Name: Enes Abdülhalik
Student No: 1901042803

CSE312 - Operating Systems
Homework 1 - Bonus - Report

The following requested requirements were implemented successfully:

- fork system call : uint16_t Kernel::fork();
  This system call takes no parameters. It creates an exact copy of the caller process that continues the execution from the same point the caller called fork. Stack, and instruction pointer are also copied. The new process becomes a child of the caller process.
  - Returns:
    - 0 in the child process.
    - The pid of the child process in the parent process.

- execve system call: void Kernel::execve(void* pathname, void* argv[])
  This system call changes the execution of a process to a new function. Everything, other than the stack pointer and the instruction pointer, remains the same. Pathname argument is a pointer to the main function of the new program. The function should be of the type void fun(void*). argv argument is a pointer to the parameters of the function.
  - Returns -1 if the execution was not changed.

- waitpid system call: void Kernel::waitpid(uint16_t pid, ProcessState state)
  This system call checks if a child process with the given pid has changed to the given state recently. The function scans for messages received by the caller (these messages can be received by children only), if a message coming from the given pid with the given state, it's removed from the message queue and the pid of the child process is returned.
  - Returns:
    - the pid of the child process whose state has changed.
    - -1 if no such change happened before the call.
  - Note: Only termination messages are supported at the moment.

- Context switching using the keyboard and mouse is also supported. For more information, read the multitasking and input devices segments.

- During the tests, the input for the functions will be taken from the user using the keyboard. Instructions about the format of the input will be displayed on the screen whenever input is required.

The following system calls were implemented to help with the operation of the above calls:

- exit system call : uint16_t Kernel::exit();
  This system call takes no parameters. It removes the process from the process table and sends a message to the parent telling it that this child has terminated. During the exit,
  - the children of the terminated process are adopted by the process init.
  - Both the stack and the process entry spaces are freed and returned back for further use.
  - A message is sent to the parent saying that this process has terminated.
  - The process is removed from the parent's children list.
  - The process's place in the process description table remains empty until a new process occupies its place and takes it's pid.

Implementation Info:

OOP Design:

The video series provided was very useful as start to do the homework, but the overall code structure did not look good to me since the idea behind the series was to write the OS in OOP style. Let’s not waste time on this, in short, I redesigned some of the already provided classes and changed some naming conventions to make it look more OOP like.

Memory Segments:

6 memory segments were used:

1. Null Segment: Used by the bootloader.
2. Unused Segment: I have no idea about it. Other sources never mentioned it, but the OS wouldn’t boot without it, so I guess it’s a must.
3. Code Segment: Where the instructions of the kernel are stored.
4. Data Segment: Global variables and constants get stored in there.
5. Stack Segment: I couldn’t make the CPU use it, but using software, all of the stack allocated for processes is in here.
6. Process Segment: Every Process descriptor entry is stored in here. The process descriptor entries are allocated dynamically in this segment. Also, messages between processes take space in this segment, they are also allocated dynamically.

For both the stack and process segments, a memory manager was implemented for each of them. These memory managers allocate chunks from the free space, and have free lists to store unused old allocations.

Process Table:

The process table is stored in the ProcessManager class. It holds the following information for each process entry:

- Pointer to CPU state.
- Pointer to stack.
- Stack size.
- PID.
- Pointer to the parent’s process entry.
- Process’s state.
- Pointer to sibling process. (The next child of the parent)
- pointer to the first child process.
- Number of child processes.
- Pointer to the waiting message list.

Multitasking:

A process manager class was implemented to handle task switching and their management. Since the class will handle process switching and whatsoever, it extends the interrupt handler class and is assigned as the interrupt handler for timer interrupts. All timer interrupts are handled by it directly. This class also handles forking, changing execution, and terminating operations for the system.

Timer interrupts do not switch tasks, instead, pressing the TAB button on the keyboard or left clicking on the mouse will trigger a context switch. This was done to make task switching easier to perform tests, where user input will be requested.

System Calls:

System calls are static functions that belong to the class Kernel. When these methods are called, they trigger the interrupt 0x80 and provide parameters to tell the handler what is the call. The main Kernel class is also an interrupt handler, which directly receives and handles system calls. The real implementation of the system calls can be found as private methods of the Kernel class. These methods always start with "sys_" in their name. This is done to ensure that they are provided with the objects and parameters they need using data fields inside the Kernel class.

Input Devices:

Both mouse and keyboard input is available. Most of the keyboard's buttons are supported. Mouse clicks and cursor movements are also supported. For task switching, the keyboard's TAB button and the left mouse button can be used to trigger a single task switch.

Keyboard input will be stored in a 256 byte buffer, and will only be used after a gets or getchar call is made, and the Enter key on the keyboard is pressed right after them.

Testing:

Three flavors were implemented, the only difference among them is the process init, which has different behavior in each flavor. The parameters of the created test processes are global variables, because the system call execve resets the stack, which results in losing whatever local variables that were given as parameters in the original process. Input for these tests will be provided by the user as keyboard input. Below are examples of running instances of the OS.

Note: No screenshots that include user input were provided, but I assure you it works as expected. Taking screenshots is a bit annoying so it was avoided to avoid the hassle. Other than the user input, every thing will look the same.

Flavor1:

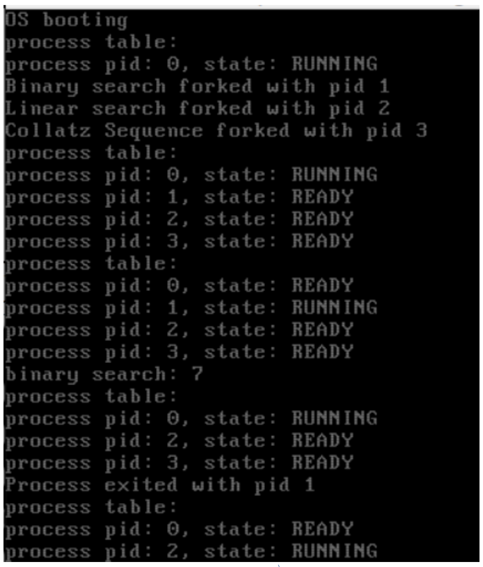
Forks three processes, one executing binary search, one executing linear search, and another printing Collatz sequences. The info about the processes is as follows:

- BinarySearch:
  - Input : {10, 20, 30, 50, 60, 80, 100, 110, 130, 170}, target = 110;
  - Expected Output : 7.

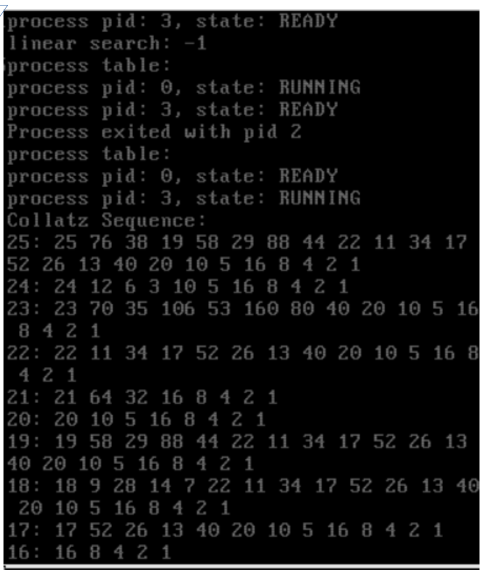
- LinearSearch
  - Input : {10, 20, 80, 30, 60, 50, 110, 100, 130, 170} x = 175;
  - Expected Output : -1.

- Collatz Sequence:
  - Input : 25.
  - Expected Output : All of the Collatz sequences of the numbers from 1 to the given number. See the output screenshots.

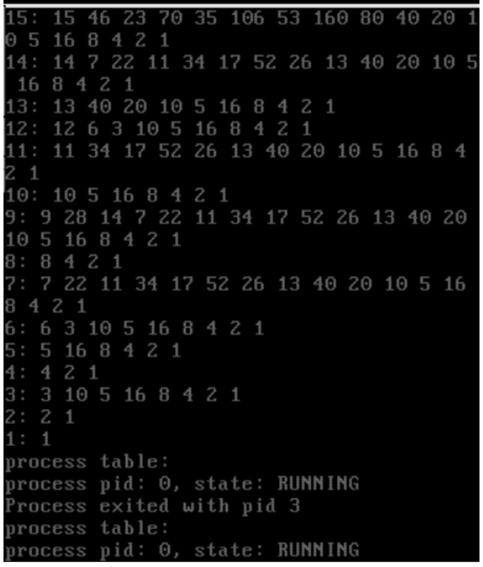
The process init will fork the child processes, and then using waitpid, it will detect when a process of the above exits. The output is as follows:



```
OS booting
process table:
process pid: 0, state: RUNNING
Binary search forked with pid 1
Linear search forked with pid 2
Collatz Sequence forked with pid 3
process table:
process pid: 0, state: RUNNING
process pid: 1, state: READY
process pid: 2, state: READY
process pid: 3, state: READY
process table:
process pid: 0, state: READY
process pid: 1, state: RUNNING
process pid: 2, state: READY
process pid: 3, state: READY
binary search: 7
process table:
process pid: 0, state: RUNNING
process pid: 2, state: READY
process pid: 3, state: READY
Process exited with pid 1
process table:
process pid: 0, state: READY
process pid: 2, state: RUNNING
```



```
process pid: 3, state: READY
linear search: -1
process table:
process pid: 0, state: RUNNING
process pid: 3, state: READY
Process exited with pid 2
process table:
process pid: 0, state: READY
process pid: 3, state: RUNNING
Collatz Sequence:
25: 25 76 38 19 58 29 88 44 22 11 34 17
52 26 13 40 20 10 5 16 8 4 2 1
24: 24 12 6 3 10 5 16 8 4 2 1
23: 23 70 35 106 53 160 80 40 20 10 5 16
 8 4 2 1
22: 22 11 34 17 52 26 13 40 20 10 5 16 8
 4 2 1
21: 21 64 32 16 8 4 2 1
20: 20 10 5 16 8 4 2 1
19: 19 58 29 88 44 22 11 34 17 52 26 13
40 20 10 5 16 8 4 2 1
18: 18 9 28 14 7 22 11 34 17 52 26 13 40
 20 10 5 16 8 4 2 1
17: 17 52 26 13 40 20 10 5 16 8 4 2 1
16: 16 8 4 2 1
```



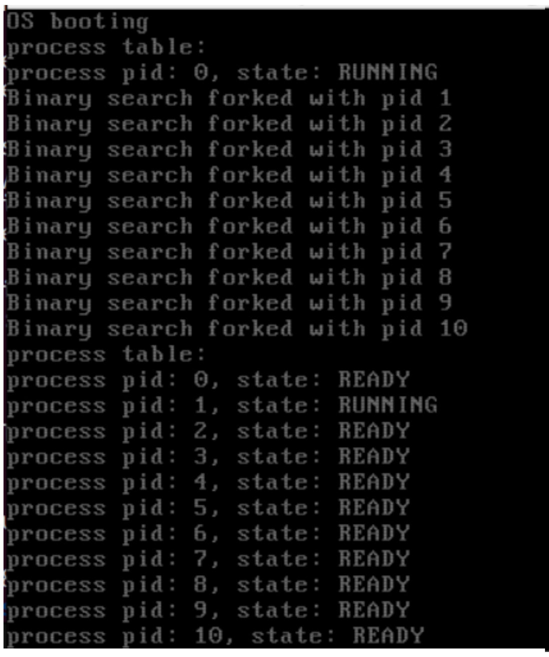
```
15: 15 46 23 70 35 106 53 160 80 40 20 1
0 5 16 8 4 2 1
14: 14 7 22 11 34 17 52 26 13 40 20 10 5
 16 8 4 2 1
13: 13 40 20 10 5 16 8 4 2 1
12: 12 6 3 10 5 16 8 4 2 1
11: 11 34 17 52 26 13 40 20 10 5 16 8 4
2 1
10: 10 5 16 8 4 2 1
9: 9 28 14 7 22 11 34 17 52 26 13 40 20
10 5 16 8 4 2 1
8: 8 4 2 1
7: 7 22 11 34 17 52 26 13 40 20 10 5 16
8 4 2 1
6: 6 3 10 5 16 8 4 2 1
5: 5 16 8 4 2 1
4: 4 2 1
3: 3 10 5 16 8 4 2 1
2: 2 1
1: 1
process table:
process pid: 0, state: RUNNING
Process exited with pid 3
process table:
process pid: 0, state: RUNNING
```

Flavor2:

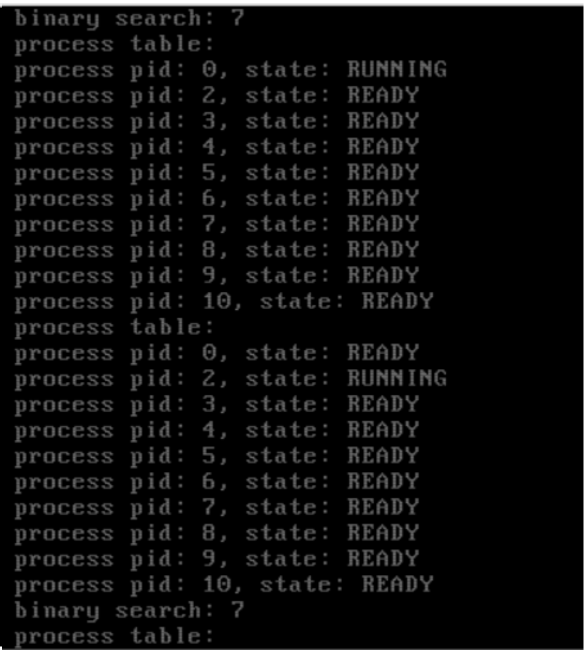
forks 10 processes executing binary search. The info about the processes is as follows:

- BinarySearch:
    - Input : {10, 20, 30, 50, 60, 80, 100, 110, 130, 170}, target = 110;
    - Expected Output : 7.

The process init will fork the child processes, and then using waitpid, it will detect when a process of the above exits. The output is as follows:

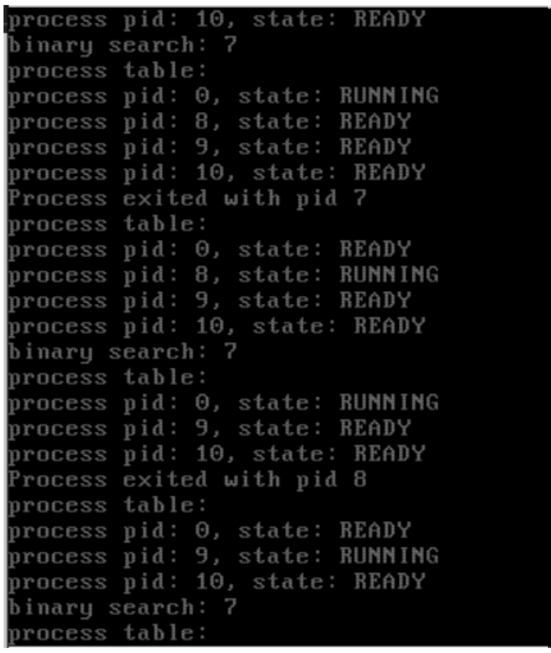


```
OS booting
process table:
process pid: 0, state: RUNNING
Binary search forked with pid 1
Binary search forked with pid 2
Binary search forked with pid 3
Binary search forked with pid 4
Binary search forked with pid 5
Binary search forked with pid 6
Binary search forked with pid 7
Binary search forked with pid 8
Binary search forked with pid 9
Binary search forked with pid 10
process table:
process pid: 0, state: READY
process pid: 1, state: RUNNING
process pid: 2, state: READY
process pid: 3, state: READY
process pid: 4, state: READY
process pid: 5, state: READY
process pid: 6, state: READY
process pid: 7, state: READY
process pid: 8, state: READY
process pid: 9, state: READY
process pid: 10, state: READY
```

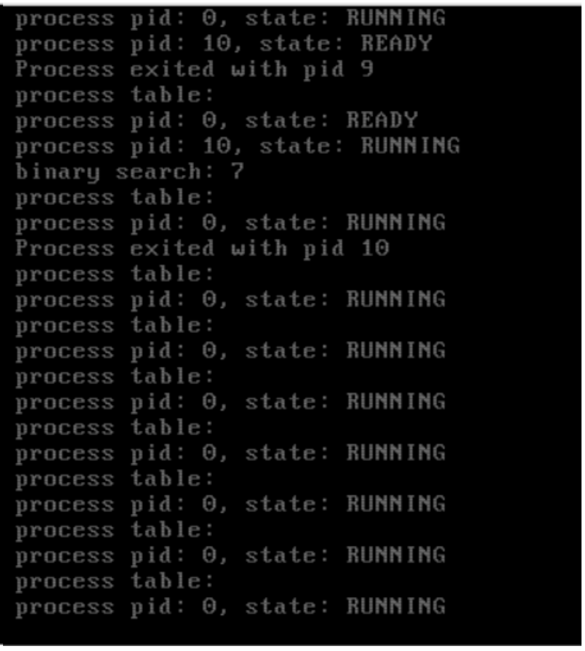


```
binary search: 7
process table:
process pid: 0, state: RUNNING
process pid: 2, state: READY
process pid: 3, state: READY
process pid: 4, state: READY
process pid: 5, state: READY
process pid: 6, state: READY
process pid: 7, state: READY
process pid: 8, state: READY
process pid: 9, state: READY
process pid: 10, state: READY
process table:
process pid: 0, state: READY
process pid: 2, state: RUNNING
process pid: 3, state: READY
process pid: 4, state: READY
process pid: 5, state: READY
process pid: 6, state: READY
process pid: 7, state: READY
process pid: 8, state: READY
process pid: 9, state: READY
process pid: 10, state: READY
binary search: 7
process table:
```

Some processes later…..



```
process pid: 10, state: READY
binary search: 7
process table:
process pid: 0, state: RUNNING
process pid: 8, state: READY
process pid: 9, state: READY
process pid: 10, state: READY
Process exited with pid 7
process table:
process pid: 0, state: READY
process pid: 8, state: RUNNING
process pid: 9, state: READY
process pid: 10, state: READY
binary search: 7
process table:
process pid: 0, state: RUNNING
process pid: 9, state: READY
process pid: 10, state: READY
Process exited with pid 8
process table:
process pid: 0, state: READY
process pid: 9, state: RUNNING
process pid: 10, state: READY
binary search: 7
process table:
```



```
process pid: 0, state: RUNNING
process pid: 10, state: READY
Process exited with pid 9
process table:
process pid: 0, state: READY
process pid: 10, state: RUNNING
binary search: 7
process table:
process pid: 0, state: RUNNING
Process exited with pid 10
process table:
process pid: 0, state: RUNNING
process table:
process pid: 0, state: RUNNING
process table:
process pid: 0, state: RUNNING
process table:
process pid: 0, state: RUNNING
process table:
process pid: 0, state: RUNNING
process table:
process pid: 0, state: RUNNING
process table:
process pid: 0, state: RUNNING
```

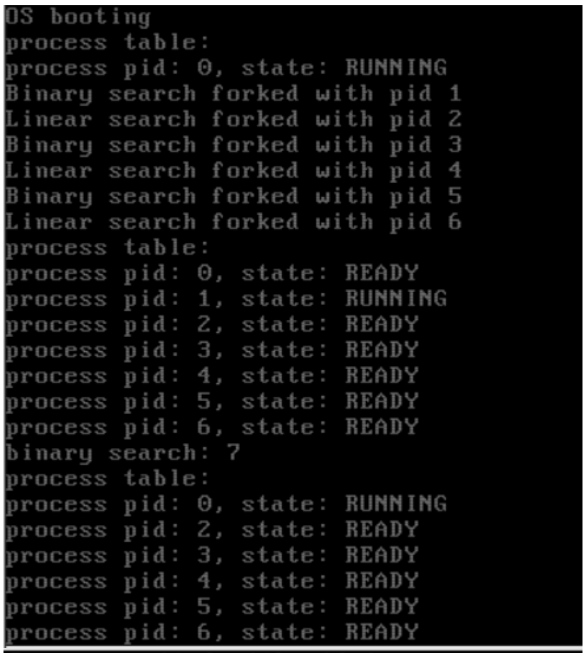
Flavor3:

forks 6 processes, 3 executing binary search, and another 3 executing linear search. The info about the processes is as follows:

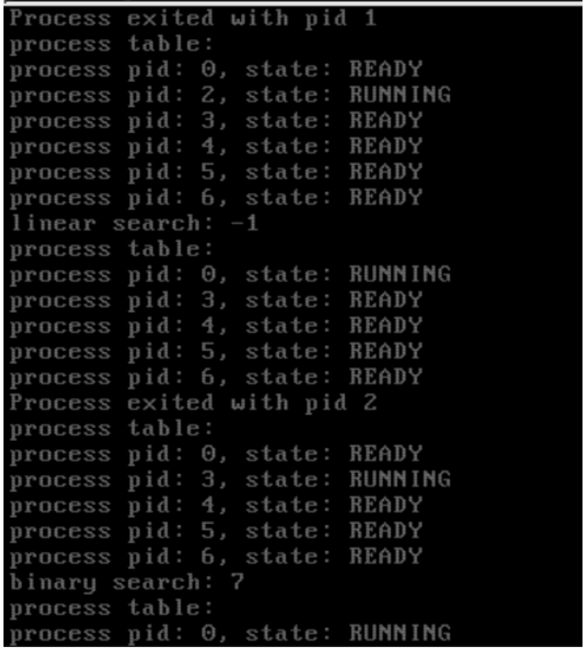
- BinarySearch:
  - Input : {10, 20, 30, 50, 60, 80, 100, 110, 130, 170}, target = 110;
  - Expected Output : 7.

- LinearSearch
  - Input : {10, 20, 80, 30, 60, 50, 110, 100, 130, 170} x = 175;
  - Expected Output : -1.

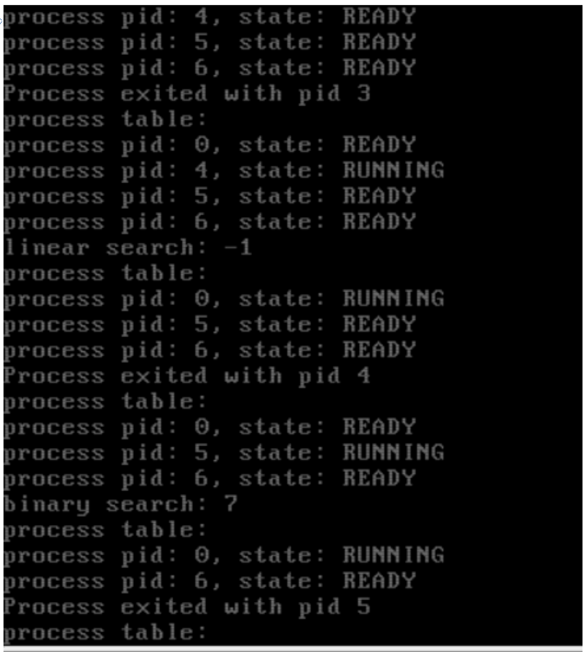
The process init will fork the child processes, and then using waitpid, it will detect when a process of the above exits. The output is as follows:



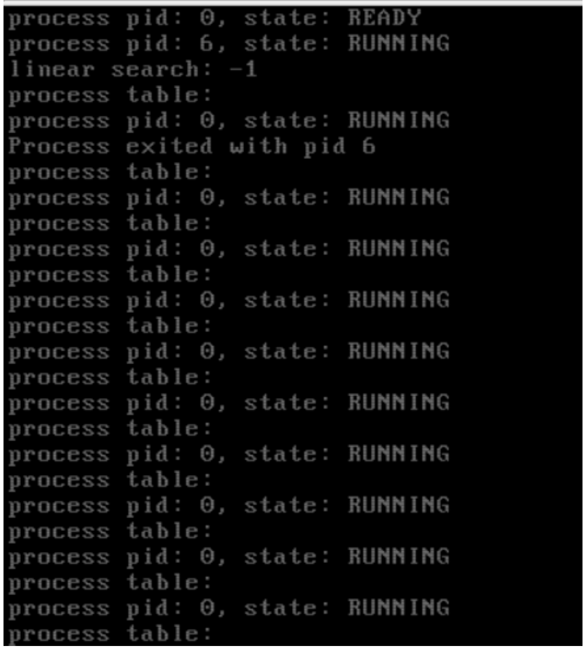
```
OS booting
process table:
process pid: 0, state: RUNNING
Binary search forked with pid 1
Linear search forked with pid 2
Binary search forked with pid 3
Linear search forked with pid 4
Binary search forked with pid 5
Linear search forked with pid 6
process table:
process pid: 0, state: READY
process pid: 1, state: RUNNING
process pid: 2, state: READY
process pid: 3, state: READY
process pid: 4, state: READY
process pid: 5, state: READY
process pid: 6, state: READY
binary search: 7
process table:
process pid: 0, state: RUNNING
process pid: 2, state: READY
process pid: 3, state: READY
process pid: 4, state: READY
process pid: 5, state: READY
process pid: 6, state: READY
```



```
Process exited with pid 1
process table:
process pid: 0, state: READY
process pid: 2, state: RUNNING
process pid: 3, state: READY
process pid: 4, state: READY
process pid: 5, state: READY
process pid: 6, state: READY
linear search: -1
process table:
process pid: 0, state: RUNNING
process pid: 3, state: READY
process pid: 4, state: READY
process pid: 5, state: READY
process pid: 6, state: READY
Process exited with pid 2
process table:
process pid: 0, state: READY
process pid: 3, state: RUNNING
process pid: 4, state: READY
process pid: 5, state: READY
process pid: 6, state: READY
binary search: 7
process table:
process pid: 0, state: RUNNING
```



```
process pid: 4, state: READY
process pid: 5, state: READY
process pid: 6, state: READY
Process exited with pid 3
process table:
process pid: 0, state: READY
process pid: 4, state: RUNNING
process pid: 5, state: READY
process pid: 6, state: READY
linear search: -1
process table:
process pid: 0, state: RUNNING
process pid: 5, state: READY
process pid: 6, state: READY
Process exited with pid 4
process table:
process pid: 0, state: READY
process pid: 5, state: RUNNING
process pid: 6, state: READY
binary search: 7
process table:
process pid: 0, state: RUNNING
process pid: 6, state: READY
Process exited with pid 5
process table:
```



```
process pid: 0, state: READY
process pid: 6, state: RUNNING
linear search: -1
process table:
process pid: 0, state: RUNNING
Process exited with pid 6
process table:
process pid: 0, state: RUNNING
process table:
process pid: 0, state: RUNNING
process table:
process pid: 0, state: RUNNING
process table:
process pid: 0, state: RUNNING
process table:
process pid: 0, state: RUNNING
process table:
process pid: 0, state: RUNNING
process table:
process pid: 0, state: RUNNING
process table:
process pid: 0, state: RUNNING
process table:
```

Important Notes:

- Execution details can be found in the README.txt file.
- “Binarysearch: 7” means that the binary search function returned 7. And the same goes for linear search.
- Make sure that your VirtualBox has a virtual machine called “My Micro Kernel”, and have it point to the compiled .iso file, in order for the makefile’s run command to work.