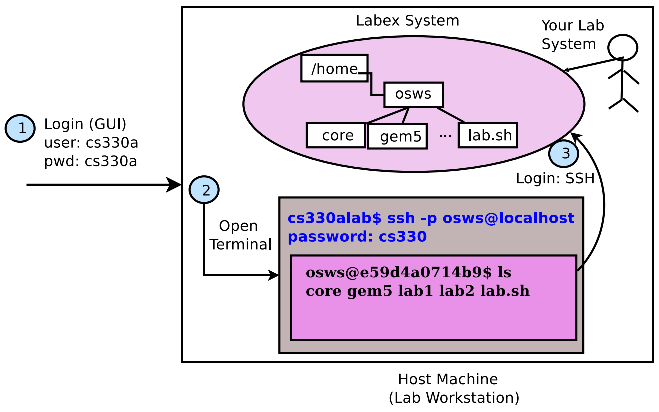
Operating Systems Lab (CS330-2025)
Lab #5

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-Lab5.pdf` from Labex to the current directory on the host, run `scp -P 2020 -r osws@localhost:lab-5/labex/CS330-2025-Lab5.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-3/labex directory, execute `scp -P 2020 -r abc.c osws@localhost:lab-5/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 gemOS ecosystem. 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

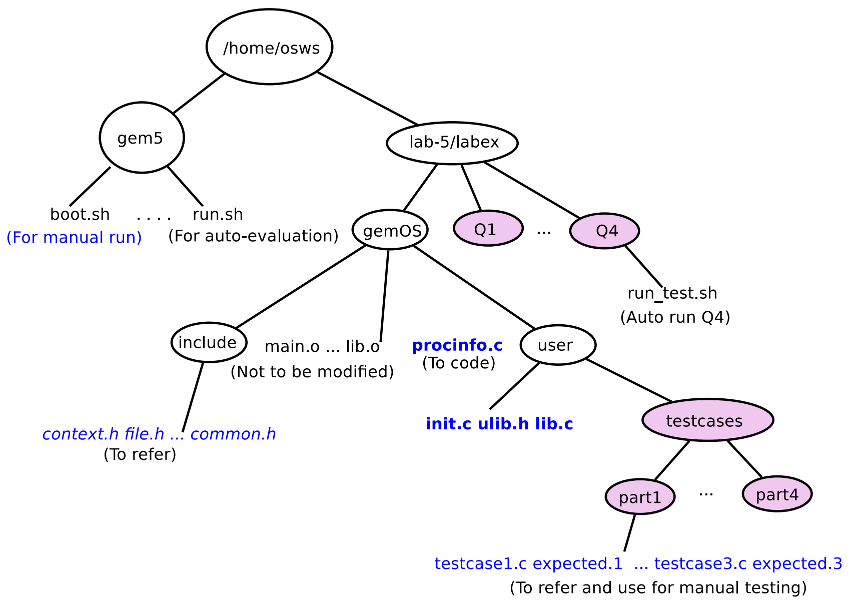


Figure 2: Directory layout of the gemOS lab exercise. The directories shown with colored fill should not be modified in any manner.

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 Labex 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-5/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-5/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 `procinfo.c`, `user/ulib.h`, `user/lib.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-5/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 implement the handler for a system call to extract some of the information related to file and address space from the PCB. The help material can be accessed using the following link: `http://172.27.21.215` As shown in the Figure 2, you are required to primarily modify the `procinfo.c` file in the OS side. To refer to the definitions of different structures and macros, refer to the below mentioned OS header files from the `include` directory. In the user space (i.e., the `user` directory), you are primarily required to fill-up the code for the `procinfo` library function in the `user/lib.c`. The header file for the user space is the `user/ulib.h` file.

The PCB in gemOS is represented by the `struct exec_context` in the gemOS header file `include/context.h`. The relevant fields of the structure is shown below,

```
struct exec_context{
      u32 pid;   //Process ID
      u32 ppid;  //Parent process ID
        ...
      struct mm_segment mms[MAX_MM_SEGS]; //Linear segments
      struct vm_area* vm_area;  //MMAPed segments
      char name[CNAME_MAX];     //Process names
        ...
      struct file* files[MAX_OPEN_FILES]; //files[i] represents fd = i
        ...
}
```

Inside the gemOS, You are required to implement the handler for the system call `SYSCALL_PROC_INFO` (defined in `include/entry.h` and `user/ulib.h`) by modifying the template implementation given in the function `get_process_info`. The details of the function parameters and expected return values are described below.

long get_process_info(struct exec_context *ctx, long cmd, char *ubuf, long len)

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

cmd: Specifies the type of information to be collected. For each part the value of this parameter will be different. The possible values are defined in a enum (`GET_PINFO_*`) in the `include/procinfo.h` header file (and in the `user/ulib.h` for user space).

ubuf: A pointer to the user space memory as passed from the user space; which will be filled up with the required information based on the **cmd** type.

len: The length of the user space memory as passed from the user space.

Note that the above handler is called from the generic system call handler after saving the user state. Therefore, your code is not required to handle anything related to saving/restoring the user execution state.

In-built functions that you may want to use (for debugging and implementation) for OS mode implementation are given below.

```
- printk {usage: printk("Hello My value is %d\n", val);}
- strcpy {usage: strcpy(dst, src), src and dst are strings (with terminating '\0')}
- memcpy {usage: memcpy(dst, src, len)}
```

Q1. General Information [20 Marks]

From the user space, to get the general process information, one passes the **cmd** value as `GET_PINFO_GEN` along with other arguments. The information required to be filled up to serve this request from the user space is specified in the `struct general_info` defined in `include/procinfo.h` (and in `user/ulib.h` for user space). The information should be filled up in the **ubuf** maintaining the format of the structure. Note that, you are required to check that the **ubuf** argument is not NULL and the **len** passed from the user space is sufficient to hold the information. If the check fails, you should return `-EINVAL`. If the system call is handled successfully, return 1.

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. File information [35 Marks]

From the user space, to get the file descriptor usage information, **cmd** value is passed as `GET_PINFO_FILE` along with other arguments. The information required to be filled up to serve this request from the user space is specified in the `struct file_info` defined in `include/procinfo.h` (and in `user/ulib.h` for user space). The `files` member of the `struct exec_context` is an array of pointers to the file object where if `files[pos] != NULL`, then it represents the file object for the descriptor FD such that `FD = pos`. The `struct file` is defined in the `include/file.h`. The file types (e.g., `STDIN`, `STDOUT` etc.) is defined in the same header file and the corresponding names are defined in the `file_types` variable in the `procinfo.c` file which can be used to fill the `file_type` field. The information should be filled up in the **ubuf** as an array of structures (of type `struct file_info`). The total number of structures that will be packed into **ubuf** is the total number of open file descriptors for the process. Note that, you are required to check that the **ubuf** argument is not NULL and the **len** passed from the user space is sufficient to hold the information for *all used file descriptors*. If the check fails, you should return `-EINVAL`. If the system call is handled successfully, return the *number of open files* whose information is packed into the user buffer. Note that, you should return `-EINVAL` if the `len` is not sufficient to hold information for *all open files*.

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. Memory segment information [20 Marks]

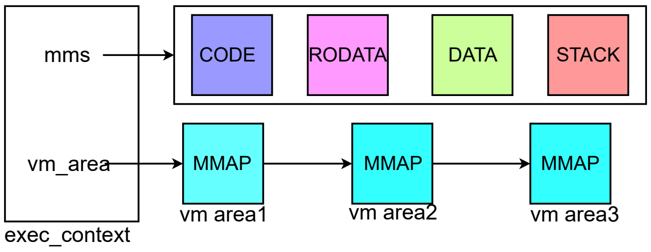


Figure 3: Address space information maintenance in gemOS PCB.

Figure 3 shows the address space layout meta-data in execution context (`struct exec_context`) in (`include/context.h`). The execution context contains an array (`struct mm_segment mms`) of different linearly maintained memory segments such as *CODE*, *DATA*, *STACK* etc. (defined as an enum in `include/context.h`). The virtual memory areas (`vm_area`) allocated using dynamic memory allocation system call (e.g., `mmap()`) are maintained using a list of VM areas. The head of this list is maintained as a member variable (`vm_area`) in the `exec_context` as shown in Figure 3. In this part, we will handle the information related to memory segments while in the next part, we will handle the VM areas.

From the user space, to get the memory segment usage information, **cmd** value is passed as `GET_PINFO_MSEG` along with other arguments. The information required to be filled into the user buffer to serve this request from the user space is specified in the `struct mem_segment_info` defined in `include/procinfo.h` (and in `user/ulib.h` for user space). The `segname` field is to be filled by using the strings defined in the `segment_names` array in the provided `procinfo.c` template. The `perm` corresponds to the access permissions for the segment and should be filled as a string representation `RWX`. If any permission is missing for a segment, it should be encoded using _. For example, a segment with read and execute permission, should be encoded as `R_X`. Permission is maintained in the `access_flags` member of `struct mm_segment` as a bit-wise OR of `PROT_READ`, `PROT_WRITE` and `PROT_EXEC`.

The information should be filled up in the **ubuf** as an array of structures (of type `struct mem_segment_info`). The total number of structures that will be packed into **ubuf** is the total number of segments. Note that, you are required to check that the **ubuf** argument is not NULL and the **len** passed from the user space is sufficient to hold the information for *all memory segments*. If the check fails, you should return `-EINVAL`. If the system call is handled successfully, return the *number of segments* whose information is packed into the user buffer. Note that, you should return `-EINVAL` if the `len` is not sufficient to hold information for *all segment information*.

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. VM [25 Marks]

The virtual memory areas (`vm_area`) allocated using dynamic memory allocation system call `mmap()` are maintained as a list of VM areas where the `vm_area` member (`struct vm_area *vm_area`) in exec context points to the head of the list as shown in Figure 3. *Note that the first VMA in the list of VMAs is a dummy VMA (of size 4KB) created for easier management. You should discount this VMA from all processing in this exercise.*

From the user space, to get the VMA usage information, **cmd** value is passed as `GET_PINFO_VMA` along with other arguments. The information required to be filled into the user buffer to serve this request is specified in the `struct vm_area_info` defined in `include/procinfo.h` (and in `user/ulib.h` for user space). The `perm` corresponds to the access permissions for the segment and should be filled as a string representation `RWX`. If any permission is missing for a segment, it should be encoded using _. For example, a segment with read and execute permission, should be encoded as `R_X`. Permission is maintained in the `access_flags` member of `struct vm_area` as a bit-wise OR of `PROT_READ`, `PROT_WRITE` and `PROT_EXEC`.

The information should be filled up in the **ubuf** as an array of structures (of type `struct vm_area_info`). The total number of structures that will be packed into **ubuf** is the total number of VM areas (excluding the dummy VMA). Note that, you are required to check that the **ubuf** argument is not NULL and the **len** passed from the user space is sufficient to hold the information for *all used VM areas* (excluding the dummy VMA). If the check fails, you should return `-EINVAL`. If the system call is handled successfully, return the *number of VMAs* whose information is packed into the user buffer. Note that, you should return `-EINVAL` if the `len` is not sufficient to hold information for *all VM area information.*

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`*).*