Report 2

> What is a system call:
> 什么是系统调用：

A system call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.

A system call is a way for programs to interact with the operating system.

> What is fork:
> 简述fork调用:

Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process).

After a new child process is created, both processes will execute the instruction following the fork() system call. A child process uses the same pc (program counter), same CPU registers, same open files which use in the parent process.

> How to realize inter-process communication:
> 如何实现进程间的通信:

1. Pipe (Same Process): This allows flow of data in one direction only.
2. Names Pipes (Different Processes): This is a pipe with a specific name it can be used in processes that don't have a shared common process origin.
3. Message Queuing: This allows messages to be passed between processes using either a single queue or several message queue. This is managed by system kernel these messages are coordinated using an API.
4. Signal
5. Semaphores: This is used in solving problems associated with synchronization and to avoid race condition.
6. Shared memory: This allows the interchange of data through a defined area of memory. Semaphore values have to be obtained before data can get access to shared memory.
7. Sockets: This method is mostly used to communicate over a network between a client and a server. It allows for a standard connection which is computer and OS independent.

> How to realize inter-process connection:
> 如何实现进程间的连接:

For inter-process connection, Pipe is a communication medium between two or more related or interrelated processes.

Pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe. It opens a pipe, which is an area of main memory that is treated as a "virtual file".

The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this "virtual file" or pipe and another related process can read from it.

If a process tries to read before something is written to the pipe, the process is suspended until something is written.

> Write the prototype of function "fork": 写出函数"fork"的原型:

```
pid_t fork(void);
```

> Write the prototype of function "signal":
> 写出函数"signal"的原型:

```
sighandler_t signal(int signum, sighandler_t handler);
```

> Write the prototype of function "pipe":
> 写出函数"pipe"的原型:

```
int pipe(int pipefd[2]);
```

> Write the prototype of function "tcsetpgrp":
> 写出函数"tcsetpgrp"的原型:

```
int tcsetpgrp(int fd, pid_t pgrp);
```

> Execute "fork.c" and observe, please describe the result (not execution result):
> 运行" fork.c ", 观察结果并简述结果（不是执行结果）:

```
gcc fork.c -o fork
```

```
total 84
drwxr-xr-x   2 root root  4096 Jan 20 16:28 bin
drwxr-xr-x   3 root root  4096 Jan 20 16:36 boot
drwxr-xr-x  18 root root  3740 Jan 21 19:30 dev
drwxr-xr-x  91 root root  4096 Mar  4 12:59 etc
drwxr-xr-x   2 root root  4096 Apr 13  2016 home
lrwxrwxrwx   1 root root    33 Jan 20 16:32 initrd.img -> boot/initrd.img-
4.15.0-74-generic
lrwxrwxrwx   1 root root    33 Jan 20 16:32 initrd.img.old ->
boot/initrd.img-4.4.0-170-generic
drwxr-xr-x  21 root root  4096 Jan 20 16:32 lib
drwxr-xr-x   2 root root  4096 Jan 20 16:32 lib64
drwx------   2 root root 16384 Dec 25 14:22 lost+found
drwxr-xr-x   4 root root  4096 Dec 25 14:23 media
drwxr-xr-x   2 root root  4096 Feb 27  2019 mnt
drwxr-xr-x   2 root root  4096 Feb 27  2019 opt
dr-xr-xr-x 109 root root     0 Jan 21 19:30 proc
drwx------  13 root root  4096 Mar 10 20:06 root
drwxr-xr-x  23 root root   760 Mar 10 20:07 run
drwxr-xr-x   2 root root 12288 Jan 21 19:43 sbin
drwxr-xr-x   2 root root  4096 Feb 27  2019 srv
dr-xr-xr-x  13 root root     0 Mar 10 20:14 sys
drwxrwxrwt  14 root root  4096 Mar 10 20:14 tmp
drwxr-xr-x  10 root root  4096 Dec 25 14:23 usr
drwxr-xr-x  12 root root  4096 Jan 20 17:09 var
lrwxrwxrwx   1 root root    30 Jan 20 16:32 vmlinuz -> boot/vmlinuz-
```

```
4.15.0-74-generic
lrwxrwxrwx   1 root root    30 Jan 20 16:32 vmlinuz.old -> boot/vmlinuz-
4.4.0-170-generic
-rw-r--r--   1 root root     0 Feb  4 02:59 wget-log
-rw-r--r--   1 root root     0 Feb  5 16:51 wget-log.1
-rw-r--r--   1 root root     0 Feb 21 14:09 wget-log.2
-rw-r--r--   1 root root     0 Feb 21 14:13 wget-log.3
-rw-r--r--   1 root root     0 Feb 22 18:44 wget-log.4
-rw-r--r--   1 root root     0 Mar  1 12:15 wget-log.5
-rw-r--r--   1 root root     0 Mar  1 23:24 wget-log.6
-rw-r--r--   1 root root     0 Mar  3 14:52 wget-log.7
-rw-r--r--   1 root root     0 Mar 10 20:07 wget-log.8
```

`errno`：记录系统的最后一次错误代码，代码是一个int型的值 `perror(s)`：用来将上一个函数发生错误的原因输出到标准设备(stderr) `execvp`: `int execvp(const char* file, const char* argv[]);`

Executing the function `fork()`, If an error occurs, the program will print an error message.

Otherwise, a child process is created. Then the child executes the command `ls -l /` and display the output. After executing the command and printing the output, the child process terminated and sent a signal to the parent process. The parent process wakes up(stops the waiting state) by `waitpid(pid,NULL,0)`.

But `printf()`'s output didn't show on the screen. `printf()` does not output directly, it saves data in buffer. After the child process terminated(returnd), the buffer didn't output.(Add `\n` or `fflush(stdout)`, can output `printf()`)

The parent process end the program and terminated(returned).

> Execute "fork.c" and observe, please describe how to distinguish between parent and child processes in a program：
> 运行" fork.c ", 观察结果，并简述程序中如何区分父进程和子进程：

For the parent process, the return value of `fork()` is the process id(PID) of the child process. So the parent process will execute the `if(pid)` block.

For the child process, the return value of `fork()` is 0. So the child process will execute the `if(!pid)` block.

> Execute "pipe.c" and observe, please describe the result (not execution result)：
> 运行" pipe.c", 观察结果并简述结果（不是执行结果）：

```
------------------------|send--------------------


------------------------start1|send--------------------


----------------------start1start2|rec--------------------------
```

```
total 808
-rw-r--r-- 1 root root    3028 Feb 27  2019 adduser.conf
-rw-r--r-- 1 root root      18 Dec 25 14:29 adjtime
drwxr-xr-x 2 root root    4096 Feb  4 02:46 alternatives
drwxr-xr-x 3 root root    4096 Dec 25 14:24 apm
drwxr-xr-x 3 root root    4096 Jan 20 16:31 apparmor
drwxr-xr-x 8 root root    4096 Feb  4 02:46 apparmor.d
drwxr-xr-x 7 root root    4096 Mar  4 13:00 apt
-rw-r----- 1 root daemon   144 Jan 15  2016 at.deny
-rw-r--r-- 1 root root    2319 Jun  7  2019 bash.bashrc
-rw-r--r-- 1 root root      45 Aug 13  2015 bash_completion
drwxr-xr-x 2 root root    4096 Jan 20 16:51 bash_completion.d
-rw-r--r-- 1 root root     367 Jan 27  2016 bindresvport.blacklist
drwxr-xr-x 2 root root    4096 Apr 12  2016 binfmt.d
drwxr-xr-x 3 root root    4096 Dec 25 14:24 ca-certificates
-rw-r--r-- 1 root root    6864 Jan 20 16:31 ca-certificates.conf
--More--
```

The parent process creates a pipe with two ends and two child processes. If an error occurs, the program will print an error message.

First, the child 1 process waits for receiving the data from the read end of pipe. The parent process write data "start1" to the write end of the pipe.

After the child 1 process receiving the data, it copys the write end descriptor to the standard output.

Then, the parent process write data "start2" to the write end of the pipe.

The child 1 process executes `ls -l /etc/` and sends the result to the standard output, which is the write end of the pipe now.

THe child 2 process copys the read end descriptor to the standard input, which is the read end of the pipe now. Then it executes `more`.

> Execute "pipe.c" and observe. Is execvp(prog2_argv[0],prog2_argv)(Line 56) executed? And why? :
> 运行" pipe.c", 观察结果。execvp(prog2_argv[0],prog2_argv)（第56行）是否执行，如果没有执行是什么原因：

Yes, it executed.

Before `more` executed, the standard input is the read end of the pipe. The content which send to the standard output can be received from the end of pipe. So that the command can be executed.

> Execute "signal.c" and observe, please describe the result (not execution result)：
> 运行" signal.c", 观察结果并简述结果（不是执行结果）

The program prints the parent process id and its child process id continuously.

When we send a kill signal to kill the child process, the fucntion `ChildHandler` responses the PID of the child process and prints "The child is gone!!!!!"

When we send a kill signal to kill the parent process, the child process is still alive.

> Execute "signal.c" and observe. Please answer, how to execute function ChildHandler? :
> 运行" signal.c", 观察结果。请回答，怎样让函数ChildHandler执行?

Because the function `ChildHandler` is registered for the SIGCHLD, we can send signal by `kill -9 <Child PID>` to kill the child process to let `ChildHandler` exexute.

> Execute "process.c" and observe, please describe the result (not execution result):
> 运行" process.c", 观察结果并简述结果（不是执行结果）：

Modified the `process.c`:

- `execvp` -> `execlp`
- Add `signal(SIGTTOU,SIG_IGN);`, the parent process ignores `SIG_IGN`.
- `/bin/vi` -> `/usr/bin/vi`

对于父进程忽略`SIG_UGN`，它将子进程独立成一个进程组，并让子进程设为前台进程组。然后等待子进程结束后，是自己所在进程组成为前台进程组。然后进行`ECHO`回显操作。

对于子进程，它用`setpgid(0,0);`将自己的PID作为进程组的ID，也就是从原来父进程独立出来。不过执行`tcsetpgrp(0,getpid());`这一句是没有权限的，非前台进程组是不能主动变成前台进程组，需要当前前台进程赋予。当它被父进程设为前台进程组后，它执行`vi`。

> Execute "process.c" and observe. Please answer, how many ./process in the process list? And what's the difference between them?:
> 运行" process.c", 观察结果。请回答，进程列表中有几个./process，区别在哪里：

当子进程在`vi`里面时，只有一个`./process`，使用`ps -aux | grep ./process`，这时候在前台的进程组是子进程，父进程显示为`./process`，如果使用 `ps -a`， 子进程显示为`vi`

当子进程退出，父进程执行ECHO回显时，只有一个`./process`，这时候在前台的进程组是父进程

> Execute "process.c" and observe. Please answer, what happens after killing the main process:
> 运行" process.c", 观察结果。请回答，杀死主进程后，出现什么情况：

在运行`./process`后，如果在`vi`时杀死父进程（此时前台进程是子进程），shell会退出`vi`，父进程和子进程都会退出，并显示：

Vim: Error reading input, exiting... Vim: Finished.

在运行`./process`后，如果在`vi`时杀死子进程（此时前台进程是子进程），shell会退出`vi`，子进程退出，并显示：

Vim: Vim Caught deadly signal TERM Vim: Finished.

在运行`./process`后，`:q`退出子进程，此时前台进程为父进程，在父进程ECHO阶段杀死父进程（此时子进程已经`exit()`），显示：

Terminated