Q1.

What is a system call:

什么是系统调用：

所选答案：system cal 即系统调用，是一种内核层级的“函数调用”，是进程从用户态进入内核态的唯一方式，在一定程度上提高了操作系统的安全性。

Q2.

What is fork:

简述 fork 调用:

所选答案：新建一个用户态信息和父进程完全一致的子进程，但子进程的 PID、运行时间、父进程以及 fork 调用的返回值等内核态数据和父进程不同。

Q3.

How to realize inter-process communication:

如何实现进程间的通信:

所选答案：通过无名管道 pipe、有名管道 named pipe（FIFO）、消息队列 message queue、信号 signal、信号量 ssemaphore、共享内存 shared memory、套接字 socket 等方式来实现进程之间的通信。

Q4.

How to realize inter-process connection:

如何实现进程间的连接:

所选答案：主要通过有名管道 named pipe(FIFO)来实现进程间的连接。

有名管道 named pipe(FIFO)连接与无名管道 pipe 的区别体现在前者可以实现在没有关系的多进程间实现通信，且写进程输入的数据要被读出（即成功建立连接）之后这个进程才能解除堵塞执行下一步的操作，建立的 FIFO 有名管道不会随进程的终止而消失，其会以文件的形式继续存在于 linux 系统中等待下一次的使用。

Q5.

Write the prototype of function "fork":

写出函数"fork"的原型:

所选答案：

```
#include<unistd.h>
#include<sys/types.h>
pid_t fork(void);
```

Q6.

Write the prototype of function "signal":

写出函数"signal"的原型:

所选答案：
```
#include<signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Q7.

Write the prototype of function "pipe":

写出函数"pipe"的原型:

所选答案：
```
#include<unistd.h>
struct fd_pair{
    long fd[2];
};
struct fd_pair pipe();
int pipe(int pipefd[2]);
```

Q8.

Write the prototype of function "tcsetpgrp":

写出函数"tcsetpgrp"的原型:

所选答案：
```
#include<unistd.h>
int tcsetpgrp(int fd, pid_t pgrp);
```

Q9.

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（描述执行结果，和出现该结果的原因）：

所选答案：执行结果：等价于命令“/bin/ls -l /”的结果

原因：代码最初以 prog_argv 数组来保存四个参数，之后执行 fork 命令，创建与当前进程代码相同的子进程。在子进程段获取 prog_argv 数组的参数并调用 execvp 执行“ /bin/ls -l / ”，在父进程段调用 waitpid 方法使得父进程在子进程后执行。

Q10.

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：

所选答案： 程序以 fork()的返回值来区分父子进程，执行不同的分支。
fork()在子进程中的返回值为 0， 在父进程中的返回值为子进程的 pid。
若 fork 失败，则 fork()的返回值小于 0.

Q11.

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（ 描述执行结果，和出现该结果的原因 ）：

所选答案： 执行结果：等价于将命令“/bin/ls -l /etc/”的输出作为“/bin/more”的输入（通过 pipe 实现的进程间通信），将结果显示在命令行上。
原因： 0、1、2 默认状态下在 linux 系统中对应着标准输入、标准输出以及标准错误输出，利用 dup2(oldfd, newfd)可以实现输入输出的重定向, 即通过管道将原本的标准输出作为下一次的标准输入。
具体执行过程为：定义 pipe 相关参数，用数组 prog1_argv 以及 prog2_argv 保存命令 “/bin/ls -l /etc/” 以及 “/bin/more” 的参数。第一次 fork，先执行子进程，调用 execvp()执行 “/bin/ls -l /etc/” ，输出写入管道 ，等待子进程结束后（使用 waitpid 命令）父进程进行第二次 fork，让第二次 fork 的子进程调用 execvp 执行 “/bin/more”，从管道获取输入。

Q12.

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）是否一定会执行，为什么

所选答案： 在实验观察中， execvp(prog2_argv[0],prog2_argv) 会执行，因为第一次 fork 之后的父进程第二次 fork 也成功了，在第二次的子进程中执行了本条命令。
但若 fork 失败，则此条命令不会执行。

Q13.

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（ 描述执行结果，和出现该结果的原因 ）：

所选答案： 执行结果：循环输出父子进程的 pid（父：6647 子：6648），在另一终端执行 “kill -9 6648” 后，输出：
The process generating the signal is PID: 6648
The child is gone!!!!!

然后循环输出 PID(parent): 6647
原因：程序注册信号处理函数，然后通过 fork 创建父子进程，分别循环输出本进程的 pid。执行 “kill -9 6648” 后，sigaction()接受到信号 SIGCHLD 并执行自定义的信号处理函数 ChildHandler。

Q14.

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？

所选答案： 从原 terminal 的输出获得子进程的 pid（即 6648），新开一个 terminal，输入“kill -9 6648” 杀死子进程则可执行 ChildHandle 函数

Q15.

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（ 描述执行结果，和出现该结果的原因 ）：

所选答案： 执行结果（修改后）：子进程执行 vi 操作并正常退出
原因：父进程将终端控制权交给子进程，子进程组成为前台进程组，子进程从而成功进入 vi 指令，子进程退出后控制权交回给父进程

Q16.

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：

所选答案： 有两个，即 fork 产生的父子进程, 区别在于它们的 pid 不同

Q17.

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，出现什么情况并分析原因：

所选答案： 结果：程序终止并发生回显，即将输入显示出来
原因：父进程组重新变成了前台进程组