CS236 Spring 2025

Lab Quiz2

4 questions, 20 marks

Instructions

Before you begin, please check that you can see the following files on the Desktop:

- This question paper `labquiz2.pdf`, with 4 questions
- A tarball `labquiz2_code.tgz`

Untar the code file as shown below:
```
tar -zxvf labquiz2_code.tgz
```

You will now find a folder `labquiz2_code` in the same directory, which has the following code:

- xv6 tarball in top-level directory
- Template code for each question in separate subdirectories, which will be described in the questions below

Now, create your submission folder titled `submission_rollnumber` in the Desktop directory, where you must replace the string `rollnumber` above with your own roll number (e.g., your directory name should look like `submission_12345678`). Place the files you wish to submit in this folder, in separate subdirectories (named `q1`, `q2`, `q3`, `q4`) for each question. When you finish the quiz, please ensure that the submission folder has all your solution code.

Note: For the xv6 questions, you must only submit the xv6 files you have changed, and not the entire xv6 folder.

Submission Instructions

1. Once you are ready to submit, please run the command `check` from your terminal. This command will create a tarball of your submission folder. If the folder is not in the proper place or is empty, this script will throw an error. In that case, please fix the errors and try again.
2. Next, call one of the TAs and ask them to run the `submit` command from your terminal. The TA will enter the password to upload your submission tarball to our remote submission server. Please note that we will only be grading the files that are submitted in this manner. So please ensure that you submit the correct files.
3. **IMPORTANT NOTE: Please ensure that you are submitting the correct files with the correct filenames. Test your code properly before submission. Strictly NO code changes will be allowed after the exam.**

Instructions related to xv6

- You are provided with an xv6 source directory that includes a simple test case in `simple.c`. This test case has already been added to the Makefile given to you.
- Each of the xv6 questions below come with their own test cases. You can add these test cases to the Makefile when you solve the question, or you can simply replace the simple test case with other test cases to test your code.
- For each question or subpart on xv6, create a separate new xv6 directory to preserve the unmodified original copy, and make changes to this new directory. Remember the files you are changing in this directory, and place them in the submission folder.
- To compile and run xv6 code, run `make`, followed by `make qemu` or `make-qemu-nox`.
- Below are the files that you may need to modify when adding new system calls in xv6.
  - `user.h` contains the system call definitions in xv6.
  - `usys.S` contains a list of system calls exported by the kernel, and the corresponding invocation of the trap instruction.
  - `syscall.h` contains a mapping from system call name to system call number. Every system call must have a number assigned here.
  - `syscall.c` contains helper functions to parse system call arguments, and pointers to the actual system call implementations.
  - `sysproc.c` contains the implementations of process related system calls.
  - `defs.h` is a header file with function definitions in the kernel.
  - `proc.h` contains the `struct proc` structure.
  - `proc.c` contains implementations of various process related system calls, and the scheduler function. This file also contains the definition of `ptable`, so any code that must traverse the process list in xv6 must be written here.

Questions

1. **[4 marks]** In this question, you will implement two programs, `socket-client.c` and `socket-server.c`, that communicate using UNIX domain sockets to transfer a file. Template files are provided to you.

   The client and server programs open a Unix domain socket to communicate between them. The server program accepts the name of the file to be transferred as a command-line argument. It then reads the file and sends its contents to the client via the socket. The client program accepts the name of the output file as a command-line argument. It receives the data from the server and writes them to the specified output file. The client and server programs must terminate after the file transfer completes.

   The contents of the file are sent in chunks of a size that is given as a command-line argument to both the client and server programs. The chunk size can be one of the following values: $\{128, 256, 512, 1024\}$ bytes.

   The socket creation is already implemented in the provided template code. Further, hints for reading an input file and writing to an output file are also included in the code. You need to implement the code for sending data in the server and receiving it at the client.

   Now, we also want the server and client to terminate gracefully after completing their respective tasks. To indicate the end of the transmission, the server should first send the file size before sending the file contents. The client should use this file size information to know when the file has been completely transferred, and exit after that. The file size can be read through the `fstat` system call. This has already been done for you in the provided template file.

   You will receive **3 marks** for successful transfer of a file from server to client, and **1 mark** for ensuring that the client and server programs terminate after the file transfer completes.

   The server code can be compiled and run as follows.

   ```
   gcc socket-server.c -o server
   ./server input.txt 128
   ```

   The client program must be run in a different terminal.

   ```
   gcc socket-client.c -o client
   ./client output.txt 128
   ```

   After the file transfer completes, the content of `input.txt` and `output.txt` should be identical.

2. In this problem, you have to implement three system calls in xv6. You are given test cases to test your system calls with. You must write all the remaining code to implement the system calls, and submit all files you changed in the xv6 folder.

   (a) **[2 marks]** The system call `void getChildren()` should print PIDs of all the children of a calling process, and also print the total number of children.

   (b) **[2 marks]** The system call `void getSiblings()` should print PIDs of all the siblings (i.e., other processes which have the same parent) of the calling process, and also print the total number of siblings.

   (c) **[2 marks]** The system call `void pstree()` should print pstree (process tree) starting from the calling process, i.e., print the PIDs of all descendents of the current process. Further, the print output should be indented differently for the children in different generations, and the number of leading whitespaces in the output should be equal to the depth of the process in the tree. To simplify your code, you can assume that the depth of the family tree is at most 2, i.e., you will only need to worry about printing the children and grandchildren. So, you can use two for-loops to find all descendents, instead of performing a depth-first-search.

   Across all parts, you must use `cprintf` to print output inside the system call implementations in kernel code. All code that needs to access the process table `ptable` structure must be written in `proc.c`, and you can see system calls like `kill` to understand how to access the `ptable`.

   A sample execution of running the test cases given to you is shown below. Your output should roughly match what is given below, and the exact PIDs may differ when you run.

```
$ getchildren
Children PID's are:
4
5
6
No. of Children: 3
$ getsibling
Sibling PID's are:
8
9
10
No. of Siblings: 3
$ pstree
12 [pstree]
 13 [pstree]
  15 [pstree]
 14 [pstree]
```

3. **[4 marks]** In this question you will write code to communicate between two separate processes using shared memory. You are expected to extend the given template programs `producer.c` and `consumer.c`, and submit these files. The producer and consumer programs both map a shared memory segment of size 8 bytes, and repeatedly use these 8 bytes to exchange messages. (Note: even though the OS may allocate a page, we will use only the first 8 bytes for shared memory communication here.)

   The producer first writes into the shared memory segment the 8-byte string "emptyyy" (7 characters plus null termination character) indicating that the shared memory is empty. Then, the producer repeatedly produces 8-byte strings, e.g., "present", and writes them to the same 8-byte slot in the shared memory segment. The consumer must read these strings from the shared memory segment, display them to the screen, and "erase" the string from the shared memory segment by replacing them with the empty string. The consumer also sleeps for some time after consuming each string. The producer and consumer should exchange 10 strings in this manner, and terminate when they are done with the exchange.

   Since there is only one slot in the shared memory segment, the producer will have to reuse the same slot after the string has been consumed and freed up by the consumer. So, the producer must wait till the consumer has consumed the string. The amount of time the consumer sleeps for is taken as an argument from the user when the process is run. Thus, hardcoding in the producer to wait for k seconds before producing the next string would not be sufficient. Instead, you are expected to use signals between producer and consumer to coordinate when a string has been produced or consumed. When the producer writes a string into the memory, it must signal the consumer to consume the particular 8 byte string. Similarly, the consumer must signal the producer once it consumes the string. You must use user-defined signals for this coordination. The signal handlers are initialized in the template code, and you must complete them.

   The template code given to you sets up the shared memory segment between producer and consumer. We also use another IPC mechanism in the beginning of the program to communicate the PIDs of the processes to each other. Please do NOT modify this code for sharing PIDs. The pids are loaded into the variables `pid_consumer` and `pid_producer` respectively. Make sure to utilise these PIDs while sending signals.

   You will be awarded **3 marks** for exchanging and printing 10 strings between producer and consumer, and **1 mark** for gracefully terminating at the end. For example, the producer must not terminate immediately after it sends the last message. It must wait for a signal from the consumer that the consumer is done consuming all the strings only after which it must exit.

   Hint: Make sure to test with various sleep times in the consumer to verify your implementation. An incorrect implementation tends to break down with longer sleep times.

   Sample output is shown below. You must compile and run your producer and consumer programs in different terminals.

The producer is run as shown below.

```
$gcc producer.c –o producer
$ ./producer
Producer started
Data overwritten in shared memory: emptyyy, Index:1
Data overwritten in shared memory: emptyyy, Index:2
Data overwritten in shared memory: emptyyy, Index:3
Data overwritten in shared memory: emptyyy, Index:4
Data overwritten in shared memory: emptyyy, Index:5
Data overwritten in shared memory: emptyyy, Index:6
Data overwritten in shared memory: emptyyy, Index:7
Data overwritten in shared memory: emptyyy, Index:8
Data overwritten in shared memory: emptyyy, Index:9
Data overwritten in shared memory: emptyyy, Index:10
Done
```

The consumer is run as shown below.

```
$gcc consumer.c –o consumer
$ ./consumer
Consumer started
Pid's of producer, consumer communicated!
Enter the time taken to consume a string (in seconds): 1
Data consumed from shared memory: present, Index: 1
Data consumed from shared memory: present, Index: 2
Data consumed from shared memory: present, Index: 3
Data consumed from shared memory: present, Index: 4
Data consumed from shared memory: present, Index: 5
Data consumed from shared memory: present, Index: 6
Data consumed from shared memory: present, Index: 7
Data consumed from shared memory: present, Index: 8
Data consumed from shared memory: present, Index: 9
Data consumed from shared memory: present, Index: 10
Done
```

4. The goal of this question is to implement signals and signal handlers inside xv6. To simplify, we shall assume that there is only one type of signal that can be sent or handled. For this question, you will need to implement the following two system calls.

   - `int signal(void* handler)` takes as argument a pointer to a signal handler function, which must be executed when the process receives the signal.
   - `int sigsend(int pid)` takes as argument the PID of the process to which the signal is to be sent. This signal must be handled by the target (receiving) process just before it returns from a trap the next time. You may assume that each process can track at most one signal at a time, so if a new signal arrives while there is already a pending signal, then the newer signal is discarded.

   Note that the trap may be the result of an interrupt or syscall or any such event. You will simply need to invoke the signal handler before returning from a trap. We will be running this code on 1 CPU so you need not worry about parallel process execution.

   Hint: You may create new fields inside `struct proc` in `proc.h` to keep track of the signal handler function pointer (note that the value of the address of the handler function may be 0), received signals and so on.

   There are two different ways in which the signal handler can be invoked by the process that receives the signal. When returning from trap, the process can either execute the signal handler in the kernel space itself, or after it returns to userspace. These two different implementations will be explored in the two parts of the question below. You are given separate test cases for the two parts below in the template code. You must submit your solution code separately for the two parts.

   (a) **[3 marks] Kernel Mode Execution**: In this part, you will execute the signal handler function directly in the kernel space just before returning from trap into userspace. Of course, if the process has not received a signal or has not registered a signal handler, then it should return from trap as usual. Remember to reset the signal status once you have handled it.

   Hint: If you have a `void func() {// does something;}` and you want to execute it given a `void* func_ptr;`, then you can invoke the function as follows:
   `((void (*)(void))func_ptr)();`

   (b) **[3 marks] User Mode Function Execution**: Executing user-defined functions in kernel mode is never a good idea for obvious reasons. However it is not so straightforward to implement userspace signal handler function execution. You will need to change your privilege level to user level, execute the signal handler function, then trap back to kernel mode, restore the old trapframe and go back to the code that was being executed before the trap. Note that it is important to keep track of the original trap frame before returning from the kernel space to execute the signal handler in user space. This is because, once you trap into the kernel space after finishing execution of the signal handler, the trap frame will be overwritten.

   To simplify the signal handler, let us assume that there is a system call, called `sigreturn()` that must be called at the end of the signal handler function. That is, it is the user's responsibility to place this system call at the end of their signal handling function. This syscall will allow us to return to kernel mode at the end of the execution of the signal handling function. So, signal handlers in this part of the question would look something like this:

```
void signal_handler(){
---signal handler code---
sigreturn();
}
```

With this new system call, we get the opportunity to restore the old trapframe inside the kernel during `sigreturn()`, and return the process to the code that was being executed before the trap, thereby enabling us to execute the signal handler function in user mode.

Hint: Instead of directly calling the signal handling function, save the trapframe of the process somewhere. Perhaps you could define a `struct trapframe* old_tf` inside of `struct proc`, and allocate memory for it in `allocproc` using `p->old_tf = (struct trapframe*)kalloc();`, then use `memmove(char *dest, char *src, int nbytes)` to copy the trapframe. Then, set the eip of the trapframe to signal handler function (you may need to typecast). This old trapframe can then be restored during the `sigreturn()` syscall.

We have given you three test cases each for the two parts. If you are making your own test cases, then note that you should not be using any system calls inside the signal handler function when executing the handler in kernel mode, as they will not work.

A sample execution of the test cases for part (a) is as shown below.

```
$ test-simple
x: 1
$ test-fork
X: 0
x: 1
$ test-syscall
x: 1
```

A sample execution of the test cases for part (b) is shown below. The PIDs printed may differ across executions. You can assume that `esp` always points to the top of the working stack.

```
$ test-fork2
X: 0
Hey, I exist!
x: 1
$ test-siblings
Hey, I exist!
I am 6, I have a misbehaving child 7
I am 7. I am now exitting
$ test-sigreturn
x: 1
val: 3
```