合肥工業大學

操作系统实验报告

实验题目	进程的创建
学生姓名	袁子超
学　　号	2023217509
专业班级	智能科学与技术 23-1
指导教师	田卫东、罗月童
完成日期	2025.12.25

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。

2 实验原理

Linux 0.11 中进程创建与程序加载执行的核心依赖 fork 和 execve 两个系统调用，二者遵循“复制-替换”的底层逻辑。fork 系统调用通过复制父进程的进程控制块、内存空间（代码段、数据段、栈）等资源创建子进程，子进程继承父进程的绝大多数属性（优先级、打开文件、终端等），仅进程号、父进程号等少数字段不同，且 fork 在父进程中返回子进程号、在子进程中返回 0 以区分父子流程；execve 系统调用则会彻底替换当前进程的内存镜像，释放原有代码段、数据段，加载新程序的可执行文件，初始化新的堆栈和程序入口，使进程执行全新逻辑，且成功执行后不会返回原程序。父进程可通过 wait 系统调用阻塞自身，等待子进程终止后再继续运行，以此实现进程间的同步控制，这一机制共同构成了 Linux 0.11 进程管理与程序执行的核心基础。

3 实验内容

3.1 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：

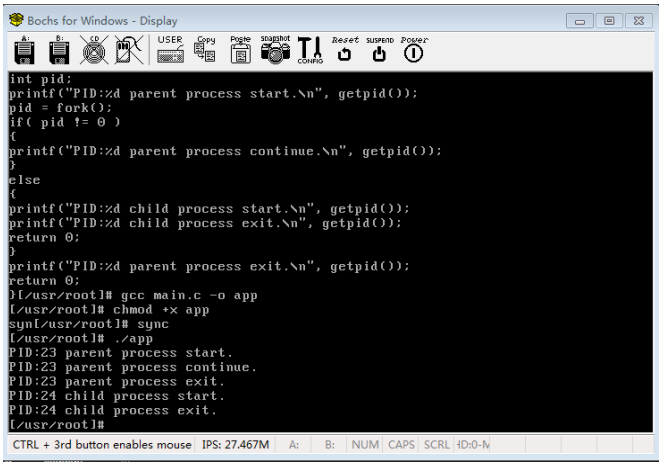
1. 按 F5 启动调试。

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。
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。



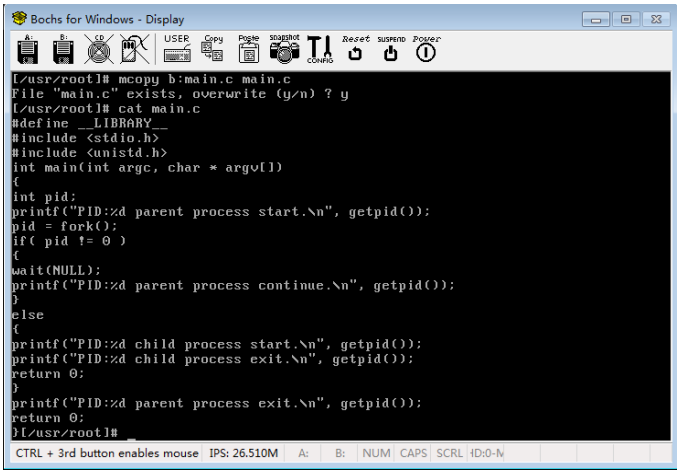
app 运行结果

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)

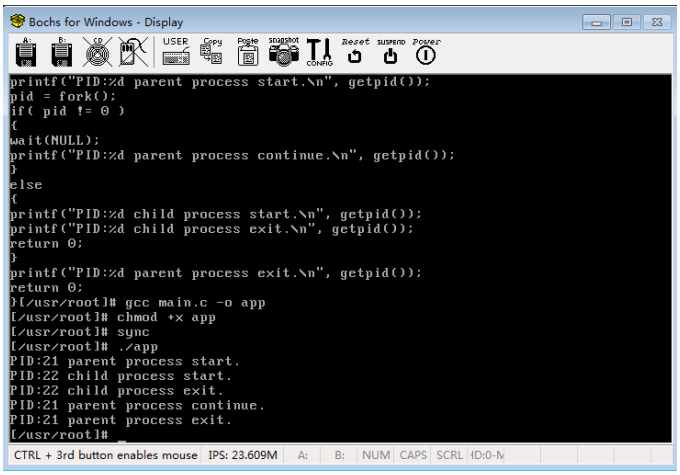
按照下面的步骤继续使用 vi 编辑器修改 main.c 文件：

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



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。



app 运行结果

查看父进程与子进程的运行轨迹

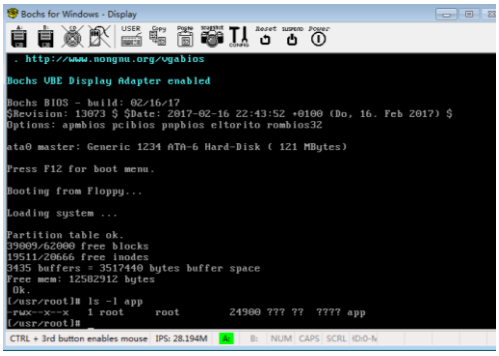
1. 为了方便观察父进程和子进程的运行轨迹，需要在进程结束的位置添加一个断点。请读者在 kernel/exit.c 文件的第 166 行添加一个断点即可，这里就是一个进程在结束运行后，让调度程序选择其他进程继续运行的代码。

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。

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

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。



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。

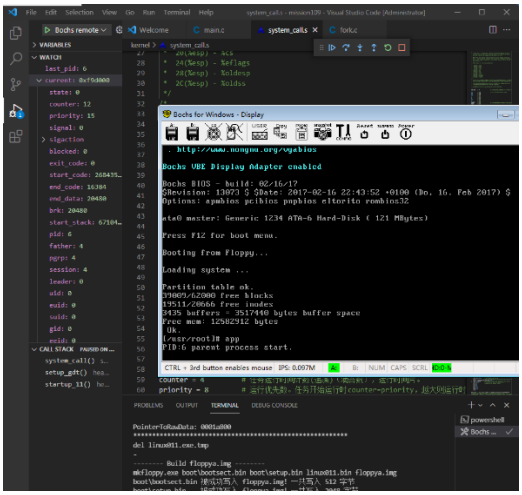
5. 按 F5 启动调试（注意，由于添加了一个条件断点，调试器需要频繁验证条件是否满足，这会导致启动过程明显变慢，请读者耐心等待启动完毕）。

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。

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：

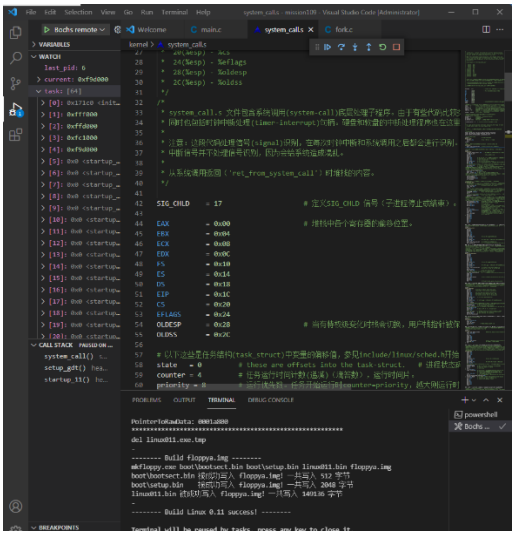
1. 在“WATCH”窗口添加 last_pid 和 current->pid，查看它们的值。全局变量 last_pid（在文件 kernel/fork.c 的第 30 行定义）记录了最新的进程号。current->pid 的值是当前正在运行的进程的进程号，也就是 app 应用程序的进程号。

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。



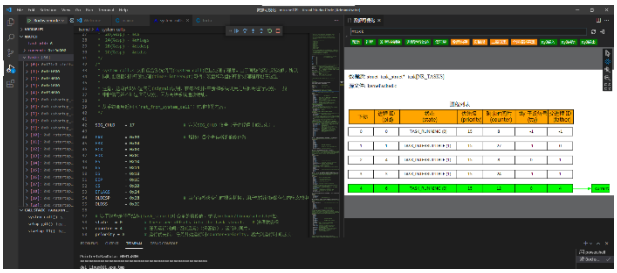
current 值

2. 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。



task 值

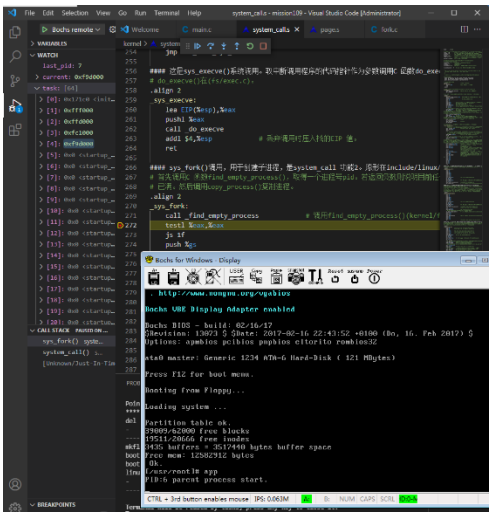
3. 在 VSCode 的“View”菜单中选择“Command Palette...”，会在 VSCode 的顶部中间位置显示命令面板，输入“Lab: New Visualizer View”命令后，VSCode 会在其右侧弹出一个窗口让读者查看可视化视图。在右侧可视化视图顶部的编辑框中输入命令“#task”后按回车，就可以查看进程列表了，如图 13 所示。其中背景色为绿色并且使用 current 游标指向的进程是当前进程，其 state 字段的值为 0 表示当前进程（即使用可执行文件 app 创建的进程）正处于运行态；counter 的值为 11 表示其剩余时间片的大小；priority 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。



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。

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。



lastpid 变化

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。

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。

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。

3.1 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。

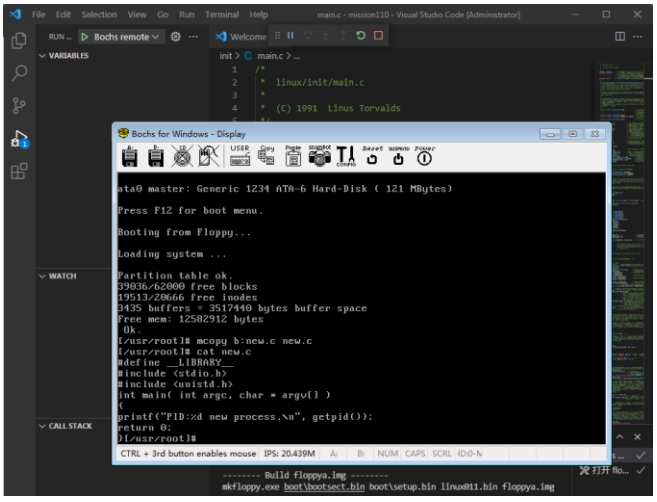
调用 execve 函数加载执行一个新程序

首先编写一个供 execve 函数加载的应用程序：

1. 按 F5 启动调试。

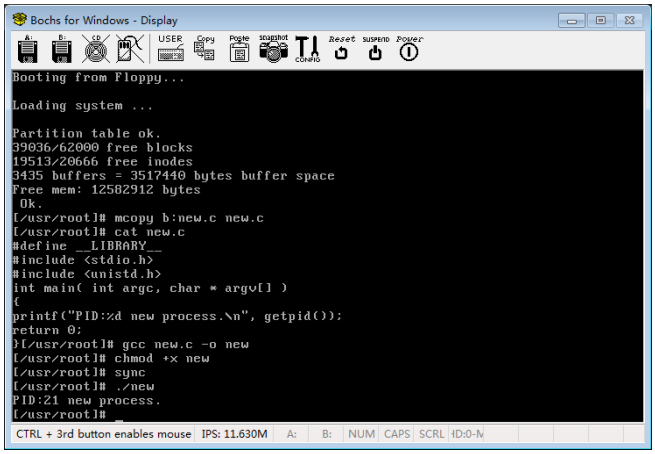
2. 待 Linux 0.11 启动后，使用 vi 编辑器新建一个 new.c 文件，编写如下的代码。

```
#define __LIBRARY__
#include <stdio.h>
#include <unistd.h>
int main( int argc, char * argv[] )
{
printf("PID:%d new process.\n", getpid());
return 0;
```



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。



运行 new 可执行文件

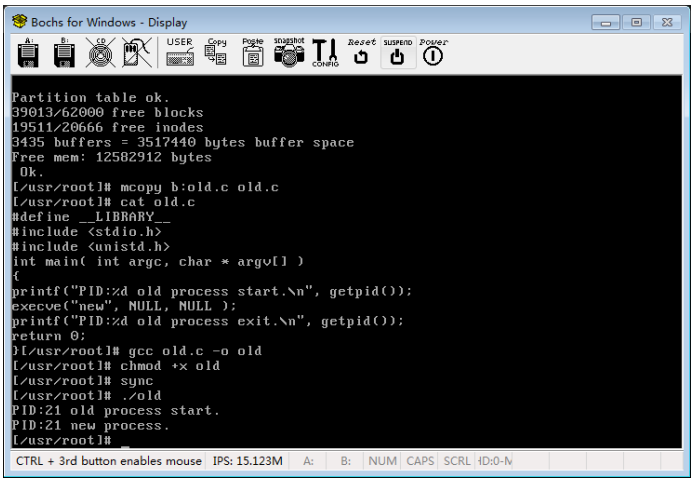
接下来编写调用 execve 函数的应用程序：

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;
}
```

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。



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

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。

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。

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。

4 实验的思考与问题分析

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。

```
#define __LIBRARY__
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    int pid, i;

    // 打印父进程启动信息
    printf("PID:%d parent process start.\n", getpid());

    // 循环创建 10 个子进程
    for (i = 0; i < 10; i++) {
        pid = fork();
        if (pid == 0) {
            // 子进程逻辑：输出自身 PID 后立即退出
            printf("PID:%d child process start.\n", getpid());
            printf("PID:%d child process exit.\n", getpid());
            return 0; // 子进程退出，避免继续创建子进程
        }
        // 父进程继续执行循环，创建下一个子进程
    }

    // 父进程等待所有 10 个子进程结束
    for (i = 0; i < 10; i++) {
```

```
            wait(NULL); // 阻塞等待任意子进程退出，循环 10 次确保全部等待完成
        }

        // 所有子进程结束后，父进程退出
        printf("PID:%d parent process exit.\n", getpid());
        return 0;
    }
```

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。

答：

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

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

    // 创建子进程
    pid = fork();
    if (pid == 0) {
        // 子进程逻辑
        char *newargv[] = { NULL };     // 传递给新程序的参数列表（无参数）
        char *newenviron[] = { NULL }; // 传递给新程序的环境变量（无环境变量）

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

        // 调用 execve 加载并执行指定可执行文件
        // 需将"path_to_new_program"替换为实际的程序路径（如"./test"）
        execve("path_to_new_program", newargv, newenviron);

        // 仅当 execve 执行失败时才会执行以下代码
        perror("execve"); // 输出错误信息
        return 1;               // 子进程异常退出
    } else if (pid > 0) {
        // 父进程逻辑
        printf("PID:%d parent process wait.\n", getpid());

        // 阻塞等待子进程退出
        wait(NULL);

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

```
        // fork 失败处理
        perror("fork");
        return 1;
    }

    return 0;
}
```

5 总结和感想体会

围绕“进程的创建与程序加载执行”，通过在 Linux 0.11 环境下实践 fork 和 execve 系统调用，我不仅掌握了具体的技术方法，更深入理解了操作系统进程管理的核心逻辑。

通过本次实验，我不仅夯实了操作系统进程管理的理论基础，更锻炼了实践操作和内核调试能力。我深刻认识到，操作系统的核心功能往往通过简洁高效的机制实现，而深入内核源码、跟踪代码执行流是理解这些机制的关键。这次实验的经历，为我后续学习更复杂的操作系统概念（如进程调度、内存管理、文件系统）奠定了坚实的基础，也让我对计算机系统的底层工作原理产生了更浓厚的探索兴趣。