CS 333: Operating Systems Lab
Autumn 2018

Lab #8 no to spinning!

Goal

In this lab you will learn about non-spinning/non-blocking locks and use them to build a synchronization based use case.

Before we begin

- 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. kernel support for user space mutexes.

The following variables have been defined as part of the xv6 kernel source:

```
struct mutex {
    int value;      // stores lock state if used as mutex and
                    // stores buffer size for producer-consumer
    spinlock s;     // the lock protecting the mutex state
}m[NLOCKS];
```

Implement the following system calls (skeleton code is available in `sysproc.c` and `syscall.h`),

- `acquire_mutex_spinlock(int index)`
  Acquire spinlock in the mutex at the location *index* in the mutex array.
- `release_mutex_spinlock(int index)`
  Release spinlock in the mutex at the location *index* in the mutex array.
- `cond_wait(int variable, int index)`
  Call to use mutex pointed at *index* and associated spinlock to sleep on the condition variable specified. This system call should use the custom sleep function `cv_sleep` mentioned below.
- `cond_signal(int variable)`
  Call to wakeup all processes blocked on the condition variable *variable.*
- `get_mutex_value(int index)`
  Get the value associated with the mutex at the specified mutex array index.
- `set_mutex_value(int index, int value)`
  Set the value associated with the mutex at the specified mutex array index.
- `init_mutex()`
  Initialize all the mutex variables in the mutex array.

The following kernel functions are provided, and need to be used in the corresponding system call implementations.

- `spinlock_acquire(spinlock *s)` A kernel function to acquire a spinlock.
- `spinlock_release(spinlock *s)` A kernel function to release a spinlock.
- `cv_sleep(int variable, spinlock *s)` A kernel function to put a process to sleep on a condition variable.

1. just *mutex* it!

In this section, we will build the mutex synchronization primitive using condition variables.
Use the system call implementation in Part 0 to implement an user-space mutex to synchronize processes.

- Reuse the system calls:
  `int init_counters()`, `int get_var(int num)`, and `int set_var(int num, int newVal)` from Lab #7.
- Write two user-space functions (refer to test cases for skeleton code), `mutex_lock(int index)` and `mutex_unlock(int index)`.
  Note that each mutex will need associated condition variables for non-spinning based example. Note that all shared data and locks are stored in the kernel, and the user space processes use the mutex structure and the locks to implement a condition variable based non-blocking syncrhonization mechanism.
- **Testing**
  Add the mutex implementation to the source file of testcases provided, `counter.c`, `lock-and-delay.c`, and `nlocks.c`.
  For `nlocks.c`, implement 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.
    Note that parent and child accesses can happen in parallel.
  - 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`.

2. the consumption era

Implement a synchronization mechanism to solve the producer-consumer synchronization problem. Assume that a producer can produce objects till production reaches capacity, this is captured using a simple variable, `bufsize` can be use to denote the current size of buffer produced goods. On each consumption, the buffer size decreases by one. To produce an item, the producer invokes the function `produce()`, and to consume an entry, the consumer invokes `consume()`.
Note that production cannot exceed capacity and consumption is possible only when there are produced goods.

Use the sample program `prodcon.c` as a skeleton code to get started.
`./prodcon <file1> <file2>`
The program reads takes two file names as command line arguments, and creates a child process. The parent process reads the first input file, and performs a combination of `produce()` and `consume()` actions based on the input file. The child process uses the second file as input and performs a combination of `produce()` and `consume()` as well.

Format of input file:
`<P/C> <delay before next operation>`
`<P/C> <delay before next operation>`
The lab description has two sets of input files to test your implementation. Create your own test cases to further verify correctness of your implementation.

3. to read or to write, is that the question? (*no submission required*)

Assume a situation where several threads read and write to a shared data item. Reads do not necessarily need locked access, no modifications on reads! But with parallel reads and writes, locks are required.

With a **reader-writer** lock, multiple readers can concurrently access the data. However, a writer must not access the data concurrently when either readers or writers want to access the data simultaneously. A `reader-writer` lock has two lock methods and two unlock methods. If a thread wants to read the shared data, it invokes `ReaderLock()`, and when it finishes the critical section, it calls `ReaderUnlock()`. A reader lock does not block on other active reads, but locks if a writer is active and holds the writer lock. Similarly, the writer thread calls `WriterLock()` when it wants to enter a critical section and invokes `WriterUnlock()` when it leaves the critical section. In addition, you need one more function, `InitalizeReadWriteLock()`, to initialize the `read-write lock`.

Submission Guidelines

- All submissions via moodle. Name your submission as: **<rollno_lab8>.tar.gz**
- The tar should contain the following files in the following directory structure:

  ```
  <roll_number_lab8>/
  |__lab8-xv6-public-<rollno>/
  |____<all files in the xv6>
  |__outputs/
  |____<outputs of sample runs (exercise 1, & 2)>
  ```

- We will evaluate your submission by reading through your code and executing it over several test cases.
- **Deadline: Monday, 1st October 2018 - 05:00 PM**.