Process management via shell

Week 03 – Lab 3

https://goo.gl/6w9KIR

Process creation

Four principal events that cause processes to be created:

- System initialization
- Execution of a process creation system call by a running process.
- A user request to create a new process.
- Initiation of a batch job.

Process termination

Typical conditions which terminate a process:

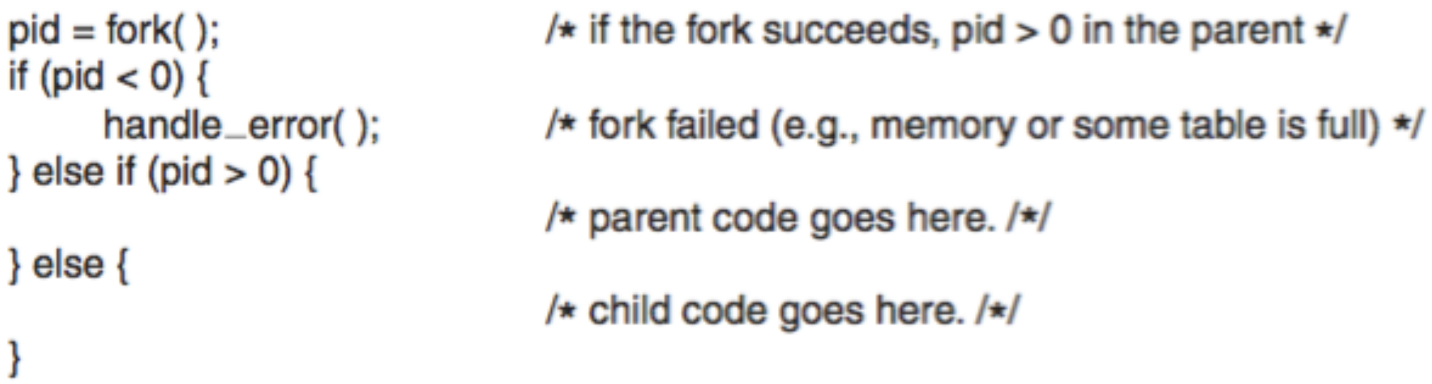
- Normal exit (voluntary).
- Error exit (voluntary).
- Fatal error (involuntary).
- Killed by another process (involuntary).

fork system

- fork() - a system call that creates an exact copy of the original process
- The forking process is called **the parent** process. The new process is called **the child** process
- The parent and the child processes initially have identical memory images, variables, registers, open files etc.

How do you know which process is a child?

- fork() returns 0 to the child process
- fork() returns a non-zero integer (Process ID - PID) to the parent process
- fork() returns a negative integer in case of error



```
pid = fork( );                /* if the fork succeeds, pid > 0 in the parent */
if (pid < 0) {
        handle_error( );      /* fork failed (e.g., memory or some table is full) */
} else if (pid > 0) {
                              /* parent code goes here. /*/
} else {
                              /* child code goes here. /*/
}
```

Exercise 1

- Compile and run the following program. Run it 10 times. Explain the output

```
#include <stdio.h>
int num = 0;
int main(int argc, char*argv[]){
    int pid;
    pid = fork();
    printf("%d\n", num);
    if(pid == 0){       /*child*/
        num = 1;
    }else if(pid > 0){  /*parent*/
        num = 2;
    }
    printf("%d\n", num);
}
```

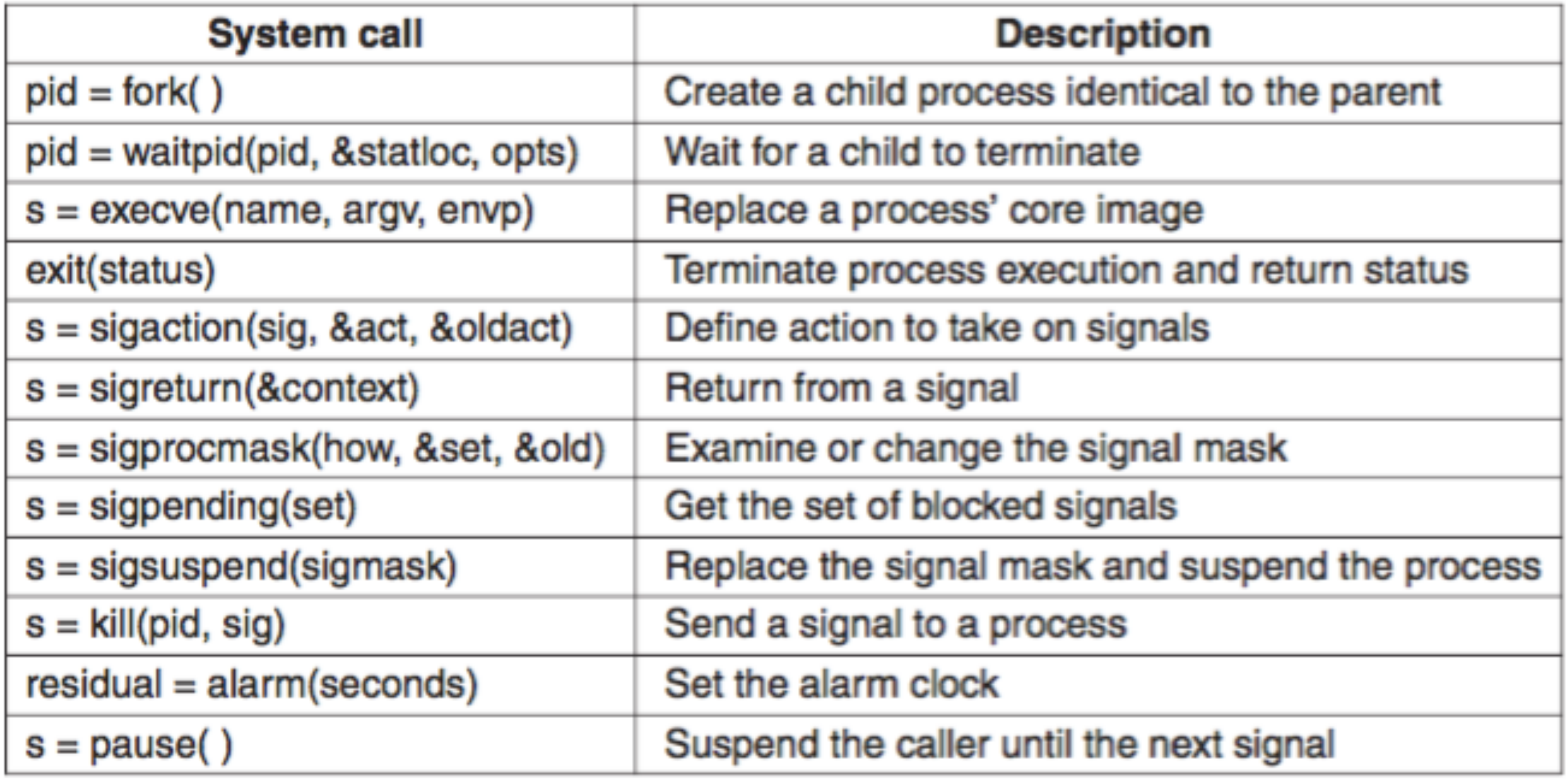
Simple fork server

```
while(1) {  // main accept() loop
        sin_size = sizeof their_addr;
        new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
        if (new_fd == -1) {
            perror("accept");
            continue;
        }

        inet_ntop(their_addr.ss_family,
            get_in_addr((struct sockaddr *)&their_addr),
            s, sizeof s);
        printf("server: got connection from %s\n", s);

        if (!fork()) { // this is the child process
            if (send(new_fd, "Hello, world!", 13, 0) == -1) {
                perror("send");
            }
            …
            …
            close(new_fd);
            exit(0);
        }
        close(new_fd);  // parent doesn't need this
    }
```

Exercise 2

- Write an infinite loop that sleeps for 1 second and then increments the counter on each iteration. Every 10 iterations print “Parent ” + the value of the counter and fork a new process. Within the child process simply print “Child” + the value of the counter variable and exit

Other process management system calls



System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

Simplified source code of the shell

```
while (TRUE) {                              /* repeat forever /*/
    type_prompt( );                         /* display prompt on the screen */
    read_command(command, params);          /* read input line from keyboard */

    pid = fork( );                          /* fork off a child process */
    if (pid < 0) {
        printf("Unable to fork0);           /* error condition */
        continue;                           /* repeat the loop */
    }

    if (pid != 0) {
        waitpid (–1, &status, 0);           /* parent waits for child */
    } else {
        execve(command, params, 0);         /* child does the work */
    }
}
```

exec system call

- Replaces image core of the current process by the file name in the first parameter of the call
- Library procedures that use exec system call: execl, execv, execle, execve (they all differ in number and types of parameters, but all use exec sys call underneath)

Exercise 3

- Write a C program that creates a file called dummy.txt in the current directory using execv() library call. Compile and run the program

Exercise 4

- Write a program that creates 2 processes. The second process should be created via fork(). After the fork, the child should execute (exec*) the binary /bin/ls. The parent process should print a hello world message after forking. How many hello world messages will be printed?

Extra exercise

- Implement the following server and client:
  - **./shell_sv <server-port>**

    The server creates a listening socket that listens on the specified port and accepts client requests containing shell commands. The server concurrently handles clients, executing each client’s command, and passing the results back across the client’s socket.
  - **./shell_cl <server-host> <server-port> ’shell command’**

    The client connects to the shell server, sends it a single shell command, reads the results sent back across the socket by the server, and displays the results on stdout.

Hints

- Easy execution of a shell command: execl("/bin/sh", "sh", "-c", cmd, (char *) NULL);
- To have a shell command send stdout (and stderr!) to the socket, use dup2().
- Checking all system calls for errors will save you a lot of time
- Need to write debugging output in the server? Open /dev/tty