_tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp

Gebze Technical University
Computer Engineering

OPERATING SYSTEMS

Emre YILMAZ: 1901042606

Homework-1, May 2024

Profesor: Yusuf Sinan AKGÜL

1901042606

May 13, 2024

Contents

1 Important Notes

1. You call me whenever you want, +905319346629
2. I show a lot of screenshots but some tasks need to be shown as video. So, I put a lot of video links in sections. You reach them by clicking some link. You can show some texts like "Click here". Click them and watch the videos.
3. Also, you can see all videos from this link: https://drive.google.com/drive/folders/1iZ4MM2mWToyBvjFFXlMLW
RZ3AuL49?usp=sharing
4. First, I explained all the syscalls such as fork, execve, and exit. Then, I demonstrated that they work. After that, I showed each strategy individually. You can verify that Part-1 works from Part-2, and that Part-2 works from Part-3. However, I still provided evidence for each one individually.
5. In Part-2, and Part-1, I found it inappropriate to run all strategies at once as it becomes impossible to observe. Therefore, while one strategy is active in the code, the other strategies are commented out. If you read the Readme files, you can properly remove these comments and run each strategy individually. This approach helps in better understanding and evaluating each strategy's behavior and performance without interference from others.
6. I run all tasks by forking. Also, I sometimes use execve function to start a function.
7. Best Regards.

2 System Call Implementations

2.1 Fork

2.1.1 Fork Algorithm

```
common::uint32_t TaskManager::ForkTask(CPUState *cpustate)
{
    if(numTasks >= 256)
        return -1;

    if(tasks[currentTask].forked)
    {
        tasks[currentTask].forked = false;
        return 0;
    }

    tasks[numTasks].state = TASK_READY;
    tasks[numTasks].pPid = tasks[currentTask].pid;
    tasks[numTasks].pid = nextpid++;

    for (int i = 0; i < sizeof(tasks[currentTask].stack); i++)
    {
        tasks[numTasks].stack[i] = tasks[currentTask].stack[i];
    }

    common::uint32_t currentTaskOffset = (((common::uint32_t)cpustate) - ((common::uint32_t)tasks[currentTask].stack));
    tasks[numTasks].cpustate = (CPUState*)(tasks[numTasks].stack + currentTaskOffset);

    // tasks[numTasks].cpustate -> ecx = 0;
    tasks[numTasks].forked = true;
    tasks[numTasks].cpustate->eip -= 2;
    numTasks++;

    return tasks[numTasks-1].pid;
}
```

Fig 1: Fork Code

Firstly, I would like to emphasize that I do not use any helper function for the return value of the fork function. Fork returns a value to both the parent and the child; one of these values is the child's pid, while the other is 0.

The code Ilhan Aytutuldu posted on Teams generally resembles the following: The stack of the parent task is copied as is, and the child task's cpustate (i.e., corresponding to the stack pointer) is modified to the extent of the parent's. This ensures that they are both positioned at the exact same stack location.

However, significant changes have been made:

1. The shared algorithm with the fork function cannot return two different values. Therefore, a boolean value "forked" has been added to the Task class.

2. Also, the instruction pointer of the child task has been moved back by 2 bytes. This is the critical point because with this change to the instruction pointer, while the parent returns to the called place, the child re-enters the Fork function. But as you can see in my code, if the "forked" boolean value of the task entering the fork is true, it directly returns 0, thus allowing the main task to understand that the returned value is from the child task. After this boolean check, the "forked" boolean value of the child is set back to false, making it eligible for forking itself.

General Flow of Fork: The fork call first invokes the assembly function in the syscall.cpp file, which triggers an interrupt using assembly code. When the trap is set to the 0x80 interrupt, the

value 2, which is the syscall value for fork, is written into the eax register. Subsequently, the syscall handler function calls the fork function in the TaskManager class, as shown in Fig-1 based on the value in the eax register.

2.1.2 Fork Test

```c
void Test1()
{
    int test = -1;
    int pid;
    pid = fork(&test);

    if(pid==0)
    {
        printf("Child is started.\n");
        test = 23;
        printfHex(test);
        printf("\n");
        for(int i=0;i<1000000000;i++);
        exit();
    }

    else
    {
        // waitpid(pid);
        printf("Parent is started.\n");
        test = 31;
        printfHex(test);
        printf("\n");
        // execve(taskB);
    }

    printf("Parent is terminating...\n");
    exit();

}
```

Fig 2: Fork Test

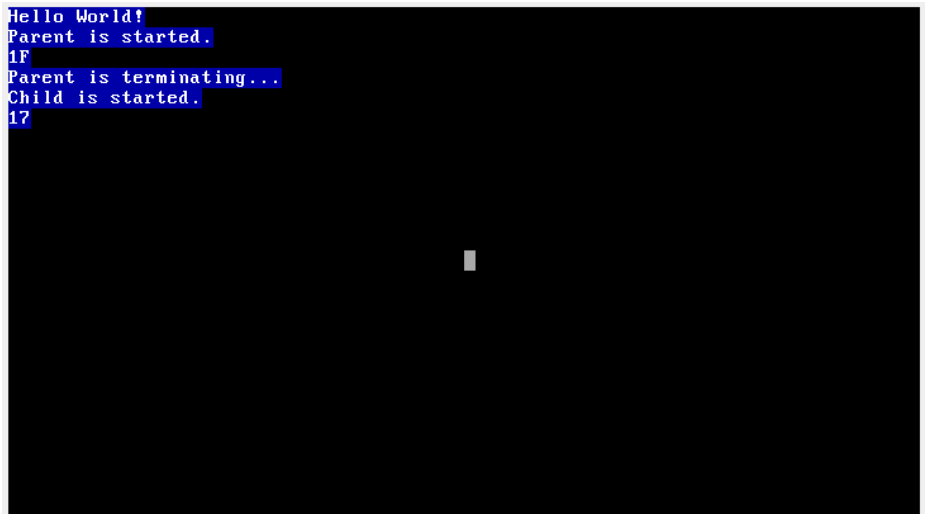


Fig 3: Fork Result

As you can see, the fork is working successfully. When the task forks, if pid==0, meaning the child is running, it prints a message indicating that it is running, modifies a test variable, prints it, and then terminates the process with EXIT.

The parent, on the other hand, continues and prints a message indicating that it is continuing. It updates the test value to 31, prints it as well.

I must emphasize that the print values are in hexadecimal. Moreover, both tasks update the same variable name and we see different values printed for each, indicating that the stack separation is also being successfully implemented.

2.2 Exit

2.2.1 Exit Algorithm

The exit syscall is extremely simple compared to fork. We again throw a syscall trap with asm. The exit function in the TaskManager runs. Here, the state of the currently running task (tasks[currentTask]) is updated to TERMINATED. Additionally, the scheduler skips over tasks that are terminated when making decisions, so that task will not run again.

2.2.2 Exit Test

When you examine Fig-2 and Fig-3 in the fork section examples, you can see that the exit syscall is working successfully because the string "Parent is terminating" is printed only once. This proves that the child did not reach that part and was marked as terminated by the exit call.

2.3 WaitPID

2.3.1 WaitPID Algorithm

```
bool TaskManager::WaitPID(common::uint32_t pid, CPUState* cpu)
{
    int index = getIndex(pid);
    if(index > -1)
    {
        tasks[currentTask].state = TASK_WAITING;
        tasks[currentTask].waitPid = pid;
        return true;
    }

    return false;
}
```

Fig 4: WaitPID Code

The wait pid system call, unlike exit and fork, operates with a parameter. The assembly code again triggers the syscall handler, and along with the cpu state, it also passes the PID parameter. This PID parameter is then passed to the waitpid function in the TaskManager.

The waitpid function in the TaskManager, as you can see in the figure-4 above, changes the state of the currently running task to WAITING, or blocked, and places the information on which task it is waiting for (the waitpid value in the parameter) into the waitpid parameter within the Task class.

Of course, for this mechanism to work, updating the scheduler is essential. There, while searching for tasks that are READY, if a task is found to be WAITING or blocked, the pid it is waiting for is checked. If the task being waited for has terminated, the task that is WAITING is moved back to READY and scheduled again. You can see the scheduler and changed part in Fig-5.

```
CPUState* TaskManager::Schedule(CPUState* cpustate)
{
    if(numTasks <= 0)
        return cpustate;

    if(currentTask >= 0)
        tasks[currentTask].cpustate = cpustate;

    int searchedTask=(currentTask+1)%numTasks;

    while(tasks[searchedTask].state != TASK_READY)
    {
        if(tasks[searchedTask].state == TASK_WAITING && tasks[tasks[searchedTask].waitPid-1].state == TASK_TERMINATED)
        {
            tasks[searchedTask].state = TASK_READY;
            tasks[searchedTask].waitPid = 0;
            break;
        }
        searchedTask=(searchedTask+1)%numTasks;
    }

    // PrintAll();
    currentTask = searchedTask;
    return tasks[currentTask].cpustate;
}
```

Fig 5: Scheduler Code

2.3.2 WaitPID Test

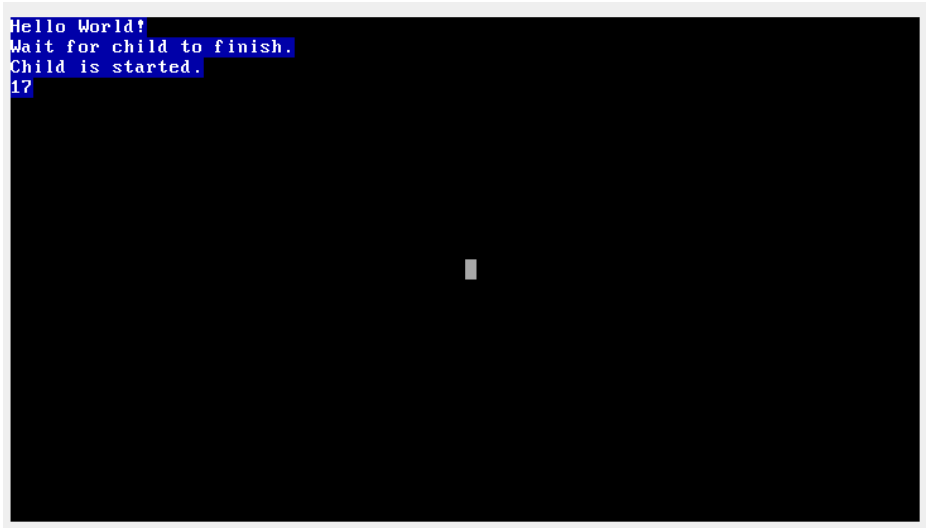


Fig 6: Parent Process does not start yet since child is not finished

Fig 6: Parent just started since child is finished

Click here to watch the WaitPID Test Video

As you see in the video above, parent is waiting for a time to child is finished. Then it continues.

2.4 Execve

2.4.1 Execve Algorithm

The execve function operates on a simple logic. This syscall takes an entry point as a parameter. It terminates the currently running task, adds a new task while placing the EIP (Entry Instruction Pointer) from the parameter and schedule it, thus completely changes the task that is running.

2.4.2 Execve Test

```
void ExecveTest()
{
    int test = -1;
    int pid;
    pid = fork(&test);

    if(pid==0)
    {
        printf("Child is started.\n");
        test = 23;
        printfHex(test);
        printf("\n");
        for(int i=0;i<1000000000;i++);
        exit();
    }

    else
    {
        // waitpid(pid);
        printf("Parent is started.\n");
        test = 31;
        printfHex(test);
        printf("\n");
        execve(printA);
    }

    printf("Parent is terminating...\n");
    exit();
}
```

Fig 7: Execve Test Code

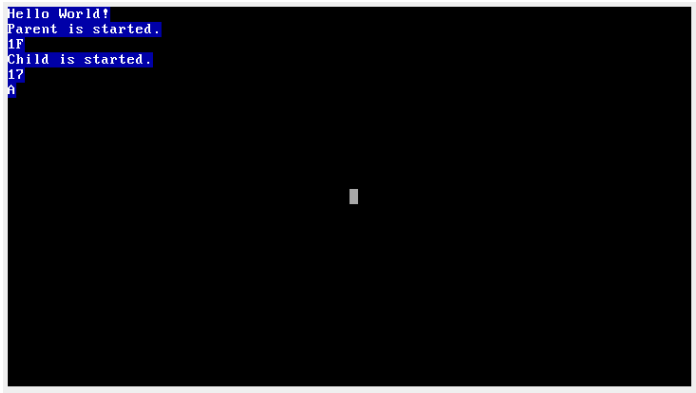


Fig 8: Execve Result

As you can see in Figure-7 and 8, after the child finishes, the parent continues and gives a function that prints the character "A" as the entry point to execve. You can tell that execve is functioning from the printed "A" character and from the fact that you never see the phrase "parent process is terminating...". The child has exited, and the parent has completely changed the core image with execve. As a result, there are no functions left that reach the phrase "parent process is terminating".

3 Part-A Strategy Test

Click here to watch the PART-A-STRATEGY Test Video

In the strategy, the requirement is for each of the two tasks to be loaded three times each. You can see this in the function on line 313 of the kernel.cpp file in the Part-1 folder, resulting in a total of six forks. When you watch the video, you will observe that the scheduler prints the current processes at every interrupt, and this printing process stops when no task is ready to be executed. After all tasks have completed, you can see that the printing in the scheduler has ended in the video. One task still appears in the ready state because the scheduler does not run again after it finishes, so we can't see its completion in the final process table. However, since the scheduler does not print the process table again, we can conclude that all tasks have successfully finished. When child processes complete running either the long_running_program or the collatz function, they exit, updating their state to TERMINATED.

Moreover, you can observe that there are a total of seven processes. One is the parent, and the other six are the forked child processes.

Additionally, since the process table is printed in every scheduling action, it might be challenging to follow the prints from the long_running_program and the collatz functions. However, if you look carefully, you can catch these prints in between. Alternatively, you could comment out the line that prints the process table in the "multitasking.cpp" file (line 204) in the demo to better follow the prints of the programs.

In the end, as the scheduler stops after all tasks have completed, we confirm that all six children have finished their tasks and successfully exited.

Additionally, the strategy required in PART-A can also be observed in strategies 1 and 2 of PART-B. You can look into those as well.

4 PART - B

4.1 Modifications on Task Class

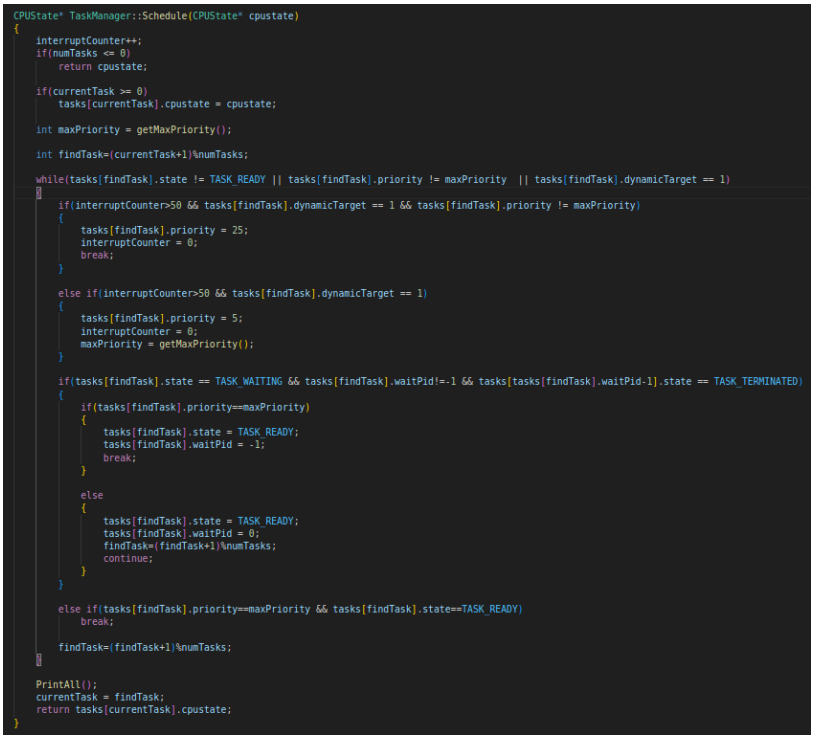
1. our tasks will have priorities, so an integer value for priority has been added to the task class.
2. For Strategy-4, we need to indicate a task's priority willbe set dynamically or not. So, a isDynamic class member is added to Task class
3. Related getter and setter functions are written for Task class

4.2 Modifications on TaskManager Class and Written New System Calls

1. InterruptCounter is added to TaskManager to decrease and increase the priority of dynamic task priority targets.
2. setPriority system call is added to set a priority for a task.
3. getPriority system cal is added to get the priority of a task.
4. ExecveLow system call is added for strategy-4 and strategy-3. It changes core image of a task and add new one like execve. The difference between execve and execveLow is priority of new task. Execve puts lower priority than default one to Task.
5. ExecveHigh system call is added for strategy-4 and strategy-3. It changes core image of a task and add new one like execve. The difference between execve and execveLow is priority of new task. Execve puts higher priority than default one to Task.
6. getIndex function is added to TaskManager. It gets a pid and returns its index in Task list
7. getMaxPriority function is added to TaskManager. It is used in scheduler to find out max priority in the task lists. But it ignores TERMINATED tasks.
8. **IMPORTANT NOTE**: The default priority is set 10. It means that when we add Task its priority is 10.
9. **IMPORTANT NOTE**: When you fork, child process will have the same priority as its parent.

4.3 Modifications on Scheduler

In this part, the scheduler is somewhat complex. As you can see in the screenshot, the scheduler operates simultaneously for both Strategy-3 and Strategy-4. However, I prefer to explain it in two subsections.



```
CPUState* TaskManager::Schedule(CPUState* cpustate)
{
    interruptCounter++;
    if(numTasks <= 0)
        return cpustate;

    if(currentTask >= 0)
        tasks[currentTask].cpustate = cpustate;

    int maxPriority = getMaxPriority();

    int findTask=(currentTask+1)%numTasks;

    while(tasks[findTask].state != TASK_READY || tasks[findTask].priority != maxPriority  || tasks[findTask].dynamicTarget == 1)
    {
        if(interruptCounter>50 && tasks[findTask].dynamicTarget == 1 && tasks[findTask].priority != maxPriority)
        {
            tasks[findTask].priority = 25;
            interruptCounter = 0;
            break;
        }

        else if(interruptCounter>50 && tasks[findTask].dynamicTarget == 1)
        {
            tasks[findTask].priority = 5;
            interruptCounter = 0;
            maxPriority = getMaxPriority();
        }

        if(tasks[findTask].state == TASK_WAITING && tasks[findTask].waitPid!=-1 && tasks[tasks[findTask].waitPid-1].state == TASK_TERMINATED)
        {
            if(tasks[findTask].priority==maxPriority)
            {
                tasks[findTask].state = TASK_READY;
                tasks[findTask].waitPid = -1;
                break;
            }

            else
            {
                tasks[findTask].state = TASK_READY;
                tasks[findTask].waitPid = 0;
                findTask=(findTask+1)%numTasks;
                continue;
            }
        }

        else if(tasks[findTask].priority==maxPriority && tasks[findTask].state==TASK_READY)
            break;

        findTask=(findTask+1)%numTasks;
    }

    PrintAll();
    currentTask = findTask;
    return tasks[currentTask].cpustate;
}
```

Fig 9: Scheduler Part-B

4.3.1 Scheduler Modifications for Strategy-3

There is no dynamic priority adjustment in strategy-3. The task selected is expected to have the highest priority and be in a ready state. As seen in the code, the first two conditions of the while loop ensure this. If the candidate task is not in a ready state or its priority is not the maximum, it looks at another task. If a task is blocked, as in the first part, if the task it is waiting for is terminated, its block is lifted, and it is scheduled.

4.3.2 Scheduler Modifications for Strategy-4

The scheduler increases the interruptCounter by one each time it runs. Additionally, a third condition has been added to the while loop. If the candidate task is a dynamic target and the interrupt counter exceeds the limit, its priority is set higher than all other tasks. Similarly, if the candidate task is a dynamic target, is currently running, and the interrupt counter limit is exceeded, its priority is reset to be lower than all tasks again. This method ensures that tasks with dynamic priorities

are given a chance to execute according to their urgency and importance, which can dynamically change over time depending on their interaction with the system's state.

4.4 Strategy-1 and Strategy-2

Click here to watch the PART-B-STRATEGY-1 Test Video

Click here to watch the PART-B-STRATEGY-2 Test Video

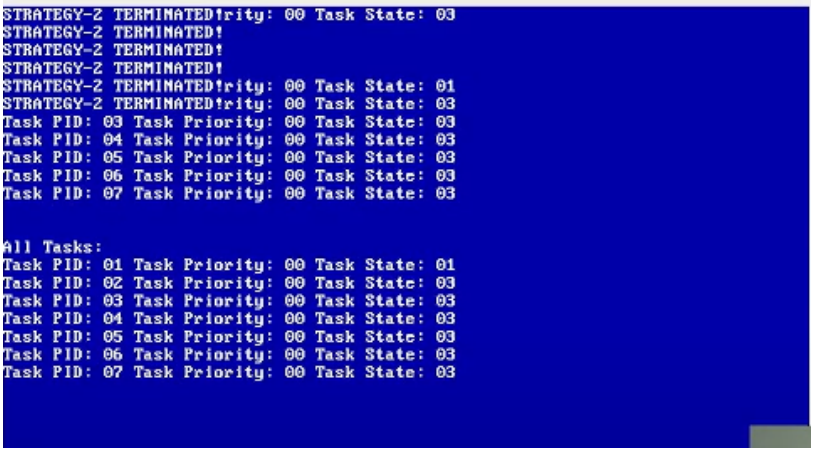


Fig 10: Last Print of Strategy-2 in PART-B

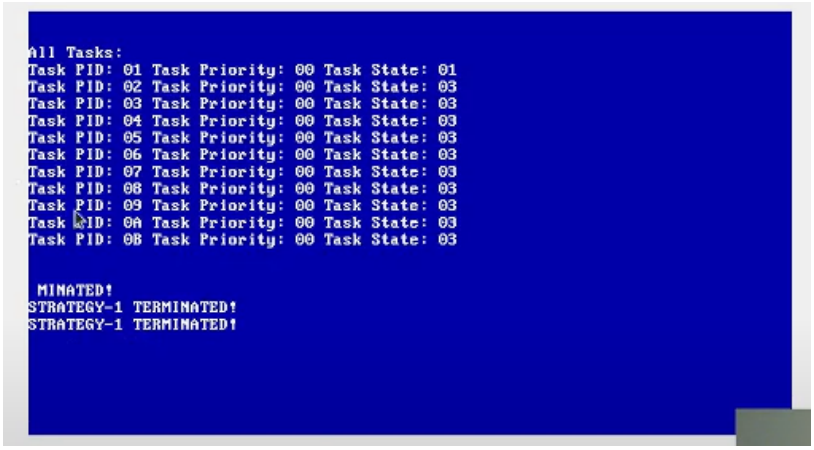


Fig 11: Last Print of Strategy-1 in PART-B

In this part, Strategy-1 and Strategy-2 are not logically different from PART-A. In both strategies, the necessary number of fork operations are performed and functions are run. For Strategy-1,

a binary search is selected, and for Strategy-2, a linear search is used. Strategies 3 and 4 will also include the Collatz and long-running programs.

In strategy-1 and strategy-2 all tasks have same proiority 10 as default since PDF does not say about priorities. You can watch the videos to see they work successfuly.

Again, you might see a task's status as READY in the final print. This is because the scheduler no longer runs after all processes have finished, so we don't see the final print. In reality, all tasks have actually completed, and the scheduler has stopped adding new tasks. This means that all tasks have successfully concluded. This behavior demonstrates the effective management and termination of tasks by the scheduler, ensuring that no tasks are left pending once all have been processed to completion. Also, you can see waitpid works successfuly.

4.5 Strategy-3

```
void TaskV23()
{
    int pid;
    pid = fork(&pid);

    if (pid == 0) {
        execve(printCollatz);
        exit();
    }

    else
    {
        printf("\nChildren:\n");
        int clockCounter;
        while(getInterruptCounter(&clockCounter) < 30);

        pid = fork(&pid);

        if (pid == 0) {
            int c_pid;
            setPriority(getPid(&c_pid), 20);
            int  child;
            long_running_program(50);
            exit();
        }
        printf("Parent pid: ");
        printfHex(getPid(&pid));
        printf("\n");

        pid = fork(&pid);

        if (pid == 0) {
            int c_pid;
            setPriority(getPid(&c_pid), 20);
            int  child;
            int test_array_bs[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
            binarySearch(test_array_bs, 10, 5);
            exit();
        }

        pid = fork(&pid);

        if (pid == 0) {
            int c_pid;
            setPriority(getPid(&c_pid), 20);
            int  child;
            int test_array_ls[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
            linearSearch(test_array_ls, 10, 5);
            exit();
        }
    }
    exit();
}
```

Fig 12: Strategy-3 Code of Part-B

As you can see, the process starts, a fork operation is performed, and the first child begins working through an execve call. By default, execve gives this process a priority of 10 (the same as the parent). The parent waits until the interrupt counter exceeds 20. Once it surpasses 20, three more tasks are loaded, and syscalls are made to ensure that these three have a higher priority than the first child. The expected behavior is for the first child to start running first, and then, after the 20th interrupt, the other tasks are added with a higher priority than the first child. In this scenario, after the 20th interrupt, the three newly added tasks will run, and the first child will not. After these three tasks finish, the first child should resume running.

It is impossible to show this as screenshot, so, you must watch the video below. Click it and watch it. It shows strategy-3 is working Click here to watch the PART-B-STRATEGY-3 Test Video

4.6 Strategy-4

```
void TaskV24()
{
    int pid;
    pid = fork(&pid);

    if (pid == 0) {
        execveLow(taskD);
    }

    else
    {
        pid = fork(&pid);

        if (pid == 0) {
            while(1) printf("B");
            exit();
        }

        pid = fork(&pid);

        if (pid == 0) {
            while(1) printf("A");
            exit();
        }

        pid = fork(&pid);

        if (pid == 0) {
            while(1) printf("C");
            exit();
        }
    }
    exit();
}
```

Fig 13: Strategy-4 Code of Part-B

Here, we will make dynamic priority changes. When the interrupt counter exceeds a certain limit, the priority of the first child will increase, and when it exceeds that limit again, it will decrease.

The changes made to the scheduler for this strategy are critical. You can revisit that section (4.3.2) for details. Initially, when we perform the first fork, a task that prints the letter 'D' in an infinite loop begins to run due to an execve call but starts with a low priority (5) because we use the execveLow system call. The other three children forked and added subsequently begin operating with the default priority of 10. These tasks print the letters 'A', 'B', and 'C' respectively in an infinite loop. At first, because the first child has a low priority, we should not see the letter 'D' on the screen. After the interrupt counter exceeds the necessary limit, the priority of the first child printing 'A' is set higher than all other tasks. At this point, we should expect to see only 'D' being printed,

with the other tasks waiting in the ready position. And when the counter once again surpasses a certain limit, the priority of the task printing 'D' drops again, and we should see the letters 'A', 'B', and 'C' printed on the screen as initially.

By watching the video below, you can observe this behavior.

Click here to watch the PART-B-STRATEGY-4 Test Video

5 PART-C

5.1 Modifications on the class PrintfKeyboardEventHandler and General Algorithm

In this modified class, instead of displaying on the screen whatever key the user presses, it now stores each keystroke in a character array and keeps track of the number of elements in another variable. When the user presses the enter key, a boolean value "isEnterPressed" within the class is set to true.

On the task side, if the task is interactive, its priority is set to be the highest, and it continuously checks for the "isEnterPressed" value to become true in an infinite loop. Once it becomes true, the getIntegerArray function from the PrintKeyboardHandle class retrieves the user-entered, space-separated text as an integer array along with its counter. This function also resets the counter and the character array at the end, preparing the system to receive the next input.

This setup ensures that interactive tasks, which depend on user input, have the highest priority and can efficiently process input data immediately after it's entered, enhancing responsiveness and user interaction.

Additionally, there is a function written to print the character array, but it was not needed in this part. This function, although unused in the current scenario, remains part of the class and could be utilized in future implementations or for debugging purposes, providing flexibility in handling the character array data.

5.2 Test of PrintfKeyboardEventHandler

```
void part3_test()
{
    int pid;
    pid = fork(&pid);

    if(pid == 0)
    {
        taskD();
        exit();
    }

    pid = fork(&pid);

    if(pid == 0)
    {
        long_running_program();
        exit();
    }

    pid = fork(&pid);

    if(pid == 0)
    {
        binarySearch();
        exit();
    }

    pid = fork(&pid);

    if(pid == 0)
    {
        linearSearch();
        exit();
    }

    exit();
}
```

Fig 14: Test Code of Part-C

In here, binary search, long running progam and linearSearch are interactive processes. They are waiting input from user.

```
setPriority(getPid(0), 20);
printf("\nGive me the number: ");
while(!kbhandler.getEnterPressed())
```

```
{
}

printf("\n");

int ct;
int array[25];

kbhandler.getIntegerArray(array, &ct);
```

This code segment above, is beginning of an interactive process. It increases the priority of current process, and waits for input. When user presses enter, it breaks the loop and continue to work and exit.

It is impossible to show this task as screenshots. You should watch the video below.
Click here to watch the PART-B-STRATEGY-4 Test Video

As you see from the video, when interactive tasks start, the task that infinitly prints "D" stops since its priority is lower than interactive process. After all interactive processes finish, printing D task continues.