

CS344 Assignment-1

Group Details:

Group Number - C31

Chandrabhushan Reddy - 200101027

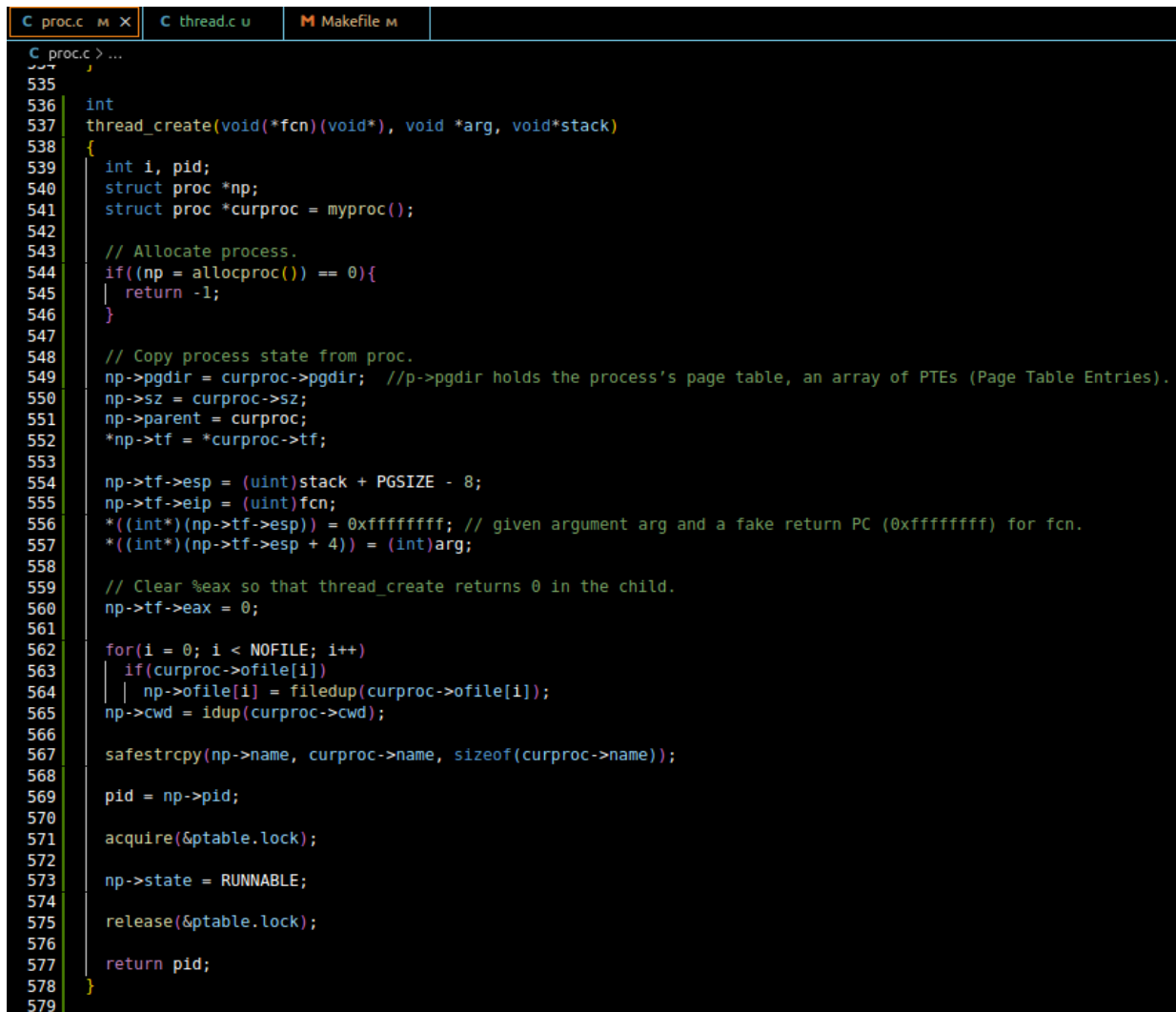
Pramodh Billa - 200101025

Kalangi Sathvika - 200101048

Part-1: Kernel threads

In the first part we should define three new system calls. They are
thread_create() - To create a new kernel thread, thread_join() - To wait
for the thread to finish and thread_exit() - To allow the thread to exit.

1) thread_create():

A screenshot of a code editor with three tabs: 'C proc.c', 'C thread.c', and 'Makefile'. The 'C thread.c' tab is active, showing the implementation of the `thread_create` function. The code is in C and implements a kernel thread. It starts with a function signature `int thread_create(void(*fcn)(void*), void *arg, void*stack)`. Inside, it declares `int i, pid;` and `struct proc *np;`. It then calls `myproc()` to get the current process. It allocates a new process structure `np` using `allocproc()`. If allocation fails, it returns -1. It then copies process state from the current process to the new one, including page directory, size, parent, and trap frame. It sets the new process's stack pointer to the provided stack and its instruction pointer to the provided function pointer. It also sets the return PC to `0xffffffff` and passes the argument `arg` to the function. It clears the `eax` register. It then duplicates the current process's open files and current working directory into the new process. It copies the process name. Finally, it sets the new process's PID, acquires the process table lock, sets its state to `RUNNABLE`, releases the lock, and returns the new PID.

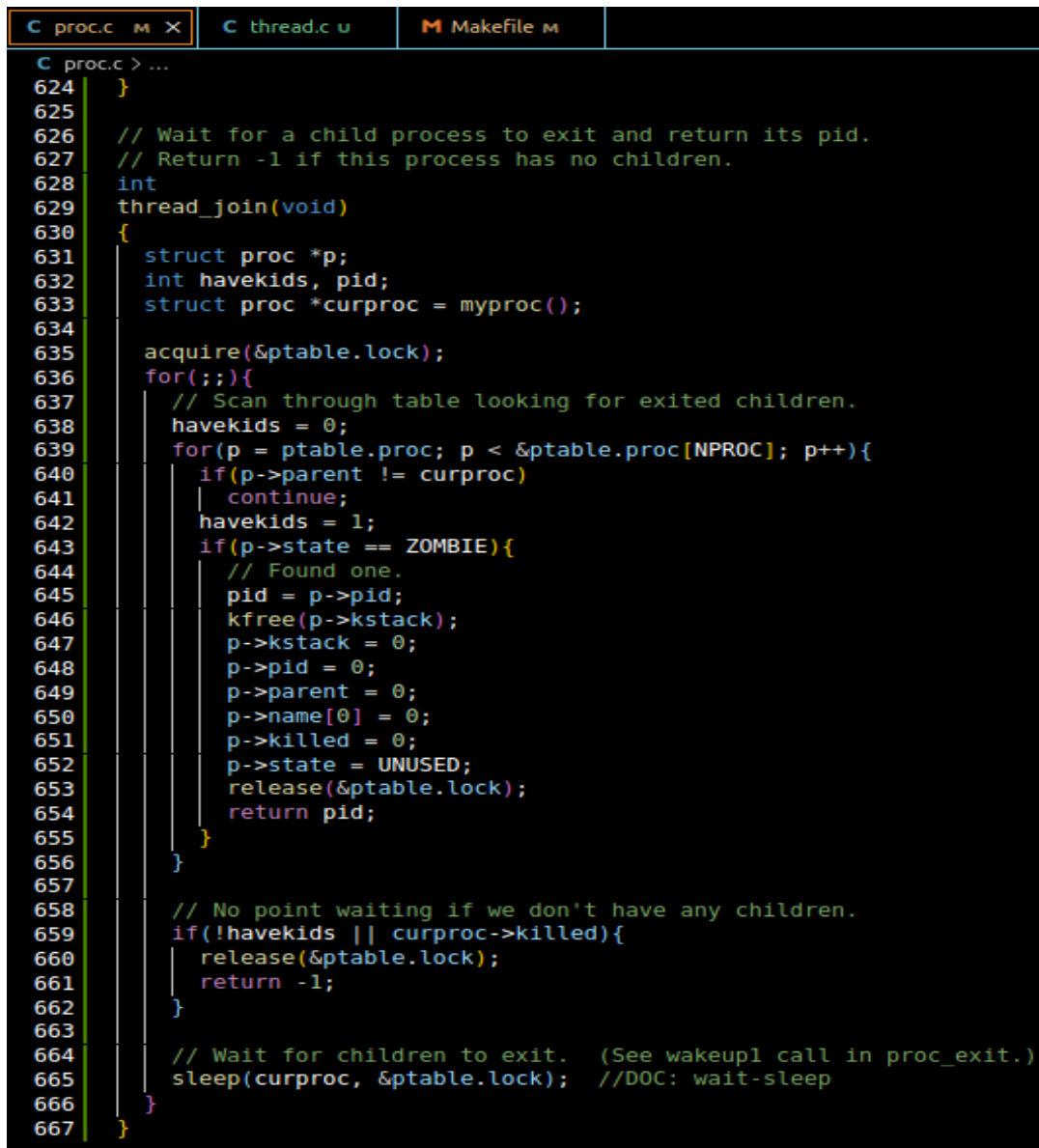
```
535
536 int
537 thread_create(void(*fcn)(void*), void *arg, void*stack)
538 {
539     int i, pid;
540     struct proc *np;
541     struct proc *curproc = myproc();
542
543     // Allocate process.
544     if((np = allocproc()) == 0){
545         return -1;
546     }
547
548     // Copy process state from proc.
549     np->pgdir = curproc->pgdir; //p->pgdir holds the process's page table, an array of PTEs (Page Table Entries).
550     np->sz = curproc->sz;
551     np->parent = curproc;
552     *np->tf = *curproc->tf;
553
554     np->tf->esp = (uint)stack + PGSIZE - 8;
555     np->tf->eip = (uint)fcn;
556     *((int*)(np->tf->esp)) = 0xffffffff; // given argument arg and a fake return PC (0xffffffff) for fcn.
557     *((int*)(np->tf->esp + 4)) = (int)arg;
558
559     // Clear %eax so that thread_create returns 0 in the child.
560     np->tf->eax = 0;
561
562     for(i = 0; i < NOFILE; i++)
563         if(curproc->ofile[i])
564             np->ofile[i] = filedup(curproc->ofile[i]);
565     np->cwd = idup(curproc->cwd);
566
567     safestrcpy(np->name, curproc->name, sizeof(curproc->name));
568
569     pid = np->pid;
570
571     acquire(&ptable.lock);
572
573     np->state = RUNNABLE;
574
575     release(&ptable.lock);
576
577     return pid;
578 }
579
```

`thread_create()` function which was asked to implement will be of the format **`int thread_create(void(*fcn)(void*), void *arg, void*stack)`**.

This call creates a new kernel thread which shares the address space with the calling process. The new process uses stack as its user stack, which is passed the given argument `arg` and uses a fake return PC (`0xffffffff`) for `fcn`. The created thread starts executing at the address specified by `fcn`. PID of the new thread is returned to the parent.

Above image is the code for implementation of `thread_create()`.

2) `thread_join()`:



```
C proc.c > ...
624 }
625
626 // Wait for a child process to exit and return its pid.
627 // Return -1 if this process has no children.
628 int
629 thread_join(void)
630 {
631     struct proc *p;
632     int havekids, pid;
633     struct proc *curproc = myproc();
634
635     acquire(&ptable.lock);
636     for(;;){
637         // Scan through table looking for exited children.
638         havekids = 0;
639         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
640             if(p->parent != curproc)
641                 continue;
642             havekids = 1;
643             if(p->state == ZOMBIE){
644                 // Found one.
645                 pid = p->pid;
646                 kfree(p->kstack);
647                 p->kstack = 0;
648                 p->pid = 0;
649                 p->parent = 0;
650                 p->name[0] = 0;
651                 p->killed = 0;
652                 p->state = UNUSED;
653                 release(&ptable.lock);
654                 return pid;
655             }
656         }
657
658         // No point waiting if we don't have any children.
659         if(!havekids || curproc->killed){
660             release(&ptable.lock);
661             return -1;
662         }
663
664         // Wait for children to exit.  (See wakeup1 call in proc_exit.)
665         sleep(curproc, &ptable.lock); //DOC: wait-sleep
666     }
667 }
```

int thread_join(void) waits for a child thread that shares the address space with the calling process. It waits for a child process to exit and return its PID. If the process has no children it returns -1.

This system call is very similar to the already existing **int wait(void)** system call in xv6. Join waits for a thread child to finish whereas wait waits for a process child to finish.

Above image is the code for implementation of thread_join()

3) thread_exit():

```
C proc.c M X C thread.c u M Makefile M
C proc.c > ...
580 // Exit the current process. Does not return.
581 // An exited process remains in the zombie state
582 // until its parent calls wait() to find out it exited.
583 void
584 thread_exit(void)
585 {
586     struct proc *curproc = myproc();
587     struct proc *p;
588     int fd;
589
590     if(curproc == initproc)
591         panic("init exiting");
592
593     // Close all open files.
594     for(fd = 0; fd < NOFILE; fd++){
595         if(curproc->ofile[fd]){
596             fileclose(curproc->ofile[fd]);
597             curproc->ofile[fd] = 0;
598         }
599     }
600
601     begin_op();
602     iput(curproc->cwd);
603     end_op();
604     curproc->cwd = 0;
605
606     acquire(&ptable.lock);
607
608     // Parent might be sleeping in wait().
609     wakeup1(curproc->parent);
610
611     // Pass abandoned children to init.
612     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
613         if(p->parent == curproc){
614             p->parent = initproc;
615             if(p->state == ZOMBIE)
616                 wakeup1(initproc);
617         }
618     }
619
620     // Jump into the scheduler, never to return.
621     curproc->state = ZOMBIE;
622     sched();
623     panic("zombie exit");
624 }
625
626 // Wait for a child process to exit and return its pid
```

The **void thread_exit(void)** system call allows a thread to terminate.

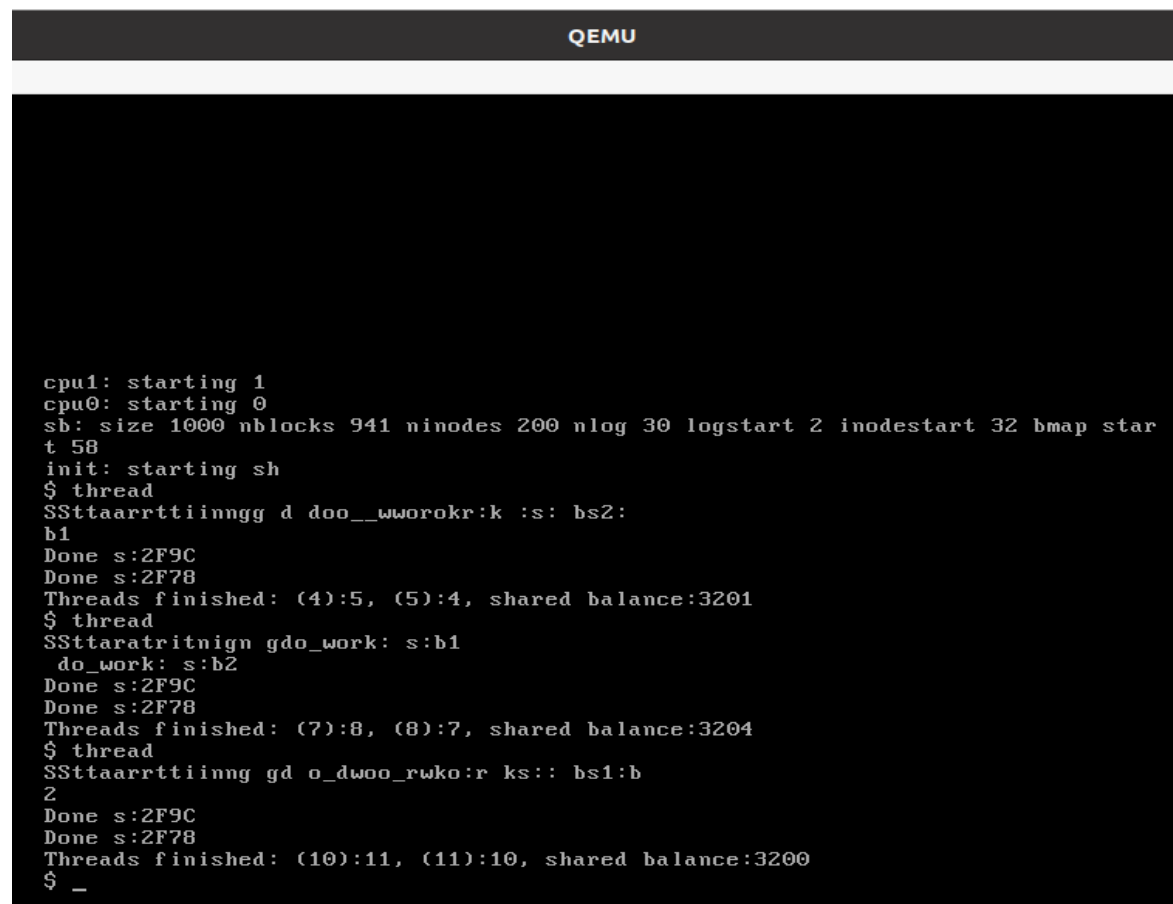
This system call is very similar to **exit()**. However, there is a small difference between the two. In the **exit()** system call we deallocate the page table of the entire process but in **thread_exit()** system call we shouldn't deallocate the page table of the entire process when one of the threads exits.

This is because all threads of the same process share the same address space to communicate and collaborate on computing a complex result in parallel. So, if one of the threads exit, the page table of the entire process shouldn't be deallocated.

The above image shows the code for implementation of thread_exit()

Testing thread implementations on the given sample code thread.c:

For the given sample code, following are the outputs obtained when thread.c was compiled and run in xv6 operating system



```
QEMU

cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ thread
SSttaarrttiinnngg d doo__wworokr:k :s: bs2:
b1
Done s:2F9C
Done s:2F78
Threads finished: (4):5, (5):4, shared balance:3201
$ thread
SSttaratritnign gdo_work: s:b1
do_work: s:b2
Done s:2F9C
Done s:2F78
Threads finished: (7):8, (8):7, shared balance:3204
$ thread
SSttaarrttiinnng gd o_dwoo_rwko:r ks:: bs1:b
2
Done s:2F9C
Done s:2F78
Threads finished: (10):11, (11):10, shared balance:3200
$ _
```

From the above image it is clear that we got 3201, 3204 and 3200 as the shared balance when we ran thread executable 3 times. None of them is the correct actual answer 6000.

This is because 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. So, we get a total shared balance which is less than 6000. Also, we got different values for shared balance when we ran different times because context switching can happen at any point of the code.

Part-2: Synchronization

Now to remove the error we got in Part-1, we use synchronization techniques. There are many synchronization techniques to solve critical section problem but, in this assignment, two such techniques are mentioned. They are **spinlocks** and **mutexes**.

Spinlocks:

```
C thread.c > do_work(void *)
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(&mutex);
87         old = total_balance;
88         delay(100000);
89         total_balance = old + 1;
90         thread_spin_unlock(&lock);
91         // thread_mutex_unlock(&mutex);
92     }
93     printf(1, "Done s:%s\n", b->name);
94     thread_exit();
95     return;
96 }
97
98 int main(int argc, char *argv[]) {
99     struct balance b1 = {"b1", 3200};
100    struct balance b2 = {"b2", 2800};
101    void *s1, *s2;
102    int t1, t2, r1, r2;
103    s1 = malloc(4096);
104    s2 = malloc(4096);
105    thread_spin_init(&lock);
106    // thread_mutex_init(&mutex);
107    t1 = thread_create(do_work, (void*)&b1, s1);
108    t2 = thread_create(do_work, (void*)&b2, s2);
109    r1 = thread_join();
110    r2 = thread_join();
111    printf(1, "Threads finished: (%d):%d, (%d):%d, shared balance:%d\n",
112    t1, r1, t2, r2, total_balance);
113    exit();
114 }
```

```

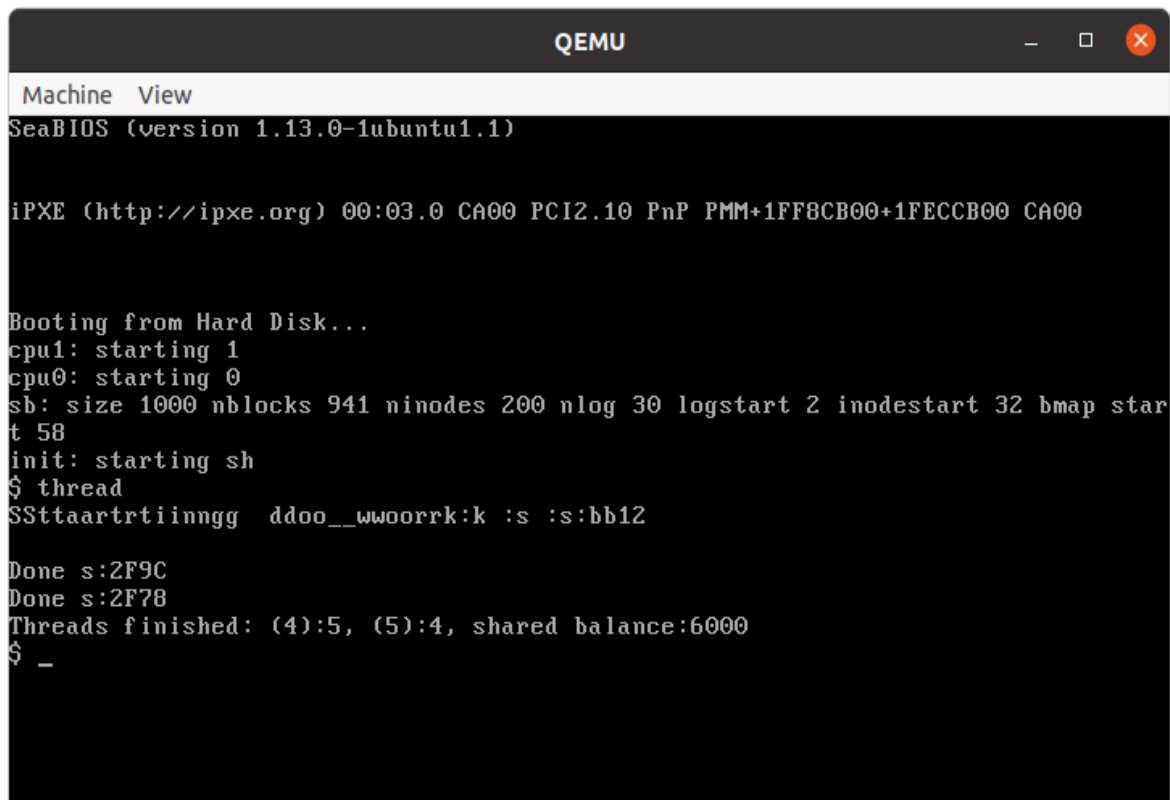
16  },
17
18  struct thread_spinlock {
19  |   uint locked;
20  };
21
22
23  void thread_spin_init(struct thread_spinlock *lk)
24  {
25  |   lk->locked = 0;
26  }
27
28  void thread_spin_lock(struct thread_spinlock *lk)
29  {
30  |   while(xchg(&lk->locked, 1) != 0)
31  |       ;
32  |   __sync_synchronize();
33  }
34
35  void thread_spin_unlock(struct thread_spinlock *lk)
36  {
37  |   __sync_synchronize();
38  |   asm volatile("movl $0, %0" : "+m" (lk->locked) : );
39  }
40
41
42

```

The above image depicts the spinlock data structure and the implementations of three functions.

The three functions are **thread_spin_init()** - Initialises the lock to correct initial state, **thread_spin_lock()** - Function to acquire lock and **thread_spin_unlock()** - Function to release the lock

Following is the output of the sample code when executed with spinlock:



```
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 star
t 58
init: starting sh
$ thread
SSttaartrtiinngg ddoo__wwoorrk:k :s :s:bb12

Done s:2F9C
Done s:2F78
Threads finished: (4):5, (5):4, shared balance:6000
$ _
```

Now we get the correct value of the shared balance as 6000 because we have used spinlock for synchronization. So, even when context switch happens only atmost one thread will be in the critical section at any given point of time. So, there won't be any data inconsistency and we get the correct answer.

Mutexes:

C thread.c > main(int, char *[])

```
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(&mutex);
87         old = total_balance;
88         delay(100000);
89         total_balance = old + 1;
90         // thread_spin_unlock(&lock);
91         thread_mutex_unlock(&mutex);
92     }
93     printf(1, "Done s:%x\n", b->name);
94     thread_exit();
95     return;
96 }
97
98 int main(int argc, char *argv[]) {
99     struct balance b1 = {"b1", 3200};
100    struct balance b2 = {"b2", 2800};
101    void *s1, *s2;
102    int t1, t2, r1, r2;
103    s1 = malloc(4096);
104    s2 = malloc(4096);
105    // thread_spin_init(&lock);
106    thread_mutex_init(&mutex);
107    t1 = thread_create(do_work, (void*)&b1, s1);
108    t2 = thread_create(do_work, (void*)&b2, s2);
109    r1 = thread_join();
110    r2 = thread_join();
111    printf(1, "Threads finished: (%d):%d, (%d):%d, shared balance:%d\n",
112           t1, r1, t2, r2, total_balance);
113    exit();
114 }
```

```

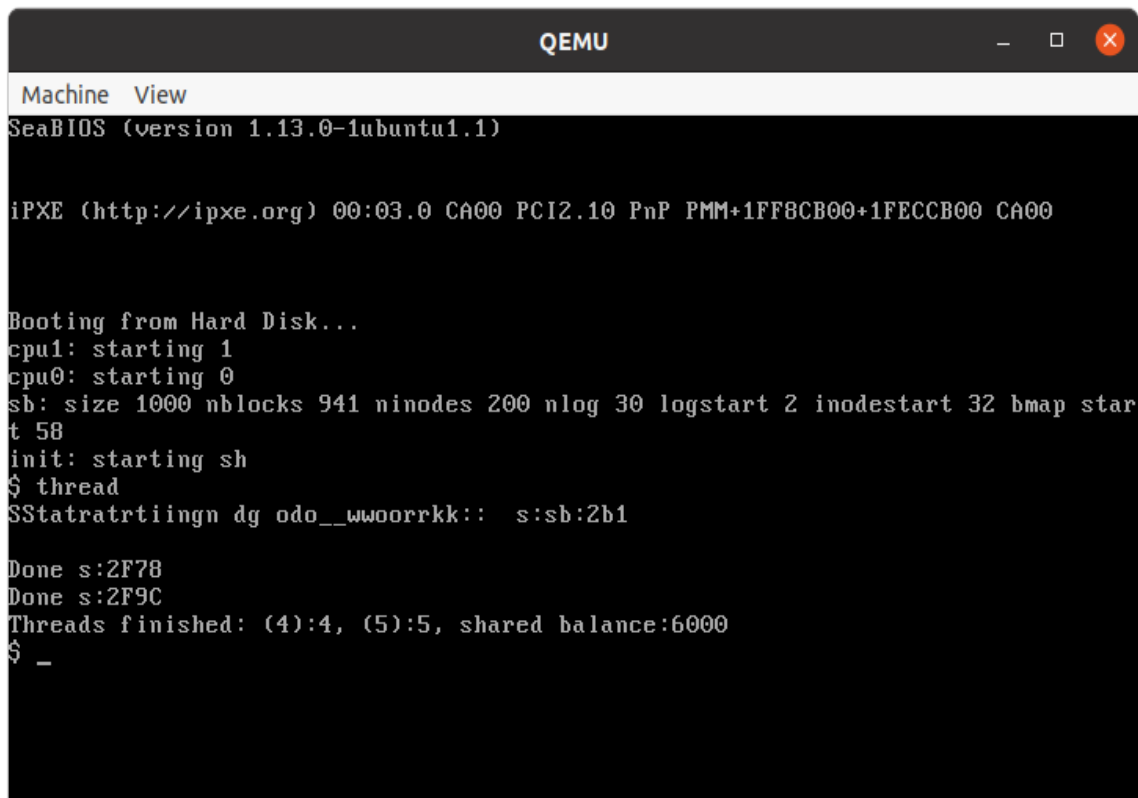
38
39  struct thread_mutex {
40  |  uint locked;
41  };
42
43  void thread_mutex_init(struct thread_mutex *mutex)
44  {
45  |  mutex->locked = 0;
46  }
47
48  void thread_mutex_lock(struct thread_mutex *mutex)
49  {
50  |  while(xchg(&mutex->locked, 1) != 0)
51  |      sleep(1);
52  |  __sync_synchronize();
53  }
54
55  void thread_mutex_unlock(struct thread_mutex *mutex)
56  {
57  |  __sync_synchronize();
58  |  asm volatile("movl $0, %0" : "+m" (mutex->locked) : );
59  }
60
61
62

```

The above image depicts the mutex data structure and the implementations of three functions.

The three functions are **thread_mutex_init()** - Initialises the mutex to correct initial state, **thread_mutex_lock()** - Function to acquire mutex and **thread_mutex_unlock()** - Function to release the mutex

Following is the output of the sample code when executed with mutex:



```
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
SStatratrtiingn dg odo__wwoorrk:: s:sb:2b1

Done s:2F78
Done s:2F9C
Threads finished: (4):4, (5):5, shared balance:6000
$ _
```

As we discussed in the case of spinlocks, even here we get the correct value for shared balance which is 6000. This is because of the mutex synchronization technique used. At any point of time only one thread can be in the critical section. So, there won't be any inconsistency in the data of the shared variables or data in the critical section. So, we get the correct value for the shared balance.

NOTE:

1) Since these functions are system calls, we need to do appropriate changes to the following files: Syscall.h, Syscall.c, Sysproc.c, User.h, Usys.h, Defs.h, Proc.c etc. All these changes have been made and uploaded in the zip file.

2) To run the thread.c test file in xv6 operating system, follow the below commands:

- i) Go to the updated xv6 directory and open a terminal.
- ii) Run make clean

iii) Run make

iv) Run make qemu

v) Now xv6 operating system's terminal would be opened. In this type thread