1. 针对你所下载的（非运行的）Linux 内核，系统有多少个系统调用？（对内核源码中相关文件内容截屏）

答：Linux-0.11 中有 72 个系统调用，源代码中相关内容（include/linux/sys.h）的截屏如下：

```
extern int sys_setup();
extern int sys_exit();
extern int sys_fork();
extern int sys_read();
extern int sys_write();
extern int sys_open();
extern int sys_close();
extern int sys_waitpid();
extern int sys_creat();
extern int sys_link();
extern int sys_unlink();
extern int sys_execve();
extern int sys_chdir();
extern int sys_time();
extern int sys_mknod();
extern int sys_chmod();
extern int sys_chown();
extern int sys_break();
extern int sys_stat();
extern int sys_lseek();
extern int sys_getpid();
extern int sys_mount();
extern int sys_umount();
extern int sys_setuid();
extern int sys_getuid();
extern int sys_stime();
extern int sys_ptrace();
extern int sys_alarm();
extern int sys_fstat();
extern int sys_pause();
extern int sys_utime();
extern int sys_stty();
extern int sys_gtty();
extern int sys_access();
extern int sys_nice();
extern int sys_ftime();
extern int sys_sync();
extern int sys_kill();
extern int sys_rename();
extern int sys_mkdir();
extern int sys_rmdir();
extern int sys_dup();
extern int sys_pipe();
extern int sys_times();
extern int sys_prof();
extern int sys_brk();
extern int sys_setgid();
extern int sys_getgid();
extern int sys_signal();
extern int sys_geteuid();
extern int sys_getegid();
extern int sys_acct();
extern int sys_phys();
extern int sys_lock();
extern int sys_ioctl();
extern int sys_fcntl();
extern int sys_mpx();
extern int sys_setpgid();
extern int sys_ulimit();
extern int sys_uname();
extern int sys_umask();
extern int sys_chroot();
extern int sys_ustat();
extern int sys_dup2();
extern int sys_getppid();
extern int sys_getpgrp();
extern int sys_setsid();
extern int sys_sigaction();
extern int sys_sgetmask();
extern int sys_ssetmask();
extern int sys_setreuid();
extern int sys_setregid();

fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
sys_setreuid,sys_setregid };
```

图 1

2. 以 helloword 程序为例（主要是 printf()调用）解释在此 Linux 中系统调用（从 int0x80 开始）的处理过程（包括中断的响应、系统调用参数的传递、具体调用函数（sys-）入口地址的确定、系统调用返回、中断返回等细节）。

答：当用户态进程发起一个系统调用，CPU 将切换到内核态并开始执行一个内核函数。内核函数负责响应应用程序的要求，例如操作文件、进行网络通讯或者申请内存资源等。

（1）调用流程：

我们以一个假设的系统调用 xyz 为例，介绍一次系统调用的所有环节。

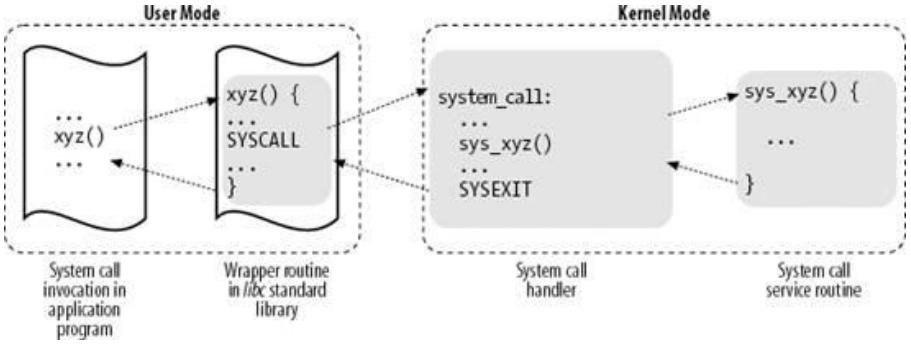

图 2

如上图，系统调用执行的流程如下：

1. 应用程序代码调用系统调用(xyz)，该函数是一个包装系统调用的库函数；
2. 库函数(xyz)负责准备向内核传递的参数，并触发软中断以切换到内核；
3. CPU 被软中断打断后，执行中断处理函数，即系统调用处理函数(system_call)；
4. 系统调用处理函数调用系统调用服务例程(sys_xyz)，真正开始处理该系统调用；

（2）执行态切换

应用程序(application program)与库函数(libc)之间，系统调用处理函数(system call handler)与系统调用服务例程(system call service routine)之间，均是普通函数调用，而库函数与系统调用处理函数之间，由于涉及用户态与内核态的切换，要复杂一些。

Linux 通过软中断实现从用户态到内核态的切换。用户态与内核态是独立的执行流，因此在切换时，需要准备执行栈并保存寄存器。

内核实现了很多不同的系统调用(提供不同功能)，而系统调用处理函数只有一个。因此，用户进程必须传递一个参数用于区分，这便是系统调用号(system call number)。在 Linux 中，系统调用号一般通过 eax 寄存器来传递。

总结起来，执行态切换过程如下：

1. 应用程序在用户态准备好调用参数，执行 int 指令触发软中断，中断号为 0x80；
2. CPU 被软中断打断后，执行对应的中断处理函数，这时便已进入内核态；
3. 系统调用处理函数准备内核执行栈，并保存所有寄存器(一般用汇编语言实现)；
4. 系统调用处理函数根据系统调用号调用对应的 C 函数——系统调用服务例程；
5. 系统调用处理函数准备返回值并从内核栈中恢复寄存器；
6. 系统调用处理函数执行 ret 指令切换回用户态；

printf()函数是建立在系统调用之上，更高层次的库函数，应用程序要输出文字，需调用 write 这个系统调用。int 指令触发软中断 0x80，程序将陷入内核态并由内核执行系统调用。

这个过程中将会首先进入系统调用处理程序（kernel/system_call.s，如图 3）：

```
_system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx      # push %ebx,%ecx,%edx as parameters
    pushl %ebx      # to the system call
    movl $0x10,%edx     # set up ds,es to kernel space
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx     # fs points to local data space
    mov %dx,%fs
    call _sys_call_table(,%eax,4)
    pushl %eax
    movl _current,%eax
    cmpl $0,state(%eax)     # state
    jne reschedule
    cmpl $0,counter(%eax)       # counter
    je reschedule
ret_from_sys_call:
    movl _current,%eax      # task[0] cannot have signals
    cmpl _task,%eax
    je 3f
    cmpw $0x0f,CS(%esp)     # was old code segment supervisor ?
    jne 3f
    cmpw $0x17,OLDSS(%esp)      # was stack segment = 0x17 ?
    jne 3f
    movl signal(%eax),%ebx
    movl blocked(%eax),%ecx
    notl %ecx
    andl %ebx,%ecx
    bsfl %ecx,%ecx
    je 3f
    btrl %ecx,%ebx
    movl %ebx,signal(%eax)
    incl %ecx
    pushl %ecx
    call _do_signal
    popl %eax
3:  popl %eax
    popl %ebx
    popl %ecx
    popl %edx
    pop %fs
    pop %es
    pop %ds
    iret
```

图 3

系统调用处理程序首先检查寄存器 eax 中存放的系统调用号（sys_write 的系统调用号为 4），并在 include/unistd.h（图 4）中检查相应的系统调用号是否存在：

```
#define __NR_setup  0   /* used only by init, to get system going */
#define __NR_exit   1
#define __NR_fork   2
#define __NR_read   3
#define __NR_write  4
#define __NR_open   5
#define __NR_close  6
#define __NR_waitpid    7
#define __NR_creat  8
#define __NR_link   9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir  12
#define __NR_time   13
#define __NR_mknod  14
#define __NR_chmod  15
#define __NR_chown  16
#define __NR_break  17
#define __NR_stat   18
#define __NR_lseek  19
#define __NR_getpid 20
#define __NR_mount  21
#define __NR_umount 22
#define __NR_setuid 23
#define __NR_getuid 24
#define __NR_stime  25
#define __NR_ptrace 26
#define __NR_alarm  27
#define __NR_fstat  28
#define __NR_pause  29
#define __NR_utime  30
#define __NR_stty   31
#define __NR_gtty   32
#define __NR_access 33
#define __NR_nice   34
#define __NR_ftime  35
#define __NR_sync   36
#define __NR_kill   37
#define __NR_rename 38
#define __NR_mkdir  39
#define __NR_rmdir  40
#define __NR_dup    41
#define __NR_pipe   42
#define __NR_times  43
#define __NR_prof   44
#define __NR_brk    45
#define __NR_setgid 46
#define __NR_getgid 47
#define __NR_signal 48
#define __NR_geteuid    49
#define __NR_getegid    50
#define __NR_acct   51
#define __NR_phys   52
#define __NR_lock   53
#define __NR_ioctl  54
#define __NR_fcntl  55
#define __NR_mpx    56
#define __NR_setpgid    57
#define __NR_ulimit 58
#define __NR_uname  59
#define __NR_umask  60
#define __NR_chroot 61
#define __NR_ustat  62
#define __NR_dup2   63
#define __NR_getppid    64
#define __NR_getpgrp    65
#define __NR_setsid 66
#define __NR_sigaction  67
#define __NR_sgetmask   68
#define __NR_ssetmask   69
#define __NR_setreuid   70
#define __NR_setregid   71
```

图 4

然后根据 include/linux/sys.h（图 1）中的 sys_call_table 函数指针数组找到相应的 write 系统调用服务例程（sys_write 函数位于 linux/fs/read_write.c，如图 5）：

```
83  int sys_write(unsigned int fd,char * buf,int count)
84  {
85      struct file * file;
86      struct m_inode * inode;
87
88      if (fd>=NR_OPEN || count <0 || !(file=current->filp[fd]))
89          return -EINVAL;
90      if (!count)
91          return 0;
92      inode=file->f_inode;
93      if (inode->i_pipe)
94          return (file->f_mode&2)?write_pipe(inode,buf,count):-EIO;
95      if (S_ISCHR(inode->i_mode))
96          return rw_char(WRITE,inode->i_zone[0],buf,count,&file->f_pos);
97      if (S_ISBLK(inode->i_mode))
98          return block_write(inode->i_zone[0],&file->f_pos,buf,count);
99      if (S_ISREG(inode->i_mode))
100         return file_write(inode,file,buf,count);
101     printk("(Write)inode->i_mode=%06o\n\r",inode->i_mode);
102     return -EINVAL;
103 }
104
```

图 5

使用寄存器 ebx、ecx、edx 传递参数，执行完毕后在 eax 中保存写入的字节数并向调用程序返回 0；若执行出错则将错误类型码取反存入全局变量 errno，并向调用程序返回-1，然后执行 ret_from_sys_call 结束系统调用，通过 iret 结束中断。系统调用执行完毕后，内核将负责切换回用户态，恢复之前保存的所有寄存器，应用程序继续执行之后的指令。

参考文献：

[1] 赵炯．Linux 内核完全注释修正版 3.0[DB/OL]．2007-06-07．

[2] 小菜学编程．系统调用原理[DB/OL]．https://learn-linux.readthedocs.io/zh_CN/latest/system-programming/syscall/principle.html．