**《操作系统》实验报告**

徽标, 公司名称

描述已自动生成

**题 目 内存管理**

**学 院 计算机学院**

**班 级 2021211304**

**学 号 2021212484**

**姓 名 张梓良**

**2024年 4 月 14日**

目录

[1. 实验4-1-1 观察进程虚拟地址空间内的变量和数据分布 1](#_Toc164082852)

[1.1 实验环境 1](#_Toc164082853)

[1.2 实验内容及目的 1](#_Toc164082854)

[1.3 实验步骤 2](#_Toc164082855)

[1.3.1 实验代码 2](#_Toc164082856)

[1.4 实验结果及分析 4](#_Toc164082857)

[1.4.1 实验结果 4](#_Toc164082858)

[1.4.2 结果分析 4](#_Toc164082859)

[2. 实验4-4-1内存监控检查回收 5](#_Toc164082860)

[2.1 实验环境 5](#_Toc164082861)

[2.2 实验内容及目的 5](#_Toc164082862)

[2.3 实验步骤 5](#_Toc164082863)

[2.3.1 用free命令监测内存使用情况： 5](#_Toc164082864)

[2.3.2 用vmstat命令监视虚拟内存使用情况： 7](#_Toc164082865)

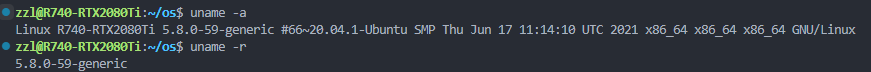
[2.3.3 使用Linux命令回收内存 8](#_Toc164082866)

[3. 总结 9](#_Toc164082867)

# 实验4-1-1 观察进程虚拟地址空间内的变量和数据分布

# 实验环境

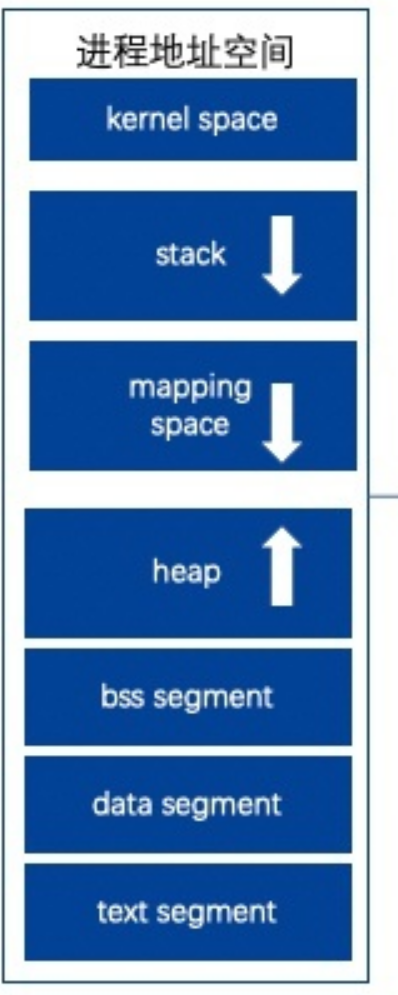
* Linux R740-RTX2080Ti 5.8.0-59-generic #66~20.04.1-Ubuntu SMP Thu Jun 17 11:14:10 UTC 2021 x86\_64 x86\_64 x86\_64 GNU/Linux
* 5.8.0-59-generic



# 实验内容及目的

实验4-1-1 观察进程虚拟地址空间内的变量和数据分布。了解Linux进程虚拟地址空间的组成；编写C语言程序，分配位于堆和栈上的变量，观察变量的地址分布和变化

* 1. Linux进程的虚拟地址空间通常可以分为以下几个部分：
     1. 内核空间：这是操作系统内核和设备驱动程序所使用的空间。在64位架构中，内核空间通常是从0xFFFF\_8000\_0000\_0000到0xFFFF\_FFFF\_FFFF\_FFFF。
     2. 用户空间：这是用户程序所使用的空间。在64位架构中，用户空间通常是从0x0000\_0000\_0000\_0000到0x0000\_7FFF\_FFFF\_FFFF。
     3. 共享库空间：这是共享库文件所使用的空间。在64位架构中，共享库空间通常是从0x0000\_7FFF\_FFFF\_FFFF到0xFFFF\_7FFF\_FFFF\_FFFF。
     4. 堆空间：这是程序运行时使用的动态内存空间。堆空间通常从高地址向低地址扩展，可以使用malloc()和free()等函数来动态分配和释放堆内存。
     5. 栈空间：这是程序运行时使用的函数调用栈空间。栈空间通常从低地址向高地址扩展，可以使用局部变量和函数参数等方式来分配栈内存。
  2. 每一个进程的地址空间如下图所示：



* + 1. 代码段(text)：程序可执行代码，字符串字面值，只读变量，一般为只读区域。代码段指令由操作码和操作对象(或对象地址引用)组成。如果操作对象是立即数(具体数值)，将直接包含在代码中；若是局部数据，将在栈区分配空间，然后引用该数据地址；若位于BSS段和数据段，同样引用该数据地址
    2. 数据段(Data)：程序中已初始化且初值不为0的全局变量和静态局部变量。属于静态内存分配(静态存储区)，可读可写
    3. BSS(Block Started by Symbol)段：未初始化的全局变量和静态局部变量。初始值为0的全局变量和静态局部变量(依赖于编译器实现)。未定义且初值不为0的符号(该初值即common block的大小)。程序运行时，数据段和BSS段的整个区段通常称为数据区，数据段与BSS段的区别如下：
       1. BSS段不占用物理文件尺寸，但占用内存空间；数据段占用物理文件，也占用内存空间。E.g. 对于大型数组如int ar0[10000] = {1, 2, 3, ...}和int ar1[10000]，ar1放在BSS段，只记录共有10000\*4个字节需要初始化为0，而不是像ar0那样记录每个数据1、2、3...，此时BSS为目标文件所节省的磁盘空间相当可观
       2. 当程序读取不在内存的数据段数据时，系统产生page fault，为数据段数据分配相应的物理内存；当程序读取BSS段的数据时，内核会将其转到一个全零页面，不会发生page fault ，也不会为其分配相应的物理内存。
    4. 栈stack：存储局部变量、函数参数、返回地址等，用于为函数内部声明的非静态局部变量(C语言中称“自动变量”)提供存储空间。以及记录函数调用过程相关的维护性信息，称为栈帧(Stack Frame)或过程活动记录(Procedure Activation Record)，包括：函数返回地址，不适合装入寄存器的函数参数及一些寄存器值的保存。除递归调用外，堆栈并非必需。因为编译时可获知局部变量，参数和返回地址 所需空间，并将其分配于BSS段。临时存储区，用于暂存长算术表达式部分计算结果或alloca()函数分配的栈内内存。每个进程/线程都有属于自己的栈，栈的内容随着线程执行动态变化（增长，收缩）。栈有最大值限制RLIMIT\_STACK(通常是8M)，使用Linux命令ulimit -s可查看和设置堆栈最大值
    5. 堆heap：存放进程运行时动态分配的内存段，由程序自己控制，可动态扩张或缩减
       1. 调用malloc(C)/new(C++)等分配内存时，新分配的内存动态添加到堆上(扩张)
       2. 调用free(C)/delete(C++)等释放内存时，被释放的内存从堆中剔除(缩减)

堆管理器通过链表管理程序申请的内存，由于堆申请和释放是无序的，最终会产生内存碎片。堆内存一般由应用程序分配释放，回收的内存可供重新使用。若程序不释放，程序结束时操作系统可能会自动回收。堆中内容是匿名的，无法按名字直接访问，只能通过指针间接访问。堆的末端由break指针标识，当堆管理器需要更多内存时，可通过系统调用brk()和sbrk()来移动break指针以扩张堆，一般由系统自动调用。堆使用时经常出现两种问题：1. 内存破坏：释放或改写仍在使用的内存。2. 内存泄漏：未释放不再使用的内存。释放次数少于申请次数，可能造成内存泄漏。

* + 1. 内存映射段(mmap)：将硬盘文件的内容直接映射到内存—内存映射文件。应用程序使用Windows CreateFileMapping()/MapViewOfFile()或Linux的mmap()系统调用建立这种映射。内存映射是一种方便高效的文件I/O方式， 被用于装载动态共享库。内存映射段用于映射可执行文件用到的动态链接库。用户也可创建匿名内存映射，该映射没有对应的文件, 用于存放程序数据。在 Linux中，通过malloc()请求一大块内存，C运行库将创建一个匿名内存映射，而不使用堆内存。“大块” 意味着比阈值 MMAP\_THRESHOLD还大，缺省为128KB，可通过mallopt()调整。

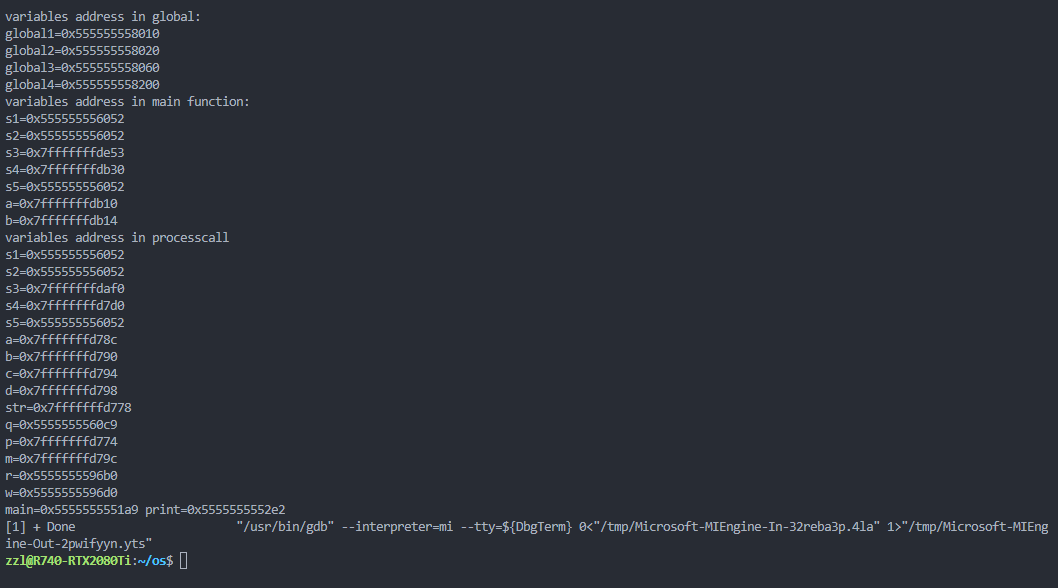
# 实验步骤

## 实验代码

1. #include <stdio.h>
2. #include <malloc.h>
3. void print(char \*, int);
4. int global1[3] = {1, 2, 3};
5. int global2[3] = {4, 5, 6};
6. int global3[100];
7. int global4[100];
8. int main()
9. {
10. printf("variables address in global:\nglobal1=%p\nglobal2=%p\nglobal3=%p\nglobal4=%p\n",
11. global1, global2, global3, global4);
12. *//"abcde"作为字符串常量存储在常量区 s1、s2、s5拥有相同的地址*
13. char \*s1 = "abcde";
14. char \*s2 = "abcde";
15. char s3[] = "abcd";
16. long int \*s4[100];
17. char \*s5 = "abcde";
18. int a = 5;
19. int b = 6; *// a,b在栈上，&a>&b地址反向增长*
20. printf("variables address in main function:\ns1=%p\ns2=%p\ns3=%p\ns4=%p\ns5=%p\na=%p\nb=%p\n",
21. s1, s2, s3, s4, s5, &a, &b);
22. printf("variables address in processcall\n");
23. print("ddddddddd", 5);
24. *// 参数入栈从右至左进行,p先进栈,str后进 &p>&str*
25. printf("main=%p print=%p \n", main, print);
26. *// 打印代码段中主函数和子函数的地址，编译时先编译的地址低，后编译的地址高main < print*
27. }
28. void print(char \*str, int p)
29. {
30. char \*s1 = "abcde"; *// abcde在常量区，s1在栈上*
31. char \*s2 = "abcde"; *// abcde在常量区，s2在栈上，s2 - s1可能等于0，编译器优化了相同的常量，只在内存保存一份*
32. *// 而&s1>&s2*
33. char s3[] = "abcdeee"; *// abcdeee在常量区，s3在栈上，数组保存的内容为abcdeee的一份拷贝*
34. long int \*s4[100];
35. char \*s5 = "abcde";
36. int a = 5;
37. int b = 6;
38. int c;
39. int d; *// a,b,c,d均在栈上，&a>&b>&c>&d地址反向增长*
40. char \*q = str;
41. int m = p;
42. char \*r = (char \*)malloc(1);
43. char \*w = (char \*)malloc(1); *// r<w 堆正向增长*
44. printf("s1=%p\ns2=%p\ns3=%p\ns4=%p\ns5=%p\na=%p\nb=%p\nc=%p\nd=%p\nstr=%p\nq=%p\np=%p\nm=%p\nr=%p\nw=%p\n",
45. s1, s2, s3, s4, s5, &a, &b, &c, &d, &str, q, &p, &m, r, w);
46. */\* 栈和堆是在程序运行时候动态分配的，局部变量均在栈上分配。*
47. 栈是反向增长的，地址递减；malloc等分配的内存空间在堆空间。堆是正向增长的，地址递增。
48. r,w变量在栈上(则&r>&w)，r,w所指内容在堆中(即r<w)。\*/
49. }

# 实验结果及分析

## 实验结果



## 结果分析

**variables address in global:**

* global1,global2是指向已经初始化的全局数组的指针，全局数组位于**Data segment**。
* global3,global4是指向未初始化的全局数组的指针，全局数组位于**Bss segment**。

**variables address in main function:**

* s1、s2、s5是指向常量“abcde”的指针（所以它们的值相同），s1、s2、s5位于**Stack**，常量“abcde”位于**Text segment。**
* s3是指向常量“abcdeee”副本的指针，s3位于**Stack，**s3指向的常量副本也在**Stack。**
* s4是指向一个指针数组，s4位于**Stack，**s4指向的指针数组也在Stack**。**
* a,b为变量，它们都位于**Stack，**同时&a>&b地址反向增长。

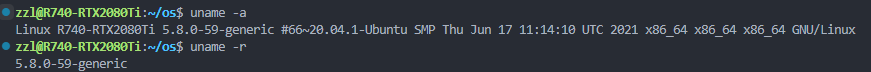
**variables address in processcall：**

* s1、s2、s5是指向常量“abcde”的指针（所以它们的值相同），s1、s2、s5位于**Stack**，常量"abcde"位于**Text segment。**
* s3是指向常量“abcdeee”副本的指针，s3位于**Stack，**s3指向的常量副本也在**Stack。**
* s4是指向一个指针数组，s4位于**Stack，**s4指向的指针数组也在Stack**。**
* a,b,c,d为变量，它们都位于**Stack，**同时&a>&b>&c>&d地址反向增长。
* str和p为函数的参数，位于**Stack，**参数是从右往左入栈，p先入栈，str后入栈，&str<&p。
* q是指向常量“ddddddddd”的指针，q位于**Stack，**常量“ddddddddd”位于**Text segment。**
* m为变量，m位于**Stack。**
* r和w是指向动态分配的内存的指针，r和w位于**Stack**，它们指向的内容位于**Heap**，同时堆是正向增长的，所以r<w。

# 实验4-4-1内存监控检查回收

# 实验环境

* Linux R740-RTX2080Ti 5.8.0-59-generic #66~20.04.1-Ubuntu SMP Thu Jun 17 11:14:10 UTC 2021 x86\_64 x86\_64 x86\_64 GNU/Linux
* 5.8.0-59-generic



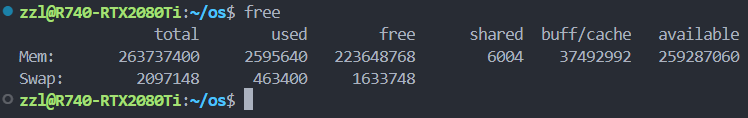
# 实验内容及目的

参照“资料1-自定义实验指导**/附件1-1-2计算机操作系统实验指导**”中12.2、12.3.1节内容，完成本实验。

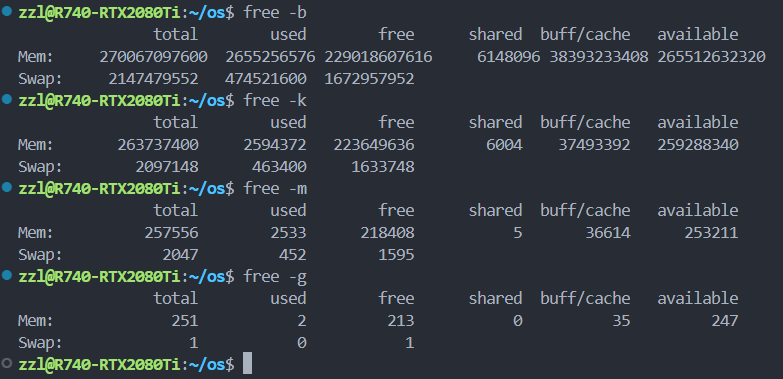
1. 掌握查看实时监控内存、内存回收的方法。
2. 进一步掌握虚拟存储器的实现方法。
3. 掌握各种页面置换算法。
4. 比较各种页面置换算法的优缺点。

# 实验步骤

### 用free命令监测内存使用情况：



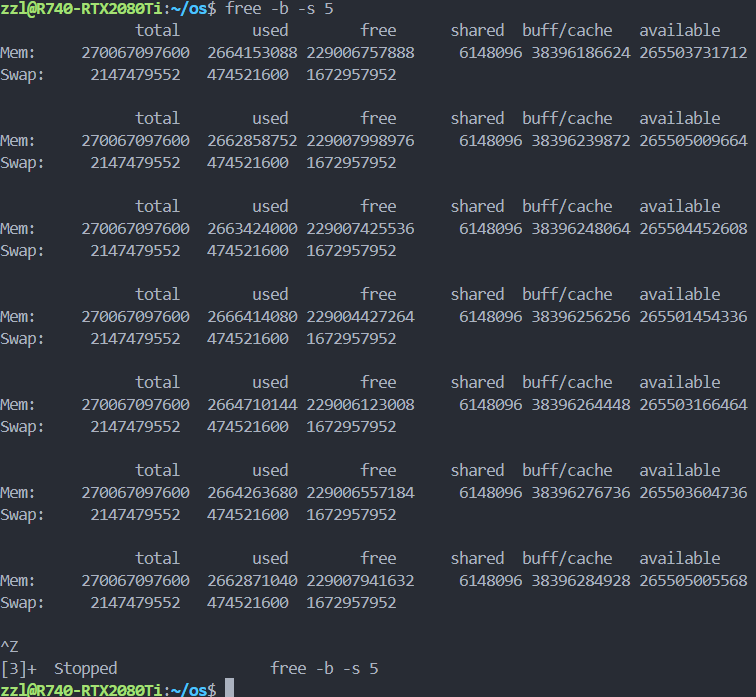
free单独使用默认后面为-k，即显示内存情况以KB为单位，当然也可通过更改此参数显示不同单位下的内存情况：



以KB为单位的显示结果分析其表示的含义:

* total 是总内存数；
* used 是已经使用的内存数；
* free 是**完全未被使用的内存量**，即这部分内存中没有任何数据。这部分内存**可以立即被任何需要额外内存的进程使用**。；
* shared 是多个进程共享的内存总数；
* buff/cache 是缓冲/缓存内存数;
* available 是**指系统中当前可用于启动新应用程序而不必交换出任何内存的估计内存量**。这个数字**考虑了缓存、缓冲区和其它因素**。（因此一般**available>free**）
* 即当前系统的物理内存为263737400KB，已使用了2594372KB，空闲223649636KB；SWAP分区总大小为2097148KB，当前已使用了463400KB，空闲1633748KB。

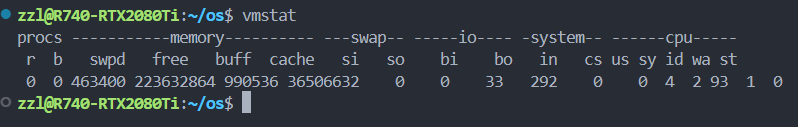
我们可以添加参数-s <second>的方式定时监控物理内存的使用状况：



其中shared作为共享内存数，基本不变，除了used和free的变化，关键在于buff/cache的变化。分析如下：

* buffers：作为buffer cache的内存，是用来**缓冲块设备**的，对块的操作会使用buffer cache进行缓存，它只记录文件系统的元数据（metadata）以及 tracking in-flight pages。比如我们在格式化文件系统的时候，查找系统文件等。
* cached：作为page cache的内存，是用来**给文件做缓冲**。如果 cache 的值很大，说明cache的文件数很多。如果频繁访问到的文件都能被cache住，那么磁盘的I/O时间会大大缩短。
* buffers主要用来存放**目录里面有什么内容\文件的属性\权限等等**。cached直接用来记忆我们**打开过的文件和程序**。

### 用vmstat命令监视虚拟内存使用情况：



1. **procs**：
   * **r**：等待运行的进程数量。
   * **b**：处于不可中断睡眠状态的进程数量。
2. **memory**：
   * **swpd**：使用的交换空间大小（KB）。
   * **free**：空闲的物理内存大小（KB）。
   * **buff**：用作缓冲的内存大小（KB）。
   * **cache**：用作缓存的内存大小（KB）。
3. **swap**：
   * **si**：从磁盘交换到内存的速度（KB/s）。
   * **so**：从内存交换到磁盘的速度（KB/s）。
4. **io**：
   * **bi**：从块设备读入的数据量（块/s）。
   * **bo**：写入块设备的数据量（块/s）。
5. **system**：
   * **in**：每秒的中断次数，包括时钟中断。
   * **cs**：每秒的上下文切换次数。
6. **cpu**（以百分比表示）：
   * **us**：用户态的CPU时间百分比。
   * **sy**：系统态的CPU时间百分比。
   * **id**：空闲的CPU时间百分比。
   * **wa**：等待I/O的CPU时间百分比。
   * **st**：被虚拟机偷走的时间百分比。

根据**vmstat**输出可知：

* 当前没有进程等待运行，也没有处于不可中断睡眠状态的进程。
* 系统使用了约 453 MB 的交换空间。
* 系统有大量的空闲内存，约 223.6 GB。
* CPU主要处于空闲状态，有 93% 的时间处于空闲。
* 系统I/O活动较低，几乎没有交换活动，表明系统当前负载不重。

当系统的物理内存不足以满足进程的需求时，Linux会使用虚拟内存来扩展可用的内存空间。虚拟内存是通过将部分数据存储在磁盘上的交换空间（swap）中实现的。

当系统的物理内存不足时，操作系统会将部分不经常使用的内存页面（页）移出物理内存，并将其存储在交换空间中。这样，系统可以释放物理内存用于其他需要更紧急的进程。当进程再次访问被移出的内存页面时，它会从交换空间中加载回物理内存。

Linux系统中，可以使用swap分区或者swap文件来作为交换空间。swap分区是在磁盘上划分的专用分区，而swap文件则是一个普通的文件。

当系统的物理内存不足时，**首先会尝试使用虚拟内存（交换空间）来满足进程的内存需求。**如果虚拟内存也不足以满足需求，**系统将开始使用swap分区来提供额外的内存支持**。

使用交换空间（swap）会导致系统性能下降，因为从磁盘中读取和写入数据的速度比从物理内存中读取和写入数据的速度慢得多。因此，最好确保系统具有足够的物理内存来满足运行中的进程的需求，以避免过多地依赖交换空间。

当物理内存和交换空间（swap）都用尽时，系统会出现内存耗尽（out of memory）的情况。在这种情况下，系统将无法为新的进程分配内存。

交换空间的使用通常是**由操作系统自动管理的，而不需要手动清空**。

### 使用Linux命令回收内存

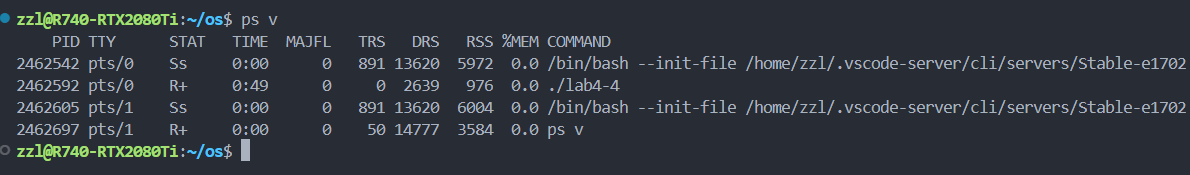
编写以下程序用于测试：

1. #include <stdio.h>
2. int main()
3. {
4. long long a = 0;
5. while (1)
6. {
7. a++;
8. }
9. }

编译程序并运行：



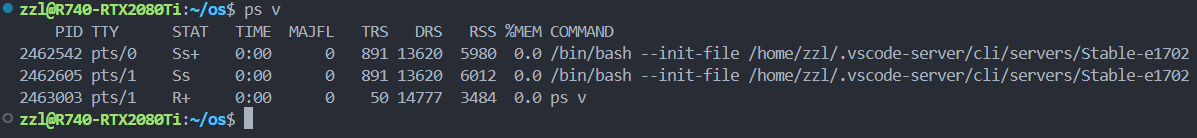
使用ps v命令查看进程占用资源情况：



使用kill -9 2462592终止死循环程序：



再次使用ps v命令查看是否终止成功：



可以发现已经成功终止死循环程序lab4-4。

# 总结

在本次操作系统课程的实验中，我深入研究了Linux操作系统中内存管理的机制，特别关注了动态内存分配、虚拟内存利用和资源回收策略。实验中，我通过编程和系统命令实践了这些概念，并观察了不同配置下的系统表现。

**动态内存分配：**

我们首先探索了malloc函数的工作原理，了解了它如何通过brk和mmap系统调用来请求操作系统分配虚拟内存。这一过程涉及到操作系统内核的堆管理策略，即如何在堆区寻找足够的连续空间以满足内存请求。这个理解帮助我更好地把握应用程序在运行时内存的动态变化。

**虚拟内存的映射与管理：**

虚拟内存的概念核心在于它提供了一种机制，允许操作系统将一个大的、连续的地址空间呈现给应用程序，同时背后的物理内存可能是非连续的、更小的。通过实验，我观察到了虚拟地址到物理地址的映射过程，以及操作系统如何在物理内存不足时使用交换空间（swap space）。

**内存监控与优化：**

通过使用工具如free和vmstat，我能够实时监控内存使用情况和系统性能。这些工具显示了内存的分配和回收如何随着系统运行状态变化，以及如何调整系统配置以优化性能。