

# CSCI 4061: Introduction

Chris Kauffman

*Last Updated:  
Wed Jan 20 03:58:53 PM CST 2021*

# CSCI 4061 Lec 001 - Logistics

## Goals Today

- ▶ Motivation
- ▶ Unix Systems Programming
- ▶ C programs
- ▶ Course Mechanics

## In and Out of Class

- ▶ Common Misconception: Everything you need to know happens in lecture
- ▶ Truth: Much of what you'll learn will be when you're reading and **doing** things on your own

## Reading

### REQUIRED:

Stevens and Rago, *Advanced Programming in the UNIX Environment*

- ▶ **Required** textbook
- ▶ Will go somewhat out of order
- ▶ Read: Ch 1

### OPTIONALLY:

- ▶ Robbins and Robbins, *Unix Systems Programming*
- ▶ Silberschatz, *OS Concepts*

Won't deal with either of these in much detail

# Logistics Issues

## Course Materials

<https://www.cs.umn.edu/~kauffman/4061/schedule>

- ▶ Will usually post slides prior to meetings
- ▶ Open today's slides so you can see the following questions
- ▶ As students go into Breakout rooms, won't be able to see MY video feed: examine the slides yourself

## Breakout Rooms

- ▶ During meetings, will use Zoom breakout rooms for discussions
- ▶ Breakout Rooms will be semi-permanent: folks you meet today are your discussion **teammates**, get to know them
- ▶ During discussions, **write down your answers so you can share your screen later**
- ▶ Designate a team member that has this capability (e.g. computer with video sharing)

## Exercise: OS Ice Breaker

Write your answers in a text file so 1-2 teams can screen-share their answers

1. Who is writing answers down so they can share?
2. What Operating System(s) do you use frequently?
3. One thing you like and one that you dislike about this OS?
4. Something you should know about me

Answer the following questions for **each team member**

An example is show to the right

Korra

- I use OS K most of the time
- I like that it has a fancy GUI but HATE updates are always installing
- I can bend anything

Mako

- FireOS on my tablet!
- It's runs fast but also hot, literally
- Sometimes I think I brood too much

Bolin

- DirtLinux
- It's super stable but bit dusty
- Mako is my brother!

Asami

- I use OS K for fun but FutureOS for work/school
- OS K has a slick GUI but FutureOS makes it easier to write code
- I'm working while in school

## Exercise: Plethora of Operating Systems

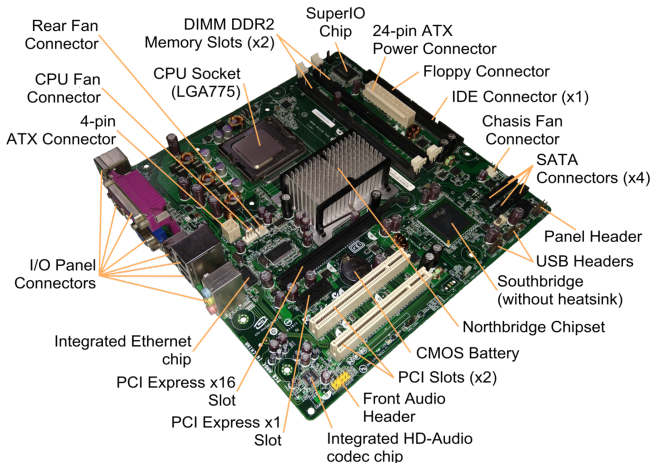
1. What is the job of the operating systems?
2. Why do we even need them?
3. Why do you have a required class about them?

Write your answers down so 1-2 teams can screen-share their answers

1. *An OS basically does . . . .*
  2. *OSs make computing easier by . . .*
  3. *???*
- ▶ Team consensus answers, don't need separate answers for each member
  - ▶ Including differing ideas if there is no consensus

## Answers: Responsibilities of the OS?

- ▶ Interact with **hardware** like CPU, hard drives, mice, monitors, DRAM, network interface card etc. is tedious / difficult
- ▶ Electrical signals / voltage changes / timing / protocols
- ▶ Some hardware follows standard protocols, others are special



## Answers: Responsibilities of the OS?

OSs play many parts but primarily they glue Hardware and Software together

Create a "virtual machine" on top of hardware

- ▶ OS creates an abstraction layer over hardware
- ▶ Similar programming interface regardless of underlying hardware environment:
- ▶ Phones, Laptops, Cars, Planes, Nuclear Reactors
- ▶ all see **Processes, Memory, Files, Network**

Enforce Discipline / Referee Software

- ▶ Limit damage done by one party to another
- ▶ Processes communicate along fixed lines
- ▶ Multiple users must explicitly share info
- ▶ Shared resources are managed

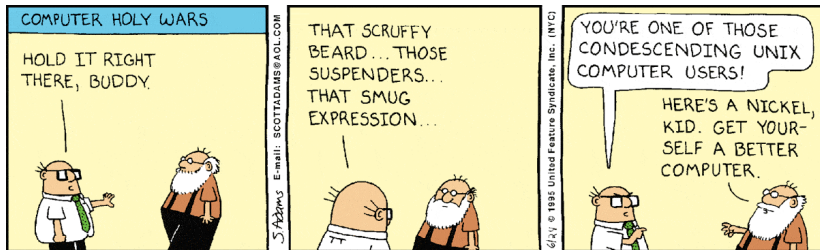
## Why Unix?



- ▶ All OSs on this page owe some influence to Unix
- ▶ Except maybe MS-DOS



# Unix is Old, Tested, and often Open



- ▶ Developed from the 70s, honed under pressure from academia and industry for widely varying uses
- ▶ Among the first projects to benefit from shared source code
- ▶ Philosophy: Simple, Sharp tools that Combine Flexibly
- ▶ Keep the **Kernel** functionality small but useful
- ▶ Abstractions provided in Unix are well-studied, nearly universal

# The Unix "Virtual" Machine

Unix **Kernel** provides basic facilities to manage its high level abstractions of hardware, translate to actual hardware

- ▶ Link: [Interactive Map of the Linux Kernel](#)
- ▶ Examples Below

## Processes: Executing Code

- ▶ Create new processes
- ▶ Status of other processes
- ▶ Pause until events occur
- ▶ Create/Manage threads within process

## Process Communication

- ▶ Messages between processes
- ▶ Share memory / resources
- ▶ Coordinate resource use

## File System: Storage / Devices

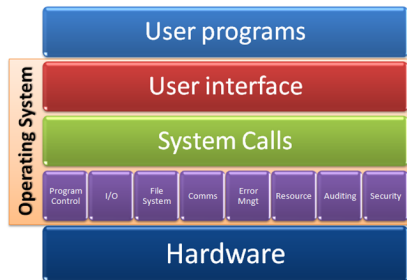
- ▶ Create / Destroy Files
- ▶ `read()` / `write()`
- ▶ Special files for communication, system manipulation

## Networking

- ▶ Open sockets which connect to other machines
- ▶ `send()/recv()` data over connections

# Outsides vs Insides of the OS

- ▶ Operating Systems are layered like everything else in computer science
- ▶ 4061: outer layer
- ▶ 5103: inner layers
- ▶ EE Degree: bottom layer



## CSCI 4061

- ▶ Systems Programming
- ▶ Use functionality provided by kernel
- ▶ Gain some knowledge of internals but focus on external practicalities

## CSCI 5103

- ▶ Creation of a kernel / OS internals
- ▶ Theory and practice of writing / improving operating systems
- ▶ Implement system calls

## System Calls : The OS's Privilege

- ▶ User programs will never actually read data from a file
- ▶ Instead, will make a request to the OS to read data from a file
- ▶ Usually done with a C function like in

```
int nbytes_read = read(file_des, in_buf, max_bytes);
```
- ▶ After a little setup, OS takes over
- ▶ Elevates the CPU's privilege level to allow access to resources not normally accessible using assembly instructions
  - ▶ Modern CPUs have security models with normal / super status
  - ▶ Like `sudo make me a sandwich` for hardware
- ▶ At completion of `read()` CPU drops back to normal level
- ▶ User program now has stuff in `in_buf` or an error to deal with
- ▶ Same framework for process creation, communication, I/O, memory management, etc.: make a **system call** to request an OS service

# Details of System Calls

```
1  ## 32-bit write linux system call in assembly (hello32.s)
2  _start:
3      movl    $4, %eax        # system call number for write: 4,
4      movl    $1, %ebx        # first arg: file descriptor, stdout = 1
5      movl    $msg,%ecx       # second arg: address of message to write
6      movl    $13, %edx       # third arg: message length, 13 bytes
7      int     $0x80           # interrupt to call kernel
8      # write(1, message, 13) // equivalent C call in hello.c
9
10 ## 64-bit write linux system call in assembly (hello64.s)
11 _start:
12     movq    $1, %rax        # system call number for write: 1
13     movq    $1, %rdi        # first arg: file descriptor, stdout = 1
14     movq    $msg,%rsi       # second arg: address of message to write
15     movq    $13, %rdx       # third arg: message length, 13 bytes
16     syscall                # make a system call, x86-64 convention
17     # write(1, message, 13) // equivalent C call in hello.c
```

Call	x86_64	i386
read	rax = 0	eax = 3
write	1	4
open	2	5
close	3	6
stat	4	106
fork	57	2

- ▶ Linux has ~300+ system calls provided by the kernel
- ▶ C/Assembly calls for each

Why system calls like this?

Write down answers with team

# Answers: Why System Calls like this? 1 / 2

## Broad Reason 1: Control + Safety

- ▶ System call allows OS to control access to shared/sensitive resources
- ▶ If user programs could directly access/modify such resources, **bad stuff** can happen such as...
  - ▶ Read other users' files and process memory (security)
  - ▶ Steal CPU / memory / disk space from other users (resource management)
  - ▶ Mess up hardware like printers or network by sending them bad data, screw up OS by clobbering critical files/memory (safety / stability)
  - ▶ Shut down a machine terminating other user programs (fairness)
- ▶ The OS layer enforces discipline for the above
- ▶ Notice some properties pertain to any system while others are relevant to **shared computer systems**

## Answers: Why System Calls like this? 2 / 2

### Broad Reason 2: Portability + Universality

- ▶ A portable OS runs on many different kinds of hardware (processor, memory, disks, etc.)
- ▶ Allows many different devices to be supported (laptop, desktop, watch, phone, dog, etc.)
- ▶ OS should provide system calls that are
  - ▶ Not too hard to implement efficiently
  - ▶ Relevant to many hardware devices
  - ▶ Useful to application programmers
- ▶ Port OS to new hardware → applications don't need to change as they use system calls

# Distinction of Application vs Systems Programming

*The primary distinguishing characteristic of systems programming when compared to application programming is that application programming aims to produce software which provides services to the user directly (e.g. word processor), whereas systems programming aims to produce software and software platforms which provide services to other software, are performance constrained, or both.*

*System programming requires a great degree of hardware awareness. Its goal is to achieve efficient use of available resources, either because the software itself is performance critical (AAA video games) or because even small efficiency improvements directly transform into significant monetary savings for the service provider (cloud based word processors).*

– [Wikipedia: Systems Programming](#)

In short: systems programmers write the code between the OS and everything else. *But*, systems vs application is more of a continuum than a hard boundary.



# General Topics Associated with Systems Programming

- Concurrency** Multiple things can happen, order is unpredictable
- Asynchrony** An event can happen at any point
- Coordination** Multiple parties must avoid deadlock / starvation
- Communication** Between close entities (threads/processes) or distant entities (network connection)
  - Security** Access to info is restricted
- File Storage** Layout of data on permanent devices, algorithms for efficient read/write, buffering
  - Memory** Maintain illusion of a massive hunk of RAM for each process (pages, virtual memory)
- Robustness** Handle unexpected events gracefully
  - Efficiency** Use CPU, Memory, Disk to their fullest potential as other programs are built from here

In our projects, we'll hit on most of these.

## Assumption: You know some C

- ▶ CSCI 2021 is a prereq, covers some hardware, basic C programming and interaction with hardware
- ▶ Assume that you know C syntax, basic semantics
- ▶ Why C vs other languages?

## Computers are well-represented in C

*You just have to know C. Why? Because for all practical purposes, every computer in the world you'll ever use is a von Neumann machine, and C is a lightweight, expressive syntax for the von Neumann machine's capabilities.*

– Steve Yegge, [Tour de Babel](#)

## C and Unix Go Way Back

*Aside from the modular design, Unix also distinguishes itself from its predecessors as the first portable operating system: almost the entire operating system is written in the C programming language that allowed Unix to reach numerous [hardware] platforms.*

– [Wikipedia: Unix](#)

## Exercise: Recall these C things

### Odd Teams

- ▶ Two different syntaxes to loop (repeat code)
- ▶ The meaning of void
- ▶ struct: aggregate, heterogeneous data
- ▶ Pointers to and Address of variables
- ▶ malloc() and free()
- ▶ Dynamically allocated arrays and structs
- ▶ Stack versus heap allocation

### Even Teams

- ▶ #define : Pound define constants
- ▶ Local scope, global scope
- ▶ Pass value vs pass reference
- ▶ printf() / fprintf() and format strings
- ▶ scanf() / fscanf() and format strings
- ▶ Commands to compile, link, execute

## Answers: Recall these C things

- ▶ A good C reference will introduce preceding aspects of C
- ▶ Kernighan and Ritchie's **The C Programming Language** does so, may be worth picking up a copy
- ▶ The remaining demos cover some of these things to refresh
- ▶ **Make sure you get comfortable with all of them quickly** as C programming is a **prerequisite** for 4061
- ▶ HW01 has some additional C programs to inspect
- ▶ Lab01 will review some C programming techniques

## Exercise: Actual C Code

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    long n = 1;
    void *mem = NULL;
    while( (mem = malloc(n)) != NULL){
        printf("%12ld bytes: Success\n",n);
        free(mem);
        n *= 2;
    }
    printf("%12ld bytes: Fail\n",n);
    n /= 2;

    long kb = n / 1024;
    long mb = kb / 1024;
    long gb = mb / 1024;

    printf("\n");
    printf("%12ld b limit\n",n);
    printf("%12ld KB limit\n",kb);
    printf("%12ld MB limit\n",mb);
    printf("%12ld GB limit\n",gb);
    return 0;
}
```

1. Describe at a high level what this C program does
2. Explain the line  
`while( (mem = malloc(n)) != NULL){`  
in some detail
3. What kind of output would you expect on your own computer?

## Answers: Actual C Code

```
1 // max_memory.c: test the total memory available in a single malloc by
2 // repeatedly increasing the limit of the request
3
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main(){
8     long n = 1;                // int cannot hold large enough numbers
9     void *mem = NULL;         // Pointer to memory
10    while( (mem = malloc(n)) != NULL){ // allocate and check result
11        printf("%12ld bytes: Success\n",n); // %ld to print a long, %d for int
12        free(mem);            // free last allocation
13        n *= 2;                // double size of next request
14    }                          //
15    printf("%12ld bytes: Fail\n",n); // failed last allocation, no need to free
16    n /= 2;                    // back up one step for max success
17
18    long kb = n / 1024;        // sizes of allocations
19    long mb = kb / 1024;
20    long gb = mb / 1024;
21
22    printf("\n");
23    printf("%12ld b limit\n",n); // Output human readable sizes
24    printf("%12ld KB limit\n",kb);
25    printf("%12ld MB limit\n",mb);
26    printf("%12ld GB limit\n",gb);
27    return 0;                  // return 0 to indicate successful completion
28 }
```

## Exercise: C Program with Input

```
typedef struct int_node_struct {
    int data;
    struct int_node_struct *next;
} int_node;
int_node* head = NULL;

int main(int argc, char **argv){
    int x;
    FILE *input = fopen(argv[1], "r");
    while(fscanf(input, "%d", &x) != EOF){
        int_node *new = malloc(sizeof(int_node));
        new->data = x;
        new->next = head;
        head = new;
    }
    int_node *ptr = head;
    int i=0;
    printf("\nEnter list\n");
    while(ptr != NULL){
        printf("list(%d) = %d\n", i, ptr->data);
        ptr = ptr->next;
        i++;
    }
    fclose(input);
    return 0;
}
```

- ▶ What data structure is being used?
- ▶ Are there any global variables?
- ▶ What's going on here:  

```
new->data = x;
new->next = head;
```
- ▶ Where do input numbers come from?
- ▶ In what order will input numbers be printed back?
- ▶ Does the program have a memory leak? (What is a memory leak?)

## Answers: C Program with Input

```
1 // read_all_numbers_file.c: simple demonstration of reading input from
2 // a file using a linked list with dynamic memory allocation.
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 typedef struct int_node_struct {                               // Definition of a node
7     int data;                                                 // integer data
8     struct int_node_struct *next;                             // link to another node
9 } int_node;
10
11 int_node* head = NULL;                                       // global variable, front of list
12
13 int main(int argc, char **argv){
14     int x;
15     FILE *input = fopen(argv[1], "r");                       // open input file named on command line
16     while(fscanf(input, "%d", &x) != EOF){                   // read a number, check for end of input
17         int_node *new = malloc(sizeof(int_node));           // allocate space for a node
18         new->data = x;                                       // set data, -> derefernces and sets
19         new->next = head;                                    // point at previous front of list
20         head = new;                                         // make this node the new front
21     }
22     int_node *ptr = head;                                    // prepare to iterate through list
23     int i=0;
24     printf("\nEnter list\n");
25     while(ptr != NULL){                                      // iterate until out of nodes
26         printf("list(%d) = %d\n", i, ptr->data);             // print data for one node
27         ptr = ptr->next;                                     // move pointer forward one node
28         i++;
29     }
30     fclose(input);                                          // close the input file
31     return 0;                                              // Should free list but program is ending
32 }                                                         // so memory will automatically return to system
```



## Slides, Code, Videos On the Course Site

- ▶ Canvas has links to course materials
- ▶ Lecture slides will be available before lecture, may be updated after with corrections / additions
- ▶ Code we use in class will also be available
- ▶ Take your own notes but know that resources are available
- ▶ Will be recording all Lecture meetings and posting to Youtube for later viewing

# Course Mechanics

See separate slides for specific course mechanics