**4ITRC2 Operating System Lab**

**Lab Assignment 4**

**1. Process Management System calls**

**fork()**

The fork() is one of the syscalls that is very special and useful in Linux/Unix systems. It is used by processes to create the processes that are copies of themselves. With the help of such system calls, the child process can be created by the parent process. Until the child process is executed completely, the parent process is suspended.

Some of the important points on fork() are as follows.

* The parent will get the child process ID with non-zero value.
* Zero Value is returned to the child.
* If there will be any system or hardware errors while creating the child, -1 is returned to the fork().
* With the unique process ID obtained by the child process, it does not match the ID of any existing process group.

#include <unistd.h>

#include <stdio.h>

int main() {

pid\_t pid = fork();

if (pid == 0)

printf("Child process\n");

else

printf("Parent process\n");

return 0;

}

**exec()**

The exec() is such a system call that runs by replacing the current process image with the new process image. However, the original process remains as a new process but the new process replaces the head data, stack data,etc. It runs the program from the entry point by loading the program into the current process space.

#include <unistd.h>

int main() {

execl("/bin/ls", "ls", "-l", NULL);

return 0;

}

**wait()**

As in the case of a fork, child processes are created and get executed but the parent process is suspended until the child process executes. In this case, a wait() system call is activated automatically due to the suspension of the parent process. After the child process ends the execution, the parent process gains control again.

#include <sys/wait.h>

#include <unistd.h>

#include <stdio.h>

int main() {

if (fork() == 0)

execl("/bin/ls", "ls", NULL);

else

wait(NULL);

return 0;

}

**exit()**

The exit() is such a function or one of the system calls that is used to terminate the process. This system call defines that the thread execution is completed especially in the case of a multi-threaded environment. For future reference, the status of the process is captured.

After the use of exit() system call, all the resources used in the process are retrieved by the operating system and then terminate the process. The system call Exit() is equivalent to exit().

#include <stdlib.h>

int main() {

exit(0); // Exit successfully

}

**2. File Management System calls**

**open()**

The **open()** function provides the open system call in C. It is used to open the file for reading, writing, or both. It is also capable of creating the file if it does not exist. It is defined inside **<unistd.h>** header file and the flags that are passed as arguments are defined inside **<fcntl.h>** header file.

**Syntax:**

int open(const char \*pathname, int flags, mode\_t mode);

**close()**

The **close system call** is provided as**close(**) function in C that tells the operating system that you are done with a file descriptor and closes the file pointed by the file descriptor. It is defined inside**<unistd.h>** header file.

**Syntax:**

int close(int fd);

**Read()**

The**read system call**, implemented as the **read()** function reads the specified amount of bytes **cnt** of input into the memory area indicated by **buf** from the file indicated by the file descriptor **fd**. A successful **read()** updates the access time for the file. The **read()** function is also defined inside the **<unistd.h>** header file

**Syntax:**

size\_t read(int fd, void \*buf, size\_t count);

**Write()**

The **write system call**is provided as **write()** function. It writes **cnt**bytes from **buf**to the file or socket associated with **fd**. **cnt**should not be greater than **INT\_MAX**(defined in the limits.h header file). If **cnt**is zero, **write()** simply returns 0 without attempting any other action.

Thewrite() is also defined inside **<unistd.h>** header file.

**Syntax:**

write (fd, buf, cnt);

**3. Device Management System calls**

**read()**

Reads raw data from a device file or standard input.

**Syntax:**

ssize\_t read(int fd, void \*buf, size\_t count);

* fd: File descriptor of the device.
* buf: Buffer to store the data.
* count: Maximum number of bytes to read.

**Example: Read input from terminal**

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

int main() {

char buffer[100];

int fd = open("/dev/tty", O\_RDONLY);

read(fd, buffer, sizeof(buffer));

printf("You typed: %s", buffer);

close(fd);

return 0;

}

**write()**

Sends raw data to a device file or terminal.

**Syntax:**

ssize\_t write(int fd, const void \*buf, size\_t count);

**Example:**

#include <fcntl.h>

#include <unistd.h>

int main() {

int fd = open("/dev/tty", O\_WRONLY);

write(fd, "Hello from device write!\n", 25);

close(fd);

return 0;

}

**ioctl()**

Performs **device-specific control operations** that cannot be expressed using regular system calls. It communicates with device drivers for configuration or querying.

**Syntax:**

int ioctl(int fd, unsigned long request, ...);

**Example:**

#include <stdio.h>

#include <sys/ioctl.h>

#include <unistd.h>

int main() {

struct winsize w;

ioctl(STDOUT\_FILENO, TIOCGWINSZ, &w);

printf("Rows: %d, Cols: %d\n", w.ws\_row, w.ws\_col);

return 0;

}

**select()**

Waits for multiple file descriptors (like device inputs) to become ready for I/O operations. Commonly used for monitoring multiple devices or sockets.

**Syntax:**

int select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);

**4. Network Management System calls**

**socket()**

Creates a new socket (an endpoint for communication).

**Syntax:**

int socket(int domain, int type, int protocol);

* domain: Communication domain (e.g., AF\_INET for IPv4)
* type: Type of service (SOCK\_STREAM for TCP, SOCK\_DGRAM for UDP)
* protocol: Usually set to 0 to choose the default for the given type

**Example:**

int sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

**connect()**

Connects the client socket to the server’s address and port.

**Syntax:**

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

**Example (Client-side):**

#include <stdio.h>

#include <string.h>

#include <arpa/inet.h>

#include <unistd.h>

int main() {

int sock = socket(AF\_INET, SOCK\_STREAM, 0);

struct sockaddr\_in server;

server.sin\_family = AF\_INET;

server.sin\_port = htons(8080);

server.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

if (connect(sock, (struct sockaddr \*)&server, sizeof(server)) < 0) {

perror("Connection failed");

return 1;

}

printf("Connected to server.\n");

close(sock);

return 0;

}

**send()**

Sends data to the connected socket.

**Syntax:**

ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);

* sockfd: Socket file descriptor
* buf: Buffer containing the message
* len: Length of the message
* flags: Usually 0 for default

**Example:**

char \*message = "Hello from client!";

send(sock, message, strlen(message), 0);

**recv()**

Receives data from the connected socket.

**Syntax:**

ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

**Example:**

char buffer[1024] = {0};

recv(sock, buffer, sizeof(buffer), 0);

printf("Server: %s\n", buffer);

**5. System Information Management System calls**

**getpid()**

Returns the **Process ID (PID)** of the calling process.

**Syntax:**

pid\_t getpid(void);

**Example:**

#include <stdio.h>

#include <unistd.h>

int main() {

printf("Current Process ID: %d\n", getpid());

return 0;

}

**getuid()**

Returns the **User ID (UID)** of the user running the process.

**Syntax:**

uid\_t getuid(void);

**Example:**

#include <stdio.h>

#include <unistd.h>

int main() {

printf("User ID: %d\n", getuid());

return 0;

}

**gethostname()**

Retrieves the **network hostname** of the system.

**Syntax:**

int gethostname(char \*name, size\_t len);

* name: Buffer to hold the hostname.
* len: Length of the buffer.

**Example:**

#include <stdio.h>

#include <unistd.h>

int main() {

char hostname[1024];

gethostname(hostname, sizeof(hostname));

printf("Hostname: %s\n", hostname);

return 0;

}

**sysinfo()**

Provides detailed **system statistics** like uptime, RAM usage, load average, etc.

**Syntax:**

int sysinfo(struct sysinfo \*info);

Requires <sys/sysinfo.h>.

**Example:**

#include <stdio.h>

#include <sys/sysinfo.h>

int main() {

struct sysinfo info;

if (sysinfo(&info) == 0) {

printf("Uptime: %ld seconds\n", info.uptime);

printf("Total RAM: %lu MB\n", info.totalram / (1024 \* 1024));

printf("Free RAM: %lu MB\n", info.freeram / (1024 \* 1024));

printf("Number of processes: %u\n", info.procs);

} else {

perror("sysinfo");

}

return 0;

}