Operating Systems Lab (CS330-2025)
Lab #9

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!

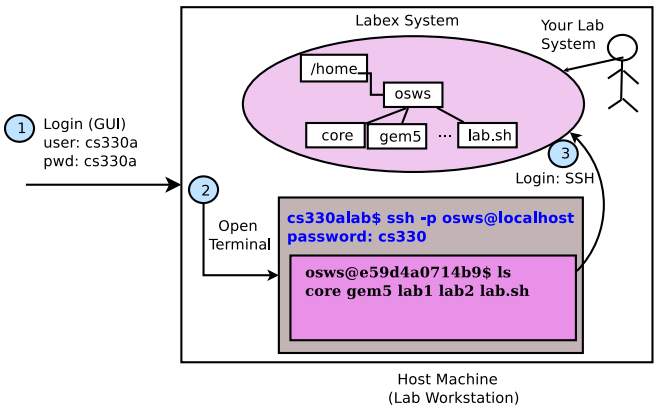

Figure 1: Overview of the Lab environment

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-Lab9.pdf` from Labex to the current directory on the host, run `scp -P 2020 -r osws@localhost:lab-9/labex/CS330-2025-Lab9.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-9/labex directory, execute `scp -P 2020 -r abc.c osws@localhost:lab-9/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.

./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 file API subsystem 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

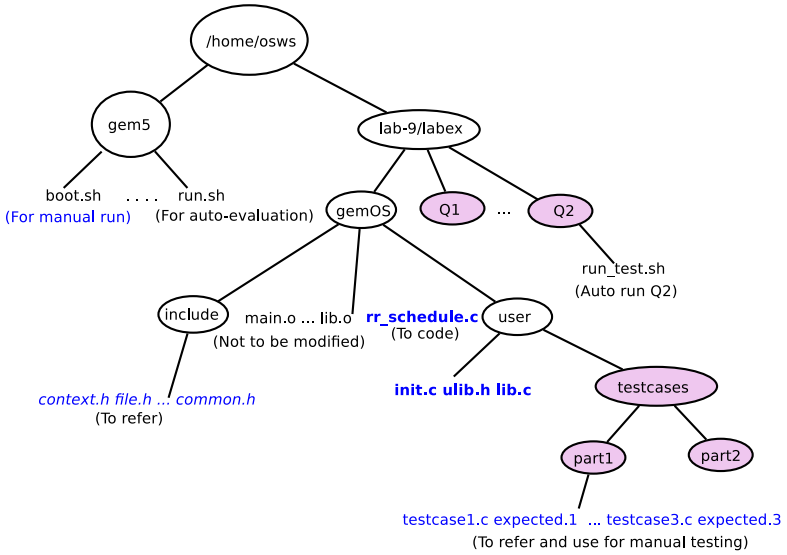

Figure 2: Directory layout of the gemOS lab exercise. The directories shown with colored fill should not be modified in any manner.

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-9/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-9/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

Helper macros and functions in GemOS

The OS infrastructure does not use (and link with) the standard *C* library functions and therefore, you can not invoke known *C* functions while writing the OS and user mode code. For user mode (`user/init.c`), you can invoke all extern functions in `user/ulib.h`. While changing/adding code in OS mode, you should not even use the functions available in user space. Therefore, we provide some commonly required OS functionalities while doing the assignment.

- **Getting PCB of the current process**: Use `get_current_ctx()` to get the `exec_context` corresponding the current process.
  Example usage: `struct exec_context *ctx = get_current_ctx();`
- **Getting PCB of a process with particular pid**: Use `get_ctx_by_pid(u32 pid)` to get the `exec_context` corresponding a process with particular pid.
  Example usage: `struct exec_context *ctx = get_ctx_by_pid(0);`
  (`exec_context` of the process with pid 0 is returned)
- **Printing output:** You may need to output some debug messages into the console. You can use `printk` function for OS mode. The `printk` function should be used just like a `printf` but the format specifier support is minimal.
- **Physical addr to virtual address:** Use `osmap(u64 pfn)` function to find virtual address corresponding a physical page frame number (PFN).
  Example usage: `u64 *vaddr_base = (u64 *)osmap(ctx->pgd);`
- **Useful Macros:** To extract 9 bits (from a given virtual address) needed to index into a PGD level page, use `PGD_MASK` and `PGD_SHIFT` macros.
  Example usage: `((virtual_address & PGD_MASK) >> PGD_SHIFT)`
  Similar macros exist for PUD, PMD and PTE levels also. Refer *gemOS/include/page.h*.

Q1: Scheduling Policy [50 marls]

In this part of the lab, you will be modifying the scheduling policy used by gemOS. You will be introducing **Round-Robin scheduling** policy in gemOS. To achieve this, you have to provide implementation for the following **kernel functions** present in *gemOS/rr_schedule.c*,

- `void rr_add_context(struct exec_context *ctx)`
- `void rr_remove_context(struct exec_context *ctx)`
- `struct exec_context *rr_pick_next_context(struct exec_context *ctx)`

struct exec_context *rr_list_head

`rr_list_head` is the head of the linked list maintained to implement round robin scheduling (referred to as *RR-list* in further discussions). `rr_list_head` is a global variable and can be directly accessed in any kernel function. Initially, *RR-list* list is empty i.e. `rr_list_head` is initialised to **NULL**. *RR-list* should contain processes in **Ready/Running** state only. The gemOS PCB i.e., `exec_context` contains a pointer (`struct exec_context *next`) that can be used to maintain the *RR-list*.

```
struct exec_context{
        u32 pid;
        u32 ppid;
        ....
        struct exec_context *next;
};
```

void rr_add_context(struct exec_context *ctx)

Arguments: ctx: `exec_context` of the process to the added in the *RR-list*.
Description: This function is called when a new process is created (*init is created*, child is created after `fork()`) or when a process wakes up from sleep. In this function, add the process (whose exec_context is passed as the argument to this function) at the end/tail of *RR-list*.
Return value: This function does not return any value.

void rr_remove_context(struct exec_context *ctx)

Arguments: ctx: `exec_context` of the process to the removed from the *RR-list*.
Description: This function is called when a process exits (after `exit()`) or when a process goes to sleep (after `sleep()`) In this function, remove the process (whose exec_context is passed as the argument to this function) from the *RR-list*.
Note: This function is called before `rr_pick_next_context()` is called. Therefore, *do not modify* the `next` pointer of `ctx`.
Return value: This function does not return any value.

struct exec_context *rr_pick_next_context(struct exec_context *ctx)

Arguments: ctx: `exec_context` of the currently running process.
Description: This function is called to choose the next process to run. It is called when a process exits (after `exit()`) or when a process goes to sleep (after `sleep()`) or when a timer interrupt comes (i.e. time slice allocated to the currently running process is over). In this function, pick the process queued after the currently running process in the *RR-list*. If end of the *RR-list* is reached and no process is found, start looking for the process from the head of the *RR-list*. If there is only one process in the *RR-list*, and the function is called because of a timer event, pick the same process again for scheduling. If the *swapper process* is the currently

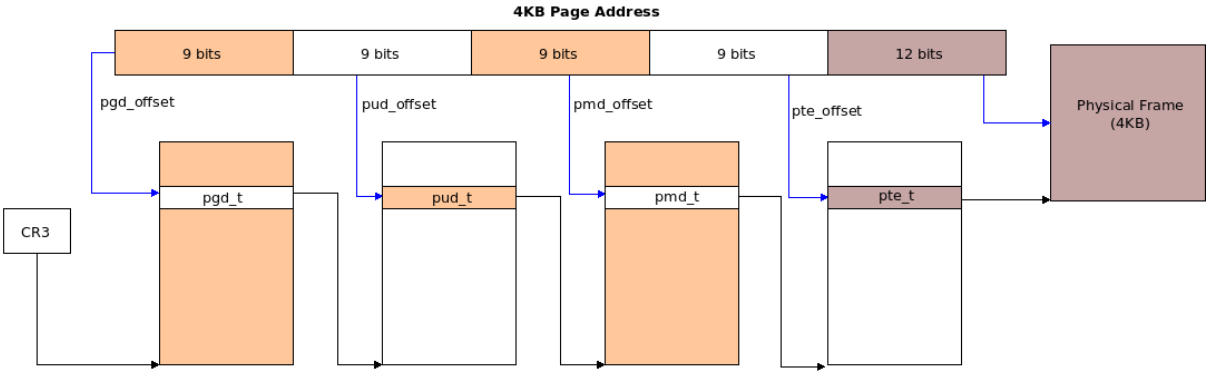

Figure 3: x86-64 4 Level Page Table

running process, pick the process at the head of the *RR-list* to run. If the *RR-list* is empty, pick the *swapper process* (process with pid == 0) to run.
Note: Swapper process is **not** *part of the RR-list.*
Return value: Return the pointer to the `exec_context` of the process chosen by this function to run.

Testing

For this part, you may use the testcases (1 to 5) in the `user/testcases/part1/` directory. **Do not run automated test for this part without manual run. The script will hang.** *Note: Do not forget to answer the questions based on your observations (see* `qns.txt`*).*

Q2. Examine the Page Table [50 marks]

In this part of the lab, you will provide implementation for the handler (`do_walk_pt()`, present in *rr_schedule.c*) for the system call (`walk_pt(address)`. The user-space linking to the system call handler is done and you are required to print the page table entries for any given virtual address in the specified format mentioned below (also in the test-cases) by providing suitable logic in the `do_walk_pt()` function. GemOS uses 4-level page table for a 48-bit virtual address to implement virtual addressing. Figure 3 shows page table layout for 48 bit virtual to physical address translation for a 4KB page.

int do_walk_pt(struct exec_context *ctx, unsigned long addr)

Argument: ctx: `exec_context` of the currently running process.
Argument: addr: Virtual address of the currently running process corresponding which the page table entries needs to be printed.
Description: In this function, walk the page table from the OS and print the page table entries present at each level of the page table. To access the first level (L_1) of the page table, use the `pgd` field in the `exec_context` which is the PFN number for the L_1 page table. If bit 0 is set in a page table entry, it indicates that the page table entry is valid else page table entry is invalid.

While walking the page table, if a page table entry is valid, print the following information about the entry:

```
1) Virtual address at which page table entry is present
2) Contents of the page table entry
3) Physical address of the next level page present in the page table entry
4) Flags present in the page table entry
```

While walking the page table, if a page table entry is invalid, print 'No L_i entry' message ('i' indicates the page table level) for the current and the subsequent page table levels and return.

Note:

- L_1 implies PGD level of the page table, L_2 implies PUD level, L_3 implies PMD level and L_4 implies PTE level of the page table.
- Assume that bits 0 to 11 of a page table entry store the flags and bits 12 to 43 store the physical address of the next level page of the page table.

Sample output 1:

```
L1-entry addr: 0x2093000, L1-entry contents: 0x2094027, Physical addr: 0x2094, Flags: 0x27
L2-entry addr: 0x20940F8, L2-entry contents: 0x2097027, Physical addr: 0x2097, Flags: 0x27
L3-entry addr: 0x2097FF8, L3-entry contents: 0x2098027, Physical addr: 0x2098, Flags: 0x27
L4-entry addr: 0x2098FF8, L4-entry contents: 0x6411007, Physical addr: 0x6411, Flags: 0x7
```

Sample output 2:

```
L1-entry addr: 0x2093000, L1-entry contents: 0x2094027, Physical addr: 0x2094, Flags: 0x27
L2-entry addr: 0x2094030, L2-entry contents: 0x2095007, Physical addr: 0x2095, Flags: 0x7
No L3 entry
No L4 entry
```

Return value: Return 0 on success and -1 in case of any error.

Testing

For this part, you may use the testcases (1 to 5) in the `user/testcases/part2/` directory.
Note: Do not forget to answer the questions based on your observations (see `qns.txt`*).*