

## E08: UNIX 进程控制二

### 一、简答题:

1. 当以下标志出现时, UNIX 会产生什么动作?

- `RunRun==1` 表示有比当前进程更适合上台的进程, 此时会发生进程调度, 现运行进程下台
- `Runin==1` 表示盘交换区上有就绪进程图像需要调入内存, 但是内存空间不够, 此时0#进程将上台, 将现运行进程的图像搬到盘交换区, 让盘交换区上就绪进程图像可以放到内存
- `Runout==1` 表示盘交换区上没有进程图像可以调入内存, 0#进程将入睡等待盘交换区上出现第一个就绪状态进程

### 二、应用题

1. 假设某 UNIX 系统中只存在 4 个进程, 分别是: 睡眠中的 0#进程, 正在 CPU 上执行用户程序的 pa 进程, 内存中就绪状态的 pb 进程, 盘交换区上低优先级睡眠的 pc 进程。请尽量详细地分析以下时刻系统中与进程调度相关的行为。

- (1) T0 时刻, 现运行进程 pa 执行 read (读文件) 系统调用
- (2) T1 时刻, pa 启动的 I/O 操作完成
- (3) T2 时刻, pa 正在执行用户程序时, pc 等待的 I/O 操作完成

(1) 执行系统调用, 引发INT 80中断, 进入核心态, 如果系统调用中使用外设, 那么进程调用Sleep入睡, 登记睡眠原因, 修改进程调度状态并且设置进程优先数, 此时pa是低优先级睡眠状态, 此时pa让出处理机, 执行swtch切换调度, 然后保护pa运行现场并将现运行进程改为0#进程, 0#进程执行ProcessManager: : Select() 从处于就绪状态 (p\_stat=SRUN) 且图像在内存

(p\_flag&Process::SLOAD==1)的队列中选择优先级最高的进程, 本题中为pb。0#进程选中pb, 将pb的PPDA区所在的物理页框号写入0x201#页框的1023#记录, 然后恢复pb现场, 即从pb的USER结构中取出下台时保存的esp和ebp, pb变为现运行进程, 并执行Swtch程序的最后一段, 首先设置0x202和203两张用户页表, 然后判断SSWAP标志是否被设置, 如果被设置说明进程已经做好了交换出内存的准备, 所以要把USER结构中u\_ssav中两个地址值再次恢复esp和ebp。

(2) T1时刻, 进程paI/O操作完成, pa被唤醒, pb此时进入核心态, 调用ProcessManager::WakeupAll()清除睡眠原因, 修改状态, 然后比较pa和pb的优先权, 如果pa优先权高于现运行进程pb, 那么设置RunRun标志, 由于进程图像在内存, 无需判断RunOut标志, 然后pa进入核心态就绪, 被调度上台完成系统调用的剩余部分, 然后系统调用中断返回, pa仍然是优先级最高的进程, 于是pa直接返回用户态执行用户程序。

(3) 此时到了T2, pc等待的I/O操作完成, 则pc在盘交换区上被唤醒, 由于在唤醒pc前盘交换区上没有进程图像可以调入内存, 0#进程也会入睡, 所以此次唤醒操作也会唤醒0#进程, 0#进程上台运行, 此时内存有空, 则将pc调入内存, 此时现运行进程pa进入核心态, 执行ProcessManager::Sched操作, 计算pc所需的内存空间大小, 假设能分配到足够大的空闲区, 则将进程图像调入, 并释放磁盘空间, 修改p\_addr, p\_textp, p\_time等等, 然后中断返回, 执行例行调度, 计算pa和pc的优先级并选择进行上台

情景分析题: 假设某 UNIX 系统中, 所有进程在当前时刻 T0 进程状态如下表所示。且内存空间已满。请尽量详细地分析以下时刻系统中与进程调度相关的行为, 并修改下表中的相关字段。

序号	占用空间	状态	位置	年龄 (p_time)
0#	-	高睡 (RunOut)	SLOAD	-
p1	40K	执行	SLOAD	2
p2	30K	就绪	SLOAD	3
p3	30K	低睡	~SLOAD	3

(1) T0 时刻, p1 执行 read 系统调用。

(2) 此后 1 秒内, 系统中未发生进程的切换调度。当 T1 (T1=T0+1 秒) 时刻, 进程 p2 在用户态下执行, p3 等待的 I/O 操作完成。

(1) p1 执行系统调用, 产生中断, 调用外设, p1 变成低睡状态, p2 上台, 变成执行状态,

(2) 未发生进程切换调度说明p1等待的I/O操作还未完成, p1p2p3的年龄都+1, p3被唤醒, 此时盘交换区上有就绪进程需要调入内存, 但内存已满, RunIn标志被设置为1, 此时p1会唤醒0#进程, 将p1图像换入磁盘, 并将p3图像换入内存, 此时p1 低睡, ~SLOAD, 3 p3, 就绪, SLOAD 4

3. 假如有下面一段代码:

```
=====
main( )
{
    int i=0;
    int a=0;
    if( i=fork( ) )
    {
        a=a+1;
        printf(" i= %d; a= %d\n", i, a);
    }
    else
    {
        a=a+2;

        printf(" i= %d; a= %d\n", i, a);
    }
}
=====
```

- (1) 请指出上述程序中, `fork()`之后, 哪些语句为父进程执行, 哪些语句为子进程执行?
- (2) 写出程序的运行结果 (假设父进程的 ID 号为 500, 子进程的 ID 号为 505)。

父进程会执行if语句块中的第一个语句 `a=a+1;` 和 `printf(" i=%d; a=%d\n", i, a);`, 而子进程则会执行if语句块中的第二个语句 `a=a+2;` 和 `printf(" i=%d; a=%d\n", i, a);`。

父进程输出: `i=505; a=1` 子进程输出: `i=0; a=2`



或  
`i=0; a=2`  
`i=505; a=1`

4. 请阅读下列程序：

```
=====
#include <stdio.h>
#include <sys.h>
main1( )
{
    int i,j;
    if(fork())
    {
        i=wait(&j);
        printf("It is parent process. \n");
        printf("The finished child process is %d. \n", i);
        printf("The exit status is %d. \n", j);
    }
    else
    {
        printf("It is child process. \n");
        exit(1);
    }
}
=====
```

请写出程序的输出结果（假设父进程的 ID 号为 500，子进程的 ID 号为 505）。

这里实现了父子进程的同步

输出： It is child process

It is parent process

The finished child process is 505

The exit status is 1



