CS 344 ASSIGNMENT - 1

GROUP MEMBERS :-

- Jwalit Bharatkumar Devalia 200123026
- Akshat Jain 200123005
- Varun Bhardwaj 200123068

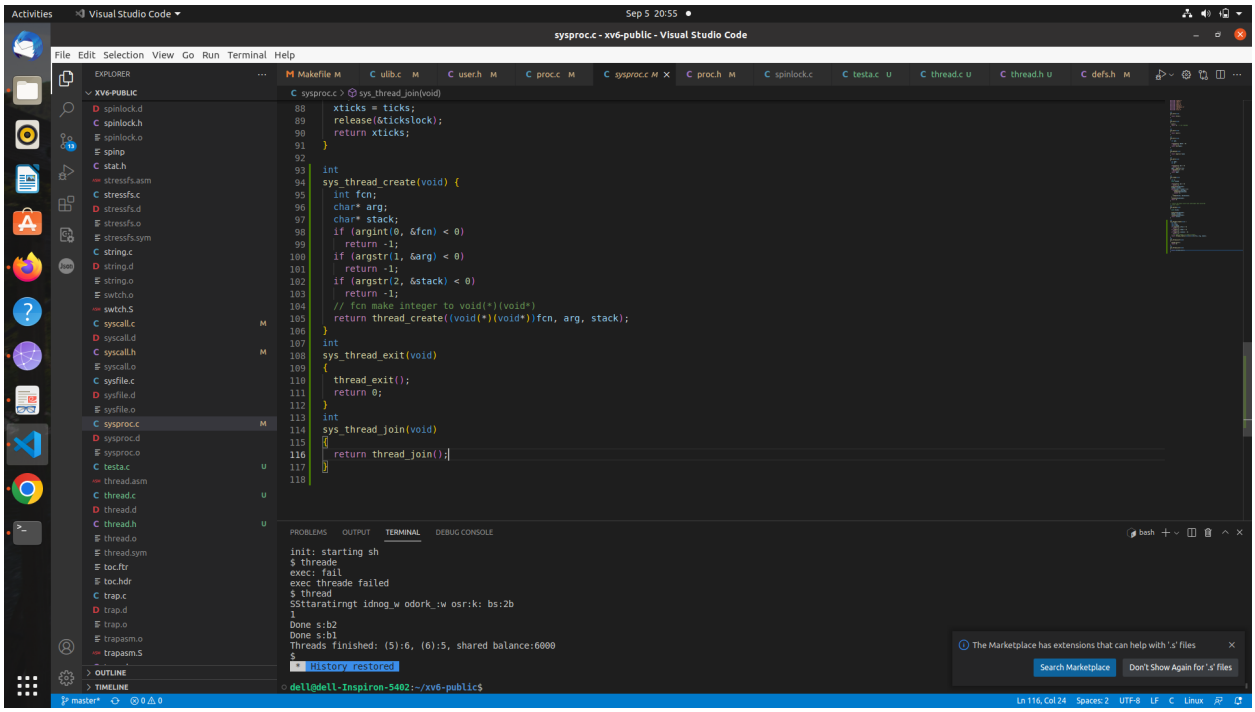
We have implemented kernel threads and then built spinlocks and mutexes to perform synchronization among the threads.

PART 1: Implementation of kernel threads

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

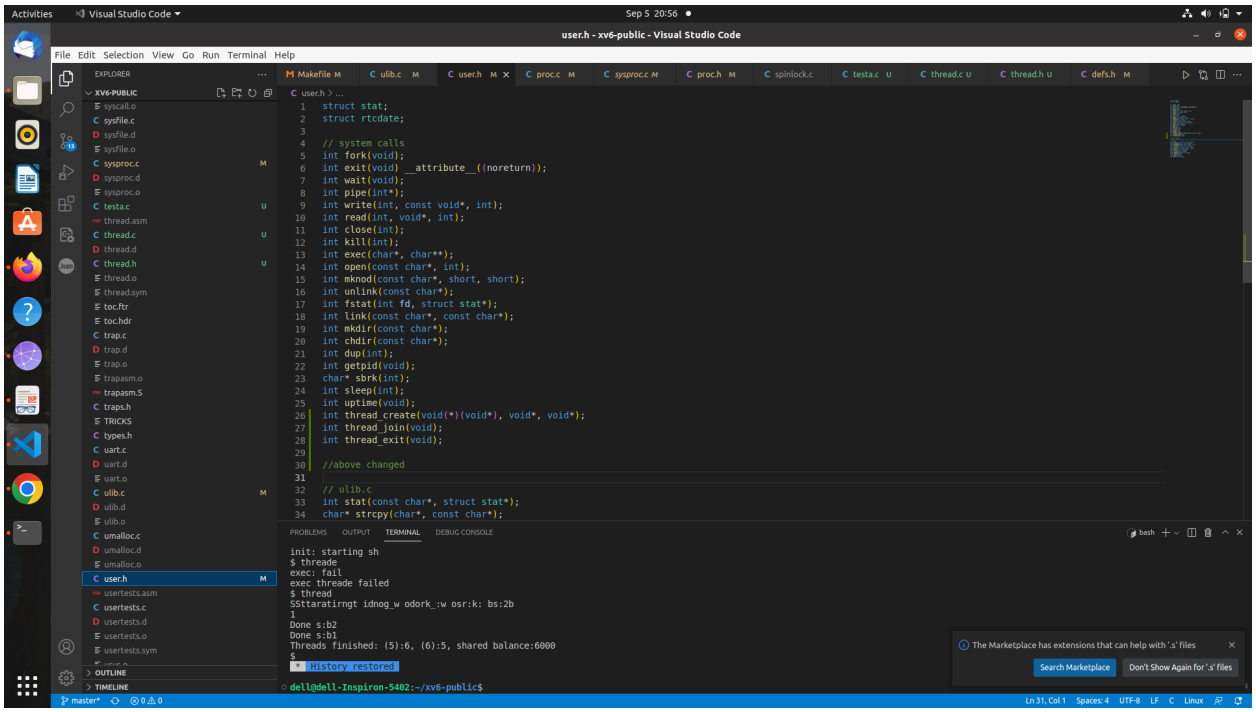
Sysproc.c

We have defined the functions which call the main functions in proc.c where we have implemented threads.



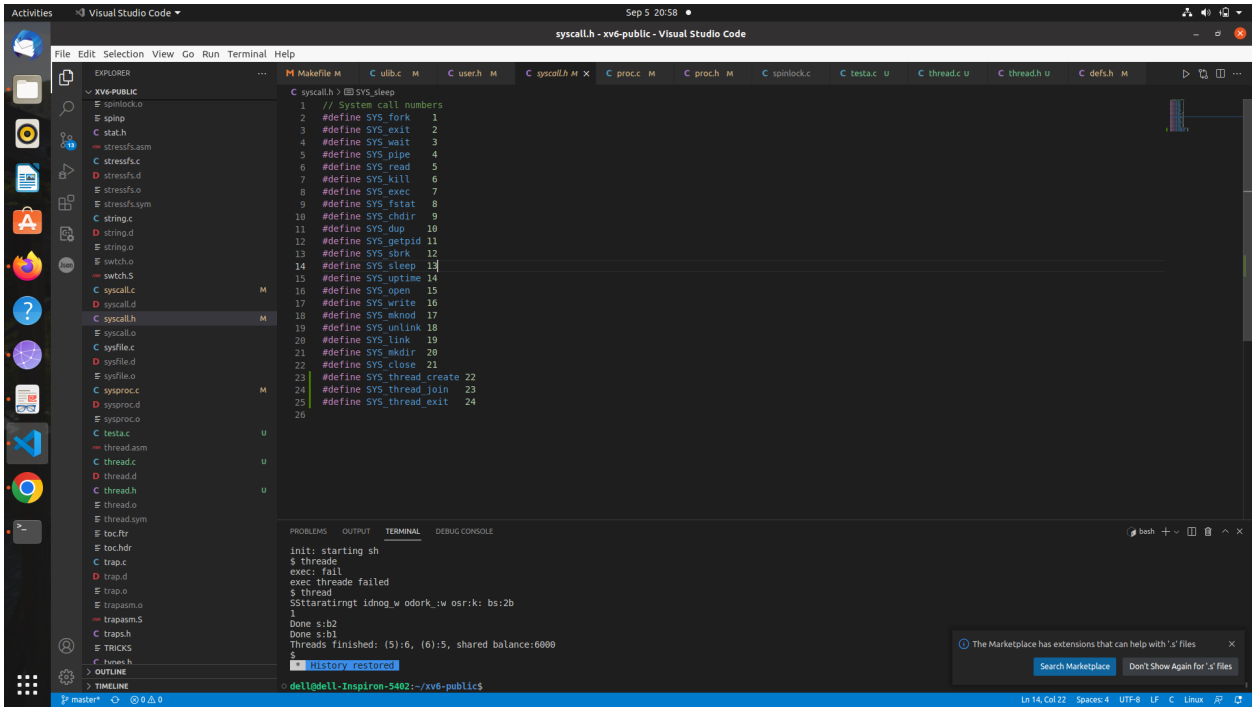
User.h

Here we have declared our functions, thread_create, thread_join and thread_exit.



Syscall.h

We have just assigned numbers to system calls namely SYS_ thread_create, SYS_thread_exit and SYS_thread_join.



Proc.c

thread_create():

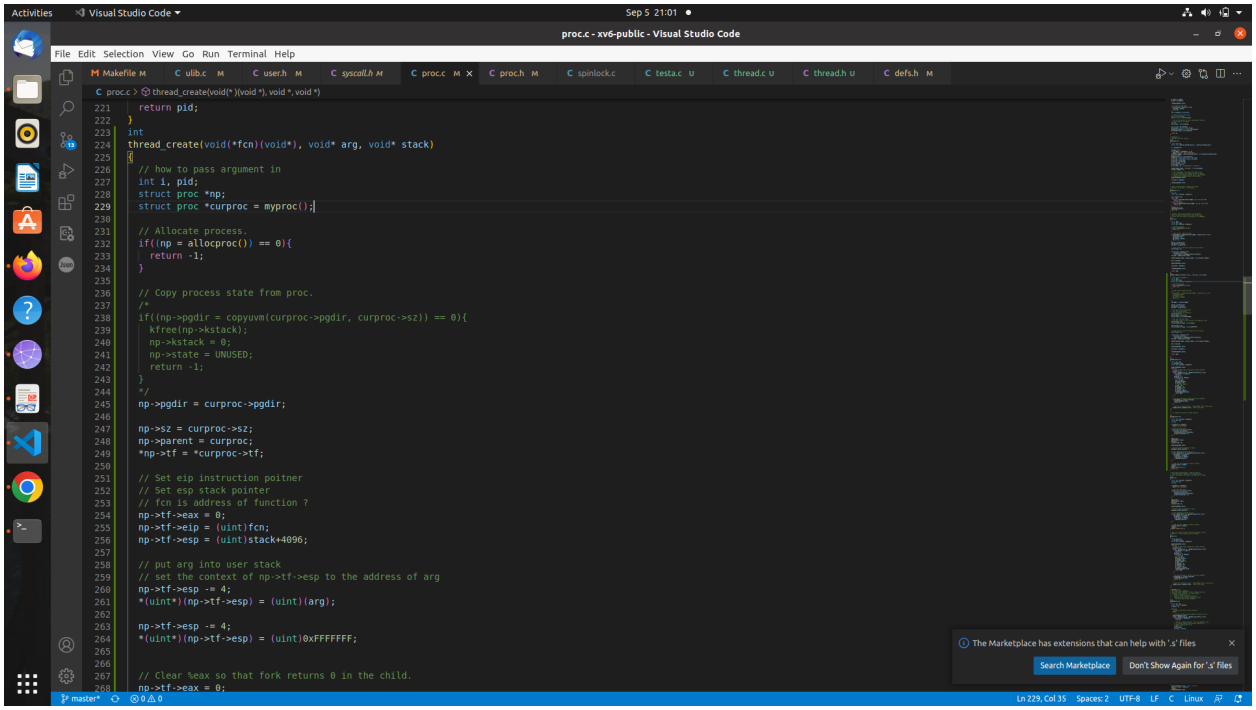
This function creates a new thread. This function is similar to the inbuilt fork function used for making new processes.

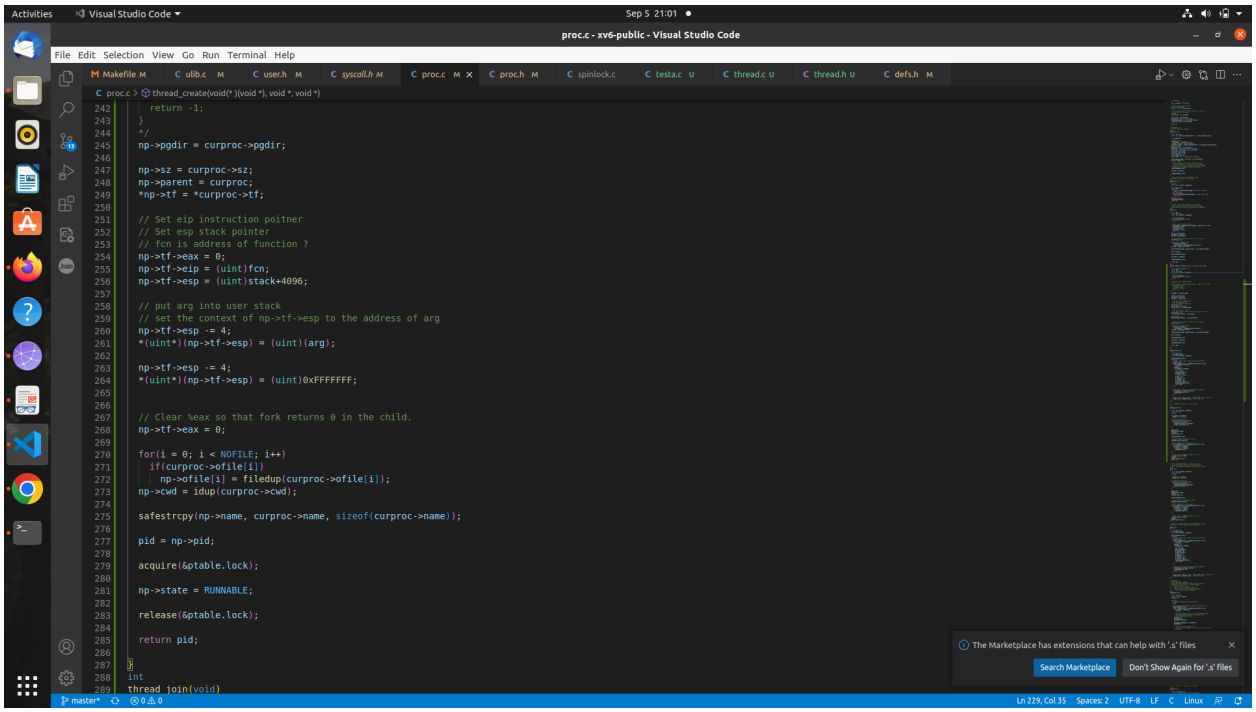
The following changes have been made:-

The function thread_create takes in 3 arguments:

1. The function that will use this thread for implementation.The new thread starts execution 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.

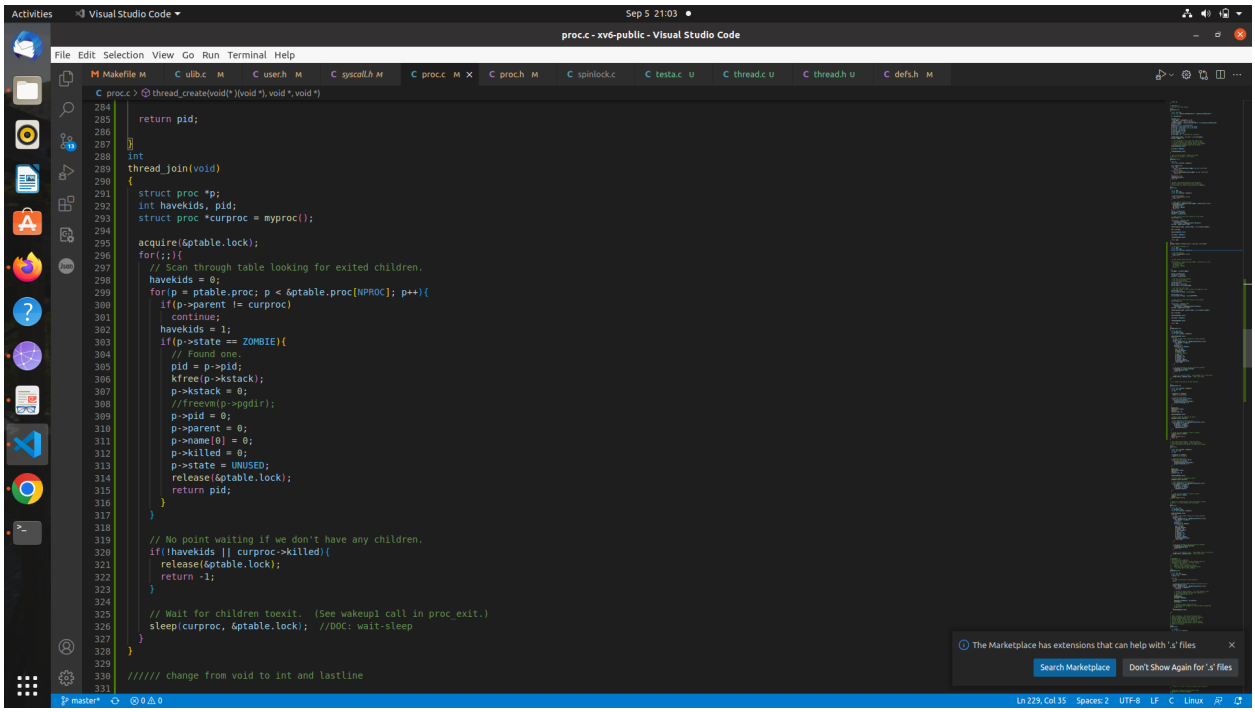
We are ensuring that there is a new process to be allocated to our thread.Few of the next lines have been commented on because they copy the process state of the parent process to the new process . We don't need to do this because we are spawning a new thread and not a process. The size of the new thread is set as the parent's size, the value of the trapeframe of the parent process is given to that of the trapeframe of our thread ,the page directory(address space) for the new thread is set to that of the parent. 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 we have to see what all files will be required by the thread and open them. This is what is happening in the for loop. If the required file has already been opened by curproc, then it gets duplicated, otherwise the new thread opens the file. 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 is set to RUNNABLE indicating that this thread is ready to be executed.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.


Activities
Visual Studio Code
Sep 5 21:01
proc.c - xv6-public - Visual Studio Code
File Edit Selection View Go Run Terminal Help
Makefile M
ulib.c M
user.h M
syscall.h M
proc.c M
proc.h M
spinlock.c
testa.c U
thread.c U
thread.h U
defs.h M
proc.c > thread_create(void(*)(void *), void *, void *)
return pid;
}
int
thread_create(void(*fcn)(void*), void* arg, void* stack)
{
// how to pass argument in
int i, pid;
struct proc *np;
struct proc *curproc = myproc();
// Allocate process.
if((np = allocproc()) == 0){
return -1;
}
// Copy process state from proc.
/*
if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
kfree(np->kstack);
np->kstack = 0;
np->state = UNUSED;
return -1;
}
*/
np->pgdir = curproc->pgdir;
np->sz = curproc->sz;
np->parent = curproc;
*np->tf = *curproc->tf;
// Set eip instruction poitner
// Set esp stack pointer
// fcn is address of function ?
np->tf->eax = 0;
np->tf->eip = (uint)fcn;
np->tf->esp = (uint)stack+4096;
// put arg into user stack
// set the context of np->tf->esp to the address of arg
np->tf->esp -= 4;
(uint)(np->tf->esp) = (uint)(arg);
np->tf->esp -= 4;
(uint)(np->tf->esp) = (uint)0xFFFFFFFF;
// Clear %eax so that fork returns 0 in the child.
np->tf->eax = 0;
The Marketplace has extensions that can help with '.s' files
Search Marketplace
Don't Show Again for '.s' files
master*
0 0
Ln 229, Col 35
Spaces: 2
UTF-8
LF
C
Linux


Activities
Visual Studio Code
Sep 5 21:01
proc.c - xv6-public - Visual Studio Code
File Edit Selection View Go Run Terminal Help
Makefile M
ulib.c M
user.h M
syscall.h M
proc.c M
proc.h M
spinlock.c
testa.c U
thread.c U
thread.h U
defs.h M
proc.c > thread_create(void(*)(void *), void *, void *)
return -1;
}
*/
np->pgdir = curproc->pgdir;
np->sz = curproc->sz;
np->parent = curproc;
*np->tf = *curproc->tf;
// Set eip instruction poitner
// Set esp stack pointer
// fcn is address of function ?
np->tf->eax = 0;
np->tf->eip = (uint)fcn;
np->tf->esp = (uint)stack+4096;
// put arg into user stack
// set the context of np->tf->esp to the address of arg
np->tf->esp -= 4;
(uint)(np->tf->esp) = (uint)(arg);
np->tf->esp -= 4;
(uint)(np->tf->esp) = (uint)0xFFFFFFFF;
// Clear %eax so that fork returns 0 in the child.
np->tf->eax = 0;
for(i = 0; i < NOFILE; i++)
if(curproc->ofile[i])
np->ofile[i] = filedup(curproc->ofile[i]);
np->cwd = idup(curproc->cwd);
safestrcpy(np->name, curproc->name, sizeof(curproc->name));
pid = np->pid;
acquire(&ptable.lock);
np->state = RUNNABLE;
release(&ptable.lock);
return pid;
}
int
thread_join(void)
The Marketplace has extensions that can help with '.s' files
Search Marketplace
Don't Show Again for '.s' files
master*
0 0
Ln 229, Col 35
Spaces: 2
UTF-8
LF
C
Linux

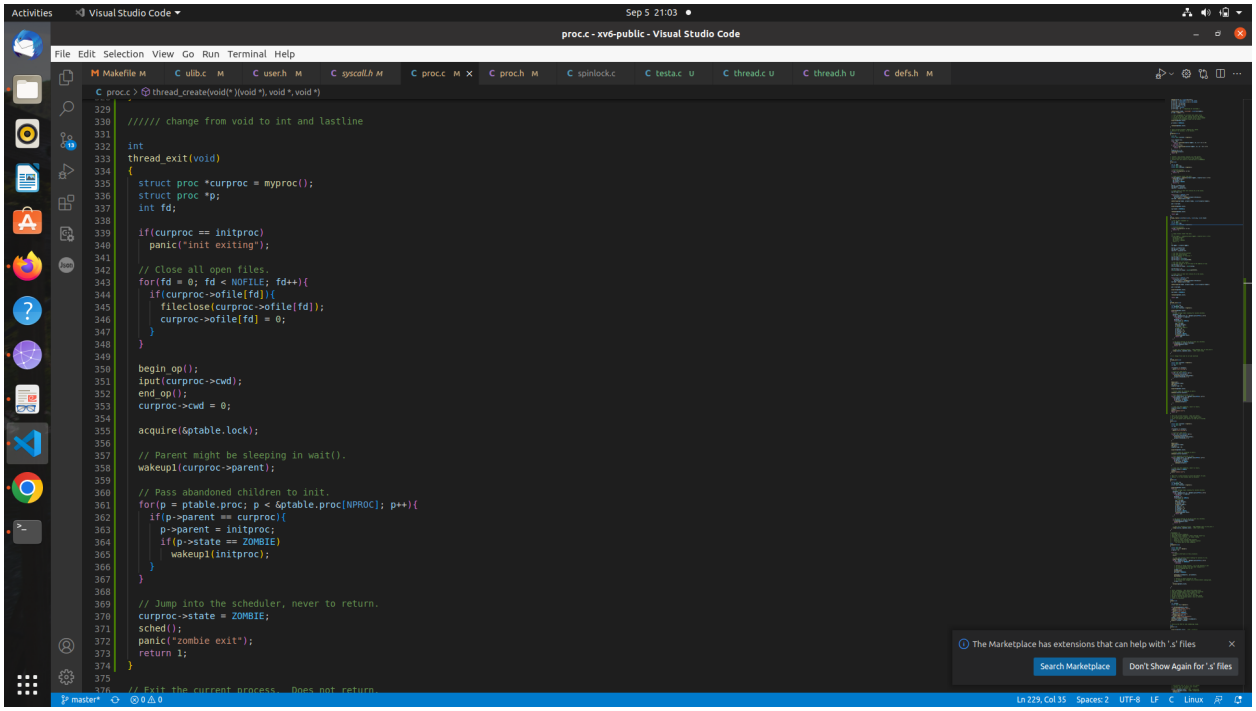
thread_join():

This function takes no argument. It has been created with the help of the wait() function which is already available in xv6 inside the same file, *proc.c*. In the function, we scan through all the threads/processes available in the ptable and then, for all the processes(threads) which are child processes(threads) of our main process, we check if the thread has completed the execution and has entered the zombie state. If the thread has completed the execution and has entered the zombie state, then we deallocate all the resources given to that thread and finally kill it and return the pid of that killed thread. If there is no thread in the zombie state i.e. all are still executing, the thread_join() function will wait for them to complete the execution. Once their execution is complete, they will enter in the zombie state so as to be killed, and therefore, enter the sleep state. At the end, when there is no thread alive, i.e. when all the threads are killed and when *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 thread killed is returned.



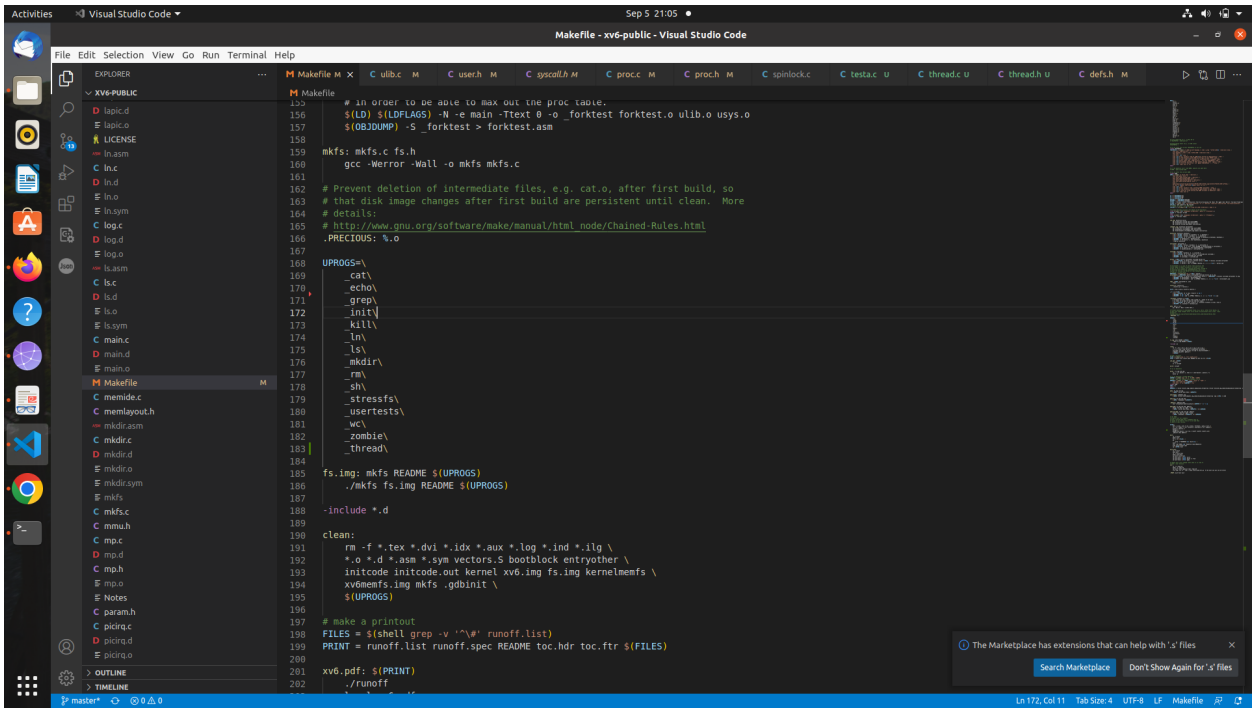
thread_exit():

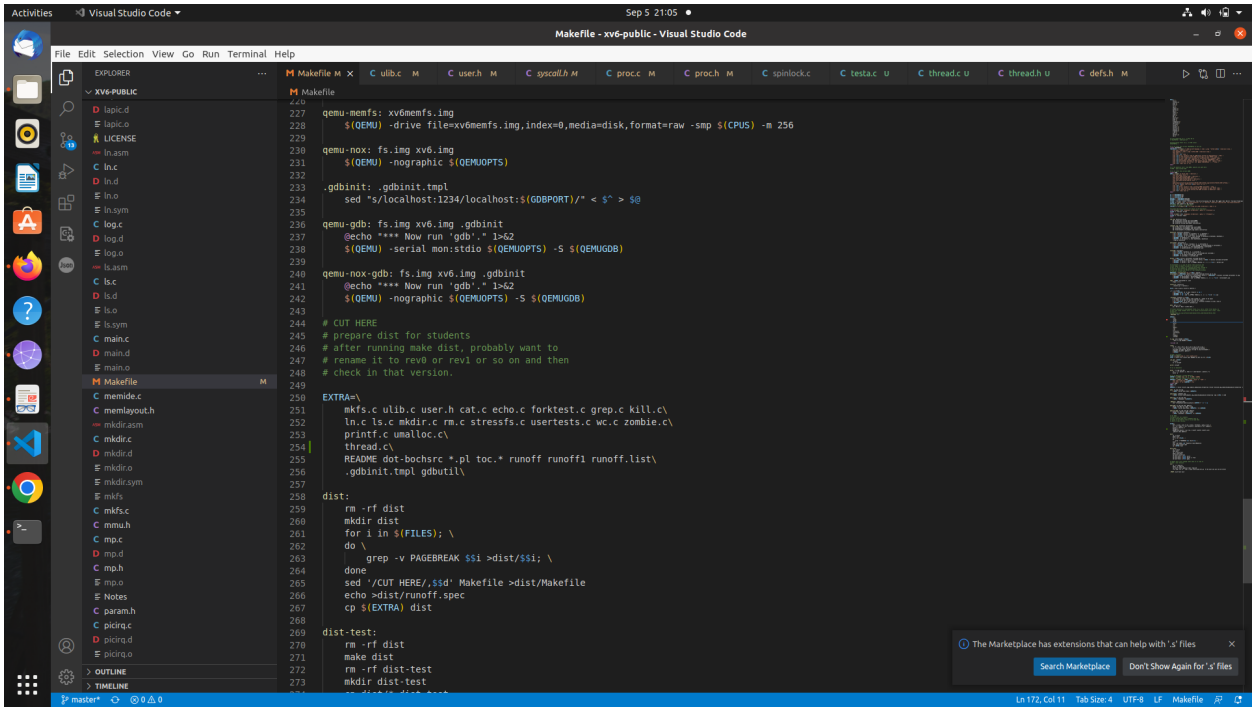
This function takes no argument. It allows a thread to terminate. It checks for all the processes in the ptable if they are completed. It has been created with the help of exit() function which is already available in xv6 inside the same file, *proc.c*. The first thing we do is make sure there are no open files associated with this thread, and if there are, we close them. Now we check if the parent is in sleep state. If the parent is indeed in sleep state, then we wake up the parent by running wakeup1 function. We then look through the ptable and if we find any threads which are associated with the parent process, and if the thread is in zombie state, then we wake up the parent using wakeup1 function call. When all children threads are completed, we then put the current process into Zombie state, and leave this function.



Makefile

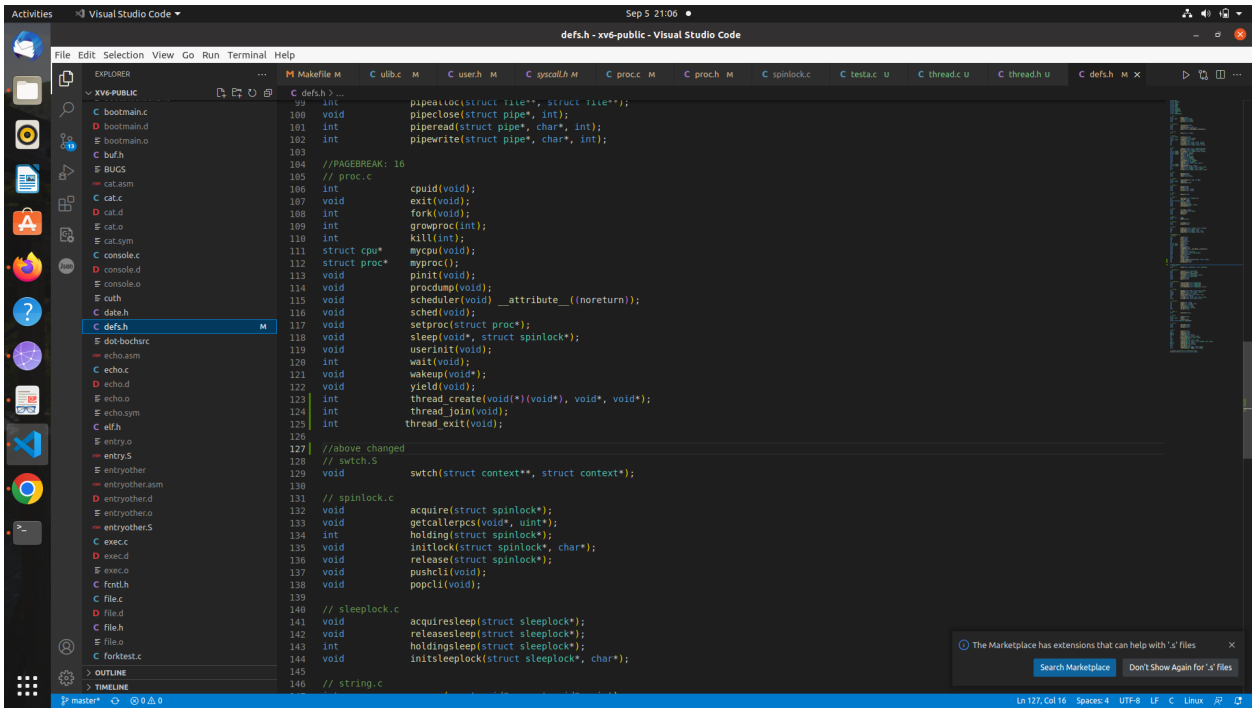
In the UPROGS we are also including ‘threads’ to show that this is also a command that the user can enter.Under extra we are mentioning that the thread.c file also has to be compiled and run.





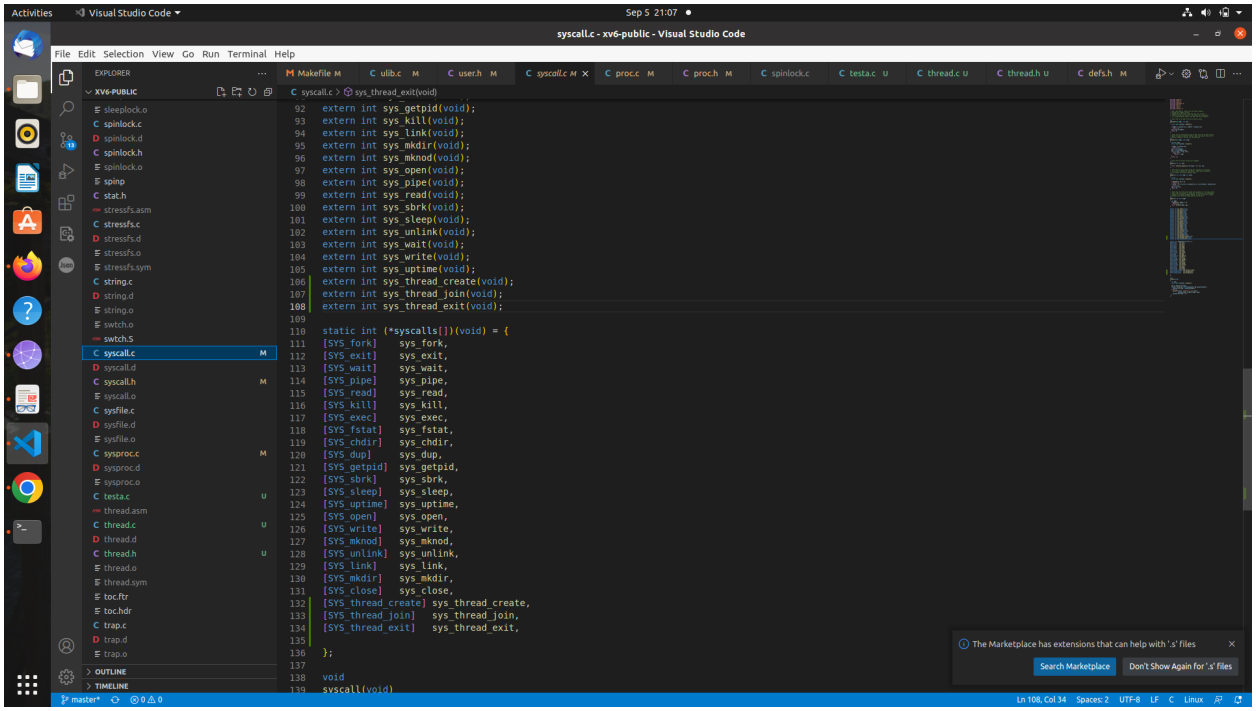
Defs.h

The declarations for thread_create, thread_join and thread_exit were created in this file.



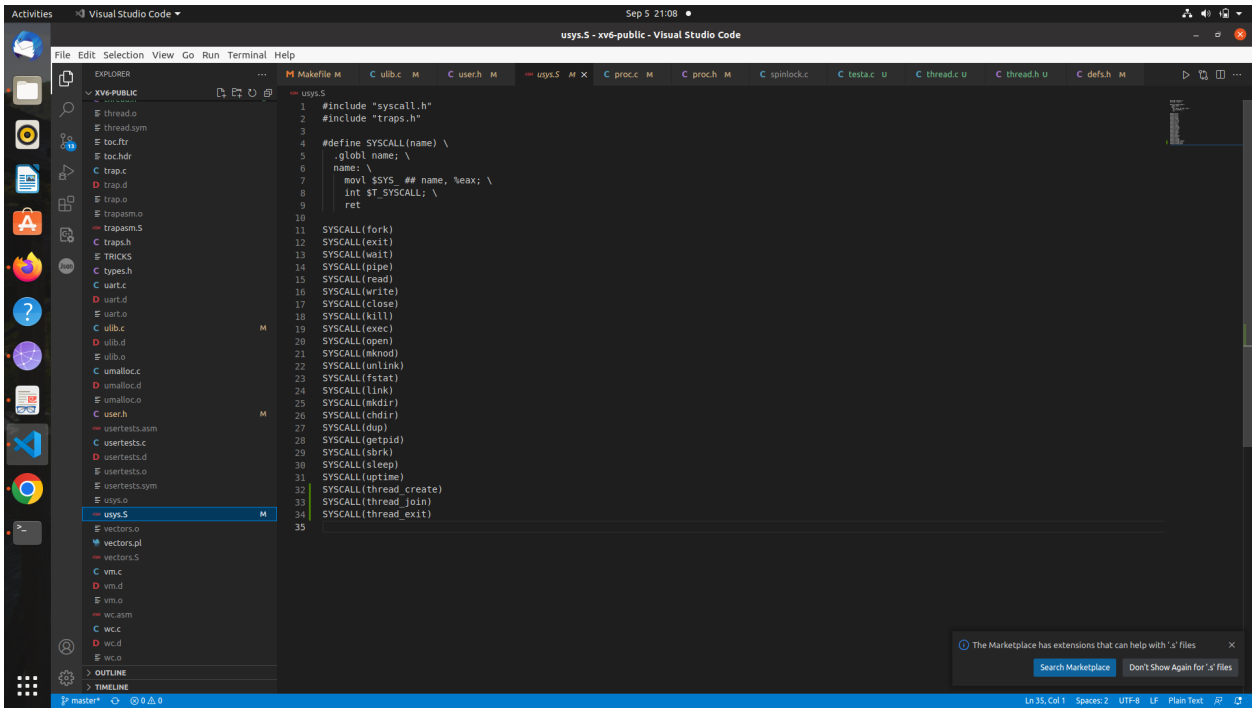
Syscall.c

Here we are declaring new functions sys_thread_create, sys_thread_join, sys_thread_exit. We will be calling the main function which we have written in proc.c



usys.S

In this file, we have added declaration of new functions to implement system calls for thread.

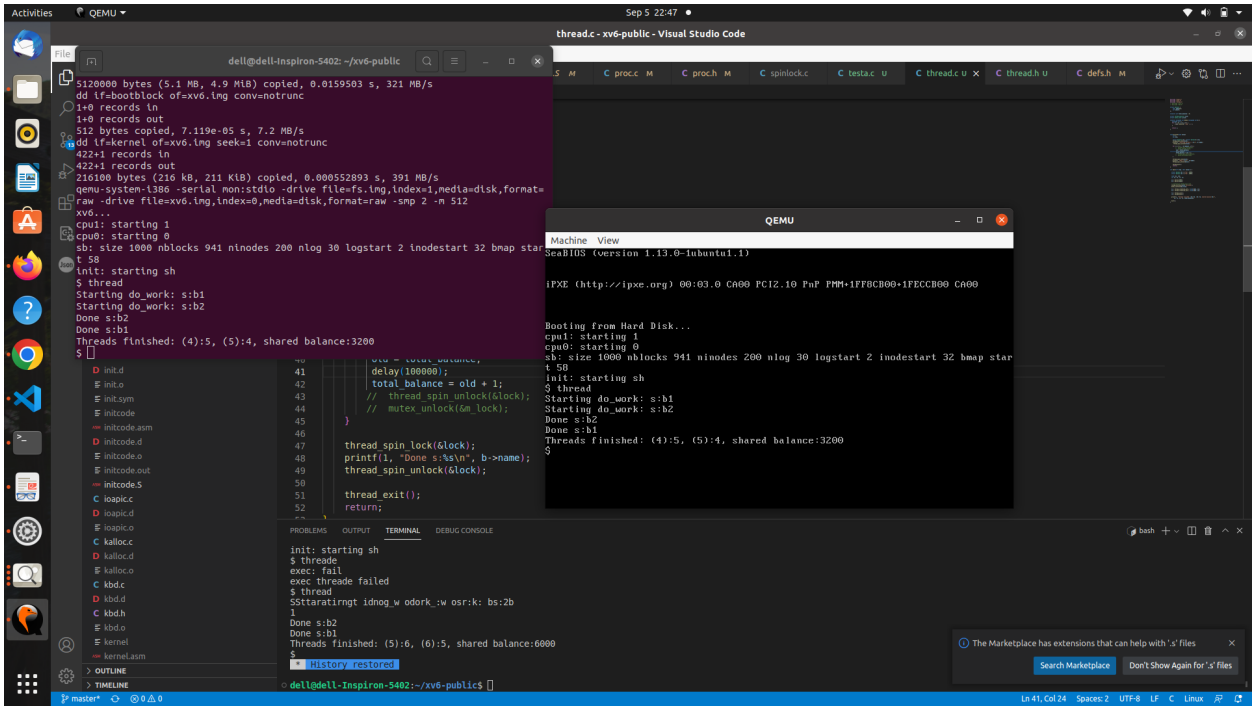


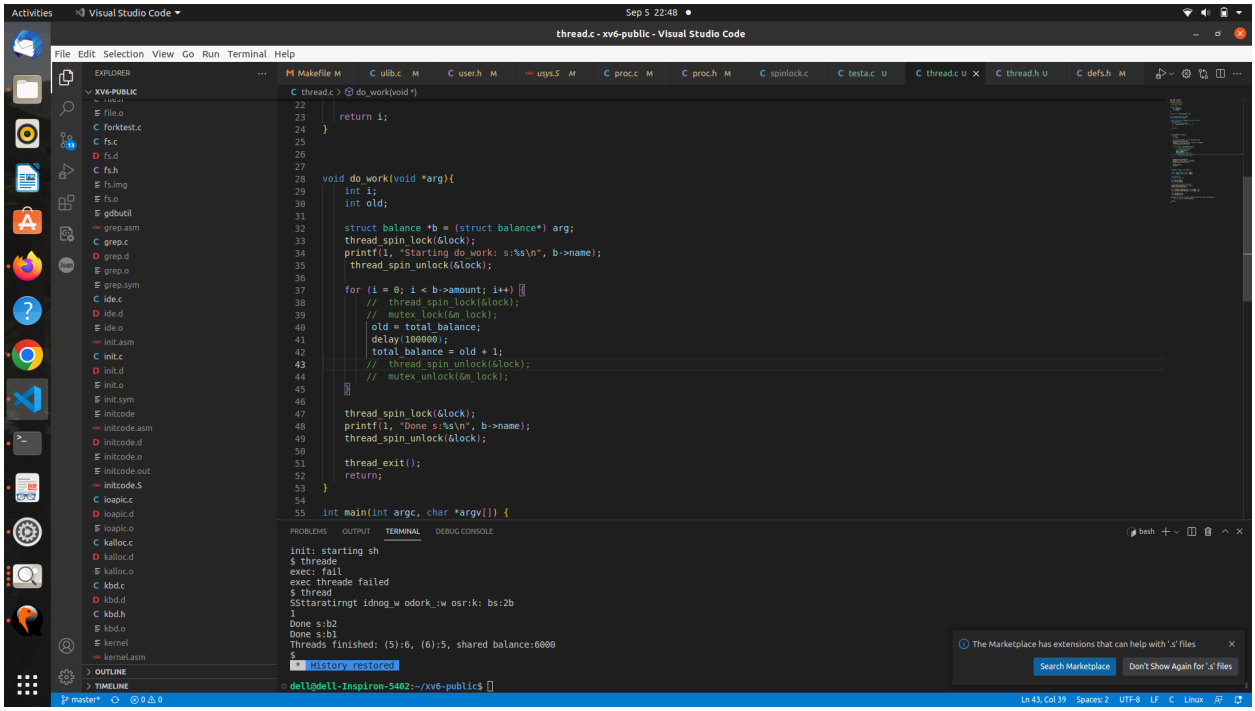
PART 2:SYNCHRONIZATION

Output without synchronization

We can clearly see that, without implementing locks, the total shared balance !=6000. Context switching keeps happening. Even if the entire critical section instructions of code doesn't complete its execution, context switching can happen .It might happen that both threads read an old value of the total_balance at the same time, and then update it at almost the same time as well. As a result the deposit the increment of the balance) from one of the threads is lost.. Locks are useful in preventing this issue.

The delay statement gives time for context switching to happen and it can be used to show that not adding locks can cause a problem.


Activities
QEMU
Sep 5 22:47
thread.c - xv6-public - Visual Studio Code
dell@dell-Inspiron-5402: ~/xv6-public
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0159503 s, 321 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 7.119e-05 s, 7.2 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
422+1 records in
422+1 records out
216100 bytes (216 kB, 211 KiB) copied, 0.000552893 s, 391 MB/s
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 2 -m 512
xv6...
cpu1: starting 1
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:b2
Done s:b1
Threads finished: (4):5, (5):4, shared balance:3200
$
QEMU
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00
Booting from Hard Disk...
cpu1: starting 1
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:b2
Done s:b1
Threads finished: (4):5, (5):4, shared balance:3200
$
PROBLEMS
OUTPUT
TERMINAL
DEBUG CONSOLE
init: starting sh
$ threade
exec: fail
exec threade failed
$ thread
SSttaratirngt idnog_w odork_:w osr:k: bs:2b
1
Done s:b2
Done s:b1
Threads finished: (5):6, (6):5, shared balance:6000
$
History restored
dell@dell-Inspiron-5402:~/xv6-public$
The Marketplace has extensions that can help with '.s' files
Search Marketplace
Don't Show Again for '.s' files
master*
Ln 41, Col 24
Spaces: 2
UTF-8
LF
C
Linux


Activities
Visual Studio Code
Sep 5 22:48
thread.c - xv6-public - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
XV6-PUBLIC
Makefile
ulib.c
user.h
usys.S
proc.c
proc.h
spinlock.c
testa.c
thread.c
thread.h
defs.h
thread.c > do_work(void *)
return i;
}
void do_work(void *arg){
int i;
int old;
struct balance *b = (struct balance*) arg;
thread_spin_lock(&lock);
printf(1, "Starting do_work: s:%s\n", b->name);
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;
// thread_spin_unlock(&lock);
// mutex_unlock(&m_lock);
}
thread_spin_lock(&lock);
printf(1, "Done s:%s\n", b->name);
thread_spin_unlock(&lock);
thread_exit();
return;
}
int main(int argc, char *argv[]) {
PROBLEMS
OUTPUT
TERMINAL
DEBUG CONSOLE
init: starting sh
$ threade
exec: fail
exec threade failed
$ thread
SSttaratirngt idnog_w odork_:w osr:k: bs:2b
1
Done s:b2
Done s:b1
Threads finished: (5):6, (6):5, shared balance:6000
$
History restored
dell@dell-Inspiron-5402:~/xv6-public$
The Marketplace has extensions that can help with '.s' files
Search Marketplace
Don't Show Again for '.s' files
master*
Ln 43, Col 39
Spaces: 2
UTF-8
LF
C
Linux

Output with spinlocks

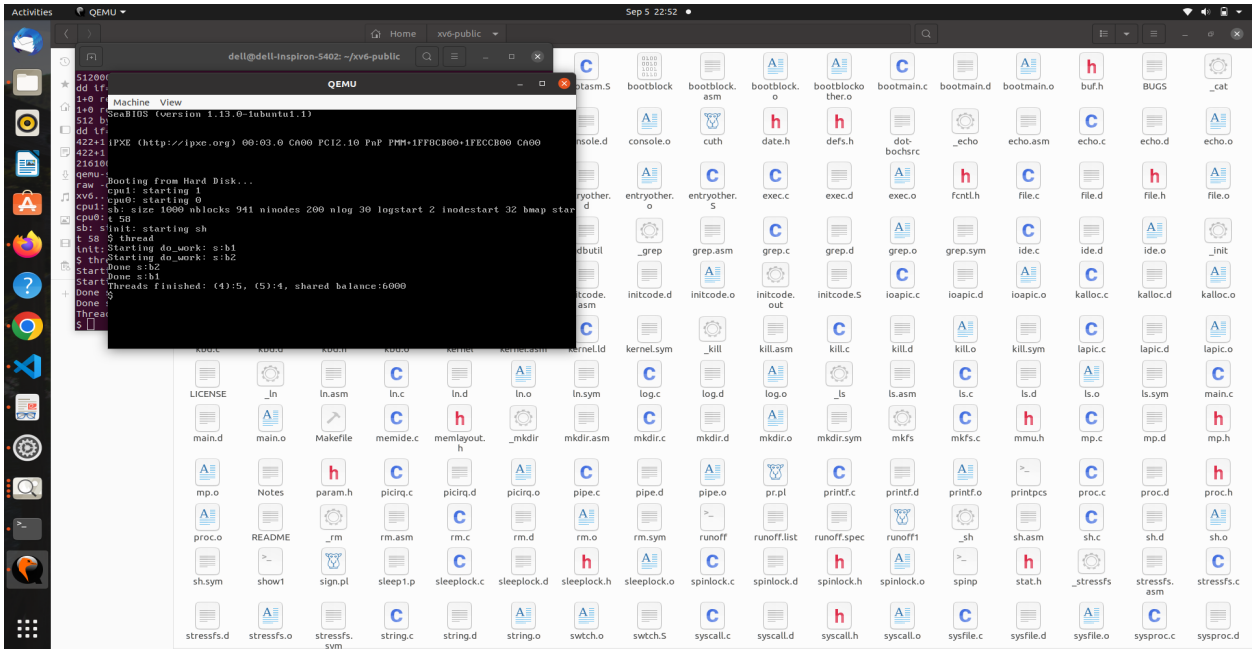
To fix this synchronization error we have to implement a spinlock that will allow us to execute the update atomically, i.e, We have to implement the thread_spin_lock() and thread_spin_unlock() functions and put them around our atomic section.

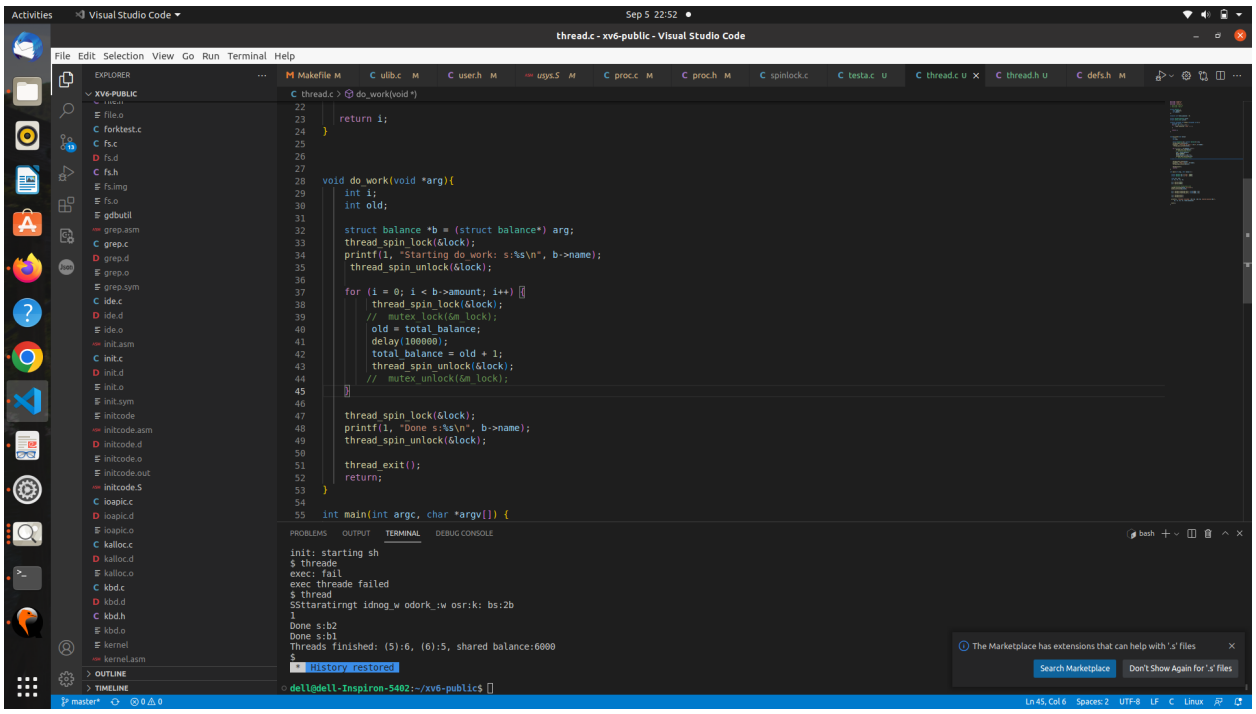
To implement spinlock we have used 3 functions:

(1)Function to Initialize the lock to the correct initial state

(2)Function to acquire a lock

(3)Function to release lock





Output with mutexes

We define a simple mutex data structure and implement three functions that:

(1)Initialize the mutex to the correct initial state

(2)Function to acquire a mutex

(3) a function to release it

Mutexes implementation is very similar to spinlocks. We use sleep(1) instead as xv6 doesn't have an explicit yield(0)