

CS 333 Operating Systems Lab

Autumn 2023

Lab7: to share, or not to share?

The goal of this lab is to understand *race conditions* with multiple instances of execution and shared data objects, and implement solutions to overcome the same. Our work platform remains xv6.

Task 1: Blast from the past

Implement a system call called **smalloc** with declaration **char* smalloc(void)** which increases the size of the process (virtual) space by **one** page (4096 bytes), and maps it to a physical page. The virtual address of the process if to be kept page aligned.

The characteristic of this system call is that while every call changes the process (virtual) address space usage (size) the **same** physical page is used for mapping the virtual address range (in effect providing a shared memory region). Note that **smalloc** is a system call and can be invoked from across processes or multiple times in a single process and sets up multiple instances of execution with a shared physical page.

Implementation notes:

- Needs logic to store and reuse a reserved physical page for multiple mappings.
- Needs careful handling of freeing memory mappings to a shared page. By default xv6 has no support for page sharing and frees pages whenever a process exits. Support needs to be added to make sure that pages are not freed on page table cleanup or on any other action when the physical page may still be shared via other mappings.
- xv6 functions of interest —
`growproc, allocuvm, deallocuvm, freevm, mappages, kalloc, ...`

Uncomment the lines corresponding to counter1 and final, compile and execute the file **t1.c** to test your implementation.

Usage: `t1 <number of processes to fork>`

Sampe usage:

```
$ t1 2
PID: 4, VA: 3000, Counter Value: 56
PID: 5, VA: 3000, Counter Value: 56
PID: 3, VA: 3000, Final Count: 56
Total Ticks Taken: 5
```

Now, uncomment the two lines corresponding to counter2 present in t1.c to observe that different smalloc() calls return the same shared area even though the virtual addresses differ

```
$ t1 1
PID: 4, VA: 3000, Counter Value: 56
PID: 4, VA: 4000, Counter2 Value: 56
PID: 3, VA: 3000, Final Count: 56
Total Ticks Taken: 4
```

p.s.: PID values, Total Ticks taken need not match.

Task 2: race to the bottom

This task depends on and uses the **smalloc** system call.

Understand the program specified in the file **t2.c** and execute it in xv6 to observe its outputs. Uncomment all the lines present in t2.c before compiling

Usage: t2 <number of processes>

Sample usage

```
Actual Count: 13
Expected Count: 80
Total Ticks Taken: 16
$ t2 50
Actual Count: 37
Expected Count: 500
Total Ticks Taken: 39
$ t2 61
Actual Count: 43
Expected Count: 610
Total Ticks Taken: 46
$ t2 44
Actual Count: 33
Expected Count: 440
Total Ticks Taken: 35
```

Do we have a race condition yet?

Task3: Where is the lock?

To solve the above problem we need a synchronization mechanism to provide mutual exclusion for read-modify-update operations.

Task 3a: *spin* with tension

Implement a spin lock synchronization primitive for updates to shared memory. A **spinlock** causes a process trying to acquire it to simply wait in a loop (called spinning) while repeatedly checking whether the lock is available. This is also known as a busy-wait approach as the process keeps spinning until the lock is available (Can you observe the downside of this approach?)

To achieve this locking mechanism, implement the two system calls

```
void acquirespinlk(void) – System call to acquire the spin lock
void releasespinlk(void) – System call to release the spin lock
```

Hint: You can look at the xv6's spinlock implementation present in **spinlock.c** to understand the use of the atomic **xchg** instruction.

Understand the program specified in the file **t3.c** and execute it in xv6 to observe its outputs. Uncomment all the lines present in t3.c before compiling

Usage: t3 <number of processes>

Sample usage

```
$ t3 4
Actual Count: 40
Expected Count: 40
Total Ticks Taken: 75
$ t3 10
Actual Count: 100
Expected Count: 100
Total Ticks Taken: 444
$ t3 50
Actual Count: 500
Expected Count: 500
Total Ticks Taken: 11760
```

p.s.: Total Ticks taken need not match, but the final count values should match.

Task 3b: one does not simply sleep

Implement the mutex synchronization primitive, we will call it the sleeplock. **sleeplock** causes the process to move into a waiting/blocked state and transfers control back to the cpu scheduler when the lock is held by another process. On release of the lock, all processes waiting on the lock are contenders for the lock and moved to the RUNNABLE (Ready) state, before transferring control to the scheduler.

Can you reason about the benefits of sleep lock over spin lock!?
In which cases does a spinlock perform better than a sleeplock, are there any?

To achieve this mutex based synchronization mechanism, implement the two system calls
void acquiresleeplk(void): System call to acquire the sleep lock
void releasesleeplk(void): System call to release the sleep lock

Hints:

Look up xv6's sleeplock implementation present in **sleeplock.c** and the **sleep** and **wakeup** functions in **proc.c**

Observe how xv6's sleeplock makes use of spinlock for implementing the sleeplock.

You should use the spinlock implemented in the previous part as the spinlock required by the sleeplock.

Understand the program specified in the file **t4.c** and execute it in xv6 to observe its outputs.
Uncomment all the lines present in t4.c before compiling

Usage: t4 <number of processes>

Sample usage

```
$ t4 4
Actual Count: 40
Expected Count: 40
Total Ticks Taken: 43
$ t4 10
Actual Count: 100
Expected Count: 100
Total Ticks Taken: 102
$ t4 50
Actual Count: 500
Expected Count: 500
Total Ticks Taken: 503
```

p.s.: Total Ticks taken need not match, but the final count values should match.

Can you observe the benefits of sleep lock over spin lock!?
Why are the Total Ticks Taken provided with the test cases?

Submission instructions

- Submissions are on the assignment link via Moodle.
Name your submission as {rollnumber_lab7}.tar.gz
`tar -czvf {rollnumber_lab7}.tar.gz {rollnumber_lab7}`
(e.g., 200050183_lab7.tar.gz)
- The tar submission should contain all xv6 source files along with the test case files and the makefile
- Run `make clean` before making a tar file for submission

Deadline: 6th October, Friday, 17:00