CSE312 OPERATING SYSTEMS

HOMEWORK 1

REPORT

Burak Demirkaya

210104004274

20/05/2024

1. Design Decisions

I have watched the Youtube playlist up until the system calls part and after that I have started adjusting the code depending on the needs from the homework PDFs.

```
namespace myos{
    class SyscallHandler : public InterruptHandler{
        private:
            TaskManager* taskManager;

        public:

            SyscallHandler(InterruptManager* interruptManager, uint8_t interruptNumber);
            ~SyscallHandler();

            virtual uint32_t HandleInterrupt(uint32_t esp);
    };
    // Asm functions for syscalls
    int getPid(); // Get PID
    void waitpid(uint8_t pid); // Waitpid
    void fork();
    void fork(int* pid); // Fork
    void exit(); // Exit
    int exec(void (*entrypoint)()); // Execve
}
```

I have added necessary system call functions to the syscall handler class.

```
class InterruptHandler{
    protected:
        myos::common::uint8_t InterruptNumber;
        InterruptManager *interruptManager;
        InterruptHandler(InterruptManager *interruptManager, myos::common::uint8_t InterruptNumber);
        ~InterruptHandler();
        uint32_t Execve(uint32_t entrypoint); // Execve
        uint32_t GetPid(); // Get PID
        uint32_t Fork(CPUState* cpustate); // Fork
        bool Exit(); // Exit
        bool WaitPid(uint32_t pid); // Waitpid
```

I have defined the necessary functions for each system call from the interrupt handler so that I can call the multitasking's functions.

```
class Task{
        friend class TaskManager;
    private:
        static uint32_t nextTaskId; // Static variable to keep track of the next task id
        uint32_t pid; // Process ID
        uint32_t ppid; // Parent Process ID
        TaskState state; // Task State
        Priority priority; // Task Priority
        uint32_t waitpid; // Waitpid
        uint8_t stack[4096]; // 4KB
        CPUState* cpustate; // CPU State
    public:
        Task(GlobalDescriptorTable* gdt, void entrypoint(), Priority priority);
        ~Task();

};
```

```cpp
enum TaskState{
    READY,
    WAITING,
    TERMINATED
};

enum Priority{
    HIGH,
    MEDIUM,
    LOW
};
```

In the multitasking.h file, I have declared two enums to represent the state (TaskState) and priority (Priority) of a task. Furthermore, within the Task class, I have included several data members. These data members are the process ID (pid), parent process ID (ppid), task state (state), task priority (priority), and the wait process ID (waitpid).

```cpp
class TaskManager{
    friend class InterruptHandler;


    private:
        Task* tasks[256]; // Array of tasks
        int numTasks; // Number of tasks
        int currentTask; // Current Task
        GlobalDescriptorTable* gdt; // Global Descriptor Table
        int getTaskIndex(uint32_t pid); // Get Task Index

    protected:

    public:
        TaskManager(GlobalDescriptorTable* gdt);
        ~TaskManager();
        bool AddTask(Task* task);
        CPUState* Schedule(CPUState* cpustate); // Round Robin

        void PrintProcessTable(); // Print the process table
        uint32_t ExecveTask(void entrypoint()); // Execve
        uint32_t getPid(); // Get PID
        uint32_t ForkTask(CPUState* cpustate); // Fork
        bool ExitTask(); // Exit
        bool WaitTask(uint32_t esp); // Waitpid
};
```

Within the TaskManager class, I have stored the tasks as an array of pointers. Additionally, I have added system call functions into the class to handle essential operations such as task scheduling, process table printing, task execution, getPid, task forking, exiting a task, and waiting for a task to complete.

```
int myos::getPid(){ // Get PID
    int ret;
    asm("int $0x80" : "=c" (ret) : "a" (20));
    return ret;
}

void myos::waitpid(uint8_t pid){ // Waitpid
    asm("int $0x80" : : "a" (7), "b" (pid));
}

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

void myos::exit(){ // Exit
    asm("int $0x80" : : "a" (1));
}

int myos::exec(void (*entrypoint)()){ // Execve
    int ret;
    asm("int $0x80" : "=c" (ret) : "a" (11), "b" ((uint32_t)entrypoint));
    return ret;
}
```

In the syscalls.cpp file, I have implemented several system call functions using inline assembly. The “a” means the eax register of the CPU, to handle the system calls. I have searched online for the corresponding eax values for Linux System Calls and defined accordingly.

```
uint32_t SyscallHandler::HandleInterrupt(uint32_t esp){ // Handle Interrupt
    CPUState* cpu = (CPUState*) esp; // Cast the esp to a CPUState

    switch(cpu->eax){ // Switch on the syscall number
        case 1: // Exit
            if(InterruptHandler::Exit()){
                return InterruptHandler::HandleInterrupt(esp);
            }
            break;
        case 2: // Fork
            cpu->ecx = InterruptHandler::Fork(cpu);
            return InterruptHandler::HandleInterrupt(esp);
            break;
        case 7: // Waitpid
            if(InterruptHandler::WaitPid(esp)){
                return InterruptHandler::HandleInterrupt(esp);
            }
            break;
        case 11: // Execve
            esp = InterruptHandler::Execve(cpu->ebx);
            break;
        case 20: // Get PID
            cpu->ecx = InterruptHandler::GetPid();
            break;
        default:
            break;
    }
    return esp;
}
```

Later inside the syscalls.cpp depending on the eax register’s value I have called the corresponding functions.

```
uint32_t InterruptHandler::Execve(uint32_t entrypoint){ // execve
    return interruptManager->taskManager->ExecveTask((void (*)())entrypoint);
}

uint32_t InterruptHandler::GetPid(){ // getpid
    return interruptManager->taskManager->getPid();
}

uint32_t InterruptHandler::Fork(CPUState* cpustate){ // fork
    return interruptManager->taskManager->ForkTask(cpustate);
}

bool InterruptHandler::Exit(){ // exit
    return interruptManager->taskManager->ExitTask();
}

bool InterruptHandler::WaitPid(uint32_t pid){ // waitpid
    return interruptManager->taskManager->WaitTask(pid);
}
```

By calling these functions through the switch case at the syscalls.cpp I have called the multitasking.cpp's system call functions because those functions are the ones that handle most of the system call part.

EXECVE

```
uint32_t TaskManager::ExecveTask(void entrypoint()){ // Execute a new task
    tasks[currentTask]->state = READY; // Set the state of the current task to READY
    tasks[currentTask]->cpustate = (CPUState*)(tasks[currentTask]->stack + 4096 - sizeof(CPUState)); // Set the CPU state of the current task to the top of the stack

    tasks[currentTask]->cpustate->eax = 0; // Set the eax register of the CPU state of the current task to 0
    tasks[currentTask]->cpustate->ebx = 0; // Set the ebx register of the CPU state of the current task to 0
    tasks[currentTask]->cpustate->ecx = tasks[currentTask]->pid; // Set the ecx register of the CPU state of the current task to the PID of the current task
    tasks[currentTask]->cpustate->edx = 0; // Set the edx register of the CPU state of the current task to 0

    tasks[currentTask]->cpustate->esi = 0; // Set the esi register of the CPU state of the current task to 0
    tasks[currentTask]->cpustate->edi = 0; // Set the edi register of the CPU state of the current task to 0
    tasks[currentTask]->cpustate->ebp = 0; // Set the ebp register of the CPU state of the current task to 0

    tasks[currentTask]->cpustate->eip = (uint32_t)entrypoint; // Set the eip register of the CPU state of the current task to the entry point of the new task
    tasks[currentTask]->cpustate->cs = gdt->CodeSegmentSelector(); // Set the cs register of the CPU state of the current task to the code segment selector of the GDT
    tasks[currentTask]->cpustate->eflags = 0x202; // 0x202 is the value of the flags register when the CPU is in kernel mode
    return (uint32_t)tasks[currentTask]->cpustate; // Return the CPU state of the current task
}
```

Inside ExecveTask function I have set the state of the currentTask to be READY and set the cpustate registers of the currentTask accordingly. The only important part here is that the eip register which is the instruction pointer of the task that means where the program will start executing.

GETPID

```
uint32_t TaskManager::getPid(){ // Get the PID of the current task
    return tasks[currentTask]->pid;
}
```

getPID function just simply returns the PID of the task.

FORK

```cpp
uint32_t TaskManager::ForkTask(CPUState* cpustate){ // Fork the current task
    if(numTasks >= 256){
        return 0;
    }

    typedef void (*EntryPointType)();  // Define a function pointer type
    EntryPointType entrypoint = (EntryPointType)cpustate->eip;  // Cast to the function pointer type
    tasks[numTasks] = new Task(gdt, entrypoint, tasks[currentTask]->priority);


    for(int i = 0; i < sizeof(tasks[currentTask]->stack); i++){ // Copy the stack of the current task to the stack of the new task
        tasks[numTasks]->stack[i] = tasks[currentTask]->stack[i];
    }

    uint32_t currentTaskOffset = (uint32_t)cpustate - (uint32_t)tasks[currentTask]->stack; // Calculate the offset of the CPU state of the current task
    tasks[numTasks]->cpustate = (CPUState*)((uint32_t)tasks[numTasks]->stack + currentTaskOffset); // Set the CPU state of the new task to the offset of the CPU state of the current task

    tasks[numTasks]->state = READY; // Set the state of the new task to READY
    tasks[numTasks]->ppid = tasks[currentTask]->pid; // Set the PPID of the new task to the PID of the current task
    //tasks[numTasks]->pid = Task::nextTaskId; // Set the PID of the new task to the next task ID
    tasks[numTasks]->priority = tasks[currentTask]->priority; // Set the priority of the new task to the priority of the current task

    tasks[numTasks]->cpustate->ecx = 0; // Set the ecx register of the CPU state of the new task to 0
    numTasks++; // Increment the number of tasks

    return tasks[numTasks-1]->pid; // Return the PID of the new task
}
```

ForkTask function defines a function pointer type EntryPointType and sets it to the instruction pointer (eip) of the current task's CPU state. This will be the entry point of the new task. A new Task object is then created with the same priority as the current task and added to the task array. The function then copies the stack of the current task to the new task. This is done to ensure that the new task has the same execution context as the parent task. The offset of the CPU state of the current task is calculated and used to set the CPU state of the new task. This ensures that the new task starts executing at the same point in the code as the parent task. The state of the new task is set to READY. The parent process ID (ppid) of the new task is set to the process ID (pid) of the current task, and the priority of the new task is set to the priority of the current task. The ecx register of the new task's CPU state is set to 0 because it should understand that it is the child process. Finally, the function returns the process ID (pid) of the new task so that the parent can understand that it will wait this pid.

```cpp
uint32_t Task::nextTaskId = 1;

Task::Task(GlobalDescriptorTable* gdt, void entrypoint(), Priority priority){
    cpustate = (CPUState*)(stack + 4096 - sizeof(CPUState));

    cpustate->eax = 0;
    cpustate->ebx = 0;
    cpustate->ecx = 0;
    cpustate->edx = 0;

    cpustate->esi = 0;
    cpustate->edi = 0;
    cpustate->ebp = 0;
    // cpustate->esp = 0;

    // cpustate->gs = 0;
    // cpustate->fs = 0;
    // cpustate->es = 0;
    // cpustate->ds = 0;

    // cpustate->error = 0;

    // cpustate->esp = ;
    cpustate->eip = (uint32_t)entrypoint;
    cpustate->cs = gdt->CodeSegmentSelector();
    // cpustate->ss = gdt->DataSegmentSelector();
    cpustate->eflags = 0x202; // 0x202 is the value of the flags register when the CPU is in kernel mode

    this->state = READY;
    this->pid = Task::nextTaskId++;
    this->priority = priority;
}
```

In the Task constructor within the multitasking.cpp file, I have set up the process for creating new tasks. This constructor is used both when creating parent processes in the kernel main and when creating child processes via the forkTask function.

EXIT

```cpp
bool TaskManager::ExitTask(){
    tasks[currentTask]->state = TERMINATED; // Set the state of the current task to TERMINATED
    return true;
}
```

ExitTask just simply sets the state of the task to TERMINATED so that the task will not be scheduled inside the schedule function.

WAITPID

```cpp
bool TaskManager::WaitTask(uint32_t esp){ // Wait for a task to finish
    CPUState* cpustate = (CPUState*) esp; // Get the CPU state of the current task
    uint32_t pid = cpustate->ebx; // Get the PID of the task to wait for

    int taskIndex = getTaskIndex(pid); // Get the index of the task to wait for
    if(taskIndex == -1){
        printf("Task not found\n");
        return false;
    }

    if(numTasks <= taskIndex || tasks[taskIndex]->state == TERMINATED){ // If the task to wait for has terminated
        printf("Task already terminated\n");
        return true;
    }

    tasks[currentTask]->cpustate = cpustate; // Set the CPU state of the current task to the CPU state of the task to wait for
    tasks[currentTask]->waitpid = pid; // Set the waitpid of the current task to the PID of the task to wait for
    tasks[currentTask]->state = WAITING; // Set the state of the current task to WAITING

    printf("Waiting for task: ");

    return true;
}
```

I have implemented the WaitTask function within the TaskManager class. The function begins by retrieving the CPU state of the current task from the stack pointer (esp). It then extracts the Process ID (PID) of the task to wait for from the ebx register of the current task's CPU state which was set by the asm function. Next, it calls the getTaskIndex function to get the index of the task to wait for in the task array. The function then checks if the task to wait for has already terminated. If the task to wait for is still running, the function sets the CPU state of the current task to the CPU state of the task to wait for. It also sets the waitpid of the current task to the PID of the task to wait for and changes the state of the current task to WAITING.

```cpp
CPUState* TaskManager::Schedule(CPUState* cpustate){ // Schedule the next task to run using Round Robin
    if (numTasks <= 0) {
        return cpustate; // No task to schedule.
    }

    if (currentTask >= 0) {
        tasks[currentTask]->cpustate = cpustate; // Save the CPU state of the current task.
    }

    PrintProcessTable(); // Print the process table

    int highestPriority = 4; // Assume lower numbers are higher priority.
    int nextTask = -1;

    // Iterate over all tasks to find the highest-priority READY task.
    for (int i = 0; i < numTasks; i++) {
        int idx = (currentTask + 1 + i) % numTasks; // Round Robin index
        Task* task = tasks[idx];

        // Check if the task is READY and has higher priority than the currently found highest.
        if (task->state == READY && task->priority < highestPriority) {
            highestPriority = task->priority;
            nextTask = idx;
        }

        // Handle tasks in WAITING state.
        if (task->state == WAITING && task->waitpid > 0) {
            int waitpid = getTaskIndex(task->waitpid); // Get the index of the task that the current task is waiting for.
            if (waitpid >= 0 && tasks[waitpid]->state == TERMINATED) {
                task->waitpid = 0; // Reset the waitpid of the current task.
                task->state = READY; // Set the state of the current task to READY.
                if (task->priority < highestPriority) {
                    highestPriority = task->priority;
                    nextTask = idx;
                }
            }
        }
    }
```

```
if (nextTask != -1) {
    currentTask = nextTask; // Set the current task to the highest-priority task found.
} else if (currentTask == -1 || tasks[currentTask]->state != READY) {
    // Fallback if no READY tasks are found and current task is not eligible.
    for (int i = 0; i < numTasks; i++) {
        if (tasks[i]->state == READY) {
            currentTask = i;
            break;
        }
    }
}

return tasks[currentTask]->cpustate; // Return the CPU state of the scheduled task.
```

In the multitasking.cpp file, I have implemented the Schedule function within the TaskManager class. This function is designed to schedule the next task to run using a Round Robin scheduling algorithm. The function begins by checking if there are any tasks to schedule. If there are no tasks it returns the current CPU state. If there is a current task it saves the CPU state of the current task. This is done to preserve the execution context of the task when it is resumed. Next, it initializes highestPriority to 4 and nextTask to -1. These variables are used to keep track of the highest priority task that is ready to run. The function then iterates over all tasks to find the highest-priority task that is ready to run. It uses a Round Robin scheduling algorithm, which cycles through tasks in a circular queue.

If a task is in the READY state and has a higher priority than the currently found highest, it updates highestPriority and nextTask. If a task is in the WAITING state and is waiting for another task it checks if the task it is waiting for has terminated. If so, it resets the waitpid of the current task, sets its state to READY. If a highest-priority task was found it sets the current task to this task. If no ready tasks were found and the current task is not ready, it falls back to the first ready task it can find. Finally, it returns the CPU state of the scheduled task. This will be used by the operating system to resume execution of the scheduled task.

2. Test Results

```
run: mykernel.iso
#	(killall VirtualBox && sleep 1) || true
#	VirtualBox --startvm "My Operating System" &
	/mnt/c/'Program Files'/Oracle/VirtualBox/VBoxManage.exe startvm "My Operating System" &
```

I worked with WSL and Virtual Machine so I modified the makefile run part as this way to test the results in VM. When I send the homework as zip I have commented this part.

Tested Algorithms

```
int long_running_program(int n){
    int result = 0;
    for(int i = 0; i < n; i++){
        for(int j=0; j<n; ++j){
            result += i*j;
        }
    }
    return result;
}

void call_long_running_program(){
    printf("Calling Long Running Program\n");
    int result = long_running_program(10000);
    printf("Result: ");
    printNumber(result);
    printf("\n");
    exit();
}
```

```
void collatz(int n){
    printf("Collatz ");
    printNumber(n);
    printf(": ");
    while(n != 1){
        if(n % 2 == 0){
            n = n / 2;
        }else{
            n = 3*n + 1;
        }
        printNumber(n);
        printf(", ");
    }
    printf("\n");
}

void callCollatz(){
    printf("Calling Collatz\n");
    for(int i = 1; i <= 7; i++){
        collatz(i);
    }
    exit();
}
```

By changing the parameters of the functions it is possible to test with different values.

```c
int binarySearch(int low, int high, int arr[], int x){
    if(low > high){
        return -1;
    }
    int mid = (low + high) / 2;
    if(arr[mid] == x){
        return mid;
    }
    else if(arr[mid] > x){
        return binarySearch(low, mid - 1, arr, x);
    }
    else{
        return binarySearch(mid + 1, high, arr, x);
    }
}

void callBinarySearch(){
    printf("Calling Binary Search ");
    int arr[] = {10, 20, 80, 30, 60, 50, 100, 110, 130, 170};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Input array: [");
    for(int i = 0; i < n; i++){
        printNumber(arr[i]);
        if(i != n-1)
            printf(", ");
    }
    printf("]");
    printf("\n");
    int x = 110;
    printf("Search for: ");
    printNumber(x);
    printf("\n");
    int index = -1;
    index = binarySearch(0, n-1, arr, x);
    printf("Output: ");
    printNumber(index);
    printf("\n");
    exit();
}
```

```c
int linearSearch(int arr[], int x, int n){
    for(int i = 0; i < n; i++){
        if(arr[i] == x){
            return i;
        }
    }
    return -1;
}

void callLinearSearch(){
    printf("Calling Linear Search\n");
    int arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Input array: [");
    for(int i = 0; i < n; i++){
        printNumber(arr[i]);
        if(i != n-1)
            printf(", ");
    }
    printf("]: ");
    printf("\n");
    int x = 175;
    printf("Search for: ");
    printNumber(x);
    printf("\n");
    int index = -1;
    index = linearSearch(arr, x, n);
    printf("Output: ");
    printNumber(index);
    printf("\n");
    exit();
}
```

For binary search I sent the array as ordered since binary search works for sorted arrays.

- ## Testing the system calls

```
void singlefork(){
    int pid = getPid();
    fork(&pid);
    printf("After fork: ");
    printNumber(pid);
    printf("\n");
    if(pid == 0){
        printf("Child PID: ");
        printNumber(getPid());
        printf("\n");
        exec(call_long_running_program);
    }
    else{
        printf("Parent PID: ");
        printNumber(getPid());
        printf("\n");
        waitpid(pid);
        exit();
    }
    while(1);
}
```

I have created a function inside kernel.cpp to test fork, exec, waitpid and exit system calls.

```
Task singlefork1(&gdt, singlefork, myos::Priority::LOW);
taskManager.AddTask(&singlefork1);
```

Inside kernel main I have added this function as a task and loaded it to the taskManager.

```
Welcome to My OS
************** PROCESS TABLE **************
PID        PPID        STATE        PRIORITY
1           0         READY         LOW

*******************************************
************** PROCESS TABLE **************
PID        PPID        STATE        PRIORITY
1           0         RUNNING        LOW

2           1         READY         LOW

*******************************************
After fork: 0
Child PID: 2
Calling Long Running Program
```

When the program starts running, there is only one task until the fork system call is invoked. After the fork system call, two tasks exist: the parent and the child. The fork function differentiates between these two tasks by returning the child's PID to the parent and 0 to the child. Upon receiving the return value from fork, the child and parent tasks execute different instructions. The child task uses the exec function to replace its current program with a new one (call_long_running_program), and begins executing that function. Meanwhile, the parent task invokes waitpid to wait for the child task to finish. After the child task has finished executing its program and prints the result, it invokes exit to terminate itself. Once the child has terminated, the parent task also terminates by invoking exit. The scheduling of these tasks, including the transition from the parent task to the child task and the termination of tasks, is managed by the Schedule function mentioned earlier.

- ## Part A First Strategy

```c
void lifecyclePartA(){
    int pid = getPid();
    fork(&pid);
    printf("After fork1: ");
    printNumber(pid);
    printf("\n");
    if(pid == 0){
        // printf("Child PID: ");
        // printNumber(getPid());
        // printf("\n");
        exec(callCollatz);
    }
    else{
        // printf("Parent PID: ");
        // printNumber(getPid());
        // printf("\n");
        int pid2 = getPid();
        fork(&pid2);
        printf("After fork2: ");
        printNumber(pid2);
        printf("\n");
        if(pid2 == 0){
            // printf("Child PID: ");
            // printNumber(getPid());
            // printf("\n");
            exec(callCollatz);
        }
        else{
            // printf("Parent PID: ");
            // printNumber(getPid());
            // printf("\n");
            int pid3 = getPid();
            fork(&pid3);
            printf("After fork3: ");
            printNumber(pid3);
            printf("\n");
            if(pid3 == 0){
                // printf("Child PID: ");
                // printNumber(getPid());
                // printf("\n");
                exec(callCollatz);
```

```c
            else{
                // printf("Parent PID: ");
                // printNumber(getPid());
                // printf("\n");
                int pid4 = getPid();
                fork(&pid4);
                printf("After fork4: ");
                printNumber(pid4);
                printf("\n");
                if (pid4 == 0) {
                    // printf("Child PID: ");
                    // printNumber(getPid());
                    // printf("\n");
                    exec(call_long_running_program);
                }
                else {
                    // printf("Parent PID: ");
                    // printNumber(getPid());
                    // printf("\n");
                    int pid5 = getPid();
                    fork(&pid5);
                    printf("After fork5: ");
                    printNumber(pid5);
                    printf("\n");
                    if (pid5 == 0) {
                        // printf("Child PID: ");
                        // printNumber(getPid());
                        // printf("\n");
                        exec(call_long_running_program);
                    }
                    else {
                        // printf("Parent PID: ");
                        // printNumber(getPid());
                        // printf("\n");
                        int pid6 = getPid();
                        fork(&pid6);
                        printf("After fork6: ");
                        printNumber(pid6);
                        printf("\n");
                        if (pid6 == 0) {
```

```c
else {
    // printf("Parent PID: ");
    // printNumber(getPid());
    // printf("\n");
    // while(1);
    waitpid(pid6);
    exit();
}
```

Lifecycle of part A loads collatz and long running program 3 times by utilizing the fork system call.

```
Welcome to My OS
After fork1: 0
Calling Collatz
Collatz 1:
Collatz 2: 1,
Collatz 3: 10, 5, 16, 8, 4, 2, 1,
Collatz 4: 2, 1,
After fork1: 2
After fork2: 0
Calling Collatz
Collatz 1:
Collatz 2: 1,
Collatz 3: 10, 5, 16, 8, 4, 2, 1,
Collatz 4: 2, 1,
After fork2: 3
After fork3: 0
Calling Collatz
Collatz 1:
Collatz 2: 1,
Collatz 3: 10, 5, 16, 8, 4, 2, 1,
Collatz 4: 2, 1,
After fork3: 4
After fork4: 0
Calling Long Running Program
```

```
After fork5: 0
Calling Long Running Program
Result: 857419840
After fork5: 6
After fork6: 0
Calling Long Running Program
Result: 857419840
After fork6: 7
Task already terminated
```

In this lifecycle both the parent and child prints the “after fork” part after they execute the fork. And it can be clearly seen that for child processes which executes the programs returns 0 from the fork and the parent returns the pid of the corresponding child. Child processes use exec system call to execute the programs for part A which are collatz and long running program. I did not take any input from the user at this part and gave the input as parameter to the functions.

- ## Part B First Strategy

```
int32_t rand(int min, int max) {
    uint64_t counter;
    int32_t num;
    asm("rdtsc": "=A"(counter)); // Read the Time Stamp Counter

    counter = counter * 1103515245 + 12345;
    num = (int)(counter / 65536) % (max - min);
    if (num < 0)
        num += max;
    return num + min;
}

void firstStrategyPartB(){
    int forkCount = 10;
    int processCount = 4;

    void (*processes[processCount])() = {callCollatz, call_long_running_program, callBinarySearch, callLinearSearch};

    int randomProcess = rand(0, processCount);
    printf("Random Process: ");
    printNumber(randomProcess);
    printf("\n");

    for(int i = 0; i < forkCount; i++){
        int pid = getPid();
        fork(&pid);
        printf("After fork: ");
        printNumber(pid);
        printf("\n");
        if(pid == 0){
            exec(processes[randomProcess]);
        }
    }
    while(1);
}
```

For part B I made a function that returns a random value between the min and max values to have the randomness effect when choosing the program the load. First strategy is randomly choosing one of the programs and loading it 10 times. Since we have to do fork in case of creating processes I have set a loop that will create a task by utilizing fork system call and executing the random process.

```
Welcome to My OS
Calling Binary Search Input array: [10, 20, 80, 30, 60, 50, 100, 110, 130, 170]
Search for: 110 Output: 7
Calling Binary Search Input array: [10, 20, 80, 30, 60, 50, 100, 110, 130, 170]
Search for: 110 Output: 7
Calling Binary Search Input array: [10, 20, 80, 30, 60, 50, 100, 110, 130, 170]
Search for: 110 Output: 7
Calling Binary Search Input array: [10, 20, 80, 30, 60, 50, 100, 110, 130, 170]
Search for: 110 Output: 7
Calling Binary Search Input array: [10, 20, 80, 30, 60, 50, 100, 110, 130, 170]
Search for: 110 Output: 7
Calling Binary Search Input array: [10, 20, 80, 30, 60, 50, 100, 110, 130, 170]
Search for: 110 Output: 7
Calling Binary Search Input array: [10, 20, 80, 30, 60, 50, 100, 110, 130, 170]
Search for: 110 Output: 7
Calling Binary Search Input array: [10, 20, 80, 30, 60, 50, 100, 110, 130, 170]
Search for: 110 Output: 7
Calling Binary Search Input array: [10, 20, 80, 30, 60, 50, 100, 110, 130, 170]
Search for: 110 Output: 7
Calling Binary Search Input array: [10, 20, 80, 30, 60, 50, 100, 110, 130, 170]
Search for: 110 Output: 7
```

It can be seen that 10 same program has worked as child processes by using fork system call. In this strategy I also did not include any input from the user and sent the input as an argument to the function. Also I have not included PrintProcessTable() function here.

```
*****************************************
Task PID inside constructor: 2
************** PROCESS TABLE **************
PID      PPID        STATE       PRIORITY
1         0        RUNNING         LOW
2         1        READY        LOW
*****************************************
Calling Linear Search
Input array: [10, 20, 80, 30, 60, 50, 110, 100, 130, 170]:
Search for: 175
Output: -1
************** PROCESS TABLE **************
PID      PPID        STATE       PRIORITY
1         0        READY        LOW
2         1        TERMINATED        LOW
*****************************************
```

In order to show the context switches and the process table I have included the PrintProcessTable() function inside Schedule and it can be seen that during context switching the state of the processes changes and after child has finished executing its state is set to terminated

- # Part B Second Strategy

```c
void secondStrategyPartB(){
    int forkCount = 3;
    int processCount = 4;

    void (*processes[processCount])() = {callCollatz, call_long_running_program, callBinarySearch, callLinearSearch};

    // Select two different programs
    int program1 = rand(0, processCount);
    int program2;
    do{
        program2 = rand(0, processCount);
    }while(program1 == program2);

    void (*selectedProcesses[2])() = {processes[program1], processes[program2]};

    for(int i=0; i<forkCount; i++){
        int pid = getPid();
        fork(&pid);
        if(pid == 0){
            exec(selectedProcesses[0]);
        }
    }

    for(int i=0; i<forkCount; i++){
        int pid2 = getPid();
        fork(&pid2);
        if(pid2 == 0){
            exec(selectedProcesses[1]);
        }
    }
    while(1);
}
```

This strategy also uses “rand” function that will have the randomness for choosing two different programs and loading each of them 3 times using fork system call.

```
Welcome to My OS
Calling Linear Search
Input array: [10, 20, 80, 30, 60, 50, 110, 100, 130, 170]:
Search for: 175
Output: -1
Calling Linear Search
Input array: [10, 20, 80, 30, 60, 50, 110, 100, 130, 170]:
Search for: 175
Output: -1
Calling Linear Search
Input array: [10, 20, 80, 30, 60, 50, 110, 100, 130, 170]:
Search for: 175
Output: -1
Calling Binary Search Input array: [10, 20, 80, 30, 60, 50, 100, 110, 130, 170]
Search for: 110
Output: 7
Calling Binary Search Input array: [10, 20, 80, 30, 60, 50, 100, 110, 130, 170]
Search for: 110
Output: 7
Calling Binary Search Input array: [10, 20, 80, 30, 60, 50, 100, 110, 130, 170]
Search for: 110
Output: 7
```

Each program loaded 3 times and executed by using fork, exec and exit system calls. Exit() system call is inside the functions for the algorithms.

- **Part B Third Strategy**

```
void initTask(){
    printf("Init Task Started\n");
    while(true);
}
```

This is the init task that runs on infinite loop.

```
void thirdStrategyPartB(){
    int pid = -1;
    fork(&pid);
    if(pid == 0){
        exec(callCollatz);
    }
    else{
        int pid2 = -1;
        fork(&pid2);
        if(pid2 == 0){
            exec(call_long_running_program);
        }
        else{
            int pid3 = -1;
            fork(&pid3);
            if(pid3 == 0){
                exec(callBinarySearch);
            }
            else{
                int pid4 = -1;
                fork(&pid4);
                if(pid4 == 0){
                    exec(callLinearSearch);
                }
                else{
                    while(1);
                }
            }
        }
    }
}
```

This is the other task that runs the other programs.

```
Welcome to My OS
Init Task Started
Forked task PID: 3
Calling Collatz
Collatz 1:
Collatz 2: 1,
Collatz 3: 10, 5, 16, 8, 4, 2, 1,
Collatz 4: 2, 1,
Forked task PID: 4
Calling Long Running Program
Result: 392146832
Forked task PID: 5
Calling Binary Search Input array: [10, 20, 80, 30, 60, 50, 100, 110, 130, 170]
Search for: 110
Output: 7
Forked task PID: 6
Calling Linear Search
Input array: [10, 20, 80, 30, 60, 50, 110, 100, 130, 170]:
Search for: 175
Output: -1
```

First init task starts running and then the collatz program starts running by using fork system call. Finally the rest of the programs are created by using fork and their PID's can be seen in the terminal and they use exec system call to execute the programs. At the PDF file the array for binary search algorithm wasn't sorted so I sort it and sent that to binary search.