OS Lab Assignment-1 (CS344)

Aditya Pandey (200123004)

Aman Bucha (200123006)

Sarthak Diwan (200123076)

Lab Group No: M20

Question-1

1. Created 3 syscalls, `thread_create`, `thread_join`, `thread_exit` .

   Edited these files:

   a. syscall.c

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

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

   b. syscall.h

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

   c. sysproc.c

```c
int
sys_thread_create(void){
  int fcn;
  char* arg;
  char* stack;
  if(argint(0, &fcn) < 0) return -1;
  if(argint(1, (int*)&arg) < 0) return -1; //changed
  if(argint(2, (int*)&stack)<0) return -1; //changed

  return thread_create((void(*)(void*))fcn, arg, stack);
}
```

```c
int sys_thread_join(void)
{
  int ret_val = thread_join();
  return ret_val;
}

int sys_thread_exit(void){
  thread_exit();
  return 0;
}
```

d. defs.h

```c
int             wait(void);
void            wakeup(void*);
void            yield(void);
int             thread_create(void(*)(void*), void*, void*);
int             thread_join(void);
void            thread_exit(void);
```

e. user.h

```c
int sleep(int);
int uptime(void);
int thread_create(void(*)(void*), void*, void*);
int thread_join(void);
void thread_exit(void);
```

2. Implemented `thread_create` , `thread_join` and `thread_exit` in `proc.c`

Implementation:

int sys_thread_create(void)

It takes the 3 arguments (`fcn` , `arg` , `stack`) from the stack of the current process. Here `fcn` is the address of the function to execute. `arg` is the argument that will be passed to the function `fcn` . `stack` is the address of a memory region which will be used as **user stack** for the thread.

After taking the arguments it calls `thread_create` which is defined in `proc.c` .

---

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

It creates a new process by calling `allocproc` .

`allocproc` first acquires the lock for the `ptable` (the table which contains all the processes, defined as an array with 64 enteries). It then searches the table for a process with state `UNUSED` , then changing the state to `EMBRYO` (meaning that stack is not yet allocated). It then sets up the kernel stack by allocating a free page using `kalloc` and then setting up space for `trapframe` , `trapret` and `context` .

In `trapframe` **userspace registers** are saved when the cpu changes from user mode to kernel mode. It will be loaded after `forkret` which is executed hen the process starts executing.

`trapret` pops all the registers and save them.

`context` contains registers at the time of context switching.

Now in `thread_create` the page table of the new process(thread) is set as the same of the parent process. Now we set the `trapframe` of the new process.

`ESP` is the extended stack pointer which contains the address to the bottom of the stack. `stack` contains the address to the top of the memory allocated by `malloc` . Therefore, we set `esp` to `stack + 4096` . (4096 is the size of memory allocated).

`EIP` is the extended instruction pointer, which points to the next instruction to be executed. We set it to the address of the fuction `fcn` .

Now we pushed `arg` and `0xFFFFFFFF` (the return address) to the stack.

Lastly, the `cwd` (current working directory) and `name` are copied from the parent process and the process state is set to `RUNNABLE` (which means it can be scheduled by the scheduler). The ptable lock is released and `pid` is returned.

```
int
thread_create(void(*fcn)(void*), void* arg, void* stack)
{
  // how to pass argument in
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  /*
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
```

```
    np->state = UNUSED;
    return -1;
  }
  */

  // this portion is particularly different
  // alloting the page directory
  np->pgdir = curproc->pgdir;
  // alloting size of current process address space
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // setting the instruction pointer to the addresss the passed function fcn
  np->tf->eax = 0;
  np->tf->eip = (uint)fcn;
  // pointing the stack pointer to the end of the page alloted to it
  np->tf->esp = (uint)stack+4096;

  // similar to the fork function the child thread process also returns 0 to the parent thread/ process
  np->tf->eax = 0;
  // defining pop operation by traversing the memory
  np->tf->esp -= 4;
  *(uint*)(np->tf->esp) = (uint)(arg);

  while(np->sz != curproc->sz)
  {
      np->tf->esp -= 4;
    *(uint*)(np->tf->esp) = (uint)(arg);
      np->sz = curproc->sz;
    np->parent = curproc;
    *np->tf = *curproc->tf;
    if(!wait()) break;
  }

  // giuving a fake return address tp the function
  np->tf->esp -= 4;
  *(uint*)(np->tf->esp) = (uint)0xFFFFFFF;


  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;

}
```

---

int sys_thread_join(void)

It calls `thread_join` which is defined in `proc.c` .

---

int thread_join(void)

It first acquires the `ptable.lock` . Then it scans the process table to find the threads of the current process. For a process to be thread of the current process, its `parent` should be the cuurent process **and** address of page directory of both the processes should be same. If such a process is not found then it returns `-1` . If a process is found with state `ZOMBIE` then it makes the process `UNUSED` and returns its pid. If the thread exists and not in `ZOMBIE` state, it waits for it to complete and get in the `ZOMBIE` state.

```
int
thread_join(void)
{
  struct proc *p;
  int havechild, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  while(1){
    // Scan through table looking for exited children.
    havechild = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc || p->pgdir != curproc->pgdir) // waiting for only threads and not forked processes to end
      {
        continue;
      }
      havechild = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        //freevm(p->pgdir);
        p->parent = 0;
        p->pid = 0;
        p->state = UNUSED;
        p->name[0] = 0;
        p->killed = 0;
        release(&ptable.lock);
        return pid;
      }
    }

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

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

---

int sys_thread_exit(void)

It calls `thread_exit` which is defined in `proc.c` .

---

int thread_exit(void)

It gets the current process using `myproc` . It checks and panics if the current process is the `initproc` which is the first process which runs when `xv6` starts. Further it closes all the open files and resets the `curproc` . If parent does not exist, then control is passed to `initproc` . Finally the process state is set to `ZOMBIE` and scheduler is called.

```
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");
}
```

Result:

We can see that without using any mechanism like spinlocks/mutex the result (**3394**) is not the expected result (**6000**).

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


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



ls             2 10 17756
mkdir          2 11 15368
rm             2 12 15348
sh             2 13 27992
stressfs       2 14 16260
usertests      2 15 67368
wc             2 16 17124
zombie         2 17 14940
Drawtest       2 18 15676
thread         2 19 19484
console        3 20 0
$ thread
SSttarartitnign gd od_ow_owrokr:k :s: bs1:
b2
Done s:b2
Done s:b1
Threads finished: (5):6, (6):5, shared balance:3394
$
```

Question-2

For this question, we referred to the following lines in `spinlock.c` :

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if (holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while (xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}

// Release the lock.S
void release(struct spinlock *lk)
```

```
{
  if (!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both not to.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0"
               : "+m"(lk->locked)
               :);

  popcli();
}
```

We tried to create a separate file for spinlocks and mutexes, wherein we had defined all the functions. However, we faced issues while including the c files, so we instead declared all the functions in `thread.c`

(a) Spinlocks

We defined the structure `thread_spinlock` as

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

We implemented the following functions:

void thread_spin_init(struct thread_spinlock *lk)

This function initializes the value of `lk->locked` to 0. This means the lock has not been acquired yet by any process.

---

void thread_spin_lock(struct thread_spinlock *lk)

This function waits for the lock to be freed. Implementation is done similar to the `acquire` function in `spinlock.c`. We first wait till the lock is acquired. Then, we call `__sync_synchronize`. This function tells CPU not to move loads or stores. This ensures critical section's memory references happen after the lock is acquired.

---

void thread_spin_unlock(struct thread_spinlock *lk)

We first call `__sync_synchronize`. Then we change the value of `lk->locked` to 0 (i.e. we release the lock). It was done using assembly language instruction to make it atomic.

Assembly Language Instruction: `asm volatile("movl $0, %0" : "+m" (lk->locked) : );not`

Result:

Now we are getting the expected result.

The final value of the total_balance matches the expected 6000, i.e., the sum of individual
balances of each thread. This means that only one thread was allowed to enter the critical section at a time.

So implementation of spinlocks was done correctly.

```
console        3 20 0
$ thread
SSttaarrttiinngg  ddoo__wworokr:k :s :sb:1
b2
Done s:b2
Done s:b1
Threads finished: (5):6, (6):5, shared balance:6000
$
```

(b) Mutexes

Implementation is done similar to spinlocks. The only difference is that we use the function `sleep` inside the `while` loop of `mutex_lock()` function.

We defined structure `mutex_lock` as

```
struct mutex_lock{
  uint locked;
};
```

We implemented the following functions:

void thread_mutex_init(struct thread_mutex *m)

This function initializes the value of `m->locked` to 0. This means the lock has not been acquired yet by any process.

---

void thread_mutex_lock(struct thread_mutex *m)

Similar to `thread_spin_lock` function, the difference being the use of `sleep` function inside the `while` loop. We use the sleep function because this releases CPU to another thread.

---

void thread_mutex_unlock(struct thread_mutex *m)

Function definition is exactly the same as that of `thread_spin_unlock` function, the only difference being the function parameter is `m` pointer not `lk` pointer.

Result:

Here too, we get the expected result.

The final value of the total_balance matches the expected 6000, i.e., the sum of individual
balances of each thread. This means that only one thread was allowed to enter the critical section at a time.

So implementation of mutex locks was done correctly.

```
init: starting sh
$ thread
SStatratirntg idon_gw ordko: _sw:orkb1:
 s:b2
Done s:b1
Done s:b2
Threads finished: (4):4, (5):5, shared balance:6000
```