CS344

Assignment-1

Aditya Patidar 200101009

Aman Soni 200101012

Kartik Malasiya 200101053

Part 1- ***Kernel Threads***

The following three system calls have been created in this part:

1) thread_create()
2) thread_join()
3) thread_exit()

thread_create() -- This call create a new kernel thread which shares the address space with the calling space.

thread_join() -- This call waits for a child thread that shares the address space with the calling process. It will return the PID of the child or -1 if there is none.

thread_exit() -- This call allows a thread to terminate.

The following xv6 OS files need to be changed in order to implement the above functions.

- ➔ proc.c
- ➔ syscall.c
- ➔ syscall.h
- ➔ defs.h
- ➔ sysproc.c
- ➔ user.h
- ➔ usys.S

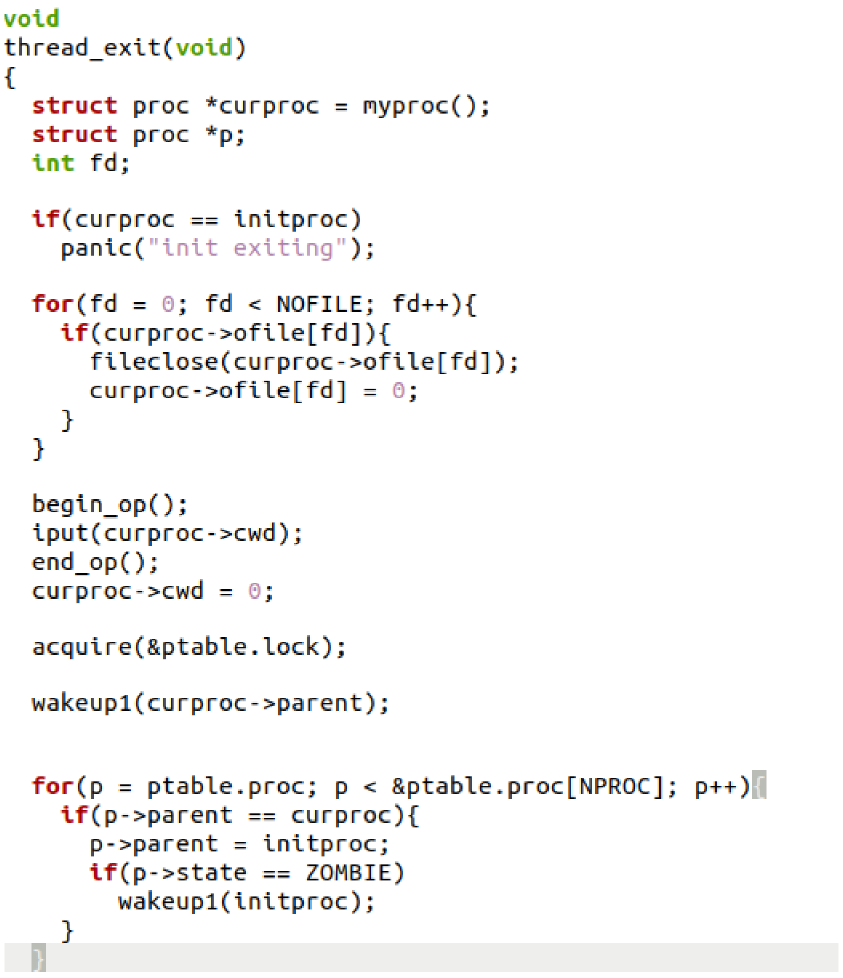
Now we will change the above files individually.

1. ***proc.c***-

The functions mentioned above(thread_create(), thread_join(), thread_exit()) are added in this file.

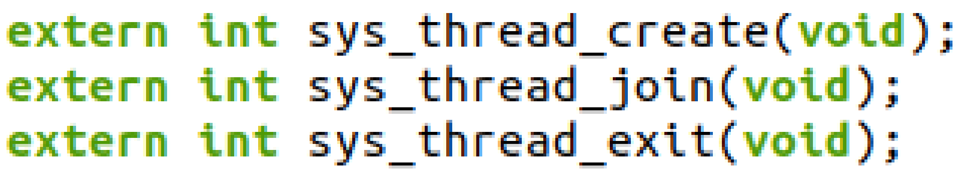
```c
int thread_create(void(*fcn)(void*),void* arg,void* stack){
        int i,pid;
        struct proc *np;
        struct proc *curproc = myproc();

        if((np= allocproc()) == 0){
                return -1;
        }
        np->pgdir = curproc->pgdir;
        np->sz = curproc->sz;
        np->parent = curproc;
        *np->tf = *curproc->tf;

        np->tf->eax = 0;
        np->tf->eip = (uint)fcn;
        np->tf->esp = (uint)stack+4069;

        np->tf->esp -= 4;
        *(uint*)(np->tf->esp) = (uint)(arg);

        np->tf->esp -= 4;
        *(uint*)(np->tf->esp) = (uint)0xFFFFFFF;
        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;

}
```

```c
int
thread_join(void)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;
      }
    }

    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

    sleep(curproc, &ptable.lock);
  }
}
```


```
void
thread_exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  acquire(&ptable.lock);

  wakeup1(curproc->parent);


  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }
```

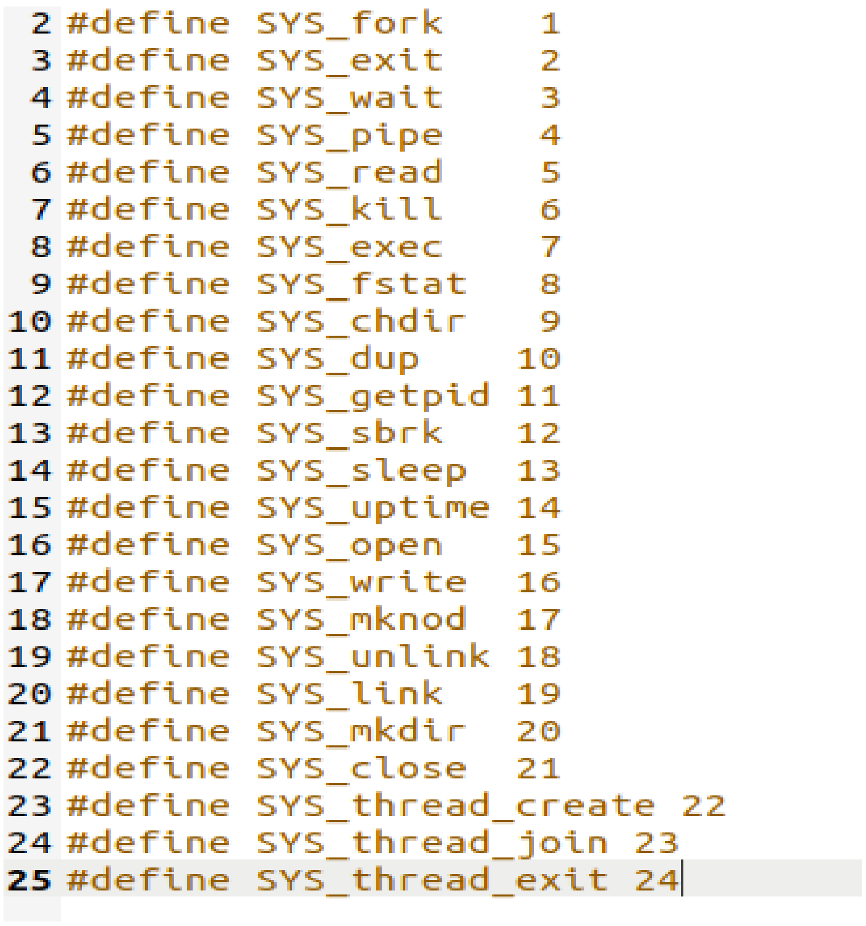
2. *syscall.c-*

The following lines of the code as shown in the screenshot below have been added to the file to make the system understand that these are also the system calls.


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

3. *syscall.h*

The file syscall.h has been edited to add three new system call macros to the list of already existing system call macros with their respective IDs. The screenshot of the same is given below.



```
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_thread_create 22
#define SYS_thread_join 23
#define SYS_thread_exit 24
```

4. defs.h

As we know file defs.h contains the prototype for all the system calls and we have created three new system calls so we will have to add prototype for these system calls in defs.h. The screenshot for the same is attached below.

```c
// proc.c
int             cpuid(void);
void            exit(void);
int             fork(void);
int             growproc(int);
int             kill(int);
struct cpu*     mycpu(void);
struct proc*    myproc();
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            setproc(struct proc*);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(void);
void            wakeup(void*);
void            yield(void);
//user defined below
int            thread_create(void(*)(void*),void*,void*);
int            thread_join(void);
void           thread_exit(void);
```

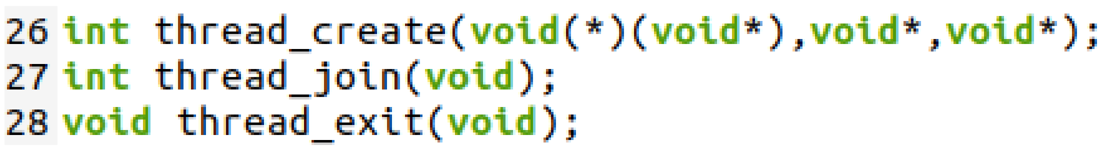
5. sysproc.c

These lines of code are the function corresponding to the system call made by us that will call the actual function of thread_exit(), thread_join() and thread_exit() when the system calls one of these functions.

```c
int
sys_thread_create(void){
        int fcn;
        char* arg;
        char* stack;
        if(argint(0,&fcn)<0)
                return -1;
        if(argstr(1,&arg) <0)
                return -1;
        if(argstr(2,&stack)<0)
                return -1;
        return thread_create((void(*)(void*))fcn,arg,stack);
}

int sys_thread_exit(void){
        thread_exit();
        return 0;
}

int sys_thread_join(void){
        return thread_join();
}
```

6. user.h

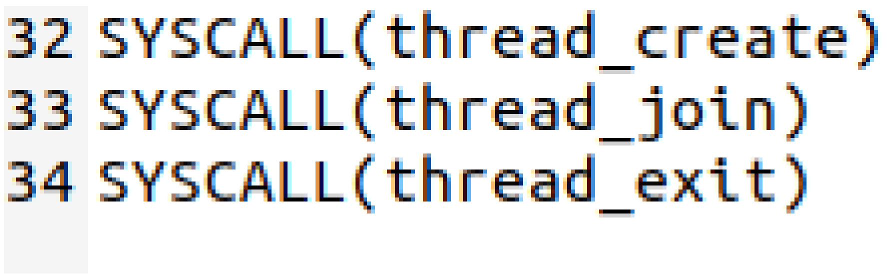
This file also contains prototypes for the system calls. Given below is the screenshot of the added lines.



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

7. usys.S

This file contains code corresponding to binding the function with the system.



```
SYSCALL(thread_create)
SYSCALL(thread_join)
SYSCALL(thread_exit)
```

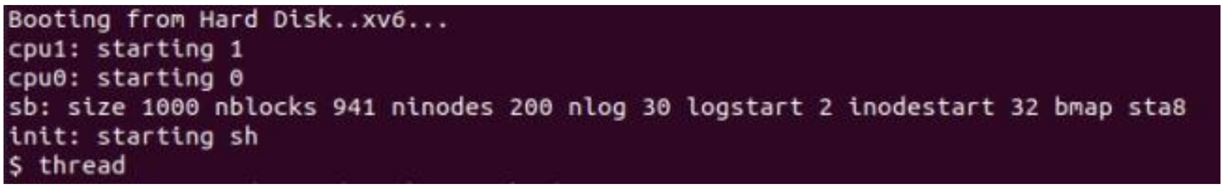
Part 2: Synchronization

By creating spinlocks and mutexes, we must focus on the synchronisation component in this section.

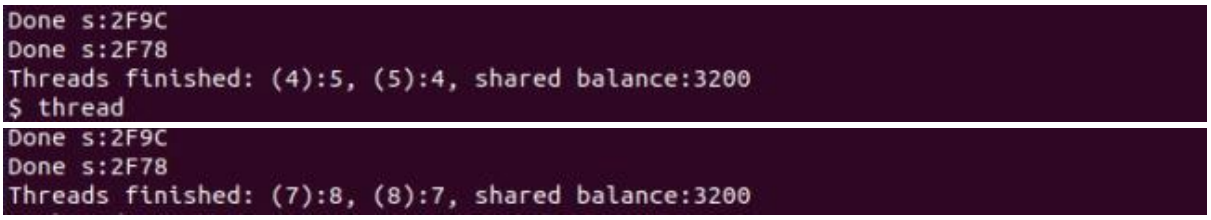
Two balance structures, b1 and b2, with initial values of 3200 and 2800 each, have been built. And by using the do work function in both of them, we created two threads to update the shared balance variable. However, if they don't execute with synchronisation, the value of the shared balance doesn't come out as expected (6000), so in order to fix this problem, we had to create spinlocks. However, the problem with spinlocks is that they can't operate properly on a single processor system or when the system is under a lot of load because all the spinlocks will spin endless So, to solve this problem we are going to use mutexes in place of spin locks. The results of five time calls made without synchronisation.

To solve this we will use mutex locks.

Without using mutex locks-

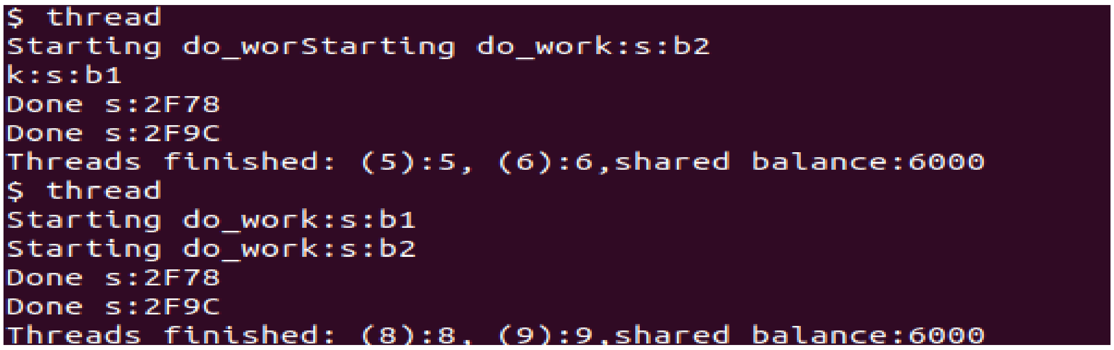


```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ thread
```



```
Done s:2F9C
Done s:2F78
Threads finished: (4):5, (5):4, shared balance:3200
$ thread
Done s:2F9C
Done s:2F78
Threads finished: (7):8, (8):7, shared balance:3200
```

with using mutex locks-

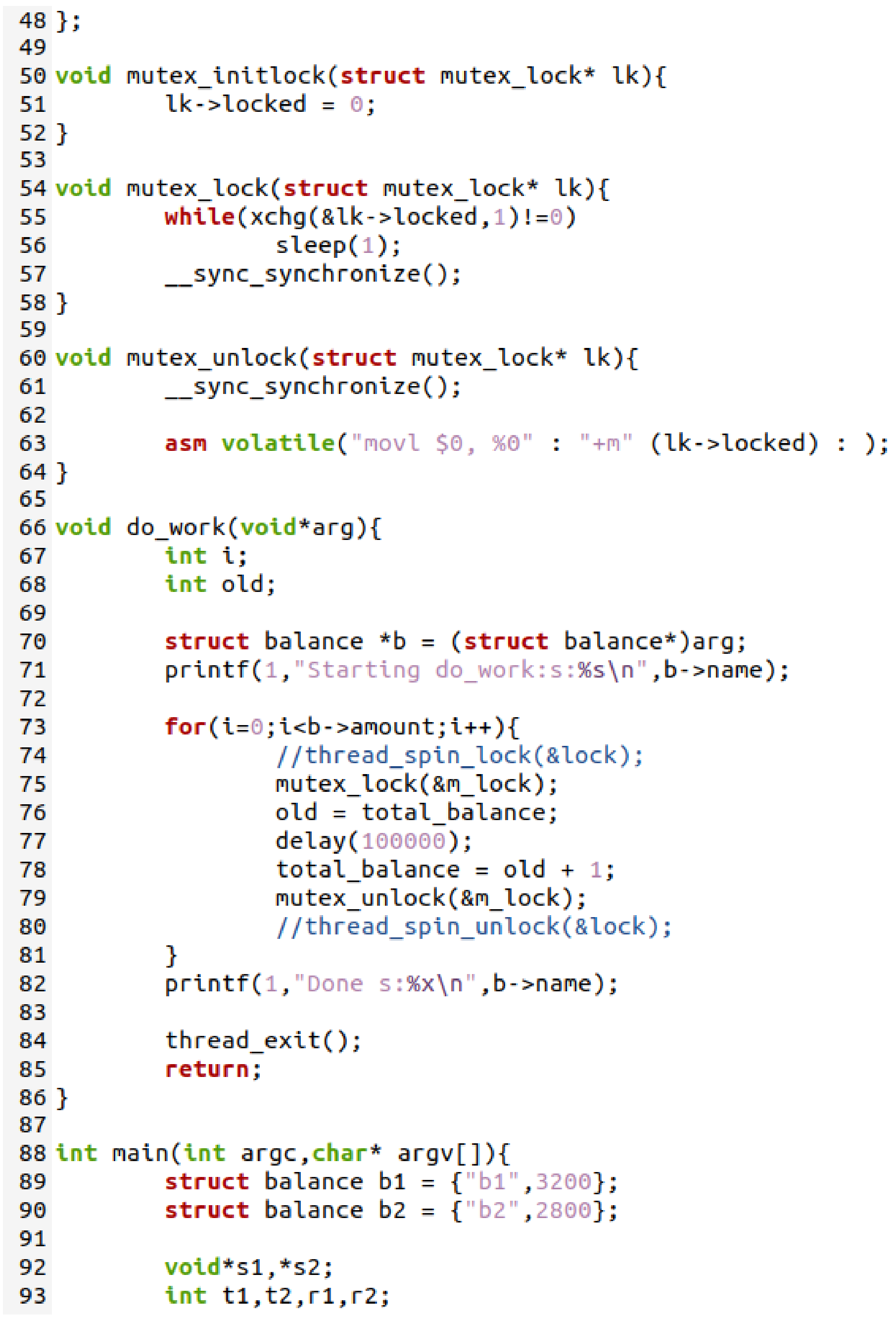


```
$ thread
Starting do_worStarting do_work:s:b2
k:s:b1
Done s:2F78
Done s:2F9C
Threads finished: (5):5, (6):6,shared balance:6000
$ thread
Starting do_work:s:b1
Starting do_work:s:b2
Done s:2F78
Done s:2F9C
Threads finished: (8):8, (9):9,shared balance:6000
```

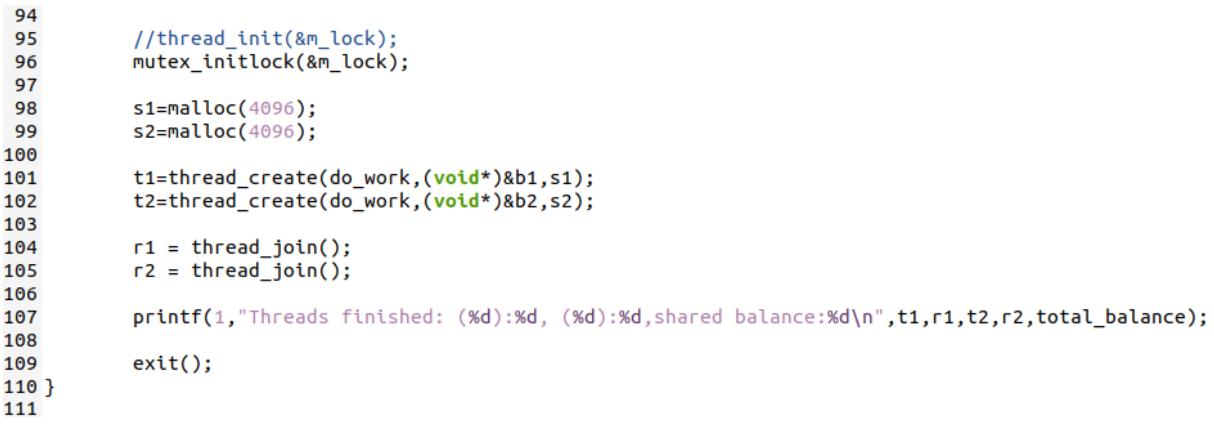
Thread.c file which is created in xv6 is given below-

Thread.c

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "x86.h"

struct balance{
        char name[32];
        int amount;
};

volatile int total_balance = 0;

struct thread_spinlock lock;
struct mutex_lock m_lock;


volatile unsigned int delay(unsigned int d){
        unsigned int i;
        for(i=0;i<d;i++){
                __asm volatile("nop":::);
        }
        return i;
}

struct thread_spinlock{
        uint locked;

        char* name;
};

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

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

void thread_spin_unlock(struct thread_spinlock* lk){
        __sync_synchronize();
        asm volatile("movl $0, %0" : "+m" (lk->locked) :);
}

struct mutex_lock{
        uint locked;
```



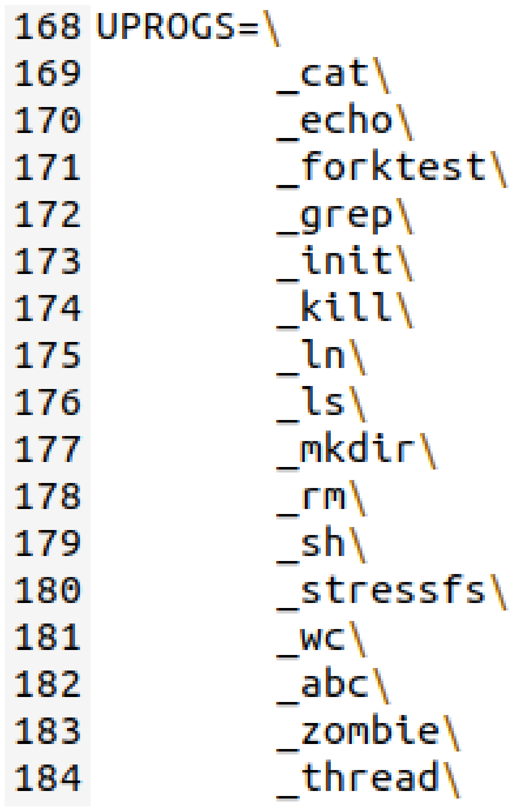
```c
};

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

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

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

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

void do_work(void*arg){
        int i;
        int old;

        struct balance *b = (struct balance*)arg;
        printf(1,"Starting do_work:s:%s\n",b->name);

        for(i=0;i<b->amount;i++){
                //thread_spin_lock(&lock);
                mutex_lock(&m_lock);
                old = total_balance;
                delay(100000);
                total_balance = old + 1;
                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[]){
        struct balance b1 = {"b1",3200};
        struct balance b2 = {"b2",2800};

        void*s1,*s2;
        int t1,t2,r1,r2;
```



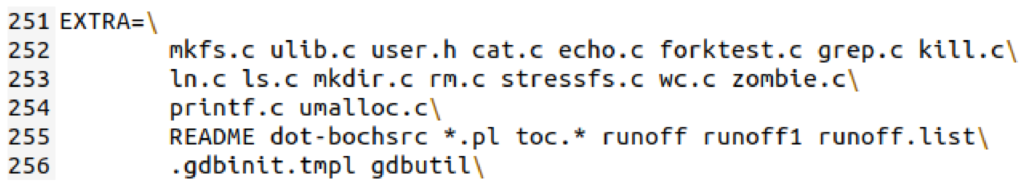
```c

        //thread_init(&m_lock);
        mutex_initlock(&m_lock);

        s1=malloc(4096);
        s2=malloc(4096);

        t1=thread_create(do_work,(void*)&b1,s1);
        t2=thread_create(do_work,(void*)&b2,s2);

        r1 = thread_join();
        r2 = thread_join();

        printf(1,"Threads finished: (%d):%d, (%d):%d,shared balance:%d\n",t1,r1,t2,r2,total_balance);

        exit();
}

```

The call relating to the thread.c has been added to include this in the list of xv6 commands in this makefile.

This line of code has been added to UPROGS.



```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_wc\
	_abc\
	_zombie\
	_thread\
```



```
EXTRA=\
	mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
	ln.c ls.c mkdir.c rm.c stressfs.c wc.c zombie.c\
	printf.c umalloc.c\
	README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
	.gdbinit.tmpl gdbutil\
```

How to run the test case in steps

Enter the xv6-public directory first, then issue the make clean command. make command, followed by make qemu-nox command. After that, execute the test case thread.

Remark: When we use mutexes, we achieve the intended results, but when we don't, we get the incorrect results.