1使用下图所示的程序，说明LINE A的输出内容。

#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**;**

**}**

**}**

**答：**PARENT: value = 5

2包括初始父进程，下面显示的程序创建了多少进程？

#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：4个**

3当进程使用fork（）操作创建新进程时，父进程和子进程之间共享以下哪些状态？

a.堆

b.堆

c.共享内存段

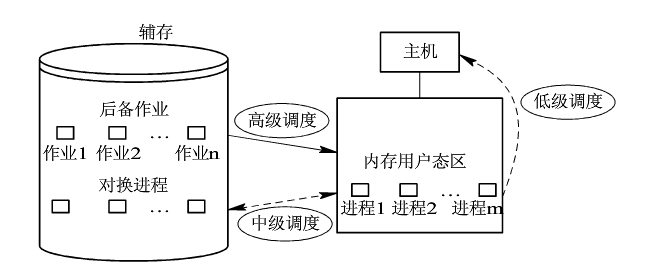
4描述短期，中期和长期调度之间的差异。

回答：a.短期调度:在内存作业中选择就绪执行的作业,并为他们分配CPU。

b.中期调度:作为一种中等程度的调度程序，尤其被用于分时系统，一个交换方案的实施，将部分运行程序移出内存，之后，从中断处继续执行。

c.长期调度（作业调度程序）:确定哪些作业调入内存以执行.

它们主要的不同之处是它们的执行的频率。短期调度必须经常调用一个新进程，由于在系统中，长期调度处理移动的作业时，并不频繁被调用，可能在进程离开系统时才被唤起。



5描述内核对进程之间上下文切换所采取的操作。

回答：总的来说，操作系统必须保存正在运行的进程的状态，恢复进程的状态。保存进程的状态主要包括CPU寄存器的值以及内存分配，上下文切换还必须执行一些确切体系结构的操作，包括刷新数据和指令缓存。

进程关联是由进程的PCB来表示的，它包括CPU寄存器的值和内存管理信息等。当发生上下文切换时，内核会将旧进程的关联状态保存在其PCB中，然后装入经调度要执行的新进程的已保存的关联状态。

6包括初始父进程，下面显示的程序创建了多少进程？

#include < stdio.h > #include < unistd.h > int main**() {**

inti**;**

**for(**i**=**0**;**i**<**4**;**i**++)**

fork**();**

# return 0;

**}**

**回答：5个（包括父进程在内）**

7说明将在下面的代码中标记printf（“LINE J”）的代码行的情况。

#include < sys/types.h >

#include < stdio.h > #include < unistd.h > int main**() {**

pid t pid**;**

/\* fork a child process \*/ pid **=** fork**();**

**if (**pid **<** 0**) {** /\* error occurred \*/ fprintf**(**stderr**,** "Fork Failed"**); return** 1**;**

**} else if (**pid **==** 0**) {** /\* child process \*/ execlp**(**"/bin/ls"**,**"ls"**,NULL);** printf**(**"LINE J"**);**

**} else {** /\* parent process \*/

/\* parent will wait for the child to complete \*/ wait**(NULL);**

printf**(**"Child Complete"**);**

# } return 0;

**}**

8使用以下程序，识别A，B，C和D行的pid值。假设父和子的实际pid分别为2600和2603。）

#include < sys/types.h >

#include < stdio.h > #include < unistd.h > int main**() {**

pid t pid**,** pid1**;** /\* fork a child process \*/ pid **=** fork**();**

**if (**pid **<** 0**) {** /\* error occurred \*/ fprintf**(**stderr**,** "Fork Failed"**); return** 1**;**

**} else if (**pid **==** 0**) {** /\* child process \*/ pid1 **=** getpid**();**

printf**(**"child: pid = %d"**,**pid**);** /\* A \*/ printf**(**"child: pid1 = %d"**,**pid1**);** /\* B \*/

**} else {** /\* parent process \*/ pid1 **=** getpid**();**

printf**(**"parent: pid = %d"**,**pid**);** /\* C \*/ printf**(**"parent: pid1 = %d"**,**pid1**);** /\* D \*/ wait**(NULL);**

# } return 0;

**}**

9举例说明普通管道比命名管道更合适的情况，以及命名管道比普通管道更合适的情况示例。

有名字在父子进程之间

10使用下面显示的程序，解释X和Y行的输出结果。

#include < sys/types.h >

#include < stdio.h >

#include < unistd.h > #define SIZE 5 int nums**[**SIZE**] = {** 0**,**1**,**2**,**3**,**4 **} ;** int main**() {**

int i**;** pid t pid**;** pid **=** fork**(); if (**pid **==** 0**) { for (**i **=** 0**;** i **<** SIZE**;** i**++) {**

nums**[**i**] \*= -**i**;**

printf**(**"CHILD: %d "**,**nums**[**i**]);** /\* LINE X \*/ **}**

# } else if (pid > 0) { wait(NULL);

**for (**i **=** 0**;** i **<** SIZE**;** i**++)**

printf**(**"PARENT: %d "**,**nums**[**i**]);** /\* LINE Y \*/

# } return 0;

**}**

11创建线程时使用了哪些资源？它们与创建流程时使用的有何不同？

12提供两个编程示例，其中多线程不提供比单线程解决方案更好的性能。

回答：（1）任何形式的顺序程序对线程来说都不是一个好的形式。例如一个计算个人报酬的程序。

（2）另外一个例子是一个“空壳”程序，如C-shell和korn shell。这种程序必须密切检测其本身的工作空间。如打开的文件、环境变量和当前工作目录。

13多线程进程中的线程之间共享以下哪个程序状态组件？

a.**寄存值（这里我一开始翻译成了注册值2333）**

b.堆内存

c.全局变量

d.栈内存

答：一个线程程序的线程共享b.堆内存和c.全局变量，但每个线程都有属于自己的一组寄存值和栈内存。

14考虑以下代码段：

pid t pid; pid = fork();

if (pid == 0) { /\* child process \*/

fork();

thread create( . . .);

}

fork();

a.创建了多少个唯一进程？

b.创建了多少个唯一线程？

15下面显示的程序使用Pthreads API。 LINE C和LINE P的程序输出是什么？

#include <pthread.h>

#include <stdio.h>

#include <types.h> int value **=** 0**;**

void **\***runner**(**void **\***param**);** /\* the thread \*/ int main**(**int argc**,** char **\***argv**[]) {**

pid\_t pid**;** pthread\_t tid**;** pthread\_attr\_t attr**;** pid **=** fork**();**

**}**

**if (**pid **==** 0**) {** /\* child process \*/ pthread\_attr\_init**(&**attr**);** pthread\_create**(&**tid**,&**attr**,**runner**,NULL);** pthread\_join**(**tid**,NULL);**

printf**(**"CHILD: value = %d"**,**value**);** /\* LINE C \*/

**} else if (**pid **>** 0**) {** /\* parent process \*/ wait**(NULL);**

printf**(**"PARENT: value = %d"**,**value**);** /\* LINE P \*/

**}**

void **\***runner**(**void **\***param**) {**

value **=** 5**;**

pthread\_exit**(**0**);**

**}**

**6我们提到经常禁用中断会影响系统的时钟。解释为什么会发生这种情况以及如何最大限度地减少这种影响。**

**17忙等待一词的含义是什么？操作系统还有哪些其他类型的等待？可以一共忙碌等待吗？解释你的答案。**

回答：忙等待意味着进程正在等待在紧密循环中满足条件而且不放弃处理器。或者说，进程可以等待放弃处理器，阻塞条件并等待在将来的某个适当时间唤醒。可以避免繁忙的等待，但是会产生与使进程进入睡眠状态相关的开销，并且在达到适当的程序状态时必须将其唤醒。

**18解释为什么自旋锁不适用于单处理器系统，但通常用于多处理器系统。**

答：自旋锁不适合在单处理器系统是因为从自旋锁中打破一个进程的条件只有在执行一个不同的进程时才能获得。如果这个进程没有闲置处理器，其他进程不能够得到这个机会去设定一个第一个进程进展需要的程序条件。在一个多处理器系统中，其他进程在其他处理器上执行，从而为了让第一个进程从自旋锁中释放修改程序状态。

**19显然，如果wait（）和signal（）信号量操作未以原子方式执行，则可能违反了互斥。**

答：等待操作以原子方式递减与信号量相关联的值。如果在信号量为1时对信号量执行了两次等待操作，如果两次操作不是原子执行的，则两个操作都可能继续递减信号量值，从而违反互斥。

**20许多计算机系统都可以使用竞争条件。考虑一个银行系统，它通过两种功能维持账户余额：存款（金额）和提款（金额）。这两项功能通过了从银行账户余额中存入或取出的金额。假设丈夫和妻子共用一个银行账户。同时，丈夫调用withdraw（）函数，妻子调用deposit（）。描述竞争条件如何可能以及可以采取哪些措施来防止竞争条件的发生。**

**21解释为什么中断不适合在多处理器系统中实现同步元。**

答：由于只有在防止其他进程在一个中断不能实现的处理器上执行来停止中断，中断在多处理器系统中是不够的。在对于进程能在其他处理器上执行是没有心智的，所以进程停止中断不能保证互斥进入程序状态。