

# OS report

Assign 1

Group **C17**

Aadi Aarya Chandra - 200101002

Prakhar Pandey - 200101081

Pratham Pekamwar - 200101087

## Part 1 Steps and Observations

1. **Thread.c** created, with code in the assignment.
2. **Makefile** edited, Filename added to UPROGS and EXTRA
3. The 3 sys calls to be implemented, their **function prototypes** put in **user.h** file as follows

```
int thread_create(void (*) (void* ), void* , void* );
```

```
int thread_join(void);
```

```
int thread_exit(void);
```

So that when thread.c is executed, these functions are recognised

4. In **usys.S** added these

```
SYSCALL(thread_create)
```

```
SYSCALL(thread_join)
```

```
SYSCALL(thread_exit)
```

So that when those 3 functions are invoked, this file will say that treat them as sys calls and call SYS\_thread\_create, SYS\_thread\_join, etc

5. In **syscall.h** added these

```
#define SYS_thread_create 22
```

```
#define SYS_thread_join 23
```

```
#define SYS_thread_exit 24
```

This defines the syscall numbers for these system calls

6. In **syscall.c** add these

```
extern int sys_thread_create(void);
```

```
extern int sys_thread_join(void);
```

```
extern int sys_thread_exit(void);
```

And in the syscalls array, the below lines

```
[SYS_thread_create] sys_thread_create,
```

```
[SYS_thread_join] sys_thread_join,
```

```
[SYS_thread_exit] sys_thread_exit,
```

7. Then the 3 syscall functions are implemented in sysproc.c and proc.c.

**sysproc.c** just has **wrapper functions**

**proc.c** has **actual implementation.**

#### 8. **Changes in proc.c -**

- In wait() - if(p->parent != curproc || curproc->pgdir == p->pgdir)  
continue;

(this is added because process wait call should only check for terminated child processes and not terminated child threads)

- int thread\_create(void) - in this call, we first use argptr to get the various arguments (fcn, args, stack) from user call of the same name. Then rest of the implementation is same as fork except for these changes:

np->pgdir = curproc->pgdir; // so that new 'process' and current process use same page directory, hence same address space, hence they are essentially threads

(this is in place of if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0{

```
kfree(np->kstack);  
np->kstack = 0;  
np->state = UNUSED;  
return -1;
```

```
})
```

(Stack here grows downwards)

```
void* args1 = stack + PGSIZE - 1*sizeof(void*);
```

```
// pushing args onto the stack
```

```
*(void**)args1 = args;
```

```
void* ret_addr = stack + PGSIZE - 2*sizeof(void*);
```

```
// pushing fake return address onto stack
```

```
*(uint*)ret_addr = 0xFFFFFFFF;
```

Tf is trap frame, which stores many user context related info when os is in kernel mode

```
np->tf->esp = (uint) stack+PGSIZE-2*sizeof(void*);
```

```
// stack pointer set to top of stack
```

```
np->tf->ebp = np->tf->esp;
```

```
// stack base pointer set to same location as stack ptr
```

```
np->tf->eip = (uint) fnptr;      // eip is basically tf storing the PC value to go to in the // user mode of this thread. We set it to point to the function we want to run on this thread.
```

- int thread\_join(void) - same as wait() but

```
if(p->parent != curproc || curproc->pgdir != p->pgdir)
```

```
    Continue;
```

Because here need to check only for child threads, ie those which have same

address space (hence same page dir)

Also, actual deallocation of page table of process (hence the whole address space and other data of process) happens in wait, which shouldn't happen in thread\_join. In thread\_join, only that thread's user stack should be deallocated. For this we remove the line freevm(p->pgdir); from for loop.

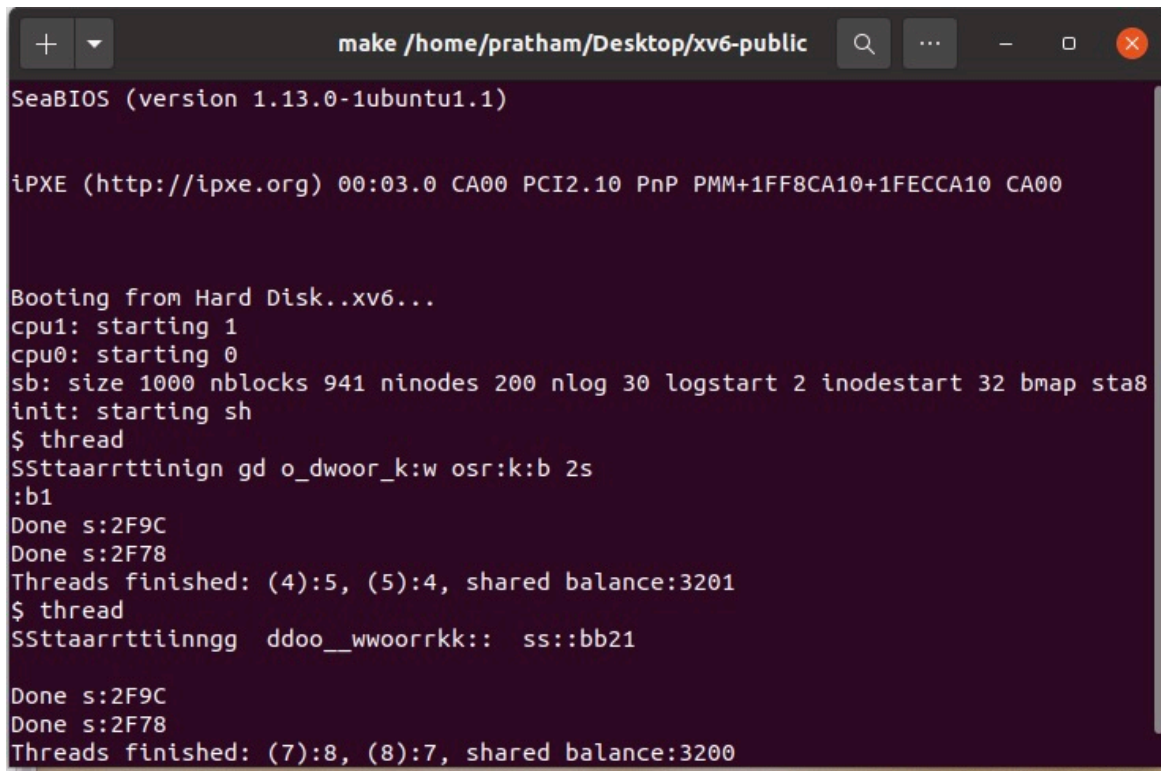
- `int thread_exit(void)` - same as `exit()` but pid found and return pid added at end, since return type is `int`, not `void`.

Also open files are not closed in threads because other threads use it so the following lines were removed from `exit` in `thread_exit` –

```
// Close all open files.
for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
        fclose(curproc->ofile[fd]);
        curproc->ofile[fd] = 0;
    }
}
begin_op();
input(curproc->cwd);
end_op();
curproc->cwd = 0;
```

```
77 void do_work(void *arg)
78 {
79     int i;
80     int old;
81     struct balance *b = (struct balance*) arg;
82     printf(1, "Starting do_work: s:%s\n", b->name);
83     for (i = 0; i < b->amount; i++)
84     {
85         // thread_spin_lock(&lock);
86         // thread_mutex_lock(&mLock);
87         old = total_balance;
88         delay(100000);
89         total_balance = old + 1;
90         // thread_spin_unlock(&lock);
91         // thread_mutex_unlock(&mLock);
92     }
93     printf(1, "Done s:%x\n", b->name);
94     thread_exit();
95 }
```

When there are no locks in Place, the output is



```
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00

Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ thread
SSttaarrttinign gd o_dwoor_k:w osr:k:b 2s
:b1
Done s:2F9C
Done s:2F78
Threads finished: (4):5, (5):4, shared balance:3201
$ thread
SSttaarrttiinnngg ddoo__wwoorrk:: ss::bb21
Done s:2F9C
Done s:2F78
Threads finished: (7):8, (8):7, shared balance:3200
```

Observations:

A. The print function's output is jumbled because of context switching between the two threads.

B. Total balance doesn't match 6000 which is sum of individual balances of the threads because of context switching between the critical section.

Solution : Synchronisation needs to be Implemented.

## Part 2 Steps and Observations

In **thread.c** file the following things were added

### Spinlock Implementation

Changes made in thread.c – added struct thread\_spinlock, with only uint locked (others in kernel implementation were for debugging, so removed), thread\_spin\_lock(), thread\_spin\_unlock(). The implementation of thread\_spinlock is very simplified compared to actual spinlock in xv6, as many of the functions like cli, sti, etc couldn't be used in thread.c

```
11 struct thread_spinlock {
12     uint locked;
13 };
14
15 struct thread_spinlock lock;
16
17 void thread_spin_init(struct thread_spinlock *lk){
18     lk->locked = 0;
19 };
20 void thread_spin_lock(struct thread_spinlock *lk){
21     // The xchg is atomic.
22     while(xchg(&lk->locked, 1) != 0)
23         ;
24     // Tell the C compiler and the processor to not move loads or stores
25     // past this point, to ensure that the critical section's memory
26     // references happen after the lock is acquired.
27     __sync_synchronize();
28 };
29 void thread_spin_unlock(struct thread_spinlock *lk){
30     // Tell the C compiler and the processor to not move loads or stores
31     // past this point, to ensure that all the stores in the critical
32     // section are visible to other cores before the lock is released.
33     // Both the C compiler and the hardware may re-order loads and
34     // stores; __sync_synchronize() tells them both not to.
35     __sync_synchronize();
36     asm volatile("movl $0, %0" : "+m" (lk->locked) : );
37 };
```

The “\_\_sync\_synchronize();” is used in the lock and unlock function implementation of the spinlock and the mutexlock.

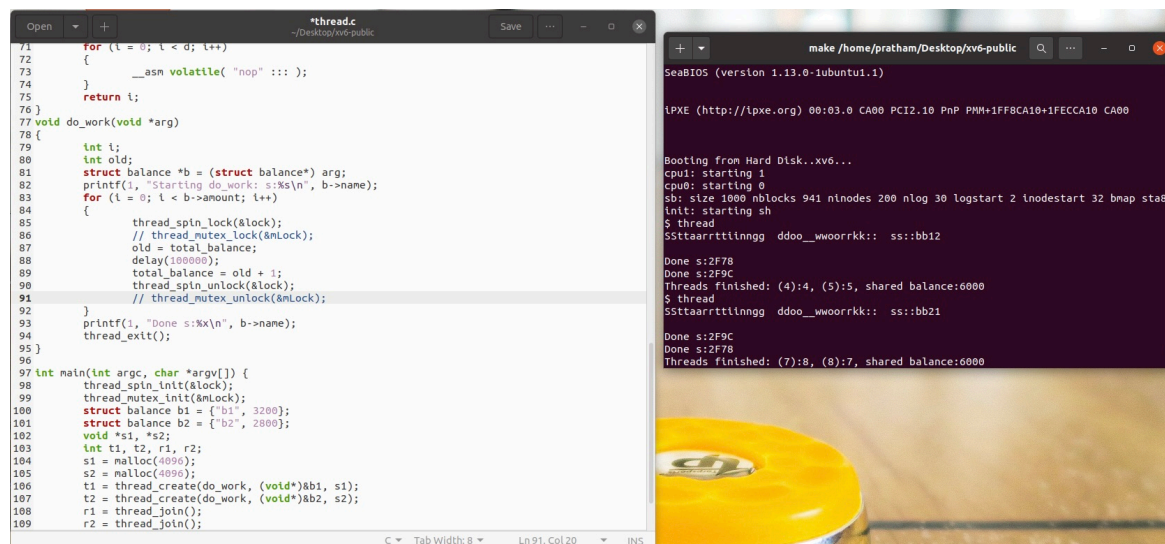
It tells the c compiler and the processor to not allow the load and store instructions before this to be moved after this point because of compiler rescheduling.

This allows us to keep the variable updates before the critical section to complete before any access from the critical section and the variable updates in the critical section to be done before the ending of critical section

so that any access to the variables after the critical section would have the correct variables.

The spinlock is initialised at the beginning of the main function.

Included x86.h header in thread.c for the xchg function



```
71 for (i = 0; i < d; i++)
72 {
73     __asm volatile( "nop" ::: );
74 }
75 return i;
76 }
77 void do_work(void *arg)
78 {
79     int i;
80     int old;
81     struct balance *b = (struct balance*) arg;
82     printf(i, "Starting do work: %s\n", b->name);
83     for (i = 0; i < b->amount; i++)
84     {
85         thread_spin_lock(&lock);
86         // thread_mutex_lock(&lock);
87         old = total_balance;
88         delay(100000);
89         total_balance = old + i;
90         thread_spin_unlock(&lock);
91         // thread_mutex_unlock(&lock);
92     }
93     printf(i, "Done %s\n", b->name);
94     thread_exit();
95 }
96
97 int main(int argc, char *argv[]) {
98     thread_spin_init(&lock);
99     thread_mutex_init(&lock);
100     struct balance b1 = {"b1", 3200};
101     struct balance b2 = {"b2", 2800};
102     void *s1, *s2;
103     int t1, t2, r1, r2;
104     s1 = malloc(4096);
105     s2 = malloc(4096);
106     t1 = thread_create(do_work, (void*)&b1, s1);
107     t2 = thread_create(do_work, (void*)&b2, s2);
108     r1 = thread_join();
109     r2 = thread_join();
110 }
```

```
make /home/pratham/Desktop/xv6-public
SeaBIOS (version 1.13.0-1ubuntu1.1)

lpxe (http://lpxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00

Booting from Hard Disk..xv6...
cpu0: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta0
init: starting sh
$ thread
SSttaarrttllnngg ddoo_wwoorrk:: ss::bb12
Done s:2F78
Done s:2F9C
Threads finished: (4):4, (5):5, shared balance:6000
$ thread
SSttaarrttllnngg ddoo_wwoorrk:: ss::bb21
Done s:2F78
Done s:2F9C
Threads finished: (7):8, (8):7, shared balance:6000
```

## Observations :

A. The print function's output is still jumbled as locks are just used around the critical section

B. Because of spinlock synchronisation in critical section, Total balance becomes 6000 which is sum of individual balances of the threads.



## Mutex Implementation

If Delay is more, computation becomes slow (because spinlock runs in loop until delay is crossed if the resource is shared). Hence, mutex is used.

```
39 struct thread_mutex {
40     uint mutexLock;
41 };
42
43 struct thread_mutex mLock;
44
45 void thread_mutex_init(struct thread_mutex *lk){
46     lk->mutexLock = 0;
47 };
48 void thread_mutex_lock(struct thread_mutex *lk){
49     // The xchg is atomic.
50     while(xchg(&lk->mutexLock, 1) != 0)
51         sleep(1);
52     // Tell the C compiler and the processor to not move loads or stores
53     // past this point, to ensure that the critical section's memory
54     // references happen after the lock is acquired.
55     __sync_synchronize();
56 };
57 void thread_mutex_unlock(struct thread_mutex *lk){
58     // Tell the C compiler and the processor to not move loads or stores
59     // past this point, to ensure that all the stores in the critical
60     // section are visible to other cores before the lock is released.
61     // Both the C compiler and the hardware may re-order loads and
62     // stores; __sync_synchronize() tells them both not to.
63     __sync_synchronize();
64     asm volatile("movl $0, %0" : "+m" (lk->mutexLock) : );
65 };
66
```

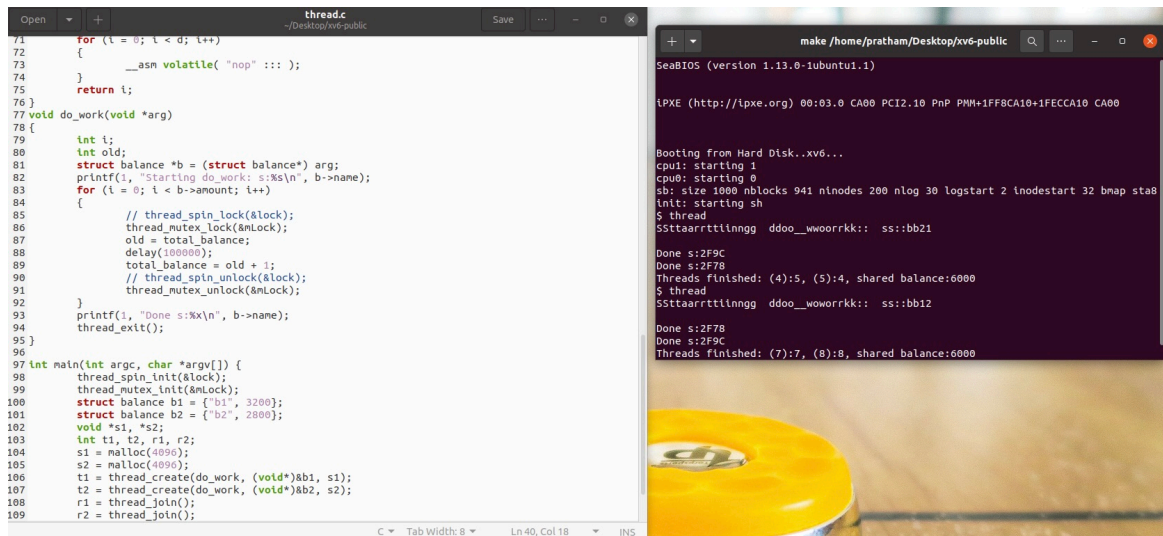
Mutex implementation is very similar to spinlock. In while loop in lock function of mutex we just added sleep(1);

The “\_\_sync\_synchronize();” is used in the lock and unlock function implementation of the spinlock and the mutexlock.

It tells the c compiler and the processor to not allow the load and store instructions before this to be moved after this point because of compiler rescheduling.

This allows us to keep the variable updates before the critical section to complete before any access from the critical section and the variable updates in the critical section to be done before the ending of critical section so that any access to the variables after the critical section would have the correct variables.

The mutexlock is initialised at the beginning of the main function.



```
71 for (i = 0; i < d; i++)
72 {
73     __asm volatile( "nop" ::: );
74 }
75 return i;
76 }
77 void do_work(void *arg)
78 {
79     int i;
80     int old;
81     struct balance *b = (struct balance*) arg;
82     printf(1, "Starting do_work: s:%s\n", b->name);
83     for (i = 0; i < b->amount; i++)
84     {
85         // thread_spin_lock(&lock);
86         thread_mutex_lock(&lock);
87         old = total_balance;
88         delay(1000000);
89         total_balance = old + 1;
90         // thread_spin_unlock(&lock);
91         thread_mutex_unlock(&lock);
92     }
93     printf(1, "Done s:%s\n", b->name);
94     thread_exit();
95 }
96
97 int main(int argc, char *argv[]) {
98     thread_spin_init(&lock);
99     thread_mutex_init(&lock);
100     struct balance b1 = {"b1", 3200};
101     struct balance b2 = {"b2", 2800};
102     void *s1, *s2;
103     int t1, t2, r1, r2;
104     s1 = malloc(4096);
105     s2 = malloc(4096);
106     t1 = thread_create(do_work, (void*)&b1, s1);
107     t2 = thread_create(do_work, (void*)&b2, s2);
108     r1 = thread_join();
109     r2 = thread_join();
110 }
```

```
SeaBIOS (version 1.13.0-1ubuntu1.1)
iPXE (http://ipxe.org) 00:03:0 CA00 PCI2.10 PnP PMW+1FF8CA10+1FECCA10 CA00

Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 lnodestart 32 bnmap sta8
init: starting sh
$ thread
Ssttaarrttiinnngg ddoo__wwoorrrkk:: ss::bb21
Done s:2F9C
Done s:2F78
Threads finished: (4):5, (5):4, shared balance:6000
$ thread
Ssttaarrttiinnngg ddoo__wwoorrrkk:: ss::bb12
Done s:2F78
Done s:2F9C
Threads finished: (7):7, (8):8, shared balance:6000
```

## Observations :

- A. The print function's output is still jumbled as locks are just used around the critical section
- B. Because of mutexlock synchronisation in critical section, Total balance becomes 6000 which is sum of individual balances of the threads.

## Other Observations

- Four out of the six threads execution and completion are out of order, since the first process has total balance 3200 and the second has total balance 2800, the first process has higher probability of finishing later than the second process.

In `do_work()`, “t1:r1, t2:r2” is printed where t1 and t2 are the PID of the two created threads, r1 is the PID of the thread which finishes first and r2 is the PID of the remaining thread.

- Observation – when no. of cpus was changed in `params.h` we saw that in when it was set to 1 (uniprocessor system), mutex gave the result very quickly even for large values of delay, while spinlock took some time. With more processors (no. of cpus set to 16), mutex also got slower, while spinlock became relatively faster.