CSE 312

HW1

Burak Kocausta

1901042605

Content

1- Install, Compilation and Run

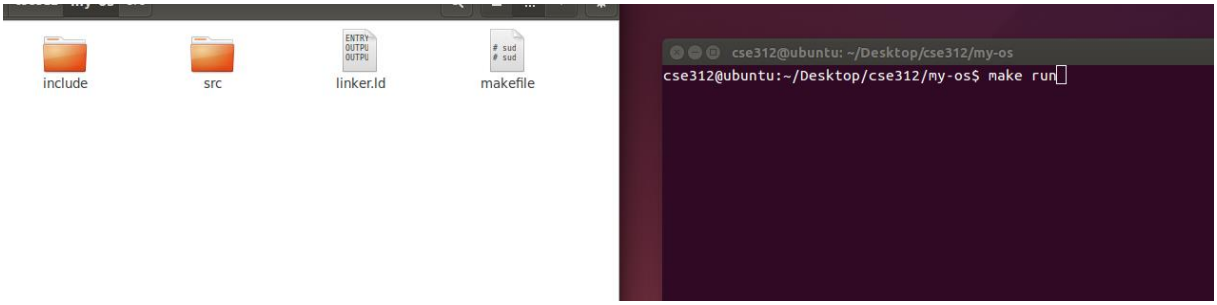
All parts are compiles and runs with ‘make run’ command. ISO image must be installed on virtualbox after it is created with `make mykernel.iso´ command. If it is tested on ubuntu environment, make run will compiles, then starts vm.

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

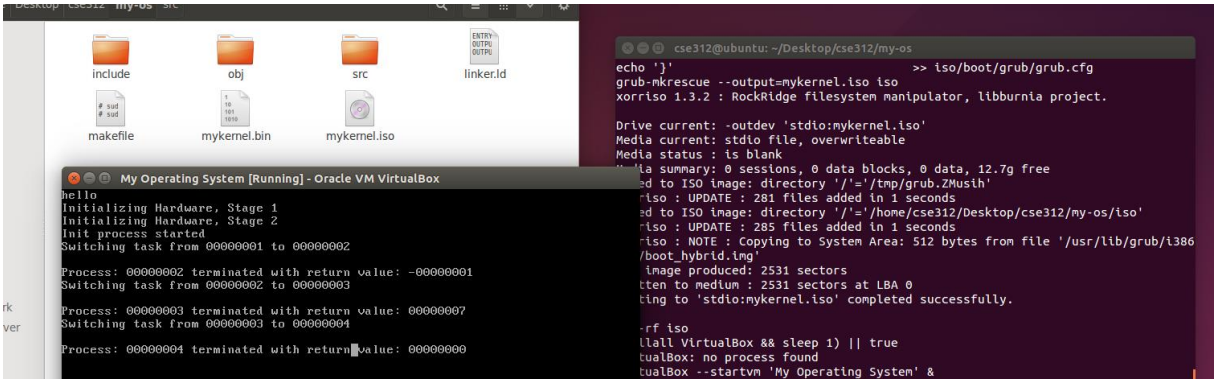
Install

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- General Design

It is asked us to design a multiprogramming structure, which uses robin round scheduling. I started developing on top of Viktor Engelmann's multitasking implementation. Firstly, I created a process class for holding process id, state, and other process information. State is an important addition, because Engelmann's implementation doesn't include process states, and processes do not terminate in his implementation. I defined process states as enum, they are ready, terminated, and blocked. I added return value field for this on processes. Each process holds its child processes ids, and parent process id. Then I created a process table structure, which simply holds process array, offset of each process is its id. With this approach, it is accessed constantly. I added process id to task class also. Then, I related the process, and task class in task manager. The Task Manager is responsible for adding tasks, updating process ids, terminating processes, and scheduling. The task manager holds a process table for accessing process information, and processes can be related with each other with that table. Since we cannot use dynamic memory, I changed the implementation of Engelmann's a bit. For dynamically adding tasks on runtime, I changed the pointer array of tasks to array of tasks. Then, I modified the addTask function, while adding tasks it sets the task entries with deep copy.

For scheduling, I developed Engelmann's round robin with states. I prevented switching if the process is terminated. To be running process must be ready state. It switches ready from running or running from ready. During termination it switches running from terminated.

I used Engelmann's system call file to implement posix system calls on top of it. I made my system calls functions with prefix "sys". They simply make an interrupt "0x80" and provide necessary registers to the interrupt handler. I made it with using inline assembly. Then the handler updates cpu registers and it might be invoking task manager. I don't use this interrupt mechanism for getpid because it simply returns the current tasks id. I implemented the execve a little bit differently because in our operating system there is no file system. I give an entry point to change memory image and discard old process. I spent most of my time with trying to implement fork, but I failed. And my explanations about fork implementation is on the "problems, incomplete parts" section.

For the implementation of binary search, linear search, and collatz. I implemented them as 3 different processes. If print mode is open, they print their result. Binary search, and linear search print result with just giving output result. For the collatz sequence, I researched it and it is a simple algorithm which works with just 2 condition for even and odd number. Purpose is reaching the 1. I printed all numbers sequence till 25. Also, I updated the result of binary search because, in the pdf it is incorrect. Array is not sorted, and output is -1. I searched 100 instead of 110, therefore output will be 6.

I created macros for loading 3 different flavors of microkernel. It is decided before compilation which one is loaded. I referred to the first strategy as microkernel1, second as microkernel2, third one microkernel3. All microkernels will start with loading the init process, init loads other processes, and init is parent of all processes. Init will not terminate and cannot be killed. Init loads processes according to the chosen microkernel.

For the first strategy, all 3 programs will simply loaded by init process. There will be four processes. Processes perform their action till they are terminated. After termination of all 3, only init process will run, and context switch ends.

For the second strategy, the same program will be loaded by init process 10 times. To implement this, I created dummy task objects, and after choosing which program is loaded, I set the tasks entry point to the chosen task. For randomly choosing I used "rdtsc" instruction to get time as seed.

In the third one, I again randomly chose and 6 process are loaded to the memory. I used a similar strategy with second strategy, after creating the objects, changed their instruction pointer to the process that I want to load.

I defined printing using macros, when os runs with switching involvement it can be confusing to track. With commenting out macros that I defined in multitasking.cpp, and kernel.cpp, process table in each context switch, switches from which pid to which, and every process output can be seen.

I also write additional functions for printing decimals, it simply prints in decimal format, I used it for printing collatz sequence.

3- 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;

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

    // return value
    int* retval;

    // process state
    ProcessState state;

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

```
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.

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

```
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

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

    // number of terminated tasks
    int terminatedTask;

    // process table
    ProcessTable processTable;
public:
    TaskManager();
    ~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
    void ForkProcess(GlobalDescriptorTable* gdt, CPUState* cpustate);
    void ExecProcess(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.

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

```
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);
    }
```

```
    // next task shouldn't be terminated
    do
    {
        // round robin till next task is not terminated
        if(++currentTask >= numTasks)
            currentTask %= numTasks;
    } while (processTable.processes[tasks[currentTask].pid].GetState() == TERMINATED);

    // print switching message
    #ifdef SWITCH_PRINT_MODE
    if (numTasks - terminatedTask > 1)
    {
        printf(" to ");
        printfHex32(tasks[currentTask].pid);
        printf("\n");
    }
    #endif

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

    // print current process table
    #ifdef PT_PRINT_MODE
        if (numTasks - terminatedTask > 1)
            PrintProcessTable();
    #endif

    return tasks[currentTask].cpustate;
}
```

If print modes are on switching, and process table information will be printed on each schedule if there are more than one process running.

Process Constructors

```
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

```
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

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

// fork() system call
int sysfork()
{
    int ret;
    asm("int $0x80" : "=a" (ret) : "a" (2));
    asm volatile("int $0x20");
    return ret;
}
```

eax value will be return value of sysfork().

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

int sysgetpid()
{
    int ret;

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

    return ret;
}
```

execve sends eip on ebx register.

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

Syscall Handler

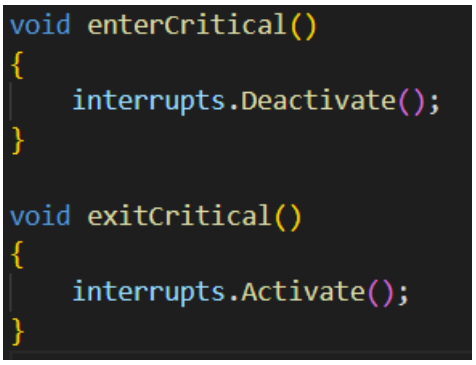
```
uint32_t SyscallHandler::HandleInterrupt(uint32_t esp)
{
    CPUState* cpu = (CPUState*)esp;

    switch(cpu->eax)
    {
        // fork()
        case 2:
            fork(cpu);
            //cpu->eax = 0;
            break;

        // waitpid()
        case 7:
            // implement waitpid

            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;

        default:
            break;
    }

    return esp;
}
```

Set the eip for execve, cpu should be sent to the implementations for execve and fork, because I update those processes registers also.

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);
}
```

Critical Region



For handling race conditions, I used disabling and enabling interrupts solution.

init Function

Microkernel1

```
void init()
{
    Task task_init(&gdt, &InitProcess);
    // first strategy, load 3 processes
    #ifdef MICROKERNEL1
        Task task_binarySearch(&gdt, &binarySearch);
        Task task_linearSearch(&gdt, &linearSearch);
        Task task_collatz(&gdt, &collatz);

        taskManager.AddTask(&task_init);
        taskManager.AddTask(&task_linearSearch);
        taskManager.AddTask(&task_binarySearch);
        taskManager.AddTask(&task_collatz);
    #endif
```

init process is added first, then other 3 process are added.

Microkernel2

```
// second strategy, pick one process randomly, and load it 10 times
#ifdef MICROKERNEL2
    // array of processes
    Task tasks[] = {Task(&gdt), Task(&gdt), Task(&gdt), Task(&gdt),
                    Task(&gdt), Task(&gdt), Task(&gdt), Task(&gdt),
                    Task(&gdt), Task(&gdt)};

    uint32_t time;

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

    if (time < 0)
        time = -time;

    // pick a random process
    uint32_t random = time % 3;

    // load the process 10 times
    for (int i = 0; i < 10; i++) {
        if (random == 0)
            tasks[i].setEntryPoint(&binarySearch);
        else if (random == 1)
            tasks[i].setEntryPoint(&linearSearch);
        else if (random == 2)
            tasks[i].setEntryPoint(&collatz);
    }

    // add the processes to the task manager
    taskManager.AddTask(&task_init);
    for (int i = 0; i < 10; i++) {
        taskManager.AddTask(&tasks[i]);
    }
#endif
```

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

Microkernel3

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

        // array of processes
        Task tasks[] = {Task(&gdt), Task(&gdt), Task(&gdt),
                        Task(&gdt), Task(&gdt), Task(&gdt)};

        uint32_t time;
        // get the time
        asm volatile("rdtsc" : "=a" (time) : : "edx");

        if (time < 0)
            time = -time;

        // pick 2 random processes
        uint32_t random1 = time % 3;
        uint32_t random2 = (random1 + 1) % 3;

        // load the processes 3 times
        for (int i = 0; i < 3; i++) {
            if (random1 == 0)
                tasks[i].setEntryPoint(&binarySearch);
            else if (random1 == 1)
                tasks[i].setEntryPoint(&linearSearch);
            else if (random1 == 2)
                tasks[i].setEntryPoint(&collatz);
        }
        for (int i = 3; i < 6; i++) {
            if (random2 == 0)
                tasks[i].setEntryPoint(&binarySearch);
            else if (random2 == 1)
                tasks[i].setEntryPoint(&linearSearch);
            else if (random2 == 2)
                tasks[i].setEntryPoint(&collatz);
        }

        // add processes through the task manager
        taskManager.AddTask(&task_init);
        for (int i = 0; i < 6; i++) {
            taskManager.AddTask(&tasks[i]);
        }
#endif
```

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

binarySearch Function

```
void binarySearch ()
{
    int arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170};
    int size = 10;
    int target = 100;

    int low = 0;
    int high = size - 1;
    int mid;
    int fail = -1;
    while(low <= high)
    {
        mid = (low + high) / 2;
        if (arr[mid] == target) {

            #ifdef PRINT_MODE
                printf("ID: ");
                printfHex32(taskManager.GetCurrentPid());
                printf(" OUTPUT: ");
                printfDec(mid);
                printf("\n");
            #endif
            sysexit(&mid);
        }

        else if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
```

```
    #ifdef PRINT_MODE
        printf("ID: ");
        printfHex32(taskManager.GetCurrentPid());
        printf(" OUTPUT: ");
        printfDec(fail);
        printf("\n");
    #endif

    sysexit(&fail);
```

It is a simple binary search implementation. Process returns found index. If print mode is open, it also prints output.

LinearSearch Function

```
void linearSearch ()
{
    int arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170};
    int target = 175;
    int size = 10;
    int fail = -1;


    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {

            #ifdef PRINT_MODE
                printf("ID: ");
                printfHex32(taskManager.GetCurrentPid());
                printf(" OUTPUT: ");
                printfDec(i);
                printf("\n");
            #endif
            sysexit(&i);
        }
    }
```

```
   #ifdef PRINT_MODE
       printf("ID: ");
       printfHex32(taskManager.GetCurrentPid());
       printf(" OUTPUT: ");
       printfDec(fail);
       printf("\n");
   #endif

   sysexit(&fail);
```

It searches the target one by one, if print mode is open prints the result.

```c
void collatz ()
{
    // 2d array to store the sequence
    int sequence[25][50];

    // initialize all elements to 0
    for (int i = 0; i < 25; i++)
        for (int j = 0; j < 50; j++)
            sequence[i][j] = 0;

    // 1's sequence is 1
    sequence[0][0] = 1;

    int success;

    for (int i = 0; i < 25; i++) {
        int num = i + 1;

        int j = 0;

        // calculate till num becomes 1
        while (num != 1)
        {
            // if it is even
            if (num % 2 == 0)
                num = num / 2;
            else
                num = 3 * num + 1;

            sequence[i][j] = num;
            j++;
        }
    }
```

```c
#ifdef PRINT_MODE
    enterCritical();
    for (int i = 0; i < 25; i++) {
        int j = 0;
        printfDec(i + 1);
        printf(": ");
        while (sequence[i][j] != 0) {
            printfDec(sequence[i][j]);
            printf(" ");
            j++;
        }
        printf("\n");
    }
    exitCritical();
#endif

sysexit(&success);
```

Stores sequence in 2d array, and prints one by one each sequence. Sequence is printed in critical region, so it is not interrupted by timer interrupt.

printfDec Function (on the other page)

```
// print a decimal number
void printfDec(int num)
{
    if (num == 0)
    {
        printf("0");
        return;
    }

    char buffer[32];
    int i = 0;
    bool negativeFlag = false;

    if (num < 0)
    {
        num = -num;
        negativeFlag = true;
    }

    while (num != 0)
    {
        // convert number to string
        buffer[i] = "0123456789"[num % 10];
        num /= 10;
        i++;
    }
```

```
if (negativeFlag)
{
    // put a negative sign to end of the string
    buffer[i] = '-';
    i++;
}

buffer[i] = 0;

char temp;

// set cursors to beginning and end of the string
int j = 0;
i--;

// reverse the string
while (j < i)
{
    temp = buffer[j];
    buffer[j++] = buffer[i];
    buffer[i--] = temp;
}

printf(buffer);
```

I've written a simple printf for printing in decimal format. It simply puts every digit in a buffer, then reverses it. I used it in printing decimals for collatz algorithm.

PrintProcessTable Function

```
void TaskManager::PrintProcessTable()
{
    printf("\nProcess table:\n");
    printf("\nTotal tasks: ");
    printfHex32(numTasks - terminatedTask);
    printf("\n");
    for (int i = 0; i < 256; i++)
    {
        if (processTable.processes[i].GetPid() != 0)
        {
            printf("PID: ");
            printfHex32(processTable.processes[i].GetPid());
            printf(" PPID: ");
            printfHex32(processTable.processes[i].GetPpid());
            printf(" State: ");
            switch (processTable.processes[i].GetState())
            {
                case READY:
                    printf("READY");
                    break;
                case RUNNING:
                    printf("RUNNING");
                    break;
                case TERMINATED:
                    printf("TERMINATED");
                    break;
                case BLOCKED:
                    printf("BLOCKED");
                    break;
                default:
                    printf("UNKNOWN");
                    break;
            }
            printf(" Return Value: ");
            int* val = processTable.processes[i].GetReturnValue();
            if (*val < 0)
            {
                printf("-");
                printfHex32(*val * -1);
            }
            else
                printfHex32(*val);
            printf(" Num Children: ");
            printfHex32(processTable.processes[i].GetNumChildren());
            printf("\n");
        }
    }
    printf("\n");
}
```

4- Problems, Incomplete Parts

Fork system call

I've failed to implement the fork system call. I've tried to implement it using 0x80 interrupt. I've passed the eax register, then in the handler I called the fork function. Inside fork I invoked the task managers fork because I need access to cpu registers of a particular task. My plan is setting the eax register for child to 0, and setting the eax register for parent to child id. But for some reason, I always get the unhandled interrupt error 0x0d. I researched about that error, and it means this is an exception and it is a "General Protection Fault" exception. I think the problem is I cannot copy registers properly. Each cpu field is uint32 type, when I simply assign it for child, they are actually pointing same place with parent. After that one of them executes and when other one is scheduled, it throws a protection fault exception because other changed that location. I spent most of my time trying to solve this issue, but I've failed. This is the main reason for that error according to my understanding.

Other

Sometimes if there are too much printing involved(ex: printing table, or all sequence) process might not work accurate(not terminating properly).

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 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
- PRINT_MODE -> Prints every output 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.

In the screenshots, after initializing hardware, it prints bus and devices, I commented out them after some point on testing, so some of them appear on some screenshots.

Multiprogramming Test

CASE:

kernel.cpp

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

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

// #define PRINT_MODE
```

multitasking.cpp

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

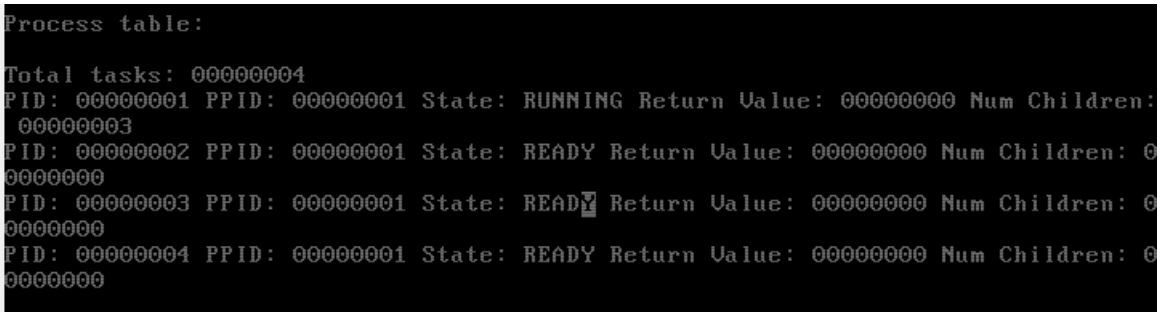
```
void taskA()
{
    int pid;
    pid = sysgetpid();
    printf("TASKA\n");
    printf("PID: ");
    printfHex32(pid);
    printf("\n");

    while(true);
}
```

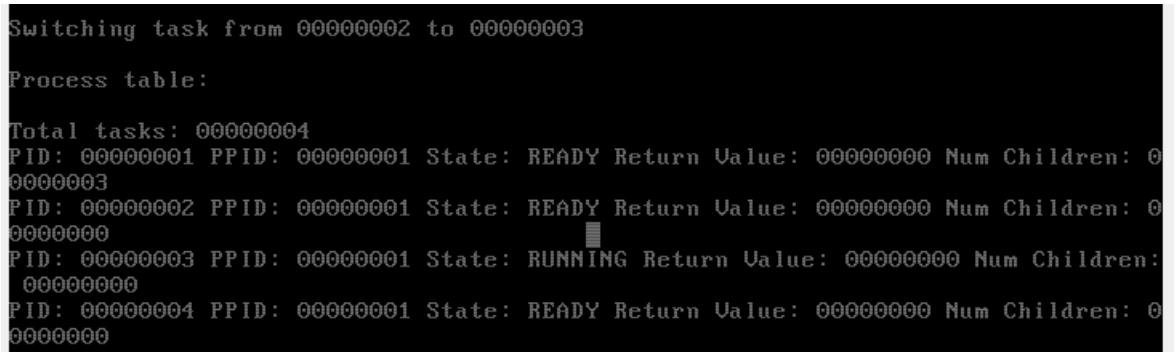
```
void taskB()
{
    int pid;
    pid = sysgetpid();
    printf("TASKB\n");
    printf("PID: ");
    printfHex32(pid);
    printf("\n");

    while(true);
}
```

```
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 timer interrupt, they are non-terminating processes.

RESULT:





```
Switching task from 00000004 to 00000001

Process table:

Total tasks: 00000004
PID: 00000001 PPID: 00000001 State: RUNNING Return Value: 00000000 Num Children:
 00000003
PID: 00000002 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
PID: 00000003 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
PID: 00000004 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
```

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

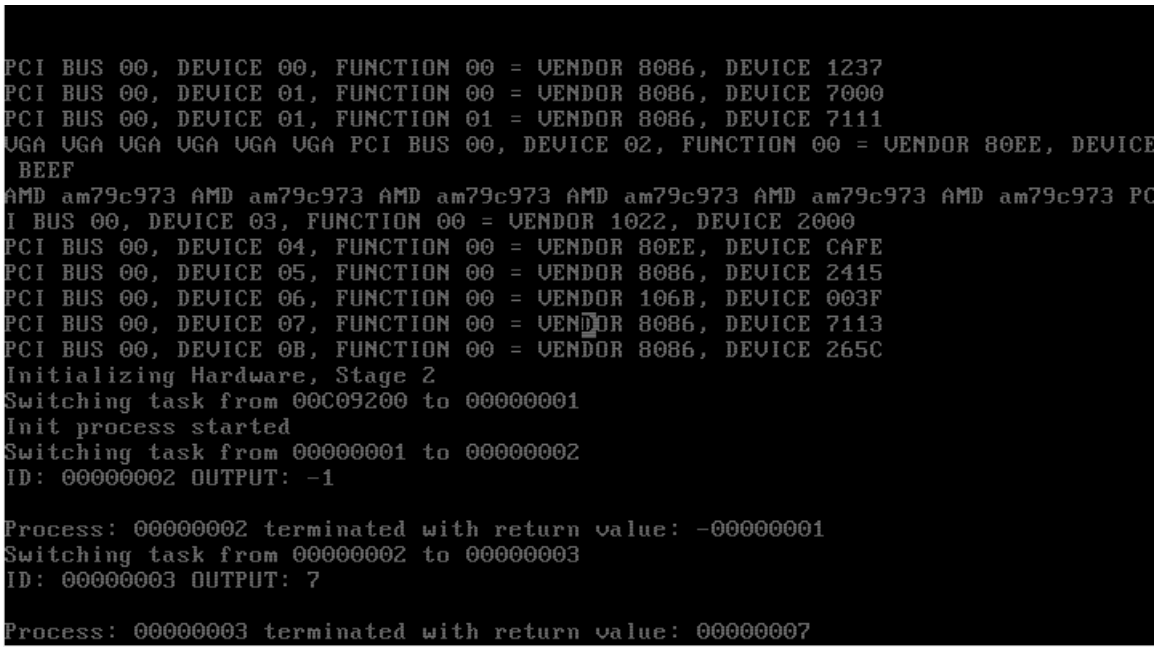
Strategy 1 – Microkernel1 Test

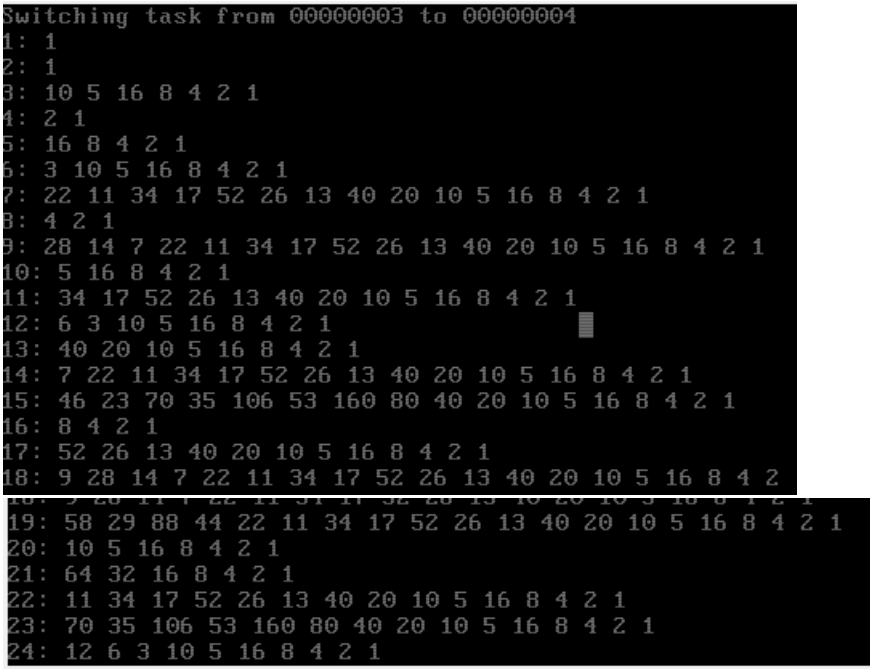
CASE1:

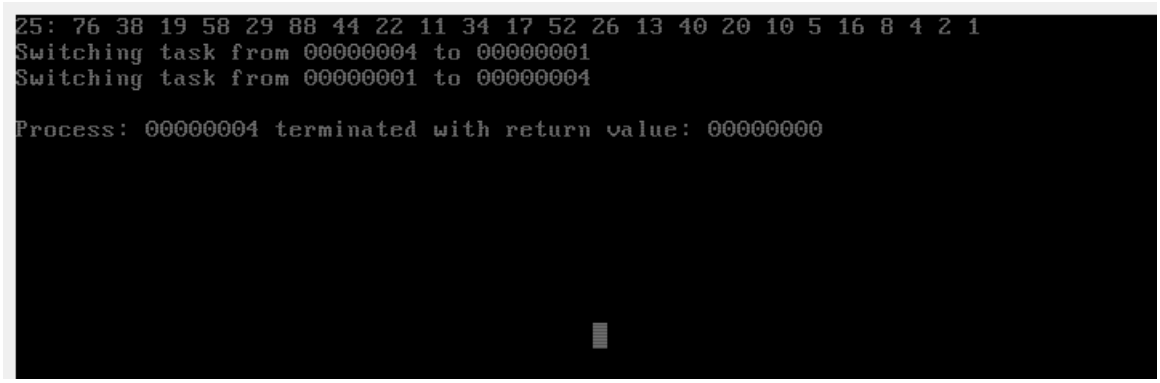
```
#define MICROKERNEL1
// #define MICROKERNEL2
// #define MICROKERNEL3

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

#define PRINT_MODE
```

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

RESULT1:







- First output is -1 for linear search as expected. 7 is the expected output for binary search(in the pdf it is said 6, but it must be -1, so I modified

the target to show the case when one of algorithms output is not -1). For the collatz, it prints the sequence.

- Switch is done correctly, it starts with init process, then goes one by one, terminated processes are not switched, in the end it switches from 4 to 1 2 times because printing might take long, and I locked printing part when printing finishes it immediately gets timer interrupt and switches. Then when it is scheduled again it is terminated.

CASE2:

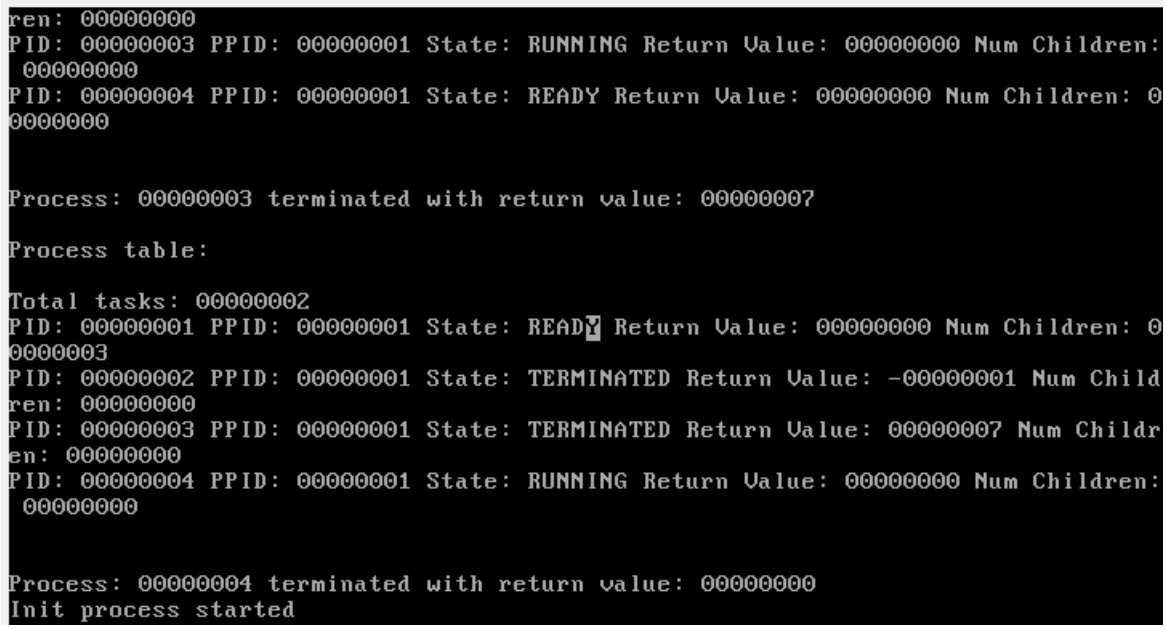
```
#define MICROKERNEL1
// #define MICROKERNEL2
// #define MICROKERNEL3

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

// #define PRINT_MODE
```

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

RESULT2:

```
hello
Initializing Hardware, Stage 1
PCI BUS 00, DEVICE 00, FUNCTION 00 = VENDOR 8086, DEVICE 1237
PCI BUS 00, DEVICE 01, FUNCTION 00 = VENDOR 8086, DEVICE 7000
PCI BUS 00, DEVICE 01, FUNCTION 01 = VENDOR 8086, DEVICE 7111
VGA VGA VGA VGA VGA VGA PCI BUS 00, DEVICE 02, FUNCTION 00 = VENDOR 80EE, DEVICE
 BEEF
AMD am79c973 AMD am79c973 AMD am79c973 AMD am79c973 AMD am79c973 AMD am79c973 PC
I BUS 00, DEVICE 03, FUNCTION 00 = VENDOR 1022, DEVICE 2000
PCI BUS 00, DEVICE 04, FUNCTION 00 = VENDOR 80EE, DEVICE CAFE
PCI BUS 00, DEVICE 05, FUNCTION 00 = VENDOR 8086, DEVICE 2415
PCI BUS 00, DEVICE 06, FUNCTION 00 = VENDOR 106B, DEVICE 003F
PCI BUS 00, DEVICE 07, FUNCTION 00 = VENDOR 8086, DEVICE 7113
PCI BUS 00, DEVICE 0B, FUNCTION 00 = VENDOR 8086, DEVICE 265C
Initializing Hardware, Stage 2

Process table:

Total tasks: 00000004
PID: 00000001 PPID: 00000001 State: RUNNING Return Value: 00000000 Num Children:
 00000003
PID: 00000002 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
PID: 00000003 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
00
```

```
PID: 00000004 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000


Process table:

Total tasks: 00000004
PID: 00000001 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000003
PID: 00000002 PPID: 00000001 State: RUNNING Return Value: 00000000 Num Children:
 00000000
PID: 00000003 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
PID: 00000004 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000


Process: 00000002 terminated with return value: -00000001
```



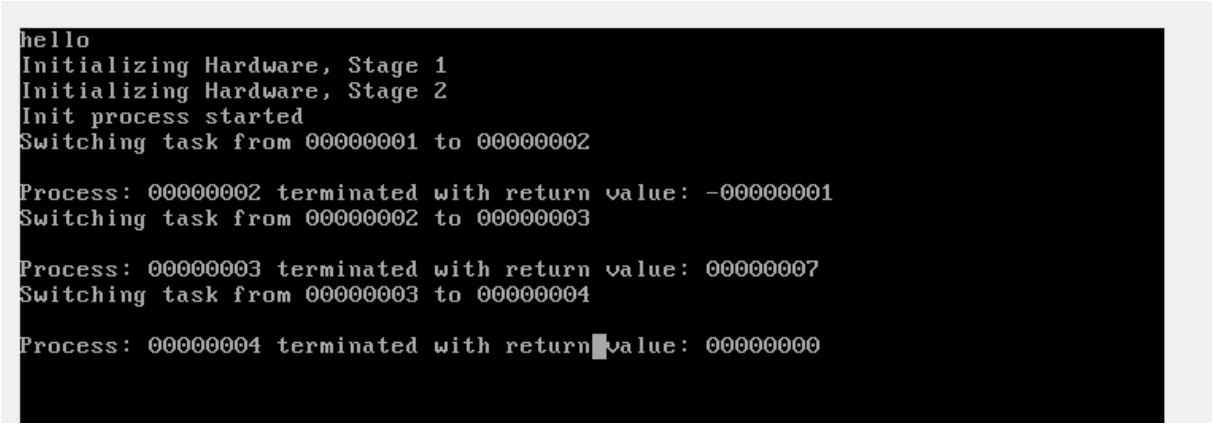
- Every processes parent id is 1 which is init process. 0 is default return value, and other processes -1, and 7 as expected.
- After one process terminates, it can be seen their state becomes terminated. In the last table process 4 is running, after it is terminated process 1 runs, and since all processes except init are terminated, then there is no context switch.
- It prints init process started in the end because, in this case timer interrupt comes immediately, and this cause to process 1 cannot print its message, after round robin comes to starting point it prints it message. It can be prevented using enterCritical, exitCritical functions.

CASE3:

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

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

// #define PRINT_MODE
```

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

RESULT3:



- This is the simplest output of what happens.

Strategy 2 – Microkernel2 Test

CASE1:

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

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

// #define PRINT_MODE
```

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

RESULT1:

```
hello
Initializing Hardware, Stage 1
Initializing Hardware, Stage 2
Init process started
Switching task from 00000001 to 00000002

Process: 00000002 terminated with return value: 00000000
Switching task from 00000002 to 00000003

Process: 00000003 terminated with return value: 00000000
Switching task from 00000003 to 00000004

Process: 00000004 terminated with return value: 00000000
Switching task from 00000004 to 00000005

Process: 00000005 terminated with return value: 00000000
Switching task from 00000005 to 00000006

Process: 00000006 terminated with return value: 00000000
Switching task from 00000006 to 00000007

Process: 00000007 terminated with return value: 00000000
```

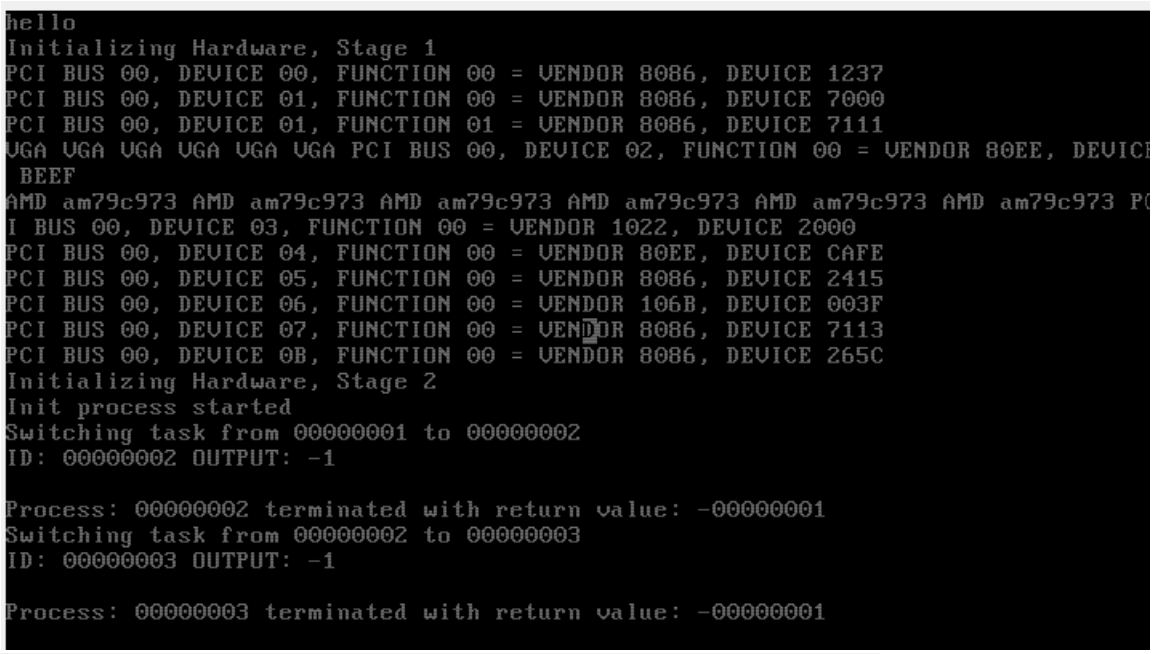
```
Switching task from 00000008 to 00000009

Process: 00000009 terminated with return value: 00000000
Switching task from 00000009 to 0000000A

Process: 0000000A terminated with return value: 00000000
Switching task from 0000000A to 0000000B

Process: 0000000B terminated with return value: 00000000
```

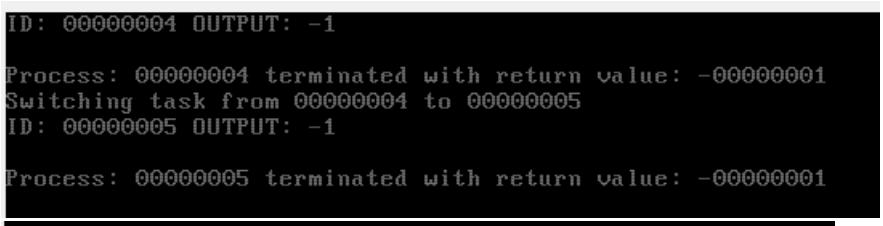
CASE2:

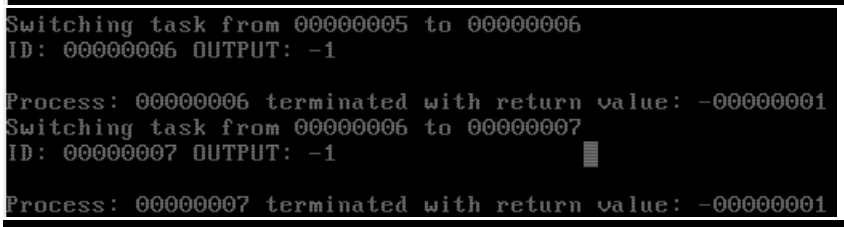
```
// #define MICROKERNEL1
#define MICROKERNEL2
// #define MICROKERNEL3

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

#define PRINT_MODE
```

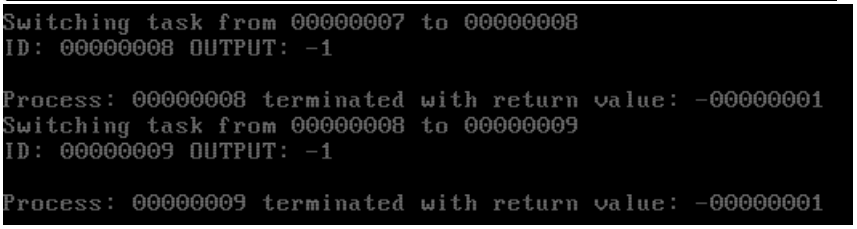
```
#define SWITCH_PRINT_MODE
// #define PT_PRINT_MODE
```

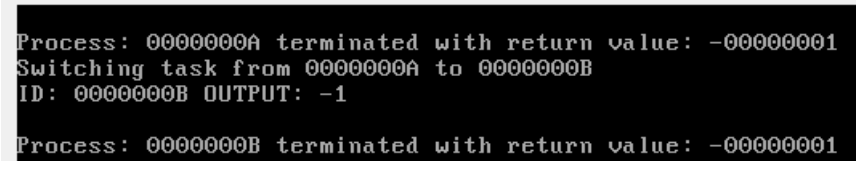
RESULT2:





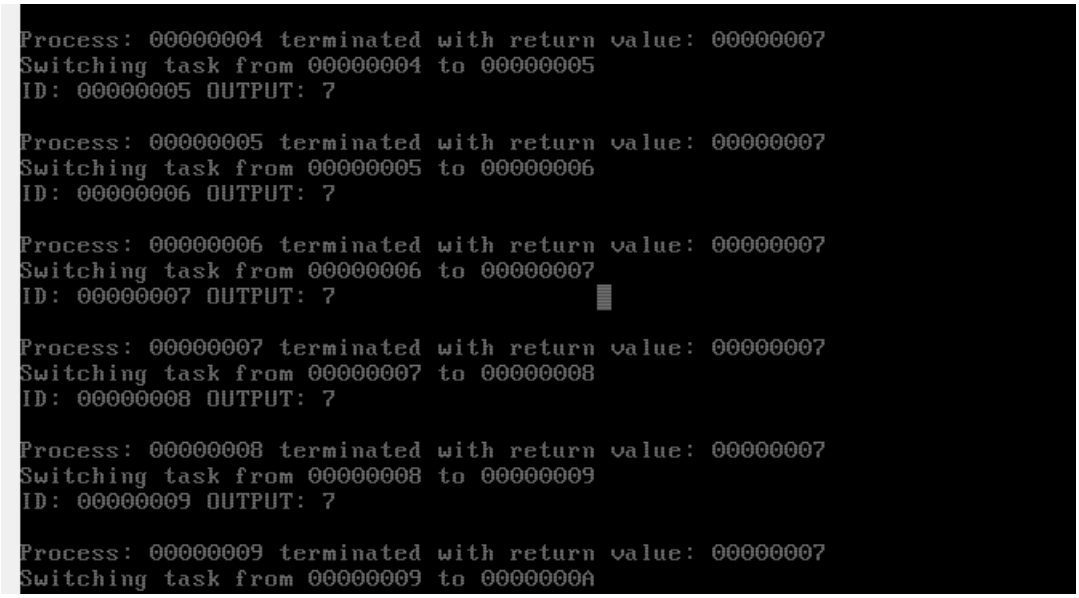






- Random generation chose linear search, and they are processed one by one. 10 processes and init processes are loaded. Each process terminated one by one, round couldn't happen here because all of them terminated before they are scheduled again.

RESULT3:



- For different runs, it loaded different processes.

Strategy 3 – Microkernel3 Test

CASE1:

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

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

// #define PRINT_MODE
```

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

RESULT1:

```
hello
Initializing Hardware, Stage 1
Initializing Hardware, Stage 2
Init process started
Switching task from 00000001 to 00000002

Process: 00000002 terminated with return value: 00000000
Switching task from 00000002 to 00000003

Process: 00000003 terminated with return value: 00000000
Switching task from 00000003 to 00000004

Process: 00000004 terminated with return value: 00000000
Switching task from 00000004 to 00000005

Process: 00000005 terminated with return value: 00000007
Switching task from 00000005 to 00000006

Process: 00000006 terminated with return value: 00000007
Switching task from 00000006 to 00000007

Process: 00000007 terminated with return value: 00000007
```

- It choses binary search, and collatz. They are loaded, and switched with round robin. They all terminated in each switch, after all of them terminated. It returned to the init process, and switching done.

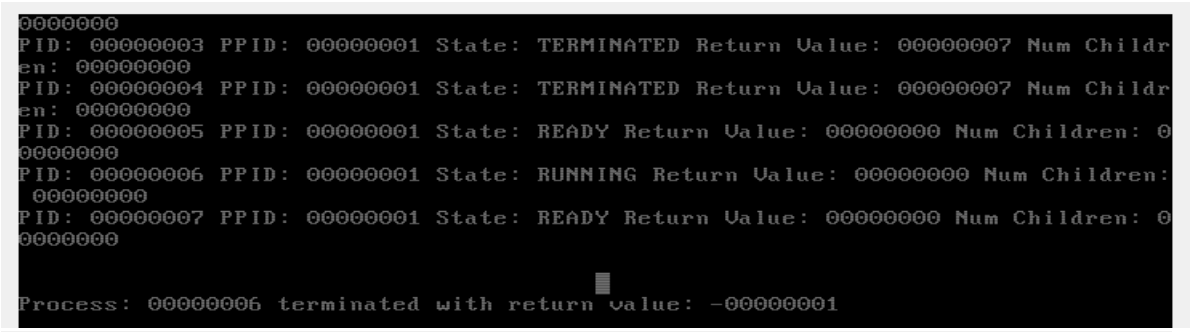
CASE2:

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

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

// #define PRINT_MODE
```

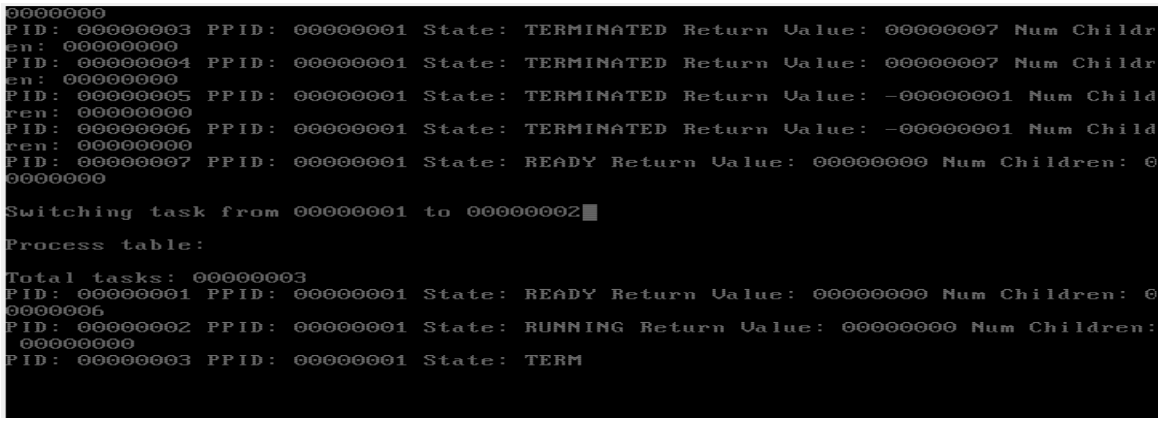
```
#define SWITCH_PRINT_MODE
#define PT_PRINT_MODE
```

RESULT2:


Switching task from 00000003 to 00000004
Process table:
Total tasks: 00000006
PID: 00000001 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000006
PID: 00000002 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
PID: 00000003 PPID: 00000001 State: TERMINATED Return Value: 00000007 Num Childr
en: 00000000
PID: 00000004 PPID: 00000001 State: RUNNING Return Value: 00000000 Num Children:
00000000
PID: 00000005 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
PID: 00000006 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
PID: 00000007 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
Process: 00000004 terminated with return value: 00000007


0000000
PID: 00000003 PPID: 00000001 State: TERMINATED Return Value: 00000007 Num Childr
en: 00000000
PID: 00000004 PPID: 00000001 State: TERMINATED Return Value: 00000007 Num Childr
en: 00000000
PID: 00000005 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
PID: 00000006 PPID: 00000001 State: RUNNING Return Value: 00000000 Num Children:
00000000
PID: 00000007 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
Process: 00000006 terminated with return value: -00000001


Switching task from 00000002 to 00000005
Process table:
Total tasks: 00000004
PID: 00000001 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000006
PID: 00000002 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
PID: 00000003 PPID: 00000001 State: TERMINATED Return Value: 00000007 Num Childr
en: 00000000
PID: 00000004 PPID: 00000001 State: TERMINATED Return Value: 00000007 Num Childr
en: 00000000
PID: 00000005 PPID: 00000001 State: RUNNING Return Value: 00000000 Num Children:
00000000
PID: 00000006 PPID: 00000001 State: TERMINATED Return Value: -00000001 Num Child
ren: 00000000
PID: 00000007 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
Process: 00000005 terminated with return value: -00000001


0000000
PID: 00000003 PPID: 00000001 State: TERMINATED Return Value: 00000007 Num Childr
en: 00000000
PID: 00000004 PPID: 00000001 State: TERMINATED Return Value: 00000007 Num Childr
en: 00000000
PID: 00000005 PPID: 00000001 State: TERMINATED Return Value: -00000001 Num Child
ren: 00000000
PID: 00000006 PPID: 00000001 State: TERMINATED Return Value: -00000001 Num Child
ren: 00000000
PID: 00000007 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
Switching task from 00000001 to 00000002
Process table:
Total tasks: 00000003
PID: 00000001 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000006
PID: 00000002 PPID: 00000001 State: RUNNING Return Value: 00000000 Num Children:
00000000
PID: 00000003 PPID: 00000001 State: TERM

```
00000000


ProcessSwitching task from 00000007 to 00000001

Process table:

Total tasks: 00000002
PID: 00000001 PPID: 00000001 State: RUNNING Return Value: 00000000 Num Children:
 00000006
PID: 00000002 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000000
PID: 00000003 PPID: 00000001 State: TERMINATED Return Value: 00000007 Num Childr
en: 00000000
PID: 00000004 PPID: 00000001 State: TERMINATED Return Value: 00000007 Num Childr
en: 00000000
PID: 00000005 PPID: 00000001 State: TERMINATED Return Value: -00000001 Num Child
ren: 00000000
PID: 00000006 PPID: 00000001 State: TERMINATED Return Value: -00000001 Num Child
ren: 00000000
PID: 00000007 PPID: 00000001 State: TERMINATED Return Value: -00000001 Num Child
ren: 00000000
```

```
Process table:

Total tasks: 00000002
PID: 00000001 PPID: 00000001 State: READY Return Value: 00000000 Num Children: 0
0000006
PID: 00000002 PPID: 00000001 State: RUNNING Return Value: 00000000 Num Children:
 00000000
PID: 00000003 PPID: 00000001 State: TERMINATED Return Value: 00000007 Num Childr
en: 00000000
PID: 00000004 PPID: 00000001 State: TERMINATED Return Value: 00000007 Num Childr
en: 00000000
PID: 00000005 PPID: 00000001 State: TERMINATED Return Value: -00000001 Num Child
ren: 00000000
PID: 00000006 PPID: 00000001 State: TERMINATED Return Value: -00000001 Num Child
ren: 00000000
PID: 00000007 PPID: 00000001 State: TERMINATED Return Value: -00000001 Num Child
ren: 00000000


Process: 00000002 terminated with return valu
```

- Process table is printed, linear search and binary search are chosen in this case, I couldn't take all screenshots it flow too fast, but this is the general image. At the end all processes except first process are terminated and switching is done.

CASE3:

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

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

#define PRINT_MODE
```

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

RESULT3:

```
hello
Initializing Hardware, Stage 1
PCI BUS 00, DEVICE 00, FUNCTION 00 = VENDOR 8086, DEVICE 1237
PCI BUS 00, DEVICE 01, FUNCTION 00 = VENDOR 8086, DEVICE 7000
PCI BUS 00, DEVICE 01, FUNCTION 01 = VENDOR 8086, DEVICE 7111
VGA VGA VGA VGA VGA VGA PCI BUS 00, DEVICE 02, FUNCTION 00 = VENDOR 80EE, DEVICE
 BEEF
AMD am79c973 AMD am79c973 AMD am79c973 AMD am79c973 AMD am79c973 AMD am79c973 PC
I BUS 00, DEVICE 03, FUNCTION 00 = VENDOR 1022, DEVICE 2000
PCI BUS 00, DEVICE 04, FUNCTION 00 = VENDOR 80EE, DEVICE CAFE
PCI BUS 00, DEVICE 05, FUNCTION 00 = VENDOR 8086, DEVICE 2415
PCI BUS 00, DEVICE 06, FUNCTION 00 = VENDOR 106B, DEVICE 003F
PCI BUS 00, DEVICE 07, FUNCTION 00 = VENDOR 8086, DEVICE 7113
PCI BUS 00, DEVICE 0B, FUNCTION 00 = VENDOR 8086, DEVICE 265C
Initializing Hardware, Stage 2
Init process started
Switching task from 00000001 to 00000002
ID: 00000002 OUTPUT: 7

Process: 00000002 terminated with return value: 00000007
Switching task from 00000002 to 00000003
ID: 00000003 OUTPUT: 7

Process: 00000003 terminated with return value: 00000007
```

```
ID: 00000004 OUTPUT: 7

Process: 00000004 terminated with return value: 00000007
Switching task from 00000004 to 00000005
ID: 00000005 OUTPUT: -1

Process: 00000005 terminated with return value: -00000001
Switching task from 00000005 to 00000006
ID: 00000006 OUTPUT: -1

Process: 00000006 terminated with return value: -00000001
Switching task from 00000006 to 00000007
ID: 00000007 OUTPUT: -1

Process: 00000007 terminated with return value: -00000001
```

Execve Test

CASE:

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

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

// #define PRINT_MODE
```

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

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

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

It is expected to print “old”, then “success!”.

RESULT:

```
hello
Initializing Hardware, Stage 1
Initializing Hardware, Stage 2
Init process started
Switching task from 00000001 to 00000002
old
execve() called
success!

Process: 00000002 terminated with return value: 00000000
```

- It initally printed old, then system call is made and its changed its core image, old process is discarded.

Getpid Test

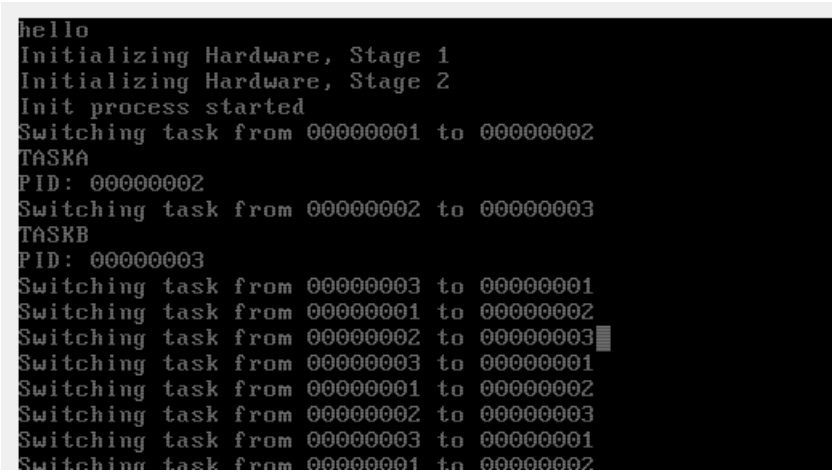
CASE:

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

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

// #define PRINT_MODE
```

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

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

    while(true);
}

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

    while(true);
}
```

RESULT:



- They print the correct id, and since taskA, and taskB does not terminate, it will be scheduled continuously.