: *sh*, *csh*, *ksh*, *bash*, etc.

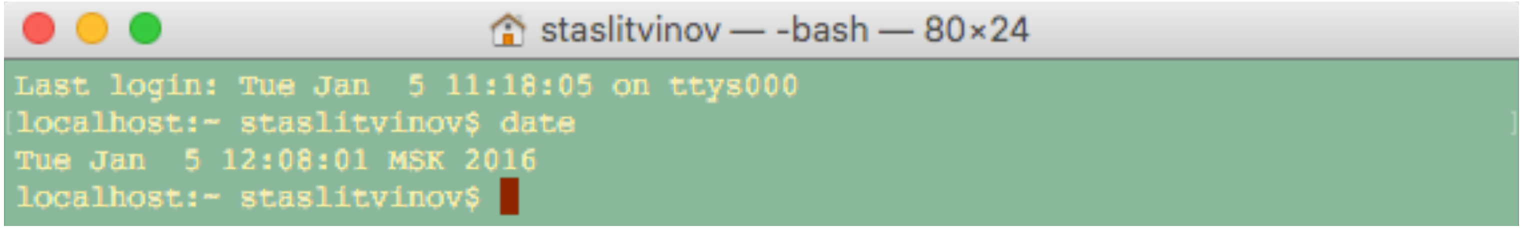
The Shell (2)

- When any user logs in, a shell is started up
- The shell has the terminal as standard input and standard output
- It starts out by typing the prompt, a character (or a string) such as a dollar sign, which tells the user that the shell is waiting to accept a command



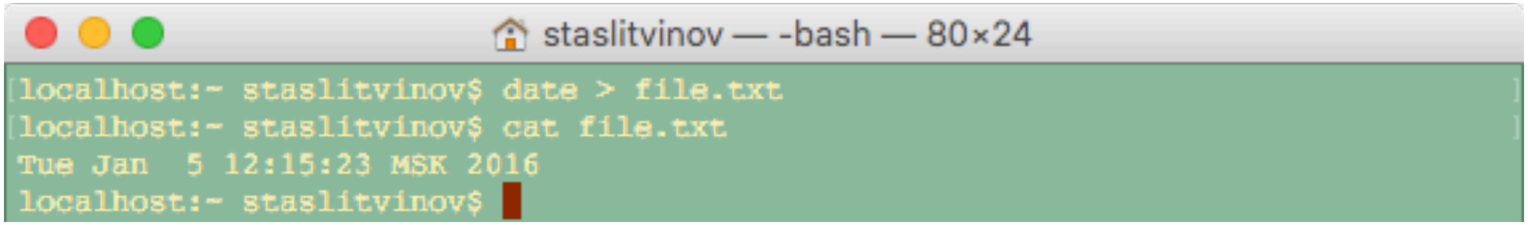
The Shell (3)

- Let's type *date* command:
  - the shell creates a child process
  - then it runs the date program as the child
  - while the child process is running, the shell waits for it to terminate
  - at the end, the shell types the prompt again and tries to read the next input line



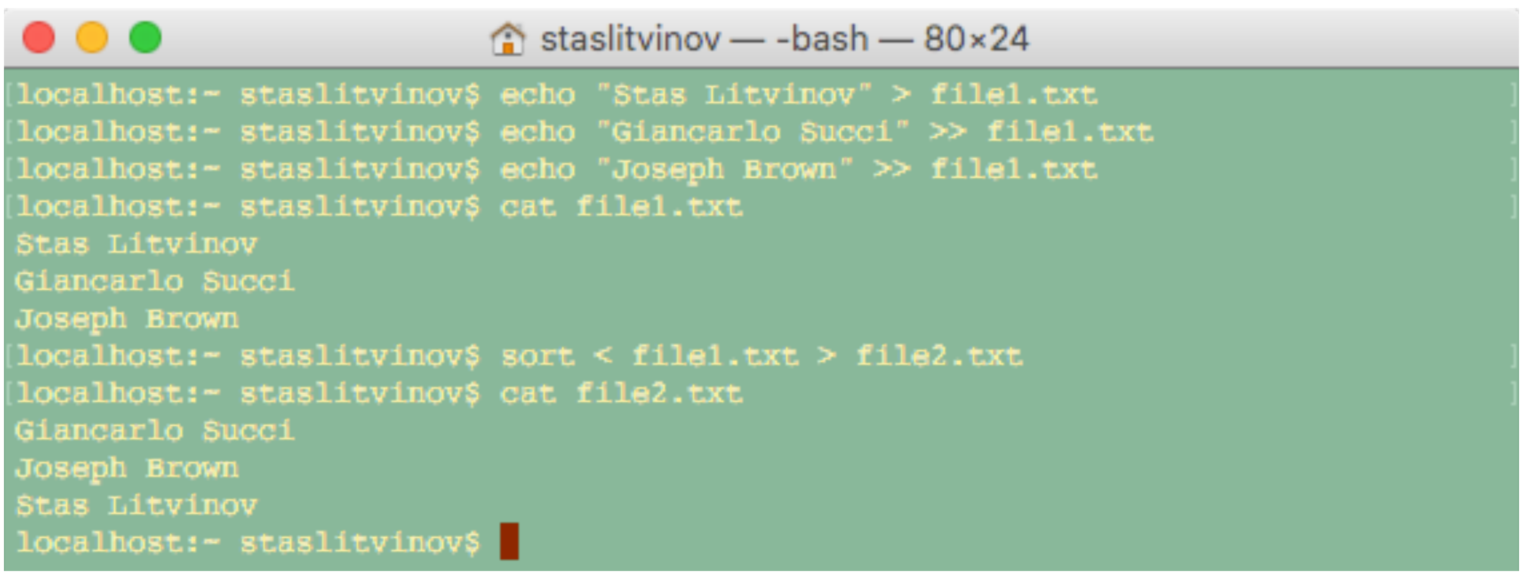
The Shell (4)

- The user can specify that standard output be redirected to a file, for example:
  - *date > file.txt*



The Shell (5)

- Standard input can be redirected too:
  - *sort < file1.txt > file2.txt*
- It invokes the sort program with input taken from *file1.txt* and output sent to *file2.txt*.

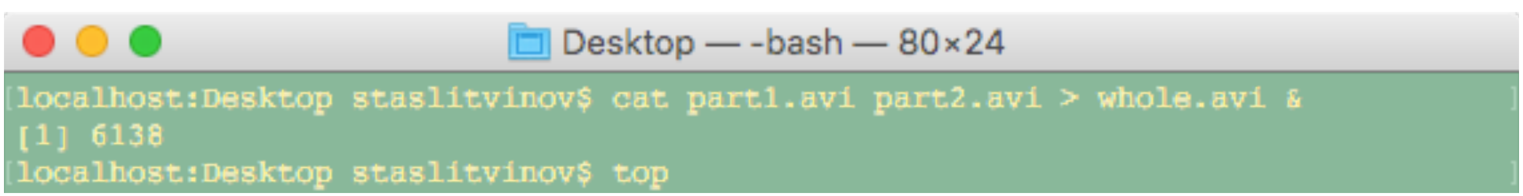


The Shell (6)

- The output of one program can be used as the input for another program by connecting them with a pipe:
  - *cat file1 file2 file3 | sort >/dev/lp*
- The command con*cat*enates three files and sends the output to *sort* to arrange all the lines in alphabetical order
- The output of sort is redirected to the file /*dev/lp*, typically the printer

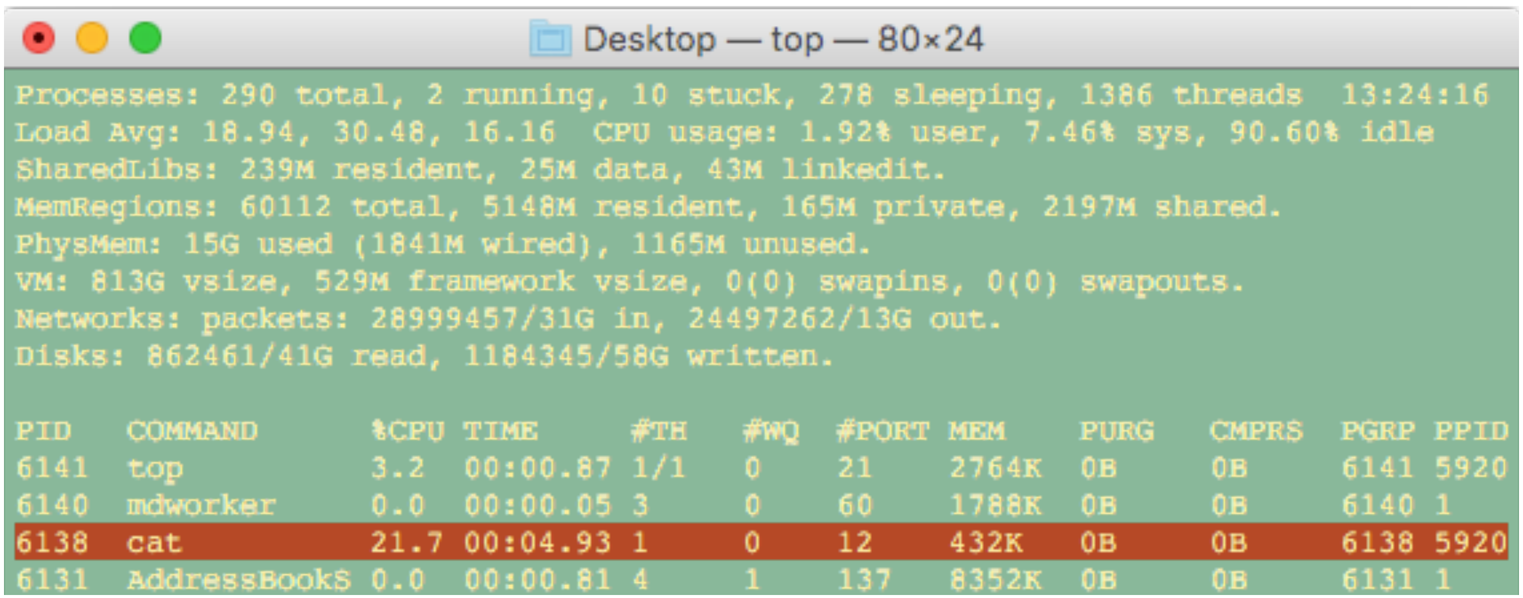
The Shell (7)

- If a user puts an ampersand sign (&) after a command, the shell will not wait for a process to complete
- Instead, it will run a background job and present the prompt to the user immediately
- Let's imagine that we want to merge two large video files:



The Shell (8)

- The shell returns **PID (Process ID)** and displays prompt again, so it is possible to run another command
- If we run **top** command, we will see the process running in background



System Calls Intro (1)

- OSs have two main functions:
  - to provide abstractions to user programs
  - to manage the computer's resources
- The interface between user programs and the OS is primarily about dealing with the abstractions
- The system calls available in the interface vary from one OS to another
- We will discuss material specific for one OS or family of OSs, rather then describing it in general

System Calls Intro (2)

- A single-CPU computer can execute only one instruction at a time
- If a process is running a user program in user mode and needs a system service, it has to execute a trap instruction to transfer control to the OS
- The OS figures out what the calling process wants by inspecting the parameters
- Then it carries out the system call and returns control to the instruction following the system call

System Call Example (1)

- Let's take a look at the ***read*** system call:
  - count = read(fd, buffer, nbytes);
- It has three parameters:
  - the first one specifying the file
  - the second one pointing to the buffer
  - the third one is number of bytes to read
- The system call (and the library procedure) return the number of bytes actually read in *count*
- If the system call cannot be carried out owing to an invalid parameter or a disk error, *count* is set to −1, and the error number is put in a global variable, *errno*

System Call Example (1)

- Steps 1–3: In preparation for calling the ***read*** library procedure, which actually makes the ***read*** system call, the calling program first pushes the parameters onto the stack (Fig. 1-17)
- Step 4: the actual call to the library procedure
- Step 5: The library procedure, possibly written in assembly language, typically puts the system-call number in a place where the operating system expects it, such as a register

System Call Example (2)

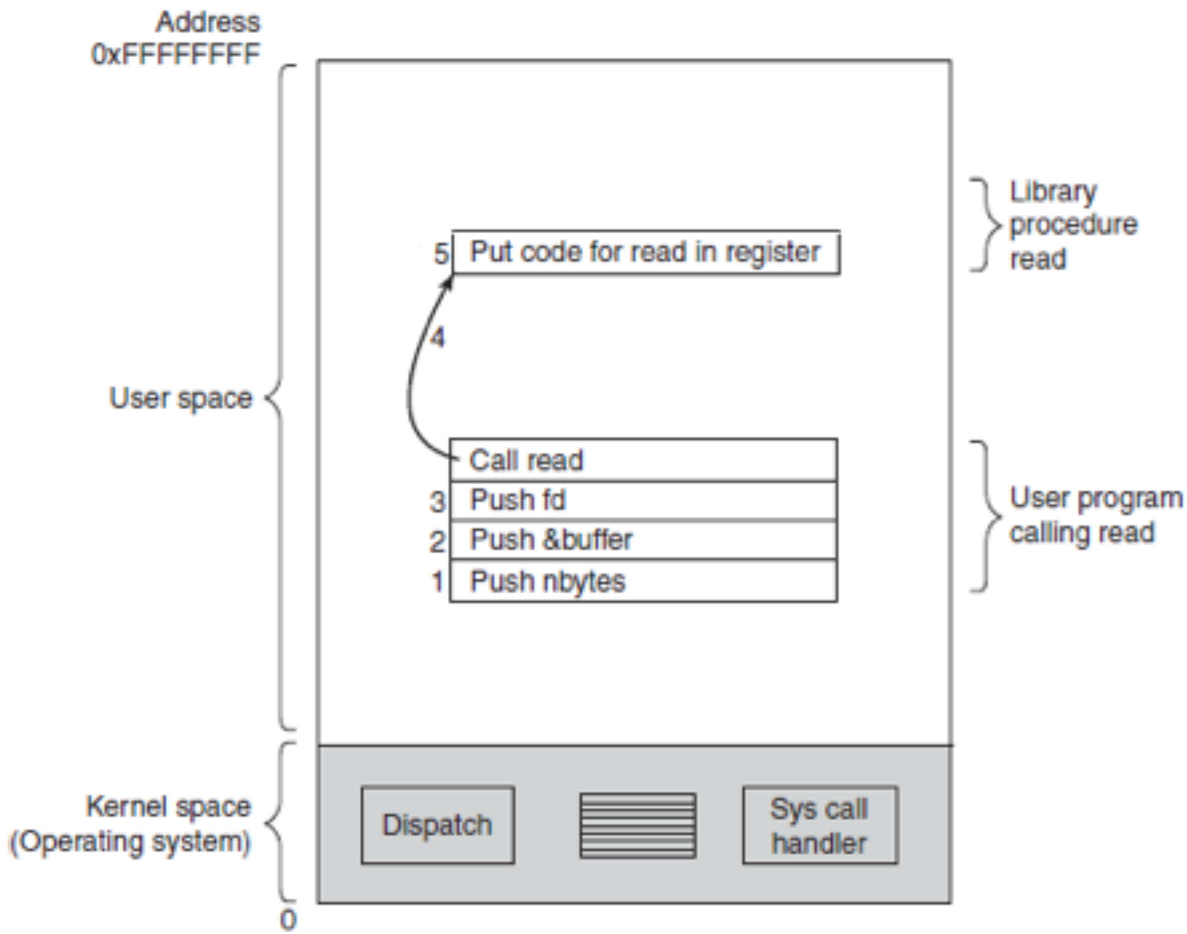


Figure 1-17. The 11 steps in making the system call *read(fd, buffer, nbytes)*. Steps 1-5

System Call Example (3)

- Step 6: it executes a ***TRAP*** instruction to switch from **user mode** to **kernel mode** and start execution at a fixed address within the kernel
- Step 7: the kernel code that starts following the TRAP examines the system-call number and then dispatches to the correct system-call handler, usually via a table of pointers to system-call handlers indexed on system-call number
- Step 8: the system-call handler runs

System Call Example (4)

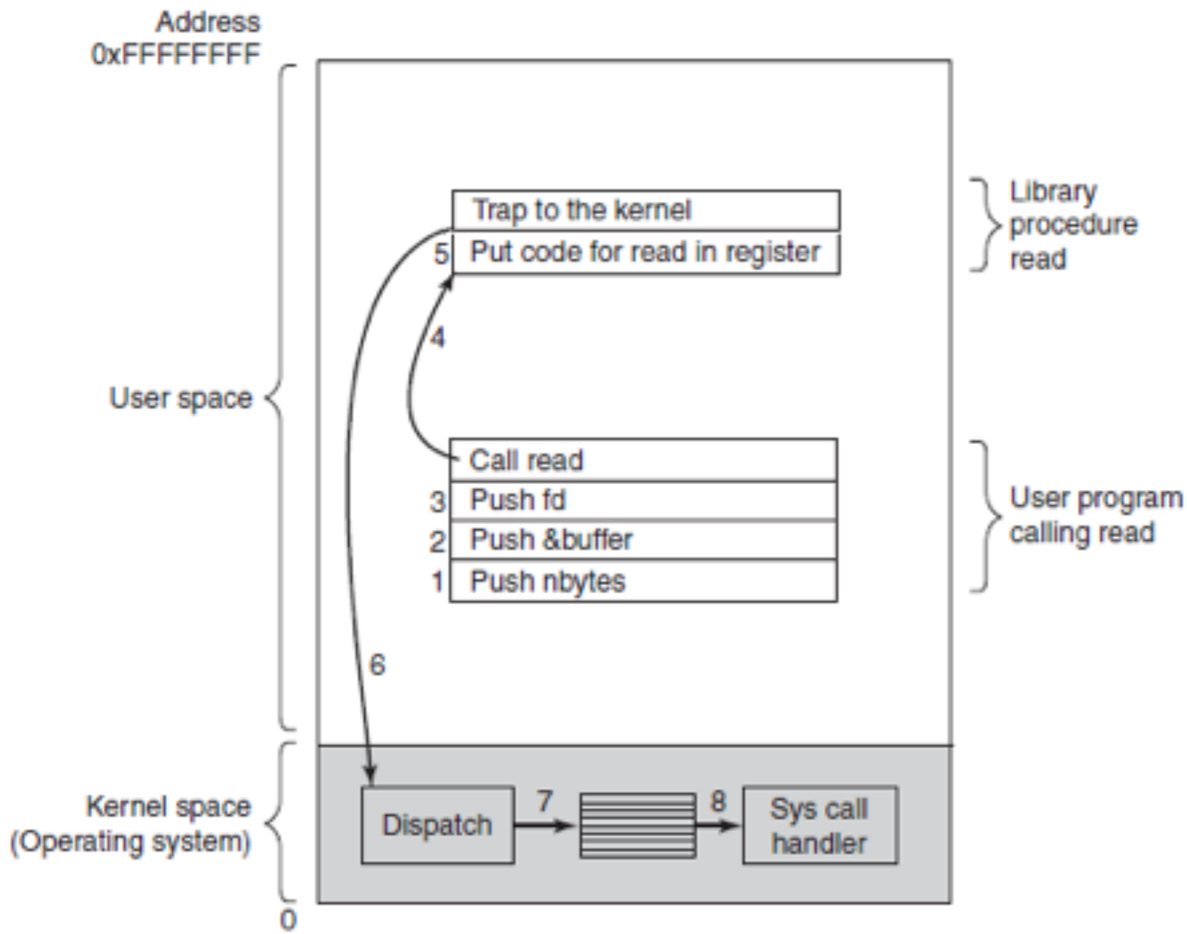


Figure 1-17. The 11 steps in making the system call *read(fd, buffer, nbytes)*. Steps 1-8

System Call Example (5)

- Step 9: once the handler has completed its work, control may be returned to the user-space library procedure at the instruction following the TRAP instruction
- Step 10: this procedure then returns to the user program in the usual way procedure calls return
- Step 11: to finish the job, the user program has to clean up the stack, as it does after any procedure call

System Call Example (6)

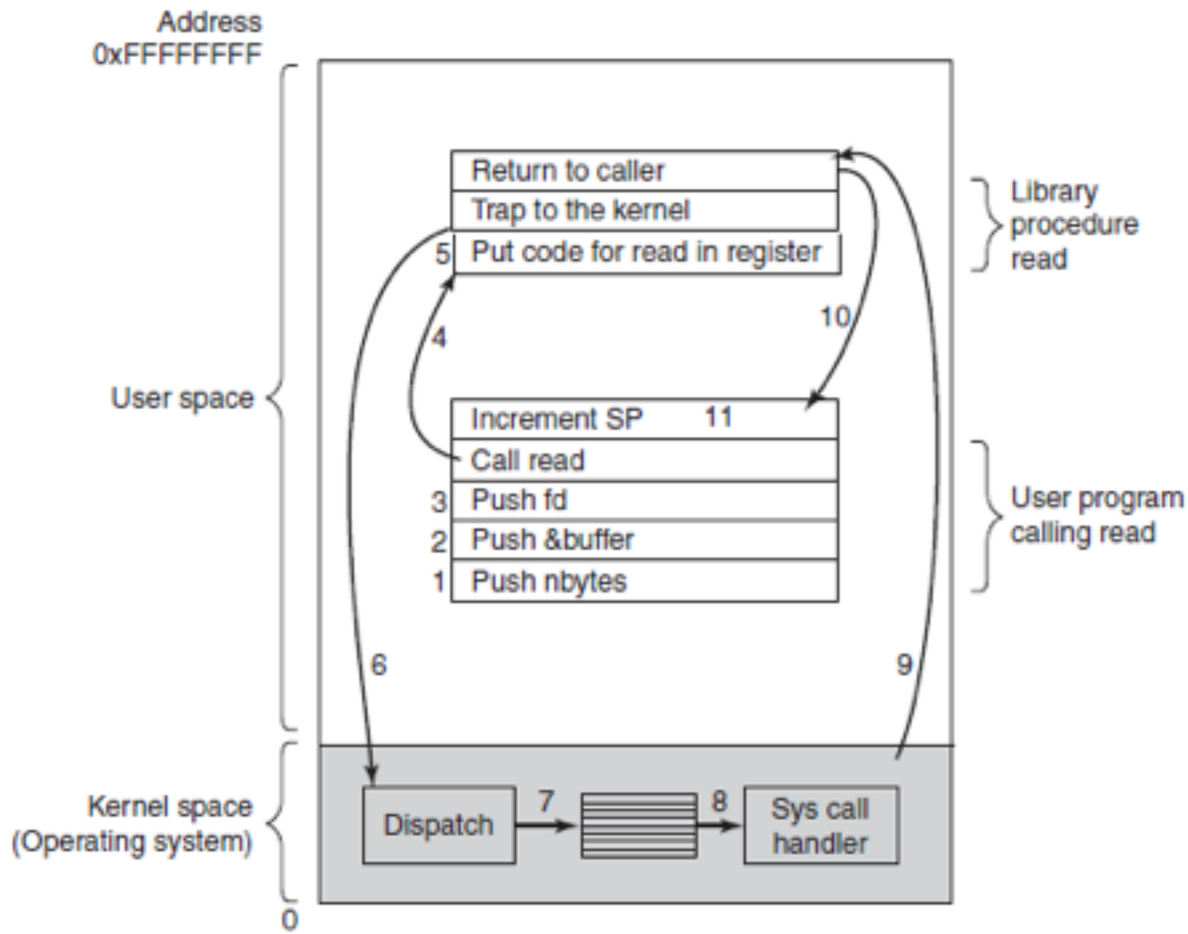


Figure 1-17. The 11 steps in making the system call *read(fd, buffer, nbytes)*.

System Calls (1)

- We will examine some of the most heavily used POSIX system calls
- Since the resource management on personal computers is minimal, the services offered by these calls determine most of what the operating system has to do
- They are grouped in four categories and include things like creating and terminating processes, creating, deleting, reading, and writing files, managing directories, and performing input and output

System Calls (2)

- Four categories of system calls are:
  - System Calls for Process Management
  - System Calls for File Management
  - System Calls for Directory Management
  - Miscellaneous System Calls

End

Week 03 – Lecture 1

References

- Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013 Prentice-Hall, Inc.