System Calls

-An O/S kernel acts as the moderator between applications and the resources of the computer. It provides a set of functions to other programs called *system calls*.
-System calls are used to request access to the resources of the machine, to communicate with other currently running programs, and to start new programs. A collection of system calls is sometimes referred to as an application programming interface (API).
-Different operating systems provide different sets of system calls.
-Standards (for example, POSIX) have been developed to regularize at least a subset of system calls common to all systems.
-Modern Unix kernels provide several hundred different system calls. These system calls are typically broken up into families of functions, each of which target a specific purpose or type of operation.

Categories of system calls:-

1. Memory management system calls:-
   They ask the O/S to manipulate a block of memory in some manner, typically so that the memory can be used by an application program. Example system calls include mmap(), shmget(), mprotect(), mlock(), and shmctl().
   These operations involve manipulating the low-level attributes of memory as managed by the operating system. For example, an O/S typically maintains levels of memory protection, where the lower levels (such as that used by the O/S) cannot be accessed by other programs. Some of these system calls allow the access permissions of portions of memory to be changed so that application programs can use it.
2. Time management system calls:
   Ask the O/S to access the system clock, in some cases taking action based upon its value.
   Examples include time(), gettimer(), settimer(), settimeofday(), and alarm(). These system calls either retrieve values from the system clock, or start or stop timers based upon the system clock.
   The alarm() function arranges for a signal to be sent to a process once the given amount of time has passed.
3. File system calls
   Ask the O/S to access a file or device. Example system calls include open(), read(), write(), close(), creat(), lseek(), and link().
   These bear a resemblance to the C standard library functions fopen(), fread(), fwrite(), and fclose().

4. Process system calls
   Ask the O/S to run another program, or control how it runs. Examples include fork(), execl(), execv(), and wait().

5. Signal system calls
   Provide a rudimentary form of interprocess communication. Examples include signal(), pause(), kill(), and sigaction(). They are typically used to send a message to a process, telling it about an error it has committed and asking it to terminate.

6. Socket system calls
   They allow a program on one computer to communicate with a program on a second computer through a network.
   Examples include socket(), bind(), connect(), listen(), accept(), send(), and recv().

Other system calls include those used for message passing, shared memory, semaphores, and thread management.

Libraries and System Calls
-A common hierarchical relationship between library functions and system calls is that many library functions use system calls as part of their code. For example, the malloc() family of functions is built on top of the mmap() and brk() functions, meaning that the latter are called upon to get the job done. The malloc() function is in the C standard library, while mmap() is a system call.
-Library functions sometimes provide additional capabilities not available directly through system calls. For example, the C standard library functionsf open(), fread(), fwrite(), and fclose() provide buffering, whereas the system calls open(), read(), write(), and close() generally do not. When an fread() call is made, more than the requested amount is read() from the file. The extra bytes are held in a buffer managed by the library code. When the program next calls fread(), the system may be able to satisfy the request using bytes already in the buffer, eliminating the need for another read() system call.
-Standard library functions often optimize operations to minimize the number of system calls, thus speeding up program execution.
-Library functions tend to be more portable than system calls. They both have standards: the ANSI C standard defines many library functions, while the POSIX standard defines many system calls. However, since system calls provide direct access to the kernel, different operating systems (including different variations of Unix) will have at least slightly different system calls. Therefore, applications that are intended to be ported to many different systems are usually coded using library functions instead of system calls whenever possible.
-The man pages for system calls, library calls, and system programs are all stored in different directories, often called sections or chapters. The man pages for system programs are in section 1, those for system calls in section 2, and those for library functions in section 3. Using man, the section can be specified by a command line argument. For example:

```
man 2 stat
[... provides man page for system call stat() ...]
man 3 printf
[... provides man page for library function printf() ...]
man 1 ls
[... provides man page for system program ls ...]
```

Process System Calls
Process system calls deal with the starting up of new programs. Example process system calls include fork(), execl(), execv(), and wait().
In order to start a new program, a currently running program makes a clone of itself; the clone then replaces its code with the code of the new program. All programs are therefore clone-descendants of the first program run on the system after the kernel boots.

A process is a running program, that is, a program currently in execution. On a Unix system, one can run the ps program to look at the current process list:

```
          user@my_machine> ps
          or
```

to show all processes currently running on the system:

```
          ahoover@video> ps -ef
```

Shell commands used to alter how a program is run.

Command/key stroke	**Effect**
CTRL-C	terminate process currently connected to stdin
&	run program in background; stdin stream is disconnected
CTRL-Z	suspend process currently connected to stdin
bg	restart suspended process; stdin still disconnected
fg	reconnect suspended/background process to stdin
kill #	terminate the given process ID

these commands affect how a process connects its stdin ("standard in") stream.

The fork/exec process model

The Unix process management model is split into two distinct operations :

- The creation of a process
- The running of a new program

The creation of a new process is done using the fork() system call and a new program is run using the exec(l,lp,le,v,vp) family of system calls.

These are two separate functions which may be uses independently, a call to fork() will create a completely separate sub-process which will be exactly the same as the parent. Conversely calling one of the exec family of functions will terminate the currently running program and starts executing a new one in the context of the existing process.

The main reason this model is used is the simplicity of operation, when creating a new sub-process the current environment from the parent is used so the programmer need not setup a new environment to run the program. After the fork call the program may then use system calls to modify the environment to suit the child process and then use the exec functions to run the new process required.

fork()

The fork() system call creates a clone of the currently running program. The original program continues execution with the next line of code after the fork() function call. The clone also starts execution at the next line of code. For example, consider the following code:

```
#include <stdio.h>
#include <unistd.h>
main()
{
int i;
printf("Ready to fork...\n");
i=fork();
printf("Fork returned %d\n",i);
while (1);
}
```

Explanation: The original program(the parent process) displays the output “Ready to fork . . . ” and then calls the fork() function.
At that point, the program is cloned (the child process starts) and there are two copies of it running. Each executes the line to display the code “Fork returned . . . ”. However, notice that the return values are different. The parent process (the original) gets a different return value from fork() than the child process (the clone) does. In the parent process, fork() returns the PID of the new child process, while in the child process, fork() returns zero.

The exec family of functions
The main use for the fork function is to allow for the execution of another process within a program, fork creates a child process which is an exact clone of the parent process if we wish to execute a new function after this has been done we use the exec family of functions.
These functions replace the current process (which after fork is a clone of the parent) with the new process called by exec. This will replace all of the text, data, stack segments and heap of the child process with that of the new process called.
There are six different exec functions which are described in the exec man pages
Function Declaration: Exec family of functions

```
int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg , ..., char * const envp[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
```

The first difference in these functions is that the first four take a pathname argument while the last two take a filename argument. When a filename argument is specified

- if filename contains a slash, it is taken as a pathname,
- otherwise the executable file is searched for in the directories specified by the PATH environment variable

A simple fork and exec example
The following program demonstrates how to use fork and exec to run a process and uses a small program called child to demonstrate how the parent and the child are running at the same time.
The code for child.c (child program to be run by parent program) is as follows

```
#include <stdio.h>
int main(void)
{
while(1)
printf("C");
}
```

Know the parent program which executes the child program

Parent program to run child program

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <error.h>
#include <signal.h>
#include <errno.h>
```

```
int main(void)
{
pid_t pid;
int status;
if((pid =fork()) < 0)
{
// probably out of processes
status =-1;
}
else if (pid == 0)
{
// in child so we execute process
// use the execl function to to run a shell an execute the child program
execl("/bin/sh","sh","-c","child",(char *)0);
}
while(1)
printf("P");
printf("end of program");
}
```

When this program is executed the child program is spawned as a child process, and the main (parent) process continues now the console will print out both C for child and P for parent.
The execl function is used in the above example by calling the /bin/sh shell command and running sh -c which tells the shell (in this case sh the C shell) to start in command mode where the next argument is the command to be run which in the above example is the child program.

fork exec example 2
The following example executes a child process and waits for it to die before continuing the parent process.
This is split into two sections, first is child2.c
Child program to be run by parent program

```
#include <stdio.h>
int main(void)
{
int i=0;
for(i=0; i<100; i++)
printf("c");
}
```

This program loops 100 times and prints out the character c to the console
The next program executes the above program (child2) and waits for it to finish before it continues
Parent program to run child program

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <error.h>
#include <signal.h>
#include <errno.h>
int main(void)
{
int c=0;
pid_t pid;
int status;
if((pid =fork()) < 0)
{
// probably out of processes
status =-1;
}
else if (pid == 0)
{
// in child so we execute process
execl("/bin/sh","sh","-c","child2",(char *)0);
}
else
{
// now wait for child to die
while(waitpid(pid, &status,0) < 0)
{
printf("in wait pid \n");
if(errno !=EINTR)
{
status=-1;
break;
}
}
}
for(c=0; c<100; c++)
printf("p");
printf("\nend of program");
}
```

If the above program is modified to run the original child program the child will run at the same time as the parent process but when the parent die so will the child.