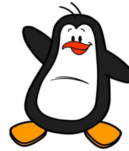


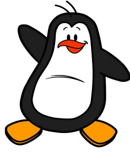
# Lecture # 1.0
# Introduction to the Course

## Course: Advanced Operating System

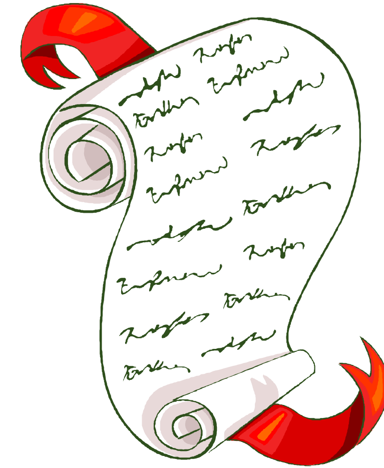
**Instructor: Arif Butt**

**Punjab University College of Information Technology (PUCIT)**
**University of the Punjab**

Source Code files available at: **`https://bitbucket.org/arifpucit/spvl-repo/src`**
Lecture Slides available at: **`http://arifbutt.me`**



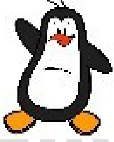
# Today's Agenda



- Course Info and Protocols
- Discussion on Course Outline
- Overview of Operating System
- Review of Computer Hardware
- Learning Environment
- Accessing OS Services
- How System call work (Behind the curtain)

# Course Info

- **Textbook(s):** Advanced Programming in UNIX Environment, 3rd, Ritchard Stevens
  Distributed Systems, 4th Ed, George Coulouris
  UNIX: The Text Book, 3rd Ed, Mansoor, Robert Koretsky
- **Lectures Website:** http://arifbutt.me
- **Resources Website:** http://arifbutt.me
- **Grades Website:** http://online.pucit.edu.pk
- **Prerequisites (Can be covered from Video Lecs):** OS with Linux, COAL, Inter-networking with Linux, C Refresher
- **Office:** Building-C, PUCIT (NC)
- **Students Counseling hours:**
  Mon: 1130 hrs – 1330 hrs
  Wed: 1130 hrs – 1330 hrs
- **24 hour turnaround for email:** arif@pucit.edu.pk





# Who cares to get an A

Final exam: 40

Mid-exam: 35

Sessionals: 25

- Quizzes: 30%
- Programming Assignments: 30%
- Research Papers: 40%

## M.phil.

Minimum GP to pass a course: GP >= 2.3 [C+ or 61 mks]

Degree Completion Requirement: CGPA >= 2.5

Probation: 2.3 <= CGPA < 2.5 [Only one probation allowed]

Dropped out: CGPA < 2.3

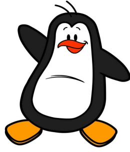
## Ph.D.

Minimum GP to pass a course: GP >= 2.7 [B- or 65 mks]

Degree Completion Requirement: CGPA >= 2.8

Probation: 2.8 <= CGPA < 3.0

Dropped out: CGPA < 2.8





# Course Outline

# What we will do in this course?

## **Module 1**: Preparing your Tool Box (Review)

- **Lecture 1.1:** UNIX C compilation tools (`gcc`, `gdb`, `nasm`, `readelf`, `objdump`, `od`, `nm`, `ldd`, `ldconfig`). Creating your own Static and Dynamic libraries
- **Lecture 1.2:** UNIX `make`, `autoconf` and `cmake` utilities
- **Lecture 1.3:** Versioning Systems (`git`)
- **Lecture 1.4:** How a C program starts and terminates
- **Lecture 1.5:** Understanding Process Stack (Behind the curtain)
- **Lecture 1.6:** Understanding Process Heap (Behind the curtain)

**Reading Assignments:**

- S. Kehsav, "How to read a paper".
- Dennis M. Ritchie and Ken Thompson, "The UNIX Time-Sharing System".

## **Module 2**: File System Architecture

- **Lecture 2.1:** UNIX universal I/O model and FSA
- **Lecture 2.2:** Design and code of UNIX `ls` utility
- **Lecture 2.3:** Design and code of UNIX `more` utility
- **Lecture 2.4:** Digging out information from system files (`uname, who, time`)
- **Lecture 2.5:** Programming the Terminals
- **Lecture 2.6:** UNIX I/O Models (Blocking, Non-Blocking, Multiplexed, Signal Driven, Asynchronous)

**Reading Assignments:**

- Jerome H. Saltzer, "Protection and the Control of Information Sharing in Multics".
- Marshall K., William N., Samuel J., Robert S.,"A Fast File System for UNIX"

# What we will do in this course?

**Module 3**: OS Structures, Virtualization, Kernel Compilation

- **Lecture 3.1:** Introduction to Operating System Structures (DOS, Monolithic, Microkernel, SPIN, Exokernel, L3 Microkernel)
- **Lecture 3.2:** Virtualization (Memory, CPU and Device)
- **Lecture 3.3:** Linux Kernel Compilation
- **Lecture 3.4:** Linux Loadable Kernel Modules

**Reading Assignments:**

- Edsger W. Dikjstra, “The Structure of the "THE"-Multiprogramming System”.
- Brian et.el., "SPIN: An extensible Microkernel for Application-specific Operating System Services”.
- Emin et.el., “Writing an Operating Ssystem with Modula-3”.

# What we will do in this course?

## **Module 4**: Process Management and Scheduling

- **Lecture 4.1:** Process IDs, trees, chains, fans, zombies, orphans, groups, sessions and controlling terminals
- **Lecture 4.2:** Design and coding UNIX daemon processes
- **Lecture 4.3:** Achieving Parallelism using Threads
- **Lecture 4.4:** System-VR3 and System-VR4, Linux O(1), and CFS Scheduler

**Reading Assignments:**

- M. A. Butt and M. Akram, “A novel fuzzy decision-making system for cpu scheduling algorithm”.
- Josh Aas, “Understanding the Linux 2.6.8.1 CPU Scheduler”.
- A white paper on systemd in SUSE® Linux Enterprise 12.

# What we will do in this course?

## **Module 5**: Inter Process Communication

- **Lecture 5.1:** UNIX Signals
- **Lecture 5.2:** Design and code of UNIX pipes and fifos
- **Lecture 5.3:** UNIX message queues and shared memory
- **Lecture 5.4:** UNIX memory mapped files

**Reading Assignments:**

- David L. Presptto, Dennis M. Ritchie, "Interprocess communication in the ninth edition unix system".

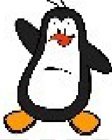
# What we will do in this course?

## Module 6: Synchronization

- **Lecture 6.1:** Synchronization using mutex and condition variables
- **Lecture 6.2:** Synchronization using POSIX semaphores
- **Lecture 6.3:** Synchronization in Parallel and Distributed Systems

**Reading Assignments:**

- C.A.R. Hoare, "Monitors: An Operating System Structuring Concept".
- Paul E. McKenney, et.al., "Read Copy Update".



# What we will do in this course?

## Module 7: Concurrent, Parallel and Distributed Systems

- **Lecture 7.1:** Overview of Computer Networks
- **Lecture 7.2:** Overview of parallel and distributed systems
- **Lecture 7.3:** Clint Server Programming using UNIX socket API
- **Lecture 7.4:** Writing Concurrent servers
- **Lecture 7.5:** Remote Procedure Calls
- **Lecture 7.6:** Lamport Clocks
- **Lecture 7.7:** Global Memory Systems
- **Lecture 7.8:** Distributed Shared Memory Systems
- **Lecture 7.9:** Distributed File Systems
- **Lecture 7.10:** Giant Scale Services and Map Reduce

**Reading Assignments:**

- Andrew and Bruce, “Implementing Remote Procedure Calls”.
- Leslie Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System”, Communications of the ACM, 21, 7, pgs. 558-565, July 1978.
- Russel Sandberg, et.al., “The Sun Network File system: Design , Implementation and Experiences”.
- Jeffrey and Sanjay, “MapReduce: Simplified Data Processing on Large Clusters”.

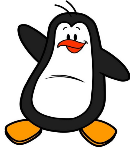
# What we will do in this course?

## Module 8: Cyber Security

- **Lecture 8.1:** Overview of Cyber Security
- **Lecture 8.2:** Stack based buffer overflow and mitigation techniques
- **Lecture 8.3:** Controlling flow of process execution using PEDA
- **Lecture 8.4:** Writing Shell Code using assembly language, metasploit framework and msfvenom
- **Lecture 8.5:** Injecting shell codes in vulnerable programs
- **Lecture 8.6:** Bypassing exploit mitigation techniques

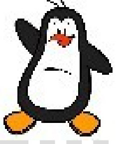
**Reading Assignments:**

- Aleph One, “Smashing The Stack For Fun And Profit”.
- Piotr Bania, “Security mitigations for Return-Oriented Programming Attacks”.
- Ștefan, and Daniel Zota, “Exploiting stack-based buffer overflow using modern day techniques techniques”.
- Andrea et.el., “Hacking Blind”.



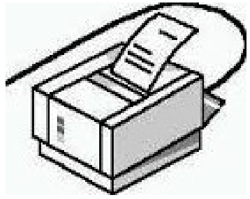


# Overview of Operating Systems



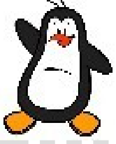


# Role of Operating System





```
int main()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
    return 0;
}
```





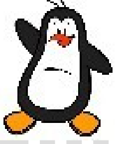
# Role of Operating System

Scanner
Speaker
USB Stick
Ethernet

P11 P9 P2 P1
P10 P8 P3 P4
P7 P6 P5

**Lots of users**
**Lots and lots of programs**
**Lots of disk files**
**Lots of devices**
**Lots and lots and lots of connections**

**Who is managing all these resources and connecting various devices to correct programs**



# Role of Operating System

Scanner

Speaker

USB Stick

Ethernet

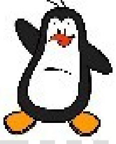
P11 P9 P2 P1

P10 P8 P3 P4

P7 P6 P5

Operating System Kernel

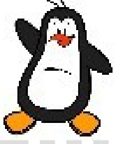
Role of Operating System is to manage all these resources and to connect various devices to the correct programs



# Role of Operating System



- All connections are done through Kernel, and no direct intervention by user
- Program request services from the OS, which contains code to provide services
- An operating system provides these services via its API

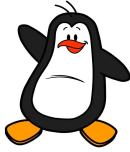


# Role of Operating System

Program

Internet

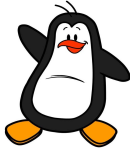
Program





# Operating Systems

- Operating Systems are written for a particular or for multiple h/w architectures.
- Operating System (kernel) is a *program* running at all times on the computer, that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware.
- Operating System is also called a resource manager that manages the hardware/software resources and allocates/deallocates these to applications by using time multiplexing or space multiplexing techniques.
- Objective of an OS is to tame the wild hardware, so that people can get something useful out of it.
- Some famous desktop operating systems in no particular order are Mac, MS Windows, UNIX, PC-BSD, Solaris, Linux, ...
- OSs for hand held devices are android, iOS, blackberry, symbian, ...





# Power of Abstraction

- An abstraction means a well defined interface that hides all the details within a sub-system. This table describes the hierarchy of abstraction in a computer system

| |
|---|
| **Applications** (written in high level languages) |
| **System Softwares** (OS, Compilers etc) |
| **Instruction Set Architecture** |
| **Machine Organization** (Data path and Control) |
| **Seq and Cmb Circuits** |
| **Logic Gates** |
| **Transistors** |
| **Electrons and Holes** |

Get in a taxi and tell the driver, "Take me to the airport"

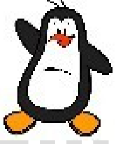
How google earth application use this stack to have an interactive session with the user?

For the abstraction to work it is assumed that every thing about the details is just fine. What if every thing about the details is not just fine? Then to be successful our ability to abstract must be combined with our ability to un-abstract. (Stories)

# Operating System Abstractions

Select the choices that apply to the functionalities provided by an Operating System?

- OS is a resource manager
- OS provides consistent interface to the hardware resources like CPU, memory
- OS schedules applications on the CPU
- OS stores personal information like credit card numbers, email addresses...



# What is an Operating System

- Provide protected access to h/w resources
- Arbitrate among competing requests

well defined APIs for OS services →

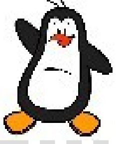
App Code

request

OS code

response ← gets services from OS via APIs

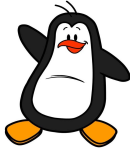
- What happens when you double click a word file icon on desktop?
- What happen when you click a point on the globe in a google earth application?



# Catering to Resource Requirements

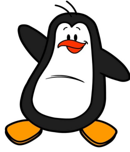
- Resource needs of applications
  - CPU, Memory, peripheral devices
- Application launch time
  - Know how to create memory foot print

- Applications may ask for additional resources at run time



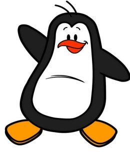


# Review of Computer Hardware



# Computer Hardware

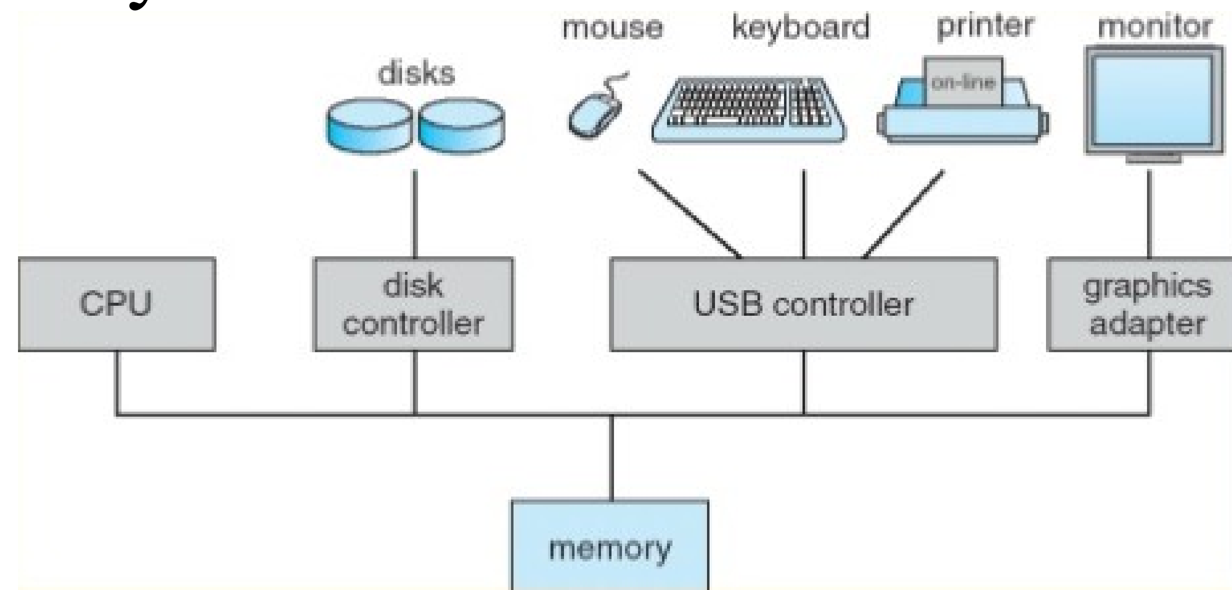
ENIAC (the Electronic Numerical Integrator and Calculator), a general purpose electronic computer that could be reprogrammed for different tasks. It was designed and built in 1943–1945 at the University of Pennsylvania by Presper Eckert and his colleagues. It contained more than 17,000 vacuum tubes. It was approximately 8 feet high, more than 100 feet wide, and about 3 feet deep (about 300 square feet of floor space). It weighed 30 tons and required 140 kW to operate.



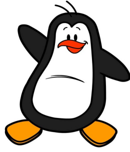


# Computer Hardware

**$100 Question:** Does the internal organization of different computer systems vary vastly?




- Desktop machines
- Laptop machines
- Tablets
- Smart phones
- High end server machines
- Nodes in a data center providing cloud services



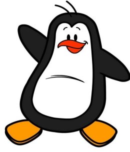


# Computer Hardware

x86, ia64, powerpc, mips, arm, sun sparc, alpha, motorola, cell, IBM S/390,

CPU
Clock Generator
Front-side bus
Graphics card slot
Chipset
High-speed graphics bus (AGP or PCI Express)
Northbridge (memory controller hub)
Memory Slots
Memory bus
Internal Bus
PCI Bus
Onboard graphics controller
Southbridge (I/O controller hub)
IDE
SATA
USB
Ethernet
Audio Codec
CMOS Memory
Cables and ports leading off-board
PCI Slots
LPC Bus
Super I/O
Serial Port
Parallel Port
Floppy Disk
Keyboard
Mouse
Flash ROM (BIOS)

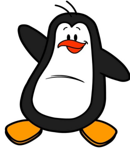
**Remember: H/W is designed first**





# Computer Hardware (cont...)

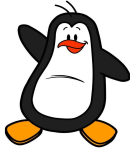
- **Chipset.** North bridge (Memory controller hub) and South bridge (I/O controller hub) both combines to make a chipset. MMU is usually in North bridge, whose main job is logical address to physical address translation. *Firmware* of chipset manages the data flow between processor, memory and peripherals. Chipsets are designed to work with specific family of microprocessors.
- **BIOS.** The firmware of BIOS is stored on a nonvolatile ROM chip on the motherboard. Its main purpose is to initialize and test system hardware components during system boot.
- **MBR.** This is the 1$^{st}$ stage boot loader located at 0 sector of disk. (446B boot loader, 64B partition table, 2B boot signature)
- **Boot Loader.** 2$^{nd}$ stage boot loaders (GRUB, NTLDR) are programs that loads the kernel and transfer execution to it. The OS kernel then initializes itself and may load extra device drivers.





# History of Microprocessors

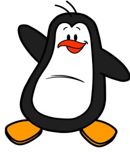
- History of microprocessors goes back to 1971 when Intel introduced its first 4-bit microprocessor Intel-4004, consisting of 2300 transistors, with a support of up to 740 KHz clock, and 4 KB of main memory.
- It was followed by 8-bit Intel-8008 and Intel-8080.
- Later Intel introduced its famous 16 bit Intel-8086 processor with a support of up to 10MHz clock, and 1MB of main memory. It has 16-bit internal registers, 16-bit data bus, and 20-bit address bus (1MB of physical memory)
- In 1982, Intel introduced Intel-80186 (i186) and Intel-80286 (i286), which included a DMA controller and an interrupt controller.





# History of Microprocessors (cont...)

- The third generation of x86 microprocessors, Intel 80386 (i386) was a 32-bit microprocessor with a clock rate support up to 32 MHz and an addressable memory of 4GB. Although in this mode the CPU still used memory segment architecture similar to the one present in earlier x86 microprocessors, the size of memory segments was increased to 4 GB. It became possible to switch from *protected mode* back to *real-mode* without simulating processor reset.
- In 1989, Intel introduced the 80486 and then the Pentium series, which went up to 64 bit with a maximum clock rate support up to 3.8 GHz.
- This rat race of increasing the performance of microprocessors by increasing its clock speed suffered from the limitation of more consumption of power and more and more dissipation of heat.
- Later thinking that two heads are better than one, designers shifted their concentration on multi-core technologies.

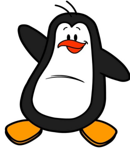




# History of Microprocessors (cont...)

- **Multi-core** means multiple processing cores on the same chip/die. These cores can be homogeneous or heterogeneous. The challenges of multi-cores were memory coherence, cache coherence, communication between cores. Last but not the least the programmers need to learn how to write parallel programs that can execute on multiple cores.
- **Hyper Threading.** For each physical processor core, the OS addresses two logical cores and shares the work load between them when possible. Each logical core can execute a specified thread.

| Intel Processor | Brief Description |
|---|---|
| Core Duo | 2 cores with 2MB L2 cache |
| Core2 Duo | 2 cores with 6MB L2 cache |
| Core2 Quad | Two dies each of which is Core2 Duo |
| Core i3 | 2 cores (4-threads) with 4 MB L3 cache (H.T) |
| Core i5 | 4 cores (4-threads) with 8 MB L3 cache (no H.T) |
| Core i7 (Sandy Bridge) | 8 cores (16-threads) with 20 MB L3 cache (H.T) |



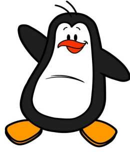


# 32 Bit vs 64 Bit Machines

The processors family started the race of increasing the register size from 4-bits (Intel-4004) to now 64-bits. Benefits of increasing the number of bits is manifold:

- Data can be processed in larger chunks
- More precision of data
- System can point to larger number of locations in main memory, e.g., 32 bit systems could address 4GB of memory while 64 bit machines can address 4 billion times to 4G locations

Note: If you have a 64-bit hardware, it should run a 64-bit Operating System, which should further run 64-bit applications to get the maximum benefit out of it. There can be 8 combinations of these three, out of which only 4 are valid. However, using h/w virtualization you can have a 32 bit CPU running 64 bit OS and a 64 bit application.





# Microprocessor

**Attributes:**
- Size of Registers
- Number of Registers
- RISC/CISC
- Addressing modes
- Endieness
- Pipeline stages

Control Unit

Program Counter

Memory Address Register

Address Bus

Instruction Register

Memory Data Register

Data Bus

Instruction Decoder

Control Bus

General Purpose Registers

Arithmetic and Logic Unit (ALU)

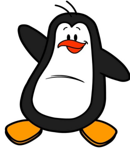
**GPRs**
EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI

**Segment Regs**
CS, DS, SS, ES, FS, GS

EFLAGS, EIP

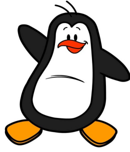
LDT, GDT, CR0

**Every processor has its own Machine Language and its own Assembly Language having one-to-one correspondence**





# Learning Environment





# Learning Environment

**Option-I:** Use a desktop/laptop computer of your own running either a

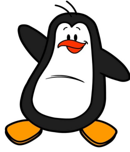
- real UNIX operating system, (may be dual boot)
- install bash shell (WSL) on your Windows 10 PC
- guest UNIX operating system using some virtualization software (e.g., Sun Virtual Box, VM ware, Xen, … )

**Option-II:** You may like to remotely login using `ssh`, `telnet`, `putty`, or some other remote login facility on PUCIT LAN

`ssh arif@172.16.0.21` (Linux OpenSuse)
`ssh arifbutt@172.16.0.103` (PC BSD)

You can also login using WAN on following machines as well (if permitted)

`ssh arifbutt@202.147.169.197` (Solaris 11.0)
`ssh arifbutt@202.147.169.196` (PC BSD)





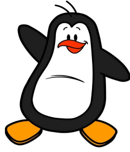
# Linux Distributions

A Linux Distribution is a compilation of

- Linux Kernel (Maintained by the community headed by Linus)
- Startup Scripts
- Configuration files
- Critical support softwares

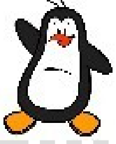
Some famous Linux distributions are:

- Ubuntu
- Kali Linux
- RedHat
- Fedora
- CentOS
- OpenSuse
- Linux Mint
- Turbo Linux





# Accessing OS Services



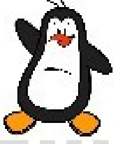
# Operating System Services

Some important tasks a kernel performs are:

- File Management
- Process management
- Memory management
- Information management
- Signal handling
- Synchronization
- Communication (IPC)
- Device management

Two methods by which a program can make requests for services from the Kernel:

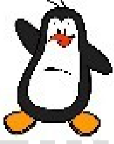
- By making a system call (entry point built directly into the kernel)
- By calling a library routine that makes use of this system call





# Library Functions

1. Libraries are pre-compiled set of functions, ready to be used by programs. For example on Kali Linux and Kernel ver. 4.6.0:
   (a) Standard C Library: `/usr/lib/x86_64-linux-gnu/libc.so`
   (b) Standard Math Library: `/usr/lib/x86_64-linux-gnu/libm.so`
2. Some library functions don't make use of system calls (eg. String manipulation functions, **`strcmp()`**, **`strcat()`**, **`strstr()`**, **`strtok()`**)
3. Some library functions are layered on the top of the system calls (e.g., **`fopen()`** library function uses **`open()`** system call to actually open a file)
4. Often library functions are designed to give more caller friendly interface than the underlying system calls:
   - **`printf()`** library function provides output formatting and data buffering, while **`write()`** system call just outputs a block of bytes
   - **`malloc()`** and **`free()`** are easy to allocate and free memory than the **`brk()`** system call



# System Calls

A system call is the *controlled* entry point into the kernel code, allowing a process to request the kernel to perform a privileged operation. Before going into the details of how a system call works, following points need to be understood:

- A system call changes the processor state from user mode to kernel mode, so that the CPU can access protected kernel memory
- The set of system calls is fixed. Each system call is identified by a unique number. (**`/usr/include/x86_64-linux-gnu/asm/unistd_64.h`**)
- Each system call may have a set of arguments that specify information to be transferred from user space to kernel space and vice versa

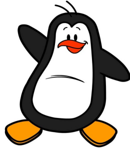
One must go through the man pages for better understanding:

**`man 2 intro`** Introduction to Section 2 of man pages

**`man syscalls`** List of system calls (wrappers)

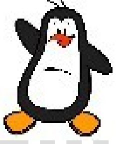
**`man syscall`** Used to invoke a syscall having no wrapper with its ID

**`man _syscall`** Macro used to make a system call (deprecated)





# System Call Invocation



# System Call Invocation

Application Program

```
---
---
read(fd,buff,count);
---
---
```

glibc wrapper function

```
read(...)
{
---
syscall(SYS_read,fd,buff,count);
---
return;
}
```

User Mode

Kernel Mode

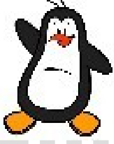
System call Service routine

```
SYS_read()
{
    ---
    ---
    return error
}
```

Trap Handler

```
system_call:
---
---
---
```

1 2 3 4 5 6



# System Calls (cont...)

1. Application program:
   - (a) Makes a system call by invoking a wrapper function in the C library.
   - (b) Passes the arguments by pushing them on the stack in reverse order
2. Wrapper Function:
   - (a) Copies the system call number to a specific CPU register (rax)
   - (b) Copies arguments from stack to rdi, rsi, rdx, r10, r8, r9
   - (c) Execute trap machine instr, that causes the processor to switch from user mode to kernel mode
3. Trap Handler:
   - (a) Save register values (parameters and syscall#) on to process's kernel stack
   - (b) Index the sys_call_table with syscall# to find address of appropriate system call service routine
4. System call routine:
   - (a) Performs specific task and return result/status (in rax) to trap handler
5. Trap Handler:
   - (a) Execute trap machine instr to switch from kernel to user mode and return to wrapper
6. Wrapper Function:
   - (a) In case of error, sets the global variable `errno`
   - (b) Returns to caller, providing an integer return value indicating success/failure of syscall
7. Application program continues

**The registers mentioned are for x86_64 architecture, for i386 read `$ man 2 syscall` for details**

Source code file(s)

`gcc -E hello.c 1> hello.i`

**Preprocessor (cpp)**
- Interpret preprocessor directives
- Include header files
- Expand macros
- Remove comments

Preprocessed code file(s)

`gcc -S hello.i`

**Compiler (cc)**
- Checks for syntax errors
- Converts the src to assembly of underlying processor

Assembly code file(s)

`gcc -c hello.s`

**Assembler (as)**
- Generates relocatable object files to be used by linker
- Contains symbol table

Object code file(s)

Library `libc`

`gcc hello.o -o myexe`

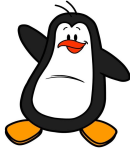
**Linker (ld)**
- Static vs Dynamic linking
- Contains code and data for all functions defined in src files
- Contains global symbol table

Executable file (myexe)

Stored in secondary storage as an executable image in a specific format

**Loader**

Process Address Space in main memory





# System Calls
# Proof of Concept

# Call a system call using its wrapper

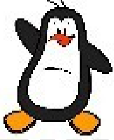
~/spvl/02/systemcalls/usingwrapper.c

```
#include <stdio.h>
#include <unistd.h>
int main(){
  char str[]={"Welcome to System Programming...\n"};
   int rv = write(1, str, sizeof str);
   if(rv == -1){
      perror("Write failed");
      printf("errno contains: %d\n",errno);
   }
   return rv;
}
```

# Call a system call w/o its wrapper

~/spvl/02/systemcalls/usingsyscall/syscall.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>
int main(){
 char str[]="Welcome to the world of system calls"};
    int rv = syscall(1, 1, str, sizeof str);
    return rv;
}
```





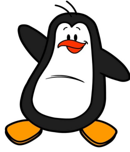
# Call a system call from Assembly Code

~/spvl/02/systemcalls/fromassy/syscall.nasm

```
SECTION .text
global _start
_start:
   mov rax,1   ;syscall # of write()
   mov rdi,1   ;1st arg is file descriptor of stdout
   mov rsi,msg ;2nd arg is address of string
   mov rdx,len_msg;3rd arg is length of string
   syscall

   mov rax,60 ;system call number of exit()
   mov rdi,54 ;1st arg to exit()
   syscall
SECTION .data
msg: db "Hello system call using nasm!", 0Ah, 0h
len_msg: equ $ - msg
```

```
$ nasm -f elf64 hello.nasm
$ ld hello.o -o hello
$./hello
Hello system call using nasm!
```





# Things to do!



**If you have problems visit me in counseling hours**