CS333:Operating Systems Lab
Autumn 2023
LabQuiz 2, 20 marks

- **No Internet. No phone. No devices. No handwritten notes. No books.**
- **No friends** (strictly for the duration of the exam only!)
- **Please read the submission instructions carefully.**

- **Statutory warning:**
  Test cases are to be used at your own risk. There is no claim of comprehensiveness or grading completeness via the test cases.

Question 1	Question 2	Question 3
6 marks	7 marks	7 marks

Notes regarding the labquiz tarball:

- The tarball contains the xv6 source files along with the source files of the test cases for each of the problems in this lab quiz.

- Source files for the test cases are already added to the Makefile. Currently, the key system calls, which are part of the solutions, are commented out in these source files. As a result, these programs compile correctly, but do not have the intended functionality.

- After implementing your solution for a specific problem, uncomment the commented code from the test cases for that problem, recompile the xv6 source, and then execute the test cases to verify your solutions.

Q1. the family tree (6 marks)

Implement a xv6 system call `int getLevel(int pid)` which takes as argument **pid** of a process and returns the level at which that particular process is in the process tree.

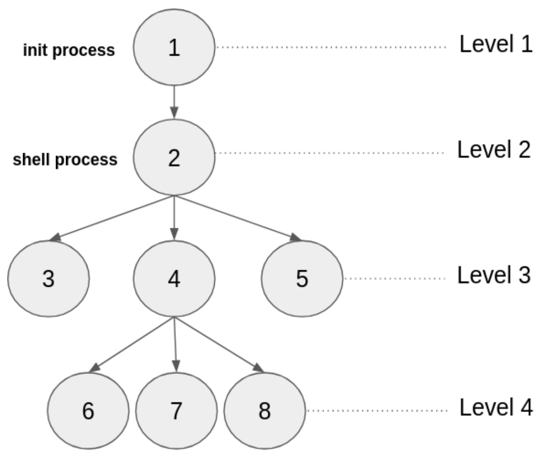


Figure 1: Process tree containing 8 processes, with their levels.

Figure 1 shows an example process tree—nodes represent processes with corresponding pids and edges represent the parent-child relation. The following observations hold:

- The root of the process tree is always the `init` process (pid 1).
- The init process forks the shell (sh) process which is pid 2.
- The shell is used to execute different programs which can create new processes at different levels or create new levels.
- For the process tree shown in the figure, `getLevel(8)` should return 4 as the process with pid 8 lies on level 4.
- The call to `getLevel` should return 1 for the init process and 2 for the shell process.
- The call to `getLevel` on an invalid process id should return 0.

Note: Level of a process can change during the lifetime of a process.

Test programs **leveltest1.c** and **leveltest2.c** are provided as part of xv6 tar provided with the labquiz description.

Hints:

- `usys.S, user.h, sysproc.c, syscall.h, proc.c, defs.h, syscall.c, proc.h` is a partial list of files which may intersect with the solution to this problem.
- In xv6, the proc structure of a process stores information about the parent (not a parent pid, but …)

Sample usage with leveltest1.c

```
$ leveltest1
parent level- 3 with pid 6
child level- 4 with pid 7
child level- 4 with pid 8
```

Sample usage with leveltest2.c

```
$ leveltest2
parent level- 3 with pid 3
child level - 4 with pid 4
grand child level before orphan - 5 with pid 5
grand child level after orphan - 2 with pid 5
Zombie
```

`Note that pid values may vary in the actual output.`

Q2. who let the reads out? (7 marks)

Create a new `xv6` system call `trace()`, that enables ***tracing*** of the **`read()`** system call.
The system call declaration is as follows — **`int trace(void);`**

Tracing implies tracking information everytime the read system call is used/executed within the kernel. The `trace()` system call enables tracing for the process that calls the system call and any child processes that the process forks subsequently. If a child process for whom tracing is enabled forks processes further, tracing remains enabled for the new (nested) child processes as well.

If tracing is enabled, a line of information about the read system call is printed, whenever the read system call is about to return from the call. The information to be printed per system call is — the **pid**, **number of bytes read** and the **file descriptor** used by the read system call, in the format specified below in sample usage.

The **trace** system call should return 1 on successful tracing enable, otherwise return 0.

User level programs (`tracetest1.c, tracetest2.c and tracetest3.c`) which use the `trace()` system call are provided as part of xv6 tar provided with the labquiz description.

Hint:

- `sys_read()` in `sysfile.c` contains all the information which needs to be printed on a `read()` system call after tracing is enabled.
- `usys.S, user.h, sysproc.c, syscall.h, proc.c, defs.h, syscall.c, sysfile.c, proc.h` is a partial list of files which may intersect with

the solution to this problem.

Sample usage with tracetest1.c

```
$ tracetest1
pid:3: read: 5 bytes from fd 3
pid:3: read: 5 bytes from fd 3
```

Sample usage with tracetest2.c

```
$ tracetest2
pid:4: read: 7 bytes from fd 3
pid:5: read: 8 bytes from fd 3
pid:5: read: 12 bytes from fd 4
```

Sample usage with tracetest3.c

```
$ tracetest3
pid:4: read: 8 bytes from fd 3
pid:5: read: 512 bytes from fd 4
pid:5: read: 512 bytes from fd 4
pid:5: read: 512 bytes from fd 4
pid:5: read: 512 bytes from fd 4
pid:5: read: 238 bytes from fd 4
pid:5: read: 0 bytes from fd 4
50 329 2286 README
```

`Note that pid values may vary in the actual output`

Q3. wanted to read, decided to write! (7 marks)

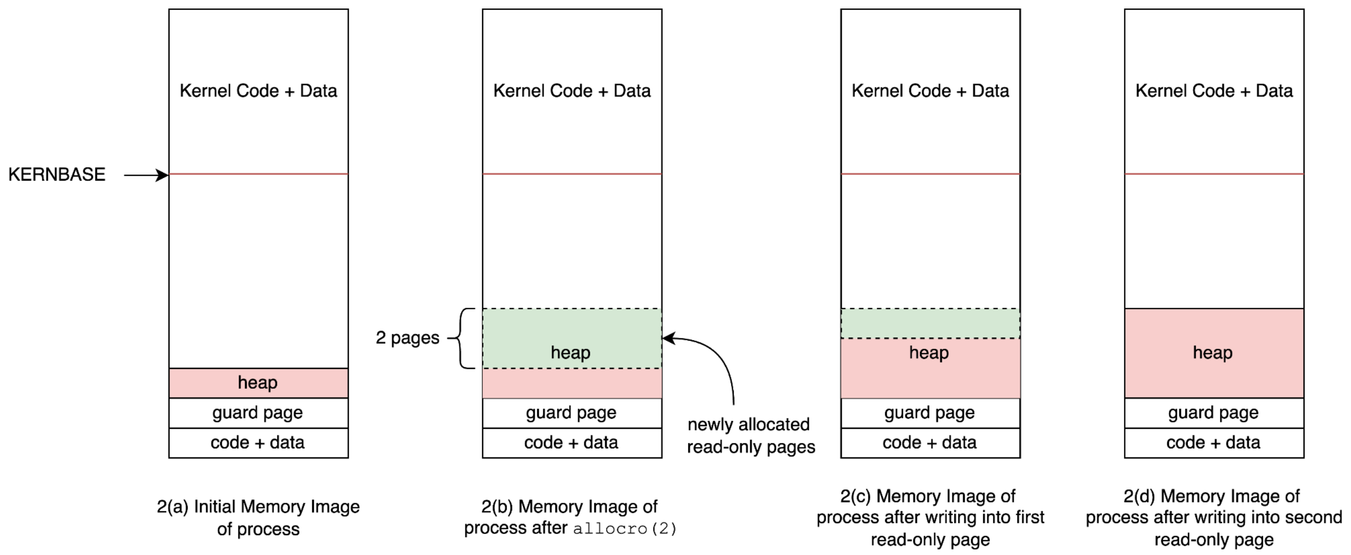
Create a new xv6 system call `allocro()` that allocates read-only pages. Subsequently, design and implement the necessary mechanisms to handle the page fault when write operations are attempted on these read-only pages. The objective is to handle page faults due to attempted write on the read-only page by converting it to a writable page and returning control to the executing process to fulfill the write.

Specifications:

- The `allocro()` system call allocates a specified number of read-only pages, returning the starting virtual address of the allocated region. The declaration of the system call should be:
  `char* allocro(int npages);`

- The system call increases the heap size, maps physical memory, and sets the allocated pages as read-only. Users should be able to read the pages without errors, but writing to these pages should generate page faults (look up the **`sbrk()`** system call for similarities).

- Handle page faults for writing to the read-only pages allocated by the `allocro()` system call.
    - Only make a single page writable that maps the virtual address accessed, and not the entire **`allocro`** allocated region for each page fault handling action.
    - The default xv6 trap handler must handle any other page faults (e.g., invalid memory accesses)
    - The page fault handler for **`allocro`** related faults should print the following line: **`Page Fault at 0x<va> due to read only pages, making it writable!`** `<va>` is the virtual address that generated the page fault.



Kernel Code + Data
KERNBASE
heap
guard page
code + data
2(a) Initial Memory Image of process
2 pages
newly allocated read-only pages
2(b) Memory Image of process after allocro(2)
2(c) Memory Image of process after writing into first read-only page
2(d) Memory Image of process after writing into second read-only page

Figure 2: Status of the heap with the `allocro()` system call. Green indicates read-only pages and pink indicates readable+writable pages.

Figure 2 demonstrates the `allocro()` system call's operation.
- `allocro(2)` allocates two read-only pages, increases the heap size, maps physical pages and returns the virtual address of the starting address of the new allocated region [Figure 2(b)].
- Reads on these addresses are successful, no page fault handling required.
- Write to a virtual address mapped within the first read-only page triggers a page fault which via the page fault handler is converted to a writeable page [Figure 2(c)].
- Another write using an virtual address to the second read-only page, causes a page fault and the handler now makes the second page writable [Figure 2(d)].

User level programs (`rotest1.c, rotest2.c, rotest3.c, rotest4.c and rotest5.c`) are provided for testing use of the `allocro()` system call. The test programs use `getvasize` and `getpasize` system calls, which return the total user virtual memory size and corresponding mapped physical memory size (in pages) of a given process. These system calls are already implemented in the starter code provided and should not be modified as part of this question.

Hint:

- `usys.S, user.h, sysproc.c, syscall.h, proc.c, trap.c, vm.c defs.h, syscall.c` are a partial list of files which may be useful to solve this question.

Sample usage with rotest1.c:

```
$ rotest1
Before allocro  V: 3 | P: 3
After allocro   V: 4 | P: 4
Value at allocated address 0x3000 is 0
Page Fault at 0x3000 due to read only pages, making it writable!
Value at allocated address 0x3000 after write is 5
```

Sample usage with rotest2.c:

```
$ rotest2
Before allocro  V: 3 | P: 3
After allocro   V: 6 | P: 6
Page Fault at 0x3000 due to read only pages, making it writable!
Value at allocated address 0x3000 is 5
Value at allocated address 0x4000 is 0
Value at allocated address 0x5000 is 0
Page Fault at 0x5000 due to read only pages, making it writable!
Value at allocated address 0x3000 is 5
Value at allocated address 0x4000 is 0
Value at allocated address 0x5000 is 5
Page Fault at 0x4000 due to read only pages, making it writable!
Value at allocated address 0x3000 is 10
Value at allocated address 0x4000 is 10
Value at allocated address 0x5000 is 10
```

Sample usage with rotest3.c:

```
$ rotest3
Before allocro  V: 3 | P: 3
After allocro   V: 5 | P: 5
Value at allocated address 0x3000 is 0
Value at allocated address 0x4000 is 0
Page Fault at 0x3000 due to read only pages, making it writable!
Page Fault at 0x4000 due to read only pages, making it writable!
Value at allocated address 0x3000 is 5
Value at allocated address 0x4000 is 5
Value at allocated address 0x3000 is 10
Value at allocated address 0x4000 is 10
```

Sample usage with rotest4.c:

```
$ rotest4
Before allocro  V: 3 | P: 3
After allocro   V: 4 | P: 4
pid 6 rotest4: trap 14 err 4 on cpu 0 eip 0x79 addr 0x8000--kill proc
```

Sample usage with rotest5.c:

```
$ rotest5
Before allocro  V: 3 | P: 3
After allocro   V: 4 | P: 4
Value at allocated address 0x3000 is 0
Page Fault at 0x3000 due to read only pages, making it writable!
Value at allocated address 0x3000 after write is 5
Before sbrk  V: 4 | P: 4
After sbrk   V: 5 | P: 5
Value at allocated address 0x4000 is 0
Value at allocated address 0x4000 after write is 15
Before allocro  V: 5 | P: 5
After allocro   V: 7 | P: 7
Value at allocated address 0x5000 is 0
Page Fault at 0x5000 due to read only pages, making it writable!
Value at allocated address 0x5000 after write is 10
```

Submission instructions

- For submission, create a folder on the Desktop named: `submission_<rollno>`
  E.g., a directory named `submission_22d0371`
  Please strictly adhere to this format otherwise your submission will not count.

- Add the xv6 folder in the submission folder. The folder structure should be as follows:

```
submission_<rollno>/
                        ├───xv6_public/
                                  ├──── asm.h
                                  ├──── bio.c
                                  .
                                  .
                                  .
                                  └──── zombie.c
```

- Next, from the xv6 folder execute the following commands:
  - **`make clean`**
  - **`check`**
    check ensures that the submission folder exists, is not empty and generates the required tar file. **(This step for student and TA)**
- Make sure that the required tar file is visible on the Desktop, and then run the **`submit`** command. You'll require a password for the final submission which the TAs will use. **(This step is for TAs only).**

Due Date: 14th September 2023, 5:00 pm