ASSIGNMENT 2

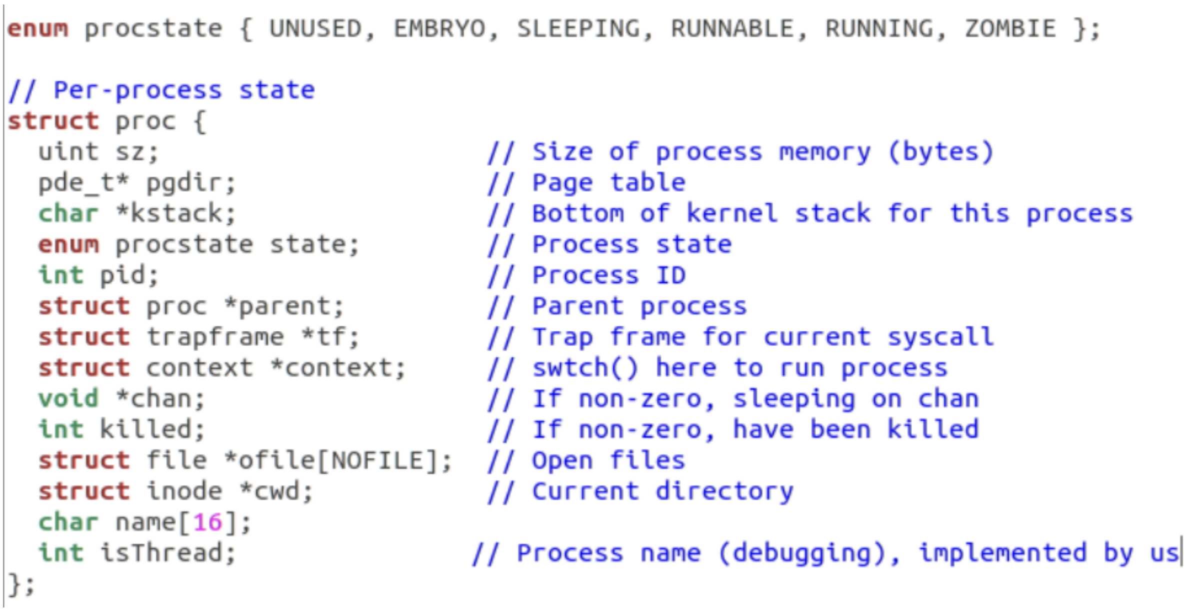
CS344 Operating System

Group : M9

Members:

Harsh	200123022
Pragyan Banerjee	200123080
Sahil Kumar Gupta	200123081

Part 1:- Kernel Threads



```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];
  int isThread;              // Process name (debugging), implemented by us
};
```

We modified the proc struct present in proc.h which basically depicts a process in XV6. We just added a isThread flag which denotes whether *this process is a thread or not.

Thread Create System Call()

```
int thread_create(void (*fcn)(void *),void *arg,void* stack)
```

It is present in proc.c and requires three arguments.

void (*fcn)(void *):- the function for which we are making the thread

void *arg:- extra arguments

void *stack:- memory to allocate for stack

This function will basically create a new thread of the existing process and return the pid of the created process.

```
if((uint)stack==0)return -1;
```

Have also handled the case when stack size is 0.

```
struct proc *newproc;
```

This is a new process which will run inside this thread

```
struct proc *curproc = myproc();
```

curproc is the process from which thread is being created

```
newproc=allocproc()
```

Allocating the current running process to np

```
newproc->pgdir = curproc->pgdir;
```

Making page table of new process to current process

```
newproc->sz = curproc->sz;
```

Making size of process memory of new process to current process

```
newproc->parent = curproc;
```

Making parent of new process to current process

```
*newproc->tf = *curproc->tf;
```

Making trap frames of new process and current process equal

```
newproc->isThread = 1;
```

Mentioning that new process is a thread

```
nnewproc->tf->eax = 0;
```

Making eax register equal to 0 so that when this thread terminates 0 is returned mentioning it was a child process

```
nnewproc->tf->eip = (int)fcn;
```

Eip contains the return address from address from where the process resumes execution

```
newproc->tf->esp = (int) stack + 4096;
```

Pointing to the bottom of the stack

```
newproc->tf->esp -= 4;
 *((int*)(newproc->tf->esp)) = (int) arg;
```

In the next address we are storing the arguments

```
newproc->tf->esp -= 4;
 *((int*)(newproc->tf->esp)) = 0xffffffff;
```

Point the next address to 0

```
for(i=0;i<NOFILE;++i){
    if(curproc->ofile[i])newproc->ofile[i]        =
filedup(curproc->ofile[i]);
}
```

Copying all the files which were open in the current process to the new process.

```
newproc->cwd = idup(curproc->cwd);
```

Making the current working directory of the new process to the current process.

```
safestrcpy(newproc->name,curproc->name,sizeof(curpro
c->name));
```

Copying name of current process to new process

```
acquire(&ptable.lock);
 newproc->state = RUNNABLE;
 release(&ptable.lock);
```

While changing the state we don’t want any other process to interfere in this process hence did the locking.

```
return par_id;
```

Returning pid of the current created thread.

Thread Join System Call()

It doesn’t require any argument. In this process what we are doing is checking whether any of the children is in a zombie state or not. If there is one then finish it and run itself. Otherwise wait for one of the running children to go into a zombie state.

Variable:

Havekids :- Means whether there exist a child of current process
struct proc *p :- processes

```
struct proc *curproc = myproc();
```

Curproc contains the current running process

```
acquire(&ptable.lock);
```

We don't want any current process to be created or destroyed while iterating through the processes.

```
for(p = ptable.proc;p< &ptable.proc[NPROC];++p){
if(p->isThread==0||p->parent!=curproc)continue;
```

This if conditions eliminates those processes which are not a thread and also those whose parent is not equal to curproc.

```
havekids = 1;
```

Marking that p is a child of current process

```
if(p->state==ZOMBIE){
```

If this child is in a zombie state then we are clearing its data and releasing the data it holds.

```
if(!havekids||curproc->killed)
```

If there was no child process or this process was killed we will release the lock and come out of this function.

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

If none of the above conditions was true then put the current process in sleep mode.

Thread Exit System Call()

This function also doesn't require any parameter and exits out of the current running thread.

```
struct proc *curproc = myproc();
```

Storing the current running thread to curproc

```
if(curproc==initproc)
 {
   panic("init exiting");
 }
```

If this current process was the init process then we are starting to exit.

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

Here we are iterating through all the files and if this file was open in the current process then we are closing this file.

```
begin_op();
     iput(curproc->cwd);
end_op();
```

Erasing the existence of current process from the current directory

```
curproc->cwd = 0;
```

Marking current processes current directory as 0.

```
 acquire(&ptable.lock);
```

While iterating the process table we don't want any other process to interfere that's why locking is done.

```
wakeup1(curproc->parent);
```

Waking the current processes parent.

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

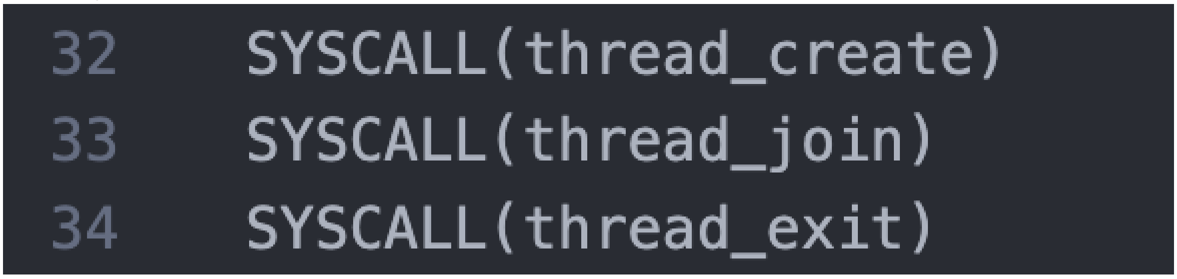
Iterating over all children and making their parent equal to init process.

```
curproc->state = ZOMBIE;
```

Declaring current processes state as zombie.

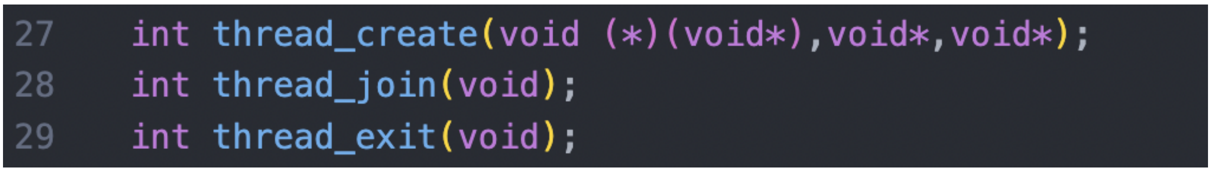
And then declaring these as system call for that we changed the following files.

In Usys.S



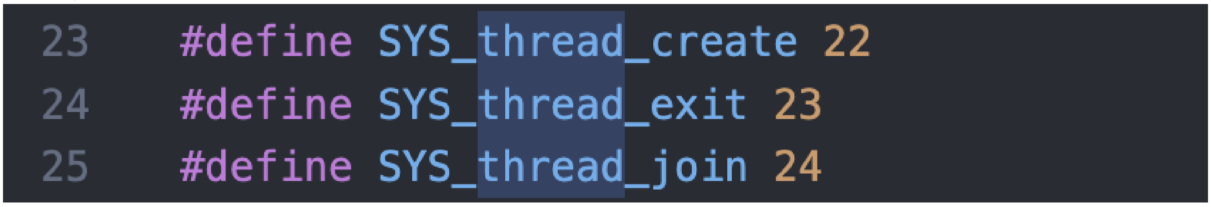
```
SYSCALL(thread_create)
SYSCALL(thread_join)
SYSCALL(thread_exit)
```

In User.h



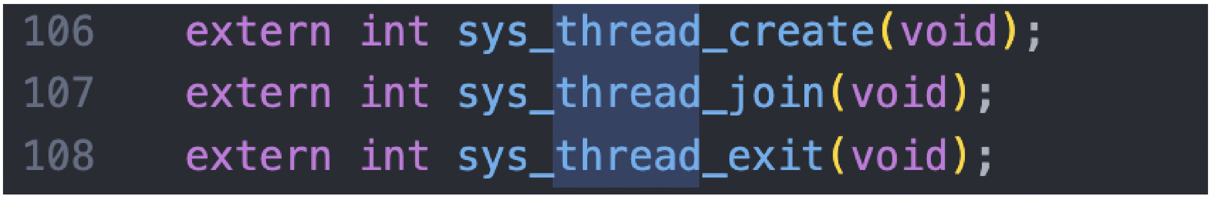
```
int thread_create(void (*)(void*),void*,void*);
int thread_join(void);
int thread_exit(void);
```

In Syscall.h

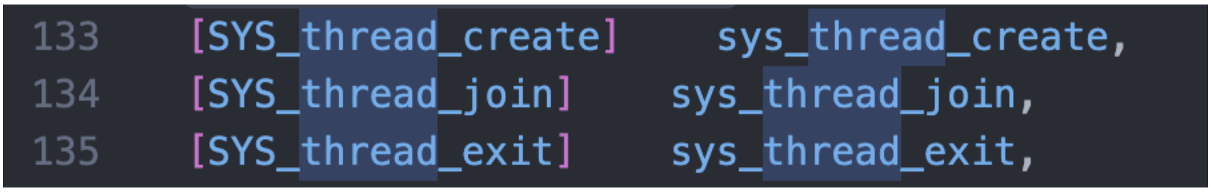


```
#define SYS_thread_create 22
#define SYS_thread_exit 23
#define SYS_thread_join 24
```

In syscall.c

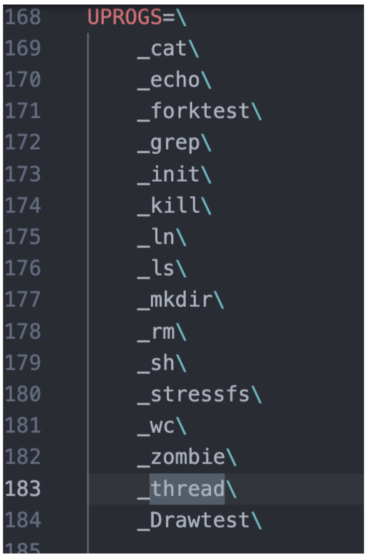


```c
extern int sys_thread_create(void);
extern int sys_thread_join(void);
extern int sys_thread_exit(void);
```

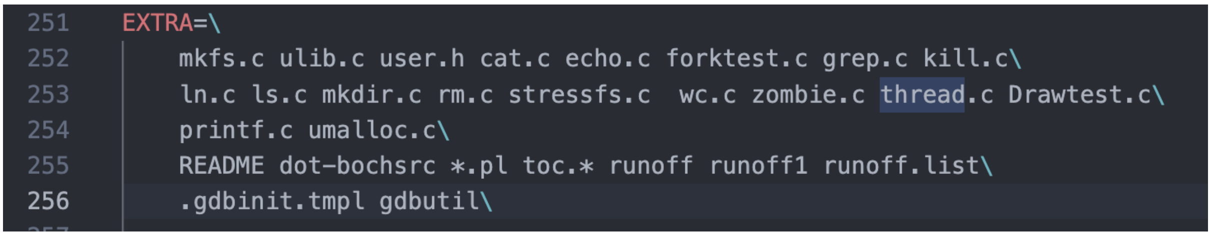


```c
[SYS_thread_create]    sys_thread_create,
[SYS_thread_join]    sys_thread_join,
[SYS_thread_exit]    sys_thread_exit,
```

In Makefile

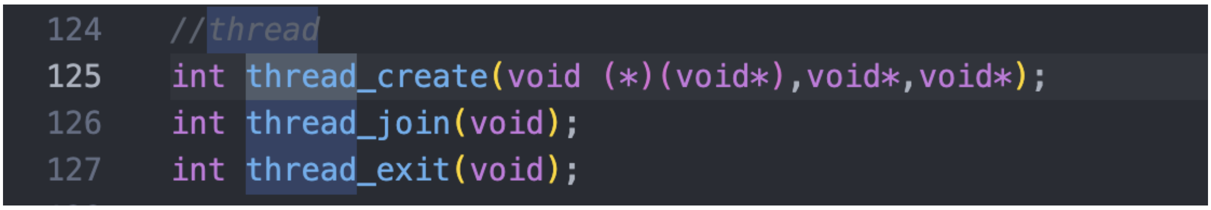


```makefile
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_wc\
	_zombie\
	_thread\
	_Drawtest\
```



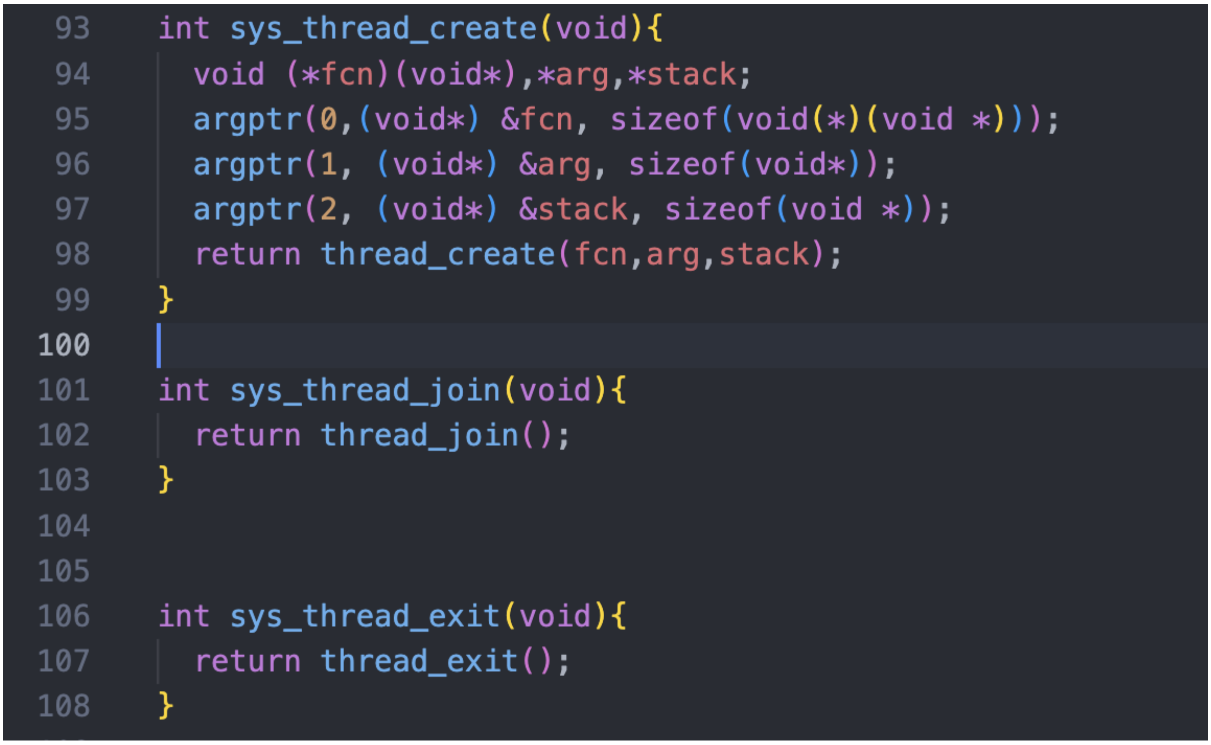
```makefile
EXTRA=\
	mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
	ln.c ls.c mkdir.c rm.c stressfs.c  wc.c zombie.c thread.c Drawtest.c\
	printf.c umalloc.c\
	README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
	.gdbinit.tmpl gdbutil\
```

In defs.h



```c
//thread
int thread_create(void (*)(void*),void*,void*);
int thread_join(void);
int thread_exit(void);
```

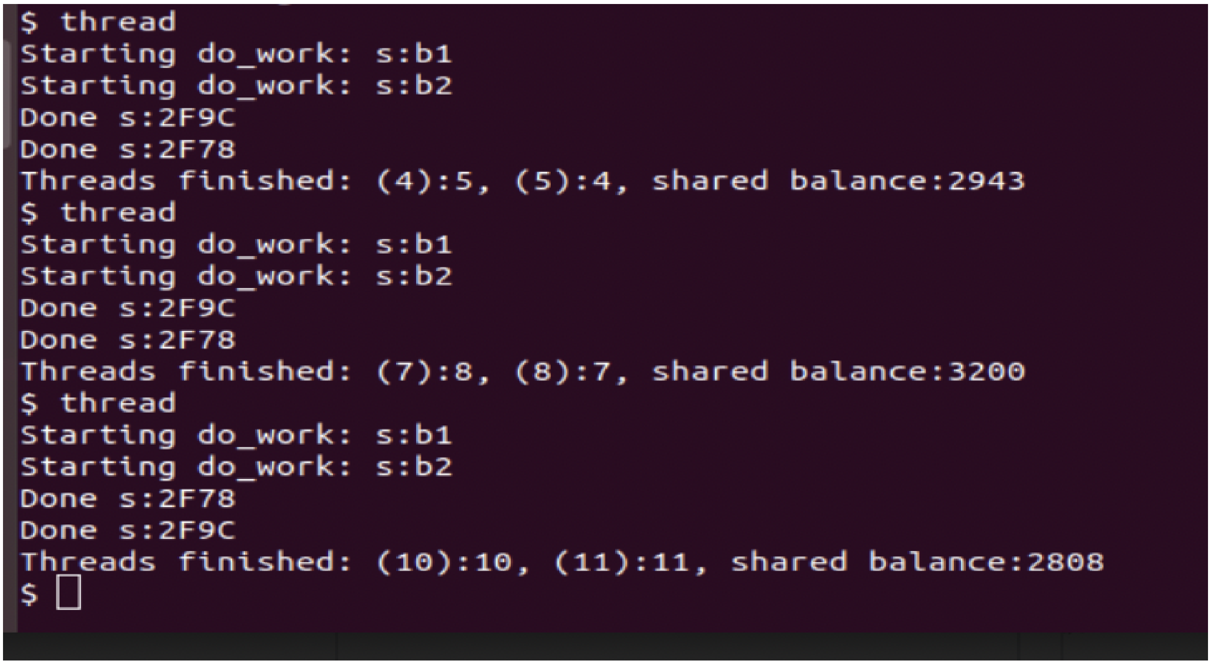
In sysproc.c



```c
int sys_thread_create(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_create(fcn,arg,stack);
}

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


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

After running the following commands-
Make clean
Make
Make qemu
thread

We got this as output



```
$ thread
Starting do_work: s:b1
Starting do_work: s:b2
Done s:2F9C
Done s:2F78
Threads finished: (4):5, (5):4, shared balance:2943
$ thread
Starting do_work: s:b1
Starting do_work: s:b2
Done s:2F9C
Done s:2F78
Threads finished: (7):8, (8):7, shared balance:3200
$ thread
Starting do_work: s:b1
Starting do_work: s:b2
Done s:2F78
Done s:2F9C
Threads finished: (10):10, (11):11, shared balance:2808
$ 
```

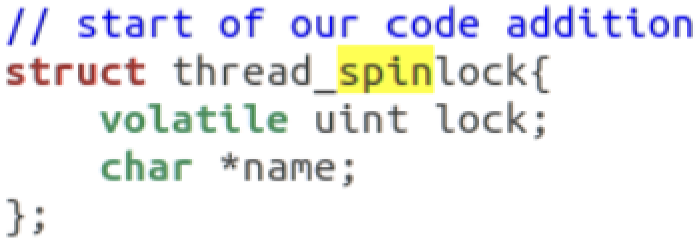
Which is not giving the correct answer which is (3200+2800) 6000 and also throwing different answers as expected because we haven't implemented any sort of synchronisation so far. Now let's do the second part of the assignment to fix this wrong answer.

Part 2: Synchronisation

Spinlocks

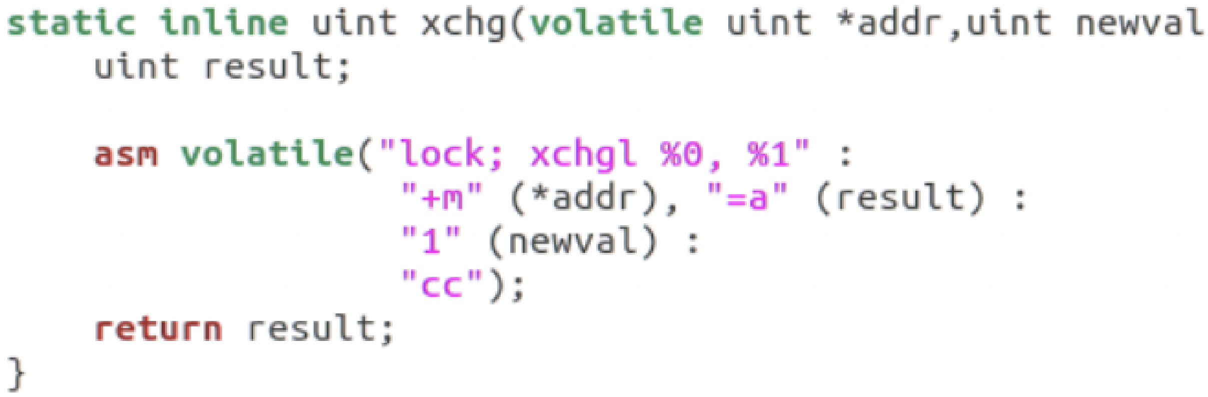
We implemented it in a separate file named as our_spinlock.h

Here is our implementation



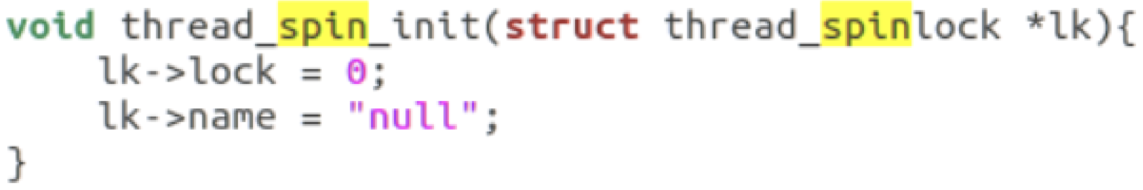
```
// start of our code addition
struct thread_spinlock{
    volatile uint lock;
    char *name;
};
```

Defined the structure of spinlock i.e., a name for it and lock state



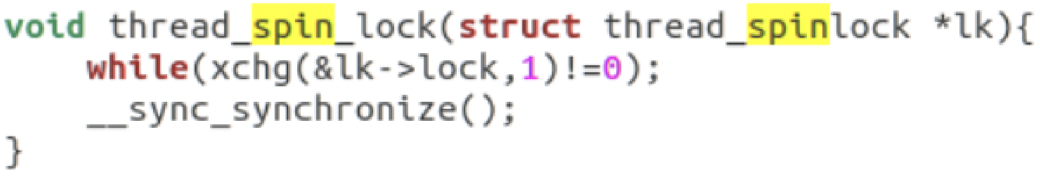
```
static inline uint xchg(volatile uint *addr,uint newval
    uint result;

    asm volatile("lock; xchgl %0, %1" :
                 "+m" (*addr), "=a" (result) :
                 "1" (newval) :
                 "cc");
    return result;
}
```

Assembly code to run infinitely till it doesn't hold true.



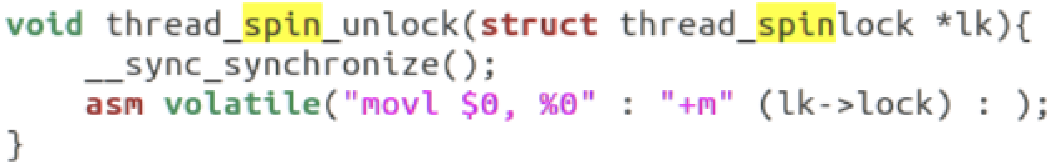
```
void thread_spin_init(struct thread_spinlock *lk){
    lk->lock = 0;
    lk->name = "null";
}
```

Initialising thread



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

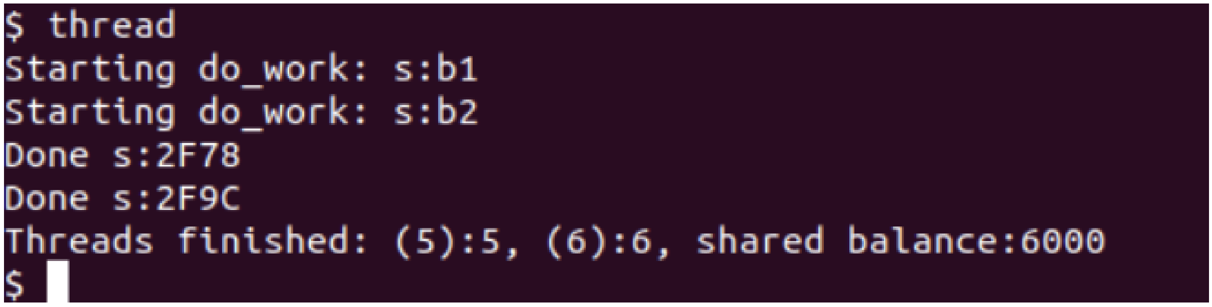
Lock implementation



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

Unlock implementation

After implementing this we ran the same commands as earlier we got the answer as 6000 finally.



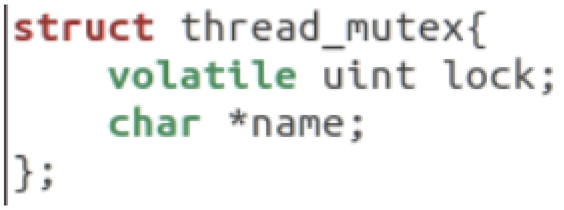
```
$ thread
Starting do_work: s:b1
Starting do_work: s:b2
Done s:2F78
Done s:2F9C
Threads finished: (5):5, (6):6, shared balance:6000
$ 
```

But as mentioned in the question it is slower and to confirm that we changed CPU from 2 to 1 in makefile and ran the code again and got the answer as and yeah it took 2-3 seconds to run the code.

To fix that we need to implement mutex which takes us to second part of second question.

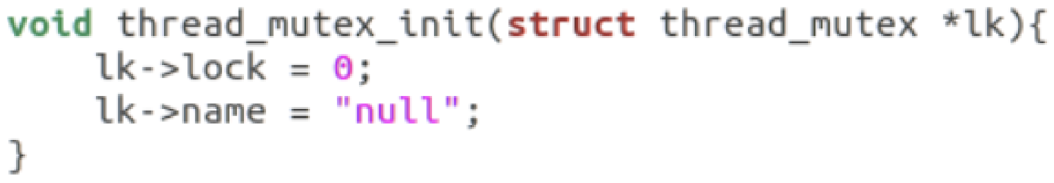
Mutex

We implemented it in a separate file named as mutex.h



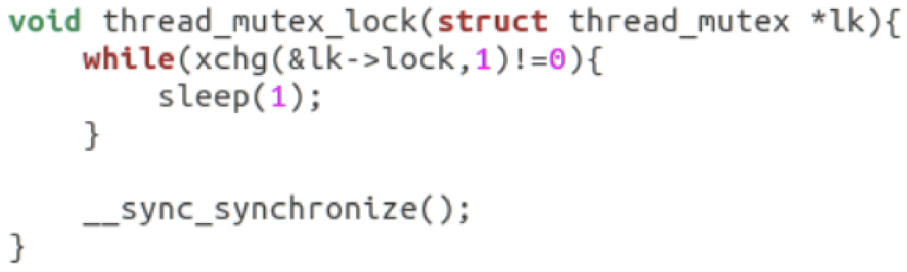
```
struct thread_mutex{
    volatile uint lock;
    char *name;
};
```

Defined the structure of spinlock, i.e a name for it and lock state



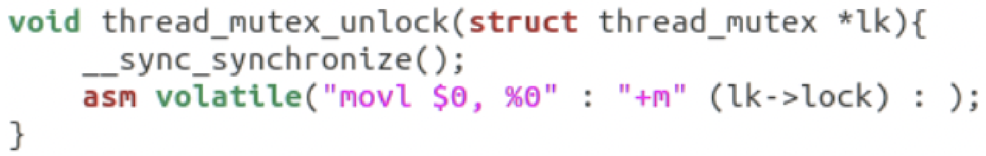
```
void thread_mutex_init(struct thread_mutex *lk){
    lk->lock = 0;
    lk->name = "null";
}
```

Initialising thread
Assembly code to run infinitely till it doesn’t hold true.



```
void thread_mutex_lock(struct thread_mutex *lk){
    while(xchg(&lk->lock,1)!=0){
        sleep(1);
    }

    __sync_synchronize();
}
```

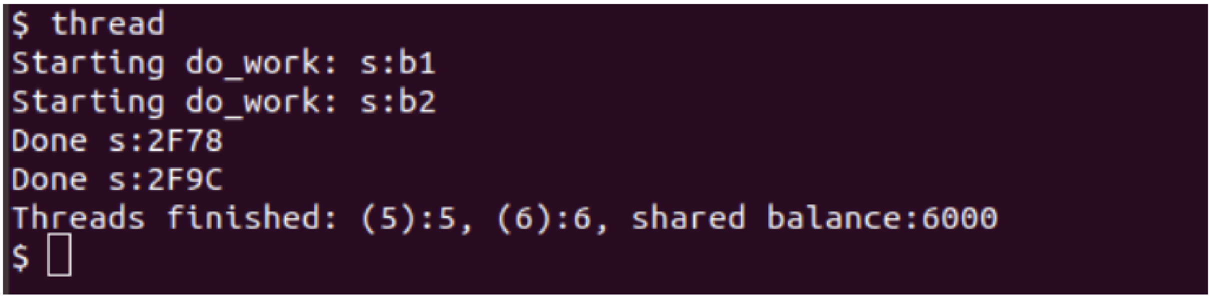
Lock implementation



```
void thread_mutex_unlock(struct thread_mutex *lk){
    __sync_synchronize();
    asm volatile("movl $0, %0" : "+m" (lk->lock) : );
}
```

Unlock implementation

We again ran the codes and commented and uncommented at necessary places and we got the same 6000 as output.



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

And again we changed cpu to 1 and ran the code again and there wasn’t any significant difference in the runtime.