Gebze Technical University

Department of Computer Engineering

CSE312 Spring 2024

Operating Systems

Homework #01

Mehmet Mahir Kayadelen, Student no. 210104004252

At first I implemented the 4 system calls.

-Fork

-Exit

-Waitpid

-Execve

1-Fork

```
uint32_t TaskManager::Fork(CPUState* cpustate) {
    Task* cur_Task = tasks[currentTask];

    Task* child = new Task(gdt, 0, this);

    // copy parent stack to the child stack
    for(int i = 0; i < sizeof(cur_Task->stack); i++){
        child->stack[i] = cur_Task->stack[i];
    }

    // calculating offset of current esp(cpustate)
    uint32_t parentEspRelativeToStackStart = (uint32_t)cpustate - (uint32_t)cur_Task->stack;

    // calculating childs current esp(cpustate) with calculating  parent esp offset
    child->cpustate = (CPUState *) (child->stack + parentEspRelativeToStackStart);


    uint32_t parentEbpRelativeToStackStart = (uint32_t)cpustate->ebp - (uint32_t)cur_Task->stack ;
    child->cpustate->ebp = (uint32_t) (child->stack + parentEbpRelativeToStackStart);




    child->pPid = cur_Task->pId;
    child->cpustate->ecx = 0;

    AddTask(child);

    printf("\n");

    return child->pId;
}
```

The current task being executed is retrieved from the tasks array using currentTask as the index. This current task is stored in the pointer cur_Task.

A new task object (child) is created. This task represents the forked (child) task.

The constructor for the Task class is called with gdt, 0, and this as arguments. gdt represents the Global Descriptor Table, and this is a pointer to the TaskManager instance.

The entire stack of the parent task (cur_Task) is copied to the child task's stack.

Loop iterates through each byte of the parent's stack and copies it to the corresponding location in the child's stack.

The offset of the parent's stack pointer (esp) relative to the start of the parent's stack is calculated. This is done by subtracting the base address of the parent's stack from the address of the cpustate (which includes the esp).

The child's stack pointer (esp) is calculated by adding the previously computed offset (parentEspRelativeToStackStart) to the base address of the child's stack.

This effectively positions the child's esp at the same relative position within the child's stack as the parent's esp within the parent's stack.

Similar to the esp, the offset of the parent's base pointer (ebp) relative to the start of the parent's stack is calculated

The child's base pointer (ebp) is then set to the same relative position within the child's stack as the parent's ebp within the parent's stack.

The child's parent process ID (pPid) is set to the current task's process ID (pId).

The ecx register in the child's CPU state is set to 0. This is done to differentiate between the parent and child processes after a fork, as the child process returns 0 from the fork call.

```
uint8_t myos::fork(){
    uint8_t childPid;
    asm("int $0x80" : "=c" (childPid) : "a" (SYSCALLS::FORK));
    return childPid;
}
```

The fork function uses inline assembly to invoke a system call for forking a new process. It uses the int $0x80 instruction to make the system call, passes the syscall number for fork via the eax register, and stores the result (child process ID) in the ecx register. The child process ID is then returned as an 8-bit unsigned integer.

2-Exit

```
bool TaskManager::Exit(){
    tasks[currentTask]->taskState = FINISHED;

    return true;
}
```

This system call basically changes current task’s taskState into FINISHED and returns.

3-Waitpid

```
bool TaskManager::Waitpid(uint32_t esp){

    CPUState *cpustate = (CPUState *) esp;
    uint32_t pid = cpustate->ebx;

    if(tasks[currentTask]->pId == pid || pid == 0){ // for self waiting
        return false;
    }

    int index = -1; // default value of non found
    for(int i =0 ; i < numTasks;i++){ // iterates through tasks and finds if waiting Process exist
        if(tasks[i]->pId == pid){
            index = i;
            break;
        }
    }

    if(index == -1) // process doesnt exist
        return false;

    if(numTasks<= index || tasks[index]->taskState == FINISHED){
        printf("returned \n");
        return false;
    }

    tasks[currentTask]->cpustate=cpustate;
    tasks[currentTask]->waitPid = pid;
    tasks[currentTask]->taskState = WAITING;
    return true;
}
```

It takes a uint32_t argument esp, which represents the stack pointer of the CPU state.

The function returns a bool indicating whether the wait operation was successful or not.

The esp argument is cast to a CPUState pointer, cpustate. This allows access to the CPU state (registers and other information) at the time of the call.

Checks if the current task's PID is the same as the PID to wait for or if the PID is 0.If either condition is true, the function returns false, as a task cannot wait for itself or for a PID of 0

A variable index is initialized to -1 to indicate that the target task has not been found.

The loop iterates over all tasks to find a task with the specified PID.

If a task with the specified PID is found, index is set to the task's index and the loop is exited.

If index remains -1, it means no task with the specified PID was found, so the function returns false.

The CPU state of the current task is updated to the cpustate obtained earlier.

The current task's waitPid is set to the PID it is waiting for.

The current task's state is set to WAITING.

```
void myos::waitpid(common::uint8_t wPid){
    asm("int $0x80" : : "a" (SYSCALLS::WAITPID), "b" (wPid));
}
```

The waitpid function uses inline assembly to invoke a system call for waiting on a specific process. It sets up the necessary registers (eax for the syscall number and ebx for the PID) and then triggers interrupt 0x80 to make the system call. This function does not return a value.

4-Execve

```
int TaskManager::Execve(void (*entrypoint)()){

    tasks[currentTask]->cpustate = (CPUState*) (tasks[currentTask]->stack + 4096 - sizeof(CPUState));

    tasks[currentTask]->cpustate->eip = (uint32_t) entrypoint; // move instructor pointer to the function

    tasks[currentTask]->cpustate->cs = gdt->CodeSegmentSelector(); // for kernel

    tasks[currentTask]->cpustate->ebp = 0; // for make sure there will be no problem
    tasks[currentTask]->cpustate->eflags = 0x202;

    return 0;
}
```

It takes a pointer to a function (entrypoint) as an argument. This pointer represents the new entry point (the starting address of the function to execute) for the current task.

The CPU state (cpustate) of the current task is set to point to a location within the task's stack.This location is calculated by taking the base address of the task's stack, adding 4096 (indicating the size of the stack), and then subtracting the size of a CPUState structure.This sets up the cpustate at the top of the task's stack

The instruction pointer (eip) in the cpustate is set to the address of the entrypoint function.

This means when the task is scheduled to run, it will start executing from the entrypoint function.

The code segment register (cs) in the cpustate is set to the code segment selector obtained from the Global Descriptor Table (gdt). This ensures that the task will use the correct code segment for execution, set up for kernel mode.

The base pointer register (ebp) in the cpustate is set to 0.

This is done to ensure there are no issues with the base pointer, as it may have been used previously by the task.

```
int myos::execve(void entrypoint()){
    int result;

    asm("int $0x80" : "=c" (result) : "a" (SYSCALLS::EXECVE), "b" ((uint32_t)entrypoint));

    return result;
}
```

The execve function uses inline assembly to invoke a system call for executing a new program. It sets up the necessary registers (eax for the syscall number and ebx for the entry point address) and triggers interrupt 0x80 to make the system call. The result of the system call is stored in the result variable and returned by the function. This function allows the current process to start executing a new function specified by the entrypoint argument.

I made improvements and configurations in the scheduler function, here is the full schedule function.

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


    // idleprocess

    static int idleTime = 0;

    if(idleTask->taskState == RUNNING)
    {
        idleTask->cpustate = cpustate;
    }
    else if(idleTask->taskState == READY)
    {
        idleTask->taskState = RUNNING;
        if(currentTask >= 0)
            tasks[currentTask]->cpustate = cpustate;
    }

    if(++idleTime % IDLE != 0)
    {
        return idleTask->cpustate;
    }
    else
    {
        idleTime = 0;
        idleTask->taskState = READY;
    }

    // idleprocess



    int start = currentTask == -1 ? 0 : currentTask;

    do{
        if(++currentTask >= numTasks)
            currentTask %= numTasks;
        if(tasks[currentTask]->taskState == WAITING )
        {
            if(tasks[tasks[currentTask]->waitPid -1]->taskState == FINISHED)
                tasks[currentTask]->taskState = READY;
        }
    }
    while(tasks[currentTask]->taskState != READY);


    if(tasks[start]->taskState == RUNNING)
        tasks[start]->taskState = READY;

    tasks[currentTask]->taskState = RUNNING;
    printTasks();
    return tasks[currentTask]->cpustate;



[redacted]
```

idleprocess is a void function without a function. I used it to slow down the ProcessTable while printing to increase readability.

The variable start is set to 0 if currentTask is -1 (no current task); otherwise, it is set to currentTask

The currentTask index is incremented. If it exceeds the number of tasks, it wraps around to 0 using the modulo operation.

If the task state is WAITING, it checks if the task it is waiting for has finished. If so, it sets the current task's state to READY.

This loop continues until a task with the READY state is found.

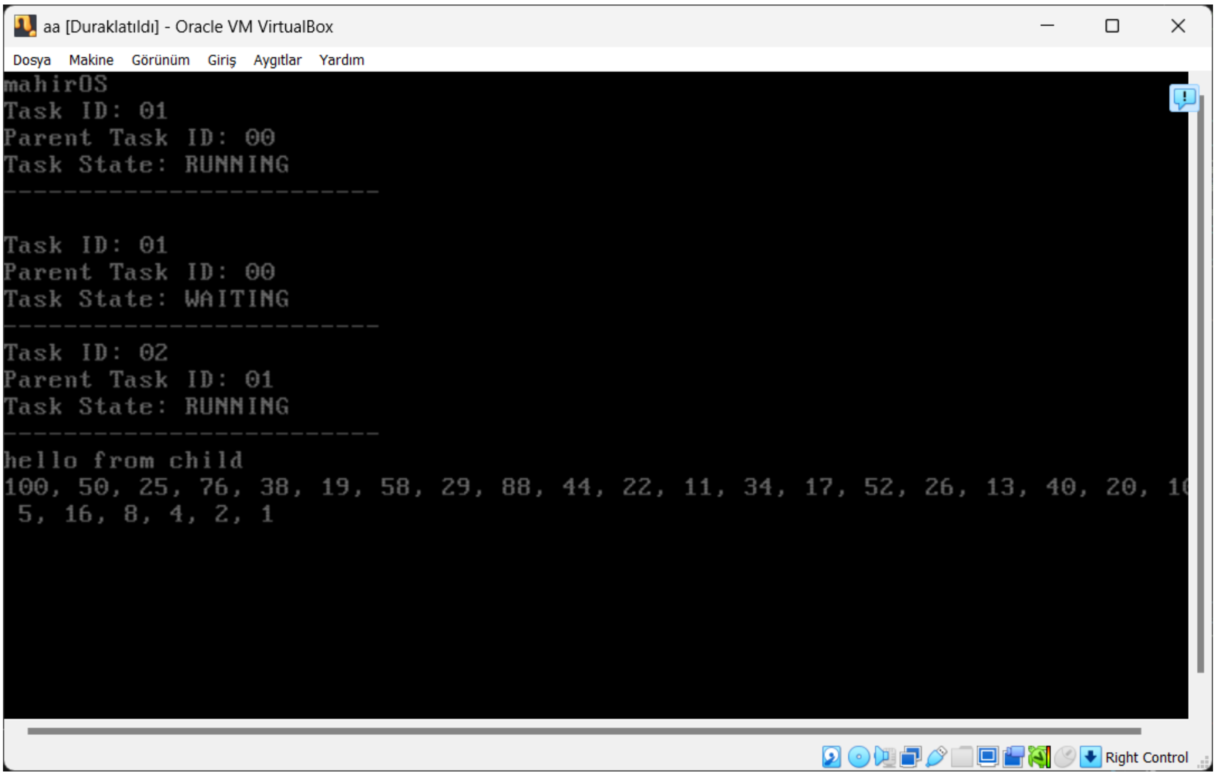
If the task that was initially running (start) is still in the RUNNING state, its state is changed to READY.

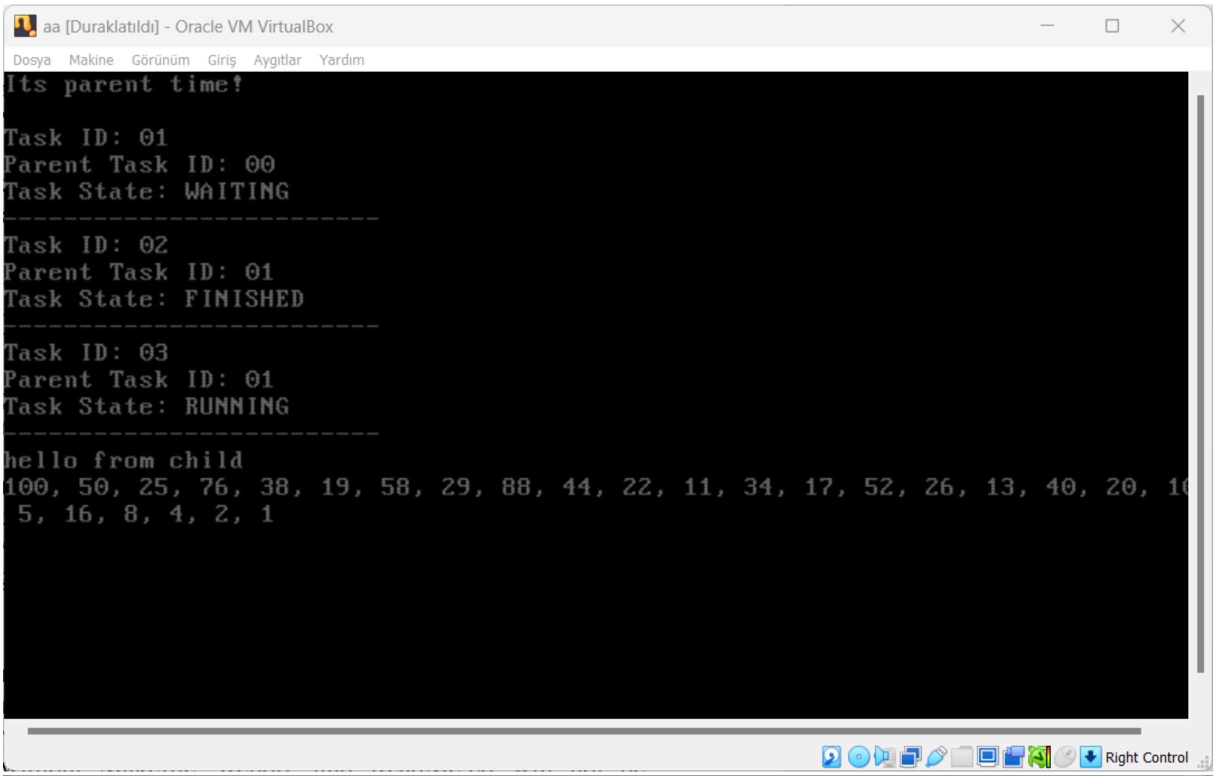
The newly selected task's state is set to RUNNING.

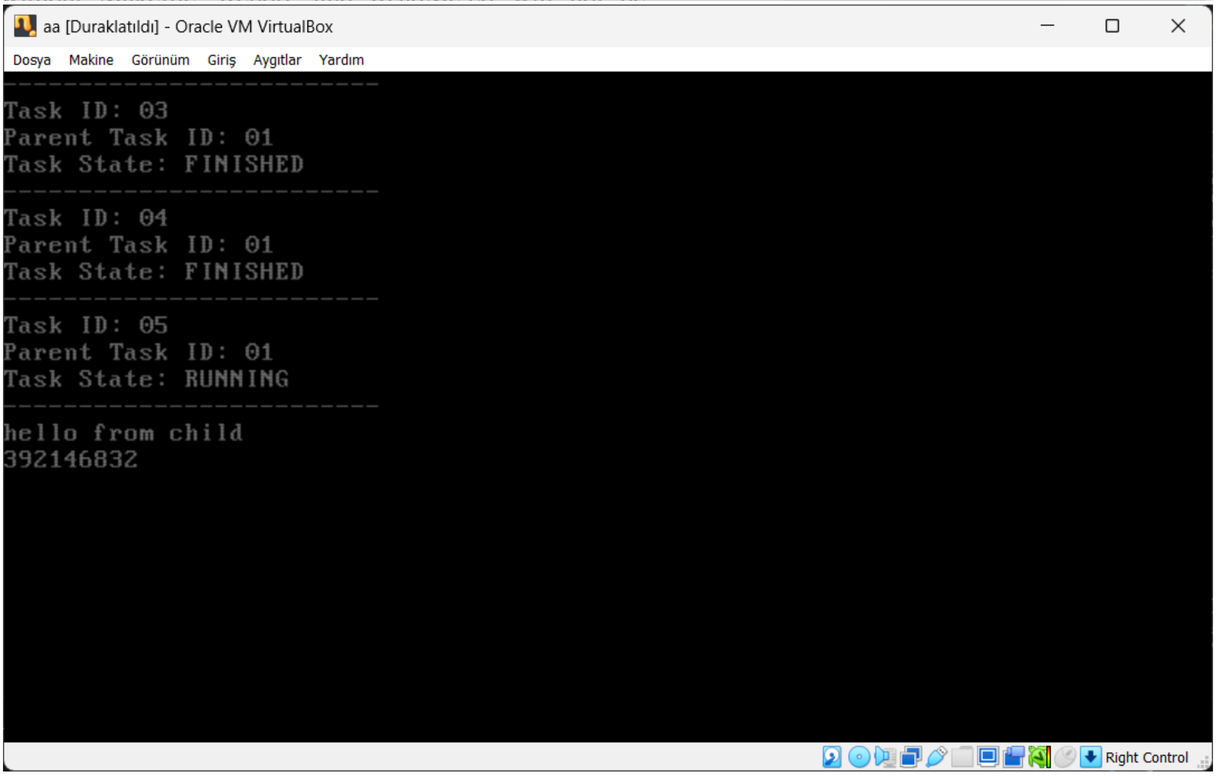
The function printTasks() is called, to print the status of all tasks in each Schedule.

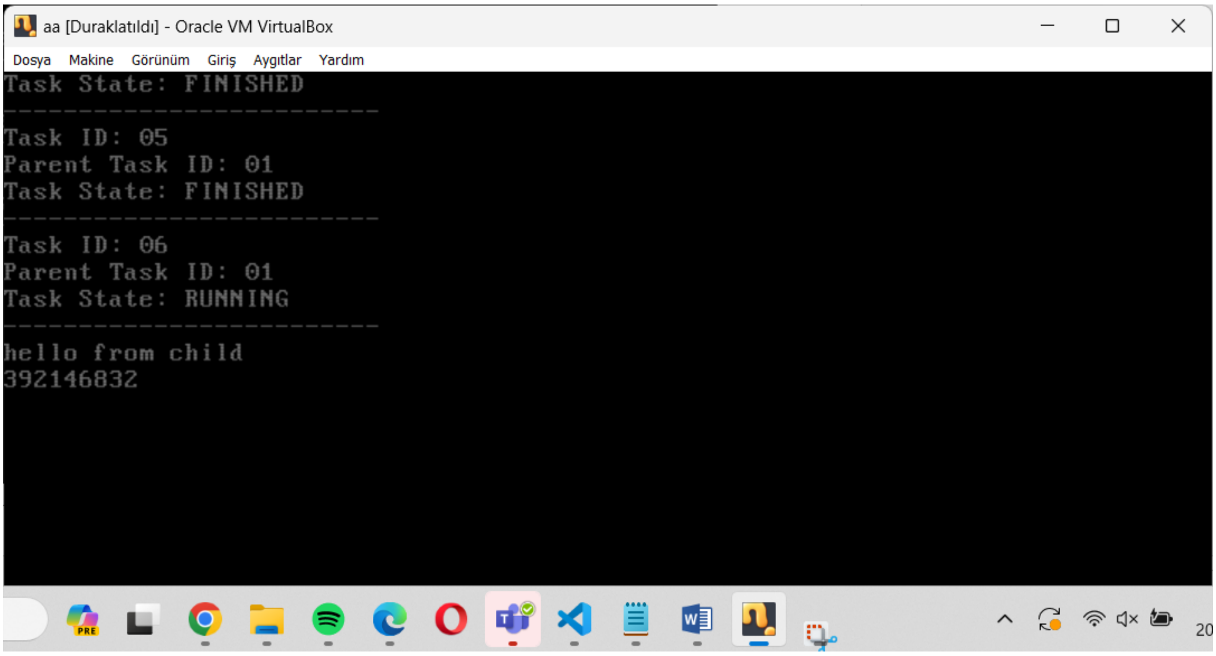
The function returns the CPU state of the newly selected task.

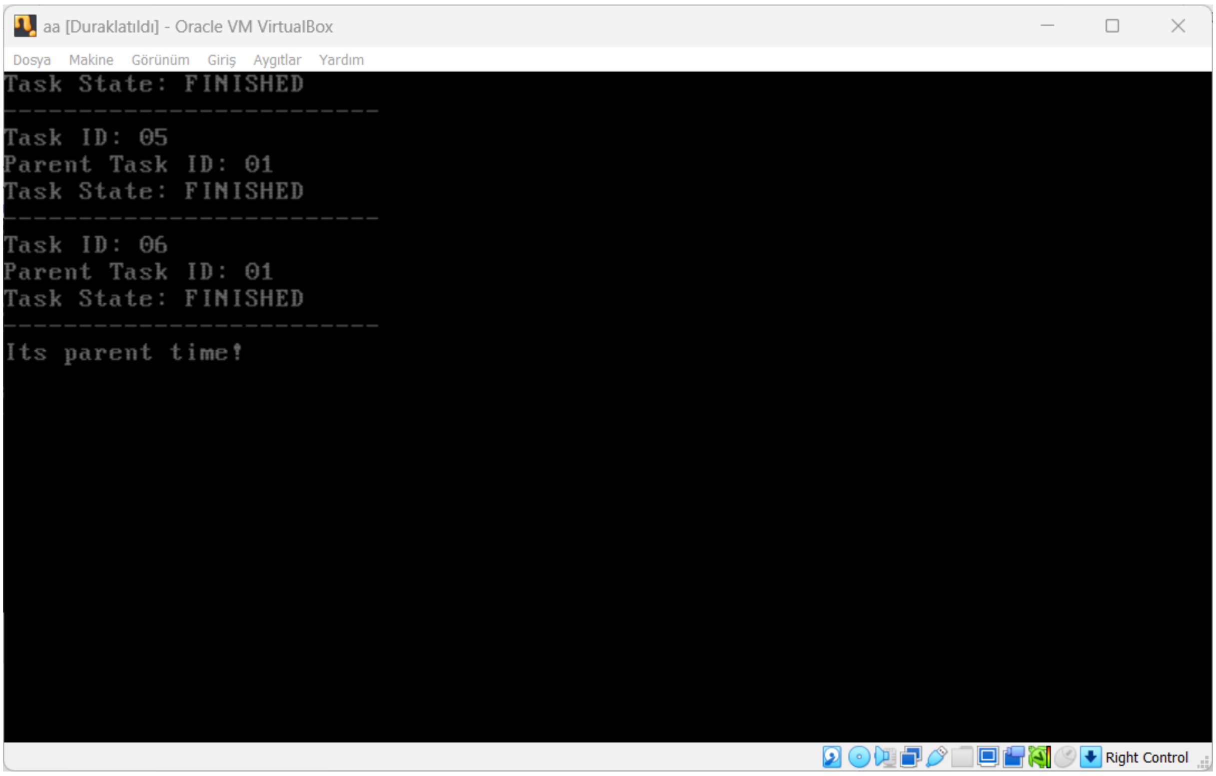
Test:




aa [Duraklatıldı] - Oracle VM VirtualBox
Dosya Makine Görünüm Giriş Aygıtlar Yardım
Its parent time!
Task ID: 01
Parent Task ID: 00
Task State: WAITING
Task ID: 02
Parent Task ID: 01
Task State: FINISHED
Task ID: 03
Parent Task ID: 01
Task State: RUNNING
hello from child
100, 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 1
5, 16, 8, 4, 2, 1
Right Control


aa [Duraklatıldı] - Oracle VM VirtualBox
Dosya Makine Görünüm Giriş Aygıtlar Yardım
Task ID: 03
Parent Task ID: 01
Task State: FINISHED
Task ID: 04
Parent Task ID: 01
Task State: FINISHED
Task ID: 05
Parent Task ID: 01
Task State: RUNNING
hello from child
392146832
Right Control





My init function forks a total of 6 times. In the first 3 forks, child processes run collatz, and in the next 3 forks, child processes run the long program. The parent process waits for the child processes to finish and then continues. I print the Process Table in each context switch operation. After the 3rd context switch, the output of function is not visible well because the Process Table output expands too much. I used 6 forks and last child runs program with execve system call.

Here is my test task(init)