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

UNIX系统调用

现代计算机系统中，中断的概念被扩展……

中断源

CPU外

1. I/O完成或出错引起的设备中断 — 中断

CPU停下正在执行的代码，转去执行一段特定的内核代码

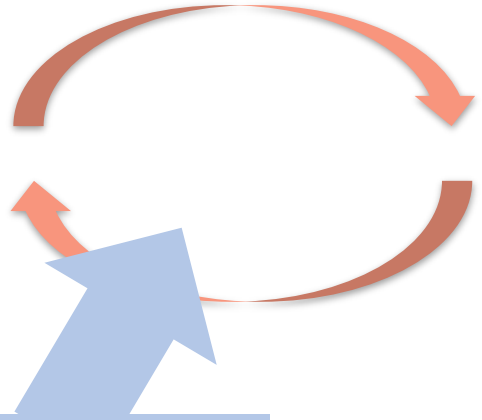
CPU内

2. 断点跟踪
3. 硬件故障（奇、偶校验错、电源）
4. 程序性故障（浮点、除0、地址溢出） — 异常/陷入
5. 系统调用（特殊指令，如：INT 80） — 自陷

UNIX系统调用

用户态
User Mode

执行用户程序，
提供用户功能

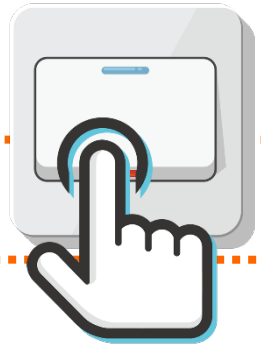


核心态
Kernel Mode

执行内核程序，
提供系统功能

可以在一定时机相互转换

内核不是与用户进程平行运行的孤立的进程集合。



中断、异常、系统调用

在核心态下执行内核代码的进程完成了内核功能！

UNIX系统调用

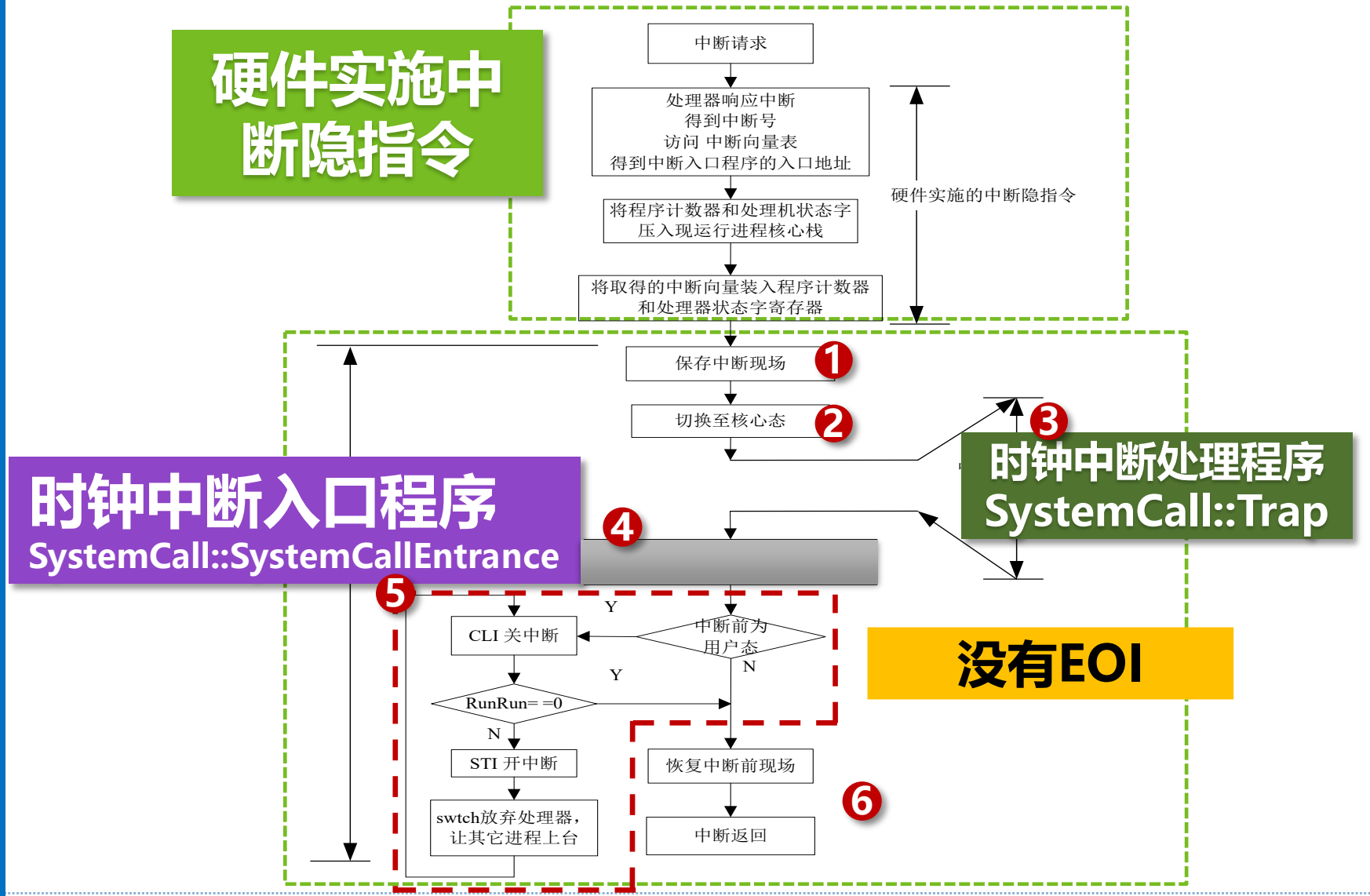
UNIX V6++定义的中断入口程序

中断号/中断源		中断入口程序	中断处理子程序
0x0	除0错	Exception::DivideErrorEntrance()	Exception::DivideError (struct pt_regs* regs, struct pt_context* context);
0x1	调试异常	Exception::DebugEntrance();	Exception::Debug (struct pt_regs* regs, struct pt_context* context);
0x2	NMI非屏蔽中断	Exception::NMIEntrance();	Exception:: NMI (struct pt_regs* regs, struct pt_context* context);
0x3	调试断点	Exception::BreakpointEntrance();	Exception:: Breakpoint (struct pt_regs* regs, struct pt_context* context);
……	……	……	……
0x1F	保留异常		
0x20	时钟中断	Time::TimeInterruptEntrance()	void Time::Clock(struct pt_regs* regs, struct pt_context* context)
0x21	键盘中断	KeyboardInterrupt::KeyboardInterruptEntrance()	void Keyboard::KeyboardHandler(struct pt_regs* reg, struct pt_
…	…		
0x2E	硬盘中	nce()	void ATADriver…ATAHandler(struct pt_regs *reg, struct pt_context *context)
….	…		…
0x80	系统调用	void SystemCall::SystemCallEntrance()	void SystemCall::Trap(struct pt_regs* regs, struct pt_context* context)
…	…	…	…

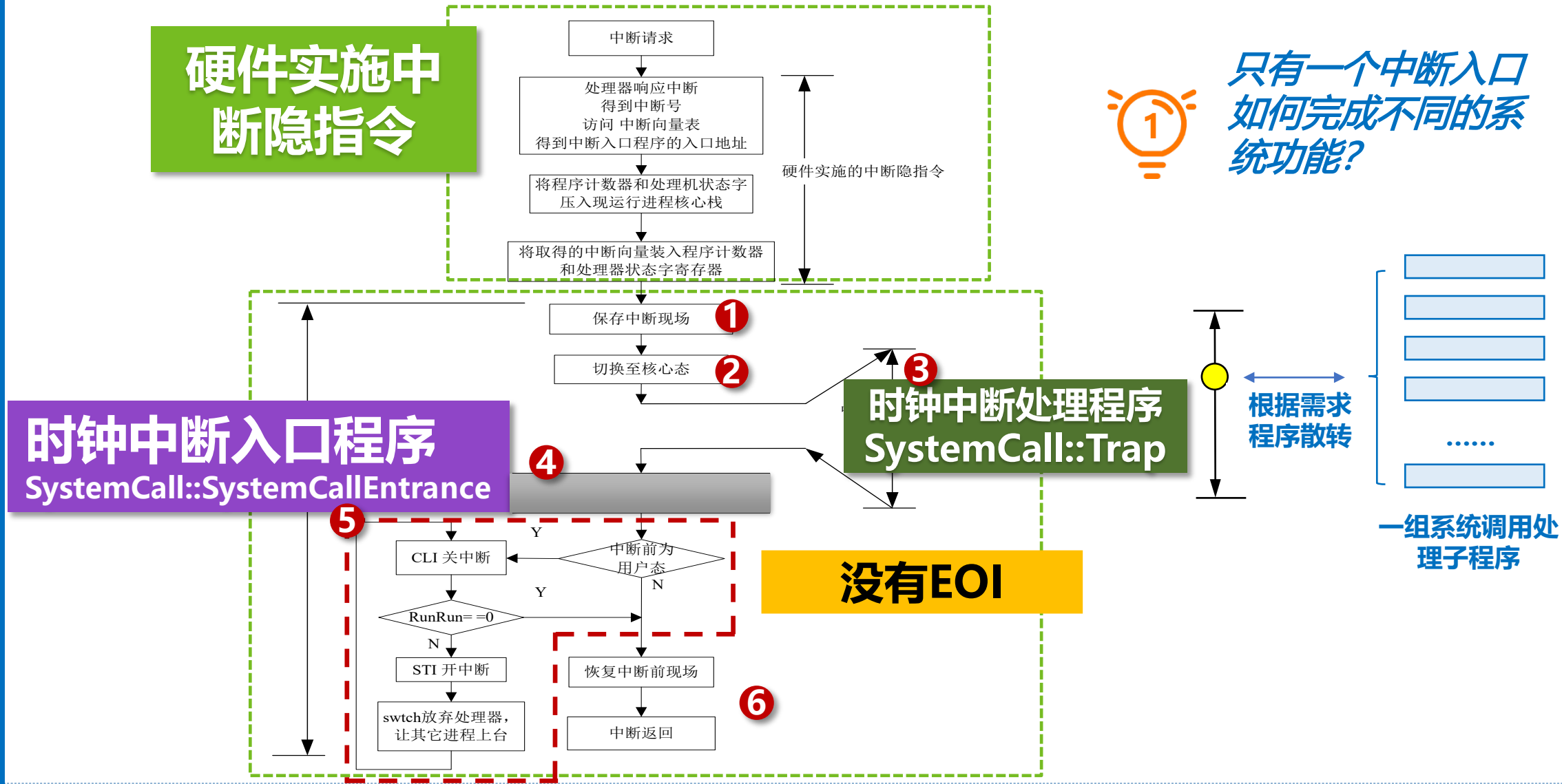
在中断描述符表的0x80号中断处登记地址

入口程序跳转至处理程序

UNIX系统调用

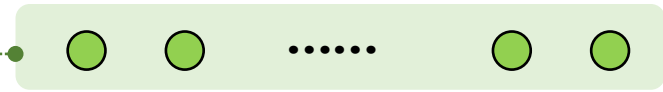

硬件实施中断隐指令
中断请求
处理器响应中断
得到中断号
访问 中断向量表
得到中断入口程序的入口地址
将程序计数器和处理机状态字压入现运行进程核心栈
将取得的中断向量装入程序计数器和处理器状态字寄存器
硬件实施的中断隐指令
保存中断现场
1
切换至核心态
2
3
时钟中断处理程序
SystemCall::Trap
时钟中断入口程序
SystemCall::SystemCallEntrance
4
5
Y
CLI 关中断
中断前为用户态
N
Y
RunRun= =0
N
STI 开中断
swtch放弃处理器，让其它进程上台
恢复中断前现场
6
中断返回
没有EOI

UNIX系统调用


硬件实施中断隐指令
中断请求
处理器响应中断
得到中断号
访问 中断向量表
得到中断入口程序的入口地址
将程序计数器和处理机状态字压入现运行进程核心栈
将取得的中断向量装入程序计数器和处理器状态字寄存器
硬件实施的中断隐指令
保存中断现场
切换至核心态
时钟中断入口程序
SystemCall::SystemCallEntrance
时钟中断处理程序
SystemCall::Trap
CLI 关中断
中断前为用户态
RunRun= =0
STI 开中断
swtch放弃处理器，让其它进程上台
恢复中断前现场
中断返回
Y
N
没有EOI
只有一个中断入口如何完成不同的系统功能?
根据需求程序散转
.....
一组系统调用处理子程序

UNIX系统调用


```
struct SystemCallTableEntry
{
unsigned int count;//系统调用的参数个数
int(*call)();       //相应系统调用处理函数的指针
};

class SystemCall
{
public:
static const unsigned int SYSTEM_CALL_NUM = 64;
private:
static SystemCallTableEntry m_SystemEntranceTable[SYSTEM_CALL_NUM];

public:
static void SystemCallEntrance();
static void Trap(struct pt_regs* regs, struct pt_context* context);
static void Trap1(int (*func)());

private:
static int Sys_NullSystemCall();
static int Sys_Rexit();
static int Sys_Fork();
……
};
```



提供一组系统调用处理子程序

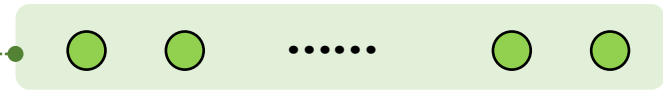
UNIX系统调用

```
struct SystemCallTableEntry
{
unsigned int count;//系统调用的参数个数
int(*call)();       //相应系统调用处理函数的指针
};

class SystemCall
{
public:
static const unsigned int SYSTEM_CALL_NUM = 64;
private:
static SystemCallTableEntry m_SystemEntranceTable[SYSTEM_CALL_NUM];

public:
static void SystemCallEntrance();
static void Trap(struct pt_regs* regs, struct pt_context* context);
static void Trap1(int (*func)());

private:
static int Sys_NullSystemCall();
static int Sys_Rexit();
static int Sys_Fork();
……
};
```

定义一个包含64个分量的系统调用处理子程序入口表

系统调用子程序入口表

0	0	&Sys_NullSystemCall
1	1	&Sys_Rexit
2	0	&Sys_Fork
3	3	&Sys_Read
4	3	&Sys_Write
5	2	&Sys_Open
…		……
63	0	&Sys_Nosys

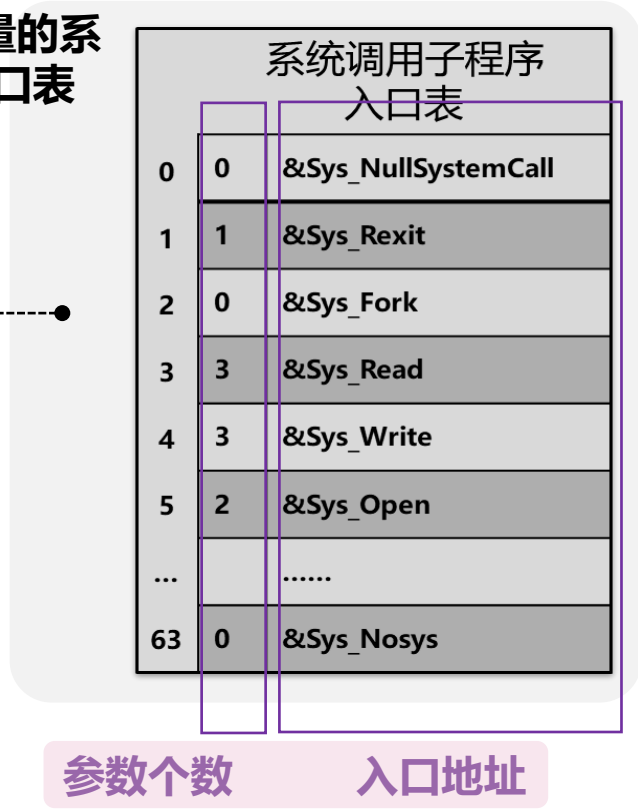


提供一组系统调用处理子程序

UNIX系统调用

```
struct SystemCallTableEntry
{
unsigned int count;//系统调用的参数个数
int(*call)();       //相应系统调用处理函数的指针
};

class SystemCall
{
public:
static const unsigned int SYSTEM_CALL_NUM = 64;
private:
static SystemCallTableEntry m_SystemEntranceTable[SYSTEM_CALL_NUM];

public:
static void SystemCallEntrance();
static void Trap(struct pt_regs* regs, struct pt_context* context);
static void Trap1(int (*func)());

private:
static int Sys_NullSystemCall();
static int Sys_Rexit();
static int Sys_Fork();
……
};
```

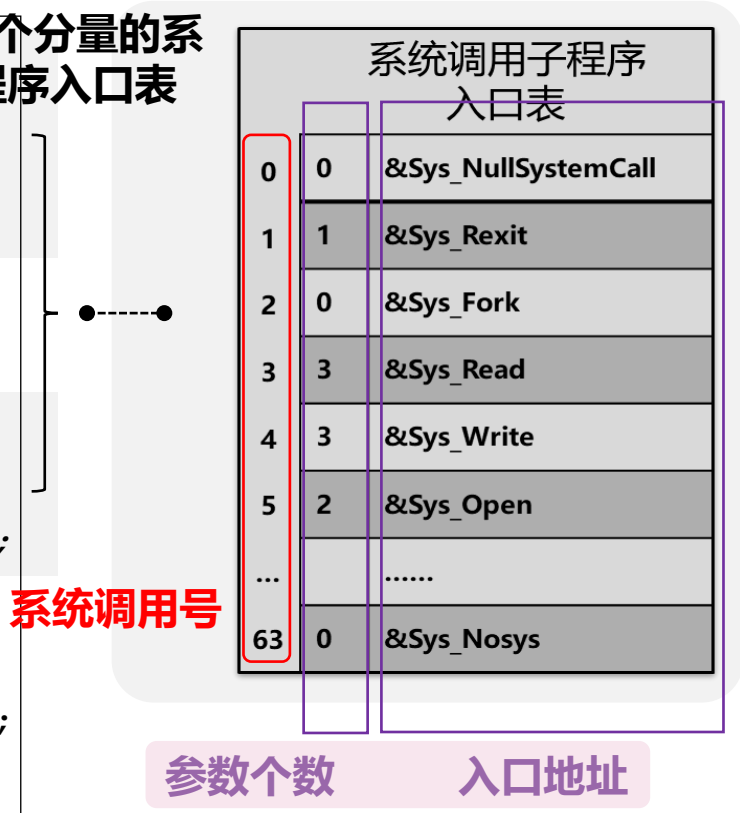
定义一个包含64个分量的系统调用处理子程序入口表



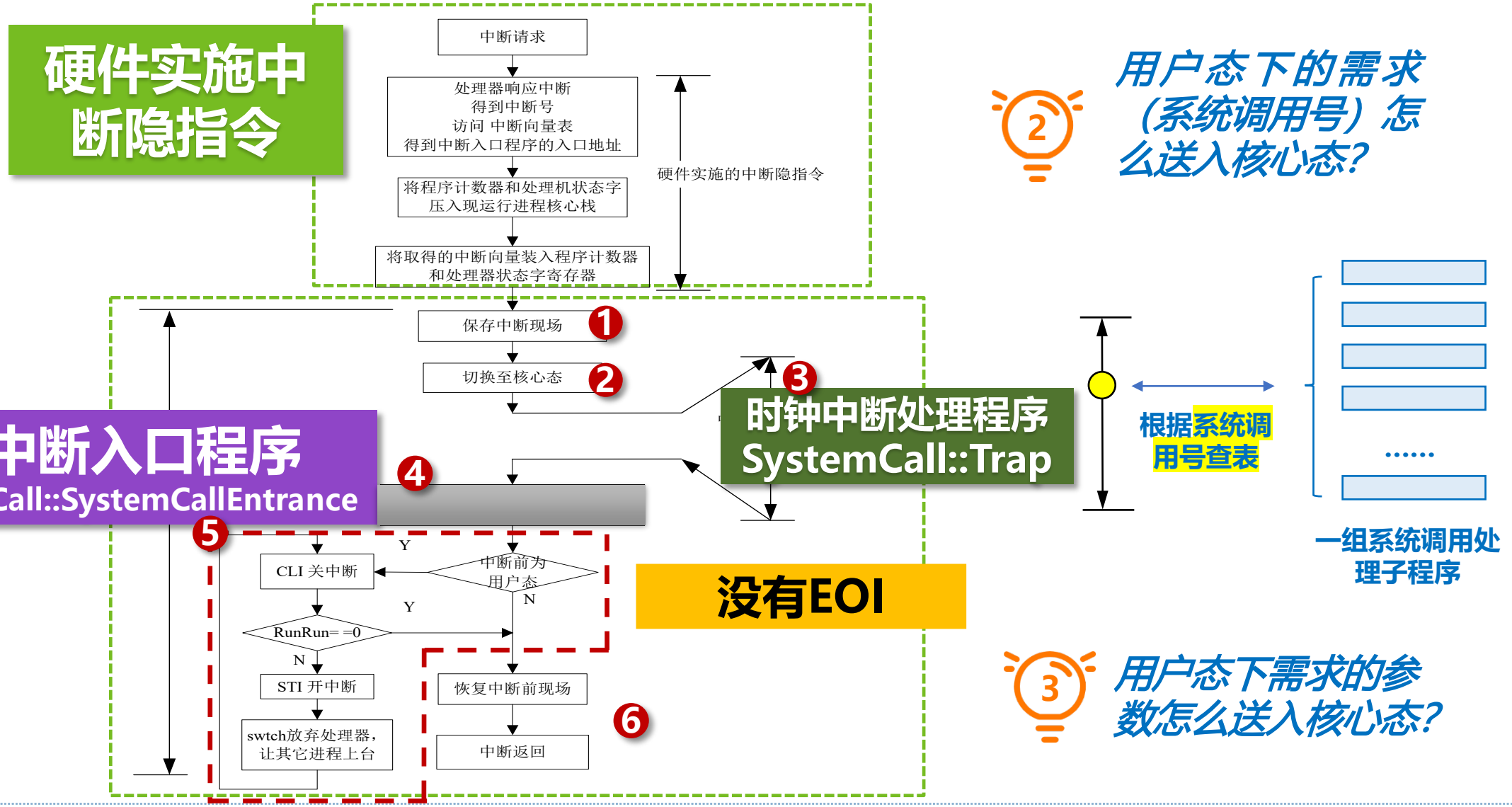
UNIX系统调用

```
struct SystemCallTableEntry
{
unsigned int count;//系统调用的参数个数
int(*call)();       //相应系统调用处理函数的指针
};

class SystemCall
{
public:
static const unsigned int SYSTEM_CALL_NUM = 64;
private:
static SystemCallTableEntry m_SystemEntranceTable[SYSTEM_CALL_NUM];

public:
static void SystemCallEntrance();
static void Trap(struct pt_regs* regs, struct pt_context* context);
static void Trap1(int (*func)());

private:
static int Sys_NullSystemCall();
static int Sys_Rexit();
static int Sys_Fork();
……
};
```

定义一个包含64个分量的系统调用处理子程序入口表



UNIX系统调用


硬件实施中断隐指令
中断请求
处理器响应中断
得到中断号
访问 中断向量表
得到中断入口程序的入口地址
将程序计数器和处理机状态字压入现运行进程核心栈
将取得的中断向量装入程序计数器和处理器状态字寄存器
硬件实施的中断隐指令
保存中断现场
1
切换至核心态
2
3
时钟中断处理程序
SystemCall::Trap
时钟中断入口程序
SystemCall::SystemCallEntrance
4
5
CLI 关中断
中断前为用户态
Y
N
RunRun= =0
STI 开中断
swtch放弃处理器，让其它进程上台
恢复中断前现场
中断返回
6
没有EOI
根据系统调用号查表
.....
一组系统调用处理子程序
2
用户态下的需求（系统调用号）怎么送入核心态?
3
用户态下需求的参数怎么送入核心态?

UNIX系统调用

用户态
User Mode

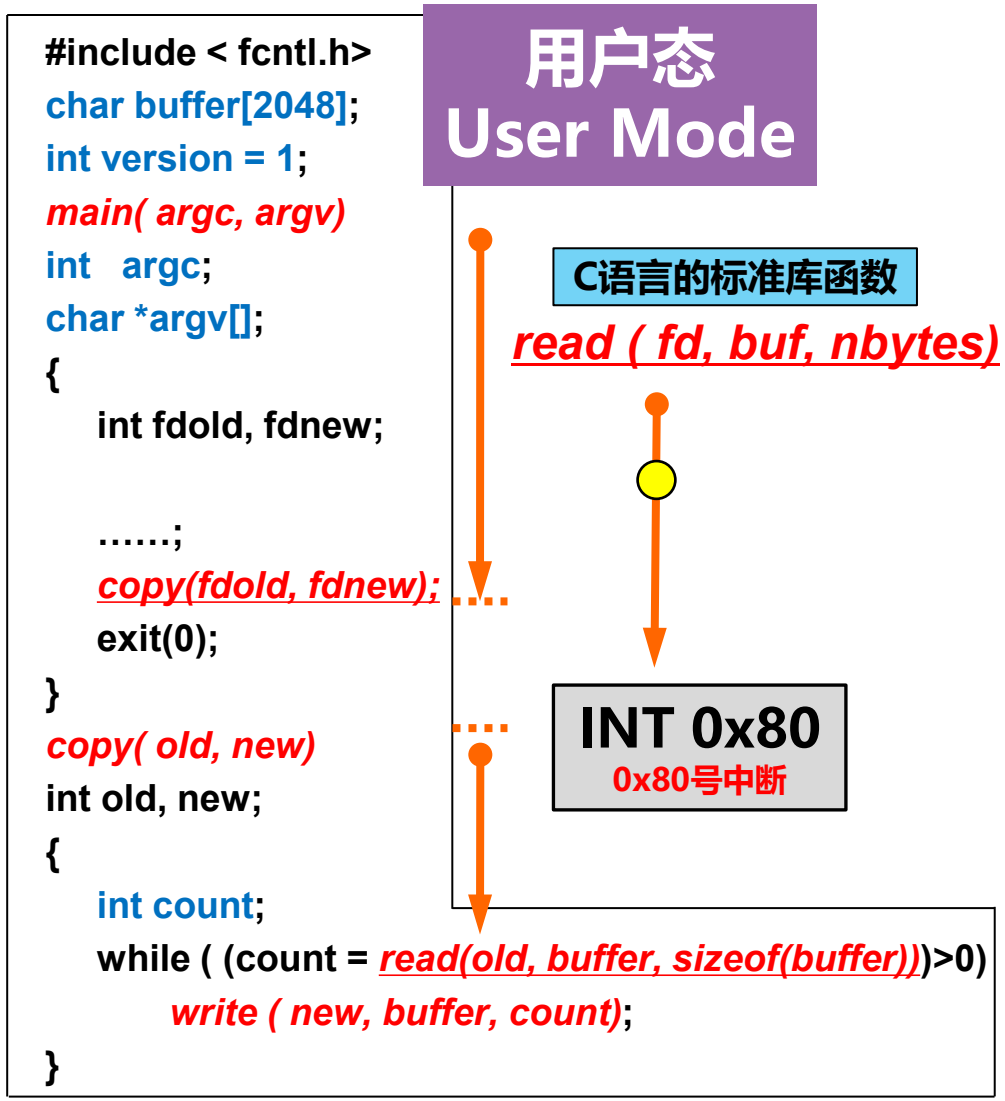
```
#include < fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int   argc;
char *argv[];
{
    int fdold, fdnew;

    ……;
    copy(fdold, fdnew);
    exit(0);
}
copy( old, new)
int old, new;
{
    int count;
    while ( (count = read(old, buffer, sizeof(buffer))>0)
        write ( new, buffer, count);
}
```

UNIX系统调用

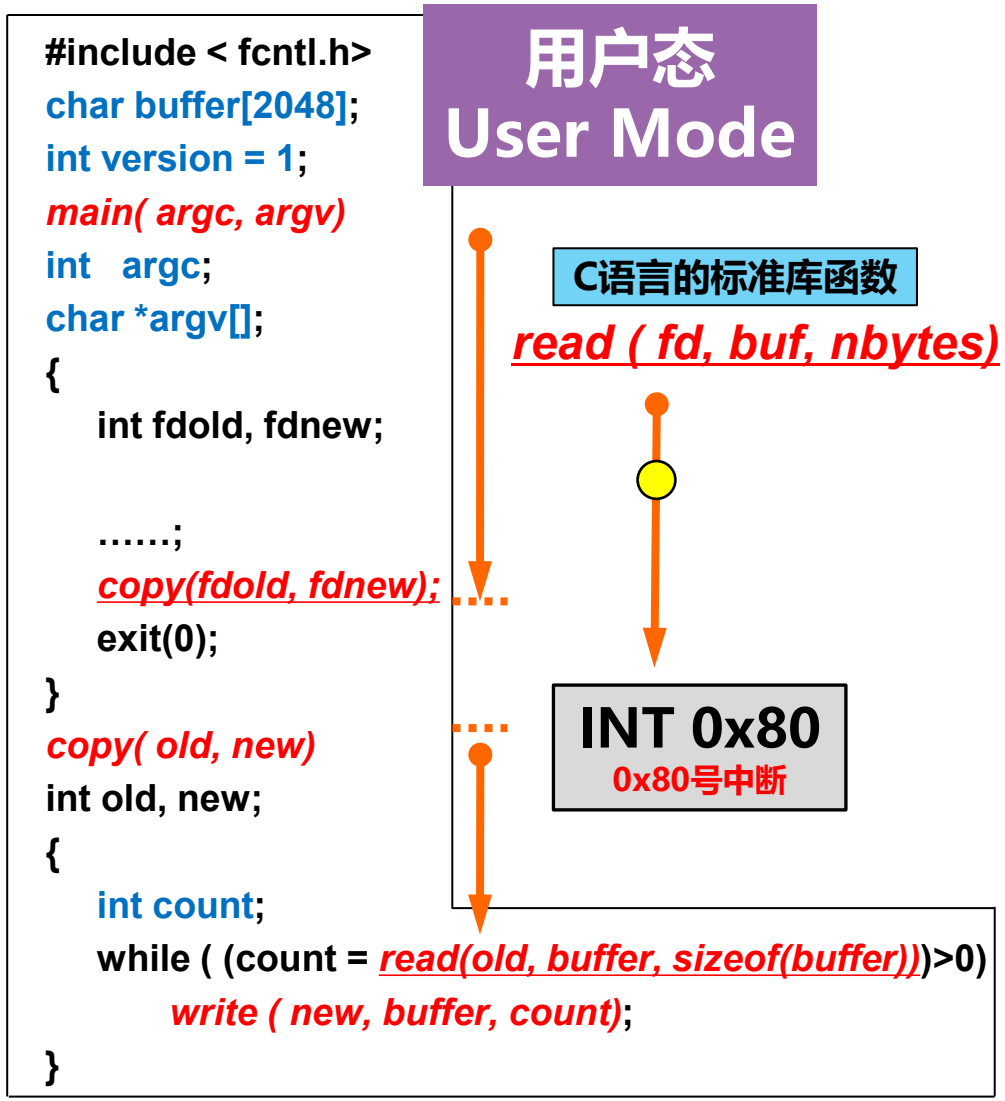
用户态
User Mode

```
#include < fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int   argc;
char *argv[];
{
    int fdold, fdnew;

    ……;
    copy(fdold, fdnew);
    exit(0);
}
copy( old, new)
int old, new;
{
    int count;
    while ( (count = read(old, buffer, sizeof(buffer)))>0)
        write ( new, buffer, count);
}
```

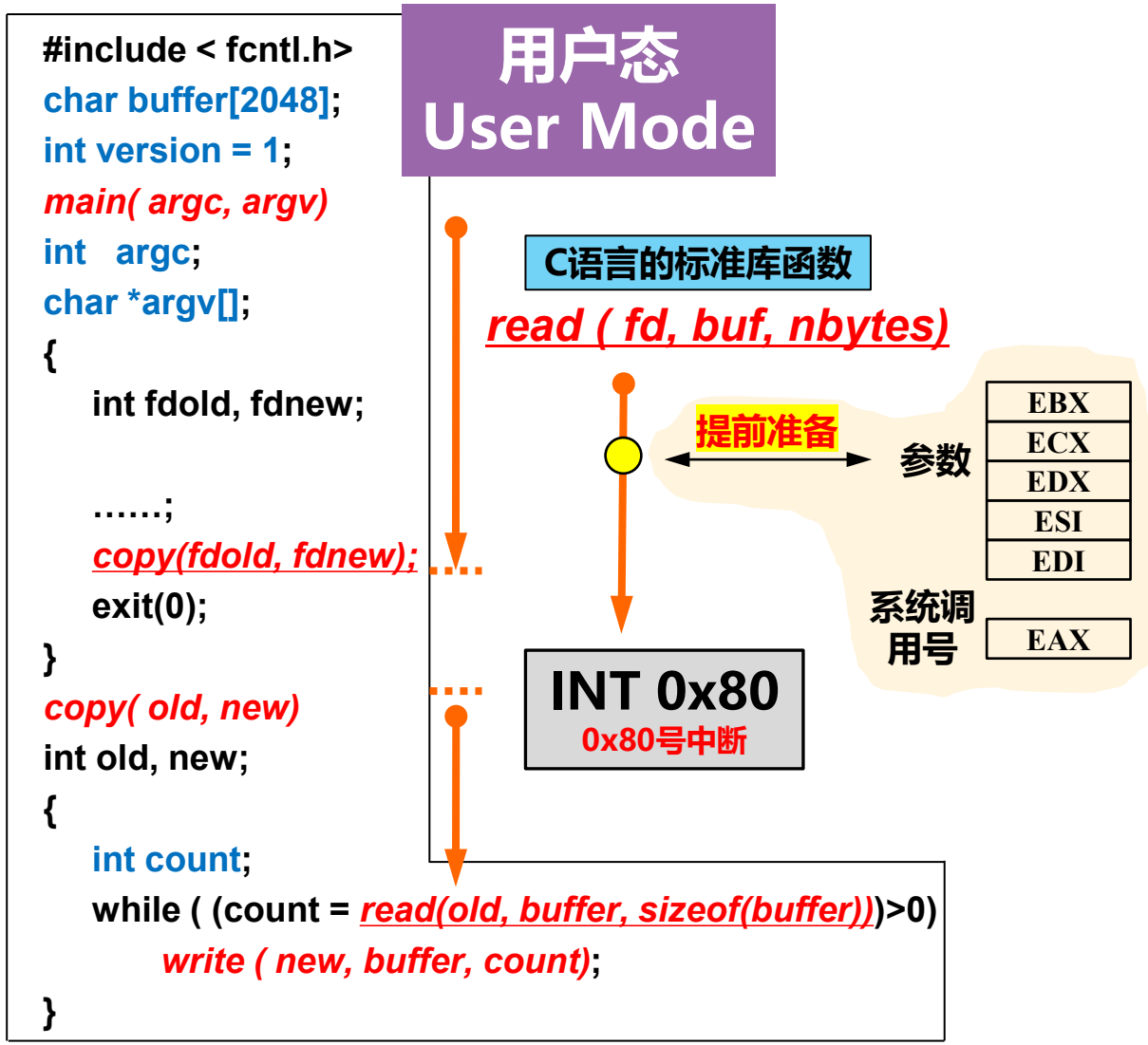
UNIX系统调用


#include < fcntl.h>
char buffer[2048];
int version = 1;
main(argc, argv)
int argc;
char *argv[];
{
int fdold, fdnew;
……;
copy(fdold, fdnew);
exit(0);
}
copy(old, new)
int old, new;
{
int count;
while ((count = read(old, buffer, sizeof(buffer))>0)
write (new, buffer, count);
}
用户态
User Mode
C语言的标准库函数
read (fd, buf, nbytes)
INT 0x80
0x80号中断

UNIX系统调用

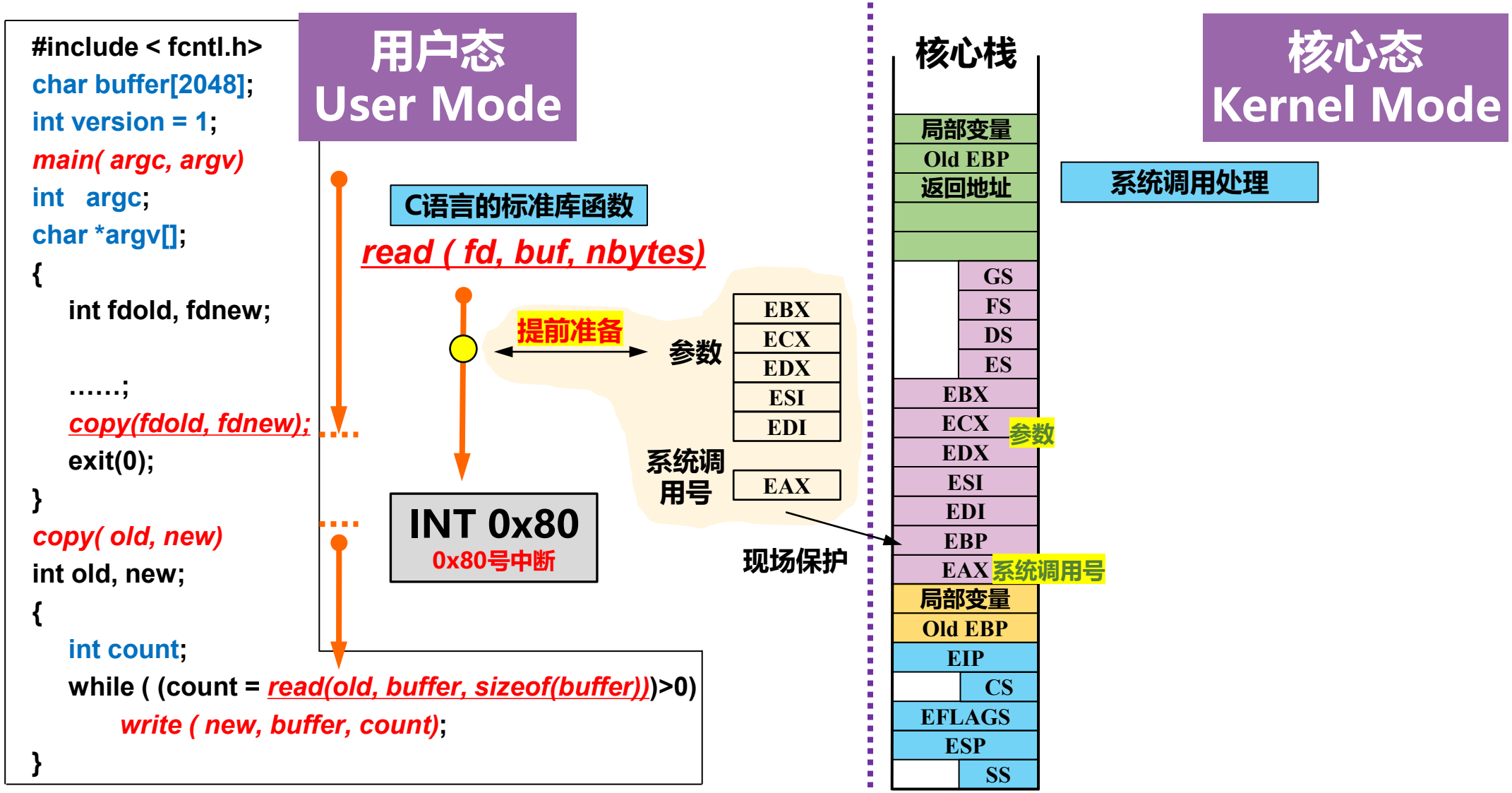

用户态
User Mode
核心态
Kernel Mode
#include < fcntl.h>
char buffer[2048];
int version = 1;
main(argc, argv)
int argc;
char *argv[];
{
int fdold, fdnew;
……;
copy(fdold, fdnew);
exit(0);
}
copy(old, new)
int old, new;
{
int count;
while ((count = read(old, buffer, sizeof(buffer))>0)
write (new, buffer, count);
}
C语言的标准库函数
read (fd, buf, nbytes)
INT 0x80
0x80号中断
系统调用处理

UNIX系统调用


#include < fcntl.h>
char buffer[2048];
int version = 1;
main(argc, argv)
int argc;
char *argv[];
{
int fdold, fdnew;
……;
copy(fdold, fdnew);
exit(0);
}
copy(old, new)
int old, new;
{
int count;
while ((count = read(old, buffer, sizeof(buffer)))>0)
write (new, buffer, count);
}
用户态
User Mode
C语言的标准库函数
read (fd, buf, nbytes)
提前准备
参数
EBX
ECX
EDX
ESI
EDI
系统调用号
EAX
INT 0x80
0x80号中断
核心态
Kernel Mode
系统调用处理

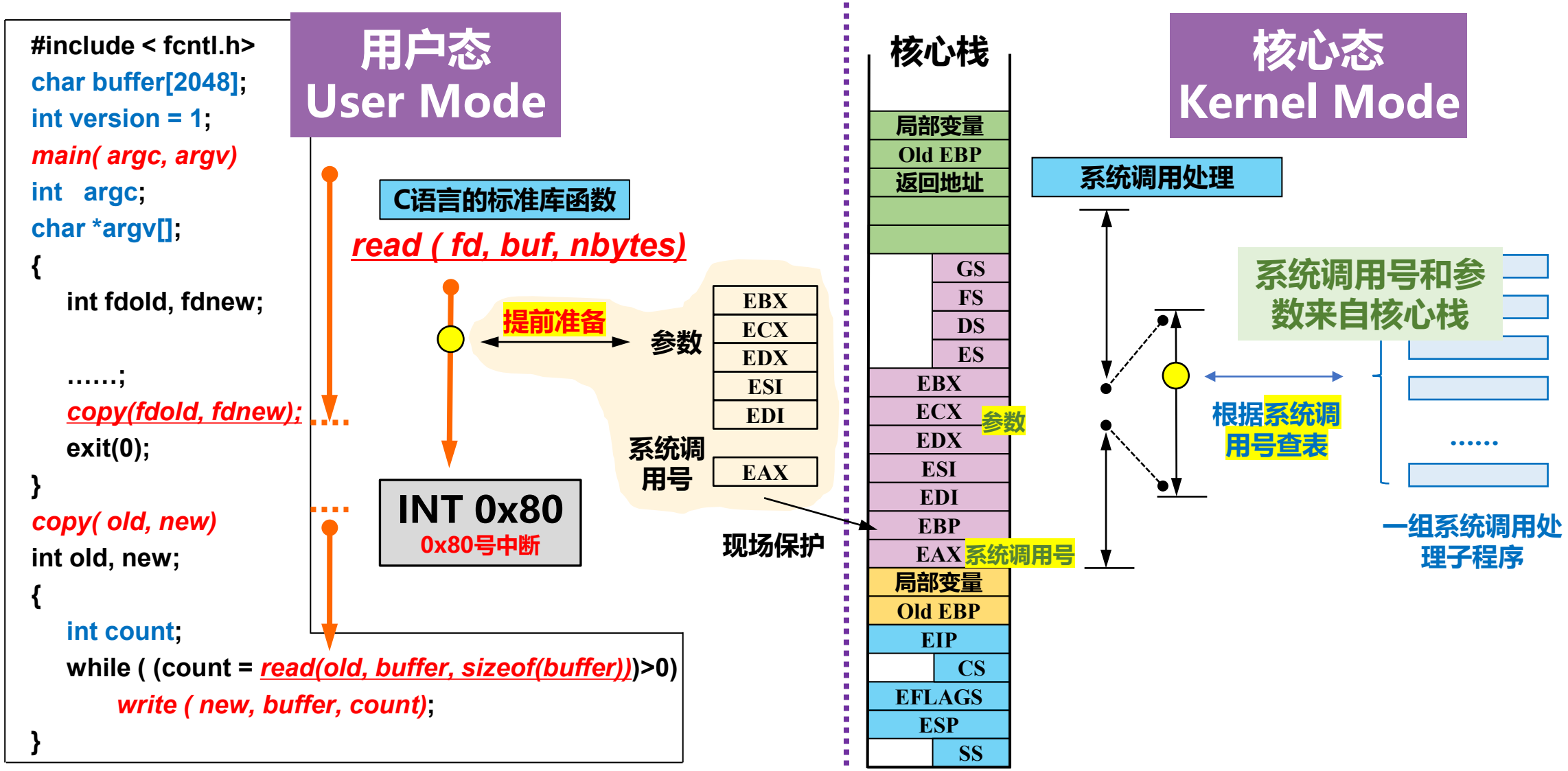
UNIX系统调用

```
#include < fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int   argc;
char *argv[];
{
    int fdold, fdnew;

    ……;
    copy(fdold, fdnew);
    exit(0);
}
copy( old, new)
int old, new;
{
    int count;
    while ( (count = read(old, buffer, sizeof(buffer))>0)
        write ( new, buffer, count);
}
```



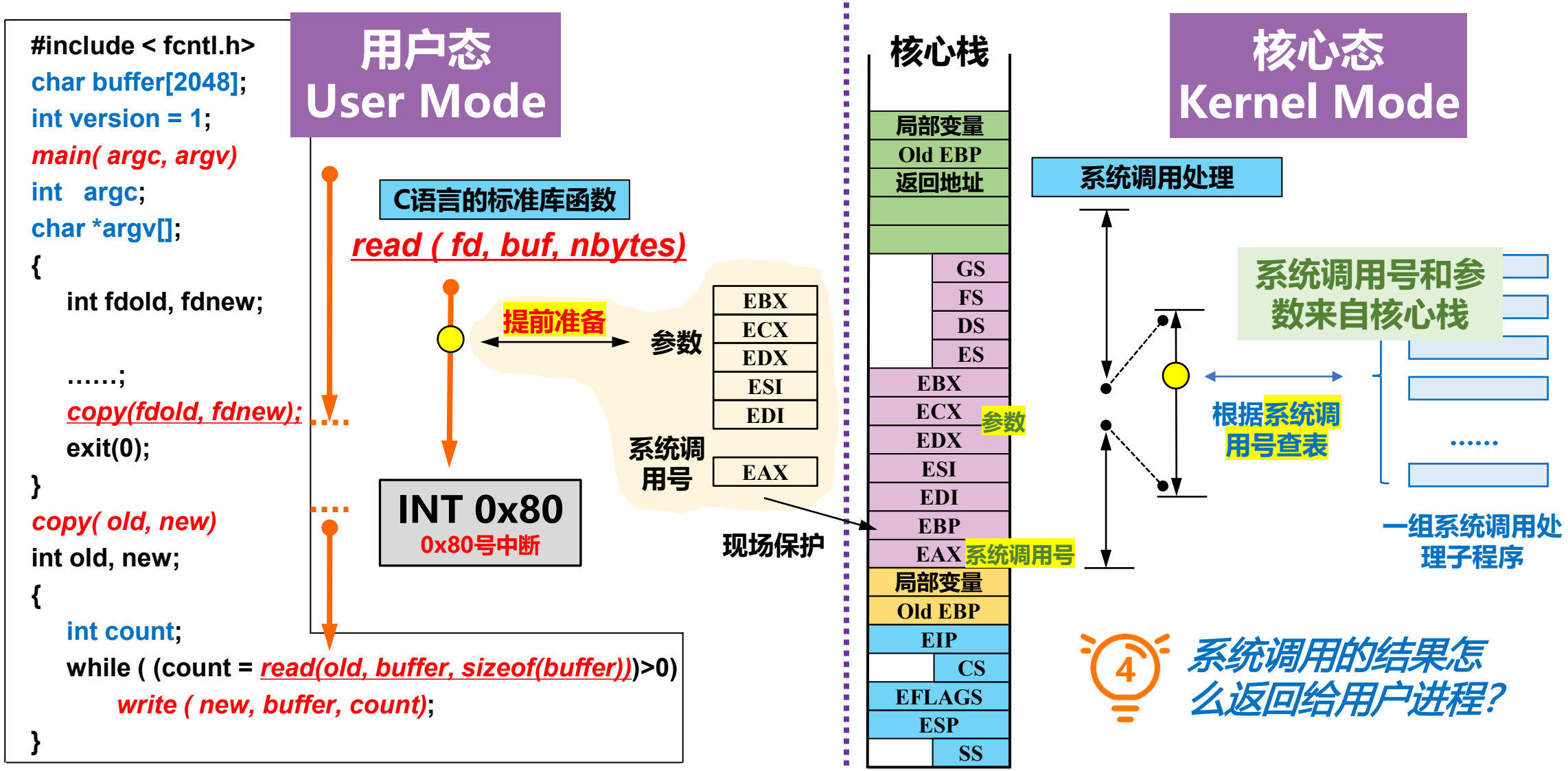
UNIX系统调用

```
#include < fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int   argc;
char *argv[];
{
    int fdold, fdnew;

    ……;
    copy(fdold, fdnew);
    exit(0);
}
copy( old, new)
int old, new;
{
    int count;
    while ( (count = read(old, buffer, sizeof(buffer)))>0)
        write ( new, buffer, count);
}
```



UNIX系统调用

```
#include < fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int   argc;
char *argv[];
{
    int fdold, fdnew;

    ……;
    copy(fdold, fdnew);
    exit(0);
}
copy( old, new)
int old, new;
{
    int count;
    while ( (count = read(old, buffer, sizeof(buffer))>0)
        write ( new, buffer, count);
}
```



UNIX系统调用

```
#include < fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int   argc;
char *argv[];
{
    int fdold, fdnew;

    ……;
    copy(fdold, fdnew);
    exit(0);
}
copy( old, new)
int old, new;
{
    int count;
    while ( (count = read(old, buffer, sizeof(buffer))>0)
        write ( new, buffer, count);
}
```

用户态 User Mode

C语言的标准库函数

read (fd, buf, nbytes)

提前准备

参数

EBX
ECX
EDX
ESI
EDI

系统调用号

EAX

INT 0x80
0x80号中断

现场保护

核心栈

局部变量
Old EBP
返回地址
GS
FS
DS
ES
EBX
ECX 参数
EDX
ESI
EDI
EBP
EAX 返回值
局部变量
Old EBP
EIP
CS
EFLAGS
ESP
SS

核心态 Kernel Mode

系统调用处理

系统调用号和参数来自核心栈

根据系统调用号查表

……

一组系统调用处理子程序

返回值存入核心栈

利用寄存器作为用户态和核心态之间传递信息的通道

UNIX系统调用

以打开文件为例

```
fd = open( “fileName” , mode);
if ( fd < 0 ) {
    printf( “Can not open file!” );
    exit();
}
```

每个系统调用均唯一对应于一个高级语言用户库函数（钩子函数）。运行在用户态。

```
int open(char* pathname, unsigned int mode)
{
    int res;
    __asm__ __volatile__ ("int $0x80" : "=a"(res) : "a"(5), "b"(pathname), "c"(mode));
    if ( res >= 0 )
        return res;
    return -1;
}
```

1. 系统调用号存入EAX

2. 系统调用参数依次写入寄存器EBX，ECX，EDX，ESI和EDI（最多5个）

3. 执行INT 0x80指令，引发80号中断

4. 将EAX寄存器带回的返回值赋给res

UNIX系统调用

应用程序
(用户态运行)

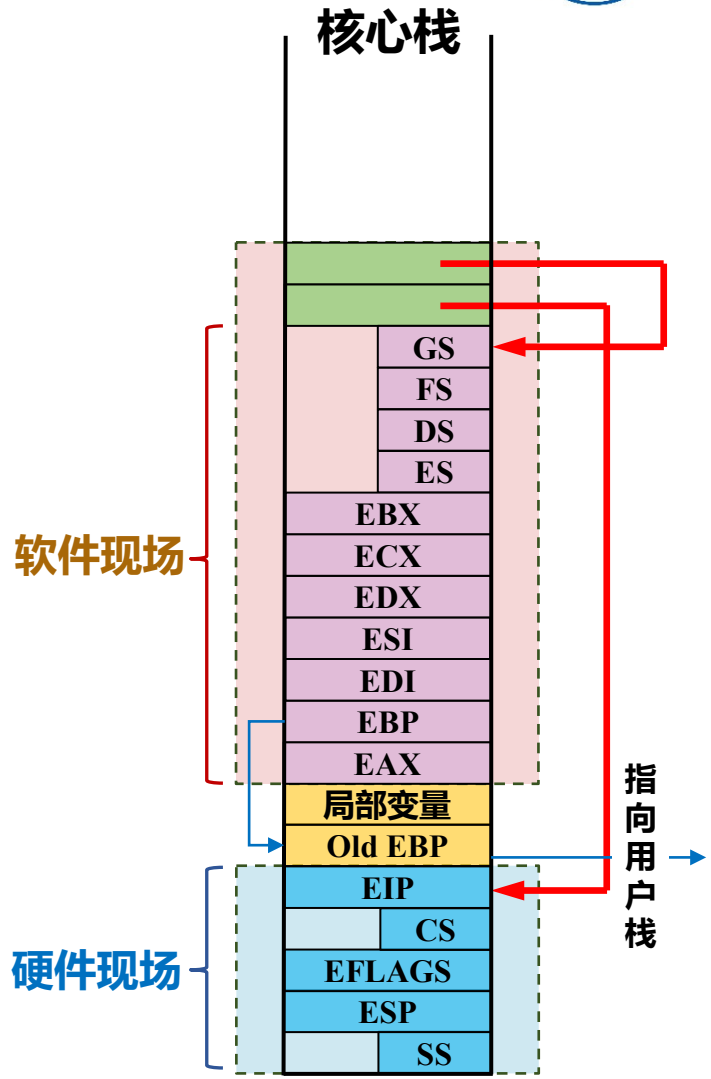
INT 0x80

表示系统调用的中断号(0x80)

处理中断的硬件机构
(执行中断隐指令)

系统调用
入口程序

❶ 软件现场
❷ 转核心态



中断发生时现运行进程在用户态运行

UNIX系统调用

应用程序
(用户态运行)

INT 0x8

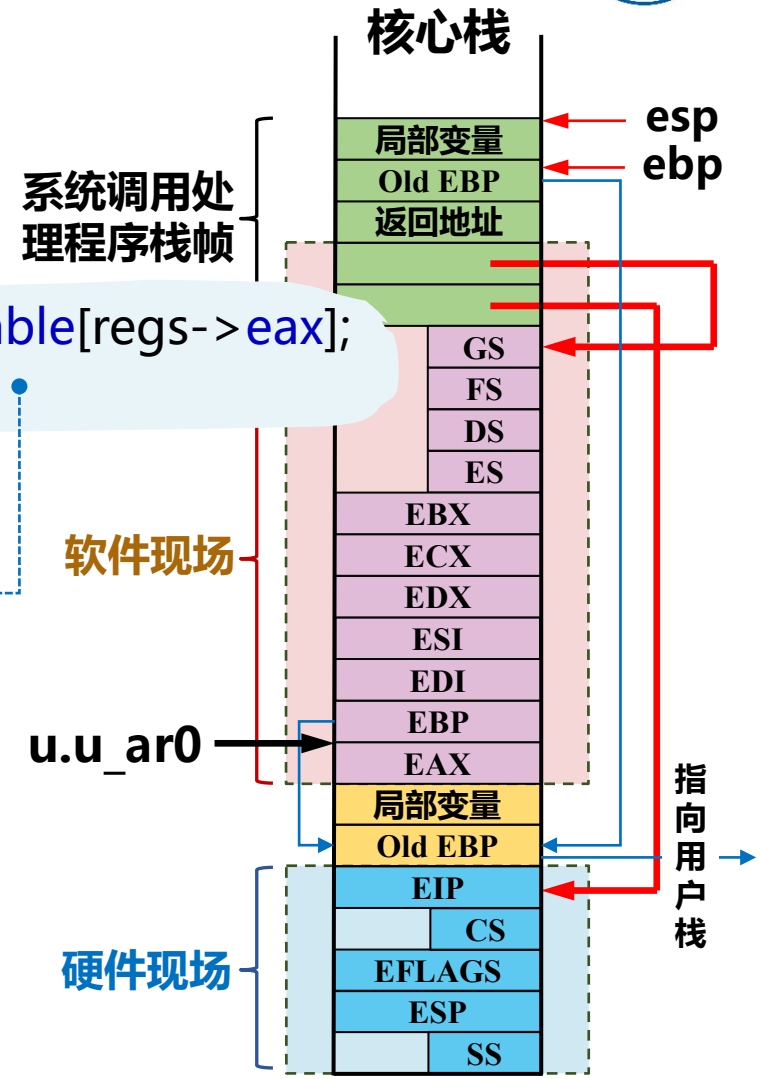
处理中断的硬件执行
(执行中断隐指令)

系统调用
入口程序

u.u_ar0 = &regs->eax;
SystemCallTableEntry *callp = &m_SystemEntranceTable[regs->eax];
callp为指向系统调用处理子程序入口表对应表项的指针

1. 软件现场
2. 转核心态
3. 系统调用处理程序 (Trap)

3.1 根据系统调用号查询系统调用处理子程序入口表，获得
- 参数个数
- 系统调用处理子程序入口



中断发生时现运行进程在用户态运行

UNIX系统调用

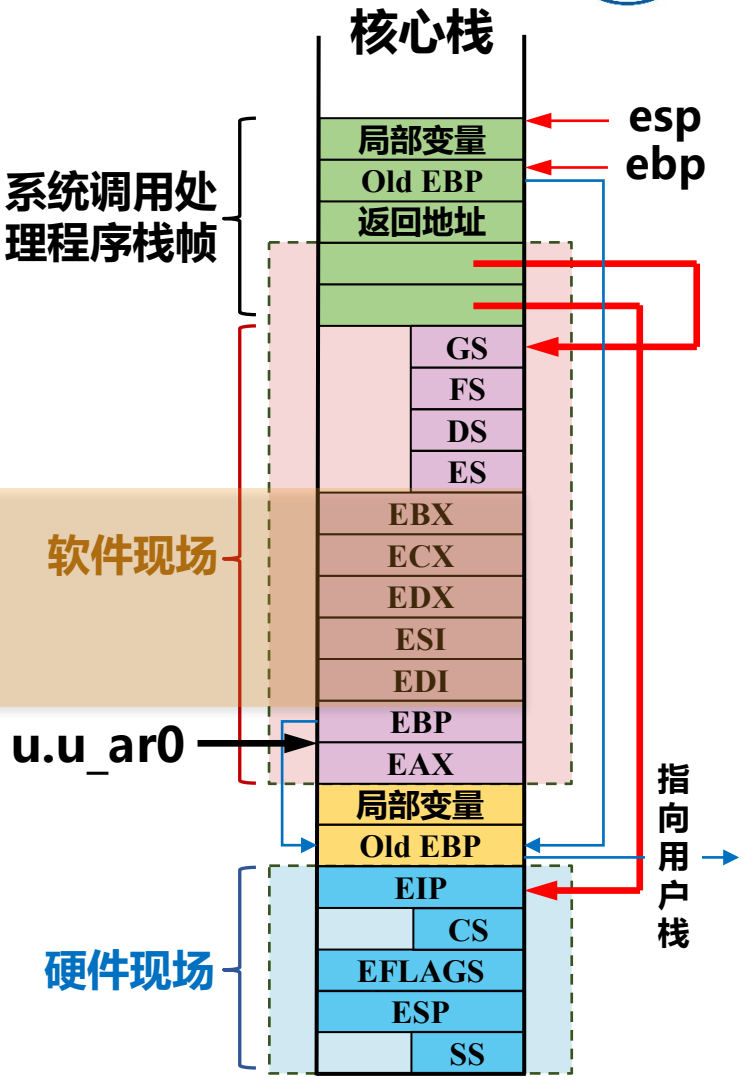
进程的user结构

u.u_arg[5]

根据 参数个数

将所需的参数由栈帧复制到user结构中

```
unsigned int * syscall_arg = (unsigned int *)&regs->ebx;
for( unsigned int i = 0; i < callp->count; i++ )
{
    u.u_arg[i] = (int)(*syscall_arg++);
}
```



中断发生时现运行进程在用户态运行

UNIX系统调用

应用程序
(用户态运行)

INT 0x80

表示系统调用的中断号(0x80)

处理中断的硬件机构
(执行中断隐指令)

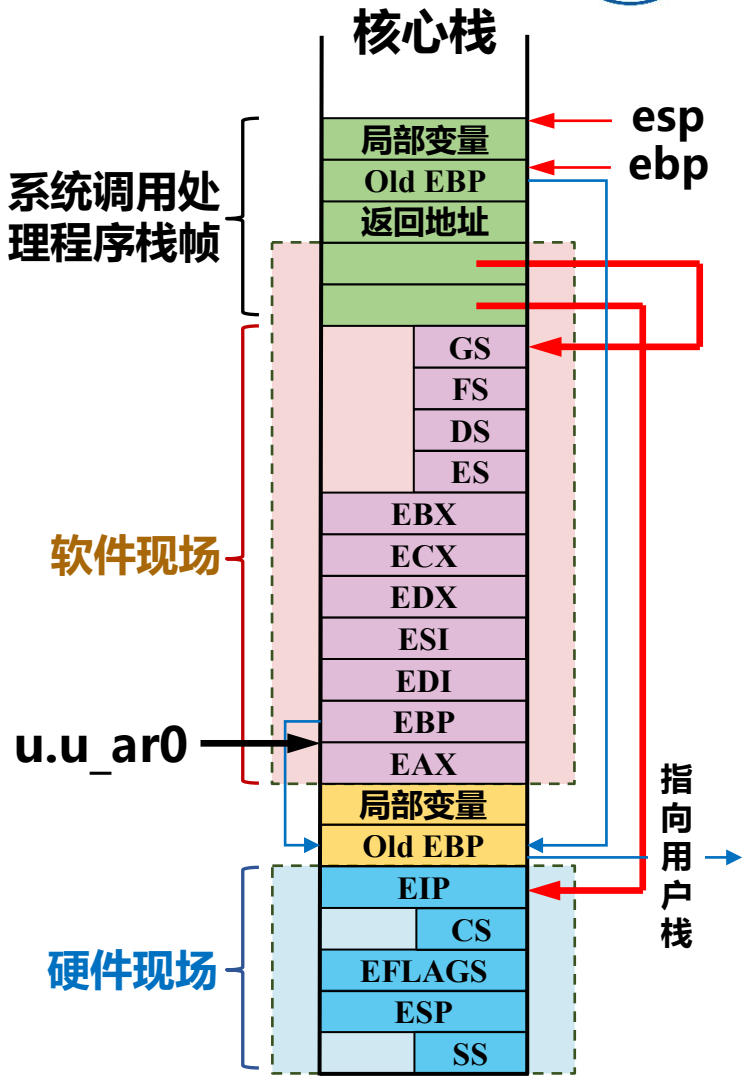
系统调用
入口程序

1. 软件现场
2. 转核心态
3. 系统调用处理程序(Trap)

3.1 根据系统调用号查询系统调用处理子程序入口表，获得

- 参数个数
- 系统调用处理子程序入口

3.2 执行系统调用处理子程序



中断发生时现运行进程在用户态运行

UNIX系统调用

应用程序
(用户态运行)

INT 0x80

表示系统调用的中断号(0x80)

处理中断的硬件机构
(执行中断隐指令)

系统调用入口程序

1. 软件现场
2. 转核心态
3. 系统调用处理程序(Trap)

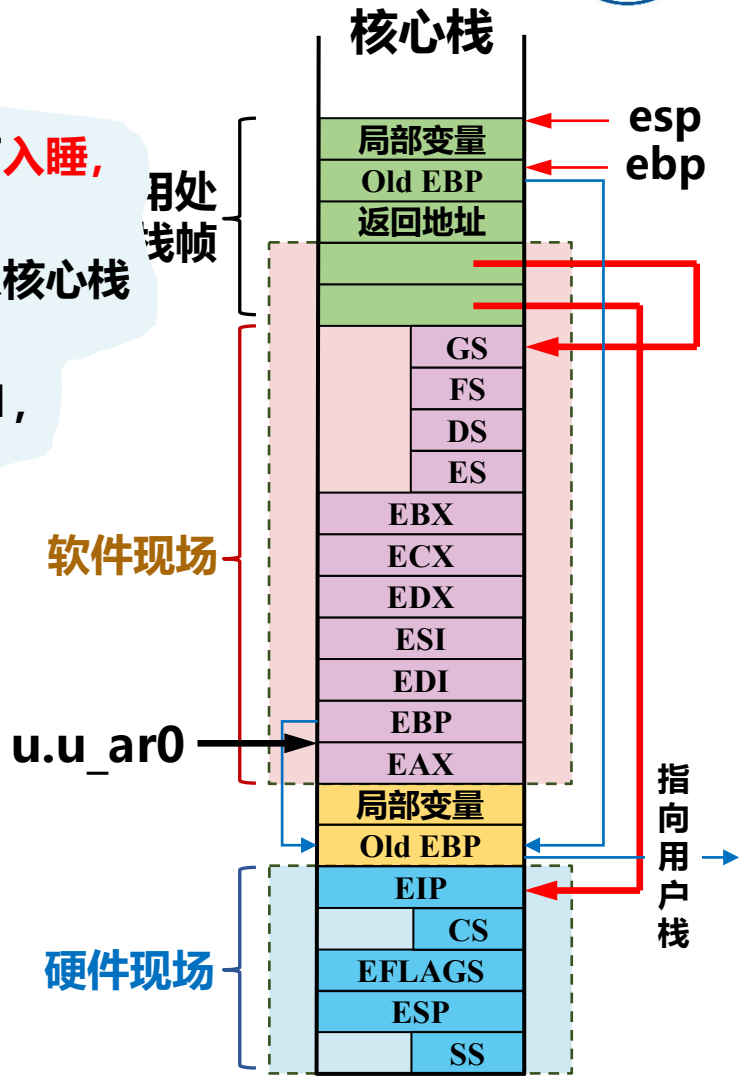
功能虽不同，但遵循以下规则：

1. 参数取自user结构；
2. 进程可能会因为使用系统资源而入睡，下台，等待被唤醒后重新上台；
3. 如果系统调用成功，返回值填入核心栈中保存EAX的位置；
   如果系统调用失败，该位置 = 1，
   置系统调用出错码 u_error

3.1 根据系统调用号查询系统调用处理子程序入口表，获得

- 参数个数
- 系统调用处理子程序入口

3.2 执行系统调用处理子程序



中断发生时现运行进程在用户态运行

UNIX系统调用

应用程序
(用户态运行)

INT 0x80

表示系统调用的中断号(0x80)

处理中断的硬件机构
(执行中断隐指令)

系统调用入口程序

3 系统调用处理程序(Trap)

1. 软件现场
2. 转核心态

功能虽不同，但遵循以下规则：

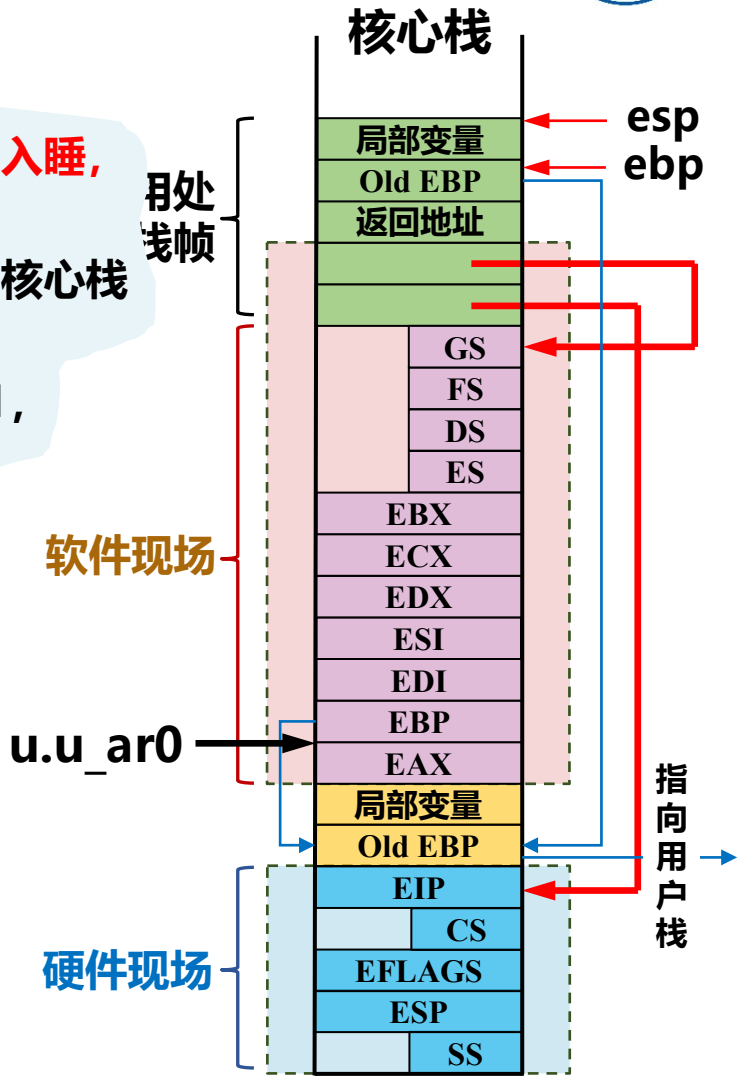
1. 参数取自user结构；
2. 进程可能会因为使用系统资源而入睡，下台，等待被唤醒后重新上台；
3. 如果系统调用成功，返回值填入核心栈中保存EAX的位置；
   如果系统调用失败，该位置 = 1，
   置系统调用出错码 u_error

3.1 根据系统调用号查询系统调用处理子程序入口表，获得

- 参数个数
- 系统调用处理子程序入口

3.2 执行系统调用处理子程序

3.3 重算现运行进程优先数 3



中断发生时现运行进程在用户态运行

UNIX系统调用

应用程序
(用户态运行)

INT 0x80

表示系统调用的中断号(0x80)

处理中断的硬件机构
(执行中断隐指令)

系统调用入口程序

1. 软件现场
2. 转核心态
3. 系统调用处理程序(Trap)
5. 例行调度

进程在这里有可能下台

功能虽不同，但遵循以下规则：

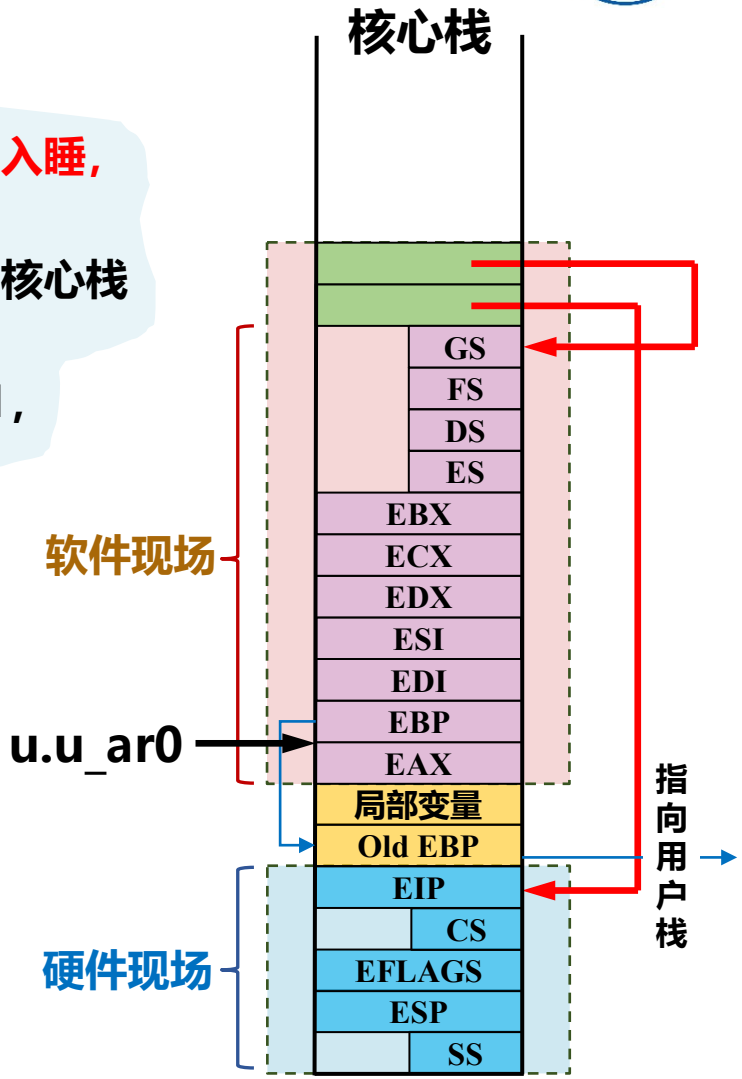
1. 参数取自user结构；
2. 进程可能会因为使用系统资源而入睡，下台，等待被唤醒后重新上台；
3. 如果系统调用成功，返回值填入核心栈中保存EAX的位置；
   如果系统调用失败，该位置 = 1，置系统调用出错码 u_error

3.1 根据系统调用号查询系统调用处理子程序入口表，获得

- 参数个数
- 系统调用处理子程序入口

3.2 执行系统调用处理子程序

3.3 重算现运行进程优先数 3



中断发生时现运行进程在用户态运行

UNIX系统调用

应用程序
(用户态运行)

INT 0x80

表示系统调用的中断号(0x80)

处理中断的硬件机构
(执行中断隐指令)

系统调用入口程序

1. 软件现场
2. 转核心态
3. 系统调用处理程序(Trap)
5. 例行调度
6. 恢复返回

功能虽不同，但遵循以下规则：

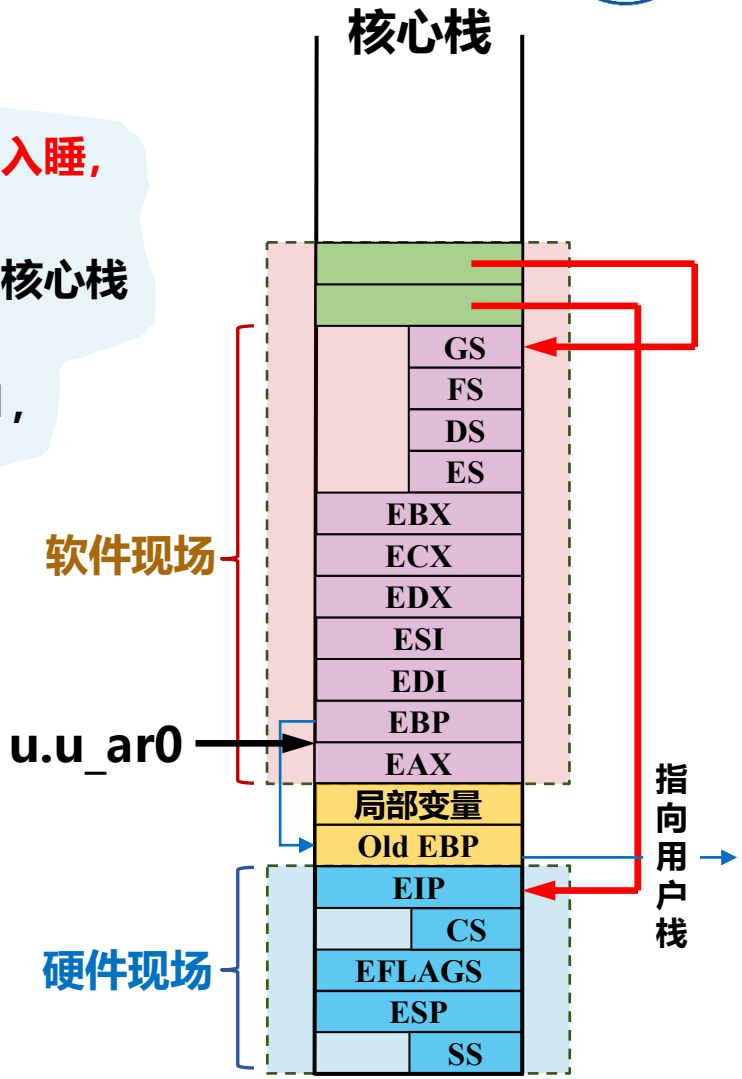
1. 参数取自user结构；
2. 进程可能会因为使用系统资源而入睡，下台，等待被唤醒后重新上台；
3. 如果系统调用成功，返回值填入核心栈中保存EAX的位置；
   如果系统调用失败，该位置 = 1，
   置系统调用出错码 u_error

3.1 根据系统调用号查询系统调用处理子程序入口表，获得

- 参数个数
- 系统调用处理子程序入口

3.2 执行系统调用处理子程序

3.3 重算现运行进程优先数 3



中断发生时现运行进程在用户态运行

UNIX系统调用

应用程序
(用户态运行)

INT 0x80

表示系统调用的中断号(0x80)

处理中断的硬件机构
(执行中断隐指令)

系统调用入口程序

3 系统调用处理程序(Trap)

1. 软件现场
2. 转核心态
5. 例行调度
6. 恢复返回

功能虽不同，但遵循以下规则：

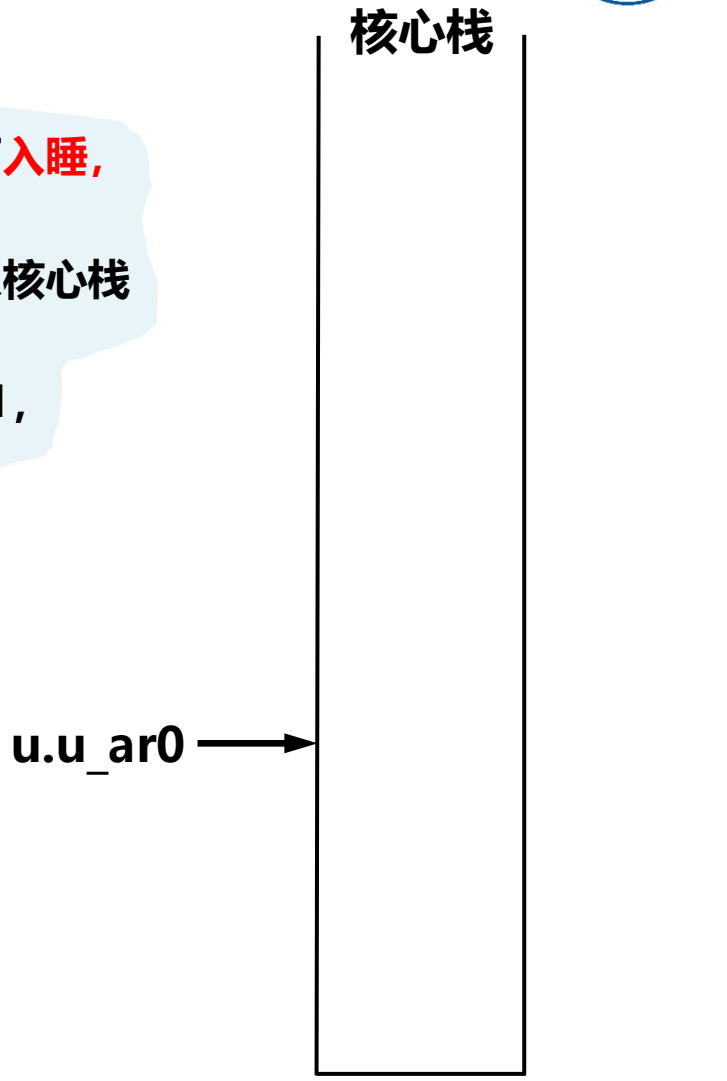
1. 参数取自user结构；
2. 进程可能会因为使用系统资源而入睡，下台，等待被唤醒后重新上台；
3. 如果系统调用成功，返回值填入核心栈中保存EAX的位置；
   如果系统调用失败，该位置 = 1，
   置系统调用出错码 u_error

3.1 根据系统调用号查询系统调用处理子程序入口表，获得

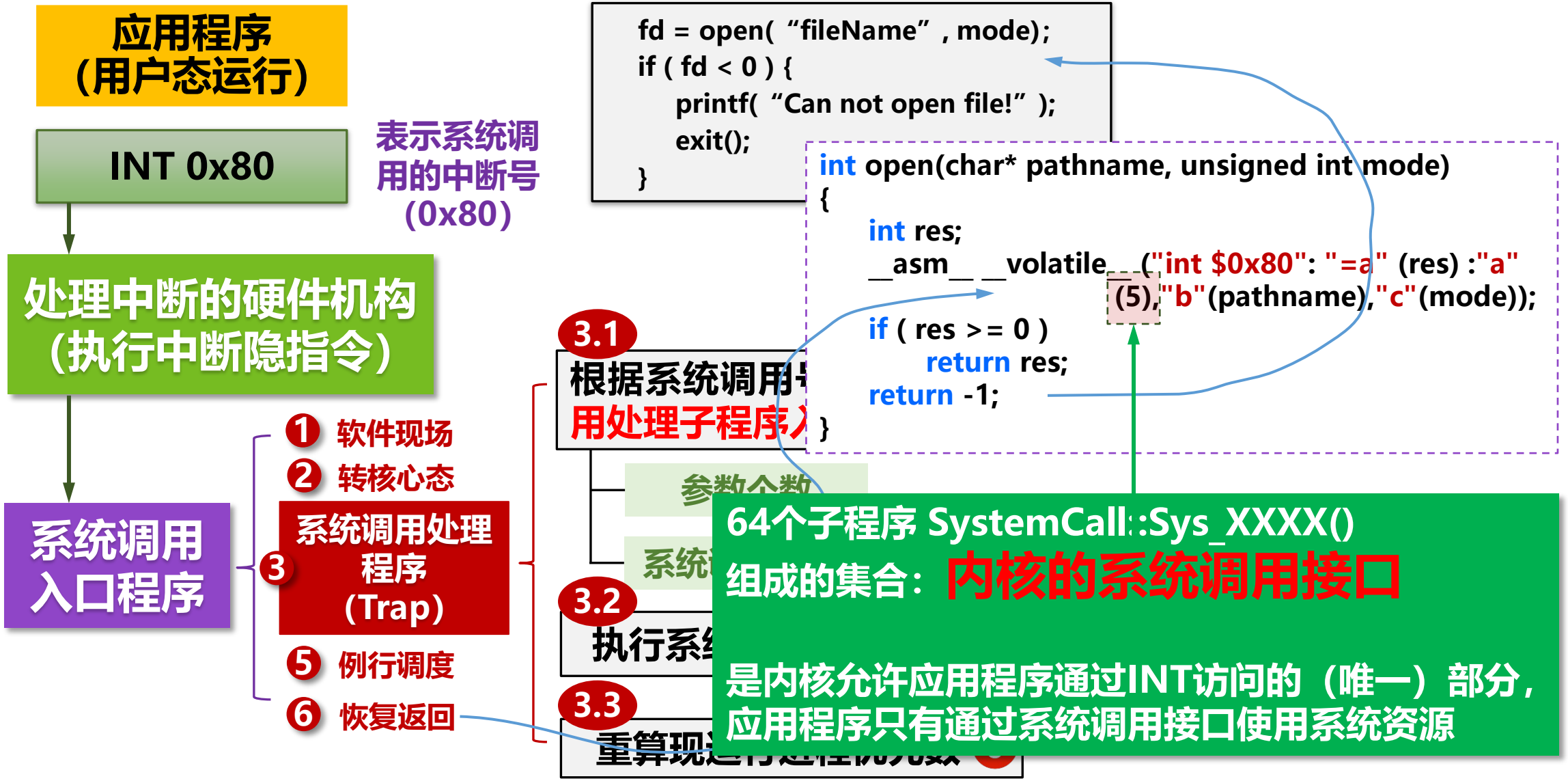
- 参数个数
- 系统调用处理子程序入口

3.2 执行系统调用处理子程序

3.3 重算现运行进程优先数 3



UNIX系统调用

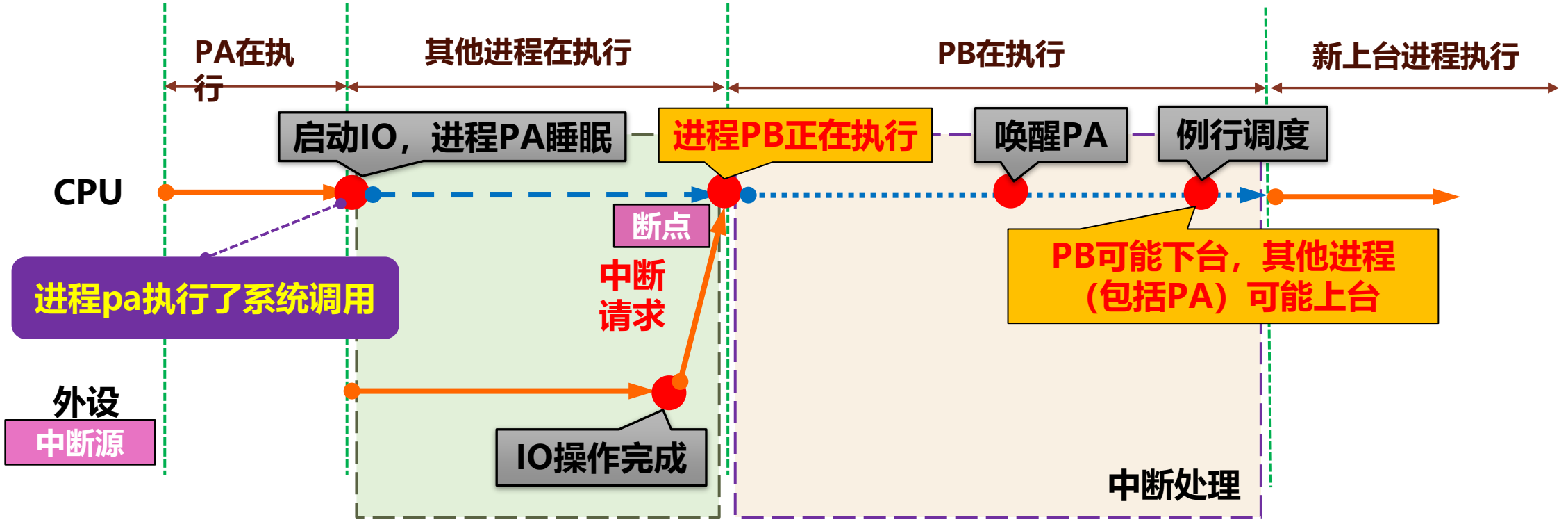

应用程序
(用户态运行)
INT 0x80
表示系统调用的中断号(0x80)
处理中断的硬件机构
(执行中断隐指令)
系统调用入口程序
1 软件现场
2 转核心态
3 系统调用处理程序(Trap)
5 例行调度
6 恢复返回
3.1
根据系统调用号
用处理子程序入
参数个数
系统
3.2
执行系统
3.3
重算现
fd = open(“fileName” , mode);
if (fd < 0) {
printf(“Can not open file!”);
exit();
}
int open(char* pathname, unsigned int mode)
{
int res;
__asm__ __volatile__ ("int $0x80": "=a" (res) :"a" (5),"b"(pathname),"c"(mode));
if (res >= 0)
return res;
return -1;
}
64个子程序 SystemCall::Sys_XXXX()
组成的集合：内核的系统调用接口
是内核允许应用程序通过INT访问的（唯一）部分，
应用程序只有通过系统调用接口使用系统资源

UNIX系统调用

系统调用和一般函数调用的区别

项目	一般子程序调用	系统调用
调用者	应用程序或内核中的任意子程序	应用程序中的某个子程序
被调用者	同一个程序中的另一个子程序	操作系统内部供应用程序调用的子程序
调用者和被调用者之间的关系	链接在同一个可执行文件中	分别链接在两个不同的可执行文件中，被调用者一定是操作系统内核
引发调用的事件	call指令	INT 指令
被调用者栈帧位置	当前堆栈的栈顶	核心栈
参数如何传递	调用者向当前堆栈push实参	调用者和被调用者的栈帧一定分别位于同一个进程的用户栈和核心栈。参数传递需借助寄存器和user结构
调用者如何得到返回值	CPU内的寄存器	将系统调用的返回值写入核心栈底保存EAX寄存器的位置，待中断返回用户态后该值将pop回该寄存器，作为系统调用的返回值

UNIX系统调用

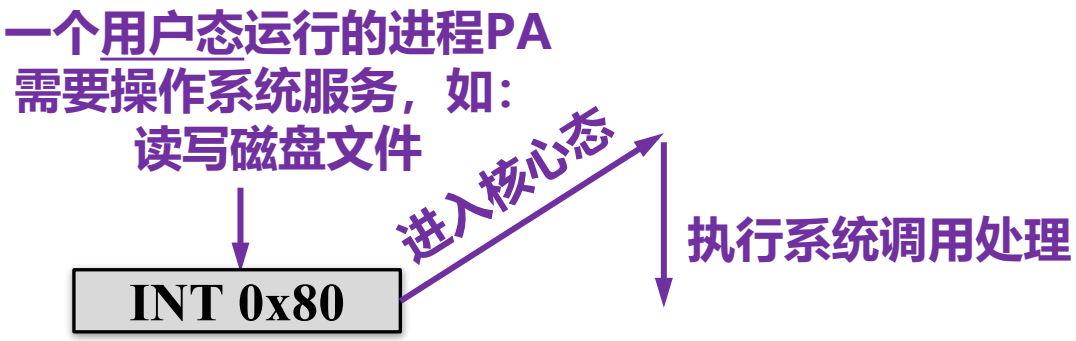


1. 保证了CPU和设备之间的并行操作

2. 提供了进程执行内核代码的机会

3. 多道程序并发的硬件基础

UNIX系统调用



UNIX系统调用

一个用户态运行的进程PA
需要操作系统服务，如：
读写磁盘文件

INT 0x80

进入核心态

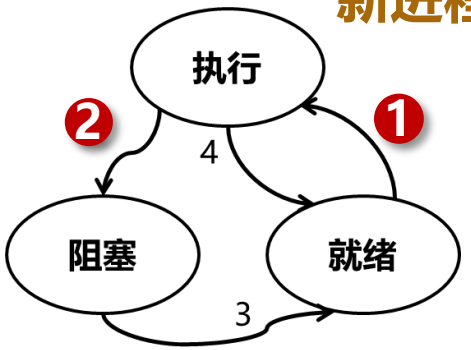
执行系统调用处理

如果系统调用中使用外设，进程调用Sleep入睡

下台1

Swtch进程切换

进程PA下台，执行 → 阻塞 ❷
新进程上台，就绪 → 执行 ❶



抢占式/剥夺式调度
现运行进程PCB中的进程状态
“执行” → “就绪”

非抢占式/进程主动放弃
现运行进程PCB中的进程状态
“执行” → “阻塞”

UNIX系统调用

一个用户态运行的进程PA
需要操作系统服务，如：
读写磁盘文件

INT 0x80

进入核心态

执行系统调用处理

如果系统调用中使用外设，进程调用Sleep入睡

下台1

Swtch进程切换

被唤醒

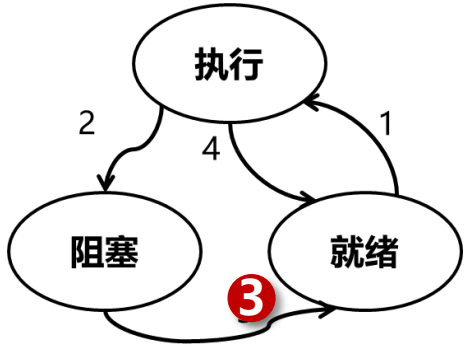
进程PB在执行

pb进入核心态

执行中断处理过程

设备中断

阻塞 → 就绪 ❸
等待下一次上台的机会



UNIX系统调用

一个用户态运行的进程PA
需要操作系统服务，如：
读写磁盘文件

INT 0x80

进入核心态

执行系统调用处理

如果系统调用中使用外设，进程调用Sleep入睡

下台1

Swtch进程切换

被唤醒

上台1

Swtch选中上台

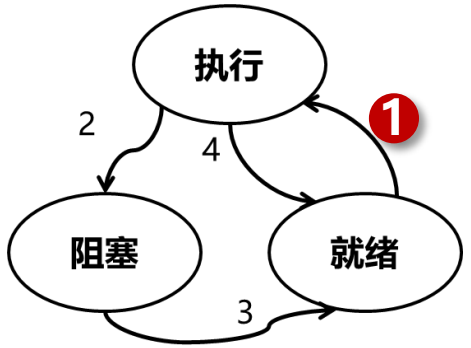
（完成系统调用的剩余代码）

进程PB在执行

pb进入核心态

执行中断处理过程

设备中断



就绪 → 执行 ❶

UNIX系统调用

一个用户态运行的进程PA
需要操作系统服务，如：
读写磁盘文件

INT 0x80

进入核心态

执行系统调用处理

如果系统调用中使用外设，进程调用Sleep入睡

下台1

Swtch进程切换

被唤醒

上台1

Swtch选中上台

（完成系统调用的剩余代码）

Trap末尾重算进程优先数

因为系统调用
前为用户态

RunRun

下台2

Swtch进程切换

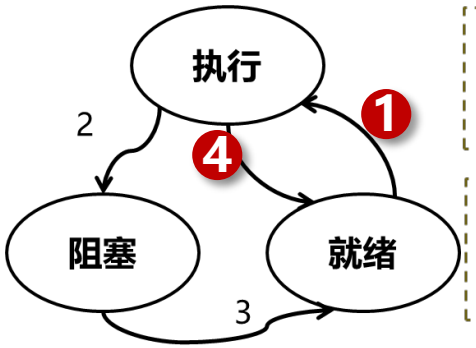
进程PA下台，执行 → 就绪 ❹
新进程上台，就绪 → 执行 ❶

进程PB
在执行

pb进入核心态

执行中断处理过程

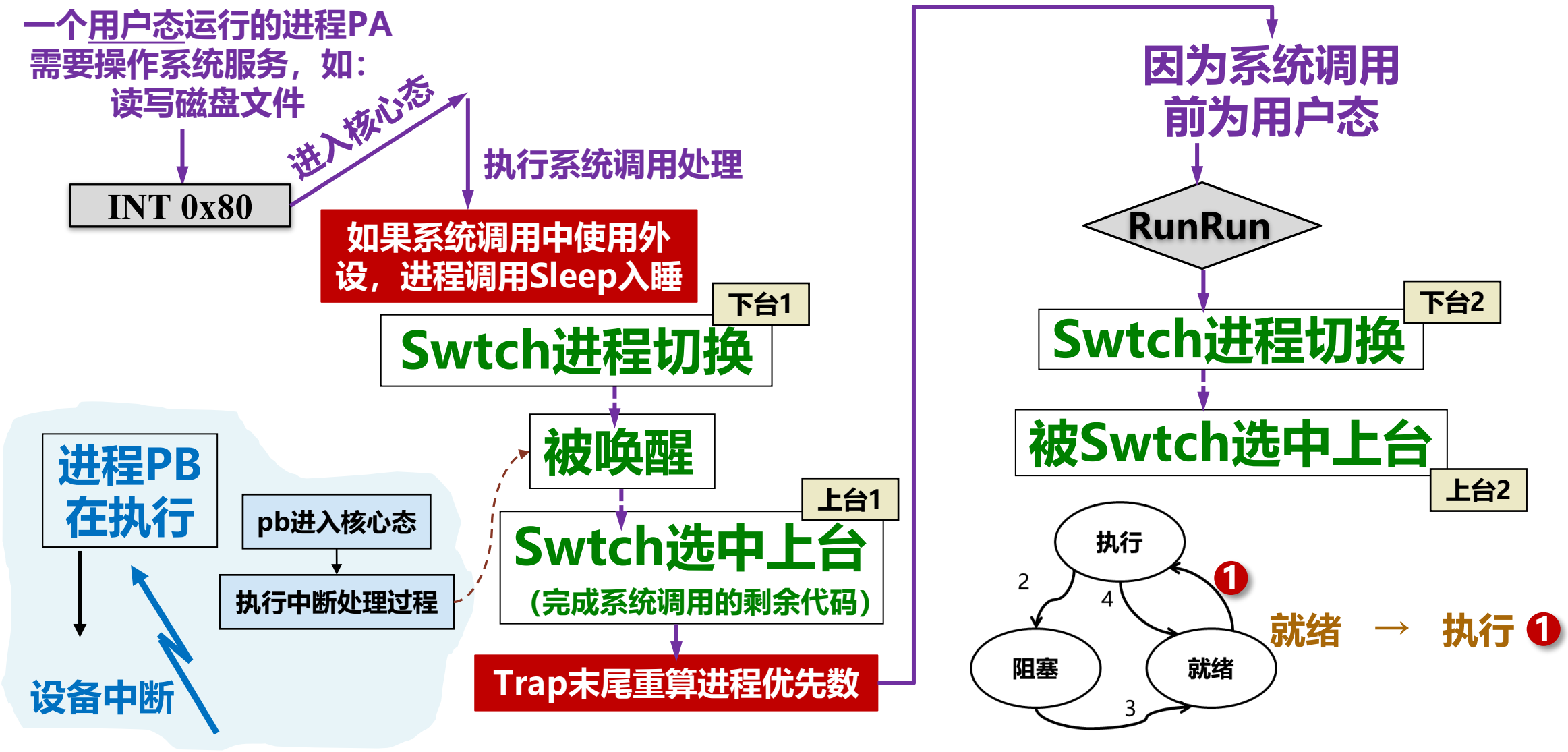
设备中断



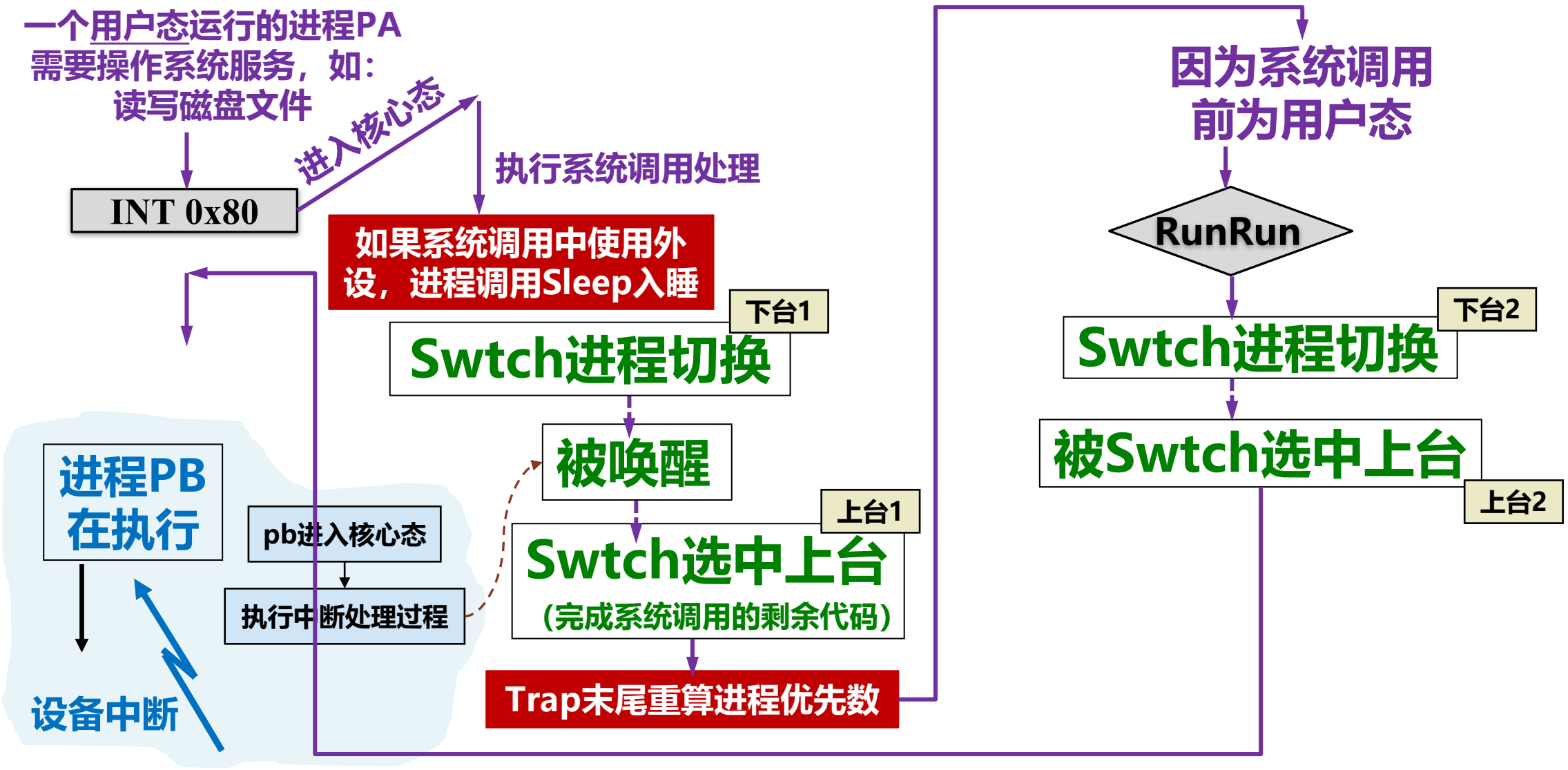
抢占式/剥夺式调度
现运行进程PCB中的进程状态
“执行” → “就绪”

非抢占式/进程主动放弃
现运行进程PCB中的进程状态
“执行” → “阻塞”

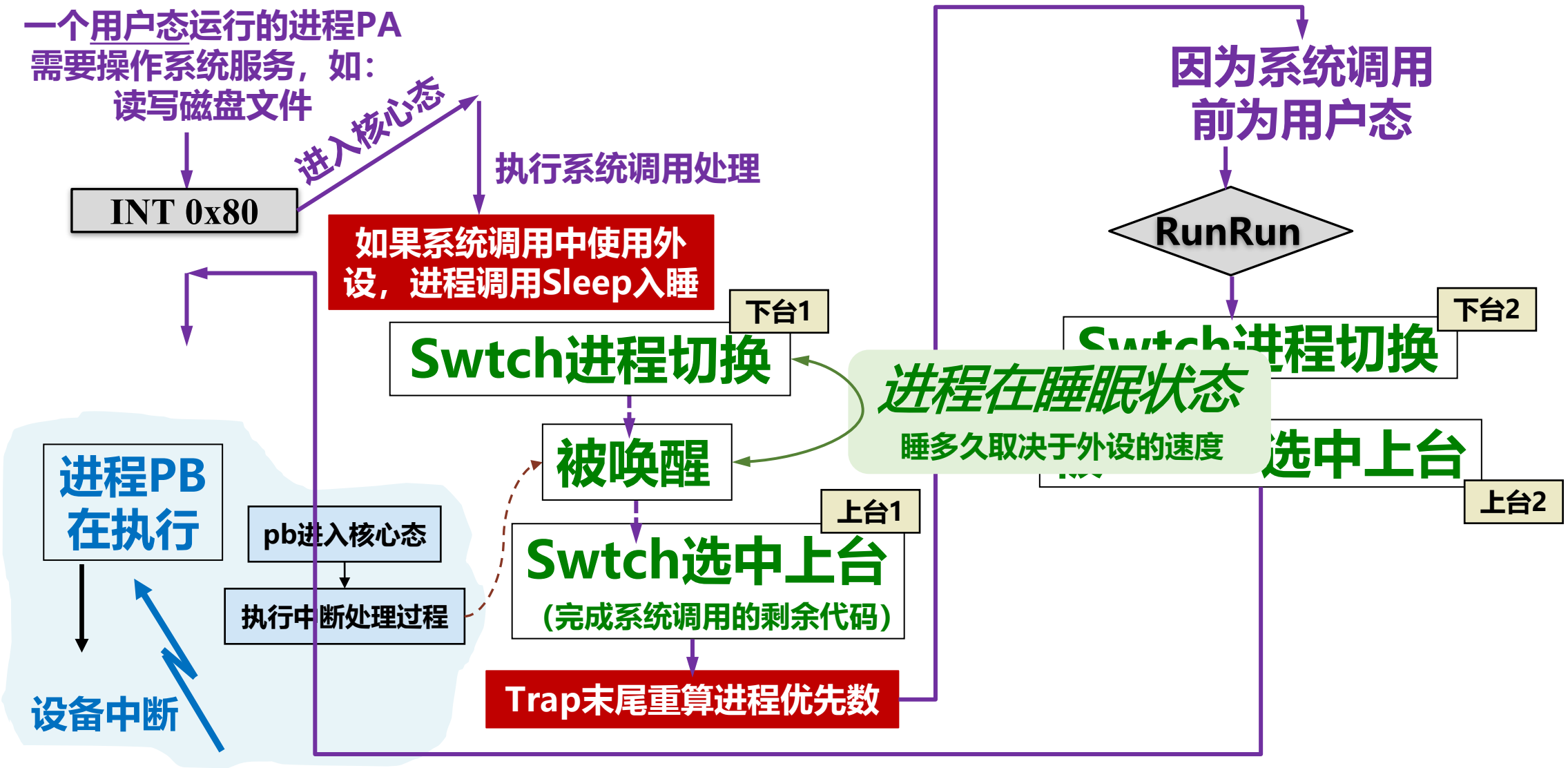
UNIX系统调用


一个用户态运行的进程PA
需要操作系统服务，如：
读写磁盘文件
INT 0x80
进入核心态
执行系统调用处理
如果系统调用中使用外设，进程调用Sleep入睡
下台1
Swtch进程切换
被唤醒
上台1
Swtch选中上台
(完成系统调用的剩余代码)
Trap末尾重算进程优先数
因为系统调用前为用户态
RunRun
下台2
Swtch进程切换
被Swtch选中上台
上台2
进程PB在执行
pb进入核心态
执行中断处理过程
设备中断
执行
阻塞
就绪
1
2
3
4
就绪 → 执行 1

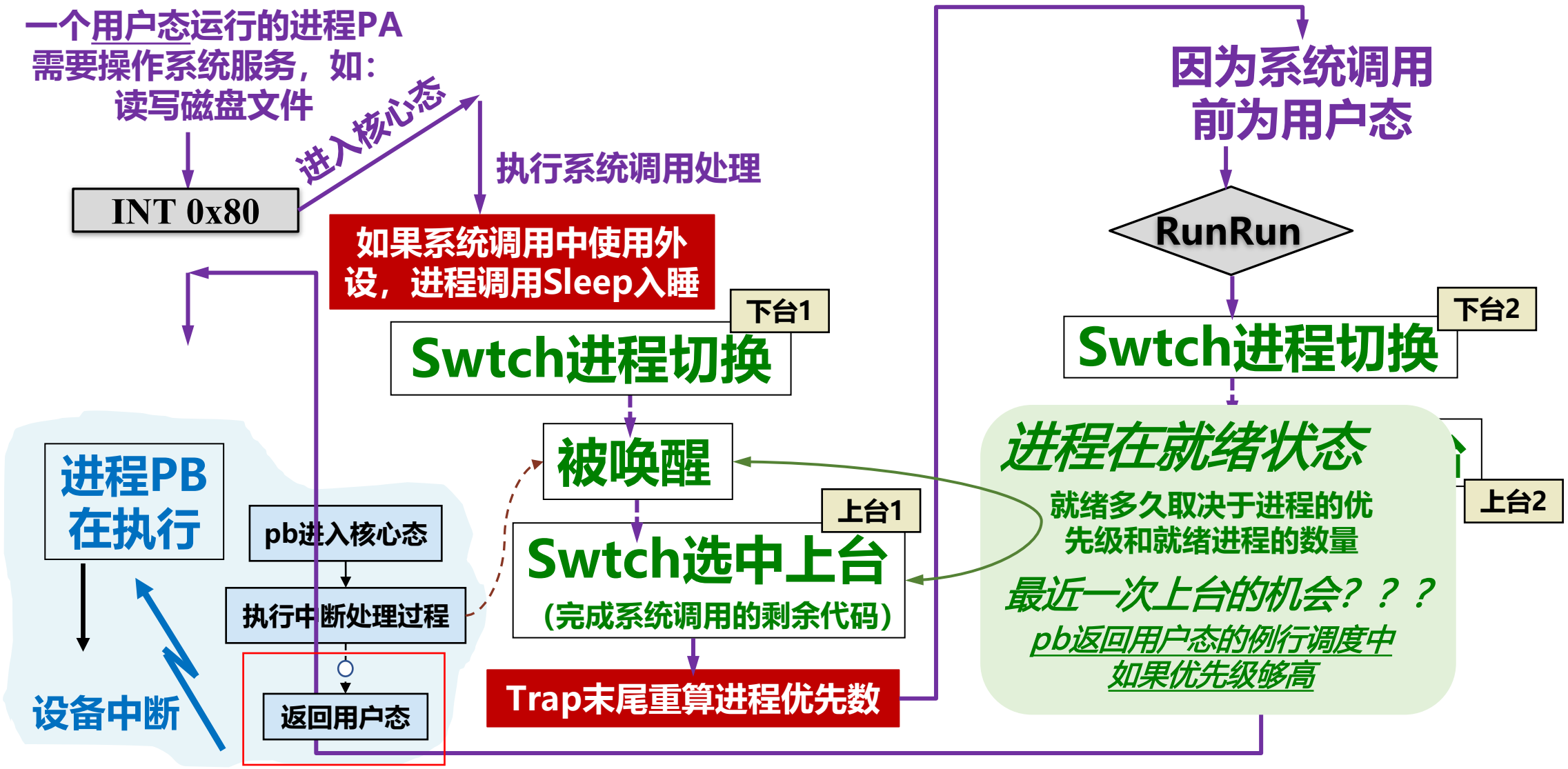
UNIX系统调用


一个用户态运行的进程PA
需要操作系统服务，如：
读写磁盘文件
进入核心态
INT 0x80
执行系统调用处理
如果系统调用中使用外设，进程调用Sleep入睡
下台1
Swtch进程切换
被唤醒
上台1
Swtch选中上台
（完成系统调用的剩余代码）
Trap末尾重算进程优先数
因为系统调用前为用户态
RunRun
下台2
Swtch进程切换
被Swtch选中上台
上台2
进程PB在执行
pb进入核心态
执行中断处理过程
设备中断

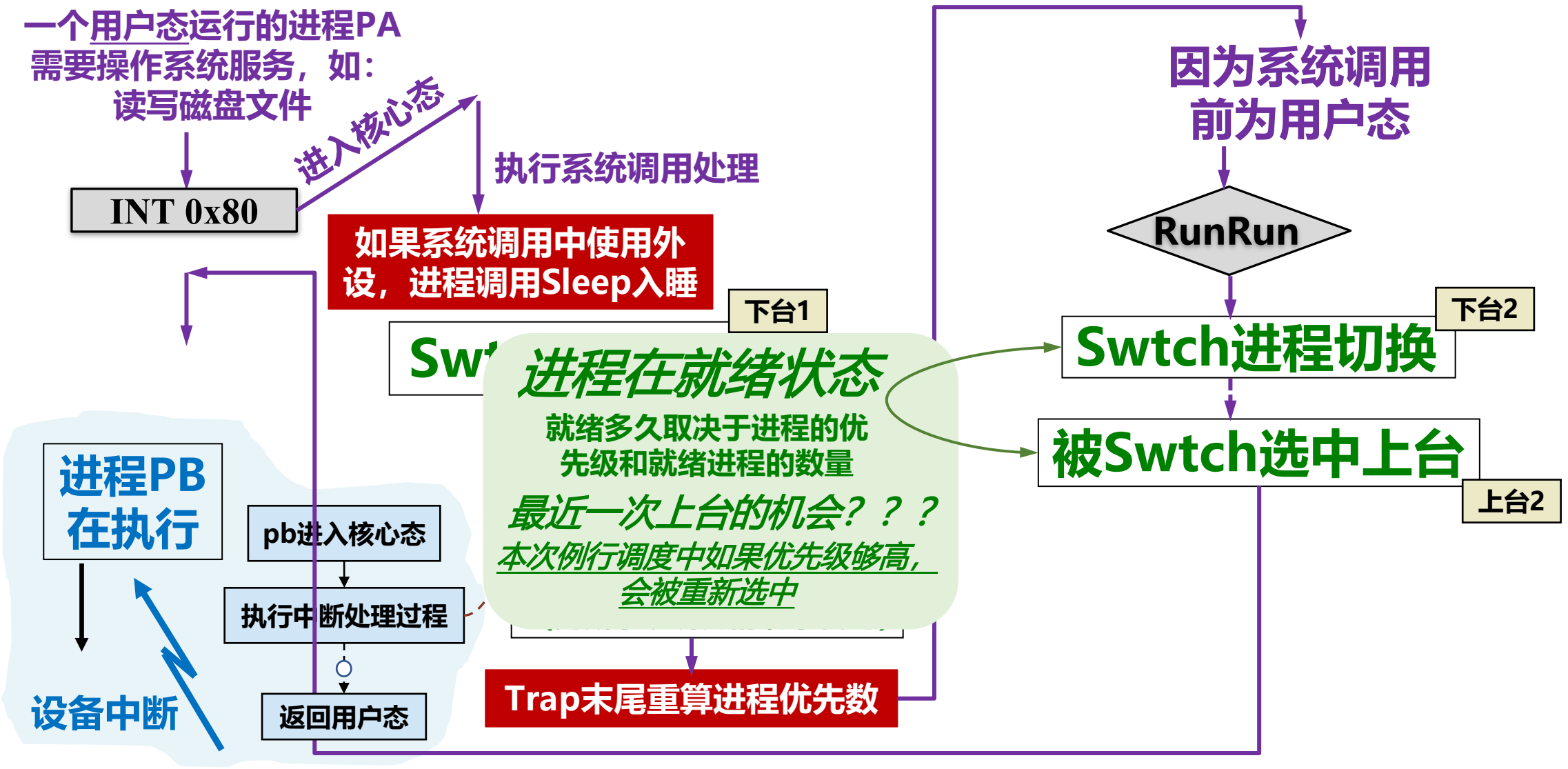
UNIX系统调用


一个用户态运行的进程PA
需要操作系统服务，如：
读写磁盘文件
进入核心态
INT 0x80
执行系统调用处理
如果系统调用中使用外设，进程调用Sleep入睡
下台1
Swtch进程切换
被唤醒
上台1
Swtch选中上台
（完成系统调用的剩余代码）
Trap末尾重算进程优先数
因为系统调用前为用户态
RunRun
下台2
Swtch进程切换
选中上台
上台2
进程在睡眠状态
睡多久取决于外设的速度
进程PB在执行
pb进入核心态
执行中断处理过程
设备中断

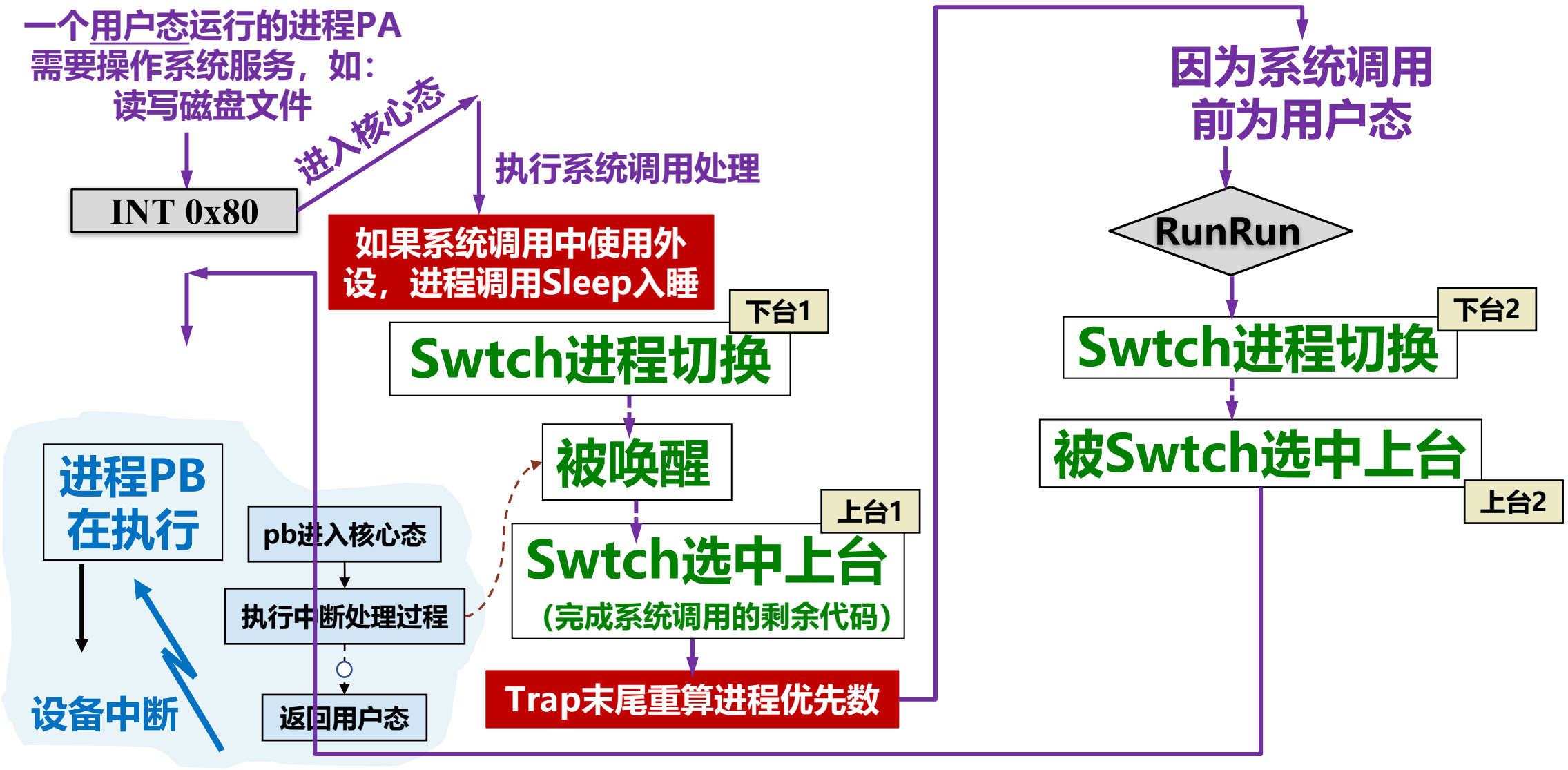
UNIX系统调用


一个用户态运行的进程PA
需要操作系统服务，如：
读写磁盘文件
进入核心态
INT 0x80
执行系统调用处理
如果系统调用中使用外设，进程调用Sleep入睡
下台1
Swtch进程切换
被唤醒
上台1
Swtch选中上台
（完成系统调用的剩余代码）
Trap末尾重算进程优先数
因为系统调用前为用户态
RunRun
下台2
Swtch进程切换
进程在就绪状态
就绪多久取决于进程的优先级和就绪进程的数量
最近一次上台的机会？？？
pb返回用户态的例行调度中如果优先级够高
上台2
进程PB在执行
pb进入核心态
执行中断处理过程
返回用户态
设备中断

UNIX系统调用

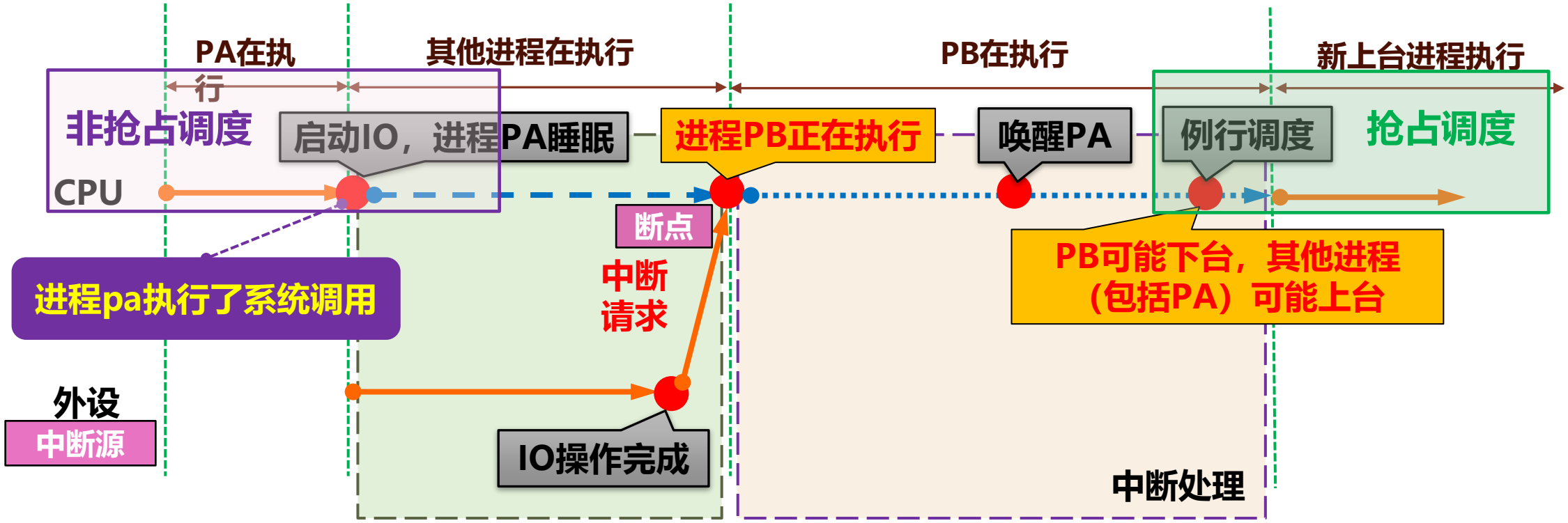

一个用户态运行的进程PA
需要操作系统服务，如：
读写磁盘文件
进入核心态
INT 0x80
执行系统调用处理
如果系统调用中使用外设，进程调用Sleep入睡
下台1
Sw
进程在就绪状态
就绪多久取决于进程的优先级和就绪进程的数量
最近一次上台的机会？？？
本次例行调度中如果优先级够高，会被重新选中
进程PB在执行
pb进入核心态
执行中断处理过程
返回用户态
设备中断
Trap末尾重算进程优先数
因为系统调用前为用户态
RunRun
Swtch进程切换
下台2
被Swtch选中上台
上台2

UNIX系统调用


一个用户态运行的进程PA
需要操作系统服务，如：
读写磁盘文件
进入核心态
INT 0x80
执行系统调用处理
如果系统调用中使用外设，进程调用Sleep入睡
下台1
Swtch进程切换
被唤醒
上台1
Swtch选中上台
（完成系统调用的剩余代码）
Trap末尾重算进程优先数
因为系统调用前为用户态
RunRun
下台2
Swtch进程切换
被Swtch选中上台
上台2
进程PB在执行
pb进入核心态
执行中断处理过程
返回用户态
设备中断

UNIX系统调用

系统调用与中断的关系



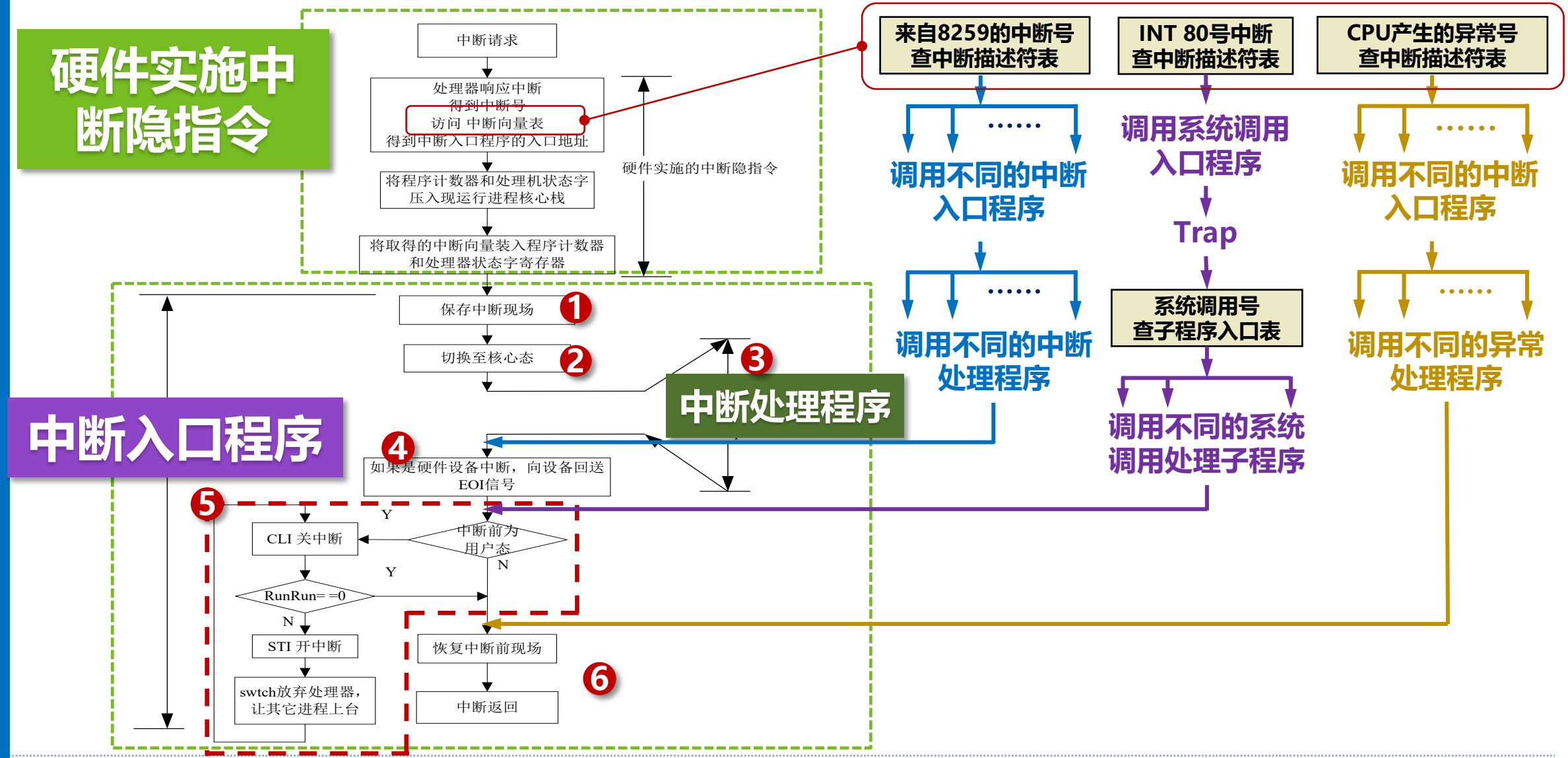
1. 保证了CPU和设备之间的并行操作

2. 提供了进程执行内核代码的机会

3. 多道程序并发的硬件基础

总结

关于中断


硬件实施中断隐指令
中断请求
处理器响应中断
得到中断号
访问 中断向量表
得到中断入口程序的入口地址
将程序计数器和处理机状态字压入现运行进程核心栈
将取得的中断向量装入程序计数器和处理器状态字寄存器
硬件实施的中断隐指令
中断入口程序
保存中断现场
1
切换至核心态
2
3
中断处理程序
4
如果是硬件设备中断，向设备回送EOI信号
5
Y
CLI 关中断
中断前为用户态
N
Y
RunRun= =0
N
STI 开中断
swtch放弃处理器，让其它进程上台
恢复中断前现场
6
中断返回
来自8259的中断号查中断描述符表
INT 80号中断查中断描述符表
CPU产生的异常号查中断描述符表
调用不同的中断入口程序
调用系统调用入口程序
Trap
调用不同的中断入口程序
调用不同的中断处理程序
系统调用号查子程序入口表
调用不同的异常处理程序
调用不同的系统调用处理子程序

本节小结

1　UNIX系统调用的详细流程

2　UNIX系统调用与中断的关系

3　三次重算优先数的时机

阅读教材：111页 ~ 116页

E11：进程管理（UNIX系统调用）

P04：在UNIX V6++中添加新的系统调用