GEBZE TECHNİCAL UNIVERSITY  
  
CSE312

## Operating Systems

# Homework 1 Report

### How to Run?

Open the terminal and navigate to the source where the makefile directory. Then, compile the program by typing “**make**” and the .iso file will be created. Then construct the virtual machine on VirtualBox Program with connecting this file. Once executed, the program will produce output as specified by strategy 1 which is asked for in the pdf file. To clean up generated files, use “**make clean**”, which will remove the files generated.

### Syscall Implementations

The system call implementation uses the **SyscallHandler** class, derived from InterruptHandler, to manage system call interrupts via the IDT. System calls are invoked using a software interrupt (int 0x80), with the **SyscallHandler** dispatching these calls based on the value in the eax register. This design treats system calls as a special type of interrupt, allowing user programs to safely request kernel services. And provided source code was already have an interrupt mechanism with 3 structure:

* Interrupt Descriptor Table (IDT)

Each entry in the IDT points to an interrupt service routine (ISR). The **SetInterruptDescriptorTableEntry** function is used to set up individual entries in the IDT and initialized and connected to the appropriate routines within the constructor of the **InterruptManager** class. After setting up the IDT entries, the IDT is loaded into the CPU using the **lidt** instruction. This instruction tells the CPU where to find the IDT.

* Interrupt Manager Class

The **InterruptManager** class is responsible for setting up the IDT, initializing the PIC, and managing the registration and dispatching of interrupts.

* Interrupt Handler Class

#### Sets up the entry for the system call interrupt is part of the InterruptManager constructor:

*// Set up entry for system call interrupt*

SetInterruptDescriptorTableEntry(0x80, CodeSegment, &HandleInterruptRequest0x80, 0, IDT\_INTERRUPT\_GATE);

#### Triggers system calls using a software interrupt (int $0x80):

*// Function to get the process ID*

int myos::getPid()

{

int pId = -1;

asm("int $0x80" : "=c" (pId) : "a" (SYSCALLS::GETPID));

return pId; }

*// Function to wait for a specific process to finish*

void myos::waitpid(common::uint8\_t wPid)

{

asm("int $0x80" : : "a" (SYSCALLS::WAITPID), "b" (wPid)); }

*// Function to exit the current process*

void myos::sys\_exit()

{

asm("int $0x80" : : "a" (SYSCALLS::EXIT)); }

*// Function to print a string using a system call*

void myos::sysprintf(char\* str)

{

asm("int $0x80" : : "a" (SYSCALLS::PRINTF), "b" (str)); }

*// Function to create a new process (fork)*

void myos::fork()

{

asm("int $0x80" :: "a" (SYSCALLS::FORK)); }

*// Function to create a new process and get the child PID*

int myos::fork\_with\_pid(int pid)

{

asm("int $0x80" : "=c" (pid) : "a" (SYSCALLS::FORK));

return pid; }

*// Function to execute a new program in the current process*

int myos::exec(void entrypoint())

{

int result;

asm("int $0x80" : "=c" (result) : "a" (SYSCALLS::EXEC), "b" ((uint32\_t)entrypoint));

return result; }

*// Function to add a new task to the task manager*

int myos::addTask(void entrypoint())

{

int result;

asm("int $0x80" : "=c" (result) : "a" (SYSCALLS::ADDTASK), "b" ((uint32\_t)entrypoint));

return result; }

#### Implementation and handle the actual logic for each system call and interacts with the TaskManager to perform operations:

*// Implementation of system call handlers*

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);

}

#### Initialize the SyscallHandler and add it to the InterruptManager:

*// SyscallHandler constructor to initialize the interrupt handler for system calls*

SyscallHandler::SyscallHandler(InterruptManager\* interruptManager, uint8\_t InterruptNumber)

:InterruptHandler(interruptManager, InterruptNumber + interruptManager->HardwareInterruptOffset())

{}

*// Function to handle system call interrupts*

uint32\_t SyscallHandler::HandleInterrupt(uint32\_t esp) {

CPUState\* cpu = (CPUState\*)esp;

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); }

break;

case SYSCALLS::GETPID:

cpu->**ecx** = InterruptHandler::sys\_getpid();

break;

case SYSCALLS::ADDTASK:

cpu->**ecx** = InterruptHandler::sys\_addTask(cpu->**ebx**);

break;

default:

break;

}

return esp; }

### Round Robin Scheduling

The scheduling part also provided by the source code. This function does:

* + Triggered by a hardware timer.
  + Calls the Schedule method in the TaskManager to potentially switch processes.
  + Selects the next process in the ready queue.
  + Performs a context switch to the new process.
  + Prints the process table with relevant information.
  + Display all entries in the process table.
  + Include details such as PID, PPID, state.

CPUState\* TaskManager::Schedule(CPUState\* cpustate)

{

*// Simple delay loop for demonstration purposes*

for (int i = 0; i < 10000000; i++) {}

if (**numTasks** <= 0)

return cpustate;

if (**currentTask** >= 0)

**tasks**[**currentTask**].**cpustate** = cpustate;

*// Find the next READY task*

int findTask = (**currentTask** + 1) % **numTasks**;

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;

}

else if (**tasks**[waitTaskIndex].**taskState** == READY)

{

findTask = waitTaskIndex;

continue;

}

}

}

findTask = (findTask + 1) % **numTasks**;

}

**currentTask** = findTask;

PrintProcessTable();

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

### Lifecycle (Strategy One and KernelMaın)

#### Collatz Function:

void collatz(int n) {

printf("Collatz with parameter: ");

printNum(n);

printf("\n");

printf("Result");

printf(": ");

while (n != 1) {

if (n % 2 == 0) {

n /= 2; }

else {

n = 3 \* n + 1;}

printNum(n);

printf(" ");}

printf("\n");}

#### Long Running Program Function:

common::uint32\_t 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; }

Below function strategyOne does:

* Forks three child processes to run the Collatz function.
* Forks three child processes to run the long-running program.
* Waits for all six child processes to finish before printing that all child processes have ended.

#### strategyOne Function:

void strategyOne(){

int childPID[5];

int childCounter = 1;

printf("{{{{{{{{{ Test Strategy 1 }}}}}}}}}");

printf("\n");

int initialPid = getPid();

printf("Initial Process PID: ");

printNum(initialPid);

printf("\n");

int collatzTest, longTest;

int numForks = 3; *// Number of times to fork to create 6 programs*

for (int i = 0; i < numForks; ++i){

int childPid;

int result = fork\_with\_pid(childPid);

if (result == 0) { *// This is the child process*

printNum(childCounter);

printf(". Child Process with");

printf(" PID: ");

printNum(getPid());

printf(" runs collatz");

printf("\n");

collatzTest = getPid()+5;

collatz(collatzTest);

sys\_exit(); }

else { childPID[i] = childCounter++; } }

for(int i = 0; i < numForks; ++i){

int childPid;

int result = fork\_with\_pid(childPid);

if (result == 0) { *// This is the child process*

printNum(childCounter);

printf(". Child Process with");

printf(" PID: ");

printNum(getPid());

printf(" runs long running program with ");

longTest = 1000 + i;

printNum(longTest);

printf("\n");

printf("Result:");

printNum(long\_running\_program(longTest));

printf("\n");

sys\_exit(); } else { childPID[i] = childCounter++; }}

for(int i = 0; i < 6; ++i) { waitpid(childPID[i]); }

printf("All child processes are ended."); }

#### kernelMain Function:

extern "C" void kernelMain(const void\* multiboot\_structure, uint32\_t */\*multiboot\_magic\*/*)

{

printf("-----------------Welcome to the ZortOS--------------\n");

GlobalDescriptorTable gdt;

TaskManager taskManager(&gdt);

*//Task taskTestExec(&gdt,execTestExamle);*

*//taskManager.AddTask(&taskTestExec);*

*//Task taskTestFork(&gdt,forkTestExample);*

*//taskManager.AddTask(&taskTestFork);*

Task taskStrategyOne(&gdt,strategyOne);

taskManager.AddTask(&taskStrategyOne);

InterruptManager interrupts(0x20, &gdt, &taskManager);

SyscallHandler syscalls(&interrupts, 0x80);

// Remanining parts...

#### Necessary System Calls Test Functions:

void executeThisFunction() {

printf("Exec System Call Runned Properly!");

printf("\n");

sys\_exit(); }

void execTestExamle() {

*//printf("{{{{{{{{{ Test Exec }}}}}}}}}");*

*//printf("\n");*

printf("Exec syscall is testing with PID: ");

printNum(getPid());

printf("\n");

int exec1 = exec(executeThisFunction);

sys\_exit(); }

void forkTestExample() {

*//printf("{{{{{{{{{ Test Fork }}}}}}}}}");*

*//printf("\n");*

int parentPID=getPid();

printf("Starting with parent pid: ");

printNum(parentPID);

printf("\n");

int childPID = fork\_with\_pid(getPid());

if(childPID==0){

printf("Forked child pid ");

printNum(getPid());

printf("\n");

sys\_exit();}

else {

printf("Parent pid which should be the same as before: ");

printNum(getPid());

printf("\n");}

sys\_exit(); }

### Outputs

#### Sysem Calls Test and Process Table

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

#### Strategy One Test and Process Table

A screenshot of a computer

Description automatically generated

It was hard to takle screenshots during this test but because of timer interrupts but I’ve managed to see chil process’s were task switching by callint process table print function insde scheduling function.

A screen shot of a black screen

Description automatically generatedA computer screen with white text

Description automatically generated