CS 333: Operating Systems Lab
Autumn 2022, Lab 9
Implementing Synchronization to be used with the `clone` System Call

---

This lab explores the introduction of synchronization primitives in xv6. In this lab, we will modify the xv6 code to create a lock and semaphore for synchronization across threads.

Part 1: Locks For Cloned Processes

Skeleton Code Provided: You should copy the xv6 files provided to you to your xv6 directory for the starting point of your implementation. You have to write your code in these files accordingly, wherever TODO is mentioned.

We have already provided the xv6 files which you have to copy and replace in the original xv6 directory, in which `clone` and `join` system calls are implemented according to Lab8. In this lab, you will have to implement **locking system calls** for multi-threaded programs built using clone system calls. You have to implement three system calls `init_lock`, `acquire_lock`, and `release_lock` for the same.

For this, use the given `lock_t` structure, shown below, that will contain the lock variable (to be initialized to zero), which you can use later as an argument for the functions required.

```
typedef struct lock_t {
     uint locked;
} lock_t;
```

```
int init_lock(lock_t *);
int acquire_lock(lock_t *);
int release_lock(lock_t *);
```

All the structures are declared in `ulock.h` and system call declarations are in multiple files, you have to code their definitions in `proc.c` and `ulock.c`. This is for both parts of this Lab.

Implementation Instructions:

The `init` function initializes the lock. The lock function's internal logic is similar to the kernel spinlock already present in xv6. You have to implement a spinlock, as opposed to a sleeping lock. This will be a lighter version of kernel spinlock already implemented in xv6.

You will have to **use the atomic function `xchg`**, in the user space which you will need for lock logic, for releasing and acquiring the locks. The function is already a part of kernel xv6 locking logic and can be found in the **`xv6.h`** file. You can directly use this function in the userspace file after including `"xv6.h"`.

Acquire function will use the **`xchg`** function in a while loop and spin till the lock is released. The release function will use `asm volatile` (a C function used for passing assembly code, you can use the internet to look into this in detail). `asm volatile` function is also used in kernel `xchg` implementation and kernel spinlock implementation in xv6. You can refer to those functions for understanding their use in releasing the lock.

Also, the C compiler and hardware may re-order loads and stores. To avoid this you have to use a synchronize function in your user space lock implementation, which can be found in kernel spinlock implementation.

Part 2: Semaphores

In this part you need to implement semaphores **system calls** for synchronization amongst userthreads. You have to implement **`init_sem`**, **`up_sem`**, and **`down_sem`** for the same.

```
typedef struct sem_t {
      uint value;
      lock_t lk;
} sem_t;

int init_sem(sem_t *, int);
int up_sem(sem_t *);
int down_sem(sem_t *);
```

The structure is declared in `ulock.h` and functions definitions should be coded in `ulock.c` and `proc.c` in the TODO part.

Implementation Instructions:

- The `init` function initializes the semaphore. First argument is a pointer to your semaphore object and the second argument is the positive value with which your semaphore is to be initialized. The semaphore's internal logic is similar to the kernel sleeplock already present in xv6.

- To implement `down_sem` we need to check if the value is positive. If it's positive we need to decrement it by 1 while holding the semaphore's lock

- If the value is 0 we sleep on some identifier which indicates this thread is to be resumed iff this semaphore is upped. This identifier can be the address of semaphore's lock or address of semaphore itself.

- You need to define a function block_sem in `proc.c`, for suspending the thread. Skeleton is already provided, you just have to fill the logic. In the function, chan is the identifier.

  `void block_sem(void* chan);`

  This function will acquire ptable lock, then release the semaphore lock, and then call sleep with chan and second argument as `&ptable.lock`. Then release ptable lock. (We acquire ptable lock as the sleep function expects the kernel code to hold some kernel-spinlock before calling it and the sleep function nonetheless acquires ptable lock so we do it earlier as we have no other kernel-spinlock to acquire) Then again acquire semaphore lock.

  Additional info: The ordering of acquiring/releasing of locks is like this to avoid edge cases where a thread is interrupted before going to sleep and during this time another thread comes and calls up_sem which will broadcast resume signal to all the threads waiting on the semaphore. The thread which got interrupted before sleeping will fail to receive this signal even though it was supposed to wake up from the signal. It will instead sleep now.

- When we resume we again check the value and sleep if the value is still 0. Else we can go ahead and decrement the value. You can refer to `acquiresleep` function in sleeplock.c

- For implementing `up_sem` we need to increment the value by 1 and then awaken the threads blocked on the semaphore. Resumption of threads is done by calling **signal_sem** function which will resume all the threads waiting on the channel (chan) passed as an argument. You can refer to `releasesleep` function in sleeplock.c.signal_sem

void signal_sem(void* chan);

You need to implement **signal_sem** which calls wakeup to resume all the processes waiting on chan.

Note: Make sure to release the semaphore's lock before sleeping and acquire it back after you awaken from sleep. In both signal_sem and block_sem chan is passed which is the virtual address of semaphore/semphore's lock. But we want our threads to sleep on the physical address so as to avoid conflict with some other process. So convert this va to pa before passing it to wakeup or sleep call.

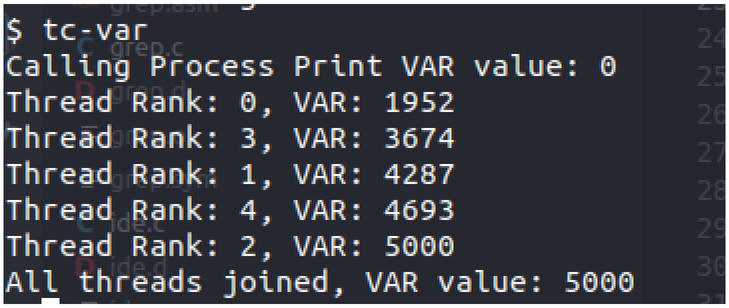
Part 3: Testing Your Implementation

You can make sure that your lock mechanisms are properly implemented. Two test case files are given with other xv6 files. Changes to MakeFile are already made for compiling them.

For a test case named **tc-<something>.c**, run make and make qemu-nox. On the command prompt of xv6, execute **tc-<something>** to run the test case.

1. Global Variable (tc-var.c)

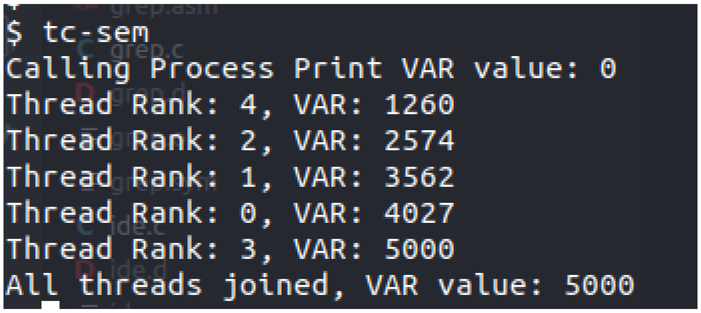
   This test case contains a global variable **initialized** to **zero** and the main process creates N threads and each thread increments the global variable value by 1000. The increment operation is protected by locks so that there will be no interleaving. The print statement is not protected by locks, **so it may print thread_id differently**. Finally, the main program prints the updated variable**, which should be N*1000 (**We will check this only**).** For the given test case, N is 5. To see the difference, you can run your implementation with Lab8 test cases, after removing the sleep statement where the thread is created.



```
$ tc-var
Calling Process Print VAR value: 0
Thread Rank: 0, VAR: 1952
Thread Rank: 3, VAR: 3674
Thread Rank: 1, VAR: 4287
Thread Rank: 4, VAR: 4693
Thread Rank: 2, VAR: 5000
All threads joined, VAR value: 5000
```

2. Semaphore Basic (tc-sem.c)

   This test case is similar to the above scenario, the only difference is we are using semaphores for the lock mechanism. As stated earlier, printing can be interleaved, but final value should be **N*1000.**



```
$ tc-sem
Calling Process Print VAR value: 0
Thread Rank: 4, VAR: 1260
Thread Rank: 2, VAR: 2574
Thread Rank: 1, VAR: 3562
Thread Rank: 0, VAR: 4027
Thread Rank: 3, VAR: 5000
All threads joined, VAR value: 5000
```

Submission Instructions

- All submissions are to be done on moodle only.
- Name your submission as <rollnumber>_lab9.tar.gz (e.g 190050004_lab9.tar.gz)
- The tar should contain the following files in the following directory structure:
  <rollnumber>_lab9/

  |__< all modified files in xv6 such as

  syscall.c, syscall.h, sysproc.c, user.h, usys.S, proc.c, trap.c, defs.h, proc.h, . . . >

  |__Makefile

Please adhere to it strictly.

- Your modified code/added code should be well commented on and readable.
- **`tar -czvf <rollnumber>_lab9.tar.gz <rollnumber>_lab9`**

Deadline:Friday, 28th October 2022, 11:55 PM via moodle.