Operating System (CSC 3150)

Tutorial 1

YUANG CHEN

E-MAIL: YUANGCHEN@LINK.CUHK.EDU.CN

Target

In this tutorial, we will setup environment of virtual machine with Ubuntu 16.4, and learn to create process in user mode.

- Environment setup
- Process
- Process creation
- Parent and child process

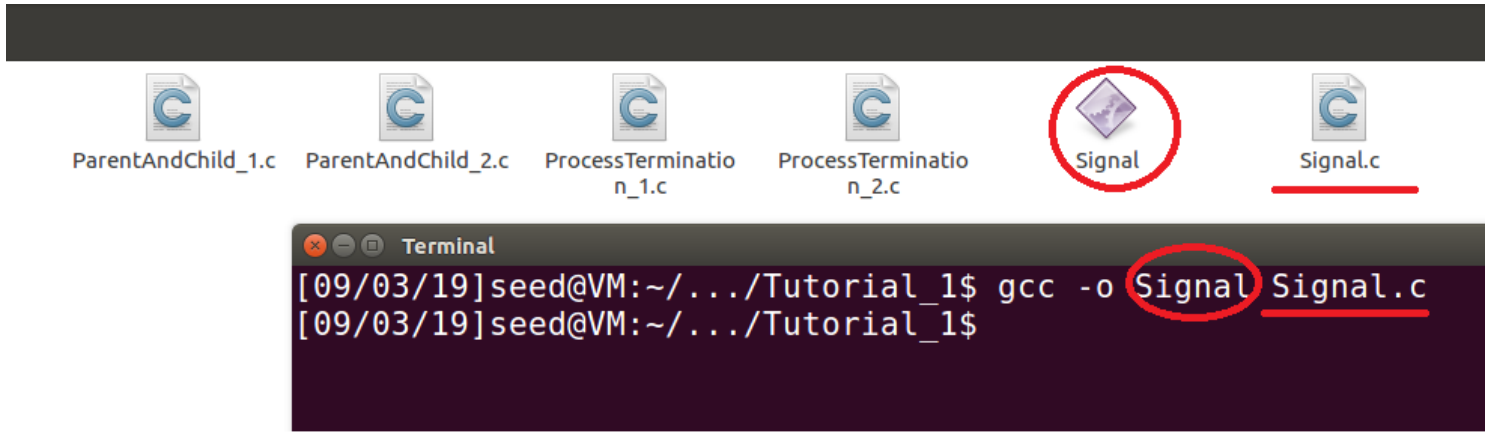
Environment setup

- Compile c file in Linux (gcc command)

  Compile Command: gcc –o ExecutableFile CFileName.c

  Execute Command: ./ExecutableFile (./ means running in current folder)

  After running gcc command, an executable file will be generated.



Environment setup

- Makefile: defining some compile commands into make command

  Command: make (compile all .c files as executable files)

  Command: make clean (for all executable files which are ending with '.c', remove them)

  ERROR here

Makefile (~/Documents/Tutorial_1) - gedit

```
CFILES:= $(shell ls|grep .c)
PROGS:=$(patsubst %.c,%,$(CFILES))

all: $(PROGS)

%:%.c
        $(CC) -o $@ $<

clean:$(PROGS)
        rm $(PROGS)
```

cuments Tutorial_1

execve, execve.c, Makefile, ParentAndChild_1, ParentAndChild_1.c, ParentAndChild_2, ParentAndChild_2.c, ProcessTerminatio n_1, ProcessTerminatio n_1.c, ProcessTerminatio n_2, ProcessTerminatio n_2.c, Signal, Signal.c, test, test.c, WIFEXITED, WIFEXITED.c

Terminal

```
[09/03/19]seed@VM:~/.../Tutorial_1$ make
cc -o execve execve.c
cc -o ParentAndChild_1 ParentAndChild_1.c
cc -o ParentAndChild_2 ParentAndChild_2.c
cc -o ProcessTermination_1 ProcessTermination_1.c
cc -o ProcessTermination_2 ProcessTermination_2.c
cc -o Signal Signal.c
cc -o test test.c
cc -o WIFEXITED WIFEXITED.c
[09/03/19]seed@VM:~/.../Tutorial_1$
```

execve.c, Makefile, ParentAndChild_1.c, ParentAndChild_2.c, ProcessTerminatio n_1.c, ProcessTerminatio n_2.c, Signal.c, test.c, WIFEXITED.c

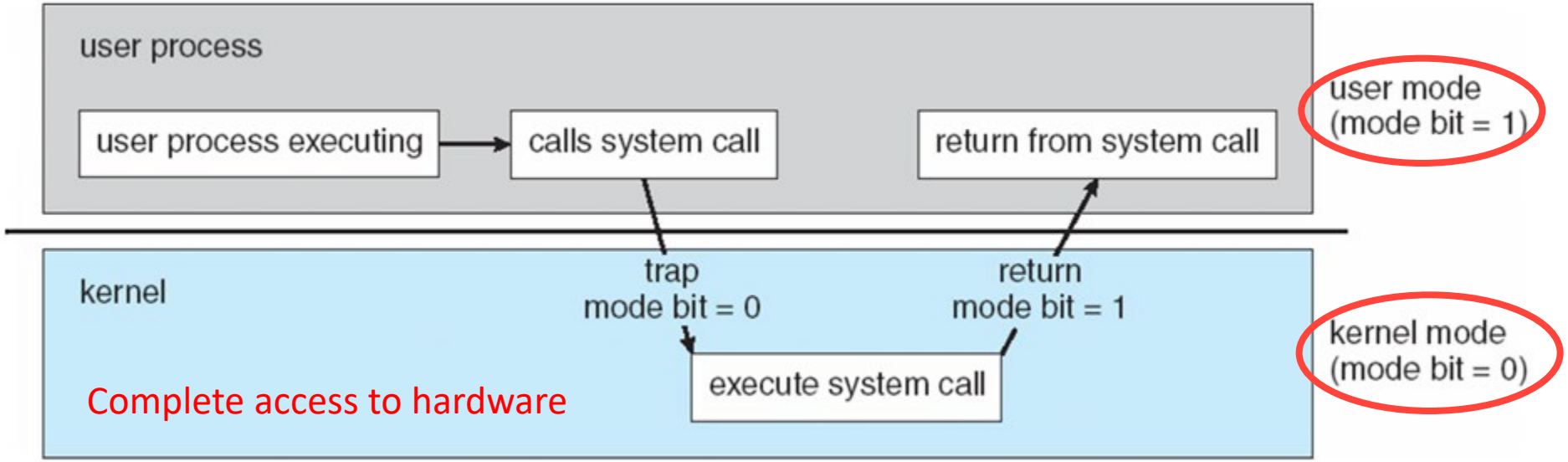
Terminal

```
[09/03/19]seed@VM:~/.../Tutorial_1$ make clean
rm execve execve ParentAndChild_1 ParentAndChild_2 ProcessTermination_1 ProcessT
ermination_1 ProcessTermination_2 ProcessTermination_2 Signal test WIFEXITED
```

Process

- User Mode
- Kernel Mode

用户态程序将一些数据值放在寄存器中, 或者使用参数创建一个堆栈(stack frame), 以此表明需要操作系统提供的服务. 用户态程序执行陷阱指令， CPU切换到内核态, 并跳到位于内存指定位置的指令, 这些指令是操作系统的一部分, 他们具有内存保护, 不可被用户态程序访问， 这些指令称之为陷阱(trap)或者系统调用处理器(system call handler). 他们会读取程序放入内存的数据参数, 并执行程序请求的服务系统调用完成后, 操作系统会重置CPU为用户态并返回系统调用的结果



Process

- Process: Program in execution

  Multiple processes: Concurrent vs Parallel. Processor vs CPU vs Core

- In multitasking operating systems, processes (running programs) need a way to create new processes, e.g. to run a program.

- Process state:
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Process

- Each process is named by a process ID number. Generally, process is identified and managed via a **process identifier (pid)**

- A unique process ID is allocated to each process when it is created.

- The lifetime of a process ends when its termination is reported to its parent process. At that time, all of the process resources, including its process ID, are freed.

Process Creation

- Processes are created with the fork system call (so the operation of creating a new process is sometimes called forking a process).

- The child process created by fork is an exact clone of the original parent process, except that it has its own process ID.

Process Creation

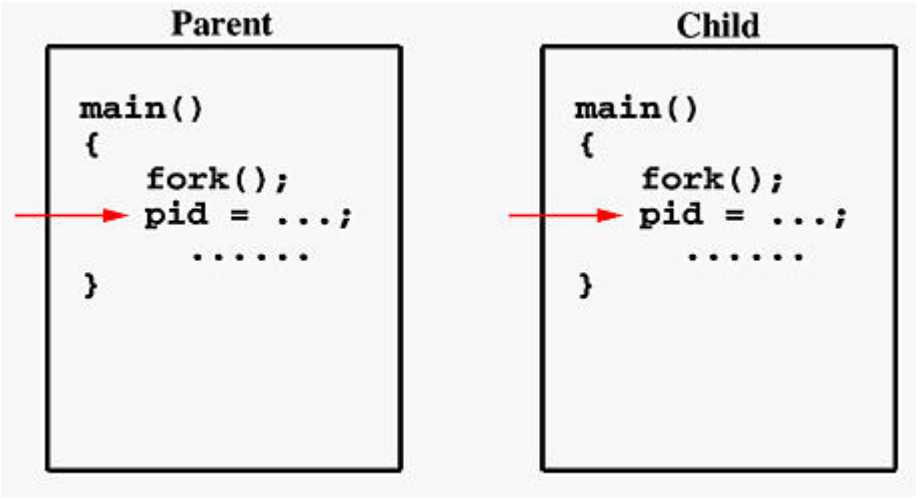
- Functions (Unix-like system):
  - **pid_t fork(void)** system call creates new process

- Return value
  - fork() returns -1, the creation of a child process was unsuccessful.
  - fork() returns a zero to the newly created child process.
  - fork() returns a positive value, the process ID of the child process, to the parent.

- The returned process ID is of type **pid_t** defined in ”sys/types.h”.

Parent and Child Process

- If the call to fork() is executed successfully, Unix will
  - make two identical copies of address spaces, one for the parent and the other for the child.
  - Both processes will start their execution at the next statement following the fork() call.



Parent and Child Process 1 - pid

Both parent and child process start execution from next statement after fork call.

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


int main(int argc, char *argv[]){

    char buf[50] = "Original test strings";
    pid_t pid;

    printf("Process start to fork\n");
    pid=fork();

    if(pid==-1){
        perror("fork");
        exit(1);
    }
    else{

        //Child process
        if(pid==0){
            strcpy(buf, "Test strings are updated by child.");
            printf("I'm the Child Process: %s\n", buf);
            exit(0);
        }

        //Parent process
        else{
            sleep(3);
            printf("I'm the Parent Process: %s\n", buf);
            exit(0);
        }
    }

    return 0;
}
```

Terminal

```
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o ParentAndChild_1 ParentAndChild_1.c
[09/03/19]seed@VM:~/.../Tutorial_1$ ./ParentAndChild_1
Process start to fork
I'm the Child Process: Test strings are updated by child.
I'm the Parent Process: Original test strings
[09/03/19]seed@VM:~/.../Tutorial_1$
```

一个进程，包括代码、数据和分配给进程的资源。fork（）函数通过系统调用创建一个与原来进程几乎完全相同的进程，也就是两个进程可以做完全相同的事，但如果初始参数或者传入的变量不同，两个进程也可以做不同的事。

Parent and Child Process 1

Set original test string

Update string when executing child process

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


int main(int argc, char *argv[]){

    char buf[50] = "Original test strings";
    pid_t pid;

    printf("Process start to fork\n");
    pid=fork();

    if(pid==-1){
        perror("fork");
        exit(1);
    }
    else{

        //Child process
        if(pid==0){
            strcpy(buf, "Test strings are updated by child.");
            printf("I'm the Child Process: %s\n", buf);
            exit(0);
        }

        //Parent process
        else{
            sleep(3);
            printf("I'm the Parent Process: %s\n", buf);
            exit(0);
        }
    }

    return 0;
}
```

```
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o ParentAndChild_1 ParentAndChild_1.c
[09/03/19]seed@VM:~/.../Tutorial_1$ ./ParentAndChild_1
Process start to fork
I'm the Child Process: Test strings are updated by child.
I'm the Parent Process: Original test strings
[09/03/19]seed@VM:~/.../Tutorial_1$
```

The test string is updated in child process only.
The parent and child processes have separate address spaces.

Parent and Child Process 2

Open ▾ Save

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

int main(int argc, char *argv[]){

    pid_t pid;

    printf("Process start to fork\n");
    pid=fork();

    if(pid==-1){
        perror("fork");
        exit(1);
    }
    else{

        //Child process
        if(pid==0){
            printf("I'm the Child Process, my pid = %d, my ppid = %d\n",getpid(), getppid());
            exit(0);
        }

        //Parent process
        else{
            sleep(3);
            printf("I'm the Parent Process, my pid = %d\n",getpid());
            exit(0);
        }
    }
    return 0;
}
```

- **getpid()** returns PID of calling system.
- **getppid()** returns PID of the parent of calling system.

Let parent process to sleep for 3s, and then print messages.

Terminal

```
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o ParentAndChild_2 ParentAndChild_2.c
[09/03/19]seed@VM:~/.../Tutorial_1$ ./ParentAndChild_2
Process start to fork
I'm the Child Process, my pid = 2708, my ppid = 2707
I'm the Parent Process, my pid = 2707
[09/03/19]seed@VM:~/.../Tutorial_1$
```

Parent and Child Process 2

Open ▾ Save

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

int main(int argc, char *argv[]){

    pid_t pid;

    printf("Process start to fork\n");
    pid=fork();

    if(pid==-1){
        perror("fork");
        exit(1);
    }
    else{

        //Child process
        if(pid==0){
            printf("I'm the Child Process, my pid = %d, my ppid = %d\n",getpid(), getppid());
            exit(0);
        }

        //Parent process
        else{
            //sleep(3);
            printf("I'm the Parent Process, my pid = %d\n",getpid());
            exit(0);
        }
    }
    return 0;
}
```

Parent and child process runs concurrently after forking.

Terminal

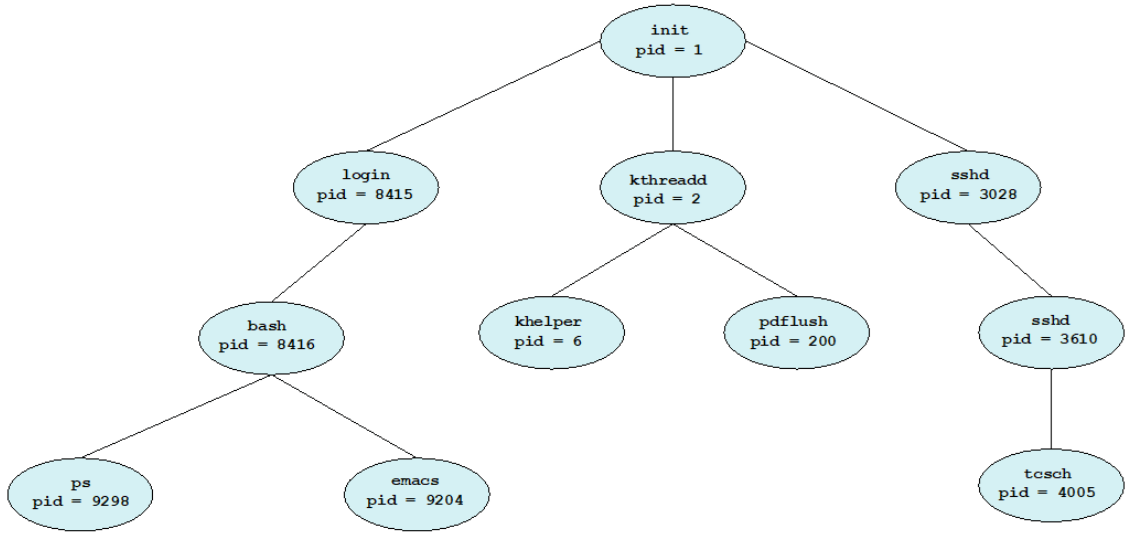
```
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o ParentAndChild_2 ParentAndChild_2.c
[09/03/19]seed@VM:~/.../Tutorial_1$ ./ParentAndChild_2
Process start to fork
I'm the Parent Process, my pid = 2729
I'm the Child Process, my pid = 2730, my ppid = 1378
[09/03/19]seed@VM:~/.../Tutorial_1$
```

Why?

Parent process terminates, and the child process is inherited by init process, whose pid is 1378 in my example. It may change every time rebooting the OS.
When testing in Mac OS, it might be -1.

Parent and Child Process

A Tree of Processes in Linux



Operating System Concepts – 9th Edition 3.21 Silberschatz, Galvin and Gagne ©2013

Process Termination

- Process executes last statement and asks the operating system to delete it (**`exit()`**)
  - Output data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system

- If no parent waiting, then terminated process is a **zombie**
- If parent terminated, processes are **orphans**

Process Termination 1 - orphan

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


int main(int argc, char *argv[]){

    pid_t pid;

    printf("Process start to fork\n");
    pid=fork();

    if(pid==-1){
        perror("fork");
        exit(1);
    }
    else{

        //Child process
        if(pid==0){
            printf("I'm the Child Process:\n");
            sleep(10);
            printf("\t My pid is:%d.   My ppid is:%d\n", getpid(), getppid());
            exit(0);
        }

        //Parent process
        else{
            sleep(3);
            printf("I'm the Parent Process:\n");
            printf("\t My pid is:%d\n", getpid());
            exit(0);
        }
    }

    return 0;

}
```

Let parent process to terminates ahead of child process

```
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o ProcessTermination_1 ProcessTerminati
on_1.c
[09/03/19]seed@VM:~/.../Tutorial_1$ ./ProcessTermination_1
Process start to fork
I'm the Child Process:
I'm the Parent Process:
         My pid is:2788
[09/03/19]seed@VM:~/.../Tutorial_1$
```

Parent process terminates, and the child process becomes orphans, and will be adopted by init process.

参数status用来保存被收集进程退出时的一些状态，它是一个指向int类型的指针。但如果我们对这个子进程是如何死掉的毫不在意，只想把这个僵尸进程消灭掉，我们就可以设定这个参数为NULL

进程一旦调用了wait，就立即阻塞自己，由wait自动分析是否当前进程的某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程，wait就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait就会一直阻塞在这里，直到有一个出现为止。

Process Termination 2

- Functions: (defined in "sys/wait.h")
  - pid_t **wait** *(*int *status_ptr*)*
  - pid_t **waitpid** (pid_t pid, int *status_ptr, int options)

0 / WNOHANG / WUNTRACED

- 0 skips the option, and keeps waiting till the specified child process terminates.
- WNOHANG demands status information immediately. If status information is immediately available on an appropriate child process, waitpid() returns this information. Otherwise, waitpid() returns immediately with an error code indicating that the information was not available. In other words, it checks child processes without causing the caller to be suspended.
- WUNTRACED reports on stopped child processes as well as terminated ones.

- Differences
  - wait() requests status for any child process
  - waitpid() requests status for specific child process.
  - The waitpid() function behaves the same as wait() if the pid argument is (pid_t) -1 and the options argument is 0.

https://stackoverflow.com/questions/33508997/waitpid-wnohang-wuntraced-how-do-i-use-these/34845669

若在option中设置WNOHANG位，与那么该系统调用就是非阻塞的，也就是说会立刻返回而不是等待子进程的状态发生变化

https://www.cnblogs.com/33debug/p/7017215.html

https://linux.die.net/man/2/waitpid

Process Termination 2

pid>0时，只等待进程ID等于pid的子进程，(即指定wait函数等待的到底是具体哪个子进程)
不管其它已经有多少子进程运行结束退出了，只要指定的子进程还没有结束，waitpid就会一直等下去。
pid=-1时，等待任何一个子进程退出，没有任何限制，此时waitpid和wait的作用一模一样。
pid=0时，等待同一个进程组中的任何子进程，如果子进程已经加入了别的进程组，waitpid不会对它做任何理睬。
pid<-1时，等待一个指定进程组中的任何子进程，这个进程组的ID等于pid的绝对

- Return value
  - wait() or waitpid() returns PID of child process when the status of a child process is available.
  - If unsuccessful, wait() or waitpid() returns -1.

- When waitpid() returns with a valid process ID (pid), below macros can analyze the status referenced by the status argument.
  - int **WIFEXITED** (int status)
  - int **WIFSIGNALED** (int status)
  - int **WIFSTOPPED** (int status)
  - etc.

Process Termination 2 – waitpid()

Open

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


int main(int argc, char *argv[]){

    pid_t pid;
    int status;

    printf("Process start to fork\n");
    pid=fork();

    if(pid==-1){
        perror("fork");
        exit(1);
    }
    else{

        //Child process
        if(pid==0){
            printf("I'm the Child Process:\n");
            sleep(10);
            printf("\t My pid is:%d.   My ppid is:%d\n", getpid(), getppid());
            exit(0);
        }

        //Parent process
        else{
            waitpid(pid, &status, 0);
            printf("I'm the Parent Process:\n");
            printf("\t My pid is:%d\n", getpid());
            printf( "\t Child process exited with status %d \n", status );
            exit(0);
        }
    }

    return 0;

}
```

Terminal

```
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o ProcessTermination_2 ProcessTerminati
on_2.c
[09/03/19]seed@VM:~/.../Tutorial_1$ ./ProcessTermination_2
Process start to fork
I'm the Child Process:
         My pid is:2814.   My ppid is:2813
I'm the Parent Process:
         My pid is:2813
         Child process exited with status 0
[09/03/19]seed@VM:~/.../Tutorial_1$
```

Child process sleeps 10s and parent process will suspend until status information of child process is available.

Process Signals

- Linux supports the standard signals listed below.
  - SIGQUIT 3
  - SIGKILL 9
  - SIGTERM 15
  - SIGSTOP 19
  - etc.

https://stackoverflow.com/questions/50299429/c-reporting-which-signal-terminated-a-child

Process Signals

- Send a signal to caller
  - int **raise** (int sig)

important!!!

- Evaluate child process's status (zero or non-zero)
  - int **WIFEXITED** (int status)
  - int **WIFSIGNALED** (int status)
  - int **WIFSTOPPED** (int status)

https://blog.csdn.net/duyuguihua/article/details/38986197

三者的比较

https://www.cnblogs.com/xiao0913/p/11846212.html

信号表

- Evaluate child process's returned value of status argument(exact values)
  - int **WEXITSTATUS** (int status)
  - int **WTERMSIG** (int status)
  - int **WSTOPSIG** (int status)

Process Signals 1 – wait()

Open ▾ Save

```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]){

    pid_t pid;
    int status;

    printf("Process start to fork\n");
    pid=fork();

    if(pid==-1){
        perror("fork");
        exit(1);
    }
    else{

        //Child process
        if(pid==0){
            printf("I'm the Child Process:\n");
            printf("I'm raising SIGCHLD signal!\n\n");
            raise(SIGCHLD);
        }

        //Parent process
        else{
            wait(&status);
            printf("Parent process receives the signal\n");

            if(WIFEXITED(status)){
                printf("Normal termination with EXIT STATUS = %d\n",WEXITSTATUS(status));
            }
            else if(WIFSIGNALED(status)){
                printf("CHILD EXECUTION FAILED: %d\n", WTERMSIG(status));
            }
            else if(WIFSTOPPED(status)){
                printf("CHILD PROCESS STOPPED: %d\n", WSTOPSIG(status));
            }
            else{
                printf("CHILD PROCESS CONTINUED\n");
            }
            exit(0);
        }
    }

    return 0;
}
```

Raise SIGCHLD in child process

Check if child process exits normally

Get status value of child process

Terminal

```
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o Signal Signal.c
[09/03/19]seed@VM:~/.../Tutorial_1$ ./Signal
Process start to fork
I'm the Child Process:
I'm raising SIGCHLD signal!

Parent process receives the signal
Normal termination with EXIT STATUS = 0
[09/03/19]seed@VM:~/.../Tutorial_1$
```

Process Signals 1 – wait()

Open

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

int main(int argc, char *argv[]){

    pid_t pid;
    int status;

    printf("Process start to fork\n");
    pid=fork();

    if(pid==-1){
        perror("fork");
        exit(1);
    }
    else{

        //Child process
        if(pid==0){
            printf("I'm the Child Process:\n");
            printf("I'm raising SIGKILL signal!\n\n");
            raise(SIGKILL);
        }

        //Parent process
        else{
            wait(&status);
            printf("Parent process receives the signal\n");

            if(WIFEXITED(status)){
                printf("Normal termination with EXIT STATUS = %d\n",WEXITSTATUS(status));
            }
            else if(WIFSIGNALED(status)){
                printf("CHILD EXECUTION FAILED: %d\n", WTERMSIG(status));
            }
            else if(WIFSTOPPED(status)){
                printf("CHILD PROCESS STOPPED: %d\n", WSTOPSIG(status));
            }
            else{
                printf("CHILD PROCESS CONTINUED\n");
            }
            exit(0);
        }
    }

    return 0;
```

Raise SIGKILL in child process

Check if child process received a terminating signal

Get status value for child progress' terminating signal

Terminal

```
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o Signal Signal.c
[09/03/19]seed@VM:~/.../Tutorial_1$ ./Signal
Process start to fork
I'm the Child Process:
I'm raising SIGKILL signal!

Parent process receives the signal
CHILD EXECUTION FAILED: 9
[09/03/19]seed@VM:~/.../Tutorial_1$
```

Process Signals 2 – waitpid( )

```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <signal.h>

int main(int argc, char *argv[]){

    pid_t pid;
    int status;

    printf("Process start to fork\n");
    pid=fork();

    if(pid==-1){
        perror("fork");
        exit(1);
    }
    else{

        //Child process
        if(pid==0){
            printf("I'm the Child Process:\n");
            printf("I'm raising SIGSTOP signal!\n\n");
            raise(SIGSTOP);
        }

        //Parent process
        else{
            waitpid(pid, &status, WUNTRACED);
            printf("Parent process receives the signal\n");

            if(WIFEXITED(status)){
                printf("Normal termination with EXIT STATUS = %d\n",WEXITSTATUS(status));
            }
            else if(WIFSIGNALED(status)){
                printf("CHILD EXECUTION FAILED: %d\n", WTERMSIG(status));
            }
            else if(WIFSTOPPED(status)){
                printf("CHILD PROCESS STOPPED: %d\n", WSTOPSIG(status));
            }
            else{
                printf("CHILD PROCESS CONTINUED\n");
            }
            exit(0);
        }
    }

    return 0;
}
```

Raise SIGSTOP in child process

Reports child process' stop signal

Check if child process received a stop signal

Get status value for child progress' terminating signal

```
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o Signal_2 Signal_2.c
[09/03/19]seed@VM:~/.../Tutorial_1$ ./Signal_2
Process start to fork
I'm the Child Process:
I'm raising SIGSTOP signal!

Parent process receives the signal
CHILD PROCESS STOPPED: 19
[09/03/19]seed@VM:~/.../Tutorial_1$
```

To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

(for argc and argc[]

Executing a file

- **exec** is a functionality of an operating system that runs an executable file in the context of an already existing process, replacing the previous executable. This act is also referred to as an overlay.

- Exec function family
  - int **execl** (const char *filename, const char *arg0, ...)
  - int **execve** (const char *filename, char *const argv[], char *const env[])
  - int **execle** (const char *filename, const char *arg0, char *const env[], ...)
  - int **execvp** (const char *filename, char *const argv[])
  - int **execlp** (const char *filename, const char *arg0, ...)

argc (ARGument Count) is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)

argv(ARGument Vector) is array of character pointers listing all the arguments.
If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/

Executing a file

- New process will not be created, the original PID does not change, but the machine code, data, heap and stack of the process are replaced by those of the new program.

- Return value
  - A successful exec replaces the current process image, so it cannot return anything to the program that made the call.
  - If an exec function does return to the calling program, an error occurs, the return value is −1

fork创建了一个新的进程，产生一个新的PID
execve用被执行的程序完全替换了调用进程的映像。
execve启动一个新程序，替换原有进程，所以被执行进程的PID不会改变。

Executing a file – execve()

Open

execve.c

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


int main (int argc, char *argv[]) {

    int state;
    pid_t pid = fork();

    if (pid < 0) {
        printf ("Fork error!\n");
    }
    else {

        //Child process
        if (pid == 0) {

            int i;
            char *arg[argc];

            printf("This is child process.\n");

            for(i=0;i<argc-1;i++){
                arg[i]=argv[i+1];
            }
            arg[argc-1]=NULL;

            printf("Child process id is %d\n", getpid());
            printf("Child process start to execute test program:\n");
            execve(arg[0],arg,NULL);

            printf("Continue to run original child process!\n");

            perror("execve");
            exit(EXIT_FAILURE);
        }

        //Parent process
        else{

            wait(&state);
            printf("This is farther process.\n");
            exit(1);
        }
    }

    return 0;
```

In child process, execute a new file.

Check if the child process is replaced by new program

test.c

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

int main (void) {

    printf("Test process id is %d\n.",getpid());
    printf("\tTest completed!\n");
    printf("\tExit child process!\n");
    return 0;
}
```

Check if the test program is replacing child process but will not create new process

Terminal

```
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o execve execve.c
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o test test.c
[09/03/19]seed@VM:~/.../Tutorial_1$ ./execve ./test
This is child process.
Child process id is 3119
Child process start to execute test program:
Test process id is 3119
.       Test completed!
        Exit child process!
This is farther process.
[09/03/19]seed@VM:~/.../Tutorial_1$
```

path of the executed file

References

- Fork system call
  - http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html
  - https://en.wikipedia.org/wiki/Fork_(system_call)

- waitpid()
  - https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.bpxbd00/rtwaip.htm

- Zombie process
  - https://en.wikipedia.org/wiki/Zombie_process

References

- Linux standard signals:
  - https://en.wikipedia.org/wiki/Signal_(IPC)

- Exec function family:
  - https://en.wikipedia.org/wiki/Exec_(system_call)

Thank you