Operating System Assignment Report

Eren Çakar
1901042656

May 13, 2024

1 General Details

It is to be noted that any and all numbers seen on screen are in hexadecimal format and not int decimal format. As it is implemented in the original source code.

2 General Additions

2.1 Screen Wait

The assignment demands tremendous amounts of printing of values and the screen size of the operating system is relatively small and fills quickly. To make reading the printed messages easier where it's appropriate some additions and changes were made.

The print keyboard handler in the `kernel.cpp` is upgraded with new fields and methods for saving pressed keys to a buffer and then retrieving it.

The content of `printf` function was also changed to pausing the system whenever the screen is filled and the user has not yet pressed the enter key.

The changes are shown below with only writing the necessary parts where the unchanged parts are left unwritten and mentioned as comments.

```
// kernel.cpp
void printf(char *str)
{
    for (int i = 0; str[i] != '\0'; ++i)
    {
        // ...
        // print until screen is full
        // ...

        if (screeny >= 25) // screen is full
        {
            taskManager.WaitEnter();    // wait for enter press
            while (!newLinePressed)     // newLinePressed is set by keyboard interrupt
                ;
            newLinePressed = false;     // reset the global flag
            taskManager.SignalEnter();  // signal that enter has been pressed

            // ...
            // clear screen
            // ...
        }
    }
}
```

```
// kernel.cpp
class PrintfKeyboardEventHandler : public KeyboardEventHandler
{
public:
    void OnKeyDown(char c)
    {
        // ...
        // Handle the key press
        // ...
        if (c == '\n') newLinePressed = true;
    }
};
```

3 Part A

3.1 Task Description

Task for this part of the assignment was to implement various POSIX system calls and handle multi-programming using fork, waitpid, execve, and other necessary system calls.

3.2 Implementation Details

3.2.1 Scheduler Modifications

The scheduler is designed as Round Robin scheduler out of the box from the original source code, although it was not specified that Round Robin is needed for this part of the assignment. But to enable processes to wait other processes it is needed to introduce process states and modify the existing scheduler, the modified scheduler is shown at the end of this subsection in sub-subsection 3.3.

3.2.2 Fork

Fork functionality is implemented using implementation of its respective system call number in the `SyscallHander` class of the system. Then the `fork` function available for processes' usage in the system executes this syscall.

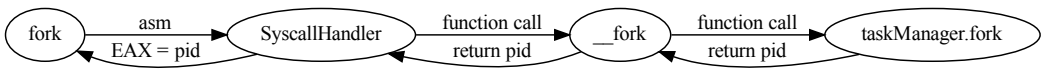
The syscall handler gets the current status of the system and passes it to the kernel side `fork` function. Which directs the call to the `taskManager` (already existing class) of the system.

The task manager's fork essentially initializes a new task with same process table, copies everything of the parent process down to the stack and CPU registers. Then it returns the newly created process's id.

The syscall handler receives the return value from the fork function of task manager and puts it into the `EAX` register of the CPU state of the **process which called the syscall**. New tasks are always created with their `EAX` registers (like others) set to 0.

At this point the syscall is handled and CPU is given back to the process, since the last called function was `fork` (one that is available to the processes), it takes the value in the `EAX` register and returns it. This equates to parent receiving child's `pid` and child receiving 0.

Visualization of `fork`:



Here is the pseudocode:

```
Function: fork()
Input: None
Output: int (pid)

Description:
    1. Invoke a system call interrupt with the fork operation code.
    2. Retrieve the result of the fork operation from the EAX register.
    3. Return the pid (process ID) obtained from the system call.
```

3.2.3 Waitpid

The `waitpid` function call its respective syscall number similar to `fork` function. This time however it puts the **pid** value to be waited into the `EBX` register of the current process's, so that the syscall handler can receiver information on it.

Then the syscall handler calls scheduler and scheduler changes current task's state to blocked and sets it's respective `waitPid` value to the value passed by syscall handler. And scheduler immediately starts to search a new *ready* process to run.

Anytime scheduler checks a process's state, if it's `BLOCKED` then it also checks for it's `waitPid` value to see if the process with this value is in state `EXITED`. If so the process's state is set to `RUNNING` and scheduler let's that process to be run on CPU.

Here is the pseudocode:

```
Function: waitpid(pid)
    Input: int pid (Process ID of the task to wait for)
    Output: None

    Description:
        1. Invoke a system call interrupt with the waitpid operation code.
        2. Pass the process ID (pid) as an argument in the EBX register.
```

3.2.4 Exit

To enable `waitpid` function to work `exit` is also implemented, which stops the current process and sets its state to `EXITED`.

Here is the pseudocode:

```
Function: exit()
    Input: None
    Output: None

    Description:
        1. Invoke a system call interrupt with the exit operation code.
```

3.2.5 Execve

Although `execve` functionality is practically the same as it is in C language, the implementation differs.

The `execve` function simply creates a new process with given parameters (it's essentially only the entry point, name of the user function in this case), and calls `waitpid` on it and immediately exits afterwards.

Reason for round about implementation is due to not engage with "deleting" or "overwriting" of existing process stacks, which proves to be difficult to implement safely.

Here is the pseudocode:

```
Function: execve(entrypoint)
    Input: Function pointer entrypoint (Entry point of the new task)
```

```
    Output: None

    Description:
        1. Create a new Task object with the provided entry point.
        2. Add the new task to the task manager.
        3. Wait for the task to finish execution using waitpid().
        4. Exit the current task.
```

3.3 Lifecycle

The loading and execution of multiple programs, including the Collatz sequence and the long-running program, involves several steps within the system.

When a program is loaded into memory, the system initializes the necessary data structures to manage its execution. For example, when a program implementing the Collatz sequence or the long-running program is loaded, the system sets up the process table entries and allocates memory for the program's code, data, and stack.

Once a program is loaded, the system enters a loop where it continuously schedules and executes processes based on their states. The scheduler, implemented within the system's task manager which uses Round Robin, selects a process from the ready queue to run on the CPU.

For programs executing the Collatz sequence or the long-running program, the system handles interrupts, including timer interrupts, to perform context switching between processes. Upon receiving a timer interrupt, the system checks the status of running processes and decides whether to switch to a new process or continue executing the current one.

During the execution of the Collatz sequence or the long-running program, the system may encounter waitpid system calls. These calls are handled by the syscall handler, which communicates with the scheduler to block the current process and search for a new ready process to run.

Overall, the system's lifecycle involves the continuous loading, scheduling, and execution of multiple programs, including the Collatz sequence and the long-running program, with proper handling of system calls, interrupts, and process states. This approach ensures efficient resource utilization and responsive execution of user programs.

See modified Round Robin scheduler implementation just below:

```
// multitasking.cpp
void myos::TaskManager::FindNextTask()
{
    while (true)
    {
        if (++currentTask >= numTasks)
            currentTask %= numTasks;

        if (tasks[currentTask]->state == TaskState::READY)
            break;

        if (tasks[currentTask]->state == TaskState::BLOCKED &&
            tasks[tasks[currentTask]->waitingPid]->state == TaskState::EXITED)
            break;
    }
    tasks[currentTask]->state = TaskState::RUNNING;
}

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

```
    if (currentTask >= 0)
    {
        if (cpustate->eax == 7) // waitpid
        {
            tasks[currentTask]->state = TaskState::BLOCKED;
            tasks[currentTask]->waitingPid = cpustate->ebx;
        }
        else if (cpustate->eax == 1) // exit
        {
            tasks[currentTask]->state = TaskState::EXITED;
        }
        else
        {
            tasks[currentTask]->state = TaskState::READY;
        }
        tasks[currentTask]->cpustate = cpustate; // Save the CPU state of the task
    }
    FindNextTask();
    return tasks[currentTask]->cpustate;
}
```

3.4 Test

The test for part A is running init program which forks and executes six times; thrice for long running program and thrice for Collatz sequence.

Collatz sequence is set to print results of first 100 positive numbers as it is stated in the assignment PDF.

Below is the each frame of the test run of this part:

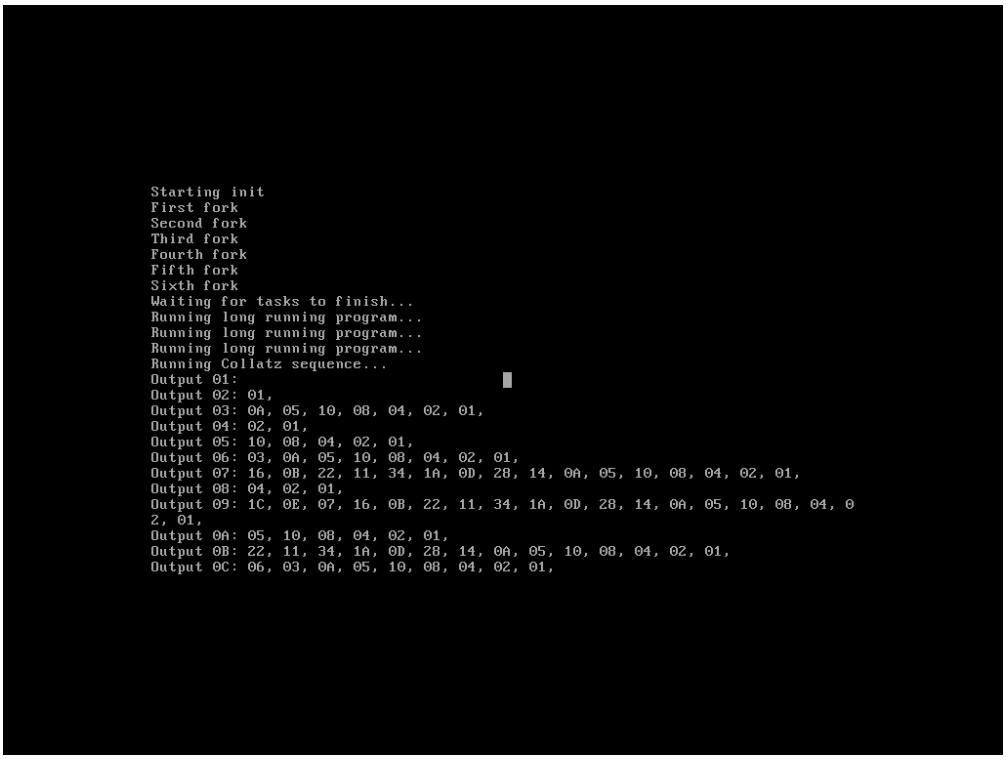


Figure 1: frame 1. Init process starting both collatz and long running programs thrice

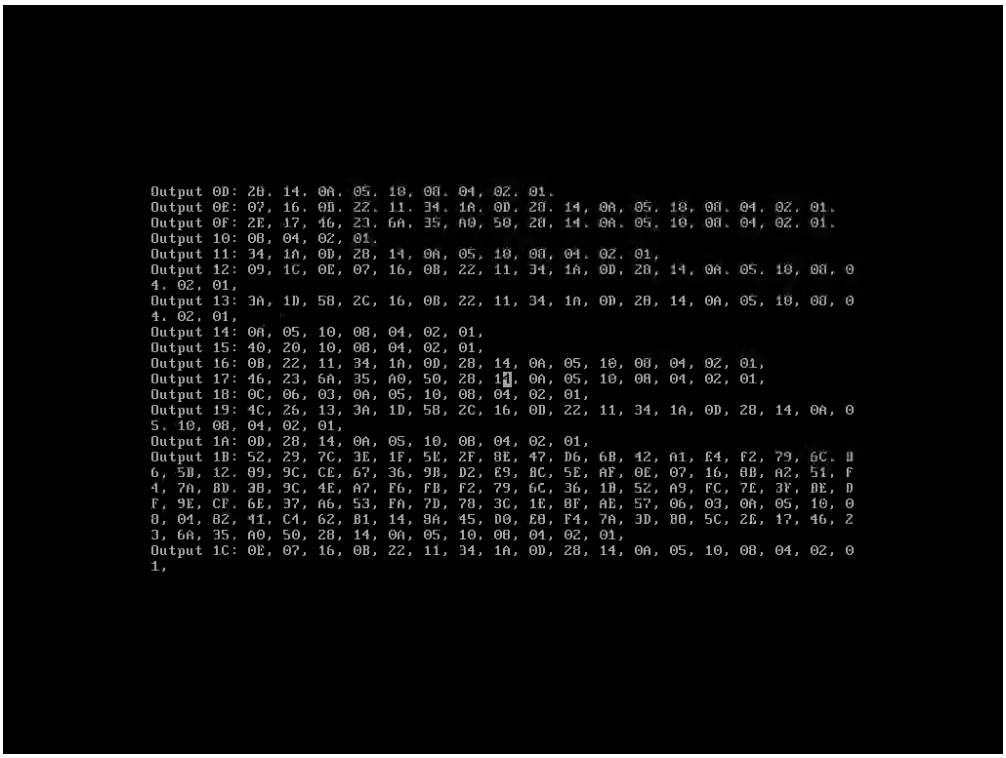


Figure 2: frame 2

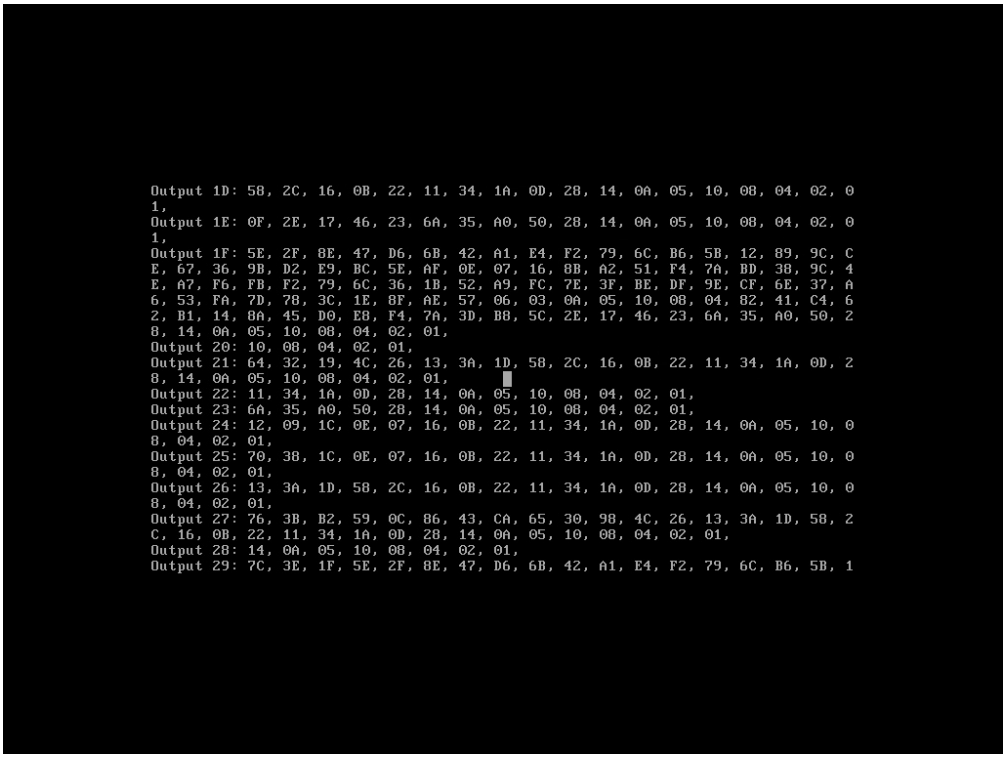


Figure 3: frame 3

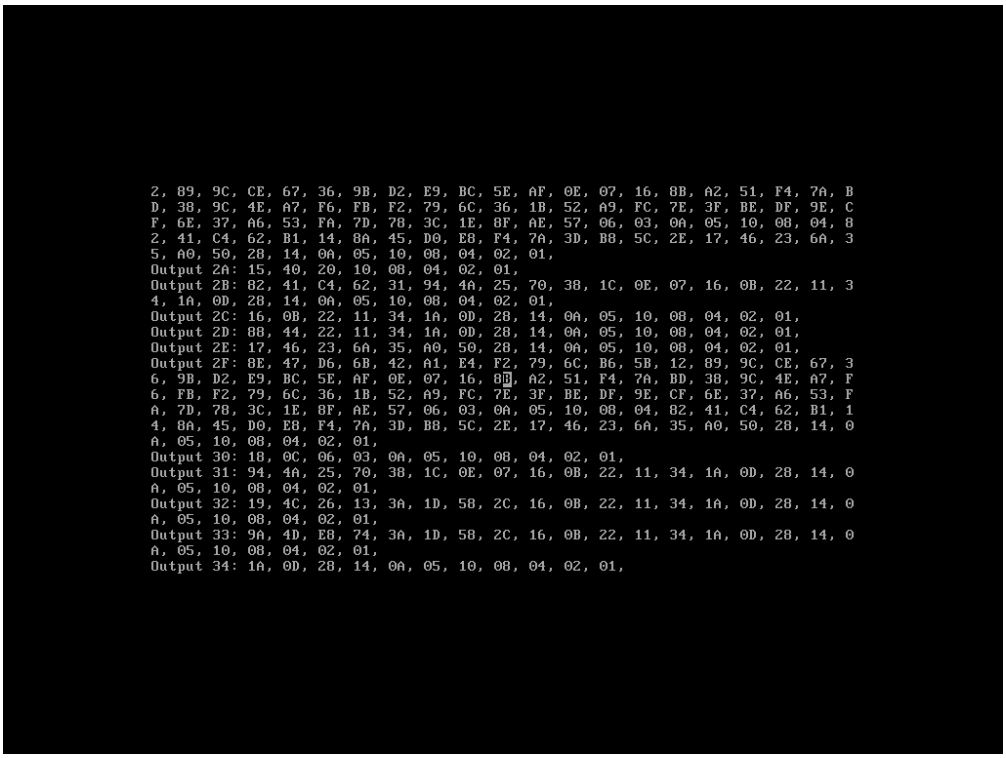


Figure 4: frame 4

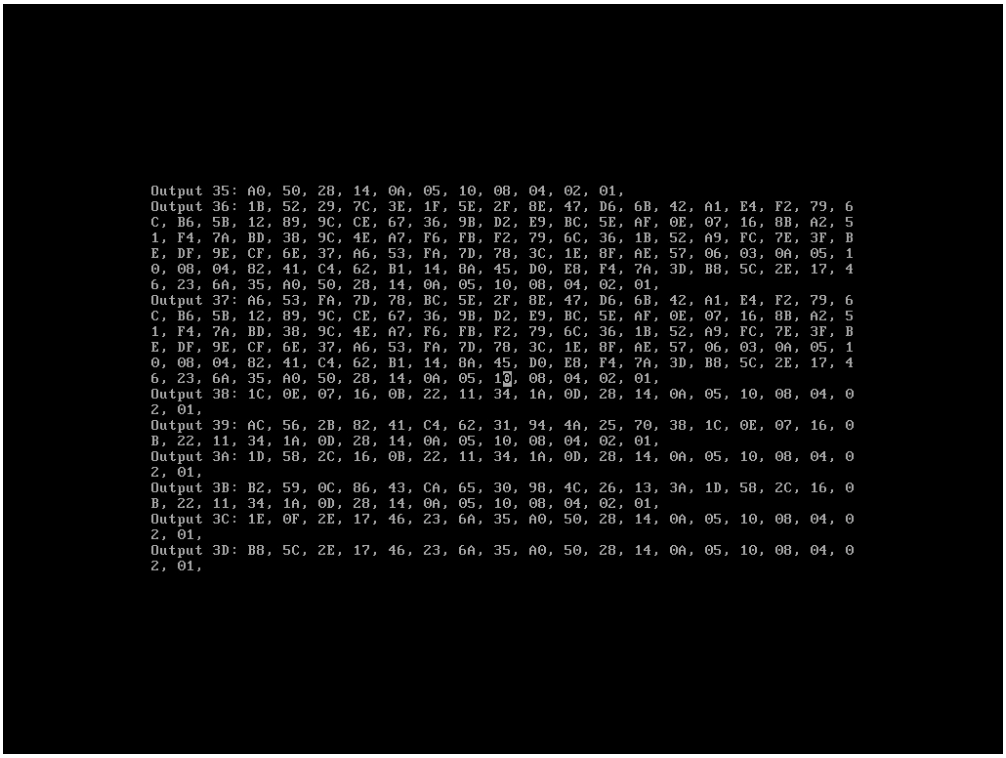


Figure 5: frame 5

Figure 6: frame 6

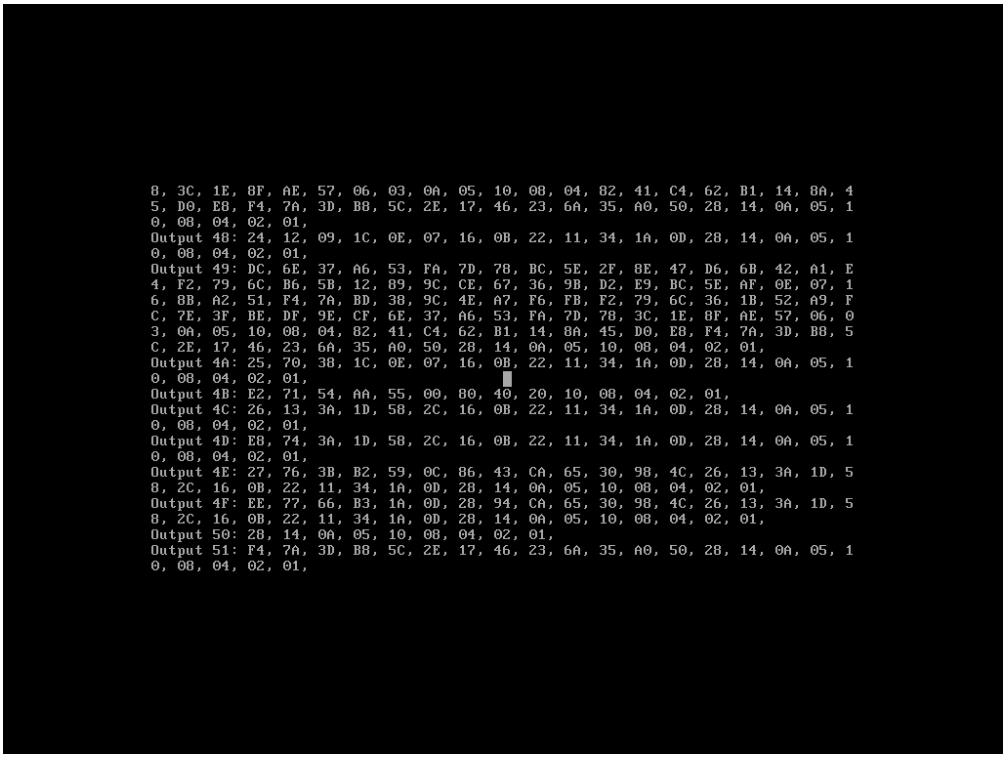


Figure 7: frame 7

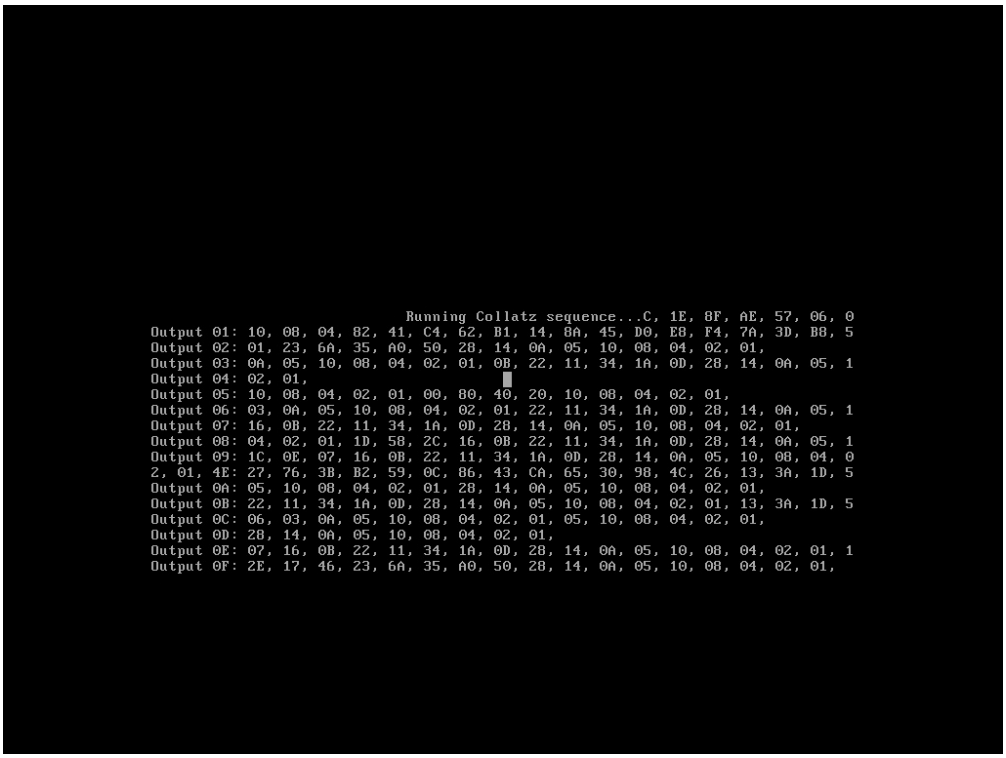


Figure 8: frame 8. Second Collatz program started before first one finished

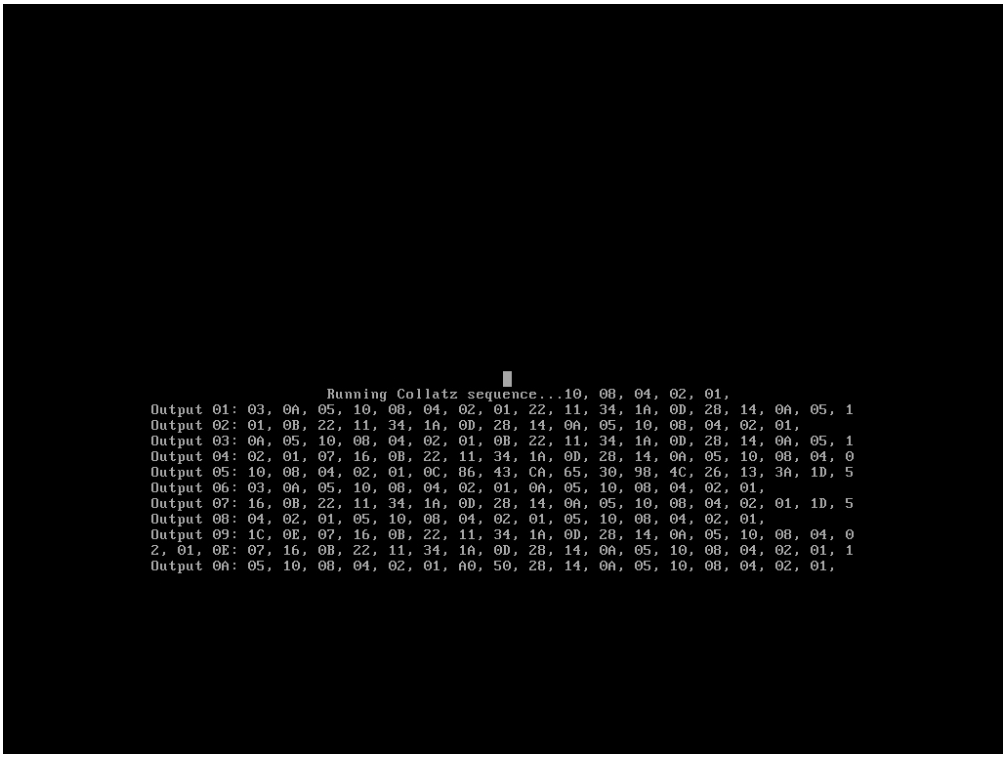


Figure 9: frame 9. Third Collatz program started before first one finished

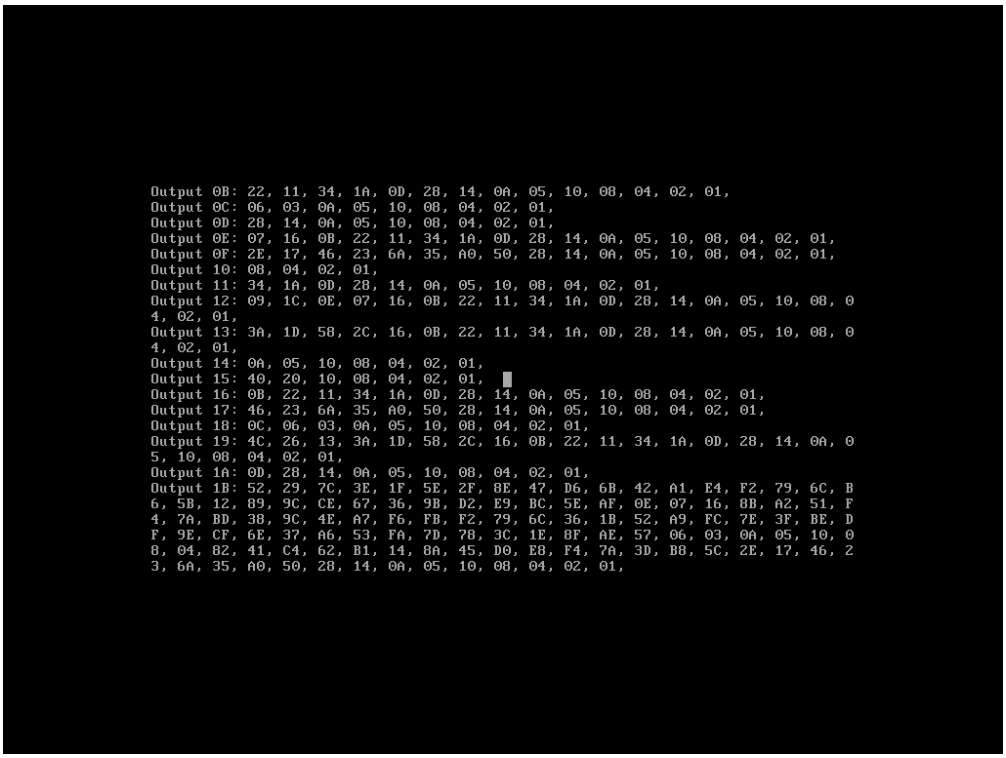


Figure 10: frame 10

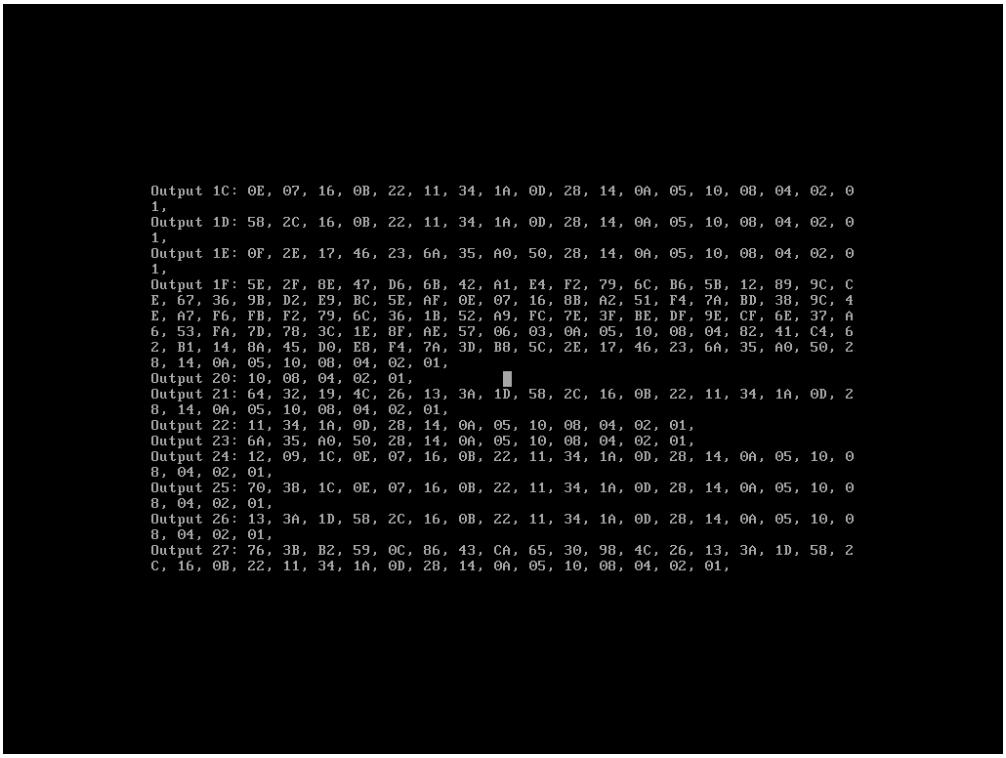


Figure 11: frame 11

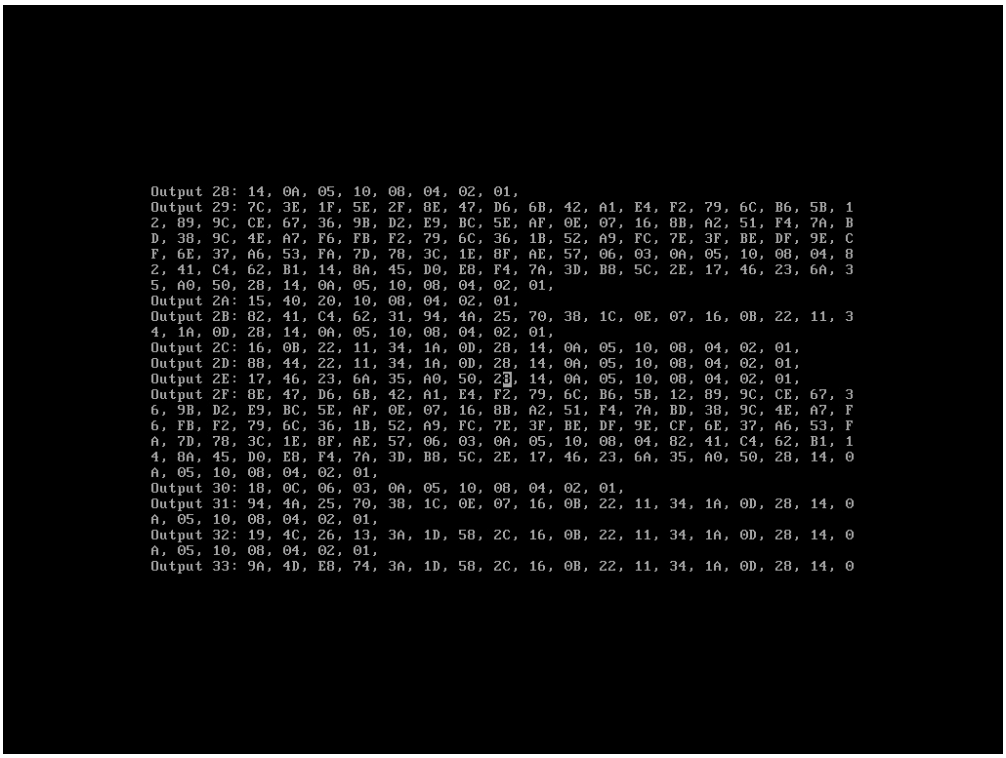


Figure 12: frame 12

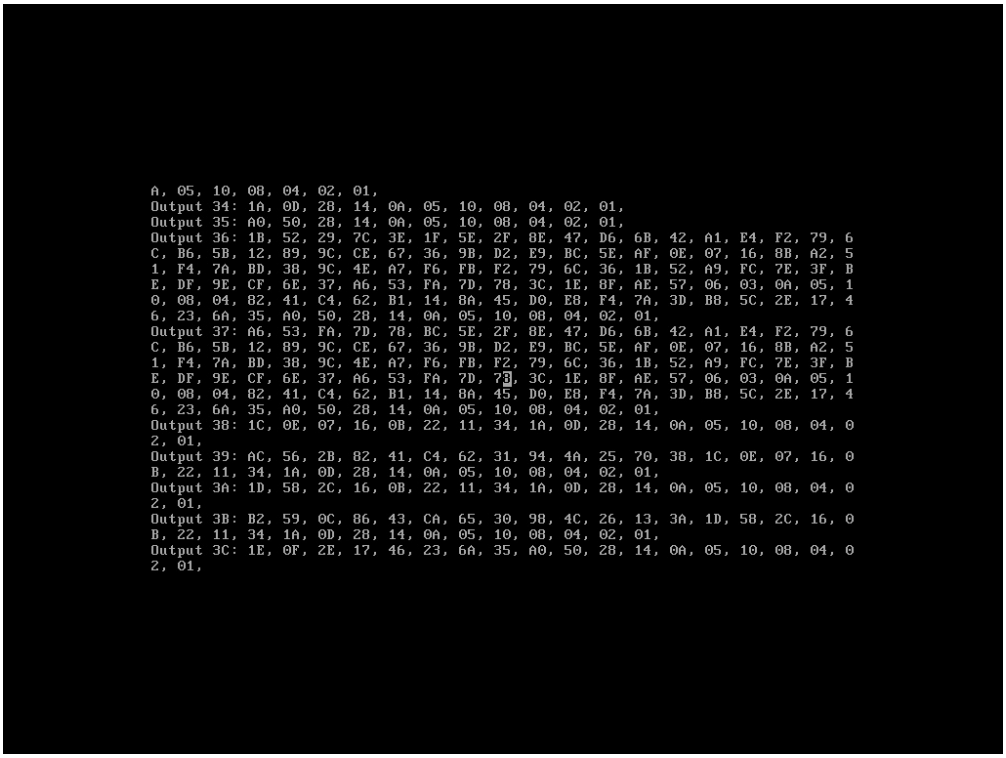


Figure 13: frame 13

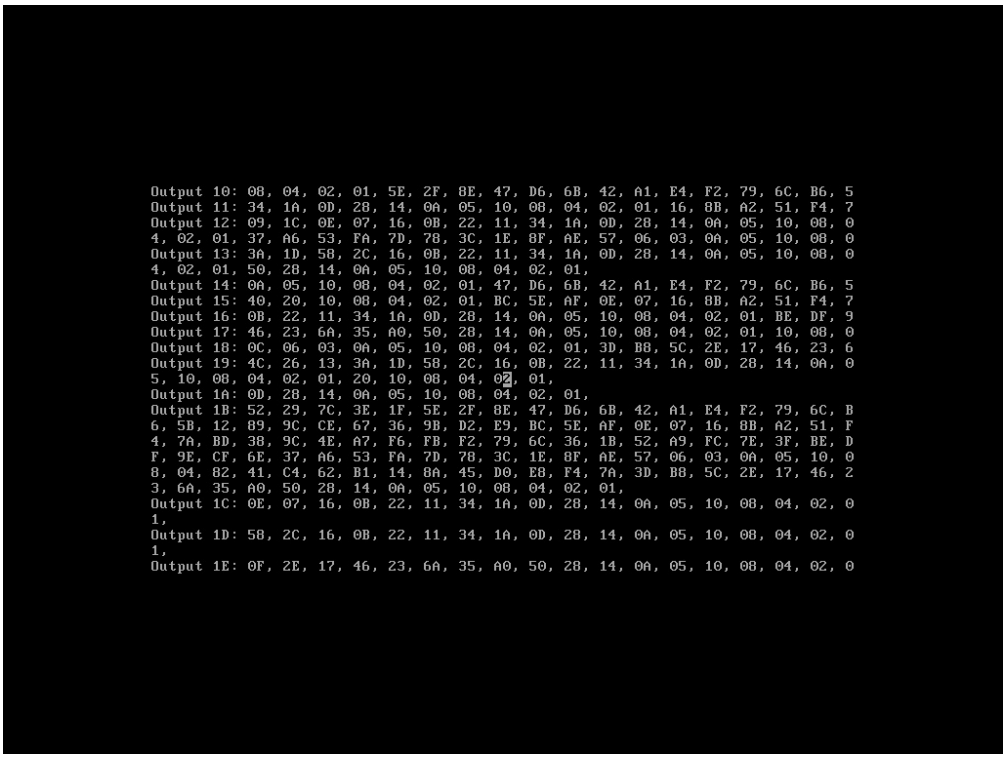


Figure 14: frame 14

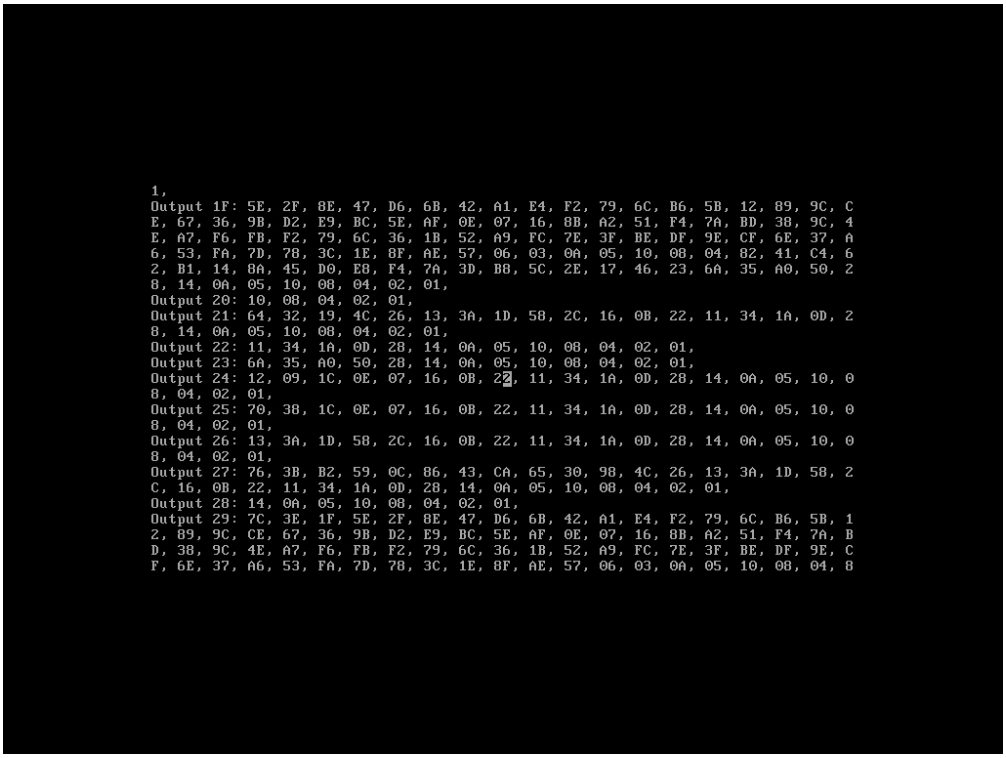


Figure 15: frame 15

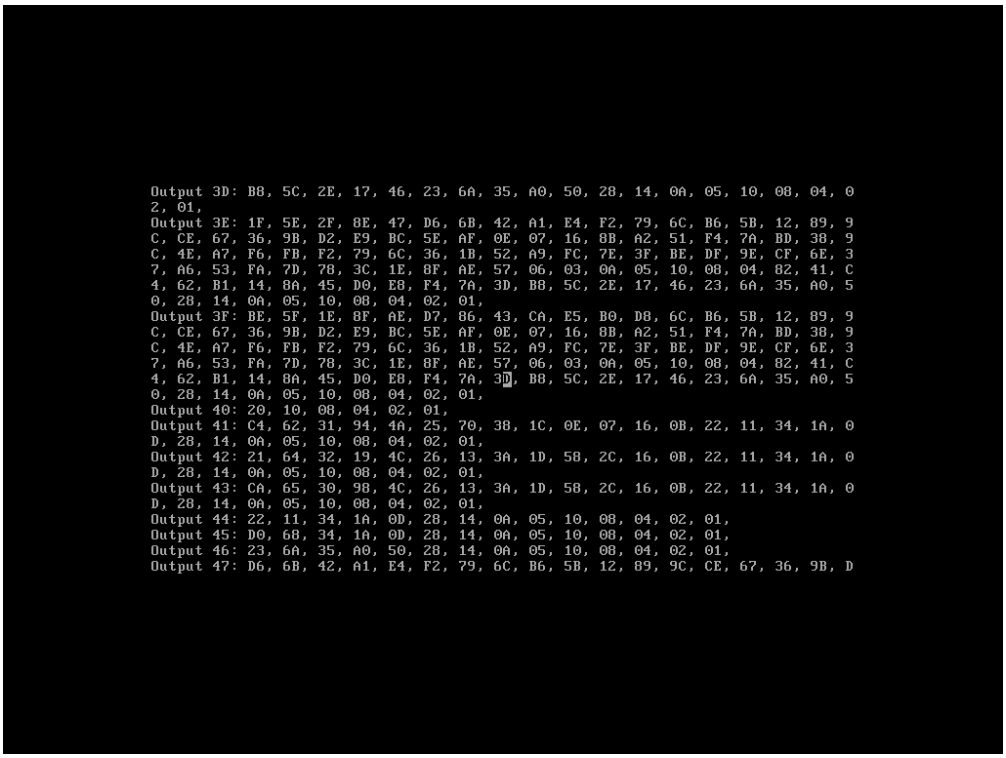


Figure 16: frame 16

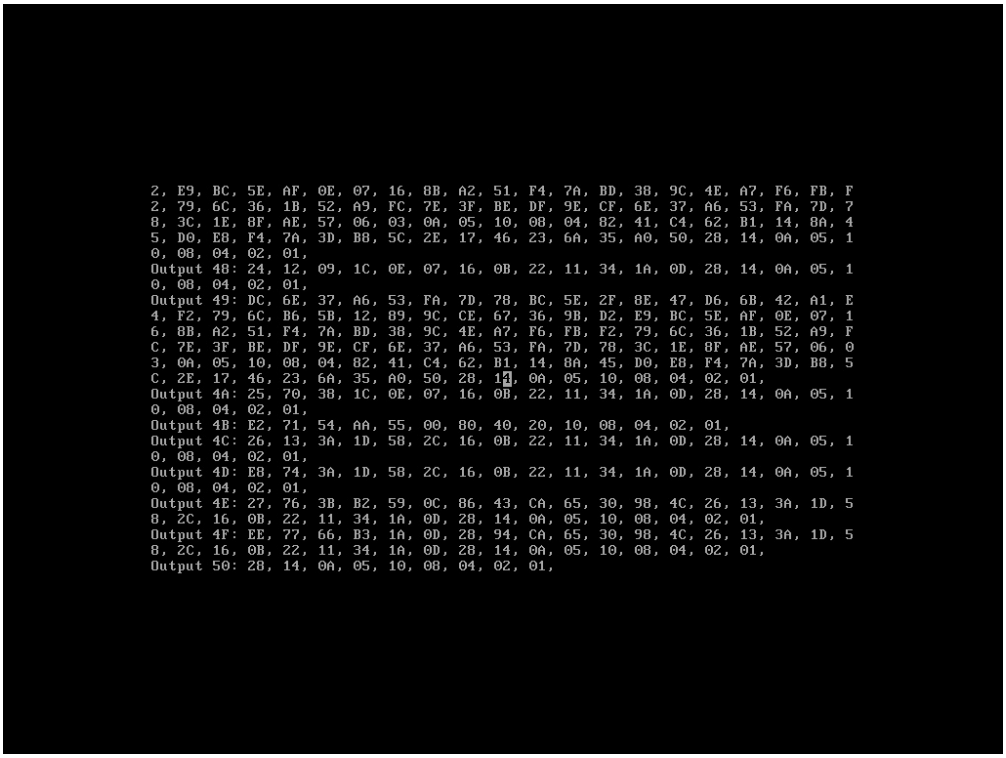


Figure 17: frame 17

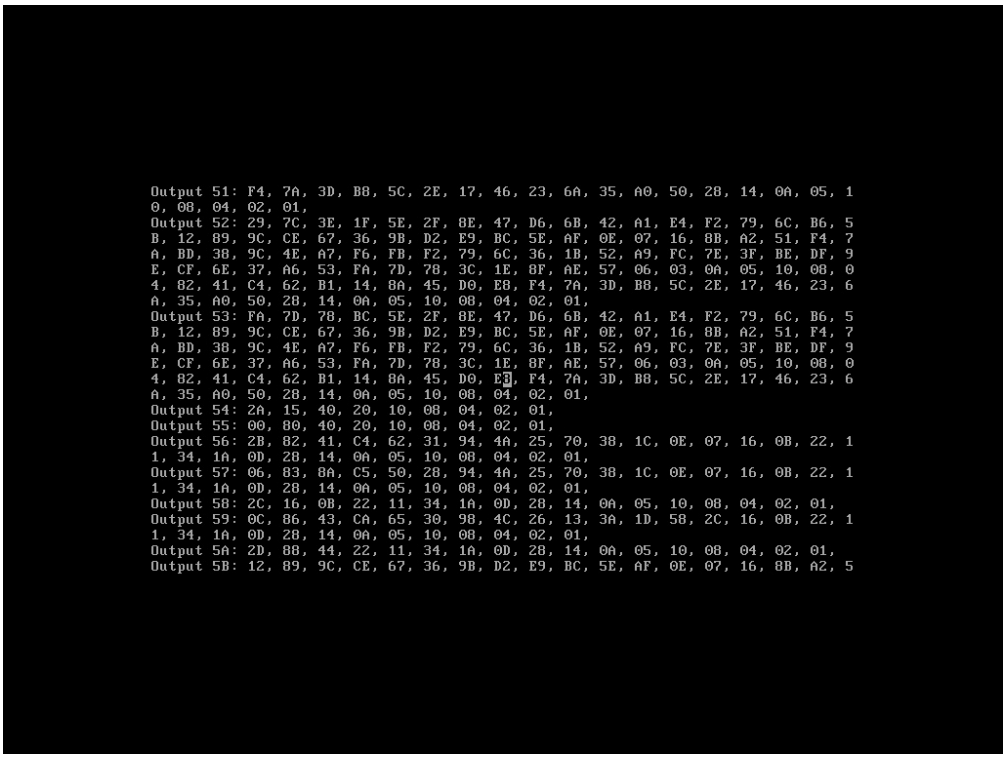


Figure 18: frame 18

Figure 19: frame 19

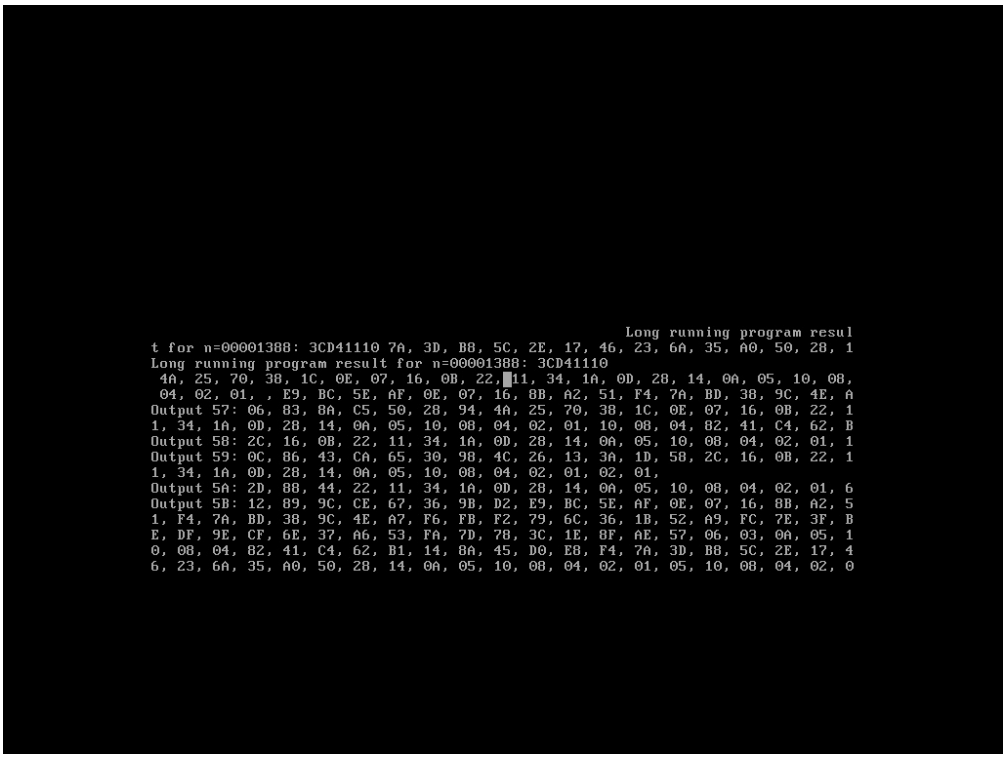


Figure 20: frame 20. Long running programs finished computing and printed their results.

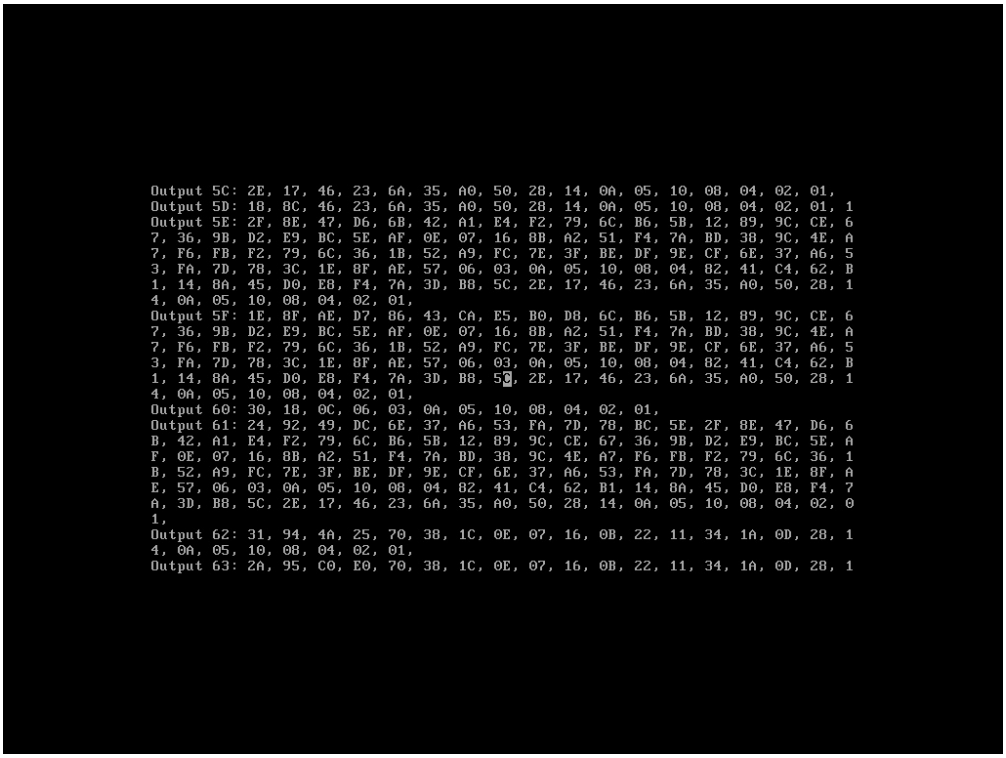


Figure 21: frame 21

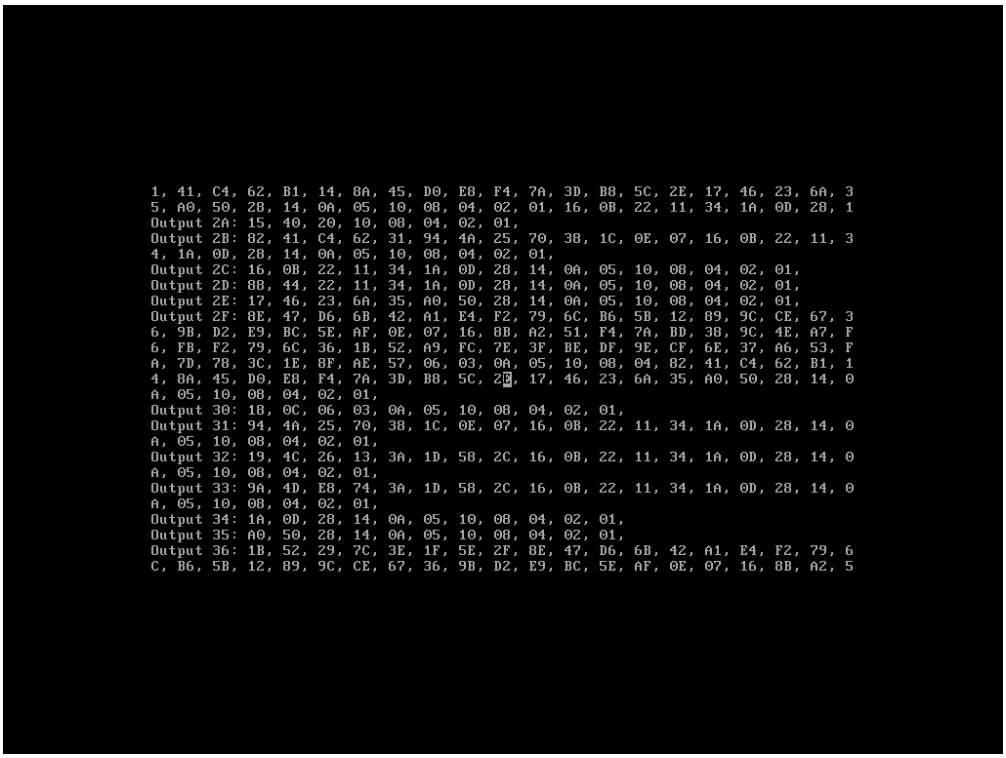


Figure 22: frame 22

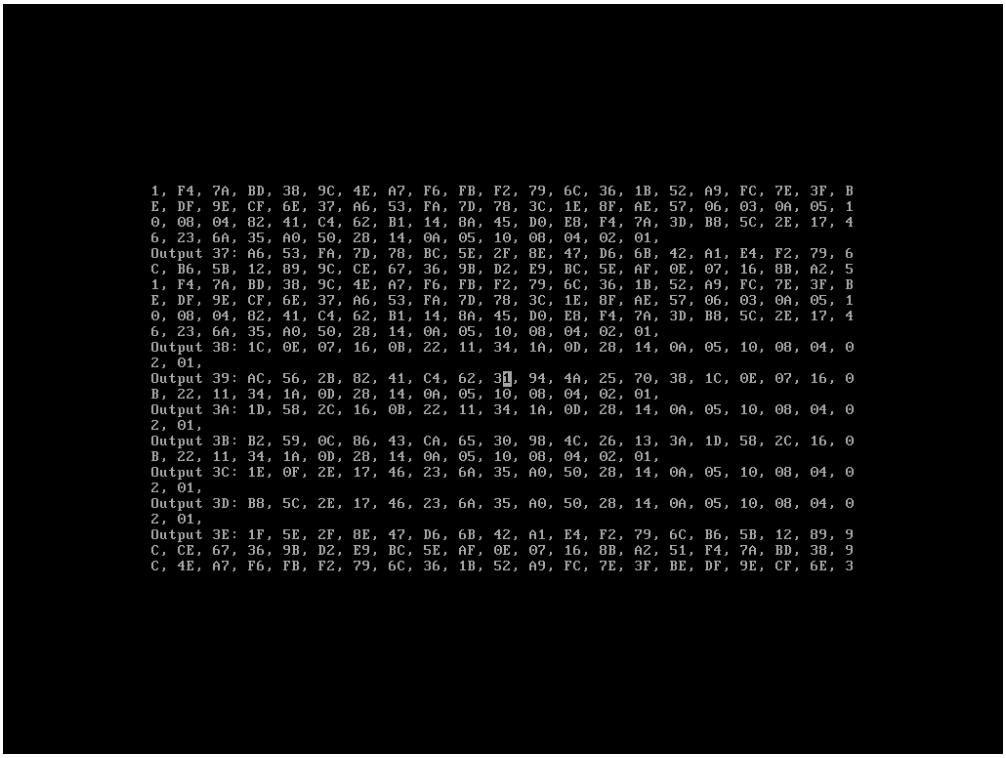


Figure 23: frame 23

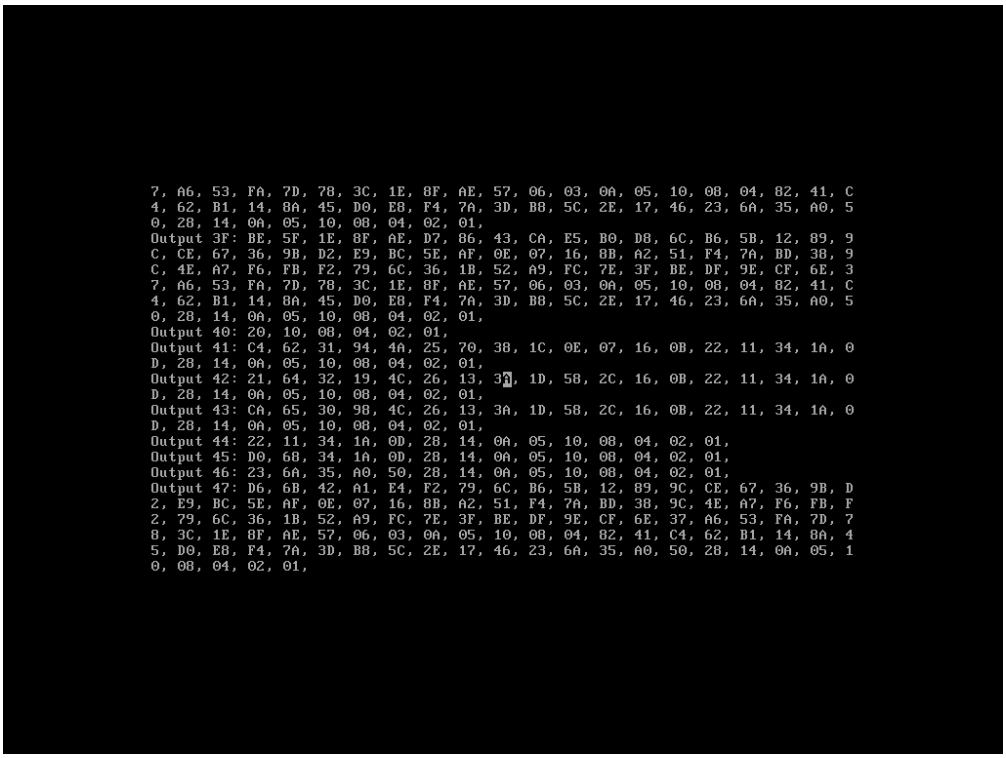


Figure 24: frame 24

Figure 25: frame 25

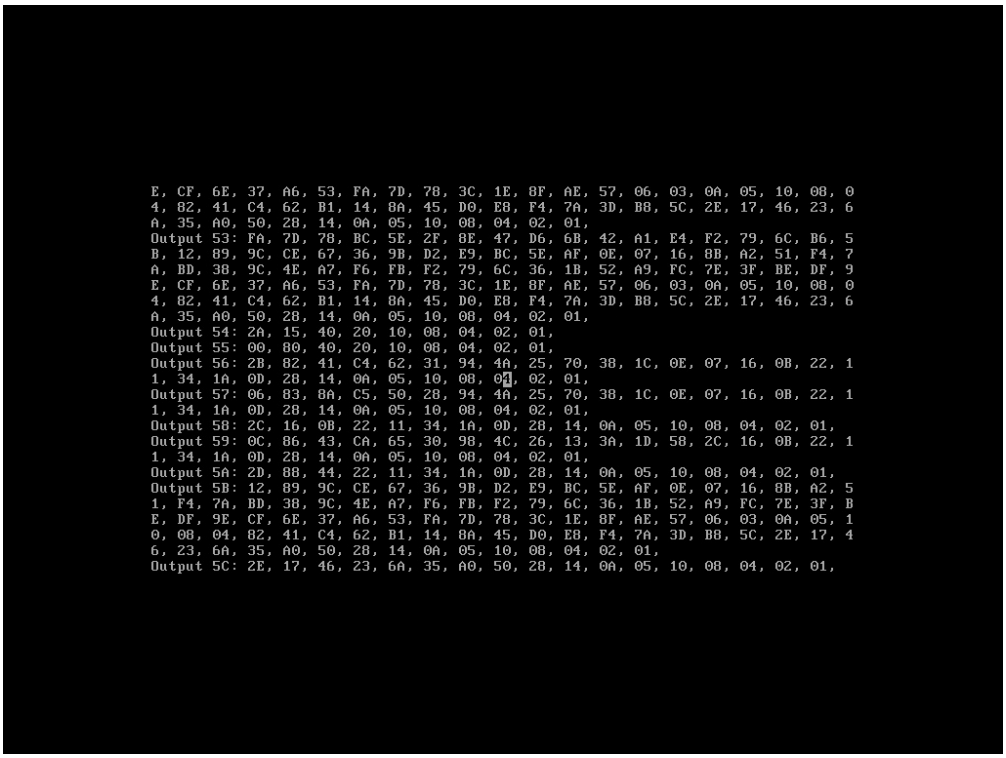


Figure 26: frame 26

Figure 27: frame 27

Figure 28: frame 28. All processes are finished

4 Part B

4.1 Task Description

In this part, it was required to implement Round Robin and Priority Based Scheduling along with loading multiple programs into memory.

4.2 Implementation Details

4.2.1 Round Robin

The Round Robin scheduling is actually implemented throughout each part of the assignment, the difference for this part is that priority was introduced to the scheduler from part A.

4.2.2 Process Priority

The process (task as in the source code) class is modified to have a priority field and task manager is also modified to have `checkPriority`.

These modifications enable the task manager to or not to favor processes with higher priority. Whenever the flag `checkPriority` is set to `true` the scheduler looks for the process with the highest priority and runs as long as it's in `READY` state.

The modification done to function finding task to run with priority can be seen below:

```
// multitasking.cpp
void myos::TaskManager::FindNextTask()
{
    if (checkPriority)
    {
```

```
        if (taskQueue.isEmpty())
        {
            return;
        }
        currentTask = taskQueue.peek();
        return;
    }
    // ...
    // Same as in part A scheduler, no priority case
    // ...
}
```

Where `taskQueue` is simply a classic priority queue with the design below:

```
#ifndef PRIORITY_QUEUE_H
#define PRIORITY_QUEUE_H

#include <common/types.h>

namespace myos
{
    // Node structure for the priority queue
    struct Node
    {
        int data;
        int priority;
        Node *next;
    };

    // Priority Queue class
    class PriorityQueue
    {
    public:
        // Constructor
        PriorityQueue();

        // Destructor
        ~PriorityQueue();

        // Check if the priority queue is empty
        bool isEmpty();

        // Insert an element with a given priority
        void enqueue(int data, int priority);

        // Remove and return the element with the highest priority
        int dequeue();

        // Get the element with the highest priority without removing it
        int peek();

    private:
        Node *head; // Pointer to the head of the linked list
    };
}
```

```
#endif // PRIORITY_QUEUE_H
```

The `taskQueue` actually only stores the indices of the tasks corresponding to their positions in the original `tasks` array, instead of replacing it altogether.

This is design choice is preferred since it does not require deleting and replacing existing source code, and let's task manager to switch modes on the fly by supporting both. Having `taskQueue` and `tasks` array separated makes it easier for handling priority and non-priority strategies.

4.2.3 Multiple Strategies

Each strategy sets the relevant fields of the task manager as they desire to enable their processes to run in that strategy.

This approach of using same task manager instead of instantiating one task manager for each strategy is better for OS's that work on one CPU, since no real parallelism is possible and only one strategy is actually running at the same time.

4.3 Lifecycle

This part involves implementing four different strategies for loading and executing multiple programs within the operating system.

4.3.1 First Strategy: Random Program Selection

In this strategy, the OS randomly selects one out of the four available programs (Collatz sequence, long-running program, binary search, and linear search) and loads it into memory ten times. Each program instance is then started, and the system enters an infinite loop until all processes terminate.

4.3.2 Second Strategy: Random Program Pairs

In the second strategy, the OS randomly selects two out of the four available programs and loads each selected program into memory three times. The instances of these programs are then started, and the system enters an infinite loop until all processes terminate.

4.3.3 Third Strategy: Priority Initialization

For the third strategy, the initialization process (`init`) initializes the Process Table and the ready queue. The Collatz sequence program is placed in the ready queue with the lowest priority. After the fifth interrupt, the remaining programs are loaded into memory and added to the ready queue, as their priorities are the same.

4.3.4 Fourth Strategy: Dynamic Priority Adjustment

Is not implemented

4.4 Test

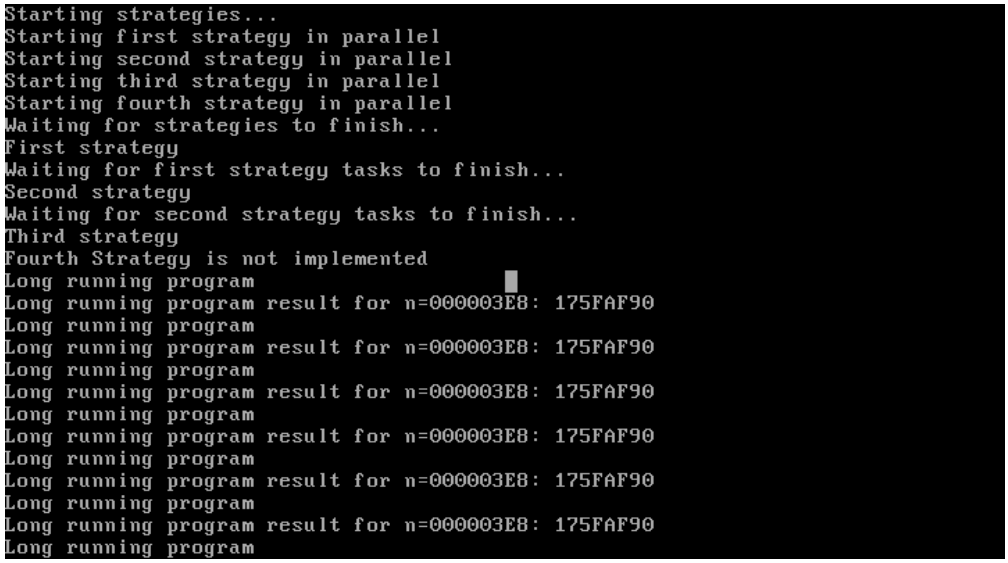


Figure 29: frame 1. Starting each strategy and each starting their tasks. First strategy prints some results

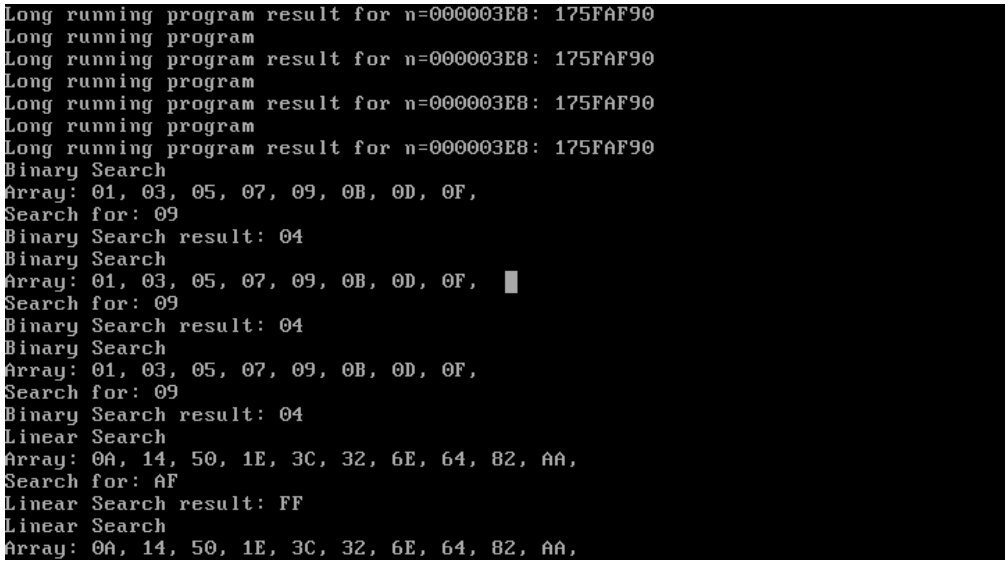


Figure 30: frame 2. First strategy tasks are finished. Second strategy randomly selected binary search and linear search

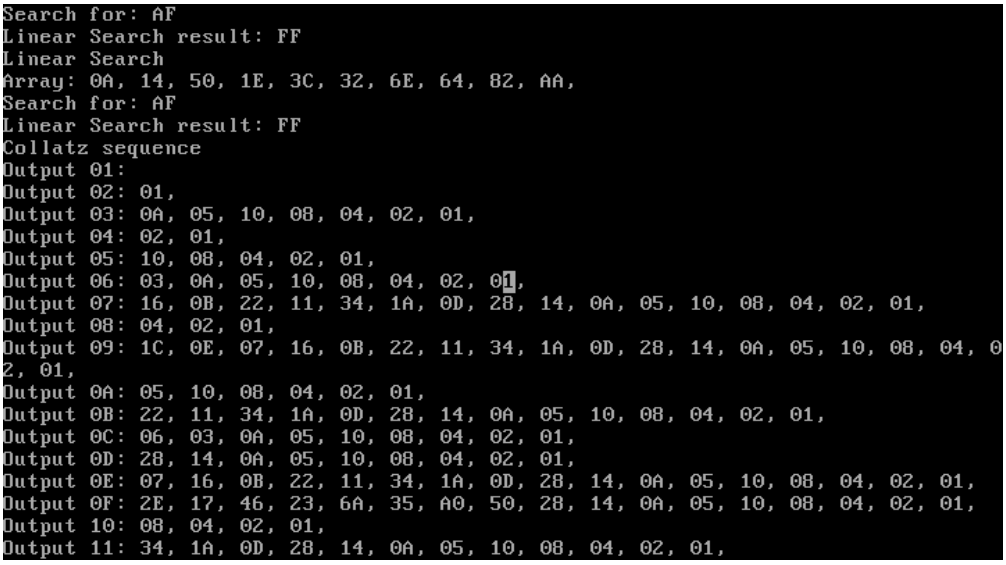


Figure 31: frame 3. Third strategy initialized Collatz

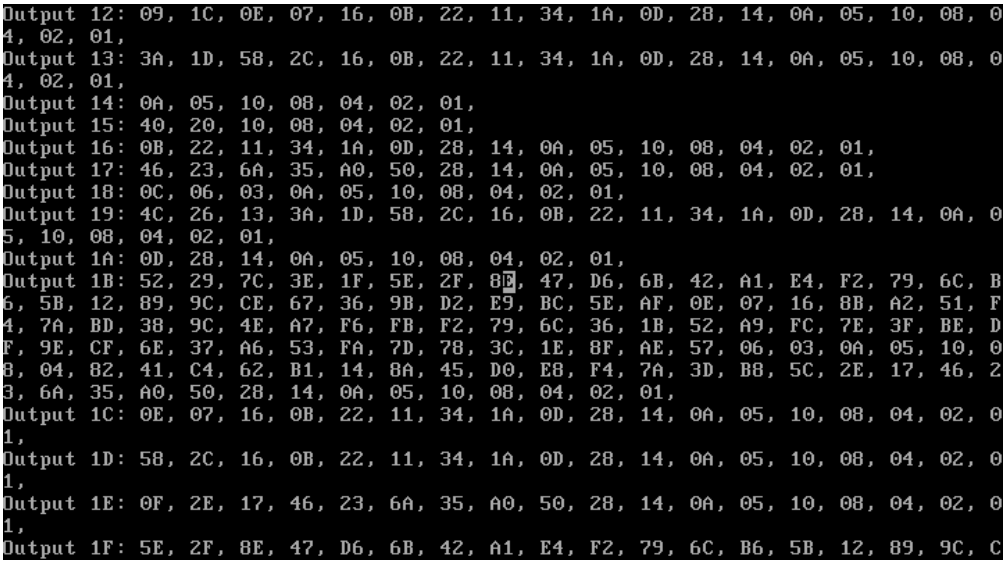


Figure 32: frame 4

```
E, 67, 36, 9B, D2, E9, BC, 5E, AF, 0E, 07, 16, 8B, A2, 51, F4, 7A, BD, 38, 9C, 4
E, A7, F6, FB, F2, 79, 6C, 36, 1B, 52, A9, FC, 7E, 3F, BE, DF, 9E, CF, 6E, 37, A
6, 53, FA, 7D, 78, 3C, 1E, 8F, AE, 57, 06, 03, 0A, 05, 10, 08, 04, 82, 41, C4, 6
2, B1, 14, 8A, 45, D0, E8, F4, 7A, 3D, B8, 5C, 2E, 17, 46, 23, 6A, 35, A0, 50, 2
8, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 20: 10, 08, 04, 02, 01,
Output 21: 64, 32, 19, 4C, 26, 13, 3A, 1D, 58, 2C, 16, 0B, 22, 11, 34, 1A, 0D, 2
8, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 22: 11, 34, 1A, 0D, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 23: 6A, 35, A0, 50, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 24: 12, 09, 1C, 0E, 07, 16, 0B, 22, 11, 34, 1A, 0D, 28, 14, 0A, 05, 10, 0
8, 04, 02, 01,
Output 25: 70, 38, 1C, 0E, 07, 16, 0B, 22, 11, 34, 1A, 0D, 28, 14, 0A, 05, 10, 0
8, 04, 02, 01,
Output 26: 13, 3A, 1D, 58, 2C, 16, 0B, 22, 11, 34, 1A, 0D, 28, 14, 0A, 05, 10, 0
8, 04, 02, 01,
Output 27: 76, 3B, B2, 59, 0C, 86, 43, CA, 65, 30, 98, 4C, 26, 13, 3A, 1D, 58, 2
C, 16, 0B, 22, 11, 34, 1A, 0D, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 28: 14, 0A, 05, 10, 08, 04, 02, 01,
Output 29: 7C, 3E, 1F, 5E, 2F, 8E, 47, D6, 6B, 42, A1, E4, F2, 79, 6C, B6, 5B, 1
2, 89, 9C, CE, 67, 36, 9B, D2, E9, BC, 5E, AF, 0E, 07, 16, 8B, A2, 51, F4, 7A, B
D, 38, 9C, 4E, A7, F6, FB, F2, 79, 6C, 36, 1B, 52, A9, FC, 7E, 3F, BE, DF, 9E, C
F, 6E, 37, A6, 53, FA, 7D, 78, 3C, 1E, 8F, AE, 57, 06, 03, 0A, 05, 10, 08, 04, 8
2, 41, C4, 62, B1, 14, 8A, 45, D0, E8, F4, 7A, 3D, B8, 5C, 2E, 17, 46, 23, 6A, 3
5, A0, 50, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
```

Figure 33: frame 5

```
Output 2A: 15, 40, 20, 10, 08, 04, 02, 01,
Output 2B: 82, 41, C4, 62, 31, 94, 4A, 25, 70, 38, 1C, 0E, 07, 16, 0B, 22, 11, 3
4, 1A, 0D, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 2C: 16, 0B, 22, 11, 34, 1A, 0D, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 2D: 88, 44, 22, 11, 34, 1A, 0D, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 2E: 17, 46, 23, 6A, 35, A0, 50, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 2F: 8E, 47, D6, 6B, 42, A1, E4, F2, 79, 6C, B6, 5B, 12, 89, 9C, CE, 67, 3
6, 9B, D2, E9, BC, 5E, AF, 0E, 07, 16, 8B, A2, 51, F4, 7A, BD, 38, 9C, 4E, A7, F
6, FB, F2, 79, 6C, 36, 1B, 52, A9, FC, 7E, 3F, BE, DF, 9E, CF, 6E, 37, A6, 53, F
A, 7D, 78, 3C, 1E, 8F, AE, 57, 06, 03, 0A, 05, 10, 08, 04, 82, 41, C4, 62, B1, 1
4, 8A, 45, D0, E8, F4, 7A, 3D, B8, 5C, 2E, 17, 46, 23, 6A, 35, A0, 50, 28, 14, 0
A, 05, 10, 08, 04, 02, 01,
Output 30: 18, 0C, 06, 03, 0A, 05, 10, 08, 04, 02, 01,
Output 31: 94, 4A, 25, 70, 38, 1C, 0E, 07, 16, 0B, 22, 11, 34, 1A, 0D, 28, 14, 0
A, 05, 10, 08, 04, 02, 01,
Output 32: 19, 4C, 26, 13, 3A, 1D, 58, 2C, 16, 0B, 22, 11, 34, 1A, 0D, 28, 14, 0
A, 05, 10, 08, 04, 02, 01,
Output 33: 9A, 4D, E8, 74, 3A, 1D, 58, 2C, 16, 0B, 22, 11, 34, 1A, 0D, 28, 14, 0
A, 05, 10, 08, 04, 02, 01,
Output 34: 1A, 0D, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 35: A0, 50, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 36: 1B, 52, 29, 7C, 3E, 1F, 5E, 2F, 8E, 47, D6, 6B, 42, A1, E4, F2, 79, 6
C, B6, 5B, 12, 89, 9C, CE, 67, 36, 9B, D2, E9, BC, 5E, AF, 0E, 07, 16, 8B, A2, 5
1, F4, 7A, BD, 38, 9C, 4E, A7, F6, FB, F2, 79, 6C, 36, 1B, 52, A9, FC, 7E, 3F, B
E, DF, 9E, CF, 6E, 37, A6, 53, FA, 7D, 78, 3C, 1E, 8F, AE, 57, 06, 03, 0A, 05, 1
```

Figure 34: frame 6

```
0, 08, 04, 82, 41, C4, 62, B1, 14, 8A, 45, D0, E8, F4, 7A, 3D, B8, 5C, 2E, 17, 4
6, 23, 6A, 35, A0, 50, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 37: A6, 53, FA, 7D, 78, BC, 5E, 2F, 8E, 47, D6, 6B, 42, A1, E4, F2, 79, 6
C, B6, 5B, 12, 89, 9C, CE, 67, 36, 9B, D2, E9, BC, 5E, AF, 0E, 07, 16, 8B, A2, 5
1, F4, 7A, BD, 38, 9C, 4E, A7, F6, FB, F2, 79, 6C, 36, 1B, 52, A9, FC, 7E, 3F, B
E, DF, 9E, CF, 6E, 37, A6, 53, FA, 7D, 78, 3C, 1E, 8F, AE, 57, 06, 03, 0A, 05, 1
0, 08, 04, 82, 41, C4, 62, B1, 14, 8A, 45, D0, E8, F4, 7A, 3D, B8, 5C, 2E, 17, 4
6, 23, 6A, 35, A0, 50, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 38: 1C, 0E, 07, 16, 0B, 22, 11, 34, 1A, 0D, 28, 14, 0A, 05, 10, 08, 04, 0
2, 01,
Output 39: AC, 56, 2B, 82, 41, C4, 62, 31, 94, 4A, 25, 70, 38, 1C, 0E, 07, 16, 0
B, 22, 11, 34, 1A, 0D, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 3A: 1D, 58, 2C, 16, 0B, 22, 11, 34, 1A, 0D, 28, 14, 0A, 05, 10, 08, 04, 0
2, 01,
Output 3B: B2, 59, 0C, 86, 43, CA, 65, 30, 98, 4C, 26, 13, 3A, 1D, 58, 2C, 16, 0
B, 22, 11, 34, 1A, 0D, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 3C: 1E, 0F, 2E, 17, 46, 23, 6A, 35, A0, 50, 28, 14, 0A, 05, 10, 08, 04, 0
2, 01,
Output 3D: B8, 5C, 2E, 17, 46, 23, 6A, 35, A0, 50, 28, 14, 0A, 05, 10, 08, 04, 0
2, 01,
Output 3E: 1F, 5E, 2F, 8E, 47, D6, 6B, 42, A1, E4, F2, 79, 6C, B6, 5B, 12, 89, 9
C, CE, 67, 36, 9B, D2, E9, BC, 5E, AF, 0E, 07, 16, 8B, A2, 51, F4, 7A, BD, 38, 9
C, 4E, A7, F6, FB, F2, 79, 6C, 36, 1B, 52, A9, FC, 7E, 3F, BE, DF, 9E, CF, 6E, 3
7, A6, 53, FA, 7D, 78, 3C, 1E, 8F, AE, 57, 06, 03, 0A, 05, 10, 08, 04, 82, 41, C
4, 62, B1, 14, 8A, 45, D0, E8, F4, 7A, 3D, B8, 5C, 2E, 17, 46, 23, 6A, 35, A0, 5
```

Figure 35: frame 7

```
0, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 3F: BE, 5F, 1E, 8F, AE, D7, 86, 43, CA, E5, B0, D8, 6C, B6, 5B, 12, 89, 9
C, CE, 67, 36, 9B, D2, E9, BC, 5E, AF, 0E, 07, 16, 8B, A2, 51, F4, 7A, BD, 38, 9
C, 4E, A7, F6, FB, F2, 79, 6C, 36, 1B, 52, A9, FC, 7E, 3F, BE, DF, 9E, CF, 6E, 3
7, A6, 53, FA, 7D, 78, 3C, 1E, 8F, AE, 57, 06, 03, 0A, 05, 10, 08, 04, 82, 41, C
4, 62, B1, 14, 8A, 45, D0, E8, F4, 7A, 3D, B8, 5C, 2E, 17, 46, 23, 6A, 35, A0, 5
0, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 40: 20, 10, 08, 04, 02, 01,
Output 41: C4, 62, 31, 94, 4A, 25, 70, 38, 1C, 0E, 07, 16, 0B, 22, 11, 34, 1A, 0
D, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 42: 21, 64, 32, 19, 4C, 26, 13, 3A, 1D, 58, 2C, 16, 0B, 22, 11, 34, 1A, 0
D, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 43: CA, 65, 30, 98, 4C, 26, 13, 3A, 1D, 58, 2C, 16, 0B, 22, 11, 34, 1A, 0
D, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 44: 22, 11, 34, 1A, 0D, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 45: D0, 68, 34, 1A, 0D, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 46: 23, 6A, 35, A0, 50, 28, 14, 0A, 05, 10, 08, 04, 02, 01,
Output 47: D6, 6B, 42, A1, E4, F2, 79, 6C, B6, 5B, 12, 89, 9C, CE, 67, 36, 9B, D
2, E9, BC, 5E, AF, 0E, 07, 16, 8B, A2, 51, F4, 7A, BD, 38, 9C, 4E, A7, F6, FB, F
2, 79, 6C, 36, 1B, 52, A9, FC, 7E, 3F, BE, DF, 9E, CF, 6E, 37, A6, 53, FA, 7D, 7
8, 3C, 1E, 8F, AE, 57, 06, 03, 0A, 05, 10, 08, 04, 82, 41, C4, 62, B1, 14, 8A, 4
5, D0, E8, F4, 7A, 3D, B8, 5C, 2E, 17, 46, 23, 6A, 35, A0, 50, 28, 14, 0A, 05, 1
0, 08, 04, 02, 01,
Output 48: 24, 12, 09, 1C, 0E, 07, 16, 0B, 22, 11, 34, 1A, 0D, 28, 14, 0A, 05, 1
0, 08, 04, 02, 01,
```

Figure 36: frame 8

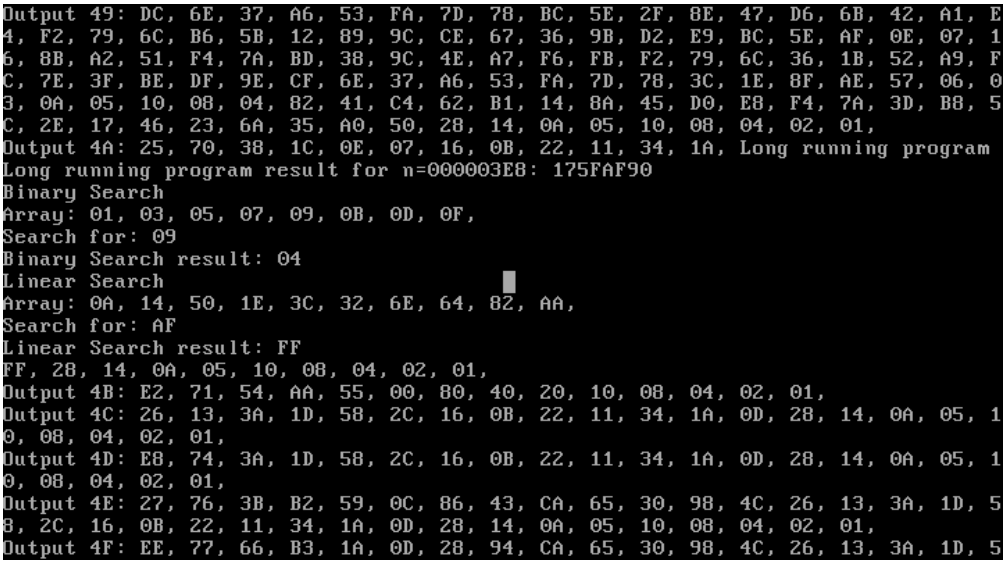


Figure 37: frame 9. Third strategy added remaining programs with higher priority so they cut Collatz program. Then Collatz continues

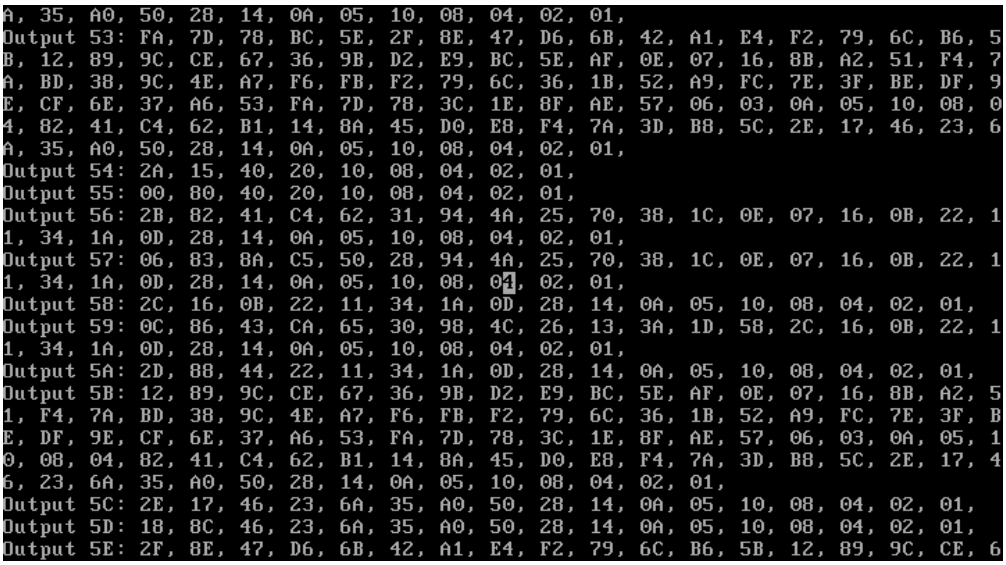


Figure 38: frame 10

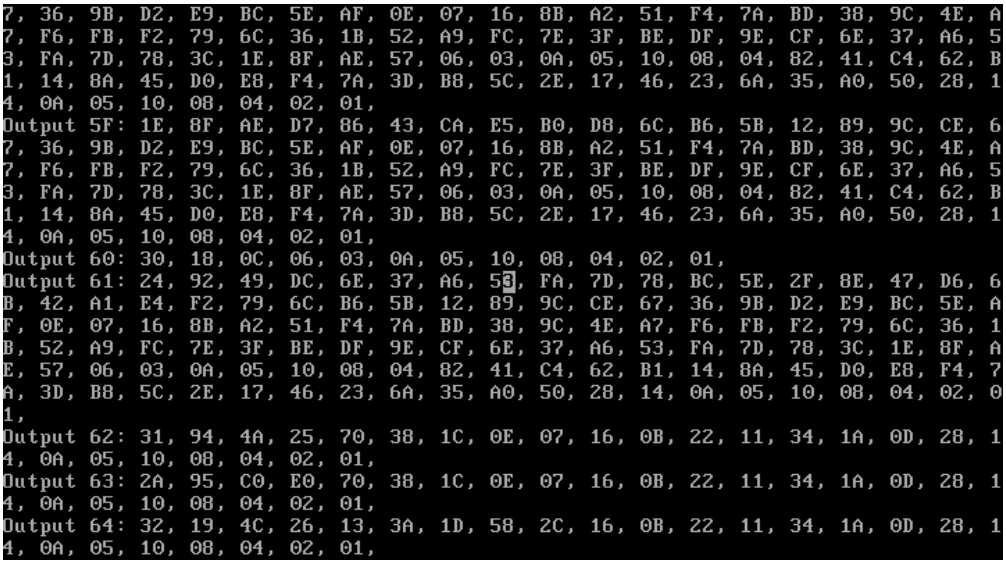


Figure 39: frame 11

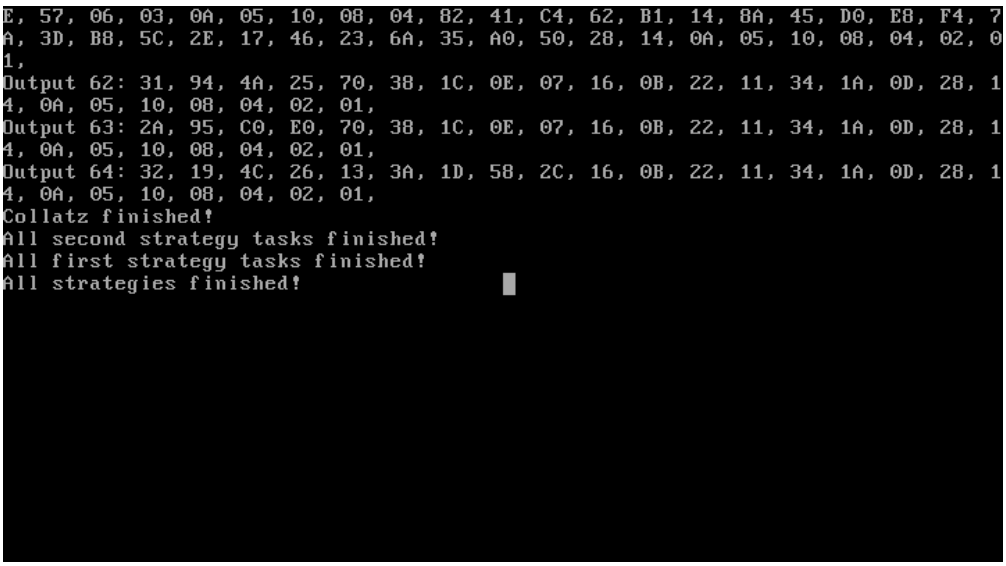


Figure 40: frame 12. Strategies terminated

5 Part C

5.1 Task Description

This part Interactive Input Priority Strategy was to be implemented.

5.2 Implementation Details

5.2.1 Priority Based On Interactivity

The Interactive Input Priority Strategy is basically implemented as:

- If a process is not waiting user input, then it's priority is the default priority.

- If a process is waiting user, then input it's priority is elevated to the higher than all other processes which prevents scheduler letting other processes run.

Since none of the provided programs require any mouse interaction, this input based priority handling is unique to keyboard inputs, which is needed when:

- Collatz sequence:
  - To enter the number of first positive integers to calculate Collatz sequence of
- Long running program:
  - To enter the number of iterations
- Binary search:
  - To enter 8 integers to binary search in (search array)
  - To enter integer to be searched
- Linear search:
  - To enter 10 integers to binary search in (search array)
  - To enter integer to be searched

To enable processes to get input from user the changes below are made to the `PrintfKeyboardEventHandler` class:

```
// kernel.cpp
class PrintfKeyboardEventHandler : public KeyboardEventHandler
{
public:
    void OnKeyDown(char c)
    {
        // ...
        // original part
        // ..
        if (buffer_position == 256)
            buffer_position = 0;
        if (c == '\n')
        {
            pressedEnter = true;
            enter_position = buffer_position;
        }
        buffer[buffer_position++] = c;
    }

    void GetBuffer(char *buf)
    {
        for (int i = 0; i < buffer_position; i++)
            buf[i] = buffer[i];
        buf[buffer_position] = 0;
        buffer_position = 0;
        pressedEnter = false;
    }

    bool PressedEnter()
    {
```

```
        return pressedEnter;
    }

private:
    char buffer[256];
    int buffer_position = 0;
    int enter_position = 0;
    bool pressedEnter = false;
};
```

Added functions below to change the running process's priority on the fly so that it is updated accordingly whenever input is waited:

```
// multitasking.cpp
void myos::TaskManager::SetPriority(int priority)
{
    tasks[currentTask]->SetPriority(priority);
    taskQueue.setPriority(currentTask, priority);
}

// mutlitasking.cpp
void myos::Task::SetPriority(int priority)
{
    this->priority = priority < 0 ? 0 : priority;
}
```

5.3 Lifecycle

The `init` process forks four times and executes each program and then waits for them to finish.

Each program's initial priority is the same therefore the scheduler selects the oldest/first one and runs it first.

Then the first process (Collatz in this case) demands an input from the user which triggers update on the process's priority. From that point the scheduler can only select a new process until the user enters 'newline' key.

This is repeated for each program until the last one finishes and system terminates.

6 Test

Since this part relies on **waiting** user input, making it time dependant, showcasing the test with a video is thought to be more proper:

See test video at: https://drive.google.com/file/d/1e5-mxQ9DKkGw-Kr5zCxDoaQbaVQaoW6y/view