

附件（四）

深圳大学实验报告

课程名称： 操作系统

实验项目名称： 内存管理实验

学院： 数学科学学院

专业： 信息与计算科学（数学与计算机实验班）

指导教师： 张 滇

报告人： 王曦 学号： 2021192010 班级： 数计

实验时间： 2024 年 05 月 09 日

实验报告提交时间： 2024 年 05 月 09 日

教务处制

一、实验目的与要求

- (1) 加深对内存分配与使用操作的直观认识。
- (2) 掌握 Linux 操作系统的内存分配与使用的编程接口。
- (3) 了解 Linux 操作系统中进程的逻辑编程地址和物理地址间的映射。

实验内容:

- (1) 可以使用 Linux 或其它 Unix 类操作系统。
- (2) 学习该操作系统提供的分配、释放的函数使用方法。
- (3) 学习该操作系统提供的进程地址映射情况的工具。

实验环境:

- (1) 硬件: 桌面 PC。
- (2) 软件: Linux 或其他操作系统。

二、方法、步骤

(说明程序相关的算法原理或知识内容, 程序设计的思路和方法, 可以用流程图表述, 程序主要数据结构的设计、主要函数之间的调用关系等)

操作部分 (参考): (90 分)

- 1) 借助 google 工具查找资料, 学习使用 Linux 进程的内存分配、释放函数;
- 2) 借助 google 工具查找资料, 学习 Linux proc 文件系统中关于内存映射的部分内容 (了解/proc/pid/目录下的 maps、status、smap 等几个文件内部信息的解读);
- 3) 编写程序, 连续申请分配六个 128MB 空间 (记为 1~6 号), 然后释放第 2、3、5 号的 128MB 空间。然后再分配 1024MB, 记录该进程的虚存空间变化 (/proc/pid/maps), 每次操作前后检查/proc/pid/status 文件中关于内存的情况, 简要说明虚拟内存变化情况。推测此时再分配 64M 内存将出现在什么位置, 实测后是否和你的预测一致? 解释说明用户进程空间分配属于课本中的离散还是连续分配算法? 首次适应还是最佳适应算法? 用户空间存在碎片问题吗?
- 4) 设计一个程序测试出你的系统单个进程所能分配到的最大虚拟内存空间为多大。
- 5) 编写一个程序, 分配 256MB 内存空间 (或其他足够大的空间), 检查分配前后 /proc/pid/status 文件中关于虚拟内存和物理内存的使用情况, 然后每隔 4KB 间隔将相应地址进行读操作, 再次检查/proc/pid/status 文件中关于内存的情况, 对比前后两次内存情况, 说明所分配物理内存 (物理内存块) 的变化。然后重复上面操作, 不过此时为写操作, 再观察其变化。

提高部分: (10 分)

- 1) 编写并运行 (在第 5 步的程序未退出前) 另一进程, 分配等于或大于物理内存的空间, 然后每隔 4KB 间隔将相应地址的字节数值增 1, 此时再查看前一个程序的物理内存变化, 观察两个进程竞争物理内存的现象。
- 2) 分配足够大的内存空间, 其容量超过系统现有的空闲物理内存的大小, 1) 按 4KB 的间隔逐个单元进行写操作, 重复访问数遍 (使得程序运行时间可测量); 2) 与前面访问总量和次数不便, 但是将访问分成 16 个连续页为一组, 逐组完成访问, 记录运行时间。观察系统的状态, 比较两者运行时间, 给出判断和解释。

三. 实验过程及内容:

(对程序代码进行说明和分析, 越详细越好, 代码排版要整齐, 可读性要高)

1. Linux 进程的内存分配、释放函数

1.1 内核虚拟内存布局

```
Memory: 1024MB = 1024MB total    // 内存的大小是 1GB
Memory: 810820k/810820k available, 237756k reserved, 272384K highmem
Virtual kernel memory layout:
vector : 0xffff0000 - 0xffff1000 ( 4 kB)    // 存放中断向量表
fixmap : 0xffff0000 - 0xfffe0000 ( 896 kB)    // 固定映射区, 留作特定用途
vmalloc : 0xef800000 - 0xee000000 ( 246 MB)    // vmalloc()分配内存的区
lowmem : 0xc0000000 - 0xef600000 ( 758 MB)    // 低端内存区, 属于 GFP_KERNEL 标志
pkmap : 0xbfe00000 - 0xc0000000 ( 2 MB)    // 永久映射, 高端内存页框到内核地址空间的长期映射
modules : 0xbf000000 - 0xbfe00000 ( 14 MB)    // #insmod *.ko module 存放段
.text : 0xc0008000 - 0xc0a51018 (10533 kB)    // 代码段和只读的数据段
.init : 0xc0a52000 - 0xc0a8f100 ( 245 kB)    // 初始化段, 使用 __init 修饰初始化函数
.data : 0xc0a90000 - 0xc0b297d8 ( 614 kB)    // 可读写的的数据段
.bss : 0xc0b297fc - 0xc0d09488 (1920 kB)    // 未初始化的数据段
SLUB: Genslabs=11, HWalign=64, Order=0-3, MinObjects=0, CPUs=8, Nodes=1
```

由上知:

- (1) 使用 `kmalloc()`函数和 `kzalloc()`函数会在 `lowmem: 0xc0000000 ~ 0xef600000` 的地址空间中申请内存。
- (2) 使用 `vmalloc()`函数会在 `vmalloc: 0xef800000 ~ 0xee000000` 的地址空间中申请内存。

1.2 `kmalloc()`函数与 `kfree()`函数

原型	<code>void *kmalloc(size_t size, gfp_t flags);</code>
功能	申请一块位于物理内存映射区域的内存。
参数	<p>(1) <code>size</code>: 申请的内存大小 (单位: Byte)</p> <p>(2) <code>flgs</code>: 内存分配方法。</p> <p>① <code>GFP_ATOMIC</code>: 分配内存的过程是一个原子过程, 分配内存的过程不会被 (高优先级进程或中断) 打断。</p> <p>② <code>GFP_KERNEL</code>: 正常分配内存。</p> <p>③ <code>GFP_DMA</code>: 给 DMA 控制器分配内存, 需要使用该标志 (DMA 要求分配虚拟地址和物理地址连续)。</p>
备注	申请的内存存在物理上连续, 与真实的物理地址只有一个固定的偏移。因为转换简单, 故对申请大小有限制, 不超过 128 kB。

其对应的内存释放函数:

原型	<code>void kfree(const void *objp);</code>
----	--

1.3 kzalloc()函数

原型	void *kzalloc(size_t size, gfp_t flags);
功能	申请一块位于物理内存映射区域的内存，并附加__GFP_ZERO 标志，即将申请到的内存内容清零。
参数	(1) size: 申请的内存大小（单位：Byte） (2) flgs: 内存分配方法。 ① GFP_ATOMIC: 分配内存的过程是一个原子过程，分配内存的过程不会被（高优先级进程或中断）打断。 ② GFP_KERNEL: 正常分配内存。 ③ GFP_DMA: 给 DMA 控制器分配内存，需要使用该标志（DMA 要求分配虚拟地址和物理地址连续）。
备注	申请的内存物理上连续，与真实的物理地址只有一个固定的偏移。因为转换简单，故对申请大小有限制，不超过 128 kB。

其对应的内存释放函数也是 kfree()函数。

1.4 vmalloc()函数与 vfree()函数

原型	void *vmalloc(unsigned long size);
功能	在虚拟内存申请一块连续的内存区，这块内存区在物理内存中未必连续。
备注	vmalloc()不保证申请的内存物理上连续，故可申请的内存大小较大。

其对应的内存释放函数是 vfree()函数。

备注：

- (1) vmalloc() 和 vfree() 可以睡眠，因此不能从中断上下文调用。
- (2) vmalloc()函数可申请的内存大小由/proc/meminfo 文件中的 VmallocChunk 的剩余空间决定。

1.5 对比

1.5.1 kmalloc()、kzalloc()、vmalloc()的共同点

- (1) 用于申请内核空间的内存。
- (2) 内存以字节为单位分配。
- (3) 分配的虚拟内存地址连续。

1.5.2 kmalloc()、kzalloc()、vmalloc()的共同点

- (1) kzalloc()函数本质是加了清零的 kmalloc()函数。下面不再区分 kzalloc()函数和 kmalloc()函数。
- (2) kmalloc()函数申请的内存大小限制 128 kB，而 vmalloc()函数申请的内存大小由剩余空间决定。
- (3) kmalloc()函数保证申请的内存的物理地址连续，但 vmalloc()函数不保证。
- (4) kmalloc()函数的申请过程可以是原子过程，但 vmalloc()函数的申请过程可能被阻塞。
- (5) kmalloc()函数分配内存开销小，比 vmalloc()函数快。

1.5.3 备注

- (1) 一般只有内存被 DMA 访问时才需物理上连续。
- (2) 为保证性能，内核中一般用 kmalloc()函数，只有需要大块内存时采用 vmalloc()函数。

2. Linux 的内存映射

Linux 的 `/proc` 文件系统提供了轻量级的与内核通信的方式。`/proc/pid/`目录下包含关于特定进程的信息，其中包括内存映射。

2.1 maps 文件

`/proc/pid/maps` 文件列出了进程当前的内存映射情况，每一行代表了进程中的一个内存区域，包括了该内存区域的起始地址、结束地址、权限等信息。通常的格式如下：

address	perms	offset	dev	inode	pathname
00400000-0040c000	r-xp	00000000	fd:01	918579	/usr/bin/cat

其中：

参数	含义
address	内存区域的起始地址和结束地址。
perms	内存区域的权限，包括读取（r）、写入（w）、执行（x）等。
offset	内存区域相对于文件起始位置的偏移量。
dev	文件所在设备的编号。
inode	文件的索引节点号。
pathname	映射到该内存区域的文件路径，如果是匿名映射则显示为[heap]、[stack]等。

2.2 status 文件

`/proc/pid/status` 文件包含了关于进程状态的各种信息，其中也包括了一些关于内存映射的统计数据，例如：

VmPeak:	23456 kB
VmSize:	12345 kB
VmLck:	12 kB
VmPin:	34 kB
VmHWM:	5678 kB
VmRSS:	910 kB
VmData:	3456 kB
VmStk:	78 kB
VmExe:	910 kB
VmLib:	2345 kB
VmPTE:	56 kB

其中：

参数	含义
VmPeak	进程所使用的最大虚拟内存空间。
VmSize	进程当前的虚拟内存空间。
VmLck	已锁定在内存中的虚拟内存大小。
VmPin	已固定在内存中的虚拟内存大小。
VmHWM	进程使用的最大物理内存大小（高水位标记）。
VmRSS	当前进程占用的物理内存大小（常驻集大小）。
VmData	进程数据段的大小。

VmStk	进程堆栈的大小。
VmExe	进程代码段的大小。
VmLib	共享库的大小。
VmPTE	页表项的大小。

2.3 smaps 文件

/proc/pid/smaps 文件提供了更详细的关于进程内存映射的信息，包括了每个内存区域的更详细的统计数据，例如：

00400000-0040c000 r-xp 00000000 fd:01 918579 /usr/bin/cat	
Size:	48 kB
Rss:	16 kB
Pss:	8 kB
Shared_Clean:	16 kB
Shared_Dirty:	0 kB
Private_Clean:	0 kB
Private_Dirty:	0 kB
Referenced:	16 kB
Anonymous:	0 kB
AnonHugePages:	0 kB
Swap:	0 kB
KernelPageSize:	4 kB
MMUPageSize:	4 kB
Locked:	0 kB

其中：

参数	含义
Size	内存区域的大小。
Rss	驻留集大小，即当前被进程使用的物理内存大小。
Pss	按比例分享的物理内存大小，与共享库相关。
Shared_Clean	共享的干净页面大小。
Shared_Dirty	共享的脏页面大小。
Private_Clean	私有的干净页面大小。
Private_Dirty	私有的脏页面大小。
Referenced	被引用的页面大小。
Anonymous	匿名页面大小。
AnonHugePages	匿名巨大页面大小。
Swap	被换出到交换空间的页面大小。
KernelPageSize	内核页大小。
MMUPageSize	硬件页大小。
Locked	已锁定的大小。

3. 内存分配算法实验

编写程序，连续申请分配六个 128MB 空间（记为 1~6 号），然后释放第 2、3、5 号的 128MB 空间。然后再分配 1024MB，记录该进程的虚存空间变化（/proc/pid/maps），每

次操作前后检查/proc/pid/status 文件中关于内存的情况，简要说明虚拟内存变化情况。推测此时再分配 64M 内存将出现在什么位置，实测后是否和你的预测一致？解释说明用户进程空间分配属于课本中的离散还是连续分配算法？首次适应还是最佳适应算法？用户空间存在碎片问题吗？

3.1 程序

将下面的代码保存到 memoryAllocation.cpp 中。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

using uint = unsigned int;
const uint MB = 1024 * 1024;
const uint Sigma = 128;
const uint MAXN = 15;

char* myMalloc(uint memSize) {
    char* buffer = (char*)malloc(memSize);

    // 写
    for (uint i = 0; i < memSize; i++) {
        buffer[i] = i % Sigma;
    }

    printf("Allocated %d MB of Mememory: %p - %p\n", memSize / MB, buffer, buffer +
memSize - 1);

    return buffer;
}

int main() {
    getchar(); // 阻塞

    // 连续申请 6 个 128 MB 的空间
    char* buffers[MAXN];
    for (uint i = 1; i <= 6; i++) {
        buffers[i] = myMalloc(128 * MB);
    }

    getchar(); // 阻塞

    // 释放第 2、3、5 号
    free(buffers[2]);
    free(buffers[3]);
    free(buffers[5]);
```

```

    getchar(); // 阻塞

    // 申请 1024 MB 的空间
    buffers[7] = myMalloc(1024 * MB);

    getchar(); // 阻塞

    // 申请 64 MB 的空间
    buffers[8] = myMalloc(64 * MB);

    getchar(); // 阻塞

    return 0;
}

```

上述程序在申请空间的操作前后都用 `getchar()`阻塞，以便观察内存变化。

上述程序除申请空间外，还写申请的内存，防止只申请不用的内存被 OS 优化，不能体现真实情况。

3.2 运行前后内存情况

用如下命令编译并运行 `memoryAllocation.cpp`。

```

g++ memoryAllocation.cpp -o memoryAllocation
./memoryAllocation

```

用如下命令查询 `memoryAllocation` 的 PID。

```
ps aux | grep memoryAllocation
```

运行结果如下图所示，其 PID 为 305591。

```

hytidel@hytidel-virtual-machine:~$ ps aux | grep memoryAllocation
hytidel 305591 0.0 0.0 2776 1152 pts/4 S+ 07:58 0:00 ./memoryAllocation
hytidel 305849 0.0 0.0 12304 2816 pts/6 S+ 07:59 0:00 grep --color=auto memoryAllocation
hytidel@hytidel-virtual-machine:~$

```

3.2.1 初始内存

用如下命令查询该进程的初始内存情况。

```

cat /proc/305591/maps
cat /proc/305591/status

```

运行结果如下图所示，主要关注初始的 heap 区大小和虚存大小 `VmSize`。


```

hytidel@hytidel-virtual-machine:~$ cat /proc/305591/maps
5aa9c3e58000-5aa9c3e59000 r--p 00000000 08:03 924817 /home/hytidel/05exp/exp3/memoryAllocation
5aa9c3e59000-5aa9c3e5a000 r-xp 00001000 08:03 924817 /home/hytidel/05exp/exp3/memoryAllocation
5aa9c3e5a000-5aa9c3e5b000 r--p 00002000 08:03 924817 /home/hytidel/05exp/exp3/memoryAllocation
5aa9c3e5b000-5aa9c3e5c000 r--p 00003000 08:03 924817 /home/hytidel/05exp/exp3/memoryAllocation
5aa9c3e5c000-5aa9c3e5d000 rw-p 00004000 08:03 924817 /home/hytidel/05exp/exp3/memoryAllocation
5aa9c3e5d000-5aa9c3e7e000 rw-p 00000000 00:00 0 [heap]
7c6f02200000-7c6f02228000 r--p 00000000 08:03 526707 /usr/lib/x86_64-linux-gnu/libc.so.6
7c6f02228000-7c6f0223bd000 r-xp 00028000 08:03 526707 /usr/lib/x86_64-linux-gnu/libc.so.6
7c6f0223bd000-7c6f022415000 r--p 001bd000 08:03 526707 /usr/lib/x86_64-linux-gnu/libc.so.6
7c6f022415000-7c6f022416000 ---p 00215000 08:03 526707 /usr/lib/x86_64-linux-gnu/libc.so.6
7c6f022416000-7c6f02241a000 r--p 00215000 08:03 526707 /usr/lib/x86_64-linux-gnu/libc.so.6
7c6f02241a000-7c6f02241c000 rw-p 00219000 08:03 526707 /usr/lib/x86_64-linux-gnu/libc.so.6
7c6f02241c000-7c6f022429000 rw-p 00000000 00:00 0
7c6f02253a000-7c6f02253d000 rw-p 00000000 00:00 0
7c6f02254c000-7c6f02254e000 rw-p 00000000 00:00 0
7c6f02254e000-7c6f022550000 r--p 00000000 08:03 526701 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7c6f022550000-7c6f02257a000 r-xp 00002000 08:03 526701 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7c6f02257a000-7c6f022585000 r--p 0002c000 08:03 526701 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7c6f022586000-7c6f022588000 r--p 00037000 08:03 526701 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7c6f022588000-7c6f02258a000 rw-p 00039000 08:03 526701 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffda1b30000-7ffda1b51000 rw-p 00000000 00:00 0 [stack]
7ffda1b8f000-7ffda1b93000 r--p 00000000 00:00 0 [vvar]
7ffda1b93000-7ffda1b95000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 --xp 00000000 00:00 0 [vsyscall]
hytidel@hytidel-virtual-machine:~$

```

```

hytidel@hytidel-virtual-machine:~$ cat /proc/305591/status
Name: memoryAllocatio
Umask: 0002
State: S (sleeping)
Tgid: 305591
Ngid: 0
Pid: 305591
PPid: 218183
TracerPid: 0
Uid: 1000 1000 1000 1000
Gid: 1000 1000 1000 1000
FDSize: 256
Groups: 4 24 27 30 46 122 135 136 1000
NSTgid: 305591
NSpid: 305591
NSpgid: 305591
NSSid: 218183
Kthread: 0
VmPeak: 2776 kB
VmSize: 2776 kB
VmLck: 0 kB
VmPin: 0 kB
VmHWM: 1152 kB
VmRSS: 1152 kB

```

3.2.2 连续申请 6 个 128 MB 的内存

在运行着 memoryAllocation 的终端按回车，程序连续申请 6 个 128 MB 的内存。

运行结果如下图所示，观察到申请的内存从高地址到低地址分配，且几乎连续，其中相邻两块内存间都相差 0x1000。

```

hytidel@hytidel-virtual-machine:~/05exp/exp3$ ./memoryAllocation
Allocated 128 MB of Mememory: 0x7c6efa1ff010 - 0x7c6f021ff00f
Allocated 128 MB of Mememory: 0x7c6ef21fe010 - 0x7c6efa1fe00f
Allocated 128 MB of Mememory: 0x7c6eea1fd010 - 0x7c6et21fd00f
Allocated 128 MB of Mememory: 0x7c6ee21fc010 - 0x7c6eea1fc00f
Allocated 128 MB of Mememory: 0x7c6eda1fb010 - 0x7c6ee21fb00f
Allocated 128 MB of Mememory: 0x7c6ed21fa010 - 0x7c6eda1fa00f

```

相差 0x1000 的原因：虚存的地址空间按页对齐，页大小为 4 kB，即 4096 Bytes，亦即 0x1000 Bytes。

用如下命令查询该进程的此时的内存情况。

```
cat /proc/305591/maps
cat /proc/305591/status
```

运行结果如下图所示。

(1) heap 区下方多了一块内存 0x7c6ed21fa000 – 0x7c6f02200000 。

```
hytidel@hytidel-virtual-machine:~$ cat /proc/305591/maps
5aa9c3e58000-5aa9c3e59000 r--p 00000000 08:03 924817 /home/hytidel/0Sexp/exp3/memoryAllocation
5aa9c3e59000-5aa9c3e5a000 r-xp 00001000 08:03 924817 /home/hytidel/0Sexp/exp3/memoryAllocation
5aa9c3e5a000-5aa9c3e5b000 r--p 00002000 08:03 924817 /home/hytidel/0Sexp/exp3/memoryAllocation
5aa9c3e5b000-5aa9c3e5c000 r--p 00002000 08:03 924817 /home/hytidel/0Sexp/exp3/memoryAllocation
5aa9c3e5c000-5aa9c3e5d000 rw-p 00003000 08:03 924817 /home/hytidel/0Sexp/exp3/memoryAllocation
5aa9c3e5d000-5aa9c3e7e000 rw-p 00000000 00:00 0 [heap]
7c6ed21fa000-7c6f02200000 rw-p 00000000 00:00 0
7c6f02200000-7c6f02228000 r--p 00000000 08:03 526707 /usr/lib/x86_64-linux-gnu/libc.so.6
7c6f02228000-7c6f023bd000 r-xp 00028000 08:03 526707 /usr/lib/x86_64-linux-gnu/libc.so.6
7c6f023bd000-7c6f02415000 r--p 001bd000 08:03 526707 /usr/lib/x86_64-linux-gnu/libc.so.6
7c6f02415000-7c6f02416000 ---p 00215000 08:03 526707 /usr/lib/x86_64-linux-gnu/libc.so.6
7c6f02416000-7c6f0241a000 r--p 00215000 08:03 526707 /usr/lib/x86_64-linux-gnu/libc.so.6
7c6f0241a000-7c6f0241c000 rw-p 00219000 08:03 526707 /usr/lib/x86_64-linux-gnu/libc.so.6
```

其大小为 805330944 Bytes，换算约 768 MB，即约 $6 * 128$ MB。稍多出来的部分内存用于对齐。

7C6ED21FA000 - 7C6F02200000 = FFFF FFFF CFFF A000 HEX FFFF FFFF CFFF A000 DEC -805,330,944	$805330944 \div 1024 \div 1024 =$ 768.0234375
--	---

heap 区下方只多出一块内存区，故程序申请的内存连续。

(2) 虚存大小变为 789232 kB，增量为 $789232 \text{ kB} - 2776 \text{ kB} = 786456 \text{ kB}$ ，即 $805330944 / 1024$ 。

```
hytidel@hytidel-virtual-machine:~$ cat /proc/305591/status
Name: memoryAllocatio
Umask: 0002
State: S (sleeping)
Tgid: 305591
Ngid: 0
Pid: 305591
PPid: 218183
TracerPid: 0
Uid: 1000 1000 1000 1000
Gid: 1000 1000 1000 1000
FDSize: 256
Groups: 4 24 27 30 46 122 135 136 1000
NSTgid: 305591
NSpid: 305591
NSpgid: 305591
NSSid: 218183
Kthread: 0
VmPeak: 789232 kB
VmSize: 789232 kB
```

3.2.3 释放 2、3、5 号内存区

在运行着 memoryAllocation 的终端按回车，程序释放 2、3、5 号内存区。

用如下命令查询该进程的此时的内存情况。

```
cat /proc/305591/maps
cat /proc/305591/status
```

运行结果如下。

(1) heap 区分裂为 3 段，对比程序的输出知，从下往上依次为 buffer1、buffer4 和 buffer6。


```
hytidel@hytidel-virtual-machine:~$ cat /proc/305591/maps
5aa9c3e58000-5aa9c3e59000 r--p 00000000 08:03 924817 /home/hytidel/0Sexp/exp3/memoryAllocation
5aa9c3e59000-5aa9c3e5a000 r-xp 00001000 08:03 924817 /home/hytidel/0Sexp/exp3/memoryAllocation
5aa9c3e5a000-5aa9c3e5b000 r--p 00002000 08:03 924817 /home/hytidel/0Sexp/exp3/memoryAllocation
5aa9c3e5b000-5aa9c3e5c000 r--p 00003000 08:03 924817 /home/hytidel/0Sexp/exp3/memoryAllocation
5aa9c3e5c000-5aa9c3e5d000 rw-p 00004000 08:03 924817 /home/hytidel/0Sexp/exp3/memoryAllocation
5aa9c3e5d000-5aa9c3e5e000 rw-p 00005000 00:00 0 [heap]
7c6ed21fa000-7c6eda1fb000 rw-p 00000000 00:00 0 buffer6
7c6ee21fc000-7c6eea1fd000 rw-p 00000000 00:00 0 buffer4
7c6efa1ff000-7c6f02200000 rw-p 00000000 00:00 0 buffer1
```

(2) 虚存大小变为 396004 kB, 减量为 789232 kB - 396004kB = 393228 kB, 即 3 * 128 MB * 1024 kB/MB。

```
hytidel@hytidel-virtual-machine:~$ cat /proc/305591/status
Name: memoryAllocatio
Umask: 0002
State: S (sleeping)
Tgid: 305591
Ngid: 0
Pid: 305591
PPid: 218183
TracerPid: 0
Uid: 1000 1000 1000 1000
Gid: 1000 1000 1000 1000
FDSize: 256
Groups: 4 24 27 30 46 122 135 136 1000
NSTgid: 305591
NSpid: 305591
NSpgid: 305591
NSSid: 218183
Kthread: 0
VmPeak: 789232 kB
VmSize: 396004 kB
```

3.2.4 再申请 1024 MB

在运行着 memoryAllocation 的终端按回车, 程序再申请 1024 MB 的内存。
运行结果如下图所示。

```
hytidel@hytidel-virtual-machine:~/0Sexp/exp3$ ./memoryAllocation

Allocated 128 MB of Mememory: 0x7c6efa1ff010 - 0x7c6f021ff00f
Allocated 128 MB of Mememory: 0x7c6ef21fe010 - 0x7c6efa1fe00f
Allocated 128 MB of Mememory: 0x7c6eea1fd010 - 0x7c6ef21fd00f
Allocated 128 MB of Mememory: 0x7c6ee21fc010 - 0x7c6eea1fc00f
Allocated 128 MB of Mememory: 0x7c6eda1fb010 - 0x7c6ee21fb00f
Allocated 128 MB of Mememory: 0x7c6ed21fa010 - 0x7c6eda1fa00f

Allocated 1024 MB of Mememory: 0x7c6e921f9010 - 0x7c6ed21f900f
```

用如下命令查询该进程的此时的内存情况。

```
cat /proc/305591/maps
cat /proc/305591/status
```

运行结果如下。

(1) heap 区下方 buffer6 的空间变大, 增量为 1073745920 Bytes, 约 1024 MB。稍多出来的内存用于对齐。

<pre>5aa9c3e5d000-5aa9c3e7e000 rw-p 00000000 00:00 0 [heap] 7c6ed21fa000-7c6eda1fb000 rw-p 00000000 00:00 0 buffer6 7c6ee21fc000-7c6eea1fd000 rw-p 00000000 00:00 0 7c6efa1ff000-7c6f02200000 rw-p 00000000 00:00 0</pre> <p>申请前</p>	<pre>5aa9c3e5d000-5aa9c3e7e000 rw-p 00000000 00:00 0 [heap] 7c6e921f9000-7c6eda1fb000 rw-p 00000000 00:00 0 buffer6 7c6ee21fc000-7c6eea1fd000 rw-p 00000000 00:00 0 7c6efa1ff000-7c6f02200000 rw-p 00000000 00:00 0</pre> <p>申请后</p>
--	--

$7C6ED21FA000 - 7C6E921F9000 =$ 4000 1000 <div> <div>HEX</div> <div>4000 1000</div> </div> <div> <div>DEC</div> <div>1,073,745,920</div> </div>	$1073745920 \div 1024 \div 1024 =$ 1,024.00390625
--	---

(2) 虚存大小增加，增量 $1444584 \text{ kB} - 396004 \text{ kB} = 1048580 \text{ kB}$ ，即 $1073745920 / 1024$ 。

```
hytidel@hytidel-virtual-machine:~$ cat /proc/305591/status
Name: memoryAllocatio
Umask: 0002
State: S (sleeping)
Tgid: 305591
Ngid: 0
Pid: 305591
PPid: 218183
TracerPid: 0
Uid: 1000    1000    1000    1000
Gid: 1000    1000    1000    1000
FDSize: 256
Groups: 4 24 27 30 46 122 135 136 1000
NSTgid: 305591
NSpid: 305591
NSpgid: 305591
NSSid: 218183
Kthread: 0
VmPeak: 1444584 kB
VmSize: 1444584 kB
```

3.2.5 再申请 64 MB

在运行着 memoryAllocation 的终端按回车，程序再申请 64 MB 的内存。
 猜测新申请的内存应在原 2 号内存区的位置。
 运行结果如下图所示。

```
hytidel@hytidel-virtual-machine:~/0Sexp/exp3$ ./memoryAllocation

Allocated 128 MB of Mememory: 0x7c6efa1ff010 - 0x7c6f021ff00f
Allocated 128 MB of Mememory: 0x7c6ef21fe010 - 0x7c6efa1fe00f
Allocated 128 MB of Mememory: 0x7c6eeaf1fd010 - 0x7c6ef21fd00f
Allocated 128 MB of Mememory: 0x7c6ee21fc010 - 0x7c6eeaf1fc00f
Allocated 128 MB of Mememory: 0x7c6eda1fb010 - 0x7c6ee21fb00f
Allocated 128 MB of Mememory: 0x7c6ed21fa010 - 0x7c6eda1fa00f

Allocated 1024 MB of Mememory: 0x7c6e921f9010 - 0x7c6ed21f900f

Allocated 64 MB of Mememory: 0x7c6ef61fe010 - 0x7c6efa1fe00f
```

新申请的内存的起点是 3.2.3 中释放的 2 号内存区。

但事实上，这不足以证明内核采用首次适应算法，因为释放的 2、3、5 号缓冲区大小相同，可能内核采用最佳适应算法时，规定分配后缓冲区剩余大小相同时高地址的优先。

将如下代码保存到 allocationAlgorithm.cpp 中。

```
#include <stdio.h>
```

```

#include <stdlib.h>
#include <unistd.h>

using uint = unsigned int;
const uint MB = 1024 * 1024;
const uint Sigma = 128;
const uint MAXN = 15;

char* myMalloc(uint memSize) {
    char* buffer = (char*)malloc(memSize);

    // 写
    for (uint i = 0; i < memSize; i++) {
        buffer[i] = i % Sigma;
    }

    printf("Allocated %d MB of Memory: %p - %p\n", memSize / MB, buffer, buffer +
memSize - 1);

    return buffer;
}

int main() {
    // 连续申请 6 个 128 MB 的空间
    char* buffers[MAXN];
    buffers[1] = myMalloc(128 * MB);
    buffers[2] = myMalloc(128 * MB);
    buffers[3] = myMalloc(128 * MB);
    buffers[4] = myMalloc(65 * MB);
    buffers[5] = myMalloc(128 * MB);
    buffers[6] = myMalloc(128 * MB);

    // 释放第 2 、 4 号
    free(buffers[2]);
    free(buffers[4]);

    // 申请 64 MB 的空间
    buffers[7] = myMalloc(64 * MB);

    return 0;
}

```

上述程序将 `buffer4` 的大小改为 65 MB，其它 5 个 `buffer` 的大小都为 128 MB。释放 `buffer2` 和 `buffer4`。

(1) 若内核采用首次适应算法，则新申请的内存应在原 `buffer2` 的位置。

(2) 若内核采用最佳适应算法，则新申请的内存应在原 `buffer4` 的位置。
用如下命令编译并运行 `allocationAlgorithm.cpp`。

```
g++ allocationAlgorithm.cpp -o allocationAlgorithm
./allocationAlgorithm
```

运行结果如下图所示，新申请的内存存在原 `buffer2` 的位置，证明内核采用首次适应算法。

```
hytidel@hytidel-virtual-machine:~/0Sexp/exp3$ ./allocationAlgorithm
Allocated 128 MB of Mememory: 0x7946ea1ff010 - 0x7946f21ff00f
Allocated 128 MB of Mememory: 0x7946e21fe010 - 0x7946ea1fe00f buffer2
Allocated 128 MB of Mememory: 0x7946da1fd010 - 0x7946e21fd00f
Allocated 65 MB of Mememory: 0x7946d60fc010 - 0x7946da1fc00f
Allocated 128 MB of Mememory: 0x7946ce0fb010 - 0x7946d60fb00f
Allocated 128 MB of Mememory: 0x7946c60fa010 - 0x7946ce0fa00f
Allocated 64 MB of Mememory: 0x7946e61fe010 - 0x7946ea1fe00f
hytidel@hytidel-virtual-machine:~/0Sexp/exp3$
```

3.2.6 总结

- (1) `/proc/pid/maps` 中将用户进程空间划分为[heap]段、[stack]段等，说明内存离散分配。
- (2) 3.2.5 的结果说明，内核采用首次适应算法分配内存。
- (3) 因新申请的空间会划分大的空闲区，故用户空间可能有碎片。

4. 单个进程的最大虚拟内存

设计一个程序测试出你的系统单个进程所能分配到的最大虚拟内存空间为多大。

4.1 程序

将如下代码保存到 `maxVirtualMemory1.cpp` 中。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

using uint = unsigned int;
const uint MB = 1024 * 1024;
const uint Sigma = 128;
const uint MAXN = 15;

char* myMalloc(uint memSize) {
    char* buffer = (char*)malloc(memSize);

    // 写
    for (uint i = 0; i < memSize; i++) {
        buffer[i] = i % Sigma;
    }

    printf("Allocated %d MB of Mememory: %p - %p\n", memSize / MB, buffer, buffer +
memSize - 1);
}
```



```

        return buffer;
    }

int main() {
    uint memSize = 256 * MB;
    uint cnt = 0;
    while (1) {
        printf("[%d] ", ++cnt);

        char* buffer = myMalloc(memSize);
        if (!buffer) { // 申请失败
            break;
        }
    }

    return 0;
}

```

4.2 进程的虚存限制

用如下命令编译并运行 maxVirtualMemory1.cpp。

```

g++ maxVirtualMemory1.cpp -o maxVirtualMemory1
./maxVirtualMemory1

```

运行结果如下图所示，申请了 $18 * 256 \text{ MB} = 4608 \text{ MB}$ 后进程被 OS 杀死。

```

hytidel@hytidel-virtual-machine:~/0Sexp/exp3$ ./maxVirtualMemory1
[1] Allocated 256 MB of Mememory: 0x7368df9ff010 - 0x7368ef9ff00f
[2] Allocated 256 MB of Mememory: 0x7368cf9fe010 - 0x7368df9fe00f
[3] Allocated 256 MB of Mememory: 0x7368bf9fd010 - 0x7368cf9fd00f
[4] Allocated 256 MB of Mememory: 0x7368af9fc010 - 0x7368bf9fc00f
[5] Allocated 256 MB of Mememory: 0x73689f9fb010 - 0x7368af9fb00f
[6] Allocated 256 MB of Mememory: 0x73688f9fa010 - 0x73689f9fa00f
[7] Allocated 256 MB of Mememory: 0x73687f9f9010 - 0x73688f9f900f
[8] Allocated 256 MB of Mememory: 0x73686f9f8010 - 0x73687f9f800f
[9] Allocated 256 MB of Mememory: 0x73685f9f7010 - 0x73686f9f700f
[10] Allocated 256 MB of Mememory: 0x73684f9f6010 - 0x73685f9f600f
[11] Allocated 256 MB of Mememory: 0x73683f9f5010 - 0x73684f9f500f
[12] Allocated 256 MB of Mememory: 0x73682f9f4010 - 0x73683f9f400f
[13] Allocated 256 MB of Mememory: 0x73681f9f3010 - 0x73682f9f300f
[14] Allocated 256 MB of Mememory: 0x73680f9f2010 - 0x73681f9f200f
[15] Allocated 256 MB of Mememory: 0x7367ff9f1010 - 0x73680f9f100f
[16] Allocated 256 MB of Mememory: 0x7367ef9f0010 - 0x7367ff9f000f
[17] Allocated 256 MB of Mememory: 0x7367df9ef010 - 0x7367ef9ef00f
[18] Allocated 256 MB of Mememory: 0x7367cf9ee010 - 0x7367df9ee00f
Killed
hytidel@hytidel-virtual-machine:~/0Sexp/exp3$

```

上述结果大于虚拟机的内存 4 GB，如下图所示。



这表明：进程被杀死是因为其申请的空间超过 OS 为其分配的虚存限制，而非超过系统的物理内存限制。

4.3 能分配到的最大虚存

4.2 的结果仍不能说明单个程序能分配的最大虚存，因为其中涉及了写操作，它受内存大小限制。

将如下代码保存到 maxVirtualMemory2.cpp 中。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

using uint = unsigned int;
const uint MB = 1024 * 1024;
const uint Sigma = 128;
const uint MAXN = 15;

char* myMalloc(uint memSize) {
    char* buffer = (char*)malloc(memSize);

    // 写
    /*for (uint i = 0; i < memSize; i++) {
        buffer[i] = i % Sigma;
    }*/

    printf("Allocated %d MB of Mememory: %p - %p\n", memSize / MB, buffer, buffer +
```



```
memSize - 1);

    return buffer;
}

int main() {
    uint memSize = 1024 * MB;
    uint cnt = 0;
    while (1) {
        printf("[%d] ", ++cnt);

        char* buffer = myMalloc(memSize);
        if (!buffer) { // 申请失败
            break;
        }
    }

    return 0;
}
```

用如下命令编译并运行 maxVirtualMemory2.cpp。

```
g++ maxVirtualMemory2.cpp -o maxVirtualMemory2
./maxVirtualMemory2
```

运行结果如下图所示，进程申请了非常非常多次虚存。

```

[21129] Allocated 1024 MB of Mememory: 0x66d6e7577010 - 0x66d72757700f
[21130] Allocated 1024 MB of Mememory: 0x66d6a7576010 - 0x66d6e757600f
[21131] Allocated 1024 MB of Mememory: 0x66d667575010 - 0x66d6a757500f
[21132] Allocated 1024 MB of Mememory: 0x66d627574010 - 0x66d66757400f
[21133] Allocated 1024 MB of Mememory: 0x66d5e7573010 - 0x66d62757300f
[21134] Allocated 1024 MB of Mememory: 0x66d5a7572010 - 0x66d5e757200f
[21135] Allocated 1024 MB of Mememory: 0x66d567571010 - 0x66d5a757100f
[21136] Allocated 1024 MB of Mememory: 0x66d527570010 - 0x66d56757000f
[21137] Allocated 1024 MB of Mememory: 0x66d4e756f010 - 0x66d52756f00f
[21138] Allocated 1024 MB of Mememory: 0x66d4a756e010 - 0x66d4e756e00f
[21139] Allocated 1024 MB of Mememory: 0x66d46756d010 - 0x66d4a756d00f
[21140] Allocated 1024 MB of Mememory: 0x66d42756c010 - 0x66d46756c00f
[21141] Allocated 1024 MB of Mememory: 0x66d3e756b010 - 0x66d42756b00f
[21142] Allocated 1024 MB of Mememory: 0x66d3a756a010 - 0x66d3e756a00f
[21143] Allocated 1024 MB of Mememory: 0x66d367569010 - 0x66d3a756900f
[21144] Allocated 1024 MB of Mememory: 0x66d327568010 - 0x66d36756800f
[21145] Allocated 1024 MB of Mememory: 0x66d2e7567010 - 0x66d32756700f
[21146] Allocated 1024 MB of Mememory: 0x66d2a7566010 - 0x66d2e756600f
[21147] Allocated 1024 MB of Mememory: 0x66d267565010 - 0x66d2a756500f
[21148] Allocated 1024 MB of Mememory: 0x66d227564010 - 0x66d26756400f
[21149] Allocated 1024 MB of Mememory: 0x66d1e7563010 - 0x66d22756300f
[21150] Allocated 1024 MB of Mememory: 0x66d1a7562010 - 0x66d1e756200f
[21151] Allocated 1024 MB of Mememory: 0x66d167561010 - 0x66d1a756100f
[21152] Allocated 1024 MB of Mememory: 0x66d127560010 - 0x66d16756000f
[21153] Allocated 1024 MB of Mememory: 0x66d0e755f010 - 0x66d12755f00f
[21154] Allocated 1024 MB of Mememory: 0x66d0a755e010 - 0x66d0e755e00f
[21155] Allocated 1024 MB of Mememory: 0x66d06755d010 - 0x66d0a755d00f
[21156] Allocated 1024 MB of Mememory: 0x66d02755c010 - 0x66d06755c00f
[21157] Allocated 1024 MB of Mememory: 0x66cfe755b010 - 0x66d02755b00f
[21158] Allocated 1024 MB of Mememory: 0x66cfa755a010 - 0x66cfe755a00f
[21159] Allocated 1024 MB of Mememory: 0x66cf67559010 - 0x66cfa755900f
[21160] Allocated 1024 MB of Mememory: 0x66cf27558010 - 0x66cf6755800f
[21161] Allocated 1024 MB of Mememory: 0x66cee7557010 - 0x66cf2755700f
[21162] Allocated 1024 MB of Mememory: 0x66cea7556010 - 0x66cee755600f
[21163] Allocated 1024 MB of Mememory: 0x66ce67555010 - 0x66cea755500f
^C
hytidel@hytidel-virtual-machine:~/OSexp/exp3$ █

```

这表明：只要不写数据，进程几乎可无限申请虚存。

此外，因为地址是无符号数，故猜测一段时间后会溢出，再次说明了虚存可无限申请。

4.4 补充

在 64 位 Linux 系统中，单个进程所能分配到的最大虚拟内存空间取决于内核的配置以及系统的地址空间布局。通常情况下，单个进程所能分配到的最大虚拟内存空间为 128 TB。

这个限制是由 CONFIG_VMSPLIT_64 内核配置选项决定的。在这种配置下，用户空间地址范围是从 0x0000000000000000 到 0x00007fffffffffff，共计 128 TB。这个范围中，一部分用于用户空间的虚拟内存，一部分用于内核空间的虚拟内存。

但需要注意的是，即使是 64 位系统，由于地址空间的布局，实际可用的虚拟内存空间可能会受到一些限制。

4.5 虚存限制因素

(1) 系统的地址长度。

- (2) 是否有写操作。
- (3) 物理内存和交换区的大小之和。
- (4) 内核的参数，如 `vm.max_map_count` 参数限制系统中最大的内存映射数。
- (5) 内存将换出到外存，故受辅存大小的限制。
- (6) 使用虚拟机时，虚拟机的虚存限制。

5. 虚存与物存的关系

编写一个程序，分配 256MB 内存空间（或其他足够大的空间），检查分配前后 `/proc/pid/status` 文件中关于虚拟内存和物理内存的使用情况，然后每隔 4KB 间隔将相应地址进行读操作，再次检查 `/proc/pid/status` 文件中关于内存的情况，对比前后两次内存情况，说明所分配物理内存（物理内存块）的变化。然后重复上面操作，不过此时为写操作，再观察其变化。

5.1 程序

将如下代码保存到 `VMAndPM.cpp` 中。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

using uint = unsigned int;
const uint KB = 1024;
const uint MB = 1024 * 1024;
const uint Sigma = 128;
const uint MAXN = 15;

char* myMalloc(uint memSize) {
    char* buffer = (char*)malloc(memSize);

    // 写
    /*for (uint i = 0; i < memSize; i++) {
        buffer[i] = i % Sigma;
    }*/

    printf("Allocated %d MB of Mememory: %p - %p\n", memSize / MB, buffer, buffer +
memSize - 1);

    return buffer;
}

int main() {
    getchar();

    // 申请 256 MB 的内存
    uint memSize = 256 * MB;
```

```
char* buffer = myMalloc(memSize);

printf("Allocation completed.\n");
getchar();

// 每隔 4 kB 读一次
for (uint i = 0; i < memSize; i += 4 * KB) {
    uint tmp = buffer[i];
}

printf("Reading completed.\n");
getchar();

// 每隔 4 kB 写一次
for (uint i = 0; i < memSize; i += 4 * KB) {
    buffer[i] = i % Sigma;
}

printf("Writing completed.\n");
getchar();

return 0;
}
```

5.2 读操作

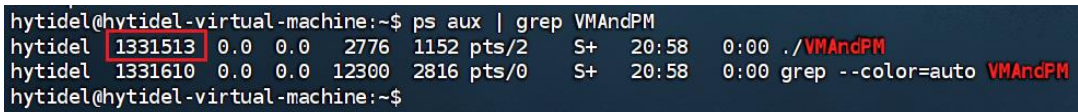
用如下命令编译并运行 VMAndPM.cpp。

```
g++ VMAndPM.cpp -o VMAndPM
./VMAndPM
```

用如下命令查询 VMAndPM 的 PID。

```
ps aux | grep VMAndPM
```

结果如下图所示，其 PID 为 1331513。



```
hytidel@hytidel-virtual-machine:~$ ps aux | grep VMAndPM
hytidel 1331513 0.0 0.0 2776 1152 pts/2 S+ 20:58 0:00 ./VMAndPM
hytidel 1331610 0.0 0.0 12300 2816 pts/0 S+ 20:58 0:00 grep --color=auto VMAndPM
hytidel@hytidel-virtual-machine:~$
```

用如下命令查询初始时该进程的虚存和物存情况。

```
cat /proc/1331513/status | grep Vm
```

运行结果如下，虚存 2776 kB，物存 1152 kB。


```
hytidel@hytidel-virtual-machine:~$ cat /proc/1331513/status | grep Vm
VmPeak:      2776 kB
VmSize:      2776 kB 虚存
VmLck:        0 kB
VmPin:        0 kB
VmHWM:       1152 kB
VmRSS:       1152 kB 物存
VmData:       224 kB
VmStk:        132 kB
VmExe:         4 kB
VmLib:       1796 kB
VmPTE:        44 kB
VmSwap:        0 kB
hytidel@hytidel-virtual-machine:~$
```

在运行着 VMAndPM 的终端按回车，运行结果如下图所示，程序申请 256 MB 内存。

```
hytidel@hytidel-virtual-machine:~/0Sexp/exp3$ ./VMAndPM
Allocated 256 MB of Memory: 0x7559befff010 - 0x7559cefff00f
Allocation completed.
```

用如下命令查询该进程此时的虚存和物存情况。

```
cat /proc/1331513/status | grep Vm
```

运行结果如下，虚存 264924 kB，物存 1408 kB。

```
hytidel@hytidel-virtual-machine:~$ cat /proc/1331513/status | grep Vm
VmPeak:      264924 kB
VmSize:      264924 kB
VmLck:        0 kB
VmPin:        0 kB
VmHWM:       1408 kB
VmRSS:       1408 kB
VmData:      262372 kB
VmStk:        132 kB
VmExe:         4 kB
VmLib:       1796 kB
VmPTE:        52 kB
VmSwap:        0 kB
hytidel@hytidel-virtual-machine:~$
```

虚存、物存增量如下图所示，虚存增大约 256 MB，稍多出的内存用于对齐；物存增量 256 kB。

这表明：申请内存的操作只增加了虚存，对物存无明显增加。

$(264924 - 2776) \div 1024 =$	$1408 - 1152 =$
256.00390625	256
虚存增量	物存增量

在运行着 VMAndPM 的终端按回车，运行结果如下图所示，程序在申请的内存中每隔 4 kB 读一次。

```
hytidel@hytidel-virtual-machine:~/0Sexp/exp3$ ./VMAndPM
Allocated 256 MB of Memory: 0x7559befff010 - 0x7559cefff00f
Allocation completed.
Reading completed.
```

用如下命令查询该进程此时的虚存和物存情况。

```
cat /proc/1331513/status | grep Vm
```

运行结果如下，虚存 264924 kB，物存 1408 kB。不变。

```
hytidel@hytidel-virtual-machine:~$ cat /proc/1331513/status | grep Vm
VmPeak:    264924 kB
VmSize:    264924 kB
VmLck:      0 kB
VmPin:      0 kB
VmHWM:     1408 kB
VmRSS:     1408 kB
VmData:    262372 kB
VmStk:      132 kB
VmExe:       4 kB
VmLib:     1796 kB
VmPTE:      564 kB
VmSwap:      0 kB
hytidel@hytidel-virtual-machine:~$
```

这表明：读内存的操作不增加物存。

5.3 写操作

在运行着 VMAndPM 的终端按回车，运行结果如下图所示，程序在申请的内存中每隔 4 kB 写一次。

```
hytidel@hytidel-virtual-machine:~/0Sexp/exp3$ ./VMAndPM

Allocated 256 MB of Memory: 0x7559befff010 - 0x7559cefff00f
Allocation completed.

Reading completed.

Writing completed.
```

用如下命令查询该进程此时的虚存和物存情况。

```
cat /proc/1331513/status | grep Vm
```

运行结果如下，虚存 264924 kB 不变，物存增加至 263424 kB。

```
hytidel@hytidel-virtual-machine:~$ cat /proc/1331513/status | grep Vm
VmPeak:    264924 kB
VmSize:    264924 kB
VmLck:      0 kB
VmPin:      0 kB
VmHWM:    263424 kB
VmRSS:    263424 kB
VmData:    262372 kB
VmStk:      132 kB
VmExe:       4 kB
VmLib:     1796 kB
VmPTE:      564 kB
VmSwap:      0 kB
hytidel@hytidel-virtual-machine:~$
```

物存增量如下图所示，约 256 MB。

$(263424 - 1408) \div 1024 \div$

255.875

这表明：写操作会增加物存。

5.4 总结

Linux 的内存只有被写时才会真正占用物存，分配、读取内存都只占用虚存。

6. 竞争物存

编写并运行（在第 5 步的程序未退出前）另一进程，分配等于或大于物理内存的空间，然后每隔 4KB 间隔将相应地址的字节数值增 1，此时再查看前一个程序的物理内存变化，观察两个进程竞争物理内存的现象。

6.1 可用内存

用如下命令查询系统此时可用内存的大小。

```
free -m
```

运行结果如下图所示，有 2796 MB 的可用内存，交换区有 1561 MB 的空闲内存，共 4357 MB 的可用内存。

```
hytidel@hytidel-virtual-machine:~$ free -m
              total        used         free       shared    buff/cache   available
Mem:           3870         831         2096           0          942        2796
Swap:          2139         578         1561
hytidel@hytidel-virtual-machine:~$
```

6.2 程序

将如下代码保存到 memoryCompetitor.cpp 中。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

using uint = unsigned int;
const uint KB = 1024;
const uint MB = 1024 * 1024;
const uint Sigma = 128;
const uint MAXN = 15;

char* myMalloc(uint memSize) {
    char* buffer = (char*)malloc(memSize);

    // 写
    /*for (uint i = 0; i < memSize; i++) {
        buffer[i] = i % Sigma;
    }*/

    printf("Allocated %d MB of Mememory: %p - %p\n", memSize / MB, buffer, buffer +
memSize - 1);

    return buffer;
}
```

```

}

int main() {
    getchar();

    // 申请 3500 MB 的内存
    uint memSize = 3500 * MB;
    char* buffer = myMalloc(memSize);

    printf("Allocation completed.\n");
    getchar();

    // 每隔 4 kB 将字节值 + 1
    for (uint i = 0; i < memSize; i += 4 * KB) {
        buffer[i]++;
    }

    printf("Writing completed.\n");
    getchar();

    return 0;
}

```

用如下命令编译并运行 memoryCompetitor.cpp。

```

g++ memoryCompetitor.cpp -o memoryCompetitor
./memoryCompetitor

```

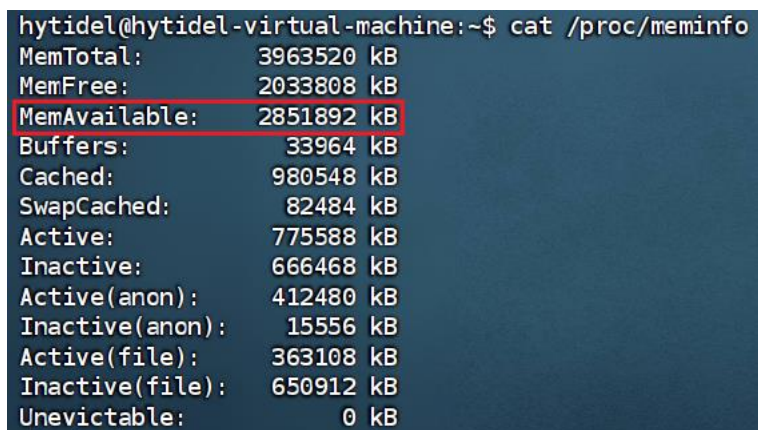
用如下命令查询此时系统的可用内存大小。

```

cat /proc/meminfo

```

运行结果如下图所示，有 2851892 kB 的可用内存。



```

hytidel@hytidel-virtual-machine:~$ cat /proc/meminfo
MemTotal:       3963520 kB
MemFree:        2033808 kB
MemAvailable:   2851892 kB
Buffers:        33964 kB
Cached:         980548 kB
SwapCached:     82484 kB
Active:         775588 kB
Inactive:       666468 kB
Active(anon):   412480 kB
Inactive(anon): 15556 kB
Active(file):   363108 kB
Inactive(file): 650912 kB
Unevictable:    0 kB

```

在运行着 memoryCompetitor 的终端按回车，程序申请 3500 MB 的内存。运行结果如下图所示。


```
hytidel@hytidel-virtual-machine:~/0Sexp/exp3$ ./memoryCompetitor  
Allocated 3500 MB of Mememory: 0x783d05dff010 - 0x783de09ff00f  
Allocation completed.
```

用如下命令查询此时系统的可用内存大小。

```
cat /proc/meminfo
```

运行结果如下图所示，可用内存大小稍有减小，但非常不明显。

```
hytidel@hytidel-virtual-machine:~$ cat /proc/meminfo  
MemTotal:       3963520 kB  
MemFree:        2012996 kB  
MemAvailable:   2831136 kB  
Buffers:        34012 kB  
Cached:         980548 kB  
SwapCached:     82484 kB  
Active:         775916 kB  
Inactive:       666516 kB  
Active(anon):   412808 kB  
Inactive(anon): 15556 kB  
Active(file):   363108 kB  
Inactive(file): 650960 kB  
Unevictable:    0 kB  
Mlocked:        0 kB  
SwapTotal:     2191356 kB  
SwapFree:      1600040 kB
```

在运行着 memoryCompetitor 的终端按回车，程序每隔 4 kB 令字节值 + 1。运行结果如下图所示。

```
hytidel@hytidel-virtual-machine:~/0Sexp/exp3$ ./memoryCompetitor  
Allocated 3500 MB of Mememory: 0x783d05dff010 - 0x783de09ff00f  
Allocation completed.  
Writing completed.
```

用如下命令查询系统此时可用内存的大小。

```
free -m
```

运行结果如下图所示，有 179 MB 的可用内存，交换区有 696 MB 的空闲内存，共 875 MB 的可用内存。

```
hytidel@hytidel-virtual-machine:~$ free -m  
              total        used         free      shared  buff/cache   available  
Mem:          3870         3472           195           0          202         179  
Swap:         2139         1443           696
```

用如下命令查询此时系统的可用内存大小。

```
cat /proc/meminfo
```

运行结果如下图所示，可用内存减小至 191900 kB。

```
hytidel@hytidel-virtual-machine:~$ cat /proc/meminfo
MemTotal:        3963520 kB
MemFree:         209516 kB
MemAvailable:    191900 kB
Buffers:         5596 kB
Cached:          154924 kB
SwapCached:      75308 kB
Active:          1814916 kB
Inactive:        1477448 kB
Active(anon):    1729088 kB
Inactive(anon):  1403144 kB
Active(file):    85828 kB
Inactive(file):  74304 kB
```

用如下命令查询 VMAAndPM 进程的虚存情况。

```
cat /proc/1331513/status | grep Vm
```

结果如下图所示，有 91776 kB 被换出。

```
hytidel@hytidel-virtual-machine:~$ cat /proc/1331513/status | grep Vm
VmPeak:      264924 kB
VmSize:      264924 kB
VmLck:        0 kB
VmPin:        0 kB
VmHWM:       263424 kB
VmRSS:       171904 kB
VmData:      262372 kB
VmStk:        132 kB
VmExe:         4 kB
VmLib:        1796 kB
VmPTE:        564 kB
VmSwap:      91776 kB
hytidel@hytidel-virtual-machine:~$
```

这表明：物存不够时，会将一部分未用的内存换出，逻辑上扩充了物存。

7. 访存方式对比

分配足够大的内存空间，其容量超过系统现有的空闲物理内存的大小，1) 按 4KB 的间隔逐个单元进行写操作，重复访问数遍（使得程序运行时间可测量）；2) 与前面访问总量和次数不变，但是将访问分成 16 个连续页为一组，逐组完成访问，记录运行时间。观察系统的状态，比较两者运行时间，给出判断和解释。

7.1 系统空闲内存

用如下命令查询系统此时可用内存的大小。

```
free -m
```

运行结果如下图所示，有 3056 MB 的可用内存，交换区有 1553 MB 的空闲内存，共 4609 MB 的可用内存。

```
hytidel@hytidel-virtual-machine:~$ free -m
              total        used         free      shared  buff/cache   available
Mem:           3870          593          3070           0         207        3056
Swap:          2139          586           1553
hytidel@hytidel-virtual-machine:~$
```

7.2 逐单元访问

将如下代码保存到 readPerUnit.cpp 中。

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

using uint = unsigned int;
const uint KB = 1024;
const uint MB = 1024 * 1024;
const uint Sigma = 128;
const uint MAXN = 15;
const uint ROUND = 10;

char* myMalloc(uint memSize) {
    char* buffer = (char*)malloc(memSize);

    // 写
    /*for (uint i = 0; i < memSize; i++) {
        buffer[i] = i % Sigma;
    }*/

    printf("Allocated %d MB of Mememory: %p - %p\n", memSize / MB, buffer, buffer +
memSize - 1);

    return buffer;
}

int main() {
    // 申请 3000 MB 的内存
    uint memSize = 3000 * MB;
    char* buffer = myMalloc(memSize);

    printf("Allocation completed.\n");

    clock_t my_clock = clock();

    for (uint round = 0; round < ROUND; round++) {
        // 每隔 4 kB 写一次
        for (uint i = 0; i < memSize; i += 4 * KB) {
            buffer[i] = i % Sigma;
        }
    }

    uint timeCost = clock() - my_clock;
    printf("Writing completed.\n");
}

```

```
printf("Time cost %d ms.\n", timeCost);

return 0;
}
```

用如下命令编译并运行 readPerUnit.cpp。

```
g++ readPerUnit.cpp -o readPerUnit
./readPerUnit
```

运行结果如下图所示，耗时 18199217 ms。

```
hytidel@hytidel-virtual-machine:~/0Sexp/exp3$ ./readPerUnit
Allocated 3000 MB of Memory: 0x7de9835ff010 - 0x7dea3edff00f
Allocation completed.
Writing completed.
Time cost 18199217 ms.
hytidel@hytidel-virtual-machine:~/0Sexp/exp3$
```

7.3 按组访问

Linux 的页大小为 4 kB。

将如下代码保存到 readPerGroup.cpp 中。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

using uint = unsigned int;
const uint KB = 1024;
const uint MB = 1024 * 1024;
const uint PAGE = 4 * KB;
const uint Sigma = 128;
const uint MAXN = 15;
const uint ROUND = 10;

char* myMalloc(uint memSize) {
    char* buffer = (char*)malloc(memSize);

    // 写
    /*for (uint i = 0; i < memSize; i++) {
        buffer[i] = i % Sigma;
    }*/

    printf("Allocated %d MB of Memory: %p - %p\n", memSize / MB, buffer, buffer +
memSize - 1);

    return buffer;
}
```

```

int main() {
    // 申请 3000 MB 的内存
    uint memSize = 3000 * MB;
    char* buffer = myMalloc(memSize);

    printf("Allocation completed.\n");

    uint pageCount = (memSize + PAGE - 1) / PAGE; // 内存的页数
    uint batchSize = 16 * PAGE; // 每组的大小

    auto work = [&]() {
        for (uint page = 0; page < pageCount; page += 16) {
            for (uint i = 0; i < batchSize; i += 4 * KB) {
                uint addr = page * PAGE + i;
                if (addr >= memSize) {
                    return;
                }

                buffer[addr] = i % Sigma;
            }
        }
    };

    clock_t my_clock = clock();

    work();

    uint timeCost = clock() - my_clock;
    printf("Writing completed.\n");
    printf("Time cost %d ms.\n", timeCost);

    return 0;
}

```

用如下命令编译并运行 readPerGroup.cpp。

```

g++ readPerGroup.cpp -o readPerGroup
./readPerGroup

```

运行结果如下图所示，耗时 2226051 ms。

```

hytidel@hytidel-virtual-machine:~/OSexp/exp3$ ./readPerGroup
Allocated 3000 MB of Memory: 0x71dffa000 - 0x71e0b67ff
Allocation completed.
Writing completed.
Time cost 2226051 ms
hytidel@hytidel-virtual-machine:~/OSexp/exp3$

```

7.4 对比

按 16 个连续页为一组逐组写的时间远小于按 4 kB 的间隔逐单元写所需的时间，原因：

(1) 页表更新开销：按 4 kB 的间隔逐单元写时，需频繁更新页表。因 Linux 的页表大小也为 4 kB，每次写入都导致页表项更新，将产生额外开销；按 16 个连续页为一组写时，页表更新次数少，开销小。

(2) 缓存命中率：按 16 个连续页为一组写时，根据局部性原理，CPU 更好地利用缓存来访问，提高访存效率和写入效率。

(3) 缓冲区：按 16 个连续页为一组写时，可利用内存写入缓冲区 (write-back buffer) 来缓解内存一致性带来的开销。

8. TLB 效率

显然按 4 kB 的间隔逐单元写涉及到的 TLB 替换的次数会多于按 16 个连续页为一组逐组写的 TLB 替换次数。本部分讨论 TLB 的效率是否会对运行时间产生主要影响。

8.1 传统 TLB

8.1.1 传统 LTB 的效率实验

1992 年的论文 A Simulation Based Study of TLB Performance (<https://dl.acm.org/doi/pdf/10.1145/146628.139708>, 或见【附件】)中提到: Early studies have shown that TLB miss penalties consume 6% of all machine cycles [4] and 4% of execution time [3], and hence can have a significant impact on machine performance.

论文中的实验测得 TLB 的 1 次 miss 约增加 100 个时钟周期的罚时。

8.1.2 换算

用如下命令查询当前系统的时钟频率。

```
grep 'MHz' /proc/cpuinfo | uniq
```

运行结果如下图所示，本次实验的机器的时钟频率为 2918.401 MHz。

```
hytidel@hytidel-virtual-machine:~$ grep 'MHz' /proc/cpuinfo | uniq
cpu MHz          : 2918.401
hytidel@hytidel-virtual-machine:~$
```

时钟周期 = $1 / (2918.401 * 10^6) \approx 3.4e-10$ s。

即使 7. 访存方式对比 中的 $10 * (3000 \text{ MB} / 4 \text{ kB}) = 7680000$ 次写操作中有 10% 的 miss，则增加的时间约 $7680000 * 10\% * 3.4e-10 \text{ s} = 2.6e-3 \text{ s} = 2.6 \text{ ms}$ 。

事实上远达不到这么高的 miss 率，故 TLB 的效率不会对运行时间产生主要影响。

8.2 现代 TLB

3.5.3.1. Translation Lookaside Buffers

Each CPU in the Cortex® -A53 MPCore™ contains micro and main translation lookaside buffers (TLBs).

Table 32. MMU Features of Each CPU in the Cortex® -A53 MPCore™

TLB Type	Memory Type	Number of Entries	Associativity
Micro TLB	Instruction	10	Fully associative
Micro TLB	Data	10	Fully associative
Main TLB	Instruction and Data	512	Four-way set-associative

Each CPU also includes:

- 4-way set associative 64-entry walk cache that holds the result of a stage 1 translation. The walk cache holds entries fetched from the secure and non-secure state.
- 4-way set associative 64-entry intermediate physical address (IPA) cache. This cache holds map points between intermediate physical addresses and physical addresses. Only non-secure exception level 1 (EL1) and exception level 0 (EL0) stage 2 translations use this cache.

TLB entries include global and application specific identifiers to prevent context switch TLB flushes. The architecture also supports virtual machine identifiers (VMIDs) to prevent TLB flushes on virtual machine switches by hypervisor.

The micro TLBs are the first level of caching for the translation table information. The unified main TLB handles misses from the micro TLBs.

When the main TLB performs maintenance operations it flushes both the instruction and data micro TLBs.

现代 CPU 普遍采用乱序多发射和多线程同步（Simultaneous multithreading, SMT），故现代缓存设计基于 Multied-Bank 和 Pipeline Cache，TLB 的设计也遵循该规则，故在现代的机器上难以通过实验验证。

四、实验结论

(提供运行结果, 对结果进行探讨、分析、评价, 并提出结论性意见和改进想法)

1. Linux 进程的内存分配、释放函数

1.1 kmalloc()、kzalloc()、vmalloc()的共同点

- (1) 用于申请内核空间的内存。
- (2) 内存以字节为单位分配。
- (3) 分配的虚拟内存地址连续。

1.2 kmalloc()、kzalloc()、vmalloc()的共同点

- (1) kzalloc()函数本质是加了清零的 kmalloc()函数。下面不再区分 kzalloc()函数和 kmalloc()函数。
- (2) kmalloc()函数申请的内存大小限制 128 kB, 而 vmalloc()函数申请的内存大小由剩余空间决定。
- (3) kmalloc()函数保证申请的内存的物理地址连续, 但 vmalloc()函数不保证。
- (4) kmalloc()函数的申请过程可以是原子过程, 但 vmalloc()函数的申请过程可能被阻塞。
- (5) kmalloc()函数分配内存开销小, 比 vmalloc()函数快。

1.3 备注

- (1) 一般只有内存被 DMA 访问时才需物理上连续。
- (2) 为保证性能, 内核中一般用 kmalloc()函数, 只有需要大块内存时采用 vmalloc()函数。

2. Linux 的内存映射

- (1) /proc/pid/maps 文件列出了进程当前的内存映射情况, 每一行代表了进程中的一个内存区域, 包括了该内存区域的起始地址、结束地址、权限等信息。
- (2) /proc/pid/status 文件包含了关于进程状态的各种信息, 其中也包括了一些关于内存映射的统计数据。
- (3) /proc/pid/smaps 文件提供了更详细的关于进程内存映射的信息, 包括了每个内存区域的更详细的统计数据。

3. 内存分配算法实验

- (1) /proc/pid/maps 中将用户进程空间划分为[heap]段、[stack]段等, 说明内存离散分配。
- (2) 3.2.5 的结果说明, 内核采用首次适应算法分配内存。
- (3) 因新申请的空间会划分大的空闲区, 故用户空间可能有碎片。

4. 单个进程的最大虚存

- (1) 进程被杀死是因为其申请的空间超过 OS 为其分配的虚存限制, 而非超过系统

的物理内存限制。

(2) 只要不写数据，进程几乎可无限申请虚存。

(3) 地址是无符号数，一段时间后会溢出，再次说明了虚存可无限申请。

5. 虚存与物存的关系

(1) 读内存的操作不增加物存，写内存的操作会增加物存。

(2) Linux 的内存只有被写时才会真正占用物存，分配、读取内存都只占用虚存。

6. 竞争物存

物存不够时，会将一部分未用的内存换出，逻辑上扩充了物存。

7. 访存方式对比

按 16 个连续页为一组逐组写的时间远小于按 4 kB 的间隔逐单元写所需的时间，原因：

(1) 页表更新开销：按 4 kB 的间隔逐单元写时，需频繁更新页表。因 Linux 的页表大小也为 4 kB，每次写入都导致页表项更新，将产生额外开销；按 16 个连续页为一组写时，页表更新次数少，开销小。

(2) 缓存命中率：按 16 个连续页为一组写时，根据局部性原理，CPU 更好地利用缓存来访问，提高访存效率和写入效率。

(3) 缓冲区：按 16 个连续页为一组写时，可利用内存写入缓冲区 (write-back buffer) 来缓解内存一致性带来的开销。

五、实验体会

(根据自己情况填写)

在本次《内存管理实验》中，我深入学习了 Linux 操作系统的内存分配与使用的编程接口，并加深了对内存分配与使用操作的直观认识。通过实验，我掌握了 Linux 操作系统中进程的逻辑编程地址和物理地址间的映射关系，并学会了使用一些重要的内存管理函数和查看内存使用情况的命令。

通过本次实验，我深入了解了 Linux 操作系统的内存管理机制，掌握了相关的编程接口和命令，并通过实验加深了对内存分配与使用操作的理解。同时，通过提高部分的实验，我进一步探究了进程之间竞争物理内存的情况，并比较了不同内存访问方式的效率，对操作系统的内存管理有了更深入的认识。这次实验让我收获颇丰，对操作系统课程的学习也更加深入和扎实了。

注：“指导教师批阅意见”栏请单独放置一页

指导教师批阅意见：

成绩评定：

指导教师签字：

年 月 日

备注：