Crtl 的快捷键

<Backspace>或<Ctrl-H>删除前一个字符
<Ctrl-U>删除当前行
<Ctrl-C>终止现在的命令，终止一个前台进程
<Ctrl-Z>挂起一个前台进程
<Ctrl+D>退出当前的 shell，eof，必须从登陆 shell 退出，必须关闭所有的 shell
<Ctrl-K>删除一行光标后字符
<Ctrl-P>上一次执行的命令，扫描过的不会再次出现

Shell 命令搜索路径

Shell 搜索的目录名字都保存在 shell 变量 PATH（在 TC shell 中是 path）中。
变量 PATH 中的目录名用符号分开。在 bash 中 “:”
变量 PATH 保存在~/.profile 或者~/.login 中(~:主目录)

Shell 元字符

"	引用多个字符,允许替换	"$file".bak
‘	引用多个字符	‘$100,000’
$	一行的结束/显示变量的值	$PATH
&	让一个命令在后台执行	command &
( )	在子 shell 中执行命令	(cmd1;cmd2)
{ }	在当前 shell 中执行命令	{cmd1;cmd2}
*	匹配 0 个或者多个字符	chap*.ps
?	匹配单个字符	lab.?
[ ]	插入通配符	[a-s],[1,5-9]
^	一行的开始/否定符号	[^3-8]
`	替换命令	PS1=`cmd`
|	创建命令间的管道 cmd1|cmd2	
;	分割顺序执行的命令	cmd1;cmd2
<	重定向命令的输入	cmd<file
>	重定向命令的输出	cmd>file
\	转义字符/允许在下一行中继续 shell 命令	
!	启动历史记录列表中的命令和当前命令 !!,!4	
%	TC shell 的提示符，或者指定一个任务号时作为起始字符 %或者%3	

常用命令

whatis：得到任何 LINUX 命令的更短的描述
whoami：显示用户名 leafior
which：当某个工具或程序有多个副本时，用 which 来识别哪个副本在运行
who：显示现在正在使用系统的用户的信息
w：比 who 更加详细地列出系统上用户的信息
hostname：显示登录上的主机的名字 Ubuntu
uname：显示操作系统的信息 Linux
PATH=~/bin:$PATH:. 搜索路径中增加~/bin 和.目录
cal [[month] year]如 cal 4 2011
alias [name[=string]⋯] 为 name 命令建立别名 string。如 alias more=’pg’
unalias 删除别名 alias ll=’ ls －C’
uptime 显示系统运行时间命令
su [-][-c <command>] [username]
-c< cmd >执行完指定的指令后，即恢复原身份。
-改变身份时，也同时变更工作目录，及 HOME、SHELL、PATH 变量等
Username 指定要变更的用户名，默认为 root

文件

目录文件 d,字符设备文件 c,块设备文件 b 普通文件（文件名不超过 255）
字符设备文件和块设备文件：
fd0 (for floppy drive 0)
hda (for harddisk a)
lp0 (for line printer 0)
tty(for teletype terminal)
管道(FIFO)文件,链接文件,socket 文件
/根目录：包含了所有的目录和文件。
/bin：也称二进制目录，包含了那些供系统管理员和普通用户使用的重要的 Linux 命令的可执行文件。目录/usr/bin 下存放了大部分的用户命令。
/boot ：包括 Linux 内核的二进制映像。内核文件名是 vmlinux 加上版本和发布信息。
/dev：包含所有 linux 系统中使用的外部设备。但是这里并不是放的外部设备的驱动程序。
/etc：存放了系统管理时要用到的各种配置文件和子目录。网络配置文件，文件系统，x 系统配置文件，设备配置信息，设置用户信息等都在这个目录下。
/sbin：系统管理员的系统管理程序。
/home：/home/leafior
/lib：几乎所有的应用程序都会用到这个目录下的系统动态连接共享库。
/mnt ：这个目录主要用来临时装载文件系统 mount
/opt：该目录用来安附加软件包
/proc ：进程和系统得信息，可以在这个目录下获取系统信息。这些信息是在内存中，由系统自己产生的。
/root ：根（root）用户的主目录
/sbin，/usr/sbin，/usr/root/sbin：存放了系统管理的工具、应用软件和通用的 root 用户权限的命令
/tmp：用来存放不同程序执行时产生的临时文件
/usr：存放了可以在不同主机间共享的只读数据。
/lost+found：存放所有和其他目录没有关联的文件，这些文件可以用 Linux 工具 fsck 查找得到。
/var：存放易变数据。/var/spool/mail 存放收到的电子邮件，/var/log 存放系统的日志，/var/ftp

mount [-t fstype] [-o options] /dev/xxyN dirname
如：mount –t vfat /dev/hda1 /mnt/c
类型 设备文件 挂在目录
挂接 U 盘：mount –t vfat /dev/sda1 /mnt/usb

正则表达式

支持工具：awk, ed, egrep, grep, sed, vi

x|y|z	x 或 y 或 z
/L..e/	Love, Live, Lose, …
^x	以 x 开始的 string
x$	以 x 结束的 string
*	*
(xy)+或\(xy\)+	xy,xyxy,xyxyxy, …
xy?	x, xy
xy*	x, xy, xyy, xyyy
xy+	xy, xyy, xyyy, ..
[], [^]	/[Hh]ello/ /[^A-KM-Z]ove/ Love
{n}	匹配 n 次
{n,}	匹配 n 或 n+次
{n,m}	>=n, <=m

权限
对于目录：
r :列出目录的内容 w：建立，删除
x :允许用户搜索这个目录，如果你没有对目录的执行特权，那么就不能使用 ls －l 命令来列出目录下的内容或者是使用 cd 命令来把该目录变成当前目录。
Chmod u=rwx courses
文件访问权限 = 默认的访问权限－mask
默认访问权限：执行文件为 777 文本文件为 666
Umask 013 对于一个新建的可执行文件 764
软硬连接

进程

Shell 执行二进制文件：
1.Shell 使用 fork 创建子进程；2.子进程执行 exec，用命令对应的可执行文件覆盖自身；3.命令执行，bash 等待命令结束。
Shell 执行脚本文件：
创建一个子 shell 并让子 shell 依次执行脚本中命令，执行与从键盘输入的命令采用相同的方式。子 shell 为每一个要执行的命令创建一个子进程。子 shell 执行脚本文件中的命令时，父 shell 等待子 shell 结束。子 shell 遇到脚本文件的 EOF 终止。子 shell 终止，父 shell 结束等待状态，开始重新执行。
命令组： 命令组中的所有命令都在一个进程中执行（在当前 shell 的子 shell 中）
$ (date;echo Hello,World!)

fg[%jobid] 后台→前台
bg 挂起进程→后台，参数同 fg（后台→前台）
jobs 显示所有挂起/停止的和后台进程的作业号
suspend 可以挂起当前 shell 进程
;顺序执行 &并发执行 | 重定向

重定向

command < input-file > output-file
command > output-file < input-file
>换成>>则追加文件，否则替换
sdin— 0 stout — 1 sderr — 2
2> &1：使文件描述符 2 为文件描述符 1 的拷贝，导致错误信息送往和该命令输出相同的地方
$ cat lab1 lab2 lab3 2>&1 1>output
标准出错先设置成显示器，标准输出才改为 output

IPC

Linux 实现**进程间通信**(IPC Inter Process Communication)方法有：System V IPC 机制（信号量，消息队列，共享内存）；管道（pipe）、命名管道；套接字（socket）；信号(signal)

概要

最为重要的决策之一是采用 GPL（GNU General Public License）。设计 Linux 三原则（实用|有限目标|简单设计）
内核版本的序号： major(主版本号).minor(次版本号).patchlevel（对当前版本的修订次数）
Linux 系统结构(计算机硬件|linux 内核|shell|应用层|用户)
Linux 是一个单内核，Linux 内核运行在单独的内核地址空间.Linux 吸取了微内核的精华：其引以为豪的是模块化设计、抢占式内核、支持内核线程以及动态装载和卸载内核模块。
GNU C Library（glibc）提供了连接内核的系统调用接口
Linux 内核源程序安装在/usr/src/linux
Linux 系统引导过程使用内核镜像，/boot 目录下文件名称如：vmlinuz-2.6.15.5：普通内核镜像：zImage（Image compressed with gzip），大小不能超过 512k.大内核镜像：bzImage(big Image compressed with gzip)，包含了大部分系统核心组件：系统初始化、进程调度、内核管理模块

Linux 线程

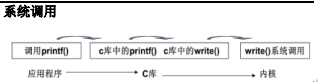
Linux 2.6 内核支持 clone()系统调用创建线程
pthread_create()：创建线程函数；
pthread_exit()：主动退出线程；
pthread_join()：用于将当前线程挂起来等待线程的结束。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源就被收回。
pthread_cancel()：终止另一个线程的执行。

重建内核

make clean 删除大多数的编译生成文件， 但是会保留内核的配置文件.config， 还有足够的编译支持来建立扩展模块
make mrproper 删除所有的编译生成文件， 还有内核配置文件， 再加上各种备份文件
make distclean mrproper 删除的文件， 加上编辑备份文件和一些补丁文件。

```
# cp /boot/config-`uname -r` .config
# make menuconfig
# make –j4
# make modules_install
# make install
Sudo mkinitramfs -o /boot/initrd.img-2.6.36
Sudo update-initramfs -c -k 2.6.36
Sudo update-grub2 //自动修改系统引导配置，产生 grub.cfg启动文件
```

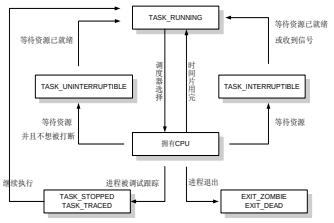
系统调用



系统调用是用户进程进入内核的接口层，它本身并非内核函数，但它是由内核函数（服务例程）实现的。
运行模式（mode）
Linux 使用了其中的两个：特权级 0 和特权级 3 ，即内核模式(kernel mode)和用户模式(user mode)
上下文（context）。三个部分：
用户级上下文：正文、数据、用户栈以及共享存储区；
寄存器上下文：通用寄存器、程序寄存器（IP）、处理机状态寄存器（EFLAGS）、栈指针（ESP）；
系统级上下文：进程控制块 task_struct、内存管理信息(mm_struct、vm_area_struct、 pgd、pmd、pte 等)、核心栈等。
程序执行系统调用步骤：
1、程序调用 libc 库的封装函数。
2、调用软中断 int 0x80 进入内核。
3、在内核中首先执行system_call函数，接着根据系统调用号在系统调用表中查找到对应的系统调用服务例程 。
4、执行该服务例程。
5、执行完毕后，转入ret_from_sys_call例程，从系统调用返回

进程状态

TASK_RUNNING：正在运行的进程即系统的当前进程或准备运行的进程即在 Running 队列中的进程。只有处于该状态的进程才实际参与进程调度。
TASK_INTERRUPTIBLE：处于等待资源状态中的进程，当等待的资源有效时被唤醒，也可以被其他进程或内核用信号、中断唤醒后进入就绪状态。
TASK_UNINTERRUPTIBLE：处于等待资源状态中的进程，当等待的资源有效时被唤醒，不可以被其它进程或内核通过信号、中断唤醒。
TASK_STOPPED：进程被暂停，一般当进程收到下列信号之一时进入这个状态：SIGSTOP，SIGTSTP，SIGTTIN 或者 SIGTTOU。通过其它进程的信号才能唤醒。
TASK_TRACED：进程被跟踪，一般在调试的时候用到。
EXIT_ZOMBIE：正在终止的进程，等待父进程调用 wait4()或者 waitpid()回收信息。是进程结束运行前的一个过度状态（僵死状态）。虽然此时已经释放了内存、文件等资源，但是在内核中仍然保留一些这个进程的数据结构（比如 task_struct）等待父进程回收。
EXIT_DEAD：进程消亡前的最后一个状态，父进程已经调
用了 wait4()或者 waitpid()。
TASK_NONINTERACTIVE：表明这个进程不是一个交互式进程，在调度器的设计中，对交互式进程的运行时间片会有一定的奖励或者惩罚。



进程创建的原理：

系统创建的第一个进程是 init 进程。系统中所有的进程都是由当前进程使用系统调用 fork()创建的。子进程被创建后继承了父进程的资源。子进程共享父进程的虚存空间。
用 exec 系列函数执行真正的任务。
Fork() 函数进程创建的过程:
1) 为新进程分配 task_struct 内存空间;
2) 把父进程 task_struct 拷贝到子进程的 task_struct;
3) 为新进程在其虚拟内存建立内核堆栈;
4) 对子进程 task_struct 中部分进行初始化设置
5) 把父进程的有关信息拷贝给子进程，建立共享关系;
6) 把子进程的 counter 设为父进程 counter 值的一半;
7) 把子进程加入到可运行队列中;
8) 结束 do_fork()函数返回 PID 值.
Linux 把**线程**和进程一视同仁，每个线程拥有唯一属于自己的 task_struct 结构。不过线程本身拥有的资源少，共享进程的资源，如共享地址空间、文件系统资源、文件描述符和信号处理程序。内核线程是通过系统调用 clone()来实现的

Exec()

在 Linux 系统中，使程序执行的唯一方法是使用系统调用 exec()。exec 函数族把当前进程映像替换成新的程序文件，而且该程序通常 main 函数开始执行。
其中只有 execve 是真正意义上的系统调用，其它都是在此基础上经过包装的库函数。exec 函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，就是在调用进程内部执行一个可执行文件。

fork（）函数

fork:创建一个新子进程
#include <sys/types.h>#include <unistd.h>pid_t fork(void);
返回值：调用一次，返回两次。子进程的返回值是 0，父进程的返回值则是子进程的进程 ID,出错为-1
if ((pid=fork()) < 0) { /* error handling */}
else if (pid == 0) { /* child */}
else { /* parent */};
用 fork 函数创建子进程后，子进程往往要调用一种 exec 函数以执行另一个程序。当进程调用一种 exec 函数时，该进程完全由新程序代换，而新程序则从其 main 函数开始执行。调用 exec 并不创建新进程，所以前后的进程 PID 并未改变。exec 只是用另一个新程序替换了当前进程的正文、数据、堆和栈段。
子进程调用 g e t p p i d 以获得其父进程的进程 I D
子进程和父进程共享很多资源，除了打开文件之外，很多父进程的其他性质也由子进程继承：如实际用户 ID、实际组 ID、有效用户 ID、有效组 ID 等。父、子进程之间的区别包括 fork 的返回值，进程 ID，不同的父进程 ID，父进程设置的锁，子进程不继承等。

do_fork()的执行过程(源代码在 kernel/fork.c 文件中):
1) 调用 alloc_task_struct()分配子进程 task_struct 空间。严格地讲，此时子进程还未生成。
2) 把父进程 task_struct 的值全部赋给子进程 task_struct。
3) 检查是否超过了资源限制，如果是，则结束并返回出错信息。更改一些统计量的信息。
4) 修改子进程 task_struct 的某些成员的值使其正确反映子进程的状况，如进程状态被置成 TASK_UNINTERRUPTIBLE。
5) 调用 get_pid( )函数为子进程得到一个 pid 号。
6)共享或复制父进程文件处理、信号处理及进程虚拟地址空间等资源。
7) 调用 copy_thread( )初始化子进程的核心模式栈时，核心栈保存了进程返回用户空间的上文。此处与平台相关，以 i386 为例，其中很重要的一点是存储寄存器 eax 值的位置被置 0,这个值就执行系统调用后子进程的返回值。
8) 将父进程的当前的时间配额 counter 分一半给子进程。
9) 利用宏 SET_LINKS 将子进程插入所有进程都在其中的双向链表。调用 hash_pid（）,将子进程加入相应的 hash 队列。
10) 调用 wake_up_process( ),将该子进程插入可运行队列。至此，子进程创建完毕,并在可运行队列中等待被调度运行。
11) 如果 clone_flags 包含有 CLONE_VFORK 标志，则将父进程挂起直到子进程释放进程空间。进程控制块中有一个信号量 vfork_sem 可以起到将进程挂起的作用。
12) 返回子进程的 pid 值，该值就是系统调用后父进程的返回值。

进程调度

Linux 系统采用抢占调度方式。无论内核态还是用户态。
分时技术，对于优先级相同进程进程采用时间片轮转法。
根据进程的优先级对它们进行分类。进程的优先级是动态的。
Linux 2.6 的进程设置 140 个优先级。实时进程优先级为 0-99，普通进程优先级 100-139 的数。0 为最高优先权，139 为最低优先权。优先级数值越大，优先级越低，分配的时间片越少
实时进程的 static_prio 不参与优先级 prio 的计算
unsigned long policy：进程调度策略

162	#define SCHED_NORMAL	0 普通进程时间片轮转
163	#define SCHED_FIFO	1 实时进程先进先出
164	#define SCHED_RR	2 实时进程时间片轮转
165	#define SCHED_BATCH	3 后台处理进程（无交互性）

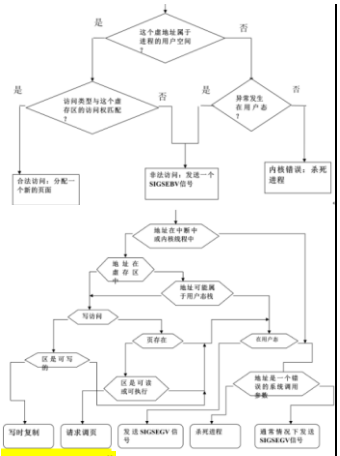
调度对象是可运行队列，每个处理器有一个可运行队列
普通进程的**权值**就是它的 counter 的值（处置 21），而实时进程的权值是它的 rt_priority 的值加 1000

Task_struct 结构

int pid //进程标识号
unsigned short uid, gid //用户标识号，组标识号
unsigned short euid, egid //用户有效标识号，组有效标识号
unsigned short suid, sgid //用户备份标识号，组备份标识号
unsigned short fsuid, fsgid //用户文件标识号，组文件标识号
3.进程的族亲关系
struct task_struct *p_opptr //指向祖先进程 PCB 的指针
struct task_struct *p_pptr //指向父进程 PCB 的指针
struct task_struct *p_cptr //指向子进程 PCB 的指针
struct task_struct *p_ysptr //指向弟进程 PCB 的指针
struct task_struct *p_osptr //指向兄进程 PCB 的指针
4. 进程间的链接信息
struct task_struct *next_task //指向下一个 PCB 的指针
struct task_struct *prev_task //指向上一个 PCB 的指针
struct task_struct *next_run //指向可运行队列的下一个 PCB 的指针
struct task_struct *prev_run //指向可运行队列的上一个 PCB 的指针
5.进程的调度信息
long counter //时间片计数器
long nice //进程优先级
unsigned long rt_priority //实时进程的优先级
unsigned long policy //进程调度策略
6.进程的时间信息
long start_time //进程创建的时间
long utime //进程在用户态下耗费的时间
long stime //进程在核心态下耗费的时间
long cutime //所有子进程在用户态下耗费的时间
long cstime //所有子进程在核心态下耗费的时间
unsigned longtimeout //进程申请延时
7.进程的虚存信息
struct mm_struct *mm //进程的虚存信息
struct desc_struct *ldt //进程的局部描述符表指针
unsigned long saved_kernel_stack //核心态下堆栈的指针
unsigned long kernel_stack_page //核心态下堆栈的页表指针
8. 进程的文件信息
struct fs_struct *fs //进程的可执行映象所在的文件系统
struct files_struct *files //进程打开的文件
9.与进程间通信有关的信息
unsigned longsignal //进程接收到的信号
unsigned longblocked //阻塞信号的掩码
struct signal_struct*sig //信号处理函数表的指针
int exit_signal //进程终止的信号
struct sem_undo *semundo //进程要释放的信号量
struct sem_queue *semsleeping //与信号量操作相关的等待队列
10. 其它信息
int errno //系统调用的出错代码
long debugreg[8] //进程的 8 个调试寄存器
char comm[16] //进程接收到的信号
PID 是 32 位的无符号整数，它被顺序编号，最大值为 32768。
内核堆栈：thread_info 代替了原先 task_struct 的位置，跟内核堆栈放在一块，thread_info 中放置一个指向 task_struct 的指针

存储管理

一个进程的用户地址空间主要由 mm_struct 结构和 vm_area_structs 结构来描述. mm_struct 结构它对进程整个用户空间进行描述, vm_area_structs 结构对用户空间中各个区间(简称虚存区)进行描述. mm_struct 结构首地址在 task_struct 成员项 mm 中：
struct mm_struct *mm：vm_area_struct 结构是虚存空间中一个连续的区域，在这个区域中的信息具有相同的操作和访问特性
fork()是通过拷贝或共享父进程的用户空间来实现的，即内核调用 copy_mm()函数，为新进程建立所有页表和 mm_struct 结构



do_page_fault()

页面异常的处理程序两个参数：
一个是指针,指向异常发生时寄存器值存放的地址。
另一个错误码,由三位二进制信息组成：
第 0 位——访问的物理页帧是否存在：
第 1 位——写错误还是读错误或执行错误：
第 2 位——程序运行在核心态还是用户态。
do_page_fault( )函数定义在 arch/i386/mm/fault.c 文件中
写时拷贝的处理过程：
1.改写对应页表项的访问标志位，表明其刚被访问过,调度时不会优先考虑。2.如果该页帧目前只为一个进程单独使用,则只需把页表项置为可写。3.如果该页帧为多个进程共享,则申请一个新的物理页面并标记为可写,复制原来物理页面的内容,更改当前进程相应的页表项,同时原来的物理页帧的共享计数减一。
按需调页的处理过程：
1. 页面从未被进程访问过,这种情况页表项的值全为 0。(1)如果所属区间的 vm_ops->nopage 不为空,表示该区间映射到一个文件,并且 vm_ops->nopage 指向装入页面的函数,此时调用该函数装入该页面。(2)如果 vm_ops 或 vm_ops->nopage 为空,则该调用 do_anonymous_page( )申请一个页面；
2. 该页面被进程访问过,但是目前已被写到交换分区,页表项的存在标志位为 0,但其他位被用来记录该页面在交换分区中的信息。调用 do_swap_page( )函数从交换分区调入该页面。
选择被换出的页面策略

Linux 内核利用守护进程 **kswapd** 定期地检查系统内的空闲页面数是否小于预定义的极限
Buddy 算法是把内存中的所有页帧按照 2^n 划分，其中 n=0～10。划分后形成了大小不等的存储块，称为页帧块，简称页块。数组 free_area[]来管理各个空闲页块组，申请空间的函数为 alloc_pages( )；释放函数为 free_pages( )；
slab 分配器：为经常使用的小对象建立缓冲,小对象的申请与释放都通过 slab 分配器来管理。slab 分配器再与伙伴系统打交道。基于伙伴系统的 slab 分配器

VFSVFS 仅存在于内存
超级块对象 superblock :存储已安装文件系统的信息，通常对应磁盘文件系统的文件系统超级块或控制块。
索引节点对象 inode object ：存储某个文件的信息。通常对应磁盘文件系统的文件控制块
目录项对象 dentry object ：dentry 对象主要是描述一个目录项，是路径的组成部分。
文件对象 file object：存储一个打开文件和一个进程的关联信息。只要文件一直打开，这个对象就一直存在与内存
File 对象
文件对象 file 表示进程已打开的文件，只有当文件被打开时才在内存中建立 file 对象的内容。
该对象由相应的 open()系统调用创建，由 close()系统调用销毁。

添加系统调用：
system_call()函数实现了系统调用中断处理程序：
1.它首先把系统调用号和该异常处理程序用到的所有 CPU 寄存 器保存到相应的栈中， SAVE_ALL
2.把当前进程 task_struct （thread_info）结构的地址存放在 ebx 中
3.对用户态进程传递来的系统调用号进行有效性检查。若调 用号大于或等于 NR_syscalls，系统调用处理程序终止。 （sys_call_table）
4.若系统调用号无效，函数就把-ENOSYS 值存放在栈中 eax 寄 存器所在的单元，再跳到 ret_from_sys_call()
5.根据 eax 中所包含的系统调用号调用对应的特定服务例程
实验修改的主要有 3 处地方：
for (p = &init_task; (p = next_task(p)) != &init_task;)
//遍历进程

编译内核：
make clean 删除大多数的编译生成文件， 但是会保留内核的配置文件.config， 还有足够的编译支持来建立扩展模块
make mrproper 删除所有的编译生成文件， 还有内核配置文件， 再加上各种备份文件
make distclean mrproper 删除的文件， 加上编辑备份文件和一些补丁文件。
apt-get install kernel-package libncurses5-dev fakeroot wget bzip2
//安装工具包
make config 是有问必答的方式，每个内核选项它都会问你要、不要、模块，选错了一个就必须从头再来一遍；
make menuconfig 提供一个基于文本的图形界面，它依赖于 ncurses5 这个包，键盘操作，可以修改选项，一般推荐用这个；
make xconfig 需要你有 x window system 支持，就是说你要在 KDE、GNOME 之类的 X 桌面环境下才可用，好处是支持鼠标，坏处是 X 本身占用系统周期，而且 X 环境容易引起编译器的不稳定
make -j4 启动 4 个线程（双核）来编译内核文件生成.o 等中间文件
内核文件 bzImage 的位置在/usr/src/linux/arch/i386/boot 目录下。
make modules_install 安装模块
make install 使用命令 make install 将 bzImage 和 System.map 拷贝到/boot 目录下。这样，Linux 在系统引导后从/boot 目录下读取内核映像到内存中

添加系统调用：
system_call()函数实现了系统调用中断处理程序：
1.它首先把系统调用号和该异常处理程序用到的所有 CPU 寄存 器保存到相应的栈中， SAVE_ALL
2.把当前进程 task_struct （thread_info）结构的地址存放在 ebx 中
3.对用户态进程传递来的系统调用号进行有效性检查。若调 用号大于或等于 NR_syscalls，系统调用处理程序终止。 （sys_call_table）
4.若系统调用号无效，函数就把-ENOSYS 值存放在栈中 eax 寄 存器所在的单元，再跳到 ret_from_sys_call()
5.根据 eax 中所包含的系统调用号调用对应的特定服务例程

实验修改的主要有 3 处地方：
for (p = &init_task; (p = next_task(p)) != &init_task;)
//遍历进程
p->comm //comm 类型为 char[16],代表进程名
p->pid //当亲进程号
p->state //当前进程的状态
-1 unrunnable，0 runnable，>0 stopped
p-> parent //指向父进程 task_struct 的地址

wait（）函数原型为:pid_t wait(int *status)
当进程退出时,它向父进程发送一个 SIGCHLD 信号,默认情况下总是忽略 SIGCHLD 信号,此时进程状态一直保留在内存中,直到父进程使用 wait 函数收集状态信息,才会清空这些信息。用 wait 来等待一个子进程终止运行称为回收进程。 wait()要与 fork()配套出现,如果在使用 fork()之前调用 wait(),wait()的返回值则为-1,正常情况下 wait()的返回值为子进程的 PID。如果先终止父进程,子进程将继续正常进行,只是它将由 init 进程(PID 1)继承,当子进程终止时,init 进程捕获这个状态。当父进程忘了用 wait()函数等待已终止的子进程时,子进程就会进入一种无父进程清理自己尸体的状态,此时的子进程就是僵尸进程.如子进程虽然执行完毕,但父进程没有调用 wait(),出现子进程虽然死亡,而不能在内核中清理尸体的情况.父进程用 wait()会回收掉尸体.
如果参数的值不是 NULL,wait 就会把子进程退出时的状态取出并存入其中,这是一个整数值(int),指出了子进程是正常退出还是被非正常结束的,
由于这些信息被存放在一个整数的不同二进制位中,所以就设计了一套专门的宏来完成这项工作,其中最常用的两个：
1,WIFEXITED(status)这个宏用来指出子进程是否为正常退出的,如果是,它会返回一个非零值.
2,WEXITSTATUS(status)当 WIFEXITED 返回非零值时,我们可以用这个宏来提取子进程的返回值,如果子进程调用 exit(5)退出,WEXITSTATUS(status)就会返回 5;
如果子进程调用 exit(7),WEXITSTATUS(status)就会返回 7.请注意,如果进程不是正常退出的,也就是说,WIFEXITED 返回 0,这个值就毫无意义.

exit（）命令
用于退出当前 shell，在 shell 脚本中可以终止当前脚本执行。
格式：exit n--Cause the shell to exit with a status of n.
格式：exit--即为最后一个命令的退出码。
格式：$?--上一个命令的退出码。
格式：trap "commands" EXIT--退出时执行 commands 指定的命令。（ A trap on EXIT is executed before the shell terminates.）
退出码（exit status，或 exit code）的约定：
0 表示成功（Zero - Success）
非 0 表示失败（Non-Zero - Failure）
2 表示用法不当（Incorrect Usage）
127 表示命令没有找到（Command Not Found）
126 表示不是可执行的（Not an executable）
>=128 信号产生

```
struct inode {
    struct list_head    i_hash; /* inode hash 链表指针 */
    struct list_head    i_list;     /* inode 链表指针 */
    struct list_head    i_dentry; /*dentry 链表*/
    kdev_t              i_dev;      /* 主设备号*/
    unsigned long       i_ino; /* 外存的 inode 号 */
    umode_t             i_mode;     /* 文件类型和访问权限 */
    nlink_t             i_nlink;    /* 该文件的链接数 */
    uid_t               i_uid; /* 文件所有者的用户标识*/
    gid_t               i_gid;      /* 文件的用户组标识*/
    kdev_t              i_rdev;     /* 次设备号 */
    off_t               i_size;    /* 文件长度，以字节为单位 */
    time_t              i_atime;    /* 文件最后一次访问时间 */
    time_t              i_mtime;    /* 文件最后一次修改时间 */
    time_t              i_ctime;    /* 文件创建时间 */
    unsigned long       i_blksize; /* 块尺寸，以字节为单位 */
    unsigned long       i_blocks;   /* 文件的块数 */
    unsigned long       i_version; /* 文件版本号 */
    unsigned long       i_nrpages; /* 文件在内存中占用的页面数 */
    struct semaphore    i_sem; /* 文件同步操作用的信号量 */
    struct inode_operations     *i_op; /* 指向 inode 操作函数入口表的指针 */
    struct super_block          *i_sb;     /* 指向该文件系统的 VFS 超级块 */
    struct wait_queue           *i_wait;    /* 文件同步操作用等待队列 */
    struct file_lock        *i_flock; /* 指向文件锁定链表的指针 */
    struct vm_area_struct *i_mmap; /* 文件使用的虚存区域 */
    struct page *i_pages;       /* 指向文件占用内存页面 page 结构体链表 */
    struct dquot *i_dquot[MAXQUOTAS];
    struct inode *i_bound_to, *i_bound_by;
    struct inode *i_mount;      /* 指向该文件系统根目录 inode 的指针 */
    unsigned long i_count; /* 使用该 inode 的进程计数 */
    unsigned short i_flags; /* 该文件系统的超级块标志 */
            unsigned short i_writecount;/* 写计数 */
            unsigned char i_lock; /* 对该 inode 的锁定标志 */
            unsigned char i_dirt; /* 该 inode 的修改标志 */
            unsigned char i_pipe; /* 该 inode 表示管道文件 */
            unsigned char i_sock; /* 该 inode 表示套接字 */
            unsigned char i_seek;       /* 未使用 */
            unsigned char i_update;   /* inode 更新标志 */
            unsigned char i_condemned;
}
```

VFS 的 inode 与某个文件的对应关系是通过设备号 i_dev 与 inode 号 i_ino 建立的，它们唯一地指定了某个设备上的一个文件或目录。
VFS 的 inode 是物理设备上的文件或目录的 inode 在内存中的统一映像。这些特有信息是各种文件系统的 inode 在内存中的映像。如 EXT2 的 ext2_inode_info 结构。
i_lock 表示该 inode 被锁定，禁止对它的访问。i_flock 表示该 inode 对应的文件被锁定。i_flock 是个指向 file_lock 结构链表的指针， 该链表指出了一系列被锁定的文件。
VFS 的 inode 组成一个双向链表，全局变量 first_inode 指向链表的表头。在这个链表中，空闲的 inode 总是从表头加入，而占用的 inode 总是从表尾加入。
系统还设置了一些管理 inode 对象的全局变量，如：
max_inodes 给定了 inode 的最大数量，
nr_inodes 表示当前使用的 inode 数量，
nr_free_inodes 表示空闲的 inode 数量。