CS344 : Operating systems Lab

Assignment-1

Group Details

Member - 1

Name: Dhruvesh Bhure

Roll No: 200123018

Member - 2

Name: Soham Roy

Roll No: 200123055

Member - 3

Name: Dev Sandip Shah

Roll No: 200123074

Part 1: Kernel threads

We are asked to implement 3 new system calls to simulate the working of threads.

1. ***thread_create() -*** function to create a kernel thread.
2. ***thread_join() -*** function to wait for the thread to finish.
3. ***thread_exit() -*** function that allows the thread to exit.

Structure of a process -

```
// 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 denotes whether
the process is thread or not
};
```

The variable ***isThread*** on the last line is added by us in the struct to indicate whether the current process is thread or not.

thread_create() system call looks like this:

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

Where *fcn* is the pointer to the custom function on which thread is created, *arg* is the arguments required during the function running, *stack* is manually created for thread execution.

This call creates a new kernel thread which shares the address space with the calling process.

```
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);
}
```

The code above is implemented in *sysproc.c*
thread_create() is a user call which is implemented by us in *proc.c* file which makes a system call to *sys_thread_create()* which takes the arguments from *thread_create()* and makes a system call by calling the *thread_create()* function from the kernel.

```
int thread_create(void (*fcn)(void *),void *arg,void* stack){
  if((uint)stack==0) // if no memory is allocated to the stack return -1
  {
    return -1;
  }
  int i,pid;  // pid is the process id
  struct proc *newproc;                  //new process / child process
  struct proc *current_proc = myproc();  //Current process in which thread is being created

  if((newproc=allocproc())==0)return -1;  // allocating a new process to child if not successful return -1
  newproc->pgdir = current_proc->pgdir;       // making sure both have same page directory
  newproc->sz = current_proc->sz;             // making sure both have same size
  newproc->parent = current_proc;             // pointing parent to child
  *newproc->tf = *current_proc->tf;           // same trap frame

  newproc->isThread = 1;                  // telling new process it is a thread

  newproc->tf->eax = 0;                   // when new process gets finished 0 will be returned

  newproc->tf->eip = (int)fcn;            // this is the function on which thread will work

  newproc->tf->esp = (int) stack + 4096;
  newproc->tf->esp -= 4;
  *((int*)(newproc->tf->esp)) = (int) arg;
  newproc->tf->esp-=4;
  *((int*)(newproc->tf->esp)) = 0xffffffff;
  for(i=0;i<NOFILE;++i)
  {
        if(current_proc->ofile[i])
        {
                newproc->ofile[i] = filedup(current_proc->ofile[i]); // copying all opened files from current
process to new process
        }
  }
  newproc->cwd = idup(current_proc->cwd);
  safestrcpy(newproc->name,current_proc->name,sizeof(current_proc->name));
  pid = newproc->pid;
  acquire(&ptable.lock);
  newproc->state = RUNNABLE;
  release(&ptable.lock);
  return pid;
}
```

In the above code, current process_id(*pid*) is copied into a newly created thread, then page_directory(*pgdir*) and size(*sz*) are also copied. Making customised *isThread* variable 1 to denote it a thread.

In the trapframe(*tf*) of the new process(*newproc*) we have initialised *eax*, *eip*, *esp*. Then we iterated over all the files in the current process and copied those files that were open to the new process(*newproc*).

The other new system call is :

int thread_join(void)

This call waits for a child thread that shares the address space with the calling process. It returns the *pid* of the waited-for child or -1 if none.

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

The code above is implemented in *sysproc.c*
thread_join() is a user call which is implemented by us in *proc.c* file which makes a system call to *sys_thread_join().*

```
int thread_join(void){
        struct proc *i;
        int havekids; // havekids is the boolean value to check if the thread has any child or not
        int pid;
        struct proc *current_proc = myproc(); //Current process in which thread is being created

        acquire(&ptable.lock);
        while(1){
                havekids = 0;
                for(i = ptable.proc;i< &ptable.proc[NPROC];++i){
                        if(i->isThread!=0 && i->parent==current_proc){ //if the process is a thread and child of the current process
                                havekids = 1;
                                if(i->state==ZOMBIE){ //if it's in zombie state then re-initialiaze to make it available to be  used by other processes and return
                                        pid = i->pid;
                                        i->kstack = 0;
                                        i->pid = 0;
                                        i->parent = 0;
                                        i->name[0] = 0;
                                        i->killed = 0;
                                        i->state = UNUSED;
                                        release(&ptable.lock);
                                        return pid;
                                }
                        }

                }
                if(!havekids||current_proc->killed){// if it doesn't have any child or the current process itself is aborted then return -1
                        release(&ptable.lock);
                        return -1;
                }
                sleep(current_proc,&ptable.lock); // wait for any thread to complete its execution
        }

}
```

The other new system call is :

int thread_exit(void)

This system call allows a thread to terminate.

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

The code above is implemented in *sysproc.c*

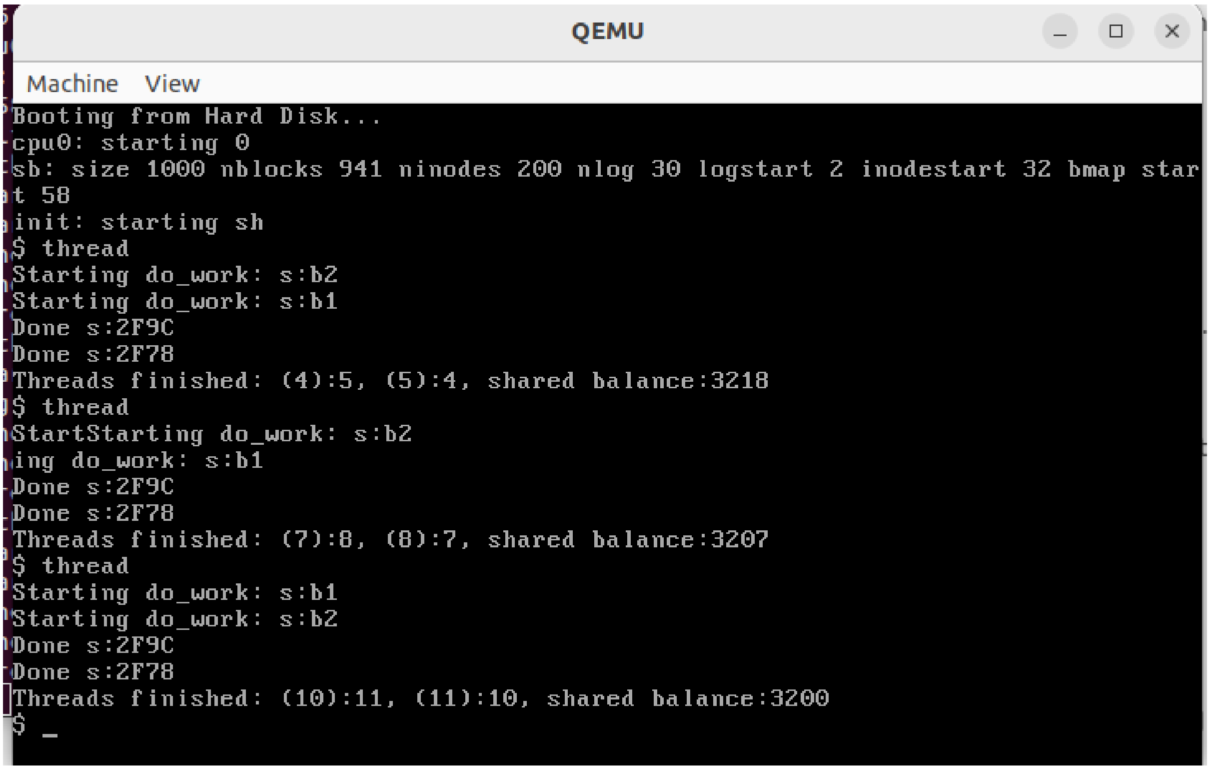
thread_exit() is a user call which is implemented by us in *proc.c* file which makes a system call to *sys_thread_exit().*

```
int thread_exit(){
        struct proc *current_proc = myproc();
        struct proc *i;
        int file;

        if(current_proc==initproc)
        {
                panic("exit init");
        }
        for(file = 0;file<NOFILE;file++){
                if(current_proc->ofile[file]){
                        fileclose(current_proc->ofile[file]); // closing all opened files of current process
                        current_proc->ofile[file] = 0;
                }
        }
        begin_op();
        iput(current_proc->cwd);
        end_op();
        current_proc->cwd = 0;
        acquire(&ptable.lock);

        wakeup1(current_proc->parent);

        for(i=ptable.proc;i<&ptable.proc[NPROC];++i){
                if(i->parent==current_proc){
                        i->parent = initproc;
                        if(i->state==ZOMBIE){
                                wakeup1(initproc);
                        }
                }
        }
        current_proc->state = ZOMBIE;
        sched();
        panic("exit zombie");
}
```



The above output is wrong (because 3200+2800 is 6000 but the received outputs are different every time we run the thread.c file). The reason behind this is that the 2 threads are changing the value of the same global variable *total_balance.* 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. This can be fixed using *synchronisation.*

Part 2: Synchronisation

To fix this synchronisation error we have implemented a ***spinlock*** that allows us to execute the update atomically.

Struct of *spinlock* :

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

There are 3 functions implemented in *spinlock* -

1. ***void thread_spin_init(struct thread_spinlock *lk)*** - initialise the spinlock to the correct initial state.
2. ***void thread_spin_lock(struct thread_spinlock *lk)*** - a function to acquire a spinlock.
3. ***void thread_spin_unlock(struct thread_spinlock *lk)*** - a function to release the spinlock.

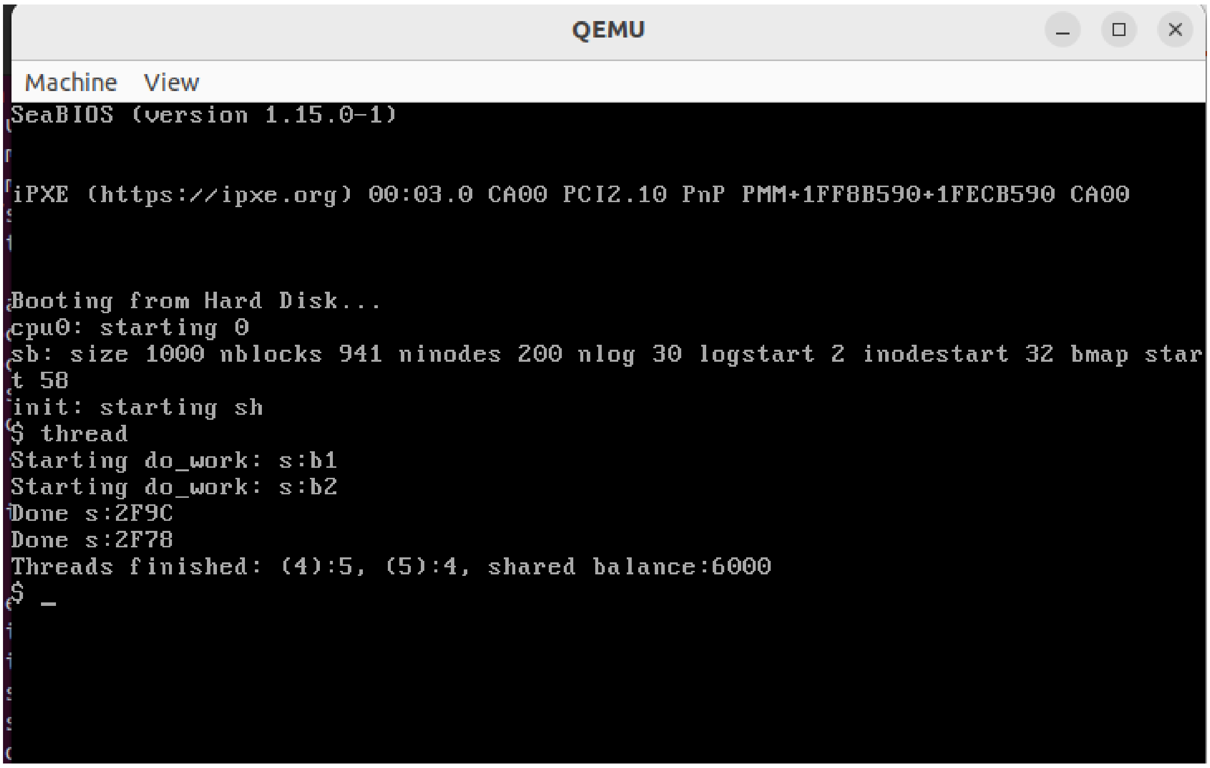
Implementation of these 3 functions are as follows :

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

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

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

Output using spinlock :



So using spinlock we got the correct output.

When the CPU count is 2, all threads of the process run in parallel on different CPUs, spinlocks are perfect, each process enters a short critical section, updates the shared balance atomically and then releases the spinlock for other threads to make progress.

```
ifndef CPUS
CPUS := 2
endif
```

However, when we change the number of CPU's in the Makefile from 2 to 1, then spinlock takes more time because all threads of the process start to spin endlessly, waiting for the interrupted (lock-holding) thread to be rescheduled and run again the spinlocks become inefficient.

```
ifndef CPUS
CPUS := 1
endif
```

To fix the above problem, we have implemented ***mutex lock.***

Struct of *mutex* :

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

There are 3 functions implemented in *mutex* -

1. ***void thread_mutex_init(struct thread_mutex *m)*** - initialise the mutex lock to the correct initial state.
2. ***void thread_mutex_lock(struct thread_mutex *m)*** - a function to acquire a mutex lock.
3. ***void thread_mutex_unlock(struct thread_mutex *m)*** - a function to release the mutex lock.

Implementation of these 3 functions are as follows :

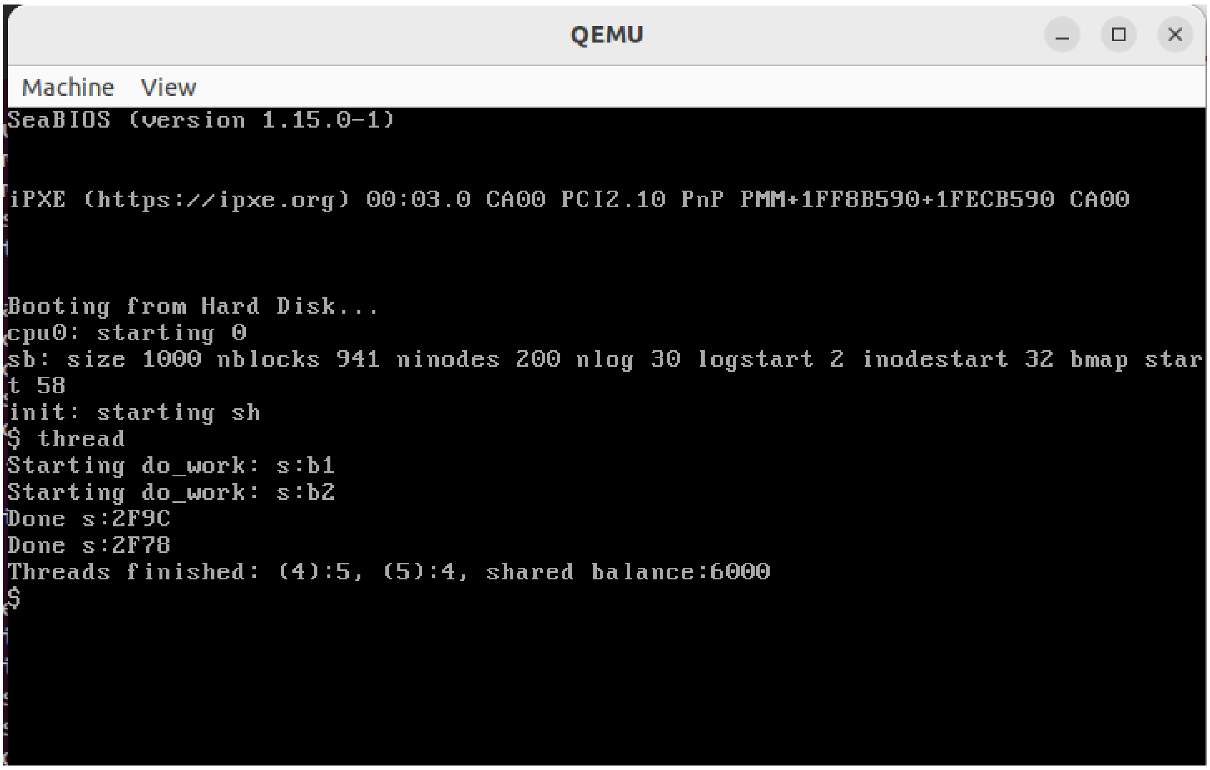
```
void thread_mutex_init(struct thread_mutex *lk){
    lk->lock = 0;
    lk->name = "null";
}

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

    __sync_synchronize();
}

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

Output using mutex :



Observations :

We observed that when we use 1 CPU, mutex lock runs much faster than spinlock because instead of spinning , mutex lock will release the CPU to another thread.

To implement all these user defined system calls, we have made the added the following pieces of codes to the respective files -

Usys.s

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

User.h

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

Syscall.h

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

Syscall.c

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

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

Defs.h

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

```

Makefile

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

```