one_to_seven.c

```c
// Assignment 1-a
// 1. Include necessary header files for system calls and standard I/O.
// 2. Define the main function.
// 3. Declare variables for process IDs and counters for child and grandchild processes.
// 4. Call fork() to create a child process.
//    a. If fork() fails, print an error message and exit with status 1.
//    b. If fork() returns 0 (child process):
//       i. Increment the child process counter.
//       ii. Print the PID and PPID of the child process.
//       iii. Call fork() again to create a grandchild process.
//           - If fork() fails, print an error message and exit with status 1.
//           - If fork() returns 0 (grandchild process):
//             * Increment the grandchild process counter.
//             * Print the PID, PPID, and roll number.
//             * Exit with status 0.
//           - Wait for the grandchild process to complete.
//           - Exit the child process with status 0.
//    c. If fork() returns a positive value (parent process):
//       i. Print the PID of the parent process.
//       ii. Wait for the child process to complete.
//       iii. Print the total number of child and grandchild processes created.
// 5. Return 0 to indicate successful execution.

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>  // Include this for wait()

int main() {
    pid_t pid, gcid;
    int child_count = 1, grandchild_count = 1;

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

    if (pid == 0) { // Child process
        child_count++;
        printf("Child Process: PID = %d, Parent PID = %d\n", getpid(), getppid());

        gcid = fork();
        if (gcid < 0) {
            perror("Fork failed");
            exit(1);
        }

        if (gcid == 0) { // Grandchild process
            grandchild_count++;
            printf("Grandchild Process: PID = %d, Parent PID = %d\n", getpid(), getppid());
            printf("Roll Number: CSM24004\n");
```

```
            exit(0);
        }

        wait(NULL); // Ensure grandchild completes
        exit(0);
    }
    else { // Parent process
        printf("Parent Process: PID = %d\n", getpid());
        wait(NULL); // Ensure child completes
        printf("Total Child Processes Created: %d\n", child_count);
        printf("Total Grandchild Processes Created: %d\n", grandchild_count);
    }
    return 0;
}


// Assignment 1-b
// Function to print logged-in users
// - Open utmp file
// - Loop through entries
// - If entry is a user process, print user details
// - Close utmp file

// Main function
// - Declare process IDs
// - Fork a child process
//   - If fork fails, print error and exit
//   - If in child process:
//     - Print child process details
//     - Fork a grandchild process
//       - If fork fails, print error and exit
//       - If in grandchild process:
//         - Print grandchild process details
//         - Call function to print logged-in users
//         - Exit grandchild process
//     - Wait for grandchild to finish
//     - Exit child process
//   - If in parent process:
//     - Print parent process details
//     - Wait for child to finish
//     - Print message indicating child process finished

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

void print_logged_in_users() {
    struct utmp *entry;
    setutent();  // Open /var/run/utmp
    while ((entry = getutent()) != NULL) {
        if (entry->ut_type == USER_PROCESS) {  // Only print active user logins
            printf("User: %s, Terminal: %s, Host: %s\n", entry->ut_user, entry->ut_line, entry-
>ut_host);
```

```c
        }
    }
    endutent();  // Close file
}

int main() {
    pid_t pid, gcid;

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

    if (pid == 0) {  // Child process
        printf("Child Process: PID = %d, Parent PID = %d\n", getpid(), getppid());

        gcid = fork();
        if (gcid < 0) {
            perror("Fork failed");
            exit(1);
        }

        if (gcid == 0) {  // Grandchild process
            printf("Grandchild Process: PID = %d, Parent PID = %d\n", getpid(), getppid());
            print_logged_in_users();
            exit(0);
        }

        wait(NULL);  // Child waits for grandchild
        exit(0);
    }
    else {  // Parent process
        printf("Parent Process: PID = %d\n", getpid());
       wait(NULL);  // Parent waits for child
        printf("Parent: Child process finished execution.\n");
    }

    return 0;
}

// Assignment 1-c: Child_program.c
// Pseudo code for child program:
// 1. Include necessary headers.
// 2. Define the main function with arguments for argc and argv.
// 3. Print the program name using argv[0].
// 4. Print the PID of the child process using getpid().
// 5. Exit the program.

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

int main(int argc, char *argv[]) {
    printf("Child Program Name: %s\n", argv[0]);
    printf("Child Process is running with PID: %d\n", getpid());

```

```c
    return 0;
}

// Assignment 1-d: Parent_program.c
// Pseudo code for the parent program:
// 1. Create a child process using fork().
// 2. In the child process:
//    a. Print the PID and parent PID.
//    b. Prepare arguments for execv().
//    c. Overlay the child process with a new program using execv().
//    d. Handle errors if execv() fails.
// 3. In the parent process:
//    a. Wait for the child process to finish using wait().
//    b. Check if the child exited normally using WIFEXITED.
//    c. Print the child's exit status or an error message.
//    d. Print the parent's PID.

#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t pid;
    int status;

    pid = fork();

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

    } else if (pid == 0) {
        // CP
        printf("Child Process before execv(): PID = %d, Parent PID = %d\n", getpid(), getppid());

        // Prepare arguments for execv()
        char *args[] = {"./child_program", NULL};

        //Overlay child process with child_program using execv()
        if (execv(args[0], args) < 0) {
            perror("execv failed");
            exit(1);
        }
    } else {
        //Parent process
        //wait for child process to finish
        if (wait(&status) < 0) {
            perror("wait failed");
            exit(1);
        }

        // Check if child exited normally
        if (WIFEXITED(status)) {
            printf("Parent Process: Child exited with status = %d\n", WEXITSTATUS(status));
        } else {
            printf("Parent Process: Child did not exit normally\n");
```

```
        }

        printf("Parent Process: PID = %d\n", getpid());
    }

    return 0;
}

// Assignment 2-a:
/*
* Assignment 2(a)
 * 1. Include necessary headers for input/output, process control, and signal handling.
 * 2. Define a signal handler function:
 *    a. Print a message indicating SIGINT was received.
 *    b. Use fork() to create a child process.
 *    c. In the child process:
 *       i. Print the child process ID and its parent process ID.
 *       ii. Exit the child process to prevent it from continuing the parent's execution flow.
 *    d. In the parent process:
 *       i. Use wait() to ensure the child process completes execution to avoid zombie processes.
 * 3. In the main function:
 *    a. Register the signal handler for SIGINT using signal().
 *    b. Use an infinite loop with pause() to keep the process alive and waiting for signals.
 * 4. Compile and run the program. Press Ctrl+C to test the signal handling and process creation.
 */
/*
 * This program demonstrates handling SIGINT to create a child process using fork().
 * When Ctrl+c is pressed, the parent and child processes print their PIDs and PPIDs.
 */

 #include <stdio.h>
 #include <unistd.h>
 #include <wait.h>
 #include <stdlib.h>
 #include <signal.h>

 void signal_handler(int num) {
         printf("Ctrl + C pressed\n");
         pid_t pid;
         pid = fork();
         if (pid < 0) {
                 perror("Fork Failed");
                 exit(1);
         }
         if (pid == 0) {
                 printf("Child Process Created \nPID = %d, Parent PID = %d\n", getpid(),
getppid());
         exit(0); // Terminates child immediately
         }

         else {
                 wait(NULL); //Ensure child completes to prevent to become zombiee
     }
 }

 int main() {
```

```
276         signal(SIGINT, signal_handler);
277 
278         // Keep process alive to handle signal
279     while(1){
280         pause(); // No busy loop
281     }
282         return 0;
283  }
284 
285 
286 // Assignment 2-b:
287 /*
288  * Assignment 2(b)
289  * 1. Include necessary headers for input/output, signal handling, and process control.
290  * 2. Define a signal handler function to display the received signal ID.
291  * 3. In the main function:
292  *    a. Declare variables for Process ID (pid) and Signal ID (sig).
293  *    b. Register the signal handler for all possible signals (1 to 31).
294  *    c. Prompt the user to enter the Process ID and Signal ID.
295  *    d. Validate the input to ensure the PID and Signal ID are valid.
296  *    e. Use the kill() system call to send the specified signal to the given process.
297  *    f. Handle errors if the kill() system call fails.
298  * 4. Display appropriate messages for successful or failed signal delivery.
299  * 5. Exit the program.
300  */
301 /*
302  * This program takes a Process ID (PID) and a Signal ID as input and sends the signal to the
    specified process.
303  * The purpose is to demonstrate the use of the kill() system call.
304  * The receiving process (modified Assignment 2(a)) should handle the signal and display the
    received signal ID.
305  */
306 
307  #include <stdio.h>
308  #include <stdlib.h>
309  #include <signal.h>
310  #include <unistd.h>
311 
312  // Signal handler function to display received signal ID
313  void signal_handler(int signum) {
314      printf("Received Signal: %d\n", signum);
315  }
316 
317  int main() {
318      pid_t pid;
319      int sig;
320 
321      // All possible signals
322      for (int i = 1; i < 32; i++) {
323          signal(i, signal_handler);
324      }
325      printf("Enter Process ID: ");
326      scanf("%d", &pid);
327      printf("Enter Signal ID: ");
328      scanf("%d", &sig);
329 
```

```
    // Validate input
    if (pid <= 0 || sig <= 0 || sig > 31) {
        printf("Invalid PID or Signal ID\n");
        return 1;
    }

    // Send signal to process using kill()
    if (kill(pid, sig) == 0) {
        printf("Signal %d sent to process %d successfully.\n", sig, pid);
    } else {
        perror("Failed to send signal");
        return 1;
    }

    return 0;
 }

// Assignment 2-c
/*
* Assignment 2(c)
 * 1. Include necessary headers for input/output and pthread library.
 * 2. Define a thread function `print_hello` that prints "HELLO WORLD" and exits using pthread_exit().
 * 3. In the main function:
 *    a. Declare a thread identifier of type `pthread_t`.
 *    b. Use `pthread_create()` to create a new thread that runs the `print_hello` function.
 *       - Check for errors in thread creation and handle them using `perror()` and `exit()`.
 *    c. Use `pthread_join()` to wait for the created thread to finish execution.
 *    d. Print a message indicating the thread has completed.
 * 4. Return 0 to indicate successful program execution.
 *
 * Purpose of System Calls:
 * - `pthread_create()`: Used to create a new thread at the user level.
 * - `pthread_exit()`: Used to terminate the thread explicitly.
 * - `pthread_join()`: Used to wait for the thread to complete before proceeding.
 */
/*
 * This program creates a user-level thread to print "HELLO WORLD".
 */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Thread function to print message
void* print_hello(void* arg) {
    printf("HELLO WORLD\n");
    pthread_exit(NULL);
}
int main()
{
    pthread_t thread_id;
    // Create a thread
    if (pthread_create(&thread_id, NULL, print_hello, NULL) != 0) {
        perror("pthread_create");

        exit(1);
```

```
    }

    // Wait for the thread to finish
    pthread_join(thread_id, NULL);
    printf("Thread completed\n");

    return 0;
}

// Assignment 3
// Include necessary header files for signal handling, process creation, and inter-process communication

// Define a signal handler function 'handler' to handle SIGINT signal
// Inside the handler:
// 1. Create a pipe for communication between parent and child processes
// 2. Use fork() to create a child process
// 3. In the child process:
//    a. Close the read end of the pipe
//    b. Write the child's process ID to the pipe
//    c. Take input N (number of Fibonacci terms) from the user
//    d. Write N to the pipe
//    e. Generate the Fibonacci series up to N terms and write each term to the pipe
//    f. Write the signal ID (SIGINT) to the pipe
//    g. Close the write end of the pipe and exit
// 4. In the parent process:
//    a. Close the write end of the pipe
//    b. Read the child's process ID from the pipe and display it
//    c. Read N from the pipe and display the Fibonacci series terms received from the child
//    d. Read the signal ID from the pipe and display it
//    e. Close the read end of the pipe and wait for the child process to terminate

// In the main function:
// 1. Register the signal handler for SIGINT
// 2. Enter an infinite loop to simulate a running program
// 3. Print a message indicating the main program is running and sleep for 1 second
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>

void handler(int sig) {
    int pipefd[2];
    if(pipe(pipefd) == -1) {
        perror("pipe Failed");
        exit(1);
    }
    pid_t pid = fork();
    if(pid == -1) {
        perror("fork Failed");
        exit(1);
    }
    if(pid == 0) {
        close(pipefd[0]);
```

```
        pid_t mypid = getpid();
        write(pipefd[1], &mypid, sizeof(pid_t));

        int N;
        printf("Enter N: ");
        scanf("%d", &N);
        write(pipefd[1], &N, sizeof(int));

        int a = 0, b = 1, c;
        for(int i = 0; i < N; i++) {
            write(pipefd[1], &a, sizeof(int));
            c = a + b;
            a = b;
            b = c;
        }
        write(pipefd[1], &sig, sizeof(int));
        close(pipefd[1]);
        exit(0);
    }
    else {
        close(pipefd[1]);
        pid_t child_pid;
    read(pipefd[0], &child_pid, sizeof(pid_t));
    printf("child PID: %d\n", child_pid);

    int N;
    read(pipefd[0], &N, sizeof(int));
    printf("Fibonacci series up to %d terms: ", N);

    int term;
    for(int i = 0; i<N; i++) {
        read(pipefd[0], &term, sizeof(int));
        printf("%d ", term);
    }
    printf("\n");
    int signal_id;
    read(pipefd[0], &signal_id, sizeof(int));
    printf("Signal ID: %d\n", signal_id);

    close(pipefd[0]);
    wait(NULL);
    }
}

int main() {
    signal(SIGINT, handler);
    while(1) {
        printf("Main program running...\n");
        sleep(1);
    }
    return 0;
}

// Assignment 4
// This program demonstrates file locking using `fcntl()` in a multi-process environment.
// It locks a file, reads its contents, holds the lock for 10 seconds, and then releases the lock.
```

```c
// 1. Include necessary headers for file operations, locking, and process control.
// 2. Define a function `lock_file` to set or release a file lock using `fcntl()`.
//    - Initialize a `struct flock` with appropriate lock type, offset, and length.
//    - Use `fcntl()` with `F_SETLKW` to set or release the lock, handling errors.
// 3. In the `main` function:
//    - Check if the filename is provided as a command-line argument.
//    - Open the file in read-write mode using `fopen()`.
//    - Handle errors if the file cannot be opened.
//    - Print the process ID and attempt to lock the file using `lock_file`.
//    - Read the file contents using `read()` and print them to the console.
//    - Handle errors during reading and release the lock if necessary.
//    - Hold the lock for 10 seconds to simulate exclusive access.
//    - Release the lock using `lock_file` and close the file.
//    - Exit the program.
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

void lock_file(int fd, short lock_type) {
    struct flock lock;
    lock.l_type = lock_type;    // F_WRLCK (write lock) or F_UNLCK (unlock)
    lock.l_whence = SEEK_SET;   // Start from beginning of file
    lock.l_start = 0;           // Offset 0
    lock.l_len = 0;             // Lock entire file (0 means to end)

    if (fcntl(fd, F_SETLKW, &lock) == -1) {
        perror("Error setting lock");
        exit(EXIT_FAILURE);
    }
}

int main(int argc, char *argv[]) {
    FILE* fptr;
    char content[255];

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1;
    }


    fptr = fopen(argv[1], "r+"); // "r+" for locking
    if (fptr == NULL) {
        printf("File open Unsuccessful\n");
        return 1;
    }

    printf("Process %d: Attempting to lock file...\n", getpid());
    lock_file(fileno(fptr), F_WRLCK);
    printf("Process %d: File locked. Reading contents...\n", getpid());

    int fd = fileno(fptr);
    ssize_t bytes_read;
    while ((bytes_read = read(fd, content, sizeof(content) - 1)) > 0) {
        content[bytes_read] = '\0';
```

```
        printf("%s", content);
    }
    if (bytes_read == -1) {
        perror("Error reading file");
        lock_file(fd, F_UNLCK);
        fclose(fptr);
        return 1;
    }

    printf("\nProcess %d: Holding lock for 10 seconds...\n", getpid());
    sleep(10);

    printf("Process %d: Unlocking file...\n", getpid());
    lock_file(fd, F_UNLCK);

    fclose(fptr);
    return 0;
}

// Assignment 5 -a (program1.c)
// This program demonstrates the use of named pipes (FIFOs) for inter-process communication.
// Pseudo code for the program:
// 1. Define the FIFO file path.
// 2. Create a named FIFO (if it doesn't already exist) using the mknod() system call.
// 3. Open the FIFO in read-only mode.
// 4. Read data from the FIFO written by another process (Program2).
// 5. Display the received data.
// 6. Close the FIFO file descriptor.
// 7. Delete the FIFO after use to clean up resources.
#include <unistd.h>
#include <stdio.h>

int main() {
    char *fifo_name = "/tmp/myfifo";
    int fd = open(fifo_name, O_RDONLY);
    if (fd == -1) {
        perror("Open failed");
        return 1;
    }

    char buffer[100];
    int bytesRead = read(fd, buffer, sizeof(buffer));
    if (bytesRead > 0) {
        buffer[bytesRead] = '\0';
        printf("Reader Received: %s\n", buffer);
    }
    // printf("%ld",sizeof(buffer));
    // printf("Reader recived: %s\n", buffer);

    close(fd);
    unlink(fifo_name); // Deleting the fifo
    return 0;
}

// Assignment 5 -a (program2.c)
// This program demonstrates the use of named pipes (FIFOs) for inter-process communication.
```

```
// Pseudo code for Program2:

/*
1. Include necessary header files for file operations, FIFO handling, and standard I/O.
2. Define the FIFO name (path) as a string constant.
3. Create the FIFO using the mknod() system call with appropriate permissions (read/write for all users).
4. Open the FIFO in write-only mode using the open() system call.
    - If the open() call fails, print an error message and exit the program.
5. Prompt the user to input a message.
6. Read the user input using fgets() and remove the trailing newline character.
7. Write the user input to the FIFO using the write() system call.
8. Print a confirmation message indicating the message has been sent.
9. Close the FIFO file descriptor using the close() system call.
10. Exit the program.
*/
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>

int main() {
    char *fifo_name = "/tmp/myfifo";
    mknod(fifo_name, S_IFIFO | 0666, 0); // Create FIFO

    int fd = open(fifo_name, O_WRONLY);
    if (fd == -1) {
        perror("open failed");
        return 1;
    }

    printf("Send your message from here!\nType: ");
    char message[100];

    fgets(message, sizeof(message), stdin);
    //scanf("%s", message);
    message[strcspn(message, "\n")] = '\0'; // Remove newline character

    write(fd, message, strlen(message) + 1);
    printf("Writer sent: %s\n", message);

    close(fd);
    return 0;
}

// Assignment 5 -b (client.c)
// Include necessary headers
// Define constants and message structure

// Main function
// 1. Generate a unique key using ftok() to identify the message queue
// 2. Connect to the message queue using msgget()
// 3. Prompt the user to enter a message
// 4. Remove the newline character from the input
// 5. Set the message type to 1 (for server)
```

```c
// 6. Send the message to the server using msgsnd()
// 7. Handle errors if message sending fails
// 8. Wait for a reply from the server (type 2) using msgrcv()
// 9. Handle errors if message receiving fails
// 10. Print the server's reply
// 11. Exit the program

// Summary:
// This client program demonstrates inter-process communication (IPC) using System V Message Queues.
// It connects to a message queue, sends a message to the server, and waits for a reply. The message
// queue ID is generated using ftok() and connected using msgget(). Messages are sent and received
// using msgsnd() and msgrcv(), respectively.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/msg.h>

#define MAX_SIZE 100

// Message structure
struct msgbuf {
    long mtype;
    char mtext[MAX_SIZE];
};

int main() {
    key_t key;
    int msqid;
    struct msgbuf message;

    // Generate the same key as server
    key = ftok("progfile", 65);

    // Connect to the message queue
    msqid = msgget(key, 0666);

    // Get user input
    printf("Client: enter a message: ");
    fgets(message.mtext, MAX_SIZE, stdin);
    message.mtext[strcspn(message.mtext, "\n")] = 0; // Remove newlind
    message.mtype = 1; // Type 1 fro server

    // Send message
    if (msgsnd(msqid, &message, strlen(message.mtext) + 1, 0) == -1) {
        perror("msgsnd failed");
        exit(1);
    }

    // Recieve reply (type 2 from server)
    if (msgrcv(msqid, &message, MAX_SIZE, 2, 0) == -1) {
        perror("msgrcv failed");
        exit(1);
    }
```

```
    printf("Client: Server replied: %s\n", message.mtext);
    return 0;
}

// Assignment 5 -b (server.c)
/*
 * 1. Include necessary headers and define constants.
 * 2. Define a structure for the message queue with a type and text field.
 * 3. In the main function:
 *    a. Generate a unique key using ftok().
 *    b. Create a message queue using msgget() with appropriate permissions.
 *    c. Print the message queue ID to stderr for debugging purposes.
 *    d. Enter an infinite loop to handle messages:
 *       i. Receive a message of type 1 from the client using msgrcv().
 *       ii. Print the received message to the console.
 *       iii. Prepare a reply message with type 2 and a predefined response.
 *       iv. Send the reply message back to the client using msgsnd().
 *       v. Break the loop after one message for demonstration purposes.
 *    e. Remove the message queue using msgctl() with IPC_RMID to clean up.
 *    f. Print a message indicating the server is exiting.
 * 4. Return 0 to indicate successful execution.
 *
 * Summary:
 * The server program demonstrates inter-process communication (IPC) using System V Message Queues.
 * It receives a message from a client, processes it, sends a reply, and cleans up the message queue.
 */
// Server Program
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_SIZE 100

// Message structuree
struct msgbuf {
    long mtype; // Message type
    char mtext[MAX_SIZE];
};

int main() {
    key_t key;
    int msqid;
    struct msgbuf message;

    // Generate a unique key
    key = ftok("progfile", 65); // progfile is dummy file, 65 id a random character

    // Create message queue
    msqid = msgget(key, 0666 | IPC_CREAT);

    // Print queue ID to stderr
```

```
        fprintf(stderr, "Message Queue ID: %d\n", msqid);
        printf("Server: Waiting for messages...\n");

        while (1) {
            // Receive message of type 1 (from client)
            if (msgrcv(msqid, &message, MAX_SIZE, 1, 0) == -1) {
                    perror("msgrcv failed");
                    exit(1);
            }

            printf("Server: Received: %s\n", message.mtext);

            // Prepare reply
            message.mtype = 2; // Type 2 for client
            strcpy(message.mtext, "Got it!");
            if (msgsnd(msqid, &message, strlen(message.mtext) + 1, 0) == -1) {
                    perror("msgsnd failed");
                exit(1);
            }

            // For demo, break after one message (remove this for continuous run)
            break;
        }

    // Remove the message queue
    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("magctl failed");
        exit(1);
    }

    printf("Server: Queue removed. Exiting.\n");
    return 0;

}

// Assignment 6 -a (program1.c)
/*
1. Include necessary headers for shared memory operations and standard I/O.
2. Define a unique key for shared memory creation.
3. Create a shared memory segment using shmget() with the defined key and required size.
    - If creation fails, print an error message and exit.
4. Attach the shared memory segment to the process's address space using shmat().
    - If attachment fails, print an error message and exit.
5. Write the process ID of the current process into the shared memory.
6. Print a message indicating that the process ID has been written to shared memory.
7. Detach the shared memory segment from the process's address space using shmdt().
8. Exit the program.

Summary:
This program (program1.c) demonstrates the creation and usage of shared memory in a Linux
environment. It writes the process ID of the current process into a shared memory segment, which
can be read by another program (program2.c). The shared memory segment is properly detached after
use.
*/
#include <stdio.h>
#include <sys/ipc.h>
```

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

int main() {
    key_t key = 1234;  // Unique key for shared memory
    int shmid = shmget(key, sizeof(int), 0666 | IPC_CREAT); // Create shared memory
    if (shmid == -1) {
        perror("shmget failed");
        return 1;
    }

    int *shared_data = (int *)shmat(shmid, NULL, 0); // Attach memory
    if (shared_data == (int *)-1) {
        perror("shmat failed");
        return 1;
    }

    *shared_data = getpid(); // Write Process ID to shared memory
    printf("Process 1 (PID: %d) wrote to shared memory.\n", *shared_data);

    shmdt(shared_data); // Detach memory
    return 0;
}

// Assignment 6 -a (program2.c)
/*
 * 1. Include necessary headers for shared memory operations and standard I/O.
 * 2. Define a unique key for shared memory identification.
 * 3. Access an existing shared memory segment using shmget with the defined key.
 *    - If shmget fails, print an error message and exit.
 * 4. Attach the shared memory segment to the process's address space using shmat.
 *    - If shmat fails, print an error message and exit.
 * 5. Read the integer value from the shared memory and print it.
 * 6. Detach the shared memory segment from the process using shmdt.
 * 7. Remove the shared memory segment using shmctl with IPC_RMID command.
 * 8. Exit the program.
 */

/*
 * Summary:
 * This program (program2.c) demonstrates shared memory usage by reading data written by another process (program1.c).
 * It accesses an existing shared memory segment, reads the data, and then cleans up by detaching and removing the shared memory.
 */
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main() {
    key_t key = 1234;
    int shmid = shmget(key, sizeof(int), 0666); // Access existing shared memory
    if (shmid == -1) {
        perror("shmget failed");
        return 1;
    }
```

```
    int *shared_data = (int *)shmat(shmid, NULL, 0); // Attach memory
    if (shared_data == (int *)-1) {
        perror("shmat failed");
        return 1;
    }

    printf("Process 2 read from shared memory: %d\n", *shared_data);

    shmdt(shared_data); // Detach memory
    shmctl(shmid, IPC_RMID, NULL); // Remove shared memory
    return 0;
}

// Assignment 6 -b (process_time.c)
/*
 * 1. Include necessary headers for file operations, process handling, and time measurement.
 * 2. Define the main function.
 * 3. Declare variables to store start and end times using `struct timeval`.
 * 4. Open a file named "process_time.txt" in write mode to store the process times.
 * 5. Record the submission time using `gettimeofday()` and store it in the `start` variable.
 * 6. Create a child process using `fork()`.
 * 7. In the child process:
 *    a. Print a message indicating the child process is running.
 *    b. Simulate some execution using `sleep()`.
 *    c. Exit the child process.
 * 8. In the parent process:
 *    a. Wait for the child process to finish using `wait()`.
 *    b. Record the finish time using `gettimeofday()` and store it in the `end` variable.
 *    c. Write the submission and finish times to the file in seconds and microseconds.
 *    d. Close the file.
 *    e. Print a message indicating the times have been recorded.
 * 9. Return 0 to indicate successful execution.
 *
 * Summary:
 * The program creates a child process and measures its submission and finish times using the `gettimeofday()` system call. These times are written to a file named "process_time.txt". The `fork()` system call is used to create the child process, and `wait()` ensures the parent waits for the child to complete. The program demonstrates basic process handling and time measurement in C.
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/time.h>
#include <stdlib.h>

int main() {
    struct timeval start, end;
    FILE *file = fopen("process_time.txt", "w");  // Open file to store times

    gettimeofday(&start, NULL);  // Record submission time
    pid_t pid = fork(); // Create child process

    if (pid == 0) { // Child process
        printf("Child process is running...\n");
```

```
        sleep(3); // Simulate execution
        exit(0);
    } else { // Parent process
        wait(NULL); // Wait for child to finish
        gettimeofday(&end, NULL);  // Record finish time

        fprintf(file, "Submission Time: %ld.%06ld seconds\n", start.tv_sec, start.tv_usec);
        fprintf(file, "Finish Time: %ld.%06ld seconds\n", end.tv_sec, end.tv_usec);
        fclose(file);

        printf("Process times recorded in 'process_time.txt'\n");
    }
    return 0;
}


// Assignment 7 a (program1.c)
/*
1. Include necessary headers for semaphore operations and file handling.
2. Define a function `semaphore_signal` to increment the semaphore value by 1.
3. In the `main` function:
    a. Generate a unique key for the semaphore using `key_t`.
    b. Create a semaphore set with one semaphore using `semget`.
    c. Initialize the semaphore value to 0 using `semctl`.
    d. Open the file "sample.txt" in read mode.
    e. Read the contents of the file character by character and print them to the console.
    f. Close the file after reading.
    g. Signal the semaphore to indicate that Program1 has completed reading the file.
    h. Print a message indicating that Program1 has completed and signaled Program2.
4. End the program.

Summary:
This program (Program1) reads the contents of a file and uses a semaphore to synchronize with another program (Program2). The semaphore ensures that Program2 can only read the file after Program1 has completed its reading. The semaphore is initialized to 0 and incremented by 1 after Program1 finishes reading, signaling Program2 to proceed.
*/
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdlib.h>

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

int main() {
    key_t key = 1234;
    int semid = semget(key, 1, 0666 | IPC_CREAT); // Create semaphore

    // Initialize semaphore to 0 initially
    semctl(semid, 0, SETVAL, 0);

    FILE *file = fopen("sample.txt", "r");

```

```
    char ch;
    printf("Program1 is reading the file:\n");
    while ((ch = fgetc(file)) != EOF) {
        printf("%c", ch);
    }
    fclose(file);

    // After reading, signal program2
    semaphore_signal(semid);
    printf("\nProgram1 completed. Signaled Program2.\n");

    return 0;
}

/*
 * 1. Include necessary headers for semaphore and file operations.
 * 2. Define a function `semaphore_wait` to decrement the semaphore value (wait operation).
 * 3. In the `main` function:
 *    a. Generate a unique key for the semaphore using `key_t`.
 *    b. Access the existing semaphore set using `semget`.
 *    c. Wait for the semaphore to be released by program1 using `semaphore_wait`.
 *    d. Open the file "sample.txt" in read mode.
 *    e. Read and print the contents of the file character by character.
 *    f. Close the file after reading.
 *    g. Remove the semaphore using `semctl` to clean up resources.
 *    h. Print a message indicating program2 has completed.
 */

/*
 * Summary:
 * This program (program2.c) ensures synchronization with program1.c using semaphores.
 * It waits for program1 to finish reading the file before it starts reading the same file.
 * After reading, it removes the semaphore to clean up resources.
 */
// Assignment 7a - program2
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdlib.h>

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

int main() {
    key_t key = 1234;
    int semid = semget(key, 1, 0666); // Access existing semaphore

    // Wait until program1 completes
    semaphore_wait(semid);

    FILE *file = fopen("sample.txt", "r");

    char ch;
    printf("Program2 is reading the file after Program1:\n");
```

```
    while ((ch = fgetc(file)) != EOF) {
        printf("%c", ch);
    }
    fclose(file);

    // Remove the semaphore after use
    semctl(semid, 0, IPC_RMID);
    printf("\nProgram2 completed and semaphore removed.\n");

    return 0;
}

// Assignment 7 b (program1.c)
/*
 * 1. Include necessary headers for shared memory and semaphore operations.
 * 2. Define a function to signal (increment) the semaphore.
 * 3. In the main function:
 *    a. Generate a unique key for shared memory and semaphore.
 *    b. Create a shared memory segment and a semaphore using the key.
 *    c. Attach the shared memory segment to the process's address space.
 *    d. Write a string (e.g., "Hello from Shared Memory!") to the shared memory.
 *    e. Initialize the semaphore to 0 to indicate it's locked.
 *    f. Print a message indicating data has been written to shared memory.
 *    g. Signal the semaphore to notify the other process (prog2) that data is ready.
 *    h. Detach the shared memory segment from the process's address space.
 * 4. Exit the program.

 * Summary:
 * prog1.c writes a string to shared memory and uses a semaphore to signal prog2.c
 * that the data is ready. It demonstrates inter-process communication using shared
 * memory and semaphores.
 */
// Assignment 7b - program1
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdlib.h>

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

int main() {
    key_t key = 5678;
    int shmid = shmget(key, 100, 0666 | IPC_CREAT);
    int semid = semget(key, 1, 0666 | IPC_CREAT);

    char *str = (char *)shmat(shmid, NULL, 0);
    sprintf(str, "Hello from Shared Memory!");

    semctl(semid, 0, SETVAL, 0); // Initialize semaphore to 0

    printf("prog1: Data written to shared memory.\n");

```

```
    semaphore_signal(semid); // Signal prog2 to read
    shmdt(str); // Detach shared memory

    return 0;
}

/*
    1. Include necessary headers for shared memory and semaphore operations.
    2. Define a function `semaphore_wait` to decrement the semaphore value (wait operation).
    3. In the `main` function:
       a. Generate a unique key for shared memory and semaphore.
       b. Access the shared memory segment and semaphore created by prog1.
       c. Wait for prog1 to write data into the shared memory using `semaphore_wait`.
       d. Attach the shared memory segment to the process's address space.
       e. Read and print the data from the shared memory.
       f. Detach the shared memory segment from the process's address space.
       g. Remove the semaphore and shared memory segment using appropriate system calls.
       h. Print a message indicating successful cleanup.
    4. End the program.

    Summary:
    This program (prog2.c) reads data from a shared memory segment written by prog1.c.
    It uses a semaphore to synchronize access, ensuring prog1 writes data before prog2 reads it.
    After reading, it cleans up by removing the semaphore and shared memory.
*/
// Assignment 7b - program2
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdlib.h>

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

int main() {
    key_t key = 5678;
    int shmid = shmget(key, 100, 0666);
    int semid = semget(key, 1, 0666);

    semaphore_wait(semid); // Wait for prog1 to write

    char *str = (char *)shmat(shmid, NULL, 0);
    printf("prog2: Data read from shared memory: %s\n", str);
    shmdt(str); // Detach shared memory

    // Remove semaphore and shared memory
    semctl(semid, 0, IPC_RMID);
    shmctl(shmid, IPC_RMID, NULL);
    printf("prog2: Semaphore and shared memory removed.\n");

    return 0;
}
```