第四章

进程管理

主 要 内 容

4.1 UNIX时钟中断与异常

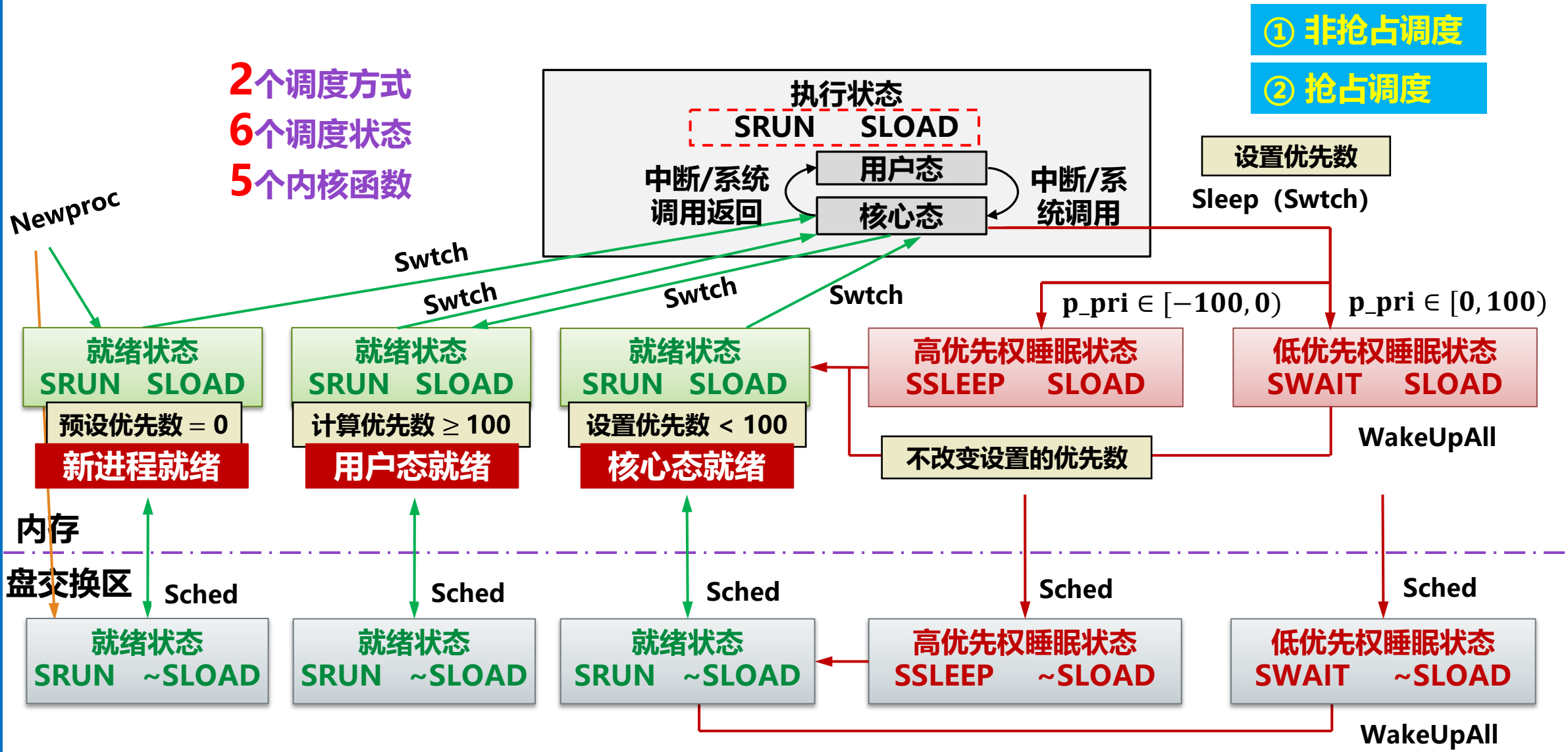
4.2 UNIX系统调用

4.3 UNIX的进程调度状态

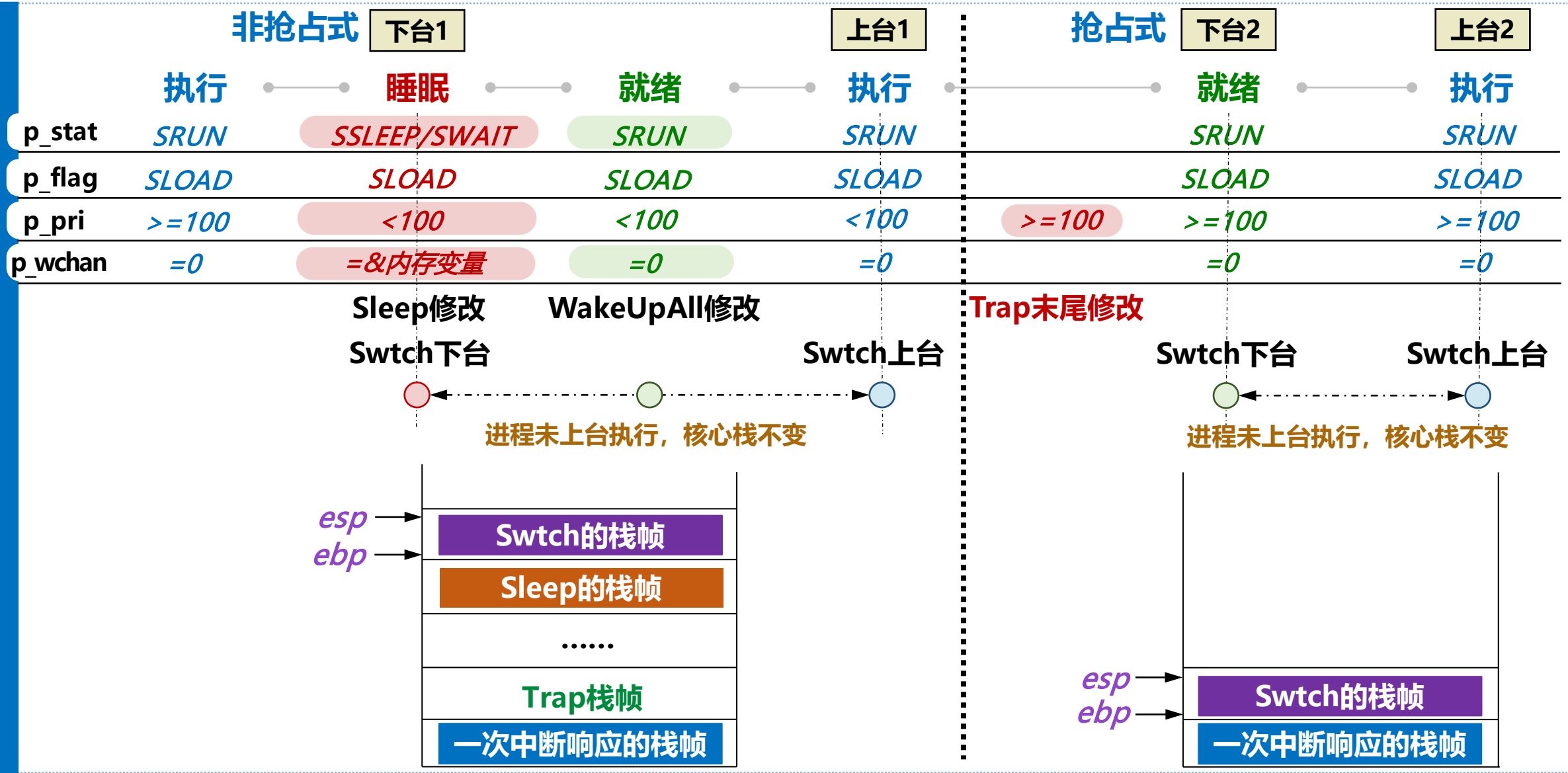
4.4 UNIX的进程调度控制

- 进程切换调度
- 进程创建与终止
- 进程图像交换

UNIX的进程调度状态



UNIX的进程调度状态



UNIX的进程调度控制

进程的睡眠与唤醒

进程的下台与上台

UNIX进程的睡眠与唤醒

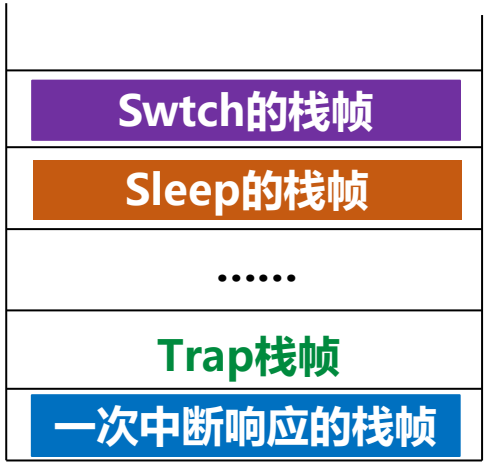
非抢占式 下台1 上台1　　抢占式 下台2 上台2

执行 — 睡眠 — 就绪 — 执行　　就绪 — 执行

	执行	睡眠	就绪	执行		就绪	执行
p_stat	SRUN	SSLEEP/SWAIT	SRUN	SRUN		SRUN	SRUN
p_flag	SLOAD	SLOAD	SLOAD	SLOAD		SLOAD	SLOAD
p_pri	>=100	<100	<100	<100	>=100	>=100	>=100
p_wchan	=0	=&内存变量	=0	=0		=0	=0

Sleep修改　WakeUpAll修改　　Trap末尾修改

Swtch下台　Swtch上台　　Swtch下台　Swtch上台



UNIX进程的睡眠与唤醒

设置优先数：

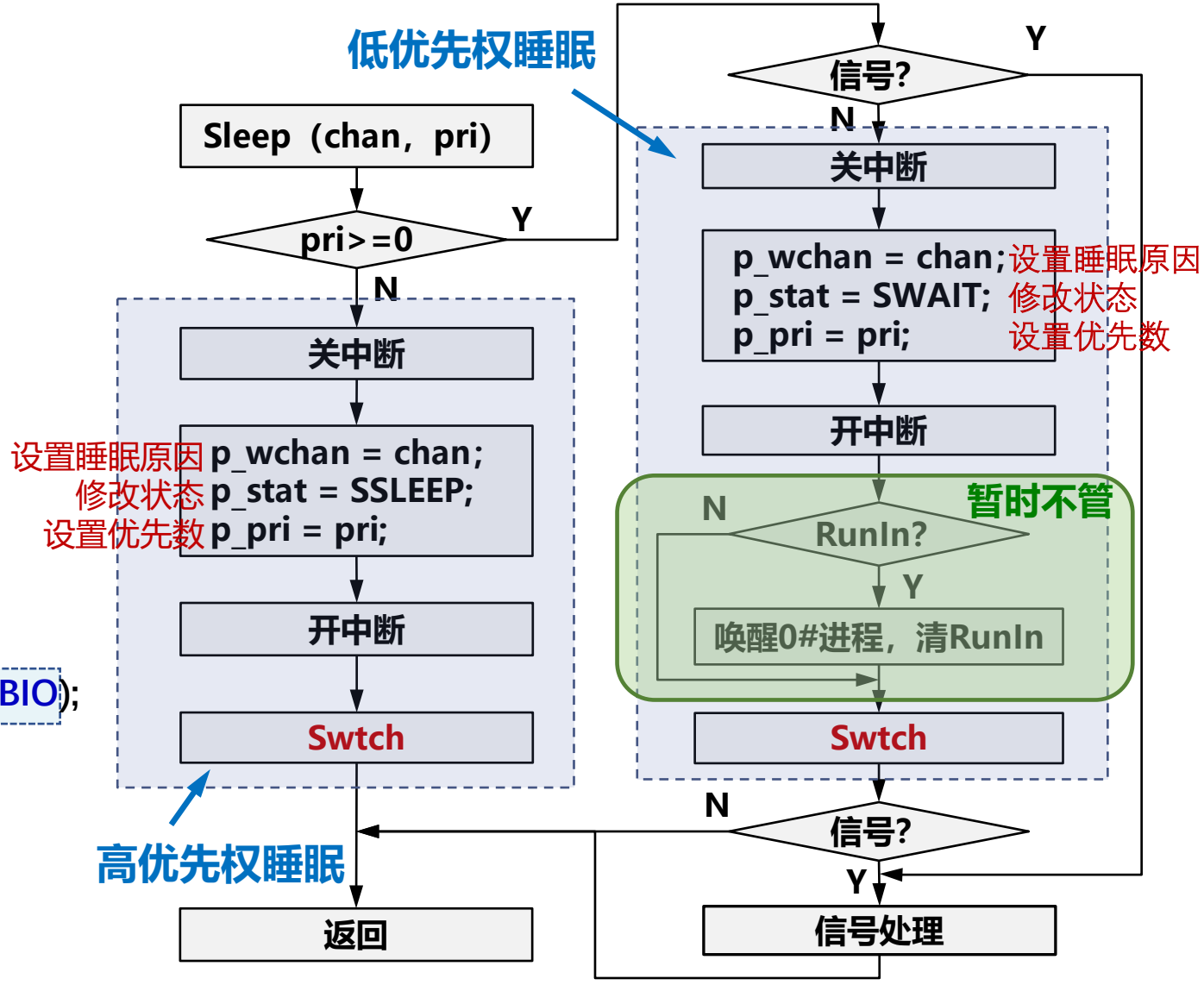
```
static const int PSWP = -100;
static const int PINOD = -90;
static const int PRIBIO = -50;
static const int EXPRI = -1;
static const int PPIPE = 1;
static const int TTIPRI = 10;
static const int TTOPRI = 20;
static const int PWAIT = 40;
static const int PSLEP = 90;
static const int PUSER = 100;
```

e.g.

```
Sleep((unsigned long)bp, ProcessManager::PRIBIO);
```

睡眠原因（一个内存地址）

优先数：-50



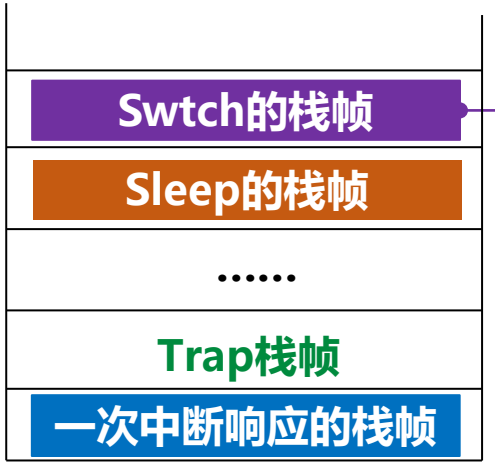
UNIX进程的睡眠与唤醒

非抢占式 下台1 上台1 抢占式 下台2 上台2

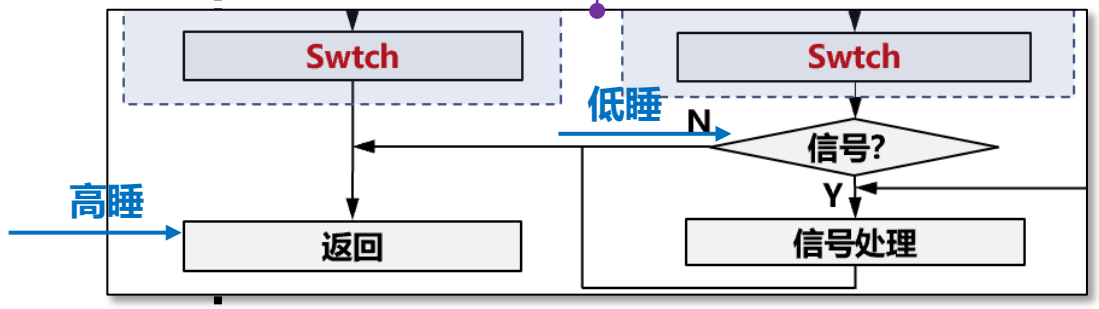
	执行	睡眠	就绪	执行	就绪	执行
p_stat	SRUN	SSLEEP/SWAIT	SRUN	SRUN	SRUN	SRUN
p_flag	SLOAD	SLOAD	SLOAD	SLOAD	SLOAD	SLOAD
p_pri	>=100	<100	<100	<100	>=100	>=100
p_wchan	=0	=&内存变量	=0	=0	=0	=0

Sleep修改 WakeUpAll修改 Trap末尾修改

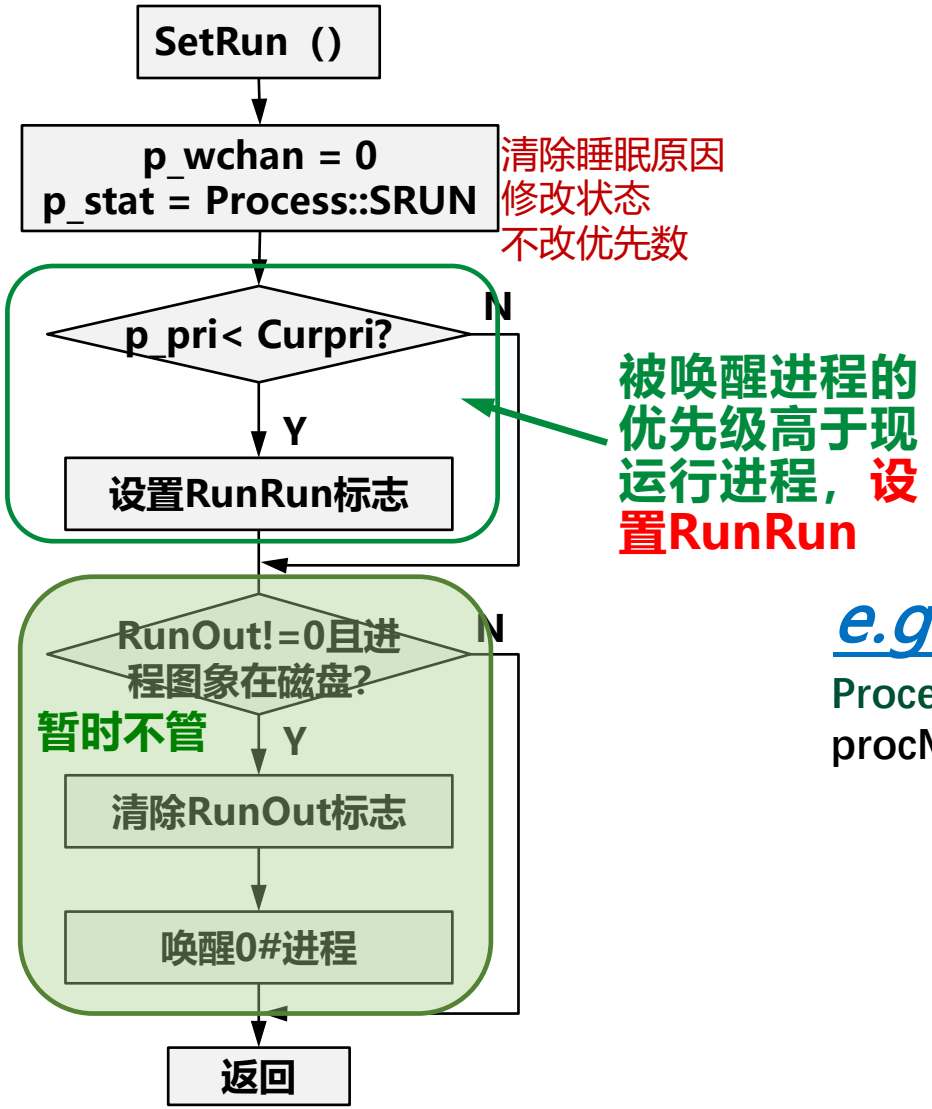
Swtch下台 Swtch上台 Swtch下台 Swtch上台



Swtch栈帧中的返回地址



UNIX进程的睡眠与唤醒



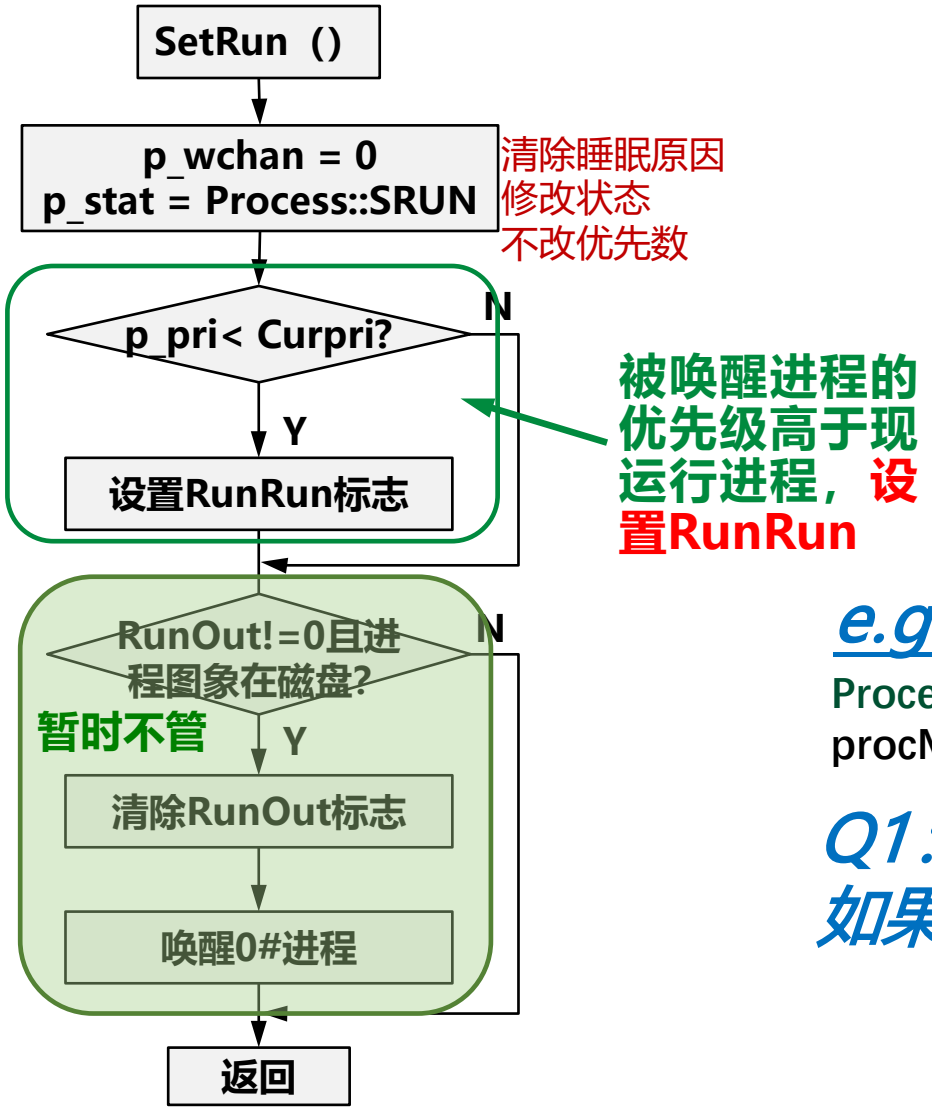

```
void ProcessManager::WakeUpAll(unsigned long chan)
{
  for(int i = 0; i < ProcessManager::NPROC; i++)
  {
    if( this->process[i].IsSleepOn(chan) )
    {
      this->process[i].SetRun();
    }
  }
}
```

1. 以睡眠原因为参数
2. 核对所有的进程，看谁睡在这个原因上?
3. 调用SetRun将其唤醒

e.g.

```
ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
procMgr.WakeUpAll((unsigned long)bp);
```

UNIX进程的睡眠与唤醒




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

1. 以睡眠原因为参数
2. 核对所有的进程，看谁睡在这个原因上?
3. 调用SetRun将其唤醒

e.g.

```
ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
procMgr.WakeUpAll((unsigned long)bp);
```

Q1:

如果睡在chan这个原因上的进程不止一个呢?

UNIX进程的睡眠与唤醒

e.g. UNIX系统中提供了一个定时睡眠的系统调用

```
main()
{
    ……;
    sleep(2); /* 睡两秒后再执行
    ……;
}
```

Time::tout：闹钟
设置为距离当前时间最近的需要唤醒进程的时间点

Time::time
当前系统时间

UNIX进程的睡眠与唤醒

进程的唤醒

e.g. UNIX系统中提供了一个定时睡眠的系统调用

```
main()
{
    ……;
    sleep(2); /* 睡两秒后再执行
    ……;
}
```

Time::tout：闹钟
设置为距离当前时间最近的需要唤醒进程的时间点

Time::time
当前系统时间

```
int SystemCall::Sys_Sslep()
{
    User& u = Kernel::Instance().GetUser();
    X86Assembly::CLI();



    unsigned int wakeTime = Time::time + u.u_arg[0];   /* sleep(second) */

    while( wakeTime > Time::time )   /* 现运行进程还没到该醒的时候 */
    {
        if ( Time::tout <= Time::time || Time::tout > wakeTime )
        /*  “闹钟已经响过了”或者“闹钟时间比现运行进程需要被叫醒的时间长”  */
        {
            Time::tout = wakeTime;    /* 将闹钟设置为现运行进程需要被唤醒的时间 */
         }
        /* 现运行进程入睡，原因为Time::tout的地址，优先级为90 */
        u.u_procp->Sleep((unsigned long)&Time::tout, ProcessManager::PSLEP);
     }
     X86Assembly::STI();
     return 0;/* GCC likes it ! */
}
```

UNIX进程的睡眠与唤醒

e.g. UNIX系统中提供了一个定时睡眠的系统调用

```
main()
{
    ……;
    sleep(2); /* 睡两秒后再执行
    ……;
}
```

wakeTime在哪儿?

Sys_Sslep的局部变量，在进程核心栈的栈帧里

Time::time在哪儿?

Time声明的static成员，在内核区

2在哪儿?

EBX->核心栈->u.u_arg[0]

```
int SystemCall::Sys_Sslep()
{
    User& u = Kernel::Instance().GetUser();
    X86Assembly::CLI();

    进程需要被唤醒的时间 = 当前的系统时间 + 进程需要入睡的时间

    unsigned int wakeTime = Time::time + u.u_arg[0];   /* sleep(second) */

    while( wakeTime > Time::time )   /* 现运行进程还没到该醒的时候 */
    {
        if ( Time::tout <= Time::time || Time::tout > wakeTime )
        /*  “闹钟已经响过了”或者“闹钟时间比现运行进程需要被叫醒的时间长”  */
        {
            Time::tout = wakeTime;    /* 将闹钟设置为现运行进程需要被唤醒的时间 */
        }
        /* 现运行进程入睡，原因为Time::tout的地址，优先级为90 */
        u.u_procp->Sleep((unsigned long)&Time::tout, ProcessManager::PSLEP);
    }
    X86Assembly::STI();
    return 0;/* GCC likes it ! */
}
```

UNIX进程的睡眠与唤醒

e.g. UNIX系统中提供了一个定时睡眠的系统调用

```
main()
{
    ……;
    sleep(2); /* 睡两秒后再执行
    ……;
}
```

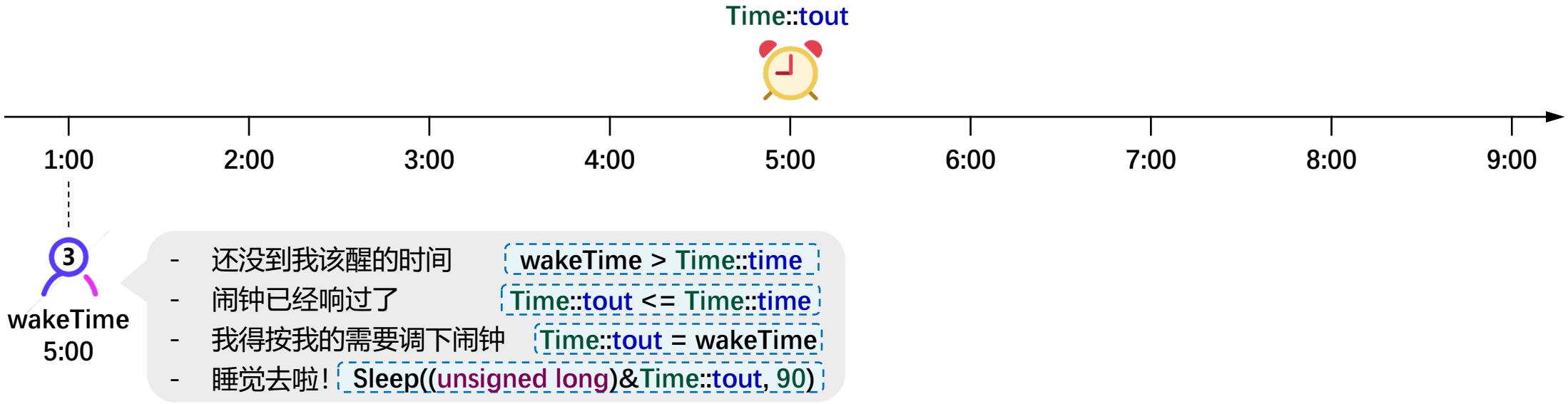
1. 该醒的进程 wakeTime == Time::time：上台后退出while循环，从Sys_Sslep返回；
2. 不该醒的进程 wakeTime > Time::time：上台后，判断是否需要修改闹钟，然后再睡。

```
int SystemCall::Sys_Sslep()
{
    User& u = Kernel::Instance().GetUser();
    X86Assembly::CLI();

    进程需要被唤醒的时间 = 当前的系统时间 + 进程需要入睡的时间

    unsigned int wakeTime = Time::time + u.u_arg[0];   /* sleep(second) */

    while( wakeTime > Time::time )   /* 现运行进程还没到该醒的时候 */
    {
        if ( Time::tout <= Time::time || Time::tout > wakeTime )
        /* “闹钟已经响过了”或者“闹钟时间晚于现运行进程需要被叫醒的时间” */
        {
            Time::tout = wakeTime;   /* 将闹钟设置为现运行进程需要被唤醒的时间 */
        }
        /* 现运行进程入睡，原因为Time::tout的地址，优先级为90 */
        u.u_procp->Sleep((unsigned long)&Time::tout, ProcessManager::PSLEP);
    }
```

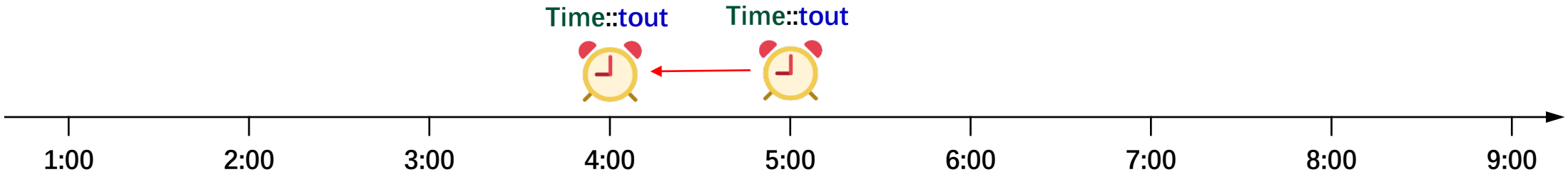
整数秒的时钟中断里：

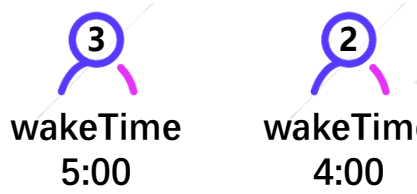
```
if ( Time::time == Time::tout ) /* 如果闹钟时间到 */
{
    procMgr.WakeUpAll((unsigned long)&Time::tout);  /* 唤醒所有睡在Time::tout的地址上的进程 */
}
```

UNIX进程的睡眠与唤醒



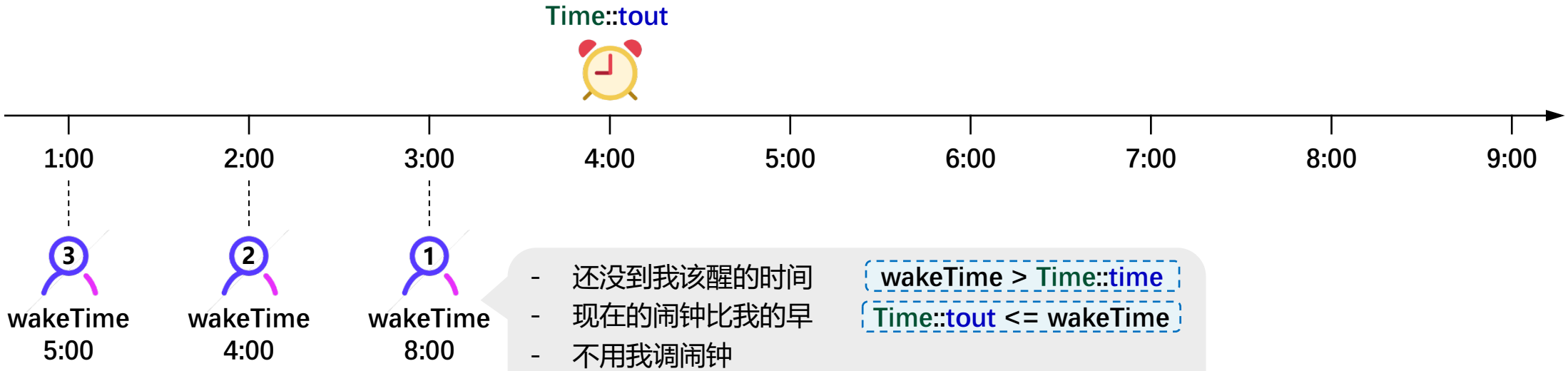
UNIX进程的睡眠与唤醒



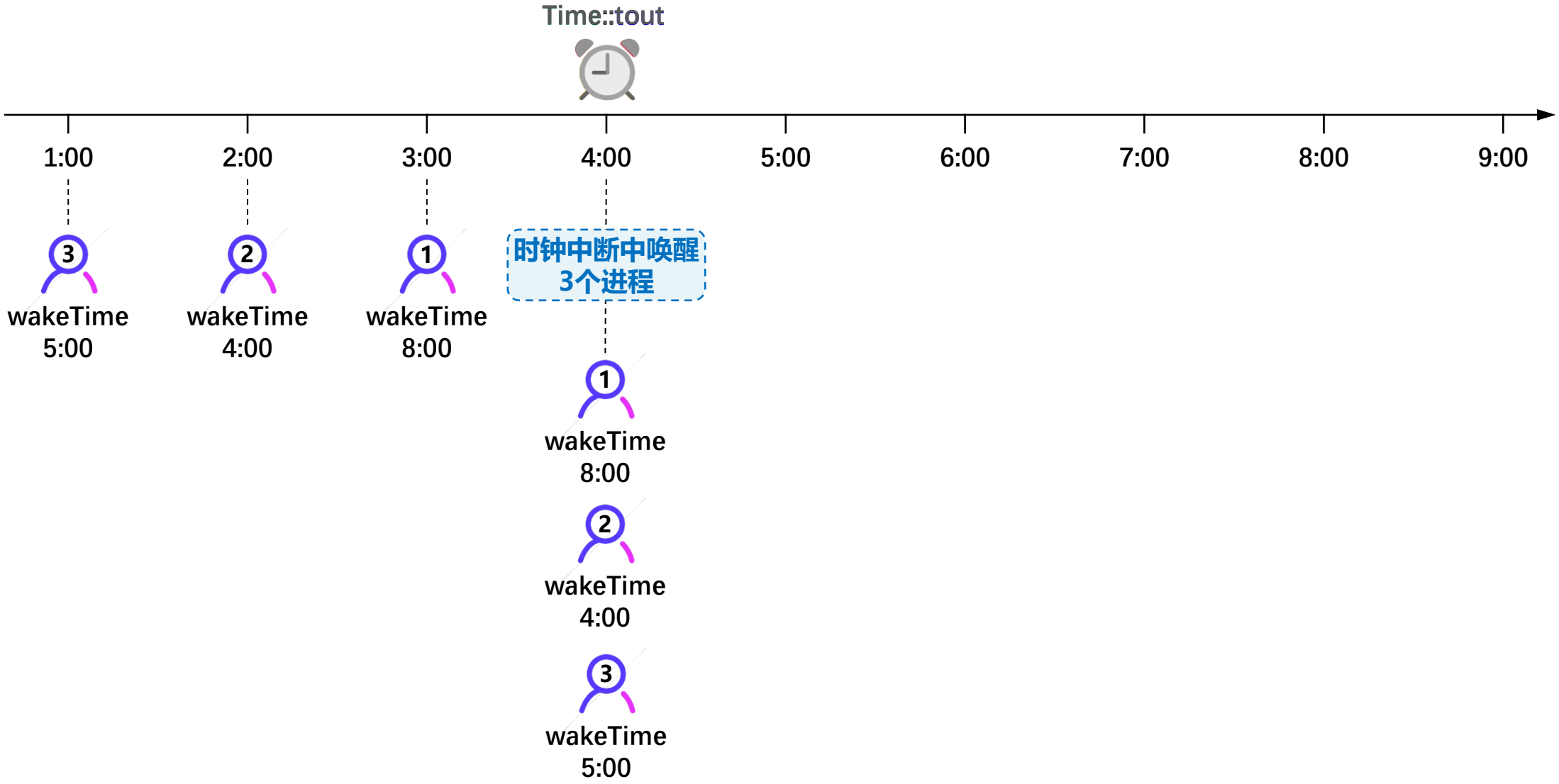


- 还没到我该醒的时间 `wakeTime > Time::time`
- 现在的闹钟太晚了 `Time::tout > wakeTime`
- 我得按我的需要调下闹钟 `Time::tout = wakeTime`
- 睡觉去啦！`Sleep((unsigned long)&Time::tout, 90)`

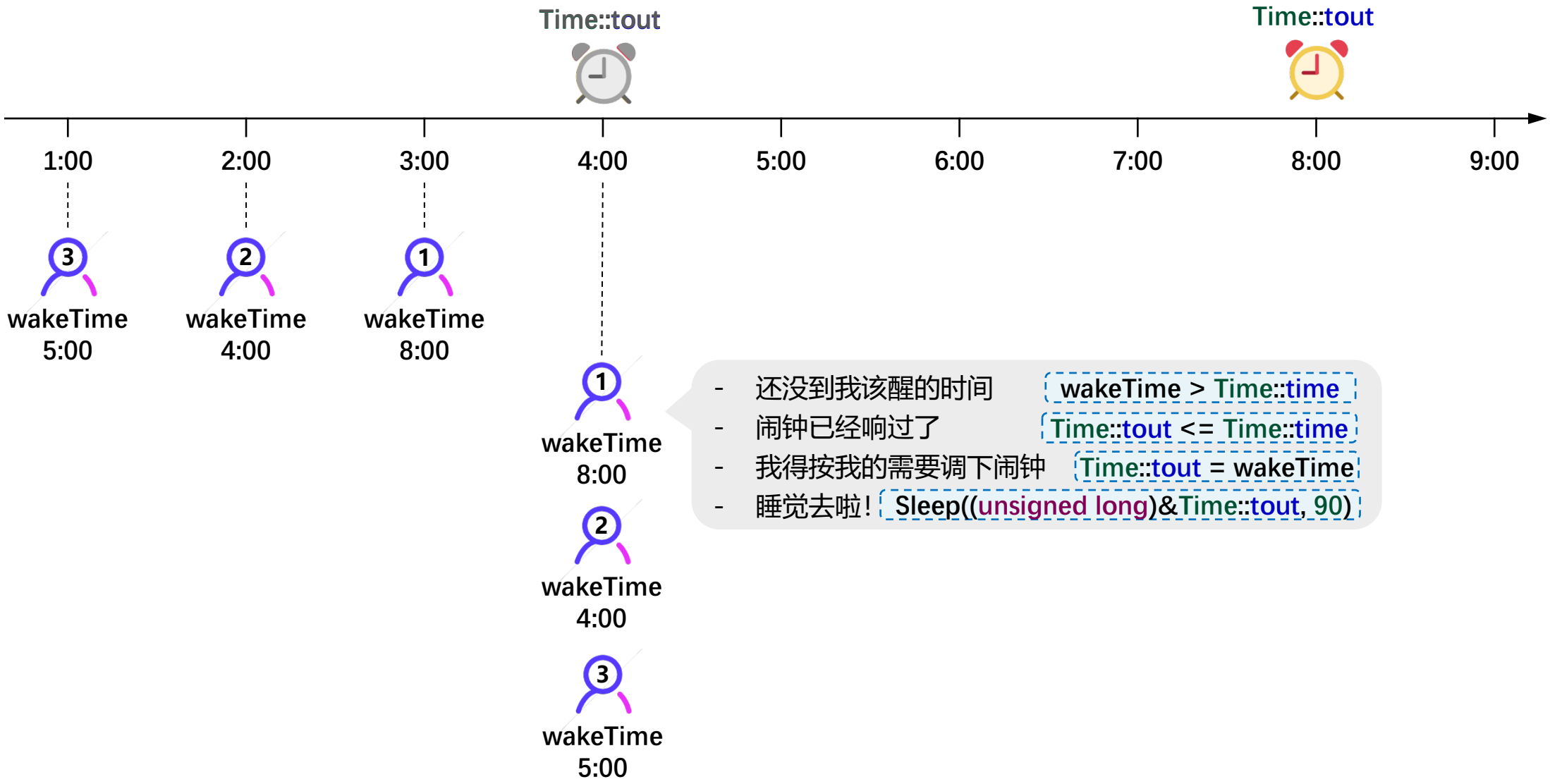
UNIX进程的睡眠与唤醒



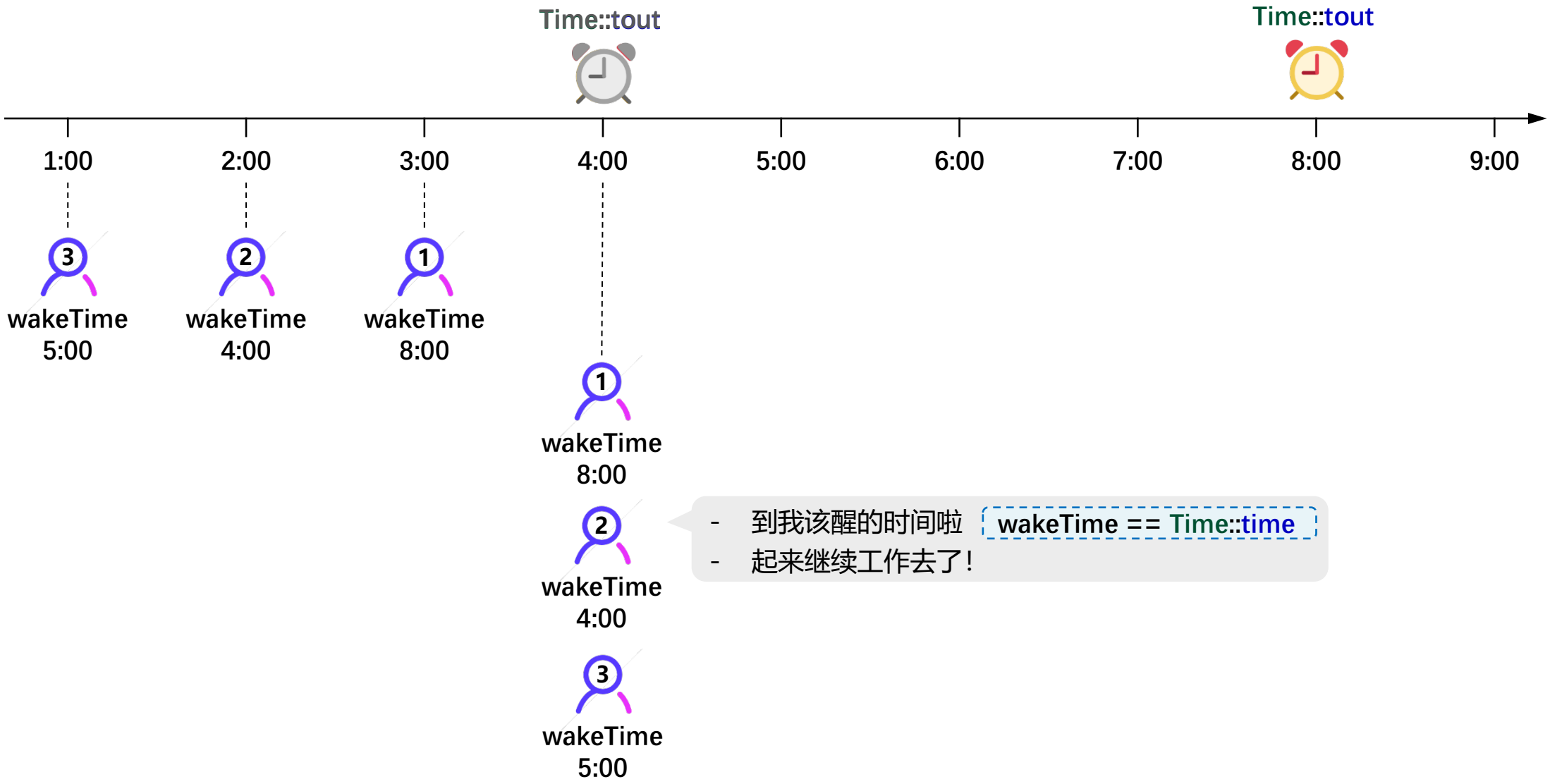
UNIX进程的睡眠与唤醒



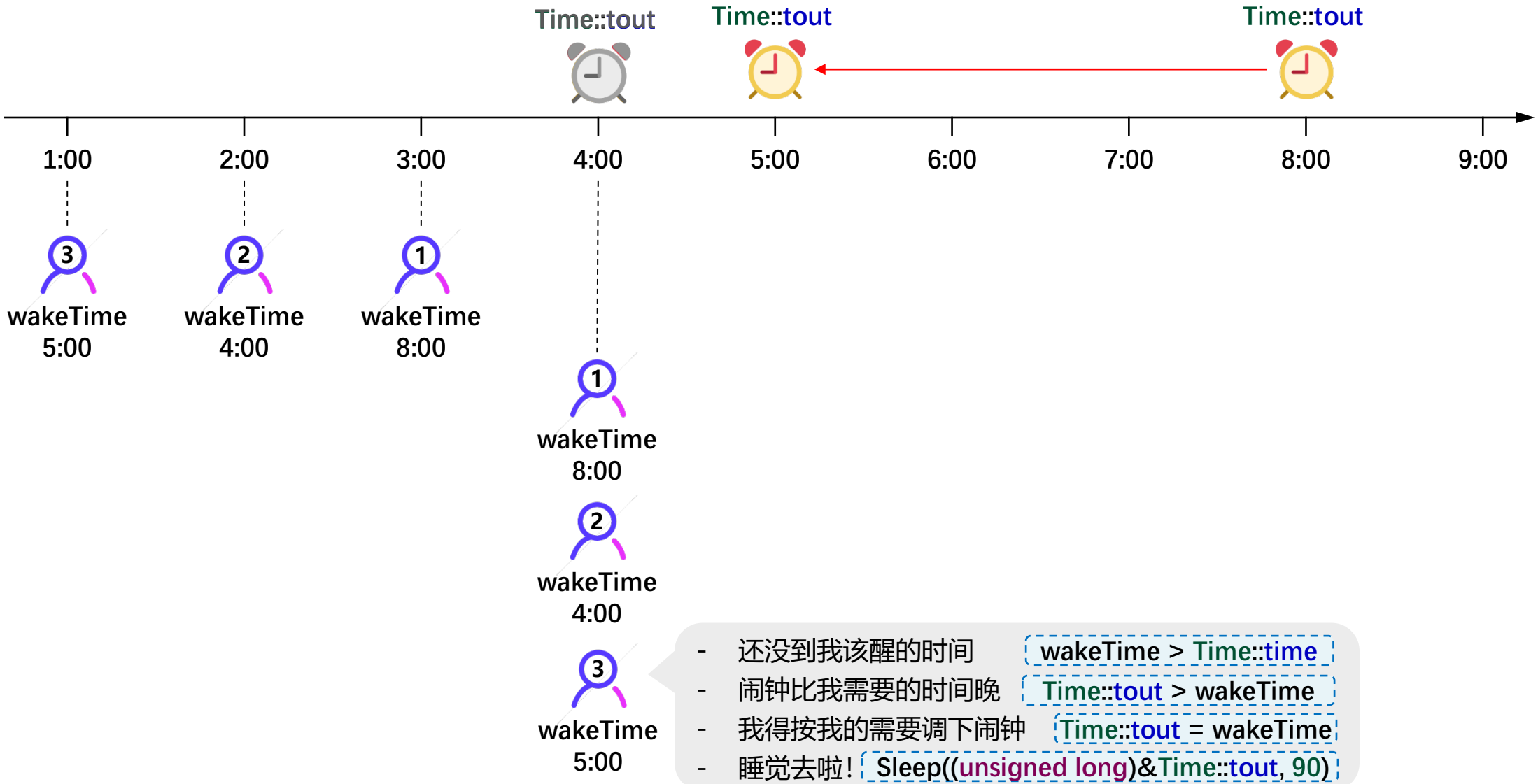
UNIX进程的睡眠与唤醒



UNIX进程的睡眠与唤醒

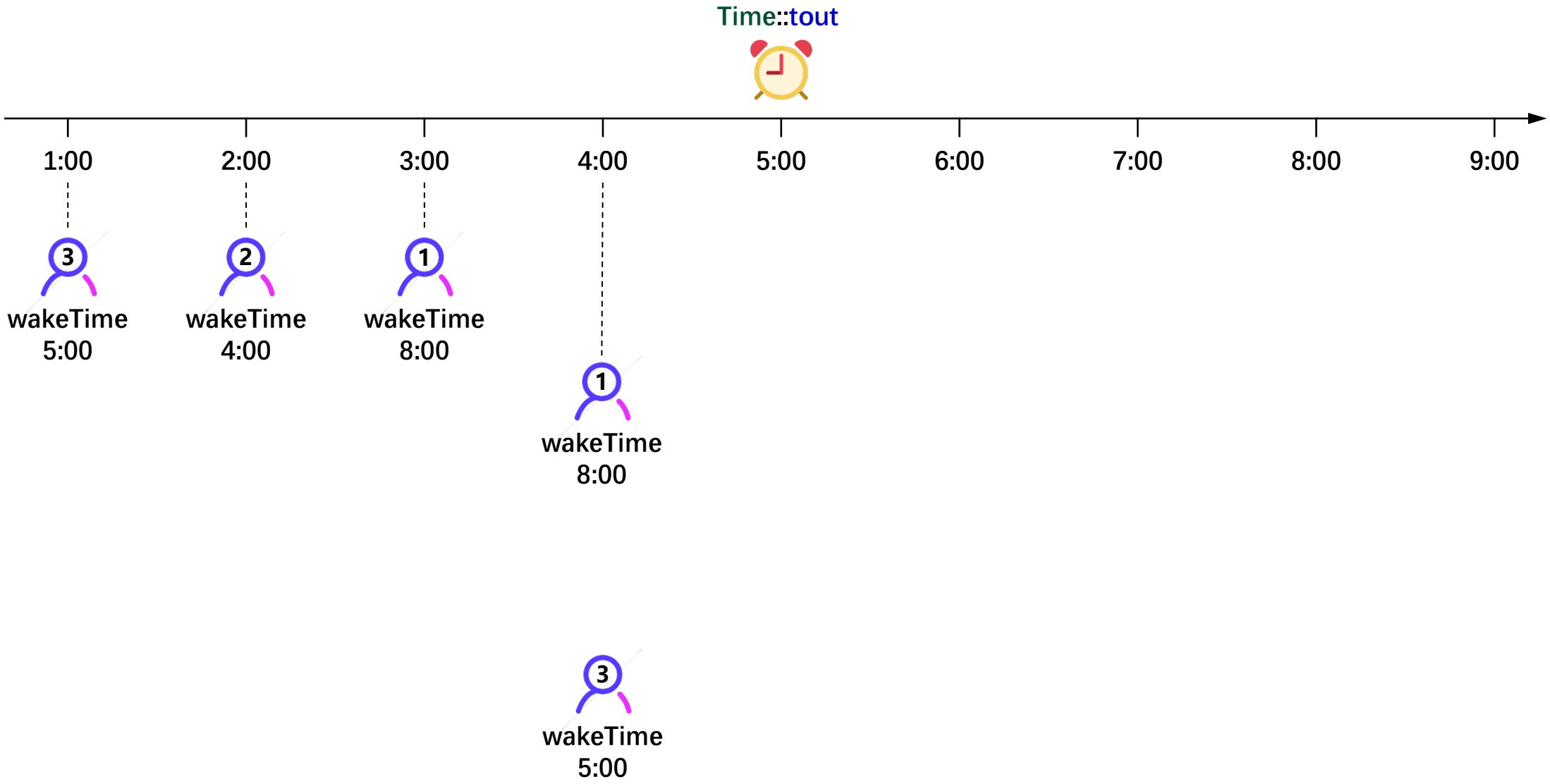


UNIX进程的睡眠与唤醒

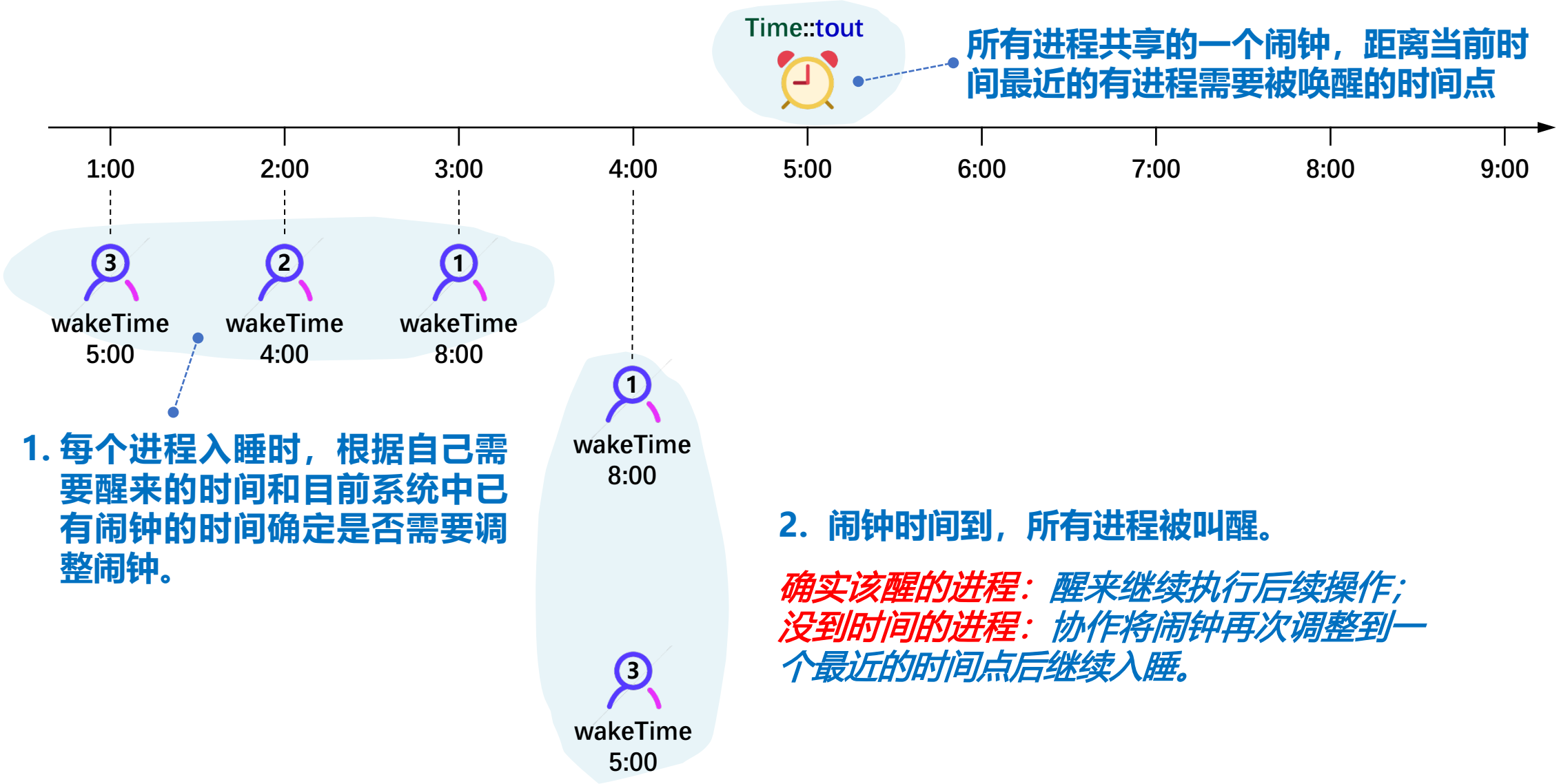


- 还没到我该醒的时间 wakeTime > Time::time
- 闹钟比我需要的时间晚 Time::tout > wakeTime
- 我得按我的需要调下闹钟 Time::tout = wakeTime
- 睡觉去啦！Sleep((unsigned long)&Time::tout, 90)

UNIX进程的睡眠与唤醒



UNIX进程的睡眠与唤醒



1. 每个进程入睡时，根据自己需要醒来的时间和目前系统中已有闹钟的时间确定是否需要调整闹钟。

2. 闹钟时间到，所有进程被叫醒。

确实该醒的进程：醒来继续执行后续操作；
没到时间的进程：协作将闹钟再次调整到一个最近的时间点后继续入睡。

UNIX进程的睡眠与唤醒

e.g. UNIX系统中提供了一个定时睡眠的系统调用

```
main()
{
    ……;
    sleep(2); /* 睡两秒后再执行
    ……;
}
```

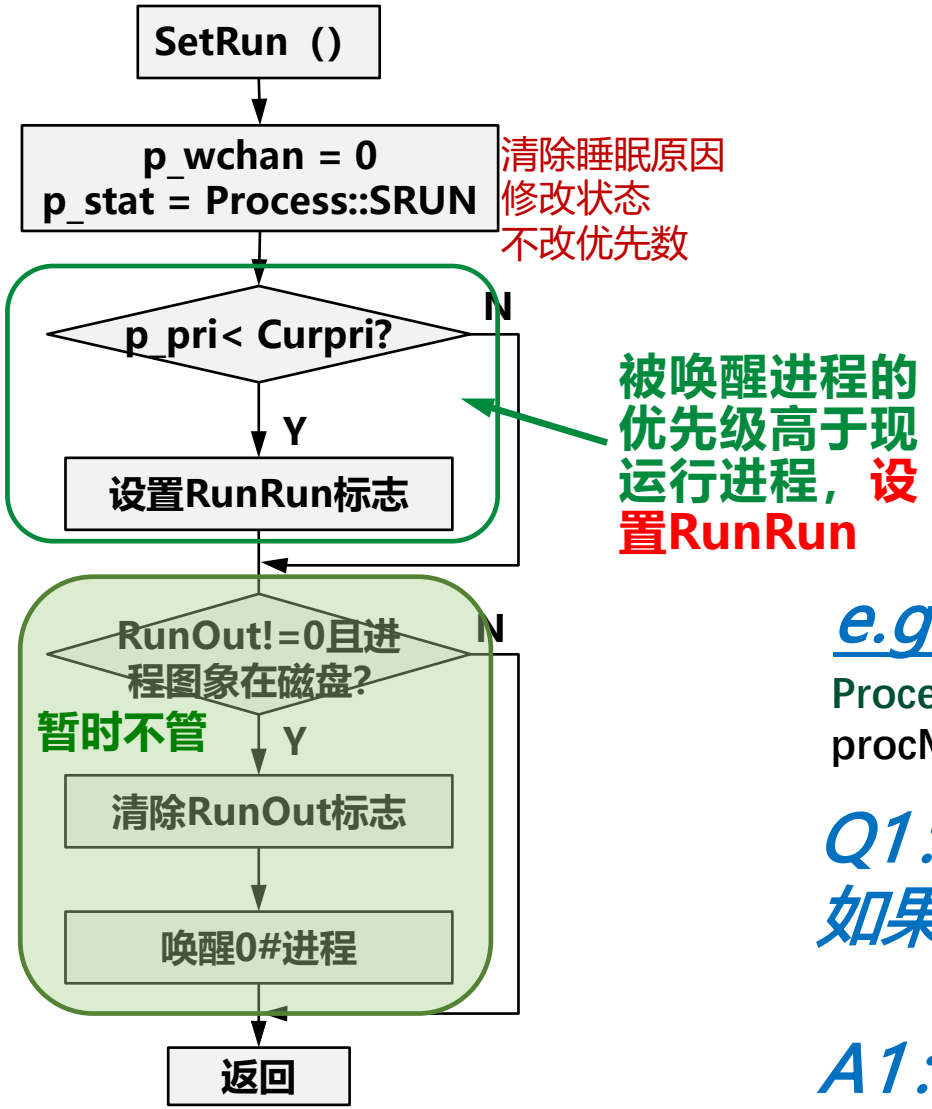
Time::tout：闹钟
设置为距离当前时间最近的需要唤醒进程的时间点

Time::time
当前系统时间

```
int SystemCall::Sys_Sslep()
{
    User& u = Kernel::Instance().GetUser();
    X86Assembly::CLI();   /* 关中断 */

    unsigned int wakeTime = Time::time + u.u_arg[0];   /* sleep(second) */

    while( wakeTime > Time::time )   /* 现运行进程还没到该醒的时候 */
    {
        if ( Time::tout <= Time::time || Time::tout > wakeTime )
        /*  “闹钟已经响过了”或者“闹钟时间比现运行进程需要被叫醒的时间长”  */
        {
            Time::tout = wakeTime;    /* 将闹钟设置为现运行进程需要被唤醒的时间 */
        }
        /* 现运行进程入睡，原因为Time::tout的地址，优先级为90 */
        u.u_procp->Sleep((unsigned long)&Time::tout, ProcessManager::PSLEP);
    }
    X86Assembly::STI();   /* 开中断 */
    return 0;/* GCC likes it ! */
}
```

开关中断的目的?

UNIX进程的睡眠与唤醒




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

1. 以睡眠原因为参数
2. 核对所有的进程，看谁睡在这个原因上?
3. 调用SetRun将其唤醒

e.g.

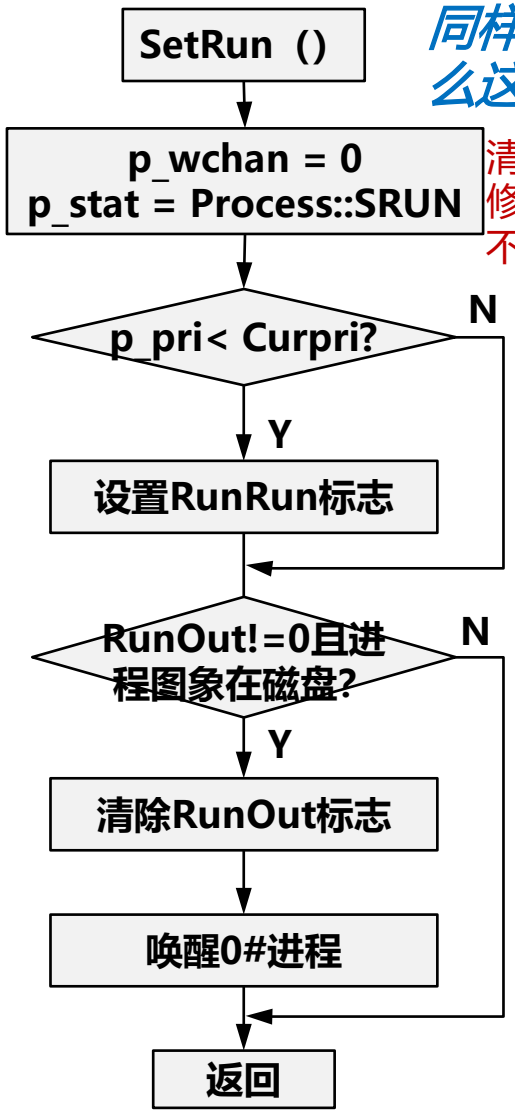
```
ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
procMgr.WakeUpAll((unsigned long)bp);
```

Q1:

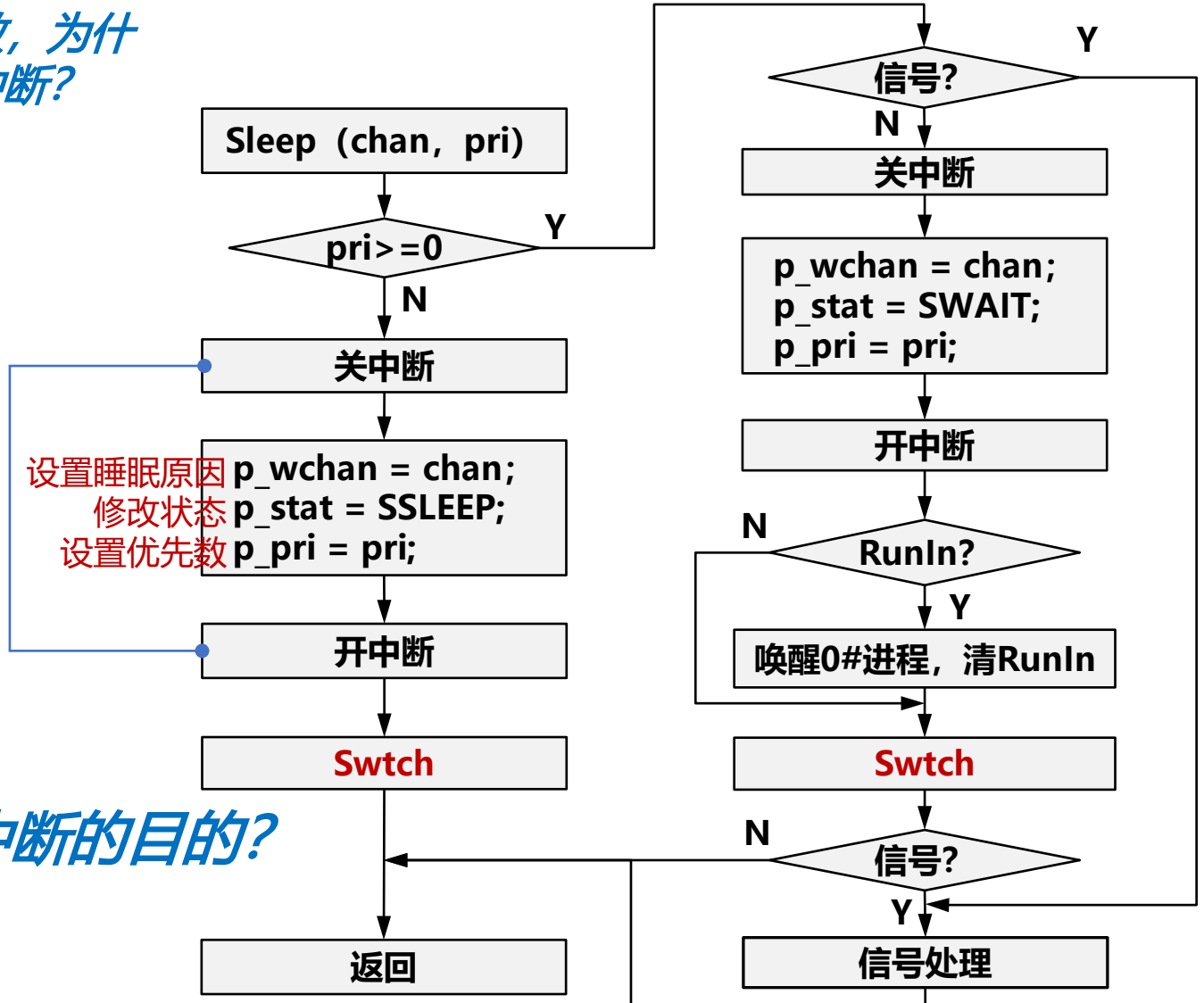
如果睡在chan这个原因上的进程不止一个呢?

A1：全叫起来，不该醒的继续睡。

UNIX进程的睡眠与唤醒

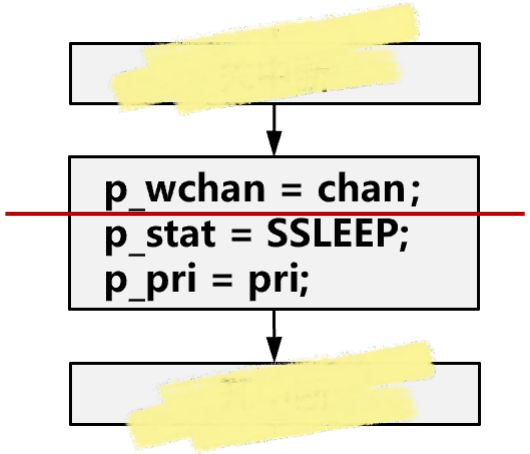


同样修改这几个参数，为什么这里不需要开关中断?



Q2:
这里开关中断的目的?

UNIX进程的睡眠与唤醒



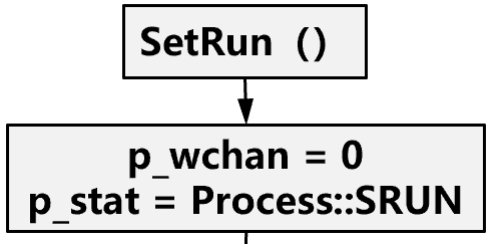
响应中断

而该中断源恰好是当前进程所等待的事件，于是在中断处理中，该进程被唤醒

p_wchan =0
p_stat = process::SRUN;

中断返回后，p_stat = SSLEEP

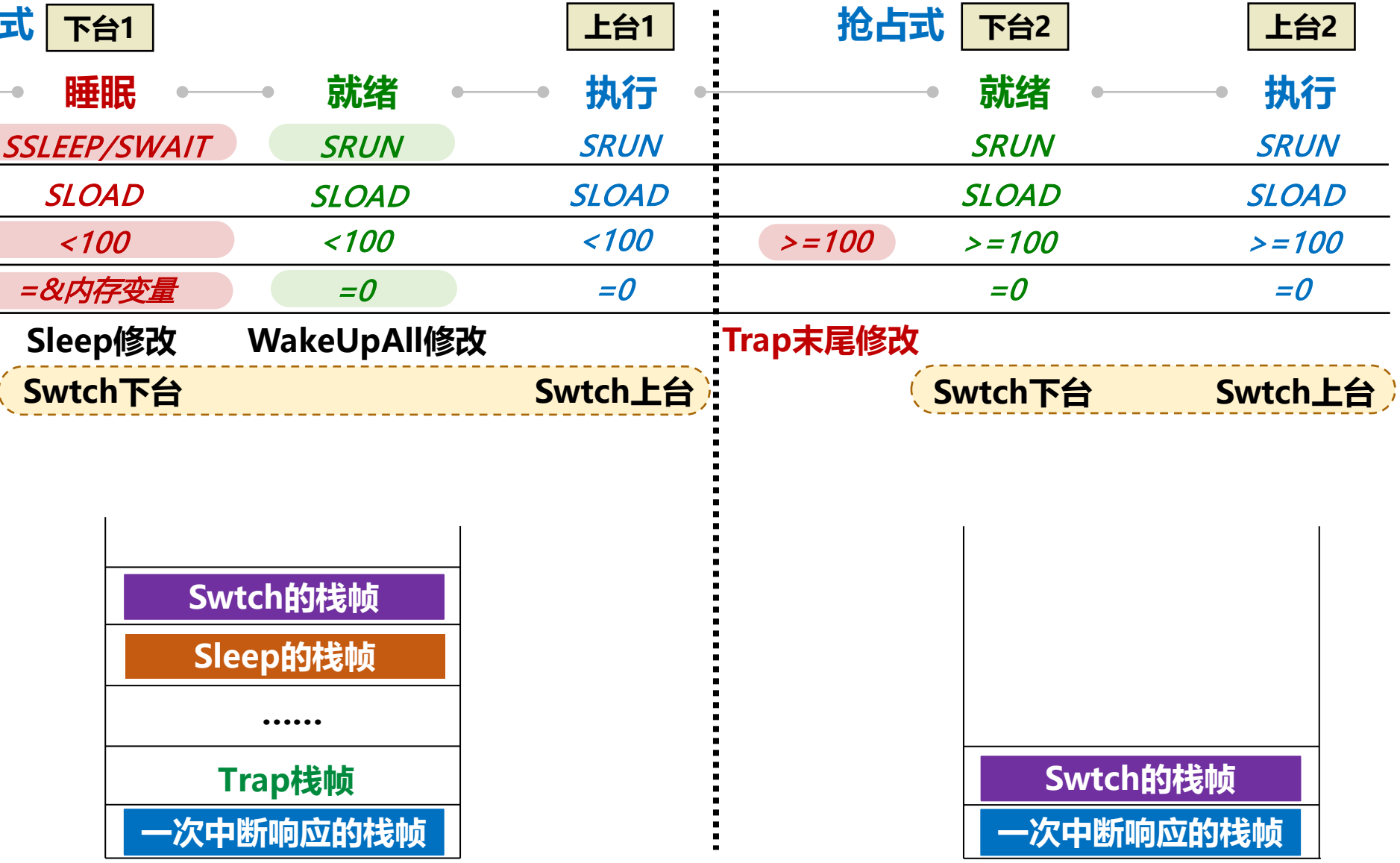
值不一致!



而这里被SetRun的进程不是现运行进程，没有上台执行，不可能发生几个参数不一致的情况。

WakeUpAll程序中永远不可能唤醒了！

UNIX进程的上台与下台



UNIX进程的上台与下台

ProcessManager::Swtch

保存现场 — 保存什么?

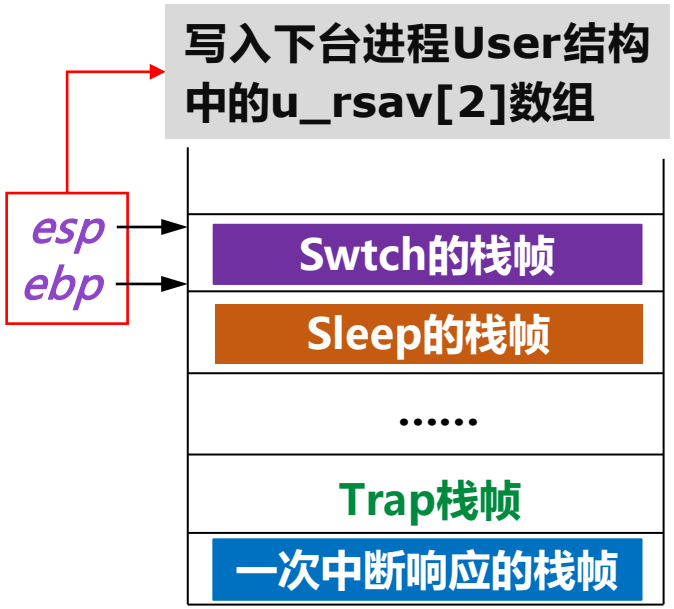
选新进程

恢复现场

return

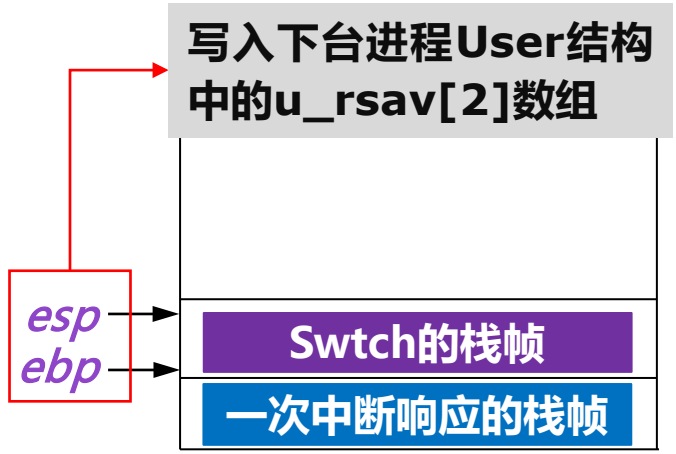
非抢占式

p_stat=SSLEEP/SWAIT, p_pri<100, p_wchan ≠ 0



抢占式

p_stat=SRUN, p_pri >=100 p_wchan = 0



UNIX进程的上台与下台

ProcessManager::Swtch

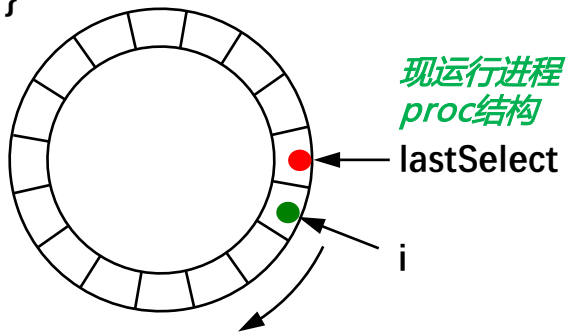
保存现场 — 保存什么?

选新进程 — 1. 怎么选?

恢复现场

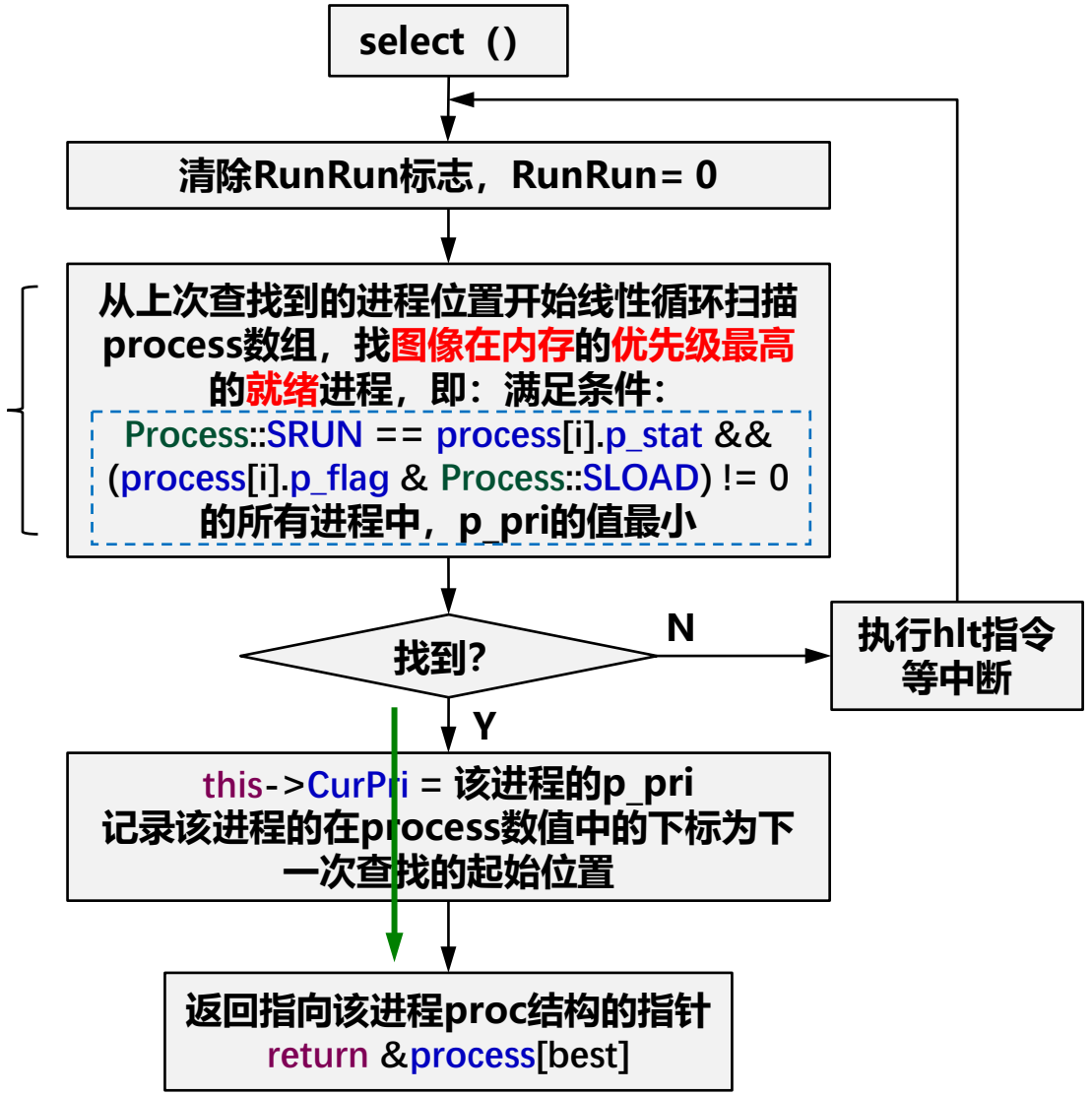
return

```
int priority = 256;
if ( process[i].p_pri < priority )
{
    best = i;
    priority = process[i].p_pri;
}
```



从上一次被选中进程的下一个开始循环扫描，而不是每次从0#进程开始，保证各进程机会均等

e.g.
如果现运行进程是优先级并列高的进程，选择的结果?



UNIX进程的上台与下台

ProcessManager::Swtch

保存现场 — 保存什么?

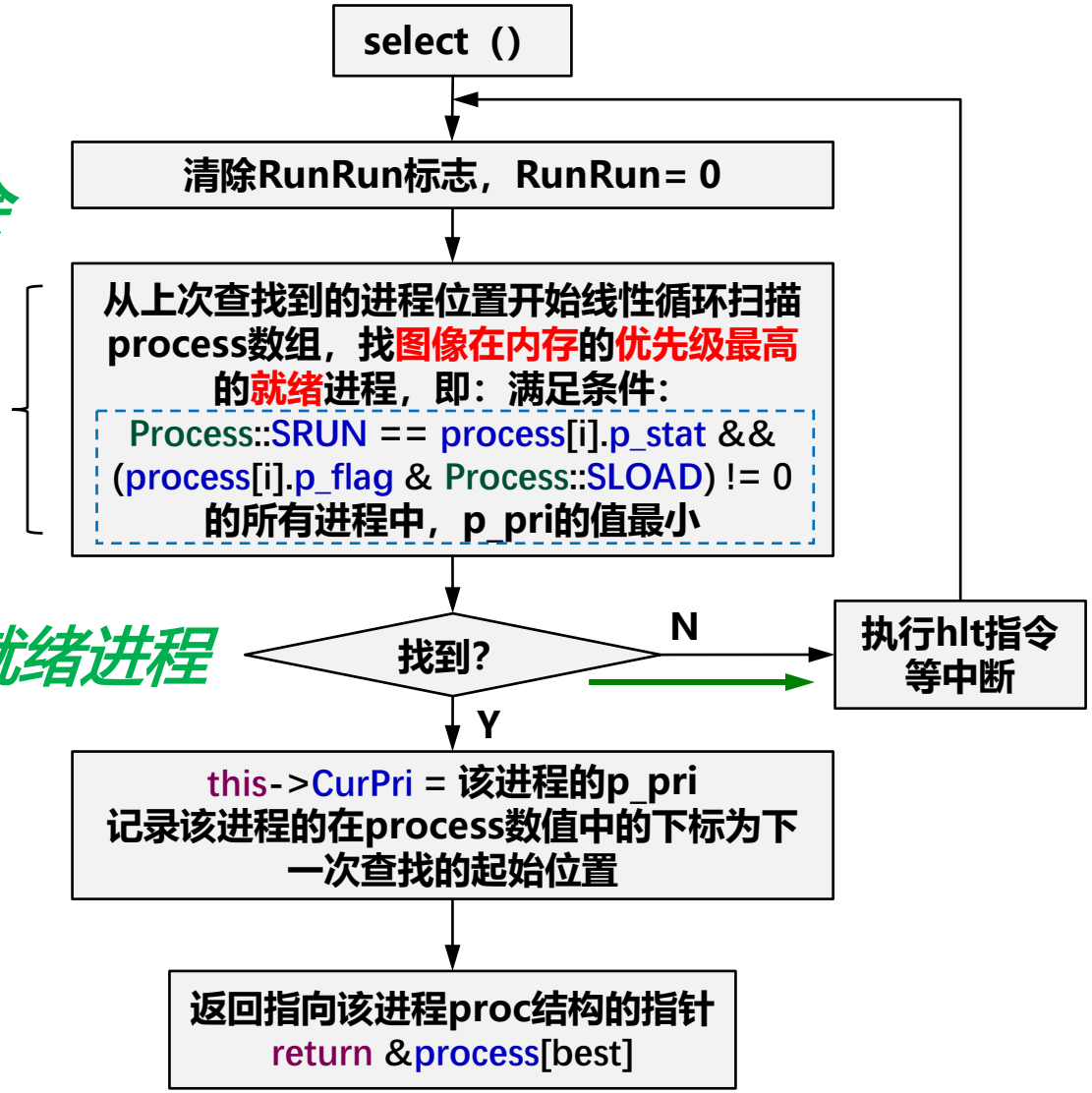
选新进程 — 1. 怎么选?

恢复现场

return

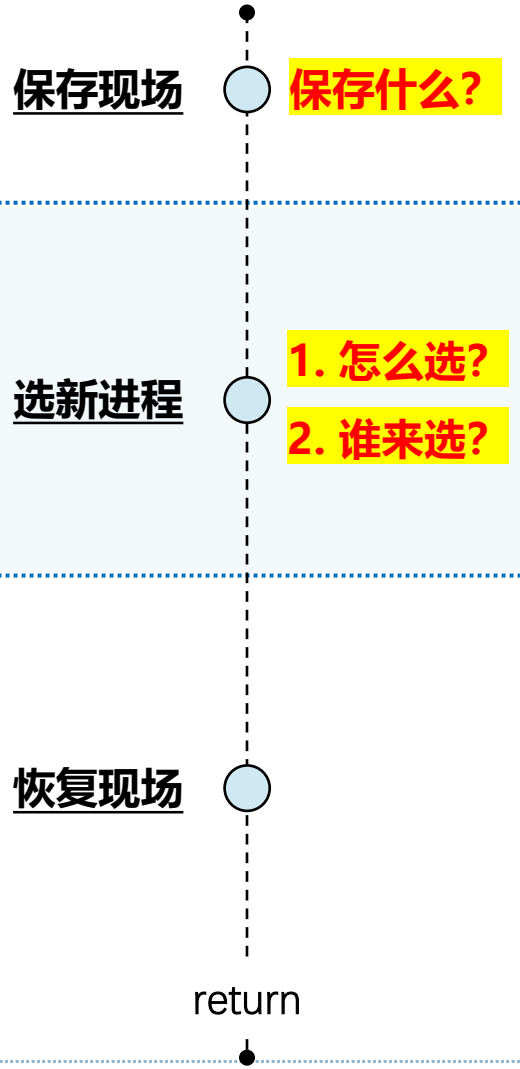
中断中会唤醒进程，中断返回后，这里会找到刚唤醒的进程

没有就绪进程



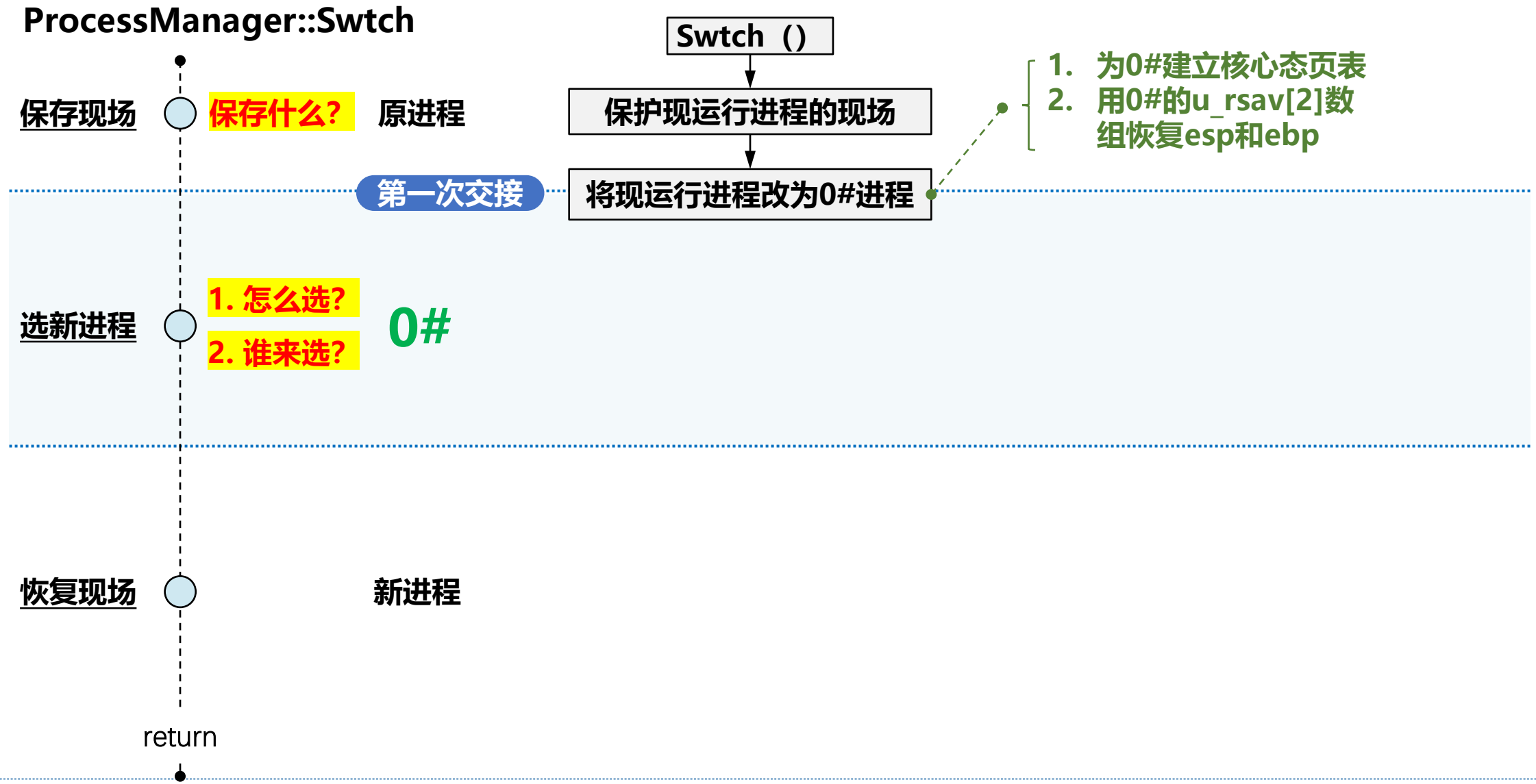
UNIX进程的上台与下台

ProcessManager::Swtch

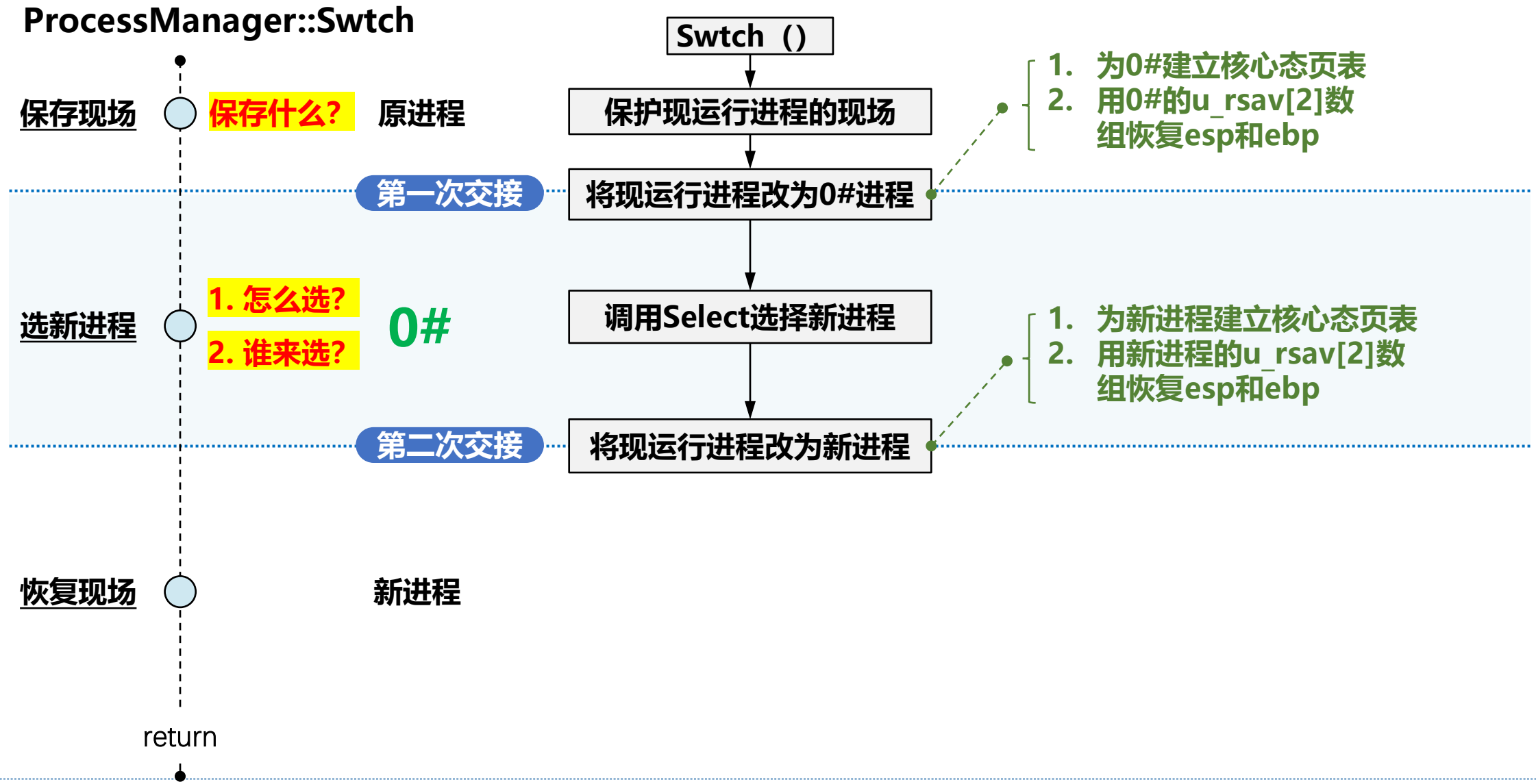


重要的事要由重要的人来完成！！！

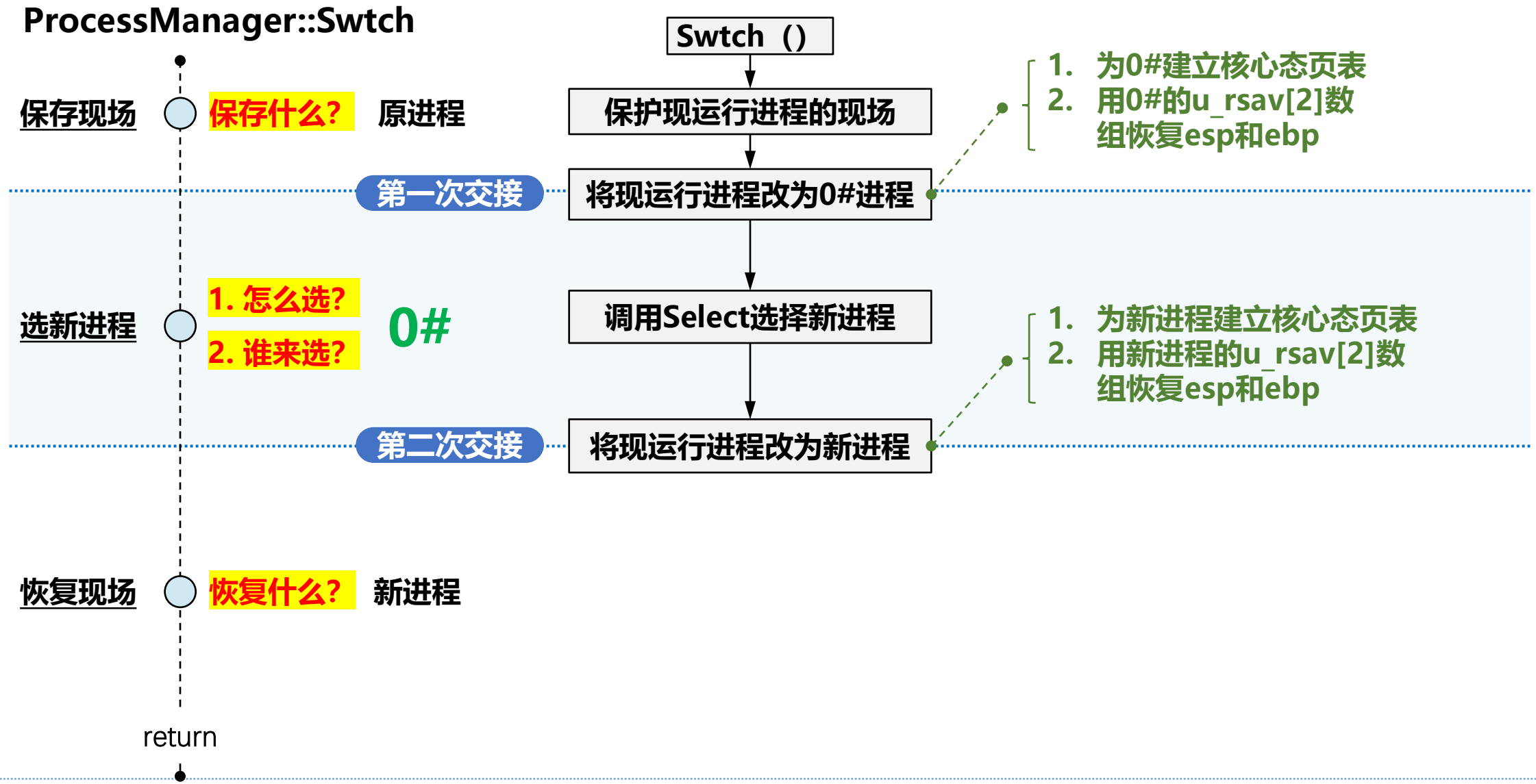
UNIX进程的上台与下台



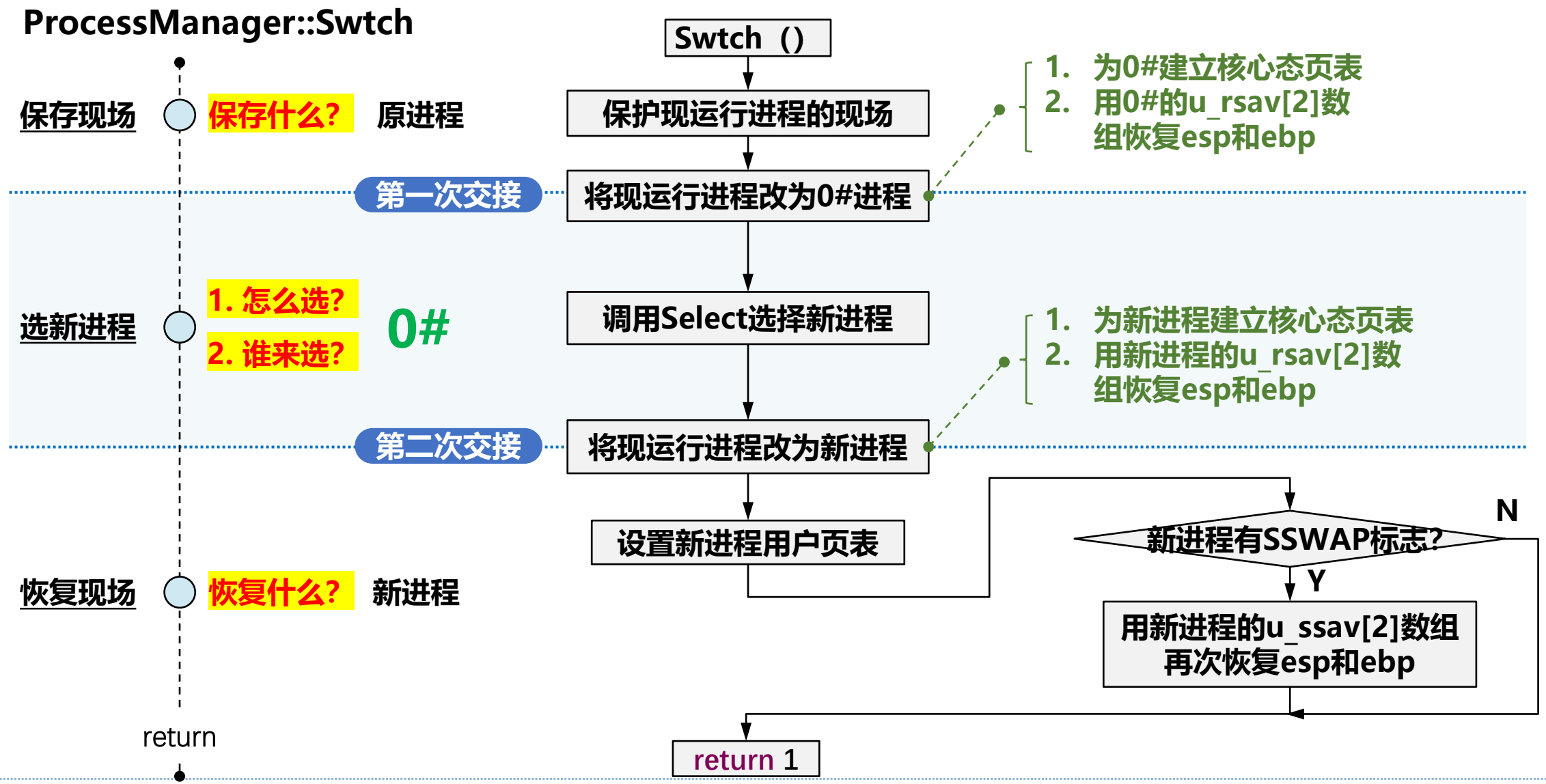
UNIX进程的上台与下台



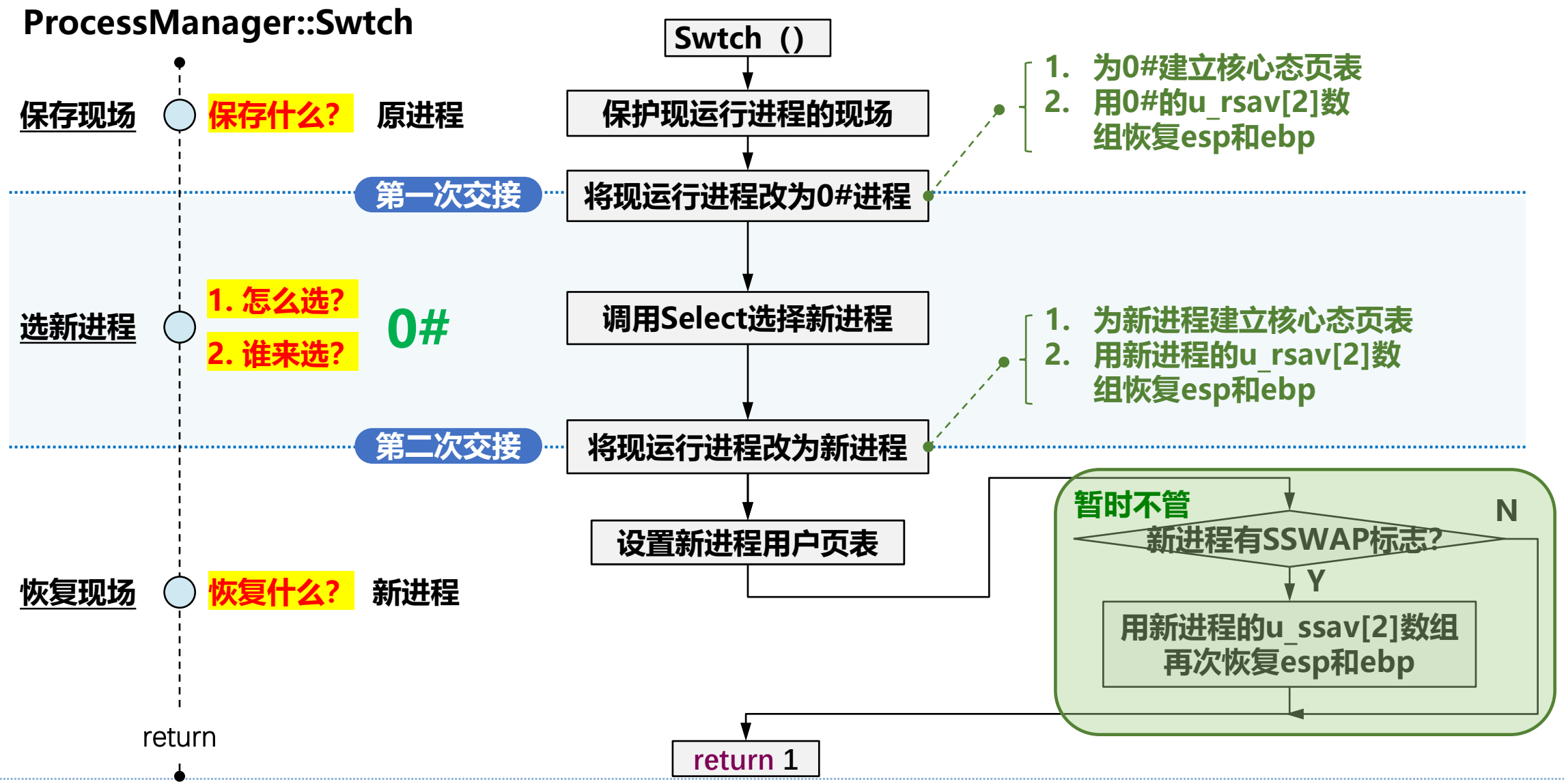
UNIX进程的上台与下台



UNIX进程的上台与下台



UNIX进程的上台与下台



UNIX进程的上台与下台

ProcessManager::Swtch

非抢占式

p_stat=SRUN , p_pri<100, p_wchan = 0

抢占式

p_stat=SRUN, p_pri >=100 p_wchan = 0

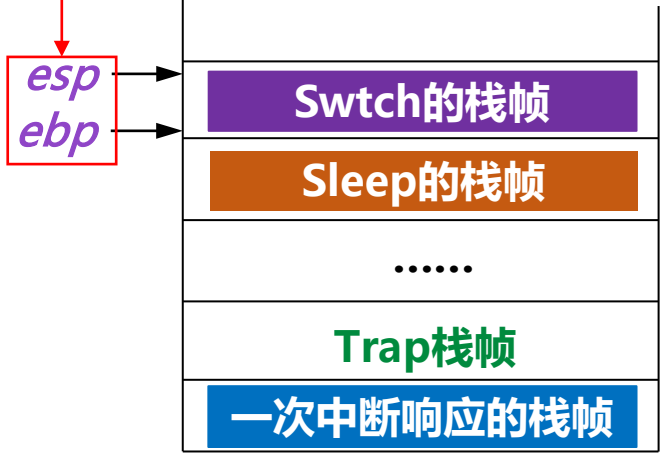
保存现场 — 保存什么？ — 原进程

第一次交接

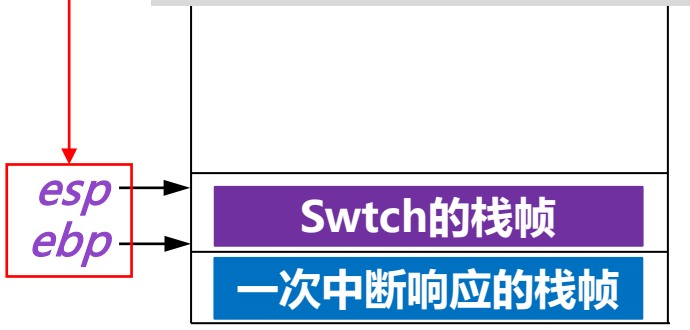
选新进程 — 1. 怎么选？ 2. 谁来选？ — 0#

建立页表，用u_rsav[2]数组中的值恢复

第二次交接



建立页表，用u_rsav[2]数组中的值恢复



恢复现场 — 恢复什么？ — 新进程

return

UNIX进程的上台与下台

ProcessManager::Swtch

保存现场 — 保存什么？ — 原进程

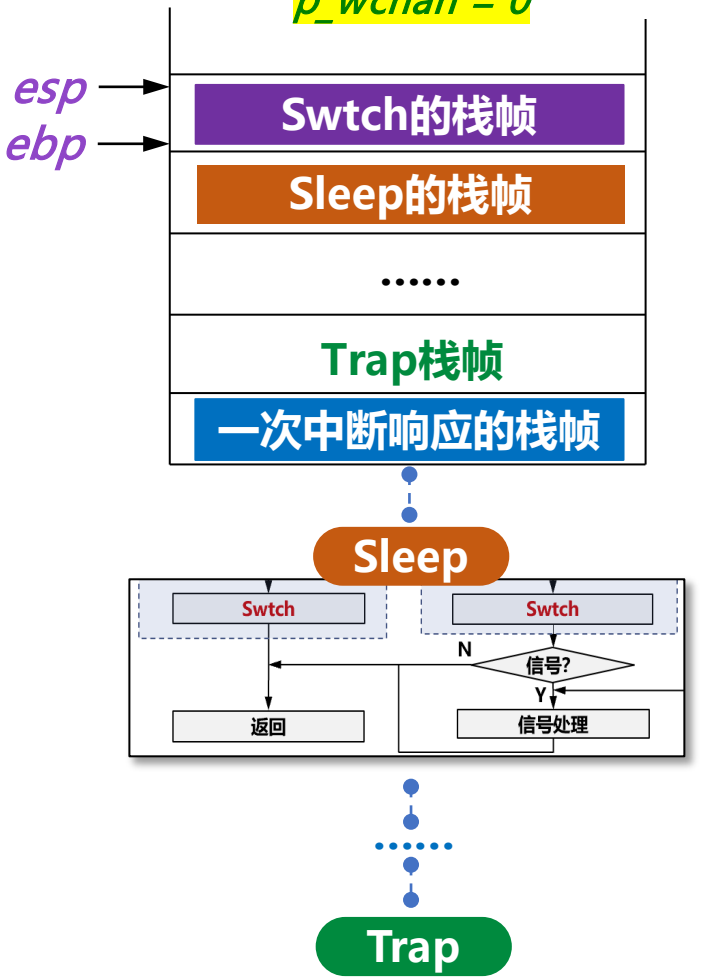
选新进程 — 1. 怎么选？ 2. 谁来选？ — 0#

恢复现场 — 恢复什么？ — 新进程

return 回哪里？

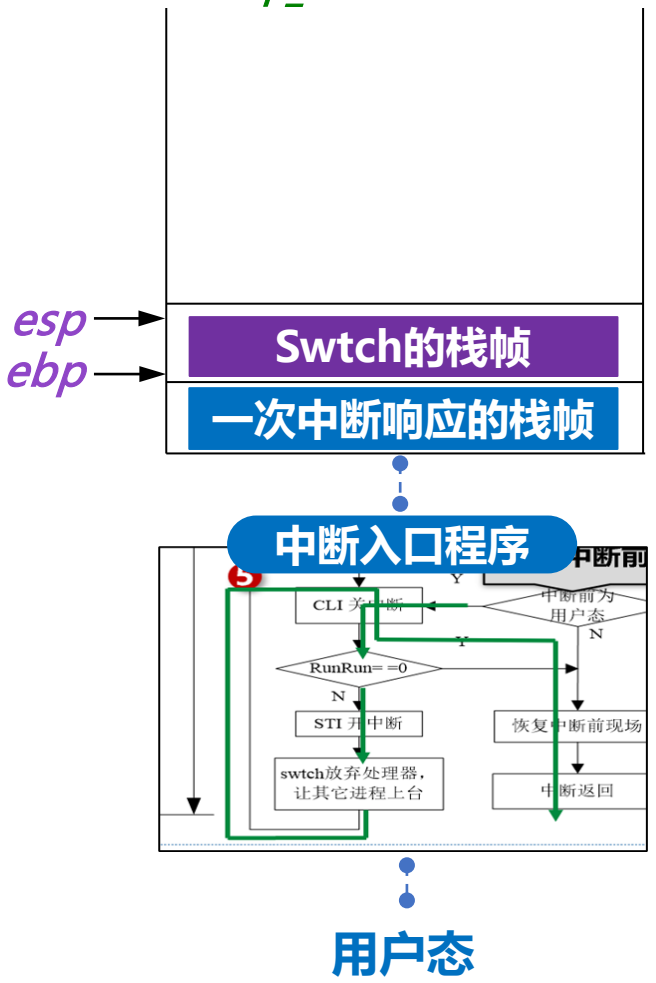
非抢占式

p_stat=SRUN　p_pri >= 100
p_wchan = 0



抢占式

p_stat=SRUN, p_pri >=100
p_wchan = 0



UNIX进程的上台与下台

ProcessManager::Swtch

保存现场 — 保存什么？ — 原进程

第一次交接

选新进程 — 1. 怎么选？ 2. 谁来选？ — 0#

第二次交接

恢复现场 — 恢复什么？ — 新进程

return

为什么三个进程可以合作完成一段代码？

三个进程经地址变换后指向相同的物理地址

EIP →

Page Table 768# （0x201号页框）

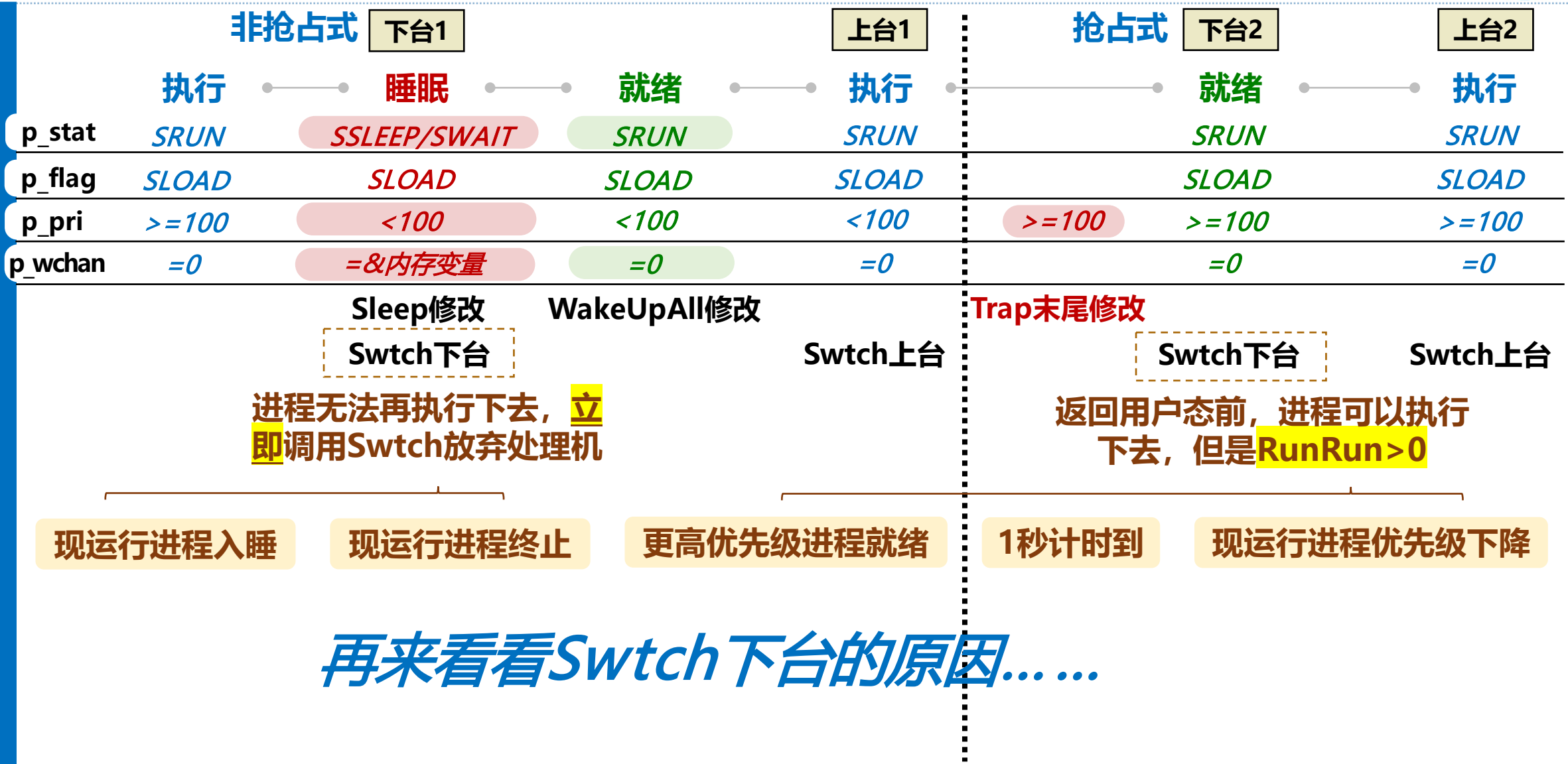
	Page Base Address		s/u	r/w	p
0#	0		0	1	1
1#	1		0	1	1
	.....				
1022#	1022		0	1	1
1023#	p_addr>>12		0	1	1

ESP, EBP →

三个进程经地址变换后指向不同的物理地址

交接棒的过程中，只改变了201#页框的最后一个PTE，EIP没变！！！

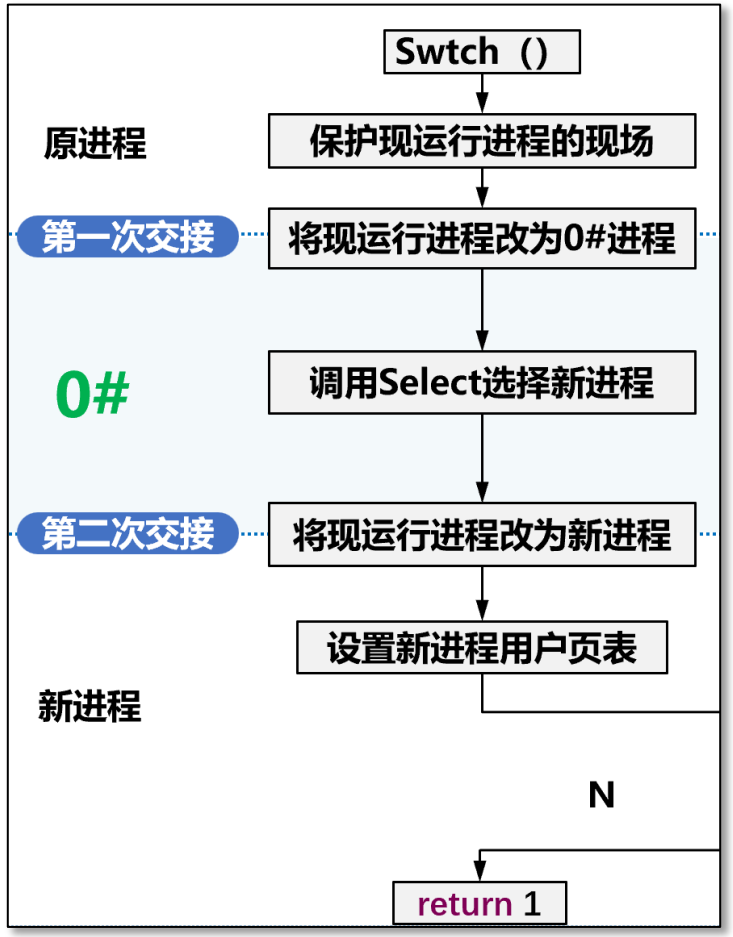
UNIX进程的上台与下台



再来看看Swtch下台的原因……

UNIX进程的上台与下台

关于进程切换的几个问题



现运行进程入睡

更高优先级进程就绪

现运行进程终止

现运行进程已经无法继续在CPU上执行

Swtch的第一棒和最后一棒一定是不同的进程

现运行进程优先级下降

1秒计时到

是时候该Swtch了！！！

但是如果没有别的进程比我优先级高呢?

仍然最高 → Swtch后依然上台

不是/并列最高 → Swtch后下台

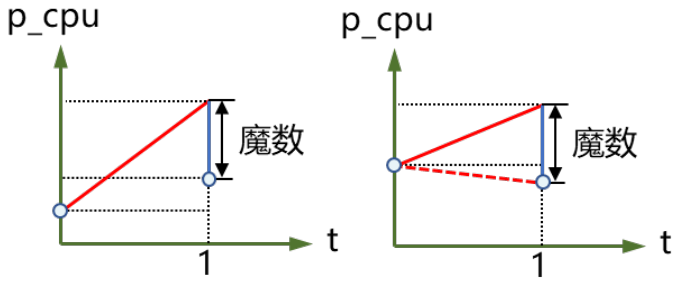
Swtch的第一棒和最后一棒有可能是同一个进程

关于UNIX动态优先权调度算法

①每次时钟脉冲+1
②每秒 - 调度魔数

p_cpu

占用CPU足够多的进程p_cpu ↗
占用CPU不够多的进程p_cpu ↘



总结

关于UNIX动态优先权调度算法

总结

①每次时钟脉冲+1
②每秒 - 调度魔数

p_cpu

占用CPU足够多的进程p_cpu ↗
占用CPU不够多的进程p_cpu ↘

计算 ---------- 三次计算的时机

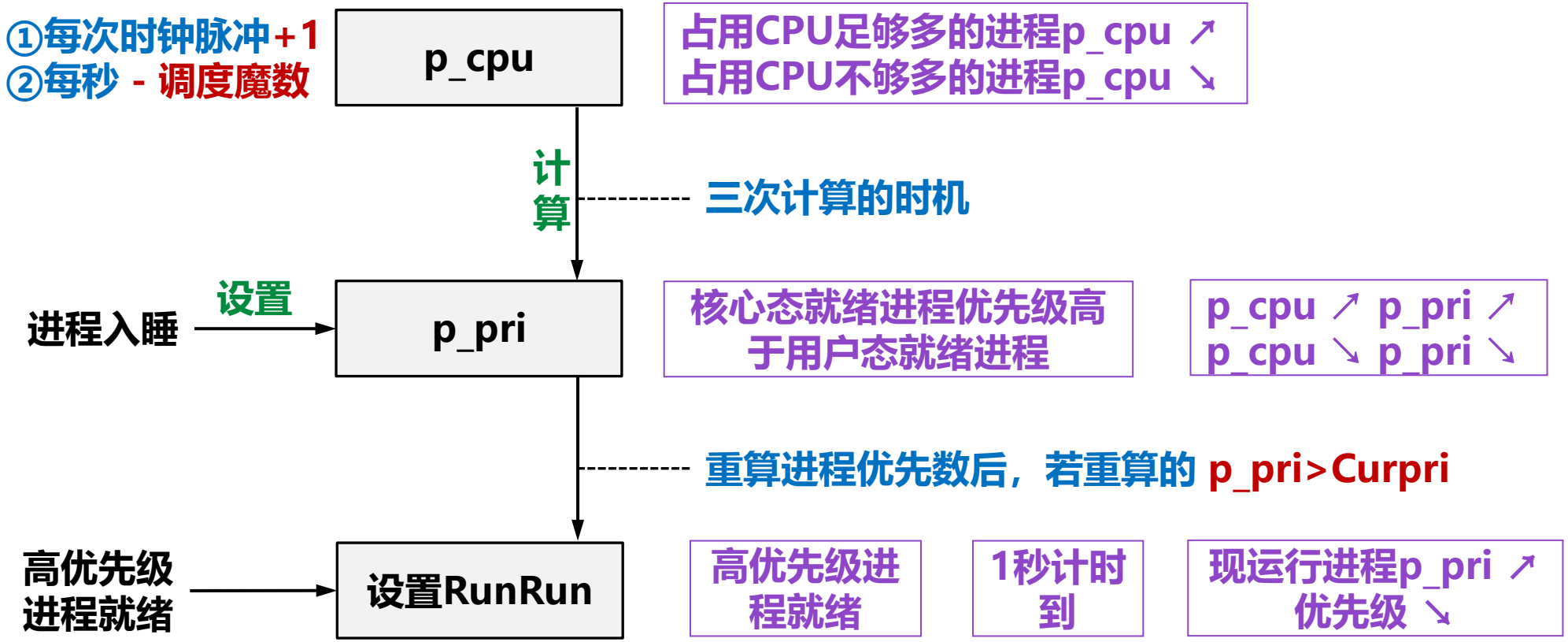
进程入睡 —设置→ p_pri

核心态就绪进程优先级高于用户态就绪进程

p_cpu ↗ p_pri ↗
p_cpu ↘ p_pri ↘

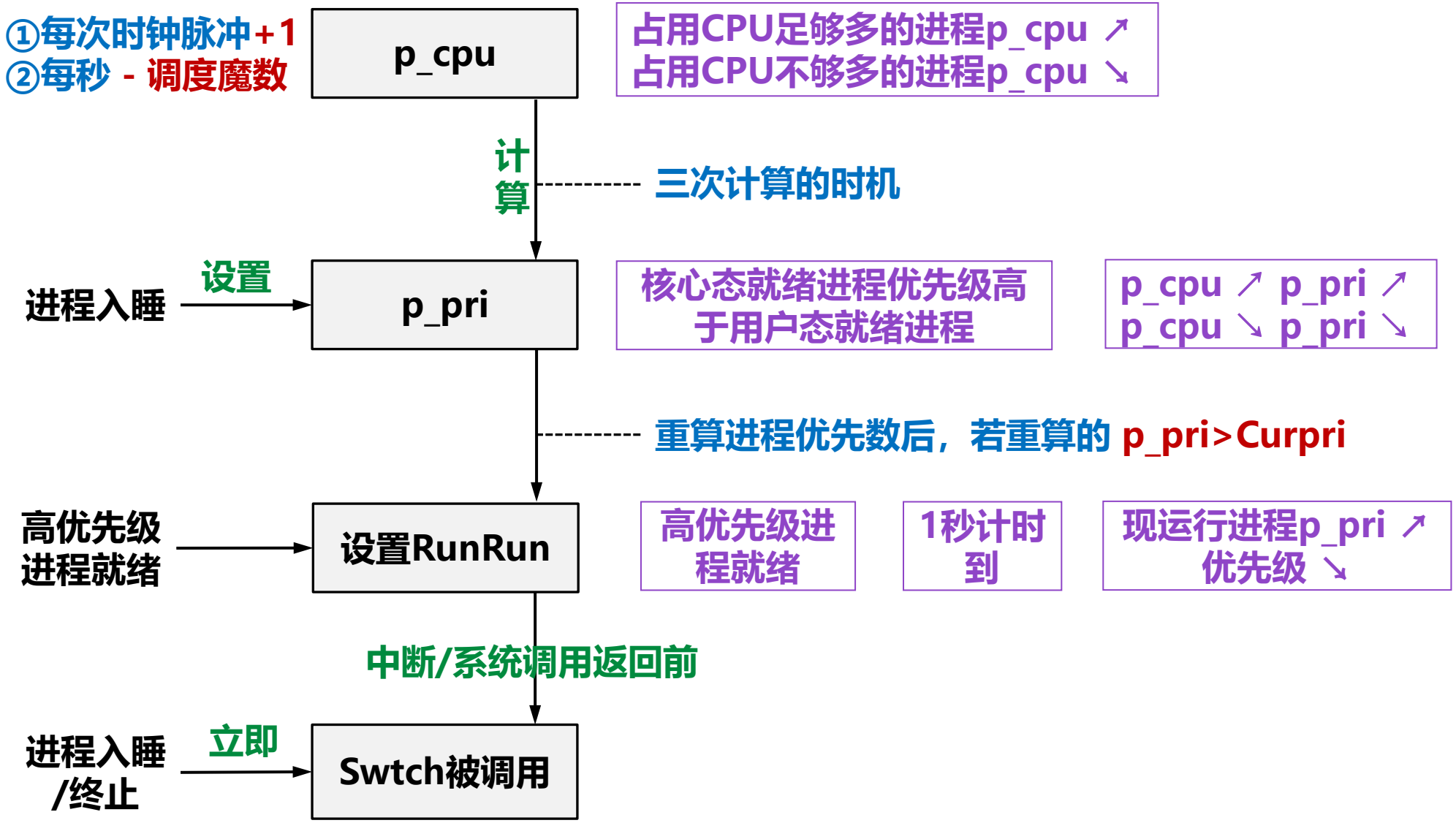
关于UNIX动态优先权调度算法

总结



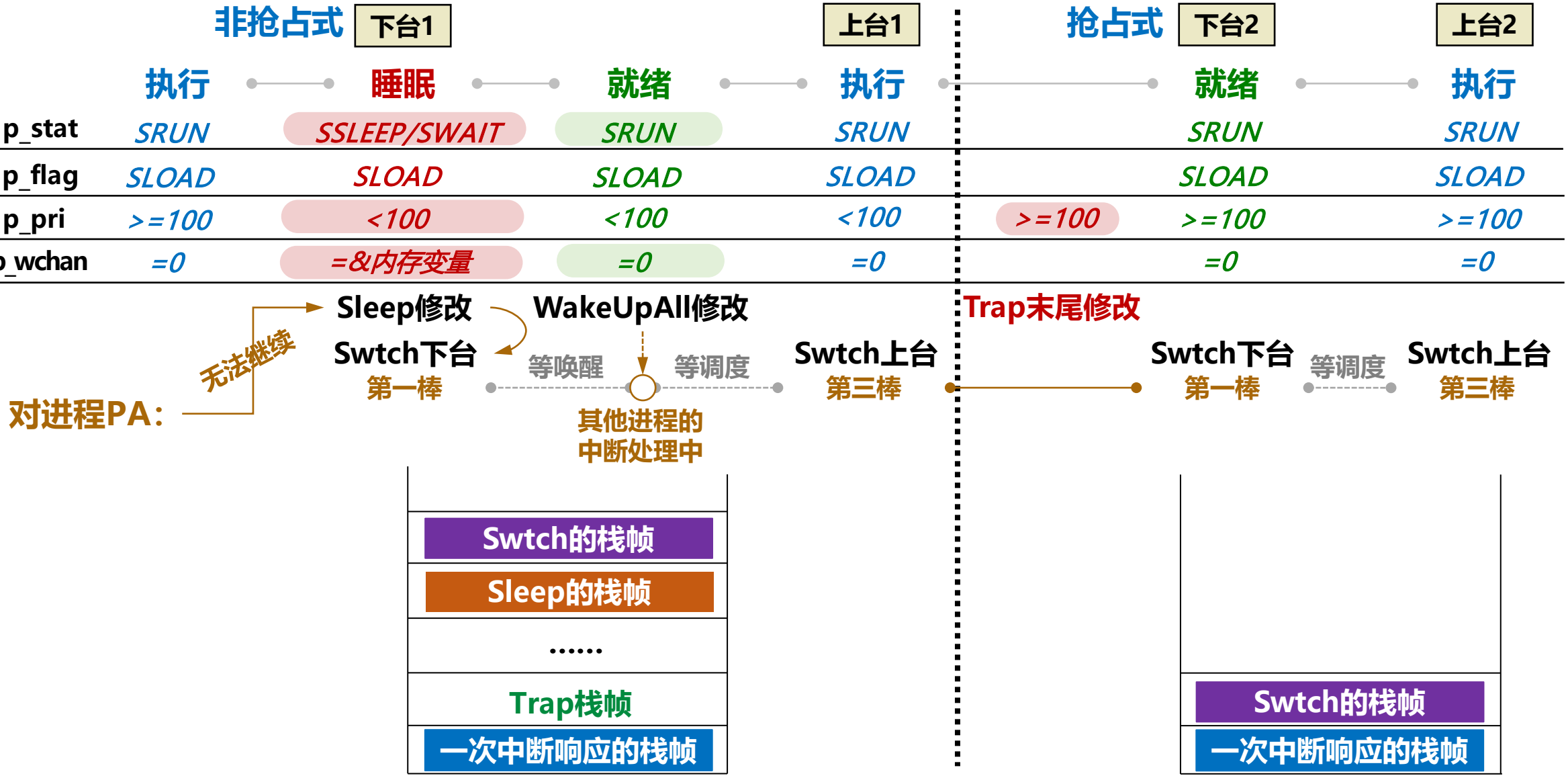
关于UNIX动态优先权调度算法

总结



UNIX的进程调度控制

总结



UNIX的进程调度控制

总结

非抢占式 下台1 上台1

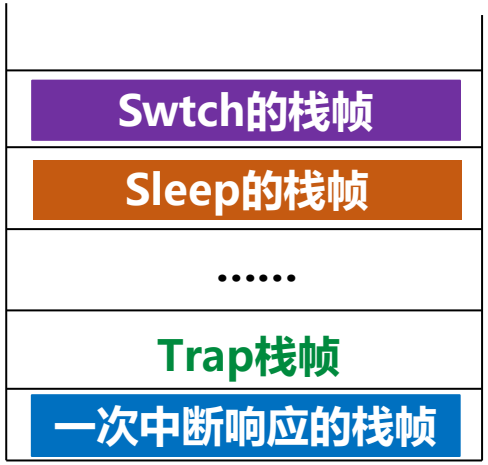
抢占式 下台2 上台2

	执行	睡眠	就绪	执行	就绪	执行
p_stat	SRUN	SSLEEP/SWAIT	SRUN	SRUN	SRUN	SRUN
p_flag	SLOAD	SLOAD	SLOAD	SLOAD	SLOAD	SLOAD
p_pri	>=100	<100	<100	<100	>=100	>=100
p_wchan	=0	=&内存变量	=0	=0	=0	=0

Sleep修改　WakeUpAll修改

Swtch下台　Swtch上台

进程无法再执行下去，立即调用Swtch放弃处理机



Trap末尾修改

Swtch下台　Swtch上台

返回用户态前，进程可以执行下去，但是RunRun>0

本节小结

1　UNIX进程的睡眠与唤醒过程

2　UNIX进程的切换调度

请阅读教材：171页 ~ 183页

E12：进程管理（UNIX的进程调度状态）