CS 333: Operating Systems Lab
Autumn 2018

Lab #7 Synchronization in xv6

Goal

In this lab we will understand how xv6 implements synchronization primitives (the spinlock) and how to implement a custom spinlock to be used by user-level programs.

Before we begin

- MAke sure you have the source code for xv6:
  `http://www.cse.iitb.ac.in/~puru/courses/autumn18/labs/xv6-public.tar.gz`
  Make sure you read the README to understand how to boot into a xv6-based system.
- Download, read and use as reference the xv6 source code companion book.
  url: `http://www.cse.iitb.ac.in/~puru/courses/autumn18/labs/xv6-rev10.pdf`
- The xv6 OS book is here:
  `http://www.cse.iitb.ac.in/~puru/courses/autumn18/labs/xv6-book-rev10.pdf`

0. breaking the lock

Dunk into the code of `spinlock.c` and study the variables and functions in the file carefully. Answer the following questions,

i What are names of the functions for obtaining and releasing a spinlock?

ii What is the role of the `holding()` function?

iii Add code at some xv6 location where a spinlock is being acquired to trigger panic via the `holding` function? Explain your addition.

iv Lookup implementation of the `xchg`, `pushcli` and `popcli` functions.

v If a `sti()` call is added just after acquire and a `cli()` just before `release()`, what is the effect?

1. just lock it

The purpose of this exercise is to implement your own spinlock (in xv6) using the `xchg` **instruction** and make a test program to understand the concurrency issues that can arise while accessing critical sections of the memory.
An array of 10 integers(stored in an array) are protected by ten of your own locks. The data and locks are **stored in the kernel**. The exercise requires writing 5 system calls to use the locks and modify the associated data via user-level programs,

- `int init_counters()` : Sets all the 10 locks as unlocked, initializes all the 10 data variables to 0, and returns 1.
- `int acquire_lock(int lockNum)` : Acquires the lock with index `lockNum`. Note that lock with `num` should protect data at index `lockNum`.

  If there is any error return -1, else return 1
- `int release_lock(int lockNum)` : Releases the lock with index `lockNum`. If there is any error return -1, else return 1
- `int get_var(int num)` : Gets the value of data item present at index `num`. If there is any error return -1.
- `int set_var(int num, int newVal)` : Sets the value at index `num` in the data array to `newVal`. If there is any error return -1, else return 1.

Two test programs are given, `counter_with_locks.c` and `counter_without_locks.c`

The program `counter_without_locks.c` is complete. It forks a child process and both the parent and child process increment the same data variable without using a lock `10000 times` each. In the end the value present in that data variable is printed. Because of concurrency issues you will notice a number less than `20000`.

Now, you are required to complete the test program `counter_with_locks.c` to ensure safe updating of the variables. If the program is correctly implemented, the count printed in the end will be `20000`. Note that lock number `num` should be used to proctect data item at the same index. Also, no locking code inside the `get` and `set` functions.

2. 10 fold scale-up.

Based on the same setup as before, 10 data items and 10 locks, write a user-level program that does the following,

- Initialize locks and data values
- The parent process creates 10 child processes.
- Each child process, adds 1 to its corresponding data value, a 1000 times. E.g., child 0 updates data[0].
- The parent also updates the data items in the following manner—adds 1 to data[0], then to data[1] etc. for all 10 data items and then repeats for a total of 1000 iterations.
- The parent process prints the values of all the data variables after all children completed the execution. With correct synchronization the each of the data values should be `2000`.
- Name this programs `nlocks.c`

3. binary confusion.

Write a c program, `toggle.c`, to create a child process and to toggle the execution of parent and child process in a deterministic order. First, the parent process increments and displays the value of a shared variable, say `data[0]`. Then the child process decrements the **same** shared variable and displays the value. Next, the parent increments and the child decrements and so on in the same pattern. Run both parent and child in a loop for 30 times.

You need to synchronize the execution of parent and child using the lock implementation in `Part 1` and possibly using multiple locks. The shared variable should be initialized to zero.
The value of the shared variable is always `0` or `1` with proper toggling.
Verify the output of your program with the sample output given below.

```
In parent, 1
In child,  0
In parent, 1
In child,  0
...........
...........
In parent, 1
In child,  0
In parent, 1
In child,  0
In parent, 1
In child,  0
```

Submission Guidelines

- All submissions via moodle. Name your submission as: **<rollno_lab7>.tar.gz**
- The tar should contain the following files in the following directory structure:

  <rollno_lab7>/
  |__README describing answers to Part 0
  |__xv6-public-<rollnumber>/
  |____<all files in xv6>
  |__outputs/
  |____<outputs of sample runs (exercise 1, 2, & 3)>

- We will evaluate your submission by reading through your code and executing it over several test cases.
- **Deadline: Monday, 24th September 2018 - 05:00 PM**.