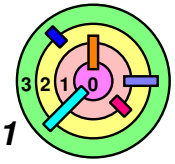
Kernel Programming Assignments (Part 2)

Bill Cheng

http://merlot.usc.edu/william/usc/



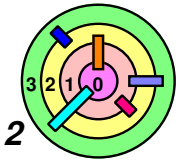
Compilation and Configuration

- `Config.mk` controls what gets compiles and configured into the kernel (weenix is a monolithic OS)
  - for PROCS, use the original `Config.mk`
    - set `DRIVERS` to 1 to complete this assignment
  - for VFS, set `DRIVERS` and `VFS` to 1
  - for VM, set `DRIVERS`, `VFS`, `S5FS`, and `VM` to 1
    - set `VM` to 0 at first to get `kernel/mm/pframe.c` working
    - then set `VM` to 1 to work on the rest of the assignment
    - set `DYNAMIC` to 1 in the end if everything is working
  - by default: `DBG = all`
    - the grader will use these for ***grading:***

```
DBG = error,test
DBG = error,print,test
```

- Every time you modify `Config.mk`, you should do:

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



Debugging with gdb

- The `weenix` documentation says to do this to debug the kernal:

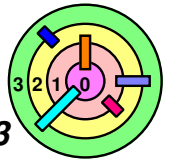
```
% ./weenix -n -d gdb
```

  - although you can use `gdb`, but you cannot see kernel debugging messages (from `dbg()` calls)

- To see kernel debugging messages AND debug the kernel, do:
  - set `GDBWAIT=1` in `Config.mk` then *recompile* kernel
  - run `weenix` under gdb with:

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

    - if you have a slow machine, you should use a larger value
    - if you have a fast machine, you should use a smaller value
  - unfortunately, if you have compiled with `GDBWAIT=1` and want to run *without gdb*, `weenix` will *freeze*
    - you have to set `GDBWAIT` back to `0` and recompile if you want to run `weenix` without gdb
    - you should set `GDBWAIT` back to `0` when you submit your assignment for grading

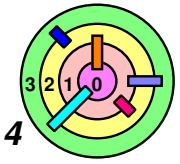


Submissions

- Processes and Threads (PROCS)

```
% make procs-submit
tar cvzf procs-submit.tar.gz \
        Config.mk \
        procs-README.txt \
        ...
```

  - must fill out `procs-README.txt`, it's your assignment's documentation
    - *IMPORTANT:* your must copy the `k1-README.txt` template in the spec into `procs-README.txt` and make changes
    - this is where you should also include
      1) how to split the points (in terms of percentages and must sum to 100%)
      2) brief justification about the split (if not equal split)
         - please understand that if I have to get involved, it can also be unfair since I won't have all the information
         - best to claim even splits
  - submit `procs-submit.tar.gz` using web form



Submissions

- Virtual File System (VFS)
  - need to fill out `vfs-README.txt` (start with `k2-README.txt`)
    - start with the `k2-README.txt` template in the spec

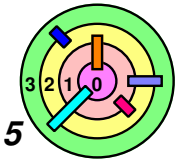
      ```
      % make vfs-submit
      ```

- Virtual Memory (VM)
  - need to fill out `vm-README.txt` (start with `k3-README.txt`)
    - start with the `k3-README.txt` template in the spec

      ```
      % make vm-submit
      ```

- Must *NOT* include *ANY OTHER* file not mentioned in "`make`" above
  - or we will *delete* it before grading
  - you must not add any files

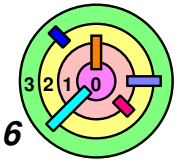
- Submit source code only
  - we will deduct 2 points if you submit binary files
  - we will deduct 2 points if you submit extra files
  - we will deduct 2 points if you do not keep the same directory structure



Verify Your Kernel Submission

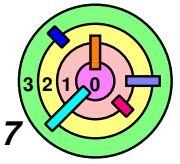
- Assume that in your home directory, you have
  - a *pristine* `weenix-assignment-3.8.0.tar.gz`
  - your submission file, e.g., `procs-submit.tar.gz`
- Do the following to verify your submission

```
% rm -rf /tmp/xyzzy
% mkdir /tmp/xyzzy
% cd /tmp/xyzzy
% tar xvzf ~/weenix-assignment-3.8.0.tar.gz
% 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 ]
[ check every line of your README file ]
```



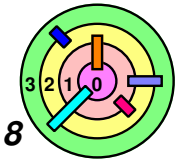
Grading

- The weenix code and comments and the weenix documentation are from Brown University
  - these are just hints and not our "requirements"
- Just like the warmup assignments, the *spec* and the *grading guidelines* are the requirements for our CS 402
  - they are related to "grading"
- Read the grading gudelines carefully!
  - e.g., altering or removing top comment block in a submitted `.c` file will *cost you 20 points* each!
    - why such stiff penalty?
      - because it's extremely easy to follow this rule
  - e.g., altering/removing a call to `dbgq()` in `bootstrap()` in your submitted `kmain.c` file will *cost you 20 points*
    - this is a signature showing who downloaded the kernel source
  - designate a teammate to check your submission against *every line in the grading guidelines*



Structure Of Grading Guidelines

- "Plus Points" section of the grading guidelines
  - *mandatory KASSERTs:* section (A)
    - we are actually trying to help you to write some of your kernel code and give you some easy points at the same time!
    - add `KASSERT()` calls (followed by a "conforming `dbg()` call")
  - *SELF-checks:* last section
    - *every code sequence* inside any function you wrote to replace a `NOT_YET_IMPLEMENTED()` call must *END* with a "conforming `dbg()` call"
      - must use *"static analysis"* (independent of how your code will actually execute) to analyze your code
  - *PRE-CANNED tests:* other sections
- Please read the grading guidelines very carefully
  - when in doubt, ask the instructor!

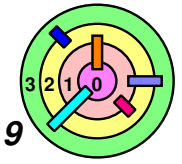


"Conforming `dbg()` Call"

- "Conforming `dbg()` call"
  - general form:

```
dbg(DBG_PRINT, "(GRADING#S X.Y)\n");
```

  - `#` is the kernel assignment number, can only be `1`, `2`, or `3`
  - `S` is the section number of the grading guidelines
    - for section (A) of the grading guidelines, you must use the corresponding `X` and `Y`
      - read the requirements very very carefully!
    - for other sections of the grading guidelines:
      - `X` is a subtest number (applicable only when the subtest can be run separately with a shell-level command)
      - never use `X.Y`
  - you must *format it exactly* according to spec or you will lose a lot of points
    - read the requirements very very carefully!
  - when in doubt, ask the instructor!



"Conforming `dbg()` Call"

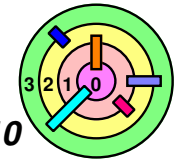
- There are only *two* reasons to make a "conforming `dbg()` call"
  1) get credit for an item in *section (A)* of the grading guidelines
     - must use this form (must specify `#`, `X`, and `Y`):

```
dbg(DBG_PRINT, "(GRADING#A X.Y)\n");
```

  2) get credit for *SELF-checks*
     - must use this form (use `X` if applicable):

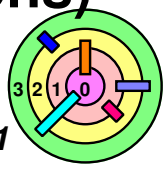
```
dbg(DBG_PRINT, "(GRADING#A)\n");
dbg(DBG_PRINT, "(GRADING#B X)\n");
dbg(DBG_PRINT, "(GRADING#C)\n");
...
```

     - `dbg(DBG_PRINT, "(GRADING#A)\n");` means start and stop the kernel (without doing anything else)
- when in doubt, more is better than less!



"SELF-checks"

- As part of our requirements, you must ***not*** put/leave useless stuff in your kernel (i.e., don't leave trash in the kernel)
  - ***every code sequence must be traversable***
    - a code sequence is a block of code that does not contain conditionals or gotos and has only one entry point at the top
      - a code path is a path your code ***may*** take, i.e., there is a way to execute your code along that path
    - if a code sequence is not traversable, you must ***delete*** it
      - ***must not leave useless code in the kernel!***
    - if you cannot demonstrate that there is way to get to it, you must ***remove*** it or we will take points off
- This is referred to as "SELF-checks" in the spec
  - when you are confused about "SELF-checks", please come back and read this slide again to remind yourself why we are doing "SELF-checks" (and hopefully, that will answer your questions)



"SELF-checks"

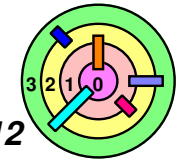
- What's useless code?

```
if (cond) {
  /* code seq */
}
```

  - if `cond` can never be true, then `code seq` is *useless*
  - to demonstrate that `code seq` is useful, you must tell the grader *which test to run* to reach the *END* of `code seq`
    - must add "conforming `dbg()` call" at the *END* of `code seq`

```
if (cond) {
  /* code seq */
  dbg(DBG_PRINT, "(GRADING#S X)\n");
}
```

      - `#` is assignment number: `1`, `2`, or `3`
      - `S` is section number: `B`, `C`, `D`, ...
      - `X` is subsection number (if applicable)
    - just need to tell the grader *one way* to get there



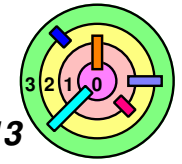
Must Use Static Analysis For SELF-checks

- To determine where you must add "conforming dbg() call", you must perform *"static analysis"* of your code (i.e., does not depend on how your code actually runs)

```
while (cond) {
  /* seq1 */
}
/* seq2 */
```

  - you may argue that the first time the while loop is executed, `cond` will be true, then later on, `cond` will be false, so you just need a "conforming dbg() call" at the end of `seq2`
    - that would be an incorrect analysis because it depends on how your code would execute
    - in this case, you must put a "conforming dbg() call" at the END of `seq1` and at the END of `seq2`

- Please read the spec for details!



"Conforming `dbg()` Call" Requirement

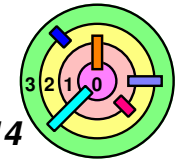
Example from spec:

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

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

A "sequence" is a list of C statements *not* containing conditionals, gotos, or a label which is the target of a goto statement

- if you are not sure, you can make a "conforming `dbg()` call" at the *END* of *every code sequence*
  - you cannot go wrong with more!



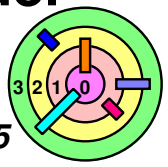
Backing Up Your Work & Collaboration

- You *have to* have a plan to *backup your code* and backup routinely
  - if you lose your work, no one can recover your files
  - can use *private* DropBox / iCloud / Microsoft cloud
  - or use a *private* `bitbucket` (must *not* use `github`)
    - share it among your team members only

- One simple way to backup your work
  - at the end of each day, do:

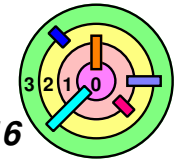
```
% make backup
```

    - it will tell you the name of the backup file
    - if you have a "`Shared-ubuntu`" shared folder in your home directory, your backup file will be copied there
      - if not, your backup file will be in the current directory and you should copy it into a shared folder
  - use your host's cloud backup facility to back up this file/folder



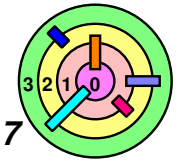
Backing Up Your Work & Collaboration

- You should use `git` to collaborate among project partners
  - read "Pro Git" (a free online book, one of our textbooks)
- But you need a shared repository in the cloud to collaborate with your teammates
  - there are free `git` repositories on the web
    - unfortunately, most of the free ones are required to be visible by the world - you must *not* use these
      - on `github.com`, private repository automatically becomes public after 2 years (if you don't pay)
      - this is why you must *not* use `github.com`
    - you can use `bitbucket.org` but you need to make sure that your files remain private
  - apply for *free academic account* on `bitbucket.org` to share with teammates
    - make sure your projects are truly *private*



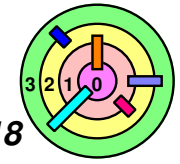
Backing Up Your Work & Collaboration

- If you have two people working on the same file and then update the repository one after another
  - `git` will attempt to merge the changes, but it may not be what you want
  - may be it's best to *coordinate* and not have two pepole modifying the same file



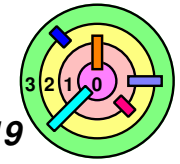
Early and Late Policies

- Same early submission policy as warmup projects
- Similar late submission policy as warmup projects
  - except that kernel 1 can be submitted by the kernel 2 deadline and get a 50% deduction
  - similarly, kernel 2 can be submitted by the kernel 3 deadline and get a 50% deduction
  - kernel 3 has regular late submission policy



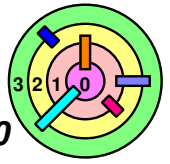
Extra Credit

- You can get *extra credit* for posting *timely* and *good/useful* answers to the class Google Group in response to questions posted by other students regarding *kernel* programming assignments
  - the maximum number of extra credit points you can get is *10* points for each of the kernel assigments (on a 100-point scale)
  - "timely" means *within 8 hours of the original post*
    - if you don't *"reply"* to another student's post, you are not eligible for this type of extra credit
  - if you just repeat what others are saying, you are being "helpful" but what you post will not be "useful" since it's already been said



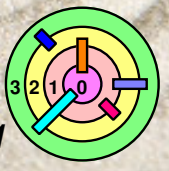
How Do You Start?

- Definitely start with the documentation, spec, and kernel FAQ
- Read code, read lots and lots of code
  - try things out and see what happens (debugging statements)
  - you need to *absorb* other people's code, make sense of it
    - although you *don't have to understand everything*
  - that's what OS hacking (in the good sense) is all about
    - it's *not* about "implementing an OS"
- It's the *process* that matters
  - *not* the *answers*
  - it's about learning how to figure out *which* 2 or 3 lines (or 20 or 30 lines) of code to insert and *where*
- So, it needs to be experienced
  - you should not expect quick/straight answers
  - this is not an OS hacking class
- Learning to write OS code is like...



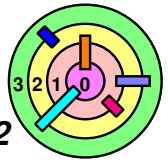
How Do You Start?

- Definitely start with the documentation, spec, and kernel FAQ
- Read code, read lots and lots of code
  - try things out and see what happens (debugging statements)
  - you need to *absorb* other people's code, make sense of it
    - although you *don't have to understand everything*
  - that's what OS hacking (in the good sense) is all about
    - it's *not* about "implementing an OS"
- It's the *process* that matters
  - *not* the *answers*
  - it's about learning how to figure out *which* 2 or 3 lines (or 20 or 30 lines) of code to insert and *where*
- So, it needs to be experienced
  - you should not expect quick/straight answers
  - this is not an OS hacking class
- Learning to write OS code is like... *Zen*



Getting Help

- If you have questions about the kernel assignments
  - read documentation, textbook, lecture slides, read more code
  - send me e-mail
    - please understand that neither I nor other teaching staff can *tell you what code to write*!
    - if you ask me if you should set this variable to x or y, I will ask you to try both and figure out what makes more sense!
      - if you send us questions like that, we may simply forward your post to the class Google Group since we cannot tell you what code to write
  - post your questions to the class Google Group
    - your classmates are a great resource!
    - sometimes, we may not immediately answer these questions to give your classmates an opportunity to earn extra credit points
      - we may wait 2 hours



Pitfalls To Avoid

- Your team need to *meet often*
  - once a day is preferred
    - work at the same place at the same time
    - have lots of discussions (and write a fair amount of code)
  - swallow your pride, be honest with your teammates, don't hide your weakness
    - everyone gets the same grade
    - if no one is really good at this (which is not unusual), someone (or more) has to step up
- You don't have to know what every piece of code is doing
  - learn how to *assume* that other code works (until proven otherwise)
    - other code works kind of like what's covered in lectures
  - use "`grep`" to *get an idea* of how a function is used and how a field in a data structure is used