Exercise 1

- Write a C program "ex1.c" which writes its pid in /tmp/ex1.pid and generates a random password using the file /dev/random or /dev/urandom. The password should start with "pass:". It should consist of only printable characters. The length is 8 characters.
- Store the password in the memory using **mmap** (Create a shared anonymous mapping) or in the heap. Then, the program waits in an infinite loop.
- Write a script "ex1_hack.sh" which reads the pages of the virtual memory of the program "ex1.c" while it is running. It should find the password that is generated by "ex1.c". Print the password and its memory address to **stdout** and send *SIGKILL* to the program "ex1.c".
- Submit **ex1.c** and **ex1_hack.sh**.
- **Hints:** You need to search in /proc/{pid}/mem, using *xxd* or *gdb* commands. You may need to add *sudo* for these commands.

Exercise 2 (1/8)

- In this exercise, you will implement a simulation of virtual memory in C to further understand how virtual memory works. In addition, you will use *mmap* and signals for process communication.
- Virtual memory allows a process of P pages to run in F frames, even if $F < P$. This mapping is achieved by use of a *page table*, which records which pages are in RAM at which frames, and a page fault mechanism by which the memory management unit (MMU) can ask the operating system (here it is the pager) to bring in a page from disk. Each process has a page table. The page table must be accessible by both the MMU and the pager, and an IPC mechanism is needed for communication between the MMU and pager. In this simulation, the job of the pager is to maintain a process' use of RAM and the page table. The page table is held in memory backed by a file */tmp/ex2/pagetable*, and signals are used for IPC.

Exercise 2 (2/8)

- For simplicity, we assume that we run the program only for one process, the RAM is represented as an array `RAM` of strings of size F. The disk is represented by a an array `disk` of strings of size P. In both arrays, each array element (in this simulation, it is considered a page) is represented by a string of size 8.
- On startup, the array `disk` (represented by the disk) should contain different random messages (you can store manually or generate random printable characters) whereas the `RAM` is empty. When the pager needs to bring in a page i from the disk to frame j, it needs to copy the requested message i from `disk` array to `RAM` array in position j, then prints the `RAM` array. When the pager needs to write to disk (move the page of the frame j from the RAM to the disk in position i), it needs to copy the message j from `RAM` array to `disk` array in position i, then prints the `RAM` array. For simpicity, we assume that the write request, does not change the message but it just sets the dirty field.

Exercise 2 (3/8)

- The page table has mainly four fields in each page table entry and defined as follows:

```
struct PTE{
    // The page is in the physical memory (RAM)
    bool valid;
    // The frame number of the page in the RAM
    int frame;
    // The page should be written to disk
    bool dirty;
    // The page is referenced/requested
    int referenced;
}
```

Exercise 2 (4/8)

- The pager process must create the page table as a memory mapped file, and initialize it to indicate that no pages are loaded (all valid and dirty fields are set to false, frame field is set to -1 and referenced is set to 0). You can add more fields to the structure based on your implementation.
- **Note:** After you create a file to be mapped, you need to truncate it to the size of the page table using `ftruncate` before you map it using *mmap*. The size of the page table is the size of the struct PTE multiplied by the number of pages.
- Write a program **mmu.c** which accepts the command line arguments:
  - The number of pages in the process.
  - A reference string of memory accesses, each of the form mode page , e.g., W3 is a write to page 3.
  - The PID of the pager process.

Exercise 2 (5/8)

- The MMU opens the mapped file */tmp/ex2/pagetable*, then runs through the reference string. For each memory access, the MMU:
  - Checks if the page is in RAM.
  - If not in RAM ($valid = 0$), it sets the referenced field of the page to the PID of the MMU, and simulates a page fault by signaling the pager process with *SIGUSR1*. After that, it sleeps indefinitely until it receives a *SIGCONT* signal from the pager process to indicate that the page has been loaded to RAM.
  - If the access is a write access, it sets the dirty field of the page.
  - It prints the updated page table.
- When all memory accesses have been processed, the MMU closes the mapped file and signals the pager one last time (SIGUSR1). That must be detected by the pager process. If the pager process did not find any page referenced then the pager process can unmap the mapped file, delete the file and exit.
- Write another program **pager.c** which must take two arguments:
  - The number of pages in the process.
  - The number of frames allocated to the process.

Exercise 2 (6/8)

- Assume that the pages and frames are numbered 0, 1, 2, ...
- The pager process manages free frames, the disk array and RAM array.
- After creating and initializing the page table in the mapped file, the pager process must wait till accepting a SIGUSR1 signal from the MMU process. When it receives a signal, it must:
  - Scan through the page table looking for a non-zero value in the referenced field.
  - If a non-zero value is found, that indicates that the MMU wants the page at that index loaded.
    - If there is a free frame, then allocate it to the page.
    - If there are no free frames, choose a random frame in the table as a victim page (page replacement next lab). If the victim page is dirty, simulate writing the page to disk by copying the change from RAM to disk array, and increment the counter of disk accesses.
    - Update the page table to indicate that the victim page is no longer present in RAM.
    - Update the page table to indicate that the page is valid in the allocated frame, not dirty, and clear the referenced field.

Exercise 2 (7/8)

- Print the updated page table.
- Send a SIGCONT signal to the MMU to indicate that the page is now loaded.
- If no non-zero referenced field was found, the pager process terminates.

- Before terminating the pager process, you must print out the total number of disk accesses, and destroy the mapped file.
- Make sure that the program prints enough informative messages for each step. For example, when there are no free frame, the program should tell the user that a victim page will be selected.
- Make sure that you cannot allocate more than F frames and check the validity of input data.
- Submit **mmu.c** and **pager.c** with a script **ex2.sh** to run the simulation.

Exercise 2 (8/8)

Some test cases:

- Test case 1
  - 4 pages, 2 frames
  - ./pager 4 2
  - ./mmu 4 R0 R1 W1 R0 R2 W2 R0 R3 W2 $*pid_pager*
- Test case 2
  - 5 pages 3 frames
  - ./pager 5 3
  - ./mmu 5 R0 R1 R0 W1 R0 R1 R0 W1 R0 R2 R0 W2 R0 R2 R0 W2 R0 R3 R0 W3 R0 R3 R0 W3 R0 R4 R0 W4 R0 R4 R0 W4 $*pid_pager*
- ...etc

Exercise 3

- Write a C program **ex3.c**, that runs for 10 seconds. Every second it should:
  - allocate 10 MB of memory
  - fill it with zeros
  - print memory usage with *getrusage()* function
  - sleep for 1 second
- Compile and run the program in the background (./ex3 &) and run '*vmstat 1*' at the same time. Observe what happens to the memory. Pay attention to **si** and **so** fields.
- Add comments to **ex3.txt** with your findings.
- Hint: use *memset(ptr, value, size)* to fill the allocated memory
- submit **ex3.c** and **ex3.txt**.