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

Code:

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

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

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

    pid1 = fork();
    if (pid1 == 0) {
        // child_1 - Addition
        printf("Addition => Name: child_1, PID: %d, PPID: %d\n", getpid(), getppid());
        printf("Result: %d + %d = %d\n", a, b, a + b);
        exit(0);
    }
    wait(NULL);

    pid2 = fork();
    if (pid2 == 0) {
        // child_2 - Subtraction
        printf("Subtraction => Name: child_2, PID: %d, PPID: %d\n", getpid(), getppid());
        printf("Result: %d - %d = %d\n", a, b, a - b);
        exit(0);
    }
    wait(NULL);

    pid3 = fork();
    if (pid3 == 0) {
        // child_3 - Multiplication
        printf("Multiplication => Name: child_3, PID: %d, PPID: %d\n", getpid(), getppid());
        printf("Result: %d * %d = %d\n", a, b, a * b);
        exit(0);
    }
    wait(NULL);

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

    return 0;
}
```

Output:

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc parent_process.c -o parent_process && ./parent_process
Parent process started (Name: parent_process, PID: 17657)
Addition => Name: child_1, PID: 17658, PPID: 17657
Result: 10 + 5 = 15
Subtraction => Name: child_2, PID: 17659, PPID: 17657
Result: 10 - 5 = 5
Multiplication => Name: child_3, PID: 17660, PPID: 17657
Result: 10 * 5 = 50
Parent process (PID: 17657) finished.
```

```
          ├─gnome-terminal-(16985)─┬─bash(16995)───parent_process(28264)─┬─parent_process(28265)
          │                        │                                     ├─parent_process(28266)
          │                        │                                     └─parent_process(28267)
```

[To see the process in the process tree, we have to keep the program in sleep for a while before calling wait() from the parent so that the program doesn't terminate and we can get enough time to see the process. To show the process use commands: "pstree -p", "pstree -p <processId>", "ps -elf", "top"]

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

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

int main() {
    pid_t pid = fork(); // Create a child process

    if (pid == 0) {
        printf("Child (PID: %d, PPID: %d) starts\n", getpid(), getppid());
        sleep(3); //parent exits during this time and Child process becomes orphan
        printf("Orphaned Child (PID: %d) now has new PPID = %d (init/systemd)\n",
               getpid(), getppid());
    }
    else if (pid > 0) {
        // Parent exits immediately
        printf("Parent (PID: %d) exiting\n", getpid());
        exit(0);
    }
    else {
        perror("fork failed");
        exit(1);
    }

    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q2.Orphan_process.c -o orphan_process && ./orphan_process
Parent (PID: 21938) exiting
Child (PID: 21939, PPID: 21938) starts
saidul@saidul-Latitude-7400:~/Desktop/os$ Orphaned Child (PID: 21939) now has new PPID = 1911 (init/systemd)
```

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

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

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

    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }

    if (pid == 0) {
        // Child exits
        printf("Child (PID: %d) exiting.\n", getpid());
        exit(0);
    } else {
        // Parent sleeps without calling wait()
        printf("Parent (PID: %d) sleeping... while child(PID: %d) becoming zombie\n", getpid(), pid);
        sleep(20);
        printf("Parent (PID: %d) terminates\n", getpid());
    }

    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q3.zombie_process.c -o zombie_process && ./zombie_process
Parent (PID: 25834) sleeping... while child(PID: 25835) becoming zombie
Child (PID: 25835) exiting.
Parent (PID: 25834) terminates
```

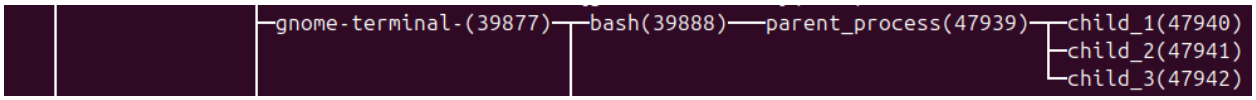
```
0 S saidul     25834   16995  0  80   0 -   670 hrtime 18:16 pts/0    00:00:00 ./zombie_process
1 Z saidul     25835   25834  0  80   0 -     0 -      18:16 pts/0    00:00:00 [zombie_process] <defunct>
```

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.

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

int main() {
    printf("Parent (PID:%d) starts\n", getpid());

    for (int i = 1; i <= 3; i++) {
        pid_t pid = fork();
        if (pid == 0) {
            if (i == 1) prctl(PR_SET_NAME, "child_1");
            else if (i == 2) prctl(PR_SET_NAME, "child_2");
            else if (i == 3) prctl(PR_SET_NAME, "child_3");

            printf("Child_%d (PID:%d, Parent:%d)\n", i, getpid(), getppid());
```

```
            exit(0);
        }
        wait(NULL);  // Parent waits for child
    }

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

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q4.parent_process.c -o parent_process && ./parent_process
Parent (PID:47939) starts
Child_1 (PID:47940, Parent:47939)
Child_2 (PID:47941, Parent:47939)
Child_3 (PID:47942, Parent:47939)
Parent (PID:47939) ends
saidul@saidul-Latitude-7400:~/Desktop/os$
```



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

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

int main(int argc, char *argv[]) {
    int n = atoi(argv[1]);
    printf("Parent (PID=%d) started\n", getpid());

    for (int i = 1; i <= n; i++) {
        pid_t pid = fork();
        if (pid == 0) {
            prctl(PR_SET_NAME, argv[i+1]);

            printf("%s (PID:%d, Parent:%d)\n", argv[i+1], getpid(), getppid());
            exit(0);
        }
        wait(NULL);  // Parent waits for child
    }
    printf("Parent (PID=%d) exiting\n", getpid());
    return 0;
}
```

==========

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

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

int main(int argc, char* argv[]) {
  int n = (int)(argv[1][0]-'0');
  for(int i = 0; i < n; i++) {
    pid_t pid = fork();
    if(pid == 0) {
      printf("Child Process %s: pid = %d, ppid = %d\n", argv[i+2], getpid(), getppid());
      prctl(PR_SET_NAME, argv[i+2]);
      exit(0);
    }
  }
  sleep(20);
  printf("Parent pid = %d is exiting...\n", getpid());
  for(int i = 0; i < n; i++) wait(0);
  return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q5.parent_process.c -o parent_process
saidul@saidul-Latitude-7400:~/Desktop/os$ ./parent_process 3 child_1 child_2 child_3
Parent (PID=27967) started
child_1 (PID:27968, Parent:27967)
child_3 (PID:27970, Parent:27967)
child_2 (PID:27969, Parent:27967)
Parent (PID=27967) exiting
```

```
          ├─gnome-terminal-(26449)─┬─bash(26459)───parent_process(27967)─┬─child_1(27968)
          │                        │                                     ├─child_2(27969)
          │                        │                                     └─child_3(27970)
```

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

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

int global_var = 10;

int main() {
    int local_var = 20;

    printf("Initial value: global_var = %d, local_var = %d\n", global_var, local_var);

    pid_t pid = fork();

    if (pid == 0) {  // Child process
```

```
        global_var++;  // Modify global
        local_var++;   // Modify local
        printf("Variables modified by child\n");
        printf("Child process (PID: %d) sees: global_var = %d, local_var = %d\n", getpid(), global_var,
local_var);
        exit(0);  // Exit child
    }
    else if (pid > 0) {  // Parent process
        wait(NULL);  // Wait for child
        printf("Parent process (PID: %d) sees: global_var = %d, local_var = %d\n", getpid(), global_var,
local_var);
    }
    else {
        perror("fork() failed");
        exit(1);
    }

    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q7.variables_effect.c -o q7.variables_effect
saidul@saidul-Latitude-7400:~/Desktop/os$ ./q7.variables_effect
Initial value: global_var = 10, local_var = 20
Variables modified by child
Child process (PID: 37855) sees: global_var = 11, local_var = 21
Parent process (PID: 37854) sees: global_var = 10, local_var = 20
```

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

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

int main() {
    int pipefd[2];  // pipefd[0] = read, pipefd[1] = write
    char buffer[100];

    // Step 1: Create the pipe
    if (pipe(pipefd) == -1) {
        perror("pipe() failed");
        return 1;
    }

    pid_t pid = fork();

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

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

```
        close(pipefd[0]);
    }
    else if (pid > 0) {  // Parent process (Writer)
        close(pipefd[0]);  // Close unused read end

        char *msg = "Hello from parent!";
        write(pipefd[1], msg, strlen(msg) + 1);
        printf("Parent sent: %s\n", msg);

        close(pipefd[1]);
        wait(NULL);  // Wait for child
    }
    else {
        perror("fork() failed");
        return 1;
    }

    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q8.unnamed_pipe.c -o unnamed_pipe
saidul@saidul-Latitude-7400:~/Desktop/os$ ./unnamed_pipe
Parent sent: Hello from parent!
Child received: Hello from parent!
```

==========

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

int main() {
  int fd[2];
  pipe(fd);
  pid_t pid = fork();
  if(pid == 0) {
    close(fd[1]);
    char buffer[100];
    read(fd[0], buffer, sizeof(buffer));
    close(fd[0]);
    printf("Child received: %s\n", buffer);
    exit(0);
  }
  close(fd[0]);
  char* msg = "Hello from Parent\n";
  write(fd[1], msg, strlen(msg));
  close(fd[1]);
  printf("Parent writes: %s\n", msg);
  wait(0);
  return 0;
}
```

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

Writer.c:

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

int main() {
    char *fifo_name = "myfifo";

    // Create FIFO with read/write permissions
    mkfifo(fifo_name, 0666);
    printf("Writer: Waiting for reader...\n");

    // Open FIFO for writing (blocks until reader opens)
    int fd = open(fifo_name, O_WRONLY);

    // Write data
    char msg[] = "Hello from writer";
    write(fd, msg, sizeof(msg));
    printf("Message sent\n");
    close(fd);
    return 0;
}
```

Reader.c:

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

int main() {
    const char *fifo_name = "myfifo";

    printf("Reader: Waiting for writer...\n");

    // Open FIFO for reading (blocks until writer opens)
    int fd = open(fifo_name, O_RDONLY);

    // Read data
    char buffer[100];
    read(fd, buffer, sizeof(buffer));
    printf("Received: %s\n", buffer);

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

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q9.writer_named_pipe.c -o writer
saidul@saidul-Latitude-7400:~/Desktop/os$ ./writer
Writer: Waiting for reader...
Message sent
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q9.reader_named_pipe.c -o reader
saidul@saidul-Latitude-7400:~/Desktop/os$ ./reader
Reader: Waiting for writer...
Received: Hello from writer
```

==========

Sender

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

int main() {
  char* pipe_name = "myfifo";
  mkfifo(pipe_name, 0666);
  int fd = open(pipe_name, O_WRONLY);
  char* msg = "Message from writter\n";
  write(fd, msg, strlen(msg));
  printf("Sender: sent %s\n", msg);
  close(fd);
  return 0;
}
```

Receiver

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

int main() {
  char* pipe_name = "myfifo";
  int fd = open(pipe_name, O_RDONLY);
  char buffer[100];
  read(fd, buffer, sizeof(buffer));
  printf("Received: %s\n", buffer);
  close(fd);
  return 0;
}
```

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

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#define FIFO_NAME "myfifo"

int main() {
    pid_t pid;
    mkfifo(FIFO_NAME, 0666);

    pid = fork();
    if (pid == 0) { // Child process - Reader
```

```
        int fd = open(FIFO_NAME, O_RDONLY);
        char buf[100];
        read(fd, buf, sizeof(buf));
        printf("Child received: %s\n", buf);
        close(fd);
    } else { // Parent process - Writer
        int fd = open(FIFO_NAME, O_WRONLY);
        char msg[] = "Hello from parent!";
        write(fd, msg, sizeof(msg));
        printf("Parent sent: %s\n", msg);
        close(fd);
    }
    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q10.related_process_named_pipe.c -o related_process_named_pipe
saidul@saidul-Latitude-7400:~/Desktop/os$ ./related_process_named_pipe
Parent sent: Hello from parent!
Child received: Hello from parent!
```

==========

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

int main() {
  char* pipe_name = "myfifo";
  mkfifo(pipe_name, 0666);
  pid_t pid = fork();
  if(pid == 0) {
    char buffer[100];
    int fd = open(pipe_name, O_RDONLY);
    read(fd, buffer, sizeof(buffer));
    close(fd);
    printf("Child: received \"%s\"\n", buffer);
    exit(0);
  }
  char* msg = "Hello from parent";
  int fd = open(pipe_name, O_WRONLY);
  write(fd, msg, strlen(msg));
  printf("Parent: sent \"%s\"\n", msg);
  close(fd);
  wait(0);
  return 0;
}
```

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

Sender:

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

#define MAX_MSG_SIZE 100

// Message structure (must start with long mtype)
struct msg_buffer {
    long mtype;
    char mtext[MAX_MSG_SIZE];
};

int main() {
    key_t key = ftok("msg_queue", 65);  // Generate unique key
    int msgid = msgget(key, 0666 | IPC_CREAT);  // Create/get queue

    struct msg_buffer message;
    message.mtype = 1;  // Message type (must be > 0)
    strcpy(message.mtext, "Hello from sender!");

    // Send message (blocks if queue is full)
    msgsnd(msgid, &message, sizeof(message.mtext), 0);
    printf("Sender: Message sent\n");

    return 0;
}
```

Receiver:

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

#define MAX_MSG_SIZE 100

struct msg_buffer {
    long mtype;
    char mtext[MAX_MSG_SIZE];
};

int main() {
    key_t key = ftok("msg_queue", 65);  // Same key as sender
    int msgid = msgget(key, 0666);  // Get existing queue

    struct msg_buffer message;

    // Receive message (blocks until message arrives)
    msgrcv(msgid, &message, sizeof(message.mtext), 1, 0);
    printf("Receiver: Message received: %s\n", message.mtext);

    // Cleanup (optional)
    msgctl(msgid, IPC_RMID, NULL);  // Remove queue
    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q11.sender_unrelated_process_msg_queue.c -o sender
saidul@saidul-Latitude-7400:~/Desktop/os$ ./sender
Sender: Message sent
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q11.receiver_unrelated_process_msg_queue.c -o receiver
saidul@saidul-Latitude-7400:~/Desktop/os$ ./receiver
Receiver: Message received: Hello from sender!
```

==========

Sender:

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

struct msgstrct {
  long mstype;
  char msg[100];
};
int main() {
  key_t key = ftok(".", 6);
  int msgid = msgget(key, 0666 | IPC_CREAT);

  struct msgstrct message;
  message.mstype = 1;
  strcpy(message.msg, "Hello from sender");

  msgsnd(msgid, &message, strlen(message.msg)+1, 0);
  printf("Message sent\n");
  return 0;
}
```

Receiver

```
#include<stdio.h>
#include<sys/msg.h>

struct msgstrct {
  long msgtype;
  char msg[100];
};
int main() {
  key_t key = ftok(".", 6);
  int msgid = msgget(key, 0666);

  struct msgstrct message;
  msgrcv(msgid, &message, sizeof(message.msg), 1, 0);
  printf("Received: %s\n", message.msg);
  return 0;
}
```

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

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

```c
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX_MSG_SIZE 100

struct msg_buffer {
    long mtype;
    char mtext[MAX_MSG_SIZE];
};

int main() {
    key_t key = ftok("progfile", 65); // Generate key
    int msgid = msgget(key, 0666 | IPC_CREAT); // Create message queue

    pid_t pid = fork();

    if (pid == 0) { // Child process (Receiver)
        struct msg_buffer message;

        // Block until message arrives
        msgrcv(msgid, &message, sizeof(message.mtext), 1, 0);
        printf("Child received: %s\n", message.mtext);

        //Send reply (bidirectional communication)
        message.mtype = 2;
        strcpy(message.mtext, "Reply from child");
        msgsnd(msgid, &message, sizeof(message.mtext), 0);
    }
    else { // Parent process (Sender)
        struct msg_buffer message;
        message.mtype = 1; // Message type for receiver
        strcpy(message.mtext, "Hello from parent");

        // Send message
        msgsnd(msgid, &message, sizeof(message.mtext), 0);
        printf("Parent sent: %s\n", message.mtext);

        // Wait for child's reply
        msgrcv(msgid, &message, sizeof(message.mtext), 2, 0);
        printf("Parent got reply: %s\n", message.mtext);

        wait(NULL); // Reap child
        msgctl(msgid, IPC_RMID, NULL); // Destroy queue
    }
    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q12.related_process_msg_queue.c -o related_msgq
saidul@saidul-Latitude-7400:~/Desktop/os$ ./related_msgq
Parent sent: Hello from parent
Child received: Hello from parent
Parent got reply: Reply from child
```

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

struct msgstrct {
  long msgtype;
  char msg[100];
};
int main() {
  key_t key = ftok(".", 6);
  int msgid = msgget(key, 0666 | IPC_CREAT);

  pid_t pid = fork();
  if(pid == 0) {
    struct msgstrct message;
    message.msgtype = 1;
    strcpy(message.msg, "Hello from child");
    msgsnd(msgid, &message, strlen(message.msg)+1, 0);
    printf("Child: Message sent\n");
    exit(0);
  }
  wait(0);
  struct msgstrct message;
  msgrcv(msgid, &message, sizeof(message.msg), 1, 0);
  printf("Parent: Received message = %s\n", message.msg);
  return 0;
}
```

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

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

int counter = 0;  // Shared global variable

void *increment(void *arg) {
    for (int i = 0; i < 100000; i++) {
        counter++;  // Critical section (race condition!)
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    // Create two threads
    pthread_create(&thread1, NULL, increment, NULL);
    pthread_create(&thread2, NULL, increment, NULL);

    // Wait for threads to finish
```

```
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // Expected result: 200000, but actual result will be inconsistent
    printf("Final counter value: %d (expected: 200000)\n", counter);

    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q13.data_inconsistency_multi_thread.c -o data_in
saidul@saidul-Latitude-7400:~/Desktop/os$ ./data_inconsistency
Final counter value: 105866 (expected: 200000)
```

==========

```
#include<stdio.h>
#include<pthread.h>
#define TOTAL_INCREMENT 100000

int counter = 0;
void *increment(void *args) {
  for(int i = 0; i < TOTAL_INCREMENT; i++) counter++;
  return NULL;
}
int main() {
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, increment, NULL);
  pthread_create(&thread2, NULL, increment, NULL);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  printf("Actual count = %d(Expected = %d)\n", counter, TOTAL_INCREMENT*2);
  return 0;
}
```

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.

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

int main() {
    // Create a shared memory space
    int *counter = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);
    if (counter == MAP_FAILED) {
        perror("mmap failed");
        exit(1);
```

```
    }

    *counter = 0; // Initialize the shared counter

    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(1);
    } else if (pid == 0) {
        // Child process
        for (int i = 0; i < 100000; i++) {
            (*counter)++;
        }
        exit(0);
    } else {
        // Parent process
        for (int i = 0; i < 100000; i++) {
            (*counter)++;
        }

        wait(NULL); // Wait for child to finish

        printf("Final counter value: %d (Expected: 200000)\n", *counter);
    }

    munmap(counter, sizeof(int)); // Clean up shared memory

    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q14.data_incosistency_related_process.c -o data_inconsistency_related_process
saidul@saidul-Latitude-7400:~/Desktop/os$ ./data_inconsistency_related_process
Final counter value: 151821 (Expected: 200000)
```

==========

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

#define TOTAL_INCREMENT 100000
int main() {
  int shmid = shmget(1234, sizeof(int), IPC_CREAT | 0666);
  int *counter = (int*)shmat(shmid, NULL, 0);
  *counter = 0;

  pid_t pid = fork();
  if(pid == 0) {
    for(int i = 0; i < TOTAL_INCREMENT; i++) {
      *counter = *counter+1;
    }
    exit(0);
  }
  for(int i = 0; i < TOTAL_INCREMENT; i++) {
```

```
    *counter = *counter+1;
  }
  wait(0);
  printf("Final counter value = %d(Expected = %d)\n", *counter, TOTAL_INCREMENT*2);
  return 0;
}
```

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

Process 1:

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

int main() {
    int i;
    int shmid;
    int *counter;

    // Create shared memory
    shmid = shmget(1234, sizeof(int), IPC_CREAT | 0666);
    if (shmid == -1) {
        printf("Shared memory creation failed.\n");
        exit(1);
    }

    // Attach to shared memory
    counter = (int *) shmat(shmid, NULL, 0);
    if (counter == (void *) -1) {
        printf("Memory attach failed.\n");
        exit(1);
    }

    *counter = 0; // Initialize counter

    // Increment the shared counter
    for (i = 0; i < 50000; i++) {
        *counter = *counter + 1;
        usleep(100); // Slow down to allow overlap
    }

    printf("Process 1 completed. Counter = %d\n", *counter);

    // Detach shared memory
    shmdt(counter);

    return 0;
}
```

Process 2:

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

int main() {
    int i;
    int shmid;
    int *counter;

    // Access existing shared memory
    shmid = shmget(1234, sizeof(int), 0666);
    if (shmid == -1) {
        printf("Shared memory access failed.\n");
        exit(1);
    }

    // Attach to shared memory
    counter = (int *) shmat(shmid, NULL, 0);
    if (counter == (void *) -1) {
        printf("Memory attach failed.\n");
        exit(1);
    }

    // Increment the shared counter
    for (i = 0; i < 50000; i++) {
        *counter = *counter + 1;
        usleep(100); // Slow down to allow overlap
    }

    printf("Process 2 completed. Counter = %d\n", *counter);

    // Detach shared memory
    shmdt(counter);

    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q15.process1_data_inconsistency_unrelated_process_shared_memory.c -o process1
saidul@saidul-Latitude-7400:~/Desktop/os$ ./process1
Process 1 completed. Counter = 73339
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q15.process2_data_inconsistency_unrelated_process_shared_memory.c -o process2
saidul@saidul-Latitude-7400:~/Desktop/os$ ./process2
Process 2 completed. Counter = 99883
```

==========

Process 1:

```c
#include<stdio.h>
#include<unistd.h>
#include<sys/shm.h>
#define TOTAL_INCREMENT 100000

int main() {
  int shmid = shmget(1234, sizeof(int), IPC_CREAT | 0666);
  int *counter = (int*)shmat(shmid, NULL, 0);
```

```
  *counter = 0;
  for(int i = 0; i < TOTAL_INCREMENT; i++) {
    *counter = *counter+1;
    usleep(10);
  }
  printf("Counter after p1 = %d\n", *counter);
  return 0;
}
```

process 2 is same but counter = 0 will not exist as it will reset the value.

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

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

#define NUM_ITERATIONS 500000
#define NUM_THREADS 2

int counter = 0;              // Shared counter
pthread_mutex_t lock;         // Mutex lock

void *increment_counter(void *arg) {
    int i;

    for (i = 0; i < NUM_ITERATIONS; i++) {
        // Lock the mutex before updating the counter
        pthread_mutex_lock(&lock);

        counter = counter + 1;

        // Unlock the mutex after updating the counter
        pthread_mutex_unlock(&lock);
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int i;

    // Initialize the mutex
    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("Mutex initialization failed.\n");
        return 1;
    }

    // Create threads
    for (i = 0; i < NUM_THREADS; i++) {
        if (pthread_create(&threads[i], NULL, increment_counter, NULL) != 0) {
            printf("Thread creation failed.\n");
            return 1;
        }
```

```
    }

    // Wait for all threads to finish
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Final counter value: %d\n", counter);
    printf("Expected counter value: %d\n", NUM_THREADS * NUM_ITERATIONS);

    // Destroy the mutex
    pthread_mutex_destroy(&lock);

    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q16.multi_threaded_race_condition_handle.c -o multi_threaded_race
saidul@saidul-Latitude-7400:~/Desktop/os$ ./multi_threaded_race
Final counter value: 1000000
Expected counter value: 1000000
```

==========

```
#include<stdio.h>
#include<pthread.h>
#define TOTAL_INCREMENT 100000

pthread_mutex_t lock;
int counter = 0;
void *increment(void *args) {
  for(int i = 0; i < TOTAL_INCREMENT; i++) {
    pthread_mutex_lock(&lock);
    counter++;
    pthread_mutex_unlock(&lock);
  }
  return NULL;
}
int main() {
  pthread_t thread1, thread2;
  pthread_mutex_init(&lock, NULL);
  pthread_create(&thread1, NULL, increment, NULL);
  pthread_create(&thread2, NULL, increment, NULL);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  pthread_mutex_destroy(&lock);
  printf("Actual count = %d(Expected = %d)\n", counter, TOTAL_INCREMENT*2);
  return 0;
}
```

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.

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

// Global variables to store results
int average;
int minimum;
int maximum;

int *numbers;
int count;

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

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

void *find_maximum(void *arg) {
    maximum = numbers[0];
    for (int i = 1; i < count; i++) {
        if (numbers[i] > maximum) {
            maximum = numbers[i];
```

```
        }
    }
    pthread_exit(NULL);
}

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

    count = argc - 1;
    numbers = (int *)malloc(count * sizeof(int));

    for (int i = 0; i < count; i++) {
        numbers[i] = atoi(argv[i + 1]);
    }

    pthread_t t1, t2, t3;

    pthread_create(&t1, NULL, find_average, NULL);
    pthread_create(&t2, NULL, find_minimum, NULL);
    pthread_create(&t3, NULL, find_maximum, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);

    printf("The average value is %d\n", average);
    printf("The minimum value is %d\n", minimum);
    printf("The maximum value is %d\n", maximum);

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

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q20.multithread_caclulation.c -o calc
saidul@saidul-Latitude-7400:~/Desktop/os$ ./calc 90 81 78 95 79 72 85
The average value is 82
The minimum value is 72
The maximum value is 95
```

==========

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

int count;
int *nums;
double avrg;
int mx, mn;
void *average(void *args) {
  double sum = 0;
  for(int i = 0; i < count; i++) {
    sum += nums[i];
```

```
  }
  avrg = sum/count;
  return NULL;
}
void *maximum(void *args) {
  mx = nums[0];
  for(int i = 1; i < count; i++) {
    if(nums[i] > mx) mx = nums[i];
  }
  return NULL;
}
void *minimum(void *args) {
  mn = nums[0];
  for(int i = 1; i < count; i++) {
    if(nums[i] < mn) mn = nums[i];
  }
  return NULL;
}

int main(int argc, char *argv[]) {
  count = argc-1;
  nums = (int*)malloc(count*sizeof(int));
  for(int i = 1; i <= count; i++) nums[i-1] = atoi(argv[i]);

  pthread_t thread1, thread2, thread3;

  pthread_create(&thread1, NULL, average, NULL);
  pthread_create(&thread2, NULL, maximum, NULL);
  pthread_create(&thread3, NULL, minimum, NULL);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  pthread_join(thread3, NULL);

  printf("Average = %.4lf\n", avrg);
  printf("Maximum = %d\n", mx);
  printf("Minimum = %d\n", mn);

  return 0;
}
```

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.

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

int *fib_array;
int n;  // Number of Fibonacci numbers to generate

void *generate_fibonacci(void *arg) {
    if (n >= 1) fib_array[0] = 0;
    if (n >= 2) fib_array[1] = 1;

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

    pthread_exit(NULL);
}

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

    n = atoi(argv[1]);
    if (n <= 0) {
        printf("Please enter a positive integer.\n");
        return 1;
    }

    fib_array = (int *)malloc(n * sizeof(int));
    if (fib_array == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
```

```
    pthread_t fib_thread;
    pthread_create(&fib_thread, NULL, generate_fibonacci, NULL);

    // Wait for the child thread to finish
    pthread_join(fib_thread, NULL);

    printf("Fibonacci sequence:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", fib_array[i]);
    }
    printf("\n");

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

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc fibonacci.c -o fibonacci
saidul@saidul-Latitude-7400:~/Desktop/os$ ./fibonacci 10
Fibonacci sequence:
0 1 1 2 3 5 8 13 21 34
```

==========

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

int n;
int *nums;
void *fib_generator(void *args) {
  nums = (int*)malloc(n*sizeof(int));
  nums[0] = nums[1] = 1;
  for(int i = 2; i < n; i++) {
    nums[i] = nums[i-1]+nums[i-2];
  }
  return NULL;
}
int main(int argc, char *argv[]) {
  n = atoi(argv[1]);
  pthread_t thread;
  pthread_create(&thread, NULL, &fib_generator, NULL);
  pthread_join(thread, NULL);

  for(int i = 0; i < n; i++) printf("%d ", nums[i]);
  return 0;
}
```

==========
==========
---

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

Process 1:

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

#define SHM_KEY 1234
#define SEM_KEY 5678
#define NUM_ITERATIONS 5000
```

```
union semun {
    int val;
};

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

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

int main() {
    int i;
    int shmid, semid;
    int *counter;

    // Create shared memory
    shmid = shmget(SHM_KEY, sizeof(int), IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("Shared memory creation failed");-
        exit(1);
    }

    // Attach shared memory
    counter = (int *) shmat(shmid, NULL, 0);

    // Create semaphore
    semid = semget(SEM_KEY, 1, IPC_CREAT | 0666);
    if (semid == -1) {
        perror("Semaphore creation failed");
        exit(1);
    }

    // Initialize semaphore to 1
    union semun sem_union;
    sem_union.val = 1;
    semctl(semid, 0, SETVAL, sem_union);

    *counter = 0; // Initialize counter

    for (i = 0; i < NUM_ITERATIONS; i++) {
        semaphore_wait(semid);

        *counter = *counter + 1;

        semaphore_signal(semid);

        usleep(500); // Slow down to allow overlap
    }

    printf("Process 1 completed. Counter = %d\n", *counter);

    shmdt(counter);
```

```
    return 0;
}
```

Process 2:

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

#define SHM_KEY 1234
#define SEM_KEY 5678
#define NUM_ITERATIONS 5000

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

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

int main() {
    int i;
    int shmid, semid;
    int *counter;

    // Access existing shared memory
    shmid = shmget(SHM_KEY, sizeof(int), 0666);
    if (shmid == -1) {
        perror("Shared memory access failed");
        exit(1);
    }

    // Attach shared memory
    counter = (int *) shmat(shmid, NULL, 0);

    // Access existing semaphore
    semid = semget(SEM_KEY, 1, 0666);
    if (semid == -1) {
        perror("Semaphore access failed");
        exit(1);
    }

    for (i = 0; i < NUM_ITERATIONS; i++) {
        semaphore_wait(semid);

        *counter = *counter + 1;

        semaphore_signal(semid);

        usleep(500); // Slow down to allow overlap
```

```
    }

    printf("Process 2 completed. Counter = %d\n", *counter);

    shmdt(counter);

    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q17.process1_unrelated_process_race_situation_handle.c -o process1
saidul@saidul-Latitude-7400:~/Desktop/os$ ./process1
Process 1 completed. Counter = 8742
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q17.process2_unrelated_process_race_situation_handle.c -o process2
saidul@saidul-Latitude-7400:~/Desktop/os$ ./process2
Process 2 completed. Counter = 10000
```

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

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

sem_t counter_semaphore;

void *work(void *arg) {
    int id = *(int *)arg;

    sem_wait(&counter_semaphore); // Decrease semaphore (Wait)

    printf("Thread %d is working...\n", id);
    sleep(1); // Simulate some work
    printf("Thread %d finished work.\n", id);

    sem_post(&counter_semaphore); // Increase semaphore (Signal)

    pthread_exit(NULL);
}

int main() {
    pthread_t t1, t2, t3;
    int id1 = 1, id2 = 2, id3 = 3;

    sem_init(&counter_semaphore, 0, 2); // Allow max 2 threads at a time

    pthread_create(&t1, NULL, work, &id1);
    pthread_create(&t2, NULL, work, &id2);
    pthread_create(&t3, NULL, work, &id3);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);

    sem_destroy(&counter_semaphore);
```

```
    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q18.semaphore_counter_example.c -o semaphore_counter
saidul@saidul-Latitude-7400:~/Desktop/os$ ./semaphore_counter
Thread 1 is working...
Thread 2 is working...
Thread 1 finished work.
Thread 2 finished work.
Thread 3 is working...
Thread 3 finished work.
```

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?

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

void *thread_function(void *arg) {
    printf("Thread: I am a thread in process %d.\n", getpid());
    sleep(3);
    pthread_exit(NULL);
}

int main() {
    pthread_t t1;
    pthread_create(&t1, NULL, thread_function, NULL);
    sleep(1); // Let the thread start

    pid_t pid = fork();

    if (pid == 0) {
        printf("Child: I am process %d after fork.\n", getpid());
        sleep(2);
        printf("Child: Exiting.\n");
        exit(0);
    } else if (pid > 0) {
        printf("Parent: I am process %d after fork.\n", getpid());
        pthread_join(t1, NULL);
        printf("Parent: Exiting.\n");
    } else {
        perror("Fork failed");
    }

    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q19.A.multithread.c -o A
saidul@saidul-Latitude-7400:~/Desktop/os$ ./A
Thread: I am a thread in process 93193.
Parent: I am process 93193 after fork.
Child: I am process 93195 after fork.
Child: Exiting.
```

B. If a thread invokes the exec() system call, does it replace the entire code of the process?

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

void *thread_function(void *arg) {
    printf("Thread: I am running in process %d.\n", getpid());
    sleep(2);
    printf("Thread: Calling exec now!\n");
    execlp("ls", "ls", NULL); // This will replace the process
    pthread_exit(NULL);
}

int main() {
    pthread_t t1;
    pthread_create(&t1, NULL, thread_function, NULL);

    printf("Main: I am the main thread in process %d.\n", getpid());
    pthread_join(t1, NULL);

    printf("Main: This line will not run because exec() replaces the process.\n");
    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q19.B.multithread.c -o B
saidul@saidul-Latitude-7400:~/Desktop/os$ ./B
Main: I am the main thread in process 93748.
Thread: I am running in process 93748.
Thread: Calling exec now!
 A                                  q11.receiver_unrelated_process_msg_queue.c
```

C. If exec() is called immediately after forking, will all threads be duplicated?

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

void *thread_function(void *arg) {
    printf("Thread: I am a thread in process %d.\n", getpid());
    sleep(5);
    pthread_exit(NULL);
}

int main() {
    pthread_t t1;
    pthread_create(&t1, NULL, thread_function, NULL);
    sleep(1); // Let the thread start

    pid_t pid = fork();

    if (pid == 0) {
        printf("Child: I am process %d, calling exec now.\n", getpid());
        execlp("ls", "ls", NULL); // This will replace the process
        exit(0);
    } else if (pid > 0) {
        printf("Parent: I am process %d.\n", getpid());
        pthread_join(t1, NULL);
        printf("Parent: Exiting.\n");
    } else {
        perror("Fork failed");
    }

    return 0;
}
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q19.C.multithread.c -o C
saidul@saidul-Latitude-7400:~/Desktop/os$ ./C
Thread: I am a thread in process 93870.
Parent: I am process 93870.
Child: I am process 93872, calling exec now.
 A                                  q11.receiver_unrelated_process_msg_queue.c
```

Explanation:

A. fork() in multithreaded process:

- Only the calling thread is duplicated in the child process
- Other threads from the parent are not copied
- The child becomes single-threaded (POSIX requirement)

B. exec() behavior:

- Replaces all threads of the calling process
- Entire process memory space is overwritten
- Only the new program's main thread exists after exec()

C. exec() after fork():

- Since fork() only copies the calling thread...
- And exec() replaces all threads...
- No thread duplication occurs in this case
- The exec()'d program starts with just one thread

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.

Server:

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

#define PORT 8080
```

```
#define BUFFER_SIZE 1024

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[BUFFER_SIZE] = {0};
    char *response = "Hello from server\n";

    // Create socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == 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);
    }

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

    printf("Server is listening on port %d...\n", PORT);

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

        read(new_socket, buffer, BUFFER_SIZE);
        printf("Received: %s\n", buffer);

        send(new_socket, response, strlen(response), 0);
        close(new_socket);
    }

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

Client:

```
// client.c
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
#include <unistd.h>
#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};

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("Socket creation error");
        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) {
        perror("Invalid address");
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        perror("Connection failed");
        return -1;
    }

    send(sock, message, strlen(message), 0);
    read(sock, buffer, sizeof(buffer));
    printf("Server response: %s\n", buffer);

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

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q22.server.c -o server
saidul@saidul-Latitude-7400:~/Desktop/os$ ./server
Server is listening on port 8080...
Received: Hello from client
```

```
saidul@saidul-Latitude-7400:~/Desktop/os$ gcc q22.client.c -o client
saidul@saidul-Latitude-7400:~/Desktop/os$ ./client
Server response: Hello from server
```

Explanation:

1) Single-threaded server serving multiple clients:

- The server handles **one client at a time**.
- While processing a client, **other clients wait** (or may get connection refused if the backlog is full).
- This causes **slow response and poor concurrency**.
- Server is simple but **not scalable**.

---

2) Multi-threaded server serving multiple clients:

- For each new client, server **creates a new thread** to handle the client independently.
- Multiple clients served **concurrently**.
- Good for **improved concurrency and responsiveness**.
- Threads share the same memory space (careful with synchronization).
- More complex code but better performance.

---

3) Single-threaded server creating multiple child processes for clients:

- On accepting a client connection, the server **forks a new child process**.
- Each child handles one client independently.
- Clients served concurrently in **separate processes**.
- Processes have separate memory → more memory overhead but safer isolation.
- This is a classical **multi-process server model**.

Ref: “Operating System Concepts”, chatgpt, deepseek