UNIX System Programming

Roee Leon

NOVEMBER 02, 2016

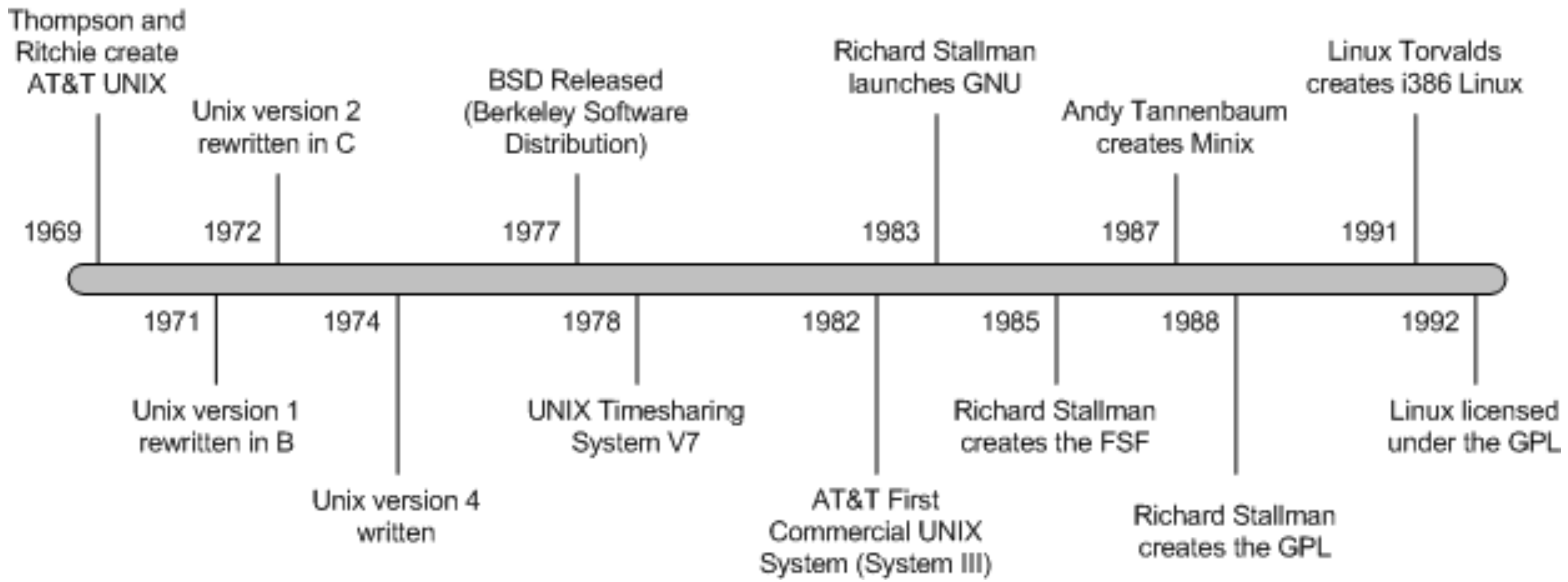
Administration

- Grading Policy:
  - 80% Home Assignments
  - 20% Projects Defense
- Email: roee.leonn@gmail.com
- Requirements:
  - Physical/Virtual UNIX Based System
    - Preferably Linux

Course Goal

- Getting familiar with development tools in GNU/Linux
  - Working understanding of UNIX
  - Basic shell commands
  - GNU development tools
    - Compiler (GCC)
    - Debugger (GDB)
    - Make
    - And more…
  - OS provided interface (System-Calls) for application development

Timeline of Unix, Linux, GNU


Thompson and Ritchie create AT&T UNIX
1969
Unix version 2 rewritten in C
1972
BSD Released (Berkeley Software Distribution)
1977
Richard Stallman launches GNU
1983
Andy Tannenbaum creates Minix
1987
Linux Torvalds creates i386 Linux
1991
1971
Unix version 1 rewritten in B
1974
Unix version 4 written
1978
UNIX Timesharing System V7
1982
AT&T First Commercial UNIX System (System III)
1985
Richard Stallman creates the FSF
1988
Richard Stallman creates the GPL
1992
Linux licensed under the GPL

GNU History

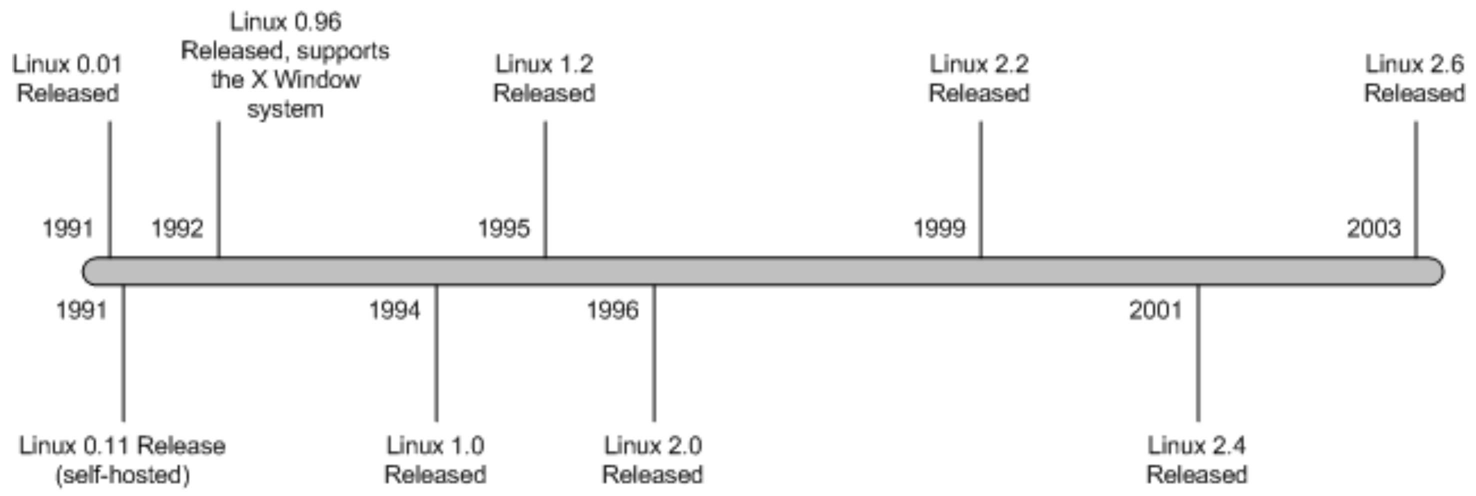
- GNU stands for GNU is Not Unix
- Created by Richard Stallman
- Composed of an operating system and free software
  - Utilities such as *ls, cat, etc.*
  - freedom to run the program, for any purpose.
  - freedom to study how the program works and adapt it to your needs.
  - freedom to redistribute copies so you can help others.
  - freedom to improve the program and release your improvements to the public, so that everyone benefits

GPL

- GPL Stands for *General Public License*
  - Intended to guarantee a programmer the ability to run, study, share and modify the software
  - Written by Richard Stallman
  - GPL is a *copyleft* license
    - A modified work can only be distributed under the same license terms
      - i.e. if you created a modified version of a GPL licensed program, this program also has to be GPL licensed

Linux History

- Developed by Linus Torvalds
  - Was a computer science student in the University of Helsinki back in 1991
  - Initially supported only i386-based computers
- In 1996, the Linux kernel becomes the GNU kernel
  - A component of GNU
- Has evolved dramatically ever-since



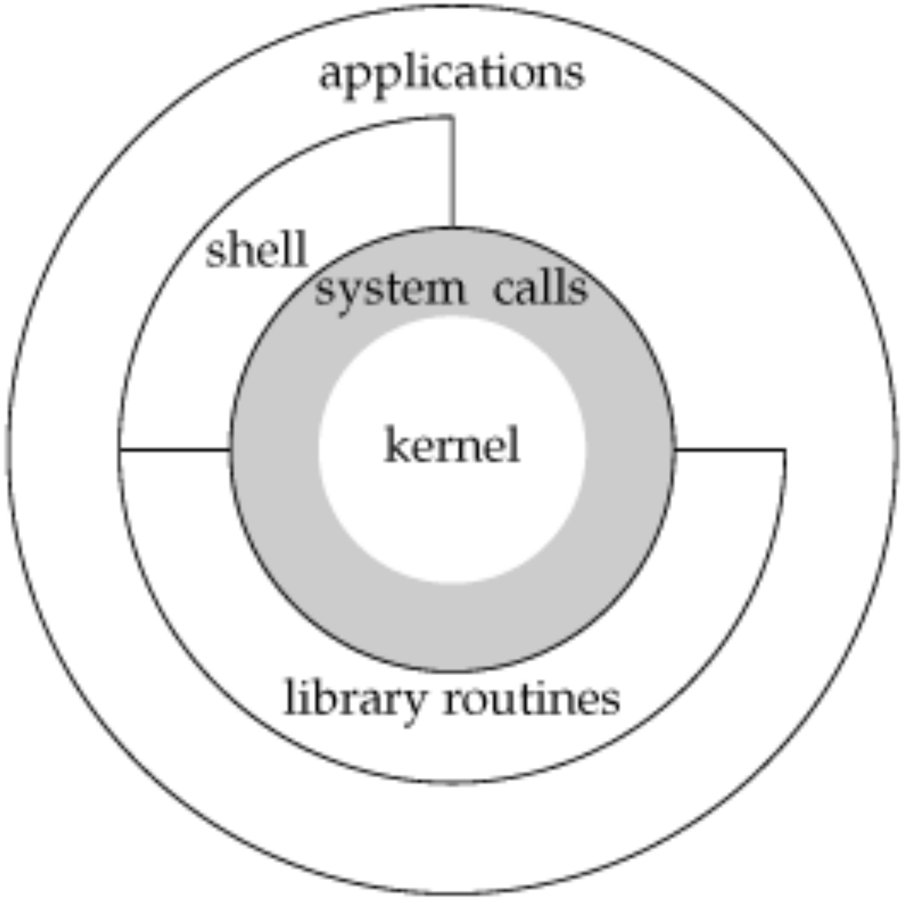
UNIX System Overview - Introduction

- Operating systems
  - Provides services for running programs
    - Opening a file
    - Reading/Writing a file
    - Memory allocation
    - Etc.
  - The UNIX operating system
    - The main focus of the course
    - Various versions and flavors (‘Unix-like/based’ systems)
    - We will explore the UNIX System from a programmer’s perspective

UNIX System Overview - Architecture

- The operating system controls the hardware resources of the computer
  - This layer of software is called the **kernel**
    - Why?
      - Relatively small
      - Resides at the core of the environment
- Interaction with the kernel is done via a layer of software
  - This layer of software is called the **system calls**
  - **Libraries** of common functions are built on top of the system call interface
- Applications can use both the system call interface and the library interface
- A special application that is built into the UNIX operating system is the **shell**
  - May be used to run other applications

UNIX System Overview - Architecture



UNIX System Overview – Logging in

- Logging in a UNIX system is done using a login name and a password
    - The system looks up for the login name & password in its 'password file'
        - Usually /etc/passwd
    - Each entry is composed of the following fields:
        - Login Name
        - Encrypted Password
        - Numeric User ID
        - Numeric Group ID
        - Common Field
        - Home Directory
        - Shell Program
        - Example:
            - **Roee:x:1:2:Roee Leon:/home/roee:/bin/bash**
        - Most systems have moved the encrypted password field into a different file.

        We will see later in the course why and what file it is.

UNIX System Overview – Shells

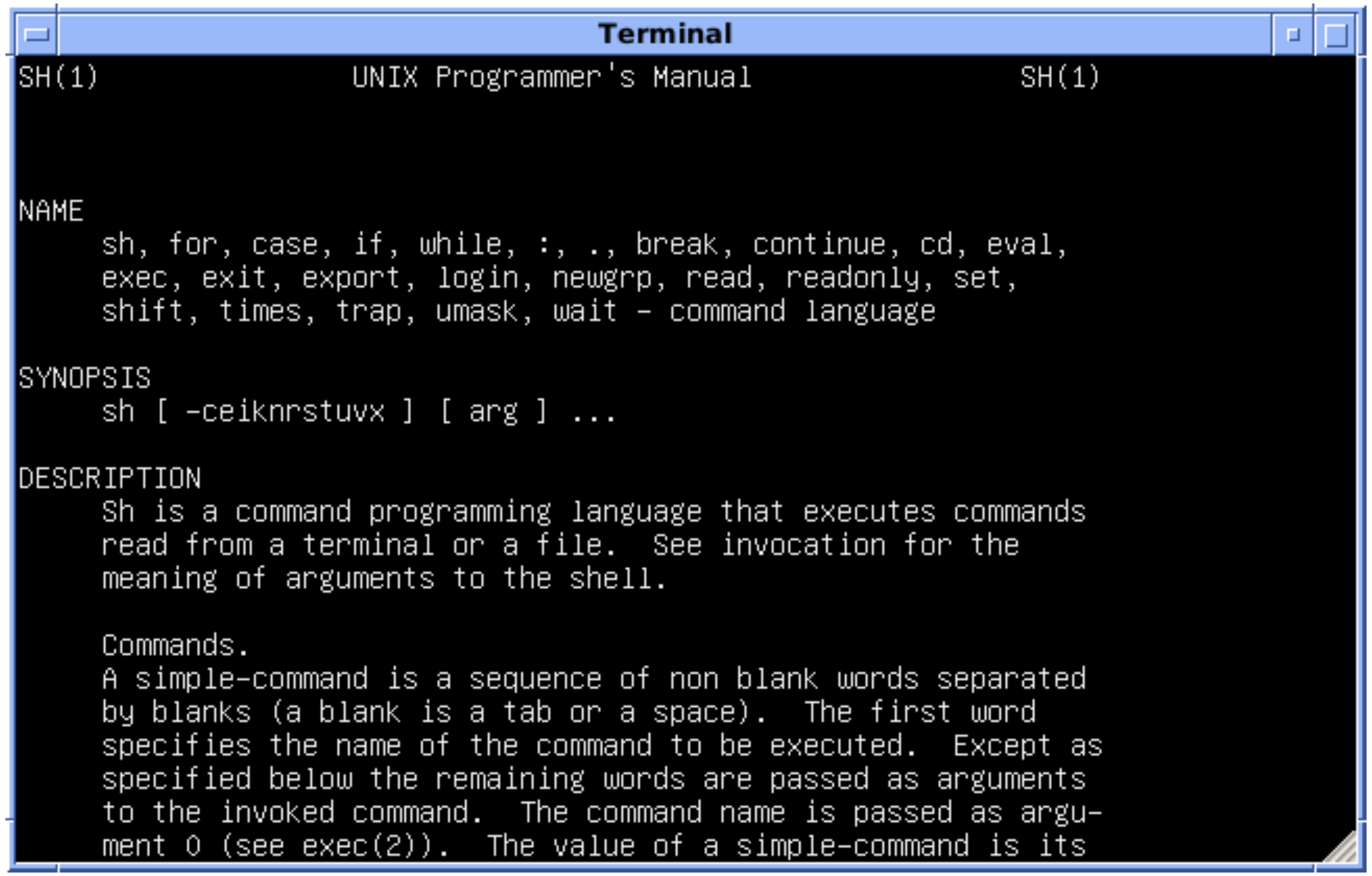
- Once logged in, commands may be typed into the **shell** program
- The shell is a command line interpreter that reads and executes user input
  - Provides some sort of abstraction layer between the Shell (User-space) and the Nut (Kernel)
- The user input to a shell is usually done from an interactive shell (Using the keyboard) or may be also done via a file (Called 'Shell Script')
- Commonly used shells on UNIX systems:
  - Bourne shell
  - Bourne-again shell
  - C Shell
  - Korn Shell

  And more ..

- The system determines which shell to use according to the last field in the password file (/etc/passwd)

UNIX System Overview – Shells – The Bourne Shell

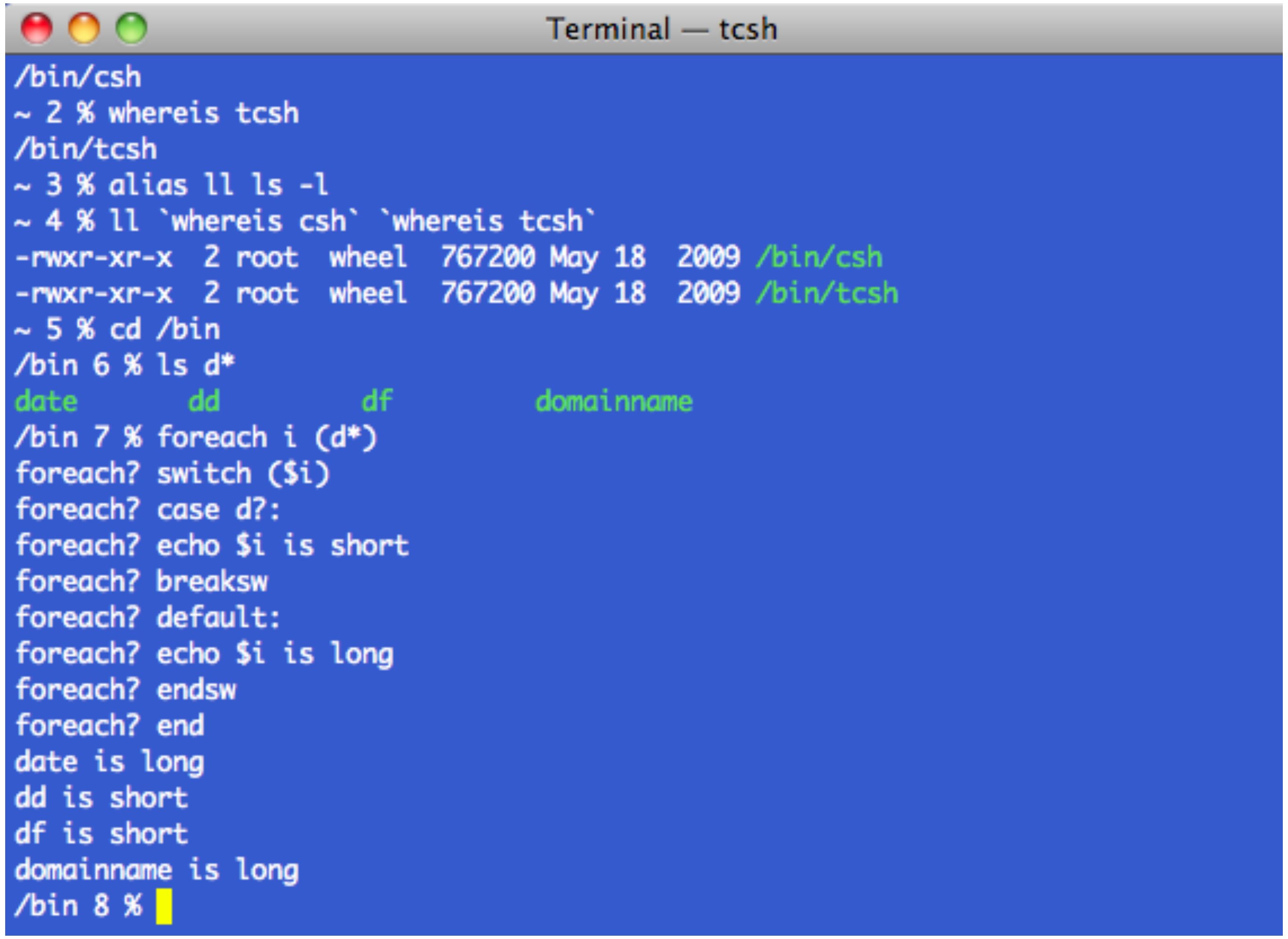
- Was developed by Stephen Bourne at Bell Labs
- Was the default shell for Unix Version 7
- Provided in almost every UNIX system



```
SH(1)                   UNIX Programmer's Manual                   SH(1)



NAME
     sh, for, case, if, while, :, ., break, continue, cd, eval,
     exec, exit, export, login, newgrp, read, readonly, set,
     shift, times, trap, umask, wait - command language

SYNOPSIS
     sh [ -ceiknrstuvx ] [ arg ] ...

DESCRIPTION
     Sh is a command programming language that executes commands
     read from a terminal or a file.  See invocation for the
     meaning of arguments to the shell.

     Commands.
     A simple-command is a sequence of non blank words separated
     by blanks (a blank is a tab or a space).  The first word
     specifies the name of the command to be executed.  Except as
     specified below the remaining words are passed as arguments
     to the invoked command.  The command name is passed as argu-
     ment 0 (see exec(2)).  The value of a simple-command is its
```

UNIX System Overview – Shells – The C Shell

- Was developed by Bill Joy at Berkeley
- Control flow looks more like the C programming language
- Provided additional features that weren't provided by the Bourne Shell (Were later added in the Bourne-again Shell)

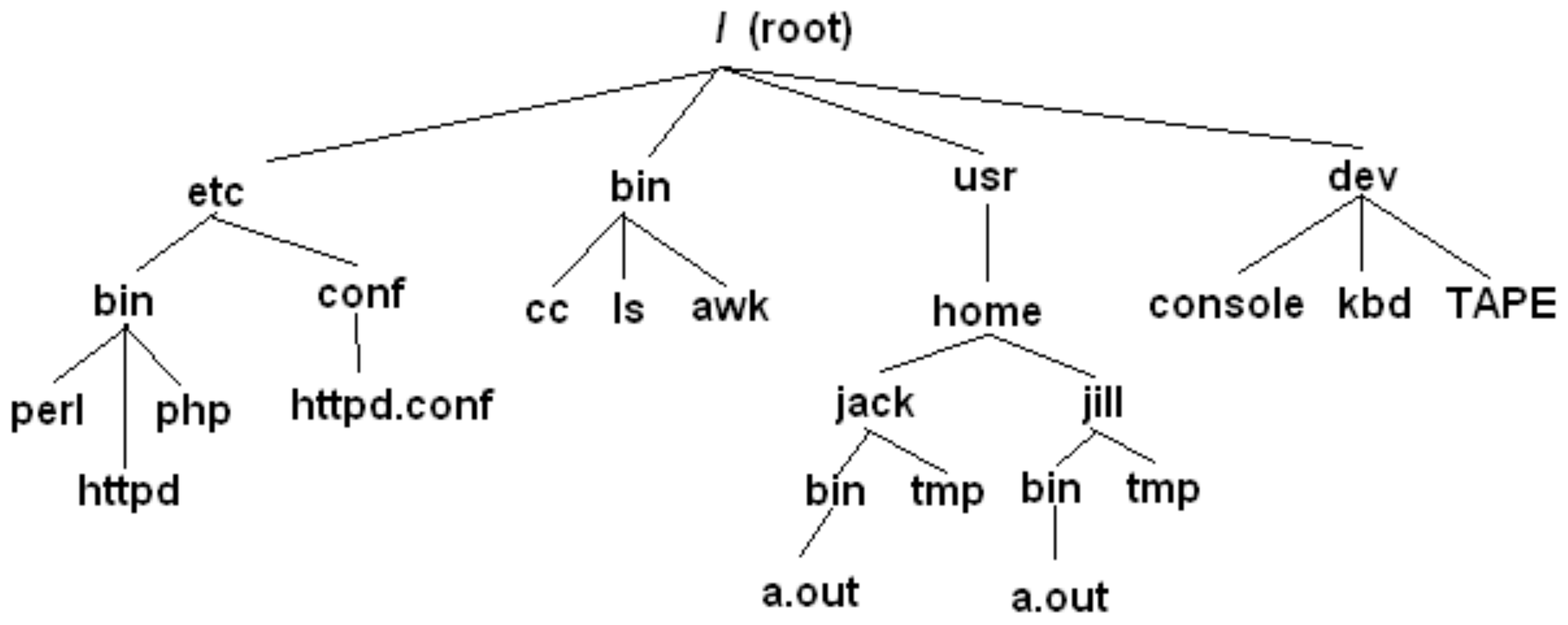


```
/bin/csh
~ 2 % whereis tcsh
/bin/tcsh
~ 3 % alias ll ls -l
~ 4 % ll `whereis csh` `whereis tcsh`
-rwxr-xr-x  2 root  wheel  767200 May 18  2009 /bin/csh
-rwxr-xr-x  2 root  wheel  767200 May 18  2009 /bin/tcsh
~ 5 % cd /bin
/bin 6 % ls d*
date        dd          df          domainname
/bin 7 % foreach i (d*)
foreach? switch ($i)
foreach? case d?:
foreach? echo $i is short
foreach? breaksw
foreach? default:
foreach? echo $i is long
foreach? endsw
foreach? end
date is long
dd is short
df is short
domainname is long
/bin 8 %
```

UNIX System Overview – Files and Directories – File System

- The UNIX file system is a hierarchical arrangement directories and files
  - Every directory/file path starts in the directory called **root**, whose name is the '/' character
  - A directory (Same as Folder) is a file that contains other directory/file entries
    - Each file/directory entry is composed of a filename and a structure of information describing its attributes
    - The attributes of a file may be:
      - Type – Regular / Directory
      - Owner
      - Permissions
      - Last modified
    - File information may be retrieved using the *stat* and *fstat* library functions

UNIX System Overview – Files and Directories – File System



UNIX System Overview – Files and Directories – File System

- Viewing file attributes example:

```
#include <stdio.h>
#include <sys/stat.h>

int main (int argc, const char* argv[])
{
    struct stat file_stats;

    if (argc != 2)
    {
        fprintf(stderr, "Usage: ./FileAttributes 'file-path'\n");
        return 1;
    }
    if (stat(argv[1], &file_stats) < 0)
    {
        perror("stat");
        return 2;
    }
    printf("%ld\n", file_stats.st_dev);
    printf("%ld\n", file_stats.st_ino);
     //…
    printf("%ld\n", file_stats.st_ctime);
    return 0;
}
```

UNIX System Overview – Files and Directories - Filenames

- The names within a directory are called filenames
- A filename may contain **any** character but the '/' and the null characters
  - The '/' is used to separate filenames that from a path
  - The null character is used to terminate a pathname
- Every time a new directory is created, two filenames are automatically created: . (Dot) and .. (Dot Dot)
  - . (Dot) refers to the current directory
  - .. (Dot Dot) refers to the parent directory
  - In / both Dot and Dot Dot refer to the current directory

UNIX System Overview – Files and Directories – Pathnames

- Generally one or more filenames separated by the ‘/’ character
  - May start with the ‘/’ character – **absolute** path
  - May not start with the ‘/’ character – **relative** path
- Relative pathnames refer to files that are relative to the current directory
- The only absolute pathname that is not composed of filenames is the root (‘/’).

UNIX System Overview – Files and Directories – Pathnames

- A program that lists the name of all files within a given directory can be easily written in the C programming language
  - Such program exists, named *ls*, in most UNIX based systems and its job is to list the files in a given directory or the files in the current directory if no directory is given
  - For further understanding of how to use *ls* (or many more built-in applications) one may use the Unix Manual Page ('man' command – e.g. *man ls*)
- In order to implement our own basic *ls* application, we will inspect two library functions:
  - opendir
  - readdir

UNIX System Overview – Files and Directories – Pathnames

man opendir

```
DIRECTORY(3)          BSD Library Functions Manual          DIRECTORY(3)
NAME
opendir, fdopendir, readdir, readdir_r, telldir, seekdir, rewinddir,
closedir, dirfd -- directory operations
LIBRARY
          Standard C Library (libc, -lc)
SYNOPSIS
          #include <dirent.h>
DIR *
opendir(const char *filename);
DIR *
fdopendir(int fd);

// ...
```

UNIX System Overview – Files and Directories – Pathnames

man readdir

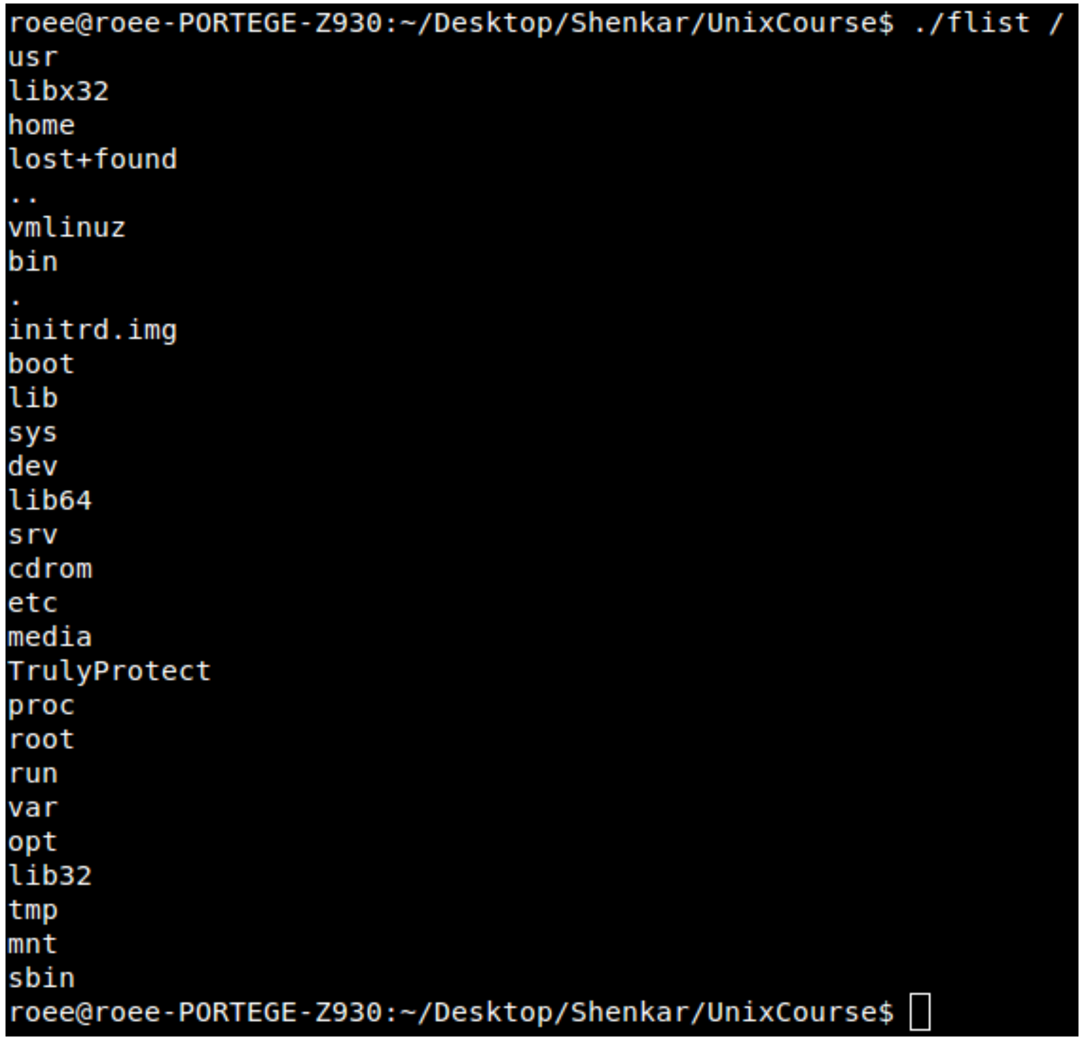
```
DIRECTORY(3)          BSD Library Functions Manual          DIRECTORY(3)
NAME
opendir, fdopendir, readdir, readdir_r, telldir, seekdir, rewinddir,
closedir, dirfd -- directory operations
LIBRARY
          Standard C Library (libc, -lc)
SYNOPSIS
          #include <dirent.h>
// ...
struct dirent *
readdir(DIR *dirp);
int
readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
// ...
```

UNIX System Overview – Files and Directories – Pathnames

- Listing files in a directory example:

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


int main (int argc, const char* argv[])
{
    DIR* directory;
    struct dirent* ent;

    if (argc != 2)
    {
        fprintf(stderr,"Usage: ./flist 'Path'\n");
        return 1;
    }

    if ((directory = opendir(argv[1])) == NULL)
    {
        fprintf(stderr, "Could not open file: %s\n", argv[1]);
        return 2;
    }
    while ((ent = readdir(directory)) != NULL)
        printf("%s\n", ent->d_name);
    return 0;
}
```

UNIX System Overview – Files and Directories – Pathnames



```
roee@roee-PORTEGE-Z930:~/Desktop/Shenkar/UnixCourse$ ./flist /
usr
libx32
home
lost+found
..
vmlinuz
bin
.
initrd.img
boot
lib
sys
dev
lib64
srv
cdrom
etc
media
TrulyProtect
proc
root
run
var
opt
lib32
tmp
mnt
sbin
roee@roee-PORTEGE-Z930:~/Desktop/Shenkar/UnixCourse$
```

UNIX System Overview – Files and Directories – Working/Home dir

- Every process has a 'Working Directory'
  - Sometimes referred to as the 'Current Working Directory'
- A process's Working Directory specifies the directory from which all relative pathnames are interpreted
  - A process's Working Directory may be changed using the 'chdir' library function
- For example, the relative pathname: Shenkar/Unix/Exercise refers to the file/directory Exercise which resides in the directory Unix which itself resides in the directory Shenkar. The directory Shenkar **must** reside within the 'Working Directory'.
  - Note that we can't tell whether Exercise is a regular file or a directory
- Upon log-in, the Working Directory is set to be our home directory as stated in the passwords file.

UNIX System Overview – Input / Output – File Descriptors

- Non-negative integers used by the kernel to identify files used by a process
  - almost any input/output resource can be accessed through a file in UNIX)
  - Sort of a file handler
- Whenever a new file is opened, the kernel returns a file descriptor that can be used to read/write from/to the file.
  - The kernel maintains a table called ‘File-Descriptor Table’ **for each process** on which each item represents an opened file

UNIX System Overview – Input / Output – STDIN/STDOUT/STDERR

- All shells open three file descriptors whenever a new program is executed:
  - Standard Input
  - Standard Output
  - Standard Error
- By default, these files are connected to the terminal
  - Standard Input – The input to the terminal is the input to the new program
  - Standard Output / Standard Error – The output of the new program is displayed on the terminal
- Most shells provide a way to redirect any of these file descriptors to **any** file!
  - Example – no redirection: *ls*
    - Stdout of *ls* is connected to the terminal; thus, the file list will be displayed on it
  - Example – with redirection: *ls > output*
    - Stdout of *ls* is redirected to a file named: output; thus, the file list will be written to this file
    - By default, most shells create the file if it does not exist

UNIX System Overview – Input / Output – Un-buffered I/O

- Reading/Writing one data element at a time
  - If the data is a file on disk – reading/writing character by character
    - Not fully correct – the programmer may control the buffer size
- Large overhead
  - I/O request on each read/write
- Mostly needed whenever the programmer wants to ensure that the data has been fully written before continuing
- In Unix, un-buffered I/O is provided by the library functions:
  - *open*
  - *read*
  - *write*
  - *lseek*
  - *close*

UNIX System Overview – Input / Output – Un-buffered I/O

- Un-buffered I/O Example:
  - What does the following code do?

```
#include <stdio.h>

#define BUF_SIZE 1024

int main (int argc, const char* argv[])
{
        int n;
        char buffer[BUF_SIZE];
        while ( (n = read(STDIN_FILENO, buffer, BUF_SIZE)) > 0)
                if (write(STDOUT_FILENO, buffer, n) != n)
                    return 1;
        if (n < 0)
            return 2;
        return 0;
}
```

UNIX System Overview – Input / Output – Un-buffered I/O

- Un-buffered I/O Example:
  - What does the following code do?
    - STDIN_FILENO and STDOUT_FILENO

      are constants defined in <unistd.h> which specify the file descriptors for the standard input and output (Their values are 0 and 1, respectively)
    - The read function returns the number of bytes read. The returned value can then be used to write the exact amount of bytes read into the standard output (The write output also returns the amount of bytes written. Therefore, we check whether the returned value is exactly *n*)
    - If an error occurs during the read library call, -1 is returned.
      - Most library functions return -1 in case of an error

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

#define BUF_SIZE 1024

int main (int argc, const char* argv[])
{
        int n;
        char buffer[BUF_SIZE];
        while ( (n = read(STDIN_FILENO, buffer, BUF_SIZE)) > 0)
                if (write(STDOUT_FILENO, buffer, n) != n)
                        return 1;
        if (n < 0)
            return 2;
        return 0;
}
```

UNIX System Overview – Input / Output – Un-buffered I/O

- Un-buffered I/O Example:
  - Variations of program execution:
    - ./run
      - Both *stdin* and *stdout* are connected to the terminal; thus, simply writes input from the keyboard into the screen
    - ./run > out
      - *stdin* is connected to the terminal while *stdout* is redirected to a file named “out”; thus, simply writes input from the keyboard into the file
    - ./run < in > out
      - *stdin* is redirected to a file named “in” and *stdout* is redirected to a file named “out”; thus, simply copies the contents of “in” to “out”

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

#define BUF_SIZE 1024

int main (int argc, const char* argv[])
{
        int n;
        char buffer[BUF_SIZE];
        while ( (n = read(STDIN_FILENO, buffer, BUF_SIZE)) > 0)
                if (write(STDOUT_FILENO, buffer, n) != n)
                    return 1;
        if (n < 0)
            return 2;
        return 0;
}
```

UNIX System Overview – Input / Output – Buffered I/O

- Standard I/O library functions that provide a buffer interface to the Un-buffered I/O library functions
  - i.e. manages for us the buffer length – choosing the buffer length
- Simplifies dealing with lines of input
  - Common case in applications
  - The *gets/fgets* library function reads an entire line into a given buffer rather than the *read* library functions that allows reading a pre-specified number of bytes
- An example for common I/O library function is *printf*

UNIX System Overview – Input / Output – Buffered I/O

- Un-buffered I/O Example:
  - Write the program shown in slide 26 using buffered I/O library functions

UNIX System Overview – Input / Output – Buffered I/O

- Un-buffered I/O Example:
  - Write the program shown in slide 26 using buffered I/O library functions
    - The library function *getc* reads one character at a time. The parameter it receives is a pointer to a FILE object (FILE*) that describes STDIN
    - *getc* returns the constant EOF upon entry of the last input character
    - *putc* acts similarly

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

int main (int argc, const char* argv[])
{
        int ch;
        while ((ch = getc(stdin))  != EOF)
                if (putc(c, stdout) == EOF)
                        return 1;
        if (ferror(stdin))
                return 2;
        return 0;
}
```

UNIX System Overview – Programs and Processes - Program

- An executable file residing on disk
  - In a directory
- Upon execution, a program is written into the memory and is executed by the kernel
  - A program, e.g. shell, sends a request (??) to the kernel to execute another program
- The kernel supplies what is called *exec* functions in order to provide such functionality for applications
  - There are seven *exec* functions

UNIX System Overview – Programs and Processes – Processes and ID

- An instance of executing program is called a process
  - Just a concept, think of it as a bunch of data that describes program being executed
  - Some operating systems refer to it as task rather than a process
- The UNIX system guarantees that every process in the system has a unique, non-negative, numeric identifier called the Process ID.
  - This number allows processes to be manipulated. For example:
    - Stopping a process
    - Killing a process
    - Setting process priority

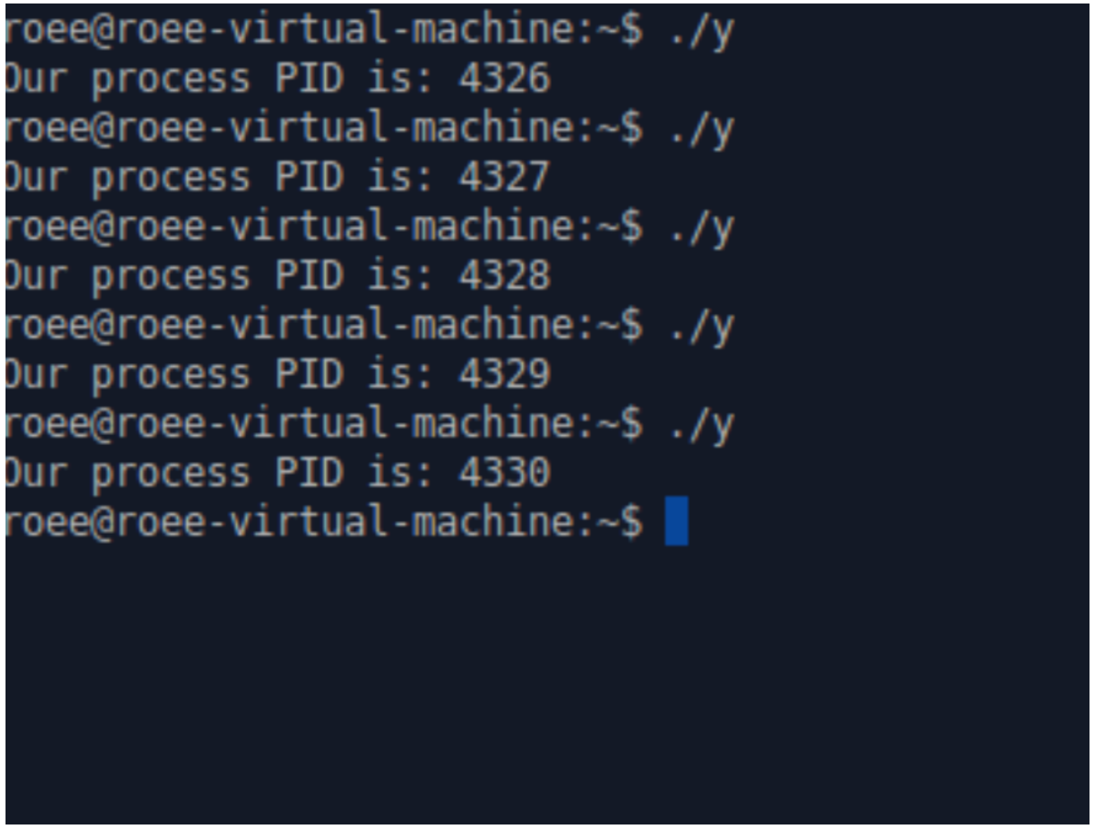
UNIX System Overview – Programs and Processes – Processes and ID

- Printing current process ID example:

```
#include <stdio.h>

int main (int argc, const char* argv[])
{
		long my_pid = getpid();
		printf("Our process PID is: %ld\n", my_pid);
		return 0;
}
```

UNIX System Overview – Programs and Processes – Processes and ID

- Printing current process ID example:



```
roee@roee-virtual-machine:~$ ./y
Our process PID is: 4326
roee@roee-virtual-machine:~$ ./y
Our process PID is: 4327
roee@roee-virtual-machine:~$ ./y
Our process PID is: 4328
roee@roee-virtual-machine:~$ ./y
Our process PID is: 4329
roee@roee-virtual-machine:~$ ./y
Our process PID is: 4330
roee@roee-virtual-machine:~$
```

UNIX System Overview – Programs and Processes – Process Control

- Three primary library functions for process control:
  - *fork*
    - Creates a new process – the newly launched process is the exact copy of the calling process
  - *exec*
    - Described earlier
  - *waitpid*
    - Waits for process termination

UNIX System Overview – Programs and Processes – Processes and ID

- *fork, exec* and *waitpid* example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#define MAX_LINE_SIZE 2048

int main (int argc, const char* argv[])
{
   char line[MAX_LINE_SIZE];
   pid_t pid;
   printf("> ");
   while (fgets(line, MAX_LINE_SIZE, stdin) != NULL)
   {
      line[strlen(line)-1] = 0;
      if ((pid = fork()) < 0)
         perror("fork");
      else if (!pid)
      {
         char* args[] = {line, NULL};
         if (execve(line, args, NULL) < 0)
         {
            perror("execve");
            exit(1);
         }
      }
      else
      {
         int status;
         if ((pid = waitpid(pid, &status, 0) < 0))
            perror("waitpid");
      }
      printf("> ");
   }
   exit(0);
}
```

UNIX System Overview – Programs and Processes – Processes and ID

- *fork, exec* and *waitpid* example:
  - Buffered I/O standard library function fgets is used to read one line at a time
  - The line returned by fgets is end-line terminated while the execve function requires a null terminated string. Therefore, we replace the new-line with the null character
  - The fork library function creates an exact copy of the program
    - The copy is called the Child Process while the creating process is called the Parent Process
  - The fork library function returns zero to the child process and a process ID (The child's) to the parent process
    - Yes, it returns twice!
  - Finally, the waitpid library function (called only by the parent process) waits for the child process to terminate

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#define MAX_LINE_SIZE 2048

int main (int argc, const char* argv[])
{
   char line[MAX_LINE_SIZE];
   pid_t pid;
   printf("> ");
   while (fgets(line, MAX_LINE_SIZE, stdin) != NULL)
   {
      line[strlen(line)-1] = 0;
      if ((pid = fork()) < 0)
         perror("fork");
      else if (!pid)
      {
         char* args[] = {line, NULL};
         if (execve(line, args, NULL) < 0)
         {
            perror("execve");
            exit(1);
         }
      }
      else
      {
         int status;
         if ((pid = waitpid(pid, &status, 0) < 0))
            perror("waitpid");
      }
      printf("> ");
   }
   exit(0);
}
```

UNIX System Overview – Threads and Threads IDs

- A process is usually consists of a singlet thread of execution
  - i.e. only one piece of machine code can be executed at a time
- Sometimes it is necessary for a process to have more than one thread of execution.
  - For example, solving problems on which things need to be done in “Parallel” (or at least that is what we think)
  - Multiple threads within the same process may also utilize the “Real Parallelism” provided by multiprocessors systems
- Other than processes, all threads within a process share the same address space
  - i.e. every single thread is capable of accessing the entire memory of the program
  - Due to that, threads need to synchronize access to shared data among themselves
- Threads are also identified by IDs (Such as processes)
- Thread IDs, other than Process IDs, need not be unique to the system
  - They are only used to be identified by the owning process
  - They are meaningless to any other process

UNIX System Overview – Error Handling

- Basically, UNIX library functions report errors according to the following:
  - A negative value is returned
  - A global integer, referred to as *errno*, is set to a value that represents the error
- For example, the library function *read* returns -1 on error and set the *errno* variable to one of 9 values. For example:
  - EIO – I/O Error
  - EFAULT – provided buffer is outside accessible address space
- However, most functions that return a pointer value (e.g. *opendir*) return a null pointer in case of a failure
- In order to use *errno*, <errno.h> needs to be included as it contains the constants for the errors *errno* is 'familiar' with

UNIX System Overview – Error Handling

- Rules to follow regarding *errno*:
  - *errno* value is never cleared if an error does not occur
    - Therefore, you have to check its value **only** in case of an error
  - *errno* value is never set to zero. Moreover, none of the constants defined in <errno.h> has the value of zero
- Library functions that can be used to examine errno in a more "User-Friendly" way:
  - *void perror (const char* msg);*
    - Outputs an ASCII string message describing the provided error
  - *char* strerror(int errnum);*
    - Given a numeric constant defined in <errno.h>, or just *errno* itself, returns a pointer to an ASCII string describing the error

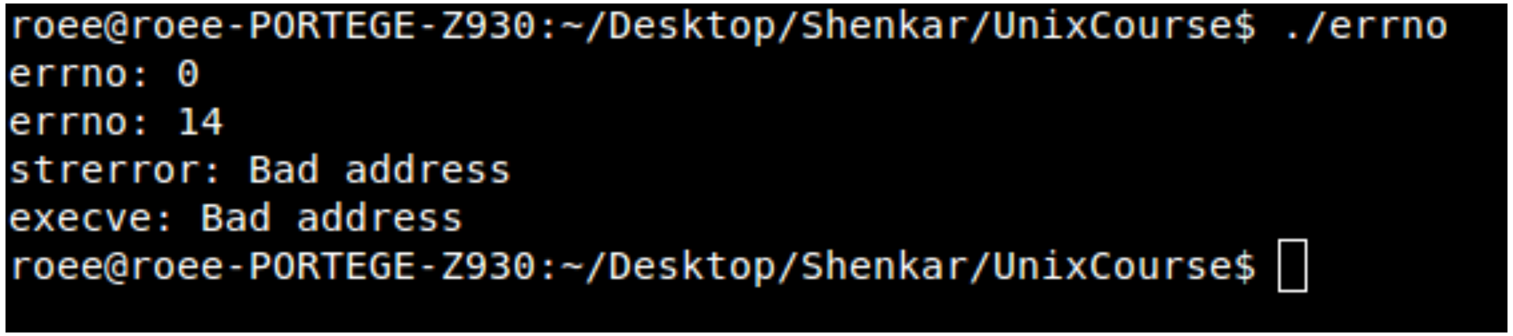
UNIX System Overview – Error Handling

- *errno* usage example:

```
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main(int argh, const char* argv[])
{
		printf(“errno: %d\n”, errno);

		if (execve(NULL, NULL, NULL) < 0)
		{
				printf(“errno: %d\n”, errno);
				printf(“strerror: %s\n”, strerror(errno));
				perror(“execute”);
		}
		return 0;
}
```

UNIX System Overview – Error Handling

- *errno* usage example:



```
roee@roee-PORTEGE-Z930:~/Desktop/Shenkar/UnixCourse$ ./errno
errno: 0
errno: 14
strerror: Bad address
execve: Bad address
roee@roee-PORTEGE-Z930:~/Desktop/Shenkar/UnixCourse$ 
```

UNIX System Overview – Error Recovery

- The errors defined in <errno.h> can be divided into two groups – Fatal and Non-fatal.
  - A program cannot recover from a fatal error
    - Probably the best you can do is printing a message to a log file or to the screen and then terminate
  - A program may recover from a non-fatal error
    - Most non-fatal errors are temporary. For example, the system may be currently low on resources. This situation may not last for long if the system activity is reduced.
    - In that case, waiting for a while and then try again may be the best choice
- The decision on how program errors need to be dealt with is eventually up to the programmer.
  - For example, in case of a connection failure, one programmer may prefer to terminate the program while the other one may prefer to try reestablishing the connection until it succeeds

UNIX System Overview – User Identification – User ID

- Each entry in the password file contains a unique numeric value that identifies us
  - This value is assigned to us by the system administrator during the username entry creation phase
  - Upon login, this number (the User ID) is assigned to us
    - It is done according to our login-name (Which is verified by the password)
    - Cannot be changed!
- The user ID number ‘0’ (Zero) is reserved for a “Special” user called the *root* or the *superuser*
  - This user has super-user privileges – we will see more on that later in the course

UNIX System Overview – User Identification – Group ID

- Each entry in the password file also contains a unique numeric value that identifies our group (the *Group ID)*
  - This value, as the User ID value, is also assigned to us by the system administrator on the username entry creation phase
  - Basically, groups are used to collect related users together
    - For example, users that work on the same project
  - Using groups, it is possible to enforce file permissions in a group resolution
- The file: /etc/group maps groups names into their corresponding group IDs

UNIX System Overview – User Identification – Group ID

- What are the User ID and Group ID used for?
  - They are primarily used for permissions
    - Each file on the disk stores both the owner user ID and the owner group ID
      - Each of these is 16bit unsigned integer (=only four bytes per file)
      - Why doesn’t UNIX store the user **name** and group **name** instead for each file?
        - Why do we use names anyway?
    - When a file is being accessed, the user ID of the user accessing the file is checked against the actual owner ID of the file
      - Based on that, but not only, it is decided whether the accessing user should be granted permissions or not
      - Most file permissions checks are bypassed if the accessing user has a user ID of zero (i.e. *root/superuser*)

UNIX System Overview – Signals

- Technique used to notify a process that something has occurred
  - For example, if the code executed by the process divides by zero, a signal named *SIGFPE* is sent to the process.
  - After receiving the signal, the process has two options:
    - Ignore the signal
    - Default behavior
      - the default behavior (i.e. ignoring the signal) is to terminate the program
    - Handle the signal
      - Providing a callback function that will be called when the signal occurs (Think of it as a "try" and "catch")
        - In this case, we will actually know when the signal occurs
  - The terminal provides two keys called the *interrupt key* (often the delete key or Control-C) and the *quit key* (often Control-backslash) are used to interrupt the current process.
  - Another way send a signal is by calling the *kill* library function
    - Used by one process to terminate another
      - Limited to the owner of the want-to-be-killed process (Unless done by the *root/superuser)*

UNIX System Overview – Signals Example

- *signal* usage example:

```
#include <stdio.h>
#include <sys/wait.h>

static void my_sig_int(int sig_number)
{
		printf("You won't terminate me … \n");
}

int main (int argc, const char* argv[])
{
		if (signal(SIGINT, my_sig_int) == SIG_ERR)
				perror("signal");
		for(;;);
		return 0;
}
```

UNIX System Overview – Signals Example

- *signal* usage example:
  - The *signal* library function registers for us a signal handler that will be called when a SIGINT signal is received
    - The SIGINT signal can be generated by the *Control-C* keyboard combination within the terminal
  - The program simply runs a do-nothing infinite loop and will not respond to received interrupt signals
    - By default, an interrupt signal terminates the program

```
#include <stdio.h>
#include <sys/wait.h>

static void my_sig_int(int sig_number)
{
		printf(“You won’t terminate me … \n”);
}

int main (int argc, const char* argv[])
{
		if (signal(SIGINT, my_sig_int) == SIG_ERR)
			perror(“signal”);
		for(;;);
		return 0;
}
```

UNIX System Overview – Implementing a simple Shell

- Your first programming exercise
  - Has no effect on the final grade
  - You have to submit it!
- A very basic one can be quite simple
  - The one you will implement
  - Around ~10 lines of code (in the C programming language)

UNIX System Overview – Implementing a simple Shell

- Minimal shell algorithm:
  - Perform initializations
  - While !EOF (End of File)
  - do
    - cmd <- next command from *stdin*
    - Interpret cmd
    - Execute cmd

UNIX System Overview – Implementing a simple Shell

- Required functionality
  - ls %ARG0
    - ARG0 represents an absolute path (You may also support relative path)
    - Lists the directories/files in ARG0
    - If no arguments are provided, lists the files in the current working directory
      - Hint: *man* getcwd
  - cd %ARG0
    - ARG0 represents an absolute path (You may also support relative path)
    - Changes the working directory of the process to ARG0
    - Hint: *man* chdir
  - Any other input is redirected into the *system* library function
    - *man* system
      - The system library function invokes the given command in the default shell (i.e. bash)