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

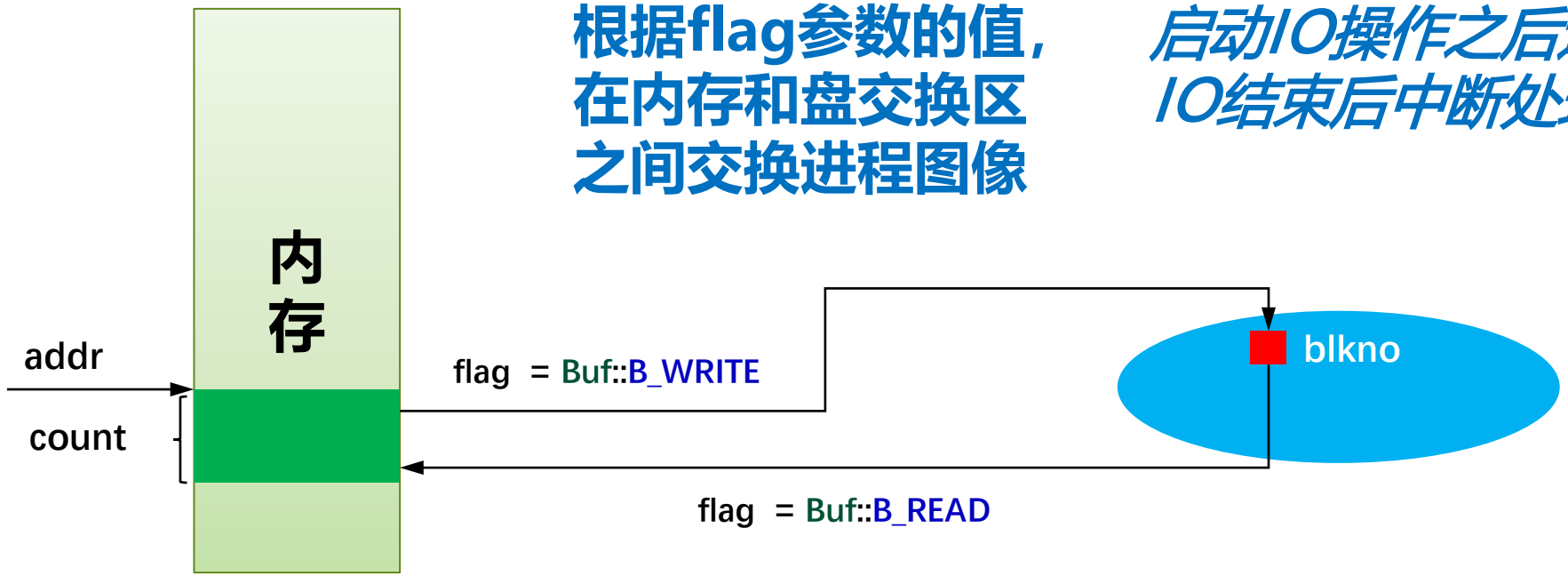
- 进程切换调度
- 进程创建与终止
- 进程图像的交换

进程图像交换

```
bool BufferManager::Swap(
int blkno,                  / * blkno: 交换区中盘块号 */
unsigned long addr,         / * addr:  内存起始地址 */
int count,                  / * count: 进行传输字节数，byte为单位 */
enum Buf::BufFlag flag      / * flag: 内存->交换区 or 交换区->内存 */
)
```

根据flag参数的值，在内存和盘交换区之间交换进程图像

启动IO操作之后进程入睡等待，IO结束后中断处理中进程被唤醒



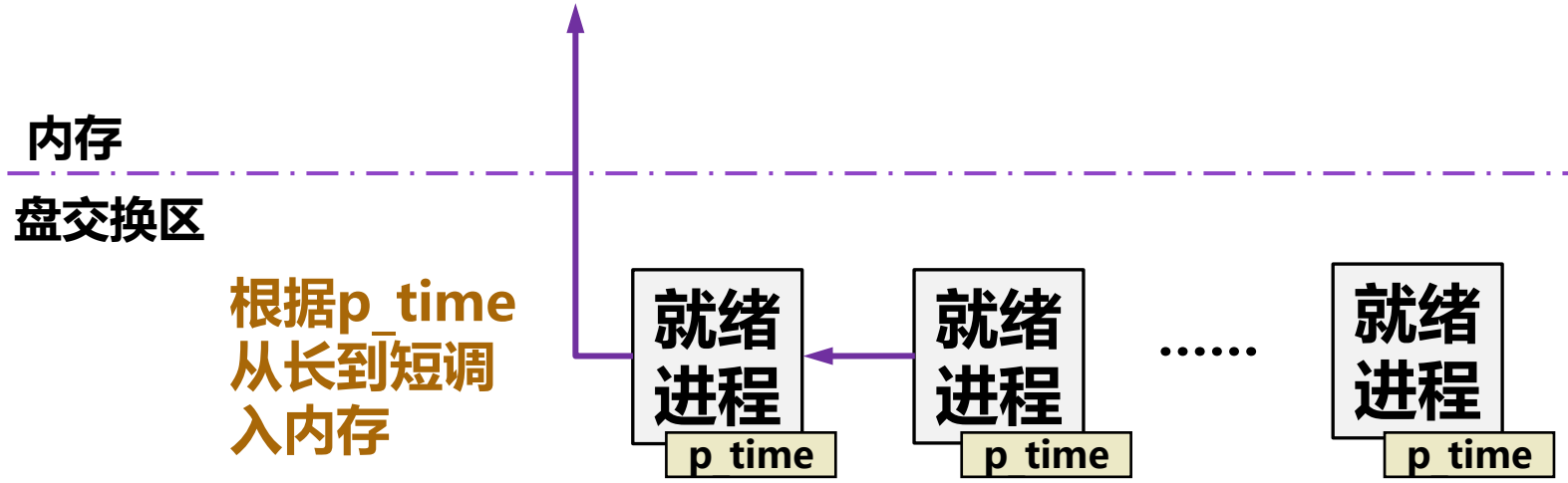
进程图像交换

1　当内存空间不足以装下所有就绪进程时……

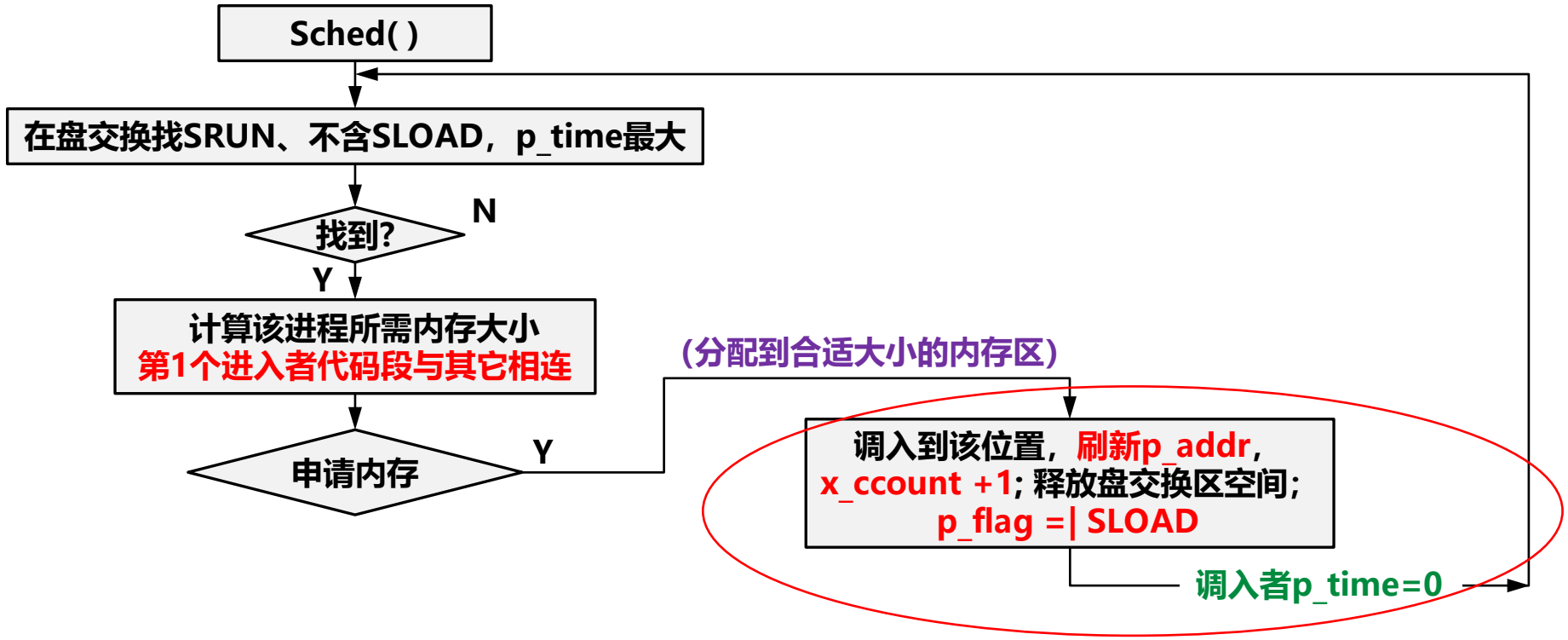
内存空间不足时的图像交换

0# 进程执行ProcessManager::Sched

每秒时钟中断：对所有进程 p_time ++;

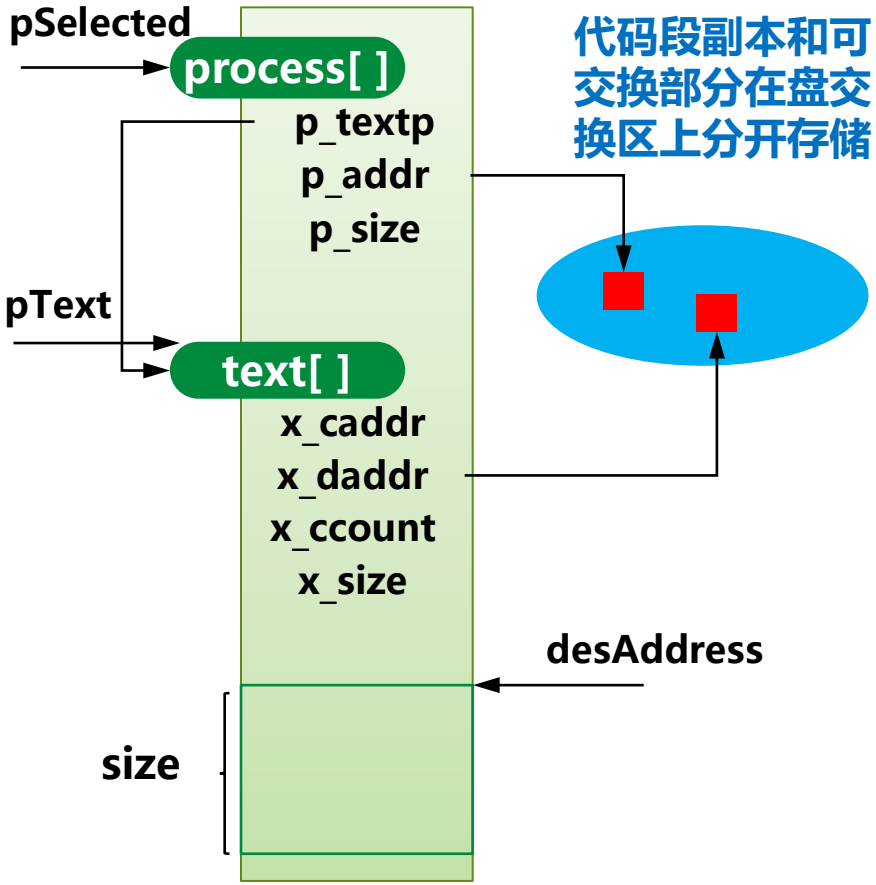


内存空间不足时的图像交换



内存空间不足时的图像交换

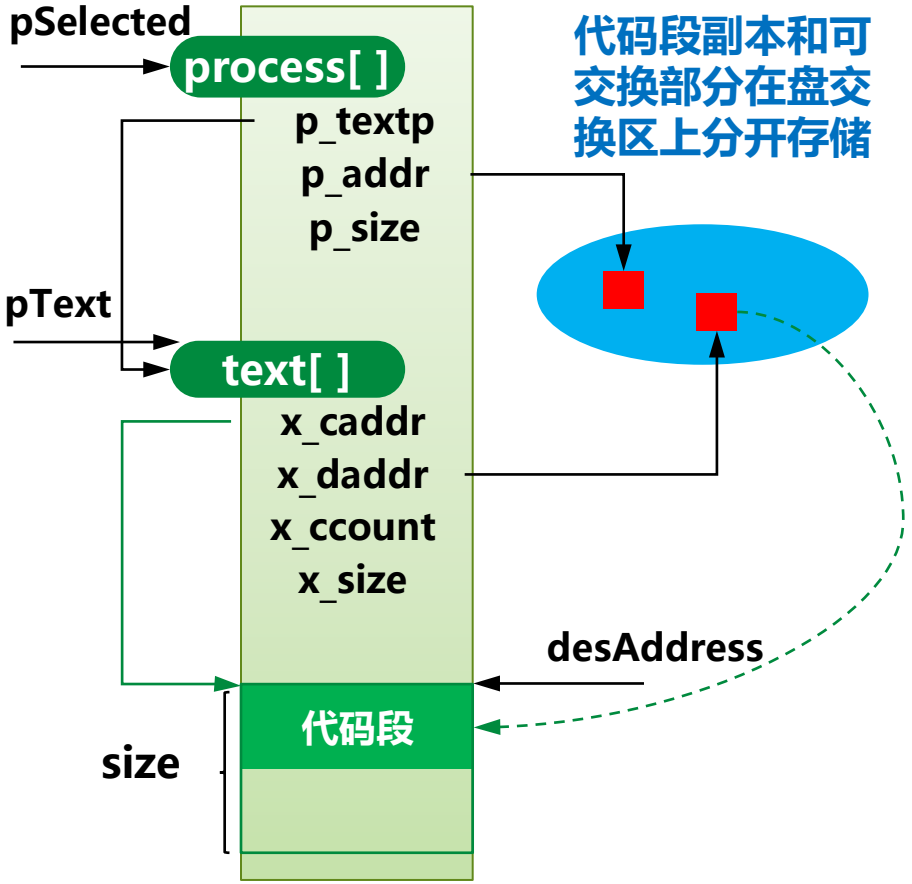
1. 进程图像的换进

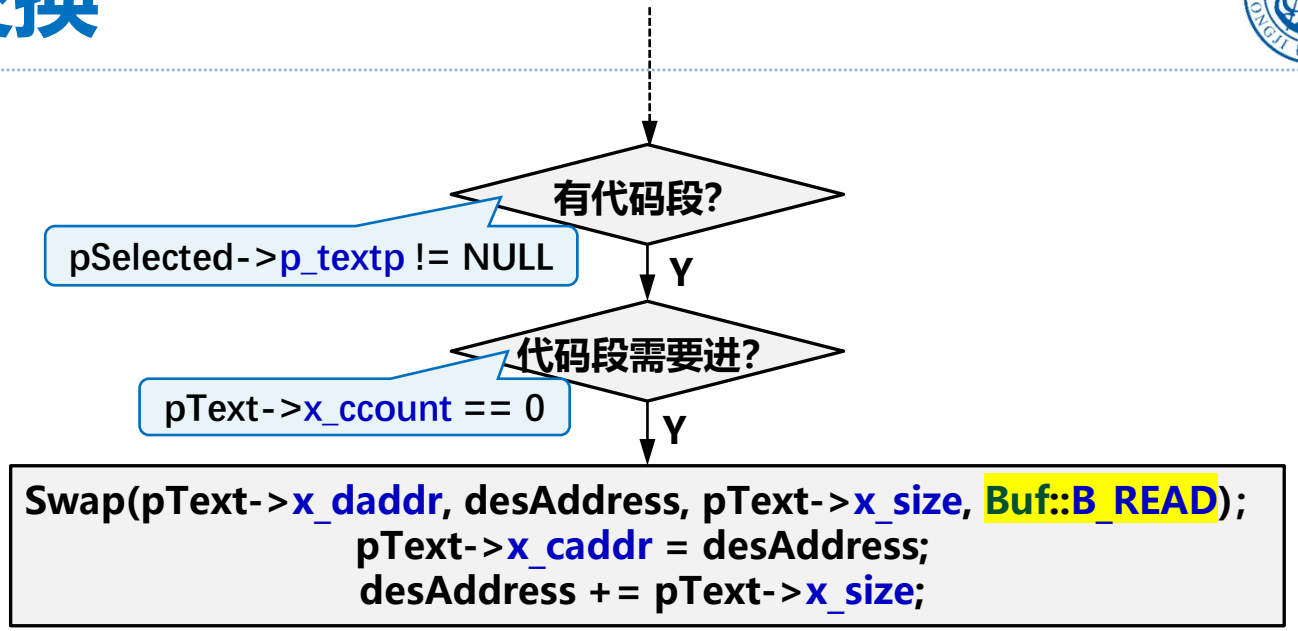


成功分配到内存。如果代码段需要进内存，
size = 代码段长度 + 可交换部分长度

内存空间不足时的图像交换

1. 进程图像的换进

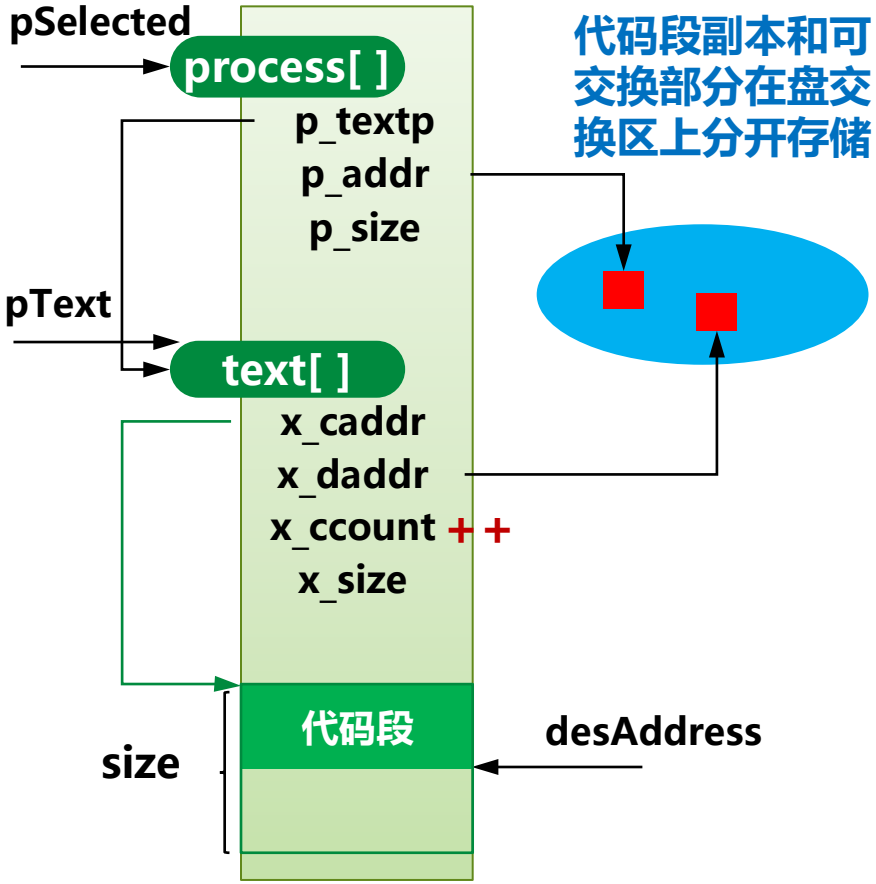


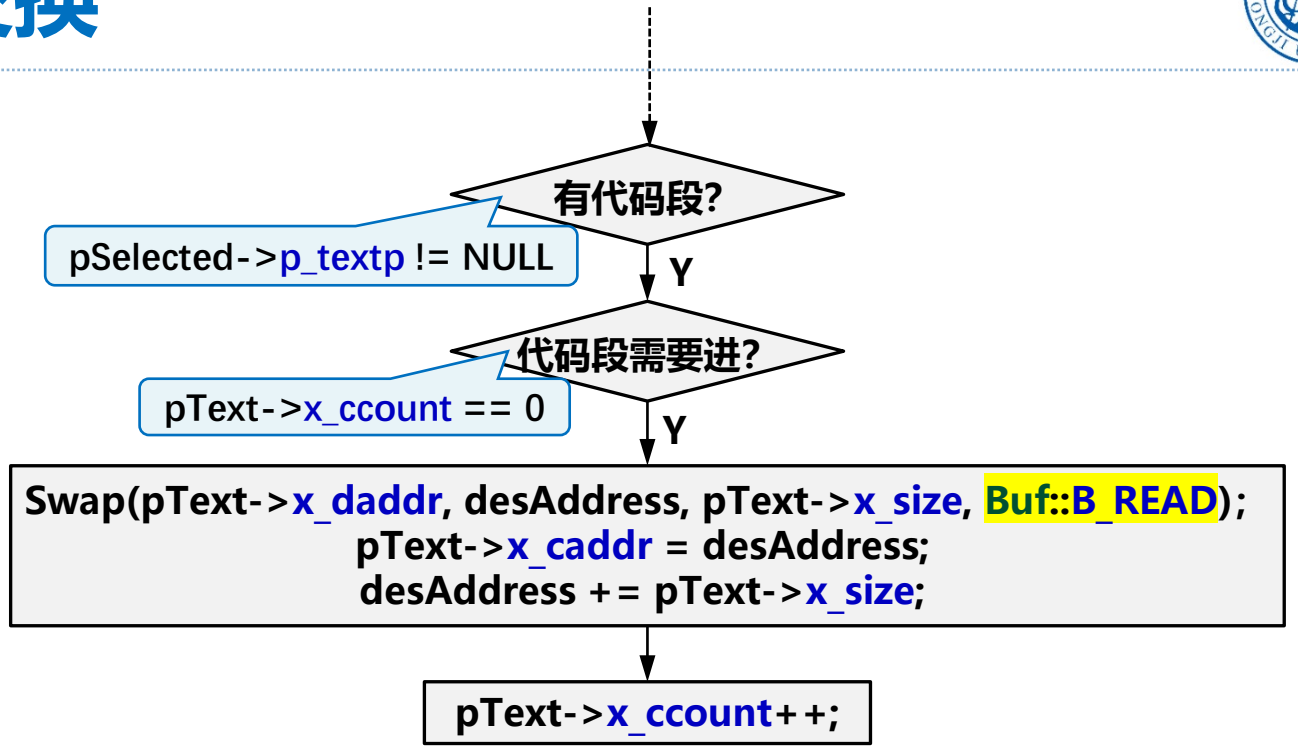


成功分配到内存。如果代码段需要进内存，
size = 代码段长度 + 可交换部分长度

内存空间不足时的图像交换

1. 进程图像的换进

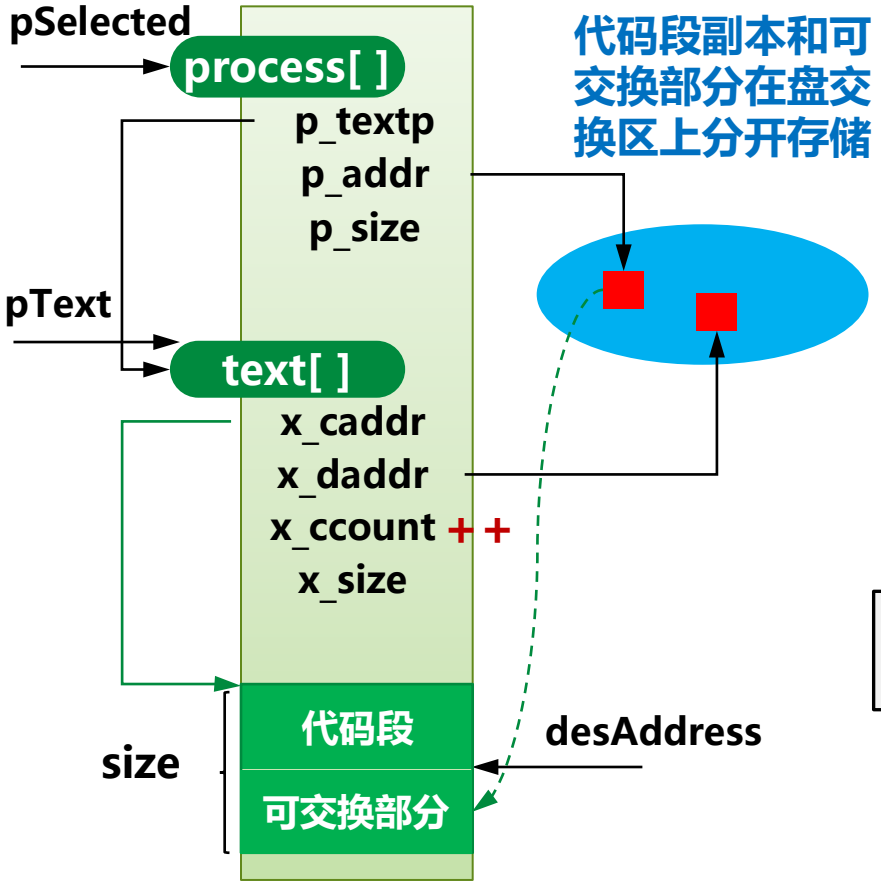


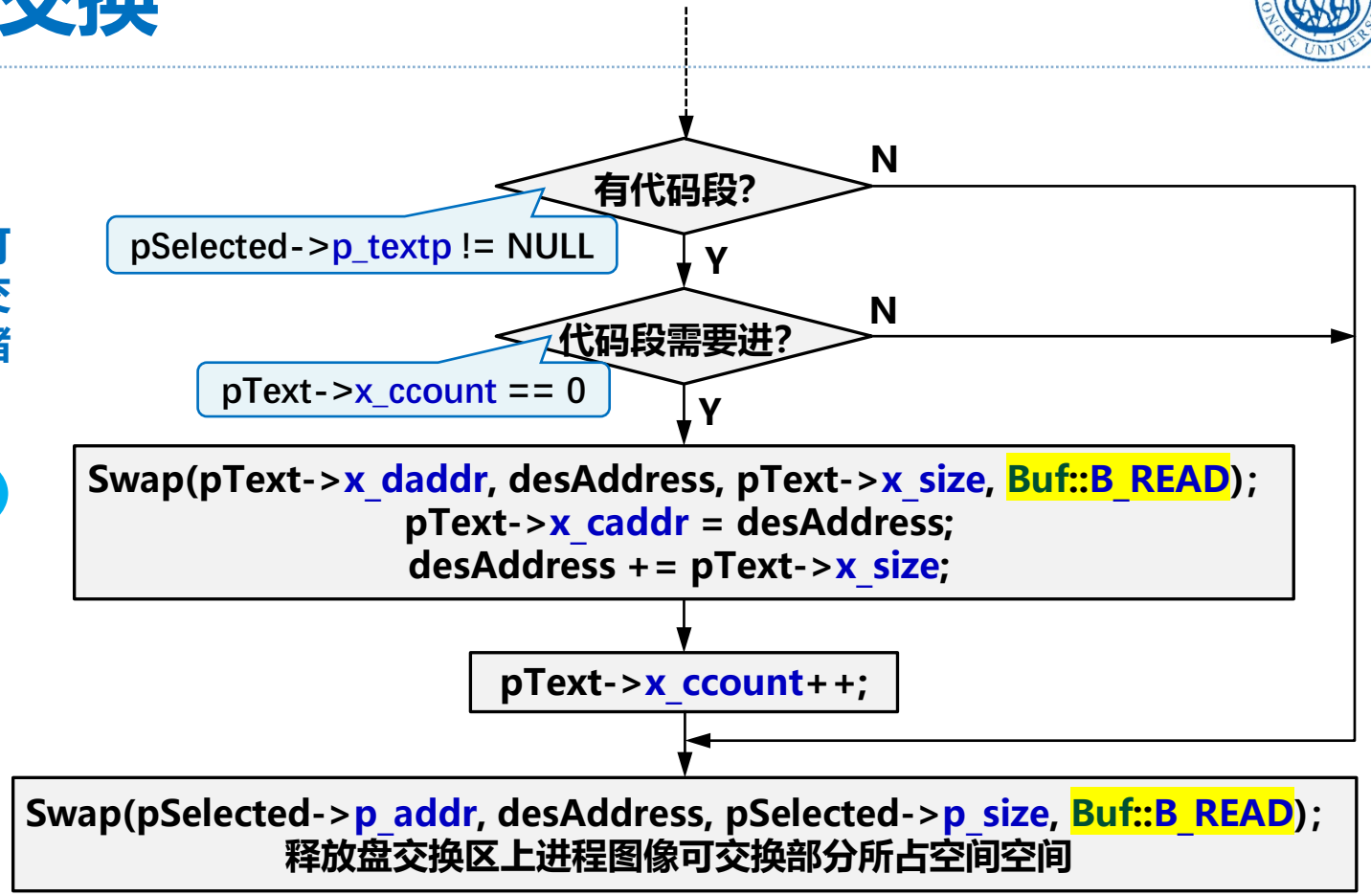


成功分配到内存。如果代码段需要进内存，
size = 代码段长度 + 可交换部分长度

内存空间不足时的图像交换

1. 进程图像的换进

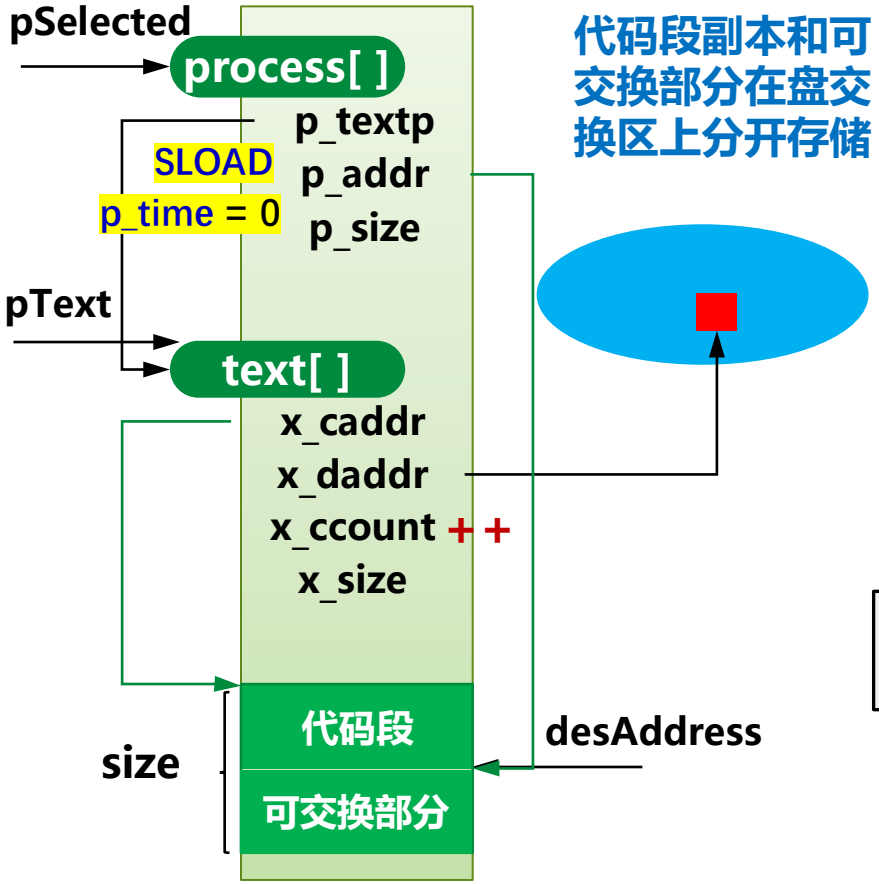




成功分配到内存。如果代码段需要进内存，
size = 代码段长度 + 可交换部分长度

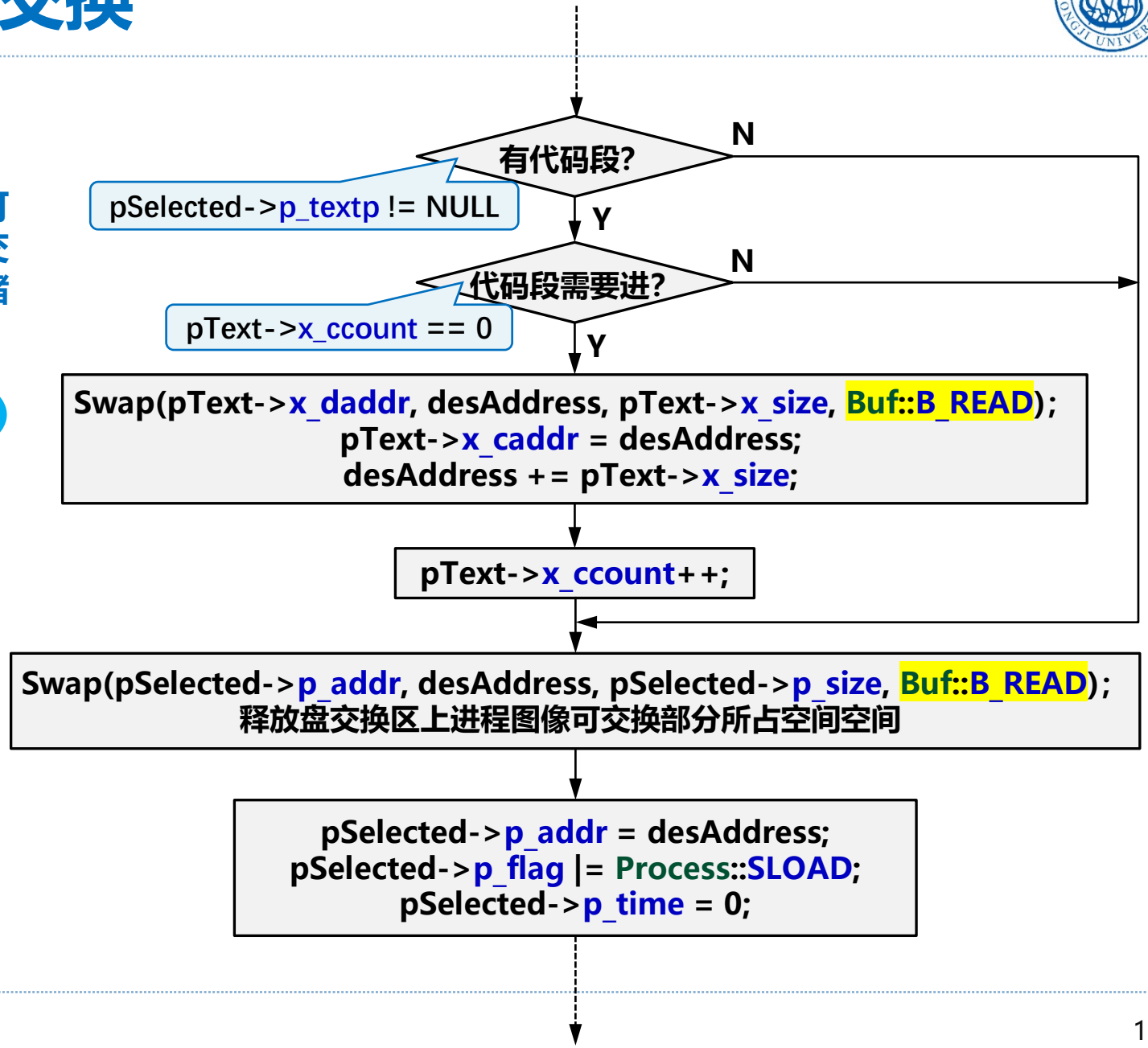
内存空间不足时的图像交换

1. 进程图像的换进



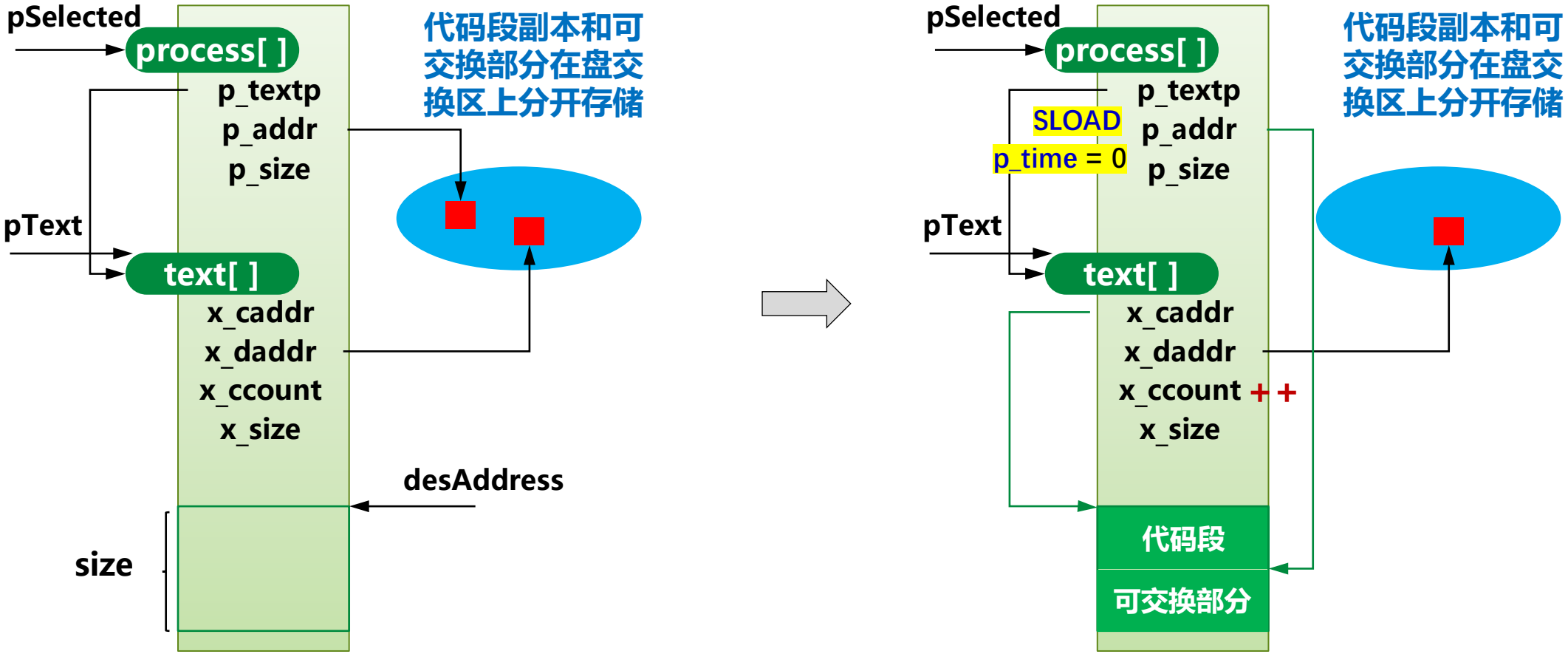
代码段副本和可交换部分在盘交换区上分开存储

成功分配到内存。如果代码段需要进内存，
size = 代码段长度 + 可交换部分长度



内存空间不足时的图像交换

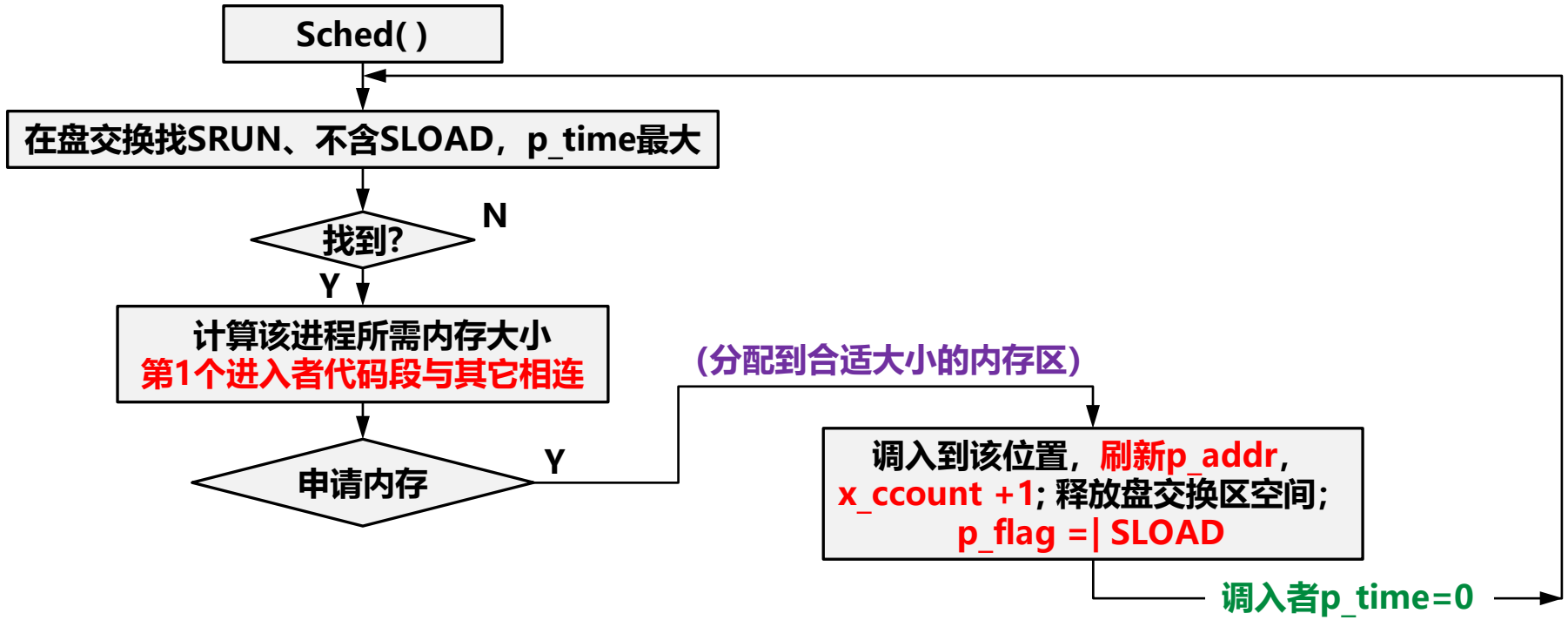
1. 进程图像的换进



成功分配到内存。如果代码段需要进内存，size = 代码段长度 + 可交换部分长度

如果代码段需要进内存，则与可交换部分连续存放

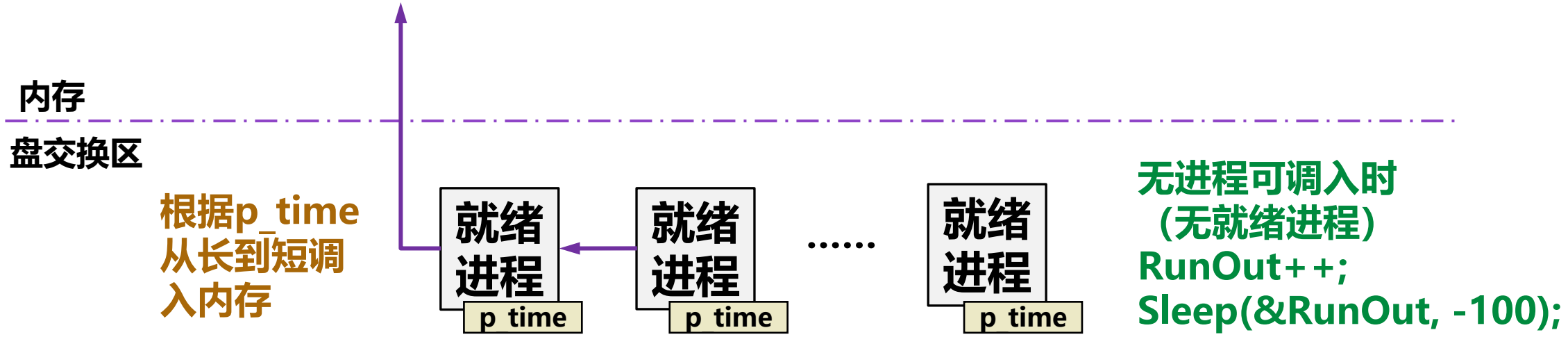
内存空间不足时的图像交换



内存空间不足时的图像交换

0# 进程执行ProcessManager::Sched

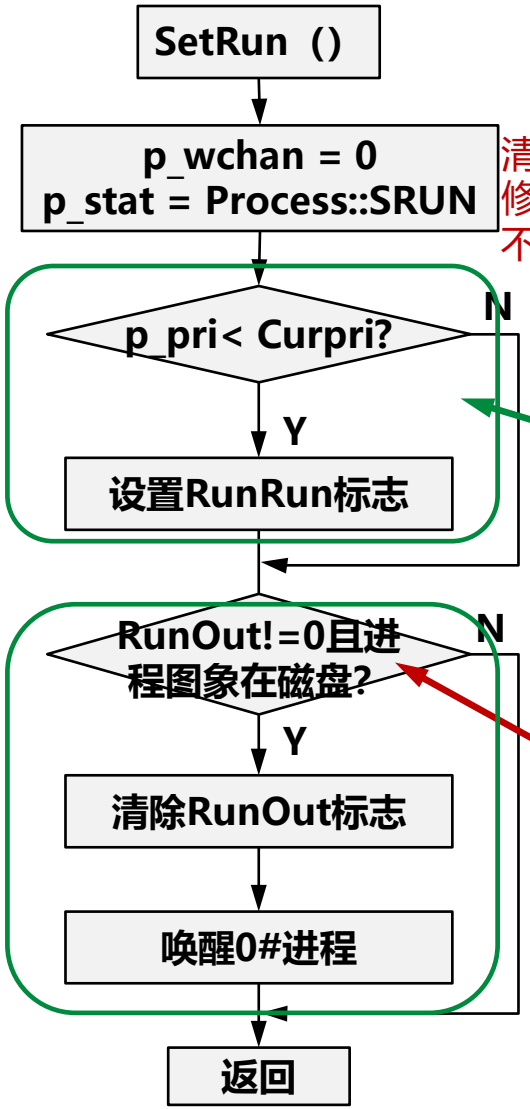
每秒时钟中断：对所有进程 p_time ++;



内存空间不足时的图像交换

睡在这里的0#进程什么时候醒过来?

内存空间不足时的图像交换



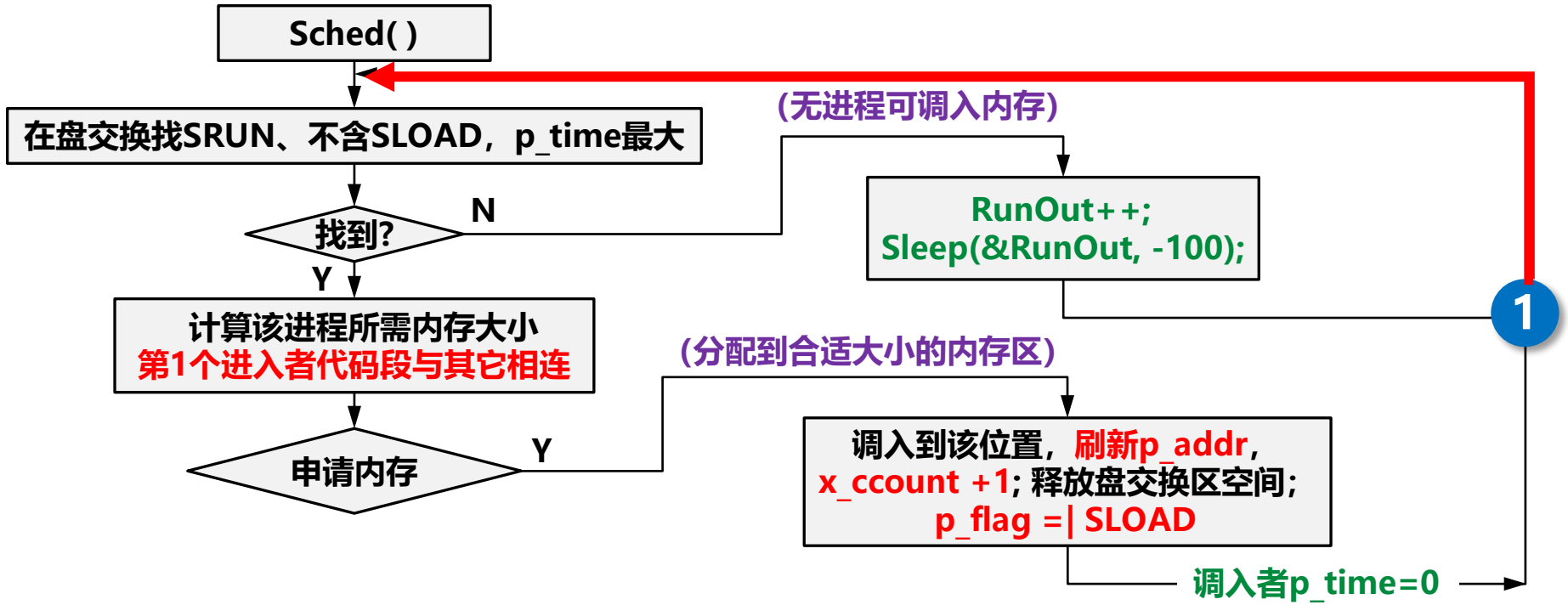
被唤醒进程的优先级高于现运行进程，设置RunRun

0#正因为盘交换区上无就绪进程需要进入内存而睡觉

此时，有进程在磁盘就绪，唤醒0#进程将其换入

0#进程具有最高的优先级，唤醒它一定会设置RunRun，下一次Swtch时，一定是0#上台。

内存空间不足时的图像交换

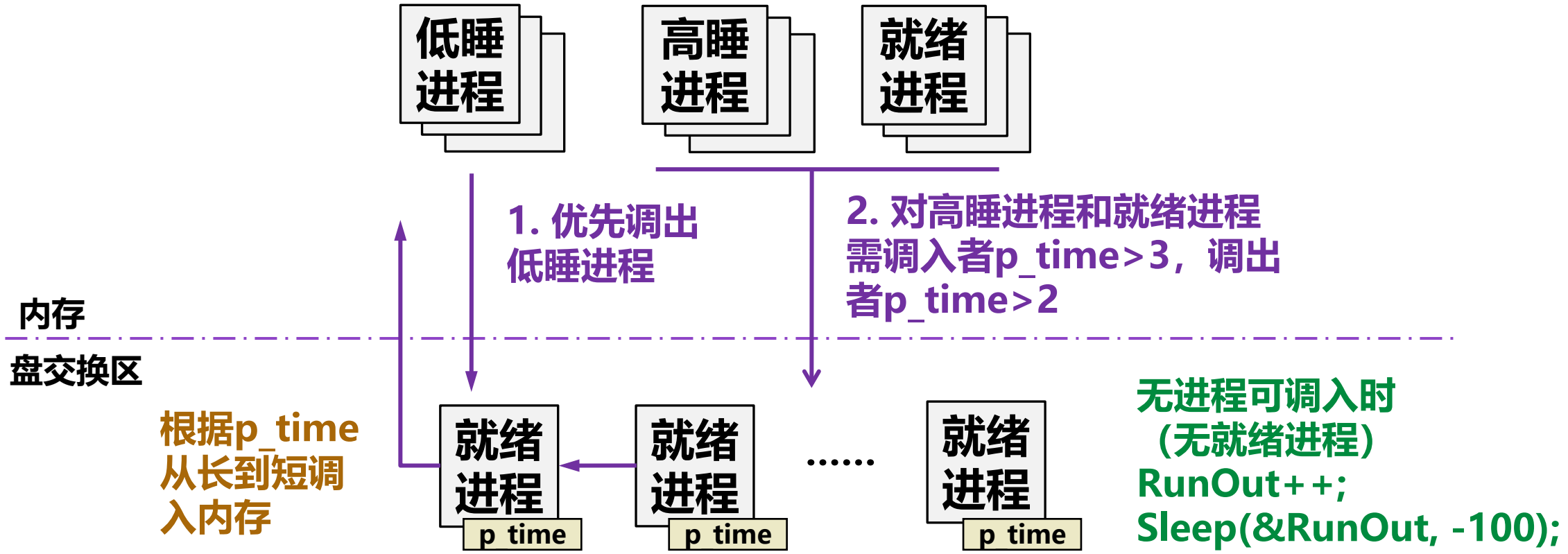


内存空间不足时的图像交换

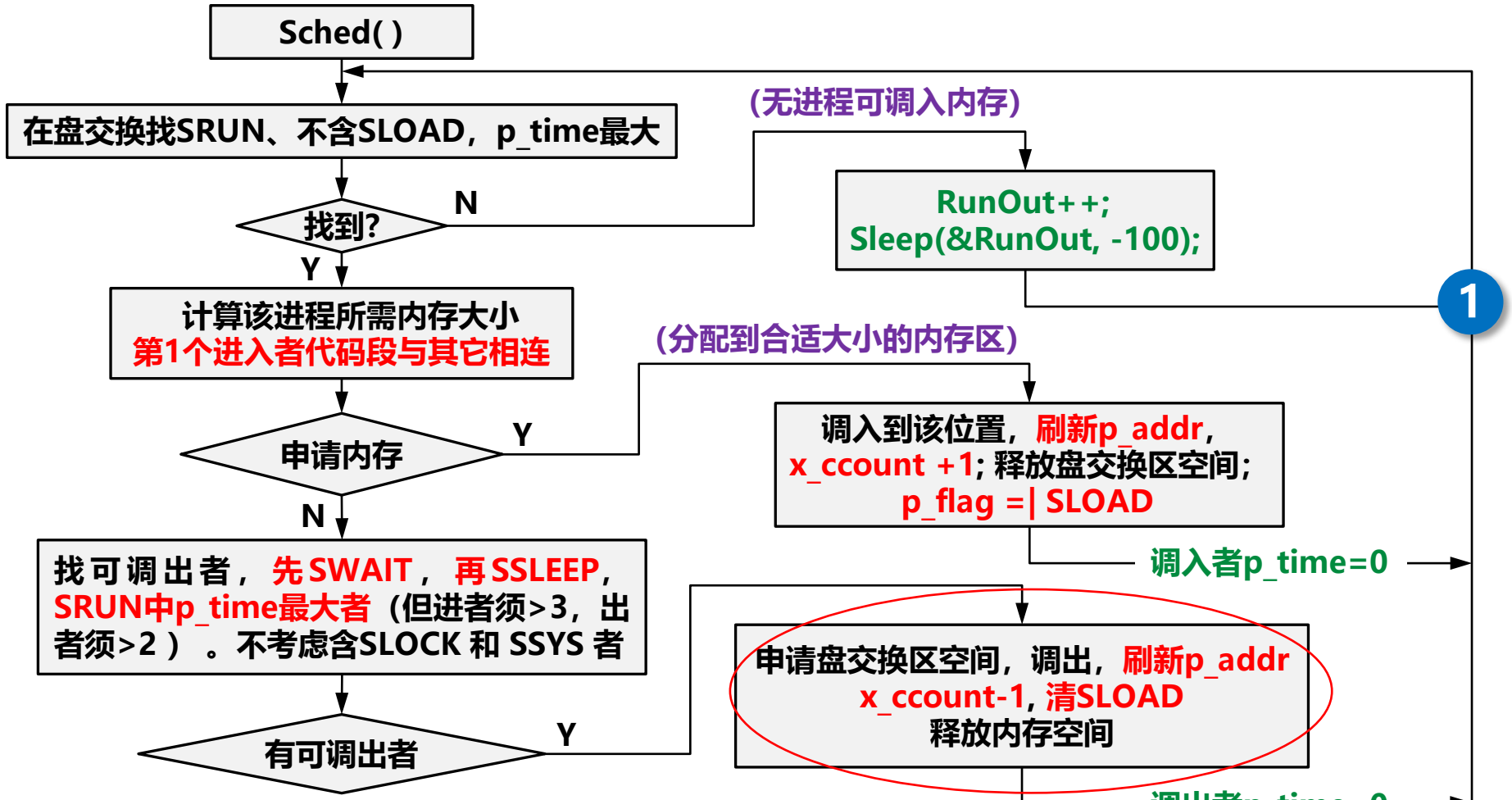
0# 进程执行ProcessManager::Sched

每秒时钟中断：对所有进程 p_time ++;

无内存空闲区时，从p_flag标志字不包含SSYS和SLOCK的进程中选择进程图象调出

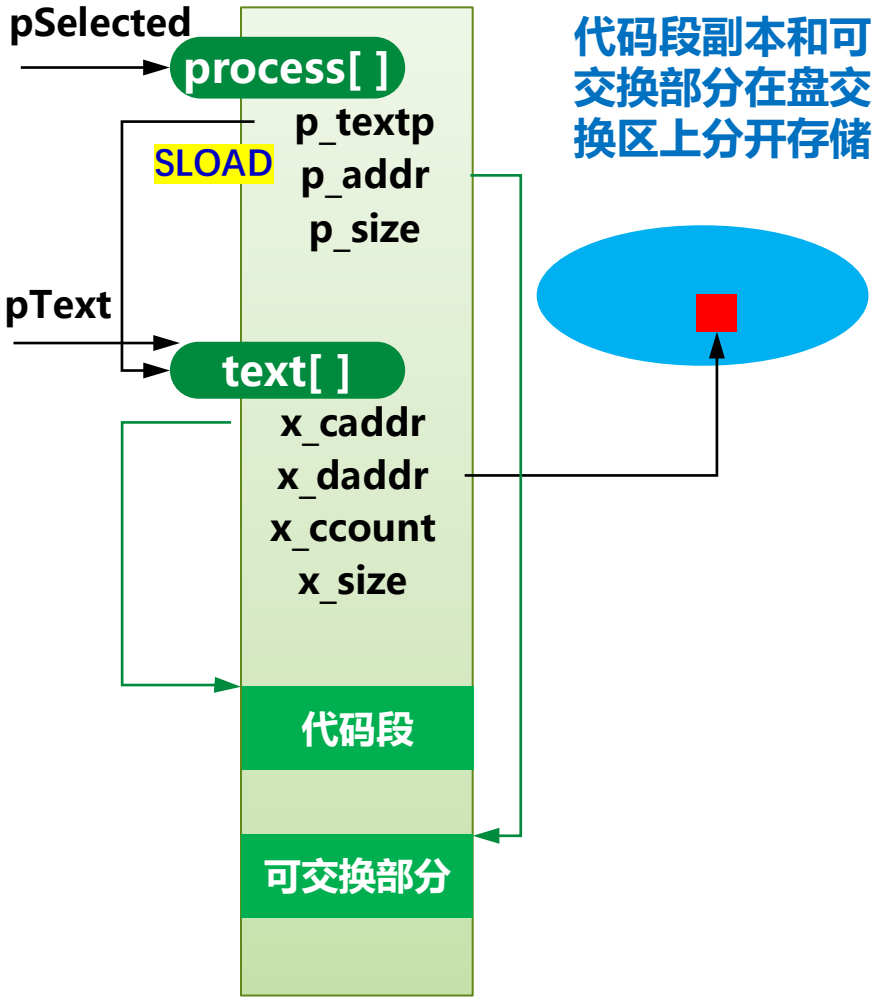


内存空间不足时的图像交换



内存空间不足时的图像交换

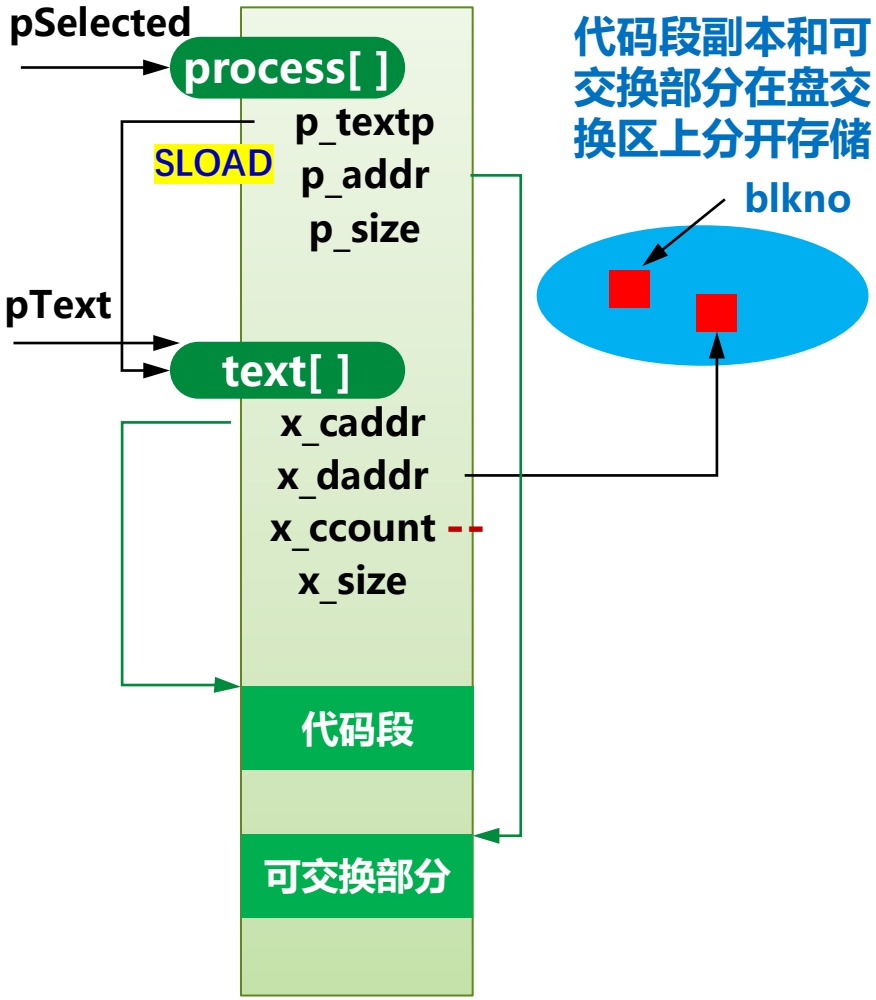
2. 进程图像的换出

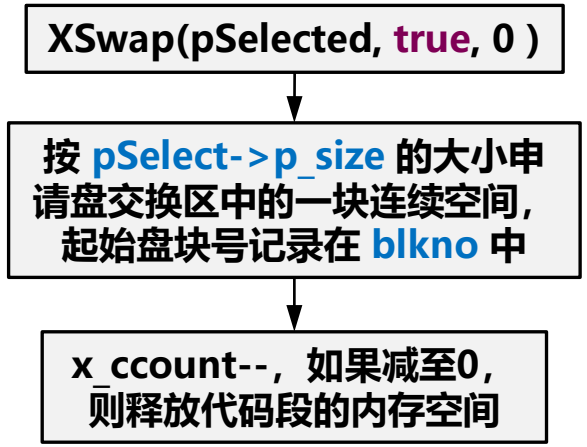



```
void ProcessManager::XSwap(
/* pProcess: 指向要换出的进程的proc */
Process* pProcess,
/* bFreeMemory: 是否释放进程图像占据的内存
bool bFreeMemory,
/* size: 可交换部分图像长度；若为0时，直接用p_size
int size
)
```

内存空间不足时的图像交换

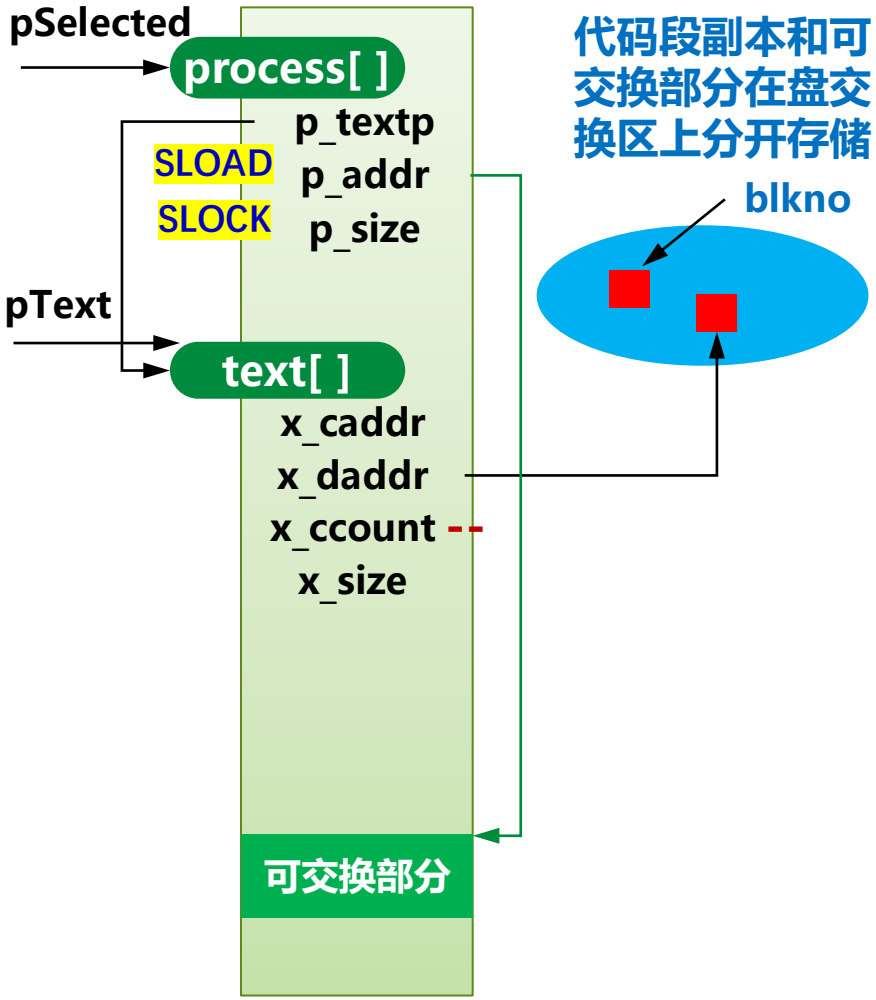
2. 进程图像的换出





内存空间不足时的图像交换

2. 进程图像的换出



代码段副本和可交换部分在盘交换区上分开存储

XSwap(pSelected, true, 0)

按 pSelect->p_size 的大小申请盘交换区中的一块连续空间，起始盘块号记录在 blkno 中

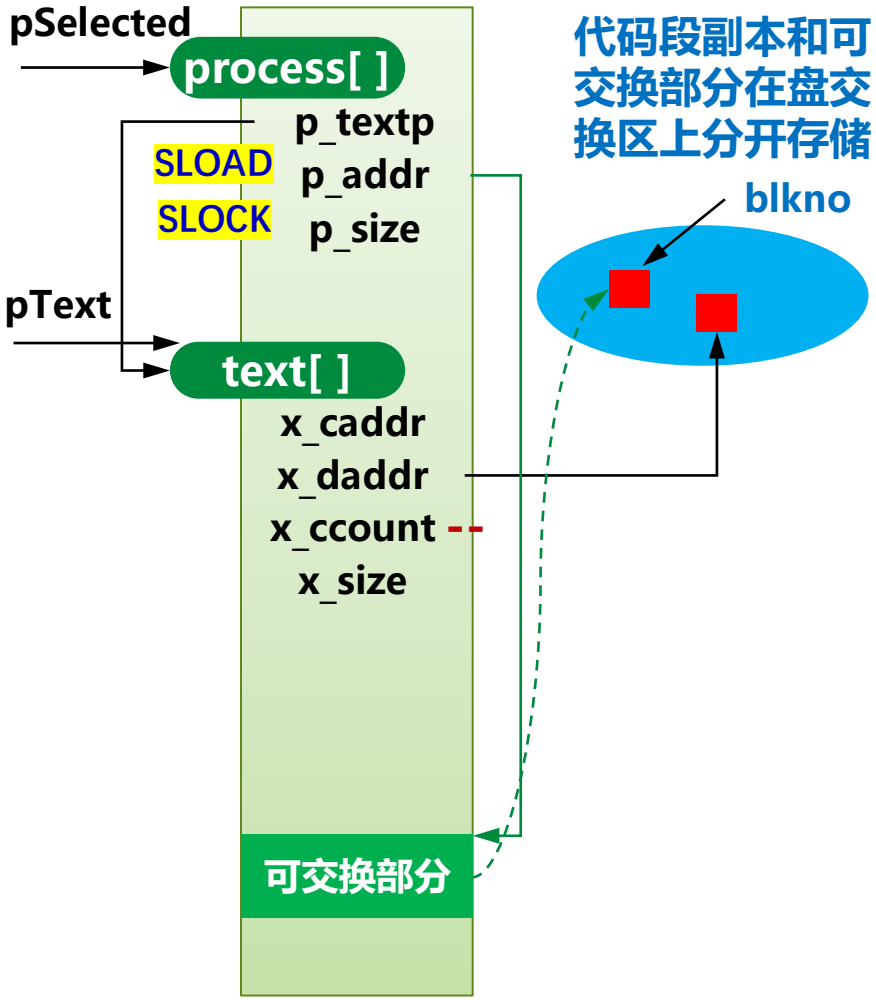
x_ccount--，如果减至0，则释放代码段的内存空间

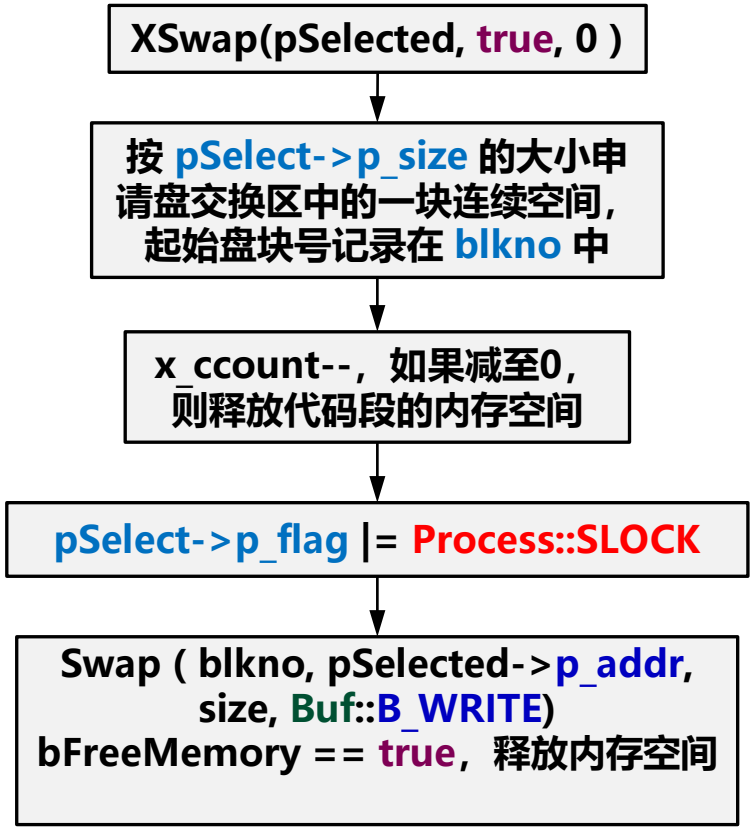
pSelect->p_flag |= Process::SLOCK

上锁，防止同一进程图像被重复换出

内存空间不足时的图像交换

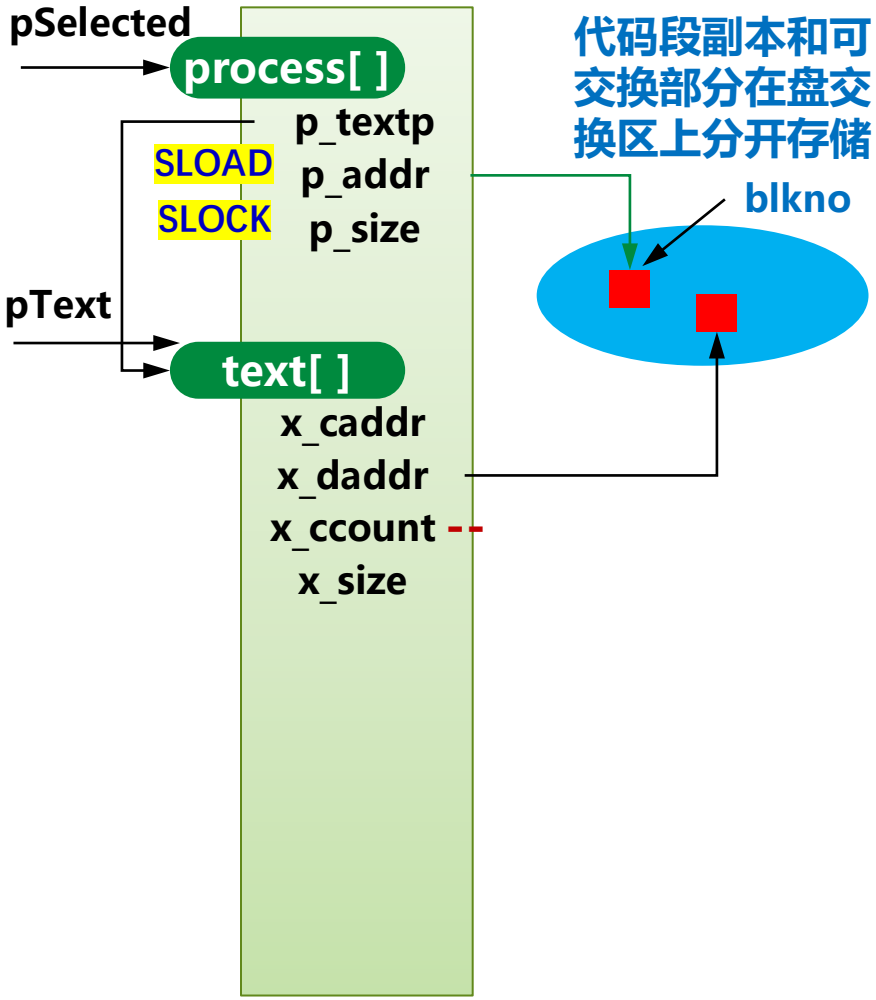
2. 进程图像的换出

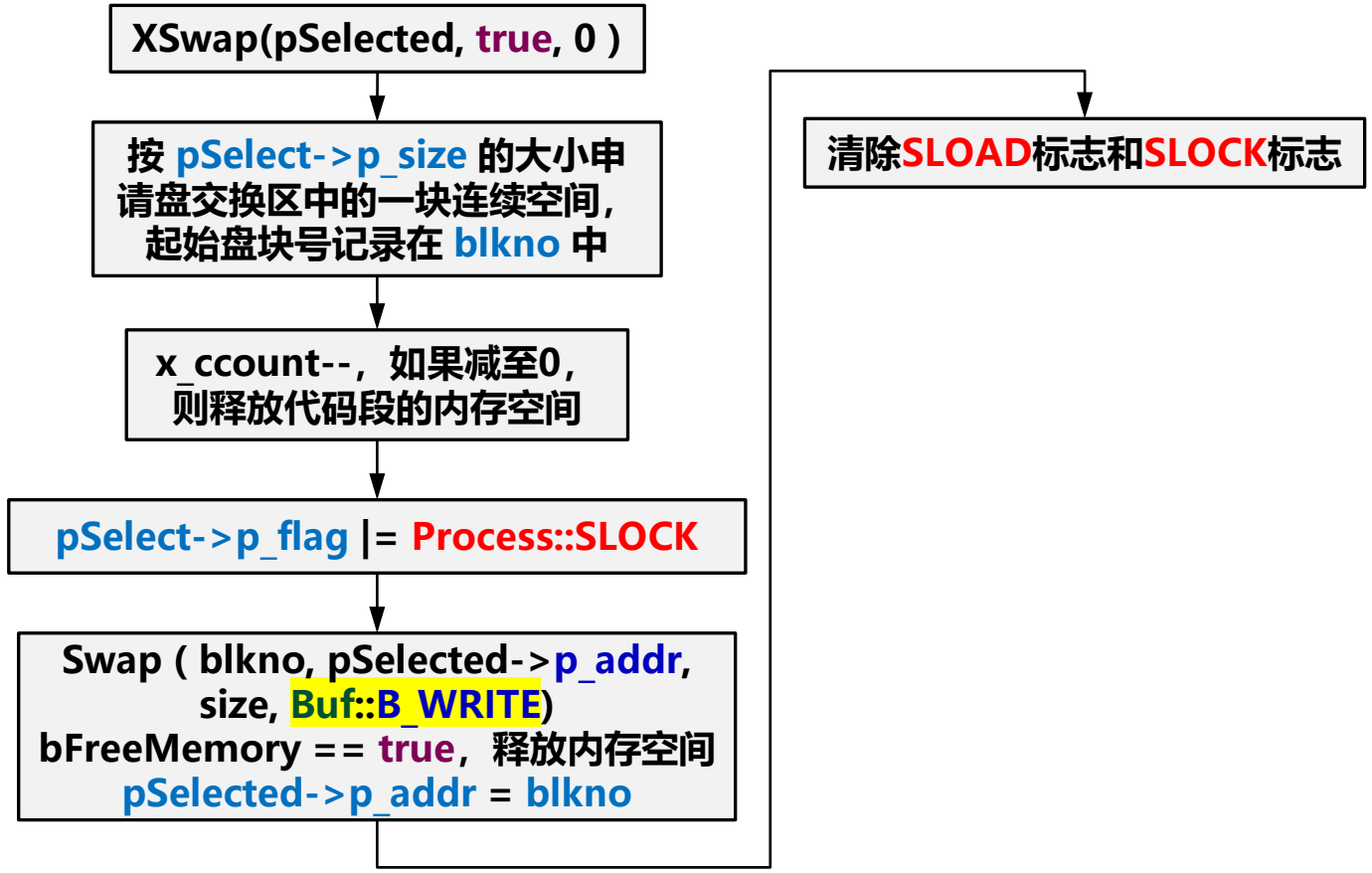




内存空间不足时的图像交换

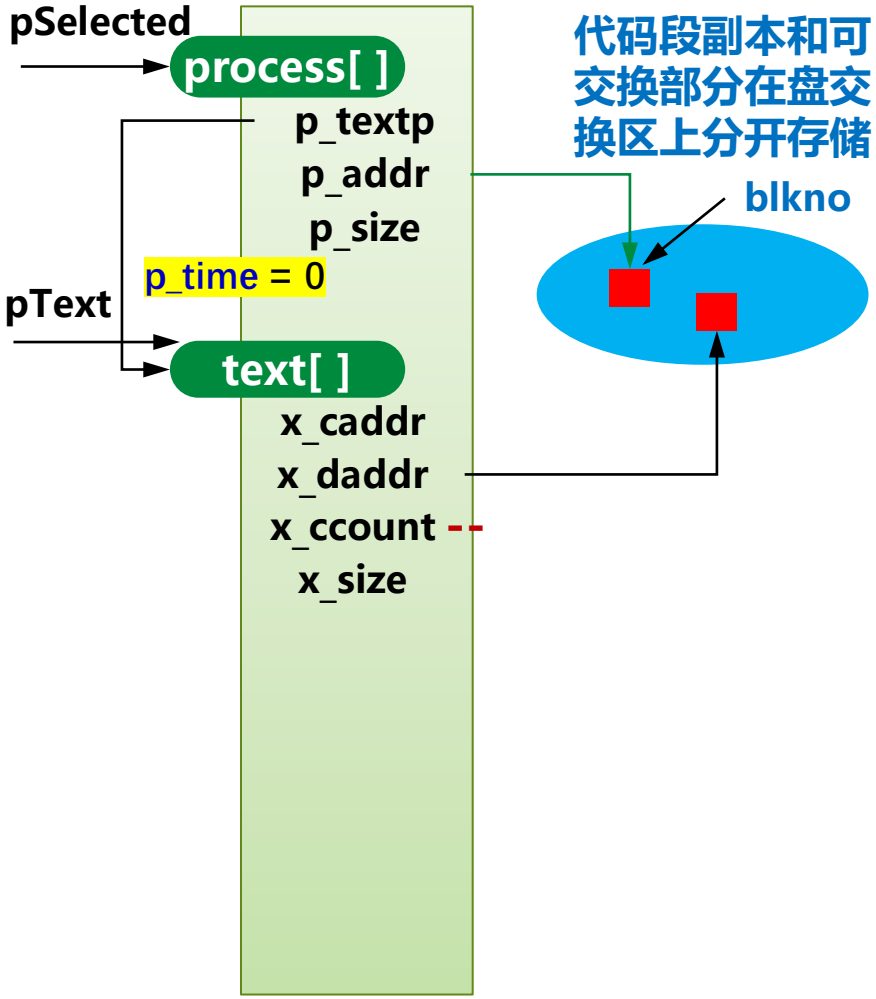
2. 进程图像的换出

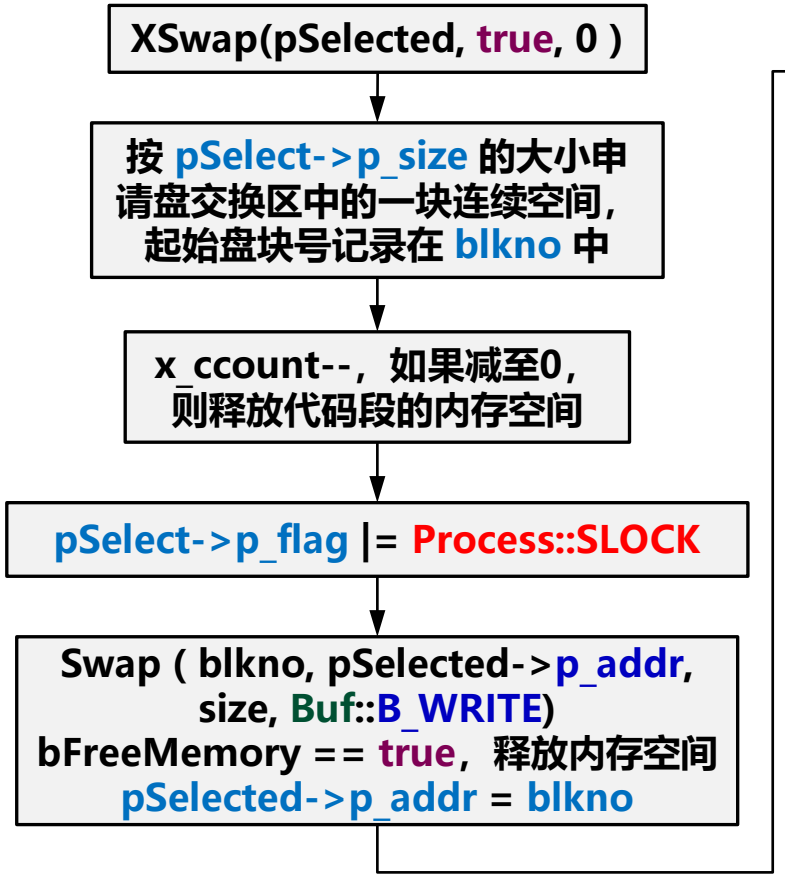


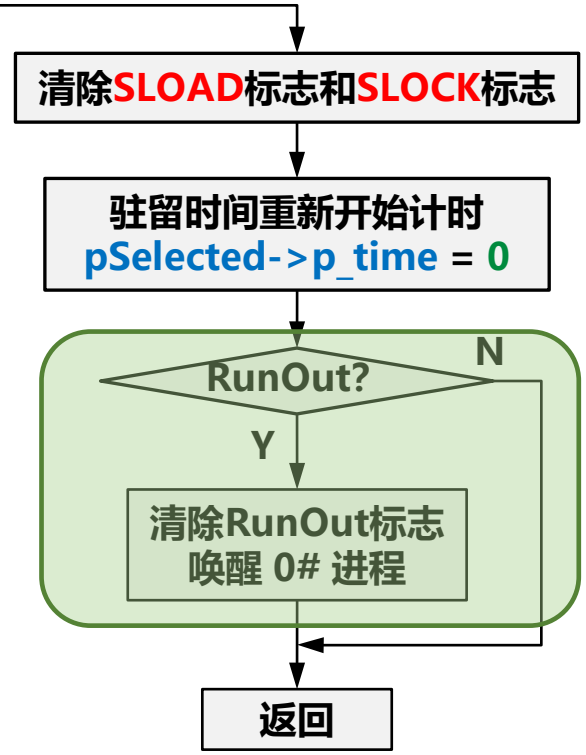


内存空间不足时的图像交换

2. 进程图像的换出

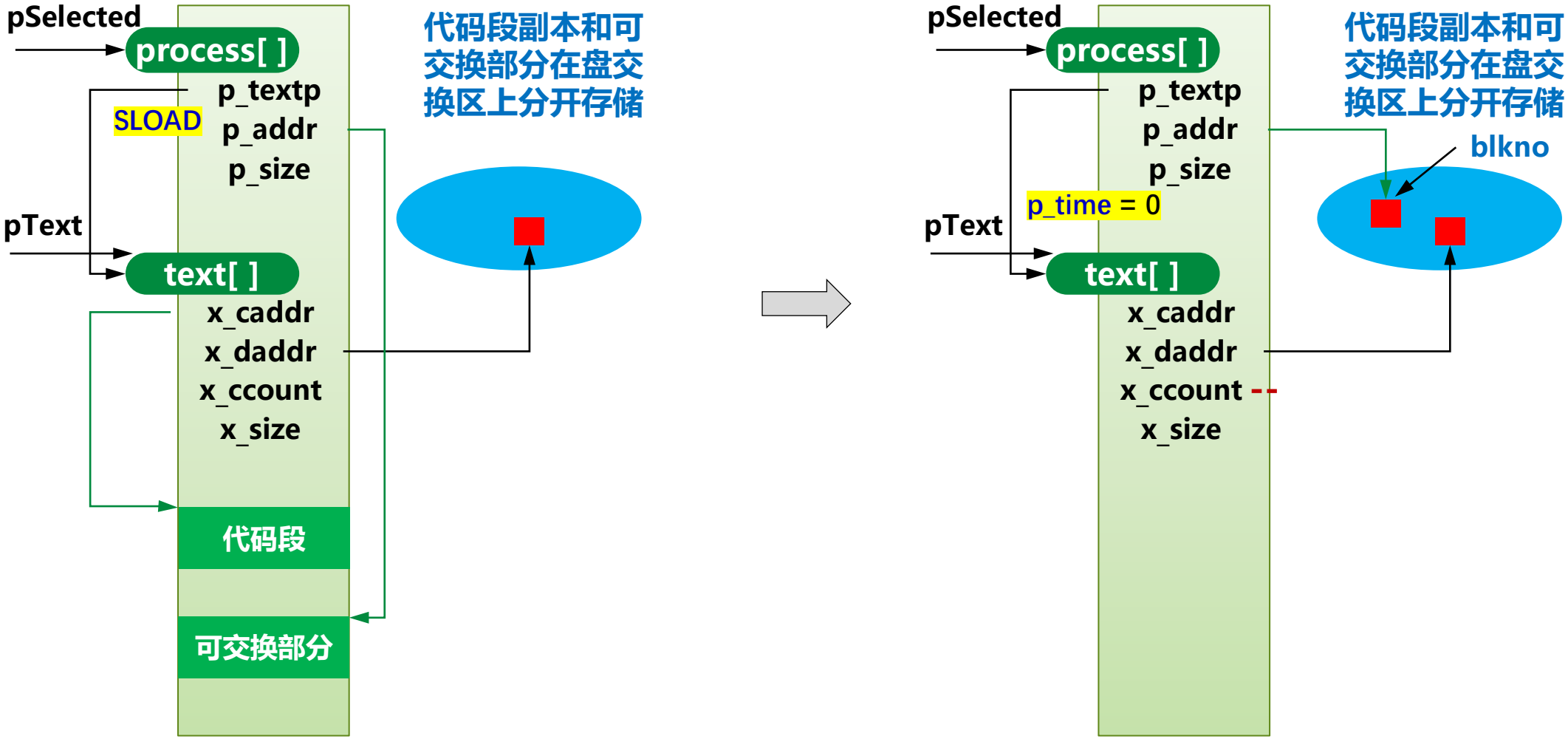




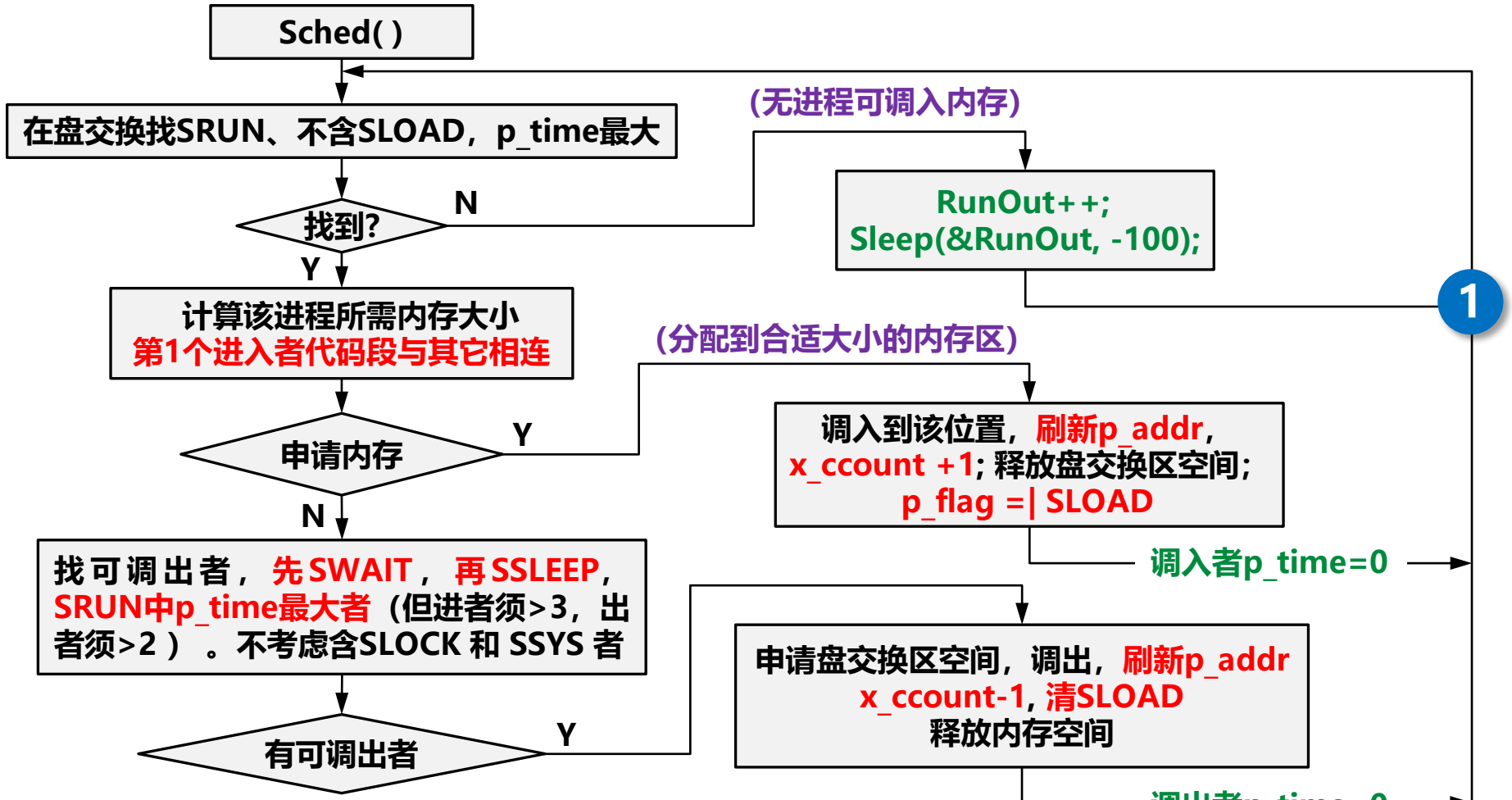


内存空间不足时的图像交换

2. 进程图像的换出



内存空间不足时的图像交换

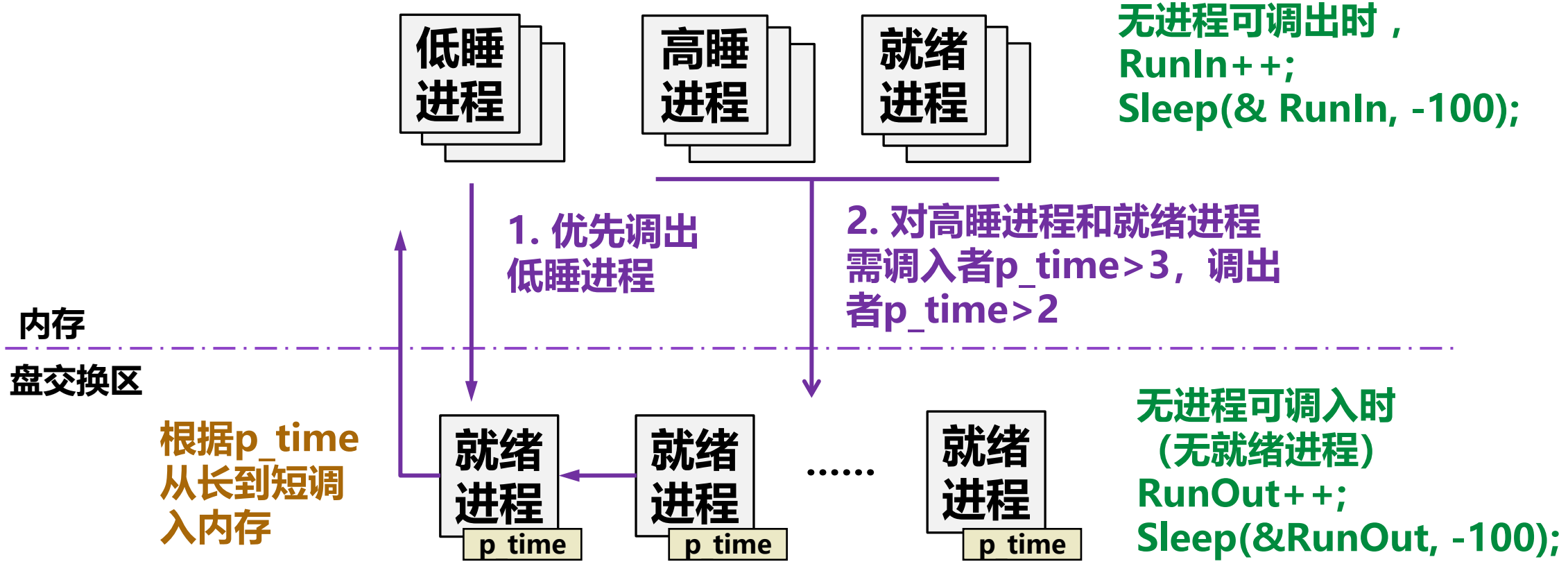


内存空间不足时的图像交换

0# 进程执行ProcessManager::Sched

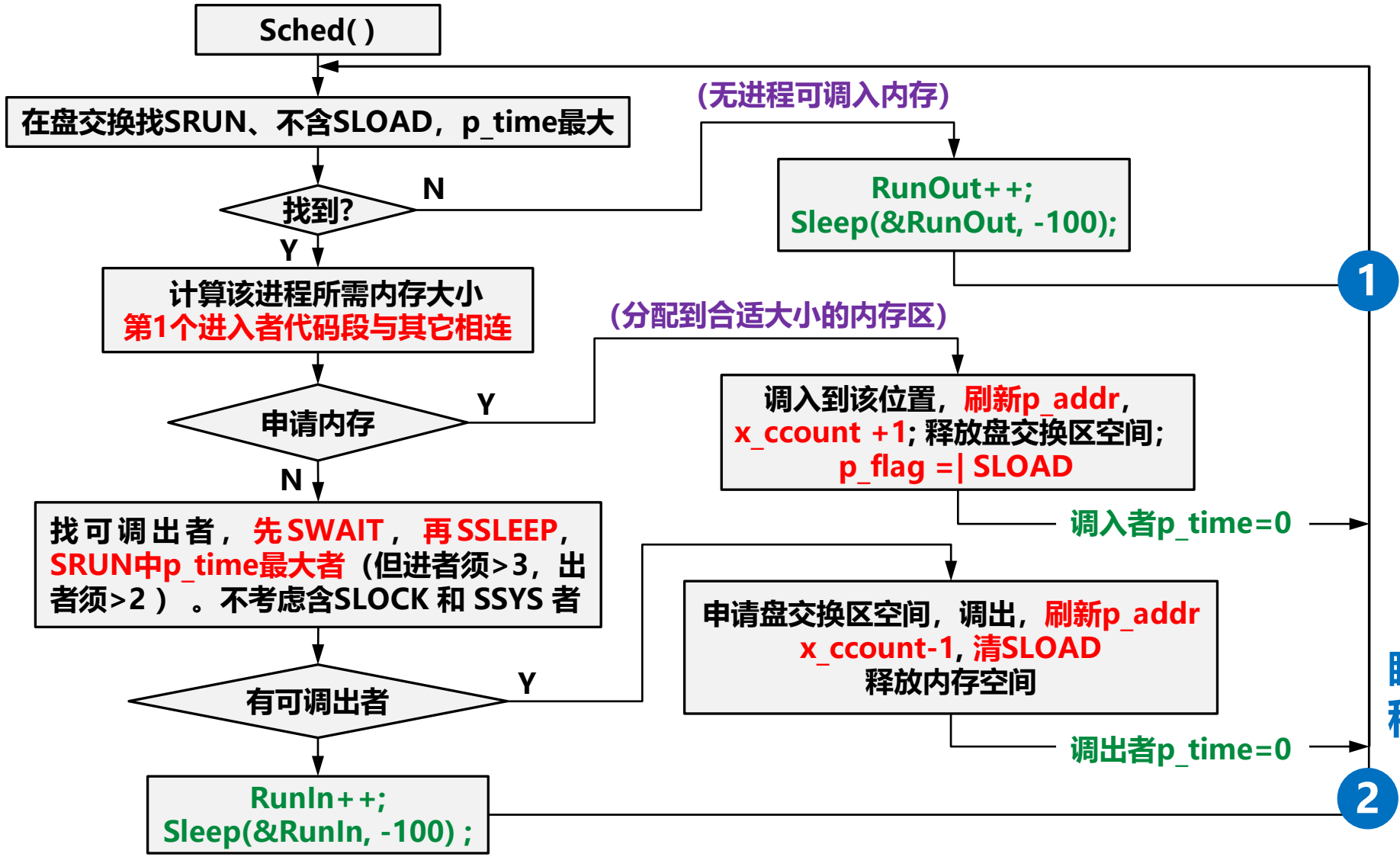
无内存空闲区时，从p_flag标志字不包含SSYS和SLOCK的进程中选择进程图象调出

每秒时钟中断：对所有进程 p_time ++;



内存空间不足时的图像交换

进程图像的换进换出



睡在这里的0#进程什么时候醒过来?

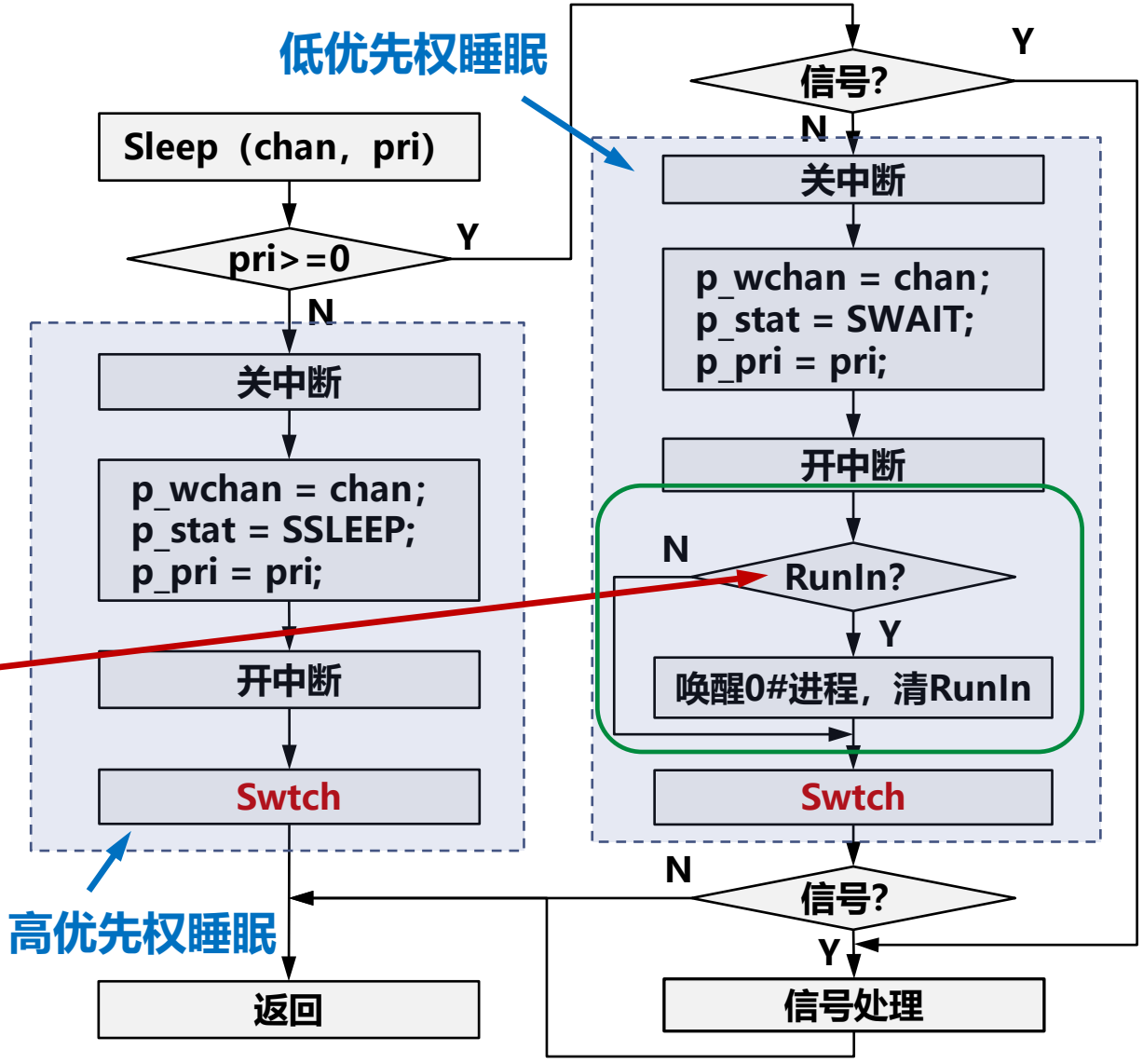
内存空间不足时的图像交换

找可调出者，先SWAIT，再SSLEEP，SRUN中p_time最大者（但进者须>3，出者须>2）。不考虑含SLOCK 和 SSYS 者

1. 没有低睡进程

0#正因为找不到可以换出的进程而睡觉

此时，有进程进入低睡状态，则唤醒0#进程将其换出

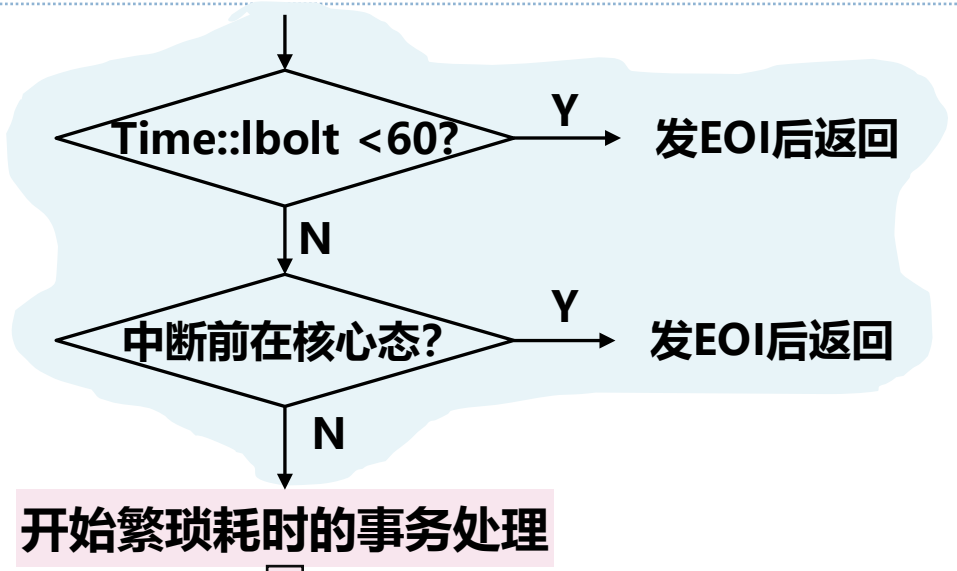


内存空间不足时的图像交换

找可调出者，先SWAIT，再SSLEEP，SRUN中p_time最大者（但进者须>3，出者须>2）。不考虑含SLOCK 和 SSYS 者

1. 没有低睡进程

2. 换进者和换出者的时间不够



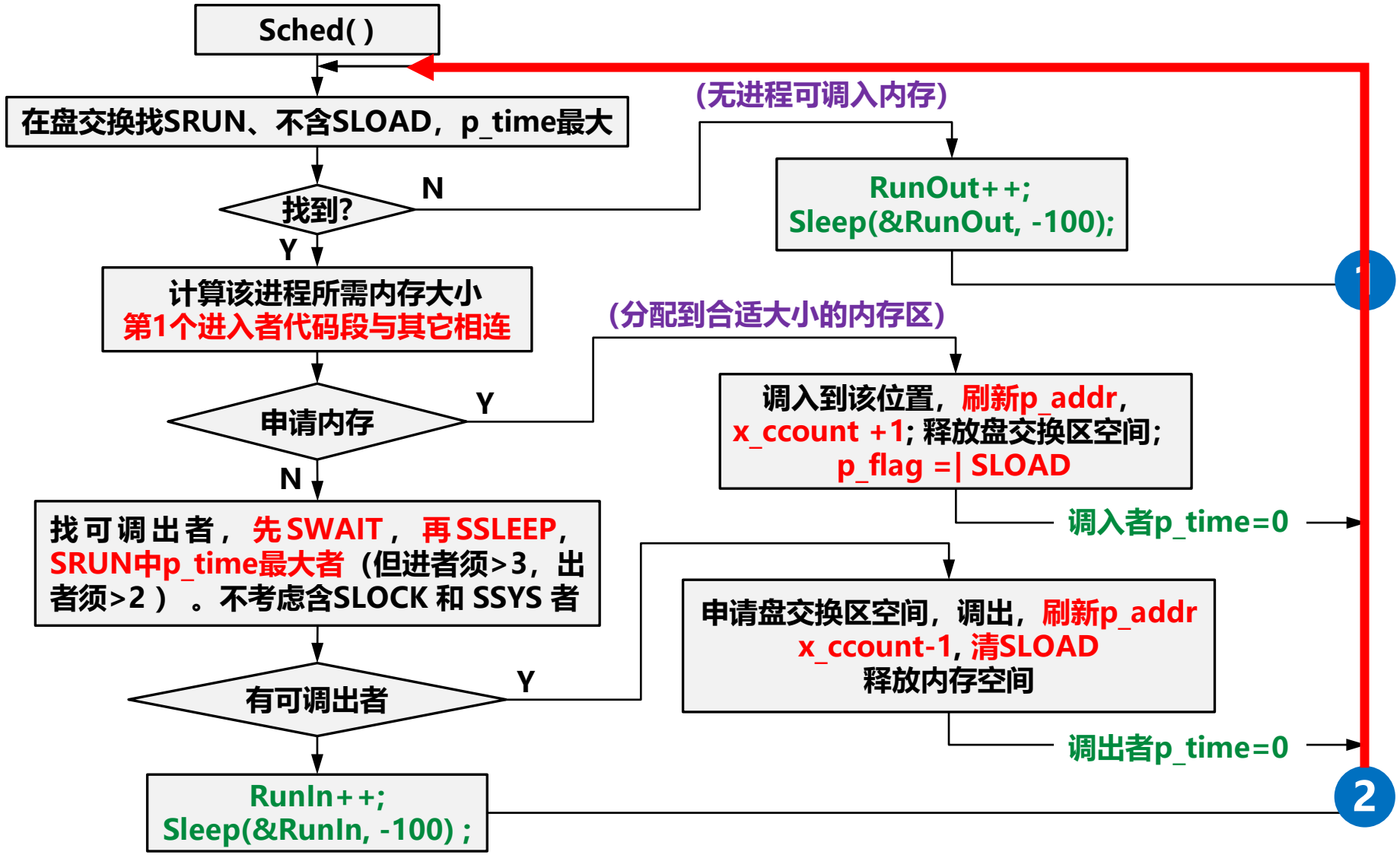
1. 维护系统时间：Time::lbolt -60; 系统时间+1秒;
2. 开中断，向中断控制器发送EOI指令;
3. 唤醒所有延时睡眠的进程;
4. 对所有进程p_time++;
5. 所有进程p_cpu=max(0, p_cpu-SCHMAG;
6. 重算部分进程的优先数;
7. 如果RunIn被设置，唤醒0#进程;
8. 现运行进程进行信号处理;
9. 重算当前进程的优先数。

0#正因为找不到可以换出的进程而睡觉

此时，系统时间又过去1秒，所有进程的p_time++,则唤醒0#进程查看是否可以换出

内存空间不足时的图像交换

进程图像的换进换出

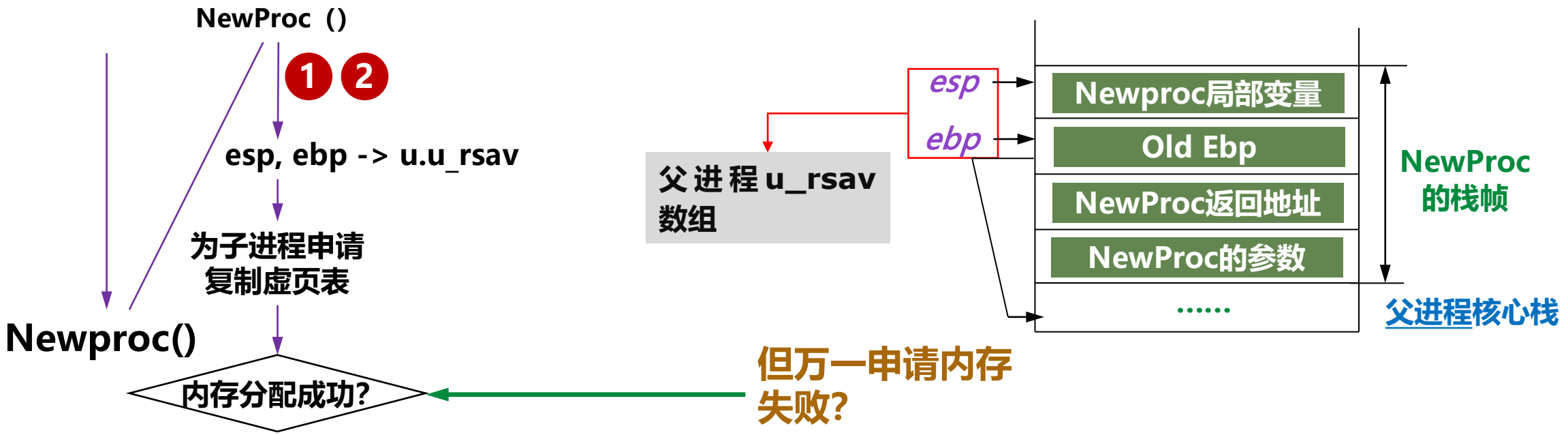


进程图像交换

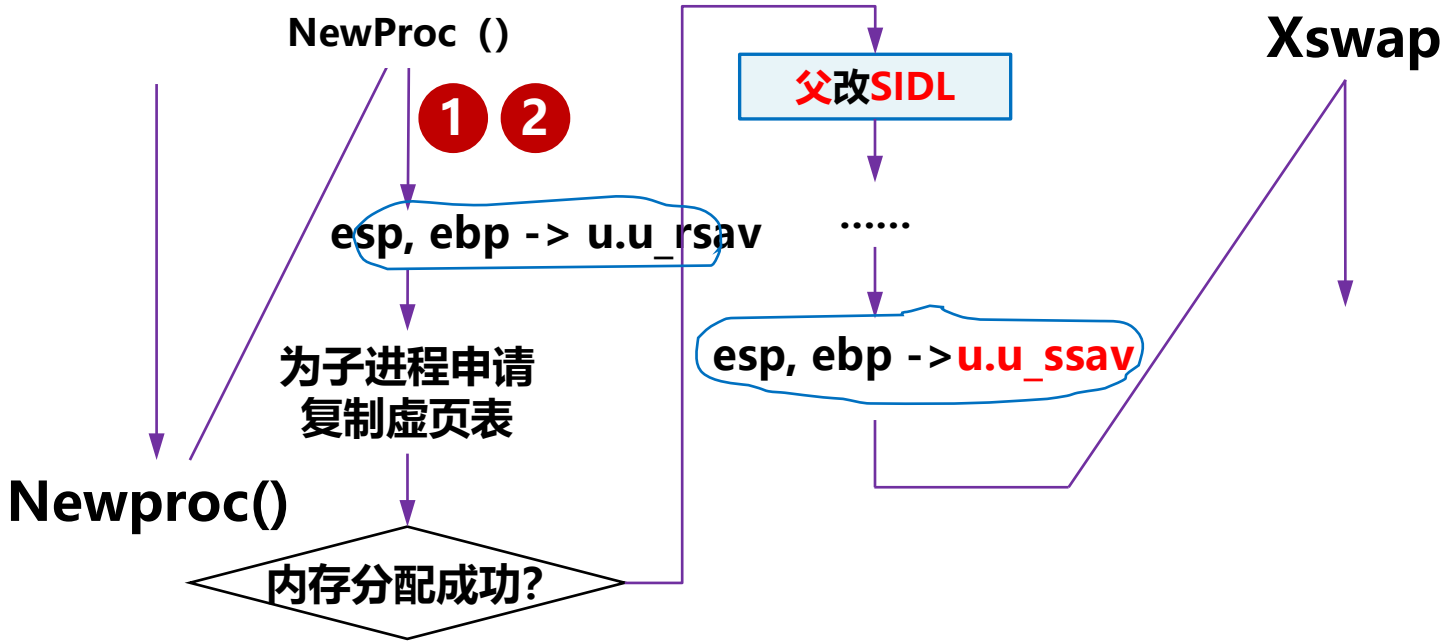
1　当内存空间不足以装下所有就绪进程时……

2　当内存空间不足以装下新建进程时……

内存空间不足时的新进程创建

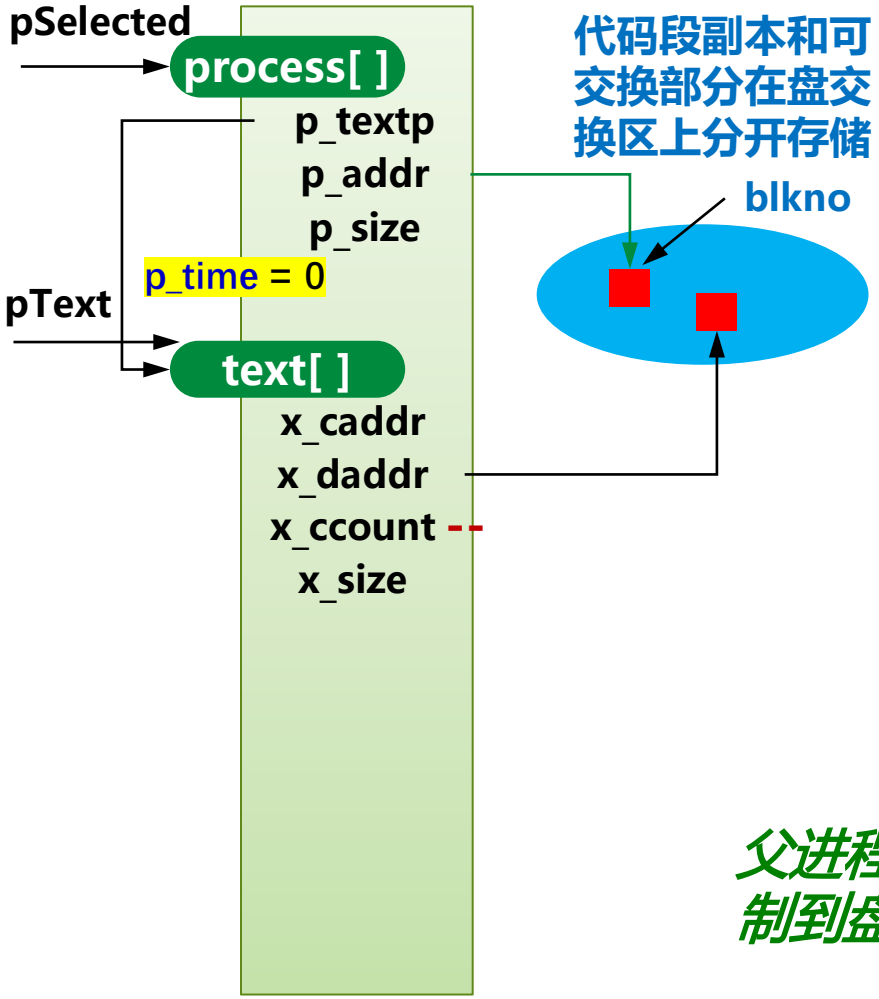


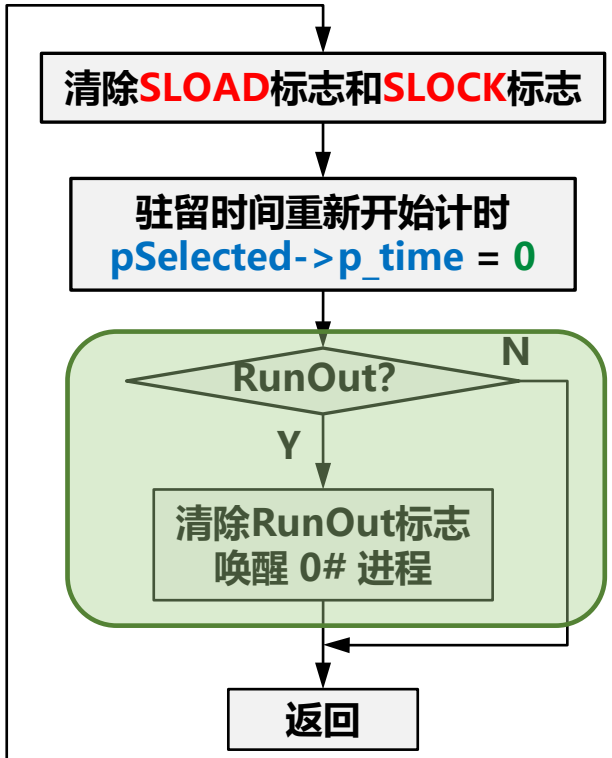
内存空间不足时的新进程创建



内存空间不足时的新进程创建

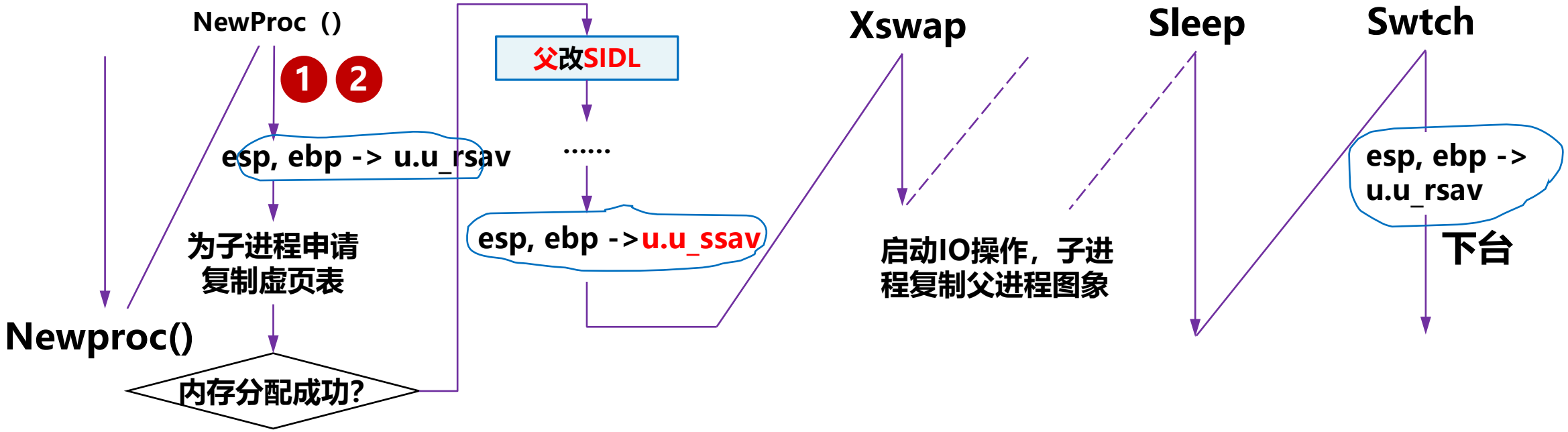
进程图像的换出



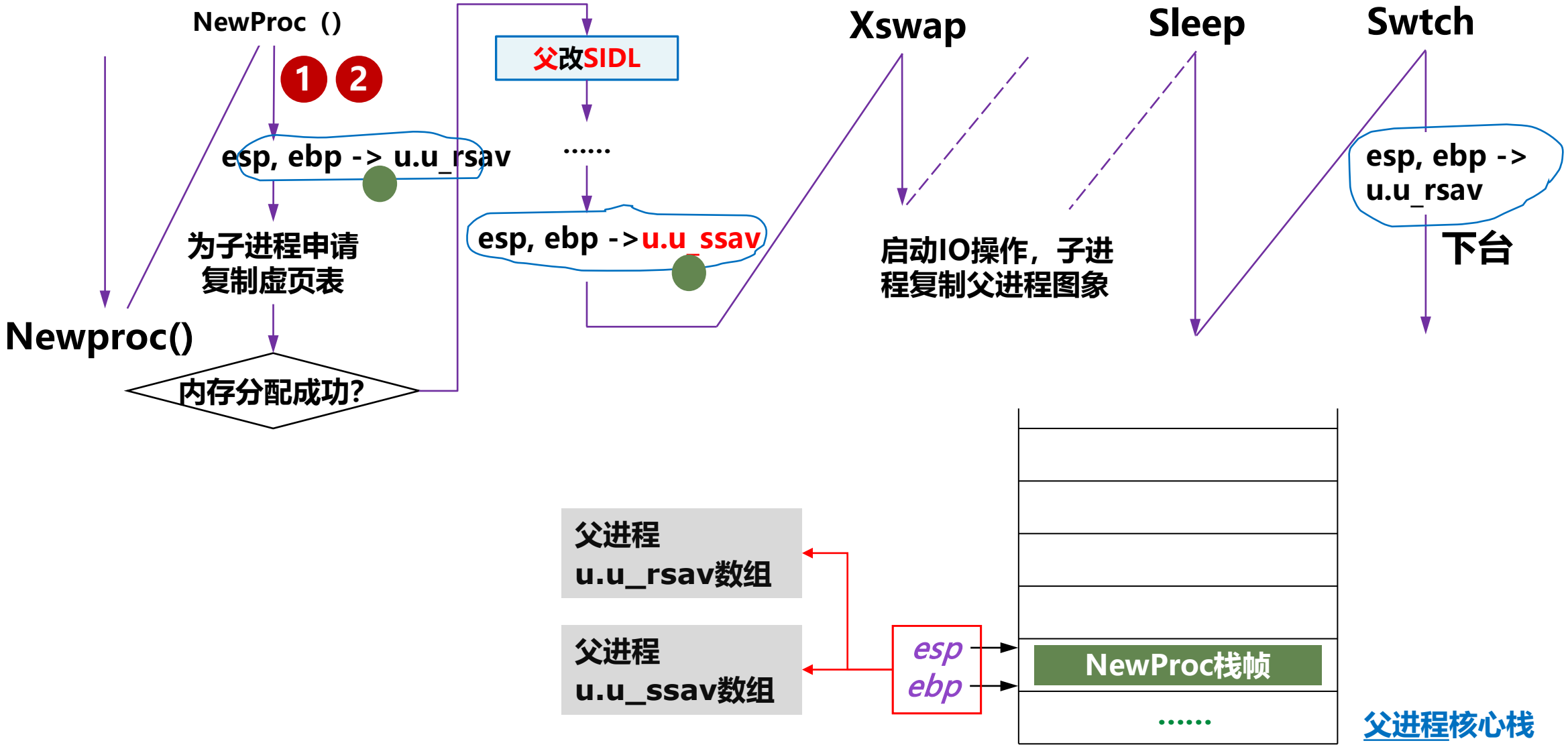


父进程执行Xswap将自己的进程图像复制到盘交换区，保留自己在内存的图像

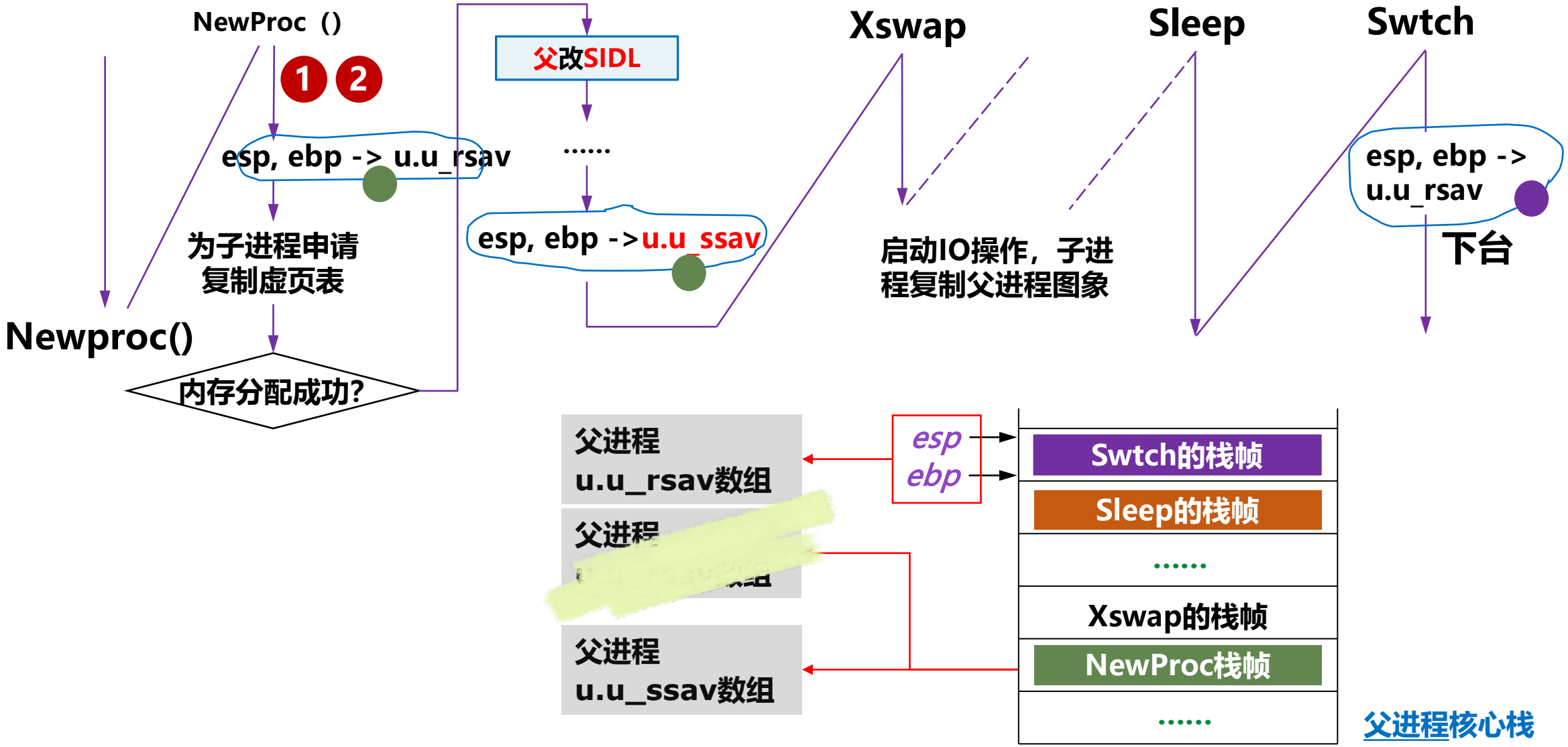
内存空间不足时的新进程创建



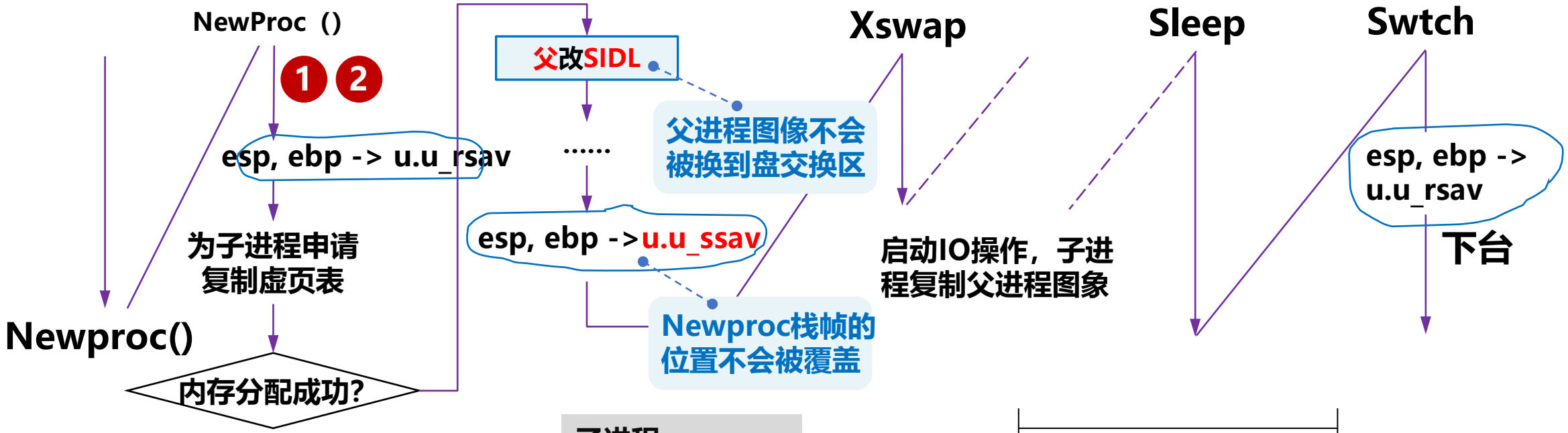
内存空间不足时的新进程创建



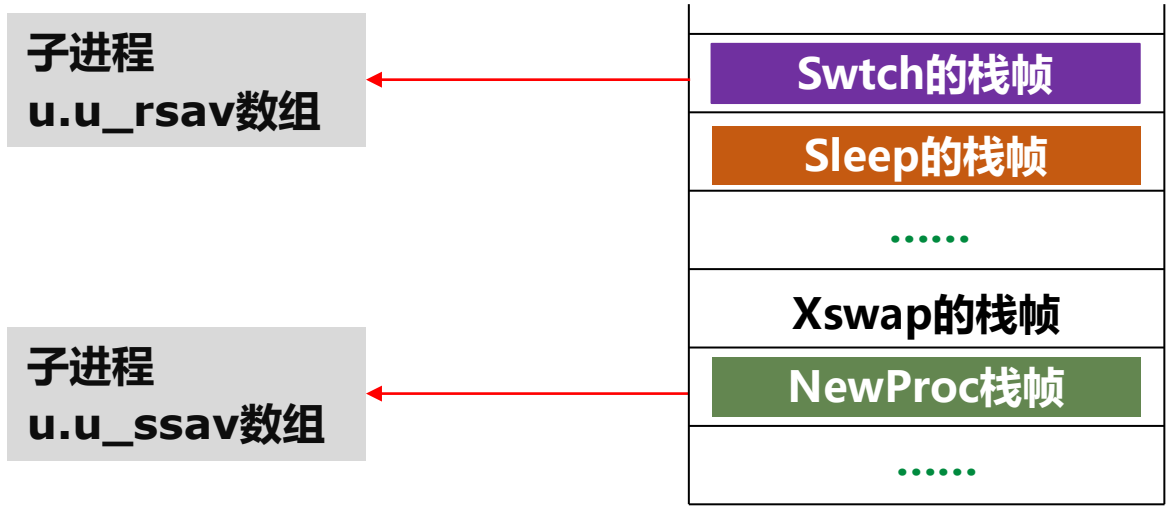
内存空间不足时的新进程创建



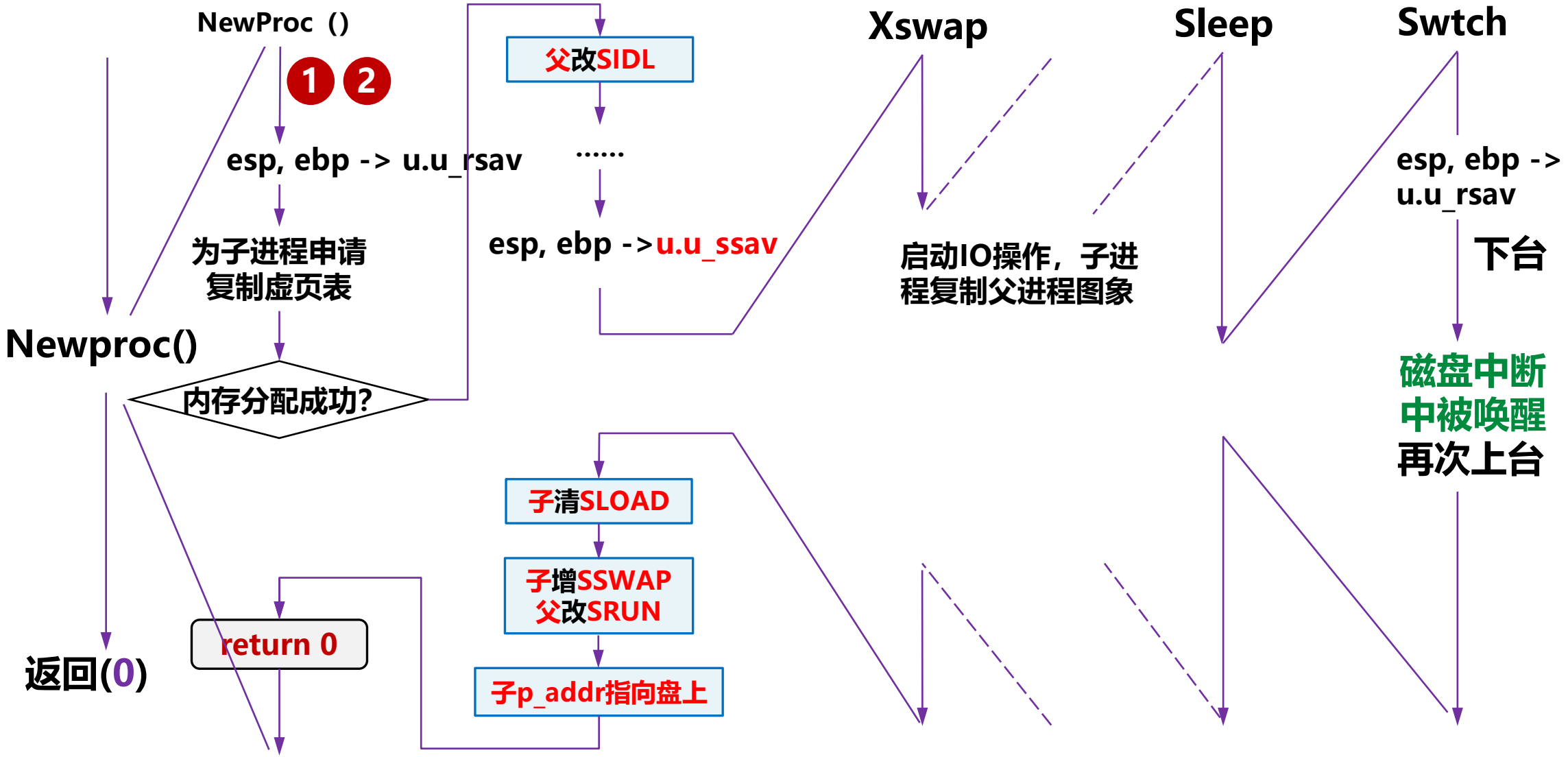
内存空间不足时的新进程创建



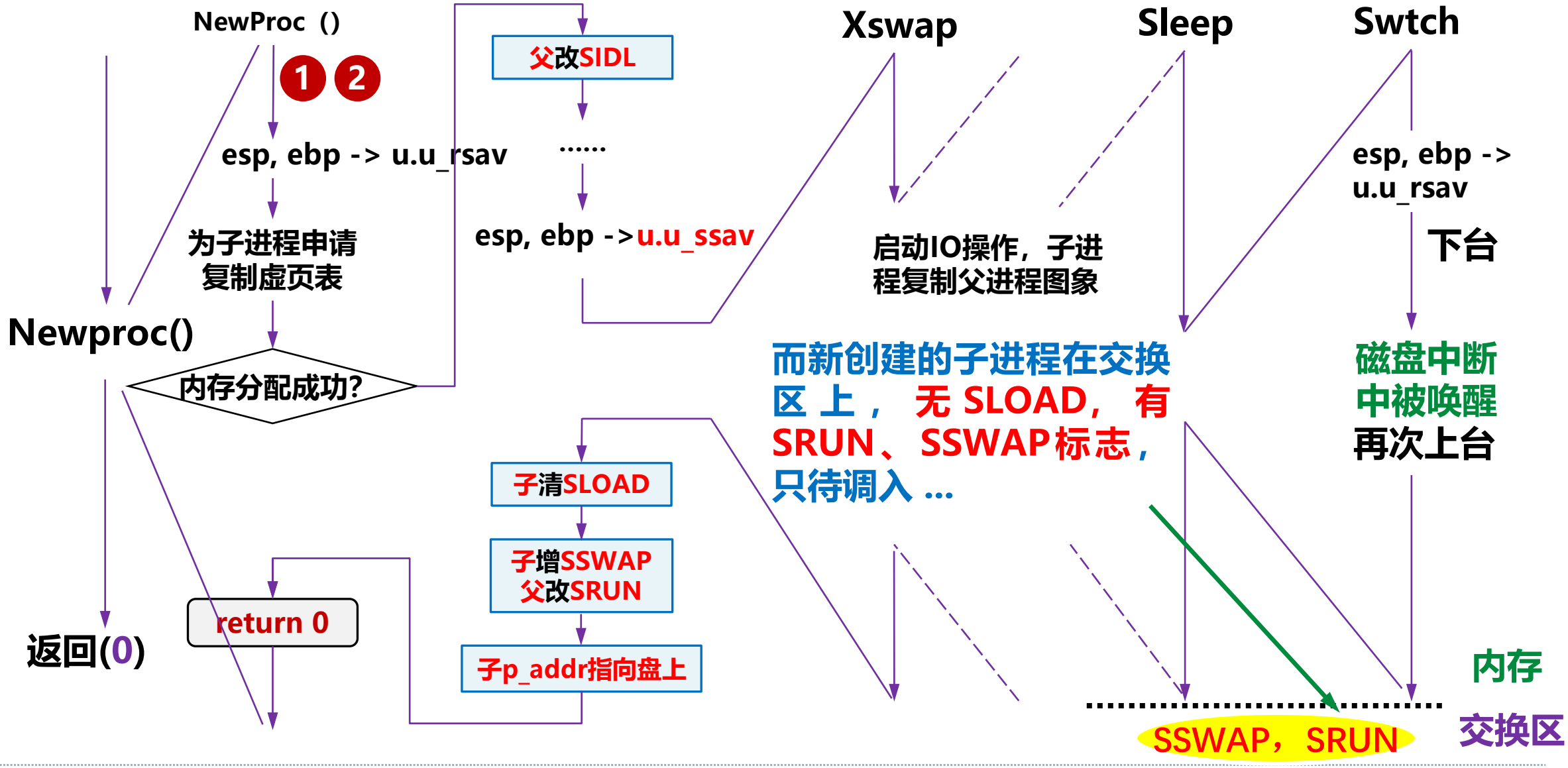
这样的核心栈和User结构复制了给子进程



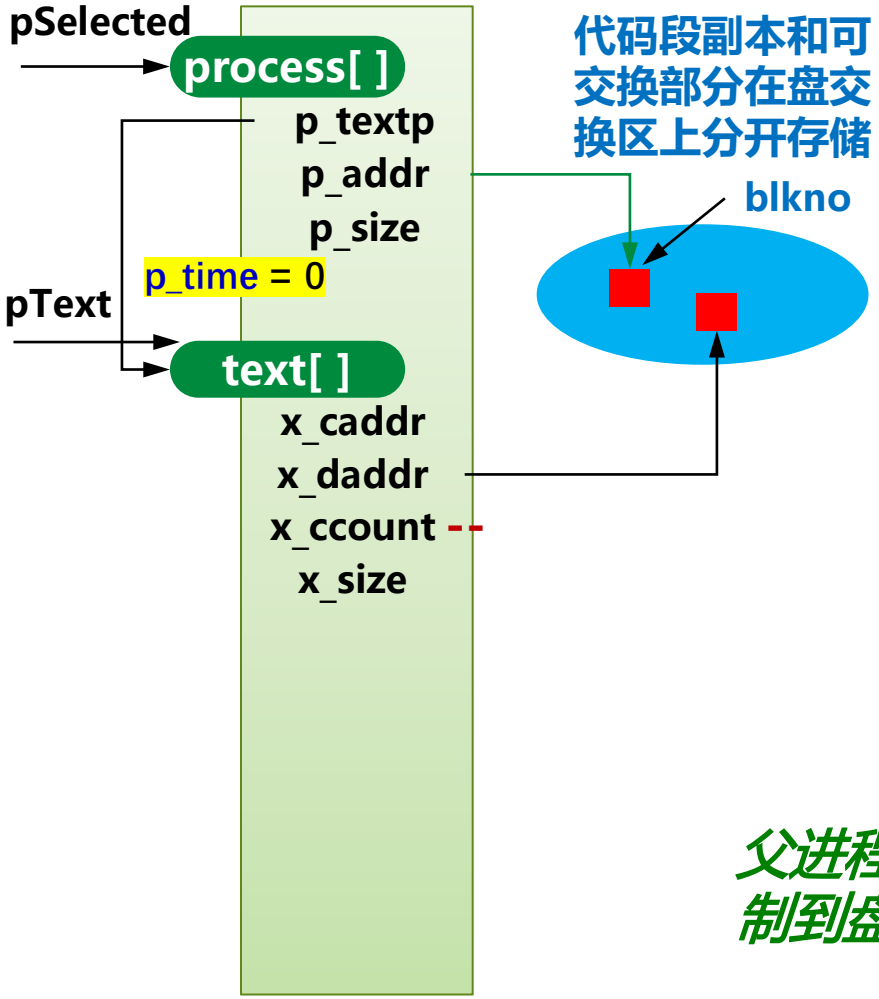
内存空间不足时的新进程创建

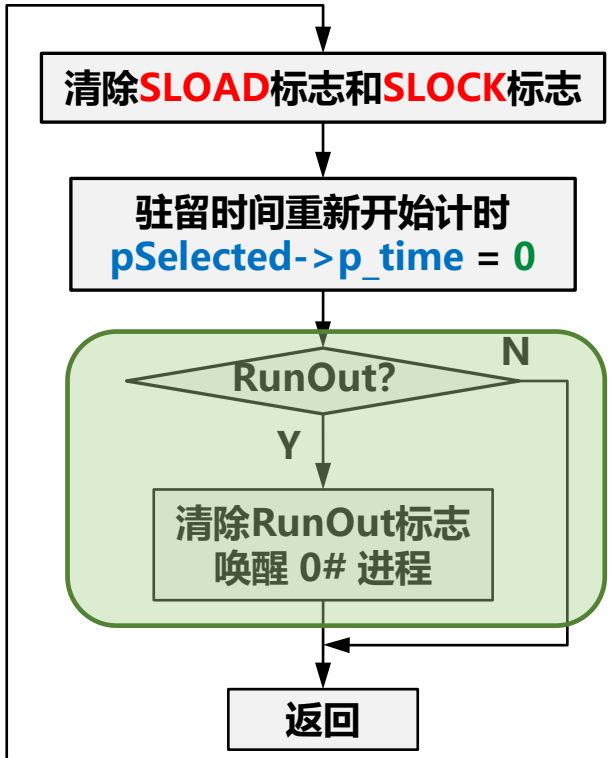


内存空间不足时的新进程创建



内存空间不足时的新进程创建

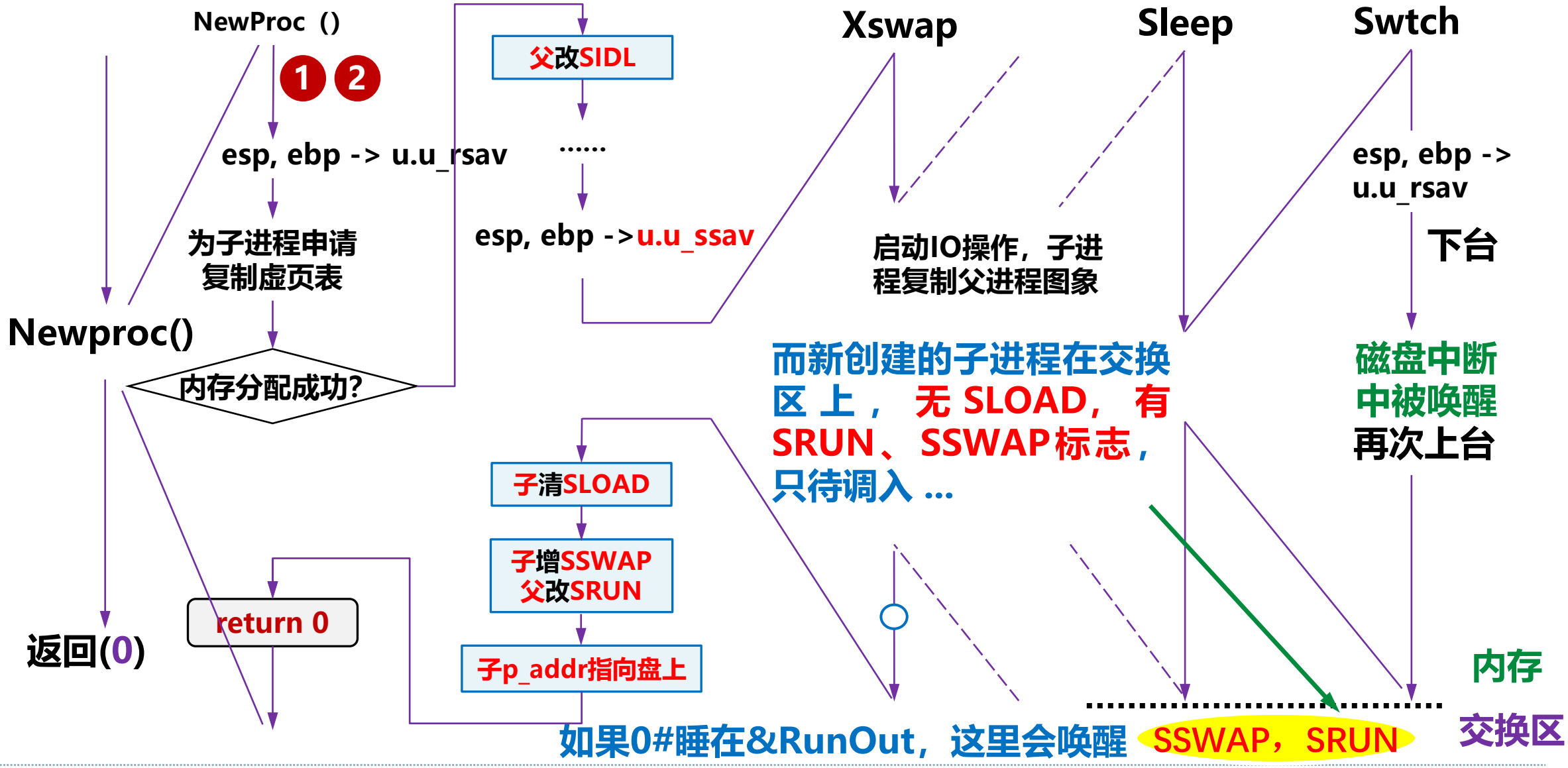




父进程执行Xswap将自己的进程图像复制到盘交换区，保留自己在内存的图像

唤醒有可能因为Run-Out在睡觉的0#进程

内存空间不足时的新进程创建

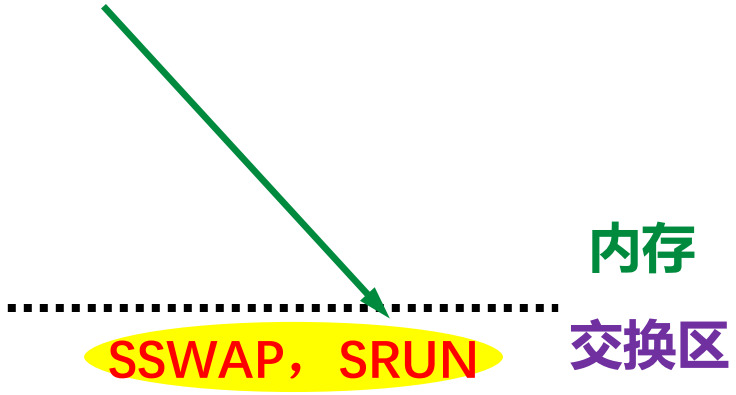


内存空间不足时的新进程创建

如果0#睡在&RunIn，则未来的时钟中断或进程进入低睡时会将其叫醒

一旦0# 发现交换区上有就绪进程，迟早会调入。

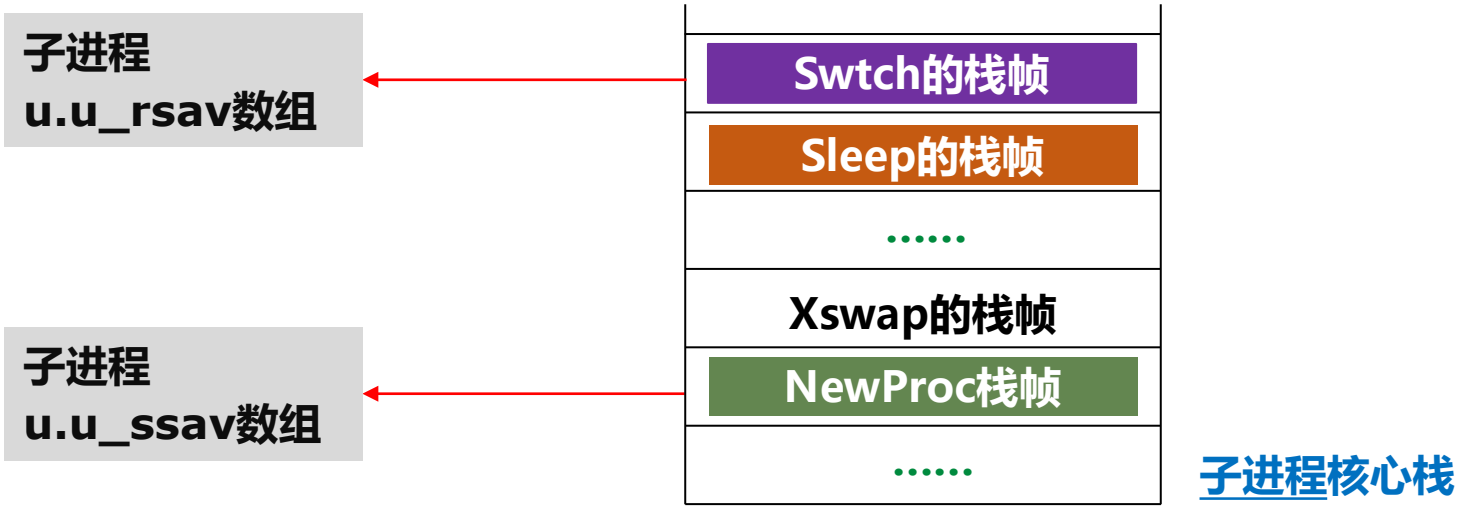
而新创建的子进程在交换区上，无 SLOAD，有SRUN、SSWAP标志，只待调入 …



内存空间不足时的新进程创建

0# 将它调入进内存后，会对它 |= SLOAD

子进程 SRUN、SLOAD，某一次Swtch中会被选中



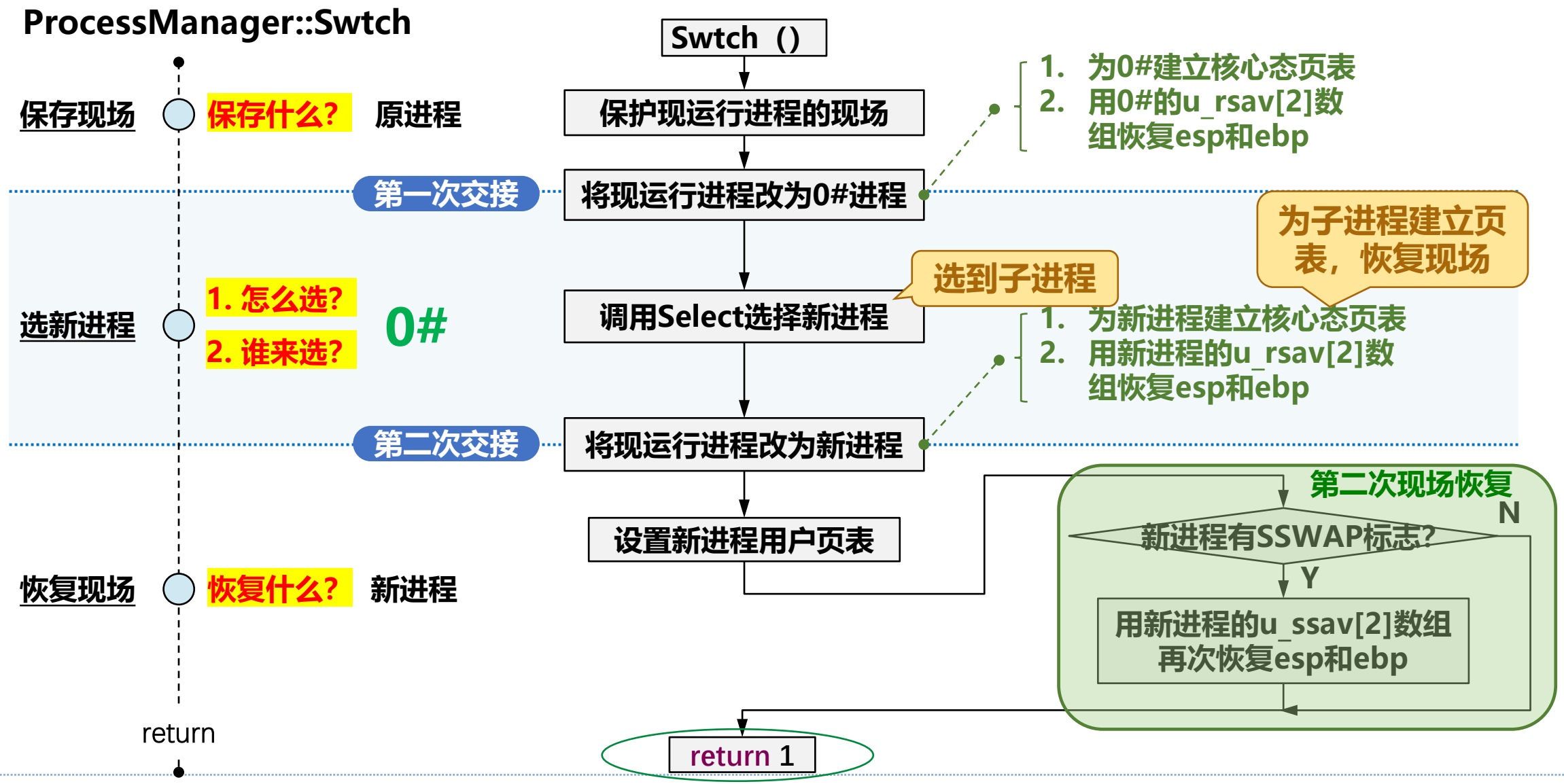
SLOAD

SSWAP，SRUN

内存

交换区

内存空间不足时的新进程创建



内存空间不足时的新进程创建

ProcessManager::Swtch

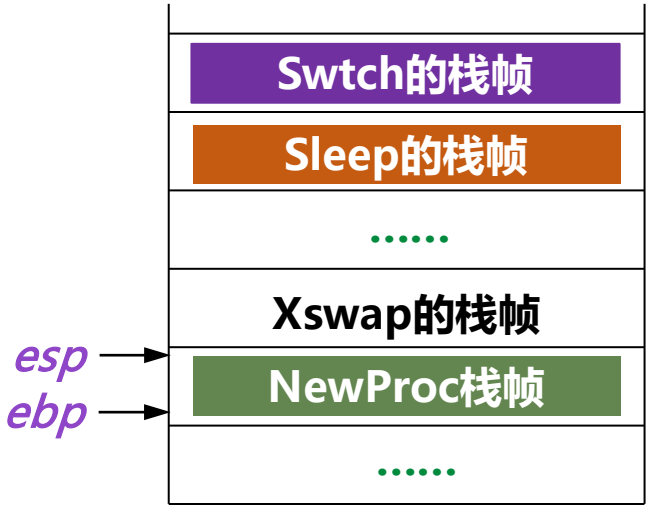
保存现场 — 保存什么？ — 原进程

选新进程 — 1. 怎么选？ 2. 谁来选？ — 0#

恢复现场 — 恢复什么？ — 新进程

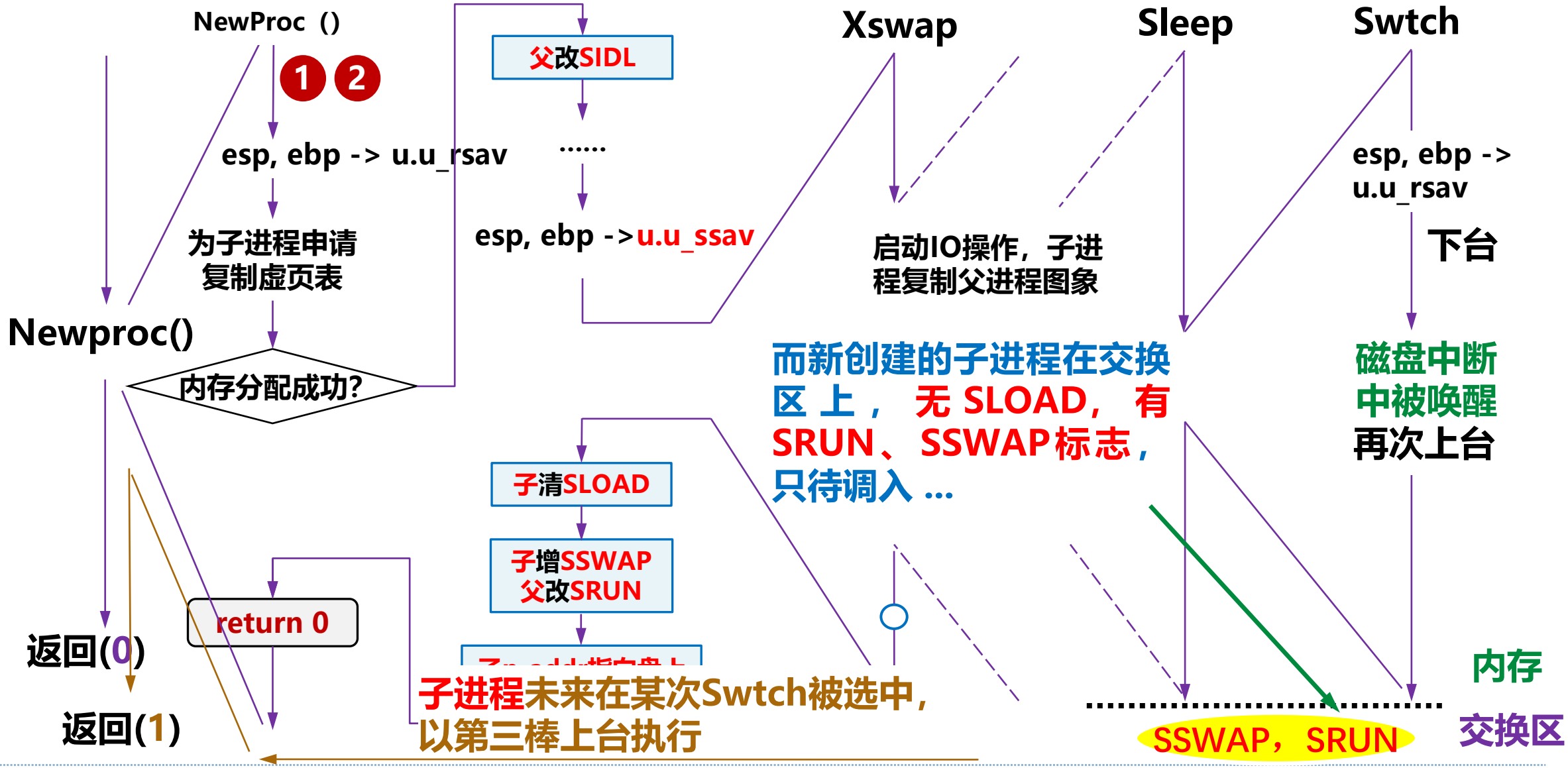
return 回哪里？

子进程：图象在内存，SSWAP，SRUN，p_pri=0；等待调度



Newproc的调用函数，调用位置的下一条语句
返回值为1

内存空间不足时的新进程创建



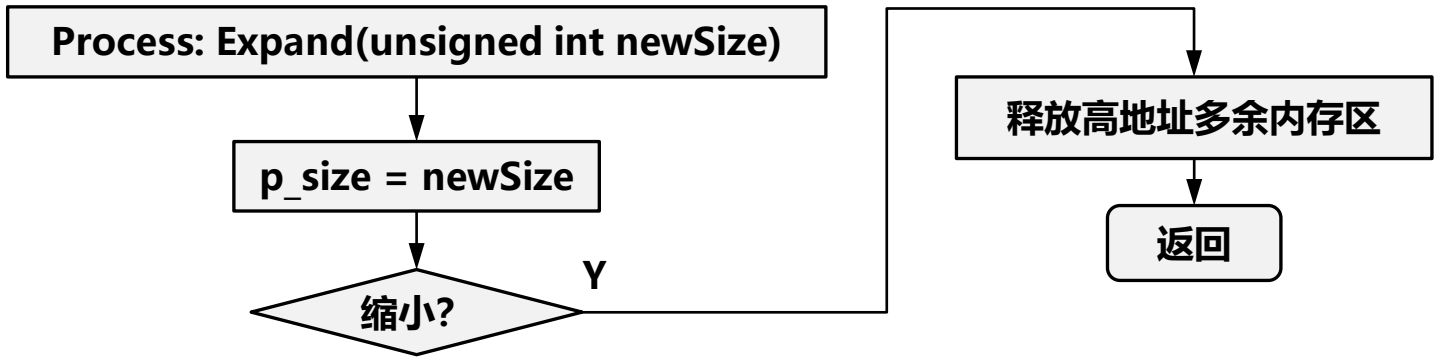
进程图像交换

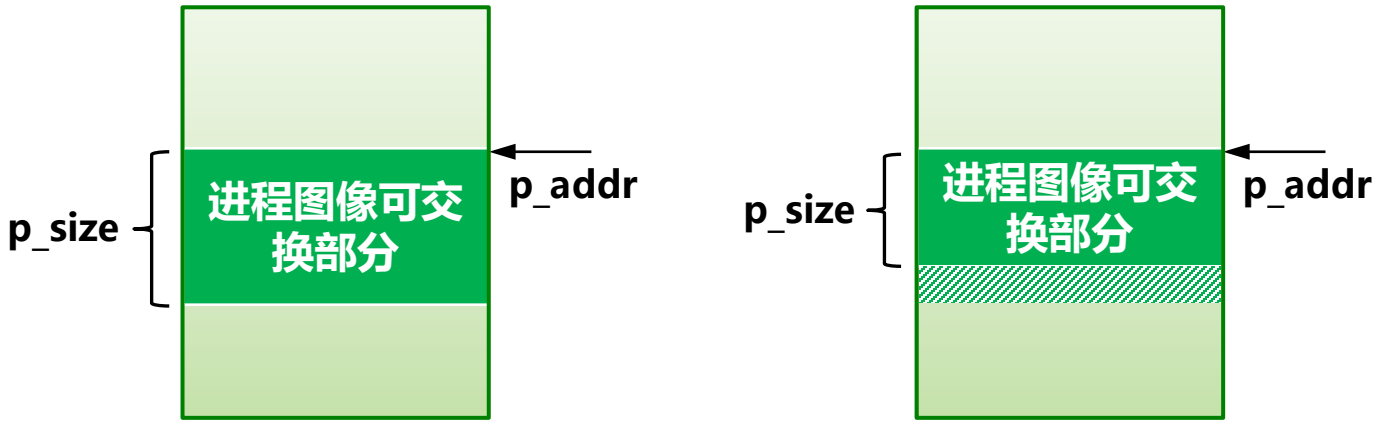
1 当内存空间不足以装下所有就绪进程时……

2 当内存空间不足以装下新建进程时……

3 当内存空间不足以让进程扩展自己的图像时……

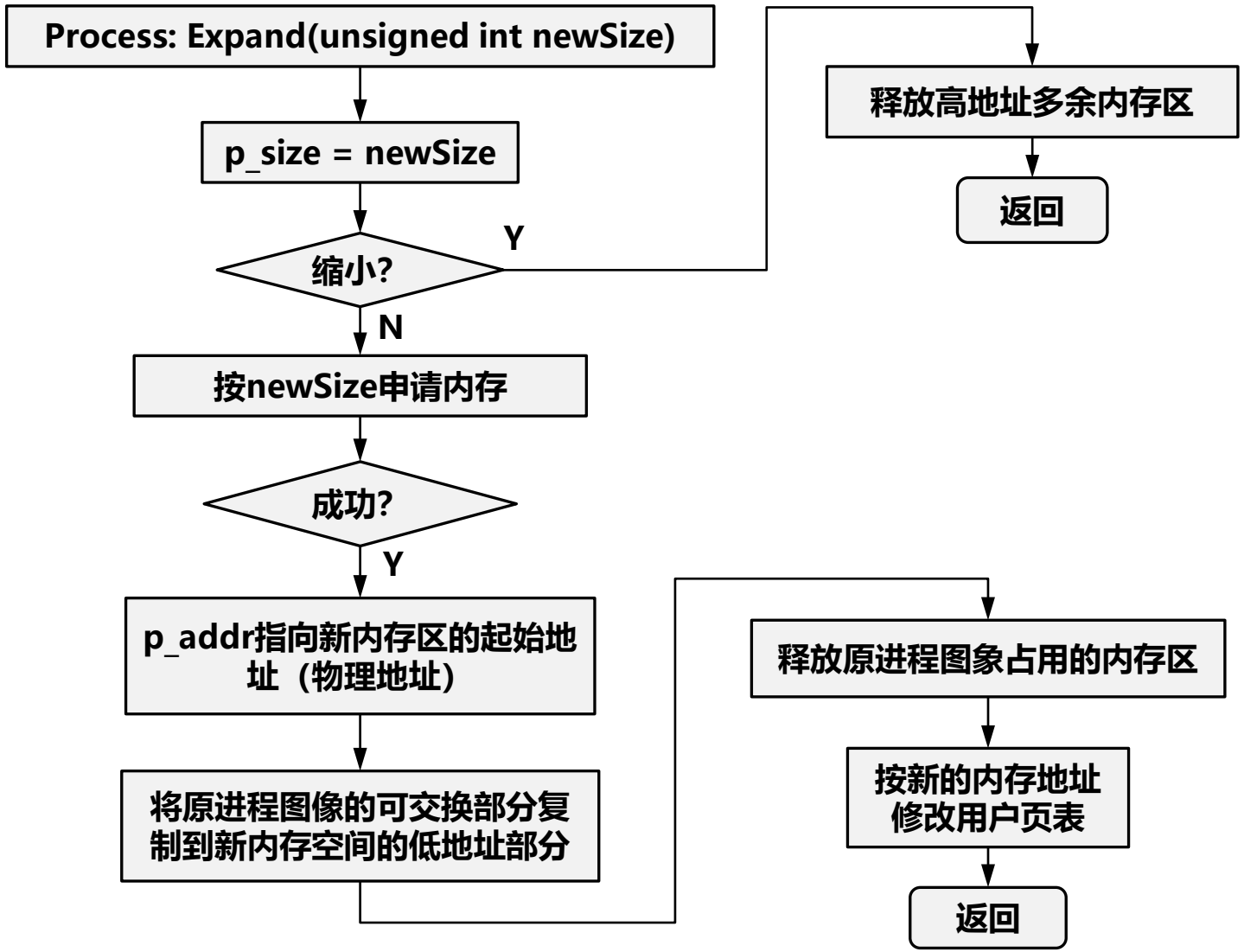
进程图像变化

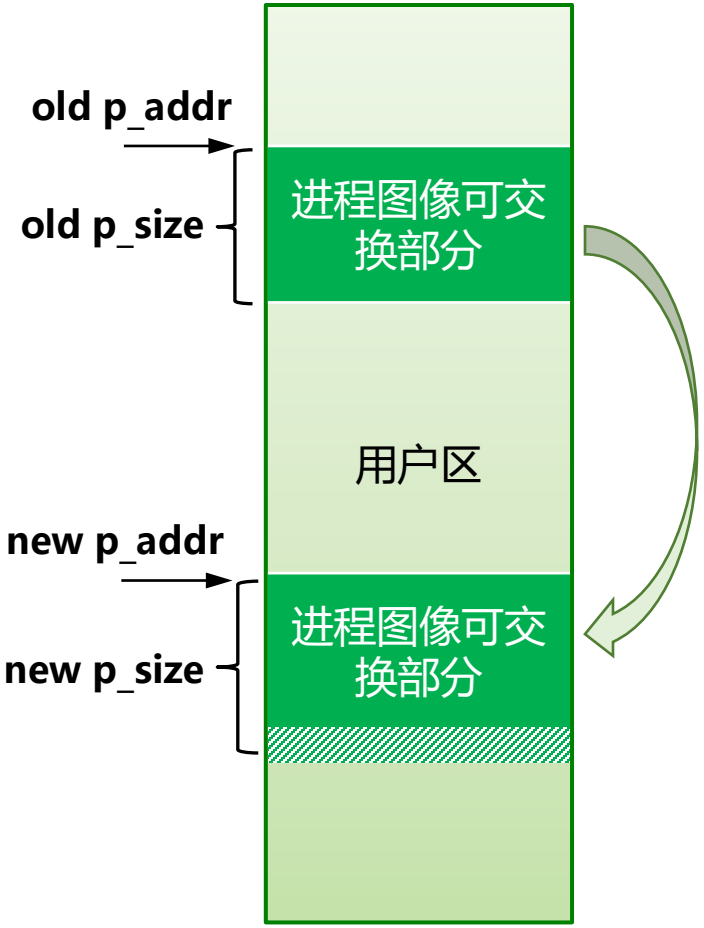




位置不变，高地址部分释放

进程图像变化


Process: Expand(unsigned int newSize)
p_size = newSize
缩小?
Y
N
释放高地址多余内存区
返回
按newSize申请内存
成功?
Y
p_addr指向新内存区的起始地址（物理地址）
将原进程图像的可交换部分复制到新内存空间的低地址部分
释放原进程图象占用的内存区
按新的内存地址修改用户页表
返回

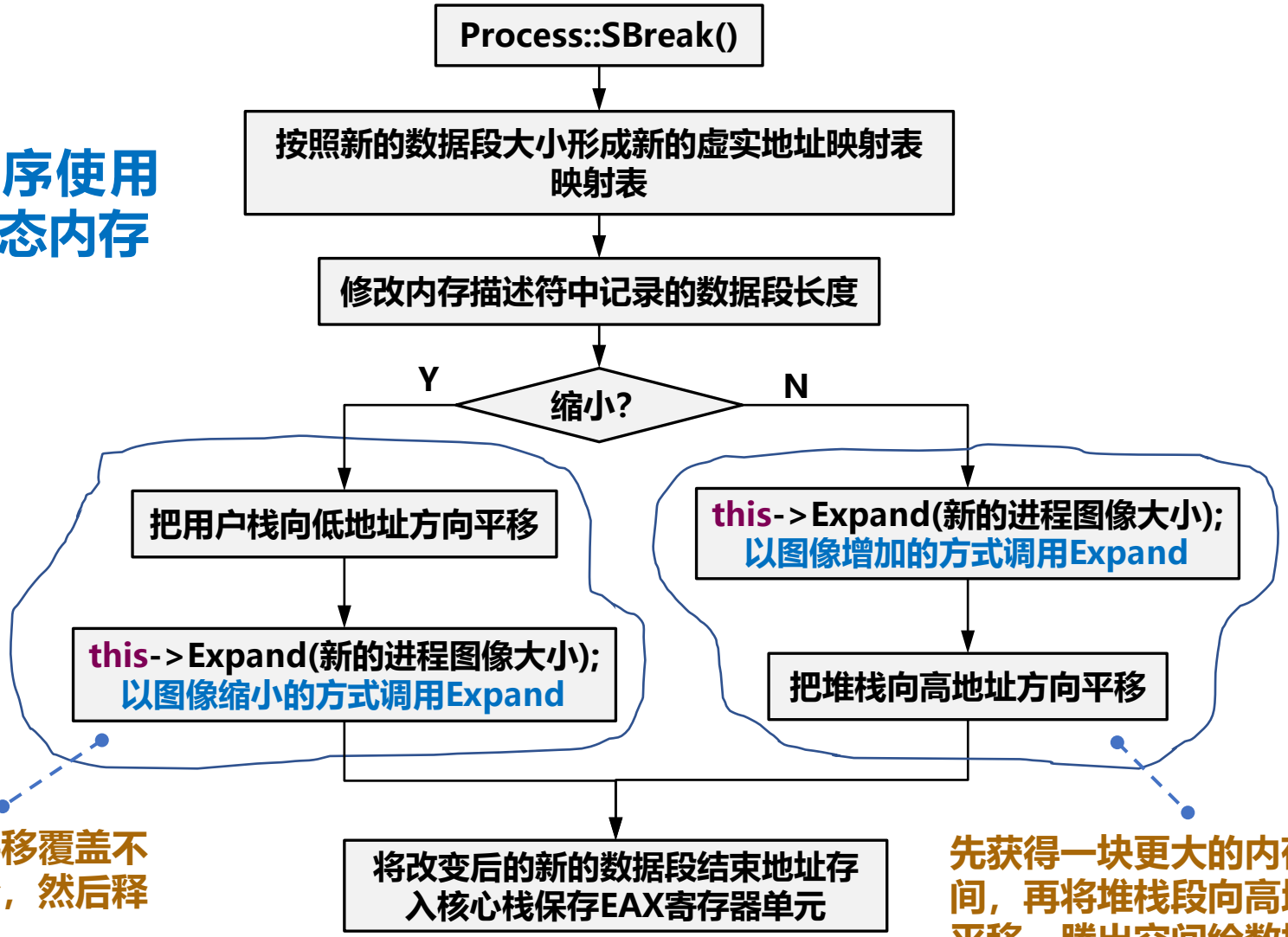

old p_addr
old p_size
进程图像可交换部分
用户区
new p_addr
new p_size
进程图像可交换部分

进程图像变化

数据段虚地址空间的变化：

UNIX V6++中，应用程序使用malloc函数为进程申请动态内存

malloc函数执行系统调用sbrk扩展用户数据段，将整数个虚拟页面追加在数据段尾部，扩展出的所有空间组成堆（heap）。



用户栈先向低地址平移覆盖不再需要的数据段部分，然后释放高地址部分

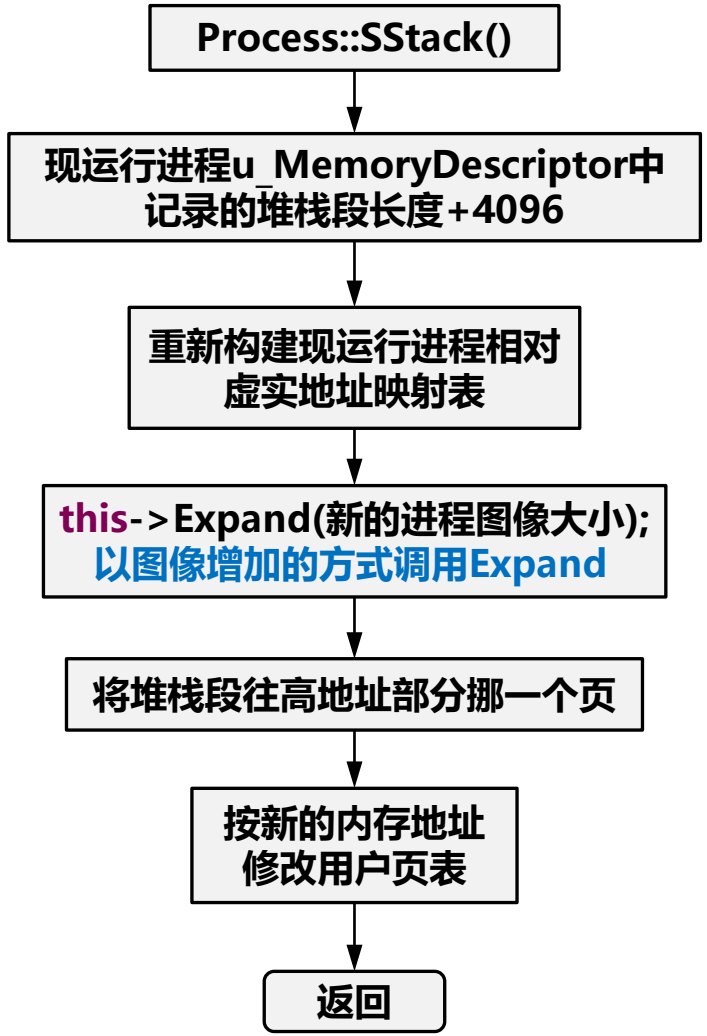
先获得一块更大的内存空间，再将堆栈段向高地址平移，腾出空间给数据段

进程图像变化

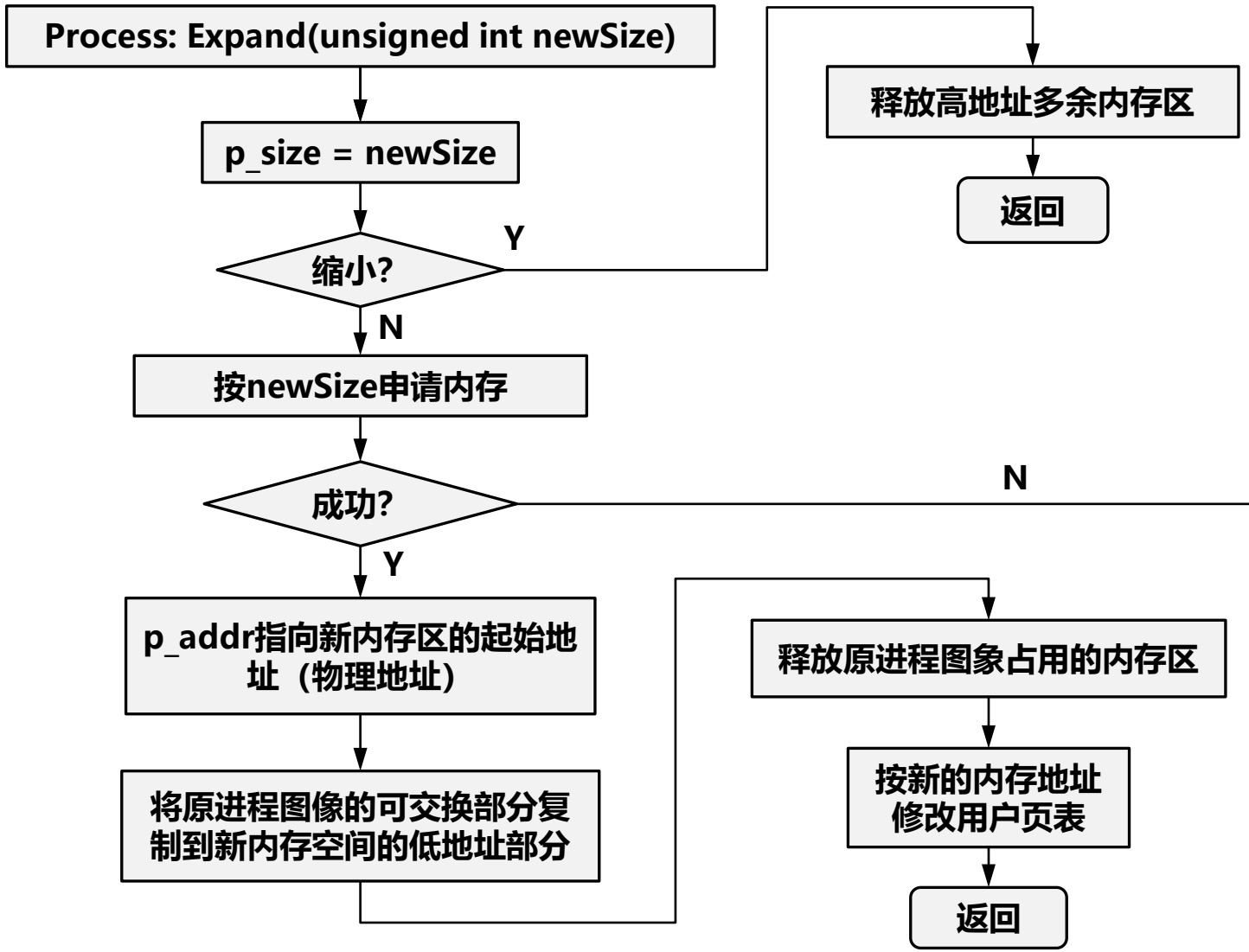
用户段虚地址空间的变化：

根据子程序调用的嵌套深度，进程的用户栈自动扩展

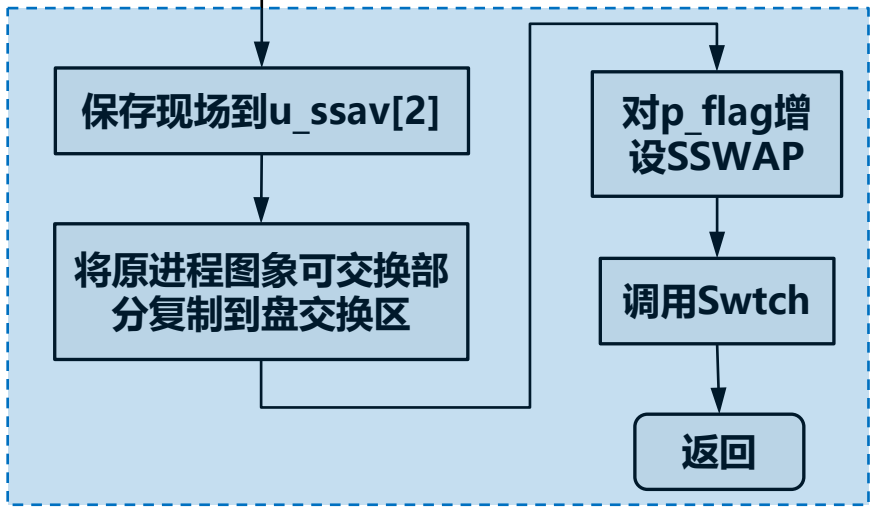
栈扩展是缺页异常引发的，进程不需要执行显式的系统调用



进程图像变化

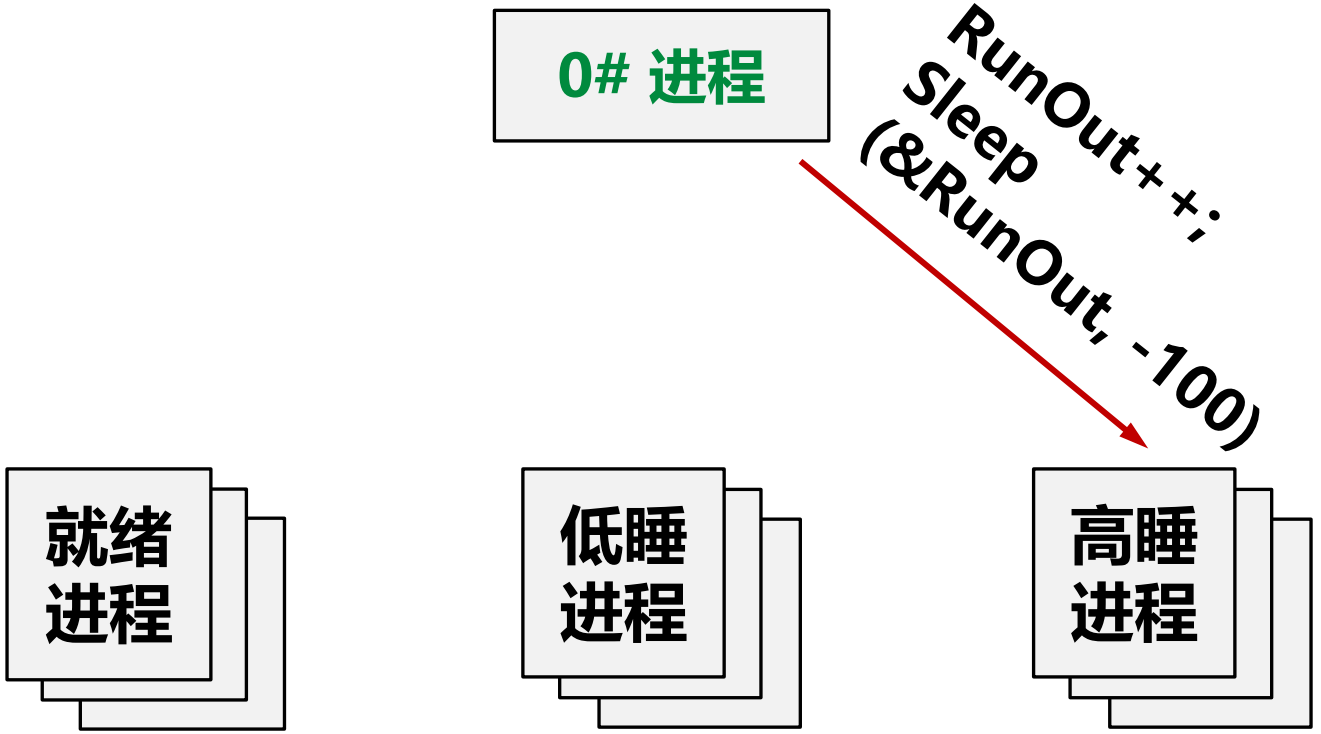


这里为什么要类似新进程创建在盘交换区时的二次现场保护处理?

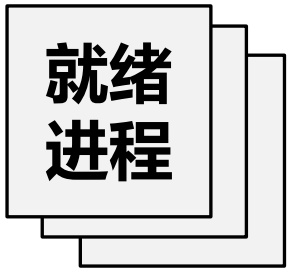


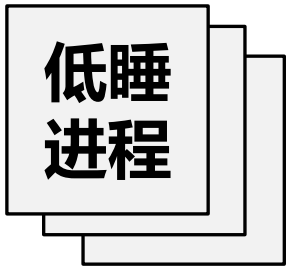
你必须知道的关于0#进程的细节

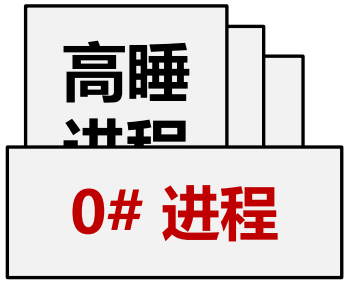
关于 0# 进程



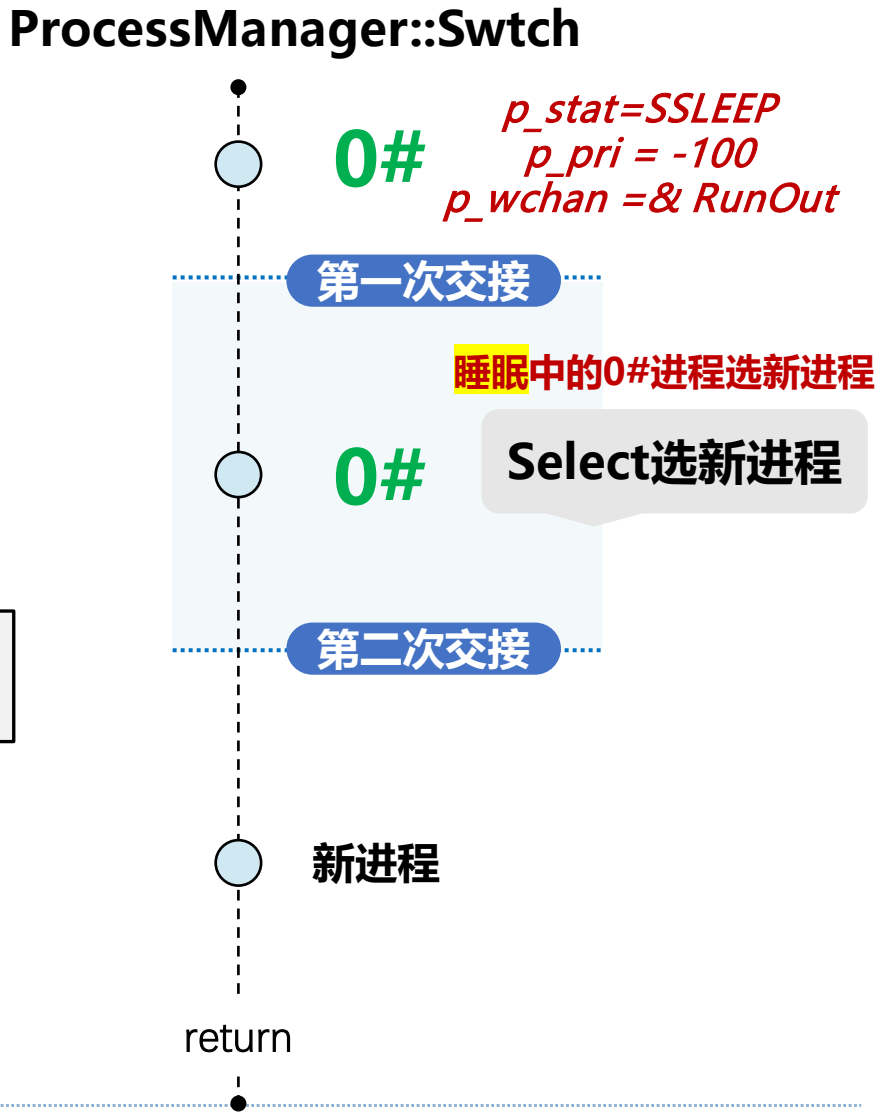
关于 0# 进程



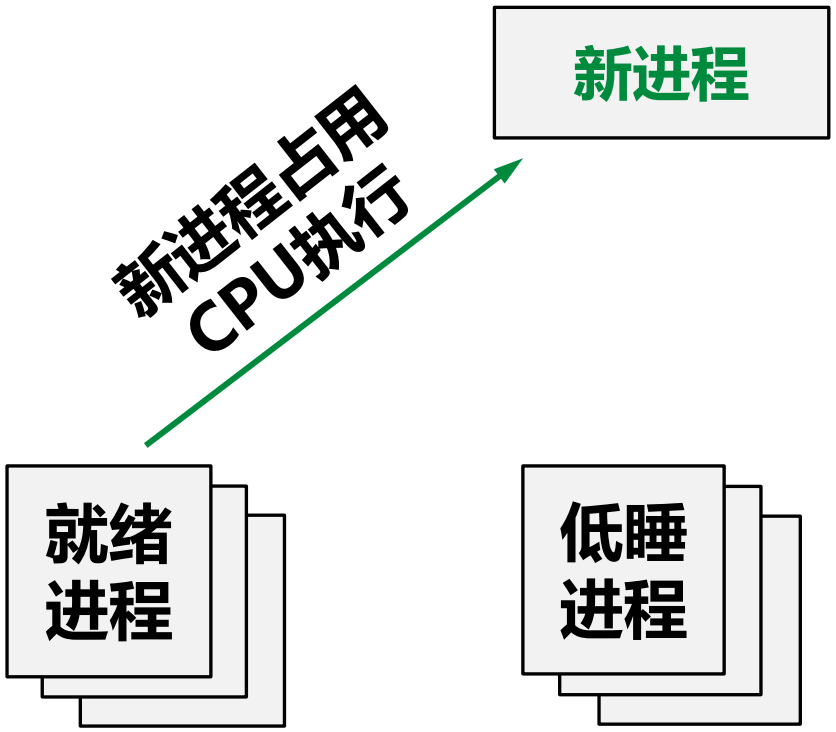


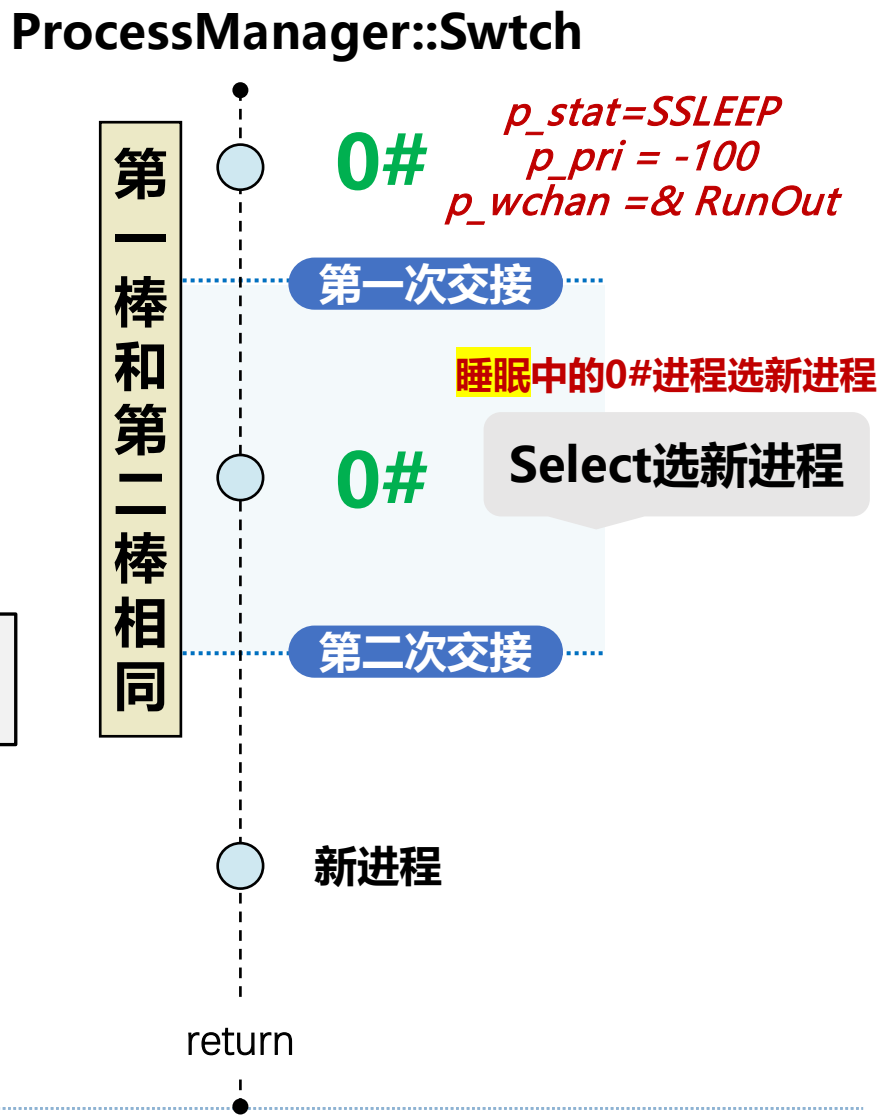



```
Sleep{
…
Swtch;
…
}
```



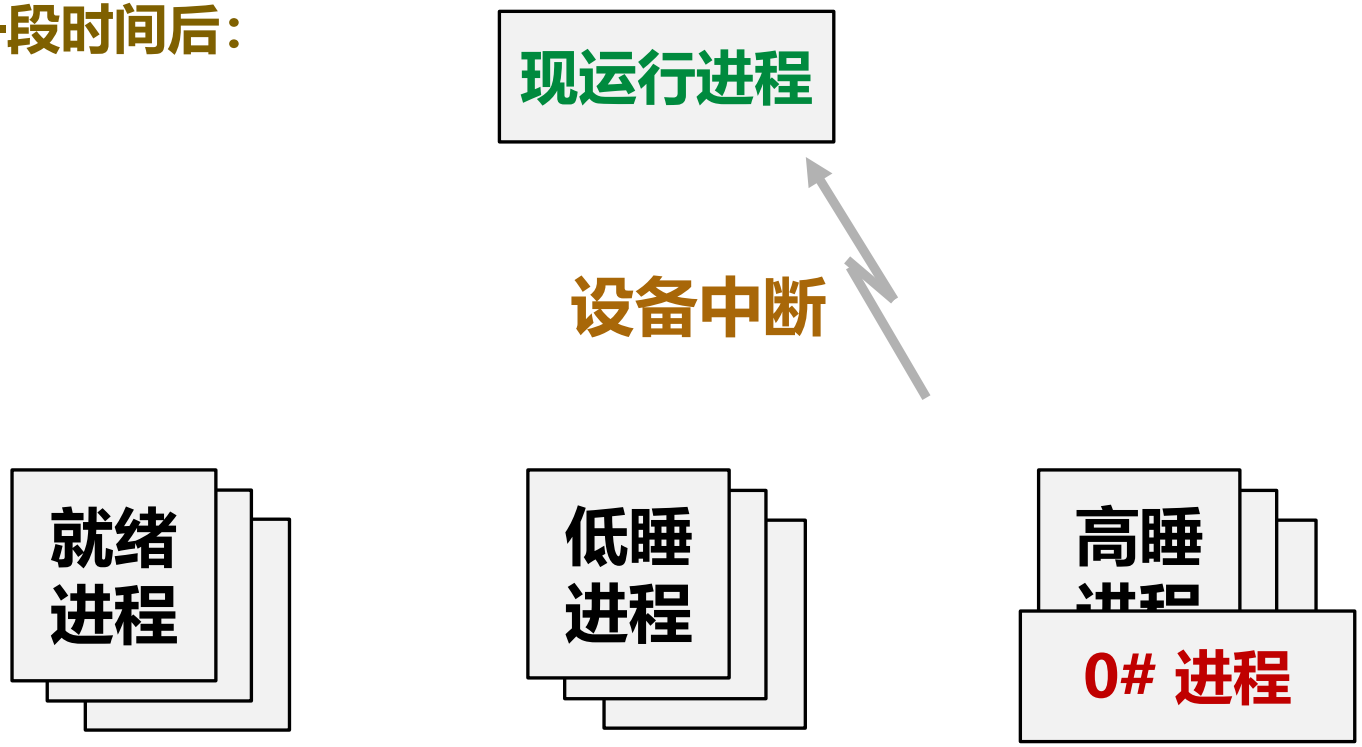
关于 0# 进程


新进程
新进程占用CPU执行
就绪进程
低睡进程
高睡进程
0# 进程
Sleep{
…
Swtch;
…
}


ProcessManager::Swtch
第一棒和第二棒相同
0#
p_stat=SSLEEP
p_pri = -100
p_wchan =& RunOut
第一次交接
睡眠中的0#进程选新进程
0#
Select选新进程
第二次交接
新进程
return

关于 0# 进程

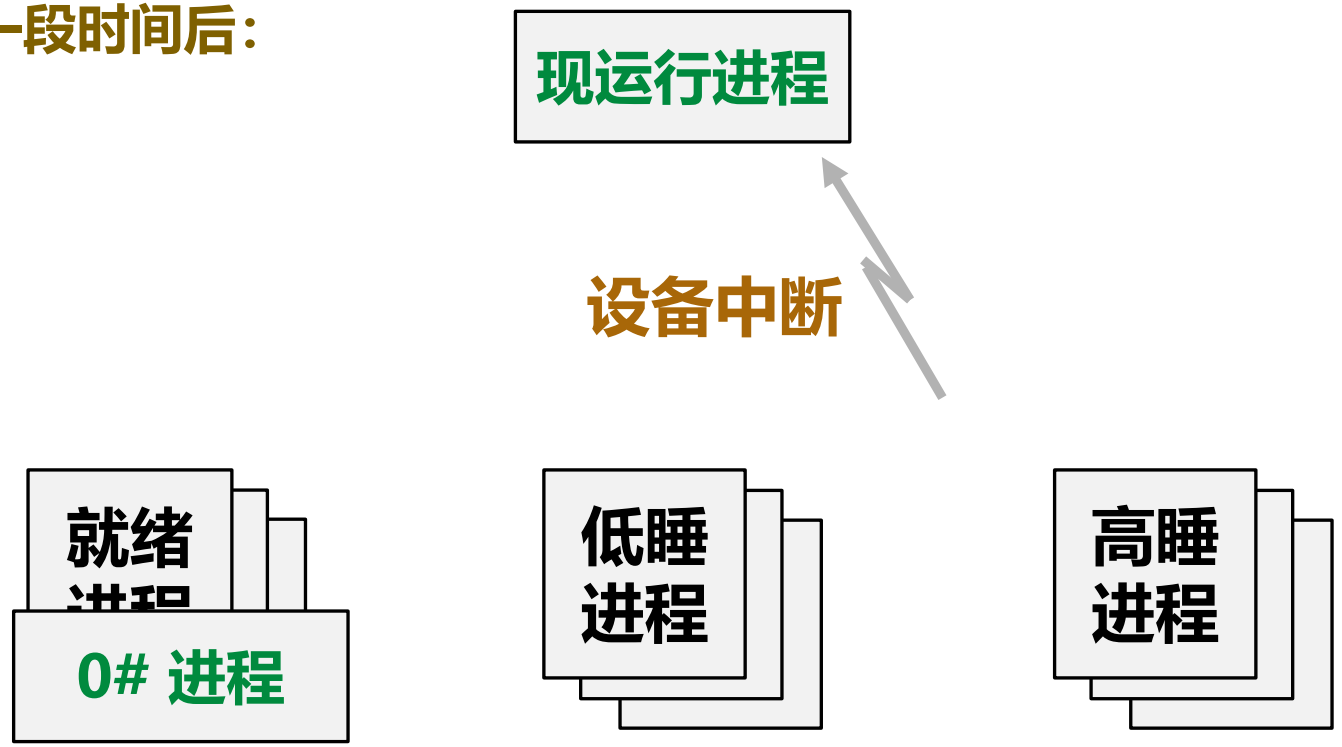
一段时间后：



中断中如果唤醒一个磁盘进程，则：
0#进程被唤醒；RunRun被设置

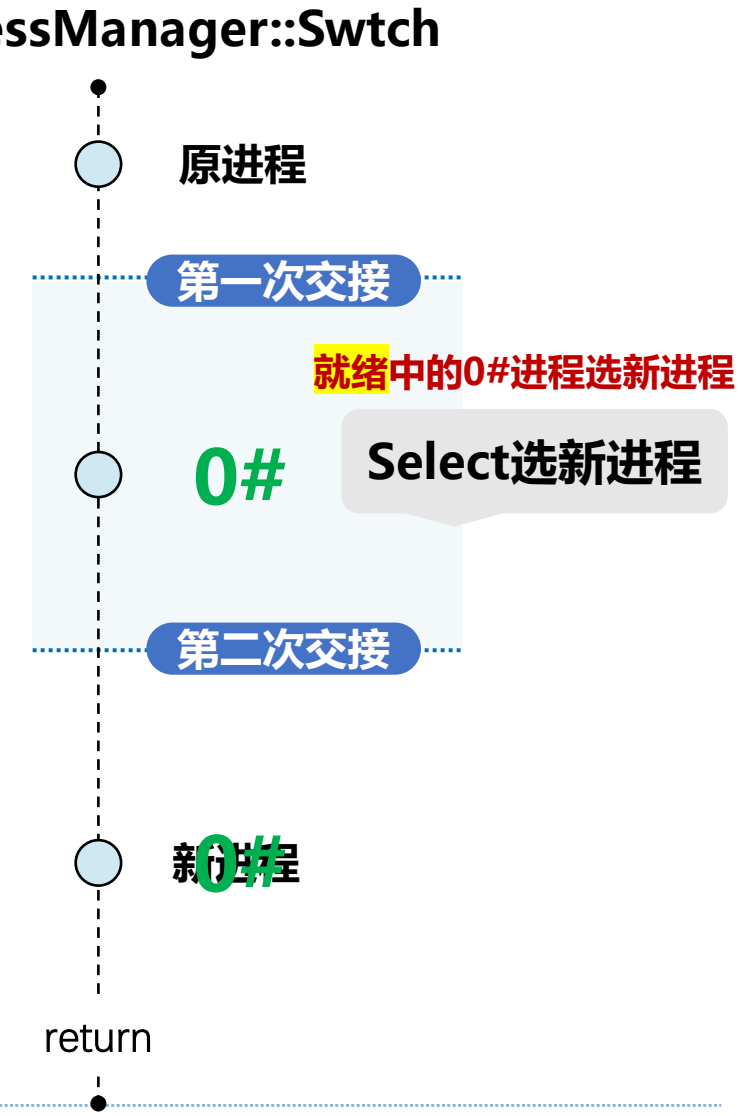
关于 0# 进程

一段时间后：



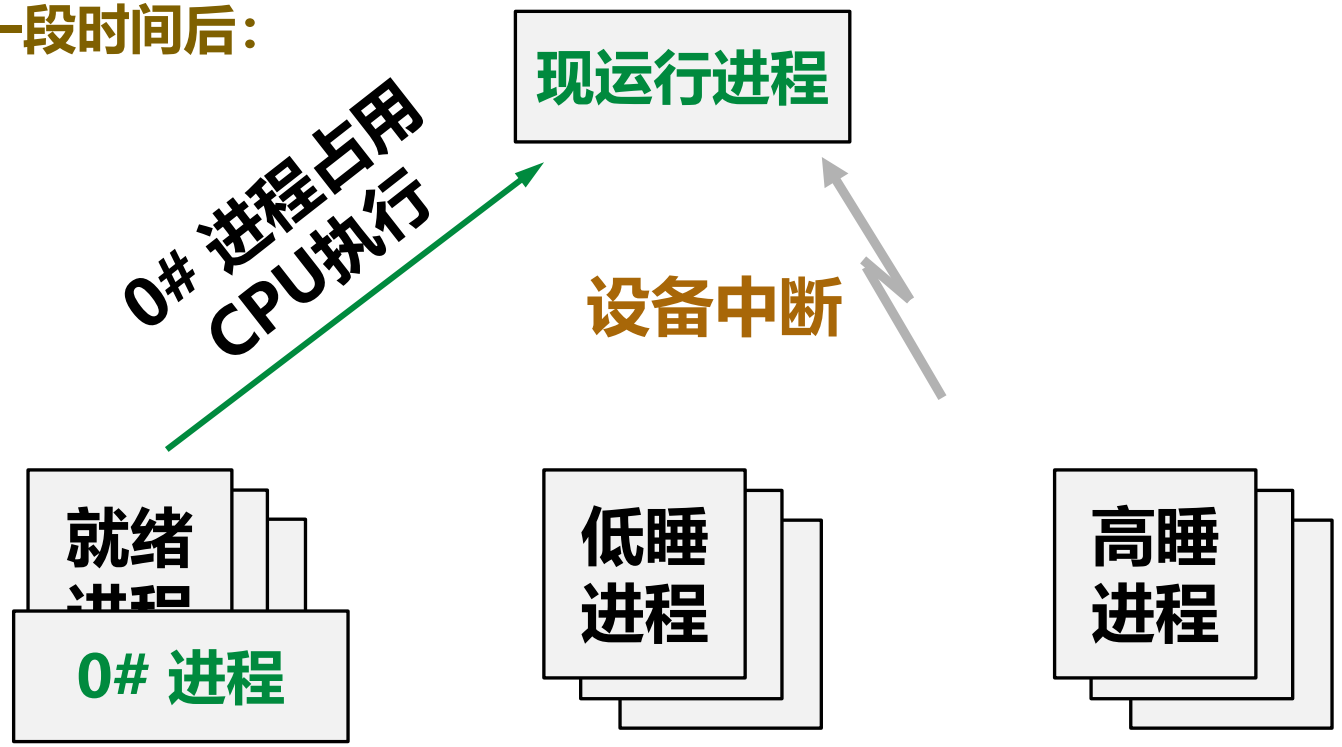
中断中如果唤醒一个磁盘进程，则：
0#进程被唤醒；RunRun被设置

中断返回用户态前，Swtch



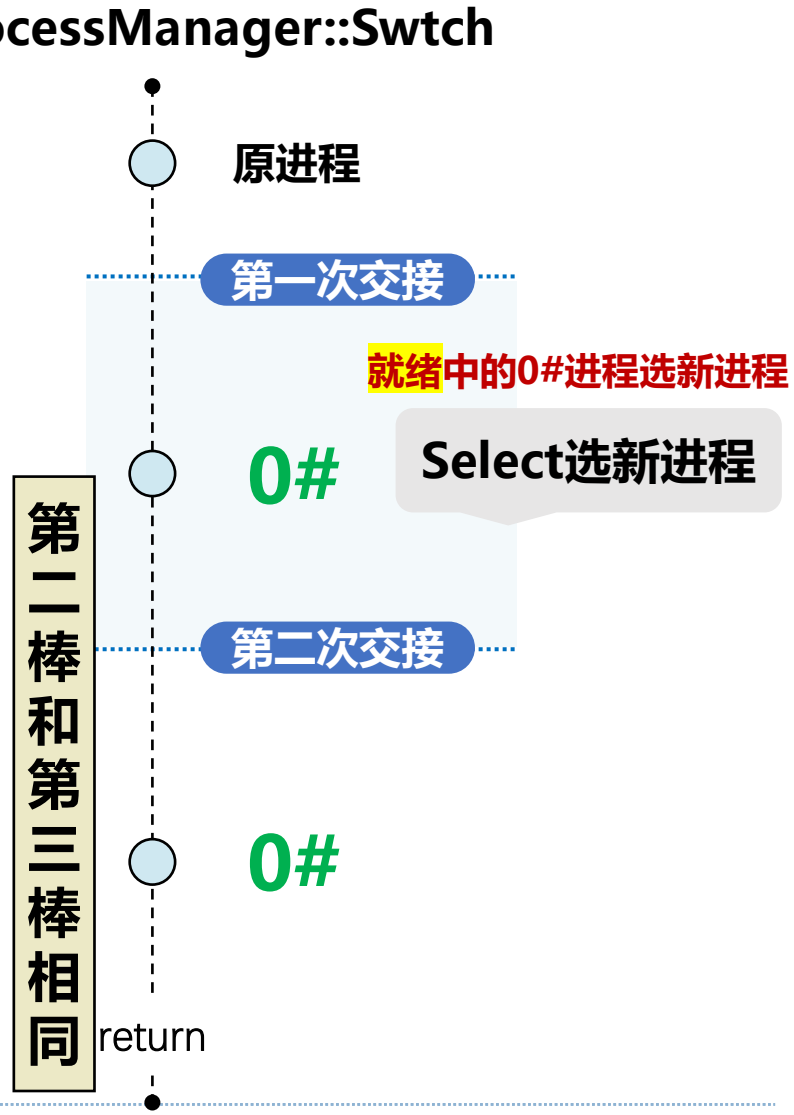
关于 0# 进程

一段时间后：



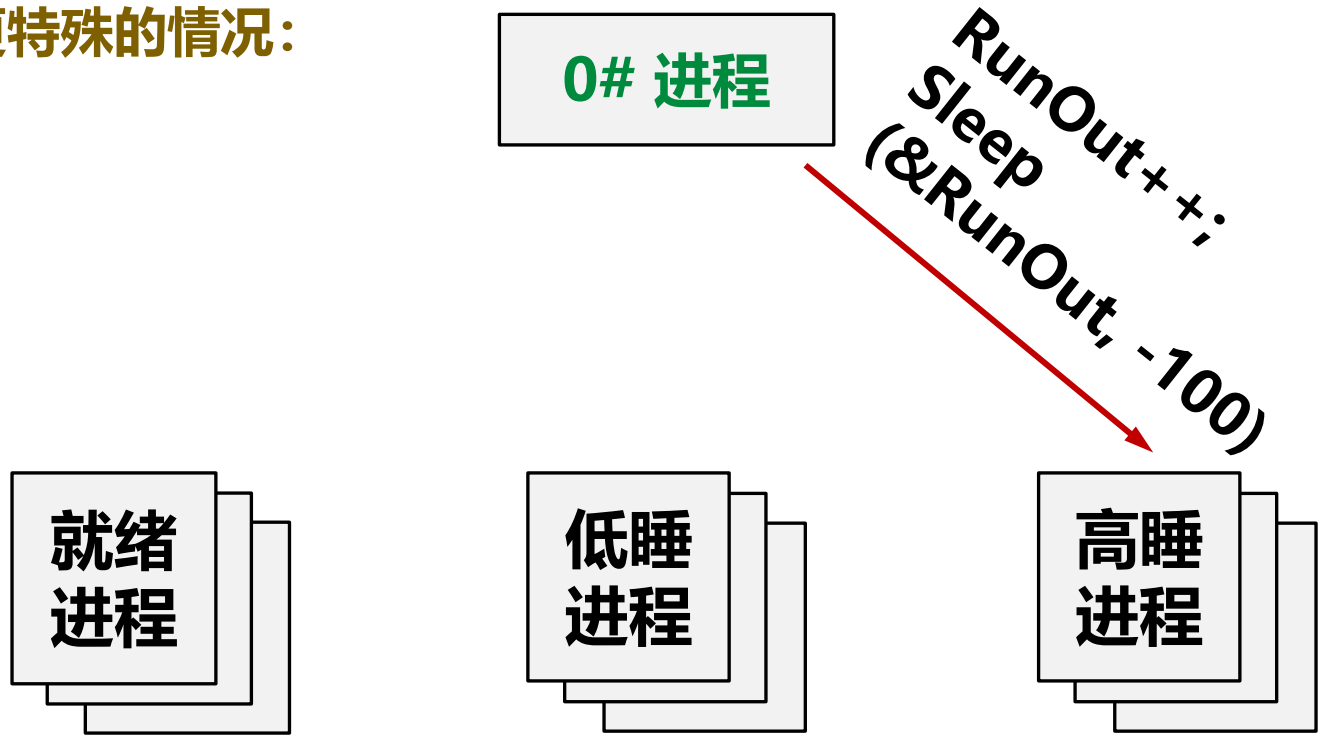
中断中如果唤醒一个磁盘进程，则：
0#进程被唤醒；RunRun被设置

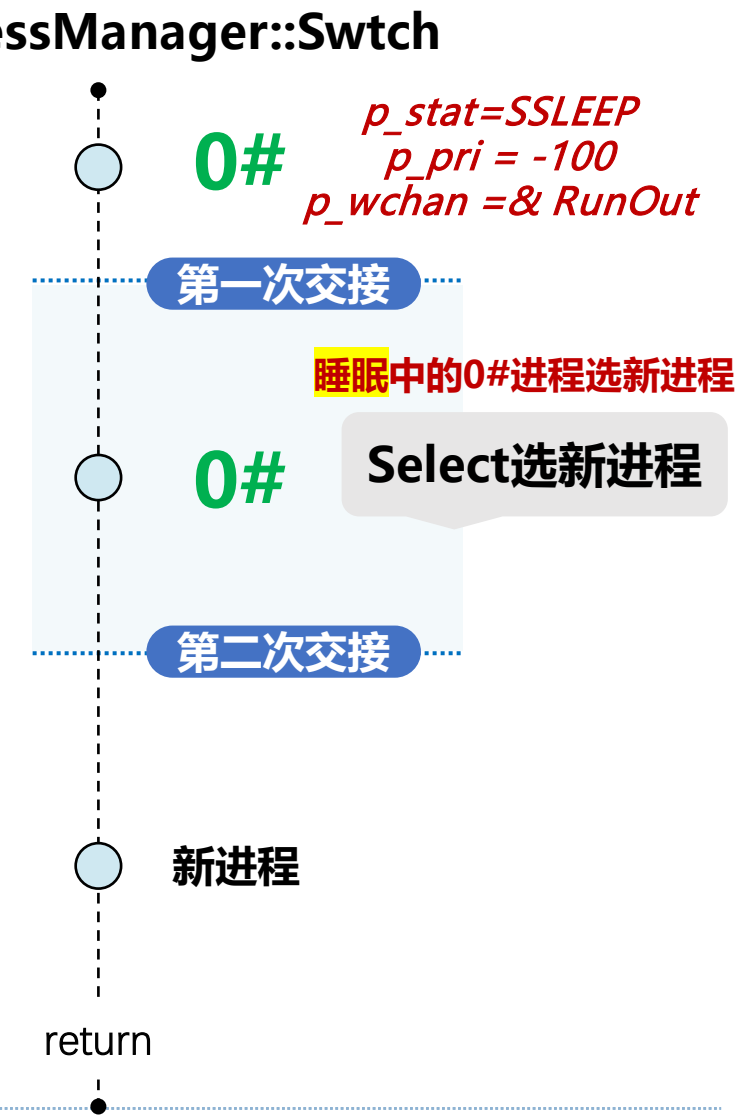
中断返回用户态前，Swtch



关于 0# 进程

更特殊的情况：

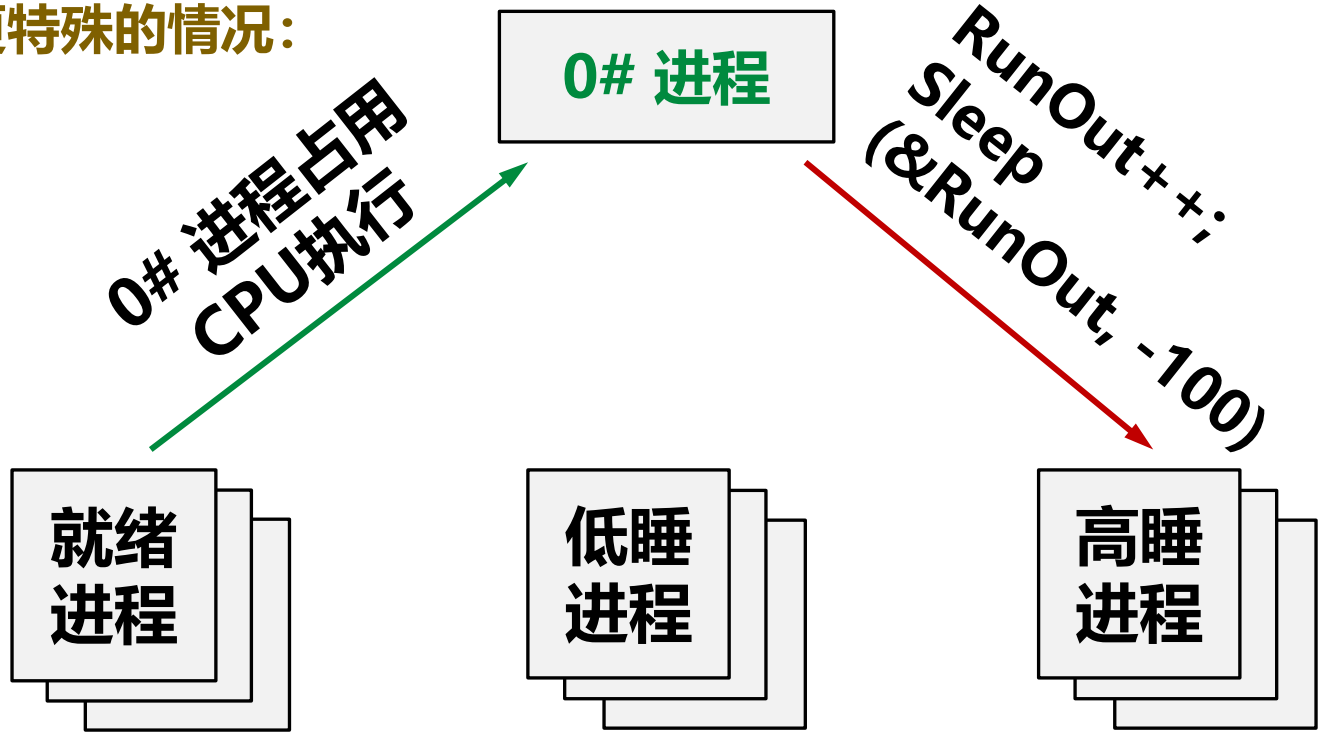




如果此时没有就绪进程，Select中等待中断；
中断中如果恰好唤醒一个盘交换区睡眠的进程，则同时可能唤醒0#。
中断返回后。。。。。。

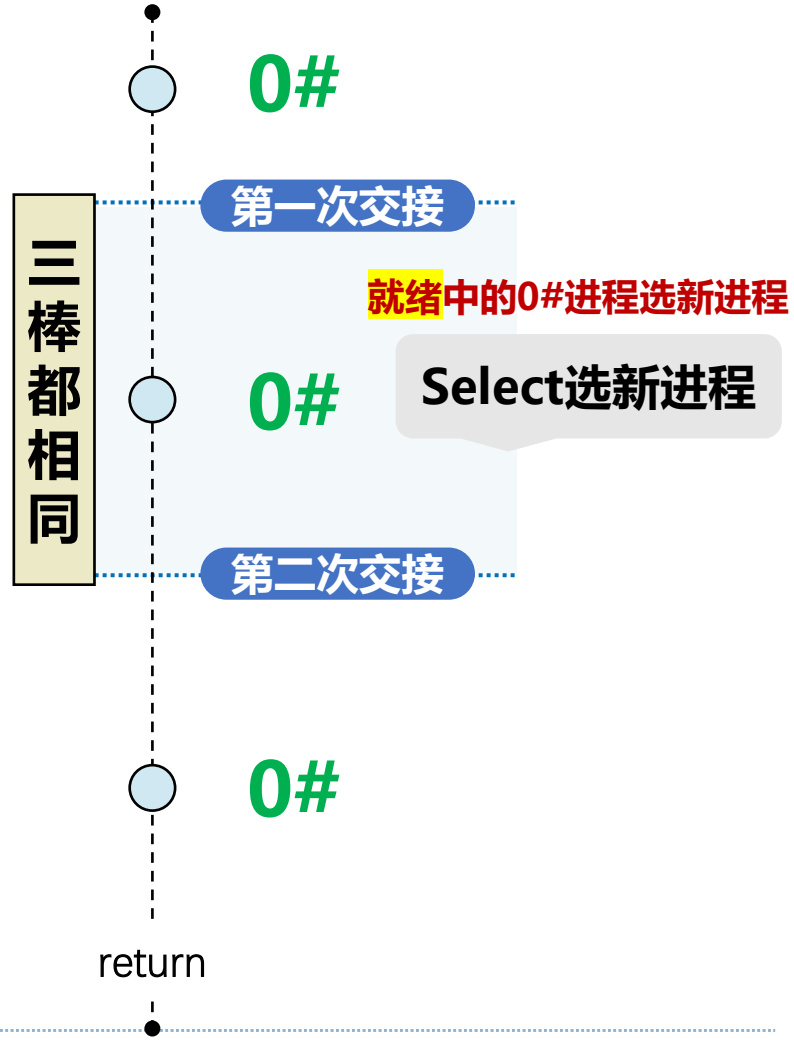
关于 0# 进程

更特殊的情况：



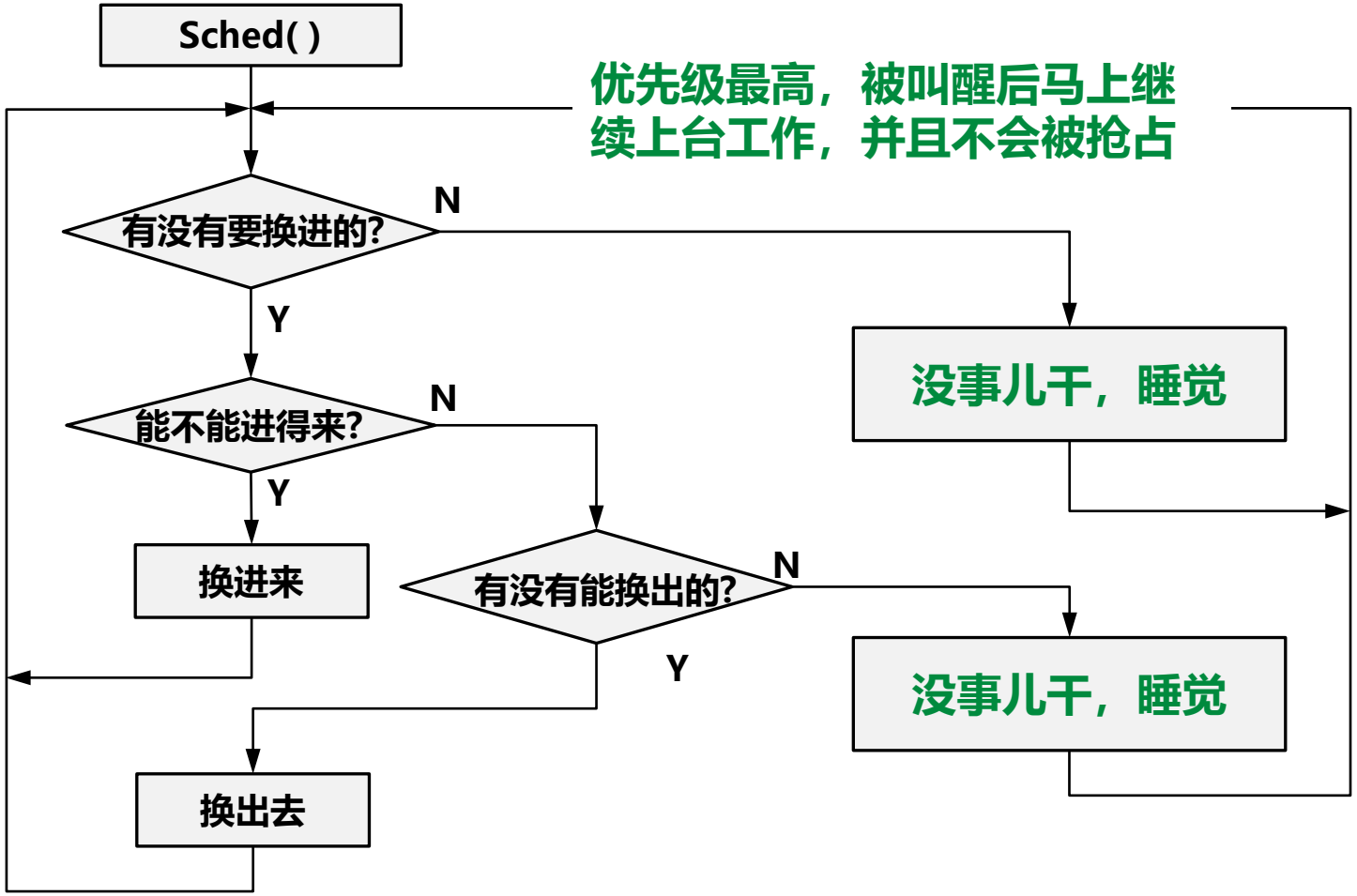
醒来的0# 选到自己上台

ProcessManager::Swtch



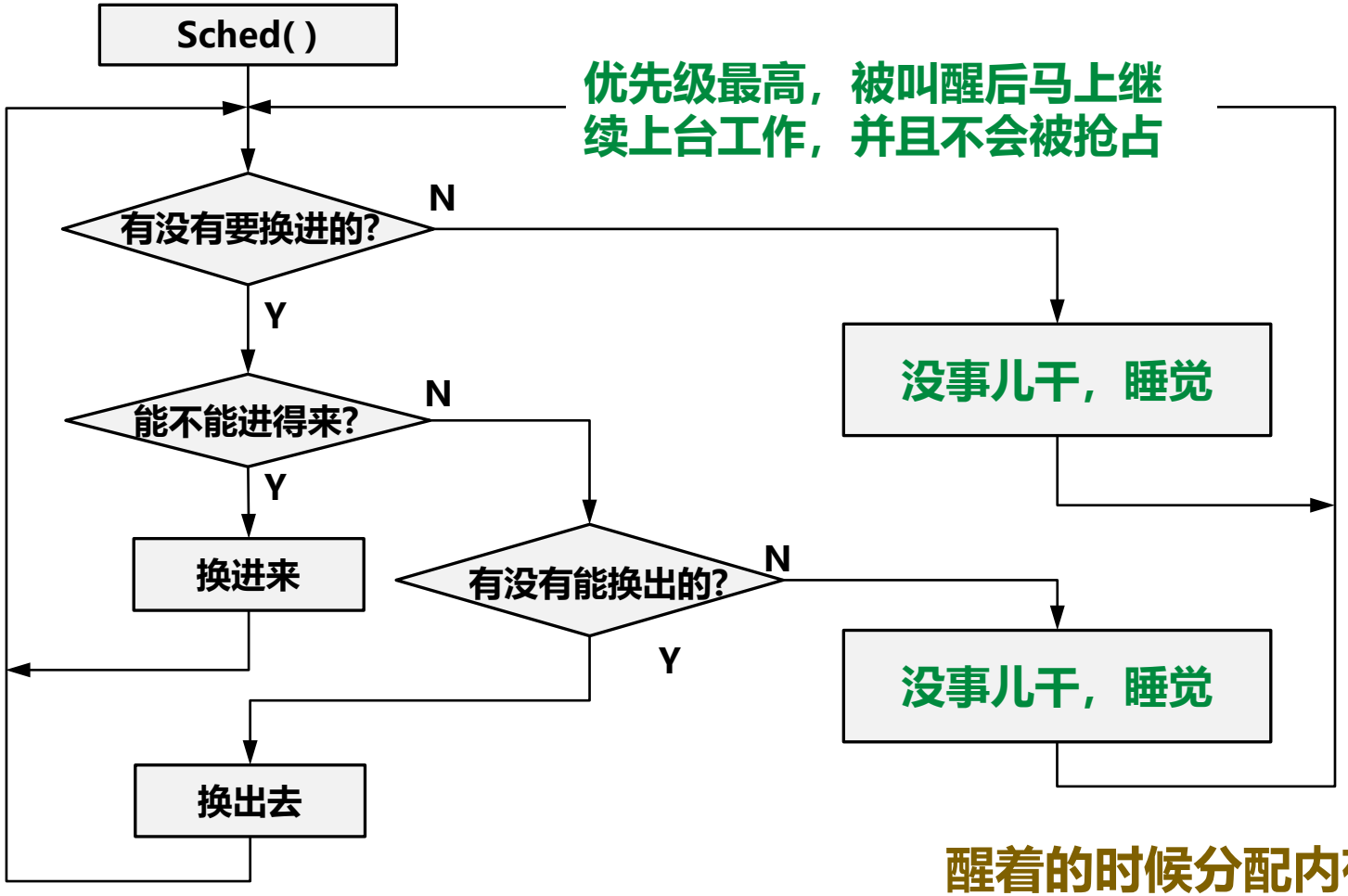
关于 0# 进程

0# 进程执行ProcessManager::Sched

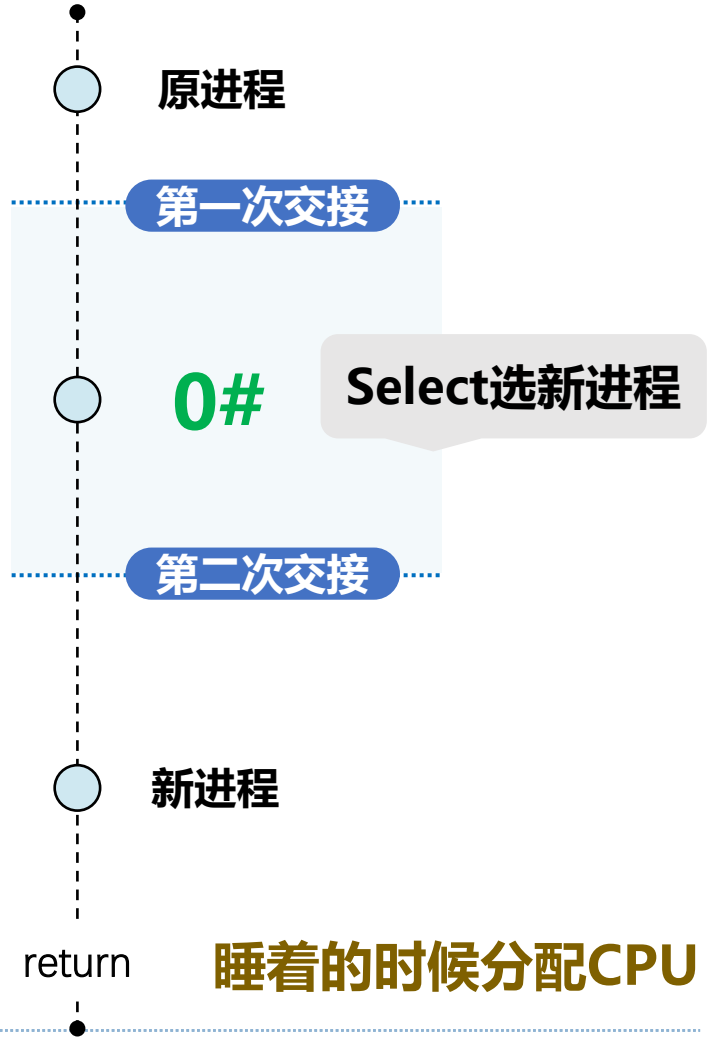


关于 0# 进程

0# 进程执行ProcessManager::Sched



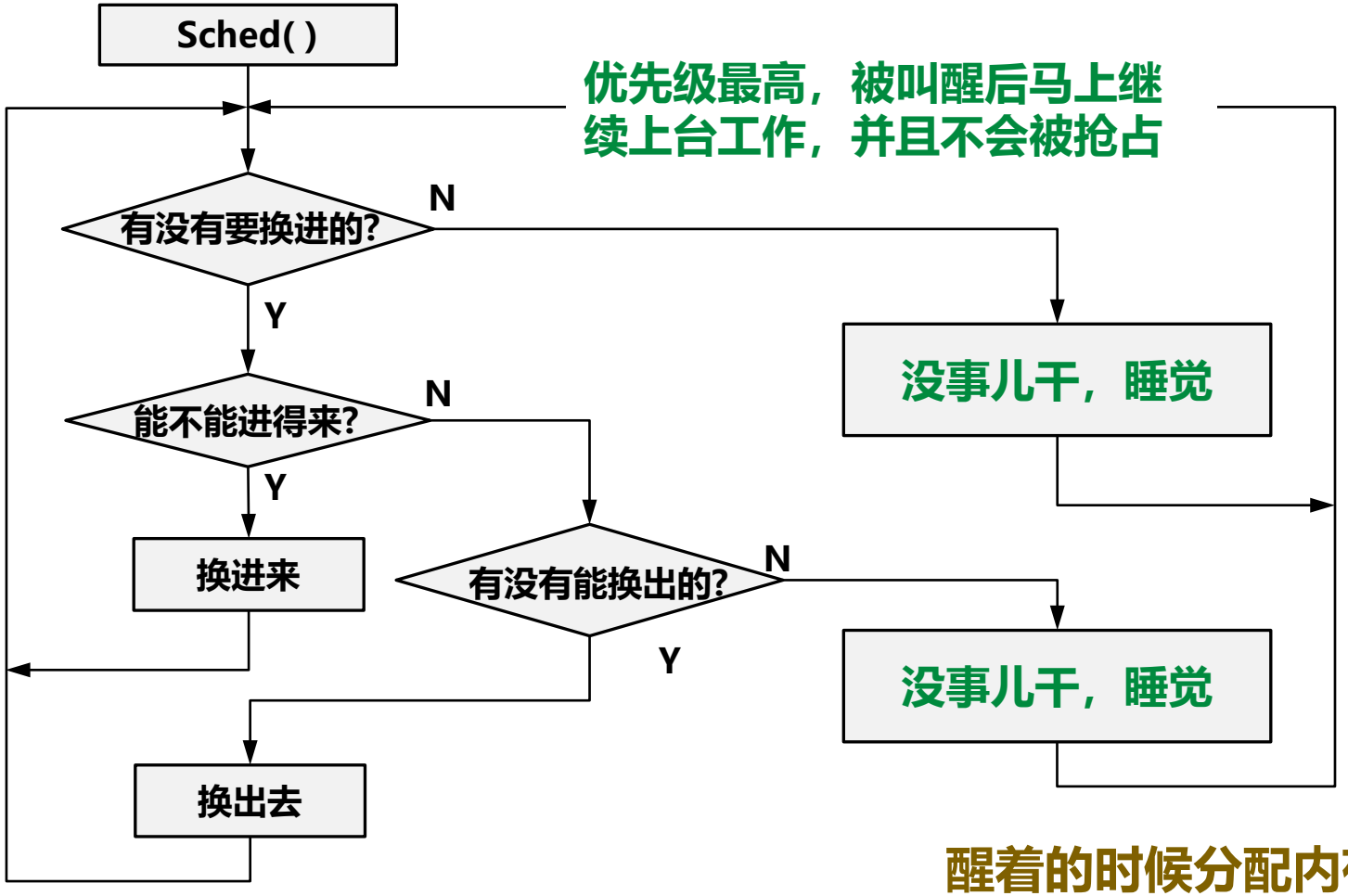
ProcessManager::Swtch



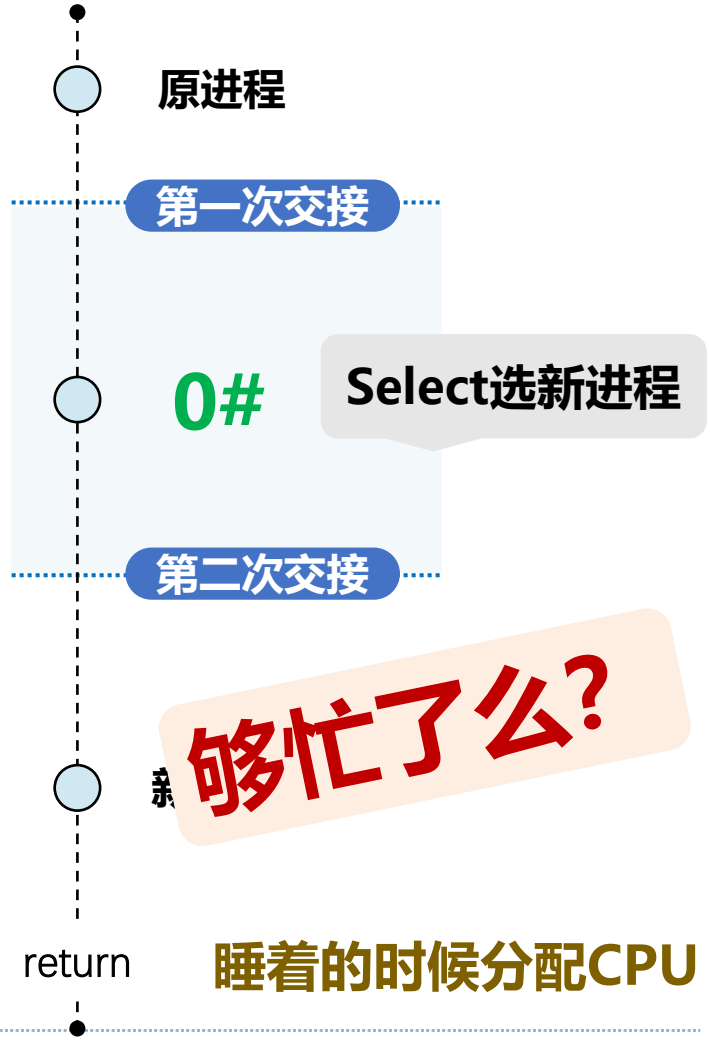
醒着的时候分配内存　　睡着的时候分配CPU

关于 0# 进程

0# 进程执行ProcessManager::Sched



ProcessManager::Swtch



醒着的时候分配内存　　睡着的时候分配CPU

关于 0# 进程

ProcessManager::Swtch

原进程

第一次交接

睡眠中的0#进程选新进程

0# Select选新进程 — 等中断

第二次交接

如果此时没有就绪进程

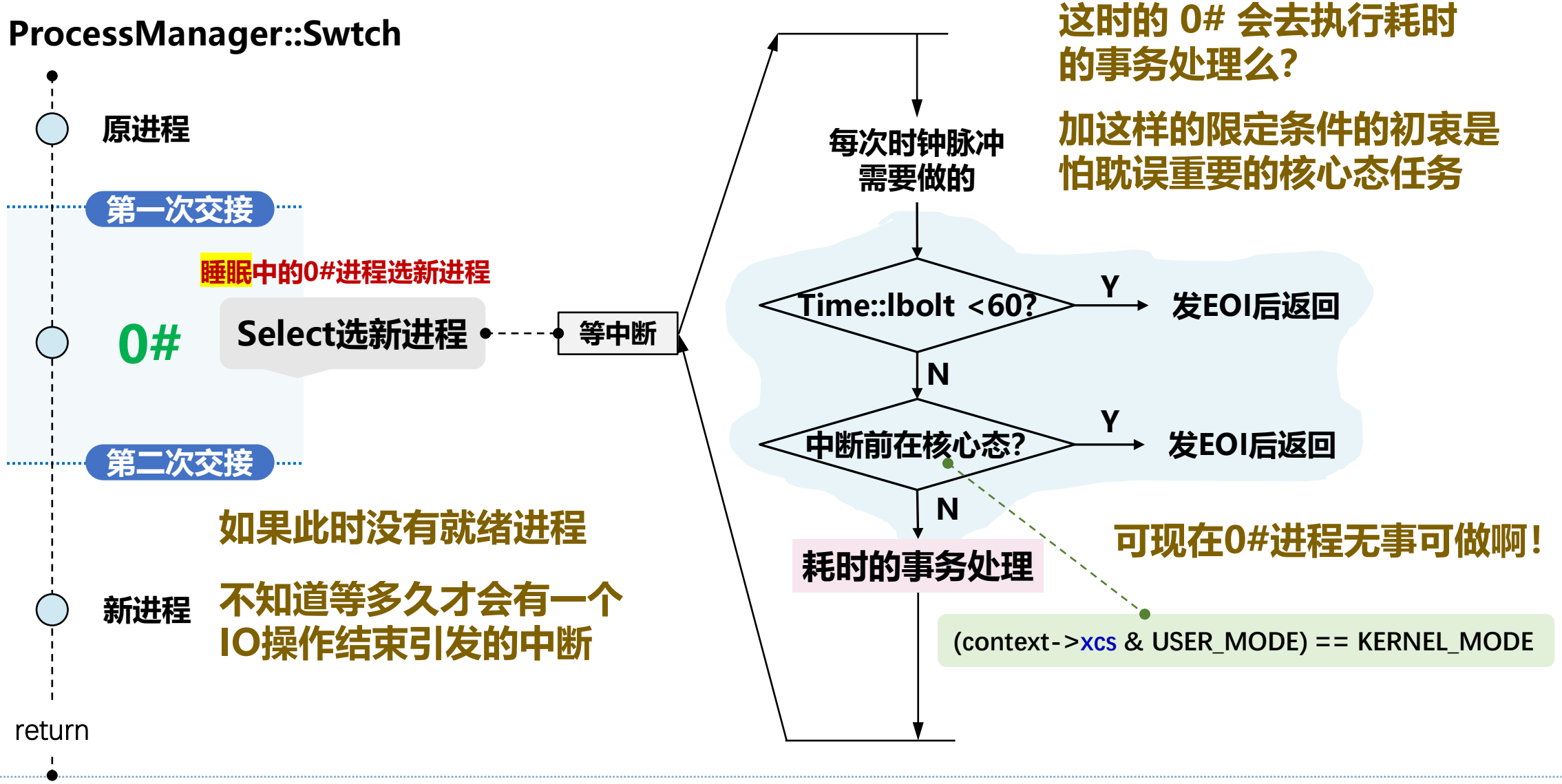
新进程

不知道等多久才会有一个IO操作结束引发的中断

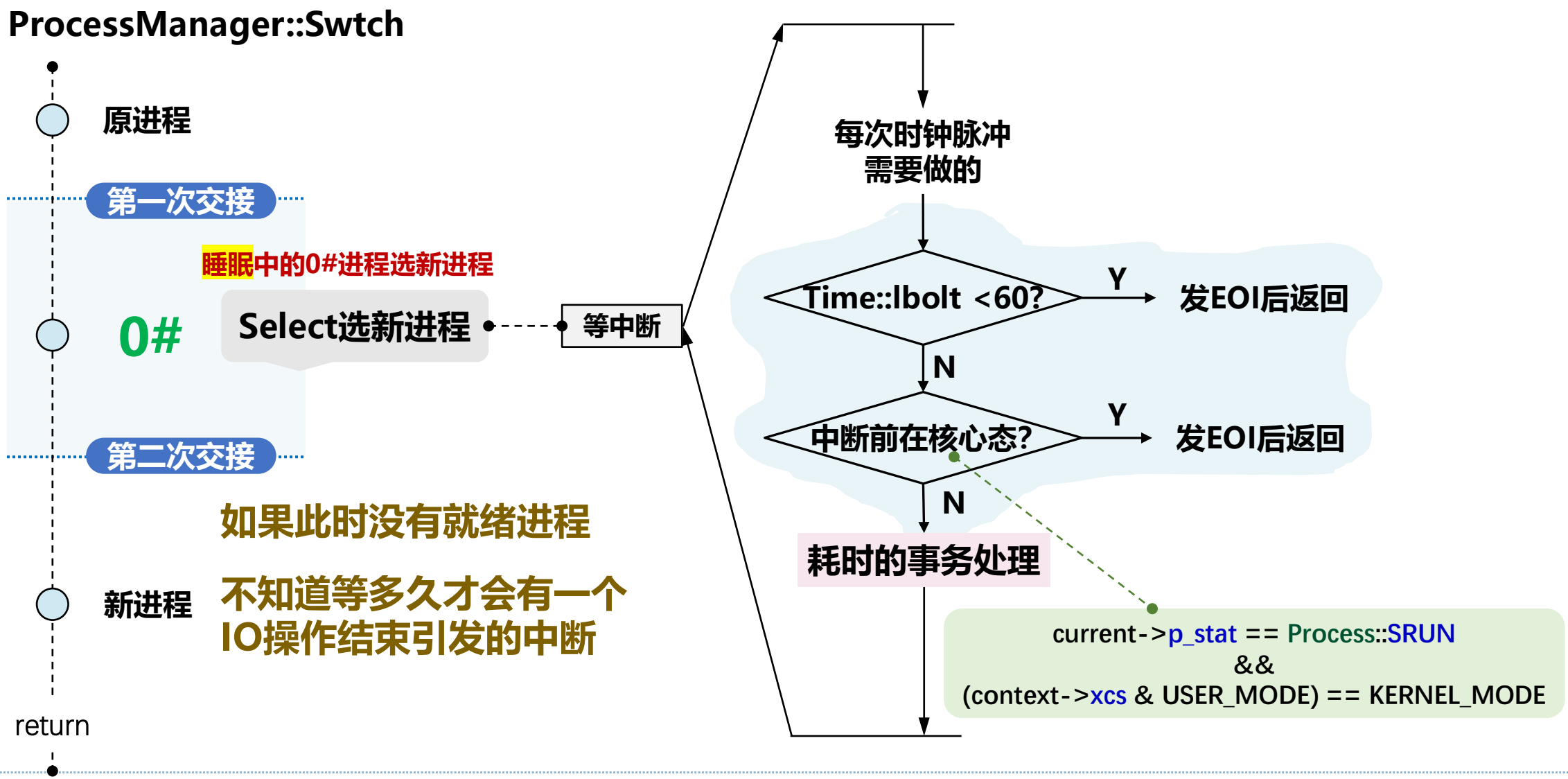
return

有没有什么中断是和之前有没有进程发起IO操作无关，一定会在确定时间内发生的?

关于 0# 进程

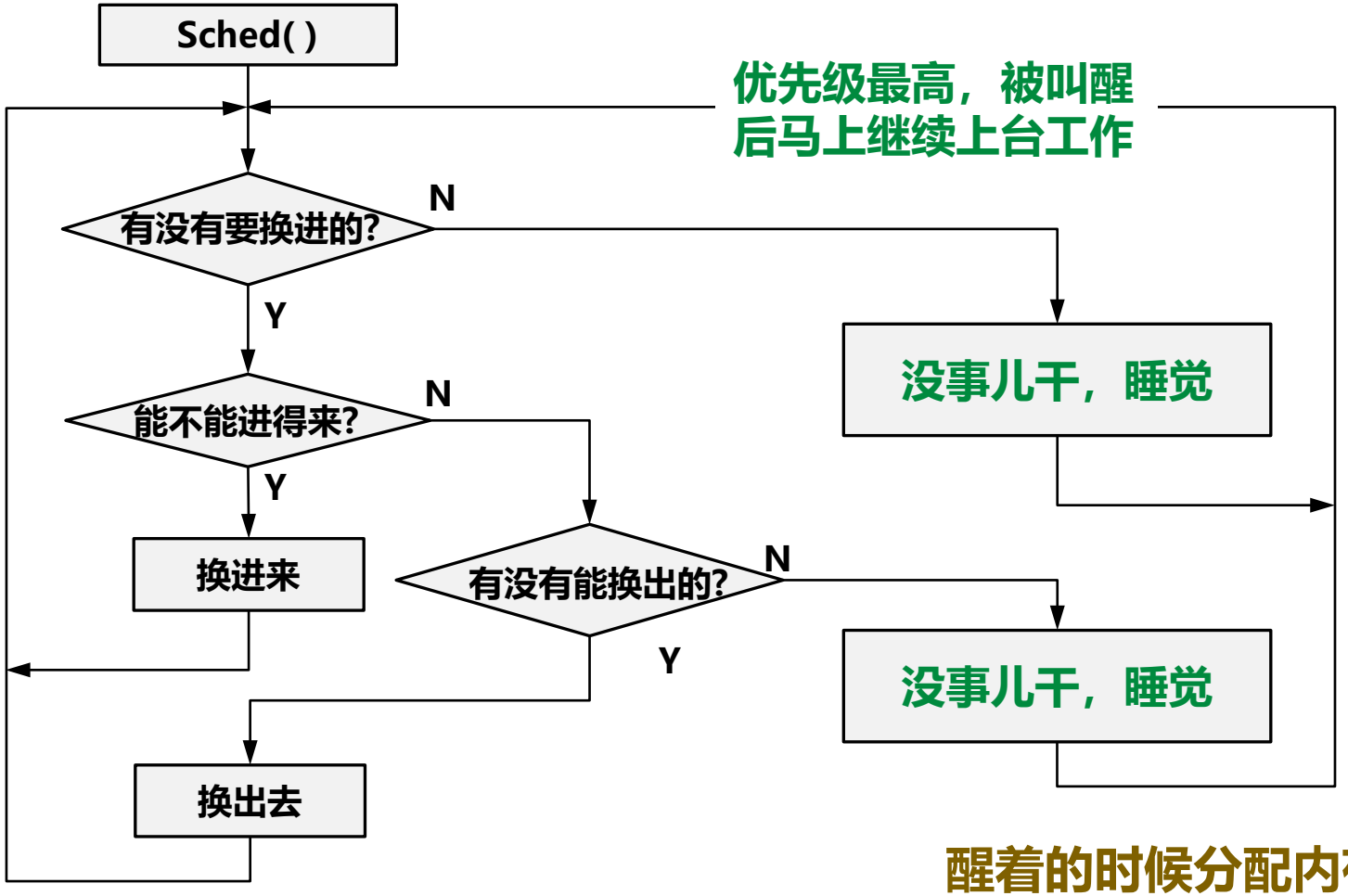

ProcessManager::Swtch
原进程
第一次交接
睡眠中的0#进程选新进程
0#
Select选新进程
等中断
第二次交接
如果此时没有就绪进程
新进程
不知道等多久才会有一个
IO操作结束引发的中断
return
每次时钟脉冲
需要做的
Time::lbolt <60?
Y
发EOI后返回
N
中断前在核心态?
Y
发EOI后返回
N
耗时的事务处理
(context->xcs & USER_MODE) == KERNEL_MODE
这时的 0# 会去执行耗时
的事务处理么?
加这样的限定条件的初衷是
怕耽误重要的核心态任务
可现在0#进程无事可做啊!

关于 0# 进程


ProcessManager::Swtch
原进程
第一次交接
睡眠中的0#进程选新进程
0#
Select选新进程
等中断
第二次交接
如果此时没有就绪进程
新进程
不知道等多久才会有一个
IO操作结束引发的中断
return
每次时钟脉冲
需要做的
Time::lbolt <60?
Y
发EOI后返回
N
中断前在核心态?
Y
发EOI后返回
N
耗时的事务处理
current->p_stat == Process::SRUN
&&
(context->xcs & USER_MODE) == KERNEL_MODE

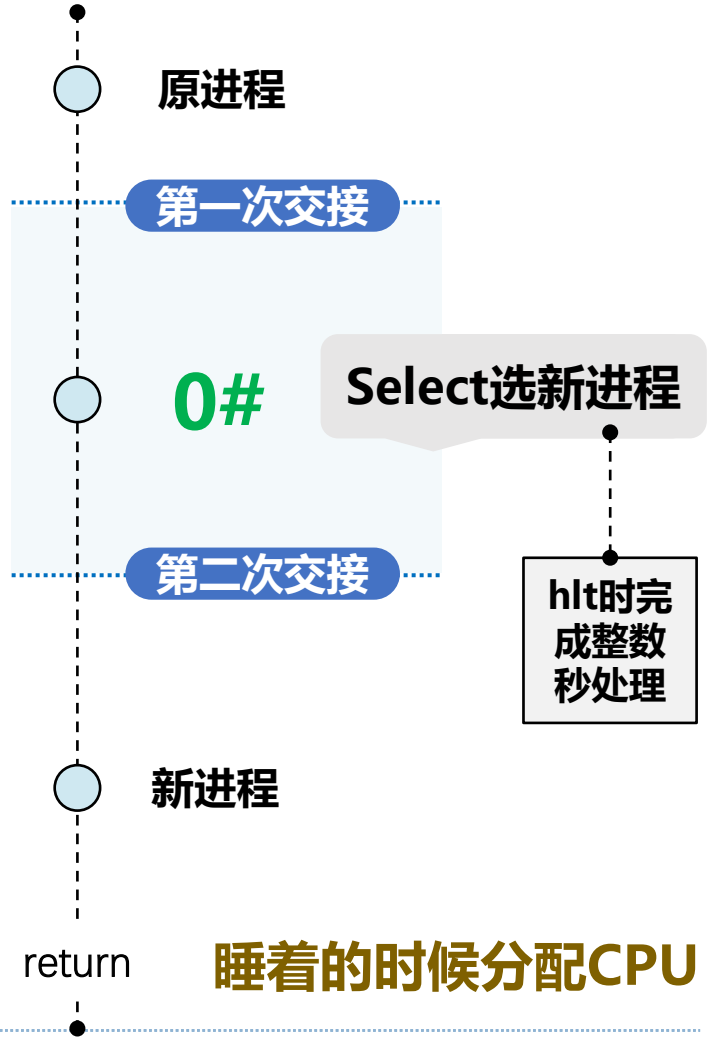
关于 0# 进程

0# 进程执行ProcessManager::Sched



醒着的时候分配内存

ProcessManager::Swtch



睡着的时候分配CPU

本节小结

1　UNIX进程图像的交换

2　0#进程

阅读教材：183页 ~ 186页

E14：进程管理（UNIX的进程图像交换）