操作系统大作业报告

沈冠霖 黄浩鹏 邓坤恬

1.配置和运行方法

本大作业基于xv6的一个开源框架https://github.com/mit-pdos/xv6-public实现，配置方法和xv6一样。

环境：`ubuntu16.04`

需求软件：qemu，make，gcc

运行方法：在代码根目录下输入命令`make qemu`即可进入qemu环境

之后输入`./MemoryInfoTest` 即可进行获取内存信息测试

`./ZeroPointerProtectionTest` 进行零指针保护测试

`./CopyOnWriteTest` 进行写复制测试

`./StackAutoGrowTest` 栈自增测试

`./SharedMemoryTest` 共享内存测试

`./VirtualMemoryTest` 虚拟页式存储测试

2.实现情况说明

2.1 获取全局和进程内存信息

xv6写死了224MB的物理内存，还给每个进程2G的虚拟内存，我们还实现了共享内存。但是，你无法查看实际的内存分配信息。因此，我们参考linux的`vmstat`指令，实现了一个系统调用`GetMemoryInfo`，来获取全局的内存信息和各个进程的内存信息。

2.1.1 实现原理

根据xv6内存分配的特点，我们选择返回全局的物理内存分配情况（在`kalloc，kfree`时更新），全局的共享内存分配情况（遍历全局共享内存数组获取），还有各个进程的物理内存情况和外部内存情况（分别由进程的内存分配表，交换表的情况获取）还有共享内存情况（遍历进程的共享内存表更新）。

我们将这些操作封装成一个系统调用`GetMemoryInfo`。因为时间不足，我们没有时间深入研究xv6系统调用，只能模拟其他系统调用，以char数组作为参数。这就要求我们在内核中进行int到char的转换，再在调用的时候转换回去。

2.1.2 测试方法

见`MemoryInfoTest.c`文件。我们先读取当前内存信息，然后分配了1块共享内存和48k的一般内存后再次读取，发现物理内存增长了48+4=52k，有一个进程的共享内存增加了4k，物理内存增加了48k，说明读取信息正确。

2.2 零指针保护

零指针指指向地址为0的指针。在C语言中，指向地址为0的指针是有效的，会访问0地址，这个地址是否有效则由平台和操作系统规定。在xv6中零地址却是可用的。但在很多其他语言中（如C++语言）指向地址为0的指针会被默认为空指针，即零指针也相当于指向NULL，这队编程是有意义的，可以防止因为编程的失误造成的对0地址的访问。因此我们计划实现了零指针保护的功能。

2.2.1 实现原理

我们通过将用户的进程加载的起点改至原本页面的第二个页面，将第一个页面空出来，并将其映射从页表中去除来实现零指针保护。原理是在用户访问零地址的时候会访问第一个页面，会触发缺页中断，根据这个缺页中断信号可以判断出用户访问了零指针，然后选择将进程杀死来达成零指针保护的目的。

具体实现是在exec函数中完成elf文件读取和装载进内存的过程时，修改装载的起点，使得程序的加载会在第二个页面开始，但此时整体内存偏移了一个页面，为了使得程序正常运行，同时修改makefile使其在编译程序时将所有的地址加上一个页面的偏移，这样就能在不对用户使用产生任何可见影响的情况下将第一个页面置空，实现零指针保护。

2.2.2 测试方法

零指针保护的实现可通过文件`ZeroPointerProtectionTest.c`文件实现，我们通过直接访问0地址进行测试，若进程被杀死，即测试成功。

2.3 写时复制

写时复制是一个对提高操作系统性能非常重要的功能。用户程序调用 `fork` 之后，进程会有自己的虚拟地址空间。
然而，在原版 xv6 中没有实现写时复制，会让每次 `fork` 都将进程的所有数据复制一遍。这会带来 `fork` 的
巨大延迟。实际上，`fork` 时很多地址空间是可以不必复制的，例如代码段，也有很多地址空间可以等到修改的时候再拷贝，例如各个堆栈数据段。

写时复制可以延缓复制操作的时间，极大地提高了 `fork` 的运行效率。

2.3.1 实现原理

我们使用了一个全局的数组来记录每个物理页正在被几个进程引用。当一个页面只被一个进程使用时，这个进程拥有正
常的页读写权限。而在 `fork` 发生时，我们会找到这个进程拥有的所有页面，把他们的引用加一，并抹去写权限。
当一个进程试图写一个页面时，会导致 `trap` 发生。在异常中断服务程序 `pagefault` 中，我们判断是否为一
个页面被多个程序占用，如果是，说明我们应该做写时复制了。这时把页面复制一份到新的页面，打开写权限，并将其
提供给试图写的进程。退出异常服务程序，就可以让进程正常的写了。

2.3.2 测试方法

见 `CopyOnWriteTest.c` 文件。在这个测试中，我们首先开了一个 global 数据 `sharedData`，然后进行 `fork`，在子进程中修改该数据。修改后，父子进程各对自己的数据进行打印，可以发现一个进程的修改不会影响
到另一个进程的使用。

2.4 栈自增

在 xv6 中，一个进程拥有两个栈——系统栈和用户栈。为了简单起见，xv6写死了这两个栈的大小，均为一个页面
（4KB）。栈在函数调用时极为有用，系统栈用于进行系统调用，大部分情况下4KB足够了；然而用户程序是野蛮的，4KB很容易无法满足程序的需求。栈自增可以解决这个问题，极大地提高系统的服务能力。

2.4.1 实现原理

我们把用户栈放在用户地址空间的最顶端，向下生长。通过维护 `stackSize` 和 `sz` 两个变量，可以明确栈目
前的大小和用户进程在下面使用的空间大小。我们保持栈顶和堆顶至少相差一个页面，提供余量。若用户栈的使用超过
了 `stackSize` 引发缺页终端，即表明我们应该增长栈了。这时候增栈即可。

2.4.2 测试方法

见 `StackAutoGrowTest.c` 文件。在这个测试中，我们递归调用一个函数，这个函数会在栈上开一个 1024 大
小的 int 数组，即每递归一次就开至少一个页的大小在栈上。然后递归 256 次（这个数随意，只要不让栈和堆相交
就行），跑通就说明可以正常运行了。

在原版 xv6 中，这程序递归一层都直接崩掉。

2.5 进程间共享内存

进程间共享内存是进程间常用的高级通信方式。我们使用类似《操作系统》课上学习到的，文件管理中的打开文件表的方式，来实现共享内存。

2.5.1 实现原理

首先，系统初始化的时候，全局会分配一个固定长度为`SHARED_MEMORY_GLOBAL`的定长表，类比文件系统的打开文件表。这个表每一页用不同的签名`Signature`来标识不同的内存块，还存储着对应的虚拟地址和共享计数。

其次，每个进程有一个长度为`SHARED_MEMORY_PER_PROC`的定长表，存储该进程的共享内存，对应文件系统的进程打开文件表。进程可以通过系统调用`AllocSharedMemory`，通过唯一的Signature来分配共享内存。如果这个内存在全局存在，就会直接在进程中记录，并将共享计数+1；否则就会在全局分配一块共享计数为1的内存。进程也可以通过系统调用`DeallocSharedMemory`来回收共享内存，全局会减少共享计数，在共享计数为0的时候会在全局回收这块内存。

最后，每个进程可以通过系统调用`ReadSharedMemory/WriteSharedMemory`来对这块内存进行读写操作。

2.5.2 测试方法

我们在`SharedMemoryTest.c`里进行了测试。测试流程如下：进程1分配一块共享内存并写入，之后切换到进程2,进程2会读取这块内存，并且写入，然后释放。之后进程1被激活，读取这块内存并释放。如果进程2读取的内容和进程1先写入的一致，而且进程2写入的内容和进程1之后读取的一致，说明操作基本正确。

2.6 虚拟页式存储

xv6 中一共只有 224MB 的物理内存，但是其每个进程的虚拟内存空间有2GB。实现虚拟页式存储可以大大增加每个进程可以访问的内存数目。为了简化问题，我们只对每个进程本身进行内存的置换，也就是每个进程新访问/分配的页面只会置换出这个进程本身优先级最低的页面。

2.6.1 实现原理

首先，我们对每个进程维护了一个内存表`MemoryTable`和交换表`SwapTable`，分别用来存储这个进程在内存中和外存中的虚拟页面地址。因为内存表和交换表也会占用物理内存，因此，我们并没有简单地用一个Table元素来记录一个物理页，而是每个table里记录成百上千个物理页，使得每个table的大小接近4096字节。因为一个table元素充其量两三个指针，也就十几字节，而内存表和交换表要动态维护，每次分配一个内存表/交换表都至少要开一个页面4096字节，为了节省物理内存，一个table存储多个物理页更加合理。

其次，我们使用FIFO算法进行页面置换。首先，这种算法最为简单，鲁棒性最好；其次，LRU等算法需要硬件支持，不然只有软件算法的话，开销过大；实现FIFO算法的方法很简单，我们将内存表中非空的元素链接成一个链表（这个几乎不需要额外空间，只需要头尾），每次添加新元素都添加在表头，这样表尾就是优先级最低的，最先被交换出内存。

最终，我们利用文件系统的文件读写功能来实现内外存交换：在每个进程初始化的时候分配固定数目的文件，专门用于记录在外存的进程内存。

在内存分配的时候，我们会判定当前进程在内存中的内存大小是否过大，如果过大，我们会将内存链表尾元素写回外存，再记录新的内存，否则就直接记录新的内存。

根据《操作系统》课程讲解的原理，我们在缺页中断处理函数实现了判断`PTE_PG`（驻留位）的操作，如果当前访问的地址在进程内存中，但是实际在外存中存储，我们就会调用置换算法，将这个地址置换回内存，并且更新页表和内存表，交换表。

2.6.2 测试方法

我们在`VirtualMemoryTest.c`里进行了测试。我们通过反复在一个进程中使用malloc函数分配内存来达到xv6初始的内存极限，并且对新分配的内存进行访问来尽可能触发缺页中断机制。我们通过在换页函数中加入输出来确定程序的确实现了换页机制，而且程序能正确运行，这就说明这个机制实现正确。

3.分工

沈冠霖负责虚拟页式存储，进程内共享内存两部分及其测试，以及读取这两部分的内存信息实现。

黄浩鹏负责栈自增，写复制两部分的实现和测试。

邓坤恬负责零指针保护的实现和测试，以及读取其他部分的内存信息的实现及测试。