ework: xv6 lazy page allocation

Submit your solutions before the beginning of the next lecture to the submission web site.

One of the many neat tricks an O/S can play with page table hardware is lazy allocation of heap memory. Xv6 applications ask the kernel for heap memory using the sbrk() system call. In the kernel we've given you, sbrk() allocates physical memory and maps it into the process's virtual address space. There are programs that allocate memory but never use it, for example to implement large sparse arrays. Sophisticated kernels delay allocation of each page of memory until the application tries to use that page -- as signaled by a page fault. You'll add this lazy allocation feature to xv6 in this exercise.

Part One: Eliminate allocation from sbrk()

Your first task is to delete page allocation from the sbrk(n) system call implementation, which is function sys_sbrk() in sysproc.c. The sbrk(n) system call grows the process's memory size by n bytes, and then returns the start of the newly allocated region (i.e., the old size). Your new sbrk(n) should just increment the process's size (proc->sz) by n and return the old size. It should not allocate memory -- so you should delete the call to growproc().

Try to guess what the result of this modification will be: what will break?

Make this modification, boot xv6, and type `echo hi` to the shell. You should see something like this:

```
init: starting sh
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x12f1 addr 0x4004--kill proc
$
```

The "pid 3 sh: trap..." message is from the kernel trap handler in trap.c; it has caught a page fault (trap 14, or T_PGFLT), which the xv6 kernel does not know how to handle. Make sure you understand why this page fault occurs. The "addr 0x4004" indicates that the virtual address that caused the page fault is 0x4004.

Part Two: Lazy allocation

Modify the code in trap.c to respond to a page fault from user space by mapping a newly-allocated page of physical memory at the faulting address, and then returning back to user space to let the process continue executing. You should add your code just before the `cprintf` call that produced the "pid 3 sh: trap 14" message. Your code is not required cover all corner cases and error situations; it just needs to be good enough to let sh run simple commands like echo and ls.

Hint: look at the cprintf arguments to see how to find the virtual address that caused the page fault.

Hint: steal code from allocuvm() in vm.c, which is what sbrk() calls (via growproc()).

Hint: use PGROUNDDOWN(va) to round the faulting virtual address down to a page boundary.

Hint: `break` or `return` in order to avoid the `cprintf` and the `proc->killed = 1`.

Hint: you'll need to call mappages(). In order to do this you'll need to delete the `static` in the declaration of mappages() in vm.c, and you'll need to declare mappages() in trap.c. Add this declaration to trap.c before any call to `mappages()`:

```
int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
```

Hint: you can check whether a fault is a page fault by checking if `tf->trapno` is equal to `T_PGFLT` in trap().

If all goes well, your lazy allocation code should result in `echo hi` working. You should get at least one page fault (and thus lazy allocation) in the shell, and perhaps two.

By the way, this is not a fully correct implementation. See the challenges below for a list of problems we're aware of.

Challenges: Handle negative sbrk() arguments. Handle error cases such as sbrk() arguments that are too large. Verify that fork() and exit() work even if some sbrk()'d address have no memory allocated for them. Correctly handle faults on the invalid page below the stack. Make sure that kernel use of not-yet-allocated user addresses works -- for example, if a program passes an sbrk()-allocated address to read().

Submit: The code that you added to trap.c in a file named *hwN.c* where *N* is the homework number as listed on the schedule.