CS333:Operating Systems Lab
Autumn 2023
LabQuiz 3, 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**
  The test cases provided along with the questions, are examples for testing your implementations. 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 lab quiz 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. a.k.a (6 marks)

This question is related to implementing a functionality where multiple virtual addresses (pages) of a process map to the same physical page.

The intended functionality is to be Implemented via the following system calls:
`int spage()` and
`int spage2(int va)` whose descriptions are as follows:

Following are the requirements:

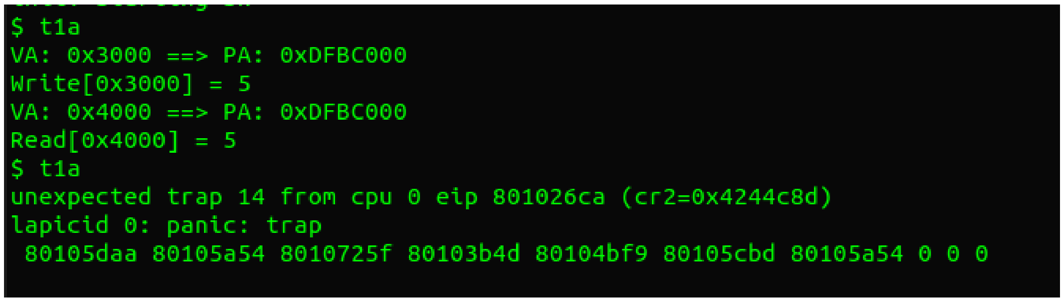
- `spage()` allocates a single page of memory for the process calling it and returns the virtual address (ensure that the virtual address be page aligned).

- `spage2(int va)` takes as an argument a virtual address **va** (returned virtual address of spage()) system call and allocates one page worth of process address space and maps it to the same physical page as that of the virtual address **va.**

- On process exit, and subsequent reap via the wait system call, the shared physical page has to be handled carefully. Since, multiple virtual addresses (pages) are mapped to the same page, they cannot be freed multiple times.

  You can assume that only one physical page will be used for sharing across multiple virtual addresses (pages) of a process.

Sample output if you do not handle the process address space clean up correctly.



```
$ t1a
VA: 0x3000 ==> PA: 0xDFBC000
Write[0x3000] = 5
VA: 0x4000 ==> PA: 0xDFBC000
Read[0x4000] = 5
$ t1a
unexpected trap 14 from cpu 0 eip 801026ca (cr2=0x4244c8d)
lapicid 0: panic: trap
 80105daa 80105a54 8010725f 80103b4d 80104bf9 80105cbd 80105a54 0 0 0
```

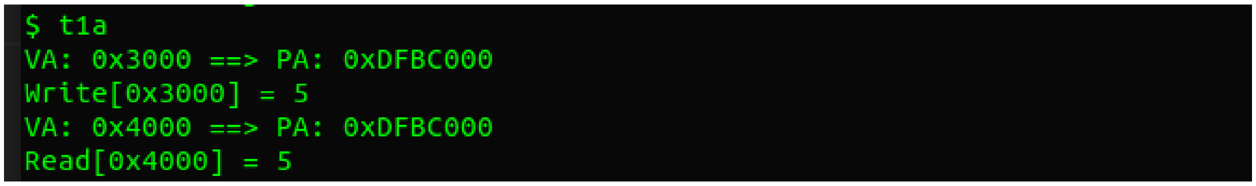
Notes:

- Only the first call of **spage** should allocate a new physical page for the process, subsequent calls to **spage** from the same process should return 0.
- **spage2** can be called multiple times and each call should yield a different virtual address mapped to the same physical page allocated via the **spage** call.
- `sbrk, walkpgdir, mappages` may come in handy for implementation
- `sbrk` takes as argument the number of bytes to be allocated unlike `spage` which allocates a page of memory

- `spage2(int va)` needs the physical address of the virtual address supplied, so that it can be reused.
- **wait, freevm, deallocuvm** are of probable interest for handling freeing of the shared memory.
- Note that the shared page is per-process and multiple processes can independently call and set up a shared physical page via **spage.** Refer to the program and output of **t1e.**
- On fork, the new process does not inherit the shared page.
- To handle the case when **spage** is called multiple times from the same process, **return 0** as the **va** and print “Page already shared” (look at sample usage of t1d) for calls to spage after the first call.

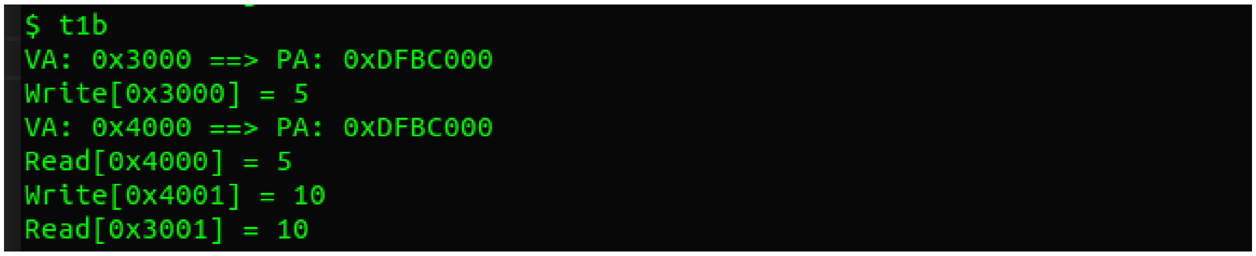
Testcase files **`t1a.c, t1b.c, t1c.c, t1d.c, t1e.c`** are provided to test your implementation.
Remember to uncomment the system calls before running the test cases (:

Sample usage with t1a



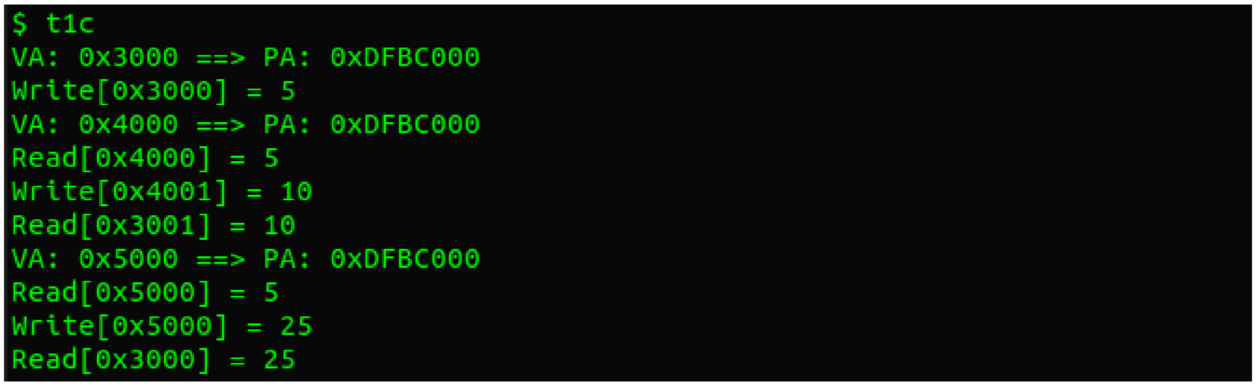
```
$ t1a
VA: 0x3000 ==> PA: 0xDFBC000
Write[0x3000] = 5
VA: 0x4000 ==> PA: 0xDFBC000
Read[0x4000] = 5
```

Sample usage with t1b



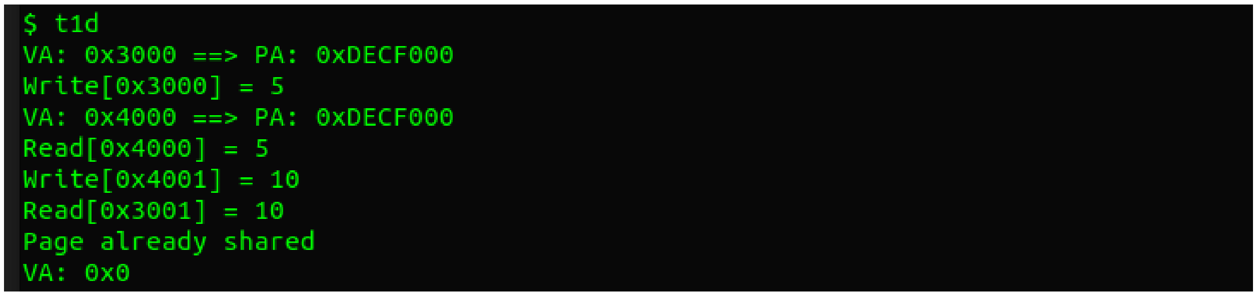
```
$ t1b
VA: 0x3000 ==> PA: 0xDFBC000
Write[0x3000] = 5
VA: 0x4000 ==> PA: 0xDFBC000
Read[0x4000] = 5
Write[0x4001] = 10
Read[0x3001] = 10
```

Sample usage with t1c



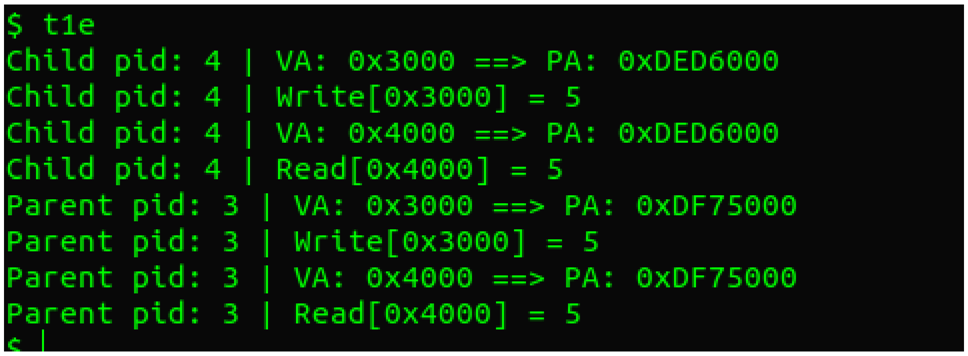
```
$ t1c
VA: 0x3000 ==> PA: 0xDFBC000
Write[0x3000] = 5
VA: 0x4000 ==> PA: 0xDFBC000
Read[0x4000] = 5
Write[0x4001] = 10
Read[0x3001] = 10
VA: 0x5000 ==> PA: 0xDFBC000
Read[0x5000] = 5
Write[0x5000] = 25
Read[0x3000] = 25
```

Sample usage with t1d



```
$ t1d
VA: 0x3000 ==> PA: 0xDECF000
Write[0x3000] = 5
VA: 0x4000 ==> PA: 0xDECF000
Read[0x4000] = 5
Write[0x4001] = 10
Read[0x3001] = 10
Page already shared
VA: 0x0
```

Sample usage with t1e



```
$ t1e
Child pid: 4 | VA: 0x3000 ==> PA: 0xDED6000
Child pid: 4 | Write[0x3000] = 5
Child pid: 4 | VA: 0x4000 ==> PA: 0xDED6000
Child pid: 4 | Read[0x4000] = 5
Parent pid: 3 | VA: 0x3000 ==> PA: 0xDF75000
Parent pid: 3 | Write[0x3000] = 5
Parent pid: 3 | VA: 0x4000 ==> PA: 0xDF75000
Parent pid: 3 | Read[0x4000] = 5
$
```

Q2. the *barrier* returns (7 marks)

Implement the barrier synchronization primitive, within the kernel space. This may be used in implementation of the Barrier.

The barrier synchronization primitive should block multiple processes from progressing until all of the processes reach the barrier condition/point (the barrier condition). When the barrier condition is met, the last process to reach the barrier opens the barrier allowing all blocked processes to proceed (we want all blocked processes to be released simultaneously).

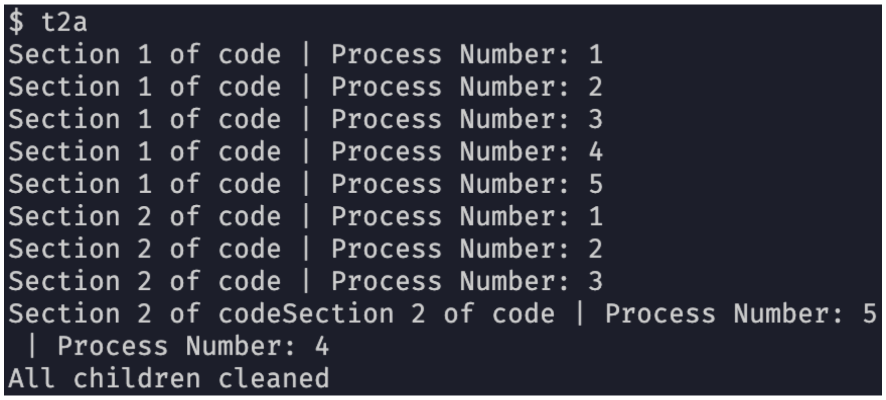
A basic `struct barrier` definition has been provided in `proc.c` for reference. Feel free to use it as it is, modify it, or change it entirely.

Work with the simplifying assumption that there is only one Barrier primitive within the kernel space, which can be allocated to user processes as and when required, and within a single test case barrier call would be made at most once.. Implement the following system calls:

- `barrier_init(int num)`
  initializes the barrier condition to wait for **num** instances to reach the barrier point. Whenever invoked, the system call should re-initialize the state, irrespective of last initialization. After every single use, the barrier may need re-initialization.
- `barrier()`
  defines the synchronization point between the execution before the call and after the call — all the instructions before the call should finish in all **num** instances, before any instructions after the call are executed in any execution instances.

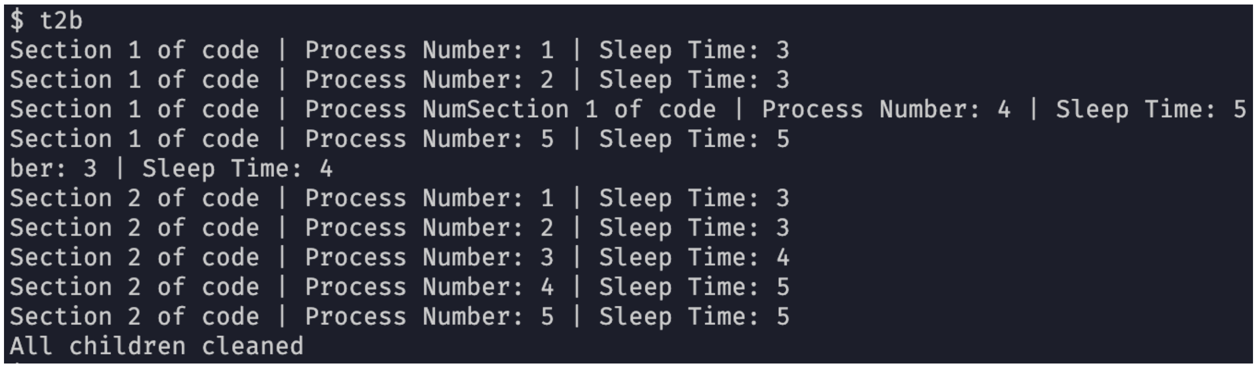
Testcase files **`t2a.c`** and **`t2b.c`** have been provided as examples to demonstrate use of the barrier implementation to achieve the desired synchronization requirements.

Sample usage with t2a



```
$ t2a
Section 1 of code | Process Number: 1
Section 1 of code | Process Number: 2
Section 1 of code | Process Number: 3
Section 1 of code | Process Number: 4
Section 1 of code | Process Number: 5
Section 2 of code | Process Number: 1
Section 2 of code | Process Number: 2
Section 2 of code | Process Number: 3
Section 2 of codeSection 2 of code | Process Number: 5
 | Process Number: 4
All children cleaned
```

Sample usage with t2b



```
$ t2b
Section 1 of code | Process Number: 1 | Sleep Time: 3
Section 1 of code | Process Number: 2 | Sleep Time: 3
Section 1 of code | Process NumSection 1 of code | Process Number: 4 | Sleep Time: 5
Section 1 of code | Process Number: 5 | Sleep Time: 5
ber: 3 | Sleep Time: 4
Section 2 of code | Process Number: 1 | Sleep Time: 3
Section 2 of code | Process Number: 2 | Sleep Time: 3
Section 2 of code | Process Number: 3 | Sleep Time: 4
Section 2 of code | Process Number: 4 | Sleep Time: 5
Section 2 of code | Process Number: 5 | Sleep Time: 5
All children cleaned
```

Explanation
The sleep time for each of the child processes have been randomized using **srand** function, implemented within the test case **t2b**. Do not toy with it. Test cases may be different during evaluation. :)

Q3. children are the future and the present (7 marks)

The goal is to create a custom scheduler that can be enabled and disabled by system calls.

The features of the scheduler are as follows:

- Child processes will always be scheduled before their parent/grandparent/ancestral processes. None of the ancestors can be scheduled before the child **finishes execution**.
- Even if the child is sleeping, the ancestors must not be scheduled.
- If there exist multiple processes, that are runnable, that do not have any children, the scheduling must proceed in a round-robin fashion (as it does in normal xv6) between these processes.
- The only process allowed to bypass the scheduler and get scheduled normally is the **init process** (to ensure that any zombies/killed processes are reaped correctly, otherwise init will never be scheduled). We won't be testing this in the implementation, but in case you have trouble debugging this might be a reason.

Hints:

- Ensuring that the child process executes before the parent process on fork is key.
- As soon as the new scheduler is turned on/off, its impact should be immediate.
- Print statements via **printf** rely on locks that processes need to obtain before writing to files/pipes and may cause deadlocks if ordering of parent and child processes is not handled carefully (with the new scheduler).

Implement two system calls - **`int scheduler_on()`** and **`int scheduler_off()`** whose descriptions are as follows:

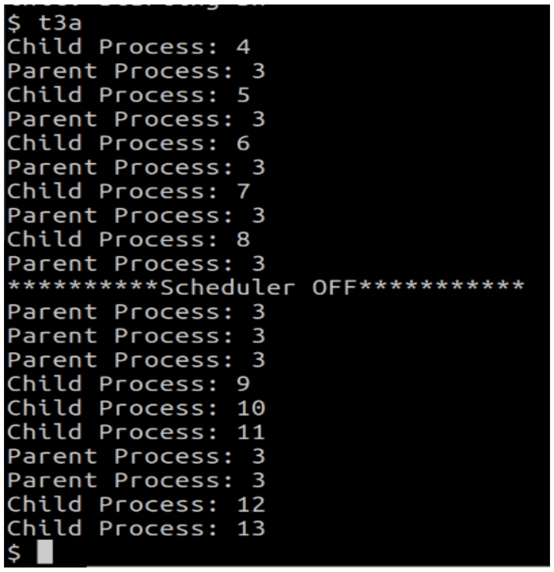
`int scheduler_on()`
Switches from the normal xv6 scheduler to the custom scheduler.
If already using the custom scheduler, does nothing, returns 1.

`int scheduler_off()`
Switches from the custom xv6 scheduler to the default scheduler.
If already using the default scheduler, does nothing, returns 0.

Change the code in proc.c (and other files if needed) to implement the above custom scheduler.

Four test case files have been provided — **`t3a.c, t3b.c, t3c.c, t3d.c`** for demonstrating use of the new scheduler. Remember to uncomment the system calls before running the test cases.

Sample usage with t3a

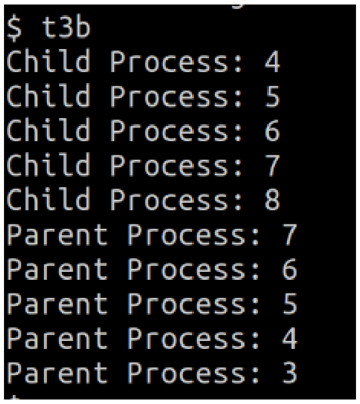


```
$ t3a
Child Process: 4
Parent Process: 3
Child Process: 5
Parent Process: 3
Child Process: 6
Parent Process: 3
Child Process: 7
Parent Process: 3
Child Process: 8
Parent Process: 3
**********Scheduler OFF***********
Parent Process: 3
Parent Process: 3
Parent Process: 3
Child Process: 9
Child Process: 10
Child Process: 11
Parent Process: 3
Parent Process: 3
Child Process: 12
Child Process: 13
$ 
```

Explanation
In the first part of the code, the custom scheduler is on, and so 3 children are forked in succession by the parent, and then exit in the reverse order. Since we have child scheduling first, the parent cannot have more than 1 child at a given point in time. The parent forks the child process with pid 4, which is scheduled and finishes execution, then processes with pid 5, which finishes and so on. This ends at pid 8. This part should match with your, except for maybe an offset in the PIDs.

In the second part, the scheduler is disabled and the normal xv6 scheduler is used. Here, the output should be haphazard with no clear predictable pattern. Possibly, there might be context switches in between print statements as well.

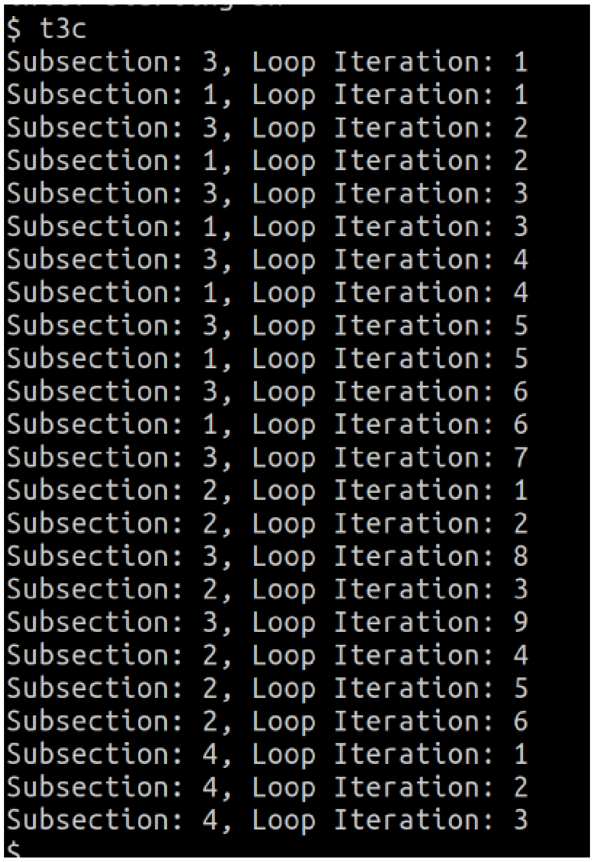
Sample usage with t3b



```
$ t3b
Child Process: 4
Child Process: 5
Child Process: 6
Child Process: 7
Child Process: 8
Parent Process: 7
Parent Process: 6
Parent Process: 5
Parent Process: 4
Parent Process: 3
```

Explanation
Here 5 children are forked in succession (i.e. parent forks a child, and the child forks another process, this repeats 5 times). The created processes then exit in the reverse order, so we have an increasing sequence of pids rom 4 to 8 which decreases to 4 again. This part should match with your output, except for maybe an offset in the PIDs.

Sample usage with t3c



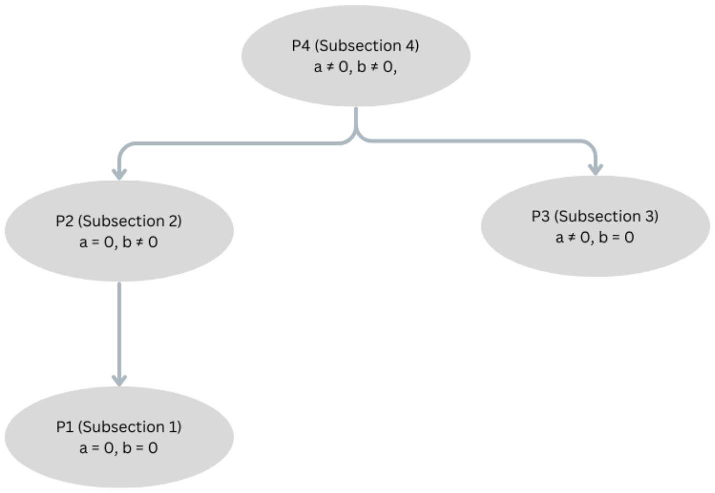
```
$ t3c
Subsection: 3, Loop Iteration: 1
Subsection: 1, Loop Iteration: 1
Subsection: 3, Loop Iteration: 2
Subsection: 1, Loop Iteration: 2
Subsection: 3, Loop Iteration: 3
Subsection: 1, Loop Iteration: 3
Subsection: 3, Loop Iteration: 4
Subsection: 1, Loop Iteration: 4
Subsection: 3, Loop Iteration: 5
Subsection: 1, Loop Iteration: 5
Subsection: 3, Loop Iteration: 6
Subsection: 1, Loop Iteration: 6
Subsection: 3, Loop Iteration: 7
Subsection: 2, Loop Iteration: 1
Subsection: 2, Loop Iteration: 2
Subsection: 3, Loop Iteration: 8
Subsection: 2, Loop Iteration: 3
Subsection: 3, Loop Iteration: 9
Subsection: 2, Loop Iteration: 4
Subsection: 2, Loop Iteration: 5
Subsection: 2, Loop Iteration: 6
Subsection: 4, Loop Iteration: 1
Subsection: 4, Loop Iteration: 2
Subsection: 4, Loop Iteration: 3
$
```

Explanation

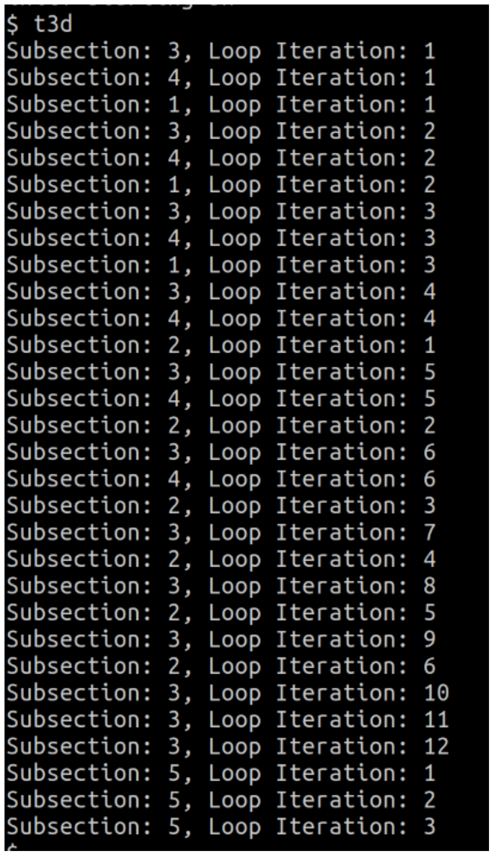
The code forks 4 children in the beginning before the custom scheduler is enabled. The original parent (P4 with a ≠ 0, b ≠ 0), its child (P2 with a = 0, b ≠ 0), its other child (P3 with a ≠ 0, b=0) and the child of P2 (P1 with a = 0, b = 0). Since P2 is the parent and P4 is the grandparent of P1, they will not be scheduled till P1 has ended (not even while P1 is sleeping).

Since both P1 and P3 do not have children, they will be scheduled in a round robin fashion till one of them ends. (P1 ends earlier than P3). This is why the initial code sections oscillate between subsections 1 and 3 (corresponding to P1 and P3). After P1 ends, P2 and P3 will be scheduled in round robin fashion till P3 ends. P4 is the parent of P2 and will remain unscheduled. Now, we will have a constant stream of subsection 2 statements (P2). In the end, P4 will execute and we will finish with a stream of subsection 4.

Of course, within each subsection the loop iterations should increase monotonously. The tree structure of the processes is shown below.



Sample usage with t3d



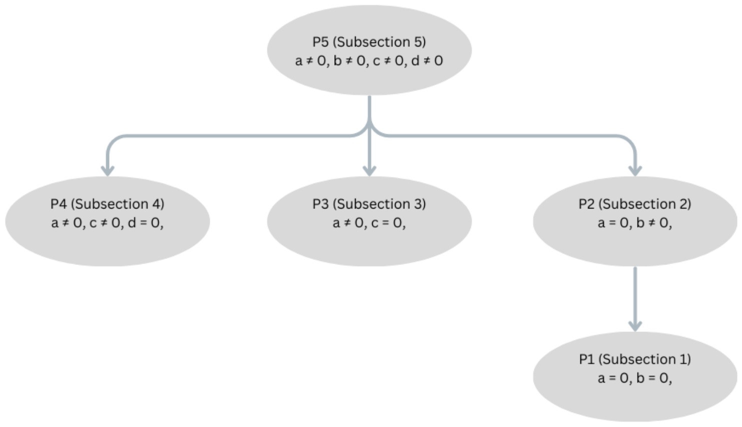
```
$ t3d
Subsection: 3, Loop Iteration: 1
Subsection: 4, Loop Iteration: 1
Subsection: 1, Loop Iteration: 1
Subsection: 3, Loop Iteration: 2
Subsection: 4, Loop Iteration: 2
Subsection: 1, Loop Iteration: 2
Subsection: 3, Loop Iteration: 3
Subsection: 4, Loop Iteration: 3
Subsection: 1, Loop Iteration: 3
Subsection: 3, Loop Iteration: 4
Subsection: 4, Loop Iteration: 4
Subsection: 2, Loop Iteration: 1
Subsection: 3, Loop Iteration: 5
Subsection: 4, Loop Iteration: 5
Subsection: 2, Loop Iteration: 2
Subsection: 3, Loop Iteration: 6
Subsection: 4, Loop Iteration: 6
Subsection: 2, Loop Iteration: 3
Subsection: 3, Loop Iteration: 7
Subsection: 2, Loop Iteration: 4
Subsection: 3, Loop Iteration: 8
Subsection: 2, Loop Iteration: 5
Subsection: 3, Loop Iteration: 9
Subsection: 2, Loop Iteration: 6
Subsection: 3, Loop Iteration: 10
Subsection: 3, Loop Iteration: 11
Subsection: 3, Loop Iteration: 12
Subsection: 5, Loop Iteration: 1
Subsection: 5, Loop Iteration: 2
Subsection: 5, Loop Iteration: 3
```

Explanation
The code forks 4 children in the beginning before the custom scheduler is enabled. The original parent (P5 with a ≠ 0, b ≠ 0, c ≠ 0, d ≠ 0), its child (P2 with a = 0, b ≠ 0), its other child (P3 with a ≠ 0, c=0), the other sibling (P4 with a ≠ 0, c ≠ 0, d = 0) and the child of P2 (P1 with a = 0, b = 0). Since P2 is the parent and P5 is the grandparent of P1, they will not be scheduled till P1 has ended (not even while P1 is sleeping).

Since both P1, P4 and P3 don't have children, they will be scheduled in a round robin fashion till one of them ends. (P1 ends first). This is why the initial code sections oscillate between subsections 1, 3 and 4 (corresponding to P1, P3 and P4). After P1 ends, P2, P3 and P4 will be scheduled in round robin fashion till P4 ends. Now, P2 and P3 will be scheduled in round robin fashion till P2 ends. Now, we will have a constant stream of subsection 3 statements (P3). In the end, P5 will execute and we will finish with a stream of subsection 5.

Of course, within each subsection the loop iterations should increase monotonously.

The tree structure of the processes is shown below.



P.S. - Questions are really not as tough as they might seem at first sight. Try and you will see!

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: 19th October 2023, 5:00 pm