Anthony Mendez | 862307065 | amend303

Jordan Kuschner | 862294132 | jkusc002

Due Date: 06/08/23

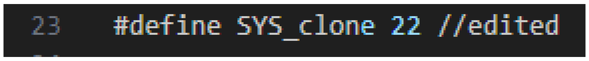
Lab 3 - XV6 Threads

Video Demo Link: https://youtu.be/2ZpMW1UNk6g

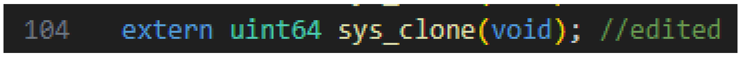
All Modified Files

- Kernel/ files
  - Syscall.h
  - Syscall.c
  - Sysproc.c
  - Proc.c
  - Defs.h
  - Proc.h
  - Trampoline.s
  - Trap.c
- User/ files
  - Usys.pl
  - User.h
  - Lab3_test.c
  - Thread.h
  - Thread.c
- Makefile

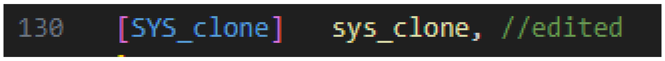
Change Explanations and Screenshots

- Syscall.h
    - Syscall number
    - 

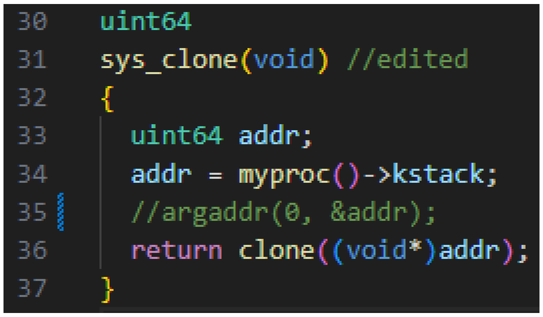
```
#define SYS_clone 22 //edited
```

- Syscall.c
    - System call function prototype
    - 

```
extern uint64 sys_clone(void); //edited
```

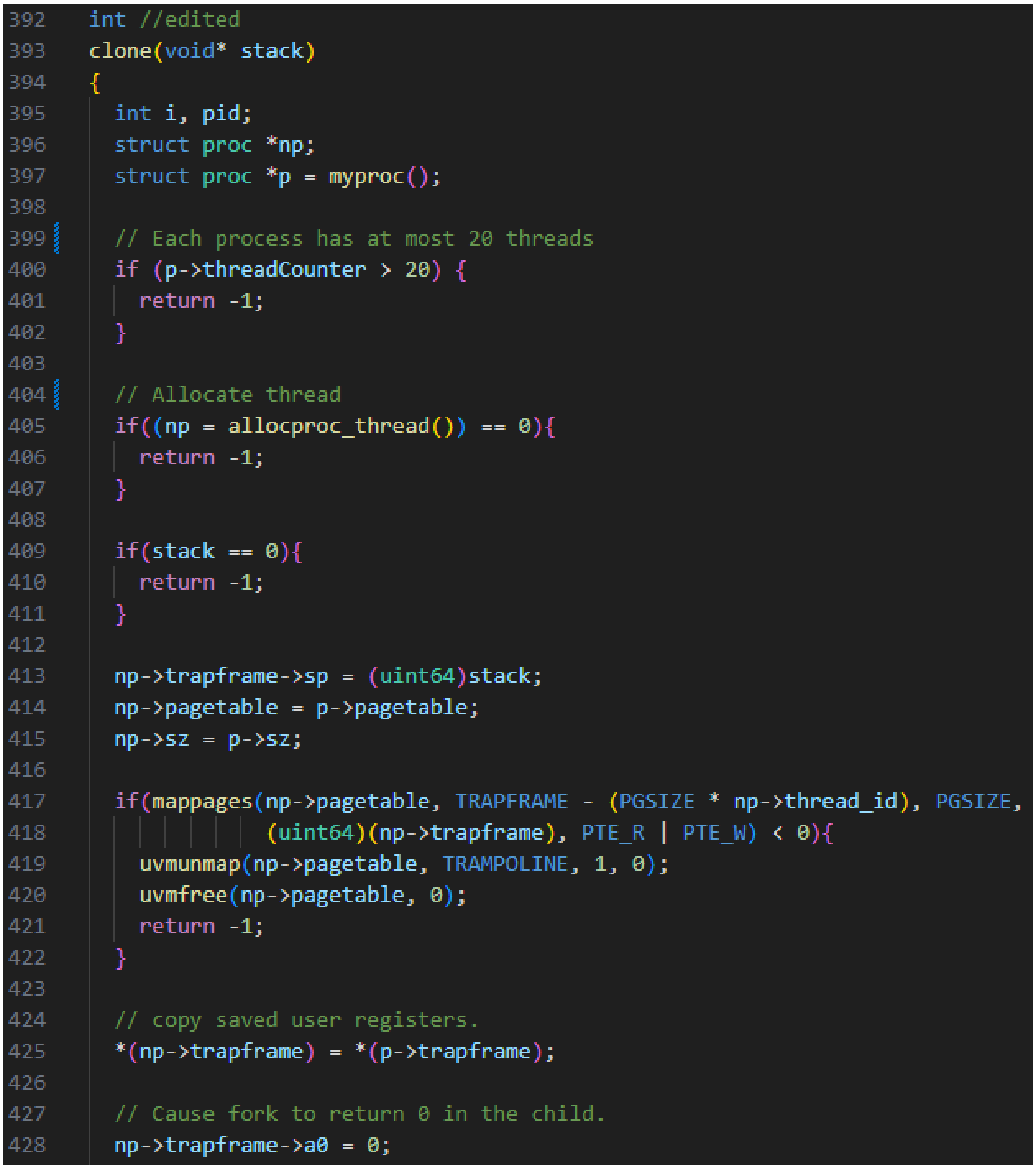
    - Syscall number to array mapping
    - 

```
[SYS_clone]   sys_clone, //edited
```

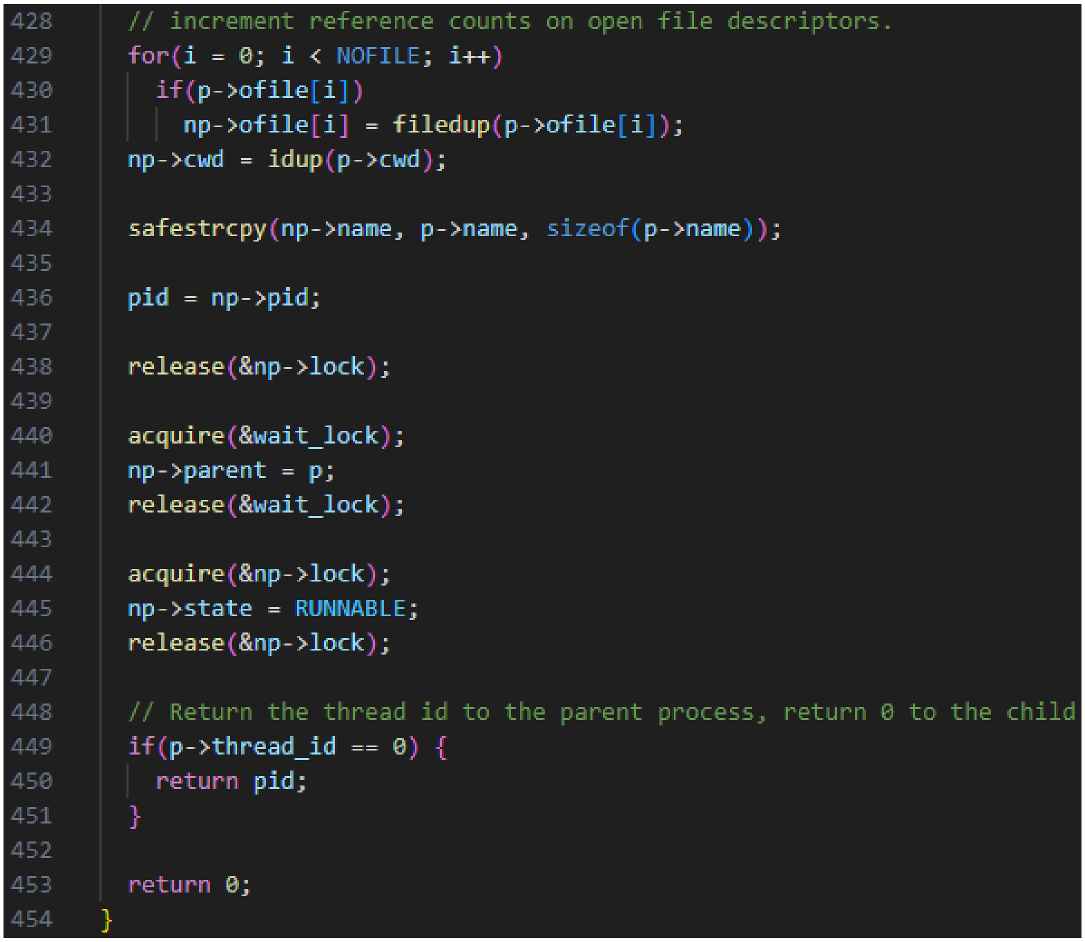
- Sysproc.c
    - Helper function to call syscall
    - 

```
uint64
sys_clone(void) //edited
{
  uint64 addr;
  addr = myproc()->kstack;
  //argaddr(0, &addr);
  return clone((void*)addr);
}
```

- Proc.c
    - Clone system call to create a new thread in a process. Similar to fork(). Calls allocproc_thread to create a child thread, sets the trapframe to the stack argument, shares the parents pagetable, maps the trapframe, and sets the state as runnable. Returns child pid to the calling process and returns 0 to child thread.

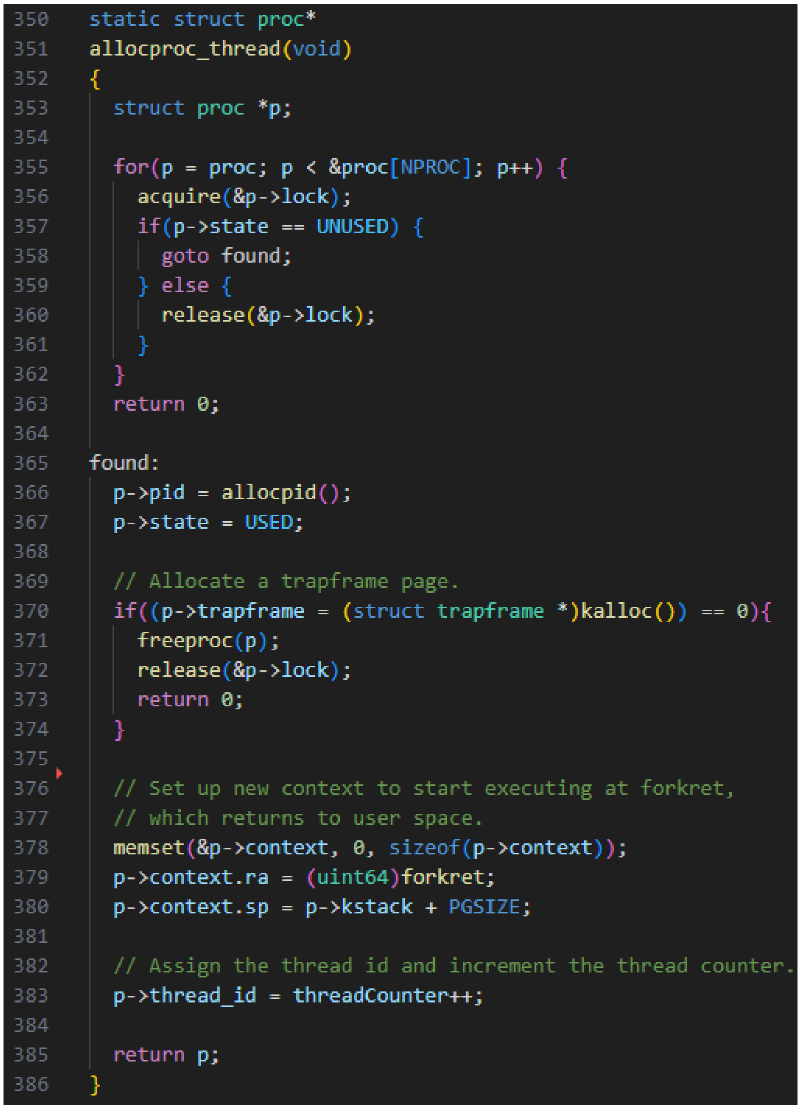


```c
int //edited
clone(void* stack)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Each process has at most 20 threads
  if (p->threadCounter > 20) {
    return -1;
  }

  // Allocate thread
  if((np = allocproc_thread()) == 0){
    return -1;
  }

  if(stack == 0){
    return -1;
  }

  np->trapframe->sp = (uint64)stack;
  np->pagetable = p->pagetable;
  np->sz = p->sz;

  if(mappages(np->pagetable, TRAPFRAME - (PGSIZE * np->thread_id), PGSIZE,
              (uint64)(np->trapframe), PTE_R | PTE_W) < 0){
    uvmunmap(np->pagetable, TRAMPOLINE, 1, 0);
    uvmfree(np->pagetable, 0);
    return -1;
  }

  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);

  // Cause fork to return 0 in the child.
  np->trapframe->a0 = 0;
```

○ 

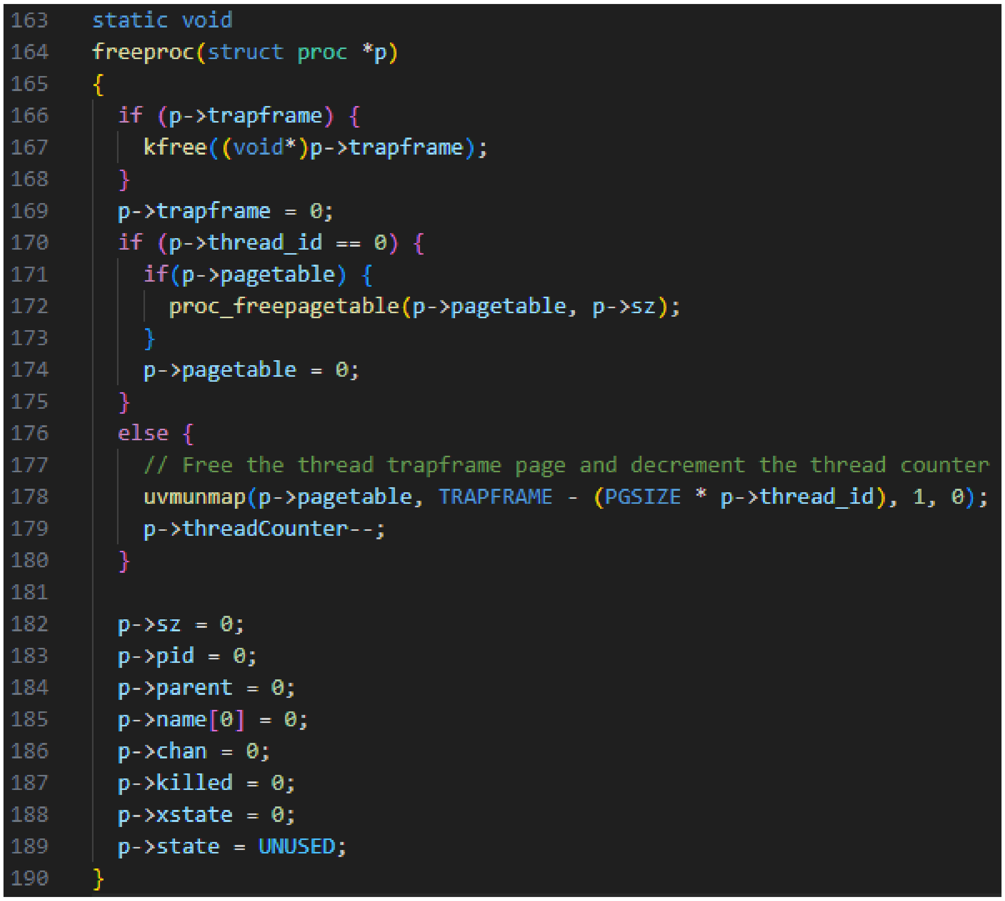
```
  // increment reference counts on open file descriptors.
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);

  safestrcpy(np->name, p->name, sizeof(p->name));

  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 the thread id to the parent process, return 0 to the child
  if(p->thread_id == 0) {
    return pid;
  }

  return 0;
}
```

○ Allocproc_thread to allocate the page and begin running the new thread. Mostly the same as allocproc() except it does not allocate a separate page table.

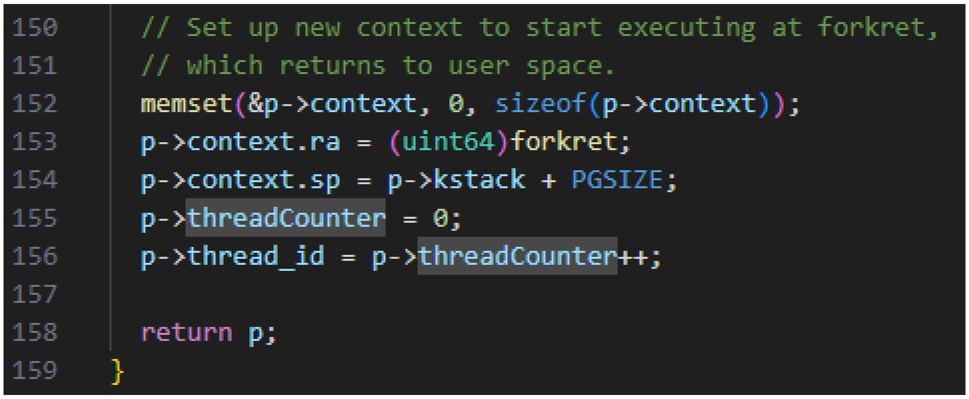
- 

```
static struct proc*
allocproc_thread(void)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == UNUSED) {
      goto found;
    } else {
      release(&p->lock);
    }
  }
  return 0;

found:
  p->pid = allocpid();
  p->state = USED;

  // Allocate a trapframe page.
  if((p->trapframe = (struct trapframe *)kalloc()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }

  // Set up new context to start executing at forkret,
  // which returns to user space.
  memset(&p->context, 0, sizeof(p->context));
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;

  // Assign the thread id and increment the thread counter.
  p->thread_id = threadCounter++;

  return p;
}
```

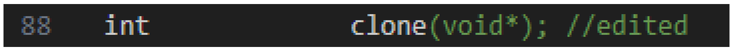
- Freeproc deallocation of child resources after the wait system call. Unmapping the thread based on its location in the stack and decreasing the counter since the thread is free'd.

- 

```
static void
freeproc(struct proc *p)
{
  if (p->trapframe) {
    kfree((void*)p->trapframe);
  }
  p->trapframe = 0;
  if (p->thread_id == 0) {
    if(p->pagetable) {
      proc_freepagetable(p->pagetable, p->sz);
    }
    p->pagetable = 0;
  }
  else {
    // Free the thread trapframe page and decrement the thread counter
    uvmunmap(p->pagetable, TRAPFRAME - (PGSIZE * p->thread_id), 1, 0);
    p->threadCounter--;
  }

  p->sz = 0;
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->chan = 0;
  p->killed = 0;
  p->xstate = 0;
  p->state = UNUSED;
}
```

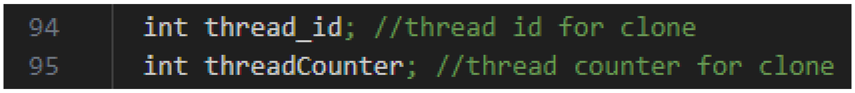
- Allocproc initializing thread count and thread id
- 

```
  // Set up new context to start executing at forkret,
  // which returns to user space.
  memset(&p->context, 0, sizeof(p->context));
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;
  p->threadCounter = 0;
  p->thread_id = p->threadCounter++;

  return p;
}
```

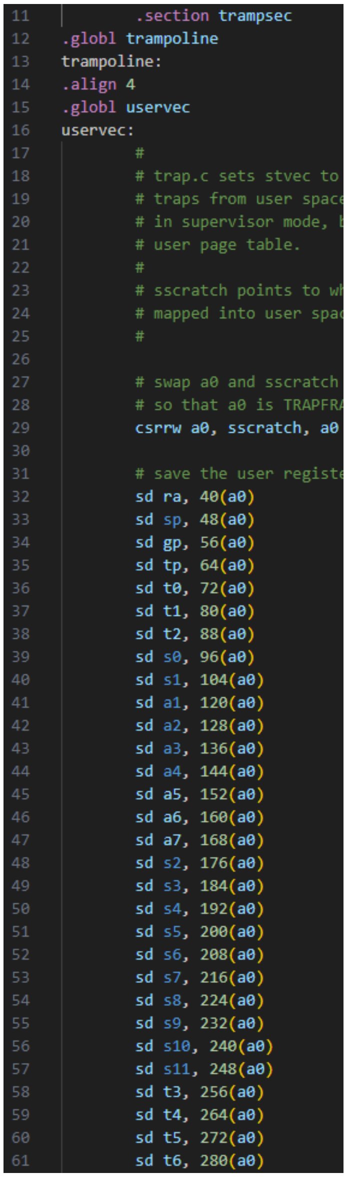
- Defs.h
  - System call function header
  - 

```
int             clone(void*); //edited
```

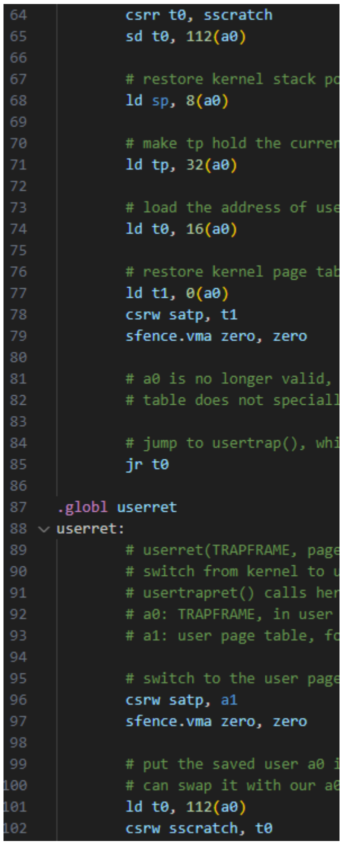
- Proc.h

- Thread id for each process control block and thread counter
- 

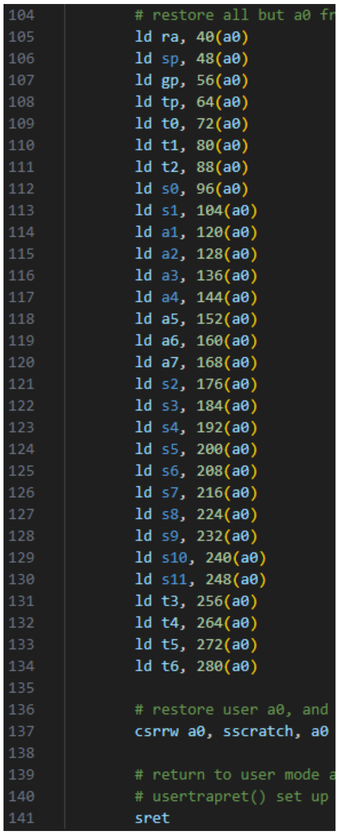
```
int thread_id; //thread id for clone
int threadCounter; //thread counter for clone
```

- Trampoline.s
    - Changed original code to code provided in lab
    - 

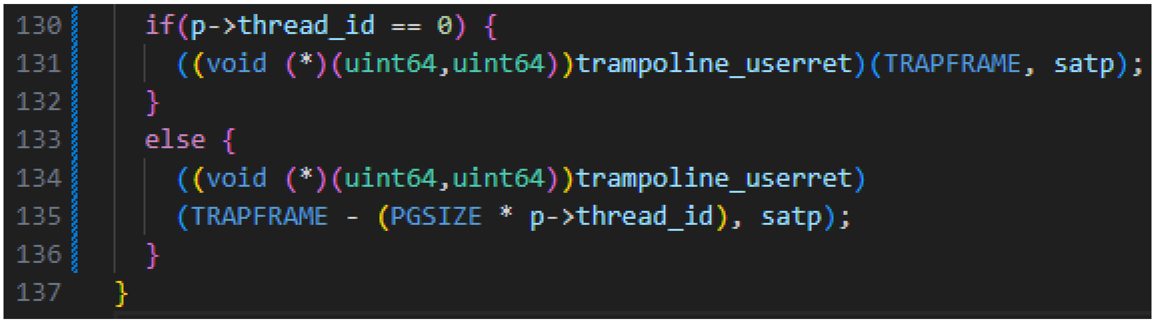
```
        .section trampsec
.globl trampoline
trampoline:
.align 4
.globl uservec
uservec:
        #
        # trap.c sets stvec to
        # traps from user space
        # in supervisor mode, t
        # user page table.
        #
        # sscratch points to wh
        # mapped into user spac
        #

        # swap a0 and sscratch
        # so that a0 is TRAPFRA
        csrrw a0, sscratch, a0

        # save the user registe
        sd ra, 40(a0)
        sd sp, 48(a0)
        sd gp, 56(a0)
        sd tp, 64(a0)
        sd t0, 72(a0)
        sd t1, 80(a0)
        sd t2, 88(a0)
        sd s0, 96(a0)
        sd s1, 104(a0)
        sd a1, 120(a0)
        sd a2, 128(a0)
        sd a3, 136(a0)
        sd a4, 144(a0)
        sd a5, 152(a0)
        sd a6, 160(a0)
        sd a7, 168(a0)
        sd s2, 176(a0)
        sd s3, 184(a0)
        sd s4, 192(a0)
        sd s5, 200(a0)
        sd s6, 208(a0)
        sd s7, 216(a0)
        sd s8, 224(a0)
        sd s9, 232(a0)
        sd s10, 240(a0)
        sd s11, 248(a0)
        sd t3, 256(a0)
        sd t4, 264(a0)
        sd t5, 272(a0)
        sd t6, 280(a0)
```

○ 

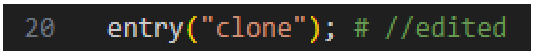
```
        csrr t0, sscratch
        sd t0, 112(a0)

        # restore kernel stack po
        ld sp, 8(a0)

        # make tp hold the curren
        ld tp, 32(a0)

        # load the address of use
        ld t0, 16(a0)

        # restore kernel page tab
        ld t1, 0(a0)
        csrw satp, t1
        sfence.vma zero, zero

        # a0 is no longer valid,
        # table does not speciall

        # jump to usertrap(), whi
        jr t0

.globl userret
userret:
        # userret(TRAPFRAME, page
        # switch from kernel to u
        # usertrapret() calls her
        # a0: TRAPFRAME, in user
        # a1: user page table, fo

        # switch to the user page
        csrw satp, a1
        sfence.vma zero, zero

        # put the saved user a0 i
        # can swap it with our a0
        ld t0, 112(a0)
        csrw sscratch, t0
```

○ 

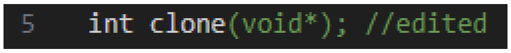
```
        # restore all but a0 fr
        ld ra, 40(a0)
        ld sp, 48(a0)
        ld gp, 56(a0)
        ld tp, 64(a0)
        ld t0, 72(a0)
        ld t1, 80(a0)
        ld t2, 88(a0)
        ld s0, 96(a0)
        ld s1, 104(a0)
        ld a1, 120(a0)
        ld a2, 128(a0)
        ld a3, 136(a0)
        ld a4, 144(a0)
        ld a5, 152(a0)
        ld a6, 160(a0)
        ld a7, 168(a0)
        ld s2, 176(a0)
        ld s3, 184(a0)
        ld s4, 192(a0)
        ld s5, 200(a0)
        ld s6, 208(a0)
        ld s7, 216(a0)
        ld s8, 224(a0)
        ld s9, 232(a0)
        ld s10, 240(a0)
        ld s11, 248(a0)
        ld t3, 256(a0)
        ld t4, 264(a0)
        ld t5, 272(a0)
        ld t6, 280(a0)

        # restore user a0, and
        csrrw a0, sscratch, a0

        # return to user mode a
        # usertrapret() set up
        sret
```

- Trap.c
  - In usertrapret(), telling kernel trapframe locations of children
  - 

```
    if(p->thread_id == 0) {
      ((void (*)(uint64,uint64))trampoline_userret)(TRAPFRAME, satp);
    }
    else {
      ((void (*)(uint64,uint64))trampoline_userret)
      (TRAPFRAME - (PGSIZE * p->thread_id), satp);
    }
  }
```

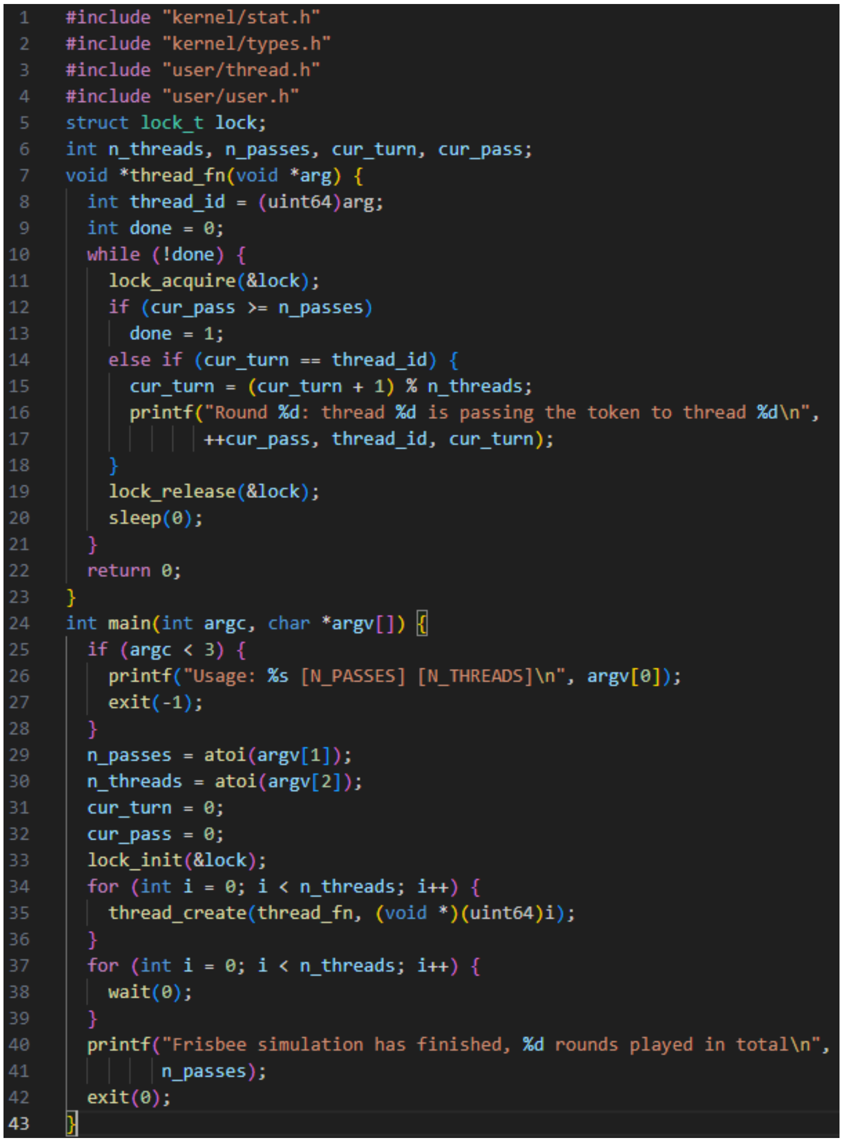
- Usys.pl
  - User system call wrapper entry
  - 

```
entry("clone"); # //edited
```

- User.h
  - Clone system call in user space
  - 

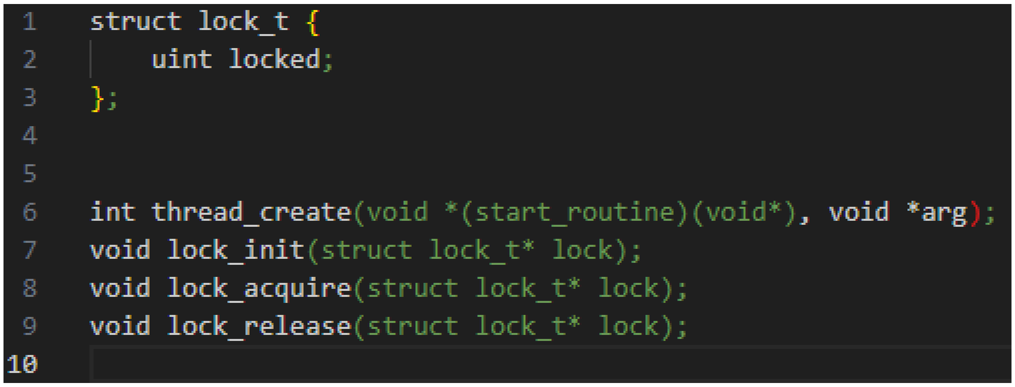
```
int clone(void*); //edited
```

- Lab3_test.c
  - Provided code to test the clone system call

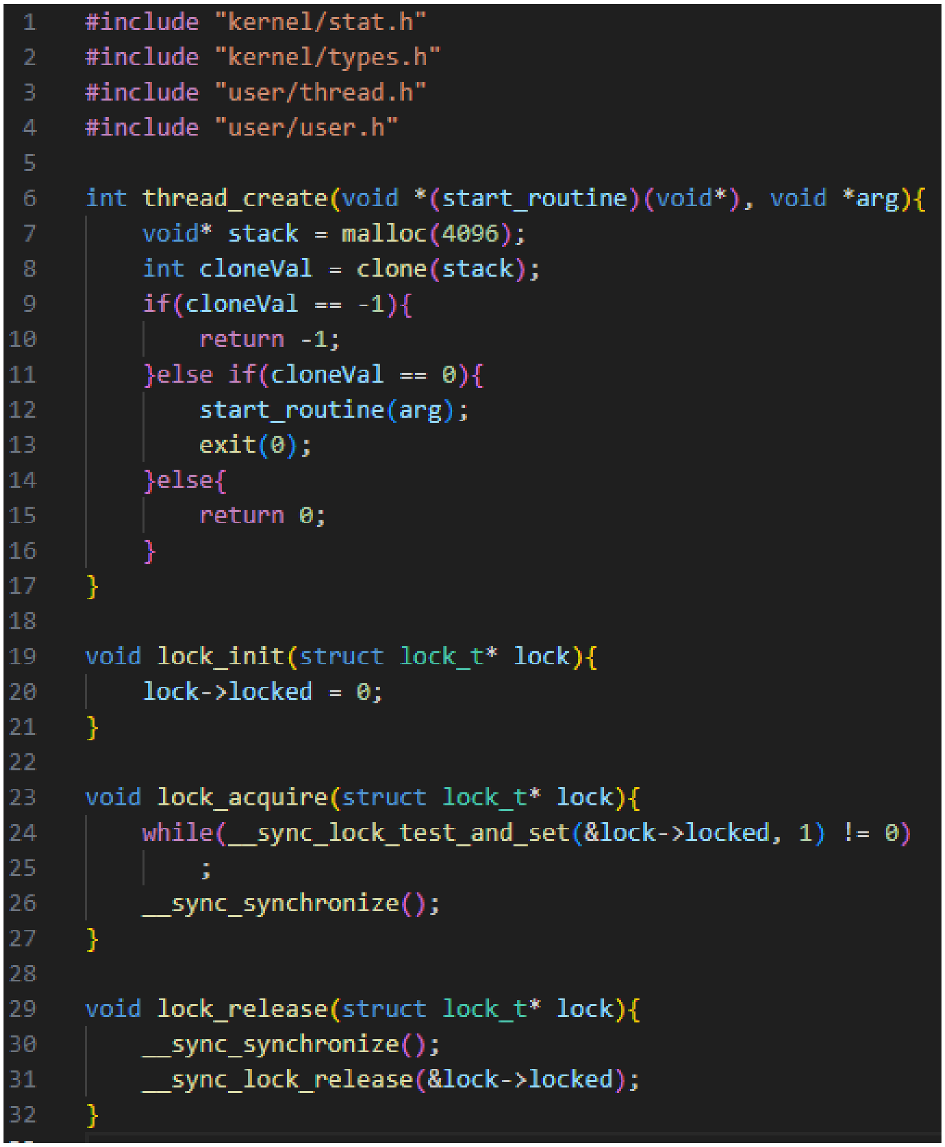


```c
#include "kernel/stat.h"
#include "kernel/types.h"
#include "user/thread.h"
#include "user/user.h"
struct lock_t lock;
int n_threads, n_passes, cur_turn, cur_pass;
void *thread_fn(void *arg) {
  int thread_id = (uint64)arg;
  int done = 0;
  while (!done) {
    lock_acquire(&lock);
    if (cur_pass >= n_passes)
      done = 1;
    else if (cur_turn == thread_id) {
      cur_turn = (cur_turn + 1) % n_threads;
      printf("Round %d: thread %d is passing the token to thread %d\n",
             ++cur_pass, thread_id, cur_turn);
    }
    lock_release(&lock);
    sleep(0);
  }
  return 0;
}
int main(int argc, char *argv[]) {
  if (argc < 3) {
    printf("Usage: %s [N_PASSES] [N_THREADS]\n", argv[0]);
    exit(-1);
  }
  n_passes = atoi(argv[1]);
  n_threads = atoi(argv[2]);
  cur_turn = 0;
  cur_pass = 0;
  lock_init(&lock);
  for (int i = 0; i < n_threads; i++) {
    thread_create(thread_fn, (void *)(uint64)i);
  }
  for (int i = 0; i < n_threads; i++) {
    wait(0);
  }
  printf("Frisbee simulation has finished, %d rounds played in total\n",
         n_passes);
  exit(0);
}
```

- Thread.h
    - Header file with lock structure and function prototypes.

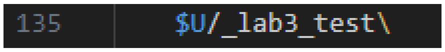


```c
struct lock_t {
    uint locked;
};


int thread_create(void *(start_routine)(void*), void *arg);
void lock_init(struct lock_t* lock);
void lock_acquire(struct lock_t* lock);
void lock_release(struct lock_t* lock);

```

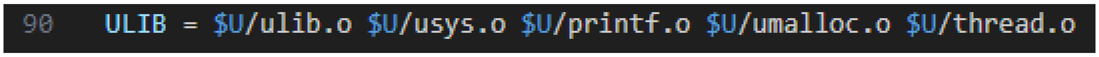
- Thread.c
    - Lock function implementations and thread creation by passing stack. Lock init begins locked. Acquire holds the lock in place and syncs it. Release syncs and releases the lock. The return pid will be 0 for the child and the routine starts and exits for the child. The parent receives the childs pid
    - 

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

int thread_create(void *(start_routine)(void*), void *arg){
    void* stack = malloc(4096);
    int cloneVal = clone(stack);
    if(cloneVal == -1){
        return -1;
    }else if(cloneVal == 0){
        start_routine(arg);
        exit(0);
    }else{
        return 0;
    }
}

void lock_init(struct lock_t* lock){
    lock->locked = 0;
}

void lock_acquire(struct lock_t* lock){
    while(__sync_lock_test_and_set(&lock->locked, 1) != 0)
        ;
    __sync_synchronize();
}

void lock_release(struct lock_t* lock){
    __sync_synchronize();
    __sync_lock_release(&lock->locked);
}
```

- Makefile
    - Added lab3_test into programs

- 

  ```
          $U/_lab3_test\
  ```

- Thread for the thread library
- 

  ```
  ULIB = $U/ulib.o $U/usys.o $U/printf.o $U/umalloc.o $U/thread.o
  ```

Description of XV6 Source Code

The default XV6 code supports process creation, but it does not support threads. In this lab, we implement the clone system call to support thread creation. The clone system call allocates a thread for the calling process and returns the child pid to the parent and 0 to the child. We also implemented a user level thread library to clone child threads with user stacks. The thread library also uses a lock that can be acquired and released. The thread create function creates a stack the size of a page, passes it to the clone function, and begins the thread execution until it finishes and exits. The test begins in the user space by calling the thread create function. Then the thread create function calls the clone function and begins the thread execution. Once the clone function is called, the system call is identified in the user space and control is transferred over to the kernel space. The clone call allocates the thread, does the required mappings and proc assignments, and returns the child thread pid to the parent process.

Summary of Contributions

Jordan implemented the thread library, trampoline, trap code, allocproc_thread() and clone(). Anthony also worked on the clone() system call, helped debug some important “panic: “ type errors, and wrote the report.