2152118 史君宝 操作系统 中断与调度的复习题 订正

第一题：

1、进程不执行系统调用就不会入睡。 错误

2、现运行进程不响应中断就不会被剥夺。 正确

3、现运行进程不响应中断就不会让出 CPU。 正确

4、现运行进程让出 CPU 后，一定是优先级最高的进程上台运行。 错误

5、Unix V6++系统使用的调度算法是时间片轮转调度。 正确

6、没有中断就没有调度（现运行进程就不会让出 CPU）。 正确

7、用户态进程，系统中至多只有一个。 错误(多道程序)

8、Unix V6++，核心态不调度。所以，如果不是入睡或终止，现运行进程不会让出 CPU。
错误

错误订正：

3、现运行进程不响应中断就不会让出 CPU。
错。入睡或终止也会让出 CPU。
（但入睡或终止需要进程执行系统调用，强调系统调用是广义中断，那这题也是对的）

5、Unix V6++系统使用的调度算法是时间片轮转调度。
错。可剥夺的动态优先级调度算法。

7、用户态进程，系统中至多只有一个。
对。现运行进程执行应用程序时是系统唯一的用户态进程。
就绪，阻塞进程全在核心态。立即返回用户态的就绪进程也要先在核心态完成恢复用户态现场的操作。

8、Unix V6++，核心态不调度。所以，如果不是入睡或终止，现运行核心态进程不会让出 CPU。
对。

第二题：
自己的答案：

3、将系统调用的返回结果传给应用程序？ 执行完毕后，内核会将系统调用的返回值放入 EAX 寄存器中。

正确答案：

3、将系统调用的返回结果传给应用程序？ 系统调用的返回值用 EAX 寄存器。其它数据，可以存入其它通用寄存器 或是 钩子函数传入的指针变量所指的用户空间内存区域。

第三题：
自己的答案：

1、描述 20#系统调用的执行过程。

解答:

(1) 用户态会调用一个钩子函数 getpid()，在里面完成系统调用的全过程:

```
int getpid()
{
1:   int res;
2:   __asm__ volatile ( "int $0x80":"=a"(res):"a"(20) ) ;
3:   if ( res >= 0 )
4:         return res ;

5:   else
6:   {
7:         errno = -res ;
8:         return -1;
9:   }
}
```

(2) 会先将 20#系统调用的调用号 20 送入 EAX 寄存器。
(3) 然后会通过 EBX，ECX，EDX，ESI，EDI 传递参数，在本题中应该没有。
(4) 之后会执行 int 0X80 指令，执行具体的系统调用的处理程序。
(5) 最后会将返回结果放入 res 变量中，通过 EAX 寄存器实现返回。

正确答案：

(一) 描述 20#系统调用的执行过程。

答:

1. 应用程序调用系统调用的钩子函数
2. 钩子函数将系统调用号 20 传入 EAX，执行 int 80h 陷入内核

3. 系统调用入口函数 SystemCallEntrance()保存用户态寄存器，将系统调用号 20 存入核心栈。
4. 系统调用处理程序 Trap()，从核心栈取出系统调用号，查系统调用表 m_SystemEntranceTable，间接调用系统调用子程序 Sys_Getpid( )。
5. Sys_Getpid( )将系统调用的返回值（现运行进程的 p_pid）存入核心栈、u_ar0 指向的单元。系统调用返回用户态后该单元的值将回传 EAX 寄存器。钩子函数将其传回应用程序。

第四题：
自己的答案：

五、擦掉红色的判断，Unix V6++系统的钟就不走了。为什么 ？　（这个题写着玩）

```
void Time::Clock( struct pt_regs* regs, struct pt_context* context )
{
    ......
    if( current->p_stat == Process::SRUN &&
          (context->xcs & USER_MODE) == KERNEL_MODE )
    {
        发 EOI 命令;
        return;
    }
```

```
    Time::lbolt -= HZ;
    Time::time++; //修改 wall clock time
    ......
}
```

解答：
我们看到时钟值变化的是下面的代码，
Time::lbolt -= HZ;
Time::time++; //修改 wall clock time

删除了 current->p_stat == Process::SRUN 之后上述代码不再执行，说明进入了相关的 if 分支，这个函数弹出去了。
这是因为在系统运行过程中，(context->xcs & USER_MODE) == KERNEL_MODE)大部分时间都是成立的，所以总是进入其相关的 if 分支，return 回去。

正确答案：

五、擦掉红色的判断，Unix V6++系统的钟就不走了。为什么？　（这个题写着玩）

```
void Time::Clock( struct pt_regs* regs, struct pt_context* context )
{
    ......
    if( current->p_stat == Process::SRUN &&
            (context->xcs & USER_MODE) == KERNEL_MODE )
    {
        发 EOI 命令;
        return;
    }

    Time::lbolt -= HZ;
    Time::time++; //修改 wall clock time
    ......
}
```

系统 idle 的时候，整数秒要调整 time。此时，CPU KERNEL_MODE，现运行的 0#进程 SSLEEP。删掉红色的判断，idle 时走 if 分支，无法调整 lbolt 和 time。