CS-3352-01
Assignment 2

Read the instructions very carefully and submit the assignment in the format asked. Failing to follow the instructions may lead to the deduction of marks.

Instructions:

1. Assignment due: **Oct 24th, 2024, By 11:59 pm**.

2. Each individual should implement the code on your own. You may not share final solutions with other individual, and you must write up your own work, expressing it in your own words/notation.

3. This assignment constitutes **10% of the final grade**.

4. The reports should be submitted if asked in the question.

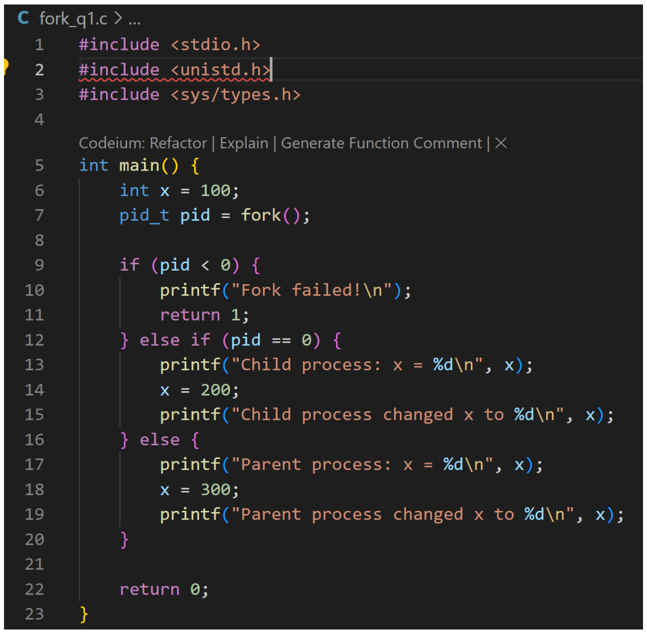
5. Assignment should be sent by zipping it into a single file (both codes and reports) through email only. Drive links are also acceptable. **No WTClass submission will be accepted.**

6. The email Subject should have this formatting:

Subject: CS-3352 (Assignment 2): Your Name

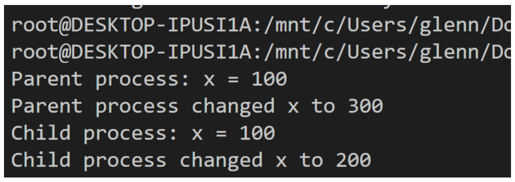
Problem 1: (10 points)

Write a program that calls fork (). Before calling fork (), have the main process access a variable (e.g., x) and set its value to something (e.g., 100). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of x? Print appropriate strings and values to signify which process is executing the code.



fork_q1.c > ...

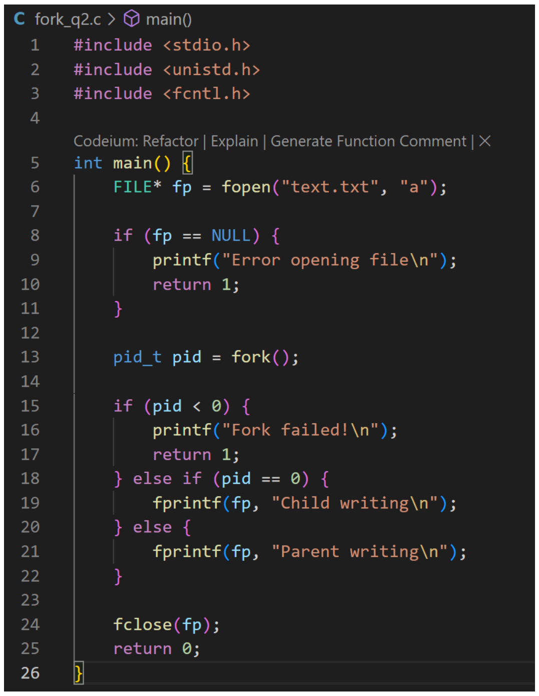
```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

Codeium: Refactor | Explain | Generate Function Comment | ×
int main() {
    int x = 100;
    pid_t pid = fork();

    if (pid < 0) {
        printf("Fork failed!\n");
        return 1;
    } else if (pid == 0) {
        printf("Child process: x = %d\n", x);
        x = 200;
        printf("Child process changed x to %d\n", x);
    } else {
        printf("Parent process: x = %d\n", x);
        x = 300;
        printf("Parent process changed x to %d\n", x);
    }

    return 0;
}
```



```
root@DESKTOP-IPUSI1A:/mnt/c/Users/glenn/Do
root@DESKTOP-IPUSI1A:/mnt/c/Users/glenn/Do
Parent process: x = 100
Parent process changed x to 300
Child process: x = 100
Child process changed x to 200
```

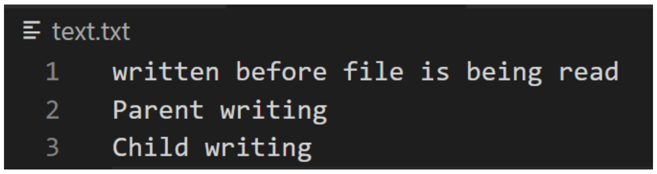
Problem 2: (10 points)

Write a program that opens a file (with the open () system call) and then calls fork () to create a new process. Can both the child and parent access the file descriptor returned by open ()? What happens when they are writing to the file concurrently, i.e., at the same time?



fork_q2.c > main()

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

Codeium: Refactor | Explain | Generate Function Comment | ×
int main() {
    FILE* fp = fopen("text.txt", "a");

    if (fp == NULL) {
        printf("Error opening file\n");
        return 1;
    }

    pid_t pid = fork();

    if (pid < 0) {
        printf("Fork failed!\n");
        return 1;
    } else if (pid == 0) {
        fprintf(fp, "Child writing\n");
    } else {
        fprintf(fp, "Parent writing\n");
    }

    fclose(fp);
    return 0;
}
```

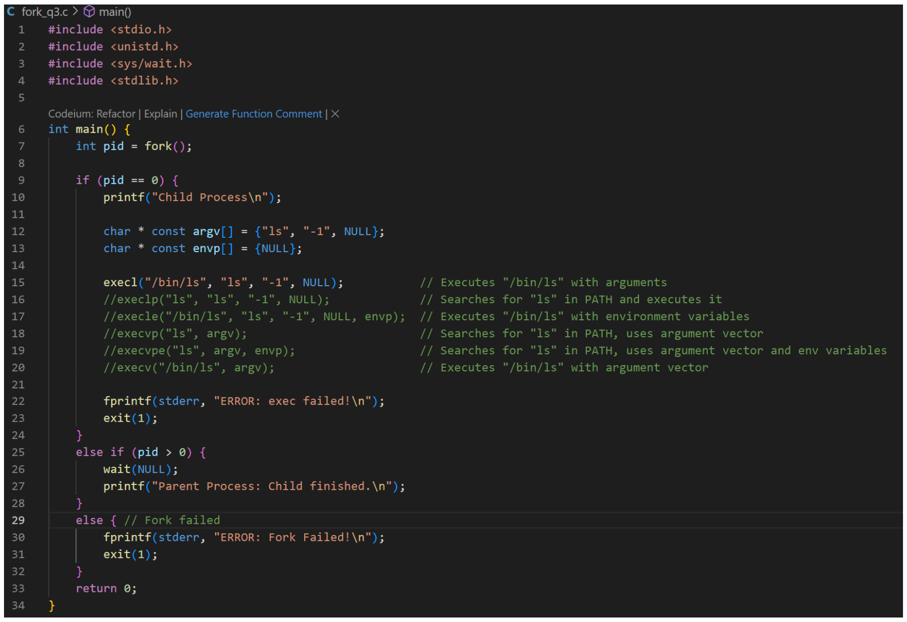


text.txt

```
written before file is being read
Parent writing
Child writing
```

Problem 3: (10 points)

Write a program that calls fork () and then calls some form of exec () to run the program (May be somewhere in : /bin/ls). See if you can try all of the variants of exec (), including (on Linux) execl(), execle(), execlp(), execv(), execvp(), and execvpe(). Why do you think there are so many variants of the same basic call?



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

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

    if (pid == 0) {
        printf("Child Process\n");

        char * const argv[] = {"ls", "-1", NULL};
        char * const envp[] = {NULL};

        execl("/bin/ls", "ls", "-1", NULL);           // Executes "/bin/ls" with arguments
        //execlp("ls", "ls", "-1", NULL);             // Searches for "ls" in PATH and executes it
        //execle("/bin/ls", "ls", "-1", NULL, envp);  // Executes "/bin/ls" with environment variables
        //execvp("ls", argv);                         // Searches for "ls" in PATH, uses argument vector
        //execvpe("ls", argv, envp);                  // Searches for "ls" in PATH, uses argument vector and env variables
        //execv("/bin/ls", argv);                     // Executes "/bin/ls" with argument vector

        fprintf(stderr, "ERROR: exec failed!\n");
        exit(1);
    }
    else if (pid > 0) {
        wait(NULL);
        printf("Parent Process: Child finished.\n");
    }
    else { // Fork failed
        fprintf(stderr, "ERROR: Fork Failed!\n");
        exit(1);
    }
    return 0;
}
```

There are many exec() variants for different use cases:

Argument Handling: l (list) variants take fixed arguments, while v (vector) variants use dynamic arrays making them better for different structures.

Path Resolution: p variants search for the program in the system's PATH simplifying execution when the full path is unknown.

Environment Passing: e variants allow custom environment variables to be passed which is useful for running programs in specific environments.

Problem 4: (5 points)

A process makes a system call to read a packet from the network device, and blocks. The scheduler then context-switches this process out. Is this an example of a voluntary context switch or an involuntary context switch? Explain

This is an example of an involuntary context switch because the process is blocked on a system call to read a packet from the network device. This means the process is waiting for the network device to provide data/info. The scheduler then decides to context switch this process out which means the scheduler is switching the CPU to another process. This is an involuntary context switch because the process is not able to continue executing until the network device provides data. Since the process does not choose to give up the CPU but is instead switched out because it is waiting on I/O, this is an involuntary context switch.

Problem 5: (8 points)

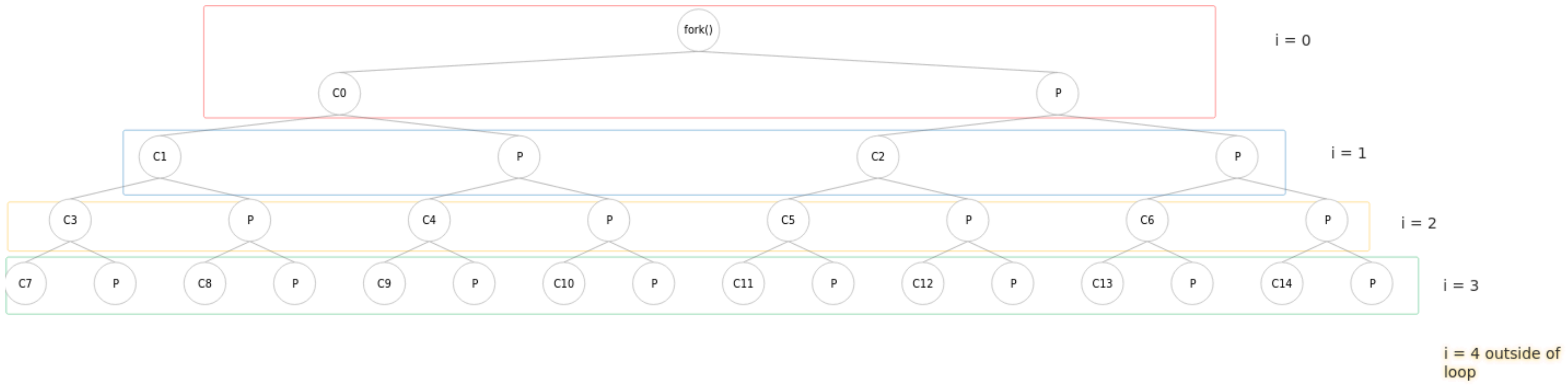
Consider the following C program. Assume there are no syntax errors and the program execute correctly. Assume the fork system calls succeed. What is the output printed to the screen when we execute the below program? Make the process tree structure to explain your answer.

```
void main(argc, argv) {
        for(int i = 0; i < 4; i++) {
        int ret = fork();
        if(ret == 0)
                printf("child %d\n", i);
        }
}
```

Output:

```
child 0
child 1
child 1
child 2
child 2
child 2
child 2
child 3
child 3
child 3
child 3
child 3
child 3
child 3
child 3
```

Tree on next page.



Problem 6: (8 points)

A computer has 4 GB of RAM of which the operating system occupies 512 MB. The processes are all 256 MB (for simplicity) and have the same characteristics. If the goal is 99% CPU utilization, what is the maximum I/O wait that can be tolerated?

RAM: 4GB (4096 MB)

RAM used by OS: 512

RAM available for processes: 4096 - 512 = 3584 MB

Each process: 256 MB

Number of processes: 3584 MB / 256 MB = 14 processes

Fraction of time each process is using CPU: 1 - p

Goal: 99% CPU utilization

CPU utilization = 1 - p^n

0.99 = 1 - p^14

p^14 = 1 - 0.99

p^14 = 0.01

p = .719…

p = .72 = 72%

We can tolerate a process up to 72% I/O wait time.

Problem 7: (4 points)

What is the biggest advantage of implementing threads in user space? What is the biggest disadvantage?

Advantages:

Faster Context Switching: Because user-space threads do not require kernel intervention, context switching between threads is much faster. This results in better performance, especially for applications that require many threads or need to switch a lot.

Customizable Scheduling: User-space threads give applications more flexibility on deciding how to manage and schedule their threads, allowing them to be tailored to the specific needs of a task.

Disadvantages:

Blocking Issues: If a user-space thread performs an operation that waits for a task, the entire program, and all its threads, can get stuck because the system doesn't know about each individual threads. This results in inefficient use of the CPU.

Problem 8: (10 points)

Consider a multiprocessor system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads N be larger than P the number of processors. Call K the number of kernel threads allocated to the program. Discuss the performance implications of the following scenarios:

1. $K < P$

   Wasted Processors: There aren't enough kernel threads to keep all the processors busy, so some processors won't have anything to do, which wastes potential power.

   Limited Task Execution: Since fewer kernel threads are available, not many tasks (user threads) can run at the same time, which would slow things down.

2. $K = P$

   Full Use of Processors: Every processor has a kernel thread to work on, making the most of available computing power. This setup makes sure everything runs efficiently.

   Limited to Processor Count: However, the number of tasks running at once is still limited to the number of processors, which could cause delays if there are many tasks.

3. $P < K < N$

Better Balance: With more kernel threads than processors, if one task gets stuck waiting, another task can take over which keeps the processors busy.

Some Overhead: More kernel threads mean the system needs to manage them, which could slow things down a little, but it's better for handling many tasks at once.

Problem 9: (5 points)

Under what circumstance s does a multithreaded solution using multiple kernel threads provide better performance than a single – threaded solution on a single - processor system?

When a multithreaded program faces a delay, like waiting for memory, the system can switch to another thread to stay working. A single-thread program has to wait during this time. When you have frequent delays or system waits, multithreading improves performance, even on a single processor, by making better use of idle time.

Problem 10: (5 points)

Can a multithreaded solution using multiple user - level threads achieve better performance on a multiprocessor system than on a single -processor system

A multithreaded solution using multiple user-level threads cannot achieve better performance on a multiprocessor system than on a single-processor system by itself, because user-level threads are managed by the user-space library and are not visible to the kernel. This means the kernel sees the entire process as one, regardless of how many user-level threads it actually has.

Since the kernel doesn't know about the user-level threads, it cannot schedule them across multiple processors. As a result, even on a multiprocessor system, only one processor will be used for the entire process, leading to a significant performance improvement compared to a single-processor system.

To take advantage of a multiprocessor system, the program must use kernel-level threads or a hybrid model where user-level threads are mapped to kernel threads that the kernel can distribute across processors.