Kernel Programming Assignments

These specs are **private** (i.e., only for students who took or are taking CSCI 402 at USC). You do **not** have permissions to **display these specs** at a public place (such as anywhere on `github.com` or a public `bitbucket.org` repository). You also do **not** have permissions to **display the code** you write to implementation these specs at a public place since your code was written to implement private specs. (If a prospective employer asks you to post your code, please tell them that you do not have permissions to do so; but you can send them a **private copy**.)

Overview

(Please check out the kernel FAQ before sending your questions to the TAs, the course producers, or the instructor.)

You are provided with a source and binary release of a skeletal operating system called **weenix**. Most of the OS pieces are filled in but there are missing pieces. You are given detailed instructions for where to modify or add your code. You are expected to complete the assignments in the order listed below, because each assignment depends upon the code for the previous assignment.

- Kernel Assignment 1 (kernel threads - **PROCS**), due **11:45PM**, **10/21/2022**. Electronic submissions only.
- Kernel Assignment 2 (VFS layer - **VFS**), due **11:45PM**, **11/12/2022** **(extended)**. Electronic submissions only.
- Kernel Assignment 3 (virtual memory - **VM**), due **11:45PM**, **12/2/2022**. Electronic submissions only.

Please begin working on the assignments early: since the assignments are "cumulative", if you don't have a working version of assignment 1 (for example), you may find it difficult to complete assignment 2.

Cheating

Accessing a submission from a previous semester (e.g., you run a submission from a previous semester to see how another student's code works) is considered cheating. If it's **evident** from your submission that you have **accessed** a submission from a previous semester, I will forward your case to USC Student Judicial Affairs for cheating investigation. The standard punishment is a grade of F in the class. Since there is only one submission from a team, even if only one team member cheated, the entire team is considered cheated. Therefore, please make sure that your teammates are not cheating! Meet often with your teammates and make sure that everyone can **explain** every line of code he/she wrote.

System Prerequisites

You are required get your code to work under Ubuntu 16.04 running QEMU 2.5 (any subversion of Ubuntu 16.04 is fine). If you don't have such a system, please see the instructions on how to install Ubuntu 16.04 on a Windows or a Mac OS X machine. Please make sure that you have installed all the required software for doing the kernel assignments. If you cannot get Ubuntu 16.04 to work, the only option left is for us to create a account for you on a server we have access to that runs an OpenStack instance. The problem with this setup is that, by default, you only get a commandline interface (i.e., no graphical user interface) and it may not be easy to use.

As soon as the Kernel Assignment 1 is made available to you, you should make sure that you can compile the assignment and debug the assignment under Ubuntu 16.04. If any of these is not working right, most likely, you did not install Ubuntu 16.04 or QEMU 2.5 properly and you should contact the instructor as soon as possible!

Please do **not** waste your time on anything newer than Ubuntu 16.04. We will **not** help you to get a newer system running. We will **ONLY** grade your kernel assignments on Ubuntu 16.04 with QEMU 2.5. We will **not** grade your kernel assignments on any other systems. Also, it's best to use a Ubuntu 16.04 in a virtual machine **only** for doing our kernel assignments and don't install things you don't need into in. If you want to play with Ubuntu and install other stuff, you can install additional virtual machines.

Download and Setup

By now, you should have received an e-mail containing a link to download your personalized kernel source code. Please click on the link in it to download the kernel source code tarball (this would be referred to as the **pristine kernel code**) and save it as "`weenix-assignment-3.8.0.tar.gz`" (which should be the default file name). Please do the following in a terminal window to unpack and compile the kernel source code on your Ubuntu 16.04 machine:

```
tar xvzf weenix-assignment-3.8.0.tar.gz
cd weenix-assignment-3.8.0/weenix
make clean
make
```

During "`make`" above, if you see an error message saying something about "`grub-mkrescue`", please run the following command:

```
sudo apt-get install -y xorriso
```

By the way, it is very important that if your Ubuntu 16.04 is running inside a virtual machine on Windows, you must **NOT** run the above commands in a **shared folder**. If you run the above commands inside a shared folder, some symbolic links will not be created properly and you will not be able to compile your kernel. (Some students truly prefer to work on the kernel assignments inside the **shared folder**. If you have to do it that way, please see this kernel FAQ item about what you can do and more importantly, **what you must be careful about**. Please remember that if you try to run your code and debug your code from a shared folder, things may be unpredictable and strange things may happen. That's why it's not recommended to do your work in a shared folder.) Also, please make sure that the full path of the directory you start with does not contain any special characters (such as parenthesis and brackets) or you can break the Makefile used in **weenix**.

Now you are ready to start `weenix` for the first time. Just do:

```
./weenix -n
```

If all goes well, you should see the following at the bottom of the terminal window:

```
Not yet implemented: PROCS: bootstrap, file main/kmain.c, line 184
panic in main/kmain.c:186 bootstrap(): weenix returned to bootstrap()!!! BAD!!!

Kernel Halting.
```

Line 184 of "`kernel/main/kmain.c`" looks like:

```
NOT_YET_IMPLEMENTED("PROCS: bootstrap");
```

and it's responsible for the printout above. Your job is to **replace** all such code with your implementation of the corresponding functions. (Please do **not** write code for a particular function and not remove or comment out the corresponding call to the `NOT_YET_IMPLEMENTED()` function.) If you look at the code and comments around lines that look like the above, you will find hints regarding what you need to do. For most of these functions, you should be able to do the replacement **in-place**. For more complex functions, you may want to split your code up and add additional functions (but **put them in the same file**). You must **not add new files to the kernel**.

A copy of the complete **weenix documentation** (in PDF) is available as "`doc/latex/documentation.pdf`" in the pristine kernel tarball. A copy of the weenix documentation (in PDF) is provided here for your convenience. Please skip any section that's not relevant or applicable (e.g., section 2.2.5). Please understand that the weenix documentation came from Brown University. There may be typos and bugs in it. Also, the pristine kernel source have been modified to make things work on Ubuntu 16.04.

General Grading Considerations

Just like grading for warmup assignments, we will follow the respective **grading guidelines** when we grade the kernel assignments. Grading can only be performed on either the grader's machine or the instructor's machine. We are not permitted to grade your assignments on your machine. Please read specific grading guidelines for each of the assignments.

In the "plus points" section of the grading guidelines, there are 3 categories of tests:

1. **KASSERTs** - This is section (A) of the grading guidelines.
2. **SELF-checks** - This is the last section of the "plus points" section of the grading guidelines.
3. **PRE-CANNED tests** - Everything else in the "plus points" section of the grading guidelines.

Let's look at them in more details.

KASSERTs

For **KASSERTs**, please see section (A) of the respective grading guidelines to see where you must call **KASSERT()** in your code. In general, for each function you are suppose to implement, you need to think about what **functionality** it provides, what **pre-conditions** it must assume, and what **post-conditions** it must satisfy right before the function returns. You need to add **`KASSERT()`** statements at appropriate places to demonstrate that you understand what a function is suppose to and what the important pre-conditions and post-conditions are. In a way, these **KASSERTs** are **hints/suggestions** to you to help you to write code with the correct behavior! (Since they are "hints/suggestions", if a `KASSERT()` is specified as a "precondition" in the grading guidelines, it does not mean that it must be the first statement inside a function. It just means that it's probably a good idea to put it near the beginning of a function. Similarly, "postcondition" means near the end and "middle" means somewhere in the middle.)

Please note that a function may have several ways of returning. Some returns may correspond to error conditions. Therefore, a **post-condition** mentioned above doesn't necesarily mean that it's a post-condition for **every** possible way of returning from that function. To satisfy the requirement, you just need to add the required **KASSERT()** call near **one** of the ways of returning from that function (plus a required "conforming dbg() call" mentioned below, immediately after the KASSERT statement with the correct string argument).

For example, in the `kmutex_unlock()` function it makes sense to assert that the current thread is the mutex holder and that after waking up a new thread, the current thread is NOT the mutex holder anymore. Usually you find hints about assertions you want to use in the comments for the method, e.g. check `kmutex_unlock()`. (Please note that although `weenix` has a non-preemptive kernel, some functionalities in DRIVERS and S5FS use **kmutex** and therefore would require **kmutex** to work correctly.)

To demonstrate that your code has successfully executed these **KASSERT()** calls, you **must** make a "conforming `dbg()` call" **immediately after** the corresponding **KASSERT()** call so that the grader can see it in the terminal printout. Please understand that in order for you to get credit for these **KASSERT()** calls, the grader must run your kernel and **see** the corresponding "conforming `dbg()` call" in action (i.e., printed on the console). Therefore, you must use the **correct notation** in your "conforming `dbg()` call" to tell the grader what test would reach each such a KASSERT() call.

PRE-CANNED tests

For **PRE-CANNED tests**, please see the respective grading guidelines to see how the grader will test your code.

SELF-checks

SELF-checks is a source of confusion, so, it's explained here in greater detail. This is is a Computer Science class. In Computer Science, we **test every line of the code we write**! Clearly, for code that runs in the OS kernel, it's even more important that we test every line of code we write. Therefore, the requirement here is that you need to demonstrate that **every line of code you wrote is useful** (i.e., you do not leave any trash in your kernel). To demonstrate that, we require that **every line of code that you have written for a required function must be "traversable"** (i.e., there is a way to **execute** that line of code when running the kernel). This sounds like a lot of work. To make this more manageable, it's important that you understand how this needs to be done. So, please read it carefully before you proceed. Also, whenever you are confused about the SELF-checks requirement, please come back and read this paragraph again and remind yourself why we are doing SELF-checks.

Not every line of code you write needs to satisfy the SELF-checks requirement. To be precise, any code you wrote to replace `NOT_YET_IMPLEMENTED("PROCS: ...")` are considered code you wrote for kernel 1, any code you wrote to replace `NOT_YET_IMPLEMENTED("VFS: ...")` are considered code you wrote for kernel 2, and any code you wrote to replace `NOT_YET_IMPLEMENTED("S5FS: ...")` or `NOT_YET_IMPLEMENTED("VM: ...")` are considered code you wrote for kernel 3. These are the code you have to demonstrate to the grader that every line of code in them is not useless. Please note that the code you wrote outside of the required functions for the purpose of testing your implementation of the required functions is **excluded** from this rule (or we would end up with an infinite recursion problem). Therefore, the SEFL-checks requirement **does not apply** to the code you wrote in `initproc_run()` and the code you wrote for your kshell commands since they are there to test your kernel. The bottom line is that the code you wrote that's **only** reachable from `initproc_run()` and not from any other code is **NOT** subject to the rule of SELF-checks.

Given that you have read and understood where you must make "conforming `dbg()` calls", The next step is to tell the grader **how to reach each code sequence**. For each code sequence, you need to tell the grader which **one action the grader can perform** to cause that code sequence to get executed. Typically, this would be a **shell-level command** that the grader can type into either the **kernel shell** for kernel assignments 1 and 2 or the **user-space shell** for kernel assignment 3. Telling the grader that more than one action can execute a given sequence would be redundant since the grader only needs to know whether that code sequence can be exercised or not and **not** all the possible ways of exercise a that piece of code. (The grader also does **not** need to know how many different way there are to reach a code sequence.)

Correct labeling of "conforming **`dbg()`** calls" is very important. If you don't do it right, you may end up **losing a lot of points**. (Again, please remember that we have only one grading policy and every submission must be graded the same way.)

For example, if you have wrtten code that looks like the following in a function that's you are suppose to implement in kernel 1:

```
/* sequence1 */
if (cond1) {
    /* sequence2 */
} else {
    /* sequence3 */
}
/* sequence4 */
```

According to the requirements, you must add "conforming **`dbg()`** calls" at the **END** of `sequence2`, `sequence3`, and `sequence4` (but not `sequence1`). (I'm highlighting the word "END" in red to stress the importance of that word! Some students in previous semesters put these SELF-check "conforming **`dbg()`** calls" at the wrong places and got upset that they lost a lot of points. Please read the requirements very carefully and ask the instructor if you are not sure! It is your responsibility to make sure you follow all our requirements.) Let's say that `sequence2` is executed when you simply start the kernel and `sequence3` is executed only when you run subtest 3 (which is a PRE-CANNED test) in section (C) of the grading guidelines and there is a **separate shell-level command the grader can run**. Then you should do the following:

```
/* sequence1 */
if (cond1) {
    /* sequence2 */
    dbg(DBG_PRINT, "(GRADING1A)\n");
} else {
    /* sequence3 */
    dbg(DBG_PRINT, "(GRADING1C 3)\n");
}
/* sequence4 */
dbg(DBG_PRINT, "(GRADING1A)\n");
```

Please note that, according to the requirements, you must add a "conforming **`dbg()`** call" after `sequence4` (even if `sequence4` contains only one line of code). In this example, if either `sequence2` or `sequence3` is executed, you know for sure that `sequence4` will be executed. Therefore, you can copy either one of the "conforming `dbg()` call" that you added after `sequence2` or `sequence3` and put it after `sequence4`. In the above example, we arbitrarily choose to copy the one after `sequence2`.

If there is no way to use a separate shell-level command to run subtest 3 (which is a PRE-CANNED test) in section (C) of the grading guidelines and there is only one shell-level command to run all the subtests together in section (C) of the grading guidelines, then you should use "`(GRADING1C)\n`" and not "`(GRADING1C 3)\n`".

On the other hand, suppose that the `sequence3` cannot be exercised by either starting and exiting the kernel or running any of the PRE-CANNED tests (i.e., `cond1` is always true), then you have two choices. The first way is to write your own test code. This would be considered a test code in the **SELF-checks** category (i.e., section (E) of the grading guidelines for kernel 1). In the README file of your submission, you **must document how to invoke such additional test that you have written for the purpose of SELF-checks**. If you have multiple shell-level commands you can run for the purpose of SELF-checks and the one to execute `sequence3` is subtest #2 (again, you need to document this in the "**SELF-checks**" section (say, section (E)) of your README file), then you would need to change the above code to:

```
    /* sequence1 */
    if (cond1) {
        /* sequence2 */
        dbg(DBG_PRINT, "(GRADING1A)\n");
    } else {
        /* sequence3 */
        dbg(DBG_PRINT, "(GRADING1E 2)\n");
    }
    /* sequence4 */
    dbg(DBG_PRINT, "(GRADING1A)\n");
```

The 2nd way is to simply **remove** the "`else`" code path! This way, you won't have to write additional test code! There are 2 ways to do this! The first way is to comment out the "`else`" code path:

```
    /* sequence1 */
    if (cond1) {
        /* sequence2 */
        dbg(DBG_PRINT, "(GRADING1A)\n");
/*  } else {
        /* sequence3 */
 */
    }
    /* sequence4 */
    dbg(DBG_PRINT, "(GRADING1A)\n");
```

In this example, since `cond1` is always true, then the follow code would be equivalent (and cleaner):

```
    /* sequence1 */
    KASSERT(cond1);
    /* sequence2 */
    /* sequence4 */
    dbg(DBG_PRINT, "(GRADING1A)\n");
```

The above is the **preferred** method if `cond1` is always true.

If by running all the tests in other sections of the grading guidelines, you have exercised all your code paths, you can just say so in the "SELF-checks" section of your README file. For example, you can simply write: "By running all the tests in the above sections of the grading guidelines, all code paths are exercised." The grader will check if such a declaration is true or not. If it's true, you will get full credit for the "SELF-checks" section without writing additional tests. Please understand that this is the **preferred** way of satisfying the SELF-checks requirement.

I want to remind everyone that you must not post 3 or more lines of code (or pseudo-code) or mention 3 or more function names when you make a post to the class Google Group. For a particular assignment, if you violate this rule for the first time by accident, you will get a warning (unless you posted a lot more than 3 lines of code). If you violate this rule for the 2nd time, since this is considered cheating (but may be unintentional), you will lose 50% of your assignment points and your posting privilege to the class Google Group will be revoked. You need to learn how to have a high-level discussion to talk about code without using anything that look like code or pseudo-code. It's an important skill to learn.

Only one member of your group needs to make a submission. It would be best if you **designate only one member to make all the submissions** to avoid confusion. It's also a good idea to designate one member of your group to **verify your kernel submission** and check the submitted README file to make sure that it is consistent with the submitted code and it contains all the necessary information.

Finally, please note that the pristine kernel code and documentation of `weenix` (in PDF) came from Brown University. Therefore, what's mentioned in the source code comment blocks and the documentation are **not** necessarily the requirements for our class, as far as grading is concerned. **Our requirements are the kernel assignment specifications (i.e., this web page) and the grading guidelines.**

README Requirements for Kernel Assignments

The README requirements for the kernel assignments are similar to the README requirements for the warmup assignments. For each of the kernel assignments, you must follow the instruction in the provided README file and **replace** the provided README file with the corresponding README tempalte in the spec. To complete the README file, the basic idea is to find **all instances of question marks** in the README template and replace them with something appropriate. For each kernel assignment, you can lose up to 10 points for an incomplete README file. For each a question mark that is not replaced with something appropriate as described below, you would lose at least 0.5 point. Here are some more details for each of the **required sections**:

- For the "**BUILD & RUN**" section, you must replace "`(Comments?)`" with the commands to be used to build the kernel. If you need to grader to modify "`Config.mk`" to be something different from the grading guidelines, please indicate what to do before a build command is issued. Please understand that if you ask the grader to change something that he/she is not allowed to change (such as changing a `.c` file), the grader is not permitted to follow your instruction.

- For the "**SELF-GRADING**" section, there are two parts. The first part corresponds to the "Plus points" section of the grading guidelines. You must **replace** each question mark with a **numerical value** which is the score you think you will get for the corresponding item (unless the text specificially instruct you to put down something else). The 2nd part corresponds to the "Minus points" section of the grading guidelines. You must **replace** each question mark with a numerical value that is less than or equal to 0 which is the deduction you think you will get for the corresponding item. In addition, please **replace** "`(Comments?)`" with appropriate information.

- For the "**CONTRIBUTION FROM MEMBERS**" section, there are two parts. For the first part, you must **replace** the queation mark by listing names and USC e-mail addresses of team members (please don't use gmail addresses). For the 2nd part, you must **replace** "`(Comments?)`" with either "`yes`" or "`no`", and if you answered "`no`", please list the percentages for each team member. Please remember that everyone in your team would get the highest possible score if you **claim** that there was equal contribution (i.e., everyone team member would lose points if you do not work well as a team).
- For the "**BUGS / TESTS TO SKIP**" section, you must **replace** the question mark with instructions for the grader if there are tests in the "Plus points" section that should be skipped because you know that they won't work (and may be running them will cause you to lose additional points in the "Minus points" section of the grading guidelines). If the grader skips an item because you ask the grader to skip it, you will get a zero for that item, but you **may not** get a deduction in the "Minus points" section of the grading guidelines if the grader will not encounter a particular type of failure. If there's nothing to skip, please **replace** the question mark with the word "none".

Please note that there may be additional assignment-specific sections for which you are **required** to provide information. They will be clearly labeled with the text "(Required)" in each of the grading guidelines. If you see them in the grading guidelines, please read them carefully and provide needed information accordingly.

Please do **not** include anything from the spec in your README file. But if you want to (so that you have a self-contained document), please mark it clearly so the grader can easily skip such description.

Conforming `dbg()` Calls

There are only **two reasons** why you would (and must) make a "conforming `dbg()` call" in your code. (1) To satisfy a **KASSERT** requirement in section (A) of the grading guidelines. In this case, each required **KASSERT()** call must be followed immediately by a corresponding "conforming `dbg()` call" (please see examples below to see when you can save some of such calls). (2) To satisfy a **"SELF-check"** requirement in the grading guidelines. Please make sure you understand these two requirements before proceeding.

A `dbg()` call can take variable number of arguments. Below is the requirement for a "conforming `dbg()` call".

1. The first argument must be **`DBG_PRINT`**. (Please understand that this is **not equivalent** to calling `dbg_print()`. Don't use `dbg_print()` if you want to **receive any credit**.)
2. The second argument must be a string literal in the form, "**`(GRADING#S X.Y)\n`**", where "**`#`**" is **always required** and is the kernel assignment number (i.e., 1, 2, or 3), "**`S`**" is **always required** and corresponds to a section number in the respective grading guidelines (e.g., A, B, C, ...), and "**`X.Y`**" corresponds to a specific subtest and item number in the respective grading guideline (e.g., "**`1.a`**" for item "`a`" of subtest "`1`"). The parentheses, the spelling of "GRADING", the position of space and "\n" characters must be done **exactly** as indicted here or **you will lose a lot of points**.

   For section (A) in a grading guideline, you must include "**`X.Y`**" part of the notation. For other sections of a grading guideline, you should change "**`X.Y`**" to something that matches what you are doing (and your documentation).

   If you don't have an item number (or there is no way to run items in subtest `X` separately), you can omit the "**`Y`**" part and use "**`(GRADING#S X)\n`**".

   If you don't have a subtest number (or there is no way to run the subtests in section `S` separately), you can omit the "**`X.Y`**" part and use "**`(GRADING#S)\n`**".

   If you simply start and stop the kernel to reach a particular "conforming `dbg()` call" (i.e., you don't have to run a specific test), you can just label it "**`(GRADING#A)\n`**" (since every KASSERT in section (A) of a grading guideline has a subtest and item numbers, so there is no ambiguity here).

   If you have a subtest that can be **independently run in a shell commandline**, you **must** include the subset number (i.e., "**`X`**" above).
3. If you have additional arguments or if the second argument contains more information than the required form, you will **not** receive credit for the corresponding "conforming `dbg()` call". Please understand that when a "conforming `dbg()` call" is executed, the module name and line number gets printed together with the 2nd argument. By looking at the program trace, you can easily tell one "conforming `dbg()` call" from another becuase their module name and line numbers together will be unique. Therefore, there is no ambiguity in telling which "conforming `dbg()` call" is executed!

   If you really really want to add something descriptive, you must use a different `dbg()` call with the first argument being **`DBG_TEMP`** (if you use **`DBG_PRINT`** for this purpose, **you will lose a lot of points**). Please understand that this is still **strongly discouraged** for the reason mentioned in this kernel FAQ item.

The grader will set "`DBG=error,print,test`" in "`Config.mk`" when grading section (A) of the grading guidelines and SELF-checks and will **only** look for "conforming `dbg()`" printout in the terminal. (Plese note that "`DBG=error,test`" in "`Config.mk`" will be used when the grader grades the PRE-CANNED tests.) Specifically, the grader will look for lines that contain the substring "`(GRADING1`", "`(GRADING2`", and "`(GRADING3`" when grading kernel 1, kernel 2, and kernel 3, respectively. Therefore, if you **don't use the exact format** mentioned here, you may end up **losing a lot of points**. When grading kernel 2, the grader will not look at any printout that contains "`(GRADING1`". When grading kernel 3, the grader will not look at any printout that contains "`(GRADING1`" or "`(GRADING2`".

If you have debugging statements for yourself that you don't want the grader to pay attention to (i.e., not for grading), you should use `dbg(DBG_TEMP, "...\n", ...)`. Please note that every such call must produce printout on a separate line or the grader will take points off.

Please note that **`dbg(DBG_PRINT, ...)`** and **`dbg_print()`** are **not** equivalent because `dbg_print()` does **not** print module information and line numbers. You must **not** use **`dbg_print()`** for anything grading related or you will end up losing a lot of points.

Finally, some students like to add identifying text between "`)`" and "`\n`" in a "conforming `dbg()` call". Please understand that if you do that, **you will lose a lot of points**.

Examples

Examples below are given for Kernel Assignment 1 only. For other kernel assignments, please change "`GRADING1`" to "`GRADING#`" where "`#`" is the assignment number and use the correct section letters in the respective grading guidelines.

- if a code sequence is executed by just starting/stopping weenix, use the following to satisfy the SELF-checks requirement:

  ```
  dbg(DBG_PRINT, "(GRADING1A)\n");
  ```

- if a code sequence is **only** executed by running kshell, use the following to satisfy the SELF-checks requirement (`X` is a valid subtest number):

  ```
  dbg(DBG_PRINT, "(GRADING1B X)\n");
  ```

- if a code sequence is **only** executed by calling `faber_thread_test()`, use the following to satisfy the SELF-checks requirement:

  ```
  dbg(DBG_PRINT, "(GRADING1C)\n");
  ```

- if a code sequence is **only** executed by calling `sunghan_test()`, use the following to satisfy the SELF-checks requirement:

  ```
  dbg(DBG_PRINT, "(GRADING1D 1)\n");
  ```

- if a code sequence is **only** executed by calling `sunghan_deadlock_test()`, use the following to satisfy the SELF-checks requirement:

  ```
  dbg(DBG_PRINT, "(GRADING1D 2)\n");
  ```

- if a code sequence can **only** be executed by running your own test and you only have one test, do:

  ```
  dbg(DBG_PRINT, "(GRADING1E)\n");
  ```

- if a code sequence can **only** be executed by running your own test #2, do:

  ```
  dbg(DBG_PRINT, "(GRADING1E 2)\n");
  ```

- to get credit for item (A.x.y), do:

  ```
  KASSERT(...);                        /* "..." means whatever that's required, according to the grading guidelines *
  dbg(DBG_PRINT, "(GRADING1A x.y)\n"); /* nothing between this line and the above KASSERT */
  ```

- to get credit for multiple and consecutive entries in item (A.x.y), you can do:

  ```
  KASSERT(...);                        /* for example, for x.y */
  KASSERT(...);                        /* also for x.y; nothing between this line and the above KASSERT */
  KASSERT(...);                        /* also for x.y; nothing between this line and the above KASSERT */
  dbg(DBG_PRINT, "(GRADING1A x.y)\n"); /* nothing between this line and the above KASSERT */
  ```

- to get credit for (A.3.b), do:

  ```
  KASSERT(...);                        /* for example, for 3.b */
  dbg(DBG_PRINT, "(GRADING1A 3.b)\n")  /* nothing between this line and the above KASSERT */;
  dbg(DBG_PRINT, "(GRADING1?)\n");     /* specify the correct test for "?";
                                          nothing between this line and the above conforming dbg() call */
  ```

- to get credit for (A.6.b), do:

  ```
  KASSERT(...);                        /* for example, for 6.b */
  dbg(DBG_PRINT, "(GRADING1A 6.b)\n")  /* nothing between this line and the above KASSERT */;
  dbg(DBG_PRINT, "(GRADING1?)\n");     /* specify the correct test for "?";
                                          nothing between this line and the above conforming dbg() call */
  ```

Again, if there are multiple ways to execute a code sequence, you only need to tell the grader one way to execute that code sequence (and you get to choose which way). There is no requirement that you need to tell the grader all possible ways of executing a code sequence. The above are just some examples (i.e., they don't cover all possible cases). For cases that are not covered exactly, if you are not sure, please ask the instructor.

Group and Grading

Please see information regarding group formation and group grading.

Late Submission Policy

Please see the late policy.

Debugging with `gdb`

You should start using the debugger (`gdb`) right away! For some basic `gdb` commands, please see slide 5 of the week 1 discussion section slides. This can save from you a lot of frustration! To debug the kernel with `gdb`, you need to set **GDBWAIT=1** in **Config.mk**, recompile the kernel (by doing "`make clean; make`"), and start `weenix` by running the following command instead:

```
./weenix -n -d gdb -w 10
```

(If you have a slow machine, you may have to use a value larger than 10. If you have a fast machine, you may have to use a value smaller than 10. If you have a super fast machine, you can use a value of 1.) The above command invokes `gdb` to debug the `weenix` kernel. If all goes well, you should see the following at the bottom of the terminal window (please note that the top part may be OS and configuration dependent):

```
/usr/bin/qemu-system-i386
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from kernel/weenix.dbg...done.
0xc0006bf9 in kmain () at main/kmain.c:143
143         while (gdb_wait) ;
Breakpoint 1 at 0xc000944e: file util/debug.c, line 190.
Breakpoint 2 at 0xc0006c77: file main/kmain.c, line 160.
Breakpoint 3 at 0xc0006c84: file main/kmain.c, line 180.

Breakpoint 3, bootstrap (arg1=0, arg2=0x0) at main/kmain.c:180
180         dbgq(DBG_TEST, "SIGNATURE: ...\n");
(gdb)
```

It's perfectly fine if you are getting different addresses. But the line numbers should match.

You should try a few `gdb` commands from the week 1 discussion section slides. When you are ready to proceed, type the following at the `gdb` prompt (you don't need to type "`(gdb) `"; it's left there so you know that the "`c/cont`" command is to be entered at the `gdb` prompt):

```
(gdb) c
```

You should get a kernel panic and see the same "Not yet implemented" message as above in a debugging window (the title of the window should show "**qemu-system-i386**"). Since breakpoint 1 of the debugger is set at the kernel panic routine, you should see the following:

```
Continuing.

Breakpoint 1, dbg_panic_halt () at util/debug.c:190
190         __asm__ volatile("cli; hlt");
(gdb)
```

To get out of debugging, press <Ctrl+C> once and then type the following at the `gdb` prompt:

```
(gdb) quit
```

Sometimes, you may have to press <Ctrl+C> to get the attention of `gdb`. Unlike debugging a user program in our warmup assignments, when you are debugging `weenix` and you press <Ctrl+C>, you would kill `QEMU` (and therefore, you would also kill `weenix`)! So, only press <Ctrl+C> if you are ready to quit `weenix`.

Please note that when you are running `weenix` with `gdb`, all the kernel debugging messages will go to a window with **"qemu-system-i386"** in the titlebar. If you don't see this window and only see the window with **"QEMU"** in the titlebar, that's not abnormal! If you want to see all the kernel debugging messages, either you have to run `weenix` without `gdb`, or you can set **GDBWAIT=1** in **Config.mk** and start `weenix` by running "**`./weenix -n -d gdb -w 10`**". Please see comments in **Config.mk** and slide 9 of the Week 6 discussion section slides.

If you have setup everything correctly but when you run "`./weenix -n -d gdb -w 10`", `gdb` would not break inside `bootstrap()` as illustrated above, please take a look at the line right below "`Reading symbols from kernel/weenix.dbg...done`". It should say "`0xc0006bf9 in kmain () at main/kmain.c:143`" (the numbers may not be exact). If you don't see that line of it has "`?`" in it, then it may be due to improper VirtualBox settings for NAT Forwarding (by default, the NAT Forwarding settings should be empty and you shouldn't have modified anything in "network" settings when you setup your VirtualBox; if you did modify something in "network" settings" and ended up you cannot do your kernel assignments, you should reinstall everything from scratch).

Kernel Assignment 1

For the assignment description, please see **Chapter 3**, **"Processes and Threads"**, of the weenix documentation (in PDF). (To complete this assignment, please also see **Chapter 4**, **"Drivers"**, of the same documentation.) When you start using `kshell`, you need to also set `DRIVERS=1` in `Config.mk`.

The code you need to implement for this assignment are labeled as:

```
NOT_YET_IMPLEMENTED("PROCS: ...");
```

To find all the code that have such as label, please enter the following command:

```
grep PROCS: kernel/*/*.c
grep PROCS: kernel/*/*/*.c
```

You need to get familiar with the grep command because that's the command you can use to search for strings in a large collection of files.

Please make code changes **in-place** (and you must delete or comment out the corresponding `NOT_YET_IMPLEMENTED()` call if you write code for it). Any helper function you add must go into a file that's part of the standard submission. If you want to include additional code to test/exercise your kernel, you must put those code in "`kernel/main/kmain.c`".

You are provided with an empty **`procs-README.txt`** file along with the pristine weenix source code. You must replace it with the content of this template ("k1-README.txt") and fill it out with your documentation information (i.e., **replace all "(Comments?)"** with your evalution and **replace all standalone "?"** with information appropriate for your submission). You must not delete a single line from this README file. For more details, please see the requirements on README.

Additional Notes

You are expected to also do the following.

When you are all done with all the **`PROCS`** code, you must try the following to perform additional testing. Since the **`DRIVERS`** code has been provided to you as part of the assignment, you can actually enable it by editing **`Config.mk`** and change the values of the **`DRIVERS`** variable to **`1`**. Then do:

```
make clean
make
./weenix -n
```

If this works without a problem, the next thing you should do is to run a shell (known as **`kshell`**) by following the instruction in kshell FAQ. Then recompile and rerun weenix. If all goes well, you should get a running kernel shell (no file system though). At the shell prompt, you can type **`help`** to see all the available shell commands.

You are required to **demonstrate** that your code works properly. The best way to do this is to add test functions in "`kernel/main/kmain.c`" and make them accessible through commands in **`kshell`**. Please see **Section 4.6** of the weenix documentation (in PDF) regarding how to get **`kshell`** to work. You should add **commands** to your **`kshell`** to invoke additional tests and to exercise your code. Make sure you document your commands well enough so that the grader knows what to do.

Three tests routines have been written for you to exercise and test your implementation. They are `faber_thread_test()` in "`kernel/proc/faber_test.c`" and `sunghan_test()` and `sunghan_deadlock_test()` in "`kernel/proc/sunghan_test.c`". You must provide 3 separate `kshell` commands to invoke these functions. Not only you need to "pass these tests", you should also read the code of these routines and figure out what printout you should expect. You need to make sure that the printout from all these programs are correct. We will not provide the "correct output". You can discuss this with your classmates in the class Google Group.

Please keep the following in mind when you are doing this assignment (to keep you out of trouble):

- As far as the machine is concerned, the kernel is **very very powerful**. There's nothing the kernel cannot do on that machine.
- The `weenix` kernel is **non-preemptive**. Think hard about what this means and what this implies.
- We are **not** implementing multiple threads per process (i.e., **`MTP=0`** in `Config.mk`). So, if you want to create a new kernel thread, you must first create a new kernel process.

Some Hints

Have you checked out the kernel FAQ? It's probably worthwhile to read through the entire kernel 1 section of it at least once.

At any point in time in the execution of your kernel, you should be able to answer the following questions:

- Where are ALL the kernel threads? (Remember, one in the CPU and all other threads are waiting in a queue somewhere.)
- How (which line of code and in which function) did they get to where they are? (Remember, we have a non-preemptive kernel.)
- For each kernel thread, who will move it to the run queue (and under what condition)? (Clearly, a thread waiting in a queue cannot move itself.)

You may want to consider walking through the test code mentioned in the grading guidelines first before you start coding to understand how some important functions are used so you know what you need to implement in these functions.

Submission

If you have made all your code changes in-place (which you are requied to do), it will be very easy to create a **gzipped tarfile** for submission using the Bistro system. To create your submission file, just do the following:

```
make procs-submit
```

and the following command should run:

```
tar cvzf procs-submit.tar.gz \
        Config.mk \
        procs-README.txt \
        kernel/main/kmain.c \
        kernel/proc/kmutex.c \
        kernel/proc/kthread.c \
        kernel/proc/proc.c \
        kernel/proc/sched.c \
        kernel/proc/sched_helper.c
```

Please note that you are **not** permitted to change **any other existing files**. If you submit another existing file, it will be **deleted** before grading. If you give instructions to the grader to modify another existing file (other than `Config.mk`), it will disregarded.

Please enter your **USC e-mail address** and your **submission PIN** below. Then click on the **Browse** button and locate and select your submission file (i.e., `procs-submit.tar.gz`). Then click on the **Upload Kernel 1** button to submit your `procs-submit.tar.gz`. If you see an error message, please read the dialogbox carefully and fix what needs to be fixed and repeat the procedure. If you don't know your submission PIN, please visit this web site to have your PIN e-mailed to your USC e-mail address.

When this web page was **last loaded**, the time at the submission server (**merlot.usc.edu**) was **17Dec2022-22:45:30**. **Reload** this web page to see the **current time** on **merlot.usc.edu**.

USC E-mail:	**@usc.edu**
Submission PIN:	
Event ID (read-only):	merlot.usc.edu_80_1557931083_238
Submission File Full Path:	Choose File No file chosen
	Upload Kernel 1

If the command is executed successfully and if everything checks out, a **ticket** will be issued to you to let you know "what" and "when" your submission made it to the Bistro server. The next web page you see would display such a ticket and the ticket should look like the sample shown in the submission web page (of course, the actual text would be different, but the format should be similar). Make sure you follow the Verify Your Ticket instructions to verify the SHA1 hash of your submission to make sure what you did not accidentally submit the wrong file. Also, an e-mail (showing the ticket) will be sent to your USC e-mail address. Please read the ticket carefully to know exactly "what" and "when" your submission made it to the Bistro server. If there are problems, please contact the instructor.

Please submit **only source code** that you have modified or added. You will lose 2 points if your submission includes binary files. You will lose 2 points if your submission includes too many files that were identical to what's given to you in the assignment. You will also lose 2 points if files in your submission are not in the correct directory (as specified in the original **`Makefile`**). For example, **`kmain.c`** must be in **`kernel/main`** in your submitted `.tar.gz` file after it is unzipped and un-tared.

It is **extreme important** that you also **verify your kernel submission** after you have submitted `procs-submit.tar.gz` electronically to make sure that every you have submitted is everything you wanted us to grade.

Grading Guidelines

The grading guidelines has been made available. It is possible that there are bugs in the guidelines. If you find bugs, please let the instructor know as soon as possible.

The grading guidelines is the **only** grading procedure we will use to grade your program. No other grading procedure will be used. (We may make minor changes if we discover bugs in the script or things that we forgot to test.)

Kernel Assignment 2

For the assignment description, please see **Chapter 5**, **"Virtual File System"**, of the weenix documentation (in PDF). Please remember to edit **`Config.mk`** and change the values of the **`DRIVERS`** and **`VFS`** variables to **`1`**.

The code you need to implement for this assignment are labeled as:

```
NOT_YET_IMPLEMENTED("VFS: ...");
```

To find all the code that have such as label, please enter the following command:

```
grep VFS: kernel/*/*.c
```

Please make code changes **in-place** (and you must delete or comment out the corresponding `NOT_YET_IMPLEMENTED()` call if you write code for it). Any helper function you add must go into a file that's part of the standard submission. If you want to include additional code to test/exercise your kernel, you must put those code in "`kernel/main/kmain.c`".

You are provided with an empty **`vfs-README.txt`** file along with the pristine weenix source code. You must replace it with the content of this template ("k2-README.txt") and fill it out with your documentation information (i.e., **replace all "(Comments?)"** with your evalution and **replace all standalone "?"** with information appropriate for your submission). You must not delete a single line from this README file. For more details, please see the requirements on README.

You are also **required** to enable **`kshell`** and get it running in your implementation. Please see **Section 4.6** of the weenix documentation (in PDF) regarding how to get **`kshell`** to work. You should add **commands** to your **`kshell`** to invoke additional tests and to exercise your code. One command you must add is **"vfstest"** and it should invoke `vfstest_main()` (with `argc=1` and `argv=NULL`) in "`kernel/test/vfstest/vfstest.c`" in a separate process. You are also required to add two separate `kshell` commands to invoke `faber_fs_thread_test()` and `faber_directory_test()`. Please notice that these two functions have function prototypes of a `kshell` command. Therefore, please run them **directly** in `kshell` and do **not** run them in a separate process. (Just to be perfectly clear, you must

NOT modify "`vfstest.c`" and "`faber_fs_test.c`". If you submit your own "`vfstest.c`" or "`faber_fs_test.c`", we will delete it and use the original one to grade your submission.)

Additional Notes

This assignment is about:

- File naming
- File protection
- File abstraction API
- Kernel data structures for open files
- Support for memory mapping files and sharing

You should be familiar with the following sections of the textbook:

- Section 1.3.5
- Section 4.1.4.2

The actual file system that's supporting VFS in this assignment is the `ramfs` file system (a RAM-based file system). Please read section 5.2 of the weenix documentation (in PDF) for an instruction to the `ramfs` file system. You should use `gdb` to step inside the code of the `ramfs` file system to see what's going on in an actual file system in `weenix` so you can understand how an actual file system is suppose to work. You should get familiar with the code in `ramfs` to understand how **polymorphism** is implemented and used. This is important because when you are done with this assignment, you will switch to use **`S5FS`** which you do **not** have **all** the source code and will have to use your imagination. So, I would strongly suggest that you try to understand pretty much every line of code in "`kernel/fs/ramfs/ramfs.c`".

Please note that although the **`S5FS`** code has been provided to you in "`kernel/libs5fs.a`", the **`S5FS`** code won't work because a crucial part (related to Virtual Memory) is still missing, which is probably the first thing you will need to implement in Kernel Assignment 3. So, even if you set the **`S5FS`** variable to **`1`** in **`Config.mk`** and all your code works perfectly, your kernel will still die because **`pframe_get()`** is not implemented.

Some Hints

Have you checked out the kernel FAQ? It's probably worthwhile to read through the entire kernel 2 section of it at least once.

Submission

If you have made all your code changes in-place (which you are requied to do), it will be very easy to create a **gzipped tarfile** for submission using the Bistro system. To create your submission file, just do the following:

```
make vfs-submit
```

and the following command should run:

```
tar cvzf vfs-submit.tar.gz \
        Config.mk \
        vfs-README.txt \
        kernel/main/kmain.c \
        kernel/proc/kmutex.c \
        kernel/proc/kthread.c \
        kernel/proc/proc.c \
        kernel/proc/sched.c \
        kernel/proc/sched_helper.c \
        kernel/fs/namev.c \
        kernel/fs/open.c \
        kernel/fs/vfs_syscall.c \
        kernel/fs/vnode.c
```

Please note that you are **not** permitted to change **any other existing files**. If you submit another existing file, it will be **deleted** before grading. If you give instructions to the grader to modify another existing file (other than `Config.mk`), it will disregarded.

Please enter your **USC e-mail address** and your **submission PIN** below. Then click on the **Browse** button and locate and select your submission file (i.e., `vfs-submit.tar.gz`). Then click on the **Upload Kernel 2** button to submit your `vfs-submit.tar.gz`. If you see an error message, please read the dialogbox carefully and fix what needs to be fixed and repeat the procedure. If you don't know your submission PIN, please visit this web site to have your PIN e-mailed to your USC e-mail address.

When this web page was **last loaded**, the time at the submission server (**merlot.usc.edu**) was **17Dec2022-22:45:30**. **Reload** this web page to see the **current time** on **merlot.usc.edu**.

USC E-mail:	**@usc.edu**
Submission PIN:	
Event ID (read-only):	merlot.usc.edu_80_1557931083_239
Submission File Full Path:	Choose File No file chosen
	Upload Kernel 2

If the command is executed successfully and if everything checks out, a **ticket** will be issued to you to let you know "what" and "when" your submission made it to the Bistro server. The next web page you see would display such a ticket and the ticket should look like the sample shown in the submission web page (of course, the actual text would be different, but the format should be similar). Make sure you follow the Verify Your Ticket instructions to verify the SHA1 hash of your submission to make sure what you did not accidentally submit the wrong file. Also, an e-mail (showing the ticket) will be sent to your USC e-mail address. Please read the ticket carefully to know exactly "what" and "when" your submission made it to the Bistro server. If there are problems, please contact the instructor.

Please submit only **source code** that you have modified or added. You will lose 2 points if your submission includes binary files. You will lose 2 points if your submission includes too many files that were identical to what's given to you in the assignment. You will lose 2 points if files in your submission are not in the correct directory (as specified in the original **`Makefile`**).

It is **extreme important** that you also **verify your kernel submission** after you have submitted `vfs-submit.tar.gz` electronically to make sure that every you have submitted is everything you wanted us to grade.

Grading Guidelines

The grading guidelines has been made available. It is possible that there are bugs in the guidelines. If you find bugs, please let the instructor know as soon as possible.

The grading guidelines is the **only** grading procedure we will use to grade your program. No other grading procedure will be used. (We may make minor changes if we discover bugs in the script or things that we forgot to test.)

Kernel Assignment 3

For the assignment description, please see **Chapter 7**, **"Virtual Memory"**, of the weenix documentation (in PDF). (It's a good idea to also read **Chapter 6**, **"System V File System"**, of the same documentation.) Please remember to edit **`Config.mk`** and change the values of the **`DRIVERS`**, **`VFS`**, **`S5FS`**, and **`VM`** variables to **`1`**. (Only use **`DYNAMIC=1`** towards the end for extra credit when everything seems to be working.) It's best **not** to attempt to **debug anything** when `DYNAMIC=1`.

The code you need to implement for this assignment are labeled as:

```
NOT_YET_IMPLEMENTED("VM: ...");
```

To find all the code that have such as label, please enter the following command:

```
grep VM: kernel/*/*.c
```

Please note that you do **not** need to implement **`zero_mmap()`** in "`kernel/drivers/memdevs.c`" and **`s5fs_mmap()`** in "`kernel/fs/s5fs/s5fs.c`" because they were implemented for you already (although it doesn't look like it). Their implementations are in "`kernel/libdrivers.a`" and "`kernel/libs5fs.a`".

Please make code changes **in-place** (and you must delete or comment out the corresponding `NOT_YET_IMPLEMENTED()` call if you write code for it). Any helper function you add must go into a file that's part of the standard submission. If you want to include additional code to test/exercise your kernel, you must put those code in "`kernel/main/kmain.c`".

You are provided with an empty **`vm-README.txt`** file along with the pristine weenix source code. You must replace it with the content of this template ("k3-README.txt") and fill it out with your documentation information (i.e., **replace all "(Comments?)"** with your evalution and **replace all standalone "?"** with information appropriate for your submission). You must not delete a single line from this README file. For more details, please see the requirements on README.

Additional Notes

In addition to the assignment description, you need to be familiar with the following sections of the the weenix documentation (in PDF).

- Section 6.3

You should also read the man pages of **`mmap()`** on Ubuntu 16.04 to get a basic understanding of the concept of **creates a new mapping in the virtual address space**. Please note that this **`mmap()`** is a system call to create a mapping at the user's request (for example, to map a file into memory). In the kernel, similar mappings needs to be created when the kernel executes a user program (to map in the text and data segments of a user program into the address of the user program). Please note that this is not exactly the same as what you need to implement in `weenix` because `weenix` is not Linux.

Assuming your kernel 2 submission was perfect. If you just set **`S5FS=1`** in **`Config.mk`** (keep **`VM=0`**) and recompile and rerun `weenix`, your kernel will die because **`pframe_get()`** is not implemented. It is **strongly recommended** that you start this assignment by first implementing **`pframe_get()`**, **`pframe_pin()`**, and **`pframe_unpin()`** in "`kernel/mm/pframe.c`" so you understand how physical page frames work. When you think that these functions are working properly, you should run **`vfstest`** (via `kshell`) again to increase your confidence that your kernel 2 was done correctly. (Please note that, at this point, you are running `vfstest` in the kernel. This is not the same as running "`/usr/bin/test/vfstest`" in "userland".) At this point, you can go ahead and set **`VM=1`** and proceed with the rest of kernel 3.

In section 7.8.1 of the weenix documentation (in PDF), it mentioned that you should call **`kernel_execve()`** in your **`init`** process and execute the "**`/sbin/init`**" program. You should not pass NULL as the 2nd and 3rd arguments to **`kernel_execve()`**.

Instead, you can invoke **`kernel_execve()`** in the following way:

```
char *argv[] = { "/sbin/init", NULL };
char *envp[] = { NULL };
kernel_execve("/sbin/init", argv, envp);
```

You can also use **`kernel_execve()`** to invoke other user space programs, such as "`/usr/bin/hello`", "`/usr/bin/args`", "`/bin/uname`", "`/usr/bin/fork-and-wait`", etc., directly from the kernel. It is **highly recommended** that you have all of these programs run in user space perfectly **before** attempting "`/sbin/init`".

Please note that when you shutdown the kernel, you should see the "weenix: halted cleanly" message in the terminal. But unlike kernel 2, you may see a few messages from `s5fs_check_refcounts()`, complaining that the refcount check of `s5fs` filesystem was completed UNSUCCESSFULLY and a **warning** about linkcount corruption. (To see this, you must include "**`print`**" in the **`DBG`** flag in **`Config.mk`**.) Since this is just a warning, it is perfectly acceptable as long as you also see the "weenix: halted cleanly" message in the end.

Some Hints

Have you checked out the kernel FAQ? It's probably worthwhile to read through the entire kernel 3 section of it at least once.

One major confusion in this assignment is that **polymorphism** got even more complicated here. In kernel 2, you were asked to explore the code for polymorphism in the file system. But the code was fairly straight-forward since you are only deal with **one** actual file system and there is not recursion. In this assignment, you will be implementing something called **`mmobj`** which has 3 actual instances (i.e., anonymous objects, shadow objects, and there is also a type of `mmobj` in a `vnode`) and some polymorphic function calls may be recursive. So, if you are not carefully, you will end up with an infinite loop. As all recursive function goes, you should analyze it. You need to figure out what the **base cases** are and when to go into recursion. Think about this carefully before you write a lot of code.

Please note that it is very important that you understand the concepts in section 7.3 of the textbook, especially how **demand paging** works. Let's take "`/usr/bin/hello`" as an example. In order to run the `hello` program, you need to first setup its address space (i.e., the memory map). When you execute the very first machine instruction in `hello` (even before you get to `main()`), you should expect a **page fault** (and you need to understand why). This is a "good" page fault. The purpose of a "good" page fault is for the kernel to find a page frame (and a physical page), fill the page frame (to be more precise, the physical page associated with the page frame) with data from disk (if it's associated with a file), fix the page table to map the corresponding virtual page to the physical page, and update kernel data structures to make everything consistent. When the kernel declares that the page fault has been fixed and returns to the user process, the user process should not page fault at the same virutal address immediately. If it page faults again immediately, it means that something is not setup properly. How do you debug this? Well, you need to know exactly what the user process should "see" and find out why the user process is not "seeing" what it is suppose to see. You need to know that the text segment of "`/usr/bin/hello`" looks like (using **`objdump`**, see an example in the weenix documentation (in PDF)) and **verify** that the user process can see these byte patterns and will be executing the right machine instructions. You need to read the machine/assembly code and see when the stack or data segment is **referenced** and know whether referencing such a memory location should cause a page fault or not. You may have to single-step machine instructions.

Submission

If you have made all your code changes in-place (which you are requied to do), it will be very easy to create a **gzipped tarfile** for submission using the Bistro system. To create your submission file, just do the following:

```
make vm-submit
```

and the following command should run:

```
tar cvzf vm-submit.tar.gz \
        Config.mk \
        vm-README.txt \
        kernel/main/kmain.c \
        kernel/proc/kmutex.c \
        kernel/proc/kthread.c \
        kernel/proc/proc.c \
        kernel/proc/sched.c \
        kernel/proc/sched_helper.c \
        kernel/fs/namev.c \
        kernel/fs/open.c \
        kernel/fs/vfs_syscall.c \
        kernel/fs/vnode.c \
        kernel/api/access.c \
        kernel/api/syscall.c \
        kernel/mm/pframe.c \
        kernel/proc/fork.c \
        kernel/vm/anon.c \
        kernel/vm/brk.c \
        kernel/vm/mmap.c \
        kernel/vm/pagefault.c \
        kernel/vm/shadow.c \
        kernel/vm/vmmap.c
```

Please note that you are **not** permitted to change **any other existing files**. If you submit another existing file, it will be **deleted** before grading. If you give instructions to the grader to modify another existing file (other than `Config.mk`), it will disregarded.

Please enter your **USC e-mail address** and your **submission PIN** below. Then click on the **Browse** button and locate and select your submission file (i.e., `vm-submit.tar.gz`). Then click on the **Upload Kernel 3** button to submit your `vm-submit.tar.gz`. If you see an error message, please read the dialogbox carefully and fix what needs to be fixed and repeat the procedure. If you don't know your submission PIN, please visit this web site to have your PIN e-mailed to your USC e-mail address.

When this web page was **last loaded**, the time at the submission server (**merlot.usc.edu**) was **17Dec2022-22:45:30**. **Reload** this web page to see the **current time** on **merlot.usc.edu**.

USC E-mail:	**@usc.edu**
Submission PIN:	
Event ID (read-only):	merlot.usc.edu_80_1557931083_240
Submission File Full Path:	Choose File No file chosen
	Upload Kernel 3

If the command is executed successfully and if everything checks out, a **ticket** will be issued to you to let you know "what" and "when" your submission made it to the Bistro server. The next web page you see would display such a ticket and the ticket should look like the sample shown in the submission web page (of course, the actual text would be different, but the format should be similar). Make sure you follow the Verify Your Ticket instructions to verify the SHA1 hash of your submission to make sure what you did not accidentally submit the wrong file. Also, an e-mail (showing the ticket) will be sent to your USC e-mail address. Please read the ticket carefully to know exactly "what" and "when" your submission made it to the Bistro server. If there are problems, please contact the instructor.

Please submit only **source code** that you have modified or added. You will lose 2 points if your submission includes binary files. You will lose 2 points if your submission includes too many files that were identical to what's given to you in the assignment. You will lose 2 points if files in your submission are not in the correct directory (as specified in the original **`Makefile`**).

It is **extreme important** that you also **verify your kernel submission** after you have submitted `vm-submit.tar.gz` electronically to make sure that every you have submitted is everything you wanted us to grade.

Grading Guidelines

The grading guidelines has been made available. It is possible that there are bugs in the guidelines. If you find bugs, please let the instructor know as soon as possible.

The grading guidelines is the **only** grading procedure we will use to grade your program. No other grading procedure will be used. (We may make minor changes if we discover bugs in the script or things that we forgot to test.)

Please note that the kernel 3 grading guidelines assumes that you can successfully run the **user space shell** (i.e., run "`/sbin/init`" by calling `kernel_execve()` from `initproc_run()`) and run all the commands in the grading guidelines from the user space shell. If you can do all that, you do **not** need kshell at all in kernel 3. **HOWEVER**, if you **cannot** get the user space shell to run, you should still try to get as much partial credit as possible, if you can run some of the user space programs successfully by themselves. If this is the case, you should implement a bunch of kshell commands, each kshell command would invoke only **one subtest in the grading guidelines** by passing the right parameters to `kernel_execve()` in a child process created by that kshell command to run the corresponding program in user space. At the end of that kshell command, you should wait for all child processes to die before giving back the kshell prompt. You should be able to run these kshell commands over and over again, without restarting weenix (unless the grading guidelines says that you must restart weenix). In this case, please document exactly how to invoke each such kshell command in the "ADDITIONAL INFORMATION FOR GRADER" section of your README file. For convenience, I would probably name these kshell commands "`b1`", "`b2`", ..., "`b7`", "`d1`", "`d2`", ..., "`d5`".

Verify Your Kernel Submission

After you submitted, for example, **`procs-submit.tar.gz`**, you should verify what you've submitted can be compiled as is and works the way you expected it to work and that your documentation is perfect. Let's say your `procs-submit.tar.gz` is in your home directory on your Ubuntu 16.04 machine. Do the following (assuming that you are using the pristine weenix source and **`procs-submit.tar.gz`** in your home directory):

```
rm -rf /tmp/xyzzy
mkdir /tmp/xyzzy
cd /tmp/xyzzy
tar xvzf ~/weenix-assignment-3.8.0.tar.gz
[ look at the output of the above command carefully, make sure you have not included extra files in your submission ]
cd weenix-assignment-3.8.0/weenix
tar xvzf ~/procs-submit.tar.gz
make clean
make
./weenix -n
[ go through grading guidelines line by line and re-run all your tests to make sure they all work ]
[ check every line of your README file and make sure that you have satisfied all the reuiqrements ]
```

A few things to note:

1. If the **`tar`** command failed, the **`procs-submit.tar.gz`** file you have submitted is not properly created.

2. If the `make` command failed, you probably forgot to include something in your submission. Please remember that if this does not work, you may lose quite a few points.
3. If any of the above failure occurs, you must recreate your submission and submit again and verify again.

Backup Your Code

It is **imperative** that you have a plan to recover from accidentally losing all your precious work. Since you are working on your own Ubuntu system, no one else will be doing the backup for you. Sometimes, when you don't shutdown your system properly (maybe because you let your battery go down to 0%), you may not be able to start Ubuntu because the disk inside the virtual machine is corrupted. The only thing you can do at that point is to install a brand new virtual machine and install a fresh 32-bit Ubuntu 16.04 system into it. If you don't have a backup copy of your code, you have to **rewrite all your code from scratch**! Please understand that when that happens, there is nothing anyone can do to recover the dead system!

Ubuntu used to come with a free service called Ubuntu One which you can use as your "personal cloud". Unfortunately, it's no longer available and you have to look for some other solutions. You can probably use DropBox or other cloud-based services that's available on your machine.

Here are a couple of ways to perform backup:

1) Simply type:

```
make backup
```

A backup file will be created and copied into the "`Shared-ubuntu`" directory inside the home directory of your Ubuntu 16.04 system. If you have setup your shared folder according to the installation instructions, this backup file will be accessible on your host machine and you can use whatever cloud backup software on your host machine to save a copy of the backup into cloud storage.

This approach is very simple and it's mainly good for backing up changes for yourself. It may be a bit difficult to share code modifications with your kernel teammates. But in general, it's a good idea to have a backup of all your work anyway.

2) Create a **`git`** repository in the cloud drive and sync with it. This approach is more suitable for a team to share code. Pradeep Nayak, a student from a previous semester, posted steps about how to do this (using DropBox instead) in the class Google Group. I have not verified these steps so I don't know if it works or suitable for you or not.