Lab 1 Report

Alp Can Aloglu

SID : 862190749

E-mail: aalog001@ucr.edu

- **All code changes you made on xv6 source code** (hint: `diff -r original_xv6 your_xv6`)
- **Detailed explanation about these changes**

Changes in sysproc.c:

```
[aalog001@sledge ~]$ diff -r xv6_original/sysproc.c xv6/sysproc.c
19,20c19,25
<   exit();
<   return 0;  // not reached
---
>     // Copied Kill sys_kill
>     int status;
passing the user-level argument to the kernel-level argument we use argint()
>     if(argint(0, &status)< 0){
>        return -1;
>     }
>     exit(status); used the new signature of exit function
>     return 0;  // not reached
26c31,38
<   return wait();
---
>     // Found a description on how to pass struct pointer to system call
>     // https://stackoverflow.com/questions/53383938/pass-struct-to-xv6-system-call
>     // used this example.
>     int *status; declared the status pointer of type int
Passing arguments from user-mode to kernel-mode using argint()
>     if(argptr(0,(void*)&status,sizeof(*status))<0){>        return -1;
>     }
>   return wait(status);used the new signature of wait function
91a104,126
>
> // Wait pid implementation
> int
> sys_waitpid(void)
> {
>     int pidToWait,options; declared variables for the process pid and option of type int
>     int *status ;declared the status pointer of type int
```

Lab 1 Report

Passing arguments from user-mode to kernel-mode by using argint() for type int and argptr() fro type of pointer

```
>     if(argint(0,&pidToWait)< 0)
>        return -1;
>     if(argptr(1,(void *)&status,sizeof(*status)) < 0)
>        return -1;
>     if(argint(2,&options)< 0)
>        return -1;
>     return waitpid(pidToWait, status, options);Called the waitpid function in Kernel mode
> }
```

Changes ins proc.c:

```
[aalog001@sledge ~]$ diff -r xv6_original/proc.c xv6/proc.c
228c228
< exit(void)
---
> exit(int status) We change the the function signature by adding the argument status of type int
233a234,236
>    // Set exit status
>     curproc->exitStatus = status; Adding this line of code in order to set the current process's status to the status passing in to the exit function
>
273c276
< wait(void)
---
> wait(int *status) Changing the function signature by adding the argument status of type int pointer
278c281
<
---
>
288a292,293
If the process goes to the ZOMBIE state:
>         if(status) Checking if the status is not nullptr
>             *status = p->exitStatus; If the status pointer is not null, then we change it to the process's status
534a540,591
>
```

```
> // Waits for specific process to finish
> int
> waitpid(int pidToWait, int *status,int options)
> {
>     struct proc *p;
>     int haveproc, pid; Changing havechild to haveproc because in waitpid
function we do not care about the child process but we care about an specific
process with a given pid.
>     struct proc *curproc = myproc();
>
>     acquire(&ptable.lock);
>     for(;;){
>       // Scan through table looking for exited process with given pid
>       haveproc = 0;Changing havechild to haveproc
>       for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
>         if(p->pid != pidToWait)
>           continue;
>         haveproc = 1;Changing havechild to haveproc
>         if(p->state == ZOMBIE){
>           // Found one.
If the process goes to the ZOMBIE state:
>           // Status null check
>           if(status)Checking if the status is not nullptr
>             *status = p->exitStatus; If the status pointer is not null, then we
change it to the process's status
>           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);
>           return pid;
>         }
>       }
>
>       // No point waiting if we don't have any process with given pid.
>       // Also if current process is killed already, no point waiting.
>       if(!haveproc || curproc->killed){Changing havechild to haveproc
>         release(&ptable.lock);
>         return -1;
>        }
>
>        // Wait for process to exit.  (See wakeup1 call in proc_exit.)
>        sleep(curproc, &ptable.lock);  //DOC: wait-sleep
>     }
> }
```

Changes in defs.h:

```
[aalog001@sledge ~]$ diff -r xv6_original/defs.h xv6/defs.h
107c107
< void          exit(void);
---
> void          exit(int);Modified exit forward declaration
120c120
< int           wait(void);Modified wait forward declaration
---
> int           wait(int*);
122a123,124
> int           waitpid(int,int*,int); // Added waitpid forward declaration
```

Changes in user.h:

```
[aalog001@sledge ~]$ diff -r xv6_original/user.h xv6/user.h
6,7c6,7
< int exit(void) __attribute__((noreturn));
< int wait(void);
---
> int exit(int) __attribute__((noreturn)); // Modified exit
> int wait(int*); //Modified wait
25a26,27
> int waitpid(int,int*,int); // Added waitpid
```

Changes in usys.S:

```
diff -r xv6_original/usys.S xv6/usys.S
31a32,33
> SYSCALL(waitpid) // Added waitpid to SYSCALL list
```

```
diff -r xv6_original/wc.c xv6/wc.c
```

Changes in syscall.h:

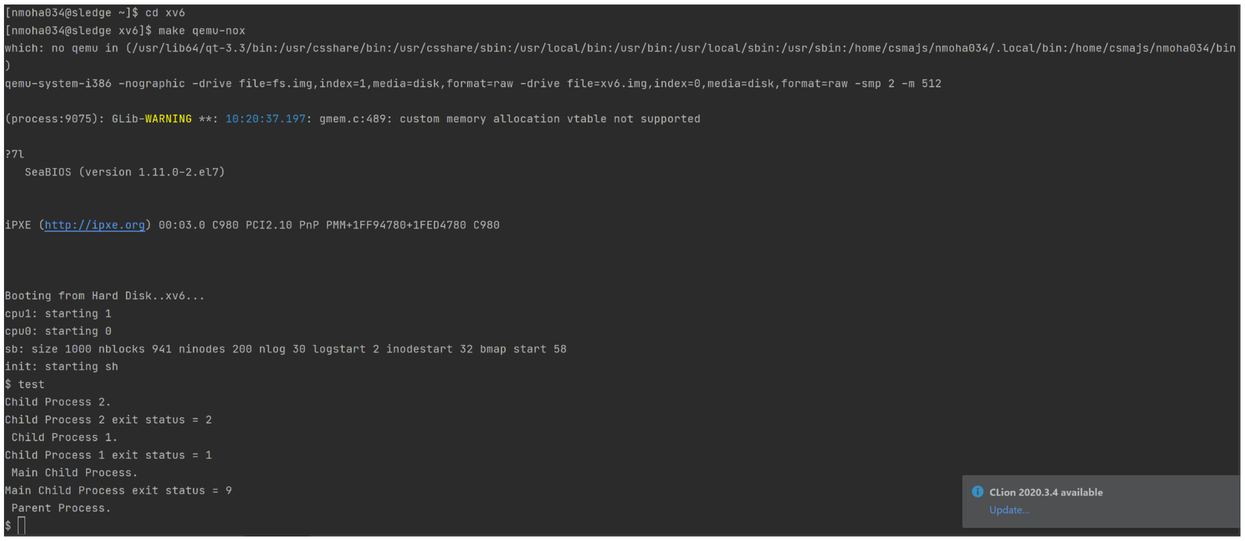
```
diff -r xv6_original/syscall.c xv6/syscall.c
105a106,107
> extern int sys_hello(void); //Toy
> extern int sys_waitpid(void); // Added waitpid
128a131,132
> [SYS_hello]   sys_hello, //Toy
> [SYS_waitpid] sys_waitpid, // Added waitpid

diff -r xv6_original/syscall.h xv6/syscall.h
22a23,24
> #define SYS_hello  22 // toy
> #define SYS_waitpid 23 // Added waitpid
```

Changes in syscall.c:

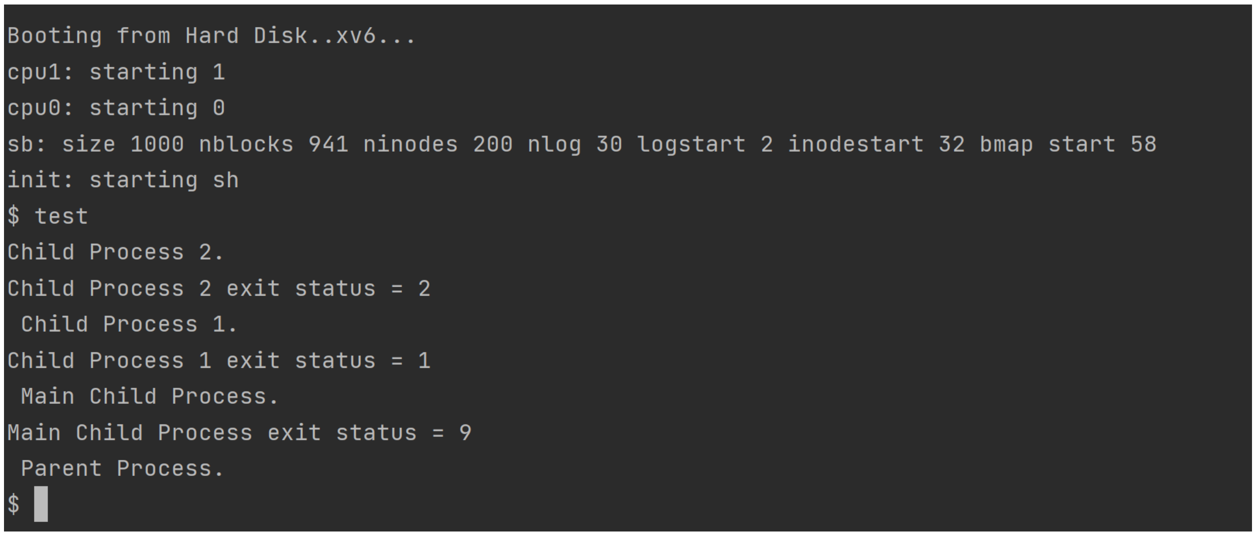
```
diff -r xv6_original/syscall.c xv6/syscall.c
105a106,107
> extern int sys_hello(void); //Toy
> extern int sys_waitpid(void); // Added waitpid
128a131,132
> [SYS_hello]   sys_hello, //Toy
> [SYS_waitpid] sys_waitpid, // Added waitpid
```

- **Screenshots about how you run the related program(s) and results**

Lab 1 Report



```
[nmoha034@sledge ~]$ cd xv6
[nmoha034@sledge xv6]$ make qemu-nox
which: no qemu in (/usr/lib64/qt-3.3/bin:/usr/csshare/bin:/usr/csshare/sbin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/csmajs/nmoha034/.local/bin:/home/csmajs/nmoha034/bin
)
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512

(process:9075): GLib-WARNING **: 10:20:37.197: gmem.c:489: custom memory allocation vtable not supported

?7l
   SeaBIOS (version 1.11.0-2.el7)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF94780+1FED4780 C980



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
Child Process 2.
Child Process 2 exit status = 2
 Child Process 1.
Child Process 1 exit status = 1
 Main Child Process.
Main Child Process exit status = 9
 Parent Process.
$
```



```
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
Child Process 2.
Child Process 2 exit status = 2
 Child Process 1.
Child Process 1 exit status = 1
 Main Child Process.
Main Child Process exit status = 9
 Parent Process.
$
```

test.c:

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

int main(int argc, char *argv[]){
   int pid = fork();
   int *status = (int *)(1);
   int *status2 = (int *)(1);
   int *status3 = (int *)(1);
   if(pid == 0){
        int pid_c1 =fork();

        if(pid_c1 == 0)  {
           int pid_c2 = fork();

           if (pid_c2 == 0){
               printf(1,"Child Process 2.\n");
               exit(2);
           }
           else{
               waitpid(pid_c2,status3,0);
               printf(1,"Child Process 2 exit status = %d\n ", *status3);
               printf(1,"Child Process 1.\n");
               exit(1);
           }
       }
       else{
           waitpid(pid_c1,status2,0);
           printf(1,"Child Process 1 exit status = %d\n ", *status2);
           printf(1,"Main Child Process.\n");
           exit(9);
       }
   }
   else {
       waitpid(pid,status,0);
       printf(1,"Main Child Process exit status = %d\n ", *status);
       printf(1,"Parent Process.\n");
   }
   exit(0);
}
```

We have called fork() system call three times to create three child processes. After each fork() system call, we check the pid, and if it is zero(meaning that the process is a child process)or is not a zero (meaning that the process is a parent process), we specify what to do next accordingly.
For example, we made the first child process(main child) by pid = fork(), then we checked on pid value. If it is not zero(parent process), we call waitpid system call(waitpid(pid, status, 0)) which means the parent process should wait for the child process with that specific pid (which is the main child process). Then we terminated each child with different exit status, and printed out the corresponding exit status when the waitpid of each child returns the exit status of the child in question. For example, the main child process terminated with a status value of 9, and we can see that the test program prints number 9 after the waitpid that is waiting for the main child process. It means that the parent process waited for the main child process to end (exit(9)), and then resumed and continued the execution.