Lab Report: Implementing System Calls

Win Thant Tin Han SID: 862258943

10/29/2023

1 Changes and Actions Taken

1.1 Part 1: Changing the 'exit' System Call

The first part of the lab involved changing the 'exit' system call to have the signature `void exit(int status)`. This modification required updates to several files, including `user.h`, `defs.h`, `sysproc.c`, `proc.c`, and all user space programs that used the 'exit' system call.

1.1.1 proc.h aand proc.c

The key section of this part of the lab is to define a variable in proc.h to store the exit status.
When we exit the current process, we need to store exit status of terminated process. An exited process remains in the zombie state until its parent calls wait() to find out it exited.
`curproc->exit_status = status;`
The line above was added into the `void exit(int status)` definition in `proc.c` to store the exit status after turning into zombie.

1.1.2 user.h and defs.h

We also have to change the system call type in `user.h` and `defs.h`, from `int exit(void);` to `void exit(int);`

1.1.3 sysproc.c

The exit system call must act as previously defined (i.e., terminate the current process) but it must also store the exit status of the terminated process in the corresponding structure change exit() signature to void exit(int status). `int sys_exit(void)` was modified to:

- Initialize a variable 'status' to store the exit status.
- Use argint to retrieve the exit status argument from the user space. This extracts an integer argument from the user and stores it in 'status'.
- Call the exit function with the provided exit status.

1.2 Part 2: Updating the 'wait' System Call

The second part focused on updating the 'wait' system call to have the signature `int wait(int *status)`. This change aimed to allow the parent process to wait for any of its child processes to terminate and retrieve the exit status. The system call would return the child process's PID or -1 if no child existed or an error occurred.

1.2.1 proc.c

We need to make the system call return the required value. So the child's exit status has to be stored and returned if status is not NULL.

1.2.2 usys.S

We need to define a new system call number for wait since it was not declared.

1.2.3 user.h and defs.h

We also have to change the system call type in `user.h` and `defs.h`, from `int wait(void);` to `int wait(int*);`

1.2.4 sysproc.c

The wait system call must prevent the current process from execution until any of its child processes is terminated (if any exists) and return the terminated child exit status through the status argument. We return the terminated child proc's exit_status through the status pointer argument:

- Declare a pointer to an integer (this will store the exit status of the child process)
- Use argptr to validate and retrieve the status argument from the user space. argptr is a function that helps ensure that the user-provided argument is valid.
- If the status argument is invalid, return -1 to indicate an error.
- Then, we call the wait function to wait for a child process to exit and return its PID. The exit status of the child process is stored in the 'status' pointer.

1.3 Part 3: Adding the 'waitpid' System Call

The final part introduced the 'waitpid' system call with the signature `int waitpid(int pid, int *status, int options)`. This system call acted similarly to 'wait' but allowed the waiting process to specify the PID of the process it was waiting for. The system call would block until the specified process terminated.

1.3.1 usys.S

We need to define a new system call number for waitpid.

1.3.2 syscall.h and syscall.c

We need to add SYSCALL(waitpid) since it was not declared here.

1.3.3 user.h and defs.h

A new system call has to be declared. `int waitpid(int, int *, int);` was added.

1.3.4 sysproc.c

It defines the 'sys_waitpid' function, which will be called when the 'waitpid' system call is invoked by a user program.

- 'pid': This will store the process ID of the child process you want to wait for.
- 'options': This will store the options for the 'waitpid' function. It's initialized to 0, which means it will have the default behavior.
- 'status': This is a pointer to an integer that will be used to store the exit status of the child process. It's passed as an argument to the system call.
- The code then checks the arguments passed to the 'sys_waitpid' function. It uses the 'argint' and 'argptr' functions to retrieve the 'pid' and 'status' arguments from the user program. If either of these operations fails (returns a negative value), the function returns -1 to indicate an error.
- if the arguments are successfully retrieved, the code calls the 'waitpid' function with the specified 'pid', 'status', and 'options'. The 'waitpid' function is typically a system call that waits for a specific child process to terminate and stores its exit status in the 'status' variable.

1.3.5 proc.c

The system call must wait for a process (not necessarily a child proc) with a pid that equals to one provided by the pid argument. This system call must act like wait system call with the following additional properties:

1. The system call must wait for a process (not necessary a child process) with a pid that equals to one provided by the pid argument.
2. The return value must be the process id of the process that was terminated or -1 if this process does not exist or if an unexpected error occurred.

We are required only to implement a blocking waitpid where the kernel prevents the current process from execution until a process with the given pid terminates.

1.4 Part 4: Implement WNOHANG and create a version of CELEBW02

1.4.1 proc.c

- Modified waitpid in proc.c.
- The original waitpid sleeps when pid is running and return pid when status==zombie. Now instead of going to sleep, we return 0 to indicate that the pid is still running by adding an option called WNOHANG,.
- Checks if waitpid returns 0. If yes, child is still running and continues once waitpid returns the pid.

1.4.2 sysproc.c

- Changed the part where options is not 0 by default.

1.4.3 test.c

- added a `int waitPidWNOHANG(void);` declaration in `test.c` to test WNOHANG implementation.

2 Results and What I learned

From this lab, I learned the basics about how to implement basic system calls. I had to deal with many files, so it was difficult for me but once I understood the concept, I think it was pretty fun and interesting. The results obtained after implementing the WNOHANG for waitpid:

Test4:

```
$ test 4
 This program tests the correctness of your lab#1
  Part d) testing waitpid(int pid, int* status, int WNOHANG):
 This is child with PID# 8
 This is the parent: Now waiting for child with PID# 8
 This is the parent: Process found but child PID# 8 is still running.
 This is child with PID# 8 and I will exit with status 8
 This is the parent: Child# 8 has exited with status 8, expected: 8
```