University of California, Riverside
Department of Computer Science and Engineering

Title: Lab 2 – XV6 Threads

Course: Advanced Operation Systems

Mahbod Afarin

862186340

Spring 2021

1. Lab Overview

For the purpose of this lab, I am going to add the real kernel threads to xv6. I will divide the description of this lab into three parts. First of all, I defined a new system call in order to create the kernel threads and named it *clone* (). Second, I used *clone* () system call to create a little thread library. Finally, using a test program, I will show that all of the things work well.

2. Defining *clone* () System Call

In this part I will talk about the details of defining the new system call called *clone* (). I will use this system call to create thread kernels and it will be similar to fork (), but there will be some differences between *clone* () and *fork* (). The fist difference will be the address space. The address space of should be shared between the parents and child, so instead of creating a new address space, it should use the parent's address space. The second difference should be the file descriptor. The file descriptor of the child should be the same file descriptor for parents. Therefore, the parents and child should not have duplicate file descriptor. Third, in *clone* (), I make sure that when I am return, I am not running on the parent's stack. The *clone* () system call will return PID of the child to parent. In listing 1, you can see the implementation of the *clone* ().

Listing 1: Implementation of the clone ()

```
1   int clone(void* stack,int size)
2   {
3     int i,pid;
4     struct proc *p;
5     if((p = allocproc()) == 0)
6     {
7       return -1;
8     }
9     struct proc *curproc = myproc();
10    p->sz = curproc->sz;
11    p->parent = curproc;
12    *p->tf = *curproc->tf;
13    p->thread = 1;
14    p->pgdir = curproc->pgdir;
15    p->tf->eax = 0;
16    ThreadNum++;
17    void *var1 = (void*)curproc->tf->ebp +16;
18    void *var2 = (void*)curproc->tf->esp;
19    uint sz1 = (uint)(var1 - var2);
20    p->tf->esp = (uint) (stack - sz1);
21    p->tf->ebp = (uint) (stack - 16);
22    memmove(stack-sz1,var2,sz1);
23    for(i=0;i<NOFILE;i++)
24    {
25      if(curproc->ofile[i])
26      {
27        p->ofile[i]=filedup(curproc->ofile[i]);
28      }
```

```
    }
    p->cwd = idup(curproc->cwd);
    pid = p->pid;
    p->state = RUNNABLE;
    safestrcpy(p->name, curproc->name, sizeof(curproc->name));

    return pid;
  }
```

We should do some changes in *exit*() and *wait*() calls. Previously, when we call *exit*() system call, it will simply close all the file descriptors which the process use. But now, the file descriptors and shared between threads of the kernel process and we cannot close all of that. For solving that problem, I keep track of number of threads that shared the same address space, and I will free the resources only when the last thread exit. Also, we can use this trick for *wait*() system call. In listing 2, I showed the thread_*exit*() system call.

Listing 2: Implementation of the thread_exit ()

```
void thread_exit()
{
  if(ThreadNum == 1)
  {
    struct proc *curproc = myproc();
    struct proc *p;
    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);
    wakeup1(curproc->parent);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if(p->parent == curproc)
      {
        p->parent = initproc;
        if(p->state == ZOMBIE)
        wakeup1(initproc);
      }
    }
```

```
36      curproc->state = ZOMBIE;
37      sched();
38      panic("zombie exit");
39    } else {
40      ThreadNum--;
41    }
42  }
```

I also had some little modifications in *def.h*, *proc.h*, *Makefile*, *syscall.c*, *sysproc.c*, *user.h*, and *syscall.h*. In the *def.h* we need to add *int clone(void *, int);* and *void threadexit();*. In the *proc.h* we need to add *int thread;*. In the *Makefile* we need add *thread.c* and *frisbee.c*. In the *syscall.c* we need to add the following:

Listing 3: Changes in the csyscall.c

```
1  extern int sys_clone(void);
2  extern int sys_threadexit(void);
3  [SYS_clone] sys_clone,
4  [SYS_threadexit] sys_threadexit,
```

We need the following changes in the *sysproc.c*:

Listing 4: Changes in the sysproc.c

```
1  int sys_clone(void)
2  {
3      void *st;
4    int sz;
5      if(argint(1, &sz) < 0)
6          return -1;
7      if(argptr(0, (char **)&st, sz) < 0)
8          return -1;
9      return clone(st, sz);
10 }
11 int sys_threadexit(void)
12 {
13     threadexit();
14     return 0;
15 }
```

3. Building Thread Library

In this part, I am going to describe the process of building a little thread library. The thread library will be built on top of this. Therefore, I create *thread_creat*() for this purpose. This routine will use *clone*() in order to create the child. Also, my thread library implements spin lock. I used *lock_t* type to declare a lock and there are two routines named *lock_acquire* (*lock_t **) and *lock_release*(*lock_t **) which can quire and release the lock for us. For initializing the lock, I used *lock_init*(*lock_t **).

Listing 5: Implementation of the thread.c

```
1  #include "types.h"
```

```
#include "user.h"
#include "mmu.h"
#include "spinlock.h"
#include "x86.h"

struct lock_t
{
    uint lock;
};

void thread_create(void*(*sr)(void*), void *arg)
{
    void* np = malloc(PGSIZE*2);
    int tp;
    tp = clone(np, PGSIZE*2);
    if(tp==0)
    {
        (*sr)(arg);
        exit();
    }
}

void lock_init(struct lock_t *lk)
{
    lk -> lock = 0;
}
void lock_acquire(struct lock_t *lk)
{
    while(xchg(&lk->lock,1) != 0);
}
void lock_release(struct lock_t *lock)
{
    xchg(&lock->lock,0);
}
```

4. Testing My Code

For testing my code, I used a simple program to create some number of threads for me using *thread_create*(). The threads are simulating the frisbee game for us where each thread passes the frisbee to the next thread. Each thread spins will check the value of the lock in the location of the frisbee and if it is its turn, it will print a message and then release the lock. In the below listing you can see the implementation of the frisbee game.

Listing 6: Implementation of the frisbee game

```
#include "types.h"
#include "user.h"
#include "stat.h"
#include "thread.h"

struct lock_t *lock;
int round = 0, j = 0;
int ThreadNum, PassNum;
```

```
9
10  void* PassThreads(void *k)
11  {
12      int i = (int)k;
13      while(round < PassNum)
14      {
15          lock_acquire(lock);
16          if(round == PassNum) {break;}
17          if(i == j)
18          {
19              round++;
20              printf(1, "Pass number no: %d, Thread %d is passing the token to
    thread ", round, i);
21              j++;
22              if(j == ThreadNum)
23                  j = 0 ;
24              printf(1," %d\n",j);
25              lock_release(lock);
26               sleep(1);
27          }
28          else
29          {
30              lock_release(lock);
31              sleep(1);
32          }
33      }
34      printf(1,"Simulation of Frisbee game has finished, 6 rounds were played in
    total!\n", round);
35      exit();
36  }
37
38  int main(int argc, char *argv[])
39  {
40      int i = 0;
41      ThreadNum = atoi(argv[1]);
42      PassNum = atoi(argv[2]);
43      lock_init(lock);
44      for(i = 0; i<ThreadNum; i++)
45      {
46          thread_create(PassThreads, (void *)i);
47      }
48      wait();
49      exit();
50  }
```

In figure 4, we can see the results for running the frisbee game with 4 threads and 6 passes. For running the frisbee program we should specify the inputs of the program. The first input parameter is the number of threads, and the second input parameter is the number of passes. The parent process will create multiple threads which is our players in frisbee game. Each thread will access to the lock and check whether it is its turn or not. If it is its run, it will throw the frisbee and the program will assign

the id of the next thread to receive the frisbee. Then, it will increase the number of passes and then release the lock.

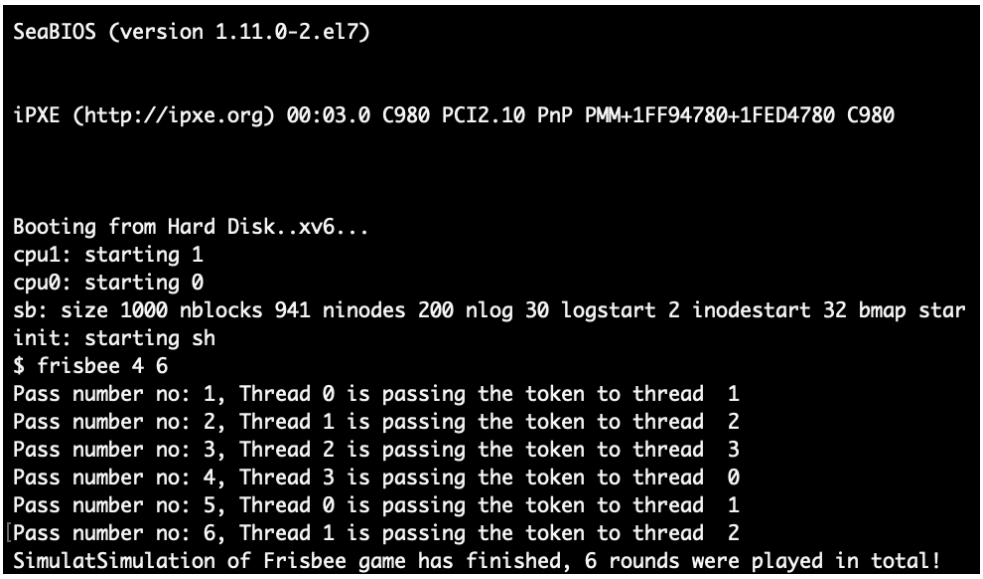

Figure 1: The result of testing frisbee with 4 threads and 6 passes

Figure 2 shows another example with 5 number of threads and 10 rounds.

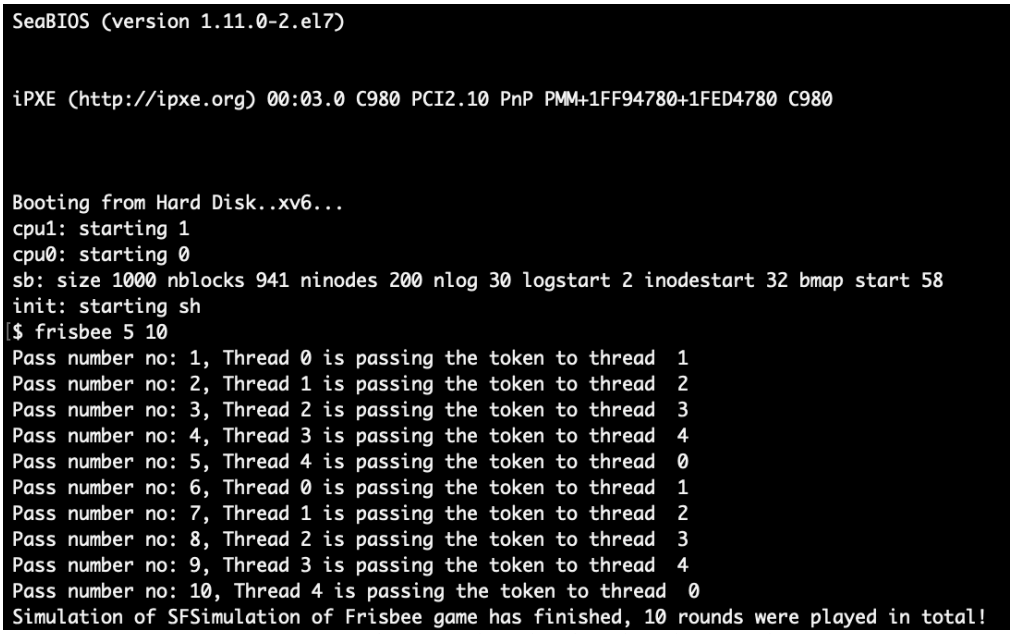

Figure 2: Another example with 5 threads and 10 rounds