Homework 4: Lazy Memory Allocation

As we discussed in class, one of the many cool *tricks* that paging hardware can be used for is *lazy allocation* -- only giving out physical memory when it's used, rather than when it's allocated. In this assignment, we'll modify xv6 to use a lazy allocator. We won't cover every corner case, but by the end of it we will have a basic working implementation.

Getting the Code from Github

As with last time, we'll be working off of a slightly modified version of xv6. The major difference is that the `cat` program has been modified so that it uses dynamic memory allocation rather than static allocation, which will allow us to uncover an edge case in a naive implementation.

If you still have your xv6 directory from last time, remove or rename it first. Then get the base xv6 code for this assignment:

```
$ git clone https://github.com/moyix/xv6-public.git
Cloning into 'xv6-public'...
remote: Counting objects: 4501, done.
remote: Compressing objects: 100% (17/17), done.
remote: Total 4501 (delta 3), reused 0 (delta 0), pack-reused 4484
Receiving objects: 100% (4501/4501), 11.68 MiB | 3.46 MiB/s, done.
Resolving deltas: 100% (1800/1800), done.
Checking connectivity... done.
$ cd xv6-public/
$ git checkout hw5
Branch hw5 set up to track remote branch hw5 from origin.
Switched to a new branch 'hw5'
```

Part 1: Use the debugger

In order to get acquanted with the debugger, we will start with debugging XV6. XV6 has a Makefile rule to make debugging simple, however you will need to have two open command line windows. Type the following commands on each:

- Command Line 1: $ make qemu-gdb
- Command Line 2: $ gdbtui kernel

Command line 1 is going to basically create a debuggeable version of XV6. Command line 2, is going to start the debugger.

Now we will setup a breakpoint in the system call that allocates memory. The call that user-process use in xv6 to allocate memory is called `sbrk()`; it's kernel-mode implementation is `sys_sbrk()` in `sysproc.c`. Set a breakpoint in this function. In order to do that, you might need a gdb cheatsheet like the one in NYU classes. Then answer the following questions when you submit your patch:

1. Run a user process like: 'ls'
2. What is the size of `n` when your breakpoint gets hit?
3. Print out the stackframe (backtrace) for this call.

Part 2: Removing Eager Allocation

Change `sys_sbrk()` so that it just adjusts `proc->sz` and returns the old `proc->sz` (i.e., remove the call to `growproc()`).

After rebuilding xv6, try running a command like `echo hello`:

```
init: starting sh
$ echo hello
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x12bb addr 0x4004--kill proc
```

What went wrong, and why? (Answering this question is not part of the assignment, just think about it).

Part 3: Implementing Lazy Allocation

The message above comes from the `trap()` function in `trap.c`. It indicates that an exception (number 14, which corresponds to a page fault on x86) was raised by the CPU; it happened when `eip` (the program counter) was `0x12bb`, and the faulting address was `0x4004`. It then kills the process by setting `proc->killed` to 1. Can you find in the code in trap.c where that happened? (Answering this question is not part of the assignment either, just getting you familiar with the code)

Add code to `trap()` to recognize this particular exception and allocate and map the page on-demand.

Once you have implemented the allocation, try running `echo hello` again. It should work normally.

Hints:

1. The constant `T_PGFLT`, defined in `traps.h`, corresponds to the exception number for a page fault.
2. The virtual address of the address that triggered the fault is available in the `cr2` register; xv6 provides the `rcr2()` function to read its value.
3. Look at the code in `allocuvm()` to see how to allocate a page of memory and map it to a specific user address.
4. Remember that the first access might be in the middle of the page, and so you'll have to round down to the nearest PGSIZE bytes. xv6 has a handy function called `PGROUNDDOWN` you can use for this.
5. You will need to use the `mappages()` function, which is declared as `static`, meaning it can't be seen from other C files. You'll need to make it non-static, and then add its prototype to some header file that is included by both `trap.c` and `vm.c` (`defs.h` is a good choice).

Part 4: Handling Some Edge Cases

Although simple commands like `echo` work now, there are still some things that are broken. For example, try running `cat README`. Depending on how you implemented Part 2, you may see:

```
init: starting sh
$ cat README
unexpected trap 14 from cpu 1 eip 80105282 (cr2=0x3000)
cpu1: panic: trap
 8010683a 801064fa 80101f4e 80101174 80105725 8010556a 801066f9 801064fa 0 0
```

Why is this happening? Debug the problem and find a fix (if this part works already, you don't need to make any changes).

Next, let's see if pipes work, by running `cat README | grep the`

```
$ cat README | grep the
cpu0: panic: copyuvm: page not present
 8010828e 80104693 8010630b 8010556a 801066f9 801064fa 0 0 0 0
```

Find out what's going on, and implement a fix.

Hints

1. Think about what happens if the kernel tries to read an address that hasn't been mapped yet (for example, if we use `sbrk()` to allocate some memory and then hand that buffer to the `read()` syscall). Read the code in `trap()` with that case in mind.
2. Making pipes work is a very tiny change -- just one line in one file. Don't overthink it.

Submitting

As with HW4, you will use git to create a patch.

Commit your changes:

```
$ git commit --all --message="Implement lazy allocation"
[hw5 ef751c0] Implement lazy allocation
 4 files changed, 30 insertions(+), 10 deletions(-)
```

(Note: if you added any new files, you will also have to use `git add <filename>` before you run `git commit`.)

Now create the patch file:

```
$ git format-patch hw5.unmodified
0001-Implement-lazy-allocation.patch
```

The command creates a file, `0001-Implement-lazy-allocation.patch`, containing the changes you've made. Submit this file on NYU Classes

Final Notes

Although this covers most use cases, there are still a few stray things that it doesn't handle:

1. Negative arguments to `sbrk()` should reduce the size of the process and deallocate the pages.
2. This code will allow multiple processes to collectively allocate more memory than the system has (*overcommit*).

It may be helpful, if you want to understand the material better, to think about how you could fix these issues.

Credit: Adapted from a homework by Brendan-Dolan Gavit.

Credit: Adapted from MIT's 6.828 in-class exercise xv6 lazy page allocation