**Unix System Calls**

A system call is a method for a computer program to request a service from the kernel of  the operating system on which it is running. A system call is a method of interacting with  the operating system via programs. A system call is a request from computer software to  an operating system's kernel.

The Application Program Interface (API) connects the operating system's functions to user  programs. It acts as a link between the operating system and a process, allowing user-level  programs to request operating system services. The kernel system can only be accessed  using system calls. System calls are required for any programs that use resources.

Unix system calls can be loosely grouped into the following three main categories: ∙ File management

∙ Process management

**File management system calls**

1. open()

Syntax: int open(char \*filename, int mode, int permissions)  Example: fd = open(“text.data”,O\_CREAT | O\_RDWR, 0600)

Note: open( ) allows to open or create a file for reading and/or writing. Filename is the  absolute or relative pathname and mode is a bit wise ORing of read/write flags together  with zero or more miscellaneous flags. permissions is a number that encodes the value of  the file permission flags, and should be supplied to when a file is being created. Main  flags are O\_RDONLY (open for read only), O\_RDWR (open for  write and read), O\_WRONLY (open for write only). O\_APPEND,  O\_CREAT, O\_EXCL are miscellaneous flags. open () returns a non  negative file descriptor if successful; otherwise it returns –1.

2. read()

Syntax: int read(int fd, char \*buf, int count)

Example: charsRead = read(fd, buffer, BUFFER\_ASIZE); Note: read() copies count number of bytes from a file referenced by a file descriptor fd  into a buffer buf. If successful it returns the number of bytes  that it read; otherwise, it returns –1.

3.write()

Syntax: int write(int fd, char \*buf, int count)

Example: charsWritten = write(fd, buffer, BUFFERSIZE);

Note: write() copies count bytes from a buffer to a file referenced by the file descriptor. If  successful it returns the number of byes that were written; otherwise, it returns –1.

4. close()

Syntax: int close (int fd)

Note: close() frees the file descriptor. If successful it returns 0; otherwise, it returns –1.

5. lseek()

Syntax: long lseek( int fd,long offset, int mode)

Example: currentOffset = lseek(fd, 0, SEEK\_CUR);

Note: lseek() allows to change a file descriptors current file position. fd is the file  descriptor, offset is a long integer, and mode describes how offset should be interpreted. SEEK\_SET offset is relative to the start of the file

SEEK\_CUR offset is relative to the current file position

SEEK\_END offset is relative to the end of file

6. unlink()

Syntax: int unlink(const char\*filename)

Note: unlink() removes the hard link from the name filename to its file. If filename is the  last link to the file, the file’s resources are deallocated. If successful, unlink() returns  zero; otherwise, it returns –1.

Exercise: 1. Copy the content of one file to other.

 2. Use lseek() to copy different parts(initial, middle and last) of the file to other. **Process management system calls**

7. fork()

Syntax: int fork( )

Note: fork() causes a process to duplicate. If it succeeds, it returns the PID of the child to  the parent process, and returns 0 to the child process. If it fails, it returns –1 to the parent  process and no child is created.

main()

{

 /\* common code \*/

 pid = fork()

 if (pid == -1)

 {

 /\* error \*/

 }

 else if (pid != 0)

 {

 /\* parent code \*/

 }

 else

 {

 /\* child code \*/

 }

 /\* optional common termination part of parent and child \*/

} /\* end of main() \*/

8. getpid() and getppid()

Syntax: int getpid(void)

 int getppid(void)

Note: getpid() and getppid() return a process’ ID and parent process’ ID numbers,  respectively. They always succeed.

Exercise: 1. Write a program to create a child process. Display the process IDs of the   process, parent and child(if any) in both the parent and child processes.  2. Create a orphan process(parent dies before child – adopted by “init”   process) and display the PID of parent of child before and after it becomes   orphan. Use sleep(n) in the child to delay the termination. 9. exit()

Syntax: int exit (int status)

Note: exit() closes all of a process’s file descriptor, deallocated code, data and stack, and  then terminated the process. When a child process terminates, it sends its parent a  SIGCHLD signal and waits for its termination code status to be accepted.

10. wait()

Syntax: int wait ( int \*status)

Note: wait() causes a process to suspend until one of its children terminates. A successful  call to wait() returns the PID of the child that terminated and places a status code into  status.

Exercise: 1.Create a zombie (defunct) child process (a child with exit() call , but no   corresponding wait() in the sleeping parent) and allow the init process to   adopt it (after parent terminates). Run the process as background process and   run “ps” command.

 2. Modify the above program to include wait(&status) in the parent and to   display the exit return code(left most byte of status) of the child. 11.exec()

Syntax: int execl( char\* path, char\* arg0, char\*arg1,…… ,char\* argn, NULL) Note: it replaces the calling process’s code, data, and stack from the executable whose  path name is given.