Operating System Concepts
COP4610.02
Mini Project 4

Noah Baldwin
Cody Carroll
Paul Teleweck

Work Breakdown:

Names	**Code**	**Report**	**Documentation**	**Presentation**
Noah Baldwin	33.33%	33.33%	33.33%	33.33%
Cody Carroll	33.33%	33.33%	33.33%	33.33%
Paul Teleweck	33.33%	33.33%	33.33%	33.33%

Abstract:

In this project, you'll be adding real kernel threads to xv6. First, define a new system call to create a kernel thread, called **clone()**, as well as one to wait for a thread called **join()**. Then, use **clone()** to build a little thread library, with a **thread_create()** call and **lock_aquire()** and **lock_release()** functions.

Changes Made:

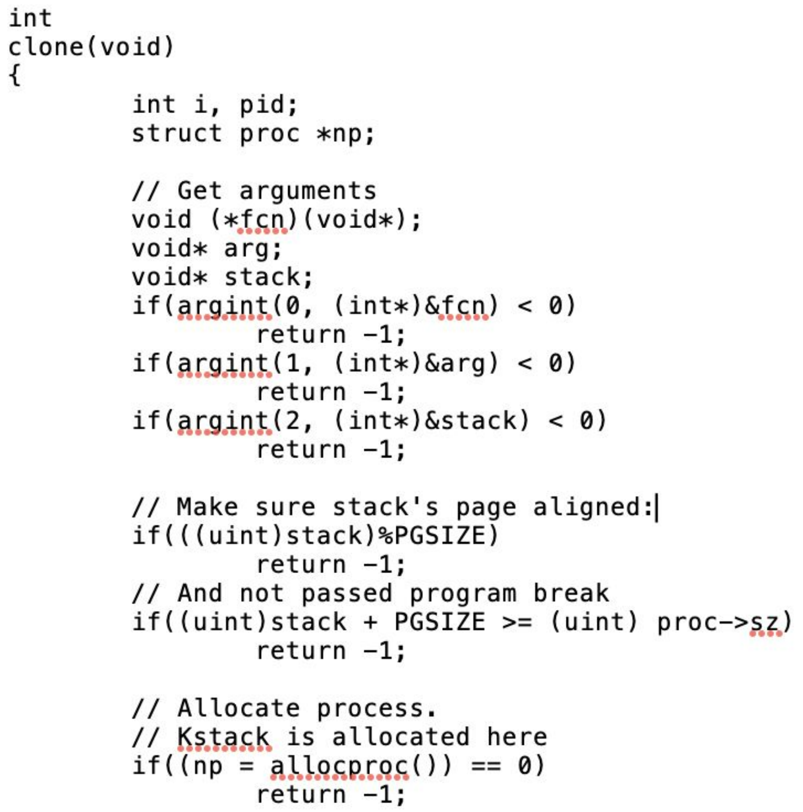
Proc.c

The structure **lock**: A lock or Mutex is a mechanism that is created for enforcing limits on access to a resource in an environment where there are many threads of execution.
This is to alleviate CPU usage when there are lots of threads or processes active at the same time.

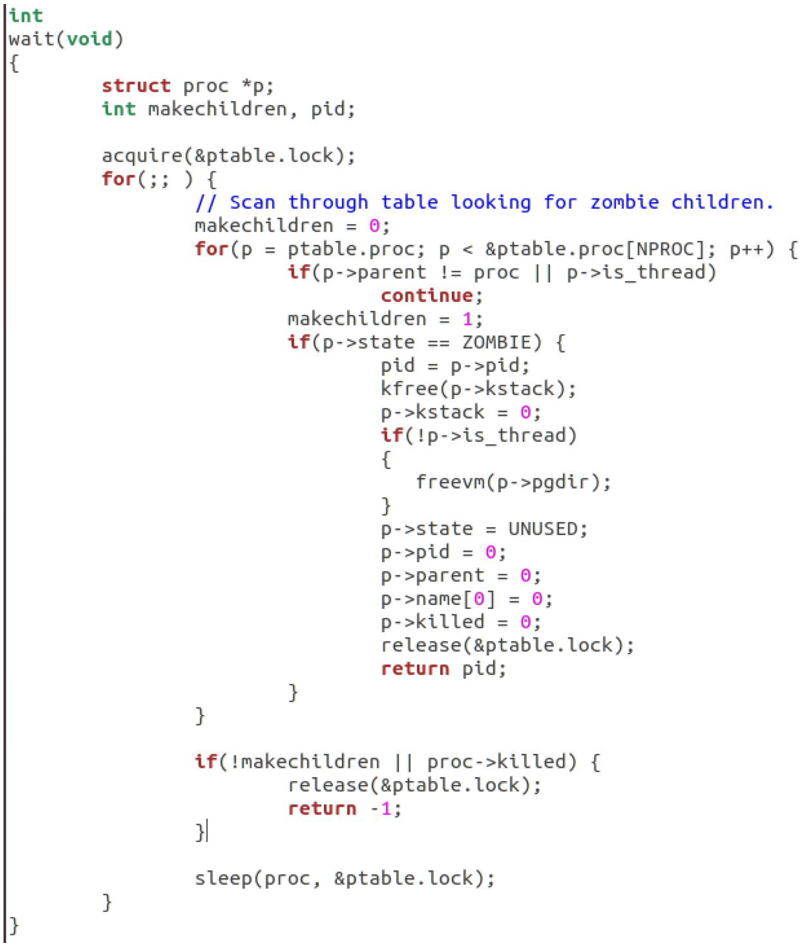
Define **clone()** system call which creates a new kernel thread which shares the calling process' address space.

Define **wait()** system call which should wait for a child process that does not share the address space with the process.

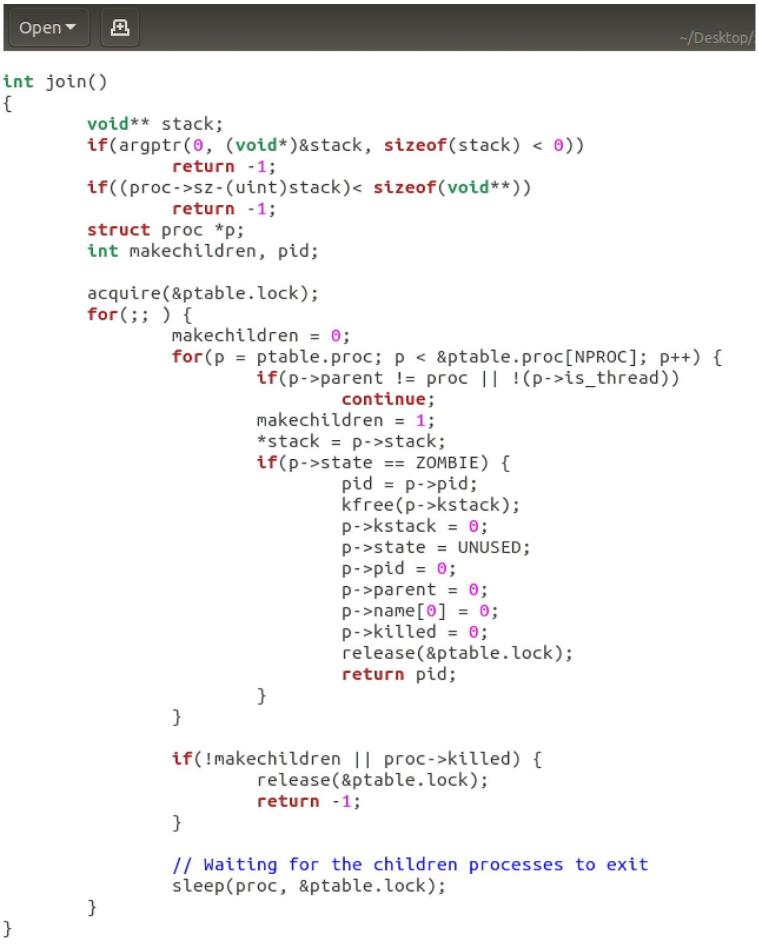
The **join()** sys call. This call waits for a child thread that shares the address space with the calling process. It returns the PID of waited for child or -1 if none.



```
int
clone(void)
{
        int i, pid;
        struct proc *np;

        // Get arguments
        void (*fcn)(void*);
        void* arg;
        void* stack;
        if(argint(0, (int*)&fcn) < 0)
                return -1;
        if(argint(1, (int*)&arg) < 0)
                return -1;
        if(argint(2, (int*)&stack) < 0)
                return -1;

        // Make sure stack's page aligned:
        if(((uint)stack)%PGSIZE)
                return -1;
        // And not passed program break
        if((uint)stack + PGSIZE >= (uint) proc->sz)
                return -1;

        // Allocate process.
        // Kstack is allocated here
        if((np = allocproc()) == 0)
                return -1;
```



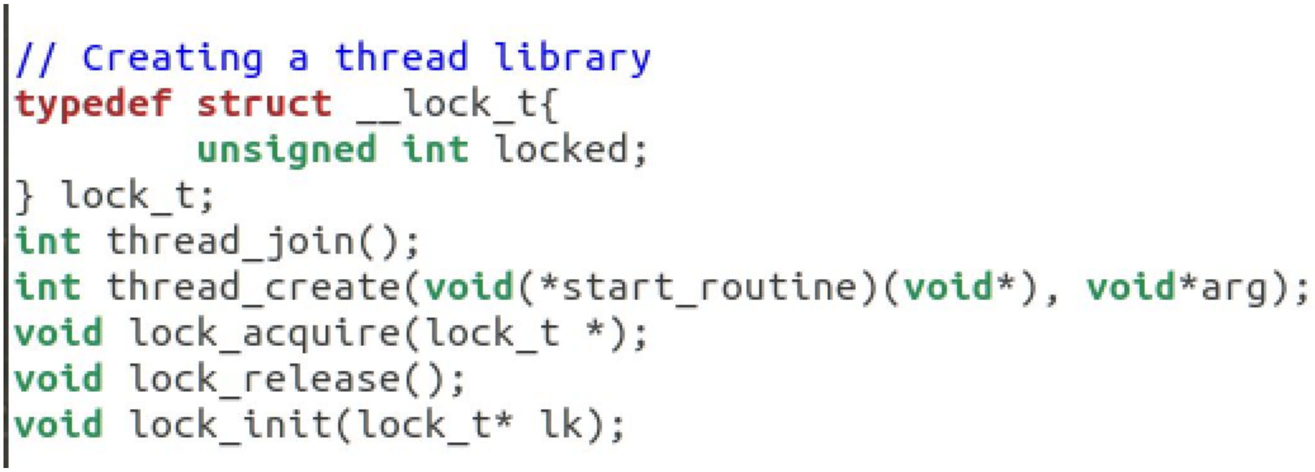
```
int
wait(void)
{
        struct proc *p;
        int makechildren, pid;

        acquire(&ptable.lock);
        for(;; ) {
                // Scan through table looking for zombie children.
                makechildren = 0;
                for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
                        if(p->parent != proc || p->is_thread)
                                continue;
                        makechildren = 1;
                        if(p->state == ZOMBIE) {
                                pid = p->pid;
                                kfree(p->kstack);
                                p->kstack = 0;
                                if(!p->is_thread)
                                {
                                   freevm(p->pgdir);
                                }
                                p->state = UNUSED;
                                p->pid = 0;
                                p->parent = 0;
                                p->name[0] = 0;
                                p->killed = 0;
                                release(&ptable.lock);
                                return pid;
                        }
                }

                if(!makechildren || proc->killed) {
                        release(&ptable.lock);
                        return -1;
                }

                sleep(proc, &ptable.lock);
        }
}
```



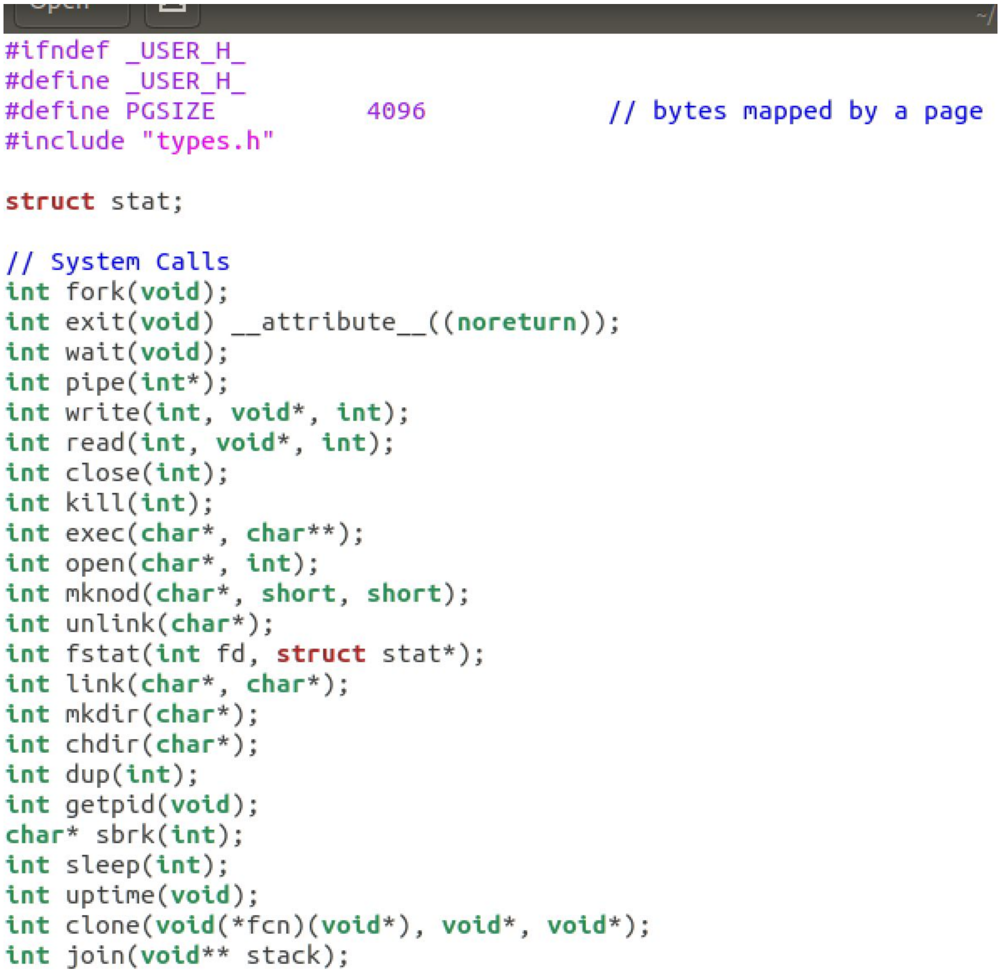
```
int join()
{
        void** stack;
        if(argptr(0, (void*)&stack, sizeof(stack) < 0))
                return -1;
        if((proc->sz-(uint)stack)< sizeof(void**))
                return -1;
        struct proc *p;
        int makechildren, pid;

        acquire(&ptable.lock);
        for(;; ) {
                makechildren = 0;
                for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
                        if(p->parent != proc || !(p->is_thread))
                                continue;
                        makechildren = 1;
                        *stack = p->stack;
                        if(p->state == ZOMBIE) {
                                pid = p->pid;
                                kfree(p->kstack);
                                p->kstack = 0;
                                p->state = UNUSED;
                                p->pid = 0;
                                p->parent = 0;
                                p->name[0] = 0;
                                p->killed = 0;
                                release(&ptable.lock);
                                return pid;
                        }
                }

                if(!makechildren || proc->killed) {
                        release(&ptable.lock);
                        return -1;
                }

                // Waiting for the children processes to exit
                sleep(proc, &ptable.lock);
        }
}
```

User.h

The **clone()**, **join()**, **lock()**, and **thread_create()** functions were added to the user.h file. This creates functionality within the user files.



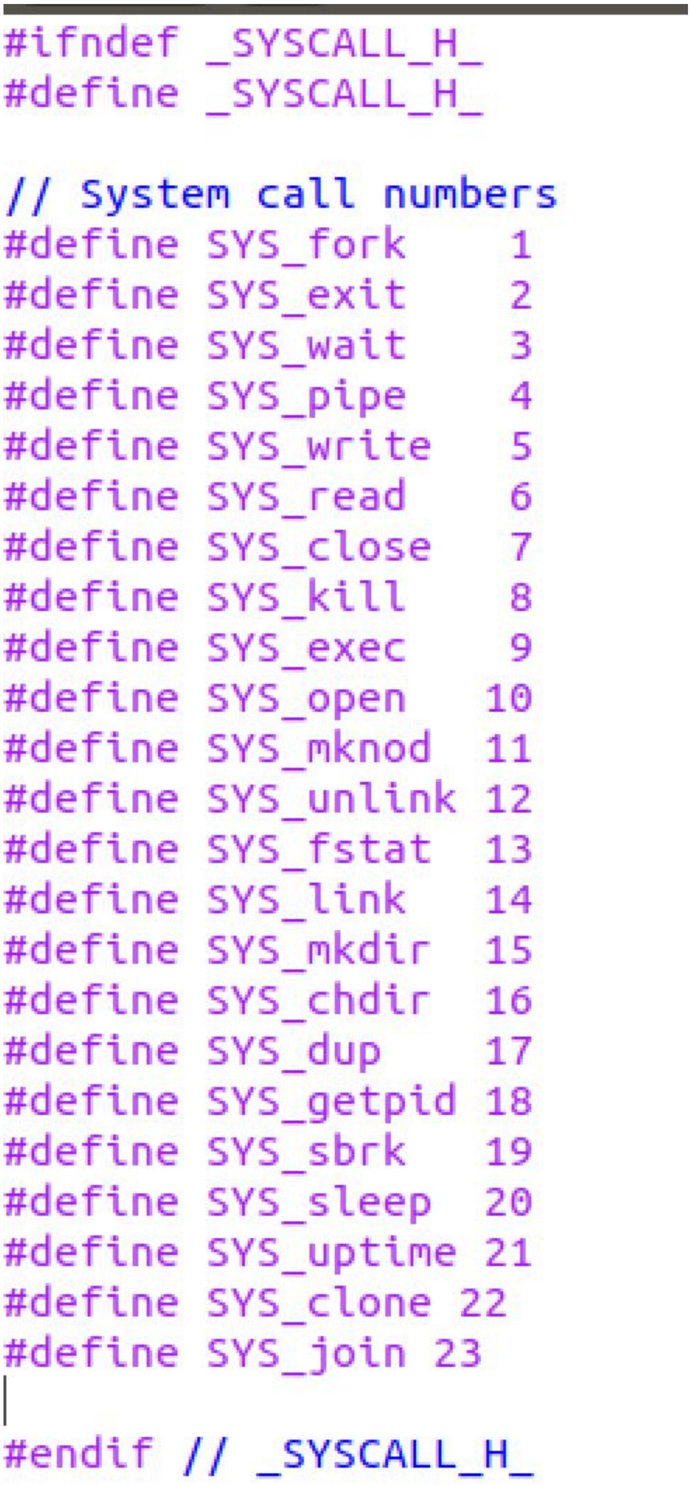
```
// Creating a thread library
typedef struct __lock_t{
        unsigned int locked;
} lock_t;
int thread_join();
int thread_create(void(*start_routine)(void*), void*arg);
void lock_acquire(lock_t *);
void lock_release();
void lock_init(lock_t* lk);
```



```
#ifndef _USER_H_
#define _USER_H_
#define PGSIZE          4096            // bytes mapped by a page
#include "types.h"

struct stat;

// System Calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(char*, int);
int mknod(char*, short, short);
int unlink(char*);
int fstat(int fd, struct stat*);
int link(char*, char*);
int mkdir(char*);
int chdir(char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int clone(void(*fcn)(void*), void*, void*);
int join(void** stack);
```

Syscall.h

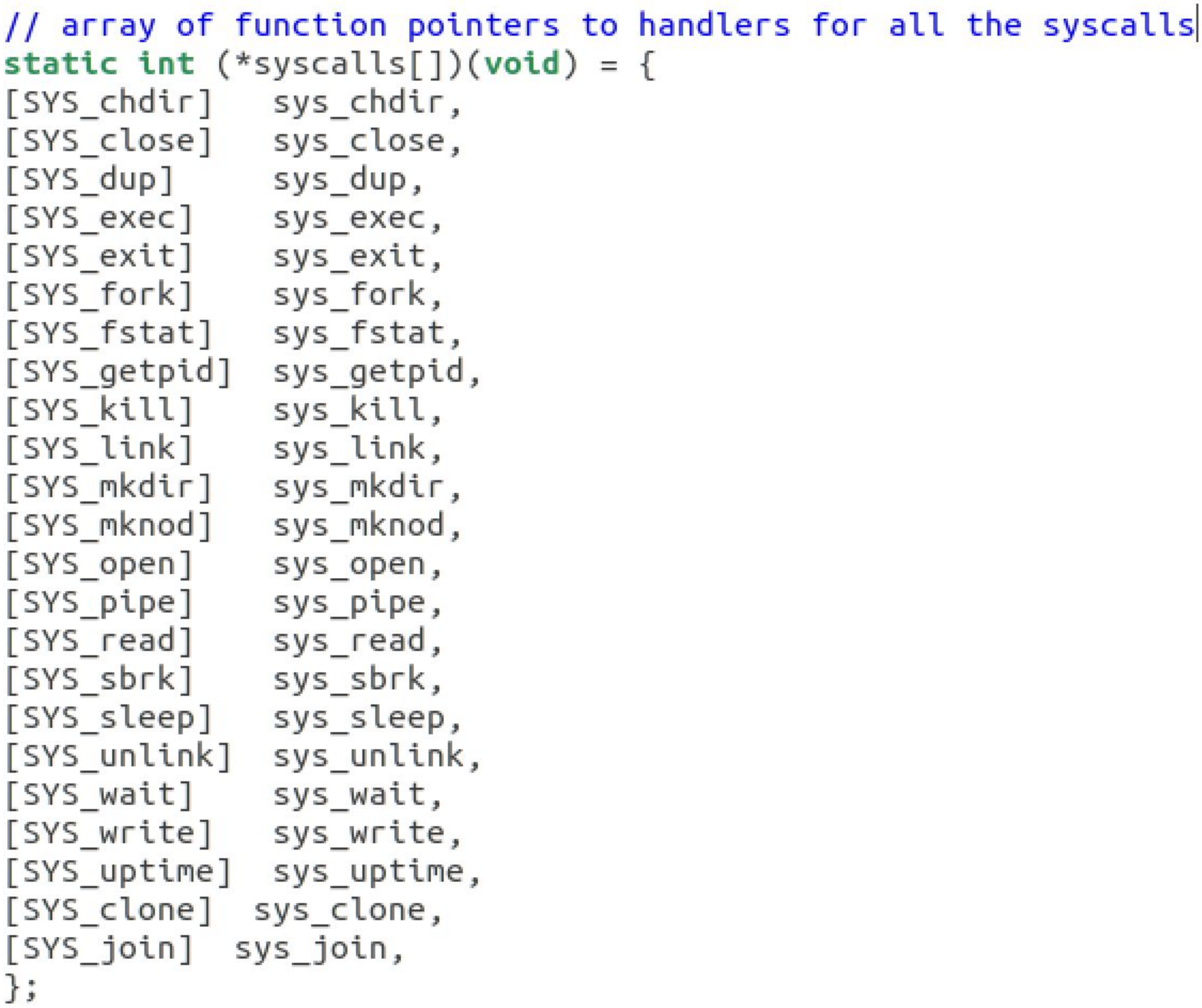
System calls are defined for **clone()** and **join()**.



```
#ifndef _SYSCALL_H_
#define _SYSCALL_H_

// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_write   5
#define SYS_read    6
#define SYS_close   7
#define SYS_kill    8
#define SYS_exec    9
#define SYS_open   10
#define SYS_mknod  11
#define SYS_unlink 12
#define SYS_fstat  13
#define SYS_link   14
#define SYS_mkdir  15
#define SYS_chdir  16
#define SYS_dup    17
#define SYS_getpid 18
#define SYS_sbrk   19
#define SYS_sleep  20
#define SYS_uptime 21
#define SYS_clone 22
#define SYS_join 23

#endif // _SYSCALL_H_
```

Syscall.c

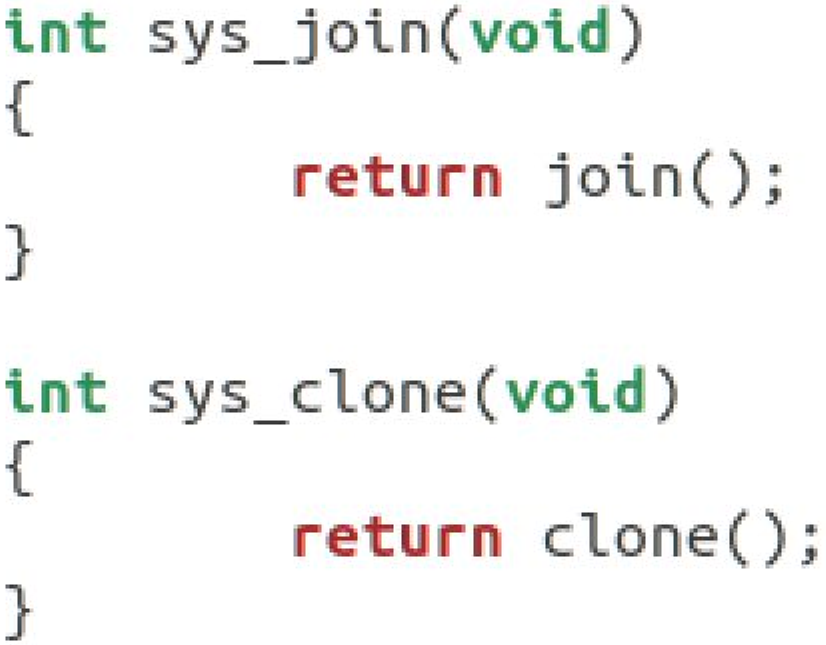
System calls are added for **clone()** and **join()**.



```
// array of function pointers to handlers for all the syscalls
static int (*syscalls[])(void) = {
[SYS_chdir]   sys_chdir,
[SYS_close]   sys_close,
[SYS_dup]     sys_dup,
[SYS_exec]    sys_exec,
[SYS_exit]    sys_exit,
[SYS_fork]    sys_fork,
[SYS_fstat]   sys_fstat,
[SYS_getpid]  sys_getpid,
[SYS_kill]    sys_kill,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_mknod]   sys_mknod,
[SYS_open]    sys_open,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_unlink]  sys_unlink,
[SYS_wait]    sys_wait,
[SYS_write]   sys_write,
[SYS_uptime]  sys_uptime,
[SYS_clone]  sys_clone,
[SYS_join]  sys_join,
};
```

Sysproc.c

Created **sys_clone**, and **sys_join** to initialize the system calls.

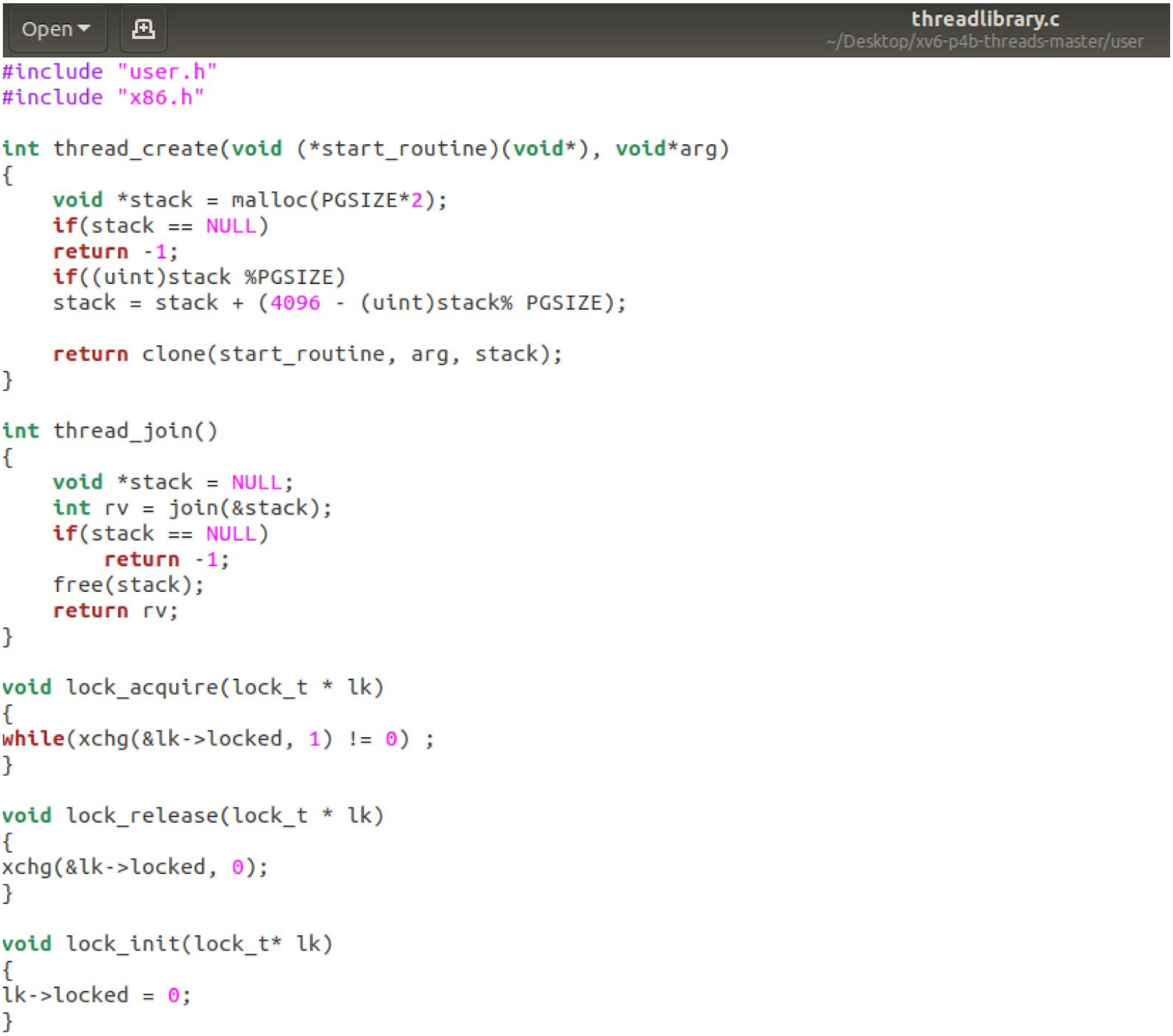


```
int sys_join(void)
{
        return join();
}

int sys_clone(void)
{
        return clone();
}
```

Threadlibrary.c

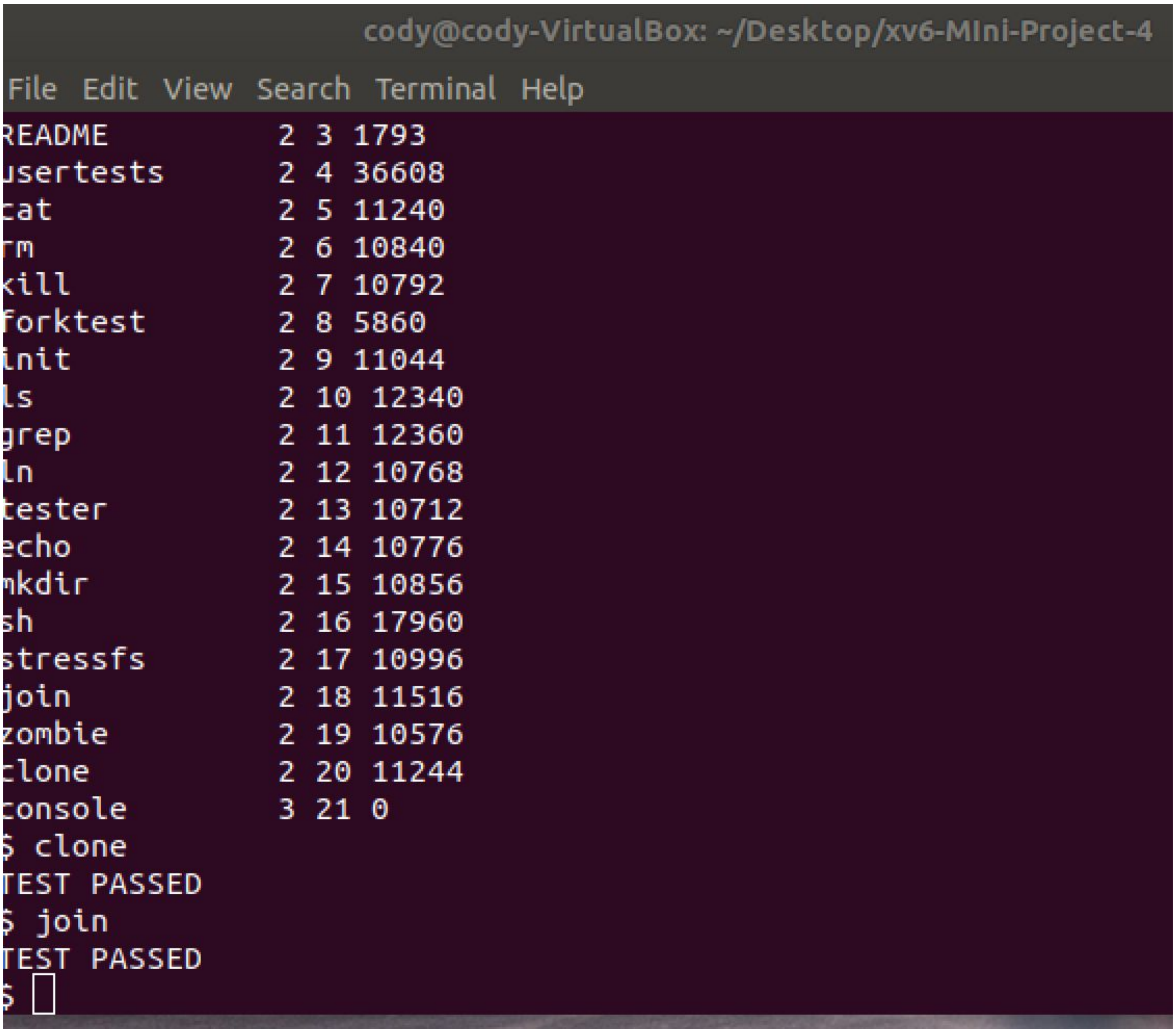
The updated lock structure prevents threads from overwriting performance critical tasks. The processes within create a thread library using the **Thread_create()** routine. This routine should call **malloc()** to create a new user stack, use **clone()** to create the child thread and get it running.

The **Thread_join()** call is also used, which calls the underlying **join()** system call, frees the user stack, and then returns.



```c
#include "user.h"
#include "x86.h"

int thread_create(void (*start_routine)(void*), void*arg)
{
    void *stack = malloc(PGSIZE*2);
    if(stack == NULL)
    return -1;
    if((uint)stack %PGSIZE)
    stack = stack + (4096 - (uint)stack% PGSIZE);

    return clone(start_routine, arg, stack);
}

int thread_join()
{
    void *stack = NULL;
    int rv = join(&stack);
    if(stack == NULL)
        return -1;
    free(stack);
    return rv;
}

void lock_acquire(lock_t * lk)
{
while(xchg(&lk->locked, 1) != 0) ;
}

void lock_release(lock_t * lk)
{
xchg(&lk->locked, 0);
}

void lock_init(lock_t* lk)
{
lk->locked = 0;
}
```

<u>Output</u>



```
README         2 3 1793
usertests      2 4 36608
cat            2 5 11240
rm             2 6 10840
kill           2 7 10792
forktest       2 8 5860
init           2 9 11044
ls             2 10 12340
grep           2 11 12360
ln             2 12 10768
tester         2 13 10712
echo           2 14 10776
mkdir          2 15 10856
sh             2 16 17960
stressfs       2 17 10996
join           2 18 11516
zombie         2 19 10576
clone          2 20 11244
console        3 21 0
$ clone
TEST PASSED
$ join
TEST PASSED
$ 
```