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 (*fcn)(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_acquire(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.

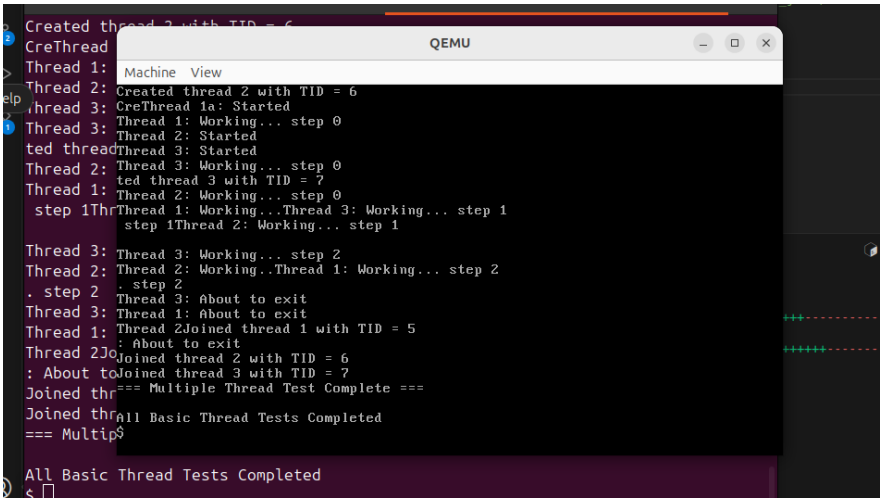
Functions Tested:

- 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!):



---

2. Spinlock Functionality and Concurrency Test.

Name of file: spinlocktest.c

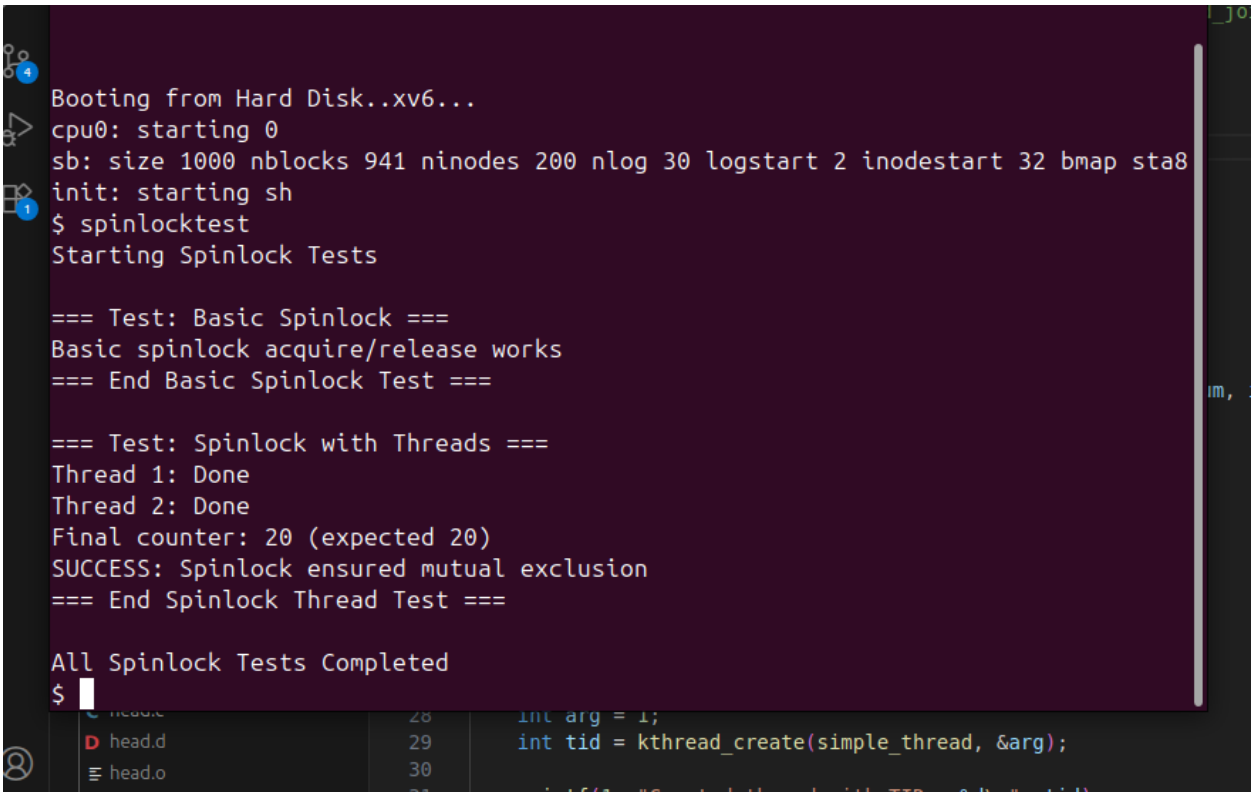
This test verifies:

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!):



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.