OS344 Assignment-1

Sep 5, 2022

Group 25

Name	Roll
Anindya Vijayvargeeya	200101015
Gunjan Dhanuka	200101038
Pranjal Singh	200101084

Part 1: Kernel threads

Threads are similar to processes, in that they allow parallel execution, but share the same address space. Threads of a process have separate stack, program counter, registers and state. To implement the functions asked in this part, we took the code of fork, wait and exit and modified them so we get threads rather than processes.

Implementation

First added the implementation of all 3 functions in proc.c

thread_create:

```
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;
 }
 np->pgdir = curproc->pgdir;
 np->sz = curproc->sz;
 np->parent = curproc;
 *np->tf = *curproc->tf;


 np->tf->eip = (uint)fcn;

 // Clear %eax so that fork returns 0 in the child.
 np->tf->eax = 0;

 np->tf->esp = (uint)stack + 4096;
 np->tf->esp -= 4;
 *((uint*)(np->tf->esp)) = (uint)arg;
 np->tf->esp -= 4;
 *((uint*)(np->tf->esp)) = 0xffffffff;


 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;
}
```

In thread_create function, we use the existing fork function as the starter code. However, what fork does is create a new process while we don’t need to create a new process from scratch.

So we use the same the pgdir of the current process for the new thread by setting the same pointer values, and the trap frame as well.

In the trap frame, we point the Instruction pointer (eip) to the function pointer that is passed to the thread_create call, this will make the thread start executing from the start of the function.

Since threads have separate stacks, we set the stack pointer(esp) to the end of the stack (passes to the thread_create call) for this thread, i.e., the input stack plus the PAGE_SIZE (=4096). Then we store the arg to the end of this stack, and the fake return PC (**`0xffffffff`**). The rest of the functionality is same as the fork() function. We copy the file descriptors and current working directory from the parent, same as in fork. Finally we set its state to RUNNABLE.

thread_join:

```
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;
       //freevm(p->pgdir);
       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
 }
}
```

First lets discuss how wait works. It looks through the ptable for any process whose parent is the current process. If there are none, the function returns -1. Else, if we find a zombie child, we're able to clean it up and return its pid. If there are children but none has completed, i.e., they don't have state = ZOMBIE, then we wait for the child processes to finish, by putting the current process to sleep.

For implementing the thread_join system call, the only change from wait call is the following line:

```
      //freevm(p->pgdir);
```

Here we are commenting this line, since we don't want to free the pgdir when a thread completes. The pgdir is shared across all threads and it shouldn't be deleted by children threads.

thread_exit:

```
int
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");
}
```

In the thread_exit(), we use the exact same code from the exit() system call, since the functionality is completely same. It first clears up open file descriptors, closes current working directory, then looks for parents which if exist, wakes them up.

Finishing up

After adding the above system call implementation, we added the definitions in the defs.h file.

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

Then we added the system call mappings in the syscall.c file

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

Then we mapped the system call with a number in the syscall.h file

```
#define SYS_thread_create 23
#define SYS_thread_join 24
#define SYS_thread_exit 25
```

Then the following changes were made in the sysproc.c file from the functions will be called:

```
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();
}
```

Then from the user side, we add the function definitions in the user.h file:

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

And also in the usys.S file:

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

Now for the testing part, we add the thread.c file and the corresponding call in the Makefile:

```
#include "types.h"
#include "stat.h"
#include "user.h"
struct balance
{
   char name[32];
   int amount;
};
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_spin_lock(&lock);
       old = total_balance;
       delay(100000);
       total_balance = old + 1;
       // thread_spin_unlock(&lock);
```

```
    }
    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_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();
}
```

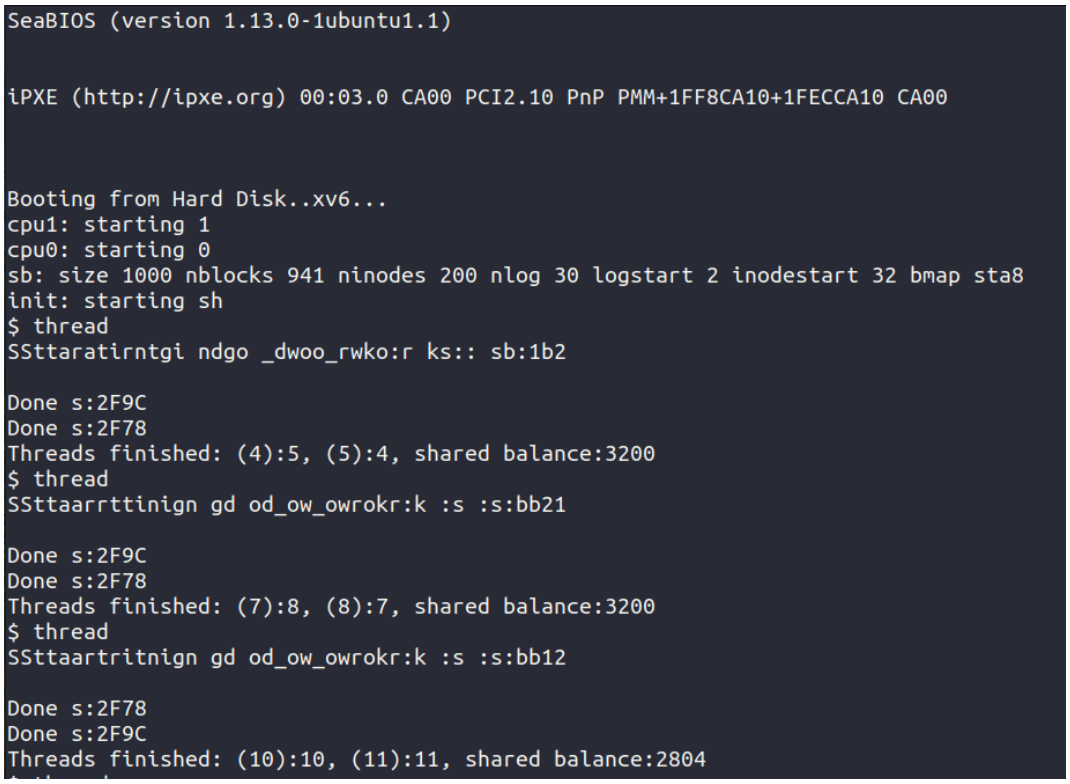
And then we added the thread call in the UPROGS and EXTRA parameters.

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

Results:

Now when we run “make clean” and “make qemu-nox”, and then run “thread” multiple times, we see these results.



```
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 sta8
init: starting sh
$ thread
SSttaratirntgi ndgo _dwoo_rwko:r ks:: sb:1b2

Done s:2F9C
Done s:2F78
Threads finished: (4):5, (5):4, shared balance:3200
$ thread
SSttaarrttinign gd od_ow_owrokr:k :s :s:bb21

Done s:2F9C
Done s:2F78
Threads finished: (7):8, (8):7, shared balance:3200
$ thread
SSttaartritnign gd od_ow_owrokr:k :s :s:bb12

Done s:2F78
Done s:2F9C
Threads finished: (10):10, (11):11, shared balance:2804
```

The shared balance is different each time due to synchronization not present, which we will do in part 2.

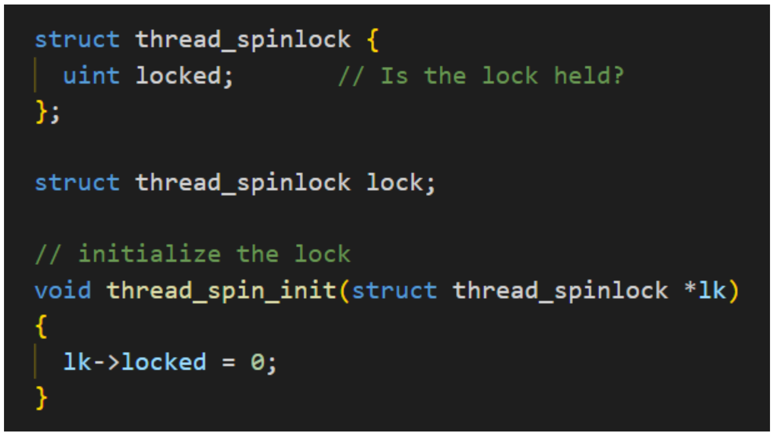
Part 2: Synchronization

We weren’t able to get the correct output for the shared balance in the last part. 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 increament of the balance) from one of the threads is lost.To fix this, we need a way to make the update instructions atomic, which we do with locks.

Spinlocks allow only one thread to access a certain section of code called the critical region. The first thread that requests for access to the critical region is granted access, but then any subsequent instructions are blocked in a while loop (in mutexes, we send the thread to sleep). Then when the thread accessing the critical region leaves the region, it changes the value of lock to allow other instructions to enter. This ensures mutual exclusion.

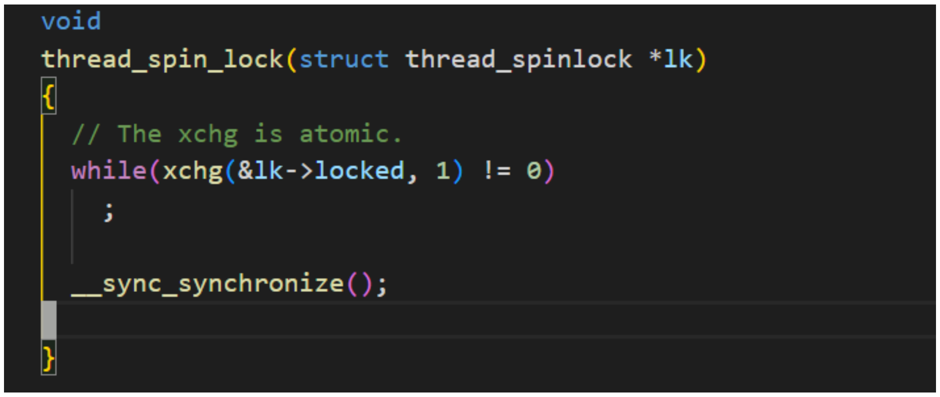
Implementing spinlocks:

First, we have the declaration of thread_spinlock struct, which contains only one field, an unsigned int denoting whether the lock is held (value 1 means lock is held by some thread, 0 otherwise). Then to allow both threads to access the same lock, we declare an instance of thread_spinlock as a global variable. We also have an initializing function that sets the locked value to 0 (the lock is not held initially)



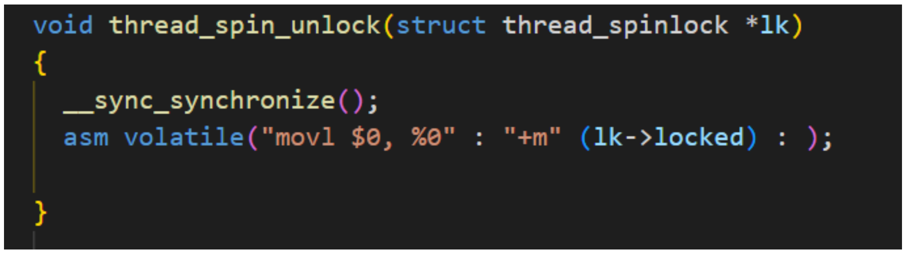
```
struct thread_spinlock {
  uint locked;       // Is the lock held?
};

struct thread_spinlock lock;

// initialize the lock
void thread_spin_init(struct thread_spinlock *lk)
{
  lk->locked = 0;
}
```

To set a lock, we use xchg instruction, which atomically exchanges 2 values, and returns the original value. Thus, calling with second value 1 sets the locks to 1, but returns its previous value, which if was 1 we know some other thread is holding the lock, else we now hold the lock. So we “spin” in the while loop till the xchg instruction returns a 0 value.



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

  __sync_synchronize();

}
```

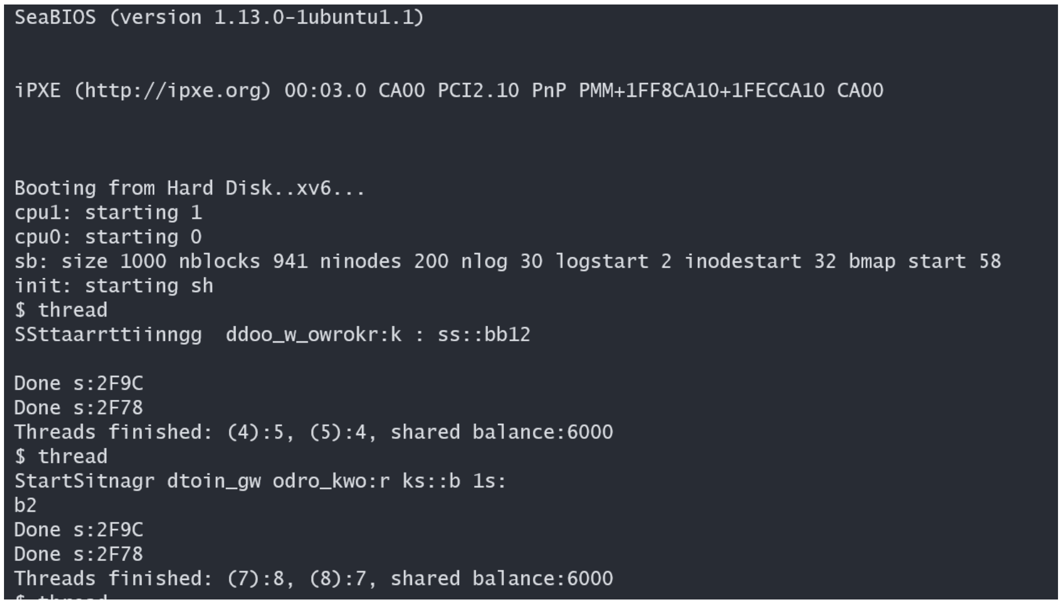
To release the lock we just use an atomic instruction to set the lock to zero.



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

}
```

The __sync_synchronize() in both of the above functions tells the compiler and the hardware not to rearrange load-store instructions, enforcing a memory barrier.

Finally running the code, here's the output.

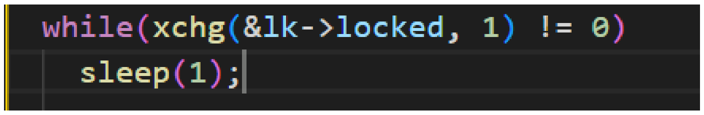


```
SeaBIOS (version 1.13.0-1ubuntu1.1)


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
SSttaarrttiinngg  ddoo_w_owrokr:k : ss::bb12

Done s:2F9C
Done s:2F78
Threads finished: (4):5, (5):4, shared balance:6000
$ thread
StartSitnagr dtoin_gw odro_kwo:r ks::b 1s:
b2
Done s:2F9C
Done s:2F78
Threads finished: (7):8, (8):7, shared balance:6000
```

As we can see, the output here is correct, i.e., exactly 3200+2800 = 6000.

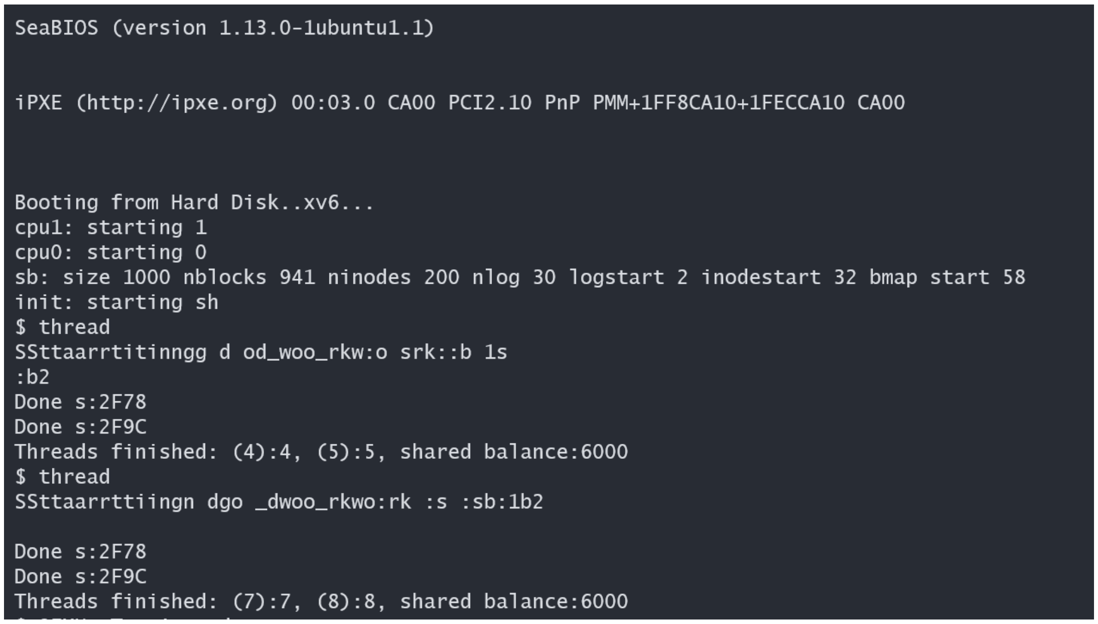
Implementing mutexes

For mutexes, the only change from spinlocks was in implementing the thread_mutex_lock function, with the xchg while loop forcing the thread to sleep rather than looping:



```
while(xchg(&lk->locked, 1) != 0)
  sleep(1);
```

Output:



```
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
SSttaarrtitinngg d od_woo_rkw:o srk::b 1s
:b2
Done s:2F78
Done s:2F9C
Threads finished: (4):4, (5):5, shared balance:6000
$ thread
SSttaarrttiingn dgo _dwoo_rkwo:rk :s :sb:1b2

Done s:2F78
Done s:2F9C
Threads finished: (7):7, (8):8, shared balance:6000
```

Again, we got the correct output of 6000 for shared balance.