

第三章

进程控制和管理

本章教学目标

- 理解进程的含义
- 掌握进程上下文的构成
- 掌握进程控制块的构成与作用
- 掌握进程切换的时机、机制，以及与模式切换的区别
- 掌握进程的三态、五态与七态模型
- 掌握线程的概念

进程及其实现

- 进程的定义：
 - 进程是可并发执行的程序在某个数据集合上的
存储内容不同也不止一个进程
多次即不同进程一次计算活动，也是操作系统进行资源分配和保护的基本单元。
- 引入进程的目的：
 - 刻画系统的动态性，发挥系统的并发性。
 - 解决共享性，正确描述程序的执行状态。

进程的含义

- **进程是正在执行中的程序**
动 静
- 程序本身并非进程
- 程序是**非活跃实体**，例如磁盘上存储的包含指令序列的文件（通常称为可执行文件）
- 进程是活跃体，包含程序计数器，用于确定下一条将要执行的指令，同时包含一组相关的资源。
- 进程是可以被赋予处理器资源并在其上运行的实体
- **当可执行程序被载入内存时变成进程**
- 同一个程序的多次运行表示为多个进程
- 进程是活动单元，由一组执行指令，当前状态以及被分配的系统资源界定。*上文？*

进程

同时写同一个文件？

- 操作系统如何保证并发执行进程的正确性？
 - 需要维护什么信息？
 - 为了保证进程的正确执行，OS需要做什么？

进程的内涵

---进程映像

- 程序代码
- 数据
- 堆栈
- 与进程执行相关的信息
 - 标识符
 - 状态
 - 优先级
 - 程序计数器
 - 地址指针
 - 通用寄存器数据
 - I/O状态信息
 - 审计信息，如使用的CPU时间

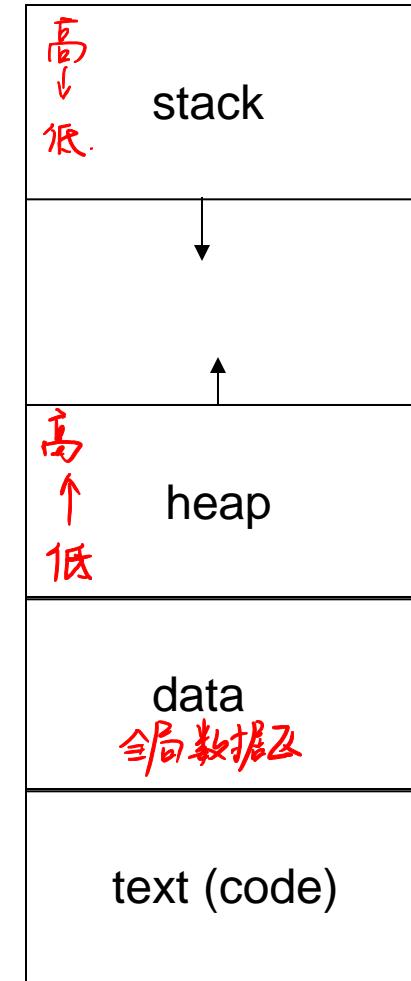
进程上下文

context

- 进程物理实体和支持进程运行的环境合称为进程上下文

进程上下文

- 包含所有运行程序所需要的状态信息
 - 进程完成任务所需要的信息：代码，数据，栈，堆。
 - 这构成了用户级上下文.



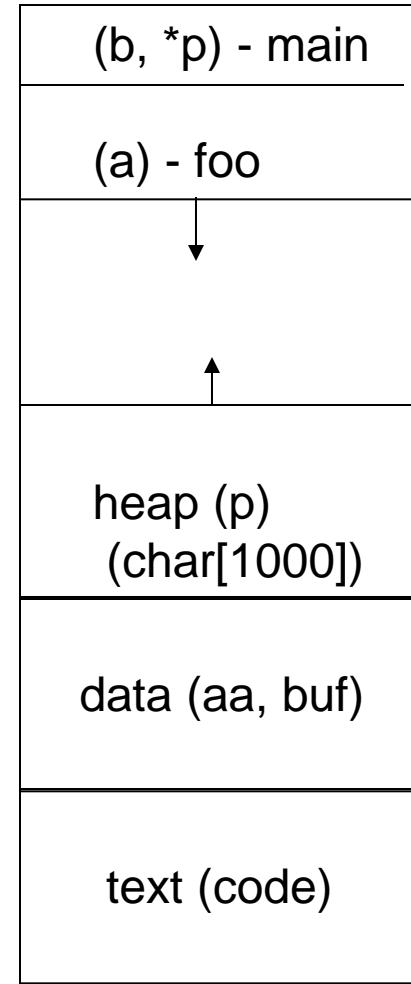
main(int argc, char * argv[])

argv[0] = "sum"
argv[1] = "100"

用户级上下文

```
...  
int aa;  
char buf[1000]; // 数组区  
void foo() {  
    int a;  
    ...  
}  
main() {  
    int b; // 栈区  
    char *p; // 栈区  
    p = new char[1000]; // 堆区 动态分配内存  
    foo();  
}
```

MAX

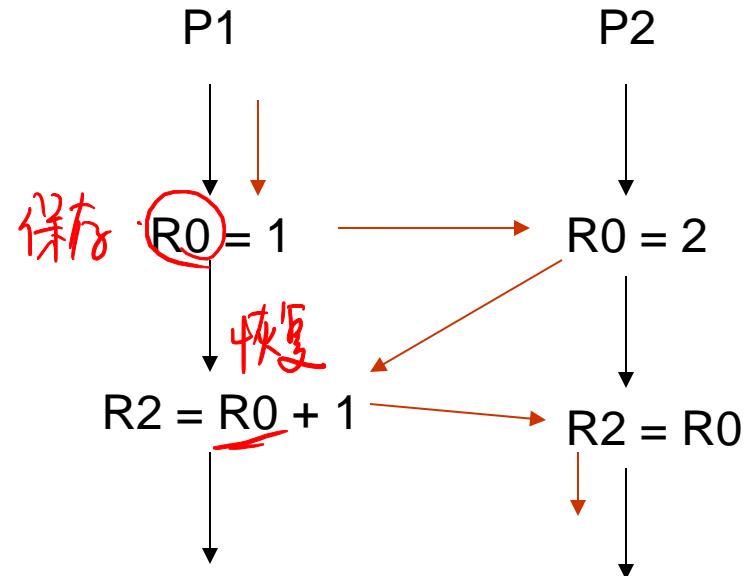


stack

进程所占用的内存

进程上下文

- 包含所有运行程序所需要的状态信息
 - 用户级上下文是否足够?
 - 仅当系统一个时刻只运行一个程序时正确。
 - 系统通常需要在不同的程序之间切换。



- P1中寄存器R2的值将出错. 如何保证其正确性?
 - 在进程切换前保存进程P1中R0的值;
 - 当从进程P2切换回P1时重新载入寄存器R0的值。
- 寄存器应该是进程上下文的组成部分! 称为寄存器上下文!

- 进程上下文:
 - 用户级上下文
 - 代码, 数据, 栈, 堆
 - 寄存器上下文(通用寄存器,..., 程序计数器, 栈指针, 程序状态字, etc).
 - 其它信息 (也称为系统级上下文)
 - 进程标识信息
 - 进程调度的相关信息, 如进程优先级
 - 打开的文件列表
 - 存储管理相关的信息, 如段/页表指针
 - 分配的设备列表
 - 记账信息, 如已执行的时间总和
 - ...

进程上下文的重要性

- 为了正确运行进程，进程的指令需要在进程的上下文中执行。

进程上下文的存储位置

- 用户级上下文存储在内存中。
- 其它的上下文信息存储在**进程控制块**中。
- 操作系统维护一张进程控制块的表，每个进程都对应于表中的一项。
- 进程控制块还包含其它操作系统用户管理进程的信息：
 - 进程状态 (running, waiting, etc)
 - 进程优先级
 -

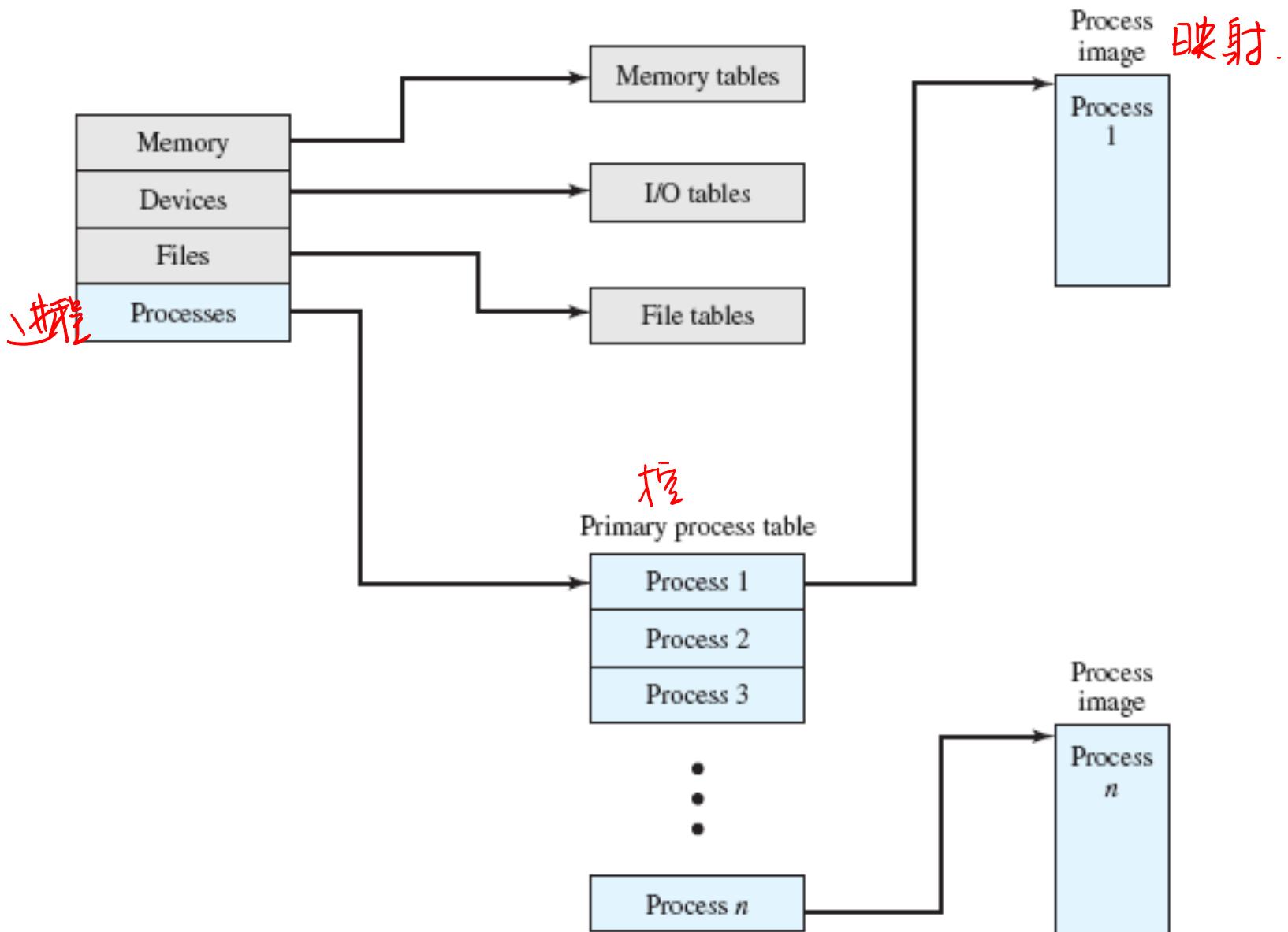


Figure 3.11 General Structure of Operating System Control Tables

进程控制块

- 每个进程有且仅有一个进程控制块(PCB)，是操作系统用来记录和刻画进程状态及有关信息的数据结构，是进程动态特征的一种汇集，也是操作系统掌握进程的唯一资料结构和管理进程的主要依据。
- 进程控制块是操作系统中最重要的数据结构。
- 进程控制块集合定义了操作系统的状态！

进程控制块

- 标识信息 *谁*
 - 进程标识ID
 - 进程组标识ID
- 现场信息 *(寄存器内容)*
 - 程序计数器
 - CPU寄存器：包括栈指针，通用寄存器等；
- 控制信息
 - 进程状态
 - CPU调度信息，如优先级，调度队列指针，调度参数等；
 - 内存管理信息：依据内存管理的方式不同而不同，如基/限长寄存器，段/页表指针等
 - 计费信息：如进程使用的CPU时间
 - I/O状态信息：包括分配给该进程的I/O设备列表，打开文件列表等；



进程控制块的例子

~~虚拟机~~: CPU、内存、I/O文件
~~中断~~

操作系统对CPU的虚拟化

- 硬件现实:
 - 一个CPU按“取指令-执行指令”的顺序运行;
~~中断~~ →
- 操作系统对CPU的虚拟化:
 - 每个进程都有一个CPU,
在该进程中按“取指令-执行指令”的顺序执行。
 - 每个进程在自己的进程上下文中运行。

```
Load PC;  
IR = MEM[PC];  
While (IR != HALT) {  
    PC ++;  
    Execute IR;  
    IR = MEM[PC];  
}
```

操作系统对CPU的虚拟化

- 操作系统应该做什么?
 - 将三个进程的指令按序嵌入硬件指令序列中

进程X的指令: x0, x1, x2,
进程Y的指令: y0, y1, y2, ...
进程Z的指令: z0, z1, z2, ...

汇编层级的指令



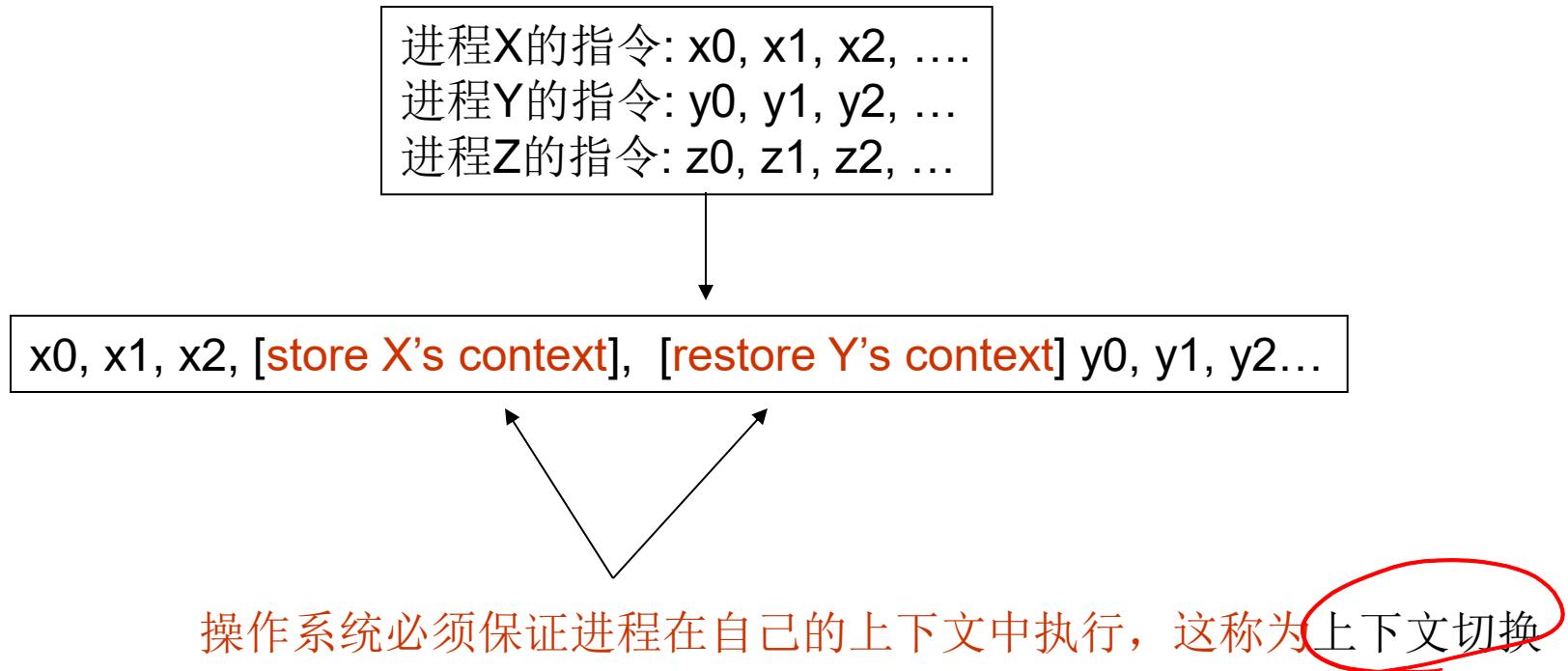
硬件指令? x0, x1, x2, y0, y1, y2, z0, z1, z2, x3, x4, x5, ...

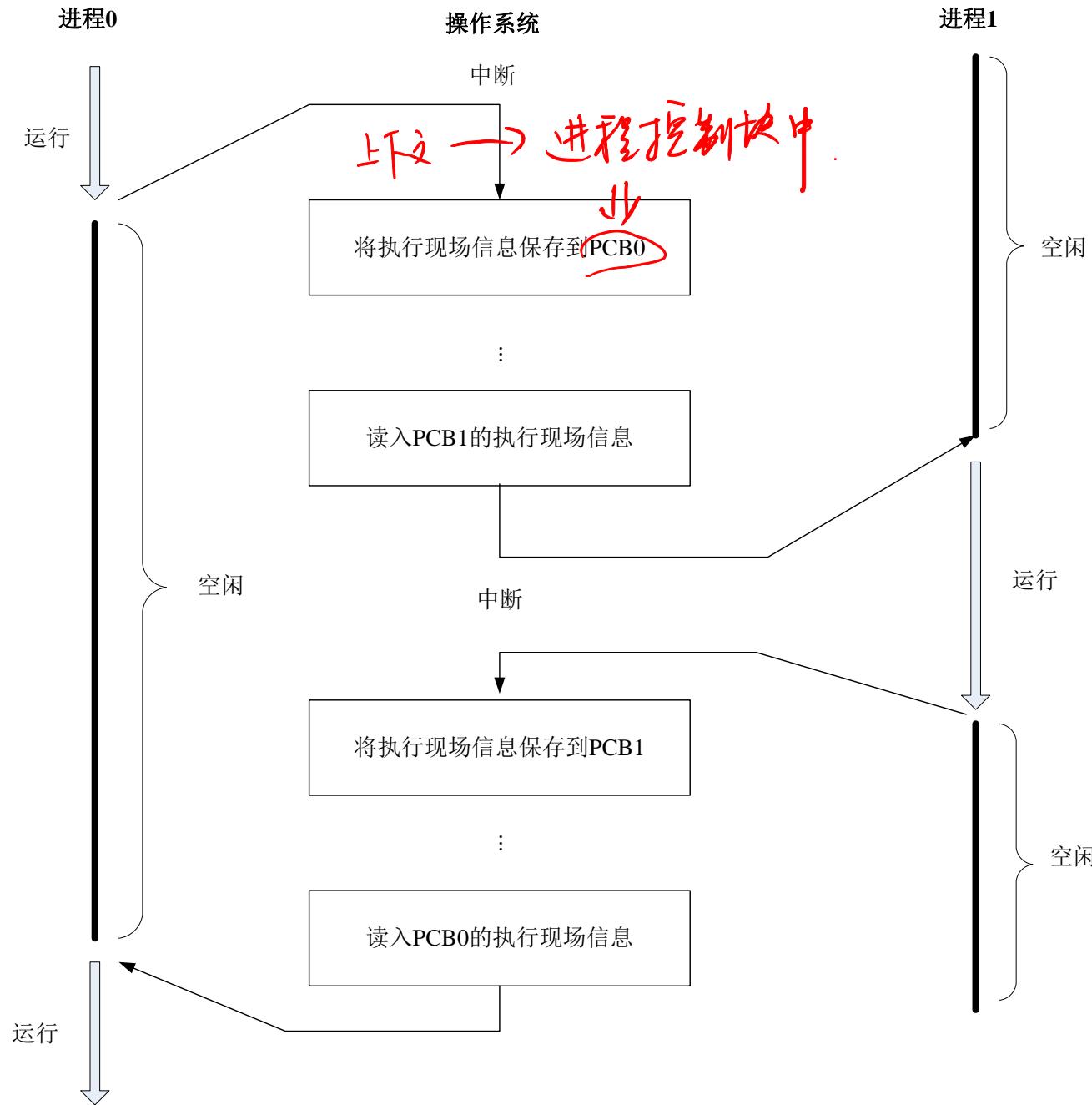
上述嵌入是否正确?

寄存器改变了!!!

No!!为了保持执行的正确性, 一个进程的指令必须在它的
进程上下文中执行

操作系统对CPU的虚拟化





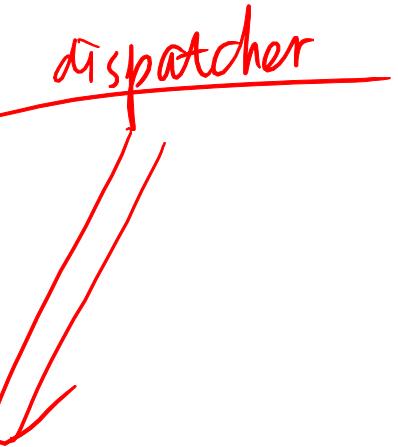
分派循环

- 系统执行的硬件视图: 分派循环

- LOOP

- 执行进程
 - 保存进程状态
 - 选择一个新的就绪进程 ←———— 调度
 - 载入被选中进程的状态

上下文
切换:
分派器
代码



进程切换引发的问题

- Q1 • 当进程开始运行时， 操作系统如何重新获得控制？
- Q2 • 进程需要保存哪些状态？
- Q3 • 分派器如何选择下一个待执行的进程/线程？

Q₁

进程切换的时机

- 进程切换可以发生在操作系统从当前运行进程获得控制权后的任何时刻
- 进程切换也只能在操作系统获得控制权，即系统处于核心态时进行 用户态 ×
- 操作系统获得控制权的方式
 - { -① 中断(interrupt) 被动
 - ② 系统调用 主动请求 陷入机制

中断

- 时钟中断
 - 操作系统确定当前运行进程的时间片是否已到，如果是，则当前进程必须切换到就绪态，而调度其它进程执行。
- I/O中断
 - 操作系统确定当前I/O操作的类型。如果所产生的I/O事件是一个或多个进程正在等待的，则这些被阻塞进程将被移动到就绪态。操作系统接着必须决定是重启被暂停的进程，还是用更高优先级的就绪进程抢占当前进程。
- 缺页
 - 处理器遇到虚拟内存地址不在主存中的错误。则操作系统必须将被访问的块（页或段）载入内存。当将其载入内存的I/O请求被发出后，发生内存缺页的进程将处于阻塞态，操作系统随后进行进程切换执行另一个进程。当所需要的块载入内存完毕后，对应的进程被置于就绪状态。

中断

- 程序性中断
 - 操作系统确定所发生的错误或异常是否是致命的。若是，则当前运行进程将被置为Exit状态，并进行进程切换；~~若不是致命错误，则操作系统的~~行为依赖于错误的性质和操作系统的设计。它可能会尝试一些恢复性操作或简单地通知用户，或者它可以进行进程切换或重启当前运行进程。

系统调用

- 系统调用有可能将用户进程置于阻塞状态，从而引发进程切换，如发出I/O操作请求。

运行 → 阻塞.

Q2:

进程切换应该保存的状态信息

- 任何下一个进程可能会破坏的信息
 - 程序计数器
 - 寄存器
 - 内存管理信息
 - ◦◦◦

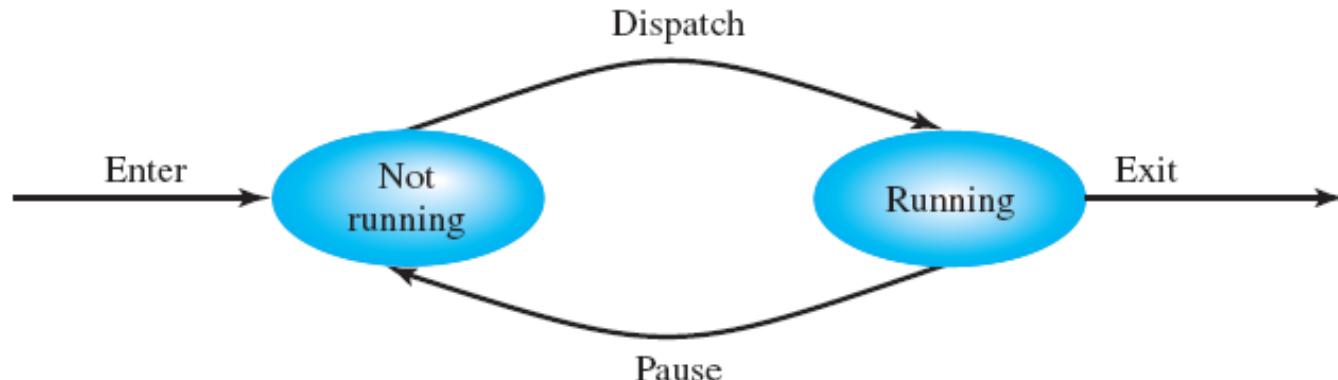
Q₃=

分派器如何选择下一个进程执行

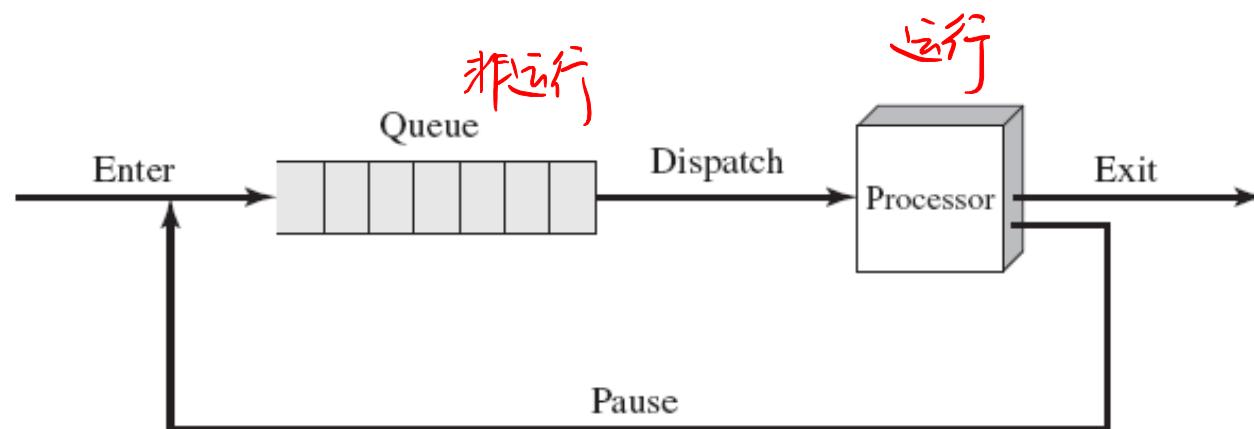
- 分派器维护了一个就绪进程列表
- 如果没有就绪进程
 - 分派器循环等待
- 否则，分派器使用一种**调度算法**来选择下一个进程。

进程状态

- 两态模型
 - 非运行态：进程不占用CPU
 - 运行态：进程占用CPU



(a) State transition diagram



(b) Queuing diagram

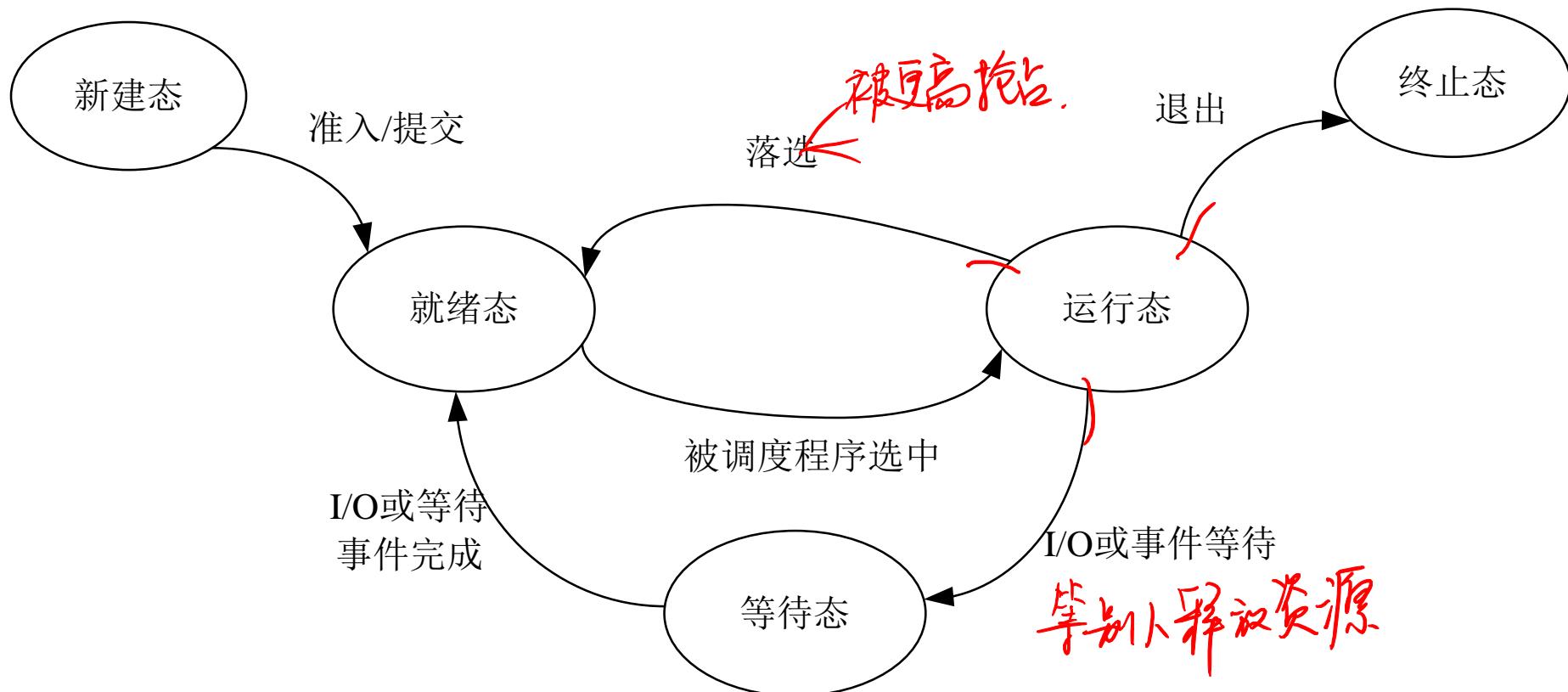
Q: 有何缺点?

进程状态

- 三态模型
 - **运行态**: 进程占用处理器运行
 - **就绪态**: 进程具备运行条件, 等待系统分配处理器以使其运行的状态
 - **等待态**: 进程不具备运行条件, 正在等待某个事件的发生
- 此外, 通常还引入下面两种状态 (五态模型)
 - **新建态**: 进程刚被创建时, 尚未进入就绪队列。有时候将根据系统性能的要求或主存容量的限制推迟新建态进程的提交。
 - **终止态**: 进程完成执行, 到达正常结束点

未完

进程状态转换图



进程状态的转换

- 运行态→等待态
 - 运行进程等待使用某种资源或某事件发生，如等待用户输入数据
- 等待态→就绪态
 - 所需资源得到满足或某事件已经完成，如数据传输结束
- 运行态→就绪态
 - 运行时间片到，或出现更高优先级的进程
- 就绪态→运行态
 - 被调度程序选中占用CPU

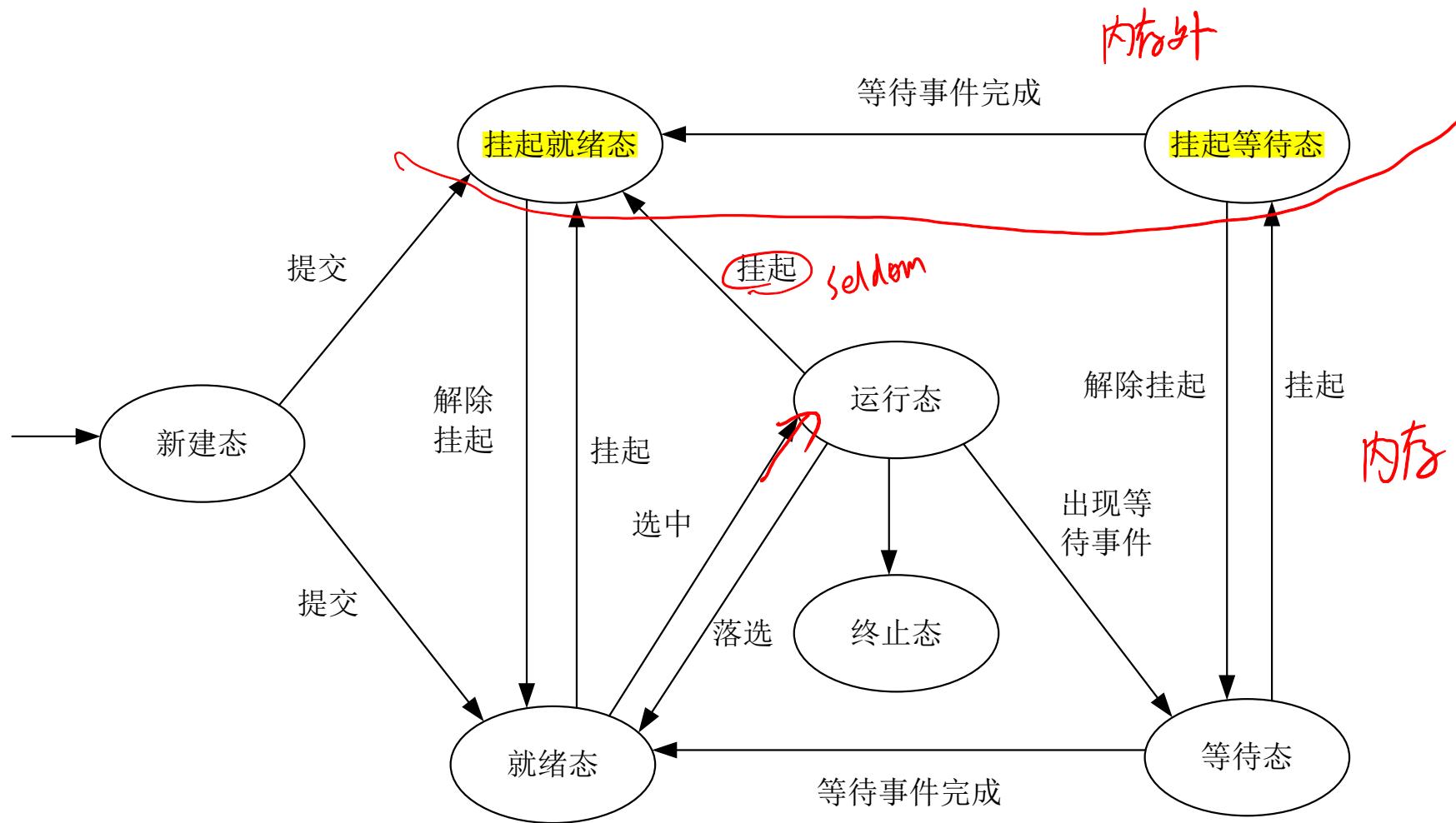
具有挂起功能的进程状态及转换

- 受限于主存资源限制，将某些进程挂起，并置于磁盘对换区。*移出内存*
 - 假设所有主存中的进程都等待I/O操作的完成
- 挂起进程不参与低级调度直到它们被对换进主存。

内存受限，多个进程等待。页面不断被替换 ⇒

具有挂起功能的进程状态及转换

- 与进程执行相关的两个相互独立事件：
 - 进程是否等待某个事件的发生
 - ready or blocked?
① ②
 - 进程是否处于内存
 - Suspended or not?
① ②
- 从而形成四个状态：
 - 就绪态 (*not suspended*)
 - 等待态 (*not suspended*)
 - 挂起等待态
 - 挂起就绪态



具有挂起进程功能的系统的进程状态及其转换图

- 等待 → 挂起等待
 - 不存在就绪进程
 - 当前运行进程或即将运行的就绪进程需要足够多的内存空间以保持性能
- 挂起等待 → 挂起就绪
 - 当等待的事件发生
- 挂起就绪 → 就绪
 - 当前主存中不存在就绪进程
 - 挂起就绪态进程比主存中所有就绪态进程的优先级都高
- 就绪 → 挂起就绪
 - 只有将某个就绪进程挂起才能释放足够多的内存空间
 - 挂起一个低优先级的就绪进程可能比挂起高优先级的阻塞进程更有利
- 挂起等待 → 等待
 - 当某个进程结束释放部分主存，同时挂起等待队列中的某个进程比挂起就绪队列中的所有进程优先级都高，并且OS认为等待事件即将发生。
- 运行 → 挂起就绪
 - 当一个具有高优先级的挂起等待态进程所等待的事件发生，操作系统决定抢占当前运行进程，则可以直接将当前进程变为挂起就绪态，释放部分内存空间

Linux中的进程控制块 PCB

--task_struct

- 每个进程一个task_struct结构
- task_struct定义在 include/linux/sched.h头文件中
- task数组有内核分配，代码在 kernel/sched.c文件中

```
struct task_struct *task[NR_TASKS]={&init_task};
```

- 内核有一个全局变量current指向当前正在运行的进程

Linux中的进程控制块 --task_struct

- 进程状态
 - running : 正在运行或就绪
 - Waiting, interruptible: 等待, 但可以被信号中断
 - Waiting, uninterruptible: 不能被信号终端 中断
 - Stopped: 停止
 - Zombie: 终止运行, 但其task_struct结构体还在, 未被回收

```
struct task_struct {  
#ifdef CONFIG_THREAD_INFO_IN_TASK  
/*  
 * For reasons of header soup (see current_thread_info()), this  
 * must be the first element of task_struct.  
 */  
    struct thread_info           thread_info;  
#endif  
    /* -1 unrunnable, 0 runnable, >0 stopped: */  
    volatile long                state; //状态
```

Linux中的进程控制块 --task_struct

①

- 调度信息

- 决定选择哪个进程运行

- 优先级

- 计数器

②

- 标识信息

- <- 进程标识符，组标识符

- 有效用户标识，有效组标识

Linux中的进程控制块 --task_struct

③ 进程间通信信息

- 信号的状态，哪些被阻止了
- 信号处理函数

```
/* Signal handlers: */
struct signal_struct
struct sighand_struct
sigset_t
sigset_t
...
.

struct signal_struct{
    int count;
    struct sigaction action[32];
}

}
```

*signal; 信号状态
*sighand; 处理器函数
blocked;
real_blocked;

Linux中的进程控制块 --task_struct

- ① 进程关系族信息 父 $\xrightarrow{\text{创建}} \text{子}$. 关系树
- 父进程、子进程等的指针

```
/* Real parent process: */
struct task_struct __rcu *real_parent;

/* Recipient of SIGCHLD, wait4() reports: */
struct task_struct __rcu *parent;

/*
 * Children/sibling form the list of natural children:
 */
struct list_head children;
struct list_head sibling;
struct task_struct *group_leader;
```

Linux中的进程控制块 --task_struct

⑤ 时间和定时信息

```
u64  
u64  
#ifdef CONFIG_ARCH_HAS_SCALED_CPUTIME  
    u64  
    u64  
#endif  
    u64  
    struct prev_cputime  
#ifdef CONFIG_VIRT_CPU_ACCOUNTING_GEN  
    struct vtime  
#endif  
  
#ifdef CONFIG_NO_HZ_FULL  
    atomic_t  
#endif  
    /* Context switch counts: */  
    unsigned long  
    unsigned long  
  
    /* Monotonic time in nsecs: */  
    u64  
    /* Boot based time in nsecs: */  
    u64  
        user  
        system  
        utime;  
        stime;  
        utimescaled;  
        stimescaled;  
        gtime;  
        prev_cputime;  
        vtime;  
        tick_dep_mask;  
        nvcs;v  
        nivcs;v  
        start_time;  
        real_start_time;
```

Linux中的进程控制块 --task_struct

⑥ 存储管理信息

struct mm_struct
struct mm_struct

*mm; |
*active_mm;
↑point to
虚拟地址空间

```
/* MM fault and swap info: this can arguably be seen as either mm-specific or thread-specific: */
unsigned long min_flt;
unsigned long maj_flt;
```

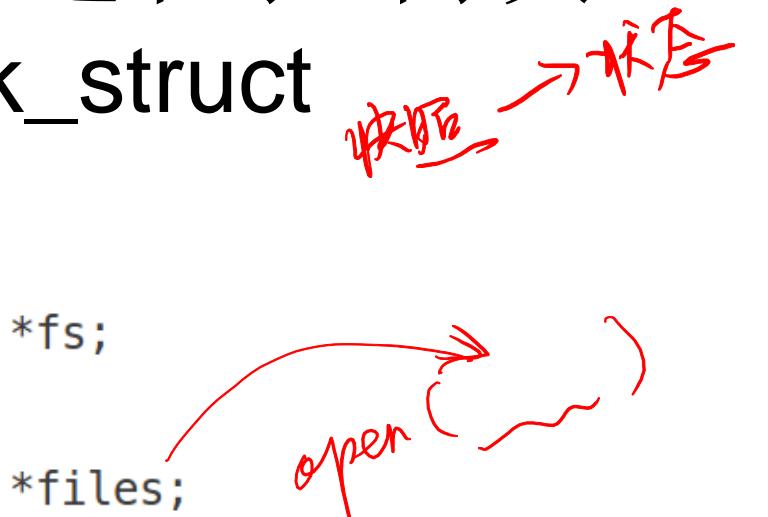
Linux中的进程控制块

--task_struct

① 文件信息

```
/* Filesystem information: */  
struct fs_struct  
  
/* Open file information: */  
struct files_struct
```

```
struct fs_struct{  
    int count; /* reserved*/  
    unsigned short umask;  
    struct inode * root, * pwd;  
    ...  
}
```

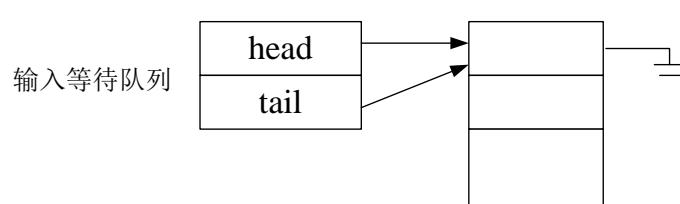
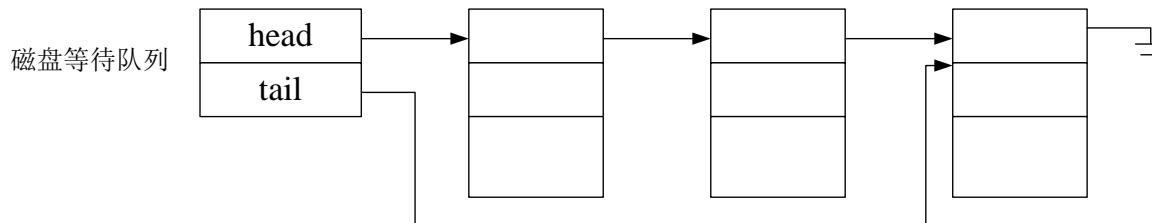
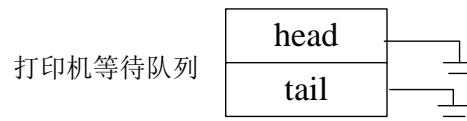
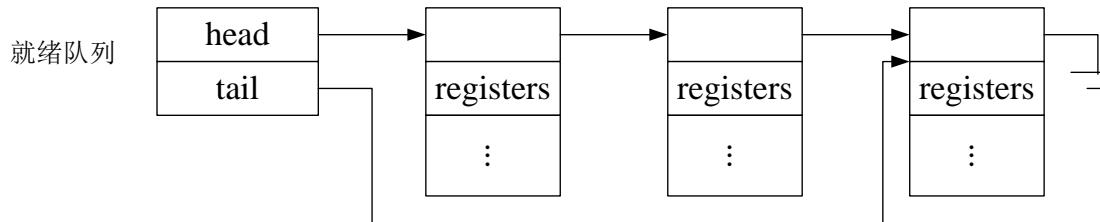


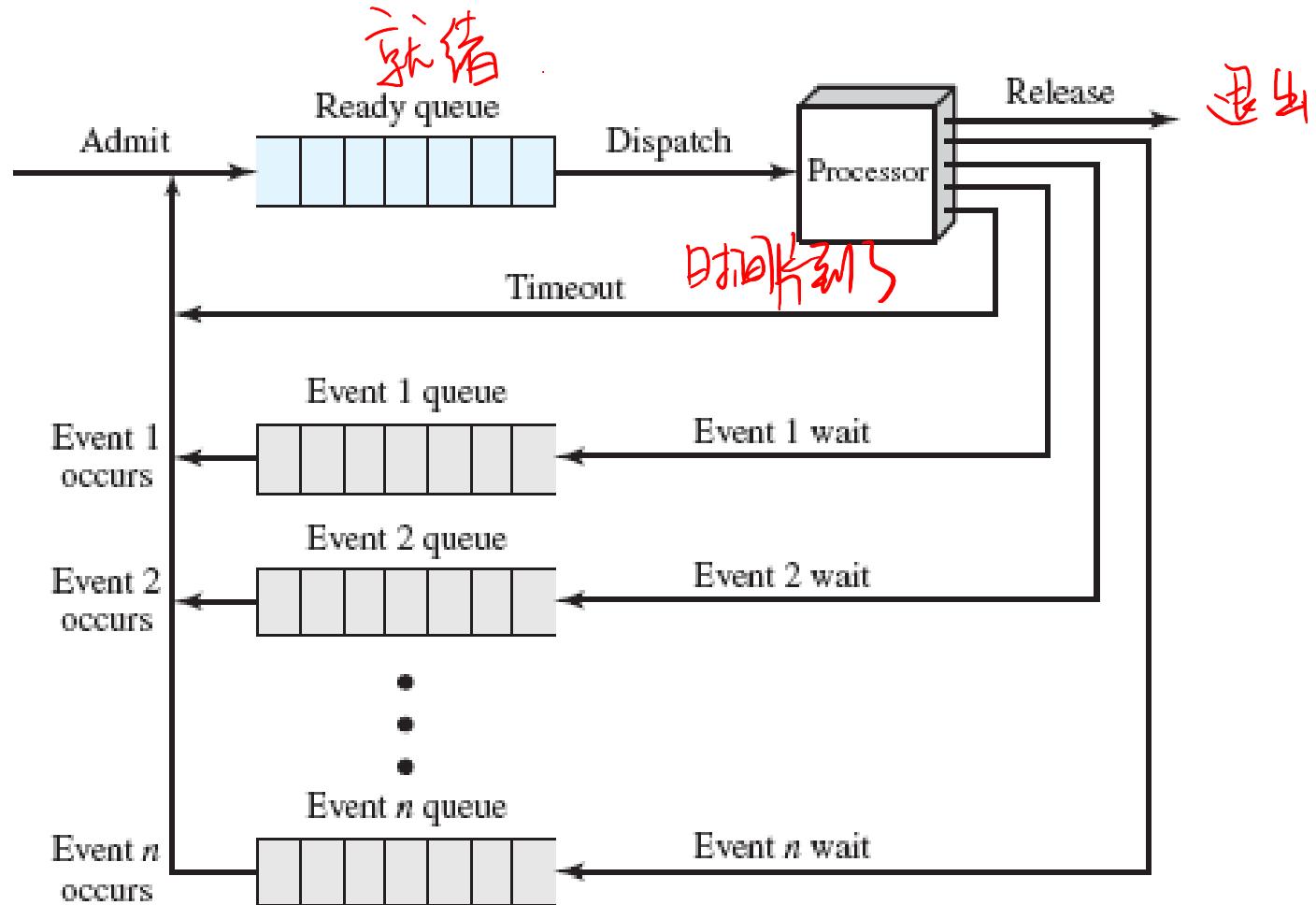
```
struct files_struct {  
    int count; /* reserved*/  
    fd_set close_on_exec;  
        /* bit map - close these on exec */  
    fd_set open_fds;  
    struct file * fd[NR_OPEN];  
        /* fds are index */  
}
```

进程队列的管理

- 逻辑上，按照进程状态分为多个组
- 物理上，可以按照下述方式管理：
 - 线性方式
 - 链接方式
 - 索引方式

进程队列的管理





(b) Multiple blocked queues

Figure 3.8 Queuing Model for Figure 3.6

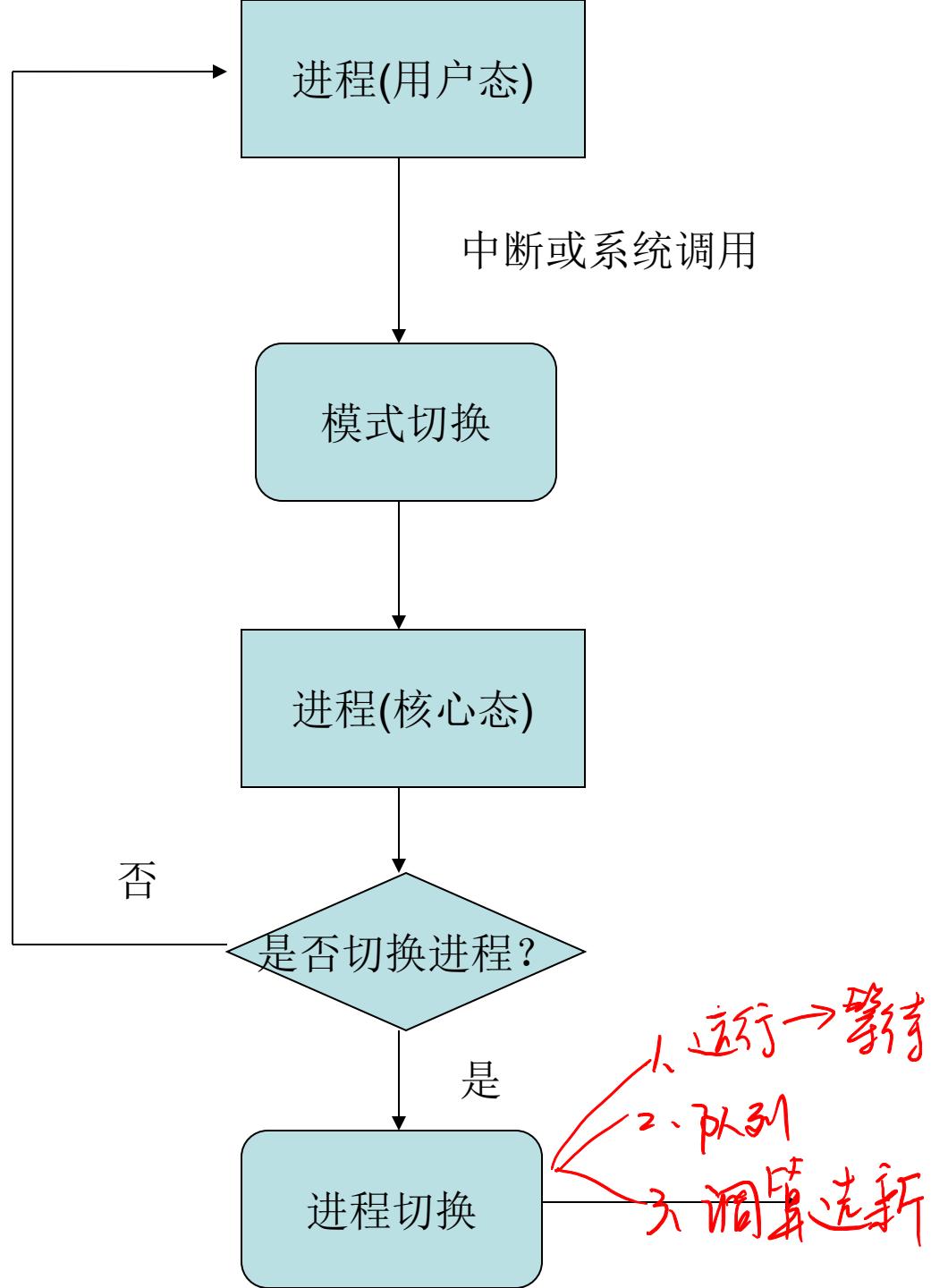
再谈进程切换

- 模式切换与进程切换的区别
- 为实现进程切换，操作系统需要干什么？

模式切换

- 模式切换
 - 当中断、异常或系统调用发生时，需要暂停当前正在运行的进程，把处理器状态从用户态切换到核心态，执行操作系统服务程序。
- 模式切换需要保存的上下文内容包括：
 - 程序计数器，处理器寄存器，栈信息；

进程还在CPU
运行



进程切换

只要那一内核

- 模式切换并不一定会引起当前进程的状态变化
- 但是若当前运行进程需要变为其它状态（就绪，阻塞等），则操作系统必须进行下面的进程切换步骤
 - 保存处理器现场，包括程序计数器和其它寄存器信息；
 - 更新当前处于运行状态进程的进程控制块信息，将其状态改为就绪、阻塞、挂起就绪等；并更新其它相关信息；
 - 将该进程控制块移动到相应的队列；
 - 基于短程调度算法，选择另一个进程执行；
 - 更新被选中进程的进程控制块；
 - 更新内存管理数据结构；
 - 恢复被选中执行的进程的上下文。

工作就绪

进程的控制和管理

- ① 进程创建
- ② 进程撤销
- ③ 进程阻塞和唤醒 *block → await*
- ④ 进程挂起和激活

① 进程创建

- 创建新进程的原因
 - 提交批处理作业
 - 有交互式作业登录终端
 - 操作系统创建服务进程
 - 已存在的进程创建新进程

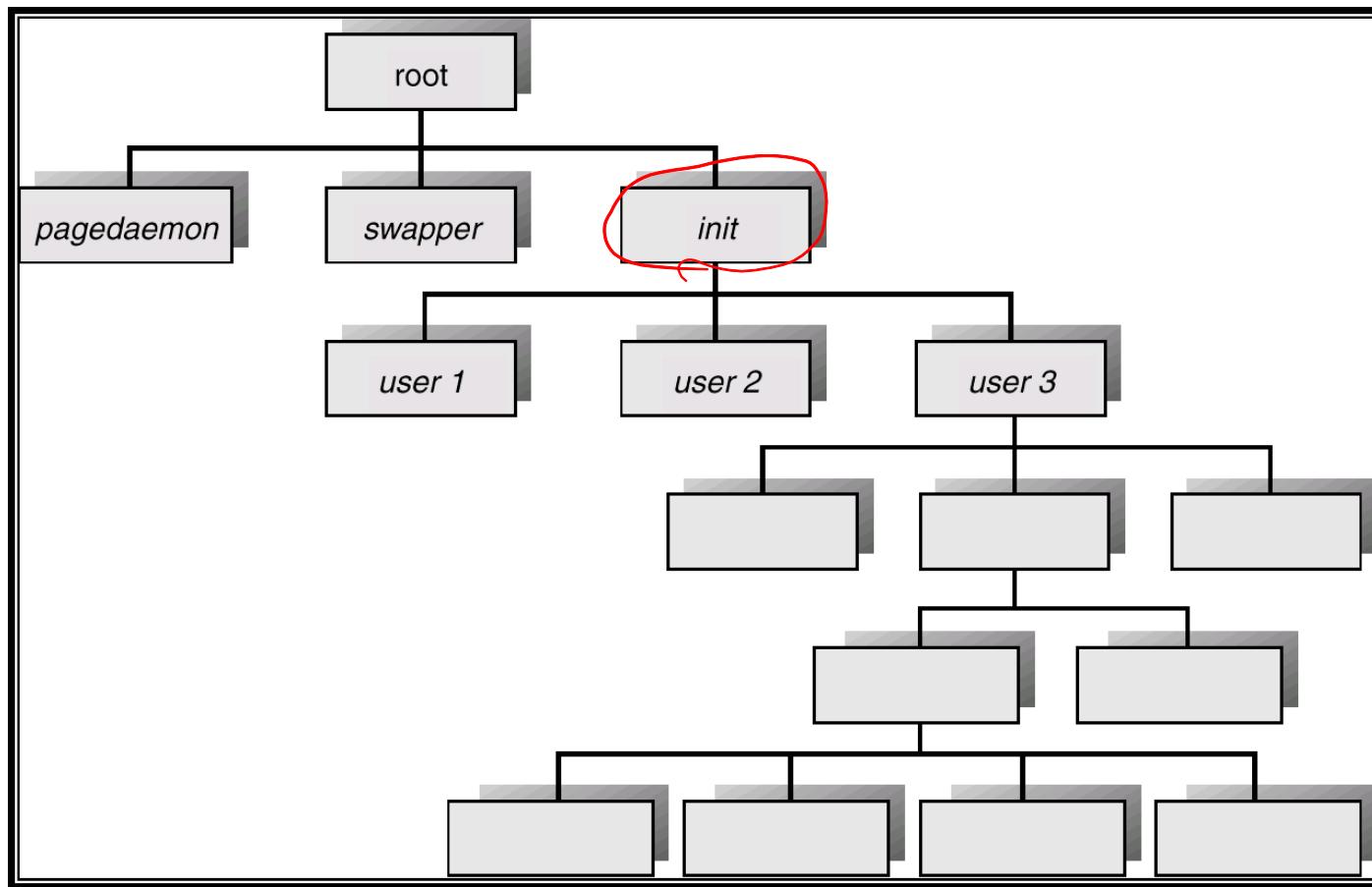
进程创建

- 创建过程
 - 在进程表中增加一项，从PCB池中申请一个空闲PCB，为新进程分配唯一的进程标识符
 - 为新进程的进程映像分配地址空间，以容纳进程实体
 - 为新进程分配除主存空间以外的各种其他资源
 - 初始化PCB
 - 把新进程的状态设为就绪态，并将其移动至就绪队列
 - 将该进程创建事件通知操作系统的某些模块

进程创建

- 进程的创建方式
 - 可以由用户通过命令接口/图形接口创建
 - 也可以由一个进程在执行中利用系统调用创建
- 进程的关系
 - A进程创建B进程，则A称为B的父进程，B称为A的子进程
 - 进程的创建关系形成一颗进程树
 - 进程用进程标识符(pid)来唯一标识 txt

UNIX系统中进程树的例子

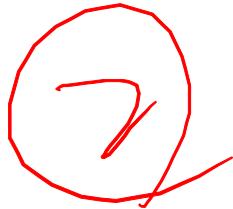


父子关系

进程创建

- 资源共享 三种方式
 - ① 父进程和子进程共享所有资源
 - ② 子进程共享父进程的部分资源
 - ③ 子进程和父进程不共享任何资源
- 执行 两种方式
 - 父进程和子进程 并发执行
 - 父进程等待子进程执行 结束后继续执行
- 地址空间
 - 子进程是父进程地址空间的 拷贝
 - 子进程在地址空间中载入新程序

废弃文件、堆、代码



进程撤销

- 进程撤销的原因
 - 进程运行结束
 - 进程执行非法指令
 - 进程在用户态执行特权指令
 - 进程的运行时间超过所分配的最大时间配额
 - 进程的等待时间超过所设定的最长等待时间
 - 进程所申请的主存空间超过系统所能提供的最大容量
 - 越界错误
 - 对共享主存区的非法使用
 - 算术错误
 - I/O操作故障
 - 父进程撤销
 - 操作系统终止

进程撤销

- 步骤
 - 根据进程标识号从相应的队列中移除它
 - 将进程所拥有的资源归还给父进程或操作系统
 - 若此进程拥有子进程，先撤销其所有子进程，以防止它们脱离控制
 - 回收PCB，并将其归还至PCB池

③

进程终止

-) 进程执行完最后一条指令后，要求操作系统删除该进程(**exit()**系统调用)
 - 向父进程返回状态信息 (**via wait**)
 - 操作系统回收进程资源
-) 父进程可以终止子进程的执行
 - 子进程使用的资源超出了其分配的数量
 - 赋给子进程的任务不再需要执行
 - 父进程结束，某些操作系统不允许子进程在父进程终止后继续执行

进程阻塞和唤醒

- 进程阻塞的步骤
 - 停止进程执行，将现场信息保存到PCB
 - 修改进程PCB的有关内容，如进程状态从运行态变为等待态
 - ~~把进程移入相应时间的~~^{事件}等待队列中
 - 转入进程调度程序，调度其他程序执行
- 进程唤醒的步骤
 - 从相应的等待队列中移出进程
 - 修改进程PCB的相关内容，如进程状态改为就绪态，并将进程移入就绪队列
 - 若被唤醒的进程比当前运行进程优先级高，且采用抢占式调度，则进行进程调度

进程的挂起和激活

- 进程挂起的步骤
 - 改进进程的状态，如为活动就绪态，改为挂起就绪态，如为等待态，改为挂起等待态
 - 将被挂起进程 **PCB** 非常驻部分交换到 ~~磁盘对换区~~
仍在内存中 *归宿*
- 进程激活的步骤
 - 将被挂起进程 **PCB** 非常驻部分调入内存
 - 改进进程的状态，并将进程移入相应的队列

Linux进程创建

- 进程创建的函数
 - pid_t fork(void)
 - pid_t vfork(void) *线程 父数*
 - pid_t clone(int (*fn)(void *arg), void *stack, int flags, void *arg); *灵活!*
- fork()系统调用 *返回 进程指针*
 - 用户创建新进程
 - 语法

newpid = fork();

返回时，两个进程拥有相同的用户级上下文，除了返回的newpid不一样。

 - 父进程中, newpid = 子进程的pid
 - 子进程中, newpid = 0

父进程和子进程均从fork()之后开始执行
- exec()系统调用用于在fork()之后载入新程序

fork()

- 父进程的所有资源通过数据结构复制传给子进程
 - 父进程与子进程共享同一个代码段
 - 父进程将数据段和堆栈段复制一份给新进程，父进程和子进程不共享数据
 - 为了提高效率，实际采用的是写时复制 ← 要修改才复制
- fork()一次调用，两次返回
 - 父进程中，fork()的返回是子进程的pid号
 - 子进程中，fork()返回0
 - 返回值用于区分父进程和子进程
- 父进程和子进程均从fork()调用后开始并发执行

进程运行新程序

- ~~exec~~ 系统调用 用于执行新程序
 - 把代码段 替换成新程序的代码段
 - 废弃原有的数据段和堆栈段, 为新程序分配新的数据段和堆栈段
 - 唯一留下的是进程号
- fork()/exec() 组合使用产生新进程, 执行新任务
- wait() 和 waitpid() 用于等待进程结束
 - wait() 用于等待任何一个子进程的结束
 - waitpid() 用于等待给定参数为 pid 的进程结束

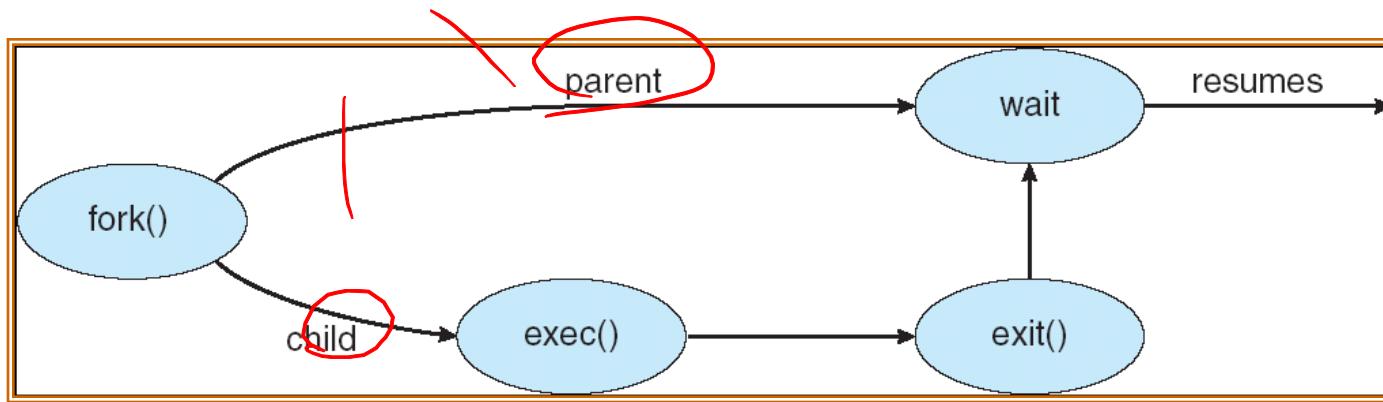
fork() + exec()

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        printf("This is child process, pid=%d\n", getpid());  
    子程序
        execp("/bin/ls", "ls", NULL); 到当前目录下执行
        printf("Child process finished\n"); 这句话不会被打印，除非execp调用未成功*/
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        printf("This is parent process, pid=%d\n", getpid());
        wait(NULL);
        printf ("Child Complete");
        exit(0);
    }
}
```

代码输出成 ls, 原程序废弃

fork() + exec()

先后顺序不清楚



This is child process

(LS输出结果)

(This is parent process)

Child Complete.

vfork()

- vfork()创建的子进程与父进程共享地址空间，子进程对虚拟地址空间的任何数据修改同样为父进程所见
- vfork()创建子进程后，父进程被阻塞，直到子进程执行exec()或exit()为止。
- 如果创建子进程的目的是为了执行新程序，则fork()对地址空间的复制是多余的，vfork()可以减少开销

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void){
    int data = 0; // 什么值
    pid_t pid;
    pid=vfork(); /*如果是pid=fork(), 结果如何*/
    if(pid<0){
        perror("Error\n");
        exit(-1);
    }
    if(pid==0){ /*Child Process*/
        printf("In child process, pid=%d\n", getpid());
        data++;      //修改同一份地址空间
        exit(0);
    }
    wait(NULL);
    if(pid>0){ /*Parent process*/
        printf("In parent process, pid=%d, data is %d\n", getpid(), data);
    }
}

```

vfork 1
 fork 0

eg. Word后台保存文件

线程

- 传统意义上的进程所具备的特征
 - 资源分配的主体 CPU、存储空间、I/O设备、文件句柄...
 - 调度执行的主体
- 上述两个特征相互独立，操作系统可以对其进行区别对待。
- 通常，调度执行的单元被称为线程或轻量级进程；而资源分配的主体则被称为进程。

多线程

共享地址空间

- 多线程指的是操作系统在一个进程中支持多个并发的执行路径的能力。
- 线程是进程中能够并发执行的实体，是处理器调度和分派的基本单位。
- 线程的组成部分
 - 线程标识符
 - 线程执行状态
 - 线程不运行时的线程上下文
 - 核心栈
 - 与每个线程相关的局部变量

多线程环境中的进程定义

- 操作系统中进行除处理器以外的资源分配和资源保护的基本单位
 - 独立的虚拟地址空间，所有线程都共享该空间
 - 以进程为单位进行资源保护，如受保护地访问处理器、文件、外设和其他进程

进程 vs 线程

- 进程
 - 资源集合
 - 线程集合
- 进程提供线程运行的支撑环境
 - 地址空间
 - 共享资源的管理
 - 全局数据
 - 指令代码
 - 打开文件表
 - 信号量 *协同工作*
- 线程封装执行信息
 - 执行状态
 - 处理器现场信息，如寄存器
 - 局部数据
 - 执行栈
 - 线程调度信息，如优先级
 - . . .

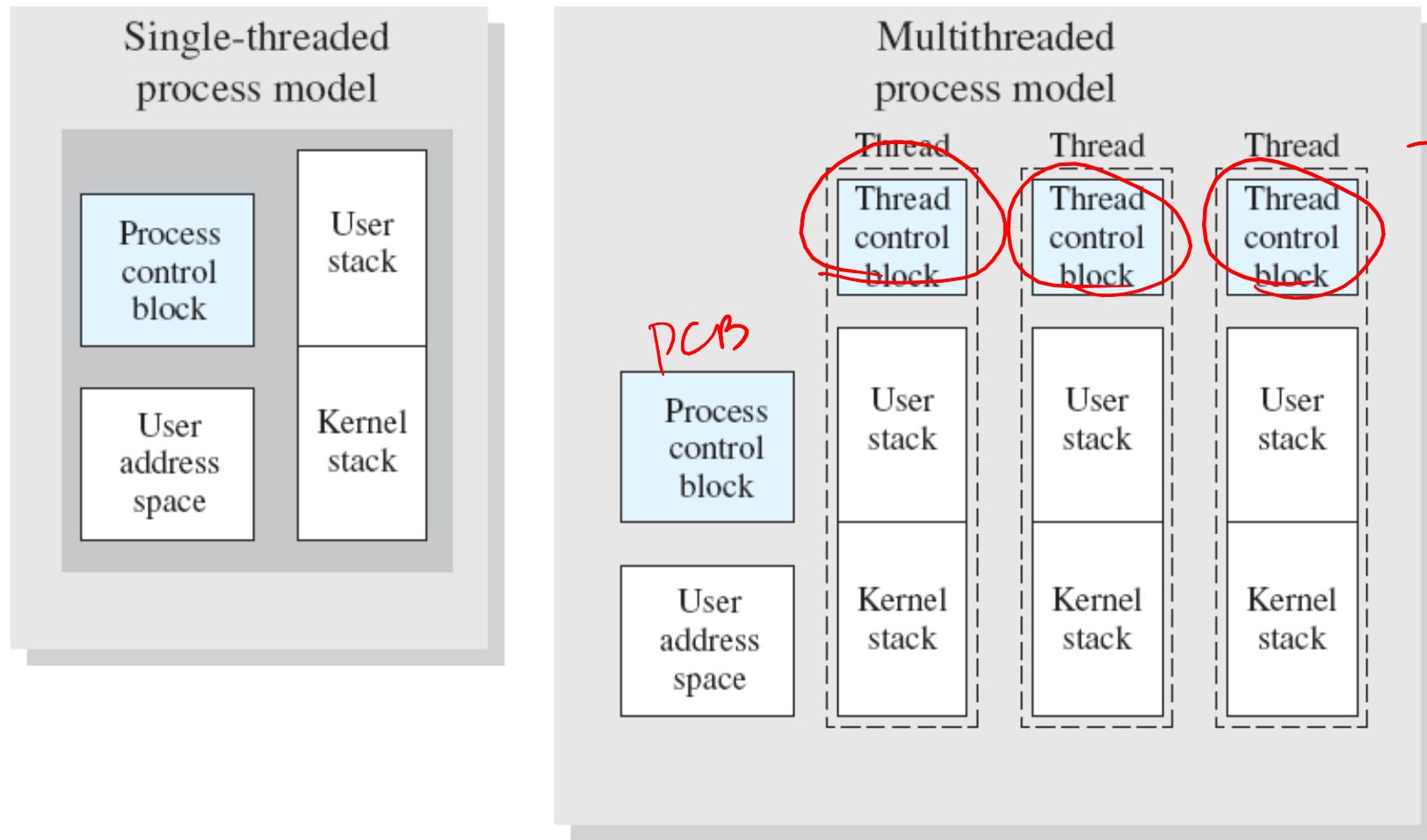


Figure 4.2 Single Threaded and Multithreaded Process Models

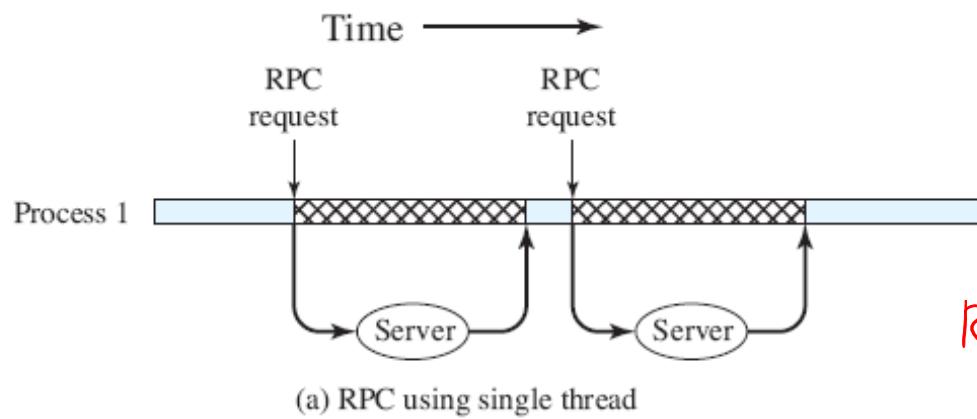
线程状态

- 与进程类似
 - 运行、就绪、等待
 - 挂起状态对线程无意义 (why?)
 - 线程不是资源的拥有单位, 内存地址空间 —> 不独立 进程才是
 - 进程被挂起后对换出主存将引起所有线程也对换出主存
- 多线程管理的问题
 - 当处于运行态的线程在执行过程中需要阻塞自己时, 是否可以调度本进程的其它线程执行? 还是需要阻塞整个进程?
 - 依赖于实现机制: 用户级线程 or 内核级线程

e.g. 同时读5个文件：
① 创建5个进程
② 一个进程创建5个线程

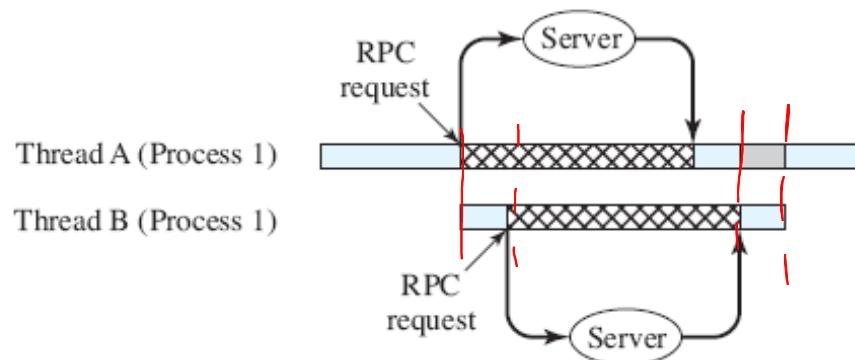
采用线程的优点

- 在已有进程创建线程要比创建一个新进程快得多；
- 线程的终止比进程的终止要快；
- 同一进程的线程切换快于不同进程的进程切换；
- 线程提高了不同执行程序之间的通信效率。大部分操作系统中，不同进程之间的通信需要内核的干预，而线程之间的通信无需内核的干预（共享存储）；
 - 共享存储也引发了线程同步的问题
- 提高了并发程度



RPC
远程过程调用

Receive = 阻塞型



等待操作可以同时发生。

(b) RPC using one thread per server (on a uniprocessor)

XXXX Blocked, waiting for response to RPC

■ Blocked, waiting for processor which is in use by Thread B

■ Running

Figure 4.3 Remote Procedure Call (RPC) Using Threads

一个进程需要进行两次远程过程调用，并组合其结果，得出最终结果

多线程的例子

- Word中的自动保存
 - 编辑和自动保存两个任务共享全局数据，即所打开的文件
- HTTP服务器
 - 等待 80 端口 Request.*
 - 迎宾 一个线程专门负责侦听连接
 - 服务器 为每个连接生成一个独立的线程与用户进行交互

单线程HTTP服务器

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Entry
{
    public static void main(String[] args)
    {
        try
        {
            ServerSocket server = new ServerSocket(8080);
            // Listen at the specified port.

            for (;;)
            {
                阻塞等TCP连接
                Socket client = server.accept(); // Accept an incoming connection.

                // Write "Hello, world!" to the client. What if a large amount of work should be done?
                // the sever will not be able to respond to new connections promptly!!!
                client.getOutputStream().write("Hello, world!\r\n".getBytes());

                client.close(); // Close the connection.
            }
        }
        catch (IOException e)
        {
            System.err.println("Error binding the specified port.");
        }
    }
}
```

多线程HTTP服务器

服务

迎宾(主)

```
import java.io.IOException;
import java.net.Socket;
public class Service extends Thread
{
    private Socket _socket; // 类里面的成员变量
    public Service(Socket socket)
    {
        _socket = socket;
    }

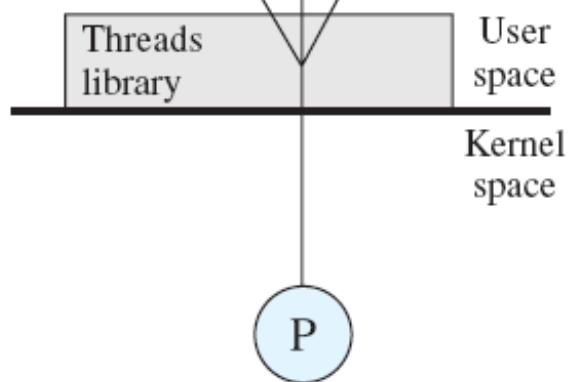
    public void run()
    {
        try{
            // Write "Hello, world!" to the client.
            _socket.getOutputStream().write("Hello,
world!\r\n".getBytes());
        }
        catch (IOException e) {
            // Abandon the current connection.
        }
        finally {
            try {
                // Close the connection.
                _socket.close();
            }
            catch (IOException e)
            {
                // Eat the IOException.
            }
        }
    }
}
```

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
public class Entry
{
    public static void main(String[] args)
    {
        try
        {
            ServerSocket server = new ServerSocket(8080);
            // Listen at the specified port.
            while for (;;)
            {
                Socket client = server.accept();
                /* Create a Service Thread to serve the client. Then the
                service thread and the listen thread can run concurrently,
                so new connections will be responded promptly.*/
                Service service = new Service(client); // 增加服务线程
                service.start();
            }
        }
        catch (IOException e){
            System.err.println("Error binding the specified port.");
        }
    }
}
```

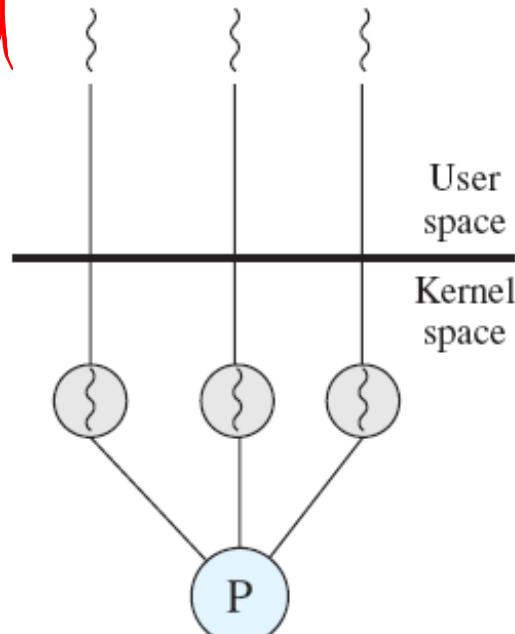
线程的实现

- 用户级线程
- 内核级线程
- 混合式线程

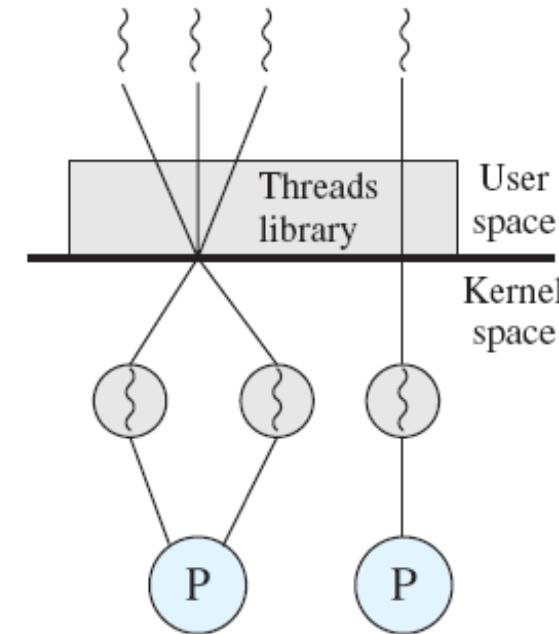
*- 一个线程被阻塞
导致整个进程被阻塞*



(a) Pure user-level



(b) Pure kernel-level



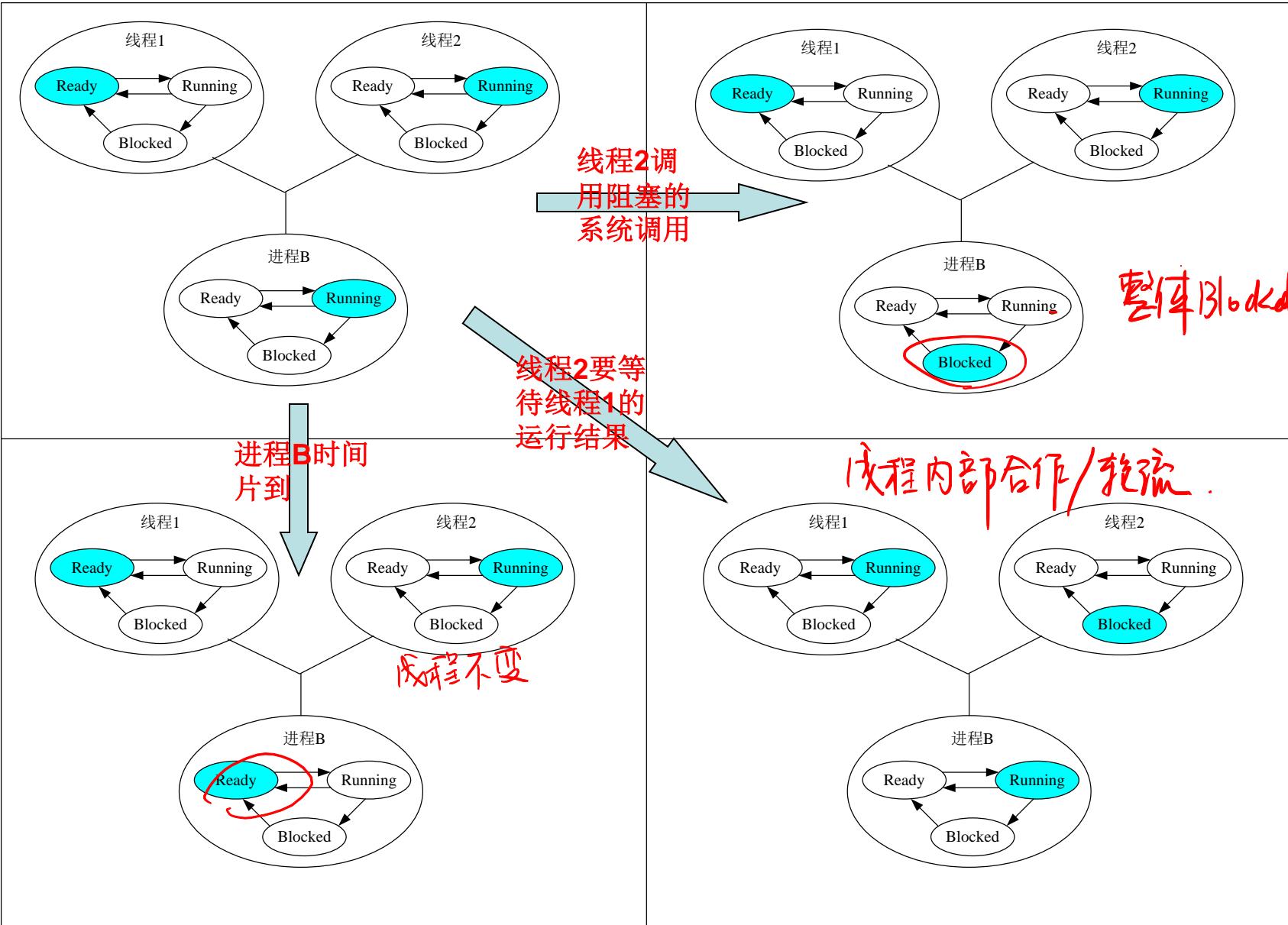
(c) Combined

{ User-level thread { Kernel-level thread P Process

Figure 4.6 User-Level and Kernel-Level Threads

用户级线程

- 线程管理的工作由线程库完成,负责:
 - 线程创建/销毁
 - 线程调度
 - 线程间消息传递
 - 保护线程上下文
- 内核不知道线程的存在，内核的调度仍以进程为单位；
- 内核赋予进程一个执行状态，而线程可能处于不同的执行状态
 - 线程的状态独立于进程的状态



用户级线程示例，进程B中有两个线程1和线程2

用户级线程

- 优点 线程库管理
 - 线程切换无需核心态支持，因为所有线程管理的数据结构均在用户地址空间中；
 - 线程调度独立于CPU调度，进程调度每个应用都可以实现自己的线程调度方法；
 - 可以运行在任何操作系统上，内核无需做任何改变。
- 缺点
 - 在典型的操作系统中，许多系统调用是阻塞型的，因此一个用户级线程发起的阻塞型系统调用将引起进程的所有线程被阻塞；
 - 由于内核只支持进程调度，仅将一个进程分配到一个处理器上执行，因此，同一进程的多个线程无法利用多处理器的优势。表现出来的只是多线程在一个处理器上的并发执行。并发执行

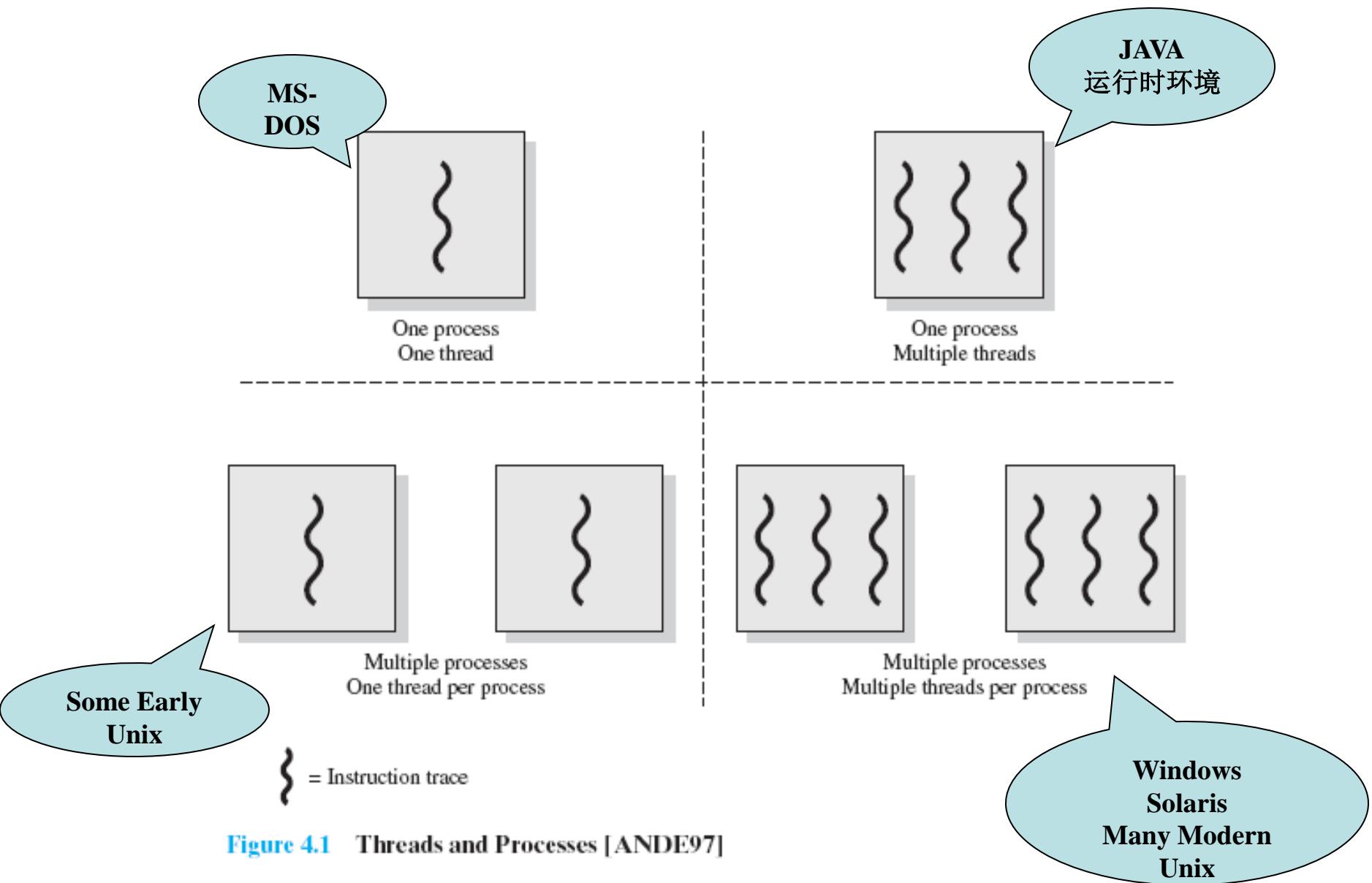
内核级线程

- 所有线程管理的工作都由内核完成
- 用户通过内核提供的线程API来使用线程
- 内核为每个进程维护信息，并且为进程中的每个线程维护信息。
 - 进程控制块PCB
 - 线程控制块TCB
- 内核的调度以线程为单元。
- 优点：
 - 同一进程的不同线程可以在多个处理器上并行执行
 - 进程中一个线程的阻塞不会引起同一进程其它线程的阻塞。
- 缺点
 - 同一进程之间的线程切换需要进行模式切换，开销较大
只会阻塞当前线程

混合模式

映射

- 线程的实现分为：用户层和核心层
- 线程在用户空间创建，并是用户程序中调度和同步的单元
- 同一应用的多个用户级线程被映射到多个（相同数目或更少）内核级线程。
 $5 \rightarrow 1$
- 程序员可以调整每个特定应用的内核线程数
 $1 \rightarrow 1$
 - 逻辑并行性较高的应用，可以将一组用户级线程映射到一个内核级线程
 多对一
 - 物理并行性较高的应用，可以让每个用户级线程绑定一个内核级线程
 一对一



线程API

- POSIX Pthread Linux
- Win32 thread API
- Java API

Pthreads

- Pthreads是POSIX的线程创建和同步的接口标准
- Pthreads定义了线程的行为，并未规定具体的实现方式
- 许多操作系统都实现了Pthreads标准
 - Solaris
 - Linux
 - Mac OS X
 - Tru64 UNIX

创建线程

- int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*start_rtn)(void), void *arg)
 - tid: 要创建的线程的线程id指针 [输出]
 - attr: 创建线程时的线程属性, 如不需要, 可设为NULL;
 - start_rtn: 返回值是void类型的指针函数, 是新线程执行的函数
(无参, 返回值任意类型)
 - arg: 传递给start_rtn的参数
任何类型参数都能传

线程等待

- int pthread_join(pthread_t thread, void **status);
 - **thread**: 指定要等待的线程的线程标识符
 - **status**: 用于存放线程采用**pthread_exit()**返回时的返回值

返回值上一个地址

```

#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the threads */ 主线程 & runner线程
void* runner(void* param); /* the thread */ 产卵

int main(int argc, char *argv[]) 命令行参数
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */
    if(argc!=2){ 没给参数
        fprintf(stderr, "Usage: %s <integer value>\n", argv[0]);
        return -1;
    }
    if(atoi(argv[1])< 0){
        fprintf(stderr, "%d must be >=0\n", atoi(argv[1]));
        return -1;
    }
    pthread_attr_init(&attr); /* get the default attributes */
    pthread_create(&tid, &attr, runner, argv[1]); /* create the thread */
    /* the main thread can do other work parallely with the new thread
    here*/
    pthread_join(tid, NULL); /* wait for the thread to exit */
    printf("sum= %d\n", sum);
}

```

```

/* The thread will begin
control in this function */

void *runner(void *param)
{
    int i, upper=atoi(param);
    sum = 0;
    for(i = 1; i <= upper; i++)
        sum+=i;
    pthread_exit(0);
}

```

Win32 API

- 与Pthreads API类似

Java线程API

- 创建线程的两种方式
 - 直接从**Thread**类继承，并改写其**run()**方法
 - 创建一个实现了**Runnable**接口的对象，并将其作为传递给**Thread**构造函数的参数；调用**Thread**对象的**start()**方法，启动线程

```
public interface Runnable  
{  
    public abstract void run();  
}
```

//方法一： 继承Thread 类， 实现run方法。

```
class ThreadTest extends Thread {  
    ThreadTest() {}  
    public void run() {  
        // code here ...  
    }  
}
```

//创建并启动一个线程：

```
ThreadTest t = new ThreadTest();  
t.start();
```

//方法二：实现 Runnable 接口的类，实现run方法。

```
class RunTest implements Runnable {
```

```
    RunTest () {}
```

```
    public void run() {
```

```
        // code here . . .
```

```
    }
```

```
}
```

//创建并启动一个线程：

```
RunTest r = new RunTest();
```

```
new Thread(r).start();
```

—— 创建函数参数