CS 3423 Operating Systems
Fall Semester 2017
Prof. Pai H. Chou

Assignment 2

Due Date: Sunday, October 1, 2017, 11:59pm
Submit electronically on iLMS

What to submit: One zip file named <studentID>-`hw2.zip` (replace <studentID> with your own student ID). It should contain four files:

- one PDF file named **hw2.pdf** for Section 1, Section 2.3, Section 2.4.1. Write your answers in **English**. Check your spelling and grammar. Include your name and student ID!
- Section 2.4.2: Your modified **exception.cc** file
- Section 2.4.3 - 2.4.4: the **typescript** file
- Section 3: Python source file named **hw2.py** (exact upper and lower case). Include your name and student ID in the program comments on top.

1. [40 points] Problem Set

From Chapter 2 of Sieberchatz 9th edition book. "OS Structures"

1. [20 points] 2.7: What are the two models of interprocess communication? What are the strengths and weaknesses of the two approaches?
2. [20 points] 2.10: What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?

2. [40 points] Nachos Exercise

Before writing a program to run on Nachos, you need to know where things are and where to make the change. We will use the following shorthand:

- {code} is for {Nachos}`/code`
- {test} is for {code}`/test`
- {userprog} is for {code}`/userprog`

etc.

Before you begin, type the command into your Ubuntu setup for Nachos:

```
$ sudo apt-get update
$ sudo apt-get install gcc-multilib g++-multilib
```

2.1 How `halt.c` Works

To recall, `halt.c` in the {test} directory from the last assignment is a user program that is loaded and and run by Nachos. It is compiled by a special version of GCC (in {Nachos}`/usr/local/nachos/bin/decstation-ultrix/bin/`, not the regular GCC

for your host system) to an executable file named `halt` in the MIPS instruction set, so it can be executed by the nachos command `nachos -e halt`

Examine the source code for `halt.c`. It contains the line

```
#include "syscall.h"
```

but there is no file named `syscall.h` in the same directory. Since the `Makefile` specifies the rule

```
INCDIR =-I../userprog -I../lib
```

GCC will look in two other directories for include files. In this case, `syscall.h` is found in {userprog} directory. It defines the trap numbers for the system calls:

```
#define SC_Halt         0
#define SC_Exit         1
#define SC_Exec         2
#define SC_Join         3
...
```

Since `Halt()` is a system call, it means it is implemented as a trap to the Nachos kernel in terms of a `syscall` instruction in MIPS with the code 0 in register `$v0` (which is another name for register `$2`).

2.2 Trap Handling by Nachos Kernel

In Nachos, traps are handled by the function {userprog}`/exception.cc:ExceptionHandler()`. It is called by the (simulated) MIPS processor when executing a trap. It contains a switch statement

```
switch(type) {
     case SC_Halt:
          ..
          break;
```

2.3 [5 points] System Call Implementation

Note that `Halt()` is called a *stub* function, because it just a way for C programs to make the call, but it redirects it to the kernel. You will not find it in any of the `.c` or `.cc` files; instead, it is written in assembly language, because it involves the use of machine registers and instructions. Assembly language files have `.S` in their names.

[5 points] On line {test}`/start.S:48`, you have the MIPS assembly instruction

```
addiu $2, $0, SC_Halt
```

which is another way of saying
register2 = 0 + SC_Halt;
Explain the purpose of this assignment statement.

2.4 [35 point] Hello World in Nachos

Suppose you want to write a Hello World program to run in Nachos by making a copy of `halt.c` as `hw2.c` and calling `Write()`:

```
#include "syscall.h"
int main() {
    Write("hello world\n", 12, 1);
    Halt();
}
```

However, the `Write()` system call has not been implemented in the given {userprog}`/exception.cc` source file, so it will crash. Nachos kernel can be modified to handle the `Write()` system call.

2.4.1 Answer the following questions for the `Write()` system call. Hint: the answers can be found in the source code of exception.cc, including the comments!

- [5 points] How does the kernel get the pointer to the array of bytes to write? Give the description and C code.
- [5 points] How does the kernel get the number of bytes to write? Give the description and C code.
- [5 points] How does the kernel return the value to the caller? (for the number of bytes written?) Give the description and C code.

2.4.2 [10 points] Modify `exception.cc` so that it

- displays that `Write` is being called
- displays the pointer value (not the content) in hex using a DEBUG statement,
- displays the number of bytes to write using a DEBUG statement,
- copy the size parameter value as its return value.

2.4.3 [5 points] Modify {test}`/Makefile` to build `hw2` as an executable. To do so, search for `halt` in the `Makefile` and copy it for `hw2`. You should be able to run `make` in the {test} directory to build `hw2` as an executable.

2.4.4 [5 points] rebuild Nachos in the {code}`/build.linux` directory. If successful, you should be able to run (from {test} directory) `../build.linux/nachos -d -e hw2`

For parts 2.4.3 and 2.4.4, submit a typescript showing compilation and running your code. To force recompiling, you and do a make clean on both hw2.c (in {test}) and Nachos itself (in {build.linux}). In the following, $ is the prompt character:

```
$ script
Script started, file is typescript
$ cd {build.linux}
$ make clean
$ make
$ cd {test}
$ make clean
$ make
$ {build.linux}/nachos -d u -e hw2
$ ^D
Script done, file is typescript
$
```

You will find a file named typescript in the directory where you started. Submit the unedited typescript. **However, if your code fails to run correctly for any reason, then you must make an appointment with a TA to get graded in person.**

3. [20 points] Python Programming

This week, you are to write the `YieldBST(T)` function, which is a slight modification to the `PrintBST(T)` function you wrote for Assignment 1. The difference is that you use the **`yield`** keyword instead of `print`, so that it can be used as an *iterator* that feeds a **`for`** loop. One thing you do have to do is that if your function calls another function that does a yield (including recursive calls), then you would have to say **`yield from`** before calling that function.

You should be able to do this in your test case:

```
if __name__ == '__main__':
      L = []
      T = … # from last week
      for v in YieldBST(T):
            L.append(v)
      print(L)   # print list
```

when you run it as a top-level file

```
$ python hw2.py
```

and then it should print out

```
[6, 12, 14, 17, 32, 35, 40]
```

The use of yield keyword in a function makes it a *generator*. You can think of it as a *coroutine*, i.e., a function that can continue execution after it produces return values. It is not quite a thread yet, but has some flavor of concurrency. The difference is generators are passive.