操作系统内存管理作业报告

陈荣钊 伍冠宇 高俊峰 孙梓健

1.环境和运行说明

本作业在教师提供的xv6版本下进行编写，配置方法同xv6。

测试环境： `WSL` 、 `Ubuntu18.04`

需求软件：qemu, make, gcc

运行方法：在xv6文件夹下输入 `make qemu-nox` 可打开无图形界面的qemu命令行，等待初始化完成出现 `$` 号时，可以输入命令。随后输入下述命令进行相应测试。

`virtualMemoryTest` 虚拟页式存储测试

`StackGrowTest` 栈增测试

`NullPtrProtectTest` 零指针保护测试

`mallocTest` 随机申请和释放内存统计外碎片大小

2.作业情况说明

本作业实现的内存管理功能有：

- 虚拟页式存储管理
- 栈自增
- 零指针保护
- 对比四种分区分配算法

2.1 虚拟页式存储管理

xv6的内存管理系统对于用户进程采用可变分区和页式存储进行管理。xv6的物理内存只有224MB，且对任意用户进程都没有空间限制，用户进程可以随意索取内存空间，直至物理内存消耗殆尽。为了增强xv6内存管理系统的可扩展性，我们实现了虚拟页式存储管理。每个进程将被限制拥有的物理内存页数量，若进程索要更多的物理内存，则将触发页面置换，若进程需要使用被置换的物理页，将触发缺页中断。

2.1.1 实现原理

好的页面置换算法离不开硬件支持，所以我们选择了FIFO算法，不需要硬件系统的支持，而且容易实现。

FIFO算法需要将已申请的物理页连成链表，还需要记录被置换出去的物理页。在 `virtualMemory.h` 中定义了物理页链表 `internalMemoryEntry` ，记录物理页对应虚拟地址的同时，添加了链表需要的 `pre` 和 `nxt` 指针。xv6是以页为单位分配空间，而一个物理页链表项仅需要 `12 Bytes` 。因此在物理页链表项的基础上，定义了内存页表 `internalMemoryTable` ，每张表存储 `340` 个内存页信息，刚好一页。每个进程分配 `189` 个页表，用完即止，因此每个进程最多占用 `250MB` 的物理内存空间。当进程需要超出 `250MB` 的物理内存时，即触发FIFO算法，从内存页链表中取出最后一页进行置换。

被置换到外存的页面同样需要记录它所代表的内存页的虚拟地址，这样才能在缺页中断发生时正确将其写回进程页表中。 `externalMemoryEntry` 定义了一个外存页项，便于查询放在外存的虚拟地址。同物理内存的记录类似，设置了外存表 `externalMemoryTable` ，每 `1022` 个外存页信息放在一起，恰好占一页空间，需要查询外存页时在当前表中遍历即可。

2.1.2 测试方法

在 `VirtualMemoryTest.c` 中，将在一个进程内申请 `252MB` 的内存，前 `251MB` 内存会直接记录在进程的内存表中，剩下 `1MB` 的内存将在申请时触发页面置换算法，进程会把最久未使用的页置换出去。FIFO算法的弱点在于，最久未使用的内存页可能储存着进程的关键信息，如进程的栈帧。因此当最后 `1MB` 内存申请完毕，回溯过程需要用到栈帧信息，触发了缺页中断。我们将内存页的置换过程以调试信息的形式输出在xv6命令行中，最后进程将关键页调回，回归测试代码，完成虚拟内存测试。

2.1.3 主要涉及的文件

`vm.c` 中的 `allocuvm` 和 `deallocuvm` 函数，将在申请页和消除页过程中对进程的内存、外存表更新。 `pageFault` 函数将处理缺页中断的情形。

`virtualMemory.h` 定义了内存表和外存表数据结构， `virtualMemory.c` 是内存表和外存表需要的算法实现。

2.2 栈自增

xv6中，栈的大小被统一写死为4KB。这对于系统栈一般是够用的，但是对于用户程序就捉襟见肘。栈自增可以解决该问题，使栈的空间在程序运行时动态地增加，大大地增强了OS的服务能力。

2.2.1 实现原理

在进程结构体 `proc` 中新增变量 `stackSize` 表示进程当前使用的栈大小。把用户栈（虚拟地址）放在最顶端即 `KERNBASE` 向下增长。保持栈底到堆顶至少一个页面的差距。当用户使用栈底之下的地址时，因为该

地址还未分配物理空间，所以会引发缺页中断，此时增长栈。

2.2.2 测试方法

在 StackGrowTest.c 中， testKB 和 testMB 以递归方式每次声明char数组在栈中申请1KB或1MB的空间。经测试可申请249MB的栈空间，即递归249层。 main 中以直接声明char数组的方式直接申请栈空间，经测试可申请251MB的栈空间。

2.2.3 主要涉及的文件

vm.c中的 stackIncre 函数，增长栈空间，新分配一页物理内存给进程，更新页表。

vm.c中的 pageFault 函数，虚拟地址处于特定范围时调用 stackIncre 函数。

exec.c 做栈的初始化， proc.c fork时复制栈空间， syscall.c 、 sysproc.c 根据栈大小判断地址合法范围。

2.3 零指针保护

零指针是指向地址为0的指针。在C语言中，零指针时有效的，会访问虚拟地址为0的地址，这个地址是否有效由操作系统规定。现代编程习惯认为指向地址为0的指针是空指针，空指针不可使用，这样规定了空指针，可以防止指针指向不确定的内存空间而引起错误。xv6中可以使用零指针，我们实现了零指针保护的功能，即零指针不可用。

2.3.1 实现原理

在 exec.c 中，加载程序进内存时，原先是从地址零开始装载，现改为从第二个页面开始，空出第一个页面。同时修改 Makefile 使编译程序时所有地址加上了一个页面的偏移量(-Ttest)。这样没有对用户行为造成改变，但虚拟地址第一页为空。然后任何对虚拟地址小于一个页面的地址进行访问都会引发缺页中断，这时再杀死进程就达到了零指针保护的目的。

2.3.2 测试方法

NullPtrProtectTest.c 中先声明了一个零指针，然后对其访问。如果程序正常结束，说明测试失败，零指针保护无效。如果显示OS杀死该进程，说明测试成功，零指针保护有效。`

2.4 对比四种分区分配算法

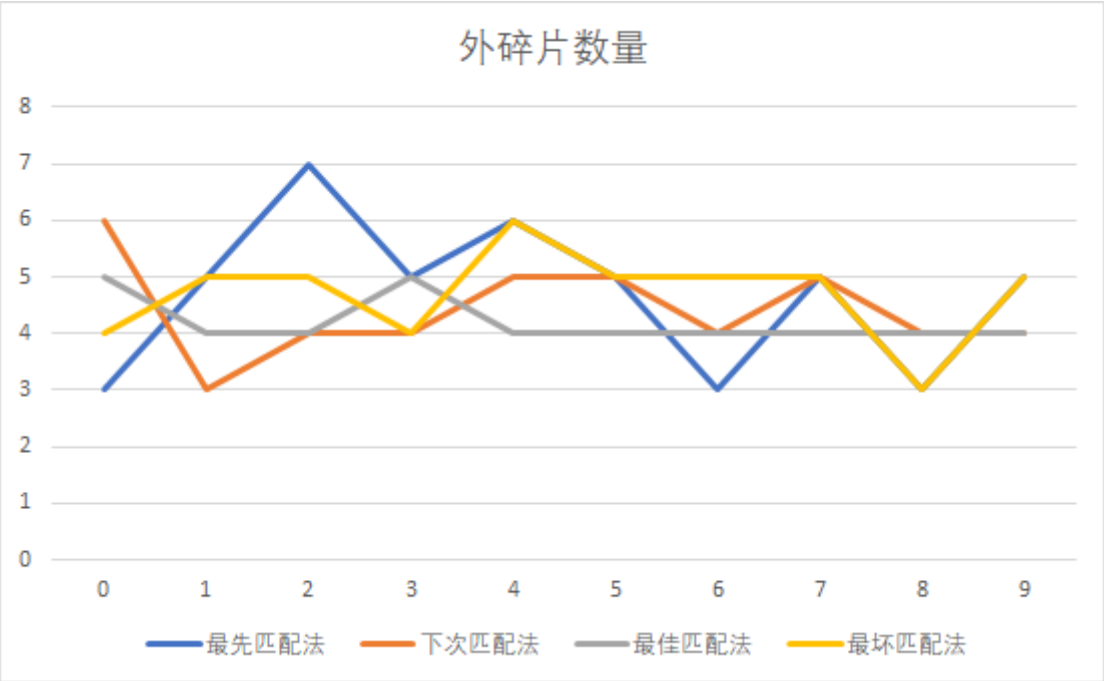
xv6的可变分区内存分配算法默认采用下次匹配法，我们在 umalloc.c 中实现了最先匹配法、最佳匹配法和最坏匹配法。在 mallocTest.c 中对四种内存分配方法进行测试，以外碎片大小为指标考察四种内存分配方法的性能。

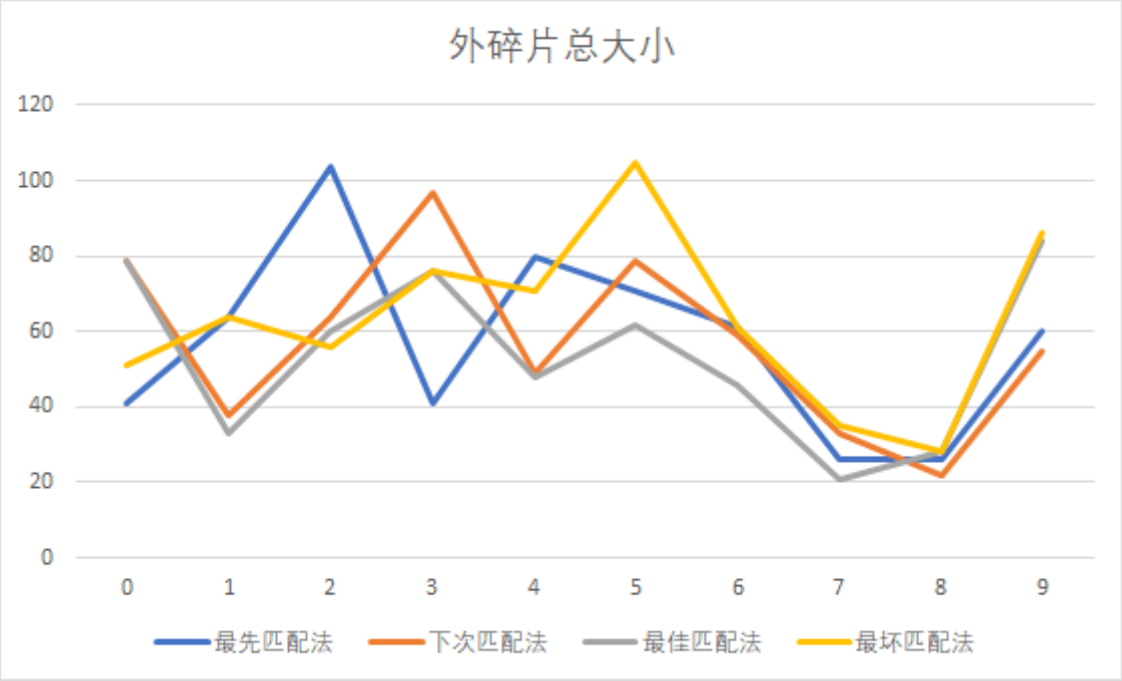
2.4.1 测试方法

每种算法分别进行10次测试，每次测试进行30次malloc和随机次数的free操作，模拟实际情况中内存的分配与回收，考察外碎片的数量和大小。

2.4.2 实验结果

十次测试的原始数据如下：





进行平均值处理后得到以下结果：

外碎片数量										
测试次数	0	1	2	3	4	5	6	7	8	9
最先匹配法	3	5	7	5	6	5	3	5	3	5
下次匹配法	6	3	4	4	5	5	4	5	4	4
最佳匹配法	5	4	4	5	4	4	4	4	4	4
最坏匹配法	4	5	5	4	6	5	5	5	3	5

外碎片总大小													
测试次数	0	1	2	3	4	5	6	7	8	9	平均值	最大值	最小值
最先匹配法	41	64	104	41	80	71	61	26	26	60	57.4	104	26
下次匹配法	79	38	64	97	49	79	59	33	22	55	57.5	97	22
最佳匹配法	79	33	60	76	48	62	46	21	28	84	53.7	84	21
最坏匹配法	51	64	56	76	71	105	61	35	28	86	63.3	105	28

2.4.3 小结

从实验结果可以看出，最佳匹配法有最好的效果；最先匹配法和下次匹配法的效果相近，比最佳匹配法略差，但是这两个算法有着更好的时间复杂度；最坏匹配发产生了最大的外碎片，但是这并不代表最坏匹配法是最差的算法，最坏匹配法产生的大碎片将很容易在下次的内存申请中被分配出去。

2.5 xv6物理内存上限

xv6的物理内存上限为限制好的 `224MB` ，由定义在 `memlayout.h` 中的 `PHYSTOP` 参数设定。当增加 `PHYSTOP` 参数并重新编译，xv6的物理内存上限随之扩大。经检验，xv6的物理内存最大值可以设置为 `512MB` ，超过该值将导致xv6无法正确进入 `init` 进程。

3. 小组分工

陈荣钊：实现虚拟内存算法及测试，调研了xv6的物理内存上限机制
伍冠宇：实现栈自增、零指针保护及测试
高俊峰：实现四种内存分配方法
孙梓健：对四种内存分配方法进行测试