Name: ANUJA PATIL
Student ID: 862123752

1) Change exit implementation to maintain exit status.
Approach: I have added another implementation rather than modifying the existing exit(). The name of newly added exit is **exitUDef()**. To make the required changes, I changed some files. Following are the files changed and a brief summary of the changes done in each:
a) syscall.h
Defined a system call and associated it with a number:
#define SYS_exitUDef 22

b) syscall.c
Added the function prototype for exitUDef() – **extern int sys_exitUDef(void);**
Added a pointer to the system call in the array of function pointers:

```
static int(*syscalls[]) (void) = {
   .
   .
   .
   [SYS_exitUDef]    sys_exitUDef,
}
```

c) usys.S
In order for a user program to call the system call, an interface was added:
SYS_CALL(exitUDef)

d) sysproc.c
Added the handler for exitUDef system call to maintain exit status:

```
int sys_exitUDef(void) {
   int exitStatus;
   if(argint(0, &exitStatus) < 0) return -1;
   return exitUDef(exitStatus);
}
```

e) proc.h
Added a field called exitStatus in the proc struct which stores the state of a process:

```
struct proc {
   .
   .
   int exitStatus;
}
```

f) proc.c

This contains the actual implementation of the exitUDef(). This is same as original exit() except that in exitUDef(), we also store status received as an argument in the current process's exitStatus state:
curproc->exitStatus = status;
where status is the argument passed to exitUDef().

g) defs.h
Added forward declaration of system call exitUDef() in "proc.c" section:
int exitUDef(int);

h) user.h
Added user level definition of exitUDef() under system calls:
int exitUDef(int);

2) Update the wait system call signature to int wait(int *status).
Approach: Updated the existing wait() to contain the pointer to integer argument 'status'. Originatl wait() returns pid of the process that current process was waiting on. If we want to return another value, in our case exit status of the process too, then we pass the address of a variable to wait() so that the exit status of the process can be stored in the passed argument. To make the required changes, I changed some files. Following are the files changed and a brief summary of the changes done in each:
a) sysproc.c
Added a field 'status' which is an integer pointer. Called argptr() to pass the address of 'status':

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

b) proc.c
Modified the wait() method to contain the pointer to integer argument 'status'. If 'status' is not null (which indicates that the user also wants to return the exit status of the process), we store the process's exitStatus in 'status' field:

```
int wait(int* status) {
  .
  .
  if(p->state == ZOMBIE) {
    .
    .
    p->state = UNUSED;
    if(status) *status = p->exitStatus;
    p->exitStatus = 0; //reset to 0
    .
}
```

c) defs.h
Modified forward declaration of system call wait() in “proc.c” section:
int wait(int*);

d) user.h
Modified user level definition of wait() under system calls:
int wait(int*);

3) Add a waitpid system call: int waitpid(int pid, int *status, int options).
Approach: Added implementation for **waitpid(int pid, int* status, int options)**. To make the required changes, I changed some files. Following are the files changed and a brief summary of the changes done in each:
a) syscall.h
Defined a system call and associated it with a number:
#define SYS_waitpid 23

b) syscall.c
Added the function prototype for waitpid() – **extern int sys_waitpid(void);**
Added a pointer to the system call in the array of function pointers:

```
static int(*syscalls[]) (void) = {
   .
   .
   .
   [SYS_ waitpid]    sys_waitpid,
}
```

c) usys.S
In order for a user program to call the system call, an interface was added:
SYS_CALL(waitpid)

d) sysproc.c
Added the handler for waitpid system call to maintain exit status:

```
int sys_ waitpid (void) {
   int pid, options = 0;
   int* status;
   if(argint(0, &pid) < 0) return -1;
   if(argptr(1, (void*)&status, sizeof(status)) < 0) return -1;
   return waitpid(pid, status, options);
}
```

e) defs.h
Added forward declaration of system call waitpid() in “proc.c” section:
int waitpid(int, int*, int);

f) user.h
Added user level definition of waitpid() under system calls:
int waitpid(int, int*, int);

g) proc.c
This contains the actual implementation of the waitpid(). This is similar to the modified wait() except that in waitpid(), we are not looping for the current process's children only. We are looping over all the processes in the process table and looking for the process whose pid matches with the one sent as a parameter to the waitpid(). I have renamed 'havekids' to 'processExists':

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
  if(p->pid != pid) continue;
  processExists = 1;
  if(p->state == ZOMBIE) {
    .
    .
    p->state = UNUSED;
    if(status) *status = p->exitStatus;
    p->exitStatus = 0;
    release(&ptable.lock);
    return pid;
  }
  .
  .
}
```

4) Modified the lab1_testfile.c to use exitUDef(x) in place of exit(x) where x is an exit status in the program. I have also changed **wait()** to **wait(0)** in files – forktest.c, init.c, sh.c, stressfs.c, usertests.c. 0 parameter in wait() indicates that we are passing null. In C, 0 means null.

5) BONUS – Implement WNOHANG and create version of CELEBW02 on same page that check of a child process still running.
Approach: lab1_testfile.c has the WNOHANG defined to be 1. I changed some files to implement WNOHANG as required in the assignment. Following are the files changed and a brief summary of the changes done in each:
a) sysproc.c
We need to check if any options argument has been passed to the method in waitpid(). By default, options takes the value 0. For this assignment, according to lab1_testfile, options can take the value 1:

```
int sys_ waitpid (void) {
  int pid, options = 0;
  int* status;
  if(argint(0, &pid) < 0) return -1;
  if(argptr(1, (void*)&status, sizeof(status)) < 0) return -1;
```

```
  if(argint(2, &options) < 0) return -1;
  return waitpid(pid, status, options);
}
```

b) proc.c

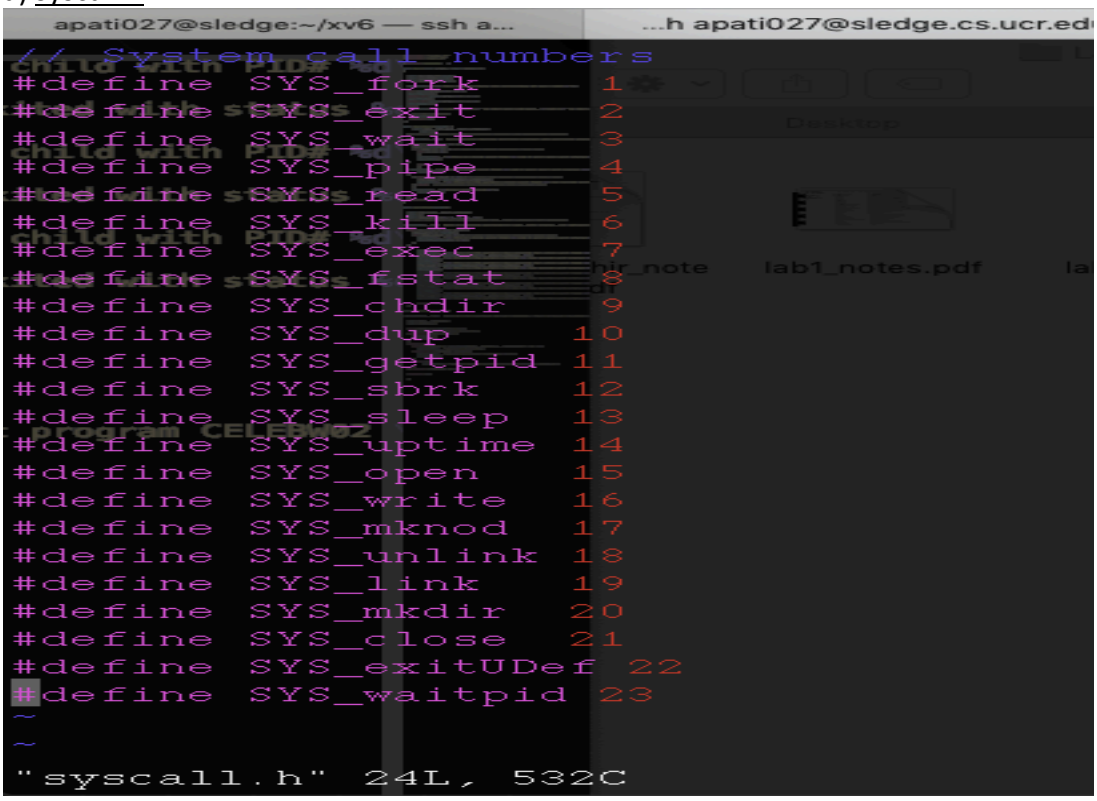
We just want to implement WNOHANG such that lab1_testfile.c runs. In this test file, for the bonus part, we call waitpid(pid, &status, WNOHANG) where WNOHANG is defined to be 1. If the waitpid returns -1, there is an error with wait(); if it returns pid, it means the process with given pid has terminated; if it returns 0, it means the process with given pid is still running and since we do not want to make the current process wait, waitpid() should return 0. In proc.c, we add an 'else-if' block to the 'if':

```
if(p->state == ZOMBIE) {
  .
  .
} else if(options == 1) {
  release(&ptable.lock)l
  return 0;
}
```

The control reaches the 'else-if' block when the process with given pid exists, is not in ZOMBIE state and options is 1.

6) Following are the screenshots of the parts of code in the files mentioned above which I have changed:

a) syscall.h

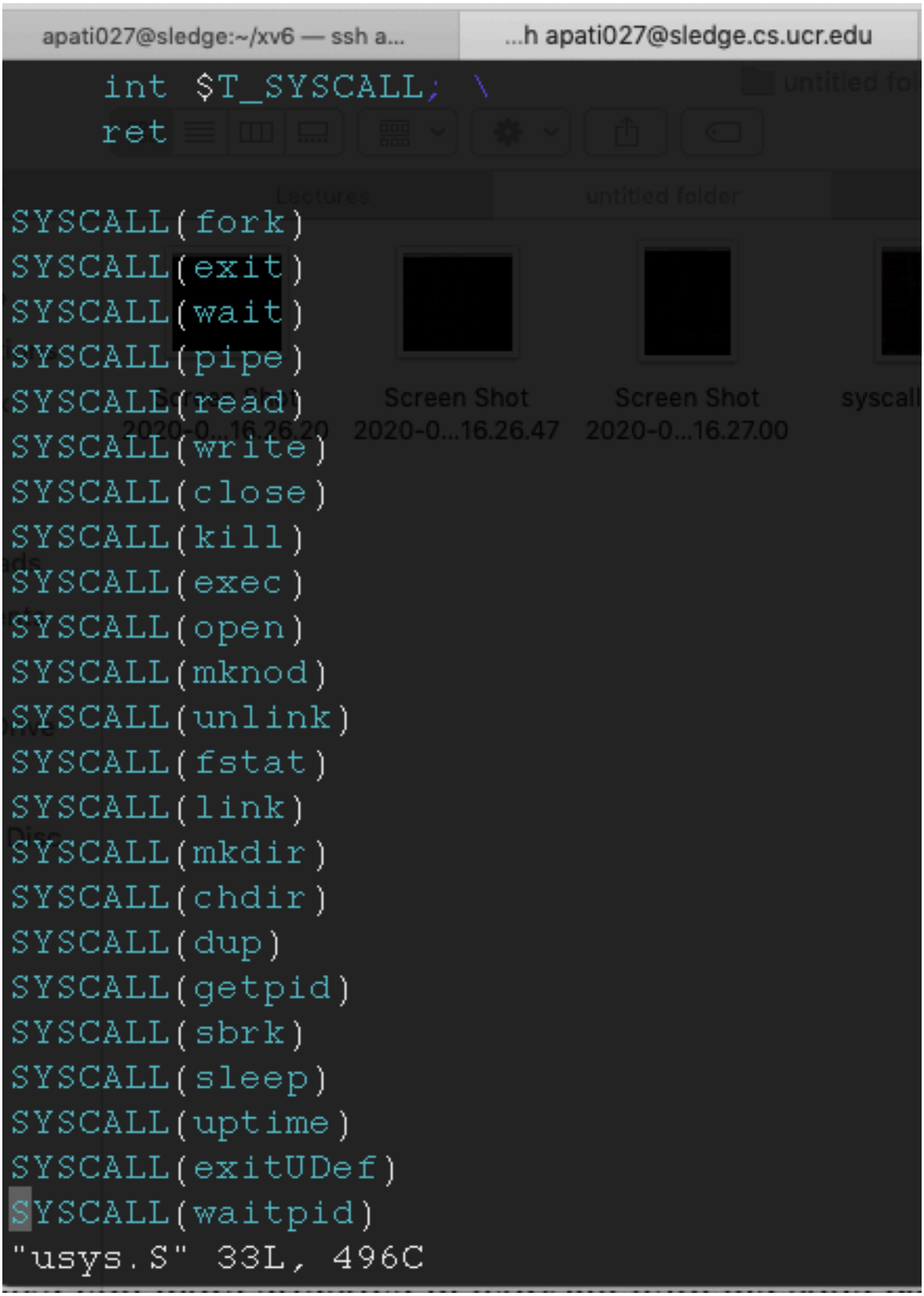


b) syscall.c

```
apati027@sledge:~/xv6 — ssh a...    ...h apati027@sledge.cs.u
extern int sys_exitUDef(void);
extern int sys_waitpid(void);
```

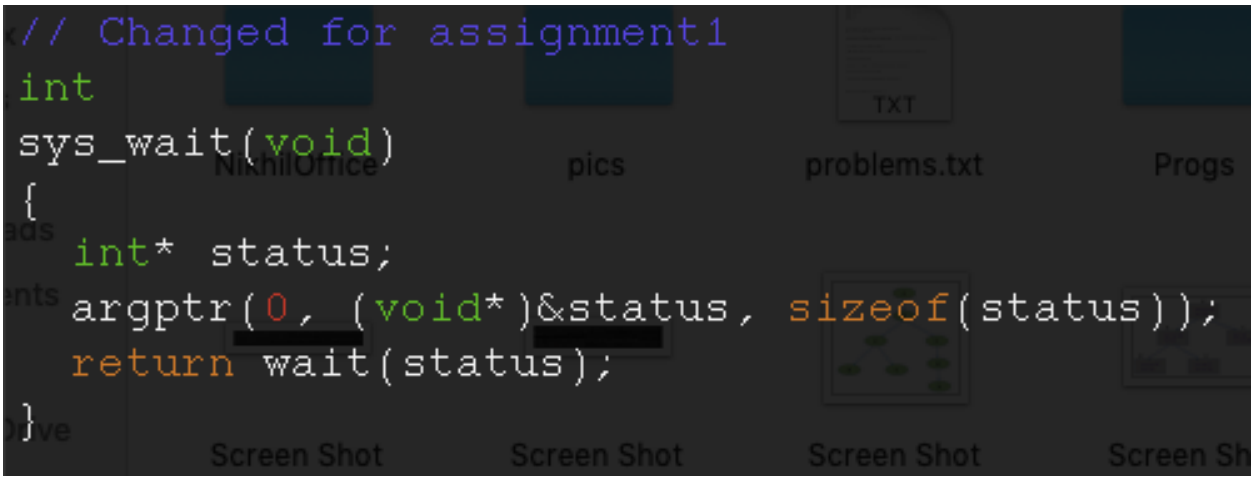
```
apati027@sledge:~/xv6 — ssh a...    ...h apati027@sledge.cs.ucr.edu
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   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_exitUDef]    sys_exitUDef,
[SYS_waitpid]   sys_waitpid,
};
```

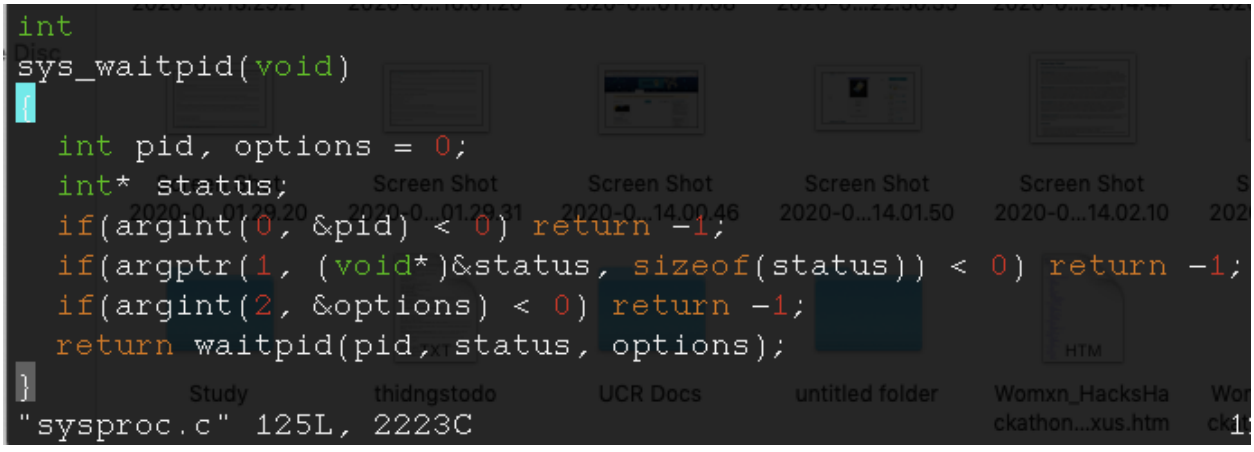
c) usys.S


apati027@sledge:~/xv6 — ssh a...
...h apati027@sledge.cs.ucr.edu
int $T_SYSCALL; \
ret
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(exitUDef)
SYSCALL(waitpid)
"usys.S" 33L, 496C

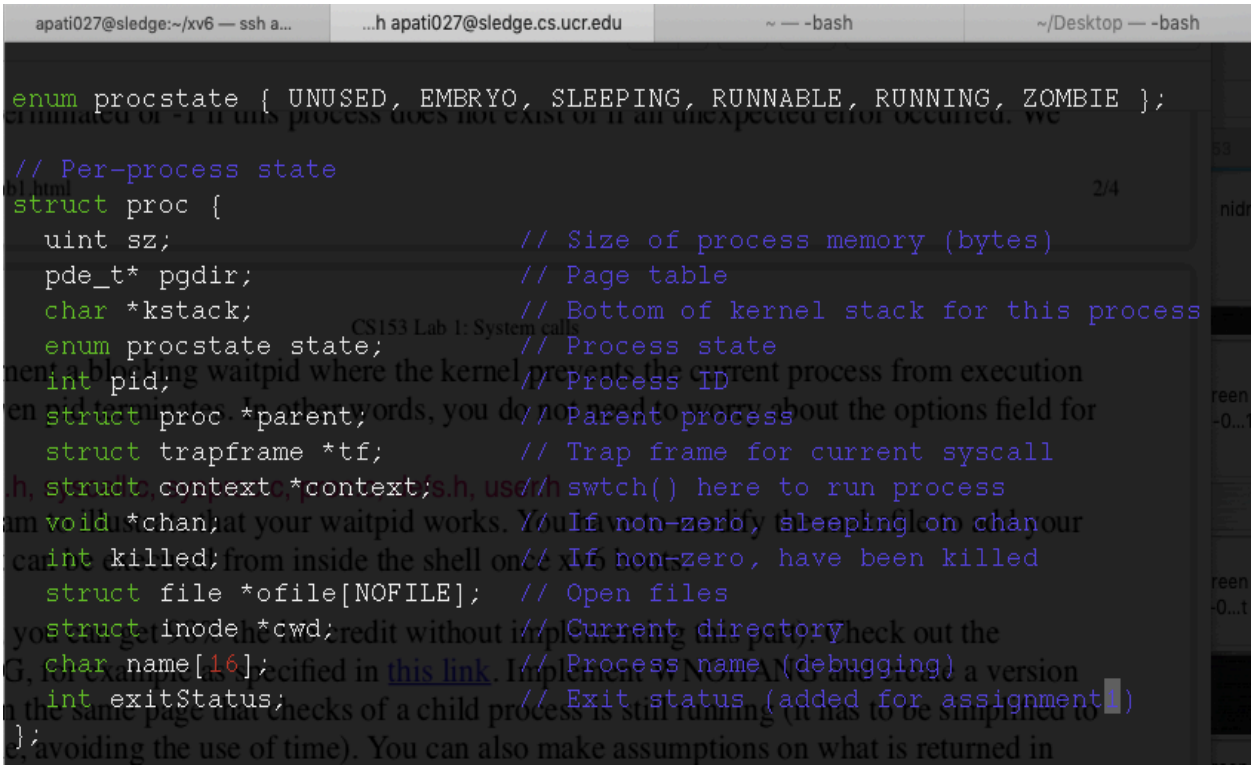
d) sysproc.c


apati027@sledge:~/xv6 — ssh a...
...h apati027@sledge.cs.ucr.edu
int
sys_exitUDef(void)
{
int exitStatus;
if(argint(0, &exitStatus) < 0) return -1;
return exitUDef(exitStatus);
}


// Changed for assignment1
int
sys_wait(void)
{
int* status;
argptr(0, (void*)&status, sizeof(status));
return wait(status);
}

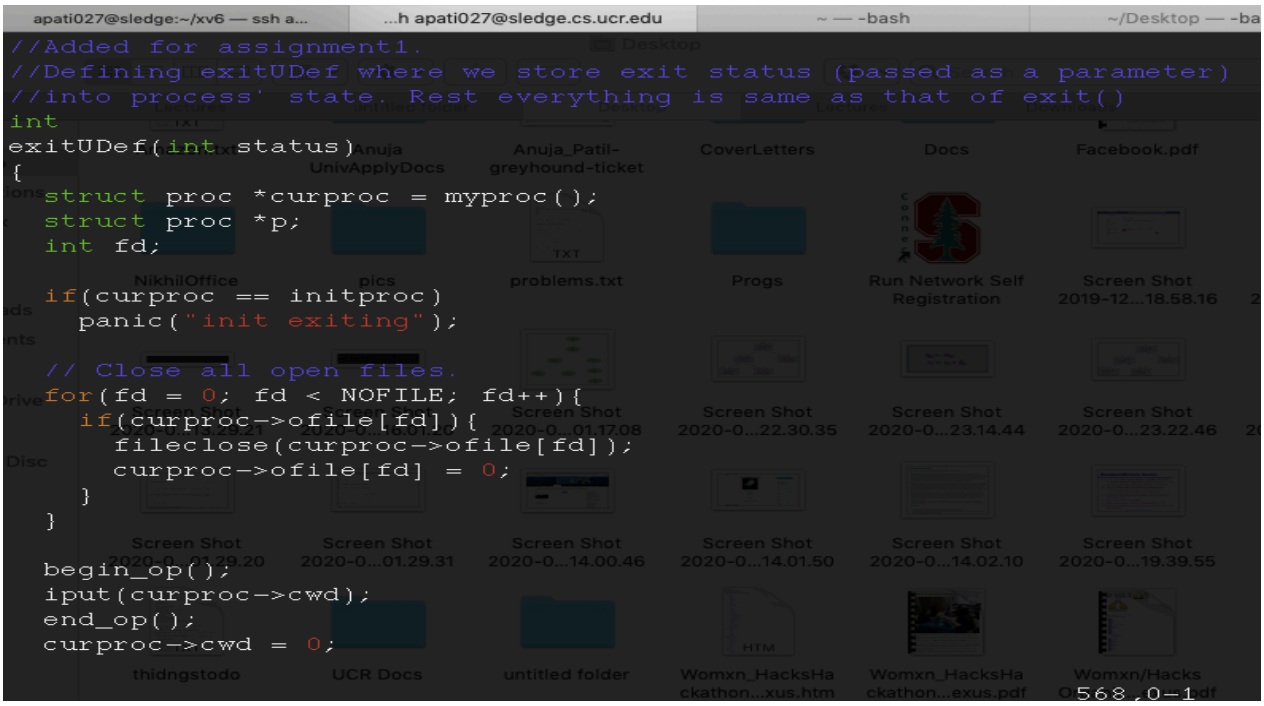

int
sys_waitpid(void)
{
int pid, options = 0;
int* status;
if(argint(0, &pid) < 0) return -1;
if(argptr(1, (void*)&status, sizeof(status)) < 0) return -1;
if(argint(2, &options) < 0) return -1;
return waitpid(pid,status, options);
}
"sysproc.c" 125L, 2223C

e) proc.h



```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// 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 exitStatus;              // Exit status (added for assignment1)
};
```

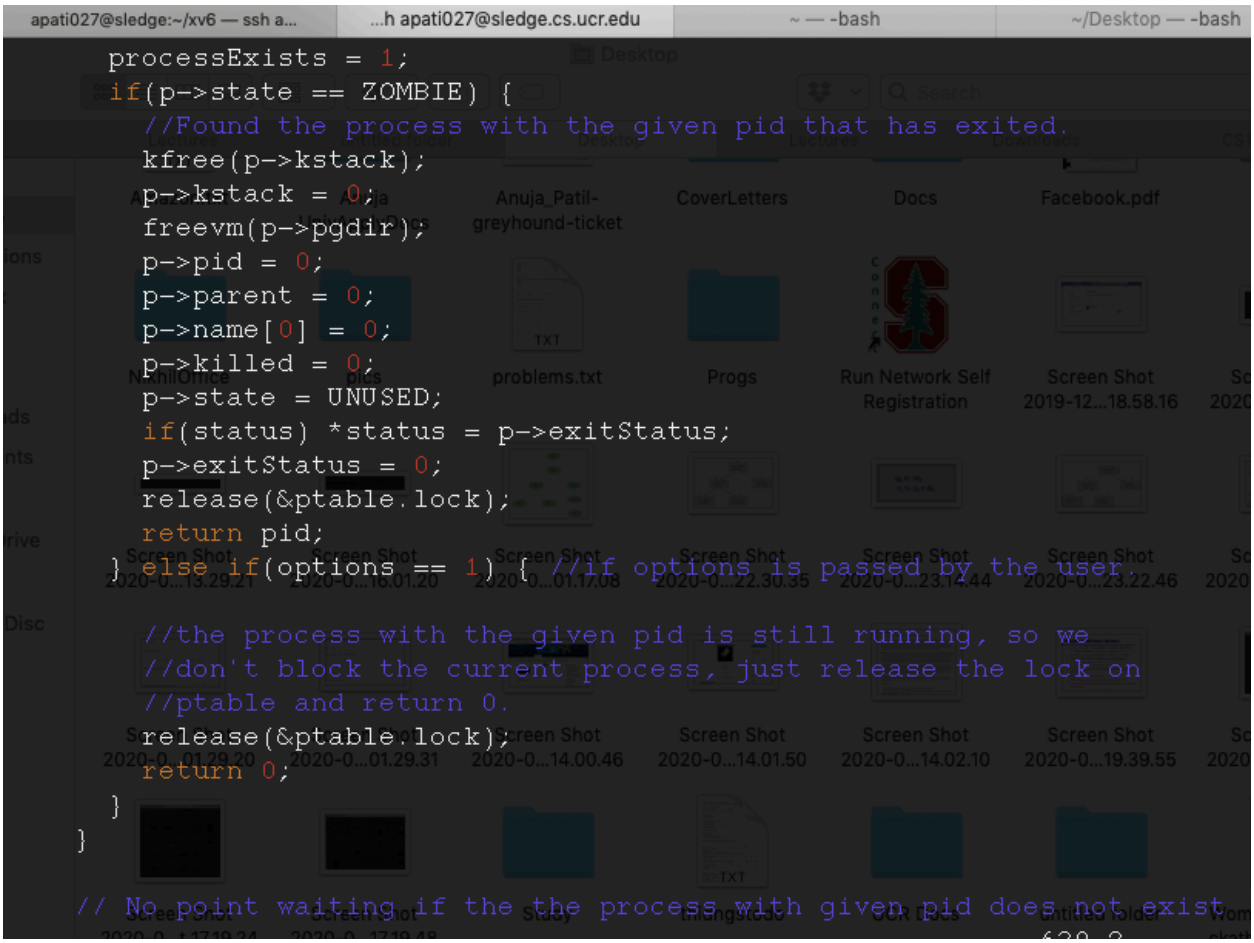
f) proc.c



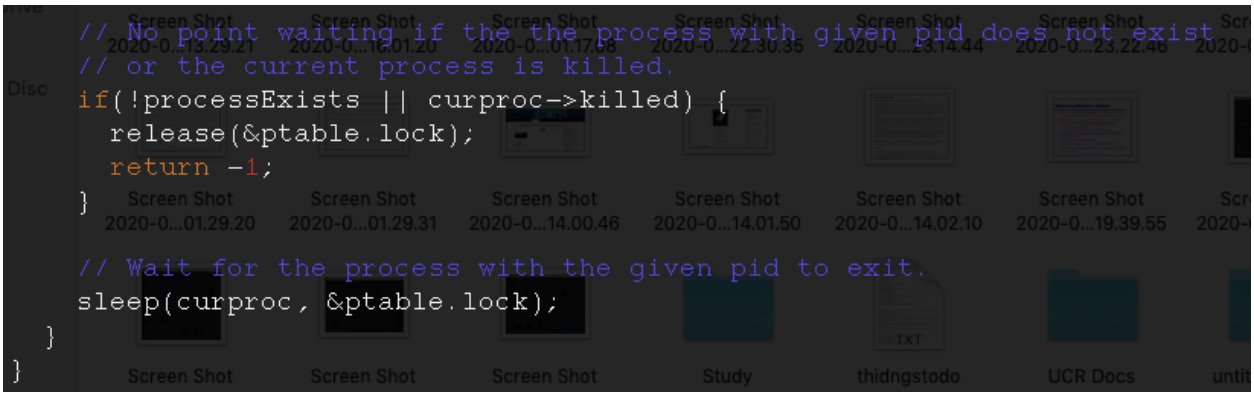
```
//Added for assignment1.
//Defining exitUDef where we store exit status (passed as a parameter)
//into process' state. Rest everything is same as that of exit()
int
exitUDef(int status)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

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

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

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;
```

```
  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);

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

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  curproc->exitStatus = status;
  sched();
  panic("zombie exit");
}
```

```
// Added for assignment1.
// This method waits for a process (not necessary a child process) with a pid
// that equals to the one provided by the pid argument. The return value must
// be the process id of the process that was terminated or -1 if this process
// does not exist or if an unexpected error occurred.
// This method is similar to the wait() defined above.
int
waitpid(int pid, int* status, int options)
{
  struct proc *p, *curproc = myproc();
  int processExists; //similar to havekids in wait()
  acquire(&ptable.lock);

  //Loops continuously till the process with given pid is terminated
  for(;;) {
    processExists = 0;

    //Scan through the process table looking for exited processes. Terminated
    //processes will be in ZOMBIE state.
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      // If the process pid does not match the given pid, no need to continue
      // with this process.
      if(p->pid != pid) continue;

      processExists = 1;
```

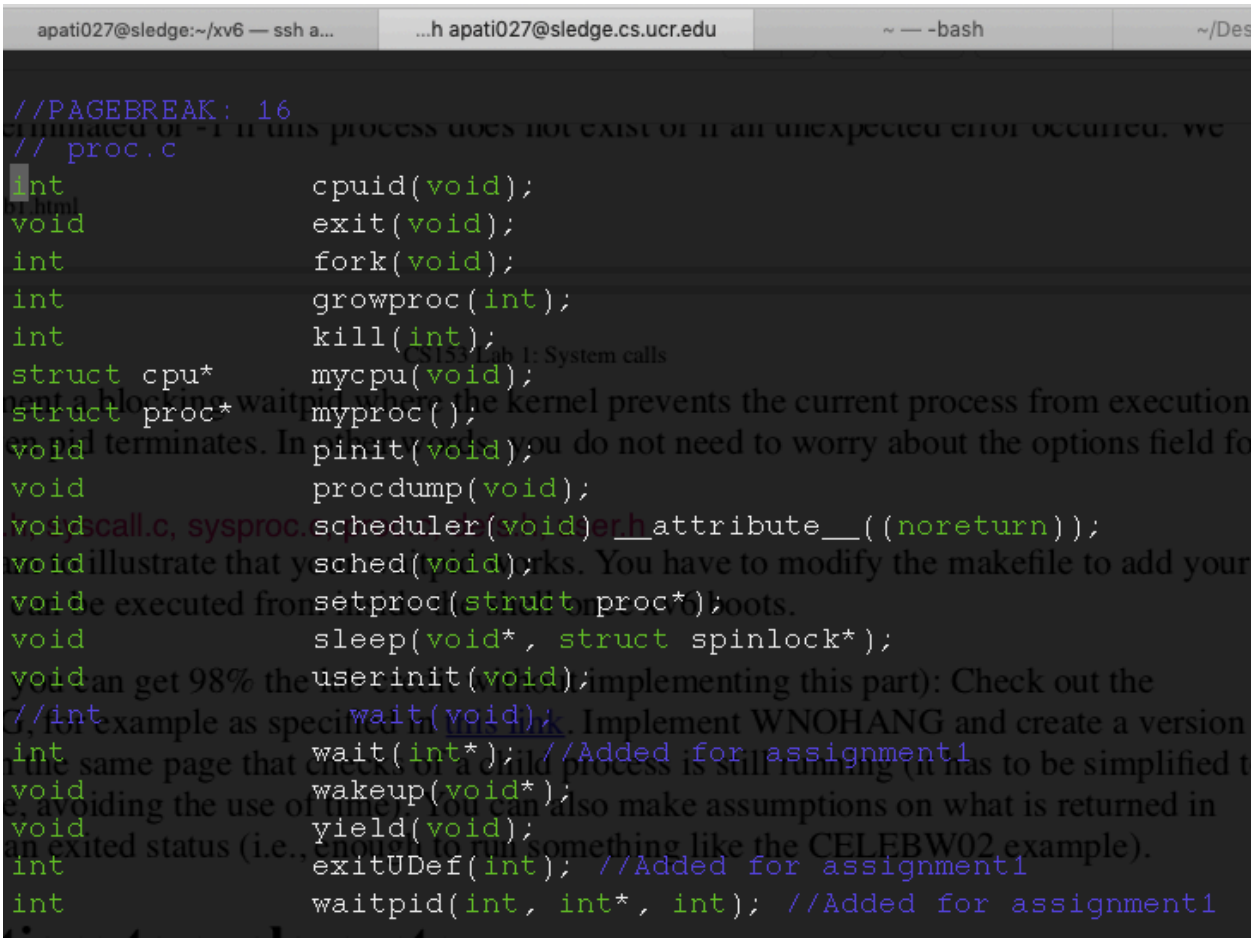


```
      processExists = 1;
      if(p->state == ZOMBIE) {
        //Found the process with the given pid that has exited.
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        if(status) *status = p->exitStatus;
        p->exitStatus = 0;
        release(&ptable.lock);
        return pid;
      } else if(options == 1) { //if options is passed by the user,

        //the process with the given pid is still running, so we
        //don't block the current process, just release the lock on
        //ptable and return 0.
        release(&ptable.lock);
        return 0;
      }
    }

    // No point waiting if the the process with given pid does not exist
```

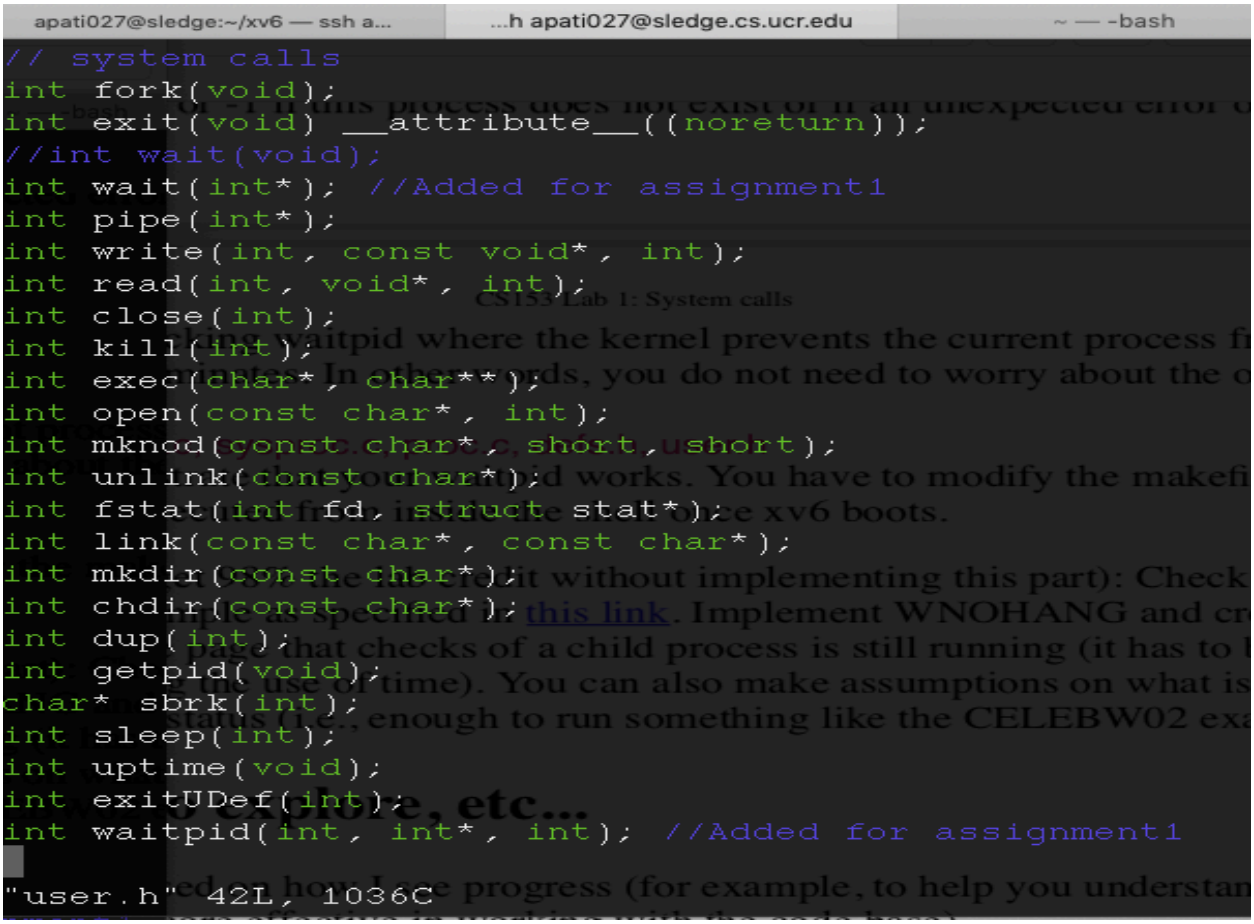


```
    // No point waiting if the the process with given pid does not exist
    // or the current process is killed.
    if(!processExists || curproc->killed) {
      release(&ptable.lock);
      return -1;
    }

    // Wait for the process with the given pid to exit.
    sleep(curproc, &ptable.lock);
  }
}
```

g) defs.h

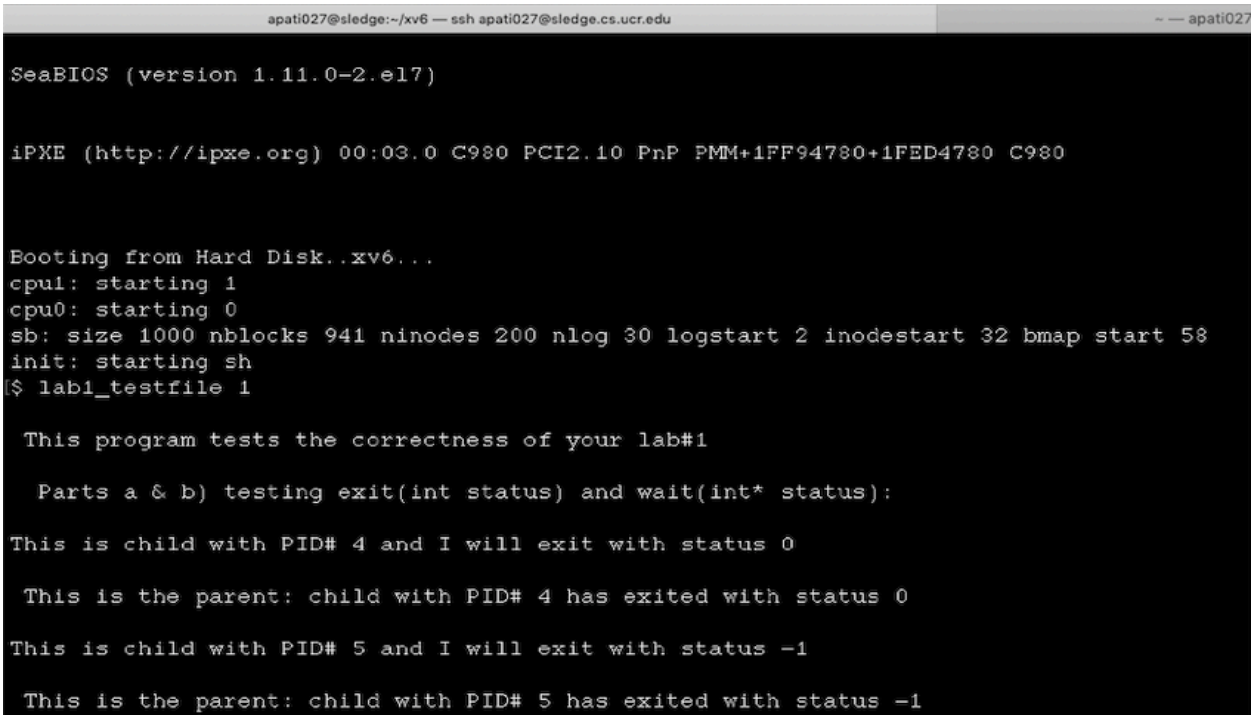

apati027@sledge:~/xv6 — ssh a...
...h apati027@sledge.cs.ucr.edu
~ — -bash
~/Des
//PAGEBREAK: 16
// proc.c
int cpuid(void);
void exit(void);
int fork(void);
int growproc(int);
int kill(int);
struct cpu* mycpu(void);
struct proc* myproc();
void pinit(void);
void procdump(void);
void scheduler(void)__attribute__((noreturn));
void sched(void);
void setproc(struct proc*);
void sleep(void*, struct spinlock*);
void userinit(void);
//int wait(void);
int wait(int*); //Added for assignment1
void wakeup(void*);
void yield(void);
int exitUDef(int); //Added for assignment1
int waitpid(int, int*, int); //Added for assignment1

h) user.h

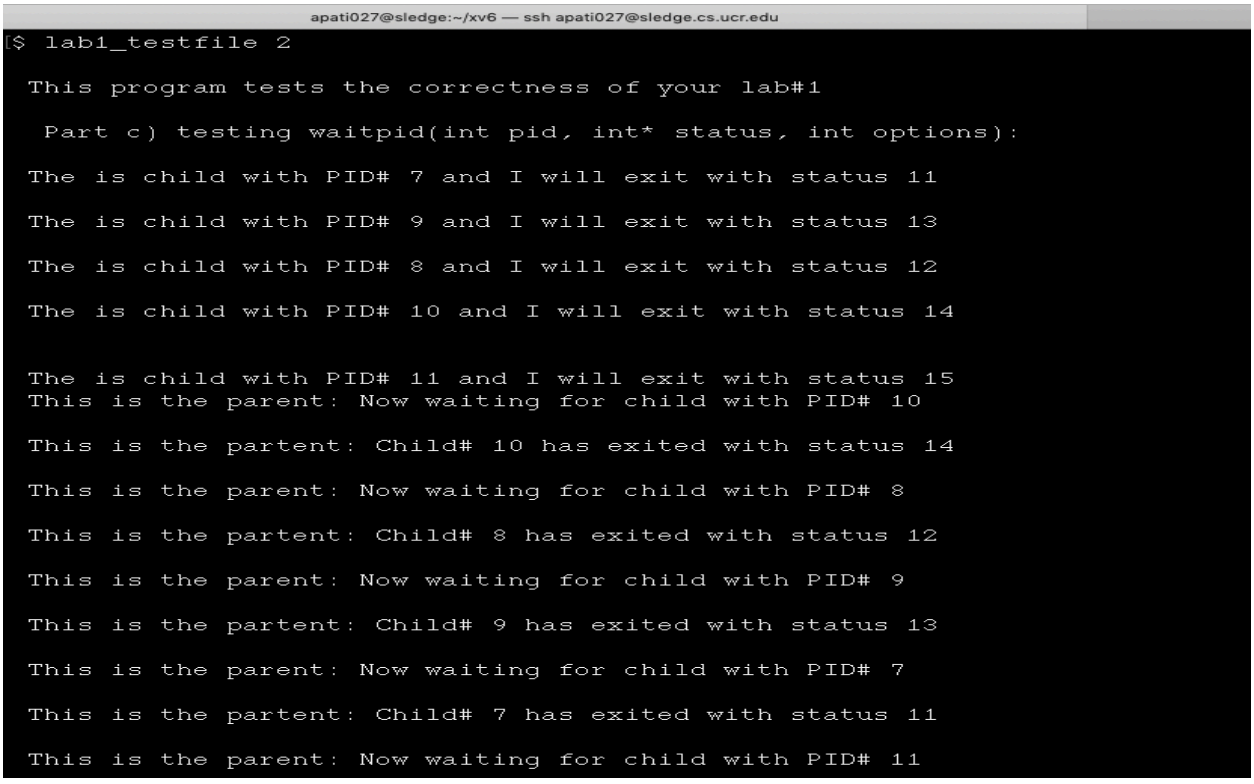


7) To make and run the lab1_testfile.c: (We are inside xv6 folder)
> make clean
> make qemu-nox

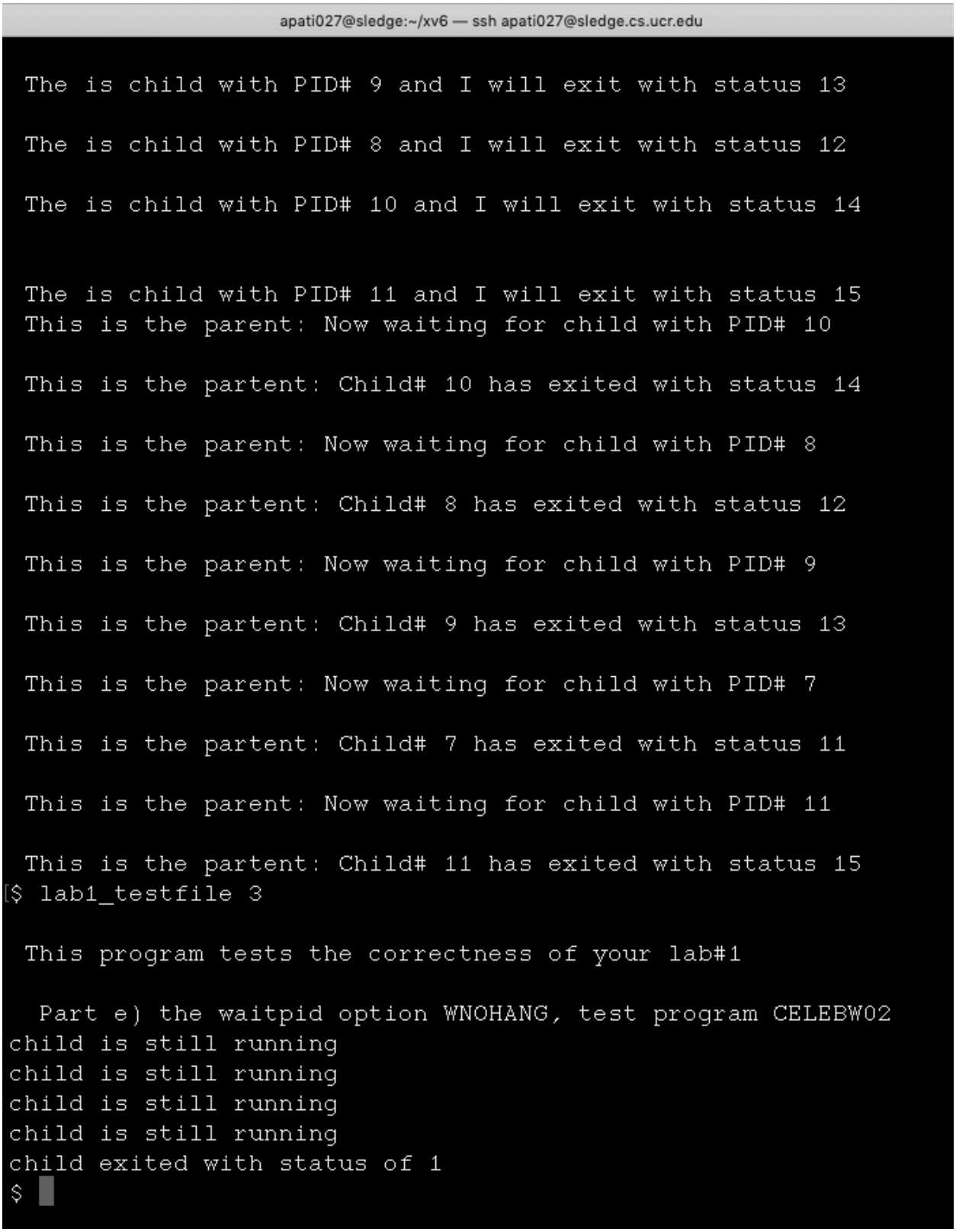
This will open qemu console. Then run:
$ lab1_testfile 1



$ lab1_testfile 2



$ lab1_testfile 3



Summary for lab1:

The lab1 was about learning how to create a system call, how the parameters are passed from user space to kernel space and how are parameters returned from kernel space to user space. The main files to change when creating a system call are – syscall.h, syscall.c, sysproc.c, usys.S, user.h, defs.h and proc.c.

syscall.h: mapping from system call name to system call number.
syscall.c: contains pointers to the system calls. We also add a function prototype.
sysproc.c: contains the implementations of process related system calls.
usys.S: contains a list of system calls exported by the kernel.
user.h: contains the system call definitions in xv6.
defs.h: contains forward declaration of system calls.
proc.c: contains the implementations of the system calls, scheduler, exit, wait etc.