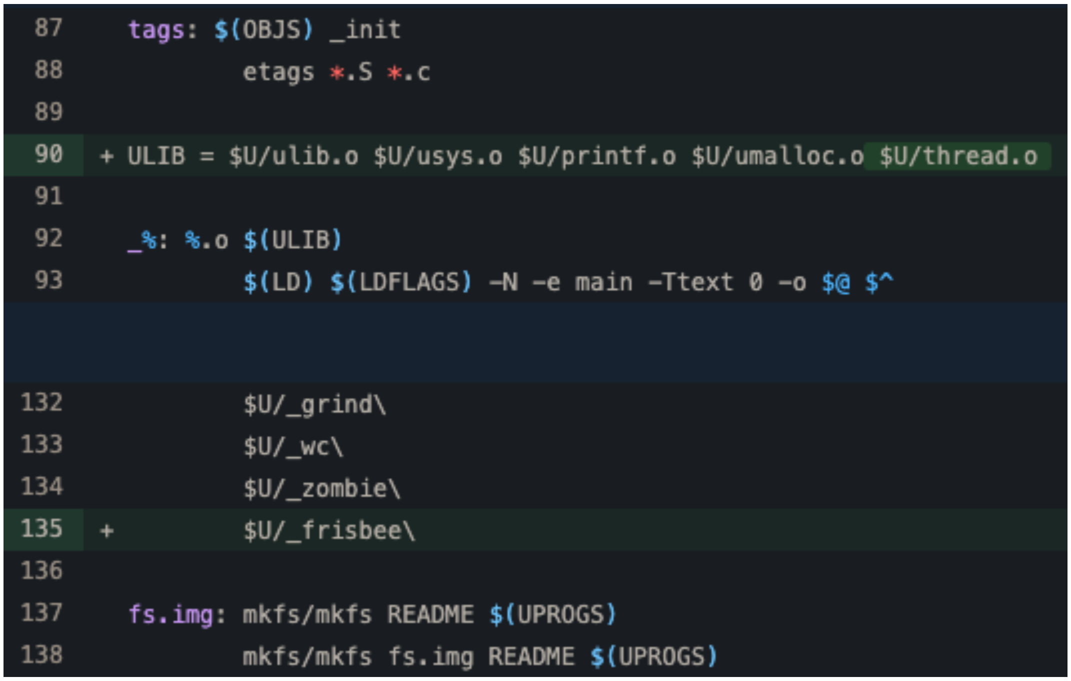
Lab 3 - Report

Karthik Harpanahalli - 862254197
Varun Sapre - 862255166

1. List of all files modified: 13 files were modified in total
    1.1. Makefile
    1.2. kernel/defs.h
    1.3. kernel/proc.c
    1.4. kernel/proc.h
    1.5. kernel/syscall.c
    1.6. kernel/syscall.h
    1.7. kernel/sysproc.c
    1.8. kernel/trap.c
    1.9. user/frisbee.c
    1.10. user/thread.c
    1.11. user/thread.h
    1.12. user/user.h
    1.13. user/usys.pl

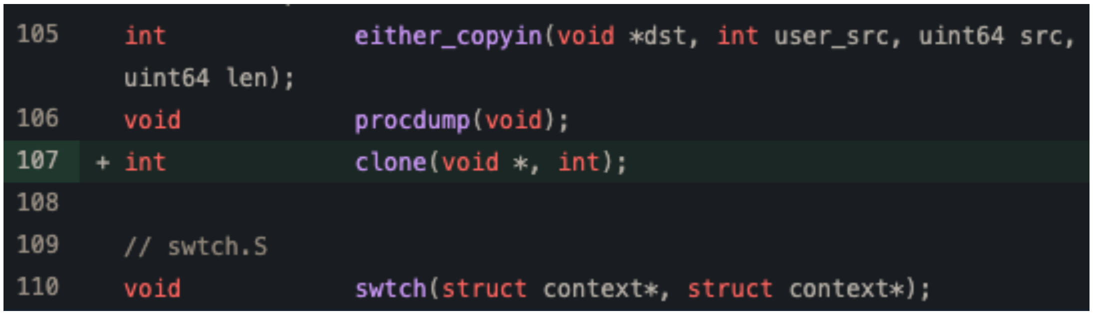
2. A detailed explanation on what changes you have made and screenshots showing your work and results
    2.1. Makefile



```
  tags: $(OBJS) _init
          etags *.S *.c

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

  _%: %.o $(ULIB)
          $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^


          $U/_grind\
          $U/_wc\
          $U/_zombie\
+         $U/_frisbee\

  fs.img: mkfs/mkfs README $(UPROGS)
          mkfs/mkfs fs.img README $(UPROGS)
```

We added our user program (frisbee) in Makefile to make our user program available for the xv6 source code compilation. We also add threading from our code.

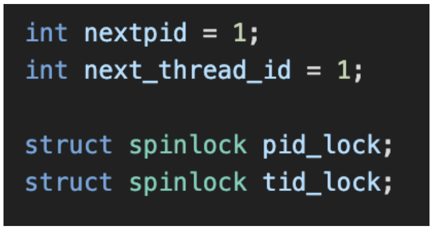
2.2. kernel/defs.h



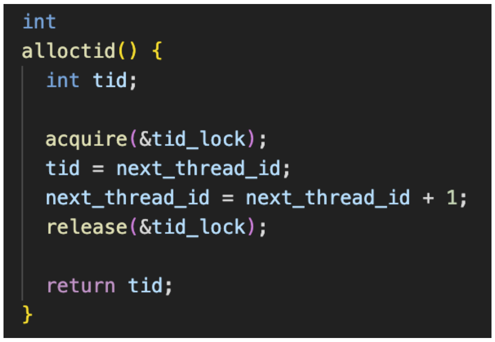
```
int             either_copyin(void *dst, int user_src, uint64 src,
uint64 len);
void            procdump(void);
+ int           clone(void *, int);

// swtch.S
void            swtch(struct context*, struct context*);
```

We add our clone procedure in the defs.h file to make them accessible.

2.3. kernel/proc.c

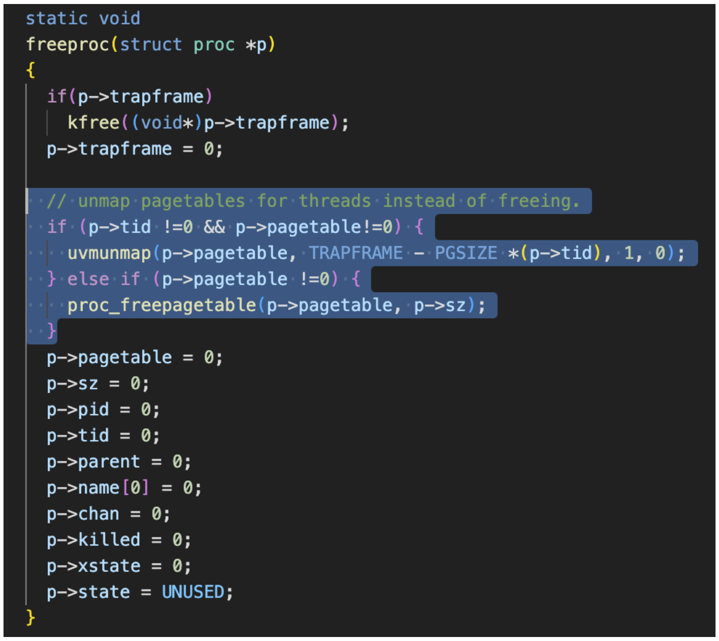


```
int nextpid = 1;
int next_thread_id = 1;

struct spinlock pid_lock;
struct spinlock tid_lock;
```



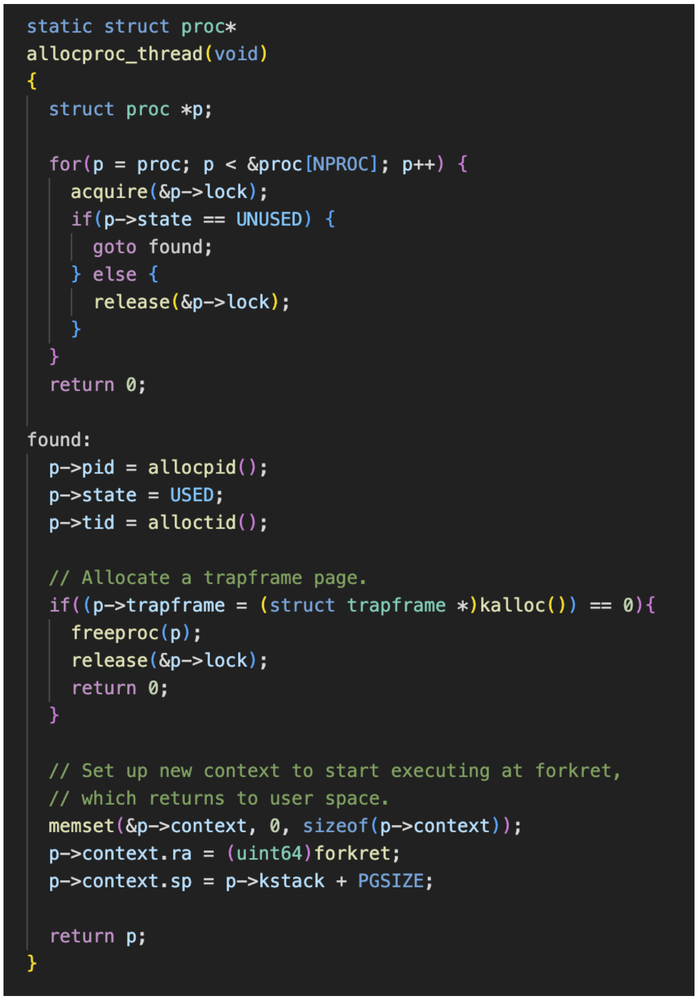
```
int
alloctid() {
  int tid;

  acquire(&tid_lock);
  tid = next_thread_id;
  next_thread_id = next_thread_id + 1;
  release(&tid_lock);

  return tid;
}
```

We add a new variable to track thread IDs



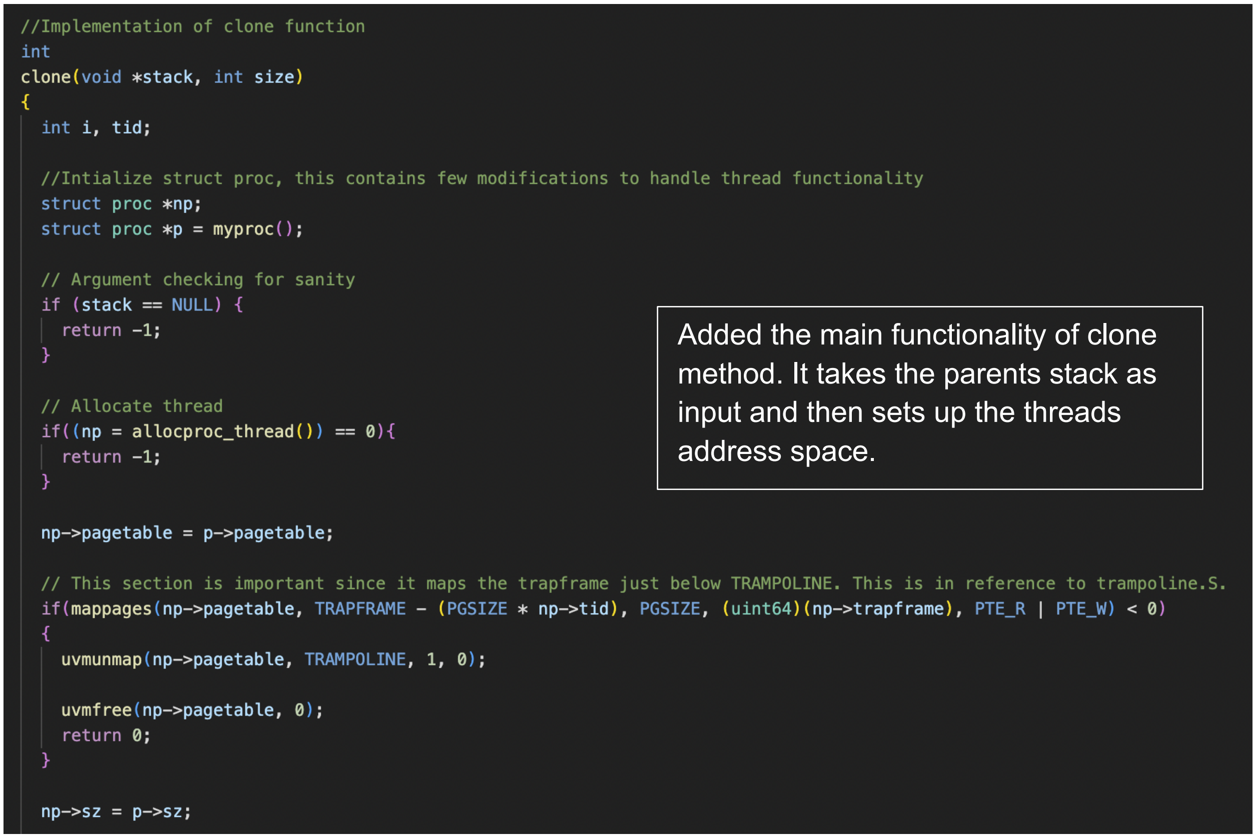
```
static void
freeproc(struct proc *p)
{
  if(p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;

  // unmap pagetables for threads instead of freeing.
  if (p->tid !=0 && p->pagetable!=0) {
    uvmunmap(p->pagetable, TRAPFRAME - PGSIZE *(p->tid), 1, 0);
  } else if (p->pagetable !=0) {
    proc_freepagetable(p->pagetable, p->sz);
  }
  p->pagetable = 0;
  p->sz = 0;
  p->pid = 0;
  p->tid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->chan = 0;
  p->killed = 0;
  p->xstate = 0;
  p->state = UNUSED;
}
```

Changes made to freeproc to take care of freeing up thread pagetables too. This is called from the wait() function.



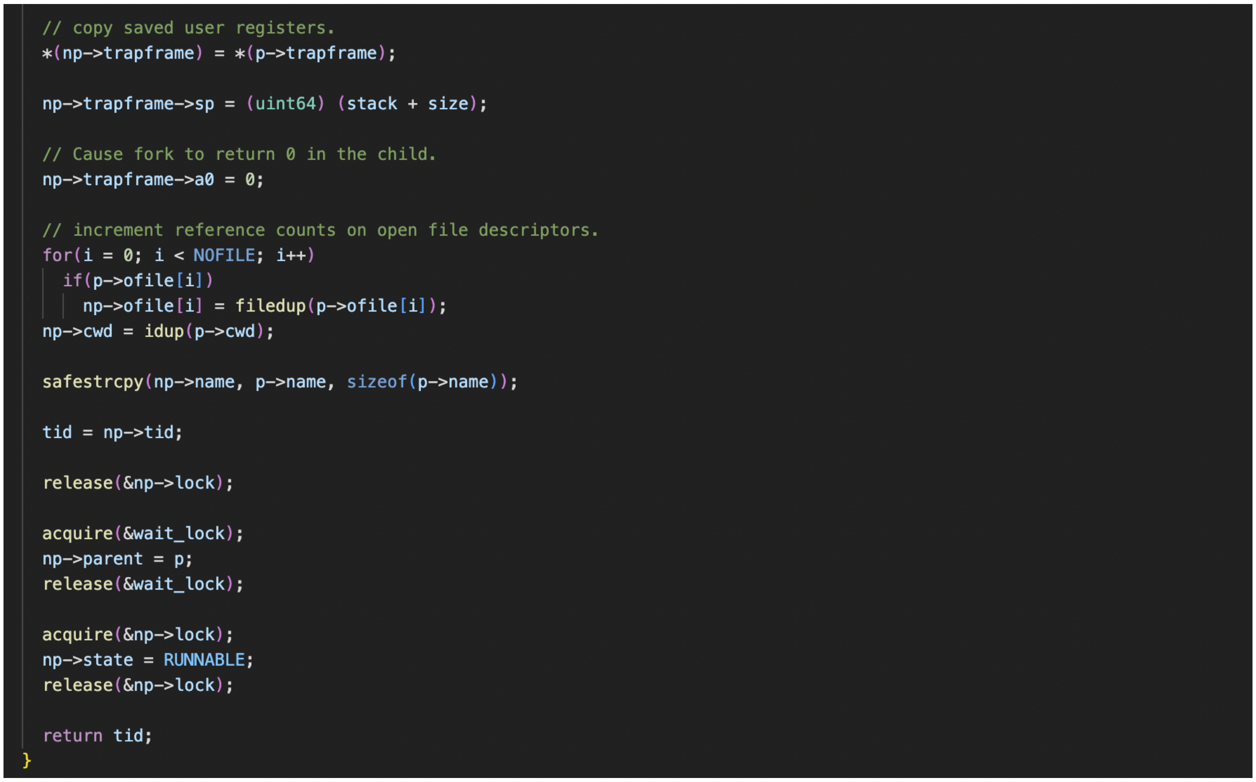
```c
static struct proc*
allocproc_thread(void)
{
  struct proc *p;

  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;
  p->tid = alloctid();

  // 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;

  return p;
}
```

Added method for holding metadata for threads. Similar to the allocproc method of xv6.

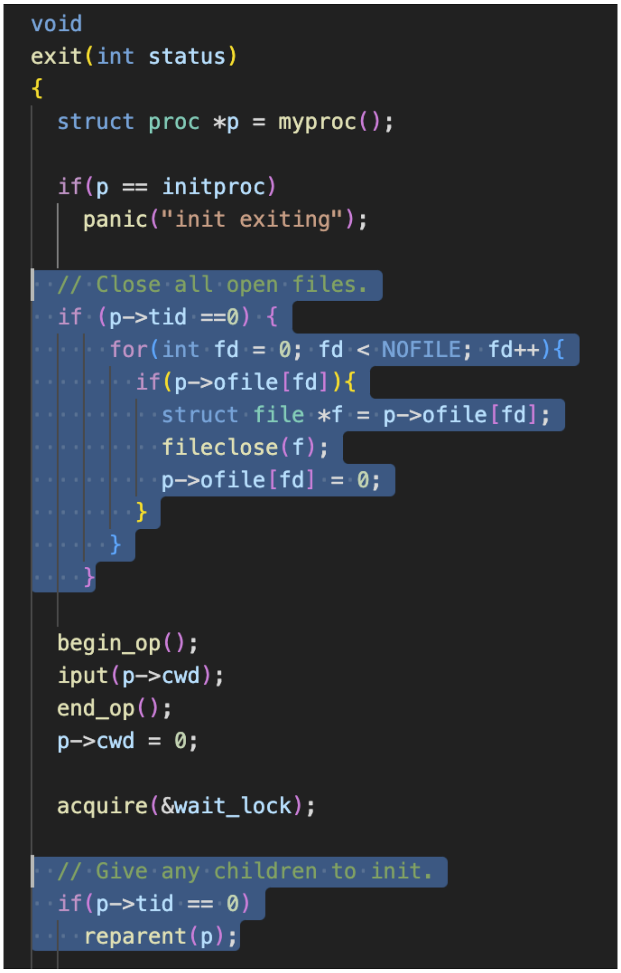


```c
//Implementation of clone function
int
clone(void *stack, int size)
{
  int i, tid;

  //Intialize struct proc, this contains few modifications to handle thread functionality
  struct proc *np;
  struct proc *p = myproc();

  // Argument checking for sanity
  if (stack == NULL) {
    return -1;
  }

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

  np->pagetable = p->pagetable;

  // This section is important since it maps the trapframe just below TRAMPOLINE. This is in reference to trampoline.S.
  if(mappages(np->pagetable, TRAPFRAME - (PGSIZE * np->tid), PGSIZE, (uint64)(np->trapframe), PTE_R | PTE_W) < 0)
  {
    uvmunmap(np->pagetable, TRAMPOLINE, 1, 0);

    uvmfree(np->pagetable, 0);
    return 0;
  }

  np->sz = p->sz;
```

Added the main functionality of clone method. It takes the parents stack as input and then sets up the threads address space.



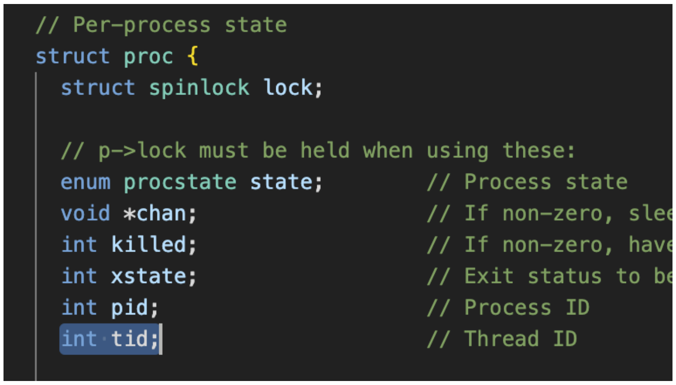
```
  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);

  np->trapframe->sp = (uint64) (stack + size);

  // 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));

  tid = np->tid;

  release(&np->lock);

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

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

  return tid;
}
```



```
void
exit(int status)
{
  struct proc *p = myproc();

  if(p == initproc)
    panic("init exiting");

  // Close all open files.
  if (p->tid ==0) {
      for(int 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;

  acquire(&wait_lock);

  // Give any children to init.
  if(p->tid == 0)
    reparent(p);
```

Changed exit() function to take care of the open files of threads and then exit gracefully.

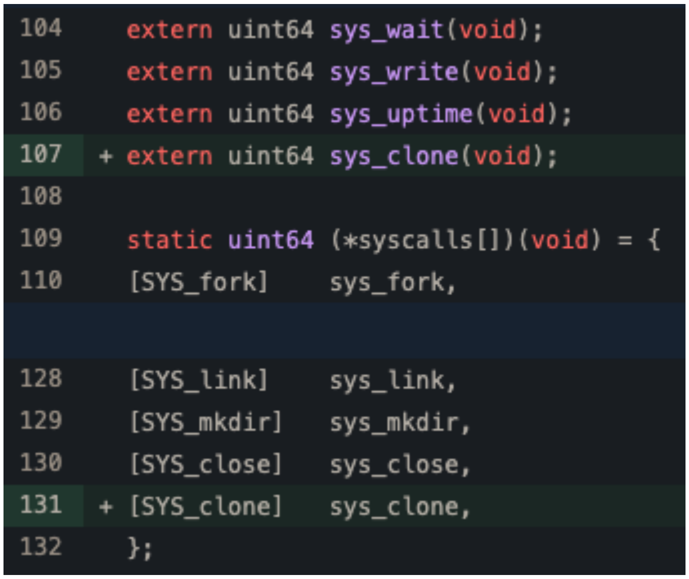
2.4. kernel/proc.h



```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, slee
  int killed;                  // If non-zero, have
  int xstate;                  // Exit status to be
  int pid;                     // Process ID
  int tid;                     // Thread ID
```

Added Thread ID in proc struct

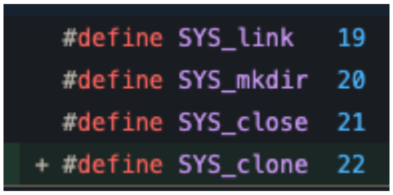
2.5. kernel/syscall.c



```
  extern uint64 sys_wait(void);
  extern uint64 sys_write(void);
  extern uint64 sys_uptime(void);
+ extern uint64 sys_clone(void);

  static uint64 (*syscalls[])(void) = {
  [SYS_fork]    sys_fork,

  [SYS_link]    sys_link,
  [SYS_mkdir]   sys_mkdir,
  [SYS_close]   sys_close,
+ [SYS_clone]   sys_clone,
  };
```

Basic changes required to add clone system call

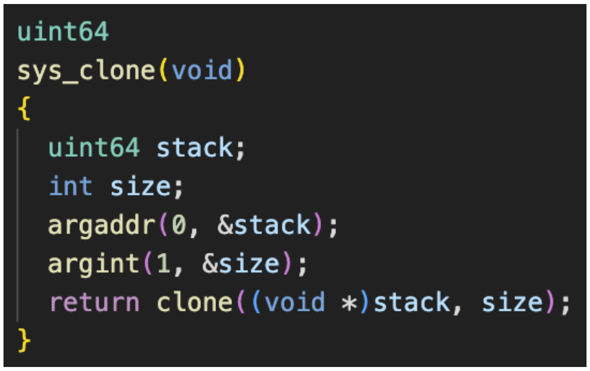
2.6. kernel/syscall.h



```
  #define SYS_link   19
  #define SYS_mkdir  20
  #define SYS_close  21
+ #define SYS_clone  22
```

Basic changes required to add clone system call

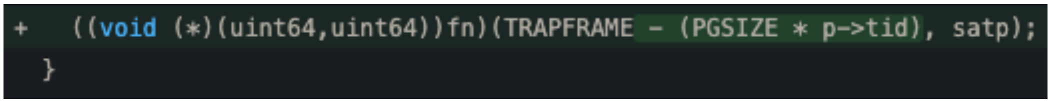
2.7. kernel/sysproc.c



```
uint64
sys_clone(void)
{
  uint64 stack;
  int size;
  argaddr(0, &stack);
  argint(1, &size);
  return clone((void *)stack, size);
}
```

System call function for clone. Reads the stack and stack size provided and calls clone function in proc.c

2.8. kernel/trap.c

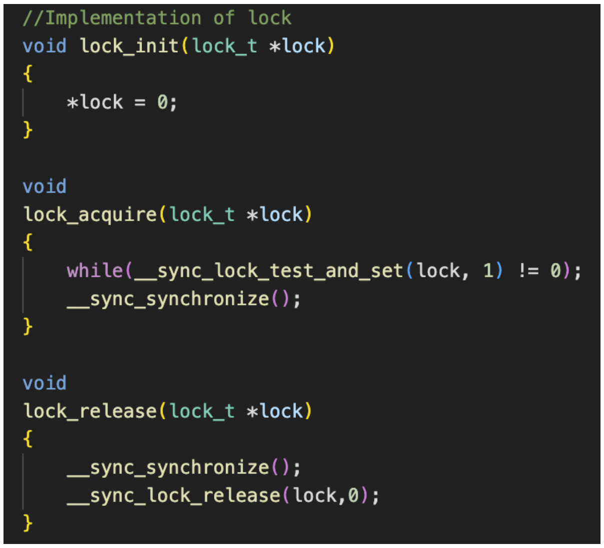


```
+   ((void (*)(uint64,uint64))fn)(TRAPFRAME - (PGSIZE * p->tid), satp);
  }
```

2.9. user/frisbee.c

> same code provided in lab3 document

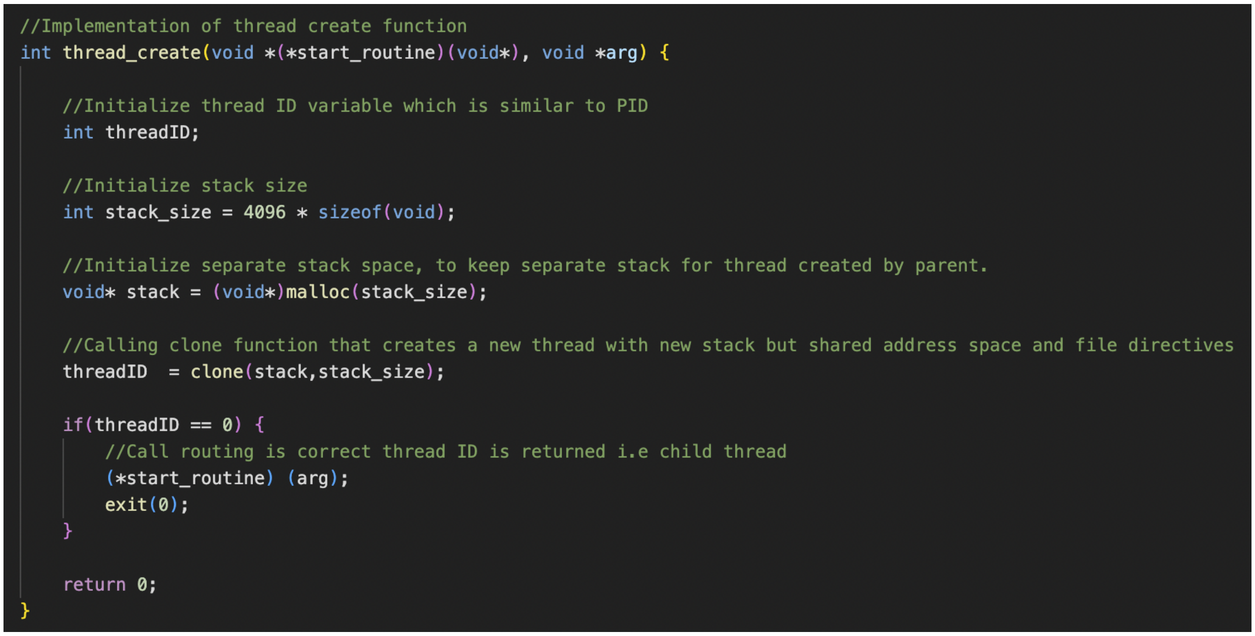
2.10. user/thread.c



```
//Implementation of lock
void lock_init(lock_t *lock)
{
    *lock = 0;
}

void
lock_acquire(lock_t *lock)
{
    while(__sync_lock_test_and_set(lock, 1) != 0);
    __sync_synchronize();
}

void
lock_release(lock_t *lock)
{
    __sync_synchronize();
    __sync_lock_release(lock,0);
}
```

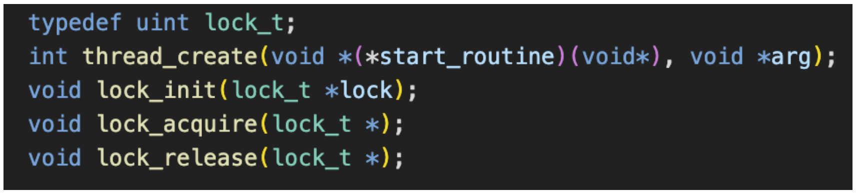
Added spinlock usage functions

Thread Create function which creates the stack and then calls the system call clone to setup the thread. After setup is complete, it calls the provided start_routine only in the executing thread.



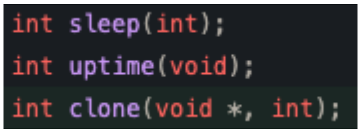
```
//Implementation of thread create function
int thread_create(void *(*start_routine)(void*), void *arg) {

    //Initialize thread ID variable which is similar to PID
    int threadID;

    //Initialize stack size
    int stack_size = 4096 * sizeof(void);

    //Initialize separate stack space, to keep separate stack for thread created by parent.
    void* stack = (void*)malloc(stack_size);

    //Calling clone function that creates a new thread with new stack but shared address space and file directives
    threadID  = clone(stack,stack_size);

    if(threadID == 0) {
        //Call routing is correct thread ID is returned i.e child thread
        (*start_routine) (arg);
        exit(0);
    }

    return 0;
}
```

2.11. user/thread.h



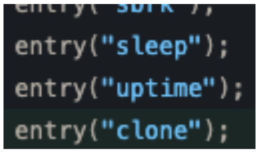
```
typedef uint lock_t;
int thread_create(void *(*start_routine)(void*), void *arg);
void lock_init(lock_t *lock);
void lock_acquire(lock_t *);
void lock_release(lock_t *);
```

2.12. user/user.h



```
int sleep(int);
int uptime(void);
int clone(void *, int);
```

2.13. user/usys.pl



```
entry("sbrk");
entry("sleep");
entry("uptime");
entry("clone");
```

Basic changes required to call clone system call from user-level code

Part 1: Clone() System call

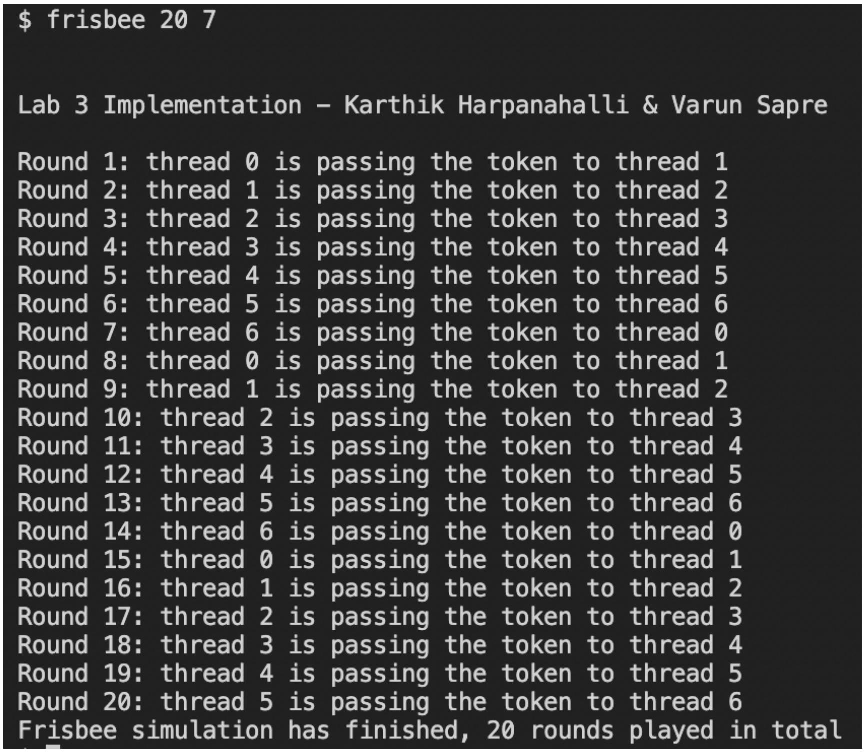
Basic changes required to call clone system call from user-level code

To add kernel-level threads support in xv6, we have implemented a new system call to create a child thread called clone() system call. This new system call creates a child thread that uses the parent's address space and share the same file descriptors as parent's. And thus requiring a few modifications to exit() and wait() calls. The stack argument points to the appropriate address on the stack. Since each thread's trapframe page needs to be mapped to a certain user space without any overlap we made necessary modifications in usertrapret() in kernel/trap.c.

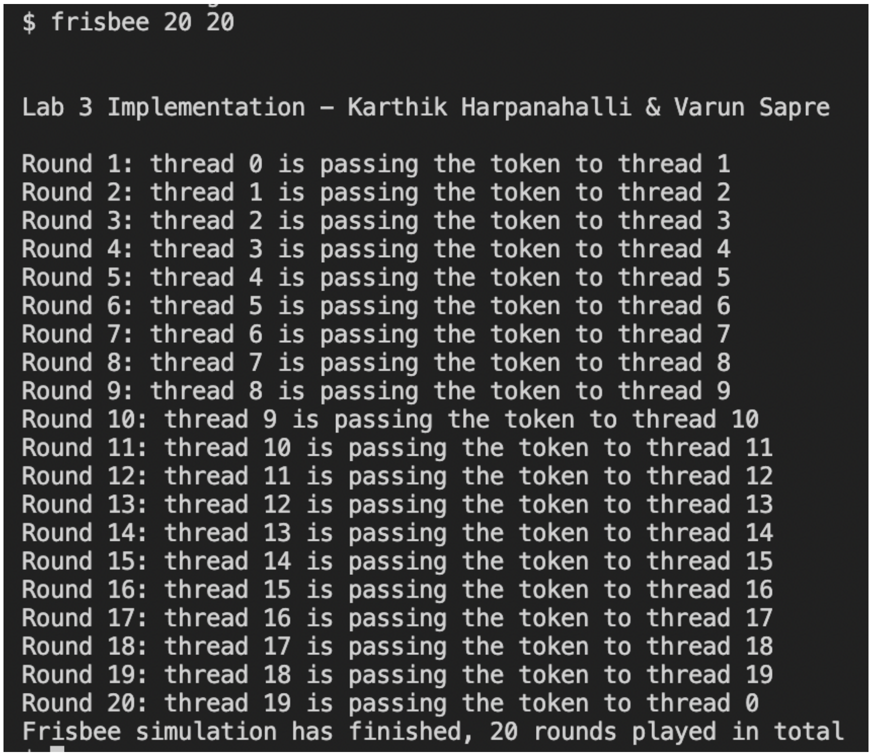
Part 2: User-level thread library

To implement a user-level thread library we implemented the required functions: thread_create(), lock_init(), lock_aquire() and lock_release(). thread_create() is a wrapper for clone() and allocates a user stack that is of size PGSIZE bytes and then invokes clone() to create a thread. The return values are 0 for success and -1 for failure and is returned to the parent. When the start_routine() returns it invokes exit() to terminate the child thread.
We also implemented a user-level spin lock mechanism. To declare a lock, we use lock_init(). lock_acquire() and lock_release(), as the name suggests acquiring and release the locks. The atomic test and set operation are used for building this mechanism, using the in-built sync_test_and_set function.

3. Screenshots of experiments



```
$ frisbee 20 7


Lab 3 Implementation - Karthik Harpanahalli & Varun Sapre

Round 1: thread 0 is passing the token to thread 1
Round 2: thread 1 is passing the token to thread 2
Round 3: thread 2 is passing the token to thread 3
Round 4: thread 3 is passing the token to thread 4
Round 5: thread 4 is passing the token to thread 5
Round 6: thread 5 is passing the token to thread 6
Round 7: thread 6 is passing the token to thread 0
Round 8: thread 0 is passing the token to thread 1
Round 9: thread 1 is passing the token to thread 2
Round 10: thread 2 is passing the token to thread 3
Round 11: thread 3 is passing the token to thread 4
Round 12: thread 4 is passing the token to thread 5
Round 13: thread 5 is passing the token to thread 6
Round 14: thread 6 is passing the token to thread 0
Round 15: thread 0 is passing the token to thread 1
Round 16: thread 1 is passing the token to thread 2
Round 17: thread 2 is passing the token to thread 3
Round 18: thread 3 is passing the token to thread 4
Round 19: thread 4 is passing the token to thread 5
Round 20: thread 5 is passing the token to thread 6
Frisbee simulation has finished, 20 rounds played in total
```



```
$ frisbee 20 20


Lab 3 Implementation - Karthik Harpanahalli & Varun Sapre

Round 1: thread 0 is passing the token to thread 1
Round 2: thread 1 is passing the token to thread 2
Round 3: thread 2 is passing the token to thread 3
Round 4: thread 3 is passing the token to thread 4
Round 5: thread 4 is passing the token to thread 5
Round 6: thread 5 is passing the token to thread 6
Round 7: thread 6 is passing the token to thread 7
Round 8: thread 7 is passing the token to thread 8
Round 9: thread 8 is passing the token to thread 9
Round 10: thread 9 is passing the token to thread 10
Round 11: thread 10 is passing the token to thread 11
Round 12: thread 11 is passing the token to thread 12
Round 13: thread 12 is passing the token to thread 13
Round 14: thread 13 is passing the token to thread 14
Round 15: thread 14 is passing the token to thread 15
Round 16: thread 15 is passing the token to thread 16
Round 17: thread 16 is passing the token to thread 17
Round 18: thread 17 is passing the token to thread 18
Round 19: thread 18 is passing the token to thread 19
Round 20: thread 19 is passing the token to thread 0
Frisbee simulation has finished, 20 rounds played in total
```

4. A brief summary of the contributions of each member

   Both team members have equally implemented all customizations needed for the lab.

5. Demo

   CS202-Lab3-Demo.mov