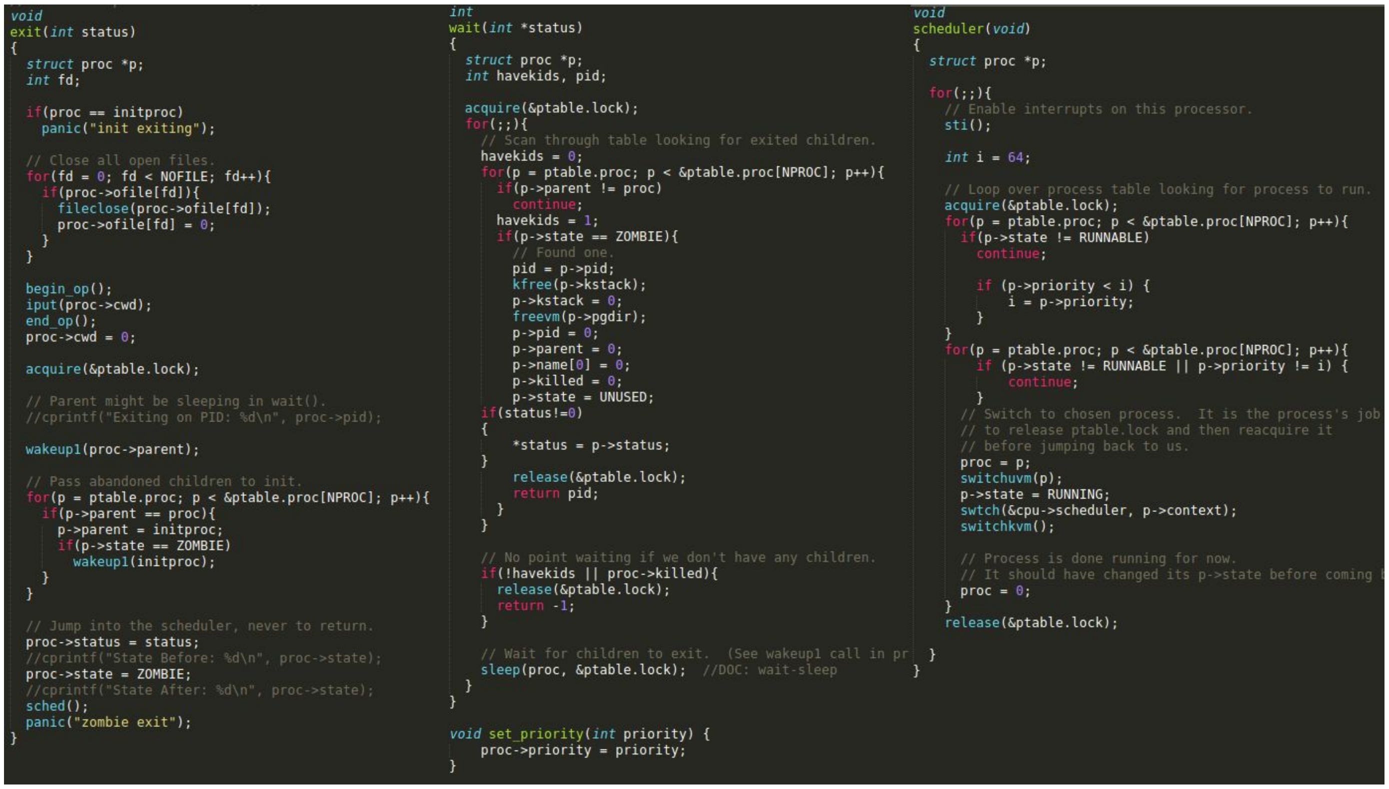
Samuel Hwang (861152953)
Michael Pena (861167712)

CS153 Lab 1 Report

Lab 1 taught us how to add new system calls to xv6 as well as change the scheduler from a round-robin, to a priority scheduler. We saw how xv6 uses multiple files to communicate and call on the system calls. Knowing how to create new system calls is the foundation of changing and adding new features to the operating system. We tested correctness by using the test file given and modifying that to check for certain outputs depending on the functions being tested.

Files Changed:
proc.c
proc.h
usys.S
syscall.h
syscall.c
user.h
defs.h
sysproc.c

Screenshots: proc.c



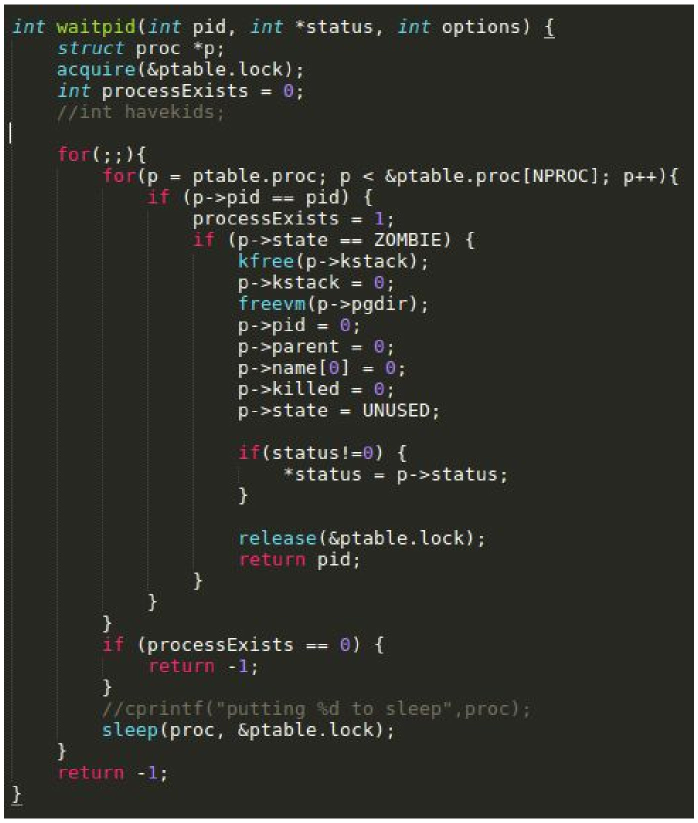
```c
void
exit(int status)
{
  struct proc *p;
  int fd;

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

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(proc->ofile[fd]){
      fileclose(proc->ofile[fd]);
      proc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(proc->cwd);
  end_op();
  proc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  //cprintf("Exiting on PID: %d\n", proc->pid);

  wakeup1(proc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == proc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  proc->status = status;
  //cprintf("State Before: %d\n", proc->state);
  proc->state = ZOMBIE;
  //cprintf("State After: %d\n", proc->state);
  sched();
  panic("zombie exit");
}
```

```c
int
wait(int *status)
{
  struct proc *p;
  int havekids, pid;

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

    // No point waiting if we don't have any children.
    if(!havekids || proc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in pr
    sleep(proc, &ptable.lock);  //DOC: wait-sleep
  }
}

void set_priority(int priority) {
    proc->priority = priority;
}
```

```c
void
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    int i = 64;

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

        if (p->priority < i) {
            i = p->priority;
        }
    }
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if (p->state != RUNNABLE || p->priority != i) {
            continue;
        }
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming
    proc = 0;
    }
    release(&ptable.lock);

  }
}
```

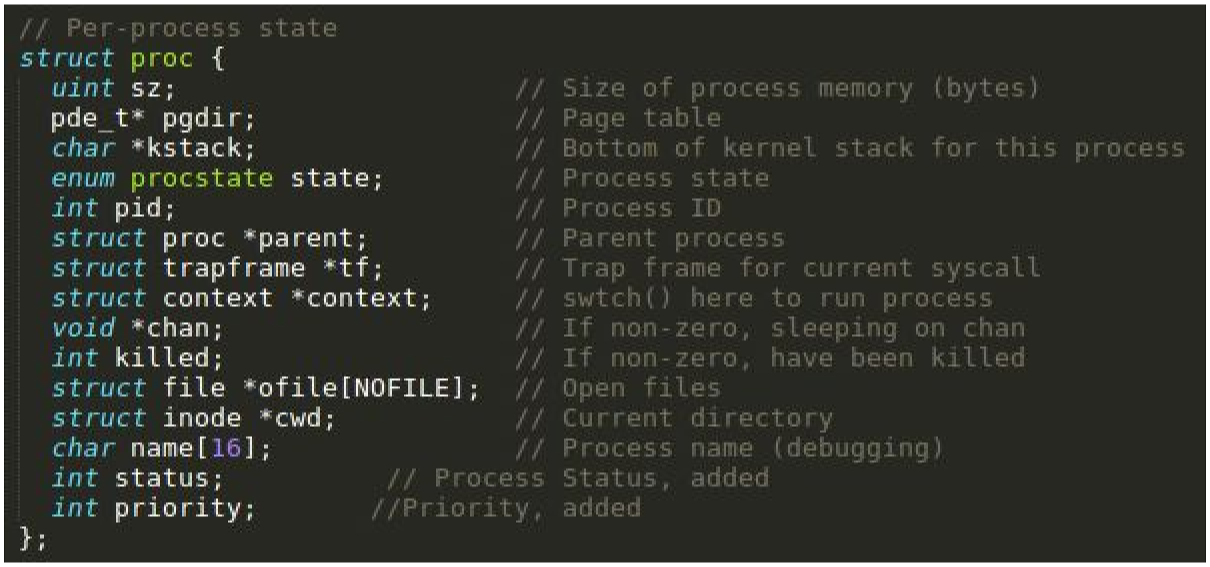
Samuel Hwang (861152953)
Michael Pena (861167712)

Proc.c



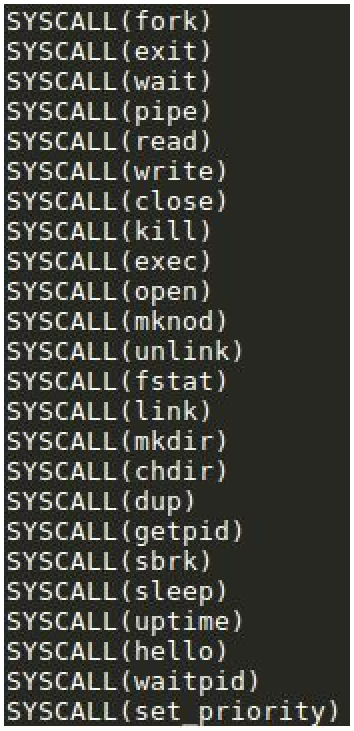
```
int waitpid(int pid, int *status, int options) {
    struct proc *p;
    acquire(&ptable.lock);
    int processExists = 0;
    //int havekids;

    for(;;){
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if (p->pid == pid) {
                processExists = 1;
                if (p->state == ZOMBIE) {
                    kfree(p->kstack);
                    p->kstack = 0;
                    freevm(p->pgdir);
                    p->pid = 0;
                    p->parent = 0;
                    p->name[0] = 0;
                    p->killed = 0;
                    p->state = UNUSED;

                    if(status!=0) {
                        *status = p->status;
                    }

                    release(&ptable.lock);
                    return pid;
                }
            }
        }
        if (processExists == 0) {
            return -1;
        }
        //cprintf("putting %d to sleep",proc);
        sleep(proc, &ptable.lock);
    }
    return -1;
}
```

Proc.h



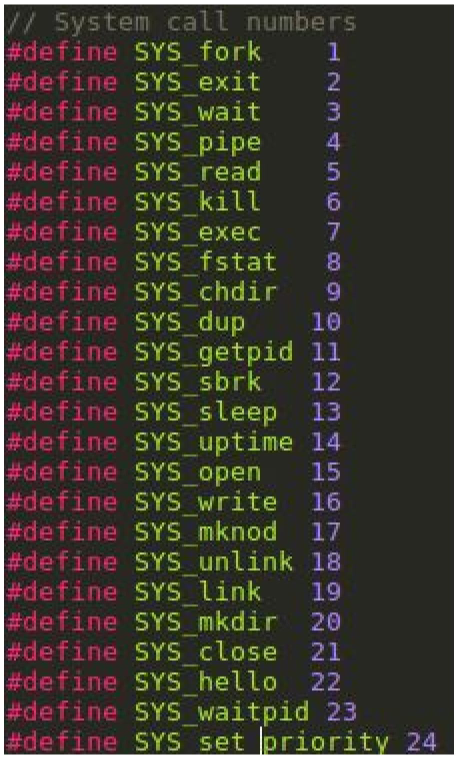
```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int status;          // Process Status, added
  int priority;       //Priority, added
};
```

Samuel Hwang (861152953)
Michael Pena (861167712)



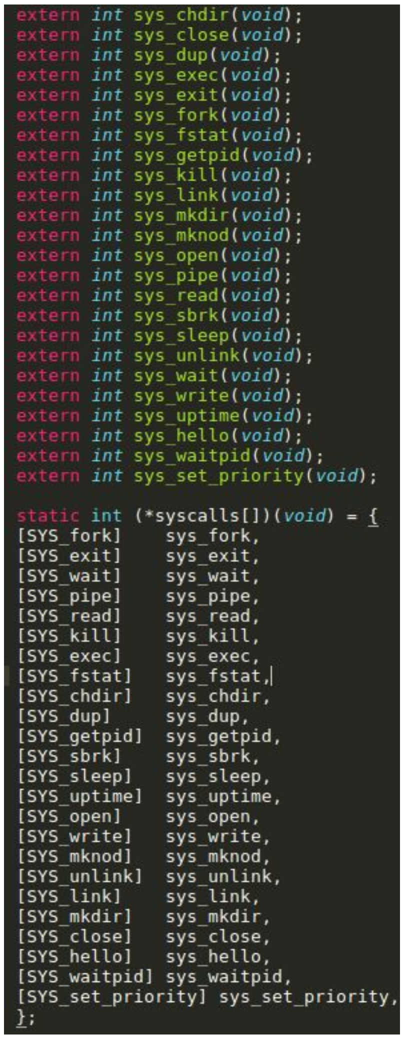
```
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(hello)
SYSCALL(waitpid)
SYSCALL(set_priority)
```

Usys.S



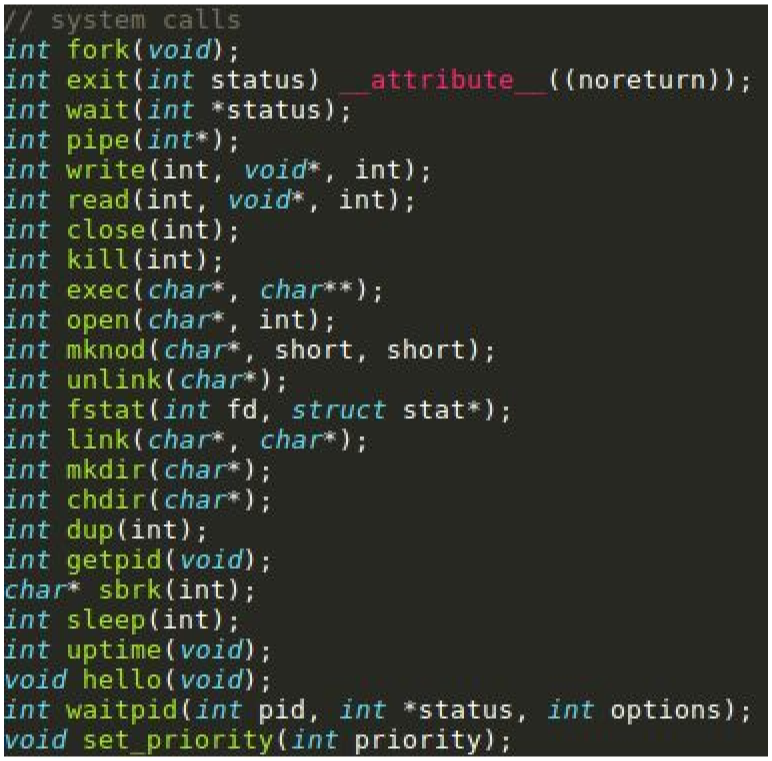
```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_hello  22
#define SYS_waitpid 23
#define SYS_set_priority 24
```

Syscall.h



```
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_hello(void);
extern int sys_waitpid(void);
extern int sys_set_priority(void);

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_hello]   sys_hello,
[SYS_waitpid] sys_waitpid,
[SYS_set_priority] sys_set_priority,
};
```

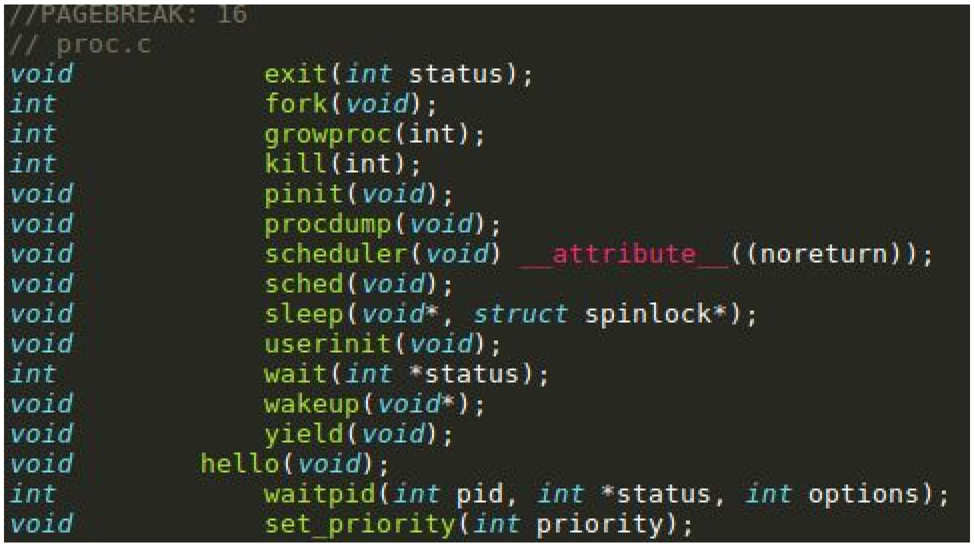
Syscall.c



```
// system calls
int fork(void);
int exit(int status) __attribute__((noreturn));
int wait(int *status);
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);
void hello(void);
int waitpid(int pid, int *status, int options);
void set_priority(int priority);
```

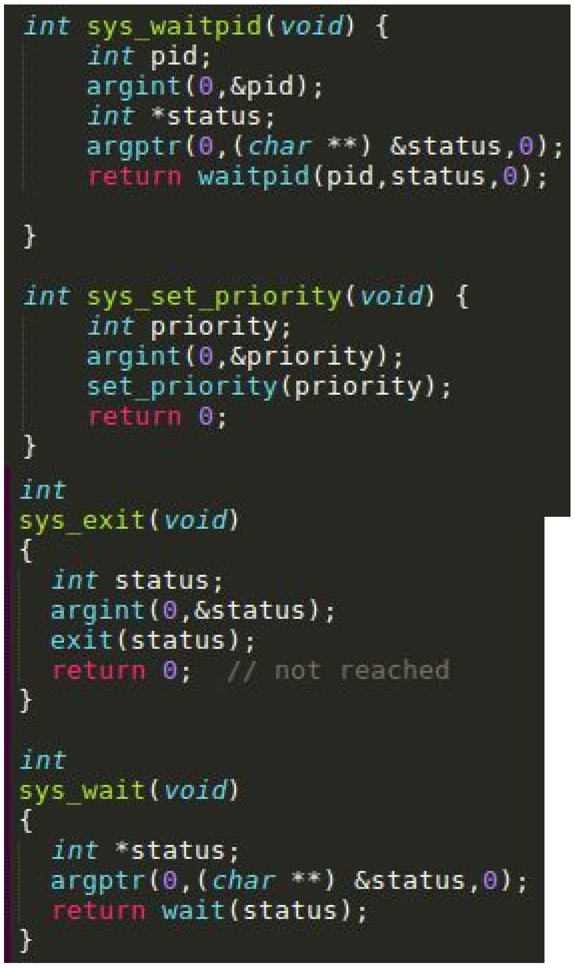
User.h

Samuel Hwang (861152953)
Michael Pena (861167712)



```
//PAGEBREAK: 16
// proc.c
void            exit(int status);
int             fork(void);
int             growproc(int);
int             kill(int);
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(int *status);
void            wakeup(void*);
void            yield(void);
void        hello(void);
int             waitpid(int pid, int *status, int options);
void            set_priority(int priority);
```

Defs.h



```
int sys_waitpid(void) {
    int pid;
    argint(0,&pid);
    int *status;
    argptr(0,(char **) &status,0);
    return waitpid(pid,status,0);

}

int sys_set_priority(void) {
    int priority;
    argint(0,&priority);
    set_priority(priority);
    return 0;
}
int
sys_exit(void)
{
  int status;
  argint(0,&status);
  exit(status);
  return 0;  // not reached
}

int
sys_wait(void)
{
  int *status;
  argptr(0,(char **) &status,0);
  return wait(status);
}
```

Sysproc.c