作业2

2.1 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

答：
（1）代码如下：

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

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

    int array[] = {1,2,3,4,5,6,7,8,9,10};
    int rc = fork();

    //son
    if (rc == 0) {
        int sum=0;
        for(int i=0;i<10;i++){
            sum+=array[i];
        }
        printf("Result of array sum is %d\n", sum);
    }
    //parent
    else {
        wait(NULL);
        printf("parent process finishes\n");
    }

    return 0;
}
```

打印结果截图如下：

```
ubuntu@ubuntu:~/Desktop$ gcc -g hw.c -o hw
ubuntu@ubuntu:~/Desktop$ ./hw
Result of array sum is 55
parent process finishes
```

说明：先fork出子进程，然后子进程中（对应rc==0）执行数组求和和打印操作，在父进程中先wait到子进程结束，再打印“parent process finishes”,最后退出。可以看到打印结果正常。

（2）代码如下：

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

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

    int array[] = {1,2,3,4,5,6,7,8,9,10};
    int rc = fork();

    //son
    if (rc == 0) {
        int sum=0;
        for(int i=0;i<10;i++){
            sum+=array[i];
        }
        char *myargs[] = {"ls", "-l", "./", 0};
        execvp(myargs[0], myargs);
        printf("Result of array sum is %d\n", sum);
    }
    //parent
    else {
        wait(NULL);
        printf("parent process finishes\n");
    }

    return 0;
}
```

打印结果截图如下：

```
ubuntu@ubuntu:~/Desktop$ ./hw
total 160
-rwxrwxr-x 1 ubuntu ubuntu 12664 Jun 26 01:50 a.out
-rwxrwxr-x 1 ubuntu ubuntu    46 Jul 10 02:16 asm_sh
-rw-rw-r-- 1 ubuntu ubuntu 23993 Jul 10 02:33 base.c
-rw-rw-r-- 1 ubuntu ubuntu     0 Jul 10 23:36 base.o
-rw-rw-r-- 1 ubuntu ubuntu  3784 Jul 10 23:34 base.S
-rwxrwxr-x 1 ubuntu ubuntu 15936 Mar  7  2022 GPA
-rw-rw-r-- 1 ubuntu ubuntu  2708 Mar  7  2022 GPA.c
-rwxrwxr-x 1 ubuntu ubuntu  1104 May 21 21:04 hello
-rw-rw-r-- 1 ubuntu ubuntu   960 May 21 21:04 hello.o
-rw-rw-r-- 1 ubuntu ubuntu   249 May 21 21:04 hello.s
-rwxrwxr-x 1 ubuntu ubuntu 11456 Sep 11 07:40 hw
-rwxrwxr-x 1 ubuntu ubuntu 12200 Sep  4 13:13 hw1
-rw-rw-r-- 1 ubuntu ubuntu  1130 Sep  4 13:12 hw1.c
-rw-rw-r-- 1 ubuntu ubuntu   550 Sep 11 07:40 hw.c
-rw-rw-r-- 1 ubuntu ubuntu  2052 May  1 02:13 hw.o
-rw-rw-r-- 1 ubuntu ubuntu   198 Jun 25 22:11 hw.s
-rwxrwxr-x 1 ubuntu ubuntu 12624 Jun 26 02:26 -O2
-rwxrwxr-x 1 ubuntu ubuntu  1260 May 22 02:21 print_hex
-rw-rw-r-- 1 ubuntu ubuntu  1112 May 22 02:21 print_hex.o
-rw-rw-r-- 1 ubuntu ubuntu  1139 May 22 02:21 print_hex.s
-rw-rw-r-- 1 ubuntu ubuntu   322 Sep  5 11:02 scores.txt
lrwxrwxrwx 1 root   root      36 Sep  8  2021 sublime -> /usr/local/sublime_text/sublime_text
-rwxrwxr-x 1 ubuntu ubuntu     0 Jul 10 23:36 test
-rw-rw-r-- 1 ubuntu ubuntu  1180 Jul 10 02:31 test.c
-rw-rw-r-- 1 ubuntu ubuntu  3120 Jun 26 01:46 test.o
parent process finishes
```

说明：先fork出子进程，然后子进程中（对应rc==0）执行数组求和操作，并利用execvp函数执行ls -l命令，列出当前文件夹（此处是桌面文件夹）中文件和子目录的详细信息。在父进程中同样地先wait到子进程结束，再打印“parent process finishes”,最后退出。可以看到打印结果正常。

值得注意的是，execvp函数替换了当前进程的内存，开始执行新加载的指令，所以打印结果的指令被覆盖了，最终在终端上没有输出数组求和结果。

为了验证我们确实做了数组求和这一操作，调换17、18行位置，输出结果如下：

```
ubuntu@ubuntu:~/Desktop$ ./hw
Result of array sum is 55
total 160
-rwxrwxr-x 1 ubuntu ubuntu 12664 Jun 26 01:50 a.out
-rwxrwxr-x 1 ubuntu ubuntu    46 Jul 10 02:16 asm_sh
-rw-rw-r-- 1 ubuntu ubuntu 23993 Jul 10 02:33 base.c
-rw-rw-r-- 1 ubuntu ubuntu     0 Jul 10 23:36 base.o
-rw-rw-r-- 1 ubuntu ubuntu  3784 Jul 10 23:34 base.S
-rwxrwxr-x 1 ubuntu ubuntu 15936 Mar  7  2022 GPA
-rw-rw-r-- 1 ubuntu ubuntu  2708 Mar  7  2022 GPA.c
-rwxrwxr-x 1 ubuntu ubuntu  1104 May 21 21:04 hello
-rw-rw-r-- 1 ubuntu ubuntu   960 May 21 21:04 hello.o
-rw-rw-r-- 1 ubuntu ubuntu   249 May 21 21:04 hello.s
-rwxrwxr-x 1 ubuntu ubuntu 11456 Sep 11 07:42 hw
-rwxrwxr-x 1 ubuntu ubuntu 12200 Sep  4 13:13 hw1
-rw-rw-r-- 1 ubuntu ubuntu  1130 Sep  4 13:12 hw1.c
-rw-rw-r-- 1 ubuntu ubuntu   559 Sep 11 07:42 hw.c
-rw-rw-r-- 1 ubuntu ubuntu  2052 May  1 02:13 hw.o
-rw-rw-r-- 1 ubuntu ubuntu   198 Jun 25 22:11 hw.s
-rwxrwxr-x 1 ubuntu ubuntu 12624 Jun 26 02:26 -O2
-rwxrwxr-x 1 ubuntu ubuntu  1260 May 22 02:21 print_hex
-rw-rw-r-- 1 ubuntu ubuntu  1112 May 22 02:21 print_hex.o
-rw-rw-r-- 1 ubuntu ubuntu  1139 May 22 02:21 print_hex.s
-rw-rw-r-- 1 ubuntu ubuntu   322 Sep  5 11:02 scores.txt
lrwxrwxrwx 1 root   root      36 Sep  8  2021 sublime -> /usr/local/sublime_text/sublime_text
-rwxrwxr-x 1 ubuntu ubuntu     0 Jul 10 23:36 test
-rw-rw-r-- 1 ubuntu ubuntu  1180 Jul 10 02:31 test.c
-rw-rw-r-- 1 ubuntu ubuntu  3120 Jun 26 01:46 test.o
parent process finishes
```

这进一步说明，execvp函数是替换顺序执行的剩余部分，而非整体替换。该函数执行前的指令仍然正常执行。

（3）代码分析介绍：

```
84  // Per-process state
85  struct proc {
86    struct spinlock lock;
87
88    // p->lock must be held when using these:
89    enum procstate state;        // Process state
90    void *chan;                  // If non-zero, sleeping on chan
91    int killed;                  // If non-zero, have been killed
92    int xstate;                  // Exit status to be returned to parent's wait
93    int pid;                     // Process ID
94
95    // wait_lock must be held when using this:
96    struct proc *parent;         // Parent process
97
98    // these are private to the process, so p->lock need not be held.
99    uint64 kstack;               // Virtual address of kernel stack
100   uint64 sz;                   // Size of process memory (bytes)
101   pagetable_t pagetable;       // User page table
102   struct trapframe *trapframe; // data page for trampoline.S
103   struct context context;      // swtch() here to run process
104   struct file *ofile[NOFILE];  // Open files
105   struct inode *cwd;           // Current directory
106   char name[16];               // Process name (debugging)
107 };
```

该结构体是进程控制块（PCB）的定义代码，位于文件proc.h中。

fork() 函数对 PCB 做了如下操作：

1. 从当前进程的页表中复制内容到新的页表，并将新的页表赋值给 pagetable；
2. 设置子进程PCB的 sz 与父进程PCB相同；
3. 设置子进程PCB的 parent 为父进程PCB；
4. 将父进程PCB的 trapframe 拷贝到子进程PCB，将 trapframe 中的 eax置0；
5. 设置子进程PCB的 ofile和name 与父进程PCB相同；
6. 设置子进程PCB的state 为 RUNNABLE。

2.2 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>
```

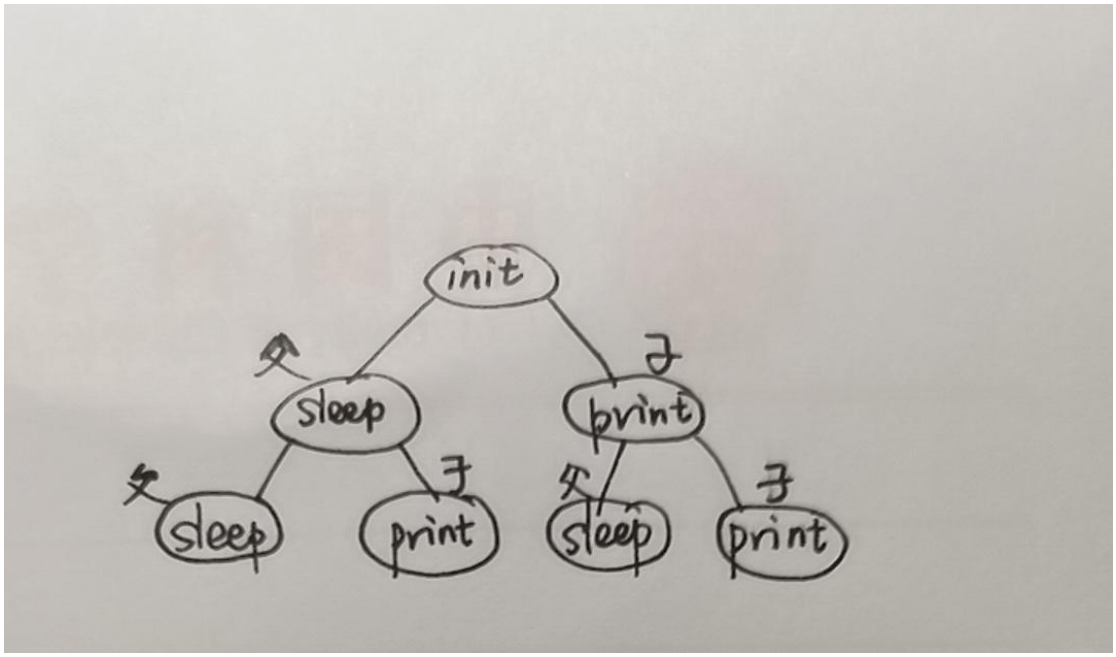
```
#define LOOP 2

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;
}
```

答：（1）总共会生成**3个子进程，关系图如下：**



说明：该关系树的每个叶子节点都代表一个父/子进程，每一层的叶子节点的父子进程性都是相对其父亲节点而言的。

理论上我们看到的输出应该是：先迅速打印出两行" I am child process\n",

停几秒后再输出一行" I am child process\n"，再停几秒后结束运行。

运行现象与预期一致！可以说明所画的关系图是正确的。

（2）**修改后的代码**如下：

```c
#include<unistd.h>
#include<stdio.h>
#include<string.h>
#define LOOP 4

int main(int argc,char *argv[])
{
    pid_t pid;
    int loop;
    int flag = 0;

    for(loop=0;loop<LOOP;loop++) {
        if(flag == 0){
            pid = fork();
         if(pid < 0)
            fprintf(stderr, "fork failed\n");
         else if(pid == 0) {
            printf(" I am child process\n");
         }
         else {
            sleep(1);
            flag = 1;
         }
        }

     }
     return 0;
}
```

结果截图：（将LOOP修改为4，输出4行）

```
ubuntu@ubuntu:~/Desktop$ gcc -g hw.c -o hw
ubuntu@ubuntu:~/Desktop$ ./hw
 I am child process
 I am child process
 I am child process
 I am child process
ubuntu@ubuntu:~/Desktop$
```

可以看到，生成4个子进程，与宏定义相符。

代码说明：该代码实质上是对关系树进行了“剪枝”，让任意层级的父进程不会衍生新的子进程，只保留纯粹子进程的“树枝”，由于循环次数与关系树的层数一致，我们得到了LOOP个子进程，应该输出LOOP行，与运行结果相符。