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():

C proc.c M × | C thread.c U | M Makefile M

C proc.c > ...

```c
}

int
thread_create(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;
  }

  // Copy process state from proc.
  np->pgdir = curproc->pgdir;  //p->pgdir holds the process's page table, an array of PTEs (Page Table Entries).
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  np->tf->esp = (uint)stack + PGSIZE - 8;
  np->tf->eip = (uint)fcn;
  *((int*)(np->tf->esp)) = 0xffffffff; // given argument arg and a fake return PC (0xffffffff) for fcn.
  *((int*)(np->tf->esp + 4)) = (int)arg;

  // Clear %eax so that thread_create returns 0 in the child.
  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;
}
```

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():

proc.c M × | thread.c U | Makefile M

proc.c > ...

```c
}

// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
thread_join(void)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        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;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

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
// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
void
thread_exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

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

  // Close all open files.
  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 **void thread_exit(void)** system call allows a thread to terminate.

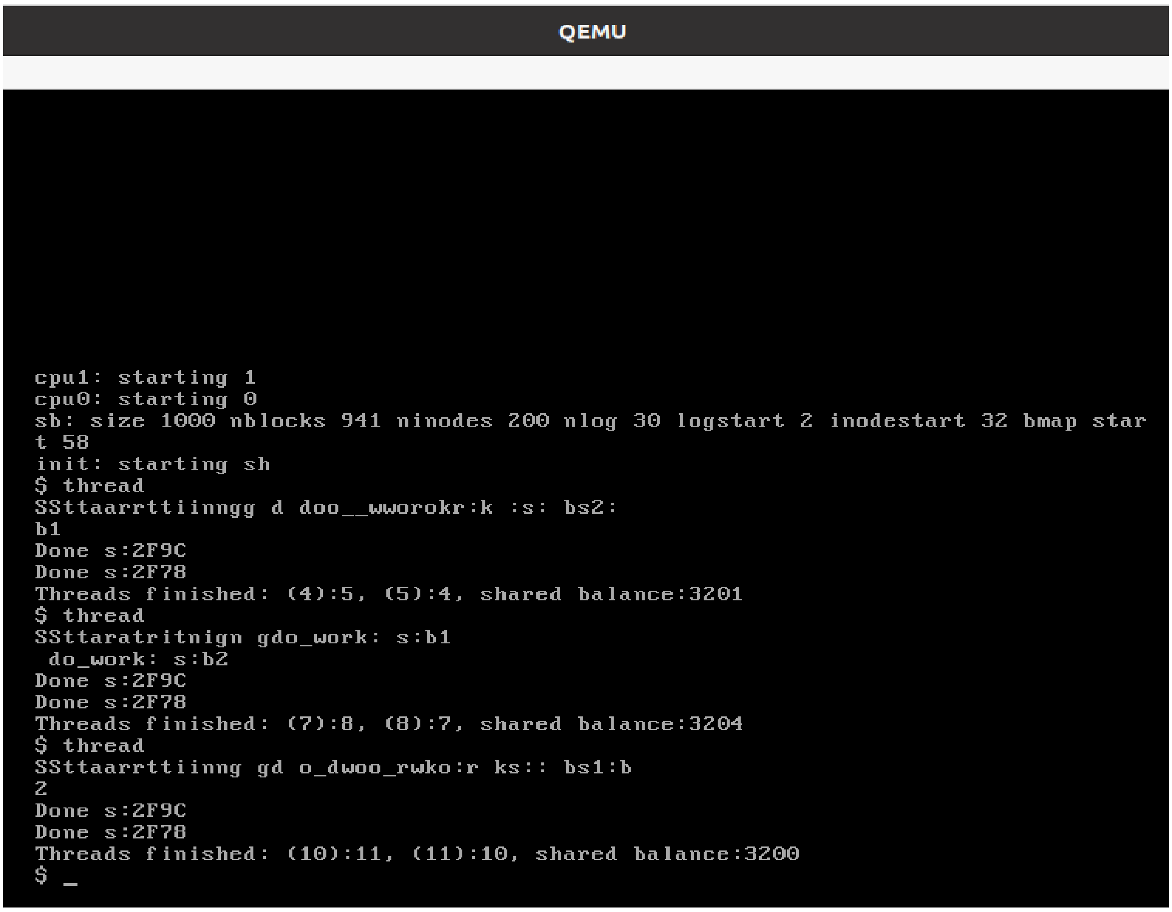
This system call is very similar to **exit()**. However, there is a small difference between the two. In the exit() system call we deallocate the page table of the entire process but in thread_exit() system call we shouldn't deallocate the page table of the entire process when one of the threads exits.

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



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:

thread.c > do_work(void *)

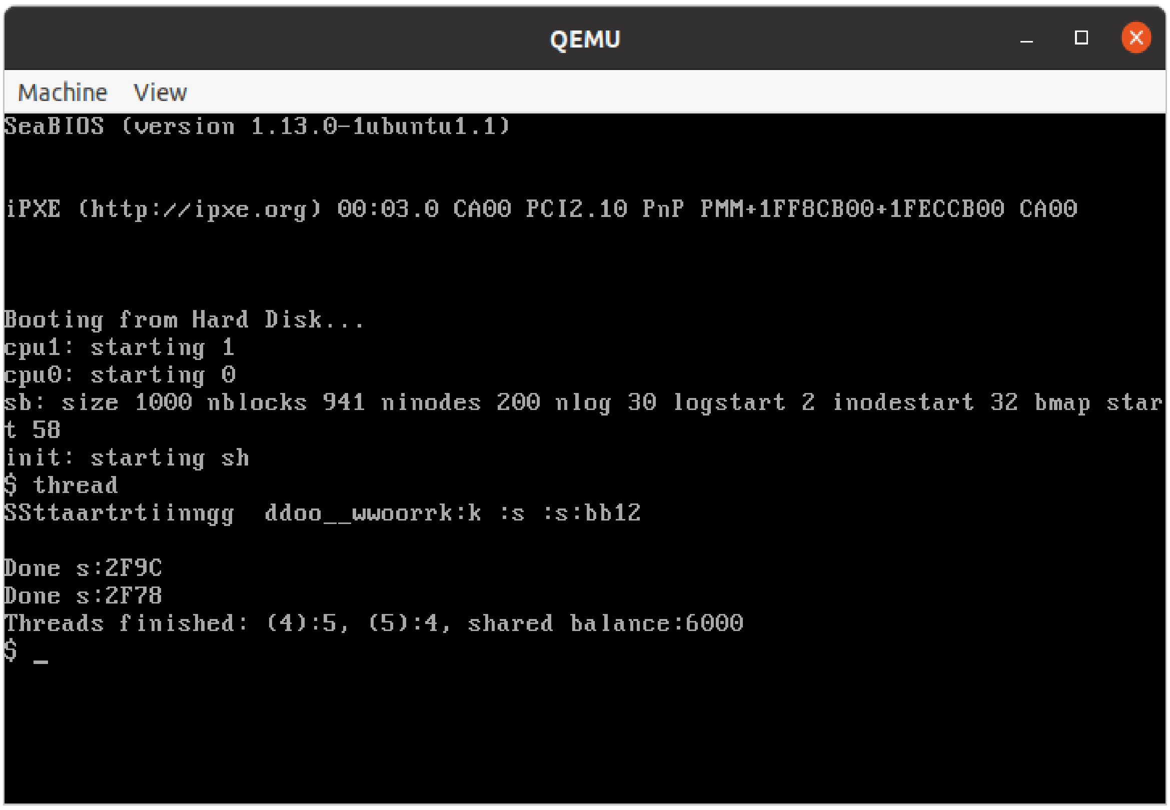
```c
void do_work(void *arg)
{
    int i;
    int old;
    struct balance *b = (struct balance*) arg;
    printf(1, "Starting do_work: s:%s\n", b->name);
    for (i = 0; i < b->amount; i++)
    {
        thread_spin_lock(&lock);
        // thread_mutex_lock(&mutex);
        old = total_balance;
        delay(100000);
        total_balance = old + 1;
        thread_spin_unlock(&lock);
        // thread_mutex_unlock(&mutex);
    }
    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);
    thread_spin_init(&lock);
    // thread_mutex_init(&mutex);
    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();
}
```

```c
struct thread_spinlock {
    uint locked;
};


void thread_spin_init(struct thread_spinlock *lk)
{
    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) : );
}
```

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:



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 * [])

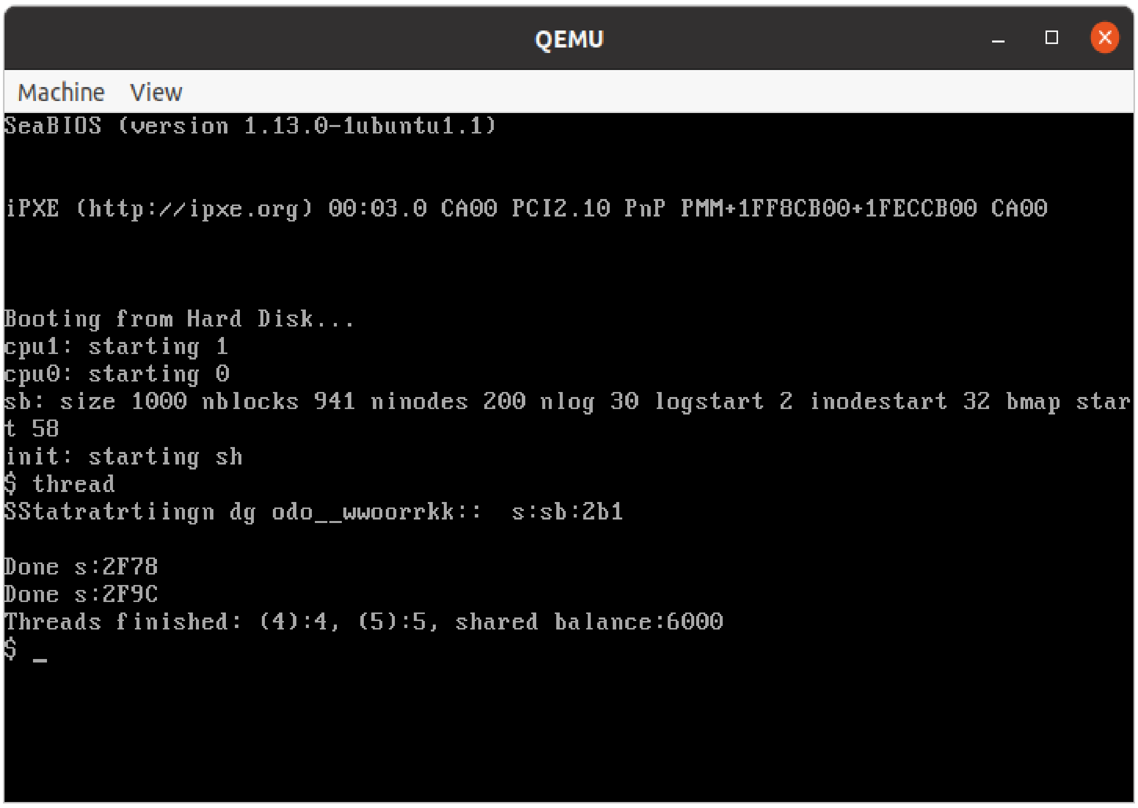
```c
void do_work(void *arg)
{
    int i;
    int old;
    struct balance *b = (struct balance*) arg;
    printf(1, "Starting do_work: s:%s\n", b->name);
    for (i = 0; i < b->amount; i++)
    {
        // thread_spin_lock(&lock);
        thread_mutex_lock(&mutex);
        old = total_balance;
        delay(100000);
        total_balance = old + 1;
        // thread_spin_unlock(&lock);
        thread_mutex_unlock(&mutex);
    }
    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);
    // thread_spin_init(&lock);
    thread_mutex_init(&mutex);
    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();
}
```

```c
struct thread_mutex {
  uint locked;
};

void thread_mutex_init(struct thread_mutex *mutex)
{
  mutex->locked = 0;
}

void thread_mutex_lock(struct thread_mutex *mutex)
{
  while(xchg(&mutex->locked, 1) != 0)
    sleep(1);

  __sync_synchronize();
}

void thread_mutex_unlock(struct thread_mutex *mutex)
{
  __sync_synchronize();

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

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:



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