Department of
Computer Science

Viterbi School of Engineering USC University of Southern California

FAQ for Kernel Programming Assignments

Note: (1) This page can change without notice. (2) Slide numbers mentioned may not be exact.

Quick index:

General

Kernel Assignment 1

Kernel Assignment 2

Kernel Assignment 3

IDE

Q: Can I use Eclipse with the kernel assignments?

A: Yes. Thanks to Zhiyi Xu, one of our graders in Spring 2014! He wrote a tutorial on how to make Eclipse work with the kernel assignments (altho like ITS has deleted his account since he is no longer a student).

- Tushar Aggarwal in Spring 2016 suggested the following modification to the above tutorial: "The line for debugging that must be changed file is 127 and not 90. Or better search for $GDB $GDB_FLAGS and comment it out using # character.".

A student from a previous semester, Luis Perez Cruz, also mentioned a link about how to use valgrind with Eclipse.

Another student from a previous semester, Kai Lu, also mentioned a link about how to install Valgrind plugin in Eclipse and some useful hints.

Another student, Tianlei Xu, also provided an updated instruction on how to use Eclipse Neon.3 with `weenix`.

The instructor knows nothing about Eclipse (your kernel assignments were designed to use `gdb` as the debugger) so please don't ask him about Ec are having trouble with Eclipse, please feel free to start a discussion in the class Google Group. There's a much better chance that your classmates to fix problems with Eclipse.

Q: What are `cscope` and `ctag` good for?

A: Two students, Vishali Somaskanthan and Vandhana Somaskanthan, found that using `cscope` and `ctags` with `vim` editor to navigate the kernel assi easily without even having to install Eclipse for that purpose. She wrote a short tutorial on how to use `cscope` and `ctags` for our kernel assignme

---

Q: Can I see source code in the same window as `gdb`?

A: Yes. Weichen Zhao, one of your classmates, mentioned the GDB Text User Interface (TUI) page. (The notation "C-x" means <Cntrl+X>. So "C-x <Cntrl+X> followed by pressing the "2" key.) You can even open up a text window inside `gdb` to see the corresponding assembly code!

An alternative way is to use the "**`layout`**" `gdb` command. Type "**`help layout`**" in `gdb` to see your choices.

My experience with different layout modes in `gdb` is that, sometimes, `gdb` would appear to be stuck because if the strage graphics it's using. There recommendation is that you should minimize the use of these fancy layout modes. If you want to use it, please understand that you may have to k restart the entire debugging section when `gdb` freezes or starting to do weird stuff.

Q: How can I get the "`kernel`" `gdb` commands mentioned in section C.2.3 of the weenix documentation to work?

A: You need to uncomment the 4 lines immediately follow the "# XXX disabled until gdb version checks are written" line in the "`weenix`" command After you do that, if you start `weenix` with `gdb`, most likely, you will get a "No module named weenix" error. You should then do:

```
pushd python; tar cvf - weenix | (cd /usr/share/gdb/python; sudo tar xvf -); popd
```

Then start `weenix` again with `gdb` and type "`help kernel`" to see what "`kernel`" gdb commands are available. Thanks to your classmate Ye Tian the solution to this problem.

Q: How can I figure out the meaning of a machine instruction?

A: At the Intel 80386 Programmer's Reference Manual web site, there is a link labeled "Instruction Set". Click on it to see a list of all x86 machine ir You should also click on a particular machine instruction to see exactly what it does.

Q: Ubuntu is running slow (and it was running fine when I started it). If I run `top`, I can see that my memory is used up. Can I free up unuse in Ubuntu"?

A: You can only free up unused memory if there is actually unused memory! So, if you are actually using too much memory (i.e., running too many then there is nothing that can be done. In general, don't run too many things (like anything that's not class-related) in Ubuntu. If you have a web b running, don't open too many tabs. Assuming that you actually have unused memory, here are a couple of commands you can try to free things up

```
sudo sysctl -w vm.drop_caches=3
sudo sync && echo 3 | sudo tee /proc/sys/vm/drop_caches
```

You can put the above in a shell-script and put it in your `~/bin` directory for easy access. To see how much memory you are using, try my "`show-` program (it prints the same two lines as the output of `top`).

Q: When I run gdb to debug the kernel, Ubuntu slows to a crawl, what can I do?

A: Running gdb shouldn't make things run much slower! So, it's probably related to your setup.

If you are running VirtualBox, it's very important to check the "Enable 3D Acceleration" checkbox. In the VirtualBox's Settings screen, click on I the left and you should see a checkbox labeled "Enable 3D Acceleration". Make sure you have it checked (and don't check the 2D checkbox).

If that checkbox is already checked, may be you are using the wrong amount of memory. You should try assigning 1GB of memory to your virtua If you give it too much, you have no memory to run the rest of your Mac and that's no good. If you give it too little, your Ubuntu can run slow.

You also need to run a few things as possible inside Ubuntu. Pretty much the ONLY things you should be running in Ubuntu are things related to assignment. So, no web browser, no music player, no games.

Q: How should I use "`CS402INITCHOICE`" in "`Config.mk`"?

A: By default, your kernel starts one way. If you have **additional** tests for the grader to run, you should use "`CS402INITCHOICE`" in "`Config.mk`" to s test to run by **conditionally compiling your kernel**. This way, when you need to run a different test, you don't have to change the kernel code; yo to change "`Config.mk`", then do "`make clean; make`".

Unfortunately, the syntax is a little bit awkward because "`CS402INITCHOICE`" can only take on a single **numerical value**. Below is an example of can put inside your `initproc_run()` to select which user space program to run from the kernel in kernel 3. You can use an analygous approach fo and kernel 2.

```
#if CS402INITCHOICE > 0
  #if CS402INITCHOICE > 1
    #if CS402INITCHOICE > 2
      #if CS402INITCHOICE > 3
        #if CS402INITCHOICE > 4
          kernel_execve("/usr/bin/memtest", argvec, envvec);
        #else
          kernel_execve("/usr/bin/vfstest", argvec, envvec);
        #endif
      #else
        kernel_execve("/bin/uname", argvec, envvec);
      #endif
    #else
      kernel_execve("/usr/bin/fork-and-wait", argvec, envvec);
    #endif
  #else
    kernel_execve("/usr/bin/hello", argvec, envvec);
  #endif
#else
  kernel_execve("/sbin/init", argvec, envvec);
#endif
```

If you read the above code carefully, you should be able to see that if `CS402INITCHOICE=0`, `kernel_execve("/sbin/init",...)` will be invoked `CS402INITCHOICE=1`, `kernel_execve("/usr/bin/hello",...)` will be invoked. If `CS402INITCHOICE=2`, `kernel_execve("/usr/bin/fork-and-wait",...)` will be invoked. And so on. Although this example shows what you can do for kernel 3, you can easily adapt the code for kernel 1 ar

Finally, if you decide to use `CS402INITCHOICE`, please document its usage clearly in your README file.

Q: Can you suggest how I can use `bitbucket.com` to collaborate with my teammates?

A: Thanks to Pratheek Bhat who took CS 402 in Spring 2017. He wrote this two-page document (in PDF) to show the steps of how to collaborate ov `bitbucket.com`.

Please make sure that you make your projects private and share it with your teammates explicitly when you use bitbucket.com. Making your CS visible to the world is a violation of USC Student Conduct Code.

Q: **Can I put the source code inside a shared folder in Windows so I can use my editor on Windows?**

A: In the spec, it was mentioned that if you run the first command there inside a **shared folder in Windows**, you will get a bunch of errors like the f

```
tar: .../weenix/kernel/test/vfstest/vfstest.c: Cannot create symlink to ‘../../../user/usr/bin/tests/vfstest.c’: Pr
tar: .../weenix/kernel/include/test/usertest.h: Cannot create symlink to ‘../../../user/include/test/test.h’: Proto
tar: .../weenix/user/lib/ld-weenix/elf.h: Cannot create symlink to ‘../../../kernel/include/api/elf.h’: Protocol er
tar: .../weenix/user/usr/bin/tests/mm.h: Cannot create symlink to ‘../../../../kernel/include/mm/mm.h’: Protocol er
tar: .../weenix/user/usr/bin/tests/page.h: Cannot create symlink to ‘../../../../kernel/include/mm/page.h’: Protoco
tar: .../weenix/user/include/limits.h: Cannot create symlink to ‘../../kernel/include/limits.h’: Protocol error
tar: .../weenix/user/include/dirent.h: Cannot create symlink to ‘../../kernel/include/fs/dirent.h’: Protocol error
tar: .../weenix/user/include/ctype.h: Cannot create symlink to ‘../../kernel/include/ctype.h’: Protocol error
tar: .../weenix/user/include/stdarg.h: Cannot create symlink to ‘../../kernel/include/stdarg.h’: Protocol error
tar: .../weenix/user/include/weenix/config.h: Cannot create symlink to ‘../../../kernel/include/config.h’: Protocol
tar: .../weenix/user/include/weenix/syscall.h: Cannot create symlink to ‘../../../kernel/include/api/syscall.h’: Pr
tar: .../weenix/user/include/fcntl.h: Cannot create symlink to ‘../../kernel/include/fs/fcntl.h’: Protocol error
tar: .../weenix/user/include/lseek.h: Cannot create symlink to ‘../../kernel/include/fs/lseek.h’: Protocol error
tar: .../weenix/user/include/errno.h: Cannot create symlink to ‘../../kernel/include/errno.h’: Protocol error
tar: .../weenix/user/include/sys/stat.h: Cannot create symlink to ‘../../../kernel/include/fs/stat.h’: Protocol err
tar: .../weenix/user/include/sys/mman.h: Cannot create symlink to ‘../../../kernel/include/mm/mman.h’: Protocol err
tar: .../weenix/user/include/sys/utsname.h: Cannot create symlink to ‘../../../kernel/include/api/utsname.h’: Proto
tar: .../weenix/user/include/sys/types.h: Cannot create symlink to ‘../../../kernel/include/types.h’: Protocol erro
tar: Exiting with failure status due to previous errors
```

The reason you are getting these errors is because Windows does not support "symlink". Therefore, the above "symlinks" were not created and yo able to compile the weenix kernel. The reason "symlinks" were used was to ensure that these files are identical. One way to solve the problem is t copies of these files using these commands:

```
pushd weenix-assignment-3.*.0
pushd weenix/kernel/test/vfstest; cp ../../../user/usr/bin/tests/vfstest.c vfstest.c; popd
pushd weenix/kernel/include/test; cp ../../../user/include/test/test.h usertest.h; popd
pushd weenix/user/lib/ld-weenix; cp ../../../kernel/include/api/elf.h elf.h; popd
pushd weenix/user/usr/bin/tests; cp ../../../../kernel/include/mm/mm.h mm.h; popd
pushd weenix/user/usr/bin/tests; cp ../../../../kernel/include/mm/page.h page.h; popd
pushd weenix/user/include; cp ../../kernel/include/stdarg.h stdarg.h; popd
pushd weenix/user/include/weenix; cp ../../../kernel/include/api/syscall.h syscall.h; popd
pushd weenix/user/include/weenix; cp ../../../kernel/include/config.h config.h; popd
pushd weenix/user/include; cp ../../kernel/include/fs/lseek.h lseek.h; popd
pushd weenix/user/include; cp ../../kernel/include/fs/dirent.h dirent.h; popd
pushd weenix/user/include; cp ../../kernel/include/fs/fcntl.h fcntl.h; popd
pushd weenix/user/include; cp ../../kernel/include/errno.h errno.h; popd
pushd weenix/user/include; cp ../../kernel/include/ctype.h ctype.h; popd
pushd weenix/user/include; cp ../../kernel/include/limits.h limits.h; popd
pushd weenix/user/include/sys; cp ../../../kernel/include/types.h types.h; popd
pushd weenix/user/include/sys; cp ../../../kernel/include/fs/stat.h stat.h; popd
pushd weenix/user/include/sys; cp ../../../kernel/include/mm/mman.h mman.h; popd
pushd weenix/user/include/sys; cp ../../../kernel/include/api/utsname.h utsname.h; popd
popd
```

If you decide to do this, you need to be very very careful and not modify these files (since the copy of the file will not be modified automatically a cause inconsistencies in your kernel).

Pointers

Q: **Why do I have a pointer that points to 0xbbbbbbbb?**

A: I think **QEMU** initializes **all** memory (i.e., not just pointers or what pointers pointn to) to be 0xbb to make it easier for you to detect uninitialized So, if you see 0xbbbbbbbb, it means that most likely you are looking at uninitialized memory (as far as YOUR code is concerned).

Please note that I didn't say weenix, I said **QEMU**.

Slab allocator

Q: **How can I figure out how to use a slab alloctor function?**

A: The slab allocator functions are:

```
slab_allocator_t *slab_allocator_create(const char *name, size_t size);
void *slab_obj_alloc(slab_allocator_t *allocator);
void slab_obj_free(slab_allocator_t *allocator, void *obj);
```

What could they possible mean? Well, you need to go back to the end of the lecture on "dynamic storage allocation" in Ch 3. A slab allocator can allocate object of a specific type. Therefore, to allocate a process control block, you need a process control block allocator; to allocate a thread co you need a thread control block allocator, etc. Guess which one of the above function would give you a slab allocator of a specific type? Once yo allocator, which knows only how to allocate object of one type, you just need to ask it to allocate one such object for you by calling one of the thr above. Again, guess which one you should use?

list in weenix

Q: **I'm really confused with list iteration macro. What does member variable is used for? Is it simular what how we used *obj in CS402list?**

A: As mentioned in slide 11 of the Warmup #1 lecture slides, there are two types of lists. My402List is of type (2). Well, "kernel/include/util/list.h" i (1).

Let's look at list traversal in the comment of "kernel/include/util/list.h":

```
To iterate over a list,
   list_link_t *link;
   for (link = list->l_next;
        link != list; link = link->l_next)
      ...
```

So, the "list" is like the anchor in My402List (but not the same).

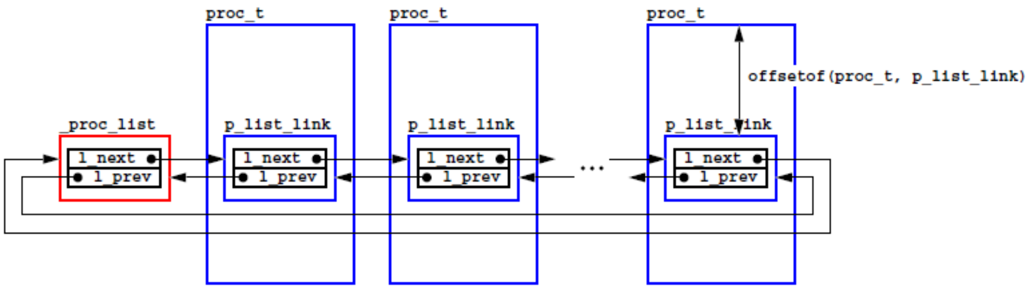
In My402List, we have a My402ListElem inside My402List and called it the **anchor**. Every element on the list is a My402ListElem and its "obj" to an object. So, it's like an object is "hanging" off of a My402ListElem. Think of a clothesline. Every pair of clothesclips is a My402ListElem an whatever you want on it.

The list in "kernel/include/util/list.h" is the opposite of My402List! Instead of hanging an object off of an list element, the object would "contain" element (which is called a "link"). So, if you have an object that you want to put it in a list, you add a "link" field (of type `list_link_t`) into this type of this object is referred as "type" in the iterator code. The name of the "link" in this object type is referred as "member" in the iterator code.

Let's look at an example of how the list is used in `prc_lookup()` of "kernel/proc/proc.c". We have:

```
proc_t *p;
list_iterate_begin(&_proc_list, p, proc_t, p_list_link) ...
```

The list looks like (where `_proc_list` is a variable, `proc_t` is a data type, and `p_list_link` is the name of a field in `proc_t`):



Since `p_list_link` is a field in `proc_t`, if we have a pointer to the `p_list_link` member of `proc_t`, we can subtract `offsetof(proc_t, p_list` get the pointer to the corresponding `proc_t`. If we then typecast this pointer to `proc_t` and return it as `p`, at each iteration of the loop, the variable magically point to the next `proc_t`. (Please look at the implementation of `list_iterate_begin()` and `list_item()` in "kernel/include/util/list.h' understand exactly what I meant there.)

By the way, the above picture should remind you about our warmup 1 assignment. The list in our warmup 1 assignment is different from the list i because an object stored in a list can be in multiple lists simultaneously. But our doublly-linked circular list part does look like the picture above ' node in red is analogous to the `anchor` in `My402ListElem`. Another difference is that in warmup 1, we call the list element pointed to by the `next` anchor the "head of the list (or, the first element on the list)" and the list element pointed to by the `prev` field of the anchor the "tail of the list (or, element on the list)". The `list` in `weenix` uses the opposite definition.

C library

Q: **Where are all the c-string manipulation functions, such as `strlen()`, in the kernel?**

A: You cannot use **`libc`** in the kernel. All the string functions are implemented in `kernel/util/string.c`, and they are:

```
int    memcmp(const void *cs, const void *ct, size_t count);
void  *memcpy(void *dest, const void *src, size_t count);
int    strncmp(const char *cs, const char *ct, size_t count);
int    strcmp(const char *cs, const char *ct);
char  *strcpy(char *dest, const char *src);
char  *strncpy(char *dest, const char *src, size_t count);
void  *memset(void *s, int c, size_t count);
size_t strnlen(const char *s, size_t count);
size_t strlen(const char *s);
char  *strchr(const char *s, int c);
char  *strrchr(const char *s, int c);
char  *strstr(const char *s1, const char *s2);
char  *strcat(char *dest, const char *src);
~~char  *strdup(const char *s);~~
char  *strtok(char *s, const char *d);
```

There are **ALL** the string utility functions that weenix need! If it's good enough for weenix (a full OS), it's probably good enough for all of us wh applications. At the user level all these functions are part of the C library. I know that some people like to use the latest and greatest functions ava whatever system they are working on. But if you do that, you will make your program not portable. So, for the rest of your Masters studies, if you make your program portable, you should just stick to this set of functions! Of course, you need to get to know these functions **really well** first.

Q: **I need a random number generator and the `weenix` code doesn't have one. What should I do?**

A: You can write your own. Here's something you can use:

```
static int rand_x = 987, rand_y = 654, rand_z = 3210;

static int random_int()
{
    rand_x = ( rand_x * 171 ) % 30269;
    rand_y = ( rand_y * 172 ) % 30307;
    rand_z = ( rand_z * 170 ) % 30323;
    double n = ((((double)rand_x)/30269.0) + (((double)rand_y)/30307.0) + (((double)rand_z)/30323.0)) * 100;
    return (int)n;
}
```

`weenix` wiki at Brown University

Q: **Can I use the `weenix` wiki at Brown University?**

A: The `weenix` wiki at Brown University is part of a course currently taught by Prof. Tom Doeppner. You are free to look at their web site, but pleas following in mind... Even though it looks like 3 of their kernel assignments are the same as ours, I have **no idea** if they are the same. You have to OUR assignments. Whatever you see on their web site MAY have NOTHING to do with our assignments. You **cannot argue the validity of you implementation** based on anything on the Brown University web site.

There's also a Hacker's Guide on the Brown University site. The information there may be out-dated.

QEMU

Q: **When I start `weenix`, I get these error messages about KVM. Should I worry about them?**

A: It's not a problem if you see the following when you start `weenix`:

```
{bc-VirtualBox:bc}[1] ./weenix -n
/usr/bin/qemu-system-i386
open /dev/kvm: No such file or directory
Could not initialize KVM, will disable KVM support
qemu-system-i386: pci_add_option_rom: failed to find romfile "pxe-rtl8139.bin"
...
```

These error messages are from QEMU and they do **not** affect our assignments. Please ignore these KVM related error messages. If these message both you, you can install `kvm-pxe`:

```
sudo apt-get install kvm-pxe
```

Q: **Why am I getting a "could not open 'kernel/weenix.iso'" error messages?**

A: It's probably because you have not installed all the required software for doing the kernel assignments.

When you do "`make`", you should see the following near the end of "`make`".

```
xorriso 1.4.2 : RockRidge filesystem manipulator, libburnia project.

Drive current: -outdev 'stdio:weenix.iso'
Media current: stdio file, overwriteable
Media status : is blank
Media summary: 0 sessions, 0 data blocks, 0 data, 5899m free
Added to ISO image: directory '/'='/tmp/grub.eYzVuV'
xorriso : UPDATE : 277 files added in 1 seconds
Added to ISO image: directory '/'='/home/william/cvs/bc-src/cs402assign3/OLD/weenix-assignment-3.1.0/weenix/kernel/
xorriso : UPDATE : 281 files added in 1 seconds
xorriso : NOTE : Copying to System Area: 512 bytes from file '/usr/lib/grub/i386-pc/boot_hybrid.img'
ISO image produced: 2963 sectors
Written to medium : 2963 sectors at LBA 0
Writing to 'stdio:weenix.iso' completed successfully.
```

If you did not see this, it's probably because `xorriso` is missing.

`gdb`

Q: **How come I cannot single step inside `sched_switch()`?**

A: By default, `gdb` thinks that it's debugging an application program, running a regular thread. But inside `sched_switch()`, you are switching from c another thread and that confuses `gdb`! So, either you switch to single-step assembly code and use the "`si`" `gdb` command, or you set a breakpoint i `sched_switch()` and use the "`cont`" `gdb` command to get there. Be careful with `next` there!

Processes

Q: **Can only a process's parent kill the process?**

A: For kernel assignments 1 and 2, everything you are doing runs in the kernel. The kernel is very powerful and have no restrictions! It's YOUR ker do anything bad!

Q: **Does the `proc_list()` function really returns a list of running processes?**

A: In "`kernel/proc/proc.h`", it says the following right above the `proc_list()` function:

```
Returns the list of running processes.
```

I'm not 100% sure (since this code is written by the people at Brown), but I think it's a incorrect comment. I think it should say that it returns a lis processes (running and exited).

Q: **The only field in the `proc_t` data structure that does not have a description is `p_pagedir`. What should I do with it?**

A: As mentioned in lectures, every process has a page table. `weenix` uses a multilevel page table scheme where the first level page table is called a **p directory** (and an entry in the page directory points to an actual page table). Each process keeps track of its page directory data structure in the `p_` field in its `proc_t` data structure. When you create a process, you should allocate a page directory data structure by callling `pt_create_pagedir(`

Actually, all kernel processes **can share the same page directory data structure** since the kernel is basically a giant process. (Again, we use ker processes as a data structure to group related kernel threads together.) Instead of calling `pt_create_pagedir()`, you can just call `pt_get()` to sha default kernel page directory. However, sharing is **not** a good way to go because in kernel assignment 3, you will see that user processes will also `proc_t` data structure. So, you might as well create a new page directory data structure for each kernel process in this assignment, even though th contain the same information.

Q: **What is `p_wait` in `proc_t`? The comment in the header file says that it is a queue for wait(2). What does that mean? Is it "terminated children" list in Ch 1 (a simple OS)?**

A: The `wait(2)` system call is used by a process to wait for one of its child processes to die (see "`man wait`"). In `weenix`, the parent process should s `p_wait` queue to wait for one of its child processes to die.

As it turns out, `weenix` does **not** use "terminated children" list in a process control block. When a child process dies, its process control block doe be moved to a special list.

Q: **Can I see a list of processes in `gdb`?**

A: Assuming that your process data structure is in good shape, you can use either of these `gdb` commands:

```
kernel proc
kernel info proc_list_info
```

These are "`kernel gdb commands`". If you see things like:

```
ImportError: No module named 'weenix'
```

or:

```
Undefined command: "kernel".
```

it's probably because you didn't run the last command (that starts with "`pushd python;`") in the first paragraph of when you setup the kernel assig You only have to run that command once.

The list of processes in weenix is stored in a global variable called `_proc_list`. The "`kernel list`" `gdb` command can be use to print the conten Therefore, if you want to list all the `proc_t` data structures on the `_proc_list` by following the `p_list_link` field inside each `proc_t`, you can u following `gdb` command:

```
kernel list _proc_list proc_t p_list_link
```

Q: **If I kill a process with `proc_kill(status)`, does `status` become the exit status of the killed process?**

A: If a kernel thread wants to ignore what another kernel thread wants it to do, it's free to do so. There's nothing that requires the target thread use st exit status. So, if it wants to use some other exit status, it should be able to do so. In most cases, a thread should use its own exit code and ignore t code used in `proc_kill()`.

Threads

Q: **Are kernel threads very different from pthreads?**

A: They sure are! Pthreads are for **user space programs**. For kernel assignments 1 and 2, everything you are doing is part of the kernel and weenix k threads are quite different from pthreads. So, you need to expect that all the processes and threads in the kernel are somehow working together. T be no surprises. A kernel thread cannot be "accidentically" sitting in the wrong place or in the wrong state. You have control over EVERTHING.

Please also keep in mind what the weenix documentation says. We are implementing a **non-preemptive kernel** and there is only **one processor**. thread doesn't want to give up the processor, there is no way to make it give up the processor. So, when you try to cancel a kernel thread X, what c thread X be doing?

Q: **Is kernel thread cancellation very different from pthreads cancellation?**

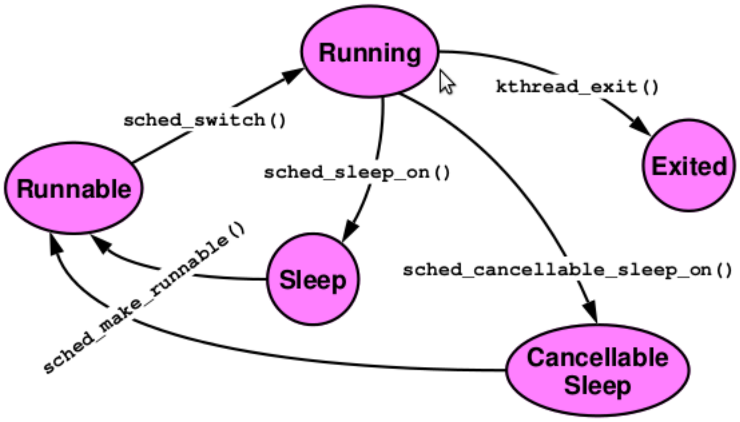
A: The idea of cancellation in weenix kernel is somewhat similar to cancellation in pthreads. Some differences are, (1) there is no preemption in wee (2) there is no cleanup routines for a weenix kernel thread; (3) if a weenix kernel is at a cancellation point, it doesn't die there; and so on.

For a weenix kernel thread, it enters an cancellation point by calling `sched_cancellable_sleep_on()`. Since a kernel thread cannot die in a canc point, whether it go canceled or not is noted in the return value of `sched_cancellable_sleep_on()`.

When you cancel a kernel thread in weenix, if it's not at a cancellation point, cancellation becomes pending (just like in pthreads). If that thread la cancellation point, it should return immediately to indicate that it has been canceled. On the other hand, if that kernel thread is already sitting in a cancellation point, it should be made to leave the cancellation point with a return code that indicates that it hsa been canceled.

Q: **Are kernel thread state transitions very different from what was in lecture slides?**

A: They sure are! The picture below depicts the `weenix` kernel thread state transitions (state names are slightly modified and the causes of state trans only shown partially).



Please note that, similar to the thread transitions in lecture slides, a `weenix` kernel thread can only get into the `KT_EXITED` state from the `KT_RUN` s

---

Q: **Should we implement MTP (multiple threads per process)?**

A: By default, **`MPT=0`** in `Config.mk`. Please do **NOT** set it to `1` for the kernel assignments! You should only have **one kernel thread in each kernel p**

If you really want to implement MTP, please mention it in the README file that's included in your submission. You will not get any extra credit implementing MTP. Please put any code you write for MTP in the following way:

```
#ifdef __MTP__
[ your MTP code goes here ]
#endif
```

If the grader sets **`MPT=0`** in `Config.mk` (which is the **only** way we will grade), your code must run with only one kernel thread per kernel process.

Q: **What does it mean to** sleep on **and** wake up on **a queue?**

A: This is similar to **condition variables** in Ch 2. Recall that a condition variable (CV) has an associated queue. A thread can call `pthread_cond_wa` **wait for a condition to be signaled/broadcasted** (another term for this is to **wait for an event**). When a thread calls `pthread_cond_wait()`, it g the CV queue and sleeps there. In `weenix`, **sleep on** a queue means the same thing, i.e., the thread gives up the CPU and sleeps in the given queue

In pthreads, if you call `pthread_cond_signal()`, you move one thread from the CV queue to the associated mutex queue. This is known as "**sign condition**" or "**signaling an event**". In `weenix`, **wake up on** a queue means pretty much the same thing, except that you move a thread in the gvie the RunQueue.

Q: **Is the comment about `kt_wchan` in "`kernel/proc/kthread.h`" correct?**

A: The comment about `kt_wchan` in "`kernel/proc/kthread.h`" says:

```
ktqueue_t *kt_wchan; /* The queue that this thread is blocked on */
```

When a thread is sleeping/blocked in a queue, `kt_wchan` should point to that queue. But what about the case when a thread is in the run queue? It' "blocked" any more. (But is it "sleeping"?!) In that case, `kt_wchan` should point to the run queue. So, the comment about `kt_wchan` in "`kernel/proc/kthread.h`" is **not** 100% correct.

May be it's more correct to change that line to:

```
ktqueue_t *kt_wchan; /* The queue that this thread is sleeping on */
```

and consider a thread "sleeping" even when it's in the run queue, waiting for the CPU to become available.

kmutex

Q: **In the lecture slides for section 5.2 of the textbook, it says that the main advantage of having a non-preemptive kernel is that you don't ha implement locking inside the kernel. So, why do we need to implement functions in "`kernel/proc/kmutex.c`" since `weenix` has a non-pree kernel?**

A: Usually, an I/O device can only handle one request at a time. To make sure that I/O operations for a device do not overlap, a queue is used for eac device. If a kernel thread wants to perform an I/O operation, it queues itself on this queue and wait for I/O completion interrupt to move it into th

`weenix` drivers are implemented to use a mutex queue in additional to an I/O queue. A kernel thread would lock the associated `kmutex`, starts an I operation, and go into cancellable sleep. In the I/O completion interrupt, the thread is moved to the run queue. When the thread gets to run again, unlock the associated `kmutex`. All this code belongs to `DRIVERS` and is compiled into the "`kernel/libdrivers.a`" library. If you set a breakpoint `kmutex_lock()` and set `DRIVERS=1` in `Config.mk`, you will see that `kmutex_lock()` is getting called. Unfortunately, you don't have access to the s in `DRIVERS` because `DRIVERS` is used as assignments by other universities.

Q: **What does "these locks are not re-entrant" mean in the comment block for`kmutex_lock()`?**

A: This is in "`kernel/include/proc/kmutex.h`".

In the first lecture slide that talked about **Thread Safety** in Ch 2, I mentioned that in a multi-threading environment, "re-entrant" is the same as "t If something is "thread-safe", it means that you can use it under multi-threading. Therefore, "not thread-safe" means that if you use it under multi it may not work.

Being "re-entrant" is more strict than "thread-safe" because you also need to consider the case where there is only one thread. If there is only one you are in the kernel, what you need to worry about is interrupts. Therefore, if something is "re-entrant", it means that you can use it in a thread A interrupt service routine. If something is "not re-entrant", it means that if you use it in a thread AND in an interrupt service routine, it may not wo

scheduler

Q: What does "wake up all threads running on the queue" mean in the comment block for `sched_broadcast_on()`?

A: This is in "`kernel/include/proc/sched.h`". I think it's a typo. It should say, "wake up all threads **sleeping** on the queue".

kshell

Q: How can I invoke `kshell`?

A: The information is in section 4.6 of the weenix documentation (in PDF). Let's say you want to implement a command called "foo". In your "`kernel/main/kmain.c`", you can add something like the following (just a silly example):

```
#ifdef __DRIVERS__

    int do_foo(kshell_t *kshell, int argc, char **argv)
    {
        KASSERT(kshell != NULL);
        dbg(DBG_TEMP, "(GRADING#X Y.Z): do_foo() is invoked, argc = %d, argv = 0x%08x\n",
                argc, (unsigned int)argv);
        /*
         * Shouldn't call a test function directly.
         * It's best to invoke it in a separate kernel process.
         */
        return 0;
    }

#endif /* __DRIVERS__ */
```

In `initproc_run()`, you can add:

```
#ifdef __DRIVERS__

    kshell_add_command("foo", do_foo, "invoke do_foo() to print a message...");

    kshell_t *kshell = kshell_create(0);
    if (NULL == kshell) panic("init: Couldn't create kernel shell\n");
    while (kshell_execute_next(kshell));
    kshell_destroy(kshell);

#endif /* __DRIVERS__ */
```

Please note that the above two fragments are only compiled if the **`__DRIVERS__`** symbol is defined (i.e., you have set **`DRIVERS=1`** in **`Config.mk`**) be compiled the kernel. It's a good idea to keep it that way because you don't want to accidentically invoke these code if you haven't finished all you code.

Please also note that when you invoke **`dbg()`**, the printout goes to the terminal where you started `weenix` and not inside a `kshell` window.

Q: Should I run `kshell` in a separate process?

A: If you go with the above suggestion, `kshell` will **not** be running in a separate process. It will run inside the `init` process (as a subroutine). You **n** `kshell` in a separate kernel process if you want to pass all the tests in the grading guidelines.

Q: My `kshell` says "Bye" right away before I typed anything, what could be the problem?

A: The `kshell` prints "Bye" when it is destroyed. Most likely, kshell_read() failed to read from the keyboard.

What suppose to happen is that reading the keyboard will lock a device-specific mutex and sleep on the I/O queue (to wait for device interrupt).

When you press a key, you will generate a device interrupt. In the interrupt handler, it will unblock the thread (who is waiting on the I/O queue).

When the thread is unblocked, it will assume that it got a key press. In this case, it was surprised because no key was pressed (and returned that 0 entered from the keyboard). Before it returns, it will unlock the mutex.

From the above, at least two things need to work. One is `mutex_lock()`. The other is sleeping on the I/O queue (or sleeping on any queue) and th woken up. I'm guessing that your problem is not with `mutex_lock()`. Most likely, it's with sleep and wake up.

Try to make sure that you can pass all teh tests in faber_thread_test() without invoking `kshell`? I'm guessing that there are bugs in your sleep and code.

Q: Why is it that sometimes `kshell` ignores the command I typed?

A: If this only happens right after you swtich to the QEMU window, it's probably a QEMU bug. So, don't worry about it. Whenever you switch to th window, always press <ENTER> first before typing a command.

Compiler

Q: How do I use a function in another module not declared in a header file?

A: Remember, `#include "foo.h"` simply replace itself with the content of `"foo.h"`. So, you just need to look for any `.h` file and the corresponding hints.

In a `.h`, it typically contains data structures and function prototypes. Function prototypes are just function declarations with the keyword **`extern`** i them. For example, the last function in "`proc/proc.c`" starts this way:

```
size_t proc_list_info(const void *arg, char *buf, size_t osize)
```

If you want to use this function in, say, "`main/kmain.c`", you can just add the following line in "`main/kmain.c`":

```
extern size_t proc_list_info(const void *arg, char *buf, size_t osize);
```

If a function is declared **`static`**, then you are out of luck because using the above trick would not work. What should you do? Well, take it as a hi are going about it the wrong way (i.e., there must be another way that does not involve calling this function).

Testing

Q: For the kernel 1 assignment, it seems that nothing is generating interrupts. How can I test my interrupt-related code?

A: If everything else is working properly in your kernel 1 implementaion, you can configure `weenix` by setting **DRIVERS=1** in **`Config.mk`**. Then yo code in your `initproc_run()` to invoke **`kshell`**. Please see the kshell FAQ to see what code you need to add to invoke `kshell`.

When you are running `kshell`, you should get keyboard interrupts as you type on the keyboard. If the keyboard does not seem very responsive bu keyboard works fine outside of `weenix`, it means that your interrupt-related code is not perfect yet (probably a race condition somewhere).

Q: I have the following code to check out-of-resource case:

```
p = slab_obj_alloc(proc_allocator);
if (p == NULL) {
    /* I want to check that this code path works */
}
```

How do I make the slab allocator to fail so I can exercise the code path for `p == NULL`?

A: Don't do it that way! Since you did not write the slab allocator, you can assume that it's implemented correctly. When it returns NULL, there's pre nothing you can do. So, just do the following instead:

```
p = slab_obj_alloc(proc_allocator);
KASSERT(p != NULL);
```

Please note that you should not do this for every situation. There are cases where a failure can be recovered. Most of those cases are in kernel 3. F should analyze every case and use `KASSERT()` when appropriate. (You can also make a note for yourself to say that you are using `KASSERT()` for k you will need to change it for kernel 3 because there will be ways to make something fail.)

Q: How can I invoke the tests mentioned in sections (C) and (D) of the grading guidelines?

A: In your "`kernel/main/kmain.c`", you should add:

```
extern void *sunghan_test(int, void*);
extern void *sunghan_deadlock_test(int, void*);
extern void *faber_thread_test(int, void*);
```

Notice that these functions are compatible with a kernel thread's first procedure, you can call `kthread_create()` and pass these functions as the s argument (and pass `0` as `arg1` and `NULL` as `arg2`).

Once you get `kshell` to work, it would be desirable to invoke these tests from kshell commands. In your "`kernel/main/kmain.c`", you should cı command functions for `kshell` to invoke:

```
kshell_add_command("sunghan", my_sunghan_test, "Run sunghan_test().");
kshell_add_command("deadlock", my_sunghan_deadlock_test, "Run sunghan_deadlock_test().");
kshell_add_command("faber", my_faber_thread_test, "Run faber_thread_test().");
```

In each of these funtions you have created, you should create a separate kernel thread with one of the test functions as its first procedure. Since wı doing "multiple threads per process", when you create a new kernel thread, you must run it in a new kernel process.

Also, you must run kernel 1 (and kernel 2) tests "in the foreground" (i.e., **not** in the background). For a command shell, to run a command in the f means that you should wait for the child process to die before returning back to the command shell. You should **not** get the prompt back when yo `kshell` command.

If you cannot get `kshell` to work, you should use `CS402INITCHOICE` in `Config.mk` to select which test to run.

Q: I'm very confused about SELF-checks. Why are we required to do SELF-checks anyway?

A: You probably have put in a lot of non-working code in your kernel when you were experimenting with this and that. I want you to clean out the ju kernel code before you move on to kernel 2! So, I'm requiring that if you want to keep a piece of code, you have to show me that the code is **usef** you show me that a piece of code is useful? Well, you have to tell me what test to run to use that code. Therefore, I'm requiring that every code se must have a "conforming `dbg()` call" in it and you need to tell me what test to run to exercise that code sequence. If the way to exercise that code

to run `faber_thread_test()`, then you would call `dbg(DBG_PRINT, "(GRADING1C)\n")`. This way, I know that you did not leave behind any no
code.

By the way, if the only code you keep in the kernel are what's needed to run all the tests in sections (A) through (D) of the grading guidelines, the
should have no test to run in section (E) of the grading guidelines, i.e., **no code sequence** should be using `dbg(DBG_PRINT, "(GRADING1E)\n")`!
way to look at this is that, if you have a code sequence that were not exercised by running all the tests in sections (A) through (D) of the grading
you should just **delete** that code sequence. This way, you don't have to write any new tests for section (E) of the grading guidelines! (This is actu
preferred way!)

The bottomline is that section (E) of the kernel 1 grading guidelines should be **empty**! If it's not empty, you should have a very good reason for it

---

Q: **What's the best way to add "conforming `dbg()` calls" to cover all code sequences?**

A: Here's what I would do... You need to perform a static analysis of all your kernel 1 code (i.e., when you replace `NOT_YET_IMPLEMENTED()` with yo
find all the code sequences. In a code sequence that doesn't have a "conforming `dbg()` call", I would add something like the following at the end
sequence:

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

Set `GDBWAIT=0` in `Config.mk` and use "`script`" to record the next time you run "`./weenix -n`" then shutdown weenix as soon as you get into ksh
"`exit`" the "`script`" command. Then do:

```
grep "(NEWPATH)" typescript
```

to see which code sequence are visited when you simply start and stop the kernel. Change those lines to use "`(GRADING1A)\n`" and repeat the pro
this time, run faber test and then shutdown weenix. Then all the lines in `typescript` that contains "`(NEWPATH)`" are the ones that are not visited b
and stopping and kernel but are visited when you run faber test. You should then change those lines to use "`(GRADING1C)\n`" and repeat the proce
tests in the grading guidelines. When you are done, you can search your code sequences for the lines that has not be changed (i.e., still containing
`(NEWPATH)`") and decide if you want to delete those code sequences or write new section (E) "SELF-check" tests. (Well, my recommendation is t
the unused code sequences.)

Q: **How do I find all the "code sequences"?**

A: Here's a basic approach to find them. Don't just follow what I said exactly. I do not guarantee that you can find **all** the code sequences this way (it
how you write your code). Think about what I'ms saying and use it as a basic approach. The basic idea is that every right-curly-bracket (explicit
is at the end of a "code sequence"! Again, this approach may not be perfect. It's your job to find all your "code sequences". This approach gives y
point.

Q: **I'm still confsued about SELF-checks, can you give an example?**

A: Let's take `proc_kill()` as an example... Let's say that your `proc_kill()` looks like (I'm not saying that it has to look like this; it's just an exampl

```
void
proc_kill(proc_t *p, int status)
{
    if (something) {
        ... /* code sequence X */
    }
    ... /* code sequence Y */
}
```

`proc_kill()` is not mentioned in section (A) of the grading guidelines. So, the only "conforming `gdb()` call" you must add to `proc_kill()` is to
"all code sequence must be exercised" requirement.

In this very simple example, there are only two code sequences. If "`something`" is true, you will execute code sequence X. Whether "`something`"
not, you will execute code sequence Y.

How the code would actually execute is irrelevant since we are suppose to perform a **static analysis** of the code and the result is that there are tw
sequences and you need to add a "conforming `dbg()` call" at the end of every code sequence. If code sequence X is executed by your kernel whe
faber subtest #8 and code sequence Y is taken by your kernel when you simply start and stop the kernel, then you can change the code to:

```
void
proc_kill(proc_t *p, int status)
{
    if (something) {
        ... /* code sequence X */
        dbg(DBG_PRINT, "(GRADING1C 8)\n");
    }
    ... /* code sequence Y */
    dbg(DBG_PRINT, "(GRADING1A)\n");
}
```

By the way, if "`something`" is always true and the "then" part of the above code is simple and does not return, then the there is only one code pat
above code. In this case, the above code would fail "SELF-check"s since the "`if (something)`" part of the code is "junk" and you are not suppos
"junk" in the kernel. Therefore, you should change it to:

```
void
proc_kill(proc_t *p, int status)
{
    KASSERT(something);
    ... /* code sequence Y */
    dbg(DBG_PRINT, "(GRADING1A)\n");
}
```

But since the grader will simply run faber test in one go, there's really no need to identify which subtest will exercise code sequence X. If you hav kshell commands each of which can run different parts of faber test (I don't see how that's possible, but in case it's possible), then you can include information in the "conforming `dbg()` call". So, the following is also perfectly fine and preferred (for this example):

```
void
proc_kill(proc_t *p, int status)
{
    if (something) {
        ... /* code sequence X */
        dbg(DBG_PRINT, "(GRADING1C)\n");
    }
    ... /* code sequence Y */
    dbg(DBG_PRINT, "(GRADING1A)\n");
}
```

Give this a try and read the output in the terminal to understand why just doing this is sufficient (and you don't have to add more text after "`(GRAD` you make a "conforming `dbg()` call", you will see that module and function information and line numbers are displayed in the output. So, it woul perfectly clear to the grader where such a line gets printed.

Q: **Should I keep "conforming `dbg()` calls" as simple as possible or can I add more descriptive information to such a call?**

A: There are two issues here. One is with the subtests mentioned in the grading guidelines. Let's take `faber_thread_test()` in kernel 1 as an examp corresponds to section (C) of the grading guidelines. The grading guidelines shows that there are 9 subtests. Should you include subtest numbers "conforming `dbg()` calls"? Well, it depends on whether you can run the subtests using different kshell commands or not. If you only have one ksh command to run all the section (C) subtests, then you should simply use:

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

You should only distinguish them and use: `dbg(DBG_PRINT, "(GRADING1C 1)\n")`, `dbg(DBG_PRINT, "(GRADING1C 2)\n")`, ..., `dbg(DBG_PRINT (GRADING1C 9)\n")` if the grader can run the subtests **separately**. Remember, for SELF-checks, you use a "conforming `dbg()` calls" to tell the gr test (i.e., a command the grader can use) to run.

The second issue is about adding "descriptive information" in a "conforming `dbg()` call". Some students like to add additional descriptive strings "\n". This may not be an issue with kernel 1. But when you get into kernel 2 and 3, you would be adding more and more descriptive strings. Pleas understand that every distinct string constant lives at a different memory location inside the data segment. If all these descriptive strings are all di kernel data segment will get bigger and bigger. At some point, the kernel data segment will be so big that weenix will not even start and you may message saying that weenix cannot find a bootable device or some other weird QEMU message! (I don't know if this can actually occur, but I hav suspecion that this indeed can happen.) If you reuse the same strings, the kernel data segment will not increase. Therefore, I really do not recomm additional descriptive information in a "conforming `dbg()` call". When you use a "conforming `dbg()` calls", the function name and line numbers c will be displayed in the printout. So you know exactly where the call is made and there is really no need for additional information.

Starting with the Summer 2017 session, adding descriptive information is **not permitted** if the first argument of `dbg()` is `DBG_PRINT`. Also, if the argument of a `dbg()` call is `DBG_PRINT`, you must use the "conforming `dbg()` call" format in the remaining argument or you will lose points.

Q: **Should I just use `dbg(DBG_PRINT, "(GRADING1B)\n")` for all SELF-checks since almost every piece of code I write in the kernel is execute running kshell commands?**

A: If you type a kshell command that runs faber test, then running that kshell command is considered running 1C.

If you type a kshell command that runs sunghan test, then running that kshell command is considered running 1D subtest 1.

If you type a kshell command that runs deadlock test, then running that kshell command is considered running 1D subtest 2.

If you simply type the "exit" kshell command right after you start weenux, then running that kshell command is considered 1A.

If you type a kshell command that runs one of the test you wrote, then running that kshell command is considered running 1E (and if you have su must document them extensively).

If you type either the "echo" or the "help" kshell command, then it is considered running 1B.

Q: **What is the Two consecutive "conforming `dbg()` calls" all about in section (A) of the grading guidelines?**

A: Very important to distinguish the two types of "conforming `dbg()` calls". One type of "conforming `dbg()` call" is for section (A) of the grading gu The other type of "conforming `dbg()` call" is for SELF-checks (i.e., section (E) of the kernel 1 grading guidelines). Since this question is about th **consecutive "conforming `dbg()` calls"**, it's related to section (A) of the grading guidelines.

For kernel 1, there are two cases requiring **Two consecutive "conforming `dbg()` calls"**. One is for `(A.3.b)` and the other one is for `(A.6.b)`. Le `(A.3.b)` as an example. Your code must look like the following near the top of `kthread_cancel()`:

```
KASSERT(NULL != kthr);
dbg(DBG_PRINT, "(GRADING1A 3.b)\n");
```

Something like this would be fine for a regular item in section (A) of the grading guidelines. But since `kthread_cancel()` is not called if you jus stop `weenix`, you need to tell the grader how to reach this KASSERT() call. The way to do it is to use a back-to-back (or consecutive) "conformin calls".

If the way to reach this KASSERT() is to run `faber_thread_test()`, since `faber_thread_test()` is invoked in section (B) of the grading guidel the way to tell the grader to run the kshell command that corresponds to `faber_thread_test()` is to use EXACTLY the following "conforming c

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

Putting all these together, in order to get credit for `(A.3.b)`, and assuming that running faber_thread_test() will execute that KASSERT(), you mu

```
KASSERT(NULL != kthr);
dbg(DBG_PRINT, "(GRADING1A 3.b)\n");
dbg(DBG_PRINT, "(GRADING1C)\n");
```

The way to read the above 3 lines is as follows. The first line is from `(A.3.b)` of the grading guidelines (you can use a different variable name fo the rest must be EXACTLY the same as written in the grading guidelines). The 3rd line is there so that the grader knows that he/she should run th command that corresponds to `faber_thread_test()`. The 2nd line is there so that the grader can see "`(GRADING1A 3.b)`" on the screen and obse module name and line number printed by `dbg()`. (You can exchange the 2nd and 3rd lines.)

If there are multiple ways to get that KASSERT() code to execute, you can just pick one. Any one.

Capture Output

Q: **How can I capture all the printout that flew by in a terminal?**

A: In Linux/Unix, there is a command called **`script`** and can create a transcript of everything that gets displayed in a termianl into a file named **`type`** you can do the following:

```
script
./weenix -n
exit
```

and **`typescript`** will contain everything that got printed when you run "`weenix -n`". (Please remember that for "`weenix -n`" to work without deb need to have GDBWAIT=0 in `Config.mk`.)

Q: **Is there a way to make so I see less of the stuff that flew by in a terminal?**

A: You can set "`DBG = -all`" in "`Config.mk`" so you see no debugging output at all.

If you just want to see selected types of debugging statement, you should checkout the possible values to use with "`DBG`" in "`Config.mk`" in "`kernel/include/util/debug.h`". For example, to turn on just things in the "General" section, set the following in "`Config.mk`":

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

Overview

Q: **Okay, I'm totally lost. What am I suppose to do in kernel 1 anyway?**

A: I'm going to give a brief overview here. Please keep in mind that we are only doing one thread per process (i.e., **`MTP=0`** in `Config.mk`). Therefore, the word "thread" and "process" interchangeably.

According to the spec, you know that you need to start with the `bootstrap()` function. In `bootstrap()`, you need to create the **idle** process and u `idleproc_run()` as its "first procedure".

The **idle** process will create the **init** process and use `initproc_run()` as its "first procedure". The **idle** process will wait for the **init** process to die **init** process dies, the **idle** process will shutdown the operating system.

The **idle** process has another child process called the **pageoutd** process. Don't worry about it. This process should never wake up until the time th process shuts down the system. At that time, the idle process will wake up the **pageoutd** process so it can self-terminate.

After you have implemented some of the required functions, start testing with an empty `initproc_run()` and make sure you can boot the kernel a shutdown cleanly (i.e., see the "**weenix: halted cleanly!**" message in the terminal where you typed "**`./weenix -n`**"). Do this with `DRIVERS=0` in C

What are the functions you have to write in order to get this far? I think the following functions need to be implemented to get this far (any why y implement them):

- in "`main/kmain.c`":
  - `bootstrap()` - if you don't write code here, your kernel will not go anywhere
  - `initproc_create()` - because `idleproc_run()` calls this function
- in "`proc/proc.c`":
  - `proc_create()` - if you don't write code here, there's no way to create a kernel process
  - `proc_cleanup()` - because `proc_thread_exited()` calls this function to cleanup the process itself
  - `proc_thread_exited()` - because `kthread_exit()` calls this function
  - `do_waitpid()` - because `idleproc_run()` calls this function
- in "`proc/kthread.c`":
  - `kthread_create()` - if you don't write code here, there's no way to create a kernel thread
  - `kthread_exit()` - because when a thread returns from its "first procedure", the thread startup routine will call this function
- in "`proc/sched.c`":
  - `sched_sleep_on()` - because `pageoutd_run()` calls this function
  - `sched_cancellable_sleep_on()` - because `pageoutd_run()` calls this function
  - `sched_wakeup_on()` - if you don't write code here, you cannot wakeup a kernel thread that's sleeping in a queue
  - `sched_broadcast_on()` - if you don't write code here, you cannot wakeup kernel threads that are sleeping in a queue
  - `sched_switch()` - if you don't wirte code here, your thread cannot give up the CPU to run another thread
  - `sched_make_runnable()` - because `idleproc_run()` calls this function

That's a lot of code that needs to be working together just to run the barebone weenix kernel (i.e., start and stop the kernel without doing anything interesting).

At this point, you should be ready to test the code you have implemented. You should start with `faber_thread_test()`. The function prototype c `faber_thread_test()` should tell you that it is meant to be used as a "first procedure" of a kernel thread. So, in your `initproc_run()`, you shoul child process and use `faber_thread_test()` as its "first procedure". The **init** process should wait for all its child processes to die before it return `initproc_run()`.

What should happen when you run `faber_thread_test()`? Well, you need to look at it one test at a time. What you should do is to read the test c `faber_thread_test()` and make an educated guess about what you think should happen. Then run the test code (under `gdb`) and see if that's exac happens. If yes, you move onto the next test. If not, there are a couple of possibilities. It may be the case that your understanding was correct and bug in your code. Just fix your bug and re-test. It may also be the case that your understanding was incorrect (and you need to figure out why that or you don't know what to expect. You should start a discussion in the class Google Group or seek help from the teaching staff.

Let's look at a particular test in `faber_thread_test()` as an example to demonstrate what you need to know in order to test your code. For exam look at the first test in "cancel me test":

```
dbg(DBG_TEST, "cancel me test\n");
start_proc(&pt, "cancel me test", cancelme_test, 0);
/* Make sure p has blocked */
stop_until_queued(1, &wake_me_len);
sched_cancel(pt.t);
wait_for_proc(pt.p);
```

You need to be able to answer the following questions **based only on** the code in "`proc/faber_test.c`" (and not on your implementation becaus implementation may have bugs).

(1) Do you see that 2 threads (and since we are doing MTP=0, that's the same as processes) are involved?
(2) How does the 1st thread enter a cancellation point and wait to be cancelled?
(3) How does the 2nd thread makes sure that the first thread is sitting in a cancellation point before proceeding?
(4) How does the 2nd thread cancel the 1st thread?
(5) How does the 1st thread get out of the cancellation point?
(6) How does the 1st thread knows that it was cancelled?
(7) What does the 1st thread do after it noticed that it was cancelled at the cancellation point?
(8) What does the 2nd thread do to wait for the 1st thread to terminate?

I will not provide the answers to the above questions here. But feel free to discuss this in the class Google Group.

After you pass all the tests in `faber_thread_test()`, do the same thing with `sunghan_test()` and `sunghan_deadlock_test()`. When everything working, you are ready to run kshell in `initproc_run()`. You should add one command for each of `faber_thread_test()`, `sunghan_test()`, a `sunghan_deadlock_test()`. In your `kshell` command, make sure you create a child process and run the corresponding test as its "first procedure `kshell` command should wait for the child process to die before returning back to `kshell` (i.e., run a `kshell` command "in the foreground").

mknod

Q: **In `idleproc_run()`, I am suppose to call `mknod()`. I don't know how to implement `do_mknod()`. What am I suppose to do?**

A: If you do "`man mknod`", it says that `mknod()` makes block or character special files. Well, that sounds filesystem-dependent to me! Therefore, it's p implemented in the actual file system. Since VFS is filesystem-independent, to implement `do_mknod()`, you need to find out if the actual file syste implements its version of `mknod()` and see what parameters it takes. Hopefully, there is a way to generate all the necessary parameters the actual f needs and pass them to the actual file system's version of `mknod()` and then convert the return value to what's required before you return from do_

Many functions you need to implement in VFS are like the above case. So, you need to go through the same analysis and may be end up doing so similar!

Polymorphism

Q: **Section 4.2 of the weenix documentation was about Object-Oriented C. It said something about virtual function table and polymorphism know C++. Is there an easy way to explain polymorphism?**

A: No. (Just kidding.)

Let's take device drivers as an example (not from `weenix`, just in general). You have a DVD device driver and a hard-drive device driver. Both of to implement `read()`. For a monolilthic kernel, you cannot have two functions with the same name. So, what do you do? You have to name them Let's say that their real names are `DVD_read()` and `HD_read()`.

At a high level, when your application calls `read()` to read a file, it shouldn't matter if the file is on the DVD device or on a hard-drive. What you if the file is on the DVD device, your call to `read()` will turn into a call to `DVD_read()`. If the file is on a hard-drive, your call to `read()` will turn to `HD_read()`. So, when you can do is to make the higher level code call `read()` **indirectly** by using a function pointer. If the file is on the DVD c `read()` function pointer will point to `DVD_read()`. If the file is on the HD device, the `read()` function pointer will point to `HD_read()`. So, this is story.

What we end up is an array of function pointers (i.e., `read()`, `write()`, `interrupt()`, etc.) One way to organize it is as follow. As far as the high is concerned, there is a data structure called `disk_driver_t`:

```
#typedef struct disk_driver {
  int (*read_ptr)(int fd, void *buf, int count);
  int (*write_ptr)(int fd, void *buf, int count);
  int (*intr_ptr)(...);
} disk_driver_t;
```

If you have a pointer to `disk_driver_t`, you can call `read()` and `write()` **indirectly**. For your DVD driver, you can use a data structure whose f looks exactly like `disk_driver_t`. For example:

```
#typedef struct DVD_driver {
  int (*DVD_read_ptr)(int fd, void *buf, int count);
  int (*DVD_write_ptr)(int fd, void *buf, int count);
  int (*DVD_intr_ptr)(...);
} DVD_driver_t;
```

If you create an instance of a `DVD_driver_t` and have `DVD_read_ptr` points to `DVD_read()`, `DVD_write_ptr` points to `DVD_write()`, etc., you can `(DVD_driver_t*)` to `(disk_driver_t*)` and the high level code would not know the difference! Similarly, for your HD driver, you can use a dat that looks like:

```
#typedef struct HD_driver {
  int (*HD_read_ptr)(int fd, void *buf, int count);
  int (*HD_write_ptr)(int fd, void *buf, int count);
  int (*HD_intr_ptr)(...);
} HD_driver_t;
```

If you create an instance of a `HD_driver_t` and have `HD_read_ptr` points to `HD_read()`, `HD_write_ptr` points to `HD_write()`, etc., you can **typec** `(HD_driver_t*)` to `(disk_driver_t*)` and the high level code would not know the difference!

So, high level code interacts with the lower level through the use of the "**virtual functions**" in `disk_driver_t`. There is no real instance of `disk_` The only thing **real** are the instances of `DVD_driver_t` and `HD_driver_t`. The `disk_driver_t` data structure is a **polymorphic** data structure beca sometimes it can be `DVD_driver_t` and sometimes it can be `HD_driver_t`.

Q: **If polymorphism is used, how I can find out what I am pointing to?**

A: You cannot at the source code level. But when you are debugging, you can actually see things.

For example, if `vn` is a pointer to a `vnode_t`, the `vn_ops` field is an array of function pointers. In `gdb`, you can do:

```
p vn->vn_ops
```

to see the name of the global variable that implements this array of function pointers. You can also do:

```
p *vn->vn_ops
```

to see the name of the functions inside the array of function pointers. Hopefully, the names of all these functions is enough hints so you know wha polymorphic pointer is pointing to. (I used `vnode_t` as an example, this trick works with other polynormphic pointers.)

Please remember that some of the lower-level functions are implemented by the Brown University people and you don't have the source code for you step into these functions inside `gdb`, you won't be able to do much there. You need to believe that the code there is bug-free.

Q: **How are the polymorphic pointers from VFS to AFS established?**

A: Using the above example, where `vn` is a pointer to a `vnode_t`, `vn->vn_ops` is an array of polymorphic function pointers. For example, `vn->vn_op` would be the polymorphic `read()` function and `vn->vn_ops->write()` would be the polymorphic `write()` function. By looking at the `ramfs` coc should see that if `ramfs` is the AFS we are using, `read()` is performed by `ramfs_read()`. So, the question here is that for kernel 2, where/when/h `>vn_ops->read()` **became** `ramfs_read()`?

If you "`grep`" for "`ramfs_read`", you should see that it's a member of `ramfs_file_vops`, which is the same type as `vn->vn_ops`. Therefore, `ramfs_file_vops` is the "array of function pointers" mentioned before! Since we cannot find how "`ramfs_read`" is used anywhere else in "`ramfs` to "`grep`" for "`ramfs_file_vops`" and see if that's used anywhere. Welll, the only place it's used is in `ramfs_read_vnode()`.

Hmm... What is `ramfs_read_vnode()`? Let's "`grep`" for it. It's inside another array of function pointers called "`ramfs_ops`".

Where is "`ramfs_ops`" used? The only place it's used is in `ramfs_mount()` and it's assigned to `fs->fs_op`. This is not going anywhere! Things se self-contained. Where should we look next?

Well, since `ramfs_mount()` is assigned to `fs->fs_op`, we might as well "`grep`" for "`fs_op`" and see if it's used anywhere. If you do that, you shou it's used in a few places. One of them looks like:

```
kernel/fs/vnode.c:        vn->vn_fs->fs_op->read_vnode(vn);
```

It contains the string "`read_vnode`" and that's what we are looking for! If you edit "`kernel/fs/vnode.c`", you should see that the above line is ne of the `vget()` function! So, you set a breakpoint there. When you get there, you should see that `vn->vn_ops` is `NULL` before that line of code is exe none of the polymorphic pointers are setup). You should step into that line of code to see what happens in the `ramfs` and see that when the `read_` function returns, `vn->vn_ops` is now pointing to something and the function points inside of it are all pointing to `ramfs` functions. Mystery solvec

vnode

Q: **How do you map a vnode to an inode in the `ramfs`?**

A: For a `vnode`, you can use `VNODE_TO_RAMFSINODE(vnode)` to get to the corresponding inode in the `ramfs`.

What happens is that when an inode is created in the `ramfs`, it stores the inode inside the `vn_i` pointer of a `vnode` data structure. If you have a `vnc` just access its `vn_i` to get to the inode. This is how the `ramfs` uses the `vn_i` pointer in a `vnode`.

The `(void*)` pointer in the `vn_i` field of a `vnode` is like our `(void*)` pointer in the `obj` field of `My402ListElem` in warmup 1! In `My402List`, this a "hang" any object on a list.

In a `vnode`, this allows the Actual File System (AFS) to "hang" an inode (or whatever data structure that it sees fit) inside a `vnode`. The `vnode` has what it is (because it's `(void*)`). As a `vnode` is passed into a function implemented in AFS, AFS knows how to typecast the `(void*)` pointer back actual data type and use it inside the function.

p_cwd

Q: **Is it okay that the `pageoutd` doesn't have a current working directory?**

A: The `pageoutd` is created before you set the `p_cwd` for the idle and init processes. So, its `p_cwd` is not set to anything (i.e., it's `NULL`). For kernel 1 a `pageoutd` should never gets woken up once it sleeps on the `pageoutd_waitq`.

vfstest

Q: How can I invoke `vfstest_main()` in "`kernel/test/vfstest/vfstest.c`"?

A: In your "`kernel/main/kmain.c`", you should add:

```
extern void *vfstest_main(int, void*);
extern int faber_fs_thread_test(kshell_t *ksh, int argc, char **argv);
extern int faber_directory_test(kshell_t *ksh, int argc, char **argv);
```

Please note that `vfstest_main()` is compatible with a kernel thread's first procedure. So, you can call `kthread_create()` and pass `vfstest_mai` second argument (and pass `1` as `arg1` and `NULL` as `arg2`).

Please also note that `faber_fs_thread_test()` and `faber_directory_test()` are **NOT** compatible with a kernel thread's first procedure. But th compatible with the 2nd argument of `kshell_add_command()`. Therefore, you can do something like:

```
kshell_add_command("thrtest", faber_fs_thread_test, "Run faber_fs_thread_test().");
kshell_add_command("dirtest", faber_directory_test, "Run faber_directory_test().");
```

Q: The first function in "`vfstest.c`" calls "`mkdir()`"; where can I find "`mkdir()`"?

A: Please look at "`kernel/include/test/vfstest/vfstest.h`". It has the following macro:

```
#define ksyscall(name, formal, actual)  \
    static int ksys_ ## name formal {   \
        int ret = do_ ## name actual ;  \
        if(ret < 0) {                   \
            errno = -ret;               \
            return -1;                  \
        }                               \
        return ret;                     \
    }
```

A few lines later, it has:

```
ksyscall(mkdir, (const char *path), (path))
```

If you scroll down more, it has:

```
#define mkdir(a,b)      ksys_mkdir(a)
```

Putting them altogether, `mkdir()` is `ksys_mkdir()`, and `ksys_mkdir()` calls `do_mkdir()` and set `errno` to the negative value the return code of `d` if the return code of `do_mkdir()` is negative.

Path name resolution

Q: The `ramfs` fails to lookup of an empty string, what should I do?

A: What should happen when you pass `""` to one of the functions in "`namev.c`"? At the begignning of `vfstest_paths()` in "`vfstest`", it has the foll

```
syscall_fail(stat("", &s), EINVAL);
```

It's saying that `stat("")` must fail with the `EINVAL` error code. But **where** should it fail so that you just have to do this once? Should it fail in `ope` `dir_namev()`, or `lookup()`? (Of course, you can handle this special case in all 3 places. But which is the most logical place for it to fail?) The rea is, "Given the implementation of `ramfs_lookup()`, where should it fail?" In order to answer this question, please read the code of `ramfs_lookup` know **exactly** what `ramfs_lookup()` do (it doesn't matter what you think it should do since we change that piece of code).

vfs_syscall

Q: Where are the `fs_op` mentioned in the comment blocks in "`kernel/fs/vfs_syscall.c`"?

A: The comment blocks say something like, "`call the read/write/readdir fs_op`" and none of these virtual functions exists in a file's `f_vnode->fs_op`. The term "`fs_op`" means "file system operations". If you look at the code in "`kernel/fs/ramfs/ramfs.c`", you will see that these functio `ramfs_dir_vops` and `ramfs_file_vops` and these are assigned to a vnode's `vn_ops` in `ramfs_read_vnode()`.

Therefore, the "`fs_op`" mentioned in the comment blocks refers to a vnode's `vn_ops`. I think since these implementation are filesystem-dependent referred as `fs_op` in the comment block.

Q: What's the difference between `open_namev()` and `dir_namev()`?

A: Let's say that the first argument, `pathname`, is "`/x/y/z`", the main difference between these two functions is that `open_namev()` will get you the v while `dir_namev()` will get you the vnode for `y` (and tell you that the last part of the path is "`z`".

Q: For functions that are never called, what should I do about SELF-checks?

A: **If** there are functions are truly never called when you run all the tests mentioned in the grading guidelines, then you have 2 choices.

1. Write new test code to exercise these functions. Then add kshell commands to execute the test code you wrote and document all this in sect the README file to tell the grader how to run your kshell commands.

2. Remove **all** the code you have written for these functions so that they look **exactly** like the way they were in the **pristine kernel source**. If then you have NOT written any code for these functions and therefore you don't have to do (1) for these functions.

Remember, the rule for SELF-checks is that, **if** you write code to replace a `NOT_YET_IMPLEMENTED` function, then you **must** demonstrate to the gra code you wrote is useful.

`dir_namev()` and (const char **)

Q: What am I suppose to do with the 3rd argument of `dir_namev()`?

A: **`dir_namev()`** is declared as:

```
int
dir_namev(const char *pathname, size_t *namelen, const char **name,
          vnode_t *base, vnode_t **res_vnode)
```

It is weird that the strange syntax of **`(const char **name)`** is used. What it really means is that you are not suppose to write into **`**name`**. For exa you do the following and try to compile it:

```
int
dir_namev(const char *pathname, size_t *namelen, const char **name,
          vnode_t *base, vnode_t **res_vnode)
{
    **name = 'a';
    return 0;
}
```

You will get:

```
  Compiling "kernel/fs/namev.c"...
fs/namev.c: In function 'dir_namev':
fs/namev.c:128:1: error: assignment of read-only location '**name'
make[1]: *** [fs/namev.o] Error 1
```

But if you do the following:

```
int
dir_namev(const char *pathname, size_t *namelen, const char **name,
          vnode_t *base, vnode_t **res_vnode)
{
    *name = &pathname[4];
    return 0;
}
```

Then the compiler will not complain. Therefore, a reasonable conclusion is that **`dir_namev()`** should be invoked as follows:

```
const char *name=NULL;
size_t namelen=0;
char *pathname="/s5fs/bin/ls";

dir_namev(pathname, &namelen, &name, ...); /* of course, you should check return code */
```

In this example, when **`dir_namev()`** returns successfully, "name" should be EQUAL to **`&pathname[10]`** and "namelen" should be 2. So, the intent have the "name" pointer POINTS to somewhere IN THE MIDDLE of the "pathname" char BUFFER. That's why "name" is declared as **`(const c *name)`**. This way, after **`dir_namev()`** returns, if you do:

```
name[0] = 'a';
```

which means that you are using "name" to change the content of "pathname", the compiler will not allow it!

You should **NOT** do the following:

```
char name[80];
size_t namelen=0;
char *pathname="/s5fs/bin/ls";

dir_namev(pathname, &namelen, (const char **)(&name), ...); /* of course, you should check return code */
```

and try to COPY "ls" into the "name" buffer.

Q: Can you give more examples of what `dir_namev()` should return?

A: The example given in the comment block right above `dir_namev()` says that if you call `dir_namev("/s5fs/bin/ls", &namelen, &name, NULL &res_vnode)`, the returned `name` should equal `&pathname[10]` and `namelen` would equal 2 (where `pathname` refers to the first argument of `dir_na` doesn't tell you what to do in all the test cases in `vfstest`. The key to understanding `dir_namev()` in the following.

The comment block right above `dir_namev()` mentioned something called "`basename`", which is a Unix/Linux program that takes a string and ret "`basename`" of that string. So, your `dir_namev()` should just do what "`basename`" does (although there are exceptions)! Here are some more exan

If you call `dir_namev("/s5fs/bin/ls///", &namelen, &name, NULL, &res_vnode)`, you should get the same result because running "`basena /s5fs/bin/ls///`" simply gives "`ls`".

What about `dir_namev("/s5fs/bin/ls///../.", &namelen, &name, NULL, &res_vnode)`? Running "`basename /s5fs/bin/ls///../.`" give Therefore, the returned `name` would equal `&pathname[18]` and `namelen` would equal 1.

What about `dir_namev("/", &namelen, &name, NULL, &res_vnode)`? Running "`basename /`" gives "`/`". Turns out this is an exception since in `dir_namev()`, you have to return a "name" and "/" is a delimiter and **NOT** a "name". In this case, you should return an empty "name", i.e., the re would equal `&pathname[1]` and `namelen` would equal 0. I am not sure if this is the **only exception**. The only way to tell is to run `vfstest` and see complains.

What about `dir_namev("///..", &namelen, &name, NULL, &res_vnode)`? Running "`basename ///..`" gives "`..`". Therefore, the returned na equal `&pathname[3]` and `namelen` would equal 2.

Q: Why is `dbg()` messing up my strings?

A: My guess is that this is after you called **`open_namev()`** or **`dir_namev()`**. I think that one of the functions you have implemented is returning a poi local variable or inside a local variable. For example, in **`dir_namev()`**, if you set "`*name`" to point inside a local variable, once **`dir_namev()`** retur stack frame becomes invalid. When you call another function (such as `dbg()`), you will create new stack frames and overwrite the values in the m location where "`*name`" was pointing to.

When a function returns, you need to make sure that you are not returning anything that points to a local variable of the function from which you returning because that local variable becomes garbage as soon as you return. Just becuase it appears to work doesn't mean you should do it!

Here is an example of something you **must not do** in **`dir_namev()`**:

```
    int dir_namev(const char *pathname, size_t *namelen, const char **name, vnode_t *base, vnode_t **res_vnode)
    {
        char buf[MAXPATHLEN];
        ...
        /*
         * copy pathname into buf
         * parse buf to find where "*name" should point to
         * let's say that you decide that *name should point to &buf[20]
         */
        *name = &buf[20];
        /* this is BAD because as soon as you return from dir_namev(), *name points to invalid space beyond ESP */
        ...
    }
```

But since `buf` was meant to have the same content as `pathname`, what you really should do in this example is:

```
        *name = &pathname[20];
```

do_open()

Q: **Should `O_TRUNC` be handled in `do_open()`?**

A: If you do "`grep`", you will see that the only place that uses O_TRUNC is in "`user/bin`". So, you don't need to implement it in kernel 2. Also, if y code there, you will see that `O_TRUNC` is always used together with `O_CREAT` (i.e., when you create a new file, its size should be zero). So, it seems `O_CREAT` implies `O_TRUNC`. In kernel 2, if you handle `O_CREAT` properly, it looks like the implied `O_TRUNC` is taken care of automatically (and you v to worry about it in kernel 3).

Q: **Do I have to handle all combinations of the `O_*` flags in `do_open()`?**

A: I think the best thing to do is to look at the `vfstest` code since you need to pass all the tests in `vfstest`.

If `vfstest` does not test a particular combination of flags, you have 2 choices. (1) Implement the particular combination and add a SELF-check te implement the particular combination (and write a little comment to yourself that you are skipping this on purpose -- this way, your teammate wo implement it)!

By the way, you should not use a giant `switch` statement in your `do_open()` code to handle the flags. You should use some sort of a decision tree then-else statements). Since all these `O_*` flags are masks (i.e., it has one bit on and all the other bits off), you can check if a particular mask bit is using a logical AND in your code. For example, to check if `O_CREAT` is specified in `flags`, you can do:

```
    if ((flags & O_CREAT) == O_CREAT) {
        /* flags has the O_CREAT bit set */
    } else {
        /* flags does NOT have the O_CREAT bit set */
    }
```

Reference counting

Q: **I'm confused! Is there a rule about when to call `vput()` in "`kernel/fs/vfs_syscall.c`"?**

A: The general rule is mentioned in the Reference Counting section in Section 5 of the weenix documentation (in PDF). I just want to add a commer what to do in general.

In the comment block right above a function in "`kernel/fs/vfs_syscall.c`", if you see something that tells you that you will need to use a poly (virtual) pointer and ask the AFS (Actual File System) do perform some operation, you should get into the debugger and observe the value of the count **BEFORE** you make the polymorphic/virtual call. Then observe what happens when the function returns under various conditions (success and see what happened to the reference count. Then think about whether it makes sense that the a reference count is incremented/decremented or then think about whether you need to do something like a `vput()` or not.

For kernel 2, you have the code of AFS (which is the "ramfs"). So, you can actually step into the polymorphic/virtual function and see how things the AFS. When we go to kernel 3, we will switch from the "ramfs" to "s5fs" and you won't have the source code of "s5fs" (you will be using bina "`kernel/libs5fs.a`"). Hopefully, by that time, you have read enough AFS code in "ramfs" that you can guess what's going on in "s5fs". Althoug major difference between "ramfs" and "s5fs"... In "ramfs", you can never get blocked when you make a polymorphic/virtual function call because real disk in a "ramfs" so you never have to "wait for I/O". When we switch to "s5fs", there's a disk there and your thread can get blocked (i.e., sle queue) when you make a polymorphic/virtual function call. So, if you don't set the reference count correctly, some data structure will get dealloca kernel end up writing data into places it's not suppose to (i.e., memory corruption bug) and your kernel will become very very unstable.

It's tricky to get reference counting right. If you over-reference, the kernel won't halt properly. If you under-reference, you will corrupt kernel me using a data structure that's already freed).

Q: **How should I go about debugging reference counting problem?**

A: Thanks to Reid Garcia who took CS 402 in Spring 2015 for coming up with the following suggestions:

1. Try to find ref count issues by looking at the code
    1. Make sure you are maintaining the right reference count for files
        1. Examine all uses of fget and fref and make sure that they have matching fput's in the same function that they are being called i ref count is supposed to be incremented/decremented in that function)
        2. For all functions that do have the reference count modified after they are called, look at every single call of this function, follo and make sure that it is changed back by a local fput or another function call.
    2. Make sure you are maintaining the right reference count for vnodes
        1. Examine all uses of vget and vref and make sure that they have matching vput's in the same function that they are being called the ref count is supposed to be incremented/decremented in that function)
        2. For all functions that do have the reference count modified after they are called, look at every single call of this function, follo and make sure that it is changed back by a local vput or another function call.
            - For this, I started with lookup, then dirnamev, then open. It's super easy to leak refcounts when calling lookup since it in inside and you have to make sure to manually decrement it in a lot of cases.
    3. For all of these cases, a big area where refcounts are often leaked are errors! Make sure to check all paths. Easiest way is to look at e and make sure it has a put before it if it's needed. When in doubt, just put one there anyways and see if it works!
2. Try to find ref count issues by following individual cases
    1. In Config.mk - after "DBG=" make sure to include "fref", "vnref", and "vfs"; this will print every time the refcount is changed for bo vnodes. For example, you can use "DBG=error,temp,print,test,fref,vnref,vfs" or "DBG=error,temp,test,fref,vnref,vfs".
    2. In terminal enter "make clean"
    3. In terminal enter "make"
    4. In terminal enter "script"
    5. "./weenix -n"
    6. Run vfstest (or whatever test that would cause the problem, but run as little as possible)
    7. Type "exit" in the kshell
    8. Write down all inode #s that still have refcounts.
    9. Click the X to close the crashed kernel
    10. Type "exit" in the terminal to end the logging
    11. Type "grep 'ino #' typescript" (replace # with one of your troublesome inodes, it's best to fix one problem at a time and re-run the enti procedure to fix the next problem)
    12. Watch how the refcount is changing for that particular inode number
    13. Examine those lines carefully and figure out where you forgot to decrement the ref count or where you incremented the ref count wh not suppose to.
    14. If needed, add a conditional breakpoint in your code with the troublesome inode number so you can step through line by line
    15. One thing you might notice is that when you see an inode number, you don't know which file it corresponds to. May be it's a good id call to `dbg(DBG_VFS,...)` in `do_open()` to print the `filename`, the address of the corresponding `vnode`, and the inode number. You n want to add a call to `dbg(DBG_VFS,...)` in `do_close()` to print the address of the `vnode`, the inode number, the refcount of the file o the refcount of the `vnode`.

Q: **Why does `vn_refcount == vn_nrespages` means that you can uncache all the pages in that vnode?**

A: Let's say that a vnode is in charge of 10 disk pages. Initially, all of them are on disk and `vn_refcount == vn_nrespages == 0`. Let's say that you reading the first few bytes of a file, so you need the first page. If we only bring in one page, `vn_nrespages` will be set to 1 to indicate that 1 out o are cached. But `vn_refcount` will be set to 2 because this page is used by the cache AND your thread. Eventually, when your thread is done with you will decrement the reference count to 1. Now this page CAN BE uncached because no thread is using it.

If the actual file system is more aggressive, when you ask for the first page, it brings in 4 pages. `vn_nrespages` will be set to 4 to indicate that 4 o pages are cached. But `vn_refcount` will be set to 5 because 4 pages is used by the cache AND one page is used by your thread. Eventually, when is done with the first page, you will decrement the reference count to 4. Now all 4 pages CAN BE uncached because no thread is using them.

gdb

Q: **How should I use the "`kernel info`" gdb commands? When I do "`help kernel`", it says that "`kernel info`" is undocumented.**

A: The "`kernel info`" gdb command works with functions that has the following function prototype (or something that's compatible with it):

```
vmmap_mapping_info(const void *vmmap, char *buf, size_t osize)
```

Usually, these function names ends with "`_info`". If I run:

```
grep '_info(' */*.c
```

I get (after I removed the stuff in "`util/debug.c`"):

```
main/gdt.c:size_t gdt_tss_info(const void *arg, char *buf, size_t osize)
mm/pagetable.c:pt_mapping_info(const void *pt, char *buf, size_t osize)
proc/proc.c:proc_info(const void *arg, char *buf, size_t osize)
proc/proc.c:proc_list_info(const void *arg, char *buf, size_t osize)
util/debug.c:size_t dbg_modes_info(const void *data, char *buf, size_t size)
vm/vmmap.c:vmmap_mapping_info(const void *vmmap, char *buf, size_t osize)
```

So, potentially, you can use any of these functions in a "`kernel info`" gdb command. You just need to pass the correct first argument. (The other arguments are used internally in the "`kernel info`" gdb command.)

Let's say that you are in `initproc_run` (or `kshell_execute_next()`) and you want to list all the processes in the system. You can do the followi

```
kernel info proc_list_info
```

The "`kernel proc`" gdb command gives a prettier output than the above command.

Let's say that you are in `handle_pagefault` and you want to print out the memory map of the current process. You can do the following:

```
kernel info vmmap_mapping_info curproc->p_vmmap
```

If you want to printn out the page table of the current process (don't do this in `bootstrap`, only do this after `initproc_run`), you can do:

```
kernel info pt_mapping_info curproc->p_pagedir
```

If you get an error message saying that the command is unknown, you should make sure that you have done this first.

Q: **How do I know if interrupt is enabled or not?**

A: Type the following in gdb:

```
info registers eflags
```

and look to see if you see "IF" (interrupt enable flag) in the printout.

One good place to do this is to set a breakpoint in `kmutex_lock()`. If your kernel get to `kmutex_lock()` because it's accessing an I/O device, inter be abled.

s5fs

Q: **Is there a difference between `ramfs` and `s5fs`?**

A: Two main differences that I can think of.

1) `ramfs` is a fake file system. It has very small file size limit and it can only handle a small number of inodes.
2) `ramfs` does not block when you try to read from it or write to it (because it doesn't have a disk to deal with). Your kernel 2 code that worked with `ramfs` may not work perfectly with `s5fs` because of this difference! With `s5fs`, if you don't increment reference count on `vnode` at the r you may end up using deallocated `vnode` that contains garbage. Since you don't have the source code of `s5fs`, you won't know exactly when count goes to zero. You may need to set `DBG=all` in `Config.mk` and look at kernel trace for `vnode` refcount changes to figure out what's going

---

Q: **How can I debug `s5fs` code?**

A: For kernel 3, `s5fs` code is given to you as "`kernel/libs5fs.a`" and you need to assume that all the code there is perfect and you have to work w

But what if you just want to debug the `s5fs` code that's in the original pristine weenix source because `s5fs_mount()` calls `pframe_get()` and `pfr` and you are trying to debug `pframe_get()`? If you just set a breakpoint in `s5fs_mount()`, you will get something that looks like the following:

```
Breakpoint ?, s5fs_mount (fs=0xc1ec8518) at fs/s5fs/s5fs.c:???
??? fs/s5fs/s5fs.c: No such file or directory.
```

where ??? are some numbers. The gdb error message is there because "`fs/s5fs/s5fs.c`" is in "`kernel/libs5fs.a`" and you don't have the sourc it. So, if you really want to debug `s5fs_mount()`, you need to **bypass** "`kernel/libs5fs.a`". To bypass "`kernel/libs5fs.a`", please edit "`kernel/Makefile`" and perform the following changes:

- Delete "`./libs5fs.a`" from `EFLAGS` in line 3.
- Append "`fs/s5fs`" from `SRCDIR` in line 14.

If you then recompile the kernel, you should be able to break inside `s5fs_mount()`. But please understand that you are running with the original p weenix source code for everything in "`fs/s5fs/*.c`". So, pretty much the **only** thing you can do at this point is to debug `s5fs_mount()` and hope will find your bugs in `pframe_get()` and `pframe_pin()`. If you want to use **any** of the real code in `s5fs`, you have to undo the two lines of chang above and recompile.

Please also understand that if you were able to get your `pframe_get()` and `pframe_pin()` to work with `s5fs_mount()`, it does **not** mean that you `pframe_get()` and `pframe_pin()` is perfect since you have **not** tested your `pframe_get()` and `pframe_pin()` against the rest of the S5FS.

Q: **I have just finished the 3 functions in `pframe.c` and now I'm getting `panic in fs/s5fs/s5fs.c:238 s5fs_read_vnode(): assertion fai` `S5_TYPE_FREE != ino->s5_type`. Is it my pframe code?**

A: If your kernel 2 works well, then it's probably your pframe code.

For that specific assertion, it Looks like it's complaining that an inode is not free. Since we must assume that there are no bugs in the s5fs code, ho that an inode that's expected to be free (for whatever reason) turns out not to be free?

Remember, the main difference between ramfs and s5fs is that there's a disk in s5fs and s5fs asks the disk to transfer data (via DMA) into a physic would mark the corresponding page frame object `busy`. If your `pframe_get()` returns a page frame when the page frame is `busy`, then you may en a page while the disk is transferring data into it. So, set a breakpoint when `pframe_get()` is about to return and make sure that it's not busy (and r comment block above `pframe_get()` about this).

Page Tables

Q: **Does `weenix` use standard page tables, linear page tables, forward-mapped page tables, or something else?**

A: `weenix` uses **forward-mapped (multilevel) page tables**. So, a virtual address has 3 parts. The most-significant 10 bits is an index into a **page dir** table, the middle 10 bits is an index into a **page table**, and the last 12 bits is the **offset into a physical page**. So, whenever you see "`pagedir`", "`p` in the `weenix` source code, you should relate that to the first 10 bits of a virtual address. The location of the page directory of the currently runnin stored in the `CR3` register of the CPU.

You should read the code of **`pt_init()`** in "`kernel/mm/pagetable.c`" to see how page directory and page tables are initialized. Please note that i some information about available physical memory. If you look for these lines in the terminal where you run `weenix`, you will notice that these ar right after you start running `weenix`. Here are the first few lines I see when I run "`./weenix -n`" with the pristine version of the assignment:

```
{bc-VirtualBox:bc}[1] ./weenix -n
/usr/bin/qemu-system-i386
open /dev/kvm: No such file or directory
Could not initialize KVM, will disable KVM support
qemu-system-i386: pci_add_option_rom: failed to find romfile "pxe-rtl8139.bin"
Kernel binary:
text: 0xc0000000-0xc003c000
data: 0xc003c000-0xc003c858
bss:  0xc003c858-0xc0047000
Physical Memory Map:
0x00000000-0x0009f400: Usable
0x0009f400-0x000a0000: Reserved
0x000f0000-0x00100000: Reserved
0x00100000-0x01ffd000: Usable
Highest usable physical memory: 0x01ffd000
Available memory: 0x01efd000
Page System adding range: 0xc0053400 to 0xc1efd000
```

The last 3 lines are printed in `pt_init()`. In the above output, it also tells you the virtual addresses of kernel's text, data, and bss segments! (Thes determined when you compile your kernel, so they may change as you add more code to your kernel.) The "start" of kernel is at virtual address `0:` and the "end" of kernel is at virtual address `0xc0047000`. This means that virtual addresses starting at `0xc0047000` is available for use for "dynam data structures. ("Dynamic" here doesn't mean `malloc/free()`.) So, if the kernel wants to create "dynamic" kernel data structures such as page di and page tables, it can be "allocated" starting here! (Again, "allocated" here doesn't mean `malloc()`.)

By the way, to dump a page table in a human-readable form, you can call **`pt_mapping_info()`** (in "`kernel/mm/pagetable.c`").

Q: **Is the page directory table really 8KB in size?**

A: Here's my understanding... In `weenix`, `pagedir_t` (which is also known as "`struct pagedir`") is defined as:

```
struct pagedir {
    pde_t      pd_physical[PT_ENTRY_COUNT];
    uintptr_t *pd_virtual[PT_ENTRY_COUNT];
};
```

So, the first 4KB is the **actual page directory table** (mentioned around slide 46 of the "virtual memory" lecture). The structure of this page direc (as well as the structure of a page table) is understood by the x86 CPU. The 2nd 4KB is just something the kernel uses to keep track of what goes page directory table (see the code in `pt_map()`).

The first argument to `pt_map()` is:

```
pagedir_t *pd
```

Given the above definition of `pagedir_t`, `pd` and `pd->pd_physical` are the same pointer! Therefore, when you store pd into the CR3 register, it lo you are asking the CR3 register to point to 8KB of data. But since `pd` and `pt->pd_physical` are the same pointer, you are actually asking the CR: point to the page directory table, which is 4KB in size!

In conclusion, page directory tables and page tables are both 4KB in size!

Q: **How can I know what to set `pdflags` and `ptflags` to in `pt_map()`?**

A: A **page table entry** is something the hardware (x86 processor) understands. You need to read section 5.2 (and may be 6.4) of the Intel 80386 Pro Reference Manual. Compare Figure 5.10 with the `PT_*` values in "`kernel/include/mm/pagetable.h`" and recall what we have talked about in le you should be able to figure out what you neeed to use. Here's one extra hint... When you call `pt_map()`, the values of `pdflags` and `ptflags` shou **same**.

Q: **How do I debug to know if I call `pt_map()` with the right arguments?**

A: Debugging is about knowning what to expect. So, let's start with the `pt_map()` function. The function prototype of `pt_map()` is:

```
pt_map(pagedir_t *pd, uintptr_t vaddr, uintptr_t paddr, uint32_t pdflags, uint32_t ptflags)
```

Its purpose is to setup a **page table entry** (PTE). Conceptually, you can relate this to slide 38 of the "virtual memory" lecture slides (the slide title (Two-level) Page Table). In this case, `pd` is kind of like the "page table", `vaddr` is the "vpn", `paddr` is "Physical Page #", `pdflags` and `ptflags` are the "V / M / R / Prot" bits (M and R are setup by the hardware, so they need to be initialized properly). I used the word "conceptually" because th shows a Basic (Two-level) Page Table scheme while x86 uses a Multilevel Page Table scheme. This is why the implementation of `pt_map()` is so because it needs to deal with the multilevel stuff. We should just **trust** that all the code in "`kernel/mm/pagetable.c`" works perfectly and all we : think about is the a Basic Page Table **abstraction** that's implemented by `pt_map()`. Trying to understand the code inside `pt_map()` can make one

So, if the PTE in question is setup correctly, next time you reference the same virtual address that just caused the page fault, the MMU inside the generate a correct physical address and you should not get a page fault.

Let's say that you are debugging the first instruction fetch in "`hello`". If you do (see below):

```
objdump --disassemble --section=".text" user/usr/bin/hello.exec
```

you should see the following lines:

```
080480f8 <__libc_static_entry>:
 80480f8:   83 c4 04                add    $0x4,%esp
 80480fb:   e8 94 ff ff ff          call   8048094 <main>
```

If `pf->pf_addr` points to the "page" that contains this data and `pf->pf_addr` is a page-aligned virtual address in the address space of the kernel, y be able to do the following in gdb right before you call `pt_map()` in `handle_pagefault()`:

```
print/x ((char*)(pf->pf_addr))[0xf8]
print/x ((char*)(pf->pf_addr))[0xf9]
print/x ((char*)(pf->pf_addr))[0xfa]
```

You should see:

```
0x83
0xc4
0x04
```

Immediately after `pt_map()` returns, you should try the following and see if you get the same 3 bytes:

```
print/x ((char*)vaddr)[0]
print/x ((char*)vaddr)[1]
print/x ((char*)vaddr)[2]
```

If you are not getting the same 3 bytes, it means that you probably didn't call `pt_map()` with the right parameters. If this is not working, you may look inside the page table to see if you have the right values in the related PTEs. This gets a little more involved, since `pt_map()` implements the that you have a two-level page table while in reality, x86 CPU uses a multi-level page table. The leading 10 bits of a virtual address is used as an to index the top-level page directory table to get a "page directory entry" (PDE) that points to a 2nd-level "page table". The next 10 bits of that vi is then used as an array index to index this 2nd-level page table to get the PTE that points to the physical page. In "`kernel/mm/pagetable.c`", the macro called `vaddr_to_pdindex()` that you can use to get the leading 10 bits of `vaddr`. You can als use the `vaddr_to_ptindex()` macro to get th bits of `vaddr`.

Here are some values you might want to check... Let's say that X is the integer that represents the leading 10 bits of `vaddr`. Since a PDE is the sar and is 4 bytes long, we can think of a page directory table is an array of 1024 unsigned integers. To print the PDE that corresponds to `vaddr`, you following gdb command:

```
p/x ((unsigned int *)curproc->p_pagedir)[vaddr_to_pdindex(vaddr)]
```

To interpret a PDE (or PTE), please see Figure 5.10 of the Intel 80386 Programmer's Reference Manual. The leading 20 bits of a PDE/PTE is a pl number. You can left-shift a physical page number by 12 bits to get a page-aligned address of the corresponding page table. Unfortunately, this is physical address and we don't know how to use a physical address! Fortunately, the `weenix` page directory data structure stores the kernel virtual this physical page. You can use the following gdb command to print this kernel virtual address:

```
p/x ((unsigned int *)curproc->p_pagedir)[vaddr_to_pdindex(vaddr)+1024]
```

If we treat a 2nd-level page table as an array of 1024 unsigned integers, we can finally get to the PTE by using the following gdb command:

```
p/x ((unsigned int *)(((unsigned int *)curproc->p_pagedir)[vaddr_to_pdindex(vaddr)+1024]))[vaddr_to_ptindex(vaddr)]
```

You should also use the following gdb commands to print the page table before and after you call `pt_map()`:

```
kernel info pt_mapping_info curproc->p_pagedir
```

If you don't see this in gdb, it means that your page frame is messed up. If you see this but you get a page fault when you retry the same virtual ac you are probably not passing the right arguments to `pt_map()`.

You should read the man pages of `objdump` to see what it does. Or, you can check out this tutorial.

Page Fault

Q: How can I debug a page fault that caused kernel panic?

A: Thanks to Zhiyi Xu, our course producer in Fall 2013, for posting the following to the class Google Group. The following procedure can be used kernel page fault, which should not happen in `weenix`. (It can also be used when debugging a user-space program, but you need to load the debug information manually first. You can use the same procedure in Eclipse as well.)

1. When you get a page fault that caused kernel panic, you may see the following:

   ```
   Breakpoint 1, dbg_panic_halt () at util/debug.c:190
   190     {
   ```

2. Look at how you get to kernel panic.

   ```
   (gdb) back
   #0  dbg_panic_halt () at util/debug.c:190
   #1  0xc00080e7 in dbg_panic (file=0xc0049cf1 "mm/pagetable.c", line=235,
       func=0xc0049fc6 "_pt_fault_handler",
       fmt=0xc0049e6c "\nPage faulted while accessing 0x%08x\n")
       at util/debug.c:208
   #2  0xc000d544 in _pt_fault_handler (regs=0xc1ebbf58) at mm/pagetable.c:235
   #3  0xc00039c8 in __intr_handler (regs=...) at main/interrupt.c:377
   ...
   ```

3. Move to stack frame #2, because we want to see the value in regs

   ```
   (gdb) frame 2
   #2  0xc000d544 in _pt_fault_handler (regs=0xc1ebbf58) at mm/pagetable.c:235
   235                     panic("\nPage faulted while accessing 0x%08x\n", vaddr);
   ```

4. Print out the `r_eip` value, this is the IP when the page fault happens.

   ```
   (gdb) p/x regs->r_eip
   $3 = 0xc0007650
   ```

5. Disassemble at this IP location. You can clearly see in which function, which instruction and which C statement cause the page fault.

```
(gdb) disas /m 0xc0007650
Dump of assembler code for function initproc_run:
339     {
   0xc0007640 <+0>:     push   %ebp
   0xc0007641 <+1>:     mov    %esp,%ebp
   0xc0007643 <+3>:     sub    $0x28,%esp

340             int *p = 0xb;
   0xc0007646 <+6>:     movl   $0xb,-0xc(%ebp)

341             int k = *p;
   0xc000764d <+13>:    mov    -0xc(%ebp),%eax
   0xc0007650 <+16>:    mov    (%eax),%eax
   0xc0007652 <+18>:    mov    %eax,-0x10(%ebp)
```

Q: **How do I know if a page fault was caused by accessing the stack, data, or text segment?**

A: To know for sure, you have to single-step in user-space code.

To debug user-space code, you need to load the debugging information of the user-space program manually. After `gdb` breaks in your user-space c can set addition breakpoints at whatever line numbers you need.

Continue to the line number you suspect that's causing the page fault. Now set a breakpoint at `handle_pagefault()` and change layout in `gdb` so assembly code. Then use the "`si`" command in `gdb` to single-step at the assembly code level. Please make sure you **read the assembly code** and u it so you know what to expect! If doing "`si`" caused a trap, you probably need to start from the beginning again and anticipate the trap the 2nd tin Read the assembly code carefully and guess what should happen and make sure what actually happens is exactly what you expected.

Q: **What are the meaning of the bits in "`kernel/include/vm/pagefault.h`"?**

A: Please see Figure 9-8 in section 9.8.14 of the Intel 80386 Programmer's Reference Manual. It shows the last 3 bits. Anything that's "reserved" me not suppose to be used. I have no idea what `FAULT_EXEC` is all about.

Please also see the wiki page on osdev.org for additional information that may be helpful.

Vmareas, Memory-mapped Objects, and Page Frames

Q: **What a good place to start to understand the relationship among `pframe`, `vma`, and `mmobj` data structures?**

A: In "`kernel/mm/pframe.c`", there's a function called `pframe_remove_from_pts()`. It's called by `pframe_clean()` and `pframe_free()`. In a comm `pframe_free()`, it says that this function is used to "Remove (a pframe) from all pagetables that map it". So, read this code carefully to see how t `pframe` from the `vmas` of all the processes that's using this `pframe`. This should give you an idea (not the whole picture) about the relationship am `vma`, and `mmobj` data structures so that when you need to create a `vma`, you know that it needs to work with this code. You should also consult som figures here.

The relationship between a `pframe` object and an `mmobj` is as follows. A `pframe` object is managed by a single `mmobj`. An `mmobj` can manage **mult** objects. If an `mmobj` **manages** k `pframe` objects, they are numbered 0, 1, 2, ..., k-1 and these `pframe` objects can be referred as `pagenum` = 0, `pager` `pagenum` = k-1 (see picture). Therefore, a `pframe` object can be uniquely identified by specifying which `mmobj` it belongs and which `pagenum` it is.

Q: **Are the `vmareas` sorted?**

A: Yes. We need to search for the right `vmarea` based on a virtual address or a virtual page/frame number in lots of different places. The `vmareas` sho sorted based on `vma_start` in increasing order to make searching faster.

Q: **Why would the actual file system call `pframe_get()`?**

A: Although we don't have the complete implementation of S5FS, if you "`grep`" for "`pframe_get`" inside `s5fs`, you can see a few places where it's i this should give you some idea of how the actual file system uses `pframe_get()`.

For S5FS, everything starts in the superblock. As you can see, in `s5fs_mount()`, it calls `pframe_get()` to read the superblock from disk to memo

```
    pframe_get(S5FS_TO_VMOBJ(s5), S5_SUPER_BLOCK, &vp);
```

The expectation here is that when `pframe_get()` returns, `vp` (the page frame object) will contain the data in the superblock. To get to the binary d superblock, you just access `vp->pf_addr` as done in:

```
    s5->s5f_super = (s5_super_t *)(vp->pf_addr);
```

The code there then checks the integrity of the superblock and pin the superblock page frame object down so that whatever `s5->s5f_super` point is `vp->pf_addr`) will not disappear accidentically.

Q: **What's the difference between `pframe_lookup()` and `pframe_get()`?**

A: `pframe_lookup()` is a higher level function than `pframe_get()`. Since you have the code of `pframe_lookup()`, you should take a look at it. All `pframe_lookup()` does is the following:

```
    return o->mmo_ops->lookuppage();
```

So, it's a polymorphic function and `o` can be a shadow object, an anonymous object, or an `mmobj` inside a `vnode`. You need to implement the `look` functions for the shadow object and the anonymous object, but you have the code for `vnode`. If you look at the code for `vnode`, you should see tha checks for some boundary conditions and calls `pframe_get()`.

There's another difference and this one is more subtle. These two functions has the same function prototype and their first argument is an `mmobj`. I picture above, we saw that an `mmobj` manages page frames for a particular `vmarea`. In reality, due to the handling of copy-on-write and fork, we h list of `mmobj` that manages all the page frames (including different "versions" of the same page frame) for a particular `vmarea` (please see Ch 7 of textbook). When you call `pframe_get(mmobj, ...)`, the returned page frame would be a page frame that's managed by the `mmobj` passed in as th argument to `pframe_get()`. When you call `pframe_lookup(mmobj, ...)`, the returned page frame would be a page frame that's managed by **one** `mmobjs` **starting at** the `mmobj` passed in as the first argument to `pframe_lookup()`.

Q: **In "`kernel/include/mm/mmobj.h`", the comment for the polymorphic `lookuppage` function says that this function may block. This functio suppose to lookup a page from the `mmobj`'s list of resident pages (`mmo_respages`). How can it block?**

A: If you do:

```
grep lookuppage kernel/*/*.c
```

You will find that the `lookuppage` is already implemented for `vnode` and block devices. In `vnode`'s version of `lookuppage`, it simply calls `pframe_` (which is what you need to implement). In the comment block above `pframe_get()`, it explains why it can block (i.e., if the page is not resident a allocate a new page and fill the new page). Filling a new page, in this case, means that you have to read data from disk (and that part of the code i written for you, you just need to find the right function to call).

Q: **I heard that polymorphism is more difficult in kernel 3 because recursion is involved. Where are we suppose to use recursion in kernel 3?**

A: If you read the comment blocks in "`kernel/vm/shadow.c`", you will see a few places mentioned recursion. It tells you that you are **not** suppose to recursion because you may blow up the kernel stack. In general, a recursive function can be "flattened" and be turned into a loop. It's certainly the So, you should definitely follow the recommendations in the comments if you don't want the stack to blow up.

There is something that kind of "look recursive". The `pframe_lookup()` function is implemented for you and it looks like this:

```
int
pframe_lookup(struct mmobj *o, uint32_t pagenum, int forwrite, pframe_t **result)
{
    return o->mmo_ops->lookuppage(o, pagenum, forwrite, result);
}
```

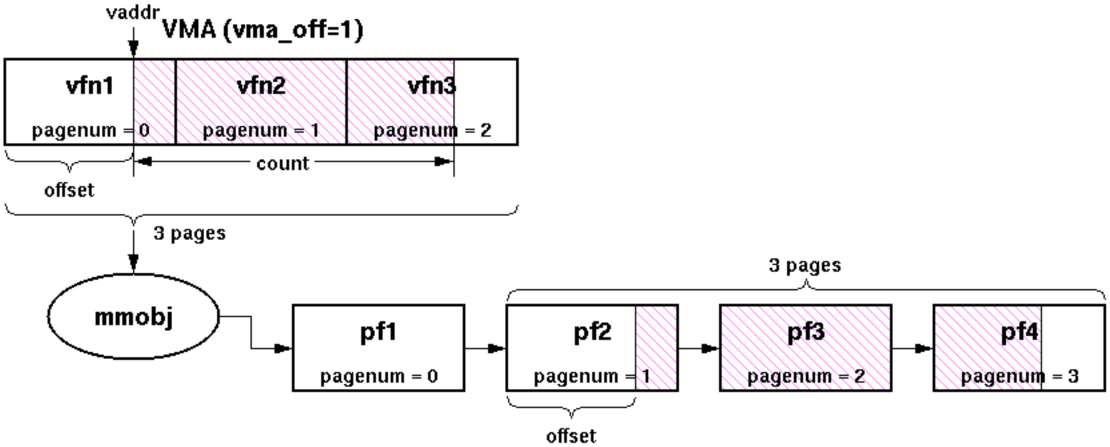
You can see that it simply invokes the `lookuppage()` function of the corresponding `mmobj`. If the `mmobj` turns out to be a **shadow object**, you may iterate through the list of shadow objects to find the page you are looking up. If you find the page you are looking for in one of the shadow object is no need for recursion. But if you cannot find the page you are looking for, then it would mean that you have reached the **bottom object**. The bo is either an `mmobj` inside a `vnode` or an **anonymous object**. In either case, you can just call `pframe_lookup()` on the bottom object. So, strictly sp is not recursion; although the code would look recursive.

Q: **Is `pagenum` (2nd argument in `lookuppage()`) a physical page number? (It also appears in `pframe_get()`, `pframe_lookup`, `pframe_get_resi`**

A: In the comment block above `lookuppage()` in "`kernel/include/mm/mmobj.h`", it uses the phrase "pagenum of the given object". So, it's not a ph number. It's also not a virtual page number. It's just a page number **index**.

A `vmarea` uses a `mmobj` to "read pages" (see the comment about `mmobj` in "`kernel/include/vm/vmmap.h`"). The first page read from this `mmobj` ha equal 0, the next page has `pagenum` equal 1, etc.

Thanks to Manan Patel who took CS 402 in Fall 2013 for the picture below.



The above picture also shows that a `vmarea` is **supported** by an `mmobj`, i.e., the `mmobj` "reads/loads/fills pages" for the corresponding `vmarea`. This picture (i.e., no shadow objects). So, the `mmobj` is either an anonymous object or an `mmobj` that lives inside a `vnode`. In the more complicated case privately mapped `vmarea`), the `mmobj` is replaced by a linked list of shadow objects that manages private pages that have been modified. At the en linked list is the so-called **bottom object** which is either an anonymous object or an `mmobj` that lives inside a `vnode`.

By the way, the "`offset`" in the picture is not related to "`vma_off`". It just illustrates that there can be an offset into the first page since `vaddr` may page-aligned and the offset into the first page in the `vmarea` is the same amount as the offset into the first page frame managaed by the `mmobj`. "`vm` the above example is `1` to indicate which page frames managed by the `mmobj` belongs to this `vmarea`.

Q: **I'm still confused about `pagenum`, can you give me an example?**

A: Let's say that **x** is an `mmobj` and it is managing 10 page frames.

Recall that a page frame is uniquely identified by an `mmobj` and a `pagenum`. We will use the notation (o,n) where o is an `mmobj` and n is a `pagenum`.

Therefore, we would refer to the 10 page frames managed by `mmobj` **x** as (x,0), (x,1), (x,2), (x,3), (x,4), (x,5), (x,6), (x,7), (x,8), and (x,9).

Suppose that **x** is the `mmobj` for `vmarea` **y**. **y** has a `vma_start` and a `vma_end`. Let's say that `vma_end` - `vma_start` = 3.

The question is, which 3 page frames are used by `vmarea` **y**? Is it (x,0), (x,1), (x,2)? Is it (x,1), (x,2), (x,3)? Is it (x,2), (x,3), (x,4)? And so on.

To indicate which page frames are used by `vmarea` **y**, we use `vma_off` to indicate which `pagenum` to start in the associated `mmobj`.

If they are page frames (x,4), (x,5), and (x,6), we would set `vma_off` = 4. If they are page frames (x,7), (x,8), and (x,9), we would set `vma_off` = 7

When you perform a lookup, the `pagenum` you must use is the `pagenum` for the corresponding `mmobj` (as shown in the picture above). So, if your v `vma_off` is wrong, you will end up asking the `mmobj` to look up the wrong page. Therefore, it's very important to get vmarea's `vma_off` to be corre

---

Q: **Can you explain more about `vma_start`, `vma_end`, `vma_off`, etc.?**

A: Remember, the words "**frame**" and "**page**" are interchangeable sometimes (I don't want to say "all the time" because I don't know if it's true or no "**vfn**" is "**vpn**" on slide 38 of the "virtual memory" lecture (where it talks about the basic/two-level page tables).

Let's say that I ask you to be in charge of 7 page frames and ask another person to be in charge of 5 page frames. What's a convenient way for me for the 3rd page frame? Instead of coming up with a complicated naming system for page frames, I will simply ask you, "Can I have YOUR 3rd p frame?" If I want the 2nd page frame from this other person, I will just ask, "Can I have YOUR 2nd page frame?" So, "YOU" here corresponds to "3rd" or "2nd" corresponds to `pagenum`.

Let's look at slide 107 of the "OS issues" lecture (where "`mmobj`" was mentioned) and apply that to the "`/usr/bin/hello`" program in `weenix`. Le the text segment/area. If you run:

```
objdump --headers --section=".text" user/usr/bin/hello.exec
```

you should get:

```
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00003d0a  08048094  08048094  00000094  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

So, the starting address of this segment is `0x08048094` and the size/length of this segment is `0x00003d0a` bytes. Therefore, this vmarea's address r `[0x08048094,0x08048094+0x00003d0a)` = `[0x08048094,0x0804bd9e)`. Please note that the notation `[a,b)` means that this interval is closed on includes `a`) but open on the right (i.e., does **not** include `b`). How many page frames do you need to hold `[0x08048094,0x0804bd9e)`? Well, you ne "virtual pages" `0x08048`, `0x08049`, `0x0804a`, and `0x0804b` and that's 4 pages. If I ask you to hold these pages, I will tell you that your first virtual p `0x08048` (i.e., `vma_start` = `0x08048`), you have 4 pages, and I will refer to these pages as page 0, 1, 2, and 3. As a result, `vma_end` = `vma_start+` `0x0804c`.

As it turns out, for the "`hello`" program, this is NOT exactly the first segment. I think weenix combines some segments together. If you set a brea the beginning of `vmmap_map()`, you will see that it's first called from `_elf32_map_segment()` in "`kernel/api/elf32.c`" when the "`hello`" progra load.

Finally, let's talk about `vma_off`. By default, `vma_off` should be zero. How does a `vma_off` become non-zero? If you look at the comment block n `vmmap_remove()` and look at case (3), if we start with `vma_off` being zero, it needs to change to a non-zero value. Similarly, for case (1), if we sta `vma_off` being zero, we need to create a new `vmarea` and its `vma_off` will not be zero.

Another possibility for `vma_off` to be non-zero is when you map only part of a file into your address space (may be using the `mmap()` system call argument not equal to zero). For example, if you have a file whose size is 40KB and you only map bytes 4096 through 4195 into your address spa a new memory segment, then `vma_off` would be 1 and `vma_end` would equal to `vma_start+1`. Another example would be when the loader builds space for a new program, it would need to map the text segment and the data segment from the executable file into two memory segments. One (o these memory segments will have `vma_off` not equal to zero. Please run "`objdump --headers user/usr/bin/hello.exec`" and look at the "File column in the printout.

Q: **I cannot find anything about backing store or swap space in the weenix documentation. Do I have to implement backing store?**

A: In lectures, it is mentioned that anonymous objects and shadow objects must be backed up in swap space (i.e., have a backing store). In `weenix`, i pages that are suppose to have a backing store, then they will never need to have a backing store! So, the short answer is that you do not have to i backing store. But you need to figure out which pages must be pinned so you don't need a backing store.

Q: **Do we need to use a separate vmarea for the "dynamic region" of a user process?**

A: The weenix documentation is a bit cryptic about this. In section 7.5.4, it says:

> ... the beginning of the heap sometimes starts halfway through the last page of another memory region.

So, it's hinting that there is "another memory region" for the heap.

In `test_start_brk()` in `"user/usr/bin/tests/memtest.c"`, there is a comment that says:

> /* Make sure region is gone */

So, it's hinting that a vmarea needs to be removed. Clearly, this cannot be the region that the dynamic region shares with the data+bss region. The must conclude that there is a separate vmarea that the dynamic region uses (in addition to the vmarea it shares with the data+bss region).

Finally, the comment above `do_brk()` in `"kernel/vm/brk.c"` says:

> The dynamic region should always be represented by at most ONE vmarea.

Putting all these together, we have the following...

1. Initially, the heap takes up the empty space in the same vmarea as the "data+bss" region. This is the "initial" region used by the heap. In thi is no dynamic region vmarea.
2. When you set the **brk** beyond the "initial" region, you need to create a "new" vmarea and extends the heap into it! Now you have one "dyn region vmarea.
3. When you set the **brk** back inside the initial region, you should remove the "new" vmarea!

Q: **The comment blocks of `anon_put()` and `shadow_put()` both said that we need to "unpin and uncache all of the object's pages and then fr object itself." What does "uncache" mean?**

A: Here, it means "free".

Q: **Why is it that when the reference count on an `mmo` (anonymous or shadow) reaches its number of resident pages, you can free the `mmo`?**

A: When an anonymous or a shadow object is created, its reference count (i.e., `mmo_refcount`) is set to 1 and its number of resident pages (i.e., `mmo_` is set to 0. When you create a page frame for it (i.e., make a page resident), the code in `pframe_alloc()` increments both `mmo_refcount` and `mmo_` When you create more page frames for it, the `mmo_refcount` should be one more than `mmo_nrespages`. If you have matching number of `ref()` an this `mmo`, the `mmo_refcount` should continue to be one more than `mmo_nrespages`. If you do one extra `put()`, you would first decrement `mmo_refc` this point, if its `mmo_refcount` equal its `mmo_nrespages`, then you know that there's no out-standing `ref()` call that needs to be offset by a `put()` can free the `mmo`.

Q: **In class, you mentioned that a page frame can be shared by multiple processes. If that's the case, how come there is no `refcount` field in a frame object?**

A: Because another level of indirection is used in weenix! Page frame objects are managed by `mmobjs` and an `mmobj` does have a `refcount` field. A p object can belong to just one mmobj (i.e., its `pf_obj` and `pf_pagenum` fields together uniquely identify the page frame, as mentioned in `"kernel/include/mm/pframe.h"`). It's the `mmobj` that can be shared by multiple processes (thus the page frames in an `mmobj` are shared by multip indirectly).

Q: **What does `VMAP_DIR_LOHI` mean?**

A: In lectures, the virtual memory map is drawn as a linked list of vmareas, layed out horizontally. These vmareas are sorted by virtual addresses anc be gaps in the address ranges between neighboring memory segments. When you need to create a new memory segment of size X, you need to fir that's big enough for X. So, basically, you need to run the First Fit algorithm. In Ch 3, our First Fit algorithm always go from small address to larg Here, you can do it 2 ways. If `VMAP_DIR_LOHI` is specified, you go from left to right. In this case, if there is a gap between the lowest possible user virtual address and the first memory segment, you need to allocate as low in the address range as possible. If `VMAP_DIR_HILO` is specified, you go to left. In this case, if there is a gap between the highest possible user-space virtual address and the last memory segment, you need to allocate as address range as possible.

User-space Programs

Q: **Is it true that I don't need to implement shadow object functions if I just want to run `hello`?**

A: If you are running "`/usr/bin/hello`" directly from `initproc_run()`, then yes, you don't need to implement shadow object functions.

Recall from the shadow objects lecture... The **only** reason we need shadow objects is to make `fork()` **and** copy-on-write work correctly **together** running "`/usr/bin/hello`" straight from `initproc_run()`, then you are **not** calling `fork()`. Therefore, you don't need any of the shadow object

But what about copy-on-write? Well, if you **never** call `fork()`, you know exactly where to copy from! You copy from the `mmobj` inside the corres `vmarea`. This `mmobj` is guaranteed to be a **bottom object** (because you haven't implemented shadow object).

But didn't we say that if a vmarea is PRIVATE, the `mmobj` must point to a shadow object? Well, it's your code. If you haven't implemented shadow don't point to a shadow object! Make a note in your code to fix it when you have implement shadow object because you want to `fork()`.

Q: **I followed the instruction in section C.3.2 of the kernel documentation to set a breakpoint in `main()` of "user/sbin/init.exec" but gdb neve there. What could be the problem?**

A: It's possible that the code for "`user/sbin/init.exec`" never got loaded into the address space. Let's say that `objdump` tells you that its code VMA start at `0x08048094` and you do the following in `gdb`:

```
add-symbol-file user/sbin/init.exec 0x08048094
b main
c
```

You should make sure that the code for "user/sbin/init.exec" is actually loaded into virtual address, starting at `0x08048094`, of this user process.

But this is probably not the first time you would need to debug a user-space program. You should get `/usr/bin/hello` to work first. In that case, ever page fault would be caused by the first instruction fetch of the `hello` program (because we are doing **demand paging**). The first instruction space program is `__libc_static_entry`. So, you would probably need to do the following:

```
add-symbol-file user/usr/bin/hello.exec 0x08048094
b __libc_static_entry
c
```

Q: How can I know what should be at a particular virtual address of a user process?

A: The easiest is probably to look at the code segment. Section C.3.2 of the kernel documentation says that you can use:

```
objdump --headers --section=".text" user/sbin/init.exec
```

to get some information about the "user/sbin/init.exec" user process. Reading the man pages of `objdump`, you would conclude that you should try:

```
objdump --disassemble --section=".text" user/sbin/init.exec
```

This output can be very long, so you should redirect the output into a file so you can look at the beginning part of the output (that's usually where `main()` is).

Q: How can I debug assembly code?

A: Thanks to Xiang Li, our grader in Spring 2014, for posting the following to the class Google Group.

- "**`display /i $pc`**" - Display the current executing assembly code, you can append a number before i to specify how many lines of assembl want to see. For example, "`display /3i $pc`". You can use a function name or an address instead of "`$pc`".
- "**`disas /r $pc`**" - Display the current executing code in both assembly and machine codes (in hex).
- "**`disas /m $pc`**" - Display the current executing code in both source and assembly.
- "**`si`**" - Step one line of assembly code.
- "**`layout asm`**" - Show all the codes in assembly layout.
- "**`layout split`**" - Show all the codes in C and assembly in separate panes.
- "**`info registers`**" - Display values of all CPU registers.
- "**`b *0x12345678`**" - Set a breakpoint at address `0x12345678`.

In fact, there are some addresses in assembly code that are special: such as the program's entry point, get it by executing the following command "Entry point address":

```
readelf -e /path/to/file
```

the top of the stack (typically `0x8047xxx` in weenix) and various system call functions' starting addresses. You can get the assembly code dump a system calls' starting addresses by executing:

```
objdump --disassemble -S --section=".text" /path/to/*.exec
```

Don't forget to append `-S` to make the code source commented. Take note of those addresses so when a page fault happens you know which funct executed. Then you can set a break point in that function to step through it (I am assuming you are executing your user land code by calling `kernel_execve()` in the `init` process, in this way gdb can still break in the kernel).

Lastly, the most powerful tool is perhaps to make an educated "guess"! Sometimes, as you guess where an error might happen and review your co carefully you can find some quite obvious mistakes.

Q: I cannot get "`/usr/bin/hello`" to work, how can I debug this?

A: You need to run weenix under gdb!

As I have mentioned in class, there are 3 things you need to do to get hello to work.

1) You need to build an address space.
2) You need to handle page faults.
3) You need implement the `write()` system call since hello writes to stdout with the Hello World message.

To know that you have done (1) correctly, set a breakpoint in `handle_pagefault()`. When you get there, type the following gdb command:

```
kernel info vmmap_mapping_info curproc->p_vmmap
```

and post what you see in the class Google Group and see if others are getting the same thing. If it looks right, you can proceed to step (2) and mal can handle every page fault in `handle_pagefault()` properly. You can do:

```
display/x vaddr
```

when you get your breakpoint in `handle_pagefault()`. You can type "`cont`" in gdb and see how many times you come into `handle_pagefault(` the printout of the "`display`" gdb command to the class Google Group and check if you are getting the same page fault addresses as everyone els checked out, you can then set a breakpoint in `sys_write()`. When hello calls `write()`, you need to verify that you pass the right string to the dev and you should see the Hello World message in the weenix console.

Q: How can I pass all those commandline arguments to "`/usr/bin/args`" using `kernel_execve()`?

A: The function prototype for `main()` is

```
int main(int argc, char *argv[]);
```

When you did warmup2, I don't know if you have tries to print out or look at the values in `argv`. For example, if you run your warmup2 this way:

```
./warmup2 -n 3 -mu 0.00001
```

you would get:

```
argv[0] = "./warmup2"
argv[1] = "-n"
argv[2] = "3"
argv[3] = "-mu"
argv[4] = "0.00001"
argv[5] = NULL
```

Therefore, to run:

```
/usr/bin/args ab cde fghi j
```

You should do:

```
char *const argvec[] = { "/usr/bin/args", "ab", "cde", "fghi", "j", NULL };
char *const envvec[] = { NULL };
kernel_execve("/usr/bin/args", argvec, envvec);
```

Q: I tried the procedure above and set a breakpoint in `main()` but I never hit that breakpoint. Why is that?

A: If your user space program cannot reach its `main()` (i.e., never got out of the **startup routine**), then you need to figure out where your code went setting up the **vmareas**, **mmobjs**, and **pframes**.

You need to figure out what your kernel must setup during the **first time** (and every time) a page fault happens (in `handle_pagefault()`). As I ha mentioned in class, this should be the result of fetching the first machine instruction in the startup routine (see the lecture that talked about "dema As you are ready to return from `handle_pagefault()`, you need to make sure that as the user program **retries** the fetch of the same machine instr will be successful. So, you need to make sure that the virtual address used in that memory reference maps correctly to a page frame where the firs is located and **verify** (using "`print/x`" command in `gdb`) that that machine instruction is right there where you expect it. You should use "`objdum` out exactly what machine code you should be seeing at that address and in that page frame.

Q: What is `atexit_handlers` and why does it contain garbage?

A: The code for the `exit()` system call looks like:

```
while (atexit_handlers--) {
    atexit_func[atexit_handlers]();
}
```

The value of `atexit_handlers` was initialized to zero (at the top of "`user/lib/libc/syscall.c`"):

```
static int atexit_handlers = 0;
```

Run your kernel under `gdb` and set a breakpoint at the user-space `exit()` function. When you get there, print the value of `atexit_handlers`. If it' then your program will die or hang inside the `exit()` system call.

Where is `atexit_handlers`? Since it's initialized to zero, it should live inside the data segment. Nevertheless, it doesn't live inside the data segme `weenix`! If you look at the symbol table of `hello` to get the address of `atexit_handlers`, then look for that address in the `objdump` of `hello`, you the address of `atexit_handlers` is actually in the `.bss` section of `hello`! As it turns out, the Linux linker is counting on the OS to initialize all pa `bss` segment to be all zeroes! In `weenix`, such as page frame is managed by an **anonymous objects**. So, if you see garbage in `atexit_handlers`, there is a bug in your code for handling anonymous objects.

Q: In `fork-and-wait`, how come my child process return from `waitpid()` and not `fork()`?

A: Okay, this is a bit tricky...

For `fork-and-wait`, the parent process returns first and blocks when it calls `waitpid()`. When the child process runs in the user space for the firs should return from `fork()`. But as we have learned from Ch 3, they way you return is simply get the return address from the current stack frame. child process is returning from `waitpid()` and not `fork()`, it's probably the case that copy-on-write was not implemented correctly.

When the parent process returns from `fork()`, it has a new shadow object for the stack segment but the PTEs for all the pages that correspond to segment has V=0. When it writes to the stack as it makes a function call, you need to make sure you perform copy-on-write. If you don't, you wo writing into the old shadow object that's shared between the parent and child processes. Later on, when the child process perform copy-on-write, from the stack with the wrong content.

Q: My kernel passed `fork-and-wait` but it's dying in `/sbin/init` with no user-space shell, what could be the problem?

A: If you are passing all the "basic" tests ("`/usr/bin/hello`", "`/usr/bin/args`", "`/bin/uname`", "`/bin/stat`", "`/bin/ls`", and "`/usr/bin/fork-and` perfectly, and you are not getting anything in the QEMU window (or simply see "init: starting shell on tty0" in it but no user-space shell prompt), you should look is `malloc()` since all the other user-space programs do not call `malloc()` so this would be your first encounter with `malloc()`.

What's special about `malloc()`, it trys to create the heap space for the user process (which should be located right after the data+bss segment). If y
into `malloc()`, you should see that it would call `malloc_init()` if `malloc()` has never been called. Inside `malloc_init()`, it has the following st

```
    page_dir = (struct pginfo **) MMAP(malloc_pagesize);
```

You need to check the return code from `MMAP()` (which is just a macro for calling `mmap()`). Use the following gdb command:

```
    p/x page_dir
```

You should see a **user-space virtual address**. If you don't, it's probably because you have a bug in your `do_mmap()`.

Another possible bug is that you did not create a new memory segment when `do_brk()` was called with a non-NULL argument for the first time.
the comment block above `do_brk()` to understand how the process break works and also read this FAQ item regarding the process break.

I have also seen a case where `malloc_active` not zero when `malloc()` is called for the first time. `malloc_active` is an uninitialized global variat
in the bss segment. For `weenix`, such a variable should be initialized to zero (and the page frame it's in should be managed by an anonymous obje
`malloc_active` has a non-zero value the first time you call `malloc()`, then the page frame managed by the corresponding anonymous object is nc
properly.

Q: I cannot get `/sbin/init` to work, can I run user-space programs under `kshell` to get partial credit?

A: First of all, I just want to stress that the procedure below should **ONLY** be used if you **CANNOT** run the user-space shell. If you use the procedu
the **best case**, you can only get **100%** for section (B) and **50%** for section (D) of the grading guidelines. In reality, you will get less than that beca
cannot halt the kernel when you run forkbomb and stress, you will lose more points.

I will assume that you can run the user-space programs in question directly from `initproc_run()` using `kernel_execve()`. If that's not do-able, y
be able to get partial credit for the corresponding test in sections (B) and (D) of the grading guidelines. If it's do-able, the procedure below will sh
how to transfer your code into a kshell command.

Recall that in kernel 1 and 2, when you run `faber_thread_test` or `vfstest`, you used a `kshell` command to create a child process and run a ker
the child process (and your `kshell` command waits for the child process to die so that the `kshell` command runs in the "foreground"). Then in th
procedure of this kernel thread, you call either `faber_thread_test()` or `vfstest_main()`.

You need to do exactly the same thing for **each** of the user-space programs in sections (B) and (D) of the grading guidelines. For example, for the
section (D) of the grading guidelines, you can create 5 different `kshell` commands. I would call these kshell commands "`user-vfstest`", "`user-`
"`user-eatmem`", "`user-forkbomb`", and "`user-stress`", to make it super clear to the grader that these commands are used to run user-space progr
can do similar things for section (B) of the grading guidelines. Please make sure you pass the right `argv` values to these user space programs so th
run identically to the way they were suppose to run as indicated in the grading guidelines.

In each of these "`user-*`" `kshell` commands, you create a child process and run a kernel thread in the child process (and your kshell command w
child process to child so that the `kshell` command runs in the foreground). Then in the first procedure of this kernel thread, you call `kernel_exe`
the same way your first ran "`/usr/bin/hello`") to run the corresponding user space program.

Please understand that for `/usr/bin/forkbomb` and `/usr/bin/stress`, if you cannot figure out how to shutdown `weenix` after you have started th
don't know if there is a way), you would wind up getting less than 50% of the points allocated for these tests. For the other tests, you have to satis
requiremented mentioned in the grading guidelines (such as being able to run the corresponding `kshell` commands twice and shutdown `weenix` p
when you exit `kshell`) in order to get 50% of the points allocated for them.

Q: My program runs perfectly but I got a weird page fault in the end, how should I debug this?

A: Well, every case is different. So, you have **analyze your user space program** and determine what to do in every step. If the bug is completely re
then all you have to do is to perform a **binary search**. A binary search is what you do when you look for a name in the phone book (where names
You open up the phone book right in the middle and you look at the first name you see. Let's call it X. If the name you are looking for is before X
must be between the beginning of the phone book and your current page. If the name you are looking for is after X, then it must be between the c
and the end of the phone book. This way, you eliminate half the phone book in one step. Then you do the same (recursively) with the remaining h
phone book. You are guaranteed to find the name you are looking for in O(log(N)) number of steps where N is the number of pages in the phone l
even if N is a billion, you can find the name in 30 steps (and this requires no "luck")!

Let's apply this to the weird crash. If everything is working perfectly, how can it get a weird page fault? You need to locate the exact instruction t
the weird page fault. So, you do your binary search. You ask yourself, "Has your program reached the end of `main()`?" If the answer is yes, then t
page fault must happen after the end of `main()`. If the answer is no. Then the weird page fault must have happend before the end of `main()`.

If your program reached the end of `main()`, you know that your startup routine would call `exit()`, which eventually wil reach your implementati
`do_exit()`. So, set a breakpoint at `do_exit()` and see if you get there. If you can't get to `do_exit()`, did you program get to the `exit()` system c

Please keep in mind that if your program can reach a certain statement, you can also do a few single steps and see if that would cause the weird p

The bottomline is that you need to be very very patient! You may have to restart your kernel many many times. But you need to understand that i
perfectly repeatable, looking for it this way will **always** find the place where your program crashed! Then you have to figure out **why** it crashed.

Q: How does binary search debugging (mentioned right above) work, can you give an example?

A: I'm going to demonstrate the first step to find memory corruption bug using this as an example. The assumption here is that this bug is repeatable
place and same time every time. Please understand this is **just an example**. You can apply the technique here to find bugs. There is no guarantee
technique will always work. If it works, great! Consider yourself lucky!

I got the following e-mail from a student:

> We found that in `vfs_shutdown`, it's trying to do `pframe_remove_from_pts` and one pframe got problem. We trace it into `pt_unmap` and `pt[index]=0` cause the fault. The fault message is 'page faulted while accessing `0xdddddf0d`. The pframe is `0xc1e7e484`.

`vfs_shutdown()` calls `s5fs_umount()`, `s5fs_umount()` calls `vnode_flush_all()`, `vnode_flush_all()` calls `pframe_clean()`, `pframe_clean()` `pframe_remove_from_pts(pf)`, and `pframe_remove_from_pts(pf)` calls `pt_unmap(pd,vaddr)`. Let's assume that memory was good at the begi `vfs_shutdown()` and gone bad somewhere between that and when your kernel crashed. (If this is not the case, when you need to find a point in ti the memory was good. For this example, let's assume that memory was good at the beginning of `vfs_shutdown()`.)

The code for `pt_unmap(pd,vaddr)` looks like:

```
void
pt_unmap(pagedir_t *pd, uintptr_t vaddr)
{
    KASSERT(PAGE_ALIGNED(vaddr));
    KASSERT(USER_MEM_LOW <= vaddr && USER_MEM_HIGH > vaddr);

    int index = vaddr_to_pdindex(vaddr);

    if (PT_PRESENT & pd->pd_physical[index]) {
        pte_t *pt = (pte_t *)pd->pd_virtual[index];

        index = vaddr_to_ptindex(vaddr);
        pt[index] = 0;
    }
}
```

and the kernel is crashing in the last `pt[index]=0`. You analyze the code above and you decide that either `index` is bad or `pt` is bad. Let's assume bad. Restart your kernel, set a breakpoint at `vfs_shutdown()`. When you get there, add a breakpoint at `pframe_remove_from_pts()`. When you g add a breakpoint at the line where it says `pt[index] = 0` in the above code. When you get there, print `pt` and continue. When you get to the last print `pt` and continue. Keep doing this until you crash. The last `pt` you printed should contain the bad address. But where did `pt` came from? It's f

```
pte_t *pt = (pte_t *)pd->pd_virtual[index];
```

So, the memory location that contains the bad value is `&pd->pd_virtual[index]`. (Be careful! In this example, `index` gets two values. We want t and not the last one in this case. So, you may have to rerun everything and set the breakpoint at the `pt = ...` line.

Let's say that you get to the `pt = ...` line when you will be getting the bat `pt`. Now you do:

```
p &pd->pd_virtual[index]
```

and you get:

```
$1 = (uintptr_t **) 0xc1ef3880
```

So, you declare that memory location `0xc1ef3880` is corrupted.

You restart your kernel. When you get at `vfs_shutdown()`, you do:

```
display/x *(unsigned int *)0xc1ef3880
```

(I typecast the address to `(unsigned int *)` because I want to print out 4 bytes of data starting at that memory location.) The assumption here is time, memory was good. So it should print a good value. Since you know that `vfs_shutdown()` calls `s5fs_umount()`, `s5fs_umount()` calls `vnode_flush_all()`, `vnode_flush_all()` calls `pframe_clean()`, `pframe_clean()` calls `pframe_remove_from_pts(pf)`, and `pframe_remove_from_pts(pf)` calls `pt_unmap(pd,vaddr)`. You can set a breakpoint at the beginning and ending of all these functions. When yo the content of the questionable memory location will get printed and you can see if it's still good.

What if you noticed that at the beginning of `vnode_flush_all()`, the memory location is good. But when `vnode_flush_all()` returns, it went ba `s5fs_umount()` is in `kernel/libs5fs.a`, you cannot do single step when you debug. But since you have the source code for this function, you sh look. You would then notice that `s5fs_umount()` calls `vref()`, `pframe_get()`, and `pframe_unpin()`. So, you can set breakpoints in these functio check the memory content in question.

If you keep doing searches like this, the hope is that eventually, you will find the place where your code corrupts a memory location.

Please understand that this is just an example! It may be the case that your code would never each `pt_unmap()` this way. And may be reaching `pt` this way **is the bug**!

System Calls

Q: How should I implement the functions in "`kernel/api/syscall.c`"?

A: Good thing that **`sys_stat()`** is already implemented! So, look at the **`sys_stat()`** code and do similar stuff!

Let's take a look at **`sys_stat()`**. Here's the basic code:

```
static int sys_stat(stat_args_t *arg)
{
    stat_args_t kern_args;
    struct stat buf;
    char *path;
    int ret;

    copy_from_user(&kern_args, arg, sizeof(kern_args));
    path = user_strdup(&kern_args.path);
    ret = do_stat(path, &buf);
    ret = copy_to_user(kern_args.buf, &buf, sizeof(struct stat));
    kfree(path);
    return 0;
}
```

When a user-space program calls the **`stat()`** system call, it eventually gets here. This function gets the function arguments from the user-space p calls the **`do_stat()`** function you implemented in kernel 2, get the result from **`do_stat()`**, and return the result to the user-space program.

What are the function arguments to the user-space program? If you do "man 2 stat" on Ubuntu, in the SYNOPSIS section, it says that **`stat()`** syst the following function prototype:

```
    int stat(const char *path, struct stat *buf);
```

So, the arguments are a pointer to a string and a pointer to a user-space buffer of type `(struct stat *)`. What's a "pointer"? A user-space **pointe** contains a **user-space virtual address**. In the **`stat()`** system call, the arguments are packaged into a data structure (of type `stat_args_t` here) ar eventually get passed to **`sys_stat()`** (see **`stat()`** in "`user/lib/libc/syscall.c`" to see how the "packaging" is done). This "packaging" is syste specific. So, for every system call, you need to look at the code in "`user/lib/libc/syscall.c`" and match it with your implementation in "`kernel/api/syscall.c`".

One important thing to understand here is that the **`arg`** argument in **`sys_stat(arg)`**, is a **user-space virtual address** and **`&kern_args`** is a **kernel-** **virtual address**. So, the **`copy_from_user(kaddr, uaddr, nbytes)`** function (in "`kernel/api/access.c`") copies **`nbytes`** from **`uaddr`** (which is virtual address) to **`kaddr`** (which is a kernel-space virtual address). If you take a look at **`copy_from_user(kaddr, uaddr, nbytes)`**, you would s is accomplished using the **`vmmap_read(curproc->p_vmmap, uaddr, kaddr, nbytes)`** function which **reads** **`nbytes`** from the address space (i.e. `>p_vmmap`), starting at user-space virtual address **`addr`** into **`kaddr`** which specified a **`nbytes`** buffer starting at kernel virtual address **`kaddr`**. This sh you an idea what code you need to write inside **`vmmap_read()`**.

Let's continue with the **`sys_stat()`** code. The **`user_strdup(ustr)`** (in "`kernel/api/access.c`") is like the C library function **`strdup()`** exceptio **`ustr.as_str`** contains a **user-space virtual address**. As you can see in the implementation of **`user_strdup(ustr)`**, a kernel buffer is allocated u **`kmalloc()`** and **`copy_from_user(kaddr, uaddr, nbytes)`** is used to copy from a buffer specified by a user-space virtual address to a buffer spe kernel-space virtual address.

The code of **`sys_stat()`** then call your kernel 2 **`do_stat()`** passing **kernel-space virtual addresses**. When **`do_stat()`** returns, **`buf`** contains the **`do_stat()`**. Then it calls **`copy_to_user(uaddr, kaddr, nbytes)`** to copy from a buffer specified by a kernel-space virtual address to a buffer sp user-space virtual address.

Finally, **`kfree()`** is called to free up the buffer allocated by **`kmalloc()`**.

Q: **Can you tell me more about "`vmmap_read()`"?**

A: Assuming you have read the above FAQ item, here's a little more about `vmmap_read()`.

The function prototype of `vmmap_read()` is:

```
    vmmap_read(map, uaddr, kaddr, nbytes)
```

It copies `nbytes` from user-space virtual address `uaddr` into a `nbytes` long buffer with kernel-space virtual address `kaddr`. If all these are in user- it's pretty simple and you can do:

```
    memcpy(kaddr, uaddr, nbytes)
```

But we are doing this in a kernel and the user-space address space is implemented with a memory map. We cannot just call `memcpy()` directly bec `nbytes` with starting address of `uaddr` may **span multiple vmareas**! So, what you need to do is to perform the copy carefully. When you do `memc` you **must not copy across page boundaries**! Let's take a look at an example.

Let's say that you want to copy 3KB (i.e., `nbytes = 0xc00`) from user-space virtual address `0x12345678` to a kernel buffer at kernel-space virtual `kaddr`. `0x12345678` plus `nbytes = 0xc00` is `0x12346278`. So, the range you need to **copy from** is `[0x12345678,0x12346278)`. This range spans frame number `0x12345` and `0x012346`. What do you need to copy from `vfn 0x12345`? You need to copy from offset `0x678` all the way to the end into the beginning of the buffer at `kaddr`. How many bytes do you need to copy? Well, the range is `[0x678,0x1000)`, which is `0x988` bytes. So, y find the page frame that corresponds to `vfn 0x12345` and do something like:

```
    memcpy(kaddr, pf->pf_addr+0x678, 0x988)
```

Then you move to the next page. What do you need to copy from `vfn 0x12346`? You need to copy from offset 0 and how many bytes do you need You have copied `0x988` bytes so far and you have `nbytes - 0x988 = 0x278` bytes left to copy. Where should you copy them to? The first `0x988` destination buffer has data written into it already, so you have to start at buffer offset `0x988`. You need to find the page frame that corresponds to `0x12346` and do something like:

```
    memcpy(&kaddr[0x988], pf2->pf_addr, 0x278)
```

Q: **What about "`vmmap_write()`"?**

A: It is important to understand `vmmap_write()` since it's one of the first functions you will have to write once you set `VM=1` in `Config.mk`. The loade "`kernel/api/elf32.c`") calls `vmmap_write()` to copy data read from the disk into certain memory segments of the user address space.

Assuming you have read the above FAQ item about `vmmap_read()`, here's a little more about `vmmap_write()`. Please notice the similarity betwee `vmmap_write()` and `vmmap_read()`.

The function prototype of `vmmap_write()` is:

```
    vmmap_write(map, vaddr, buf, count)
```

It copies `count` bytes from `buf` (which contains data from somewhere, such as the disk) into a memory buffer located at user-space virtual addres all these are in user-space, then it's pretty simple and you can do:

```
    memcpy(vaddr, buf, count)
```

But we are doing this in a kernel and the user-space address space is implemented with a memory map. We cannot just call `memcpy()` directly bec `count` with starting address of `buf` may **span multiple vmareas**! So, what you need to do is to perform the copy carefully. When you do `memcpy(` **must not copy across page boundaries**! Let's take a look at an example.

Let's say that you want to copy 3KB (i.e., `count = 0xc00`) from kernel-space virtual address `0xfedcba98` to a user-space virtual address `0x12345` `buf = 0xfedcba98` and `vaddr = 0x12345678`). `0x12345678` plus `count = 0xc00` is `0x12346278`. So, the range you need to **copy into** is `[0x12345678,0x12346278)`. This range spans virtual frame number `0x12345` and `0x012346`. What data do you need to copy into `vfn 0x12345`? Y copy the first `4KB - 0x678 = 0x988` bytes from the start of `buf` into the tail of the page that corresponds to `vfn 0x12345`. So, you need to find th frame that corresponds to `vfn 0x12345` and do something like:

```
memcpy(pf->pf_addr+0x678, buf, 0x988)
```

Notice that `pf->pf_addr` is the kernel virtual address for the corresponding user-space page that starts at user-space virtual address `0x12345000`. `memcpy()` works because it copies from one kernel virtual address to another kernel virtual address.

Then you move to the next page. What do you need to copy into `vfn 0x12346`? How many bytes do you need to copy? You have copied `0x988` b and you have `count - 0x988 = 0x278` bytes left to copy. Where should you copy from? The first `0x988` bytes of `buf` has already been copied, sc to start at buffer offset `0x988`. You need to find the page frame that corresponds to `vfn 0x12346` and do something like:

```
memcpy(pf2->pf_addr, &buf[0x988], 0x278)
```

`fork()`

Q: What should go into `fork()`?

A: The following is copied from the wiki page at Brown University about `fork()`.
(Disclaimer: Since I just copy the text from Brown University, I do not know if it's all accurate.)

> A good implementation of the previous sections is essential; `fork()` is complicated enough without having to debug the rest of your VM cc the same time. To avoid 1000+ lines of code for a single function, we are providing the documentation for `fork()` here instead of in the sou file comments.
>
> Functions you will need to write:
>
> ```
> int do_fork(struct regs *regs);
> vmmap_t *vmmap_clone(vmmap_t *map);
> kthread_t *kthread_clone(kthread_t *thr);
> ```
>
> `Fork()` is a moderately complicated system call. We present it here as one long algorithm, but it will make your life much easier if you brea down into separate subroutines. Close attention to detail will help you; an under-debugged `fork()` can cause subtle instabilities and bugs la on.
>
> Bugs in the virtual memory portion of `fork()` tend to cause bizarre behavior: user process memory may not be what it ought to be, so almc anything can happen. The user process may end up executing what should be data, jumping into the middle of a random subroutine, etc. Th sorts of bugs are very, very difficult to track down. For this reason you should code more defensively than you may be used to. Assert every you can, panic at the first sign of trouble, and include apparently unnecessary sanity checks.
>
> Above all, be sure you really understand the algorithm before you start coding. If you try to implement it before you understand what you a trying to do, you will write buggy code. You will then forget that you have written buggy code, and waste time debugging code that you sh have thrown away.
>
> Here are the steps you have to take. Note that these steps are not in the correct order, consider carefully the order in which you do these step particularly keep in mind what kind of cleanup you will need to do if one of these steps fails. Look out for steps which cannot be undone.
>
> - Allocate a `proc_t` out of the procs structure using `proc_create()`.
> - Copy the vmmap_t from the parent process into the child using `vmmap_clone()`. Remember to increase the reference counts on the underlying `mmobj_ts`.
> - For each private mapping, point the `vmarea_t` at the new shadow object, which in turn should point to the original `mmobj_t` for the `vmarea_t`. This is how you know that the pages corresponding to this mapping are copy-on-write. Be careful with reference counts. note that for shared mappings, there is no need to copy the `mmobj_t`.
> - Unmap the user land page table entries and flush the TLB (using `pt_unmap_range()` and `tlb_flush_all()`). This is necessary becau the parent process might still have some entries marked as "writable", but since we are implementing copy-on-write we would like ac to these pages to cause a trap.
> - Set up the new process thread context (`kt_ctx`). You will need to set the following:
>   - c_pdptr - the page table pointer
>   - c_eip - function pointer for the userland_entry function
>   - c_esp - the value returned by `fork_setup_stack()`
>   - c_kstack - the top of the new thread's kernel stack
>   - c_kstacksz - size of the new thread's kernel stack
>   - Remember to set the return value in the child process!
> - Copy the file descriptor table of the parent into the child. Remember to use `fref()` here.
> - Set the child's working directory to point to the parent's working directory (once again, remember reference counts).
> - Use `kthread_clone()` to copy the thread from the parent process into the child process.
> - Set any other fields in the new process which need to be set.
> - Make the new thread runnable.
>
> You will have to revisit your implementation of the `exit()` system call which you wrote in kernel 1. Be sure that your implementation is releasing all resources it should; your OS should be able to run the following program fragment forever:
>
> ```
> for (;;) {
>     if (fork() == 0)
>         exit(0);
>     else
>         (void)wait(0);
> }
> ```

Q: How do I debug `fork()` if my child process is not returning to the same place as the parent?

A: Remember that both the parent process and the child process must return to EXACTLY the same place. This means that at the instance they execu "`iret`" (return from interrupt) machine instruction, the CPU registers better have exactly the same values. Well, not all of them are identical since difference processes. You can use the "`info registers`" gdb command to print all the CPU registers values right before you execute "`iret`" and the registers that contain **user-space addresses** and they better be identical! Also, make sure that for the parent process, EAX must contain the pi child process and for the child process, EAS must contain 0.

Finally, how do you get to the "`iret`" machine instruction? For the child process, it's pretty easy. Just set a breakpoint at `userland_entry()`. Wh their, switch to assembly mode by typing "`layout asm`". Then keep typing "`si`" until you get to the "`iret`" machine instruction. For the parent pr breakpoint at the end of `do_fork()`. When you get there, you can still use "`n`" to keep executing code. But eventually, you will get into the interru (something that starts with "`__intr_handler`") and you will have to switch to assembly mode at that time. So, type "`layout asm`" and use "`si`" t step.

forkbomb

Q: **My `forkbomb` crashed after over 2000 forks, is it acceptable?**

A: Are you running with `DBG=error,temp,print,test` in `Config.mk`? If you are, I would be very surprised to see that you can have a very high for after running forkbomb for one minute. Please remember that the grader **must** follow the grading guidelines and **must** halt your kernel after runn `forkbomb` for one minute. So, stick to the grading guidelines if your concern is about grading.

Q: **Is it possible to make `forkbomb` run forever?**

A: When you run `/usr/bin/forkbomb`, even if your kernel 3 is perfect, you will eventually run out of memory because you are not freeing shadow c you read the comments in "`kernel/vm/shadowd.c`", you would see that there's something called the `shadowd` that can be used to "**migrate**" shad and free up shadow objects. By default, `shadowd` does not run in `weenix`. If you set **`SHADOWD=1`** in `Config.mk`, `shadowd` will be started in `idlepro`

But if that's all you do, the `shadowd` will just be sleeping. So, you still need to do one more thing to wake up `shadowd` at the "right time". I cannot exactly when and how to wake up the `shadowd`, but you should start a discussion with your classmates in the class Google Group if you get this f

IMPORTANT NOTE: Please do **not** attempt the above unless you are **comletely** done with kernel 3 and have nothing else to do! If you still hav kernel 3, running the `shadowd` can make your kernel very very unstable! If you run the `shadowd` and notice that your kernel become very unstable set **`SHADOWD`** back to **`0`** in `Config.mk`.

Page Frame to Address Space

Q: **How can I find out which process is using a particular page frame?**

A: To gain a better understanding of the relationship between a **page frame** and the address spaces of all the processes, you can read the code of the **daemon** to see how the pageout daemon **frees a page frame** and what needs to be done to remove references to that page frame from the **page ta** the processes.

You can start with **`pageoutd_run()`** in "`kernel/mm/pframe.c`". The code looks quite straightforward! The main thing in there is that it calls `pfra` to free a page frame. In the code of `pframe_free()`, you can see that it calls `pframe_remove_from_pts()` to remove the page frame from all pag all processes. Read the code of `pframe_remove_from_pts()` to further understand how things are related to each other.

Address Space to Page Frame

Q: **How can I find out what page frames a process owns?**

A: This is like going in the opposite direction of the above FAQ item. You can get a better understanding of the relationship between address spaces processes and memory-mapped objects, you can read the code of the **shadow daemon**, i.e., **`shadowd()`** in "`kernel/vm/shadowd.c`". The purpose shadow daemon is to reduce the size of the "inverted tree of shadow objects" in section 7.3.3 of the textbook. Well, reducing the tree is not all tha interesting. But the code there shows how to walk through the address spaces (i.e., memory maps) of all processes to get to all memory-mapped c from a memory-mapped object, determine if it has shadow objects and if so, how to traverse it. In the inner-most loop, it shows you how to move page frames from one shadow object to another.

By the way, page frames that are managed by an anonymous object or by a vnode object may be **shared** by multiple processes. (Page frames that managed by a shadow object is not shared because the corresponding memory segment is a **private** segment. Another way to look at it is that onl frames in the **bobtom object** may be shared by multiple processes.) So, a page frame is really not "owned" by a process. Therefore, you need to b careful when you manipulate page frames (and memory-mapped objects).

`tty1` and `tty2`

Q: **Where are `tty1` and `tty2`?**

A: If you can access `tty0`, you should be able to get to `tty1` and `tty2` by pressing the `<F1>` and `<F2>` keys in the **QEMU** window.

You **don't** have to get anything to work in `tty1` and `tty2`. The grader will **not** access them during grading.

Relationship among Address Space Data Structures

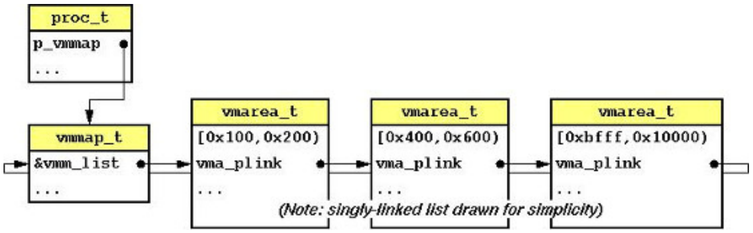
Q: **The comment block right above `vmmap_remove` says that for Case 1, you need to "increment the reference count to the file associated with vmarea". A vmarea does not necessarily associate with a "file". Is that comment block wrong?**

A: It's wrong. It should say "**mmobj**" and not "**file**".

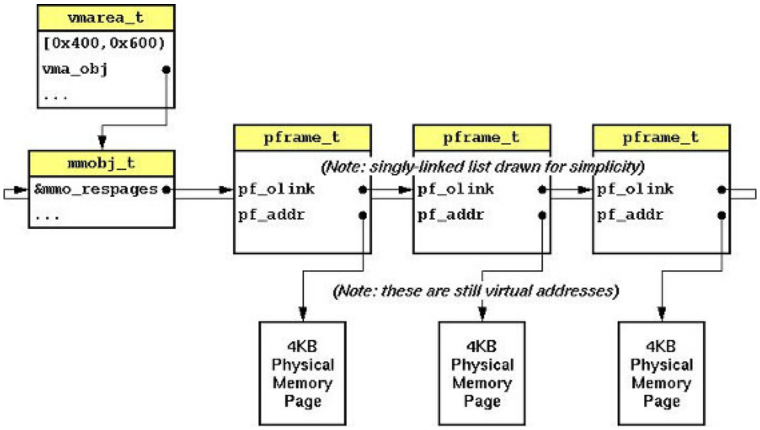
Q: **What is the relationship among some of the address space related data structures?**

A: This may take a while to explain. I'll just show a bunch of figures for now...

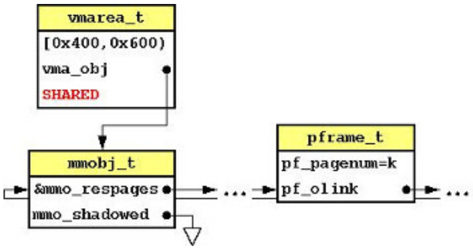
Process, Address Space (i.e., Memory Map)



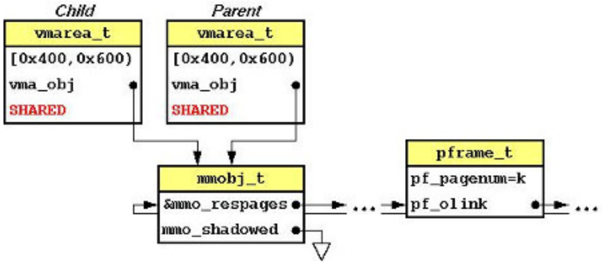
Memory Map, Memory-mapped Objects, Page Frames



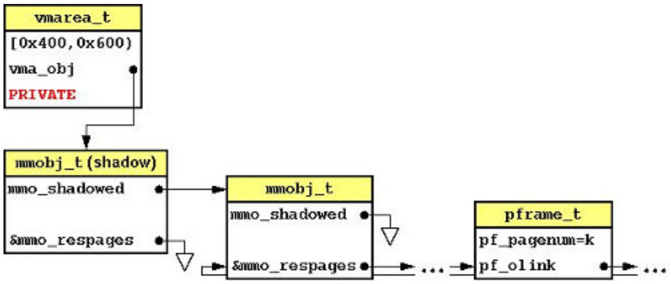
SHARED VM Area (before fork)



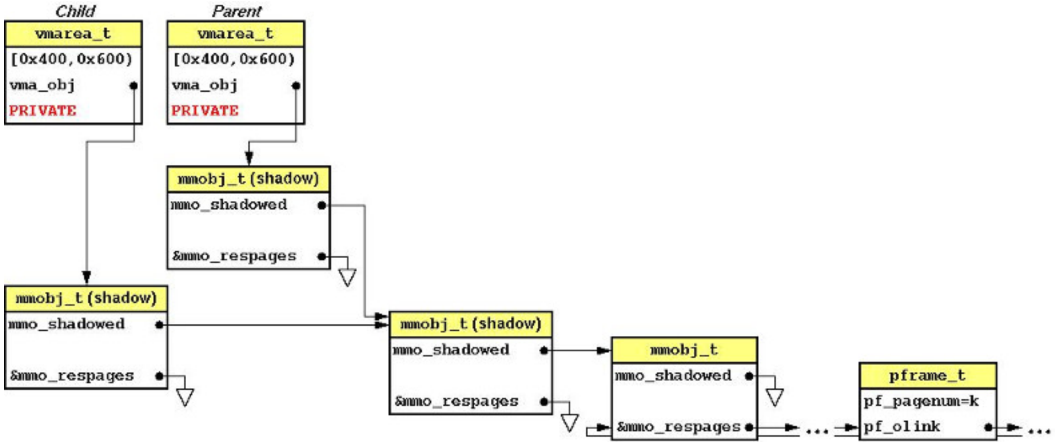
SHARED VM Area (after fork)



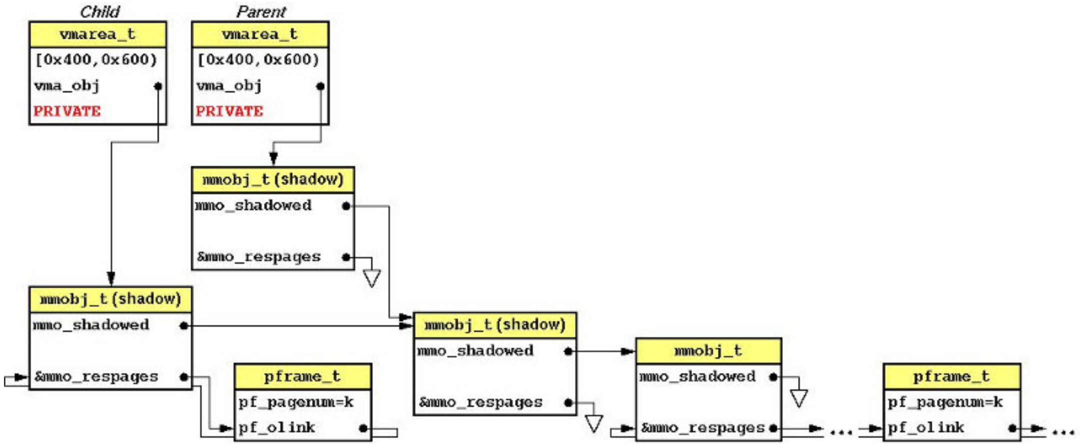
PRIVATE VM Area (before fork)



PRIVATE VM Area (after fork)



PRIVATE VM Area (after fork, after child process write to page k)



By the way, to dump a `vmmap_t` in a human-readable form, you can call **`vmmap_mapping_info()`** (in "`kernel/vm/vmmap.c`").

[Last updated Mon Nov 19 2018]

[Home | Description | Lectures | Videos | Discussions | Projects | Participation | Newsgroup]