

中断与调度部分的复习题

同济大学计算机系 操作系统作业

2023-11-20

学号

2152118

姓名

史君宝

一、判断题

- 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

二、系统调用不同于一般的子程序调用。请问：UNIX V6++和 Linux 的系统调用如何

- 1、传递应用程序想要执行的系统调用号？ 在系统调用的过程中，会通过 EAX 寄存器传递想要执行的系统调用号。
- 2、传递系统调用的参数？ 在系统调用的过程中，会通过 EBX, ECX, EDX, ESI, EDI 寄存器传递想要的参数
- 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。

三、言简意赅

- 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。

3、描述为 Unix V6++ 系统添加一个新的系统调用的过程。

解答:

这个在之前的实验中已经做过了:

添加系统调用的具体步骤:

步骤一: 在系统调用子程序入口表中添加新的入口。

步骤二: 在 SystemCall 类中添加系统调用处理子程序的定义。

步骤三: 在 SystemCall.cpp 中添加 Sys_Getppid 的具体定义。

添加库函数的具体步骤:

步骤一: 在 sys.h 文件中添加库函数的声明。

步骤二: 在 sys.c 中添加库函数的定义。

步骤三: 编译程序。

四、 请回答以下问题, 言简意赅补齐系统中中断响应和调度过程。

1、T0 时刻整数秒, 系统中 SRUN 进程 PA 和 PB。现运行进程 PA 执行 sleep (10) 系统调用。

解答:

上述的 Sleep(10)属于系统调用。我们按照具体的系统调用的过程执行就可以了。

- (1) 会将系统调用号送 EAX, 并传递参数 10(为时间长度), 之后会引发内联的汇编指令, 执行具体的中断程序。
- (2) 首先会执行中断隐指令, 保护硬件现场, 之后调用系统调用入口程序 `SystemCall::SystemCallEntrance()`, 并执行中断处理程序 `SystemCall::Ssleep()`, 在计算睡眠时间后, 调用 PA 进程的 `Sleep()`函数进行设置。
- (3) 在 `Sleep` 中会先进行关中断设置 `p_wchan`, `p_stat`, `p_pri`, 之后会开中断执行 `Swch`, 此时 PA 优先级并非最高, 会放弃 CPU, PB 进程切换上台运行

2、现运行进程 PA SRUN, 正在执行系统调用。T1 时刻, 响应中断, 唤醒一个睡眠进程 PB。问, PB 进程何时上台运行? 简述系统中断响应, 调度过程和 PB 唤醒后上台运行

解答:

- (1) T1 时刻, 系统正在执行系统调用程序, 在这时响应中断, 会发生中断嵌套, 并唤醒 PB。
- (2) 中断处理程序执行完成, 返回的时候, 由于之前为核心态, 正在执行系统调用。所以并不会例行调度, 会继续执行系统调用程序。
- (3) 系统调用程序执行结束后, 由于之前为用户态, 会进行例行调度选择优先级最高的 SRUN 进程上台运行, 此时如果 PB 进程的优先级最高就可以上台运行, 否则只能等待其他进程执行。直到进程调度选择它上台才能执行。

3、T2 时刻整数秒, CPU 关中断执行硬盘中断处理程序, 硬盘中断处理程序的先前态是用户态。时钟中断何时响应? 时钟 time 的调整是否会延迟, 延迟到什么时候?

- (1) 由于此时 CPU 进行了关中断, 无法立即响应中断。当只有当 CPU 开中断之后才会响应时钟中断。
- (2) 由于硬盘中断处理程序在核心态运行, 因此本次整数秒时钟中断会发生延迟。
- (3) 时钟 time 的调整需要延迟到下一个整数秒的时钟中断处理程序 (并且相应前需为用户态)。

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。

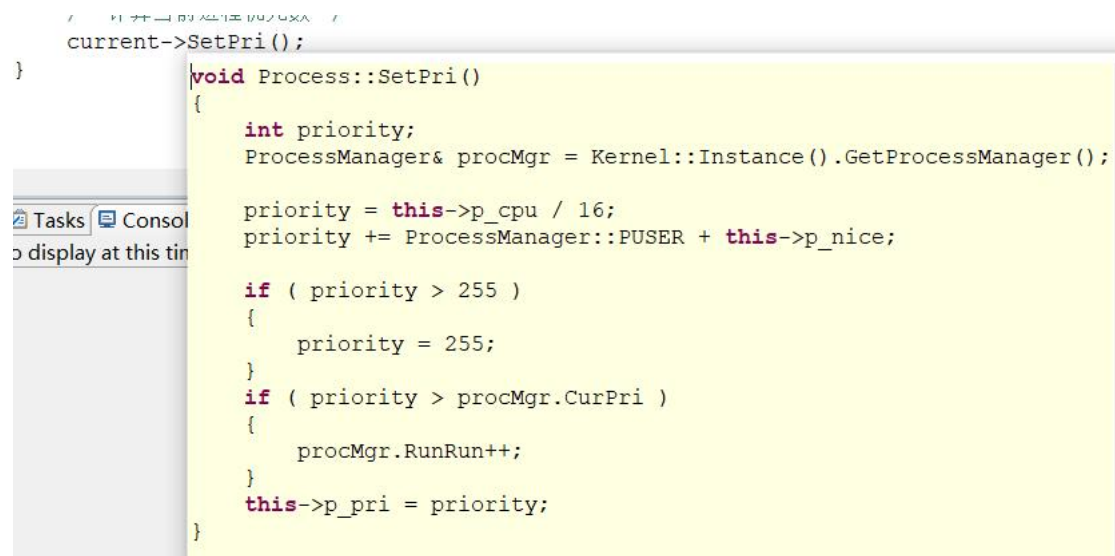
六、Setpri() 有没有一点儿怪。说出你的疑惑, 尝试解释原系统设计的合理性, 或对它进行质疑。
(这个题写着玩)

解答:

下面是 Setpri() 函数的原码:

具体过程为先是计算 $p_pri = \text{进程静态优先数} + (p_cpu/16)$

这都没有问题, 后面的大于 `priority > 255` 的计算也没有问题, 那问题大概在最后一个分支, 应该是 `p_pri` 低于当前执行进程的 `CurPri` 时, `RunRun++` 进行进程调度, 但是代码里似乎不太对。



```

current->SetPri();
}

void Process::SetPri()
{
    int priority;
    ProcessManager& procMgr = Kernel::Instance().GetProcessManager();

    priority = this->p_cpu / 16;
    priority += ProcessManager::PUSER + this->p_nice;

    if ( priority > 255 )
    {
        priority = 255;
    }
    if ( priority > procMgr.CurPri )
    {
        procMgr.RunRun++;
    }
    this->p_pri = priority;
}

```

七、你自己的任何设计或想法 (这个题写着玩)

解答：

对于上面的 SetPri()的设计似乎有些问题，应该改变相关的内容，或者说改变 Setpri()函数的意义，让他不是比较执行进程优先级，实现进程切换。相反，这种原码表述更似乎是我们的进程在随着它在 CPU 的执行时间长了，而降低优先级的意义，建议进行修改。