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

Main steps of setting up environment of virtual machine with Linux Ubuntu 16.4.

- Install virtual machine (Virtual Box is free for download)
- Download Linux OS and resize it (Ubuntu 16.04)
- Install Linux OS in virtual machine
- Disk partition
- Some useful setting (Shared folders, shared clipboard)

Environment setup

- For environment setup, follow the guide 'Install Ubuntu in VirtualBox.pdf'
- In our tutorial, we will use the prebuilt VM image for Ubuntu Linux (version 16.04).
- The kernel version is v4.8.0-36-generic. (Command to check kernel version: uname -r)
- User accounts for this prebuilt VM
  - User ID: root
    Password: seedubuntu
    (Note: Ubunt does not allow root to login directly from the login window. You have to login as a normal user, and then use command su to login to the root account)
  - User ID: seed
    Password: dees
    (This account is already given the root privilege, but to use the privilege, you should use the sudo command)

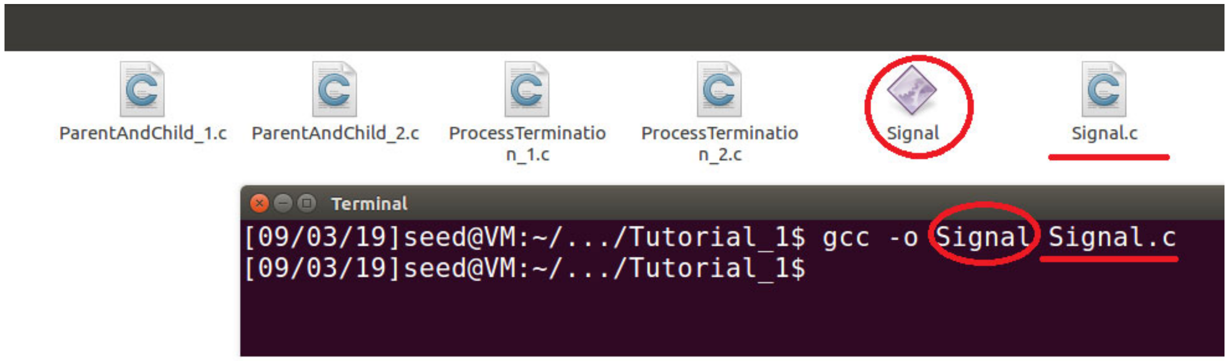
Environment setup

- 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.



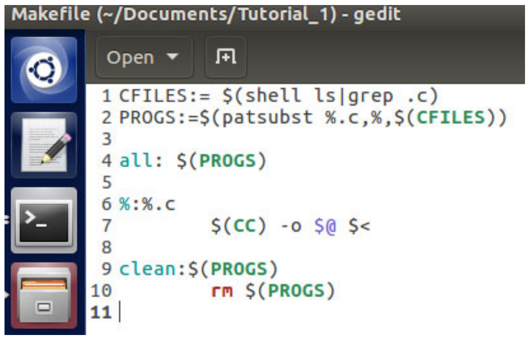
```
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o Signal Signal.c
[09/03/19]seed@VM:~/.../Tutorial_1$
```

Environment setup

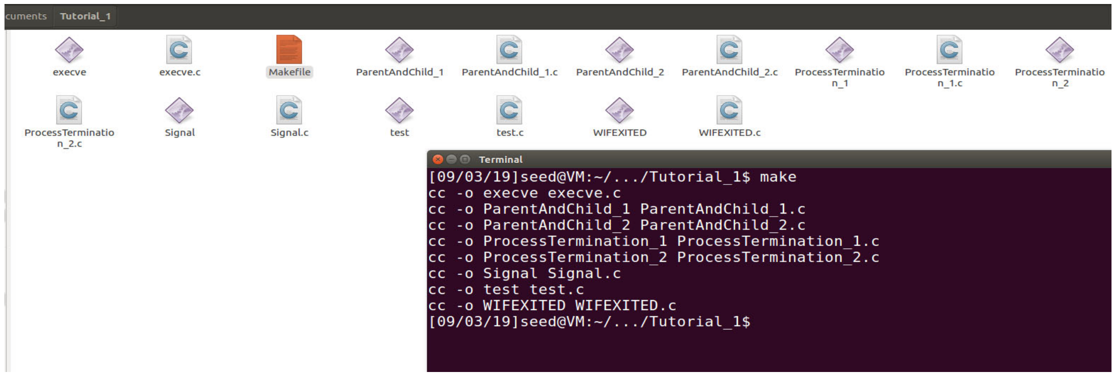
- Makefile: defining some compile commands into make command

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

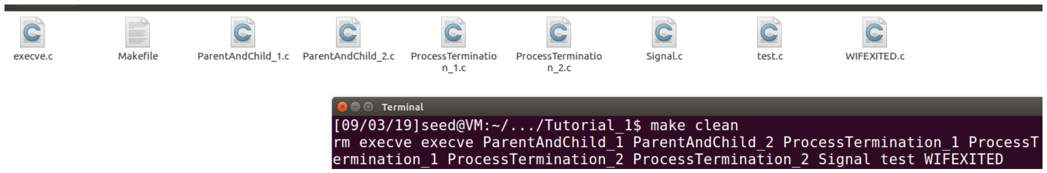
  Command: make clean (for all executable files which are ending with '.c', remove them)



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

all: $(PROGS)

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

clean:$(PROGS)
        rm $(PROGS)

```



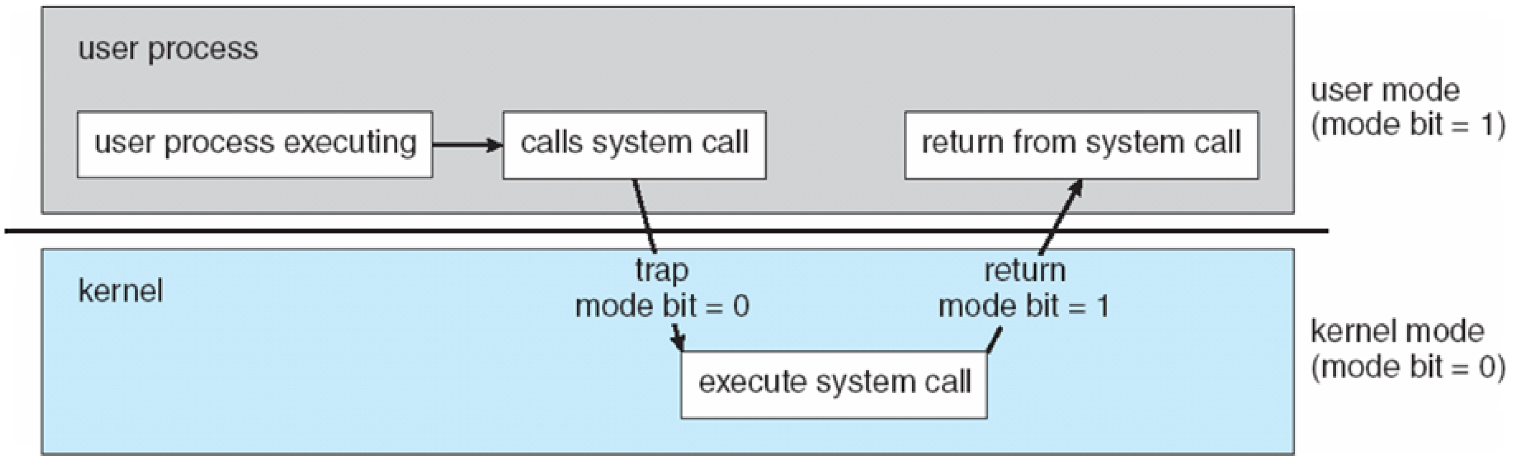
```
[09/03/19]seed@VM:~/.../Tutorial_1$ make
cc -o execve execve.c
cc -o ParentAndChild_1 ParentAndChild_1.c
cc -o ParentAndChild_2 ParentAndChild_2.c
cc -o ProcessTermination_1 ProcessTermination_1.c
cc -o ProcessTermination_2 ProcessTermination_2.c
cc -o Signal Signal.c
cc -o test test.c
cc -o WIFEXITED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



```
[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



Process

- Process: Program in execution

- 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

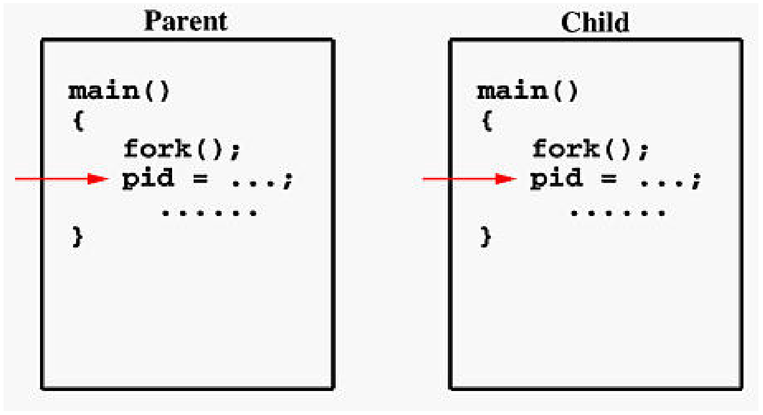
- Functions (Unix-like system):
  - **pid_t fork(void)** system call creates new process

- 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.

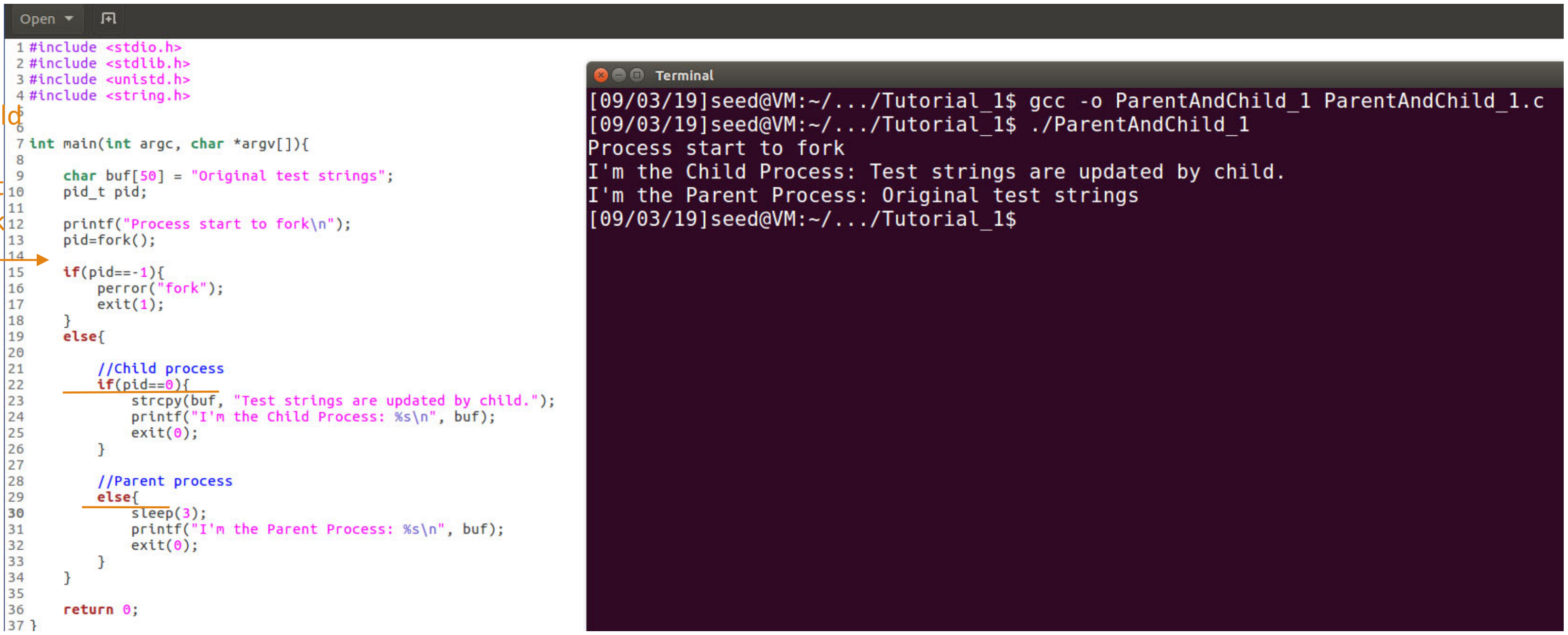


```
main()
{
    fork();
    pid = ...;
      ......
}
```

```
main()
{
    fork();
    pid = ...;
      ......
}
```

Parent and Child Process

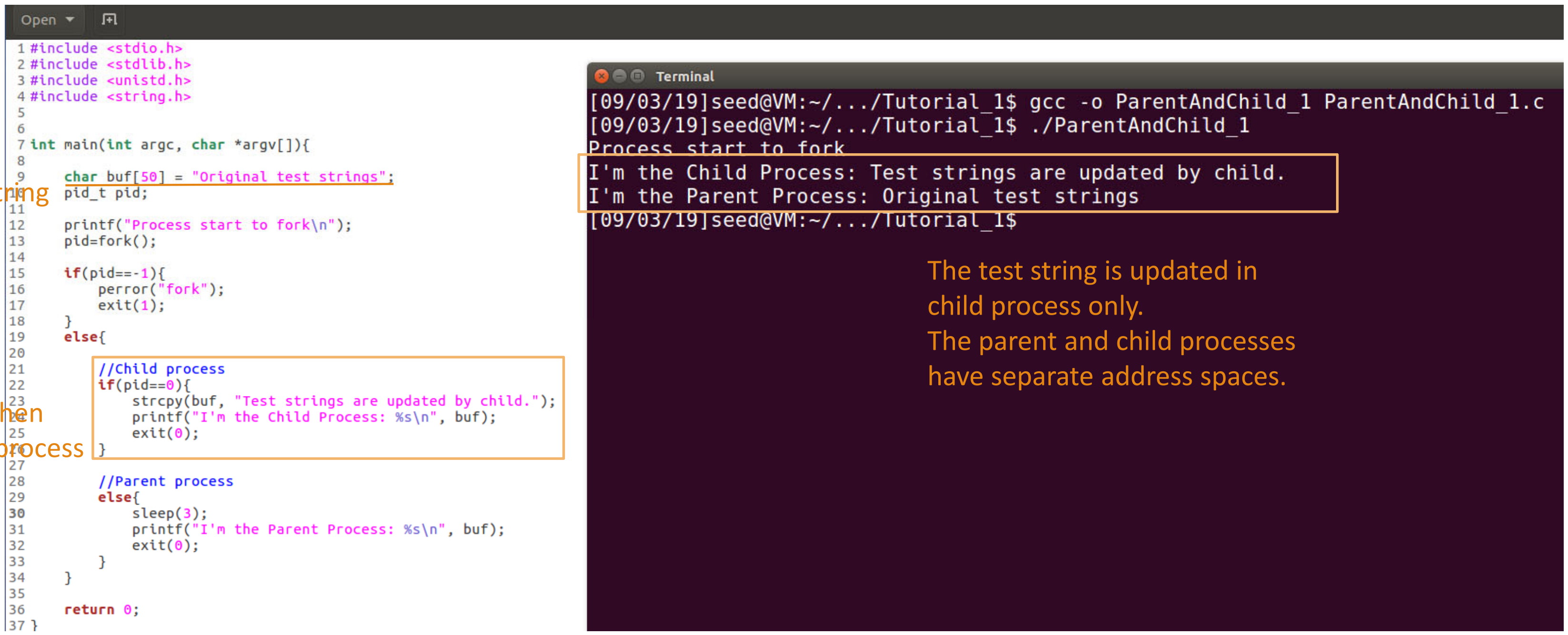
Both parent and child process start execution from next statement after fork call.



```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;
}
```

```
[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$
```

Parent and 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;
}
```

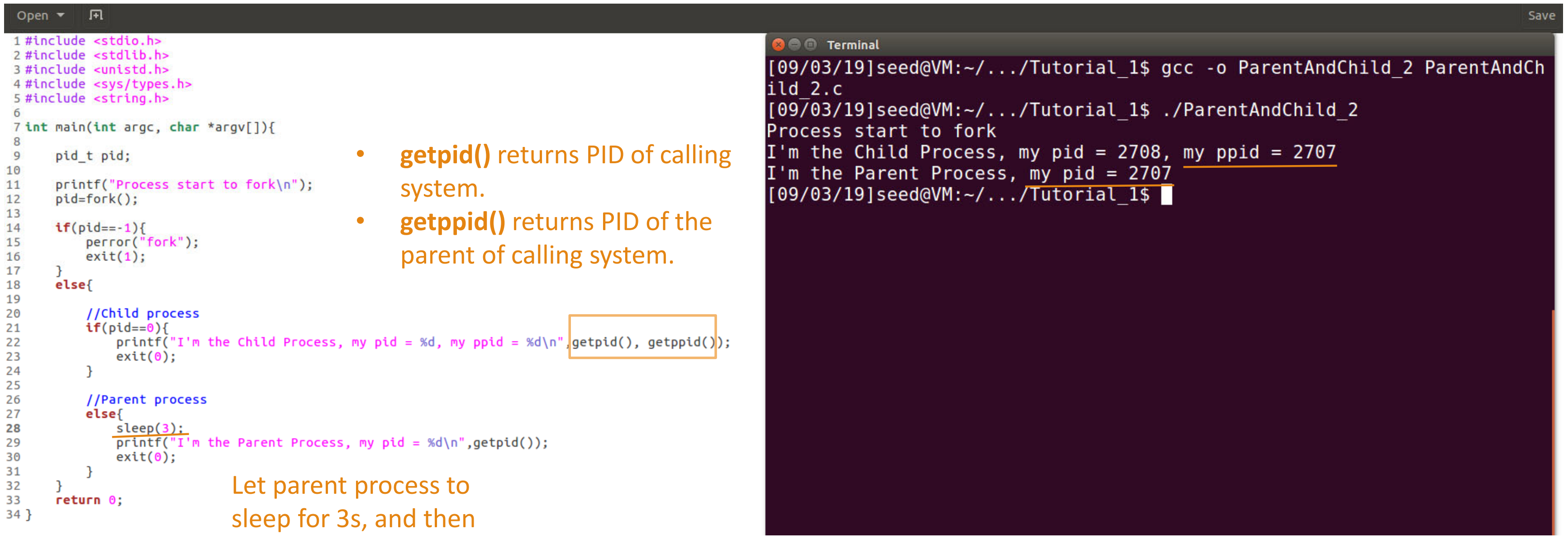
Set original test string

Update string when executing child process

```
[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



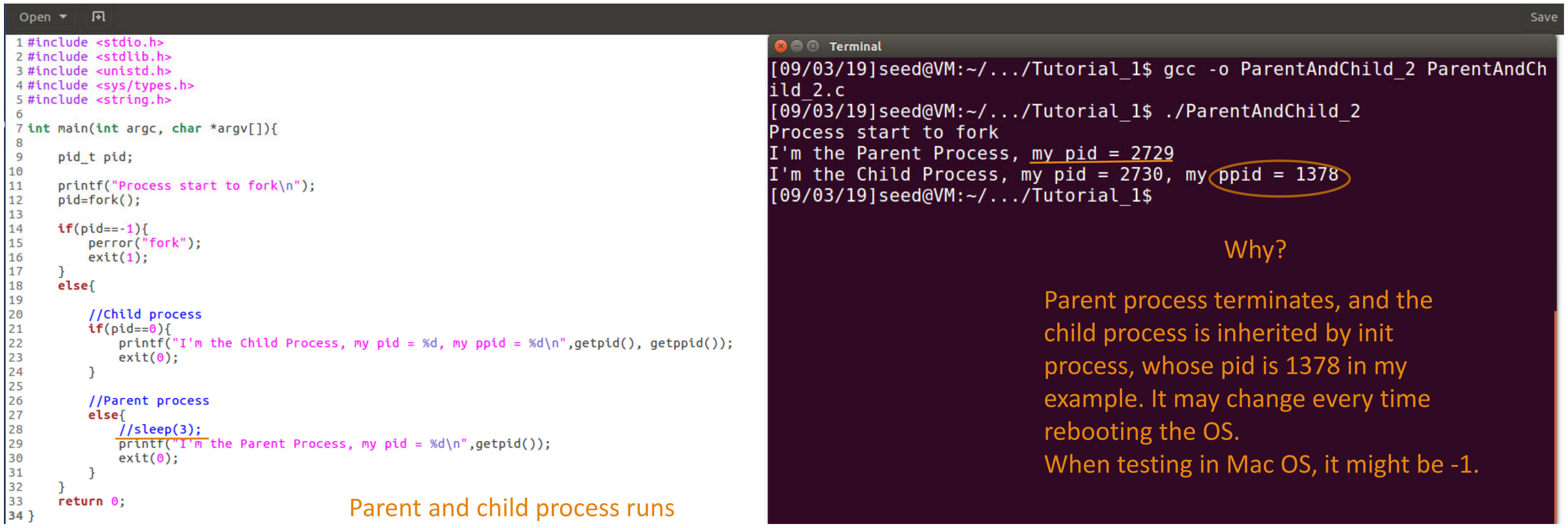
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

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;
}
```

```
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o ParentAndChild_2 ParentAndCh
ild_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$
```

- **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.

Parent and Child Process



```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;
}
```

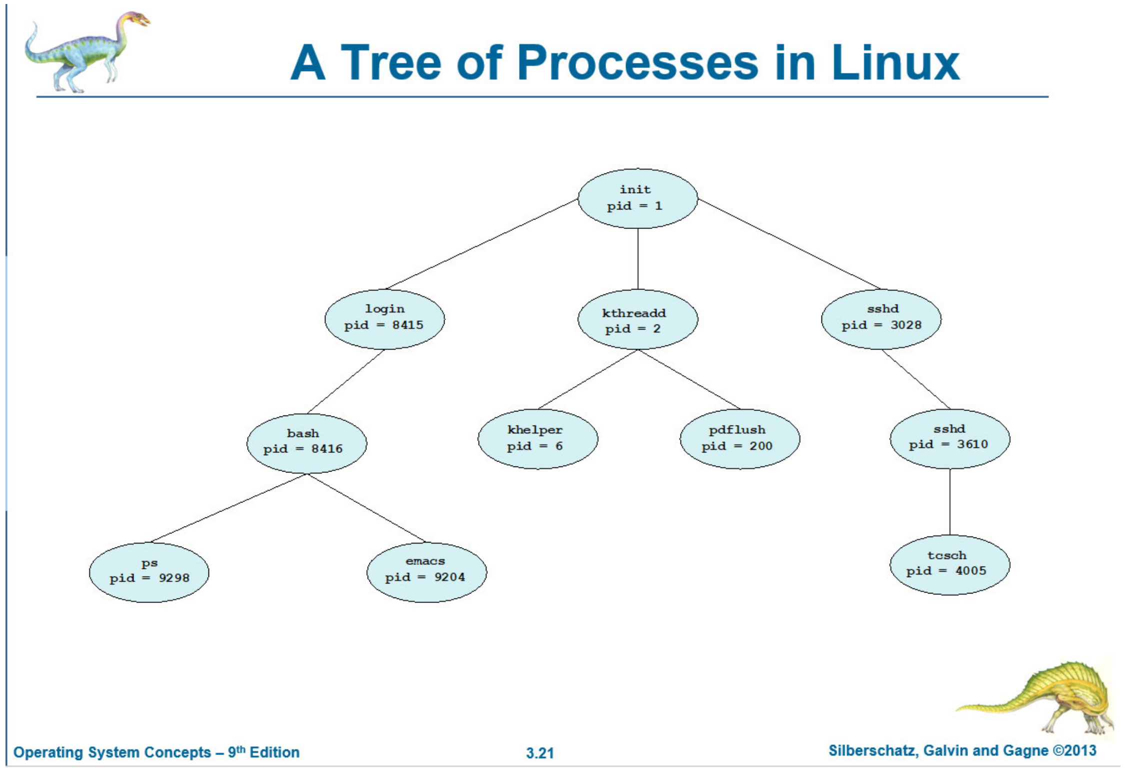
```
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o ParentAndChild_2 ParentAndCh
ild_2.c
[09/03/19]seed@VM:~/.../Tutorial_1$ ./ParentAndChild_2
Process start to fork
I'm the Parent Process, my pid = 2729
I'm the Child Process, my pid = 2730, my ppid = 1378
[09/03/19]seed@VM:~/.../Tutorial_1$
```

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 runs concurrently after forking.

Parent and Child Process



A Tree of Processes in Linux

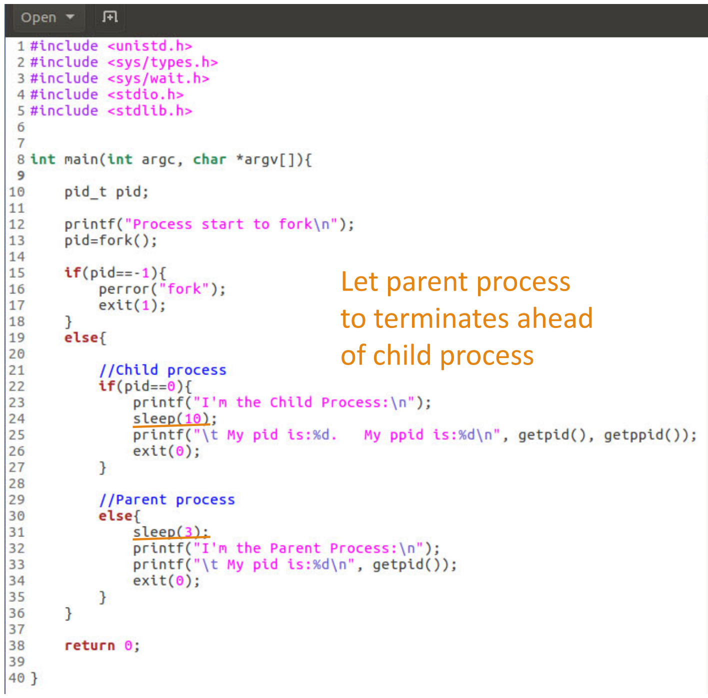
Operating System Concepts – 9th Edition 3.21 Silberschatz, Galvin and Gagne ©2013

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

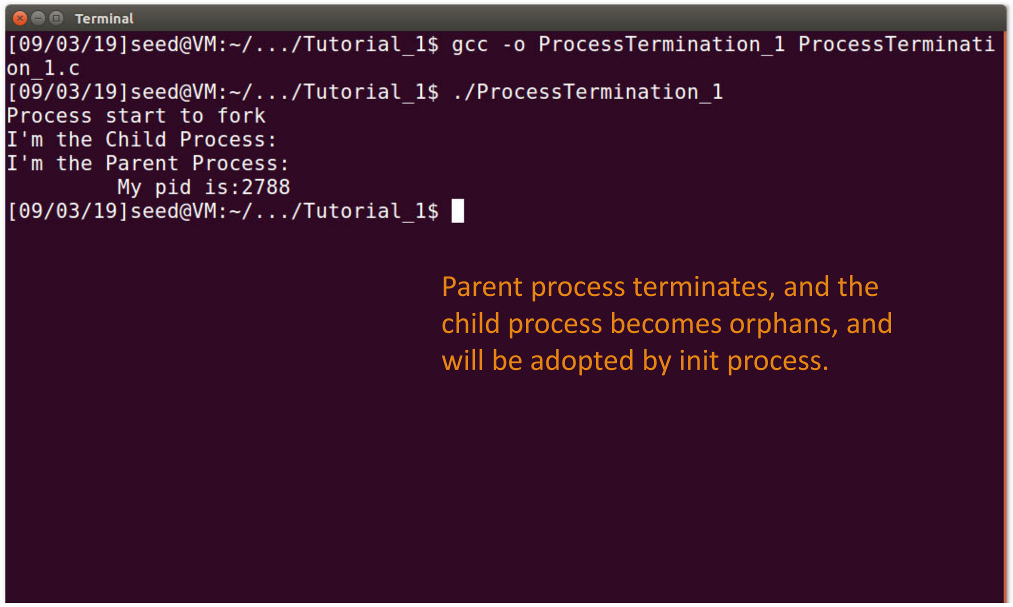
- If no parent waiting, then terminated process is a **zombie**
- If parent terminated, processes are **orphans**

Process Termination



```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.

Process Termination

- 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.

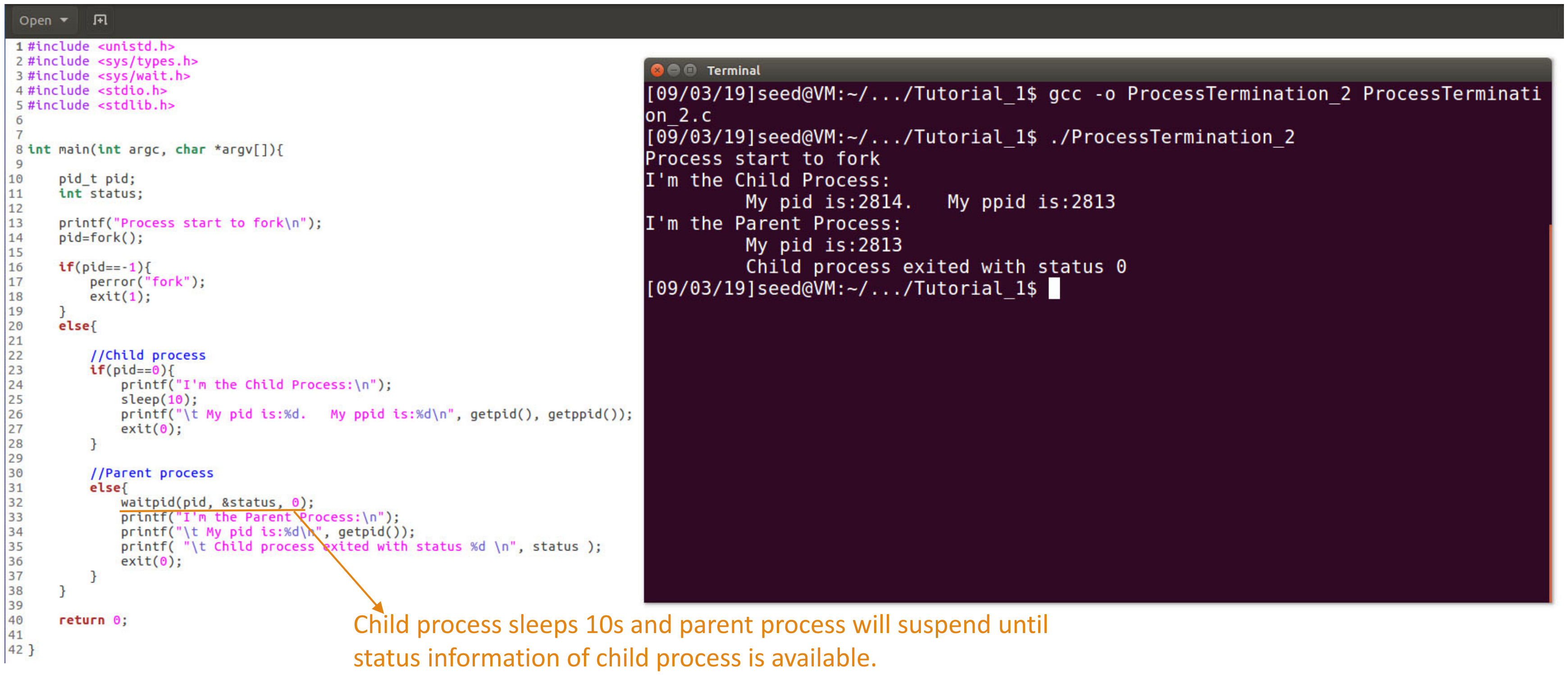
- Differences
  - wait() requests status for any child process
  - waitpid() requests status for specific child process.
  - The waitpid() function behaves the same as wait() if the pid argument is (pid_t) -1 and the options argument is 0.

Process Termination

- 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.
  - If WNOHANG was specified on the options parameter, but no child process was immediately available, waitpid() returns 0.

- 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



```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;
}
```

```
[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.	

http://man7.org/linux/man-pages/man7/signal.7.html

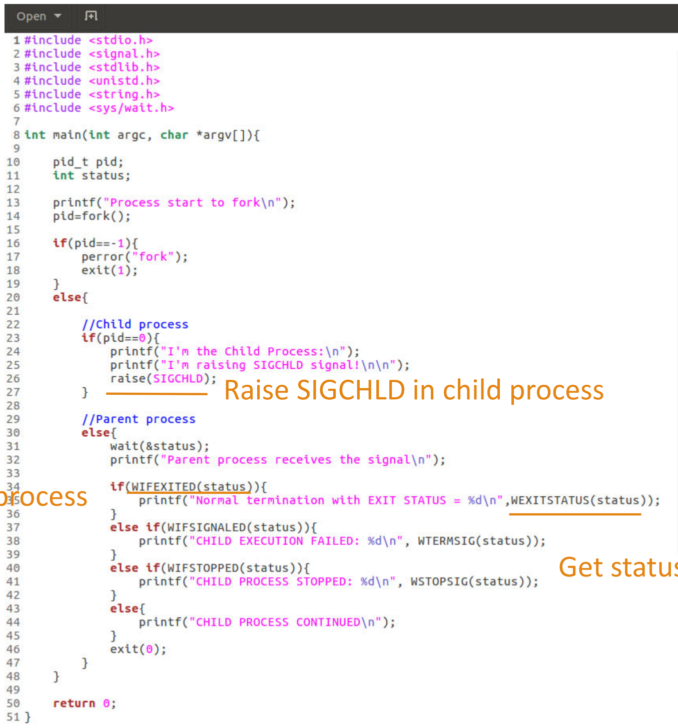
Process Signals

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

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

- 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

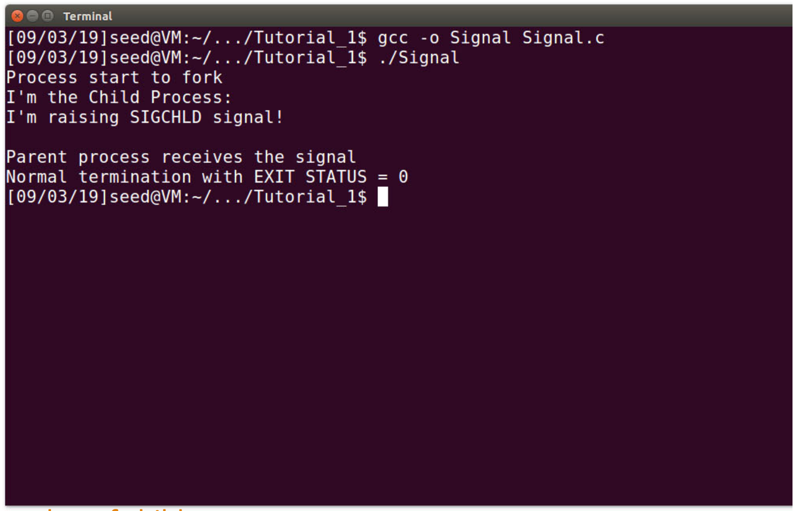


```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

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

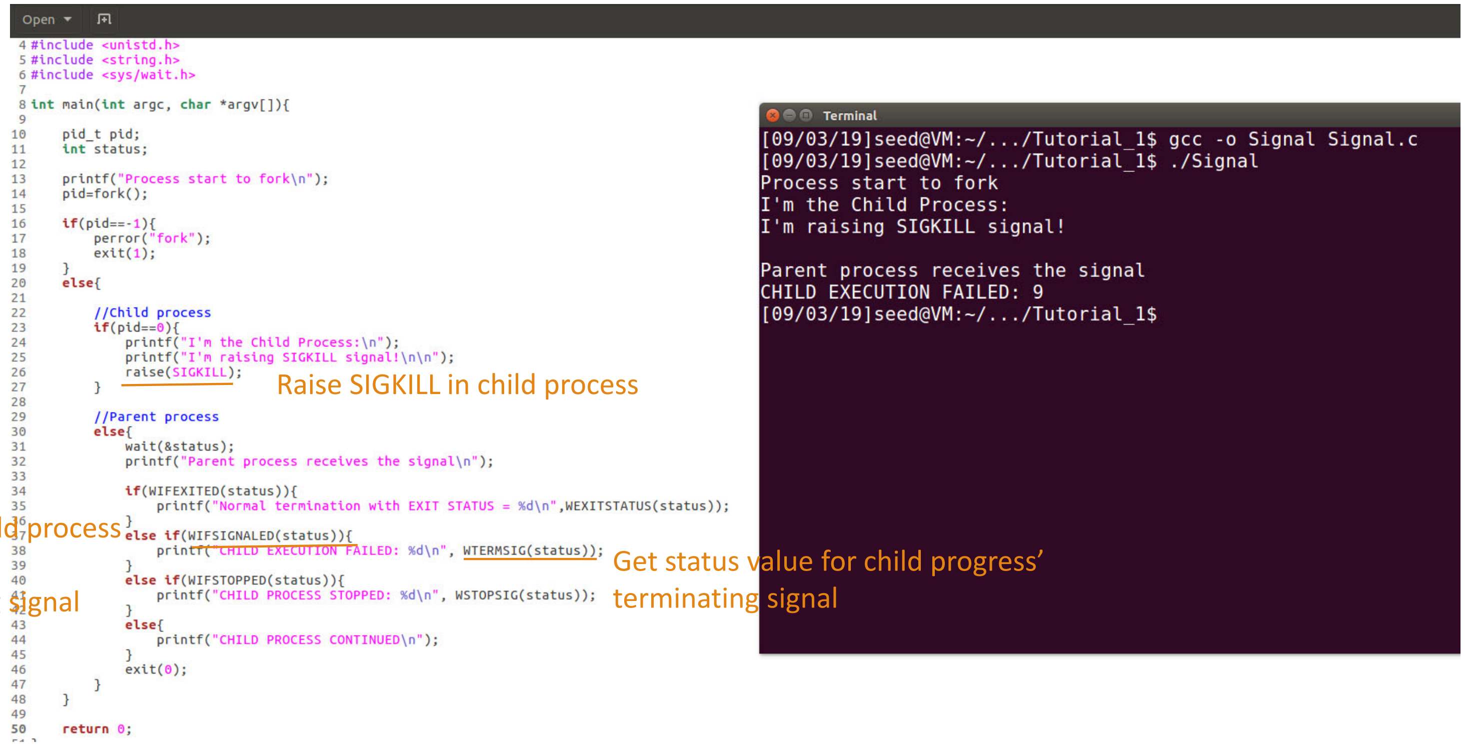
Check if child process exits normally

Get status value of child process



```
[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



```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;
```

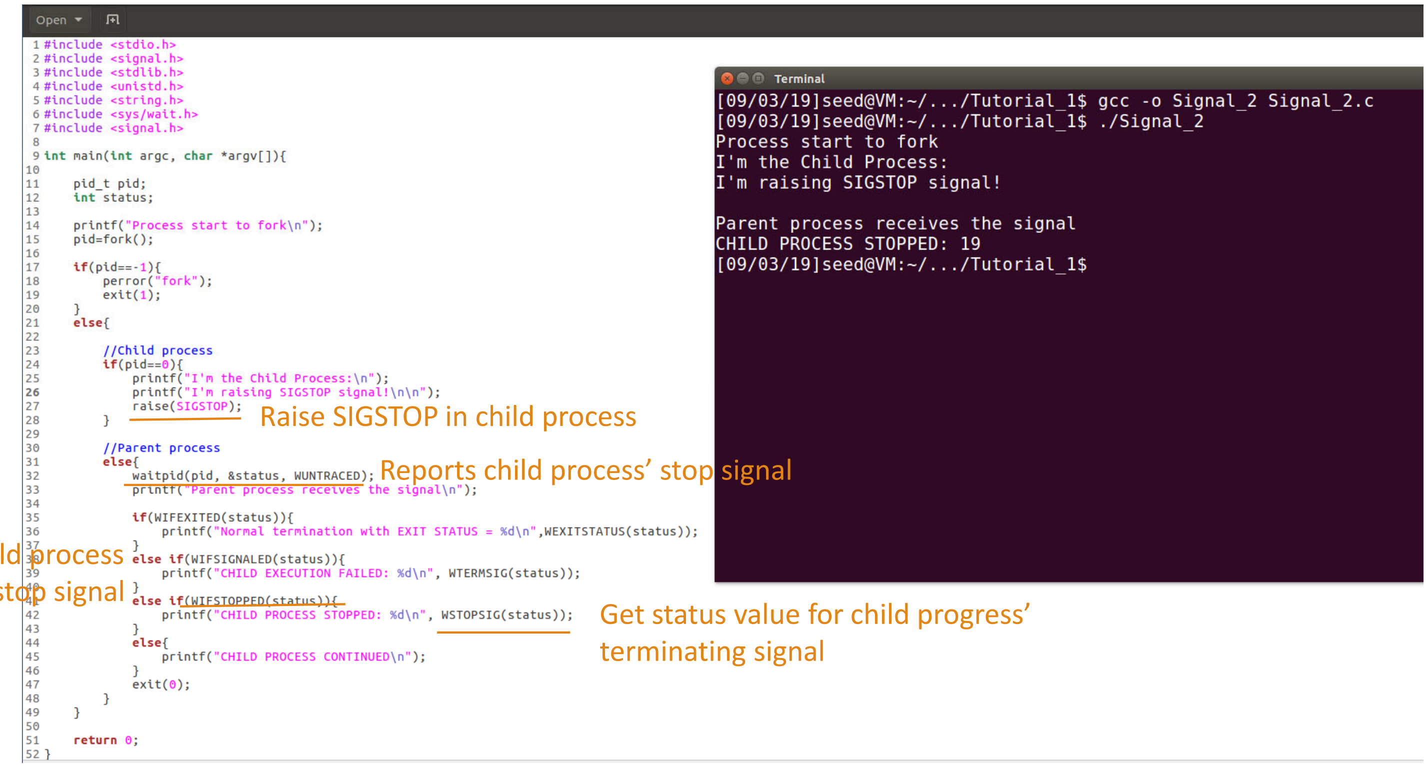
```
[09/03/19]seed@VM:~/.../Tutorial_1$ gcc -o Signal Signal.c
[09/03/19]seed@VM:~/.../Tutorial_1$ ./Signal
Process start to fork
I'm the Child Process:
I'm raising SIGKILL signal!

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

Raise SIGKILL in child process

Check if child process received a terminating signal

Get status value for child progress' terminating signal

Process Signals



```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;
}
```

```
[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$
```

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

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.

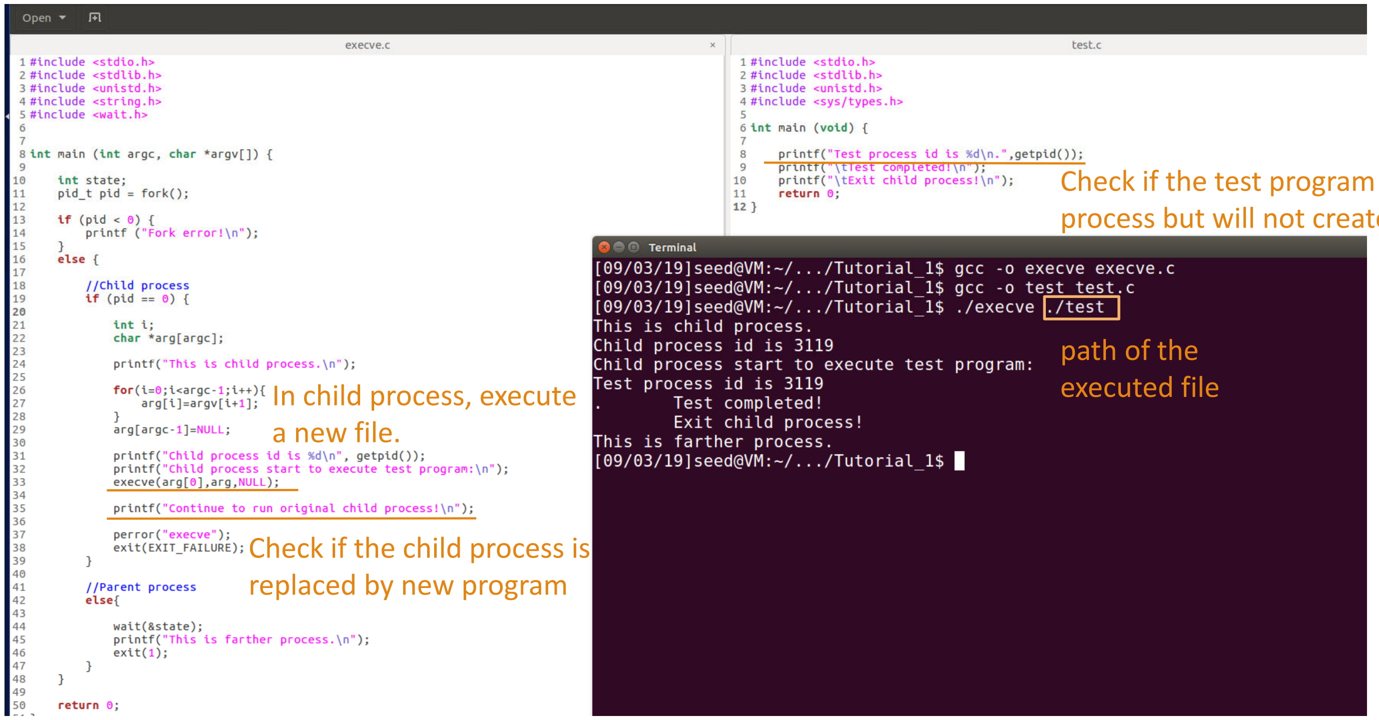
- Exec function family
  - int **execl** (const char *filename, const char *arg0, ...)
  - int **execve** (const char *filename, char *const argv[], char *const env[])
  - int **execle** (const char *filename, const char *arg0, char *const env[], ...)
  - int **execvp** (const char *filename, char *const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

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

Executing a file



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

```
[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