**CIS 657 Programming Assignment 2**

Submit two files:

* Create your document -- readme (doc or pdf) including the **disclosure form**, **screenshot(s) of your output**, **your solution code *– Must be text (image cannot be process with TurnItIn) --* (filename including full path, and your own code (created or modified by you), not Nachos original code) with description (you need to explain your solution)**  to the **Turnitin link** on the blackboard (under Assignments).
  + All of your code must be well documented.
  + Your solution/design must be clearly explained.
  + Make sure that you receive the confirmation email from the TurnItIn.
* Do a “make clean” in code/build.linux, then compress your Nachos folder on the VM, download the compressed file into your local machine and name the compressed file **Ass2\_yourNetId.zip (Note: Zip only!!)**. Submit the zipped file to the **Assignment Code Submission link** of Blackboard.
  + Make sure that your Nachos runs on the VM and your zip file is uncompressed.

Due: Nov. 16 (Friday, end of the day)

Late submission: you have total 72 hours throughout the semester.

*Follow the Lab1 instruction and create a new fresh Nachos folder.*

**Overview**

So far, all the code you have written for Nachos has been part of the operating system kernel (ThreadTest() is compiled and ran within Nachos). In a real operating system, the kernel not only uses its procedures internally, but allows user-level programs to access some of its routines via “system calls”. In this project, you are going to implement several system call routines for user programs and round-robin scheduling.

The user programs are written in C and located under test/ directory. There are several example programs provided, including halt, shell, etc. The command “./nachos -x ../test/programname” will load the program ‘programname’ to Nachos memory (see addrspace.cc and progtest.cc) and run it on Nachos.

**Task 1 (30pts) System Calls for user program:** userprog/syscall.h defines the system call prototypes of Nachos. These are kernel procedures that user programs can invoke. You need to implement the following system calls.

* **Fork\_POS()**enables the user program to fork a kernel child thread. It takes integer 1, 2 or 3 as parameter, creates a corresponding kernel thread, which executes one of the test functions basing on the parameter value, and returns an integer as the process ID of the child. For example, when child = Fork\_POS(2) is called in a user program, a kernel thread will be created in ExceptionHandler() and this thread will call the ForkTest2 function (using the Fork()call of thread.cc). The child process ID will be assigned to child.

void ForkTest1(int id)

{

printf("ForkTest1 is called, its PID is %d\n", id);

for (int i = 0; i < 3; i++)

{

printf("ForkTest1 is in loop %d\n", i);

for (int j = 0; j < 100; j++)

kernel->interrupt->OneTick();

}

Exit\_POS(id);

}

void ForkTest2(int id)

{

printf("ForkTest2 is called, its PID is %d\n", id);

for (int i = 0; i < 3; i++)

{

printf("ForkTest2 is in loop %d\n", i);

for (int j = 0; j < 100; j++)

kernel->interrupt->OneTick();

}

Exit\_POS(id);

}

void ForkTest3(int id)

{

printf("ForkTest3 is called, its PID is %d\n", id);

for (int i = 0; i < 3; i++)

{

printf("ForkTest3 is in loop %d\n", i);

for (int j = 0; j < 100; j++)

kernel->interrupt->OneTick();

}

Exit\_POS(id);

}

* **Wait\_POS()** provides the routine for the user program to wait for the execution of its child thread. It takes the child process ID as parameter. The parent will be put into sleep until the child awakens it. The child wakes up the parent when it finishes by calling Exit\_POS, which you’ll implement. *Wait\_POS() and Exit\_POS() should be able to handle certain errors such as invalid child process ID or no parent process waiting.*
* Exit\_POS() is called by a child and wakes up its parent. This function is not a system call but you need to implement in exception.cc

For this task, syscall.h, exception.cc, start.s (assembly routine for system calls and under test/ directory), and scheduler.cc must be modified. The userprog/exception.cc implements the handlers for system calls and other user-level exceptions. In the original file, only the Halt() system call is supported.

Hint: you can find a good example from system call Add.

You can find test programs (prog1.c, prog2.c and prog3.c) in the Testing section. For this task testing, you don’t need multitasking with round-robin.

**Task 2 (20pts) Multitasking (running multiple user-level programs):** when the ‘-x’ flag is used, the ‘main’ function of Nachos calls the RunUserProg() function with the string following ‘-x’ as parameter. Remember that every user program should have its own thread structure, but for now there’s only one thread, the ‘main’ thread.

The goal of this task is to make the command

./nachos -x ../test/mprog1 -x ../test/mprog2

work in Nachos. What this means is that when you type this command, Nachos will load multiple user programs into its main memory and start executing them as threads with round-robin scheduling. You need to implement Write system call.

* **Write()**writes string to a speciﬁed destination. We’re faking it using printf, therefore it always writes to the screen. This system call takes three arguments: a char\* buffer, an int size, and an OpenFileId. Since we’re faking it, we can ignore the OpenFileId parameter. You need to obtain the ﬁrst two arguments from the user program and print an character at a time using printf.

**Task 3 (45pts) Memory Allocation (Address Space for user programs):** The main issue in this implementation is how to create the address spaces of the Nachos processes and put them all in the main memory without overlapping. The allocation of address space is done in userprog/addrspace.cc.

If the total size of the user programs (for example, ./nachos -x ../test/prog1 -x ../test/prog2 …) are smaller than the memory size, you can just use contiguous memory allocation. However, for the command like ./nachos -x ../test/matmult, you need to consider larger memory allocation needed than Nachos physical memory size. matmult is a matrix multiplication program. If you run the program now, you’ll get an assertion failure in userprog/addrspace.cc because there’s not enough memory to load it. So what you need to accomplish is to let Nachos run the program with part of the address space loaded, and load the other part(s) when they are accessed. Because an exception will arise when they are accessed, the loading should be done as an exception handling in userprog/exception.cc.

You’re free to use whatever design you choose.

* For the first case, you can use the page table and assume the memory allocation is contiguous by keeping track of the first free page frame to load a given user program’s address space.
* For the second case, you can store the whole address space in a file, and load the accessed part every time an exception arises.

**Task 4 (5pts) Round-robin scheduling:** Round-robin scheduling gives each process (Nachos thread) a fix time slice (quantum) to run. After the quantum expires, the CPU will be taken from the process and given to the next process in the ready list. The older process is then inserted to the end of the ready list.

* You already added the flag “quantum” from the programming assignment 1, i.e., when you type the command “./nachos -quantum 200”, Nachos will run with the quantum set to 200 clock ticks.

**Testing:**

We will build and run your Nachos on the VM. TAs will test multitasking with user programs with their quantum size and memory allocation using matmult program.

Place the following programs into test/ directory and update Makefile. **The test programs must be compiled before Nachos is built.**

// Task 1 testing programs

/\*\*\* prog1.c \*\*\*/

#include "syscall.h"

int

main()

{

int child;

child = Fork\_POS(1);

Wait\_POS(child);

child = Fork\_POS(2);

Wait\_POS(child);

child = Fork\_POS(3);

Wait\_POS(child);

Exit(0);

}

/\*\*\* prog2.c \*\*\*/

#include "syscall.h"

int

main()

{

int child1, child2;

child1 = Fork\_POS(1);

child2 = Fork\_POS(2);

Wait\_POS(child1);

Wait\_POS(child2);

Exit(0);

}

/\*\*\* prog3.c \*\*\*/

#include "syscall.h"

int

main()

{

Fork\_POS(1);

Fork\_POS(2);

Fork\_POS(3);

Exit(0);

}

// mprog1.c for Task 2, 3 and 4 testing

#include "syscall.h"

int

main()

{

OpenFileId output = ConsoleOutput;

char\* str = "current user program: prog1";

int i,j;

for (i = 0; i < 5; i++)

{

Write(str, 27, output);

for (j = 0; j < 10000; j++);

}

Exit(0);

}

// mprog2.c for Task 2, 3 and 4 testing

#include "syscall.h"

int

main()

{

OpenFileId output = ConsoleOutput;

char\* str = "current user program: prog2";

int i,j;

for (i = 0; i < 5; i++)

{

Write(str, 27, output);

for (j = 0; j < 10000; j++);

}

Exit(0);

}