Operating Systems Lab (CS330-2025)
Lab #11

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!

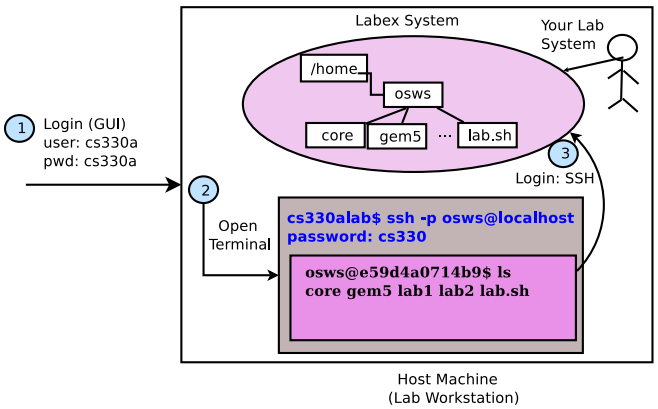

Figure 1: Overview of the Lab environment

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-Lab11.pdf` from Labex to the current directory on the host, run `scp -P 2020 -r osws@localhost:lab-11/labex/CS330-2025-Lab11.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-11/labex directory, execute `scp -P 2020 -r abc.c osws@localhost:lab-11/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

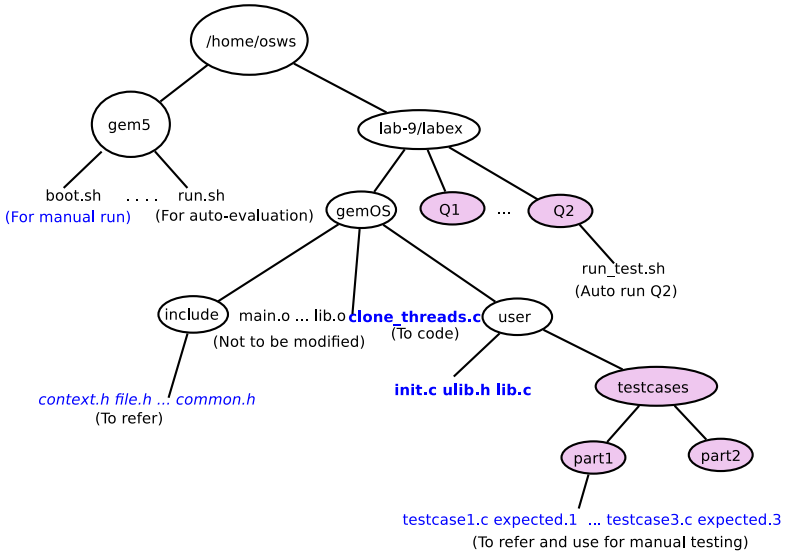

Figure 2: Directory layout of the gemOS lab exercise. The directories shown with colored fill should not be modified in any manner.

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-11/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-11/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)
- **Getting a new PCB**: Use `get_new_ctx()` to get an `exec_context` for creating a new process.
  Example usage: `struct exec_context *new_ctx = get_new_ctx();`
- **Freeing a PCB**: Use `release_context()` to free an `exec_context`, for example, on the termination of a process.
  Example usage: `release_context(new_ctx);`
- **Scheduling a new process**: Use `pick_next_context()` and `schedule()` to schedule a new process.
  Example usage: `new_ctx = pick_next_context(ctx); schedule(new_ctx);`
  Note: Argument passed to `pick_next_context()` is the pointer to the execution context of the process that calls `pick_next_context(ctx)`. Return value of `pick_next_context()` is a pointer to the execution context of the process chosen to be scheduled next. This execution context is passed to the `schedule()`.

- **Physical addr to virtual address:** Use `osmap(u64 pfn)` function to get the OS virtual address corresponding a physical frame.
  Example usage: `u64 *vaddr_base = (u64 *)osmap(ctx->pgd);`
- **Allocating memory:** Use `os_alloc` to allocates a memory region of `size` bytes. Note that you *can not* use this function to allocate regions of size greater than 2048 bytes.
  Example usage: `struct vm_area *vm = os_alloc(sizeof(struct vm_area));`
- **Deallocating memory:** Use `os_free` function to free the memory allocated using `os_alloc`.
  Example usage: `os_free(vm, sizeof(struct vm_area));`

Q1. Threads of Execution! [35marks]

A thread is a light weight process which can be created by a process and will execute some part of the code segment of the parent process that has been assigned to the thread. When a thread is created, the thread uses the same virtual address space of the parent process. The creating process allocates a part of the virtual address space for the thread to use it as the stack.

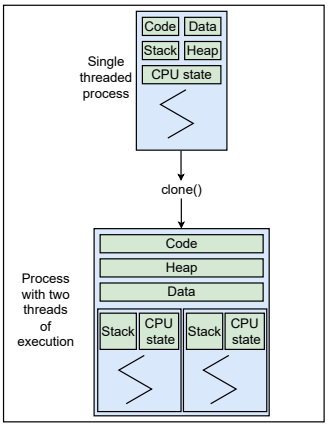

Figure 3: Effect of clone system call

In gemOS, we represent the threads using `struct exec_context` just like a process. For accounting of threads, the parent process uses a reference to the structure called `ctx_thread_info` (Refer `include/context.h` and `include/clone_threads.h`) which is initialized on the first clone request (part of the template code `clone_threads.c`). Note that, the threads have their own register state (inside the PCB) and stack while they share the same address space of the process. Just like `fork`, when a new thread is created most of the contents of the PCB are copied onto the new thread's PCB.

In this task, you have to provide implementation for the following **system call handlers** present in *gemOS/clone_threads.c*,

- `long do_clone(void *th_func, void *user_stack, void *user_arg)`
- `void do_exit(u8 normal)`

Assumptions

- You can assume there is no invocation of fork system call in this part of the assignment.
- If a thread exits, only that thread will be terminated. If the parent exits, all threads are also terminated.

- Sanity checks on the function address and stack address arguments passed to the `do_clone()` are not required to be performed.
- The prototype of the function that is being passed to the `do_clone` is fixed i.e., `void * function_name (void *args)`.
- Maximum number of threads alive at any point will be four (`MAX_THREADS`).

long do_clone(void *th_func, void *user_stack, void *user_arg)

th_func is the function pointer (address in the code segment) that will be executed once the thread returns to the user space.
user_stack is the starting address of the stack used by the newly created thread. Please note that the stack grows from higher address to the lower address.
user_arg is a pointer to the argument passed to the thread function. Note that, the thread function signature is fixed and only one argument is passed to the thread function.
Description: This system call handler (present in *gemOS/clone_threads.c*) should create a new child thread where most of the things are copied from the parent's context as most of them are pointers and are shared between parent process and child thread. As we use the `struct exec_context` to represent both process and threads, there is no distinction between a process and thread representation in the OS. However, the `type` field of the `struct exec_context` is used to distinguish them explicitly. Therefore, the type for the new context must be set to `EXEC_CTX_USER_TH`. Moreover, when a new context is allocated using `get_new_ctx()`, the state of the context is set to "UNUSED". You need to update the state of the thread to make it schedulable.

Each process has to keep track of the threads created by it. `struct exec_context` contains a member called `ctx_threads` that can be used for this purpose.

```
    struct exec_context{
            u32 pid;
            u32 ppid;
            .
            .
            struct ctx_thread_info *ctx_threads;
};
```

`ctx_threads` belonging to the parent process should be updated accordingly when a new thread is created. Assume that a thread cannot create further threads. Hence, `ctx_threads` member of the threads `exec_context` should be NULL.

The thread being created has to be initialized with its own userspace stack which has been passed as an argument to the `do_clone()`. Likewise, thread has to start execution from the function whose address is passed as an argument to the `do_clone()`. Therefore, relevant registers in the `exec_context` (`new_ctx->regs`) must be updated such that the thread uses specified userspace stack and executes the specified userspace function with the required argument.

Return value: This function returns the pid of the newly created thread on success and -1 on error.

void do_exit(u8 normal)

normal argument can be ignored for this part.
Description: This system call handler is called when a process/thread triggers `exit()` system call. A template implementation of this handler is already provided in commented form in *gemOS/clone_threads.c*. This template implementation works fine for a system that does

not support the `clone()` system call. In this part, you have to modify this template implementation so that it can handle the exiting of both threads and processes. Helper functions `handle_thread_exit()` and `cleanup_all_threads()` (defined in *gemOS/clone_threads_helpers.c*) can be used for this purpose.

Testing

For this part, you may use the testcases (1 to 5) in the `user/testcases/part1/` directory. *Do not run automated test (specifically for testcase#5) in this part without manual run. The script will hang. Do not forget to answer the questions in qns.txt.*

Q2. Order, Order! Lock the culprit [35 marks]

Semaphores allow processes and threads to synchronize their actions. A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one (`sem_post()`); and decrement the semaphore value by one (`sem_wait()`). If the value of a semaphore is currently zero, then a `sem_wait()` operation will block until the value becomes greater than zero.
In this task, you have to provide implementation for the following **system call handlers** present in *gemOS/clone_threads.c*,

- `int do_sem_init(struct exec_context *current, sem_t *sem_id, int value)`
- `int do_sem_wait(struct exec_context *current, sem_t *sem_id)`
- `int do_sem_post(struct exec_context *current, sem_t *sem_id)`

Assumptions

- You can assume that semaphores created by a process will be shared among the threads of that process only and not across different processes.
- Maximum number of semaphores created by a process will be 8 (`MAX_LOCKS`).
- You can assume that `sem_init()` will be called only once for each semaphore.

Relevant structures

```
struct semaphore
{
        unsigned long value;
        struct exec_context *wait_queue;
        struct spinlock lock;
};
```

Each semaphore is represented by the above structure. **value** represents the current value of the semaphore. It can increase on `sem_post()` and decrease on `sem_wait()`. **wait_queue** is a linked list of execution contexts of the processes waiting to acquire the semaphore. **lock** is a spinlock that protects the **value** and **wait_queue** members of the semaphore. All modifications to these members should be done after acquiring the spinlock.

```
struct exec_context
{
    u32 pid;
    u32 ppid;
```

```
struct lock
{
    struct semaphore sem;
    unsigned long id;
```

```
        .                                       int state;
        .                                   };
        struct lock *lock;
        struct exec_context *next;
    };
```

Execution context of each process maintains information about the locks associated with the process in `struct exec_context`. This is achieved by maintaining a pointer (`struct lock *lock;`) to an array of `struct lock` structures. Size of this array is `MAX_LOCKS`. Each `struct lock` structure in the array stores a semaphore (**sem**) and the metadata associated with the semaphore. **id** within the `struct lock` is the unique identifier associated with the semaphore which works as a handle for the user space for subsequent operations on the semaphore. **state** within the `struct lock` represents whether the semaphore **sem** is in use or not. It can have two values: `LOCK_USED, LOCK_UNUSED`. For example, if a process creates only one semaphore, then, only one `struct lock` structure associated with the process will have the state as `LOCK_USED` and the remaining `MAX_LOCKS - 1` structures associated with the process will have the state as `LOCK_UNUSED`.

System call handlers to implement

`int do_sem_init(struct exec_context *current, sem_t *sem_id, int value)`

current is a pointer to the execution context of the process calling the `sem_init()` system call
sem_id is the unique identifier associated with the semaphore being initialised.
value is the intial value of the semaphore.
`sem_init()` system call is called to initialise a semaphore. Code to perform the creation of the array of `struct lock` structures on the initialisation of the first semaphore of a process is already included in the provided template. Your task is to perform lookup to find an unused `struct lock` structure and initialise the members of the structure. Helper functions (present in *gemOS/clone_threads.c*) `spin_init()`, `spin_lock()` and `spin_unlock()` can be used if needed.
Note: You need to inherit the semaphore state in all threads which may require some changes in the `do_clone` *implementation.*

Return value: This function should return 0 on success and `-EAGAIN` on error.

`int do_sem_wait(struct exec_context *current, sem_t *sem_id)`

current is a pointer to the execution context of the process calling the `sem_wait()` system call
sem_id is the unique identifier associated with the semaphore to be operated on by the `sem_wait()` system call.
`do_sem_wait()` decrements (locks) the semaphore identified by `sem_id`. If the semaphore's value is greater than zero, then the this function should decrement the semaphore value and return. If the semaphore currently has the value zero, then the function should add the caller of this function to the end of the wait queue associated with the semaphore and schedule another process for execution. Helper functions (present in *gemOS/clone_threads.c*) `spin_init()`, `spin_lock()` and `spin_unlock()` can be used if needed.

Return value: This function should return 0 on success and `-EAGAIN` on error.

`int do_sem_post(struct exec_context *current, sem_t *sem_id)`

current is a pointer to the execution context of the process calling the `sem_post()` system call
sem_id is the unique identifier associated with the semaphore to be operated on by the `sem_post()` system call.
`do_sem_post()` increments (unlocks) the semaphore identified by `sem_id`. If the semaphore's

wait queue is empty, this function should increment the value of the semaphore and return. If the semaphore's wait queue is non-empty, this function should remove the process at the head of the wait queue and make it ready for scheduling in future. Helper functions (present in *gemOS/clone_threads.c*) `spin_init()`, `spin_lock()` and `spin_unlock()` can be used if needed.

Return value: This function should return 0 on success and `-EAGAIN` on error.

Testing

For this part, you may use the testcases (1 to 5) in the `user/testcases/part2/` directory. *Do not run automated test in this part without manual run. The script will hang. Do not forget to answer the questions in qns.txt.*

Q3. Copy-on-Write fork [80 marks]

In this task, you will be implementing the following system call and function:

- pid_t cfork()
- static long handle_cow_fault(struct exec_context *current, u64 vaddr, int access_flags)

Useful gemOS Functions

- **Allocate a Page Frame:** To allocate a PFN, you can use `os_pfn_alloc` by passing the region argument. If you want to use the PFN for page tables, you should use `OS_PT_REG` as the argument. For user data, use `USER_REG`

```
u64 pfn = os_pfn_alloc(OS_PT_REG)  //Allocate a PFN used for page tables
```

- **Free a Page Frame:** To free a PFN, you can use `os_pfn_free` by passing the region argument and the PFN to be freed. If you want to free a PFN used for page tables, you should use `OS_PT_REG` as the argument. For freeing page frames containing user data, use `USER_REG`. Note that, you need to check the reference count before freeing any PFN (see below).

```
os_pfn_free(USER_REG, pfn); //Free a PFN used for mapping user VA
```

- **Maintain Usage Reference Count for PFNs:** In CoW fork, you may need to share the PFNs from multiple processes. Sharing of PFN also breaks when a sharer writes to shared PFN. Therefore, we keep track of PFN usage through a reference counting mechanism using the following APIs.

```
s8 get_pfn(pfn);  //Increment refcount for the PFN
s8 put_pfn(pfn);  //Decrement refcount for the PFN
```

  Both functions return the reference count after increment/decrement. A PFN should be freed only after the reference count for the page becomes zero.

pid_t cfork()

System call handler: To implement `cfork()` system call, you are required to provide implementation for the template function `long do_cfork()` (present in *gemOS/cfork.c*).
Description: `cfork()`is a variant of the `fork()` system call which implements a copy-on-write policy for the address space of a process. The features of `cfork()` are as follows:

```
1. int main()
2.  {
3.      int pid;
4.      char * mm1 = mmap(NULL, 4096, PROT_READ|PROT_WRITE, 0);
5.      if(mm1 < 0)
6.      {
7.         printf("Map failed \n");
8.         return 1;
9.      }
10.      mm1[0] = 'A';
11.      pid = cfork();
12.      if(pid){
13.        mm1[0] = 'B';
14.        printf("mm1[0] inside parent:%c\n",mm1[0]);
15.      }
16.     else{
17.          printf("mm1[0] inside child:%c\n",mm1[0]);
18.     }
19.   }
```

Figure 4: Example: `cfork()`

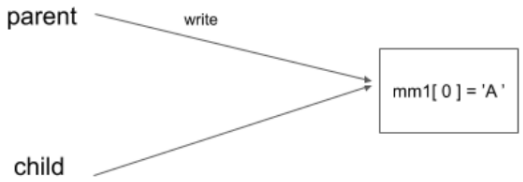

Figure 5: After parent calls cfork()

- Like `fork()`, when a process (called parent process) calls `cfork()`, a new process, called child process, is created.
- The implementation of `cfork` does not copy the memory content of the parent. However, the address space state should be copied.
- The virtual to physical mapping should be changed such that when either of the processes performs a write on any of the pages, a copy of that page is created (in the writer's address space) before proceeding with the write. Refer Figures 4, 5, 6 for a working example.
- When CoW sharing breaks (i.e., a page fault occurs), you have to duplicate frames and update the virtual to physical translation.

In the `do_cfork()` function you have to perform the following operations:

- Copy all the members of the parent process's `exec_context` to the child process's `exec_context`. For example, copy the contents of the `files` array (array of file descriptors) from the parent process to the child proces.
- Set the `ppid` (pid of the parent process) in the child process's `exec_context`
- Build a new page table for the child process. Note that gemOS uses 4-level page table. In this page table, you have to create page table entries only for the userspace part of the address space. Specifically, you have to create the page table entries for the present pages in the following memory segments of the child process—`MM_SEG_CODE`, `MM_SEG_RODATA`, `MM_SEG_DATA`, `MM_SEG_STACK` and for the VMAs of the child process. You can access the range of each memory segment using `mms[]` array in the `exec_context` of the child process. For example, `current->mms[MM_SEG_CODE]->start` and `current->mms[MM_SEG_CODE]->next_free` gives the range of address space covered by the code segment of the current process.

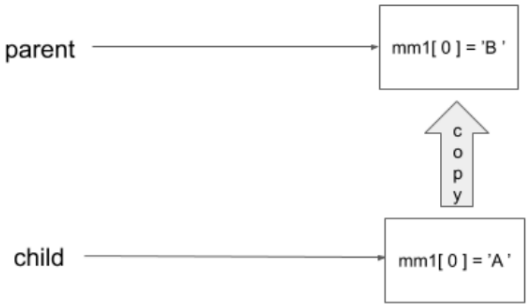

Figure 6: After parent writes to an address in shared page

- Page table entries in the PTE level/last level of the child process's page table should point to the same frames that are being pointed by the page table entries in the PTE level/last level of the parent process's page table. While updating the translation information, in both parent and child processes, the access permission in the PTE should be updated to restrict write accesses. Note that, the third bit in the flags part (bit 0 to bit 11) of the PTE represents the 'write' access. Setting this bit to zero makes the mapping read-only. You have to increase the reference count of the shared frames to indicate the number of page table entries pointing to the same frame.
- Modify the `pgd` member of the child `exec_context` to update it with the PFN value of the PGD level used for the page table of the child process.

Return Values: On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created. Note that, the template function already has some crucial calls which ensures correct execution of child process (if created correctly). You should not change them.

long handle_cow_fault(struct exec_context *current, u64 vaddr, int access_flags)

current: `exec_context` of the process that received the CoW fault
vaddr: Virtual address whose access resulted in the CoW fault
access_flags: Access flags of the VMA or memory segment to which the faulting address belongs to.
Description: `handle_cow_fault` function is called from the default page fault handler when a fault occurs due to write into a read-only mapping in the page table. Therefore, your implementation of `handle_cow_fault` should be able to handle CoW faults occurring for any userspace address. In this function, depending on the address and flags, you have to update the virtual-to-physical translation information and adjust the reference count of PFNs.

Return value: Return 1 if fault has been fixed, return -1 otherwise.

Testing

For this part, you may use the testcases (1 to 10) in the `user/testcases/part3/` directory. *Do not run automated test in this part without manual run. The script will hang. Do not forget to answer the questions in qns.txt.*