第四章

进程控制

方　钰

主 要 内 容

4.1 UNIX时钟中断与系统调用

4.2 UNIX的进程调度状态

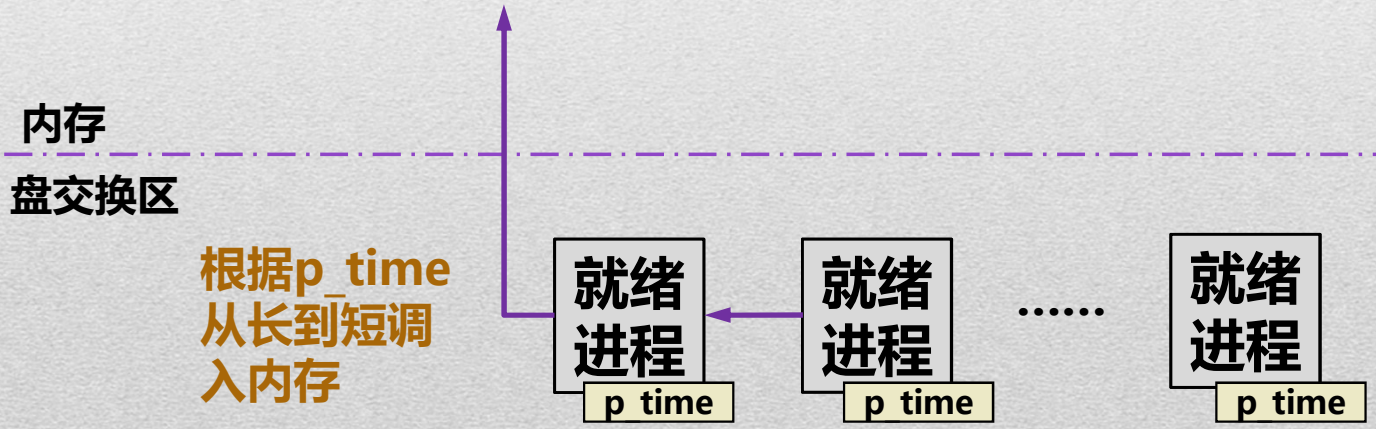
4.3 UNIX进程控制

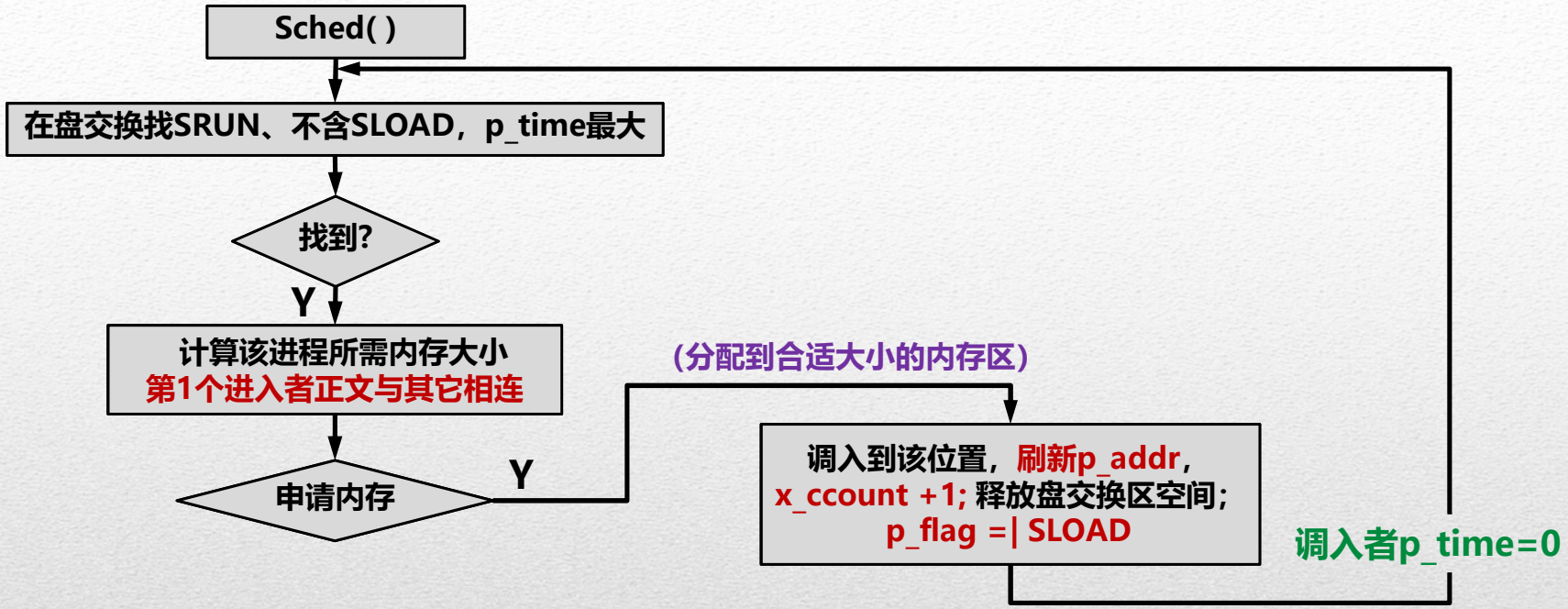
- 进程切换调度
- 进程创建与终止
- 进程图像交换

1 当内存空间不足以装下所有就绪进程时……

0#进程执行ProcessManager::Sched

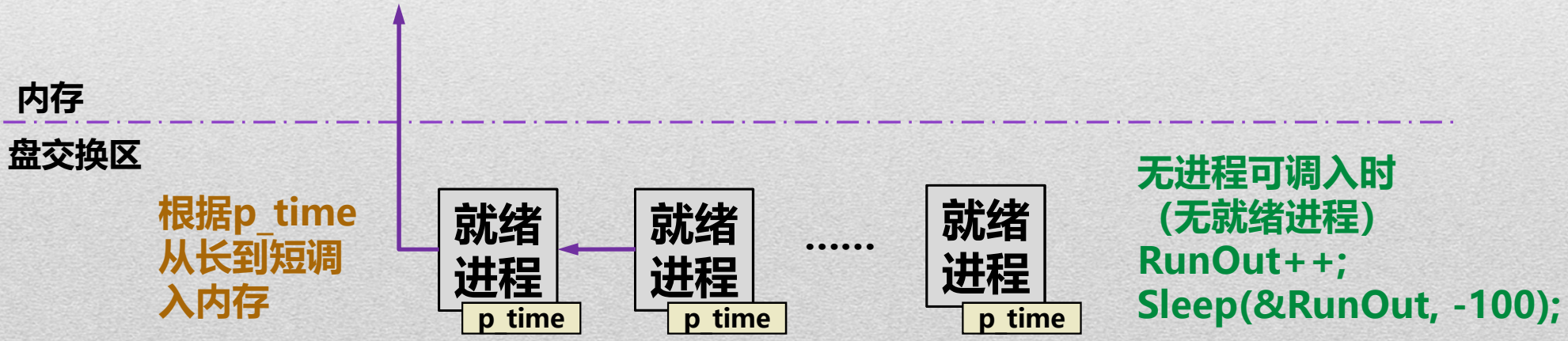
每秒时钟中断：对所有进程 p_time ++;

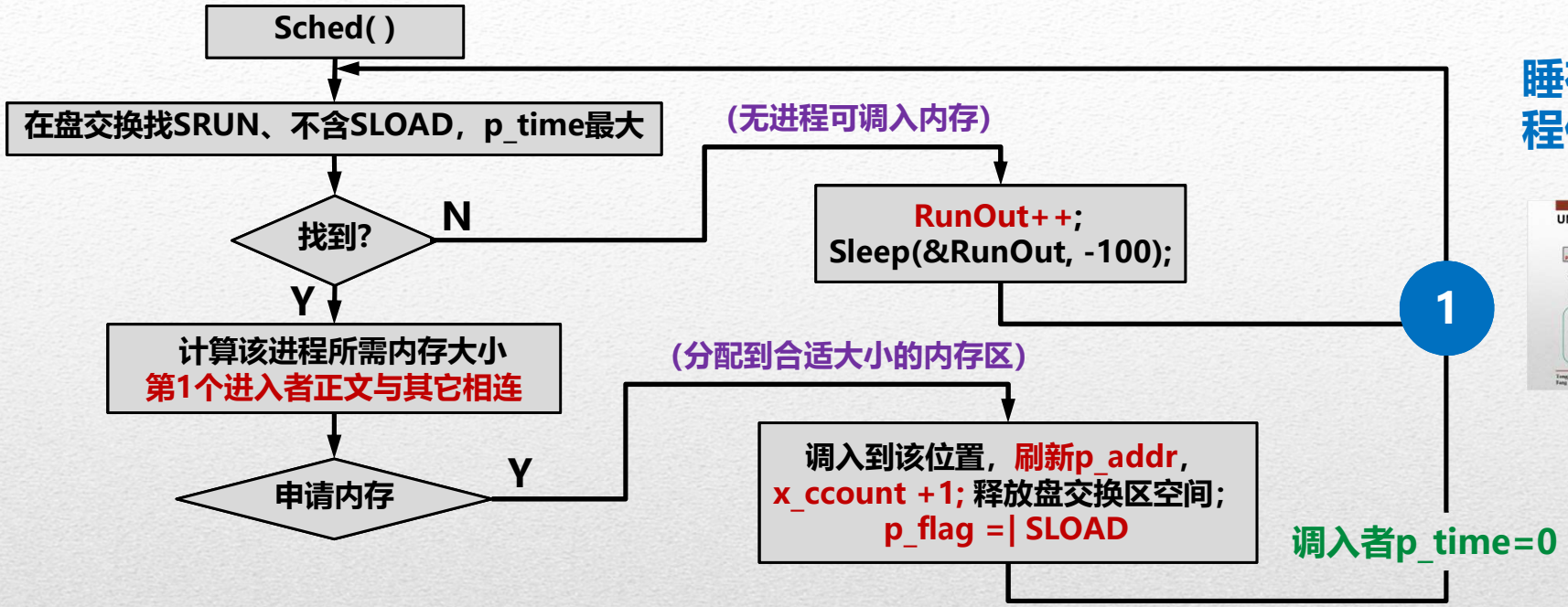



Sched( )
在盘交换找SRUN、不含SLOAD，p_time最大
找到?
Y
计算该进程所需内存大小
第1个进入者正文与其它相连
申请内存
Y
(分配到合适大小的内存区)
调入到该位置，刷新p_addr，
x_ccount +1; 释放盘交换区空间;
p_flag =| SLOAD
调入者p_time=0

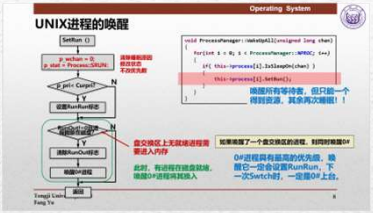
0#进程执行ProcessManager::Sched

每秒时钟中断：对所有进程 p_time ++;

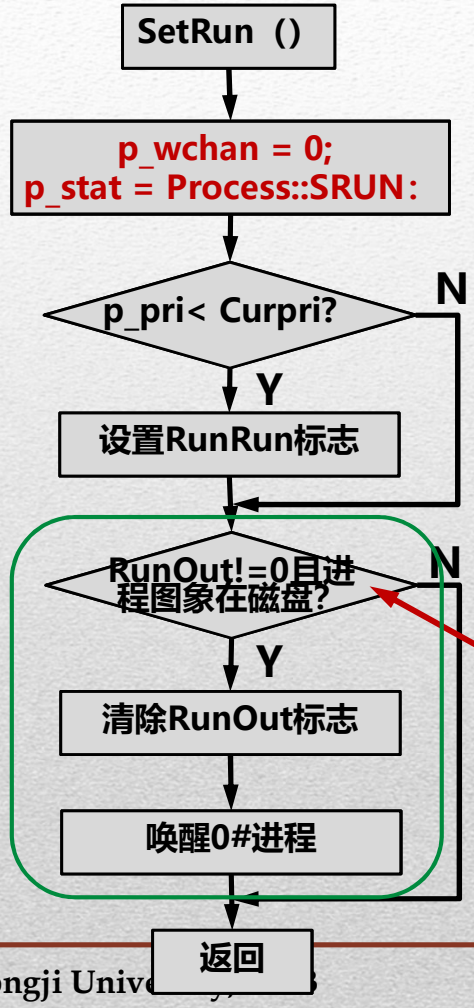




睡在这里的0#进程什么时候醒过来?



UNIX进程的唤醒



清除睡眠原因
修改状态
不改优先数

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

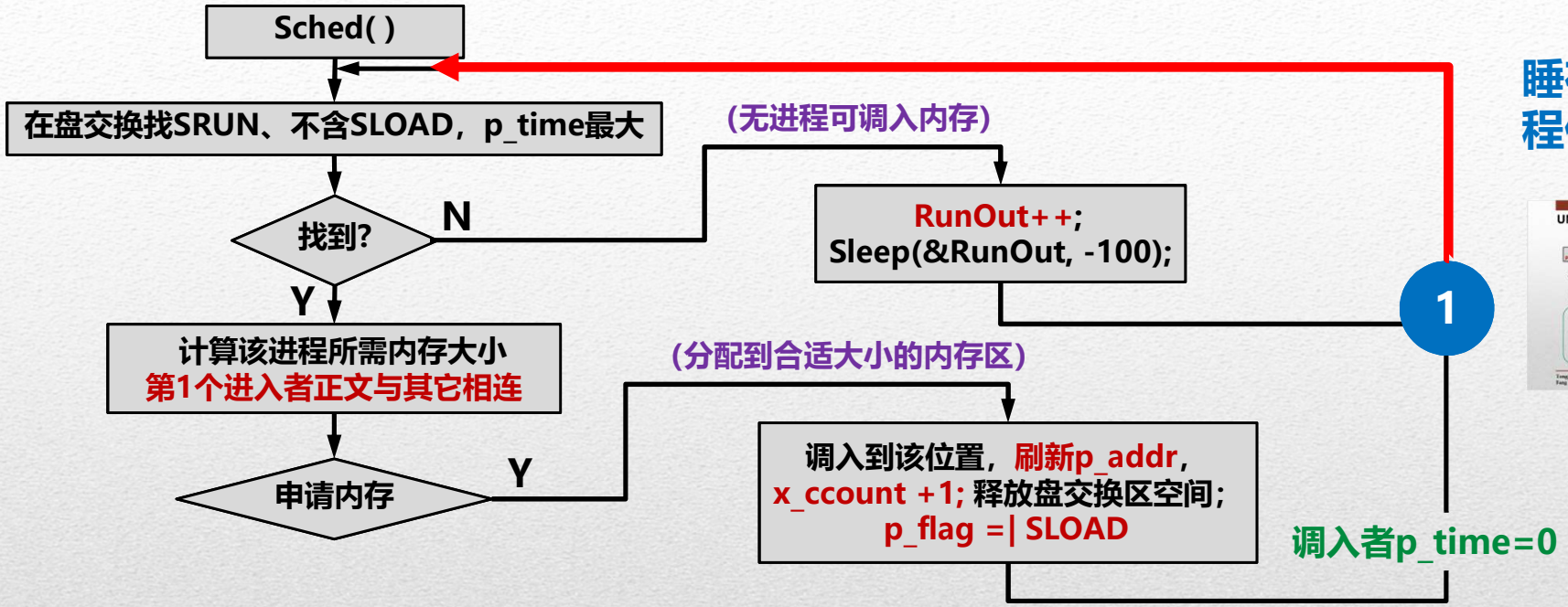
唤醒所有等待者，但只能一个得到资源，其余再次睡眠！！

盘交换区上无就绪进程需要进入内存

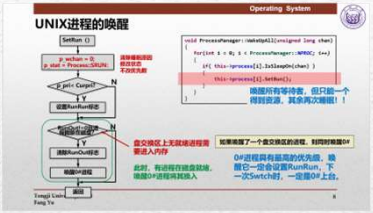
此时，有进程在磁盘就绪，唤醒0#进程将其换入

如果唤醒了一个盘交换区的进程，则同时唤醒0#

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。



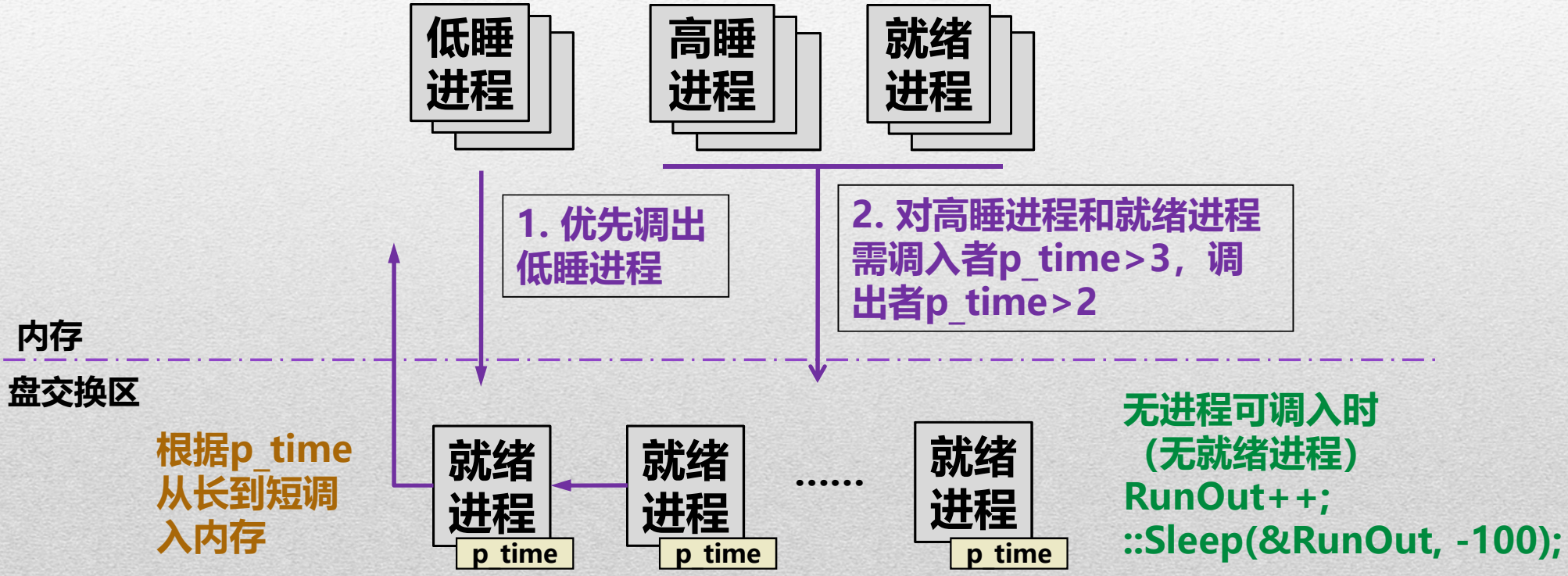
睡在这里的0#进程什么时候醒过来?

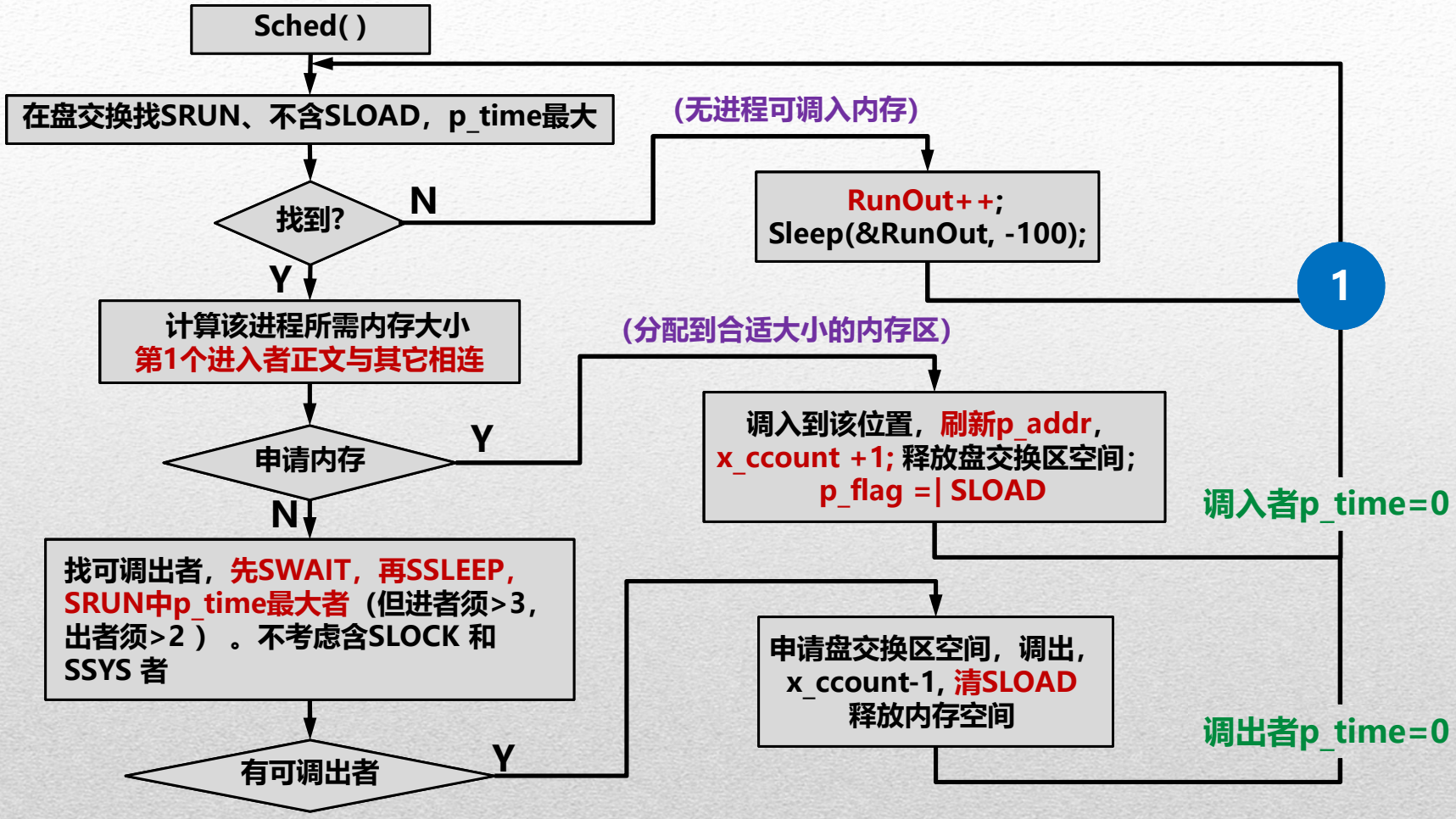


0#进程执行ProcessManager::Sched

有就绪进程在交换区，但无内存空闲区时，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 ++;

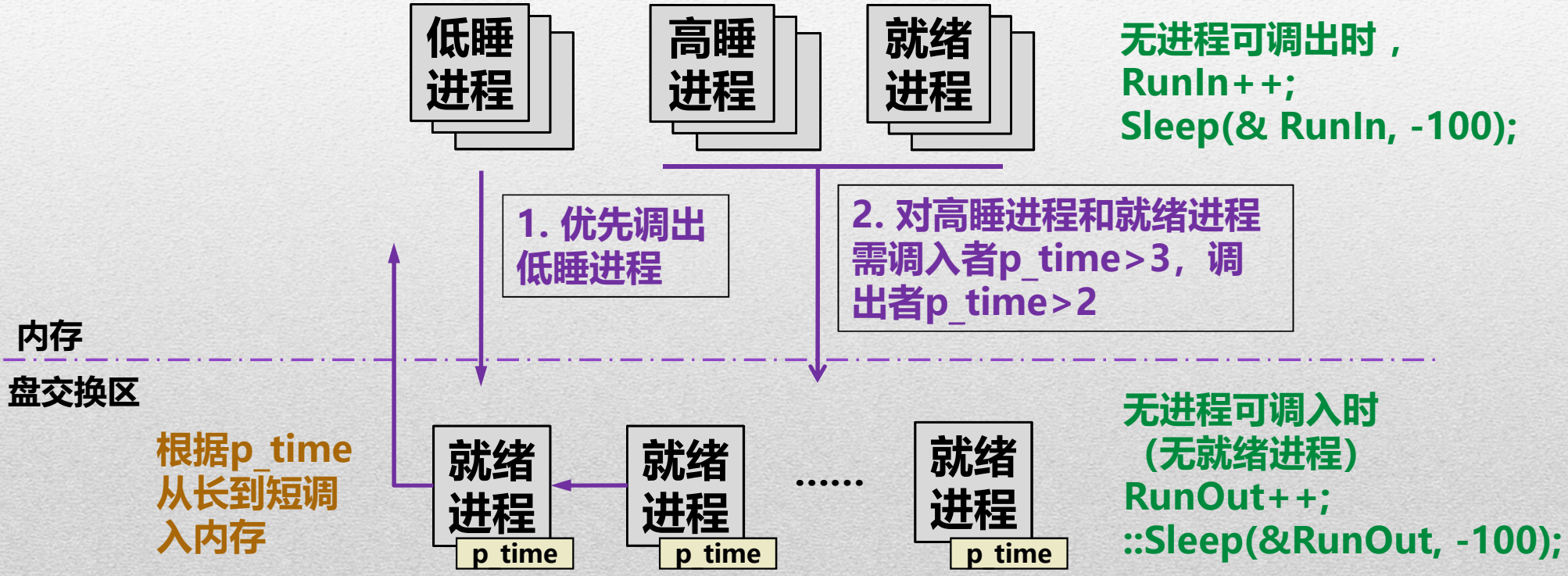


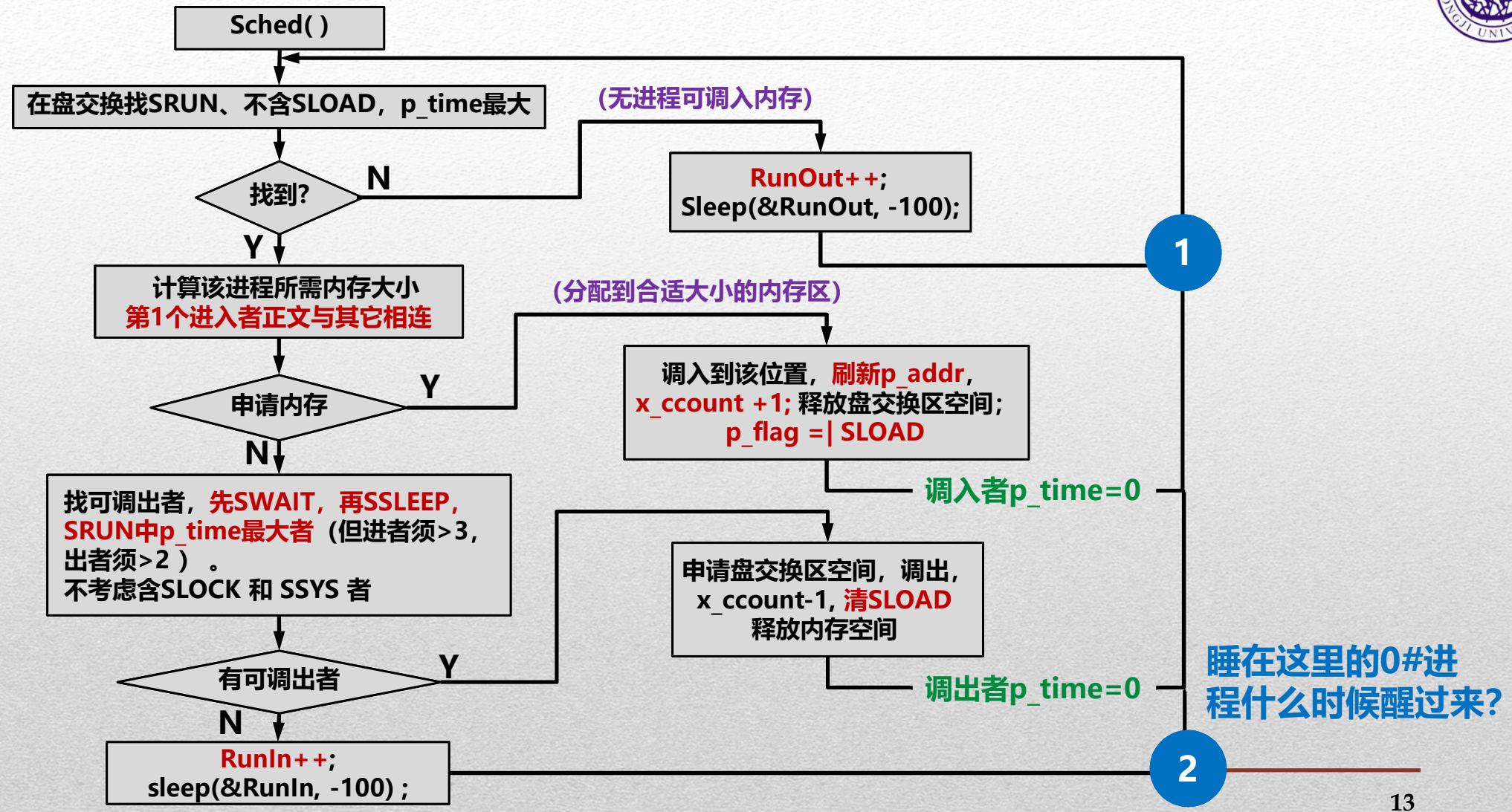

Sched( )
在盘交换找SRUN、不含SLOAD，p_time最大
找到?
N
Y
(无进程可调入内存)
RunOut++;
Sleep(&RunOut, -100);
1
计算该进程所需内存大小
第1个进入者正文与其它相连
申请内存
Y
N
(分配到合适大小的内存区)
调入到该位置，刷新p_addr，
x_ccount +1; 释放盘交换区空间;
p_flag =| SLOAD
调入者p_time=0
找可调出者，先SWAIT，再SSLEEP，
SRUN中p_time最大者（但进者须>3，
出者须>2 ） 。不考虑含SLOCK 和
SSYS 者
有可调出者
Y
申请盘交换区空间，调出，
x_ccount-1, 清SLOAD
释放内存空间
调出者p_time=0

0#进程执行ProcessManager::Sched

有就绪进程在交换区，但无内存空闲区时，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 ++;



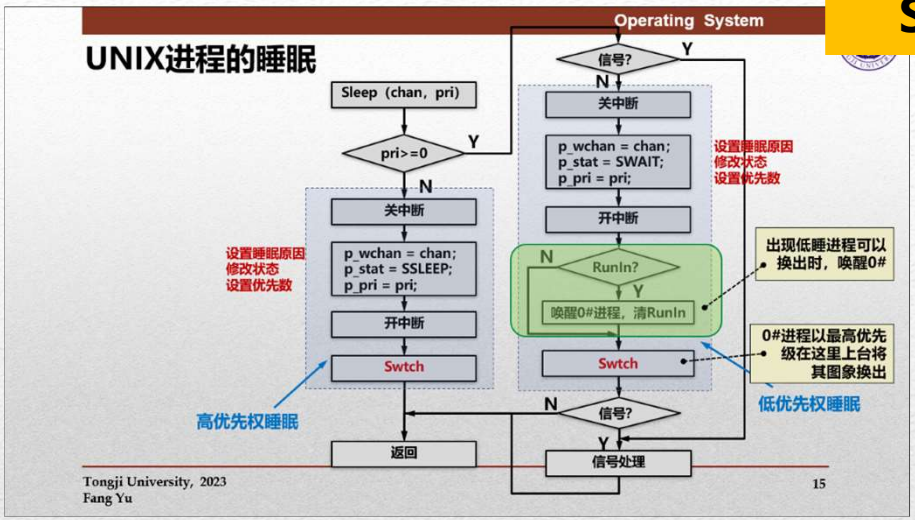

Sched( )
在盘交换找SRUN、不含SLOAD，p_time最大
找到?
N
Y
(无进程可调入内存)
RunOut++;
Sleep(&RunOut, -100);
1
计算该进程所需内存大小
第1个进入者正文与其它相连
申请内存
Y
N
(分配到合适大小的内存区)
调入到该位置，刷新p_addr，
x_ccount +1; 释放盘交换区空间;
p_flag =| SLOAD
调入者p_time=0
找可调出者，先SWAIT，再SSLEEP，
SRUN中p_time最大者（但进者须>3，
出者须>2 ） 。
不考虑含SLOCK 和 SSYS 者
有可调出者
Y
N
申请盘交换区空间，调出，
x_ccount-1, 清SLOAD
释放内存空间
调出者p_time=0
睡在这里的0#进
程什么时候醒过来?
RunIn++;
sleep(&RunIn, -100) ;
2

Sleep

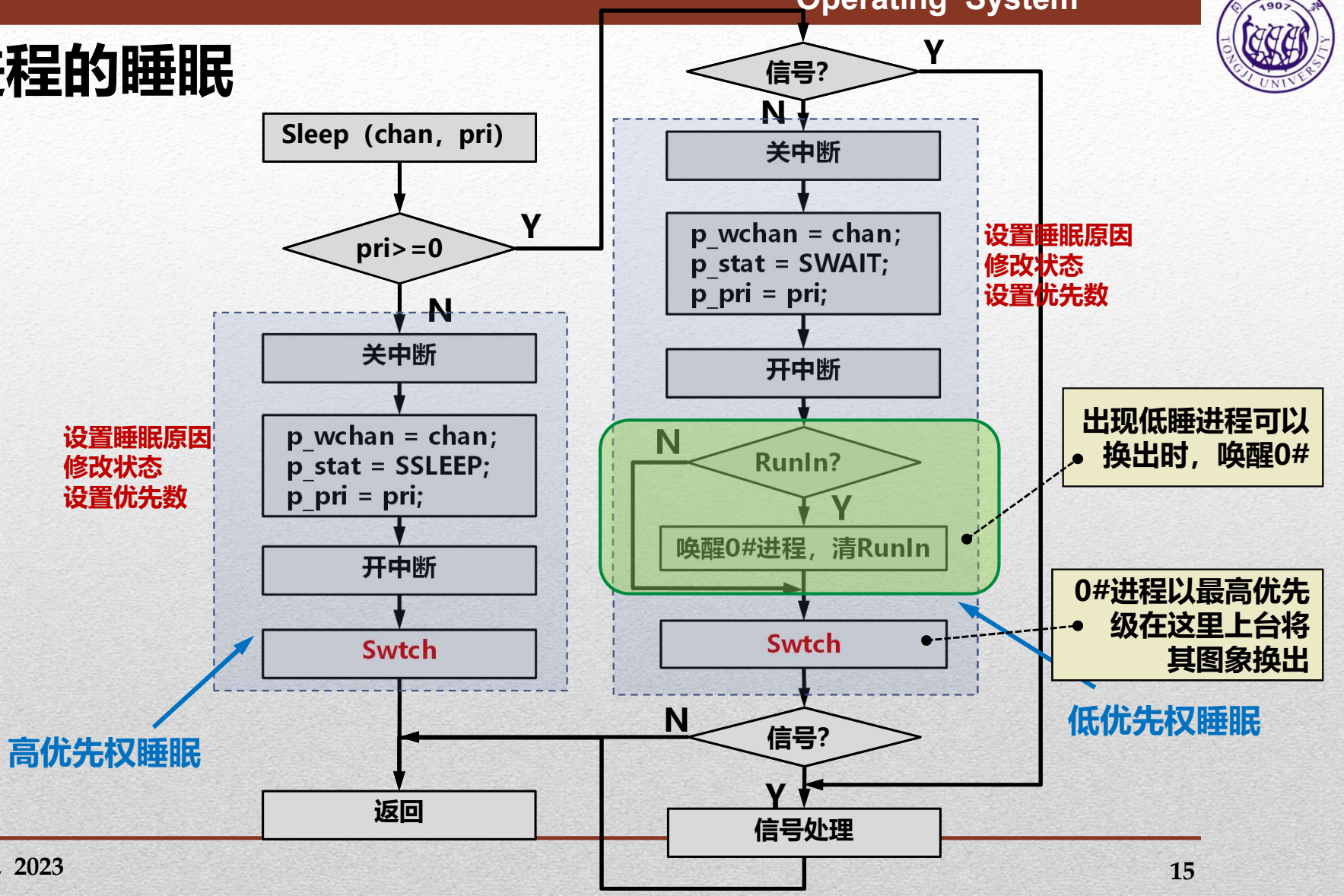
2 睡在这里的0#进程什么时候醒过来?

低睡进程

1. 没有低睡进程



UNIX进程的睡眠


Sleep（chan，pri）
pri>=0
Y
N
关中断
p_wchan = chan;
p_stat = SSLEEP;
p_pri = pri;
开中断
Swtch
返回
设置睡眠原因
修改状态
设置优先数
高优先权睡眠
信号?
Y
N
关中断
p_wchan = chan;
p_stat = SWAIT;
p_pri = pri;
开中断
RunIn?
N
Y
唤醒0#进程，清RunIn
Swtch
信号?
N
Y
信号处理
设置睡眠原因
修改状态
设置优先数
出现低睡进程可以换出时，唤醒0#
0#进程以最高优先级在这里上台将其图象换出
低优先权睡眠

2 睡在这里的0#进程什么时候醒过来?

低睡进程

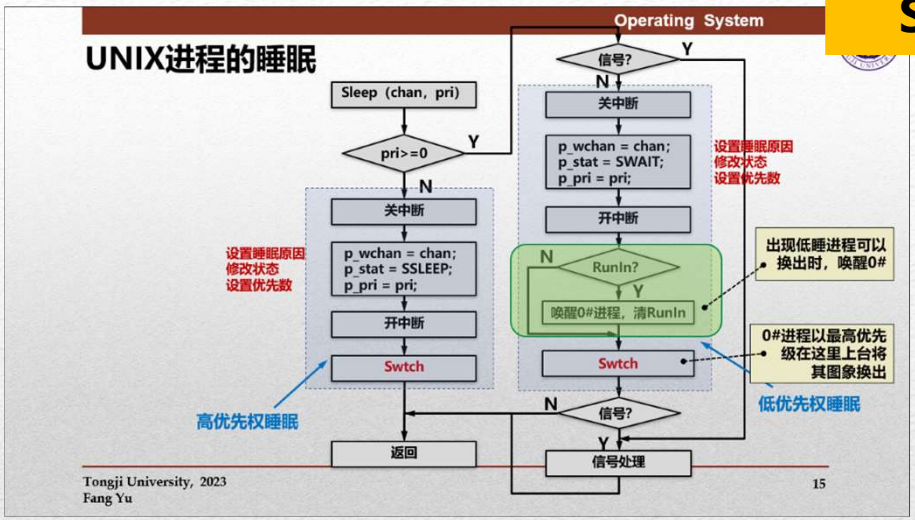
1. 没有低睡进程

高睡进程　就绪进程

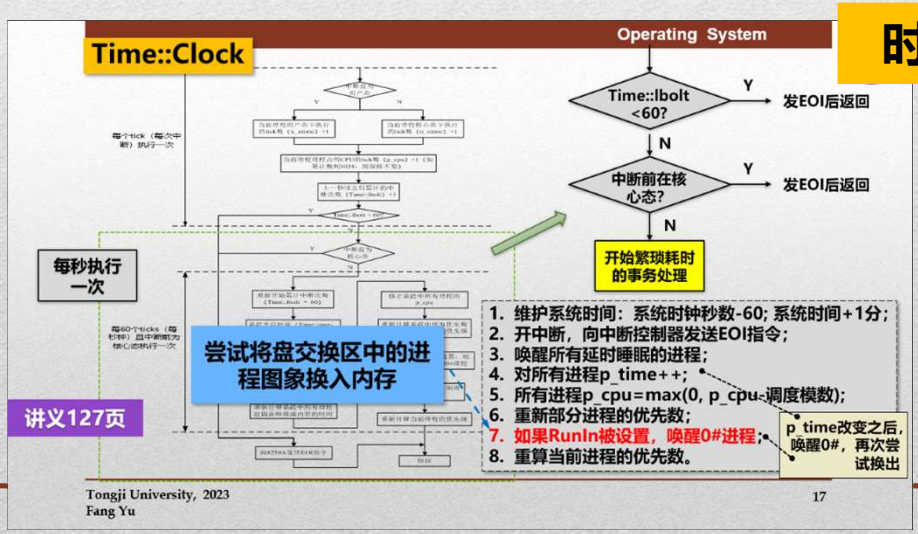
2. 对高睡进程和就绪进程需调入者p_time>3，调出者p_time>2

2. 时间不够

Sleep



时钟中断

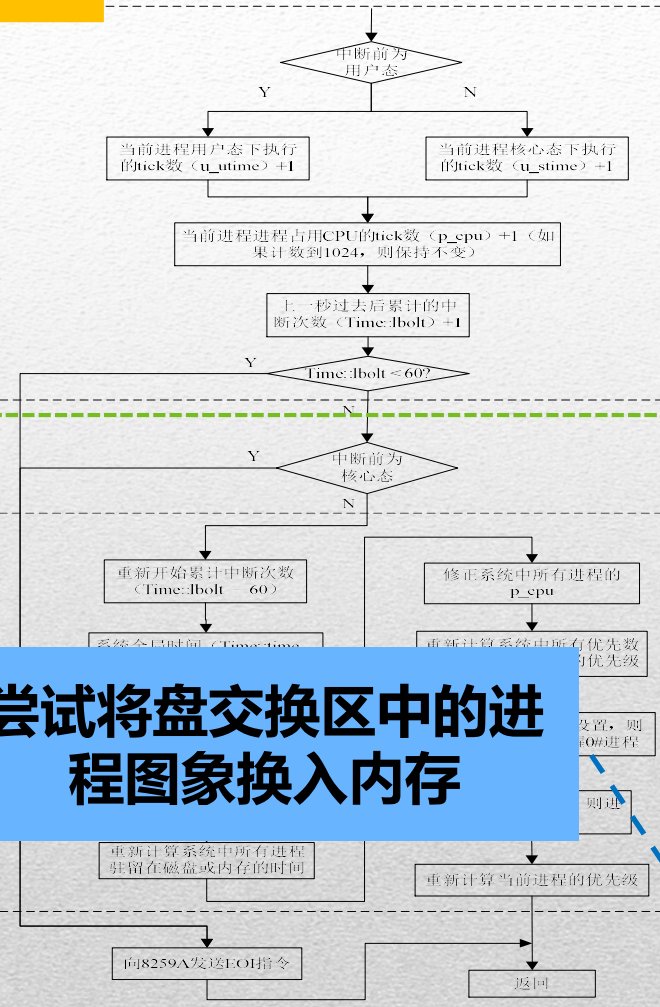


Time::Clock

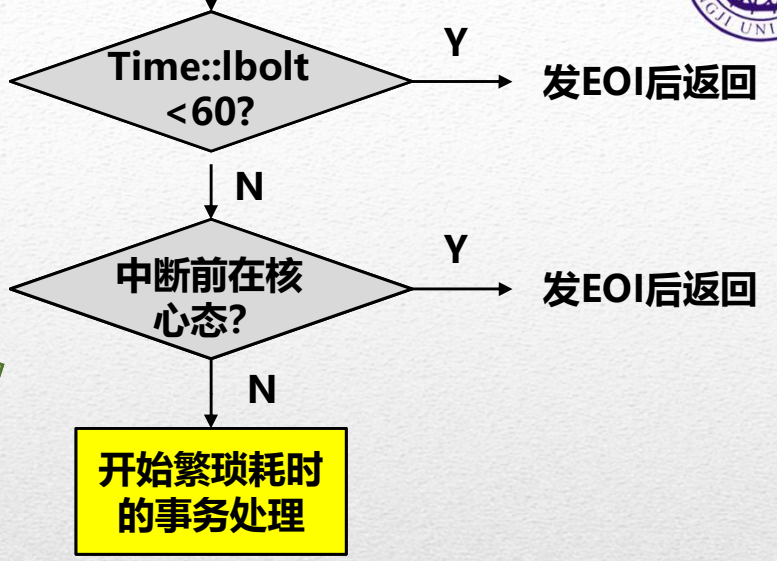
每个tick（每次中断）执行一次

每秒执行一次

每60个ticks（每秒钟）且中断前为核心态执行一次



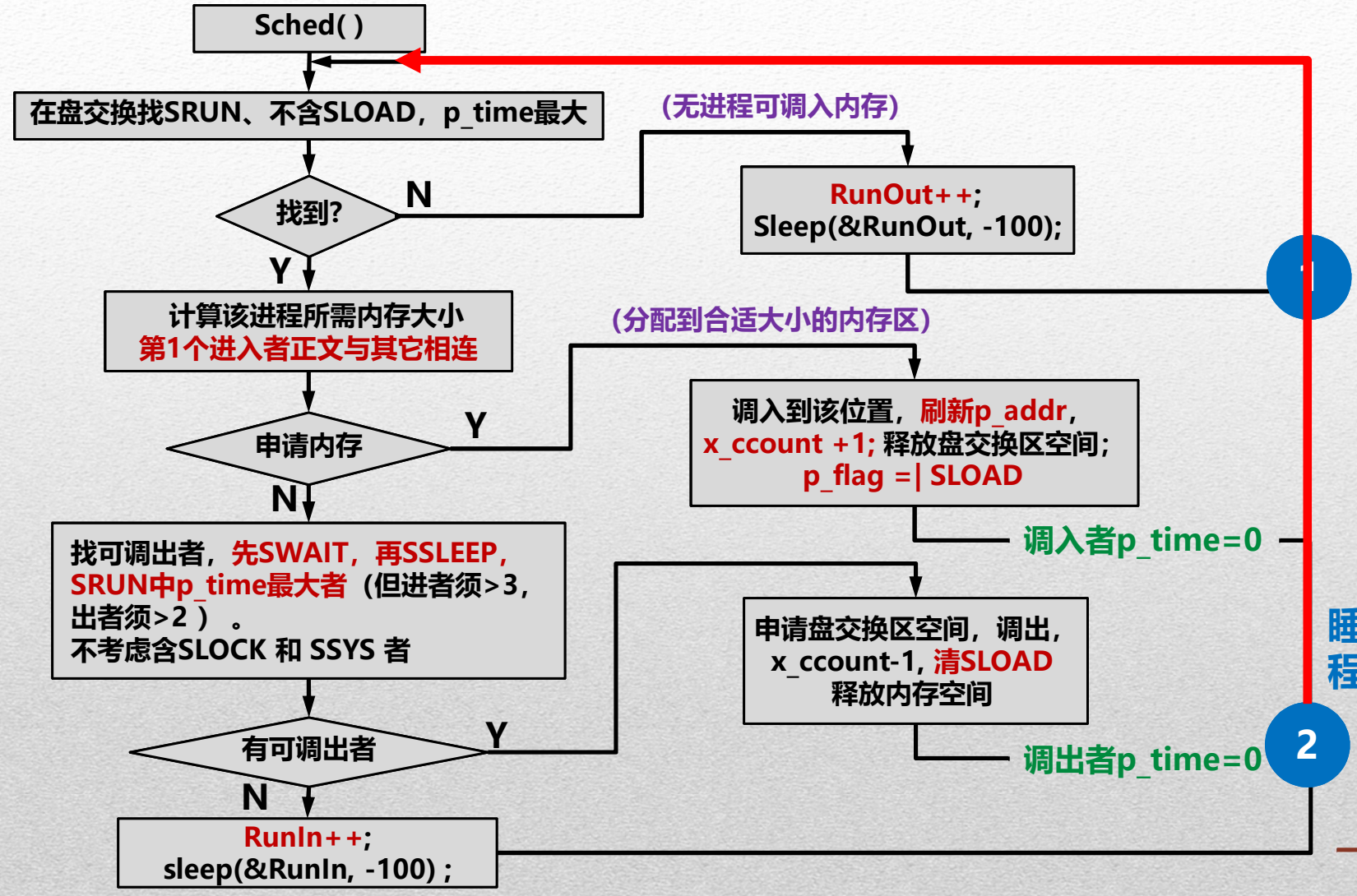
尝试将盘交换区中的进程图象换入内存



1. 维护系统时间：系统时钟秒数-60; 系统时间+1分;
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-调度模数);
6. 重新部分进程的优先数;
7. 如果RunIn被设置，唤醒0#进程;
8. 重算当前进程的优先数。

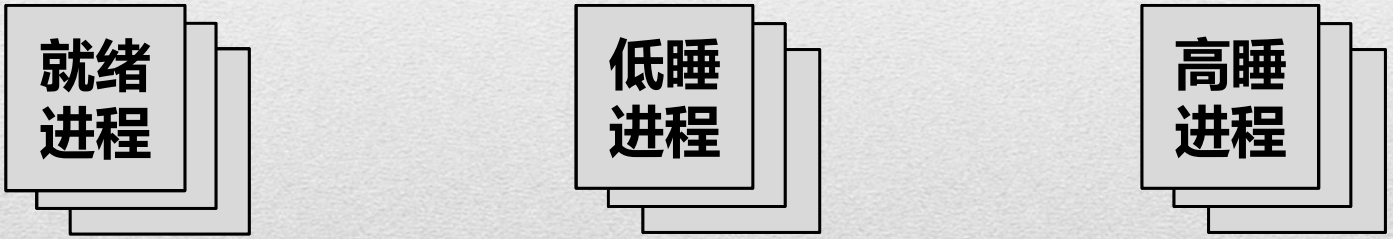
p_time改变之后，唤醒0#，再次尝试换出

讲义127页

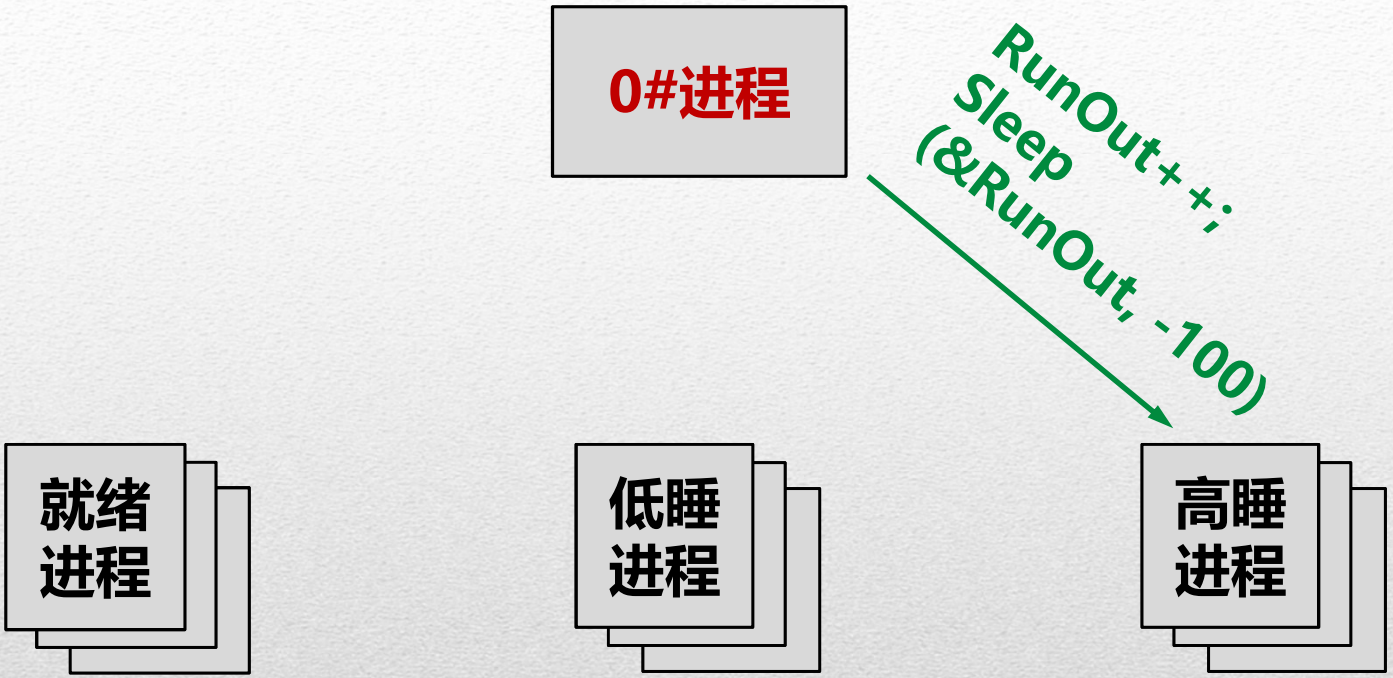


睡在这里的0#进程什么时候醒过来?

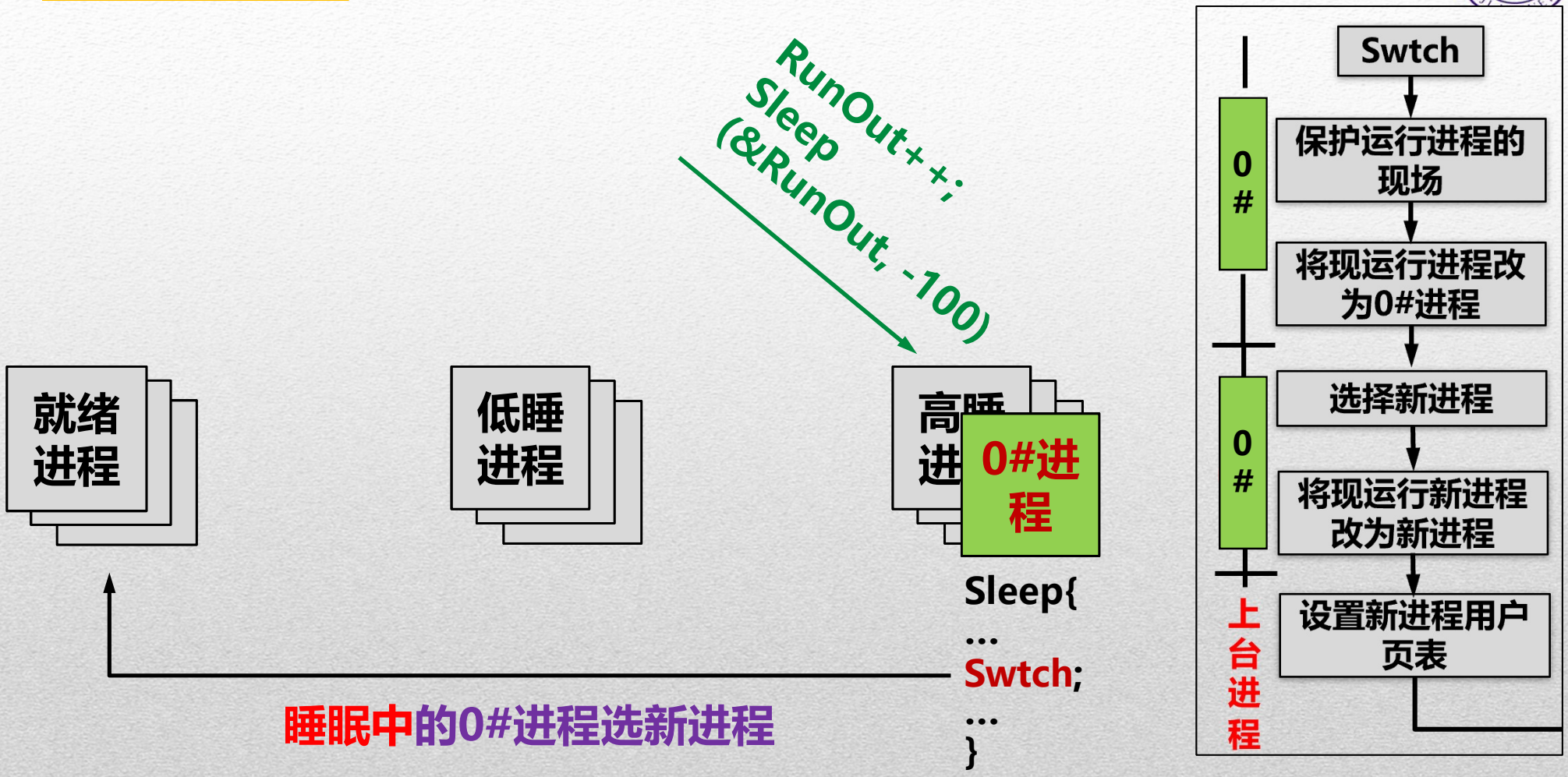
关于0#进程



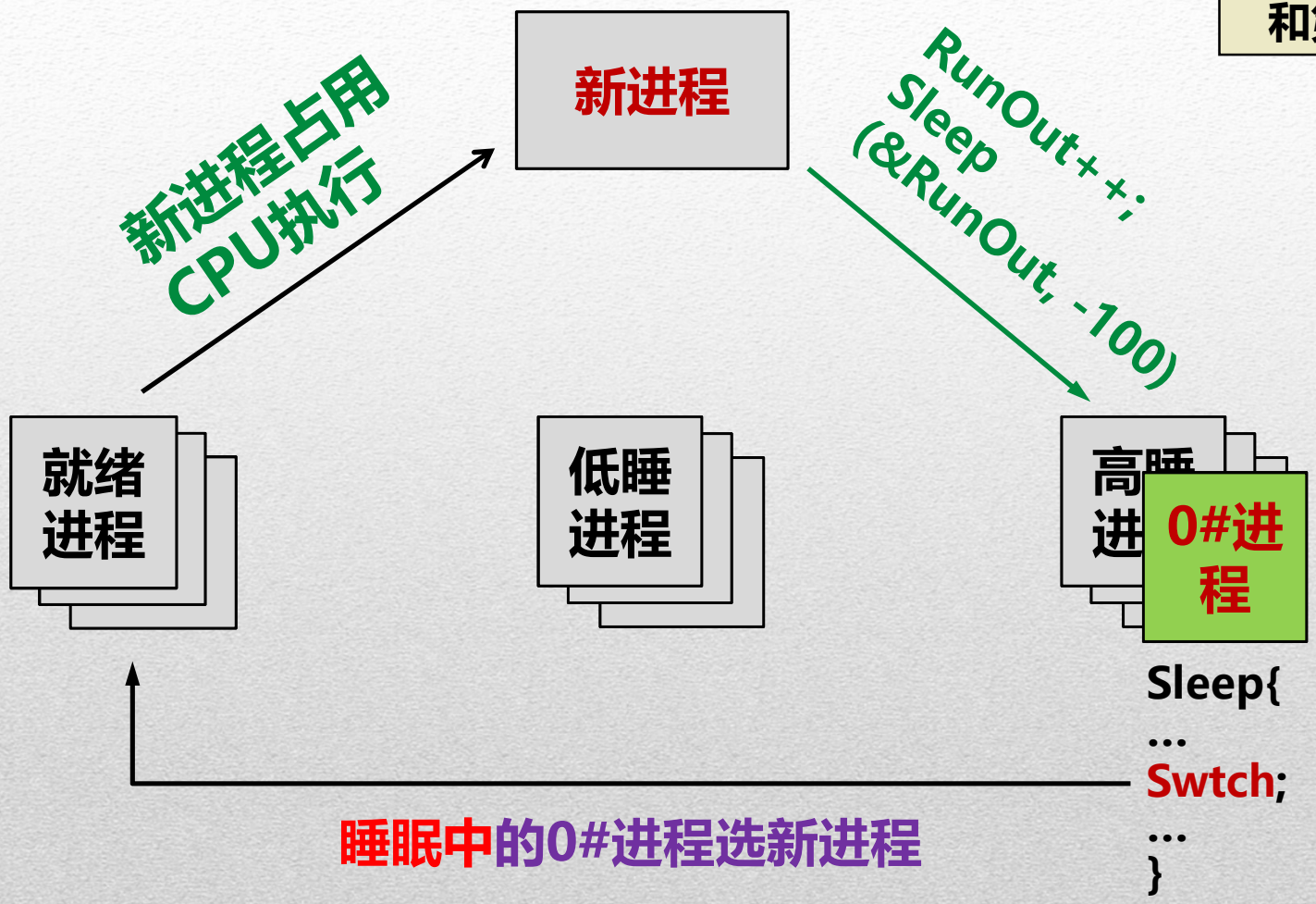
关于0#进程

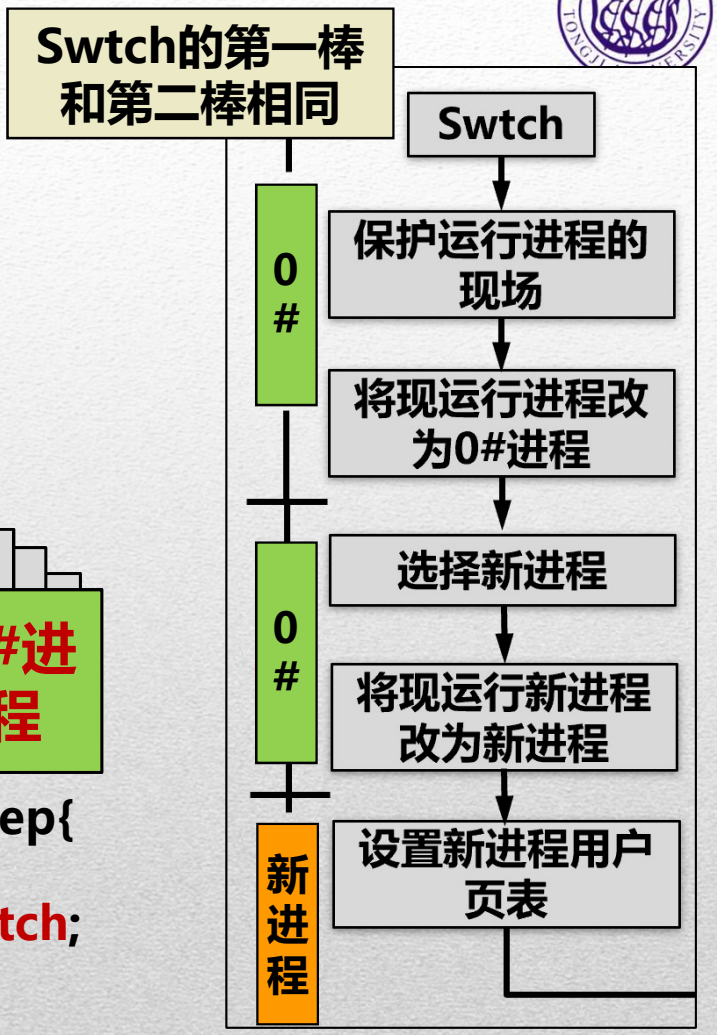


关于0#进程

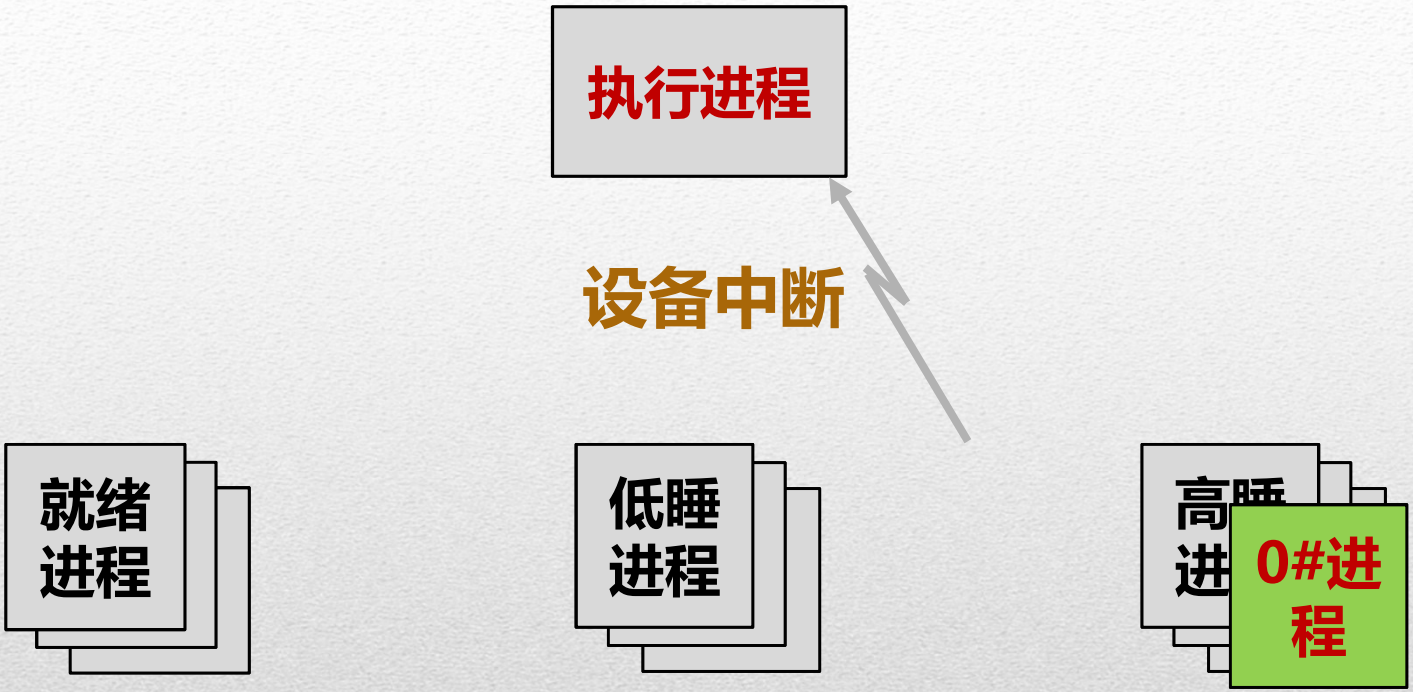

RunOut++;
Sleep
(&RunOut, -100)
就绪进程
低睡进程
高睡进
0#进程
Sleep{
…
Swtch;
…
}
睡眠中的0#进程选新进程
Swtch
0#
保护运行进程的现场
将现运行进程改为0#进程
选择新进程
0#
将现运行新进程改为新进程
设置新进程用户页表
上台进程

关于0#进程


新进程
新进程占用CPU执行
RunOut++;
Sleep(&RunOut, -100)
就绪进程
低睡进程
高睡进
0#进程
Sleep{
…
Swtch;
…
}
睡眠中的0#进程选新进程


Swtch的第一棒和第二棒相同
Swtch
0#
保护运行进程的现场
将现运行进程改为0#进程
选择新进程
0#
将现运行新进程改为新进程
新进程
设置新进程用户页表

关于0#进程



中断中如果唤醒一个磁盘进程，0#进程被唤醒；RunRun被设置

关于0#进程

执行进程

设备中断

就绪进程

0#进程

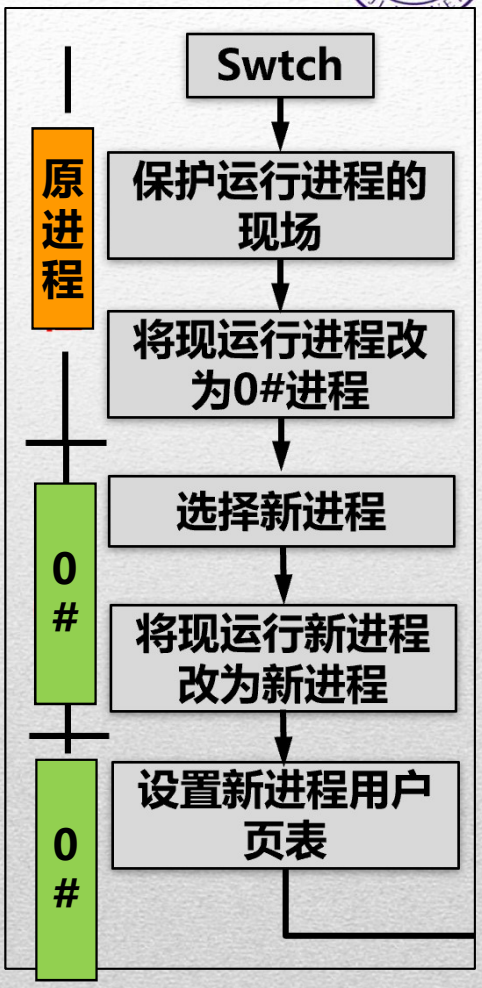
低睡进程

高睡进程

中断返回用户态前，*Swtch*

*此时0#就绪

中断中如果唤醒一个磁盘进程，0#进程被唤醒；RunRun被设置



关于0#进程

0#进程

0#进程占用CPU执行

就绪进程

0#进程

低睡进程

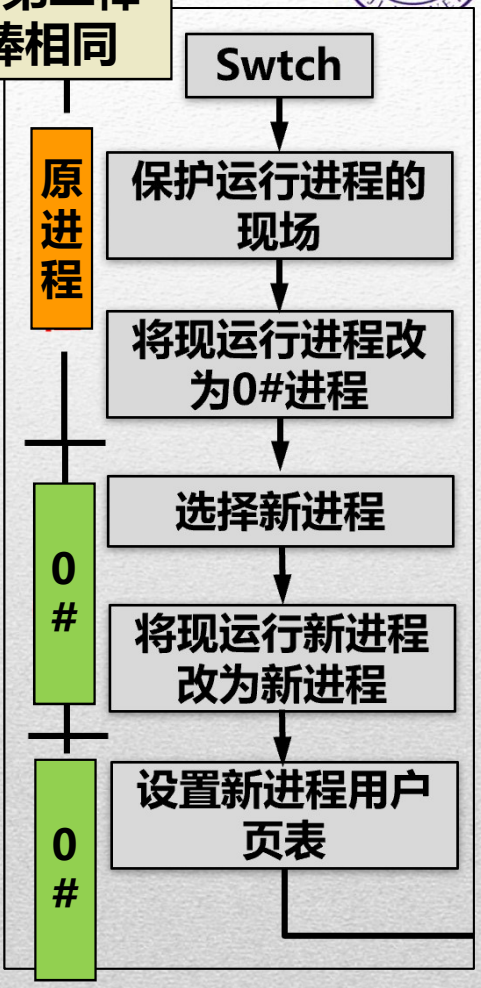
高睡进程

中断返回用户态前，*swtch*

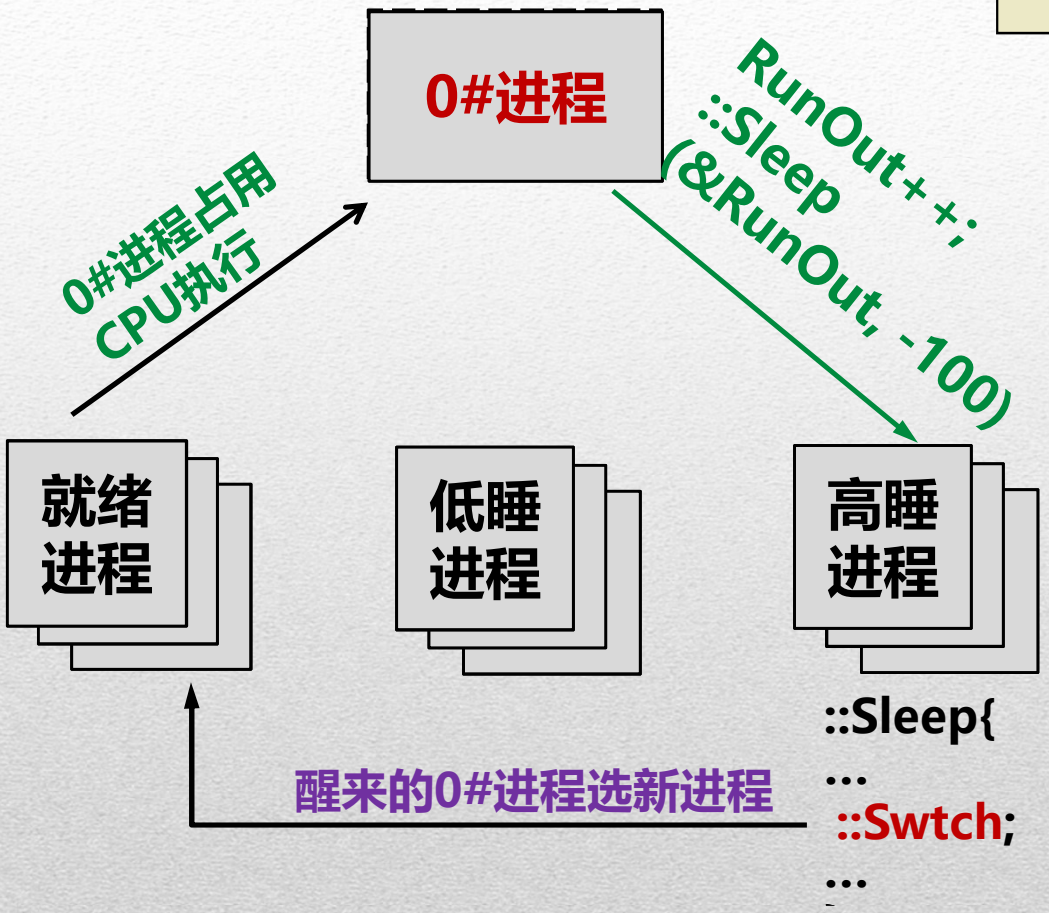
*此时0#清醒

中断中如果唤醒一个磁盘进程，0#进程被唤醒；RunRun被设置

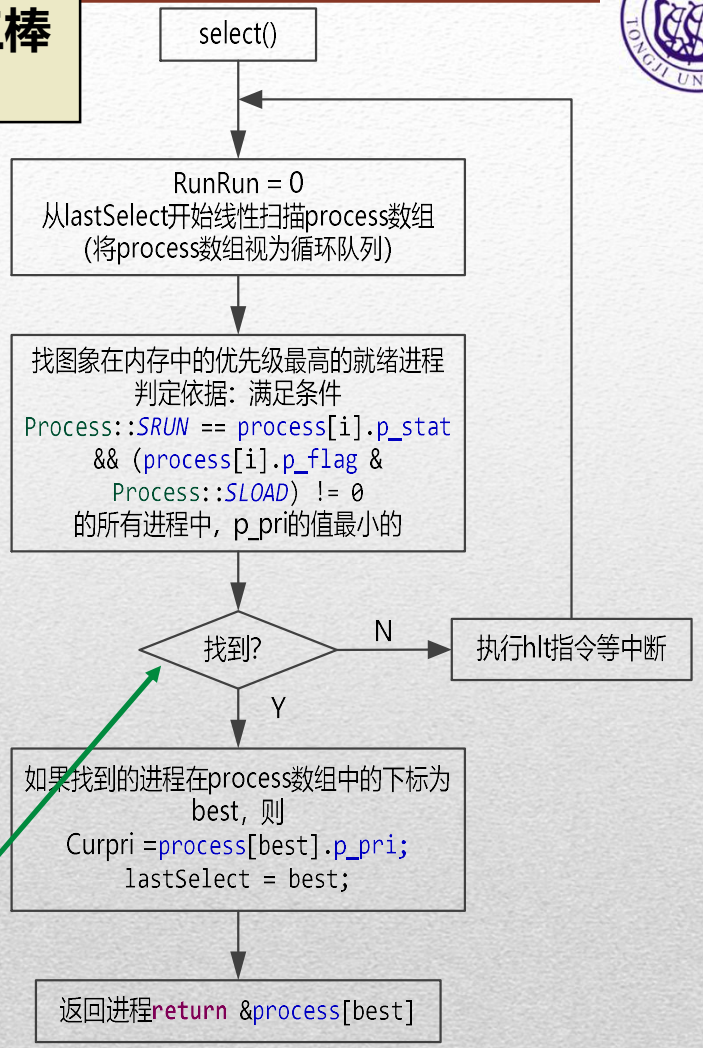
Swtch的第二棒和第三棒相同



关于0#进程

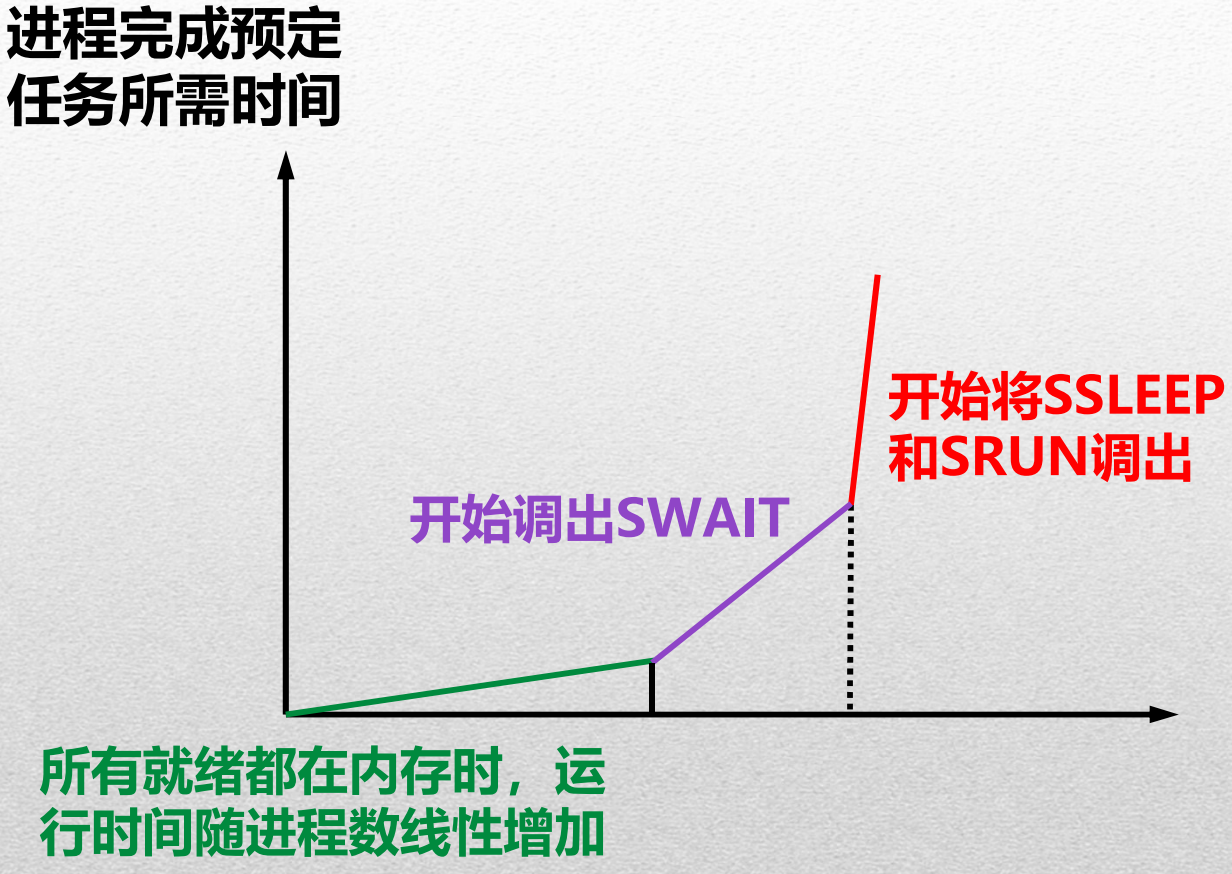


Swtch中的三棒都相同



未找到合适进程，等待中断；中断中可能由于某些事件的发生执行::WakeUpAll；WakeUpAll中可能唤醒0#。

关于内存颠簸

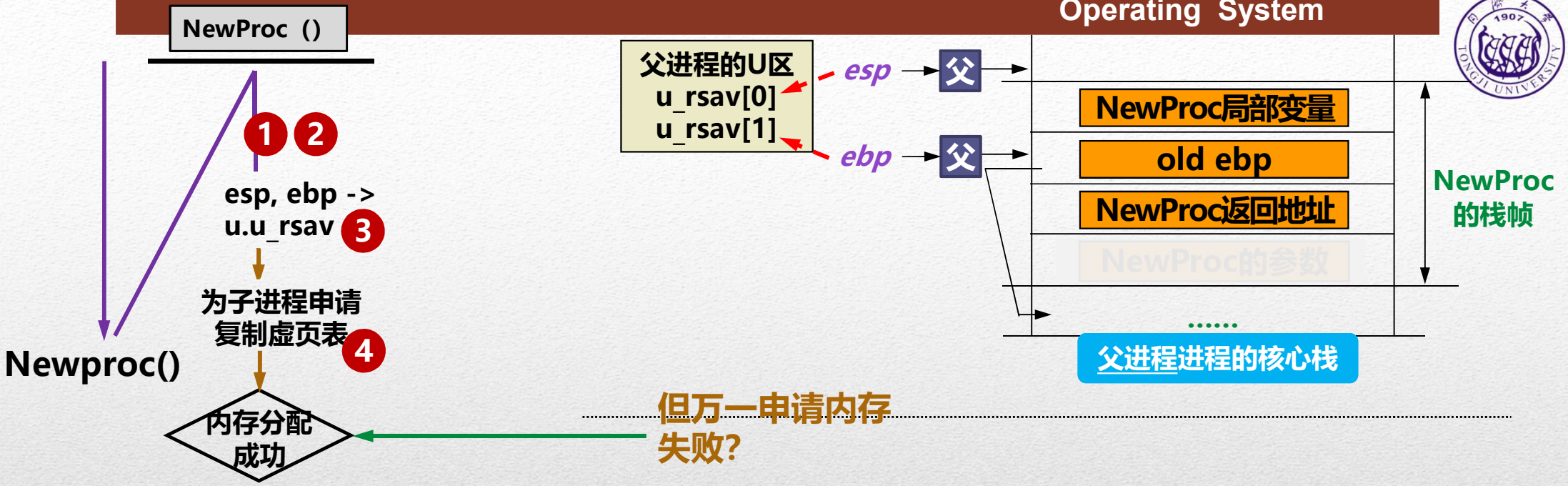


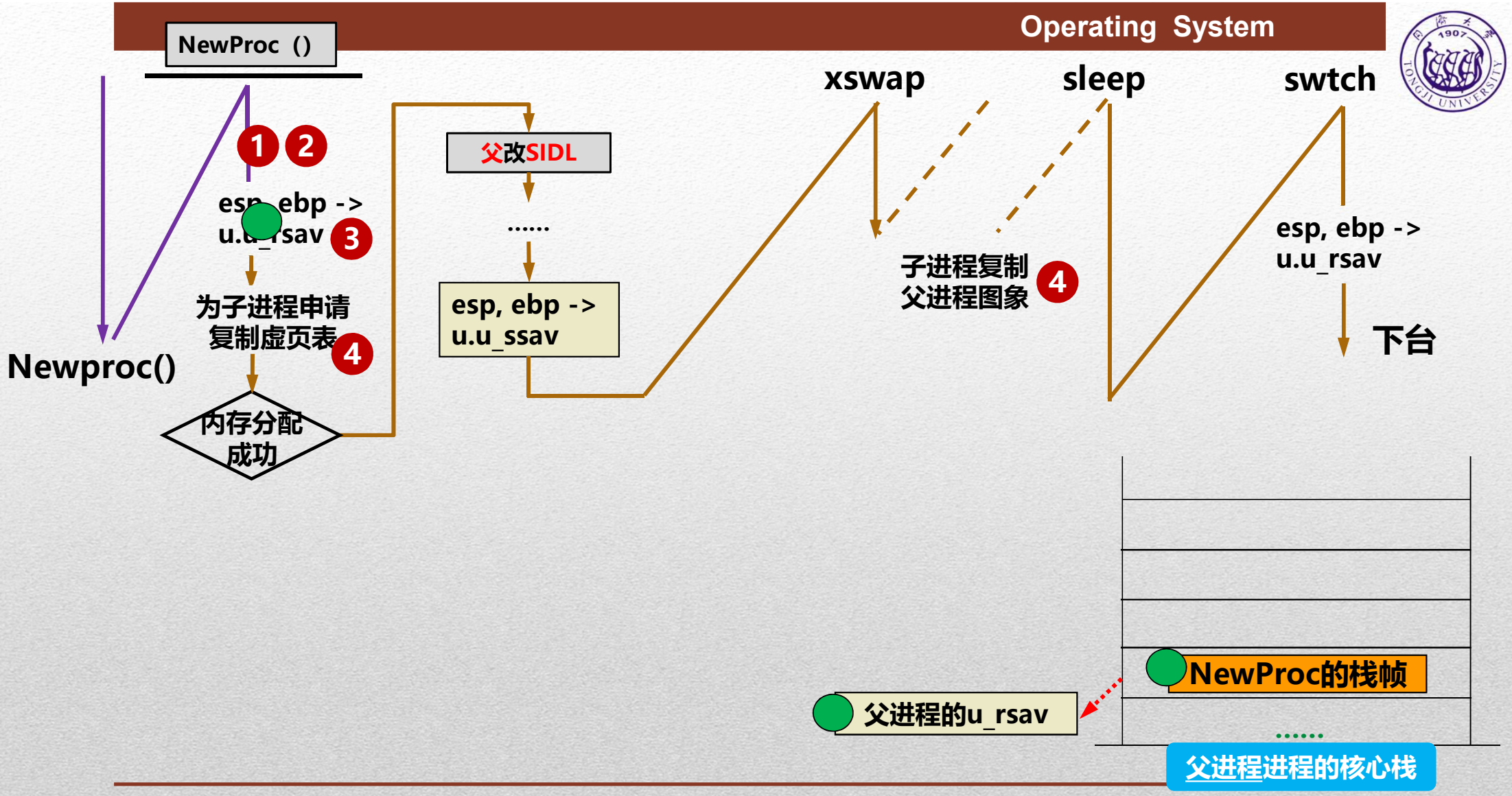
解决办法：

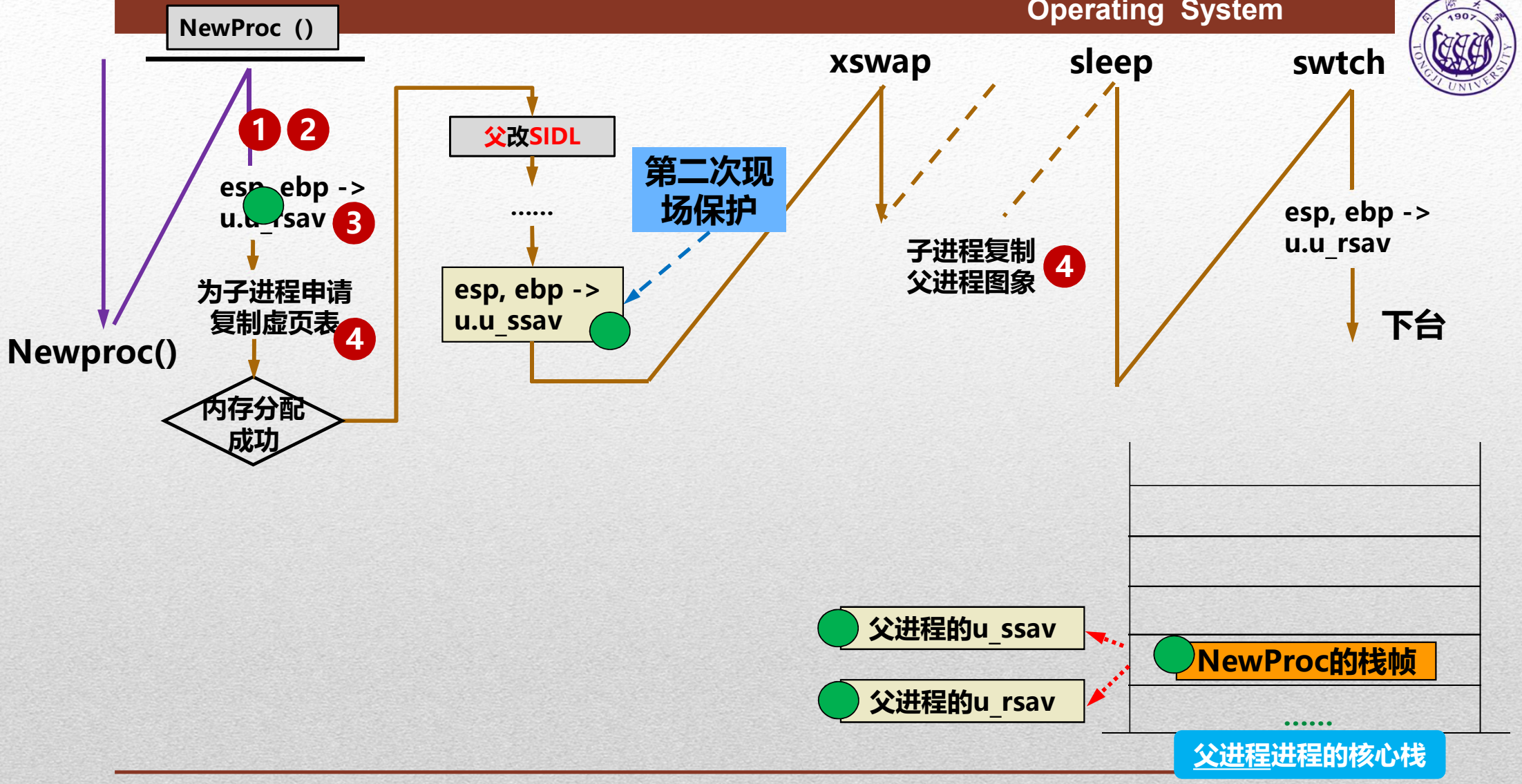
- 减小NPROC
- 加大内存
- 采用虚拟内存技术

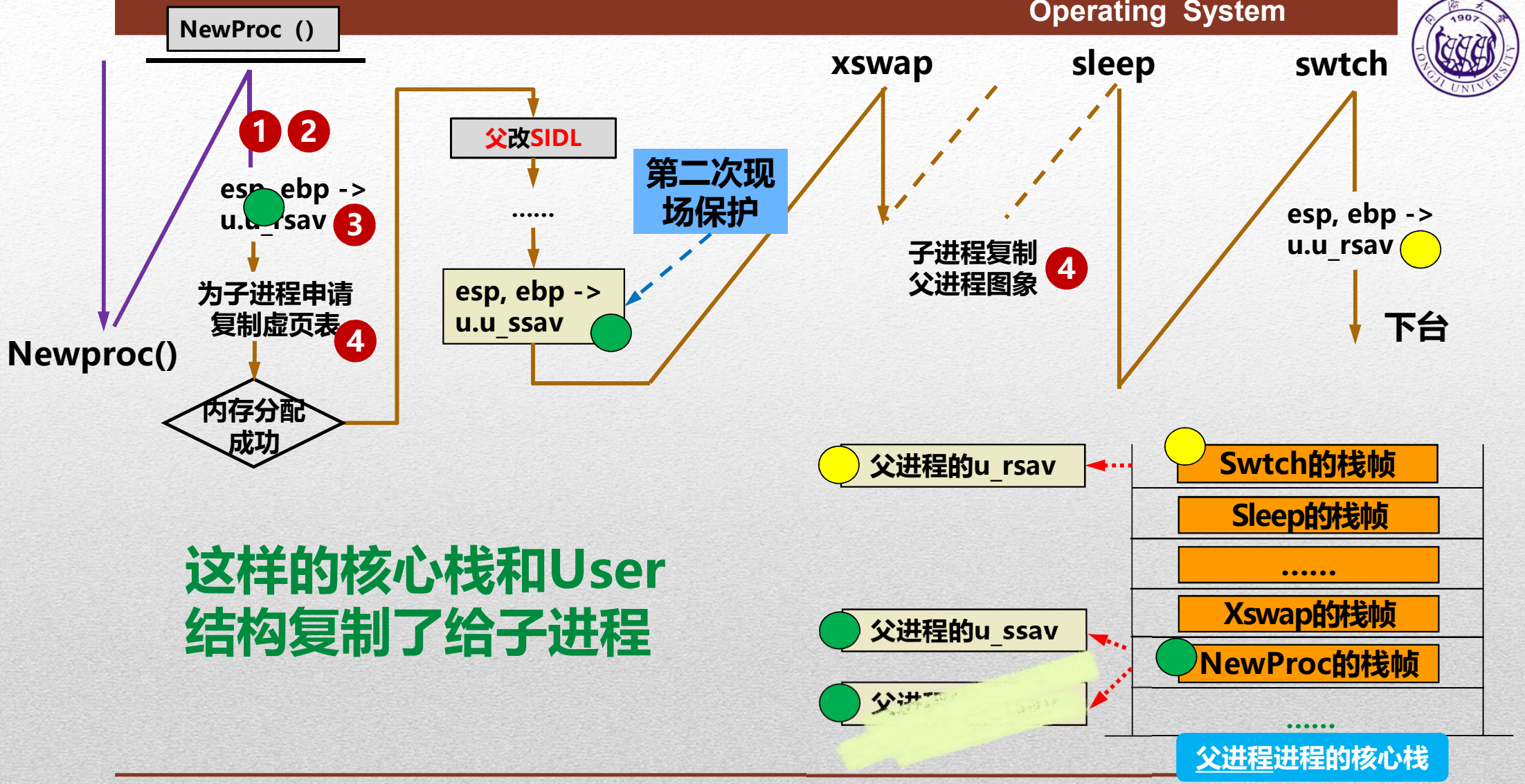
1 当内存空间不足以装下所有就绪进程时……

2 当新进程创建所需的内存空间不时……

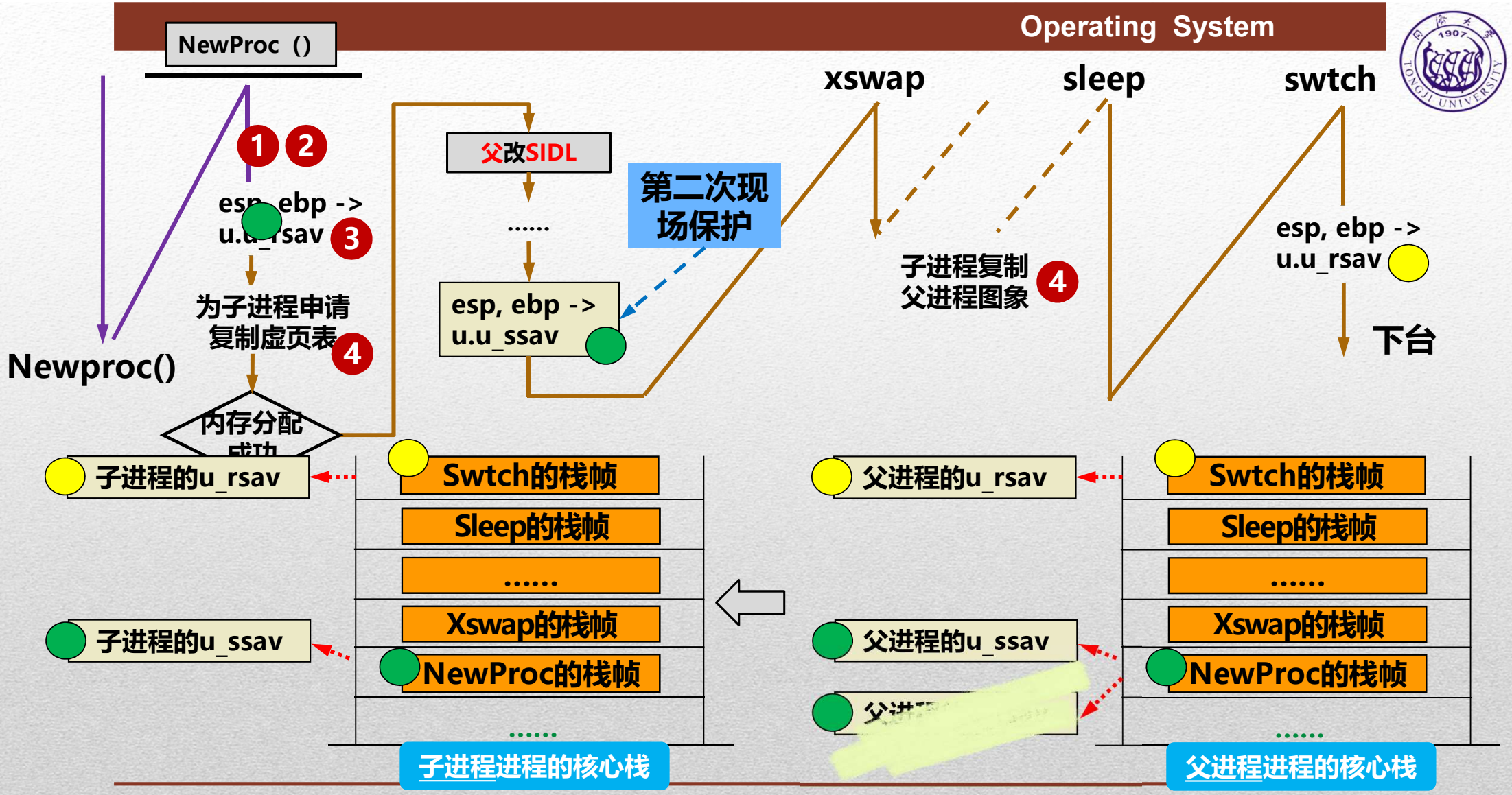

NewProc ()
1
2
esp, ebp ->
u.u_rsav
3
为子进程申请
复制虚页表
4
Newproc()
内存分配
成功
但万一申请内存
失败?
父进程的U区
u_rsav[0]
u_rsav[1]
esp
ebp
父
父
NewProc局部变量
old ebp
NewProc返回地址
NewProc的参数
.....
父进程进程的核心栈
NewProc
的栈帧

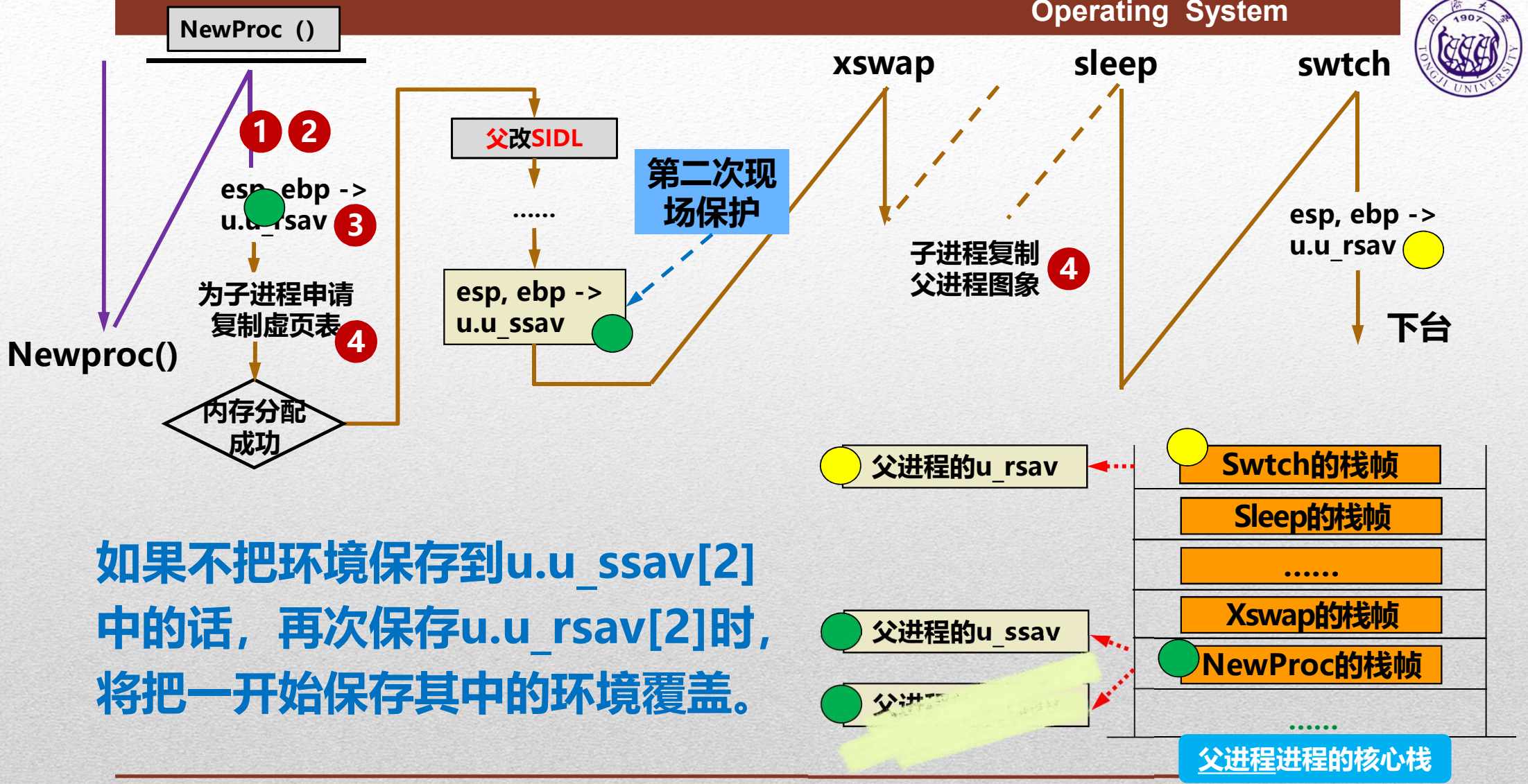

NewProc ()
Newproc()
1
2
esp, ebp ->
u.u_rsav
3
为子进程申请
复制虚页表
4
内存分配
成功
父改SIDL
.....
esp, ebp ->
u.u_ssav
xswap
sleep
swtch
子进程复制
父进程图象
4
esp, ebp ->
u.u_rsav
下台
NewProc的栈帧
父进程的u_rsav
.....
父进程进程的核心栈


NewProc ()
Newproc()
1
2
esp, ebp ->
u.u_rsav
3
为子进程申请
复制虚页表
4
内存分配
成功
父改SIDL
.....
esp, ebp ->
u.u_ssav
第二次现
场保护
xswap
sleep
swtch
子进程复制
父进程图象
4
esp, ebp ->
u.u_rsav
下台
父进程的u_ssav
父进程的u_rsav
NewProc的栈帧
.....
父进程进程的核心栈

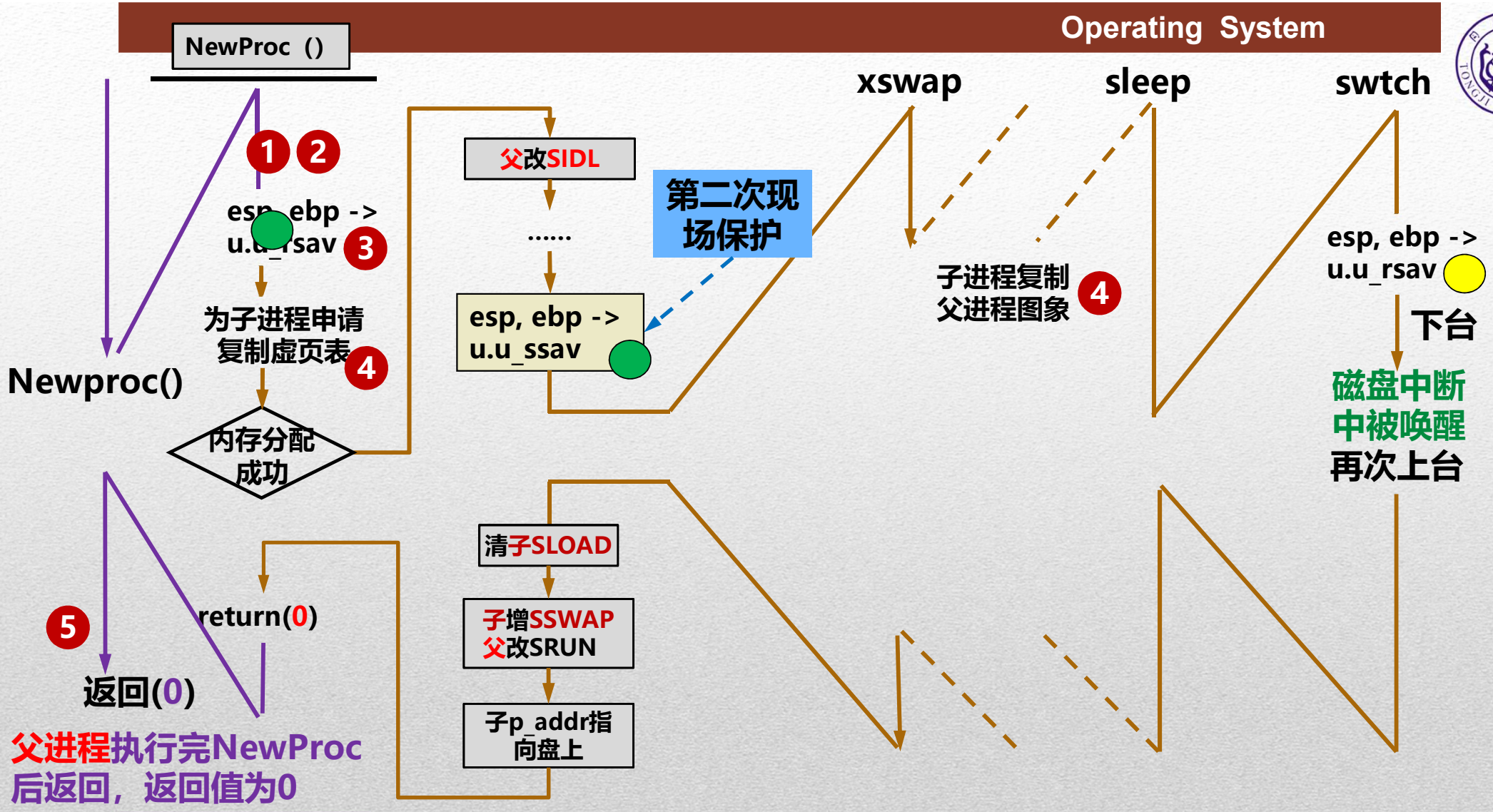


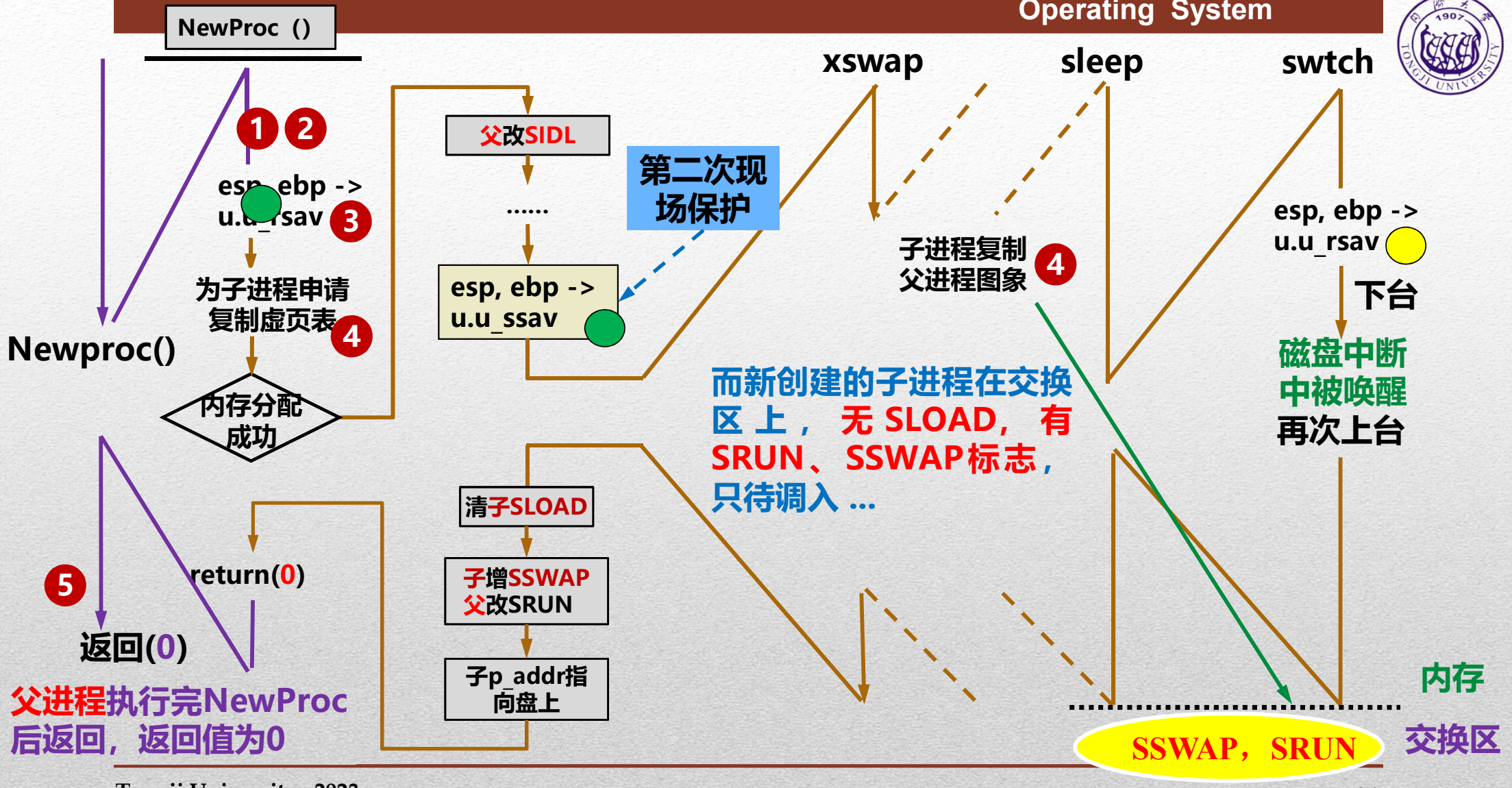
这样的核心栈和User结构复制了给子进程


NewProc ()
Newproc()
1
2
esp, ebp -> u.u_rsav
3
为子进程申请复制虚页表
4
内存分配成功
父改SIDL
.....
esp, ebp -> u.u_ssav
第二次现场保护
xswap
sleep
swtch
子进程复制父进程图象
4
esp, ebp -> u.u_rsav
下台
子进程的u_rsav
子进程的u_ssav
Swtch的栈帧
Sleep的栈帧
.....
Xswap的栈帧
NewProc的栈帧
.....
子进程进程的核心栈
父进程的u_rsav
父进程的u_ssav
Swtch的栈帧
Sleep的栈帧
.....
Xswap的栈帧
NewProc的栈帧
.....
父进程进程的核心栈



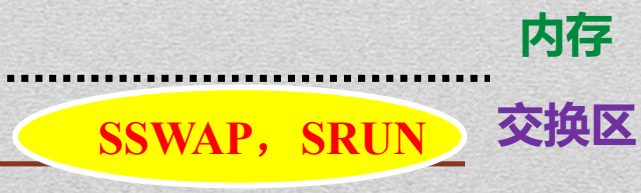
如果不把环境保存到u.u_ssav[2]中的话，再次保存u.u_rsav[2]时，将把一开始保存其中的环境覆盖。


NewProc ()
Newproc()
1
2
esp, ebp ->
u.u_rsav
3
为子进程申请
复制虚页表
4
内存分配
成功
父改SIDL
.....
esp, ebp ->
u.u_ssav
第二次现
场保护
xswap
sleep
swtch
子进程复制
父进程图象
4
esp, ebp ->
u.u_rsav
下台
磁盘中断
中被唤醒
再次上台
清子SLOAD
子增SSWAP
父改SRUN
子p_addr指
向盘上
return(0)
5
返回(0)
父进程执行完NewProc
后返回，返回值为0


NewProc ()
Newproc()
1
2
esp, ebp -> u.u_rsav
3
为子进程申请复制虚页表
4
内存分配成功
父改SIDL
.....
esp, ebp -> u.u_ssav
第二次现场保护
清子SLOAD
子增SSWAP 父改SRUN
子p_addr指向盘上
return(0)
5
返回(0)
父进程执行完NewProc后返回，返回值为0
xswap
sleep
swtch
子进程复制父进程图象
4
而新创建的子进程在交换区上，无SLOAD，有SRUN、SSWAP标志，只待调入 ...
esp, ebp -> u.u_rsav
下台
磁盘中断中被唤醒
再次上台
内存
交换区
SSWAP，SRUN

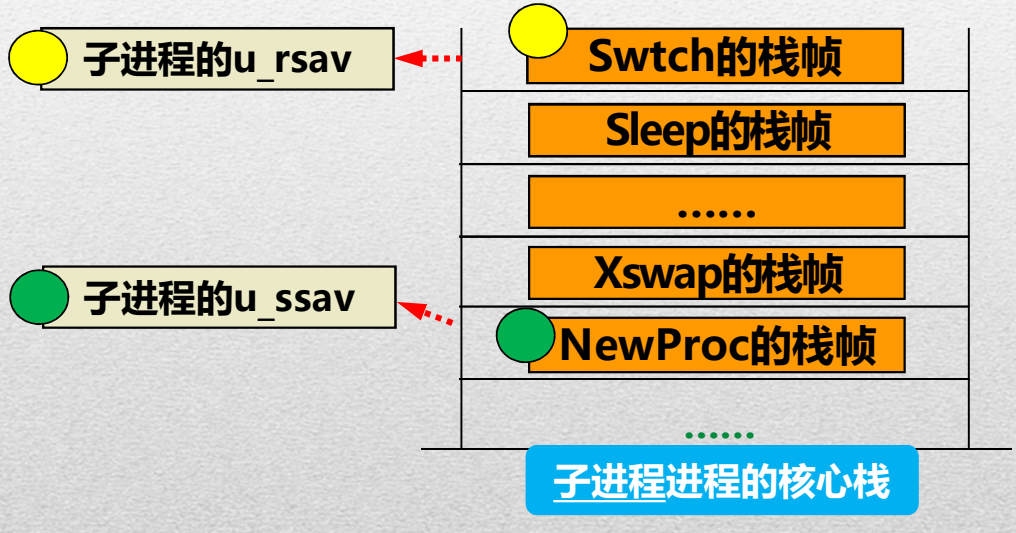
对于刚才创建完毕、位于交换区上的子进程来说，是如何上台的呢?

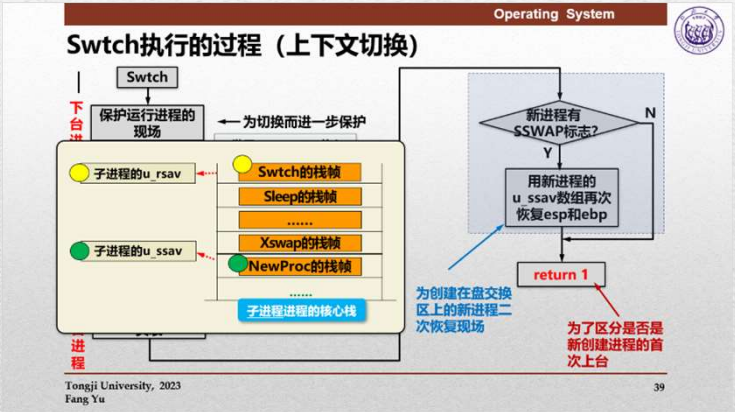
未来某个时刻，一旦0# 发现交换区上有就绪进程，迟早会调入。



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

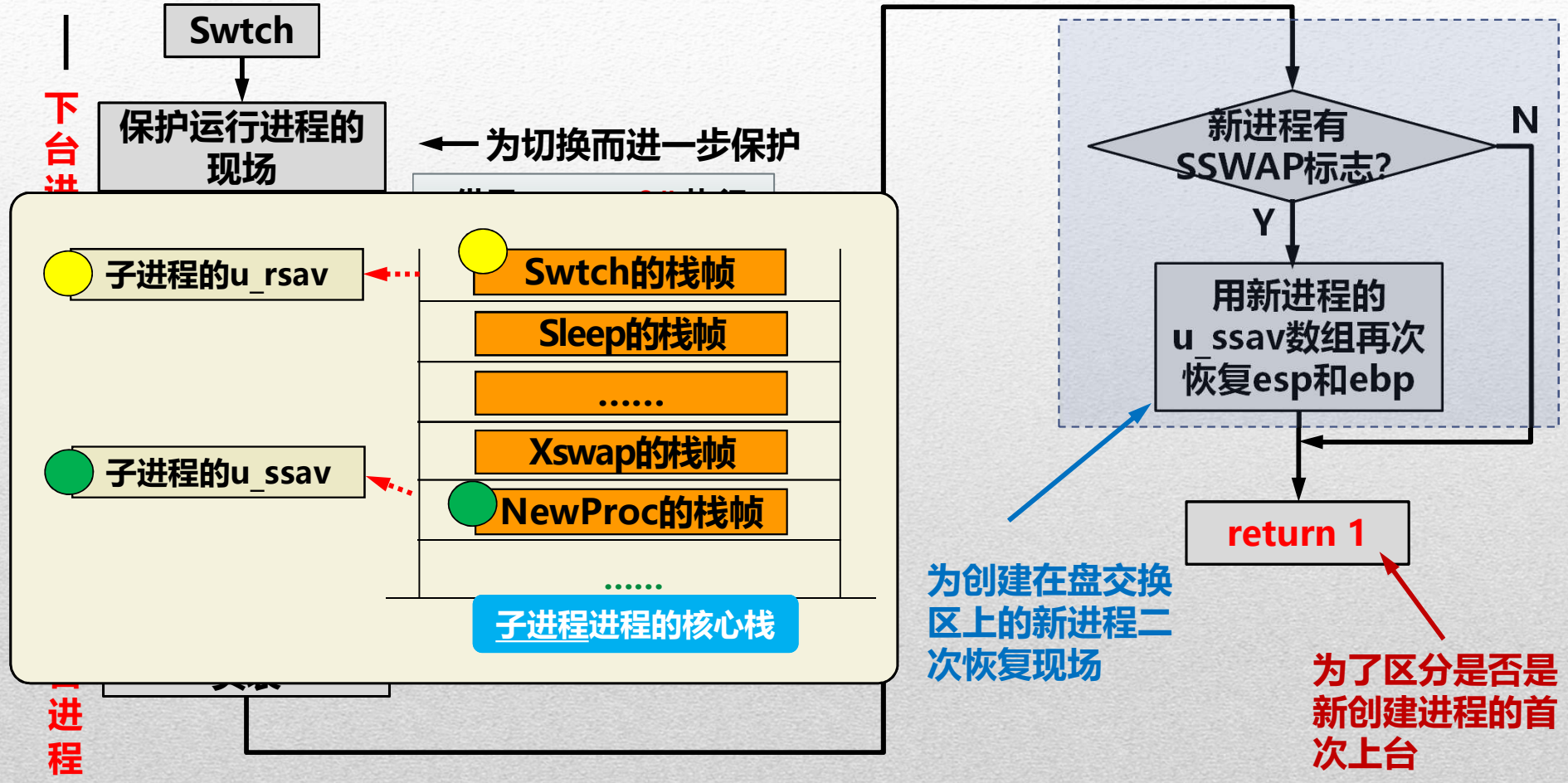


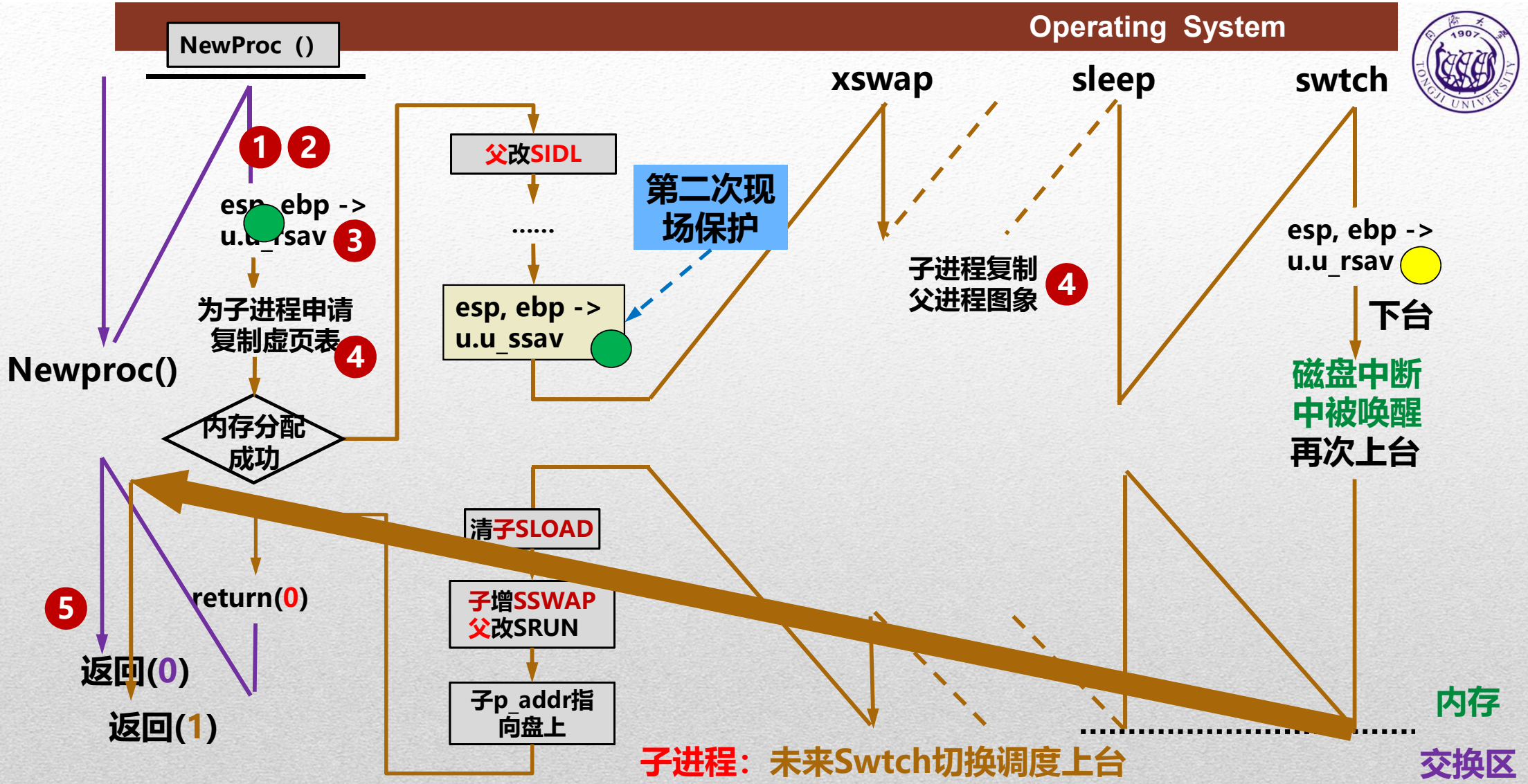


SLOAD 内存

SSWAP，SRUN 交换区

Swtch执行的过程（上下文切换）

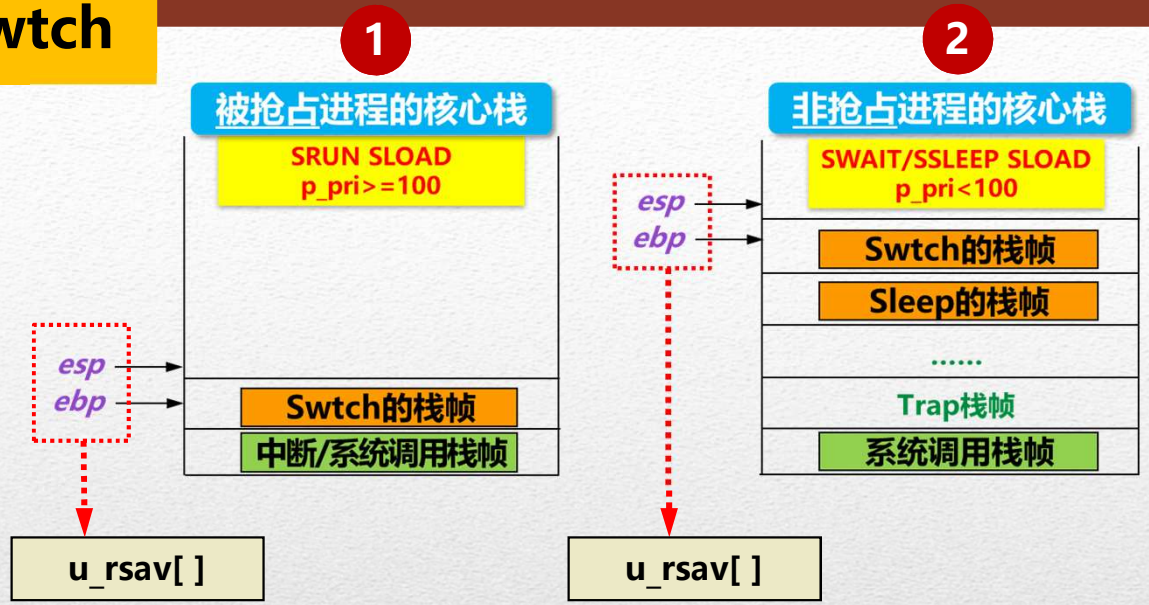


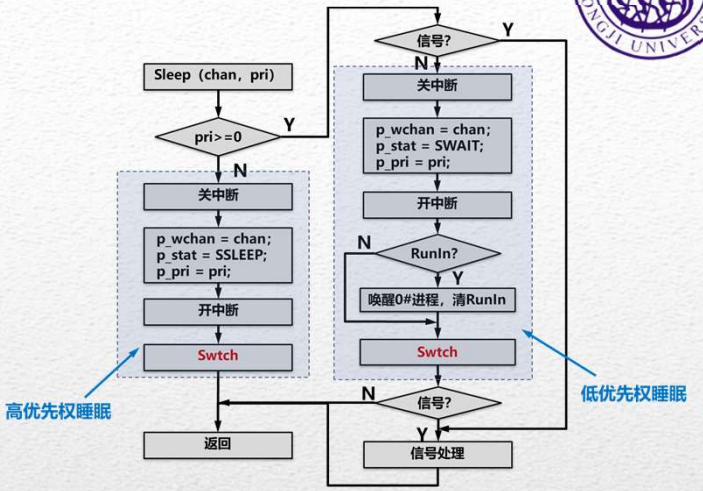

NewProc ()
xswap
sleep
swtch
1
2
esp, ebp ->
u.u_rsav
3
为子进程申请
复制虚页表
4
Newproc()
内存分配
成功
父改SIDL
.....
esp, ebp ->
u.u_ssav
第二次现
场保护
子进程复制
父进程图象
4
esp, ebp ->
u.u_rsav
下台
磁盘中断
中被唤醒
再次上台
5
return(0)
返回(0)
返回(1)
清子SLOAD
子增SSWAP
父改SRUN
子p_addr指
向盘上
子进程：未来Swtch切换调度上台
内存
交换区

进程控制相关的主要内核函数

Swtch

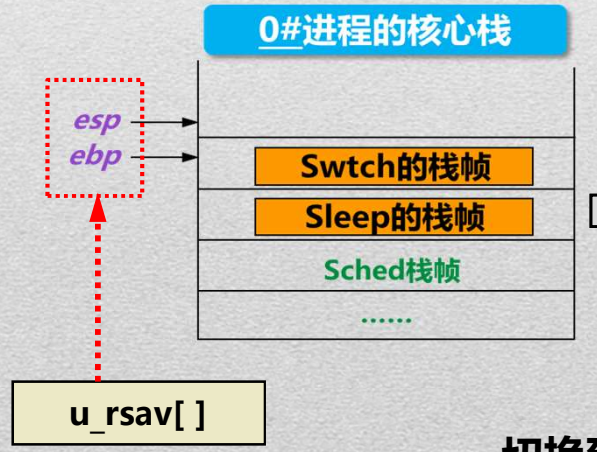
原进程保存现场后下台





切换到0#的核心态页表，恢复0#现场

0# 恢复现场选新进程



0#：SSLEEP，-100
(Sched中，&RunIn或&RunOut)

- 有内存就绪进程：选优先级最高者 (有可能仍是原进程)
- 无内存就绪进程：等中断

0#：SRUN，-100
(唤醒一个盘交换区进程、有进程低睡、1秒计时到)

就绪中的0#选到自己

切换到新进程的核心态页表，恢复新进程现场

Swtch

新进程恢复现场上台执行

1

被抢占进程的核心栈

SRUN SLOAD
p_pri>=100

esp
ebp

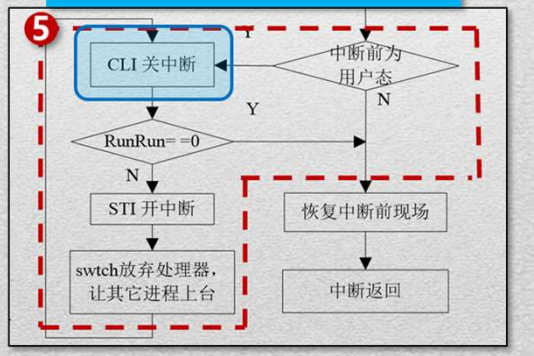
Swtch的栈帧

中断/系统调用栈帧

u_rsav[ ]

执行Swtch中return 1

Swtch栈帧中的返回地址



2

非抢占进程的核心栈

SRUN SLOAD
p_pri<100

esp
ebp

Swtch的栈帧

Sleep的栈帧

.....

Trap栈帧

系统调用栈帧

u_rsav[ ]

执行Swtch中的return 1

↓

Sleep

↓

.....

↓

Trap

↓

可能被抢占

↓

返回用户态

3.1

新进程的核心栈

SRUN SLOAD
p_pri = 0

esp
ebp

NewProc的栈帧

.....

Trap栈帧

系统调用栈帧

u_rsav[ ]

创建在内存的新进程

执行Swtch中的return 1

↓

Fork

↓

.....

↓

Trap

↓

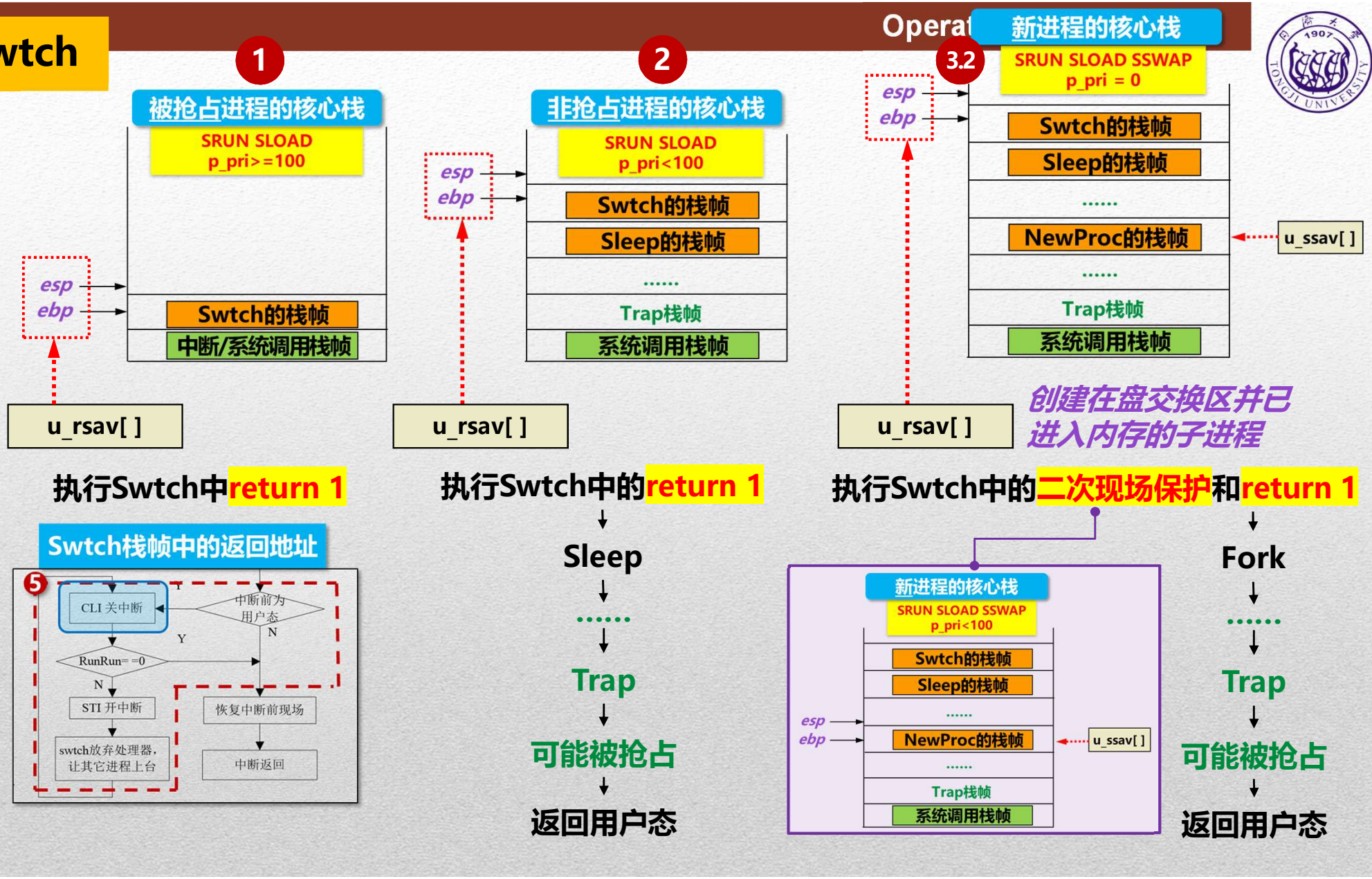
可能被抢占

↓

返回用户态

Swtch

新进程恢复现场上台执行



本节小结：

1 UNIX V6++中的进程图象交换

2 UNIX V6++中的0#进程

3 UNIX V6++中创建在盘交换区的新进程