Unix V6++进程的睡眠唤醒操作

同济大学计算机系操作系统课程讲义　　邓蓉　2023-11-15

一、void Process::Sleep(unsigned long chan, int pri)

现运行进程执行 Sleep 函数入睡。chan 是睡眠原因，一个内核数据结构的地址；pri∈[-100，100)，是系统调用的睡眠优先数。pri < 0，入高优先权睡眠状态；pri>=0，入低优先权睡眠状态。

```
void Process::Sleep(unsigned long chan, int pri)
{
1:  User& u = Kernel::Instance().GetUser();
2:  ProcessManager& procMgr = Kernel::Instance().GetProcessManager();

3:  if ( pri > 0 )
    {
4:      if ( this->IsSig() )
        {
5:          aRetU(u.u_qsav);
6:          return;
        }

7:      X86Assembly::CLI();
8:      this->p_wchan = chan;
9:      this->p_stat = Process::SWAIT;
10:     this->p_pri = pri;
11:     X86Assembly::STI();

12:     if ( procMgr.RunIn != 0 )
        {
13:         procMgr.RunIn = 0;
14:         procMgr.WakeUpAll((unsigned long)&procMgr.RunIn);
        }

15:     Kernel::Instance().GetProcessManager().Swtch();

16:     if ( this->IsSig() )
        {
17:         aRetU(u.u_qsav);
18:         return;
```

```
        }
    }
18: else
    {
19:     X86Assembly::CLI();
20:     this->p_wchan = chan;
21:     this->p_stat = Process::SSLEEP;
22:     this->p_pri = pri;
23:     X86Assembly::STI();

24:     Kernel::Instance().GetProcessManager().Swtch();
    }
}
```

1、入睡基本操作，适用于高优先级睡眠进程

第 1 行，获得入睡进程的 user 结构。

```
User& u = Kernel::Instance().GetUser();
```

第 2 行，procMgr 指向 ProcessManager 对象 g_ProcessManager。

```
ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
```

第 3 行，根据 pri 分支，分别进入高优先权睡眠状态（18~24 行） 或 低优先权睡眠状态（4~17 行）。其中，

进程入睡的核心操作是 4 步，代码中加阴影部分：

```
19:     X86Assembly::CLI();
20:     this->p_wchan = chan;
21:     this->p_stat = Process::SSLEEP;
22:     this->p_pri = pri;
23:     X86Assembly::STI();

24:     Kernel::Instance().GetProcessManager().Swtch();
```

设置睡眠原因，设置调度状态，设置系统调用的优先级，最后调用 Swtch()放弃 CPU，完成运行态→阻塞态的状态变迁。

随后，系统将 CPU 分配给优先级最高的进程，这个进程完成 就绪→运行 的状态变迁。这里有个特例。现运行进程入睡后，如果不存在就绪态进程，现运行进程是 0#进程，SSLEEP 状态。对的，0#进程是在睡眠态执行 select()选择新运行进程的。

Sleep()函数什么时候返回呢？emm。。。计算机系统中所有的事儿都是现运行进程做的。从 Sleep()返回也是一样。所以，进程执行 Sleep()函数的过程是：入睡……等待……→ 事件发

生，被唤醒 → 被 0#进程选中，成为现运行进程 → Swtch()返回 → Sleep()返回 → 执行 Sleep()函数调用点之后的下条语句，系统调用得以继续执行。阴影部分，我们的进程在睡觉，历经的所有状态变换是其它进程辅助完成的。它们是谁呢？请大家自行思考（●ˇ∀ˇ●）

2、低优先权睡眠

进程进入低优先权睡眠状态后，可能会睡很久，甚至永远不会醒来。举例，

- 进程执行 scanf( )函数入睡。用户输入字符行之后，进程才会被键盘中断处理程序唤醒。决定进程睡眠时长的是人，不是计算机系统。可能是几秒钟，也可能是几个小时。
- 进程入睡读取网络数据。会睡很久，直到收到网络对端传回的数据。如果网络连接中断，进程可能永远睡眠，无法被唤醒。决定进程睡眠时长的是网络和对端服务器。
- Sleep(seconds)，进程会睡 seconds 秒。如果 seconds 值很大，SWAIT 状态会持续很久。

鉴于此，Unix 会对低优先权睡眠状态做特殊处理。

1、提高内存资源利用率

低睡、放弃 CPU 之前，现运行进程会检查盘交换区。如果有就绪进程，激活对换操作，自己进盘交换区，释放原先占用的内存空间，把它让给盘交换区上的就绪进程。代码 12~14 行唤醒 0#进程。

```
12:     if ( procMgr.RunIn != 0 )
        {
13:         procMgr.RunIn = 0;
14:         procMgr.WakeUpAll((unsigned long)&procMgr.RunIn);
        }
```

入睡进程放弃 CPU 后，0#进程上台运行，执行 Sched( )函数实施对换操作。对换操作细节参见 Sched( )函数介绍。

2、加快信号处理速度。

及时响应信号有助于提升分布式系统的运行效率。

低优先权睡眠进程会睡太久，待其苏醒恢复运行之后再行信号处理，分布式系统对信号的敏感度会变差。此外，更重要的是，很久以后回送响应，合作进程会被拖垮，从而降低系统的运行效率。

为了加快信号响应速度，Unix 在 Sleep( )函数低优先权入睡分支的 2 处检测信号。其一，代码中的第 4~6 行：低睡、放弃 CPU 之前，现运行进程检查信号。如果有收到信号，放弃入睡去处理信号。其二，代码中的第 16~18 行：信号会唤醒低优先权睡眠进程（参见信号处理部分，kill 函数）。唤醒后的进程，很快会得到运行机会，从 Swtch( )返回（代码第 15 行）。随后进程会去处理信号。这 2 处，代码是一致的：

```
if ( this->IsSig() )
{
    aRetU(u.u_qsav);
    return;
}
```

aRetU，从 user 结构，qsav 字段恢复 ESP、EBP 寄存器。这是 Trap1()栈帧的定位指

针。随后的 return 函数作用于 Trap1()栈帧，会促使我们的进程长跳转返回 Trap( )，立即从系统调用返回用户态，执行信号处理函数。细节将在系统调用和信号处理部分详述。

走这两个分支，会导致系统调用失败返回。原因是，进程没有得到待处理的新数据，没有睡足定时器设定的时间。。。

现在我们的系统调用出错返回，错误原因，被信号打断：u_error ==EINTR。

Unix 系统怎样处理出错的系统调用呢？这要看系统调用的类型：

- IO 数据的系统调用。数据输入、输出任务尚未完成，必需重启被信号打断的系统调用。一般，信号处理程序执行完毕后，
  - Linux 会自动重启这类系统调用，这个过程对用户透明。
  - Unix V6++的信号处理机制较为古老，需要应用程序判断系统调用的出错类型，重新执行被信号打断的系统调用。
- Sleep 系统调用，被信号打断后，默认不重启。因为，设置 Sleep 系统调用，通常就是为了等信号；所以，定时器设定的时间通常会大得夸张，以确保进程一定能够收到信号。这种系统调用，不必重启。

二、void ProcessManager::WakeUpAll(unsigned long chan)

Unix 内核将睡眠进程等待的事件与一个内核变量相关联。实现上，这个变量的地址，chan，是进程的睡眠原因 p_wchan。事件发生时，系统遍历 Process 数组，调用 SetRun( )函数，唤醒 p_wchan==chan 的所有进程。

```
void ProcessManager::WakeUpAll(unsigned long chan)
{
1:  for(int i = 0; i < ProcessManager::NPROC; i++)
    {
2:      if( this->process[i].IsSleepOn(chan) ) // p_wchan==chan ?
        {
3:          this->process[i].SetRun();
        }
    }
}
```

SetRun( )函数唤醒进程 process[i]（第 3 行），后者阻塞变就绪，重新拥有使用 CPU 的权力。

```
void Process::SetRun()
{
4:  ProcessManager& procMgr = Kernel::Instance().GetProcessManager();

5:  this->p_wchan = 0;    // this指向Process[i]
6:  this->p_stat = Process::SRUN;
7:  if ( this->p_pri < procMgr.CurPri )
```

```
    {
8:      procMgr.RunRun++;
    }
9:  if ( 0 != procMgr.RunOut && (this->p_flag & Process::SLOAD)== 0 )
    {
10:     procMgr.RunOut = 0;
11:     procMgr.WakeUpAll((unsigned long)&procMgr.RunOut);
    }
}
```

第 4 行：procMgr 是 ProcessManager 对象。引用 ProcessManager 对象是为了得到现运行进程的优先数 CurPri。

第 5、6 行：是状态转换的核心。 p_wchan = 0，清除唤醒事件。p_stat = SRUN，就绪、恢复 **this** 进程的调度资格。注意 2 点，p_pri 没变，被唤醒的进程用入睡时设置的优先数参与竞争 CPU；唤醒的进程，不会立即上台，至少要等到现运行进程让出 CPU。
第 7、8 行：如果 **this** 进程优先级高于现运行进程，置剥夺（强迫调度）标识 RunRun。现运行进程返回用户态前，会检查 RunRun，非 0，让出 CPU（ 结合中断入口函数，例行调度部分阅读）。这是 **Unix 剥夺现运行进程的手段**。
第 9~11 行：如果 **this** 进程是盘交换区上的进程（*SLOAD* 是 0）并且盘交换区先前没有就绪进程（*RunOut* 非 0），激活对换操作（唤醒 0#进程）。0#进程上台后，会将 **this** 进程换入内存。参见函数 Sched( )。