作业 3

贾城昊
2021K8009929010

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

(1)

1. C 程序如下：

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

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

    for(i = 0;i < 10;i++){
        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;
}
```

2. 运行结果：

```
I'm the son process,and the sum=45
parent process finishes
```

3. 代码与结果说明：

程序 1 中，第 14 行调用了 fork 函数，当调用 fork()时，操作系统会创建一个新的进程，并复制父进程的内存、文件描述符和其他资源到子进程中。子进程会继承父进程的代码、数据、堆和栈，但它们是独立的，所以它们可以独立运行，互不影响。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，然后让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 函数用来阻塞父进程的运行直至子进程运行结束，同时 wait 函数还分析子进程是否结束运行，若在父进程中忘记调用，子进程会进入没有父进程的状态，成为僵尸进程。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。

(2)

1. C 程序如下：

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

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

    for(i = 0;i < 10;i++){
        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/sai/workspace/OS", NULL);

        exit(1);
    }
    else {
        wait(&status);
        printf("parent process finishes\n");}
    return 0;
}
```

2. 运行结果：

```
I'm the son process,and the sum=45
总用量 96
-rwxrwxr-x 1 sai sai 8880 9月  12 14:09 getpid
-rw-rw-rw- 1 sai sai 1298 9月  12 14:09 getpid.c
-rwxrwxr-x 1 sai sai 8880 9月  12 17:51 hw1
-rwxrwxr-x 1 sai sai 8776 9月   7 15:36 hw1_2
-rw-rw-r-- 1 sai sai  515 9月   7 15:35 hw1_2.c
-rwxrwxr-x 1 sai sai 8720 9月   7 15:39 hw1_3
-rw-rw-r-- 1 sai sai  668 9月   7 15:39 hw1_3.c
-rw-rw-r-- 1 sai sai 1333 9月  12 17:51 hw1.c
-rwxrwxr-x 1 sai sai 8960 9月  19 22:42 hw2
-rw-rw-r-- 1 sai sai  976 9月  19 22:42 hw2.c
-rwxrwxr-x 1 sai sai 8952 9月  24 23:17 hw3
-rw-rw-r-- 1 sai sai  610 9月  24 23:17 hw3.c
parent process finishes
```

3. 代码与结果说明：

上面代码较第一版只添加了一行，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 /home/sai/workspace/OS 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。

这个程序中， fork 首先拷贝了整个父进程的代码、数据、栈堆等，从而让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，则会把拷贝给替换了，并用要运行的文件替换内存的内容。所以加上这一行后，可以让子进程在对数组进行完求和操作后执行 ls -l 命令。而 wait 函数仍然会等待子进程结束，同时接受一个子进程退出状态的值。

（注：一共有 6 种 exec 函数，分别为 execl，execlp，execle，execv，execvp，execve。其区别在于它们的后缀，含义如下：

·l 和 v 的区别在于参数传递方式。l 表示逐个列举的方式，而 v 表示将所有参数整体构成指针数组进行传递。

·e 表示可以指定当前进程的环境变量，而替换掉继承的环境变量。

·p 表示查找文件时可以只给出文件名，系统会从环境变量“$PATH”指出的路径进行查找）

(3)

通过阅读相关资料，得知XV6下在“kernel/proc.h”内声明了进程ID，进程状态等，其中的进程控制模块结构体定义如下：

```
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

可以看到，在struct proc中有关进程的各种信息，包括进程状态（state字段）、进程ID（pid字段）、父进程指针（parent字段）、进程大小（sz字段）、文件描述符表（ofile字

段）、当前目录（cwd字段）、进程名称（name字段）、内核栈（kstack字段）等。

函数fork()在“kernel/proc.c”中定义，相关代码如下：

```
// Create a new process, copying the parent.
// Sets up child kernel stack to return as if from fork() system call.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;

  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);

  // Cause fork to return 0 in the child.
  np->trapframe->a0 = 0;

  // increment reference counts on open file descriptors.
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);
```

```
  safestrcpy(np->name, p->name, sizeof(p->name));

  pid = np->pid;

  release(&np->lock);

  acquire(&wait_lock);
  np->parent = p;
  release(&wait_lock);

  acquire(&np->lock);
  np->state = RUNNABLE;
  release(&np->lock);

  return pid;
}
```

当fork执行时，XV6会创建一个新的进程（子进程），并将父进程的所有状态复制到子进程中，包括进程控制块。具体来说，fork会做以下操作：

1. 创建一个新的进程控制块（PCB）结构体，初始化它的各个字段，包括分配内存等，通过uvmcopy()函数将父进程的程序控制块复制给子进程。。
2. 复制父进程的寄存器状态（*(np->trapframe) = *(p->trapframe);），并令a0寄存器的值为0 （np->trapframe->a0 = 0;），以确保子进程从fork调用后继续执行。
3. 复制父进程的文件描述符表（ofile字段），并父进程所在的目录复制给子进程，最后还会把父进程的名称复制给子进程
4. 设置子进程的状态为RUNNABLE，使其可以被调度执行。
5. 在父进程和子进程中返回不同的值，以便区分它们。通常情况下，fork会返回子进程的进程ID（在父进程中返回），而子进程会返回0。

这样，fork系统调用会创建一个新的进程，该进程在几乎所有方面都是父进程的副本，然后在返回后，父进程和子进程可以在不同的执行路径上继续执行。

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

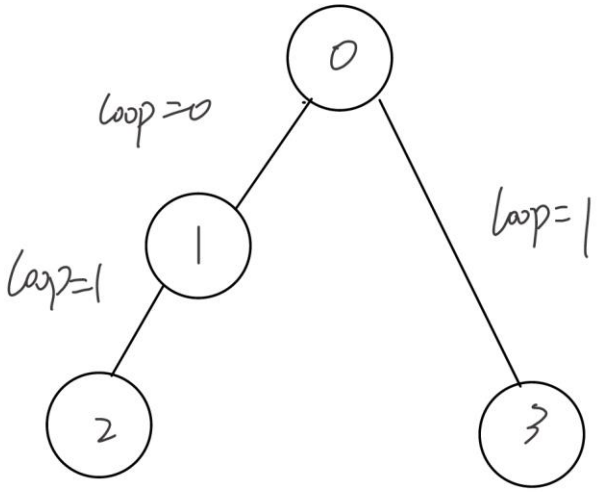
```
#include<unistd.h>
#include<stdio.h>
#include<string.h>
#include<sys/types.h>
#define LOOP 2

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)

1. 运行代码结果：

```
I am child process
I am child process
I am child process
```

2. 关系图：



3. 解释说明：

可以得知，这个程序实际上产生了三个子进程。这是因为最开始主程序 0 进入 for 循环时，loop=0，主程序 0 执行 fork()函数复制出了一个子进程，这个子进程 1 的数据与主程序 0 相同，所以其 loop 也为 0，而又由于主进程 0 休眠 5s，此时子进程 1 先运行，所以子程序 1 打印一个“I am child process”。 然后子进程 1 中 loop 变为 1，再通过 fork()函数复制出一个子进程 2，同理此时子进程 2 的 loop 也为 1，子进程 2 再打印一个“I am child process”。此时子进程 1 休眠 5s，然后子进程 2 的 loop 值先后变为 2，for 循环结束工作。然后主程序 0 休眠完毕，开始工作，其内存中 loop 值仍为 1，故还可以重复一次上面子进程 1 所做的事，再打印一个“I am child process”(而之前子进程 1 休眠完毕后，loop 变为 2，结束 for 循环)。综上，一共打印了三次，生成了三个子进程。

(2)

1. 程序代码与说明

为了让一个进程只生成一次子进，可以将 sleep(5)改为 break，并将 sleep(5)移动到 for 循环外面(这样是为了让子程序打印字符串在父程序结束之前)，即可达到效果，改后的代码如下：

```c
#include<unistd.h>
#include<stdio.h>
#include<string.h>
#include<sys/types.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 {
         break;
       }
      }

      sleep(5);

    return 0;
}
```

2. 运行结果

执行结果：

```
sai@Computer:~/workspace/OS$ ./hw3
 I am child process
 I am child process
sai@Computer:~/workspace/OS$
```

将 LOOP 的值改为 5，得到结果也满足，结果如下：

```
sai@Computer:~/workspace/OS$ ./hw3
 I am child process
 I am child process
 I am child process
 I am child process
 I am child process
sai@Computer:~/workspace/OS$
```