

Lab #3 : xv6 Threads

Group Members

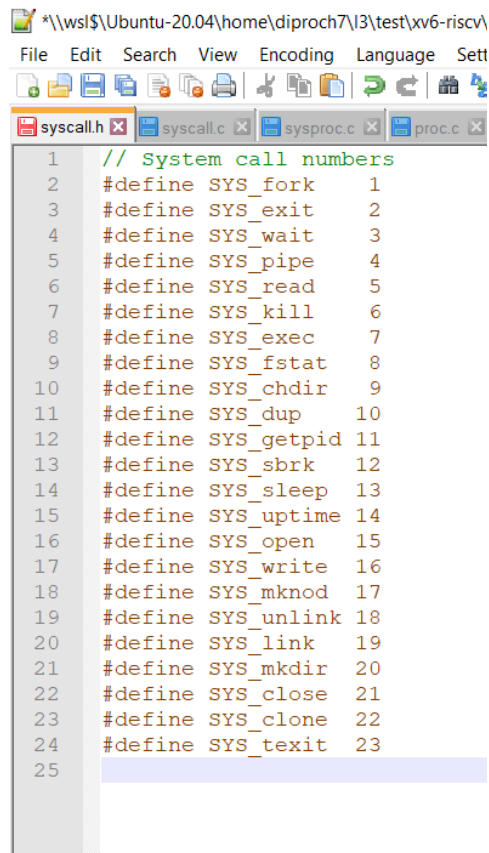
- Bhagyesh Ravindra Gaikwad (bgaik001@ucr.edu, NetID: bgaik001)
- Dipro Chakraborty (dchak006@ucr.edu, NetID: dchak006)

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.

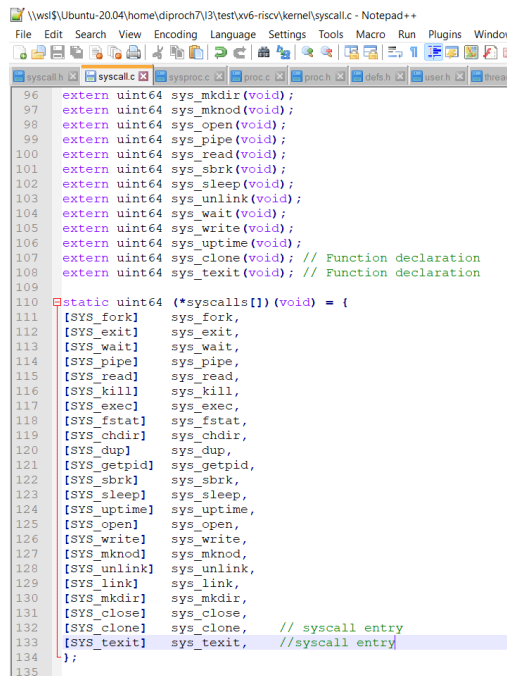
Files Modified

- 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.



```
*\\wsl$\\Ubuntu-20.04\\home\\diproch7\\l3\\test\\xv6-riscv\\
File Edit Search View Encoding Language Sett
syscall.h x syscall.c x sysproc.c x proc.c x
1 // System call numbers
2 #define SYS_fork 1
3 #define SYS_exit 2
4 #define SYS_wait 3
5 #define SYS_pipe 4
6 #define SYS_read 5
7 #define SYS_kill 6
8 #define SYS_exec 7
9 #define SYS_fstat 8
10 #define SYS_chdir 9
11 #define SYS_dup 10
12 #define SYS_getpid 11
13 #define SYS_sbrk 12
14 #define SYS_sleep 13
15 #define SYS_uptime 14
16 #define SYS_open 15
17 #define SYS_write 16
18 #define SYS_mknod 17
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_clone 22
24 #define SYS_texit 23
25
```

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



```

96 extern uint64 sys_mkdir(void);
97 extern uint64 sys_mknod(void);
98 extern uint64 sys_open(void);
99 extern uint64 sys_pipe(void);
100 extern uint64 sys_read(void);
101 extern uint64 sys_sbrk(void);
102 extern uint64 sys_sleep(void);
103 extern uint64 sys_unlink(void);
104 extern uint64 sys_wait(void);
105 extern uint64 sys_write(void);
106 extern uint64 sys_uptime(void);
107 extern uint64 sys_clone(void); // Function declaration
108 extern uint64 sys_exit(void); // Function declaration
109
110 static uint64 (*syscalls[]) (void) = {
111     [SYS_fork] sys_fork,
112     [SYS_exit] sys_exit,
113     [SYS_wait] sys_wait,
114     [SYS_pipe] sys_pipe,
115     [SYS_read] sys_read,
116     [SYS_kill] sys_kill,
117     [SYS_exec] sys_exec,
118     [SYS_fstat] sys_fstat,
119     [SYS_chdir] sys_chdir,
120     [SYS_dup] sys_dup,
121     [SYS_getpid] sys_getpid,
122     [SYS_sbrk] sys_sbrk,
123     [SYS_sleep] sys_sleep,
124     [SYS_uptime] sys_uptime,
125     [SYS_open] sys_open,
126     [SYS_write] sys_write,
127     [SYS_mknod] sys_mknod,
128     [SYS_unlink] sys_unlink,
129     [SYS_link] sys_link,
130     [SYS_mkdir] sys_mkdir,
131     [SYS_close] sys_close,
132     [SYS_clone] sys_clone, // syscall entry
133     [SYS_exit] sys_exit, //syscall entry
134 };
135

```

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



```

96 return xticks;
97 }
98
99 uint64
100 sys_clone(void)
101 {
102     void *stack;
103     int size;
104     void *func;
105     void *arg;
106
107     if(argstr(0, (char*) (&stack), sizeof(stack)) < 0)
108     {
109         return -1;
110     }
111     if(argint(1, &size) < 0)
112     {
113         return -1;
114     }
115     if(argstr(2, (char*) (&func), sizeof(func)) < 0)
116     {
117         return -1;
118     }
119     if(argstr(3, (char*) (&arg), sizeof(arg)) < 0)
120     {
121         return -1;
122     }
123
124     return clone((void*) stack, size, (void*) func, (void*) arg);
125 }
126
127
128 uint64
129 sys_exit(void)
130 {
131     _exit(0);
132     return 0;
133 }
134
135
136

```

C source file

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

```

323 // Clone function
324 int clone(void *stack, int size, void * (func) (void *), void *arg)
325 {
326     int i, pid;
327     struct proc *np;
328     struct proc *p = myproc();
329     // Allocate process.
330     if((np = allocproc()) == 0) {
331         return -1;
332     }
333     if((uint64)stack % PGSIZE != 0 || stack == 0) {
334         return -1;
335     }
336
337     // share same address space with parent
338     np->state = UNUSED;
339     np->sz = p->sz;
340     *(np->trapframe) = *(p->trapframe);
341     np->pagetable = p->pagetable;
342
343     np->context.ra = (uint64)func;
344     np->trapframe->a0 = 0;
345
346     // use the same file descriptor
347     for(i=0; i<NOFILE; i++)
348     {
349         if(p->ofile[i])
350         {
351             np->ofile[i] = filedup(p->ofile[i]);
352         }
353         np->cwd = idup(p->cwd);
354     }
355     safestrcpy(np->name, p->name, sizeof(p->name));
356
357     uint64 ustack[2];
358     ustack[0] = 0xffffffff;
359     ustack[1] = (uint64)arg;
360
361     np->context.sp = (uint64)(stack + PGSIZE - 4);
362     *((uint64*) (np->context.sp)) = (uint64)arg;
363     *((uint64*) (np->context.sp - 4)) = 0xffffffff;
364     np->context.sp = (np->context.sp) - 4;
365
366     if(copyout(np->pagetable, np->context.sp, (char*)ustack, size) < 0)
367     {
368         printf("Stack copy failed\n");
369         return -1;
370     }
371     np->state = RUNNABLE;
372
373     pid = np->pid;
374     release(&np->lock);
375
376     acquire(&wait_lock);
377     np->parent = p;
378     release(&wait_lock);
379
380     acquire(&np->lock);
381     np->state = RUNNABLE;
382     release(&np->lock);
383
384     return pid;
385 }
386
387 // Pass p's abandoned children to init.
388 // Caller must hold wait_lock.
389 void
390 reparent(struct proc *p)

```

```

446     panic("zombie exit");
447 }
448
449 void texit(void)
450 {
451     int fd;
452     struct proc *p = myproc();
453
454     if(p == initproc)
455     {
456         panic("init exiting");
457     }
458     // Close all open files.
459     for(fd=0; fd<NOFILE; fd++)
460     {
461         if(p->ofile[fd])
462         {
463             struct file *f = p->ofile[fd];
464             fileclose(f);
465             p->ofile[fd] = 0;
466         }
467     }
468
469     begin_op();
470     iput(p->cwd);
471     end_op();
472     p->cwd = 0;
473     // Parent might be sleeping in wait().
474     acquire(&wait_lock);
475     wakeup(p->parent);
476
477     p->state = ZOMBIE;
478     release(&wait_lock);
479     sched();
480     panic("zombie exit");
481 }
482 // Wait for a child process to exit and return its pid.
483 // Return -1 if this process has no children.
484 int
485 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 // switch.S
110 void switch(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.

```

1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "kernel/fcntl.h"
4  #include "user/user.h"
5  #include "kernel/spinlock.h"
6
7  void *thread_create(void*(start_routine)(void*),void *arg)
8  {
9      void *stack = malloc(2*4096U);
10     if((uint64)stack%4096U)
11     {
12         stack = stack + (4096U - (uint64)stack%4096U);
13     }
14     int size = 8;
15     int tid = clone(stack,size,start_routine,arg);
16     if(tid<0)
17     {
18         printf("Clone failed\n",tid);
19         return 0;
20     }
21     texit();
22     return 0;
23 }
24
25 int lock_init(lock_t *lock)
26 {
27     lock->flag = 0;
28     return 0;
29 }
30 void lock_acquire(lock_t *lock)
31 {
32     //lock->flag = 1;
33     while (!__sync_lock_test_and_set(&lock->flag, 1))
34         ;
35 }
36
37 void lock_release(lock_t *lock){
38     lock->flag = 0;
39 }
40

```

C source file

- 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.

```

11 void *play(void *arg)
12 {
13     int tid = *(uint64*)arg;
14     int pass_num = pass_round;
15     int i;
16     for(i=0;i<pass_num;i++)
17     {
18         if(thrower!=tid)
19         {
20             lock_acquire(&lock);
21             pass_no++;
22             printf("Pass number %d : ",pass_no);
23             printf("Thread %d is passing the token to Thread %d\n",thrower,tid);
24             thrower = tid;
25             lock_release(&lock);
26             sleep(20);
27         }
28         tid = (tid+1)%thread_num;
29     }
30     printf("Simulation of Frisbee game has finished, %d rounds were played in total\n",pass_round);
31     exit(0);
32 }
33
34 int main(int argc, char *argv[])
35 {
36     lock_init(&lock);
37     thread_num = atoi(argv[1]);
38     pass_round = atoi(argv[2]);
39
40     int i;
41     uint64 arg = 0;
42     for(i=0;i<thread_num;i++)
43     {
44         arg = i+1;
45         thread_create(play((void*)&arg), (void*)&arg);
46         sleep(10);
47     }
48     sleep(40);
49     exit(0);
50 }
51

```

C source file

length: 964 lines: 51

- 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 \
147         $U/...

```

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.

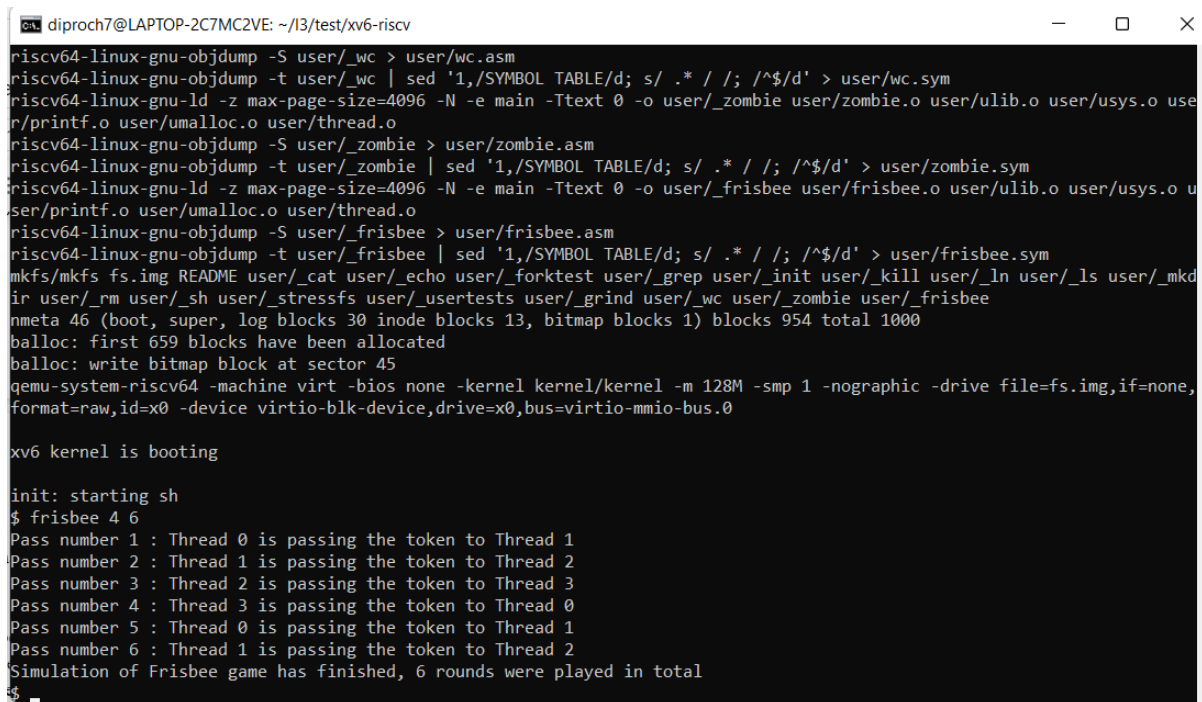
The *texit()* function is again, similar to the *exit()* function in proc.c, as the parent and children share the same file descriptors, here we remove the reparent() function and only free the shared resources, when the last one exits or reaped by wait.

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



```
dipiroch7@LAPTOP-2C7MC2VE: ~/l3/test/xv6-riscv
riscv64-linux-gnu-objdump -S user/_wc > user/wc.asm
riscv64-linux-gnu-objdump -t user/_wc | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > user/wc.sym
riscv64-linux-gnu-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_zombie user/zombie.o user/ulib.o user/usys.o user/_printf.o user/_umalloc.o user/_thread.o
riscv64-linux-gnu-objdump -S user/_zombie > user/zombie.asm
riscv64-linux-gnu-objdump -t user/_zombie | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > user/zombie.sym
riscv64-linux-gnu-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_frisbee user/frisbee.o user/ulib.o user/usys.o user/_printf.o user/_umalloc.o user/_thread.o
riscv64-linux-gnu-objdump -S user/_frisbee > user/frisbee.asm
riscv64-linux-gnu-objdump -t user/_frisbee | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > user/frisbee.sym
mkfs/mkfs fs.img README user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind user/_wc user/_zombie user/_frisbee
mknod meta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 954 total 1000
ballocc: first 659 blocks have been allocated
ballocc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 1 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

init: starting sh
$ frisbee 4 6
Pass number 1 : Thread 0 is passing the token to Thread 1
Pass number 2 : Thread 1 is passing the token to Thread 2
Pass number 3 : Thread 2 is passing the token to Thread 3
Pass number 4 : Thread 3 is passing the token to Thread 0
Pass number 5 : Thread 0 is passing the token to Thread 1
Pass number 6 : Thread 1 is passing the token to Thread 2
Simulation of Frisbee game has finished, 6 rounds were played in total
$
```

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.