

# Sprint 3 Report

## Team StuckOverflow

# Implementation.

## Overview.

We are implementing a multithreading library like pthread in linux.

## **Library Interface.**

1. `int kthread_create(void (*fnc)(void *), void *arg);`
2. `int kthread_join(int tid);`
3. `void kthread_exit(void);`
4. `void spinlock_init(spinlock_t *lk);`
5. `void spinlock_acquire(spinlock_t *lk);`
6. `void spinlock_release(spinlock_t *lk);`

Below are the steps we have done to implement each function.

## **kthread create()**

To implement `kthread_create()`, we had to do a system call called `clone`.

- i. **`int clone(void (*fnc)(void*), void* arg, void *stack);`**

This is a system call called `clone`, that functions the same way `fork()` functions but allow the children to share space with the parent and have only their own stack. The function accepts 3 arguments:

1. a pointer to void function.
2. void pointer to the argument.
3. Pointer to the user stack.

How does it work?

In `proc.c`, it is implemented and does this:

1. Validate arguments (Ensures the passed stack is page-aligned and has enough space.).
2. Allocate Thread Structure (Creates a new `proc` structure for the thread).
3. Share Address Space
  - Assign the parent to the thread.
  - Unlike `fork()`, the thread shares the same `pgdir` as the parent.
  - Copies other meta data regularly like `fork()`: size, `trapeframe` (used for register, explained below), open files, current working directory.
4. Setup Stack and Registers.
  - Since the thread has its own stack, initialize the stack pointer (`sp`) to the beginning of the page.

- Since the thread is executing function and receiving argument, move the stack pointer 8 blocks: one for the argument and one for the return address.
    - Here we did a fake return address since the threads don't return, they exit.
  - Set up the base pointer (ebp) and stack pointer (esp) registers to the location of the stack pointer now.
  - Set up the instruction pointer register (eip) to the function to be executed.
  - Place 0 in the eax pointer (like fork and how it returns 0 in the child).
5. Acquire the process table lock and change the thread as RUNNABLE.
  6. Return the thread id.

## ii. **Declare clone() as system call.**

Next, we declared it as system call to call it in kthread\_create.

## iii. **kthread\_create()**

1. In user.h, we added the signature of the function.
2. In ulib.c, we implemented the logic, which works as follows:
  - a. Allocate space in the memory for the stack using sbrk() function that is implemented in xv6.
  - b. Call clone() system call.

## **kthread\_join()**

To implement kthread\_join(), we had to do a system call called join.

## i. **int join(int thread\_id);**

This is a system call called join, that functions the same way wait() functions but allow waits for a specific threads and also doesn't free the memory since it is shared. The function accepts 1 argument, which is the thread\_id to wait for.

How does it work?

1. It acquires the lock on process table and Loops over it.
2. It checks whether there is a child thread with the same thread\_id intended and has the current process as the parent.
3. If found,
  - a. Check if the thread is ZOMBIE (i.e. finished execution and not claimed by the parent).

- i. If so, clean everything except the parent memory (**main difference between join & wait**) and return the thread id.
  - b. If not, put the process to sleep till the child thread wakes it up and then release the table lock and reclaim it when it wakes up.
- 4. If not,
  - a. Can't join -> error.

ii. **Declare join() as system call.**

Next, we declared it as system call to call it in kthread\_join.

iii. **kthread\_create()**

- 3. In user.h, we added the signature of the function.
- 4. In ulib.c, we implemented the logic, which calls join and returns its value.

### **kthread\_exit()**

just calls the exit() system call.

### **spinlock\_t struct**

this is a spinlock type struct that has an integer called locked to determine the state of the lock.

### **void spinlock\_init(spinlock\_t \*lk);**

This function initiates lk->locked = 0.

### **void spinlock\_acquire(spinlock\_t \*lk);**

This function acquires the lock atomically by using the atomic function implemented in xv6, xchg() -> that acquires the lock atomically, avoiding interrupts while acquiring the lock.

### **void spinlock\_release(spinlock\_t \*lk);**

This function releases the lock by resetting lk->locked to 0.

# Testing.

## 1. Basic Kernel Thread Functionality Tests.

**Name of file :**Kthreadtests.c

We have configured all of the testing files into 4 main ones each will test the functionality of our functions the ones that are related, in this report we will go over them one by one with explanations on what each of the functions does along with what the output should be.

So first the function that we started with testing are the basic kernel threads functionalities:

It provides unit testing that verifies that the kernel threading system correctly supports thread creation, execution, and termination. It tests both a single-thread case and multiple concurrent threads.

### **Functions Tested:**

- kthread\_create()
- kthread\_join()
- kthread\_exit()

### **Expected Output (Success Case) should have the following:**

- All threads print start and progress messages.
- The main process successfully joins all created threads.
- The test ends with confirmation messages for both single and multiple thread cases.

Output (SUCCESS!):

The screenshot shows a QEMU terminal window with a dark background. The output text is as follows:

```

Created thread 2 with TID = 6
CreThread
Thread 1: Machine View
Thread 2: Created thread 2 with TID = 6
Thread 3: CreThread 1a: Started
Thread 3: Thread 1: Working... step 0
Thread 3: Thread 2: Started
Thread 3: Thread 3: Started
Thread 2: Thread 3: Working... step 0
Thread 2: ted thread 3 with TID = 7
Thread 1: Thread 2: Working... step 0
Thread 1: step 1Thread 1: Working...Thread 3: Working... step 1
Thread 1: step 1Thread 2: Working... step 1
Thread 3: Thread 3: Working... step 2
Thread 2: Thread 2: Working...Thread 1: Working... step 2
Thread 2: . step 2
Thread 3: Thread 3: About to exit
Thread 3: Thread 1: About to exit
Thread 1: Thread 2Joined thread 1 with TID = 5
Thread 1: Thread 3: About to exit
Thread 2: Joined thread 2 with TID = 6
Thread 1: : About toJoined thread 3 with TID = 7
Thread 1: Joined thr=== Multiple Thread Test Complete ===
Thread 1: Joined thrAll Basic Thread Tests Completed
Thread 1: === Multip
Thread 1:
Thread 1: All Basic Thread Tests Completed

```

## 2. Spinlock Functionality and Concurrency Test.

Name of file: spinlocktest.c

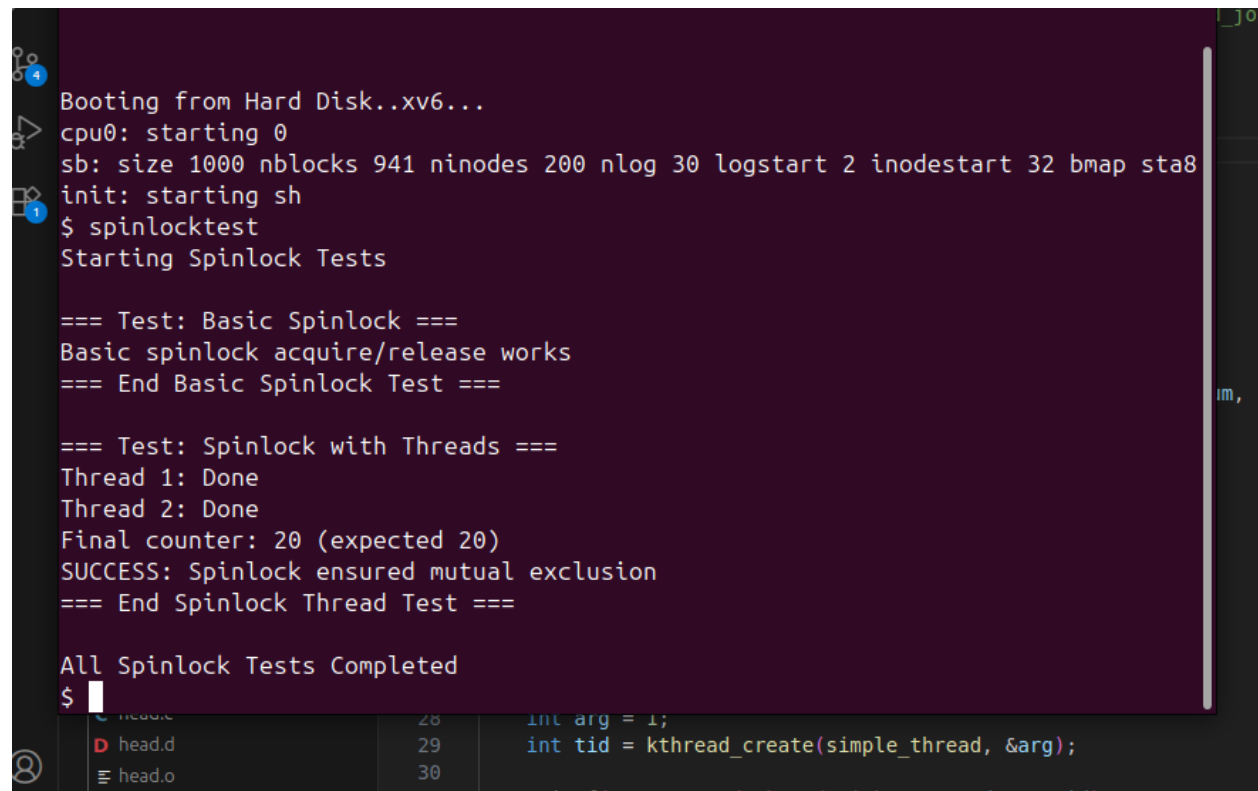
This test verifies:

1. Basic functionality of `spinlock_init`, `spinlock_acquire`, and `spinlock_release`.
2. Correct synchronization of concurrent threads modifying a shared variable using spinlocks.

### Functions Tested:

- `void spinlock_init(spinlock_t *lk)`
- `void spinlock_acquire(spinlock_t *lk)`
- `void spinlock_release(spinlock_t *lk)`
- `int kthread_create(void (*f)(void *), void *arg) // repeated`
- `int kthread_join(int tid)`
- `void kthread_exit(void)`

Output (SUCCESS!):



A terminal window with a dark purple background. The text is white. The output shows the boot process of xv6, starting from a hard disk. It then runs 'spinlocktest', which performs several tests on spinlocks. The tests include a basic spinlock test and a spinlock test with threads. The thread test involves creating two threads that increment a counter, and the final counter is 20, which matches the expected value. The output concludes with 'All Spinlock Tests Completed' and a prompt '\$ '.

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ spinlocktest
Starting Spinlock Tests

=== Test: Basic Spinlock ===
Basic spinlock acquire/release works
=== End Basic Spinlock Test ===

=== Test: Spinlock with Threads ===
Thread 1: Done
Thread 2: Done
Final counter: 20 (expected 20)
SUCCESS: Spinlock ensured mutual exclusion
=== End Spinlock Thread Test ===

All Spinlock Tests Completed
$ 
```

## **Compilation and Integration of of the Test files that we created in xv6**

To compile and run the tests (kthreadtest, spinlocktest) in xv6, we followed these integration steps:

### **1. Adding the Test Files to the UPROGS Section in the Makefile**

Each test program was placed in the user/ directory of the xv6 project (e.g., user/kthreadtest.c, user/spinlocktest.c, etc.).

To ensure they are included in the xv6 user binary image and accessible from the xv6

---

### **2. Adding to the EXTRA Section (if needed)**

If any of the new files (e.g., additional .c or .h files like spinlock.c) are not automatically compiled, they can also be added to the EXTRA section of the Makefile// we did this to ensure that no errors occur:

---

### **3. Building xv6**

After updating the Makefile and placing the test files correctly, we compiled the entire xv6 project using:

---

### **4. Running xv6 and Executing Tests**

We launched xv6 in text-only mode using:

```
make qemu-nox
```

Once inside the xv6 shell (\$), each test program could be run by typing its name:

```
$ kthreadtest
```

```
$ spinlocktest
```

Each test program produced output directly to the xv6 console, showing the result of the test cases for thread creation, synchronization primitives, and condition handling.