卡通画

描述已自动生成

**操作系统实验报告**

**实验题目**  进程的创建

**学生姓名**  杨程锦

**学 号**  2021214710

**专业班级**  计科21-1班

**指导教师**  田卫东

**完成日期**  2023.12.21

**合肥工业大学 计算机与信息学院**

1. **实验目的和任务要求**

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。

1. **实验原理**

实验五关注于创建子进程和加载执行新程序的方法，通过调试跟踪fork和execve系统调用函数的执行过程，理解进程创建和执行程序的区别。fork用于创建子进程，execve用于在已有进程中加载并执行一个新程序。

1. **实验内容**
   1. **在Linux 0.11应用程序中调用fork函数创建子进程**

**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。

#define \_\_LIBRARY\_\_

#include <stdio.h>

#include <unistd.h>

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

{

int pid;

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

pid = fork();

if( pid != 0 )

{

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

}

else

{

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

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

return 0;

}

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

return 0;

}

3. 使用命令gcc main.c -o app生成可执行文件app。

文本

描述已自动生成

图 1 生成app可执行文件

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。



图 2 app运行结果

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);

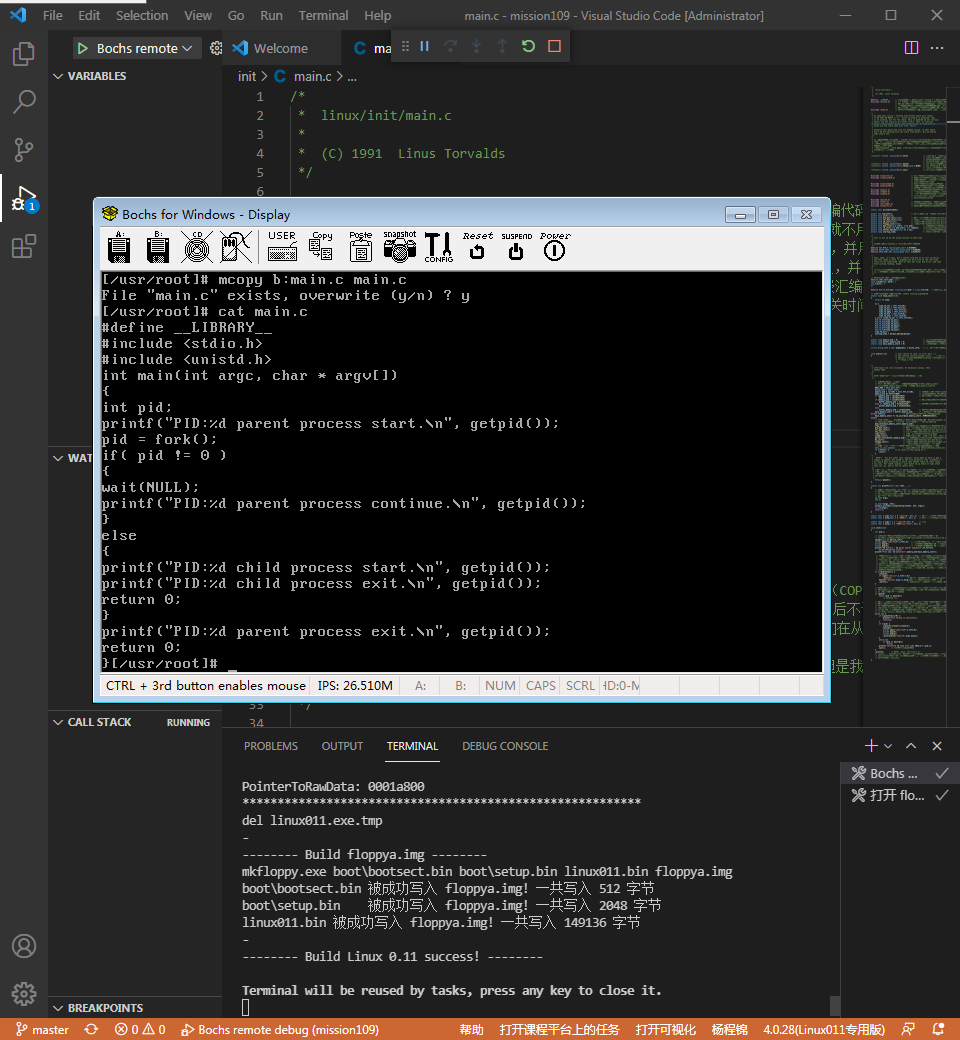


图 3 更改main.c

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。



图 4 新app运行结果

**查看父进程与子进程的运行轨迹**

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。

电脑萤幕的截图

描述已自动生成电脑萤幕的截图

描述已自动生成电脑萤幕的截图

描述已自动生成

图形用户界面, 应用程序

描述已自动生成图形用户界面

描述已自动生成图形用户界面, 文本

描述已自动生成

图形用户界面, 文本

描述已自动生成

图 5 Linux启动击中断点

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。

图形用户界面, 文本

描述已自动生成图形用户界面

描述已自动生成

图 6 子函数击中断点

图形用户界面

描述已自动生成

图 7 查看进程运行轨迹

**调试跟踪fork函数的执行过程**

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

电脑屏幕的截图

描述已自动生成

图 8 记录app文件大小

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==24900。

文本

描述已自动生成

图 9 插入断点

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。

文本

描述已自动生成

图 10 app命中断点

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11所示。

文本

描述已自动生成

图 11 current值

3. 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12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。

文本

描述已自动生成

图 12 task值

4. 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13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。

图形用户界面, 文本

描述已自动生成

图 13进程列表

5. 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。

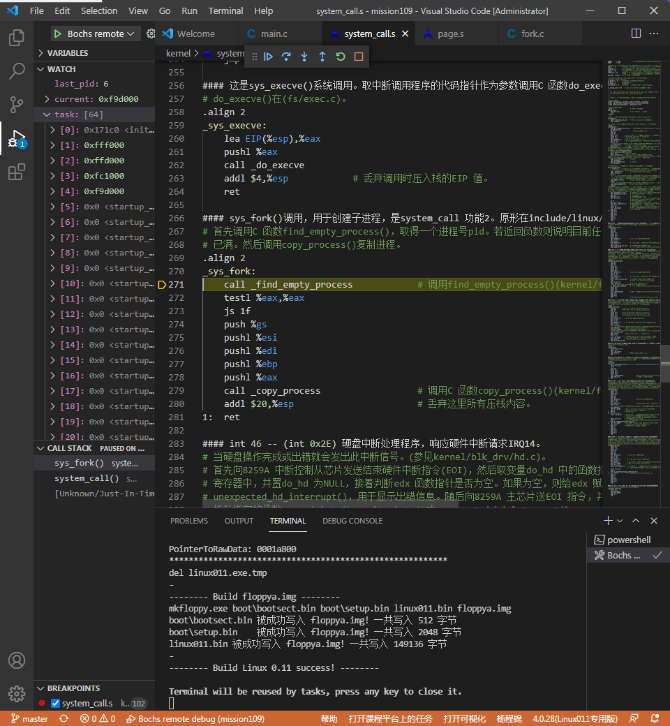


图 14 fork系统调用的内核函数

6. 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。

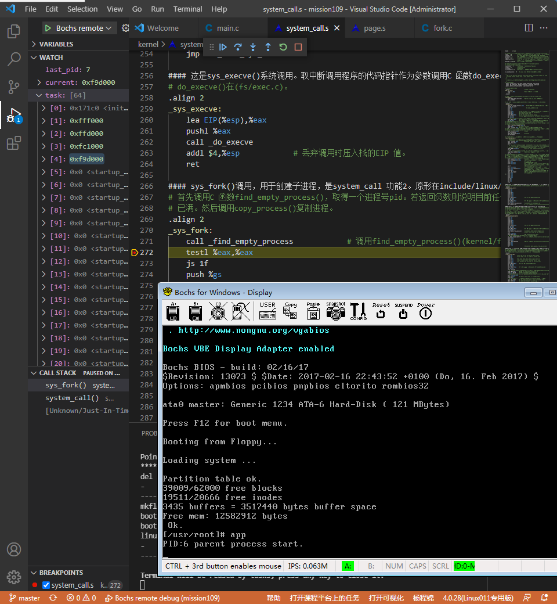


图 15 lastpid改变

7. 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。

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。

文本

描述已自动生成

图 16 进入copy\_process函数

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）。

文本

描述已自动生成

图 17 单步调试至103行

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。

文本

描述已自动生成文本

描述已自动生成

图 18 父子进程资源查看

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。

文本

描述已自动生成

图 19 调试至171行

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。

* 1. **调用execve函数加载执行一个新程序**

**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，编写如下的代码。

#define \_\_LIBRARY\_\_

#include <stdio.h>

#include <unistd.h>

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

{

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

return 0;

}

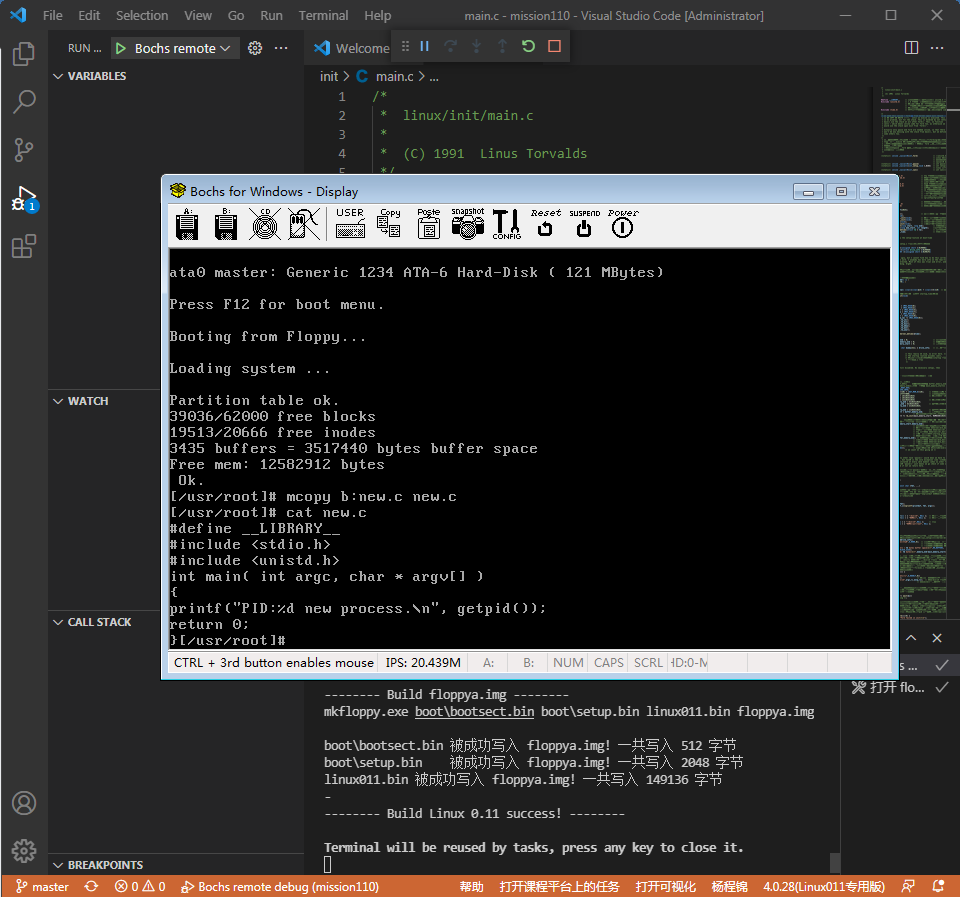


图 20 new.c文件

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。

文本

描述已自动生成

图 21 运行new可执行文件

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，编写如下的代码。

#define \_\_LIBRARY\_\_

#include <stdio.h>

#include <unistd.h>

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

{

printf("PID:%d old process start.\n", getpid());

execve("new", NULL, NULL );

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

return 0;

}

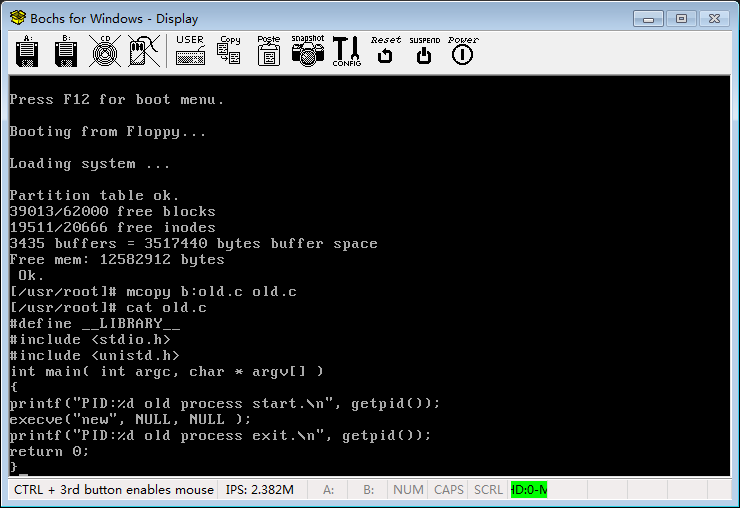


图 22 old.c文件

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。

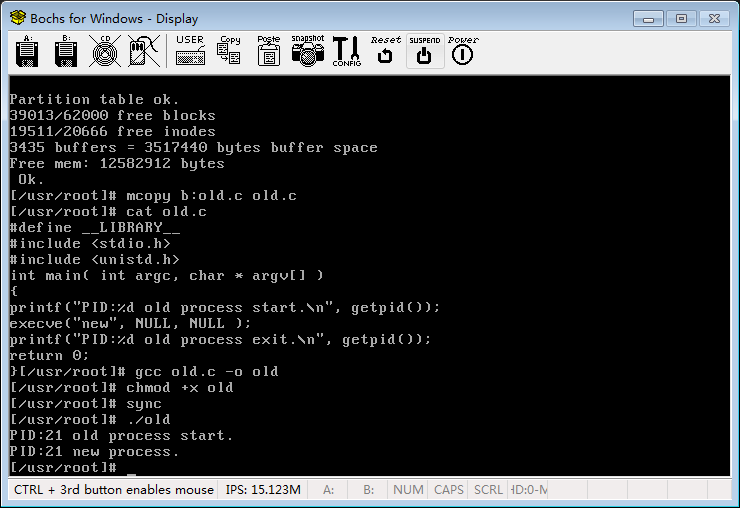


图 23 运行old可执行文件

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。

**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

文本

描述已自动生成

图 24 查看old文件大小

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。

电脑屏幕截图

描述已自动生成

图 25 添加断点

4. 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）。

5. 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。

电脑萤幕的截图

描述已自动生成

图 26 命中断点

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。

文本

描述已自动生成

图 27 单步调试

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。

文本

描述已自动生成

图 28 单步调试

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。

文本

描述已自动生成

图 29 单步调试

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。

文本

描述已自动生成

图 30 Run to Cursor

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。

文本

描述已自动生成

图 31 Run to Cursor

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。

文本

描述已自动生成

图 32 单步调试

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。

文本

描述已自动生成

图 33 Run to Cursor

8. 按F10单步调试，do\_execve函数返回到sys\_execve函数。

文本

描述已自动生成

图 34 单步调试

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。

文本

描述已自动生成

图 35 old执行结束

10. 结束调试，关闭Bochs虚拟机。

1. **实验的思考与问题分析**

1、答：

#define \_\_LIBRARY\_\_

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

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

int pid, i;

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

for (i = 0; i < 10; i++) {

pid = fork();

if (pid == 0) {

// 子进程

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

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

return 0;

}

// 父进程不做任何事情，继续下一次循环

}

// 父进程等待所有子进程结束

for (i = 0; i < 10; i++) {

wait(NULL);

}

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

return 0;

}

2、答：

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

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

int pid;

pid = fork();

if (pid == 0) {

// 子进程

char \*newargv[] = { NULL };

char \*newenviron[] = { NULL };

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

// 使用 execve 加载并运行新程序

execve("path\_to\_new\_program", newargv, newenviron);

// execve 只有在出错的情况下才会返回

perror("execve");

return 0;

} else {

// 父进程

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

// 等待子进程结束

wait(NULL);

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

}

return 0;

}

1. **总结和感想体会**

此实验旨在掌握创建子进程和加载执行新程序的方法，并理解二者的不同。主要步骤包括使用fork函数创建子进程，编写代码使得父进程和子进程执行不同的逻辑，并分析程序的运行结果。实验还包括了使用execve系统调用加载执行一个新程序，分析了fork和execve的区别与联系。通过这些步骤，实验帮助我理解进程创建、子进程的产生以及新程序的加载执行过程。实验还涉及到调试跟踪fork和execve系统调用函数的执行过程，以及使用可视化工具观察父子进程的创建和结束顺序​​。