Chapter 3 Homework

陈文迪 519021910071

作业中的引用内容均已标出

3.1 根据下方展示的程序，分析LINE A处会出现什么输出。

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int value = 5;
int main()
{
pid t pid;
    pid = fork();
    if (pid == 0) { /* child process */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d",value); /* LINE A */
        return 0;
    }
}
```

输出结果： PARENT: value = 5

从代码中我们可以看出，`fork()` 产生的子进程并没有调用 `exec()` ，因此该子进程是作为父进程的副本执行，并且子进程和父进程具有各自的数据副本，二者并发执行。在子进程中进行的数据修改并不会影响父进程中对应的数据，因此 `value` 的值仍然是初始值 `5`。

3.2 根据下方展示的程序，包括最初的父进程在内，有多少进程被创建了?

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    /* fork a child process */
    fork();
    /* fork another child process */
    fork();
    /* and fork another */
    fork();
    return 0;
}
```

回答： 8个。

第一次 `fork()` 之后总共有2个进程，这两个进程都将执行第一次 `fork()` 在之后的指令。因此他们都将执行第二次 `fork()` ， 此时总共有4个进程。同理，执行完第三次 `fork()` 之后共有 $2^3 = 8$ 个进程。

3.4 一些计算机系统提供多个寄存器组。描述如果新的上下文已经加载到其中一个寄存器组中时发生上下文切换，将会发生什么情况。如果新的上下文在内存中而不是在寄存器组中，并且所有寄存器组都在使用，会发生什么情况?

对于具有多个寄存器组的处理器来说，如果新的上下文已经加载到其中一个寄存器组中，

> 上下文切换只需简单改变当前寄存器组的指针。

这样的处理方法可以大大减少上下文切换所花费的时间。

然而，如果所有寄存器组都已在使用，而新上下文还在内存中，我们则需要

> 在寄存器与内存之间进行数据复制。

也就是将某一寄存器组中的某一上下文复制到内存中，再把将要使用的上下文复制到该寄存器组中。并且，

> 操作系统越复杂，上下文切换所要做的就越多。高级的内存管理技术再每次上下文切换时，所需切换的数据会更多。

这就要求操作系统的设计人员能设计一种高效的内存管理方法。

3.8 描述内核在进程之间进行上下文切换而采取的操作。

操作系统中，

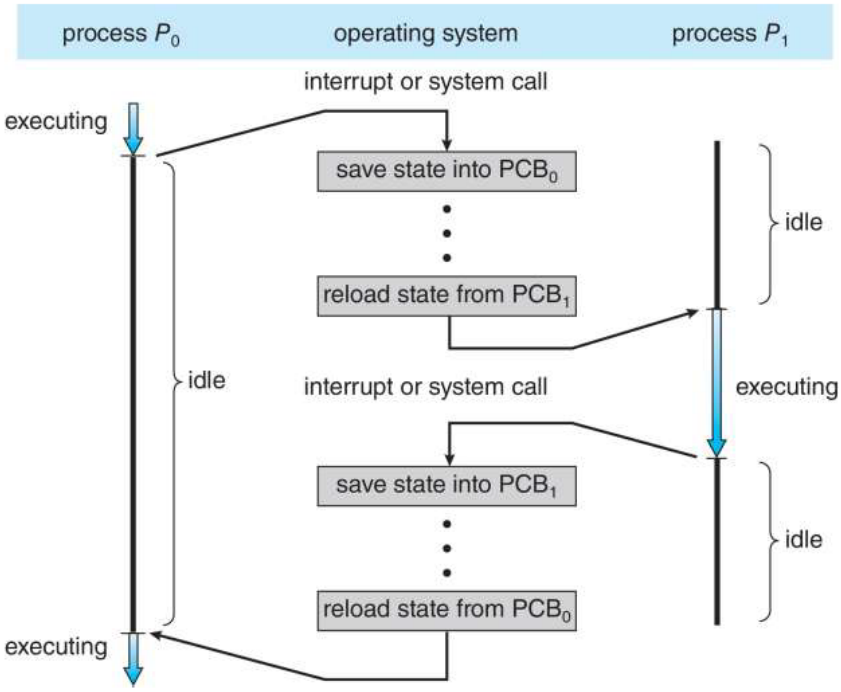
> 进程上下文采用进程PCB表示，包括CPU寄存器的值、进程状态和内存管理信息。通过执行**状态保存**，保存CPU当前状态；之后，**状态恢复**重新开始运行。

根据进程上下文的结构特点和其功能，在进行上下文切换时，内核也需要采取相应的操作。具体来说

> 内核会将旧进程状态保存在其PCB中，然后加载经调度而要执行的新进程的上下文。

整个过程的具体实现和消耗会随着机器的不同、硬件结构的不同和体系结构的不同发生变化。

下图展示了上下文切换的整个过程。



3.10 解释 init（或systemd）进程在UNIX和Linux系统上在进程终止方面的作用。

UNIX和Linux的进程管理采用了一个树状结构，每个子进程都有其对于的父进程，其中 `init` 作为所有用户进程的根进程或父进程。

> 当一个进程终止时，操作系统会释放其资源。不过，它位于进程表中的条目还是在的，直到它的父进程调用 `wait()` 。有时候，父进程在调用 `wait()` 前就提前终止了，这时候它的子进程就成为了**孤儿进程**。

为了解决这样的问题，UNIX和Linux采取的方法是把 `init` 进程作为该孤儿进程的父进程。并且，

> 进程 `init` 定期调用 `wait()` ，以便收集任何孤儿进程的退出状态，并释放孤儿进程标识符和进程表条目。