

操作系统

概论

冯·诺依曼体系结构特点：可编程、计算和存储分离

x86架构基础知识

函数调用的操作

函数返回

操作系统

内容大纲：

操作系统的特征：

分类：

引导

计算机启动过程

Boot程序v2.0

loader程序

检测硬件信息

处理器模式切换

向内核传递数据

开启A20功能

构建系统数据结构

中断和异常

段页内存管理

多任务机制

进程

进程的定义

系统调用

从一个进程到多个进程

进程控制块PCB(process control block)

进程的生命周期

父进程和子进程

僵尸进程的销毁

进程的创建

进程的管理

进程间切换

上下文切换

！！引入进程的目的

！！引入线程的目的

进程和线程的关系

进程间通信

线程的生命周期

线程的类型

！！理解用户态和系统态

进程调度

资源类型

操作系统中的资源调度

类型分类：

CPU资源调度指标

CPU资源调度策略

先来先服务策略(FIFO/FICS)

最短任务优先策略（Shortest Job First,SJF）

轮询策略（Round Robin， RR）

CPU资源调度（基于优先级的策略）

优先级调度（Priority Scheduling）

多级队列调度（Multi-level Queue Scheduling）

多级反馈队列调度（Multi-level Feedback Queue Scheduling）

- 实时任务调度
- 进程和线程相关数据结构
- 进程和线程调度方式
 - linux下多线程的代码
 - 关于函数调用
 - 系统调用
- 内存管理
 - 内存管理的目标
 - 虚拟内存到物理内存的映射
 - 分段映射机制
 - 分页映射机制
 - 段页结合映射机制
 - 反向页表映射机制
 - 进程创建与写入时复制
 - linux进程的内存管理
 - Linux进程内核代码
 - 内核态
 - 缺页操作
 - 页换入操作的具体步骤
 - 页换出操作
 - 页面替换算法
 - Bleady 异常
- 输入输出设备
 - 多种不同I/O模式
 - DMA:直接内存存取技术
 - 用驱动程序屏蔽设备控制器差异
 - 用文件系统屏蔽驱动程序差异
 - 补充: linux文件权限
 - linux添加设备流程
- I/O设备中断处理
 - 中断向量
 - 设备中断向量处理
- 文件系统
 - 磁盘相关知识回顾
 - 文件系统建立
 - 建立目的
 - 相应要求
 - 核心组件: 文件+ 目录
 - 文件
 - 目录
 - 文件和卷的关系
 - 对文件的基本操作
 - 代码举例
 - 设计磁盘文件系统的挑战
 - 磁盘文件系统
 - 建立索引 : 一般按照树形结构
 - 空闲地址映射 : 快速分配机制 -> bitmap实现
 - 局部性原理 : 相同文件的靠近内容防止在相近磁盘块
 - 分配方式讨论
 - 1. 连续分配方式
 - 2. 链表方式
 - 3. FAT文件系统
 - FAT优缺点
 - FFS文件系统 (Fast File System)
 - NTFS(ext4): 引入新概念Extents
 - 硬链接和符号链接的存储格式

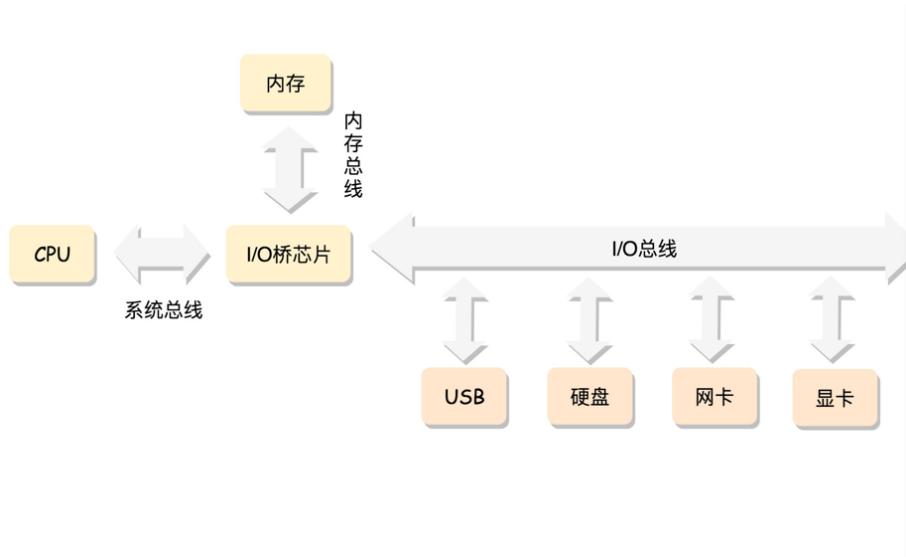
- 硬链接
- 符号链接
- 虚拟文件系统
- 进程间通信
 - 进程通信的需求
 - 数据传输
 - 事件通知
 - 资源共享
 - 进程控制
 - 进程如何通信
 - 管道模型
 - 匿名管道:
 - 命名管道
 - 消息队列模型
 - 信号
 - 共享内存模型（同步和互斥）
 - 共享内存模式（信号量）！！！
 - socket套接字**（请查看计网）
- 总结:

- 计算机网络系统
- Socket 编程
- bind**函数(服务器端)
- listen**函数(服务器端)
 - 第一次握手
 - 第二次握手
 - 第三次握手
 - 总结
- write**函数
- read**函数
- 同步互斥与信号量
- 临界区
 - 临界区的访问规则
 - 临界区的实现方法
 - 禁用中断
 - 软件方法
 - 更高级的抽象方法
- 锁
- 信号量
- 信号量概念(**semaphore**)
 - P操作
 - V操作
- 管程
- 管程的使用
 - 条件变量
 - Wait操作
 - Signal操作
- 管程条件变量的释放处理方式
 - Hansen管程
 - Hoare管程
- 哲学家就餐问题
 - 不可行方案
 - 改进方案
- 读者写者问题

操作系统

概论

冯·诺依曼体系结构示意图



分为南北桥，北桥被用来处理高速信号，处理CPU，内存和南桥芯片的通信。而南桥被用来处理低速信号，负责I/O总线之间的通信

冯·诺依曼体系结构特点：可编程、计算和存储分离

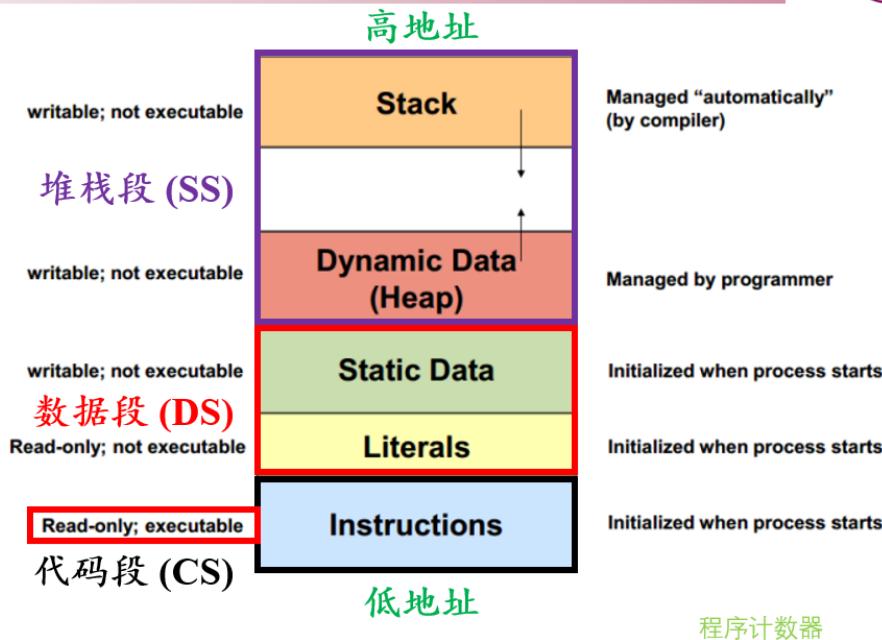
1. 采用指令和数据存储在一起的结构
2. 存储器是按地址访问、线性编址的空间
3. 指令由操作码和地址码组成
4. 指令在CPU的运算单元处理
5. 控制流由指令流产生

x86架构基础知识

ALU数据总线 (16位)	
AX——累加器,	使用频度最高
BX——基址寄存器,	常存放存储器地址
CX——计数器,	常作为计数器
DX——数据寄存器,	存放数据
SI ——源变址寄存器,	常保存存储单元地址
DI ——目的变址寄存器,	常保存存储单元地址
BP——基址指针寄存器,	表示堆栈区域中的基地址
SP——堆栈指针寄存器,	指示堆栈区域的栈顶地址
IP——指令指针寄存器,	指示要执行指令所在存储单元的地址

对于一个程序而言，一般分为代码段，数据段和堆栈段，其具体结构为：

内存布局



函数调用的操作

- 1) **参数入栈:** 将参数从右向左依次压入系统栈中
- 2) **返回地址入栈:** 将当前代码区调用指令的下一条指令地址压入栈中, 供函数返回时继续执行
- 3) **代码区跳转:** 处理器从当前代码区跳转到被调用函数的入口处
- 4) **栈帧调整:**
 - 保存当前栈帧状态值, 已备后面恢复本栈帧时使用(BP入栈)
 - 将当前栈帧切换到新栈帧 (将SP值装入BP, 更新栈帧底部)
 - 给新栈帧分配空间 (把SP减去所需空间的大小, 抬高栈顶)

函数返回

- 1) **保存返回值,** 将函数的返回值保存在寄存器AX
- 2) **弹出当前帧, 恢复上一个栈帧**
 - 给SP加上栈帧的大小, 降低栈顶, 回收当前栈帧的空间
 - 将当前栈帧底部保存的前栈帧BP值弹入BP寄存器, 恢复出上一个栈帧
 - 将函数返回地址弹给IP寄存器
- 3) **跳转:** 按照CS:IP跳回调用函数中继续执行

操作系统

操作系统是一个系统控制程序，一个资源管理器

内容大纲：

- 内存管理
 - ✓ 连续与非连续内存分配管理
 - ✓ 分页、分段、段页式管理方式的演化过程
 - ✓ 虚拟内存概念，请求分页管理方式
 - ✓ 页面分配和替换算法
- 文件系统管理
 - ✓ 文件的概念和逻辑结构
 - ✓ 目录结构，包括多级、树形、图形结构
 - ✓ 文件共享和访问控制
 - ✓ 文件系统实现
 - ✓ 磁盘组织与管理
- I/O管理
 - ✓ I/O管理概述
 - ✓ I/O控制方式
 - ✓ I/O核心子系统
 - ✓ 设备分配与回收
- 额外的知识
 - ✓ 网络系统
 - ✓ 虚拟化
 - ✓ 容器

操作系统的特征：

1. 并发：同时存在多个运行的程序（进程），需要OS管理和调度
2. 共享：互斥共享，某种资源只能同时被一个进程独享；同时访问：允许在一段时间内由多个进程交替访问
3. 虚拟：利用多道程序设计技术，让每个进程和每个用户觉得有一个处理器(CPU)专门为他服务。
4. 异步：进程执行不是一貫到底，而是走走停停且执行速度不可预知。

分类：

操作系统的分类：

➤ 单道操作系统

- 多个进程按照先来先服务的方式顺序执行(排队系统)
- 内存中只有一个进程运行且它需要执行完毕(独占内存)
- ✗ 在运行时, 高速CPU会等待低速I/O完成, 限制系统吞吐量

➤ 多道操作系统

- 内存中同时存放多个相互独立的进程
- **宏观上并行(先后开始各自的运行, 但都未运行完毕)**
- **微观上串行(轮流占有CPU)**
- ✓ 资源利用率和系统吞吐量均增大
- ✗ CPU、内存、I/O设备资源分配问题复杂
- ✗ 大量进程和数据组织和存放的安全性和一致性问题

➤ 分时操作系统

- 分时技术: 处理器的运行时间分成很短的时间片
- 按时间片轮流把处理器分配给各个用户或进程使用
- ✗ 存在系统延时(银行系统、购票系统等)

➤ 实时操作系统

- 计算机系统接收到程序请求后会立即处理, 并在严格的时限内处理完成该程序
- ✗ 可靠性和实时性保证较难
- ✗ 一般需要专门设计, 普适性较差

引导

内存为什么要分段?

- 便于迅速寻址访问
- 8086体系中使用CS:IP进行定位 (CS<<4 + IP)
- 80386中使用ECS:EIP进行定位

计算机启动过程

开机运行的第一个程序为: BIOS(base input & output system)

对其, 需要解答这样几个问题:

- 它是由谁加载到内存的?
- 它被加载到内存哪里?
- CS:IP如何定位的?
- 它如何体现“基本”?

同时, 需要对操作系统的保护模式和实模式进行理解:

• 实模式

- ✓ 32位或64位CPU在8086体系下的工作环境
- ✓ 20根地址总线 (寻址1M, 0x00000 ~ 0xFFFFF)
- ✓ 该1M寻址空间由多个部分组成

实模式下只有1MB的寻址空间，其寻址方式为之前8086定义的那样，可以看到实模式下的内存布局为：

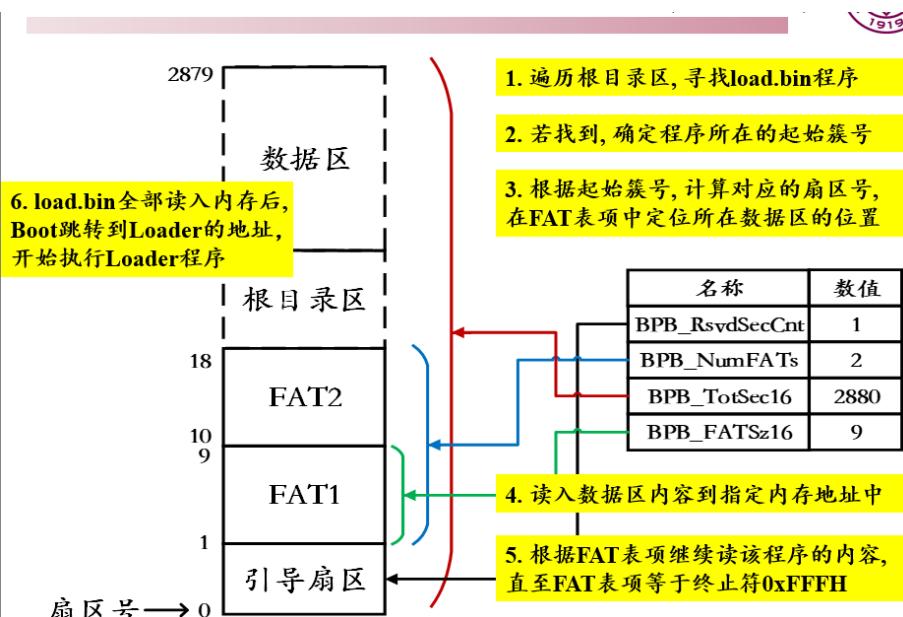
实模式下的内存布局



起始	结束	大小	用途
FFFF0	FFFFF	16B	BIOS 入口地址，此地址也属于 BIOS 代码，同样属于顶部的 640KB 字节。只是为了强调其入口地址才单独贴出来的。此处 16 字节的内容是跳转指令: jmp f000: e05b 为什么是“基本”？
F0000	FFFEF	64KB-16B	系统 BIOS 范围是 F0000~FFFFF 共 64B，为说明入口地址，将最上面的 16 字节从此处去掉了，所以此处终止地址是 0XFFFFF
C8000	EFFFF	160KB	映射硬件适配器的 ROM 或内存映射式 I/O
C0000	C7FFF	32KB	显示适配器 BIOS
B8000	BFFFF	32KB	用于文本模式显示适配器
B0000	B7FFF	32KB	用于黑白显示适配器
A0000	AFFFF	64KB	用于彩色显示适配器
9FC00	9FFFF	1KB	EBDA (Extended BIOS Data Area) 扩展 BIOS 数据区
7E00	9FBFF	622080B 约 608KB	可用区域 为什么是动态？
7C00	7DFF	512B	MBR 被 BIOS 加载到此处，共 512 字节
500	7BFF	30464B 约 30KB	可用区域 动态随机访问内存 (DRAM) 即内存条 640KB
400	4FF	256B	BIOS Data Area (BIOS 数据区)
000	3FF	1KB	Interrupt Vector Table (中断向量表)

可以看到，实际上BIOS代码只有64KB，而实际上其起始地址为0xFFFF0，故一开始的时候，会跳转到此位置。开始时，CPU的CS:IP寄存器被强制初始化为：0xF000:0xFFFF0，BIOS会调用中断0x19来检测计算机中断和软盘数，如果存在可用的磁盘，BIOS把它0盘0道1扇区的内容加载到0x7C00(一个扇区大小为512B，而刚好相应的位置也为512B)

Boot程序v2.0



loader程序

检测硬件信息

OS内核程序需要硬件信息，但OS内核运行在保护模式下，loader需要调用BIOS中断来获取上述信息，解析物理地址信息

处理器模式切换

BIOS运行的实模式 转换为 32位OS使用的保护模式 再转换为 64位操作系统使用的IA-32e模式。

向内核传递数据

控制信息：控制内核程序的执行流程

硬件信息：为内核程序的初始化提供支持

开启A20功能

构建系统数据结构

代码段全局描述符表

数据段全局描述符表

局部描述符表

中断和异常

中断描述表

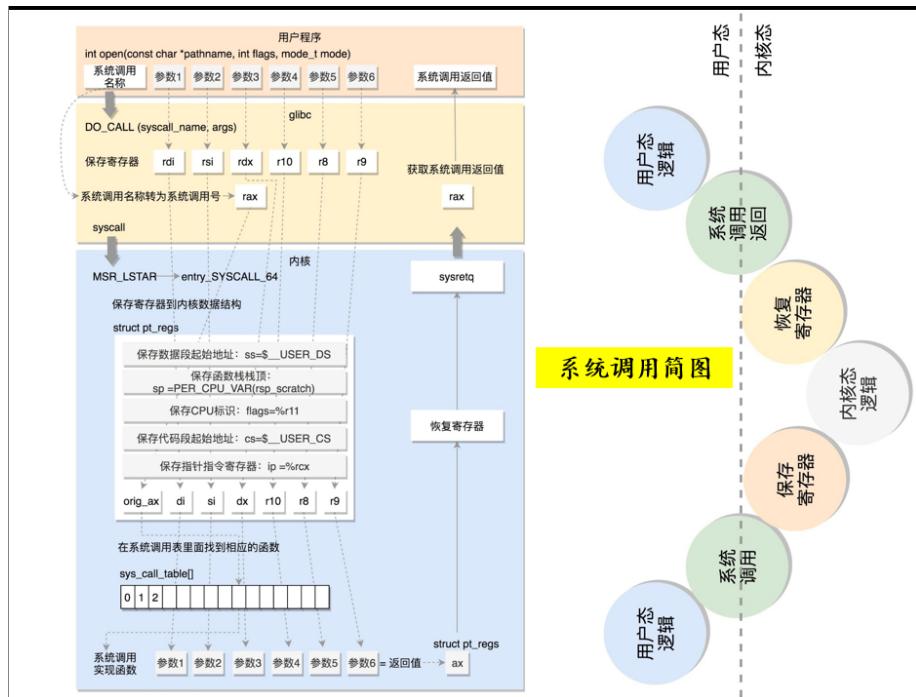
段页内存管理

多任务机制

- 内核程序入口 **start_kernel()**

1. **INIT_TASK()**: 0号进程 (**系统进程, 进程列表首位**)
2. **trap_init()**: 设置中断门，用于处理各种中断 (e.g., 系统调用)
3. **mm_init()**: 初始化内存管理模块
4. **sched_init()**: 初始化进程调度策略模块
5. **vfs_caches_init()**: 虚拟文件系统初始化
6. **kernel_thread()**: 1号进程 (**第一个用户进程, 管理其他用户进程**)
7. **kernel_thread()**: 2号进程 (**管理系统进程**)
8. 系统启动完成，等待用户创建新的进程

内核调用的系统简图如下



重点！！！！！

- 计算机系统启动流程
 - BIOS: 硬件自检, 中断向量表
 - boot程序: 0盘0道1扇区, 寄存器初始化, FAT文件系统
 - loader程序: 实模式 → 保护模式, 段页管理启动...
 - 操作系统内核程序: 初始化内存、调度、虚拟文件系统管理模块, 创建用户态和内核态管理进程(1号, 2号进程)

- 系统调用过程图
 - 保存和恢复用户态寄存器 (CS, IP, SS, SP, CPU标识位)
 - AX寄存器保存返回参数

进程

程序和进程的关系: 程序实际上是一个文件, 是用于创建进程的一个数据与指令的集合, 其是由某种程序语言来刻画的, 将程序编译形成可执行文件, 然后CPU会根据此进行调度执行。当运行一个程序时, 实际上是创建了一个进程, 但是同一程序的多个运行对应了多个进程。

进程的定义

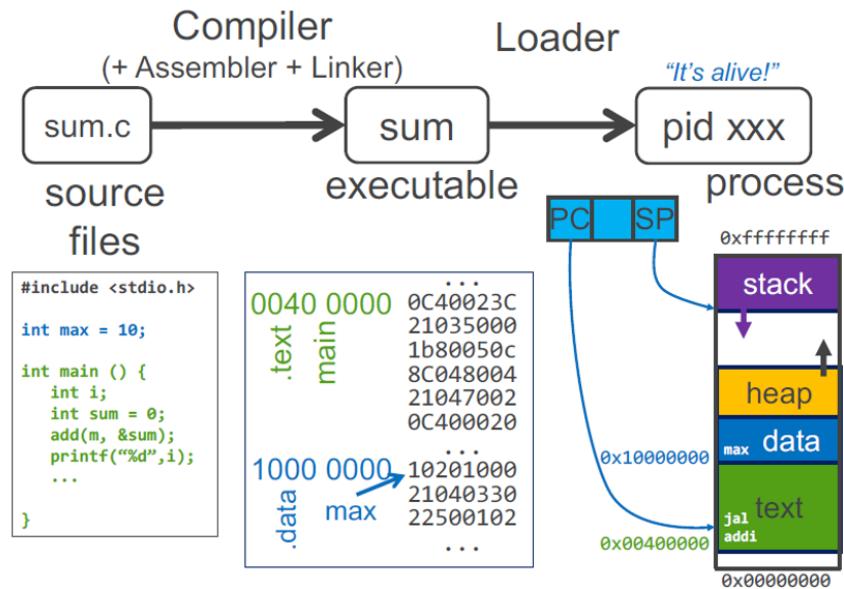
操作系统为正在运行的程序提供的抽象：具体抽象为 地址空间 + 执行上下文 + 环境，而对于一个好的程序抽象而言，其具体有：

- 轻量级且隐藏了具体实现
- 易于使用的接口
- 可以被实例化多次
- 易于实现

从程序的过程的具体过程:



程序 (Passive) → 进程 (Alive)



对于一个进程的内存布局: 其可以参考上图, 主要有这样几个段;

代码段, 数据段和堆栈段, 而可以知道, 进程是内存分配的最小单元。

进程的动作:

- 从屏幕读入、写出数据
- 创建、读取、写入、删除文件
- 创建新的进程
- 发送、接收网络数据包
- 获取系统时间、调整CPU频率
- 终止当前进程
-

进程如何实现这些动作?

进程实现这个动作: 需要系统调用, 因为这设计到了相应的硬件操作。

系统调用



由于硬件资源有限，对于系统内部硬件资源的操作，只能由操作系统实现，此时使用操作系统直接对硬件进行操作，只能是小细腰的结构。

系统调用

在进行系统调用的时候，实际上会从用户态陷入到内核态，其陷入过程可以理解为如下：

首先保存相应产生系统调用的参数和相关内容，然后通过call陷入内核，在内核态进行相应的处理，在内核态处理完之后，弹出相应产生系统调用的位置，然后从之前的位置开始继续执行。

从一个进程到多个进程

从一个进程到多个进程



- 每个进程都需要在CPU上运行
- 不同进程具有各自所需的资源
 - 寄存器
 - 内存
 - I/O资源
 - 线程
- 一般来说，进程数目要多于CPU核数
 - 复用 (Multiplexing)
 - 资源调度 (Scheduling)

如何区分不同的进程？

对于每个进程而言，其有对应的pid，根据其pid，可以对其进行区分。

进程控制块PCB(process control block)

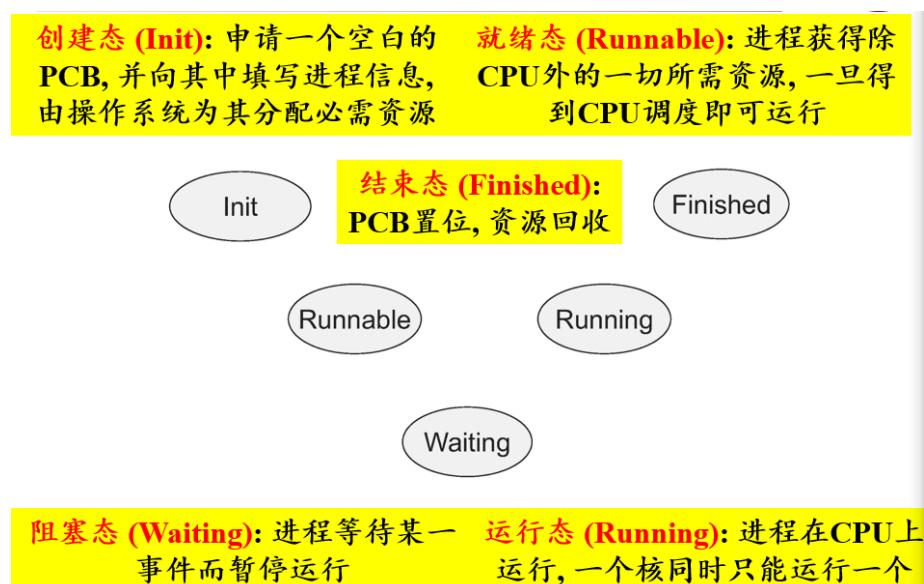
对于任意一个进程，操作系统会维护一个进程控制块来描述进程的基本信息和运行状态，进而控制和管理进程

PCB应该包含的内容：

- 进程在内存中的位置 (page table)
- 对应的可执行文件在磁盘的位置
- 哪个用户执行这个进程 (uid)
- 进程ID号 (pid)
- **进程运行状态 (运行态、等待态...)**
- 调度信息、保存的内核SP、中断栈信息...

PCB是进程存在的唯一标志

进程的生命周期



父进程和子进程

操作系统允许一个进程创建另一个进程（父→子）
子进程可以继承父进程拥有的资源
当子进程被销毁时，会归还从父进程获得的资源，
但并非马上就消失掉，而是留下一个称为僵尸进程
(Zombie) 的数据结构 (很小的内存)，等待父进程处理

僵尸进程的销毁

- ✓ 子进程的结束和父进程的运行是一个异步过程，即父进程永远无法预测子进程到底什么时候结束
- ✓ 子进程结束后，应通知父进程需要调用**wait()**系统调用取得子进程的终止状态，销毁其对应的僵尸进程
- ✗ 由于编程错误或者异常（子进程被kill）→ 僵尸进程
- ✗ 若父进程比子进程更早的结束，则由init进程（1号进程）接管

为什么要设计僵尸进程？它有危害吗？

对于一个僵尸进程而言，其主要产生的原因是为了保存一些相应的内容。

由于子进程的结束和父进程的运行是一个异步过程，即父进程永远无法预测子进程到底什么时候结束。那么会不会因为父进程太忙来不及wait子进程，或者说不知道子进程什么时候结束，而丢失子进程结束时的状态信息呢？不会。因为UNIX提供了一种机制可以保证只要父进程想知道子进程结束时的状态信息，就可以得到。这种机制就是：在每个进程退出的时候，内核释放该进程所有的资源，包括打开的文件，占用的内存等。但是仍然为其保留一定的信息（包括进程号the process ID，退出状态the termination status of the process，运行时间the amount of CPU time taken by the process等）。直到父进程通过wait / waitpid来取时才释放。但这样就导致了问题，如果进程不调用wait / waitpid的话，那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵尸进程，将因为没有可用的进程号而导致系统不能产生新的进程。此即为僵尸进程的危害，应当避免。

僵尸进程本身是没有危害的，但是如果不对对其进行释放，其就会占用相应的硬件资源，如果出现大量的僵尸进程抢占了硬件资源，那么我们的计算机就由于资源不足而难以运行，这也是大多数利用僵尸进程攻击计算机和服务器的方式。

进程的创建

创建新的进程需完成以下过程：

- 1) 为新进程分配一个唯一的进程标识号 (pid)，申请一个空白的PCB，若PCB申请失败则创建失败
- 2) 为进程分配资源，为新进程的程序和数据以及用户栈分配必要的内存（在PCB中体现）
- 3) 初始化PCB，特别是设置进程的优先级
- 4) 将新进程插入就绪队列，等待被调度运行

在程序中使用系统调用：

windows: CreateProcess

linux(unix): int pid = fork() //对于子进程而言，返回0，父进程则返回子进程的pid

进程的管理

在unix/linux操作系统中，通常使用这样几个函数来对所创建的进程进行管理：

fork()	Create a child process as a clone of the current process. Returns to both parent and child. Returns child pid to parent process, 0 to child process.
exec (prog, args)	Run the application prog in the current process with the specified arguments (<i>replacing any code and data that was in the process already</i>)
wait (&status)	Pause until a child process has exited
exit (status)	Tell the kernel the current process is complete and should be garbage collected.
kill (pid, type)	Send an interrupt of a specified type to a process. (a bit of a misnomer, no?)

fork 函数实现的是创建一个子进程，而且其有两个返回值，对父进程返回子进程pid，而对子进程返回0

exec 函数实际上是通过参数来更换原来进程中的相应数据和代码段，其主要是用于子进程

wait 函数时等待当子进程退出时，返回相应的status，根据相应状态，对其进行处理

exit 函数一般而言是用于进程的退出，但是退出不代表将进程销毁，进程这个时候会保存一个僵尸进程的数据结构，保存相应的信息

kill 函数会将进程的相应内容全部销毁，释放相应的资源

进程间切换

由于进程数量远比CPU核数多得多，因此，需要采用进程调度的方式，首先把握其目的：让每个进程都觉得自己独占CPU（硬件资源）。

- 一个进程A从运行态 → 就绪态/阻塞态
- 另一个进程B从就绪态 → 运行态
- 保存进程A的内核态寄存器到A的PCB
- 用进程B的PCB为内核态寄存器赋值
- 根据CS:IP跳转

下面根据相应的代码来进行具体的分析：

ctx_switch(&old_sp, new_sp)

ctx_switch: // ip already pushed!

```
pushq %rbp  
pushq %rbx  
pushq %r15  
pushq %r14  
pushq %r13  
pushq %r12  
pushq %r11  
pushq %r10  
pushq %r9  
pushq %r8  
movq %rsp, (%rdi)  
movq %rsi, %rsp  
popq %r8  
popq %r9  
popq %r10  
popq %r11  
popq %r12  
popq %r13  
popq %r14  
popq %r15  
popq %rbx  
popq %rbp  
retq
```

USAGE:

```
struct pcb *current, *next;  
  
void yield(){  
    assert(current->state == RUNNING);  
    current->state = RUNNABLE;  
    runQueue.add(current);  
    next = scheduler();  
    next->state = RUNNING;  
    ctx_switch(&current->sp, next->sp)  
    current = next;  
}
```

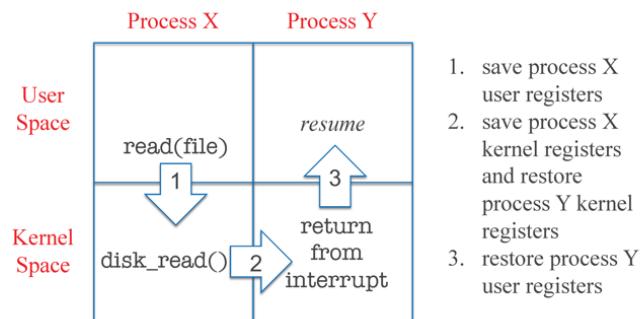
如果scheduler()返回空怎么办?

可以看到，对于相应的进程切换，其实质非常简单，只需要将当前正在运行的进程放入就绪态队列中，然后调用scheduler函数，进行相应的调度，然后将需要调用的下一个进程来进行切换即可。

如果scheduler()返回空，则代表所有进程都已经运行完毕，这个时候，会执行空转。但对于一台计算机而言，从开始而言，就会有第0号进程，这个进程从计算机开始运行到结束都一直存在，因此一般而言，不会出现这种情况。

上下文切换

- 中断、异常、系统调用
 - CPU从用户态栈 → 内核态栈 → 中断栈
- 进程间切换
 - 发生在内核，进程A的PCB → 进程B的PCB



上下文切换，如图所示，分为这两种情况：第一种情况是陷入内核，第二种情况则是进程切换。

其主要可以理解为这样三个步骤：

1. 保存当前进程的PCB到内核的内存中
2. 加载下一个需要被调度的进程的PCB到内核的内存中
3. 根据新需要被调度的PCB，更新相应的寄存器，并更新PC地址

上下文切换具有并发性和共享性

！！引入进程的目的

更好地使多道程序并发执行，提高资源利用率和系统吞吐量，增加并发程度。

！！引入线程的目的

由于进程上下文切换的代价很大，同时，进程间的通信十分复杂，一般而言，不同的进程之间的资源是非共享的，每个进程独占自己的资源。但是，对于某些操作而言，需要共同使用一些资源进行处理：**减少程序在并发执行所付出的时空开销，提高OS的并发能力**

进程和线程的关系

- **进程：OS为正在运行的程序提供的抽象**

- ✓ **并发性**: 每个进程顺序执行一系列指令流(代码段)
 - ✓ **保护性**: 每个进程具有独立的内存空间

- **线程：OS为正在运行的程序提供的并发性抽象，它可以被理解为轻量级进程**

- 一个线程代表一组顺序执行的指令序列(a.k.a, 任务)
 - OS可以像对进程一样，创建、挂起、恢复、运行线程
 - 一个线程必须从属于某一个进程，由线程ID, PC, 寄存器集合和堆栈组成，是基本的CPU执行单元(它不拥有系统资源)
 - 属于同一个进程的多个线程相互共享进程资源且是互信的(i.e., 线程解耦了并发性和保护性)

□ 引入线程后，进程将只作为除CPU外的系统资源的分配单元，线程则作为CPU调度单元（同一地址空间内的上下文切换代价较小）

线程的一些优点：提高了进程间的并发性，降低上下文切换代价，线程创建代价低，通信容易，可以利用多核。

每个线程就具有其TCB (thread control block)

进程为什么需要线程?



- 性能: 充分利用多核CPU
- 自然地呈现出程序结构
 - 描述了属于同一程序下的并发任务
 - e.g., 更新屏幕、从磁盘获取数据、获取用户输入信息
- 高响应性
 - 交互性线程(处理用户输入输出)与功能性线程分离
 - 功能性线程可以在后台运行
- 隐藏慢速的I/O操作
 - 在某个线程处于I/O等待状态时, 其他功能线程可以被调度执行

进程 VS. 线程



- | | |
|----------------------------------|---------------------------------|
| • 具有数据段, 代码段, 堆栈段 | ➤ 具有自己的栈 |
| • 至少拥有一个线程 | ➤ 必须从属于某一个进程 |
| • 进程终止→资源回收、线程终止 | ➤ 线程终止→线程拥有的栈回收 |
| • 进程间通信需要通过OS和数据传递
(e.g., 信号) | ➤ 线程间通信只需通过共享内存
(e.g., 全局变量) |
| • 拥有隔离的内存地址空间(i.e., 其他进程不能访问) | ➤ 同一进程下的其他线程可以访问自己的栈 |
| • 创建进程和进程间切换会带来较高的系统开销 | ➤ 创建线程和线程间切换不会带来较高的系统开销 |

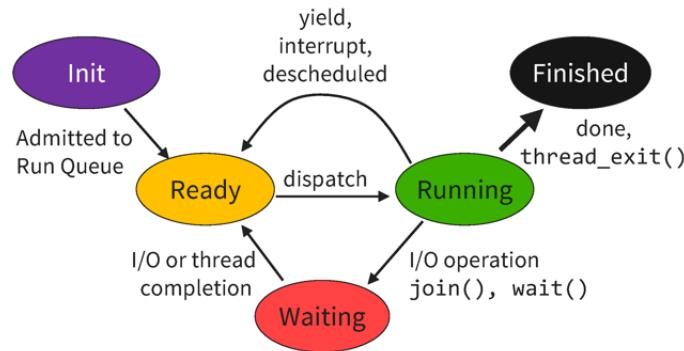
进程间通信

这一块可以留到后面具体进行了解, 暂且跳过

线程的生命周期

线程的生命周期与进程基本相同, 也分为如图的五个状态

Thread is Finished (Process = Zombie)



TCB: on Finished list (to pass exit value), ultimately deleted

Registers: TCB

线程的类型

线程主要按照相应的资源和创建分为；用户级和内核级线程，这两种线程具有不同的特点和各自的优势。

✓ 线程切换不需要内核权限	✓ OS可以调度多线程到多核中运行
✓ 不受当前OS的限制(对OS透明) <small>不受操作系统的影响</small>	✓ 如果进程的一个线程被阻塞, 可以调度其另外的进程运行
✓ 线程调度可以根据用户程序需要	✗ 创建较慢(权限, 安全等)计算机要解决性能、稳定、安全的问题(按顺序)
✓ 创建更快, 管理更方便(分布式)用户自己管理	✗ 同进程下的线程切换需要内核模式的切换(内核态的模式不一样)
✗ 需要系统调用时会阻塞(内核态)	
✗ 多线程程序无法使用多核CPU将线程都视作进程, 对用户而言是多个线程, 对操作系统只认为有一个, 很低效	没有一种方式将所有目的都达到

显然这两种线程的方式都不大合适，因此，考虑使用用户和内核结合型，来达到相应的结果：而对其的映射结果有，1对1， 多对1， 多对多这三种不同的映射方式。

！！理解用户态和系统态

- 用户态程序的本质就是使用资源来完成某种功能
- 系统态程序的本质就是管理资源进而更好地为用户服务

对于进程和线程的理解：

- 进程：用户态程序所需资源的抽象集合
- 线程：用户态程序提供功能的抽象集合

进程调度

资源类型

对于计算机而言，实际上所有的程序都是在硬件上进行运行的（也就是所有的进程都需要占用相应的资源来运行，计算机硬件的实质就是资源）

- 可抢占式资源：OS可以很容易将进程A的某些资源收回，用于其它进程，并且过段时间可以将相等的资源再次分配给进程A
- 非抢占式资源：OS不能很容易将进程A的某些资源收回，只能等待进程A主动放弃这些资源（e.g.读写磁盘内容）

可以知道的是，资源是有限的，无法满足所有进程同时的需要，因此需要对其进行相应的管理，对于不同资源类型：

抢占式资源,采取资源调度的方式，进程拥有资源的时间，并以此决定进程服务的顺序（是一种时分方式）

非抢占式资源，进程获得资源的数量，OS决定进程可以获得多少资源，这是一种空分的方式。

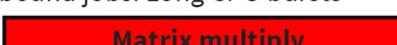
操作系统中的资源调度

- 计算机操作系统中的资源调度
 - **CPU调度**：从就绪态任务队列选择下一个使用CPU的任务
 - **磁盘调度**：选择下一个对文件或块设备的读写操作
 - **网络调度**：选择下一个发送或处理的数据包
 - **内存页替换调度**：选择cache中哪个内存页被替换

计算机通过**操作系统**主要完成**计算、通信和存储**三个方面功能

进程调度实质上是CPU调度，其是非阻塞的过程。

类型分类：

- **进程/线程的类型**
 - 从资源角度来看，本质是交替改变CPU和I/O操作
 - CPU密集型和I/O密集型任务
- CPU-bound jobs: Long CPU bursts
- 
- I/O-bound: Short CPU bursts
- 
- 批处理：用户不需要参与，一堆处理，后台自己执行（流处理：一时来
交互式：需要用户输入等（cin cout）用户参与，前台
- 从与用户参与度来看，可以分为批处理和交互式任务
 - 从安全权限角度来看，可以分为系统级和用户级任务

实际上，对于不同类型的进程和线程，我们希望的是对其进行不同的调度等级，我们尽可能的是希望使用CPU密集型的进程（线程），而非经常性地调用I/O密集型的。

CPU资源调度指标

1. 响应时间：用户态程序完成某些功能所需的总时间
2. 吞吐量：单位时间CPU完成的任务数 完成任务越多，
吞吐量越大
3. 系统开销：完成任务所需的额外资源量(时间,空间)
4. 公平性：每个任务使用CPU的时间或资源量
5. 饥饿度：任务两次被调度之间的等待时间
 - 一个完美的调度策略同时满足：
 - ✓ 最小化响应时间
 - ✓ 最大化吞吐量
 - ✓ 最大化资源利用率
 - ✓ 公平的分配所有资源

什么最重要，就最优化什么，其他都是约束，最优理论。

CPU资源调度策略

先来先服务策略(FIFO/FICS)

是一种非抢占式调度，第一个到达的请求CPU的任务获得CPU资源

优点：实现简单，在实际应用过程中不需要进行复杂的调度操作，并且满足公平性 调度公平

缺点：

1. 各项指标不高，很大受限制于进程的请求顺序
2. 头阻塞问题
 - a. 后续较短的任务会被前期较长任务阻塞
 - b. 前期大量的CPU密集型会阻塞后续的I/O密集型，反之亦然
3. 无法很好地支持交互式任务

最短任务优先策略 (Shortest Job First,SJF)

根据前面的指标，SJF试图最小化平均响应时间：存在两种算法的形式

➤根据任务类型存在两种形式

- 非抢占式：一旦任务获得CPU后，只有当其完成后才行调度下一个任务
- 抢占式：若一个任务获得的CPU执行时间少于其所需CPU的执行时间，则发生抢占（执行最短剩余时间优先）

数代码段长短（可能也无法准确界定）、程序先前在CPU内的运行时间

SJF存在的问题：

1. 由于很多问题是Online问题，故无法得到最小的平均响应时间
2. 可能会“饿死”需要执行时间较长任务
3. 在实际上，无法准确估计任务执行时间

- 指数带权平均法
- T_n 代表第 n 次被 CPU 调度的 **真实** 执行时间
- E_{n+1} 代表第 $n+1$ 次被 CPU 调度的 **预计** 执行时间
- 选择一个合适的 0-1 权值 w , $E_{n+1} = w * T_n + (1-w) E_n$

对于 SJF 的实现，一般采用的是红黑树进行实现，因为需要进行插入，删除和查找操作，达到一个平衡。

轮询策略（Round Robin， RR）

轮询策略的出发点：考虑公平性、避免任务饿死的策略

- 考虑公平性、避免任务饿死的策略
 - 每个任务会获得相同的 CPU 执行时间
 - 当一次执行时间用尽后，当前任务将被放入就绪态队尾



- 该策略的优缺点
 - ✓ 为每个任务 **分配** 相同的 CPU 资源
 - ✓ 较低的平均等待时间
 - ✓ 当任务所需执行时间差别较大时，较低的平均响应时间
 - ✓ 当任务数不是特别大时，可以很好地支持交互式任务
 - ✗ ???

问题：

- 在轮询策略中，有一个参数需要进行控制，即执行的 CPU 时间，会有这样的情况：

**越小则导致过多切换
越大则越趋近于 FIFO**

- 如果任务所需执行时间差别不大的时候，那么调度显得意义不大，反而浪费资源
- I/O 密集型和 CPU 密集型任务所获得的时间相同，但是实际上这是不必要的 不公平

CPU 资源调度（基于优先级的策略）

优先级调度（Priority Scheduling）

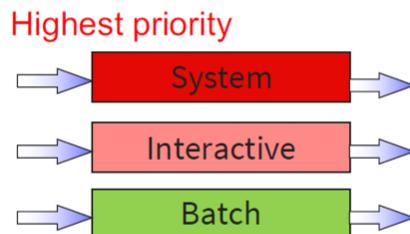
- OS为每个任务分配一个优先级号
 - 优先级号越小(大)则代表任务具有更高(低)优先级
 - 优先调度高优先级的任务
 - 最短任务优先策略也是一种优先级调度,其优先级是由预估的下次所需CPU执行时间决定的
 - 低优先级任务可能会被“饿死” 如何解决这一任务饿死问题?
 - 根据等待时间提升它们的优先级 改优先级(难题:
提谁、提多少?)
 - 使用二叉堆、AVL树或者红黑树来实现
 - 堆不能很好地支持搜索O(n)
 - 平衡搜索树→功能是搜索,平衡是效率,AVL查询快(维护代价大,在工业界中几乎不用),红黑树代价低

多级队列调度 (Multi-level Queue Scheduling)

多级队列调度



- 根据任务类型存在多个就绪态任务队列(纵向)
 - 系统型任务
 - 交互型任务(I/O密集型)
 - 批处理任务(后台处理)
 -
- 不同的任务队列可以采用不同的任务调度策略(横向)
 - FIFO(通常), SJF, RR...



这种策略有什么问题吗?

任务类型有时很难提前获得
Prediction??

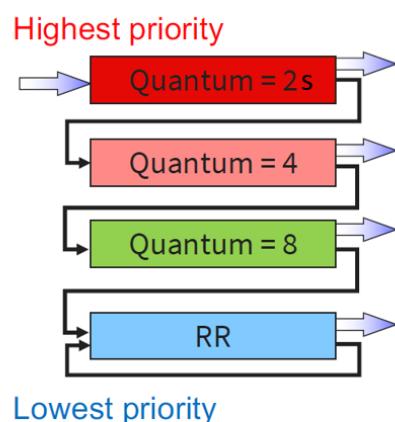
任务类型可能会随程序进行发生改变
No Re-classification

多级反馈队列调度 (Multi-level Feedback Queue Scheduling)

多级反馈队列调度 (现代操作系统使用)



- 类似多级队列调度,但是
 - 任务类别是动态的
 - 到来任务都进入最上层
 - 任务用完CPU配额时间后会进入下一层队列等待
 - 每层可使用不同的策略
- 需要考虑较多的参数
 - 队列数目以及配额时间
 - 每层的策略
 - 任务何时升级或者降级



对实时性应用(e.g., 流媒体-游戏)支持较差



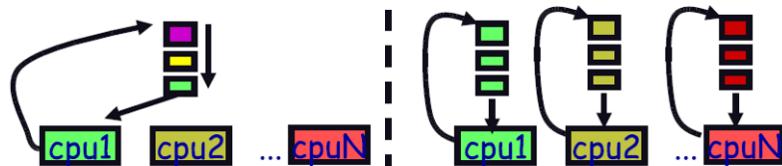
实时任务调度

- 实时任务具有延时约束要求
 - 终止时间 (deadline) 要求
 - 软deadline要求 → 超过时间QoS急剧下降
 - 硬deadline要求 → 超过时间任务终止
 - 周期性事件
 - 例如任务A,B需要每100ms和200ms调度执行，
每次执行需要20ms和40ms CPU时间
 - 若所需CPU > 任务调度周期，则OS无法完成调度
 - 实时任务调度始终是研究热点 (ML, 工业IoT, 流媒体...)
 - 目前存在大量与优先级相关的策略
 - 最经典也是最简单的是Earliest Deadline First (最近deadline优先)

多核任务调度



- 更复杂的决策：哪个任务在哪个CPU上运行
- 任务在不同CPU核切换执行会带来系统开销
- **姻亲调度 (将同一进程的线程调度在相同CPU执行)**
是否存在问题？

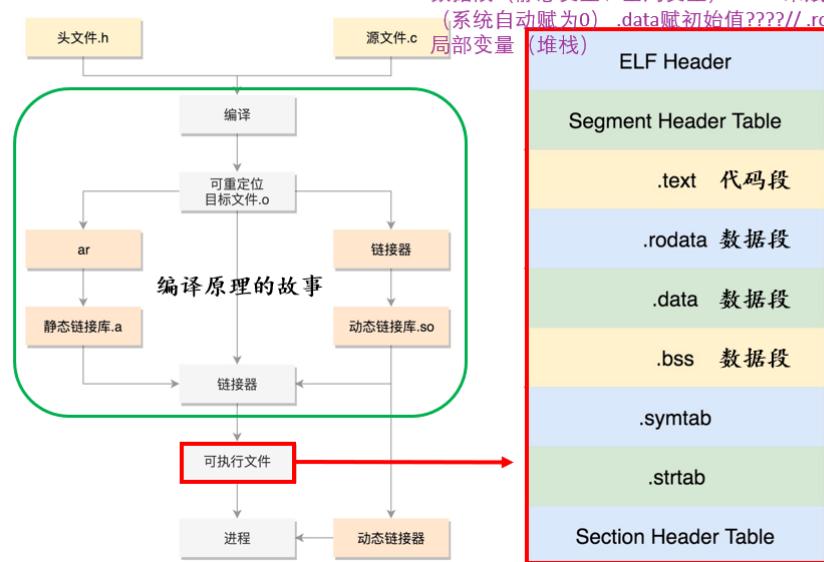


进程和线程相关数据结构

进程和线程调度方式

linux下的代码可执行文件为：

Linux下代码 → 可执行文件



问题：.rodata , .data , .bss这三者之间有什么区别？

其具体创建进程的过程为：



linux下多线程的代码

Linux下的多线程

主线程必须要等joinable的线程结束，才能结束。

不然主线程可能会比子线程结束的更早。

阻塞（完成才能结束）、非阻塞（不停地问）、异步（那东西结束了会通知）???????????



joinable属于阻塞型

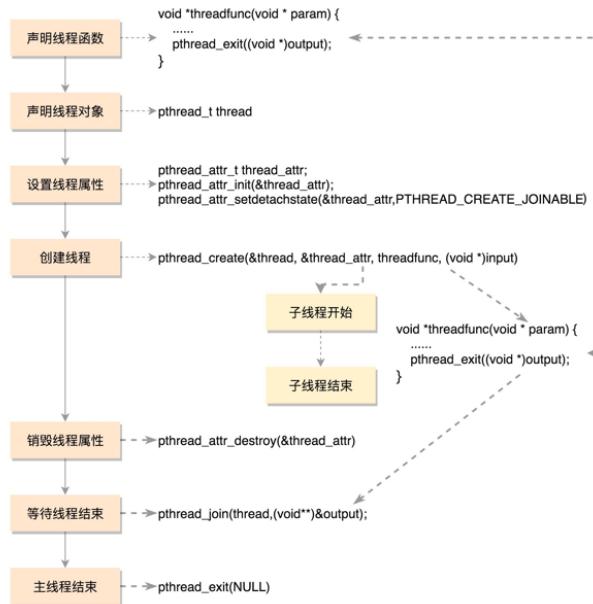
```
int main(int argc, char *argv[]){
    char files[NUM_OF_TASKS][20] = {"file1.avi", "file2.rmvb", "file3.mp4"};
    pthread_t threads[NUM_OF_TASKS];  
创建线程
    int rc; int t; int downloadtime;
    pthread_attr_t thread_attr; 线程属性
    pthread_attr_init(&thread_attr);
    pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_JOINABLE);
    for(t=0; t<NUM_OF_TASKS; t++){
        printf("creating thread %d, please help me to download %s\n", t, files[t]);
        rc = pthread_create(&threads[t], &thread_attr, (void *)files[t]);
        if (rc){
            printf("ERROR: return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_attr_destroy(&thread_attr);
    for(t=0; t<NUM_OF_TASKS; t++){
        pthread_join(threads[t], (void **)&downloadtime); 阻塞!等待线程结束
        printf("Thread %d downloads the file %s in %d minutes.", t, files[t], downloadtime);
    }
    pthread_exit(NULL); Downloadtime是长整型, 4字节
}
```

主线程会等待这个线程结束

无法保证线程的执行顺序!

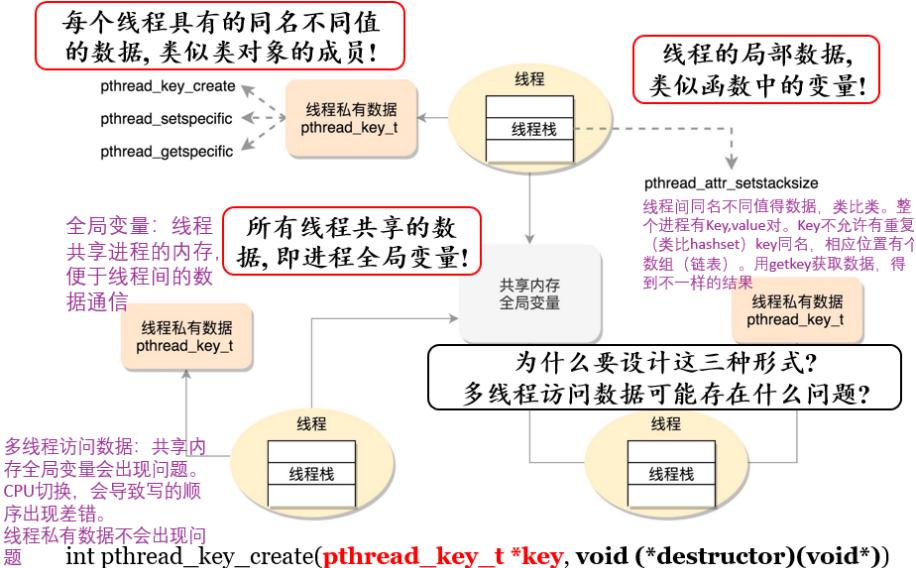
通过代码，我们可以这样理解多线程：

Linux下的多线程





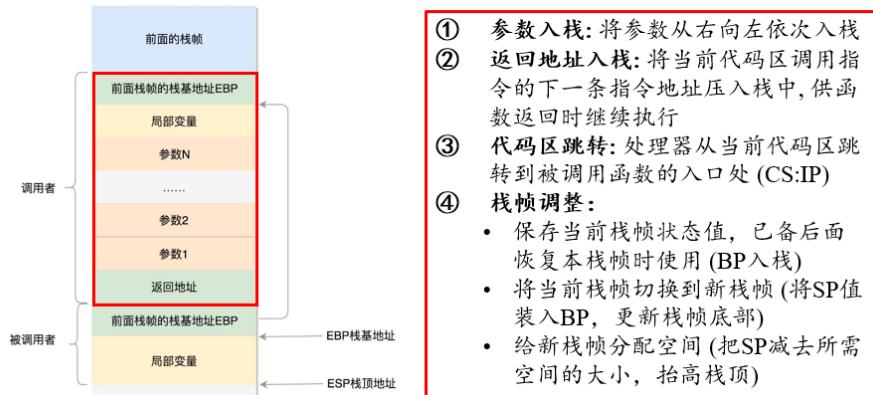
线程可使用的数据形式



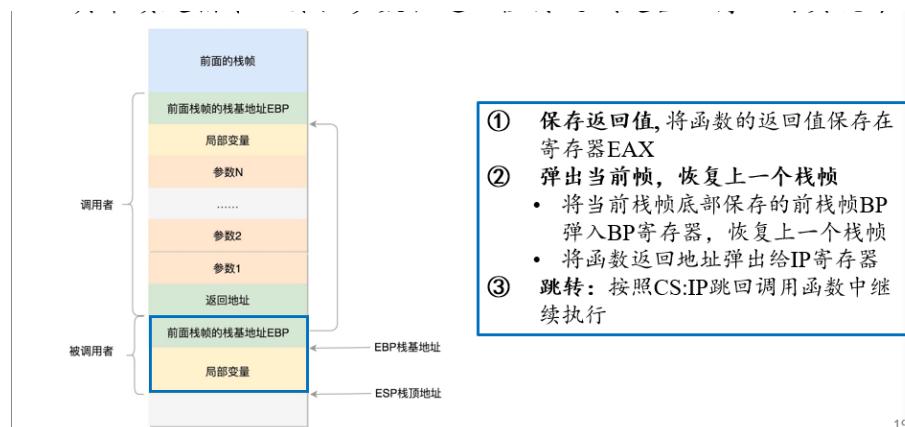
关于用户态和内核态 对于内存管理而言, 内核态的内存是所有进程共享的。

关于函数调用

- 程序的运行通常是一个函数调用另一个函数
- 函数调用通过用户栈来实现 (IP寄存器jump的过程)
- 其本质是指令跳转、参数和返回值传递 (是否记得如何实现?)



而对于函数的返回而言:



系统调用

- 以标准库**open**函数为例:


```
int open(const char *pathname, int flags, mode_t mode)
```
- 当调用**open**函数时会进一步
 1. 调用宏命令DO_CALL(syscall_name, args), 其包含
 - ✓ 保存open的参数到寄存器中
 - ✓ 根据系统调用名称得到系统调用号保存在ax寄存器中
 - ✓ 执行ENTER_KERNEL → INT 0X80 (软中断, 陷入内核)
 2. 调用ENTRY(entry_INT80_32)
 - ✓ 将用户态寄存器 (用户态的CPU上下文) 保存在pt_regs中
 - ✓ 调用do_syscall_32_irqs_on

内存管理

对于进程而言, 对于内存资源, 有两个需求:

1. 希望每个进程独享机器所有内存
2. 不希望进程之间互相访问 (保护性)

因此, 需要对相应的内存资源进行管理, 涉及到内存管理机制。

内存管理的目标

- 隔离性

不希望

不同的进程状态在物理内存中相互冲突

- 共享性

希望

有选择性的共享某些物理内存, 实现高效进程间通信

- 虚拟化

希望

为每个进程提供一种“幻觉”, 它可以独占所有物理内存 → 引入虚拟内存

- 利用率

希望

能尽可能的高效利用有限的物理资源

对于进程的内存布局, 从虚拟内存的角度来看:

虚拟地址从低到高 (32位OS下, 用户态3G, 内核态1G), 而更为重要的是: 不同进程的相同用户态虚拟地址会映射到不同的物理地址 (隔离), 不同进程的相同内核态虚拟地址会映射到相同的物理地址 (共享)

虚拟内存到物理内存的映射

分段映射机制

- 分段管理方式

1. 按照用户进程中的自然段划分虚拟内存
2. 考虑内存保护、动态增长和以及动态链接等方面的需求
3. (对物理内存) 段内要求连续, 段间不要求连续 (e.g., 二维)

例如, 用户进程由主线程、两个子线程、栈和一些全局组成, 则可以将其划分为5个段, 每个段从0编址, 并分配一段连续的物理内存

4. 32位虚拟内存地址由段号(16位)和段内偏移量(16位)组成, i.e., 一个进程最多 2^{16} 个段, 最大段长 2^{16} KB
5. 进程的虚拟内存地址需要由编译器提供!
6. 段表: 每个进程都有一个虚拟内存到物理内存的映射的段表, 每个段对应进程的一个功能段, 段表项纪录了该段在内存的起始位置以及长度

一些问题:

1. 分段映射能否解决利用率低 ($10 + 70 + 30 > 100$) 的问题?
2. 是否存在不同进程的段指向同一段物理内存?

分页映射机制

内存管理单元 (MMU -> memory management unit)

包含与地址映射有关的所有组件, 分段映射的段控制寄存器, 分页映射的页表寄存器、高速缓冲存储器 (TLB)

分页机制:

- 将进程大小进行细粒度切分 → 页的概念

- 虚拟内存页(page), 物理内存页(frame), 磁盘块均为4KB
- 虚拟页和物理页是一一对应关系
- 长时间不用的物理页会被写入磁盘块 (页换出), 待重新需要时在从磁盘块写入到物理页 (页换入)
- 32位虚拟内存地址由页号(20位)和页内偏移量(12位)组成, i.e., 一个进程最多4M个虚拟页

为什么页内偏移量是12位?

页内偏移量是12位的原因为: 页大小为 $4KB = 2^{12} bytes$

问题: 页表存储的位置和占用内存大小?

- 页表存储在每个进程中, 采用数组结构
- 32位的进程虚拟内存为4G, 每个页面4KB, 则最大存在1M个虚拟页, 即1M个页表项
- 一个页表项的大小至少要3B, 因为存在1M的物理页(20位), 一般采用4B
- 页表会占用 $4B \times 1M = 4MB$ 内存

分页映射机制中所出现的问题:

1. 存取数据或指令需要两次访问内存, 第一次是访问页表, 确定所存取的数据或指令的物理地址; 第二次是根据物理地址存取数据或指令
2. 页表太大, 内存利用率降低

针对第一个问题: 引入了高速缓冲存储器TLB

针对第二个问题, 引入了二级分页机制

- 对这4M (20位) 内存再次进行分页操作 $4M/4KB = 1K$ 虚拟页
- 引入页目录 (10位) 来指向这1K个虚拟页
- 虚拟页为4KB, 页表项大小为4B, $4KB/4B=1K$ 个页表项 (10位)
- 虚拟内存地址 (10位页目录+10位页表项+12位偏移量)
引入二级页表怎么所需内存更大了?
- **页目录所需内存大小是4KB, 即1个内存页 (二级页表)**
- **一级页表共1024项, 每一个页表项为1个内存页**

虽然看起来引入了二级页表, 所需内存变大了, 但是实际上:

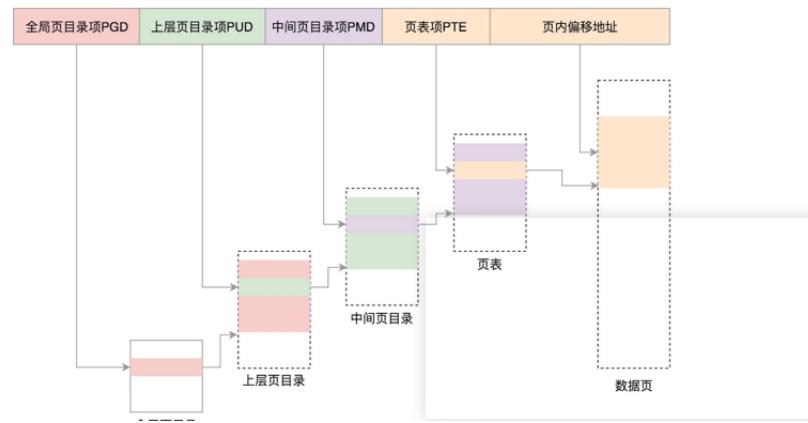
**仍以40MB进程为例,
我们需要将页目录加载到内存
(1KB), 因为它是数组!!!**

**但是我们可以根据进程执行中
所需要的页面, 动态加载其对应
的一级页表 (即1024个页中
的一些), 因为每个内存页可以动
态创建或回收!**

**在这种情况下, 二级页表占
用的内存会远远少于一级页表!**

对于64位操作系统而言, 目前只使用了其中的48位进行选址, 相应地, 会将二级分页机制变为四级分页机制

即全局页目录, 上层页目录, 中间页目录和页表项



段页结合映射机制

- 分段机制能反映程序的逻辑结构, 有利于段保护
- 分页机制能有效地提高内存利用率
- 因此就形成了段页结合映射机制!
 1. 在段页式机制下, 虚拟内存地址由段号, 页号, 页内偏移组成
 2. 段表: 段号 → 页表首地址
 3. 页表: 页号 → 物理页
 4. 页内偏移量与分页机制相同
 5. 也就是说, 一个进程现在有1个段表和N个页表, 需要访问内存3次才能存取数据和指令!

反向页表映射机制

动机:

- 每个进程都存在一个页表
- 进程运行中使用的内存页通常不多, 因此使用的页表项通常是稀疏的

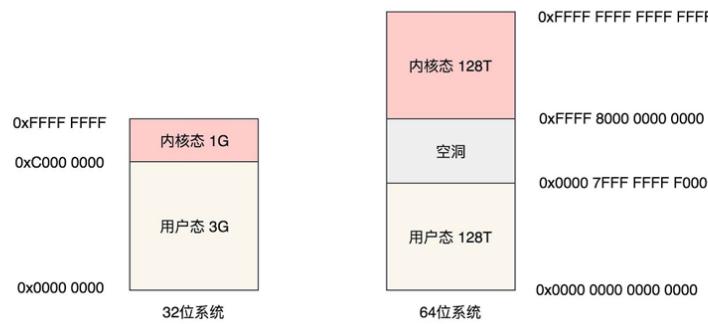
进程创建与写入时复制

当进程刚开始创建的时候, 他们虽然都具有一个单独的虚拟内存空间, 但其映射到的物理空间是相同的, 此时虚拟页均标记为 COW

- 当P1或P2写栈操作时, 它们对应的栈虚拟页均不再是COW
- 当P2执行exec()时, P2会被分配新的页表映射关系, 所以虚拟页也均不再是COW

linux进程的内存管理

- 每个进程都有一个task_struct结构 (类似PCB)
- task_struct中的内存管理为struct mm_struct *mm
- mm_struct中存在一个成员 long task_size, 当执行新的进程时, 会进行如下操作 task_size = TASK_SIZE; 后者在32位系统中定义为3G



- mm_struct结构中的部分成员
 - unsigned long mmap_base; /* 虚拟地址用于内存映射的起始地址 (该区域用于加载动态链接库以及使用malloc申请一大块内存)
 - unsigned long total_vm; /* 映射页面的数目, 不可能1024个页面都映射
 - unsigned long locked_vm; /* 不能被换出的页面
 - unsigned long pinned_vm; /* 不能被移动的页面
 - unsigned long data_vm; /* 存放数据的页的数目 (数据段)
 - unsigned long exec_vm; /* 存放代码的页的数目 (代码段)
 - unsigned long stack_vm; /* 存放堆栈的页的数目 (堆栈段)
 - unsigned long start_code, end_code, start_data, end_data; /* 数据段和代码段开始和结束为止
 - unsigned long start_brk, brk, start_stack; /* 堆和栈的起始位置, brk为堆当前的位置, malloc申请小内存时会更新brk, 栈的当前位置呢???

- mm_struct结构中还有两个重要成员
 - struct rb_root mm_rb (红黑树, 用户内存快速查找内存区域)
 - struct vm_area_struct *mmap (链表, 用以链接每个内存区域)

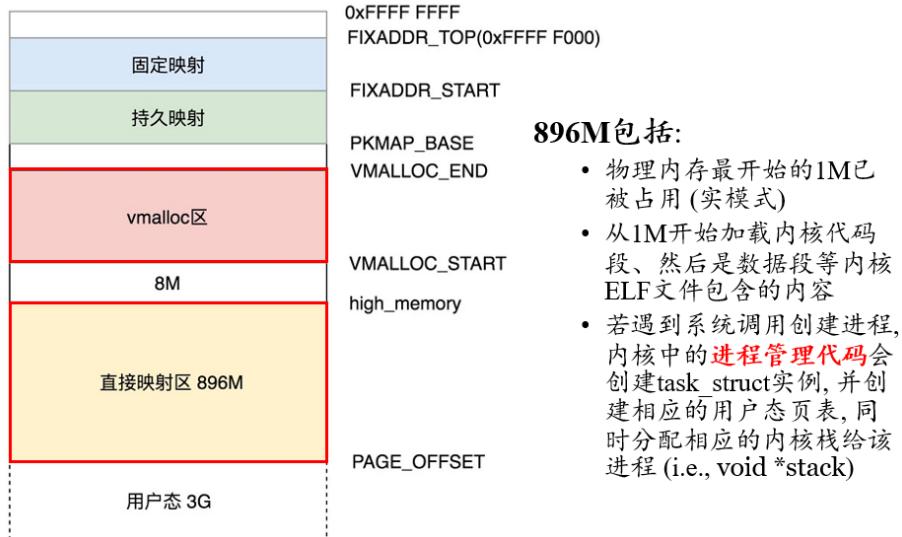

```
struct vm_area_struct {
    unsigned long vm_start; /* 该区域在用户空间的起始和终止位置
    unsigned long vm_end;
    struct vm_area_struct *vm_next, *vm_prev; /* 链表的前后指针
    struct rb_node vm_rb; /* 该区域的红黑树节点, 用于搜索
    struct mm_struct *vm_mm; /* 该区域属于哪个段 (保护性)
    struct list_head anon_vma_chain;
    struct anon_vma *anon_vma; /* 虚拟内存映射到物理内存
    const struct vm_operations_struct *vm_ops; /* 可以对该区域的操作 (保护性)
    struct file * vm_file; /* 虚拟内存映射到文件
    void * vm_private_data;} __randomize_layout;
```

Linux进程内核代码 □ 内核态

内核态的虚拟内存和某一个进程无关, 所有进程通过系统调用进入内核后的虚拟内存是相同的

内核态仍然使用的是虚拟内存

32位系统的内核态使用1G内存(高地址), 其中896M为直接映射区, 该空间是连续的, 并且和物理内存的映射很简单 (i.e., 虚拟内存地址-3G即为物理内存地址), 也就是说其对应着物理内存最开始的896M



缺页操作

• 缺页操作发生的情景

1. 访问一个没有权限的段空间 → 终止进程
2. 访问一个尚未加载到物理内存的磁盘块
 - 物理内存会分配一个frame, 并执行页换入操作
3. 对bss段的数据进行写操作
 - 物理内存会分配一个frame, 并根据内容写入相应的值
4. 对COW页进行写操作
 - 物理内存会分配一个frame, 拷贝原始COW页内容, 并根据新的内容进行更新

页换入操作的具体步骤

1. 寻找一个空闲的物理页
2. 发起磁盘块加载请求
3. 阻塞当前进程P1
4. 上下文切换到新的进程
5. 当磁盘块加载完毕后更新页表项 (frame号+标识)
6. 将进程P1放回到Runnable队列中

页换出操作

1. 寻找页表中所有与其相关的表项(共享页,COW页)
2. 将这些页表项均置为失效
3. 清楚TLB表中对应的表项
4. 如果需要的话,将物理内存页的内容写回磁盘块
 - 代码段和只读数据不需要写回

页面替换算法

- **Random:** 随机换出一个物理页面(worst case)
- **FIFO:** 根据页面换入顺序,换出最早换入的页面
- **OPT:** Belady's algorithm (offline)
- **LRU:** 换出未使用时间最长的页面
- **LFU:** 换出单位时间内最少使用的页面
-

Bleady 异常

在先进先出算法(FIFO)——选择装入最早的页面置换的过程中,可以通过链表来表示各页的装入时间先后。FIFO的性能较差,因为较早调入的页往往是经常被访问的页,这些页在FIFO算法下被反复调入和调出,并且有Belady现象。所谓Belady现象是指:采用FIFO算法时,如果对一个进程未分配它所要求的全部页面,有时就会出现分配的页面数增多但缺页率反而提高的异常现象。

对于OPT算法而言,其本质上是贪心算法,同时也是离线算法,显然这个算法是不可能的,这只是一个不可达到的最优解情况。其策略是:替换其未来被访问的时间距离当前时间最远的那个页面。

LRU算法是通过过去拟合将来,其本质是换出未使用时间最长的,需要考虑内存开销,故一般对其做近似实现:

输入输出设备

计算机系统中的输入输出设备有很多,包括键盘,鼠标,显示器,网卡.....

在计算机系统中,CPU不直接和I/O设备打交道,它们之间存在一个叫做设备控制器的组件(中间件)

- CPU通过写设备控制器的寄存器,实现对控制器下发指令
- CPU通过读设备控制器的寄存器,实现对I/O设备状态查看
- CPU读写寄存器相比直接操作硬件要更标准、简单

输入输出设备分为块设备和字符设备

1. 块设备中，将信息存储在固定大小的块中，比如硬盘，有相应的地址，可以进行寻址。
2. 发送或接受的是字节流，不用考虑任何块结构

若I/O设备传输的数据量较大，其设备控制器会设置数据缓冲区，只有缓冲区的数据达到阈值才会真正执行读写I/O设备。

- 每个设备控制器的每个寄存器会被分配一个**I/O端口**，OS支持一些特殊的汇编指令 (e.g., in/out指令) 来使得CPU操作这些寄存器
- 设备控制器的数据缓冲区，会被分配一段内存空间，这样CPU就像读写内存一样读写数据缓冲区
- 设备控制器可以自行处理一些任务 (即芯片具有一定计算功能)，若CPU给设备控制器发送一个指令 (读取一些数据)，当其完成指令后，如何通知CPU？

多种不同I/O模式

- 阻塞I/O模式（同步）
- 非阻塞I/O模式（同步）
- 异步I/O模式

对比不同的I/O控制模式

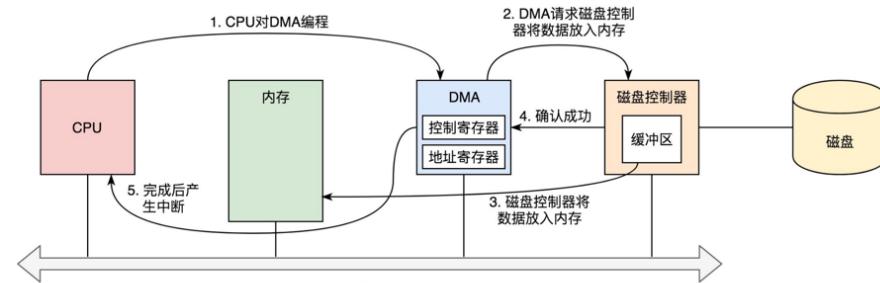
非阻塞I/O和异步I/O模式



- 非阻塞I/O模式 → 轮询等待
 - 设备控制器的寄存器会有状态标志位，通过该值来确定某个指令操作是否完成
 - while (true) {直至状态标志位显示完成}
 - ✗ 会参与进程/线程调度，影响整体系统性能
- 异步I/O模式 → 中断
 - OS提供一个**硬件中断控制器**
 - 当硬件完成某个任务触发中断到硬件中断控制器，该控制器会通知CPU，CPU会停下当前执行任务去处理中断（优先级）
 - 软中断（系统调用，INT xxH）vs. 硬件中断（中断控制器触发）

DMA:直接内存存取技术

DMA: 直接内存存取技术



感兴趣的同学可以进一步了解RDMA和Infiniband (当今研究大热)

适用场景和动机: 需要读取或写入大量数据到I/O设备, 若采用异步I/O模式, 则仍需要占用大量CPU时间(处理中断)

DMA技术: CPU对DMA控制器下指令, 指定从I/O设备读取多少数据到内存的某个地方, DMA控制器会发指令给设备控制器, 后者执行上述读数据任务, 待传输完毕, 设备控制器会通知DMA控制器, 并由后者触发中断通知CPU

用驱动程序屏蔽设备控制器差异

设备控制器不属于OS的一部分

每种设备控制器的寄存器、缓冲区的使用模式以及支持的指令都不同
OS需要将它们的差异进行屏蔽

设备驱动程序属于OS的一部分, 它既包含面向I/O设备控制器的代码, 又包括对OS统一的接口。

• I/O设备统一的中断流程:

1. 设备驱动程序初始化时要在OS中注册一个设备相关的中断处理函数:

do_IRQ()是发生中断后, OS会调用的函数(统一接口),
因此需要在该函数内注册一个Handler() (回调函数)

2. 当设备完成任务后, 会触发中断到硬件中断控制器, 最终会执行doIRQ(), 根据该设备注册的Handler(), 会在设备驱动程序中执行具体的Handler()代码

用文件系统屏蔽驱动程序差异

设备控制器 → 驱动程序(OS) → 文件系统 (用户)

在文件系统中实现了对用户使用接口的统一, 在linux中, 所有I/O设备都需要在/dev文件夹下创建一个设备文件

- 设备文件也具有inode
- inode不关联任何存储介质上的数据
- inode关联了该文件对应设备的驱动程序
- 设备文件分为块设备文件和字符设备文件
- 用户可以通过操作/dev/X来操作相应设备

补充: linux文件权限

对于linux而言，其一般具有三种文件权限：

r,w,x 其分别为读权限read 4 , 写权限write 2 和 操作权限 execute 1

b/c 代表块/字符设备，数字代表主/次设备号，主设备号代表驱动，linux支持多个i/p设备共用一个设备驱动，因此次设备号确定了具体是哪个设备 - 代表文件，d代表目录。

故而，可以利用相应指令查看文件设备驱动：

```
# ls /dev -l
crw----- 1 root root      5,  1 Dec 14 19:53 console  b/c代表块/字符设备;
crw-r---- 1 root kmem      1,  1 Dec 14 19:53 mem     数字代表主/次设备号,
crw-rw-rw- 1 root root     1,  3 Dec 14 19:53 null    主设备号代表驱动, Linux
crw-r---- 1 root kmem      1,  4 Dec 14 19:53 port   支持多个I/O设备共用一个
crw-rw-rw- 1 root root     1,  8 Dec 14 19:53 random 设备驱动, 因此次设备号确
crw--w---- 1 root tty      4,  0 Dec 14 19:53 tty0   定了具体是哪个设备;
crw--w---- 1 root tty      4,  1 Dec 14 19:53 tty1
crw-rw-rw- 1 root root     1,  9 Dec 14 19:53 urandom
brw-rw---- 1 root disk     253,  0 Dec 31 19:18 vda
brw-rw---- 1 root disk     253,  1 Dec 31 19:19 vda1
brw-rw---- 1 root disk     253, 16 Dec 14 19:53 vdb
brw-rw---- 1 root disk     253, 32 Jan 2 11:24 vdc
crw-rw-rw- 1 root root     1,  5 Dec 14 19:53 zero
```

对于windows而言，其添加新设备时，首先要看该设备有没有相应的驱动程序，如果没有则需要安装，否则该设备无法显示。

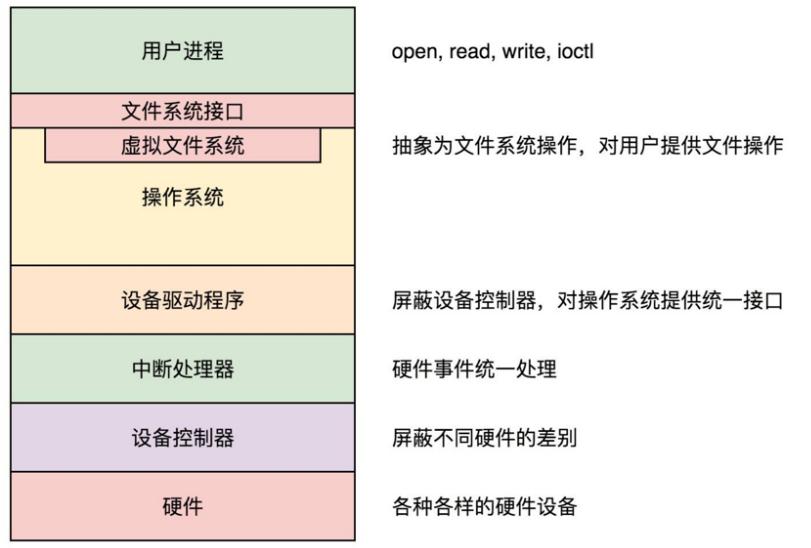
linux添加设备流程

linux安装驱动程序本质是加载一个内核模块（设备驱动程序以内核模块形式展现）

linux可以使用lsmod查看已有驱动程序，insmod加载新的驱动程序，然后在/dev目录下使用，mknod filename(设备名) type(c/b) major (驱动号)

linux中有个守护进程udev，当有设备插入系统时，其会执行上述过程，并在/dev目录下创建一个设备文件

当设备文件创建成功后，用户除了通过o/w/r等文件API操作设备外，还可以使用ioctl对设备属性修改和配置。



I/O设备工作要素总结：

- 需要一个设备驱动程序 (.ko文件)
 - 初始化函数、**中断处理函数**、设备操作函数
 - 加载设备驱动时，初始化函数被调用，并在cdev_map结构中注册该设备（利用主/次设备号进行查找），根据主设备号可以获得该设备的驱动程序
- 需要在/dev目录下创建设备文件
 - 该文件属于特殊的文件系统devtmpfs，也需要相应的dentry和inode（这里的inode与磁盘文件不同）
 - 磁盘文件的inode指向文件数据，而设备文件的inode指向设备的驱动程序（通过主/次设备号在cdev_map中获得）
- 打开和读写设备文件，类似磁盘文件操作
 - 文件描述符，file结构，path结构（inode），file_operations结构

I/O设备中断处理

1. I/O设备给中断控制器发送物理中断信号
2. 中断控制器将物理中断信号转成中断向量（interrupt vector），并发送给每个CPU（**设备号 + 中断向量号**）
3. 每个CPU都有一个中断向量表，中断发生时会根据中断向量调用**do_IRQ处理函数**
4. do_IRQ函数会将中断向量转化为抽象中断信号irq，并调用irq对应的中断描述结构irq_desc中的irq_handler_t()

中断向量

- 每个CPU必须具有一个中断向量表, 该表一共256项, 其中0-31项为系统陷入或异常, 它们必须要有中断处理函数, 在OS内核启动时会调用trap_init()函数进行设置
- 第32-127项为I/O设备中断, 在OS内核调用完trap_init()后, 还会调用init_IRQ()来初始化I/O设备中断
- 所有发生的中断最终都会调用do_IRQ()

```

__visible unsigned int do_IRQ(struct pt_regs *regs){
    struct pt_regs *old_regs = set_irq_regs(regs);
    struct irq_desc * desc; /* high bit used in ret_from_code */
    unsigned vector = ~regs->orig_ax; ..... //从AX寄存器获得中断向量号
    desc = __this_cpu_read(vector_irq[vector]); //中断向量号 → irq
    if (!handle_irq(desc, regs)) {.....}..... //处理某个设备中断???
    set_irq_regs(old_regs);
    return 1;};

```

I/O设备具有自己的中断行为, 用抽象中断信号irq表示

系统初始化时, 会调用assign_irq_vector(), 该函数的功能是将抽象中断信号irq与某个CPU的中断向量表中的表项建立映射关系 (i.e.,vector_irq)=

通过得到的中断向量号, (具体的抽象中断信号irq及其中断描述结构irq_desc结构, 并最终调用handle_irq()→_handle_irq_event_percpu()函数, 进行相应的处理。

设备中断向量处理

```

irqreturn_t __handle_irq_event_percpu(struct irq_desc *desc, unsigned int *flags){
    irqreturn_t retval = IRQ_NONE;
    unsigned int irq = desc->irq_data.irq;
    struct irqaction *action;
    record_irq_time(desc);
    for_each_action_of_desc(desc, action) { //说明某个irq已经注册过??
        irqreturn_t res = action->handler(irq, action->dev_id); //设备驱动处理??
        switch (res) {
            case IRQ_WAKE_THREAD:   __irq_wake_thread(desc, action);
            case IRQ_HANDLED:      *flags |= action->flags;   break;
            default: break;}
        retval |= res;
    }
    return retval;
}

```

- 中断是从外部设备发起, 形成外部中断, 外部中断会到达中断控制器, 控制器会发送中断向量给CPU
- 每个CPU都有一个中断向量表(idt_table), 里面存放了不同的中断向量处理函数
- 硬件中断处理函数的统一接口是do_IRQ(), 它会把中断向量通过vector_irq映射为irq_desc中断描述结构
- irq_desc中的成员irqaction描述了用户注册的中断处理函数结构

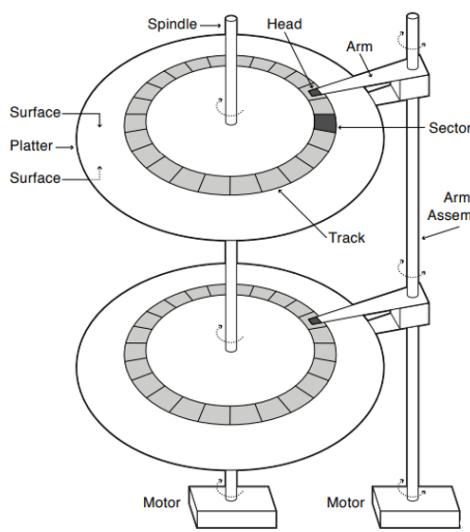
文件系统

回顾—虚拟内存知识



- 16 bits的操作系统其最大寻址空间? **2^{16}**
- 若虚拟内存的页内偏移量为8 bits, 则页大小? **$2^8 B$**
- 该系统虚拟内存最多有多少页? **$2^{(16-8)}$**
- 若考虑一级页表, 页表项(PTE)最多有多少项? **2^8**
- 若每个PTE具有的物理内存页占12 bits,
物理内存页一共有多少?
 2^{12}
- 物理内存最大多少?
 $2^{12} \times 2^8 B$
- 若每个PTE为16 bytes, 则一级页表占多少内存, 最少需要几个内存页, 若是2级页表呢?
 **$2^8 \times 2^4 B, 16\text{个};$
 $4+4+8, 17\text{个}, 17 \times 2^8$**

磁盘相关知识回顾



- 磁面, 磁臂, 磁头, 磁轴
- 磁道, 扇区 (引导扇区?)
- 读写磁盘需指定:
 - 磁道号
 - 磁头位置
 - 数据扇区号
 - 数据传输大小
 - 物理内存地址
- 读写磁盘开销 (延时)
 - Seek: 定位磁道 (ms)
 - Rotation: 定位扇区 (ms)
 - Transfer: 获得数据 (μ s)
- 磁盘调度算法 (最小化Seek)
 - FIFO
 - 最短Seek时间优先
 - 电梯算法

4

文件系统建立

建立目的

- 对于运行的进程而言, 内存仅能暂存数据, 如果希望结束后数据依然保存, 就需要保存在外部存储中 (磁盘)
- 为了方便管理磁盘上的数据, 引入了文件系统, 磁盘数据将以文件的形式进行存储。

相应要求

需要有严格的组织形式, 需要能够对应查询 (名字 \square 位置), 通常还需要有缓存, 需要便于管理 (层次结构), 需要对正在使用的文件进行维护。

核心组件: 文件 + 目录

文件

1. 元数据: 由操作系统添加的信息, 包括文件大小, 文件创建者, 文件修改时间, 访问权限等 (Inode, Index node)
2. 应用数据 (data): 用户产生的具体数据
3. 磁盘数据将以文件的形式进行存储
4. 文件创建时必须命名, 进程会根据文件名来访问文件内容
5. 文件名 = (编码名字) + 扩展名

目录

1. 绝对路径: 从根目录开始到文件
2. 相对路径: 从当前工作目录开始到文件
3. 将一个文件名映射到某个文件或目录为硬连接
4. 将一个文件名映射到另一个文件名为符号链接

文件和卷的关系

卷是指将一系列物理资源块形成的逻辑存储设备

挂载： 将某个卷与某个文件系统建立连接

其本质是将文件系统中的某个路径与被挂载的卷的根目录建立映射关系

对文件的基本操作

创建文件，写文件，读文件，删除文件，在文件中添加新内容

其本质是将文件名与磁盘块进行映射

代码举例

对文件的基本操作 (举例1)



```
#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <fcntl.h>
int main(int argc, char *argv[])
{
    int fd = -1; int ret = 1; int buffer = 1024; int num = 0;
    if((fd=open("./test", O_RDWR|O_CREAT|O_TRUNC)) == -1) {
        printf("Open Error\n"); exit(1);
    }
    ret = write(fd, &buffer, sizeof(int));
    if(ret < 0) {printf("write Error\n"); exit(1);}
    printf("write %d byte(s)\n", ret);
    lseek(fd, 0L, SEEK_SET);
    ret = read(fd, &num, sizeof(int));
    if(ret == -1) {printf("read Error\n"); exit(1);}
    printf("read %d byte(s), the number is %d\n", ret, num);
    close(fd); return 0;
}
```

12

对文件的基本操作 (举例2)



```
#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <fcntl.h>
#include <sys/types.h> #include <sys/stat.h> #include <dirent.h>
int main(int argc, char *argv[]){
    struct stat sb; DIR *dirp; struct dirent *direntp; char filename[128];
    if ((dirp = opendir("/root")) == NULL){
        printf("Open Directory Error%os\n"); exit(1); }
    while ((direntp = readdir(dirp)) != NULL){
        sprintf(filename, "/root/%os",
                direntp->d_name);
        if (lstat(filename, &sb) == -1){printf("lstat Error%os\n"); exit(1);}
        printf("name : %os, mode : %od, size : %od, user id : %od\n",
               direntp->d_name, sb.st_mode, sb.st_size, sb.st_uid); }
    closedir(dirp);
    return 0;
}
```

```

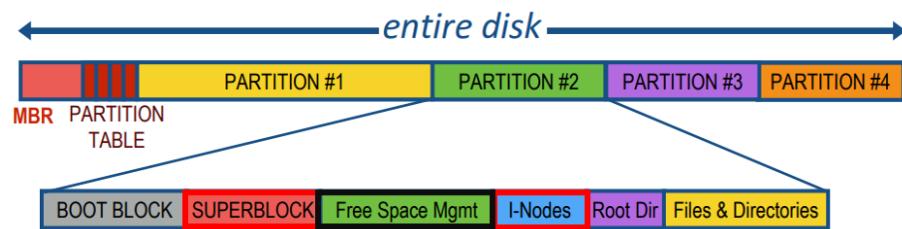
struct stat {
    dev_t st_dev;           /* ID of device containing file */
    ino_t st_ino;           /* Inode number */
    mode_t st_mode;         /* File type and mode */
    nlink_t st_nlink;       /* Number of hard links 硬链接和符号链接
                                的区别 */
    uid_t st_uid;           /* User ID of owner */
    gid_t st_gid;           /* Group ID of owner */
    dev_t st_rdev;          /* Device ID (if special file) */
    off_t st_size;           /* Total size, in bytes */
    blksize_t st_blksize;    /* Block size for filesystem I/O */
    blkcnt_t st_blocks;      /* Number of 512B blocks allocated */
    struct timespec st_atim; /* Time of last access */
    struct timespec st_mtim; /* Time of last modification */
    struct timespec st_ctim; /* Time of last status change */};
```

设计磁盘文件系统的挑战

1. 高性能
2. 普适性
3. 持久性
4. 可靠性

磁盘文件系统

磁盘文件系统存储在磁盘上，采用分区方式将磁盘进行划分，磁盘的0盘0面1扇区为引导扇区MBR(master boot record)，紧跟在后面的是磁盘分区表，每个分区的第一个块是启动块，由MBR在分区时填入，系统启动时执行MBR过程中，会执行每个分区的启动块。



建立索引：一般按照树形结构

空闲地址映射：快速分配机制 -> **bitmap**实现

局部性原理：相同文件的靠近内容防止在相近磁盘块

分配方式讨论

1. 连续分配方式

- 连续分配方式: 按创建顺序及大小依次占用磁盘块

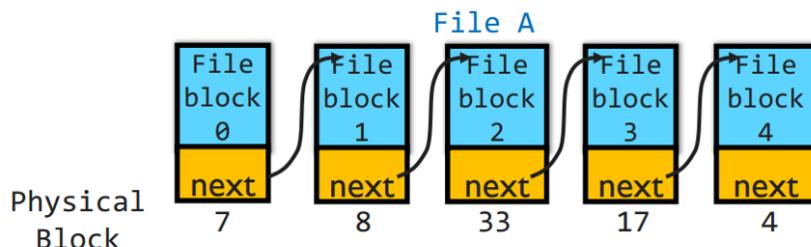
- ✓ 实现简单, 每个文件只需记录其初始和终止磁盘块
- ✓ 高效, 整个文件读写仅需要一次Seek操作
- ✗ 容易碎片化, 特别是当大量文件被移动或删除时
- ✗ 不适合文件大小频繁变化的情景 (编写代码, 文档)



2. 链表方式

- 链表方式:

- ✓ 实现简单, 文件的每个磁盘块只需维护指向下一个磁盘块的指针
- ✗ 性能不稳定, 特别是随机访问文件中的内容 (Seek频繁)
- ✗ 空间开销问题, 每个磁盘块均需要空间来保存指针

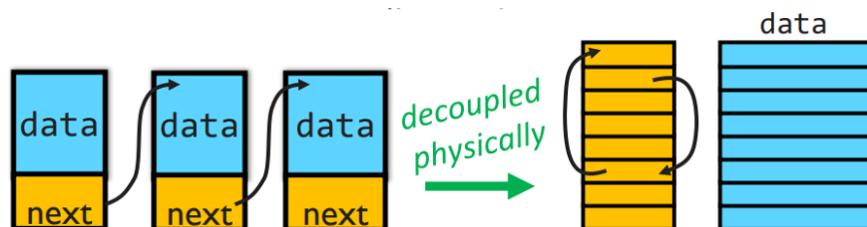


3. FAT文件系统 **File Allocation Table**

- 用在DOS和早期Windows系统 (70年代)

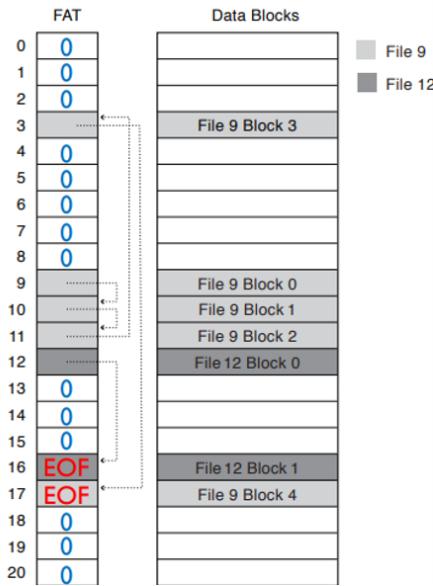
- 文件分配表

- 采用链表形式
- 将指针部分分离, 形成以<key, value, next>结构为表项的链表
- 需要为每个文件 (key) 指明其首个磁盘块 (value)



具体展示:

Directory	
bart.txt	9
maggie.txt	12



- 目录项描述了一个文件名和其首个磁盘块的关系
- 每个磁盘块对应一个FAT表项
- FAT表项为0，则没有使用
- FAT表项为EOF，代表的是某个文件最后的磁盘块

FAT优缺点

FAT文件系统的优缺点



- ✓ 实现简单
 - ✓ 较少的磁盘碎片
 - ✓ 目录和FAT表均在启动时加载到内存中，磁盘块只保存用户数据
 - ✗ 磁盘Seek可能耗时较高
 - ✗ 很难提供可靠性
 - ✗ 存储目录和FAT表内存开销可能较大
- e.g., 1TB (40位) 磁盘, 块大小为4KB (12位), 则需要表项为 $2^{(40-12)} = 2^{28}$, 若每个表项为4 bytes, 则需要 $2^{(28+2)} = 2^{30}$ B

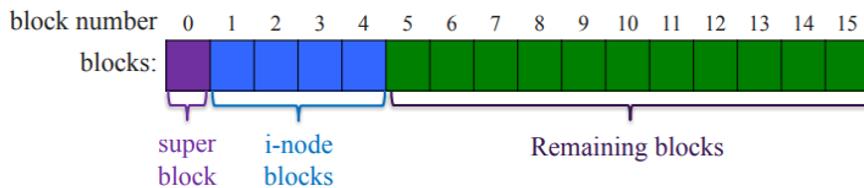
FFS文件系统 (Fast File System)

采用树形-多级索引方式

- 每个文件是非对称的树，根节点Inode，所有节点大小均固定
- 多级索引可以很好的支持大体积文件和小体积文件
- 新颖的空闲地址映射和局部性分配

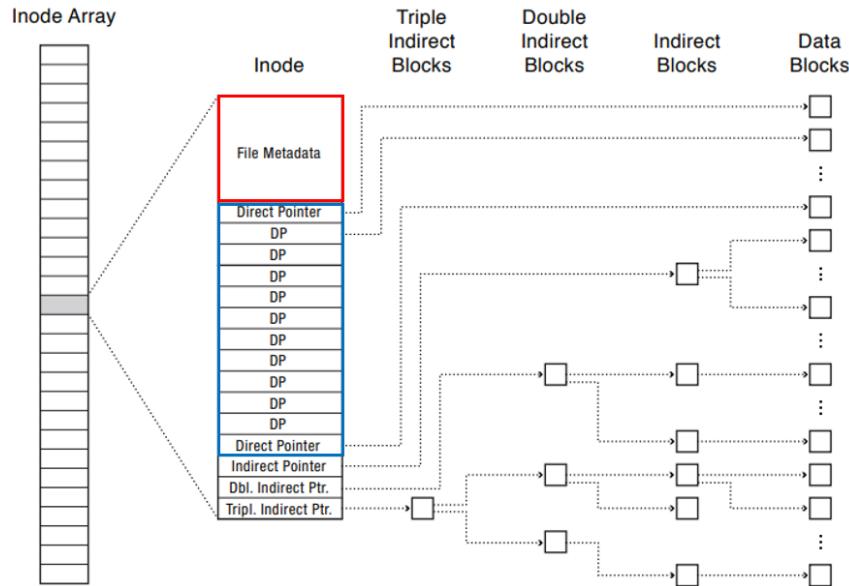
超级块：指明文件系统的关键参数

- 文件系统类型
- 块大小
- inode array 的起始位置和长度
- 空闲块的起始位置



1. 每一个**Inode**块是一个**Inode Array**
2. **Inode**数组中每个节点是**Inode**
3. **Inode**包含文件的元数据，12个数据块指针以及3个间接指针

Fast File System (FFS) 文件系统



FFS文件系统特点

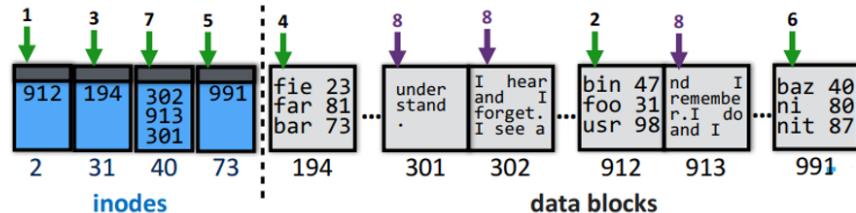
- 树形结构 → 很快地找到所需的磁盘块数据
- 局部性分配(多级指针) → 最小化Seek时间, 可以较好的支持连续读写
- 固定的结构 → 易于实现
- 非对称性
 - 数据磁盘块不在同一层 → 支持大体积文件
→ 小体积文件不会产生过多开销

FFS读文件操作

FFS读文件操作



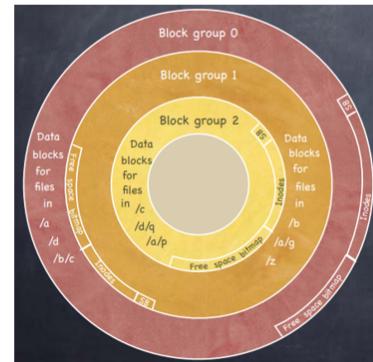
- 打开并读取文件/foo/bar/baz
- 根据根目录的inode=2, 找到其所在的磁盘块912 (1)
- 根据根目录地址912, 找到目录foo的inode=31 (2)
- 根据foo目录的inode=31, 找到其所在磁盘块194 (3-4)
- 以此类推直至找到991块中的bar文件inode=40 (5-7)
- 从inode=40找到该文件的内容 (8)



FFS空闲地址映射和局部性分配



- 引入bitmap, 每一位代表一个磁盘块
- 将磁盘分成块组, 每个块组
代表一系列相邻的磁道 (track)
- 均匀地将空闲bitmap和inode array
分发给这些块组, 并且每个块组均
具有一份超级块的拷贝
- 在当前目录下, 若创建一个新目录,
FFS会将当前目录视为新目录的
父目录并为其分配一个磁盘块
- 若创建文件, 则在当前磁盘块内找
一个空闲的inode, 根据bitmap将数据顺序放置在空闲数据块内



FFS的优缺点

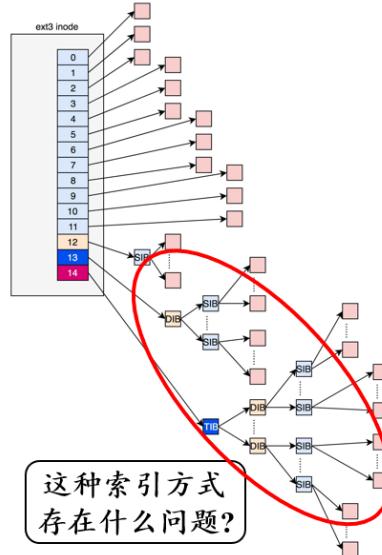


- ✓ 可以很好地支持大体积和小体积文件
- ✓ 实现了局部性放置
- ✓ 固定的字段格式使得实现较为简单
- ✗ 不太适合体积极小的文件 (tiny file), inode可能比有效
数据量还大, 空间利用率低
- ✗ 需要预留10%-20%磁盘空间给块组, 实现局部性放置

树形-多级索引方式 (FFS, 引入inode)



```
struct ext4_inode {
    __le16 i_mode;          /* File r/w/x, etc */
    __le16 i_uid;           /* Low 16 bits */
    __le32 i_size_lo;       /* File Size in bytes */
    __le32 i_atime;         /* Access time */
    __le32 i_ctime;         /* Inode Change time */
    __le32 i_mtime;         /* Modification time */
    __le32 i_dtime;         /* Deletion Time */
#define EXT4_NDIR_BLOCKS 12
#define EXT4_IND_BLOCK   EXT4_NDIR_BLOCKS
#define EXT4_DIND_BLOCK  (EXT4_NDIR_BLOCKS * EXT4_NDIR_BLOCKS)
#define EXT4_TIND_BLOCK  ((EXT4_NDIR_BLOCKS * EXT4_NDIR_BLOCKS) * EXT4_NDIR_BLOCKS)
#define EXT4_N_BLOCKS     (EXT4_NDIR_BLOCKS * EXT4_NDIR_BLOCKS * EXT4_NDIR_BLOCKS)
    __le32 i_block[EXT4_N_BLOCKS];
    __le32 i_generation;    /* File version */
    __le32 i_file_acl_lo;   /* File ACL low */
    __le32 i_size_high;     /* File size high */
    ....};
}
```



NTFS(ext4): 引入新概念 Extents

Extents可以用来存放连续的块，树形结构: header + entry

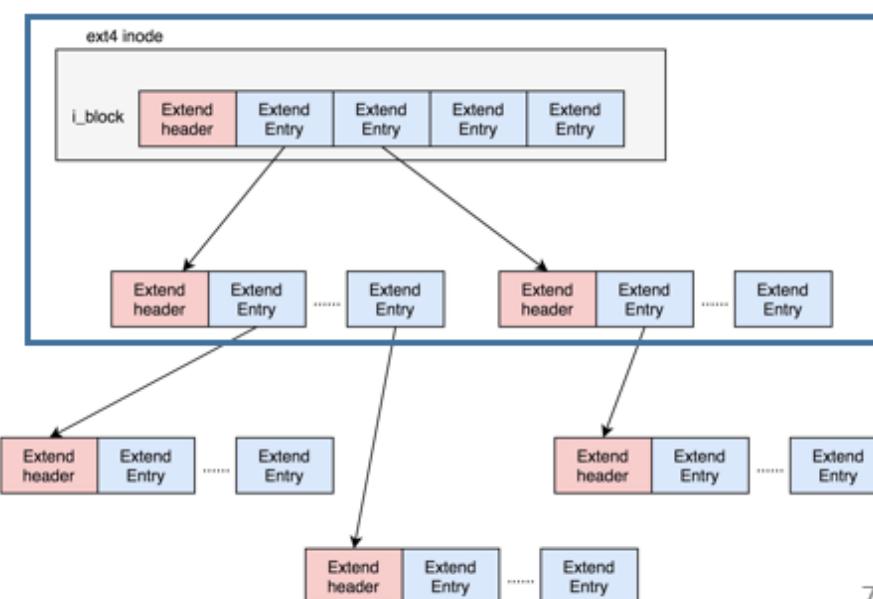
- 2层树可以表示多大的文件?

head: 12 B entry: 12 B

(4K-12)/12 = 340 个 entries

340 * 128M * 4 > 160G

具体结构为:



对于NTFS而言，其包含了这样几个部分：

超级块，块组描述符表，块位图块，inode位图块，inode列表，数据块

- 描述块组的数据结构(局部信息)
 - 块组的块位图, inode位图, inode列表, 数据块列表都有相应的成员变量
- 块组描述符表(全局信息)
- 超级块(整个文件系统的信息, 全局信息)
 - inode总数, 块总数, 每个块组的inode数目, 每个块组的数据块数目

全局信息很重要, 必须备份(产生冗余), 采用何种备份方式?

一般而言，采取的备份方式为：

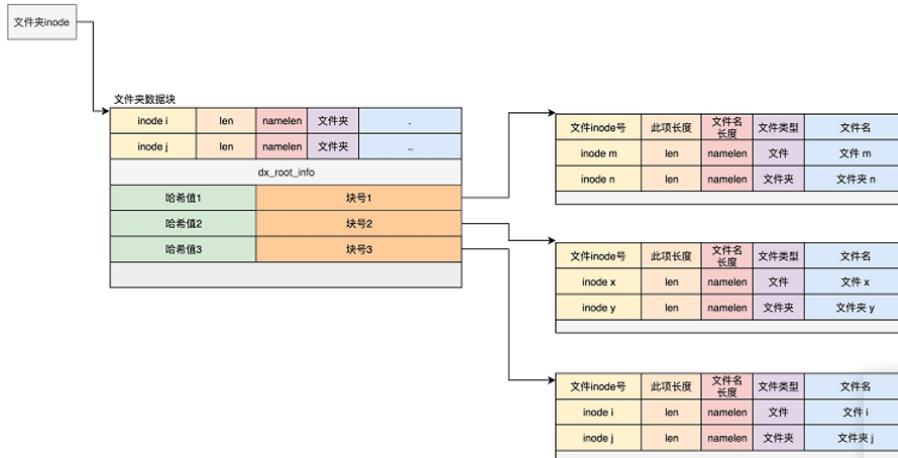
- 超级块(通常所占内存不多)
 - 每个块组都备份
 - sparse_super模式：副本只存在块组索引为0, 3, 5, 7的整数幂
- 块组描述符表(通常所占内存较多)
 - 每个块都备份(浪费空间, 限制文件系统大小)
 - sparse_super模式(限制文件系统大小)
 - 一个块组最大128MB, 若给定块组数目为N, 则文件系统大小为 $128N$
 - meta block group模式
 - 将块组进行再次分组(64个块组为一个元块组)
256个块组形成4个元块组, 一个元块组的描述符表最多64项
 - 每个块组备份自己的元块组描述符表, 一般采用3个备份
分别为第一块、第二块以及最后一块

10

NTFS (ext4) 目录存储格式



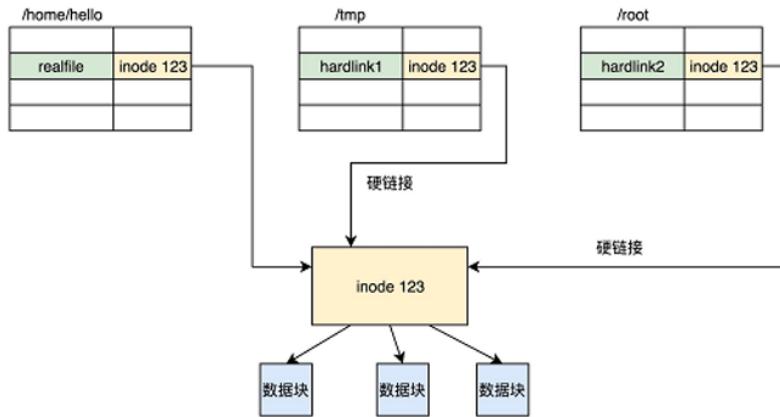
- 如果inode中iflags设置为EXT4_INDEX_FL
 - 目录的块组织形式将从链表形式变为Hashed Tree结构



硬链接和符号链接的存储格式

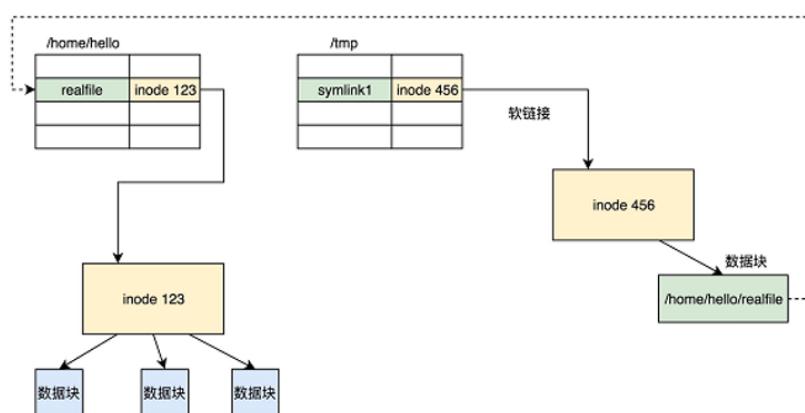
硬链接

与原始文件共用一个inode, inode不能跨文件系统 硬链接不能跨文件系统



符号链接

具有独立的inode，访问符号链接文件，其实是访问指向的另一个文件，可以跨文件系统。



虚拟文件系统

虚拟文件系统（Virtual FileSystem, VFS）：隐藏了各种硬件的具体细节，把文件系统操作和不同文件系统的具体实现细节分离开来，为所有的设备提供了统一的接口，VFS提供了多达数十种不同的文件系统。虚拟文件系统可以分为逻辑文件系统和设备驱动程序。逻辑文件系统指Linux所支持的文件系统，如ext2,fat等，设备驱动程序指为每一种硬件控制器所编写的设备驱动程序模块。

在 VFS 上面，是对诸如 open、close、read 和 write 之类的函数的一个通用 API 抽象。在 VFS 下面是文件系统抽象，它定义了上层函数的实现方式。它们是给定文件系统（超过 50 个）的插件。文件系统的源代码可以在 ./linux/fs 中找到。

文件系统层之下是缓冲区缓存，它为文件系统层提供了一个通用函数集（与具体文件系统无关）。这个缓存层通过将数据保留一段时间（或者随即预先读取数据以便在需要时可用）优化了对物理设备的访问。缓冲区缓存之下是设备驱动程序，它实现了特定物理设备的接口。

因此，用户和进程不需要知道文件所在的文件系统类型，而只需要象使用Ext2文件系统中的文件一样使用它们。

虚拟文件系统时磁盘文件系统的镜像，其实质上是将整个文件系统用一种虚拟化的过程进行了逻辑结构上的统一。

对于虚拟文件系统而言，首先需要进行挂载，即OS想使用磁盘文件系统必须在内核中注册过，将各分区通过挂载到目录来访问实际的磁盘分区。

对于挂载而言，其实际上是进行了mount系统调用

进程间通信

进程通信的需求

数据传输

一个进程需要将它的数据发送给另一个进程（Web服务，进程查询数据库）

事件通知

一个进程需要向另一个或一组进程发送消息，通知它们发生了某种事件（进程终止时需要通知父进程）

资源共享

多个进程之间共享同类资源（互斥和共享）

进程控制

有些进程希望完全控制另一个进程的执行，即能够拦截另一个进程的陷入和异常，并能够及时知道另一个进程的状态改变

进程如何通信

管道模型

管道的方式是将上一个进程的输出信息作为下一个进程的输入信息，在shell中常用。管道分为命名管道和匿名管道，其在linux终端中都有相应的例子。

- 命名管道 mkfifo pipe_name 以内存文件形式出现，例如 echo "hello world" > pipe_name 表示将'hello world'这一字符串写入到pipe_name 这一管道中（这里管道是有名称的）
- 匿名管道 cmd1 argv1 | cmd2 argv2 | cmd3 argv3 表明从1->2->3信息输入和输出。

采用管道模型实现IPC（进程通信）：通信频率高，通信量大（缓冲区有限）

匿名管道：

建立匿名管道的函数原型

```
int pipe(int fd[2]); //fd[0]表示读取端进程，fd[1]表示写入端进程

//pipe对应的系统调用
SYSCALL_DEFINE2(pipe2,int __user * ,fildes,int flags)
{
    struct file *file[2];int fd[2];int error;
    error = __do_pipe_flags(fd,files,flags);
    if(!error){
        if(unlikely(copy_to_user(fildes,fd,sizeof(fd))))
        {
            error = -EFAULT;
        }
        else{
            fd_install(fd[0],files[0]);
            //将文件描述符与file结构关联
            fd_install(fd[1],files[1]);
        }
    }
    return error;
}

static int __do_pipe_flags(int *fd,struct file **files,int flags)
{
    int error;
    int fdw,fdr;
    error = create_pipe_files(files,flags);
    //创建了管道以及2个file结构(file 和pipe缓冲区建立联系)
    error = get_unused_fd_flags(flags);
    fdr = error;
    error = get_unused_fd_flags(flags);
    //获得了两个未使用的文件描述符
    fdw = error;
    fd[0] = fdr;//read file source
    fd[1] = fdw;//write file destination
    return 0;
}

const struct file_operations pipefifo_fops = {
    .open = fifo_open,
    .llseek = no_llseek,
    .read_iter = pipe_read,
    .write_iter = pipe_write,
    .poll = pipe_poll,
    .unlocked_ioctl = pipe_ioctl,
    .release = pipe_release,
```

```

    .fasync = pipe_fasync
};

static struct file_system_type pipe_fs_type{
    .name = "pipefs", .mount = pipefs_mount, .kill_sb =
kill_anon_super
};

static int __init init_pipe_fs(void)
{
    int err = register_filesystem(&pipe_fs_type);
    if(!err){
        pipe_mnt = kern_mount(&pipe_fs_type);
    }
}

static struct inode * get_pipe_inode(void)
{
    struct inode *inode = new_inode_pseudo(pipe_mnt-
>mnt_sb);
    struct pipe_inode_info *pipe;
    inode->i_ino = get_next_ino();
    pipe = alloc_pipe_info();

    pipe->readers = pipe->writes = 1;
    inode->i_fop = &pipefifo_fops;
    return inode;
}

```

问题：在进行fork时，创建的子线程会完全复制父进程的数据结构，即fd[2]也会被复制，这样父子进程就可以通过fd操作匿名管道进行相互通信。为了避免混乱，可以让一个进程关闭读fd，另一个关闭写fd

具体流程：

1. shell 首先创建子进程A，建立管道，其中shell保留读，进程A保留写
2. shell 然后创建子进程B，这时shell保存读的fd也被复制到了进程B中，即shell和B都可以读管道
3. shell 主动关闭读取端fd，这时就形成了A写B读的形式
4. 最后将管道的两端和输入输出关联起来 dup2(fd[0],STDIN_FILENO)
dup2(fd[1],STDOUT_FILENO)

命名管道

命名管道在shell中采用mkfifo 命令进行创建，若使用代码则会通过Glibc提供的库函数，其本质最终还是会创建inode -> path(dentry) -> file -> fd

消息队列模型

该模型类比于邮件，需要对消息格式进行定义，进程间发送消息采用固定内存大小的数据单元（消息体）

消息队列的程式化操作：

- 使用msgget()创建消息队列，该函数需要一个重要参数**key**，用来标识消息队列的唯一性，实现方式是ftok(inode)->key，即指定一个文件，用其inode来生成key，或者采用IPC_PRIVATE这一私有key
- 使用msgsnd()发送消息，该函数第一个参数为**key**，第二个是消息体，第三个是消息的长度（消息体只规定了最大长度），最后是一些标志位
- 使用msgrcv()接收信息，该函数第一个参数为**key**，第二个是消息体，第三个是可接受的最大长度，第四个是消息类型，最后是一些标志位

信号

OS内核与进程之间的通信模式

- 信号触发的动机：
 - 内核发现一个系统事件，例如除0错误或者子进程退出
 - 一个进程调用kill命令，请求OS发送信号给目标进程，例如调试、挂起、恢复或超时等等。
- 信号接收的机制（进程收到内核发送的信号）
 - 忽略信号
 - 终止本进程
 - 用一个定义的用户态信号处理函数去捕捉信号

共享内存模型（同步和互斥）

- 管道模型、消息队列模型以及信号都是端到端且数据传输量较小的IPC，有些庆幸需要多进程写作，且数据交换量较大 -> 共享内存模型
- 每个进程有独立的虚拟访问空间，映射到不同的物理内存中，即不同进程访问同一虚拟内存，本质访问的物理内存不同
- 如果每个进程拿出一块虚拟地址，映射到相同的物理地址，这样一个进程写入的东西，另一个进程就可以读出，不需要发送多个消息

进程可以调用shmget()创建共享内存，在System V模式下，创建IPC对象都是xxget()

该函数的第一个参数是**key**，用于唯一确定共享内存对象，第二个参数是共享内存的大小

若一个进程想访问该共享内存，需要调用shmat()将其加载到自己的虚拟地址中，该函数第一个参数为共享内存对象的**key**，第二个参数是加载到的地址（通常为NULL，让内核去加载）

采用共享内存的进程就类比于进程内的多线程，需要同步和互斥机制（信号量）

共享内存模式（信号量）！！！

P操作 -> 申请资源操作

V操作 -> 归还资源操作

进程可以通过semget()来创建一组信号量，该函数第一个参数是信号量组的key（唯一标识），第二个参数是该信号量组的信号量数目（每种共享资源均应有多少个信号量）。初始化：利用semctl()来初始化信号量的资源数目

```
int semctl(int semid 信号量组,int semnum 信号量在组中位置, int cmd,union semun args);

union semun{int val;struct semid_ds *buf;
            unsigned short int *array;struct seminfo
            *__buf};
```

信号量的PV操作在进程中统一为semop()

```
int semop (int semid, struct sembuf semoparray[],size_t numops);

struct sembuf{
    short sem_num;//信号量组中对应的序号
    short sem_op;//信号量值在一次操作中的改变量
    short sem_flg;
}
```

sem_op 负值代表请求sem_op绝对值的资源量，(P操作)，若sem_op的绝对值小于sem_num对应的semnum对象中的val值， $val = val + sem_op$ ；否则进程将被挂起直至足够的资源。是正值则表示归还相应的资源量（V操作）， $val = val + sem_op$ ，并唤醒所有等待此信号量的进程。

作业：用上述API实现生产者--消费者问题

```
struct shm_data {int data[256];int dataLength;};
//定义了共享内存大小

union semnu{int val;struct semid_ds *buf;
            unsigned short int*array; struct
            seminfo *__buf;};
//信号量对应的共享资源大小

int get_shmid();//创建一个共享内存对象，大小为256。返回信号量组
int get_semaphoreid();//创建信号量组对象，并获得其id
int semaphore_init(int semid){};//初始化操作，设置信号量对应总
                                //资源
int semaphore_p(int semid){};//信号量中的P操作
int semaphore_v(int semid){};//信号量中的V操作

int shmid = get_shmid();//创建共享内存
```

```

int semid = get_semaphoreid(); //创建信号量组对象
int buf = semaphore_init(semid); //获得信号量资源数

Producer(semid) //生产者进程
{
    while(1)
    {
        produce an item in nextp; //生产相应的资源
        semaphore_v(semid); //将生产的资源放入共享内存中
    }
}

Consumer(semid)
{
    while(1)
    {
        semaphore_p(semid); //进行P操作，消耗共享内存中的资源
        consume an item from sharememory; //消费取出的相应资源
    }
}

```

socket套接字（请查看计网）

总结：

- (单机) 进程间通信通常分为：
 - 管道模型（匿名，命名）
 - 消息队列模型
 - 信号
 - 共享内存
- (多机) 进程间通信
 - socket机制
 - RPC机制

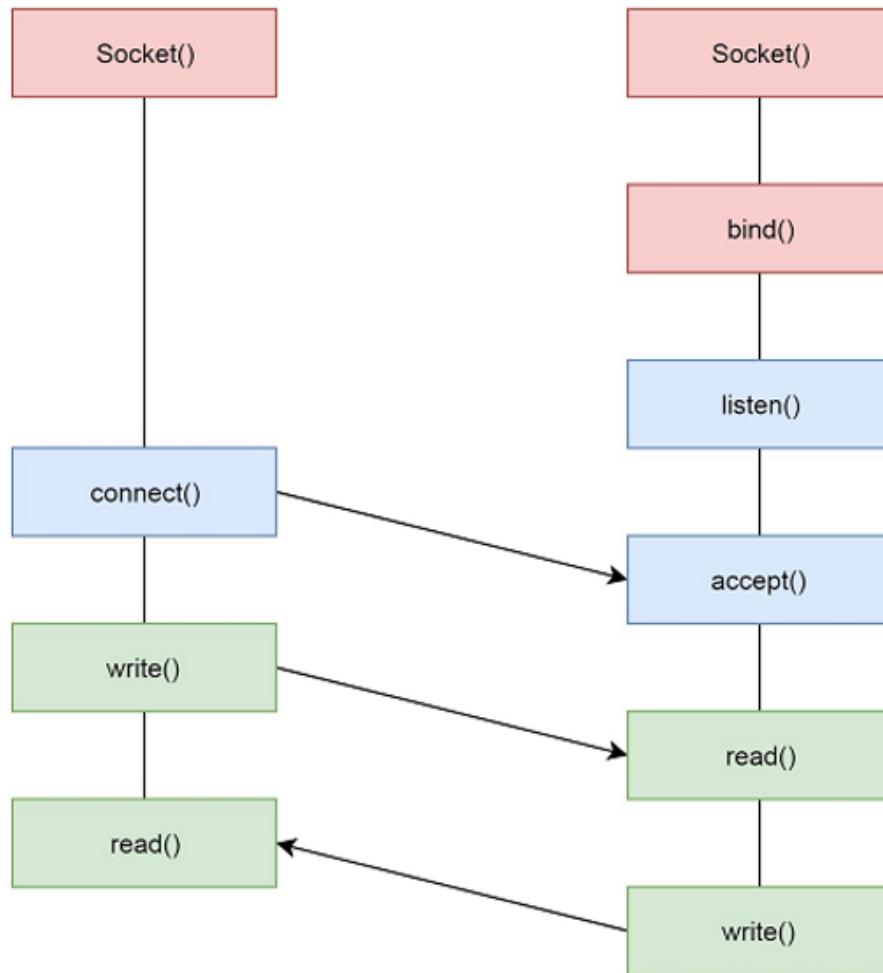
计算机网络系统

回顾：同一设备中的进程间通信：

- 信号： 进程 → OS内核 → 进程
- 管道： 进程 → 内存文件系统（*pipe*对象） → 进程
- 共享内存： 进程 → 映射到相同的物理内存 → 进程
- 消息队列： 进程 → 内存文件系统（消息队列对象） → 进程

但是对于不同设备的进程间通信，显然，只有消息队列是可能的。

Socket 编程



整体的socket编程的结构如图所示，其中connect() 和 accept()是传输过程中的相应连接，而实际上在下层中，有相应的read() 和 write()在两端进行读写

```
int socket(int domain, int type, int protocol);
//用于创建一个socket 文件描述符，即用户态和内核态接口
//socket 是在网络的应用层和传输层之间的一个接口，在操作系统上，即
//应用层运行在用户态，而传输层运行在内核态（操作系统中）

//参数domain 表示使用什么IP层协议，例如 AF_INET -> IPv4
//AF_INET6表示IPv6

//参数type 表示socket 类型， SOCK_STREAM 表示TCP,
//SOCK_DGRAM 表示UDP

//参数prototype 表示协议
//IPPROTO_TCP  IPPROTO_UDP
int serv_sock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)
```

```

SYSCALL_DEFINE3(socket, int family,int type,int protocol)
{
    int retval;
    struct socket *sock; int flags; .....
    if (SOCK_NONBLOCK != O_NONBLOCK &&
        (flags & SOCK_NONBLOCK))
        flags = (flags & ~SOCK_NONBLOCK) | O_NONBLOCK;
    retval = sock_create(family, type, protocol, &sock);
    .....
    retval = sock_map_fd(sock, flags & (O_CLOEXEC |
O_NONBLOCK));.....
    return retval;
}

```

socket_create 获得一个socket *sock 对象， sock_map_fd 创建了sock关联的文件描述符retval， 用户态操作retval， 其本质是操作内核态的sock

socket 函数会关联serv_sock 文件描述符和inet_stream_ops 结构， 后续调用bind , listen 其本质是调用inet_bind,inet_listen， 指明IP协议创建socket的函数为inet_create()

```

int bind(int sockfd, const struct sockaddr *addr,socklen_t
addrlen)
{
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr)); //每个字节都
用0填充
    serv_addr.sin_family = AF_INET;           //使用
IPv4地址
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1"); ///
具体IP地址
    serv_addr.sin_port = htons(1234);          //端
口号
    bind(serv_sock, (struct sockaddr*)&serv_addr,
sizeof(serv_addr));
}
struct in_addr{__be32 s_addr;};

```

TCP/IP协议规定网络字节顺序采用大端模式进行编址， 计算机设备的采用哪种模式由CPU处理器决定(Intel 采用小端模式)

bind函数(服务器端)

```

SYSCALL_DEFINE3(bind, int, fd, struct sockaddr __user *,
umyaddr, int, addrlen){
    struct socket *sock; struct sockaddr_storage address;

    int err, fput_needed;
    sock = sockfd_lookup_light(fd, &err, &fput_needed);
    if (sock) {

```

```

        err = move_addr_to_kernel(umyaddr, addrlen,
&address);
        if (err >= 0)
            err = sock->ops->bind(sock, (struct sockaddr
*)&address, addrlen);
            fput_light(sock->file, fput_needed); //说明socket和
文件系统有关联
    }
    return err;
}

int inet_bind(struct socket *sock, struct sockaddr *uaddr,
int addr_len){
    struct sockaddr_in *addr = (struct sockaddr_in *)uaddr;

    struct sock *sk = sock->sk; //下面是内核网络协议栈的操作
(tcp_prot)
    struct inet_sock *inet = inet_sk(sk); //强制类型转换，后
面会分析
    struct net *net = sock_net(sk);
    unsigned short snum;..... snum = ntohs(addr-
>sin_port);.....//网络 □ 主机
    inet->inet_rcv_saddr = inet->inet_saddr = addr-
>sin_addr.s_addr;
    if ((snum || !inet->bind_address_no_port) && //检测端口
是否冲突（占用）
        sk->sk_prot-
>get_port(sk, snum)) {..... }
    inet->inet_sport = htons(inet->inet_num); //主机 □ 网络
    inet->inet_daddr = 0;   inet->inet_dport = 0;
    sk_dst_reset(sk);}

```

- sockfd_lookup_light**函数完成了从文件描述符到**socket *sock**

- move_addr_to_kernel**函数完成了将**sockaddr**从用户态拷贝到内核态

- 通过**sock**调用**bind**,其本质是调用了**inet_stream_ops**结构中声明的**inet_bind**函数

listen函数(服务器端)

```

int listen(int sockfd, int backlog)
//等待客户端的连接 (e.g., listen(serv_sock, 20);)
//参数backlog表示最多同时连接的数目
SYSCALL_DEFINE2(listen, int, fd, int, backlog){
    struct socket *sock;  int err, fput_needed;  int
somaxconn;
    sock = sockfd_lookup_light(fd, &err, &fput_needed);
    if (sock) {

```

```

        somaxconn = sock_net(sock->sk)-
>core.sysctl_somaxconn; //系统上限
        if ((unsigned int)backlog > somaxconn) backlog =
somaxconn;
        err = sock->ops->listen(sock, backlog); //调用
inet_listen()
        fput_light(sock->file, fput_needed);
}
return err;

int inet_listen(struct socket *sock, int backlog){
    struct sock *sk = sock->sk; //进入内核网络栈
    unsigned char old_state; int err;
    old_state = sk->sk_state;
    if (old_state != TCP_LISTEN) err =
inet_csk_listen_start(sk, backlog);
    sk->sk_max_ack_backlog = backlog;
}

```

如果当前socket不处于TCP监听状态，则调用函数进入监听状态，否则，直接更新允许的客户端最大连接数

```

int inet_csk_listen_start(struct sock *sk, int backlog){
    struct inet_connection_sock *icsk = inet_csk(sk);
    struct inet_sock *inet = inet_sk(sk); int err = -
EADDRINUSE;
    reqsk_queue_alloc(&icsk->icsk_accept_queue);
    sk->sk_max_ack_backlog = backlog;
    sk->sk_ack_backlog = 0;
    inet_csk_dealloc_init(sk);
    sk_state_store(sk, TCP_LISTEN);
    ..... //对icsk和inet的操作
}
/*
inet_connection_sock涉及非常多的内容（流量控制和拥塞控制），需要时
会进一步分析，icsk->icsk_accept_queue队列将保存已经建立完3次握手
的连接（established），listen函数会为其分配内存空间，最后listen函
函数会设置socket的状态为TCP_LISTEN
*/

```

inet_csk函数的作用是将**sock**指针强制转换成**inet_connection_sock**指针

其本质是：**tcp_sock**结构第一个成员为**inet_connection_sock**对象，

inet_connection_sock结构第一个成员为对象

inet_sock结构第一个成员为**sock**对象，而初始化套接字时

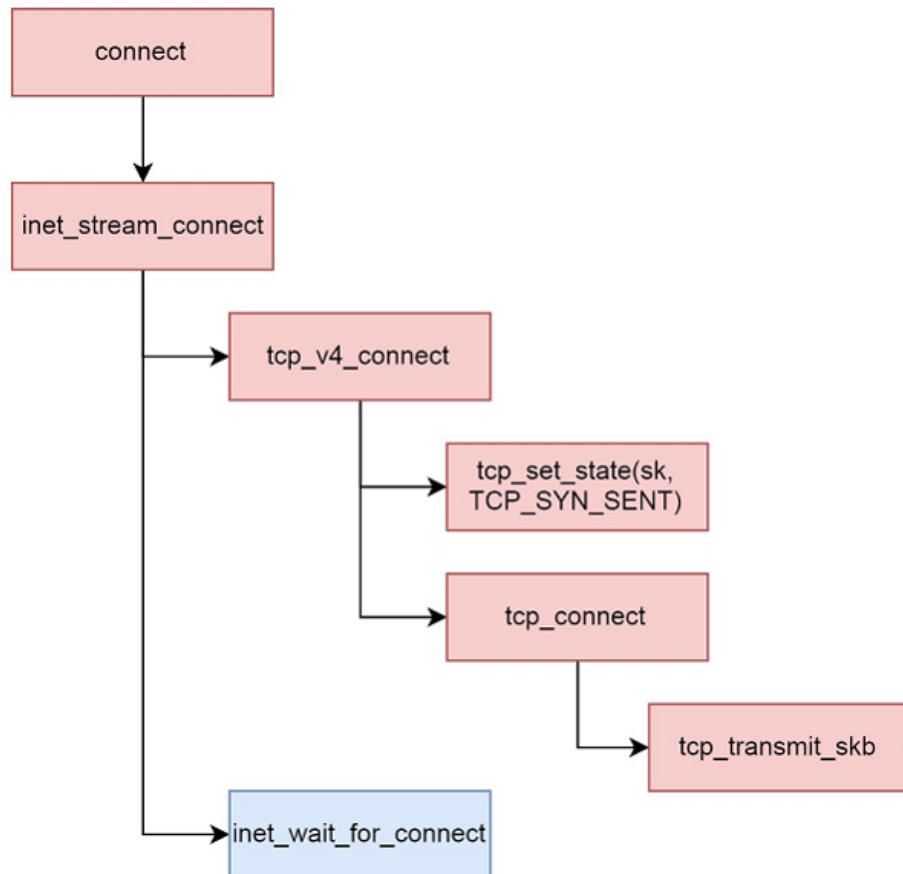
sock *sk其实指向的是**tcp_sock**对象 (**TCP**套接字)

```
struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr)); //每个字节都用0填充
serv_addr.sin_family = AF_INET; //使用IPv4
serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1"); //服务器端IP
serv_addr.sin_port = htons(1234); //服务器端口号
connect(sock, (struct sockaddr*)&serv_addr,
sizeof(serv_addr));

int connect(int sockfd, const struct sockaddr *addr,
socklen_t addrlen)
//向服务器端发起连接 (TCP三次握手!!)
//参数sockfd是客户端建立的socket对象
//参数addr包含了服务器端的IP地址和端口号
//参数addrlen表示sockaddr结构的大小
```

第一次握手

```
switch (sock->state) {
    .....
    case SS_UNCONNECTED: //只有处于未连接状态才会建立连接
        err = sk->sk_prot->connect(sk, uaddr, addr_len);
        sock->state = SS_CONNECTING; //上面connect完成不代表连接完成
        break;
}
//connect 调用 inet_stream_connect 函数，并根据此实现操作
```

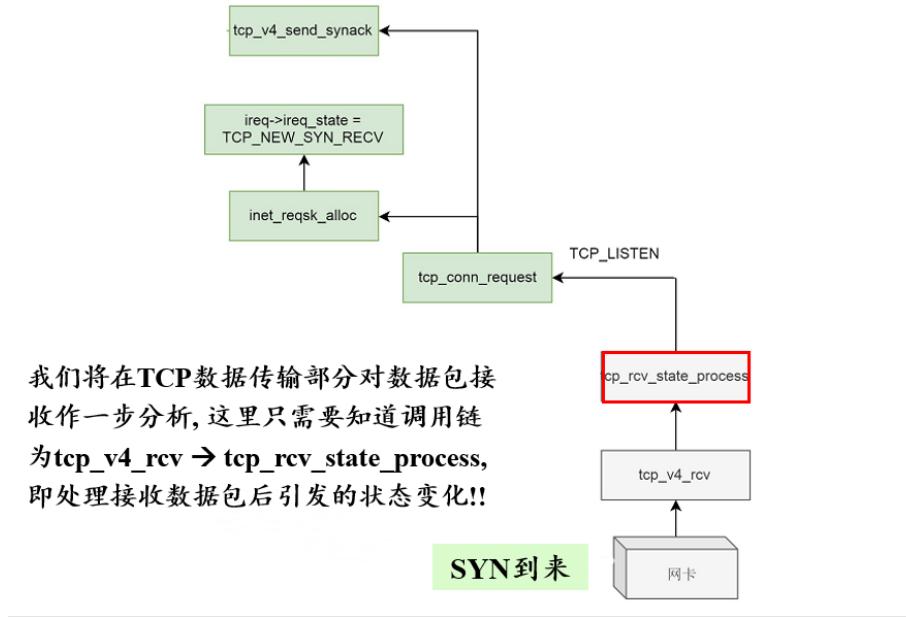


任务1—完整的源/目的IP地址以及源/目的端口号

1. 这里目的IP地址和端口号已由用户态connect给出
2. 源端口号可以使用一个随机的未使用端口号
3. 源IP地址(每个主机有多个IP) \leftarrow ip_route_connect()
即源IP地址的选择,是与路由相关的
4. ip_route_connect() 包含 flow 对象 <源/目的IP和端口, 协议>
根据 __ip_route_output_key(rp, &fl) 在路由表 rp 中查询是否已有路由, 若否, 则调用 ip_route_output_flow(rp, &fl, sk, 0)
寻找一个“合适”的源IP地址(e.g., 有线网络), 并写入 rp

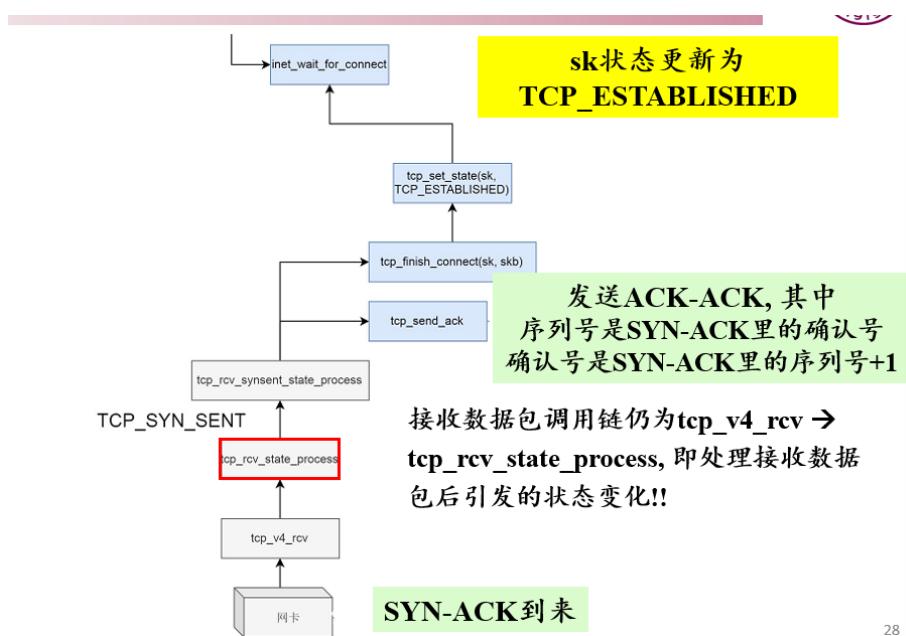
任务2即是发送SYN报文

第二次握手



实际上即是根据SYN的到来，更新sk状态，然后发送SYN-ACK包进行第二次握手

第三次握手

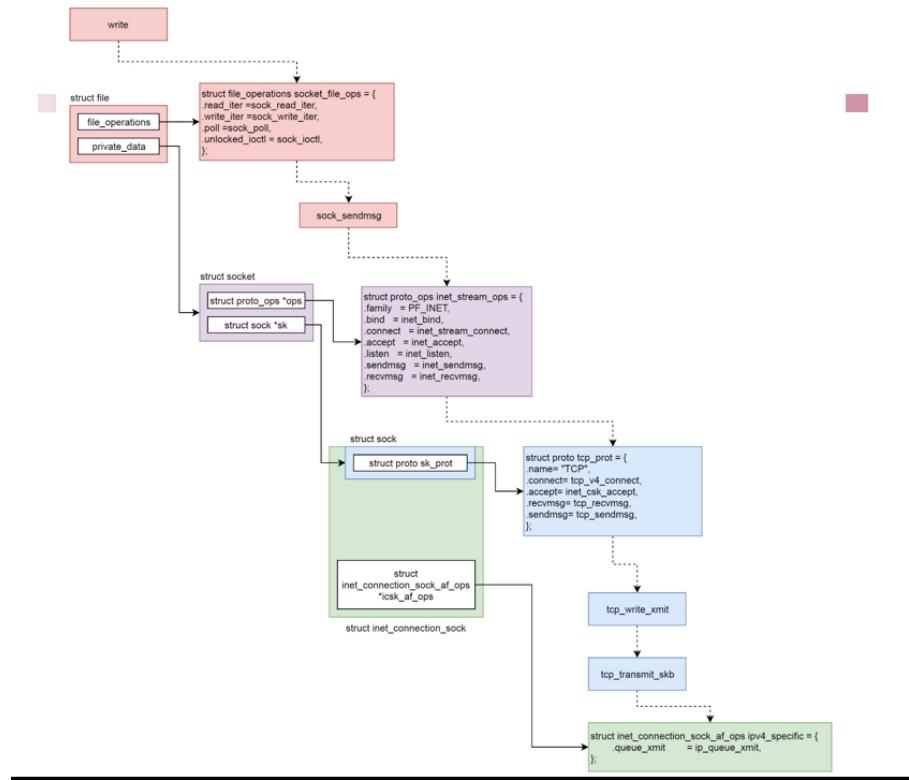


28

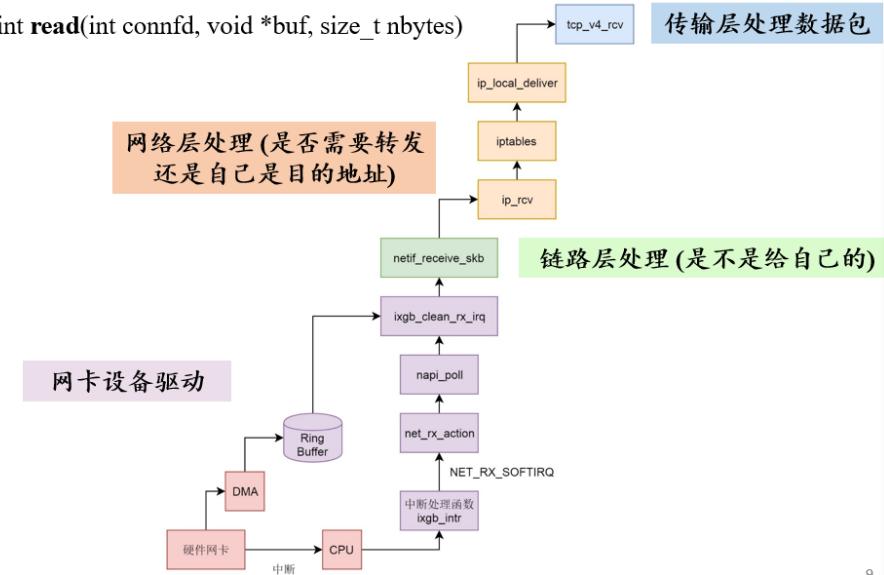
总结

- Socket编程是一切网络、web、流媒体服务的基础
这节课以TCP协议为例，简单讲解了以下函数
 - socket
 - bind
 - listen ← connect
 - accept

write函数



read函数



9

同步互斥与信号量

临界区

临界区是指用户互斥共享的资源。在进入临界区进行控制之前，需要先有一个进入区，进入区检查可否进入临界区（这一段代码），如果可以进入，则设置相应“正在访问临界区”标志。退出区，清除标志。剩余区，剩下的。

临界区的访问规则

- 空闲则入
没有进程在临界区时，任何进程可以进入
- 忙则等待
有进程在临界区时，其它进程均不能进入临界区
- 有限等待
等待进入临界区的进程不能无限期等待
- 让权等待（可选）
不能进入临界区的进程，应释放CPU（如转换到阻塞状态）

临界区的实现方法

有不同的实现方式：一般而言，对其的比较是通过性能开始的，主要是看其并发级别。

禁用中断

没有中断，没有上下文切换，因此没有并发。硬件将中断处理延迟到中断被启动之后，现代计算机体系结构都提供指令来实现禁用中断。

进入临界区，则禁止所有中断的发生，并保存相应的标志，退出临界区，则是使能所有中断，并恢复标志。

缺点：禁用中断后，进程无法被停止，整个系统都会为此停下来，可能导致其他进程处于饥饿状态。

软件方法

peterson算法：

设置共享变量

```
int turn; //当前谁进入临界区
bool flag[]; //表示进程是否准备好进入临界区

//进入区代码
flag[i] = true;
turn = j;
while(flag[j] && turn == j);
```

Dekkers算法

```
flag[0] = false; flag[1] = false; turn = 0;
do
{
    flag[i] = true;
    while(flag[j] == true)
```

```

{
    if( turn != i)
    {
        flag[i] = false;
        while turn != i;
        flag[i] = true;
    }
}
CRITICAL SECTION;
turn = j;
flag[i] = false;

}while(true);

```

特点：复杂，忙等待

更高级的抽象方法

硬件提供了一些同步原语：

锁

一个二进制变量（锁定/解锁）

原子操作指令：现代CPU体系结构都提供一些特殊的原子操作指令

测试和置位指令，交换指令

实现自旋锁：利用TS指令，特点：线程在等待的时候消耗CPU时间

同样，利用交换指令也可以实现类似的功能

优点：适用于单处理器或者共享内存的多处理器中任意数量的进程同步，简单并且容易证明，支持多临界区

缺点：

忙等待消耗处理器时间

可能导致饥饿：进程离开临界区时有多个等待进程的情况

死锁:拥有临界区的低优先级进程，请求访问临界区的高优先级进程获得处理器并等待临界区

信号量

信号量是操作系统提供的一种协调共享资源访问的方法

软件同步是平等线程间的一种同步协商禁止

OS是管理者，地位高于进程

用信号量表示系统资源的数量

信号量概念(semaphore)

信号是一种抽象数据类型：有一个整型(sem)变量和两个原子操作组成

P操作

sem - 1, 如果sem < 0, 进入等待, 否则继续。

V操作

sem + 1 ,如果sem<=0,唤醒一个等待进程

信号量是被保护的整型数量，初始化完成后，只能通过PV操作完成，操作系统保证PV是原子操作。

P()可能阻塞，V()不会阻塞

管程

将PV操作配对到一起，并发程序的编程方法

管程是一种多线程互斥访问共享资源的程序结构，面向对象方法，简化了线程间的同步控制。任一时刻最多只有一个线程执行管程代码。正在管程中的线程可以临时放弃管程的互斥访问，等待事件出现时恢复。

管程的使用

在对象/模块中，收集相关共享数据，定义访问共享数据的方法

条件变量

条件变量是管程内的等待机制，进入管程的线程因资源被占用而进入等待状态，每个条件变量表示一种等待原因，对应一个等待队列。

Wait操作

将自己阻塞在等待队列中，唤醒一个等待者或释放管程的互斥访问

Signal操作

将等待队列中的一个线程唤醒，如果等待队列为空，则等同空操作。

管程条件变量的释放处理方式

Hansen管程

正在执行的管程优先

Hoare管程

正在等待的管程优先

哲学家就餐问题

不可行方案

```
#define N 5
semaphore fork[5];
void philosopher(int i)
{
    while(true)
    {
        think();
        P(fork[i]);
        P(fork[(i+1)%N]);
        eat();
        V(fork[i]);
        V(fork[(i+1)%N]);
    }
}
```

改进方案

```
#define N 5
semaphore fork[5];
void philosopher(int i)
{
    while(true)
    {
        think();
        if(i%2)
        {
            P(fork[i]);
            P(fork[(i+1)%N]);
        }
        else
        {
            P(fork[(i+1)%N]);
            P(fork[i]);
        }
        eat();
        V(fork[i]);
        V(fork[(i+1)%N]);
    }
}
```

```
    }  
}
```

读者写者问题

```
P(writeMutex);  
write;  
V(writeMutex);  
  
//reader  
P(countMutex);  
if(Rcount++==0)  
    P(writeMutex);  
V(countMutex);  
  
read;  
  
P(countMutex);  
if(--Rcount == 0)  
    V(writeMutex);  
V(countMutex);
```