Lecture # 1 – Introduction (Chapters 1 & 2)

- Motivation

  The UNIX operating system consists of several main components:

<table>
<tr><td colspan="5">Hardware</td></tr>
<tr><td colspan="5">Kernel - Input / Output, Memory Management, Process Management, IPC</td></tr>
<tr><td colspan="5">System Call Interface</td></tr>
<tr><td>Application Programs</td><td>Bourne</td><td>C - shell</td><td>Korn shell</td><td>Perl</td></tr>
</table>

  1. Hardware

  2. Kernel

     The kernel is responsible for interacting with the hardware, and providing central services like I/O, Memory Management, Process Management, and Interprocess Communication. Application programs obtain access to the kernel through the system call interface.

  3. System Call Interface

     The system call interface is a collection of function calls, which provide wrappers for actual system calls. We will spend much of this term examining calls in the system call interface.

  4. Application Layer

     This is where application programs are written, and the different shells reside (i.e. a shell is nothing more than an application program that provides you an interface into UNIX).

- Files and Directories

  Review the directory structure of UNIX.
  Review "/", ".", and ".."
  Review absolute versus relative paths
  Review file and directory permissions and their affects.
  Review file and directory manipulation commands.

Example: C program to emulate basic "ls" command

```
#include <stdlib.h>
#include <sys/types.h>
#include <stdio.h>
#include <dirent.h>

void main(int argc, char **argv)
{
        DIR     *dp;
        struct dirent     *dirp;

        if (argc != 2)
        {
                fprintf(stderr, "usage: %s <dir>\n", argv[0]);
                exit(1);
        }

        if ((dp = opendir(argv[1])) == NULL)
        {
                fprintf(stderr, "Cannot open directory %s\n", argv[1]);
                exit(2);
        }

        while ((dirp = readdir(dp)) != NULL)
                printf("%s\n", dirp->d_name);

        closedir(dp);
        return(0);
}
```

- I/O

File descriptors are small nonnegative integers that the kernel uses to identify the files being accessed by a particular process.

All shells open 3 file descriptors by default- stdin (0), stdout (1), and stderr (2).

Unbuffered I/O - open, read, write, lseek, and close.
Standard I/O - printf, fprintf, fgets, etc.

Example: (copy stdin to stdout using buffered I/O)

```
#include "ourhdr.h"

#define BUFFSIZE        8192

int main(void)
{
        int             n;
        char    buf[BUFFSIZE];

        while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
                if (write(STDOUT_FILENO, buf, n) != n)
                        err_sys("write error");

        if (n < 0)
                err_sys("read error");

        exit(0);
}
```

Example: (copy stdin to stdout using stdio)

```
#include        "ourhdr.h"

int
main(void)
{
        int             c;

        while ( (c = getc(stdin)) != EOF)
                if (putc(c, stdout) == EOF)
                        err_sys("output error");

        if (ferror(stdin))
                err_sys("input error");

        exit(0);
}
```

- Programs and Processes

  Program - an executable file.
  Process - an executing instance of a program along with its associated environment.

  Every UNIX process is guarenteed to have a unique numeric identifier called the process identifier (PID). The PID is always a nonnegative integer.

  Example: print my PID

```
int main(void)
{
        printf("Hello world for Pid = %d\n", getpid());
        exit(0);
}
```

- Process Control

  We will spend a great deal of time working with creating and destroying processes during this course.

  Example: (very basic shell)

```
#include      <sys/types.h>
#include      <sys/wait.h>
#include      "ourhdr.h"

int main(void)
{
      char    buf[MAXLINE];
      pid_t   pid;
      int           status;

      printf("%% ");  /* print prompt (printf requires %% to print %) */

      while (fgets(buf, MAXLINE, stdin) != NULL) {
            buf[strlen(buf) - 1] = 0;       /* replace newline with null */

            if ( (pid = fork()) < 0)
                  err_sys("fork error");

            else if (pid == 0) {            /* child */
                  execlp(buf, buf, (char *) 0);
                  err_ret("couldn't execute: %s", buf);
                  exit(127);
            }
```

```
        /* parent */
        if ( (pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

- Error Handling

  When most system calls fail, they return a negative integer, and set the value of errno to include more information about the error.

  Both strerror and perror can be used to obtain error messages. Strerror allows you to specify a error number, and gives you the associated message while perror prints the message for the current value of errno.

```
extern  int     errno;

char    *strerror(int errnum);
void    perror(const char *msg);
```

  Example: (using strerror and perror)

```
#include    <errno.h>
#include    "ourhdr.h"

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));

    errno = ENOENT;
    perror(argv[0]);

    exit(0);
}
```

- User Identification

  Every user in UNIX is assigned a unique user identification number (UID).
  Every user belongs to at least one UNIX group identified by the GID.

  The root user always has a UID of 0.

Example:

```
int main(void)
{
        printf("uid = %d, gid = %d\n", getuid(), getgid());
        exit(0);
}
```

- Signals

  Signals are the most basic form of interprocess communication. They are a technique to notify a process that some condition has occurred. We will examine signals in much more detail later in the course.

- UNIX Time Values

  Two major time values maintained by UNIX:

  1. Calendar time

     This value is the number of seconds since the Epoch (00:00:00 Jan 1, 1970 UTC).
     The primitive system data type time_t holds these time values.

  2. Process time

     Also called CPU time, measures CPU resources used by a process.
     Process time is measured in clock ticks (i.e. 50, 60, or 100 ticks per second).
     CLK_TCK stores in the number of clock ticks per second.

  UNIX maintains at least 3 times for each process: clock time, user time, and system time.

  Clock time is also called wall clock time.
  User time is the CPU time attributed to executing user instructions.
  System time is the time the kernel spent executing instructions on the processes behalf.

- System calls and Library functions

  All variants of UNIX provide a well-defined, limited number of entry points directly into the kernel called **system calls**.

  In UNIX, each system call has a function of the same name in the standard C library.

  The user process calls this function, and the function invokes the correct kernel service using whatever technique is appropriate for that variant of UNIX.