System Calls

CPSC 457: Principles of Operating Systems
Winter 2024

Contains slides from Pavol Federl, Mea Wang, Andrew Tanenbaum and Herbert Bos, Silberschatz, Galvin and Gagne

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Tuesday, 28 November 2024

UNIVERSITY OF CALGARY

Topics

- Kernel
- System Calls
- Libraries
- Examples
  - C/Win32/Unix
- Unix APIs
- Timing
- Tracing
  - strace

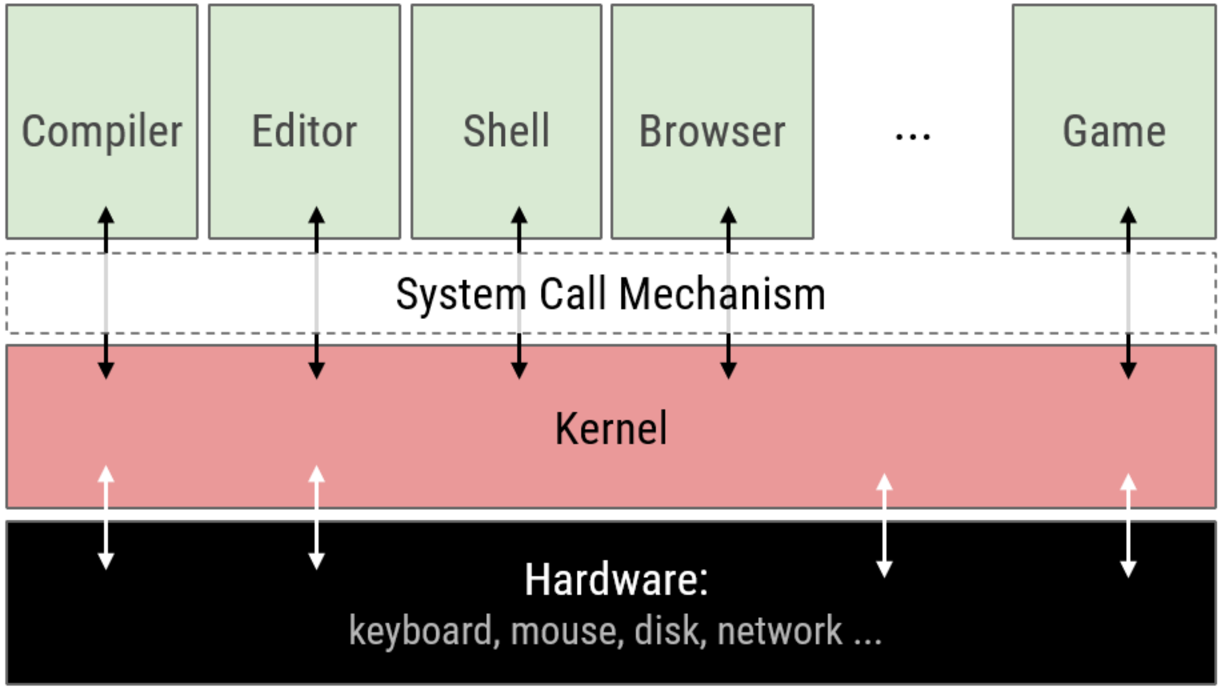
UNIVERSITY OF CALGARY

Kernel

Kernel services

- OS provides services to applications, e.g. access to hardware
- these services are accessible through **system calls**
  - often implemented using software interrupts (**traps**) or similar mechanisms
  - recall that traps allow for a safe way to switch CPU from **user-mode to kernel-mode**

Kernel vs. user mode



user mode - no access to hardware, limited instruction set

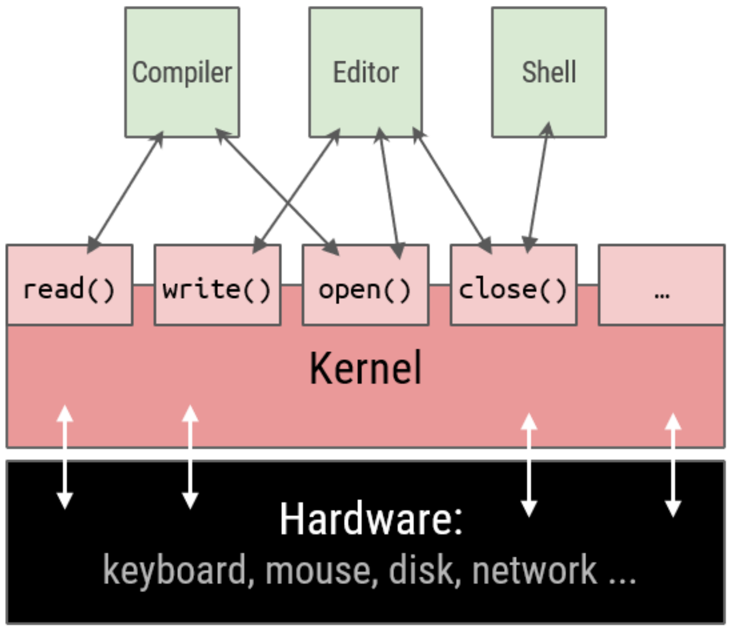
kernel mode - full access to hardware, full instruction set

System Calls

System calls

- to access a service / resource of the system, applications must make system calls
- system calls implemented using special instruction (e.g. software interrupt) that safely switch from user mode to kernel mode and then execute a kernel routine
- inside kernel routine:
  1. kernel saves application state, e.g. registers
  2. kernel performs the requested operation, e.g. involving some hardware
     - if operation takes a while, kernel suspends the application until the operation is finished, and gives CPU to another process in the meantime
  3. after operation is done, kernel switches back to user mode and restores application state, i.e. resumes application
- from application's perspective, making a system call is just like calling a library function, but the call may take quite a long time before returning

System calls



- we can think of system calls as a set of APIs provided by the OS for all applications
- system calls are different on different operating systems, but they are many similarities
- OSes often need to execute 1000s of system calls per second

UNIVERSITY OF CALGARY

- Hello-World in assembly for 64-bit Linux

```
    .global _start
    .text
_start:
    mov     $1, %rax            # system call #1 → write
    mov     $1, %rdi            # fd = 1 → stdout
    mov     $msg, %rsi          # address of first byte
    mov     $13, %rdx           # string length
    syscall                     # system call

    mov     $60, %rax           # system call #60 → exit
    xor     %rdi, %rdi          # return code 0
    syscall                     # system call
msg:
    .ascii  "Hello, world\n"
```

- Hello-World in C

```
#include <unistd.h>
int main() {
    char * s = "Hello world\n";
    write(1, s, 12);
    return 0;
}
```

Example: copying file

- even simple programs make many system calls
- example: a program that copies a file

```
int main() {
  std::string fname1, fname2; char c;
  std::cout << "Source filename:";
  std::cin >> fname1;
  std::cout << "Destination filename:";
  std::cin >> fname2;
  int fd1 = open(fname1.c_str(), O_RDONLY);
  if (fd1 < 0) err(-1, "Could not open source file.");
  int fd2 =
open(fname2.c_str(),O_WRONLY|O_EXCL|O_CREAT);
  if (fd2 < 0) err(-1, "Could not create dest. file.");
  while (1) {
    if (read(fd1, &c, 1) <= 0) break;
    write(fd2, &c, 1);
  }
  close(fd1);
  close(fd2);
  std::cout << "Success.\n";
  exit(0);
}
```

https://replit.com/@jonathanwhudson/copy-files#main.cpp

Libraries

Libraries and system calls

- system calls are usually implemented in assembly, hand optimized for performance
  e.g. system call number and parameters passed in registers or stack

```
mov      eax,4      ; system call # (sys_write on 32bit Linux)
mov      ebx,1      ; fd = stdout
mov      edx,4      ; message length
mov      ecx,msg    ; ptr to message
int      0x80       ; trap
```

- http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/
- system calls are cumbersome to invoke from higher level languages
- it is much easier (and common) to make system calls through higher-level **wrapper functions**
- on Unix-like systems:
  `libc` (C library), **`libstdc++`** or **`libc++`** (C++ library)

```
write(fd, buff, len); // write() is a C/C++ wrapper for system call sys_write
```

Libraries and system calls

- system call wrappers hide the implementation details of system calls
  e.g. convert parameters from stack into registers, and vice versa
- extra benefits of using system call wrappers:
  - an application using wrappers can compile and run on any system
    that supports the same wrapper APIs
  - if the system call ever changes / is deprecated, the program using the wrapper
    could still continue to function properly, as long as the wrapper is updated
- some common APIs:
  - **POSIX** APIs for Unix, Linux, Mac OS X
  - Win32 APIs for windows
  - Java APIs for Java virtual machine
- usually, there is a strong correlation between a wrapper and the corresponding system call, such
  as name, number and types of parameters, return value type, etc, but wrapper != system call

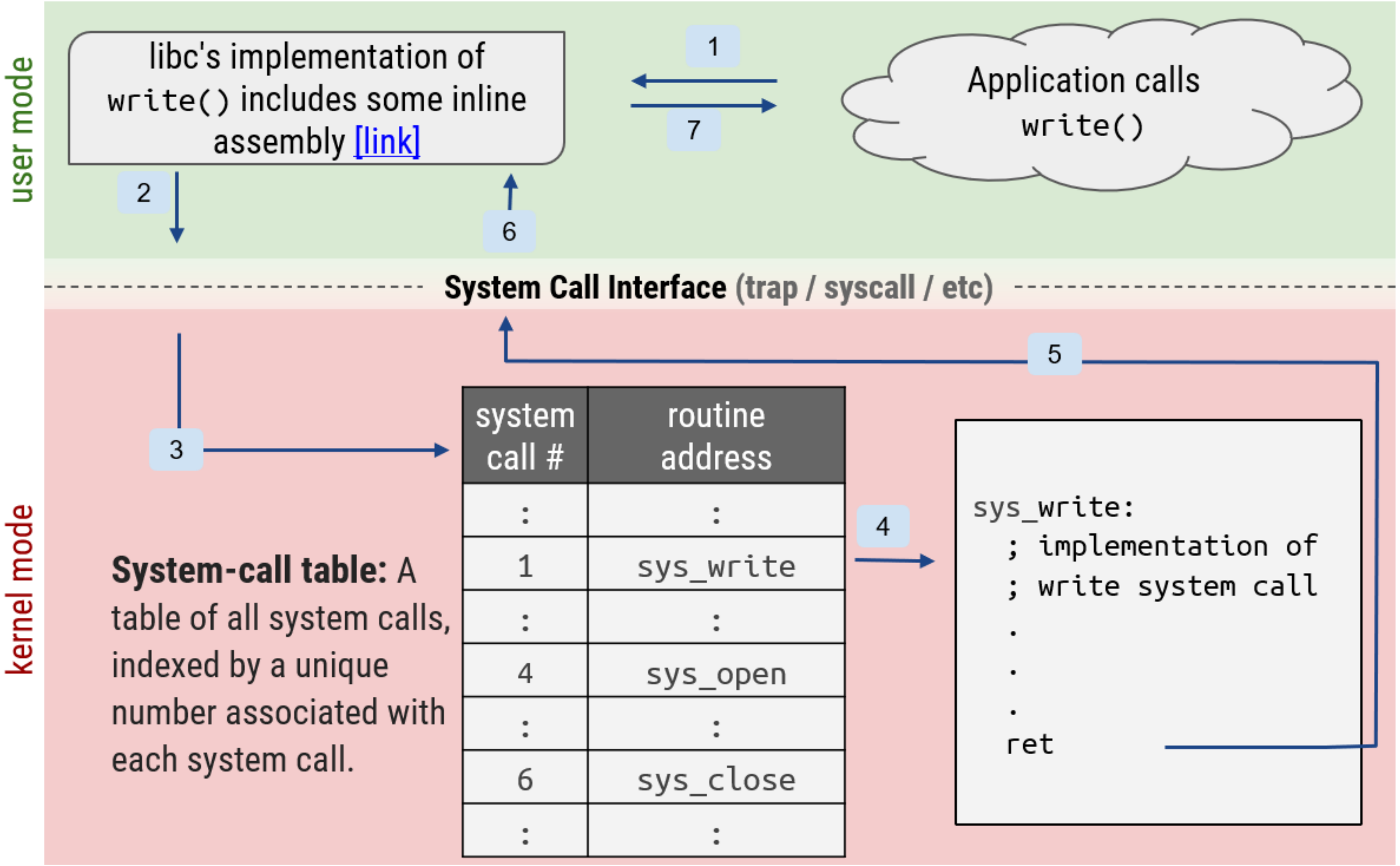
Example: write()

- standard C library provides access to many OS system calls
- for example `write()` is a wrapper for `sys_write` system call

```
$ man -s2 write
...
SYNOPSIS
       #include <unistd.h>
       ssize_t write(int fd, const void *buf, size_t count);
```

- `write()`
  - converts the arguments passed to it on the stack into appropriate registers
  - invokes `sys_write` system call, e.g. by executing a trap instruction
  - takes the value returned by `sys_write` and passes it back to the caller

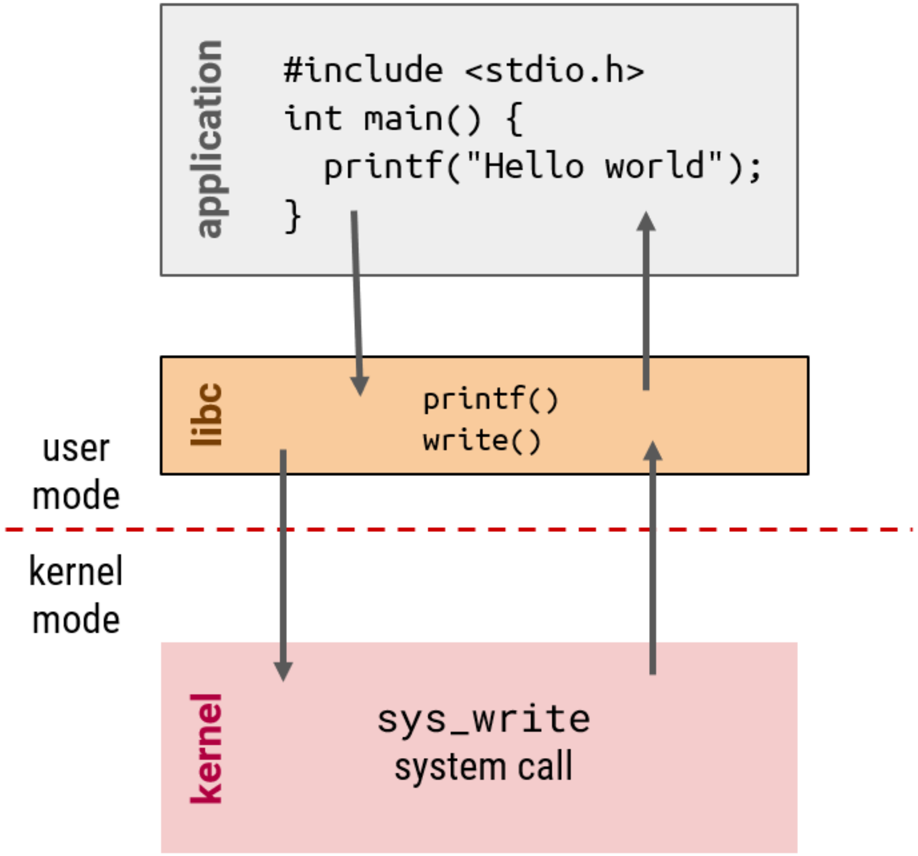
API / System calls / OS relationship



Black-box: Application writers do not need to know how the system call works, they only need to obey the API and understand the functionality of the calls.

Example: printf()

- standard C library provides also many useful higher-level convenience functions, e.g. `printf()`
- `printf()` implementation does some formatting and then calls the system call `sys_write` directly or indirectly, via `write()`
- same applies to `std::cout` in libc++



UNIVERSITY OF CALGARY

Examples

Examples of system calls in C

Common file related system calls

`fd = open(file_name, how, …)`	open file for reading, writing, ...
`s = close(fd)`	close open file
`n = read(fd, buffer, nbytes)`	read data from a file into buffer
`n = write(fd, buffer, nbytes)`	write data from buffer to an open file
`newpos = lseek(fd, offset, whence)`	move file pointer
`s = stat(name, & buf)`	get more info about a file (e.g. file length)

Examples of system calls in C

Common file related system calls

s = **mkdir**(name, mode)	create new directory
s = **rmdir**(name)	remove an empty directory
s = **link**(name1, name2)	create a file link name2 pointing to name1
s = **unlink**(name)	remove link (possibly delete file)

Examples of system calls in C

Miscellaneous

s = **chdir**(dirname)	change current working directory
s = **chmod**(name, mode)	change file's protection bits
s = **kill**(pid, signal)	send a signal to a process
seconds = **time**(& seconds)	get elapsed seconds since Jan 1, 1970

System calls (UNIX vs Win32)

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory

UNIVERSITY OF CALGARY

Unix APIs

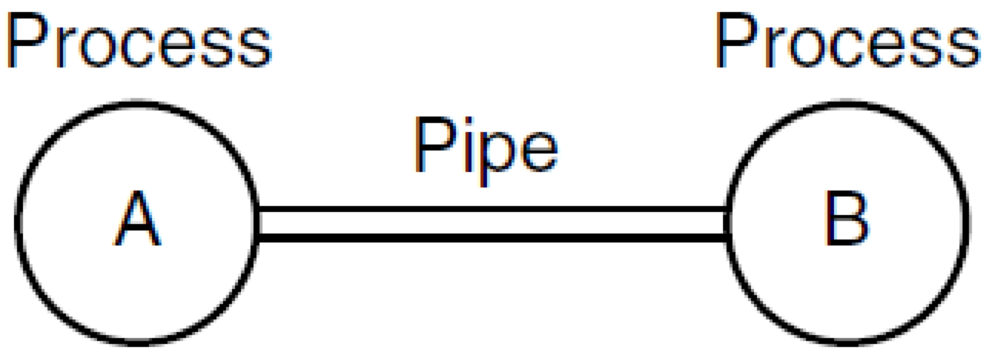
Unix file APIs

UNIX-like OSs make use of files and associated APIs for different operations and services

- pipes - communication between different programs (processes)
- sockets - networking
- communications with devices (`/dev`)
- random number generator (`/dev/random` and `/dev/urandom`)
- export kernel parameters (`/proc` and `/sys`)
  - pseudo filesystems containing virtual files
  - e.g. information about processes, memory usage, hardware devices

```
$ cat /proc/cpuinfo
$ cat /proc/meminfo
```

Pipes



- on unix systems, two processes can communicate with each other via a pipe
- pipes are accessed using file I/O APIs

```
$ ls -altr | tail -10
```

A1

Assignment 1

- the coding part is about improving performance of an existing program
- system calls are slow (they are essentially interrupts)
- making too many system calls slows down your program
- the objective is to try to reduce the number of system calls
- hint:
  - the existing program calls `read()` for every single byte
  - adjust the program so that `read()` gets multiple bytes in a single call, eg. 1MiB

Timing

time

- let's time how long it takes to calculate 40th fibonacci number recursively

```
#include <stdio.h>
long long fib(int n) {
  return n < 2 ? n : fib(n-1) + fib(n-2);
}
int main() {
  printf("%lld\n", fib(40));
}
```

```
$ g++ fib.cpp
$ ./a.out
102334155
```

- we can use a built-in time utility to get some basic timings

```
$ time ./a.out
102334155

real    0m1.190s
user    0m1.183s
sys     0m0.002s
```

real – same as if you used a stopwatch
user – time program spent executing on CPU
sys – time kernel spent executing code your application's behalf, but does not include I/O wait time

UNIVERSITY OF CALGARY

time

```
$ time ./a.out
102334155

real	0m1.190s
user	0m1.183s
sys	0m0.002s
```

- real = (user) + (sys) + (I/O) + (other)
- other = things CPU was doing while executing your application (e.g. running other applications)
- on an idle system, subtracting (user) from (real) will be a close estimate of how long an application spent waiting on I/O
- `a.out` finished in 1.19s, of which 1.183s was spent executing on CPU, and 1.19-1.183=0.007s was spent on I/O (if the computer was mostly idle, and application only made I/O related system calls)

Tracing

Tracing system calls

- tracing system calls = running an application and logging all system calls
- usually for debugging or performance optimization purposes
- on Linux: `$ strace` on Mac OS X: `$ dtruss`
- refer to the man page for further detail on these commands
- the same program/command could invoke different set of system calls on different OSes
- your program may run significantly slower when run through `strace`

- on Windows: Windows Performance Analysis Tools
  https://docs.microsoft.com/en-us/windows-hardware/test/wpt/windows-performance-analyzer

UNIVERSITY OF CALGARY

UNIX manual pages

```
$ man time
$ man read ✗
$ man -s2 read
$ man strace
```

```
Manual section:

1    Executable programs or shell commands
2    System calls (functions provided by the kernel)
3    Library calls (functions within program li-
     braries)
4    Special files (usually found in /dev)
5    File formats and conventions, e.g. /etc/passwd
6    Games
7    Miscellaneous (including macro packages and con-
     ventions), e.g. man(7), groff(7), man-pages(7)
8    System administration commands (usually only for
     root)
9    Kernel routines [Non standard]
```

Man pages

```
$ man strace

STRACE(1)                  General Commands Manual                 STRACE(1)

NAME
       strace - trace system calls and signals
SYNOPSIS
       strace  [-CdffhikqrtttTvVxxy]  [-In] [-bexecve] [-eexpr]...  [-acolumn]
       [-ofile]  [-sstrsize]  [-Ppath]...  -ppid...  /  [-D]  [-Evar[=val]]...
       [-uusername] command [args]

       strace  -c[df]  [-In]  [-bexecve]  [-eexpr]...  [-Ooverhead] [-Ssortby]
       -ppid... / [-D] [-Evar[=val]]... [-uusername] command [args]
DESCRIPTION
       In the simplest case strace runs the specified command until it  exits.
       It  intercepts  and  records  the  system  calls  which are called by a
       process and the signals which are received by a process.  The  name  of
       each  system  call,  its  arguments and its return value are printed on
       standard error or to the file specified with the -o option.
```

strace example

```
$ strace -c cat sample.txt
…
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 35.27    0.000073          18         4           open
 16.43    0.000034           3        10           mmap
  8.21    0.000017           4         4           mprotect
  8.21    0.000017           9         2           munmap
  7.73    0.000016           3         5           fstat
  4.83    0.000010           2         6           close
  4.35    0.000009           3         3           read
  3.86    0.000008           8         1           write
  3.86    0.000008           8         1         1 access
  3.38    0.000007           2         4           brk
  1.93    0.000004           4         1           execve
  0.97    0.000002           2         1           arch_prctl
  0.97    0.000002           2         1           fadvise64
------ ----------- ----------- --------- --------- ----------------
100.00    0.000207                    43         1 total
```

strace example

```
$ cat test.cpp

int main() {
  return 0;
}

$ g++ test.cpp
$ ./a.out
$
```

```
$ strace -c ./a.out

% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 42.87    0.000697         697         1           execve
 29.21    0.000475          20        23           mmap
  7.20    0.000117          23         5           openat
  5.17    0.000084          16         5           newfstatat
  4.43    0.000072          14         5           pread64
  3.57    0.000058          14         4           read
  2.95    0.000048           9         5           close
  1.85    0.000030           4         7           mprotect
  1.29    0.000021          21         1         1 access
  0.80    0.000013           4         3           brk
  0.68    0.000011           5         2         1 arch_prctl
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         1           futex
  0.00    0.000000           0         1           set_tid_address
  0.00    0.000000           0         1           set_robust_list
  0.00    0.000000           0         1           prlimit64
  0.00    0.000000           0         1           getrandom
  0.00    0.000000           0         1           rseq
------ ----------- ----------- --------- --------- ----------------
100.00    0.001626          23        68         2 total
```

?

Review

Summary

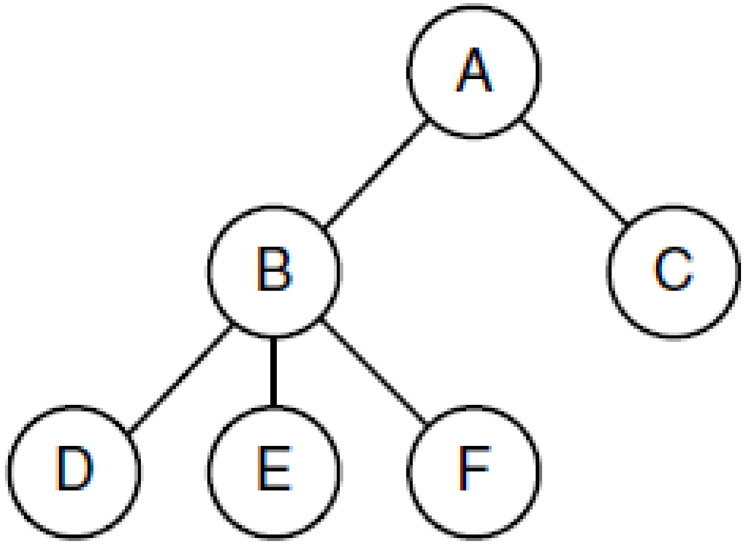
- Kernel
- System Calls
- Libraries
- Examples
  - C/Win32/Unix
- Unix APIs
- Timing
- Tracing
  - strace

Previews

Processes

- key concept in all operating systems
- quick definition: a program in execution
- process is associated with
  - an address space
  - set of resources
  - program counter, stack pointer
  - unique identifier (process ID)
  - … anything else?
- process can be thought of as a container that
  holds all information needed by an OS to run a program

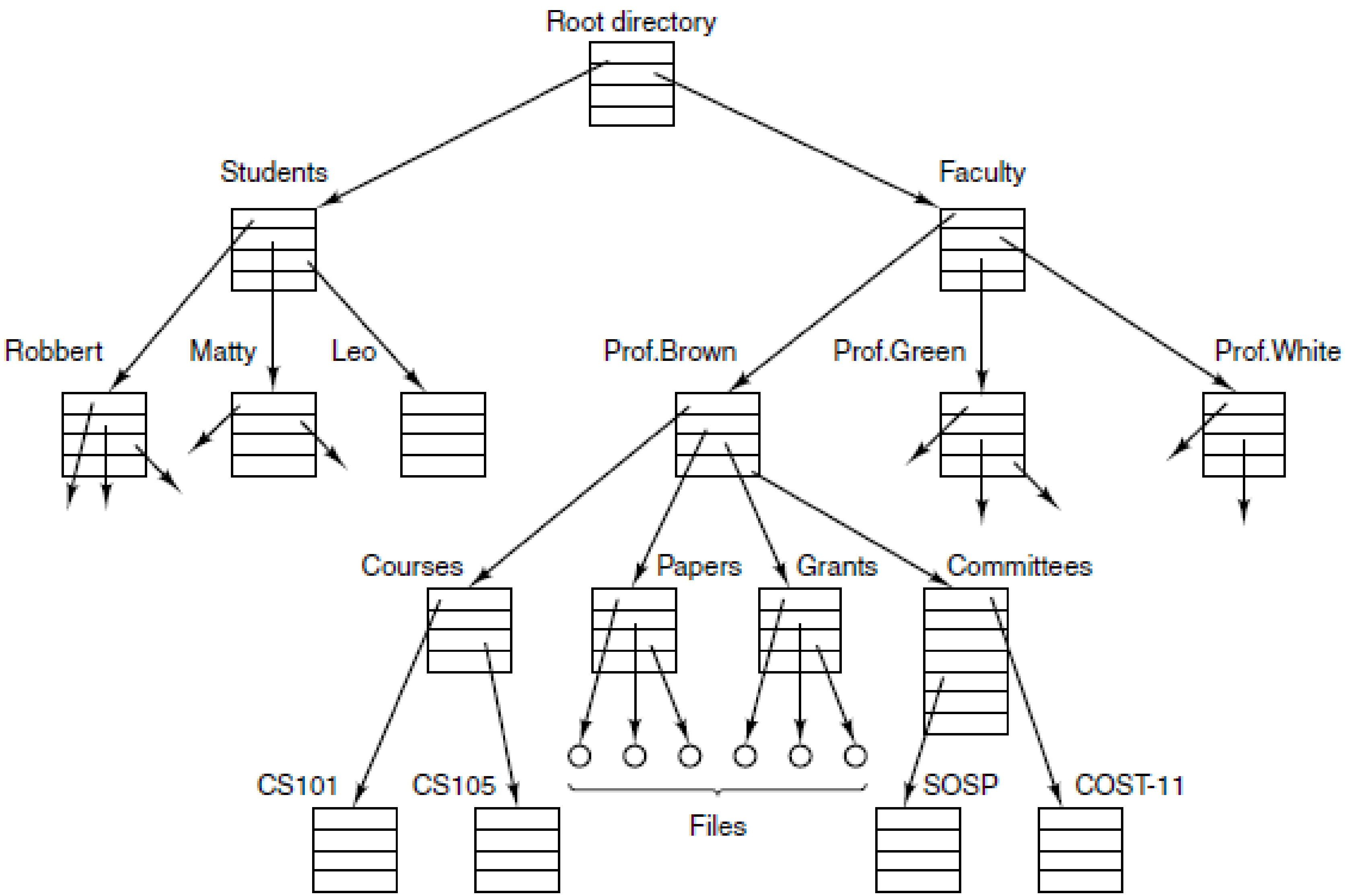
Process tree



- processes are allowed to create new processes
- A creates two child processes: B and C
- B creates three child processes: D, E and F
- A is the parent process of B
- B is a parent process of E

- A is an ancestor of F
- F is a descendant of A

File system - tree structure (subdirectories and files)



UNIVERSITY OF CALGARY

Onward to … Processes

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY