

CS 236: Operating Systems Lab Spring 2025-26 Labquiz3 (20 marks)	A
--	-----------------

Instructions

- No Internet. No phone. No devices. No handwritten notes. No books.
- No friends (strictly for the duration of the exam only!)
- Unethical practices will be an automatic recommendation for the FR grade
- **Hardcoding of outputs will result in -5 marks per instance**
- Please read the questions and submission instructions carefully
- **Solve any 2 out of the 3 questions**
- **Solving all 3 questions: up to 5 bonus marks**

Submission Guidelines

- **Untar the labquiz3 directory on your Desktop. Use:**
 - In `labquiz3/q<no>`, `./run.sh` to run xv6 in containerized environment
 - In `labquiz3`, `submit-xv6` to submit (ask a TA to enter the password)
- **Ensure that your final xv6 submission compiles without any errors. Submissions that do not compile will not be considered for evaluation.**

1. mmap

(10 marks)

Implement a new xv6 system call — **mmap**.

mmap maps a physical page into an arbitrary user-specified virtual address, rounded down to a page boundary (`uva = PGROUNDOWN(uva)`).

The mapping should succeed only if,

- the address range is NOT already mapped or
- the address range is NOT in the existing heap

Further,

- **sbrk** should not overwrite **mmap**'ed pages
- **sbrk** should **fail** if a new **sbrk** allocation crosses into a **mmapped** page
- an accompanying system call **munmap** should free and unmap a **mmap**'ed page
- it is guaranteed that **munmap** is called for each **mmap**'ed page at least once before the program ends
- **mmap** on unsuccessful run should return 0
- **sbrk** failing should return with 0 (instead of -1 as is in the current implementation)

- **munmap** must return 0 (error) if we are trying to unmap a page that has not been mmap'ed with **mmap**

Assume that each process can have up to **one** active **mmap'ed** page.

Testcase **p1f.c** needs two mmap'ed pages. All other cases assume up to a single mmap'ed page. (3 marks)

```
int mmap(uint64 uva);    // return 1 on success, 0 otherwise
int munmap(uint64 uva);  // return 1 on success, 0 otherwise
```

Test cases

p1a — basic mmap - write - unmap

```
Shell
$ p1a
mmap successful
munmap successful
```

p1b — mmap - sbrk - unmap - write (sbrk doesn't overwrite mmapmed area, write into munmapped area)

```
Shell
$ p1b
mmap successful
sbrk successful
munmap successful
Writing into unmapped area...
usertrap(): unexpected scause 0xf pid=3
          sepc=0x5a stval=0x7000
```

p1c — mmap - write - sbrk - unmap (sbrk into mmapmed area)

```
Shell
$ p1c
mmap successful
sbrk unsuccessful
munmap successful
```

p1d — mmap - write - sbrklazy - unmap (sbrklazy into mmapmed area)

```
Shell
$ p1d
mmap successful
sbrk unsuccessful
munmap successful
```

p1e — mmap - write - munmap - sbrk (sbrk into the mmapmed area)

```
Shell
$ p1e
mmap successful
munmap successful
sbrk successful
```

p1f — mmap - mmap - write - unmap - unmap - sbrk (write, sbrk into mmapmed area)

```
Shell
$ p1f
mmap successful
mmap successful
munmap successful
munmap successful
sbrk successful
```

p1g — mmap into the heap area

```
Shell
$ p1g
mmap unsuccessful
munmap unsuccessful
```

p1h — mmap at a non page aligned address

```
Shell
$ p1h
mmap successfull
```

2. start-stop-start is progress (10 marks)

Add a new system call named **signal** that provides functionality to pause and resume a process's execution.

```
int signal(int pid, int signalnum);
```

pid — specifies the pid of the target process

signalnum — SIGSTOP or SIGCONT, these are two different constants that indicate the action to be taken. SIGSTOP suspends/pauses the process, and SIGCONT re-enables its execution.

The system call returns 0 on success and -1 otherwise.

Note:

- SIGCONT and SIGSTOP have been defined in param.h to be 18 and 19, respectively, and should not be changed
- The **kill()** system call should be able to kill a SIGSTOPped process

Test cases

1. tc-cont (1) - tests whether a SIGCONT signal does nothing to a running process
2. tc-stop (1) - tests whether a SIGSTOP signal stops a process
3. tc-2cont (0.5) - tests whether 2 consecutive SIGCONT signals are the same as one
4. tc-2stop (0.5) - tests whether 2 consecutive SIGSTOP signals are the same as one
5. tc-stop-cont (2) - tests whether SIGCONT can start a stopped process
6. tc-kill (2) - tests whether kill() can kill a stopped process
7. tc-multi-proc (1) - tests whether multiple processes can send signals
8. tc-inv-proc (1) - tests whether signal() returns -1 if PID is invalid
9. tc-inv-signal (1) - tests whether signal() returns -1 if signal number is invalid

Sample output: Check appendix at the end.

3. fair and square

(10 marks)

Completely Fair Scheduling is a scheduling algorithm used by Linux. Here we present a simpler version of it:

1. The scheduler keeps track of the virtual runtime of a process, which quantifies the amount of time this process deserved to run on an ideal scheduler v/s the actual time it ran, as follows
 - a. If a process runs for t seconds before giving up the CPU, its virtual runtime is incremented by t .
 - b. If a new process is forked, or an existing sleeping process is woken up, its virtual runtime is assigned the minimum virtual runtime out of all runnable and running processes except itself and the `initproc`.
2. The scheduler picks the process with the minimum virtual runtime among all runnable processes and schedules it for **TIMESLICE** seconds. After this timeslice, the scheduler is invoked again, with the updated virtual runtime values.

This tries to ensure that within **TIMESLICE** duration, the scheduler is fair to all processes.

Modify the xv6 scheduler to use this algorithm instead of round robin scheduling. Use 10 ticks as the value of the **TIMESLICE**.

In the `q3 xv6` directory, you will find the following:

1. A new field named **virtual_runtime** has been added into the `pcb` struct. You should use this field in your logic. The testing framework uses this field.
2. You are not expected to modify **user/user.h**, **user/usys.pl**, **Makefile**, **kernel/syscall.h** or **kernel/syscall.c**.
Changes to these files will not be retained while grading this question.
3. The number of cpus in the Makefile (CPUS) has been increased to 5 to allow for thorough testing of the scheduler. Make sure your scheduler works correctly in the presence of multiple CPUs.
4. Three test cases are provided to you
 - a. **tc-slice**: This checks if a process appropriately gives up cpu after 10 ticks.
 - b. **tc-vruntime**: This checks if the virtual runtime is incremented correctly for processes.
 - c. **tc-policy**: This checks the scheduling algorithm implemented.

Note that passing the given test cases is not an indicator of correct implementation. They are only provided to help you. For example, **tc-policy** might show unfair checks even if your implementation is correct. However, the unfairness will be reduced as compared to the round robin scheduler.

Sample Outputs

```
Shell
> make qemu
...
hart 4 starting
hart 1 starting
hart 2 starting
hart 3 starting
init: starting sh
$ tc-slice
=====
-----RUNNING TESTCASE-SLICE-----
=====
[PID 4] executed for an average timeslice of 10
[PID 5] executed for an average timeslice of 10
[PID 6] executed for an average timeslice of 10
[PID 7] executed for an average timeslice of 11
[PID 8] executed for an average timeslice of 10
[PID 9] executed for an average timeslice of 10
[PID 10] executed for an average timeslice of 10
[PID 11] executed for an average timeslice of 10
[PID 12] executed for an average timeslice of 10
[PID 13] executed for an average timeslice of 9
=====
-----TESTCASE PASSED-----
=====
$ tc-slice
=====
-----RUNNING TESTCASE-SLICE-----
=====
[PID 4] executed for an average timeslice of 5
[PID 5] executed for an average timeslice of 4
[PID 6] executed for an average timeslice of 3
[PID 7] executed for an average timeslice of 4
[PID 8] executed for an average timeslice of 4
[PID 9] executed for an average timeslice of 4
[PID 10] executed for an average timeslice of 4
[PID 11] executed for an average timeslice of 5
[PID 12] executed for an average timeslice of 4
[PID 13] executed for an average timeslice of 5
```

```
=====
-----TESTCASE FAILED-----
Average timeslice is not 10
=====
$ tc-vruntime
=====
----RUNNING TESTCASE-VRUNTIME---
=====
-----TESTCASE PASSED-----
=====
$ tc-vruntime
=====
----RUNNING TESTCASE-VRUNTIME---
=====
[PROCESS ID 7] 1. Virtual Runtime: 0, Actual execution time as
calculated by uptime: 17
[PROCESS ID 0] 1. Virtual Runtime: 0, Actual execution time as
calculated by uptime: 3
=====
-----TESTCASE FAILED-----
Process not scheduled out
=====
$ tc-policy
=====
----RUNNING TESTCASE-POLICY-----
=====
-----TESTCASE PASSED-----
Unfair checks : 0/58
=====
```

The failed output for tc-policy is given in the appendix.

[illegible]


```
[pid 8] Hello from target
[pid 8] Hello from target
[pid 8] Hello from target
[pid 7] Sent start again
[pid 8] Hello from target
[pid 8] Hello from target
[pid 8] Hello from target
[pid 8] Hello from target
[pid 8] Hello from target
[pid 8] Hello from target
[pid 8] Hello from target
[pid 8] Hello from target
[pid 8] Hello from target
[pid 7] Child killed
[pid 7] Kill succeeded
$ tc-2stop
[pid 10] Hello from target
[pid 10] Hello from target
[pid 10] Hello from target
[pid 10] Hello from target
[pid 9] Target stopped
[pid 9] Sent stop again
[pid 9] Exiting
$ tc-stop-cont
[pid 12] Hello from target
[pid 12] Hello from target
[pid 12] Hello from target
[pid 12] Hello from target
[pid 11] Target stopped
[pid 11] Target started
[pid 12] Hello from target
[pid 12] Hello from target
[pid 12] Hello from target
[pid 12] Hello from target
[pid 12] Hello from target
[pid 12] Hello from target
[pid 12] Hello from target
[pid 12] Hello from target
[pid 12] Hello from target
[pid 12] Hello from target
[pid 11] Child killed
[pid 11] Kill succeeded
$ tc-kill
```

```
[pid 15] Hello from target
[pid 15] Hello from target
[pid 15] Hello from target
[pid 15] Hello from target
[pid 14] Target stopped
[pid 14] Child killed
[pid 14] Kill succeeded
$ tc-multi-proc
[pid 17] Hello from target
[pid 17] Hello from target
[pid 17] Hello from target
[pid 17] Hello from target
[pid 17] Hello from target
[pid 18] Target stopped
[pid 16] Part 1 done
[pid 19] Target started
[pid 16] Part 2 done
[pid 17] Hello from target
[pid 17] Hello from target
[pid 17] Hello from target
[pid 17] Hello from target
[pid 17] Hello from target
[pid 17] Hello from target
[pid 17] Hello from target
[pid 17] Hello from target
[pid 17] Hello from target
[pid 17] Hello from target
[pid 16] Child killed
[pid 16] Kill succeeded
$ tc-inv-proc
[pid 20] Correct return value
$ tc-inv-signal
[pid 22] Correct return value
```

Failed output for tc-policy in Q3:

Shell

```
$ tc-policy
=====
-----RUNNING TESTCASE-POLICY-----
=====

=====UNFAIR STATE OF VIRTUAL RUNTIMES=====
PID STATE NAME VIRTUAL_RUNTIME

1 sleep  init 0
2 sleep  sh 0
3 run    tc-policy 0
4 runble tc-policy 10
5 runble tc-policy 10
6 runble tc-policy 10
7 runble tc-policy 10
8 runble tc-policy 0
9 runble tc-policy 0
10 runble tc-policy 0
11 runble tc-policy 20
12 runble tc-policy 0
13 run    tc-policy 14
14 runble tc-policy 10
15 runble tc-policy 20
16 runble tc-policy 0
17 runble tc-policy 0
18 run    tc-policy 12
19 runble tc-policy 10
20 runble tc-policy 10
21 runble tc-policy 10
22 runble tc-policy 10
23 runble tc-policy 10
24 run    tc-policy 9
25 run    tc-policy 9
```

=====UNFAIR STATE OF VIRTUAL RUNTIMES=====

PID STATE NAME VIRTUAL_RUNTIME

```
1 sleep  init 0
2 sleep  sh 0
3 run    tc-policy 0
4 runble tc-policy 0
5 runble tc-policy 20
6 runble tc-policy 0
7 runble tc-policy 0
8 runble tc-policy 10
9 runble tc-policy 0
10 runble tc-policy 0
11 runble tc-policy 10
12 runble tc-policy 0
13 run    tc-policy 19
14 runble tc-policy 10
15 runble tc-policy 10
16 run    tc-policy 9
17 runble tc-policy 20
18 runble tc-policy 0
19 runble tc-policy 0
20 run    tc-policy 19
21 runble tc-policy 10
22 runble tc-policy 0
23 runble tc-policy 0
24 sleep  tc-policy 20
25 run    tc-policy 15
```

=====UNFAIR STATE OF VIRTUAL RUNTIMES=====

PID STATE NAME VIRTUAL_RUNTIME

```
1 sleep  init 0
2 sleep  sh 0
3 run    tc-policy 0
4 runble tc-policy 10
5 runble tc-policy 0
6 run    tc-policy 5
7 run    tc-policy 1
8 runble tc-policy 10
9 runble tc-policy 0
10 runble tc-policy 0
11 runble tc-policy 10
12 runble tc-policy 0
```

```
13 runble tc-policy 20
14 runble tc-policy 0
15 run    tc-policy 11
16 runble tc-policy 10
17 runble tc-policy 0
18 runble tc-policy 0
19 runble tc-policy 10
20 sleep  tc-policy 20
21 sleep  tc-policy 0
22 sleep  tc-policy 0
23 run    tc-policy 1
24 sleep  tc-policy 0
25 sleep  tc-policy 0
```

=====UNFAIR STATE OF VIRTUAL RUNTIMES=====

PID STATE NAME VIRTUAL_RUNTIME

Display: 1

```
1 sleep  init 0
2 sleep  sh 0
3 run    tc-policy 0
4 runble tc-policy 0
5 runble tc-policy 0
6 runble tc-policy 20
7 runble tc-policy 0
8 runble tc-policy 0
9 run    tc-policy 11
10 runble tc-policy 10
11 runble tc-policy 0
12 run    tc-policy 5
13 run    tc-policy 21
14 runble tc-policy 0
15 runble tc-policy 0
16 runble tc-policy 0
17 runble tc-policy 10
18 run    tc-policy 5
19 runble tc-policy 10
20 runble tc-policy 0
21 runble tc-policy 0
22 runble tc-policy 0
23 runble tc-policy 10
24 runble tc-policy 10
25 runble tc-policy 10
```

=====UNFAIR STATE OF VIRTUAL RUNTIMES=====

PID STATE NAME VIRTUAL_RUNTIME

```
1 sleep  init 0
2 sleep  sh 0
3 run    tc-policy 0
4 runble tc-policy 10
5 runble tc-policy 10
6 runble tc-policy 0
7 runble tc-policy 10
8 runble tc-policy 10
9 run    tc-policy 2
10 runble tc-policy 0
11 runble tc-policy 0
12 runble tc-policy 0
13 runble tc-policy 10
14 run    tc-policy 18
15 runble tc-policy 10
16 runble tc-policy 10
17 runble tc-policy 0
18 runble tc-policy 20
19 runble tc-policy 0
20 runble tc-policy 0
21 runble tc-policy 10
22 runble tc-policy 10
23 sleep tc-policy 18
24 run    tc-policy 17
25 run    tc-policy 12
```

=====

-----TESTCASE FAILED-----

Unfair checks : 12/16

=====