Lab #3 : xv6 Threads

Group Members

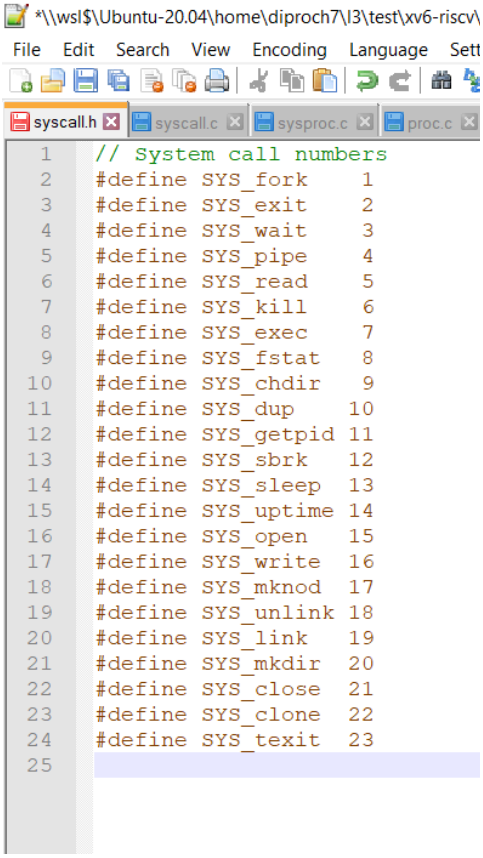
- Bhagyesh Ravindra Gaikwad (bgaik001@ucr.edu, NetID: bgaik001)
- Dipro Chakraborty (dchak006@ucr.edu, NetID: dchak006)

Objective

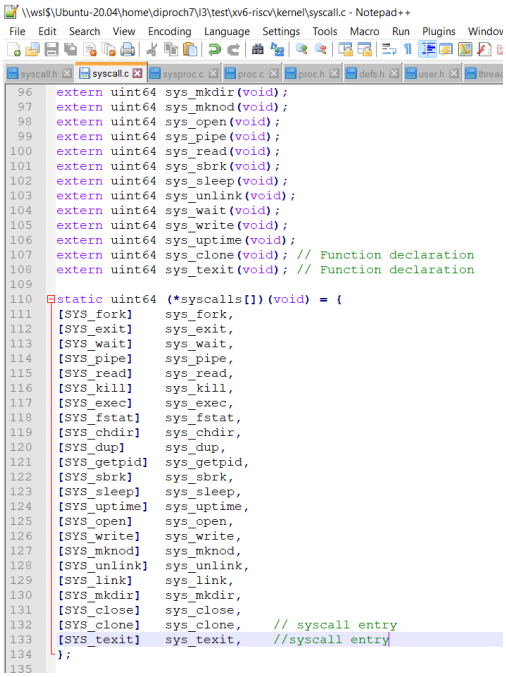
- Add kernel-level thread support to xv6
- Define a new system call *clone()* to create a kernel thread. Using *clone()* build a user-level thread library of *thread_create(), lock_acquire and lock_release()* functions.
- Build a test program, in which multiple threads are created by the parent, and each thread inserts items into a thread-safe linked list.

Files Modified

- We create two new system calls, *clone()* and *texit()* with numbers 22 and 23 in our syscall.h file, as an indication of our system calls.



- This is followed by updating the system call functions in the kernel/syscall.c file, providing the function declarations.

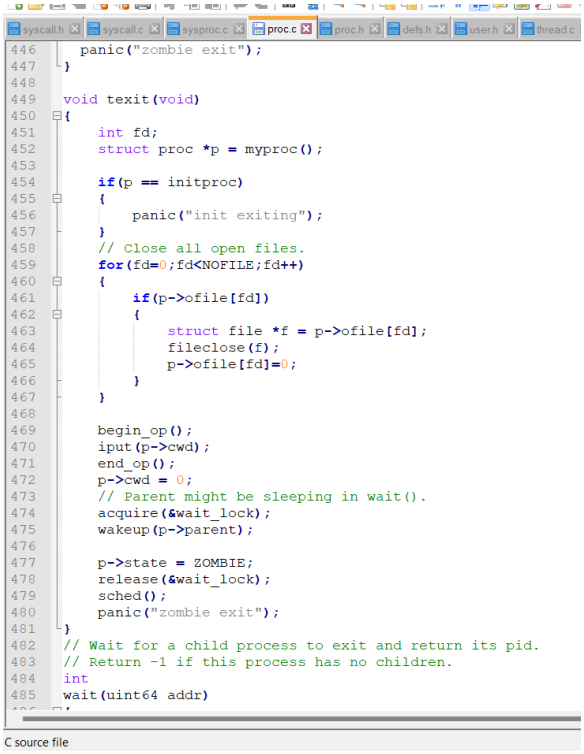


- Now, we define these syscall functions in the kernel/sysproc.c file. The explanation of the code is in the next section.

- Now we define the clone() functions and texit (modified exit) function in the kernel/proc.c file.

```
// Clone function
int clone (void *stack,int size,void*(func)(void*),void *arg)
{
    int i,pid;
    struct proc *np;
    struct proc *p = myproc();
    // Allocate process.
    if((np = allocproc()) == 0){
        return -1;
    }
    if((uint64)stack%PGSIZE != 0 || stack == 0) {
        return -1;
    }

    // share same address space with parent
    np->state = UNUSED;
    np->sz = p->sz;
    *(np->trapframe) = *(p->trapframe);
    np->pagetable = p->pagetable;

    np->context.ra = (uint64)func;
    np->trapframe->a0 = 0;

    // use the same file descriptor
    for(i=0;i<NOFILE;i++)
    {
        if(p->ofile[i])
        {
            np->ofile[i] = filedup(p->ofile[i]);
        }
        np->cwd = idup(p->cwd);
    }
    safestrcpy(np->name,p->name,sizeof(p->name));

    uint64 ustack[2];
    ustack[0] = 0xffffffff;
    ustack[1] = (uint64)arg;

    np->context.sp = (uint64)(stack+PGSIZE-4);
    *((uint64*)(np->context.sp)) = (uint64)arg;
```

```
        np->cwd = idup(p->cwd);
    }
    safestrcpy(np->name,p->name,sizeof(p->name));

    uint64 ustack[2];
    ustack[0] = 0xffffffff;
    ustack[1] = (uint64)arg;

    np->context.sp = (uint64)(stack+PGSIZE-4);
    *((uint64*)(np->context.sp)) = (uint64)arg;
    *((uint64*)(np->context.sp)-4) = 0xFFFFFFFF;
    np->context.sp = (np->context.sp) - 4;

    if(copyout(np->pagetable,np->context.sp,(char*)ustack,size)<0)
    {
        printf("Stack copy failed\n");
        return -1;
    }
    np->state = RUNNABLE;

    pid = np->pid;
    release(&np->lock);

    acquire(&wait_lock);
    np->parent = p;
    release(&wait_lock);

    acquire(&np->lock);
    np->state = RUNNABLE;
    release(&np->lock);

    return pid;


}

// Pass p's abandoned children to init.
// Caller must hold wait_lock.
void
reparent(struct proc *p)
```



```
  panic("zombie exit");
}

void texit(void)
{
    int fd;
    struct proc *p = myproc();

    if(p == initproc)
    {
        panic("init exiting");
    }
    // Close all open files.
    for(fd=0;fd<NOFILE;fd++)
    {
        if(p->ofile[fd])
        {
            struct file *f = p->ofile[fd];
            fileclose(f);
            p->ofile[fd]=0;
        }
    }

    begin_op();
    iput(p->cwd);
    end_op();
    p->cwd = 0;
    // Parent might be sleeping in wait().
    acquire(&wait_lock);
    wakeup(p->parent);

    p->state = ZOMBIE;
    release(&wait_lock);
    sched();
    panic("zombie exit");
}
// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
wait(uint64 addr)
```

- The final step in the kernel-space syscall interface is to add the clone function declaration and texit function definitions in the kernel/defs.h file.

```
78
79  // printf.c
80  void            printf(char*, ...);
81  void            panic(char*) __attribute__((noreturn));
82  void            printfinit(void);
83
84  // proc.c
85  int             cpuid(void);
86  void            exit(int);
87  int             fork(void);
88  int             growproc(int);
89  void            proc_mapstacks(pagetable_t);
90  pagetable_t     proc_pagetable(struct proc *);
91  void            proc_freepagetable(pagetable_t, uint64);
92  int             kill(int);
93  struct cpu*     mycpu(void);
94  struct cpu*     getmycpu(void);
95  struct proc*    myproc();
96  void            procinit(void);
97  void            scheduler(void) __attribute__((noreturn));
98  void            sched(void);
99  void            sleep(void*, struct spinlock*);
100 void            userinit(void);
101 int             wait(uint64);
102 void            wakeup(void*);
103 void            yield(void);
104 int             either_copyout(int user_dst, uint64 dst, void *src, uint64 len);
105 int             either_copyin(void *dst, int user_src, uint64 src, uint64 len);
106 void            procdump(void);
107 int             clone(void*,int,void*(func)(void*),void*);
108 void            texit(void);
109 // swtch.S
110 void            swtch(struct context*, struct context*);
111
112 // spinlock.c
113 void            acquire(struct spinlock*);
114 int             holding(struct spinlock*);
115 void            initlock(struct spinlock*, char*);
116 void            release(struct spinlock*);
117 void            push_off(void);
```

C++ source file

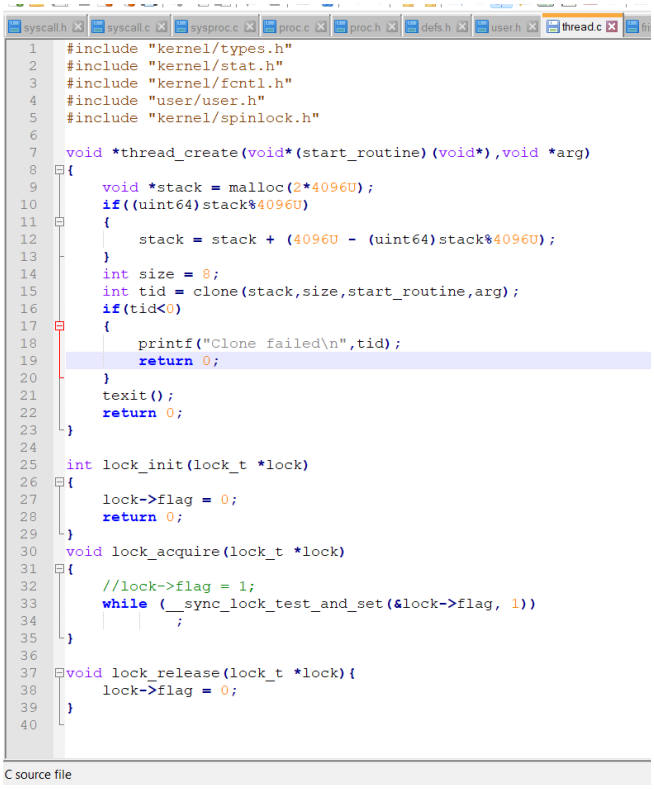
- After the kernel-space, we update the user-space syscall interface. This is done by adding the clone() and texit() entries in user/usys.pl files and also defining clone and texit functions in user/user.h. Also, we define the lock we intend to use, in user/user.h along with init_lock, lock_acquire and lock_release functions.

```
1  struct stat;
2  struct rtcdate;
3
4  struct lock_t{
5      uint64 flag;
6  };
7  typedef struct lock_t lock_t;
8
9  // system calls
10 int fork(void);
11 int exit(int) __attribute__((noreturn));
12 int wait(int*);
13 int pipe(int*);
14 int write(int, const void*, int);
15 int read(int, void*, int);
16 int close(int);
17 int kill(int);
18 int exec(char*, char**);
19 int open(const char*, int);
20 int mknod(const char*, short, short);
21 int unlink(const char*);
22 int fstat(int fd, struct stat*);
23 int link(const char*, const char*);
24 int mkdir(const char*);
25 int chdir(const char*);
26 int dup(int);
27 int getpid(void);
28 char* sbrk(int);
29 int sleep(int);
30 int uptime(void);
31 int clone(void*,int,void*(func)(void*),void*);
32 void *thread_create(void*(start_routine)(void*),void*);
33 int lock_init(lock_t *);
34 void lock_acquire(lock_t *);
35 void lock_release(lock_t *);
36 void texit(void) __attribute__((noreturn));
37
38
39
40
41 // ulib.c
```

C++ source file

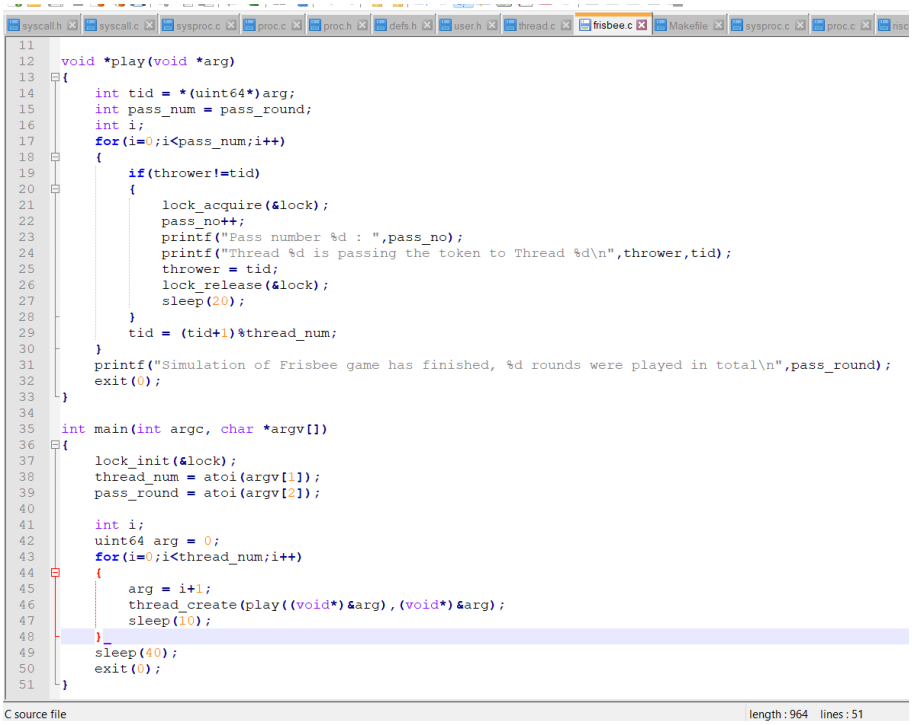
```
1  #!/usr/bin/perl -w
2
3  # Generate usys.S, the stubs for syscalls.
4
5  print "# generated by usys.pl - do not edit\n";
6
7  print "#include \"kernel/syscall.h\"\n";
8
9  sub entry {
10     my $name = shift;
11     print ".global $name\n";
12     print "${name}:\n";
13     print " li a7, SYS_${name}\n";
14     print " ecall\n";
15     print " ret\n";
16 }
17
18 entry("fork");
19 entry("exit");
20 entry("wait");
21 entry("pipe");
22 entry("read");
23 entry("write");
24 entry("close");
25 entry("kill");
26 entry("exec");
27 entry("open");
28 entry("mknod");
29 entry("unlink");
30 entry("fstat");
31 entry("link");
32 entry("mkdir");
33 entry("chdir");
34 entry("dup");
35 entry("getpid");
36 entry("sbrk");
37 entry("sleep");
38 entry("uptime");
39 entry("clone");
40 entry("texit");
41
```

- This is followed by creating the user-level thread library thread.c in the user directory.



```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "kernel/fcntl.h"
#include "user/user.h"
#include "kernel/spinlock.h"

void *thread_create(void*(start_routine)(void*),void *arg)
{
    void *stack = malloc(2*4096U);
    if((uint64)stack%4096U)
    {
        stack = stack + (4096U - (uint64)stack%4096U);
    }
    int size = 8;
    int tid = clone(stack,size,start_routine,arg);
    if(tid<0)
    {
        printf("Clone failed\n",tid);
        return 0;
    }
    texit();
    return 0;
}

int lock_init(lock_t *lock)
{
    lock->flag = 0;
    return 0;
}
void lock_acquire(lock_t *lock)
{
    //lock->flag = 1;
    while (__sync_lock_test_and_set(&lock->flag, 1))
        ;
}

void lock_release(lock_t *lock){
    lock->flag = 0;
}
```

- We implement the user-test program to test out the frisbee throw operation with threads. We create a new file, frisbee.c in the user directory.



```c
void *play(void *arg)
{
    int tid = *(uint64*)arg;
    int pass_num = pass_round;
    int i;
    for(i=0;i<pass_num;i++)
    {
        if(thrower!=tid)
        {
            lock_acquire(&lock);
            pass_no++;
            printf("Pass number %d : ",pass_no);
            printf("Thread %d is passing the token to Thread %d\n",thrower,tid);
            thrower = tid;
            lock_release(&lock);
            sleep(20);
        }
        tid = (tid+1)%thread_num;
    }
    printf("Simulation of Frisbee game has finished, %d rounds were played in total\n",pass_round);
    exit(0);
}

int main(int argc, char *argv[])
{
    lock_init(&lock);
    thread_num = atoi(argv[1]);
    pass_round = atoi(argv[2]);

    int i;
    uint64 arg = 0;
    for(i=0;i<thread_num;i++)
    {
        arg = i+1;
        thread_create(play((void*)&arg),(void*)&arg);
        sleep(10);
    }
    sleep(40);
    exit(0);
}
```

- Lastly, we add the thread.o file in user libraries (ulib) and add the frisbee program as a UPROGS entry in the Makefile.

```
89
90  ULIB = $U/ulib.o $U/usys.o $U/printf.o $U/umalloc.o $U/thread.o
91
117
118 UPROGS=\
119     $U/_cat\
120     $U/_echo\
121     $U/_forktest\
122     $U/_grep\
123     $U/_init\
124     $U/_kill\
125     $U/_ln\
126     $U/_ls\
127     $U/_mkdir\
128     $U/_rm\
129     $U/_sh\
130     $U/_stressfs\
131     $U/_usertests\
132     $U/_grind\
133     $U/_wc\
134     $U/_zombie\
135     $U/_frisbee\
136
137 fs.img: mkfs/mkfs README $(UPROGS)
138     mkfs/mkfs fs.img README $(UPROGS)
139
140 -include kernel/*.d user/*.d
141
142 clean:
143     rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
144     */*.o */*.d */*.asm */*.sym \
145     $U/initcode $U/initcode.out $K/kernel fs.img \
146     mkfs/mkfs .gdbinit \
```

Makefile

Explanation of Code

The *clone( )* function is similar in code, with the *fork( )* function. The first part of the function checks if the stack is essentially a basic sanity check for the input parameters, like if the stack is null. This is followed by sharing of the same address of the parent, by setting the parameters of the new process np, as UNUSED for the state, sz as the sz of the parent, sharing the page table of the parent and also setting the eip pointer as the function pointer, and eax pointer as 0. This is followed by sharing of the same file descriptors as the parent, and lastly setting up the user stack which is copied onto the stack pointer (sp) of the new process. It ends in setting the state of this new process as RUNNABLE and setting the parent of this new process as the proc *p, the current cpu process.

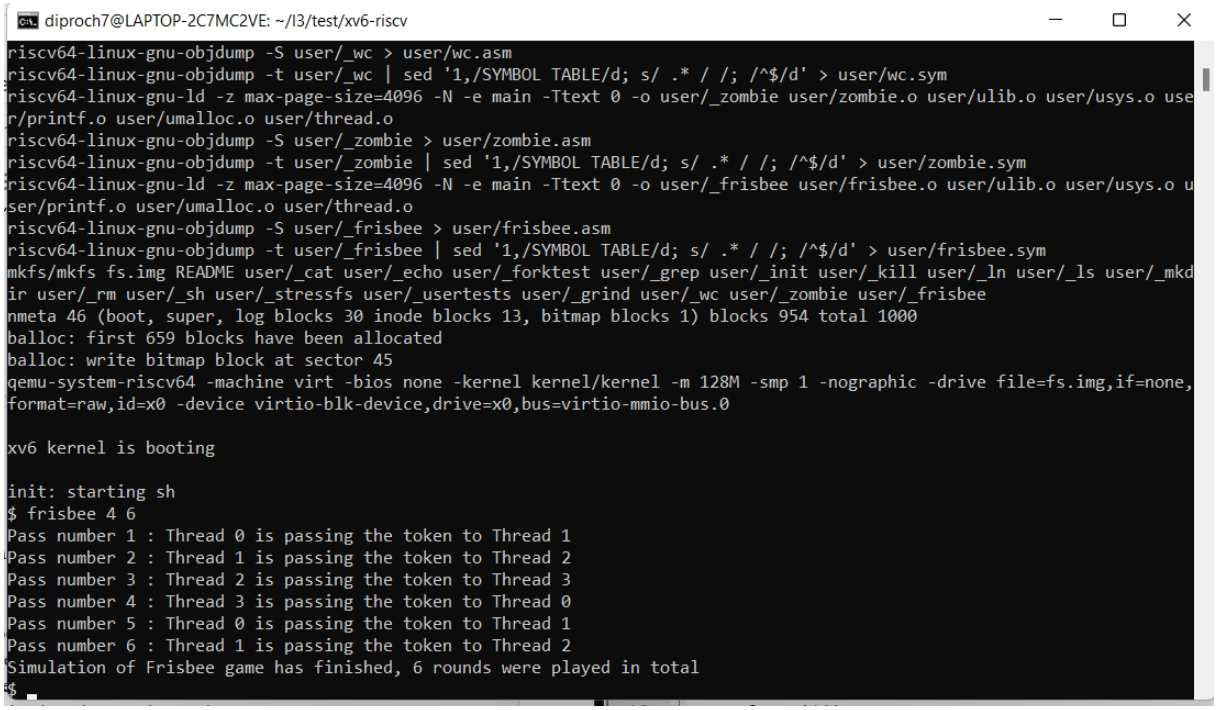
The *texit()* function is again, similar to the *exit( )* function in proc.c, as the parent and children share the same file descriptors, here we remove the reparent( ) function and only free the shared resources, when the last one exits or reaped by wait.

The thread library takes the function and arg as the argument, creates the user stack and calls the *clone( )* function, in order to create new threads. It also implements user-level spinlock routines, with the *init_lock*( ) function to initialize the lock and use *lock_acquire( )* and *lock_release( )* to acquire and

release the locks respectively. The atomic exchange is achieved in XV6-RISCV using the *sync_test_and_set* function.

The frisbee.c file is the program that simulates the frisbee throwing game, which has a *play* function and a *main* function. In the play function, it extracts the thread_id from the original argument, and passes the thread from the thrower thread to another thread, while being locked. In the main function, *thread_create* is called using the *play* function as a start routine and using iteration number as the argument.

Output Generated



Contribution of Group Members

Bhagyesh Ravindra Gaikwad

- Implementation of clone( ) and system-calls
- Screenshots of program files

Dipro Chakraborty

- Implementation of thread library, locks and frisbee test program
- Screenshots of the execution
- Recording the youtube video (https://www.youtube.com/watch?v=rzxf7oc_LV0&ab_channel=DiproChakraborty).
- Report writing- Explanation of code, Files modified and Output.