

Lab3 Survival guide!

From assignment:

"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."

Specifically, lets start by opening up `exec.c` check the `exec(...)` function which implements the system call. `Exec` does the following:

Part 1. Opens the executable file and parses it. The rest of this paragraph is FYI. Typically executable files are made up of a header including information that allows us to index the rest of the file. The file after that consists of sections including the code, the global data, and sometimes other sections like uninitialized data. These are the parts of the memory that we need to initialize from the executable. The header information includes the number of sections, the start of each section in the file, and where it maps to in virtual memory, and the length of each section.

Part 2. Initializes the kernel memory using `setupkvm()` which maps the pages of the kernel to the process address space. We dont really need to know what happens in here.

Part 3. It then moves on to load the sections of the executable file into memory using `loadvm()` which creates the memory pages for each section and maps them to the address space (by initializing the page table pointers -- more details in a bit). These sections in `xv6` are loaded starting at VA 0, and going up. Each new section starts at the beginning of a new page. Recall that sections include code, global/static data, etc..

**Conveniently, we can then keep track of where the user address space ends, which also defines the size of the process using one value (`proc->sz`). So, as we map new pages, `sz` (rounded up to the next page) can serve as their virtual address since we are simply filling in the address space sequentially.

In VM, we map virtual pages to physical frames. `XV6` has no swap so all memory pages have to be in physical memory. Physical memory is allocated by the kernel allocator `kmalloc`. If you want to dig deeper (you need to for the bonus part) you will see how these pages are initialized using `kinit()` and `kinit2()` which are called during the boot process in `main.c`. As a result, we use `kmalloc()` as we request each new page inside `vm.c` to allocate a new physical page.

The `vm.c` functions such as `allocvm()` typically follow this up with a call to `mappages` which is used to initialize the page table entries mapping the virtual address to the physical page that it just allocated. Otherwise these physical frames that we allocate cannot be used by our process.

Part 4. At this point, we loaded code and data sections, and its time to create the stack. **xv6 does not support a heap at the moment** (there is no `malloc()/free()` available to user programs if you noticed). It currently maps the stack in its virtual address space at a page right after the last page we loaded from the executable (i.e., at `sz` rounded up to the next page boundary).

Since the stack grows down, allocating a page here means there is no

room to grow the stack -- as it grows down, it will run into the code/data. To protect against that, xv6 adds one page buffer and marks it as unreadable so that in case the stack grows, we get a memory error and can stop the program. The code to create the stack is:

clearpteu() is used to create an inaccessible page beneath the user stack.

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

if((sz = allocvm(pgdir, sz, sz + 2*PGSIZE)) == 0) // our first introduction to allocvm;
it allocates and maps two pages
    goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE)); //we clear the PTE for the first page to create a
buffer page between stack and code/data
```

TODO 1: This is the part of the code that we need to change to move the stack. The current code calls allocvm to create two pages, one for the stack and one as a guard page starting at VA sz which is right after the code and data. It then clears the page table entry for the guard page.

Accessing guard page results in page fault.

We want to locate the stack starting at the top of the user address space to give it room to grow. To understand what we need to do, let's look at allocvm.

It takes 3 parameters:

a. The page table (pgdir). This will not change

b. The virtual address of the first page we are mapping -- this needs to change to point to the top page of the user part of memory (right under KERNBASE). If you use kernbase, you will try to map the page to the kernel address space.

c. The virtual address of the last page we are mapping. For us, we are creating a stack with only a single page, so this can be another address in the same page, slightly bigger than the first address.

Allocvm allocates the page, and maps it to the page table. So, basically we are done with moving the stack by just changing these parameters to the right value. However, there are a few loose ends to tie up.

Part 5: Finally, we initialize the stack pointer, currently to sz.

TODO 2: you will have to change this to the address of the top word in the stack page. Note that KERNBASE is the first word in the kernel address space, so this is the word right under that.

We proceed to initialize the stack with the parameters for main as per the linux/x86 stack convention. The details are not important to us for now.

Loose Ends/Other changes.

Now that we moved the stack, a few places in the Kernel that hard coded the previous location of the stack have to be changed. These include:

TODO 3: All of the functions that are defined in syscall.c (and sysfile.c) for accessing the user stack have some checks to see if the addresses are indeed on the stack. These checks compare the address against sz since that was the top of the stack in the old implementation. You have to change those checks (or remove them if it is easier). Check all the accessor functions such as argint, argstr, argptr, argfd,

etc...

TODO 4: `copyvm()`. This function is used as part of `fork()` to create a copy of the address space of the parent process calling `fork` to the child process. It is implemented in `vm.c` Hence, we also modify `fork()` in `exec.c` to copy `stackPages` from parent process to child process.

If you look at this function, it is one big for loop that iterates over the virtual address space and copies the pages one by one. The loop starts with:

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

since it assumes the virtual address space starts at 0 and goes to `sz`. Now this has to be changed to take into account the new stack.

If we look deeper, it reads the page table to get the PTE for the page, allocates a new physical frames, and copies the page from the parent memory to the new page. Finally it uses `mappages` to map this new copy to the child address space by adding a PTE to its page table.

How do we change it? Now `sz` tracks the bottom part of the address space, so its ok to leave that loop alone. We have to keep track of the size of the stack, and added another loop that iterates over the stack page(s) and does the same thing (`kmallocs` a page for each one, `memmoves` to create a copy from the parent, and then `mapages()` to add it to page table).

The loop will be very similar, with the exception of the virtual address ranges that iterates over. Before we add stack growth, the stack is only one page, but as the stack grows we need to keep track of the number of stack pages. To prepare for this, we need to add a variable in `struct proc` to keep track of the size of the stack (in pages or bytes--either is fine, but I recommend pages). This counter starts out with a stack of one page; set it in `exec()`.

Debugging: If your stack moved correctly, `xv6` will be able to boot into shell successfully.

If you dont allocate/map the stack correctly, you will get errors either in the allocation functions (e.g., remapping errors) or as your program runs (page faults).

If you dont take care of all the `argint()` etc.. functions some of your system calls will not be able to pass parameters correctly. The results could be weird. For example, `printf` won't print, and `wait` wont wait (leading to `init` continuing to fork processes, etc...)

Growing the stack

Now that our stack has been moved, we have room to grow it. When the a program causes the stack to grow to an offset bigger than one page, at this point, we will be accessing a page that is not allocated/mapped. This will cause a page fault. Basically, we will trap to the trap handler in `trap.c`

In there there is a switch statement with a case for every supported trap. We need to add a case for page faults. This page fault has trap number 14 (or `T_PGFLT`) as defined in `traps.h`

TODO 5: Add a case for the page fault. When a page fault occurs, you can check the address that caused the page fault in a hardware register called `CR2`. The `CR` register (standing for Control Register) keep track of important hardware state information. You can read the `CR2` register using the function `rcr2()`.

Once you have the offending address, next we need to check if it is from the page right under the current bottom of the stack. If it is, we need to grow the stack. You can use `allocvm` again, but you have to initialize it with the right parameters to allocate one page at the right place. After that, you can increment your stack size counter, which finishes your trap handler.

Voila! you should be good to go.

To check if the stack grows correctly, write a recursive program that nests deep enough to get a stack longer than 1 page. You should get a page fault and grow the stack correctly if your implementation works.