沈阳航空航天大学

计算机学院

**实验报告**

|  |  |
| --- | --- |
| 课程名称 | 操作系统 |
| 专 业 | 物联网工程 |
| 班 级 | 2234280102 |
| 学 号 | 223428010210 |
| 学生姓名 | 陈梓欣 |
| 指导教师 | 宗传玉 |
| 实验时间 | 周五3-4节 |
| 实验地点 | 工训楼310 |

# 实验名称

进程的创建

# 二、实验目的

1. 理解fork系统调用工作原理,使用fork创建子进程。
2. 理解execve系统调用工作原理，使用execve加载执行新程序。
3. 调试跟踪fork和execve系统调用函数的执行过程。。

# 三、实验内容和要求

1、按照《Linux内核实验教程》 实验五‘进程的创建’（66页-77页）中的实验步骤，调试跟踪fork和execve系统调用执行过程。

2、实验需要掌握的知识点

2.1 Linux进程管理概述

2.2创建进程实验步骤

2.2.1 准备实验

启动 Engintime Linux Lab。

使用“Linux011 应用程序”模板新建一个 Linux011 应用程序项目。项目名称可以是“linuxapp”。

2.2.2 调用 fork 函数创建子进程

编写一个 Linux011 应用程序程序，在程序中调用 fork 函数创建子进程，并分析程序运行的结果。步骤如下：

（1）编辑 LinuxApp.c 文件，将 main 函数修改为如下的代码：

int main( int argc, char \* argv[] )

{

int pid;

printf( "PID:%d hello world\n" , getpid() );

pid = fork();

if( pid != 0 )

{

printf( "PID:%d parent process\n", getpid() );

}

else

{

printf( "PID:%d child process\n", getpid() );

}

printf( "PID:%d exit\n", getpid() );

return 0;

}

其中的 getpid 函数是一个系统调用函数，返回当前进程的进程号。

（2）按 F7 生成项目。

（3）按 F5 启动调试

（4）将生成的可执行文件从软盘 B 拷贝到硬盘，命令如下：

mcopy b:linuxapp.exe app

注意：Linux 应用程序项目使用其项目名称（即“项目管理器”中根节点的名称） 做为生成的可执行文件的名称，然后添加了 EXE 后缀。所以请读者根据可执行文件的实际名称编写上面的命令。读者也可以在生成项目后在“项目管理器”窗口中双击软盘 B 确认一下可执行文件的名称。

（5）给 app 文件添加可执行权限，命令如下：

chmod +x app

（6）执行“sync”命令，将对文件的更改保存到磁盘。

（7）查看可执行文件 app 的信息，

ls –l app

将app 文件的大小记录下来，在后面调试跟踪fork 函数的执行过程时会用到此值。

（8）运行可执行文件 app，分析运行结果。

fork 是一个系统调用函数。该系统调用函数在执行时，会在进程表中创建一个与调用此函数的进程（父进程）几乎完全一样的新的进程表项（子进程），子进程与父进程执行同样的代码，但该子进程拥有自己的数据空间和环境参数。在 fork 函数的返回位置处，父进程将恢复执行，而子进程也从相同的位置开始执行。在父进程中，调用fork 返回的值是子进程的进程标识号 PID，而在子进程中 fork 函数返回的值是 0（这正是 fork 的神奇之处，调用一次，返回两次）。通常会在分支语句中使用 fork 函数的返回值，从而使父进程和子进程开始运行不同的代码。

在父进程中可以调用 wait 函数或waitpid 函数阻塞父进程，直到子进程退出后才从这 2 个函数中返回。这两个函数的原型在 include/sys/wait.h 文件中定义如下：

pid\_t wait( pid\_t \* wait\_loc )

pid\_t waitpid( pid\_t pid, pid\_t \* wait\_loc, int options )

按照下面的步骤继续修改程序：

（1）在“printf( “PID:%d parent process\n”, getpid() );”语句前，添加一行语句：

wait(NULL);

（2）生成项目，启动调试，从 B 盘复制应用程序后执行，观察结果与之前有何不同。

（3）练习使用 waitpid 函数替换 wait 函数。然后再练习在子进程中使用 getppid函数得到其父进程的 ID，并打印输出到屏幕上。

2.2.3查看父进程与子进程的运行轨迹

在前面的练习中，读者可以通过父进程和子进程在屏幕上打印输出的信息来间接的观察到父进程和子进程运行的情况，接下来，读者可以使用 Linux Lab 提供的“进程运行轨迹” 窗口，更加直观、形象的观察到父进程与子进程的运行轨迹。步骤如下：

（1）再启动一个 Linux Lab，使用“Linux011 Kernel”模板新建一个 Linux011 Kernel项目。

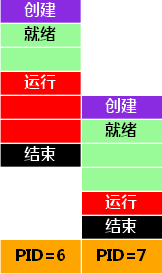
（2）用之前创建的 Linux011 应用程序项目的硬盘镜像文件 harddisk.img 覆盖当前Linux011 Kernel 项目的 harddisk.img 文件，这样就可以使用之前生成的 app 文件了。

（3）为了方便观察父进程和子进程的运行轨迹，需要在进程结束的位置添加一个断点。请读者在 kernel/exit.c 文件的第 166 行添加一个断点即可，这里就是一个进程在结束运行后，让调度程序选择其他进程继续运行的代码。

（4）按 F5 启动调试。在 Linux 操作系统启动完毕之前会命中刚刚添加的断点，继续按

F5 运行即可，直到 Linux 启动完毕。

（5）输入命令 app 后，父进程和子进程在结束时都会命中此断点，所以再第一次命中时， 可以按 F5 继续运行，第二次命中时，刷新“进程运行轨迹”窗口，查看父进程和子进程的运行轨迹（如图 1 所示 ）。



**图1. 父进程和子进程的运行轨迹**

请读者在父进程中调用 wait 函数等待子进程结束后，再按照上面的步骤查看父进程和子进程的运行轨迹，并将图片保存下来。

2.2.4调试跟踪 fork 函数的执行过程

为了调试跟踪在 Linux 应用程序中调用 fork 函数时的执行过程（在 Linux 内核中当然也会调用 fork 函数，但是为了得到更好的实验效果，这里重点研究在应用程序中调用 fork 函数的情况），需要在系统内核中添加一个条件断点，从而确保只有在一个指定的应用程序中调用 fork 函数时，此断点才会被命中，步骤如下：

（1）继续使用在5.2..3 中新建的Linux 内核项目，删除所有断点。

（2）打开 kernel/system\_call.s 文件，在第 102 行添加一个条件断点，条件为：

$eax==2 && current->executable->i\_size==文件大小

“文件大小”就是之前记录的应用程序可执行文件 app 的大小，需要读者根据实际的文件大小进行替换。这样就只有在执行 app 中的 fork 函数时，才会命中此条件断点。“$eax==2”中的 2 是fork 函数的系统调用号。current 是一个全局变量（在 kernel/sched.c 文件的第 109 行定义），总是指向当前正在运行进程的进程控制块。executable 是进程控制块中保存的用于创建此进程的可执行文件的 i 节点，其中保存了文件的一些重要信息，例如 i\_size 就是文件的大小，所以通过指定文件大小，就可以准确指定一个可执行文件。

添加条件断点的方法是，首先在源代码中添加一个断点，然后选择“调试”菜单“窗口”中的“断点”，打开“断点”窗口，在刚刚添加的断点上点击鼠标右键， 选择菜单中的“条件”，然后就可以在断点条件对话框的编辑控件中使用 C 语言编写一个布尔表达式，当此布尔表达式的值为 true 时，此断点才会被命中。注意， 如果启动调试后，条件断点上有黄色叹号图标，说明读者编写的布尔表达式有语法错误，只需将表达式修改正确即可。

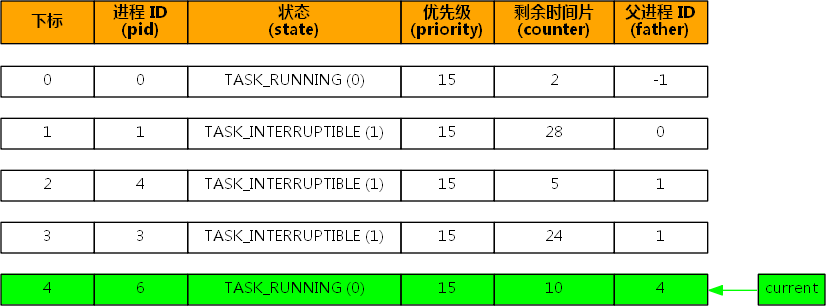
（3）按 F5 启动调试（注意，由于添加了一个条件断点，需要调试器频繁验证条件是否满足，这会导致启动过程明显变慢，请读者耐心等待启动完毕），在命令行中输入“app”，运行 app 应用程序，即可命中断点。

此时，由于在 app 应用程序中第一次调用了 fork 函数，所以就进入了 int 0x80 的中断处理程序并命中了断点。接下来会调用 fork 系统调用的内核函数，继续按照下面的步骤调试：

（1）在“监视”窗口监视 last\_pid 的值和 current->pid 的值。全局变量 last\_pid（在文件 kernel/fork.c 的第 30 行定义）记录了最新的进程号。current->pid 的值是当前正在运行的进程的进程号。

（2）刷新“进程”窗口，可以查看当前进程的相关信息，其中背景色为绿色并且使用

current 游标指向的进程是当前进程，如图 2 所示。其中，“state=0”表示当前进程（即使用可执行文件 app 创建的进程）正处于运行态；“counter=11”表示其剩余时间片的大小；“priority=15”表示其优先级；“father=4”表示其父进程的进程号。读者也可以在“进程控制块”窗口工具栏上的下拉列表中选中进程 id 为6 的进程，查看进程控制块中的详细信息，如图 5-3 所示。在“进程控制块”窗口中，除了显示进程的基本信息外，还显示了进程的 tss、ldt 等信息。在这里，读者只需要学会使用这些工具窗口即可，更深入的研究留待后面的实验中进行。



**图2. “进程”窗口**



**图3. “进程控制块”窗口**

（3）当然，读者还可以使用“快速监视”功能查看进程控制块中的信息，但是显然没有使用前面提到的工具窗口那样方便和直观。不过，读者还是有必要练习一下从多个角度来查看内核中断后的信息，这样可以帮助读者更好的理解 Linux 操作系统的行为。选择菜单“调试”中的“快速监视”,打开“快速监视”对话框。在对话框的

“表达式”编辑控件中输入“\*current”就可以查看当前进程的控制块中的所有信息，可以看到，这些信息与“进程控制块”窗口中显示的当前进程的信息是一致的。

（4）关闭“快速监视”对话框后，按 F10 单步调试至第 119 行，再按 F11 进入 fork 系统调用的内核函数，可以看到其内核函数仍然是一个汇编函数。

（5）按 F10 单步调试至第 272 行。此时，第 271 行的 find\_empty\_process 函数（在文件 kernel/fork.c 的第 175 行定义）已经执行完毕，此函数为新进程取得了一个不重复的进程号，并在 task 数组中找到了一个未被使用的任务数组项，并返回其索引(在 EAX 寄存器中返回)。查看“监视”窗口，可看到 last\_pid 的值已经发生了变化，该值后面会作为新建的子进程的进程号。

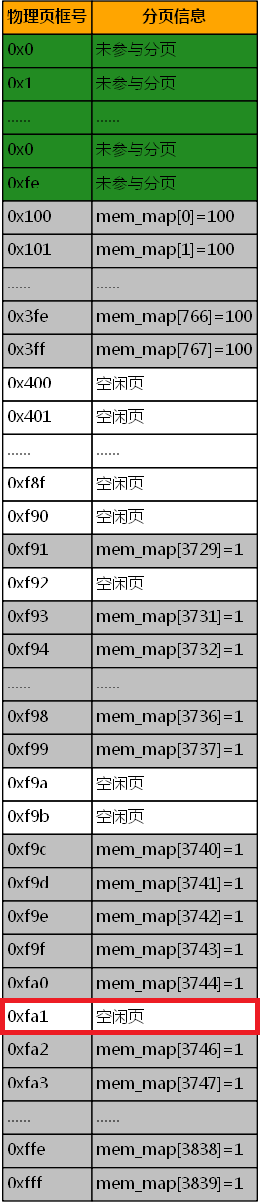
（6）按 F10 单步调试至第 279 行，然后按 F11 进入 copy\_process 函数。读者可以注意到，在第 278 行将 EAX 寄存器的值作为最后一个参数压入堆栈，根据 C 语言的函数调用约定，这也就意味着 copy\_process 函数的第一个参数为子进程在任务数组中的索引号。

copy\_process 函数（在文件 kernel/fork.c 的第 89 行定义）是 fork 过程中调用的一个重要函数，该函数主要为新创建的子进程从内存中申请一个进程控制块，并完成初始化工作。接下来，请读者按照下面的步骤继续调试 copy\_process 函数：

（1）首先，读者应该注意到，第 98 行代码会从内核存储空间中申请一个空闲的物理页

（大小为 4KB），并返回此物理页的起始物理地址。然后，在第 101 行将此物理页 的起始地址赋值给一个未被使用的任务数组项（由第一个参数作为数组下标），从 而将此物理页作为新创建进程的进程控制块（显然，一个进程控制块的大小不会超 过 4KB，所以在此物理页的后部会有一些空间被浪费掉，但是申请整页内存作为进程控制块会让程序比较简单，运行的速度也更快）。另外需要读者注意的是，这里 将物理地址直接作为逻辑地址使用了，这是由于 Linux 0.11 操作系统在管理内存时，将内核存储空间使用的所有物理页的物理地址都映射到了相同的逻辑地址，这 样就方便进行管理，在使用时也很方便。关于内存管理的内容读者会在后面的实验 中进行更加深入的研究，在这里只需要按照实验指导中的步骤观察到这种现象即可。

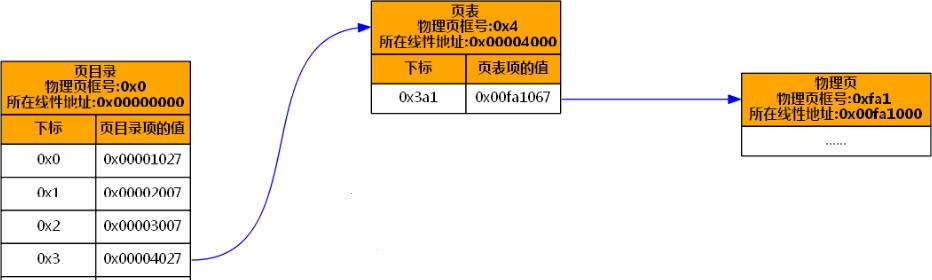
（2）读者可以选择“调试”菜单“窗口”中的“物理内存”，打开“物理内存”窗口，然后使用此窗口工具栏上的“刷新”按钮进行刷新，注意观察物理内存高地址端的第一个空闲物理页（如图 5-4 所示），此物理页在第 98 行代码执行时，会被 Linux操作系统进行分配，从而标记为已使用的页。记录下此空闲物理页的物理页框号， 物理页框号乘以 4K 就是此物理页的基址。



**图4.物理内存在高地址端的第一个空闲页**

（3）按 F10 单步执行第 98 行的代码，再次刷新“物理内存”窗口，可以发现之前记录的空闲物理页已经被使用了。将鼠标移动到第 98 行代码处的变量 p 上，可以看到此时 p 指针的值就是新分配的物理页的基址。

（4）此时读者有必要确认一下，新分配物理页的物理基址与其映射的逻辑地址是相同的， 从而允许在程序中直接进行访问，因为 X86 处理器进入保护模式后，在执行任何一 条指令时，指令中使用的地址必须是逻辑地址，然后由 X86 处理器的内存管理单元负责将逻辑地址映射为物理地址。首先，使用“调试”菜单“窗口”中的“二级页 表”打开“二级页表”窗口，然后，使用此窗口工具栏上的“刷新”按钮进行刷新， 可以显示出页目录和页表的映射关系，由于页表和物理页的映射关系数量比较大， 为了提高显示效率，读者可以指定哪些物理页需要显示。方法是使用此窗口工具栏 上的第二个按钮，弹出选择物理页的对话框，在下拉列表中选择“PDE 0x3 映射的页表”，然后在下面的列表中勾选“PTE 0x3a1 映射的物理页”，最后选择“绘制”按钮，可以在最右侧显示出映射的物理页和此物理页所在的“线性地址”（如图 5-5 所示）。注意，这里的线性地址需要加上段基址（分段变换）才能转换为逻辑地址， 但是由于 Linux 内核使用的段基址为 0，所以可以认为在 Linux 内核中使用的逻辑地址和线性地址是相同。这样，从图上就可以确认，刚刚分配的物理页的物理基址 与其映射的逻辑地址是相同的。关于地址映射的详细内容会在后面的实验中进行详 细讨论。



**图5. Linux 内核存储空间中物理地址与逻辑地址相同**

（5）按 F10 单步执行直到黄色箭头指向第 103 行。第 101 行将新创建的子进程控制块的指针放入了任务数组中，数组索引由第一个参数指定。此时刷新“进程”窗口，可以看到，下标为 5（nr 的值为 5）的进程就是新创建的子进程。在“进程控制块” 窗口工具栏的下拉列表中选择 task[5]进程，可以查看子进程控制块中各个成员的值，可以看到新创建的子进程控制块中各个成员的值都为 0，这是因为之前为进程控制块分配的物理页的内容都是 0 造成的（Linux 会将空闲物理页的内容清零）。读者也可以使用“快速监视”窗口查看表达式 \*p 的值。

（6）按 F10 单步执行第 103 行的代码，黄色箭头指向第 104 行。第 103 行的代码非常关键，此行代码将 current 指向的父进程控制块中的内容完全复制到了 p 指向的子进程控制块中，也就是子进程完全继承了父进程的各种资源。此时在“进程控制块” 窗口中，可以分别查看父进程和子进程各个成员的值，可以发现它们的值是完全相同的。这就可以解释很多现象，例如子进程和父进程的优先级相同，使用相同的tty 终端，打开了相同的文件等。

（7）由于子进程控制块除了从父进程控制块继承资源之外，还需要设置自己特有的资源， 所以，第 104 行设置子进程为“不可中断等待状态”；第 105 行设置子进程的进程号；第 106 行设置子进程的父进程号；第 125 行将子进程EAX 寄存器的值设置为 0，这也就是 fork 函数在子进程中返回 0 的原因。随后设置子进程控制块中的其他成员。

（8）第 146 行代码调用 copy\_mem 函数使用父进程的内存空间创建了一个副本，作为子进程的内存空间，这样子进程在开始运行时，就拥有了和父进程完全相同的指令、数据和栈，当然，在子进程运行的过程中，子进程对这些内存的修改就不会影响到父进程了，同样的，父进程从 fork 函数返回后对这些内存的修改也不会影响到子进程。此时，读者可以刷新“二级页表”窗口，记录下此时页目录中使用了多少页目录项。

（9）第 164 和第 165 行代码为子进程在全局描述符表中设置 TSS 和 LDT 描述符项，其作用会在后续的实验中进行讨论。此时，读者可以刷新“全局描述符表”窗口，可以看到在第 164 和第 165 行代码执行前，全局描述符表中还没有子进程（进程 ID 为7）的 TSS 和LDT 描述符项。

（10）在第 171 行添加一个断点，按 F5 继续调试，命中后删除该断点。再刷新“二级页表”窗口，可以看到 Linux 分配了两个新的页目录项，这两个新分配的页目录项就是用来映射子进程的内存空间的。再刷新“全局描述符表”窗口，可以看到已经为子进程设置了 TSS 和LDT 描述符项。此时，子进程已经进入了就绪态，可以开始运行了。读者可以刷新“进程运行轨迹”窗口，查看此时父进程和子进程的运行轨迹。

接下来，读者可以按照下面的步骤查看从 copy\_process 函数返回时的执行情况：

（1）按 F10 单步调试，直到从 copy\_process 函数返回到 kernel/system\_call.s 文件中的第 280 行。copy\_process 函数的返回值是子进程的进程号，会被放入 EAX 寄存器中，也就是父进程从 fork 函数返回时得到的返回值。

（2）按 F10 单步调试，直到从汇编函数返回到 kernel/system\_call.s 文件中的第 120行。

（3）继续按 F10 单步调试，直到第 133 行。可以看到在从 fork 系统调用返回之前，并没有执行进程调度 reschedule 函数，所以父进程会继续运行。但是，如果从 fork 返回后，在父进程调用了wait 或waitpid 系统调用函数等待子进程结束的情况下， 就会在从这些系统调用函数返回之前执行进程调度 reschedule 函数，从而让处于就绪状态的子进程先运行，待子进程退出后父进程再运行。

（4）按 F5 继续调试，可执行文件运行结束。

至此，读者已经在本练习的引导下调试跟踪了 fork 系统调用函数执行的完整过程。由于此练习涉及的内容较多，调试步骤也比较多，建议读者在时间允许的情况下，多调试几次， 加深理解。

2.2.5调用 execve 函数加载执行一个新程序

使用 fork 系统调用函数可以为父进程创建一个子进程，但是子进程和父进程执行的是同一个程序。为了让父进程加载执行一个新程序，可以使用 execve 系统调用函数。首先按照下面的步骤编写一个 Linux 0.11 应用程序，作为将要被 execve 函数加载的程序：

（1）新建一个 Linux011 应用程序项目。项目名称可以是 linuxapp。

（2）编辑 LinuxApp.c 文件中的 main 函数，代码如下：

int main( int argc, char \* argv[] )

{

printf("PID:%d hello world\n", getpid() );

return 0;

}

（3）按 F7 生成项目，F5 启动调试。

（4）将生成的可执行文件从软盘 B 拷贝到硬盘，命令如下：

mcopy b:linuxapp.exe newapp

（5）为 newapp 文件添加可执行权限，命令如下：

chmod +x newapp

（6）执行“sync”命令后运行 newapp，观察运行结果。

（7）结束本次调试。

接下来编写调用 execve 函数的应用程序：

（1）再次编辑 main 函数，代码如下：

int main( int argc, char \* argv[] )

{

execve("newapp", NULL, NULL);

printf("PID:%d hello execve\n", getpid() );

printf("PID:%d hello exit\n", getpid() );

return 0;

}

int execve( char\* file, char\*\* argv, char\*\* envp )函数用来加载执行一个新程序。参数 file 是需要被加载的程序文件名，参数 argv 是传递给新程序的命令行参数指针数组，参数 envp 是传递给新程序的环境变量指针数组。

（2）按 F7 生成项目，F5 启动调试；

（3）将生成的可执行文件从软盘 B 拷贝到硬盘，命令如下：

mcopy b:linuxapp.exe app

（4）给 app 添加可执行权限，命令如下：

chmod +x app

（5）查看可执行文件 app 的详细信息，命令如下：

ls -l app

将文件的大小记录下来，在后面调试跟踪 execve 函数的执行过程时会用到此值。

（6）执行“sync”命令后，运行可执行文件 app，观察运行结果。

（7）结束本次调试。

execve 系统调用会清理掉当前进程的页目录和页表项，并释放对应的物理页，然后为新加载的可执行文件中的指令和数据重新申请内存，并配置到当前进程的进程控制块中，还会重新设置代码执行的起始位置。此时当前进程的代码和数据将完全被新程序替换掉，并在该进程中开始执行新程序的代码。

2.2.6 调试跟踪 execve 函数的执行过程

为了调试跟踪 execve 函数的执行过程，同样需要在内核源代码中添加一个条件断点， 步骤如下：

（1）再启动一个 Linux Lab 程序，使用“Linux011 Kernel”模板新建一个 Linux011 Kernel 项目。

（2）用之前创建的 Linux011 应用程序项目的硬盘镜像文件 harddisk.img 覆盖当前Linux011 Kernel 项目的 harddisk.img 文件，这样就可以使用之前生成的的 app 和 newapp 文件了。

（3）打开 kernel/system\_call.s 文件，在第 102 行添加一个条件断点，条件为：

$eax==11 && current->executable->i\_size==文件大小

“$eax==11”中 11 是 execve 函数的系统调用号。“文件大小”是之前记录的应用程序可执行文件 app 的大小。

（4）按 F5 启动调试，运行 app 应用程序，命中断点。

此时，由于在应用程序中调用了 execve 函数，所以就进入了 int 0x80 的中断处理程序并命中了断点。接下来会调用 execve 系统调用的内核函数，继续按照下面的步骤调试：

（1）在“监视”窗口中查看 last\_pid 的值，在后续的调试过程中注意此值是否发生变化。同时，在“进程”窗口中，查看进程是否发生变化。

（2）按 F10 单步调试到第 119 行，按F11 进入到 execve 系统调用对应的汇编函数

sys\_execve，黄色箭头指向第 260 行。

（3）按 F10 单步调试到底 262 行，按 F11 进入到 do\_execve 函数中，该函数完成加载执行新程序的主要功能。

（4）在第 314 行添加一个断点，按 F5 继续调试，命中后删除该断点。第 303 到第 304 行初始化参数和环境变量空间的页面指针数组。第 306 行取得可执行文件对应的 i 节点号。第 309 到第 310 行计算参数个数和环境变量个数。

（5）在第 472 行添加一个断点，按 F5 继续调试，命中后删除该断点。该过程主要完成对文件合法性的检查以及参数和环境变量的复制。

（6）在第 494 行添加一个断点，刷新“二级页表”窗口后，按 F5 继续调试，命中后删除该断点。再次刷新“二级页表”窗口，可以看到当前进程的代码段和数据段内存被释放了，这是由第 485 和 486 行代码完成的。第 472 到第 474 行释放进程原始的可执行文件的 i 节点，并使其指向新程序的可执行文件的 i 节点， 接下来是对进程控制块信号句柄和协处理器的处理。

（7）按 F10 单步调试到第 508 行。第 494 到第 496 行创建参数和环境变量指针表，并返回该堆栈指针。第 499 行设置代码段、数据段以及堆栈段信息。第 503 到第 509 行设置进程堆栈开始字段所在页面以及用户 ID 和组 ID。在“进程控制块”窗口中， 可以查看进程的详细信息。

（8）在第 515 行添加一个断点，按 F5 继续调试，命中后删除该断点。第 508 到第 509 行初始化 bss 段数据。第 513 到第 514 行将堆栈上的代码指针替换为新程序的入口点地址，并将堆栈指针替换为新程序的堆栈指针。

（9）按 F10 单步调试，do\_execve 函数返回到 sys\_execve 函数。

（10）按 F5 继续调试，app 运行结束。

体会execve 函数的执行过程，该过程中并没有申请新的进程控制块，同时 last\_pid 的值也没有发生变化，只是对当前进程的控制块进行了相应的修改，从而加载执行了另一个程序。

2.2.7 fork 与 execve 的区别与联系

fork 会为子进程重新申请一个进程控制块（task\_struct），并拷贝父进程的进程控制块信息到子进程的进程控制块中，再对子进程的控制块做简单的修改，使子进程与父进程执行同样的程序。execve 并没有申请新的进程控制块，而是直接修改当前进程的进程控制块，并开始执行一个新程序。在为 Linux 开发应用程序时，往往会同时使用 fork 和execve 函数。一个程序在使用 fork 函数创建了一个子进程时，通常会在该子进程中调用 execve 函数加载执行另一个新程序,例如：

if( fork() )

{ /\* parent process \*/

}

else

{ /\* child process \*/

execve(…); }

# 四、实验设计

程序通常是指一个可执行文件，而进程则是一个正在执行的程序实例。利用分时技术，在 Linux 操作系统上可以同时运行多个进程。分时技术的基本原理是把 CPU 的运行时间分成一个个规定长度的时间片，让每个进程在一个时间片内运行。当一个进程的时间片用完时，操作系统就利用调度程序切换到另一个进程去运行。因此，本质上对于具有单个 CPU 的机器来说，某一时刻只有一个进程在运行，但是由于进程运行的时间片比较短（通常为几十毫秒），所以给用户的感觉是多个进程在同时运行。

对于 Linux 0.11 内核来讲，系统最多可有 64 个进程同时存在。除了第一个进程用“手工”建立以外，其余的都是现有进程使用系统调用 fork 函数创建的新进程。被创建的进程称为子进程，创建者则称为父进程。Linux 内核使用进程标识号来标识每个进程。进程由可执行的指令代码、数据区和堆栈区组成。进程中的指令代码和数据区分别对应于可执行文件中的代码段和数据段，而堆栈区是在创建进程时由操作系统为其分配的。每个进程只能执行自己的代码和访问自己的数据区和堆栈区。

Linux 操作系统内核通过进程表（数组）对进程进行管理，每个进程在进程表中占有一项。这个进程表数组在 kernel/sched.c 文件的第 112 行定义，数组长度是 64。 进程表项是一个 task\_struct 任务数据结构的指针。

任务数据结构task\_struct也称为进程控制块PCB 或进程描述符 PD。它定义在 include/linux/sched.h 文件中的第 146 行。其中保存着用于控制和管理进程的所有信息，主要包括进程当前运行的状态、信号、进程号、父进程号、运行时间累计值、正在使用的文件和本任务的局部描述符以及任务状态段等。

fork()调用创建新进程过程如下：

（1）在进程数组中找到一个还未被使用的空项，如果满了（64个进程在运行）则返回出错。

（2）在主内存区申请一页来存放task\_struct信息。复制当前进程的task\_struct到新进程，并把新进程的state更改为不可中断的睡眠状态（防止调度进程在未新进程未初始化完成时调度它）。

（3）更改新进程task\_struct其他数据项。把当前进程设置为新进程的父进程，清除信号位图并复位进程统计数据，设置初始运行时间片位15。设置tss中各寄存器值，tss.eax=0(fork()后新进程的返回值),tss.esp0为新进程tss\_struct所在页面的顶端,tss.ss0设置为内核数据段选择符。还有tss.ldt，以及如果使用协处理器，则许把协处理器状态保存到tss.i387。

（4）设置新进程数据和代码段描述符，并复制当前进程的页表。注意，系统此时不为新进程分配实际的物理内存页面，而是共享父进程的。只有当子进程中任一个写内存时，系统才为执行写操作的进程实际分配，这是写时复制技术（copy on write）。

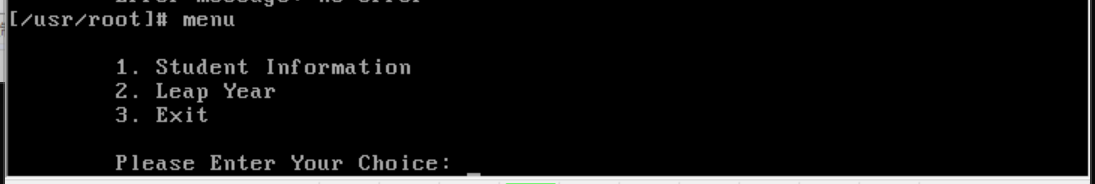
（5）父进程打开的文件的打开次数+1。

（6）设置GDT中新任务的TSS和LDT描述符。

（7）设置新任务状态位就绪态，并返回新进程号。

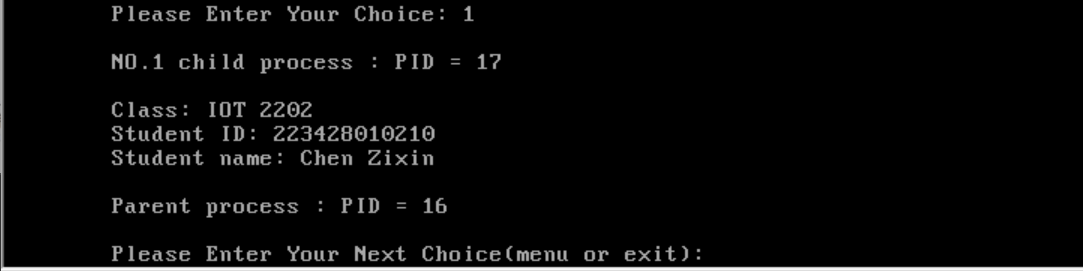
# 五、实验步骤及实验结果

1. 在Linux0.11环境下编写C程序。主程序menu首先打印一个菜单（如图6所示）：

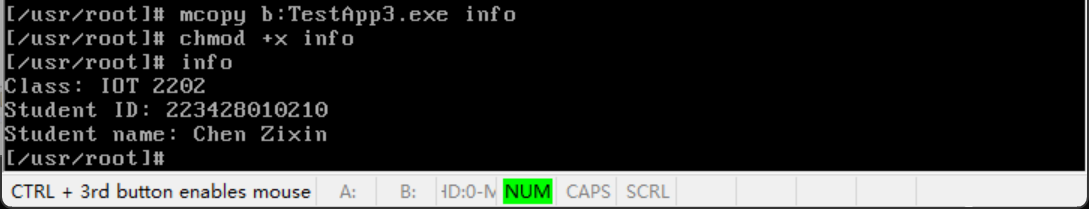


**图6. 菜单示例**

（1）用户输入1。主程序通过fork函数创建子进程1，然后打印自己的PID号。子程序1首先在屏幕上输出自己的PID号，接着调用execve函数加载执行程序info.c的可执行文件info。如图7所示。程序info功能是输出三行信息：本人班号，学号和姓名（全拼）。如图8所示。

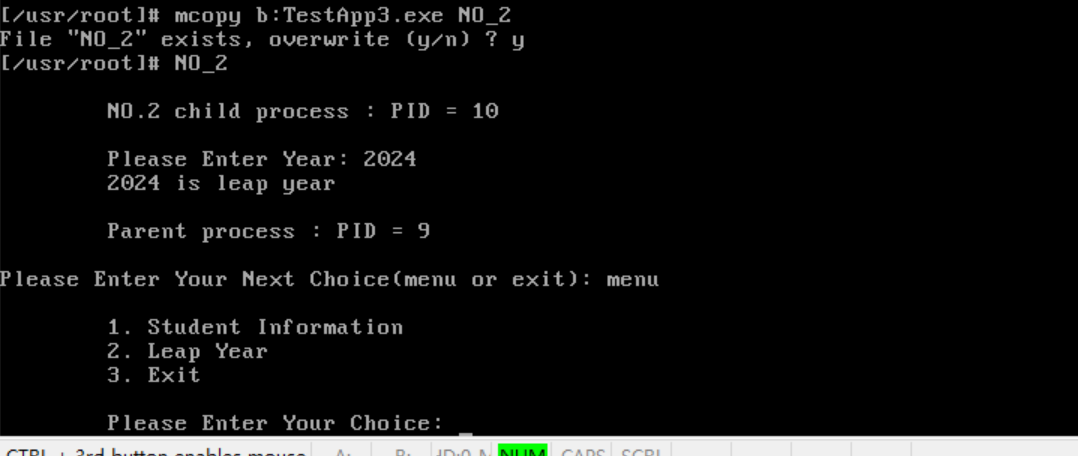


**图7. NO1打印信息示例**

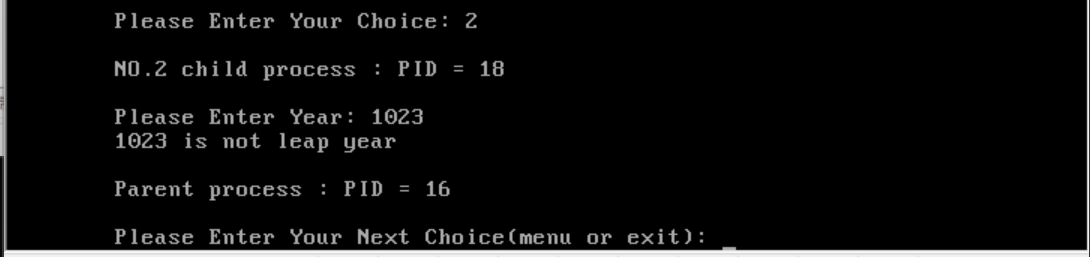
****

**图8. info打印信息示例**

（2）用户输入2。主程序通过fork函数创建子进程2，然后打印自己的PID号。子程序2首先在屏幕上输出自己的PID号，然后调用execve函数加载执行程序year.c的可执行文件year。程序year用于判断闰年：用户输入一个年号，若是闰年，系统显示：xxxx is leap year（如图9所示）,否则显示：xxxx is not leap year（如图10所示）：

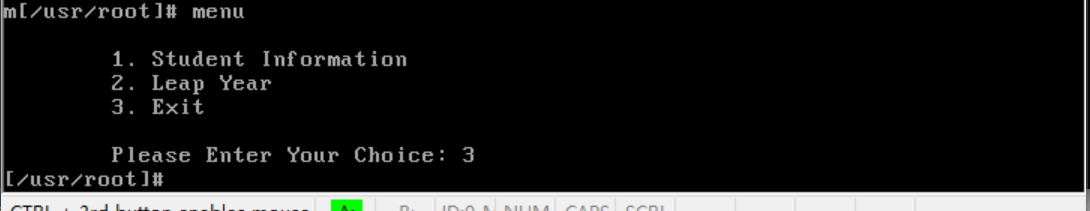


**图9. 输入闰年示例**



**图10.输入非闰年示例**

（3）用户输入3。 主程序直接终止执行。如图10所示。



**图11. 退出示例**

要求：主程序在创建子进程后要等待子程序执行结束才打印PID号，最后结束运行。

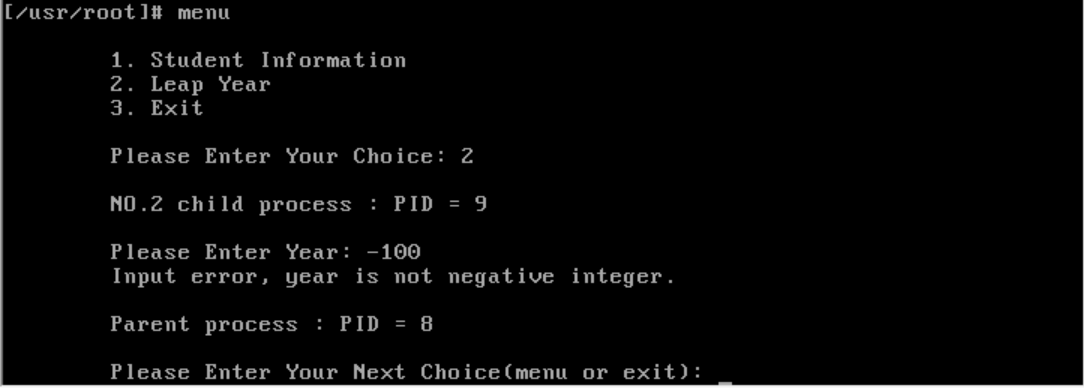
提示：父进程创建子进程后，可以调用wait函数实现等待子程序执行结束才打印PID号。

wait函数： pid\_t wait (int \* status)

函数说明：进程一旦调用了 wait，就立即阻塞自己，由wait自动分析是否当前进程的某个子进程已经退出，子进程的结束状态值会由参数status 返回，wait 就会收集这个子进程的信息， 并把它彻底销毁后返回父进程。

函数返回值：如果函数执行成功则返回子进程识别码（PID），如果有错误发生则返回-1。失败原因存于errno 中。如果不需要子进程结束状态值，则wait函数的参数status 可以设成NULL。

1. 在Linux0.11环境下运行程序，显示结果。闰年判断程序leap对输入数据进行检查：如果输入数据是负数，不是一个合法年份，系统提示“Input error , year is not negative integer. ”



**图11. 输入非法年份示例**

# 六、实验出现的问题及解决方法

问题：回到父进程时，先让我输入字符串而不是先打印提示信息(scanf函数和printf函数实现顺序与预期不符)。

解决方法：

1. 手动刷新缓冲区

使用 fflush(stdout); 手动刷新缓冲区，强制将缓冲区内容打印到屏幕上：

printf("Please Enter Your Next Choice(menu or exit): ");

fflush(stdout); // 强制输出缓冲区内容

scanf("%s", str);

1. 添加换行符

将提示信息 printf 语句末尾加上换行符 \n，因为换行符会自动刷新缓冲区：

printf("Please Enter Your Next Choice(menu or exit):\n");

scanf("%s", str);

1. 运行程序时将 stdout 设置为无缓冲

在程序开始时将标准输出设置为无缓冲，这样所有输出都会立即显示：

setvbuf(stdout, NULL, \_IONBF, 0); // 设置 stdout 为无缓冲

# 七、结论

通过本次实验，我深入理解了Linux操作系统中“进程管理”的工作机制，尤其是`fork`和`execve`系统调用的原理和执行过程。`fork`的“神奇之处”在于调用一次却能返回两次，它通过创建一个几乎与父进程完全相同的子进程，极大体现了Linux系统的高效性和灵活性。`execve`系统调用则进一步展示了加载和执行新程序的核心操作，是系统资源动态分配的重要体现。

在实验中，通过调试和跟踪`fork`函数，观察到父进程和子进程在进程表中的创建与运行情况，并通过调用`wait`和`waitpid`函数实现了父子进程间的同步。结合实验工具直观查看了进程的运行轨迹，强化了对Linux进程调度机制的理解。同时，通过设置条件断点调试`fork`系统调用的内核执行过程，我进一步掌握了内核中进程创建的底层实现，包括`last\_pid`变量的变化及新进程的控制块复制等核心细节。

本次实验让我深刻认识到Linux操作系统设计的精妙之处，特别是其对进程控制的高效实现方式。通过实验，我不仅巩固了理论知识，还提升了调试技能和分析能力，为后续深入学习Linux内核源码打下了坚实基础。这次实验让我体会到探索底层机制的乐趣，也意识到扎实掌握基础知识的重要性。

# 附录：程序源代码

*程序代码部分注释清晰，示例如下：*

#define \_\_LIBRARY\_\_

#include "LinuxApp.h"

#include <stdio.h>

#include <string.h>

#include <errno.h>

/\* 声明外部变量 errno，用于存储错误码 \*/

extern int errno;

/\* 主程序入口 \*/

int main(int argc, char\* argv[], char str[10]) {

int i; /\* 菜单选择变量 \*/

int year; /\* 年份变量，用于闰年判断 \*/

int pid; /\* 进程号变量 \*/

/\* 打印个人信息 \*/

printf("\n\tClass: IOT 2202\n\tStudent ID: 223428010210\n\tStudent name: Chen Zixin\n");

return 0;

/\* 分隔线 - 功能模块 NO\_1 开始 \*/

/\* 使用 fork 创建子进程 \*/

pid = fork();

if (pid) { /\* 如果 pid 不为 0，表示当前是父进程 \*/

wait(NULL); /\* 父进程等待子进程执行完成 \*/

printf("\n\tParent process : PID = %d\n", getpid()); /\* 输出当前父进程的 PID \*/

} else {

/\* 如果 pid 为 0，表示当前是子进程 \*/

printf("\n\tNO.1 child process : PID = %d\n", getpid()); /\* 输出当前子进程的 PID \*/

execve("info", NULL, NULL); /\* 在子进程中执行 info 程序 \*/

}

/\* 父进程继续执行，提示用户输入 \*/

printf("\n\tPlease Enter Your Next Choice(menu or exit): ");

scanf("%s", str); /\* 读取用户输入 \*/

/\* 根据用户输入的字符串决定下一步操作 \*/

if (strcmp(str, "menu") == 0) {

execve("menu", NULL, NULL); /\* 执行 menu 程序 \*/

} else if (strcmp(str, "exit") == 0) {

exit(0); /\* 用户选择退出程序 \*/

} else {

/\* 用户输入非法内容，打印错误提示 \*/

printf("\n\tError message: Illegal Entry\n");

return 1;

}

return 0;

/\* 分隔线 - 功能模块 NO\_2 开始 \*/

/\* 再次创建子进程 \*/

pid = fork();

if (pid) { /\* 如果 pid 不为 0，表示当前是父进程 \*/

wait(NULL); /\* 父进程等待子进程执行完成 \*/

printf("\tParent process : PID = %d\n\n", getpid()); /\* 输出当前父进程的 PID \*/

} else {

/\* 如果 pid 为 0，表示当前是子进程 \*/

printf("\n\tNO.2 child process : PID = %d\n", getpid()); /\* 输出当前子进程的 PID \*/

execve("year", NULL, NULL); /\* 在子进程中执行 year 程序 \*/

}

/\* 父进程继续执行，提示用户输入 \*/

printf("\tPlease Enter Your Next Choice(menu or exit): ");

fflush(stdout); /\* 手动刷新缓冲区，确保提示信息及时输出 \*/

scanf("%s", str); /\* 读取用户输入 \*/

/\* 根据用户输入的字符串决定下一步操作 \*/

if (strcmp(str, "menu") == 0) {

execve("menu", NULL, NULL); /\* 执行 menu 程序 \*/

} else if (strcmp(str, "exit") == 0) {

exit(0); /\* 用户选择退出程序 \*/

} else {

/\* 用户输入非法内容，打印错误提示 \*/

printf("\n\tError message: Illegal Entry\n");

}

return 0;

/\* 分隔线 - 闰年判断功能开始 \*/

/\* 提示用户输入年份 \*/

printf("\n\tPlease Enter Year: ");

scanf("%d", &year); /\* 读取用户输入的年份 \*/

/\* 判断年份是否合法并输出结果 \*/

if (year < 0) {

printf("\tInput error, year is not negative integer.\n\n"); /\* 输入的年份为负数，输出错误提示 \*/

} else if (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0)) {

/\* 满足闰年条件，输出闰年信息 \*/

printf("\t%d is a leap year\n\n", year);

} else {

/\* 不满足闰年条件，输出非闰年信息 \*/

printf("\t%d is not a leap year\n\n", year);

}

return 0;

/\* 分隔线 - 菜单功能开始 \*/

/\* 提示用户选择功能 \*/

printf("\n\t1. Student Information\n\t2. Leap Year\n\t3. Exit\n\n\tPlease Enter Your Choice: ");

scanf("%d", &i); /\* 读取用户的菜单选择 \*/

/\* 根据用户选择的菜单项执行不同功能 \*/

switch (i) {

case 1:

execve("NO\_1", NULL, NULL); /\* 执行功能模块 NO\_1 \*/

break;

case 2:

execve("NO\_2", NULL, NULL); /\* 执行功能模块 NO\_2 \*/

break;

case 3:

return 0; /\* 用户选择退出程序 \*/

}

return 0;

/\* 分隔线 - 提示信息和缓冲区问题说明 \*/

/\*

父进程在等待子进程完成后继续执行。

printf 和 scanf 的行为与标准输入/输出的缓冲机制有关，导致输入与提示信息的顺序不符合预期。

这种现象主要是因为标准输出缓冲机制的影响。

解决方法：

1. 使用 fflush(stdout); 手动刷新缓冲区，强制将缓冲区内容打印到屏幕上。

2. 添加换行符 \n，因为换行符会自动刷新缓冲区。

3. 设置 stdout 为无缓冲模式，确保所有输出立即生效。

\*/

}