

Lecture 1

January 9, 2018 2:38 PM

- A software system is a collection of specifications agreed upon and coded by several programmers such that multiple elements work together effectively

Lecture 2

January 11, 2018 2:36 PM

Unix & Linux

- Started in the 1960's
- Is the dominant OS format worldwide
- OSX, Ubuntu, Android are all forked off of Unix

Linux

- Any operating system based upon the Linux kernel written by Linus Torvalds in 1991
 - o Linux kernel is a re-implementation of Unix designed from the start to be free and open source
- Is developed by Linus, along with a worldwide community

BIOS

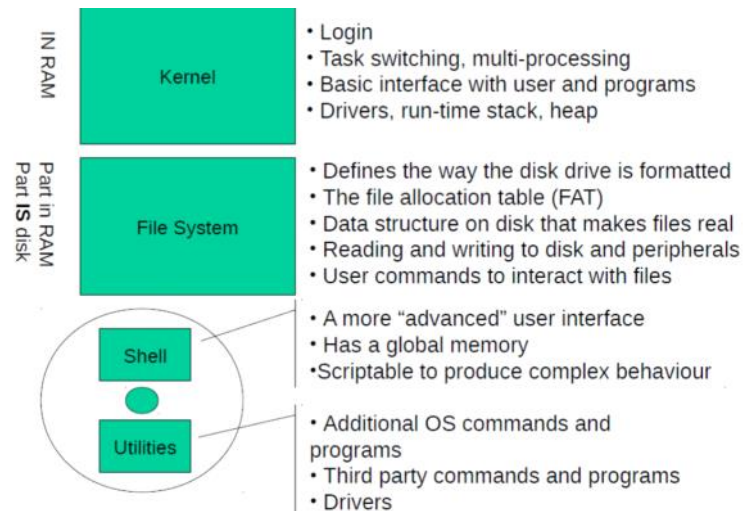
- First device to get power upon boot
- Selects which code to start running next, based on a priority list (boot order)
- A disk is bootable if it has a sequence of data at its very beginning, the Master Boot Record (MBR)
 - o The BIOS reads each device in order until it finds one that's indicated "bootable"
- Can load an OS and begin its execution, but very often, a piece of software known as a boot loader is indicated in the MBR
 - o A common boot loader for use w/ Linux is called GNU Grand Unified Bootloader (GRUB)
 - o Boot loader can provide the option to boot into different OS'
- The Kernel is the name for the part of Linux that runs first and always
 - o It is responsible for loading and running all other programs, so it gets to create the specification of what's allowed, to enforce security, to kill other jobs, to provide access to hardware, etc...
- BIOS>Boot Loader> Kernel

Shell

- Advanced user interface
- Allows you to use the OS via commands
- Scriptable to produce complex behaviour
- Is an entire programming language
 - o Is an interpreted language

Utilities

- Additional OS commands and programs
- Third party
- Drivers



Lecture 3

January 16, 2018 2:26 PM

The C programming language

- Imperative, weakly typed, procedural language, designed to support cross-platform code with minimal requirements
- A compiled language: source code can't be run directly, but rather needs to be processed by a compiler which produces machine code
 - o Contrasts with shell programs and python
 - o Compiled code can run even if there is no compiler on the system
 - o Important for creating low-level components like the kernel
- History
 - o Invented in 1970 by Denis Ritchie
 - o Used to program early versions of UNIX

Lecture 4

January 18, 2018 2:40 PM

Scope

- Local First and Top to Bottom
 - o If a variable exists locally in a scope, C uses the local version. If it does not exist, C goes to look at the enclosing and global levels
 - o A scope is created by brackets: {}
 - o A "local" variable declared in a scope exists from there to the bottom of the function
- Global variables can also be used from other files, with extern keyword

Memory, What's in it?

- The kernel process
- The shell process
- Your programs process'
 - o The programs machine code
 - o Program variables
 - o Required Libraries
 - o Book-keeping info

Lecture 5

January 23, 2018 2:41 PM

Pointers

- Every point in memory(RAM) has an address
- Pointers are variables that store info at an address
 - o `Char* string= "Hello";`
 - o `Int i = 3;`
 - o `Int* ip = &i;` (ip is storing the address of i, not its integer value
 - `&` operator means "address-of"
 - `Float f;` //has type float
 - `Float *fp = &f;` //has type float* "pointer to float"
 - `Float **fp = &fp;` //has type float** "pointer to pointer to float" or "double to pointer to float"
 - o `"*"` Operator means "de-reference" or "get the data at the address"
 - `Float *fp;` //has type float* "pointer to float"
 - `Float f= *fp;` //has type float
- An array is implemented as a series of contiguous(no gaps) memory of the array's type
 - o The array variable holds the address to the start of this memory always
 - e.g.
 - `Int array[4];`
 - `Array[0]=1; array[1]=2; array[2]=3; array[3]=4;`
 - o `Array[0]` and `*array` hold the same value
 - o `Array[2]` and `*(array+2)` are also equivalent
 - o `&(array[3])` and `array+3` store the same value
 - o Unlike a pointer, the array cannot be re-assigned:
 - o `Int *ap = array;`
 - o `Ap = ap+1;` //ALLOWED: moves ap to array[1]
 - o `Array++;` // Does not compile

Lecture 6

January 25, 2018 2:41 PM

Return Codes

- Most of these functions return a value, such as the character or string read
 - o When the input ends or error occurs, we use the value to terminate reading safely
- Special values
 - o EOF (end-of-file, typically -1) is the reason some functions return int instead of char
 - We need a reserved value to indicate something out of the ordinary, else we would always think the value just came from the text
 - o NULL for pointers indicated an error or that there's nothing left to read

Opening a FILE*

- FILE* fopen(const char* filename, const char* mode);
 - o Example: FILE* my_output = fopen("out.txt", "w");
- Always check the returned value. NULL is a special pointer value to indicate there was an error and you cannot use the file

MUST KNOW ACCESS CODE (add b for binary: rb, wb, ab)

- "r"
 - o read: open file for input operations, the file must exist
- "w"
 - o Write: create an empty file for output operations, if a file with the same name already exists, its contents are discarded and the file is treated as a new empty file
- "a"
 - o Append: open file for output at the end of a file, output operations always write data at the end of the file, expanding it
 - Repositioning operations (fseek, fsetpos, rewind) are ignored, the file is created if it does not exist

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main( int argc, char* argv[] )
{
    if( argc < 2 ){
        printf( "Usage: %s <output_name>.\n", argv[0] );
        exit(-1);
    }

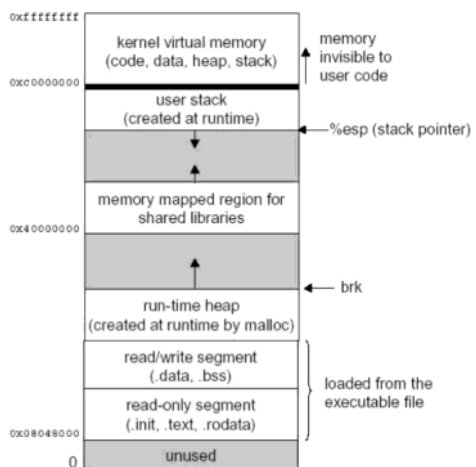
    FILE* fp = fopen( argv[1], "w" );
    if( fp == NULL ){
        printf( "%s failed to open file %s for writing. Check permissions.\n", argv[0], argv[1] );
        exit(-1);
    }

    printf( "Enter lines to place in file, %s. Use ctrl-d to end.\n", argv[1] );

    char user_input[100];

    while( fgets( user_input, 100, stdin ) ){
        fputs( user_input, fp );
    }
}
```

Terminal IO Function	File IO Function	Description
printf(char* format, ...)	fprintf(FILE* f, char* format, ...)	Print formatted output
scanf(char* format, ...)	fscanf(FILE* f, char* format, ...)	Read formatted input
putchar(int c)	fputc(int c, FILE* f)	Print a single character
getchar()	fgetc(FILE* f)	Read a single character (return it)
puts(char* string)	fputs(char*string, FILE* f.)	Print a string
gets(char* string)	fgets(char* string, int bytes, FILE* f)	Read a string



```
int main()
{
    char c;
    float f;
    char string[100];

    printf( "Enter a letter, a number and a string: " );

    scanf( "%c %f %s", &c, &f, string );

    printf( "The letter was %c.\n", c );
    printf( "The number was %f.\n", f );
    printf( "The string was %s.\n", string );

    return 0;
}
```

Lecture 7

January 30, 2018 2:36 PM

Steganography

- Is concerned with concealing the fact that a secret message is being sent

Cryptography

- The practice of protecting the contents of a message alone, steganography is concerned with the fact that a secret message is being sent at all, as well as concealing the contents of the message

Elements required for in-image steganography

- Read and write binary image data
- View the text string in binary form, so we can access one bit at a time
- Ability to modify only a single bit (the LSB) of each pixel
- Ability to extract the LSB again for decoding

Reading a BMP using C

- Do
 - o Check the magic number
 - If it matches very likely follows the rules
 - o File size field: makes it easy to access all of the data
 - o Width and height, allows finding a specific pixel
 - o Opening with code like "rb"
- Don't
 - o Checking for ASCII code values: space, newline, etc...
 - o Attempting to use "atoi" "atof", these are "ASCII to..."
 - o If we open with "r", C will do some of this automatically
 - o fgets, fscanf also typically bad choices
- Once you've got the data
 - o Each colour pixel is stored as an unsigned integer between 0 and 255
 - o Easy to read these one at a time in C and store in unsigned char
 - o We want to use the least significant bit of this value to encode a bit of our secret message
- Shifts (bit_arg<<or>>shift_arg)

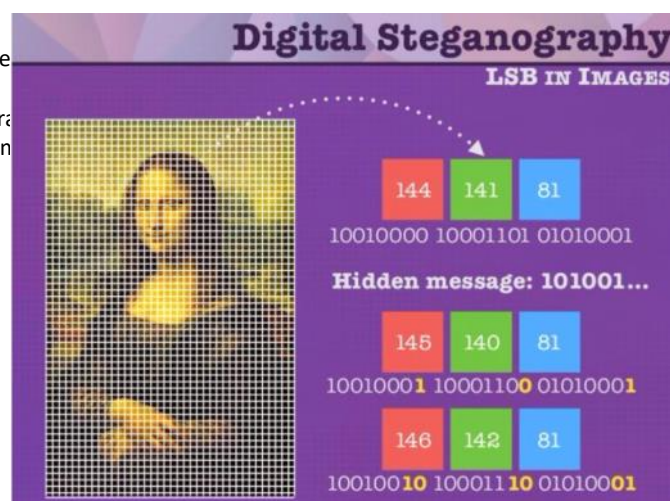
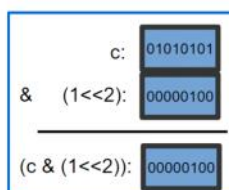
Bit-wise logic:

- left_arg & right_arg
 - Takes the bitwise AND of left_arg and right_arg
- left_arg | right_arg
 - Takes the bitwise OR of left_arg and right_arg
- left_arg ^ right_arg
 - Takes the bitwise XOR of left_arg and right_arg
- ~arg
 - Takes the bitwise complement of arg

- Examples

Bitwise Example #1

- Check the value of bit 3:
char c = 85;
if((c & (1<<2)) > 0)
printf("bit 3 of c is a 1!\n");
else
printf("bit 3 of c is a 0!\n");
- Larger example posted to Github:
 - [bit_reporting.c](#)



Lecture 8

February 1, 2018 2:37 PM

- C is a statically typed language
 - o It is permissive in allowing many type conversions
 - Powerful but dangerous
 - o Oftentimes, viewing data as a different type helps us see "under the hood"
- Safe vs Unsafe Conversions
 - o Type T is wider than type S (and S is narrower than T), if `sizeof(T) >= sizeof(S)`
 - o The narrower type is promoted to the wider type (safe), the wider type is demoted to the narrower type (unsafe)
 - o Examples
 - An int value can be safely promoted to double
 - A double value can not be safely demoted to int
- `sizeof()` returns the number of bytes associated with its argument
- Arithmetic Conversions
 - o If operands of an expression are of different types, the compiler automatically promotes to the higher precision type according to the following hierarchy
 - Int
 - Unsigned
 - Long
 - Unsigned long
 - Float
 - Double
 - Long double
- Explicit Type Casts
 - o Assignment conversions occur when the expression on the right is converted to the type of the left side
 - o The type cast expression (type) <expression>
- Pointer can re-interpret memory
 - o We have seen "pointing" to variables of a type with a pointer of that type allows us to retrieve data normally
- Word Endianness
 - o The order that bytes are stored in memory is only a convention
 - o Most systems we deal with will be little-endian
 - When we write numbers, it is natural to start with the "big-end"
 - But when we do math it is reversed
 - o It is always better to check than to assume
- Different sub-sections of process memory
 - o Kernel space: an interface to operating system functionality (system call) virtually mapped into every process
 - o The stack: Storage for local variables and function parameters associated with each function call. Controlled by the OS
 - Pushed when function called and popped when it returns, after which the memory is no longer accessible
 - o Text segment: The compiled binary resulting from your code
 - o BSS and Data: Stores constant and static (global) variables
 - o The heap: dynamically allocated memory controlled by the program
- Recall: pass-by-reference saves copying and allows changing actual parameters
- C has no built-in pass-by-reference
 - o Always by value
- Pointers are commonly used instead to achieve similar functionality
- Passing Arrays to Functions
 - o Array variables are an exception in that they are always passed by reference
- The Heap: Dynamically Allocated Memory
 - o Allows flexibility when "popping" off stack is not desired
 - o We will use the heap in 206, but we will ignore how C and the operating system organizes heap memory given user program might allocate and deallocate dynamically many times

```
#include <stdio.h>

int main()
{
    char string0[5];
    string0[0] = 'H';
    string0[1] = 'e';
    string0[2] = '\0';
    string0[3] = '\0';
    string0[4] = '\0';
    string0[5] = '\0';

    char string1[5] = "Hello";
    char string2[] = "Hello";
    char *string3 = "Hello";

    printf("Value at string0 is %s.\n", string0);
    printf("Value at string1 is %s.\n", string1);
    printf("Value at string2 is %s.\n", string2);
    printf("Value at string3 is %s.\n", string3);

    printf("Address in string0 is %p.\n", string0);
    printf("Address in string1 is %p.\n", string1);
    printf("Address in string2 is %p.\n", string2);
    printf("Address in string3 is %p.\n", string3);

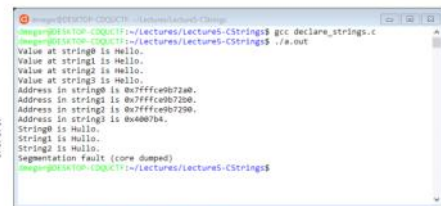
    string0[1] = 'u';
    printf("String0 is %s.\n", string0);

    string1[1] = 'u';
    printf("String1 is %s.\n", string1);

    string2[1] = 'u';
    printf("String2 is %s.\n", string2);

    string3[1] = 'u';
    printf("String3 is %s.\n", string3);

    return 0;
}
```



```
~/Desktop/206/Comp
~/Desktop/206/Comp$ gcc declare_strings.c
~/Desktop/206/Comp$ ./a.out
Value at string0 is Hello.
Value at string1 is Hello.
Value at string2 is Hello.
Address in string0 is 0x7ffffce072a0.
Address in string1 is 0x7ffffce072a0.
Address in string2 is 0x7ffffce072a0.
Address in string3 is 0x7ffffce072a0.
Address in string0 is 0x40007b4.
String0 is Hello.
String1 is Hello.
String2 is Hello.
Segmentation fault (core dumped)
~/Desktop/206/Comp$
```


Lecture 11

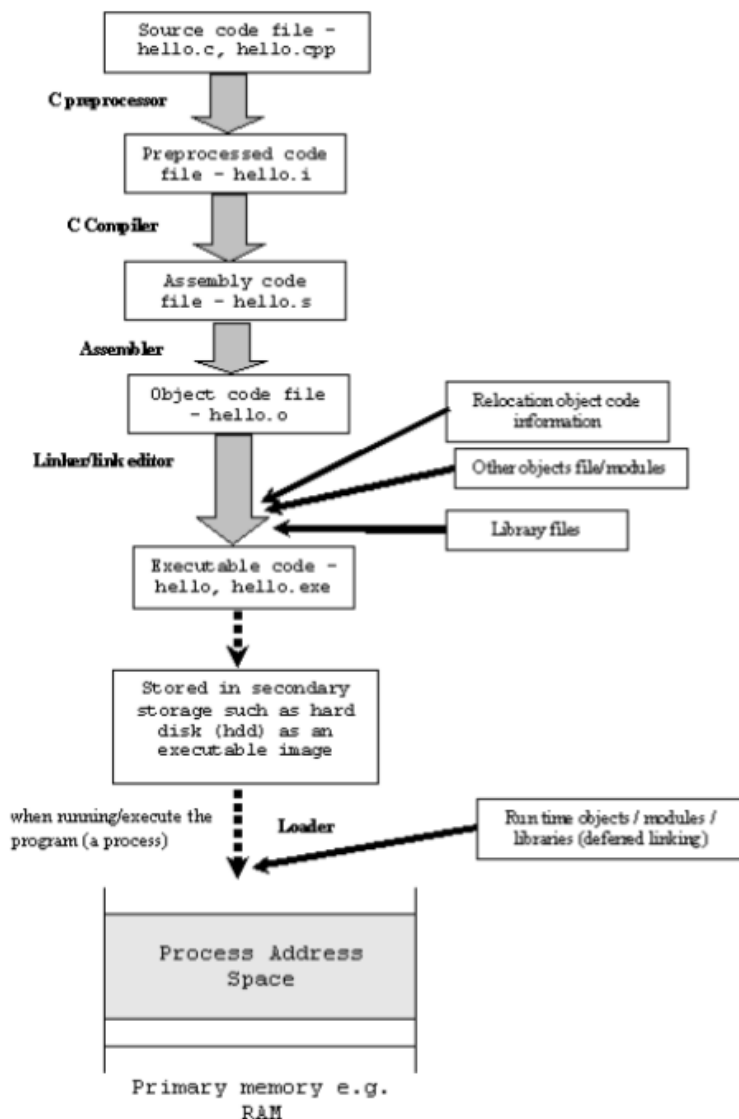
February 15, 2018 2:38 PM

Example: #define debug()

- As mentioned, want to have control over when to print verbosely

```
#ifndef DEBUG
#define debug(x,y) printf(x,y)
#else
#define debug(x,y) /*x,y*/
#endif
```

- Compile with gcc -DDEBUG to see the output, if you leave this flag out, nothing will print -> production code
- Code sample posted on GitHub:
 - [macro_debugging.c](#)



- Creating and Managing larger C programs
- Include <filename.h> at the top

Lecture 12

February 20, 2018 2:40 PM

The C Struct

- Used to create a structure (class)
 - o Fields are equivalent to attributes
- Putting typedef in front of STRUCT defines the struct as a new type of data

New Syntax

- The "arrow" operator ">" is only for structures.
- De-references the pointer and accesses the field all at once:

```
struct PERSON me = { "David", -1 };
struct PERSON *ptr = &me;
ptr->age = ptr->age + 1; // Equivalent to:
(*ptr).age = (*ptr).age + 1;
```

malloc struct Example

```
struct STUDENT
{
    char name[50];
    int age;
    double gpa;
};

struct STUDENT *p;

p = (struct STUDENT *) malloc(sizeof(struct STUDENT));

p->age = 18;
p->gpa = 3.5;

strcpy(p->name, "Mary Smith");

free(p);
p = NULL;
```

calloc works for structs also

```
int i, quantity = 0;
struct STUDENT_REC *p;

printf("Number of students: ");
scanf("%d", &quantity);

p = (struct STUDENT_REC *) calloc(quantity, sizeof(
    struct STUDENT_REC));

for(i=0; i<quantity; i++)
    (p+i)->gpa = 0.0;
```

```
struct COURSE {
    unsigned int numberOfStudent;
    char[100] nameProfessor;
    char[100] buildingName;
    unsigned int roomNumber;
}
```

To use a structure, you need to create a variable of the new type

```
struct COURSE cs206;
```

- You can then fill it with data.

```
cs206.numberOfStudent = 60;
strcpy(cs206.nameProfessor, "Alex");
strcpy(cs206.buildingName, "MacDonald");
cs206.roomNumber = 328;
```

- Or this way:

```
struct COURSE cs206 = {60, "Alex", "MacDonald", 328};
```

fscanf a struct

```
struct PERSON
{
    char name[100];
    int age;
};
```

```
struct PERSON x; // variable
```

```
FILE *p = fopen("something.txt", "rt");
```

```
fscanf(p, "%d", &x.age);
```

fread a struct

```
struct PERSON
{
    char name[100];
    int age;
};
```

```
struct PERSON x; // variable
```

```
FILE *p = fopen(.....);
```

```
fread(x, 1, sizeof(struct PERSON), p);
```

Lecture 13

February 22, 2018 2:31 PM

Structs continued

- Structs themselves cannot be compared, their attributes must be compared instead

Effects of Buffered IO

- Keyboard mode for standard input
 - o The user types many characters, but is able to "go back" by pressing backspace or delete. Our program does not see these interactions, but only gets the full line when enter is pressed
- Efficiency for filesystem interactions
 - o Reading and writing to the hard drive is "slow" (relative to memory or CPU operations). The fact that we can interact with data by byte without our program slowing to a crawl indicates some "buffering" has happened

Pointers to functions

Know the Difference Between These Declarations:

```
int *fn();           this means a function that returns an int*
int (*fn)();         this means a pointer to a function that returns
                     an integer and takes no arguments
```

The second creates a function pointer, which can be used to "point to" any existing function that matches its return and argument types, like this:

```
int return5() { return 5; }
int (*fn)() = return5;
```

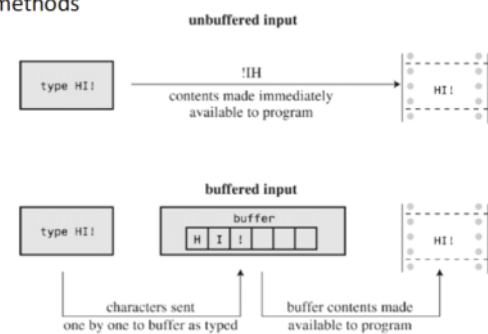
Afterwards, the word `fn` is a valid way to execute the code in the `return5` function (until `fn` might be pointed somewhere else later, which is OK):

```
int x = (*fn)(); // x will equal 5
int y = return5(); // y will return 5
int z = fn();    // this syntax is fine too.
```

A common case: Buffered vs Un-buffered IO

- Each of the standard C IO methods we saw uses buffering

- Why?



Lecture 14

February 27, 2018 2:40 PM

Recall: The UNIX/Linux Shell

- A command-line interpreter that allows us to run built-in programs and those we write

The Shell as a Programming Language

- Shell programs have:
 - o Variables that can be set and used
 - o Several kinds of loops
 - o Functions
 - o Arrays
 - o Input/output features
 - o The ability to launch other shell programs
 - o A pretty convenient debugger (itself)
- Shells do not
 - o Allow low level memory management, compilation to machine code, allow creating complex data structures (not in a convenient way)

Lecture 15

March 20, 2018 2:34 PM

Processes on Linux

- In one terminal, run "tail" and ask it to continue refreshing:
`$ tail -f junk.txt`
Hello world
Do you hear me?
- In a second terminal, start dumping text into the file in some way:
`$ echo "Hello world" >> junk.txt`
`$ echo "Do you hear me?" >> junk.txt`

Starting processes

- A simple way: the `fork()` system call
 - The operating system takes the process that is running and clones it
- Including `fork()` in a C program spawns another process by copying the current one
 - This includes all memory, variables, code, and the process pointer
- The new process starts executing at the line immediately after `fork()`
- `Fork()` returns:
 - Negative if error
 - Zero in the child process
 - A positive value, the child's process ID in the parent

Concepts in scheduling

- Core idea: Because we have only 1 (or a few) CPUs per machine, many processes are "waiting". However, operations like accessing the disk are long compared to running processor instruction, so we have a chance to be efficient!
- A good scheduler is one that provides a high quality of service for each process, measured in one of several ways:
 - Wait time: duration between requesting to start and getting first service
 - Latency: time until completion
 - Throughput: how many resources available per second
- Additionally, we may think about fairness (worst-served process) or overall performance (average of all processes)

What Scheduling does Linux Use?

- Something slightly more complicated than round robin
 - It adds a concept of priority called "niceness". Each process has a niceness level, and the scheduler tries to give more time to less nice processes. This is important, for example, if the user is interacting with one window while others are in the background
 - It estimates the running duration of each process, so that short tasks can pre-empt
 - It interacts with sleeping/waiting processes and IO delays
- This is all called "Completely fair scheduler"

Lecture 16

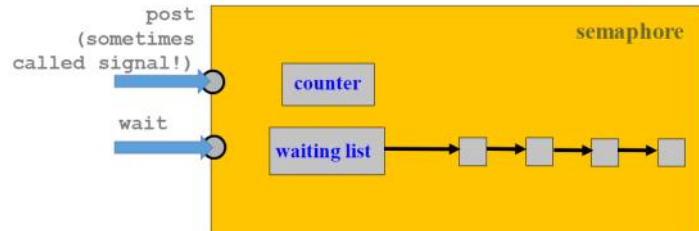
March 29, 2018 2:37 PM

Semaphores

Tie-breaking with semaphores

- Idea: have a central agent (the OS) keep track of who asked first and only let one process execute at a time
- The process that asks first gets to proceed right away, and must then also say when its finished
- All other processes are blocked by the central agent, and allowed to proceed one at a time

- A **semaphore** is an object that consists of a **counter**, a **waiting list** of processes and two **methods** (e.g., functions): **post** and **wait**.



Semaphore Method: wait

```
void wait(sem S)
{
    S.count--;
    if (S.count < 0) {
        add the caller to the waiting list ;
        block();
    }
}
```

- After decreasing the counter by 1, if the counter value becomes negative, then
 - ❖ add the caller to the waiting list, and then
 - ❖ block itself.

Semaphore Method: post

```
void post(sem S)
{
    S.count++;
    if (S.count <= 0) {
        remove a process P from the waiting list;
        resume ( P );
    }
}
```

- After increasing the counter by 1, if the new counter value is not positive, then
 - ❖ remove a process **P** from the waiting list,
 - ❖ resume the execution of process **P**, and return