CPSC 457

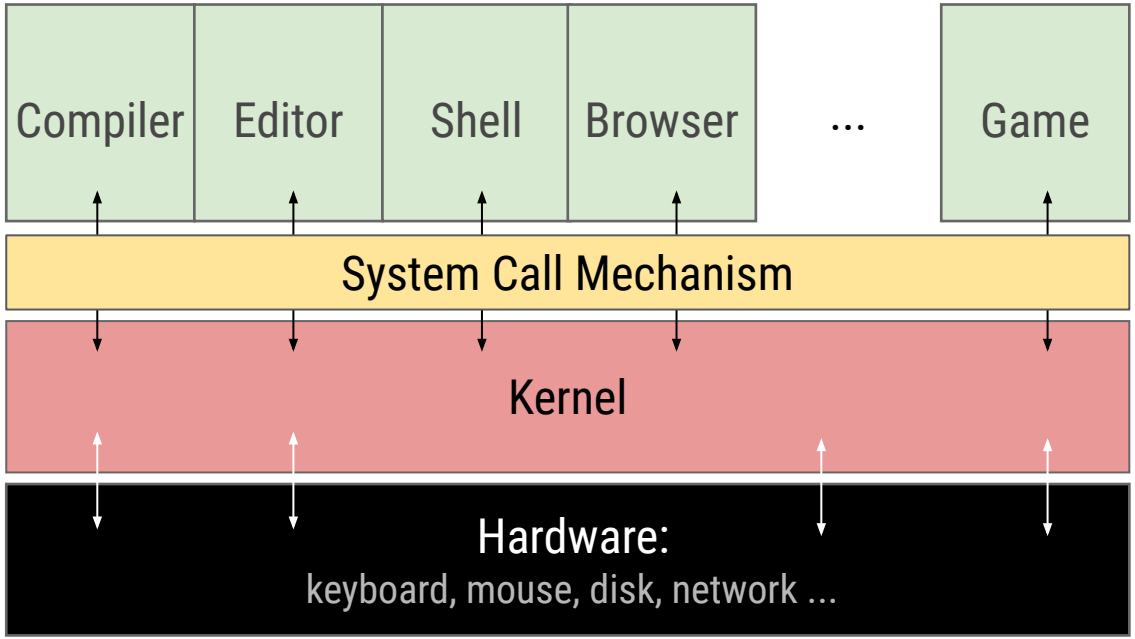
System calls

Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

Operating system

- OS provides services to applications, eg.
  - access to hardware, often via higher level abstractions
  - resource management
- these services are accessible through **system calls**, aka kernel calls
  - often implemented via **traps** - software interrupts
  - traps allow for a safe way to switch 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

- when an application wants to access a service / resource of the system:
  - the application must make an appropriate **system call** - call a routine provided by the OS
  - often by invoking a trap – recall that trap is a special CPU instruction that switches from **user mode** to **kernel mode** and invokes a pre-defined trap handler, registered by kernel
  - inside trap handler:
    - OS saves application state
    - OS performs the requested operation, eg. involving some hardware
    - OS switches back to user mode and restores application state
    - after this the application resumes
- from application's perspective, making a system call is just like calling a library function

System calls

- system calls provide an interface to the services provided by the OS
- think of system calls as an API provided by the OS for all applications
- the interface for system calls varies from OS to OS,
  although the underlying concepts tend to be similar
- OSes often need to execute 1000s of system calls per second

Example: copying file

- even the simplest programs make many `system calls`
- example: program that copies a file

```
Acquire input file name
   Write prompt to screen
   Accept input
Acquire output file name
   Write prompt to screen
   Accept input
Open the input file
If input file doesn't exist, abort
Create empty output file
If file could not be created, abort
```

```
Loop
   Read byte(s) from input file
   Write byte(s) to output file
Until read or write fails
Close input file
Close output file
Write completion message to screen
Terminate normally
```

Libraries and system calls

- system calls are usually implemented in assembly, hand optimized for performance
- system call number and parameters usually 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 inconvenient to call from higher level languages
- much easier to make system calls through higher-level wrapper libraries
- on Unix-like systems:

  libc - a C library, **libstdc++** or **libc++** for C++

```
write(fd, buff, len); // example of a calling wrapper for system call sys_write
```

Libraries and system calls

- system call wrappers are made available through libraries
- wrappers hide the implementation details of system calls
  - eg. convert input parameters into registers, and the return values
- an application using wrappers can compile and run on any system that supports the same APIs
- if the system call ever changes / is deprecated, the program using the wrapper could still continue to function properly
- some common APIs:
  - POSIX APIs for Unix, Linux, Mac OS X
  - Win32 APIs for windows
  - Java APIs for Java virtual machine
- often 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

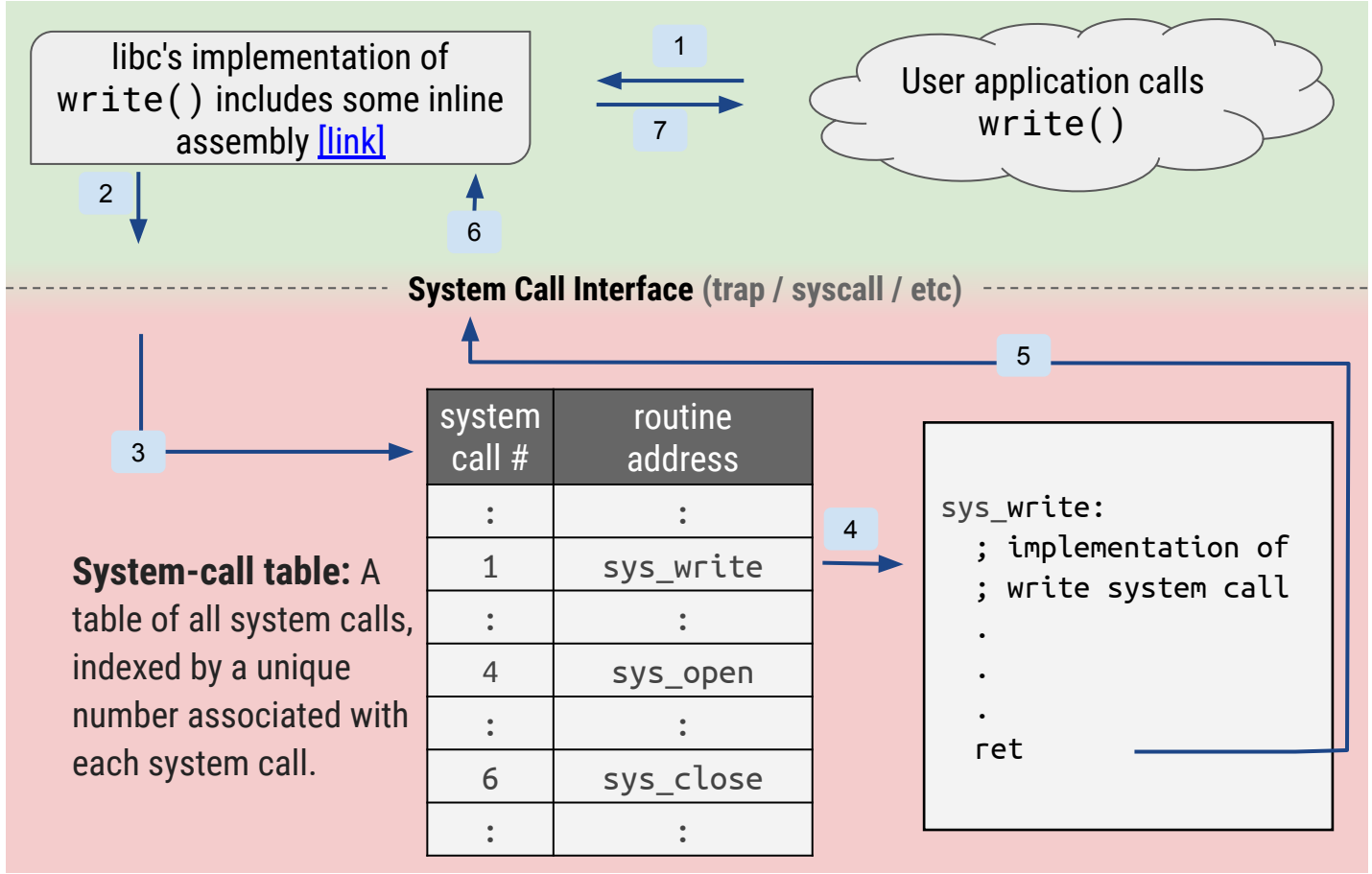
Example: `write()`

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

```
ssize_t write(int fd, const void *buf, size_t count);
```

- `write()` transfers the arguments passed to it on the stack into appropriate registers
- `write()` then calls the `sys_write` system call by executing a trap
- `write()` takes the value returned by `sys_write` and passes it back to the caller

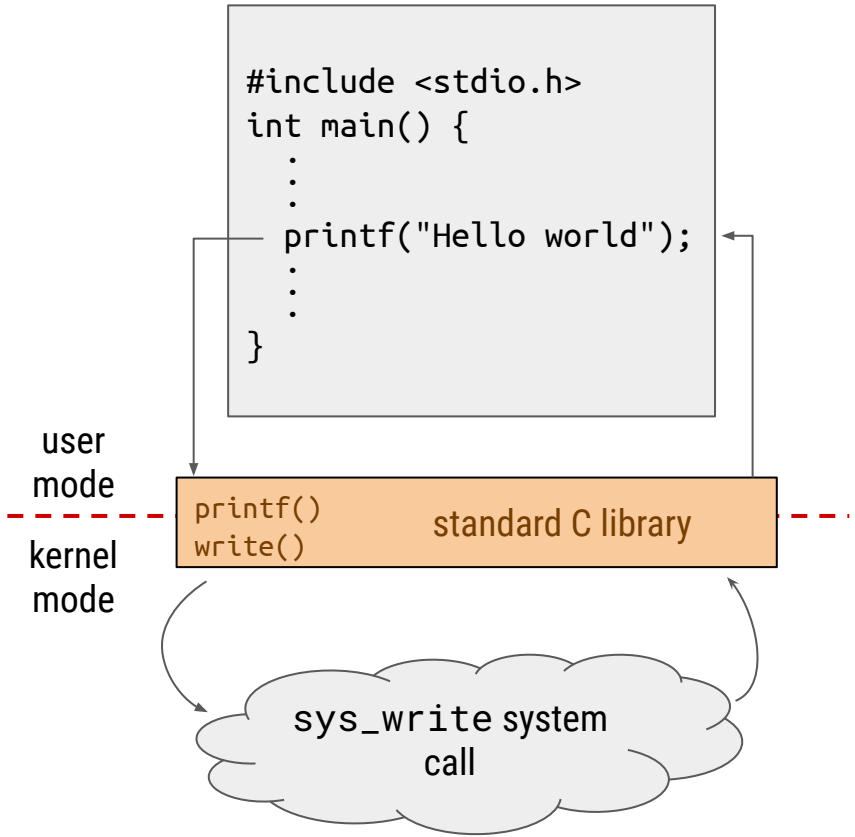
API / System calls / OS relationship



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, eg. `printf()`
- `printf()` implementation does some formatting and then calls the system call `sys_write` directly, or indirectly via `write()`



Examples of system call APIs in C

File management

Call	Description
`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 (eg. file length)

More examples of system call APIs in C

File & directory management

Call	Description
`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)

Even more examples of system call APIs in C

Miscelaneous

Call	Description
`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 examples (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

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 Solaris: **`truss`**
- on Mac OS X: **`dtruss`**
- on Windows: Windows Performance Analysis Tools
  https://msdn.microsoft.com/en-us/windows/hardware/commercialize/test/wpt/windows-performance-analyzer
- refer to the man page for further detail on these commands
- note: the same program/command could invoke different set of system calls on different OSes
- note: your program may run significantly slower

Man pages

```
$ man strace
```

Man pages

```
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

```
$ strace cat sample.txt
$ strace ./readFile sample.txt

$ strace -c cat sample.txt
$ strace -c ./readFile sample.txt
```

strace

```
$ strace cat sample.txt
...
open("readme.txt", O_RDONLY)            = 3
fstat(3, {st_mode=S_IFREG|0600, st_size=4, ...}) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap(NULL, 1056768, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fd581f6e000
read(3, "hey\n", 1048576)               = 4
write(1, "hey\n", 4hey
)                   = 4
read(3, "", 1048576)                    = 0
munmap(0x7fd581f6e000, 1056768)         = 0
close(3)                                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
...
```

strace

```
$ 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 demo

let's run strace on hello world C++ program:

```
#include <stdio.h>
int main()
{
  printf("Hello world\n");
  return 0;
}
```

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
```

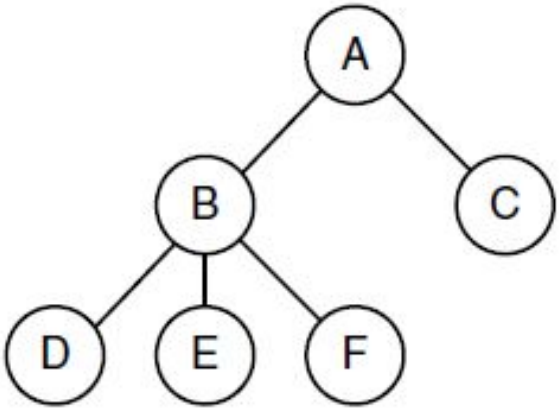
real – same as if you used a stopwatch
user – time program spent on CPU
sys – time program spent in kernel mode

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

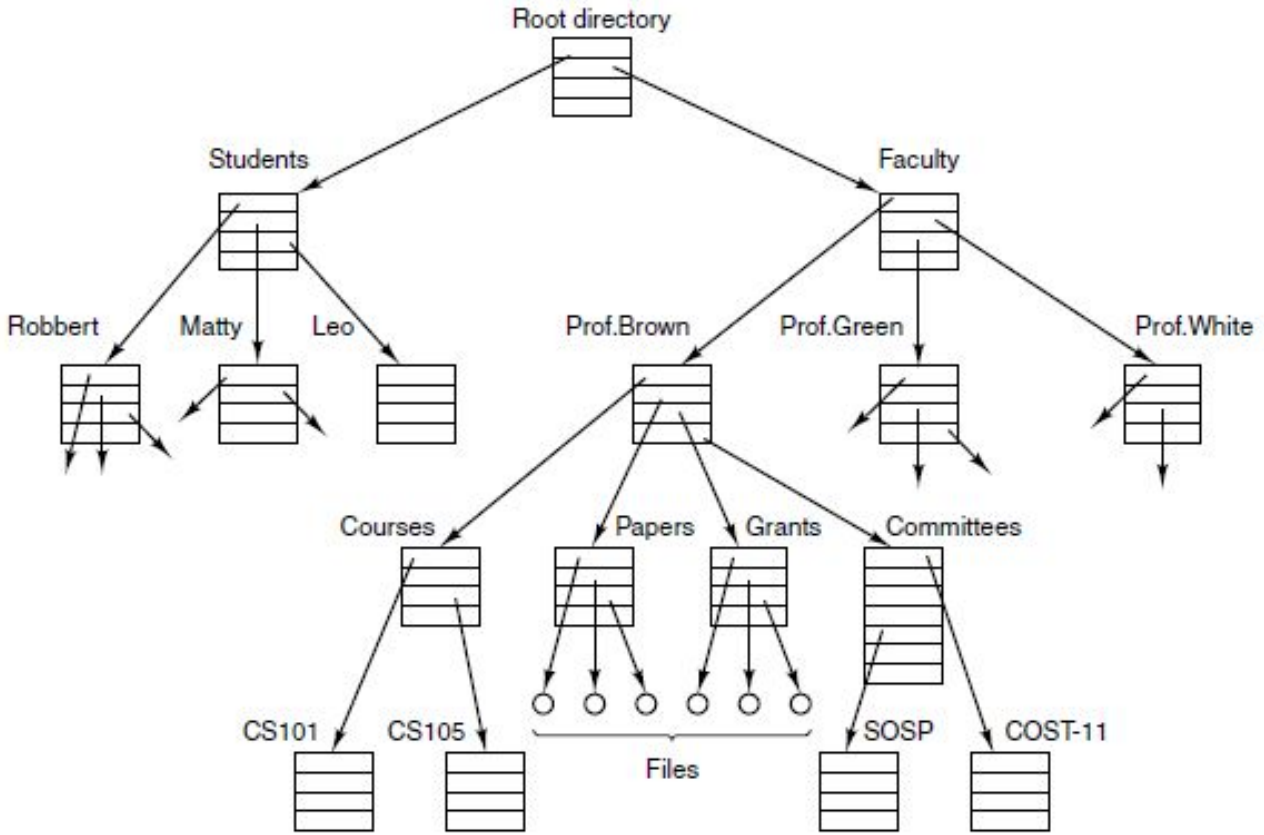
Process tree



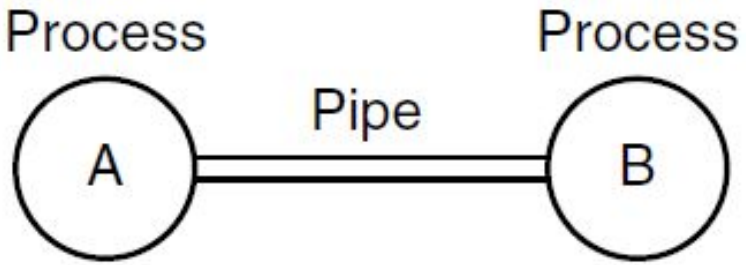
- processes are allowed to create new processes
- A creates two **child processes**: B and C
- B creates three child processes: D, E and F
- A is the **parent process** of B
- B is a parent process of E

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

File system - tree structure (subdirectories and files)



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
```

Unix file APIs

- UNIX-like OSs make use of files and associated APIs for different operations / services
- pipes - interprocess communication
- sockets - networking
- devices (`/dev`)
  - block devices - disks
  - character devices - terminals
- random number generator (`/dev/random`)
- export kernel parameters (`/proc` and `/sys`)
  - pseudo filesystems containing virtual files
  - eg. information about processes, memory usage, hardware devices
  - try `$ cat /proc/cpuinfo` or `$ cat /proc/meminfo`

Questions?

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

- 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: `read()`

Steps in making a wrapper call

```
read(fd, buffer, nbytes)
```