Chapter 3

Processes

Chapter Objectives

To understand concepts of Processes and Shell Programs used over the duration of the course.

Objectives

For this chapter, the following are the objectives:
- Introduction to Systems Programming.
- Understanding Implementation and Standards in Linux.
- Understanding Architecture and Design of Linux Kernel.

Slide #3-1

Notes

In this chapter, we examine the terms programs, process and threads.
The objective of this chapter is to provide an understanding of the concepts on processes and threads that would be used over the duration of the course.

Chapter Organization

1. **Objective**: Introduction to
   - Programs
   - Processes

2. **Description**: A process is the basic active entity. Threads are abstractions based on the process model, and have current relevance. This chapter provides an introduction to the concepts used in course.
3. **Concepts Covered in Chapter**:
   - Introduction to Programs, Processes and Threads.
   - Design UNIX Processes and Threads model.
   - LINUX Notes

4. **Prior Knowledge**:
   Discussion questions are,
   - What are Programs? .. Processes? .. and, Threads?

5. **Teaching & Learning Strategy**:
   - UNIX Processes and Threads: Structure and Layout?,
   - Processes created using `fork()` and `exec()` ?,
   - UNIX Process Model: Parent; Child; Good Parent/Bad Parent; Orphans Processes and Zombies.
   - `wait()` and `kill()`?
6. **Teaching Format**:
   Theory + Homework Assignments
7. **Study Time**: 150 Minutes (Lecture & Theory) +
   ~45 minutes (Homework Assignments)
8. **Chapter References**:

Stevens; APUE:	Ch #1, #2
Vahalia; UI:	Ch #1, #2.1-2.4
Robbins & Robbins; USP:	Ch #1.

Overview of a Process

- User Space
- Kernel Space
- Passing Data .. between User Space and Kernel Space
- Task Struct

System Call Interface Conventions

Always Returns `int` .. -1 for failure, success otherwise.

Global Variable
`errno  (?? MT Safe)`

Slide#3-3

Notes:

1. User Space
   1.1. Memory space where user processes run.
   1.2. This memory space is protected.
      Processes execute non-privileged (.. computes, moves) instructions on CPU.
      Each user is prevented from accessing and interfering with another user's memory.
   1.3. A process running user-space code, is said to be operating in "user mode".
2. Kernel Space
   2.1. Memory space where kernel services are provided.
   2.2. This memory space is a privileged area.
      Processes .. execute privileged instructions .. e.g. accessing devices .. kernel process
      A User Process accesses it only through the system call interface.
      Kernel can access any space on the system.
   2.3. A process running kernel code, is also known as "kernel" process and is said to be operating in "kernel" mode.

2.4. A user process "becomes" a kernel process, when a system call gets invoked, and starts executing kernel code.

3. Passing Data between User Space and Kernel Space

3.1. Moving data between user and kernel spaces is important as they are in different address spaces, with differences in levels of protections and privileges.

3.2. When a system call is invoked, the corresponding syscall vector# and the arguments to the call are passed from user space to kernel space.

the system call vector# .. is passed either on the stack, or in the CPU registers, depending on the UNIX implementation,

the system call arguments .. are passed via the u area

3.3. When a system call returns, data values are passed from kernel space to user space, either

using return value, mainly for flags and function status .. data which, can fit into a primitive data type such as an int or long. .. e.g. int fd=open(…);

using function arguments, mainly for strings or array data that cannot be returned as values, is passed via pointers. .. e.g. int retval=read(fd,buf);

4. Task_struct (or u area) control information needed by the kernel, for a process. It includes:

4.1. process specific information, such as files open; root and current directory; arguments to system calls; and, sizes for the code, stack and data .

4.2. A pointer to the process table entry .. e.g. for scheduling.

4.3. Table containing Open File Descriptors .. 'fd' table.

4.4. The Kernel Stack for the process that contains information on system calls. The kernel stack is empty when the process is in user mode.
Note: Some Implementations exclude the kernel stack from the u area.

Kernel Entry Points

Kernel Entry Points
- System Calls .. by applications and system utilities,
- Hardware Interrupts .. from device handlers, and,
- Error Conditions

Slide# 3-4

Notes

1. Kernel Entry Points : Entry points into the kernel provide access to functions and services provided by the kernel.
2. There are three classes of entry points that access or activate kernel services. It includes:
   2.1. system calls .. used by user applications and system utilities, are handled by a trap instruction that switches a system call from user mode to kernel mode.
   2.2. hardware interrupts .. service requests from device handlers, are interrupts from devices requesting service from device handlers in the kernel.
   2.3. error conditions .. are also handled by trap instruction.
3. All entry points interrupt kernel activity.

Programs and Processes

Program ..
File containing a sequence of instructions.
Source, Object and Executable .. Files.

Process .. running instance of a program.

Slide# 3-6

Notes

1. Programs .. are a sequence of instructions. Can be source, object or executable files.
2. Processes .. A process is a running instance of a program.
   2.1. A process has an address space, and at least one flow of control called a thread.
   2.2. The data variables in a process exist
   for the life of the process (static storage), or
   when execution enters a block and, de-allocated when execution leaves the block (automatic storage)
3. When does a Program become a Process?
   3.1. When a program is run or executed, the executable is copied into a program image in memory. A process is an instance of a program that is executing.
   3.2. The OS then adds appropriate information in the kernel data structures .. PID, state; and, allocates memory and necessary resources to run the program code. The program now becomes a process. Each process i.e. instance of a program, has its own address space and execution state.
4. Chapter Reference: Robbins & Robbins #2.1

5. Source Program Compatibility .. Thoughts

5.1. Here is a code snippet that demonstrates that, all compilers are different!! Note: The source code, per C Standard, is invalid. (and, represents bad programming practice) ..

5.2. however, run the executable code after compilation on as many systems and compilers, too (including gcc and, any native/vendor provided cc on the same box).

5.3. With different compilers, the same source program gives different output results!

```
int main() {

int a;

/***    #1. try different locations for post-increment op  */

a=5; printf("1a. %d+%d=%d\n", a++, a++, a + a);
a=5; printf("1b. %d+%d=%d\n", a, a, a++ + a++);
a=5; printf("1c. %d=%d+%d\n", a++ + a++, a, a);
a=5; printf("1d. %d=%d+%d\n\n", a + a, a++, a++);

/***    #2. try pre-incr op */

a=5; printf("2a. %d+%d=%d\n", ++a, ++a, a + a);
a=5; printf("2b. %d+%d=%d\n", a, a, ++a + ++a);
a=5; printf("2c. %d=%d+%d\n", ++a + ++a, a, a);
a=5; printf("2d. %d=%d+%d\n\n", a + a, ++a, ++a);

/***    #3. try mix of post-incr op & pre-incr op */
a=5; printf("3a. %d+%d=%d\n", ++a, a++, a + a);
a=5; printf("3b. %d+%d=%d\n", a, a, ++a + a++);
a=5; printf("3c. %d=%d+%d\n", ++a + a++, a, a);
a=5; printf("3d. %d=%d+%d\n\n", a + a, ++a, a++);

/***    #4. try another mix of post-incr op & pre-incr op */
a=5; printf("4a. %d+%d=%d\n", a++, ++a, a + a);
a=5; printf("4b. %d+%d=%d\n", a, a, a++ + ++a);
a=5; printf("4c. %d=%d+%d\n", a++ + ++a, a, a);
a=5; printf("4d. %d=%d+%d\n\n", a + a, a++, ++a);


/***
Questions:
Are all C compilers built the same?
Are the answers same as above ?

Note: The above printf() statements #1-#4, are illegal constructs
"No variable can be modified more than once in any statement".

*/

}
```

Processes and Threads

Process .. has at least one thread of execution.

Multiple threads execute in the context of same process.

Slide# 3-7

Notes

1. Linux is a Multi-Processing system.
   1.1. Multiple processes (It may be for same executable), may be running simultaneously.
   1.2. All of them have different execution contexts .. CPU state is different in each process.
   1.3. The sequence of instructions in each thread appears "uninterrupted" to the user.
   1.4. On the CPU, however, the threads of execution from the various processes are inter-mixed.
2. Processes and Threads of Execution
   2.1. Each process has atleast one thread of execution, consisting of a sequence of instructions. The program counter in a CPU keeps track of the next instruction to be executed by that processor.
   2.2. Multiple threads execute within the context of the same process.
   an extension of the process model,
   Multiple threads avoid context switches and allow sharing of code and data.
   2.3. The point at which execution switches to another process is called a context switch.
   2.4. A thread in OS terminology,

represents a thread of execution within a process.

has its own execution stack, program counter value, register set and state,

allows a programmer to achieve parallelism with low overhead.

2.5. Threads reside in the same process address space, and share the same process resources.

2.6. There is relatively less overheads of creating, maintaining and running threads. Threads, are therefore, sometimes called lightweight processes. Processes are, in contrast, heavyweight.

3. Chapter Reference: Robbins & Robbins #2.2

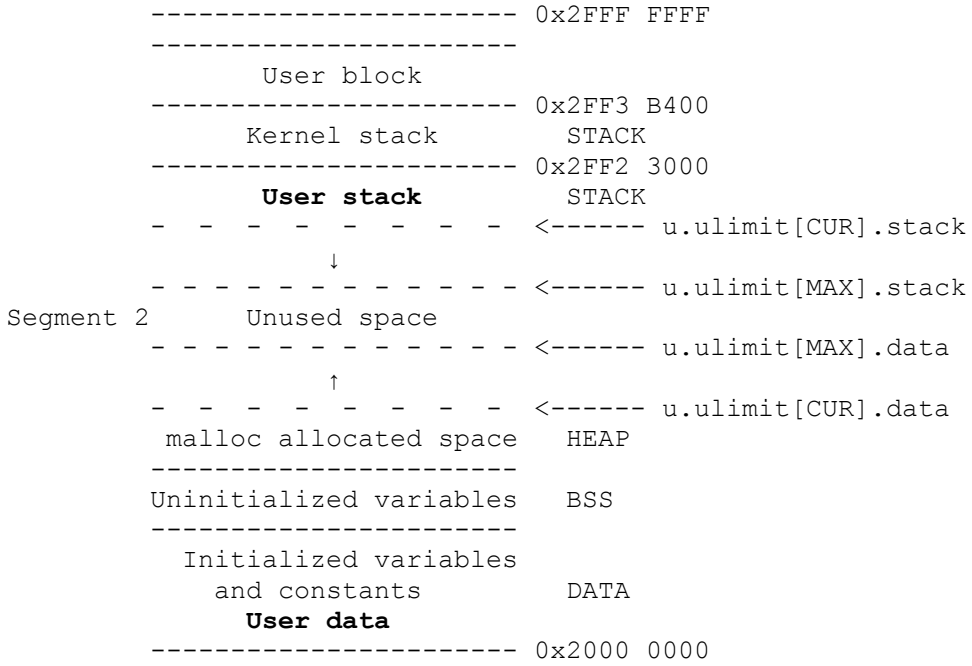
Process Memory Architecture

Process Memory Consists of
Code, and
Data
Globals (Initialized DATA; Uninitialized BSS); Heap; and, Stack.

Slide# 3-8

Notes:

1. Here is another "view" of the data component of the process address space ..



2. After loading, the program executable can appear to occupy a contiguous block of memory called a program image. The program image has several distinct sections.
   2.1. The program text or code,
   2.2. The initialized and uninitialized variables, and
   2.3. The heap, stack and environment.

Inputs to a program

inputs to a program:
- stdin
- files
- command line arguments
- environment variables

Slide# 3-9

Notes

1. stdin .. for interactive input to the program
   1.1. defaults to keyboard.
   1.2. can be re-directed from a file.
2. files .. for large volume, or complex data input
3. command line arguments .. for values that can change from run to run, and accessed using ..
   3.1. Every C program starts with main(int argc, char **argv). Command line arguments are available within a program via argv[i].

```
main(argc, argv)
        int argc;       /*  number of elements */
        char **argv;    /*  vector of cmd-line args .. null terminated */
```

4. environment variables .. for values that change from user to user.
   4.1. usually used to pass in parameter information, for configuring a program. Programs often use file locations set in environment variables, instead of hard-coded locations.
   4.2. set via the .*rc scripts (.bashrc, .cshrc, .kshrc) , and are passed to every program invoked by the shell.

Environment Variables as Inputs to a Program

inputs to a program:
- stdin
- files
- command line arguments
- environment variables

passing command line arguments, environment variables

```
main(argc, argv, envp)
    int argc;        /*  number of elements */
    char **argv;     /*  vector of cmd-line args .. null terminated */
    char **envp;     /*  vector of env vars .. null terminated */
```

Slide# 3-10

Notes

1. Envirnonment Variables are accessed using either
   1.1. global variable environ
   1.2. getenv()function .. that accesses global variable environ
   1.3. envp argument to main() .. main() .. can take 3 arguments ..

```
main(argc, argv, envp)
      int argc;       /*  number of elements */
      char **argv;    /*  vector of cmd-line args .. null terminated */
      char **envp;    /*  vector of env vars .. null terminated */
```

   1.4. Most programs use getenv(), therefore, envp is not used.
   1.5. Difference between environment and shell variables,
      1.5.1. environment variables are "exported" into the environment. The environment vector consisting of "exported" variables is passed to the main() routine of every program, started from that shell.
      1.5.2. "un-exported" shell variables are similar in scope to "local" C variables, and not passed to sub-shells ..

2. Code Example -- envp argument

```
#include <stdio.h>

int main(int argc, char** argv, char**envp) {

   int i=0;

   for (i=0; envp[i]; i++) {
        printf("%s\n", envp[i]);
   }
}
```

3. Code Example -- global variable environ

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

extern char **environ;
int main(int argc, char** argv, char**envp) {
   int i;
   for (i=0; environ[i]; i++) {
        printf("%s\n", environ [i]);
   }
}
```

4. Code Example -- getenv function

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


int main(int argc, char** argv, char**envp) {

   char *var;

   var=getenv("TERM");

   printf("%s\n", var);
}
```

execve()

```
int  execve(pathname, argv, envp)
    char *pathname;
    char **argv;
    char **envp;
```

Slide#2-10

Notes

1. execve() calling conventions .. executes "program" passed in as the first argument.
   1.1. "program" execution starts with its main(), and, is passed argv and envp.
   1.2. No new process is created. The code and data segments of the current process are overlaid with the executable file.
   1.3. A successful call to execve() does not return.

2. What does the following program do?

```
int main() {
          int  i=0;
          char *newargv[16];
          char *newenvp[16];

          newenvp[0]=0;
          newargv[i++]="cat";
          newargv[i++]="-l";
          newargv[i++]="testfile";
          newargv[i++]=0;

          execve("/bin/ls", newargv, newenvp);
}
```

The best practice to avoid issues as above, is to use:

```
int main() {
          int  i=0;
          char *newargv[16];
          char *newenvp[16];

          newenvp[0]=0;
          newargv[i++]="ls";
          newargv[i++]="-l";
          newargv[i++]="testfile";
          newargv[i++]=0;

          execve(newargv[0], newargv, newenvp);
}
```

3. Process Attributes inherited across the call to execve()

   3.1. The process attributes inherited across the call to execve are:

   process ID getpid()

parent process ID	getppid()
user ID	getuid()
working directory	getcwd()
file descriptors	open(), close()
umask	umask()
control terminal	tty()
process group id	getpgrp()
signal mask	sigaction(), sigprocmask()
interval timers	alarm()
resource usage	getrusage()
resource limits	getrlimit()

3.2. All file descriptors normally remain open across the call to execve().

The close-on-exec flag associated with the file descriptor determines whether or not a file remains open across calls to execve.

If cleared, the file descriptor remains open in the new image loaded by the call to execve.

If set, the file descriptor is closed in the new image loaded by the call to execve.

4. Process Attributes that are reset across the call to execve()

4.1. The process attributes that are not inherited across the call to execve are:

effective user ID	geteuid() .. see chapter #3
effective group ID	getegid()
signal handlers	.. see chapter 8

Alternative interfaces to execve()

```
int  execv();
int  execve();
int  execvp();

int  execl();
int  execle();
int  execlp();
```

Slide#2-10

Notes

1. exec() family of calls
   1.1. execve() system call overlays a new program.
   1.2. execve() requires that the user pass in the absolute path name of the command file to be overlayed. There are several library fun
   1.3. These are the functional interfaces available, that provide access to execve():

        execv(char *pathname, char **argv);
        .. wrapper function on execve() and uses the global environment variable environ

        execve(char *pathname, char **argv, char **envp);

        execvp(char *command, char **argv); uses PATH env. variable

These are the functional interfaces available, that provide access to execve(), using a list of arguments:

execl(char *path,char *arg0,char *arg1 … argn,0);

execle(char *path,char *arg0,char *arg1 … argn,0, envp);

execlp(char *command, char *arg0,char *arg1 … argn,0);

Real and Effective Id's

- setuid()
- seteuid()

- setgid()
- setegid()

Slide#2-11

Notes

1. Every process has 4 numbers to identify who the process belongs to, and what access permissions they have. These numbers are called
   1.1. UID real user ID.
   1.2. EUID effective user ID.
   1.3. GID real group ID.
   1.4. EGID effective GID.
2. Ordinarily, a process inherits these values from its parent and cannot change them.
3. Only a process owned by root, can change its real and effective numbers.
4. The execve() call can change the effective userid of a process. If the command file to be exec()'d has the setuid bit or the setgid bit turned on, then the appropriate user and group for the new process will be changed.

```
> ls -l /bin/passwd
-r-sr-sr-x   1 root sys  21964 Apr  6  2002 /bin/passwd
```

   4.1. This allows non-root users to run the program as root, to allow edits to the passwd file (that is owned by root, and is not editable by non-root users)
   4.2. The resulting process has the effective user id of whoever owns a.out.

fork()

```
int  fork();
```

Slide#2-11

Notes

1. fork() calling conventions .. provides a mechanism for spawning new processes under UNIX.
   1.1. fork() creates a new process, that is a copy of the current process ..

   The newly created copy is the "child process". In the child, fork() returns 0.

   The other is the "parent process". In the parent, fork() returns the PID of the new child.

   1.2. A successful call to fork() creates a copy of the entire calling process.

   u_area, code and data for the new process are copied into a new process table entry.

   both parent and child processes share the same file descriptors, and have same values for almost all process attributes.
2. Process Attributes that are different in parent and in the child are:
   2.1. The process attributes that are not inherited across the call to execve are:

process ID	getpid()
parent process ID	getppid()
child process resource usages	reset to zero.

3. What does the following program do?

```
#define _POSIX_SOURCE 1

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

main() {
printf("Before fork()\n");
fork();
printf("After fork() .. my pid=%d\n",
       (int) getpid());
}
```

4. A template for fork()

```
{
      pid_t pid;

      pid = fork();

      if (pid < 0) {
            /* FORK failed ?? */
            return -1;
      }

      if (pid==0) {
            /* CHILD process executes this code */
      }
      else {/* PARENT process executes this code */
      }

}
```

exit() vs _exit()

exit()

- part of the std C library
- flushes the FILE * buffers, and, then closes fd's,
- called only by parent process

_exit()

- system call,
- closes open fd's only
- called by child processes

Slide#2-12

Notes

1. exit().. calling conventions .. a part of the stdio library, and, originally written by Dennis Ritchie..

   1.1. exit() terminates a process, and sets an exit status ..

   exit() first flushes all FILE buffers, and calls _exit()

   children inherit all data and variables from their parents. This includes FILE * variables: stdin, stdout, stderr. These struct variables contain buffer fields which contain pending output left over from the parent.

   exit() calls fclose() on all open FILES .. regardless of parent or child.

   flushes the output.

2. _exit().. calling conventions .. is a system call ..

   _exit() closes all open , and sets the processes running state to zombie.

   _exit() is not POSIX

3. It is important that children who have not made a call to exec(), always use _exit() to terminate so that they will not cause the output from the buffers for stdout to be flushed in the child, and, then again in the parent.

wait() system call

wait()

- causes parent process to suspend execution, and
- block until the child process exits, and
- clean-up child process related kernel resource table entries.

wait(&status)

Slide#2-13

Notes

1. A parent process has three options after creating a child process using fork()

   1.1. exit() before its child .. If a process exit()s before its child, the child process becomes an orphan.

   The child processes are now inherited by init (PID 1), and,

   cleaned up upon exit, by init.

   1.2. parent does not call wait() .. If a parent process keeps running without calling wait(),

   The parent process is a "bad" parent.

   When the child process exits, it is a zombie. The kernel assumes the child process will be cleaned up by the parent, and, so does not do cleanup.

   if the parent eventually exits, dead children are cleaned up by init. However, exited processes when the parent process is still running can cause the kernel process table to fill up ..

   1.3. parent calls wait() .. If a process calls wait(),

   The parent process enters a sleep state and suspends execution,

   When the child process exits, the call to wait() returns. The wait() system call returns the PID of the exited child.

2. wait().. calling conventions .. allows parent to wait() for a child to exit, and also clean-up

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

pid_t wait(int *status);
```

2.1. returns the PID of the exited child;

2.2. takes as an "out" parameter an int variable, that provides information about how the child process terminated.

2.3. various macros available in <sys/wait.h> to determine how the child exited

Exited or Signalled .. a child either exited normally or was terminated as a result of a signal.

```
#define WIFEXITED(x)   ((!WIFSTOPPED(x) &&
(WTERMSIG(x)==0))

#define WIFSIGNALLED(x) ((!WIFSTOPPED(x) &&
(WTERMSIG(x)!=0))
```

Exit Status bits .. Exit status of a process

```
#define WEXITSTATUS(x) (((x) >> 8) & 0377)
```

Signal bits .. if the child exited as a result of a signal, then the signal# is returned. If there was a core dump as a result of the signal, then that information is available, too.

```
#define WTERMSIG(x) ((x) & 0177)

#define WIFCORE(x) ((x) & 0200)
```

Stopped Jobs .. if the child is stopped and not exited as a result of a signal, then the signal# is returned, and, also information that the job was stopped is available.

```
#define WIFSTOPPED(x) ((x) & 0377) == 255)

#define WIFCORE(x) ((x) >> 8) & 177)
```

waitpid() system call

waitpid()

- allows parent to wait for a particular child

Slide#2-13

Notes

1. waitpid().. calling conventions .. allows parent to wait() for a particular child.
   1.1. is a POSIX function,
   1.2. takes three arguments

```
#include <sys/wait.h>
pid_t waitpid(pid__t pid, int *status, int OPTION_FLAGS);
```

   1.3. pid ..

   if PID == -1 then, waitpid() waits for any child.

   if PID == 0 then, waitpid() waits for any child in the same process group as the caller.

   if PID > 0 then, waitpid() waits for the child with that PID.

   1.4. status .. as above
   1.5. OPTION_FLAGS ..

   WNOHANG .. causes non-blocking wait.

   WUNTRACED .. causes routines to return the PID of stopped processes.

fork() and wait() system call

Combination of fork() and wait() provides bootstrapping.

Notes

1. Code Sample

```
#define _POSIX_SOURCE 1

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

void ForkAndWait();

int main() {

   fprintf(stderr, "My PID is %d.\n", (int) getpid() };

   ForkAndWait();
   ForkAndWait();
   ForkAndWait();
}
```

```
void ForkAndWait() {

   pid_t     mypid;
   int       status;

   mypid = fork();

   switch ((int) mypid) {
        case -1:
             fprintf(stderr, "Err: fork() failed:%s\n",
                  strerror(errno) );
             break;
        case 0:
             fprintf(stderr, "Child Process!\n");
             sleep(1);
             _exit(0);
             break;
        default:
             fprintf(stderr, "Parent Process!\n");

             mypid=wait(&status);
             PrintStatus(stdout, mypid,status);
}

void PrintStatus(FILE *wfp, int mypid, int status) {

   fprintf(wfp, "\n");
   fprintf(wfp, "%6d = wait()", mypid);

   if (WIFEXITED(status)) {
        fprintf(wfp, "Exit Status = %3d\n",
             WEXITSTATUS(status));
   }
   if (WIFSTOPPED(status)) {
        fprintf(wfp, "Stop Signal = %3d\n",
             WSTOPSIG(status));
   }
```

```
    if (WIFSIGNALED(status)) {
         fprintf(wfp, "Exit Signal = %3d\n",
              WTERMSIG(status));
         fprintf(wfp, "Core Dump : %s\n",
              WIFCORE(status) ? "Yes" : "No");
    }
}
```

Interpreter Files

Files begin with the '#!'

Slide# 2-13

Notes

1. An interpreter file is an ASCII file that begins with the '#!'
2. When execve executes a file beginning with the '#!'
   2.1. it reads the first line from the file, and
   2.2. creates a 2 element string vector containing
   the first argument .. file or path name, and,
   the second argument ..
   2.3. and, executes the first element in the vector.
3. The '#!' provides a scripting mechanism.

Chapter Summary

This chapter, provided:
- An overview of the UNIX System Architecture and System Call Interface.
- An Understanding UNIX Process and Thread models.
- An Understanding UNIX Process Creation and Termination.

Slide #2-16

Chapter 3
Processes

What does the following program do?

```
#define _POSIX_SOURCE 1

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

void main() {
      int i=0;

      if (fork() != 0) {
            exit(0);
      }

      for (i=0;i<5;i++) {
            sleep(10);
            printf("Hello! My pid# is %d.\n", getpid());
      }
}
```

Chapter 3
Processes
Terms & Concepts Worksheet
Table #1 ("Basic")

1. Programs.	Source/EXE. Refers to a "file".
2. Process.	instance of a running program.
3. Address Space	addressable memory for a process
4. User Space	A process running CPU instructions code
5. Kernel Space	A process running kernel code ..privileged instructions, accessing devices.
6. Entry Points.	Kernel Entry Points a. System Calls .. by applications and system utilities .. handled by trap instruction. b. Hardware Interrupts .. from device handlers c. Error Conditions .. also handled by trap instruction.

Table #2 ("System Calls Interface Conventions")

7. System Calls Interface Conventions.	System Calls .. check the return status and reporting possible errors a. Return Status .. –1 is bad, 0 is good .. (or, any +ve return value) b. Error Info .. `errno` .. global variable in <errno.h>; `strerror(errno)` to get description statement of error. Generally, proper error handling is recommended. Helps debugging. Most frequent cause of system call failures is programmer-error, not end-user error.

Table #3 ("Inputs to a program")

8. Inputs to a program	a. stdin .. interactive input from program b. files .. for large volume or complex data input c. command line arguments .. inputs vary from execution to execution, so do outputs d. environment variables .. for values that change from user to user - vs. un-exported local shell variables

Table #4 ("`exec()` family of calls")

9. `exec()` family of calls	a. `execve()` .. overlay current process with code and data of another. b. some process attributes inherited .. e.g. PID; while some not .. EUID. c. other variations to execve available .. see page 21 of notes

Table #5 ("`fork()` call")

10. `fork()` call	a. `fork()` .. creates a new process; copies address space; b. returns 0 in child; returns childs PID in parent. c. some process attributes different .. e.g. PID; while some not .. fd's. d. `vfork()` .. variation of `fork()` - creates a new process, but does not copy all pages. - copy is done, when needed for access by the new process. "Copy-on-demand" feature. - in current UNIX's .. fork() is a wrapper on vfork()

Table #6 ("`exit()` vs `_exit()` call")

11. `exit()` vs `_exit()` call	a. `exit()` .. library function; flushes FILE buffers, and closes fds; b. not desirable .. to write and flush files, and affect parent/sibling processes. c. `_exit()` .. system call; heart of `exit()` .. only closes fd's.

Table #7 ("`wait()` and process termination state")

12. `wait()`	a. parent suspends execution .. till child exits; parent can check exit statuses of child b. cleans-up process table entries and kernel resources after child exits c. `waitpid()` .. variation of `wait()`; can wait for specific child PID, or any
13. Process Termination	a. parent exits before child; the child process becomes an **orphan**. b. parent not `wait()`'ing on child ; the child process upon exit marked **zombie**. c. parent `wait()`'ing on child ; good parent. ☺

Chapter 3

Processes

Assignment Questions

1.1 Terms Review: (One–Liners)
a) Process b) wait c) orphan process d) zombie process e) interpreter

Programming Questions

1. Write a simple program to print the PATH environment variable on your system.
2. Write a simple program to print the program's name and pid.
   2.1. Call '/bin/ls'.
   2.2. Call 'ls –l /bin/ls'
3. Change the above program to call '/bin/ps -f". Did the 'pid' show up in your output.
4. Determine the limit of maximum number of processes forked, using sysconf().
5. Write a simple program that prints "Hello World" in the background .. in a loop 5 times with a sleep 10 secs.

.

Extra Credit

6. Compare each of the following for program sizes.
   6.1. Explain how its size is derived.
   6.2. Explain the basic concepts involved in its process memory address.

```
/* #1.   Global Un-initialized Array */
#include <stdio.h>

int myarray[50000];

int main(void) {
  myarray[0]=1;
  return 0;
```

```
}

/* #2.   Global Initialized Array */
#include <stdio.h>

int myarray[50000]={1};

int main(void) {
  myarray[0]=1;
  return 0;
}

/* #3.   Stack Un-Initialized Array */
#include <stdio.h>

int main(void) {
  int myarray[50000];

  myarray[0]=1;
  return 0;
}

/* #4.   Stack Initialized Array */
#include <stdio.h>

int main(void) {
  int myarray[50000]={1};

  myarray[0]=1;
  return 0;
}
```

7. Compare the outputs on Page 16 of this Chapter, on different compilers.

Chapter 3

Processes

Assignment Hints

1. Hint: use either global variable, or getenv or envp
2. Hint: a) argv[0] is program name.
   b) exec() /bin/ls c) set argv[1]="-l" and argv[2]="/bin/ls"
3. Hint: a) as above b) consider time of execution …
4.
5. Hint: Parent process terminates immediately upon fork(). Child process prints ..

6. int: use the output of 'size' command and 'ls –l' also 'pmap'..
7. Hint: Concepts:
   7.1. some compilers evaluate expressions left to right, while some right to left, also
   7.2. some compilers evaluate pre-increments prior to the statement, while
   7.3. some evaluate when accessing variable.
   7.4. Similar concepts are valid for post-increments .. except that the point of the increment can be after the statement.
   7.5. Main thoughts: Compilers are "expected" to have same outputs for "standard" constructs. Any deviation, and, the outcome can be unpredictable.