**Linux系统下利用链表实现动态内存分配  
与  
用户程序读取内核空间内存管理实例实验**

1. **实验目的**
2. 了解静态内存与动态内存的区别；
3. 理解动态内存的分配和释放原理；
4. 掌握如何调整动态内存的大小；
5. 利用链表实现动态内存分配。
6. **实验环境**
7. macOS Mojave 10.14.1 (x86\_64-apple-darwin18.2.0)  
    Apple LLVM version 10.0.1 (clang-1001.0.46.4)
8. Ubuntu 18.04 (Linux 5.4.0-51-generic) x64  
    gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)   
    GNU Make 4.1
9. **实验内容**
10. 利用malloc和 calloc函数实现动态内存的分配；利用free函数实现动态内存的释放；
11. 利用realloc函数实现调整内存空间的大小；
12. 利用链表实现动态内存分配。
13. **实验原理 实验中用到的系统调用函数（包括实验原理中介绍的和自己采用的），实验步骤，**
14. 静态内存与动态内存

按分配内存空间的方式不同，一个程序所使用的内存区域可以分为静态内存与动态内存。在程序开始运行时有系统分配的内存称为静态内存，在程序运行过程中由用户自己申请分配的内存称为动态内存。

静态内存的申请是由编译器来分配的。对于用户程序中的各种变量，编译器在编译源程序时处理了为各种变量分配所需内存的工作。当程序执行时，系统就为变量分配所需的内存空间，至使用该变量的函数执行完毕返回时，自动释放所占用的内存空间。使用静态内存对用户来说是很方便的。用户并不需要了解分配内存的具体细节，也不需要时刻考虑由于程序结束前未释放所占用的内存空间而带来的可用内存泄漏。同时，静态内存也是不通过指针而使用变量的唯一方法。但是，静态内存也存在一定的缺陷：首先，由于静态内存总是预先定义了存放数据的数组大小，这就有可能因为所传入的数据量大于数组容量而引发的溢出问题。或因为定义了一个大数组，而所传入的数据量远小于数组容量，而对内存空间造成浪费。

其次，由于在某个函数中分配的静态内存将在此函数运行结束时被系统自动释放，使用指针由子函数向主函数传递数据的设想是无法被实现的。

使用动态内存时，用户可以根据需要随时申请内存，使用完毕后手动将此内存区释放。在实际应用中非常方便。但动态内存的使用也存在着巨大的隐患。任何处理过大型项目的用户都知道动态内存的使用会使内存管理变得多么复杂，以及要确切地记得在使用完毕后释放所占用的内存空间是多么困难的事情。在大型应用程序中，由于在释放某块动态内存前将指向该内存区域的指针重新赋值，从而使得此内存区域无法被释放的情况是十分常见的。通常将内存分配后没有被释放而导致可用内存减少称之为内存泄漏。避免内存泄漏耗尽系统资源正是许多服务器每隔一段时间就需要重新启动的原因。另外还要注意，由于分配动态内存时，用户得到的是一块void类型的内存，用户可以将其作为任何类型的内存空间使用，也可能引发一些无法预计的结果。

1. 动态内存的分配

分配动态内存空间所使用的函数调用如下：

#include<stdio.h>

void \*malloc(size\_t size);

void \*calloc(size\_t nmemb, size\_t size);

函数malloc和calloc都是用于分配动态内存空间的函数。

函数malloc的参数size表示申请分配的内存空间的大小，以字节记。

函数calloc的参数nmemb表示申请分配的内存空间占的数据项数目，参数size表示一个数据项的大小，以字节记。也就是说，calloc函数分配大小为nmemb\*size大小的内存空间。

函数calloc与函数malloc的最大区别就是函数calloc将初始化所分配的内存空间，把所有位置0。调用成功时，函数calloc与函数malloc的返回值都为被分配的内存空间的指针；调用失败时，返回值为NULL。

1. 动态内存的释放

释放动态内存空间所使用的函数调用如下：

#include<stdio.h>

void free(void \*ptr);

此函数的作用是释放由函数calloc或函数malloc分配的动态内存。参数ptr是指向要释放的动态内存的指针。

注意：当动态内存被释放后，原来指向它的指针就会变为悬空指针。此时使用该指针将会产生错误。

1. 调整动态内存的大小

对于用函数calloc与函数malloc分配好的动态内存，可以使用realloc函数来调整它的大小。该函数的说明如下：

#include<stdio.h>

void\*realloc(void \*ptr, size\_t size);

realloc函数的作用是重新调整一块动态内存区域的大小，参数ptr是指向要调整的动态内存的指针，应是函数calloc与函数malloc的返回值。参数size是新定义的动态内存的大小。Size可以大于或小于动态内存的原大小，调用realloc函数时，通常是在原来的内存空间调整动态内存的大小，原有数据不被改动。当size大于原大小，而原位置中无法完成调整时，将重新开辟内存空间并将原数据拷贝到新的内存空间中。

注意：如果参数ptr为NULL，则函数realloc的作用相当于函数malloc。如果参数size为0，则函数realloc的作用相当于函数free。

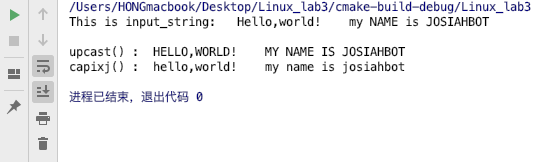
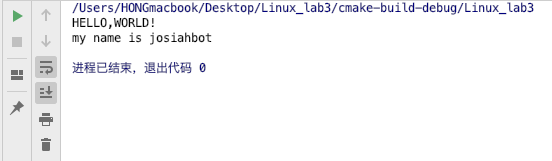
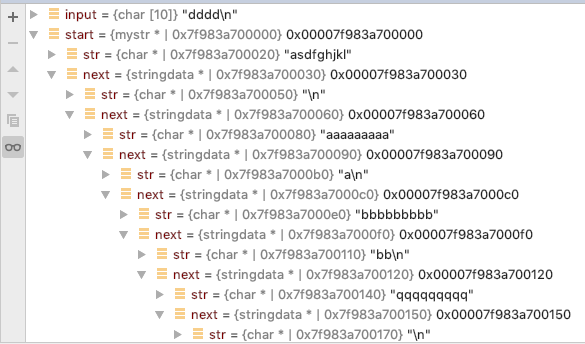
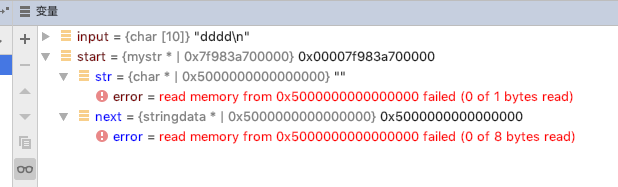
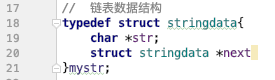
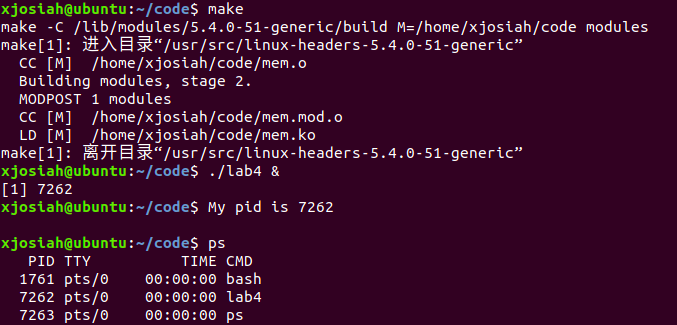
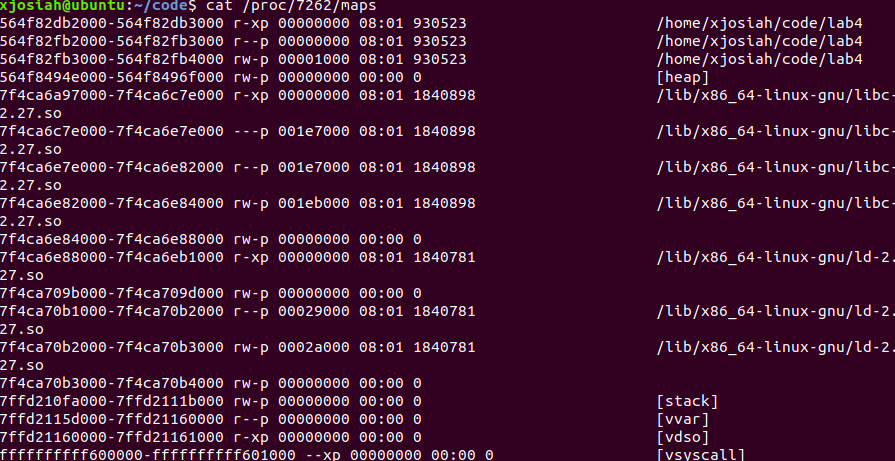
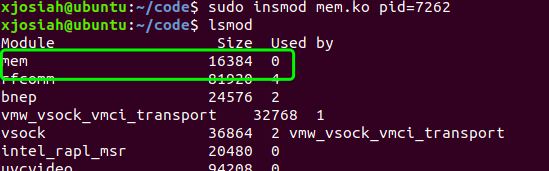
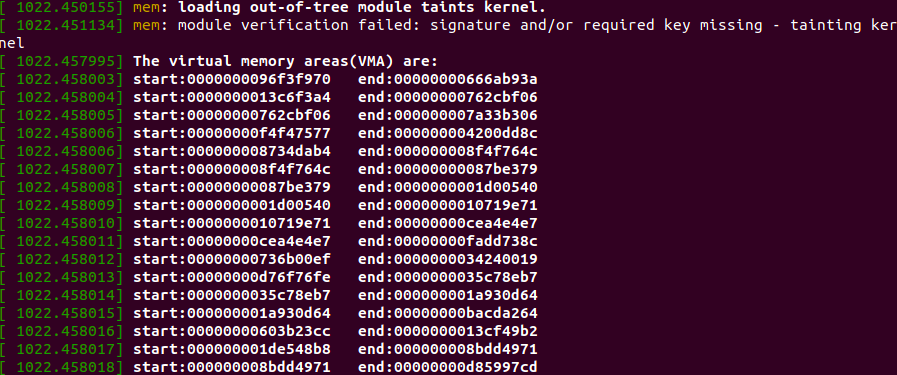
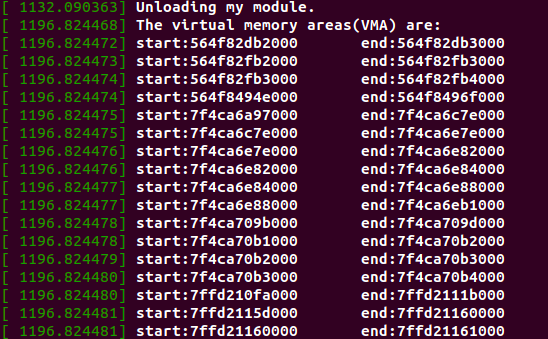
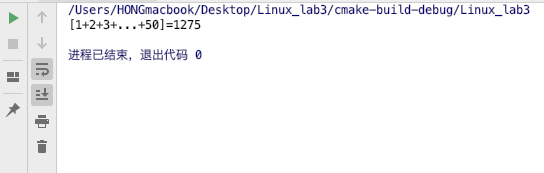
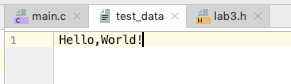
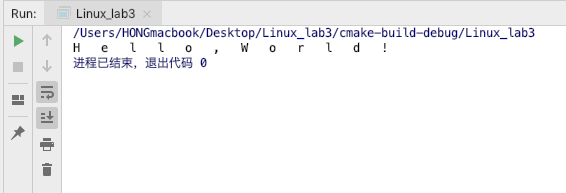
下面的例子说明了该函数的应用：在这个程序中，针对函数upcase的参数newstring是否为NULL，采取了不同的处理方式。如果newstring不为NULL，则直接分配内存空间。否则调整原空间的大小以适应新的需要。

1. 使用链表进行动态内存的分配

虽然使用动态内存可以方便地使用内存，但动态内存也有局限性，就是在数据输入到程序之前必须知道数据的大小，以便申请相应的动态内存。然而，在很多情况下，用户都无法事先知道这个值，因而也就无法申请相应的内存空间。对于这种事先未知大小的数据输入，可以使用链表将其分块保存。

链表是一种动态地进行存储分配的结构。链表中的各个元素是一个结构，每个元素称为链表的一个结点。此结构中包含有一个指向此结构的指针，用于指向链表中的下一个结点。链表的最后一个结点的指针NULL，表示链表结束。

下面的例子说明了使用链表获得动态内存的方法：这个程序将终端输入的一系列字符串用链表的形式保存下来。然后再将这些数据组装起来，回显到输出终端。链表的结点为stringdata结构。stringdata结构中的整型量iscontinuing用于表示当前结点是否为链表的末尾。如果iscontinuing有值，则表示此结点不是链表的末尾。

1. **实验结果分析（截屏的实验结果，与实验结果对应的实验分析）**
2. 利用malloc和 calloc函数实现动态内存的分配；利用free函数实现动态内存的释放；  
     
   实验中使用的是同一个地址，因此在创建完地址后想修改地址中的内容时，如果输入的字符太大则会提醒溢出，可见malloc在分配完内存空间后大小就确定了，利用实验二的realloc函数就可以有效地解决这个问题
3. 利用realloc函数实现调整内存空间的大小；  
     
   这里使用的地址是同一个且如果想对地址空间的内容进行重新赋值时，不会重新分配一个空间，而是调整之前已分配的空间，除了系统内存已耗尽的情况，一般不会出现溢出提醒
4. 利用链表实现动态内存分配  
     
   内存释放功能展示：  
   释放前堆栈内的数据展示  
     
   释放后堆栈内的数据展示  
     
   程序使用的链表节点数据结构展示：  
     
   为了实现对输入字符串原样输出到终端上，需要对换行符等特殊字符进行处理，程序通过规定每次读入字符的数量来实现这一功能：如果超过该限定值，则会再创建一个链表节点来存放，遇见换行符则直接结束读入，另外创建一个链表节点来存放后续字符。
5. 查看进程虚存区  
     
   编译程序和模块，运行程序并挂起，通过ps命令查看对应的进程  
     
   C语言char类型数据大小是1字节，而程序分配了1024个char类型大小的地址空间，由上图中也可以看出虚存映射与程序动态分配的内存空间存在一定的关系
6. 编写内核模块，打印某一进程的虚存区  
     
   装入内核模块并查看内核模块是否装入完毕  
     
   出现与maps不相符的问题，本次程序是在虚拟机中模拟的，与真机实验环境不同，通过修改程序代码来找出对应问题。  
   在本次实验中输出到内核模块的代码为：  
   printk("start:%p\tend:%p\n",(unsigned long \*)temp->vm\_start,(unsigned long \*)temp->vm\_end);  
   猜测可能与输出格式相关，尝试将%p修改成%lx直接取长整型十六进制数，代码为：  
   printk("start:%lx\tend:%lx\n",(unsigned long \*)temp->vm\_start,(unsigned long \*)temp->vm\_end);  
   重新装载内核后输出结果为：  
     
   与maps中一致，问题解决
7. 映射一个匿名虚存区，父进程和子进程共享这个匿名区  
     
   创建进程后最好要进行异常处理，本次实验时，因为为判断是否正常创建了进程而耗费了一些不必要的时间。同理，在判断进程是否为父进程时，也最好多加一个pid>0的判断条件，不要贪图便捷，因为pid也可能返回负值。
8. 映射一个名为“test\_data”的文件，文件包含的内容为“Hello,World!”  
   test\_data文件展示：  
     
   运行结果：  
     
   与实验6相同在打开文件后也要进行异常处理，避免因为读取文件错误导致程序失败，实验时最好通过只读方式打开文件，这样可以避免程序污染文件
9. **实验总结**
10. 实验一中字符串大小写转化是对照ASCII字符集来进行的，如`A`的码值为65，`a`的码值为97，因此按照字符大小+-32就可以实现大小写转化。
11. 实验3中实现链表结构时，不需要用int类型的标示来判断字符串是否已经结尾，尾节点的指向下一个节点的指针一定为空，我们只需要根据这个信息就可以判断链表是否已经结束。如果需要要剔除换行符等特殊字符时，才需要额外添加一个int数来识别：如果该节点大小不到程序设定的节点字符串大小的限制值就已经结束，则查看是否有下一个节点，如果有则该节点必定是以特殊字符结束，这时将int数设置成特定值来表示来表示该节点需剔除换行符，如果没有下一个节点，则表明则是最后结尾的换行符无须剔除，并将int数用来表示已经到达链表结尾
12. 匿名虚存区分配空间时容易产生空间溢出的问题，因此从文件中获取数据的话，最好提前知道文件的大小来设置匿名区空间，避免因为内存不足以存放足够多的数据造成程序崩溃退出。