Mark Gameng

CS 450 – Duan Yue

Lab 2

exit() to exitStatus(int status)

Rather than just changing the existing exit system call and thereby, update all the code that used exit(), I just created a new exit call, named exitStatus(int status).

For proc.c, It's the same code as the original exit() but I just saved the exit status to the curproc. This saves the exit status of the current process.

```
void
exitStatus(int status)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  // store the exit status - Lab 2
  curproc -> status = status;
```

For proc.h, I then added int status in the struct.

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this
  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;              // so can save exit status - Lab 2
};
```

I then declared exitStatus in user level in user.h

```
void exitStatus(int status); // lab 2
```

I also updated usys.S to have exitStatus.

```
32  SYSCALL(exitStatus)
```

In syscall.c, I then added sys_exitStatus.

```
106  extern int sys_exitStatus(void); // lab 2
```

```
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,

// lab 2
[SYS_exitStatus]  sys_exitStatus,
```

Same thing in syscall.h

```
23  #define SYS_exitStatus 22 // lab 2
```

In sysproc.c, I just used a similar structure as sys_exit() and passed in an integer, status.

```
int
sys_exitStatus(void) // lab 2
{
  int status;

  if(argint(0, &status) < 0){
    return -1;
  }

  exitStatus(status);

  return 0; // not reached
```

I then defined it in defs.h

```
108  void              exitStatus(int); // lab 2
```

Now, exitStatus should be all implemented and what's left is testing …

wait() -> int wait(int *status) and adding int waitpid(int pid, int *status, int options)

Now, for updating wait and adding waitpid, its very similar to the process I did previously. Similar to exit, I made a new system call rather than updating wait, as I would have to update all the other instances of wait in xv6, which isn't that much compared to exit, but still.

For wait, I added int waitStatus, which was just the same code as the original wait, but I added code to pass back the status.

```
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){
        *status = p -> status; // passing back the status
      }

      release(&ptable.lock);
      return pid;
```

For waitpid, its similar to wait, but waits for a process with the given pid. Also, must wait for any process, meaning it doesn't have to be a child process.

```
for(;;){
  // Scan through table looki
  havekids = 0;
  for(p = ptable.proc; p < &pt
    //if(p->parent != curproc
    //  continue;

    if(p -> pid != pid){ // it
      continue;
    }
```

I also modified sysproc.c for waitStatus and waitpid.

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

  if (argptr(0, (void *) &status, sizeof(*status)) < 0){
    return -1;
  }

  return waitStatus(status);
}

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

  // pid is 0, status 1, options 2. cuz waitpid(pid, status, options)
  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);
}
```

Similar to exit, for waitStatus and waitpid, I modified user.h, usys.S, syscall.c, syscall.h, defs.h

```
27  // lab 2
28  void exitStatus(int status);
29  int waitStatus(int *status);
30  int waitpid(int pid, int *status, int options);
```

```
32  SYSCALL(exitStatus)
33  SYSCALL(waitStatus)
34  SYSCALL(waitpid)
```

```
107  // lab 2
108  extern int sys_exitStatus(void);
109  extern int sys_waitStatus(void);
110  extern int sys_waitpid(void);
```

```
135  // lab 2
136  [SYS_exitStatus]  sys_exitStatus,
137  [SYS_waitStatus]  sys_waitStatus,
138  [SYS_waitpid]     sys_waitpid,
139  };
```

```
// lab 2
#define SYS_exitStatus 22
#define SYS_waitStatus 23
#define SYS_waitpid 24
```

```
121  int             wait(void);
122  int             waitStatus(int*); // lab 2
123  int             waitpid(int, int*, int) // lab 2
```

Testing

For testing, I first used the given usertests by xv6. Running it gave no errors and outputs “ALL TESTS PASSED”, so I then started to make my own tests for the above functions, using test.c

To be able to run the test file, I needed to update the makefile.

```
168  UPROGS=\
169      _cat\
170      _echo\
171      _forktest\
172      _grep\
173      _init\
174      _kill\
175      _ln\
176      _ls\
177      _mkdir\
178      _rm\
179      _sh\
180      _stressfs\
181      _usertests\
182      _wc\
183      _zombie\
184      _test\
```

For test.c, this is my code.

```
printf(1, "Testing exitStatus(int status) and waitStatus(int *status)\n");
// list of childs, so i can reference their pids
//int pids[3] = {0, 0, 0};
int exitStatuses[3] = {5, 22, 13};
int pid;
int exitStatus_;

for(int i = 0; i < 3; i++){
    pid = fork();
    if (pid < 0){ // negative pid means fork failed
        printf(1, "fork failed. Retry test\n");
        exit();
    }
    //pids[i] = pid;
    if (pid == 0){ // child has their pid = 0
        printf(1, "(Child) exiting with status %d \n", exitStatuses[i]);
        exitStatus(exitStatuses[i]);
    }
}

for(int i = 0; i < 3; i++){
    pid = waitStatus(&exitStatus_);
    printf(1, "(Parent) Child with PID = %d exited with status %d \n", pid, exitStatus_);
}
```

```
printf(1, "Testing waitpid\n");
// list of childs, so i can reference their pids
int pids[3] = {0, 0, 0};
int options[3] = {7, 6, 5};

for(int i = 0; i < 3; i++){
    pid = fork();
    if (pid < 0){ // negative pid means fork failed
        printf(1, "fork failed. Retry test\n");
        exit();
    }
    if (pid == 0){ // child has their pid = 0
        printf(1, "(Child) exiting with status %d \n", exitStatuses[i]);
        exitStatus(exitStatuses[i]);
    }
    pids[i] = pid;
}

for(int i = 0; i < 3; i++){
    printf(1, "(Parent) Waiting for child with PID = %d to exit \n", pids[2 - i]);
    pid = waitpid(pids[2 - i], &exitStatus_, options[i]);
    printf(1, "(Parent) Child with PID = %d exited with status %d \n", pid, exitStatus_);
}
```

Running these tests in xv6, results in:

```
$ test
Testing exitStatus(int status) and waitStatus(int *status)
(Child) exiting with status 5
(Parent) Child with PID = 18 exited with status 5
(Child)(Child) exiting with status 13
(Parent) Child with PID = 20 exited with status 13
 exiting with status 22
(Parent) Child with PID = 19 exited with status 22
```

```
Testing waitpid
(Child) exiting with status 5
(Child(Parent) Waiting for child with PID = 23 t) exiting with status 22
(Child) exitino exit
g with status 13
(Parent) Child with PID = 23 exited with status 13
(Parent) Waiting for child with PID = 22 to exit
(Parent) Child with PID = 22 exited with status 22
(Parent) Waiting for child with PID = 21 to exit
(Parent) Child with PID = 21 exited with status 5
$
```

Aside from overlap from print statements, due to running at the same time, the outputs are correct and show that the methods exitStatus(int status), waitStatus(int *status), and waitpid(int pid, int *status, int options) I implemented are working.

In the picture above, the parent is waiting for child (pid = 13) to exit and doesn't run any code until it does. Once that the pid = 13 exits, can see it print above, overlapping with the parent, then the parent resumes execution and prints out that the child exited and then proceeds to wait for the next child.

I also ran diff -r original_xv6 lab2_xv6 to show all my code changes. That file, lab2diff.txt, is included in the submission.