

# 操作系统

## 第2章1-3 进程的描述和控制

朱小军

南京航空航天大学计算机科学与技术学院

2025年春

# 提问环节

- 什么是内核态和用户态？
- 什么是系统调用，和普通的函数调用有何不同？
- 什么是特权指令？举一个例子。

裸机编程时，内存可以任意使用，自由度极大，但

- CPU可能混淆数据与指令（后果？）
- 程序可能修改自己（后果？）


留意操作系统提供的“进程”抽象如何避免这两个问题

# 进程概念

# 进程是什么？

- 管理CPU，需要描述CPU在执行什么
  - 多道批处理时，说CPU在执行程序A、程序B
  - 但同时执行程序A两次，如何区分？
  - 可以说，A的第一次执行，A的第二次执行
  - A程序的第x次执行
- 为了描述方便，程序的一次执行称为一个进程
- 名字是用于区分的，先有了分别，才有名字

程序之于进程，就像菜谱之于做菜



这是一段程序



## 详细信息



运行新任务



结束任务



名称	PID	状态	用户名	CPU	内存(活动的...	体系结构	描述
acrotray.exe	19112	正在运行	xzhu	00	672 K	x86	AcroTray
aesm_service.exe	21196	正在运行	SYSTEM	00	24 K	x64	Intel® SGX A
AggregatorHost.exe	9612	正在运行	SYSTEM	00	1,188 K	x64	Microsoft (R)
AGMSERVICE.exe	6172	正在运行	SYSTEM	00	1,484 K	x86	Adobe Genuin
AGSSERVICE.exe	6192	正在运行	SYSTEM	00	3,292 K	x86	Adobe Genuin
ai.exe	27124	正在运行	xzhu	00	12,508 K	x64	Artificial Intell
AlibabaProtect.exe	19372	正在运行	SYSTEM	00	19,056 K	x86	Alibaba PC Sa
AppGalleryAMS.exe	8444	正在运行	SYSTEM	00	44 K	x64	AppGalleryAM
AppGalleryService.exe	6216	正在运行	SYSTEM	00	308 K	x64	AppGallerySe
ApplicationFrameHost.exe	2420	正在运行	xzhu	00	3,952 K	x64	Application Fr
BasicService.exe	6636	正在运行	SYSTEM	00	11,776 K	x64	BasicService
chrome.exe	22828	正在运行	xzhu	00	109,916 K	x64	Google Chron
chrome.exe	316	正在运行	xzhu	00	636 K	x64	Google Chron
chrome.exe	23144	正在运行	xzhu	00	169,680 K	x64	Google Chron
chrome.exe	5924	正在运行	xzhu	00	24,956 K	x64	Google Chron
chrome.exe	5332	正在运行	xzhu	00	2,816 K	x64	Google Chron
chrome.exe	2284	正在运行	xzhu	00	1,808 K	x64	Google Chron
chrome.exe	26284	正在运行	xzhu	00	24,360 K	x64	Google Chron
chrome.exe	26560	正在运行	xzhu	00	105,540 K	x64	Google Chron
chrome.exe	26632	正在运行	xzhu	00	31,260 K	x64	Google Chron
chrome.exe	15928	正在运行	xzhu	00	10,788 K	x64	Google Chron
chrome.exe	26304	正在运行	xzhu	00	9,248 K	x64	Google Chron
ChsIME.exe	14412	正在运行	xzhu	00	444 K	x64	Microsoft IME
Code.exe	5464	正在运行	xzhu	00	19,532 K	x64	Visual Studio
Code.exe	12072	正在运行	xzhu	00	596 K	x64	Visual Studio
Code.exe	21144	正在运行	xzhu	00	24,904 K	x64	Visual Studio



```

1  [ | 0.7%] Tasks: 44, 133 thr; 1 running
2  [ || 2.0%] Load average: 0.05 0.10 0.05
Mem[|||||||||||||||||||||||||||||||||||||||||292M/1.79G] Uptime: 10 days, 00:19:45
Swp[|||||||||||||||||||||||||||||||||||||||||0K/0K]

```

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
1642	root	20	0	553M	25448	5720	S	1.3	1.4	1h20:16	barad_agent
206608	root	20	0	553M	25448	5720	S	0.7	1.4	0:00.01	barad_agent
1975	root	20	0	1015M	75828	30120	S	0.0	4.0	1h38:21	/usr/local/qcloud/YunJing/YDEyes/YDService
206506	ubuntu	20	0	9444	4596	3472	R	0.0	0.2	0:00.08	htop
1667	root	20	0	553M	25448	5720	S	0.0	1.4	19:53.67	barad_agent
1710	root	20	0	553M	25448	5720	S	0.0	1.4	2:31.29	barad_agent
173347	ubuntu	20	0	14048	5396	3904	S	0.0	0.3	0:00.03	sshd: ubuntu@pts/1
2016	root	20	0	1015M	75828	30120	S	0.0	4.0	7:54.94	/usr/local/qcloud/YunJing/YDEyes/YDService
1976	root	20	0	1015M	75828	30120	S	0.0	4.0	21:14.05	/usr/local/qcloud/YunJing/YDEyes/YDService
370	root	20	0	20916	5748	3752	S	0.0	0.3	0:02.68	/lib/systemd/systemd-udev
894	root	20	0	232M	7568	6456	S	0.0	0.4	0:10.66	/usr/lib/accounts-service/accounts-daemon
1978	root	20	0	1015M	75828	30120	S	0.0	4.0	7:25.13	/usr/local/qcloud/YunJing/YDEyes/YDService
2008	root	20	0	1015M	75828	30120	S	0.0	4.0	5:59.93	/usr/local/qcloud/YunJing/YDEyes/YDService
1977	root	20	0	1015M	75828	30120	S	0.0	4.0	17:59.73	/usr/local/qcloud/YunJing/YDEyes/YDService
2042	root	20	0	1015M	75828	30120	S	0.0	4.0	1:29.80	/usr/local/qcloud/YunJing/YDEyes/YDService
2071	root	20	0	1015M	75828	30120	S	0.0	4.0	7:27.95	/usr/local/qcloud/YunJing/YDEyes/YDService
1062	root	20	0	39348	21124	2300	S	0.0	1.1	6:25.54	/usr/local/sa/agent/secu-tcs-agent
2027	root	20	0	1015M	75828	30120	S	0.0	4.0	1:09.77	/usr/local/qcloud/YunJing/YDEyes/YDService
7979	root	20	0	1275M	10572	7964	S	0.0	0.6	0:37.59	/usr/local/qcloud/YunJing/YDLive/YDLive
2104	root	20	0	1275M	7820	4120	S	0.0	0.4	2:11.53	/bin/sh -c sleep 100
1	root	20	0	103M	11180	7948	S	0.0	0.6	0:09.59	/sbin/init
335	root	19	-1	202M	144M	143M	S	0.0	7.9	0:41.63	/lib/systemd/systemd-journald
592	root	RT	0	274M	17972	8224	S	0.0	1.0	0:02.71	/sbin/multipathd -d -s
593	root	RT	0	274M	17972	8224	S	0.0	1.0	0:00.00	/sbin/multipathd -d -s
594	root	RT	0	274M	17972	8224	S	0.0	1.0	0:00.39	/sbin/multipathd -d -s
595	root	RT	0	274M	17972	8224	S	0.0	1.0	0:22.61	/sbin/multipathd -d -s
596	root	RT	0	274M	17972	8224	S	0.0	1.0	0:00.00	/sbin/multipathd -d -s
597	root	RT	0	274M	17972	8224	S	0.0	1.0	0:00.00	/sbin/multipathd -d -s
591	root	RT	0	274M	17972	8224	S	0.0	1.0	0:35.68	/sbin/multipathd -d -s
647	_rpc	20	0	8164	3768	3332	S	0.0	0.2	0:00.61	/sbin/rpcbind -f -w
877	systemd-n	20	0	28312	7364	6480	S	0.0	0.4	0:00.87	/lib/systemd/systemd-networkd
879	systemd-r	20	0	25596	11372	7468	S	0.0	0.6	0:02.19	/lib/systemd/systemd-resolved
907	root	20	0	232M	7568	6456	S	0.0	0.4	0:10.52	/usr/lib/accounts-service/accounts-daemon
1019	root	20	0	232M	7568	6456	S	0.0	0.4	0:00.02	/usr/lib/accounts-service/accounts-daemon
898	root	20	0	8736	2960	2668	S	0.0	0.2	0:08.03	/usr/sbin/cron -f
899	messagebu	20	0	8796	4440	3688	S	0.0	0.2	0:00.44	/usr/bin/dbus-daemon --system --address=systemd: --nofo
966	root	20	0	437M	2140	1604	S	0.0	0.1	0:00.15	/usr/bin/lxcfs /var/lib/lxcfs
967	root	20	0	437M	2140	1604	S	0.0	0.1	0:00.18	/usr/bin/lxcfs /var/lib/lxcfs
135044	root	20	0	437M	2140	1604	S	0.0	0.1	0:00.16	/usr/bin/lxcfs /var/lib/lxcfs



# 进程概念

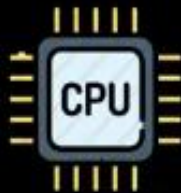
- 进程的内存抽象
- 进程的状态
- 进程控制块与进程切换
- 进程创建与终止

# 进程进入内存后，拥有独立的地址空间



内存

进程



程序在编写（以及编译）时，潜在假设了OS提供某些功能（如内存布局），所以能在Windows上运行的不能在Linux上运行。

菜谱其实也有潜在建设，欧美的菜谱能在中国的厨房做出来吗？

# 程序中哪些地方涉及到了内存?

```
#include <stdlib.h>
```

```
int data = 42;
```

```
int main() {  
    int stack;  
    int* heap = malloc(sizeof(int));  
    free(heap);  
    return 0;  
}
```

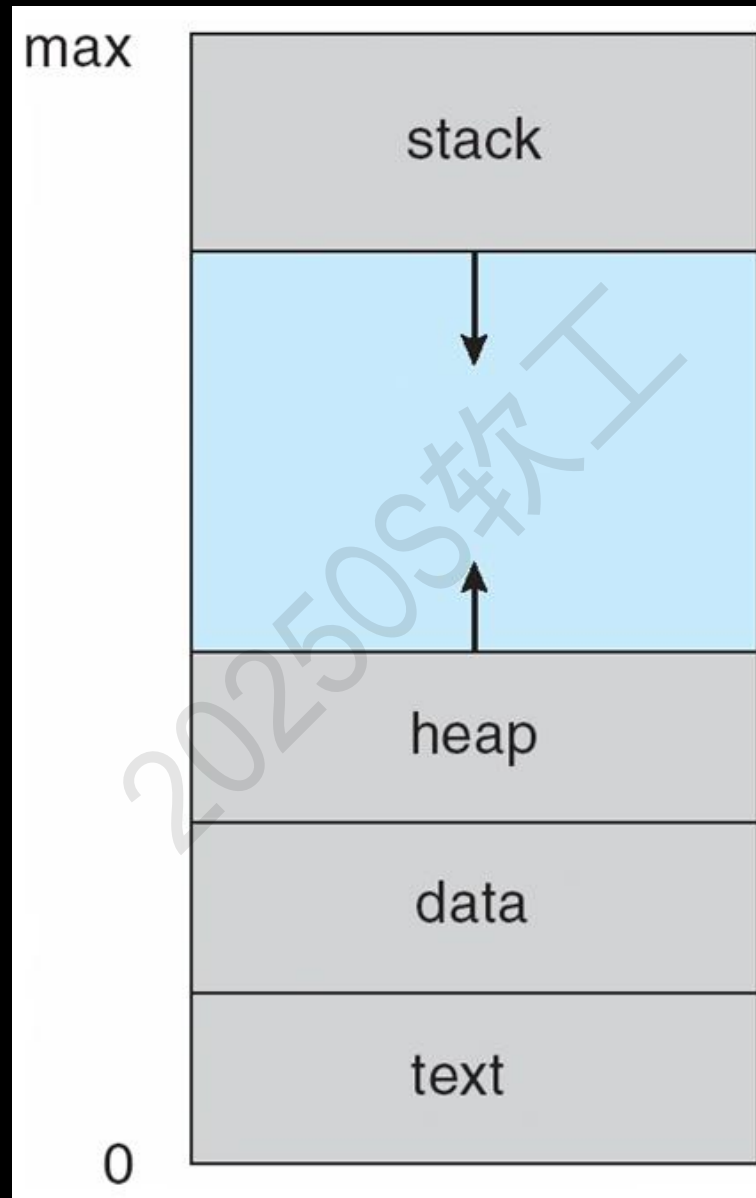
C语言



```
.globl    data  
.data  
.align 4  
  
data:  
  
    .long    42  
  
    .text  
  
main:  
  
    pushq    %rbp  
    movq     %rsp, %rbp  
    subq     $16, %rsp  
    movl     $4, %edi  
    call     malloc@PLT  
    movq     %rax, -8(%rbp)  
    movq     -8(%rbp), %rax  
    movq     %rax, %rdi  
    call     free@PLT  
    movl     $0, %eax  
    leave  
    ret
```

汇编语言

# (某操作系统提供的) 进程的内存抽象





- 代码段text
  - 指令
- 数据段data
  - 全局数据，静态变量
- 堆heap
  - new, malloc
- 栈stack
  - 局部变量，子函数返回地址
- 注意
  - 各个段之间可能有空洞
  - 不同操作系统的布局稍不同（以上为Unix）
  - 不合法访问导致异常，进程被强制终止

```
#include <stdlib.h>
```

```
int data = 42;
```

```
int main() {
    int stack;
    int* heap = malloc(sizeof(int));
    free(heap);
    return 0;
}
```

```

                                .globl    data
                                .data
                                .align 4

data:
                                .long     42

                                .text
main:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $16, %rsp
    movl     $4, %edi
    call     malloc@PLT
    movq     %rax, -8(%rbp)
    movq     -8(%rbp), %rax
    movq     %rax, %rdi
    call     free@PLT
    movl     $0, %eax
    leave
    ret

```

分段之后，操作系统可提供一定的保护。

# 向代码段写数据

```
1 #include "stdio.h"
2
3 void f(){
4     printf("world\n");
5 }
6
7 void main(){
8     printf("hello\n");
9     *(int *)f=3;
10    f();
11 }
```

# 无穷递归程序会终止吗？

```
1 void f(){  
2     int i[4096];  
3     f();  
4 }  
5  
6 void main(){  
7     f();  
8 }
```

# 小结

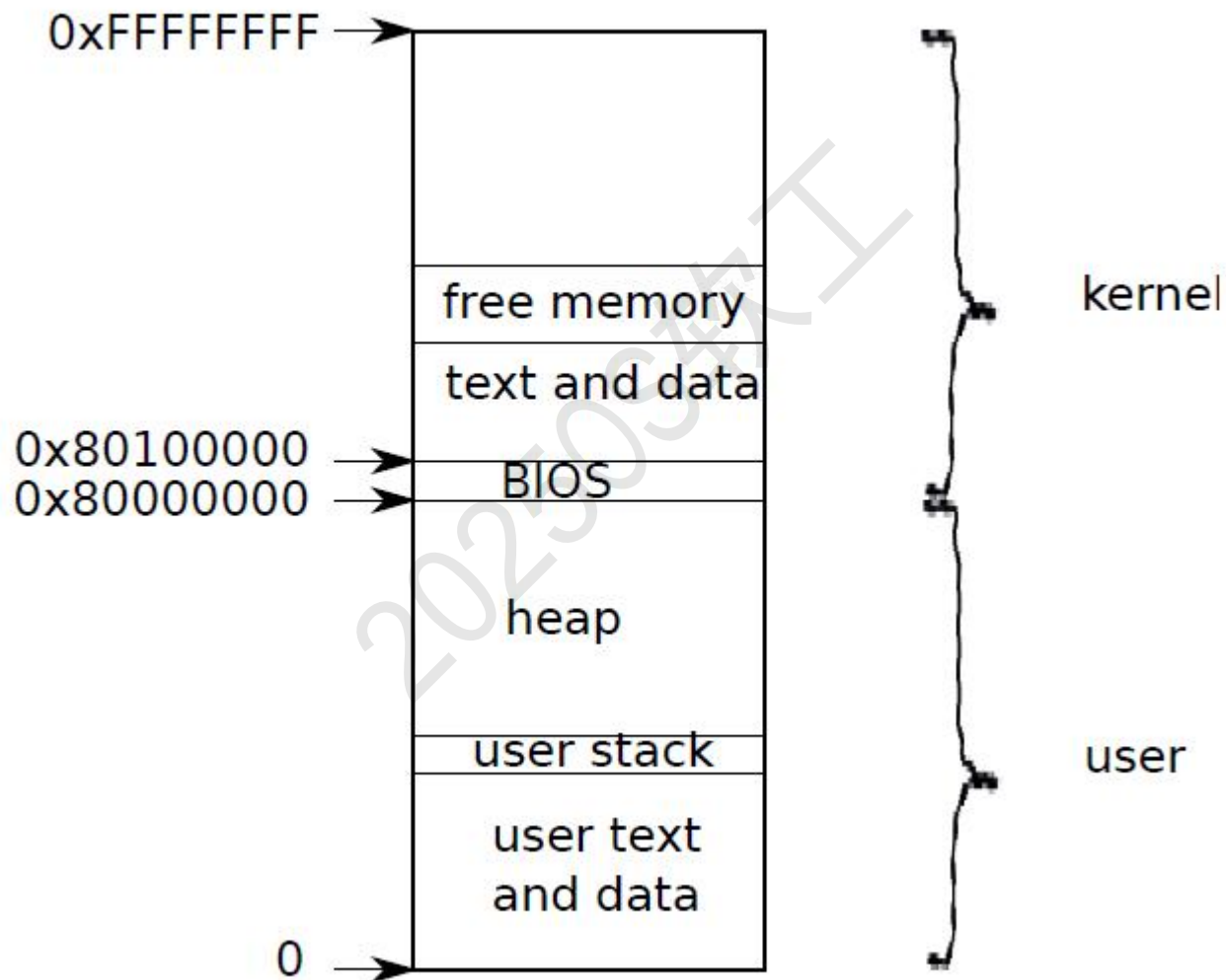
- OS创建进程时，会为每个进程创建独立的内存抽象，互不相交
- 栈溢出指的是栈段的地址用完了，不是物理内存用完了
- data、heap、stack存放的都是通常意义上说的“数据”，与指令相对
- OS根据分段对逻辑地址进行保护，但目前的保护不完善，导致缓冲区溢出攻击的存在
  - 延伸阅读：NX bit



# 从另一个角度回顾进程的内存抽象

- 程序在磁盘中，要运行时，OS会创建了一种“幻觉”(illusion)，让程序认为自己独占CPU、独占内存 == 进程
  - 按照这种幻觉执行，和真的在物理机上独占执行，结果一样！
  - 程序运行时能感知到自己在幻觉中吗？
    - 程序运行中发出的每个地址，都是幻觉中的地址！
- 幻觉中的内存分成若干区（或段），栈、堆、数据、代码，程序设计课程中提到的“堆”，就是幻觉中的堆区！“栈溢出”中的“栈”，就是幻觉中的“栈”区。
  - 操作系统提供的幻觉，还有特别的约束，比如相对位置、读写权限
  - 为什么提供的幻觉长这样？可能和最原始的编程模式有关
- 编译器在将高级语言翻译成机器语言时，需要知道操作系统提供什么样的“幻觉”。各种变量的地址分配

# 案例：xv6操作系统中进程的内存



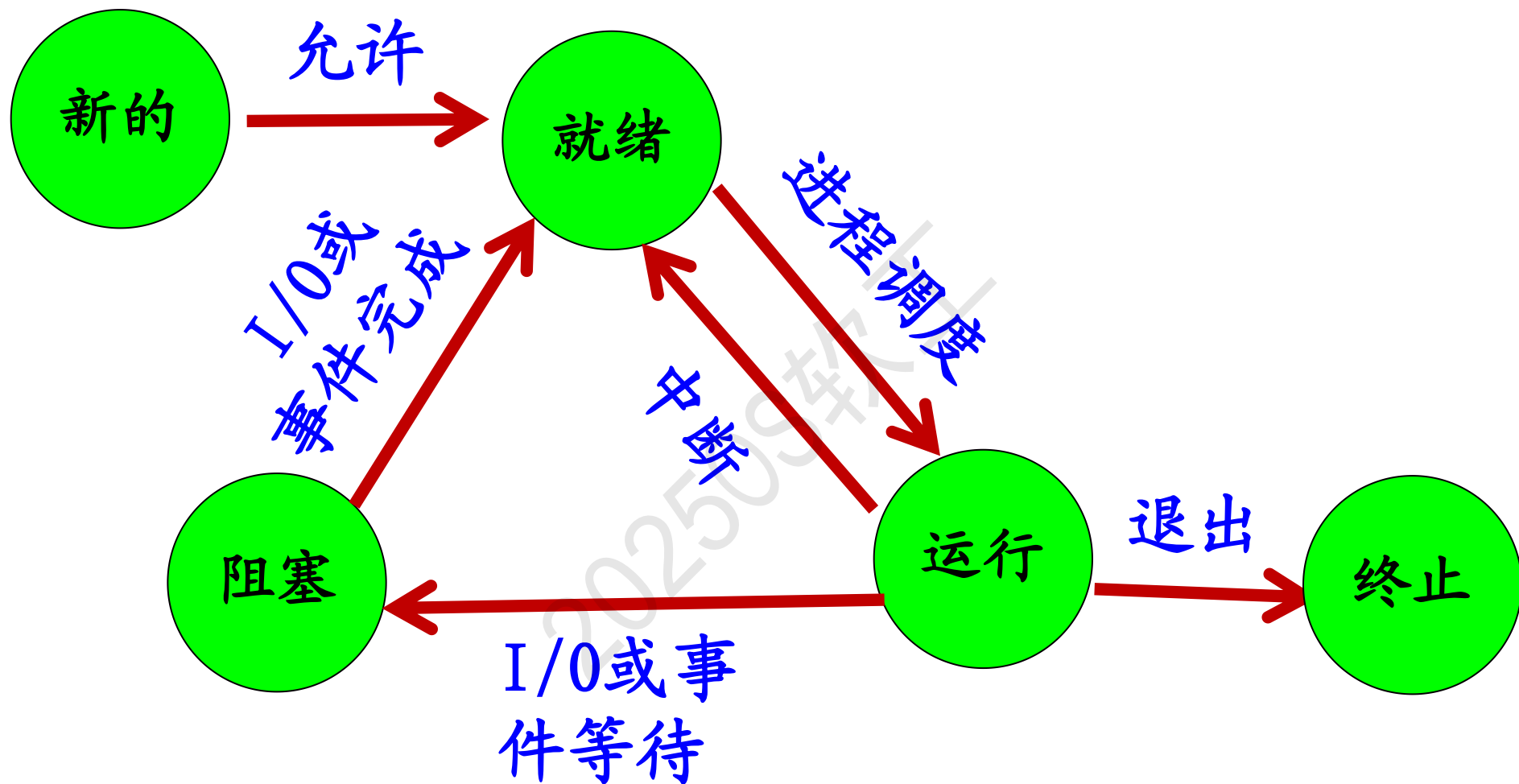
# 进程概念

- 内存中的进程
- 进程的状态
- 进程控制块与进程切换
- 进程创建与终止

# 执行、阻塞、就绪

- 单道批处理系统，进程的状态有两种（从CPU的角度看）
  - 用CPU、不用CPU（IO操作）
  - 前者命名为**执行**，后者命名为**阻塞**
- 多道批处理系统，进程的状态有几种？
  - 用CPU、不用CPU
  - 后者细分为两种，不想用（IO操作）、等着用
  - 执行、阻塞、就绪

# 进程的状态



- 部分操作系统会扩展更多状态，如僵尸、挂起阻塞、挂起就绪等



# 动画演示

---

2025OS软工

# 进程概念

- 进程的内存抽象
- 进程的状态
- 进程控制块与进程切换
- 进程创建与终止

# 进程控制块 (PCB)

- 内核管理进程的数据结构

- 与每个进程相关的信息

进程状态

进程编号

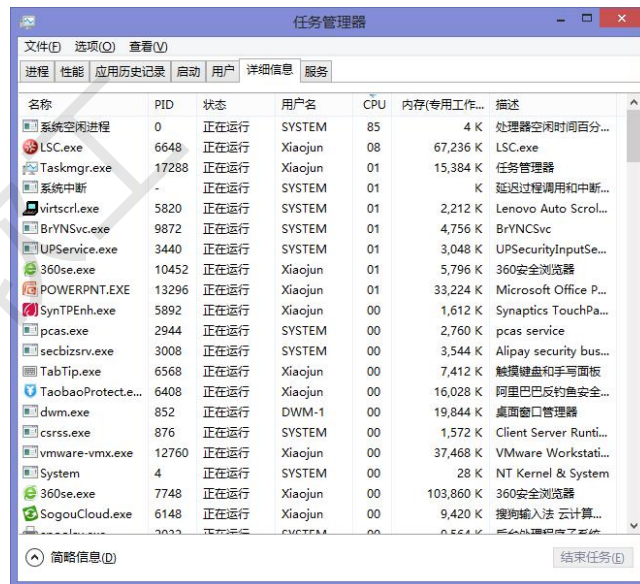
程序计数器PC (IP)

寄存器

内存界限

打开文件列表

。 。 。



The screenshot shows the Windows Task Manager window with the 'Processes' tab selected. It displays a list of running processes with columns for Name, PID, Status, Username, CPU, Memory (Private Working Set), and Description. The list includes system processes like 'System Idle Process' and 'smss.exe', as well as user applications like 'LSC.exe', 'Taskmgr.exe', and 'SogouCloud.exe'.

名称	PID	状态	用户名	CPU	内存(专用工作...	描述
系统空闲进程	0	正在运行	SYSTEM	85	4 K	处理器空闲时间百分...
smss.exe	6648	正在运行	Xiaojun	08	67,236 K	LSC.exe
Taskmgr.exe	17288	正在运行	Xiaojun	01	15,384 K	任务管理器
系统中断	-	正在运行	SYSTEM	01	K	延迟过程调用和中断...
virtscrl.exe	5820	正在运行	SYSTEM	01	2,212 K	Lenovo Auto Scrol...
BrYNSvc.exe	9872	正在运行	SYSTEM	01	4,756 K	BrYNSvc
UPService.exe	3440	正在运行	SYSTEM	01	3,048 K	UPSecurityInputSe...
360se.exe	10452	正在运行	Xiaojun	01	5,796 K	360安全浏览器
POWERPNT.EXE	13296	正在运行	Xiaojun	01	33,224 K	Microsoft Office P...
SynTPEnh.exe	5892	正在运行	Xiaojun	00	1,612 K	Synaptics TouchPa...
pcas.exe	2944	正在运行	SYSTEM	00	2,760 K	pcas service
secbizsrv.exe	3008	正在运行	SYSTEM	00	3,544 K	Alipay security bus...
TabTip.exe	6568	正在运行	Xiaojun	00	7,412 K	触摸键盘和手写面板
TaobaoProtect.e...	6408	正在运行	Xiaojun	00	16,028 K	阿里巴巴安全防护...
dwm.exe	852	正在运行	DWM-1	00	19,844 K	桌面窗口管理器
csrss.exe	876	正在运行	SYSTEM	00	1,572 K	Client Server Runti...
vmware-vmx.exe	12760	正在运行	Xiaojun	00	37,468 K	VMware Workstati...
System	4	正在运行	SYSTEM	00	28 K	NT Kernel & System
360se.exe	7748	正在运行	Xiaojun	00	103,860 K	360安全浏览器
SogouCloud.exe	6148	正在运行	Xiaojun	00	9,420 K	搜狗输入法 云计算...

- 不同OS的PCB不一样，就像不同教务系统中学生的属性不一样

- 数据存储在内核空间中，应用程序不可直接访问

- 类比：个人档案在组织部门，个人不可直接访问

如果是你写程序，你打算用什么数据结构组织 PCB？

- ☐ A 链表，每个PCB是一个链表节点
- ☐ B 数组，每个PCB是一个数组元素
- ☐ C 树，每个PCB是一个树节点
- ☐ D 图，每个PCB是一个图节点

提交

# xv6 操作系统中的proc结构

- proc.h

```
struct context {
    uint edi;
    uint esi;
    uint ebx;
    uint ebp;
    uint eip;
};

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
    uint sz; // Size of process memory (bytes)
    pde_t* pgdir; // Page table
    char *kstack; // Bottom of kernel stack for this process
    enum procstate state; // Process state
    int pid; // Process ID
    struct proc *parent; // Parent process
    struct trapframe *tf; // Trap frame for current syscall
    struct context *context; // switch() here to run process
    void *chan; // If non-zero, sleeping on chan
    int killed; // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd; // Current directory
    char name[16]; // Process name (debugging)
};
```



# 静态分配了NPROC个proc结构体

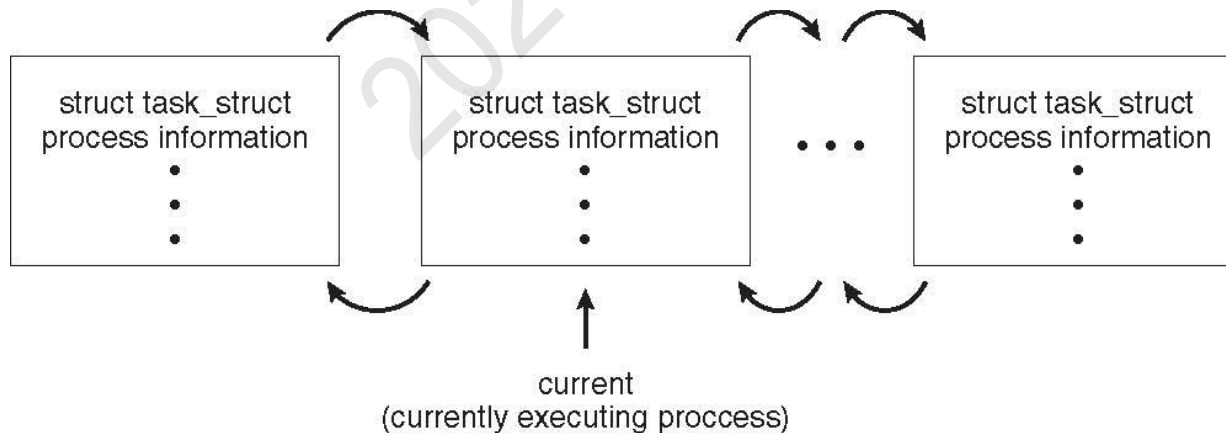
```
struct {  
    struct spinlock lock;  
    struct proc proc[NPROC];  
} ptable;
```

```
// Loop over process table looking for process to run.  
acquire(&ptable.lock);  
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  
    if(p->state != RUNNABLE)  
        continue;
```

# Linux操作系统中的任务控制块

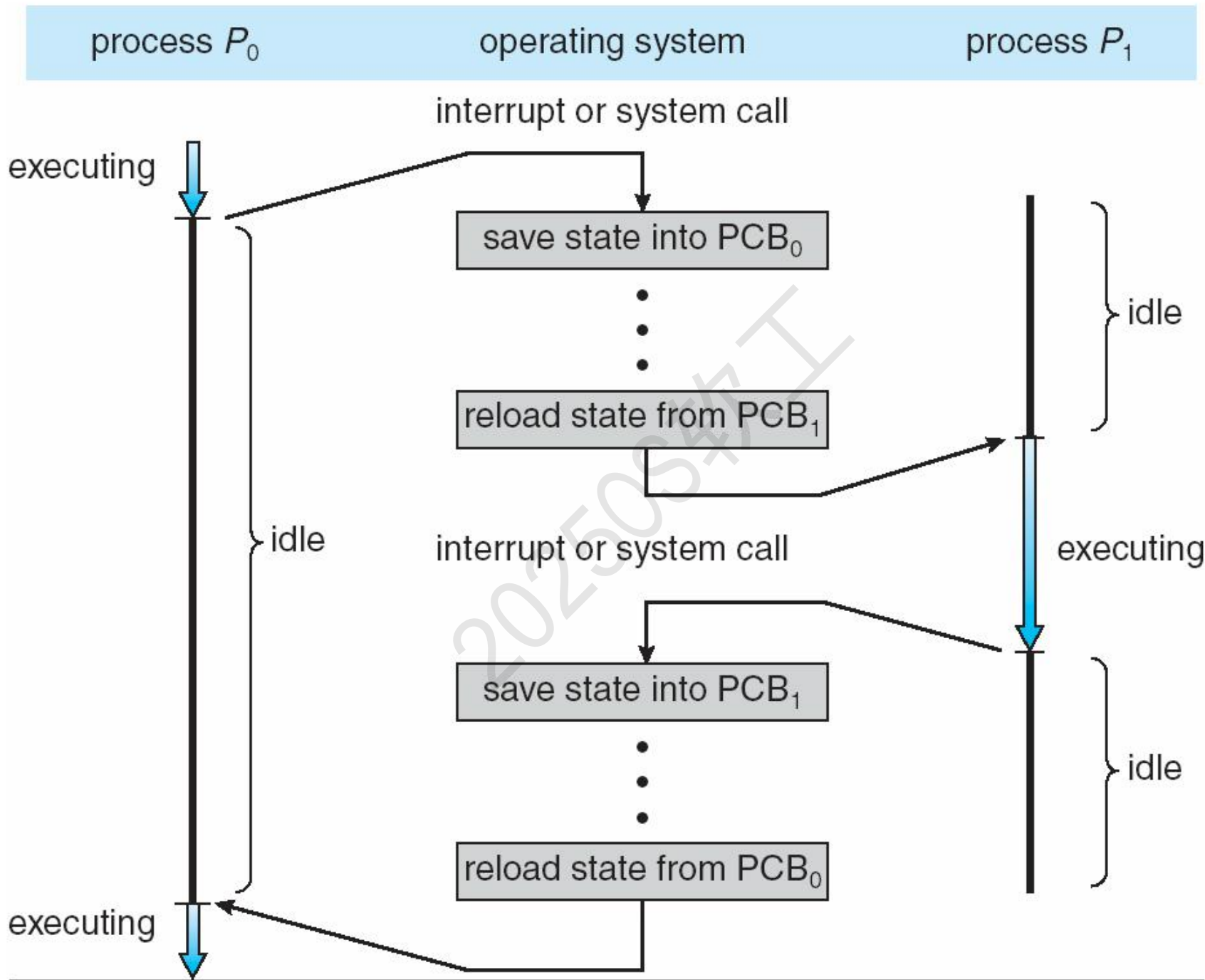
## task\_struct

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



# 上下文切换 (Context Switch)

- 当CPU切换到另一进程时，原先进程的状态应保存，然后载入新进程已保存的状态，此过程称为上下文切换
- 进程的上下文保存在PCB中
- 上下文切换的时间为额外开销；系统在此过程中只是进行了管理，没有执行实际任务
  - OS 和 PCB越复杂，上下文切换的时间越长



# xv6中的进程切换

AT&T语法

mov src, dst

注意:

- 切换的是进程的**内核态**上下文
- 用户态上下文在内核态上下文中隐含
- **eip**怎么没保存?

struct context

```
{  
    uint edi;  
    uint esi;  
    uint ebx;  
    uint ebp;  
    uint eip;  
};
```

```
1  # Context switch  
2  #  
3  # void swtch(struct context **old, struct context *new);  
4  #  
5  # Save the current registers on the stack, creating  
6  # a struct context, and save its address in *old.  
7  # Switch stacks to new and pop previously-saved registers.  
8  
9  .globl swtch  
10 swtch:  
11     movl 4(%esp), %eax  
12     movl 8(%esp), %edx  
13  
14     # Save old callee-save registers  
15     pushl %ebp  
16     pushl %ebx  
17     pushl %esi  
18     pushl %edi  
19  
20     # Switch stacks  
21     movl %esp, (%eax)  
22     movl %edx, %esp  
23  
24     # Load new callee-save registers  
25     popl %edi  
26     popl %esi  
27     popl %ebx  
28     popl %ebp  
29     ret  
30
```

保存旧的上下文到**内核内存**

更换栈

从内存弹出新的上下文到寄存器中

# 进程概念

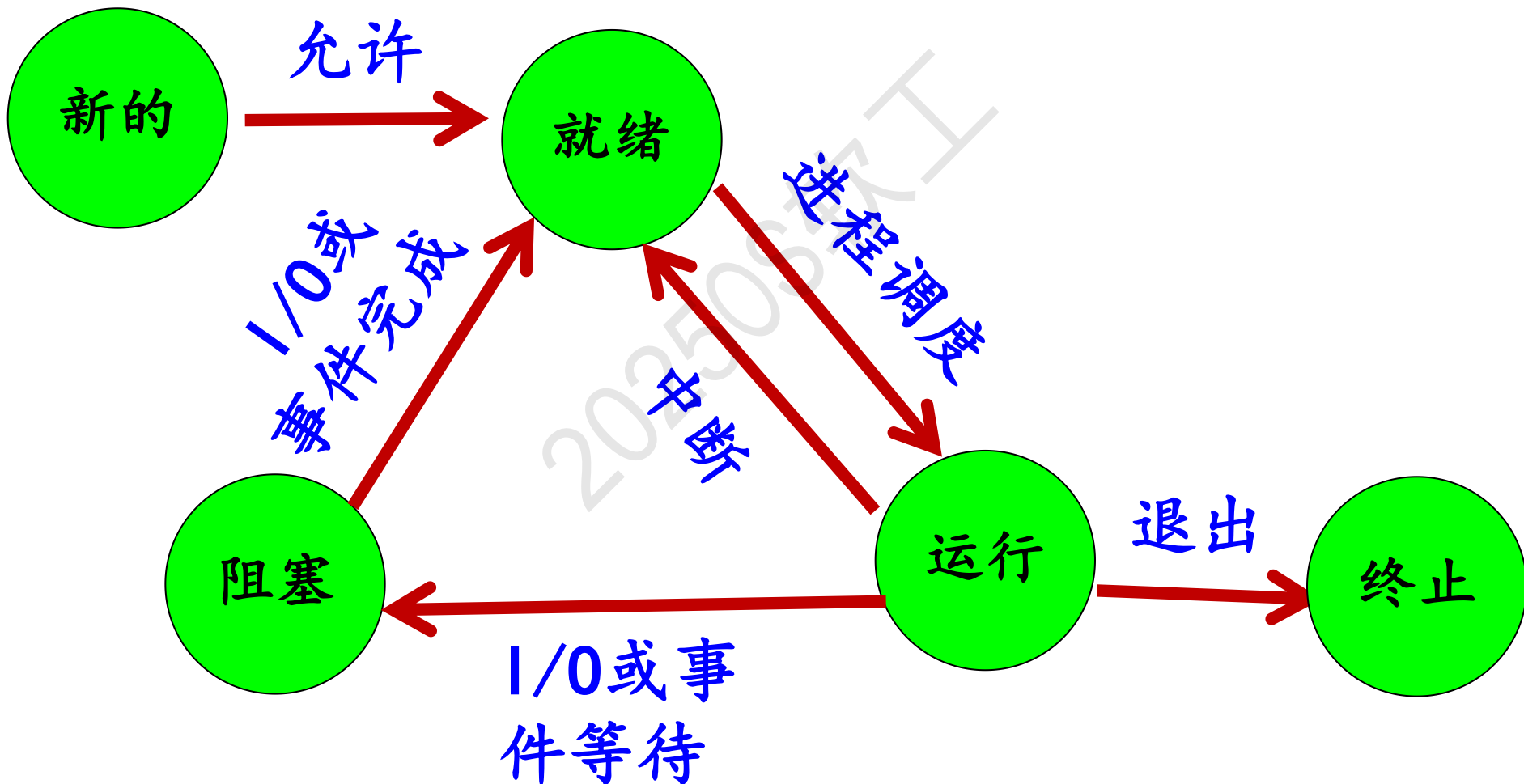
- 进程的内存抽象
- 进程的状态
- 进程控制块与进程切换
- 进程创建与终止

# 进程创建

- 父进程可创建子进程，子进程可以继续创建子进程，构成进程树
  - Linux下的pstree命令
- 如何设计：父进程与子进程的执行顺序
  - 并发执行
  - 父进程优先或子进程优先
  - 常见的编程模式：等待子进程结束，然后执行
- 父进程与子进程的地址空间
  - 子进程复制父进程地址空间（unix）[危险!]
  - 子进程装入新程序（windows）

# 提问环节

- 进程的状态转移中，哪个转移一般是“被迫”发生的？





# 课后作业一反馈

- 计算题要写过程
- 第1题，变量d在栈区，\*d在堆区
- 把 (int \*) 读做“整数地址”变量

```
int a;  
void b(){  
    char c;  
}  
void main(){  
    int * d=malloc(sizeof(int));  
}
```

# Unix下创建新进程

- fork( )释义与演示

```
#include "unistd.h"
```

```
void main(){
```

```
    int i=1;
```

```
    int pid=2;
```

```
    pid=fork();
```

```
    i=3;
```

```
}
```

000011cd <main>:

11cd: f3 0f 1e fb

11d1: 55

11d2: 89 e5

11d4: 53

11d5: 83 e4 f0

11d8: 83 ec 10

11db: e8 2e 00 00 00

11e0: 05 f8 2d 00 00

11e5: c7 44 24 08 01 00 00

11ec: 00

11ed: c7 44 24 0c 02 00 00

11f4: 00

11f5: 89 c3

11f7: e8 84 fe ff ff

11fc: 89 44 24 0c

1200: c7 44 24 08 03 00 00

1207: 00

1208: 90

1209: 8b 5d fc

120c: c9

120d: c3

endbr32

push %ebp

mov %esp,%ebp

push %ebx

and \$0xffffffff,%esp

sub \$0x10,%esp

call 120e <\_\_x86.get\_pc\_thunk.a

add \$0x2df8,%eax

movl \$0x10x8(%esp)

*i*

movl \$0x20xc(%esp)

*pid*

mov %eax,%ebx

call 1080 <fork@plt>

mov %eax,0xc(%esp)

*pid*

movl \$0x3,0x8(%esp)

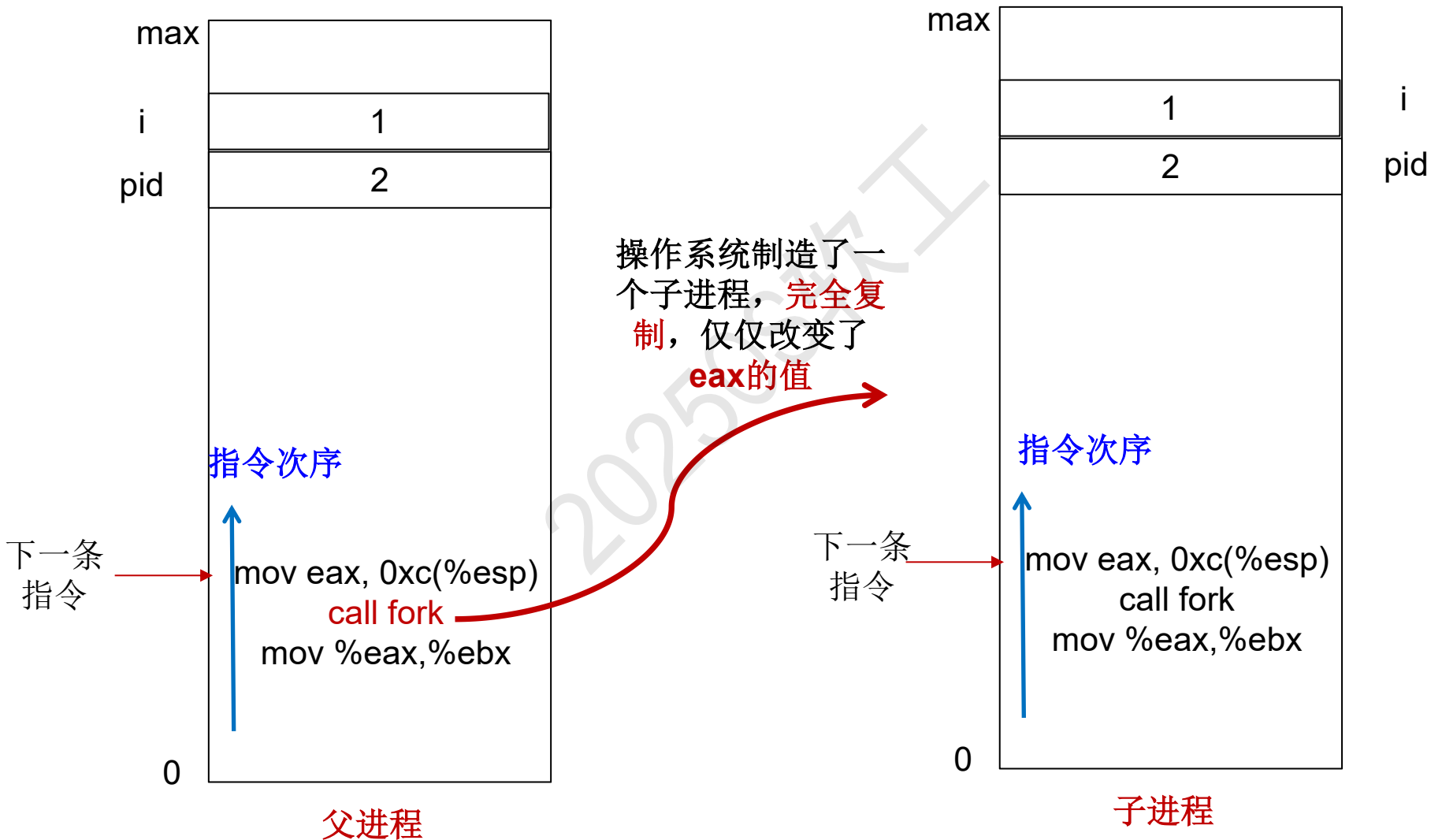
nop

mov -0x4(%ebp),%ebx

leave

ret

# Unix下创建新进程



fork出子进程后，二者仅能通过返回值判断。

# xv6中fork的实现

```
int
fork(void)
{
    int i, pid;
    struct proc *np;
    struct proc *curproc = myproc();

    // Allocate process.
    if((np = allocproc()) == 0){
        return -1;
    }

    // Copy process state from proc.
    if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1;
    }
    np->sz = curproc->sz;
    np->parent = curproc;
    *np->tf = *curproc->tf;

    // Clear %eax so that fork returns 0 in the child.
    np->tf->eax = 0;

    for(i = 0; i < NOFILE; i++)
        if(curproc->ofile[i])
            np->ofile[i] = filedup(curproc->ofile[i]);
    np->cwd = idup(curproc->cwd);

    safestrcpy(np->name, curproc->name, sizeof(curproc->name));

    pid = np->pid;

    acquire(&ptable.lock);

    np->state = RUNNABLE;

    release(&ptable.lock);

    return pid;
}
```

用户地址空间复制

内核栈复制

改子进程返回值

父进程返回值

# Unix下创建新进程

- 那么，以下程序输出是？

```
#include "unistd.h"
```

```
fork();
```

```
printf("a\n");
```

```
fork();
```

```
printf("b\n");
```

```
fork();
```

```
printf("c\n");
```

20250505 软工

# fork bomb!

---

```
while(1) fork();
```

- 服务器会因何崩溃?
- 现代操作系统已做了一定防御
- 试试看

# exec

- 要运行新程序怎么办？exec！
- 读取文件头
  - 对文件头进行检查，确保其格式是正确的。
  - 为需要load的段分配虚拟地址空间。
  - 从磁盘中读相应的段到内存中。
- 准备用户栈，配置参数。
  - 注意假的返回地址
- 设置上下文，让进程“醒来”时，程序计数器（PC）指向新程序的入口点。
- 一起读源码



# shell的核心: fork + exec

c

sh.c

×

```
int
main(void)
{
    // Read and run input commands.
    while(getcmd(buf, sizeof(buf)) >= 0){
        if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
            // Chdir must be called by the parent, not the child.
            buf[strlen(buf)-1] = 0; // chop \n
            if(chdir(buf+3) < 0)
                printf(2, "cannot cd %s\n", buf+3);
            continue;
        }
        if(fork1() == 0)
            runcmd(parsecmd(buf));
        wait();
    }
}
```

处理内置命令

处理外部命令

等待子进程结束

```
// Execute cmd. Never returns.
```

```
void
```

```
runcmd(struct cmd *cmd)
```

```
{
```

```
case EXEC:
```

```
    ecmd = (struct execcmd*)cmd;
```

```
    if(ecmd->argv[0] == 0)
```

```
        exit();
```

```
    exec(ecmd->argv[0], ecmd->argv);
```

```
    printf(2, "exec %s failed\n", ecmd->argv[0]);
```

```
    break;
```

# 提问环节

- 进程运行后，argc和argv是谁赋值给它的？

← → ↺ 🏠 🔍 gnu.org/software/libc/manual/html\_node/Program-Argum... ☆ 📷 📁 | ⬇️ 🌐 ⋮

Next: [Environment Variables](#), Up: [The Basic Program/System Interface](#) [[Contents](#)][[Index](#)]

## 25.1 Program Arguments

The system starts a C program by calling the function `main`. It is up to you to write a function named `main`—otherwise, you won't even be able to link your program without errors.

In ISO C you can define `main` either to take no arguments, or to take two arguments that represent the **command line arguments** to the program, like this:

```
int main (int argc, char *argv[])
```

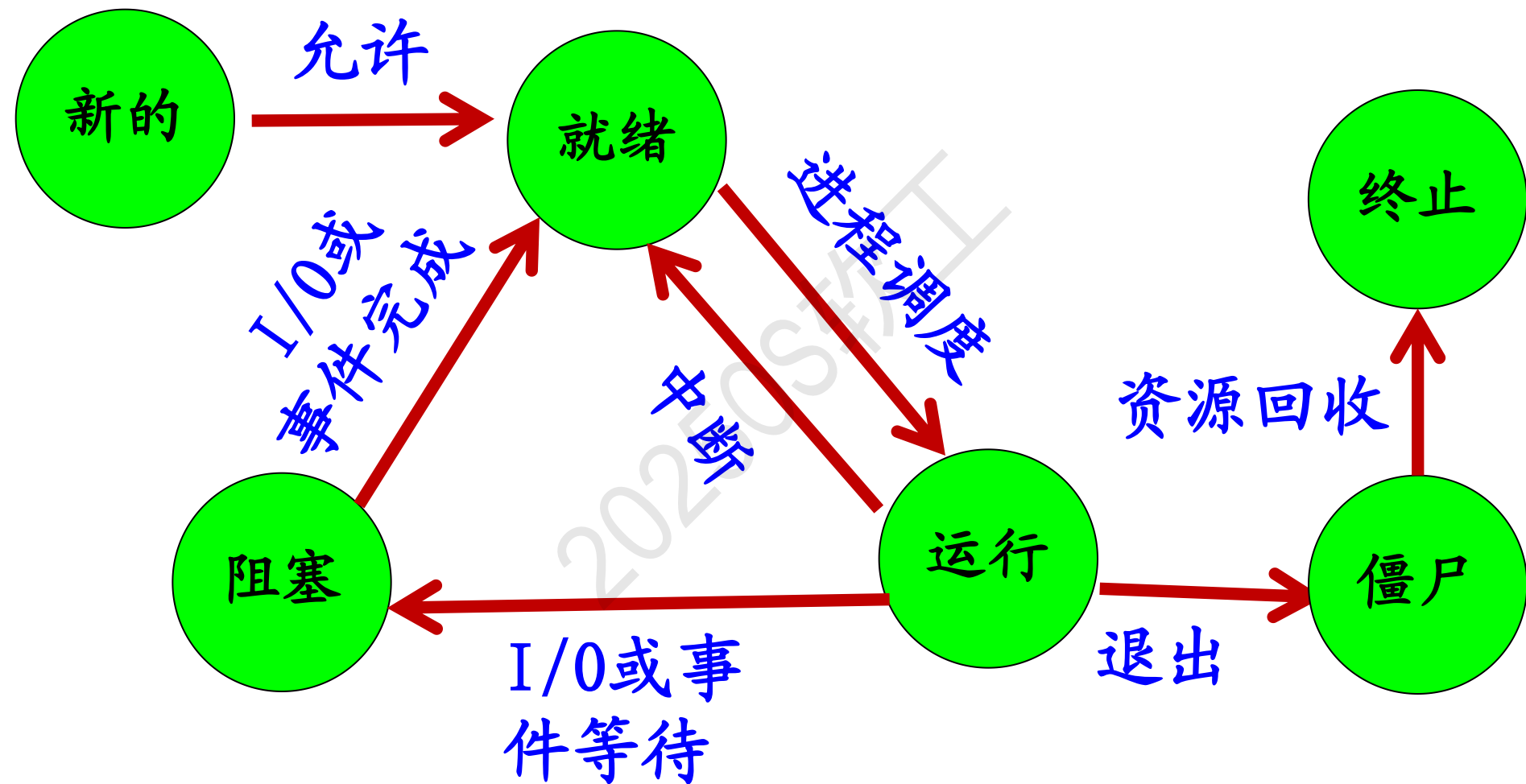
命令行参数

The command line arguments are the whitespace-separated tokens given in the shell command used to invoke the program; thus, in `'cat foo bar'`, the arguments are `'foo'` and `'bar'`. The only way a program can look at its command line arguments is via the arguments of `main`. If `main` doesn't take arguments, then you cannot get at the command line.

# 进程的终止

- 通过系统调用 **exit()** 请求删除自身
  - 返回状态数据给父进程 (父进程通过**wait()**获取)
  - 进程资源被操作系统释放 (何时释放? 在**wait**中)
- 父进程也可强制终止子进程, 如通过 **abort()** 系统调用, 或者**TerminateProcess()**
- 僵尸进程、孤儿进程
  - **僵尸状态**: 已经调用**exit**, 但父进程尚未**wait**
    - 僵尸进程, **zombie process**
  - 孤儿进程 (**短暂存在**)
    - 父进程已结束的进程, **orphaned process**
    - **Linux**会自动接管孤儿进程, 将其父进程设置为**init** (**systemd**) 进程
  - 僵尸孤儿进程 (**短暂存在**), **orphaned zombies**

# 僵尸状态插在哪个位置？



# 案例：xv6中proc.c中的exit和wait

```
226 // until its parent calls wait() to find out it exited.
227 void
228 exit(void)
229 {
230     struct proc *curproc = myproc();
231     struct proc *p;
232     int fd;
233
234     if(curproc == initproc)
235         panic("init exiting");
236
237     // Close all open files.
238     for(fd = 0; fd < NOFILE; fd++){
239         if(curproc->ofile[fd]){
240             fileclose(curproc->ofile[fd]);
241             curproc->ofile[fd] = 0;
242         }
243     }
244
245     begin_op();
246     iput(curproc->cwd);
247     end_op();
248     curproc->cwd = 0;
249
250     acquire(&table.lock);
251
252     // Parent might be sleeping in wait().
253     wakeup1(curproc->parent);
254
255     // Pass abandoned children to init.
256     for(p = table.proc; p < &table.proc[NPROC]; p++){
257         if(p->parent == curproc){
258             p->parent = initproc;
259             if(p->state == ZOMBIE)
260                 wakeup1(initproc);
261         }
262     }
263
264     // Jump into the scheduler, never to return.
265     curproc->state = ZOMBIE;
266     sched();
267     panic("zombie exit");
268 }
```

exit中，将进程状态改为ZOMBIE

```
int
wait(void)
{
    struct proc *p;
    int havekids, pid;
    struct proc *curproc = myproc();

    acquire(&table.lock);
    for(;;){
        // Scan through table looking for exited children.
        havekids = 0;
        for(p = table.proc; p < &table.proc[NPROC]; p++){
            if(p->parent != curproc)
                continue;
            havekids = 1;
            if(p->state == ZOMBIE){
                // Found one.
                pid = p->pid;
                kfree(p->kstack);
                p->kstack = 0;
                freevm(p->pgdir);
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 0;
                p->killed = 0;
                p->state = UNUSED;
                release(&table.lock);
                return pid;
            }
        }
    }
}
```

wait中，查找 ZOMBIE 子进程



# 那么，如何删除僵尸进程？

- 在Unix下，孤儿进程会被init/systemd进程设置为子进程
- 因此，看到僵尸进程时
  - 僵尸进程的父进程未结束
  - 可以通过杀死父进程来清除僵尸进程

```
1 #include "unistd.h"
2 #include "stdio.h"
3
4 void main(){
5     int pid;
6     pid=fork();
7     if(pid==0){
8         //child,
9         printf("I'm the child, I am exiting\n");
10    }else{
11        //parent
12        printf("I'm the parent, I am running a dead loop\n");
13        while(1);
14        wait();
15    }
16 }
```

运行此程序，  
会产生一个  
僵尸进程

```
ubuntu:~$ ps aux | grep a.out
99.8  0.0  3576   616 pts/0    R+   21:57   7:01 ./a.out
0.0   0.0    0     0 pts/0    Z+   21:57   0:00 [a.out] <defunct>
```

命令行观测  
到僵尸进程

```

→ cache git:(master) x ps aux | grep python3
root      6846 29.5  0.0      0      0 pts/1    Z+   17:49   1:05 [python3] <defunct>
root      7084 61.1  4.1 1509672 78008 pts/1    Dl+  17:50   1:26 python3 /usr/local/python/CV/SR/real-esr
gan-show.py -m /usr/local/python/CV/SR/model.pth -d cpu -i ./data/cache/f73b3527-eceb-4e59-8341-1d1fca046
c36.png -o ./data/cache/f73b3527-eceb-4e59-8341-1d1fca046c36.out.png -t ./data/cache/f73b3527-eceb-4e59-8
341-1d1fca046c36
root      7505  0.0  0.0 112812   992 pts/3    S+   17:53   0:00 grep --color=auto --exclude-dir=.bzip --e
xclude-dir=CVS --exclude-dir=.git --exclude-dir=.hg --exclude-dir=.svn --exclude-dir=.idea --exclude-dir=
.tox python3
→ cache git:(master) x ps aux | grep python3
root      6846 10.2  0.0      0      0 pts/1    Z+   17:49   1:05 [python3] <defunct>
root      7084 15.6  0.0      0      0 pts/1    Z+   17:50   1:27 [python3] <defunct>
root      7531 16.4  0.0      0      0 pts/1    Z+   17:53   1:06 [python3] <defunct>
root      8483  0.0  0.0 112812   992 pts/3    S+   18:00   0:00 grep --co=auto --exclude-dir=.bzip --e
xclude-dir=CVS --exclude-dir=.git --exclude-dir=.hg --exclude-dir=.svn --clude-dir=.idea --exclude-dir=
.tox python3
→ cache git:(master) x █

```

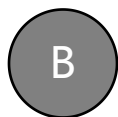
僵尸进程



讨论：操作系统是进程吗？



是



不是

202505软工

提交

# 本部分内容的要求（2025考研大纲）

- 程序运行时的内存映像与地址空间
- 进程的基本概念
- 进程的状态与转换
- 进程的上下文及其切换机制