COSC 439: Operating Systems

Assignment#2

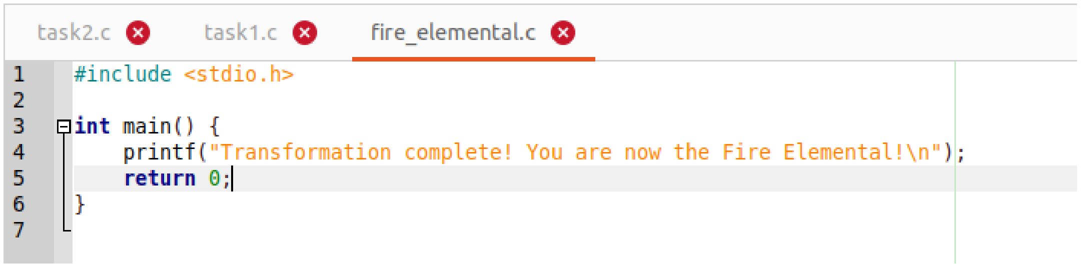
Computer and Information Sciences, Towson University

Task 1: The Enchanted Fork and Exec Journey

In the mystical land of Codeor, a legendary alchemist named Forkenstern possessed the ancient arts of "fork()" and "exec()". These arts could create magical beings and transmute them into extraordinary forms. Your quest is to script this enchanted journey!

Write a C program named task1.c that utilizes the powers of Forkenstern to create a child process using fork() and then transmute it into another magical being using exec(). The program should follow these steps:

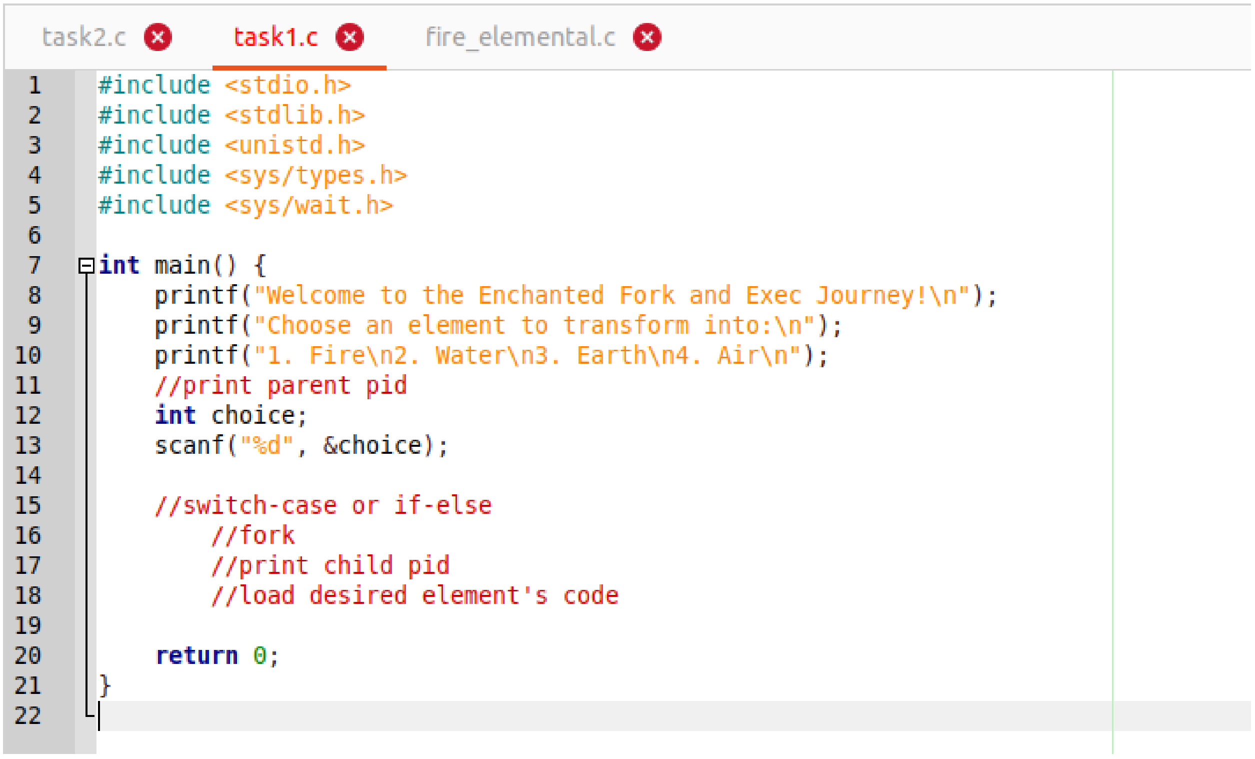
1. Fork a child process.
2. In the child process, execute a program (e.g., print a message) using exec() or execv()
3. The user should be able to choose the type of transformation they want to perform (e.g., fire, water, earth, or air).
4. Based on the user's choice, exec() should transform the child process into the chosen elemental being.
5. Each element has its own separate c code. The **fire_elemental.c** is given below for your reference. Please remember you have to compile and build each element code first before they are ready to be loaded from the **task1.c**.



```c
#include <stdio.h>

int main() {
    printf("Transformation complete! You are now the Fire Elemental!\n");
    return 0;
}
```

The code structure and comments for **task1.c** are provided below for your help.



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

int main() {
    printf("Welcome to the Enchanted Fork and Exec Journey!\n");
    printf("Choose an element to transform into:\n");
    printf("1. Fire\n2. Water\n3. Earth\n4. Air\n");
    //print parent pid
    int choice;
    scanf("%d", &choice);

    //switch-case or if-else
        //fork
        //print child pid
        //load desired element's code

    return 0;
}
```

Task 2: Threads of Harmony and Cooperation

In the mystical realm of Numerica, an age-old ritual unfolds, summoning magical beings known as "Factorions" to compute factorials of specific numbers. However, in this enchanting tale, a twist awaits! The revered sorcerer, the "Parent," yearns to partake in this mystical calculation.

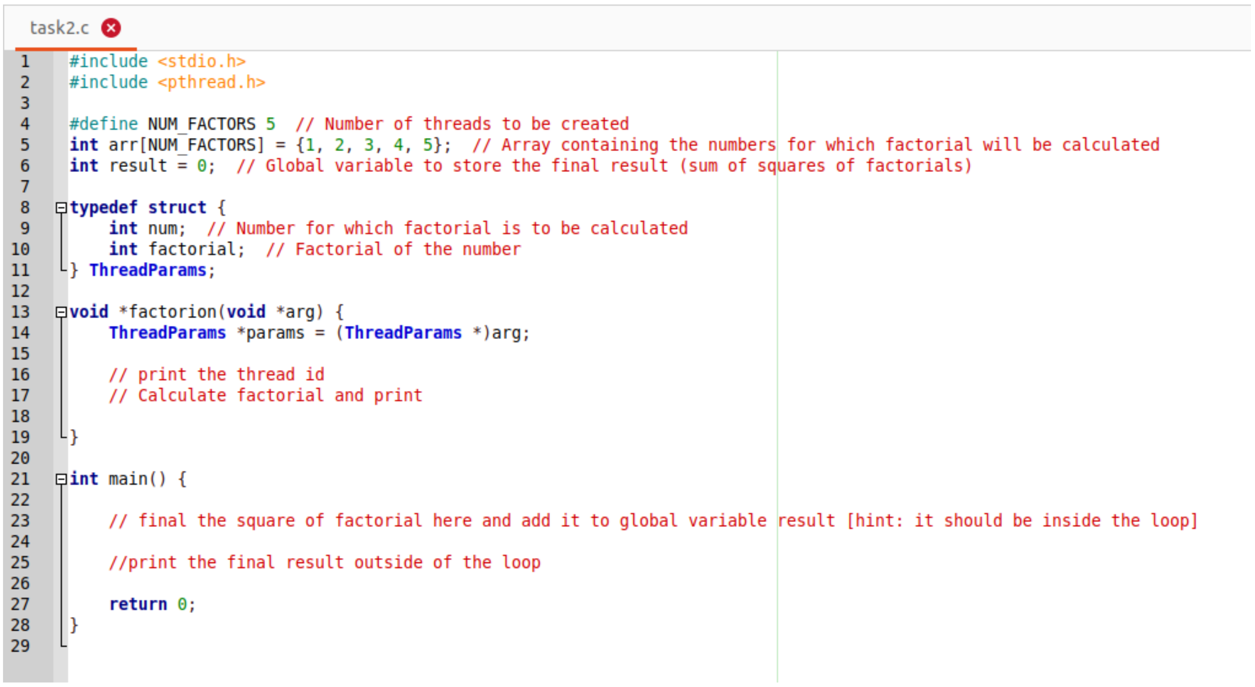
Task: Write a captivating C program named task2.c that breathes life into this mystical tale. The program must follow these steps:

1. Create five Factorions (threads), each assigned to compute the factorial of a specific number from an array named "arr".
2. The arr consists of numbers [1, 2, 3, 4, 5].
3. Simultaneously, the Parent, who conjured these Factorions, desires to perform a magical operation on the result returned by each Factorion. The operation involves calculating the summation of the squares of the factorial results. The Parent should execute this operation and display the final result.

Additional Guidelines:

- Ensure that each Factorion independently computes the factorial of its designated number and stores the value using the ThreadParam struct.
- Utilize a global variable "result" to store the result and display it as output.

The code structure and comments for task2.c are provided below for your help.



task2.c

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

#define NUM_FACTORS 5  // Number of threads to be created
int arr[NUM_FACTORS] = {1, 2, 3, 4, 5};  // Array containing the numbers for which factorial will be calculated
int result = 0;  // Global variable to store the final result (sum of squares of factorials)

typedef struct {
    int num;  // Number for which factorial is to be calculated
    int factorial;  // Factorial of the number
} ThreadParams;

void *factorion(void *arg) {
    ThreadParams *params = (ThreadParams *)arg;

    // print the thread id
    // Calculate factorial and print

}

int main() {

    // final the square of factorial here and add it to global variable result [hint: it should be inside the loop]

    //print the final result outside of the loop

    return 0;
}

```

Task 3: Interprocess Communication

Description: The operating system provides mechanisms for facilitating communications and data sharing between applications. Collectively, the activities enabled by these mechanisms are called Interprocess communications (IPC). Some forms of IPC facilitate the division of labor among several specialized processes. Other forms of IPC facilitate the division of labor among computers on a network. Typically, applications can use IPC categorized as clients or servers. A client is an application or a process that requests a service from some other application or process. A server is an application or a process that responds to a client request. Many applications act as both a client and a server, depending on the situation. The purpose of this assignment is for students to gain practical experience with the mechanism of Interprocess communication.

- Skills: Some of the practical skills students will gain from this assignment include:
    - Learning how design and develop programs that creates IPC objects
- Knowledge: This assignment will also help students become familiar with:
    - The mechanism of Interprocess Communication

OS encompasses a diverse range of IPC mechanisms, numbering more than ten. This assignment focuses on three key mechanisms:

Pipes
Remote Procedure Call (RPC)
Sockets

Your task is to develop a **client/server application** in **C or C++** that uses one of the **IPC (Inter-Process Communication) mechanisms** listed above.

- You must explicitly **create or establish the IPC channel** between the client and server processes (or between parent and child).
- Once the IPC is set up, each process should **send at least one message** to the other, demonstrating successful **two-way communication**.
- The application should run on **Windows**.

Submission Guidelines:

1. Please submit the **source code files** (C/C++ files) along with a **documentation file** (in **doc/pdf/txt** format) explaining how the instructor can compile and run your program. All files must be **compressed into a zip file**. Failure to include a proper instruction file will result in a **1-mark deduction**, and submitting files outside of a zip archive will also incur a **1-mark penalty**. **Follow the submission instructions carefully** to avoid these penalties.

2. Tasks **1** and **2** are specifically designed for **Unix-based operating systems**, such as **Mac** or **Ubuntu**, and require the use of the **pThread library**. Ensure that your source code runs correctly on a Unix system, as I will be testing it on **Ubuntu**. **Solutions designed for Windows will not be accepted for these tasks**.

3. **Task 3** is designed for **Windows systems**, and submissions will be tested on **Windows 10/11**. Please verify that your code runs properly on these platforms.

4. **Late submissions** will incur a maximum penalty of **50% of the marks**, so make sure to submit your assignment on time.

I generally don't grant extensions, and I would like to clarify why:

1. Dropped Lowest Assignments: Since I drop the two lowest assignment marks, you can still achieve full marks even if you miss two assignments.
2. Late Submissions Allowed with Penalty: I allow late submissions throughout the semester with a 50% penalty. This means you can still submit any assignment after the deadline until the last week and earn half credit. However, you must inform me because I don't check submission logs daily, and I might miss your late submission otherwise.
3. Extensions Disrupt the Course Timeline: We need to complete 8 assignments throughout the semester. If extensions are granted for the first few assignments, that means less time to complete the remaining ones. Additionally, both the TA and I have exam week responsibilities, so

we prefer to finish grading assignments early to free up time for final exams and project evaluations. Completing assignments on time also helps you understand where you stand before the final exam and project deadlines.