PART 1: (55 points) XV6 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.

Rubric:

The following is an approximate rubric. We will take into account work on each part but if the program doesn't compile you automatically get -25

- Doesn't Compile: -40. You will get a max of 15 points on a sliding scale. We will consider partial work but it's hard to grade.
- Debugger: - 20 points if you don't do this part.
- Removing eager allocation: -30 points if you don't do this part.
- Implementing Lazy Allocation: -30 points if you don't do this part.
- Edge Cases: -20 points.
- Nothing on this part: -55 points

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.

Same as the last homework assignment we will be using the Anubis autograder, and github for submitting work. Please use this link to get your repo:

https://classroom.github.com/a/HNMaddNE

You will be using a web interface to view the status of your submissions. Enter the git commit hash, and verify your netid to see the details of your submission. To find a git commit hash, you can either use the command line, or github itself. *Once you have pushed to your github repo*, you can use `git rev-parse master` to see the most recent commit hash.

```
jc@aion ‹ master › : ~/nyu/os/os3224-assignment-3
[0] % git rev-parse master
e0b313a4574f00767a1e5d67c31372fea791d575
```

Once you have your commit hash, you can navigate to the autograder site at https://nyu.cool. Enter your hash into the search bar and verify your netid to see the status of your submission. Please keep in mind that attempting to view other students submissions will be considered an act of academic dishonesty, and will be treated accordingly.

Part A: Use the debugger

Follow the instructions on how to debug XV6 from here. Submit a screenshot of your debugger (gdb or the vscode debugger will be accepted) in a file called partA.png or partA.jpg. If you are using the class VM, or Vital, make sure your screenshot takes up the entire VM window not just the terminal.

Now we will set a breakpoint in the system call that allocates memory. The call that user-processes use in xv6 to allocate memory is called `**sbrk()**`; it's kernel-mode implementation is `**sys_sbrk()**` in `sysproc.c`. Set a breakpoint on the first line **inside** this function. Then answer the following questions when you submit your patch:

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

Part B: 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 C: 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](https://en.wikipedia.org/wiki/Function_prototype) to some header file that is included by both `trap.c` and `vm.c` (`defs.h` is a good choice).

Part D: 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.md | 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 the previous homeworks, your repo will be your submission. Push to your repo often. You have unlimited submissions until the deadline. Please use this tutorial for using git:

https://piazza.com/class/k5wsd9vc9x81es?cid=63

Use https://nyu.cool to verify that the autograder has processed your submission and that the tests have passed.

PART 2: (45 Points) Short answer questions

Put your answers to the following questions in a text file called answers.txt in your repo.

A) (15 points) Below is most of the code of the solution to the dining philosophers we saw in class:

```
void philosopher(int i)                    /* i: philosopher number, from 0 to N–1 */
{
      while (TRUE) {                       /* repeat forever */
            think( );                      /* philosopher is thinking */
            take_forks(i);                 /* acquire two forks or block */
            eat( );                        /* yum-yum, spaghetti */
            put_forks(i);                  /* put both forks back on table */
      }
}

void take_forks(int i)                     /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                        /* enter critical region */
      state[i] = HUNGRY;                   /* record fact that philosopher i is hungry */
      test(i);                             /* try to acquire 2 forks */
      up(&mutex);                          /* exit critical region */
      down(&s[i]);                         /* block if forks were not acquired */
}

void put_forks(i)                          /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                        /* enter critical region */
      state[i] = THINKING;                 /* philosopher has finished eating */
      test(LEFT);                          /* see if left neighbor can now eat */
      test(RIGHT);                         /* see if right neighbor can now eat */
      up(&mutex);                          /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

Why is the state variable set to HUNGRY in the procedure take_forks?

B) (15 points) Recall the parallel hash table implementation that we worked in class.

```
1 #define NUM_BUCKETS 5     // Buckets in hash table
2
3 typedef struct _bucket_entry {
4     int key;
5     int val;
6     struct _bucket_entry *next;
7 } bucket_entry;
8
9 bucket_entry *table[NUM_BUCKETS];
10
11 // Inserts a key-value pair into the table
12 void insert(int key, int val) {
13     int i = key % NUM_BUCKETS;
14     bucket_entry *e = (bucket_entry *) malloc(sizeof(bucket_entry));
15     if (!e) panic("No memory to allocate bucket!");
16     e->next = table[i];
17     e->key = key;
18     e->val = val;
19     table[i] = e;
20 }
```

Suppose we have two threads inserting keys using insert() at the same time.

1. (7 points) Describe the exact sequence of events (i.e, the order that the statements in insert() would have to be run by each thread) that results in a key getting lost.
2. (8 points) In an attempt to fix the problem, suppose we add code that locks the table just before line 19 and unlocks it right afterward. Would this fix the problem? Why or why not?

C) Recall that in 32-bit x86, page directories and page tables are each made up of 1024 32-bit entries. Suppose we have 4 processes on a system, each of which has 256MB worth of virtual address space mapped. No need to use a calculator here. Just show the numbers you would use and simplify as much as you can.

1. (7 points) How much memory is used to store the page directories and page tables if 4KB pages are used?
2. (8 points) If 4MB pages (super pages) are used, then the entries in the page directory point directly to the page frame (i.e., no second-level page tables are used). How much memory would be taken up by page directories in this case?