CSE 312

HW BONUS

Burak Kocausta

1901042605

Contents

1- Install, Compilation and Run
2- Updated Parts
3- General Design
4- Code Explanations
5- Test Cases and Results

1- Install, Compilation and Run

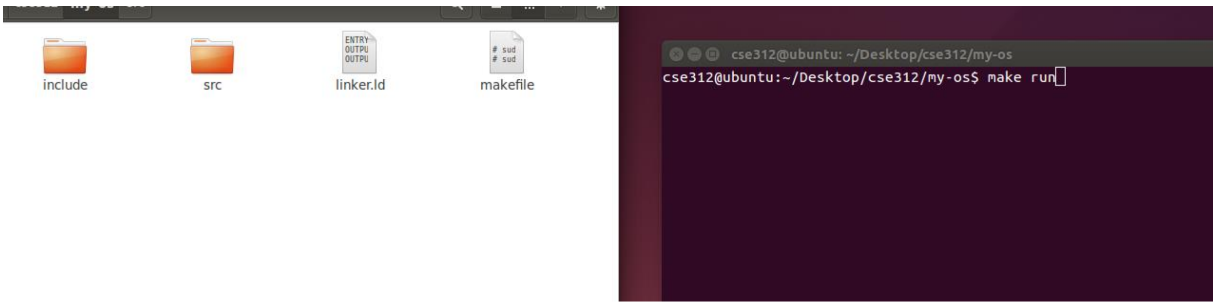
All parts are compiles and runs with ‘make run’ command. ISO image must be installed on virtualbox after it is created with `make mykernel.iso´ command. If it is tested on ubuntu environment, make run will compiles, then starts vm.

I added macros to kernel.cpp, and multitasking.cpp for testing purposes. They are explained in the test cases and results part.

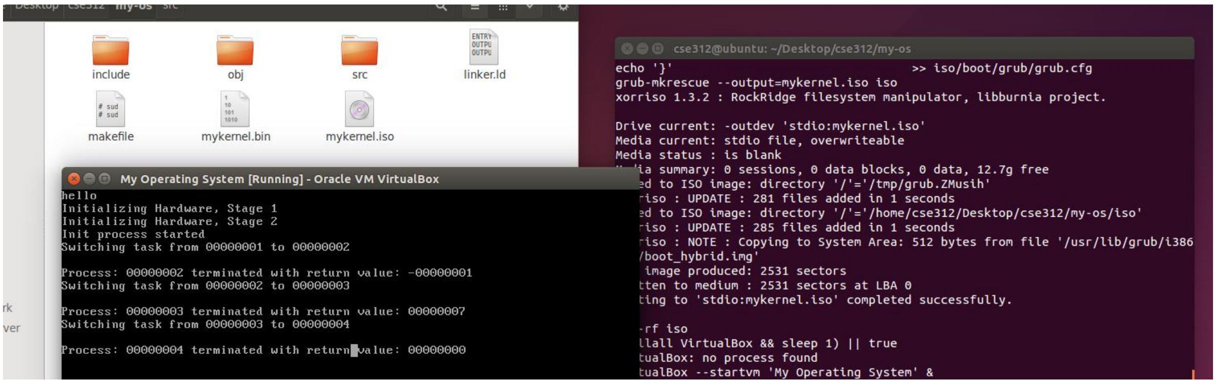
Install

Iso image is created with `make mykernel.iso` command. After this is created, it must be installed to virtualbox vm, as Engelmann explained.

Compilation



Run



2- Updated Parts

- Fork, waitpid, read are implemented.
- Other processes are forked by init process, they are added to memory using fork.
- Switch is happening with left mouse click, and any keyboard click except newline. Cannot switch during read.
- Timer interrupt happens once in 256.
- Arguments are taken from keyboard for collatz, binary, and linear search.

3- General Design

I started with the incomplete parts of the first homework. I don't directly implement on top of the shared works, but I got help for implementing fork system call. Their fork is not posix compliant either but they work for adding process, so I added fork system call with the help of shared works. Fork is implemented twice, one of them is simple fork, it only copies the parent process. Other one does fork exec together. It adds the process to the table and changes its memory image. One of the shared works(hw1) did it like that, and I implemented my fork with the help of that implementation. After implementing fork, I implemented waitpid, to implement this I had to add process id of waited process to the members of the process class. This information is also held in process table. It is needed, because when scheduler, tries to schedule blocked process it needs to know if child is terminated or not. After adding waitpid system call, BLOCKED state is used for that. In the first one this state could not be used, because of the failed implementation of fork on previous homework.

As stated in the instructions, init process adds all processes using fork. In the first homework, I added them in compile time, but in this homework, they are added using fork by init. I implemented the keyboard, and mouse click interrupts. Context switch immediately occurs if one of the keyboard buttons is pressed or left click of the mouse is clicked. Switch can happen with timer interrupt also. It happens once in 256 timer interrupt, I made it the number greater than 10 because it happens so fast. And every switch it is printed that which switching is happened (timer, keyboard, mouse). I added a counter to count number of interrupts.

Lastly, I added the read function to get arguments from keyboard, each key press means interrupt, so to make read size of the read must given as parameter. It reads all the size. During reading, I prevented the interrupt, in order to communicate between managers(driver, task, interrupt) I stated a global flag, therefore when read I/O is happening, they know it and switching does not happen. For collatz, binary, and linear reading a decimal number is needed, therefore I implemented a function that reads integer from user and it reads till the newline character. It uses system call read.

Rest of the homework is same with previous homework, I changed the collatz implementation, in the first homework I stored the numbers in an array, and I realized it is the source of some problems because of using too much memory, and it has no problem in this version.

4- Code Explanations

Process States

```
// process states
enum ProcessState {
    RUNNING,
    BLOCKED,
    READY,
    TERMINATED
};
```

Process Class

```
// process class
class Process {

friend class TaskManager;

private:
    // process ID
    common::uint32_t pid;

    // parent process ID
    common::uint32_t ppid;

    // process waiting for
    common::uint32_t waitPid;

    // child processes
    common::uint32_t children[256];

    // return value
    int* retval;

    // process state
    ProcessState state;

    // number of children
    common::uint32_t numChildren;
public:
```

```
public:
    Process(common::uint32_t pid, common::uint32_t ppid);
    Process();
    ~Process();

    // Add child process
    void AddChild(common::uint32_t pid);

    // Remove child process
    void RemoveChild(common::uint32_t pid);

    // getters
    common::uint32_t GetPid();
    common::uint32_t GetPpid();
    int* GetReturnValue();
    ProcessState GetState();
    common::uint32_t GetNumChildren();

    // setters
    void SetPid(common::uint32_t pid);
    void SetPpid(common::uint32_t ppid);
    void SetReturnValue(int* retval);
    void SetState(ProcessState state);
    void SetNumChildren(common::uint32_t numChildren);
```

- This class is mostly used by task manager. So I made it friend. I added waitPid field to know which child is waited if process is blocked.

Process Table

```
// holds space for all processes
// pid = 0 means inactive process
struct ProcessTable
{
    Process processes[256];
    int numProcesses;
};
```

Each process is accessed with process id as an offset in this array.

Task Class

```cpp
class Task
{
friend class TaskManager;
private:
    common::uint8_t stack[4096]; // 4 KiB
    CPUState* cpustate;

    // process ID
    common::uint32_t pid;
public:
    Task(GlobalDescriptorTable *gdt, void entrypoint());

    Task(GlobalDescriptorTable *gdt);

    Task();
    void setEntryPoint(void entrypoint());
    ~Task();

    // get entry point
    void* GetEntryPoint();

    // set the task
    void Set(Task* task);
};
```

- I added process id entry, and I implemented no parameter constructor. Also I implemented a setter function for the task.

Task Manager Class

```cpp
class TaskManager
{
private:
    Task tasks[256];
    int numTasks;
    int currentTask;

    // number of terminated tasks
    int terminatedTask;

    // process table
    ProcessTable processTable;
public:
    TaskManager(GlobalDescriptorTable *gdt);
    ~TaskManager();
    bool AddTask(Task* task);

    CPUState* Schedule(CPUState* cpustate);

    void PrintProcessTable();

    // take process to terminated state
    void TerminateProcess(int* returnVal);

    common::uint32_t GetCurrentPid();

    // system call implementations
    common::uint32_t ForkProcess(CPUState* cpustate);
    common::uint32_t ForkProcess2(CPUState* cpustate);
    void ExecProcess(CPUState* cpustate);
    void WaitProcess(CPUState* cpustate);

};
```

- I added process, and process table to the task manager class. I also updated the AddTask, Schedule functions. Additionally, I implemented member functions for fork, exec, termination, and getter for current pid. I changed the pointer to the task array to task array because of the reasons on design part. Task manager updates process table and process class for necessary actions. Task and Process class works in harmony. I added, other system call implementations, second fork process is for doing fork and exec together.

AddTask implementation

```
bool TaskManager::AddTask(Task* task)
{
    if(numTasks >= 256)
        return false;

    // assign minimum possible pid
    for (int i = 1; i < 256; i++)
    {
        if (processTable.processes[i].GetPid() == 0)
        {
            task->pid = i;
            break;
        }
    }

    // get process from process table
    Process* process = &(processTable.processes[task->pid]);

    // init is parent of all processes
    process->SetPid(task->pid);
    process->SetPpid(1);
    process->SetState(READY);
    process->SetNumChildren(0);

    // if not init, add to parent's children
    if (task->pid != 1)
    {
        Process* parent = &(processTable.processes[process->GetPpid()]);
        parent->AddChild(task->pid);
    }

    tasks[numTasks++].Set(task);
    return true;
}
```

TerminateProcess Function

```
void TaskManager::TerminateProcess(int* returnVal)
{
    // if it is init, do nothing
    if (tasks[currentTask].pid == 1)
        return;

    enterCritical();

    // if it is not init, set return value
    Process* process = &(processTable.processes[tasks[currentTask].pid]);
    process->SetReturnValue(returnVal);

    // set state to terminated
    process->SetState(TERMINATED);
    terminatedTask++;

    // print termination message
    printf("\nProcess: ");
    printfHex32(process->GetPid());
    printf(" terminated with return value: ");
    if (*returnVal < 0)
    {
        printf("-");
        printfHex32(*returnVal * -1);
    }
    else
        printfHex32(*returnVal);
    printf("\n");
    exitCritical();

    while(true);
}
```

It changes the process state to terminated, with this way it is known that this process won't be switched again.

Schedule Function

```cpp
CPUState* TaskManager::Schedule(CPUState* cpustate)
{
    // if there is no task, return
    if(numTasks - terminatedTask <= 0)
        return cpustate;

    // set current task's cpu state
    if(currentTask >= 0)
        tasks[currentTask].cpustate = cpustate;

    #ifdef SWITCH_PRINT_MODE
        if (numTasks - terminatedTask > 1)
        {
            printf("Switching task from ");
            printfHex32(tasks[currentTask].pid);
        }
    #endif

    // current task is on the ready state
    if (tasks[currentTask].pid != 0)
    {
        Process* process = &(processTable.processes[tasks[currentTask].pid]);
        if (process->GetState() == RUNNING)
            process->SetState(READY);
    }
```

```cpp
ProcessState state = TERMINATED;
// next task shouldn't be terminated
do
{
    // round robin till next task is not terminated
    if(++currentTask >= numTasks)
        currentTask %= numTasks;

    // if it is blocked, check if waiting process is terminated
    if (processTable.processes[tasks[currentTask].pid].GetState() == BLOCKED)
    {
        Process* process = &(processTable.processes[tasks[currentTask].pid]);
        if (processTable.processes[process->waitPid].GetState() == TERMINATED)
        {
            process->SetState(READY);
            process->waitPid = 0;
        }
    }

    state = processTable.processes[tasks[currentTask].pid].GetState();
} while (state == TERMINATED || state == BLOCKED);

// print switching message
```

Round robin is done like that, if next process is not terminated or blocked it is switched, if it is blocked it is checked that waited child is done or not.

Process Constructors

```cpp
Process::Process()
{
    this->pid = 0;
    this->ppid = 1;
    this->state = TERMINATED;
    this->retval = &default_val;
    // initialize children array
    for (int i = 0; i < 256; i++)
        this->children[i] = 0;
}

// create a new process with pid and ppid
Process::Process(common::uint32_t pid, common::uint32_t ppid)
{
    this->pid = pid;
    this->ppid = ppid;
    this->state = READY;
    this->retval = &default_val;
    // initialize children array
    for (int i = 0; i < 256; i++)
        this->children[i] = 0;
}
```

AddChild, and RemoveChild Functions

```cpp
void Process::AddChild(uint32_t pid)
{

    if (this->numChildren >= 256 || pid >= 256 || pid < 0) {
        return;
    }

    this->children[pid] = pid;
    this->numChildren++;
}

void Process::RemoveChild(uint32_t pid)
{
    if (this->numChildren >= 256 || pid >= 256 || pid < 0) {
        return;
    }

    this->children[pid] = 0;
    if (this->numChildren > 0)
        this->numChildren--;
}
```

System Calls

```c
// write system call
void sysprintf(char* str)
{
    asm("int $0x80" : : "a" (4), "b" (str));
}

// fork() system call
void fork(int *pid)
{
    asm("int $0x80" : "=c" (*pid) : "a" (2));
}

// fork_exec() system call
int fork_exec(void entrypoint())
{
    int ret;
    asm("int $0x80" : "=a" (ret) : "a" (15), "b" ((uint32_t)entrypoint));
    return ret;
}

// execve() system call
void sysexecve(void entry())
{
    asm("int $0x80" : : "a" (11), "b" (entry));
}
```

execve sends eip on ebx register. First fork uses ecx to store pid. fork_exec needs an entry point to change core image.

```
// execve() system call
void sysexecve(void entry())
{
    asm("int $0x80" : : "a" (11), "b" (entry));
}

// getpid() system call, no interrupt for this one
int sysgetpid()
{
    int ret;

    enterCritical();
    ret = taskManager.GetCurrentPid();
    exitCritical();

    return ret;
}

// waitpid() system call
void syswaitpid(uint32_t pid)
{
    asm("int $0x80" : : "a" (7), "b" (pid));
}

// exit() system call
void sysexit(int* ret)
{
    taskManager.TerminateProcess(ret);
}
```

```
// sysread() system call
void sysread(char* str, uint32_t size)
{
    keyboard.SetReadBytes(size);
    readIOFlag = true;

    // wait for keyboard interrupt to finish
    while(readIOFlag);

    // copy the string from keyboard buffer to str
    for (int i = 0; i < size; i++)
        str[i] = keyboard.ReadBuffer(i);

    str[size] = '\0';
    keyboard.ResetBuffer();

    readIOFlag = false;
}
```

- Read sets flag when it enters critical I/O region, and it makes busy waiting till keyboard interrupt finishes.

Syscall Handler

```
switch(cpu->eax)
{
    case 2:
        cpu->ecx = fork(cpu);
        return InterruptHandler::HandleInterrupt((uint32_t) cpu);
        break;

    // waitpid()
    case 7:

        wait(cpu);
        return InterruptHandler::HandleInterrupt((uint32_t) cpu);
        break;

    // execve()
    case 11:
        // implement execve
        execve(cpu);
        cpu->eax = 0;
        cpu->eip = cpu->ebx;
        esp = (uint32_t)cpu;
        break;
    case 4:
        printf((char*)cpu->ebx);
        break;

    // fork_exec()
    case 15:
        cpu->eax = fork2(cpu);

        return InterruptHandler::HandleInterrupt((uint32_t) cpu);
        break;
```

Set the eip for execve, cpu should be sent to the implementations for execve and fork, because I update those processes registers also. After fork, exec, and wait rescheduling is done.

Timer Interrupt Handling

```
if(interrupt == hardwareInterruptOffset)
{
    interruptCounter++;
    if(interruptCounter >= MAX_INTERRUPT_NUMBER && !readIOFlag && !criticalFlag)
    {
        printf("\nTimer interrupt! Switching\n");
        interruptCounter = 0;
        esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
    }
}
```

- Checking I/O flag, critical flag, and it max interrupt number. Then scheduling.

Keyboard Interrupt

```
if (readIOFlag)
{
    // write to the keyboard buffer
    if (keyPress)
    {
        if (keyBufferIndex < 256 || readKey != '\n' || keyBufferIndex < bytesToRead)
            keyBuffer[keyBufferIndex++] = readKey;

        if (keyBufferIndex >= 256 || keyBufferIndex >= bytesToRead || readKey == '\n')
        {
            keyBufferIndex = 0;
            bytesToRead = 0;
            readIOFlag = false;
            keyPress = false;
        }
    }
}
```

- Write to buffer during interrupt, then copy that buffer to read input.

```
if (keyPress && !readIOFlag && (readKey != '\n')) {
    printf("\nKey pressed! switching\n");

    esp = (uint32_t) interruptManager->ScheduleTransmitter((CPUState*) esp);
}
```

- I make switching using a transmitter function in interrupt manager. Don't make interrupt for newline character.

Mouse Interrupt

```
// if button is clicked
if (buffer[0] & 0x1)
{
    if (!readIOFlag)
        printf("\nLeft Click! switching\n");
    clickFlag = true;
}
```

```
if (clickFlag && !readIOFlag)
    esp = (uint32_t) interruptManager->ScheduleTransmitter((CPUState*)esp);
```

- Check if left click is happened, then make interrupt.

ExecProcess Function

```
/*
* exec system call with function pointer
* gets the entry point from ebx register
*/
void TaskManager::ExecProcess(CPUState* cpustate)
{
    tasks[currentTask].cpustate = cpustate;
    tasks[currentTask].cpustate -> eip = cpustate -> ebx;

    // set process state to ready
    Process* process = &(processTable.processes[tasks[currentTask].pid]);
    process->SetState(READY);
}
```

ForkProcess Function

```
common::uint32_t TaskManager::ForkProcess(CPUState* cpustate)
{
    // if memory is full, return
    if (numTasks >= 256)
        return 0;

    for (int i = 0; i < sizeof(tasks[currentTask].stack); i++)
    {
        tasks[numTasks].stack[i] = tasks[currentTask].stack[i];
    }
    tasks[numTasks].cpustate = (CPUState*)(tasks[numTasks].stack + 4096 - sizeof(CPUState));
    common::uint32_t currentTaskOffset = ((common::uint32_t)tasks[currentTask].cpustate) - ((common::uint32_t)cpustate);
    tasks[numTasks].cpustate = (CPUState*)(((common::uint32_t)tasks[numTasks].cpustate) - currentTaskOffset);
```

```
// assign minimum possible pid
for (int i = tasks[currentTask].pid + 1; i < 256; i++)
{
    if (processTable.processes[i].GetPid() == 0)
    {
        tasks[numTasks].pid = i;
        break;
    }
}

tasks[numTasks].cpustate -> ecx = 0;

// add child to parent
Process* parent = &(processTable.processes[tasks[currentTask].pid]);
parent->AddChild(tasks[numTasks].pid);

// add parent to child
Process* child = &(processTable.processes[tasks[numTasks].pid]);
child->SetPpid(tasks[currentTask].pid);

// add process to process table
Process* process = &(processTable.processes[tasks[numTasks].pid]);
process->SetPid(tasks[numTasks].pid);
process->SetPpid(tasks[currentTask].pid);
process->SetState(READY);
```

```
numTasks++;

// return child pid
return tasks[numTasks-1].pid;
```

- I get the offset calculation from shared hw3, other implementations are same as before.

ForkProcess2 Function

```
common::uint32_t TaskManager::ForkProcess2(CPUState* cpustate)
{
    // if memory is full, return
    if (numTasks >= 256)
        return 0;

    tasks[numTasks].cpustate->eip = cpustate->ebx;

    // assign minimum possible pid
    for (int i = tasks[currentTask].pid + 1; i < 256; i++)
    {
        if (processTable.processes[i].GetPid() == 0)
        {
            tasks[numTasks].pid = i;
            break;
        }
    }
```

```
// add child to parent
Process* parent = &(processTable.processes[tasks[currentTask].pid]);
parent->AddChild(tasks[numTasks].pid);

// add parent to child
Process* child = &(processTable.processes[tasks[numTasks].pid]);
child->SetPpid(tasks[currentTask].pid);

// add process to process table
Process* process = &(processTable.processes[tasks[numTasks].pid]);
process->SetPid(tasks[numTasks].pid);
process->SetPpid(tasks[currentTask].pid);
process->SetState(READY);

numTasks++;

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

- It sets eip to ebx, and rest is same with fork.

WaitProcess Function

```
void TaskManager::WaitProcess(CPUState* cpustate)
{
    common::uint32_t pid = cpustate->ebx;

    // self waiting is not allowed
    if (tasks[currentTask].pid == cpustate->ebx || pid == 0 || pid == 1)
        return;

    // cannot wait for a process that is not a child
    if (!processTable.processes[tasks[currentTask].pid].IsChild(pid))
        return;

    Process* process = &(processTable.processes[tasks[currentTask].pid]);
    process->SetState(BLOCKED);
    process->waitPid = pid;
}
```

- Changing process state to blocked, and adding the waited child to the table.

Critical Region

```
void enterCritical()
{
    // interrupts.Deactivate();
    criticalFlag = true;
}

void exitCritical()
{
    // interrupts.Activate();
    criticalFlag = false;
}
```

For handling race conditions, I used disabling and enabling interrupts solution, but I changed it to using flags. It only prevents timer interrupts for certain sections.

initProcess Function

Microkernel1

```
// first strategy, load 3 processes
#ifdef MICROKERNEL1
    void (*process[])() = {binarySearch, linearSearch, collatz};

    fork_exec(process[0]);
    fork_exec(process[1]);
    fork_exec(process[2]);
#endif
```

- Init process forks, three process.

Microkernel2

```
// second strategy, pick one process randomly, and load it 10 times
#ifdef MICROKERNEL2
    void (*process[])() = {binarySearch, linearSearch, collatz};

    uint32_t time;

    // get the time
    asm volatile("rdtsc" : "=a" (time) : : "edx");

    if (time < 0)
        time = -time;

    int index = time % 3;

    for (int i = 0; i < 10; ++i)
        fork_exec(process[index]);
#endif
```

choose randomly with rdtsc instruction as a seed. Then set the entry points of each empty task, then load them using fork.

Microkernel3

```
// third strategy, pick 2 out of 3 randomly, and load each one 3 times
#ifdef MICROKERNEL3

    void (*process[])() = {binarySearch, linearSearch, collatz};

    uint32_t time;

    // get the time
    asm volatile("rdtsc" : "=a" (time) : : "edx");

    if (time < 0)
        time = -time;

    int index1 = time % 3;
    int index2 = (time + 1) % 3;

    for (int i = 0; i < 3; ++i) {
        fork_exec(process[index1]);
        fork_exec(process[index2]);
    }
#endif
```

This is similar to the microkernel2, it chooses randomly then loads this time 2 different process each one is 3 times.

Tests

```
#ifdef FORK_TEST
    fork_exec(&taskFork);
#endif

#ifdef EXECVE_TEST
    fork_exec(&taskExecve);
#endif

#ifdef GETPID_TEST
    fork_exec(&taskA);
    fork_exec(&taskB);
#endif

#ifdef MULTIPROGRAMMING_TEST
    fork_exec(&taskA);
    fork_exec(&taskB);
    fork_exec(&taskC);
#endif

#ifdef WAITPID_TEST
    int pid = fork_exec(&taskWait);
    syswaitpid(pid);
#endif
```

Other test macros inside init process.

5- Test Cases and Results

Macros

kernel.cpp file

```
#define MICROKERNEL1
// #define MICROKERNEL2
// #define MICROKERNEL3

// #define MULTIPROGRAMMING_TEST
// #define FORK_TEST
// #define EXECVE_TEST
// #define GETPID_TEST
// #define WAITPID_TEST

#define PRINT_MODE
```

multitasking.cpp file

```
#define SWITCH_PRINT_MODE
#define PT_PRINT_MODE
```

- MICROKERNEL1 -> tests first strategy
- MICROKERNEL2 -> tests second strategy
- MICROKERNEL3 -> tests third strategy
- MULTIPROGRAMMING_TEST -> tests scheduling, and round robin
- FORK_TEST -> tests fork system call (incomplete)
- EXECVE_TEST -> tests execve system call
- GETPID_TEST -> tests getpid system call
- WAITPID_TEST -> tests waitpid system call
- PRINT_MODE -> Prints every outputs in processes
- SWITCH_PRINT_MODE -> Prints every context switch.
- PT_PRINT_MODE -> Prints process table in every context switch

Comment out for each part you want to test. Aside from printings, other macros should be commented out one by one, they must be tested separately.

Multiprogramming Test

CASE:

kernel.cpp

```
// #define MICROKERNEL1
// #define MICROKERNEL2
// #define MICROKERNEL3

#define MULTIPROGRAMMING_TEST
// #define FORK_TEST
// #define EXECVE_TEST
// #define GETPID_TEST
// #define WAITPID_TEST

// #define PRINT_MODE
```

multitasking.cpp

```
#define SWITCH_PRINT_MODE
#define PT_PRINT_MODE
```

```c
void taskA()
{
    int pid;
    pid = sysgetpid();
    printf("TASKA\n");
    printf("PID: ");
    printfHex32(pid);
    printf("\n");

    while(true);
}
```

```c
void taskB()
{
    int pid;
    pid = sysgetpid();
    printf("TASKB\n");
    printf("PID: ");
    printfHex32(pid);
    printf("\n");

    while(true);
}
```

```c
void taskC()
{
    int pid;
    pid = sysgetpid();
    printf("TASKC\n");
    printf("PID: ");
    printfHex32(pid);
    printf("\n");

    while(true);
}
```

These tasks are loaded, it is expected that they are continuously scheduled by every 1/256 timer interrupt, or mouse, keyboard interrupts, they are non-terminating processes.

RESULT:

```
hello

Key pressed! switching

Process table:

Total tasks: 1
PID: 1 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0

Init process started
```

- Switch with keyboard press

```
Timer interrupt! Switching
Switching task from 1 to 2

Process table:

Total tasks: 4
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 3
PID: 2 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 3 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 4 PPID: 1 State: READY Return Value: 0 Num Children: 0

TASKA
PID: 00000002
```

```
Timer interrupt! Switching
Switching task from 2 to 3

Process table:

Total tasks: 4
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 3
PID: 2 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 3 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 4 PPID: 1 State: READY Return Value: 0 Num Children: 0

TASKB
PID: 00000003
```

- Switch with timer interrupt

```
Left Click! switching
Switching task from 3 to 4

Process table:

Total tasks: 4
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 3
PID: 2 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 3 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 4 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
```

```
Left Click! switching
Switching task from 4 to 1

Process table:

Total tasks: 4
PID: 1 PPID: 1 State: RUNNING Return Value: 0 Num Children: 3
PID: 2 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 3 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 4 PPID: 1 State: READY Return Value: 0 Num Children: 0
```

- Switch with left mouse click
- Since they are non-terminating processes, they make continuously round robin. After they are printing their id, it makes switching on every timer interrupt.

Strategy 1 – Microkernel1 Test

CASE:

```
#define MICROKERNEL1
// #define MICROKERNEL2
// #define MICROKERNEL3

// #define MULTIPROGRAMMING_TEST
// #define FORK_TEST
// #define EXECVE_TEST
// #define GETPID_TEST
// #define WAITPID_TEST

#define PRINT_MODE
```

```
#define SWITCH_PRINT_MODE
#define PT_PRINT_MODE
```

RESULT:

```
hello

Timer interrupt! Switching

Process table:

Total tasks: 1
PID: 1 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0

Init process started

Key pressed! switching
Switching task from 1 to 2

Process table:

Total tasks: 4
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 3
PID: 2 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 3 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 4 PPID: 1 State: READY Return Value: 0 Num Children: 0


PID: 2: enter array size (max 15): 4
```

```
PID: 2: index 0: 2

PID: 2: index 1: 4

PID: 2: index 2: 6

PID: 2: index 3: 8

enter target: 6
ID: 00000002 OUTPUT: 2

Timer interrupt! Switching
Switching task from 2 to 3

Process table:

Total tasks: 3
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 3
PID: 2 PPID: 1 State: TERMINATED Return Value: 2 Num Children: 0
PID: 3 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 4 PPID: 1 State: READY Return Value: 0 Num Children: 0
```

```
PID: 3: enter array size (max 15): 10
```

```
PID: 3: index 0: 10

PID: 3: index 1: 12

PID: 3: index 2: 21

PID: 3: index 3: 33

PID: 3: index 4: 45

PID: 3: index 5: 56

PID: 3: index 6: 63

PID: 3: index 7: 72

PID: 3: index 8: 84

PID: 3: index 9: 95

enter target: 45
```

```
Process table:

Total tasks: 2
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 3
PID: 2 PPID: 1 State: TERMINATED Return Value: 2 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 4 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
```

```
PID: 4: enter number of collatz sequences(max 30): 7
```

```
PID: 4: enter number of collatz sequences(max 30): 7
1:
2: 1
3: 10  5  16  8  4  2  1
4: 2  1
5: 16  8  4  2  1
6: 3  10  5  16  8  4  2  1
7: 22  11  34  17  52  26  13  40  20  10  5  16  8  4  2  1
```

```
Key pressed! switching

Process table:

Total tasks: 1
PID: 1 PPID: 1 State: RUNNING Return Value: 0 Num Children: 3
PID: 2 PPID: 1 State: TERMINATED Return Value: 2 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 4 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
```

- Arguments of all 3 processes are taken one by one, sometimes timer interrupt occurred, sometimes I pressed keyboard and perform switch. All 3 processes work correct, then they terminated.

Strategy 2 – Microkernel2 Test

CASE:

```
// #define MICROKERNEL1
#define MICROKERNEL2
// #define MICROKERNEL3

// #define MULTIPROGRAMMING_TEST
// #define FORK_TEST
// #define EXECVE_TEST
// #define GETPID_TEST
// #define WAITPID_TEST

#define PRINT_MODE
```

```
#define SWITCH_PRINT_MODE
#define PT_PRINT_MODE
```

RESULT:

```
hello

Key pressed! switching

Process table:

Total tasks: 1
PID: 1 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0

Init process started
```

```
PID: 9 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 10 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 11 PPID: 1 State: READY Return Value: 0 Num Children: 0
```

(after interrupt, screen is filled, this is the last 3 element of the table)

```
PID: 2: enter array size (max 15): 6

PID: 2: index 0: 10

PID: 2: index 1: 14

PID: 2: index 2: 16

PID: 2: index 3: 25

PID: 2: index 4: 31

PID: 2: index 5: 42

enter target: 31
```

```
Timer interrupt! Switching
Switching task from 2 to 3

Process table:

Total tasks: 10
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 10
PID: 2 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 3 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 4 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 5 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 6 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 7 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 8 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 9 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 10 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 11 PPID: 1 State: READY Return Value: 0 Num Children: 0
```

```
PID: 3: enter array size (max 15): 10
```

```
PID: 3: index 2: 187

PID: 3: index 3: 762

PID: 3: index 4: 800

PID: 3: index 5: 911

PID: 3: index 6: 956

PID: 3: index 7: 975

PID: 3: index 8: 989

PID: 3: index 9: 1000

enter target: 957
```

(it should give -1)

```
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 10
PID: 2 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 4 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 5 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 6 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 7 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 8 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 9 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 10 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 11 PPID: 1 State: READY Return Value: 0 Num Children: 0
```

```
Total tasks: 9
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 10
PID: 2 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 4 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 5 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 6 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 7 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 8 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 9 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 10 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 11 PPID: 1 State: READY Return Value: 0 Num Children: 0


PID: 5: enter array size (max 15):
```

(timer happened during p4)

```
PID: 5: enter array size (max 15): 8

PID: 5: index 0: 2

PID: 5: index 1: 4

PID: 5: index 2: 15

PID: 5: index 3: 20

PID: 5: index 4: 36
```

```
PID: 5: index 5: 74

PID: 5: index 6: 81

PID: 5: index 7: 92

enter target: 36
```

(screen is filled result will be seen on table, it should be 4)

```
PID: 6: enter array size (max 15): 5

PID: 6: index 0: 24

PID: 6: index 1: 36

PID: 6: index 2: 42

PID: 6: index 3: 55

PID: 6: index 4: 61

enter target: 79
```

(result should be -1)

```
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 10
PID: 2 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 4 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 5 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 6 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 7 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 8 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 9 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 10 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 11 PPID: 1 State: READY Return Value: 0 Num Children: 0


PID: 7: enter array size (max 15):
```

(results are correct)

```
PID: 7: enter array size (max 15): 4

PID: 7: index 0: 12

PID: 7: index 1: 15

PID: 7: index 2: 24

PID: 7: index 3: 42

enter target: 12
```

```
Total tasks: 7
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 10
PID: 2 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 4 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 5 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 6 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 7 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 8 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 9 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 10 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 11 PPID: 1 State: READY Return Value: 0 Num Children: 0
```

(7 task is active)

```
PID: 8: enter array size (max 15): 20

invalid input!
enter array size (max 15): 10
```

(error handling)

```
PID: 8: index 3: 30

PID: 8: index 4: 35

PID: 8: index 5: 40

PID: 8: index 6: 45

PID: 8: index 7: 67

PID: 8: index 8: 72

PID: 8: index 9: 100

enter target: 100
```

```
PID: 2 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 4 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 5 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 6 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 7 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 8 PPID: 1 State: TERMINATED Return Value: 9 Num Children: 0
PID: 9 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 10 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 11 PPID: 1 State: READY Return Value: 0 Num Children: 0
```

```
PID: 3 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 4 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 5 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 6 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 7 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 8 PPID: 1 State: TERMINATED Return Value: 9 Num Children: 0
PID: 9 PPID: 1 State: TERMINATED Return Value: 1 Num Children: 0
PID: 10 PPID: 1 State: TERMINATED Return Value: 3 Num Children: 0
PID: 11 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
```

(other processes are terminated similarly)

```
PID: 4: index 3: 110

PID: 4: index 4: 130

PID: 4: index 5: 160

PID: 4: index 6: 180

PID: 4: index 7: 200

enter target: 180
```

(returning to the process 4)

```
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 10
PID: 2 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 4 PPID: 1 State: TERMINATED Return Value: 6 Num Children: 0
PID: 5 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 6 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 7 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 8 PPID: 1 State: TERMINATED Return Value: 9 Num Children: 0
PID: 9 PPID: 1 State: TERMINATED Return Value: 1 Num Children: 0
PID: 10 PPID: 1 State: TERMINATED Return Value: 3 Num Children: 0
PID: 11 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
```

(process 4 returned correctly)

```
Key pressed! switching

Process table:

Total tasks: 1
PID: 1 PPID: 1 State: RUNNING Return Value: 0 Num Children: 10
PID: 2 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 4 PPID: 1 State: TERMINATED Return Value: 6 Num Children: 0
PID: 5 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 6 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 7 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 8 PPID: 1 State: TERMINATED Return Value: 9 Num Children: 0
PID: 9 PPID: 1 State: TERMINATED Return Value: 1 Num Children: 0
PID: 10 PPID: 1 State: TERMINATED Return Value: 3 Num Children: 0
PID: 11 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
```

(last version of table)

- One of the binary or linear search is chosen in this case randomly, this is the case that same process is loaded 10 times using fork. All of them gave correct results, also a timer interrupt happened during process 4 execution, it successfully returned when turn is on that process and completed process 4.

Another Run

```
PID: 9 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 10 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 11 PPID: 1 State: READY Return Value: 0 Num Children: 0


PID: 2: enter number of collatz sequences(max 30):
```

```
PID: 7: enter number of collatz sequences(max 30): 8
1:
2: 1
3: 10  5  16  8  4  2  1
4: 2  1
5: 16  8  4  2  1
6: 3  10  5  16  8  4  2  1
7: 22  11  34  17  52  26  13  40  20  10  5  16  8  4  2  1
8: 4  2  1
```

(7th process)

```
7: 22  11  34  17  52  26  13  40  20  10  5  16  8  4  2  1
8: 4  2  1
9: 28  14  7  22  11  34  17  52  26  13  40  20  10  5  16  8  4  2  1
10: 5  16  8  4  2  1
11: 34  17  52  26  13  40  20  10  5  16  8  4  2  1
12: 6  3  10  5  16  8  4  2  1
13: 40  20  10  5  16  8  4  2  1
14: 7  22  11  34  17  52  26  13  40  20  10  5  16  8  4  2  1
15: 46  23  70  35  106  53  160  80  40  20  10  5  16  8  4  2  1
16: 8  4  2  1
17: 52  26  13  40  20  10  5  16  8  4  2  1
18: 9  28  14  7  22  11  34  17  52  26  13  40  20  10  5  16  8  4  2  1
19: 58  29  88  44  22  11  34  17  52  26  13  40  20  10  5  16  8  4  2  1
20: 10  5  16  8  4  2  1
21: 64  32  16  8  4  2  1
22: 11  34  17  52  26  13  40  20  10  5  16  8  4  2  1
23: 70  35  106  53  160  80  40  20  10  5  16  8  4  2  1
24: 12  6  3  10  5  16  8  4  2  1
25: 76  38  19  58  29  88  44  22  11  34  17  52  26  13  40  20  10  5  16  8
  4  2  1
```

(another process when input is 25)

```
Total tasks: 5
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 10
PID: 2 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 4 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 5 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 6 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 7 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 8 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 9 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 10 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 11 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0


PID: 11: enter number of collatz sequences(max 30): 3
1:
2: 1
3: 10  5  16  8  4  2  1
```

(still 5 left)

```
PID: 9 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 10 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 11 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0

   17  52  26  13  40  20  10  5  16  8  4  2  1
12: 6  3  10  5  16  8  4  2  1
13: 40  20  10  5  16  8  4  2  1
14: 7  22  11  34  17  52  26  13  40  20  10  5  16  8  4  2  1
15: 46  23  70  35  106  53  160  80  40  20  10  5  16  8  4  2  1
16: 8  4  2  1
17: 52  26  13  40  20  10  5  16  8  4  2  1
18: 9  28  14  7  22  11  34  17  52  26  13  40  20  10  5  16  8  4  2  1
19: 58  29  88  44  22  11  34  17  52  26  13  40  20  10  5  16  8  4  2  1
20: 10  5  16  8  4  2  1
21: 64  32  16  8  4  2  1
22: 11  34  17  52  26  13  40  20  10  5  16  8  4  2  1
23: 70  35  106  53  160  80  40  20  10  5  16  8  4  2  1
24: 12  6  3  10  5  16  8  4  2  1
25: 76  38  19  58  29  88  44  22  11  34  17  52  26  13  40  20  10  5  16  8
  4  2  1
```

(when they have their turn, they are completing their printing job which is interrupted by timer)

```
Key pressed! switching

Process table:

Total tasks: 1
PID: 1 PPID: 1 State: RUNNING Return Value: 0 Num Children: 10
PID: 2 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 4 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 5 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 6 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 7 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 8 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 9 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 10 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 11 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
```

(completed)

Strategy 3 – Microkernel3 Test

CASE:

```
// #define MICROKERNEL1
// #define MICROKERNEL2
#define MICROKERNEL3

// #define MULTIPROGRAMMING_TEST
// #define FORK_TEST
// #define EXECVE_TEST
// #define GETPID_TEST
// #define WAITPID_TEST

#define PRINT_MODE
```

```
#define SWITCH_PRINT_MODE
#define PT_PRINT_MODE
```

RESULT:

```
hello

Key pressed! switching

Process table:

Total tasks: 1
PID: 1 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0

Init process started
```
(init started)

```
Total tasks: 7
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 6
PID: 2 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 3 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 4 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 5 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 6 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 7 PPID: 1 State: READY Return Value: 0 Num Children: 0
```
(init loads processes)

```
PID: 2: ls - enter array size (max 15): 6

PID: 2: index 0: 4

PID: 2: index 1: 2

PID: 2: index 2: 15

PID: 2: index 3: 23

PID: 2: index 4: 31

PID: 2: index 5: 26

enter target: 23
```
(linear search)

```
PID: 2 PPID: 1 State: TERMINATED Return Value: 3 Num Children: 0
PID: 3 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 4 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 5 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 6 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 7 PPID: 1 State: READY Return Value: 0 Num Children: 0
```
(result is correct)

```
PID: 3: enter number of collatz sequences(max 30): 6
1:
2: 1
3: 10  5  16  8  4  2  1
4: 2  1
5: 16  8  4  2  1
6: 3  10  5  16  8  4  2  1
```
(other chosen process is collatz)

```
PID: 4: ls - enter array size (max 15): 10
PID: 4: index 0: 4
PID: 4: index 1: 12
PID: 4: index 2: 24
PID: 4: index 3: 5
PID: 4: index 4: 8
PID: 4: index 5: 61
PID: 4: index 6: 27
PID: 4: index 7: 32
PID: 4: index 8: 45
```

```
PID: 4: index 9: 41

enter target: 5
```
(output should be 3)

```
enter target: 5
ID: 00000004 OUTPUT: 3

Timer interrupt! Switching
Switching task from 4 to 5

Process table:

Total tasks: 4
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 6
PID: 2 PPID: 1 State: TERMINATED Return Value: 3 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 4 PPID: 1 State: TERMINATED Return Value: 3 Num Children: 0
PID: 5 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 6 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 7 PPID: 1 State: READY Return Value: 0 Num Children: 0
```

(it is correct, and timer interrupt occurred)

```
PID: 5: enter number of collatz sequences(max 30): 25
```

(it printed but screen is filled)

```
Total tasks: 2
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 6
PID: 2 PPID: 1 State: TERMINATED Return Value: 3 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 4 PPID: 1 State: TERMINATED Return Value: 3 Num Children: 0
PID: 5 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 6 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 7 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0


PID: 7: enter number of collatz sequences(max 30):
```

(other process is also computed, last process is 7)

```
PID: 7: enter number of collatz sequences(max 30): 7
```

```
Key pressed! switching

Process table:

Total tasks: 1
PID: 1 PPID: 1 State: RUNNING Return Value: 0 Num Children: 6
PID: 2 PPID: 1 State: TERMINATED Return Value: 3 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 4 PPID: 1 State: TERMINATED Return Value: 3 Num Children: 0
PID: 5 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 6 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 7 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
```

(all processes are terminated correctly, collatz returns 0)

Another Run

```
PID: 2: bs - enter array size (max 15): 5
PID: 2: index 0: 10
PID: 2: index 1: 15
PID: 2: index 2: 22
PID: 2: index 3: 34
PID: 2: index 4: 45
enter target: 10
```

(binary search)

```
PID: 3: ls - enter array size (max 15): 5

PID: 3: index 0: 17

PID: 3: index 1: 28

PID: 3: index 2: 333

PID: 3: index 3: 2

PID: 3: index 4: 4

enter target: 4
```

(linear search)

```
Total tasks: 5
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 6
PID: 2 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 4 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 5 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 6 PPID: 1 State: READY Return Value: 0 Num Children: 0
PID: 7 PPID: 1 State: READY Return Value: 0 Num Children: 0
```

(their results are correct)

```
PID: 4: bs - enter array size (max 15): 10

PID: 4: index 0: 20

PID: 4: index 1: 35

PID: 4: index 2: 47

PID: 4: index 3: 63

PID: 4: index 4: 81

PID: 4: index 5: 7622
```

```
PID: 4: index 6: 8000

PID: 4: index 7: 8057

PID: 4: index 8: 9063

PID: 4: index 9: 9901

enter target: 8000
```

```
PID: 5: ls - enter array size (max 15): 10

PID: 5: index 0: 20

PID: 5: index 1: 31

PID: 5: index 2: 84

PID: 5: index 3: 72

PID: 5: index 4: 11

PID: 5: index 5: 61

PID: 5: index 6: 62

PID: 5: index 7: 89

PID: 5: index 8: 93

PID: 5: index 9: 75

enter target: 11
```

(linear)

```
Timer interrupt! Switching
Switching task from 5 to 6

Process table:

Total tasks: 3
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 6
PID: 2 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 4 PPID: 1 State: TERMINATED Return Value: 6 Num Children: 0
PID: 5 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 6 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
PID: 7 PPID: 1 State: READY Return Value: 0 Num Children: 0
```
(results are correct)

```
PID: 6: bs - enter array size (max 15): 3

PID: 6: index 0: 10

PID: 6: index 1: 20

PID: 6: index 2: 30

enter target: 50
```
(result must be -1)

```
ID: 00000006 OUTPUT: -1

Timer interrupt! Switching
Switching task from 6 to 7

Process table:

Total tasks: 2
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 6
PID: 2 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 4 PPID: 1 State: TERMINATED Return Value: 6 Num Children: 0
PID: 5 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 6 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 7 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0
```

```
PID: 7: ls - enter array size (max 15): 3

PID: 7: index 0: 20

PID: 7: index 1: 30

PID: 7: index 2: 60
```

```
enter target: 70
```

```
enter target: 70
ID: 00000007 OUTPUT: -1
```

```
Key pressed! switching

Process table:

Total tasks: 1
PID: 1 PPID: 1 State: RUNNING Return Value: 0 Num Children: 6
PID: 2 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 0
PID: 3 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 4 PPID: 1 State: TERMINATED Return Value: 6 Num Children: 0
PID: 5 PPID: 1 State: TERMINATED Return Value: 4 Num Children: 0
PID: 6 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
PID: 7 PPID: 1 State: TERMINATED Return Value: -1 Num Children: 0
```

- In this test, it is ran 2 times, in the first one collatz, and linear search is generated, in the second one linear search and binary search are generated. They both work correctly. Sometimes interrupt can happen during processes, and it may not print appropriate but in table results will be correct.

Fork & EXECVE Test

- In this test, successful fork, and exec after fork is indicated.

CASE:

```
// #define MICROKERNEL1
// #define MICROKERNEL2
// #define MICROKERNEL3

// #define MULTIPROGRAMMING_TEST
#define FORK_TEST
// #define EXECVE_TEST
// #define GETPID_TEST
// #define WAITPID_TEST

// #define PRINT_MODE
```

```
#define SWITCH_PRINT_MODE
#define PT_PRINT_MODE
```

```
void taskFork()
{
    int pid = 0;
    fork(&pid);

    if(pid == 0) {
            printf("\nID: ");
            printfHex32(pid);
            printf(" CHILD\n");
            sysexecve(&taskExecve);
    }
    else {
            printf("\nID: ");
            printfHex32(taskManager.GetCurrentPid());
            printf(" PARENT\n");
    }

    sysexit(&pid);
    // while(true);
}
```

```
void execHere()
{
    int ret = 0;
    printf("success!\n");

    sysexit(&ret);
}
void taskExecve()
{
    printf("old\n");
    sysexecve(&execHere);
    printf("here\n");


    while(true);
}
```

- It simply makes fork, then execve. It must print child and parent, then change child to another execution.

RESULT:

```
hello

Key pressed! switching

Process table:

Total tasks: 1
PID: 1 PPID: 1 State: RUNNING Return Value: 0 Num Children: 0

Init process started
```

```
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 1
PID: 2 PPID: 1 State: READY Return Value: 0 Num Children: 1
PID: 3 PPID: 2 State: RUNNING Return Value: 0 Num Children: 0


ID: 00000000 CHILD
execve() called
old
execve() called
success!
```

(enter the child, and successfully executes execve 2 times as expected)

```
Key pressed! switching
Switching task from 3 to 1

Process table:

Total tasks: 2
PID: 1 PPID: 1 State: RUNNING Return Value: 0 Num Children: 1
PID: 2 PPID: 1 State: READY Return Value: 0 Num Children: 1
PID: 3 PPID: 2 State: TERMINATED Return Value: 0 Num Children: 0
```

```
Process table:

Total tasks: 2
PID: 1 PPID: 1 State: READY Return Value: 0 Num Children: 1
PID: 2 PPID: 1 State: RUNNING Return Value: 0 Num Children: 1
PID: 3 PPID: 2 State: TERMINATED Return Value: 0 Num Children: 0


ID: 00000002 PARENT

Timer interrupt! Switching

Process table:

Total tasks: 1
PID: 1 PPID: 1 State: RUNNING Return Value: 0 Num Children: 1
PID: 2 PPID: 1 State: TERMINATED Return Value: 3 Num Children: 1
PID: 3 PPID: 2 State: TERMINATED Return Value: 0 Num Children: 0
```

(then enters, the parent, and all processes are terminated correctly)

Waitpid Test

CASE:

```c
// #define MICROKERNEL1
// #define MICROKERNEL2
// #define MICROKERNEL3

// #define MULTIPROGRAMMING_TEST
// #define FORK_TEST
// #define EXECVE_TEST
// #define GETPID_TEST
#define WAITPID_TEST

// #define PRINT_MODE
```

```c
// #define SWITCH_PRINT_MODE
#define PT_PRINT_MODE
```

```c
void taskD()
{
    int pid;

    pid = fork_exec(&execHere);
    syswaitpid(pid);

    int ret = 0;
    sysexit(&ret);
}

void taskWait()
{
    int pid;
    int ret = 0;
    pid = fork_exec(&taskD);
    syswaitpid(pid);

    sysexit(&ret);
}
```

```c
#ifdef WAITPID_TEST
    int pid = fork_exec(&taskWait);
    syswaitpid(pid);
#endif
```

```c
void execHere()
{
    int ret = 0;
    printf("success!\n");

    sysexit(&ret);
}
```

- Initially it is expected that 3 process will be blocked, and one by one when their wait is resulted, they terminate.

RESULT:

```
Process table:

Total tasks: 4
PID: 1 PPID: 1 State: BLOCKED Return Value: 0 Num Children: 1
PID: 2 PPID: 1 State: BLOCKED Return Value: 0 Num Children: 1
PID: 3 PPID: 2 State: BLOCKED Return Value: 0 Num Children: 1
PID: 4 PPID: 3 State: RUNNING Return Value: 0 Num Children: 0

success!
```

(with the execution of init, each forked process calls wait, this leads them to be blocked)

```
Key pressed! switching

Process table:

Total tasks: 3
PID: 1 PPID: 1 State: BLOCKED Return Value: 0 Num Children: 1
PID: 2 PPID: 1 State: BLOCKED Return Value: 0 Num Children: 1
PID: 3 PPID: 2 State: RUNNING Return Value: 0 Num Children: 1
PID: 4 PPID: 3 State: TERMINATED Return Value: 0 Num Children: 0
```

(4 is terminated, its parent is 3, therefore 3 will be run)

```
Total tasks: 2
PID: 1 PPID: 1 State: BLOCKED Return Value: 0 Num Children: 1
PID: 2 PPID: 1 State: RUNNING Return Value: 0 Num Children: 1
PID: 3 PPID: 2 State: TERMINATED Return Value: 0 Num Children: 1
PID: 4 PPID: 3 State: TERMINATED Return Value: 0 Num Children: 0
```

(3 is terminated its parent is 2 then 2 will be run)

```
Key pressed! switching

Process table:

Total tasks: 1
PID: 1 PPID: 1 State: RUNNING Return Value: 0 Num Children: 1
PID: 2 PPID: 1 State: TERMINATED Return Value: 0 Num Children: 1
PID: 3 PPID: 2 State: TERMINATED Return Value: 0 Num Children: 1
PID: 4 PPID: 3 State: TERMINATED Return Value: 0 Num Children: 0
```

(all terminated)