Operating System: History and Three Easy Pieces

COMP402127: Introduction to Computer Systems

Hao Li
Xi'an Jiaotong University

Today

- **History of Operating Systems**
- **Three Easy Pieces**

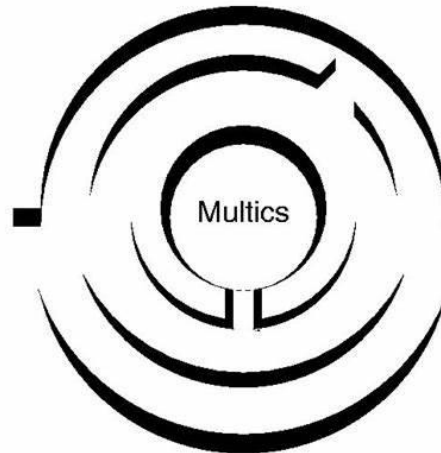# Earliest days: One batch job at a time

IBM 704 at Langley Research Center (NASA), 1957
https://commons.wikimedia.org/w/index.php?curid=6455009
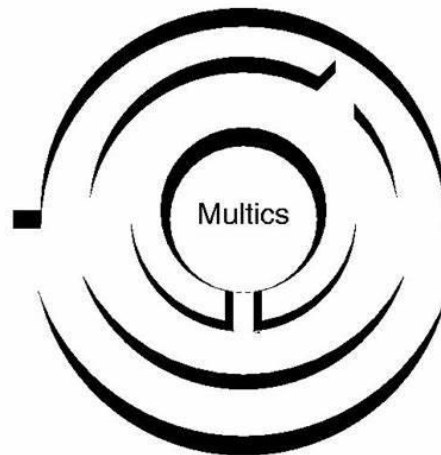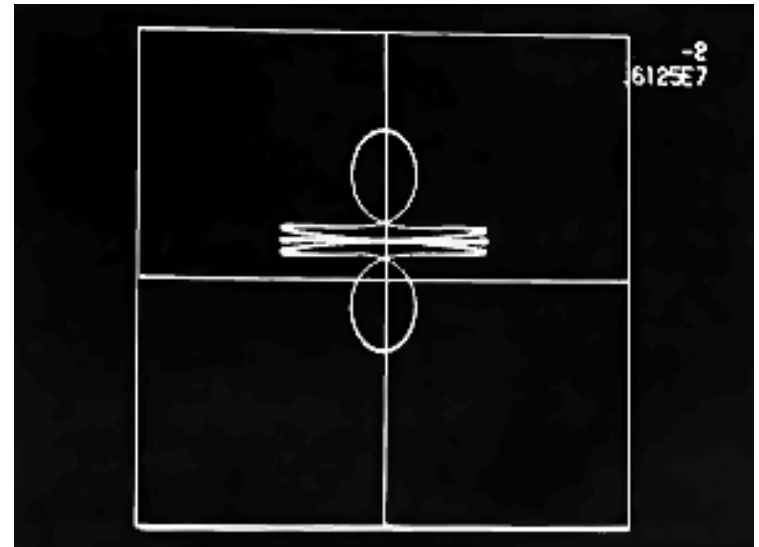
1960s: Operating System for Multitasking

1960s: MULTICS Fails

1969: BELL Labs in AT&T

Ken Thompson (1943-)

Space Travel Game
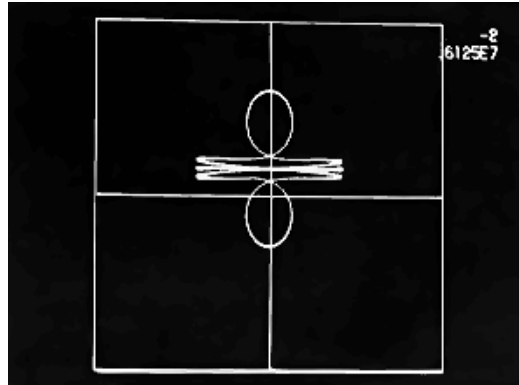
1969: BELL Labs in AT&T

Digital Equipment Corporation PDP-7

1969: How to Play Space Travel in PDP-7?

Applications

Operating System

Hardware

with Assembly

in ONE month!

1969: BELL Labs in AT&T
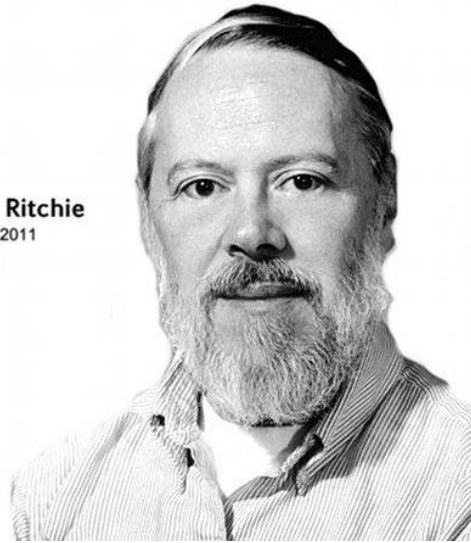
**Dennis Ritchie
(1941-2011)**

1969: BELL Labs in AT&T

Dennis Ritchie
1941-2011
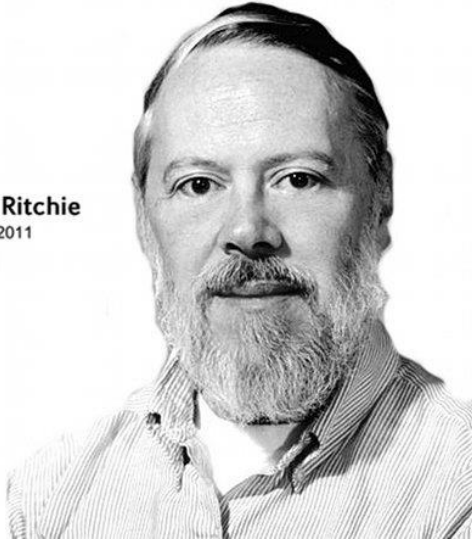
"Your OS is much worse than MULTICS"

"Call it UNICS"

1971: Re-Architect UNICS

- **B Language: extended from BCPL**
- **A high-level programming language**

Dennis Ritchie
1941-2011
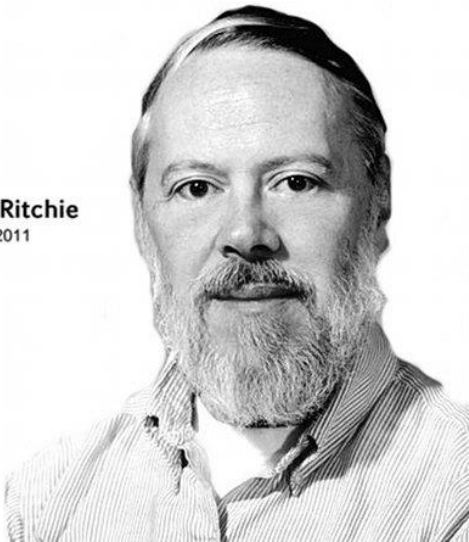
- **C Language: extended from B**
- **Decouple PL from hardware**
- **You are still using it!**

UNICS → UNIX

1973: Xerox PARC

1973: Xerox PARC
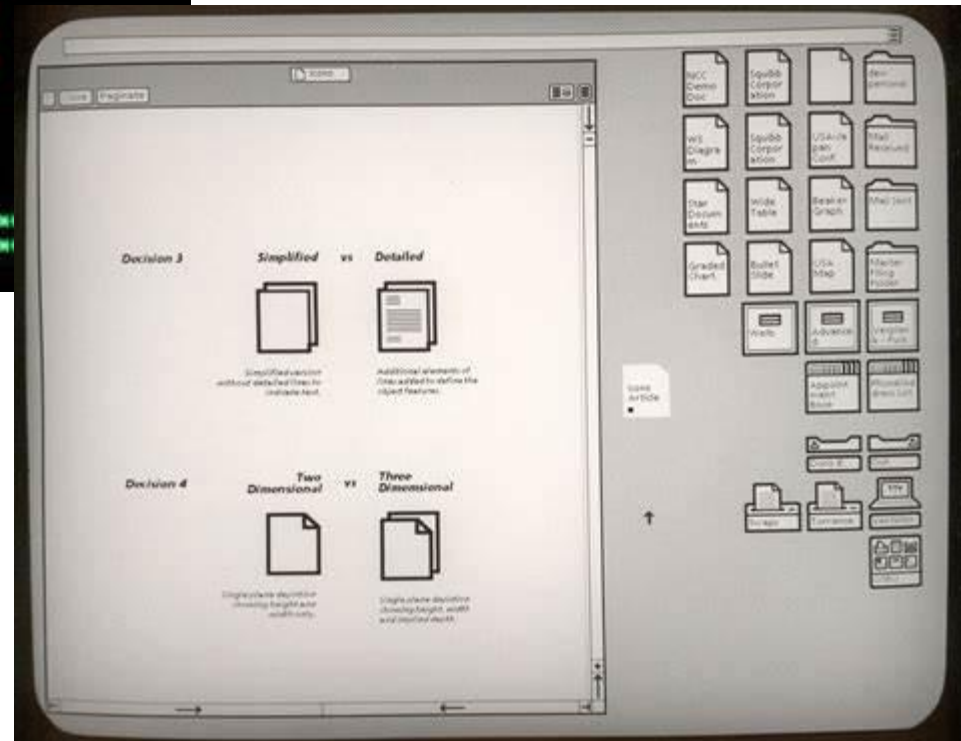
1979: Xerox PARC Alto

**Steve Jobs
(1955-2011)**

1983: Apple Lisa

1980: A Quick and Dirty Operating System

Tim Paterson (1956-)

- **Develop an OS with 4 months**
- **For 16bit Intel 8086 CPU**
- **QDOS**
  - Quick and Dirty Operating System

1980: A Quick and Dirty Operating System

**Tim Paterson
(1956-)**

**Paul Allen
(1953-2018)**

1981: Microsoft

**Bill Gates
(1955-)**

**Paul Allen
(1953-2018)**

1981: MS-DOS

$50,000

QDOS
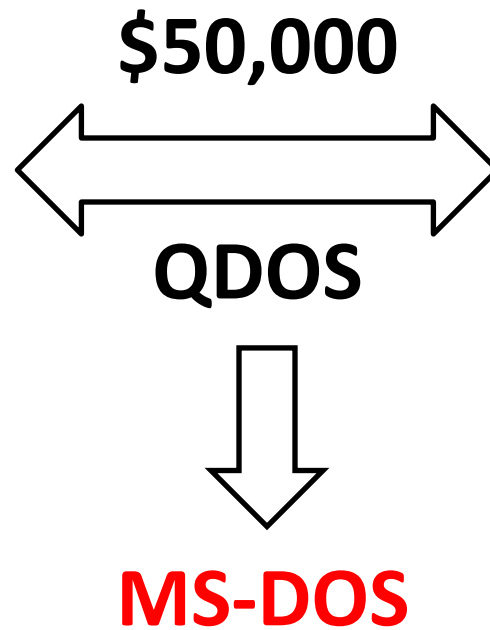
MS-DOS

1990: Should OS and Hardware Decouple?

MS-DOS:
IBM, Intel, Apple, ...

Apple:
One OS for One Machine

1990: Windows
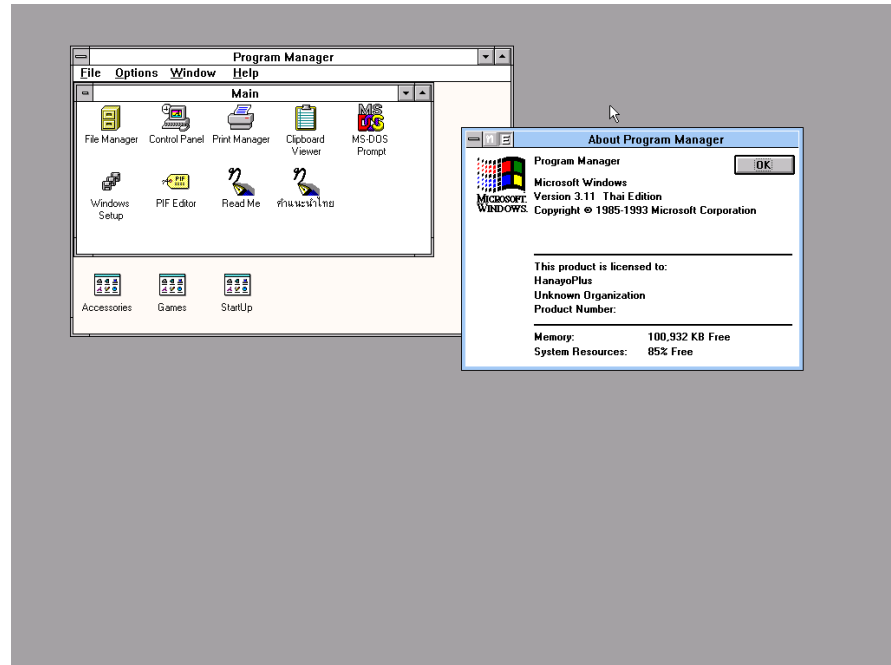
Windows 3.0

1990: Two Thieves

You Are Stealing
Our Operating System!

Well, Steve, I think there's more than one way of looking at it. I think it's more like we both had this rich neighbor named Xerox and I broke into his house to steal the TV set and found out that you had already stolen it.

1991: A New and Open OS

1987: Andy
Tanenbaum
Includes source
code for Minix
("toy" Unix)
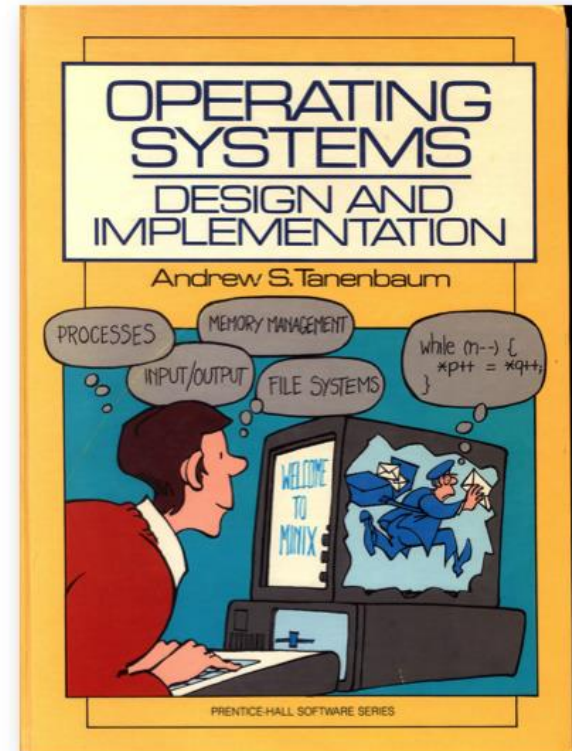
**Andrew Tanenbaum
(1944-)**

1991: A New and Open OS

Linus Torvalds
(1969-)
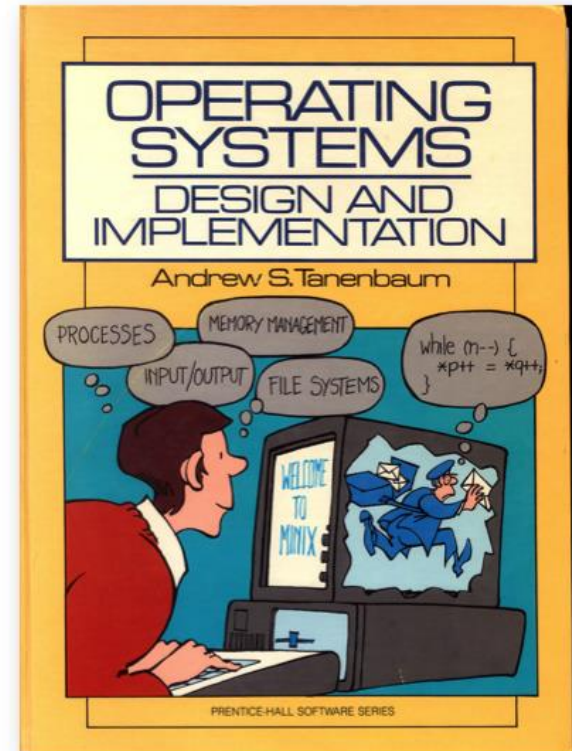
1991: The First Email

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID: <1991Aug25.205708.9541@klaava.Helsinki.FI>
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki


Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and
professional like gnu) for 386(486) AT clones.  This has been brewing
since april, and is starting to get ready.  I'd like any feedback on
things people like/dislike in minix, as my OS resembles it somewhat
(same physical layout of the file-system (due to practical reasons)
among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work.
This implies that I'll get something practical within a few months, and
I'd like to know what features most people would want.  Any suggestions
are welcome, but I won't promise I'll implement them :-)

                Linus (torvalds@kruuna.helsinki.fi)

PS.  Yes - it's free of any minix code, and it has a multi-threaded fs.
It is NOT protable (uses 386 task switching etc), and it probably never
will support anything other than AT-harddisks, as that's all I have :-(.
```

1991: Free Software

**Richard Stallman
(1953-)**

- **GPL**
  - **Copy-Left**
- **GNU**
  - **GNU is Not Unix**
  - **Emacs, gcc, gdb, …**

1991: Linux and Open Source Project

■ **Applications**

■ **Operating Systems**

Nowadays
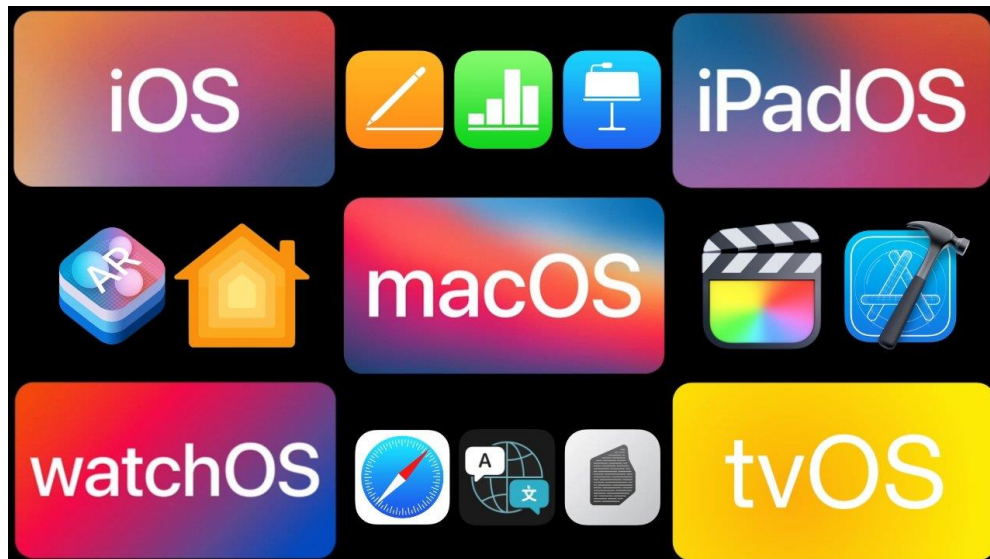
Today

- **History of Operating Systems**
- **Three Easy Pieces**

What a happens when a program runs?

■ **A running program executes instructions.**

1. The processor **fetches** an instruction from memory.

2. **Decode**: Figure out which instruction this is

3. **Execute**: i.e., add two numbers, access memory, check a condition, jump to function, and so forth.

4. The processor moves on to the **next instruction** and so on.

Operating System (OS)

■ **Responsible for**

- ▪ Making it easy to **run** programs
- ▪ Allowing programs to **share** memory
- ▪ Enabling programs to **interact** with devices

> **OS is in charge of making sure the system operates correctly and efficiently.**

1st Piece: Virtualization

- **The OS takes a physical resource and transforms it into a virtual form of itself.**
    - **Physical resource**: Processor, Memory, Disk …
  - The virtual form is more <u>general</u>, <u>powerful</u> and <u>easy-to-use</u>.
  - Sometimes, we refer to the OS as a **virtual machine**.

System call

- **System call allows user to tell the OS what to do.**
  - The OS provides some interface (APIs, standard library).
  - A typical OS exports a few hundred system calls.
    - Run programs
    - Access memory
    - Access devices

The OS is a resource manager.

- **The OS manage resources such as *CPU*, *memory* and *disk*.**
- **The OS allows**
  - Many programs to run → Sharing the <u>CPU</u>
  - Many programs to *concurrently* access their own instructions and data → Sharing <u>memory</u>
  - Many programs to access devices → Sharing <u>disks</u>

Virtualizing the CPU

- **The system has a very large number of virtual CPUs.**
    - Turning a single CPU into a <u>seemingly infinite number</u> of CPUs.
    - Allowing many programs to <u>seemingly run at once</u>
      → **Virtualizing the CPU**

Virtualizing the CPU (Cont.)

```
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <sys/time.h>
4       #include <assert.h>
5       #include "common.h"
6
7       int
8       main(int argc, char *argv[])
9       {
10              if (argc != 2) {
11                      fprintf(stderr, "usage: cpu <string>\n");
12                      exit(1);
13              }
14              char *str = argv[1];
15              while (1) {
16                      Spin(1); // Repeatedly checks the time and
                                 returns once it has run for a second
17                      printf("%s\n", str);
18              }
19              return 0;
20      }
```

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

Virtualizing the CPU (Cont.)

■ **Execution result 1.**

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

Run forever; Only by pressing "Control-c" can we halt the program

Virtualizing the CPU (Cont.)

■ **Execution result 2.**

```
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
C
B
D
...
```

Even though we have only one processor, all four of programs seem to be running at the same time!

Virtualizing Memory

- **The physical memory is _an array of bytes_.**

- **A program keeps all of its data structures in memory.**

  - **Read memory** (load):

    - Specify an <u>address</u> to be able to access the data

  - **Write memory** (store):

    - Specify the data to be written to the given address

Virtualizing Memory (Cont.)

■ **A program that Accesses Memory (`mem.c`)**

```c
1          #include <unistd.h>
2          #include <stdio.h>
3          #include <stdlib.h>
4          #include "common.h"
5
6          int
7          main(int argc, char *argv[])
8          {
9                  int *p = malloc(sizeof(int));  // a1: allocate some
                                                      memory
10                 assert(p != NULL);
11                 printf("(%d) address of p: %08x\n",
12                         getpid(), (unsigned) p);  // a2: print out the
                                                        address of the memmory
13                 *p = 0;  // a3: put zero into the first slot of the memory
14                 while (1) {
15                         Spin(1);
16                         *p = *p + 1;
17                         printf("(%d) p: %d\n", getpid(), *p); // a4
18                 }
19                 return 0;
20         }
```

Virtualizing Memory (Cont.)

■ **The output of the program `mem.c`**

```
prompt> ./mem
(2134) memory address of p: 00200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

▪ The newly allocated memory is at address `00200000`.

▪ It updates the value and prints out the result.

Virtualizing Memory (Cont.)

■ **Running `mem.c` multiple times**

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) memory address of p: 00200000
(24114) memory address of p: 00200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
...
```

▪ It is as if each running program has its **own private memory**.

  ▪ Each running program has allocated memory at <u>the same address</u>.

  ▪ Each seems to be updating the value at `00200000` independently.

Virtualizing Memory (Cont.)

- **Each process accesses its own private virtual address space.**
    - The OS maps address space onto the physical memory.
    - A memory reference within one running program <u>does not affect</u> the address space of other processes.
    - Physical memory is a <u>shared resource</u>, managed by the OS.

2nd Piece: Concurrency

- **The OS is juggling many things at once, first running one process, then another, and so forth.**

- **Modern multi-threaded programs also exhibit the concurrency problem.**

Concurrency Example

■ **A Multi-threaded Program (`thread.c`)**

```c
1          #include <stdio.h>
2          #include <stdlib.h>
3          #include "common.h"
4
5          volatile int counter = 0;
6          int loops;
7
8          void *worker(void *arg) {
9                  int i;
10                 for (i = 0; i < loops; i++) {
11                         counter++;
12                 }
13                 return NULL;
14         }
15
16         int
17         main(int argc, char *argv[])
18         {
19                 if (argc != 2) {
20                         fprintf(stderr, "usage: threads <value>\n");
21                         exit(1);
22                 }
```

Concurrency Example (Cont.)

```
23                 loops = atoi(argv[1]);
24                 pthread_t p1, p2;
25                 printf("Initial value : %d\n", counter);
26
27                 Pthread_create(&p1, NULL, worker, NULL);
28                 Pthread_create(&p2, NULL, worker, NULL);
29                 Pthread_join(p1, NULL);
30                 Pthread_join(p2, NULL);
31                 printf("Final value : %d\n", counter);
32                 return 0;
33         }
```

- The main program creates **two threads**.

  - <u>Thread</u>: a function running within the same memory space. Each thread start running in a routine called `worker()`.

  - `worker()`: increments a counter

Concurrency Example (Cont.)

- **`loops` determines how many times each of the two workers will increment the shared counter in a loop.**
  - `loops`: 1000.

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value : 2000
```

  - `loops`: 100000.

```
prompt> ./thread 100000
Initial value : 0
Final value : 143012 // huh??
prompt> ./thread 100000
Initial value : 0
Final value : 137298 // what the??
```

Why is this happening?

- **Increment a shared counter → take three instructions.**
    1. Load the value of the counter from memory into register.
    2. Increment it
    3. Store it back into memory

- **These three instructions do not execute atomically. → Problem of concurrency happen.**

3rd Piece: Persistence

- **Devices such as DRAM store values in a <u>volatile</u>.**

- *Hardware* **and** *software* **are needed to store data persistently.**

  - **Hardware**: I/O device such as a hard drive, solid-state drives(SSDs)

  - **Software**:

    - File system manages the disk.

    - File system is responsible for <u>storing any files</u> the user creates.

Persistence (Cont.)

- **Create a file (`/tmp/file`) that contains the string "hello world"**

```
1       #include <stdio.h>
2       #include <unistd.h>
3       #include <assert.h>
4       #include <fcntl.h>
5       #include <sys/types.h>
6
7       int
8       main(int argc, char *argv[])
9       {
10              int fd = open("/tmp/file", O_WRONLY | O_CREAT
                            | O_TRUNC, S_IRWXU);
11              assert(fd > -1);
12              int rc = write(fd, "hello world\n", 13);
13              assert(rc == 13);
14              close(fd);
15              return 0;
16      }
```

`open()`, `write()`, and `close()` system calls are routed to the part of OS called the file system, which handles the requests

Persistence (Cont.)

- **What OS does in order to write to disk?**
  - Figure out **where** on disk this new data will reside
  - **Issue I/O** requests to the underlying storage device

- **File system handles system crashes during write.**
  - **Journaling** or **copy-on-write**
  - Carefully <u>ordering</u> writes to disk

Design Goals

- **Build up abstraction**
  - Make the system convenient and easy to use.

- **Provide high performance**
  - Minimize the overhead of the OS.
  - OS must strive to provide virtualization <u>without excessive overhead</u>.

- **Protection between applications**
  - <u>Isolation</u>: Bad behavior of one does not harm other and the OS itself.

Design Goals (Cont.)

- **High degree of reliability**
  - The OS must also run non-stop.

- **Other issues**
  - Energy-efficiency
  - Security
  - Mobility