Chris Hung Huynh, SID: 862024281
Edwin Leon, SID: 862132870

Lab1 Report

In lab1 we were asked to familiarize ourselves with xv6, modify the exit and wait system calls, and finally add the waitpid system call. First we will talk about the exit system call. The changes we had to implement to the exit system call was that instead of just exiting it had to store the terminated process status. The steps are as follows:

1. Added a new field called “int status” to the proc struct in the proc.h file
2. Modified the exit system call signature to “void exit(int status)” in user.h and defs.h files.
3. Modified sysproc.c file so that the sys_exit function would fetch an argument and return the status of the exited process.
4. Modified proc.c file so that inside the exit function, a status is assigned to the current process.
5. Modified files containing the old exit system call to reflect new changes.

```
// 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)
{
  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.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }
  curproc->status = status;
  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

```
int
sys_exit(void)
{
  int status;

  if(argint(0, &status) < 0)
        return -1;
  exit(status);
  return 0;  // not reached
}
```

The next system call we modified was wait. The changes we had to implement to the wait system call was that it had to return the terminated child exit status through the status argument. The steps we took to make this modification are as follows:

1. Modified the system call structure to “int wait(int* status)” in the user.h and defs.h file.
2. Modified sysproc.c file so that the sys_wait function would fetch an argument and return the process id of the terminated child.
3. Modified the proc.c file so that it would return the status argument of the terminated child.
4. Updated the files containing the old wait system call to reflect the new changes

```
// 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;
        if(status != NULL) {
                *status = p->status;
        }
        else {
                *status = 0;
        }
        release(&ptable.lock);
        return pid;
      }
    }

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

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

```
int
sys_wait(void)
{
  int *status;

  if(argptr(0,(char**) &status, sizeof(*status) < 0))
        return -1;
  return wait(status);
}
```

The next thing we did was create a new syscall function called waitpid. This syscall waits for a process containing the pid that matches the pid provided to the waitpid argument. The steps taken are as follows:

1. Added a new waitpid function signature ("int waitpid(int pid, int* status, int options)") to defs.h and user.h files.
2. Added the system call "SYSCALL(waitpid)" to the usys.S file.
3. Define the system call "SYS_waitpid" in the syscall.h file.
4. Added "extern int sys_waitpid(void)" and " [SYS_waitpid] sys_waitpid" to the syscall.c file.
5. Added the function "int sys_waitpid(void)" in the sysproc.c file, so that it would fetch arguments and return the process id of a specific terminated child.
6. Added the function "int waitpid(int pid, int* status, int options)" in the proc.c file, so that it would implement the same functionality as wait but the waitpid instead waits for a specific child to terminate.

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

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    same = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->pid != pid)
        continue;
      same = 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 != NULL) {
                *status = p->status;
        }
        else {
                *status = 0;
        }
        release(&ptable.lock);
        return pid;
      }
    }

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

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

```
int
sys_waitpid(void)
{
  int pid;
  int* status;
  int options;

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

Finally, after modifying the exit and wait system calls and adding the waitpid system call, we created a test program to test the correctness of our modifications and additions in the xv6 operating system.

1. Created a test.c file to test the new system calls.
2. Added the provided “int exitWait(void)” function in the test.c file to test the correctness of the exit and wait system calls.
3. Created the “int waitpidTest(void)” function in test.c file to test the correctness of our waitpid system call.
4. In the waitpidTest function, using a for-loop we called fork 5 times to create 5 different children or processes, and assigned its return value to an array (pid[5]).
5. Moreover, a message was printed showing the child’s process id and exit status.
6. In another for-loop we called the waitpid function so that the parent would wait for a specific child to terminate and the return value of the waitpid function would be assigned to “ret_pid”.
7. Moreover, a message was printed showing that the current process was a parent and printing the child’s process id and its exit status once the child process terminated.

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

int exitWait(void);
int waitpidTest(void);

int main(int argc, char *argv[])
{
        printf(1, "\n This program tests the correctness of the new wait and exit systemcalls\n");
        exitWait();
        printf(1, "\n This program tests the correctness of waitpid ststemcall\n");
        waitpidTest();
        exit(0);
        return 0;
}
```

```c
int exitWait(void) {
       int pid, ret_pid, exit_status;
       int i;
       // use this part to test exit(int status) and wait(int* status)

       printf(1, "\n  Parts a & b) testing exit(int status) and wait(int* status):\n");

       for (i = 0; i < 2; i++) {
              pid = fork();
              if (pid == 0) { // only the child executed this code
                     if (i == 0){
                            printf(1, "\nThis is child with PID# %d and I will exit with status %d\n", getpid(), 0);
                            exit(0);
                     }
                     else{
                            printf(1, "\nThis is child with PID# %d and I will exit with status %d\n" ,getpid(), -1);
                            exit(-1);
                     }
              }
              else if (pid > 0) { // only the parent executes this code
                     ret_pid = wait(&exit_status);
                     printf(1, "\n This is the parent: child with PID# %d has exited with status %d\n", ret_pid, exit_status);
              }
              else {  // something went wrong with fork system call
                     printf(2, "\nError using fork\n");
                     exit(-1);
              }
       }
       return 0;
}
```

```c
int waitpidTest(void){
        int exit_status;
        int ret_pid;
        int pid[5];
        int i;
        for(i = 0; i < 5; i++){
                if((pid[i] = fork()) == 0){
                        printf(1, "\n This is the child with PID# %d and I will exit with status %d\n", getpid(), i);
                        printf(1, "\n");
                        exit(i);
                }
        }
        for(i = 0; i < 5; i++){
                sleep(5);
                ret_pid = waitpid(pid[i], &exit_status, 0);
                printf(1, "\n This is the parent: child with PID# %d has exit status %d\n", ret_pid, exit_status);
        }
        return 0;
}
```

```
Booting from Hard Disk..xv6...
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
$ ./test

 This program tests the correctness of the new wait and exit systemcalls

  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

 This program tests the correctness of waitpid ststemcall

 This is the child with PID# 6 and I will exit with status 0


 This is the child with PID# 7 and I will exit with status 1


 This is the child with PID# 8 and I will exit with status 2


 This is the child with PID# 9 and I will exit with status 3


 This is the child with PID# 10 and I will exit with status 4


 This is the parent: child with PID# 6 has exit status 0

 This is the parent: child with PID# 7 has exit status 1

 This is the parent: child with PID# 8 has exit status 2

 This is the parent: child with PID# 9 has exit status 3

 This is the parent: child with PID# 10 has exit status 4
```