Assignment 2

CS330: Operating Systems

1 Introduction

As part of this assignment, you will be implementing system calls in a teaching OS (gemOS), some of which you became familiar during Assignment-1. We will be using a minimal OS called gemOS to implement these system calls and provide system call APIs to the user space. The gemOS source can be found in the `src` directory. This source provides the OS source code and the user space process (i.e., `init` process) code (in `src/user` directory).

gemOS is a teaching operating system. Typically OS boots on a hardware (bare metal or virtual). We will be using gem5 (http://gem5.org/Main_Page), an open source architectural simulator to boot **gemOS**. An architectural simulator simulates the hardware in software. In other words, all the hardware functionalities are implemented in software using some programming language. For example, Gem5 simulator implements architectural elements for different architectures like ARM, X86, MIPS etc. Advantages of using software simulators for OS development are,

- Internal hardware state and operations at different components (e.g., decoder, cache etc.) can be profiled to better understand the hardware-software interfacing.
- Hardware can be modified for several purposes—make it simpler, understand implications of hardware changes on software design etc.
- Bugs during OS development can be better understood by debugging both hardware code and the OS code.

The getting started guide helps you to setup Gem5 and execute `gemOS` on it. You can refer to this online tutorial to know more about gem5.

You need to setup gem5 by following the instructions (Step-0) mentioned in Section 2. We suggest to complete **Step-0** immediately to have a working gem5 simulator to start with the assignment. **Step-0** also helps you to understand how to boot **gemOS** binary in gem5. You can build the gemOS source provided in the `src` directory to test the working.

The assignment is divided into five tasks. The implementation of system calls for each task should be POSIX compliant. Which means, you need to read the man pages of each of corresponding system calls to know about their exact behavior. Testing procedure and submission guidelines are mentioned at the end of the document.

2 Step-0: Getting Ready

This section explains the setup procedure of gem5. Further, it explains the process to build and execute gemOS on gem5 platform and access the terminal to see the messages printed by the gemOS.

2.1 Preparing Gem5 simulator

System pre-requisites

- Git
- gcc 4.8+
- python 2.7+
- SCons
- protobuf 2.1+

On Ubuntu install the packages by executing the following command.

```
$ sudo apt-get install build-essential git m4 scons zlib1g zlib1g-dev
libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-dev
python-dev python automake
```

Gem5 installation

Clone the gem5 repository from `https://gem5.googlesource.com/public/gem5`.

```
$ git clone https://gem5.googlesource.com/public/gem5
```

Change the current directory to `gem5` and build it with scons:

```
$ cd gem5
$ scons build/X86/gem5.opt -j9
```

In place of 9 in [-j9], you can use a number equal to available cores in your system plus one. For example, if your system have 4 cores, then substitute -j9 with -j5. For first time it will take around 10 to 30 minutes depending on your system configuration. After a successful Gem5 build, test it using the following command,

```
$ build/X86/gem5.opt configs/example/se.py --cmd=tests/test-progs/hello/bin/x86/linux/hello
```

The output should be as follows,

```
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.
gem5 compiled Aug  4 2018 11:00:44
gem5 started Aug  4 2018 17:15:06
```

```
gem5 executing on BM1AF-BP1AF-BM6AF, pid 8965
command line: build/X86/gem5.opt configs/example/se.py \
              --cmd=tests/test-progs/hello/bin/x86/linux/hello
/home/user/workspace/gem5/configs/common/CacheConfig.py:50: SyntaxWarning: import * only \
              allowed at module level def config_cache(options, system):
Global frequency set at 1000000000000 ticks per second
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7000
**** REAL SIMULATION ****
info: Entering event queue @ 0. Starting simulation.....
Hello world!
Exiting @ tick 5941500 because exiting with last active thread context
```

2.2 Booting gemOS using Gem5

Gem5 execute in two modes—system call emulation (SE) mode and full system (FS) simulation mode. The example shown in the previous section, was a SE mode simulation of Gem5 to execute an application. As we want to execute an OS, Gem5 should be executed in FS mode. There are some initial setup to do before we can execute `gemOS` using Gem5 FS mode. To run OS in full-system mode, where we are required to simulate the hardware in detail, we need to provide the following files,

- `gemOS.kernel`: OS binary built from the `gemOS` source.
- `gemOS.img`: root disk image
- `swap.img`: swap disk image

Gem5 is required to be properly configured to execute the `gemOS` kernel. The configuration requires changing some existing configuration files (in gem5 directory) as follows,

- Edit the `configs/common/FSConfig.py` file to modify the `makeX86System` function where the value of `disk2.childImage` is modified to `(disk('swap.img'))`.
- Edit the `configs/common/Benchmarks.py` file to update it as follows,

```
elif buildEnv['TARGET_ISA'] == 'x86':
      return env.get('LINUX_IMAGE', disk('gemOS.img'))
```

Create a directory named `gemos` in `gem5` directory and populate it as follows,

```
/home/user/gem5$ mkdir gemos
/home/user/gem5$ cd gemos
/home/user/gem5/gemos$ mkdir disks; mkdir binaries
/home/user/gem5/gemos$ dd if=/dev/zero of=disks/gemOS.img bs=1M count=128
/home/user/gem5/gemos$ dd if=/dev/zero of=disks/swap.img bs=1M count=32
```

For the time being, you can use `gemOS.kernel` provided with the assignment (can be found in `src` directory). Copy the `gemOS.kernel` to `gemos/binaries` directory.
We need to set the `M5_PATH` environment variable to the `gemos` directory path as follows,

```
/home/user/gem5$ export M5_PATH=/home/user/gem5/gemos
```

Now, we are ready to boot GemOS.

```
gem5$ build/X86/gem5.opt configs/example/fs.py
--kernel=/home/user/gem5/gemos/binaries/gemOS.kernel --mem-size=2048MB
```

gem5 output will look as follows,

```
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled Aug 21 2019 23:45:13
gem5 started Aug 22 2019 10:45:01
gem5 executing on kparun-BM1AF-BP1AF-BM6AF, pid 28942
command line: build/X86/gem5.opt configs/example/fs.py --kernel=/home/kparun/gem5/gemos/binaries/gemOS.kernel --mem-size=2048MB

Global frequency set at 1000000000000 ticks per second
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
info: kernel located at: /home/kparun/gem5/gemos/binaries/gemOS.kernel
system.pc.com_1.device: Listening for connections on port 3456
      0: rtc: Real-time clock set to Sun Jan  1 00:00:00 2012
0: system.remote_gdb: listening for remote gdb on port 7000
warn: Reading current count from inactive timer.
**** REAL SIMULATION ****
info: Entering event queue @ 0.  Starting simulation...
warn: Don't know what interrupt to clear for console.
```

Execute the following command in another terminal window to access the `gemOS` console

```
/home/user$ telnet localhost 3456
```

At this point, you should be able to see the `gemOS` shell.

2.3 How to build gemOS

To build **gemOS.kernel**, you need to run **make** inside **src** folder. After that you need to copy **gemOS.kernel** binary to `gemos/binaries` directory. This step is necessary every time you build the gemOS and want to test it. After copying `gemOS.kernel`, run

```
gem5$ build/X86/gem5.opt configs/example/fs.py
--kernel=/home/user/gem5/gemos/binaries/gemOS.kernel --mem-size=2048MB
```

Open a terminal window as before and access the console using the following command,

```
/home/user$ telnet localhost 3456.
```

2.4 How to test your Implementation

In the **GemOS#** terminal (accessed using the `telnet` command as shown above), you can type **init** to execute the user space process i.e, `init`. The user space code is available in

`src/user/init.c`. Three user space files are used to implement the user space logic. They are

`init.c`: Implements the first user space process which can invoke `fork()` to create more processes. Note that, there is no `exec` system call yet in the version provided to you. For changing the user space logic, you are required to modify *only* `init.c`.

`lib.c`: Implements system call wrappers and provide different user space libraries (e.g., `printf`). Note that you *do not* modify this file.

`lib.c`: Provides declarations of macros and functions. Note that you *do not* modify this file.

You need to write your test cases in **init.c** to validate your implementation. The sample test-cases (in `src/user/test_cases/testcase*.c`) can be copied into `init.c` to make use of them. If your implementation is correct, the output of executing test cases should match the expected output provided in `src/user/test_cases/testcase*.output`. The user and kernel code are compiled into a single binary file, i.e., **gemOS.kernel** when built using `make` from the `src` directory.

The real assignment

The process control block (PCB) is implemented using a structure named `exec_context` defined in `src/include/context.h`. One of the important member of `exec_context` for this assignment is an array of `struct file` (declared in `include/file.h`) named as `files`. This is the file descriptor table where the index of the array (in `files`) corresponds to the file descriptor. For example, `files[0]` represents the file descriptor 0 and points to the file object for the standard input (STDIN). You are required to manipulate the `files` structure and provide appropriate logic for the file object to implements the assignment. The template code provides detailed documentation for understanding the tasks further. Note that, there are several function pointers in `struct file` which can be implemented to provide file system functionalities as we discuss further.

3 Task-1: Basic file operations (20 Marks)

List of Syscalls to Implement

- `int open(const char *pathname, int flags,int mode)`
- `int read(int fd, void *buf, int count)`
- `int write(int fd, const void *buf,int count)`

3.1 open

To implement `open` system call, you are required to provide implementation for the template function `do_regular_file_open` (in `file.c`) which takes the current context, filename, flags

and mode as arguments. Open call can be used to open an existing file or create a new one by passing the `O_CREAT` flag as per the POSIX semantics. For regular files, an underlying `inode` is provided through the FS APIs which you are required to invoke.
While creating a file, the first step is to get an `inode` from the underlying FS (File System) layer by invoking `create_inode` (implemented in fs.c). The signature of `create_inode` is as follows,

```
struct inode *create_inode (char *filename, u64 mode)
```

where `filename` and `mode` should be same as it is passed to the `do_regular_file_open` function. The mode can take **O_READ, O_WRITE, O_EXEC** values which corresponds to Read, Writ and Execute permissions (passed by the user). Permission check is performed on read/write access based on mode value, i.e., write call on a file which is created with O_READ mode should return an **EACCES** error.

Now let us look at the second scenario of opening an existing file. The first step here is look up the inode corresponding to the filename from the underlying FS layer by invoking `lookup_inode` (in fs.c). The signature of **lookup_inode** is

`struct inode* lookup_inode(char *filename).`

A valid inode is returned on success (NULL on error) and you need to ensure that the access flags mentioned in `open` are compatible with the mode in which file was created. After getting the inode from the FS layer, you need to find a free file descriptor, allocate a file object (using `alloc_file` method in `file.c`) and fill-in the fields of corresponding *struct file* object which is pointed to by **files** (in context.h) field of current execution context. Here you need to look for a free position in **files** array starting from **index 3**. Index positions **0, 1, 2** corresponds to **stdin, stdout, stderr**. You need to implement `do_regular_read`, `do_regular_write` and `std_close` functions and assign them to read, write and close function pointers of `struct fileops` by accessing `fops` field in the struct file. As last step of open call, you need to return the file descriptor which is returned back to the user and used for subsequent file operations.

The implementation of file objects and operations for STDIN, STDOUT and STDERR are already provided to help you with the understanding of the task.

3.2 read

You need to implement the `do_read_regular` function (in `file.c`). This function is to be assigned as the read handler in the file object while opening the file. The `inode` provides a read method (`flat_read`) with the following signature

`int flat_read(struct inode *, char *buf, int count, int *offset).`

where, `buf` and `count` are the user buffer and count, respectively, passed to `do_read_regular` from the read system call handler. The above function returns the number of bytes read from the underlying file. Read implementation for STDIN, `do_read_kbd`, is provided in `file.c` as an illustration.

3.3 write

You need to implement the `do_write_regular` function (in `file.c`). This function is to be assigned as the write handler in the file object while opening the file. The `inode` provides a write method (`flat_read`) with the following signature

`int flat_write(struct inode *, char *buf, int count, int *offset)`.

where, `buf` and `count` are the user buffer and count, respectively, passed to `do_write_regular` from the read system call handler. The above function returns the number of bytes written to the underlying file. Write implementation for stdout/stderr, `do_write_console`, is provided in `file.c`.

Notes

- You are required to modify only `file.c`.
- Sample test case and expected output for this task are in `src/user/test_cases/testcase1.c` and `src/user/test_cases/testcase1.output`, respectively.
- Refer to the section detailing the test procedure to know about the limits and assumptions related to this task.

4 Task-2: More file operations (15 Marks)

List of Syscalls to implement

- `int dup(int oldfd)`
- `int dup2(int oldfd, int newfd)`
- `long lseek(int fd, long offset, int whence)`

4.1 dup

You have to implement `fd_dup` function (in file.c). It takes current execution context and oldfd as arguments. You need to return error codes (in entry.h) based on the error conditions as explained in the section detailing the error codes.

4.2 dup2

You have to implement `fd_dup2` function (in file.c). It takes current execution context, oldfd and newfd. Before making newfd as a copy of oldfd, you need to close newfd if it is open.

4.3 lseek

You need to implement **do_lseek_regular** (in file.c). It takes pointer to `struct file`, `offset` and `whence` as arguments. You need to implement the functionality for three `whence` options `SEEK_SET, SEEK_CUR, SEEK_END` (in file.h). You need to return error codes (in entry.h) based on the error conditions. Note that, if `lseek` results in taking the file offset beyond the file end, you need to return error code −`EINVAL`.

Notes

- You are required to modify only `file.c`.
- Sample test case and expected output for this task are in `src/user/test_cases/testcase2.c` and `src/user/test_cases/testcase2.output`, respectively.
- Refer to the section detailing the test procedure to know about the limits and assumptions related to this task.

5 Task-3: Pipe it! (15 Marks)

List of Syscalls to implement

- `int pipe(int fd[2])`

5.1 pipe

You need to implement `create_pipe`, `pipe_read` and `pipe_write` functions (in pipe.c). `create_pipe`, invoked to implement `pipe` system call, takes pointer to current execution context and file descriptor array (of two elements) as arguments. File descriptors for read and write ends of pipe are assigned by looking for available indices in `files` array of the current context. You can use `alloc_file` API (in file.c) to allocate a `file` objects and associate it with read and write end of the pipe. Use `alloc_pipe_info` (in pipe.c) to get a pointer to a pipe object (`struct pipe_info`) and attach it with the `pipe` field of the `file` object. Note that, in `struct pipe_info`, `pipe_buf` can be used to implement data write and read. You should fill-in the fields of `struct pipe_info` in `alloc_pipe_info` function. You need to implement `pipe_read` and `pipe_write` functions and assign them to read and write function pointers of `struct fileops` by accessing `fops` field in struct file.

Notes

- You are required to modify only `pipe.c`.
- Sample test case and expected output for this task are in `src/user/test_cases/testcase3.c` and `src/user/test_cases/testcase3.output`, respectively.
- Refer to the section detailing the test procedure to know about the limits and assumptions related to this task.

6 Task-4: Handling `close()`, `fork()` and `exit()`

(10 Marks)

List of functionalities to implement

- System call `int close(fd)`
- Handler for process exit `void do_file_exit(struct exec_context *ctx` in `file.c`
- Handler for process creation through fork `void do_file_fork(struct exec_context *child` in `file.c`

Details on Implementation

You have to implement `generic_close` (in file.c) as part of this task. Note that, the generic close implements closing regular files and pipes. You need to ensure that the reference count in the file object associated with the fd is maintained correctly. When the last reference to the file object is dropped, you need to invoke `free_file_object(stuct file *)` and `free_pipe_object(struct pipe_info *)` as applicable. As a program may exit without closing the files, you need to perform file close on `exit` system call by appropriately implementing `do_file_exit`. When a child process is created using `fork`, the file objects are shared, as the FDs in files are already copied while creating the child process. You are required to adjust the reference counts as applicable (in `do_file_fork`).

Notes

- You are required to modify only `file.c`.
- Sample test case and expected output for this task are in `src/user/test_cases/testcase4.c` and `src/user/test_cases/testcase4.output`, respectively.
- Refer to the section detailing the test procedure to know about the limits and assumptions related to this task.

7 Task-5: Putting it all together!(40 Marks)

As part of this task, your implementation will be tested for all syscall APIs that you have implemented in Task 1-4. You need to verify that your implementation works in a holistic manner.

Notes

- You may be required to modify `file.c` and `pipe.c` if your first-cut logic is incorrect. If you have got everything correct, Bravo!

- Sample test case and expected output for this task are in `src/user/test_cases/testcase5.c` and `src/user/test_cases/testcase5.output`, respectively.
- Refer to the section detailing the test procedure to know about the limits and assumptions related to this task.

Error codes

You should only use following error codes to mention error conditions. All these error codes should be negated before returning (Example: EINVAL should be returned as -EINVAL).

- **EINVAL**(Invalid Argument) It should be used in-case of invalid argument such as filename does not exist, invalid file descriptor, accessing closed file or pipe.
- **EACCES**(Invalid Access) It should be used in-case of invalid access such as writing to read-only file or pipe etc
- **ENOMEM**(No Memory) It should be used if memory allocation function used to allocate file, pipe_info fails.
- **EOTHERS**(Others) In case of any other errors which is not specified above use EOTHERS.

Test Procedure

We have provided you with five test cases(test_cases folder) to test Tasks 1-5. Apart from Task-5, We will be testing each Task individually and won't be mixing it up with other Tasks. We will be following below assumptions when we are testing your code. So lets go thorough all the assumptions

- There will be at-most four process that will be running at any point of time. No need to fork more than 4 process.
- There can be at-most 16 files of each 4KB size at any point of time.
- There can be at-most 16 file descriptor which can be created using dup or dup2 system calls.
- The max length of pipe buffer will be 4KB. We wont read or write more than 4KB into the pipe.
- We will be testing the error conditions using standard error codes which is specified above. Don't forget to negate the error code before returning.
- Don't try to create or allocate memory by yourself. Try to use the specified function. In-case of any issues reach out to us.

- Don't modify any other function or file. We will be evaluating the changes in the files (file.c and pipe.c)
- You need not worry about concurrency as all accesses are guaranteed to be performed from a single process.

Submission guidelines

The assignment is to be done individually. You have to submit only two files(file.c and pipe.c). Put these two files in a directory named as your roll number. Create a zip archive of the directory and upload in canvas. *Don't modify any other files.* We will not consider any file other than file.c and pipe.c for evaluation. In-case any issues you should reach out to us at the earliest. All the best!