

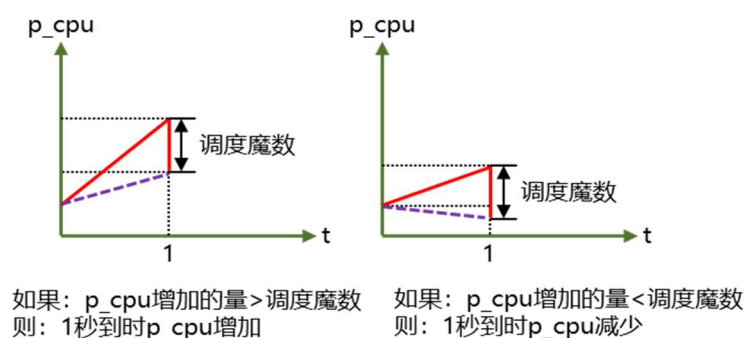
E06: 并发进程 (UNIX 进程与中断)

参考答案与说明

1. D
2. B
3. C
4. A
5. D

6. 【参考答案】

答: (1) 每次时钟中断中, 现运行进程的 p_cpu 加 1; (2) 每秒计时到的时钟中断中, 所有进程的 p_cpu 减去调度模数。这样做的结果是: (1) 如果进程一秒内 p_cpu 增加的量 $>$ 调度魔数, 则: 1 秒到时 p_cpu 增加; (2) 如果进程 p_cpu 增加的量 $<$ 调度魔数, 则: 1 秒到时 p_cpu 减少。如下图所示。



7. 【参考答案】

答: UNIX 中有 3 次重新计算进程优先数的时机: (1) 每秒计时到的时钟中断里, 重算所有用户态就绪进程的优先数; (2) 每秒计时到的时钟中断里, 重算现运行进程的优先数; (3) 系统调用结束时, 重算现运行进程的优先数。

8. 【参考答案】(详细过程供大家参考, 黄色标注部分要求掌握)

1. 进程在用户态下执行对库函数 `getpid()` 的调用, 库函数 `getpid()` 的代码如下:

```
=====
int getpid()
{
    int res;
    __asm__ volatile ( "int $0x80 \":\" =a\" (res):\"a\"(20) );
    if ( res >= 0 )
        return res;
    else{
        error = -res;
        return -1;
    }
}
=====
```

执行库函数时进程的核心栈如图 1 所示。完成的工作包括：

- 1.1. 将系统调用号 20 送入 EAX 寄存器。
- 1.2. 设置系统调用的返回值由 EAX 寄存器带回，并赋值给 res。
- 1.3. 通过 INT 80H 指令启动中断。转向步骤 2。
- 1.4. 中断返回后，将 EAX 寄存器中的返回值（当前进程的 ID 号）赋值给 res。
- 1.5. 如果 res>=0，返回 res 的值，否则，返回-1，并设置出错码。
- 1.6. getpid() 函数返回。此时用户栈如图 2 所示。

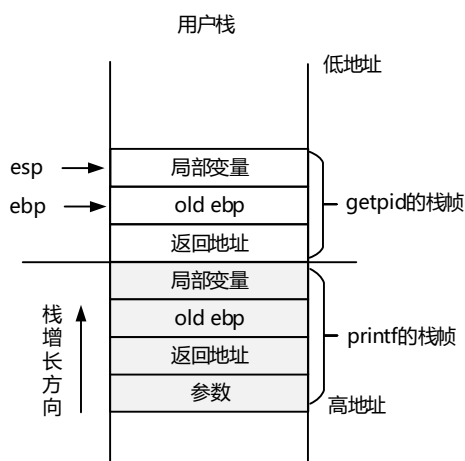


图 1

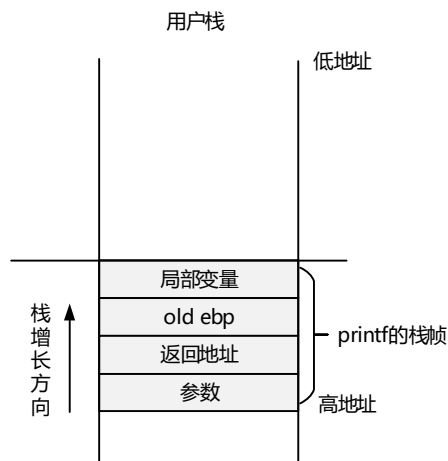


图 2

2. CPU 执行中断隐指令响应 80H 号中断，完成下列工作：

- 2.1. 关中断。
- 2.2. 装入中断向量：根据中断号 80H，查询 IDT 表，获取中断向量。将其中的 Segment Selector 装入 CS，使 CPU 转入核心态；Offset 指向的系统调用入口程序 SystemCall::SystemCallEntrance() 的入口地址装入 EIP，以实现程序跳转。
- 2.3. 将 CS, EIP, EFLAGS 等寄存器的值压入现运行进程核心栈，形成硬件现场，如图 3 所示。

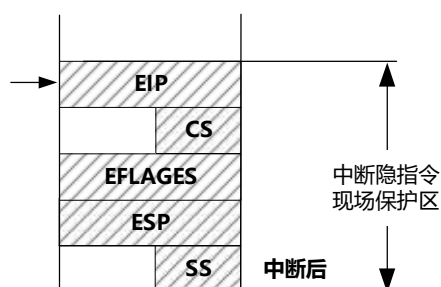


图 3

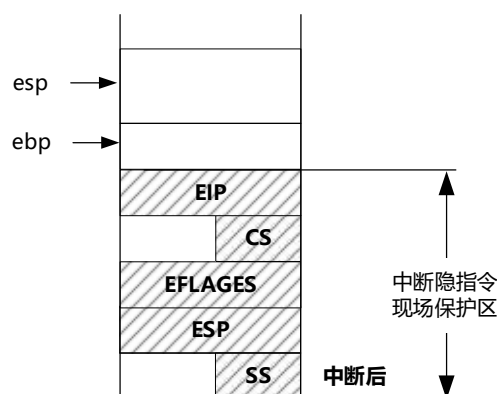


图 4

- 2.4. 调用系统调用入口程序 SystemCall::SystemCallEntrance()，核心栈变化如图 4 所示（由于此处机器指令调用了 C 语言函数，所以栈帧中只有 Old Ebp 和局部变量区）。转向步骤 3。

3. 执行系统调用入口程序 `SystemCall::SystemCallEntrance()`，完成如下工作：

- 3.1. 宏 `SaveContext()` 继续保存中断现场，形成软件现场如图 5 所示。
- 3.2. 宏 `SwiThToKernel()` 完成对 DS, ES, SS 的赋值，指向核心态数据段描述符。
- 3.3. 调用中断处理程序 `SystemCall::Trap(struct pt_regs* regs, struct pt_context* context)`，核心栈变化如图 6 所示（由于此处为汇编指令 `call` 调用了 C 语言函数，所以栈帧中包含返回地址，Old Ebp 和局部变量区）。转向步骤 4。
- 3.4. 例行调度：因为系统调用前为用户态，检查 `RunRun` 的标志位是否为 0。如果不为 0，进行进程的切换调度。否则，
- 3.5. 宏 `RestoreContext()` 弹出核心栈中的软件现场，恢复 CPU 中各个寄存器的值。核心栈回到图 4 所示的状态。
- 3.6. 宏 `Leave()` 删除核心栈中的中断入口程序的栈帧。核心栈回到图 3 所示的状态。
- 3.7. 宏 `InterruptReturn()` 执行中断返回指令，删除核心栈中的硬件现场。CPU 返回用户态。

转向步骤 1.4。

4. 执行系统调用处理程序 `SystemCall::Trap`，完成如下工作：

- 4.1. 根据核心栈中保存 EAX 单元（`regs->eax` 指向）中记录的系统调用号 20，查询系统调用处理子程序入口表，获得 `Sys_Getpid()` 的入口地址和所需的参数个数（为 0）。
- 4.2. 读入参数：因为 `Sys_Getpid()` 没有参数，所以无需从核心栈中读入参数到 User 结构中。
- 4.3. 调用 `Sys_Getpid()`，代码如下：

```
=====
int SystemCall::Sys_Getpid()
{
    User& u = Kernel::Instance().GetUser();
    u.u_ar0[User::EAX] = u.u_procp->p_pid;
    return 0;    /* GCC likes it ! */
}
=====
```

完成的主要工作为：将当前进程的 ID 号写入核心栈中保存 EAX 的单元。

- 4.4. `Sys_Getpid()` 函数返回。
- 4.5. 重新计算当前进程的优先数。
- 4.6. 系统调用处理程序 `SystemCall::Trap` 返回。核心栈恢复到图 5 所示的状态。转向步骤 3.4。

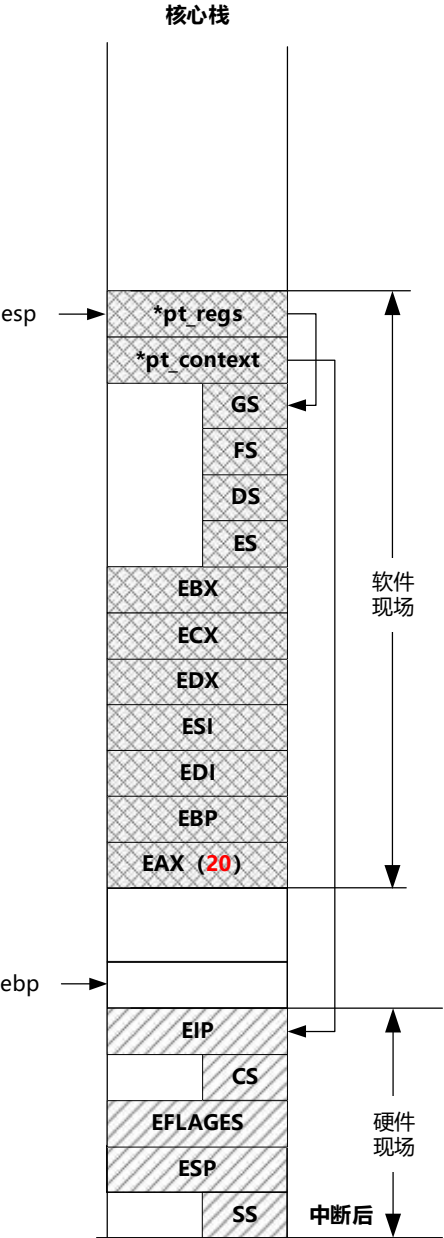


图 5

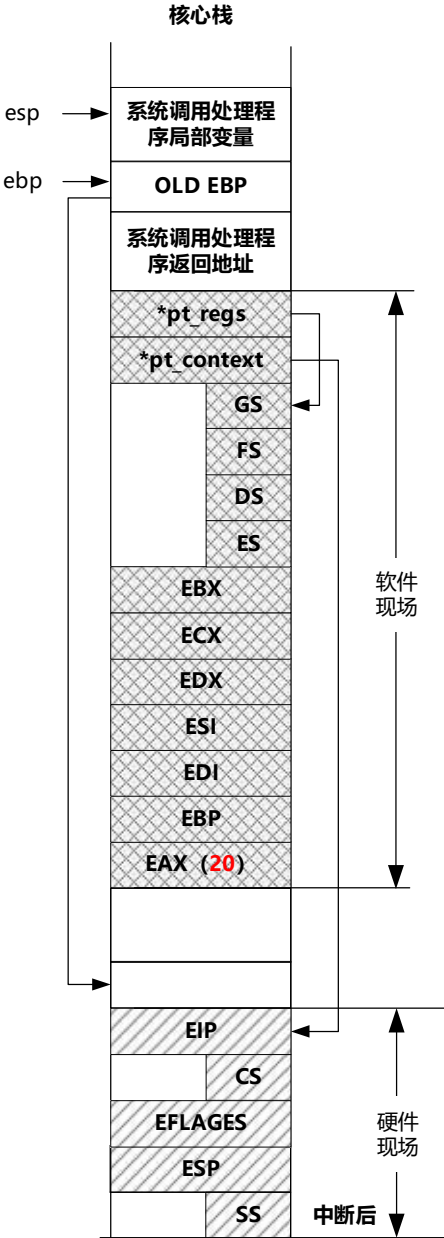


图 6