**实验三 – 基于BPF机制的系统跟踪与探测**

2022年8月29日

1. **实验内容**
2. 学习BPF机制，了解BCC（BPF Compiler Collection）和bpftarce的实现原理；
3. 利用BPF工具实现对系统的跟踪和探测，如跟踪新创建的进程，统计线程占用CPU的时间，统计某内核函数的调用次数，跟踪系统中的内存调用和页错误，跟踪文件打开信息等。
4. **实验目的**

通过实验，让学生了解BPF机制，掌握利用BPF工具对系统进行跟踪与探测的方法。

1. **设计思路和流程图**
2. 了解BPF

首先我们需要了解BPF的背景知识：

BPF是Berkely Packet Filter（伯克利数据包过滤器）的缩写，是用来提升pcap过滤性能的，比当时最快的包过滤技术快20倍，只所以性能高，是因为它工作在内核中，避免包从内核态复制到用户态所以速度快，后来Alexei Starovoitov 大牛在2014年重新实现了BPF，将其扩展成了通用的执行引擎，称为eBPF，官方缩写仍是BPF。

为什么需要有BPF呢？

多年前很多程序，例如网络监控器，都是作为用户级进程运行的。为了分析只在内核空间运行的数据，它们必须将这些数据从内核空间复制到用户空间的内存中去，并进行上下文切换。这与直接在内核空间分析这些数据相比，导致了巨大的性能开销。

而随着近年来网络速度和流量井喷式增长，一些应用程序必须处理大量的数据（如音频、视频流媒体数据）。要在用户空间监控分析那么多的流量数据已经不可行了，因而BPF应运而生——一种在内核空间执行高效安全的程序的机制。

简单解释BPF作用，它提供了一种当内核或应用特定事件发生时候，执行一段代码的能力。BPF 采用了虚拟机指令规范，所以也可以看一种虚拟机实现，使我们可以在不修改内核源码和重新编译的情况下，提供一种扩展内核的能力的方法。

BPF程序不像一般程序可以独立运行，它是被动运行的，需要事件触发才能运行，有点类似js里面的监听，监听到按钮点击执行一小段代码。这些事件包括系统调用，内核跟踪，内核函数，用户函数，网络事件等。

1. 环境搭建与配置

在了解完BPF的背景知识与作用后就可以进行实验了。前面两次实验一直使用的是Fedora 7系统，内核简单编译快，但在本次实验的环境配置时，发现很多功能并不能很好的支持，而且在查阅资料的过程中发现很多地方都提到eBPF需要较新的Linux内核来支持，所以改用较新的系统，这里选用了SEED-Ubuntu20.04系统。



1. 安装BCC与bpftrace

**安装BCC：**

ubuntu 系统下需要执行：

*sudo apt install bpfcc-tools linux-headers-$(uname -r)*

或者：

*sudo snap install bcc*

RedHat系统下需要执行：

*sudo yum install bcc-tools*

也可以直接使用官方封装好的 [docker](https://cloud.tencent.com/product/tke?from=10680) 镜像来启动：

*sudo docker run -it -v /usr/src:/usr/src:ro -v /lib/modules/:/lib/modules:ro -v /sys/kernel/debug/:/sys/kernel/debug:rw --net=host --pid=host --privileged quay.io/iovisor/bcc:latest bash*

安装完成后，BCC 工具通常会被默认放置在 /sbin 目录下，并且以 -bpfcc 为后缀，或者在 /usr/share/bcc/tools 目录下。

**安装bpftrace：**

ubuntu 下需要执行：

*sudo apt install bpftrace*

fedora 下需要执行：

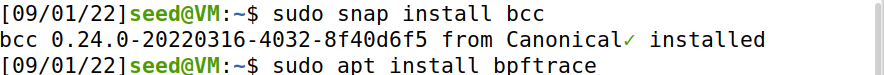
*sudo dnf install -y bpftrace*

同样，也可以使用 docker 镜像来启动：

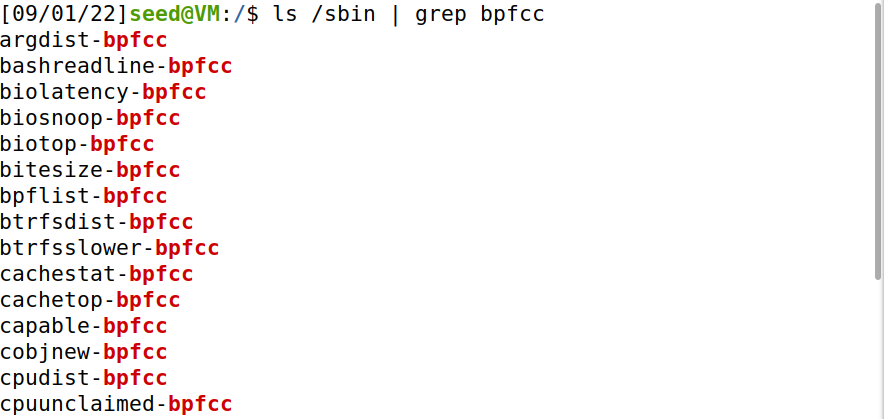
*sudo docker run -it -v /usr/src:/usr/src:ro -v /lib/modules/:/lib/modules:ro -v /sys/kernel/debug/:/sys/kernel/debug:rw --net=host --pid=host --privileged quay.io/iovisor/bpftrace:latest bash*

安装完成后，bpftrace 的二进制文件通常会放到 /usr/local/bin/bpftrace，bpftrace 工具则会被安装到 /usr/local/share/bpftrace/tools 中，或者直接放置到 /usr/local/bin 下。

输入命令*sudo snap install bcc*和*sudo apt install bpftrace*进行安装：

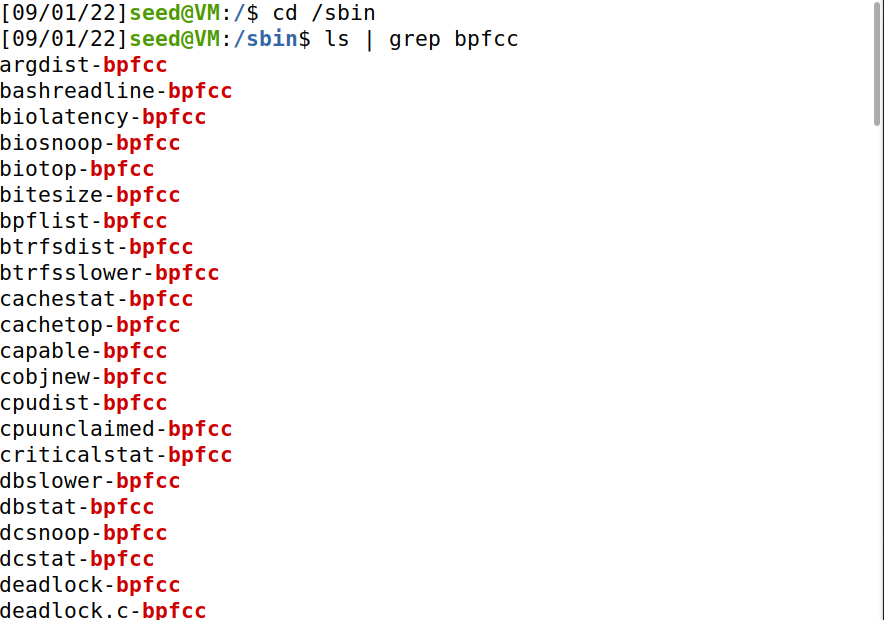


输入*ls /sbin | grep bpfcc*，看到/sbin路径下已经有了安装好的BCC 工具：



1. 部分BCC工具介绍以及功能演示

进入/sbin路径下，输入*ls | grep bpfcc*，显示所有BCC工具：

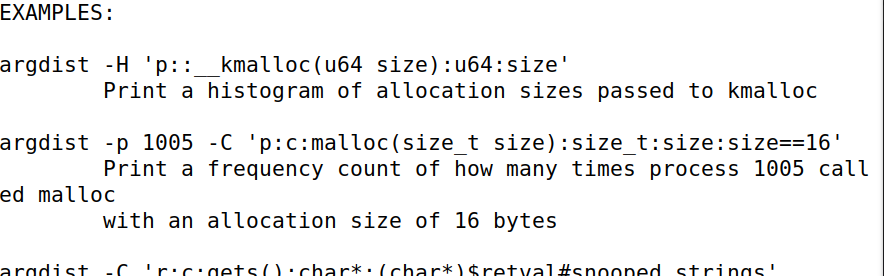


1. **argdist-bpfcc：**

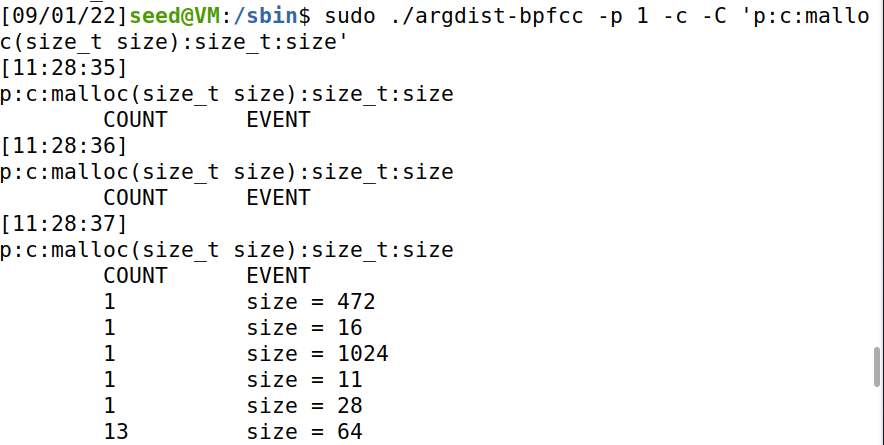
以第一个工具argdist为例。

argdist工具用于探测指定的函数，并收集参数到一个直方图和频率计数器。可以在没有调试器的情况下，过滤并打印敢兴趣的参数，从而理解指定参数的分布图。

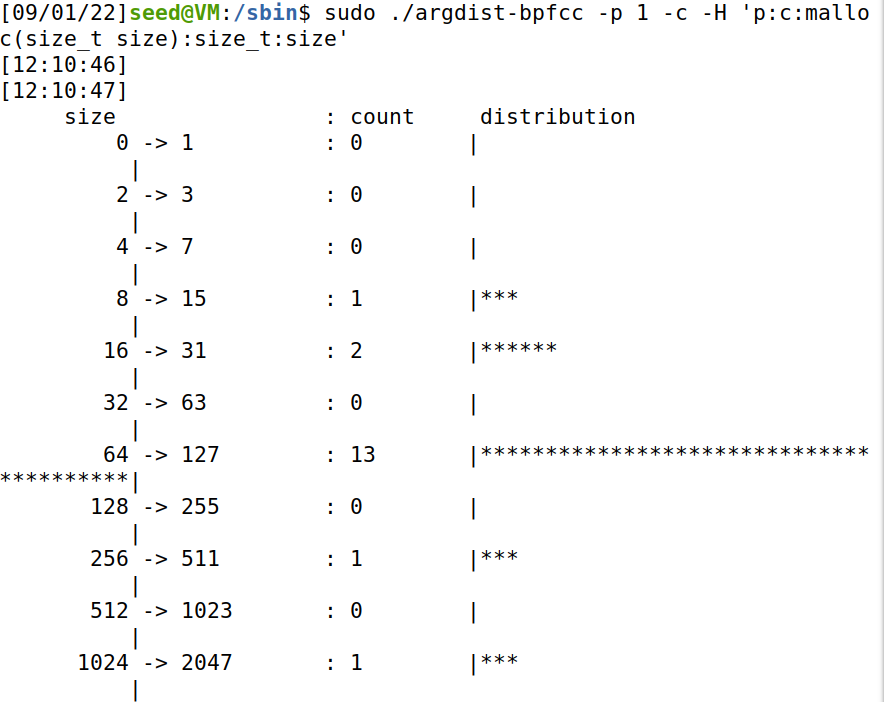
输入*./argdist-bpfcc -h*可以查看该工具的介绍以及一些使用样例：



比如输入*sudo ./argdist-bpfcc -p 1 -c -C 'p:c:malloc(size\_t size):size\_t:size'*，其中的1为进程号可以查看一个进程中的内存分配块大小情况，如下图所示：

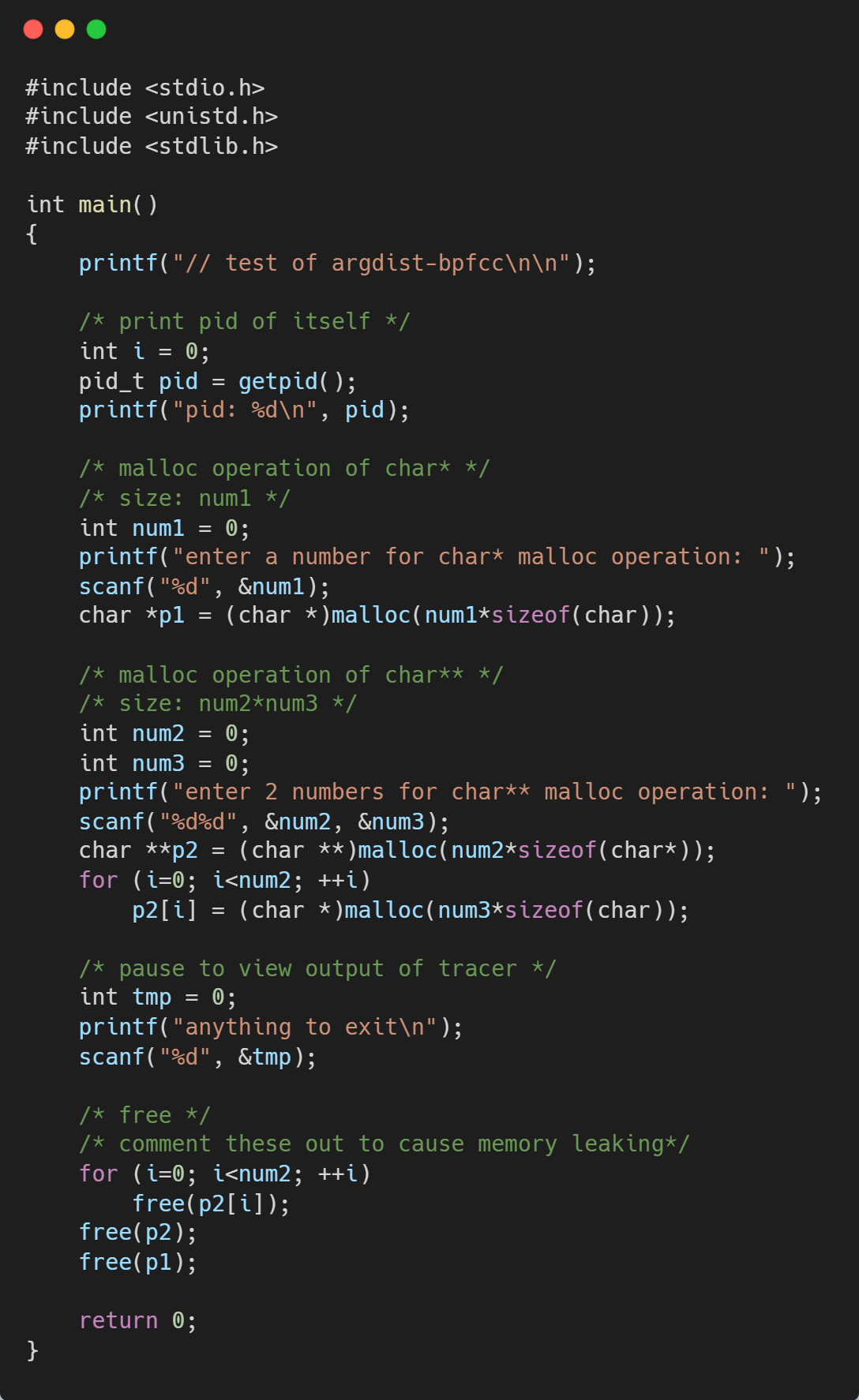


将上述命令中的*-C*参数改为*-H*，即输入*sudo ./argdist-bpfcc -p 1 -c -C 'p:c:malloc(size\_t size):size\_t:size'*，可以柱形图形式显示内存分配情况，如下图所示：



下面，我们自己编写一个程序，分配一定的内存空间，然后再使用该工具来监测内存分配情况，查看是否与分配相同。

程序代码如下，前面命令的执行需要进程号为依据，所以在测试程序中获取自己的进程号并打印，然后先后进行char\*和char\*\*的分配，最后释放内存：



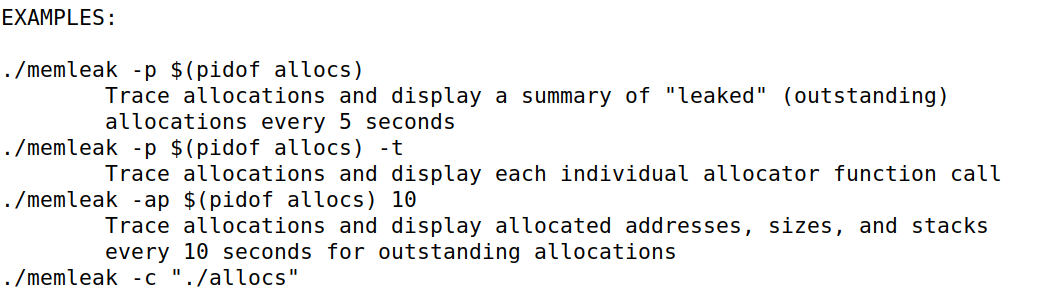
具体测试及结果分析见报告第六部分。

1. **memleak-bpfcc：**

memleak 是一个C语言内存泄漏检测工具。 原理是利用 C 语言的宏调用来替代原有的函数调用， memleak 维护了一个链表，在这个链表中保存着程序中对内存函数

调用的记录，这些函数包括：malloc、calloc、realloc、free。

输入./memleak-bpfcc –h可以获取详细介绍和使用示例：



在实际应用中，可以在shell中输入top，查看内存占用一直在增长的程序，将进程号作为memleak的参数输入，来查看分析该进程是否出现内存泄漏，还可以定位内存泄露发生的位置。

这里继续使用前面的test1.c程序做内存分配与释放的测试。

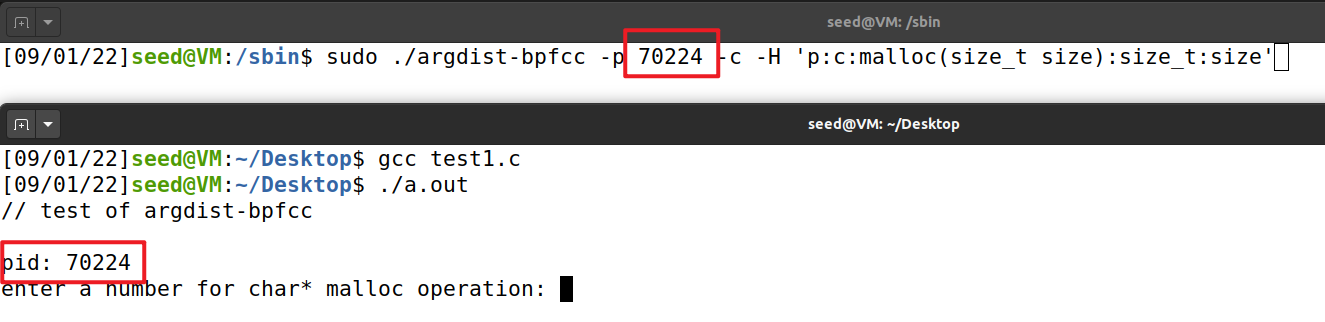
具体测试及结果分析见报告第六部分。

1. **源程序并附上注释**

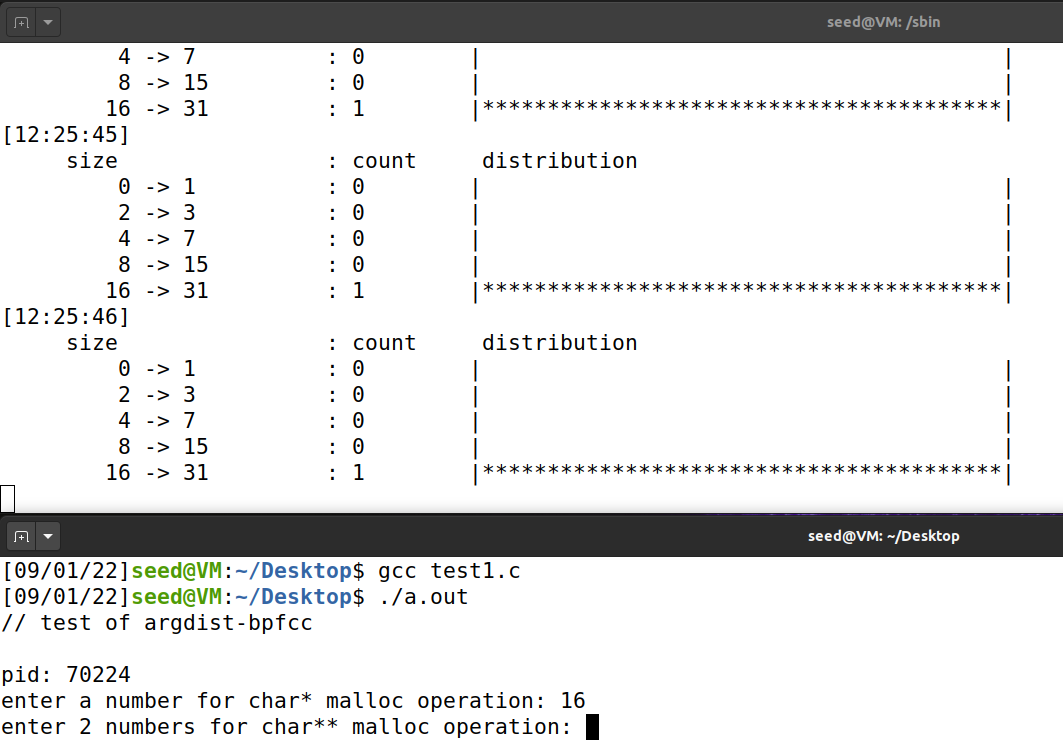
这是实验中测试所用的test1.c程序：

1. #include <stdio.h>
2. #include <unistd.h>
3. #include <stdlib.h>
5. **int** main()
6. {
7. printf("// test of argdist-bpfcc\n\n");
9. /\* print pid of itself \*/
10. **int** i = 0;
11. pid\_t pid = getpid();
12. printf("pid: %d\n", pid);
14. /\* malloc operation of char\* \*/
15. /\* size: num1 \*/
16. **int** num1 = 0;
17. printf("enter a number for char\* malloc operation: ");
18. scanf("%d", &num1);
19. **char** \*p1 = (**char** \*)malloc(num1\***sizeof**(**char**));
21. /\* malloc operation of char\*\* \*/
22. /\* size: num2\*num3 \*/
23. **int** num2 = 0;
24. **int** num3 = 0;
25. printf("enter 2 numbers for char\*\* malloc operation: ");
26. scanf("%d%d", &num2, &num3);
27. **char** \*\*p2 = (**char** \*\*)malloc(num2\***sizeof**(**char**\*));
28. **for** (i=0; i<num2; ++i)
29. p2[i] = (**char** \*)malloc(num3\***sizeof**(**char**));
31. /\* pause to view output of tracer \*/
32. **int** tmp = 0;
33. printf("anything to exit\n");
34. scanf("%d", &tmp);
36. /\* free \*/
37. /\* comment these out to cause memory leaking\*/
38. **for** (i=0; i<num2; ++i)
39. free(p2[i]);
40. free(p2);
41. free(p1);
43. **return** 0;
44. }
45. **程序运行结果及分析**
46. **argdist-bpfcc：**

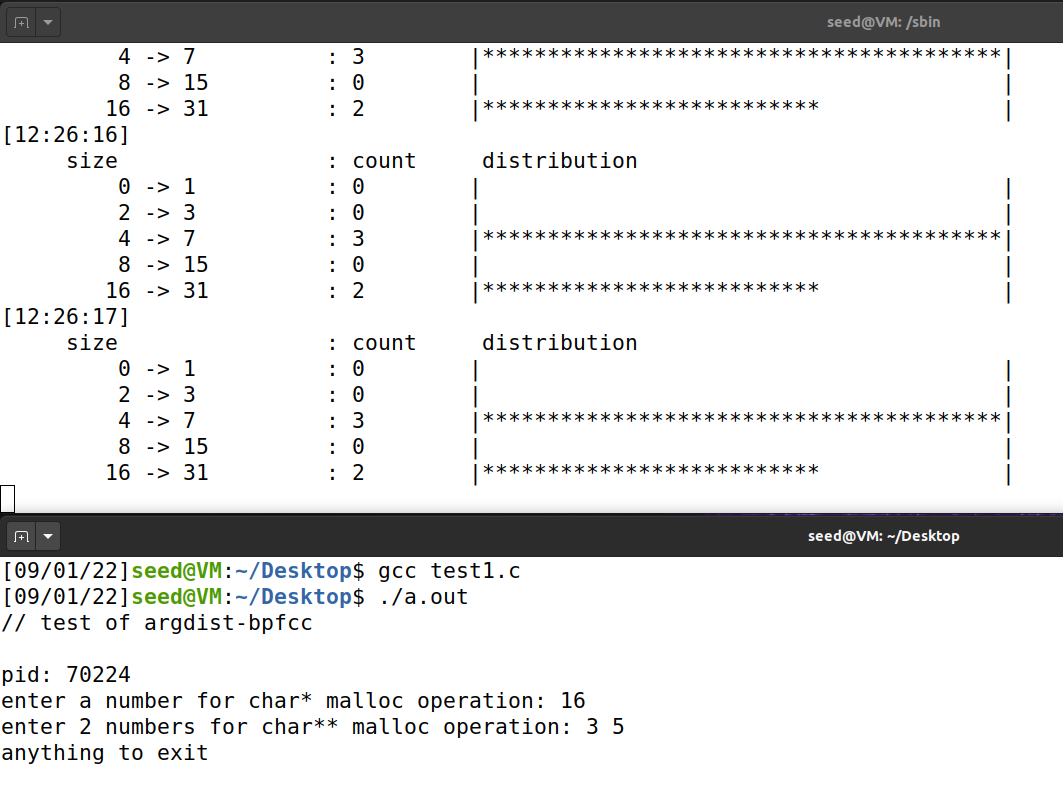
编译并运行test1.c程序，将获取到的进程号填入argdist的命令中，运行：



为char\*分配16的空间，可以在上面的shell中看到相应的内存分配情况：



为char\*\*分配3\*5的空间，同样可以看到有相应的输出：

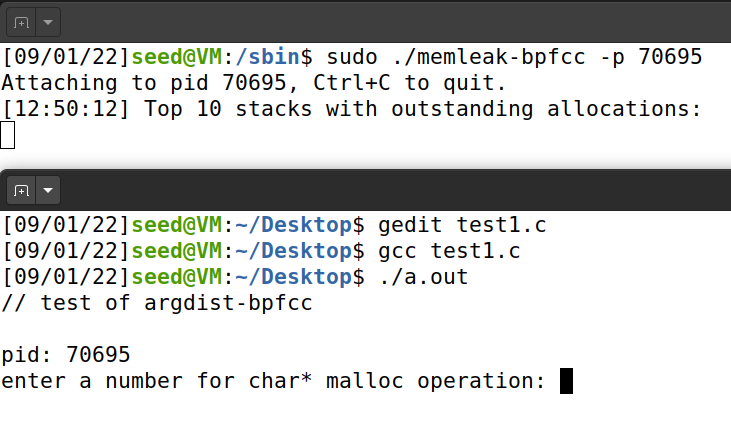


输入任意内容结束程序。

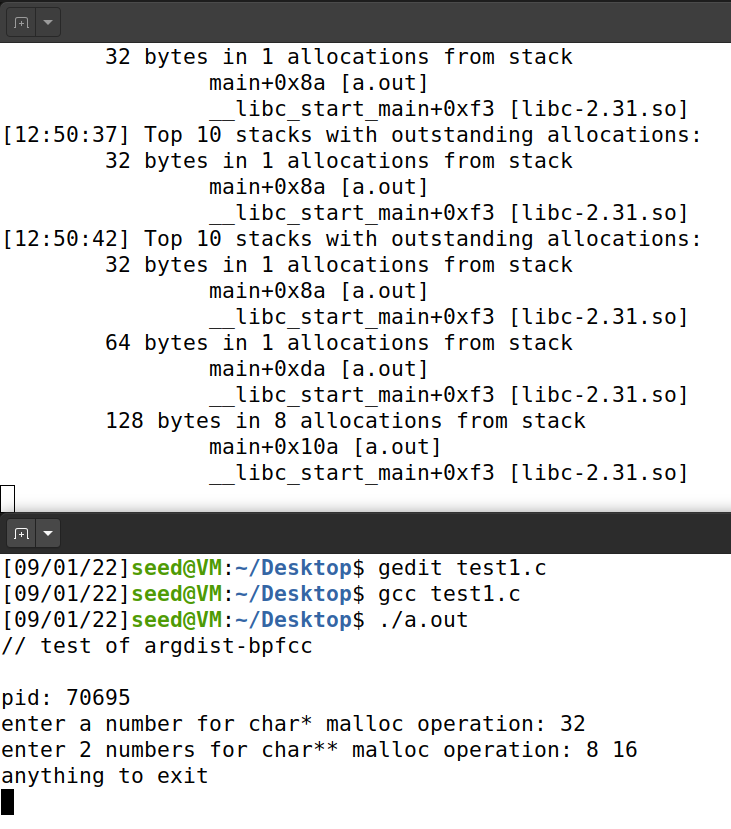
但发现尽管已经正确地free()了已分配的空间，但监视器中没有减少，查到的解释有两种：一是argdist是显示出已经分配过的内存，另一种是linux系统下free()后的内存不会释放回OS，而是释放回系统的内存缓冲池，进程退出时才释放回OS，但进程结束后仍无变化，所以倾向于第一种。

1. **memleak-bpfcc：**

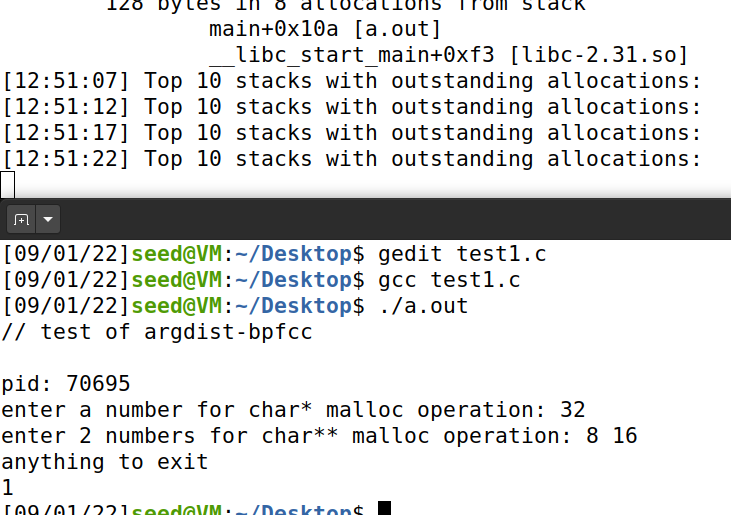
运行test1.c程序，将获取到的进程号写入memleak的命令中：*sudo ./memleak-bpfcc -p 70695*，执行命令，可以看到已经开始监视内存泄露情况：



输入，进行内存的分配，可以看到在监视器中有了相应的输出，说明这一进程有内存泄漏的风险：

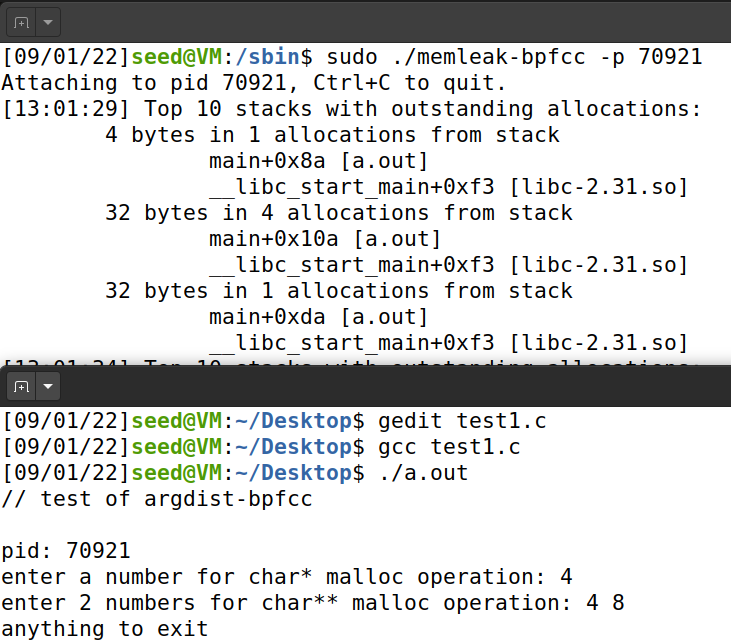


结束test1.c程序，释放已经分配的内存，再次查看监视器，发现又变为空了，没有发生内存泄漏：

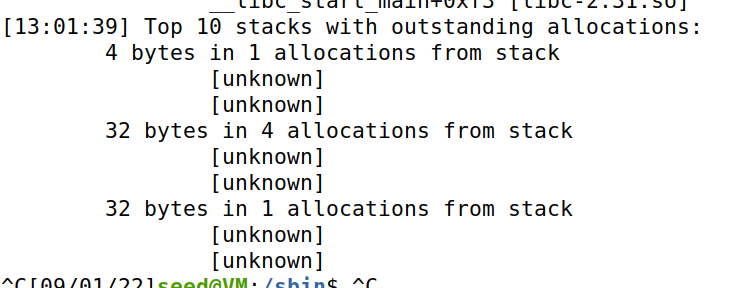


接下来，将测试程序的代码修改一下，看看不释放内存会怎么样，把程序中free()的部分注释掉：

编译后，再次运行程序，将进程号70921填入命令参数中，分配一些空间，先不着急结束程序运行，可以在监视器中看到对应的输出内容：



下面我们输入任意内容，结束程序的运行，注意这里的程序没有了最后释放内存的步骤，然后看到这次监视器的输出与前面一次不同了，分配过的空间还留在监视器中，但所属进程等信息变为了未知，可以看到发生了内存泄漏，与预期是一致的：



BCC工具的测试完成。

1. **实验体会**
2. 实验中遇到的问题及解决过程

遇到的困难主要是在BCC、BPFTracer工具的安装上，一开始使用的是原来的Fedora7系统的虚拟机，但是因为版本太老很多功能都不支持，在多次尝试都没找到解决办法后换到ubuntu 20.0.4，一下解决了很多问题，也才发现BCC需要较新的内核版本来支持。在安装过程中也有一些小问题，不过都比较容易解决，大部分是缺少某些依赖项，安装即可。

在测试程序的编写上比较顺利，因为前面操作系统课程学习、实验了很多，所以对于pid、malloc、free这些的用法还比较熟悉。

BCC工具可以在通过输入-h查看帮助和使用方法，也可以网上查阅具体的使用示例和介绍，比较方便。

1. 实验的体会及收获

了解到了BPF机制以及了解了一些相关工具的功能、使用方法等，挑了两个与内存分配相关的工具来进行更详细的学习与测试，也是因为在编写程序中内存分配与回收的部分总是容易被忽视，但是错误的操作可能造成的后果却不容小觑。

通过这些工具，我们可以更清楚的监视程序的运行状况，发现程序中的错误或是对程序处理不当的地方做进一步优化。