Service Oriented Computing

Lab 1. Process Control and Signal Handling

Lab 1. Process Control and Signal Handling

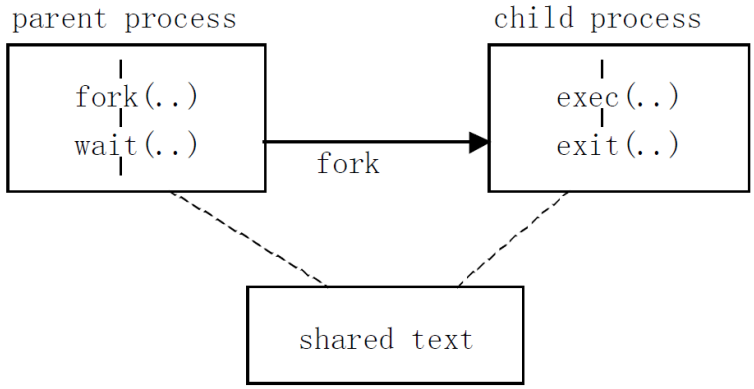
Objectives

- Understand the process control mechanism of Linux
- Learn the usage of fork(), exec(), wait()
- Understand the signal handling and zombie process

Practice

1. fork() procedure

Please make a program to examine fork() procedure in simple. Use sample code presented in class.




```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)      {
            pid_t pid;
            if((pid = fork()) <    0)      {
                        printf("fork error\n");
                        exit(0);
            } else if (pid == 0) {
                        printf("CHILD\n");
            } else {
                        sleep(2);
                        printf("PARENT \n");
            }
            sleep(10);
}
```

Requirements

1) Please check fork() procedure with execution as background process (i.e. ./fork_test &)
2) Please identify Process ID and Parent Process ID using getpid() and getppid()
3) Please execute program as background process(i.e. ./fork_test &) and figure out the Process ID and Parent Process ID with "ps –f" command (Within 10 seconds after execution of process)
4) Please implement below simple code about exec() and check out the result output before entering the '2. Process Control'

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
        int x = 3;
        int err = execl(argv[1], argv[1], argv[2], NULL);    //WARN: Actually, second argument should not include the path.
        printf("error = %d, x = %d\n", err, x);
}
```

```
#include <stdio.h>

int main(int argc, char* argv[]) {
        printf("Hello, %s!\n", argv[1]);
}
```

Example of execution

```
[ancl@anclab2 Lab]$ ./lab1_1a /home/ancl/Lab/lab1_1b ANCL
Hello, ANCL!
[ancl@anclab2 Lab]$ ./lab1_1a /home/ancl/Lab
error = -1, x = 3
```

For your reference

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid(void), ----- returns process ID
pid_t getppid(void),------returns parent process ID
pid_t getpgid(void),------returns process group ID
uid_t getuid(void),------returns process user ID
gid_t getgid(void),------returns group ID
```

2. Process Control

Please make a sample shell to examine fork(), exec() and wait() procedure in simple. Use sample code presented in class.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUF_SIZE 1024
#define NUM_ARG 256

void parse_str(char* str, char* arg[NUM_ARG]) {
        int cnt = 0;
        char *p = strtok(str, " ");
        if((arg[cnt++] = p) == NULL)
                return;
        while((p = strtok(NULL, " ")) != NULL)
                arg[cnt++] = p;
        arg[cnt] = NULL;
}

int main(int argc, char* args[]) {
        char buf[BUF_SIZE];
        char* arg[NUM_ARG];
        int status;
        if(argc != 2)
                exit(1);
        while(1) {
                pid_t pid;
                printf("%s > ", args[1]);
                gets(buf);
                parse_str(buf, arg);
                if(strcmp(arg[0], "exit") == 0)
                        exit(0);
                if((pid = fork()) == 0)
                        execvp(arg[0], arg);          ----------ⓐ
                else
                        wait(&status);                ----------ⓑ
        }
}
```

Requirements

1) Please compile the source code and execute the program
2) Please modify ⓐ to execl(), execlp(), execv() and show the execution with any command.

   Constraint 1. When you are using absolute path (not for $PATH), set directory as /bin

   Constraint 2. When you are using list (not vector class), it is acceptable to use only one arguments (i.e. ls –al).
3) Please modify ⓑ to waitpid(pid_t pid, int *status, int options) with setting option field as 0 and print out the status (using the function WEXITSTATUS(status) in 'sys/wait.h')

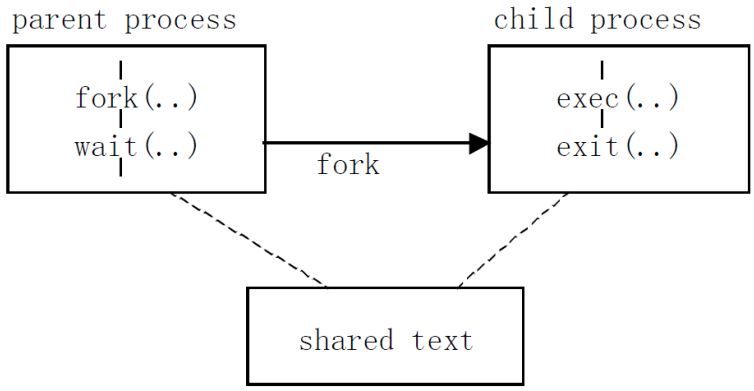
For your reference

- $PATH is environment variable for Linux system. When you register your own folder to $PATH, you can access to the execution file in that folder without the path. You can check it with 'echo $PATH' command. Many Linux commands that we use like 'ls', 'cp', etc., are located in /bin folder. And with environment variable $PATH, we can access it easily unlike '/bin/ls'. There are many registrations, and these are divided by ':'.
- We can append $PATH with our own folder. With 'export PATH=$PATH:your_folder' command like below example. After enroll the folder to $PATH, we can use it without path like below figure.

```
[ancl@anclab2 bin]$ export PATH=$PATH:/home/ancl/Lab/
[ancl@anclab2 bin]$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/home/ancl/bin:/home/ancl/Lab/
[ancl@anclab2 bin]$ lab1_2 myshell
myshell >
```

- 4 version of the exec() primitive:

1	Absolute path	Search in $PATH
n parameters (**l**ist)	execl ()	execlp ()
1 array (**v**ector)	execv ()	execvp ()



1) **int fork():** creates a copy of the current process called the child process. They may share the same text segment.

   fork( ) is called once, but returns twice:

- -1: if an error occurred
- 0: returned to the child process
- PID: returned to the parent process

The child process inherits but cannot influence the environment of the parent process.

2) **int exec( ):** the child process can then use the exec( ) to start a new program. Every process in the Unix except the first process is created by the combined efforts of fork() and exec().

3) **int exit(int status):** child process finishes by passing status to kernel. Kernel can transfer status to parent which is executing wait( ).

4) **int wait(int *status):** the parent process can wait(…) for the child process to finish. Otherwise if parent process doesn't execute wait(…), the child will become a zombie process when the child process ends with exit(…). But a zombie process will last only until the parent process terminates. If the parent process ends first, the parent process ID of the child (an orphan now) is set to 1. wait(status) returns the process ID of the child process. status gets the status value of exit(status) of the terminating child process.

3. Signal Handling

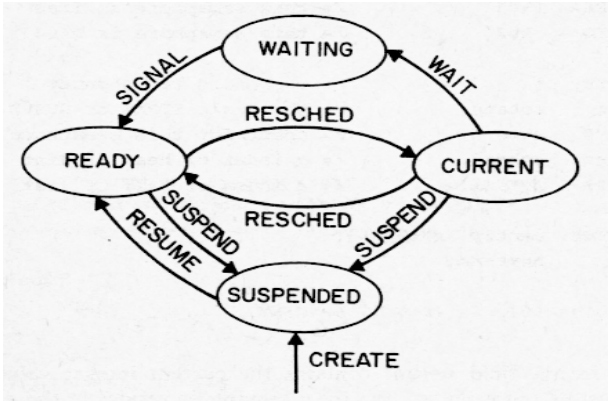
Please make a sample signal handling program to examine signal() procedure in simple. Use sample code presented in class.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void sigint_handler (int sig);
int main()
{
        int state;
        int num=0;
        signal(ⓐ);
        while(1)
        {
                printf("%d : waiting\n", num++);
                sleep(2);
                if(num>5)
                        break;
        }
        return 0;
}

/* Signal handling function */
void sigint_handler(int sig)
{
        signal(ⓐ);
        printf("delivered signal : %d \n", sig);
}
```

Requirements

1) Please complete the source code with filling the blank ⓐ. In this program, a signal function registers 'int sigint_handler(int sig)' as signal handler of SIGINT.
2) Please run the process as background and check the result with continuous send of SIGINT signal with "ctrl+c". And think about transition status of the process as shown in the below figure.



For your reference

Signals are software interrupts and occur asynchronously at any time. For example, a segmentation fault produces a SIGSEGV signal. By default, SIGSEGV dumps core + terminate the process. SIGALRM signal is generated by using unsigned int alarm(unsigned int seconds). Default is to terminate the process.

1) Each signal is generated from process→process or from kernel → process.
2) Each signal has a name defined in "/usr/include/sys/iso/signal_iso.h", description and default action,

- **How to generate signals:**

  process → process:

  ① System call: int kill(int pid, int sig). It is because signals can be produced by program using kill( ) that signals are also called as software interrupts. kill( ) is not only used to kill a process, but also used in most cases to send a signal sig to a process pid.

  ② Command: kill –9 pid, Generate SIGTERM. Don't confuse kill command with system call of the same name.

  kernel → process:

  ① Keyboard: control-C or Delete→SIGINT. Control-backslash→SIGQUIT. Control-Z→SIGTSTP.

  ② Hardware: SIGSEGV or SIGFPE for Floating Point Arithmetic errors.

  ③ Software condition: Special software conditions are noticed by kernel such as SIGURG for out-of-band data on a socket.

  3) Condition to be invoked

  Alarm clock, bus error, death of child process, Illegal instruction, Interrupt character, I/O, kill, power fail, quit, stop, bad argument to system call, trace trap, etc.

  4) signal()
     - to indicate the events among processes
     - Signals related to socket
       - SIGINT: interrupt key input (Ctrl+C)
       - SIGFPE: error for floating point mathematical processing
       - SIGIO: indicate the I/O status
       - SIGURG: indicate urgent socket such as arrival of out-of-band message

- **How to handle signals in detail:**

  1) When sig occurs, signal handler is called to handle the condition.

```
#include <signal.h>
void (*signal (int signum, void (*sighandler)(int) ) ) (int);
```

  2) 2 special values for the func argument:

  SIG_DFL: perform the default action,

SIG_IGN: ignore the signal except SIGKILL and SIGSTOP.

4. Zombie Process

Please make a sample Zombie Process program to examine the condition of creation of zombie process in simple. Use sample code presented in class.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
        pid_t pid;
        int data=10;
        pid=fork();
        if(pid<0)
                printf("fork fail. (process id : %d) \n", pid);
        printf("fork success. (process id : %d) \n", pid);
        if(pid==0) /* child process */
        {
                data += 10;
                printf("child data : %d \n", data);
        }
        else        /* parent process */
        {
                data-=10;
                sleep(20); /* waiting 20 seconds */
                printf("parent data : %d \n", data);
        }

        return 0;
}
```

Requirements

1) Please execute program as background process (i.e. ./ZombieTest &)
2) Please check process status with “ps –f” command within 20 second and check process status with “ps –s”. Compare the status and explain the result
3) Please implement below source code and think about why below source code doesn’t make zombie process. And modify the source code to make value of ‘status’ to 3.

```
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
int main()
{
        pid_t pid, child;
        int data=10;
        int state;
        pid=fork();
        if(pid<0)
                printf("fork fail. (process id : %d) ", pid);
        printf("fork success. (process id : %d) \n", pid);
        if(pid==0)    /* child process */ {
                data+=10;
                printf("child data : %d \n", data);
        }
        else          /* parent process */ {
                data-=10;
                child = wait(&state); /* waiting for child process to terminate*/
                printf("child process id = %d \n", child);
                printf("return value = %d \n", WEXITSTATUS(state));
                sleep(20); /* waiting 20 seconds */
                printf("parent data : %d \n", data);
        }
        return 0;
}
```

For your reference

1) When a process terminates, it is not immediately removed from the system

- The process which created it may be interested in its return value
- It gets to zombie state:
  - It has finished execution
  - It stays there until its parent has received its termination status
  - If the parent process has already finished, then the process is adopted by process 1

2) Zombie processes must be dealt with

- Otherwise, they eventually exhaust the system's resources

3) To allow a zombie to die, its parent must explicitly "wait" for its children to finish

- It can block until one of its children has died
- Or it can setup a signal handler for the SIGCHLD signal to be asynchronously notified

4) wait() waits for any child to complete

- It also handles an already completed (zombie) child
- status indicates the process' return status

5) waitpid() gives you more control:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- pid species which child is waited for (pid=-1 means any)
- option=WNOHANG makes the call return immediately, if no child has already completed (otherwise, use option=0)