GTU Department of Computer Engineering

CSE 312 – Spring 2024

HW 1 Report

Barış Batuhan Bolat

210104004029

HW 1 Part A

- **syscalls.cpp**
  - **getPid() :** Gets the process ID
  - **waitpid() :** Waits for a child process with the given ID
  - **sys_exit() :** Exits the current process
  - **fork() :** Creates a child process (without returning the child's PID)
  - **fork(int *pid) :** Creates a child process and returns the child's PID in the provided pointer
  - **exec(void entrypoint()) :** Executes and replaces curren task.
  - **addTask(void entrypoint()) :** Adds a new task to the system
  - **printf :** Prints the desired char sequence to screen**.** Implementation is in **kernel.cpp**.

```
// Gets the process ID
int myos::getPid()
{
    int pId = 1;
    asm("int $0x80" : "=c"(pId) : "a"(SYSCALLS::GETPID));
    return pId;
}
// Waits for a child process with the given ID
void myos::waitpid(common::uint8_t wPid)
{
    asm("int $0x80" : : "a"(SYSCALLS::WAITPID), "b"(wPid));
}
// Exits the current process
void myos::sys_exit()
{
    asm("int $0x80" : : "a"(SYSCALLS::EXIT));
}
// Creates a child process (without returning the child's PID)
void myos::fork()
{
    asm("int $0x80" : : "a"(SYSCALLS::FORK));
}
// Creates a child process and returns the child's PID in the provided pointer
void myos::fork(int *pid)
{
    asm("int $0x80" : "=c"(*pid) : "a"(SYSCALLS::FORK));
}

int myos::exec(void entrypoint())
{
    int result;
    asm("int $0x80" : "=c"(result) : "a"(SYSCALLS::EXEC), "b"((uint32_t)entrypoint));
    return result;
}
// Adds a new task to the system
int myos::addTask(void entrypoint())
{
    int result;
    asm("int $0x80" : "=c"(result) : "a"(SYSCALLS::ADDTASK), "b"((uint32_t)entrypoint));
    return result;
}
```

```
enum SYSCALLS
{
    EXIT,GETPID,WAITPID,FORK,EXEC,PRINTF,ADDTASK
};
```

- ## Syscalls HandleInterrupt

  Acts as a bridge between user-space system call requests and the kernel's system call implementation a.k.a interrupts.

```
uint32_t SyscallHandler::HandleInterrupt(uint32_t esp)
{
    // Get a pointer to the CPU state structure from the stack pointerr
    CPUState *cpu = (CPUState *)esp;
    // Switch statement to handle different system calls based on the value in cpu->eax
    switch (cpu->eax)
    {
    case SYSCALLS::EXEC:
        esp = InterruptHandler::sys_exec(cpu->ebx);
        break;
    case SYSCALLS::FORK:
        cpu->ecx = InterruptHandler::sys_fork(cpu);
        return InterruptHandler::HandleInterrupt(esp);
        break;
    case SYSCALLS::PRINTF:
        printf((char *)cpu->ebx);
        break;
    case SYSCALLS::EXIT:
        if (InterruptHandler::sys_exit())
        {
            return InterruptHandler::HandleInterrupt(esp);
        }
        break;
    case SYSCALLS::WAITPID:
        if (InterruptHandler::sys_waitpid(esp))
        {
            return InterruptHandler::HandleInterrupt(esp);
        }
    case SYSCALLS::GETPID:
        cpu->ecx = InterruptHandler::sys_getpid();
        break;
    case SYSCALLS::ADDTASK:
        cpu->ecx = InterruptHandler::sys_addTask(cpu->ebx);
        break;
    default:
        break;
    }
    return esp;
}
```

- ## interrupts.cpp(Changes)
  - **sys_getpid() :** Gets the process ID of the current task by calling task managers **GetPId.**
  - **sys_exec(common::uint32_t entrypoint) :** Executes a new task by calling task managers **ExecTask.**
  - **sys_addTask(common::uint32_t entrypoint) :** Adds a new task by calling task managers **AddTask.**
  - **sys_fork(CPUState *cpustate) :** Forks the current task by calling task managers **ForkTask**.
  - **sys_exit() :** Exits the current task by calling task managers **ExitCurrentTask**.
  - **sys_waitpid(common::uint32_t pid) :** Wait for a child task to exit by calling task managers **WaitTask.**

```
common::uint32_t InterruptHandler::sys_getpid()
{
    return interruptManager->taskManager->GetPId();
}

common::uint32_t InterruptHandler::sys_exec(common::uint32_t entrypoint)
{
    return interruptManager->taskManager->ExecTask((void (*)())entrypoint);
}
common::uint32_t InterruptHandler::sys_addTask(common::uint32_t entrypoint)
{
    return interruptManager->taskManager->AddTask((void (*)())entrypoint);
}
common::uint32_t InterruptHandler::sys_fork(CPUState *cpustate)
{
    return interruptManager->taskManager->ForkTask(cpustate);
}
bool InterruptHandler::sys_exit()
{
    return interruptManager->taskManager->ExitCurrentTask();
}
bool InterruptHandler::sys_waitpid(common::uint32_t pid)
{
    return interruptManager->taskManager->WaitTask(pid);
}
```

- Interrupt handlers **HandleInterrupt** function calls task manager scheduler function **Schedule**.

```
uint32_t InterruptHandler::HandleInterrupt(uint32_t esp)
{
    return (uint32_t)interruptManager->taskManager->Schedule((CPUState *)esp);
}
```

- ## multitasking.cpp
  - **ForkTask(CPUState *cpustate) :** ForkTask creates a new process.
    - It checks for available slots and copies the parent task's state.
    - Increases pId by using static pIdCounter variable.
    - It copies current tasks stack to new created one.
    - On success, it returns the new task's process ID (pId).

```cpp
common::uint32_t TaskManager::ForkTask(CPUState *cpustate)
{
    if (taskCount >= 256)
    {
        printf("There are 256 processes are active. No more can be created.\n");
        return 0;
    }
    tasks[taskCount].taskState = READY;
    tasks[taskCount].pPId = tasks[currentTask].pId;
    tasks[taskCount].pId = (Task::pIdCounter)++;

    // Copy the parent task's stack to the new task's stack
    for (int i = 0; i < sizeof(tasks[currentTask].stack); i++)
    {
        tasks[taskCount].stack[i] = tasks[currentTask].stack[i];
    }

    // Calculate the offset of the CPU state within the parent task's stack
    common::uint32_t currentTaskOffset = (((common::uint32_t)cpustate - (common::uint32_t)tasks[currentTask].stack));

    // Set the new task's CPU state pointer based on the offset and its own stack
    tasks[taskCount].cpustate = (CPUState *)(((common::uint32_t)tasks[taskCount].stack) + currentTaskOffset);

    // Initialize specific CPU register (ECX) to 0
    tasks[taskCount].cpustate->ecx = 0;
    taskCount++;

    // Set the new task's time slice (timeRemaining)
    tasks[taskCount].timeRemaining = timeQuantum;
    return tasks[taskCount - 1].pId;
}
```

  - **AddTask(Task *task) :** Adds a new task with the provided task object.
    - It checks for available slots and copies the state, cpu registers, program counter and specific flags from the provided task.
    - Sets the time of the new task with timeQuantum.
    - On success, it returns true.

```cpp
bool TaskManager::AddTask(Task *task)
{
    if (taskCount >= 256)
        return false;
    tasks[taskCount].taskState = READY;
    tasks[taskCount].pId = task->pId;
    // Set the new task's CPU state pointer to the top of its stack with an offset for CPUState size
    tasks[taskCount].cpustate = (CPUState *)(tasks[taskCount].stack + 4096 - sizeof(CPUState));

    // Copy specific CPU registers (EAX, EBX, ECX, EDX) from the provided task
    tasks[taskCount].cpustate->eax = task->cpustate->eax;
    tasks[taskCount].cpustate->ebx = task->cpustate->ebx;
    tasks[taskCount].cpustate->ecx = task->cpustate->ecx;
    tasks[taskCount].cpustate->edx = task->cpustate->edx;

    // Copy additional CPU registers (ESI, EDI, EBP) from the provided task
    tasks[taskCount].cpustate->esi = task->cpustate->esi;
    tasks[taskCount].cpustate->edi = task->cpustate->edi;
    tasks[taskCount].cpustate->ebp = task->cpustate->ebp;

    // Copy program counter (EIP) and code segment (CS) from the provided task
    tasks[taskCount].cpustate->eip = task->cpustate->eip;
    tasks[taskCount].cpustate->cs = task->cpustate->cs;

    // Copy status flags (EFLAGS) from the provided task
    tasks[taskCount].cpustate->eflags = task->cpustate->eflags;
    // Set the new task's time slice (timeRemaining)
    tasks[taskCount].timeRemaining = timeQuantum;
    taskCount++;
    return true;
}
```

- **AddTask(void entrypoint()) :** Adds a new task with the provided task object.
    - It checks for available slots and copies the state, cpu registers, program counter and specific flags from the provided entry point function.
    - Sets the time of the new task with timeQuantum.
    - On success, it returns true.

```cpp
common::uint32_t TaskManager::AddTask(void entrypoint())
{
    tasks[taskCount].taskState = READY;
    tasks[taskCount].pId = (Task::pIdCounter)++;
    // Set the new task's CPU state pointer to the top of its stack with an offset for CPUState size
    tasks[taskCount].cpustate = (CPUState *)(tasks[taskCount].stack + 4096 - sizeof(CPUState));

    // Initialize specific CPU registers (EAX, EBX, ECX, EDX) to 0
    tasks[taskCount].cpustate->eax = 0;
    tasks[taskCount].cpustate->ebx = 0;
    tasks[taskCount].cpustate->ecx = 0;
    tasks[taskCount].cpustate->edx = 0;

    // Initialize additional CPU registers (ESI, EDI, EBP) to 0
    tasks[taskCount].cpustate->esi = 0;
    tasks[taskCount].cpustate->edi = 0;
    tasks[taskCount].cpustate->ebp = 0;

    // Set the program counter (EIP) to the entry point function address
    tasks[taskCount].cpustate->eip = (uint32_t)entrypoint;
    // Set the code segment selector from the global descriptor table (gdt)
    tasks[taskCount].cpustate->cs = gdt->CodeSegmentSelector();
    // Set specific status flags (EFLAGS)
    tasks[taskCount].cpustate->eflags = 0x202;
    taskCount++;
    tasks[taskCount].timeRemaining = timeQuantum;
    return tasks[taskCount - 1].pId;
}
```

- **WaitTask(common::uint32_t esp) :** Suspends the current task until another task with the specified process ID (pid) finishes.
    - It prevents a task from waiting for itself and checks for invalid targets or finished tasks.
    - Set the current task's CPU state pointer
    - Set the current task's state to WAITING.
    - Set the process ID to wait.
    - On success, it returns true.

```cpp
bool TaskManager::WaitTask(common::uint32_t esp)
{
    CPUState *cpustate = (CPUState *)esp;
    // Extract the process ID to wait for from a CPU register (EBX)
    common::uint32_t pid = cpustate->ebx;
    if (tasks[currentTask].pId == pid || pid == 0) // prevention self waiting
        return false;
    int index = getIndex(pid);
    // Check if the target task exists and isn't already finished
    if (index == -1)
        return false;
    if (taskCount <= index || tasks[index].taskState == FINISHED)
        return false;
    tasks[currentTask].cpustate = cpustate;
    tasks[currentTask].waitPid = pid;
    tasks[currentTask].taskState = WAITING;
    return true;
}
```

- **ExecTask(void entrypoint()) :** Replaces the current task's state with the provided entry point function(Task).
    - Set the current task's CPU state pointer to the top of its stack with an offset for CPUState size.
    - Initialize specific and additional CPU registers, program counter, code segment selector and flags.
    - Returns a pointer to the CPU state for context switching.

```
common::uint32_t TaskManager::ExecTask(void entrypoint())
{
    tasks[currentTask].taskState - READY;
    // Set the current task's CPU state pointer to the top of its stack with an offset for CPUState size
    tasks[currentTask].cpustate = (CPUState *)(tasks[currentTask].stack + 4096 - sizeof(CPUState));

    // Initialize specific CPU registers (EAX, EBX, ECX, EDX) to 0
    tasks[currentTask].cpustate->eax = 0;
    tasks[currentTask].cpustate->ebx = 0;
    tasks[currentTask].cpustate->ecx = tasks[currentTask].pId;
    tasks[currentTask].cpustate->edx = 0;

    // Initialize additional CPU registers (ESI, EDI, EBP) to 0
    tasks[currentTask].cpustate->esi = 0;
    tasks[currentTask].cpustate->edi = 0;
    tasks[currentTask].cpustate->ebp = 0;

    // Set the program counter (EIP) to the entry point function address
    tasks[currentTask].cpustate->eip = (uint32_t)entrypoint;
    // Set the code segment selector from the global descriptor table (gdt)
    tasks[currentTask].cpustate->cs = gdt->CodeSegmentSelector();
    // Set specific status flags (EFLAGS)
    tasks[currentTask].cpustate->eflags = 0x202;
    return (uint32_t)tasks[currentTask].cpustate;
}
```

- **ExitCurrentTask() :** Marks the current task as finished.
    - Prints process table when task is ended.

```
bool TaskManager::ExitCurrentTask()
{
    tasks[currentTask].taskState = FINISHED;
    PrintProcessTable();
    return true;
}
```

- **GetPID():** Returns the process ID of the current task.

```
common::uint32_t TaskManager::GetPID()
{
    return tasks[currentTask].pId;
}
```

- **getIndex(common::uint32_t pid) :** Finds the index of a task based on its process ID (pId).

```
int TaskManager::getIndex(common::uint32_t pid)
{
    int index = -1;
    for (int i = 0; i < taskCount; i++)
    {
        if (tasks[i].pId == pid)
        {
            index = i;
            break;
        }
    }
    return index;
}
```

- **PrintProcessTable :** Prints process table with their pid, parent pid and their states

```
void TaskManager::PrintProcessTable()
{
    printf("\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");
    printf("PID\tPPID\tSTATE\n");
    for (int i = 1; i < taskCount; i++)
    {
        printNum(tasks[i].pId);
        printf("\t  ");
        printNum(tasks[i].pPId);
        printf("\t  ");
        if (tasks[i].taskState == TaskState::READY)
        {
            if (i == currentTask)
                printf("RUNNING");
            else
                printf("READY");
        }
        else if (tasks[i].taskState == TaskState::WAITING)
        {
            printf("WAITING");
        }
        else if (tasks[i].taskState == TaskState::FINISHED)
        {
            printf("FINISHED");
        }
        printf("\n");
    }
```

- **Scheduler :** Round-robin scheduling algorithm for tasks. It selects the next task to run based on the following logic:

  - **Check for tasks:** It first verifies if there are any tasks available for scheduling (checks task count). If not, it simply returns the current CPU state.
  - **Save current task state:** If tasks exist, it saves the CPU state of the currently running task (if any).
  - **Initialize next task search:** It starts searching for the next ready task by setting an index to the task after the current one (wrapping around if needed).
  - **Loop for ready task:** It iterates through tasks until a task in the "**READY**" state is found.
    - **Waiting tasks:** During this loop, it checks for tasks waiting on another task to finish (state: WAITING). If a waiting task is encountered, it searches for the task it's waiting on (using waitPid).
      - **Waited-on task finished:** If the waited-on task is finished, the waiting task's state is updated to "**READY**" and it is removed from the waiting list.
    - **Wrap around:** The search for a ready task is circular, meaning it wraps back to the beginning of the task list if no ready tasks are found after iterating through all tasks.
    - **Break on loop back:** The loop breaks if it reaches the current task index again, indicating no ready tasks were found.
  - **Handle current task time:** If the current task was in a "**READY**" state, it reduces its remaining time quantum.
    - **Time quantum expired:** If the time quantum reaches zero, the current task's time is reset, and the scheduler moves to the next task.
  - **Move to next task (default):** If the current task wasn't ready or its time quantum didn't expire, the scheduler simply moves to the next task.
  - **Final loop for ready task (optional):** In case the initial loop for a ready task didn't find one due to waiting tasks becoming ready later, it does another loop with the same logic as step 4.
  - **Update current task and return:** Finally, it updates the current task index to the one found ready and returns the CPU state of the newly selected task for context switching.

```cpp
CPUState *TaskManager::Schedule(CPUState *cpustate)
{
    // Check if there are no tasks to schedule
    if (taskCount <= 0)
    {
        return cpustate;
    }

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

    // Initialize the next task index to check
    int findTask = (currentTask + 1) % taskCount;
    // Loop until a ready task is found
    while (tasks[findTask].taskState != READY)
    {
        // Check for tasks waiting on another task to finish
        if (tasks[findTask].taskState == WAITING && tasks[findTask].waitPid > 0)
        {
            int waitTaskIndex = getIndex(tasks[findTask].waitPid);

            // If the waited-on task is done, update waiting task and continue
            if (waitTaskIndex > -1 && tasks[waitTaskIndex].taskState != WAITING)
            {
                if (tasks[waitTaskIndex].taskState == FINISHED)
                {
                    tasks[findTask].waitPid = 0;
                    tasks[findTask].taskState = READY;
                }
            }
        }

        // Move to the next task (circular fashion)
        findTask = (findTask + 1) % taskCount;
        // Break if we looped back to the current task (no ready tasks)
        if (findTask == currentTask)
        {
            break;
        }
    }
```

```cpp
    if (tasks[currentTask].taskState == READY)
    {
        // If the task finishes its time quantum, reset and move to next task
        tasks[currentTask].timeRemaining -= timeQuantum;
        if (tasks[currentTask].timeRemaining <= 0)
        {
            tasks[currentTask].timeRemaining = timeQuantum;
            findTask = (currentTask + 1) % taskCount;
        }
    }
    // If current task wasn't ready, just move to the next one
    else
    {
        findTask = (currentTask + 1) % taskCount;
    }

    // Loop again until a ready task is found (in case previous loop didn't)
    while (tasks[findTask].taskState != READY)
    {
        if (tasks[findTask].taskState == WAITING && tasks[findTask].waitPid > 0)
        {
            int waitTaskIndex = getIndex(tasks[findTask].waitPid);
            if (waitTaskIndex > -1 && tasks[waitTaskIndex].taskState != WAITING)
            {
                if (tasks[waitTaskIndex].taskState == FINISHED)
                {
                    tasks[findTask].waitPid = 0;
                    tasks[findTask].taskState = READY;
                }
            }
        }
        findTask = (findTask + 1) % taskCount;
        if (findTask == currentTask)
        {
            break;
        }
    }
    // Update the current task index and return its CPU state
    currentTask = findTask;
    return tasks[currentTask].cpustate;
}
```

- ## kernel.cpp
  - **printNum :** Converts the provided integer to char sequence and prints.

```
void printNum(int num)
{
    char numberStr[10];
    itoa(num, numberStr, 10);
    printf(numberStr);
}
```

  - **printf :** Prints the provided char sequence to the screen

```
void printf(char *str)
{
    static uint16_t *VideoMemory = (uint16_t *)0xb8000;

    static uint8_t x = 0, y = 0;

    for (int i = 0; str[i] != '\0'; ++i)
    {
        switch (str[i])
        {
        case '\n':
            x = 0;
            y++;
            break;
        default:
            VideoMemory[80 * y + x] = (VideoMemory[80 * y + x] & 0xFF00) | str[i];
            x++;
            break;
        }

        if (x >= 80)
        {
            x = 0;
            y++;
        }

        if (y >= 25)
        {
            for (y = 0; y < 25; y++)
                for (x = 0; x < 80; x++)
                    VideoMemory[80 * y + x] = (VideoMemory[80 * y + x] & 0xFF00) | ' ';
            x = 0;
            y = 0;
        }
    }
}
```

  - **printArray :** Prints the array with n integer elements to the screen.

```
void printArray(int arr[], int n)
{
    printf("Array : {");
    for (int i = 0; i < n; i++)
    {
        printNum(arr[i]);
        if (i + 1 != n)
        {
            printf(",");
        }
    }
    printf("} ");
}
```

- **forExample**
  - The code utilizes a loop to call `fork` six times, each time storing the returned value in a corresponding `pid` variable.
  - The return value of `fork` determines the execution path:

  - **Parent Process (return value > 0):**
    - The parent process continues execution after the **fork** call.
    - It uses **waitpid** to wait for the child process (identified by `pid`) to finish before proceeding.
  - **Child Process (return value == 0):**
    - The child process executes the code within the if block.
    - It prints a message indicating its purpose using **printf**.
    - It calls custom functions like **TaskCollatz1**, **TaskCollatz2**, **TaskCollatz3**, and **long_running_program**. These functions likely perform specific tasks.
    - Finally, the child process exits using **sys_exit**.

```c
void forkExample()
{
    int pid1 = 0;
    int pid2 = 0;
    int pid3 = 0;
    int pid4 = 0;
    int pid5 = 0;
    int pid6 = 0;

    int parentPid = getPid();
    printf("Task Pid:");
    printNum(parentPid);
    printf("\n");

    fork(&pid1);
    if (pid1 == 0)
    {
        printf("Collatz_1 : ");
        TaskCollatz1();
        sys_exit();
    }
    waitpid(pid1);

    fork(&pid2);
    if (pid2 == 0)
    {
        printf("Collatz_2 : ");
        TaskCollatz2();
        sys_exit();
    }
    waitpid(pid2);

    fork(&pid3);
    if (pid3 == 0)
    {
        printf("Collatz_3 : ");
        TaskCollatz3();
        sys_exit();
    }
    waitpid(pid3);

    fork(&pid4);
    if (pid4 == 0)
    {
        printf("LongRunning_10000 : ");
        printNum(long_running_program(14000));
        printf("\n");
        sys_exit();
    }
    waitpid(pid4);

    fork(&pid5);
    if (pid5 == 0)
    {
        printf("LongRunning_20000 : ");
        printNum(long_running_program(18000));
        printf("\n");
        sys_exit();
    }
    waitpid(pid5);

    fork(&pid6);
    if (pid6 == 0)
    {
        printf("LongRunning_15000 : ");
        printNum(long_running_program(16000));
        printf("\n");
        sys_exit();
    }
    waitpid(pid6);
    sys_exit();
}
```

- **kernelMain (Changes)**
  - **Task task1(&gdt, EmptyTask);:** Creates a Task object named task1 using the GDT and assigns it the EmptyTask function. This is a igniter task. It must be added to run next tasks.
  - **Task task2(&gdt, forkExample);:** Creates a Task object named task2 using the GDT and assigns it the forkExample function.(This function explained above)
  - taskManager.AddTask(&task1);: Adds task1 to the task manager.
  - taskManager.AddTask(&task2);: Adds task2 to the task manager.

```
TaskManager taskManager;

Task task1(&gdt, EmptyTask);
Task task2(&gdt, forkExample);

taskManager.AddTask(&task1);
taskManager.AddTask(&task2);
```

System Enhancements

- **System Call Implementations**
  - New System Call Functions:
    - sys_getpid(), sys_exec(), sys_addTask(), sys_fork(), sys_exit(), sys_waitpid().
  - Handling System Calls:
    - The HandleInterrupt function is used to interrupt calls.
- **Multitasking Enhancements**
  - Task Management:
    - Added pIdCounter for process IDs, taskState for managing task states, and waitPid for tracking wait conditions.
    - ForkTask() duplicates the current task; AddTask() initializes new tasks.
  - Round Robin Scheduling:
    - Implemented in Schedule(), cycling through tasks based on state and timeQuantum.
  - PrintProcessTable() method prints task states.
- **Kernel Improvements**
  - Initialization and Task Management:
    - TaskManager manages tasks; forkExample initializes tasks and processes.
  - Interrupt Management

Implementation Strategies

- System Call Implementation:
  - Added system call handling methods and integrated them within the
  - InterruptManager.
- Multitasking Enhancements:
  - Improved task management and scheduling.
- Kernel Improvements:
  - Added support for multiple tasks and system calls.
  - Implemented priority-based task switching

General Flow

1. kernel.cpp uses system calls from syscalls.h
2. syscalls.cpp calls interrupts.cpp functions
3. interrupts.cpp uses multitasking.cpp functions for system calls and scheduler.