

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进程控制



Process类



进程基本控制块

	名称	类型	含义
进程标识	p_uid	short	用户ID
	p_pid	int	进程标识数，进程编号
	p_ppid	int	父进程标识数
进程图象在内存中的位置信息	p_addr	unsigned long	ppda区在物理内存中的起始地址
	p_size	unsigned int	进程图象（除代码段以外部分）的长度，以字节单位
	p_textp	Text *	指向该进程所运行的代码段的描述符
进程调度相关信息	p_stat	ProcessState	进程当前的调度状态
	p_flag	int	进程标志位，可以将多个状态组合
	p_pri	int	进程优先数
	p_cpu	int	cpu值，用于计算p_pri
	p_nice	int	进程优先数微调参数
	p_time	int	进程在盘交换区上（或内存内）的驻留时间
	p_wchan	unsigned long	进程睡眠原因
信号与控制台终端	p_sig	int	进程信号
	p_ttyp	TTy*	进程tty结构地址



Process类



进程基本控制块

	名称	类型	含义
进程标识	p_uid	short	用户ID
	p_pid	int	进程标识数，进程编号
	p_ppid	int	父进程标识数
进程图象在内存中的位置信息	p_addr	unsigned long	ppda区在物理内存中的起始地址
	p_size	unsigned int	进程图象（除代码段以外部分）的长度，以字节单位
	p_textp	Text *	指向该进程所运行的代码段的描述符
进程调度相关信息	p_stat	ProcessState	进程当前的调度状态
	p_flag	int	进程标志位，可以将多个状态组合
	p_pri	int	进程优先数
	p_cpu	int	cpu值，用于计算p_pri
	p_nice	int	进程优先数微调参数
	p_time	int	进程在盘交换区上（或内存内）的驻留时间
	p_wchan	unsigned long	进程睡眠原因
信号与控制台终端	p_sig	int	进程信号
	p_ttyp	TTy*	进程tty结构地址



```
class Process
{
public:
    enum ProcessState/* 进程状态 */
    {
        SNULL= 0, /* 未初始化空状态 */
        SSLEEP= 1, /* 高优先权睡眠 */
        SWAIT= 2, /* 低优先权睡眠 */
        SRUN= 3, /* 运行、就绪状态 */
        SIDL= 4, /* 进程创建时的中间状态 */
        SZOMB= 5, /* 进程终止时的中间状态 */
        SSTOP= 6 /* 进程正被跟踪 */
    };
    .....
};
```

**p_stat一定为7个状态
其中之一!**

e.g.

1. **if (process[i].p_stat == Process::SNULL)**
判断某一个process表中某一个是否为空
2. **if (this->p_stat == Process::SWAIT || this->p_stat == Process::SSLEEP)**
判断进程是否在高睡或者低睡状态
3. **this->p_stat = Process::SRUN;**
将进程的调度状态改为就绪状态



进程标识

名称

类型

含义

p_uid

p_pid

p_ppid

p_addr

p_size

p_textp

进程图像在内存中的位置信息

进程调度相关信息

p_stat

ProcessState

进程当前的调度状态

p_flag

int

进程标志位，可以将多个状态组合

p_pri

进程优先数

信号与控制台终端

5	4	3	2	1	0
STWED	STRC	SSWAP	SLOCK	SSYS	SLOAD

p_flag = SLOAD | SLOCK → p_flag=101; //二进制

p_flag &= ~SLOCK; → p_flag=001;

p_flag &= ~SLOAD; → p_flag=000; //二进制

if (p_flag & SLOAD) != 0)

enum ProcessFlag /* 进程标志位（用于进程图像换进换出）

{

SLOAD = 0x1,

/* 进程图像在内存中

SSYS = 0x2,

/* 系统进程图像，不允许被换出

SLOCK = 0x4,

/* 含有该标志的进程图像暂不允许换出

SSWAP = 0x8,

/* 该进程被创建时图像就在交换区上

STRC = 0x10,

/* 父子进程跟踪标志，UNIX V6++未使用到

STWED = 0x20

/* 父子进程跟踪标志，UNIX V6++未使用到

};

p_flag可以是6个值的合理组合！



Process类

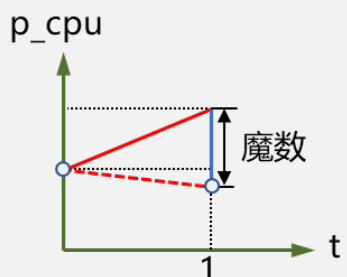
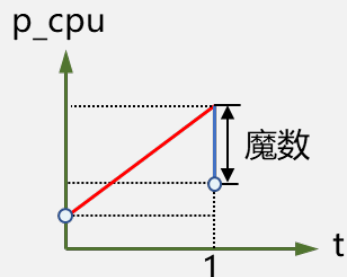


进程基本控制块

进程标识

名称	类型	含义
p_uid	short	用户ID

进
置



$$p_pri = \min \{ 255, (p_cpu / 16 + \text{PUSER} + p_nice) \}$$



1. 整数秒，重算所有用户态就绪的进程的优先数



2. 整数秒，重算现运行进程的优先数



3. 系统调用末尾，重算现运行进程的优先数

进

p_...		进程标识位，可以将多个状态组合
p_pri	int	进程优先数
p_cpu	int	cpu值，用于计算p_pri
p_nice	int	进程优先数微调参数
p_time	int	进程在盘交换区上（或内存内）的驻留时间
p_wchan	unsigned long	进程睡眠原因
p_sig	int	进程信号
p_ttyp	TTy*	进程tty结构地址

信号与控制台终端



Process类



进程基本控制块

	名称	类型	含义
进程标识	p_uid	short	用户ID
	p_pid	int	进程标识数, 进程编号
	p_ppid	int	父进程标识数
进程图象在内存中的位置信息	p_addr	unsigned long	ppda区在物理内存中的起始地址
	p_size	unsigned int	进程图象 (除代码段以外部分) 的长度, 以字节单位
	p_textp	Text *	指向该进程所运行的代码段的描述符
进程调度相关信息	p_stat	ProcessState	进程当前的调度状态
	p_flag	int	进程标志位, 可以将多个状态组合
	p_pri	int	进程优先数
	p_cpu	int	时间每过去1秒, p_time++; 图像每交换一次, p_time = 0
	p_nice	int	
	p_time	int	进程在盘交换区上 (或内存内) 的驻留时间
	p_wchan	unsigned long	进程睡眠原因
信号与控制台终端	p_sig	int	进程信号
	p_ttyp	TTy*	进程tty结构地址



Process类



进程基本控制块

	名称	类型	含义
进程标识	p_uid	short	用户ID
	p_pid	int	进程标识数, 进程编号
	p_ppid	int	父进程标识数
进程图象在内存中的位置信息	p_addr	unsigned long	ppda区在物理内存中的起始地址
	p_size	unsigned int	进程图象 (除代码段以外部分) 的长度, 以字节单位
	p_textp	Text *	指向该进程所运行的代码段的描述符
进程调度相关信息	p_stat	ProcessState	进程当前的调度状态
	p_flag	int	进程标志位, 可以将多个状态组合
	p_pri	int	进程优先数
	p_cpu	int	cpu值, 用于计算p_pri
	p_nice	int	进程入睡时, p_wchan = & 某一内存变量; 进程未睡时, p_wchan = 0
	p_time	int	
	p_wchan	unsigned long	进程睡眠原因
信号与控制台终端	p_sig	int	进程信号
	p_ttyp	TTy*	进程tty结构地址



Process类



进程基本控制块

<code>void SetRun();</code>	<code>/* 唤醒当前进程，转入就绪状态 */</code>	和单个进程的调度控制相关
<code>bool IsSleepOn(unsigned long chan);</code>	<code>/* 检查当前进程睡眠原因是否为chan */</code>	
<code>void Sleep(unsigned long chan, int pri);</code>	<code>/* 使当前进程转入睡眠状态 */</code>	
<code>void Exit();</code>	<code>/* Exit()系统调用处理过程 */</code>	
<code>void Clone(Process& proc);</code>	<code>/* 除p_pid之外子进程拷贝父进程Process结构 */</code>	
<code>void SetPri();</code>	<code>/* 根据占用CPU时间计算当前进程优先数 */</code>	和进程优先数的计算相关
<code>void Nice();</code>	<code>/* 用户设置计算进程优先数的偏置值 */</code>	
<code>void Expand(unsigned int newSize);</code>	<code>/* 改变进程占用的内存大小 */</code>	与进程图像的改变相关
<code>void SStack();</code>	<code>/* 堆栈溢出时，自动扩展堆栈 */</code>	
<code>void SBreak();</code>	<code>/* brk()系统调用处理过程 */</code>	
<code>void PSignal(int signal);</code>	<code>/* 向当前进程发送信号 */</code>	和进程的信号处理相关
<code>void PSig(struct pt_context* pContext);</code>	<code>/* 对当前进程接收到的信号进行处理 */</code>	
<code>void Ssig();</code>	<code>/* 设置用户自定信号处理方式的系统调用处理函数 */</code>	
<code>int IsSig();</code>	<code>/* 判断当前进程是否接收到信号 */</code>	

和单个进程的控制相关的操作作为Process类的成员函数



ProcessManager类



名称	类型	含义
process[NPROC]	<u>Process</u>	进程基本控制块数组
text[NTEXT]	<u>Text</u>	代码段控制块数组
CurPri	int	现运行占用CPU时优先数
RunRun	int	强迫调度标志
RunIn	int	内存中无合适进程可以调出至盘交换区
RunOut	int	盘交换区中无进程可以调入内存
ExeCnt	int	同时进行图像改换的进程数
SwchNum	int	系统中进程切换次数



ProcessManager类



```
void Initialize();  
void SetupProcessZero();          /* 手工创建系统0#进程 */
```

和初始化相关

```
Process* Select();                /* 选出最适合上台运行的进程 */  
int Switch();                     /* 进程的切换调度 */  
void WakeUpAll(unsigned long chan); /* 唤醒系统中所有因chan而进入睡眠的进程 */  
void Wait();                      /* 父进程等待子进程结束的Wait()系统调用 */
```

和所有进程的
调度控制相关

```
int NewProc(); /* 用于生成当前正在运行进程的拷贝 */  
void Fork();   /* 进程创建Fork()系统调用 */  
void Exec();   /* Exec()系统调用，进程图像改换 */  
void Kill();   /* 终止进程Kill()系统调用 */
```

与新进程的创
建相关

```
void Sched(); /* 进程图像内存和交换区之间的传送 */  
void XSwap(Process* pProcess, bool bFreeMemory, int size); /* 将进程从内存换出至交换区 */
```

和进程的图像
交换相关

和所有进程的控制相关的操作作为ProcessManager类的成员函数



ProcessManager类



进程管理



Manager

... ..

由我来完成进程的初始化



只有我手上有所有进程的花名册 `process[NPROC]`, 详细记录了每个进程的运行状况

需要进程切换的时候, 由我来执行 `Swth`, 在所有进程中选择 (`Select`) 一个我认为最合适的进程, 并让其上台。


由我根据系统中资源使用的情况决定唤醒 (`WakeUpAll`) 哪个进程

只有我有能力创建一个新的进程 (`Newproc`, `Fork`, `Exec`), 并根据内存使用状况决定新进程创建在什么位置

只有我有能力根据内存的使用状况决定是否需要进行进程图像的交换 (`Sched`)

我只知道自己的运行状况

我在运行过程中, 可以在适当时机重算自己的优先数 (`SetPri`, `Nice`), 可以根据需要改变自己的地址空间 (`Expand`, `SStack`, `SBreak`), 可以发送和进行信号处理。

如果在运行过程中, 我无法得到继续运行需要的资源, 我会去睡觉 (`Sleep`), 直到  来叫我; 我醒来 (`SetRun`) 后, 等待下一次上来的机会。



Process

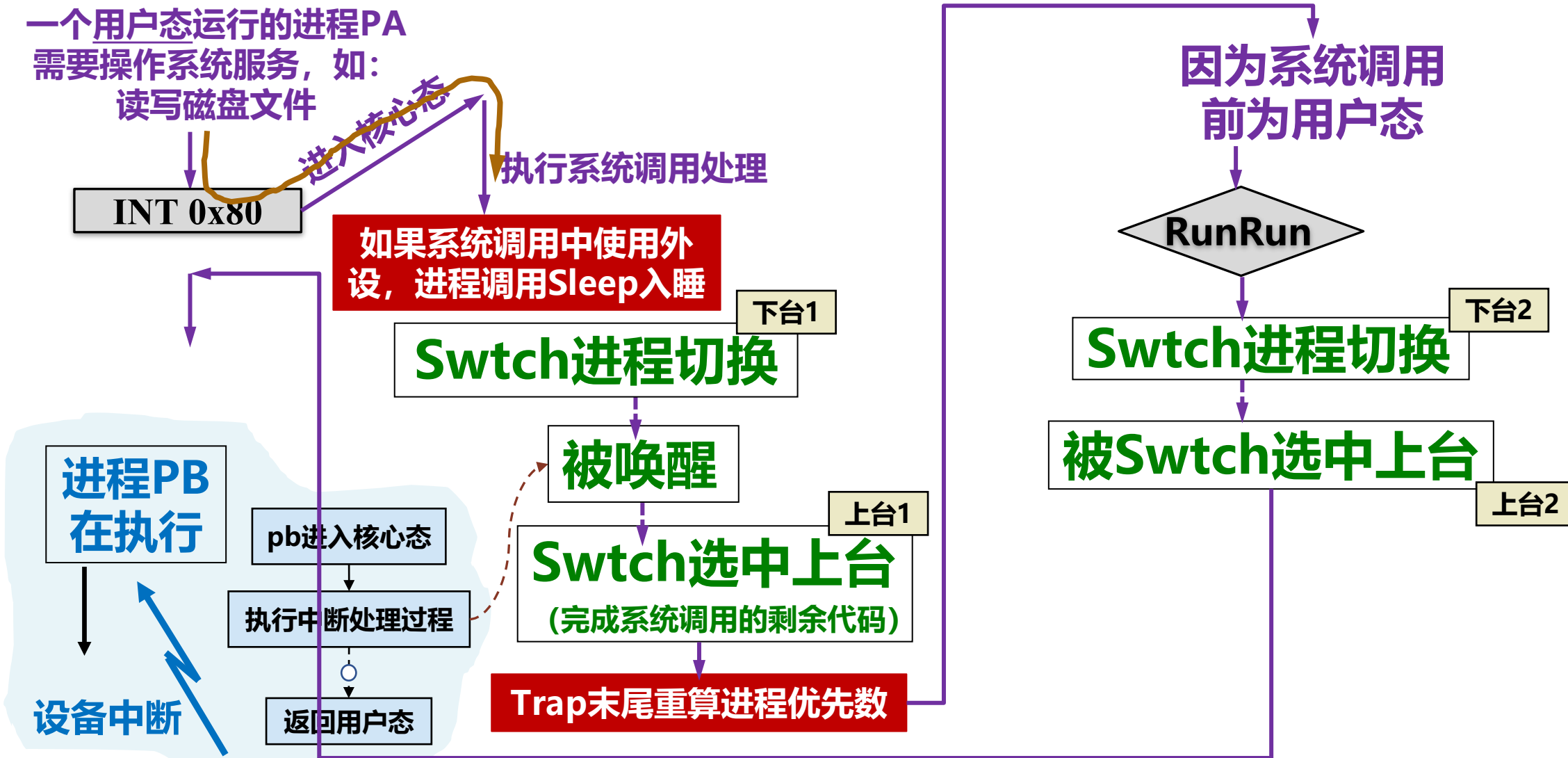
... ..



UNIX进程的调度状态



系统调用与中断的关系





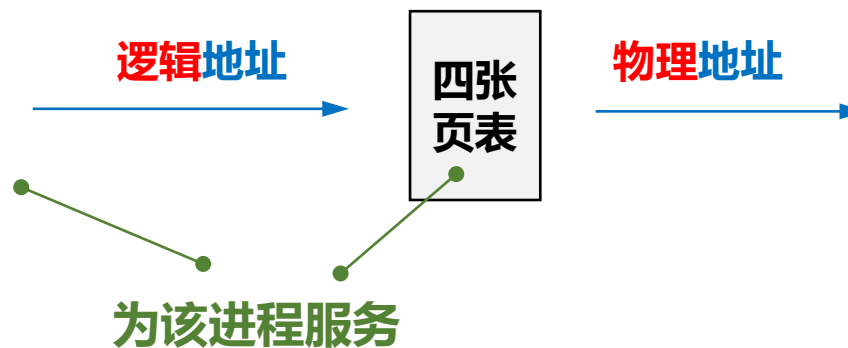
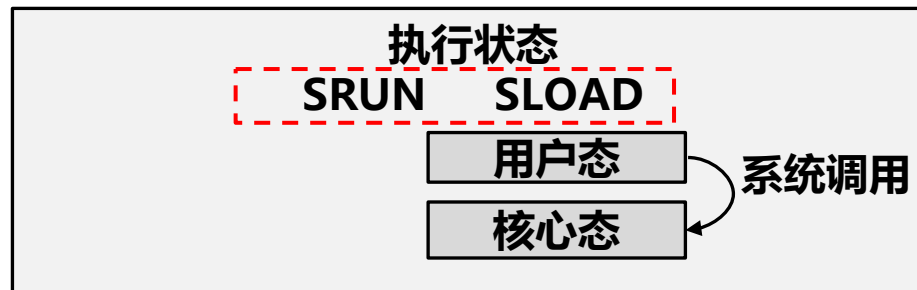
UNIX进程的调度状态



执行状态

p_stat=SRUN
p_flags & SLOAD != 0
p_wchan = 0

CPU内部各个地址寄存器：
EIP、EBP、ESP
.....





UNIX进程的调度状态



执行

p_stat *SRUN*

p_flag *SLOAD*

p_pri *>=100*

p_wchan *=0*

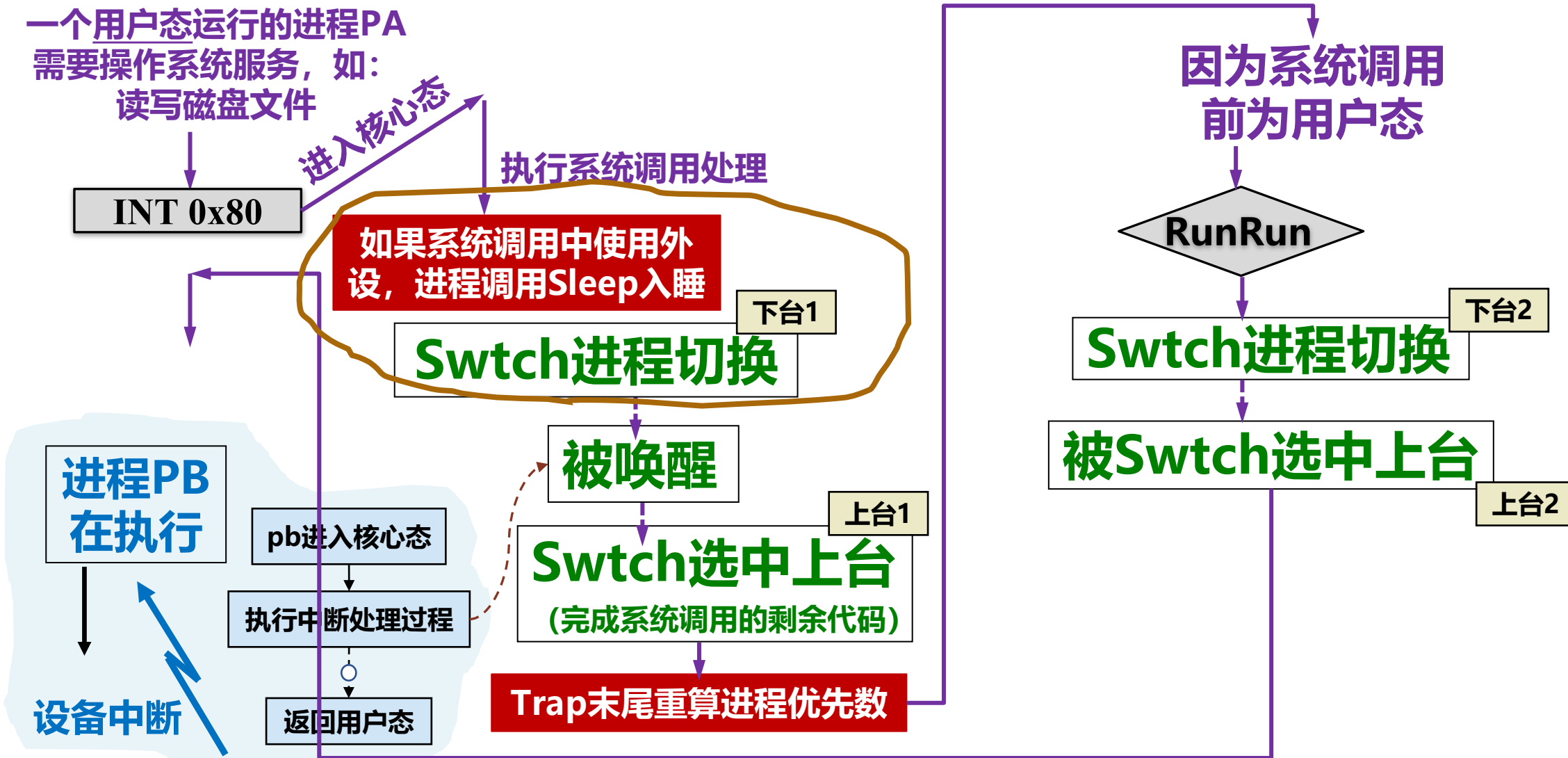
进程的调度状态



UNIX进程的调度状态



系统调用与中断的关系





UNIX进程的调度状态



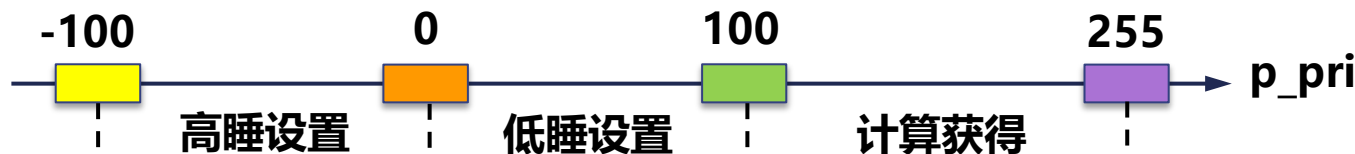
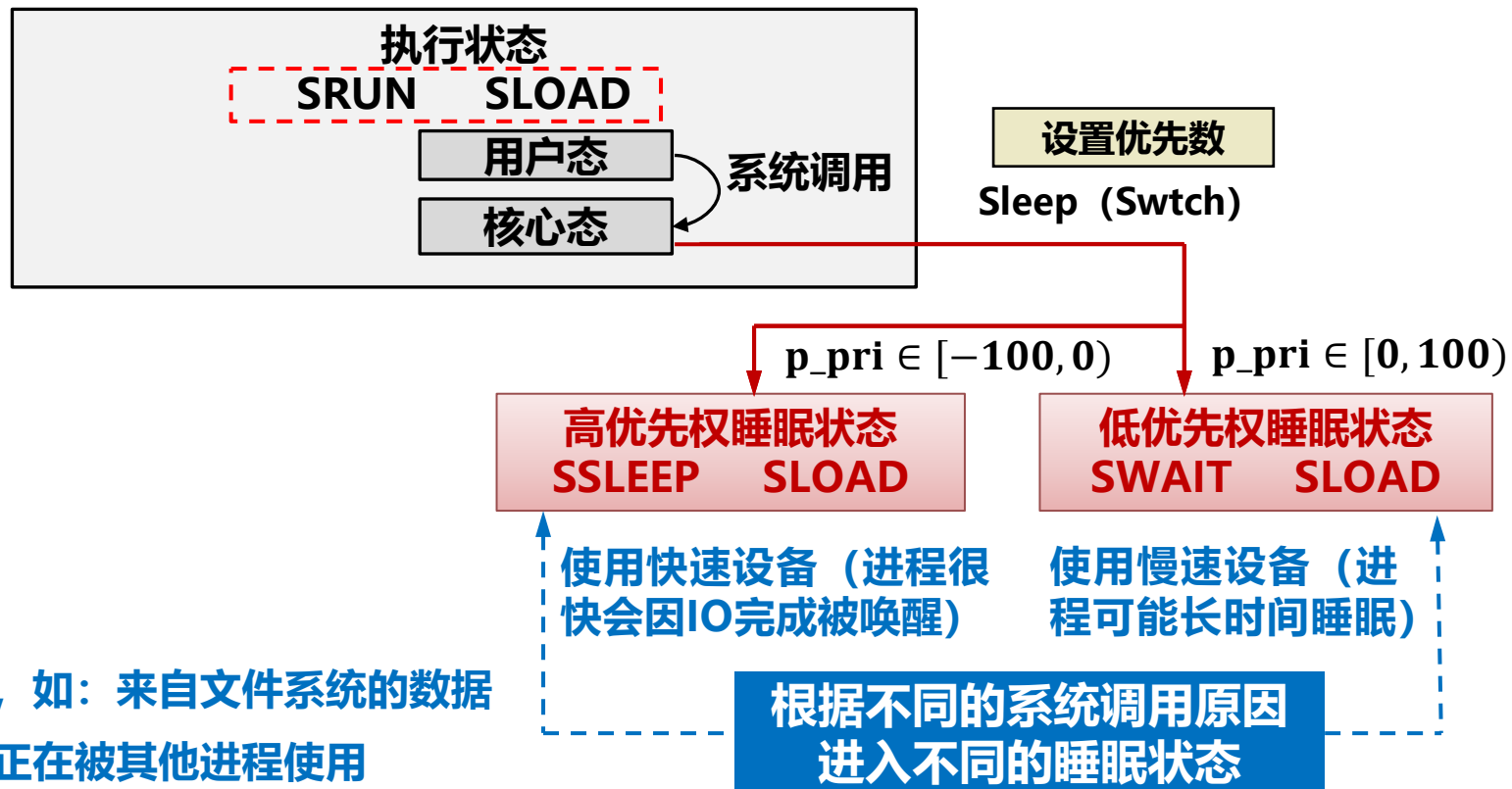
① 非抢占调度

睡眠状态

`p_stat=SSLEEP/SWAIT`
`p_flags & SLOAD != 0`
`p_wchan = & 某内存变量`
`p_pri < 100`

导致进程入睡的原因可能有：

1. 进程需要处理的外部数据不在内存，如：来自文件系统的数据
2. 进程需要使用的外设或共享数据，正在被其他进程使用
3. 进程的前驱任务没有结束





UNIX进程的调度状态



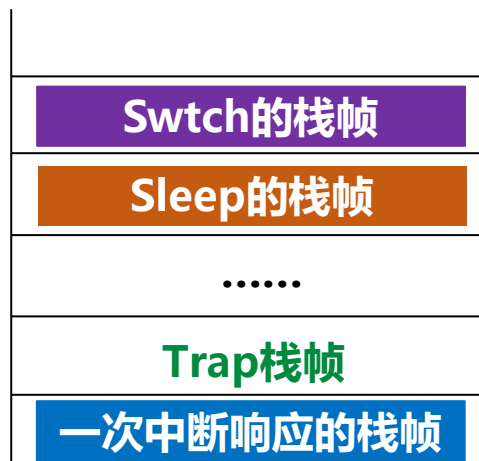
非抢占式 下台1

执行

睡眠

p_stat	SRUN	SSLEEP/SWAIT
p_flag	SLOAD	SLOAD
p_pri	≥ 100	< 100
p_wchan	=0	=&内存变量
		Sleep修改
		Swch下台

此时进程的核心栈?

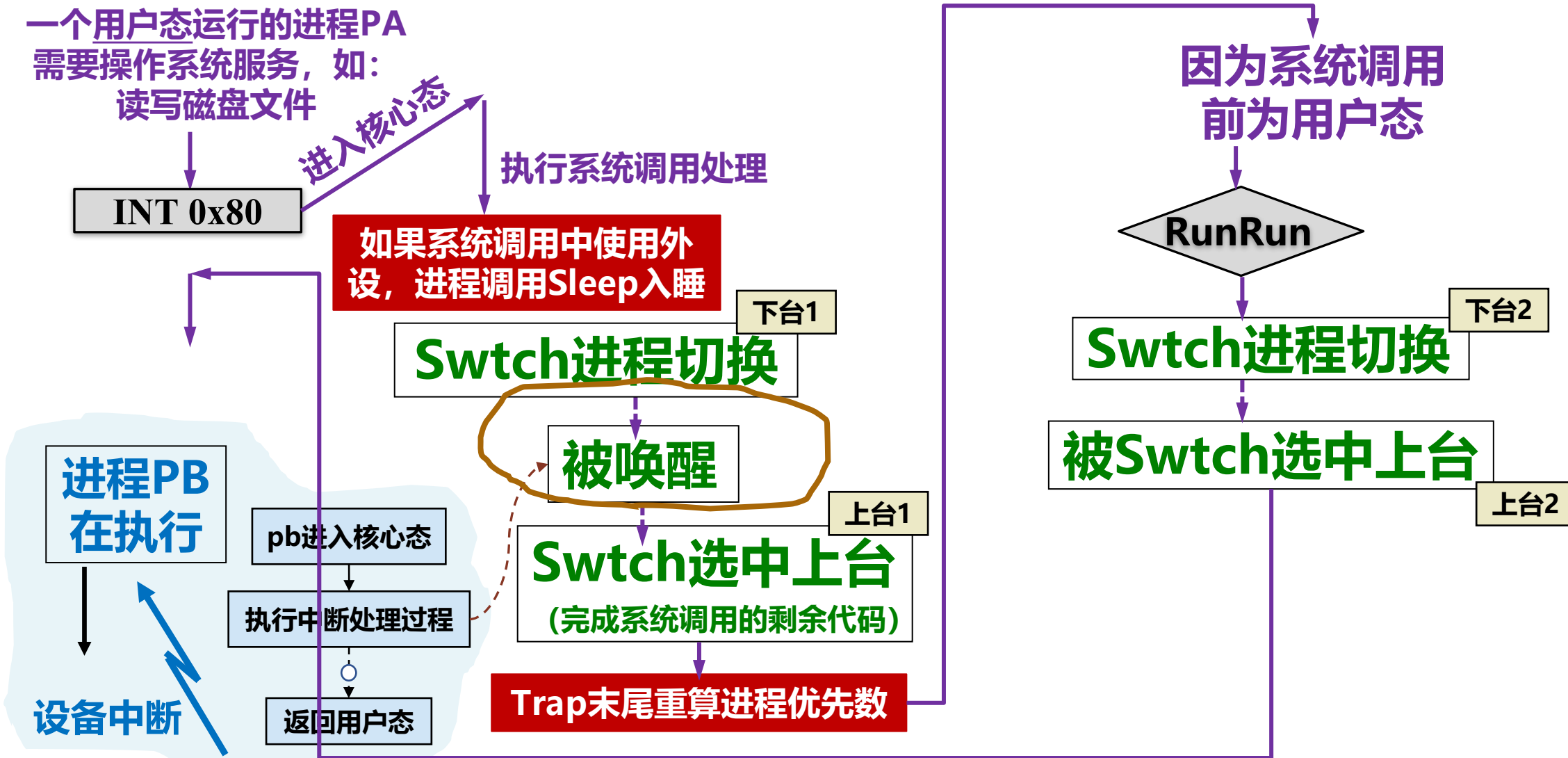




UNIX进程的调度状态



系统调用与中断的关系





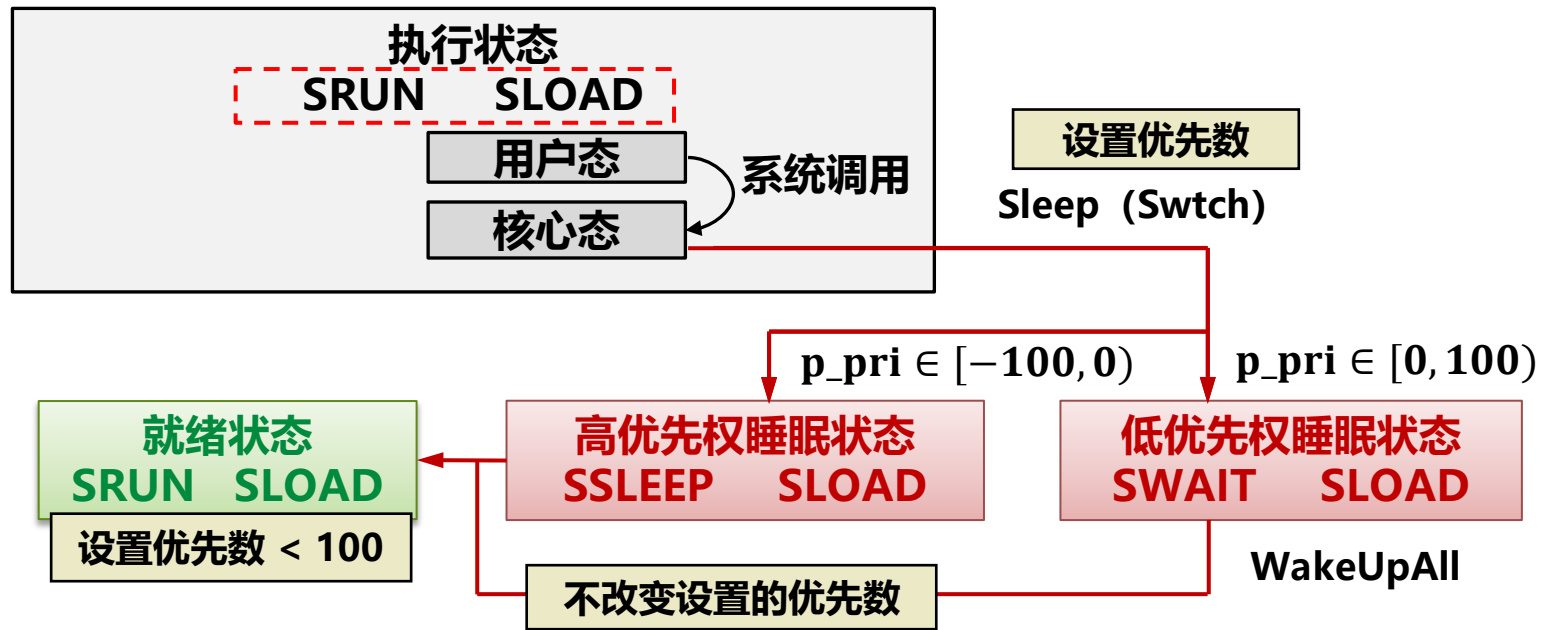
UNIX进程的调度状态



① 非抢占调度

就绪状态

$p_stat = \text{SRUN}$
 $p_flags \ \& \ \text{SLOAD} \neq 0$
 $p_wchan = 0$





UNIX进程的调度状态



非抢占式 下台1

执行

睡眠

就绪

p_stat	SRUN	SSLEEP/SWAIT	SRUN
p_flag	SLOAD	SLOAD	SLOAD
p_pri	≥ 100	< 100	< 100
p_wchan	=0	=&内存变量	=0

Sleep修改
Swch下台

WakeUpAll修改

Swch的栈帧

Sleep的栈帧

.....

Trap栈帧

一次中断响应的栈帧

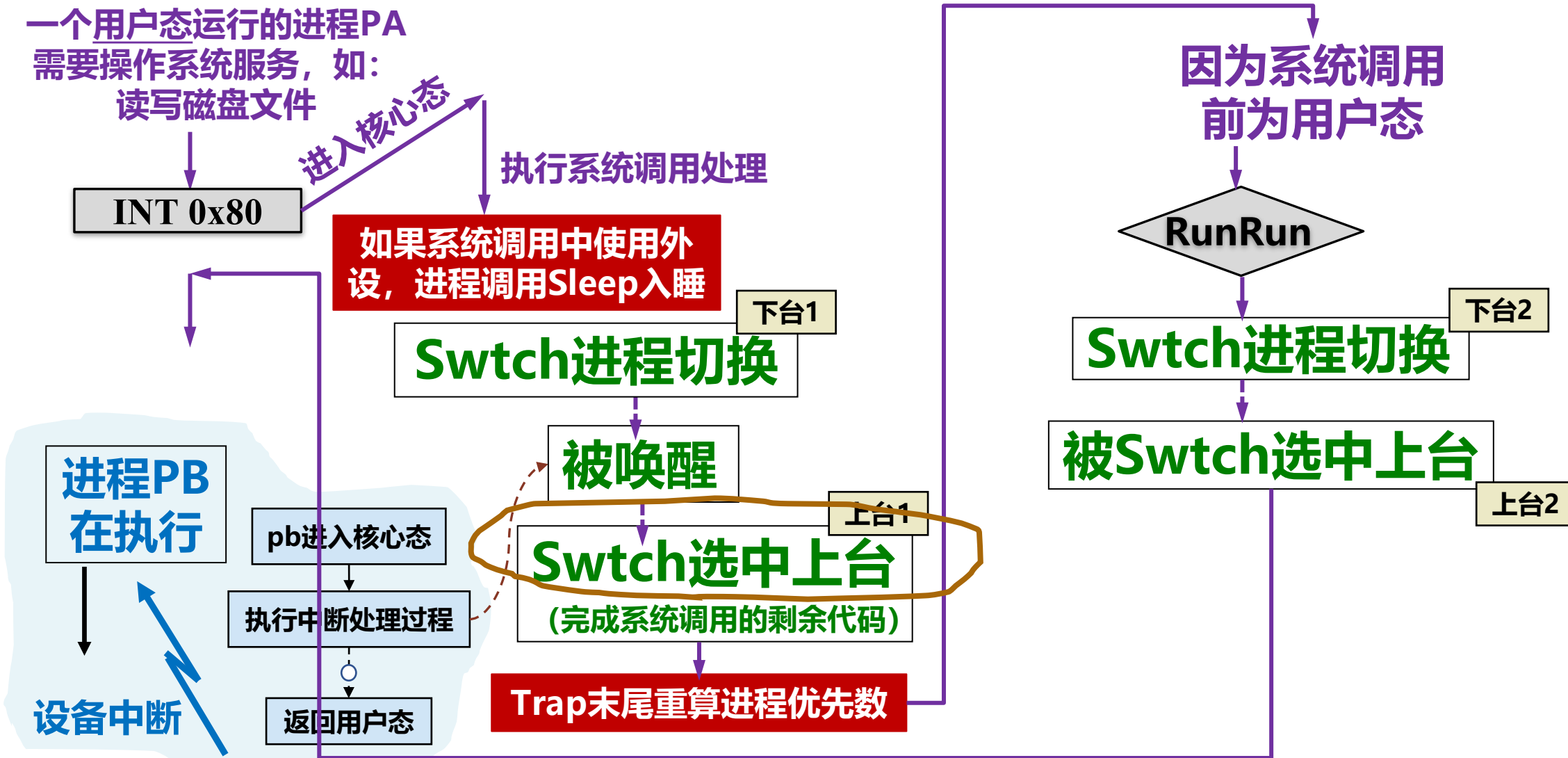
进程的调度状态



UNIX进程的调度状态



系统调用与中断的关系

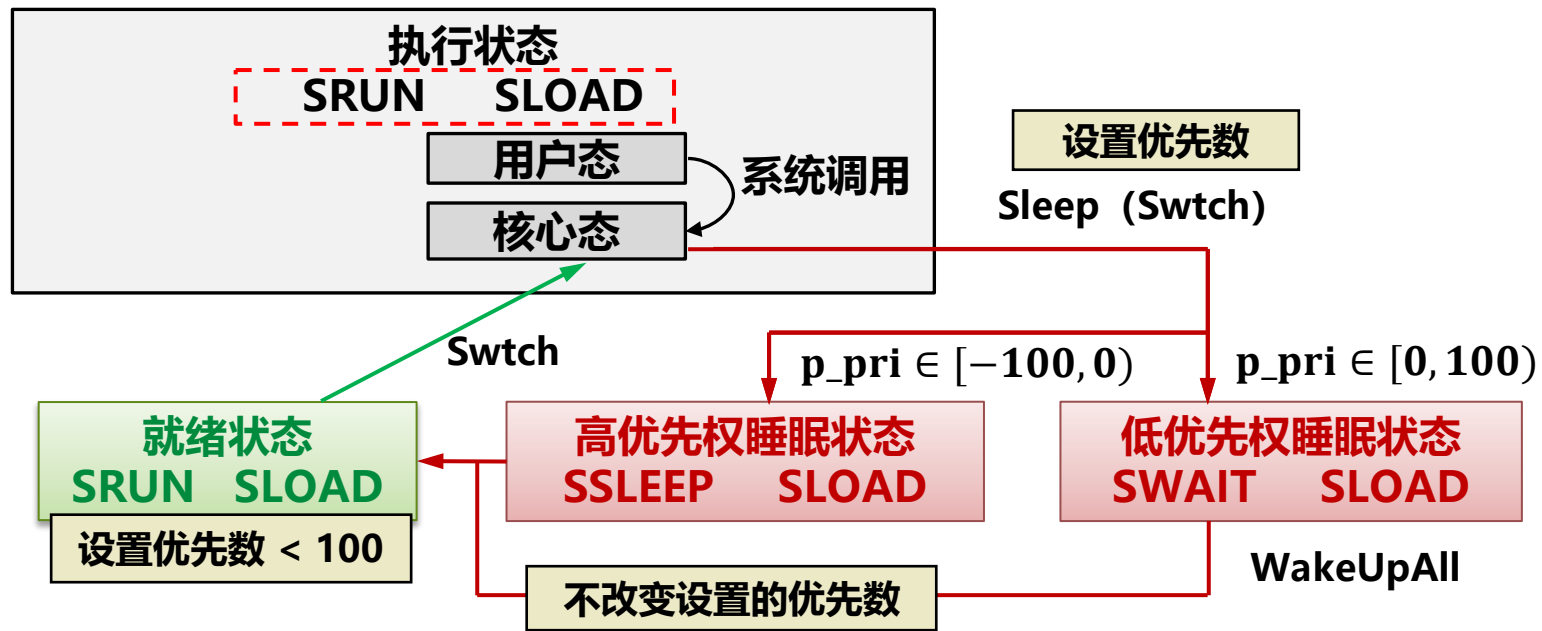




UNIX进程的调度状态



① 非抢占调度





UNIX进程的调度状态



非抢占式

下台1

上台1

执行

睡眠

就绪

执行

p_stat

SRUN

SSLEEP/SWAIT

SRUN

SRUN

p_flag

SLOAD

SLOAD

SLOAD

SLOAD

p_pri

≥ 100

< 100

< 100

< 100

p_wchan

=0

=&内存变量

=0

=0

Sleep修改

WakeUpAll修改

Swch下台

Swch上台



进程未上台执行，核心栈不变

Swch的栈帧

Sleep的栈帧

.....

Trap栈帧

一次中断响应的栈帧

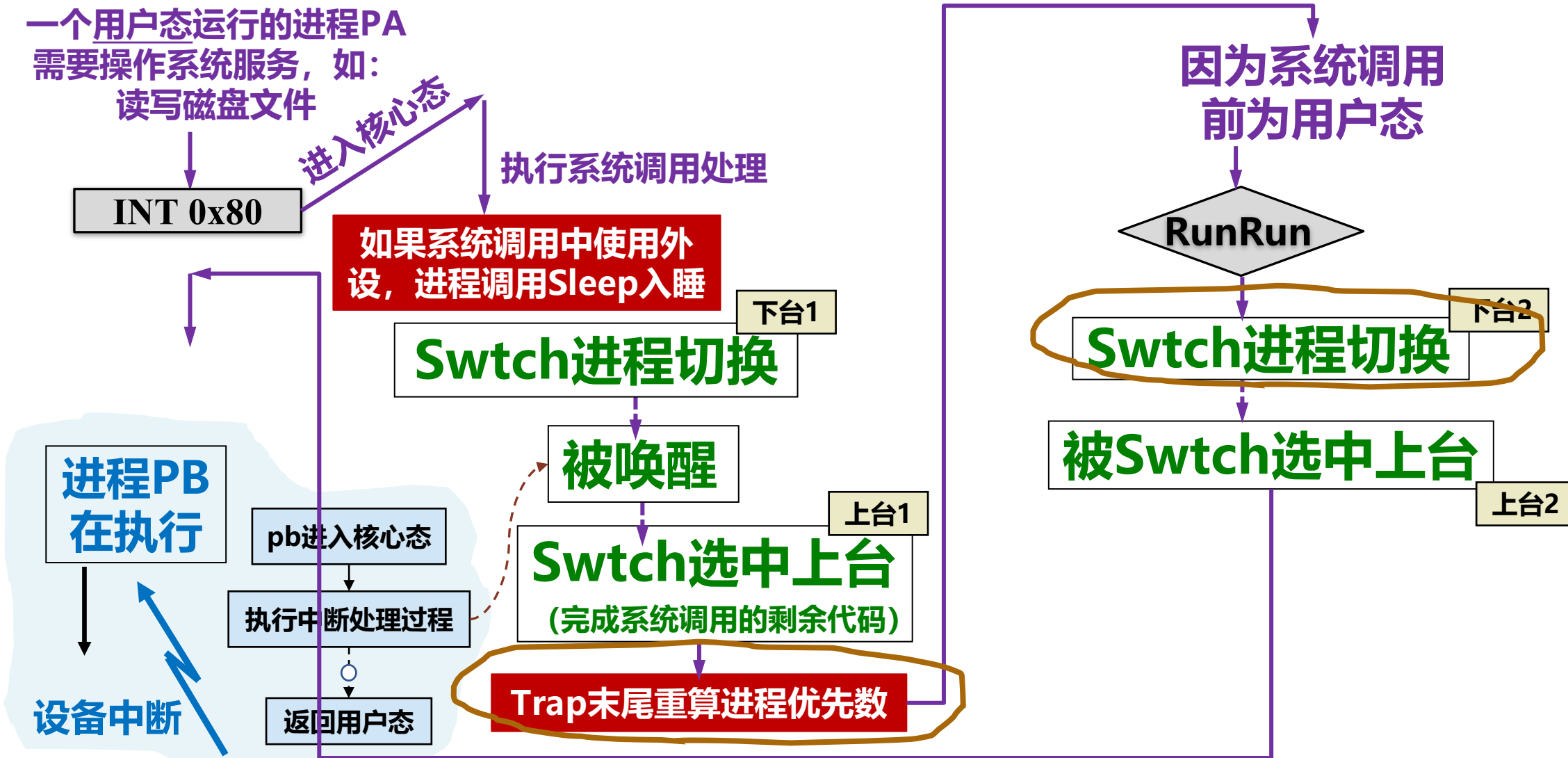
进程的调度状态



UNIX进程的调度状态



系统调用与中断的关系

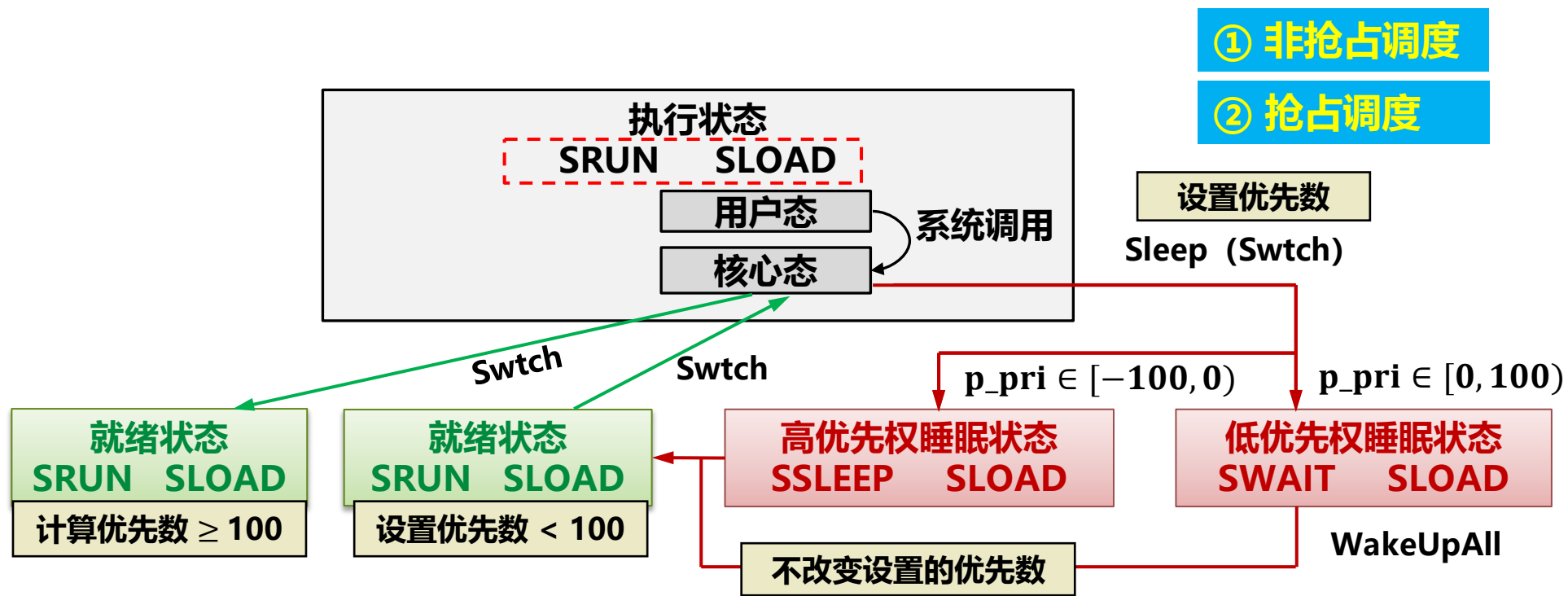




UNIX进程的调度状态



进程的调度状态

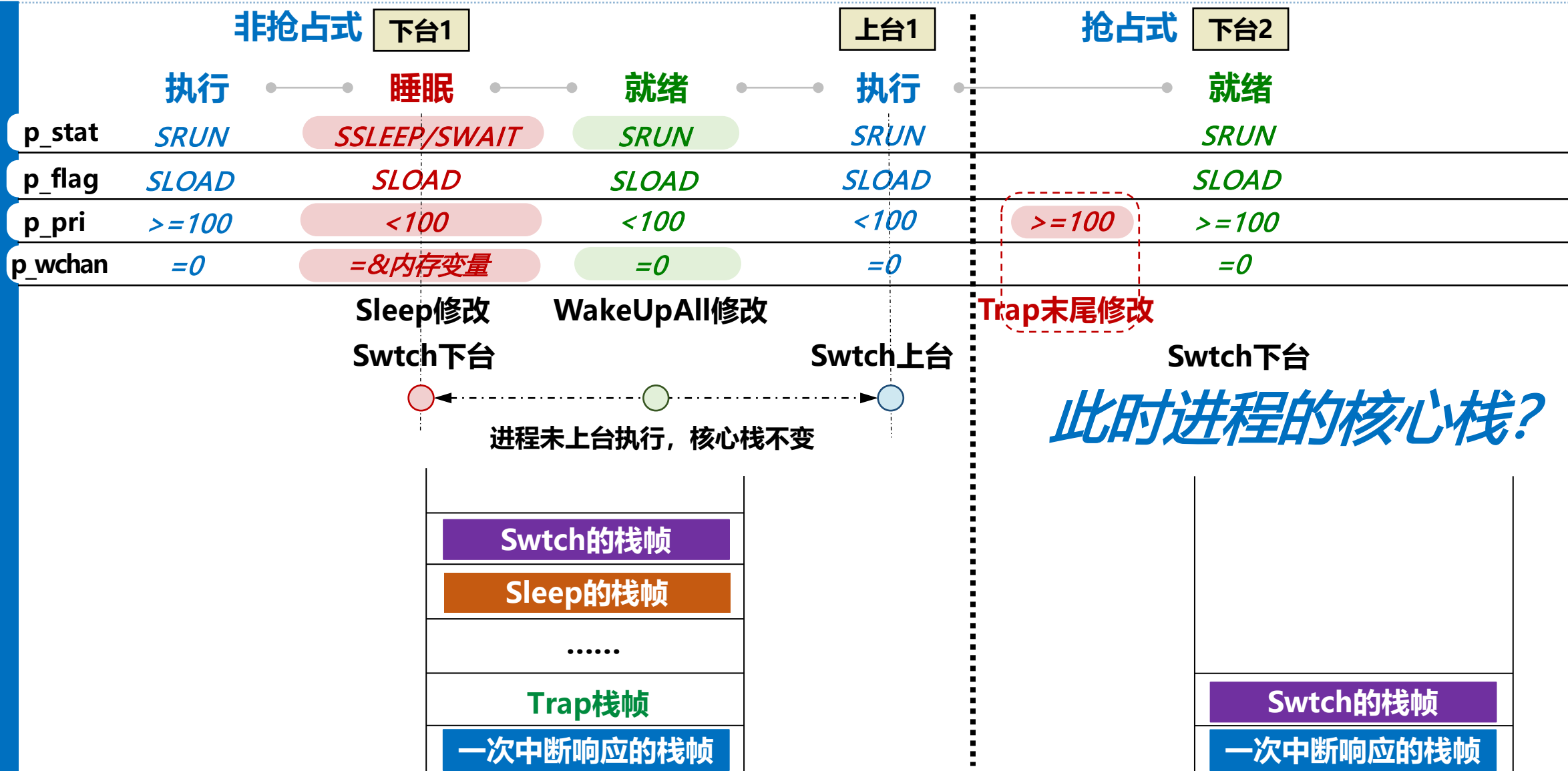




UNIX进程的调度状态



进程的调度状态

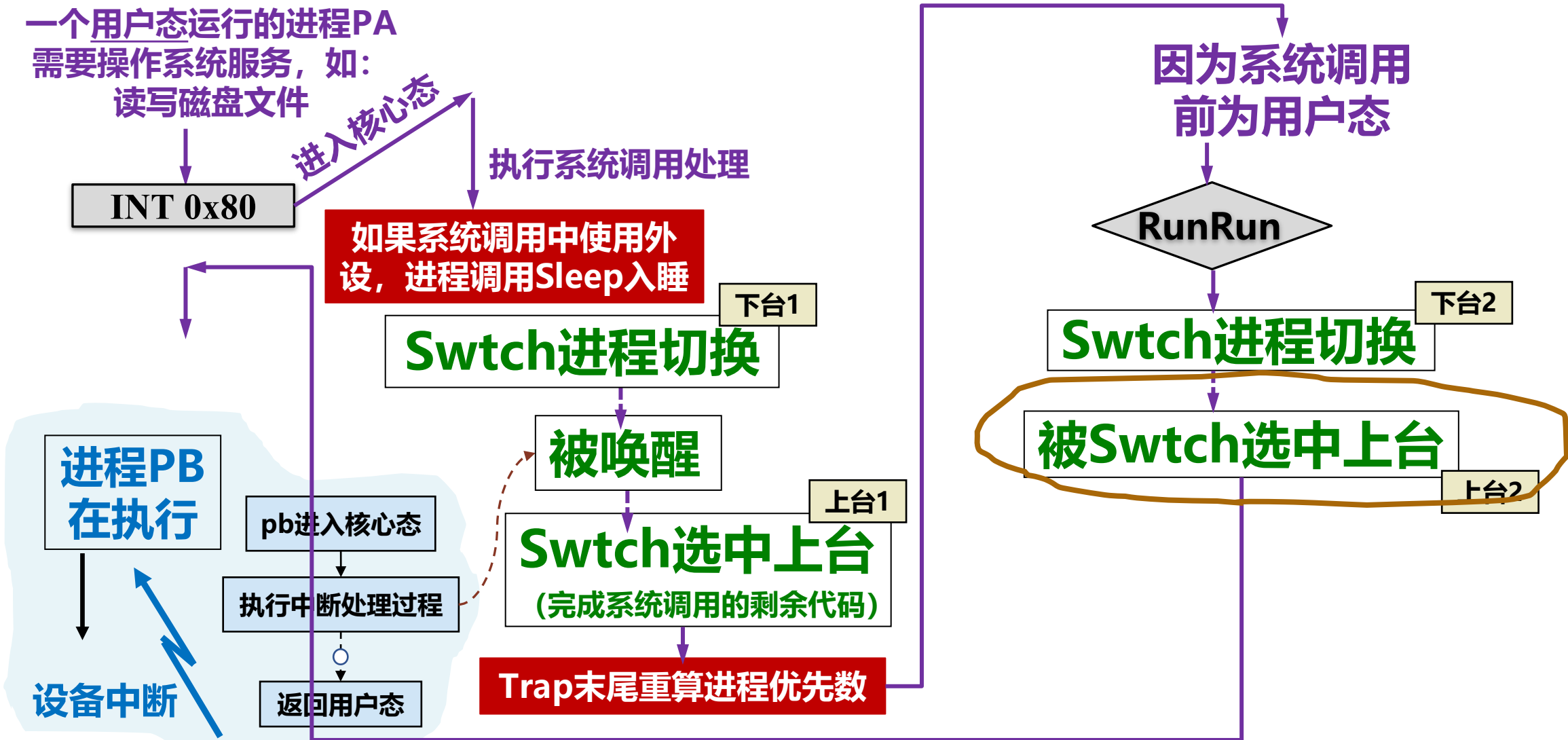




UNIX进程的调度状态



系统调用与中断的关系

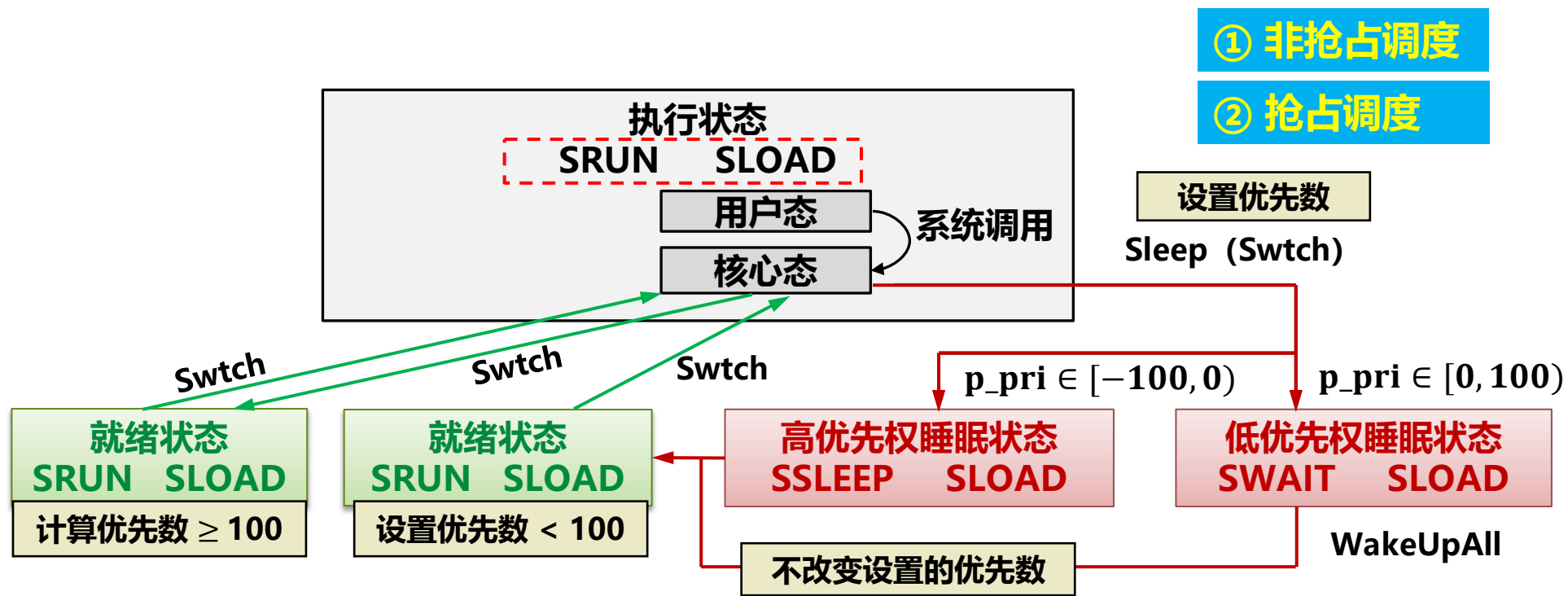




UNIX进程的调度状态



进程的调度状态

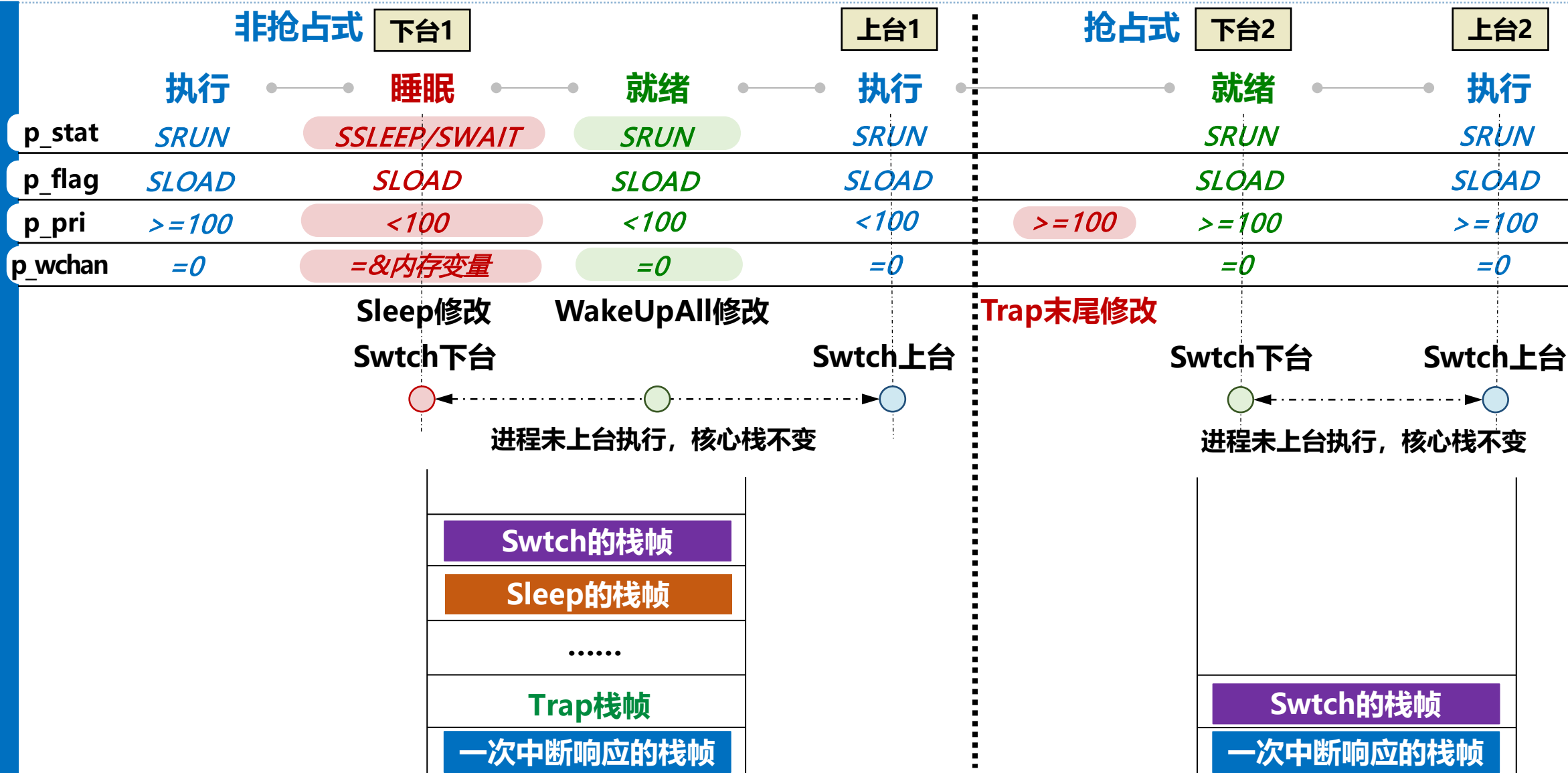




UNIX进程的调度状态



进程的调度状态

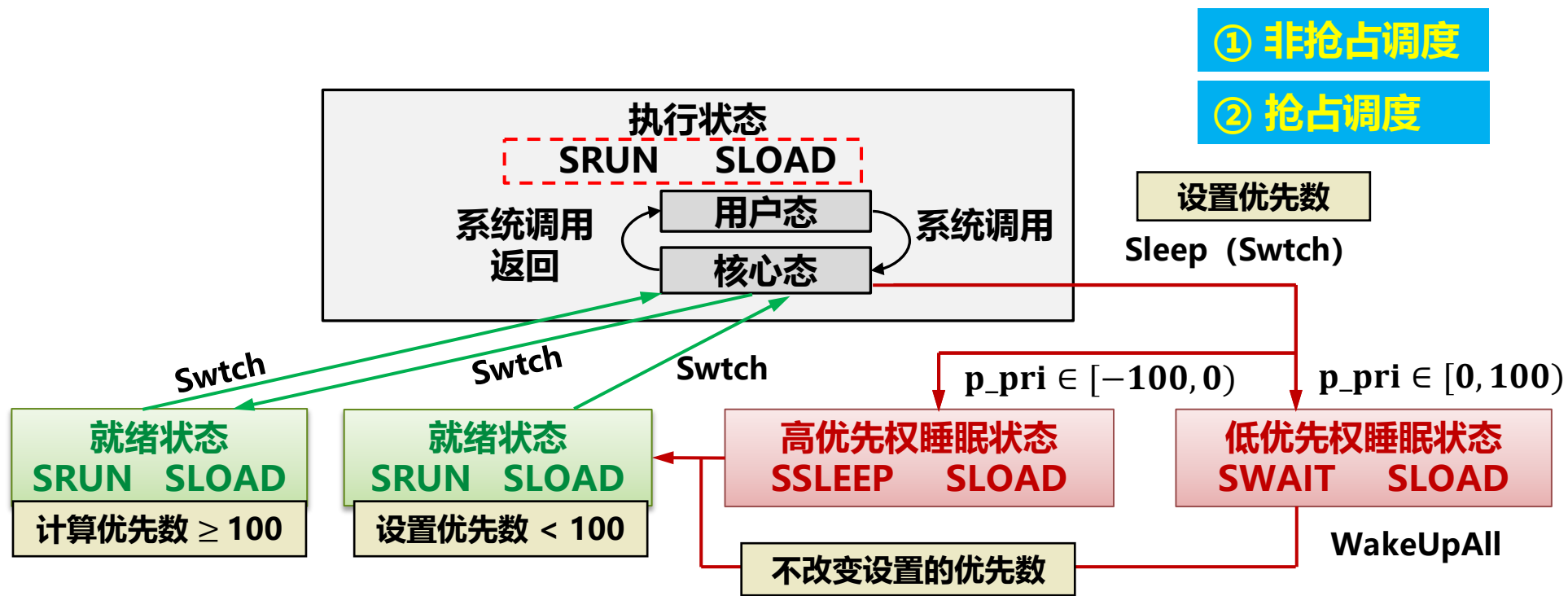




UNIX进程的调度状态



进程的调度状态



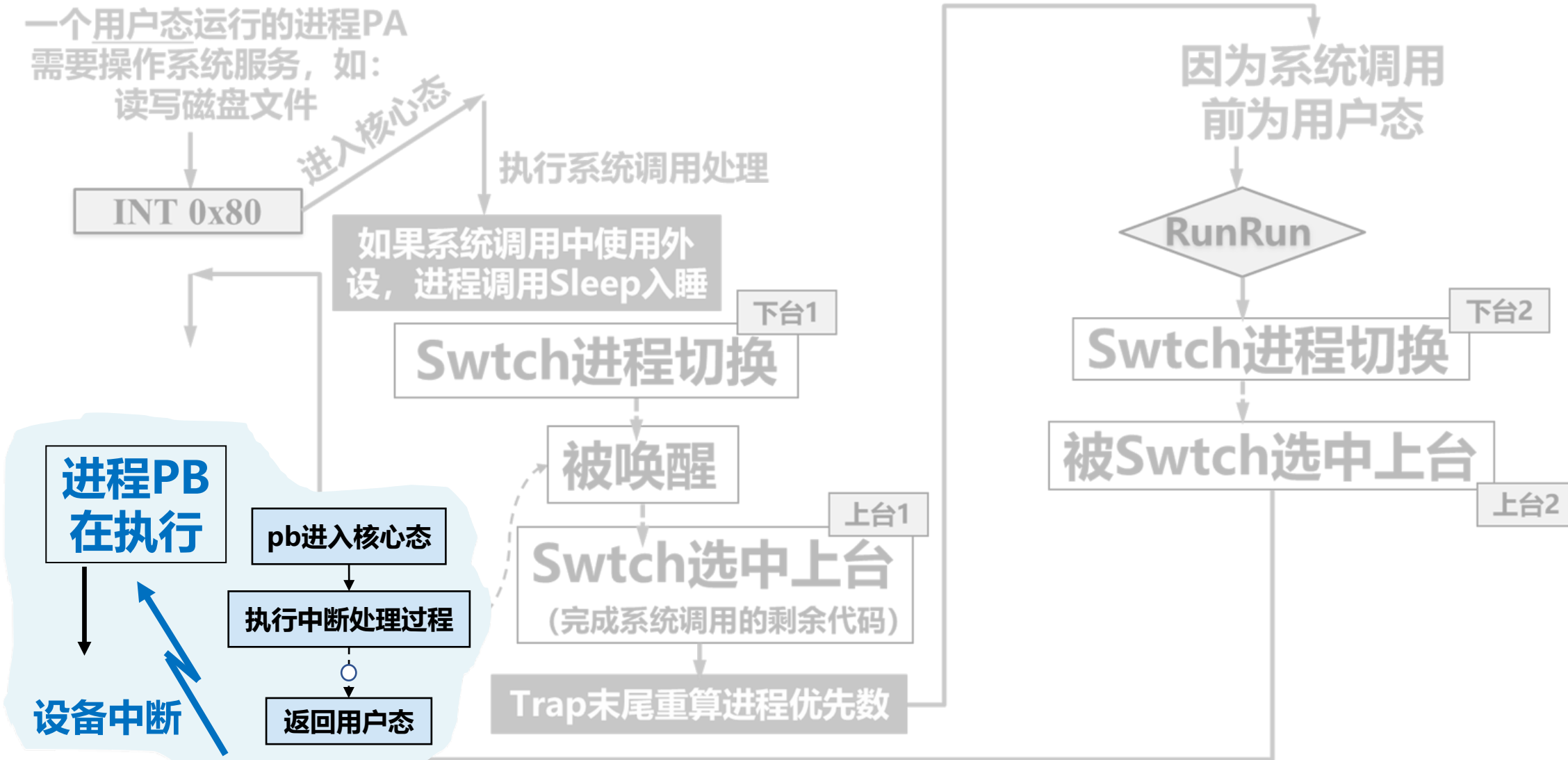
一个完整的进程从用户态进入系统调用, 最终返回用户态的过程



UNIX进程的调度状态



系统调用与中断的关系





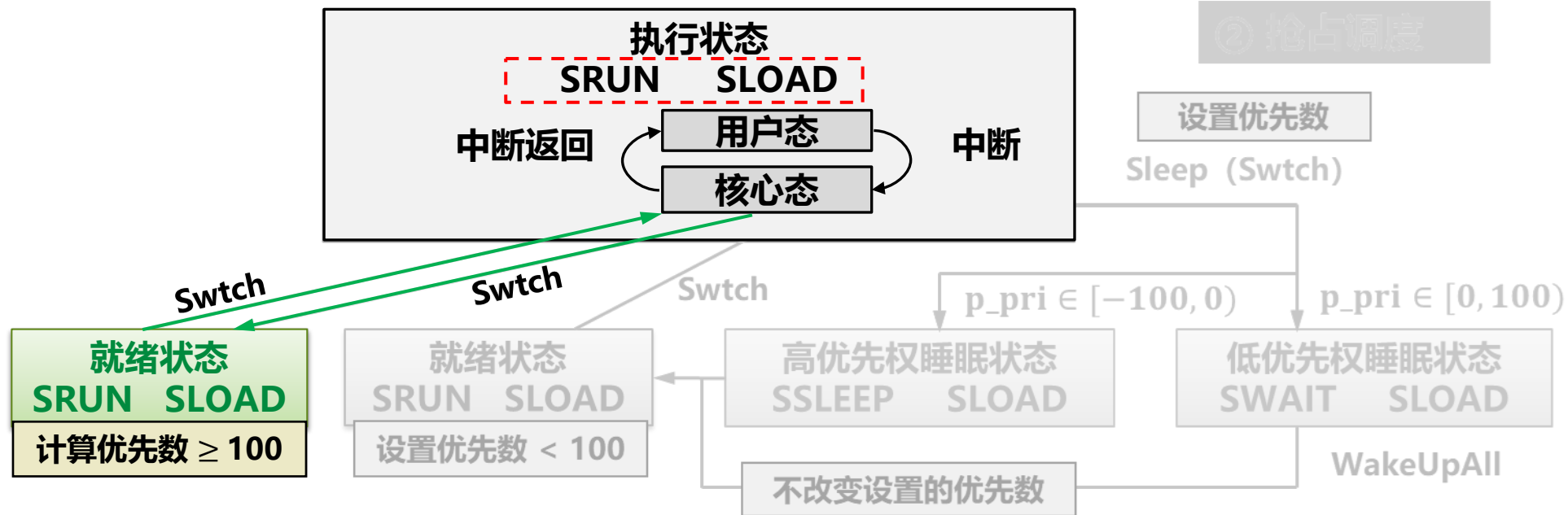
UNIX进程的调度状态



进程的调度状态

① 非抢占调度

② 抢占调度





UNIX进程的调度状态



抢占式

下台2

上台2

执行

就绪

执行

p_stat

SRUN

SRUN

SRUN

p_flag

SLOAD

SLOAD

SLOAD

p_pri

≥ 100

≥ 100

≥ 100

p_wchan

=0

=0

=0

Swch下台

Swch上台



进程未上台执行，核心栈不变

Swch的栈帧

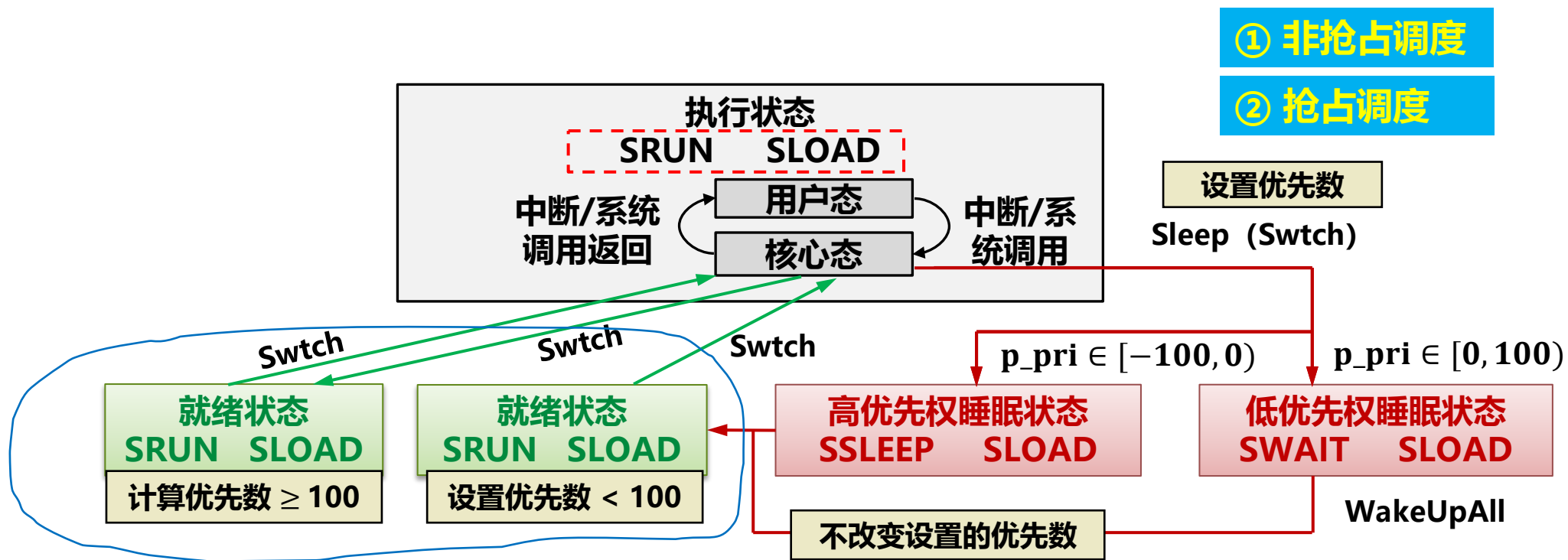
一次中断响应的栈帧



UNIX进程的调度状态



进程的调度状态



关于这两个就绪状态,

Q1: 这两个就绪状态有什么不同?



UNIX进程的调度状态



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

≥ 100

p_wchan

=0

=&内存变量

=0

=0

=0

=0

Sleep修改
Swch下台

WakeUpAll修改

Swch上台

Trap末尾修改

Swch下台

Swch上台

Swch的栈帧

Sleep的栈帧

.....

Trap栈帧

一次中断响应的栈帧

Swch的栈帧

一次中断响应的栈帧

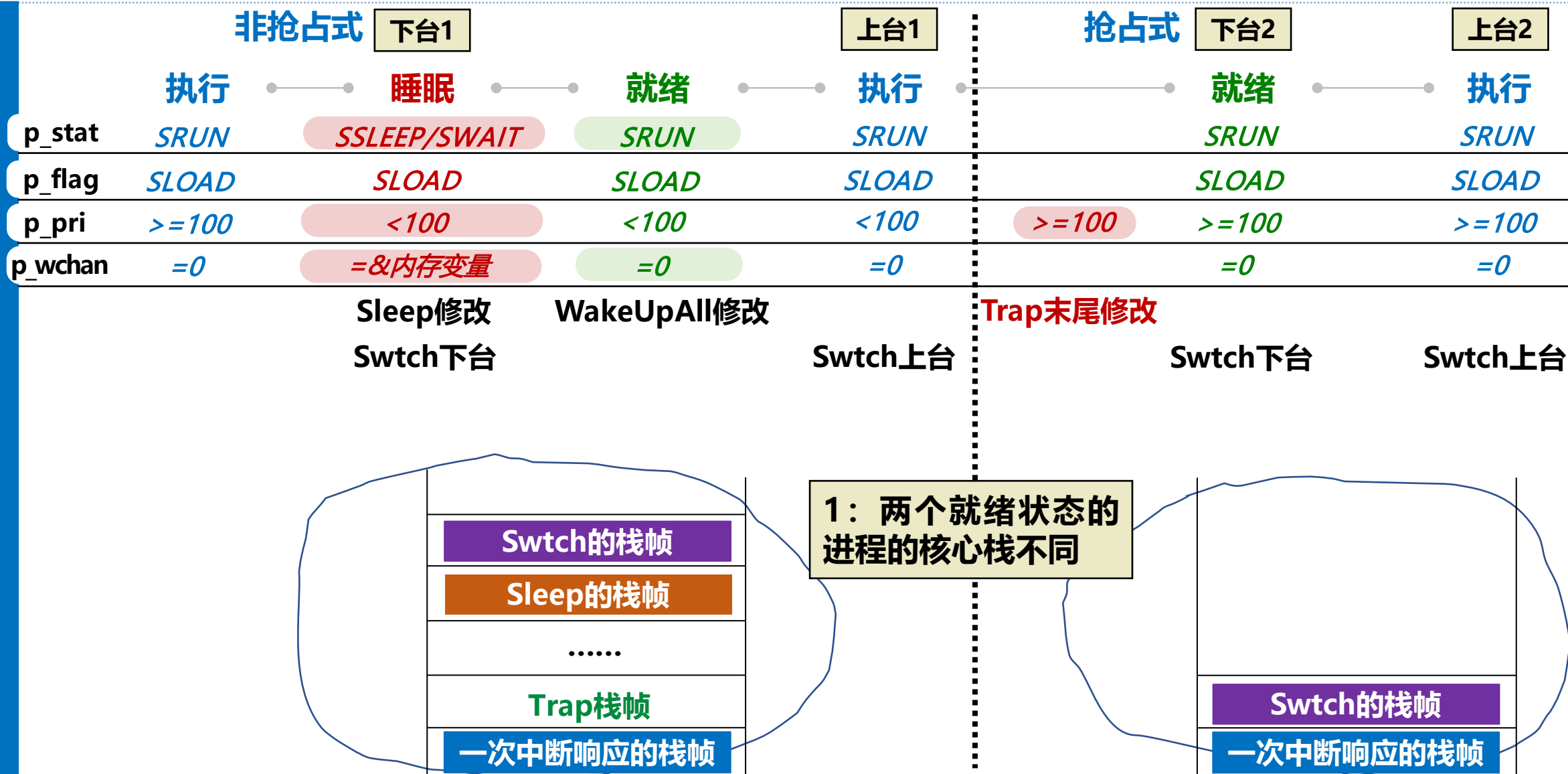
进程的调度状态



UNIX进程的调度状态



进程的调度状态

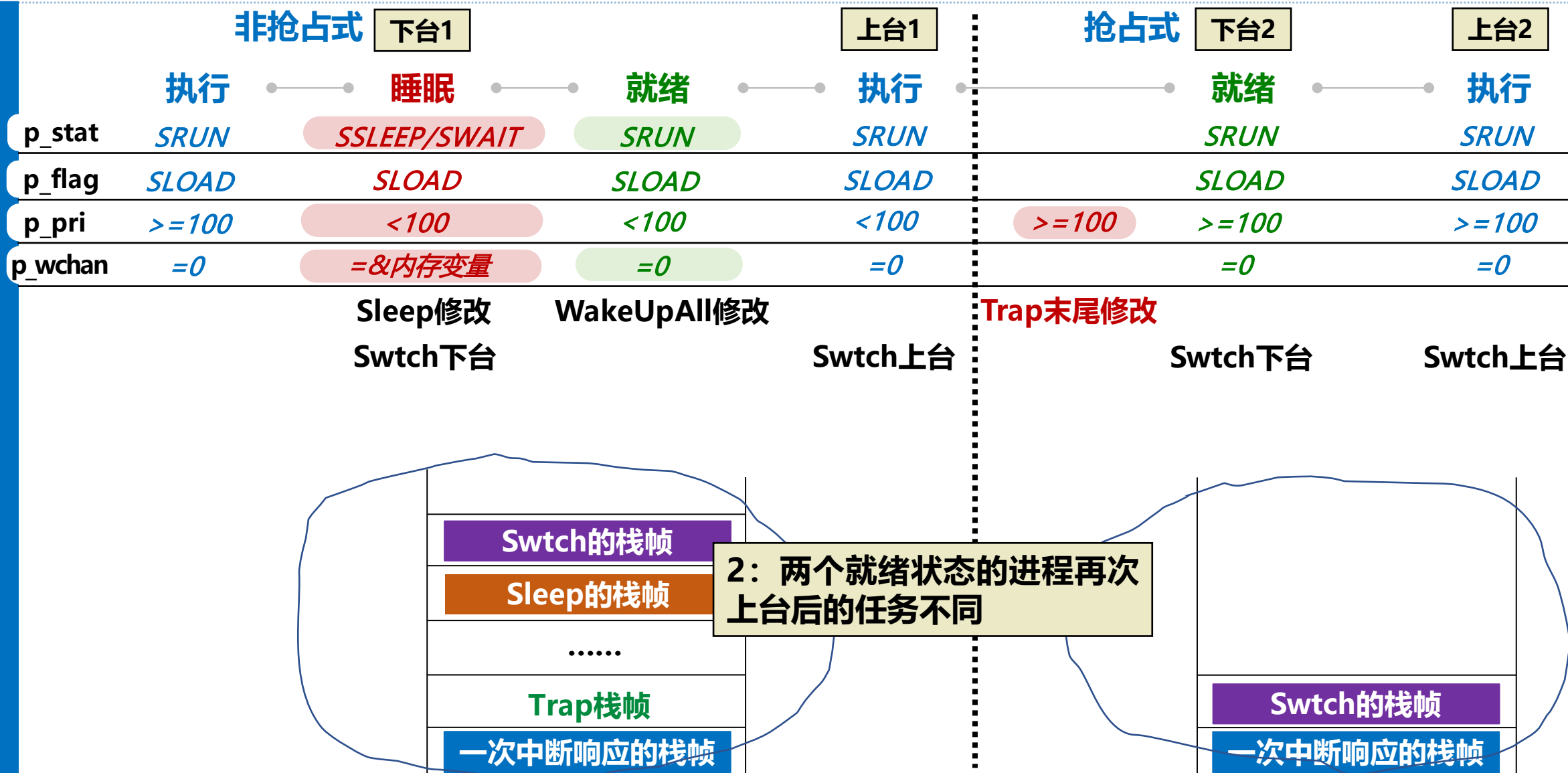




UNIX进程的调度状态



进程的调度状态

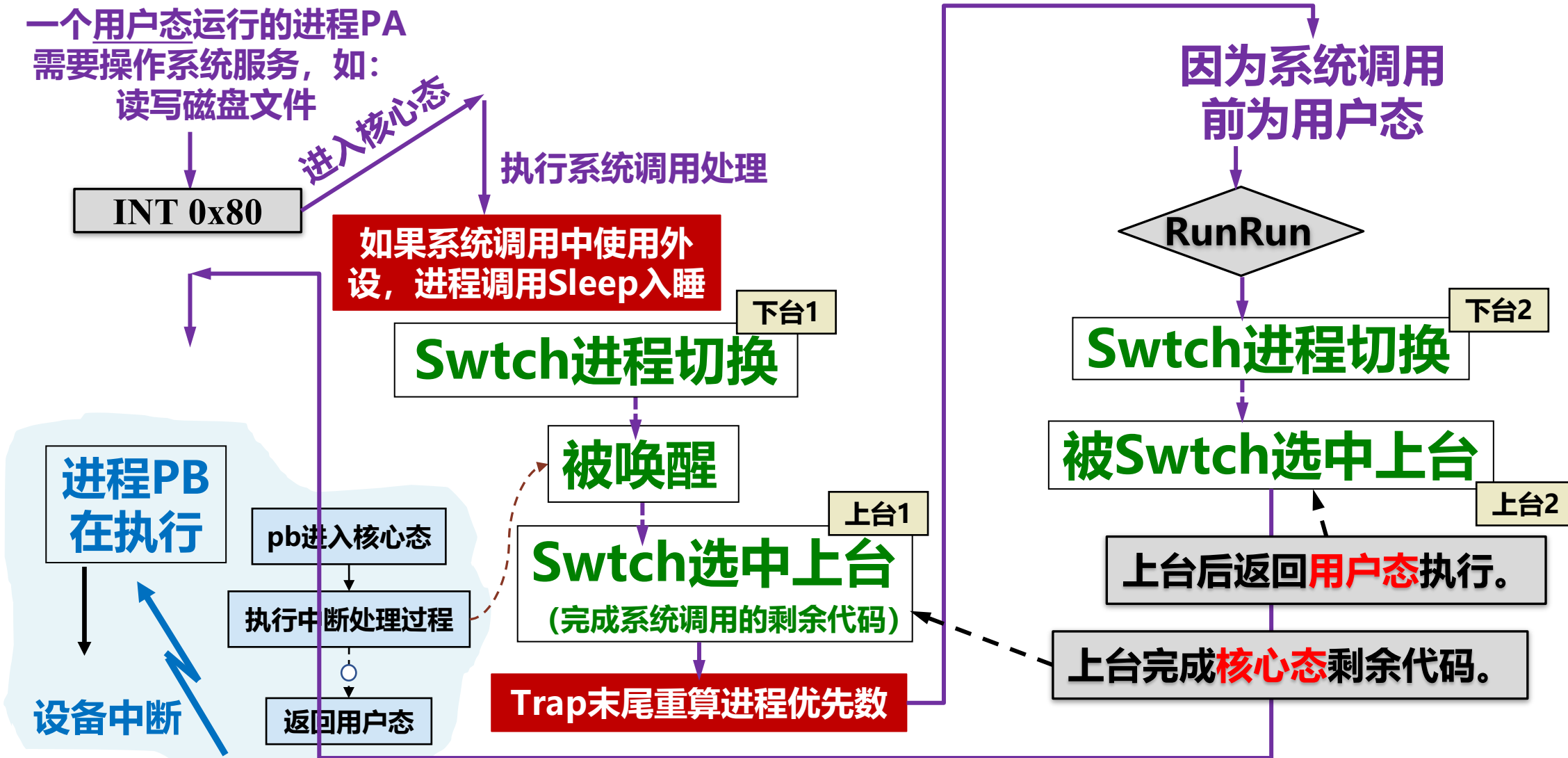




UNIX进程的调度状态



进程的调度状态

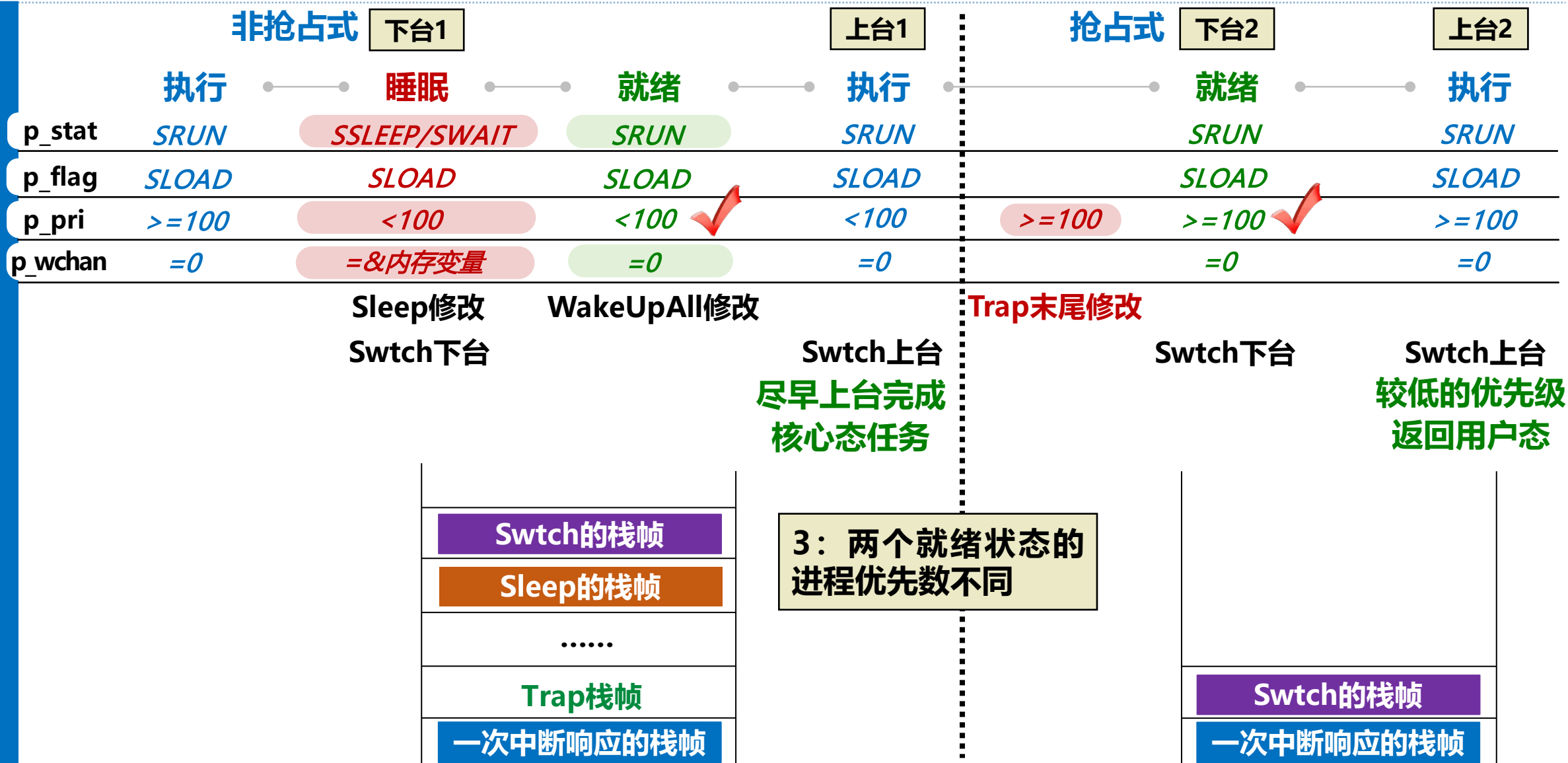




UNIX进程的调度状态



进程的调度状态

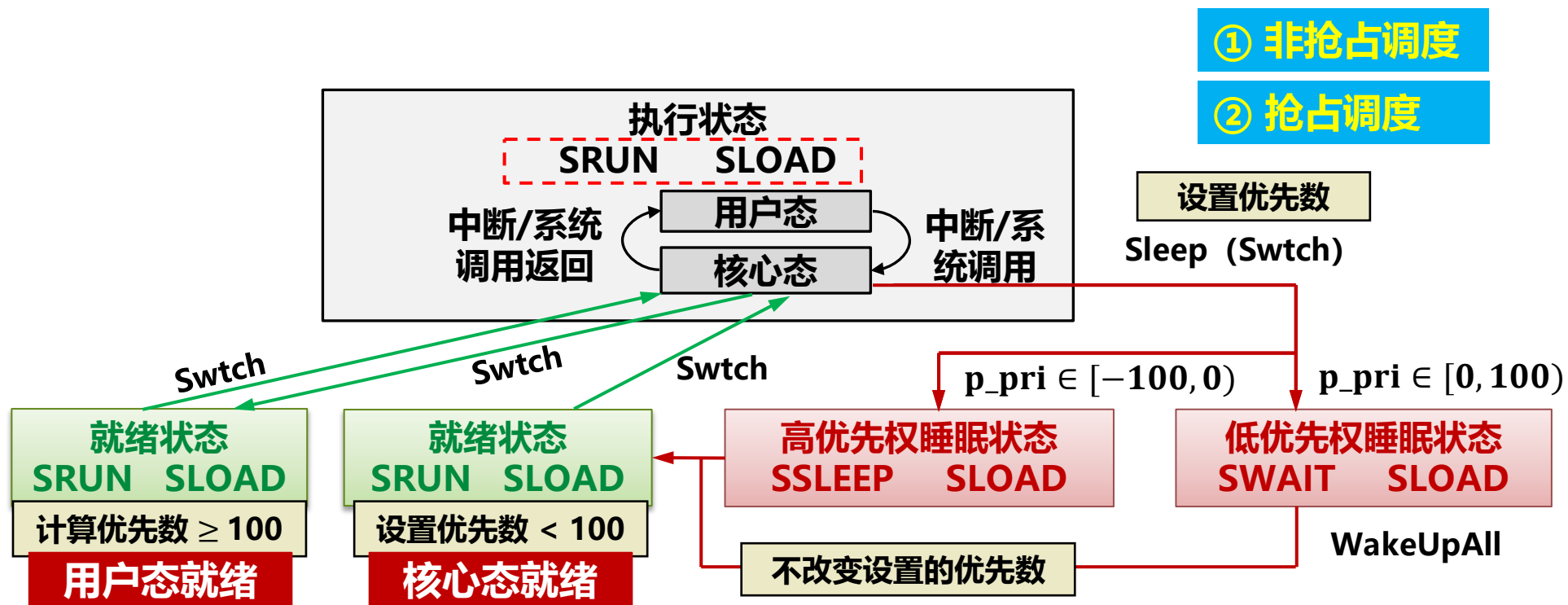




UNIX进程的调度状态



进程的调度状态





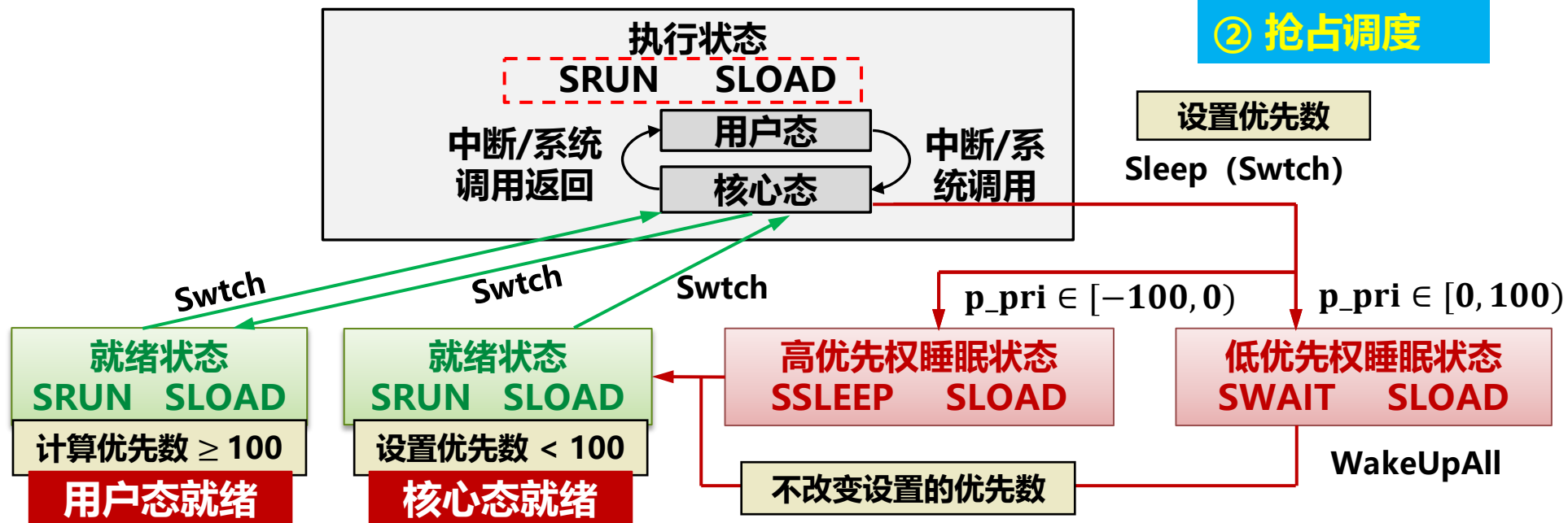
UNIX进程的调度状态



进程的调度状态

① 非抢占调度

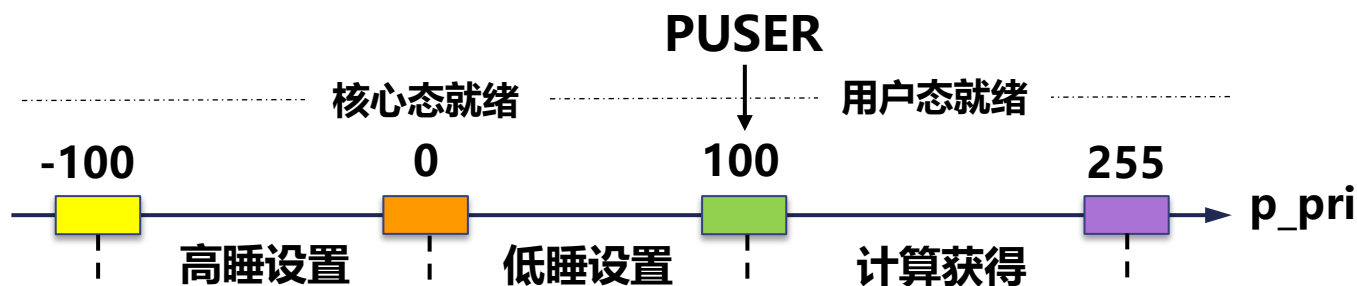
② 抢占调度



$p_pri > PUSER$



1. 整数秒, 重算部分进程的优先数





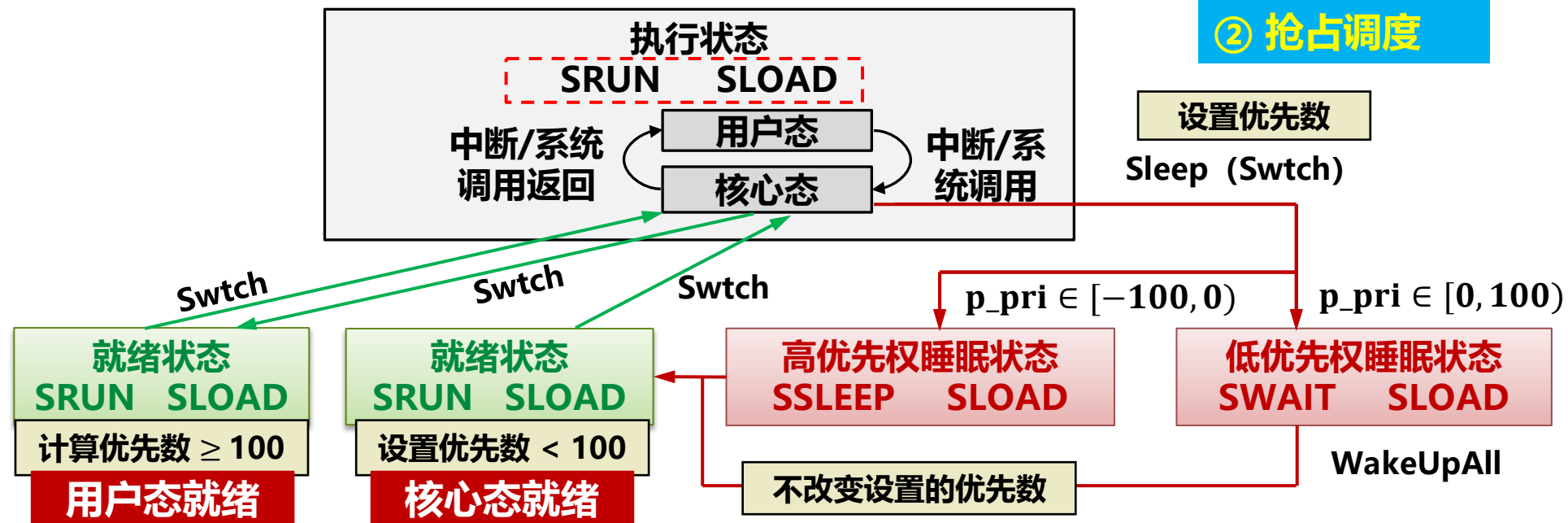
UNIX进程的调度状态



进程的调度状态

① 非抢占调度

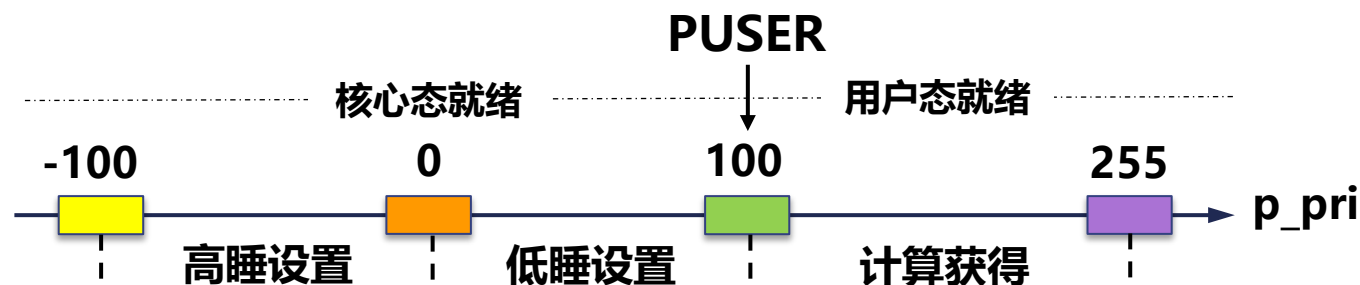
② 抢占调度



$p_pri > PUSER$



1. 整数秒, 重算所有 $p_pri > PUSER$ 进程的优先数

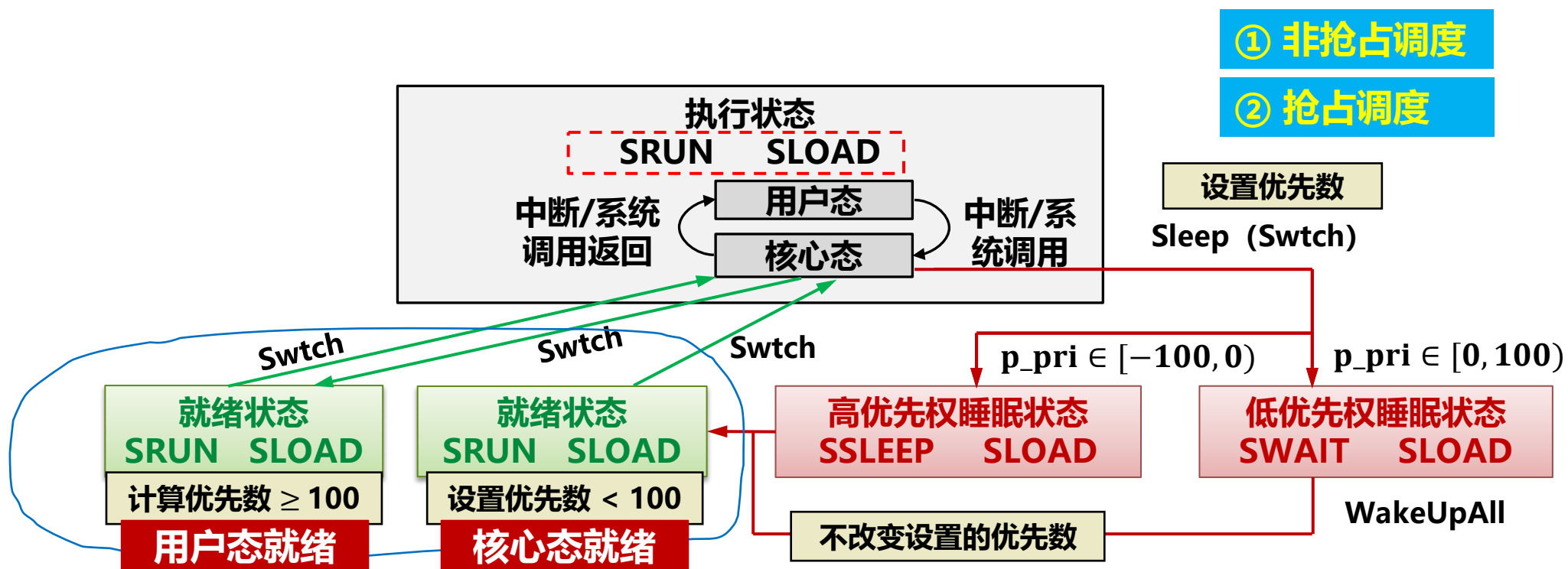




UNIX进程的调度状态



进程的调度状态



关于这两个就绪状态,

Q2: 进程进入这两个就绪状态的原因有什么不同?



UNIX进程的调度状态



进程的调度状态





UNIX进程的调度状态



进程的调度状态





UNIX进程的调度状态



1. 重算优先数时，对RunRun的修改：



1. 整数秒，重算所有用户态就绪的进程的优先数



2. 整数秒，重算现运行进程的优先数



3. 系统调用末尾，重算现运行进程的优先数

Setpri函数完成进程优先数的计算

```
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;
}
```

计算进程优先数

如果算得的优先数 > 现运行进程上台时的优先数

RunRun标志位被设置



重算优先数



UNIX进程的调度状态



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



3. 系统调用末尾，重算现运行进程的优先数

Trap末尾修改

Setpri有两个目的:

1. 刷掉核心态下的优先级，恢复计算获得的优先数
2. 现运行进程优先级下降，设置RunRun

重算优先数



UNIX进程的调度状态



1. 整数秒，重算所有 p_pri $>$ $PUSER$ 进程的优先数

这里的重算，既重算现运行进程，也重算其他所有的用户态就绪进程

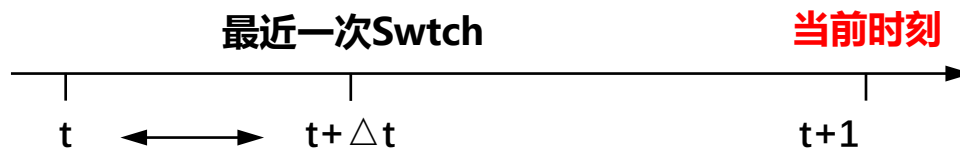
重算优先数



UNIX进程的调度状态

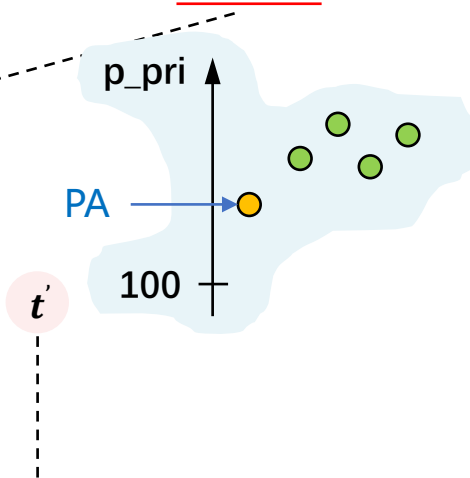


重算优先数



这些进程的优先数:

1. 在 t 时刻时钟中断中获得计算优先数 (此时, p_pri 体现截至到 t 时刻的 p_cpu) ;
2. 在 $[t, t + \Delta t]$ 时刻内曾在核心态下执行Trap末尾重算 (此时, p_pri 体现截至到该时刻的 p_cpu) 。



1. 整数秒, 重算所有 $p_pri > PUSER$ 进程的优先数

这里的重算, 既重算现运行进程, 也重算其他所有的用户态就绪进程

最近一次Swch中, p_pri 最小的进程PA (截止到 $t + \Delta t$, p_pri 最小, 优先级最高的进程) 抢到CPU



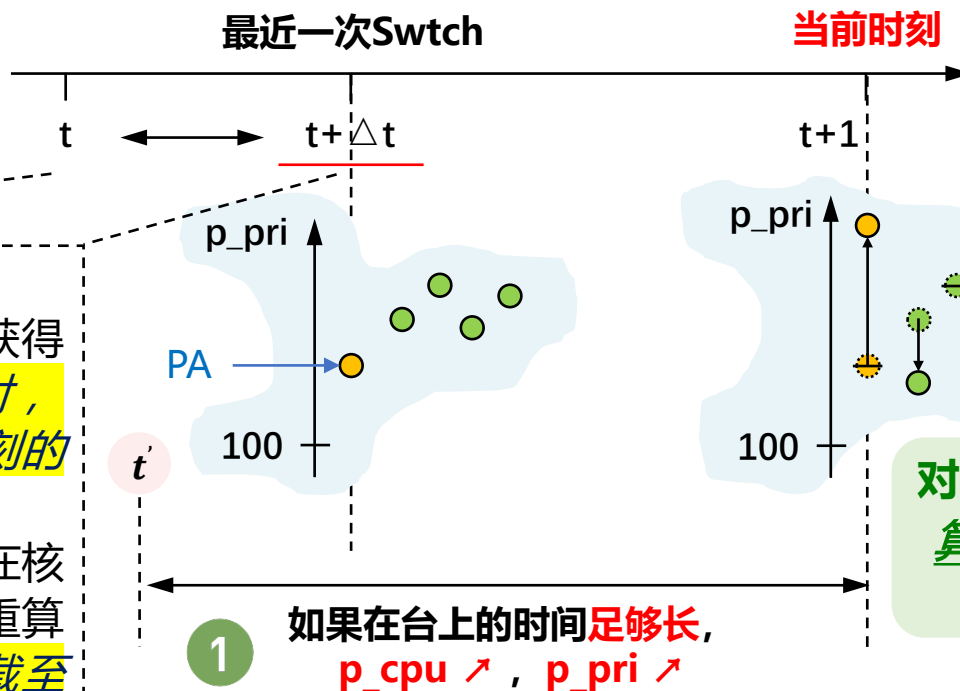
UNIX进程的调度状态



重算优先数

这些进程的优先数:

1. 在 t 时刻时钟中断中获得计算优先数 (此时, p_pri 体现截至到 t 时刻的 p_cpu) ;
2. 在 $[t, t + \Delta t]$ 时刻内曾在核心态下执行Trap末尾重算 (此时, p_pri 体现截至到该时刻的 p_cpu) 。



1. 整数秒, 重算所有 $p_pri > PUSER$ 进程的优先数

这里的重算, 既重算现运行进程, 也重算其他所有的用户态就绪进程

对现运行进程的重算满足:
算得的优先数 $>$ PA上台时的优先数

现运行进程优先级下降
设置RunRun

最近一次Swch中, p_pri 最小的进程PA (截止到 $t + \Delta t$, p_pri 最小, 优先级最高的进程) 抢到CPU



UNIX进程的调度状态



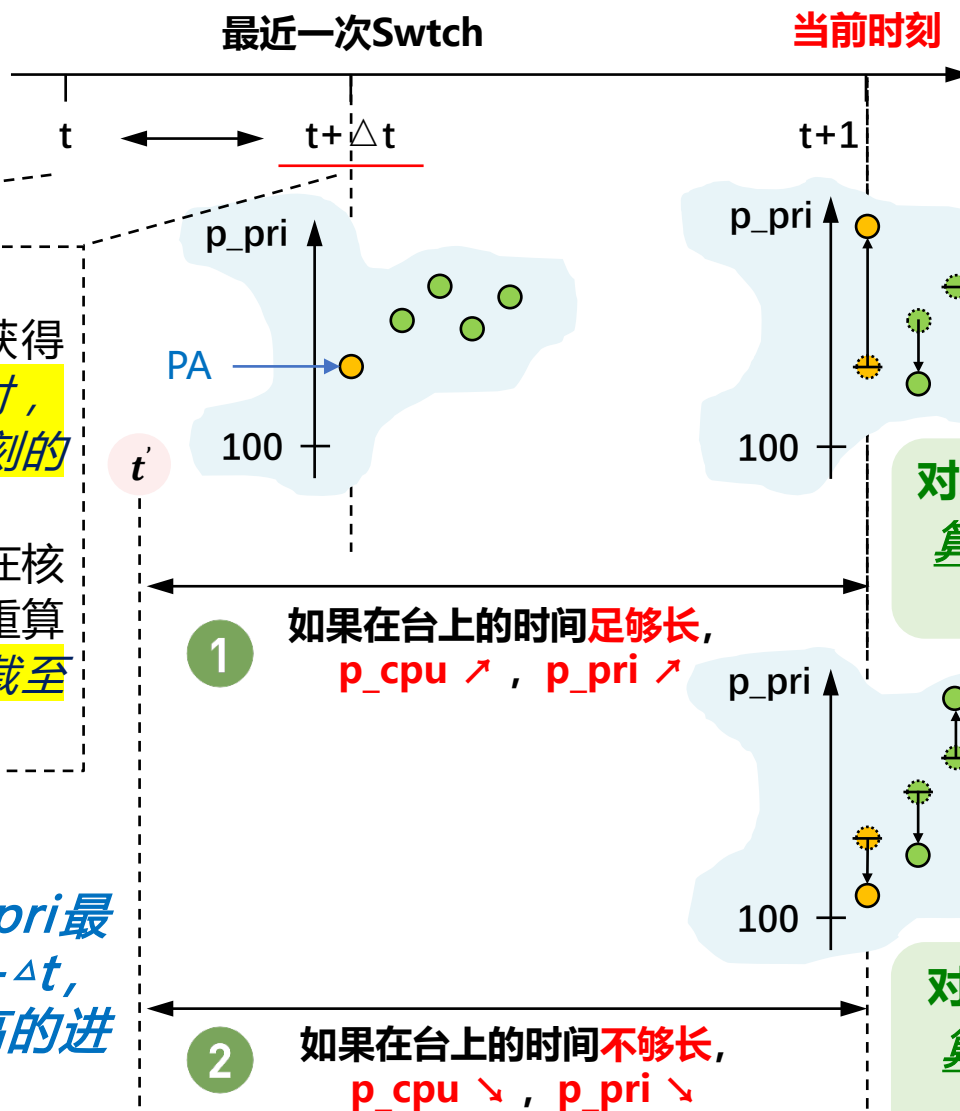
重算优先数

没有核心态就绪的进程
只有用户态就绪的进程

这些进程的优先数:

1. 在 t 时刻时钟中断中获得计算优先数 (此时, p_pri 体现截至到 t 时刻的 p_cpu) ;
2. 在 $[t, t + \Delta t]$ 时刻内曾在核心态下执行Trap末尾重算 (此时, p_pri 体现截至到该时刻的 p_cpu) 。

最近一次Swch中, p_pri 最小的进程PA (截止到 $t + \Delta t$, p_pri 最小, 优先级最高的进程) 抢到CPU



1. 整数秒, 重算所有 $p_pri > PUSER$ 进程的优先数

这里的重算, 既重算现运行进程, 也重算其他所有的用户态就绪进程

对现运行进程的重算满足:
算得的优先数 $>$ PA上台时的优先数

现运行进程优先级下降
设置RunRun

对其他进程的重算满足:
算得的优先数 $>$ PA上台时的优先数

1秒计时到
设置RunRun



UNIX进程的调度状态



2. 整数秒，重算现运行进程的优先数

开始繁琐耗时的事务处理



1. 维护系统时间: `Time::lbolt -60`; 系统时间+1秒;
2. 开中断, 向中断控制器发送EOI指令;
3. 唤醒所有延时睡眠的进程;
4. 对所有进程 `p_time++`;
5. 所有进程 `p_cpu = max(0, p_cpu - SCHMAG)`;
6. 重算部分进程的优先数;
7. 如果RunIn被设置, 唤醒0#进程;
8. 现运行进程进行信号处理;
9. 重算当前进程的优先数。

可能花费较长时间
给现运行进程多一次重算优先数的机会

重算优先数



UNIX进程的调度状态



1. 重算优先数时，对RunRun的修改：



1. 整数秒，重算所有用户态就绪的进程的优先数



2. 整数秒，重算现运行进程的优先数



3. 系统调用末尾，重算现运行进程的优先数

Setpri函数完成进程优先数的计算

```
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;
}
```

计算进程优先数

如果算得的优先数 > 现运行进程上台时的优先数

RunRun标志位被设置



重算优先数



UNIX进程的调度状态



重算优先数

1. 重算优先数时，对RunRun的修改：



1. 整数秒，重算所有用户态就绪的进程的优先数



2. 整数秒，重算现运行进程的优先数



3. 系统调用末尾，重算现运行进程的优先数

Setpri函数完成进程优先数的计算

计算进程优先数

如果算得的优先数 > 现运行进程上台时的优先数

RunRun标志位被设置

$$p_pri = \min \{ 255, (p_cpu / 16 + PUSER + p_nice) \}$$

1秒计时到

现运行进程优先级下降

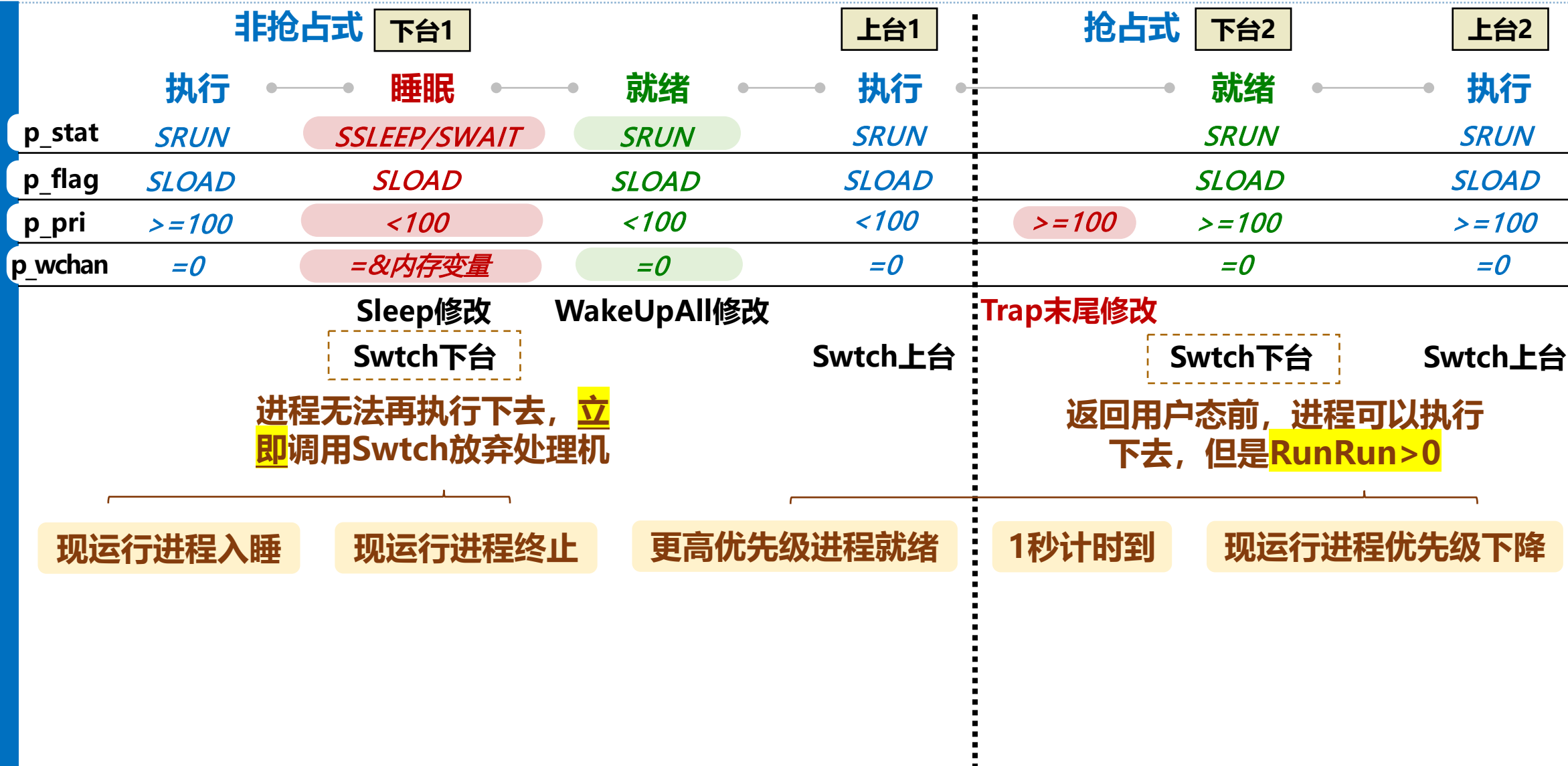
现运行进程可能已经不适合继续在CPU上执行了



UNIX进程的调度状态



进程的调度状态

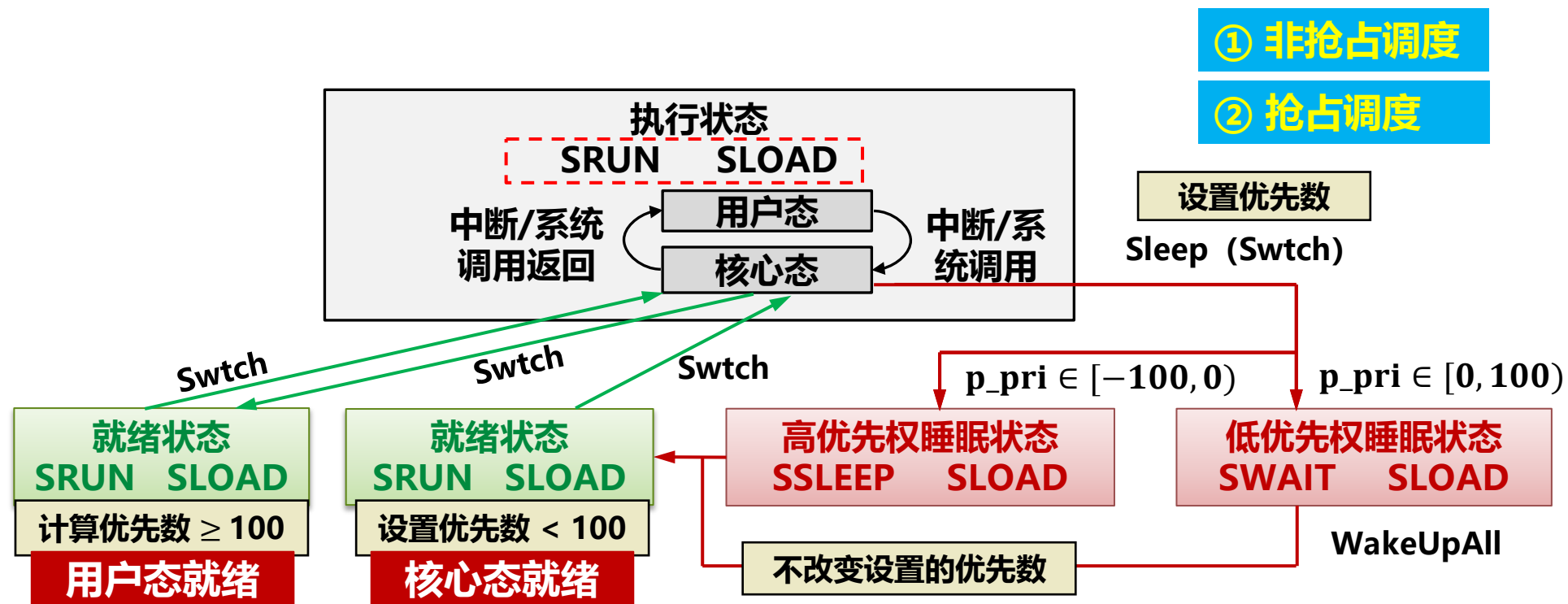




UNIX进程的调度状态



进程的调度状态



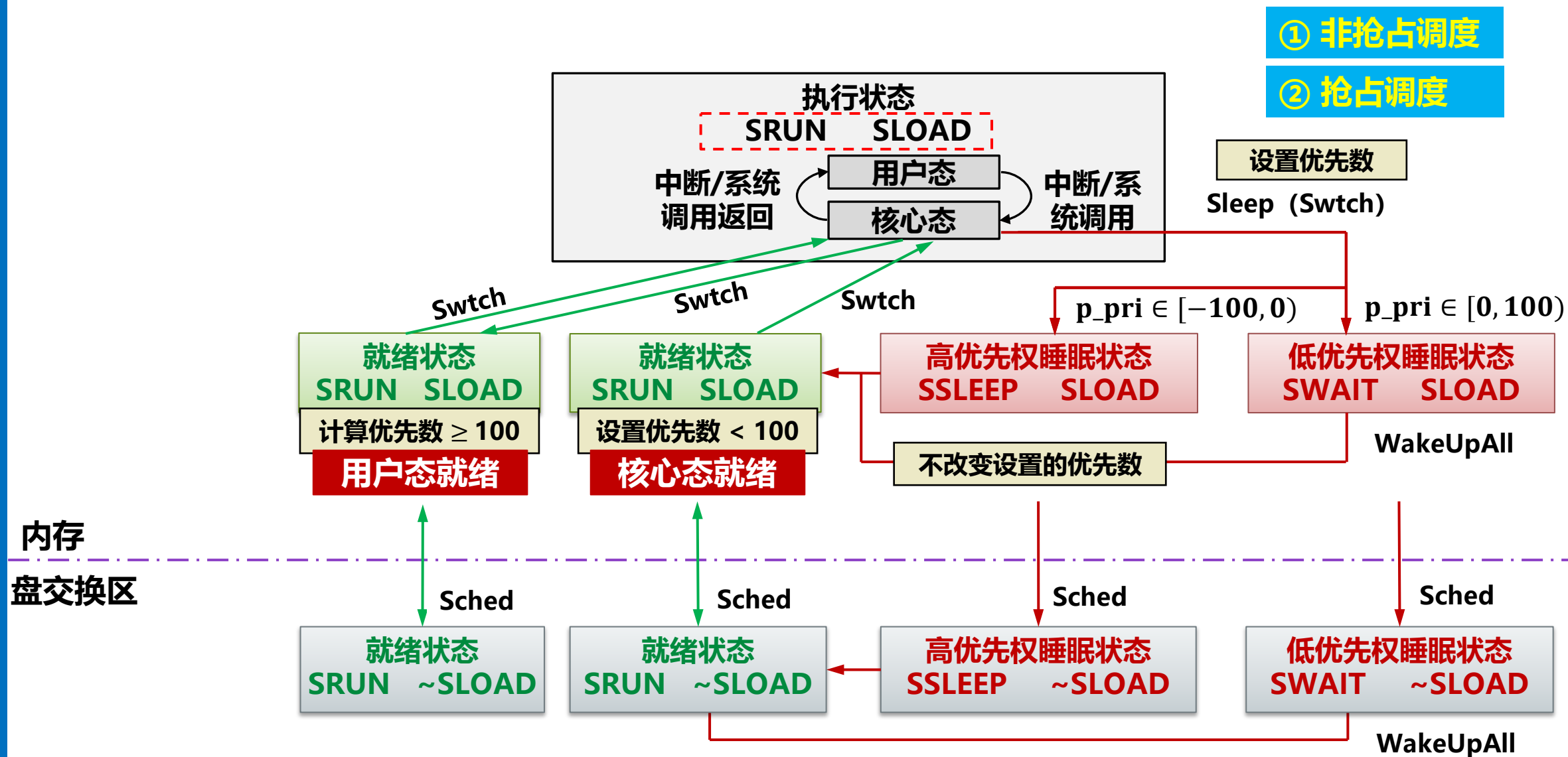
如果内存不足以容纳所有的进程图象？？？



UNIX进程的调度状态



进程的调度状态



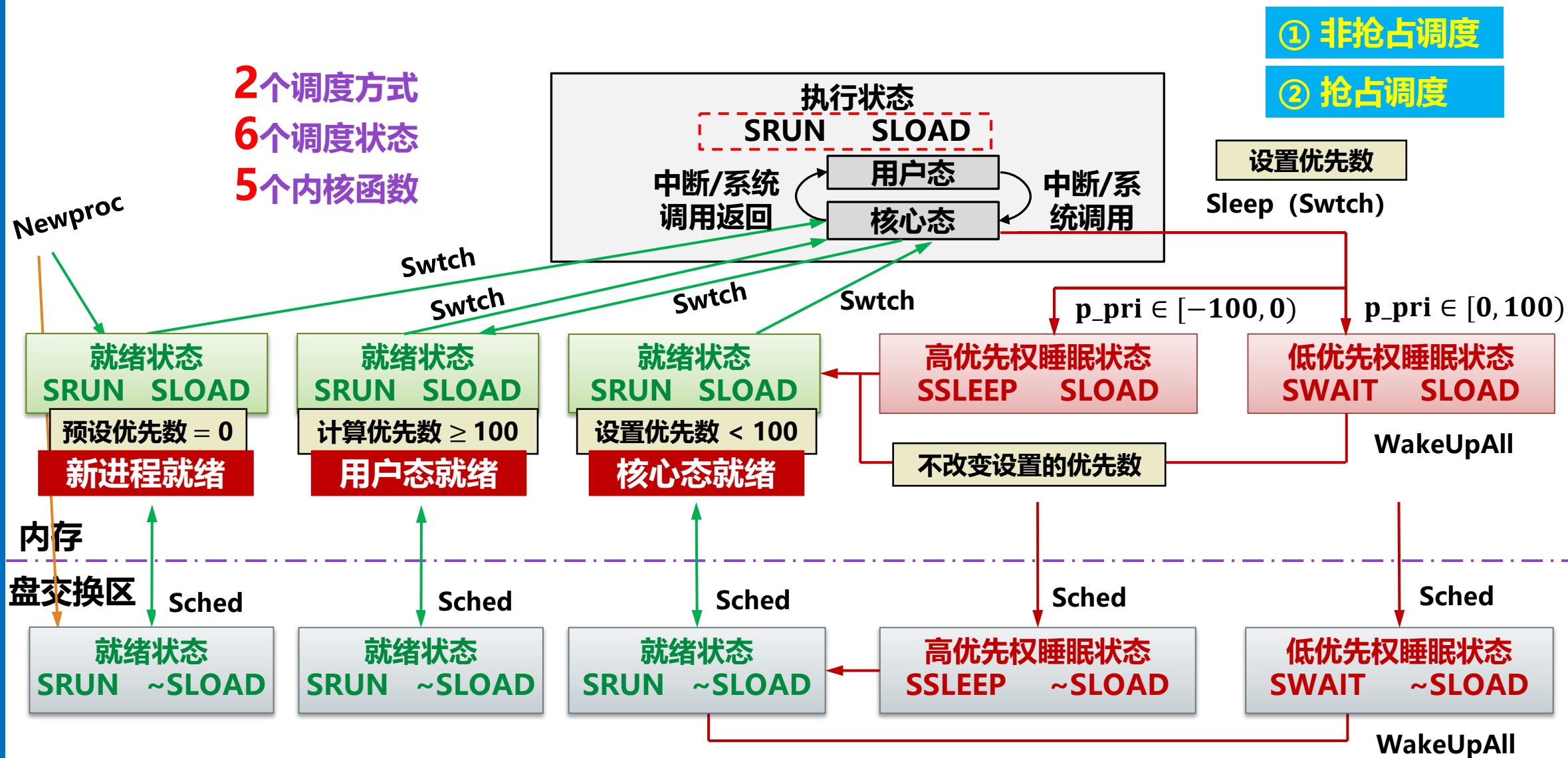


UNIX进程的调度状态



进程的调度状态

2个调度方式
6个调度状态
5个内核函数





本节小结



- 1 **UNIX进程调度状态及变化条件**
- 2 **UNIX进程控制涉及的主要内核函数**

请阅读教材：166页 ~ 171页
