Name : Sahanur Alam Prodhan Sujan

ID : 2110476128

Assignment : Operating System Lab

Department : Computer Science and engineering

1.	**Write a C program to create a main process named 'parent_process' having 3 child processes without any grandchildren processes.** **Trace parent and child processes in the process tree.** **Show that child processes are doing addition, subtraction and multiplication on two variables initialized in the parent_process**	10

1. Creates a parent process named `parent_process`.
2. Forks 3 child processes.
3. Each child performs a separate arithmetic operation (addition, subtraction, multiplication) on two variables initialized by the parent.
4. Displays the **process tree trace** using `getpid()` and `getppid()`.

☑ C Program

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

int main() {
    int a = 10, b = 5;  // Initialized by parent_process
    pid_t pid1, pid2, pid3;

    printf("Parent Process (parent_process) started. PID: %d\n", getpid());

    pid1 = fork();

    if (pid1 == 0) {
        // Child 1: Addition
        printf("Child 1 (Addition) PID: %d, PPID: %d\n", getpid(), getppid());
        printf("Addition: %d + %d = %d\n", a, b, a + b);
        exit(0);
    } else {
        pid2 = fork();
        if (pid2 == 0) {
            // Child 2: Subtraction
            printf("Child 2 (Subtraction) PID: %d, PPID: %d\n", getpid(),
getppid());
            printf("Subtraction: %d - %d = %d\n", a, b, a - b);
            exit(0);
        } else {
            pid3 = fork();
            if (pid3 == 0) {
                // Child 3: Multiplication
                printf("Child 3 (Multiplication) PID: %d, PPID: %d\n", getpid(),
getppid());
```

```
                printf("Multiplication: %d * %d = %d\n", a, b, a * b);
                exit(0);
            } else {
                // Parent process waits for all children
                wait(NULL);
                wait(NULL);
                wait(NULL);
                printf("Parent Process (PID: %d) finished waiting for all
children.\n", getpid());
            }
        }
    }

    return 0;
}
```

🔍 Output Example (on terminal)

```
Parent Process (parent_process) started. PID: 12345
Child 1 (Addition) PID: 12346, PPID: 12345
Addition: 10 + 5 = 15
Child 2 (Subtraction) PID: 12347, PPID: 12345
Subtraction: 10 - 5 = 5
Child 3 (Multiplication) PID: 12348, PPID: 12345
Multiplication: 10 * 5 = 50
Parent Process (PID: 12345) finished waiting for all children.
```

📌 Notes

- The child processes don't fork any grandchildren.
- `getpid()` returns current process ID.
- `getppid()` returns parent process ID.
- `wait(NULL)` ensures the parent waits for each child to finish.

Here are two separate C programs:

- One for creating an **orphan process**
- One for creating a **zombie process**

2.	**Write a C program to create an orphan process.**	05

The orphan is adopted by `init` (or `systemd`), usually PID 1.

Code:

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

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

    if (pid > 0) {
        // Parent process
        printf("Parent process (PID: %d) exiting...\n", getpid());
        exit(0); // Parent exits immediately
    } else if (pid == 0) {
        // Child process
        sleep(5); // Delay to ensure parent exits first
        printf("Child process (PID: %d) now orphan, new PPID: %d\n", getpid(),
getppid());
    } else {
        perror("fork failed");
    }

    return 0;
}
```

Sample Output:

```
Parent process (PID: 1234) exiting...
Child process (PID: 1235) now orphan, new PPID: 1
```

3.	Write a C program to create a zombie process.	05

A zombie process is a child that has terminated but still has an entry in the process table because the parent hasn't called `wait()`.

Code:

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

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

    if (pid > 0) {
        // Parent process
        printf("Parent process (PID: %d) sleeping...\n", getpid());
```

```
        sleep(10); // Parent sleeps without waiting
    } else if (pid == 0) {
        // Child process
        printf("Child process (PID: %d) exiting...\n", getpid());
        exit(0); // Child exits immediately
    } else {
        perror("fork failed");
    }

    return 0;
}
```

How to Check Zombie:

1. Compile and run this program in a terminal.
2. In another terminal, run:

or

You'll see the child process marked as `<defunct>`.

Here are the required C programs for:

4.	**Write a C program to create a main process named 'parent_process' having 3 child processes without any grandchildren processes. Child Processes' names are child_1, child_2, child_3. Trace the position in the process tree.**

This program:

- Creates a `parent_process`.
- Creates 3 child processes: `child_1`, `child_2`, `child_3`.
- Traces the process tree via `getpid()` and `getppid()`.

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

int main() {
    printf("Parent process (parent_process) started. PID: %d\n", getpid());

    for (int i = 1; i <= 3; i++) {
        pid_t pid = fork();

        if (pid == 0) {
```

```
            // Child process
            printf("Child process %d (child_%d) started. PID: %d, PPID: %d\n", i,
i, getpid(), getppid());
            exit(0); // Ensure no grandchildren are created
        } else if (pid < 0) {
            perror("Fork failed");
        }
    }

    // Parent waits for all children
    for (int i = 0; i < 3; i++) {
        wait(NULL);
    }

    printf("Parent process (PID: %d) finished.\n", getpid());
    return 0;
}
```

☑ Sample Output (PIDs will vary each time)

```
Parent process (parent_process) started. PID: 3456
Child process 1 (child_1) started. PID: 3457, PPID: 3456
Child process 2 (child_2) started. PID: 3458, PPID: 3456
Child process 3 (child_3) started. PID: 3459, PPID: 3456
Parent process (PID: 3456) finished.
```

🔍 What Happens Behind the Scenes

- The **parent process** runs a loop 3 times.
- Each time, it calls `fork()`:
  - The **child prints its PID and PPID**, then exits.
  - The **parent continues looping and then waits for all children** using `wait()`.

🧠 Process Tree View

You can visualize the process tree using `pstree -p` (run it while program runs):

```
parent_process(3456)─┬─child_1(3457)
                     ├─child_2(3458)
                     └─child_3(3459)
```

All children are terminated properly (`exit(0)`), and the parent waits using `wait(NULL)`, so **no zombie processes are created**.

5	Write a C program to create a main process named ‘parent_process’ having ‘n’ child processes without any grandchildren processes. Child Processes’ names are child_1, child_2, child_3,....., child_n. Trace the position in the process tree. Number of child processes (n) and name of child processes will be given in the CLI of Linux based systems. Example: $ ./parent_process 3 child_1 child_2 child_3

Here's a **simple and easy C program** that:

- Creates n child processes (no grandchildren),
- Takes n and child process names from **command-line arguments**,
- Prints **PID and PPID** to trace their position in the **process tree**.

☑ 📜 Example Usage:

```
$ ./parent_process 3 child_1 child_2 child_3
```

☑ 🔧 Code:

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

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage: %s <number_of_children> <child_name_1> <child_name_2>
...\n", argv[0]);
        return 1;
    }

    int n = atoi(argv[1]); // Number of child processes

    if (argc != n + 2) {
        printf("Error: You must provide %d child names.\n", n);
        return 1;
    }

    printf("Parent process (parent_process) started. PID: %d\n", getpid());

    for (int i = 0; i < n; i++) {
        pid_t pid = fork();

        if (pid == 0) {
            // Inside child process
```

```
            printf("Child process '%s' created. PID: %d, PPID: %d\n", argv[i + 2],
getpid(), getppid());
            exit(0); // Prevent creating grandchildren
        } else if (pid < 0) {
            perror("Fork failed");
            return 1;
        }

        // Parent waits for each child to finish
        wait(NULL);
    }

    return 0;
}
```

☑ Output Sample:

For:

```
$ ./parent_process 3 child_1 child_2 child_3
```

- Output:

```
Parent process (parent_process) started. PID: 12345
Child process 'child_1' created. PID: 12346, PPID: 12345
Child process 'child_2' created. PID: 12347, PPID: 12345
Child process 'child_3' created. PID: 12348, PPID: 12345
```

🧰 Compile and Run:

```
gcc parent_process.c -o parent_process
./parent_process 3 child_1 child_2 child_3
```

6.	**Write a C program to create a main process named 'parent_process' having 3 child processes without any grandchildren processes. Child Processes' names are child_1, child_2, child_3. Trace the position in the process tree. (same as 4)**	10

This program:

- Creates a `parent_process`.
- Spawns exactly 3 child processes (`child_1`, `child_2`, `child_3`).
- Traces each process's PID and PPID to visualize the process tree.

Code:

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

int main() {
    printf("Parent process (parent_process) started. PID: %d\n", getpid());

    for (int i = 1; i <= 3; i++) {
        pid_t pid = fork();

        if (pid == 0) {
            // Child process
            printf("Child_%d started. PID: %d, PPID: %d\n", i, getpid(),
getppid());
            exit(0); // Prevent grandchildren
        } else if (pid < 0) {
            perror("Fork failed");
        }
    }

    // Parent waits for all children
    for (int i = 0; i < 3; i++) {
        wait(NULL);
    }

    printf("Parent process (parent_process) finished. PID: %d\n", getpid());
    return 0;
}
```

Sample Output:

```
Parent process (parent_process) started. PID: 2024
Child_1 started. PID: 2025, PPID: 2024
Child_2 started. PID: 2026, PPID: 2024
Child_3 started. PID: 2027, PPID: 2024
Parent process (parent_process) finished. PID: 2024
```

You can also verify the process tree using:

```
pstree -p <parent_pid>
```

7.	**Write a C program to analyze the effect of local and global variables on a parent process and a child process.**	**05**

This program shows how global and local variables behave after a `fork()`:

- Global variables are copied — changes in child don't affect the parent.
- Local variables are also copied.

Code:

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

int global_var = 100; // Global variable

int main() {
    int local_var = 50; // Local variable

    pid_t pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    } else if (pid == 0) {
        // Child Process
        global_var += 10;
        local_var += 10;
        printf("Child Process (PID: %d): global_var = %d, local_var = %d\n",
getpid(), global_var, local_var);
    } else {
        // Parent Process
        sleep(1); // Delay to ensure child runs first
        printf("Parent Process (PID: %d): global_var = %d, local_var = %d\n",
getpid(), global_var, local_var);
    }

    return 0;
}
```

Output:

```
Child Process (PID: 2045): global_var = 110, local_var = 60
Parent Process (PID: 2044): global_var = 100, local_var = 50
```

Conclusion:

- **Each process has its own copy** of both local and global variables after `fork()`.

- Changes in one do **not affect** the other.

Here are the C programs for inter-process communication (IPC) using unnamed and named pipes as requested:

8.	**Write a C program to show how two related processes can communicate with each other by an unnamed pipe.**	05

Here's a **simple C program** to demonstrate **inter-process communication (IPC)** using an **unnamed pipe** between a **parent** and **child** process.

☑ Goal:

- The **parent process** sends a message through the **pipe**.
- The **child process** reads that message and prints it.

☑ 💡 Key Concepts:

- `pipe(fd)`: creates a pipe with two file descriptors `fd[0]` (read end) and `fd[1]` (write end).
- `fork()`: creates a child process.
- `write()` and `read()` are used for communication via the pipe.

☑ 🔧 Code:

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

int main() {
    int fd[2]; // fd[0] = read end, fd[1] = write end
    pid_t pid;
    char message[] = "Hello from parent!";
    char buffer[100];

    // Create the pipe
    if (pipe(fd) == -1) {
        perror("Pipe failed");
        return 1;
    }

    // Create child process
    pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        return 1;
```

```
    }

    if (pid == 0) {
        // Child process
        close(fd[1]); // Close write end

        read(fd[0], buffer, sizeof(buffer));
        printf("Child received: %s\n", buffer);

        close(fd[0]); // Close read end
    } else {
        // Parent process
        close(fd[0]); // Close read end

        write(fd[1], message, strlen(message) + 1); // +1 for '\0'
        printf("Parent sent: %s\n", message);

        close(fd[1]); // Close write end
    }

    return 0;
}
```

☑ 📤 Sample Output:

```
Parent sent: Hello from parent!
Child received: Hello from parent!
```

🛠️ Compile & Run:

```
gcc pipe_example.c -o pipe_example
./pipe_example
```

9.	**Write a C program to show how two unrelated processes can communicate with each other by a named pipe.**	05

Two unrelated programs can communicate through a **named pipe** (FIFO). This is a **two-program setup**: writer and reader.

🔸 (A) Writer Program: `writer.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
```

```c
int main() {
    int fd;
    char *fifo = "/tmp/myfifo";
    mkfifo(fifo, 0666);

    fd = open(fifo, O_WRONLY);
    char msg[] = "Hello from writer!";
    write(fd, msg, strlen(msg) + 1);
    close(fd);

    return 0;
}
```

🔸 (B) Reader Program: `reader.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd;
    char buffer[100];
    char *fifo = "/tmp/myfifo";

    fd = open(fifo, O_RDONLY);
    read(fd, buffer, sizeof(buffer));
    printf("Reader received: %s\n", buffer);
    close(fd);

    return 0;
}
```

▶️ Compile & Run:

In **one terminal**:

```
gcc reader.c -o reader
./reader
```

In **another terminal**:

```
gcc writer.c -o writer
./writer
```

☑️ Sample Output:

Terminal 1 (reader):

```
Reader: Received message: Hello from writer process!
```

Terminal 2 (writer):

```
Writer: Message sent.
```

Code	User (Owner)	Group	Others	Symbolic	Meaning
0600	Read/Write	✗	✗	-rw-------	Only owner can access
0666	Read/Write	Read/Write	Read/Write	-rw-rw-rw-	Everyone can access

☑ Use `0600` for **private** pipes.
☑ Use `0666` for **public** communication.

10.	**Write a C program to show how two related processes can communicate with each other by a named pipe.**	**05**

Even related processes (parent & child) can communicate through a **named pipe.**

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>

int main() {
    char *fifo = "/tmp/related_fifo";
    mkfifo(fifo, 0666);

    pid_t pid = fork();

    if (pid == 0) {
        // Child process: Reader
        char buffer[100];
        int fd = open(fifo, O_RDONLY);
        read(fd, buffer, sizeof(buffer));
        printf("Child received: %s\n", buffer);
        close(fd);
    } else {
        // Parent process: Writer
        int fd = open(fifo, O_WRONLY);
        char msg[] = "Hello child, this is parent!";
        write(fd, msg, strlen(msg) + 1);
        close(fd);
        wait(NULL);
    }
```

```
    unlink(fifo); // Remove the FIFO after use
    return 0;
}
```

☑ Output Explanation of the Given Program (Named Pipe between Related Processes)

Your program demonstrates communication between **related processes** (parent and child) using a **named pipe (/tmp/related_fifo)**.

🧪 Sample Output:

When you run the program, you should see something like:

```
Child received: Hello child, this is parent!
```

🧠 How It Works:

1. **Named pipe created** using:
2. **fork()** creates a child process.
3. **Parent process (Writer):**
   - Opens FIFO in **write-only** mode.
   - Sends: `"Hello child, this is parent!"`
4. **Child process (Reader):**
   - Opens FIFO in **read-only** mode.
   - Reads and prints the message.
5. FIFO is removed by:

☑ So the program output is:

```
Child received: Hello child, this is parent!
```

11.	**Write a C program to show how two unrelated processes can communicate with each other by a message queue.**	05

Use **SysV message queue** (`msgget`, `msgsnd`, `msgrcv`) — two unrelated programs: one sends, one receives.

🔸 (A) Sender: `sender.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
```

```c
#include <sys/msg.h>
#include <string.h>

struct msg_buffer {
    long msg_type;
    char msg_text[100];
};

int main() {
    key_t key = ftok("progfile", 65);
    int msgid = msgget(key, 0666 | IPC_CREAT);

    struct msg_buffer message;
    message.msg_type = 1;
    strcpy(message.msg_text, "Hello from sender process!");

    msgsnd(msgid, &message, sizeof(message.msg_text), 0);
    printf("Sender: message sent.\n");

    return 0;
}
```

(B) Receiver: `receiver.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msg_buffer {
    long msg_type;
    char msg_text[100];
};

int main() {
    key_t key = ftok("progfile", 65);
    int msgid = msgget(key, 0666 | IPC_CREAT);

    struct msg_buffer message;
    msgrcv(msgid, &message, sizeof(message.msg_text), 1, 0);

    printf("Receiver: received message = %s\n", message.msg_text);

    msgctl(msgid, IPC_RMID, NULL); // delete queue
    return 0;
}
```

Compile & Run:

```
touch progfile
gcc sender.c -o sender
gcc receiver.c -o receiver

./receiver    # in one terminal
./sender      # in another
```

12.	**Write a C program to show how two related processes can communicate with each other by a message queue.**	**05**

Same concept as above, but one program with `fork()`.

- **Code:**

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

struct msg_buffer {
    long msg_type;
    char msg_text[100];
};

int main() {
    key_t key = ftok("progfile", 65);
    int msgid = msgget(key, 0666 | IPC_CREAT);

    pid_t pid = fork();

    if (pid == 0) {
        // Child process: Receiver
        struct msg_buffer message;
        msgrcv(msgid, &message, sizeof(message.msg_text), 1, 0);
        printf("Child received: %s\n", message.msg_text);
    } else {
        // Parent process: Sender
        struct msg_buffer message;
        message.msg_type = 1;
        strcpy(message.msg_text, "Hello from parent!");

        msgsnd(msgid, &message, sizeof(message.msg_text), 0);
        wait(NULL);
        msgctl(msgid, IPC_RMID, NULL); // delete queue
    }

    return 0;
```

```
}
```

নিচে **related** এবং **unrelated** প্রসেসের মধ্যে **message queue ব্যবহার করে যোগাযোগের পার্থক্যের একটা সহজ টেবিল** দিলাম:

বিষয়	Related Processes	Unrelated Processes
প্রসেসের সম্পর্ক	Parent এবং Child (fork() এর মাধ্যমে)	কোন পারেন্ট-চাইল্ড সম্পর্ক নেই
প্রোগ্রাম সংখ্যা	সাধারণত একটাই প্রোগ্রামে করা যায়	সাধারণত আলাদা দুইটি প্রোগ্রাম লাগে
কোডের অবস্থান	একই প্রোগ্রামে (fork() এর পরে)	আলাদা আলাদা প্রোগ্রামে
রানটাইম শুরু	একসাথে শুরু (parent fork করে child তৈরি করে)	আলাদা আলাদা সময়ে শুরু হতে পারে
Message Queue key ব্যবহারের নিয়ম	একই key শেয়ার করে communication করে	একই key ব্যবহার করে communication করে
Synchronization সহজ	অপেক্ষা (wait) দিয়ে parent অপেক্ষা করতে পারে child এর জন্য	আলাদা প্রোগ্রাম হওয়ায় sync করা একটু জটিল
উদাহরণ	Parent process sends message, child receives	এক প্রোগ্রাম sender, অন্য প্রোগ্রাম receiver

13.	**Write a C program to show how data inconsistency arises in a multi-threaded process.**	05

No mutex is used when multiple threads update a shared counter.

- **Code:**

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

int counter = 0;

void* increment(void* arg) {
    for (int i = 0; i < 1000000; i++) {
        counter++;  // not thread-safe
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Expected: 2000000, Actual: %d\n", counter); // likely less than 2M due
to race condition
    return 0;
```

```
}
```

Expected: 2000000, Actual: 1765423

14.	Write a C program to show how data inconsistency arises in two related processes (e.g., parent & child processes) when they share a memory space.	05

Two related processes (parent & child) use shared memory but don't synchronize.

Code:

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

int main() {
    int shmid = shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | 0666);
    int *shared_var = (int *)shmat(shmid, NULL, 0);
    *shared_var = 0;

    pid_t pid = fork();

    if (pid == 0) {
        // Child process
        for (int i = 0; i < 100000; i++) {
            (*shared_var)++;
        }
        shmdt(shared_var);
        exit(0);
    } else {
        // Parent process
        for (int i = 0; i < 100000; i++) {
            (*shared_var)++;
        }
        wait(NULL);
        printf("Expected: 200000, Actual: %d\n", *shared_var); // likely
inconsistent
        shmdt(shared_var);
        shmctl(shmid, IPC_RMID, NULL);
    }

    return 0;
}
```

Here are the complete solutions for Questions 15–18 involving shared memory, race conditions, and semaphore handling in C:

15.	Write a C program to show how data inconsistency arises in two unrelated processes when they share a memory space.	05

Two unrelated processes (executed separately) share a memory segment without synchronization → **race condition occurs**.

Step 1: Create Shared Memory Key File

```
touch sharedfile
```

(A) `writer1.c`

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

int main() {
    key_t key = ftok("sharedfile", 65);
    int shmid = shmget(key, sizeof(int), 0666 | IPC_CREAT);
    int *counter = (int *)shmat(shmid, NULL, 0);

    for (int i = 0; i < 100000; i++) {
        (*counter)++;
    }

    shmdt(counter);
    return 0;
}
```

(B) `writer2.c`

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

int main() {
    key_t key = ftok("sharedfile", 65);
    int shmid = shmget(key, sizeof(int), 0666);
    int *counter = (int *)shmat(shmid, NULL, 0);

    for (int i = 0; i < 100000; i++) {
```

```
        (*counter)++;
    }

    printf("Final value (should be 200000): %d\n", *counter);
    shmdt(counter);
    shmctl(shmid, IPC_RMID, NULL); // cleanup
    return 0;
}
```

▶️ Compile & Run:

```
gcc writer1.c -o writer1
gcc writer2.c -o writer2

./writer1 & ./writer2
```

You'll usually get a value less than 200000 due to **race condition**.

16.	Write a C program to handle racing situations in a multi-threaded process.	05

Use `pthread_mutex_t` to protect critical section.

🔸 Code:

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

int counter = 0;
pthread_mutex_t lock;

void* increment(void* arg) {
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock, NULL);

    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
```

```
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Counter: %d\n", counter); // Should be exactly 2,000,000

    pthread_mutex_destroy(&lock);
    return 0;
}
```

17.	Write a C program to handle racing situations in multiple unrelated processes. 05	

Use a **System V semaphore** to synchronize access to shared memory.

Code (Single file, two forks)

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

union semun {
    int val;
};

void sem_wait(int semid) {
    struct sembuf sb = {0, -1, 0};
    semop(semid, &sb, 1);
}

void sem_signal(int semid) {
    struct sembuf sb = {0, 1, 0};
    semop(semid, &sb, 1);
}

int main() {
    key_t shmkey = ftok("sharedfile", 65);
    key_t semkey = ftok("sharedfile", 75);

    int shmid = shmget(shmkey, sizeof(int), IPC_CREAT | 0666);
    int semid = semget(semkey, 1, IPC_CREAT | 0666);

    union semun sem_val;
    sem_val.val = 1;
    semctl(semid, 0, SETVAL, sem_val);
```

```
    int *counter = (int *)shmat(shmid, NULL, 0);
    *counter = 0;

    pid_t pid = fork();
    for (int k = 0; k < 2; k++) {
        if (pid == 0 || k == 1) {
            for (int i = 0; i < 100000; i++) {
                sem_wait(semid);
                (*counter)++;
                sem_signal(semid);
            }
            shmdt(counter);
            exit(0);
        }
    }

    wait(NULL);
    wait(NULL);

    printf("Final counter: %d\n", *counter);
    shmdt(counter);
    shmctl(shmid, IPC_RMID, NULL);
    semctl(semid, 0, IPC_RMID);
    return 0;
}
```

18.	Write a C program to show the usage of a Counter semaphore.	05

This simulates a resource pool using a counting semaphore.

Code (3 threads, only 2 allowed in at once)

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t count_sem;

void* access_resource(void* arg) {
    int id = *(int*)arg;
    sem_wait(&count_sem);
    printf("Thread %d accessing resource\n", id);
    sleep(2); // Simulate work
    printf("Thread %d releasing resource\n", id);
    sem_post(&count_sem);
    return NULL;
```

```
}

int main() {
    pthread_t threads[3];
    int ids[3] = {1, 2, 3};

    sem_init(&count_sem, 0, 2); // Only 2 threads can enter at once

    for (int i = 0; i < 3; i++)
        pthread_create(&threads[i], NULL, access_resource, &ids[i]);

    for (int i = 0; i < 3; i++)
        pthread_join(threads[i], NULL);

    sem_destroy(&count_sem);
    return 0;
}
```

Here are detailed C programs for tasks 19, 20, and 21, including explanations for each part:

19.	Write a C program for creating a multi-threaded process and check: A. If one thread in the process calls fork(), does the new process duplicate all threads, or is the new process single-threaded? B. If a thread invokes the exec() system call, does it replace the entire code of the process? C. If exec() is called immediately after forking, will all threads be duplicated?	15

This program checks:

- **(A)**: If `fork()` duplicates all threads → **No**, only calling thread is duplicated.
- **(B)**: If `exec()` in one thread replaces the entire process → **Yes**, whole process image is replaced.
- **(C)**: If `exec()` is called after `fork()` → only **child** process executes new image.

Code:

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

void* thread_func(void* arg) {
    printf("Thread %ld started. PID: %d\n", (long)arg, getpid());

    if ((long)arg == 1) {
```

```
        sleep(1);
        printf("Thread %ld calling fork()\n", (long)arg);
        pid_t pid = fork();

        if (pid == 0) {
            // Child process
            printf("Child after fork: PID = %d, PPID = %d\n", getpid(), getppid());
            printf("Child now calling exec()...\n");
            execlp("ls", "ls", "-l", NULL);
            perror("exec failed");
            exit(1);
        } else {
            wait(NULL);
        }
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t t1, t2;

    printf("Main thread: PID = %d\n", getpid());

    pthread_create(&t1, NULL, thread_func, (void*)1);
    pthread_create(&t2, NULL, thread_func, (void*)2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Main thread ending.\n");
    return 0;
}
```

📝 Observations:

- Only **calling thread** exists in forked child.
- If `exec()` is called in **any thread**, it **replaces the entire process image**.
- `exec()` **after fork()** runs in **child only**, no threads duplicated.

20.	Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers 90 81 78 95 79 72 85 The program will report A. The average value is 82 B. The minimum value is 72 C. The maximum value is 95 The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited.	15

Input: numbers via CLI
Output: average, max, min (calculated by separate threads)

Code:

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

int *numbers;
int count;
float average;
int maximum;
int minimum;

void* calc_average(void* arg) {
    int sum = 0;
    for (int i = 0; i < count; i++) sum += numbers[i];
    average = (float)sum / count;
    pthread_exit(NULL);
}

void* calc_maximum(void* arg) {
    maximum = numbers[0];
    for (int i = 1; i < count; i++)
        if (numbers[i] > maximum) maximum = numbers[i];
    pthread_exit(NULL);
}

void* calc_minimum(void* arg) {
    minimum = numbers[0];
    for (int i = 1; i < count; i++)
```

```
        if (numbers[i] < minimum) minimum = numbers[i];
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s num1 num2 ...\n", argv[0]);
        return 1;
    }

    count = argc - 1;
    numbers = malloc(count * sizeof(int));
    for (int i = 0; i < count; i++)
        numbers[i] = atoi(argv[i + 1]);

    pthread_t t_avg, t_max, t_min;
    pthread_create(&t_avg, NULL, calc_average, NULL);
    pthread_create(&t_max, NULL, calc_maximum, NULL);
    pthread_create(&t_min, NULL, calc_minimum, NULL);

    pthread_join(t_avg, NULL);
    pthread_join(t_max, NULL);
    pthread_join(t_min, NULL);

    printf("Average: %.2f\n", average);
    printf("Maximum: %d\n", maximum);
    printf("Minimum: %d\n", minimum);

    free(numbers);
    return 0;
}
```

Example Run:

```
./stat_calc 90 81 78 95 79 72 85
```

Output:

```
Average: 82.86
Maximum: 95
Minimum: 72
```

21.	The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, ... Formally, it can be expressed as: fib0 = 0 fib1 = 1 fibn = fibn−1 + fibn−2 Write a multithreaded program that generates the Fibonacci sequence. This program should work as follows: On the command line, the user will enter the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, the parent thread will have to wait for the child thread to finish.

and prints them.

- ## Code:

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

int *fib;
int n;

void* generate_fib(void* arg) {
    if (n > 0) fib[0] = 0;
    if (n > 1) fib[1] = 1;

    for (int i = 2; i < n; i++) {
        fib[i] = fib[i-1] + fib[i-2];
    }
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    if (argc != 2 || (n = atoi(argv[1])) <= 0) {
        printf("Usage: %s <number_of_terms>\n", argv[0]);
        return 1;
    }

    fib = malloc(n * sizeof(int));

    pthread_t t;
    pthread_create(&t, NULL, generate_fib, NULL);
    pthread_join(t, NULL);

    printf("Fibonacci Sequence:\n");
```

```
    for (int i = 0; i < n; i++)
        printf("%d ", fib[i]);
    printf("\n");

    free(fib);
    return 0;
}
```

Example Run:

```
./fib 10
```

Output:

```
Fibonacci Sequence:
0 1 1 2 3 5 8 13 21 34
```

Here's a complete explanation and example implementation of a basic **server-client model** using sockets in C, plus an explanation of the behaviors you experience with different server designs:

22.	Implement a server-client model to provide services to client processes running in different terminals. Explain what you experience when you: ● Server process being a single threaded process tries to provide services to multiple client processes. ● Server process being a multi-threaded process tries to provide services to multiple client processes. ● Server process being a single threaded process tries to provide services to multiple client processes with multiple child processes.

We'll implement a simple TCP server and client programs. The server listens on a port and replies to clients.

Single-threaded Server (handles clients one by one)

Server code (`server_single.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
```

```
int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};
    char *response = "Hello from single-threaded server";

    // Creating socket file descriptor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Bind
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address))<0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, 3) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    printf("Single-threaded server listening on port %d...\n", PORT);

    while (1) {
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                        (socklen_t*)&addrlen))<0) {
            perror("accept");
            exit(EXIT_FAILURE);
        }

        read(new_socket, buffer, 1024);
        printf("Received from client: %s\n", buffer);
        send(new_socket, response, strlen(response), 0);
        close(new_socket);  // Close client socket
    }
    return 0;
}
```

Client code (`client.c`):

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

```c
#include <arpa/inet.h>

#define PORT 8080

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char *message = "Hello from client";
    char buffer[1024] = {0};

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\n Socket creation error \n");
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0) {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        printf("\nConnection Failed \n");
        return -1;
    }

    send(sock, message, strlen(message), 0);
    read(sock, buffer, 1024);
    printf("Message from server: %s\n", buffer);
    close(sock);
    return 0;
}
```

How to run:

- Open one terminal, compile and run server:

```
gcc server_single.c -o server_single
./server_single
```

- Open other terminal(s), compile and run client multiple times:

```
gcc client.c -o client
./client
```

What you will experience:

1. Single-threaded server serving multiple clients

- The server **handles clients sequentially**.
- While serving one client, other clients **must wait**.
- If a client takes long or never disconnects, others are blocked.
- Not scalable for many clients.

---

2. Multi-threaded server serving multiple clients

- Server creates a **new thread for each client**.
- Multiple clients are served **concurrently**.
- Much better scalability.
- Requires thread synchronization if shared resources are used.

---

3. Single-threaded server using multiple child processes

- Server forks a child for each client (`fork()` instead of thread).
- Each child handles one client independently.
- Similar scalability as multi-threaded.
- Processes have separate memory spaces — safer but more resource-heavy.

Bonus: Simple multi-process server sketch

```
// On accept(), fork a child to handle client, parent goes back to accept().
if ((new_socket = accept(...)) < 0) { ... }

pid_t pid = fork();
if (pid == 0) {
    // child process
    close(server_fd);
    // handle client on new_socket
    exit(0);
} else {
    close(new_socket);
}
```