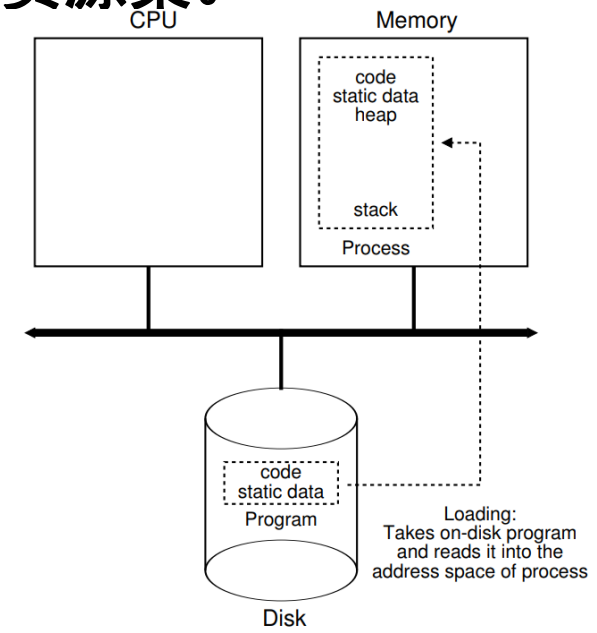
第3章 进程描述和控制 ★★★★★

- 主要内容
  - 3.1 什么是进程
  - 3.2 进程状态
  - 3.3 进程描述
  - 3.4 进程控制
  - 3.5 操作系统的执行
  - 3.6 UNIX SVR4进程管理

3.1 什么是进程 ★★★★★

- 进程定义
  - 一个正在执行中的程序。
  - 一个正在计算机上执行的程序实例。
  - 能分配给处理器并由处理器执行的实体。
  - 一个具有以下特征的活动单元：一组指令序列的执行、一个当前状态和相关的系统资源集。
- 构成进程的基本元素
  - 程序代码
  - 数据集
- 进程控制块



进程控制块

进程控制块

属性	含义
标识符	跟这个进程相关的唯一标识符，用来区别其他进程。
状态	进程当前的状态：运行态、就绪态、阻塞态等。
优先级	相对于其他进程的优先级。
程序计数器	程序中即将被执行的执行的下一条指令的地址。
内存指针	包括程序代码和进程相关数据的指针，还有和其他进程共享内存块的指针。
上下文数据	进程执行时处理器的寄存器中的数据。
I/O状态信息	包括显式的I/O请求、分配给进程的I/O设备和被进程使用的文件列表等。
记账信息	可能包括处理器时间总和、使用的时钟数总和、时间限制、记账号等。

The xv6 Proc Structure

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If !zero, sleeping on chan
  int killed;                 // If !zero, has been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

提问

1. 判断：程序就是进程。

A. 正确　　B. 错误

2. 操作系统用（）来管理和控制进程所需的内部数据。

A. 执行上下文　B. IR寄存器　　C. PC　　D. 数据寄存器

3. 进程是程序的一次执行，（）是进程存在的唯一标志。

A. 程序　　B. 数据　　C. 进程控制块　D. IR寄存器

4. 进程被中断时，操作系统会把上下文数据保存到其进程控制块中的相应位置，进程状态随后被改为其他值。

A. 正确　　B. 错误

3.2 进程状态

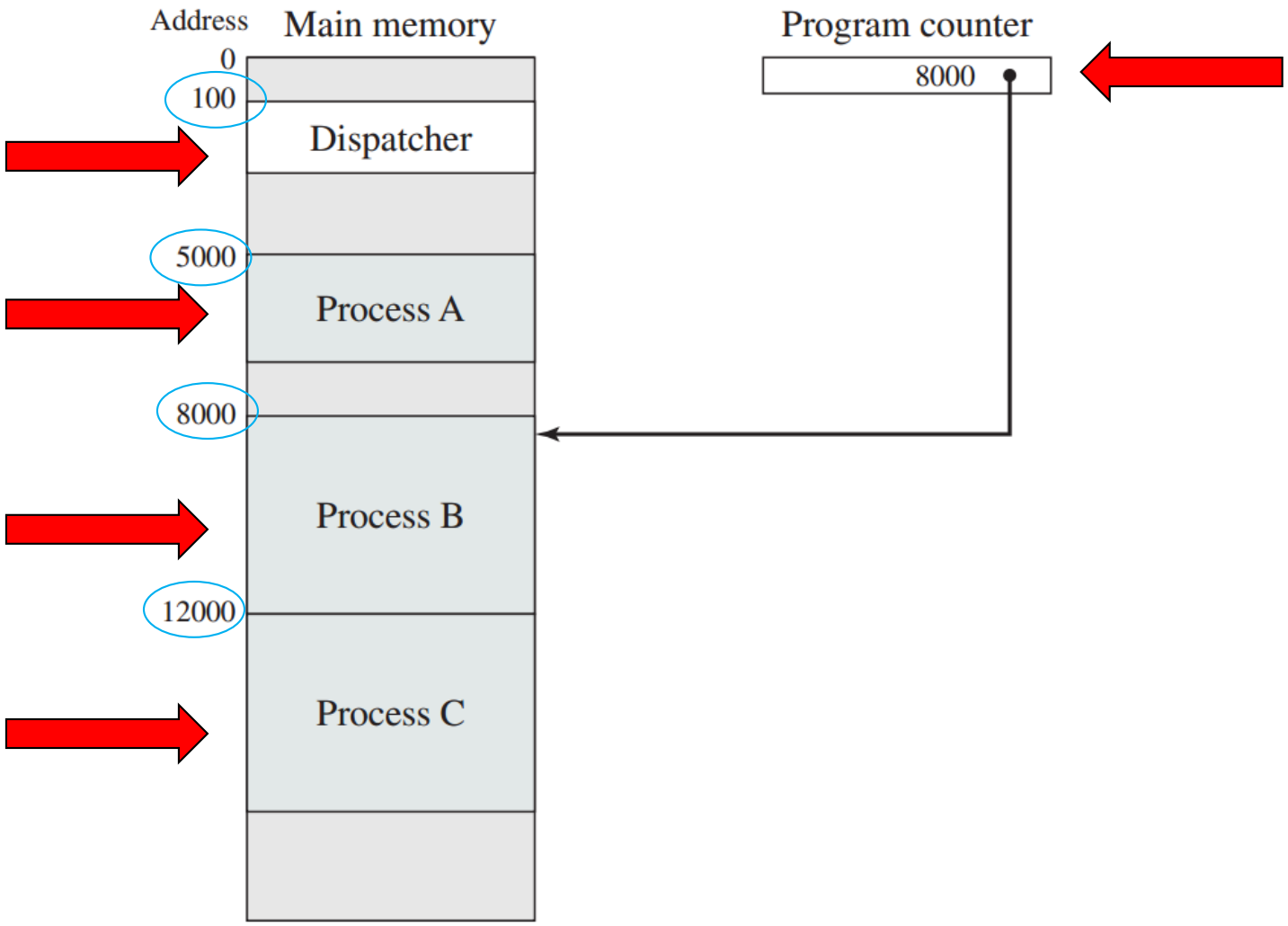


Figure 3.2 Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13

3.2 进程状态

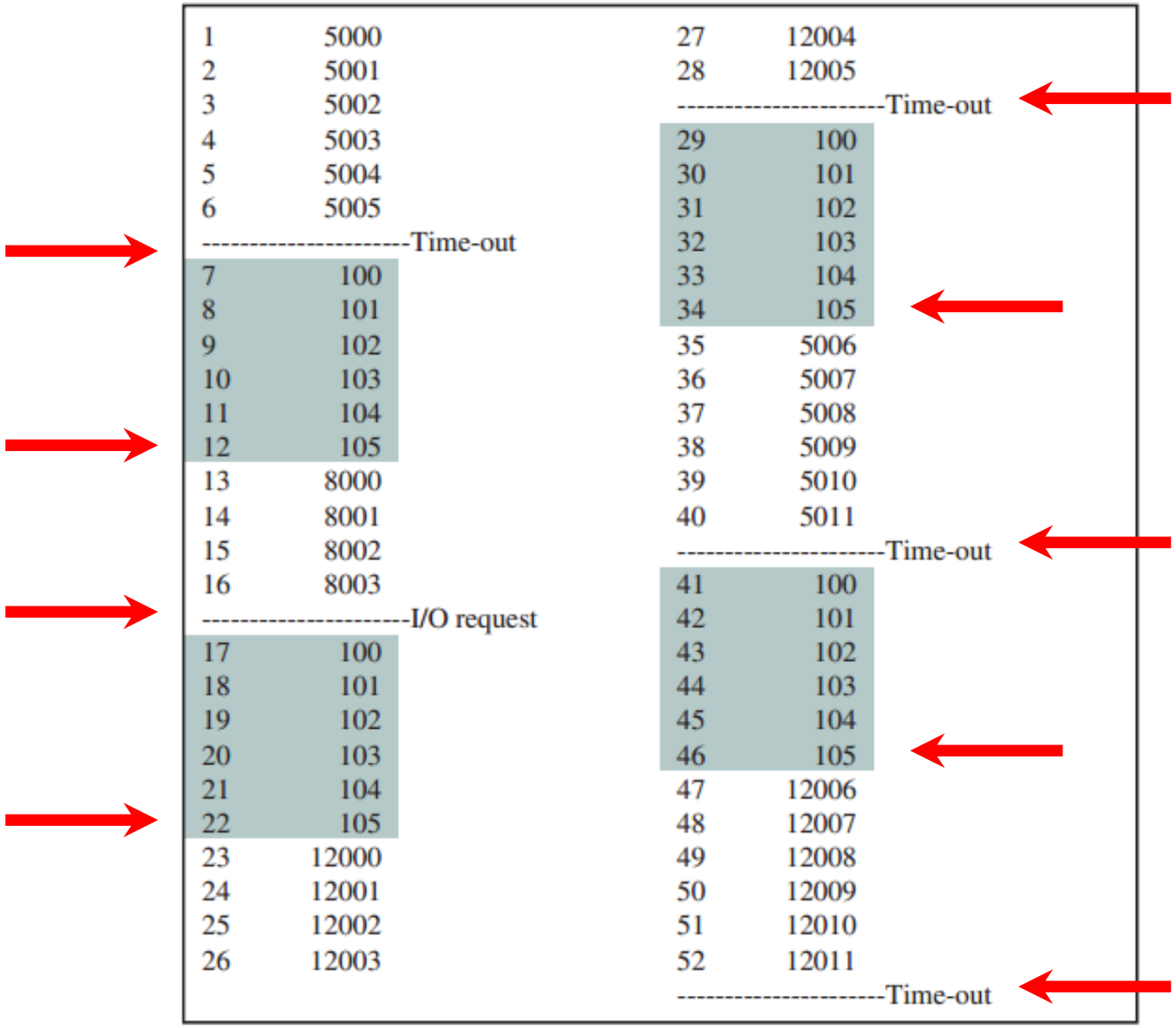
(a) Trace of process A	(b) Trace of process B	(c) Trace of process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Starting address of program of process A
8000 = Starting address of program of process B
12000 = Starting address of program of process C

Figure 3.3 Traces of Processes of Figure 3.2

3.2 进程状态

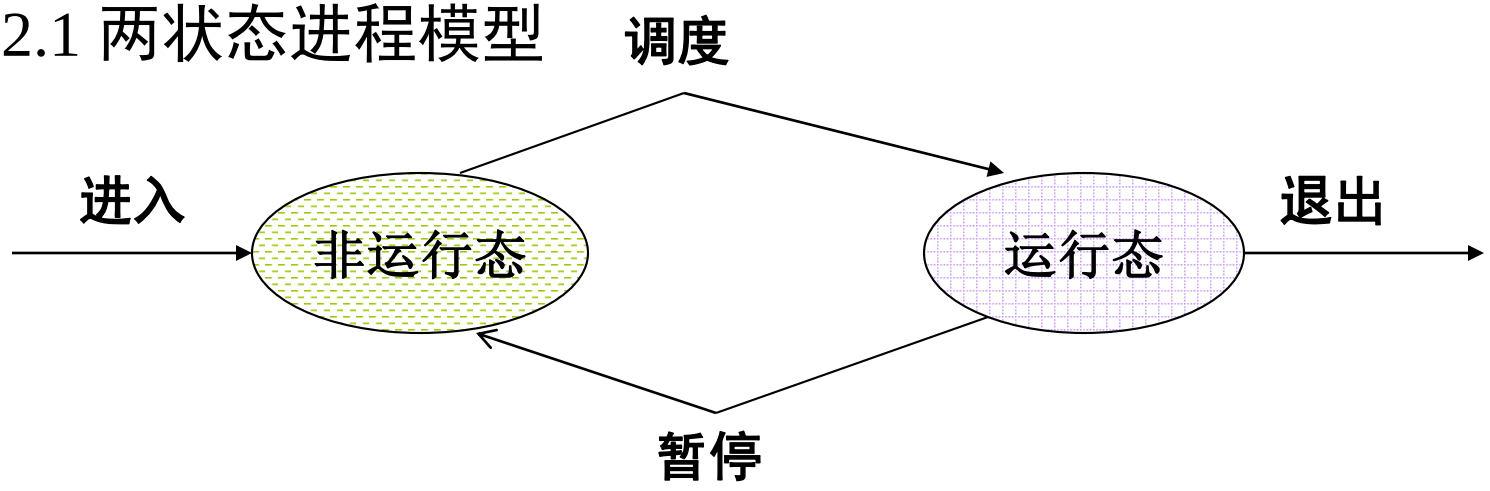
假设：操作系统仅允许一个进程连续最多执行6个指令周期



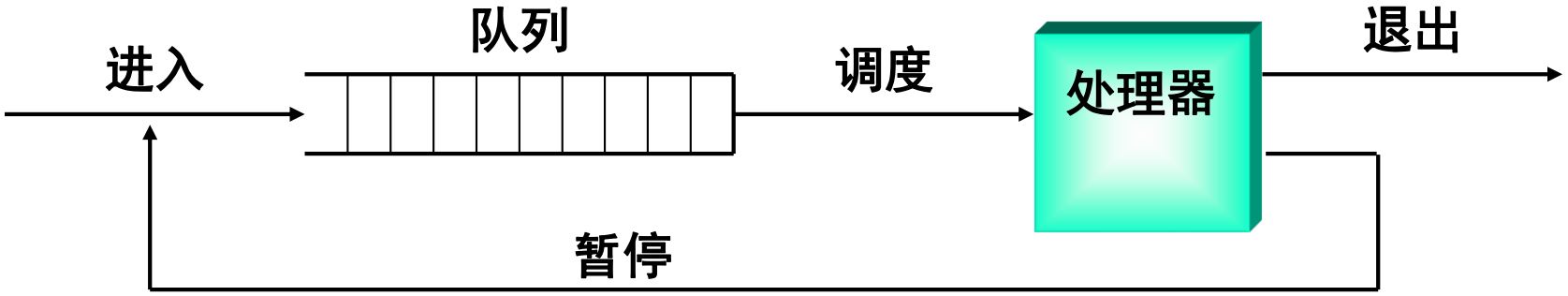
1	5000		27	12004	
2	5001		28	12005	
3	5002		----------	Time-out	←
4	5003		29	100	
5	5004		30	101	
6	5005		31	102	
----------	Time-out	→	32	103	
7	100		33	104	
8	101		34	105	←
9	102		35	5006	
10	103		36	5007	
11	104		37	5008	
12	105	→	38	5009	
13	8000		39	5010	
14	8001		40	5011	
15	8002		----------	Time-out	←
16	8003		41	100	
----------	I/O request	→	42	101	
17	100		43	102	
18	101		44	103	
19	102		45	104	
20	103		46	105	←
21	104		47	12006	
22	105	→	48	12007	
23	12000		49	12008	
24	12001		50	12009	
25	12002		51	12010	
26	12003		52	12011	
			----------	Time-out	←

3. 2 进程状态 ★★★★★

3.2.1 两状态进程模型



(a) 状态转换图



(b) 队列轮转图

3.2.2 进程的创建和终止

- 进程的创建：将一个新进程添加到正被管理的进程集时，OS建立用于管理该进程的数据结构，并为其分配地址空间。
- 创建进程的原因
  - 新批处理作业提交给操作系统
  - 交互系统终端用户登录到系统
  - 操作系统创建
  - 进程派生
- 进程派生
  - 父进程
  - 子进程

3.2.2 进程的创建和终止

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main(int argc, char *argv[]) {
6    printf("hello world (pid:%d)\n", (int) getpid());
7    int rc = fork();
8    if (rc < 0) {
9      // fork failed
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) {
13     // child (new process)
14     printf("hello, I am child (pid:%d)\n", (int) getpid());
15   } else {
16     // parent goes down this path (main)
17     printf("hello, I am parent of %d (pid:%d)\n",
18            rc, (int) getpid());
19   }
20   return 0;
21 }
```

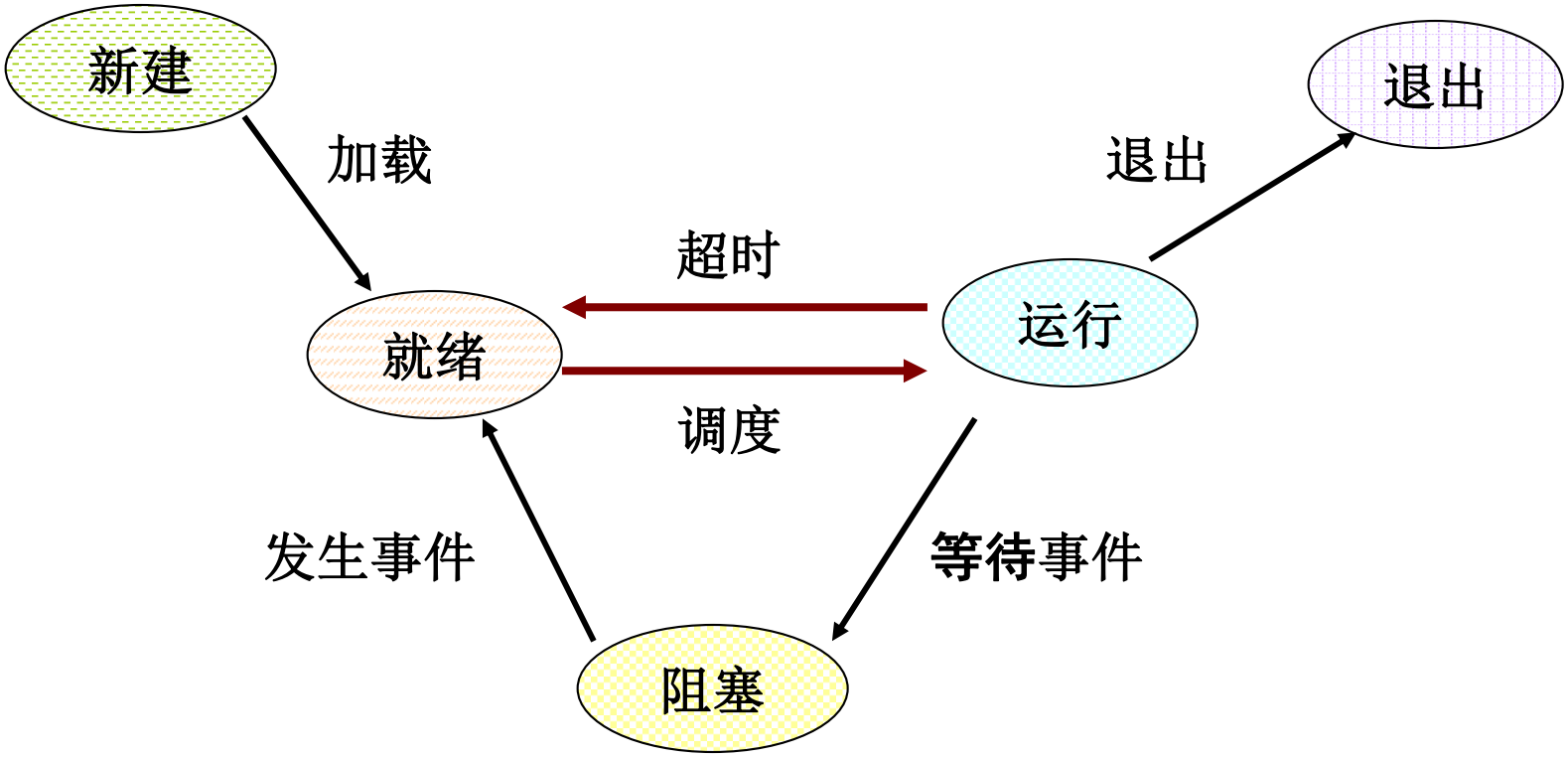
3.2.2 进程的创建和终止

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(int argc, char *argv[]) {
7    printf("hello world (pid:%d)\n", (int) getpid());
8    int rc = fork();
9    if (rc < 0) {           // fork failed; exit
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) { // child (new process)
13     printf("hello, I am child (pid:%d)\n", (int) getpid());
14   } else {                // parent goes down this path (main)
15     int rc_wait = wait(NULL);
16     printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
17            rc, rc_wait, (int) getpid());
18   }
19   return 0;
20 }
21
```

进程的终止

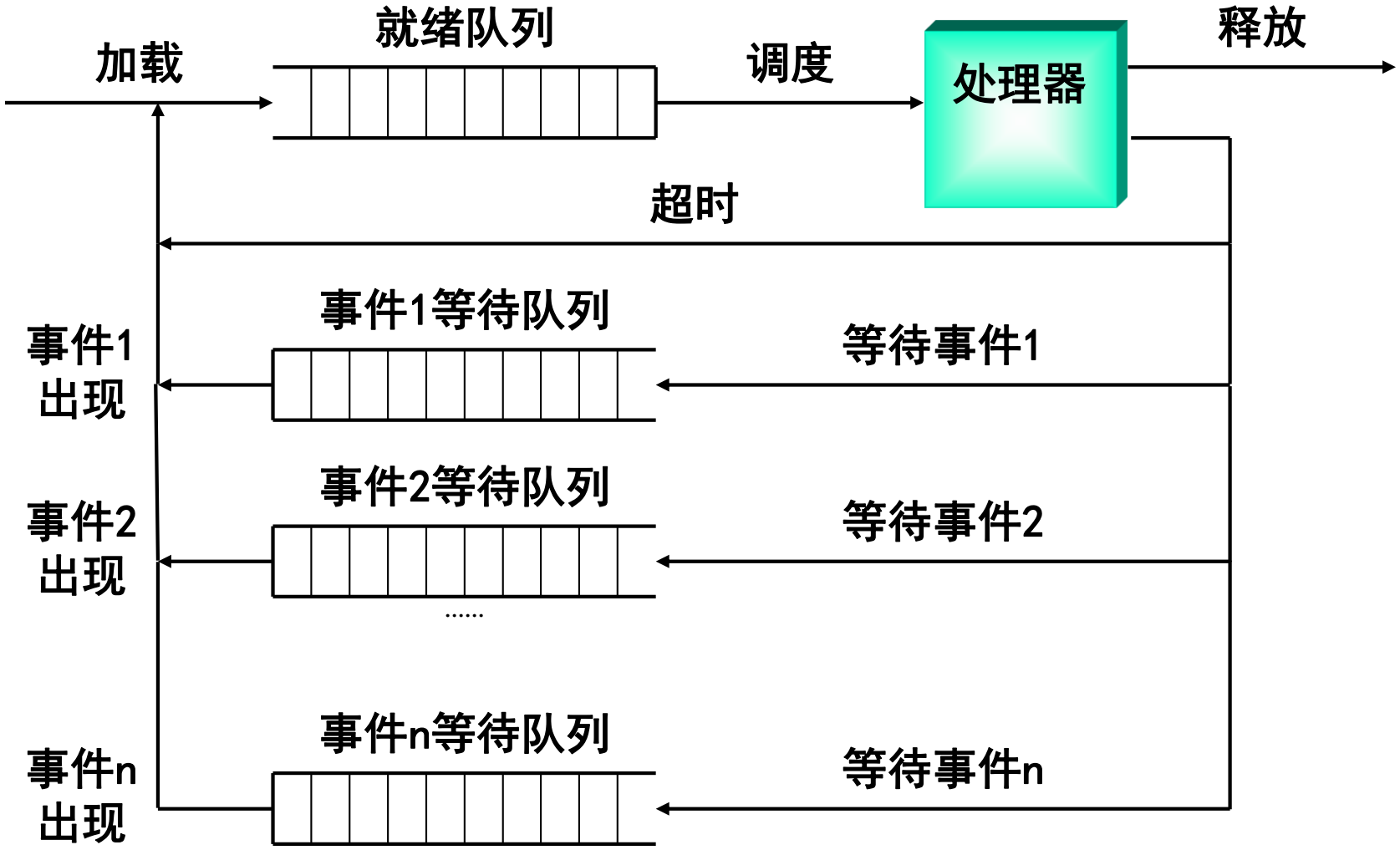
- 进程终止的原因
  - 正常完成
  - 各种错误和故障
  - 操作员或操作系统干涉
  - 父进程终止
  - 父进程请求终止子进程

3.2.3 五状态进程模型 ★★★★★



五状态进程模型

阻塞队列

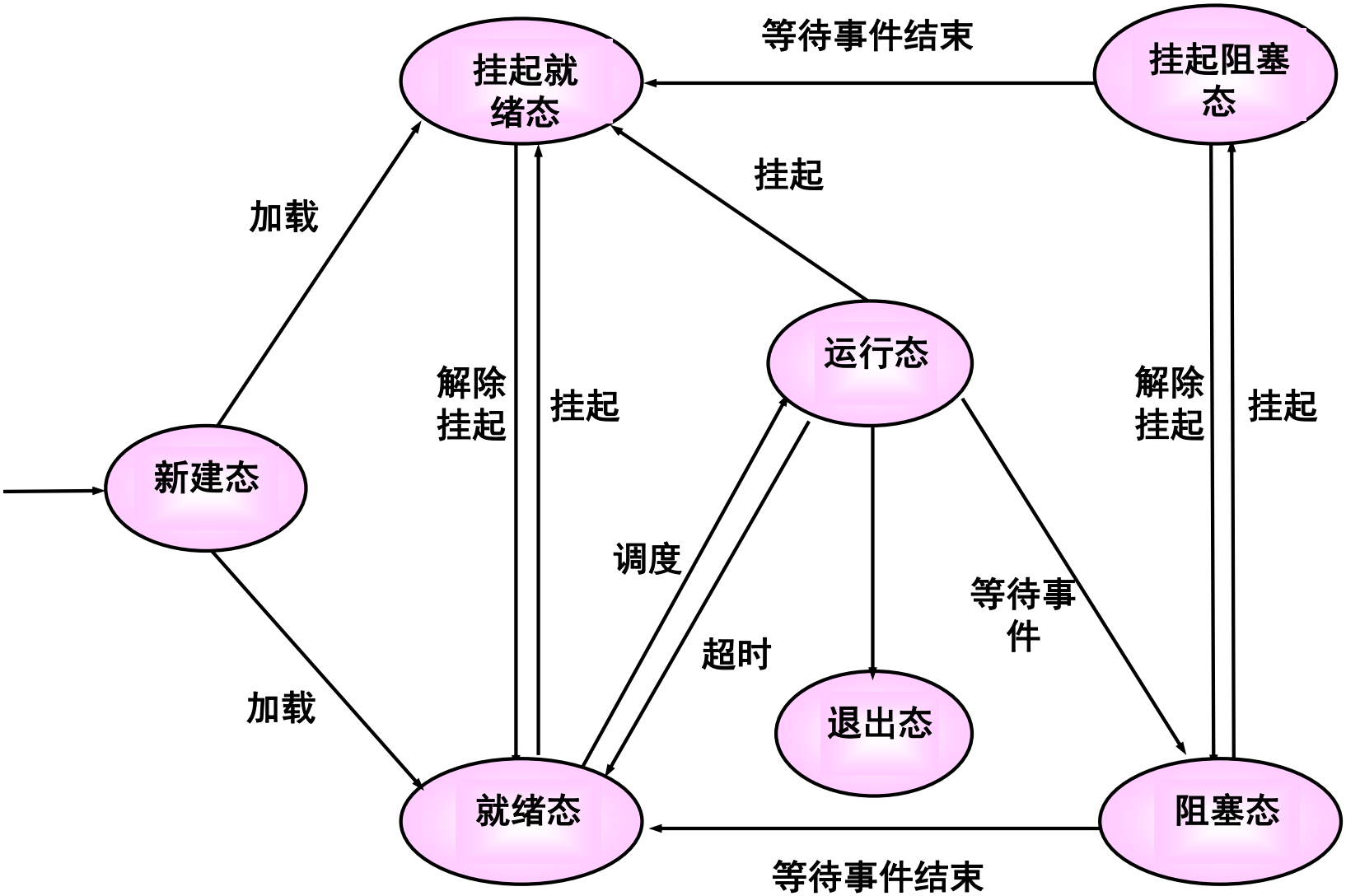

加载
就绪队列
调度
处理器
释放
超时
事件1等待队列
事件1
出现
等待事件1
事件2等待队列
事件2
出现
等待事件2
.....
事件n等待队列
事件n
出现
等待事件n

3.2.4 被挂起的进程

Table 3.3 Reasons for Process Suspension

Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The OS may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

3.2.4 被挂起的进程


挂起就绪态
等待事件结束
挂起阻塞态
加载
挂起
解除挂起
挂起
运行态
解除挂起
挂起
新建态
调度
等待事件
超时
加载
退出态
就绪态
阻塞态
等待事件结束

挂起进程的特点

- 进程不能立即执行；
- 若进程正在等待一个事件，阻塞条件不依赖于挂起条件，阻塞事件发生不会使进程立即被执行；
- 为阻止进程执行，可以通过代理把这个进程置于挂起状态，代理可以是进程自己，也可以是父进程或操作系统；
- 除非代理显式地命令系统进行状态转换，否则进程无法从这个状态中转移。

提问

1. 判断：程序就是进程。

A. 正确　　B. 错误

2. 操作系统用（）来管理和控制进程所需的内部数据。

A. 执行上下文　B. IR寄存器　　C. PC　　D. 数据寄存器

3. 进程是程序的一次执行，（）是进程存在的唯一标志。

A. 程序　　B. 数据　　C. 进程控制块　D. IR寄存器

4. 进程被中断时，操作系统会把上下文数据保存到其进程控制块中的相应位置，进程状态随后被改为其他值。

A. 正确　　B. 错误

5. 判断：在单CPU的系统中，任何时刻处于就绪状态的进程有多个，而且只有处于就绪状态的进程经调度程序选中后才可进入运行状态 。

A. 正确　　B. 错误

6. 判断：在单CPU的系统中，当有一个进程从阻塞态变成就绪态时，必然的另一个进程从就绪态变成运行态。

A. 正确　　B. 错误

提问

7. 进程的状态中，（ ）是指进程拥有除了CPU之外的所有资源的进程。

A. 执行状态　　B. 就绪状态

C. 挂起状态　　D. 等待状态

8. 若进程正在等待一个事件，它将会被挂起。

A.正确　　B.错误

3.3 进程描述 ★★★★★

3.3.1 操作系统的控制结构 ★★★★★

- 操作系统维护的信息可分为四类：
  - 内存表：跟踪内存和外存。
  - I/O表：管理计算机系统中的I/O设备和通道。
  - 文件表：提供关于文件的相关信息。
  - 进程表：管理进程。
- 内存、I/O和文件是代表进程而被管理的。

3.3 进程描述 ★★★★★

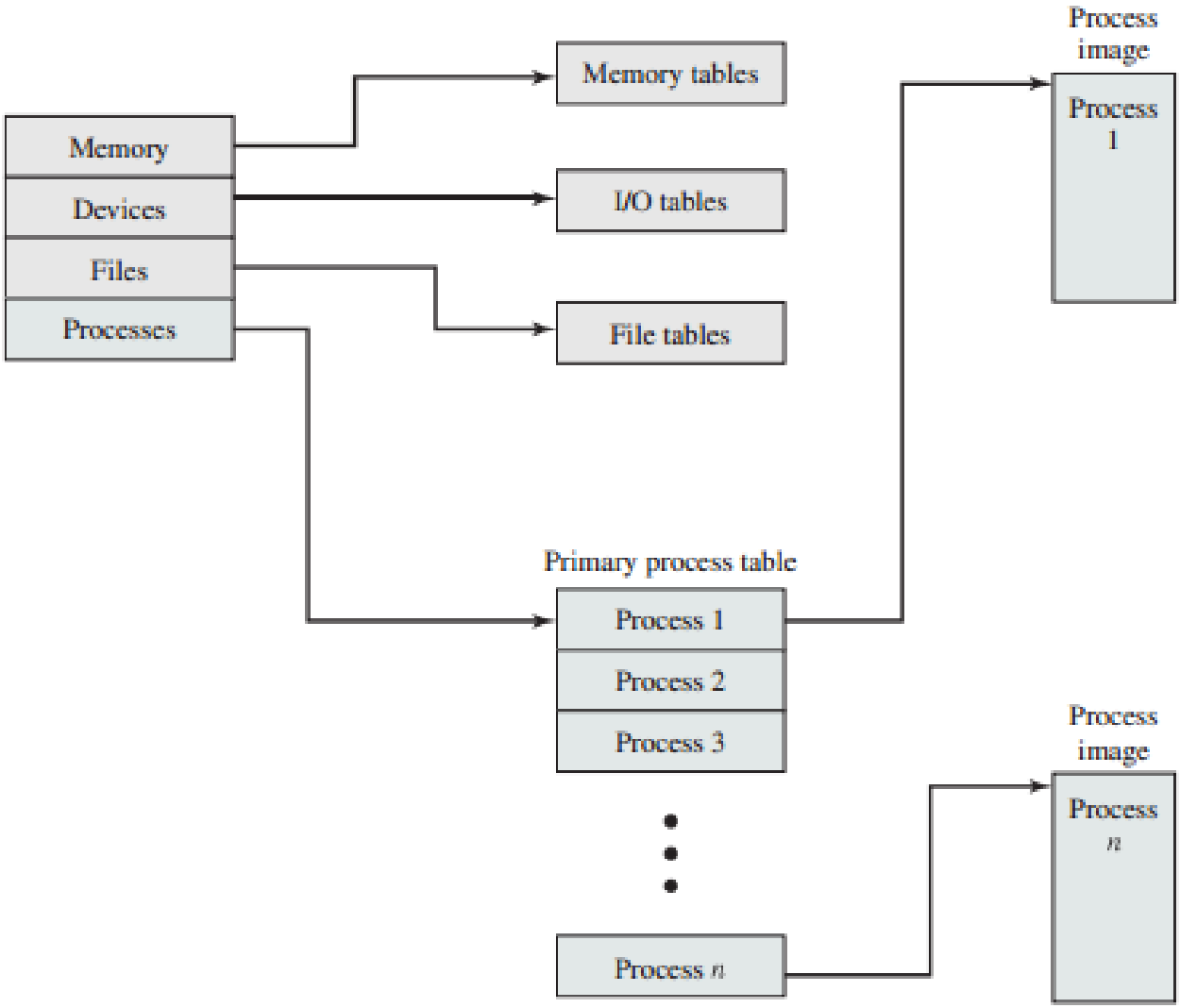


Figure 3.11 General Structure of Operating System Control Tables

3.3.2 进程控制结构 ★★★★★

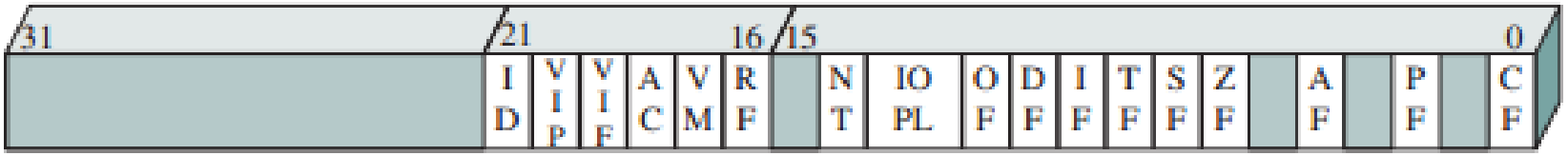
- 进程映像

项目	说明
用户数据	用户空间中的可修改部分，包括程序数据、用户栈区域和可修改的程序
用户程序	将被执行的程序
系统栈	每个进程有一个或多个系统栈，用于保存参数、过程调用地址和系统调用地址
进程控制块	操作系统控制进程所需要的数据

进程属性

- 进程控制块
  - 进程标识信息
    - ☒进程ID、父进程ID、用户ID
  - 处理器状态信息
    - ☒用户可见寄存器、控制和状态寄存器、栈指针
  - 进程控制信息
    - ☒调度和状态信息、数据结构、进程通信、进程特权、存储管理、资源的所有权和使用情况

进程属性



ID = Identification flag
VIP = Virtual interrupt pending
VIF = Virtual interrupt flag
AC = Alignment check
VM = Virtual 8086 mode
RF = Resume flag
NT = Nested task flag
IOPL = I/O privilege level
OF = Overflow flag
DF = Direction flag
IF = Interrupt enable flag
TF = Trap flag
SF = Sign flag
ZF = Zero flag
AF = Auxiliary carry flag
PF = Parity flag
CF = Carry flag

Figure 3.12 x86 EFLAGS Register

进程属性

每个进程映像的地址范围看似连续，但实际情况可能并非如此。

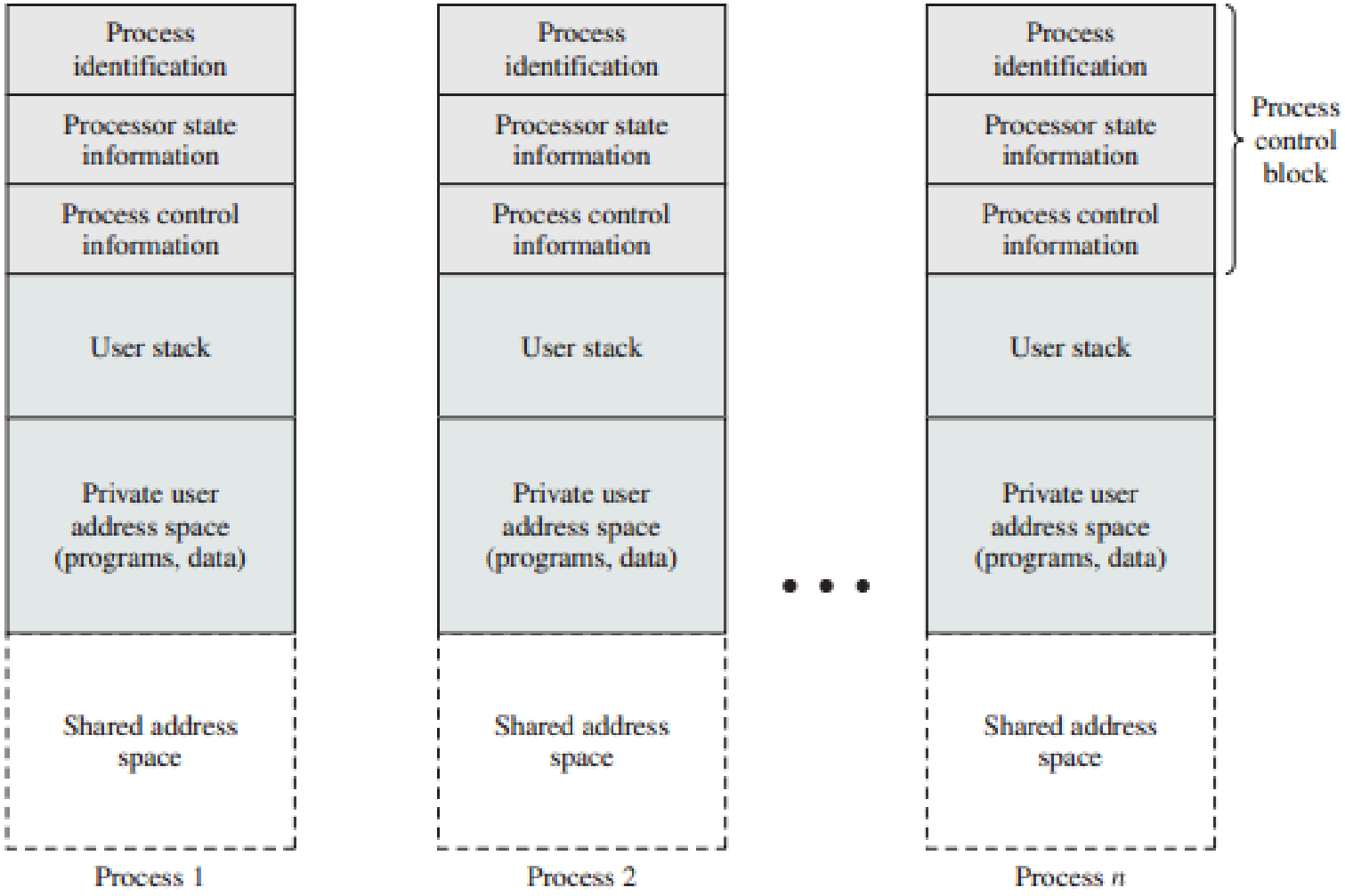
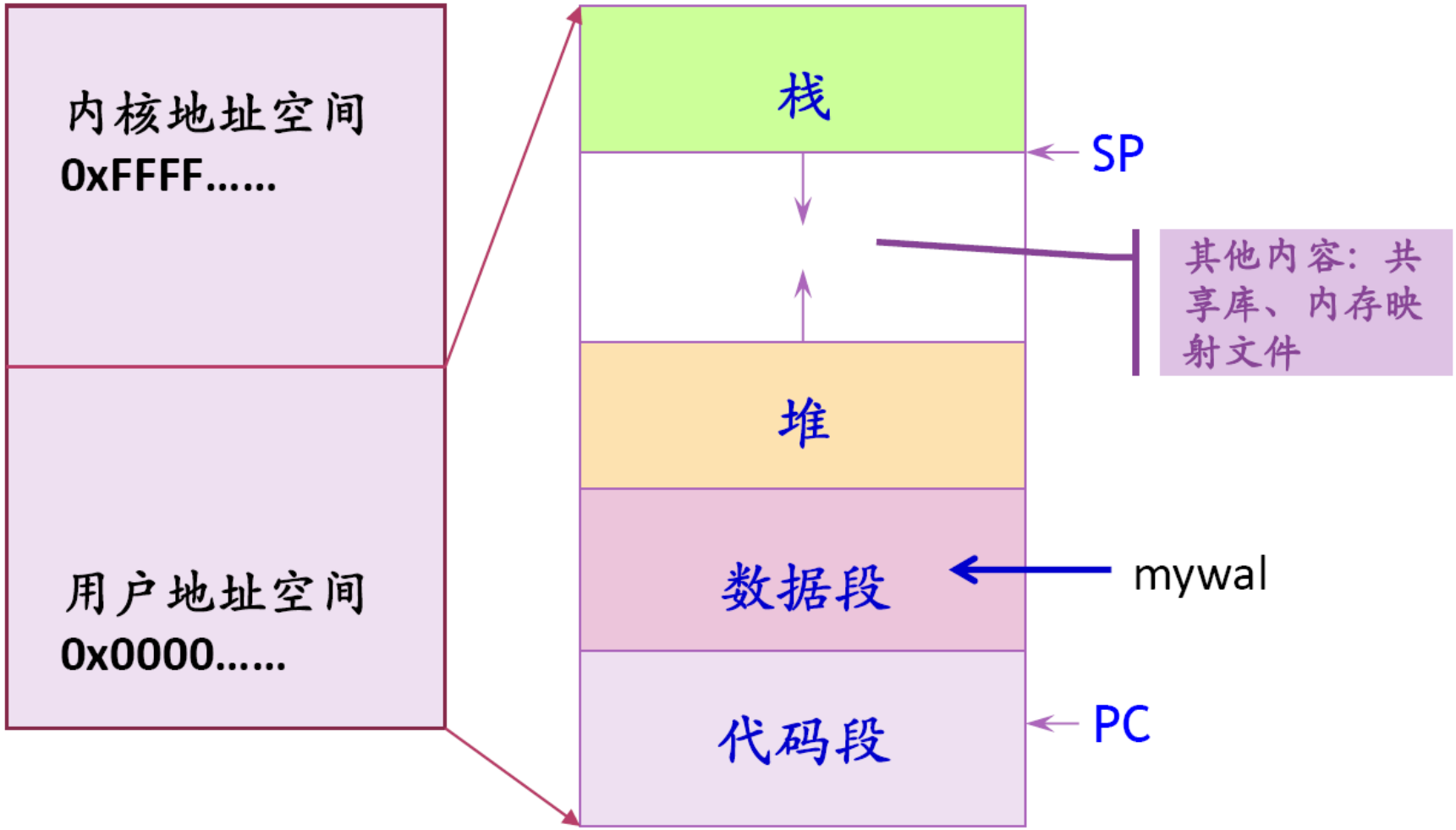


Figure 3.13 User Processes in Virtual Memory

进程属性


内核地址空间
0xFFFF……
用户地址空间
0x0000……
栈
SP
其他内容：共享库、内存映射文件
堆
数据段
mywal
代码段
PC

进程属性

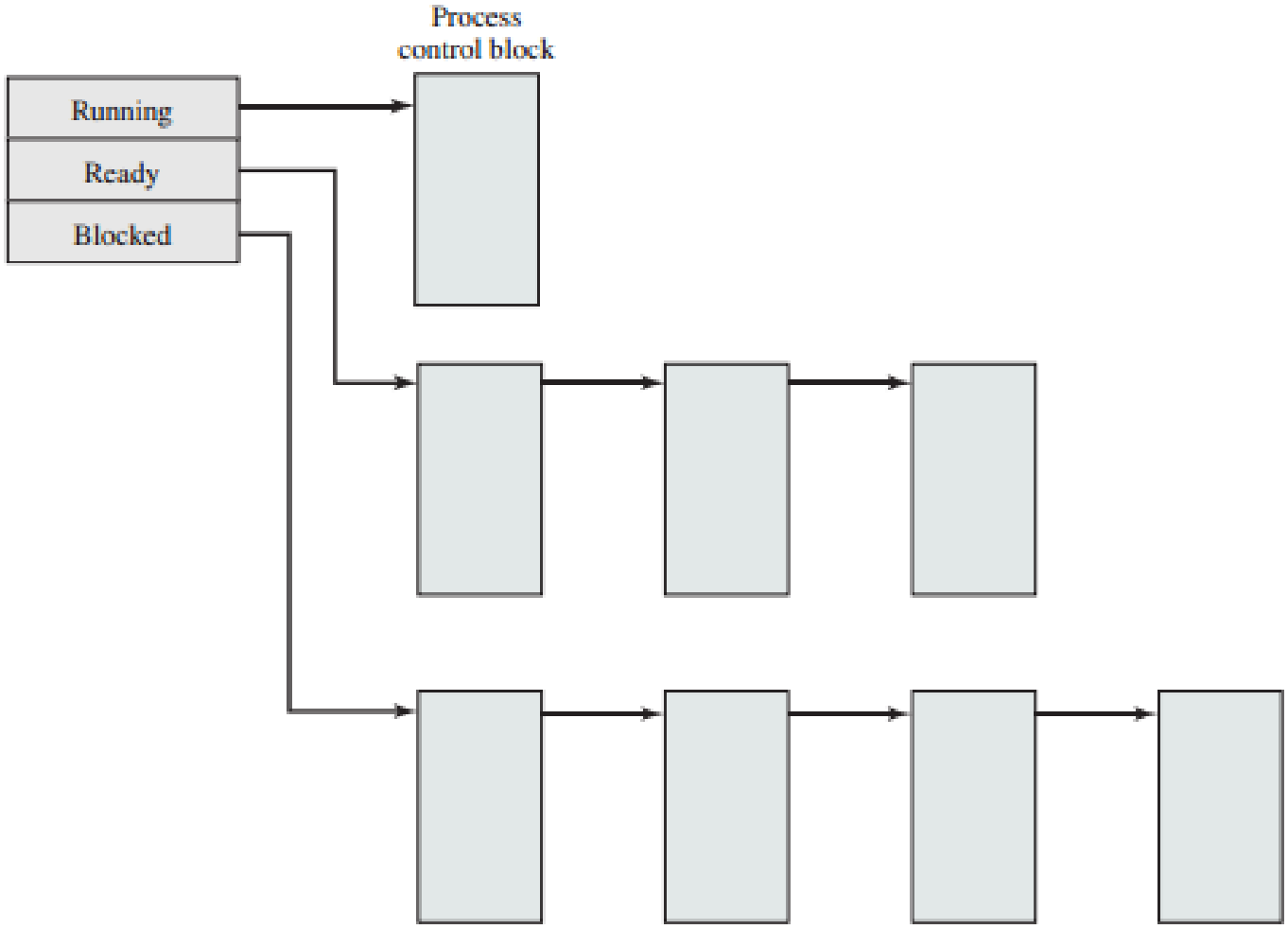


Figure 3.14 Process List Structures

进程属性

- 进程控制块的保护问题
  - 例程可以破坏PCB
  - PCB结构或语义中的设计变化可能会影响OS的许多模块
  - 解决方法：要求OS的所有例程都通过一个处理程序例程来解决，即处理程序例程是保护PCB、读写这些块的唯一仲裁程序。

3.4 进程控制 ★★★★★

3.4.1 执行模式

- 用户模式
  - 用户程序通常在该模式下运行，非特权态
- 系统模式
  - 控制模式、内核模式
  - 操作系统内核在该模式下运行，特权态
- 处理器如何获知和改变所处模式？
  - 程序状态字中有表示执行模式的位
  - 该位应某些事件要求而改变

3.4.2 进程创建 ★★★★★

- 创建进程的步骤
  - 分配进程标识符
  - 分配空间
  - 初始化进程控制块
  - 设置正确的连接
  - 创建或扩充其它数据结构

3.4.3 进程切换

- 进程切换是让处于运行态的进程中断运行，让出处理器，让操作系统指定的新进程运行。被中断进程的上下文环境需要保存。

1、切换时机

- 中断
  - 时钟中断
  - I/O中断
  - 内存失效
- 陷阱
- 系统调用

2、模式切换

- 当中断发生时，处理器需要做如下工作：
  - 把程序计数器置成中断处理程序的开始地址；
  - 暂时中断正在执行的用户进程，把进程从用户态切换到内核态，去执行操作系统例行程序以获得服务。
- 保存的进程上下文环境包括：
  - 所有中断处理可能改变的信息；
  - 恢复被中断程序时所需要的信息。
- 模式切换可以不改变正处于运行态的进程状态，保存和恢复上下文环境开销小；
- 进程切换涉及进程状态的变化，开销较大。

3、进程状态的变化

- 完整的进程切换步骤：
  - 保存处理器上下文环境；
  - 更新当前处于运行态进程的进程控制块；
  - 将进程的进程控制块移到相应的队列；
  - 选择另一个进程运行；
  - 更新所选择进程的进程控制块；
  - 更新内存管理的数据结构；
  - 恢复处理器在被选择的进程最近一次切换出运行状态时的上下文环境。

提问

9. 陷阱是作用是处理异步外部事件。

A. 正确　　B. 错误

10. 进程切换一定会引起模式切换，反之亦然。

A. 正确　　B. 错误

11. 多数操作系统中，中断发生后并不一定进行进程切换。

A. 正确　　B.错误

12. 发生中断时要保存处理器状态信息，并在控制权返回时恢复这些信息，保存和恢复功能通常由硬件实现。

A. 正确　　B. 错误

13. 进程是拥有资源的基本单位。

A. 正确　　B. 错误

14. 进程与程序是一一对应的关系。

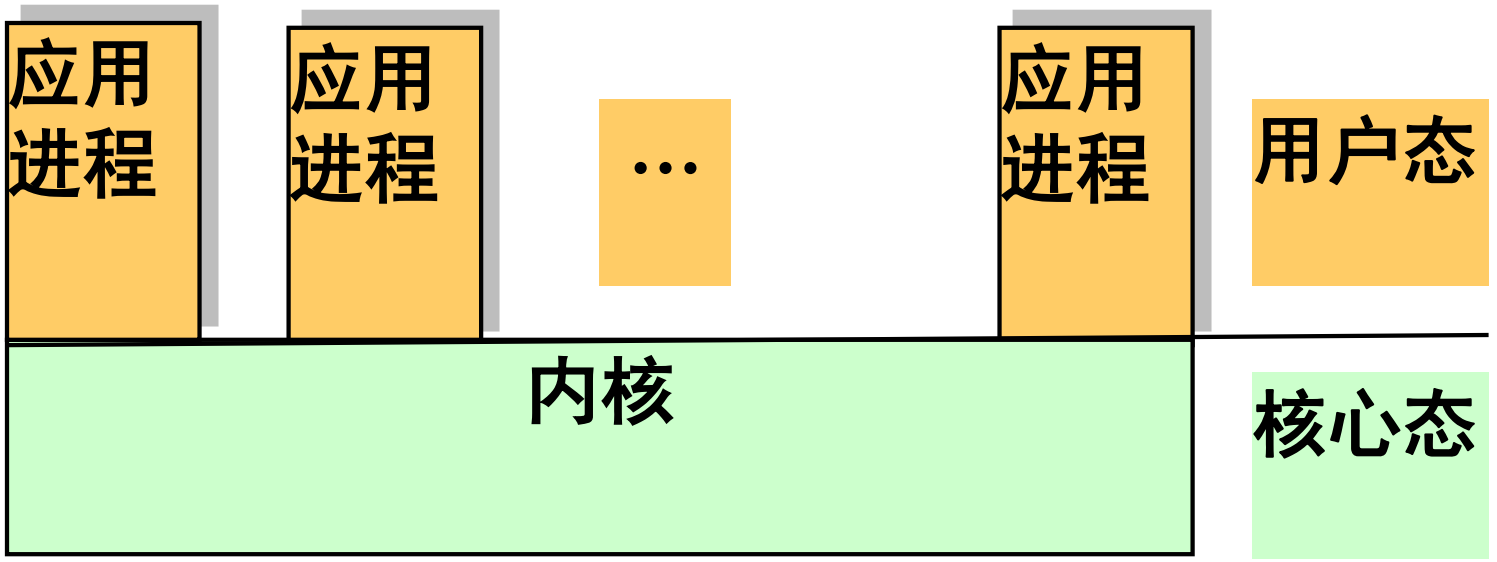
A.正确　　B.错误

3.5 操作系统的执行 ★★★★★

- 操作系统也是由处理器执行的一个程序，那么，操作系统是一个进程吗？如果是，如何控制？

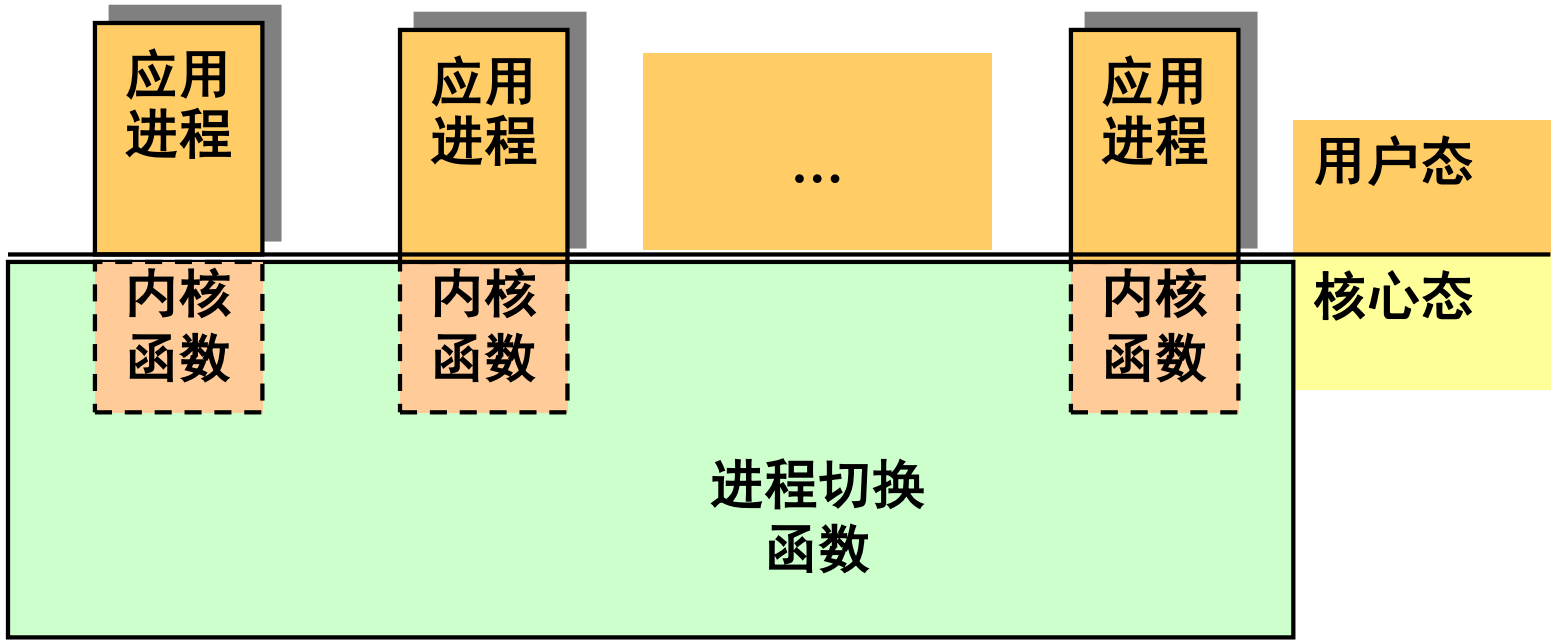
3.5.1 无进程的内核

- 进程的概念仅适用于用户程序，操作系统代码作为一个在特权模式下工作的独立实体被执行。



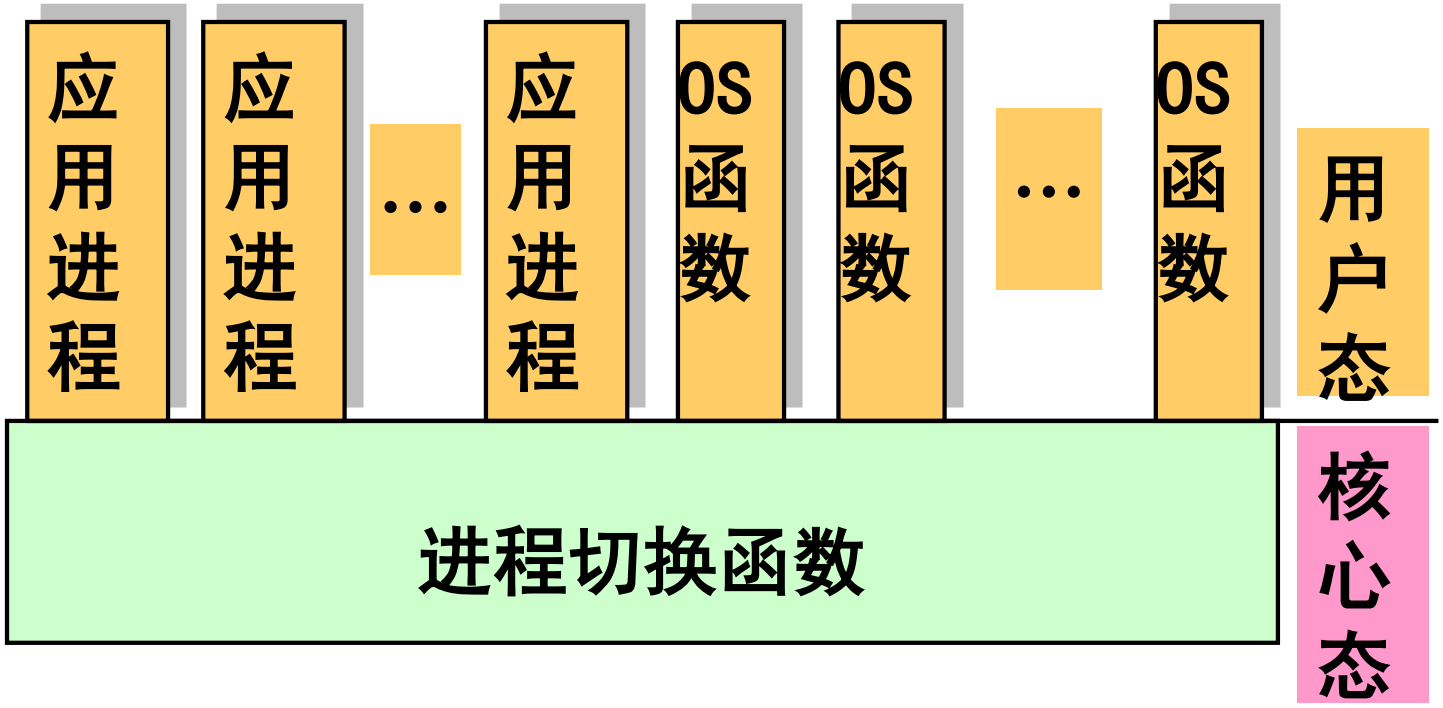
3.5.2 在用户进程中执行

- 操作系统是用户调用的一组例程，在用户进程环境中执行，用于实现各种功能。



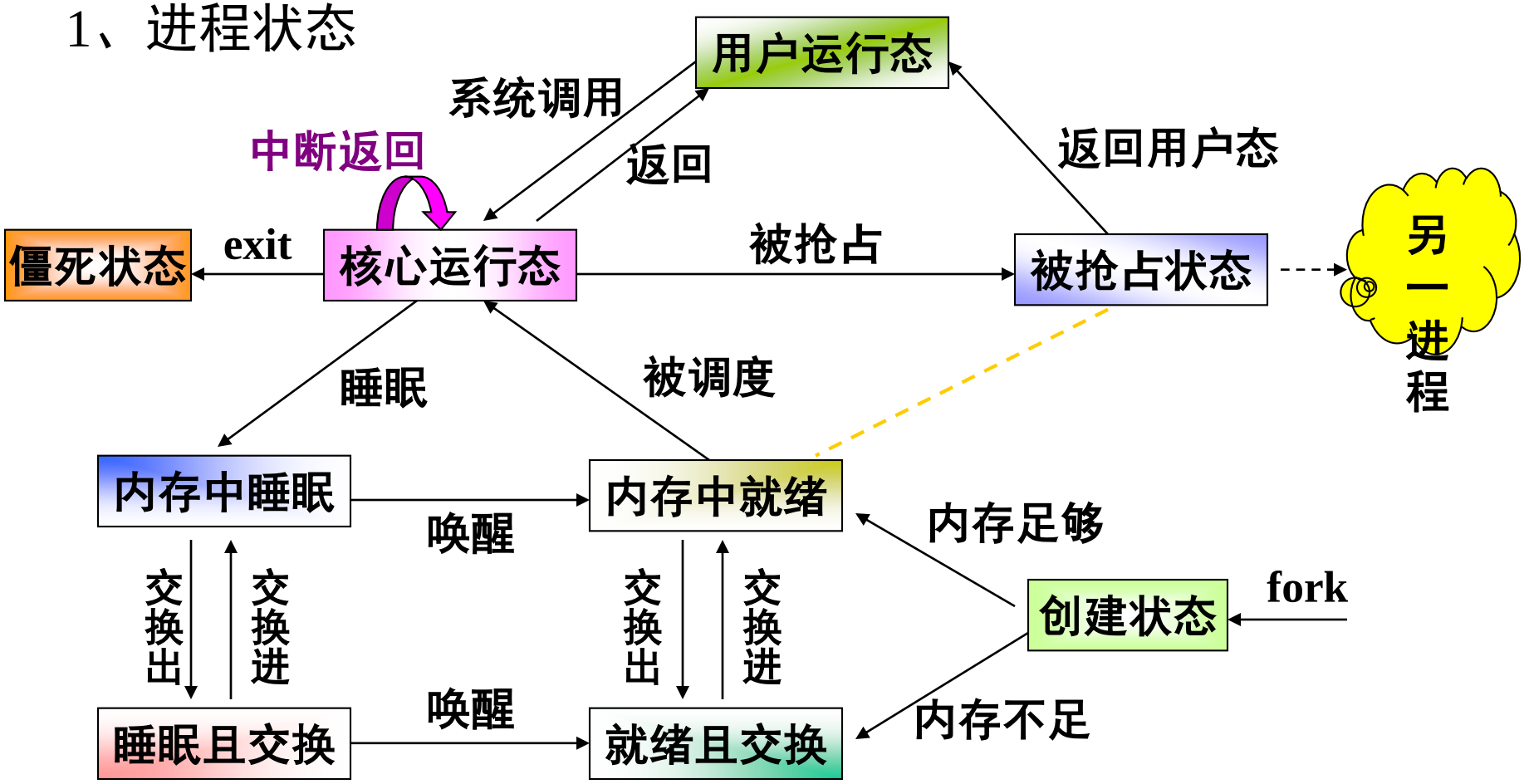
3.5.3 基于进程的操作系统

- 把操作系统作为一组系统进程来实现，主要的内核函数被组织成独立的进程。



3.6 UNIX SVR4 进程管理 ★★★★★

1、进程状态



2、进程创建

- UNIX中，父进程通过系统调用fork()创建子进程，之后可能进行下面三种操作之一：
  - 在父进程中继续执行。控制返回用户态下父进程进行fork调用处。
  - 处理器控制权交给子进程。子进程开始执行代码，执行点与父进程相同。
  - 控制转交给另一个进程。父进程和子进程都置于就绪态。
- 从fork中返回时，测试返回参数：
  - 若值为0，则是子进程，可以转移到相应的用户程序中继续执行；
  - 若值不为0（子进程的PID），则是父进程，继续执行主程序。

作业

- 复习题3.3，3.14
- 习题3.2