| Year:- | Academic Year- 2025-26 | Semester:- |
|---|---|---|

**PART B**

| Roll No:- DO84 | Name:-Somish Jain |
|---|---|
| Class:-BTECH(CE) | Batch:-02 |
| Date of Experiment:-21-07-2025 | Date of Submission:-21-07-2025 |

**Study / Implementation details:**

**Output:**

**Task 2:**

```c
#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>



int main()

{

   int fd;

   char buffer[100];



   fd = open("sample.txt", O_CREAT | O_RDWR, 0644);

   if (fd < 0) {

       perror("open");

       return 1;

   }
```

```
    write(fd, "OS Lab System Call File Write Example.\n", 39);

    lseek(fd, 0, SEEK_SET);

    read(fd, buffer, 39);
    buffer[39] = '\0';

    printf("Read from file: %s", buffer);

    close(fd);
    return 0;
}
```

**Output:**

```
Read from file: OS Lab System Call File Write Example.
```

**Task 3:**

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

int main() {
    pid_t pid = fork();

    if (pid == 0) {
```

```
        // Child process
        printf("Child Process (PID: %d)\n", getpid());
        execlp("echo", "echo", "Executing from Child Process!", NULL);
        perror("exec failed");
        exit(1);
    } else if (pid > 0) {
        // Parent process
        wait(NULL);
        printf("Parent Process (PID: %d) - Child Completed\n", getpid());
    } else {
        perror("fork failed");
        return 1;
    }

    return 0;
}
```

**Output:**

```
Child Process (PID: 5585)
Executing from Child Process!
Parent Process (PID: 5584) - Child Completed
```

**Task 4:**

```
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
```

```c
int main() {

   pid_t pid = getpid();

   pid_t ppid = getppid();

   struct timeval tv;



   gettimeofday(&tv, NULL);



   printf("Current PID: %d\n", pid);

   printf("Parent PID: %d\n", ppid);

   printf("Current Time: %ld seconds and %ld microseconds\n", tv.tv_sec, tv.tv_usec);



   return 0;

}
```

**Output:**

```
Current PID: 5619
Parent PID: 4932
Current Time: 1753079509 seconds and 245648 microseconds
```

**What is a System Call and How is it Different from a Function Call?**

A **system call** allows a user program to request services from the operating system, like reading a file, launching a new process, or accessing hardware components. It acts as a bridge between **user mode** and **kernel mode**.

**Key Differences:**

| Aspect | System Call | Function Call |
|---|---|---|
| **Access Level** | Switches from user space to kernel space | Stays within user space |
| **Purpose** | Performs system-level tasks (e.g., I/O) | Reuses logic or code within programs |
| **Speed** | Slower due to mode switching | Faster, no context switch |
| **Access Rights** | Allows controlled access to hardware resources | Cannot interact with hardware directly |

**What Happens When fork() is Used?**

Calling fork() duplicates the current process, creating a **child process** that shares the same memory and file descriptors as the parent.

- Both processes begin executing from the same instruction after the call.
- The **child** receives a return value of 0.
- The **parent** receives the child's **process ID (PID)**.

Modern operating systems use **copy-on-write**, meaning the memory is not physically copied unless one of the processes modifies it.

**How Does exec() Replace a Process?**

The exec() family of functions loads a **new program** into the current process, replacing everything from code to memory.

- The old program is **completely removed**.
- A **new executable** replaces it in the same process.
- Execution starts from the beginning of the new program (usually main()).

If successful, exec() never returns. If there's an error, it returns -1. It's commonly used right after fork() to run a different program in the child process.

**Difference Between read() and fread()**

| Feature | read() (System Call) | fread() (Library Function) |
|---|---|---|
| **Type** | Low-level, direct OS interaction | High-level, standard library-based |
| **Input** | Uses file descriptors (int) | Uses FILE* pointers |
| **Buffering** | No built-in buffering | Internally buffered |
| **Compatibility** | Platform-specific (mainly Unix/Linux) | Cross-platform friendly |
| **Syntax** | read(fd, buf, size) | fread(ptr, size, count, FILE*) |

Use read() for fine-grained control, and fread() for simpler and buffered file operations.

---

**How Do System Calls Help Abstract Hardware?**

System calls offer a **standard interface** for interacting with hardware, making it unnecessary for applications to handle hardware-specific details directly.

The OS provides consistent functions like open(), read(), and write(), which internally handle all the hardware differences.

**Advantages:**

- Applications can run on different hardware without changes.
- The OS handles safety and sharing of hardware resources.

---

**Purpose of gettimeofday()**

The gettimeofday() function retrieves the **current time** with microsecond accuracy since the Unix Epoch (Jan 1, 1970).

It's useful for:

- Measuring execution time of operations.
- Creating detailed timestamps.
- Monitoring system or program performance.

---

**How Do Parent and Child Processes Communicate?**

In Unix-like systems, parent and child processes communicate using **Inter-Process Communication (IPC)** techniques:

- **Pipes** – Simple one-directional data flow.
- **Shared Memory** – Both processes share the same memory area.
- **Message Queues** – Exchange structured messages.
- **Signals** – For sending simple alerts or instructions.
- **Files or Sockets** – Used for more complex, ongoing communication.

The parent can also use wait() or waitpid() to pause until the child process finishes running.

---