**中国矿业大学计算机学院**

**2020 级本科生课程报告**

课程名称 操作系统课程设计

报告时间 2022年6月4日

学生姓名 杨再润

学 号 06203203

专 业 数据科学与大数据

任课教师 鲍 宇

目录

[实验五 进程的创建 3](#_Toc106649621)

[【实验目的】 3](#_Toc106649622)

[【实验步骤】 3](#_Toc106649623)

[【实验内容】 3](#_Toc106649624)

[【运行结果】 10](#_Toc106649625)

[【思考与练习】 16](#_Toc106649626)

[【感受与体会】 16](#_Toc106649627)

# 实验五 进程的创建

## 【实验目的】

* 掌握创建子进程和加载执行新程序的方法，理解创建子进程和加载执行程序的不同。
* 调试跟踪fork和execve系统调用函数的执行过程。

## 【实验步骤】

* **调用fork函数创建子进程**
* **查看父进程与子进程的运行轨迹**
* **调用execve函数加载执行一个新程序**

## 【实验内容】

1. **任务（一）：在Linux 0.11应用程序中调用fork函数创建子进程**

**准备实验**

使用浏览器登录平台领取本次实验对应的任务，从而在平台上创建个人项目（Linux 0.11 内核项目），然后使用VSCode将个人项目克隆到本地磁盘中并打开。

**调用fork函数创建子进程**

在Linux 0.11应用程序中调用fork函数创建子进程，并分析程序运行的结果。步骤如下：

1. 按F5启动调试。
2. 待Linux 0.11启动后，使用vi编辑器新建一个main.c文件，编写如下的代码。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。
3. 使用命令gcc main.c -o app生成可执行文件app。
4. 执行 chmod +x app命令为app文件添加可执行权限。
5. 执行sync命令，将文件保存到硬盘。
6. 使用命令./app运行可执行文件app，分析运行结果。

系统调用函数fork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，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）。**所以，通常会在分支语句中使用fork函数的返回值作为判断条件，从而使父进程和子进程开始运行不同的代码，而且习惯在子进程分支的最后使用return语句退出子进程（不是强制的），而不会让子进程继续运行分支语句后面的代码。

在父进程中可以调用wait函数阻塞父进程，直到子进程退出后才会从这个函数中返回，从而让父进程继续运行。该函数的原型在include/sys/wait.h文件中定义如下：

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

按照下面的步骤继续使用vi编辑器修改main.c文件：

1. 在printf(“PID:%d parent process continue\n”, getpid());语句前面添加一行语句：  
   wait(NULL);
2. 重新编译、运行应用程序app，观察运行结果与之前有何不同。

**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，读者可以使用可视化窗口，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. 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。
2. 按F5启动调试。在Linux操作系统启动完毕之前会多次命中刚刚添加的断点，每次命中断点后都按F5继续运行即可，直到Linux启动完毕。
3. 在Linux的终端输入命令./app后，父进程和子进程在结束时都会命中此断点，所以在第一次命中断点时，可以按F5继续运行，在第二次命中断点时，在VSCode的“View”菜单中选择“Command Palette...”，会在VSCode的顶部中间位置显示命令面板，输入“Lab: New Visualizer View”命令后，VSCode会在其右侧弹出一个窗口让读者查看可视化视图。在右侧可视化视图顶部的编辑框中输入命令“#sched”后按回车（需要等待较长时间完成刷新），就可以查看进程的运行轨迹。注意观察父进程和子进程创建和结束的顺序，如果读者在父进程中调用了wait函数等待子进程结束的话，还可以看到父进程首先创建子进程，然后父进程进入阻塞状态等待子进程结束，在子进程结束后父进程才会被唤醒的过程。

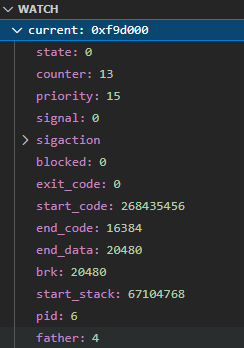
**调试跟踪fork函数的执行过程**

为了调试跟踪在Linux 0.11应用程序中调用fork函数时的执行过程（在Linux 0.11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. 在VSCode中删除所有断点，然后按F5启动调试。在Linux 0.11的终端输入下面的命令，查看可执行文件app的信息，将app文件的大小记录下来，在后面添加条件断点时会用到此值。  
   ls -l app
2. 结束调试，关闭Bochs虚拟机。
3. 使用VSCode打开kernel/system\_call.s文件，在第102行添加一个断点。
4. 在刚刚添加的断点上点击鼠标右键，在弹出的菜单中选择“Edit Breakpoint”，会在编辑器中显示出用于输入条件表达式的编辑框。在编辑框中设置断点条件为下面的表达式后按回车确认：  
   $eax==2 && current!=0 && current->executable->i\_size==*文件大小*  
   “文件大小”就是之前记录的应用程序可执行文件app的大小，需要读者根据实际的文件大小进行替换。这样就只有在执行app中的fork函数时，才会命中此条件断点，也就是说只有当条件断点中的布尔表达式的值为true时，此断点才会被命中。“$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126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，其中保存了可执行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。注意，如果启动调试后，条件断点变成一个空心的圆环，说明读者为条件断点编写的布尔表达式有语法错误，只需将表达式语法修改正确即可。
5. 按F5启动调试（注意，由于添加了一个条件断点，调试器需要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）。
6. 在Linux 0.11的终端输入命令app，运行app应用程序，即可命中刚刚添加的条件断点。

此时，由于在app应用程序中调用了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：

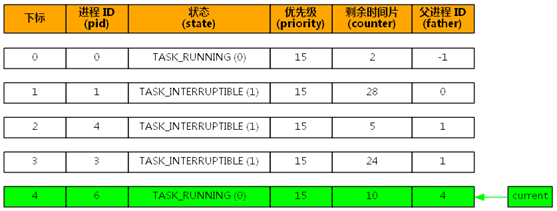
1. 在“WATCH”窗口添加last\_pid和current->pid，查看它们的值。全局变量last\_pid（在文件kernel/fork.c的第30行定义）记录了最新的进程号。current->pid的值是当前正在运行的进程的进程号，也就是app应用程序的进程号。
2. 在“WATCH”窗口添加全局变量current并展开它的值，可以查看当前进程的信息。其中，“state=0”表示当前进程（即使用可执行文件app创建的进程）正处于运行态；“counter=13”表示其剩余时间片的大小；“priority=15”表示其优先级；“father=4”表示其父进程的进程号。如图5-1所示。

****

**图5-1：查看全局变量current**

|  |
| --- |
| **注意**，current是一个结构体的指针，所以在其右侧显示了它所指向的地址。从C语言的角度理解，指针的值就是一个地址，只有在指针变量的前面添加了\*后，才能访问地址所指向的内存，也就是结构体的各个域。VSCode为了让用户有更好的体验，在这里进行了优化，当在“WATCH”窗口中添加一个指针变量时，VSCode会尝试根据其指针类型访问其指向的内存，从而允许读者展开此变量。读者也可以尝试在“WATCH”窗口中添加\*current，可以得到同样的效果。 |

1. 在“WATCH”窗口添加全局变量task并展开它的值，可以查看进程表中的所有进程的信息，如图5-2所示。其中，下标为4的那一项存储的地址，与图5-1中current指针所指向的地址是一致的。也就是说，app应用程序进程和其它进程一样，都在task中进行了管理，但是由于该进程现在正在运行，所以让current指向了它的进程控制块。注意，task中的元素是进程控制块的指针，所以在每一个元素的右侧显示了它所指向的地址，同样适用于上面的注意事项。
2. 在VSCode的“View”菜单中选择“Command Palette...”，会在VSCode的顶部中间位置显示命令面板，输入“Lab: New Visualizer View”命令后，VSCode会在其右侧弹出一个窗口让读者查看可视化视图。在右侧可视化视图顶部的编辑框中输入命令“#task”后按回车，就可以查看进程列表了，如图5-3所示。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，其state字段的值为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的值为11表示其剩余时间片的大小；priority的值为15表示其优先级；father的值为4表示其父进程的进程号。读者也可以从图中直观的掌握其他进程的重要信息。



**图 5-3：进程列表**

1. 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。
2. 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“WATCH”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。
3. 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nr为子进程在任务数组task中的下标。

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，读者需要注意到，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task中的一项（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同样的逻辑地址，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. 按F10单步执行第98行的代码，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。
3. 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。此时在“WATCH”窗口中，可以看到task中下标为5（nr的值为5）的进程就是新建的子进程，展开后可以查看子进程控制块中各个成员的值，可以看到新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 0.11会将空闲物理页的内容清零）。
4. 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“WATCH”窗口中，可以分别查看父进程task[4]和子进程task[5]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。
5. 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。
6. 第146行代码调用copy\_mem函数为父进程的内存空间创建了一个副本，该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。
7. 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。
8. 在第171行点击鼠标右键，在弹出的菜单中选择“Run to Cursor”，会运行到第171行后中断。此时，子进程已经进入了就绪态，可以开始运行了。

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。
4. 按F5继续调试，在Bochs的Display窗口中可以看到app可执行文件运行结束。
5. 结束调试，关闭Bochs虚拟机。

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，一边调试一边阅读代码和注释，加深理解。

**提交作业**

读者首先需要将Linux 0.11硬盘中的main.c文件通过软盘B复制到Linux 0.11内核项目的根目录中。然后使用VSCode左侧的“源代码版本控制窗口”查看文件变更详情，确认无误后再将本地项目提交到平台的个人项目中，方便教师通过平台查看读者提交的作业。

1. **任务（二）：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。如果需要让父进程加载执行一个新程序，可以使用 execve系统调用函数。

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。

**准备实验**

使用浏览器登录平台领取本次实验对应的任务，从而在平台上创建个人项目（Linux 0.11 内核项目），然后使用VSCode将个人项目克隆到本地磁盘中并打开。

**调用execve函数加载执行一个新程序**

首先编写一个供execve函数加载的应用程序：

1. 按F5启动调试。
2. 待Linux 0.11启动后，使用vi编辑器新建一个new.c文件
3. 使用命令gcc new.c -o new生成可执行文件new。
4. 执行 chmod +x new命令为new文件添加可执行权限。
5. 执行sync命令，将文件保存到硬盘。
6. 使用命令./ new运行可执行文件new，确保其可以正常运行。

接下来编写调用execve函数的应用程序：

1. 使用vi编辑器新建一个old.c文件
2. 使用命令gcc old.c -o old生成可执行文件old。
3. 执行 chmod +x old命令为old文件添加可执行权限。
4. 执行sync命令，将文件保存到硬盘。
5. 使用命令./old运行可执行文件old，注意观察输出的PID的值，以及输出的内容与读者期望的是否一致，并尝试说明原因。

系统调用execve会清理掉当前进程的内存空间，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，还会将新加载程序的入口点设置为执行的起始位置。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。所以在old程序中，调用execve加载new程序后面的代码就没有机会执行了，而是在当前进程中开始执行new程序了，并且PID保持不变。

**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. 在Linux 0.11的终端输入下面的命令，查看可执行文件old的信息，将old文件的大小记录下来，在后面添加条件断点时会用到此值。  
   ls -l old
2. 结束调试，关闭Bochs虚拟机。
3. 使用VSCode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!=0 && current->executable->i\_size==*文件大小*

“$eax==11”中的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old的大小。

1. 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）。
2. 在Linux 0.11的终端输入命令./ old，运行old应用程序，即可命中刚刚添加的条件断点。

此时，由于在应用程序old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. 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。
2. 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。
3. 在第314行点击鼠标右键，在弹出的菜单中选择“Run to Cursor”，会运行到第314行后中断。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。
4. 在第472行点击鼠标右键，在弹出的菜单中选择“Run to Cursor”，会运行到第472行后中断。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。
5. 在第494行点击鼠标右键，在弹出的菜单中选择“Run to Cursor”，会运行到第494行后中断。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。
6. 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行设置进程栈开始字段所在页面以及用户ID和组ID。
7. 在第515行点击鼠标右键，在弹出的菜单中选择“Run to Cursor”，会运行到第515行后中断。第508到第509行初始化bss段数据。第513到第514行将栈上的代码指针替换为新程序的入口点地址，并将栈指针替换为新程序的栈指针。
8. 按F10单步调试，do\_execve函数返回到sys\_execve函数。
9. 按F5继续调试，在Bochs的Display窗口中可以看到old可执行文件运行结束。
10. 结束调试，关闭Bochs虚拟机。

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。

**提交作业**

读者首先需要将Linux 0.11硬盘中的new.c文件和old.c文件通过软盘B复制到Linux 0.11内核项目的根目录中。然后使用VSCode左侧的“源代码版本控制窗口”查看文件变更详情，确认无误后再将本地项目提交到平台的个人项目中，方便教师通过平台查看读者提交的作业。

1. **fork 与 execve 的区别与联系**

系统调用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。系系统调用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()!=0 )

{

/\* parent process \*/

}

else

{

/\* child process \*/

execve(…);

}

## 【运行结果】

**调用fork函数创建子进程**

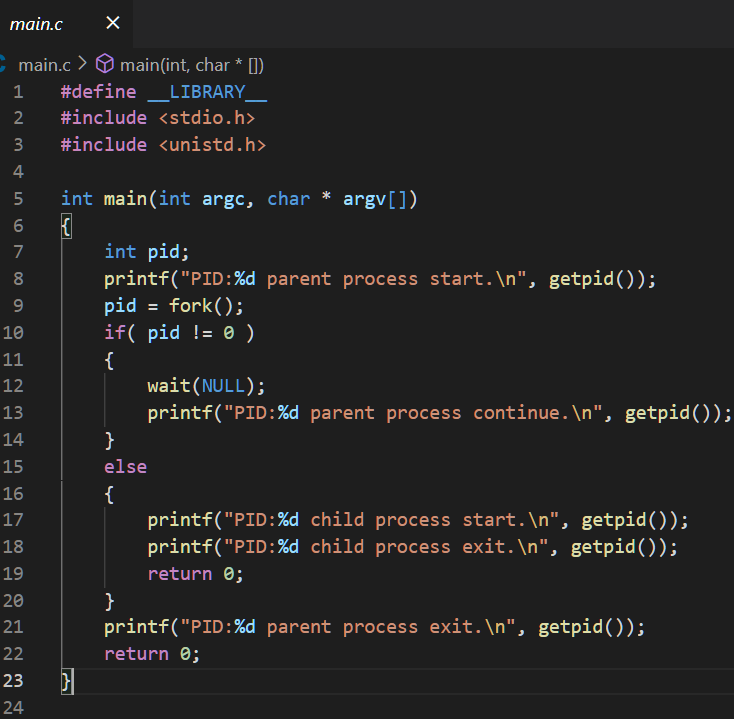


图 mian.c 文件内容

使用vi编辑器新建一个main.c文件，编写如图的代码。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。

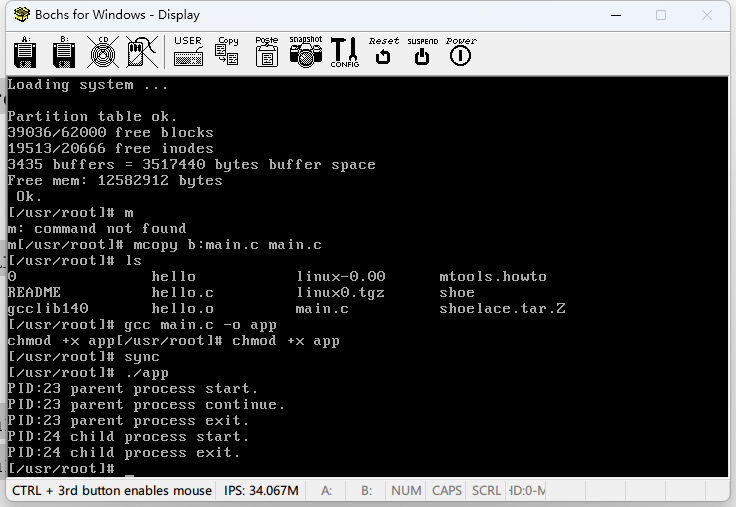


图 main.c运行结果

系统调用函数fork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，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）。

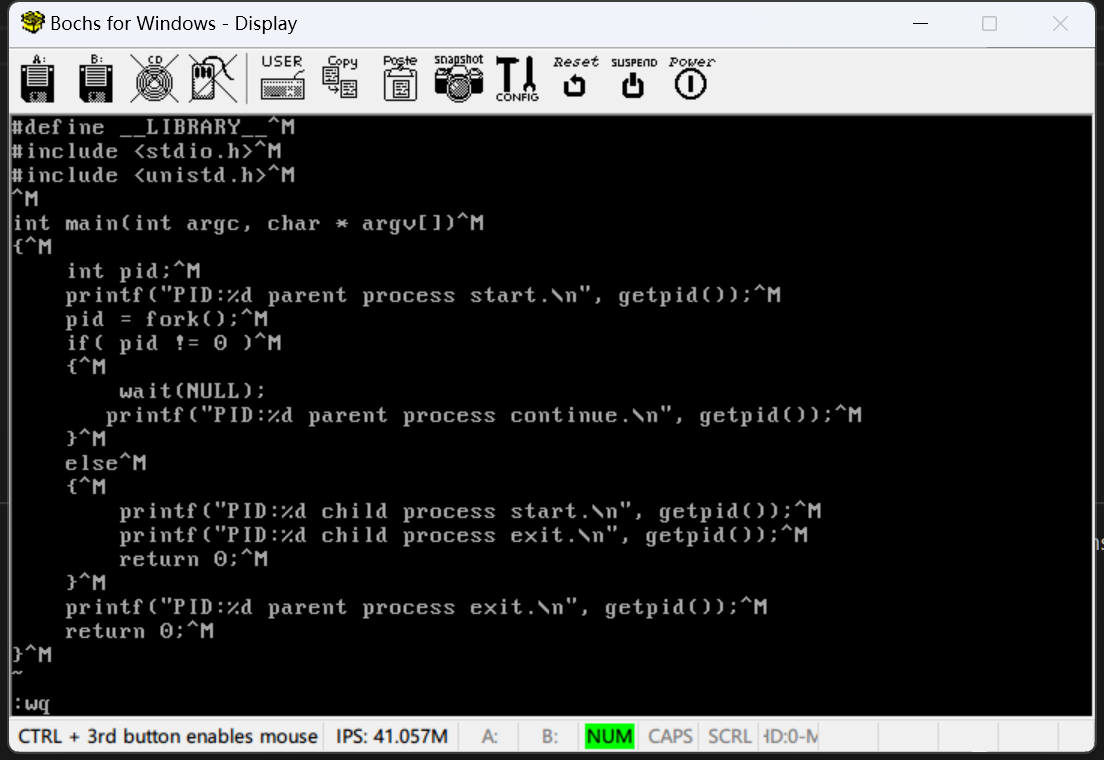


图 按要求在printf前一行加入wait(NULL)

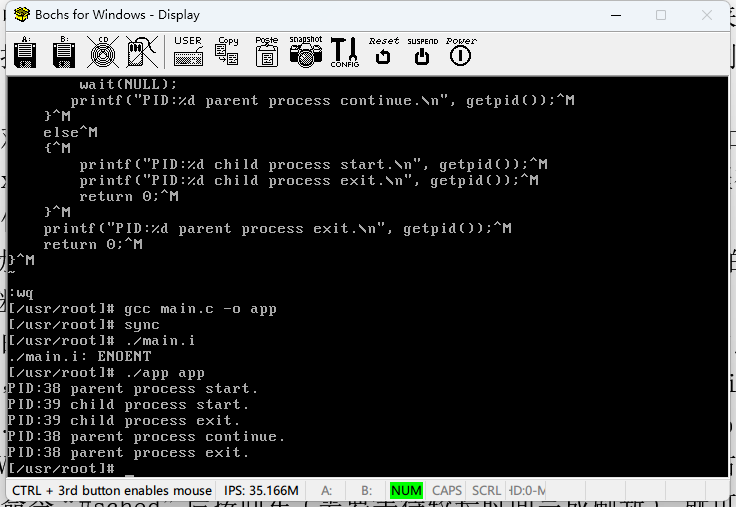


图 添加了wait(NULL)后再次运行main.c

此时的不同是，子进程的创建会导致父进程的中断。

**查看父进程与子进程的运行轨迹**

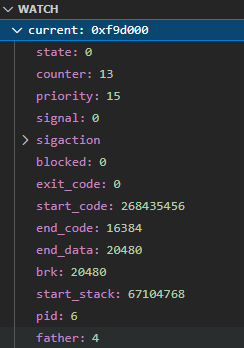
****

图 查看全局变量current

在“WATCH”窗口添加全局变量current并展开它的值，可以查看当前进程的信息。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=13”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。

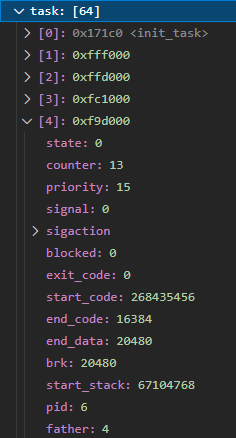


图 查看全局变量task

在“WATCH”窗口添加全局变量task并展开它的值，可以查看进程表中的所有进程的信息，如图所示。其中，下标为4的那一项存储的地址，与上图中current指针所指向的地址是一致的。也就是说，app应用程序进程和其它进程一样，都在task中进行了管理，但是由于该进程现在正在运行，所以让current指向了它的进程控制块。

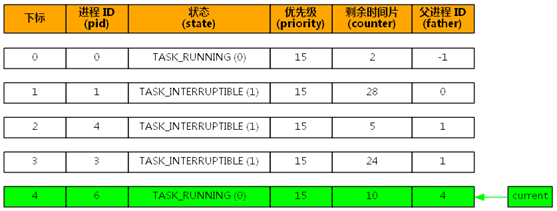


图 进程列表

counter的值为11表示其剩余时间片的大小；priority的值为15表示其优先级；father的值为4表示其父进程的进程号。

**调用execve函数加载执行一个新程序**

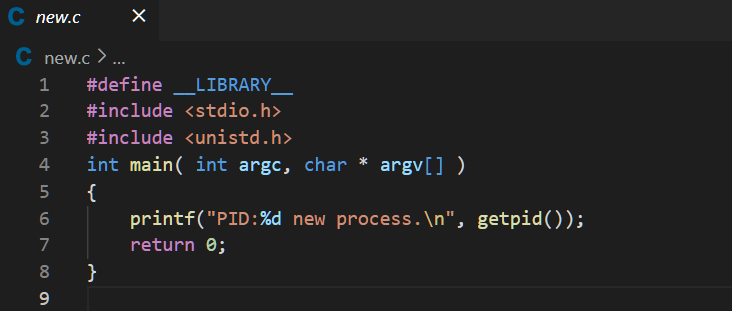


图 new.c文件

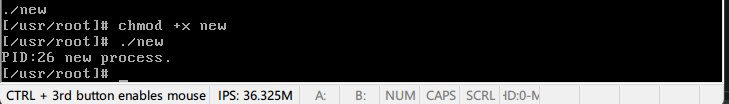


图 new.c运行结果

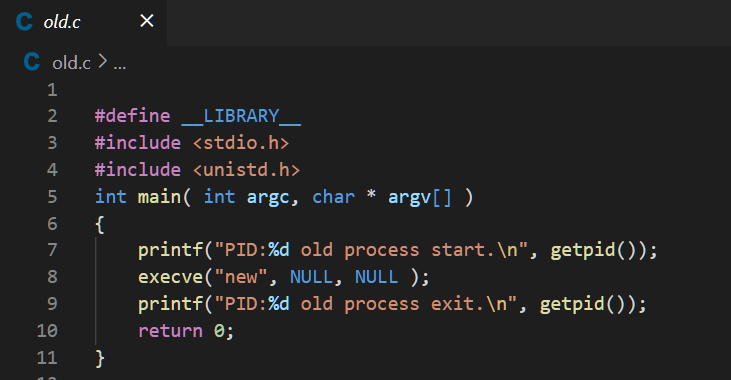


图 old.c文件

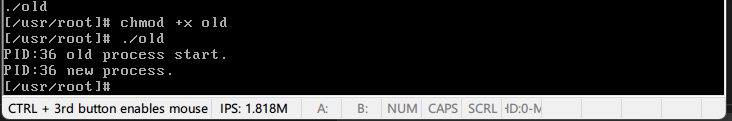
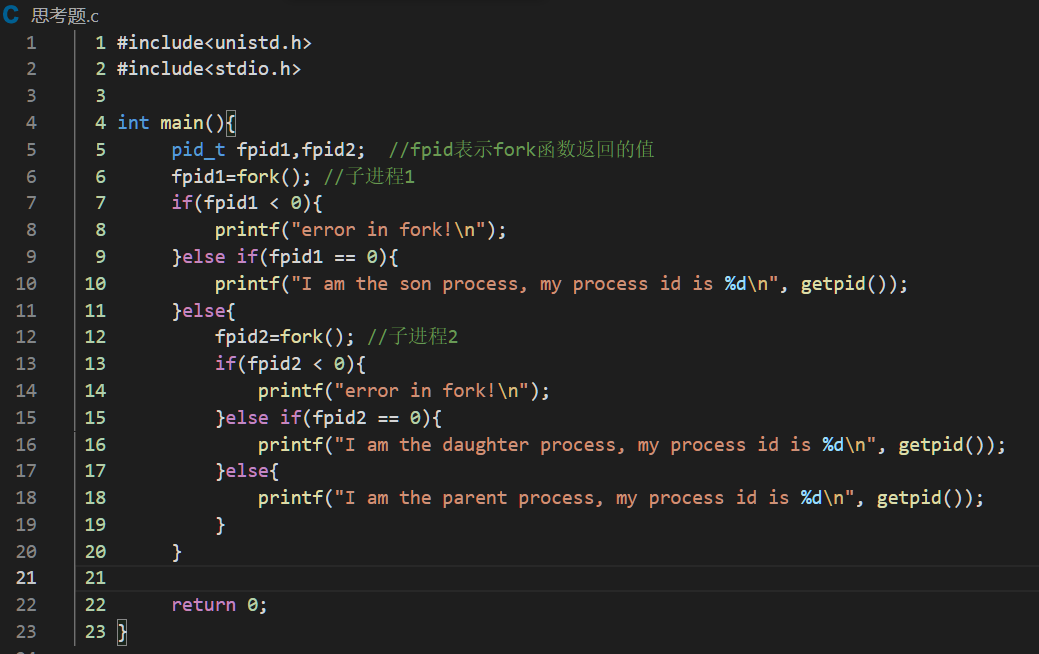


图 old.c运行结果

系统调用execve会清理掉当前进程的内存空间，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，还会将新加载程序的入口点设置为执行的起始位置。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。所以在old程序中，调用execve加载new程序后面的代码就没有机会执行了，而是在当前进程中开始执行new程序了，并且PID保持不变。

## 【思考与练习】

1. 模仿3.1中Linux 0.11应用程序的源代码，使用for语句编写一个循环，使父进程能够循环创建10个子进程，每个子进程在输出自己的pid后退出，父进程等待所有子进程结束后再退出。
2. 结合3.3中的内容编写一个Linux应用程序，在main函数中使用fork函数创建一个子进程，在子进程中使用execve函数加载执行另外一个程序的可执行文件，并且让父进程在子进程退出后再结束运行。



**图 思考题代码**

## 【感受与体会】

**1.** fork与execve对比：**系统调用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。系系统调用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。**

**2.** **作为一个Linux系统新手，刚开始看fork()函数时也是很懵逼的，也是看了好多博客才逐渐明白的，上面的解释可能写得不是很清楚，自己Linux系统的学习之路还很遥远，但是慢慢来慢慢练，一天了解几个命令总会用熟的。**