Ford St. John

CS153 – Lab 1/Assignment 1

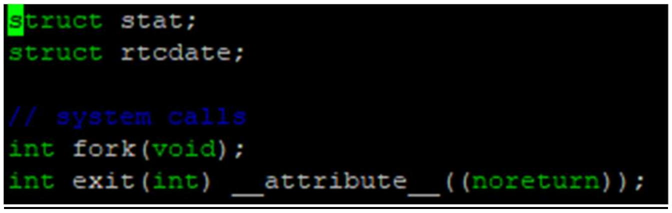
Lab Section 021

862125078

(a) To change the exit system call signature so that the status of the exited program is stored in the calling program's data structure, I amended the following files: user.h, defs.h, proc.h, proc.c, sysproc.c. Amending these files enabled xv6 to handle the exit() system call with an integer exit status passed in as an argument. Below are screenshots of the amendments to the selected files:

user.h

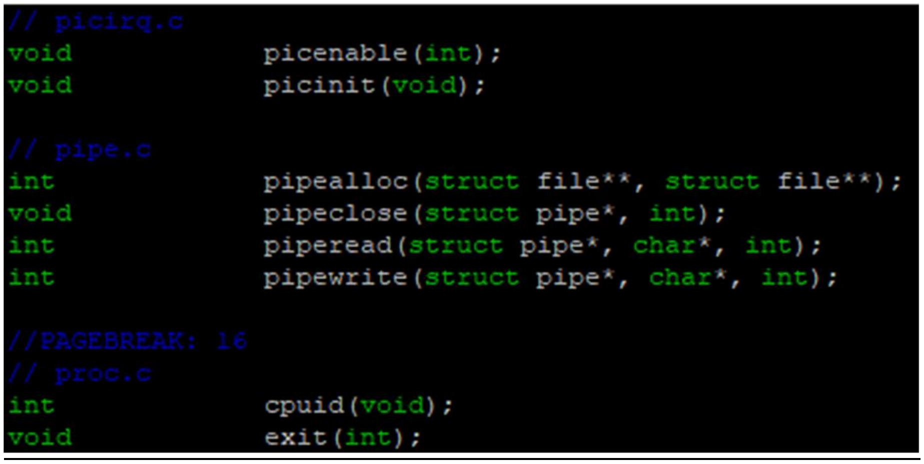
Updated the signature of exit from `int exit(void)` to `int exit(int)`



```
struct stat;
struct rtcdate;

// system calls
int fork(void);
int exit(int) __attribute__((noreturn));
```

defs.h

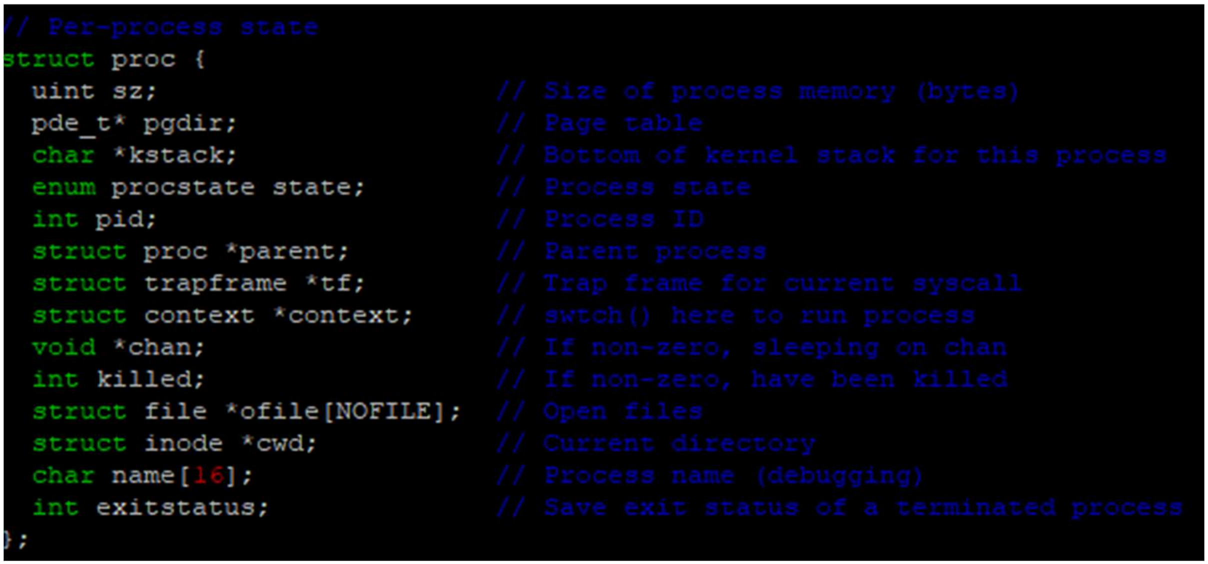
Updated the signature of exit from `void exit(void)` to `void exit(int)`



```
// picirq.c
void            picenable(int);
void            picinit(void);

// pipe.c
int             pipealloc(struct file**, struct file**);
void            pipeclose(struct pipe*, int);
int             piperead(struct pipe*, char*, int);
int             pipewrite(struct pipe*, char*, int);

//PAGEBREAK: 16
// proc.c
int             cpuid(void);
void            exit(int);
```

proc.h

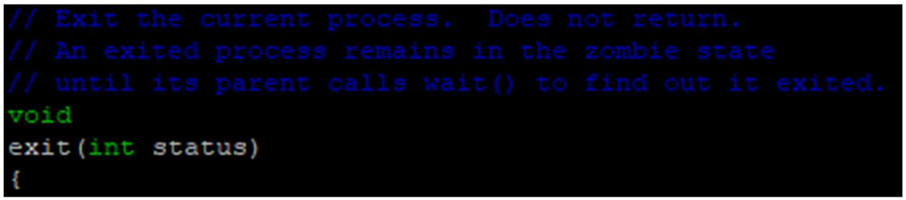
Added an integer member to `struct proc` so that the exit status of the calling process can be stored in that process' data structure.



```
// 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;              // Save exit status of a terminated process
};
```

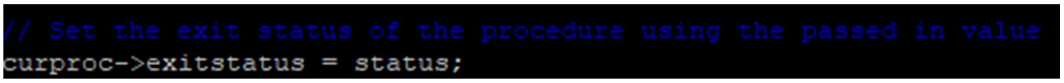
proc.c

Matched the function signature to the function signature defined in defs.h



```
// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
void
exit(int status)
{
```

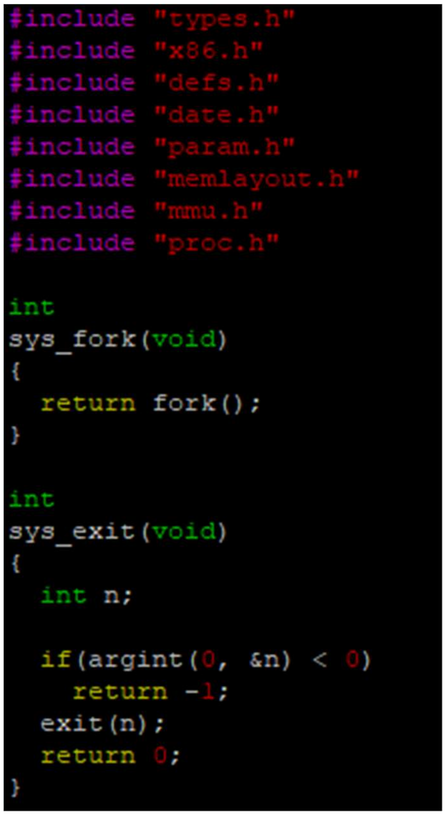
Added the code below before the process jumps to the scheduler so that the exit status of the process is saved before quitting the process



```
// Set the exit status of the procedure using the passed in value
curproc->exitstatus = status;
```

sysproc.c

Added the code below to the `int sys_exit(void)` function so that when exit() is called as a system call, the integer value n is initialized (the system call will exit with error status -1 if the integer variable n fails to initialize) and passed to the function so that the exit status is properly stored



```
#include "types.h"
#include "x86.h"
#include "defs.h"
#include "date.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"

int
sys_fork(void)
{
  return fork();
}

int
sys_exit(void)
{
  int n;

  if(argint(0, &n) < 0)
    return -1;
  exit(n);
  return 0;
}
```

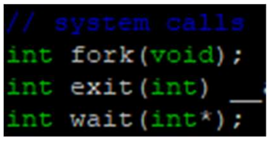
The following files needed to be modified for backwards compatibility, as all utilized the exit() system call, and therefore needed their function signature updated to match the newly amended exit() signature:

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, usertests.c, wc.c, zombie.c

It should be noted that most of these files utilized exit() to return from processing when an error was encountered, or to return from processing upon successful completion of the implemented task. There were no explicit instructions on what exit status should be stored upon successful completion of a task or error in task processing, so I used the following convention: `exit(0)` where status = 0 indicates successful processing, and `exit(-1)` where status = -1 to indicate a failure of some sort

(b) To update the wait() call signature to `int wait(int* status)` I amended the following main xv6 files: user.h, defs.h, proc.c, sysproc.c user.h

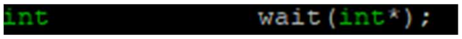
Updated the wait() signature from `int wait(void)` to `int wait(int* status)`



```
// system calls
int fork(void);
int exit(int) __
int wait(int*);
```

defs.h

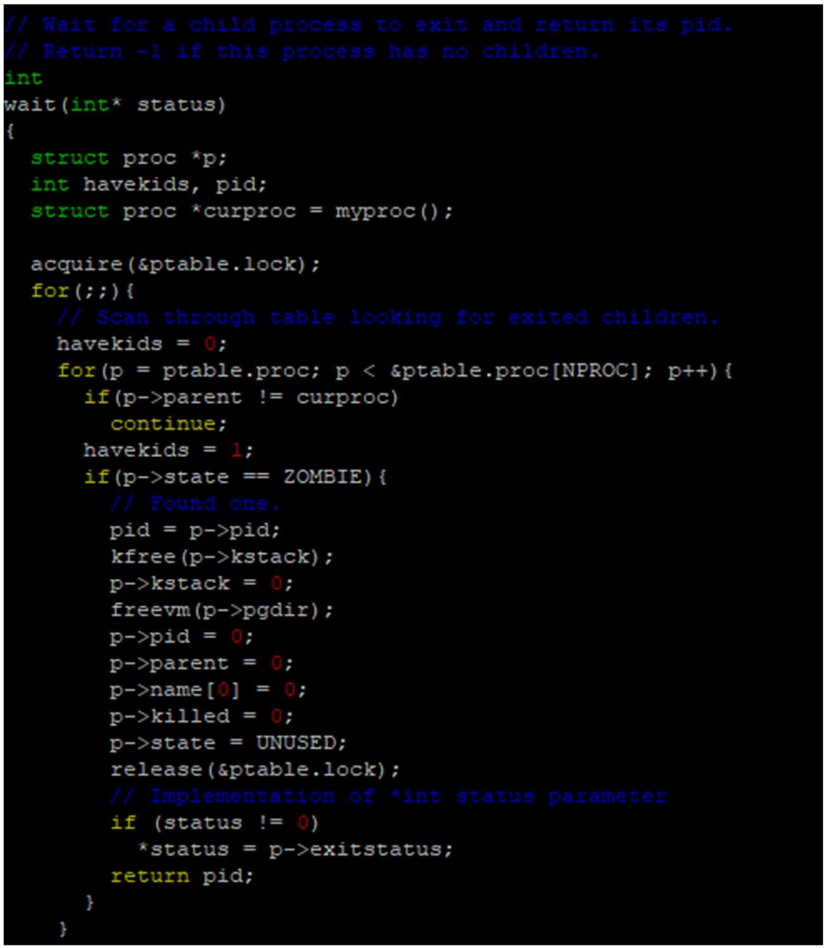
Updated the wait() signature from `int wait(void)` to `int wait(int* status)`



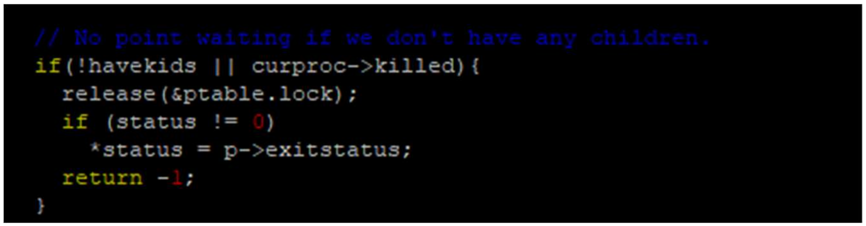
```
int                    wait(int*);
```

proc.c

Updated the wait() function signature to match defs.h. Per the assignment specifications, the wait() function must wait until a child process terminates, then return the exit status of the terminated child (through the int* status argument). The user is allowed to pass a NULL argument to wait(), meaning that the exit status of the terminating child process should be "discarded" (e.g. ignored). The code in the screenshot below implements this logic in two places; first in the for(;;) loop that loops through child processes, and then again in the if statement that checks if the calling process has any children upon which it should wait. Note that status = 0 indicates the status pointer argument of wait is NULL.



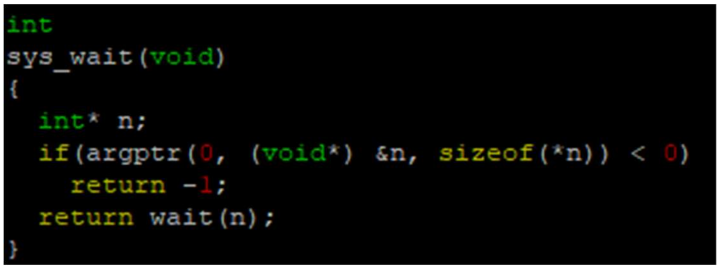
```
// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
wait(int* status)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  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 != 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;
        release(&ptable.lock);
        // Implementation of *int status parameter
        if (status != 0)
          *status = p->exitstatus;
        return pid;
      }
    }
```



```
  // No point waiting if we don't have any children.
  if(!havekids || curproc->killed){
    release(&ptable.lock);
    if (status != 0)
      *status = p->exitstatus;
    return -1;
  }
```

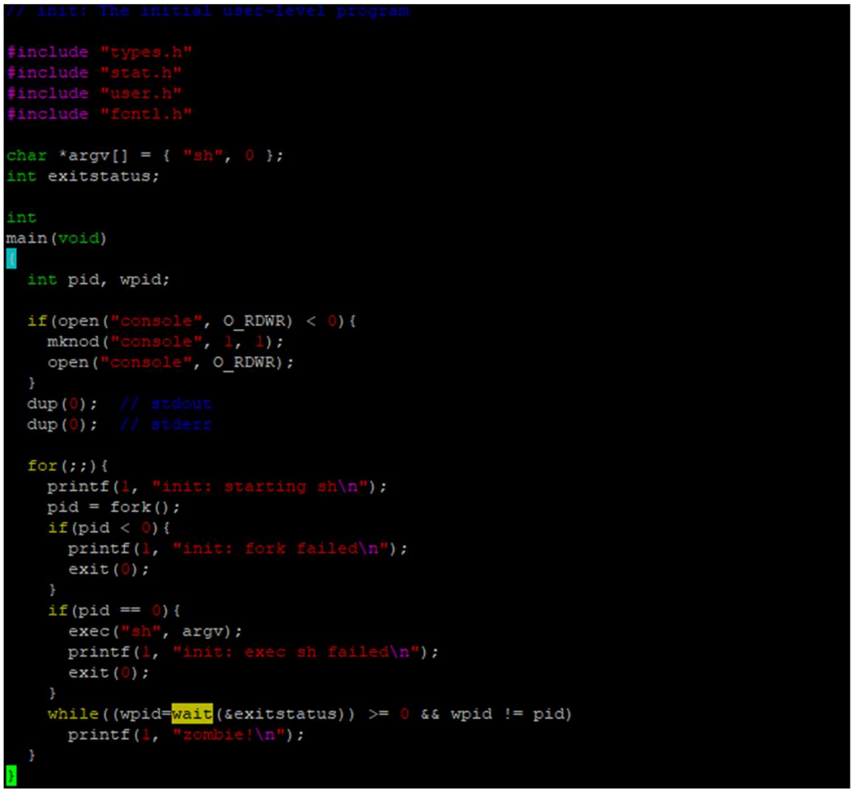
sysproc.c

Similar to the amendment of the exit() function, I amended the `int sys_wait(void)` system call so that an integer pointer n is initialized prior to calling wait(). If the integer pointer fails to initialize, the system call will exit with error status -1



```
int
sys_wait(void)
{
  int* n;
  if(argptr(0, (void*) &n, sizeof(*n)) < 0)
    return -1;
  return wait(n);
}
```

The following files needed to be amended for backwards compatibility with the updated wait() function signature: forktest.c, stressfs.c, sh.c, init.c, usertests.c. To update these files, I introduced an `int exitstatus;` global variable to the file, and passed the memory address of that variable to the wait() function call. Below is an example of the implementation in init.c

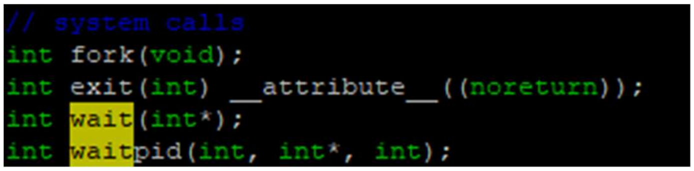


```
// init: The initial user-level program

#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

char *argv[] = { "sh", 0 };
int exitstatus;

int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit(0);
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit(0);
    }
    while((wpid=wait(&exitstatus)) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

(c) To add a brand new system call `int waitpid(int pid, int* status, int options)` to the xv6 implementation, I had to modify the following files: user.h, defs.h, proc.c, sysproc.c, syscall.h, syscall.c, usys.S in order to introduce the system call

user.h

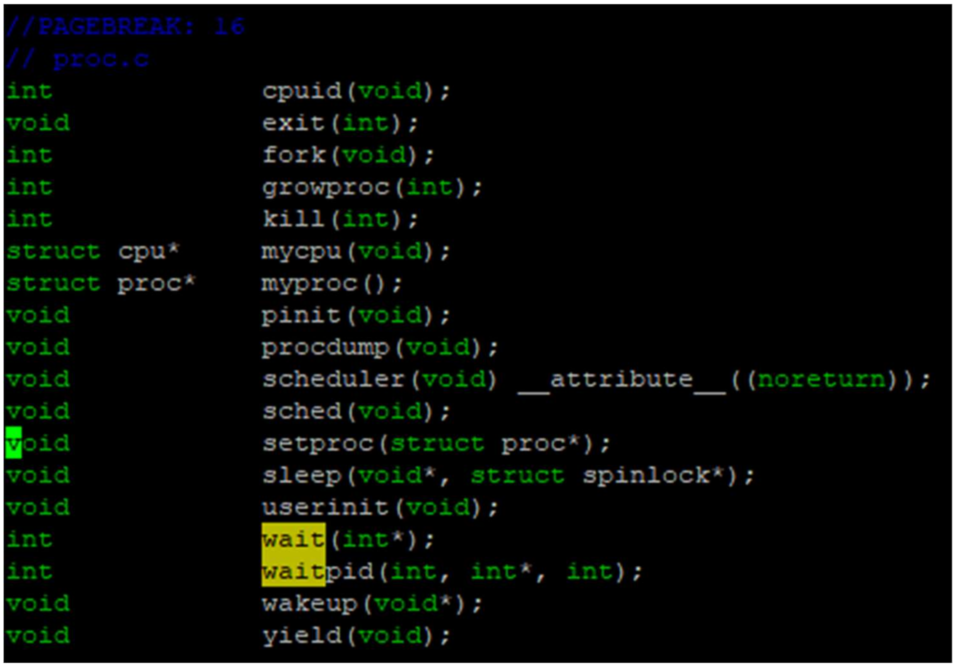
Note the new function signature added to the system call signatures, `int waitpid(int, int*, int)`



```
// system calls
int fork(void);
int exit(int) __attribute__((noreturn));
int wait(int*);
int waitpid(int, int*, int);
```

defs.h

The same function signature was added to this file as well



```
//PAGEBREAK: 16
// proc.c
int             cpuid(void);
void            exit(int);
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*);
int             waitpid(int, int*, int);
void            wakeup(void*);
void            yield(void);
```

proc.c

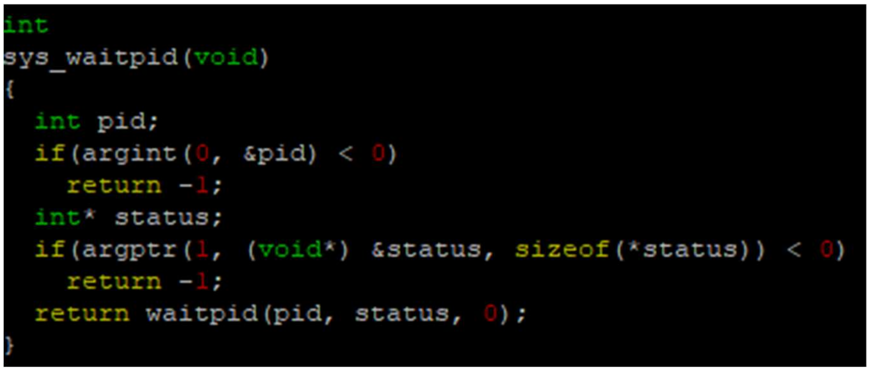
Below is the code (not a screenshot) for the new function implementation. I should note that I've also implemented the functionality for the WNOHANG option. My assumption is that waitpid() should return as normal when the options argument = WNOHANG (implemented as any integer option entered not equal to 0), but should simply not wait for the child process to exit and should exit immediately

```
int
waitpid(int pid, int* status, int options)
{
	struct proc *p;
	int havekids, npid;
	struct proc *curproc = myproc();
```

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

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

        if(options == 0){
            // Only wait on children if options = 0.  Otherwise assume
WNOHANG entered as an option
            sleep(curproc, &ptable.lock);
        }
    }
    }
```

sysproc.c

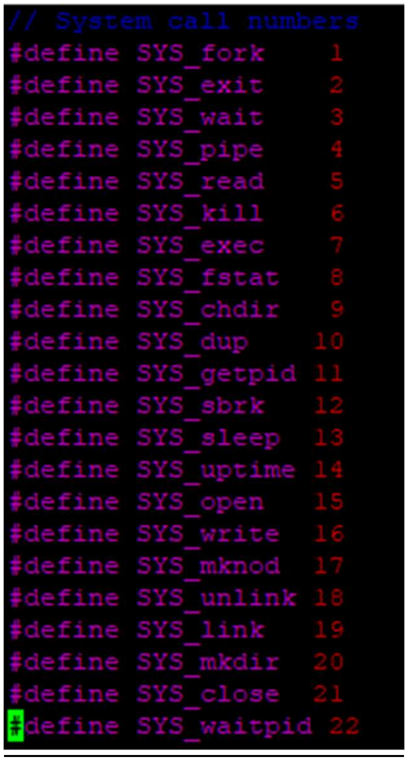
To implement the waitpid() function as a system call, I initialized an integer pid and integer pointer status to be passed to waitpid() when the system call is made



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

syscall.h

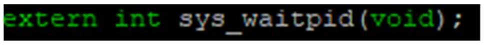
Define waitpid() as a new system call number



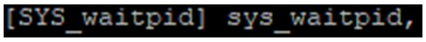
```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  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_waitpid 22
```

syscall.c

Added the following lines of code to implement the new system call number within syscall.c



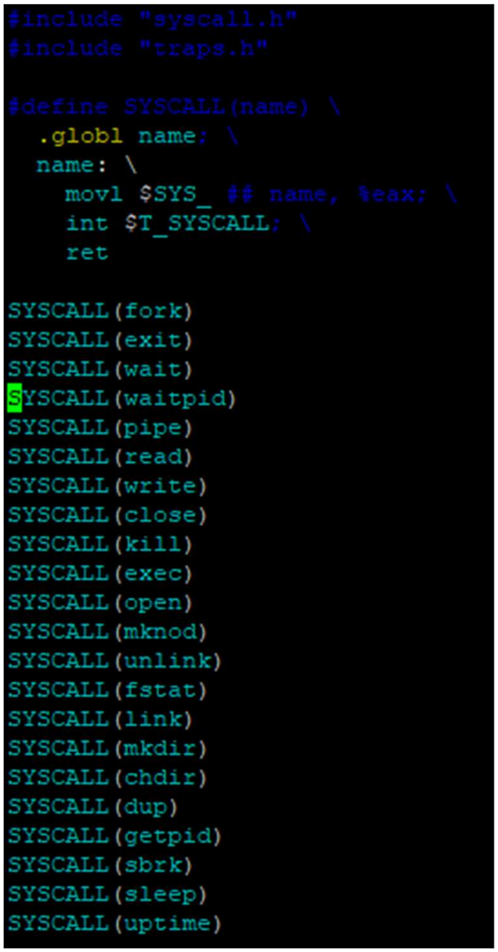
```
extern int sys_waitpid(void);
```



```
[SYS_waitpid] sys_waitpid,
```

usys.S

Added SYSCALL(waitpid) to this file which defines a waitpid as a system call



```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(waitpid)
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)
```

(d) Below is the testing code we were given to ensure our waitpid() implementation works:

```
int waitPid(void){

  int ret_pid, exit_status;
  int i;
  int pid_a[5]={0, 0, 0, 0, 0};
 // use this part to test wait(int pid, int* status, int options)

 printf(1, "\n  Part c) testing waitpid(int pid, int* status, int
options):\n");

      for (i = 0; i <5; i++) {
            pid_a[i] = fork();
            if (pid_a[i] == 0) { // only the child executed this code
                  printf(1, "\n The is child with PID# %d and I will exit
with status %d\n", getpid(), getpid() + 4);
                  exit(getpid() + 4);
            }
      }
      sleep(5);
      printf(1, "\n This is the parent: Now waiting for child with PID#
%d\n",pid_a[3]);
      ret_pid = waitpid(pid_a[3], &exit_status, 0);
      printf(1, "\n This is the partent: Child# %d has exited with status
%d\n",ret_pid, exit_status);
      sleep(5);
      printf(1, "\n This is the parent: Now waiting for child with PID#
%d\n",pid_a[1]);
      ret_pid = waitpid(pid_a[1], &exit_status, 0);
      printf(1, "\n This is the partent: Child# %d has exited with status
%d\n",ret_pid, exit_status);
      sleep(5);
      printf(1, "\n This is the parent: Now waiting for child with PID#
%d\n",pid_a[2]);
      ret_pid = waitpid(pid_a[2], &exit_status, 0);
      printf(1, "\n This is the partent: Child# %d has exited with status
%d\n",ret_pid, exit_status);
      sleep(5);
      printf(1, "\n This is the parent: Now waiting for child with PID#
%d\n",pid_a[0]);
      ret_pid = waitpid(pid_a[0], &exit_status, 0);
      printf(1, "\n This is the partent: Child# %d has exited with status
%d\n",ret_pid, exit_status);
      sleep(5);
      printf(1, "\n This is the parent: Now waiting for child with PID#
%d\n",pid_a[4]);
      ret_pid = waitpid(pid_a[4], &exit_status, 0);
      printf(1, "\n This is the partent: Child# %d has exited with status
%d\n",ret_pid, exit_status);

      return 0;
  }
```

(e) Extra Credit: The WNOHANG option was implemented in part (c) in the proc.c file, which is the file containing the actual implementation of the waitpid() function. Per the documentation at https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.bpxbd00/rtwaip.htm, the waitpid() function with the WNOHANG option enabled should return the status of the child matching the int pid argument immediately, or return with an error code indicating that the information was not available. Per the documentation, the WNOHANG causes waitpid() to return without causing the caller to be suspended, which means that waitpid() should not wait on the child process to finish executing. The way I've implemented the WNOHANG option is to return the exit status of the child assuming the status integer pointer is valid (e.g. non-null), and to suspend the sleep() call (which is used to wait for the child processes to exit) when WNOHANG is entered as an argument