作业3

3.1 fork、exec、wait等是进程操作的常用API，请调研了解这些API的使用方法。
（1）请写一个C程序，该程序首先创建一个1到10的整数数组，然后创建一个子进程，并让子进程对前述数组所有元素求和，并打印求和结果。等子进程完成求和后，父进程打印“parent process finishes”,再退出。
（2）在（1）所写的程序基础上，当子进程完成数组求和后，让其执行ls -l命令(注：该命令用于显示某个目录下文件和子目录的详细信息)，显示你运行程序所用操作系统的某个目录详情。例如，让子进程执行 ls -l /usr/bin目录，显示/usr/bin目录下的详情。父进程仍然需要等待子进程执行完后打印“parent process finishes”,再退出。
（3）请阅读XV6代码（https://pdos.csail.mit.edu/6.828/2021/xv6.html），找出XV6代码中对进程控制块（PCB）的定义代码，说明其所在的文件，以及当fork执行时，对PCB做了哪些操作？

提交内容
（1） 所写C程序，打印结果截图，说明等
（2） 所写C程序，打印结果截图，说明等
（3） 代码分析介绍
解答部分：
（1）

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

int main(){
    static int array[10]={0};
    int sum=0;
    int status=0;
    int i=0;
    for(i=0;i<10;array[i++]=i){
    }
    int pid=fork();
    if(pid==0){
        i=0;
        while(i<10)
            sum+=array[i++];
        printf("I'm the son process,and the sum=%d\n",sum);
        exit(1);
    }
    else
```

```
    {   wait(&status);
        printf("parent process finishes\n");}
    return 0;
}
```

代码说明：

程序 1 中，第 14 行调用了 fork 函数，fork 会拷贝当前进程的内存，并创建一个新的进程，这里的内存包含了进程的指令和数据。然后就有了两个拥有完全一样内存的进程。fork 系统调用在两个进程中都会返回，在原始的进程中，fork 系统调用会返回大于 0 的整数，这个是新创建进程的 ID。而在新创建的进程中，fork 系统调用会返回 0。所以可以利用返回值来判断是否为子进程，然后子进程执行求和的操作。在父进程中，需要等待子进程执行完毕，这需要利用 wait 函数实现，wait 函数等待子进程结束，同时接受一个子进程退出状态的值。所以，整个程序的运行结果就是子进程求和结束后，父进程打印出输出。

运行结果：

```
solomon@DESKTOP-23PER74:~/CODES/C$
solomon@DESKTOP-23PER74:~/CODES/C$
solomon@DESKTOP-23PER74:~/CODES/C$ gcc -o homework3 homework3.c
solomon@DESKTOP-23PER74:~/CODES/C$ ./homework3
I'm the son process,and the sum=55
parent process finishes
```

（2）

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

int main(){
    static int array[10]={0};
    int sum=0;
    int status=0;
    int i=0;
    for(i=0;i<10;array[i++]=i){
    }
    int pid=fork();
    if(pid==0){
        i=0;
        while(i<10)
            sum+=array[i++];
        printf("I'm the son process,and the sum=%d\n",sum);
        execl("/bin/ls", "ls", "-l", "/home/solomon/CODES/C", NULL);
        exit(1);
```

```
    }
    else
    {   wait(&status);
        printf("parent process finishes\n");}
    return 0;
}
```

代码分析：

仅需增加一条代码，即可在子进程中执行ls程序，这个功能是使用execl函数来实现的，execl() 函数用于启动一个新的程序，新程序的命令行参数以可变参数列表的形式传递。

函数签名：

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ...);
```

参数：

path：新程序的路径；

arg0：新程序的名称；

可变参数列表：新程序的命令行参数，以NULL结束。

返回值：若正常返回，则不返回，否则返回-1。

这个系统调用会从指定的文件中读取并加载指令，并替代当前调用进程的指令。相当于丢弃了调用进程的内存，并开始执行新加载的指令。操作系统从名为ls的文件中加载指令到当前的进程中，并替换了当前进程的内存，之后开始执行这些新加载的指令。同时，你可以传入命令行参数，各个字符串分别代表执行的参数，利用NULL值代表参数结束。执行代码可以看到如下效果，若将参数改为"/home/solomon/CODES/C"，即可看到子进程先求和后再执行ls，最后父进程打印信息。

fork首先拷贝了整个父进程的代码、数据、栈堆等，子进程可以实现父进程完全一模一样的功能，但是如果调用exec函数族内的函数，则会将这整个拷贝丢弃了，并用要运行的文件替换内存的内容。所有拷贝的内存都被丢弃并被exec替换。wait系统调用只能等待当前进程的子进程。如果当前进程有任何子进程，那么wait会返回。可以利用copy-on-write fork，实现对fork调用的优化，这种方式会消除fork的几乎所有的明显的低效，而只拷贝执行exec所需要的内存。

```
solomon@DESKTOP-23PER74:~/CODES/C$ gcc -o homework3 homework3.c
solomon@DESKTOP-23PER74:~/CODES/C$ ./homework3
I'm the son process,and the sum=55
total 48
-rwxr-xr-x 1 solomon solomon 17000 Sep 24 09:50 homework3
-rw-r--r-- 1 solomon solomon   572 Sep 24 09:32 homework3.c
-rwxr-xr-x 1 solomon solomon 19808 Sep 18 21:19 test
-rw-r--r-- 1 solomon solomon  1007 Sep 18 21:23 test.c
parent process finishes
```

同样，如果是/usr/bin,则可以看到如下效果。

C > C homework3.c

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

int main(){
    static int array[10]={0};
    int sum=0;
    int status=0;
```

问题 输出 调试控制台 终端 端口

```
-rwxr-xr-x 1 root   root       14648 Feb 29  2020  xkill
-rwxr-xr-x 1 root   root       14648 Feb 29  2020  xlsatoms
-rwxr-xr-x 1 root   root       18752 Feb 29  2020  xlsclients
-rwxr-xr-x 1 root   root       27032 Feb 29  2020  xlsfonts
-rwxr-xr-x 1 root   root       27840 Feb 29  2020  xmessage
-rwxr-xr-x 1 root   root       49768 Feb 29  2020  xprop
-rwxr-xr-x 1 root   root        5164 Oct 19  2020  xsubpp
-rwxr-xr-x 1 root   root       18744 Feb 29  2020  xvinfo
-rwxr-xr-x 1 root   root       51592 Feb 29  2020  xwininfo
-rwxr-xr-x 1 root   root       18712 Feb  1  2022  xxd
-rwxr-xr-x 1 root   root       80384 Apr 21  2020  xz
lrwxrwxrwx 1 root   root           2 Apr 21  2020  xzcat -> xz
lrwxrwxrwx 1 root   root           6 Apr 21  2020  xzcmp -> xzdiff
-rwxr-xr-x 1 root   root        6632 Apr 21  2020  xzdiff
lrwxrwxrwx 1 root   root           6 Apr 21  2020  xzegrep -> xzgrep
lrwxrwxrwx 1 root   root           6 Apr 21  2020  xzfgrep -> xzgrep
-rwxr-xr-x 1 root   root        5628 Apr 21  2020  xzgrep
-rwxr-xr-x 1 root   root        1802 Apr 21  2020  xzless
-rwxr-xr-x 1 root   root        2161 Apr 21  2020  xzmore
-rwxr-xr-x 1 root   root       39256 Sep  5  2019  yes
lrwxrwxrwx 1 root   root           8 Nov  7  2019  ypdomainname -> hostname
-rwxr-xr-x 1 root   root        1984 Dec 13  2019  zcat
-rwxr-xr-x 1 root   root        1678 Dec 13  2019  zcmp
-rwxr-xr-x 1 root   root        5880 Dec 13  2019  zdiff
-rwxr-xr-x 1 root   root       26840 Dec 16  2020  zdump
-rwxr-xr-x 1 root   root          29 Dec 13  2019  zegrep
-rwxr-xr-x 1 root   root          29 Dec 13  2019  zfgrep
-rwxr-xr-x 1 root   root        2081 Dec 13  2019  zforce
-rwxr-xr-x 1 root   root        7585 Dec 13  2019  zgrep
-rwxr-xr-x 1 root   root       50718 Oct 19  2020  zipdetails
-rwxr-xr-x 1 root   root        2206 Dec 13  2019  zless
-rwxr-xr-x 1 root   root        1842 Dec 13  2019  zmore
-rwxr-xr-x 1 root   root        4553 Dec 13  2019  znew
parent process finishes
solomon@DESKTOP-23PER74:~/CODES/C$
```

(3)理论分析：

XV6关于进程的控制部分从3000开始，具体PCB控制模块定义在proc.h头文件中 ， 该头文件位于第350行，具体定义如下：

```
Sep  1 09:28 1988  unix/proc.h Page 1


0350 /*
0351  * One structure allocated per active
0352  * process. It contains all data needed
0353  * about the process while the
0354  * process may be swapped out.
0355  * Other per process data (user.h)
0356  * is swapped with the process.
0357  */
0358 struct     proc
0359 {
0360  char      p_stat;
0361  char      p_flag;
0362  char      p_pri;  /* priority, negative is high */
0363  char      p_sig;  /* signal number sent to this process */
0364  char      p_uid;  /* user id, used to direct tty signals */
0365  char      p_time; /* resident time for scheduling */
0366  char      p_cpu;  /* cpu usage for scheduling */
0367  char      p_nice; /* nice for scheduling */
0368  int       p_ttyp; /* controlling tty */
0369  int       p_pid;  /* unique process id */
0370  int       p_ppid; /* process id of parent */
0371  int       p_addr; /* address of swappable image */
0372  int       p_size; /* size of swappable image (*64 bytes) */
0373  int       p_wchan;/* event process is awaiting */
0374  int       *p_textp;/* pointer to text structure */
0375
0376 } proc[NPROC];
0377 /* ------------------------       */
```

Fork函数定义在第3320行，具体执行操作如下：

```
3322 fork()
3323 {
3324    register struct proc *p1, *p2;
3325
3326    p1 = u.u_procp;
3327    for(p2 = &proc[0]; p2 < &proc[NPROC]; p2++)
3328            if(p2->p_stat == NULL)
3329                    goto found;
3330    u.u_error = EAGAIN;
3331    goto out;
3332
3333 found:
3334    if(newproc()) {
3335            u.u_ar0[R0] = p1->p_pid;
3336            u.u_cstime[0] = 0;
3337            u.u_cstime[1] = 0;
3338            u.u_stime = 0;
3339            u.u_cutime[0] = 0;
3340            u.u_cutime[1] = 0;
3341            u.u_utime = 0;
3342            return;
3343    }
3344    u.u_ar0[R0] = p2->p_pid;
3345
3346 out:
3347    u.u_ar0[R7] =+ 2;
3348 }
3349 /* ------------------------       */
```

根据代码，可以看到fork的执行过程，这段代码的作用是创建一个新的进程，或者在当前进程中执行一个fork调用。当调用fork时，它会检查是否有可用的空闲进程，如果有，则将当前进程的PID赋值给新的进程，并返回0。如果没有可用的空闲进程，它会返回一个错误EAGAIN。

在创建新进程时，这段代码会执行以下操作：

- ◆将当前进程的PID赋值给新的进程。
- ◆初始化新进程的上下文，例如时间戳、寄存器等。
- ◆返回0，表示新进程已创建。

◆最后退出函数并将R7寄存器中的值+2.

Ps:R7是Linux系统中的一个寄存器，用于存储当前进程的PID。将PID加2后，可以将其转换为相应的线性地址，是通过将PID与进程表的起始地址相加得到的。进程表通常位于内存中的一个固定位置，因此将PID加2后可以得到一个接近于进程表起始地址的线性地址，从而访问到进程数据结构。（这一部分来自gpt）

3.2 请阅读以下程序代码，回答下列问题

（1）该程序一共会生成几个子进程？请你画出生成的进程之间的关系（即谁是父进程谁是子进程），并对进程关系进行适当说明。

（2）如果生成的子进程数量和宏定义LOOP不符，在不改变for循环的前提下，你能用少量代码修改，使该程序生成LOOP个子进程么？

提交内容

（1） 问题解答，关系图和说明等

（2） 修改后的代码，结果截图，对代码的说明等

```
#include<unistd.h>
#include<stdio.h>
#include<string.h>
#define LOOP 2

int main(int argc,char *argv[])
{
   pid_t pid;
   int loop;

   for(loop=0;loop<LOOP;loop++) {

      if((pid=fork()) < 0)
         fprintf(stderr, "fork failed\n");
      else if(pid == 0) {
```

```
            printf(" I am child process\n");
        }
        else {
            sleep(5);
        }
    }
    return 0;
}
```

（1）、

最初只有一个父进程 P 存在。在第一次循环中，父进程 P 调用 fork()创建一个子进程 P1,此时父子进程的代码是一样的,它们会继续执行循环。在第一次循环中，父进程 P 会进入 sleep(5)，而子进程 P1 会输出"I am child process"。

在第二次循环中，父进程 P 再次调用 fork()创建第二个子进程 P2，此时父子进程的代码是一样的，它们会继续执行循环，同上，P 会进入 sleep(5),但 P2 会输出“I am child process”。

与此同时，P1 会进入它的第一次循环，但实际上是 loop 为 1，父进程 P1 会再次创建一个进程 P3，但后 P1 进入 sleep(5)，而 P3 则会输出"I am child process"，loop 变为 2，循环结束后，总共生成了 LOOP（此处为 2）+1 个子进程，每个子进程输出"I am child process"，而父进程在每次循环中都会进入 sleep(5)。表现的效果为，首先打印了两条“I am child process”，分别为 P1 P3 打印的结果，5s 后 P2 被创建，P2 打印出“I am child process”,P1 和 P 进入 sleep(5).

结果如图：

```
solomon@DESKTOP-23PER74:~/CODES/C$ ./homework3_2
I am child process
I am child process
I am child process
```

（2）、

```c
#include<unistd.h>
#include<stdio.h>
#include<string.h>
#define LOOP 2

int main(int argc,char *argv[])
{
   pid_t pid;
   int loop;

   for(loop=0;loop<LOOP;loop++) {

      if((pid=fork()) < 0)
         fprintf(stderr, "fork failed\n");
```

```
      else if(pid == 0) {
         printf(" I am child process\n");
         break;
      }
      else {
         sleep(5);
      }
    }
    return 0;
}
```

仅需在子进程中加入一个 break 语句，即可让子进程不再进入循环，即不再产生新的子进程，所以实际生成的进程数即为 LOOP 的数量。

结果如下：

每句“I am child process”间隔为 5s.

```
solomon@DESKTOP-23PER74:~/CODES/C$ ./homework3_2
 I am child process
 I am child process
```

如果在 sleep(5)上面，输出 pid 的值，可以得到三个子进程的 pid 值，其中前三个分别为 P1 和 P3,最后一个为最后创建的 P2 的进程号。