**Lab 5: Memory management**

**This is a group lab. You can make groups of three students.**

**Objectives**

* Modify memory layout to move stack to top of address space (70%)
* Implement stack growth (30%)

修改内存布局，将堆栈移动到地址空间的顶部

实现堆栈增长

**Part 1: Changing memory layout**

**Overview**

In this part, you'll be making changes to the xv6 memory layout. Sound simple? Well, there are a few tricky details.

**Details**

In xv6, the VM system uses a simple two-level page table. You may find the description in Chapter 1 of the xv6 manual sufficient (and more relevant to the assignment).

The xv6 address space is currently set up like this:

code

stack (fixed-sized, one page)

heap (grows towards the high-end of the address space)

In this part of the xv6 project, you'll rearrange the address space to look more like Linux:

code

heap (grows towards the high-end of the address space)

... (gap)

stack (at end of address space; grows backwards)

You can see the general map of the kernel memory in memlayout.h; the user memory starts at 0 and goes up to KERNBASE. **Note that we will not be changing the kernel memory layout at all, only the user memory layout**

Right now, the program memory map is determined by how we load the program into memory and set up the page table (so that they are pointing to the right physical pages). This is all implemented in exec.c as part of the exec system call using the underlying support provided to implement virtual memory in vm.c. To change the memory layout, you have to change the exec code to load the program and allocate the stack in the new way that we want.

Moving the stack up will give us space to allow it to grow, but it complicates a few things. For example, right now xv6 keeps track of the end of the virtual address space using one value (sz). Now you have to keep more information potentially e.g., the end of the bottom part of the user memory (i.e., the top of the heap, which is called brk in un\*x), and bottom page of the stack.

Once you figure out in exec.c where xv6 allocates and initializes the user stack; then, you'll have to figure out how to change that to use a page at the high-end of the xv6 user address space, instead of one between the code and heap.

Some tricky parts: Let me re-emphasize: one thing you'll have to be very careful with is how xv6 currently tracks the size of a process's address space (currently with the sz field in the proc struct). There are a number of places in the code where this is used (e.g., to check whether an argument passed into the kernel is valid; to copy the address space). We recommend keeping this field to track the size of the code and heap, but doing some other accounting to track the stack, and changing all relevant code (i.e., that used to deal with sz ) to now work with your new accounting. Note that this potentially includes the shared memory code that you are writing for part 2.

**Part 2: Growing the Stack**

The final item, which is challenging: automatically growing the stack backwards when needed. Getting this to work will make you into a kernel boss, and also get you those last 10% of credit. Briefly, here is what you need to do. When the stack grows beyond its allocated page(s) it will cause a page fault because it is accessing an unmapped page. If you look in traps.h, this trap is T\_PGFLT which is currently not handled in our trap handler in trap.c. This means that it goes to the default handling of unknown traps, and causes a kernel panic.

So, the first step is to add a case in trap to handle page faults. For now, your trap handler should simply check if the page fault was caused by an access to the page right under the current top of the stack. If this is the case, we allocate and map the page, and we are done. If the page fault is caused by a different address, we can go to the default handler and do a kernel panic like we did before.

Bonus (5%): Write code to try and get the stack to grow into the heap. Were you able to? If not explain why in detail showing the relevant code.

**Hints**

\*IMPORTANT\* Check the [help file](help-lab5.txt) posted to walk you through this assignment.

**Particularly useful for this project:** Chapter 1 of xv6 + anything else about fork() and exec(), as well as virtual memory.

Take a look at the exec code in exec.c which loads a program into memory. It will be using VM functions from vm.c such as allocuvm (which uses mappages). These will be very instructive for implementing shm\_open (in lab 4) -- we are allocating a new page the first time (similar to allocuvm) and adding it to the page table (similar to mappages).

It may be helpful to try to answer these questions to yourself:

* Read chapter 2 in the xv6 book. Briefly explain the operation of allocuvm() and mappages() and Figure 1-2. Check how exec uses them for an idea.
* Explain how you would given a virtual address figure out the physical address if the page is mapped otherwise return an error. In other words, how would you find the page table entry and check if its valid, and how would you use it to find the physical address.
* Find where in the code we can figure out the location of the stack.

从任务:

现在，程序的内存映射取决于我们如何将程序加载到内存中并设置页表(以便它们指向正确的物理页)。这都是在exec.c中实现的，作为exec系统调用的一部分，使用vm.c中提供的底层支持来实现虚拟内存。要更改内存布局，您必须更改exec代码以以我们想要的新方式加载程序并分配堆栈。

具体来说，首先打开exec.c检查执行系统调用的exec(…)函数。Exec执行以下操作:

第1部分。

打开可执行文件并解析它。这一段的其余部分仅供参考。通常，可执行文件由一个包含信息的头组成，这些信息允许我们索引文件的其余部分。之后的文件由部分组成，包括代码、全局数据，有时还有其他部分，如未初始化的数据。这些是我们需要从可执行文件中初始化的内存部分。头信息包括节的数量、文件中每个节的开始、它在虚拟内存中的映射位置以及每个节的长度。

第2部分。

使用setupkvm()初始化内核内存，它将内核的页面映射到进程地址空间。我们不需要知道这里发生了什么。

第3部分。

然后，它继续使用loaduvm()将可执行文件的各个节加载到内存中，loaduvm()为每个节创建内存页，并将它们映射到地址空间(通过初始化页表指针——详细信息稍后介绍)。xv6中的这些部分从vva 0开始加载，并一直向上加载。每个新部分都从一个新页面的开始开始。回想一下，section包括代码、全局/静态数据等。

然后，我们可以方便地跟踪用户地址空间的结束位置，这也使用一个值定义了进程的大小(proc->sz)。因此，当我们映射新页面时，sz(向上四舍五入到下一页)可以作为它们的虚拟地址，因为我们只是按顺序填充地址空间。

像allocuvm()这样的vm.c函数通常会在此之后调用映射函数，该映射函数用于初始化将虚拟地址映射到它刚刚分配的物理页的页表项。否则这些我们分配的物理帧就不能被我们的进程使用。

第4部分。

现在，我们加载了代码和数据段，该创建堆栈了。xv6目前不支持堆(如果您注意到的话，用户程序没有malloc()/free()可用)。它目前将栈映射到其虚拟地址空间中的一个页面，该页面正好在我们从可执行文件加载的最后一页之后(例如，在sz向上舍入到下一页边界)。

由于堆栈向下增长，在这里分配一个页面意味着没有空间来增长堆栈——当它向下增长时，它将运行到代码/数据。为了防止这种情况发生，xv6添加了一个页缓冲区，并将其标记为不可读，以便在堆栈增长的情况下，我们得到一个内存错误，并可以停止程序。创建堆栈的代码是:

**//** round sz up to the next page boundary since stack must start**//** **in** a new pagesz **=** PGROUNDUP(sz);

**//** our first introduction to allocuvm; it allocates **and** maps two pages**if**((sz **=** allocuvm(pgdir, sz, sz **+** 2**\***PGSIZE)) **==** 0)

goto bad;

**//** we clear the PTE **for** the first page to create a buffer page**//** between stack **and** code**/**dataclearpteu(pgdir, (char**\***)(sz **-** 2**\***PGSIZE));

待办事项1:这是我们需要更改的代码部分，以移动堆栈。当前代码调用allocuvm来创建两个页面，一个用于堆栈，另一个作为从VA sz开始的保护页面，它就在代码和数据之后。然后清除保护页的页表条目。

我们希望从用户地址空间的顶部开始定位堆栈，以便给它空间来增长。为了理解我们需要做什么，让我们看看allocuvm。

它有3个参数:

1. 页表(pgdir)。这不会改变

2. 我们正在映射的第一个页面的虚拟地址——这需要更改为指向内存中用户部分的顶部页面(就在KERNBASE下面)。如果您使用kernbase，您将尝试将页面映射到内核地址空间。

3.我们正在映射的最后一个页面的虚拟地址。对于我们来说，我们创建的堆栈只有一个页面，这样就可以在同一页面中创建一个比第一个地址稍大的地址。

allocuvm分配页面，并将其映射到页表。所以，基本上我们已经完成了移动栈的操作，只需要将这些参数更改为正确的值。然而，还有一些未解决的问题需要解决。

第5部分。

最后，我们初始化堆栈指针，当前指向sz。

待办事项2:你必须把它改成堆栈页面中最上面的单词的地址。注意KERNBASE是内核地址空间的第一个单词，所以这个单词就在它的正下方。

按照linux/x86栈约定，我们继续使用main的参数初始化栈。(目前细节对我们不重要。)

宽松的结束/其他变化。

现在我们移动了堆栈，内核中硬编码堆栈之前位置的几个地方必须更改。这些包括:

待办事项3:所有在sycall .c(和sysfile.c)中定义的用于访问用户堆栈的函数都要进行一些检查，看看地址是否确实在堆栈上。这些检查将地址与sz进行比较，因为sz在旧的实现中是堆栈的顶部。您必须更改这些检查(或者如果更容易的话删除它们)。检查所有的访问器函数，如argint, argstr, argptr, argfd等…

任务4:copyuvm()。这个函数用作fork()的一部分，用于创建父进程地址空间的副本，并将fork调用给子进程。在vm.c中实现.

如果查看这个函数，就会发现它是一个很大的for循环，遍历虚拟地址空间并逐个复制页面。循环开始于:

**for**(i **=** 0; i **<** sz; i **+=** PGSIZE){

因为它假设虚拟地址空间从0开始到sz。现在，这必须改变，以考虑到新的堆栈。

如果我们深入观察，它会读取页表以获得该页的PTE，分配新的物理帧，并将该页从父内存复制到新的页。最后，通过向其页表添加PTE，它使用映射将这个新副本映射到子地址空间。

我们如何改变它?现在sz跟踪地址空间的底部部分，所以可以不去管这个循环。我们必须跟踪堆栈的大小，并添加另一个循环来遍历堆栈页并做同样的事情(kmallocs为每个页，memmoves创建一个来自父页的副本，然后mapages()将其添加到页表)。

除了遍历的虚拟地址范围之外，循环将非常相似。在添加堆栈增长之前，该堆栈只有一个页面，但随着堆栈增长，我们需要跟踪堆栈页面的数量。为了做好准备，我们需要在struct proc中添加一个变量来跟踪堆栈的大小(以页或字节为单位——都可以，但我建议使用页)。这个计数器从一页的堆栈开始;在exec()中设置。

调试:如果堆栈移动正确，xv6将能够成功引导到shell。

如果你没有正确地分配/映射堆栈，你会在分配函数中得到错误(例如，重新映射错误)，或者在你的程序运行时得到错误(页面错误)。

如果你不注意所有的argint()等。一些系统调用将不能正确传递参数。结果可能会很奇怪。例如，printf不会打印，wait也不会等待(导致init继续fork进程，等等…)

日益增长的堆栈。

现在我们的堆栈已经被移动，我们有空间来增加它。当a程序导致堆栈增长到大于一个页的偏移量时，此时，我们将访问一个没有被分配/映射的页。这将导致页面错误。基本上，我们将捕获到trap.c中的捕获处理程序

其中有一个switch语句，对每个支持的trap都有一个case。我们需要为页面错误添加一个案例。这个页面错误的trap编号为14(或T\_PGFLT)，在trap .h中定义。

待办事项5:为页面错误添加一个case。当出现页面错误时，您可以在一个叫CR2的硬件寄存器中检查导致页面错误的地址。CR寄存器(控制寄存器)记录重要的硬件状态信息。可以使用函数rcr2()读取CR2寄存器。

一旦你有了错误的地址，接下来我们需要检查它是否来自当前堆栈底部的页面。如果是，我们需要增加堆栈。您可以再次使用allocuvm，但必须使用正确的参数初始化它，以便在正确的位置分配一个页面。之后，您可以增加堆栈大小计数器，从而完成陷阱处理程序。

瞧!你应该可以走了。

要检查堆栈是否正确增长，请编写一个递归程序，使其嵌入足够深，以获得大于1页的堆栈。如果您的实现工作正常，您应该得到一个页面错误，并正确地增长堆栈。