Recitation 5

TA Hanxiong Chen
hc691@rutgers.edu

Process

Brief review

- fork(): create child process by copying everything in parent process memory area to another memory space.
  - One call two returns
  - Parent process: pid != 0 ; child process: pid = 0
- exec(): replaces the program in the current process with a brand new program.
  - Only return if it is failed. It returns to the caller.
  - Use exec() in child process

Process

Zombie process: **a zombie process or defunct process is a process that has completed execution but still has an entry in the process table (PCB)**.

Orphan process: An orphan process is a computer **process whose parent process has finished or terminated**, though it remains running itself.

What happens after a child process becomes orphan?

How does a process become zombie process?

Which one is even worse?

Process

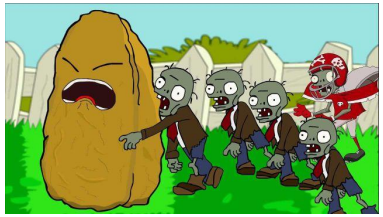
What happens after a child process becomes orphan?

In a Unix-like operating system any orphaned process will be immediately adopted by the special `init` system process. This operation is called re-parenting and occurs automatically.

How does a process become zombie process?

If the parent process doesn't wait for child process, the child process will be a zombie process.

Which one is even dangerous, why??



Cannot be killed by KILL command.

Process

Wait / waitpid

- Once a child process is dead, it will send a `SIGCHLD` signal to the parent process.
- Parent process can read status information of dead child process by wait/waitpid system call.
- After this is done, child process will be entirely removed from PCB
- The wait call may be executed in sequential code, but it is commonly executed in a handler for the `SIGCHLD` signal, which the parent receives whenever a child has died. (WHY?)
- If the parent explicitly ignores `SIGCHLD` by setting its handler to `SIG_IGN` (rather than simply ignoring the signal by default) or has the `SA_NOCLDWAIT` flag set, all child exit status information will be discarded and no zombie processes will be left

Process (cont.)

Example of ignore child status info:

signal(SIGCHLD, SIG_IGN);

or

```
struct sigaction act;
act.sa_handler = something;
act.sa_flags = SA_NOCLDWAIT;
sigaction (SIGCHLD, &act, NULL);
```

Process

```
int main()
{
        …
        int pid = fork();
        if (pid < 0)
                perror(“error”);
        else if (pid == 0)
        {
                printf(“This is a child process.\n”);
        }
        else
        {
                printf(“This is a parent process.\n”);
                waitpid(pid, &status, WUNTRACED);
        }
        return 0;
}
```

```
int main()
{
        int pid = fork();
        if (pid < 0)
                perror(“error”);
        else if (pid == 0)
        {
                execv(“filepath”, “arguments”);
                exit(EXIT_FAILURE);
        }
        else
        {
                printf(“This is a parent process.\n”);
                waitpid(pid, &status, WUNTRACED);
        }
        return 0;
}
```

Process

Is it necessary to copy everything all the time when we do fork()? Can we optimize that?

Yes!

We can do “Copy-On-Write” (COW)!

HW 5

Write your own ls command.

Sample code time!

directory / file

1. Use d_type in dirent struct
   a. If xxx->d_type == DT_DIR;
   b. else if xxx->d_type == DT_REG
2. Use stat

Eg.
```
int is_regular_file(const char *path)
{
        struct stat path_stat;
        if (stat(path, &statbuf) != 0)
                return 0;
        return S_ISREG(path_stat.st_mode);
 }
```

Eg.
```
int is_Directory(const char *path)
{
        struct stat statbuf;
        if (stat(path, &statbuf) != 0)
                return 0;
        return S_ISDIR(statbuf.st_mode);
 }
```

HW 6

Write your own ps command.

Modify the ls we wrote last week except output /proc

Then open the file 'status', look for the uid section and extract the owner's uid.
Using pwd.h, determine the name of the user who owns the process and print it out as well.

HW 6 (cont.)

0. Sample solution:

```
char * base = "/proc"
DIR * stuff = opendir(base, RD_ONLY)
dirent * pidnumber = NULL;

char * newCmdline = NULL;
char * workingName = NULL;

int fd = -1;
```

```
do
{
        pidnumber = readdir(stuff);
        if( pidnumber != NULL && pidnumber->d_type == DT_DIR) {
                newCmdline = ... malloc strlen(base)+strlen(pidnumber->d_name)+9;
                newCmdline[0] = '\0';
              strcat(newCmdline, base);
                strcat(newCmdline, "/");
                strcat(newCmdline, pidnumber->d_name);
                strcat(newCmdline, "/cmdline");

                fd = open(newCmdline, RD_ONLY);

                ... read loop ...
               ...while read from fd != 0, printf it out …
                close(fd);
        }
} while (pidnumber != NULL);
```

... this gets you all command lines run for all pids for all procs on the system
printf(command run and its pid(i.e. directory name of cmdline))

HW 6 (cont.)

1. Then open the file 'status', look for the uid section and extract the owner's uid. Using pwd.h, determine the name of the user who owns the process and print it out as well.

open status alongside/after cmdline (but before clocking your readdir loop! readdir is destructive!) read through and parse the status file looking for 'uid'

Sample:

```
while reading in status file ...
if buffer[i] == 'u'
if bufferLength - i >= 2
if (buffer[i+1] == 'i' && buffer[i+2] == 'd')
    ... can start reading in uid that called this code ... w00t!

printf( command run, its pid and the uid that called it) ..
```

boring .. want userNAME, not UID .. :^P .. but only place in system this information is together is in passwd file ... very well, then...

struct passwd * getUname = getpwuid(UID parsed out of status above)

now you can print out:
1. command run (from cmdline file)
2. username that called it (passwd->pw_name)
   .. for every current PID

Congrats .. you wrote basic ps

HW 6 (cont.)

2. Then open the file *schedstat* and read in order: time spent running on CPU (in nanoseconds), time spent waiting on a runqueue, # of times context switched

(be careful of decimals! you may want to check the status file to be sure your degree is correct)

Then, for some fun times, print out some stats.

Maybe also add command line options! Like default ps only prints out info for YOUR procs by default - so can get current uid within your code using getuid and comb through /proc and only print out status and schedstat information for stuff whose uid matches

References

https://www.gmarik.info/blog/2012/orphan-vs-zombie-vs-daemon-processes/

http://alumni.cs.ucr.edu/~drougas/code-samples-tutorials/signals.c

https://www.win.tue.nl/~aeb/linux/lk/lk-5.html