CPSC 457

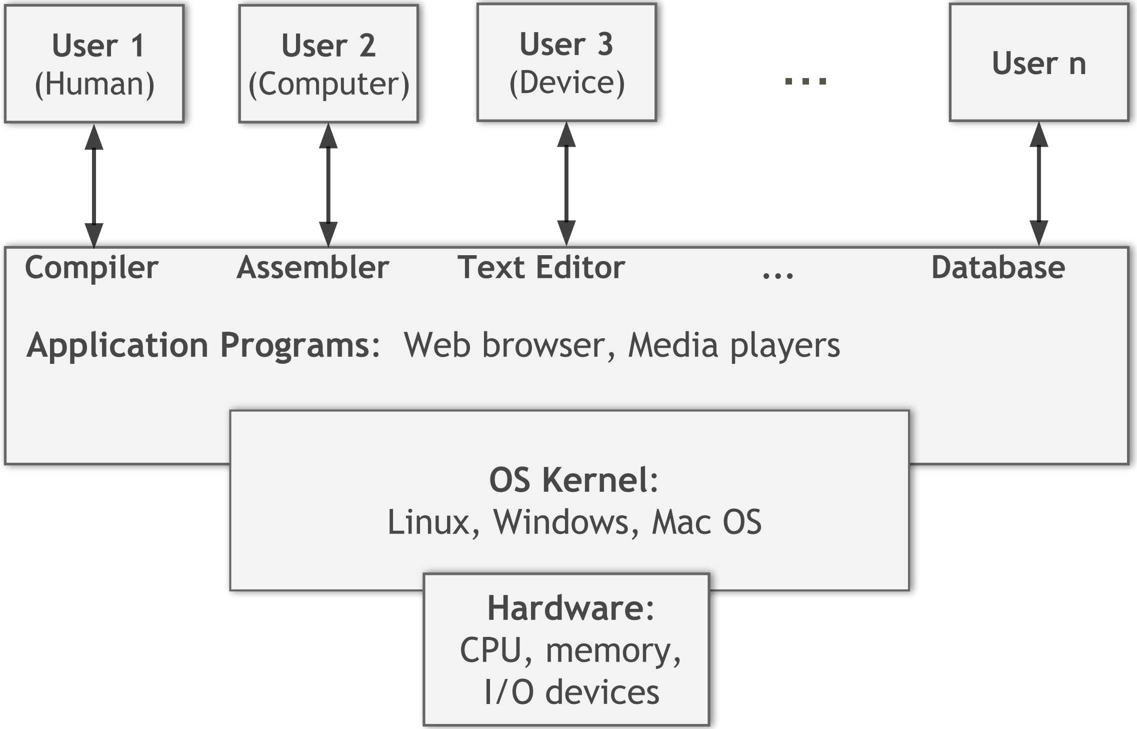
System calls

Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

Operating system

- provides services to applications, eg.
  - access to hardware, often via higher level abstractions
  - resource management
- these OS services are accessible through **system calls**, aka kernel calls
  - usually via **traps** - software interrupts

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
  - usually done through a mechanism called **trap**
    - trap is a special CPU instruction
    - switches from **user mode** to **kernel mode**
    - invokes a pre-defined trap handler, registered by kernel
  - inside trap handler:
    - OS saves application state
    - OS does the requested operation, eg. involving some hardware
    - OS switches back to user mode and restores application state
  - the application resumes
- from application's perspective, making a system call is just like calling any other routine

System calls

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

Example: copy file

Source File
Destination 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 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
```

even the most simple programs make many system calls

Libraries and system calls

- system calls are minimalistic, and not very easy to use
  - usually implemented in assembly, optimized for performance
  - system call number and parameters usually passed in registers

```
mov  eax,4      ; system call # (sys_write)
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/
  - quite inconvenient to use from higher level languages
- preferred way to make system calls is through higher-level wrappers
- on Unix-like systems:

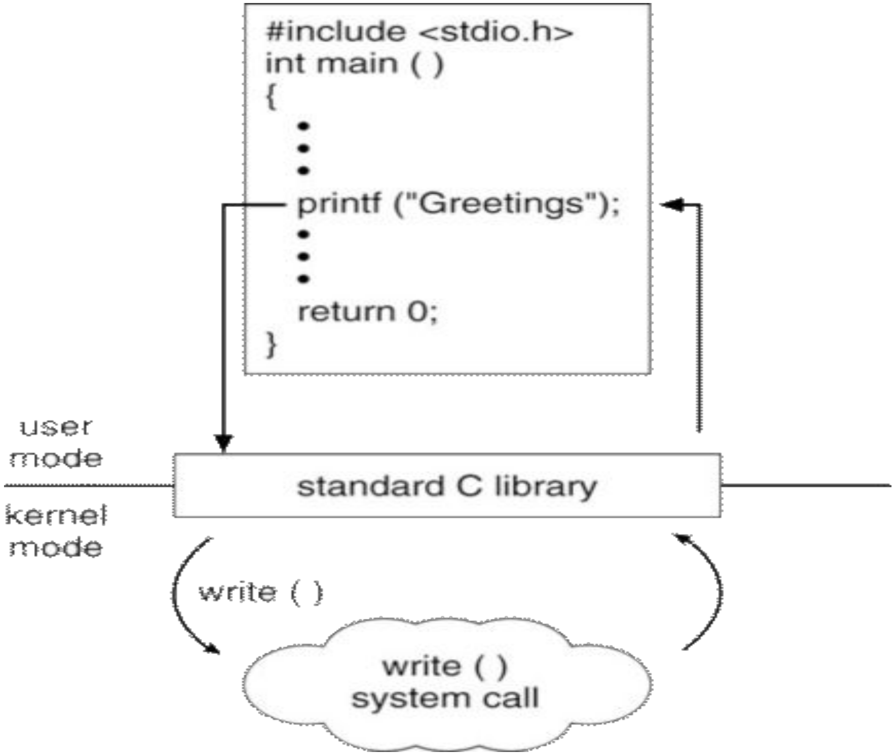
  libc - a C library, **libstdc++** or **libc++** for C++

Libraries and system calls

- a library can provide a set of functions (APIs) that are are available to an application programmer, including the parameters and the return values
- these APIs often hide the implementation details of system calls
- making system calls via wrappers is more convenient
- an application can compile and run on any system that supports the same API
- added benefit – 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 function in the API and its associated system call within the kernel, but API != system call

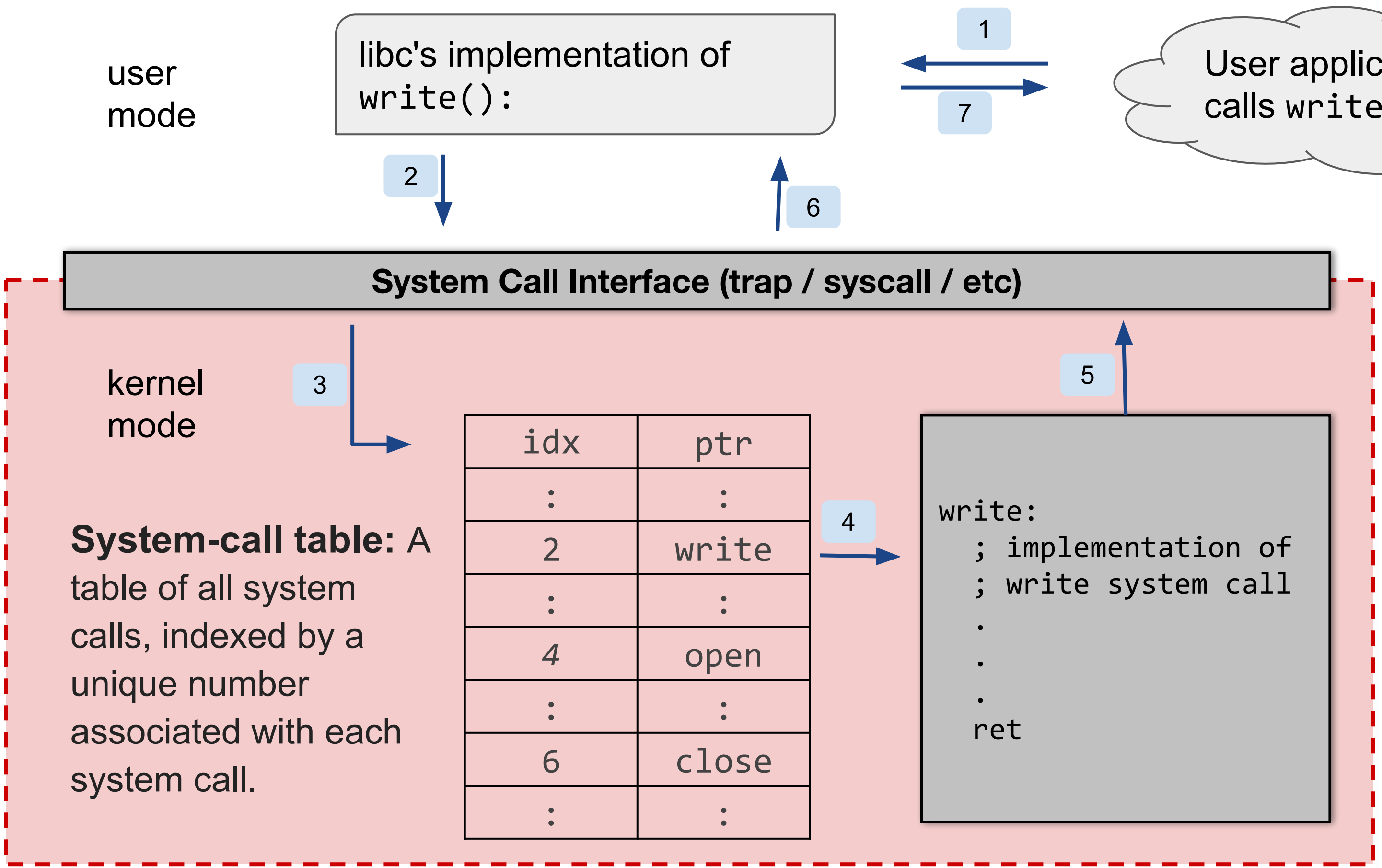
Example: printf(...)

- standard C library provides access to many OS system calls
- for example: **`write()`**
  - **`write()`** prepares arguments in registers
  - **`write()`** calls the **`write`** system call
  - **`write()`** takes the value returned by **`write`** and passes it back to the caller
- but also many useful higher-level APIs, eg. **`printf()`**
  - **`printf()`** does some formatting and then calls the system call **`write`**



```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

API / System calls / OS relationship


user mode
libc's implementation of write():
1
7
User application calls write()
2
6
System Call Interface (trap / syscall / etc)
kernel mode
3
4
5

idx	ptr
:	:
2	write
:	:
4	open
:	:
6	close
:	:

```
write:
   ; implementation of
   ; write system call
   .
   .
   .
   ret
```

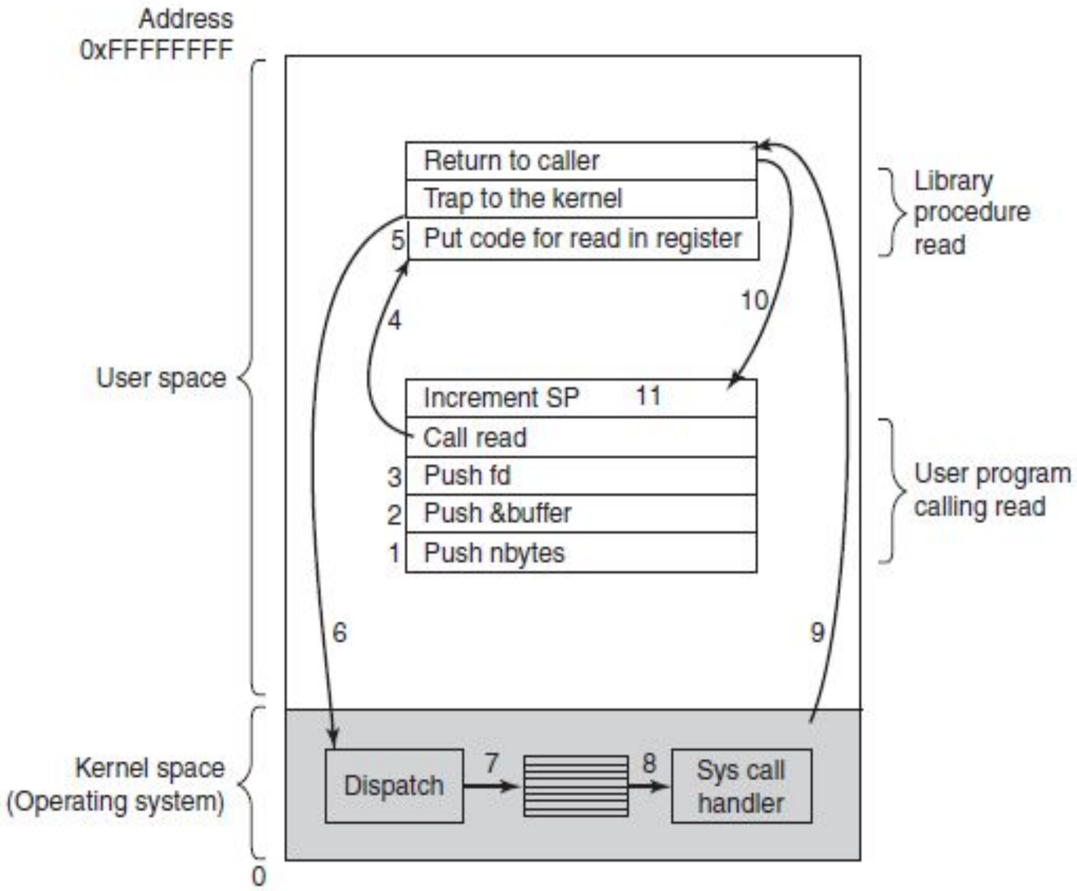
System-call table: A table of all system calls, indexed by a unique number associated with each system call.

Black-box: No need to know how the system call works, only need to obey the API and understand the functionality of the calls.

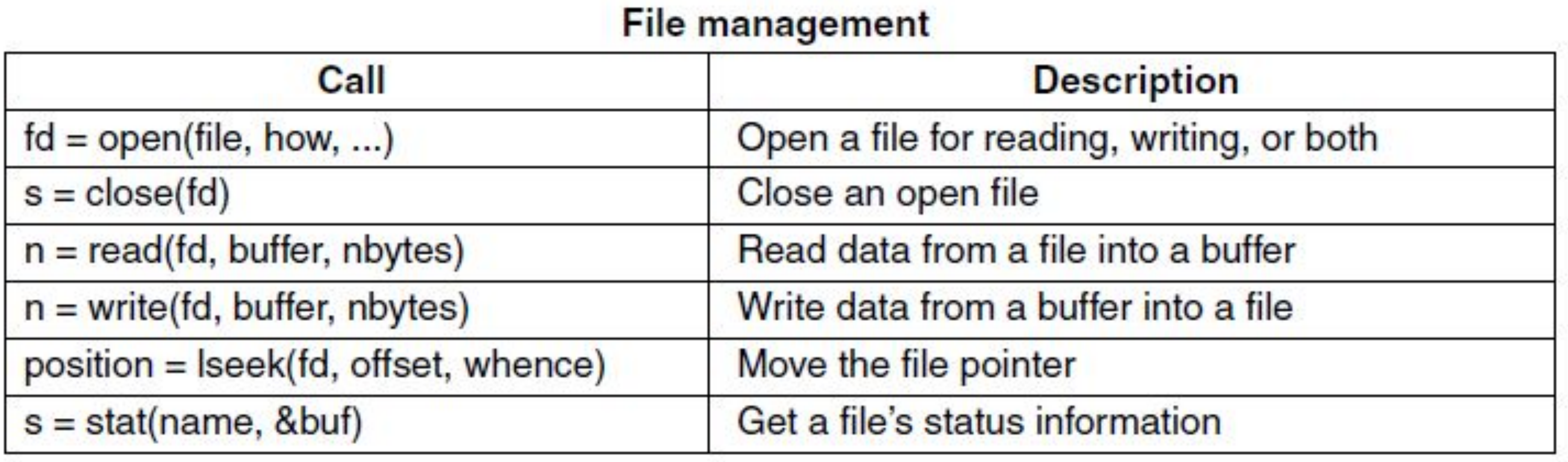
Example: read()

Steps in making a wrapper call

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



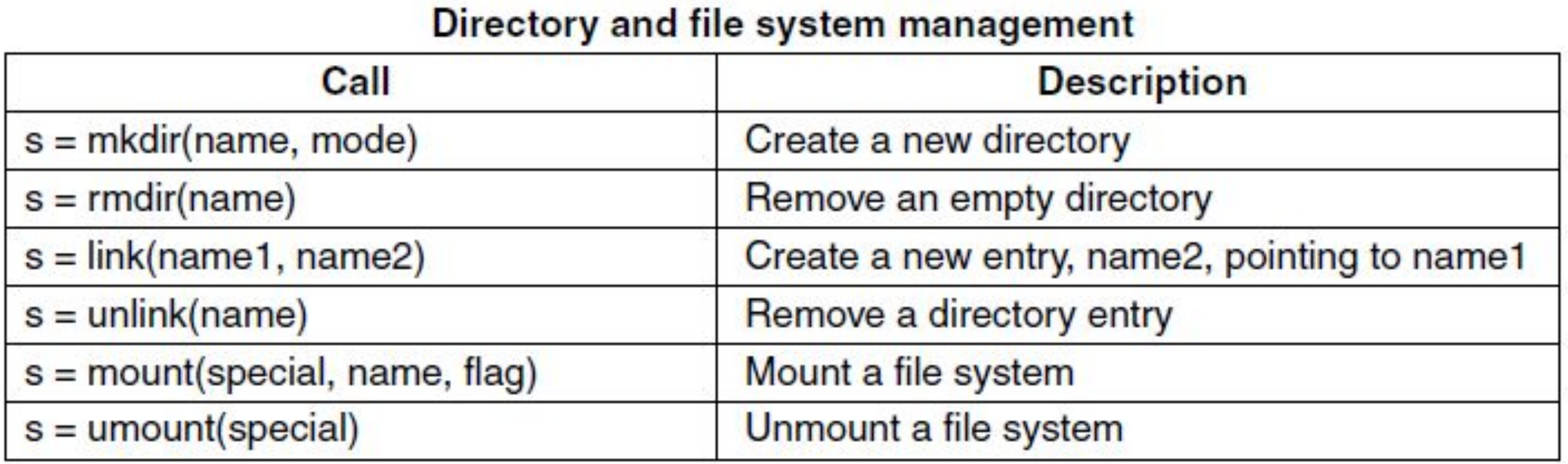
Examples of system call APIs in C



File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

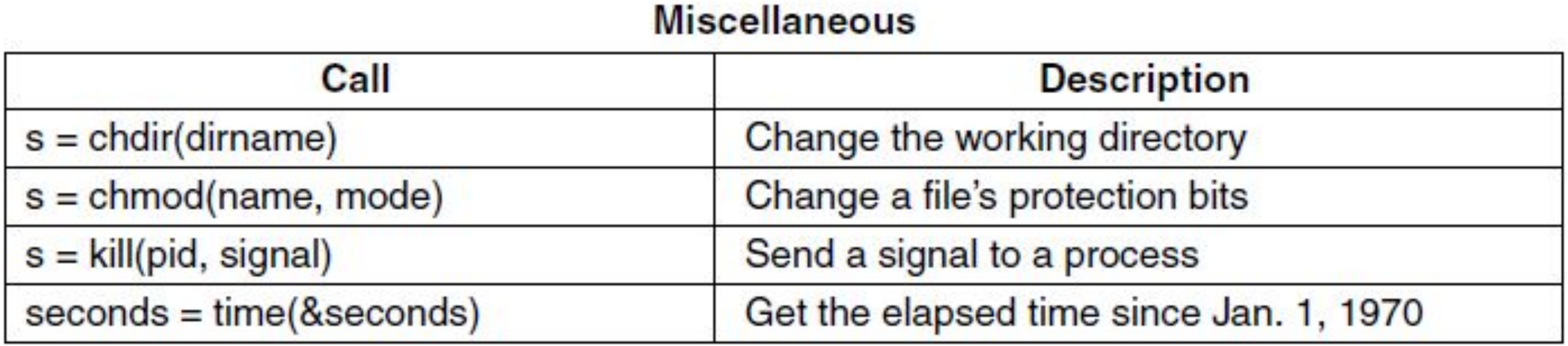
More examples of system call APIs in C



Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

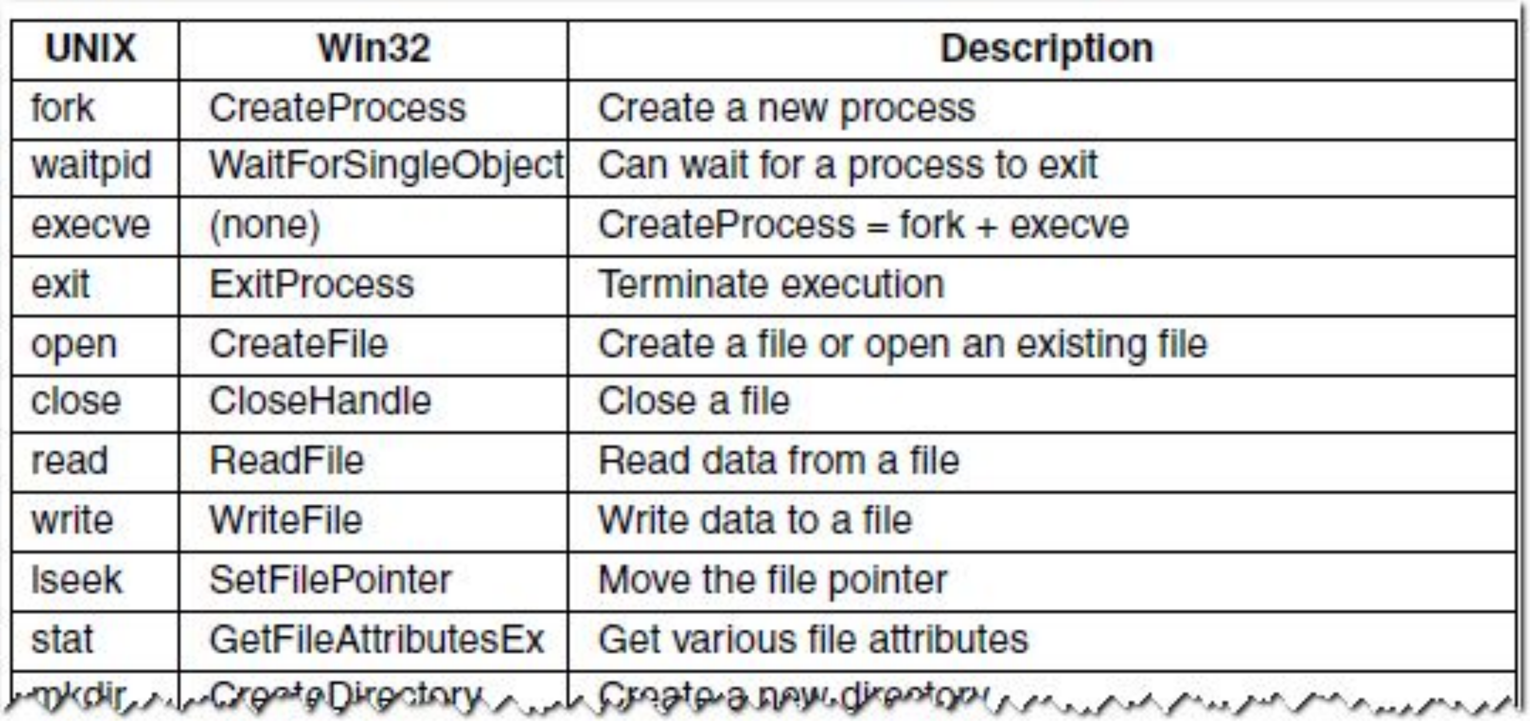
Even more examples of system call APIs in C



Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time 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

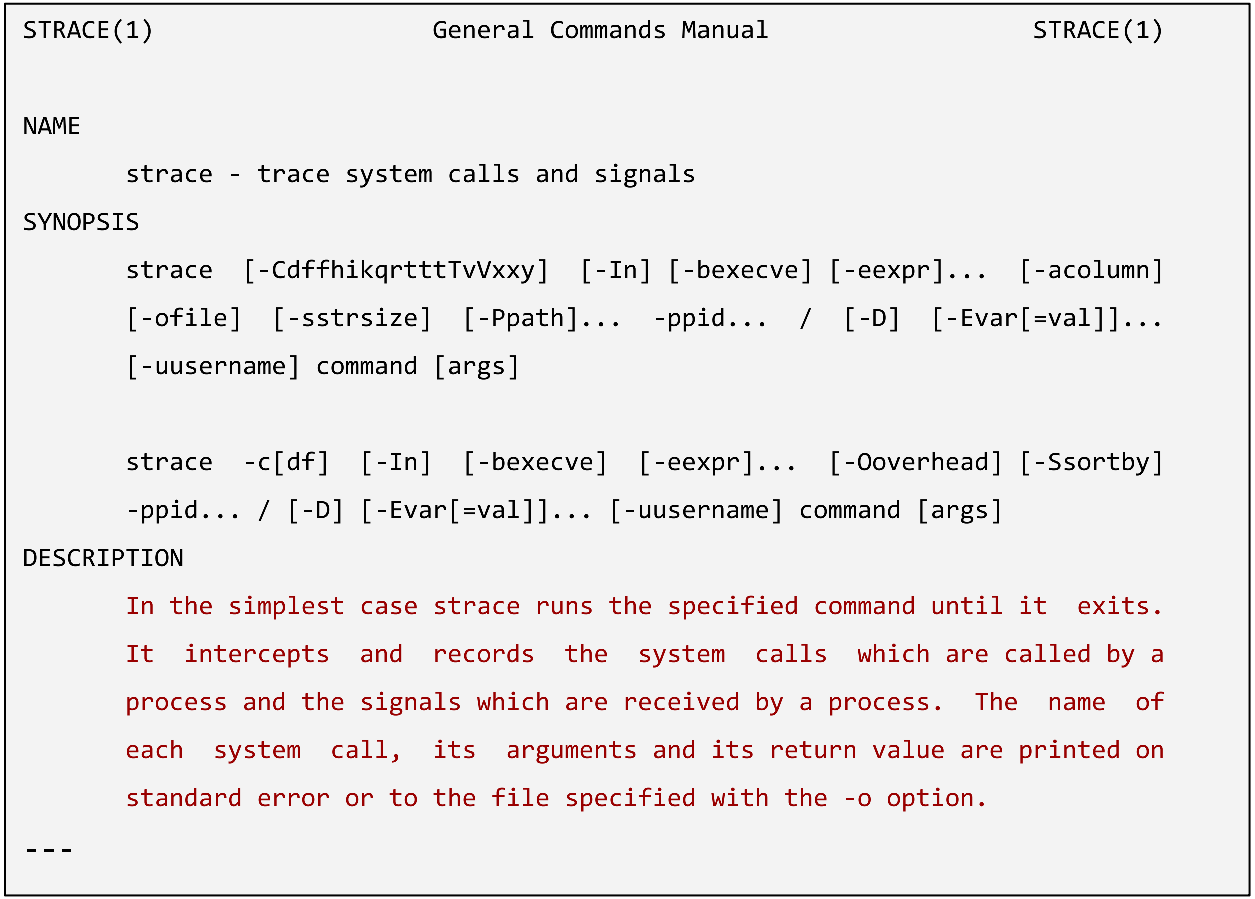
- tracing system calls = running an application and logging all system calls
- usually for debugging or performance optimization purposes
- on Linux: `strace`
- on Solaris: `truss`
- on Mac OS X: `dtruss`
- on Windows: Windows Performance Analysis Tools

  https://msdn.microsoft.com/en-us/windows/hardware/commercialize/test/wpt/windows-performance-analyzer
- note: the same program/command could invoke different set of system calls on different OSes
- refer to the man page for further detail on these commands

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

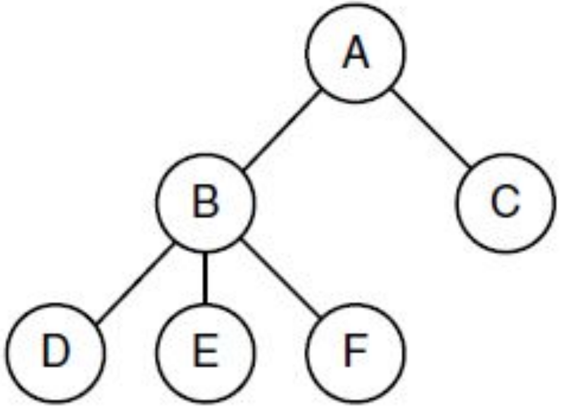
```
$ strace -c cat sample.txt

…

% 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
```

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

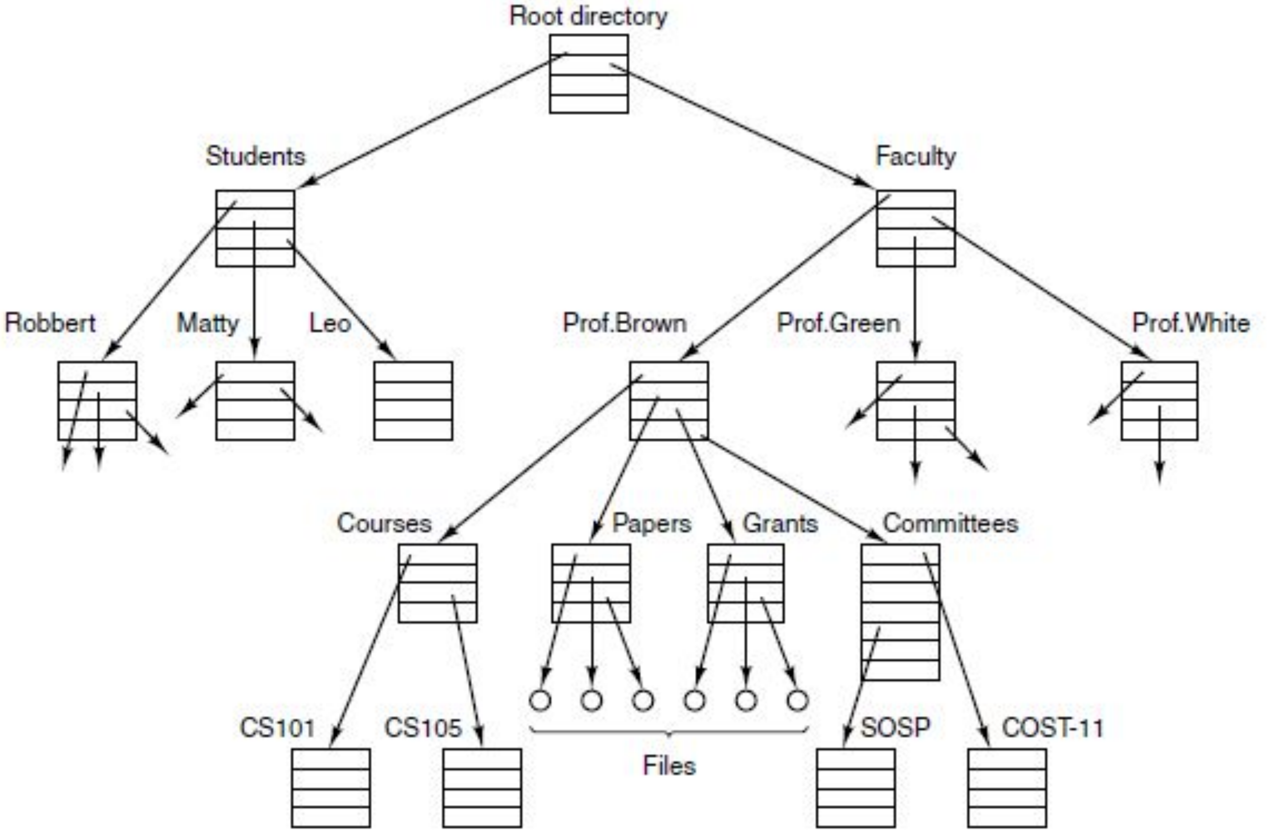
  holds all information needed by an OS to run a program

Process tree

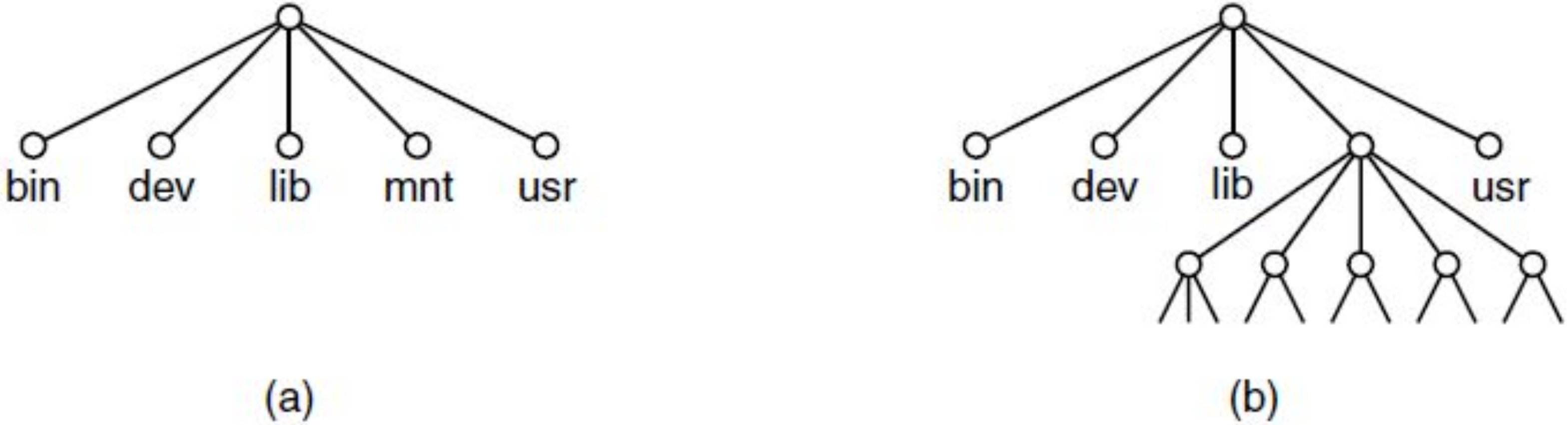


- processes are allowed to create new processes
- A creates two **child processes**: B and C
- B creates three child processes: D, E and F
- A is the **parent process** of B
- B is a parent process of E

File system - tree structure (subdirectories and files)

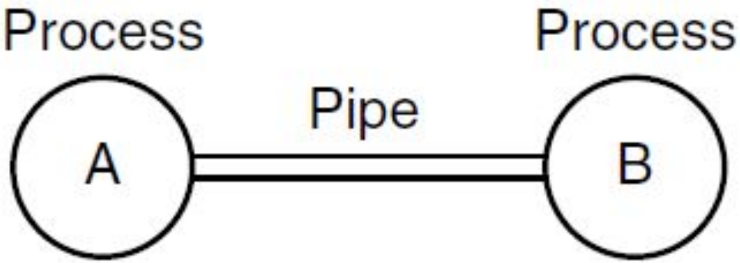


File system mounting



- on most Unix filesystems, other filesystems are mounted into an existing filesystems at an arbitrary location (subdirectory)
- (a) before mounting
- (b) after mounting

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

Questions?