CS214
Recitation
Sec.7

Oct. 24, 2017

Topics

1. wait()

2. Signals

3. HW5: Implementing “ps”

System Call wait()

System call to *wait for a child*

- pid_t wait (int *status)

wait() are used to *wait for state changes* in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be:

the child terminated; *the child was stopped by a signal*;

or *the child was resumed by a signal*.

System Call wait()

A process that calls ***wait()*** will:

- **block the calling process**(if all of its children are *still running*)
- **return immediately with the termination status of a child** (if a child has *terminated* and is waiting for its parent to accept its return code)
- **return immediately with an error** (if it *doesn't have any child* processes)

Return Value:

wait(): on success, returns the *process ID* of the terminated child; on error, -1 is returned

Zombie processes

- A child process that terminates, but has not been waited for by its parent becomes a "**zombie**".
- A parent accepts a child's return code by executing **wait()**
- The kernel maintains a **minimal set of information** in the process table about the zombie process (PID, termination status, resource usage information, showing up as **Z** in ***ps -a***) in order to allow the parent to later perform a wait to obtain information about the child.

Orphan processes

- A process whose parent process no more exists
- When a parent process is died, it is soon adopted by init process (*PID 1*)
- The parent process is either finished or terminated without waiting for its child process to terminate is called an **orphan process**

Zombie process example

```c
// A C program to demonstrate Zombie Process.
// Child becomes Zombie as parent is sleeping
// when child process exits.
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // Fork returns process id
    // in parent process
    pid_t child_pid = fork();

    // Parent process
    if (child_pid > 0)
        sleep(50);

    // Child process
    else
        exit(0); // Finishes execution

    return 0;
}
```

Orphan process example

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main ()
{
    pid_t pid; //pid is the return value of fork()
    int count=0;
    pid=fork();
    if (pid < 0)
        printf("error in fork!\n");
    else if (pid == 0) {
        printf("i am the child process, my process id is %d\n",getpid());
        printf("i am the child process, the process id of my parent is %d\n",getppid());
        count++;
    }
    else {
        printf("i am the parent process, my process id is %d\n",getpid());
        count++;
    }
    printf("count: %d\n",count);
    return 0;
}
```

parent process is finished by the time the child asks for its parent's pid, so the getppid() will get 1;
if you want parent process to wait for child process, call wait()

Wait() example

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

void prExit(int status)
{
    if(WIFEXITED(status))
        printf("normal termination\n");
    else if(WIFSTOPPED(status))
        printf("child stopped, signal no.= %d\n",
            WSTOPSIG(status));
    else if(WIFSIGNALED(status))
        printf("abnormal termination, "
            "signal no.= %d\n", WTERMSIG(status));
}
```

```c
int main()
{
    pid_t pid; //pid is the return value of fork()
    int status;
    pid=fork();
    if (pid < 0)
        printf("error in fork!\n");
    else if (pid == 0) {
        exit(0);
        //abort();
        //status /= 0;
    }
    else {
        wait(&status); prExit(status);
    }
    return 0;
}
```

/*SIGCHLD/

/*SIGABRT*/

/*SIGFPE*/

Signals

When the kernel recognizes an **unexpected/unpredictable asynchronous events**, it sends a **signal** to the process.

Events are called interrupts:

- floating point error
- death of a child
- interval timer expired (alarm clock)
- control-C (termination request)
- control-Z (suspend request)

Signals (Cont.)

What are signals for?

- When a program forks into 2 or more processes, rarely do they execute independently
- The processes usually require some form of synchronization, often handled by signals.

What are the two sources of signals?

- Machine interrupts (such as typing the keyboard or hardward breakdown)
- The program itself, other programs or the user (sent by system functions)

Signals (Cont.)

What are the most well known signals and what do they do?

Signal	Default Action	Comment
SIGINT	Terminate	Interrupt from keyboard
SIGSEGV	Terminate/Dump core	Invalid memory reference.
SIGKILL	Terminate (cannot ignore)	Kill
SIGCHLD	Ignore	Child stopped or terminated.
SIGSTOP	Stop (cannot ignore)	Stop process.
SIGCONT		Continue if stopped.

Sending a Signal

Using the misleadingly named "**kill**" command:

- *kill [-signal] pid*

Signal can be specified by the number or name without the SIG

If no signal is specified, kill sends the TERM signal to the process

Examples:

```
kill -QUIT 8883
kill -STOP 78911
kill -9 76433        (9 == KILL)
```

HW5.0 - Implementing "ps"

0. Modify the *ls* we wrote this 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.

PS – where is process information

/proc - “proc filesystem” is a pseudo-filesystem which provides an interface to kernel data structures. It is commonly mounted at /proc. It includes all the information about processes. The information for each process is stored in a folder name by its PID:

```
-sh-4.2$ ls /proc
1      1296   16406  21232  2473   27847  4263  6318  779   9449
10     12972  16442  21247  2475   27859  43    6320  7830  9452
1005   13     16446  21265  24870  27861  4329  6330  784   9453
1011   1307   16453  21267  24990  27883  4331  6331  7841  946
10251  1312   16456  21283  25040  27903  449   6335  785   9462
10353  13327  16465  21295  25088  2793   4551  6350  786   9471
1073   1334   16518  21315  2556   2794   4560  6358  7868  9476
```

PS – where is process information (Cont.)

file '*stat*'

```
/proc/[pid]/stat
       Status information about the process.  This is used by ps(1).
       It is defined in the kernel source file fs/proc/array.c.

       The fields, in order, with their proper scanf(3) format speci-
       fiers, are listed below.  Whether or not certain of these
       fields display valid information is governed by a ptrace
       access mode PTRACE_MODE_READ_FSCREDS | PTRACE_MODE_NOAUDIT
       check (refer to ptrace(2)).  If the check denies access, then
       the field value is displayed as 0.  The affected fields are
       indicated with the marking [PT].

       (1) pid  %d
                 The process ID.

       (2) comm  %s
                 The filename of the executable, in parentheses.
                 This is visible whether or not the executable is
                 swapped out.

       (3) state  %c
                 One of the following characters, indicating process
                 state:

                 R  Running

                 S  Sleeping in an interruptible wait
```

PS – where is process information (Cont.)

file '*status*'

/proc/[pid]/status

Provides much of the information in */proc/[pid]/stat* and */proc/[pid]/statm* in a format that's easier for humans to parse. Here's an example:

```
$ cat /proc/$$/status
Name:   bash
Umask:  0022
State:  S (sleeping)
Tgid:   17248
Ngid:   0
Pid:    17248
PPid:   17200
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    100     100     100     100
FDSize: 256
Groups: 16 33 100
NStgid: 17248
NSpid:  17248
NSpgid: 17248
NSsid:  17200
VmPeak:     131168 kB
VmSize:     131168 kB
VmLck:           0 kB
VmPin:           0 kB
VmHWM:       13484 kB
VmRSS:       13484 kB
```

PS – where is process information (Cont.)

file '*schedstat*'

includes time spent running on CPU (in nanoseconds), time spent waiting on a run queue, # of times context switched.

Use the comand */proc/<pid>/schedstat* and get the following output:

```
-sh-4.2$ cat /proc/76/schedstat
17882 298 2
```

HW5.0 - Implementing "ps"

Coding reference:

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

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

int fd = -1;
```

```
while
{
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);
}
}
```

```
do(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))

HW5.1 - Implementing "ps"

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

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)

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

Congrats! ... you wrote basic “ps”

HW5.2 - Implementing "ps"

2. Then open the file *schedstat* and read in order: time spent running on CPU (in nanoseconds), time spent waiting on a run queue, # 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 staff whose uid matches