**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）

#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：



（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：

execl函数用于在当前进程中执行一个新程序。这里，我们执行了ls -l /usr/bin命令。第一个参数是要执行的程序的路径，接下来的参数是传递给该程序的命令行参数，最后一个参数必须为NULL。

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。

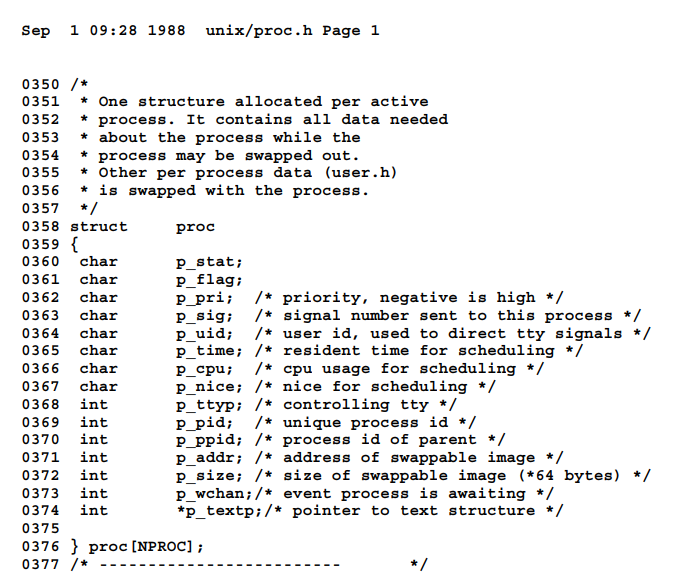
fork首先拷贝了整个父进程的代码、数据、栈堆等，子进程可以实现父进程完全一模一样的功能，但是如果调用exec函数族内的函数，则会将这整个拷贝丢弃了，并用要运行的文件替换内存的内容。所有拷贝的内存都被丢弃并被exec替换。wait系统调用只能等待当前进程的子进程。如果当前进程有任何子进程，那么wait会返回。可以利用copy-on-write fork，实现对fork调用的优化，这种方式会消除fork的几乎所有的明显的低效，而只拷贝执行exec所需要的内存。

运行结果如下：

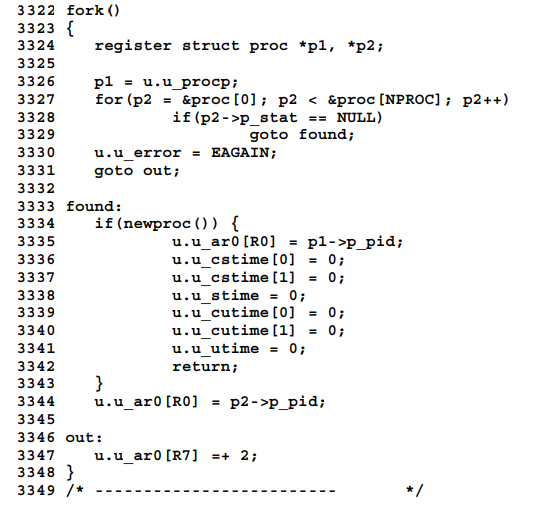


(3)理论分析：

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：



Fork函数定义在第3320行，具体执行操作如下：



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）、

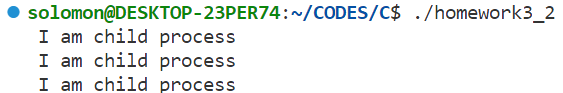
最初只有一个父进程P存在。在第一次循环中，父进程P调用fork()创建一个子进程P1，此时父子进程的代码是一样的，它们会继续执行循环。在第一次循环中，父进程P会进入sleep(5)，而子进程P1会输出"I am child process"。

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：



（2）、

#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);

      }

    }

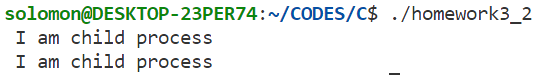
    return 0;

}

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.



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。