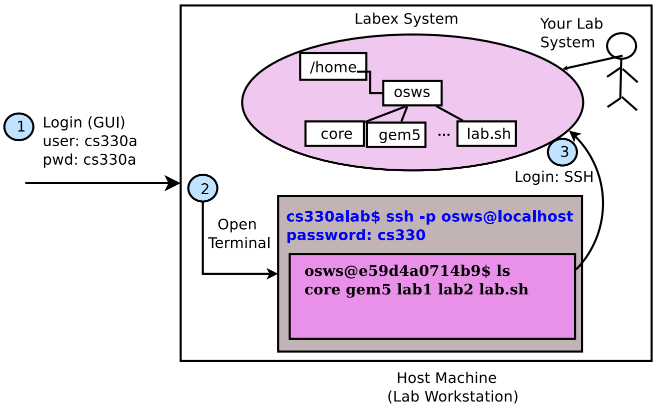
Operating Systems Lab (CS330-2025)
Lab #6

General Instructions

- *Switch off* all electronic devices during the lab.
- Read each part *carefully* to know the restrictions imposed for each question.
- *Do not modify* any of the test cases or evaluation scripts. Modify only the file(s) mentioned.
- Please take the help of the teaching staff, if you face any issues.
- Best of Luck!

Know your environment!



```
cs330alab$ ssh -p osws@localhost
password: cs330
```

```
osws@e59d4a0714b9$ ls
core gem5 lab1 lab2 lab.sh
```

Figure 1: Overview of the Lab environment

When you login into the lab machine, you get a GUI desktop environment. Let us call this the “Host” machine. The lab exercise environment is in a separate sandbox environment in the same machine which you can treat as a separate computer system. Let us call this the “Labex” system. The host machine *is not* the lab setup, rather it hosts the lab exercise system. You may use the host machine to access `man` pages and open exercise document using GUI applications such as the PDF viewer.

Interacting with the “Labex” system

Login into the Labex system

1. Open the terminal application on the Host system. You can also use ‘ALT+CTRL+t’ and ‘ALT+CTRL+n’ to open a new tab in or a new terminal from an existing terminal. *Note that, this shell in this terminal refers to the Host system.*

2. On the terminal, execute `ssh -p 2020 osws@localhost`. It will prompt for the password. Once you have entered the password, you will get the shell to the Labex system.

3. Check the terminal heading or the shell prompt. If you see `osws@`, you are in the Labex system whereas if you see `cs330a@`, you are in the host system.

Exchange files between the Host and Labex systems

1. Open the terminal application on the Host system. Change your directory to "Desktop" (using `cd Desktop` if you want to download/transfer files stored in the host system Desktop.

2. To download `CS330-2025-Lab6.pdf` from Labex to the current directory on the host, run `scp -P 2020 -r osws@localhost:lab-6/labex/CS330-2025-Lab6.pdf ./`. It will prompt for the password. Once you have entered the password, you will see that the file is downloaded.

3. To upload 'abc.c' from the current directory in the host to the Labex system lab-6/labex directory, execute `scp -P 2020 -r abc.c osws@localhost:lab-6/labex/`. It will prompt for the password. Once you have entered the password, you will see that the file is uploaded.

Lab Actions

The current working directory *must be* the home directory of the Labex environment to execute different lab actions through the `lab.sh` utility. Executing `cd /home/osws` or simply `cd`) will take you to the home directory of the Labex environment. The usage semantic of `lab.sh` script is shown below.

```
USE ./lab.sh to initialize the session and get started.

usage:

./lab.sh --roll|-r  <roll1_roll2> --labnum|-n <lab number> --action|-a <init|get|evaluate|
                                                                        prepare|prepare-save|
                                                                        submit|save|reload|
                                                                        detach|swupdate|
                                                                        signoff>

Note your roll number string. Never forget or forge!

        [--action] can be one of the following

        init: Initialize the lab session
        get: Download the assignment
        evaluate: Evaluate the assignment
        prepare: Prepare a submission archive
        prepare-save: Prepare an archive to save
        submit: Submit the assignment. Can be perfomed only once!
        save: Save the assignment
        reload: Reload the last saved solution and apply it to a fresh lab archive
        detach: The lab session is detached. Can be reloaded using 'reload', if supported
        swupdate: Peform software update activities. Caution: Use only if instructed
        signoff: You are done for the lab session. Caution: After signoff, you will not be
                 allowed to submit anymore

EXAMPLE
=======
Assume that your group members have the roll nos 210010 and 211101.
Every lab will have a lab number (announced by the TAs).
Assume lab no to be 5 for the examples shown below.



Fresh Lab? [Yes]
{

    STEP 1        Initialize session  -->  $./lab.sh -r 210010_211101 -n 5 -a init
    STEP 2        Download the lab  -->  $./lab.sh -r 210010_211101 -n 5 -a get

   STEP {3 to L}   ----- WORK ON THE EXERCISE ------

   Completed? [Yes]

        STEP L     Evaluate the exercise --> $./lab.sh -r 210010_211101 -n 5 -a evaluate
        STEP L+1   Prepare submission -->  $./lab.sh -r 210010_211101 -n 5 -a prepare
        STEP L+2   Submit -->  $./lab.sh -r 210010_211101 -n 5 -a submit
        STEP L+3   Signoff --> $./lab.sh -r 210010_211101 -n 5 -a signoff

   Completed? [No]

        STEP L+1   Prepare to save -->  $./lab.sh -r 210010_211101 -n 5 -a prepare-save
        STEP L+2   Save your work  -->  $./lab.sh -r 210010_211101 -n 5 -a save
        STEP L+3   Detach --> $./lab.sh -r 210010_211101 -n 5 -a detach
}
Saved Lab? [Yes]
{
   STEP 1        Reload the lab   -->  $./lab.sh -r 210010_211101 -n 5 -a reload

   STEP {2 to L}   ----- WORK ON THE EXERCISE ------

   Completed? [Yes]

        STEP L    Evaluate the exercise --> $./lab.sh -r 210010_211101 -n 5 -a evaluate
        STEP L+1  Prepare submission -->  $./lab.sh -r 210010_211101 -n 5 -a prepare
        STEP L+2  Submit -->  $./lab.sh -r 210010_211101 -n 5 -a submit
        STEP L+3  Signoff --> $./lab.sh -r 210010_211101 -n 5 -a signoff

   Completed? [No]

        STEP L+1  Prepare to save -->  $./lab.sh -r 210010_211101 -n 5 -a prepare-save
        STEP L+2  Save your work  -->  $./lab.sh -r 210010_211101 -n 5 -a save
        STEP L+3  Detach --> $./lab.sh -r 210010_211101 -n 5 -a detach

}
```

```
***** IMPORTANT ****
- Check the evaluation output
- Make sure you submit before signing off
- Make sure you save the lab before detaching (if you want to continue next)
- Make sure you signoff (STEP L+3) or else you will not get marks and will not get
  the submissions emailed
- Make sure you logout from the system (not just the docker container)

***** CAUTION  *****
DO NOT DELETE lab.sh core or gem5
```

Setup Overview

This lab is designed to get ourselves familiarized with the exception handling behavior of gemOS. The lab environment already contains the *gem5* full system simulator (in the home directory i.e., `/home/osws/gem5`) which will be used to launch/boot gemOS. Once you download the lab using the action as "get", you will see the following directory layout

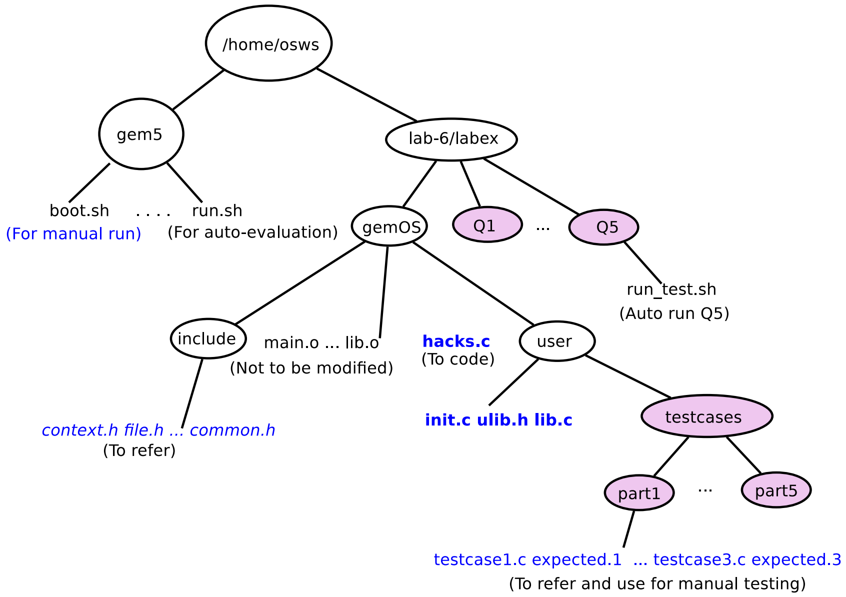


Figure 2: Directory layout of the gemOS lab exercise. The directories shown with colored fill should not be modified in any manner.

You can test your code by executing the gemOS using two approaches—*manual testing* and *automated script-based evaluation.*

Manual Testing

You should use this approach to perform manual checks (with additional debug statements) while you are developing the required functionality. The steps for performing this kind of testing are as follows,

Step 1: Ensure you are logged into the docker environment (using SSH) using two terminals, say T_1 and T_2 (or two tabs of a single terminal).

Step 2: In T_1, the current working directory should be `gem5`. In T_2 the current working directory should be `lab-6/labex/gemOS`.

Step 3: In T_2, run `make` to compile the gemOS. Ensure there are no compilation errors.

Step 4: In T_1, run `./boot.sh /home/osws/lab-6/labex/gemOS/gemOS.kernel`. This should launch the simulator (shell prompt will not come back)

Step 5: In T_2, run `telnet localhost 3456` to see the gemOS output. The gemOS boots into a kernel shell from where the *init* process can be launched by typing in the `init` command.

Step 6: Observe the output in T_2 and try to correlate with the code in `user/init.c`. For test cases, you can find the expected output in the same location as the test case file (see Step 7). Type `exit` to shutdown the OS which will also terminate the `gem5` instance executing in T_1.

Step 7: In T_2, perform the necessary changes (as required for the exercise) in the designated files. For this lab exercise, you are required to incorporate changes in `hacks.c` and `user/init.c`. The `user/init.c` contains the code for the first user space process i.e., *init* process. While you can write any code in this file and test it, to test a particular testcase for any part you should copy the testcase file to over-write the `user/init.c` file. For example, to test your implementation against testcase one for `Q1`, you need to copy the `user/testcases/part1/testcase1.c` to `user/init.c`.

Step 8: Repeat Steps 3-7 every time you change your code or testcase.

Automated evaluation

Note: Remove additional debug prints before performing automated evaluation

This step can be performed either completing each part of the exercise or before making a final submission. To test a particular part (say Q1), change your current working directory to `lab-6/labex/Q1`. Execute `./run_tests.sh`. After completion of the script, the output produced will be stored in the `output` directory. To evaluate the complete exercise, use the usual procedure of executing an "evaluate" action using the `lab.sh` script.

Exercise Overview

In this assignment, you are required to understand and change the exception handling behavior of gemOS. *Importantly*, you need to answer the questions in `gemOS/qns.txt`. In this lab, we want to handle the division-by-zero exception in different ways (specified in `include/hacks.h`). There are two related aspects to meet the objectives of different parts in this lab exercise.

1.System call. To configure the "OS handler behavior" (e.g., skip the div instruction etc.) when division-by-zero exception occurs, we are required to implement a system call `config_hack_semantics`. The user space logic for the system call is already implemented in the given template. You are required to write the logic for `sys_config_hs` in the `hacks.c` file.

long sys_config_hs(struct exec_context *ctx, long hack_mode, void *uhaddr)

ctx: This is the PCB of the process that invoked the system call

hack_mode: Specifies the behavior of the division-by-zero handler. For each part the value of this parameter will be different. The possible values are defined in a enum (`DIV_ZERO_*`) in the `include/hacks.h` header file (and in the `user/ulib.h` for user space).

uhadr: Address of an user space function. This is relevant for hack_mode = DIV_ZERO_USH_EXIT. For others, it should be ignored. The `uhaddr` should be checked to be a valid code address for the *relevant* command types.

While handling the system call, you need to check if the `hack_mode` value is valid. Furthermore, you need to check if the `uhaddr` value is a valid code address *only for* hack mode value

DIV_ZERO_USH_EXIT. If any of the checks fail, return -EINVAL after setting the current exception handling behavior as invalid. After performing the checks, you need to save the hack_mode and uhaddr values into the hconfig struct (already defined in hacks.c file).

2. Div-by-zero handler. To manipulate the user state (captured in the regs parameter) in the OS exception handler for divide-by-zero (do_div_by_zero defined in hacks.c). The manipulation of the user state is based on the current configuration of the exception handling behavior which is saved into the hconfig structure during the system call. Different parts in this exercise requires implementation of different behaviors when division-by-zero occurs in user space. The regs parameter is of type struct user_regs which is defined in include/context.h.

int do_div_by_zero(struct user_regs *regs)

regs: The user execution state can be accessed/modified using this parameter. Note that, the modifications will be applied to the CPU registers at the time of returning to the user space.

For different parts of this lab, you are required to implement the logic for division by zero handling in this function.

Q1. Changing Register Operands [18 Marks]

If the cur_hack_config in hconfig structure is set to DIV_ZERO_OPER_CHANGE, you are required to change the user space registers such that the division by zero is fixed. For this part, whenever a division-by-zero occurs, you need to find the registers involved in the 'div'/'idiv' for the given testcases. Finding the involved registers would require a *manual run* with the following steps,

- Copy the test case from user/testcases/part1/ onto user/init.c.
- Build gemOS using 'make'.
- Run objdump -D user/init.o to see the assembly.
- Locate the division instruction in the main function.
- Note down the operand register to 'div' instruction. Please note that rax is as implicit register in many x86 instructions.

Once you figured out the numerator and denominator registers (you can also use 'printk' in kernel to validate) make sure the *result of division* comes out to be zero. Note that, you are required to check that if the current exception handling behavior invalid, you should print *Error...exiting* before invoking do_exit(0) in the handler.

Testing: For this part, you may use the testcases in the user/testcases/part1/ directory. *NOTE Do not forget to answer the questions based on your observations (see* qns.txt*).*

Q2. Skipping the instruction [18 Marks]

If the cur_hack_config in hconfig structure is set to DIV_ZERO_SKIP, you are required to change the user instruction execution behavior by changing the appropriate user register. For this part, whenever a division-by-zero occurs, you need to find the size of the instruction using the following steps.

- Copy the test case from user/testcases/part2/ onto user/init.c.
- Build gemOS using 'make'.
- Run objdump -D user/init.o to see the assembly.
- Locate the division instruction in the main function and note down the size.

Once you figured out the size of the instruction you need to skip the division instruction. Note that, you are required to check that if the current exception handling behavior invalid, you should print *Error...exiting* before invoking `do_exit(0)` in the handler.

Testing: For this part, you may use the testcases in the `user/testcases/part2/` directory. *NOTE Do not forget to answer the questions based on your observations (see* `qns.txt`*).*

Q3. An exiting user space handler [24 Marks]

If the `cur_hack_config` in `hconfig` structure is set to `DIV_ZERO_USH_EXIT` (previously through the `config_hack_semantics` system call), you are required to change the user instruction execution behavior such that the control will resume in the user function at address `hconfig.usr_handler_addr`. The user function is expected to invoke the `exit` system call in the end and makes use of *at most* one argument which is the code address where the division by zero occurred. Note that, you are required to check that if the current exception handling behavior invalid, you should print *Error...exiting* before invoking `do_exit(0)` in the handler.

Testing: For this part, you may use the testcases in the `user/testcases/part3/` directory. *NOTE Do not forget to answer the questions based on your observations (see* `qns.txt`*).*

Q4. Skipping the culprit function [40 Marks]

If the `cur_hack_config` in `hconfig` structure is set to `DIV_ZERO_SKIP_FUNC` (previously through the `config_hack_semantics` system call), you are required to change the user instruction execution behavior such that the control will resume in the user mode *bypassing* the execution of the remaining part of the function where *division by zero* occurred. For example, if the function call sequence is $main \rightarrow func1 \rightarrow func2$ where in the middle of *func2* a division by zero takes place, the remaining part of *func2* should not be executed and the control should resume in *func1* at the point of return. Please refer to the `user/testcases/part4/testcase1.c` and `user/testcases/part4/expected.1` as an illustration. You are required to access/manipulate the user stack and other registers to achieve this functionality. Note the following stack state when a *func2* is called from *func1*.

```
|              |
|              |
|------------  |   <----- rsp
|   locals     |
|   for func2  |
|    ...       |
--------------   <------rbp
|  saved rbp   |
 --------------
|RetAddr(func1)|
----------------
|              |
|  Stack Frame |
|  for func1   |
|   ...        |
```

You can also use `objdump` for the `user/init.o` for any of the testcases in a manual mode (after compilation). Note that, the expected return value of the skipped function is `1` which you should ensure.

Testing: For this part, you may use the testcases in the `user/testcases/part4/` directory. *NOTE Do not forget to answer the questions based on your observations (see* `qns.txt`*).*