Based on

Chapter 1 slides of the book Operating System Concepts, Silberschatz, Galvin and Gagne:
https://www.os-book.com/OS10/slide-dir/index.html
https://www.scs.stanford.edu/25sp-cs212/notes/intro.pdf
https://pages.cs.wisc.edu/~remzi/OSTEP/intro.pdf

Intro to OS

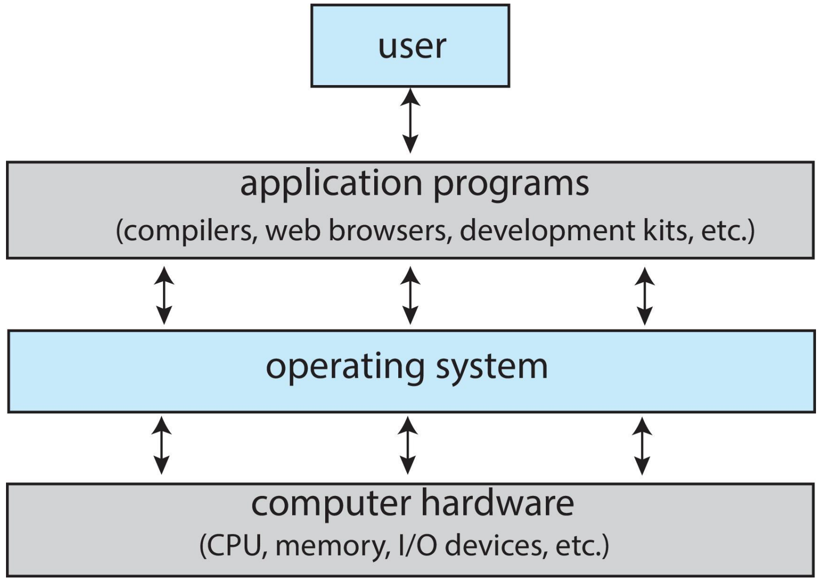
What Does the Term Operating System Mean?

- An operating system is “.....................................”?
- What about:
    - Car
    - Airplane
    - Printer
    - Washing Machine
    - Toaster
    - Compiler
    - Etc.

Computer System Structure

- Computer system can be divided into four components:
  - **Hardware** – provides basic computing resources
    - CPU, memory, I/O devices
  - **Operating system**
    - Controls and coordinates use of hardware among various applications and users
  - **Application programs** – define the ways in which the system resources are used to solve the computing problems of the users
    - Word processors, compilers, web browsers, database systems, video games
  - **Users**
    - People, machines, other computers

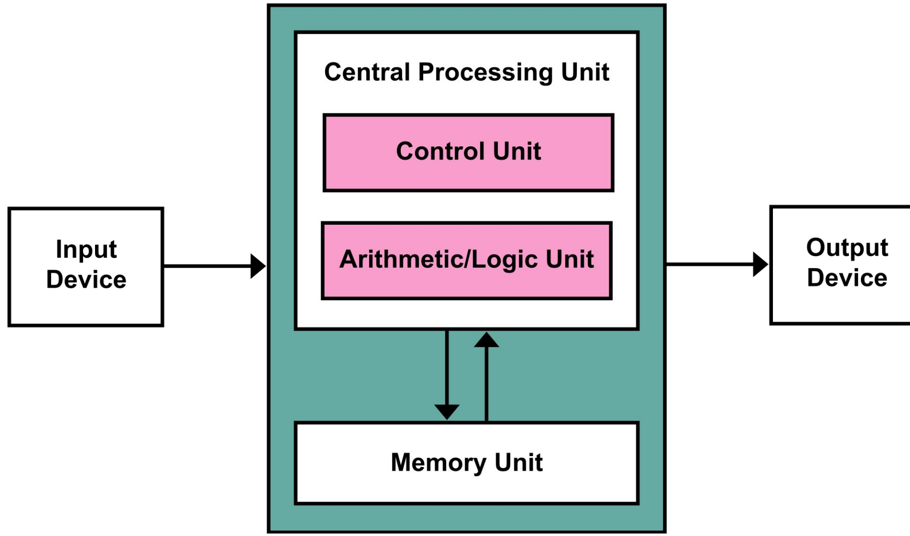
What is an Operating System?



- An OS is a program that acts as an intermediary between a user of a computer and the computer hardware

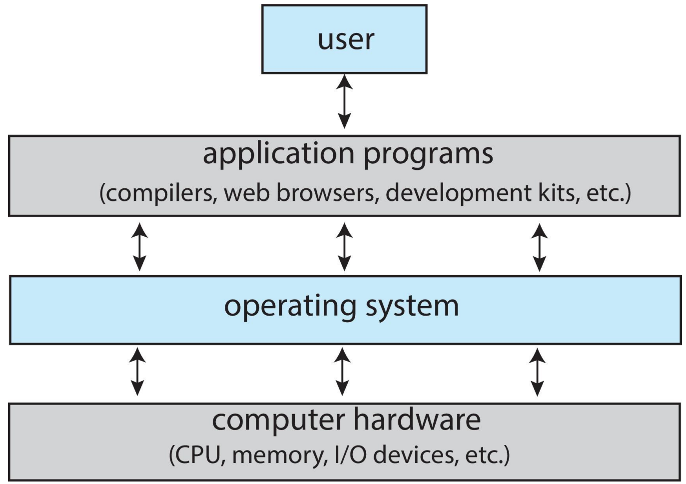
What happens when a program runs?

- it simply executes instructions:
    - von Neumann model of computing:
        - the processor fetches an instruction from memory, decodes it (figures out which instruction this is), and executes it.
        - After it is done with this instruction, the processor moves on to the next instruction, and so on, and so on, until the program finally completes
- billions of times every second,



https://en.wikipedia.org/wiki/Von_Neumann_architecture

What is an Operating System?



- Makes hardware useful to the programmer(goal):
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner

```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;
}
```

```c
#ifndef __common_h__
#define __common_h__

#include <assert.h>
#include <sys/stat.h>
#include <sys/time.h>

double GetTime() {
    struct timeval t;
    int rc = gettimeofday(&t, NULL);
    assert(rc == 0);
    return (double)t.tv_sec + (double)t.tv_usec / 1e6;
}

void Spin(int howlong) {
    double t = GetTime();
    while ((GetTime() - t) < (double)howlong);  // do nothing in loop
}

#endif  // __common_h__
```

$ gcc -Wall -o cpu cpu.c

Running multiple programs at once

$ ./cpu "A" & ./cpu "B" & ./cpu "C"

—-------------------------------

Virtualizing CPU

```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;
}
```

Running mem program multiple times
$./mem &; ./mem &

——--------------
Virtualizing Memory

```c
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;
}
```

What primitives needed from OS to build correct and efficient concurrent programs?

```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;
}
```

HOW TO STORE DATA PERSISTENTLY

The file system is the part of the OS in charge of managing persistent data.

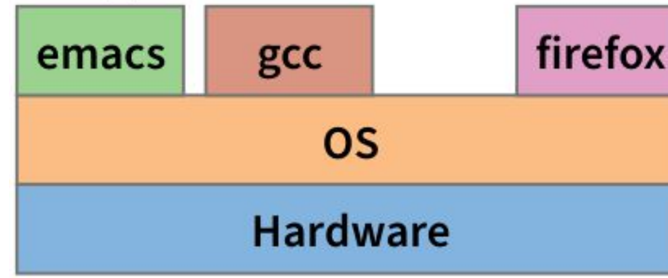
So far

OS **virtualizes** resources (CPU, memory, etc.), works as a **resource manager**

OS stores files **persistently**

OS helps tricky issues such as **concurrency**

What is an Operating System?



Design goals

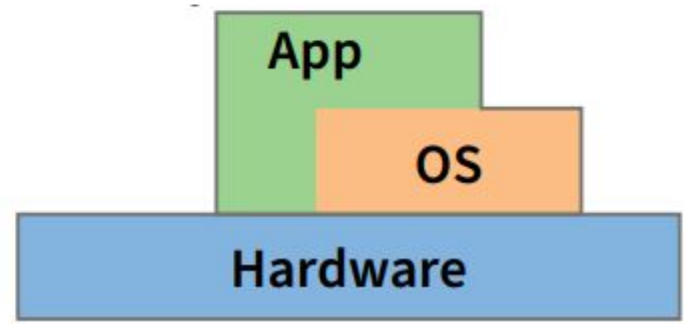
- [Usually] Provides **abstractions** for applications
  - Manages and hides details of hardware
  - Accesses hardware through low/level interfaces unavailable to applications
  - makes the system convenient and easy to use.
- [Often] Provides **protection**
  - Prevents one process/user from clobbering another
- Other design goal(issues):
  - high performance
  - must also run non-stop(reliability)
  - security
  - energy-efficiency
  - mobility

Why study OS?

- Operating systems are a mature field
  - Most people use a handful of mature OSes
  - Hard to get people to switch operating systems
  - Hard to have impact with a new OS
- Still open questions in operating systems
  - Security
  - Hard to achieve security without a solid foundation
  - Scalability
  - How to adapt concepts when hardware scales 10× (fast networks, low service times, high core counts, big data. . .)
- High-performance servers are an OS issue
  - Face many of the same issues as OSes, sometimes bypass OS
  - Resource consumption is an OS issue
  - Battery life, radio spectrum, etc.
- New “smart” devices need new OSes
  - Robotics(ROS), IoTs

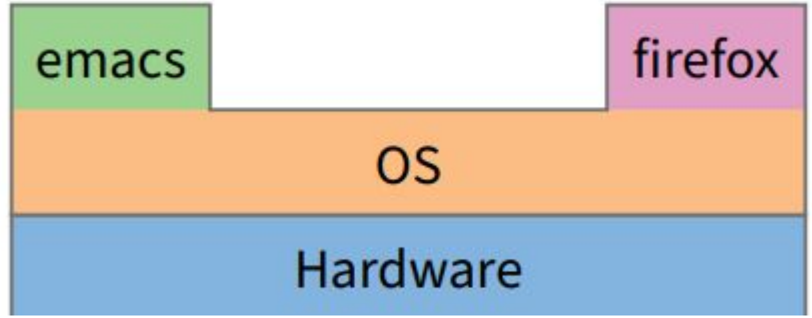
Early-primitive Operating Systems: Just Libraries

- Just a library of standard services **[no protection]**



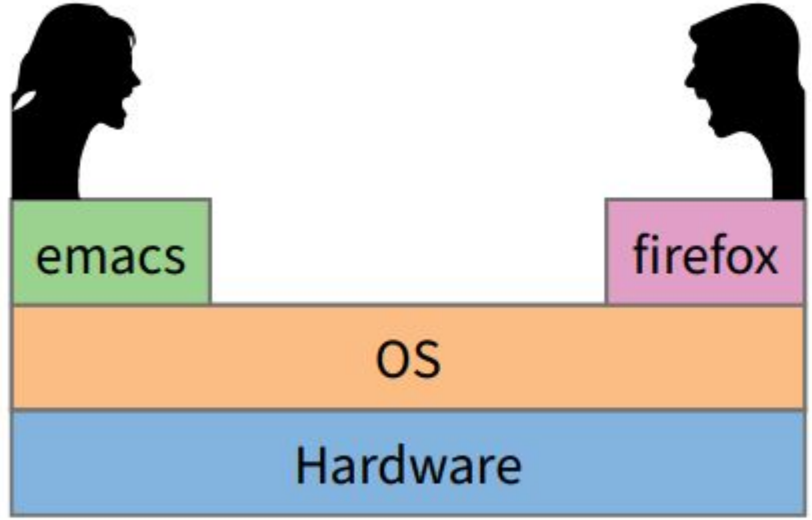
- Simplifying assumptions
  - System runs one program at a time (the program is chosen from a **batch**)
  - No bad users or programs (often bad assumption)
- Problem: Poor utilization
  - of hardware (e.g., CPU idle while waiting for disk)
  - of human user (must wait for each program to finish)

Beyond libraries: Multi-tasking & protection



- **Idea:** More than one process can be running at once
  - When one process blocks (waiting for disk, network, user input, etc.) run another process
- **Problem:** What can ill-behaved process do?
  - Go into infinite loop and never relinquish CPU
  - Scribble over other processes' memory to make them fail
- **OS provides mechanisms to address these problems**
  - Preemption – take CPU away from looping process
  - Memory protection – protect processes' memory from one another

Multi-user OS

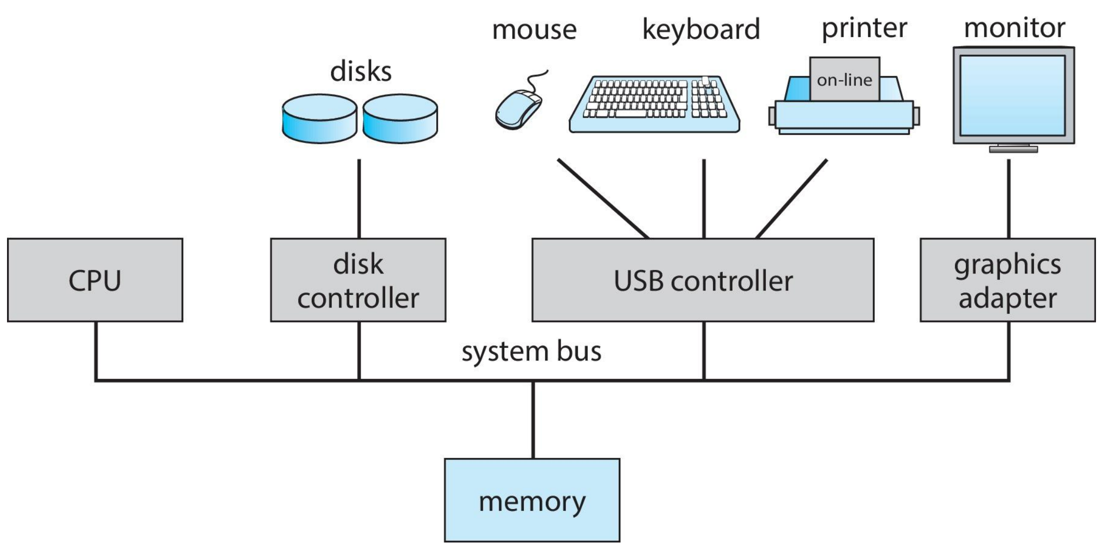


- Many OSes use **protection** to serve distrustful users/apps
- **Idea:** With N users, system not N times slower
  - Users' demands for CPU, memory, etc. are bursty
  - Win by giving resources to users who actually need them
- **What can go wrong?**
  - Users are gluttons, use too much CPU, etc. (need policies)
  - Total memory usage greater than machine's RAM (must virtualize)
  - Super-linear slowdown with increasing demand (thrashing)

Protection

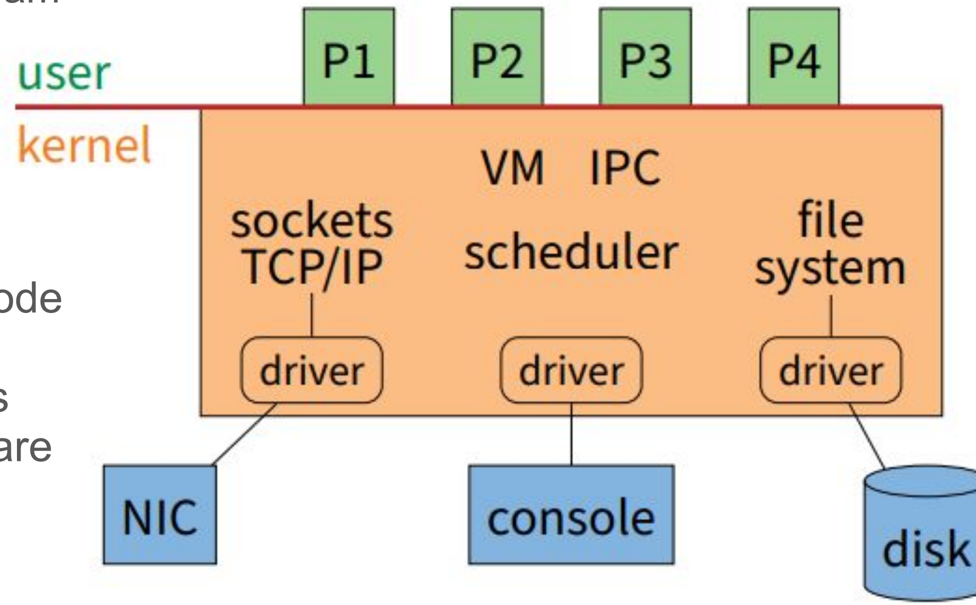
- Mechanisms that **isolate** bad programs and people
  - also isolating processes from one another
- Pre-emption:
  - Give application a resource, take it away if needed elsewhere
- Interposition/mediation:
  - Place OS between application and “stuff”
  - Track all pieces that application allowed to use (e.g., in table)
  - On every access, look in table to check that access legal
- Privileged & unprivileged modes in CPUs:
  - Applications unprivileged (unprivileged user mode)
  - OS privileged (privileged supervisor/kernel mode)
  - Protection operations can only be done in privileged mode

- Computer-system operation
  - One or more CPUs, device controllers connect through common **bus** providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



Typical OS structure

- Most software runs as **user-level processes (P[1-4])**
  - process ≈ instance of a program



- **OS kernel** runs in privileged mode (**orange)**
  - Creates/deletes processes
  - Provides access to hardware

System calls

Code run on behalf of the OS is special!

The idea of **a system call** was pioneered by the Atlas computing system.

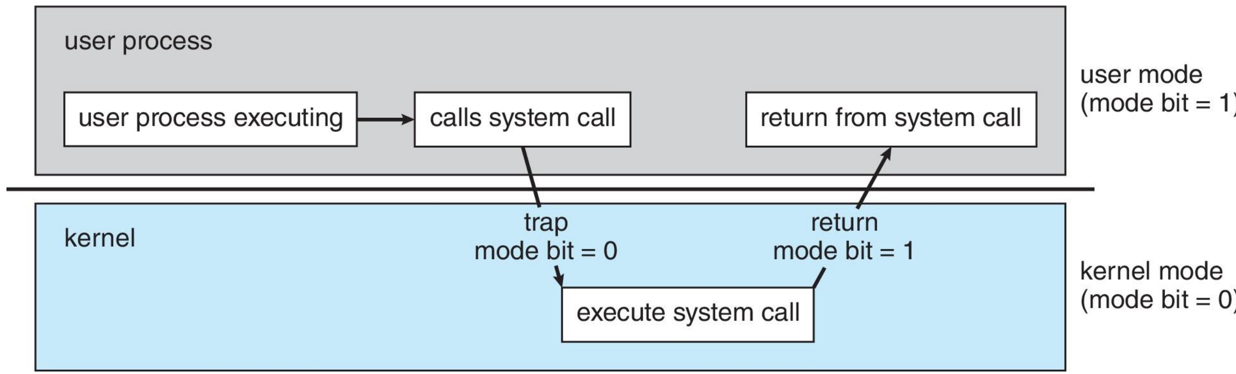
Instead of providing OS routines as a library (where you just make a procedure call to access them),

the idea here was to add **a special pair of hardware instructions** and hardware state to make the transition into the OS a more formal, controlled process.

Dual-mode Operation

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware
    - 1 mode bit is “user”
    - 0 mode bit is “kernel”
- How do we guarantee that user does not explicitly set the mode bit to “kernel”?
  - System call changes mode to kernel, return from call resets it to user
- Some instructions designated as **privileged**, only executable in kernel mode

Transition from User to Kernel Mode

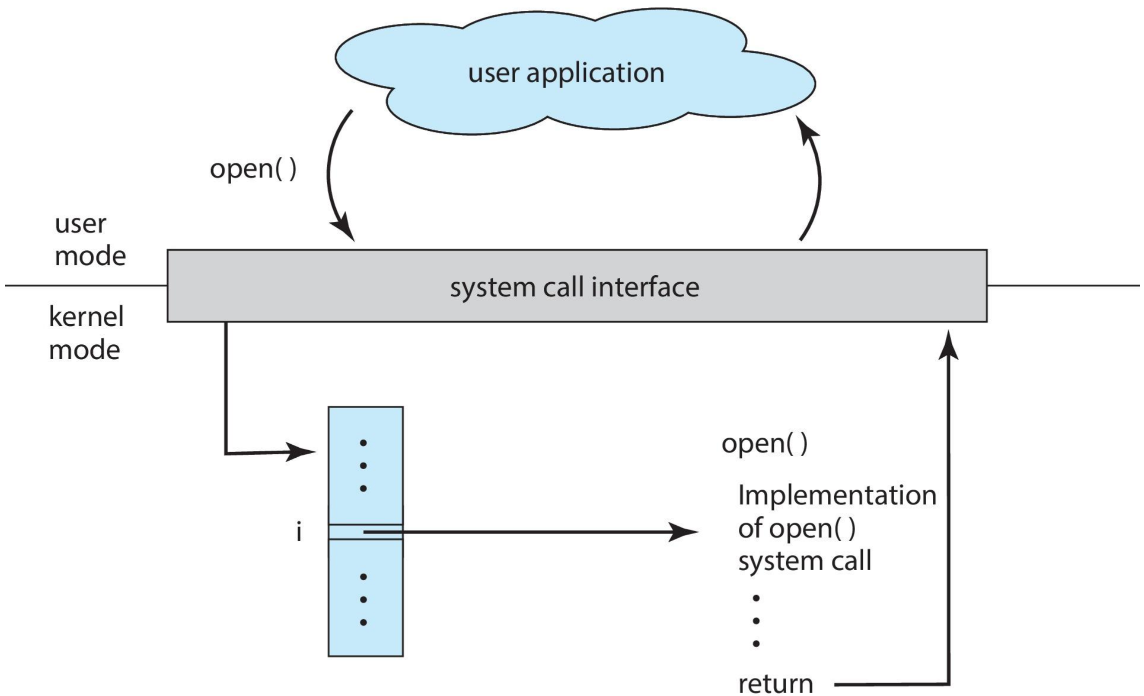


System calls

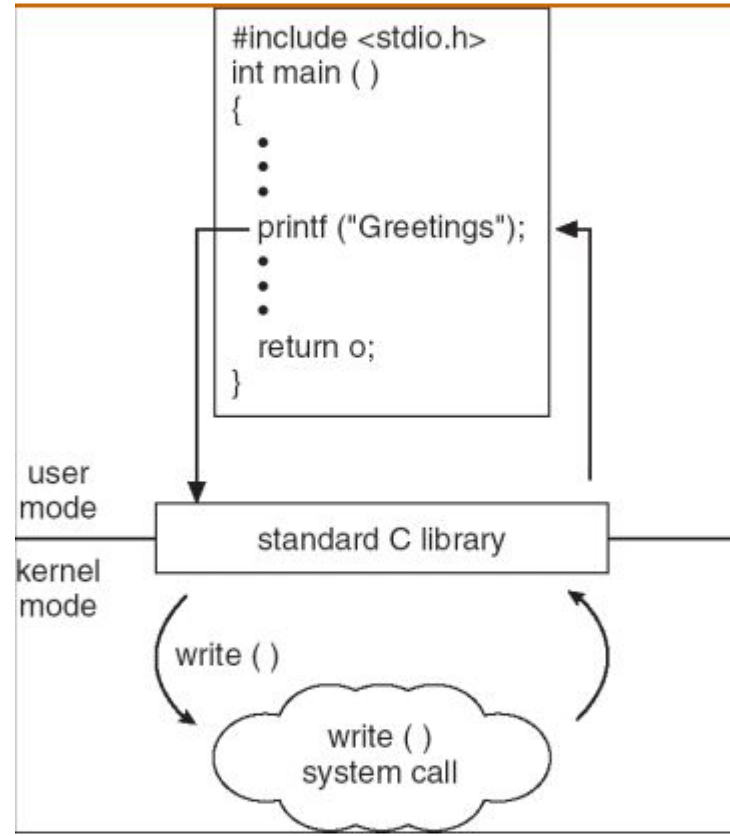
- Goal: Do things application can’t do in unprivileged mode
  - Like a library call, but into more privileged kernel code
- Kernel supplies well-defined system call interface
  - Applications set up syscall arguments and trap to kernel
  - Kernel performs operation and returns result
- Higher-level functions built on syscall interface
  - printf, scanf, fgets, etc. all user-level code
- Example: POSIX/UNIX interface
  - open, close, read, write, ...

System call example

- Applications can invoke kernel through system calls
  - Special instruction transfers control to kernel
  - . . . which dispatches to one of few hundred syscall handlers



- Standard library implemented in terms of syscalls
  - printf – in libc, has same privileges as application
  - calls write – in kernel, which can send bits out serial port



UNIX file system calls

- Applications "open" files (or devices) by name
  - I/O happens through open files
- ***int open(char *path, int flags, /*int mode*/...);***
  - flags: O_RDONLY, O_WRONLY, O_RDWR
  - O_CREAT: create the file if non-existent
  - O_EXCL: (w. O_CREAT) create if file exists already
  - O_TRUNC: Truncate the file
  - O_APPEND: Start writing from end of file
  - mode: final argument with O_CREAT
- Returns file descriptor—used for all I/O to file

Error returns

- What if open fails? Returns -1 (invalid fd)

- **Most system calls return -1 on failure**
  - Specific kind of error in global int errno
  - In retrospect, bad design decision for threads/modularity

- #include <sys/errno.h> for possible values
  - 2 = ENOENT “No such file or directory”
  - 13 = EACCES “Permission Denied”
- perror function prints human-readable message
  - perror ("initfile");

  → “initfile: No such file or directory”

Operations on file descriptors

- int read (int fd, void *buf, int nbytes);
    - Returns number of bytes read
    - Returns 0 bytes at end of file, or -1 on error
- int write (int fd, const void *buf, int nbytes);
    - Returns number of bytes written, -1 on error
- off_t lseek (int fd, off_t pos, int whence);
    - whence: 0 – start, 1 – current, 2 – end
    - ▹ Returns previous file offset, or -1 on error
- int close (int fd);

File descriptor numbers

- File descriptors are inherited by processes
  - When one process spawns another, same fds by default
- Descriptors 0, 1, and 2 have special meaning
  - *0 – “standard input” (stdin in ANSI C)*
  - *1 – “standard output” (stdout, printf in ANSI C)*
  - *2 – “standard error” (stderr, perror in ANSI C)*
  - Normally all three attached to terminal
- Example: type.c
  - Prints the contents of a file to stdout

example

```
void typefile(char *filename) {
    int fd, nread;
    char buf[1024];
    fd = open(filename, O_RDONLY);
    if (fd == -1) {
        perror(filename);
        return;
    }
    while ((nread = read(fd, buf, sizeof(buf))) > 0)
        write(1, buf, nread);
    close(fd);
}
```

Can see system calls using strace utility (ktrace on BSD)

Protection example: CPU preemption

Protection mechanism to prevent monopolizing CPU

- E.g., kernel programs timer to interrupt every 10 ms
  - Must be in supervisor mode to write appropriate I/O registers
  - User code cannot re-program interval timer
- Kernel sets interrupt to vector back to kernel
  - Regains control whenever interval timer fires
  - Gives CPU to another process if someone else needs it
  - Note: must be in supervisor mode to set interrupt entry points
  - No way for user code to hijack interrupt handler
- Result: Cannot monopolize CPU with infinite loop
  - At worst get 1/N of CPU with N CPU-hungry processes

Protection is not security

How can you monopolize CPU?

- Use multiple processes
- For many years, could wedge most OSes with
  - **int main() { while(1) fork(); }**
  - Keeps creating more processes until system out of proc. slots
- Other techniques: use all memory (chill program)

Typically solved with technical/social combination

- Technical solution: Limit processes per user
- Social: Reboot and yell at annoying users
- Social: Ban harmful apps from play store

Protection by Address translation

Protect memory of one program from actions of another

- Definitions
  - Address space: all memory locations a program can name
  - Virtual address: addresses in process' address space
  - Physical address: address of real memory
  - Translation: map virtual to physical addresses
- Translation done on every load, store, and instruction fetch
  - Modern CPUs do this in hardware for speed
- Idea: If you can't name it, you can't touch it
  - Ensure one process's translations don't include any other process's memory

More memory protection

- CPU allows kernel-only virtual addresses
  - Kernel typically part of all address spaces,
    - e.g., to handle system call in same address space
  - But must ensure apps can't touch kernel memory
- CPU lets OS disable (invalidate) particular virtual addresses
  - Catch and halt buggy program that makes wild accesses
  - Make virtual memory seem bigger than physical
    - (e.g., bring a page in from disk only when accessed)
- CPU enforced read-only virtual addresses useful
  - E.g., allows sharing of code pages between processes
  - + many other optimizations
- CPU enforced execute disable of VAs
  - Makes certain code injection attacks harder

Different system contexts

At any point, a CPU (core) is in one of several contexts

- User-level
  - CPU in user mode running application
- Kernel process context
  - running kernel code on behalf of a particular process
    - E.g., performing system call, handling exception (memory fault, numeric exception, etc.)
  - Or executing a kernel-only process (e.g., network file server)
- Kernel code not associated with a process
  - Timer interrupt (hardclock)
  - Device interrupt
  - “Softirqs”, “Tasklets” (Linux-specific terms)
- Context switch code – change which process is running
  - Requires changing the current address space
- Idle – nothing to do (bzero pages, put CPU in low-power state)

Transitions between contexts

- User → kernel process context: syscall, page fault, . . .
- User/process context → interrupt handler: hardware
- Process context → user/context switch: return
- Process context → context switch: sleep
- Context switch → user/process context

Resource allocation & performance

Multitasking permits higher resource utilization

- Simple example:
  - Process downloading large file mostly waits for network
  - You play a game while downloading the file
  - Higher CPU utilization than if just downloading
- Complexity arises with cost of switching
- Example: Say disk 1,000 times slower than memory
  - 1 GiB memory in machine
  - 2 Processes want to run, each use 1 GiB
  - Can switch processes by swapping them out to disk
  - Faster to run one at a time than keep context switching

Useful properties to exploit

- Skew
  - 80% of time taken by 20% of code
  - 10% of memory absorbs 90% of references
  - Basis behind cache: place 10% in fast memory, 90% in slow, usually looks like one big fast memory
- Past predicts future (a.k.a. temporal locality)
  - What’s the best cache entry to replace?
  - If past ≈ future, then least-recently-used entry
- Note conflict between fairness & throughput
  - Higher throughput (fewer cache misses, etc.) to keep running same process
  - But fairness says should periodically preempt CPU and give it to next process