

## 2、进程管理

---

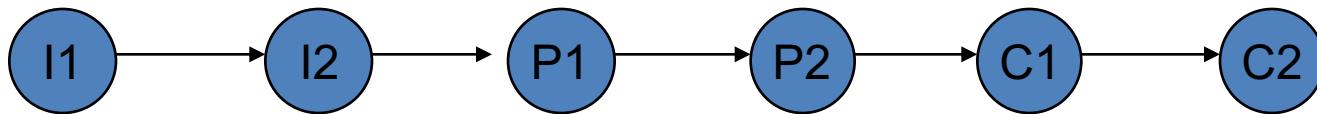
薛瑞尼

计算机科学与工程学院

16/3/19

# 进程的基本概念

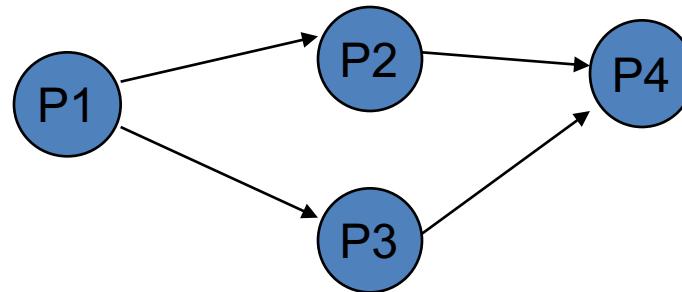
- 程序的顺序执行及特征
  - 程序执行有固定的时序



- 特征：
  - 顺序性、封闭性、可再现性

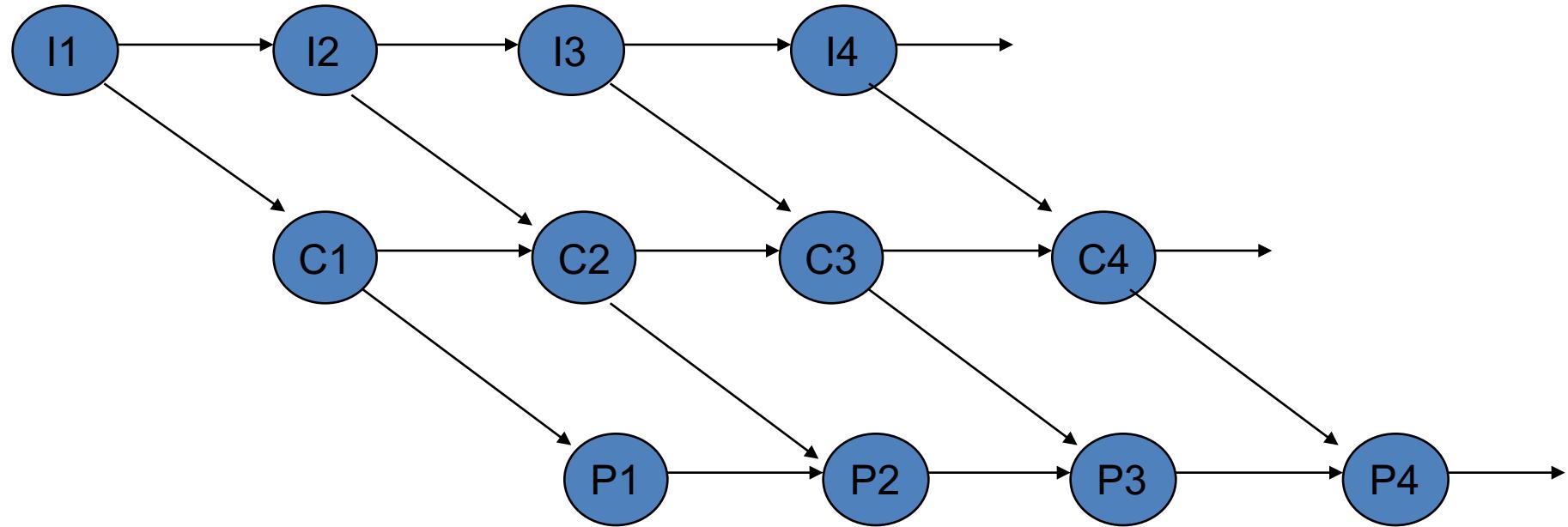
# 前趋图定义

- 有向无循环图
- 表示方式
  - $p_1 \rightarrow p_2$
  - $\rightarrow = \{(p_1, p_2) | p_1 \text{ 必须在 } p_2 \text{ 开始前完成}\}$
- 节点表示：一条语句，一个程序段，一进程。



# 程序的并发执行

- 多个程序的并发执行（可能性分析）



# 程序的并发执行(2)

- 特征
  - 间断性
  - 失去封闭性：主要由共享资源引起
  - 不可再现性：设N的初值为n。
  - 有2个循环程序A和B，它们共享一个变量N，

A

```
N := N + 1
```

B

```
print(N);  
N := 0;
```

# 程序的并发执行(3)

- $N := N + 1$  在  $\text{print}(N)$  和  $N := 0$  之前, 则  $N$  分别为
    - $n+1, n+1, 0$
  - $N := N + 1$  在  $\text{print}(N)$  和  $N := 0$  之后, 则  $N$  分别为
    - $n, 0, 1$
  - $N := N + 1$  在  $\text{print}(N)$  和  $N := 0$  之间, 则  $N$  分别为
    - $n, n+1, 0.$
- A  $N := N + 1;$
  - B  $\text{print}(N);$
  - $N := 0$

# 并发程序

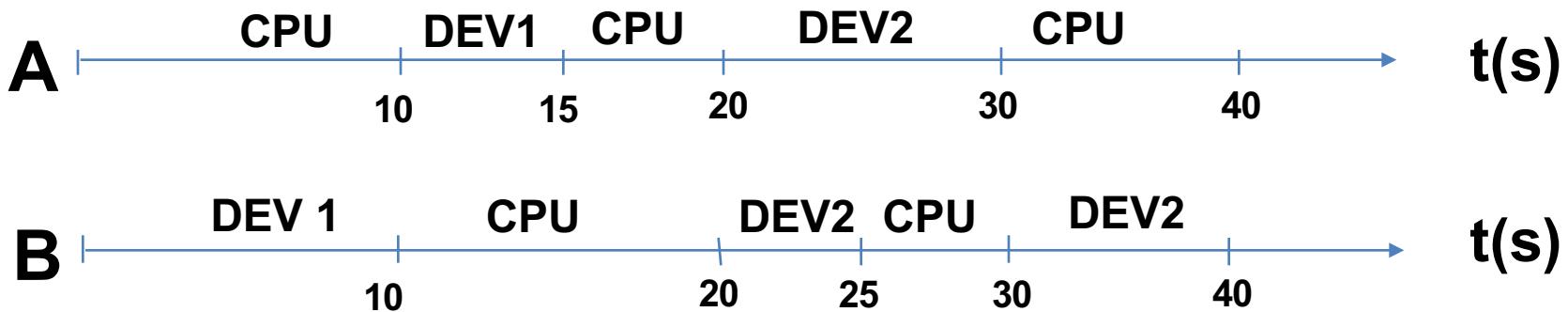
- 资源共享
  - 系统中资源被多个程序使用
- 独立性和制约性
  - 独立的相对速度、起始时间
  - 程序之间可相互作用（相互制约）
- 程序和计算不再一一对应
  - 计算：一个程序的执行

# 并发程序 (1/3)

- 一定时间内，物理机器上有两个或两个以上的程序同时处于开始运行但尚未结束的状态，并且次序不是事先确定的
- 引入并发的目的
  - 为了提高资源利用率，从而提高系统效率

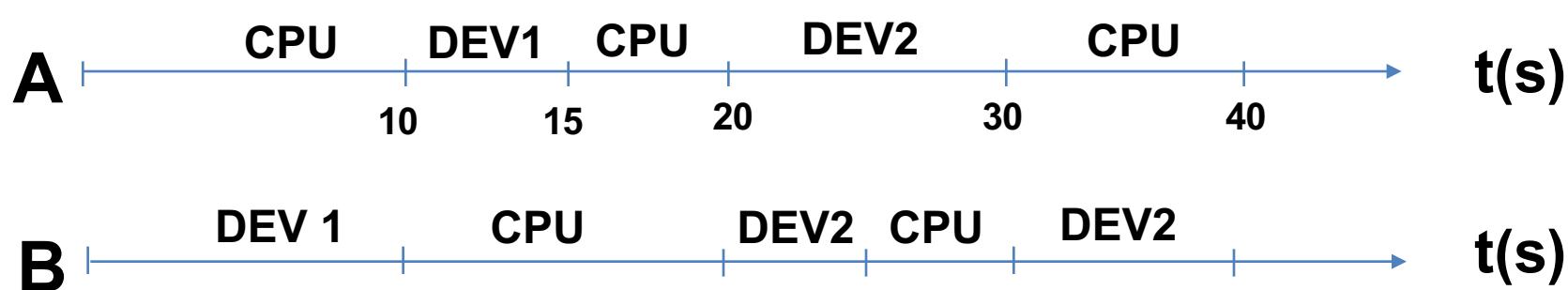
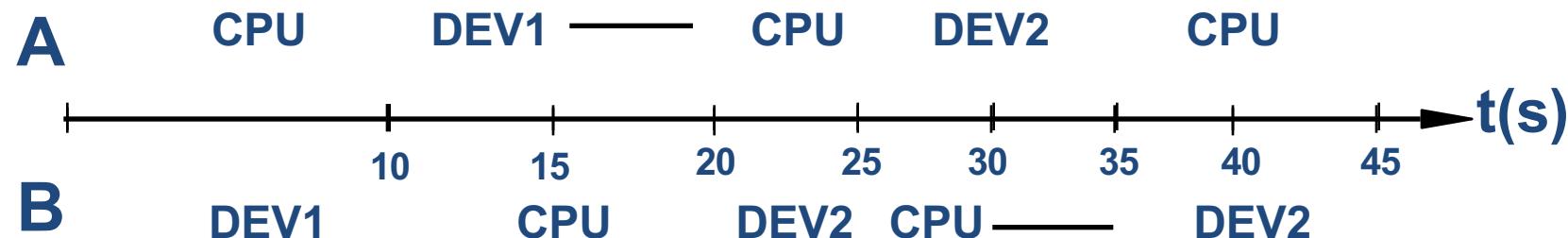
# 并发程序 (2/3)

- 在顺序环境下，A先执行，B再执行
- CPU利用率=  $40/80 = 50\%$
- DEV1利用率=  $15/80 = 18.75\%$
- DEV2利用率=  $25/80 = 31.25\%$



# 并发程序 (3/3)

- 在并发环境下 CPU利用率 = 89%
- DEV1并发环境下利用率= 33%
- DEV2并发环境下利用率= 66%



# 进程-定义和特征

- 一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。
  - 简言之，进程是程序的一次执行活动。
- 进程描述了程序的动态执行过程；
- 对应处理机、存储器和外设等资源的分配和回收；
- 反映系统中程序执行的并发性、随机性和资源共享
- 多进程，提高了对硬件资源的利用率，但又带来额外的空间和时间开销，增加了OS 的复杂性。

# 进程-定义和特征

- 动态性：
  - 进程对应程序的执行
  - 进程是动态产生，动态消亡的
- 独立性：
  - 各进程的地址空间相互独立，除非采用进程间通信手段；
- 并发性：
  - 任何进程都可以同其他进程一起向前推进
- 异步性：
  - 每个进程都以其相对独立的不可预知的速度向前推进
- 结构化：
  - 进程 = 代码段 + 数据段 + PCB

# 进程-与程序的区别

- 进程是动态的，程序是静态的：程序是有序代码的集合；通常对应着文件、静态和可以复制。进程是程序的执行。
- 进程是暂时的，程序是永久的：进程是一个状态变化的过程，程序可长久保存。
- 进程与程序的组成不同：进程的组成包括程序、数据和PCB(进程控制块)。

# 进程-与程序的区别

- 进程更能真实地描述并发，而程序不能
- 进程可创建其他进程，而程序没有
- 同一程序同时运行于若干个数据集合上，它将属于若干个不同的进程。也就是说同一程序可以对应多个进程
- 进程是资源申请和系统调度的基本单位

# 其它定义

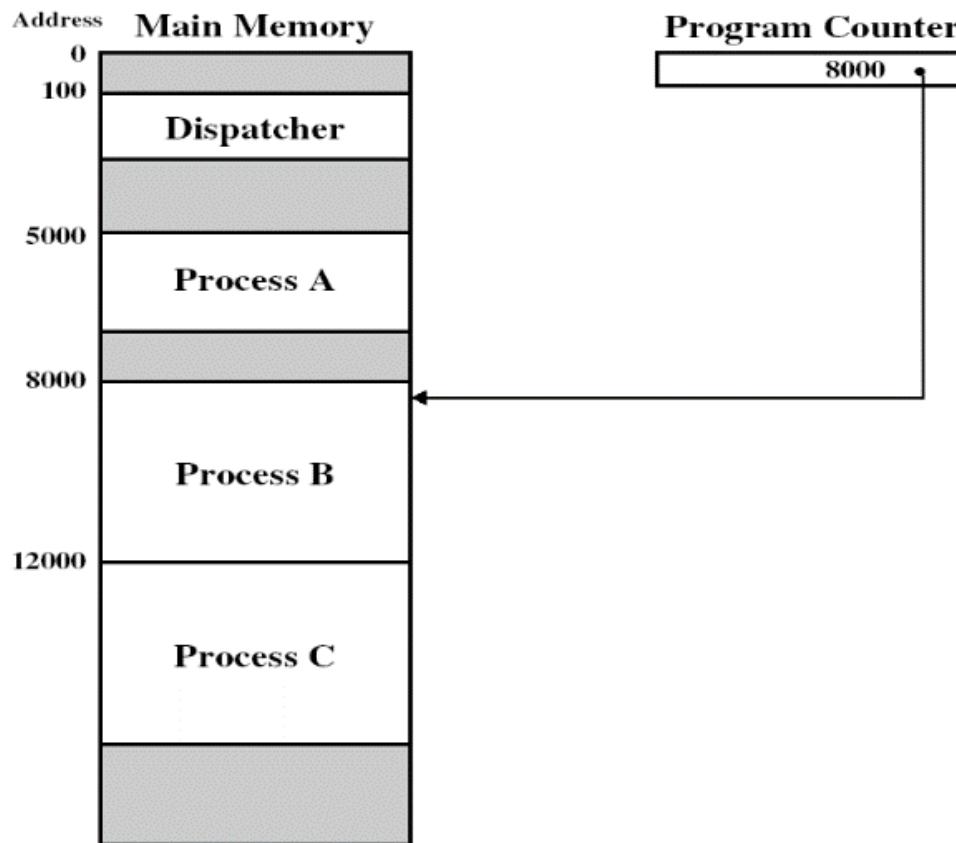
- 进程是程序的一次执行。
- 进程是一个程序及其数据在处理机上顺序执行时所发生的活动。
- 进程是程序在一个数据集合上运行的过程，它是系统进行资源分配和调度的一个独立单位。

# 问题1

- 引入进程后带来的挑战 (challenges) ?
  - 空间开销 (space overhead)
  - 时间开销 (time overhead)
  - 控制复杂性 (complexity of control)
  - .....

# 进程 - 并发示例

- 3个进程并发执行的图示



# 进程的轨迹

- 进程的执行指令序列，用于描述单个进程的行为。

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

(a) Trace of Process A

(b) Trace of Process B

(c) Trace of Process C

5000 = Starting address of program of Process A

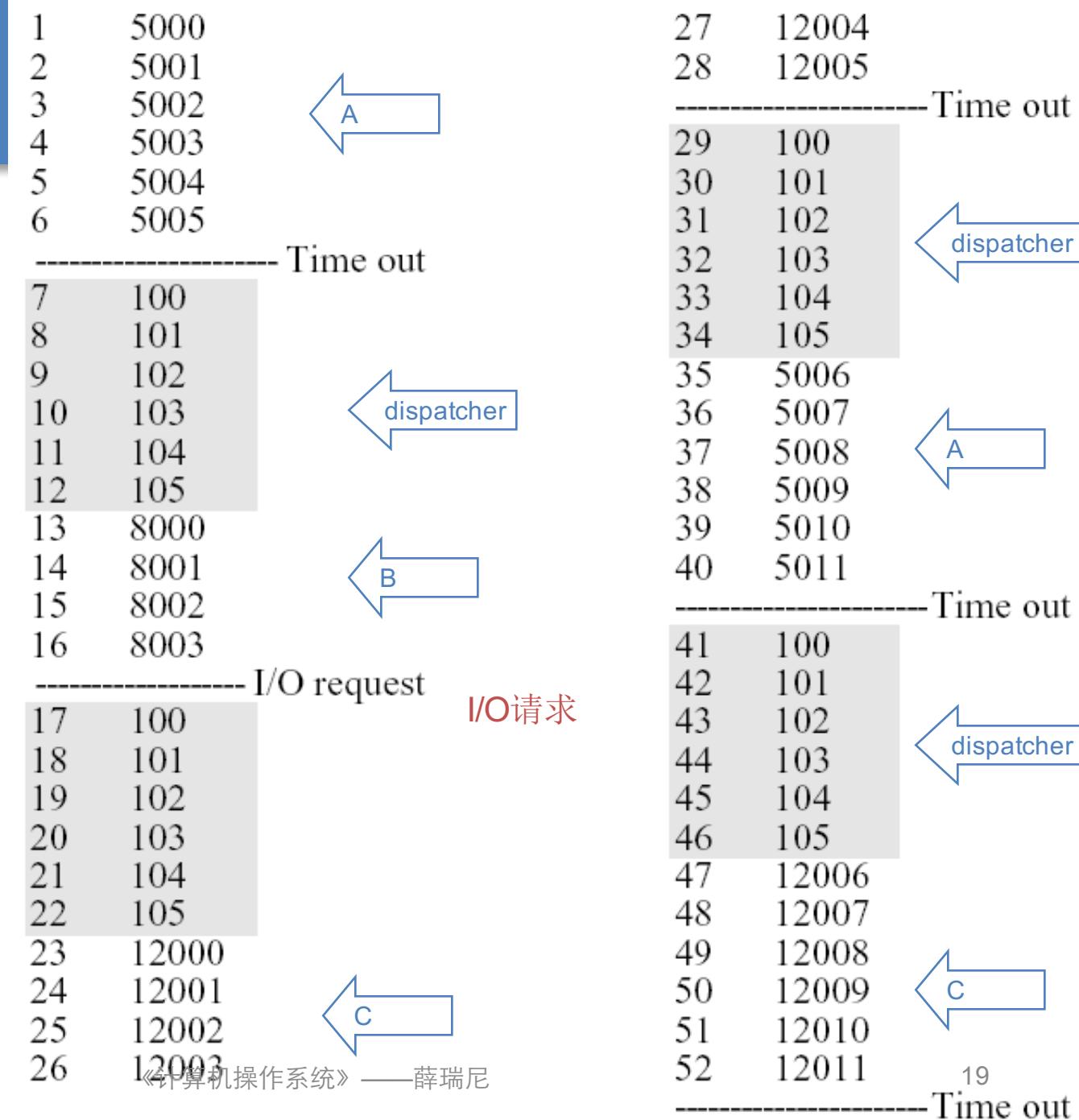
8000 = Starting address of program of Process B

12000 = Starting address of program of Process C

# 3进程并发执行的轨迹：理解处理器的行为，如何在三个进程间交替执行

规定：每个进程仅允许最多连续执行6个指令周期，之后被中断(避免独占)

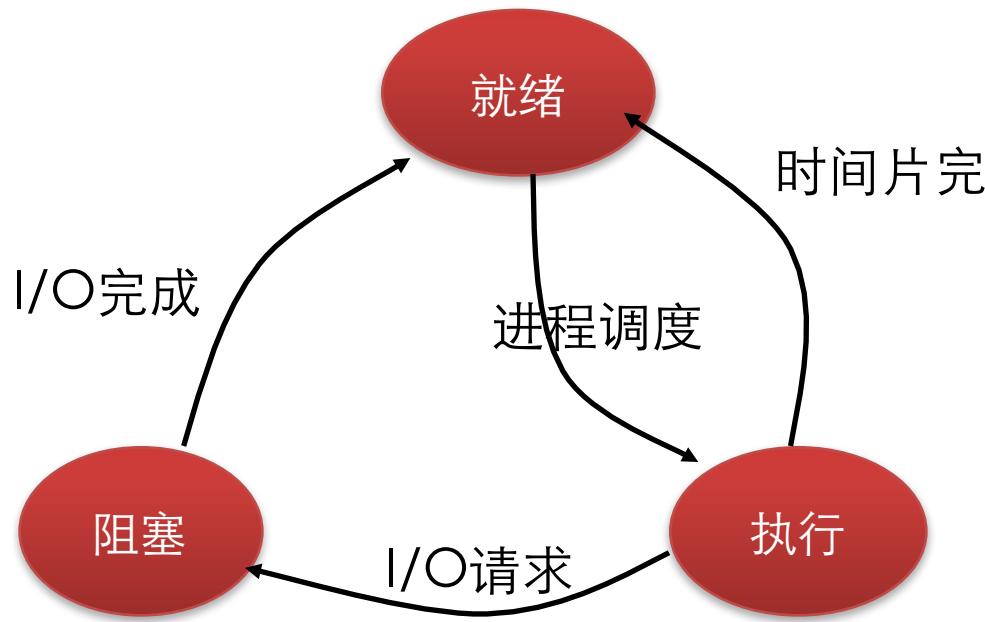
3/19/16



# 进程的特征和状态

- 进程的三种基本状态
  - 就绪态（Ready）：一个进程已经具备运行条件，但由于无CPU暂时不能运行的状态，当调度给其CPU时，立即可以运行。位于“就绪队列”中
  - 执行态（Running）：进程占有了包括CPU在内的全部资源，并在CPU上运行
  - 等待态(阻塞态，Waiting/Blocked)：指进程因等待某种事件的发生而暂时不能运行的状态（即使CPU空闲，该进程也不可运行）。位于“等待队列”中。

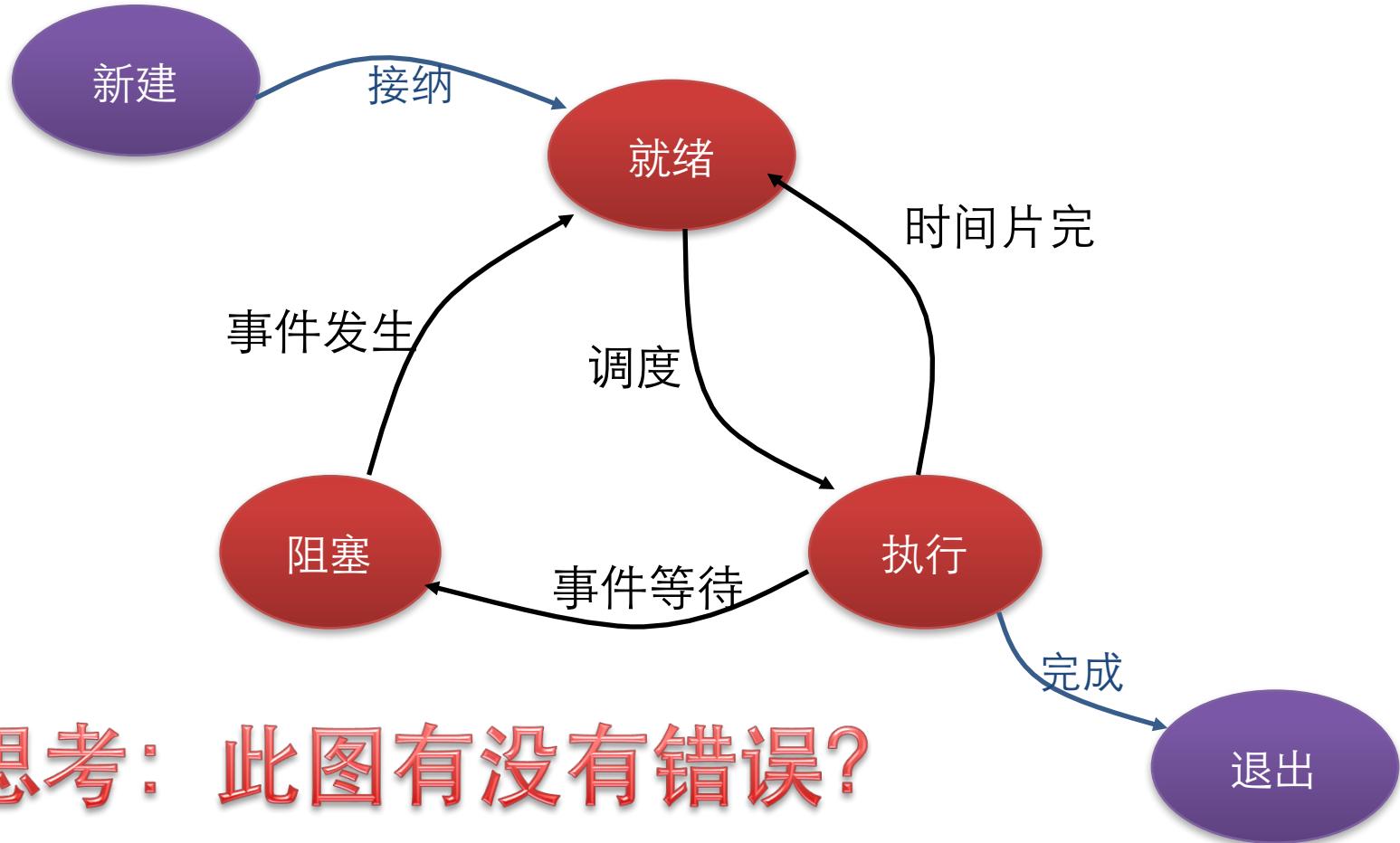
# 三状态



# 五状态

- 新建状态：
  - OS 已完成为创建一进程所必要的工作
    - 已构造了进程标识符；已创建了管理进程所需的表格
  - 但还没有允许执行该进程（尚未同意）
    - OS 所需的关于该进程的信息保存在主存的进程表中，但进程自身还未进入主存，也没有为与这个程序相关的数据分配空间，程序保留在辅存中。
- 退出状态
  - 它不再有执行资格
  - 表格和其它信息暂时由辅助程序保留

# 五状态



思考：此图有没有错误？

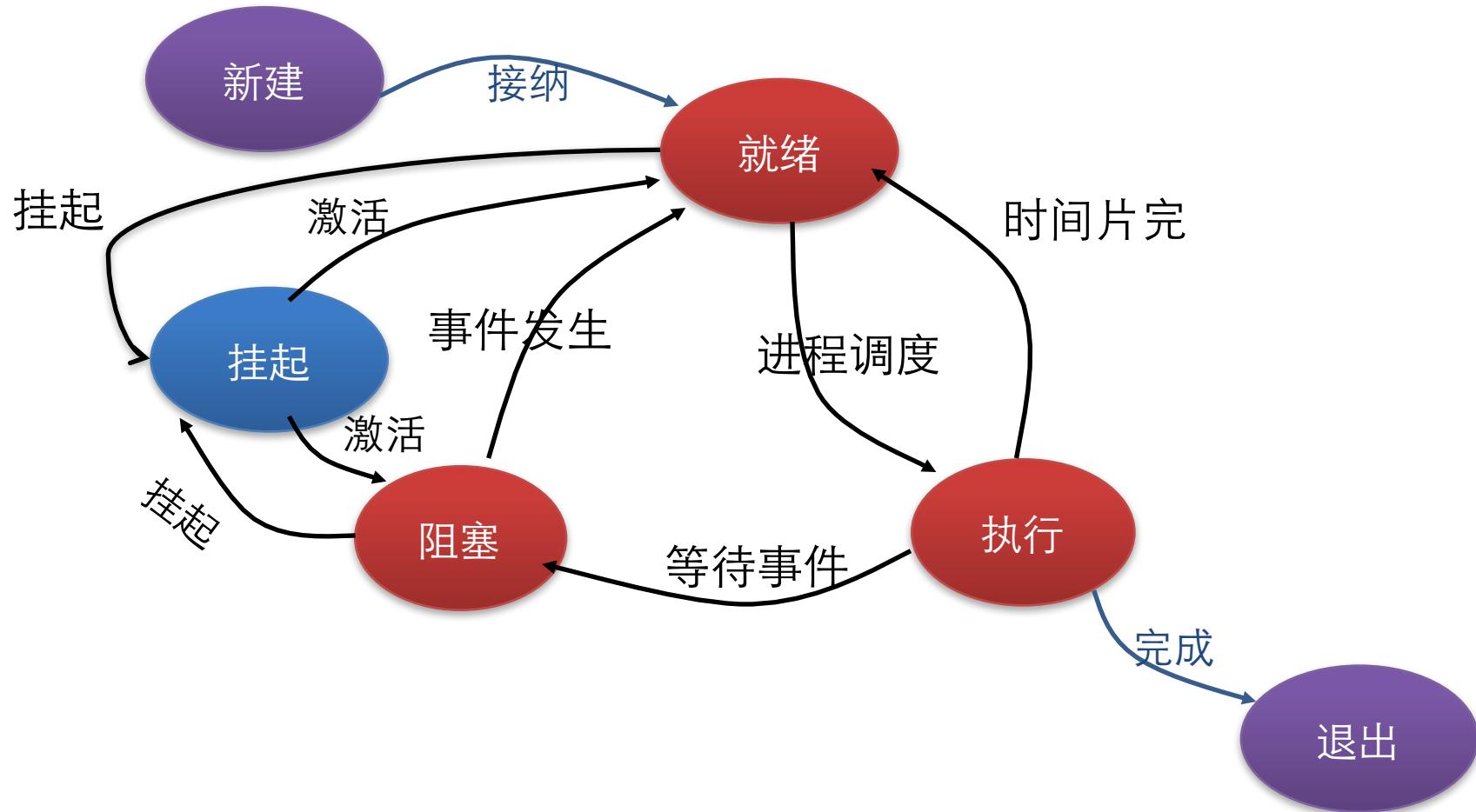
# 交换与挂起

- 问题
  - 主存中同时有多个进程
  - I/O速度比计算速度慢很多
  - 其他作业因没有主存空间不能投入运行
- 引入原因
  - 终端用户请求
  - 父进程请求
  - 负荷调节需要
  - 操作系统需要

# 交换

- 交换 (Swapping)
  - 挂起(Suspend): 把一个进程从内存转到外存
  - 激活(Activate/Resume): 把一个进程从外存转到内存
- 负作用: 交换是I/O操作, 费时间

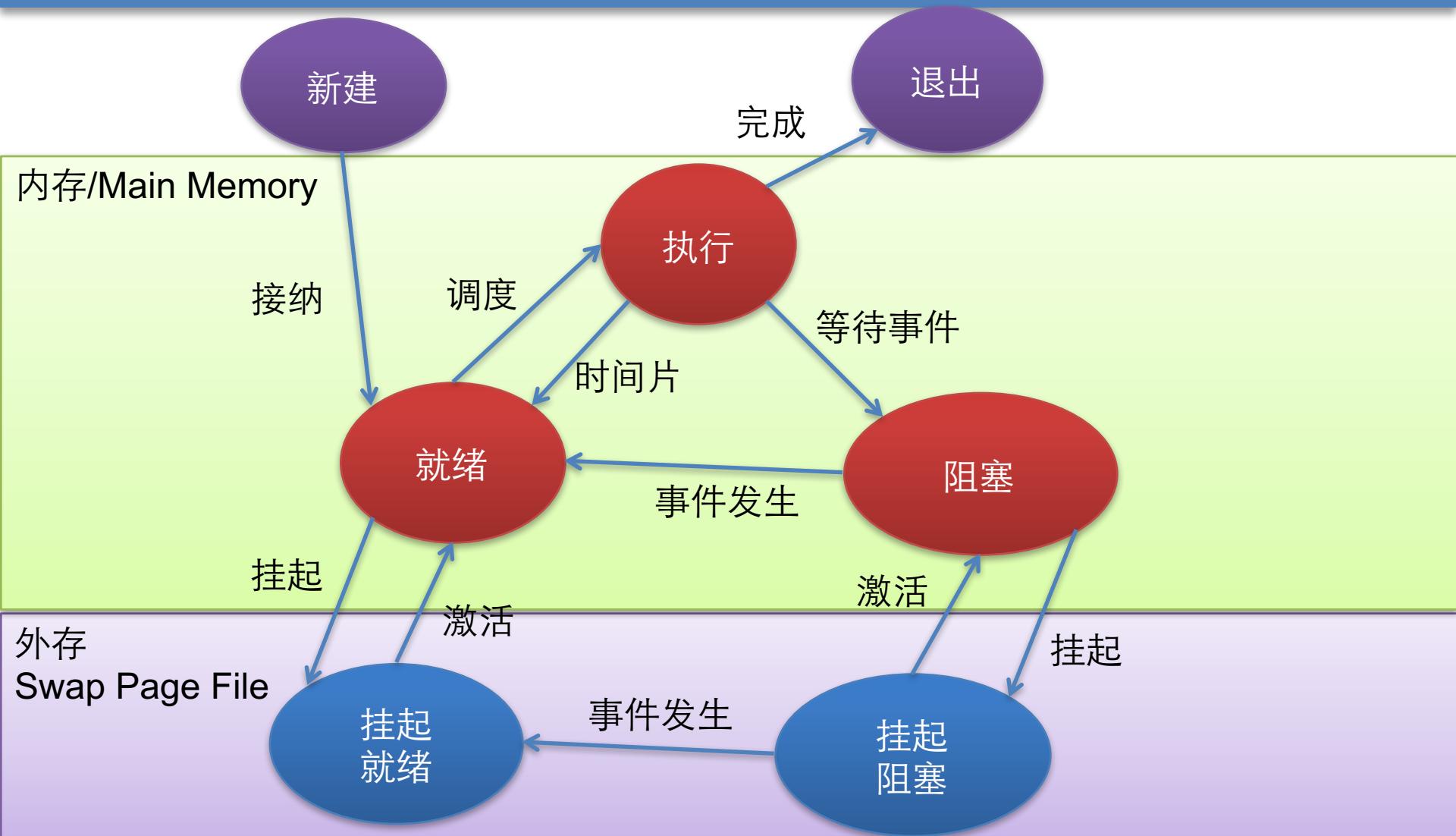
# 具有挂起状态的进程状态图



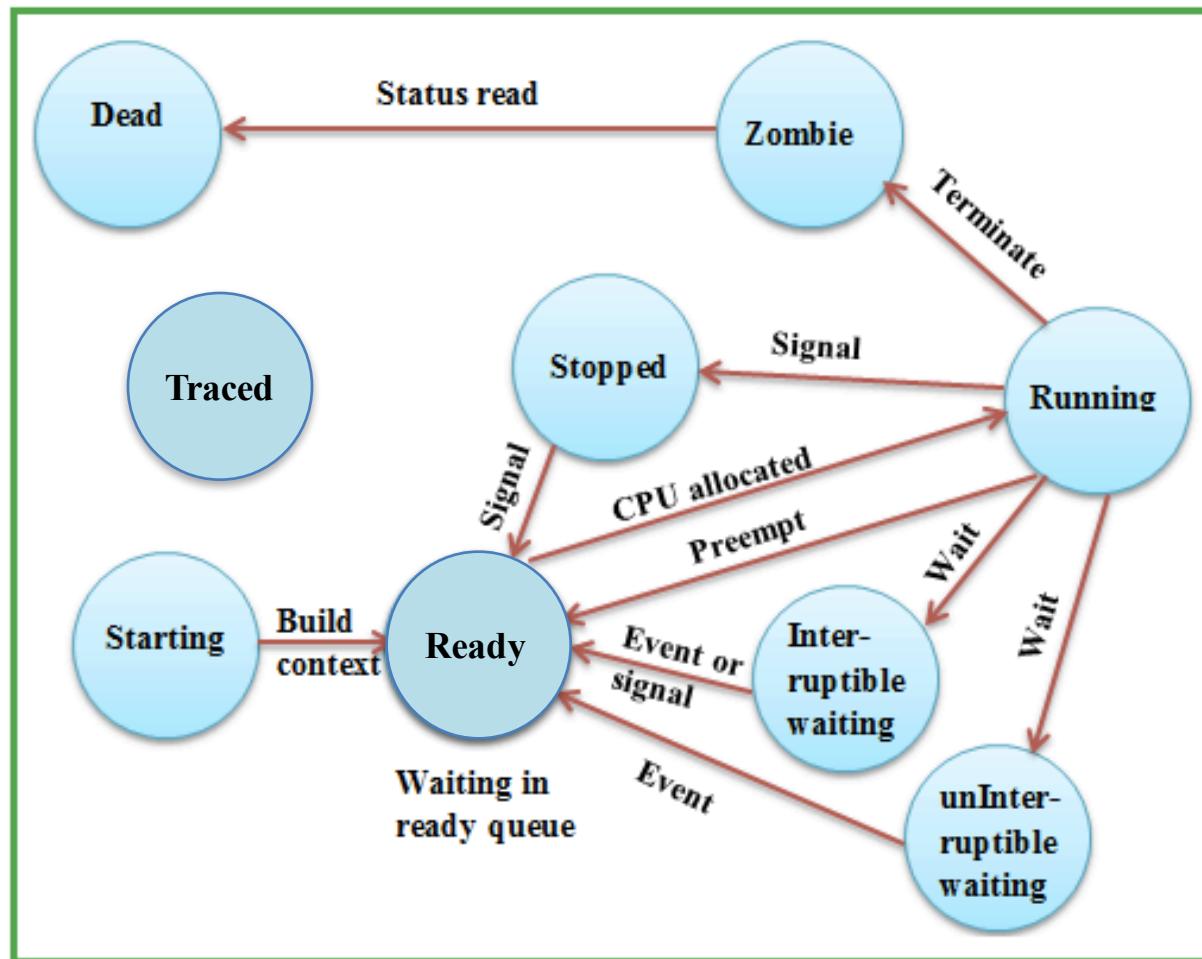
# 双挂起

- 就绪(Ready)：进程在内存且可立即进入运行状态（活动就绪）；
- 阻塞(Blocked)：进程在内存，并等待某事件的出现（活动阻塞）；
- 阻塞 / 挂起 (Blocked, suspend) : 进程在外存并等待某事件的出现；
- 就绪 / 挂起 (Ready, suspend) : 进程在外存，但只要进入内存，即可运行；

# 具有双挂起状态的进程状态图



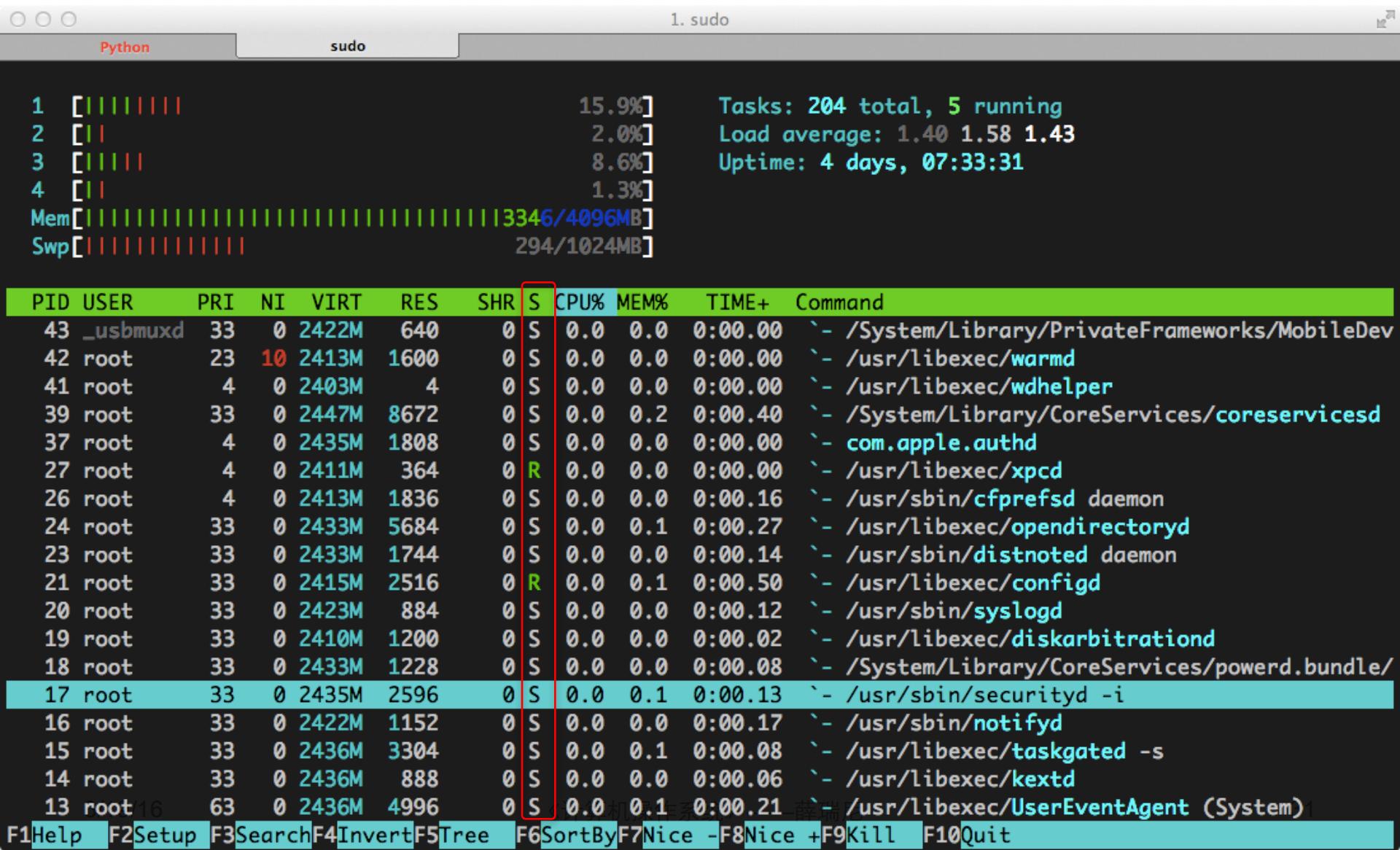
# Linux



# top: mac os X

1. top																		
PID	COMMAND	%CPU	TIME	#TH	#WQ	#PORT	#MREG	MEM	RPRVT	PURG	CMPRS	VPRVT	VSIZE	PGRP	PPID	STATE	UID	
19660	Grab	1.5	00:00.59	7	4	167	139+	7948K+	5364K+	48K+	0B	48M+	2590M+	19660	255	sleeping	502	
19659-	mdworker32	0.0	00:00.22	4	1	58	90	3848K	2272K	0B	0B	61M	639M	19659	255	sleeping	502	
19658	com.apple.hi	0.0	00:00.01	2	0	34	46	1044K	484K	0B	0B	45M	2411M	19658	1	sleeping	502	
19656	com.apple.au	0.0	00:00.03	3	2	49	62	1476K	788K	0B	0B	46M	2413M	19656	1	sleeping	502	
19655	com.apple.au	0.0	00:00.01	2	1	28	48	1176K	612K	0B	0B	37M	2404M	19655	1	sleeping	502	
19654	com.apple.hi	0.0	00:00.01	2	0	35	45	1024K	460K	0B	0B	37M	2403M	19654	1	sleeping	502	
19653	J8RPQ294UB.c	0.0	00:00.57	4	1	201	109	10M	2976K	84K	52K	46M	2479M	19653	255	sleeping	502	
19652	CVMCompiler	0.0	00:00.69	3	2	32	76	16M	16M	0B	0B	58M	2443M	19652	255	sleeping	502	
19650	AppSandboxSM	0.0	00:00.11	2	1	44	62	2136K	1316K	0B	0B	55M	2447M	19650	1	sleeping	502	
19648	Skitch	0.0	00:01.96	6	1	242	411	25M	15M	2476K	52K	93M	2704M	19648	255	sleeping	502	
19647	QuickLookSat	0.0	00:00.59	8	1	87	150	12M	11M	784K	60K	57M	3555M	19647	1	sleeping	502	
19645	QuickLookSat	0.0	00:00.15	2	0	49	62	2336K	1456K	0B	0B	45M	2435M	19645	1	sleeping	502	
19644	quicklookd	0.0	00:00.30	5	1	92	96	6128K	4852K	0B	0B	558M	2961M	19644	255	sleeping	502	
19643	top	9.2	00:06.34	1/1	0	35	40	2484K	2260K	0B	0B	44M	2403M	19643	4922	running	0	
19628-	Google Chrom	2.1	00:10.55	13	1	148	491	71M+	66M+	0B	96K	161M	953M	315	315	sleeping	502	
19627	mdworker	0.5	00:00.33	4	0	54	94+	5792K+	4488K+	0B	0B	51M+	2454M+	19627	255	sleeping	502	
19626	mdworker	0.0	00:00.20	4	0	54	82	6156K	4896K	0B	0B	51M	2430M	19626	255	sleeping	502	
19623	mdworker	0.0	00:00.19	4	0	54	84	6088K	4968K	0B	0B	50M	2422M	19623	255	sleeping	502	
16332	ocspd	0.0	00:08.29	6	0	87	67	3248K	1824K	0B	0B	31M	2436M	16332	1	sleeping	0	
16195	VBoxSVC	0.0	00:04.82	14	1	100	101	1088K	900K	0B	2008K	49M	2428M	16195	1	sleeping	502	
16193	VBoxXPCOMIPC	0.0	00:00.02	1	0	18	51	140K	104K	0B	2272K	38M	2401M	16193	1	sleeping	502	
16190	VirtualBox	0.0	00:01.11	6	1	161	317	10M	5936K	0B	3548K	43M	2659M	16190	255	sleeping	502	
16143-	Microsoft AU	0.0	00:00.31	3	1	97	93	1204K	708K	0B	2096K	49M	661M	16143	255	sleeping	502	
16139-	Microsoft Po	0.6	05:50.38	10	1	219	3337	214M	93M	52K	51M	221M	1757M	16139	255	sleeping	502	
16137	cookied	0.0	00:00.05	2	0	51	71	976K	508K	0B	932K	29M	2411M	16137	255	sleeping	502	
15895	netbiosd	0.0	00:00.03	2	1	42	41	508K	352K	0B	476K	45M	2412M	15895	1	sleeping	222	
15693	Image Captur	0.0	00:00.63	4	1	137	86	1752K	1456K	0B	2128K	56M	2466M	15693	255	sleeping	502	

# htop



# 思考

- 这些模型是不是完善的?
  - Error handling

# 操作系统对进程的控制

- 操作系统内核
  - 一些与硬件紧密相关的模块或运行频率较高的模块，公用基本操作模块等常驻内存，便于提高操作系统运行效能的软件，称为操作系统的内核。

# 内核功能

- 进程管理：创建、撤消、调度、控制
- 存储管理：分配或回收空间、虚拟存储管理等。
- I/O设备管理：设备、通道的分配和回收、设备的管理、虚拟设备的实现等。
- 中断处理：操作系统的重要活动都依赖于中断。

# 原语(Primitive)

- 由若干机器指令构成用以完成特定功能的一段程序，并在执行中不可分割的
  - 操作系统内核的功能大都通过执行各种原语实现
- 原子操作
  - 在一个操作中的所有动作，要么全做，要么全不做
  - All-or-None

# 原子操作

- 在单机系统
  - 单条指令
  - 采用屏蔽中断的方式来保证操作的原子性。
- 多核系统中：内存栅障（memory barrier）
  - 一个CPU核在执行原子操作时，其他CPU核必须停止对内存操作或者不对指定的内存进行操作，避免数据竞争（data race）问题。

# 操作系统控制结构

- 操作系统是管理计算机资源的
  - 用表格（或数据结构）来记载各资源的信息
  - 用代码来实现对资源的管理、维护、更新等。
- 常见的表（或数据结构）
  - 存储表、设备表、文件表、进程表等。

# 进程控制块

- PCB (process control block)
- 一个数据结构
  - 进程存在的唯一标志；
  - 常驻内存

pid
进程状态
现场
优先级
阻塞原因
程序地址
同步机制
资源清单
链接指针

# 进程控制块中的信息

- 进程标识符
  - 惟一地标识一个进程，通常有两种标识符：
    - 内部标识符：为每一个进程赋予一个惟一的数字标识符，方便系统使用。
    - 外部标识符：由创建者提供，通常是由字母、数字组成，往往是由用户（进程）在访问该进程时使用。

# 处理机状态

- 主要由处理机的各种寄存器中的内容组成
  - 通用寄存器，又称为用户可视寄存器。
  - 指令计数器PC (Program Counter) ，存放了要访问的下一条指令的地址。
  - 程序状态字PSW，其中含有状态信息，如条件码、执行方式、中断屏蔽标志等。
  - 用户栈指针，存放过程和系统调用参数及调用地址。栈指针指向该栈的栈顶。

# 进程调度信息

- 进程状态，指明进程的当前状态。
- 进程优先级。
- 进程调度所需的其它信息。比如，进程已等待CPU的时间总和、进程已执行的时间总和等。
- 事件，由执行状态转变为阻塞状态所等待发生的事件，即阻塞原因。

# 进程控制信息

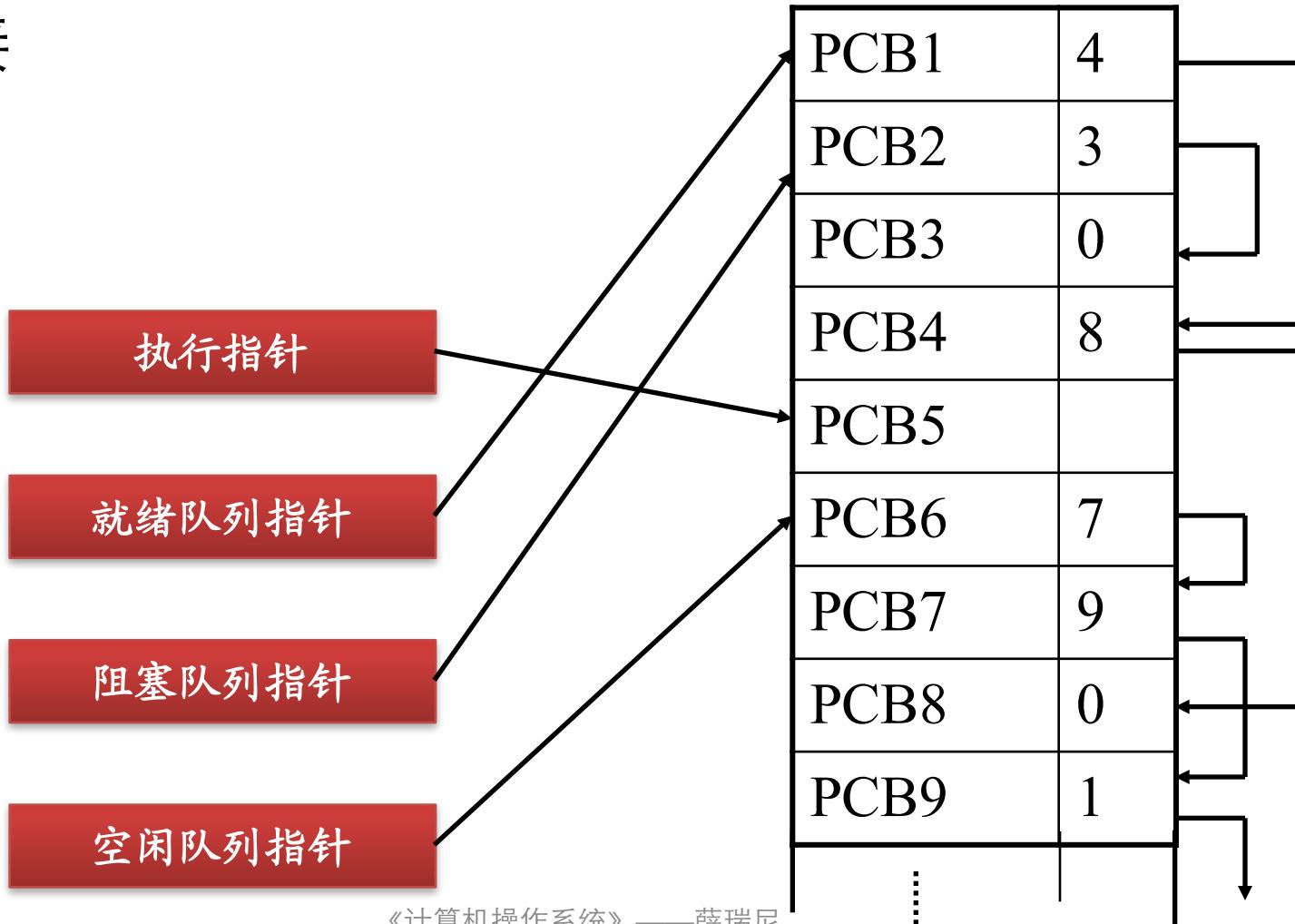
- 程序和数据的地址，是指进程的程序和数据所在的内存或外存地址。
- 进程同步和通信机制，指实现进程同步和进程通信时必需的机制，如消息队列指针、信号量等。
- 资源清单，进程所需的全部资源及已经分配到该进程的资源的清单；
- 链接指针。

# 进程控制块的组织方式

- 链接方式
  - 把具有同一状态的PCB，用其中的链接字链接成一个队列。
- 索引方式
  - 系统根据所有进程的状态建立几张索引表。

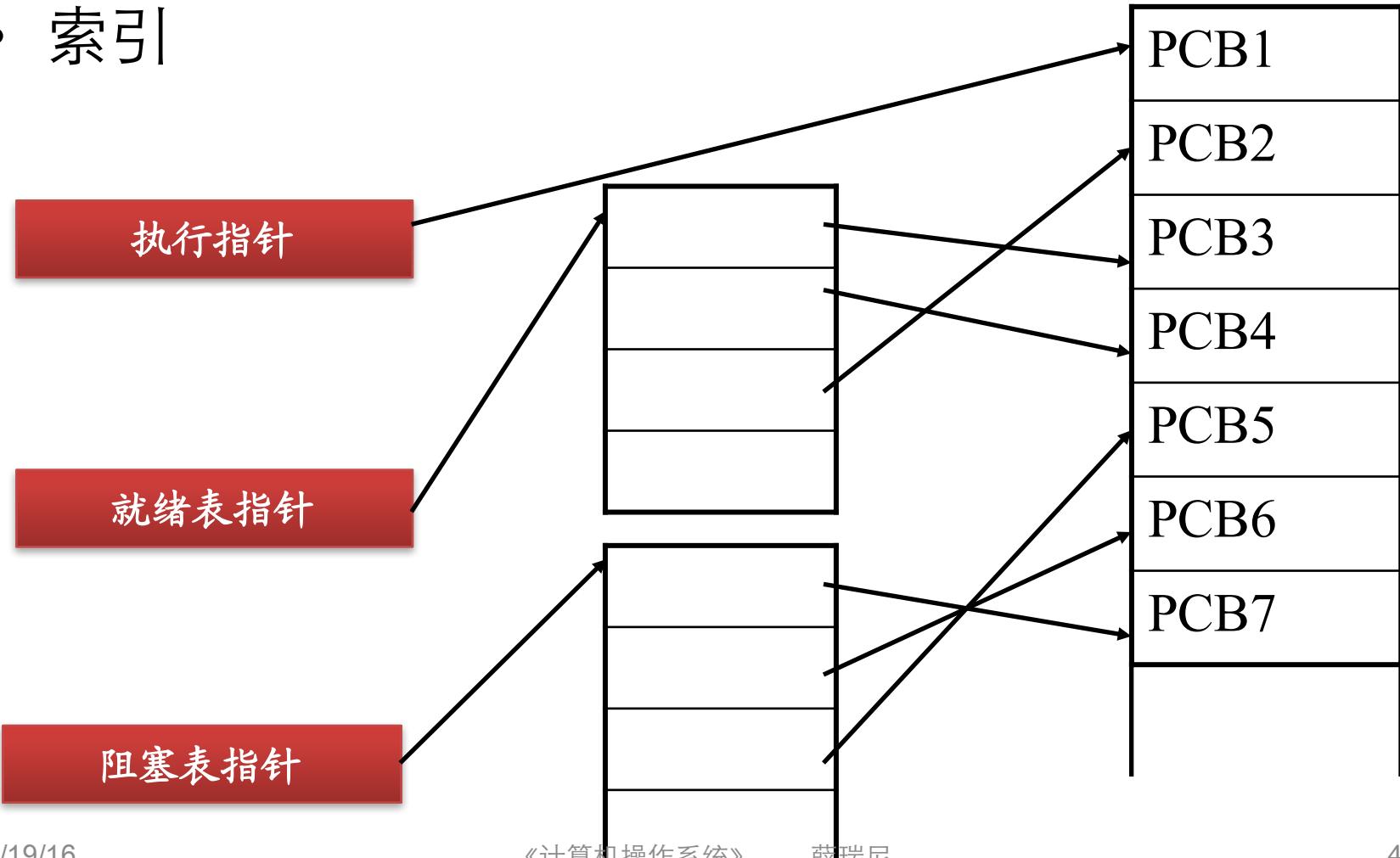
# 进程控制块连接方式

- 链接



# 进程控制块

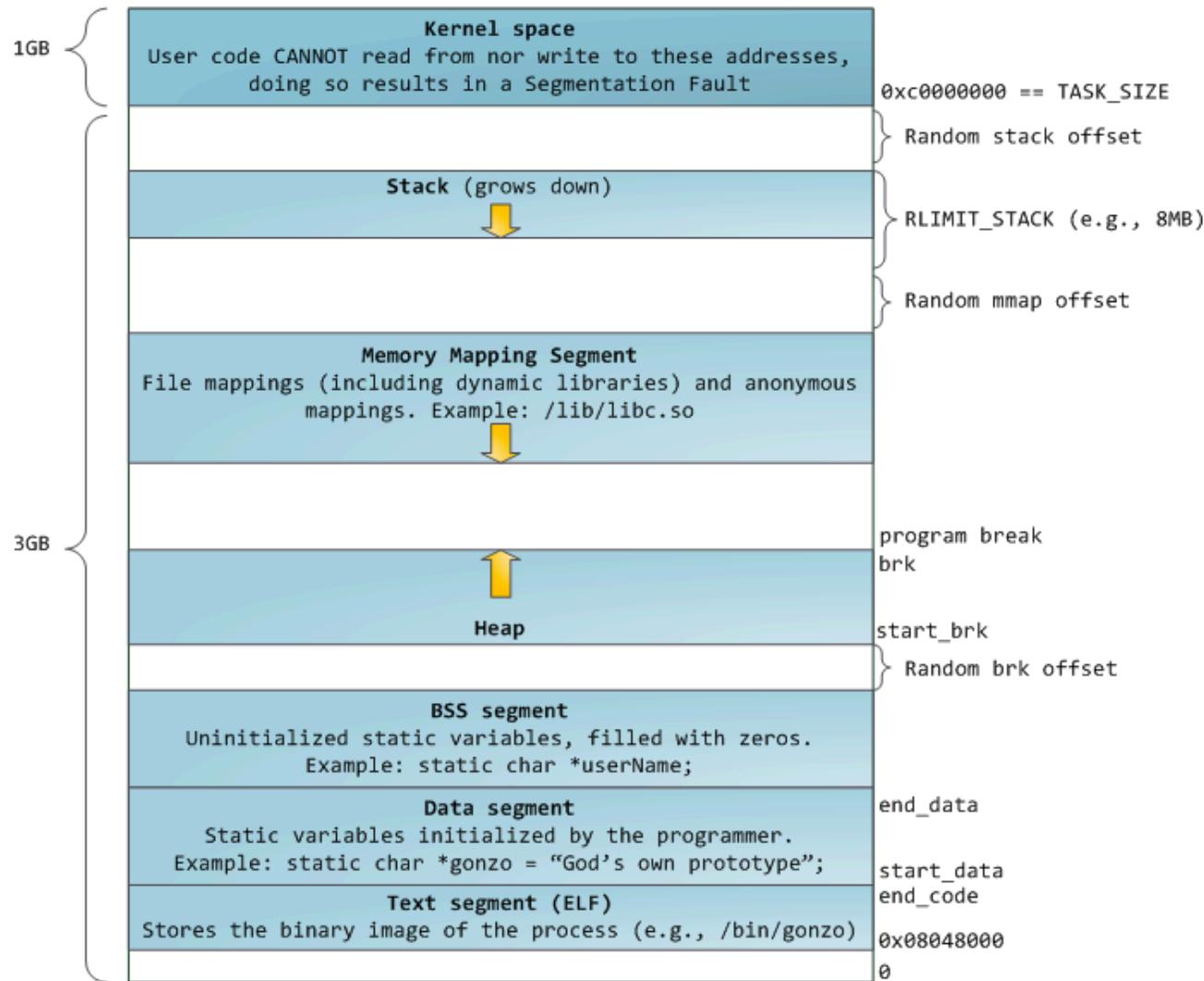
- 索引



# Linux task\_struct

```
struct task_struct {  
    /* these are hardcoded - don't touch */  
    /* -1 unrunnable, 0 runnable, >0 stopped */  
    volatile long          state;  
    long                  priority;  
    /* per process flags */  
    unsigned              long flags;  
    int      errno;  
    struct task_struct    *next_task, *prev_task;  
    struct task_struct    *next_run,   *prev_run;  
    int      pid;  
    /* memory management info */  
    struct mm_struct      *mm;  
    /* signal handlers */  
    struct signal_struct  *sig;  
};
```

# 内存中的样子



# 思考

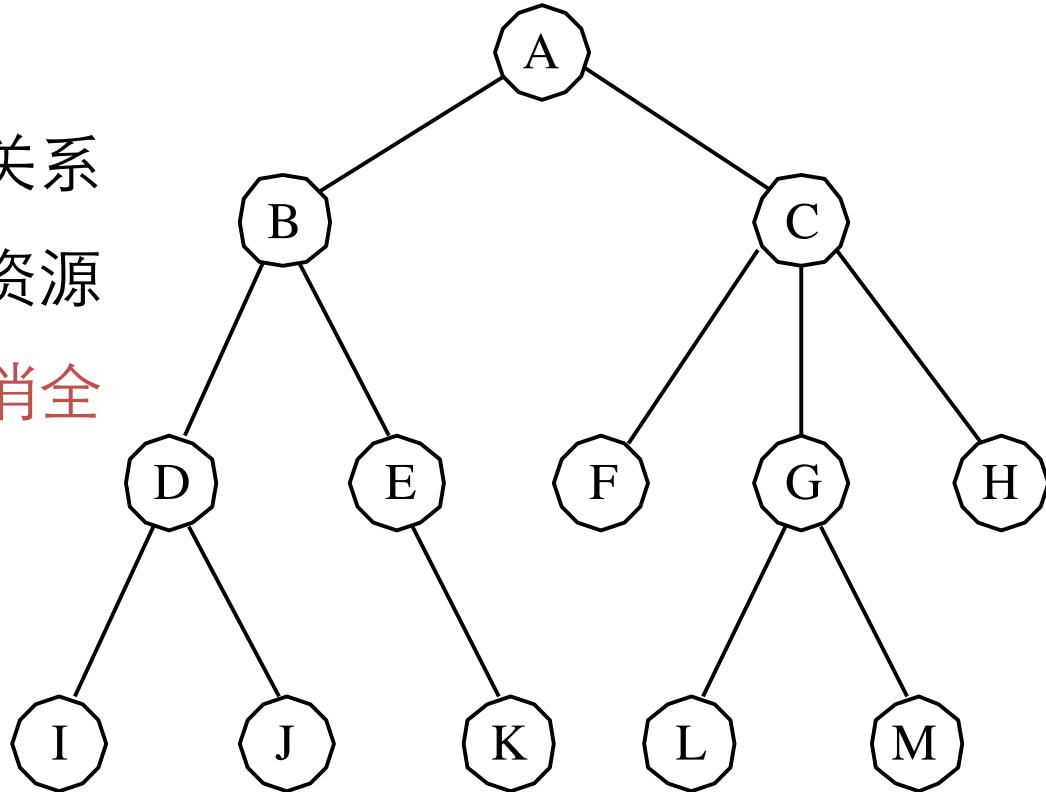
- PCB和进程的代码数据放在一起吗?
  - 内核态和用户态
  - 系统空间和用户空间
- 进程和程序的数量对应关系?
  - 1-1; 1-n; n-1; n-n?

# 推荐练习

- Linux内核有list.h，理解其设计原理，并用此list重新完成数据结构的部分链表操作。
- 认真体会其设计思想

# 进程控制

- 进程图（树）
  - 描述了进程的家族关系
  - 子进程可继承父的资源
  - 父进程的撤消会撤消全部子进程。
  - init, launchd 进程
    - pid = 1



# 例子： mac OS X

```

1 [|||||] 22.4% Tasks: 199 total, 5 running
2 [|||] 4.7% Load average: 1.38 1.46 1.50
3 [|||||] 13.9% Uptime: 5 days, 19:10:49
4 [|||] 5.9%
Mem[||||||||||||||||| 3184/4096MB]
Swp[||||||||||||||||| 968/2048MB]

```

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
1	root	31	0	2415M	4164	0	S	0.0	0.1	0:01.06	/sbin/launchd
22383	xrn	4	0	2414M	6564	0	S	0.0	0.2	0:00.00	- com.apple.quicklook.satellite
22374	xrn	63	0	3486M	12568	0	S	0.0	0.3	0:00.00	- com.apple.WebKit.WebContent
22297	xrn	4	0	2411M	4408	0	S	0.0	0.1	0:00.00	- com.apple.hiservices-xpcservice
22294	xrn	4	0	2411M	4452	0	S	0.0	0.1	0:00.00	- com.apple.Preview.TrustedBookmarksService
22114	_netbios	4	20	2411M	2288	0	S	0.0	0.1	0:00.00	- /usr/sbin/netbiosd
20700	xrn	4	0	2467M	744	0	S	0.0	0.0	0:00.00	- com.apple.coremedia.videodecoder
20699	xrn	4	0	2412M	664	0	S	0.0	0.0	0:00.00	- com.apple.audio.ComponentHelper
20698	xrn	4	0	2411M	136	0	S	0.0	0.0	0:00.00	- com.apple.audio.SandboxHelper
20673	xrn	4	0	2411M	1308	0	S	0.0	0.0	0:00.00	- com.apple.hiservices-xpcservice
20644	xrn	4	0	2446M	1000	0	S	0.0	0.0	0:00.00	- com.apple.AppSandboxSMLoginItemEnabler
20637	xrn	48	0	2593M	6744	0	S	0.0	0.2	0:00.05	- com.apple.appkit.xpc.openAndSavePanelService
20631	xrn	4	0	2411M	600	0	S	0.0	0.0	0:00.00	- com.apple.hiservices-xpcservice
19728	xrn	63	0	3557M	3072	0	S	0.0	0.1	0:00.00	- com.apple.WebKit.WebContent
16332	root	63	0	2436M	4156	0	S	0.0	0.1	0:00.45	- /usr/sbin/ocspd
13924	root	33	0	2394M	4	0	S	0.0	0.0	0:00.00	- /usr/sbin/aslmanager -s /var/log/performanc
13865	xrn	4	0	2411M	484	0	S	0.0	0.0	0:00.00	- com.apple.hiservices-xpcservice
10733	xrn	4	0	2452M	636	0	S	0.0	0.0	0:00.00	- com.apple.appkit.xpc.sandboxedServiceRunner
3483	root	4	0	2427M	560	0	S	0.0	0.0	0:00.01	- /usr/sbin/systemstats --xpc
3358	root	33	0	2444M	920	0	S	0.0	0.0	0:00.76	- /Library/PrivilegedHelperTools/com.ubuntu.o
2272	root	4	0	2403M	4	0	S	0.0	0.0	0:00.00	- com.apple.cmio.registerassistantservice
1997	xrn	4	0	2411M	48	0	S	0.0	0.0	0:00.00	- com.apple.appstore.PluginXPCService
1775	root	4	0	2435M	3312	0	S	0.0	0.1	0:00.00	- /usr/libexec/syspolicyd

F1Help F2Setup F3Search F4Invert F5Tree F6SortBy F7Nice -F8Nice +F9Kill F10Quit

# 例子： mac os X

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
15693	xrn	48	0	2466M	2604	0	S	0.0	0.1	0:00.00	- /System/Library/Image Capture/Support/Image Capture Extens
15678	xrn	63	0	1039M	52948	0	S	0.0	1.3	0:00.84	- /Applications/Dropbox.app/Contents/MacOS/Dropbox
15688	xrn	31	0	582M	204	0	S	0.0	0.0	0:00.00	- /Library/DropboxHelperTools/Dropbox_u502/dbfsevents
15689	xrn	31	0	590M	3076	0	S	0.0	0.1	0:00.02	- /Library/DropboxHelperTools/Dropbox_u502/dbfsevent
15690	xrn	31	0	582M	76	0	S	0.0	0.0	0:00.02	- /Library/DropboxHelperTools/Dropbox_u502/dbfse
4845	xrn	45	0	720M	1524	0	S	0.0	0.0	0:00.61	/Library/Application Support/Google/GoogleTalkPlugin.app/C
1868	xrn	33	0	2434M	544	0	S	0.0	0.0	0:00.00	- /System/Library/PrivateFrameworks/HelpData.framework/Versi
808	xrn	48	0	677M	1680	0	S	0.0	0.0	0:00.11	/Applications/QQ.app/Contents/Library/LoginItems/QQPlatfor
807	xrn	48	0	691M	2408	0	S	0.0	0.1	0:00.12	- /Applications/QQ.app/Contents/Library/LoginItems/ScreenCap
798	xrn	4	0	1415M	54508	0	S	0.0	1.3	0:28.41	- /Applications/QQ.app/Contents/MacOS/QQ
711	xrn	63	0	2642M	49368	0	S	0.0	1.2	0:01.42	- /Applications/iTerm.app/Contents/MacOS/iTerm
4921	xrn	33	0	2411M	4	0	S	0.0	0.0	0:00.00	- login -fp xrn
4922	xrn	31	0	2403M	1276	0	S	0.0	0.0	0:00.00	- bash
20131	root	33	0	2411M	2360	0	S	0.0	0.1	0:00.00	- sudo htop
20132	root	31	0	2403M	3132	0	C	0.0	0.1	0:00.00	- htop
719	xrn	33	0	2411M	4	0	S	0.0	0.0	0:00.00	- login -fp xrn
720	xrn	31	0	2395M	4	0	S	0.0	0.0	0:00.00	- bash
902	xrn	33	0	2472M	10124	0	S	0.0	0.2	0:02.14	- python Dropbox/goagent/local/proxy.py

# 进程的创建

用户登录

- 为终端用户建立一进程

作业调度

- 为被调度的作业建立进程

提供服务

- 如要打印时建立打印进程

应用请求

- 由应用程序建立多个进程

# 进程的创建

- 进程的创建
  - 申请空白PCB
  - 为新进程分配资源（内存、文件等）
  - 初始化PCB
  - 将新进程插入就绪队列。
- 创建新进程后
  - 父进程与子进程并发执行
  - 父进程等待，直到某个或者全部子进程执行完毕。

# UNIX的进程控制

- `fork()`: 创建一个新进程
- `exec()`: 执行一个可执行程序
- `exit()`: 终止
- `sleep()`: 暂停一段时间
- `pause()`: 暂停并等待信号
- `wait()`: 等待子进程终止
- `kill()`: 发送信号到某个或一组进程
- `ptrace()`: 设置执行断点(breakpoint), 允许父进程控制子进程的运行

# fork()—创建新进程

- 调用格式： `pid = fork()`
- 在调用`fork`之后， 父进程和子进程均在下一条语句上继续运行。
- 父、子进程的`fork`返回值不同
  - 在子进程中返回时， `pid`为0；
  - 在父进程中返回时， `pid`为所创建的子进程的标识。

# fork(): 两个关键点

- 运行顺序
  - 父、子进程的运行是无关的，运行顺序也不固定。若要求父子进程运行顺序一定，则要用到进程间的通信。
- 数据共享
  - 除了子进程标识符和其PCB结构中的某些特性参数不同之外，子进程是父进程的精确复制。

## 父进程

```
main()
{
    int pid;
    printf("PID");
    pid = fork();
    if (pid != 0)
        printf("parent");
    else
        printf("child");
}
```

执行

父进程

```
main( )
{
    int val;
    printf("PID");
    pid = fork();
    if (pid != 0)
        printf("parent");
    else
        printf("child");
}
```

分裂

继续执行

子进程

```
main( )
{
    int val;
    printf("PID");
    pid = fork();
    if (pid != 0)
        printf("parent");
    else
        printf("child");
}
```

继续执行

# fork()调用例子 (1)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void main(void) {
    printf("Hello \n");
    fork();
    printf("Bye \n") ;
}
```

运行结果  
Hello  
Bye  
Bye

# fork()调用例子 (2)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void main(void){
    if (fork()==0)
        printf("In the CHILD process\n");
    else
        printf("In the PARENT process\n");
}
```

程序的运行无法保证输出顺序，输出顺序依赖于内核所用的调度算法

# 写时拷贝

- COW: copy-on-write
- 共享: 精确拷贝
- 修改: 创建新的memory map
  - OS自动完成, 用户透明
- 例外
  - 文件句柄

# fork()调用例子 (3)

```
void main(void) {  
    int i;  
    static char buffer[10];  
    if (fork()==0)  
        strcpy(buffer, "Child\n");  
    else  
        strcpy(buffer, "Parent\n");  
  
    for (i=0; i < 5; i++) {  
        sleep(1);  
        write(1, buffer, sizeof(buffer));  
    }  
}
```

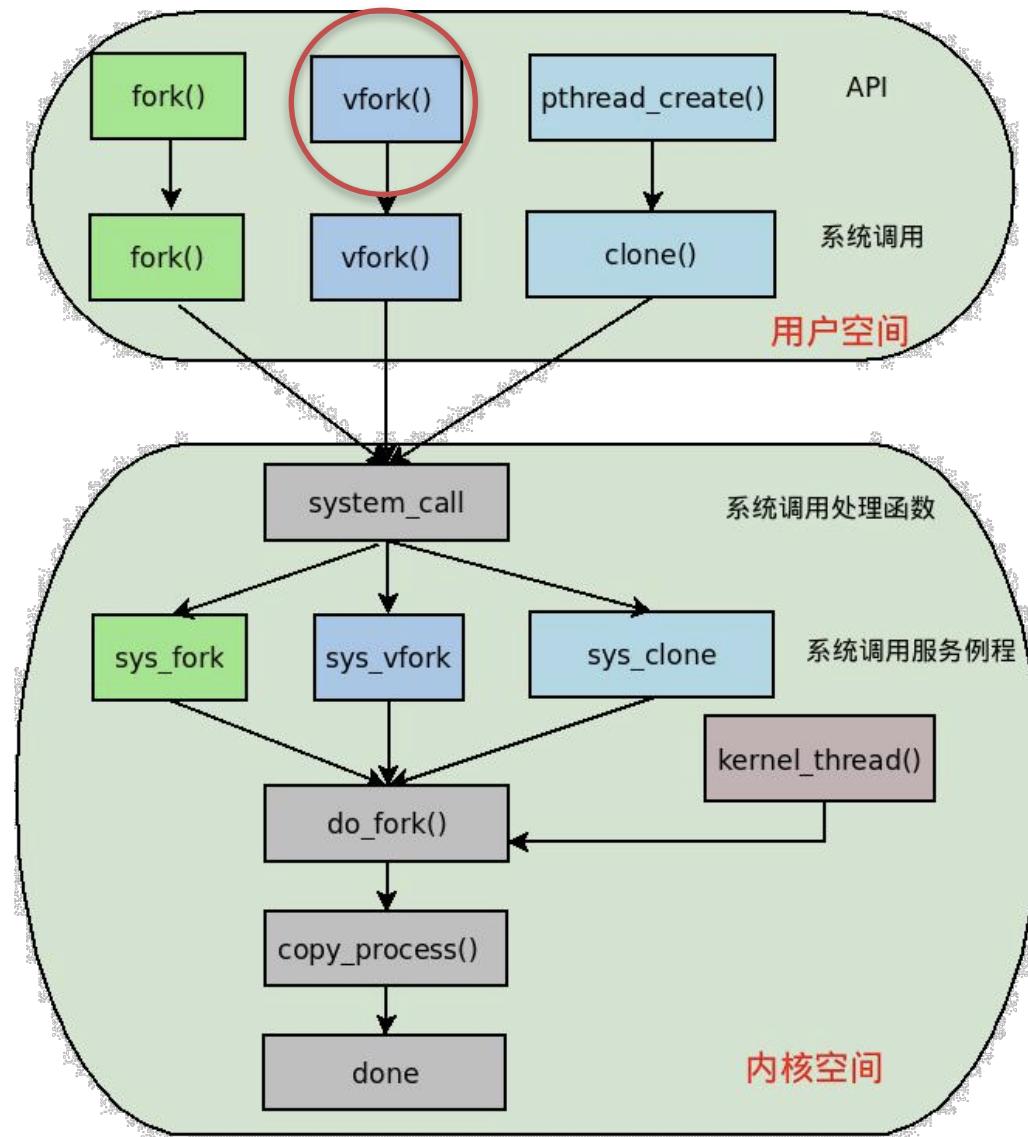
# fork()调用例子 (2)

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int global = 4;
void main(void) {
    int pid;
    int local = 5;
    printf("before fork\n");
```

```
if ((pid = fork()) < 0) {
    printf("fork error\n");
    exit(0);
} else if (pid == 0) {
    global++;
    local--;
}
printf("global=%d,
       Local=%d\n",
       global, local);
```

父进程: global=? , local=?  
子进程: global=? , local=?

# Linux fork家族



Question

子进程只能这么用么？

# exec()—执行一个文件的调用

- 子进程如何执行一个新的程序?
  - 通过exec() 调用族， 加载新的程序文件
- 子进程可以拥有自己的可执行代码， 即用一个新进程覆盖调用进程。
- 参数包括新进程对应的文件和命令行参数
  - 成功调用时， 不再返回；
  - 否则， 返回出错原因。

# exec系列调用

- 各种调用的区别在于参数的处理方法不同，常用的格式有：
  - execvp(filename, argp):
  - execlp(filename, arg0, arg1, ..., (char \*)0):
- 在大多数程序中，系统调用fork和exec是结合在一起使用的。父进程生成一个子进程，然后通过调用exec覆盖该子进程

# execv

```
int global = 0;
void main() {
    if ((child = fork()) < 0) {
        /* 创建失败 */
    } else if (child == 0) {
        if (execv(B...)) < 0) {
            /* 加载失败 */
            global += 1;
        } else {
            global += 2;
        }
    } else {
        global += 3;
    }
    printf("global=%d", global)
}
```

- 输出分别是什么
  - fork失败
    - 父进程: 0
    - 子进程: ?
  - execv失败
    - 父进程: 3
    - 子进程: 1
  - execv成功
    - 父进程: 3
    - 子进程: ?

# 进程的终止

- 引起进程终止的事件
  - 正常结束：如exit, halt, logoff
  - 异常结束：
  - 外界干预：
    - kill进程
    - 父进程终止
    - 父进程请求
- 无可用存储器
- 越界
- 保护错误
- 算术错误
- I/O失败
- 无效指令
- 特权指令

# 进程的终止(2)

- 进程的终止过程
  - 检索PCB，检查进程状态；
  - 执行态→中止；
  - 有无子孙需终止；
  - 归还资源给其父进程或系统；
  - 从PCB队列中移出PCB。

# UNIX进程终止

- 系统调用： `exit(int ret)`
- 返回`ret`到其父进程， 释放所有资源
  - 内存、 打开文件和I/O缓存。
- 父进程通过`wait()`等待子进程的结束
  - 如何识别子进程? **wait(pid)**
  - 返回结果是什么? **ret**

# 进程的阻塞与唤醒

- 引起进程阻塞和唤醒的事件
  - 请求系统服务而得不到满足时，如向系统请求打印。
  - 启动某种操作而需同步时
    - 如进程A写，进程B读，则A未写，完B不能读。

# 进程的阻塞与唤醒(2)

- 进程阻塞过程
  - 是进程自身的一种**主动**行为
    - 可以，但更多是系统来做
  - 调block/pause原语
  - 停止执行，修改PCB入阻塞队列，并转调度。

# 进程的阻塞与唤醒(2)

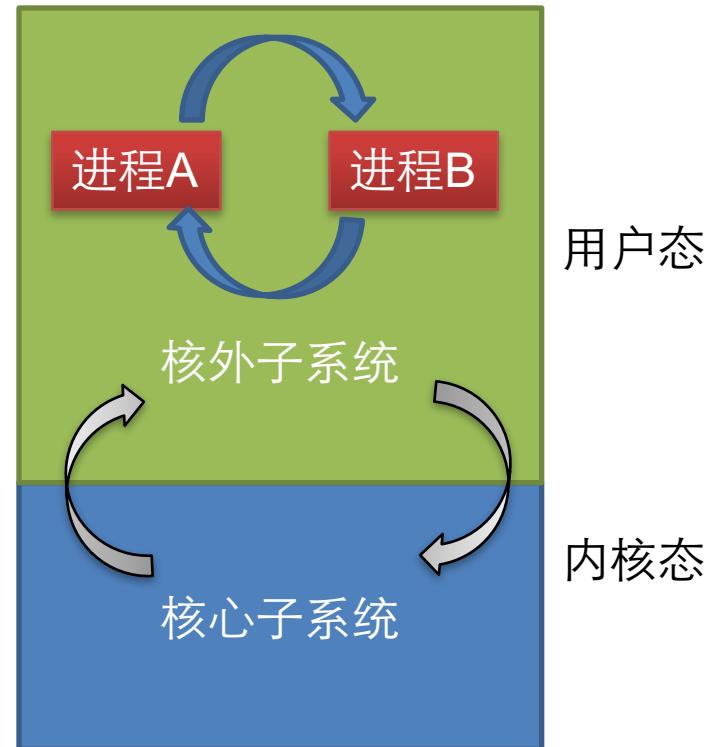
- 唤醒过程
  - 由其它相关进程完成
  - wakeup原语
  - 修改PCB，入就绪队列

# 进程的挂起与激活

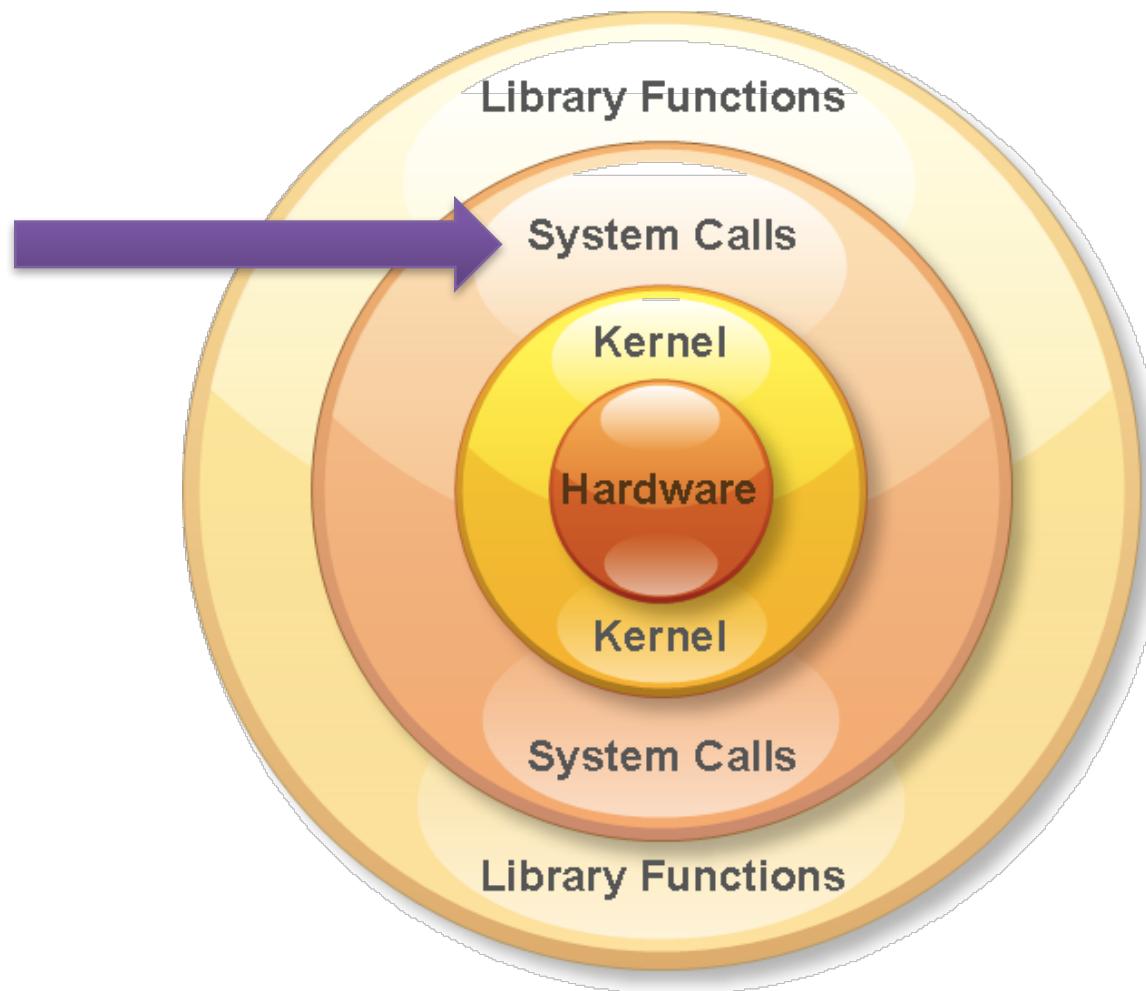
- 进程的挂起过程
  - 调用suspend原语，将该进程PCB移到指定队列
- 进程的激活过程
  - 调用active原语
- 谁来调用
  - 用户，OS?

# 进程切换

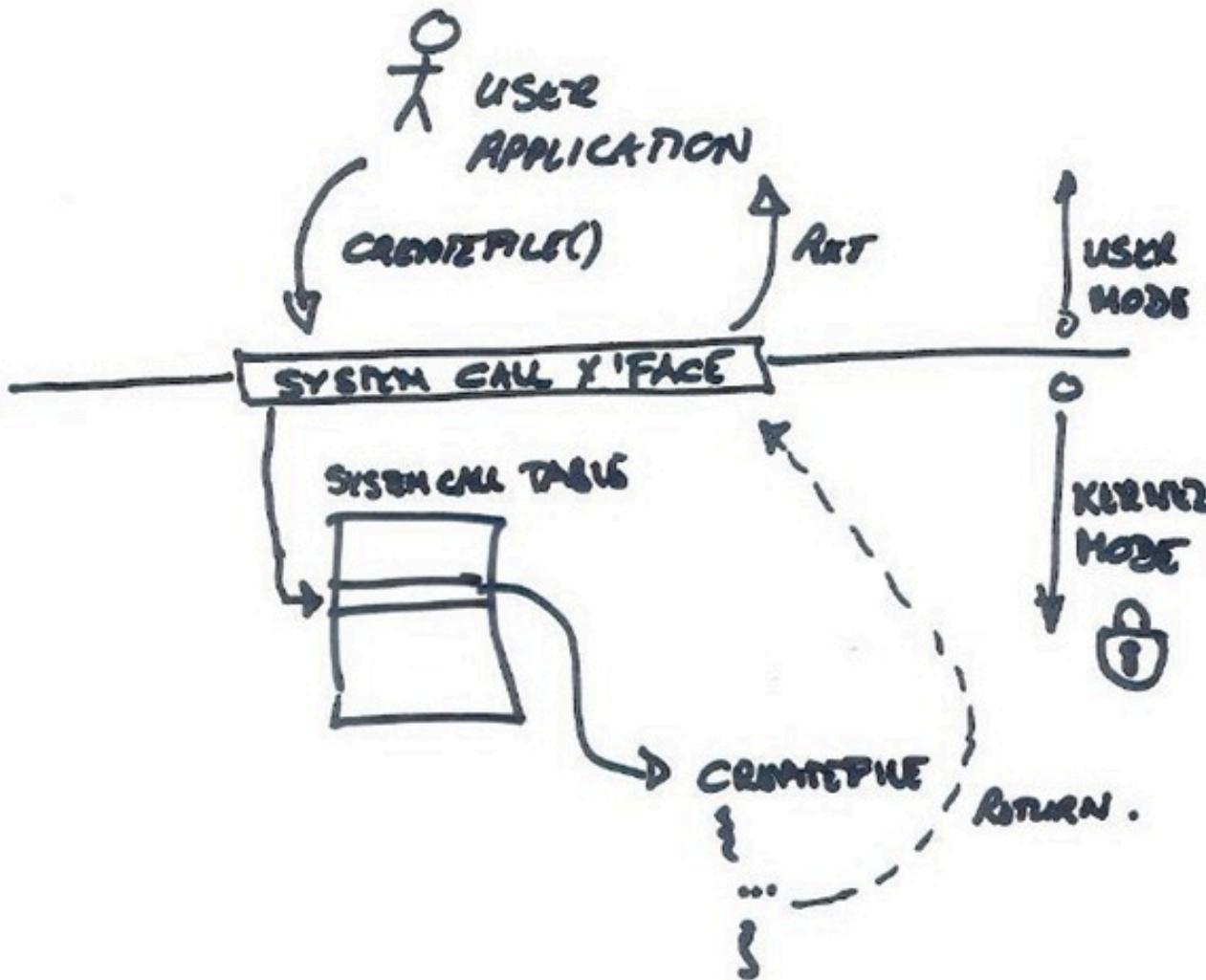
- 阻塞/唤醒，时间片
- 保存现场，恢复现场
- 什么是现场?
  - 进程上下文 (context)
- 与状态切换的区别
  - 请求系统调用



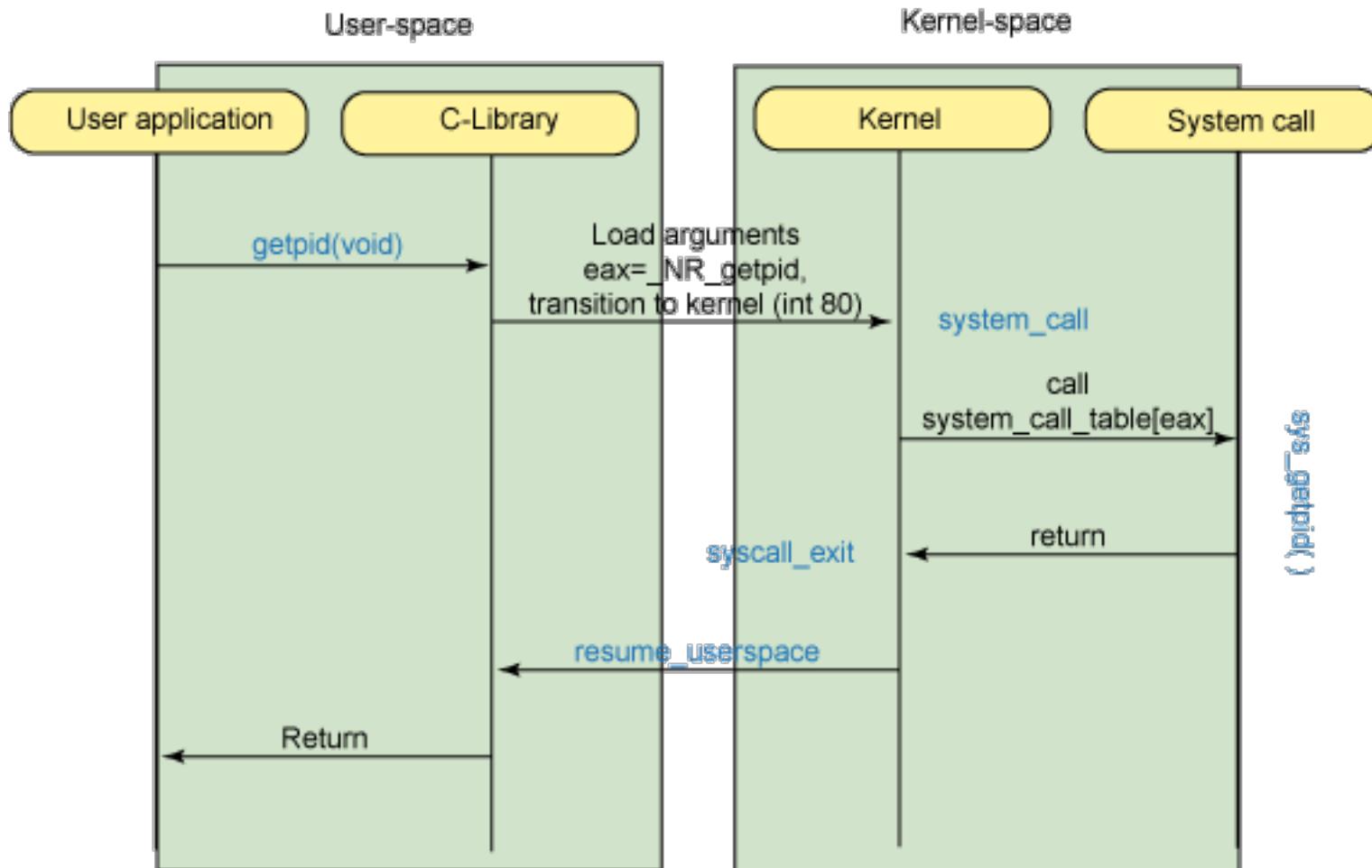
# 系统调用



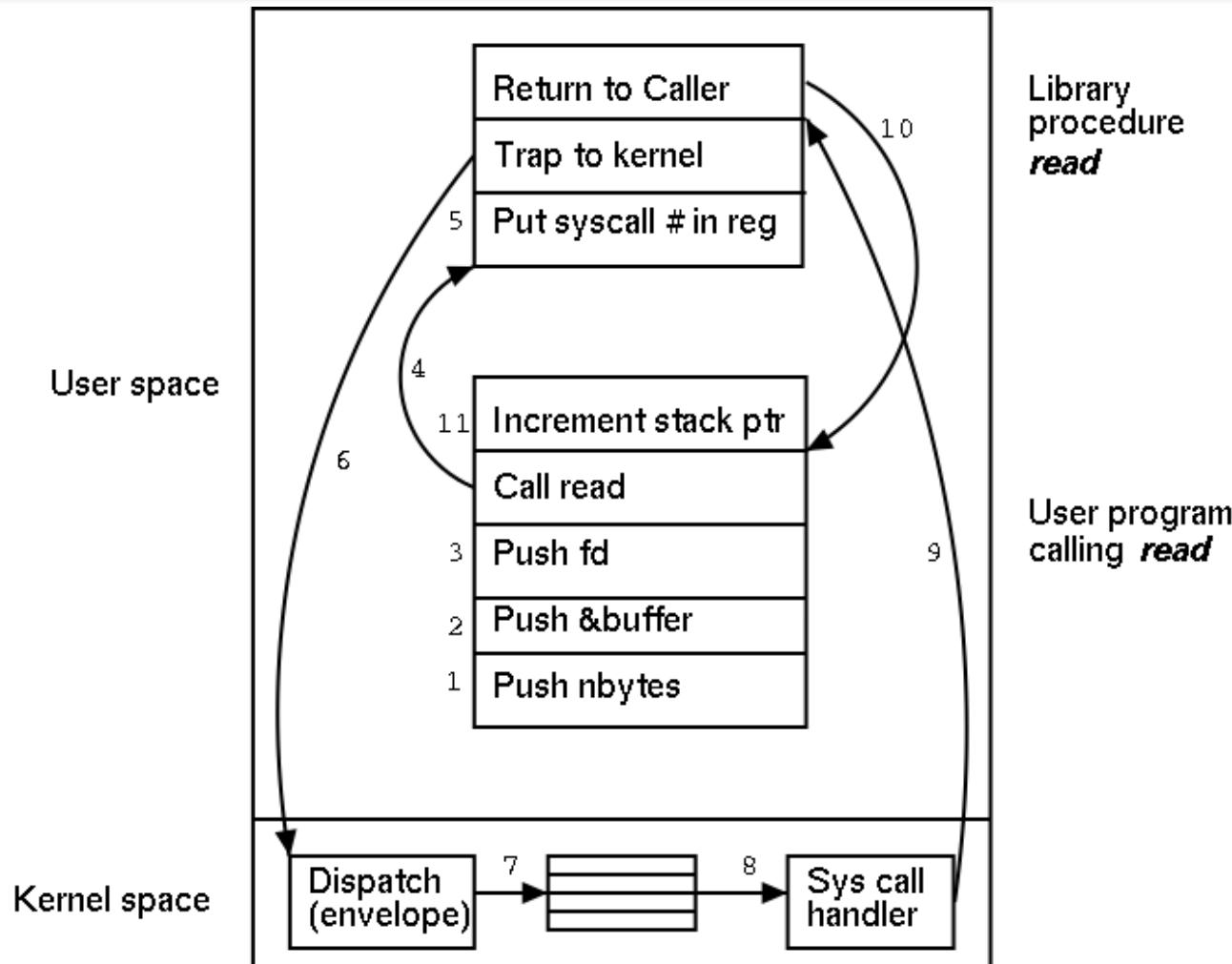
# 系统调用过程



# getpid@Linux



# 另一个角度 read(fd, &buffer, nbytes)



# 思考

- 系统调用和普通调用的区别?
  - 系统调用会引起从用户态进入核心态
  - `sum()`

# 线程

# 回顾进程

- 进程的两个特点
  - 资源所有权：一个进程包括一个保存进程映像的虚地址空间，并且不时地被分配给对资源的控制或所有权。
  - 调度／执行：一个具有执行状态和调度优先级的进程是一个被操作系统调度并分派的实体。
- 有没有改进之处？

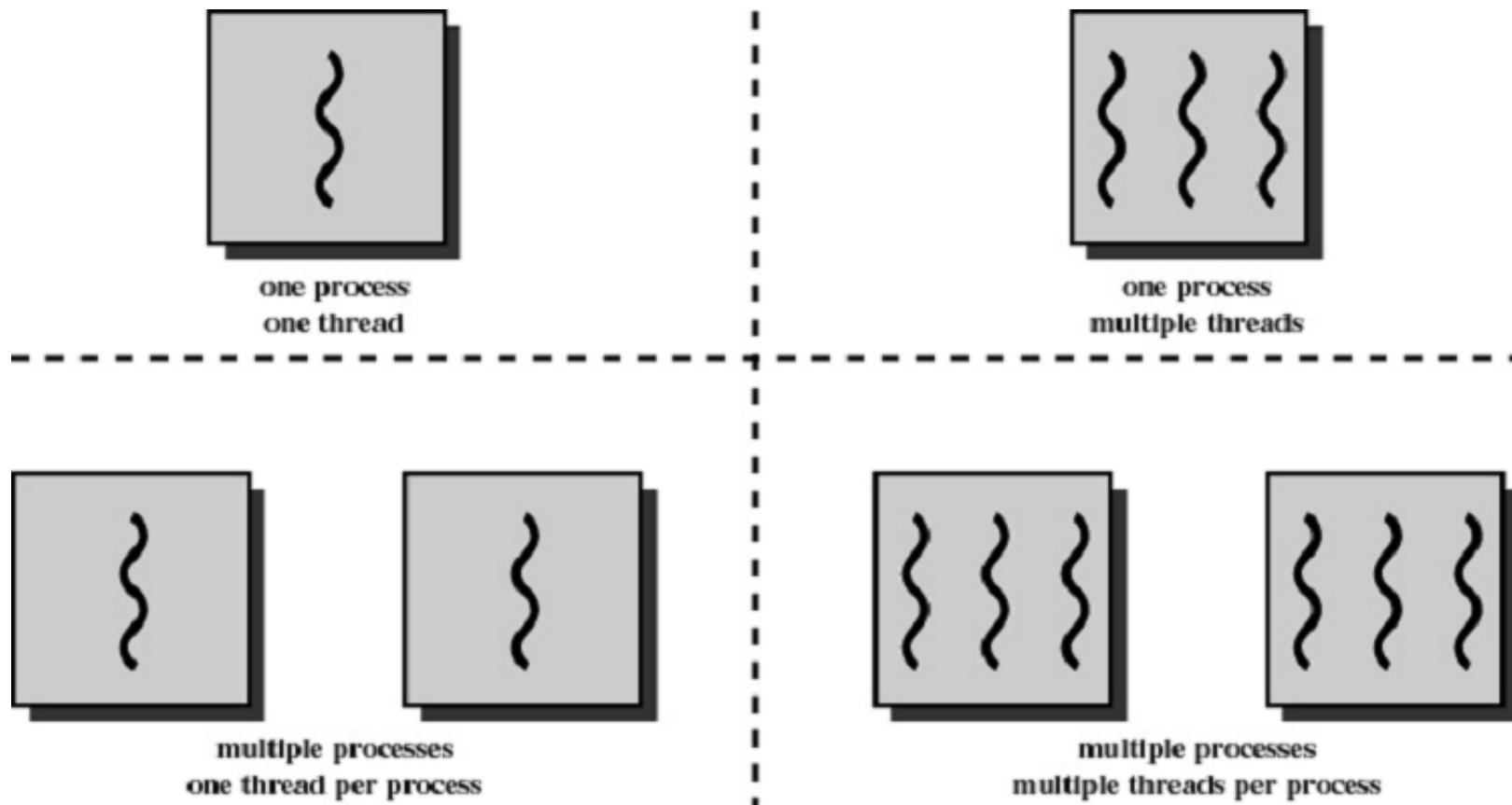
# 线程的诞生

- 为区分这两个特点，调度并分派的部分通常称为线程或轻量级进程（light-weight process），而资源所有权的部分通常称为进程。
- 传统的进程中只有一个线程，称作单线程进程。

# 发展

- 进程是拥有资源和独立运行的基本单位。
- 20世纪80年代中期，提出了比进程更小的能独立运行的基本单位——线程（Thread）
- 用来提高系统内程序并发度，进一步提高系统吞吐量。
- 20世纪90年代后，多处理机系统迅速发展，线程能比进程更好地提高程序的并行执行程度，充分地发挥多处理机的优越性。

# 进程与线程的关系



# 线程的优势

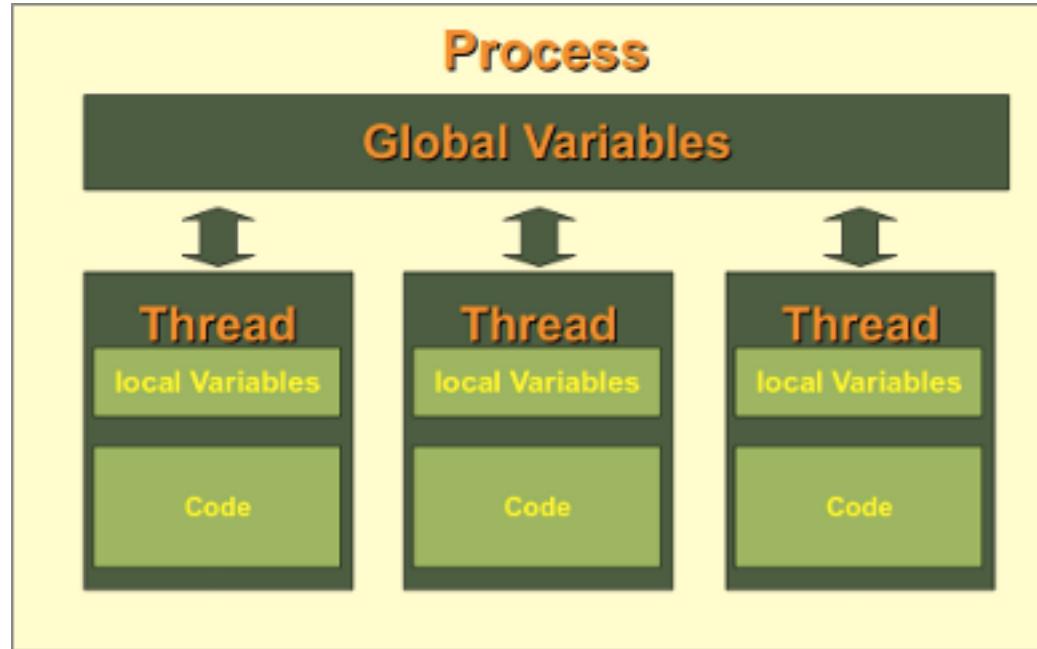
- 减少并发执行时的时空开销
  - 进程的创建、撤销、切换开销较费
- 线程是系统独立调度的基本单位
  - 基本不拥有系统资源，只有少量资源（PC，寄存器，栈），**共享其所属进程所拥有的全部资源。**

# 特点

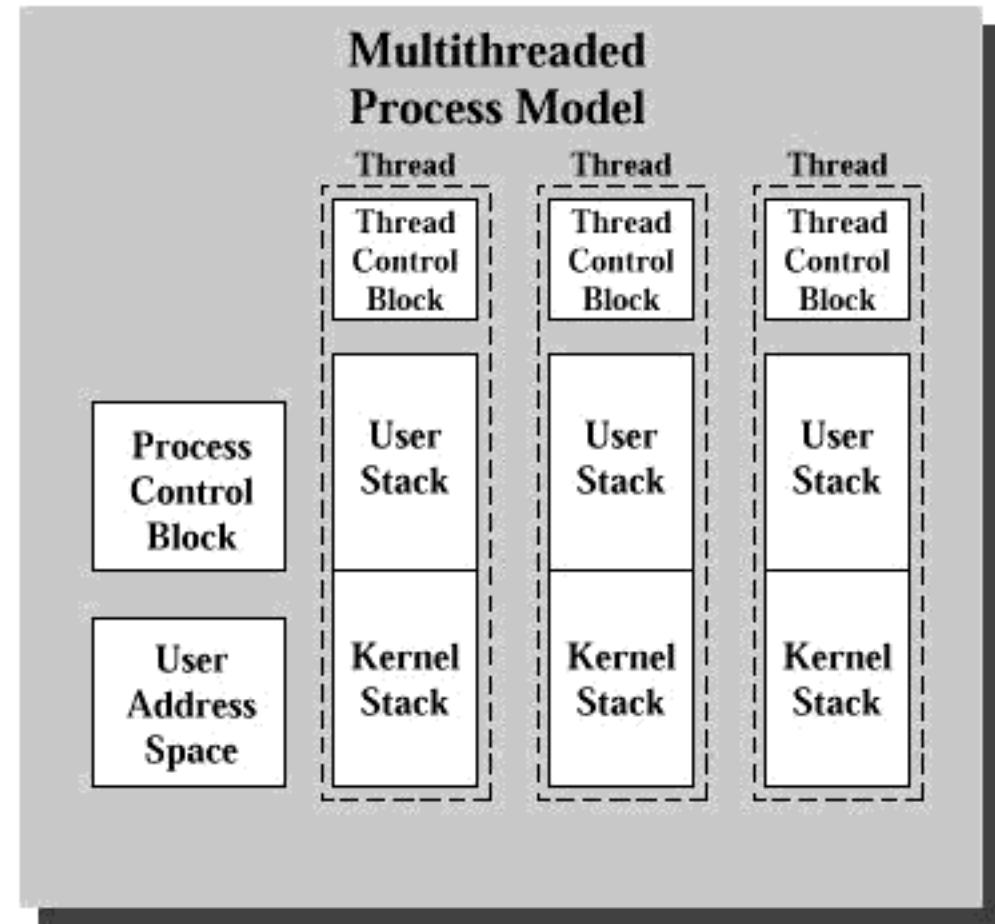
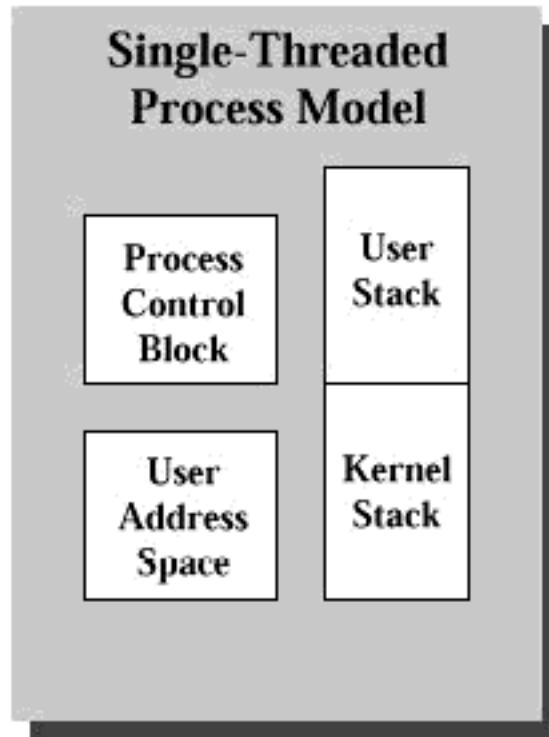
- 单线程进程中，包括进程控制块和用户地址空间，以及用户栈和内核栈。
- 多线程环境中，有一个与进程相关联的进程控制块和用户地址空间，每个线程都有一个独立的栈和独立的控制块，包含寄存器值，优先级和其他与线程相关的状态信息。
- 进程中的所有线程共享该进程的资源，驻留在同一块地址空间中，并且可以访问到相同的数据。
- 线程阻塞不一定会引起进程阻塞。

# 线程数据

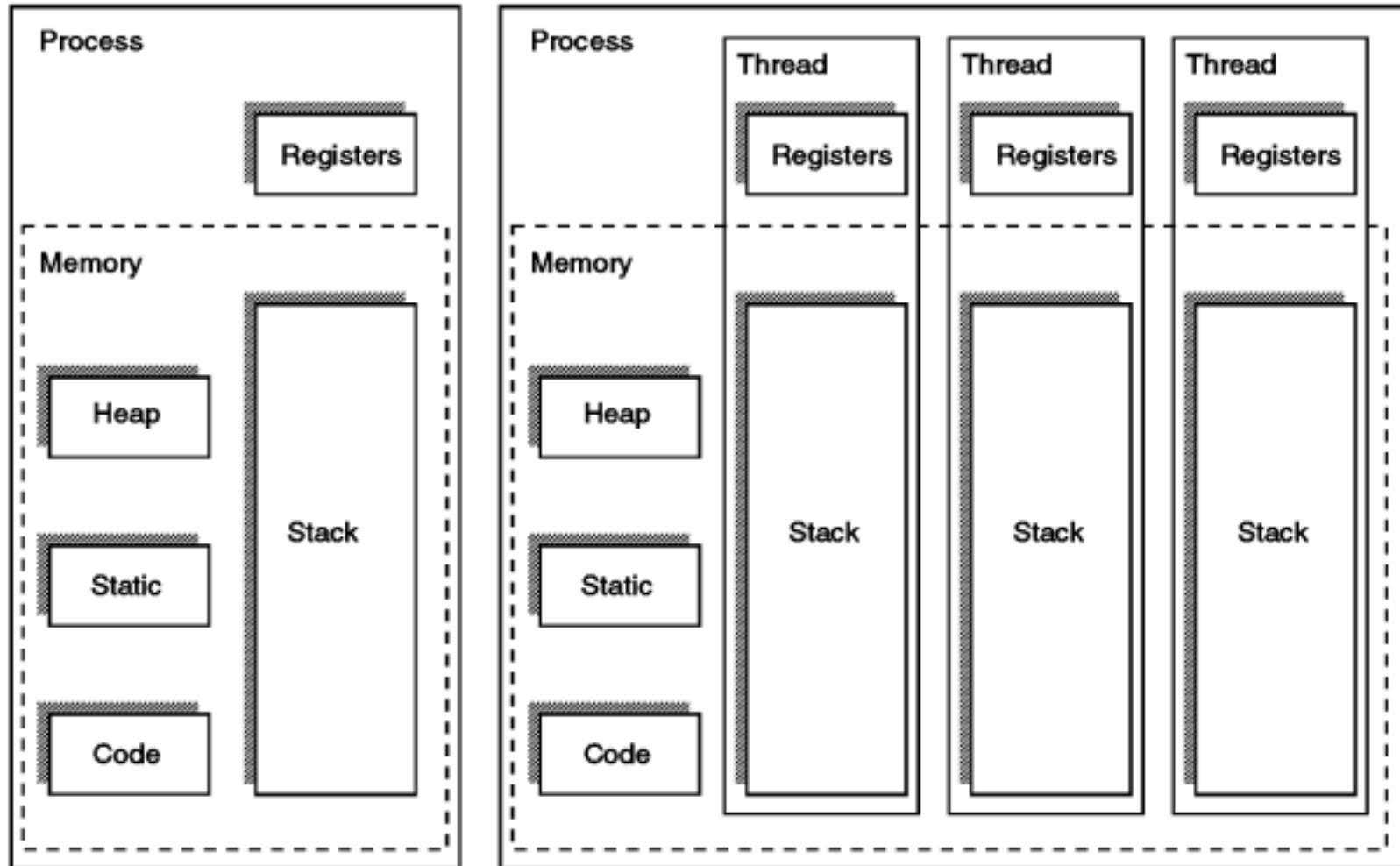
- 状态参数
  - 寄存器状态
  - 堆栈
  - 优先级
  - 线程专有存储器
    - TLS: Thread Local Storage
  - 信号屏蔽
  - 运行状态



# 单线程和多线程模式



# 进程与线程



# 线程状态

- 执行状态、阻塞状态、就绪状态等。
- 在线程切换时保存的线程信息
  - 一个执行栈
  - 每个线程静态存储局部变量
  - 对存储器和其进程资源的访问。

# 线程状态

- 派生 (Spawn)：当产生一个新进程时，同时也为该进程派生了一个线程，随后，进程中的线程可以在同一个进程中派生另一个线程，新线程被放置在就绪队列中。
- 结束 (Finish)：线程完成时，其寄存器信息和栈都被释放。

# 线程状态

- 就绪 (Ready)
- 运行 (Running)
- 阻塞 (Block): 当线程需要等待一个事件时，它将阻塞，此时处理器转而执行另一个就绪线程。

# 思考

- 在同一进程中的线程切换是否会引起进程切换？
- 在不同进程中的线程切换是否会引起进程切换？

# 线程分类

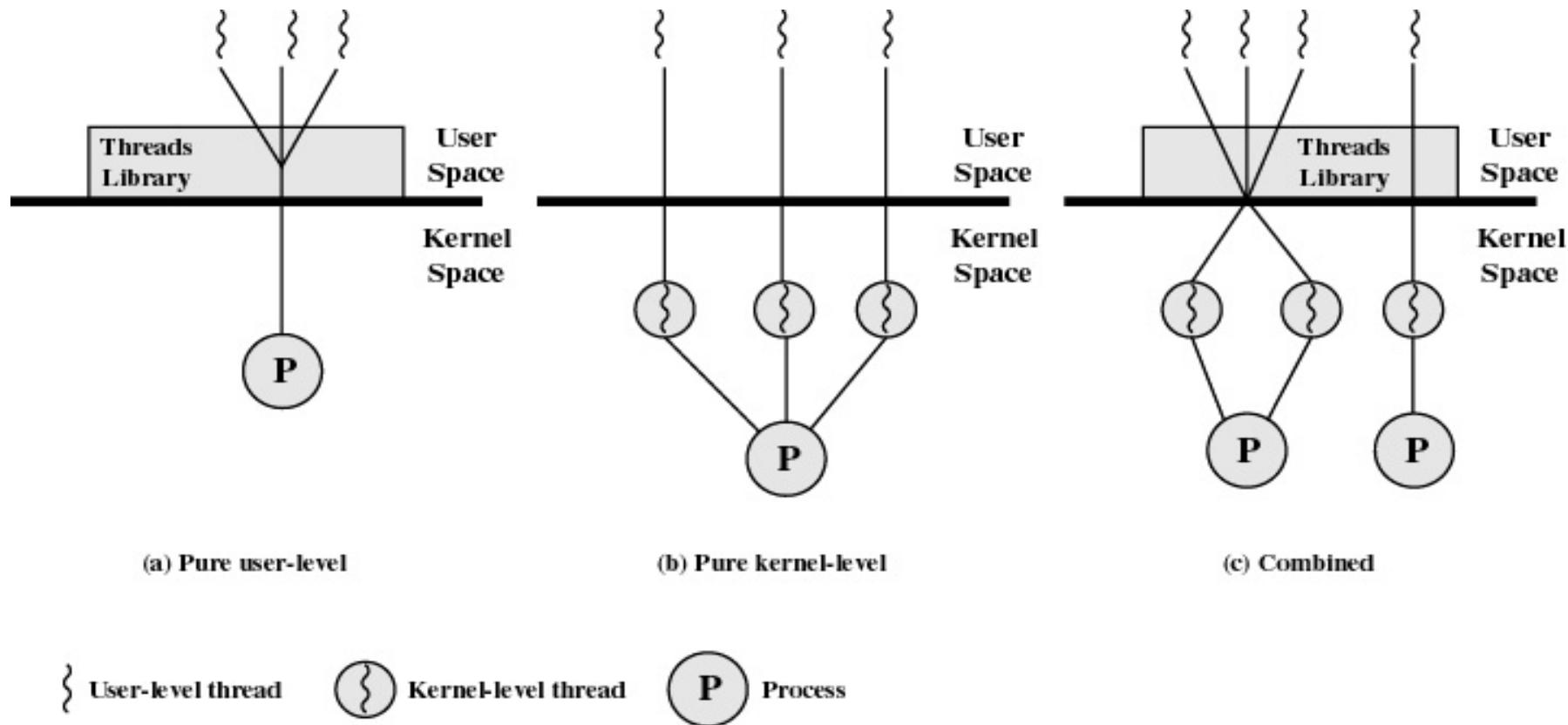
内核级  
线程

- 每个线程在内核看来都是一个进程

用户级  
线程

- 内核无法感知，用户自己控制

# 内核级和用户级线程



**Figure 4.6 User-Level and Kernel-Level Threads**

# 内核级线程

- 内核级线程，是在内核的支持下运行的，即无论是用户进程中的线程，还是系统进程中的线程，其创建、撤销和切换等，都是依靠内核实现的。
- 内核为每一个内核线程设置了一个线程控制块，根据该控制块感知线程的存在，并对其进行控制。

# 用户级线程

- 用户级线程的创建、撤消、线程之间的同步与通信等功能，都无须利用系统调用来实现。
- 用户级线程的切换，通常是发生在一个应用进程的诸多线程之间，无须内核的支持。切换的规则比进程调度和切换的规则简单，切换速度快。
- 线程控制块设置在用户空间，内核完全不知道用户级线程的存在，可以节省系统开销。
- 当引起进程阻塞时，会削弱进程中的线程的并发性。

# 比较

- 内核线程和用户线程有哪些不同？

# 调度

- 内核级线程切换类似于进程切换，开销较大。
- 用户级线程切换发生在同一用户进程中，无需进入内核，更快。

# 系统调用

- 用户线程：某一个线程进行系统调用，由于内核不知道这些线程的存在，因此将进程阻塞。
- 内核线程：某一线程进行系统调用，则阻塞该线程，进程（其它线程）仍可运行。

# 执行时间

- 用户级线程以进程为单位平均分配时间，对线程间并发执行不利（一个CPU）。
- 内核级线程以线程为单位分配时间（多个CPU）。

# 线程实例：Java

```
public class HelloRunnable implements Runnable {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```

Thread

Runnable

```
public class HelloThread extends Thread {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
}
```

# 线程实例：Windows

```
DWORD WINAPI MyThreadFunction(LPVOID lpParam) {  
    printf("The parameter: %u.\n", *(DWORD*)lpParam);  
    return 0;  
}  
  
int main(void) {  
    DWORD dwThreadId, dwThrdParam = 1;  
    HANDLE hThread;  
  
    hThread = CreateThread(NULL, // default security attributes  
                           0, // use default stack size  
                           MyThreadFunction, // thread function  
                           &dwThrdParam, // argument to thread function  
                           0, // use default creation flags  
                           &dwThreadId); // returns the thread id  
  
    printf("The thread ID: %d.\n", dwThreadId);
```

# 线程实例：Windows

```
// Check the return value for success. If something wrong...
if (hThread == NULL)
    printf("CreateThread() failed, error: %d.\n", GetLastError());
else
    printf("It seems the CreateThread() is OK lol!\n");

if (CloseHandle(hThread) != 0)
    printf("Handle to thread closed successfully.\n");

return 0;
}
```

# 线程实例：Linux pthread

```
#define NUM_THREADS      5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}
```

# 线程实例：Linux pthread

```
int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

# Windows的特例

- Fiber(纤程): 线程内部的“线程”
- A fiber is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them. Each thread can schedule multiple fibers. In general, fibers do not provide advantages over a well-designed multithreaded application. However, using fibers can make it easier to port applications that were designed to schedule their own threads.

# 线程池

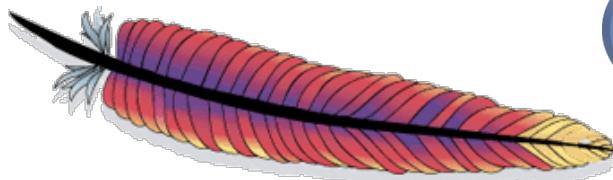
- thread pool
- 一组工人

# 推荐阅读

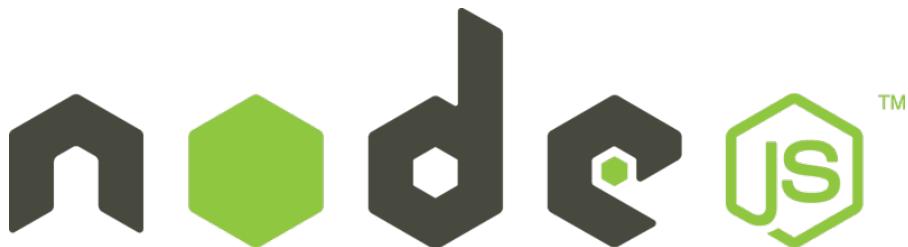
- 协程(co-routine)@1963
  - 特殊的并发模型：独立的栈，局部变量和指令指针，同时又与其它协同程序共享全局变量和其它大部分数据，并非“线程”的并发。
  - 类似用户级线程
    - 但控制方式，数据交互比线程灵活
  - 更像函数调用
    - 但可以有多个入口：破坏了现代程序设计思想中函数单入口的原则
  - 更低资源消耗

# 时代的选择：并发方案

- 多线程 vs 单线程



Apache



# 参考资料

- [http://en.wikipedia.org/wiki/Thread\\_\(computing\)](http://en.wikipedia.org/wiki/Thread_(computing))
- <http://en.wikipedia.org/wiki/Coroutine>
- [http://en.wikipedia.org/wiki/POSIX\\_Threads](http://en.wikipedia.org/wiki/POSIX_Threads)
- [http://msdn.microsoft.com/en-us/library/windows/desktop/ms684841\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms684841(v=vs.85).aspx)

# 谢谢！