CS344
Lab Assignment

Group No-M23

Group members -

1-Arti Sahu (200123011)
2-Harshul Gupta(200123023)
3-SUNNY Narzary(200123062)

Part 1- Kernel Threads-

In this part, we are required to create three system calls, given below

a) thread_creation
(b) thread_join
c)thread_exit

thread_creation - This system call will create a new kernel-level thread that shares the address space with the calling process.

thread_join - This call waits for a child thread that shares the address space with the calling process. It will return the child's PID (process ID) or -1 if the child doesn't exit.

thread_exit - This call allows a thread to terminate.

Now, we will start creating these three new system calls by adding/changing some of the files of the **xv6 OS**

Following are the files that are going to be updated in adding these system calls -

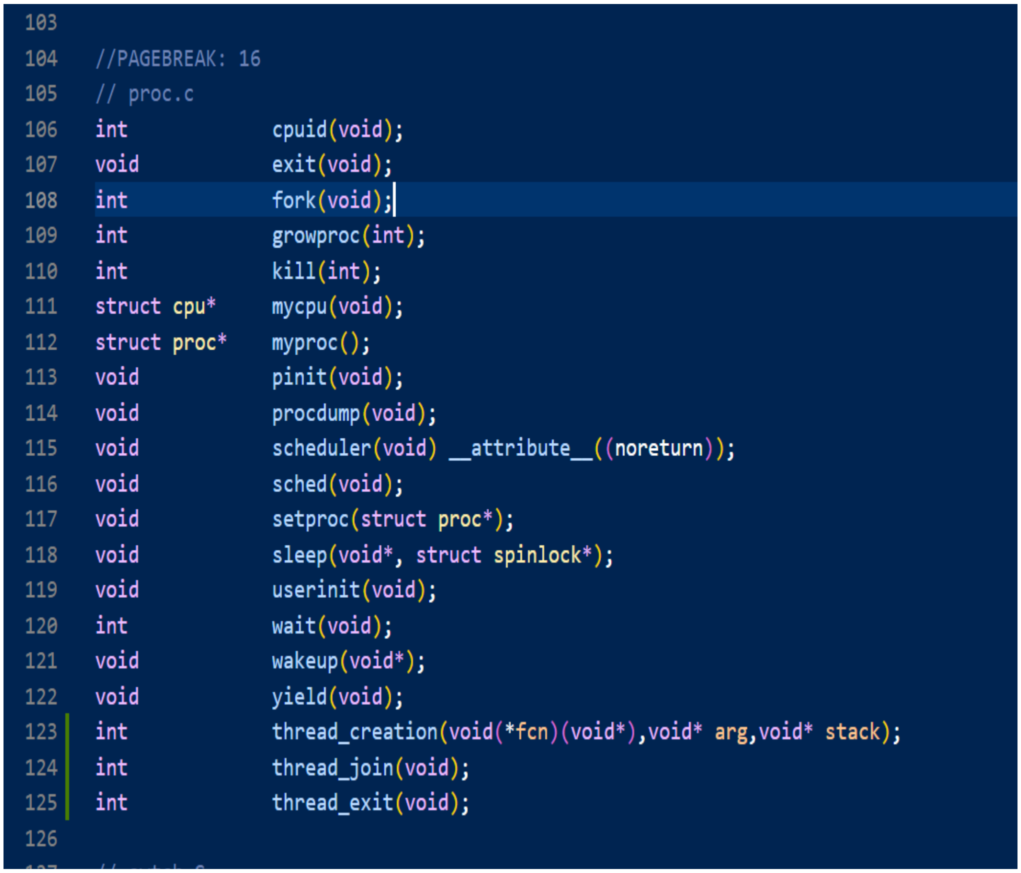
The following will be updated

1. xv6-public/defs.h
2. xv6-public/proc.c
3. xv6-public/syscall.c
4. xv6-public/syscall.h
5. xv6-public/sysproc.c
6. xv6-public/user.h
7. xv6-public/usys.S

1. xv6-public/defs.h

```
 int             thread_creation(void(*fcn)(void*),void* arg,void* stack);
int             thread_join(void);
int             thread_exit(void);
```

I added the above three lines of code inside the defs.h file because this file contains the prototype for all the system calls, and we also create new system calls, so we have to add prototypes for our system calls. The below screenshot shows where we have added these lines.



```

//PAGEBREAK: 16
// 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);
int             thread_creation(void(*fcn)(void*),void* arg,void* stack);
int             thread_join(void);
int             thread_exit(void);

```

2. xv6-public/proc.c

```
 // 3 extra added functions
int
thread_creation(void(*fcn)(void*), void *arg, void *stack)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  np->sz = curproc->sz;
  np->parent = curproc;

  np->pgdir = curproc->pgdir;
  *np->tf = *curproc->tf;
  np->tf->eax = 0;
  np->tf->eip = (uint)fcn;
  //np->stack = (uint)stack;
  np->tf->esp = (uint)stack + 4092;
  *((uint *)(np->tf->esp)) = (uint)arg;
  *((uint *)(np->tf->esp - 4)) = 0xFFFFFFFF;
  np->tf->esp -= 4;

  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)
{
  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->state = UNUSED;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        release(&ptable.lock);
        return pid;
      }
    }

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


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

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

The above lines of the code are actual function definitions of the functions

thread_creation
thread_join,
thread_exit

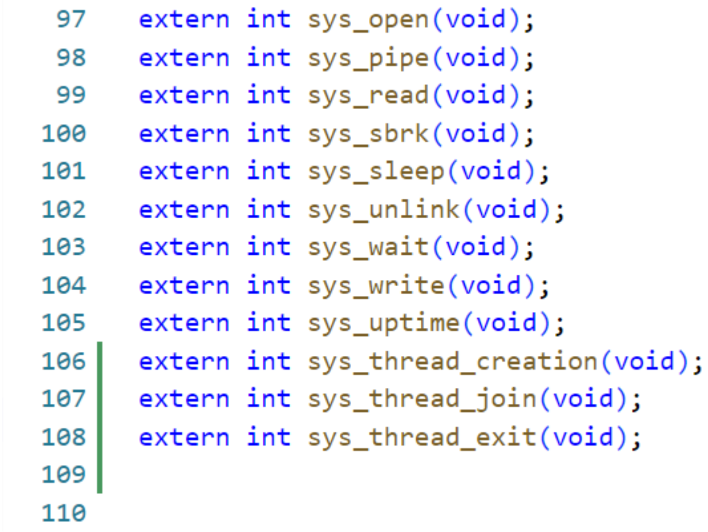
3. xv6-public/syscall.c

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

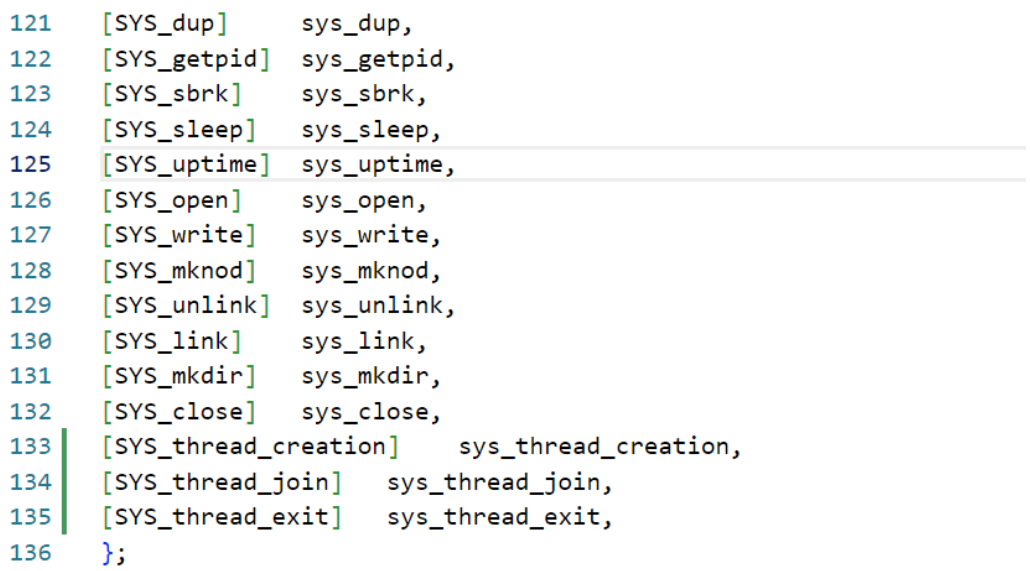
And

```
[SYS_thread_creation]    sys_thread_creation,
[SYS_thread_join]   sys_thread_join,
[SYS_thread_exit]   sys_thread_exit,
```

These lines of code are added in this file to make the system understand that these are also the system calls as the others like fork, exit, kill, etc. The below screenshot shows where we have added these lines of code



```
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_thread_creation(void);
extern int sys_thread_join(void);
extern int sys_thread_exit(void);


```

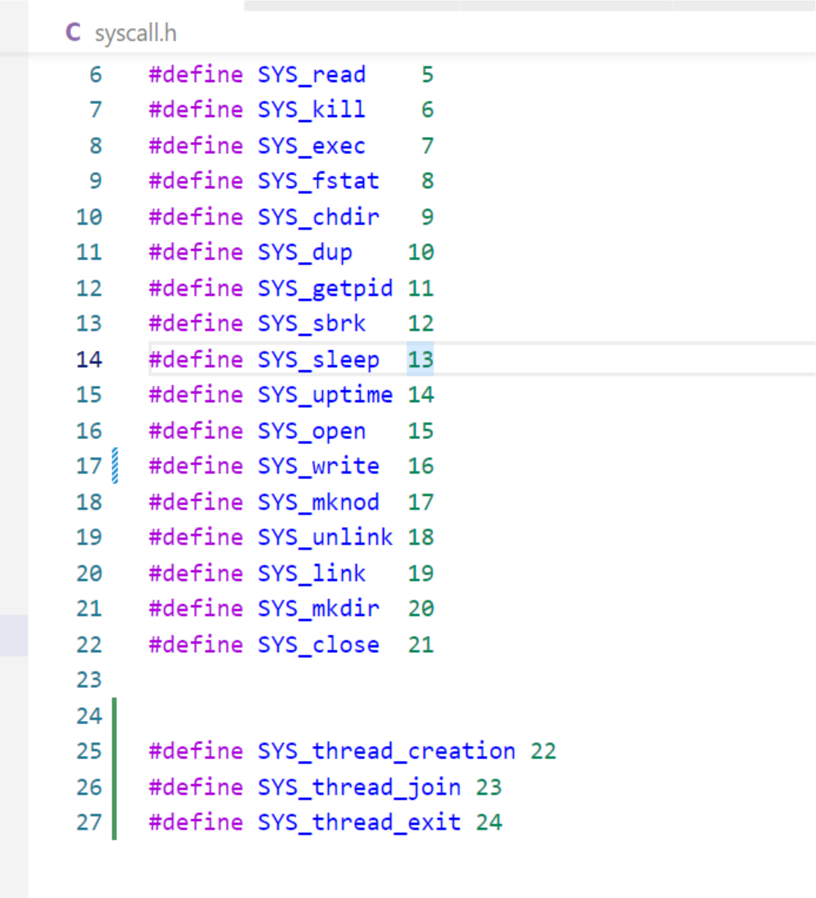


```
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_thread_creation]    sys_thread_creation,
[SYS_thread_join]   sys_thread_join,
[SYS_thread_exit]   sys_thread_exit,
};
```

4. xv6-public/syscall.h

```
#define SYS_thread_creation 22
#define SYS_thread_join 23
#define SYS_thread_exit 24
```

The above lines of code add 3 new system calls macro to the list of already existing system call macros with corresponding IDs. The following screenshot shows where these lines are added.



```
#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_creation 22
#define SYS_thread_join 23
#define SYS_thread_exit 24
```

5. xv6-public/sysproc.c

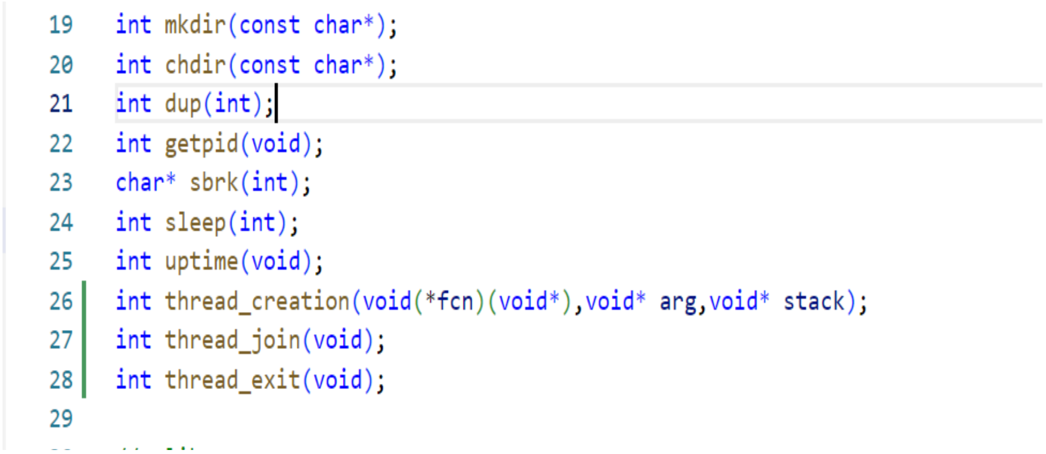
```
int sys_thread_creation(void){

  void(*fcn)(void*),*arg,*stack;
  argptr(0,(void*)&fcn,sizeof(void(*)(void*)));
   argptr(1,(void*)&arg,sizeof(void(*)));
    argptr(2,(void*)&stack,sizeof(void(*)));
    return thread_creation(fcn,arg,stack);

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

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

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.

6. xv6-public/user.h

```
int thread_creation(void(*fcn)(void*),void* arg,void* stack);
int thread_join(void);
int thread_exit(void);
```

This file also contains the prototypes for the system calls. The below screenshot shows where we have added these lines.



```
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int thread_creation(void(*fcn)(void*),void* arg,void* stack);
int thread_join(void);
int thread_exit(void);

```

7. xv6-public/usys.S

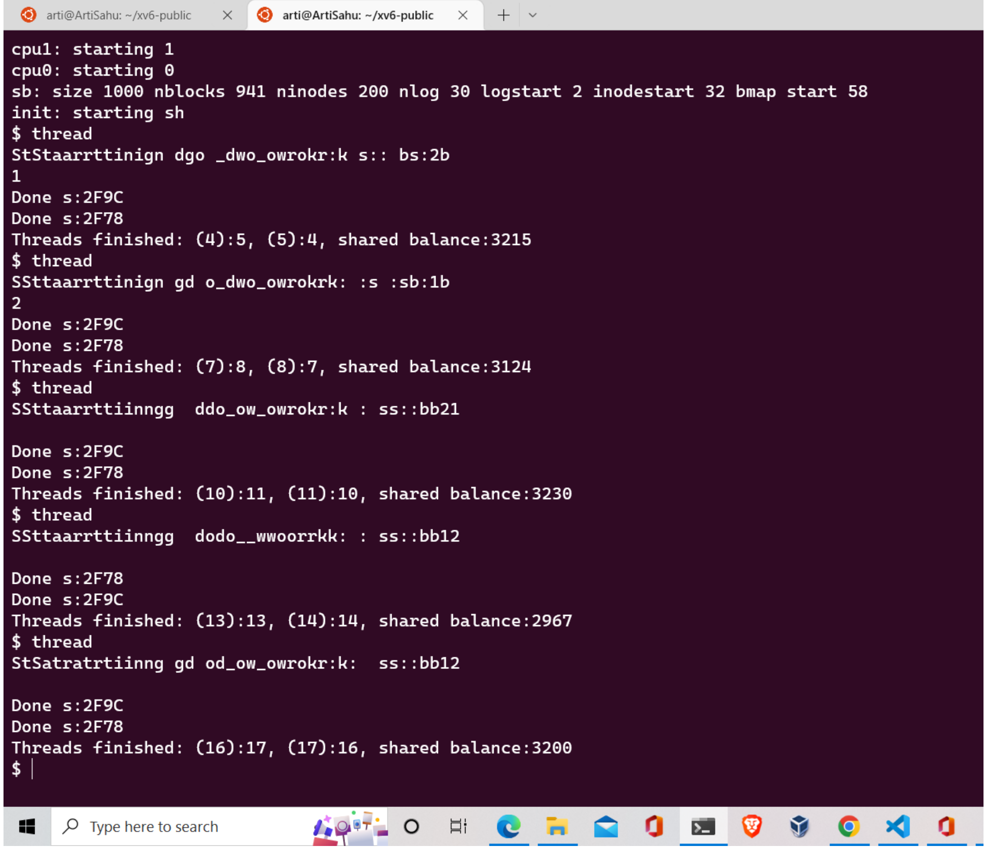
```
SYSCALL(thread_creation)
SYSCALL(thread_join)
SYSCALL(thread_exit)
```

This file contains lines of code corresponding to binding the functions with the system.

Part 02 - Synchronization

In this part, we have to work on the synchronization part by making **spinlocks** and **mutexes**. We have created two balance structures **b1** and **b2,** which have an initial amount of **3200** and **2800** respectively. And we have created two threads to update the shared balance variable by running the do_work function in both of them, but if they don't undergo execution with synchronization, the value of the share balance doesn't come as expected (6000), so to get rid of this issue we have to create **spinlocks** but the problem with spinlocks is that they can't run properly on a single processor system, or when the system is heavy load because all the spinlocks will spin endlessly, waiting for an interrupted thread to be rescheduled and run again that makes **spinlock** inefficient. So, to solve this problem, we are going to use mutexes in place of **spinlocks**

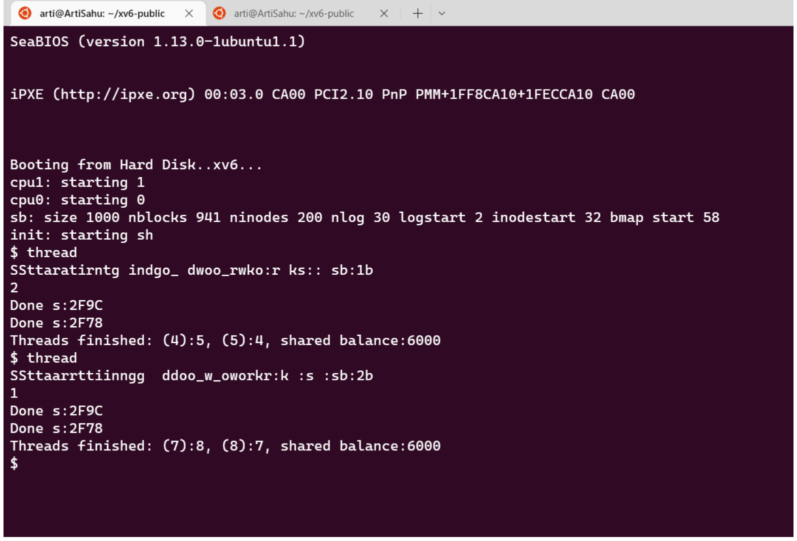
The output of 5 time calls without using synchronization



```
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
StStaarrttinign dgo _dwo_owrokr:k s:: bs:2b
1
Done s:2F9C
Done s:2F78
Threads finished: (4):5, (5):4, shared balance:3215
$ thread
SSttaarrttinign gd o_dwo_owrokrk: :s :sb:1b
2
Done s:2F9C
Done s:2F78
Threads finished: (7):8, (8):7, shared balance:3124
$ thread
SSttaarrttiinngg  ddo_ow_owrokr:k : ss::bb21

Done s:2F9C
Done s:2F78
Threads finished: (10):11, (11):10, shared balance:3230
$ thread
SSttaarrttiinngg  dodo__wwoorrkk: : ss::bb12

Done s:2F78
Done s:2F9C
Threads finished: (13):13, (14):14, shared balance:2967
$ thread
StSatratrtiinng gd od_ow_owrokr:k:  ss::bb12

Done s:2F9C
Done s:2F78
Threads finished: (16):17, (17):16, shared balance:3200
$ 
```

As we can see that the output is not coming as expected.

The output after synchronization with mutexes



```
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 start 58
init: starting sh
$ thread
SSttaratirntg indgo_ dwoo_rwko:r ks:: sb:1b
2
Done s:2F9C
Done s:2F78
Threads finished: (4):5, (5):4, shared balance:6000
$ thread
SSttaarrttiinngg  ddoo_w_oworkr:k :s :sb:2b
1
Done s:2F9C
Done s:2F78
Threads finished: (7):8, (8):7, shared balance:6000
$
```

We have added a file named **thread.c** to **xv6-public** and also updated some of the files of the **xv6-public**

xv6-public/thread.c

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

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

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

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

struct thread_mutex {
        uint locked;     // Is the lock held?

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

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

// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
```

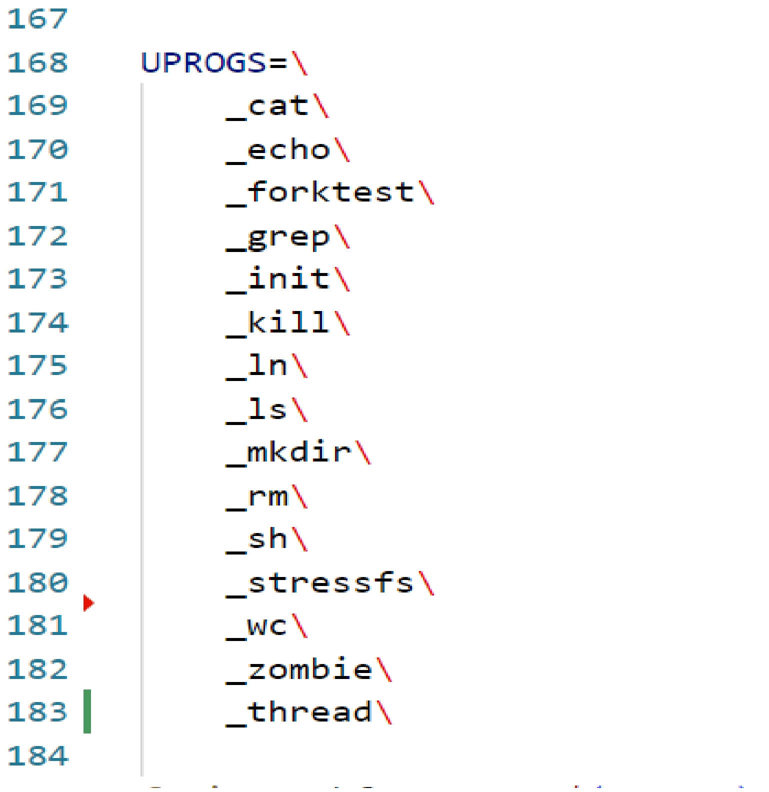
```
void
thread_spin_lock(struct thread_spinlock *lk)
{
        // The xchg is atomic.
        while (xchg(&lk->locked, 1) != 0)
                ;

        // Tell the C compiler and the processor to not move loads or stores
        // past this point, to ensure that the critical section's memory
        // references happen after the lock is acquired.
        __sync_synchronize();

}

// Release the lock.
void
thread_spin_unlock(struct thread_spinlock *lk)
{

        // Tell the C compiler and the processor to not move loads or stores
        // past this point, to ensure that all the stores in the critical
        // section are visible to other cores before the lock is released.
        // Both the C compiler and the hardware may re-order loads and
        // stores; __sync_synchronize() tells them both not to.
        __sync_synchronize();

        // Release the lock, equivalent to lk->locked = 0.
        // This code can't use a C assignment, since it might
        // not be atomic. A real OS would use C atomics here.
        asm volatile("movl $0, %0" : "+m" (lk->locked) : );
}

void
thread_mutex_init(struct thread_mutex *m, char *name)
{
        m->name = name;
        m->locked = 0;
}

// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
```

```
thread_mutex_lock(struct thread_mutex *m)
{
        // The xchg is atomic.
        while (xchg(&m->locked, 1) != 0)
                sleep(1);

        // Tell the C compiler and the processor to not move loads or stores
        // past this point, to ensure that the critical section's memory
        // references happen after the lock is acquired.
        __sync_synchronize();

}

// Release the lock.
void
thread_mutex_unlock(struct thread_mutex *m)
{

        // Tell the C compiler and the processor to not move loads or stores
        // past this point, to ensure that all the stores in the critical
        // section are visible to other cores before the lock is released.
        // Both the C compiler and the hardware may re-order loads and
        // stores; __sync_synchronize() tells them both not to.
        __sync_synchronize();

        // Release the lock, equivalent to lk->locked = 0.
        // This code can't use a C assignment, since it might
        // not be atomic. A real OS would use C atomics here.
        asm volatile("movl $0, %0" : "+m" (m->locked) : );
}

struct thread_spinlock lock;
struct thread_mutex mlock;

volatile int total_balance = 0;

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

    return i;
}
```

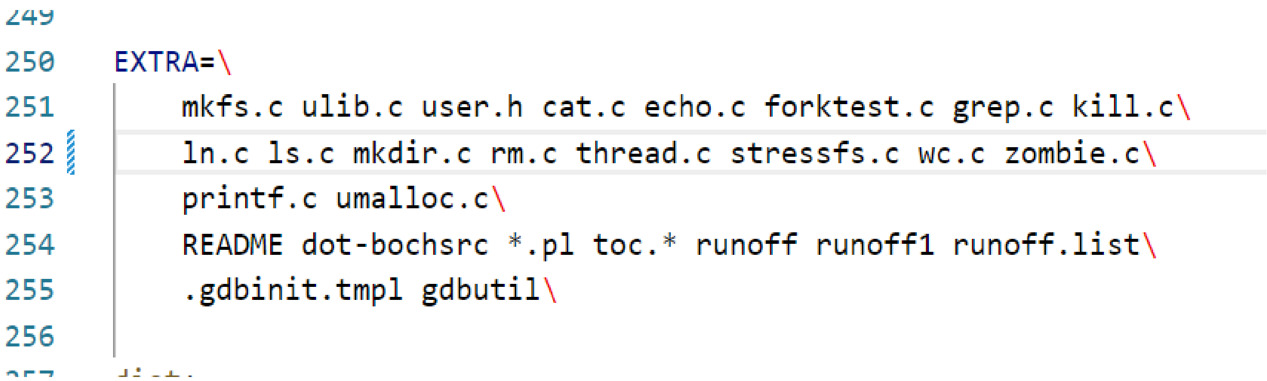
```
void do_work(void *arg){
    int i;
    int old;

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

    for (i = 0; i < b->amount; i++) {
      //  thread_mutex_lock(&mlock);
        old = total_balance;
        delay(10000);
        total_balance = old + 1;
        //thread_mutex_unlock(&mlock);
    }

    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;

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

    t1 = thread_creation(do_work, (void*)&b1, s1);
    t2 = thread_creation(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();}
```

xv6-public/Makefile

We have added the call corresponding to the **thread.c** in this makefile to add this to the list of xv6 commands list. We have added this line of code under **UPROGS _thread** And also added the following line of code under the Extras **thread.c** The following screenshots show where we have added these lines of code.



```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_wc\
	_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 thread.c stressfs.c wc.c zombie.c\
	printf.c umalloc.c\
	README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
	.gdbinit.tmpl gdbutil\

```

Now we have made all the necessary charges, and are ready to run the test case

Steps to follow to run the test case

1. Firstly go into the xv6-public directory
2. Then run make clean command
3. Then make command
4. Then make qemu-nox command
5. Then thread to run the test case

Observation -

By using the **mutexes**, we are getting the expected output, but without using that, we are getting the wrong output.

Thank You