

Linux 系统调用与实例分析

混合 961 曾铮 9630007

混合 961 戴敏雅 9630002

目 录

| | | |
|-----|-----------------------|----|
| 一 | 系统调用概述..... | 1 |
| 二 | Linux 系统调用流程..... | 2 |
| 2.1 | Linux 系统调用的中断机制..... | 2 |
| 2.2 | 相关的数据结构及函数..... | 2 |
| 2.3 | Linux 系统调用的流程..... | 7 |
| 三 | 实例分析 — fork 系统调用..... | 8 |
| 3.1 | 系统调用 fork 简介..... | 8 |
| 3.2 | 系统调用 fork 的设置..... | 8 |
| 3.3 | 函数 do_fork() 的分析..... | 9 |
| 四 | 讨论..... | 15 |

一. 系统调用概述

从一般用户的观点，操作系统是用户与计算机硬件系统之间的接口；从资源管理观点，操作系统是计算机系统资源的管理者。用户在操作系统的帮助下能够方便、快捷、安全、可靠的操纵计算机和运行自己的程序。用户可以通过以下的两种方式使用计算机：

1. 命令方式。这是指由操作系统提供了一组联机命令（语言），用户可通过键盘键入有关的命令，来直接操纵计算机系统。
2. 系统调用方式。操作系统提供了一组系统调用，用户可在应用程序中通过调用相应的系统调用来操纵计算机系统。系统调用是用户程序与 kernel 的接口。

以下是对系统调用的简介。系统调用命令是操作系统为满足用户所需的功能和保证程序的正常运转事先编制好的具有特定功能的例行子程序。每当用户在程序中需要操作系统提供某种服务时，便可利用一条系统调用命令，去调用系统过程。它一般运行在核心态；可通过中断进入，返回时通常需要重新调度。

Linux 系统调用由 0x80 号中断进入系统调用入口，使系统由用户态转为核心态。通过使用系统调用表保存系统调用服务函数的入口地址，转入特定的例行子程序去执行，完成用户当前所需要的服务来实现。

本文通过对 Linux 的一般系统调用过程分析和创建子进程的系统调用 fork 的分析来阐述 Linux 系统调用过程。

二. Linux 系统调用流程

2.1 Linux 系统调用的中断机制

linux 系统是通过中断处理来实现系统调用的。设置中断向量时，把 0x80 号中断向量和系统调用处理程序联系起来。该处理程序的功能是：做常规的现场保护后，按系统调用功能号找到对应的系统调用函数的地址，转到该函数中去执行。在用户程序中，安排一句系统调用命令(该命令已由_syscallN(parametres)展开，展开部分有指令:INT \$0x80),当程序执行到这条命令时，就发生中断，系统由用户态转为核心态，操作系统的系统调用处理程序得到控制权，它在保存所有寄存器和确定该系统调用合法后，将根据用户提供的系统调用名，利用 syscall number 确定系统调用的功能号，根据 sys_call_table 找到例行系统调用函数的入口地址，转入对应的系统调用函数执行。执行完毕后，返回到用户程序的断点继续执行。

2.2 相关的数据结构及函数

2.2.1 设定 0x80 号中断

系统启动后所进行的一系列初始化工作中较重要的一部分在 start_kernel() 函数中进行，各种中断服务程序入口在其中通过调用 trap_init() (arch/i386/kernel/traps.c 中) 被设置，与系统调用相关的是：set_system_gate(0x80,&system_call)

宏 set_system_gate() (“include/asm-i386/system.h” 中)：

```
#define set_system_gate(n, addr) \
    _set_gate(&idt[n], 15, 3, addr)
```

宏 _set_gate() (include/asm-i386/system.h 中，作用是使 addr 地址值置入 gate_addr 的地址值所指向的内存单元中，使中断向量表中的 0x80 项保存了中断服务程序 system_call 的入口地址)：

```
#define _set_gate(gate_addr, type, dpl, addr) \
    __asm__ __volatile__ (“movw %%dx, %%ax\n\t” \
    “movw %2, %%dx\n\t” \
    “movl %%eax, %0\n\t” \
    “movl %%edx, %1” \
    : “=m” (*((long *) (gate_addr))), \
    “=m” (*(1+(long *) (gate_addr))), \
    : “i” ((short) (0x8000+(dpl<<13)+(type<<8))), \
    “d” ((char *) (addr)), “a” (KERNEL_CS << 16) \
    : “ax”, “dx”
```

初始化中断向量表(其中 0x80 项保存了中断服务程序 `system_call` 的入口地址):

| | |
|-------|-----------------------------|
| 0 | divide_error |
| 1 | debug |
| 2 | nmi |
| 3 | int 3 |
| 4 | overflow |
| 5 | bounds |
| 6 | Invalid_op |
| 7 | device_not_available |
| 8 | double_fault |
| 9 | Coprocessor_segment_overrun |
| 10 | Invalid_TSS |
| 11 | Segment_not_present |
| 12 | Stack_segment |
| 13 | General_protection |
| 14 | Page_fault |
| 15 | Spurious_interrupt_bug |
| 16 | Coprocessor_error |
| 17 | Alignment_check |
| 18-48 | reserved |
| 128 | System_call |

2.2.2 数据结构

介绍系统调用主要的两种数据结构：系统调用表和寄存器帧结构。

先分析系统调用表 `sys_call_table` (`/arch/i386/Entry.S` 中):

```
ENTRY(sys_call_table)
    .long SYMBOL_NAME(sys_setup)      /* 0 */
    .long SYMBOL_NAME(sys_exit)
    .long SYMBOL_NAME(sys_fork)
    :
    :
    :
    .long SYMBOL_NAME(sys_mremap)
    .long 0, 0
    .long SYMBOL_NAME(sys_vm86)      /* 166 */
    .space (NR_syscalls-166)*4
```

表中保存了所有 Linux 基于 Intel x86 系列体系结构的计算机的 166 个系统调用入口地址（其中 3 个保留，Linux 开辟的系统调用表可容纳 `NR_syscalls` (`/include/linux/sys.h` 中定义的值为 256 的宏) 项），其中每项都被说明成 `long` 型。可根据特定系统调用在表中偏移量找到对应的系统调用代码。`.space(NR_syscalls-166)*4` 表示所剩的可供用户自己添加系统调用的空间。

在 `unistd.h` 中为每一种系统调用分配了一个唯一的编号(syscall number)，相应于系统调用在 `sys_call_table` 中的偏移量。如：

```
#define __NR_setup      0
#define __NR_exit       1
#define __NR_fork       2
#define __NR_read       3
:
:
:
```

寄存器帧结构：`pt_regs`（`include/asm-i386/ptrace.h` 中），该帧结构与系统调用时压入堆栈的寄存器的顺序保持一致，用来在系统跟踪、系统调用返回时传递参数，定义如下：

```
struct pt_regs {
    long ebx;
    long ecx;
    long edx;
    long esi;
    long edi;
    :
    :
    :
};
```

在 `entry.s` 中压入堆栈的一帧和一个 `pt_regs` 结构体相互对应；这样，只要 `pt_regs` 结构体的首地址是该帧的帧顶，则 `EAX(%esp)` 与 `regs->eax` 将指向同一内存单元：

如：（`entry.s` 中）

```
/*
 * 0(%esp) - %ebx      EBX=0X00
 * 4(%esp) - %ecx      ECX=0X04
 * 8(%esp) - %edx      EDX=0X08
:
:
:
*/
```

2.2.3 相关函数

1) 系统调用函数 `sys_NAME(parameters)`（`syscall.c`, `sys.c`, `time.c` 等文件中）

系统调用时可根据 `sys_call_table` 调用这些以 `sys_` 开头的系统调用函数。

如：

```
asmlinkage int sys_fork(struct pt_regs *regs) {...}
```

2) 一些有关的宏

`_syscallN`(type, name, type1, arg1, type2, arg2.....) (include/asm-i386/unistd.h 中)
(N=0~5, 表示系统调用的参数个数, 相应的宏的参数为 $2*N+2$ 个), 在该文件中, 共定义了 6 个宏。

以宏 `_syscall12` 即系统调用的参数个数等于 2 为例:

```
#define _syscall12(type, name, type1, arg1, type2, arg2) \
type name(type1 arg1, type2 arg2) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
: "=a" (__res) \
: "0" (__NR_##name), "b" ((long)(arg1)), "c" ((long)(arg2))); \
    if (__res >= 0) \
        return (type) __res; \
    errno = -__res; \
    return -1; \
}
```

宏指令的第一个参数说明产生函数的返回值的类型, 第二个参数为产生函数的名称。参数列表中若还有参数, 则第 $2i$ 个参数是系统调用函数的第 i 个参数的类型, 第 $2i+1$ 个参数是系统调用函数的第 i 个参数。

该宏以内联汇编形式实现, 每次调用宏 `_syscallN` (type, name, type1, arg1, type2, arg2.....), 该宏扩展成返回值类型为 type, 函数名为 name 的函数, 并通过 name 和 NR_name (syscall numbers) 建立联系, 从而确定该调用在 sys_call_table 中的偏移量, 再根据 sys_call_table 确定系统调用函数的入口地址。其中, 用户态到核心态的转换是由宏扩展后的 int \$0x80 指令完成的。如:

`_syscall0(int, fork)` 即 `int fork()`

传给系统调用的参数和系统调用的返回值是存放在 CPU 寄存器中的。所以要求系统调用的参数的数据类型不能超过 4 个字节; 并且最多可传送 5 个参数, 因此只定义了 `_syscall0~5` 这 6 个不同的 `_syscallN()`。

语句 `"=a" (__res)` 指明返回参数 (即 `__res`) 使用 `eax` 寄存器。在语句 `"0" (__NR_##name), "b" ((long)(arg1)), "c" ((long)(arg2))`; 中: `"0" (__NR_##name)` 中将 `__NR_` 与参数 name 串接起来, 形成的标志符存入 `eax` 寄存器, 作为区别系统调用类型的参数。随后将参数 `arg1, arg2` 分别传给寄存器 `ebx` 和 `ecx`, 在 `"_syscallX"` 宏中, 约定五个参数分别与五个寄存器对应:

`eax`: 名为 name 的系统调用在 sys_call_table 中的偏移量
`ebx`: `arg1`
`ecx`: `arg2`
`edx`: `arg3`

esi : arg4

edi : arg5

在系统调用返回时，syscallN()都检验返回值是否为负，如果是（不合法）把_errno 置为该返回值的绝对值，并返回-1，否则直接返回该返回值。

宏中的汇编指令“int \$0x80”使程序流转入“system_call”。

3)中断入口 system_call() (/arch/i386/entry.s 中)

system_call 是所有系统调用的入口。它的功能是：保存所有寄存器，检验是否是合法的系统调用，根据_sys_call_table 中的偏移量把控制权转给真正的系统调用代码，系统调用完毕后调用_ret_from_sys_call(),返回到用户空间。下面解释关于它的一些重要指令：

调用宏过程 SAVE_ALL 保护现场（保存寄存器）。这样保存的一帧寄存器该过程所要传递的 pt_regs 结构类型的参数结构一致。cmpl\$(NR_syscalls),%eax 比较 NR_syscalls 与 eax 的大小，如果 eax 大于或等于 NR_syscalls，表明指定的系统调用函数错误，jae ret_from_sys_call 使系统调用直接返回。再执行 movl SYMBOL_NAME(sys_call_table)(,%eax,4),%eax，以 sys_call_table 为基地址，eax 寄存器中的内容（系统调用的序号）乘以 4（long 型字节数）为偏移量，即得到所需调用的系统调用函数的入口地址，将其存入寄存器 eax。然后判断寄存器 eax 值是否为 0，若是，表明出错，直接返回。语句：

```
#ifdef __SMP__
GET_PROCESSOR_OFFSET(%edx)
movl SYMBOL_NAME(current_set)(,%edx),%ebx
#else
movl SYMBOL_NAME(current_set),%ebx
#endif
```

使寄存器 ebx 指向当前进程。语句：

```
andl $~CF_MASK,EFLAGS(%esp)
movl %db6,%edx
movl %edx,dbgreg6(%ebx)
```

将 CF 清为 0，并保存当前调试信息于 task_struct 结构中。在 entry.S 中，定义了一系列宏，用来表示当前进程的信息，它们是：

```
state          = 0
counter        = 4
priority       = 8
signal         = 12
blocked        = 16
flags          = 20
dbgreg6        = 52
dbgreg7        = 56
exec_domain    = 60
```

语句 `testb $0x20, flags(%ebx)` 检测当前进程是否正跟踪系统调用，如果不是，直接调用所选系统调用函数：`call *%eax`；如判断当前进程正处于跟踪系统调用状态（`current->flags&PF_TRACESYS==0`）就调用函数体 `syscall_trace()`（`/arch/i386/kernel/ptrace.c` 中），使当前进程状态转为 `TASK_STOPPED`，将该进程转入睡眠状态。然后从堆栈中弹出原来的 `eax` 值，再重新设置系统调用函数的偏移量，调用实现相应系统调用的函数，语句为：

```
call SYMBOL_NAME(syscall_trace)
movl ORIG_EAX(%esp), %eax
call *SYMBOL_NAME(sys_call_table)(, %eax, 4)
```

之后，就进入了系统调用服务函数，该函数返回以后进入 `ret_from_sys_call`，处理一些系统调用返回前应该处理的事情，如检测 `bottom half` 缓冲区，判断 CPU 是否需要重新调度等，之后，系统调用返回。

2.3 Linux 系统调用的流程

系统启动时，经过引导和实模式下的初始化，进入保护模式下的核心初始化，执行 `head.s`。其中的 `startup_32` 代码段中调用 `setup_idt`。`Setup_idt` 的功能是建立一个 256 项的空的中断向量表。

接着，系统转入 `start_kernel()`（`/usr/src/linux/init/main.c` 中）模块，在该模块中，调用 `trap_init()`（`/usr/src/linux/kernel/traps.c` 中）初始化中断向量表，其中使系统调用 `system_call` 项成为 `0x80` 号中断的中断服务程序。

建立及初始化中断向量表的流程：

```
startup_32
call setup_idt
↓
setup_idt
↓
start kernel()
↓
trap_init()
```

其中 `trap_init()` 含：

```
set_call_gate(&default_ldt, lcall7)
set_trap_gate(0, &divide_error)
:
:
:
for(i=18; i<48; i++)
set_trap_gate(i, &reserved)
set_system_gate(0x80, &system_call)
```

用户必须在程序中以系统调用命令（以调用函数的形式给出）进行系统调用，将该命令由相应的宏 `_syscallN (.....)` 展开。宏指令本身将在预处理时扩展为指定名称的函数（宏 `_syscall (.....)` 的第二个参数与系统调用命令时所用函数名一致）。执行由宏指令扩展的函数，将系统调用参数值存入相应的 CPU 寄存器，然后执行 `int $0x80` 中断处理指令，进入核心态，入口地址为 `&system_call`。执行 `system_call`，保存寄存器，检查调用是否合法，如合法，根据 `_sys_call_table` 中的偏移量转入相应的系统调用代码。系统调用结束时，调用 `_ret_from_sys_call`。对内核函数的返回值进行检查后返回用户态，并返回相应的值。系统调用通过可屏蔽中断 `int 0x80` 调用进行。

系统调用流程：

```
_syscallN (type, name, type1, arg1, type2, arg2.....);  
↓  
type  name(type1 arg1, type2  arg2.....);
```

三. 实例分析 — fork 系统调用

3.1 系统调用 fork 简介

`fork` 系统调用的功能是创建新进程（调用 `fork()` 的进程的子进程），该子进程是父进程的一个拷贝，只有进程号和其它少数参数不同而已。子进程将继承父进程的实际和有效的 `uid` 和 `gid`；所有父进程打开的文件传给子进程；子进程与父进程具有相同的 `umask`；子进程继承当前的工作目录；一旦子进程建立，则父进程和子进程都在 `fork` 内部继续执行。如果调用成功，`fork` 系统调用对父进程返回新生成的子进程的进程标识号 `pid`，对子进程返回 0；否则，将出错原因存入 `error` 变量，并向父进程返回 -1。产生的出错原因有：

1. `ENOMEM`: `fork` 为自己的存在申请内存空间失败。
2. `EAGAIN`: `fork` 为子进程的 PCB 的数据项分配内存空间失败。

`fork` 与 `clone` 类似，只是 `clone` 是创建一个与父进程完全相同的新进程，两者的 `pid` 也相同。

3.2 系统调用 fork 的设置

在 `include/asm-i386/unistd.h` 中进行系统调用的设置（包括 `fork`）；该系统调用 `fork` 的设置使用的宏应为：`_syscall0(int, fork)`，因为 `fork()` 是不带参数的，且返回类型为 `int`。

在 `include/asm-i386/unistd.h` 中，`fork` 的内联宏语句为：`static inline _syscall0(int, fork)`。这样，在调用 `fork` 时，系统将调用宏指令 `_syscall0(int, fork)`，

进而调用 0x80 号中断，此时寄存器 `eax` 中的值为 `__NR_fork(=2)`，作为参数传给 `int $0x80`。

调用中断 `int $0x80` 后，`System_call` 在保存所有寄存器，检验是否是合法的系统调用后，用 `eax` 中的值 (`__NR_fork`) 乘 4 得到系统调用表 (`sys_call_table`) 中的偏移，找到入口：`.long SYMBOL_NAME(sys_fork)`

接着转向函数 `sys_fork()` (`arch/i386/kernel/process.c` 中)：

```
asmlinkage int sys_fork(struct pt_regs regs)
{
    return do_fork(SIGCHLD, regs.esp, &regs);
}
```

`SIGCHLD` (`signal.h` 中) 是一种信号类型，作为 `do_fork` 的第一个参数 `clone_flags` 指明 `do_fork()` 函数应创建一子进程：

```
#define SIGCHLD    17
```

`sys_fork()` 将类型为 `struct pt_regs` (寄存器帧) 的 `regs` 的地址作为参数传递给 `do_fork()`，并且通过寄存器：`regs.esp` 传递了栈顶指针。由于系统调用是通过寄存器传递参数的，且 `system_call` 已将所有的 CPU 寄存器保存在堆栈中，根据 2.2.2 的讨论，只要 `regs` 首址为当前堆栈的栈顶指针，就可以通过 `regs` 取压入堆栈的各寄存器值 (参数值)。

实际上，`fork` 系统调用最终是由 `do_fork()` 函数完成的 (`clone` 也是借助 `do_fork` 完成的，不同之处在于传入 `do_fork()` 的参数不同)。`do_fork()` 在 `task` 数组中找到空闲位置，继承父进程现有资源，初始化进程时钟、信号、时间等数据。

3.3 函数 `do_fork()` 的分析

`int do_fork(unsigned long clone_flags, unsigned long usp, struct pt_regs *regs)` (`linux/kernel/fork.c` 中)

参数：`clone_flags`：unsigned long 类型，区分 `sys_clone` 和 `sys_fork` 的标志。

`Usp`：unsigned long 类型，用来传递 `sp` 的值。在 `sys_fork` 系统调用时，将 `regs.esp` 传给 `Usp`，通过 `copy_thread` 将其赋给子进程的 `esp`。

`Regs`：指向 `struct pt_regs` 的指针类型。

返回值：`int` 类型。如果调用成功，对父进程返回新生成的子进程的进程标识号 `pid`，对子进程返回 0；否则，将出错原因存入 `error` 变量，并向父进程返回 -1。

1. 为新进程分配空间

`do_fork()` 函数开始就将可能返回的 `error` 初始值置为 `-ENOMEM`，假设内存已被用完。然后，才进入主流程。

```
p = (struct task_struct *) kmalloc(sizeof(*p), GFP_KERNEL);
if (!p)
    goto bad_fork;
```

```

new_stack = alloc_kernel_stack();
if (!new_stack)
    goto bad_fork_free_p;

error = -EAGAIN;
nr = find_empty_process();
if (nr < 0)
    goto bad_fork_free_stack;

```

先调用 `kmalloc` 为进程申请内存空间，`GFP_KERNEL` 表示允许申请不到内存时转入睡眠状态。如果申请内存失败的话，将返回 `NULL`。这时，`do_fork()` 函数转入 `bad_fork` 执行，直接返回内存已被用完的出错信息。

不然，`do_fork()` 函数调用宏 `alloc_kernel_stack()`，分配进程所需的堆栈，如果申请失败，转入 `bad_fork_free_p` 执行。

否则，表示 `ENOMEM` 的危险已经过去；

执行 `error = -EAGAIN`；假设 `fork` 为子进程的 PCB 的数据项分配内存空间失败。

Task 数组（/kernel/sched.c 中）：

```
struct task_struct *task[NR_TASKS];
```

其中，`NR_TASKS` 的值为 512，它规定了系统可运行的最大进程数，用来存放所有进程 PCB 的指针。`find_empty_process()` 就是在不超过系统规定的最大进程数、资源限定允许的条件下在 `task` 数组中找是否有空闲的区域，有则把数组下标作为该函数的返回值，否则返回错误代码 `-EAGAIN`。在 `do_fork` 中，把 `nr` 作为 `find_empty_process` 的返回值，若为正，说明 `find_empty_precess` 正常返回，且 `nr` 为 `task` 数组中空闲 PCB 的指针；若为负，说明无法增加新的进程，进入错误处理程序 `bad_fork_free_stack`，返回值为 `-EAGAIN`。

2. 新进程初始化及有关参数设置

#1.

```
*p = *current;
```

将父进程的内容赋给子进程，这时，子进程完全继承了父进程的特征，接下来根据 `clone_flags` 对子进程数据成员进行初始化。

#2.

```
if (p->exec_domain && p->exec_domain->use_count)
    (*p->exec_domain->use_count)++;
```

```
if (p->binfmt && p->binfmt->use_count)
    (*p->binfmt->use_count)++;
```

分别将全局执行域结构和全局执行文件格式结构所对应的 use_count 加一，表示进程数增一。

#3.

```
p->did_exec = 0;
子进程正在创建，未被执行过
p->swappable = 0;
进程刚创建，暂不可调出内存
```

#4.

```
p->kernel_stack_page = new_stack;
*(unsigned long *) p->kernel_stack_page = STACK_MAGIC;
首先把核心栈所在物理页的基地址设为核心栈首址，然后把 STACK_MAGIC 与核心栈所在页的 unsigned long 形式联系起来。其中，STACK_MAGIC (kernel.h 中):
#define STACK_MAGIC 0xdeadbeef
*(unsigned long *) p->kernel_stack_page = STACK_MAGIC;为了在进程退出时，检验 stack_page 是否出错。
```

#5.

```
p->state = TASK_UNINTERRUPTIBLE;
设置新进程刚创建时状态为 TASK_UNINTERRUPTIBLE，表示本进程将被置于等待队列中，由于资源未分配好，因此置为不可中断，使其待资源有效、所有信息都设置好后才被唤醒，不可由其它进程通过信号唤醒。
```

#6.

```
p->flags &= ~(PF_PTRACED|PF_TRACESYS|PF_SUPERPRIV);
关闭 PF_PTRACED,PF_TRACESYS,PF_SUPERPRIV 信号，拒绝新建进程具有超级用户特权或被跟踪
p->flags |= PF_FORKNOEXEC;
开启 PF_FORKNOEXEC 信号，表示新建进程还没执行。
```

#7.

```
p->pid = get_pid(clone_flags);
得到新进程的 pid。get_pid() 函数先判断是否为 clone 系统调用（根据 clone_flags），若不是，则在 1 ~ 0xffff8000 中找到不等于任何进程的 pid, pgrp 或 session 的最小的数，并把这个最小数作为返回值。
```

#8.

```
p->next_run = NULL;
```

```
p->prev_run = NULL;
```

由于新产生的进程的状态还是为 TASK_UNINTERRUPTIBLE，因此不将其放入就绪队列，其 next_run 和 prev_run 均置为 NULL。

```
p->p_pptr = p->p_opptr = current;
```

```
p->p_cptr = NULL;
```

新进程的 parent 和 original parent 置为当前进程 Current，child 置为空。

#9.

```
init_waitqueue(&p->wait_chldexit);
```

为新进程的子进程初始化等待队列。

#10.

```
p->signal = 0;
```

表示现在新建进程还没有收到信号。

#11.

```
p->it_real_value = p->it_virt_value = p->it_prof_value = 0;
```

```
p->it_real_incr = p->it_virt_incr = p->it_prof_incr = 0;
```

初始化用于进程计时的数据项，置为 0。表示现在新建进程尚未定时。其中 it_real_value, it_real_incr 与 jiffies 保持一致，表示真实时间；it_virt_value, it_virt_incr 用于虚拟软件实时，它仅在进程运行时有效，因此，该数据项用于进程内计时，当时间到时，发送信号。参见 do_it_virt () (/kernel/sched.c 中)。

```
init_timer(&p->real_timer);
```

```
p->real_timer.data = (unsigned long) p;
```

初始化 timer_list 类型的 real_timer，使其 data 与 p 相等，作为区分 timer 的标志。

```
p->utime = p->stime = 0;
```

```
p->cutime = p->cstime = 0;
```

分别将进程用户态时间总和，核心态时间总和，子进程用户态时间总和，子进程核心态时间总和均初始化为零。

```
p->start_time = jiffies;
```

将当前进程的建立时间置为 jiffies。

#12.

```
#ifdef __SMP__
```

```
p->processor = NO_PROC_ID;
```

```
p->lock_depth = 1;
```

```
#endif
```

设置多处理器信息。

#13.

```
task[nr] = p;
```

将 `p` 的值赋给 `task` 数组的第 `nr` 项，使新进程的 PCB 进入 `PCB_SET` 数组。

```
SET_LINKS(p);
```

将新进程与初始进程相关联。

```
nr_tasks++;
```

当前进程数加 1。

```
error = -ENOMEM;
```

假设错误为内存不够。

#14.

```
if (copy_files(clone_flags, p))
```

```
    goto bad_fork_cleanup;
```

```
if (copy_fs(clone_flags, p))
```

```
    goto bad_fork_cleanup_files;
```

```
if (copy_sighand(clone_flags, p))
```

```
    goto bad_fork_cleanup_fs;
```

```
if (copy_mm(clone_flags, p))
```

```
    goto bad_fork_cleanup_sighand;
```

根据 `clone_flags` 判断，进程是否由 `sys_clone` 产生，如果是，则不进行拷贝，只是把当前进程中相关数据结构成员的 `count` 值加 1；否则，拷贝所有进程信息（文件信息（`copy_files`），目录信息（`copy_fs`），信号信息（`copy_sighand`）和内存信息（`copy_mm`））。只有当父进程或子进程要对虚存进行写操作时，才给予进程的 `mm` 所指向的数据结构分配内存，并将父进程的 `mm` 所指向的数据内容拷贝到子进程的 `mm` 上。这些函数的正常返回值均为 0，如果返回值非 0，就转入到相应的异常处理程序中去。

#15.

```
copy_thread(nr, clone_flags, usp, p, regs); （/arch/i386/kernel/process.c 中）
```

对子进程 PCB 的 `tss`（任务状态段）和 `ldt`（进程局部描述符表的指针）进行设置。

其中，`tss` 包括各种通用寄存器。

其中：

```
childregs = ((struct pt_regs *) (p->kernel_stack_page + PAGE_SIZE)) - 1;
```

```
p->tss.esp = (unsigned long) childregs;
```

子进程任务状态段的 `esp` 指向当前栈顶。

```
p->tss.eip = (unsigned long) ret_from_sys_call;
```

将 `ret_from_sys_call` 赋予子进程任务状态段的 `eip`。当子进程被唤醒时，执行 `ret_from_sys_call`，由核心态返回到用户态。

```
*childregs = *regs;
```

使子进程的寄存器帧的内容与当前进程的同。

```
childregs->eax = 0;
```

子进程从 fork() 的返回值为 0。

```
childregs->esp = esp;
```

把当前进程的 esp 赋给子进程的 esp，使父子进程共用一个堆栈。

3. 唤醒新进程，返回 system call

新生成的进程的参数全部设置完毕，接下来分配内存，用来保存与新进程相关的文件系统，内存页面，信号处理程序等：

```
* p->swappable = 1;
```

新进程已完成初始化，可以换出内存。

```
* p->exit_signal = clone_flags & CSIGNAL;
```

设置系统强行退出时发出的信号，其中，CSIGNAL (sched.h 中)：

```
#define CSIGNAL    0x000000ff
```

为进程终止时须发的信息。

```
* p->counter = (current->counter >>= 1);
```

子进程的 counter 为父进程的一半，即其时间片只有父进程的一半。

```
* wake_up_process(p);
```

唤醒新进程，置其状态为 TASK_RUNNING，如果它不在就绪队列中，把它加入。

```
* ++total_forks;
```

进程数增一。

```
* return p->pid;
```

返回至 system_call，返回值为新进程的 pid。

4. 其它

```
bad_fork_cleanup_sighand:
```

```
    exit_sighand(p);
```

```
bad_fork_cleanup_fs:
```

```
    exit_fs(p);
```

```
bad_fork_cleanup_files:
```

```
    exit_files(p);
```

错误处理函数 exit_sighand(p)，exit_fs(p)，exit_files(p) 三者结构类似，分别把子进程的 sighand，fs，files 置为 NULL，如果所对应的 count 为零，则释放该域所占据的空间。具体代码：

```
bad_fork_cleanup:
```

```
    if (p->exec_domain && p->exec_domain->use_count)
```

```
        (*p->exec_domain->use_count)--;
```

```
    if (p->binfmt && p->binfmt->use_count)
```

```
        (*p->binfmt->use_count)--;
```

```
    task[nr] = NULL;
```

```

        REMOVE_LINKS(p);
        nr_tasks--;
bad_fork_free_stack:
        free_kernel_stack(new_stack);
bad_fork_free_p:
        kfree(p);
bad_fork:
        return error;

```

函数 do_fork() 的流程:



以上只是粗略的流程，关于各步的出错处理，参见上文。

四. 讨论

系统调用表中留有空项，故用户可根据需要，添加系统调用。首先，编写调用函数代码，以 sys_XXX 为名加到 /usr/src/linux/kernel/sys.c 中，其次，在 /usr/src/linux/include/asm/i386/unistd.h 中加上 #define __NR_XXX N (N 为其偏移量，(167~256))，在 /usr/src/linux/arch/i386/kernel/entry.s 中加上

```
.long_sys_XXX
```

`.space(NR_syscall-N)*4`

调用时只需在用户子程序开头添加宏 `_syscallN(type,name, type1,agr1...)`，且 `include` 两个文件 `unistd.h` 及 `entry.s`，即可调用。

**因时间仓促及设备限制，加之实验是共同分析代码完成的，
故只写了一份报告，望老师谅解！**