作业 3

毕定钧 2021K8009906014

本次作业包含:

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。

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？

```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");
      }
      else {
         sleep(5);
      }
   }
   return 0;
}
```

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。

1 Task 3.1

1.1 fork、exec、wait 的使用方法

1.1.1 fork

fork 是一个在类 *Unix* 操作系统中常见的系统调用，用于创建一个新的进程。新进程是调用进程（父进程）的一个副本，这个副本包含了父进程的代码、数据和环境变量等信息。*fork* 调用会创建一个与调用进程几乎完全相同的新进程。

一个进程调用 *fork* 后，系统先给新的进程分配资源，例如存储数据和代码的空间等，然后将原来的进程的所有值都复制到新的新进程中，包括内存内容、文件描述符、环境变量等资源。这个新进程和父进程几乎是完全相同的，只有一些特定的属性会有所不同，称之为子进程。需要注意的是，调用 *fork* 之后，两个进程同时执行的代码段是 *fork* 函数之后的代码，而之前的代码已经由父进程执行完毕，即子进程事实上是继续执行父进程的工作。在父进程中，*fork* 调用返回子进程的进程 *ID* (*PID*)，而在子进程中，它返回 0。这使得父子进程可以根据返回值来执行不同的操作。通常，父进程和子进程在 *fork* 调用之后并行执行，它们共享相同的程序代码，但拥有独立的执行环境。

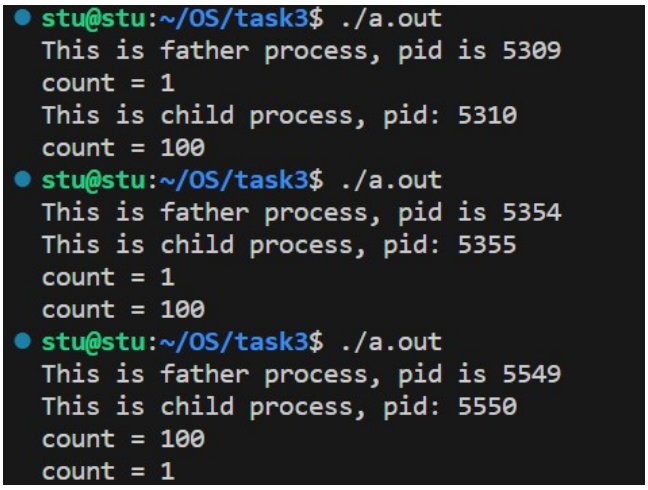
fork 仅能用于创建新进程，如果需要更复杂的进程间通信或协作，通常需要使用其他系统调用。

下面是一个简单的使用 fork 创建子进程的一个示例程序：

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    int count = 0;
    pid = fork();
    if(pid == 0) {
        printf("This is child process, pid: %d\n", getpid());
        count += 100;
        printf("count = %d\n", count);
    }
    else if(pid > 0) {
        printf("This is father process, pid is %d\n",getpid());
        count++;
        printf("count = %d\n", count);
    }
    else {
        fprintf(stderr,"ERROR:fork() failed!\n");
    }
    return 0;
}
```

其运行结果如下：



父进程将 *count* 增加 1，而子进程将 *count* 增加 100，这两个进程打印出来的 *count* 并没有呈现出相关关系。同时可以看到，四条语句的打印顺序并不是固定的，这也就说明 $fork$ 调用所创建的子进程与父进程是并行关系。

本部分参考： ChatGPT、进程系统调用——fork 函数深入理解 (代码演示)、关于理解 fork() 函数的简单例子。

1.1.2 exec

exec 的主要作用是将当前进程的地址空间和执行上下文替换为一个新程序的地址空间和执行上下文，从而实现程序的切换，它通常需要提供新程序的路径和参数列表。新程序的路径是一个指向可执行文件的字符串，参数列表是一个数组，其中包含了传递给新程序的命令行参数，这些参数通常包括程序名称本身和其他参数。*exec* 甚至还可以传递一个环境变量数组，用于设置新程序的环境变量，这允许了开发者在新程序中自定义环境变量。如果调用成功，不会返回，而是直接在新程序中继续执行，如果出错，则返回 -1，并设置全局变量 *errno* 以指示错误类型。

值得注意的是，*exec* 是以新的进程去代替原来的进程，但进程的 PID 保持不变。因此，可以认为，*exec* 系统调用并没有创建新的进程，只是替换了原来进程上下文的内容。原进程的代码段，数据段，堆栈段均被新的进程所代替。

exec 通常与 $fork$ 一起使用。可以使用 $fork$ 来创建一个新的进程，然后在子进程中使用 *exec* 来加载和执行不同的程序，这样可以实现在一个进程中执行多个不同的程序。$fork$ 建立新进程之后，父子进程共享代码段，但数据空间是分开的。上面说到，父进程会把自己数据空间的内容和上下文拷贝到子进程中去，为了提高效率，可以采用一种写时拷贝的策略，即创建子进程的时候，并不赋值父进程的地址空间，父子进程拥有共同的地址空间，只有当子进程需要写入数据时 (如向缓冲区写入数据)，才会复制地址空间、缓冲区到子进程中去。这样一来，父子进程拥有了独立的地址空间。$fork$ 之后执行 *exec* 能够很好的提高效率，如果一开始就拷贝，那么 *exec* 之后，子进程的数据会被放弃，而被新的进程所代替。

接下来是一个使用 $fork$ 和 *exec* 进行进程转换的示例程序：

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

int main() {
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        printf("This is child process, pid: %d\n", getpid());
        execl("/bin/ls", "ls", "-l", NULL);
        perror("exec");
        return 1;
    }
    else if (pid > 0) {
        printf("This is father process, pid is %d\n", getpid());
        int status;
        waitpid(pid, &status, 0);
        if (WIFEXITED(status)) {
            printf("Quit child process with status %d\n", WEXITSTATUS(status));
        }
        return 0;
    }
    else {
        fprintf(stderr,"ERROR: fork() failed!\n");
    }
}
```

其运行结果如下：

```
This is father process, pid is 6111
This is child process, pid: 6112
total 32
-rw-rw-r-- 1 stu stu   419 Sep 23 21:30 3-1-1.c
-rw-rw-r-- 1 stu stu   660 Sep 23 21:36 3-1-2.c
-rw-rw-r-- 1 stu stu   379 Sep 20 10:28 3-2.c
-rwxrwxr-x 1 stu stu 17056 Sep 23 21:36 a.out
Quit child process with status 0
```

本部分参考： ChatGPT、Linux 下的 exec 系统调用详解、exec, wait 系统调用。

1.1.3 wait

wait 通常用于处理子进程的终止状态和资源回收，在类 *Unix* 操作系统中，父进程可以使用 *wait* 来等待其创建的子进程终止，*wait* 调用会使父进程阻塞，直到它的一个子进程终止，如果父进程没有子进程或所有子进程都还没有终止，那么它将一直阻塞等待。一旦子进程终止，*wait* 调用会返回子进程的进程 *ID*（*PID*）以及子进程的终止状态信息。父进程可以使用这些信息来了解子进程的退出状态、退出原因等。*wait* 调用还负责回收子进程的资源，包括释放子进程占用的内存、文件描述符和其他资源。父进程可以多次调用 *wait*，每次等待一个子进程终止，或者可以使用 *waitpid* 函数来等待特定的子进程终止。如果父进程不希望阻塞等待子进程终止，可以使用 *waitpid* 函数的 *WNOHANG* 选

项，它允许父进程立即返回，如果没有子进程终止，返回值为 0。同样的，如果 *wait* 调用失败，通常会返回 -1，并设置全局变量 *errno* 以指示错误类型。

关于 *wait* 的返回值，如果任何进程有多个子进程，则在调用 *wait* 之后，如果没有子进程终止，则父进程必须处于 *wait* 状态。如果只有一个子进程被终止，那么 *wait* 返回被终止的子进程的进程 ID。如果多个子进程被终止，那么 *wait* 将获取任意子进程并返回该子进程的进程 ID。如果进程没有子进程，那么 *wait* 返回 -1。

父进程调用 *wait* 时传一个整型变量地址给函数，内核将子进程的退出状态保存在这个变量中。如果子进程调用 *exit* 退出，那么内核把 *exit* 的返回值存放到这个整数变量中；如果进程是被 *kill* 的，那么内核将信号序号存放在这个变量中。这个整数由三部分组成，8 个 *bit* 记录子进程 *exit* 值，7 个 *bit* 记录信号序号，另一个 *bit* 用来指明发生错误并产生了内核映像（*core dump*）。

在前面的示例中，已经使用了 wait 来等待子进程的终止，因此不再示例。

本部分参考：ChatGPT、了解 C 语言中的 wait 系统调用

1.2 C 程序及其结果说明

1.2.1 Task 3.1-(1)

```c
int main() {
    int numbers[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        int sum = 0;
        for (int i = 0; i < 10; i++) {
            sum += numbers[i];
        }
        printf("sum = %d\n", sum);
        return 0;
    }
    else if (pid > 0) {
        int status;
        waitpid(pid, &status, 0);

        if (WIFEXITED(status)) {
            printf("parent process finishes\n");
        }
        return 0;
    }
    else {
        fprintf(stderr,"ERROR: fork() failed!\n");
    }
}
```

运行结果如下：

```
sum = 55
parent process finishes
```

在该程序中，首先创建了一个数组 $numbers$ 用于存储从 1 到 10 的十个数，之后使用 $fork$ 建立子进程，$pid = 0$ 即当前为子进程，进行求和并打印结果；$pid > 0$ 说明为父进程，使用 $wait$ 调用等待子进程结束后退出父进程。

1.2.2 Task 3.1-(2)

在之前的子进程后添加如下代码：

```
char *args[] = { "ls", "-l", "/usr/bin/ls", NULL };
execvp("ls", args);

perror("execvp");
return 1;
```

运行结果如下：

```
sum = 55
-rwxr-xr-x 1 root root 142144 Sep  5  2019 /usr/bin/ls
parent process finishes
```

由于题目要求的 $/usr/bin$ 目录内容物实在太多了，我仅打印了该目录下的 ls 文件夹，看来似乎是个空的文件夹。不完整的打印 $/usr/bin$ 的结果如下：

```
-rwxr-xr-x 1 root   root         1802 Apr  8  2022  xzless
-rwxr-xr-x 1 root   root         2161 Apr  8  2022  xzmore
-rwxr-xr-x 1 root   root          590 Jul  6 18:57  y2racc2.7
lrwxrwxrwx 1 root   root           22 Aug 24  2021  yacc -> /etc/alternatives/yacc
-rwxr-xr-x 1 root   root        63720 Apr  6  2021  yelp
-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 Apr  8  2022  zcat
-rwxr-xr-x 1 root   root         1678 Apr  8  2022  zcmp
-rwxr-xr-x 1 root   root         5898 Apr  8  2022  zdiff
-rwxr-xr-x 1 root   root        26840 Apr  7  2022  zdump
-rwxr-xr-x 1 root   root           29 Apr  8  2022  zegrep
-rwxr-xr-x 1 root   root       227808 Mar 23  2020  zeitgeist-daemon
-rwxr-xr-x 1 root   root       145664 Mar 23  2020  zeitgeist-datahub
-rwxr-xr-x 1 root   root       135960 Feb 27  2020  zenity
-rwxr-xr-x 1 root   root           29 Apr  8  2022  zfgrep
-rwxr-xr-x 1 root   root         2081 Apr  8  2022  zforce
-rwxr-xr-x 1 root   root         8103 Apr  8  2022  zgrep
-rwxr-xr-x 1 root   root       216256 Apr 22  2017  zip
-rwxr-xr-x 1 root   root        93816 Apr 22  2017  zipcloak
-rwxr-xr-x 1 root   root        50718 May 24 01:17  zipdetails
-rwxr-xr-x 1 root   root         2953 Oct  8  2022  zipgrep
-rwxr-xr-x 2 root   root       186664 Oct  8  2022  zipinfo
-rwxr-xr-x 1 root   root        89488 Apr 22  2017  zipnote
-rwxr-xr-x 1 root   root        93584 Apr 22  2017  zipsplit
-rwxr-xr-x 1 root   root        26952 Mar 20  2023  zjsdecode
-rwxr-xr-x 1 root   root         2206 Apr  8  2022  zless
-rwxr-xr-x 1 root   root         1842 Apr  8  2022  zmore
```

1.3 Task 3.1-(3)

我们知道，操作系统负责管理所有进程，包括进程的创建和消亡、进程状态的转换以及分配和回收进程所需要的资源等。操作系统管理和控制进程的过程，全部要借助进程控制块才能完成。在创建每个进程时，操作系统都会额外申请一块内存空间，用来存储、管理和控制该进程所需要的信息，比如进程名称或 ID 号、当前进程的执行状态（开始、就绪、运行、等待或者终止）、进程占用的各种资

源（包括内存大小、使用的输入输出设备等）、进程已经执行的时间，占用 CPU 的时间等。我们通常将这样的存储空间称为进程控制块（$Process\ Control\ Block$，简称 PCB）。也就是说，进程控制块记录了进程当前运行情况以及所占资源的详细信息，并由操作系统负责管理和维护。操作系统中进程和进程控制块的数量始终是相等的，创建多少个进程就会相应产生多少个进程控制块。它是进程存在的唯一标识，只有借助进程控制块，操作系统才能找到目标进程，进而实施管理和控制。当进程执行结束后，操作系统只需要释放相应进程控制块占用的内存空间，目标进程也随之消亡。

事实上，PCB 是一种数据结构，其作用是使一个在多道程序环境下不能独立运行的程序（包含数据），成为一个能独立运行的基本单位，一个能与其它进程并发执行的进程。

$XV6$ 的 PCB 定义在 "$/xv6/kernel/proc.h$" 中，其源码如下：

```
// Per-process state
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，$XV6$ 的 PCB 包括进程状态、睡眠时队列、是否被杀死、退出状态、进程 ID、父进程指针、内核栈指针、内存空间大小、进程的页表、中断进程后需要保存的寄存器内容、切换进程所需要保存的进程状态、文件的组、进程当前目录、进程名这些内容。

1.3.1 fork

fork 函数定义在 "$/xv6/kernel/proc.c$" 中，其源码如下：

```c
// 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;
}
```

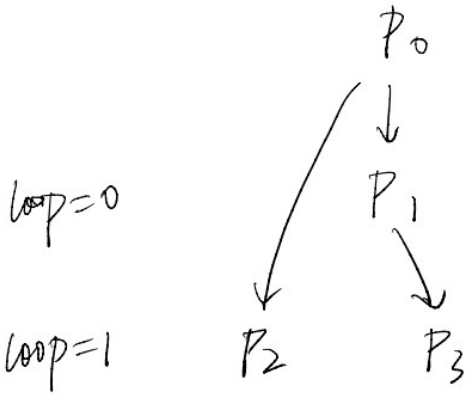
阅读代码发现，$fork$ 首先新建了一个进程控制块 np，并获取当前进程的 PCB，之后将当前进程的 PCB 复制到新建的 np 的进程控制块中。复制的内容包括用户页表、内存空间大小、数据段（其

中将子进程的数据段的 $a0$ 设为 0)、进程名称。之后将子进程的父进程指针设为 p，然后将子进程状态设为就绪状态并返回子进程 pid。其中如果有内存分配失败、拷贝失败等情况 $fork$ 均会退出并返回 -1。

2 Task 3.2

2.1 Task3.2-(1)

一共生成了三个子进程，其进程关系如下：

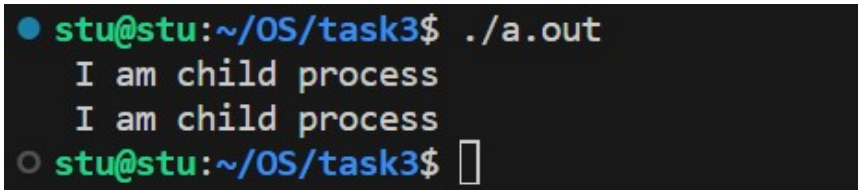


2.2 Task3.2-(2)

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

    for(loop=0;loop<LOOP;loop++) {
        if (pid)
            pid = fork();
        if(pid < 0)
            fprintf(stderr, "fork failed\n");
        else if(pid == 0) {
            printf(" I am child process\n");
        }
        else {
            sleep(5);
        }
    }
    return 0;
}
```

运行结果如下：



在创建子进程前进行判断，如果当前是父进程则继续创建，否则不创建。