文件系统的设计与实现

## 1.1 设计目的与意义

通过模拟文件系统的实现，深入理解操作系统中文件系统的理论知识, 加深对教材中的重要算法的理解。同时通过编程实现这些算法,更好地掌握操作系统的原理及实现方法,提高综合运用各专业课知识的能力。

本文件系统采用多级目录,其中第一级对应于用户账号,第二级对应于用户账号下的文件，第三级对应文件夹下的文件，以此类推。另外,为了简单本文件系统未考虑文件共享、文件系统安全以及管道文件与设备文件等特殊内容。

## 1.2 设计内容

① 用户登录

用户分为超级管理员、管理员和普通用户。超级管理员具有创建用户并设置权限的权限；理员与超级管理员权限基本相同。

② 建立文件

模拟文件系统进行“建立文件”的处理。

③ 打开文件

约定操作类型与文件属性不符合和正处在“建立”状态的文件不允许打开。

④ 关闭文件

对某个文件不需要再读或写的时候，用户应关闭该文件。

⑤ 读文件

文件打开后可顺序读文件中的记录，读长度表示本次操作需读的记录个数。

⑥ 写文件

执行写文件操作时要区分两种情况，第一种是在执行create后要求写，第二种是执行open后要求写。对第二种情况可认为用户要对一个已建立好的文件进行修改。一个文件可以分多次写，既可按记录顺序写也可随机写一个记录，采用顺序写时可省略记录号。

⑦ 删除文件

把请求撤消的文件从拥护文件目录表中除名，收回该文件站用的存储区域。

⑧ 文件列表

将目前本用户本路径的文件列出，并列出相关的属性。

1.3开发平台：

CodeBlocks17.01+Win10 编译器：MinGW

1.4设计思路

本程序文件部分由一个儿子-兄弟链表组成，节点名称为Node，其成员有：

Node\* parent;//父节点

Node\* headSon;//第一个子节点

Node\* sibiling;//下一个兄弟节点

string name;//文件或文件夹名字

vector<string> content;//文件中的内容

int permission;//可读权限的大小

int type;//0是文件夹，1是文件

里面的方法有：

CreateFolder()创建文件夹

createFile()创建文件

addContent()添加内容

openFile()打开并读取文件

insertFile()在文件中某一项插入记录

isExist（）判断本级目录下是否存在某一项文件

listAll（）把本级目录下的所有文件都列出

enter()进入某一个文件夹

goBack()返回上级文件架

deleteByName()按照名字删除某一项

~Node()析构函数

用户管理上，由vector<user>结构实现，user结构下有三个字段：

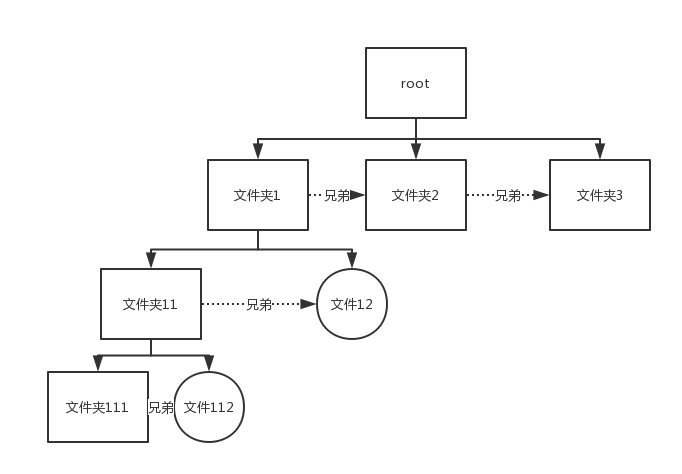
String username//用户名

String password//密码

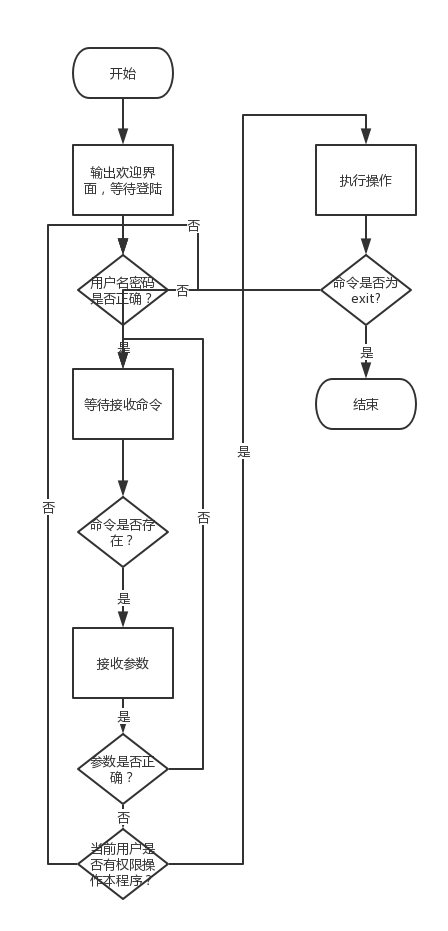
int level;//权限

vector数组里存放着所有用户的信息。

文件系统的逻辑结构为



1.5流程图

1.6 关键代码分析

创建文件或文件夹

bool createFolder(const string& newName,const int level) {

Node\* exist=this->isExist(newName);

if (exist) {

printf("文件或文件夹已存在\n");

return false;

}

Node\* folder=new Node();

folder->name=newName;

Node\* tmp=this->headSon;

this->headSon=folder;

folder->sibiling=tmp;

folder->parent=this;

folder->permission=level;

return true;

}

bool createFile(const string& newName,const string& newContent,const int level) {

Node\* exist=this->isExist(newName);

if (exist) {

printf("文件或文件夹已存在\n");

return false;

}

Node\* file=new Node();

file->name=newName;

file->type=1;

file->content.push\_back(newContent);

Node\* tmp=this->headSon;

this->headSon=file;

file->sibiling=tmp;

file->parent=this;

file->permission=level;

return true;

}

这两个函数的作用是建立一个文件或文件夹。如程序所示，首先要遍历当前目录下的所有文件（使用isExist函数）判断文件是否存在在，如果不存在，则将本级文件的第一个儿子节点设为要创建的这个新节点，新节点的兄弟节点设为父节点原来的第一个儿子节点。

顺序插入记录到文件中以及在文件中间部分插入记录

bool addContent(const string& findName,const string& newContent,const int level) {

Node\* now=this->headSon;

while (now!=0) {

if (now->name==findName) {

if (now->type!=1) {

cout<<"你输入的不是文件名而是文件夹名"<<endl;

return false;

}

if (now->permission>level){

cout<<"权限不足"<<endl;

return false;

}

now->content.push\_back(newContent);

return true;

}

}

cout<<"文件不存在"<<endl;

return false;

}

bool insertFile(const string& findName,const int pos,const string& newContent,const int level){

Node\* now=this->headSon;

while (now!=0) {

if (now->name==findName) {

if (now->type!=1) {

cout<<"你输入的不是文件名而是文件夹名"<<endl;

return false;

}

if (now->permission>level){

cout<<"权限不足"<<endl;

return false;

}

if (pos>=now->content.size()) {

cout<<"你插入的序号太大"<<endl;

return false;

}

now->content.insert(now->content.begin()+pos,newContent);

return true;

}

}

cout<<"文件不存在"<<endl;

return false;

}

这两个函数中，都要先判断当前目录下有没有这个文件，如果有这个文件还需要判断是否有权限；如果有权限方可进行操作。

删除文件或文件夹以及析构函数

bool deleteByName(const string& delName,const int level) {

if (this->headSon->name==delName) {

Node\* tmp=this->headSon->sibiling;

delete this->headSon;

this->headSon=tmp;

return true;

}

Node\* now=this->headSon->sibiling;

Node\* pre=this->headSon;

while (now!=0) {

if (now->name==delName) {

if (now->permission>level){

cout<<"权限不足"<<endl;

return false;

}

Node\* tmp=now->sibiling;

delete now;

pre->sibiling=tmp;

return true;

}

pre=now;

now=now->sibiling;

}

//printf("找不到要删除的文件或文件夹\n");

return false;

}

~Node() {

if (this->type==1) {

//pass

} else {

Node\* now=this->headSon;

while(now!=0) {

Node\* next=now->sibiling;

delete now;

now=next;

}

this->headSon=0;

}

}

删除文件或文件夹，关键是要对文件夹的删除进行处理。具体说来，在写析构函数时，应该将循环删除文件夹下的所有节点。在析构函数张总使用delete时，相当于对这一过程进行了递归调用

创建用户、列出用户等

bool createUser(const string& userName,const string&password,const int level,const int nowLevel){

if (level>nowLevel){

cout<<"无法创建比当前用户权限更高的用户"<<endl;

return false;

}

for (int i=0;i<userList.size();i++){

if (userList[i].userName==userName){

cout<<"该用户已存在"<<endl;

return false;

}

}

userList.push\_back(User(userName,password,level));

cout<<"用户创建成功";

return true;

}

void listUser(){

for (int i=0;i<userList.size();i++){

cout<<"用户名："<<userList[i].userName<<" 权限："<<userList[i].level<<endl;

}

}

LINUX源码分析——进程管理

本部分参考书籍为《深入理解LINUX内核（第三版）》，以下的内容大部分在参考书中也介绍到了，本人跟随参考书对源代码进行阅读、整理，与源代码相结合写出下列分析。LINUX版本与参考书保持一致，为2.6.11。

**1、进程的定义**

进程是程序执行时的一个实例。从内核观点看，进程的目的就是担当分配系统资源的实体。

**2、Linux系统对进程的表示**

在linux-2.6.11\include\linux\sched.h中，定义了task\_struct结构，用来描述进程。下面在task\_struct中挑几个重点的字段进行描述。

**volatile long state** 字段描述进程的状态，-1表示不能运行，0表示可以运行，大于0则表示状态停止中。在sched.h里，又定义了以下几个常量：

|  |  |  |
| --- | --- | --- |
| 状态名称 | 值 | 含义 |
| TASK\_RUNNING | 0 | 进程要么在CPU上执行，要么准备执行 |
| TASK\_INTERRUPTIBLE | 1 | 进程被挂起，知道产生硬件终端 |
| TASK\_UNINTERRUPTIBLE | 2 | 进程被挂起 |
| TASK\_STOPPED | 4 | 进程执行被暂停 |
| TASK\_TRACED | 8 | 进程执行已有调试器debugger暂停 |
| EXIT\_ZOMBIE | 16 | 进程执行被终止，但父进程还没有发布wait()类系统调用来返回有关死亡进程的信息，内核不能丢弃这一状态下的数据 |
| EXIT\_DEAD | 32 | 最终状态，父进程发出wait()类调用后执行 |

其中，后两个状态还能写在**long exit\_state**字段中

**pid\_t pid**字段存放进程标识符。新创建进程的PID通常是前一个进程的PID加1，但PID值有上限，超过上限时会循环使用已闲置的小PID号。

进程描述符存放在栈中，进程描述符之间使用双向链表连接，双向链表的结构list\_head定义在在linux-2.6.11\include\linux\list.h中:

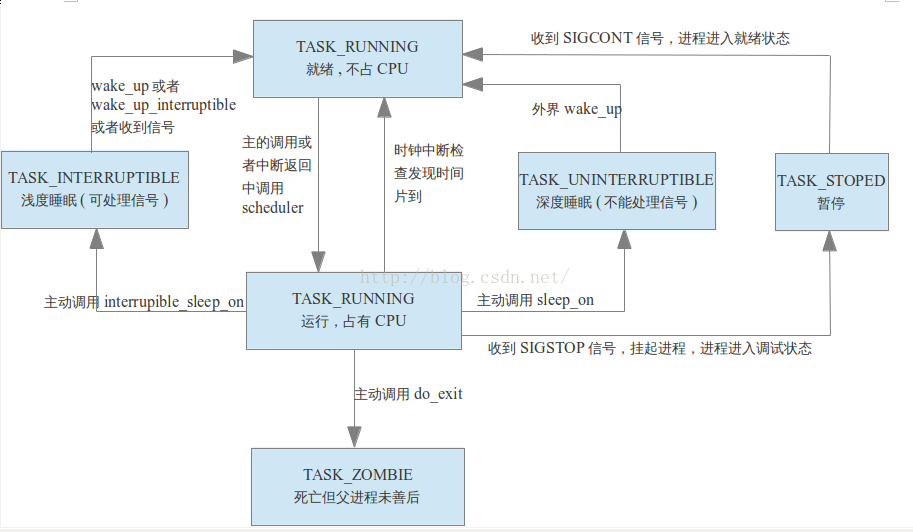
struct list\_head {

struct list\_head \*next, \*prev;

};

在task\_struct中有好几个这样的链表：**struct list\_head tasks**是所有进程间的链表；**struct list\_head run\_list**是可运行进程的链表，**struct list\_head children; struct list\_head sibling;** 用来描述进程的亲属关系等等。

**3、进程的状态转换与模式转换**

LINUX的几种状态在前面已经说明，他们之间的转换可以由下图体现：

[4]

**4、进程上下文切换机制**

（1）定义

上下文切换指内核挂起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行。每个进程虽然拥有属于自己的地址空间，但所有进程都必须共享CPU寄存器；在恢复一个进程的执行之前，内核必须确保每个寄存器装入了挂起进程的值。因此（硬件）上下文就是进程恢复执行前必须装入寄存器的一组数据。

(2)源代码分析，在C:\projects\linux源码\linux-2.6.12.1\linux-2.6.11\kernel\sched.c中定义了context\_switch函数

task\_t \* context\_switch(runqueue\_t \*rq, task\_t \*prev, task\_t \*next)

{

struct mm\_struct \*mm = next->mm;

struct mm\_struct \*oldmm = prev->active\_mm;

if (unlikely(!mm)) {

next->active\_mm = oldmm;

atomic\_inc(&oldmm->mm\_count);

enter\_lazy\_tlb(oldmm, next);

} else

switch\_mm(oldmm, mm, next);

if (unlikely(!prev->mm)) {

prev->active\_mm = NULL;

WARN\_ON(rq->prev\_mm);

rq->prev\_mm = oldmm;

}

/\* Here we just switch the register state and the stack. \*/

switch\_to(prev, next, prev);

return prev;

}

这个函数中，前面的if部分用来切换到新的地址空间，后面调用switch\_to用来切换寄存器和栈中的内容，switch\_to函数是硬件相关的，不同平台不一样。

**5、软中断**

软中断指的是正在执行中的进程发出的由CPU执行某些任务的请求。在linux-2.6.11\include\linux\interupt.h中，定义了以下几种软中断

enum

{

HI\_SOFTIRQ=0, //处理高优先级的tasklet

TIMER\_SOFTIRQ, //处理和时钟中断相关的tasklet

NET\_TX\_SOFTIRQ, //把数据包发送到网卡

NET\_RX\_SOFTIRQ, //从网卡接收数据包

SCSI\_SOFTIRQ, //SCSI命令的后台中断处理

TASKLET\_SOFTIRQ //处理常规tasklet

};

我们自己定义的软中断，归于最后一种tasklet里面，优先级最低。

**extern void open\_softirq(int nr, void (\*action)(struct softirq\_action\*), void \*data);**

用来处理软中断的初始化，三个参数分别是软中断下表、指向要执行的软中断函数以及可能由软中断函数使用的数据结构的指针

**void raise\_softirq(unsigned int nr)**负责触发本地CPU上的软中断

**fastcall unsigned int \_\_do\_IRQ(unsigned int irq, struct pt\_regs \*regs)**定义在linux-2.6.11\kernel\irq\handle.c中，负责执行软中断

**6、进程调度**

Linux进程调度基于分时技术，CPU时间被分成一个个时间片，进程在运行时占用该时间片。调度策略根据进程的优先级对它们进行分类。具体算法又与进程是普通进程或是实时进程而不同。

在kernel\sched.c中，定义了**runqueue**结构，它是LINUX调度程序最重要的数据结构。在进程描述符中，**int prio**与**int static\_prio**分别表示进程的动态优先级和静态优先级，**unsigned long rt\_priority**定义了进程的实时优先级**unsigned long sleep\_avg，unsigned long long timestamp,unsigned long long last\_run**等定义了进程平均睡眠时间、最近插入运行队列的时间、最近一次替换本进程的进程切换时间等参数，方便计算优先级。

**static void recalc\_task\_prio(task\_t \*p, unsigned long long now)**是更新进程的动态优先级的函数，**asmlinkage void schedule(void);**选择要被执行的进程。

7、与进程相关的系统调用。

LINUX与进程相关的系统调用，在这里引用

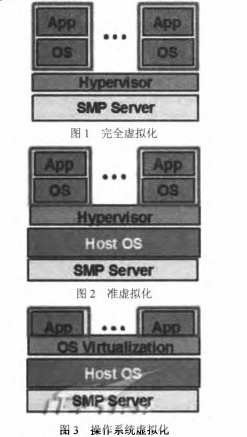
<https://blog.csdn.net/wwwdc1012/article/details/78759326>的列表

| **函数名** | **描述** | **文件** |
| --- | --- | --- |
| fork | 创建一个新进程 | [kernel/fork.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/fork.c) |
| clone | 按指定条件创建子进程 | [kernel/fork.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/fork.c) |
| execve | 运行可执行文件 | [fs/exec.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/fs/exec.c) |
| exit | 中止进程 | [kernel/exit.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/exit.c) |
| \_exit | 立即中止当前进程 |  |
| getdtablesize | 进程所能打开的最大文件数 | [kernel/sys.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sys.c) |
| getpgid | 获取指定进程组标识号 | [kernel/sys.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sys.c) |
| setpgid | 设置指定进程组标志号 | [kernel/sys.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sys.c) |
| getpgrp | 获取当前进程组标识号 | [kernel/sys.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sys.c) |
| setpgrp | 设置当前进程组标志号 | [kernel/sys.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sys.c) |
| getpid | 获取进程标识号 | [kernel/sys.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sys.c) |
| getppid | 获取父进程标识号 | [kernel/sys.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sys.c) |
| getpriority | 获取调度优先级 | [kernel/sys.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sys.c) |
| setpriority | 设置调度优先级 | [kernel/sys.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sys.c) |
| modify\_ldt | 读写进程的本地描述表 | [arch/x86/um/ldt.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/arch/x86/um/ldt.c) |
| nanosleep | 使进程睡眠指定的时间 | [kernel/hrtimer.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/hrtimer.c) |
| nice | 改变分时进程的优先级 |  |
| pause | 挂起进程，等待信号 | [kernel/signal.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/signal.c) |
| personality | 设置进程运行域 | [kernel/exec\_domain.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/exec_domain.c) |
| prctl | 对进程进行特定操作 | [kernel/sys.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sys.c) |
| ptrace | 进程跟踪 | [kernel/ptrace.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/ptrace.c) |
| sched\_get\_priority\_max | 取得静态优先级的上限 | [kernel/sched/core.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sched/core.c) |
| sched\_get\_priority\_min | 取得静态优先级的下限 | [kernel/sched/core.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sched/core.c) |
| sched\_getparam | 取得进程的调度参数 | [kernel/sched/core.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sched/core.c) |
| sched\_getscheduler | 取得指定进程的调度策略 | [kernel/sched/core.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sched/core.c) |
| sched\_rr\_get\_interval | 取得按RR算法调度的实时进程的时间片长度 | [kernel/sched/core.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sched/core.c) |
| sched\_setparam | 设置进程的调度参数 | [kernel/sched/core.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sched/core.c) |
| sched\_setscheduler | 设置指定进程的调度策略和参数 | [kernel/sched/core.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sched/core.c) |
| sched\_yield | 进程主动让出处理器,并将自己等候调度队列队尾 | [kernel/sched/core.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sched/core.c) |
| vfork | 创建一个子进程，以供执行新程序，常与execve等同时使用 | [kernel/fork.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/fork.c) |
| wait | 等待子进程终止 |  |
| wait3 | 参见wait |  |
| waitpid | 等待指定子进程终止 |  |
| wait4 | 参见waitpid | [kernel/exit.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/exit.c) |
| capget | 获取进程权限 | [kernel/capability.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/capability.c) |
| capset | 设置进程权限 | [kernel/capability.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/capability.c) |
| getsid | 获取会晤标识号 | [kernel/sys.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sys.c) |
| setsid | 设置会晤标识号 | [kernel/sys.c](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/kernel/sys.c) |

浅析操作系统新技术——操作系统虚拟化

·1、定义

在计算机领域，虚拟化是将计算机的各种实体资源进行抽象与转换后呈现出来的资源管理技术。本质上，虚拟化就是由位于下层的软件模块箱上一层软件模块提供一个与它原先所期待的运行环境完全一致的接口，使得上层软件可以直接运行在下一层软件所提供的环境上，将一份资源抽象成多份，也可以将多份资源抽象成一份。[1]。虚拟化技术打破了实体结构间的障碍，使用户能够比原来的组态更好的应用系统资源;运用虚拟化技术，可以将性能过剩的高性能硬件或者性能过低的老旧硬件重组重用，更高效地运用资源。

虚拟化技术分为硬件虚拟化、存储虚拟化、网络虚拟化、引用虚拟化等；狭义上，虚拟化常常指的是硬件虚拟化或者是虚拟机，指的是由虚拟化层提供独立的计算机系统，在上层软件看来，每一个虚拟机就是一个真实的机器。硬件虚拟化技术又分为完全虚拟化、准虚拟化与操作系统虚拟化三种。三种虚拟化方式的区别，在于实现虚拟化的中间层(hypervisor)在系统中的层次的高低。如下图所示，完全虚拟化中，hypervisor直接管理调用硬件资源，虚拟机运行在hypervisor上；在准虚拟化中，hypervisor运行在一个传统的操作系统上，是主机操作系统的一个应用软件；操作系统虚拟化没有独立的hypervisor层，由主机操作系统本身负责在多个虚拟服务器之间分配硬件资源。

在前两种虚拟化中，各个虚拟机可以运行不同的操作系统，各个系统间相互独立；在操作系统虚拟化中，虽然各个虚拟机的系统间的运行也是相互独立的，但是它们都必须运行同一种操作系统。

2、操作系统虚拟化的优缺点

前文已经提到，虚拟化能够对资源进行重组重用，更高效地利用计算资源。而相对于传统的基于hypervisor的虚拟化，操作系统虚拟化的优点有：

1. 操作系统虚拟化的效率更高。在完全虚拟化与准虚拟化中，hypervisor必须对硬件进行虚拟；而操作系统虚拟化不需要对硬件进行仿真，而是共享宿主机的内核，由宿主机操作系统调用硬件资源，因此对硬件资源的利用率非常高，接近于物理机的性能；
2. 操作系统虚拟化的启动更方便。基于hypervisor的虚拟化启动时间长，操作系统层虚拟化启动时间则非常短；
3. 操作系统虚拟化更利于共享。操作系统虚拟化的每一个容器因为不含有完整的操作系统，因此非常小，一般为MB级，有利于传输、共享；而完全虚拟化或者半虚拟化里每一个虚拟机都含有完整的操作系统，占用空间容量一般为GB级，不利于共享

当然，相对于传统的虚拟化，操作系统虚拟化也有一些缺点:

1. 操作系统虚拟化的环境依赖于宿主操作系统，每一个容器运行时的操作系统必须与宿主操作系统相同；而完全虚拟化与半虚拟化的虚拟机内可以运行不同于宿主的操作系统。例如，在Win10操作系统的虚拟机上运行Linux或者WinXP操作系统等；
2. 相比于基于hypervisor的虚拟化，操作系统虚拟化实例之间的隔离程度不够，因此安全性不如完全的虚拟化。

3、操作系统虚拟化的应用场景

1.方便代码流水线管理

在软件开发过程中，开发——测试——生产环境的基础硬件都稍有不同，虚拟化技术可以创建从开发到生产一致的运行环境。如果利用传统的虚拟机技术，在不同环节传输的数据量较大、各个环节启动的速度慢、运行时耗费资源较大，不利于实际的应用；而操作系统虚拟化中容器占用的数据空间小、启动速度快、运行时耗费资源少，适合代码流水线不同环节的共享，有利于软件开发

2．方便软件开发环境的部署。

软件开发环境的部署往往耗时长、易出错，在不同机器上不易保持统一。基于操作系统虚拟化进行软件开发环境的部署更利于保持软件开发环境间的统一，减少部署开发环境耗费的时间和精力[2]

1. 方便微服务架构的使用

利用操作系统虚拟化，可以讲传统分布式服务继续拆分解耦，形成一些更小的服务模块，每个服务模块运行在一个小容器上，使整个系统更利于维护与敏捷开发。[3]

其它的应用还有将操作系统虚拟化应用在Web服务器上、应用在一次性应用上等等

4、对操作系统虚拟化的典型——Docker进行简要分析

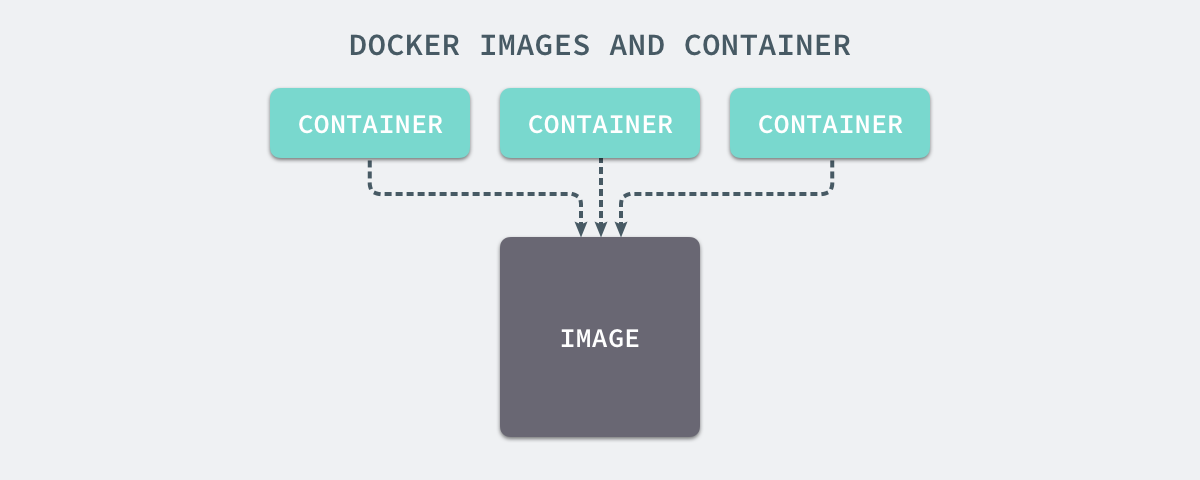
目前使用最广泛的操作系统虚拟化技术是Docker。它是基于Go语言的一款开源的操作系统虚拟化容器引擎。下面我们主要对Docker的原理进行简要的分析。

Docker的实现基于Linux提供的命名空间（Namespaces）、控制组（Control Groups）以及联合文件系统三种方法。

命名空间是Linux提供的用于分离进程树、网络接口、挂载点以及进程间通信等资源的方法。Docker通过Linux的命名空间机制对不同的容器实现了隔离。

Linux的命名空间为不同的容器隔离了文件系统、网络并与宿主机器之间的晋城相互隔离，但命名空间并不能够进行物理资源上的隔离，如果一个容器占用了过多的硬件资源，就会对整体的效率产生影响。而Control Groups就提供了对宿主机器上的物理资源上的隔离，例如CPU、内存、网络带宽等等

另外，为了实现容器的快速部署与应用，Docker采用了联合文件系统，讲Docker的文件系统分为镜像(image)与容器（container）两个部分，镜像是只读的，而容器是可以修改的，多个容器可以共享同一个镜像；从而在进行Docker的分享时，可以将镜像打包复用、快速部署。这一机制就是联合文件系统。



[1]修长虹,梁建坤,辛艳.虚拟化技术综述[J].网络安全技术与应用,2016,(5):18-19. DOI:10.3969/j.issn.1009-6833.2016.05.012.

[2] 王飞. 基于Docker的研发部署管理平台的设计与实现[D]. 北京交通大学, 2015.

[3] 《浅析Docker容器的应用场景》<http://www.dockone.io/article/1282>

[4] https://blog.csdn.net/kklvsports/article/details/52268085