CS 344 ASSIGNMENT - 1

GROUP MEMBERS :

Arya Avinash Phadke 200101020
Mansi 200101064
Pranshu Pandya 200101073

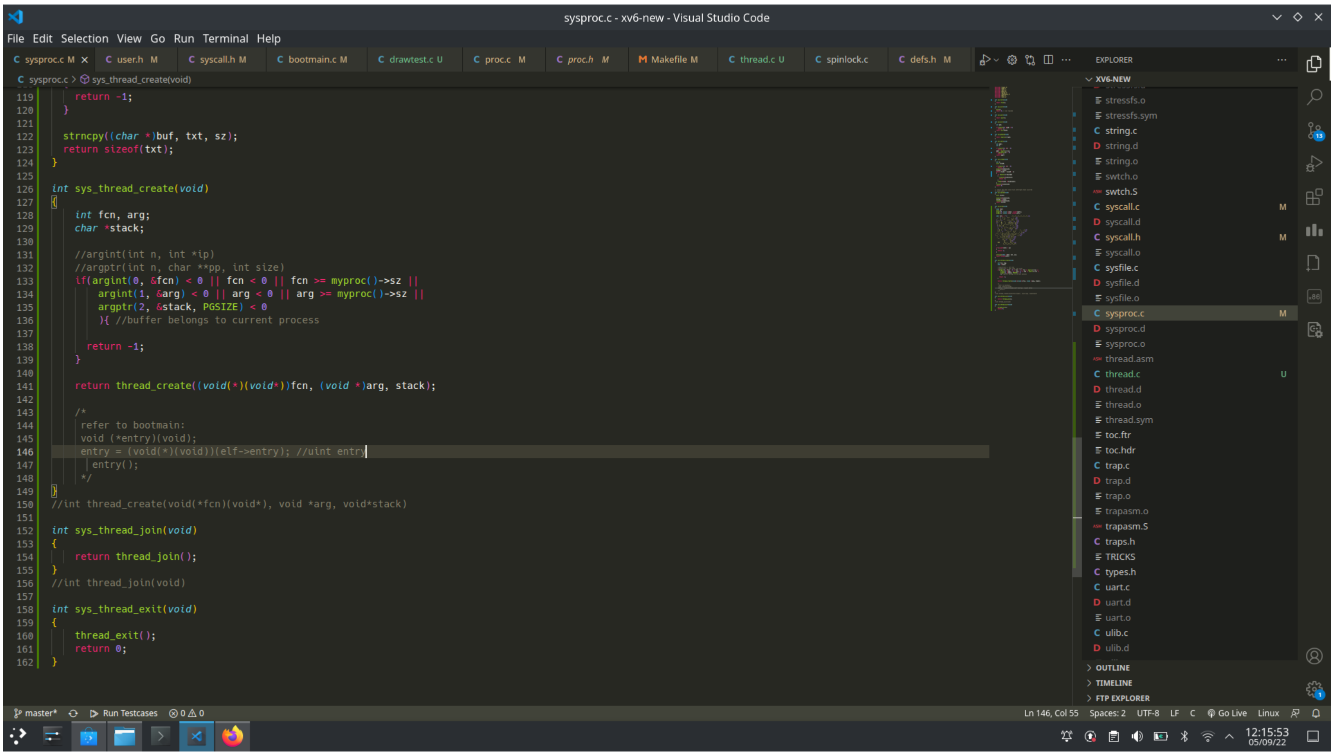
In this assignment, we had to implement kernel threads and then had to build spinlocks and mutexes to synchronize the access among the threads.

PART 1: Implementation of kernel threads

We have implemented system calls for **thread_create**, **thread_join** and **thread_exit**. For this we had to make changes in sysproc.c, user.h, syscall.h, syscall.c, proc.c, proc.h, makefile, defs.h, usys.S.

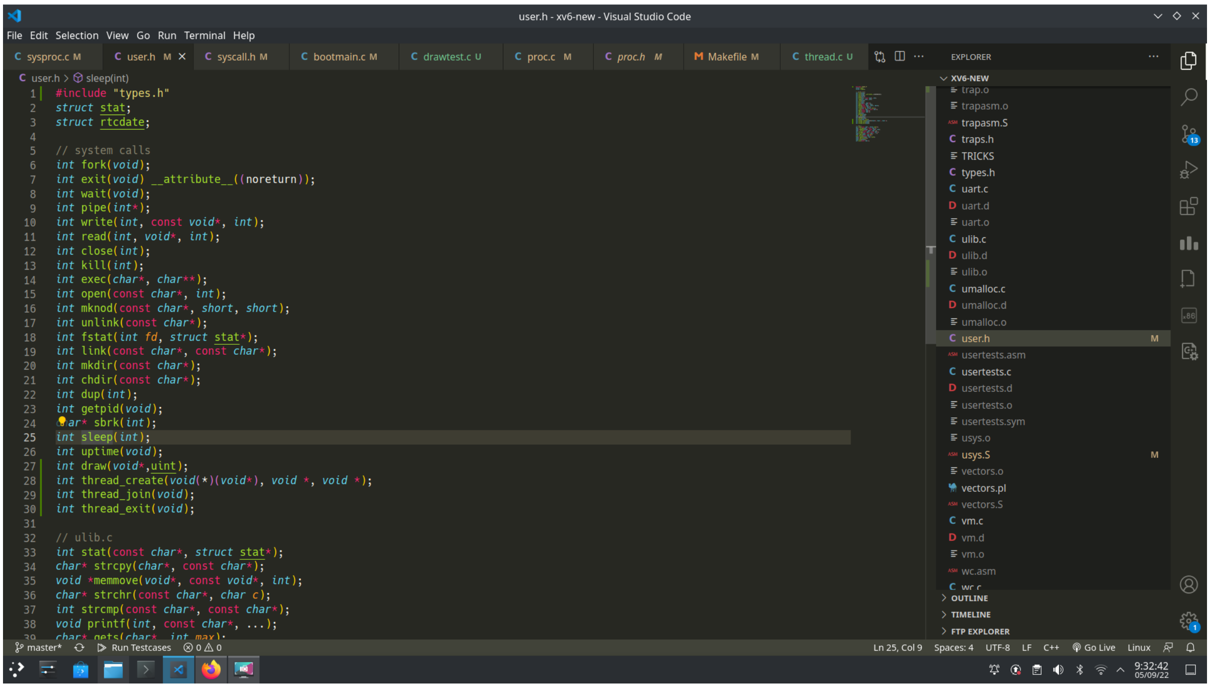
Sysproc.c

Here we have defined the system calls which call the main functions in proc.c where we have implemented thread_create, thread_join, thread_exit.



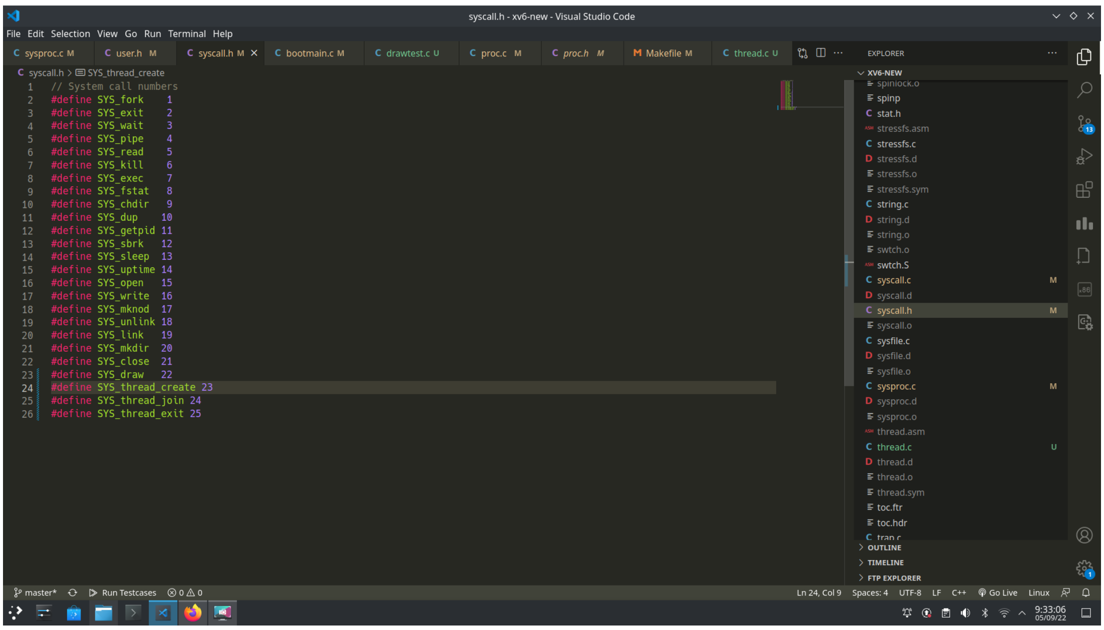
User.h

In this file, we have just declared our functions, thread_create, thread_join and thread_exit, in lines 28,29 and 30.



Syscall.h

In this file, we have just defined the names of our functions as SYS_ thread_create, SYS_thread_join, and SYS_thread_exit.



Proc.c

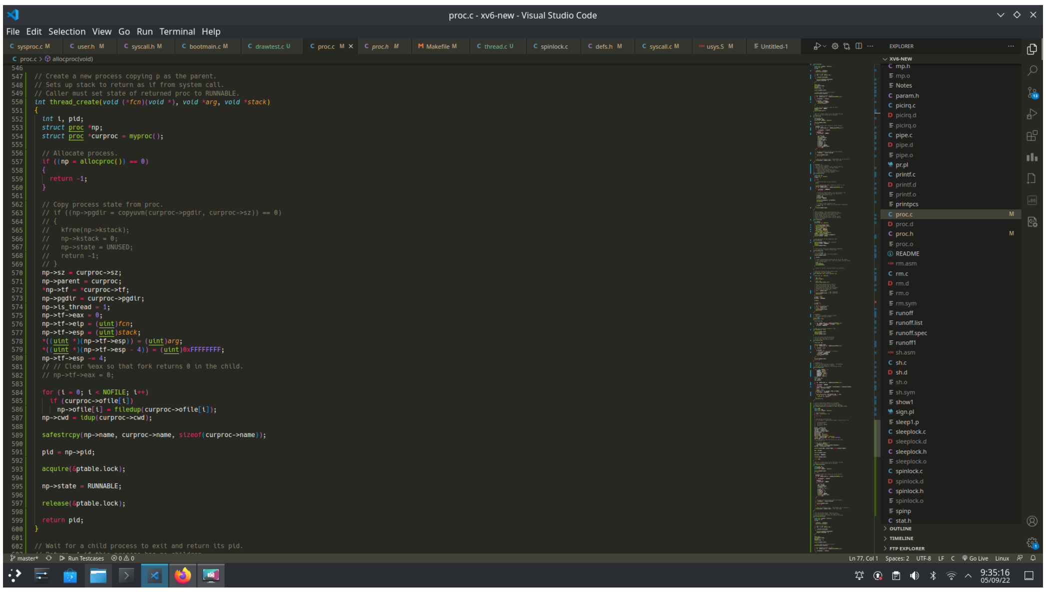
thread_create()

This function creates a new thread by forking the main process. This function has been taken up from the existing fork() function. The procedure for creating a new thread is almost the same with some changes which are explained below:

The function theread_create takes in, the following arguments:

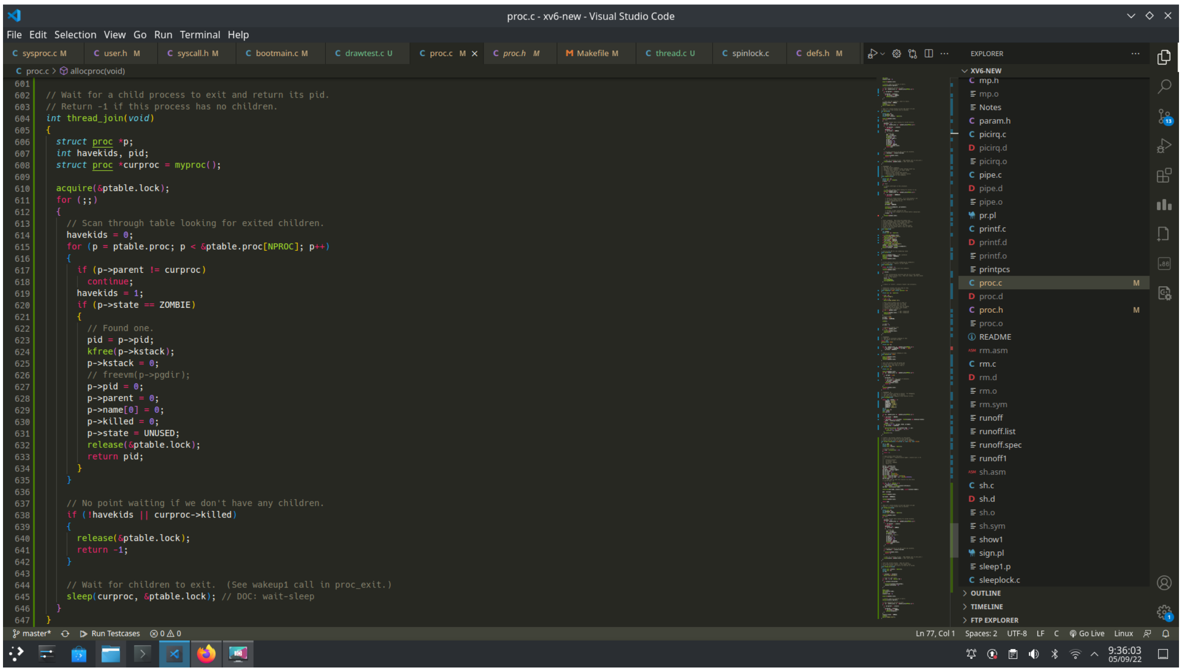
1. The function which will be using this thread for implementation. The new thread starts executing at the address space specified by this function.
2. The arguments required for the above function.
3. The user stack pointer of the stack to be used by this thread.

Firstly, we check if a process has been allocated. We are ensuring that there is a new process to be allocated to our thread. We have commented out the next few lines. In this they basically copy the process state of the parent process to the new process. We don't need to do this because we are spawning a new thread and not a process. Then a few things are initialized for the new thread we are making : the size is set as the parent's size, the trapframe for the new thread is set to that of the parent, the page directory(address space) for the new thread is set to that of the parent. Note here we are setting the is_thread flag as 1, to denote that we are creating a new thread.
Then, we are setting the extended instruction pointer to our function fcn, and we are setting np->tf->esp-4 = (uint)0xFFFFFFFF , by which we are setting a fake program counter for the user stack used to implement the threads.
Now, because the thread has access to all the files opened by the parent process, we are spanning through all the files, and are duplicating and associating the files opened by the main process to the newly created thread.
Finally the state of the new thread is set. Here the state is set to RUNNABLE indicating that this thread is ready to be executed, and the thread is put in the ready queue. It is done inside locks to make sure that it is executed together and that context switching does not happen before the state has been completely set.



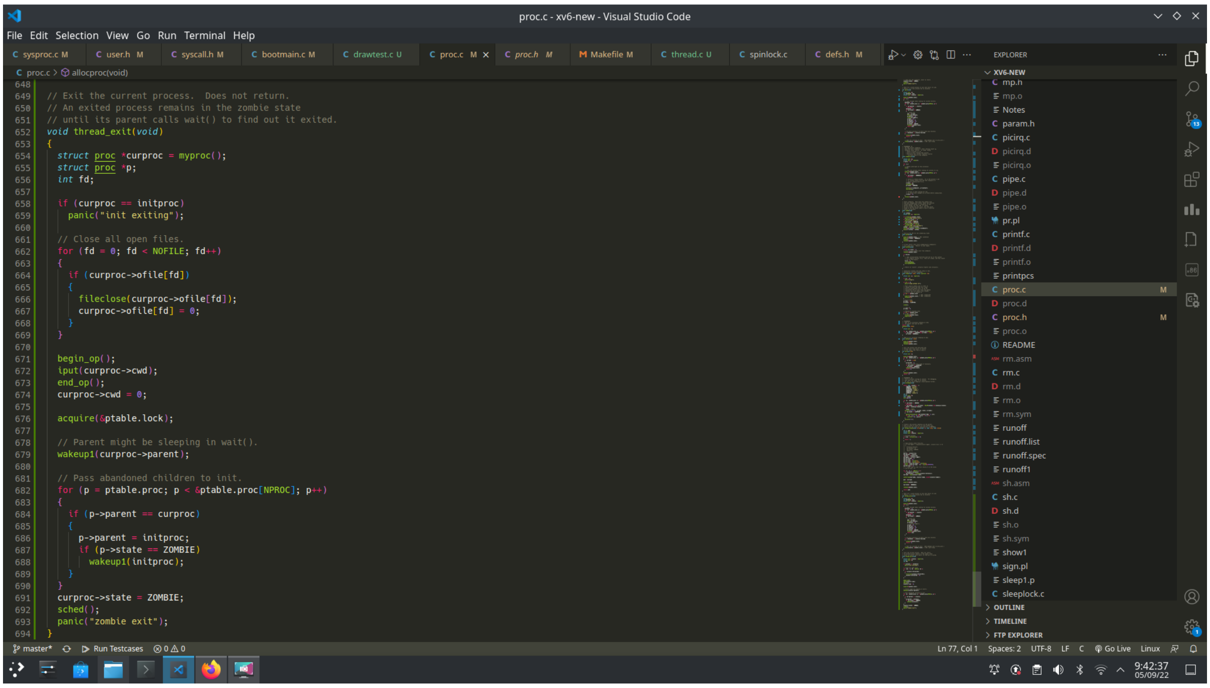
thread_join()

This function has been created using the wait() function which was already present in xv6. It takes nothing as the argument as it has to wait for the thread to complete its execution and then has to terminate the thread. In the function, we scan through all the threads/processes in the ptable and then, for the processes which are the child process(threads) of our main process, we check if the thread has completed the execution and has entered the zombie state, we deallocate all the resources given to the zombie thread and finally kill it and return the pid of the killed thread. If there is no thread in the zombie state i.e. all are still executing, it will wait for them to complete the execution and enter in the zombie state so as to be killed, and therefore, enters the sleep state. At the end, when all the threads are killed i.e. have_kids=0 or the current process is killed, we come out of the thread_join() function returning -1 to indicate that the process is completed and thread_join() has nothing to do now, otherwise, the pid of the killed thread is returned.



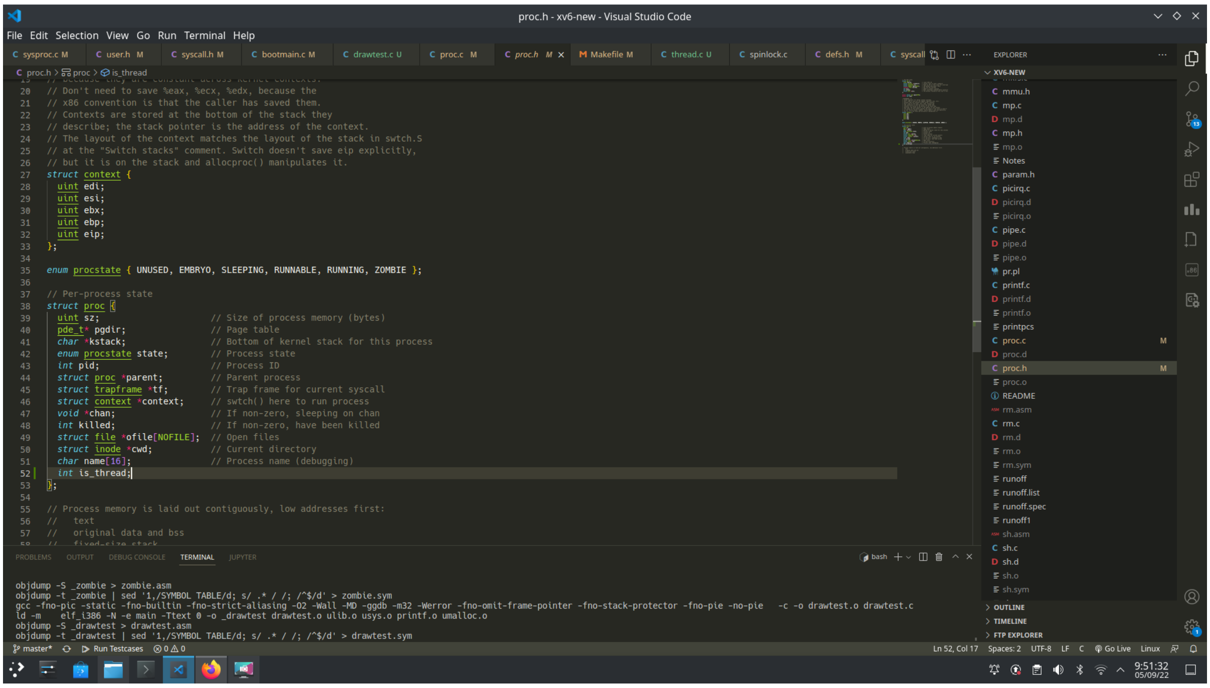
thread_exit()

This function allows a thread to terminate. Again, it takes no argument as it checks for all the processes in the ptable to check if they are completed. This has also been picked up from the already existing exit function. The first thing we do is make sure there are no open files associated with this thread, and if there are, we close them.Now we check if the parent is in sleep state, if it is then we run the wakeup1 function, and then we set the state of the thread as zombie, indicating that the process has completed.



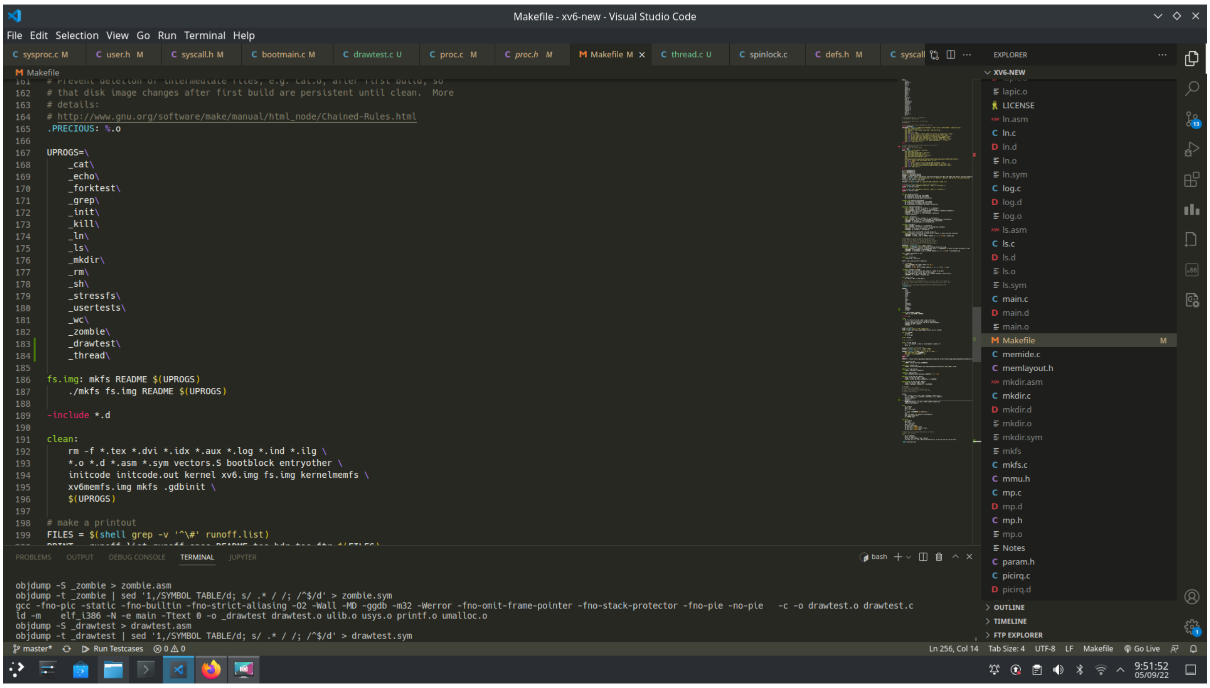
Proc.h

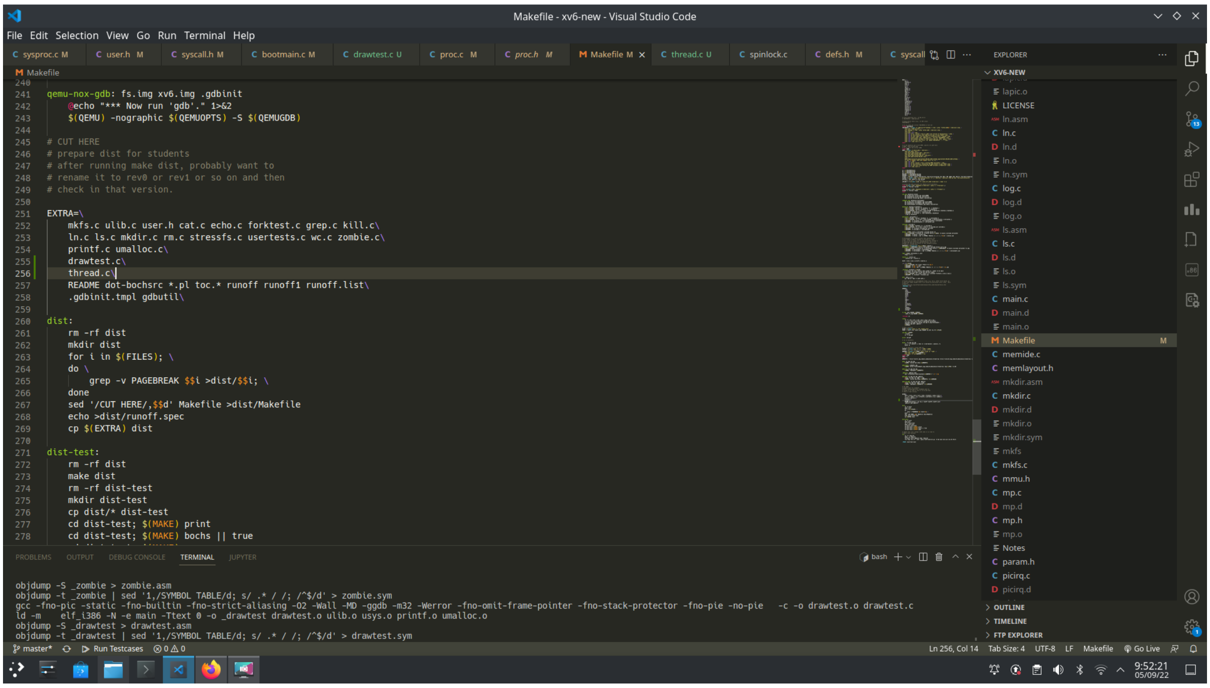
In this file, we are making a change to the struct proc - we are adding another int field called is_thread. This is done so we do not have to implement an entire new struct to represent a thread. If the value of is_thread is 1, then the struct represents a thread, and if it is 0, then the struct represents a process.



Makefile

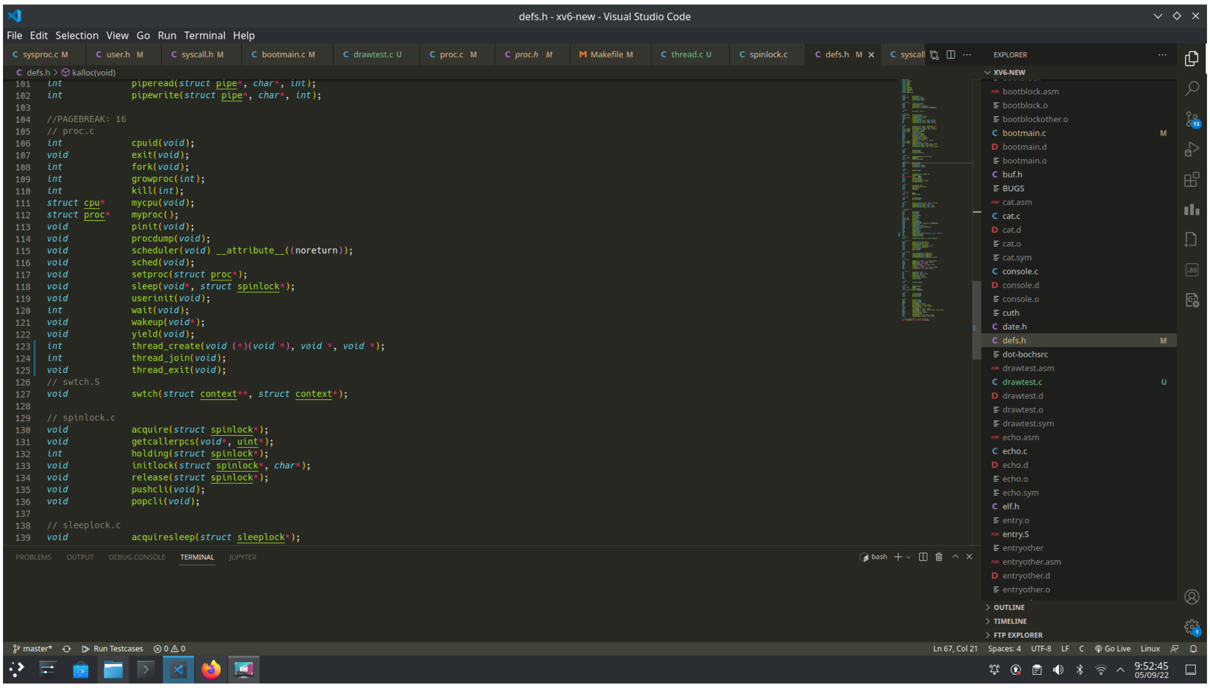
Here we are just mentioning 'thread' under UPROGS to show that this is also a command which the user can enter. Under extra we are mentioning that the thread.c file also has to be compiled and run.





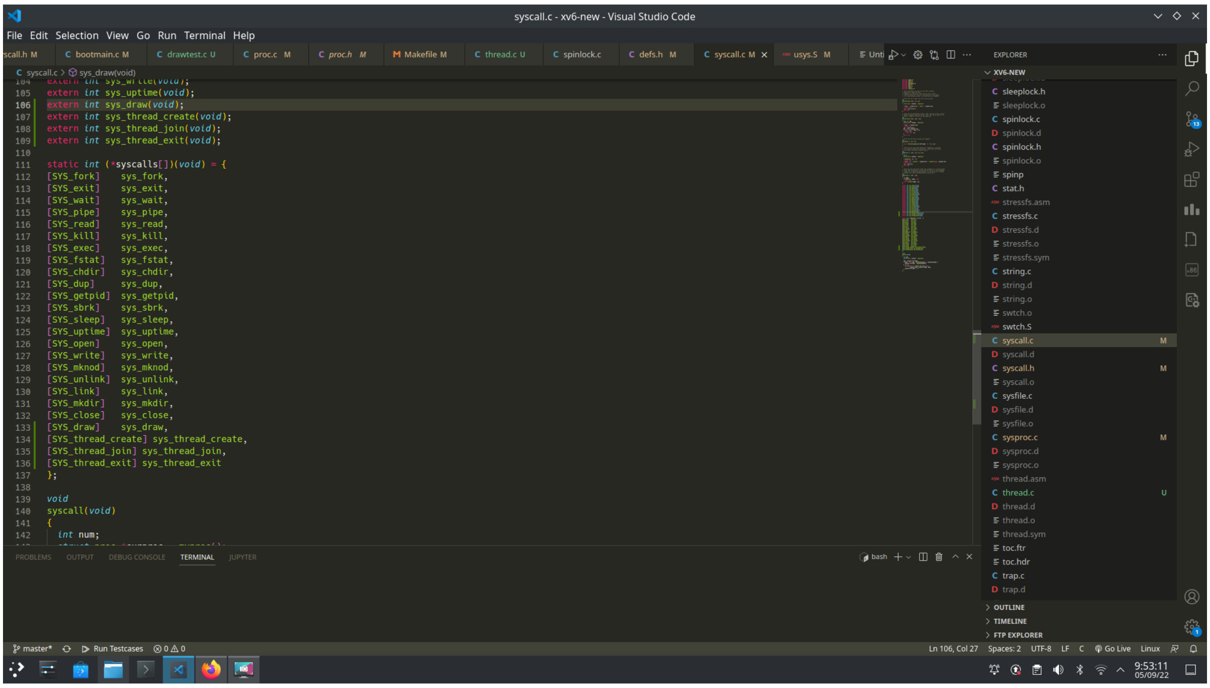
Defs.h

We are defining all the functions which we have created. For part1, we have created three new functions : thread_create, thread_join and thread_exit.



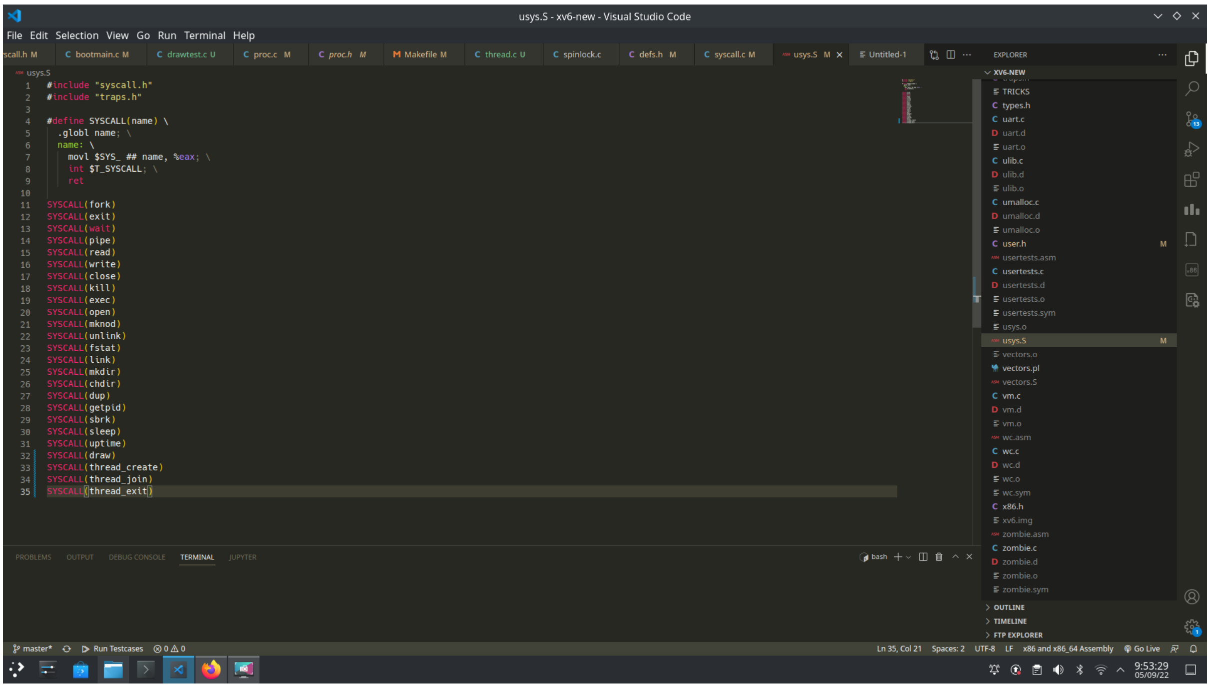
Syscall.c

Here we are defining our function and we will call the main function which we have written in proc.c



usys.S

This is added to implement the system call for thread.

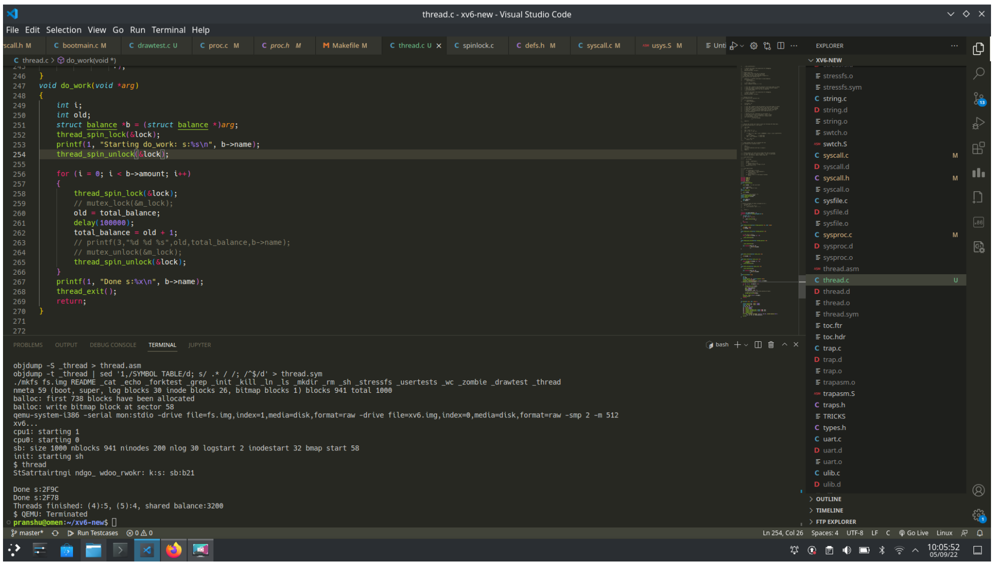


Part 2: Synchronization

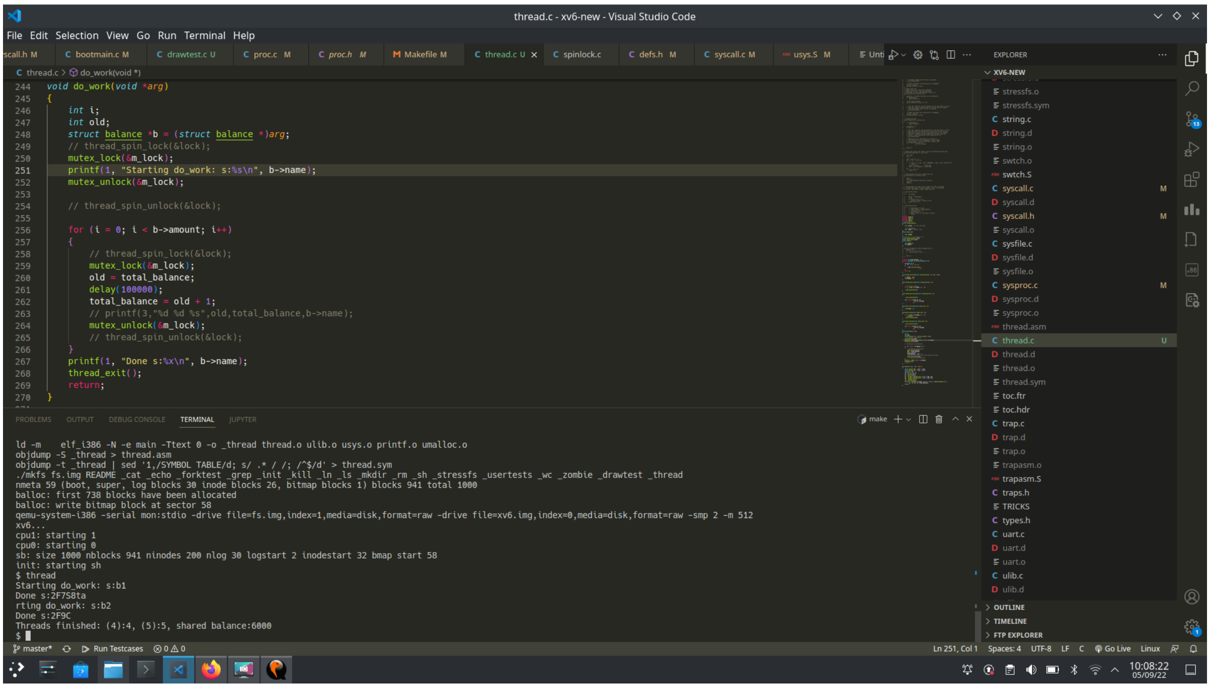
Output without synchronization

Here after executing the thread command without implementing any locks, we can see that the total balance is not 6000. This happens because there is no synchronization and context switching keeps happening. Here total balance is the shared variable, and due to context switching, before the entire set of instructions in the critical section of one process gets executed, context switching occurs and the other process uses the outdated value of the variable only hence the total does not add up to 6000. This can be prevented by adding locks.

The delay statement has been added here to show that not adding locks causes a problem. The delay statement ensures that a context switch happens in between the critical section.



Output with spinlocks



We have defined a struct called lock - it has two fields : a name given to the lock and the int which holds the value of the lock.

```
struct thread_spinlock
{
   uint locked; // Is the lock held?

   // For debugging
   char *name;  // Name of lock.
};
```

thread _initlock - this is the initialization function. We are setting the name of the lock variable and setting the value of lk to 0. This means that no thread is executing its critical section right now.

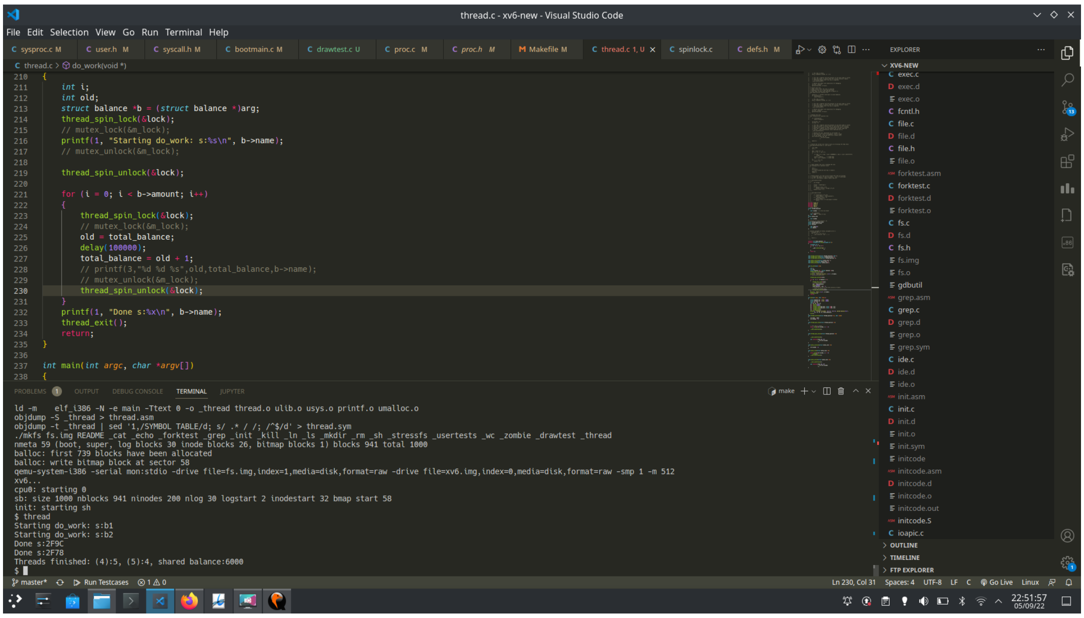
```
void thread_initlock(struct thread_spinlock *lk, char *name)
{
   lk->name = name;
   lk->locked = 0;
}
```

thread_spin_lock - the xchg instruction in the while loop condition basically compares the value of the parameters passed, here it compares the value of the lock and 1. It returns the value of lock and sets the value of the lock to 1(the second parameter). We use this special instruction as it is atomic and hence there is no risk of context switching. So the thread will be stuck in the while loop if the value of lock is 1, meaning some other thread is executing their critical section. The while loop keeps running until the value of lock becomes 0 (busy waiting) and then we come out of the while loop. Now the value of the lock is 1 again and the current thread has 'acquired' the lock, meaning it can execute its critical section safely. __sync_synchronize() is used to ensure that no memory operand will be moved across the operation, either forward or backward. Further, instructions will be issued as necessary to prevent the processor from speculating loads across the operation and from queuing stores after the operation.

```
void thread_spin_lock(struct thread_spinlock *lk)
{

   // The xchg is atomic.
   while (xchg(&lk->locked, 1) != 0)
       ;
   __sync_synchronize();
}
```

thread_unlock - This function sets the value of lock back to 0. This is executed after the critical section has been completed to set the lock back to 0, now other processes can execute their critical sections.

```
void thread_spin_unlock(struct thread_spinlock *lk)
{

   __sync_synchronize();

   asm volatile("movl $0, %0"
                : "+m"(lk->locked)
                :);
}
```



Output with spinlock

Mutexes

Now, we will implement synchronization with the help of mutex. The advantage that mutex has over spin locks is that it won't wait in the while loop while another process is running the critical section. Instead, it goes into sleep, and frees up the processor.

We have defined the following functions for mutex implementation:

Firstly, we define the structure of mutex_lock where, we define just one parameter, which is the value of the lock, 1 represents lock is activated, and 0 represents the lock is free.

```
struct mutex_lock
{
    uint locked;
};
```

mutex_initlock

This function initializes the lock, and sets it to 0.

```
void mutex_initlock(struct mutex_lock *lk)
{
    lk->locked = 0;
}
```

mutex_lock

This function takes in the lock as the argument. Here, the xchg is the swap and compare function. Therefore, if the lock has a value of 0, we won't enter the while loop as in the condition for the while loop, we compare the value of lock with 1, as it was initially 0, the value returned will be 0, and the value of lock will be set to 1. If the value of lock is 1 when the mutex_lock is called, it would enter the while loop and wait there until the value of lock is not reset to 0 as, until then, it would return 1 only, and therefore, would be stuck in the while loop. Now, after this, the process would sleep and the critical region would be executed.

```
void mutex_lock(struct mutex_lock *lk)
{
   while (xchg(&lk->locked, 1) != 0)
       sleep(1);
   __sync_synchronize();
}
```

mutex_unlock

This function releases the lock by setting the value of lock back to 0. The asm volatile("movl $0,%0"

: "+m"(lk->locked)

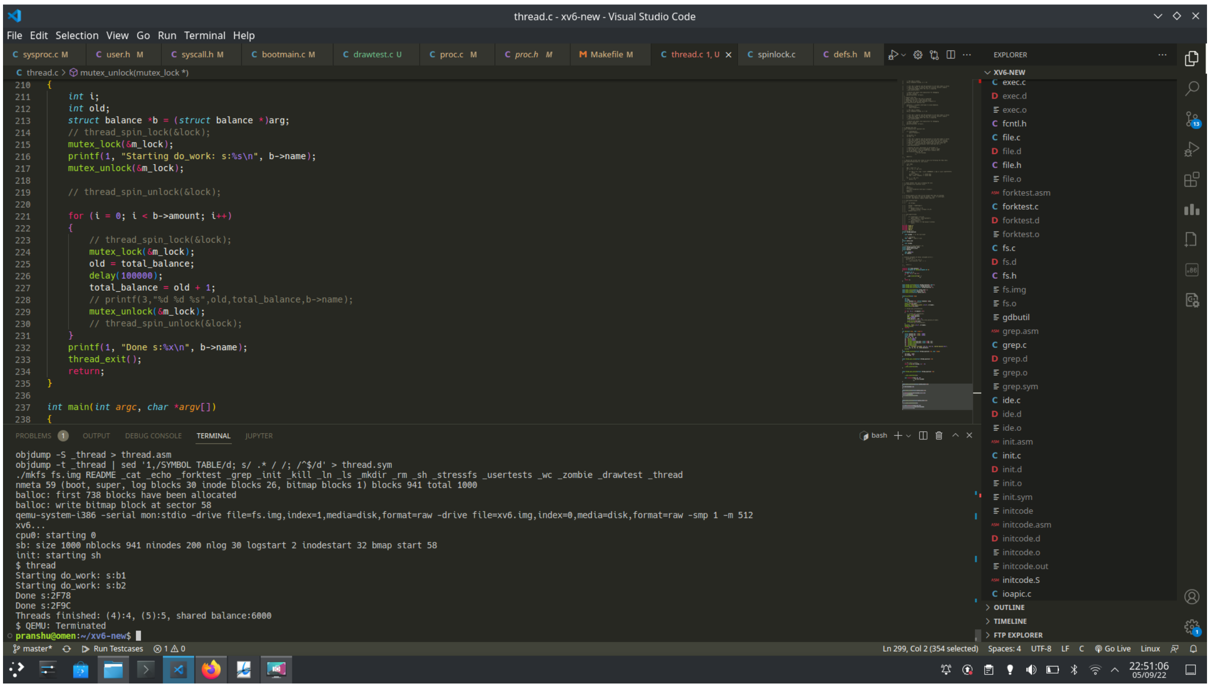
:);

part sets the lock to 0 atomically.

```
void mutex_unlock(struct mutex_lock *lk)
{
   __sync_synchronize();

   asm volatile("movl $0, %0"
                : "+m"(lk->locked)
                :);
}
```

Output

Now, using this mutex lock around the critical section (lines 225-228), we find that the total balance comes out to be 6000, which shows that the lock was implemented correctly. The screenshot of the output is shown below:


thread.c - xv6-new - Visual Studio Code
File Edit Selection View Go Run Terminal Help
thread.c > mutex_unlock(mutex_lock *)
{
int i;
int old;
struct balance *b = (struct balance *)arg;
// thread_spin_lock(&lock);
mutex_lock(&m_lock);
printf(1, "Starting do_work: s:%s\n", b->name);
mutex_unlock(&m_lock);
// thread_spin_unlock(&lock);
for (i = 0; i < b->amount; i++)
{
// thread_spin_lock(&lock);
mutex_lock(&m_lock);
old = total_balance;
delay(100000);
total_balance = old + 1;
// printf(3,"%d %d %s",old,total_balance,b->name);
mutex_unlock(&m_lock);
// thread_spin_unlock(&lock);
}
printf(1, "Done s:%x\n", b->name);
thread_exit();
return;
}
int main(int argc, char *argv[])
{
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
objdump -S _thread > thread.asm
objdump -t _thread | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > thread.sym
./mkfs fs.img README _cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests _wc _zombie _drawtest _thread
nmeta 59 (boot, super, log blocks 30 inode blocks 26, bitmap blocks 1) blocks 941 total 1000
balloc: first 738 blocks have been allocated
balloc: write bitmap block at sector 58
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 1 -m 512
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ thread
Starting do_work: s:b1
Starting do_work: s:b2
Done s:2F78
Done s:2F9C
Threads finished: (4):4, (5):5, shared balance:6000
$ QEMU: Terminated
pranshu@omen:~/xv6-new$
master* Run Testcases
22:51:06
05/09/22