ASSIGNMENT 1 - OPERATING SYSTEMS LAB CS344

Group C6 :
1. Vani Krishna Barla 200101101
2. Mahek Vora 200101062
3. Siddharth Bansal 200101093

Part 1: Kernel threads

POSIX thread libraries(thread APIs for C and C++) allow us to spawn a new current process flow. This library is used in software for faster execution.

Thread creation requires lesser overhead than forking as thread creation does not initialize new system virtual memory space and environment for the process. Further, all threads share the same address space

System calls sys_thread_create, sys_thread_join, sys_thread_exit are created in syscall.c that act as an interface for the user to implement thread_create, thread_join and thread_exit functions in proc.c respectively by taking arguments and passing it to proc.c functions.

```
int sys_thread_create(void){
  int func_add;
  int arg;
  int stack_add;

  if (argint(0, &func_add) < 0)
     return -1;
  if (argint(1, &arg) < 0)
     return -1;
  if (argint(2, &stack_add) < 0)
     return -1;

  return thread_create((void *)func_add, (void *)arg, (void *)stack_add);
}

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

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

I.thread_create()

```
struct proc {
 int is_thread;
 void *stack;
 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;     // switch() 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];               // Process name (debugging)
};
```

proc.h

- Structurally, process and thread are similar apart from the fact that threads share the same address space
- So for threads we can modify the structure "**proc**" in "**proc.h**"
    - Add a flag variable **int** "**is_thread**" to indicate if the structure is a thread (is_thread=1) or a process (is_thread=0)
    - Add a "**void* stack**" variable that points to the stack memory of the process.

```
int thread_create(void(*fcn)(void*), void *arg, void*stack){
 int i, pid;
 struct proc *np;
 struct proc *curproc = myproc();
 // int *myarg;
 // int *myret;
 // Allocate process.
 if((np = allocproc()) == 0){
   return -1;
 }

 np->sz = curproc->sz;
 np->parent = curproc;
```

```
 np->pgdir = curproc->pgdir;
 *np->tf = *curproc->tf;
 np->is_thread = 1;
 np->tf->eax = 0;
 np->tf->eip = (int) fcn; // instruction pointer
 np->stack = stack;
 np->tf->esp = (uint)stack + 4092;  // stack pointer
 *((uint *)(np->tf->esp)) = (uint)arg;
 *((uint *)(np->tf->esp - 4)) = 0xFFFFFFFF;
 np->tf->esp -= 4;
 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;
}
```

proc.c
A new function called *thread_create* is added here. It functions similarly to fork (which creates a child process).
Working:

- **allocproc()** looks into the process table for “unused proc” and if found, the state is changed to “EMBRYO”, if not found, 0 is returned by allocproc() and thread_create() returns -1.
- **sz**: size of process/thread memory = same as parent process as they share the address space
- **parent**: is set to current process
- The thread will have the same process state as the current process. So “pgdir” (i.e. mapping in physical and virtual addresses) will be the same
- **tf**: (trap frame, it holds the userspace state) will be the same as that of the current process
- **Is_thread**: set to true
- **eip:** The instruction pointer is set to the function which is taken as an input parameter of the function as required
- **stack**: the current process’s stack variable is initialized using the pointer taken as an input parameter of the function. As the esp points to the top of the stack, and 2

locations(the size of int*) are reserved, taking the page size as 4096, the esp should point at **starting_location + 4096 - 2*(sizeof(int*))**
that is, **(uint)stack + 4092**

- **esp:** (stack pointer) 2 locations in the stack are reserved for arguments and the fake address respectively
- **eax register:** is cleared
- **Open files:** of the current process are duplicated for the newly created thread
- **Current working directory** of the thread is set to the cwd of the parent process, thus bumping up the reference count of parent's cwd by calling idup().
- **Process name:** will also remain same for the new thread.
- The process state is set to RUNNABLE and this is done in the critical section as we do not want any context switches during the execution of this set of instructions
- It returns the new thread's pid.

II. thread_join()

```
int thread_join(void){
 struct proc *p;
 int havekids, pid;
 struct proc *curproc = myproc();
 acquire(&ptable.lock);
 for (;;) {
   havekids = 0;
   for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
     if (p->parent != curproc || p->is_thread != 1)
       continue;
     havekids = 1;
     if (p->state == ZOMBIE) {
       pid = p->pid;
       kfree(p->kstack);
       p->kstack = 0;
       p->state = UNUSED;
       p->pid = 0;
       p->parent = 0;
       p->name[0] = 0;
       p->killed = 0;
       release(&ptable.lock);
       return pid;
     }
   }

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

```
      release(&ptable.lock);
      return -1;
    }
    sleep(curproc, &ptable.lock);
  }
}
```

proc.c

- *thread_join* function waits for a spawned thread to exit and return its pid. If there are no spawned threads, it returns -1.
- It first checks if any process is its spawned thread (i.e. it is a child of the current process and is a thread).
- If the process state is ZOMBIE, perform cleanup and return. If a thread exists but is not in ZOMBIE state, the process goes to sleep state.
- The function thread_join() code is referenced from the function wait(). The code for which is mostly similar to wait except for two things.
    - It checks p->is_thread == 1 along with making sure the process p is the child of the current process, since we want to distinguish the thread process.
    - It doesn't free up the page directory of the thread as it is shared across all the threads of the process as well as the parent process.

III. thread_exit()

```
int thread_exit(void){
  struct proc *p;
  struct proc *curproc = myproc();
  int fd;

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

  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");
}
```

proc.c

A new process called *thread_exit* is added, it is structurally similar to exit

Working:

- The process that is exitting must not be the initproc
- Close all the open files for the current thread and setting “ofile” to 0 indicates we are no longer accessing the file
- Call wakeup1 on the parent process that might be sleeping
- Set the current process state to “ZOMBIE" and call sched, at this point it holds the ptable lock

Part 2: Synchronization

Synchronization error can be avoided by surrounding the atomic section of the code with locks. Here the atomic section in **do_work()** function:

```
    old = total_balance;
    delay(100000);
    total_balance = old + 1;
```

which is accessing a variable and updating its value, a delay is added to increase chances of contact-switching between threads accessing the same variable and updating it unsynchronized.

I. Spinlocks

Creating a struct in C to define the structure of lock which contains two variables as given below, locked signifies the state of the lock initialized and name is the name of the lock added for debugging purposes to distinguish between locks.

```
struct spinlock {
 uint locked;       // Is the lock held?
 char *name;        // Name of lock.
};
```

The init function initializes the lock's value of locked (*lk->locked*) to 0 signifying this lock is not yet locked/ is available for use.

```
void
thread_spin_init(struct spinlock *lk, char *name)
{
 lk->name = name;
 lk->locked = 0;
}
```

The *thread_spin_lock*() function is used to acquire the lock only if the lock is available (i.e. *lk->locked* value is 0) and to show it is acquired and is no longer available, the function sets the value of *lk->locked* = 1.

This is achieved by calling the **atomic function xchg(&lk->locked, 1)** which assigns the value of locked to 1 and returns the *original value* of locked. If the returned value *is not equal to* 0 signifies that it was already 1 before the function call, i.e. the lock has been acquired by some other thread.

Thus for this thread, the function xchg is called in a while loop until the lock is available, by which the original value of *lk->locked* returned would be 0 (*busy waiting*). Once the thread comes out of the while loop, it would have acquired the lock (the current value of *lk->locked* set to 1).

```
// Acquire the lock.
void
thread_spin_lock(struct spinlock *lk)
{
 while(xchg(&lk->locked, 1) != 0)
   ;
 __sync_synchronize();
}
```

While releasing the acquired lock, the value of lk->locked is set back to 0. This code segment uses asm (*inline assembly code* where volatile prevents an asm instruction from being deleted) instead of C assignment to make sure the value assignment operation is *atomic*.

```
// Release the lock.
void
thread_spin_unlock(struct spinlock *lk)
{
 __sync_synchronize();
 asm volatile("movl $0, %0" : "+m" (lk->locked) : );

}
```

The function __*sync_synchronize* is an atomic builtin used to force a full memory barrier. A full memory barrier prevents memory operations from being moved across it during compiler code optimization, neither forward nor backward.
It also stops the cpu from scheduling reads and writes, issued before the instruction, after the instruction and those issued after the instruction from being scheduled before it.
__sync_synchronize is used here to prevent the memory operations in the critical section from being scheduled before acquiring the lock/ after releasing the lock

II. Mutex Locks

Structure for mutex lock is quite similar to the *spinlock* where we use an integer locked to signify the state of the mutex lock (0 if not acquired by any process and 1 if acquired by a process/thread)

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

Initializing the mutex lock by setting the locked value to 0, signifies that lock is available to be acquired.

```
void thread_mutex_init(struct thread_mutex *m){
```

```
   m->locked = 0;
}
```

Acquiring the lock if it available (that is, **xchg(&m->locked,1)** returns 0) if not then calls **sleep(1)**.
The *sleep*() function shall cause the calling thread to be suspended from execution until the number of real time seconds specified by the argument *seconds* has elapsed.
This avoids the *busy waiting state* and allows the CPU to suspend the current thread and thus allowing the processor to be used for executing some other process/thread. This also avoids the scenario where all threads of the process start to spin endlessly waiting for lock to be released.

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

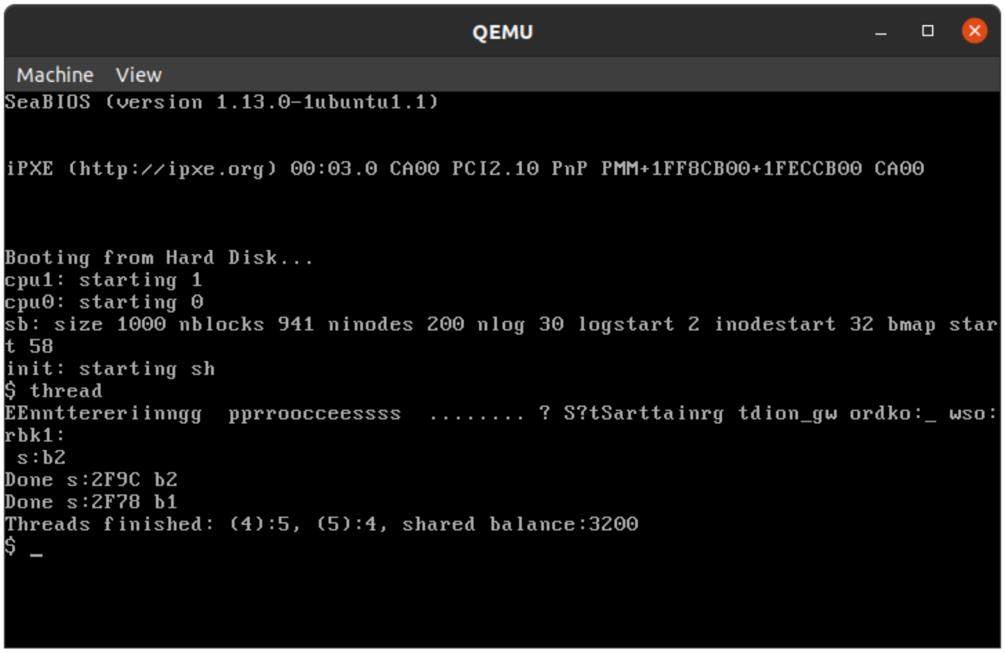
The mutex lock is unlocked similarly to the spinlock by setting the value of *m->locked* to 0 using an inline assembly command to make sure it's an atomic operation.
The utilization of __*sync_synchronize() function* is similar to that mentioned above.

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

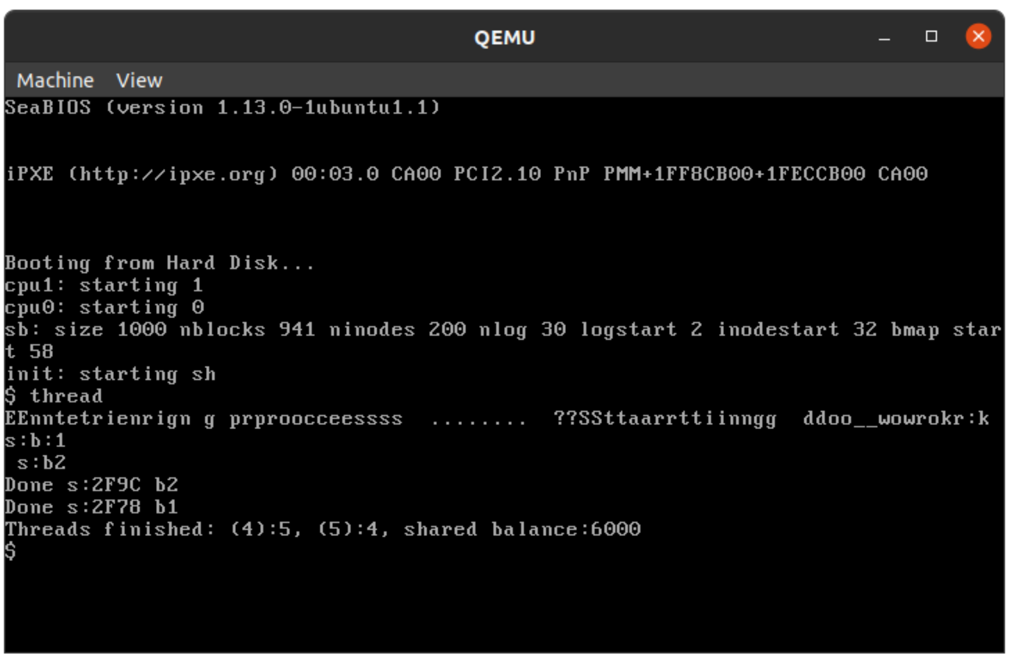
Note: *pushcli()* and *popcli()* functions used in spinlock implementation of xv6 (spinlock.c/h) from which the above code of locks is referenced from is not used because they are for disabling and enabling interrupts at OS / kernel level.
Whereas the spinlock and mutex lock implemented by us is used at user level thus these locks won't be used by interrupts and scenarios of deadlock can’t arise.

Implementing locks and testing with *thread.c*

I. **With no locks implemented** - since there is no synchronization, the balance value is being overwritten, thus the final total balance is not the correct value.

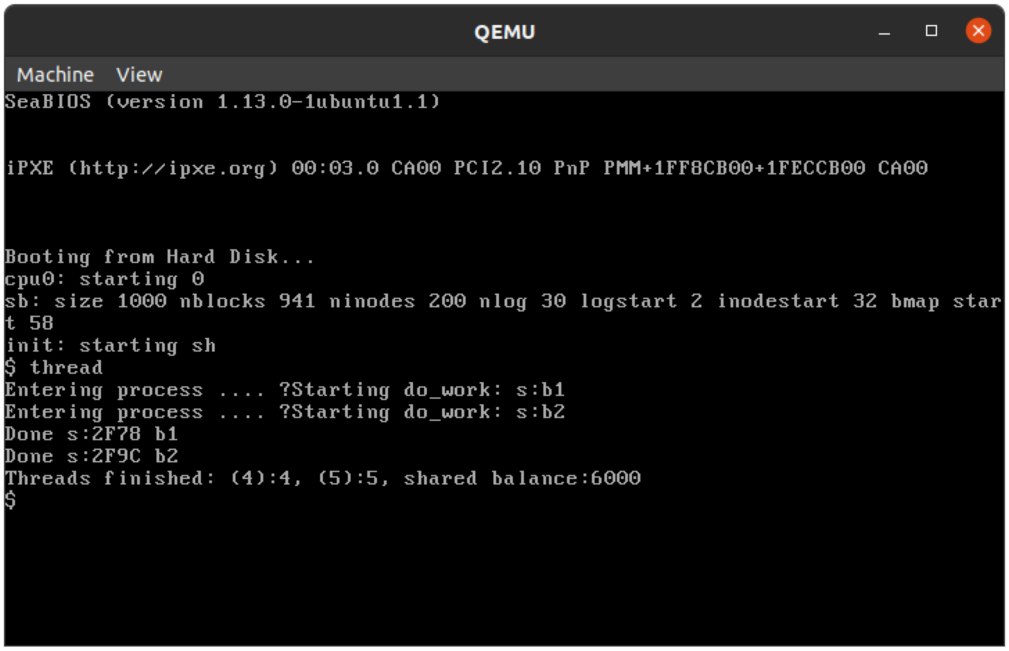


II. **Spinlock implemented with 2 processors** - since locks are implemented, the two threads do not enter the critical section at the same time and hence do not overwrite the balance while the other process is trying to increment the balance and thus the final total balance is the correct value(6000 = 3200 + 2800).



The output balance remains the same in the implementation of both spin lock and mutex lock as the critical section of the do_work function is not significantly time consuming. As this is the only process running on the os, the processors are not under heavy load, the efficiency of mutex locks over spin locks is not noticeable in this test case.

III. Mutex lock implemented with 1 processor



IV. Mutex lock with 2 processors