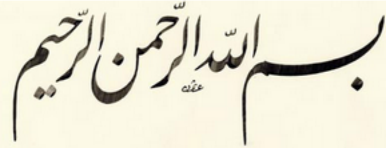

بسم الله الرحمن الرحیم

Introduction to Operating Systems

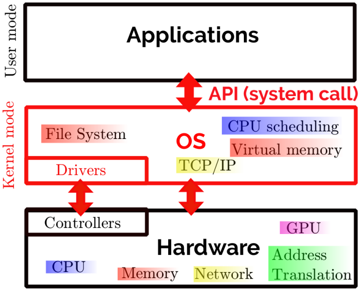
Introduction

What is operating system?

Operating System (OS) is a software that

- sits between application programs and hardware
- **makes the life easy** for programmers (hides hardware roughness)
  - OS Provides API to applications: system calls
- **shares** and **manges** physical resources among different programs: CPU, memory, ...
  - OS is sometimes known as a resource manager

- **Protects/isolates** programs from each other, while providing the means for **inter-communication**



Virtualization

The primary way that the OS can do its jobs is through **virtualization (abstraction/illusion)**.

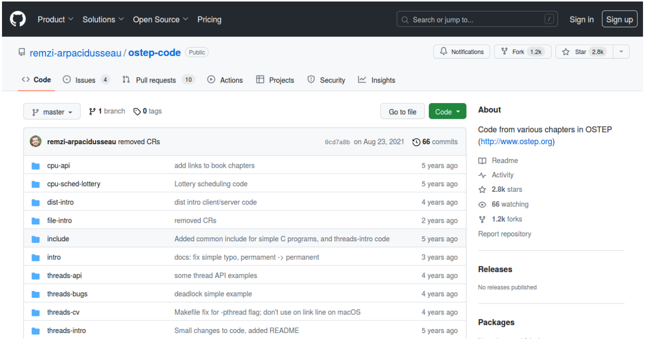
- the OS takes a physical resource (such as the processor, or memory, or a disk) and transforms it into a more general, powerful, and easy-to-use virtual form of itself.
- OS sometimes is referred to as a **virtual machine**

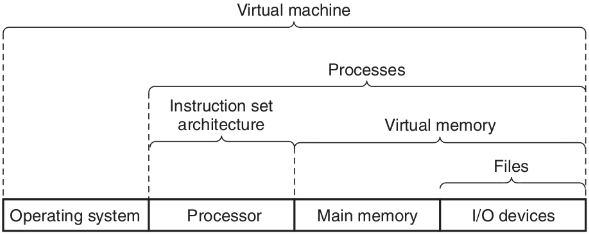
"All problems in computer science can be solved by another level of indirection/abstraction" (the "fundamental theorem of software engineering")–David Wheeler

- *Visit this link specially point 6 ☺*

In other words, system software can be simplified and verified by organizing the functions as a hierarchy that can make only downward calls and upward returns.

```
$ git clone
https://github.com/remzi-arpacidusseau/ostep-code.git
```





- **Process**: OS abstraction of the processor, main memory, and I/O devices for a running programs.
  - Multiple processes can concurrently run, each thinking itself as the exclusive user of the hardware.
- **Virtual memory**: OS abstraction of the program memory (main memory and disk).
  - Each process perceives the same picture of memory used only by itself (its address space).
- **File**: OS abstraction of I/O devices
  - All input and output in the system is performed by reading and writing files.

Virtualization Demo– the CPU

```c
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];

    while (1) {
        printf("%s\n", str);
        Spin(1);
    }
    return 0;
}
```

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

Simple Example: Code That Loops And Prints (cpu.c)

```
prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
...
```

Even though we have only one processor, somehow all four of these programs seem to be running at the same time! How does this magic (illusion) happen?

Turning a single CPU (or a small set of them) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once is what we call **virtualizing the CPU**.

Questions

- How to implement CPU time sharing? $\rightarrow$ a **mechanism** question
- which process to run $\rightarrow$ a **policy** question

Virtualization Demo– the memory

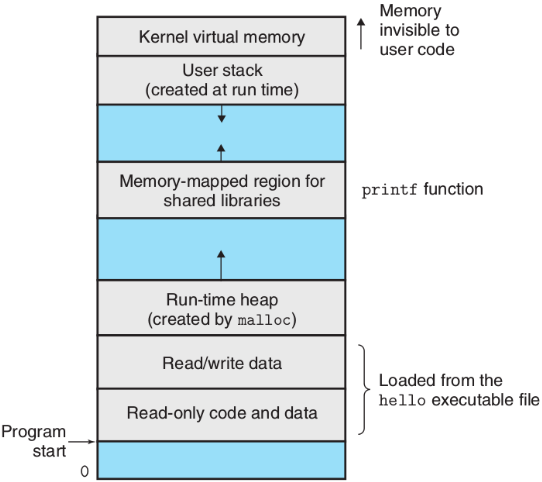
```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: mem <value>\n");
        exit(1);
    }
    int *p;
    p = malloc(sizeof(int));
    assert(p != NULL);
    printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);
    *p = atoi(argv[1]); // assign value to addr stored in p
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(%d) value of p: %d\n", getpid(), *p);
    }
    return 0;
}
```

A Program That Accesses Memory (mem.c)

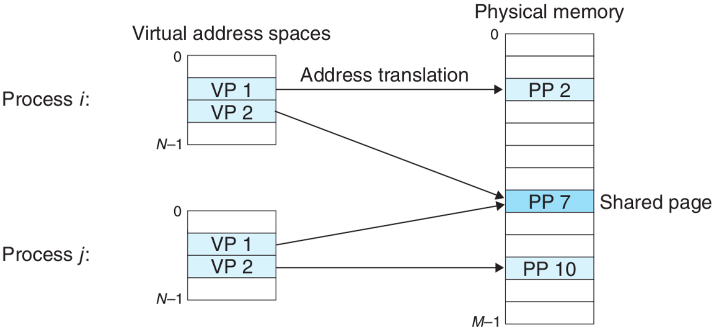
```
~/Desktop/os/remzi/codes/ostep-code/intro(master)$ gcc mem.c -o mem
~/Desktop/os/remzi/codes/ostep-code/intro(master)$ setarch $(uname --machine) --addr-no-randomize /bin/bash
~/Desktop/os/remzi/codes/ostep-code/intro(master)$ ./mem 1 & ./mem 100
[1] 6836
(6836) addr pointed to by p: 0x602010
(6837) addr pointed to by p: 0x602010
(6836) value of p: 2
(6837) value of p: 101
(6836) value of p: 3
(6837) value of p: 102
(6836) value of p: 4
(6837) value of p: 103
(6836) value of p: 5
(6837) value of p: 104
(6836) value of p: 6
(6837) value of p: 105
(6836) value of p: 7
(6837) value of p: 106
```

- each running program has allocated memory at the same address (`0x602010`), and yet each seems to be updating the value at `0x602010` independently!

Process virtual address space. (The regions are not drawn to scale.)



How VM provides processes with separate address spaces. The operating system maintains a separate page table for each process in the system.



The Crux of the Problem: How To Virtualize Resources

One central question we will answer in this book is quite simple: how does the operating system virtualize resources? This is the crux of our problem. *Why* the OS does this is not the main question, as the answer should be obvious: it makes the system easier to use. Thus, we focus on the *how*: what mechanisms and policies are implemented by the OS to attain virtualization? How does the OS do so efficiently? What hardware support is needed?

» Imagination is the basis of thinking

Concurrency

- Another main theme of this course is **concurrency**: a host of problems that arise, and must be addressed, when working on many things at once (i.e., concurrently) in the same program.

Concurrency Demo

```c
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "common_threads.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <loops>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

A Multi-threaded Program (threads.c)

```
~/Desktop/os/remzi/codes/ostep-code/intro(master)$ gcc threads.c -lpthread -o thread
~/Desktop/os/remzi/codes/ostep-code/intro(master)$ ./thread 1000
Initial value : 0
Final value   : 2000
~/Desktop/os/remzi/codes/ostep-code/intro(master)$ ./thread 1000000
Initial value : 0
Final value   : 1043349
~/Desktop/os/remzi/codes/ostep-code/intro(master)$ ./thread 1000000
Initial value : 0
Final value   : 1217516
```

```
~/Desktop/os/remzi/codes/ostep-code/intro(master)$ gcc threads.c -lpthread -o thread
~/Desktop/os/remzi/codes/ostep-code/intro(master)$ ./thread 1000
Initial value : 0
Final value   : 2000
~/Desktop/os/remzi/codes/ostep-code/intro(master)$ ./thread 1000000
Initial value : 0
Final value   : 1043349
~/Desktop/os/remzi/codes/ostep-code/intro(master)$ ./thread 1000000
Initial value : 0
Final value   : 1217516
```

Reason:

- One instruction at a time
- `counter++` takes three instructions:
  1. loads the value of the counter from memory into a register,
  2. increments it
  3. stores it back into memory.
- these three instructions do not execute **atomically** (all at once)

THE CRUX OF THE PROBLEM: HOW TO BUILD CORRECT CONCURRENT PROGRAMS

When there are many concurrently executing threads within the same memory space, how can we build a correctly working program? What primitives are needed from the OS? What mechanisms should be provided by the hardware? How can we use them to solve the problems of concurrency?

Persistence

- The third major theme of the course is **persistence**.
  - In system memory, data can be easily lost, as devices such as DRAM store values in a volatile manner; when power goes away or the system crashes, any data in memory is lost.
  - Thus, we need hardware and software to be able to store data **persistently**; such storage is thus critical to any system as users care a great deal about their data.

- The hardware comes in the form of some kind of input/output or I/O device: **hard drive, solid state drives (SSDs)**
- The software in the operating system that usually manages the disk is called the **file system**; it is thus responsible for storing any files the user creates in a reliable and efficient manner on the disks of the system.
  - Unlike the abstractions provided by the OS for the CPU and memory, the OS does not create a private, virtualized disk for each application.

Example

```c
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    assert(fd >= 0);
    char buffer[20];
    sprintf(buffer, "hello world\n");
    int rc = write(fd, buffer, strlen(buffer));
    assert(rc == (strlen(buffer)));
    fsync(fd);
    close(fd);
    return 0;
}
```

- Three system calls: open, write, close
  - routed to the part of the operating system called the **file system**
- the OS is sometimes seen as a **standard library**

**THE CRUX OF THE PROBLEM:
HOW TO STORE DATA PERSISTENTLY**

The file system is the part of the OS in charge of managing persistent data. What techniques are needed to do so correctly? What mechanisms and policies are required to do so with high performance? How is reliability achieved, in the face of failures in hardware and software?

Design Goals

- **Abstraction**: make it easy to use
  - everywhere with different levels: C, Assembly, logic gates, transistors
- **Performance** (minimize the overheads)
  - overheads: extra time, extra space
- **Protection**
  - between applications, as well as between the OS and applications
  - through **isolation**
- **Reliability** (running non-stop)
- Other goals : **security**, **energy-efficiency**, **mobility**

History

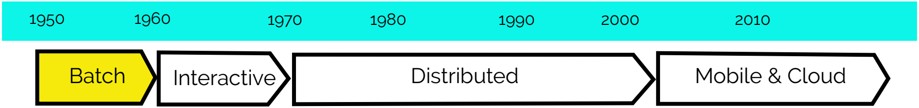
Evolution

Once upon a time ...

no operating system

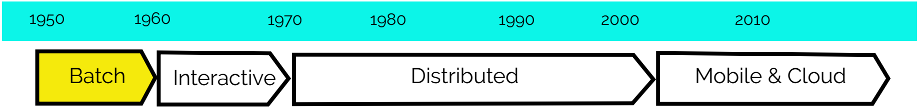
- just some libraries
- Open shop
- programmer= operator
- reservation
- setup time (for compiler, ...)

$\boxed{\textbf{\textit{IBM 7094}}\ \textit{(1959): 2 M\$} \rightarrow \textit{45 \$/hour}}$



Automatic job sequencing

- Closed shop
- professional operators
- Batching similar jobs

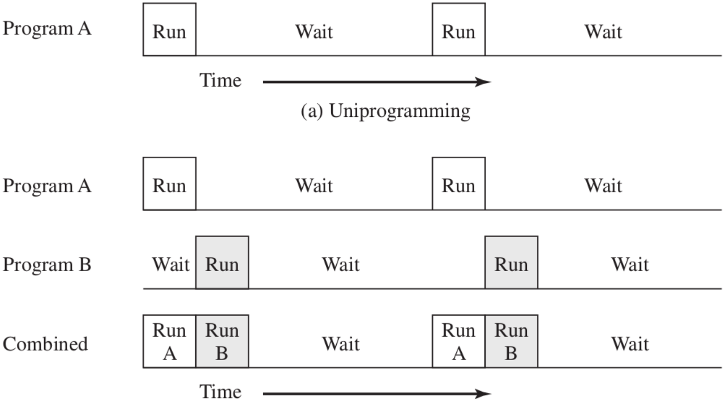


Multi programming batch systems

- System call
- Memory management

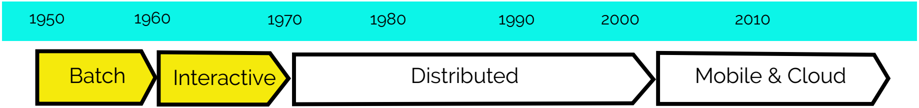
Atlas** computer at Manchester university (1962): hardware interrupt, supervisor call, virtual memory (paging), spooling $\rightarrow$ **The most significant breakthrough in the history of operating systems

Avoid waiting for peripherals



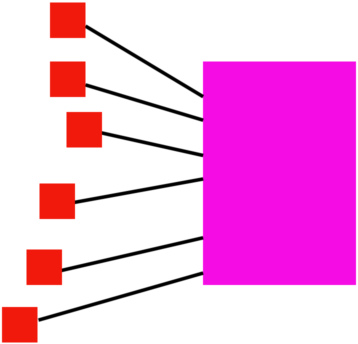
(a) Uniprogramming

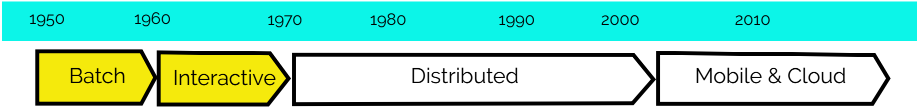
(b) Multiprogramming with two programs



Interactive/time sharing

Computer utility





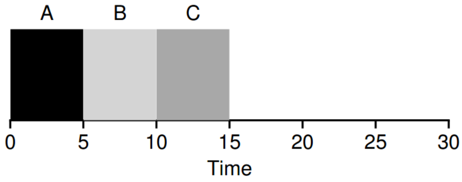
Interactive/time sharing

John McCarthy (MIT-1959)

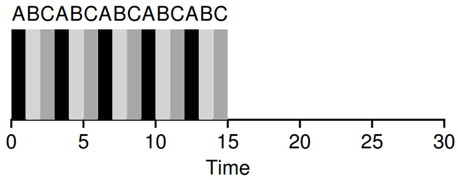
> *I want to propose an operating system for [the IBM 709] that will substantially reduce the time required to get a problem solved on the machine . . . The only way quick response can be provided at bearable cost is by time-sharing. That is, the computer must attend to other customers while one customer is reacting to some output.*

CTSS *(MIT 1962): first demonstration of time sharing by* ***Fernando Corbato***

(Batch) multiprogramming

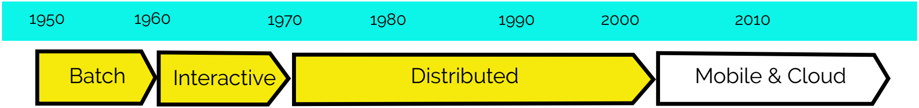


Time sharing:

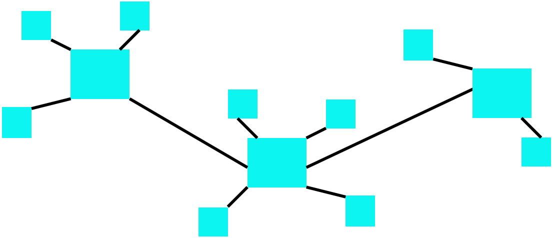


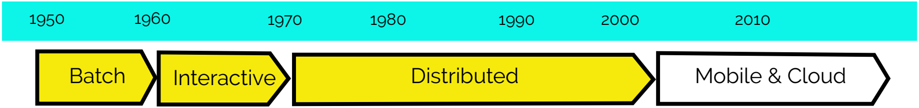
- mainframe $\rightarrow$ minicomputers
- **Multics** (MIT, Bell Labs, GE - 1966-1969): ambitious extension of CTSS to serve *computing utility* (like power utility). An OS for a fault tolerant supercomputer, with pool of CPUs and memories.
  - Practically, never used beyond MIT (a Vietnam victory)
  - Conceptually made significant contributions: hierarchical file system (with symbolic name), writing most OS in high level language, ...
- **Unix** (Bell Labs-1969): Ken Thompson took good ideas from different OSs and made them easier to use
  - A good OS for a minicomputer!
  - more discussion soon

» second-system effect



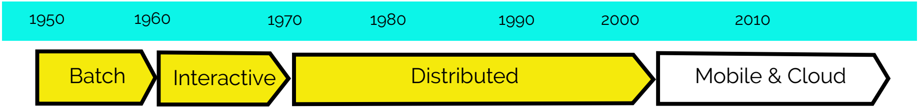
Distributed systems





Distributed systems

- minicomputers, desktop computers
  - At first, OS historical development was ignored: DOS, Mac OS(v9 and earlier): no time sharing or memory management
  - After years of suffering, historical developments found their way into PCs: Mac OS X/macOS has UNIX at its core, Windows NT
  - Even today's cell phones run operating systems (such as Linux) that are much more like what a minicomputer ran in the 1970s than what a PC ran in the 1980s



Distributed systems

- minicomputers, desktop computers

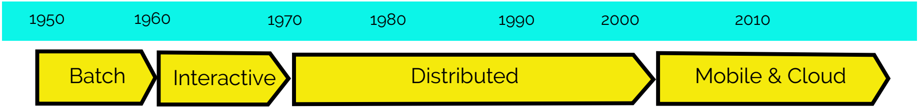
  At first, “OS interfaces with": RPC, FTP, Telnet, ...
  Then, “OS integrates with": NFS, Protocol stack (TCP/IP), Daemon processes, Browser, ...

  At first, local
  Then, Internet

1950 1960 1970 1980 1990 2000 2010

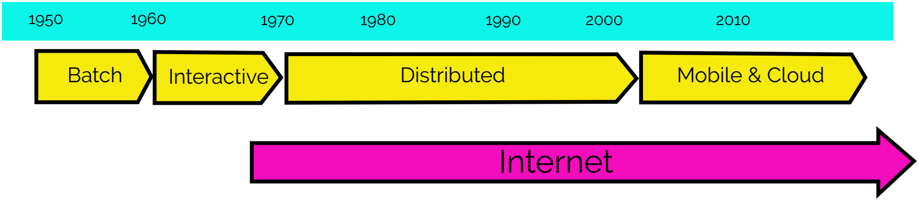
Batch → Interactive → Distributed → Mobile & Cloud

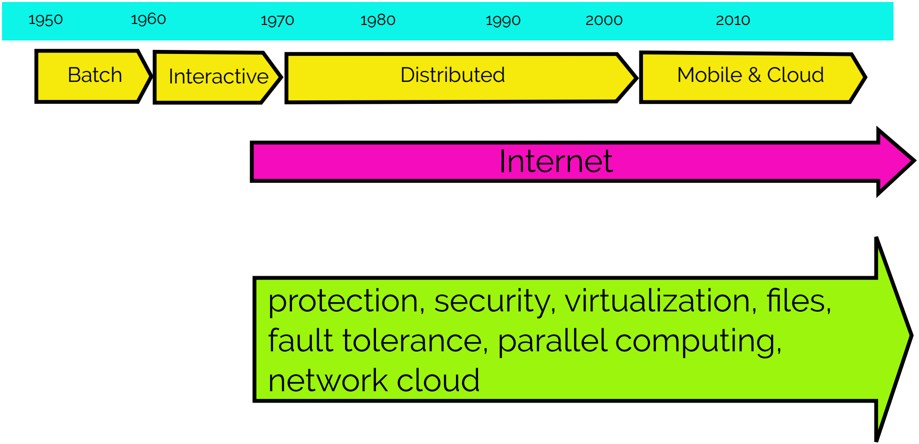
Mobile & Cloud systems

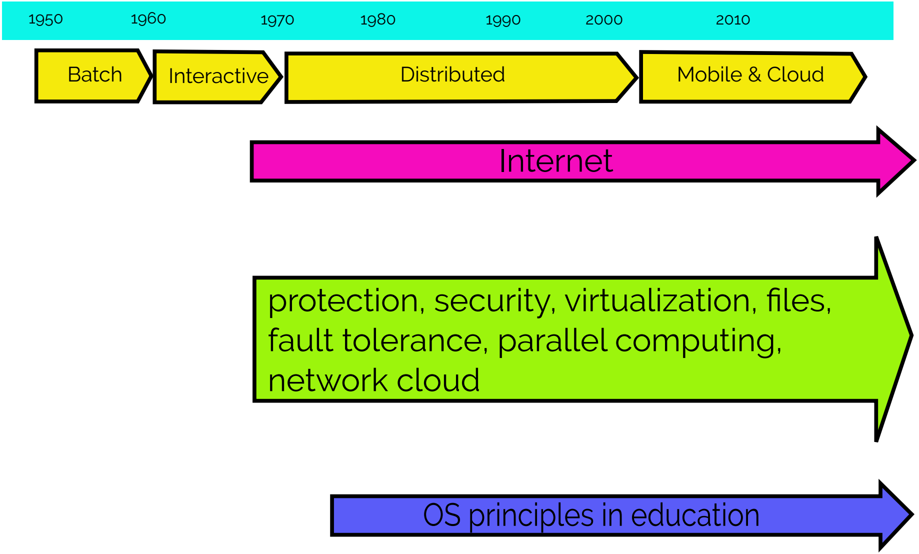


Mobile & Cloud systems

- Personalized, mobile computers
- Internet of things
- Cloud native applications

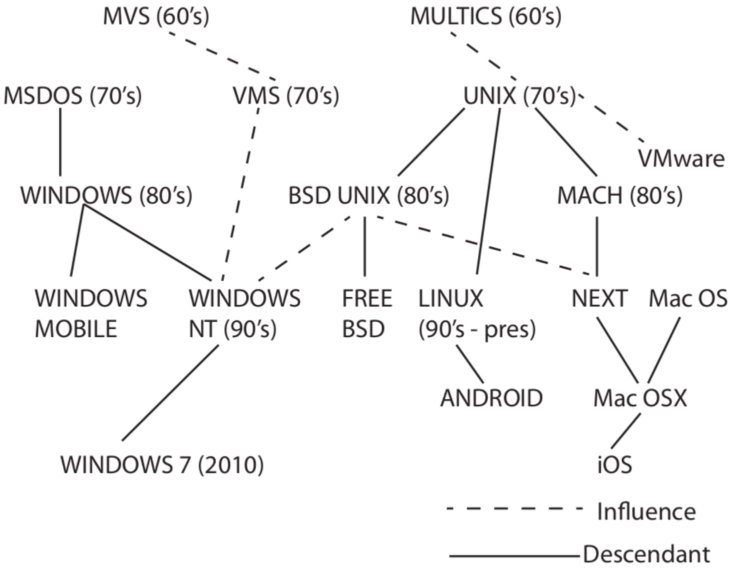

1950
1960
1970
1980
1990
2000
2010
Batch
Interactive
Distributed
Mobile & Cloud
Internet


1950
1960
1970
1980
1990
2000
2010
Batch
Interactive
Distributed
Mobile & Cloud
Internet
protection, security, virtualization, files,
fault tolerance, parallel computing,
network cloud


1950
1960
1970
1980
1990
2000
2010
Batch
Interactive
Distributed
Mobile & Cloud
Internet
protection, security, virtualization, files, fault tolerance, parallel computing, network cloud
OS principles in education

- The number of distinct new operating systems each decade is growing: 9 in 1950s to $\approx$ 350 in 2010s

  Visit wikipedia page on OS timeline:
  Timeline_of_operating_systems


MVS (60's)
MULTICS (60's)
MSDOS (70's)
VMS (70's)
UNIX (70's)
VMware
WINDOWS (80's)
BSD UNIX (80's)
MACH (80's)
WINDOWS MOBILE
WINDOWS NT (90's)
FREE BSD
LINUX (90's - pres)
NEXT
Mac OS
ANDROID
Mac OSX
WINDOWS 7 (2010)
iOS
Influence
Descendant

Unix (1969-present) @ Bell Labs

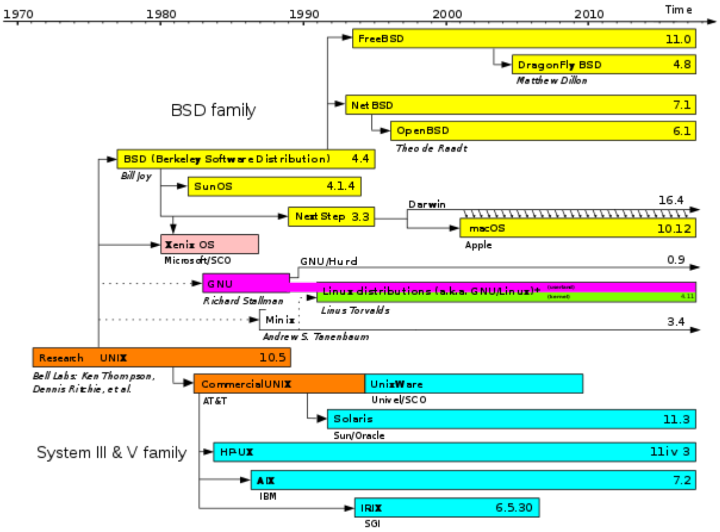
Unix (Bell Labs, 1969): by Ken Thompson and Dennis Ritchie

- not easy to convince Bell Labs to invest more on OS after Multics
- but strong ideas find their ways ...
- Ken Thompson started writing a game for a minicomputer PDP-7, a disk scheduling algorithm, and finally **an OS to test**
- the new OS: Just a 3 weak project (undisturbed! Ken's wife went on a three-week vacation): ≈ 1000 line of codes
- Ritchie's contribution by inventing the C language made Unix semi-portable (97% C, 3% assembly)
- Multics experience backed Ken and Den to pack good ideas into Unix
  - obeying a Unix philosophy: do one thing simple but well, combine few simple primitives to do more complex tasks (D&C), ...

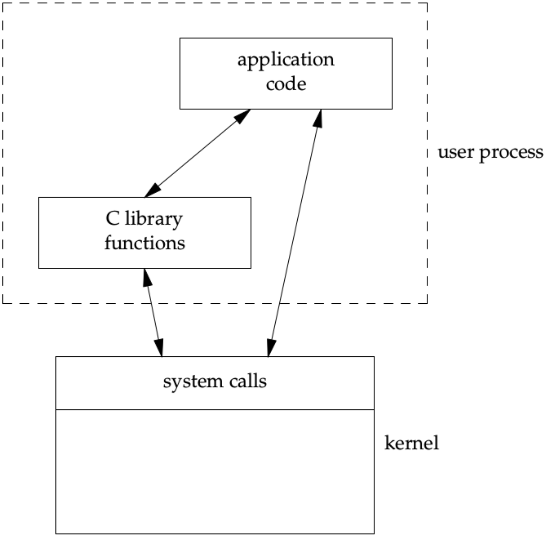
- Several internal versions improving **programming environment**: improving shell, shell script, utilities (sed, awk, ...)
  - Unix first audience (at that time): programmers
- 1975- AT&T began licensing Unix (version 6) to universities, included the source code.
  - UCB, with DARPA funding, contribution (such as adding TCP/IP protocol stack) $\rightarrow$ BSD
- Since then many variants of Unix developed with value-added features or fundamental reimplementation
  - reimplementation bypassed the AT&T license
- A need for standardization to increase portability $\rightarrow$ ISO/ANSI C, POSIX

Unix success reasons:

- mostly (97%) written in C rather than in assembly language
- distributed in source code
- Unix programming environment
  - Unix philosophy (to have small number of simple primitives and combine them to do complex jobs (**divide and conquer**), I/O redirection, ...)
  - Introducing Shell, Shell script, everything as a file



*The penetration of GNU utilities varies between distributions, some projects use GNU's implementation of the Linux kernel (Linux-libre). Some operating systems mentioned here include GNU utilities to a lesser degree.


application
code
user process
C library
functions
system calls
kernel

ISO C (ANSI C): to provide portability of conforming C programs to a wide variety of operating systems, not only the UNIX System.

Headers defined by the ISO C standard

Header	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
<assert.h>	•	•	•	•	verify program assertion
<complex.h>	•	•	•	•	complex arithmetic support
<ctype.h>	•	•	•	•	character classification and mapping support
<errno.h>	•	•	•	•	error codes (Section 1.7)
<fenv.h>	•	•	•	•	floating-point environment
<float.h>	•	•	•	•	floating-point constants and characteristics
<inttypes.h>	•	•	•	•	integer type format conversion
<iso646.h>	•	•	•	•	macros for assignment, relational, and unary operators
<limits.h>	•	•	•	•	implementation constants (Section 2.5)
<locale.h>	•	•	•	•	locale categories and related definitions
<math.h>	•	•	•	•	mathematical function and type declarations and constants
<setjmp.h>	•	•	•	•	nonlocal goto (Section 7.10)
<signal.h>	•	•	•	•	signals (Chapter 10)
<stdarg.h>	•	•	•	•	variable argument lists
<stdbool.h>	•	•	•	•	Boolean type and values
<stddef.h>	•	•	•	•	standard definitions
<stdint.h>	•	•	•	•	integer types
<stdio.h>	•	•	•	•	standard I/O library (Chapter 5)
<stdlib.h>	•	•	•	•	utility functions
<string.h>	•	•	•	•	string operations
<tgmath.h>	•	•	•	•	type-generic math macros
<time.h>	•	•	•	•	time and date (Section 6.10)
<wchar.h>	•	•	•	•	extended multibyte and wide character support
<wctype.h>	•	•	•	•	wide character classification and mapping support

Why do we need system calls besides the C?
sometimes necessary to employ system calls for maximum efficiency, or to access some facility that is not in the library.

POSIX (Portable Operating System Interface):

- to promote the portability of applications among various UNIX System environments.

Using Posix programming:

- Benefit: portability
- Cost: missing performance (which otherwise achieved by using hardware specific features)

POSIX includes the ISO C plus the following headers:

Header	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
<aio.h>	•	•	•	•	asynchronous I/O
<cpio.h>	•	•	•	•	cpio archive values
<dirent.h>	•	•	•	•	directory entries (Section 4.22)
<dlfcn.h>	•	•	•	•	dynamic linking
<fcntl.h>	•	•	•	•	file control (Section 3.14)
<fnmatch.h>	•	•	•	•	filename-matching types
<glob.h>	•	•	•	•	pathname pattern-matching and generation
<grp.h>	•	•	•	•	group file (Section 6.4)
<iconv.h>	•	•	•	•	codeset conversion utility
<langinfo.h>	•	•	•	•	language information constants
<monetary.h>	•	•	•	•	monetary types and functions
<netdb.h>	•	•	•	•	network database operations
<nl_types.h>	•	•	•	•	message catalogs
<poll.h>	•	•	•	•	poll function (Section 14.4.2)
<pthread.h>	•	•	•	•	threads (Chapters 11 and 12)
<pwd.h>	•	•	•	•	password file (Section 6.2)
<regex.h>	•	•	•	•	regular expressions
<sched.h>	•	•	•	•	execution scheduling
<semaphore.h>	•	•	•	•	semaphores
<strings.h>	•	•	•	•	string operations
<tar.h>	•	•	•	•	tar archive values
<termios.h>	•	•	•	•	terminal I/O (Chapter 18)
<unistd.h>	•	•	•	•	symbolic constants
<wordexp.h>	•	•	•	•	word-expansion definitions

<arpa/inet.h>	•	•	•	•	Internet definitions (Chapter 16)
<net/if.h>	•	•	•	•	socket local interfaces (Chapter 16)
<netinet/in.h>	•	•	•	•	Internet address family (Section 16.3)
<netinet/tcp.h>	•	•	•	•	Transmission Control Protocol definitions
<sys/mman.h>	•	•	•	•	memory management declarations
<sys/select.h>	•	•	•	•	select function (Section 14.4.1)
<sys/socket.h>	•	•	•	•	sockets interface (Chapter 16)
<sys/stat.h>	•	•	•	•	file status (Chapter 4)
<sys/statvfs.h>	•	•	•	•	file system information
<sys/times.h>	•	•	•	•	process times (Section 8.17)
<sys/types.h>	•	•	•	•	primitive system data types (Section 2.8)
<sys/un.h>	•	•	•	•	UNIX domain socket definitions (Section 17.2)
<sys/utsname.h>	•	•	•	•	system name (Section 6.9)
<sys/wait.h>	•	•	•	•	process control (Section 8.6)

Historical Lessons

- ☑ The second system effect
- ☑ Limits are good
- ☑ Strong ideas find their ways
- ☑ Do simple things well and then combine

John McCarthy
(1927 – 2011)

- ► one of the founders of the discipline of artificial intelligence
- ► Popularizing time-sharing OS
- ► Turing Award for his contributions to the topic of AI (1971)

A true visionary man!