Anthony Mejia - SID: 861152139
Christian Campos - SID: 862080812

Lab 1: Fun with System Calls Report

Part a:

Part a is to implement the existing exit system call to accept and store the exit status of the process that is terminated.
There’s two possible methods to do this:

1) Change the existing system call signature to void exit(int status)
2) Create a new exit system call.

We create a new exit system call exitStat(int) and maintain the exit() function.

Part b:

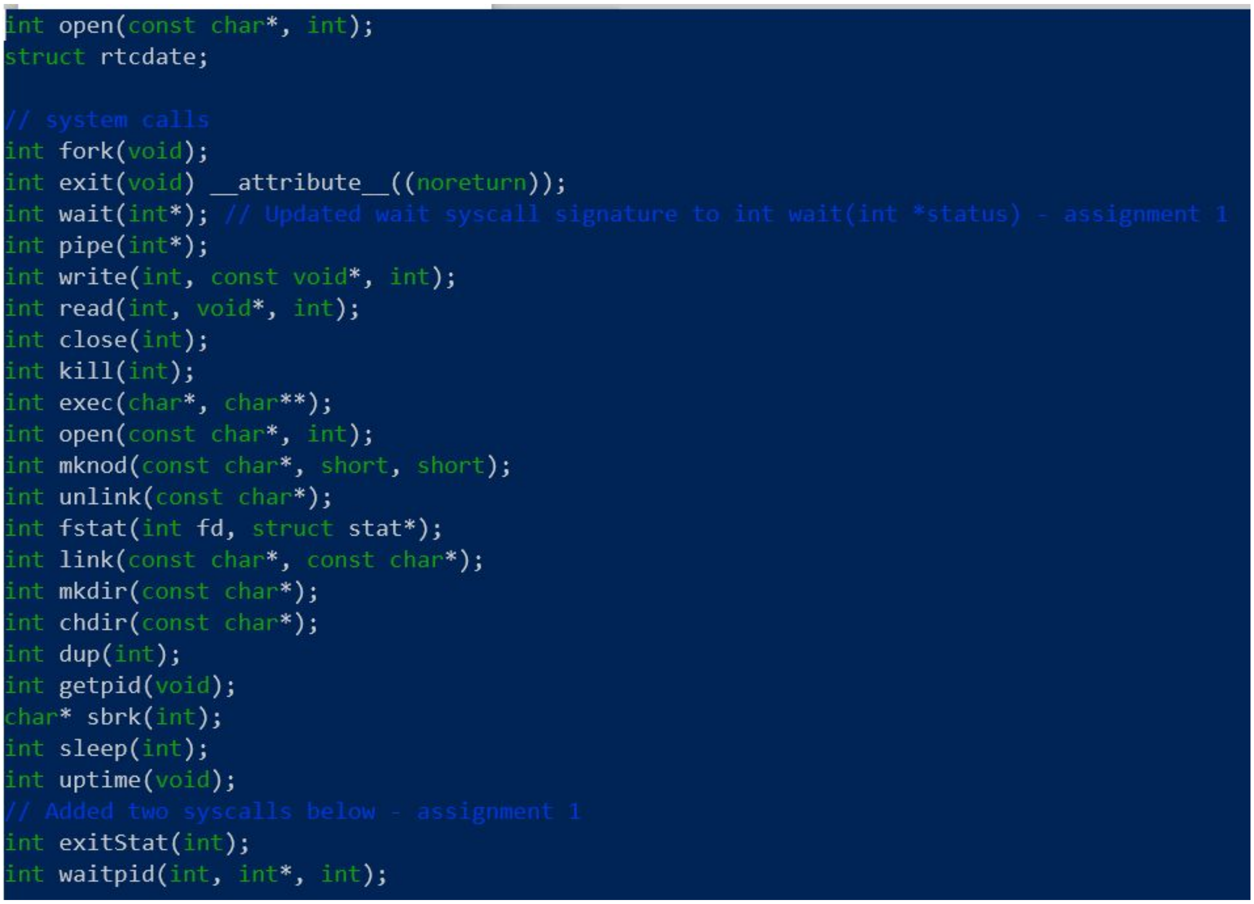
Part b is to update the existing wait system call to int wait(int*status).
To do this , we created a wait system called int wait(*) to prevent the current process until its child processes successfully exit. The int wait(*) returns the terminated child’s exit status and it returns 0 or -1 based on its execution.

Part c:

Part c is to add a waitpid system call int waitpid(int pid, int *status, int options) based on wait system call from part 2. The goal is to implement it so that it waits for a process that has match pid and returns a pid of the terminated process.Waitpid function acts like wait sys call and waits for a process (not necessary a child process) with a pid that equals to the pid provided in the parameter.

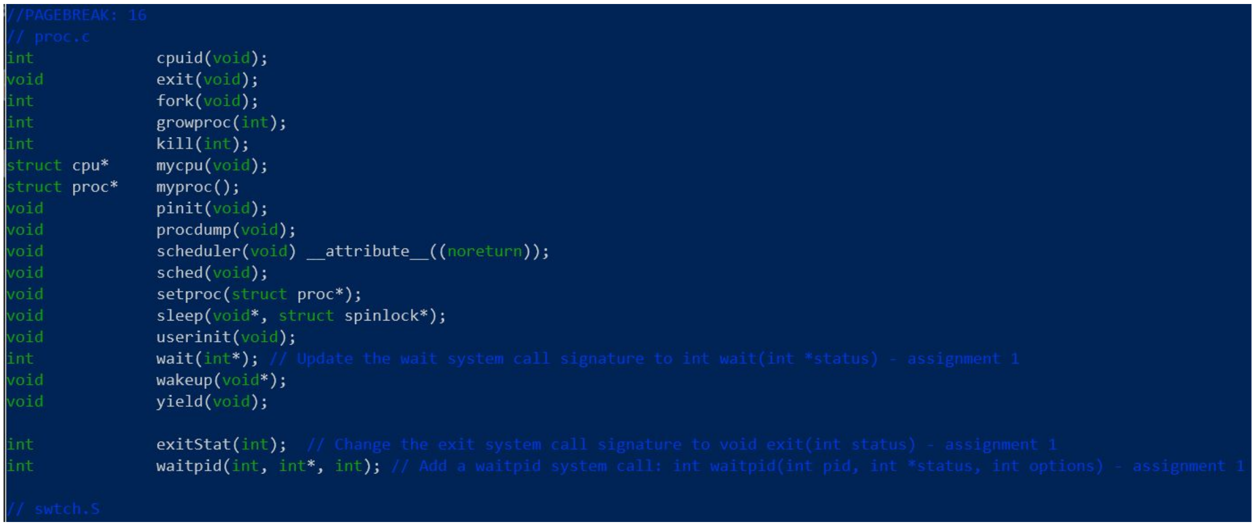
The following screenshots below are our modification of the code we made in lab 1

user.h:



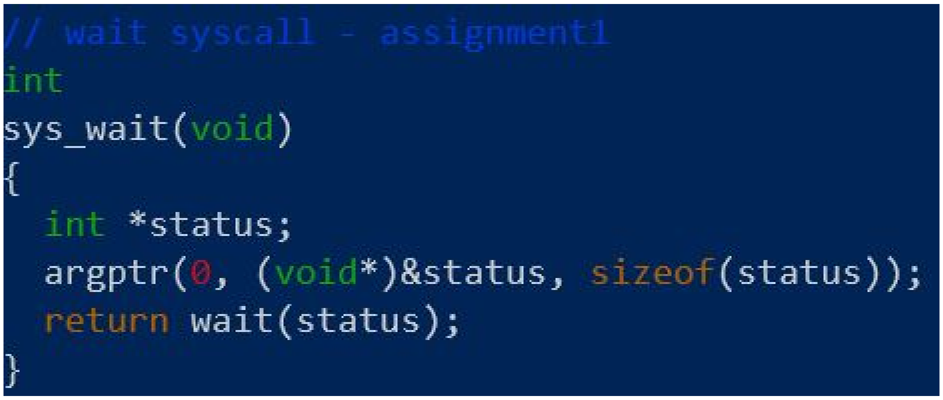
```c
int open(const char*, int);
struct rtcdate;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(int*); // Updated wait syscall signature to int wait(int *status) - assignment 1
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
// Added two syscalls below - assignment 1
int exitStat(int);
int waitpid(int, int*, int);
```

defs.h:

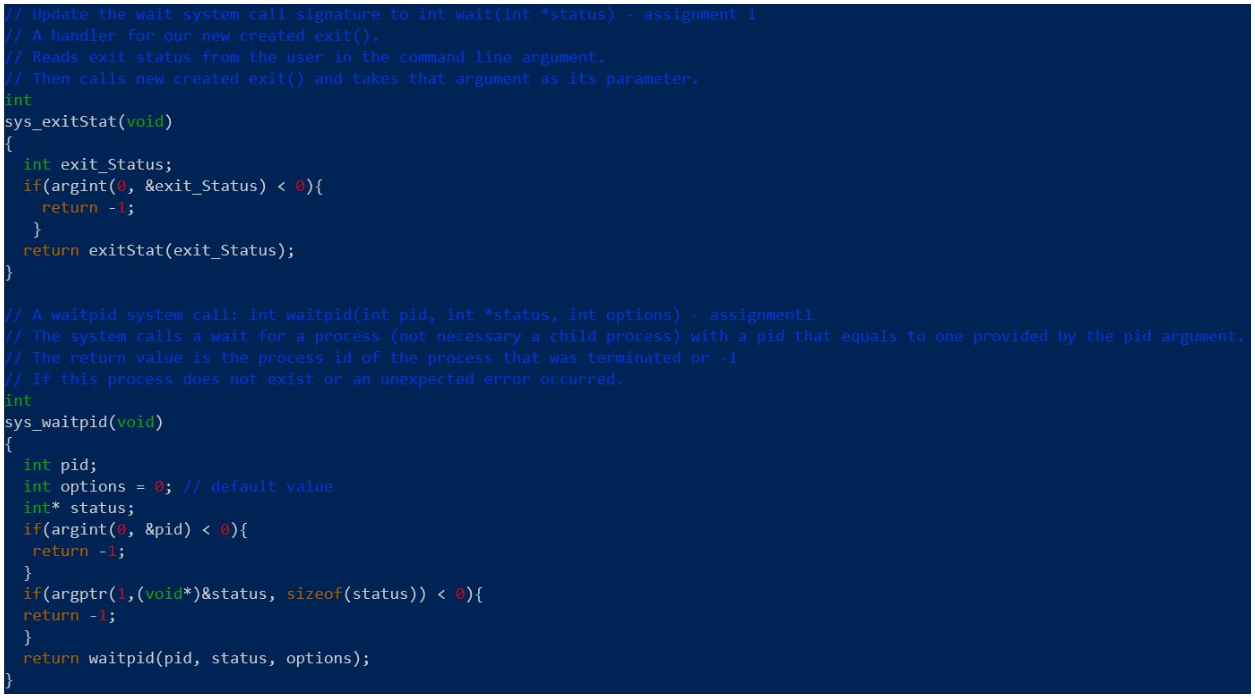


```c
//PAGEBREAK: 16
// proc.c
int             cpuid(void);
void            exit(void);
int             fork(void);
int             growproc(int);
int             kill(int);
struct cpu*     mycpu(void);
struct proc*    myproc();
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            setproc(struct proc*);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(int*); // Update the wait system call signature to int wait(int *status) - assignment 1
void            wakeup(void*);
void            yield(void);

int             exitStat(int);  // Change the exit system call signature to void exit(int status) - assignment 1
int             waitpid(int, int*, int); // Add a waitpid system call: int waitpid(int pid, int *status, int options) - assignment 1

// swtch.S
```

sysproc.c:

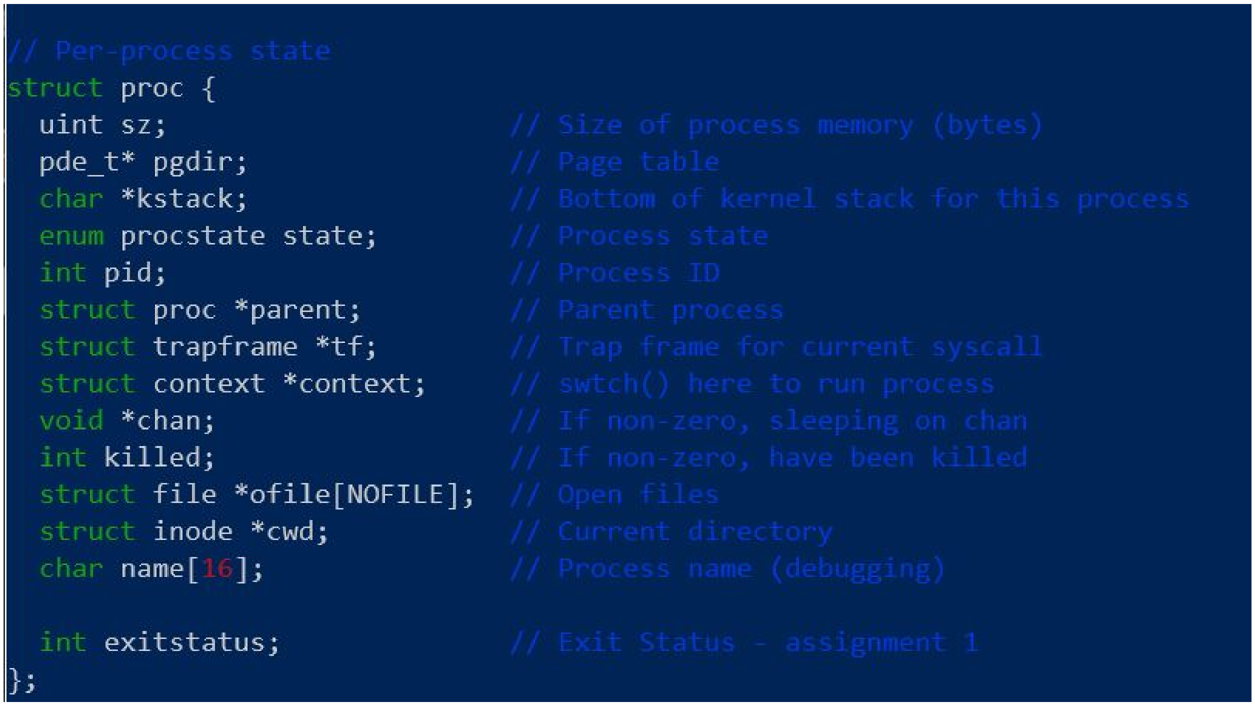


```c
// wait syscall - assignment1
int
sys_wait(void)
{
  int *status;
  argptr(0, (void*)&status, sizeof(status));
  return wait(status);
}
```



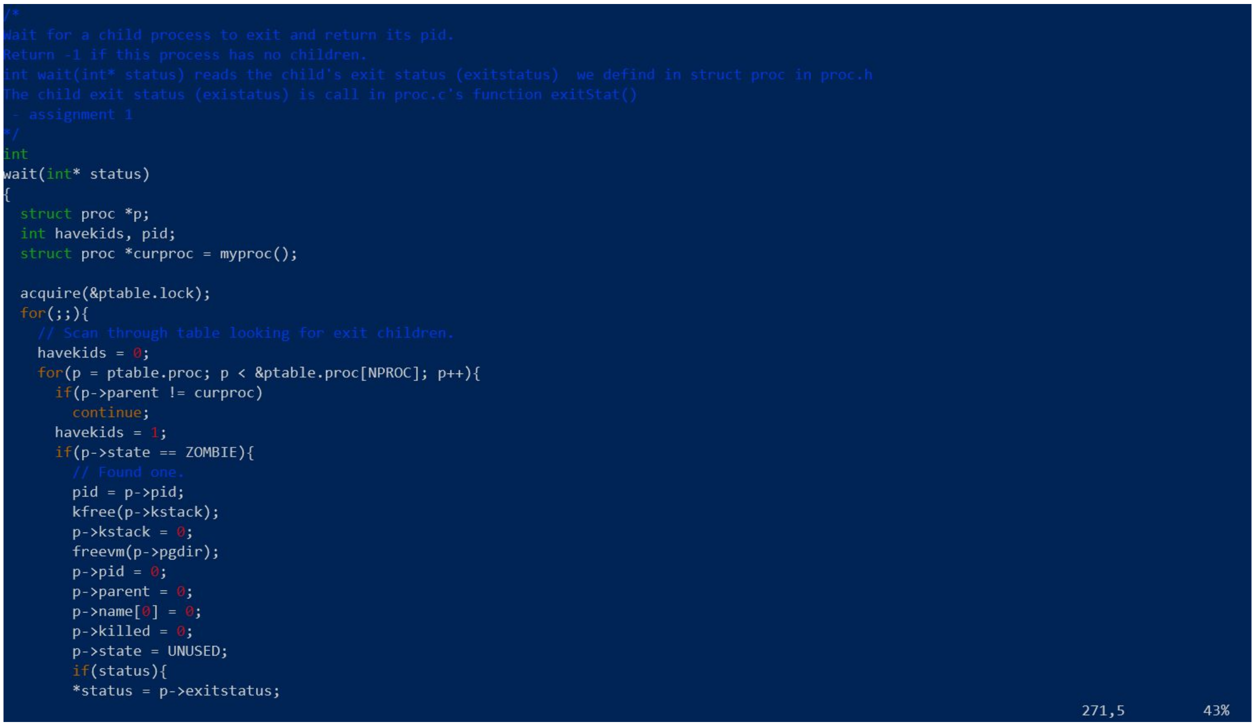
```c
// Update the wait system call signature to int wait(int *status) - assignment 1
// A handler for our new created exit().
// Reads exit status from the user in the command line argument.
// Then calls new created exit() and takes that argument as its parameter.
int
sys_exitStat(void)
{
  int exit_Status;
  if(argint(0, &exit_Status) < 0){
    return -1;
   }
  return exitStat(exit_Status);
}

// A waitpid system call: int waitpid(int pid, int *status, int options) - assignment1
// The system calls a wait for a process (not necessary a child process) with a pid that equals to one provided by the pid argument.
// The return value is the process id of the process that was terminated or -1
// If this process does not exist or an unexpected error occurred.
int
sys_waitpid(void)
{
  int pid;
  int options = 0; // default value
  int* status;
  if(argint(0, &pid) < 0){
   return -1;
  }
  if(argptr(1,(void*)&status, sizeof(status)) < 0){
  return -1;
  }
  return waitpid(pid, status, options);
}
```

proc.h:

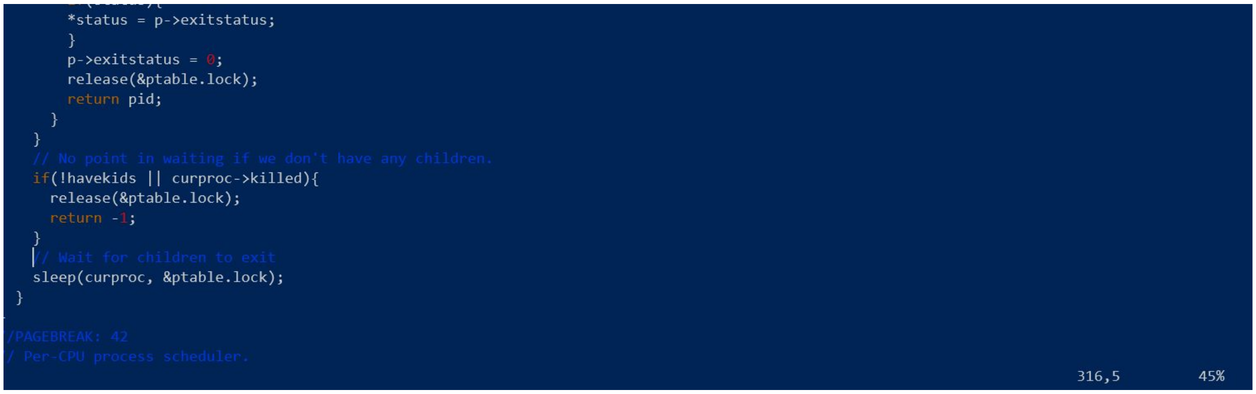


```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // 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 exitstatus;              // Exit Status - assignment 1
};
```

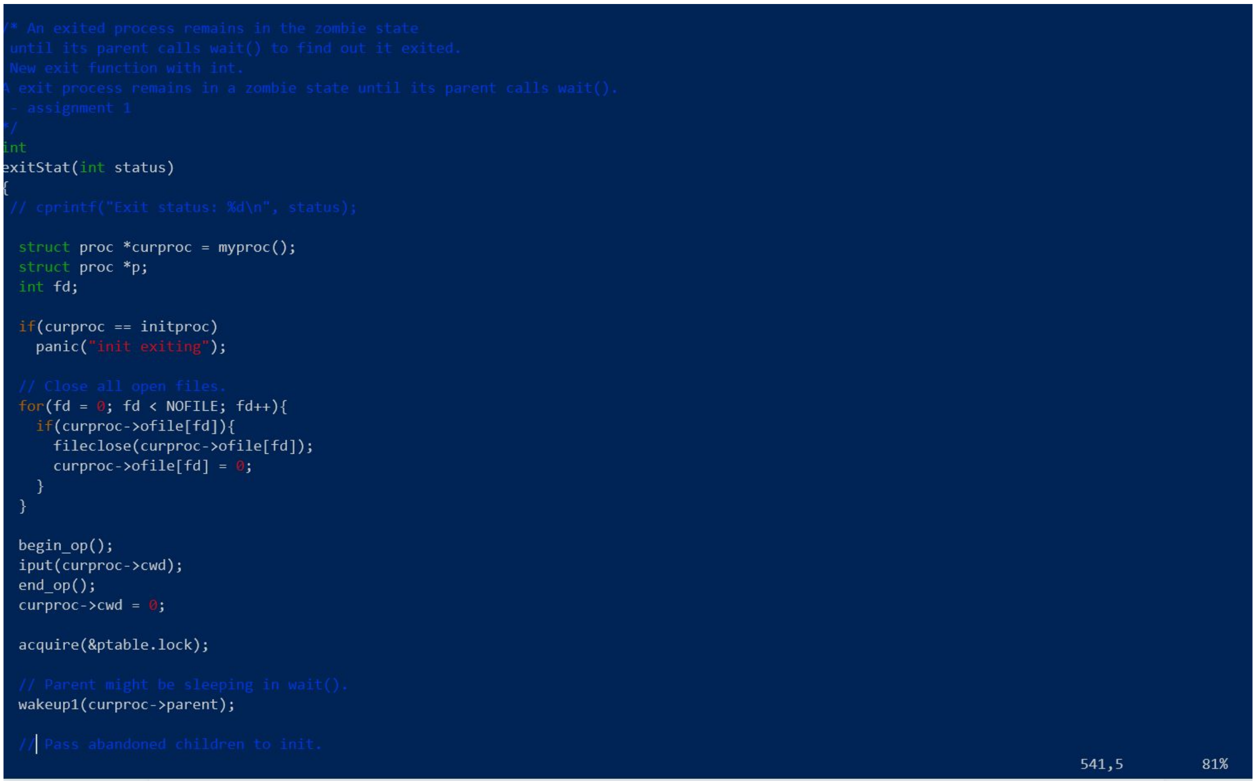
proc.c:



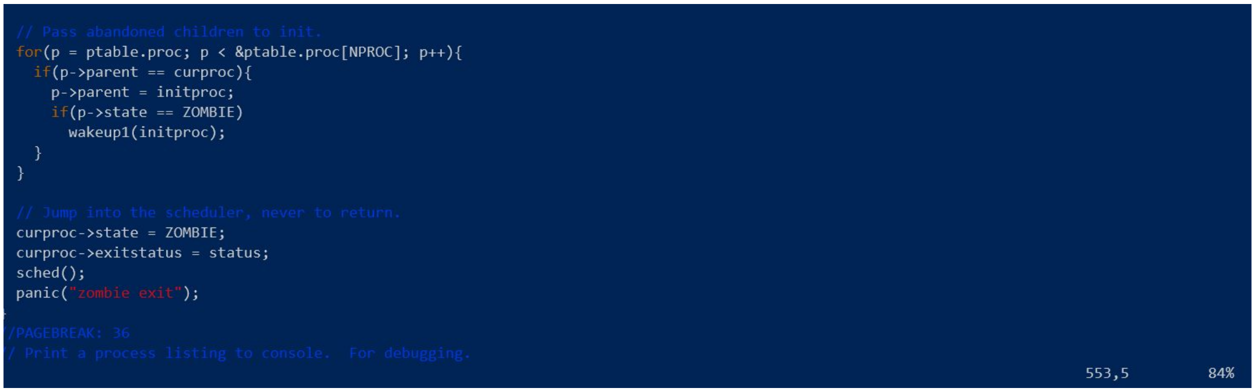
```
/*
Wait for a child process to exit and return its pid.
Return -1 if this process has no children.
int wait(int* status) reads the child's exit status (exitstatus)  we defind in struct proc in proc.h
The child exit status (existatus) is call in proc.c's function exitStat()
 - assignment 1
*/
int
wait(int* status)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exit children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        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){
        *status = p->exitstatus;
                                                                  271,5         43%
```



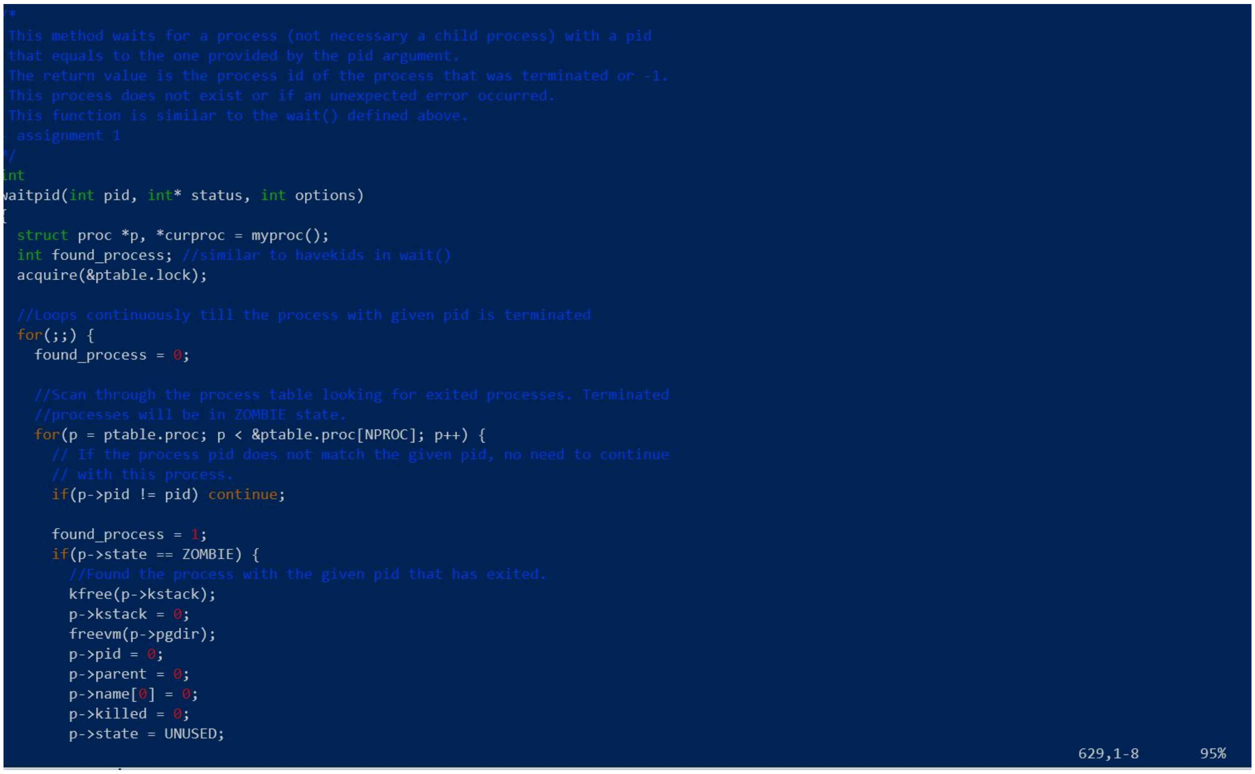
```
      if(status){
        *status = p->exitstatus;
        }
        p->exitstatus = 0;
        release(&ptable.lock);
        return pid;
      }
    }
    // No point in waiting if we don't have any children.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }
    // Wait for children to exit
    sleep(curproc, &ptable.lock);
  }
}
/PAGEBREAK: 42
/ Per-CPU process scheduler.
                                                                                    316,5          45%
```



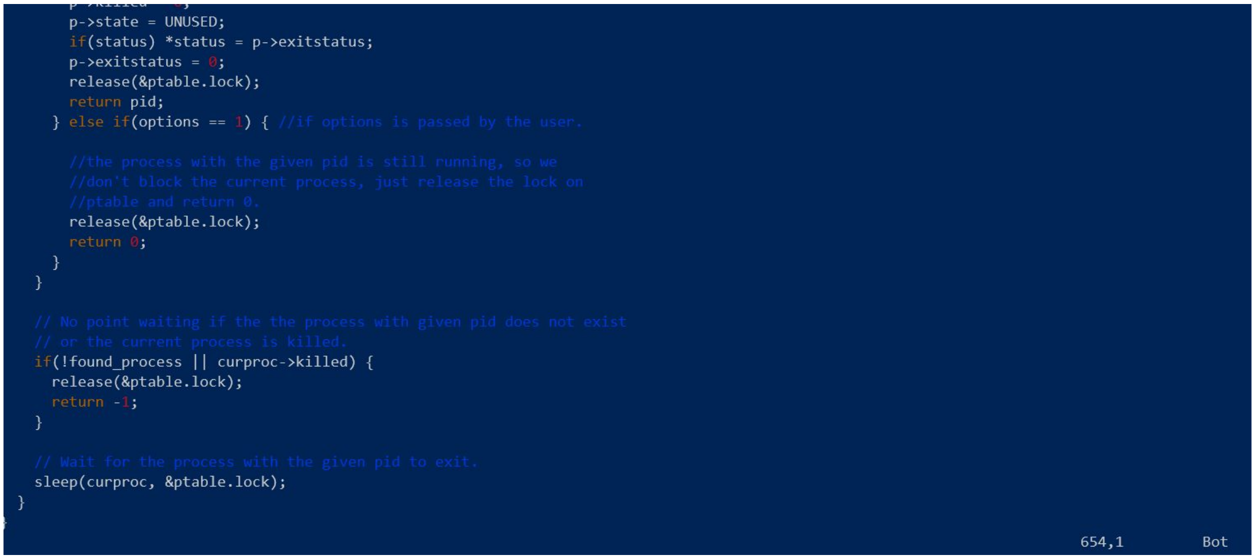
```
/* An exited process remains in the zombie state
 until its parent calls wait() to find out it exited.
 New exit function with int.
A exit process remains in a zombie state until its parent calls wait().
 - assignment 1
*/
int
exitStat(int status)
{
 // cprintf("Exit status: %d\n", status);

  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;

  acquire(&ptable.lock);

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

  // Pass abandoned children to init.
                                                                                    541,5          81%
```



```
  // 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");
}
/PAGEBREAK: 36
/ Print a process listing to console.  For debugging.
                                                                                    553,5          84%
```



```
/*
 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 is the process id of the process that was terminated or -1.
 This process does not exist or if an unexpected error occurred.
 This function is similar to the wait() defined above.
- assignment 1
*/
int
waitpid(int pid, int* status, int options)
{
  struct proc *p, *curproc = myproc();
  int found_process; //similar to havekids in wait()
  acquire(&ptable.lock);

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

      found_process = 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;
                                                                629,1-8       95%
```



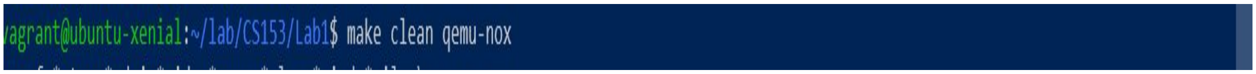
```
        p->state = UNUSED;
        if(status) *status = p->exitstatus;
        p->exitstatus = 0;
        release(&ptable.lock);
        return pid;
      } else if(options == 1) { //if options is passed by the user.

        //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
    // or the current process is killed.
    if(!found_process || curproc->killed) {
      release(&ptable.lock);
      return -1;
    }

    // Wait for the process with the given pid to exit.
    sleep(curproc, &ptable.lock);
  }
}
                                                                654,1          Bot
```

.c fIles needed to modified:

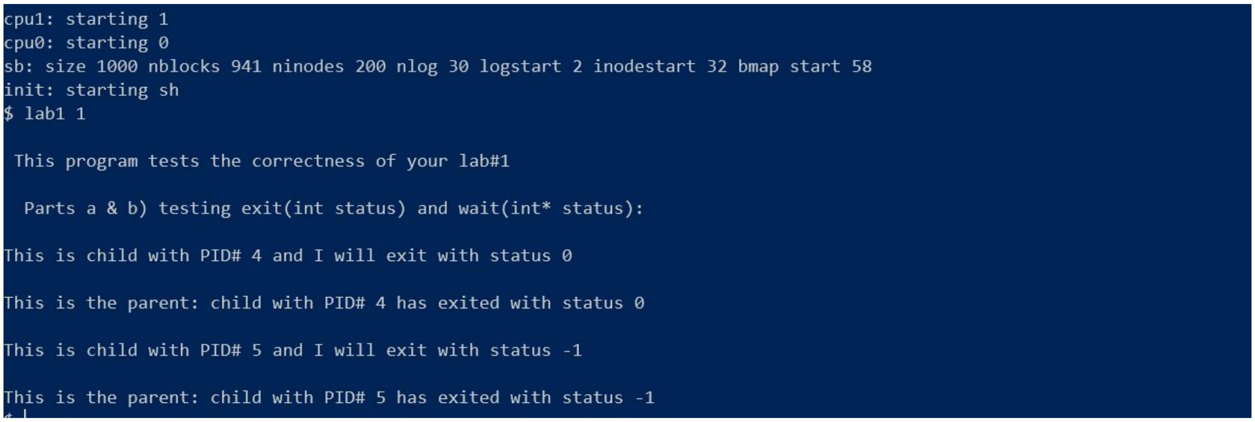
Many .c files required a change from exit() to exit(0) and wait() to wait(0).
The following files were changes that we made:
cat.c echo.c forktest.c grep.c init.c kill.c ln.c ls.c mkdir.c rm.c
sh.c stressfs.c trap.c ulib.c wc.c zombie.c usertest.c

<u>Test:</u>

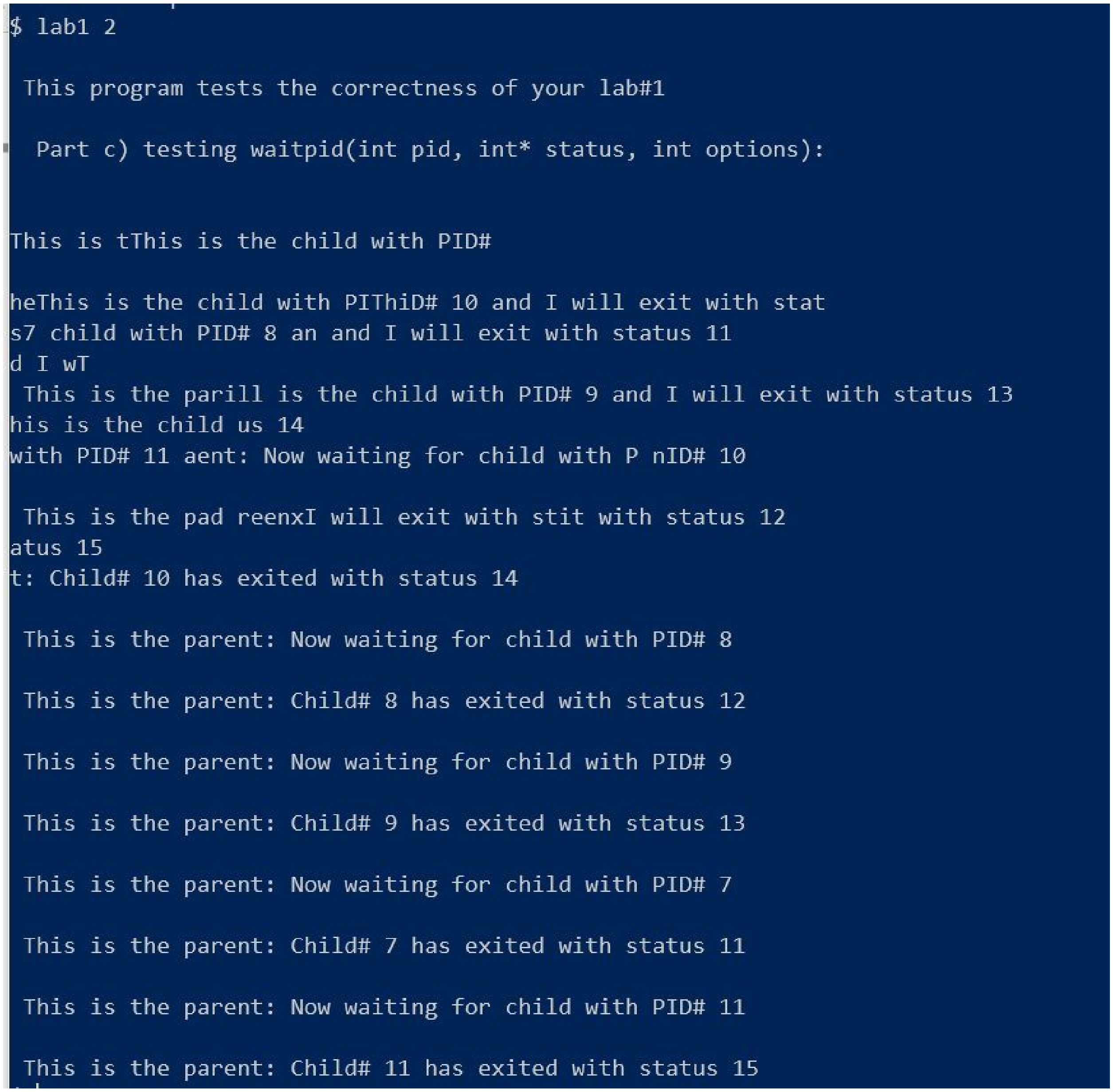
To run our test file; lab1.c
Use make clean qemu-nox inside the xv6 directory.
Then run: Lab1 1 to test part a and b.
Finally, Lab1 2 to test part c.
However, a weird format issue occurred for the part c output as shown below.



```
vagrant@ubuntu-xenial:~/lab/CS153/Lab1$ make clean qemu-nox
```



```
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ lab1 1

 This program tests the correctness of your lab#1

  Parts a & b) testing exit(int status) and wait(int* status):

This is child with PID# 4 and I will exit with status 0

This is the parent: child with PID# 4 has exited with status 0

This is child with PID# 5 and I will exit with status -1

This is the parent: child with PID# 5 has exited with status -1
```



```
$ lab1 2

 This program tests the correctness of your lab#1

  Part c) testing waitpid(int pid, int* status, int options):


This is tThis is the child with PID#

heThis is the child with PIThiD# 10 and I will exit with stat
s7 child with PID# 8 an and I will exit with status 11
d I wT
 This is the parill is the child with PID# 9 and I will exit with status 13
his is the child us 14
with PID# 11 aent: Now waiting for child with P nID# 10

 This is the pad reenxI will exit with stit with status 12
atus 15
t: Child# 10 has exited with status 14

 This is the parent: Now waiting for child with PID# 8

 This is the parent: Child# 8 has exited with status 12

 This is the parent: Now waiting for child with PID# 9

 This is the parent: Child# 9 has exited with status 13

 This is the parent: Now waiting for child with PID# 7

 This is the parent: Child# 7 has exited with status 11

 This is the parent: Now waiting for child with PID# 11

 This is the parent: Child# 11 has exited with status 15
```

Below, we have the diff output txt file.

diff output txt file: