Intern Workshop: Linux Kernel

Compiled by Jonathan Cameron, jcameron@redhat.com
Find source code that accompanies this exercise at:
https://github.com/JonoCameron/kernel_workshop

Workshop Goals and rules

- To give you an understanding of how to build, debug and install your own Linux Kernel
- Add counters to monitor page-cache hits and misses, that you will make visible in the /proc filesystem.

Workshop Description

You will go through the steps to:

1. Build your own kernel (linux master really)
2. Build and deploy a simple kernel module
3. Start the kernel in a debugger
4. Patch the kernel
5. Define and add a new syscall
6. Add page-cache hit and miss counters to the Linux Kernel
7. Export the counters to the /proc filesystem
8. Implement a user program that exercises the page-cache using file read()/write()
9. Implement a user program that monitors and displays page-cache hit and miss ratios.

Given the long number of steps, the following table of contents may be useful as you navigate through the project.

Monospaced instructions are commands you should enter.

… comment
$ … prompt on the actual lab machine
% … prompt in your VM

Configure your Virtual Machine

For this project, you need to do your work on a Virtual Machine we have created for you.

Whilst working through this tutorial, my host machine was running Fedora 32 and I was using "Boxes" to run the VM.

To launch the VM, do the following on your personal computer, or lab machine.

Please note that the machine you're using may not have the specifications to use /tmp. Therefore, you should consider making a directory to unzip the VM to.:

```
$ cd /tmp/ //or mkdir kernel_workshop

# extract the VM to local machine
$ tar xf ~/f31.tgz
# ~ 1 minute

# start the VM
$ vmware -x /tmp/f31/f31.vmx
# wait for the VM to boot

# log into the VM (check the ip address in the VMware window)
$ ssh user@192.168.62.143
```

Alternatively

If you are using a Fedora machine, you can run the VM in Boxes. (Just find a way to run the VM.)

```
# extract the VM to your local machine (put it in any folder you like).
$ tar xf ~/f31.tgz
```

Use Boxes to run the f31.vmdk VM.

```
% login: user
```

% password: user

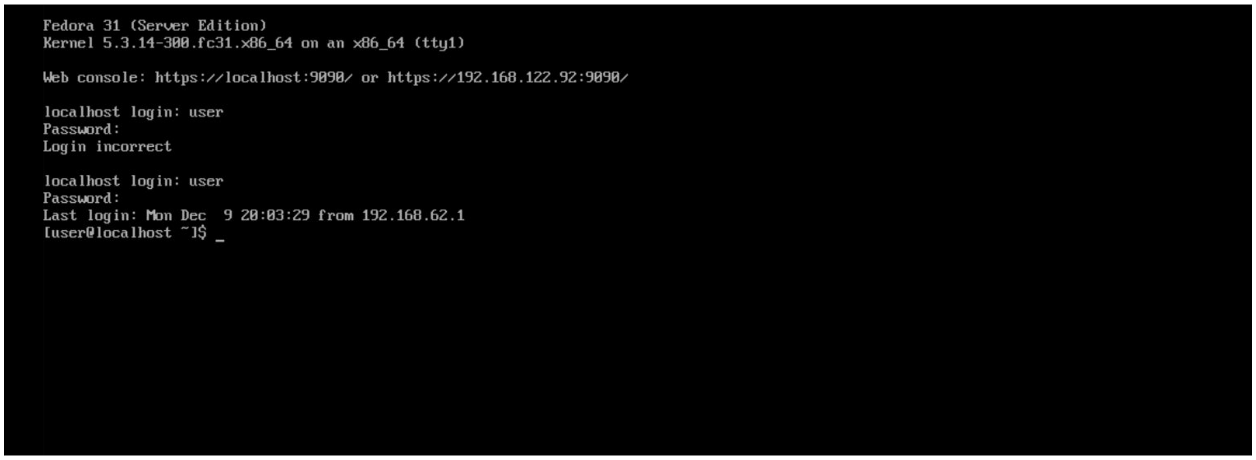
It’s not strictly necessary to log into the VM from another terminal but it does make life easier.

% ip addr

Figure out your VM's IP and use it in the next command

$ ssh user@192.168.62.143

The VM should look like this:



```
Fedora 31 (Server Edition)
Kernel 5.3.14-300.fc31.x86_64 on an x86_64 (tty1)

Web console: https://localhost:9090/ or https://192.168.122.92:9090/

localhost login: user
Password:
Login incorrect

localhost login: user
Password:
Last login: Mon Dec  9 20:03:29 from 192.168.62.1
[user@localhost ~]$ _
```

Update the kernel and download some useful packages

%sudo yum -y update
%sudo init 6

MAKE SURE YOU REBOOT THE CORRECT KERNEL. Check the most recent version with:

% uname -srm

```
% sudo dnf groupinstall "C Development Tools and Libraries"
% sudo dnf install elfutils-devel kernel-devel
% sudo dnf install git
% sudo dnf install openssl-devel ncurses-devel gcc make ctags bison
     flex elfutils-libelf-devel bc wget perl
```

```
%sudo init 6
```

Building a vanilla upstream kernel

Building an upstream Linux kernel is easy. Simply download the code, configure the kernel, compile the kernel, install the kernel, then reboot your (virtual) machine and *make sure to boot the new kernel*.

Cloning (downloading) the upstream kernel tree

```
# clone the linux source
% git clone --depth=1 https://github.com/torvalds/linux.git
```

Configuring the kernel

Change directory to the newly downloaded kernel directory. Always remember to clean-up the source tree to get rid of any left-over from previous builds

```
# change to the directory
% cd linux

# make a build directory
% mkdir build

# clean up stale content
% make mrproper && make clean
```

Assuming the current running kernel is older than the kernel you're about to build (updating the kernel to a newer upstream version/release), the quickest way to configure the new kernel is to base the config options out of the config of the running kernel. On Fedora hosts, one can check the configuration of the current running kernel by accessing the available /boot/config-$(uname -r) file.

The Linux kernel build infrastructure expects the config options to be declared within a .config file in the source directory, therefore to get the vanilla upstream kernel set up as a generic Fedora kernel, one must perform:

```
# copy the config of the current system as a basis
% sudo cp /boot/config-$(uname -r) build/.config

# use the current config with the new kernel
% make O=build olddefconfig
```

(NOTE: The second step is required in order to update the current config file, utilizing the provided .config while setting any new config symbol to its default value without prompting questions about these symbols setup.)

If further customisation is wanted/required, one can resort to a ncurses interface offered by the kernel config infrastructure to make the set up process easier.

```
# optionally run menuconfig
% make O=build menuconfig
```

Generic distro kernels (e.g., Fedora’s) commonly have hundreds or thousands of kernel components configured as modules. This is useful to ensure that the resulting kernel + modules can run on the largest variety of computer systems. However, more modules implies longer kernel compile times (i.e., a default config will easily take multiple hours to compile). Given that we know exactly which machine the new kernel will run on (i.e., the VM that we’re currently operating in), we can speed up the compile time by only building the modules we actually need.

```
# update config to only build the necessary modules (if you miss
this, the next steps will take hours)
% make O=build localmodconfig
```

If you don’t have some special configurations that you want to use from your current kernel, it is better to just do a simple make menuconfig instead of copying your old configuration file. You can simply save and close the GUI that comes up. This will also turn on kernel debugging by default so that you will have symbols while debugging the kernel.

Building the kernel

Now it is time to actually compile the kernel. It can take a long time but we can run this task on different cores simultaneously and get it done faster. To find out the number of cores your machine has, type ‘nproc’ command and it will give you a number n. Depending on your hardware, you’ll want to use a multiple of n parallel processes to compile the kernel. Some machines run on slow spinning rust hard drives. Hence, the compiler will often wait for IO to read source files from the disk and write the object files back. Choosing the correct n is an art, but with spinning rust HDDs choosing n = 2*#cores is a decent choice.

To compile the kernel, use the following command

```
# build the kernel & modules
% sudo dnf install dwarves
% time make O=build -j8 bzImage modules
```

This step will take some time. If you configured the kernel and modules correctly, as described above, it should take about 10-15 minutes on a Red Hat machine.

Installing the new kernel on the your machine

Imagine, you made really cool changes to the kernel that you want to deploy on the same machine you are working on. You can install the new kernel on your machine in the following way.

(NOTE: for the next steps, you _must_ be root or run via sudo)

```
# install the modules and kernel we just built
% sudo make O=build modules_install install

# set new kernel as default to boot
% sudo grub2-set-default 0

# reboot
% sudo init 6
```

After the reboot, log back in and check what kernel-version is running

```
$ ssh user@192.168.62.143

% uname -srm
```

(NOTE1: replaying the above steps without changing "EXTRAVERSION" will cause the kernel image and modules to be overridden by the recently compiled images. If overwriting the kernel artifacts is not the intention, EXTRAVERSION must be incremented for every build/install.)

(NOTE2: these are the basic steps to build an upstream kernel and install it on a Fedora 31 system. From here, kernel development labs can be developed)

Change EXTRAVERSion string in kernel Makefile

Let’s make a small change to the kernel, recompile, and observe that we run the newly compiled kernel. An easy thing is to change the EXTRAVERSION string of the kernel.

```
# log back into the VM (after it rebooted)
$ ssh user@192.168.62.143

# edit the kernel’s main makefile
% cd linux

% nano Makefile
# edit the EXTRAVERSION string (put a number in there) and add your
name as -name
# then save the changes to the Makefile

# rebuild kernel & modules
% time make O=build -j8 bzImage modules

# install
% sudo make O=build modules_install install

# reboot
% sudo init 6

# log back into the VM (after it rebooted)
$ ssh user@192.168.62.143

# observe the new kernel version
% uname -srm
```

Building (out of the tree) kernel modules

It's usually not necessary to download and rebuild the entire kernel if the objective is just to (re)build a module. To build a kernel module for the currently running kernel, only the C development tools, elfutils-devel and the matching kernel-devel RPMs are required (already installed in the VM)

A skeleton for a dummy-kmod

These are the bare-bones to start with building out of tree kernel modules:

```
% cd
% git clone https://github.com/megele/dummy-kmod
% cd dummy-kmod
% make
```

(If you can't clone into the git repo for whatever reason, the code can be found in the appendix. Make sure you create a new directory in /home/user. You also may want to change the name of the file you're compiling so the MAKEFILE works)

```
# load the kernel module
% sudo insmod hello.ko


# inspect the effect of loading the module
% dmesg

# unload the kernel module
% sudo rmmod hello

# inspect the effect of loading the module
% dmesg

# modify the kernel module
% nano hello.c
# change the string to something individual

# make and load again
```

% make && sudo insmod hello.ko && dmesg

Loading the module into the running kernel:

```
> [student@localhost dummy-kmod]$ sudo insmod hello.ko
> [student@localhost dummy-kmod]$ dmesg
> ...
> [ 1988.537936] hello: loading out-of-tree module taints kernel.
> [ 1988.537958] hello: module verification failed: signature and/or required key missing
- tainting kernel
> [ 1988.538067] Hello, kernel world!
```

(NOTE: here's an opportunity to discuss about KERNEL_TAINT flags and their meanings, as well as to discuss about signed modules and their importance. Signing the module binary code can be explored later)

Unloading the module:

```
> [student@localhost dummy-kmod]$ sudo rmmod hello
> [student@localhost dummy-kmod]$ dmesg
> ...
> [ 1988.538067] Hello, kernel world!
> [ 2125.301147] Goodbye, kernel world!
```

Suggestions to expand the skeleton module, as bonus exercises:

1. extend the module to take different strings to show at console from the module parameters list (using module_param() macro);
2. export kernel internal constants, like HZ and USER_HZ, to userspace via:
    a. module parameters, which will automatically will be exported to sysfs interface: /sys/module/<modname>/parameters/<params>
    b. procfs entries;

(NOTE: as soon as students realize the power of extending the kernel with out of tree modules, almost any task can be accomplished via this approach. Further reading on the topic: Corbet et al. Linux Device Drivers, Third Edition: https://lwn.net/Kernel/LDD3/)

Running the new kernel with QEMU

Now that we have compiled our kernel, we want to boot it up. For that we need something called an initramfs or initrd which is essentially a very small, memory based file system. It contains all the libraries and code required to boot up the kernel. Once the kernel boots up, it can load the actual file system and do normal operations. We use a tool called supermin to create this initrd.

Preparing to run the kernel

To install supermin, use the following command

```
% sudo dnf install supermin
```

Supermin works in two stages, prepare and build. Prepare is where we specify what packages and libraries to include in the initrd. Build is where this initrd actually gets made. Here are simple commands we can use to prepare a simple initrd.

Leave the kernel directory and go up one level by doing a cd ..

```
# make initrd & rootfs
% cd
% mkdir initrd && cd initrd
% supermin --prepare bash coreutils -o supermin.d
% echo -e '#!/bin/bash\necho Welcome\nexec bash' > init
% chmod 0755 init
% tar zcf supermin.d/init.tar.gz init
% supermin --build --format ext2 supermin.d -o appliance.d
```

Running the kernel on QEMU

Now that we have a kernel and an initrd ready, we can now boot these up. There are many ways we can do this, we will choose qemu which is a virtual machine emulator.

Now let’s go up one directory to actually run everything. This step is not required but it’ll be easier to understand relative paths of kernel and initrd from this main directory when we boot up our kernel in the next step.

Assuming you are on a 64 bit x86 machine, the qemu command you will use is qemu-system-x86_64. This command takes a bunch of options telling it where the kernel is, where the initrd is etc. Below is a generic command which will run your kernel and show the output in the same terminal window. You can read up qemu man pages and change the options if you want.

```
> [student@localhost ~]$ qemu-system-x86_64 -nodefaults
-nographic -kernel <path to kernel image> -initrd <path to
initrd> -hda <path to root disk> -serial stdio -append
"console=ttyS0 root=/dev/sda nokaslr"
```

In the above command we need to provide path to kernel image, initrd and root disk.

The kernel image that we need to use on our x86 machines would be at a specific place in our Linux directory. The path in this example is linux-4.18.14/arch/x86/boot/bzImage. For initrd, the path would be initrd/appliance.d/initrd. And the root disk will be at this location initrd/appliance.d/root

The final command for this example would be something as follows.

```
# start qemu
% qemu-system-x86_64 -nodefaults -nographic -kernel
linux/build/arch/x86/boot/bzImage -initrd initrd/appliance.d/initrd
-hda initrd/appliance.d/root -serial stdio -append "console=ttyS0
root=/dev/sda nokaslr"
```

Sit back and observe qemu boot your new kernel.

If you get an error regarding undefined libusb symbols, this means your libusbx package needs to be updated. You need libusbx-1.0.22-1.fc28 version, anything older than this will not work. You can download and install the latest rpm yourself through the following commands

```
> [student@localhost ~]$ wget
http://dl.fedoraproject.org/pub/fedora/linux/updates/28/Everyth
ing/x86_64/Packages/l/libusbx-1.0.22-1.fc28.x86_64.rpm
> [student@localhost ~]$ rpm -U
libusbx-1.0.22-1.fc28.x86_64.rpm
```

Run the qemu command again and you will see your kernel booting up.

```
# exit qemu
bash-5.0# <ctrl-c>
```

Debugging the kernel with gdb

Once you make modifications to the kernel, you might want to debug it in case something is not working.

Changing QEMU command

Qemu gives a very nice interface for debugging with GDB. When running the kernel with Qemu, add two options to the command i.e., -s -S. These two options will make sure that you can step through the kernel code.

The final qemu command will be as follows.

```
# start qemu in debug mode (add -s -S to above)
% qemu-system-x86_64 -s -S -nodefaults -nographic -kernel
linux/build/arch/x86/boot/bzImage -initrd initrd/appliance.d/initrd
-hda initrd/appliance.d/root -serial stdio -append "console=ttyS0
root=/dev/sda nokaslr"
```

Once you enter the above command, you will see that the kernel does not boot up. It is actually waiting for you to step through it. This is where GDB comes in.

Starting GDB

In a separate terminal window, open GDB by typing gdb as follows.

```
# in a different terminal (e.g., ssh again)
% gdb ~/linux/build/vmlinux

(gdb) target remote localhost:1234
0x000000000000fff0 in exception_stacks ()

(gdb) x/3i $rip
```

Stop the kernel and find the corresponding source

While booting up the kernel in qemu with GDB attached, stop the kernel’s boot process (press ctrl-c in GDB). Then GDB will show which function the kernel is currently in.

For example:

```
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
0xffffffff813f2f8e in ext4_mpage_readpages
(mapping=0xffff8880060b5400,
```

How can we find the ext4_mpage_readpages() function in the 2 million lines of kernel source?

We could “grep” the entire source, which is awfully slow and really a terrible way to navigate source code. Hence, cscope to the rescue. With cscope we can build an index of the source tree once, and then reuse that index to quickly find/jump to any symbol we want.

```
# install cscope and vim
% sudo dnf install cscope vim

# navigate to the linux source directory
% cd ~/linux

# run cscope to build the index (-b)
# recursively over all files in the directory structure (-R)
% cscope -R -b

# use vim to open whichever C file (at the correct position) has the
# implementation of the function we’re looking for
% vim -t ext4_mpage_readpages

# enter :q (that is colon and the letter q) to exit out of vim
```

Kernel Patching

In the Linux Kernel, changes to the source code are implemented via kernel patches. These kernel patches are unified diffs of the source repository and can be seen by the output of the

```
% cd linux
% git log
```

command.

Creating a kernel patch

Edit the kernel source files and save.

```
% git diff
```

produces the patch file associated with the changes.

Use git commit & git format patch to create a patch series.

```
% git commit -a       //commit your changes to the git repo
% git log             //locate commit ID of kernel before changes
```

```
% git format-patch -1 HEAD
```

Will produce something like:

<0001-"commit message".patch>

Applying and removing a patch to the unchanged kernel

```
% patch -p1 -R < 0001-"commit-message".patch //check the Makefile
```

Will remove the patch you just made, and revert to the previous commit.

You can reapply the patch with:

```
% patch -p1 < 0001-"commit message".patch //Check the Makefile again
                                             to see your changes added
                                             as a patch.
```

An alternative(if you mess up your kernel sources :)) is to remove everything from your kernel and reset the sources back to the version you got with the git clone command is:

```
% git reset --hard <commit ID> //Use the commit ID of the original
                                 commit from Torvalds.
```

Example:

```
% git diff
```

will show your changes

```
     <all of your changes here>...

% git log
     commit 5f21585384a4a69b8bfdd2cae7e3648ae805f57d
     Merge: fcc37f7 9fe5c59
     Author: Linus Torvalds <torvalds@linux-foundation.org>
     Date:   Fri Nov 2 11:25:48 2018 -0700

% git reset --hard  5f21585384a4a69b8bfdd2cae7e3648ae805f57d

% git diff
     <no changes>
```

Sending the kernel patch

If git complains that send-mail does not exists or is not recognized then install it using:

```
% sudo dnf install git-email
```

NB: If you are using git send-email for the first time you would need to configure the smtp (outgoing mail) server settings using “git

config" utility. Type the following commands in the shell to send from bu's smtp server using your bu email address:
(general format) - sudo git config --global <section_name>.<key> <value>

If you don't have a BU email, you'll need to find the settings for your email account.

1. **% sudo git config --global sendemail.smtpuser <your-bu-email>**
2. **% sudo git config --global sendemail.smtpencryption ssl**
3. **% sudo git config --global sendemail.smtpserver smtp.bu.edu**
4. **% sudo git config --global sendemail.smtpserverport 465**

You can check if the configuration is what you added in the above 4 commands by either typing the following command without the <value>:
$sudo git config --global sendmail.<key>
OR
$cat /home/fedora/.gitconfig

When prompted by the utility to enter the recipient email address enter an email you have access to so you can see you've sent the patch:

% git send-email --suppress-cc=all 0001-"commit message".patch --from=<your-account>@bu.edu

Note: Since you are running git send-email on a different system than your bu.edu email is on you night need it include "**--from=<your-account>@bu.edu**" if your main is coming from your project account, ie: **MAIL FROM:<fedora@project4-31.moclocal>**

Note: pass the directory created with git format-patch that contains the patch, not the patch file itself. If you created a directory for patches

```
     [user@localhost linux]$ git send-email --suppress-cc=all
0001-syscall.patch --from=jcameron@bu.edu
     0001-syscall.patch
     To whom should the emails be sent (if anyone)?
jcameron@gmail.com
     Message-ID to be used as In-Reply-To for the first email (if
any)?
```

```
     Password for 'smtp://jcameron@bu.edu@smtp.bu.edu:465':
     OK. Log says:
     Server: smtp.bu.edu
     MAIL FROM:<jacc@bu.edu>
     RCPT TO:<jcameron@gmail.com>
     From: jcameron@bu.edu
     To: jcameron@gmail.com
     Subject: [PATCH] syscall
     Date: Wed, 24 Jun 2020 11:20:29 -0400
     Message-Id: <20200624152029.1303-1-jcameron@bu.edu>
     X-Mailer: git-send-email 2.25.4
     MIME-Version: 1.0
     Content-Transfer-Encoding: 8bit

     Result: 250



     In git 1.7.0, the default has changed to --no-chain-reply-to
     Set sendemail.chainreplyto configuration variable to true if
     you want to keep --chain-reply-to as your default.
```

```
And this is the email I(Who should the emails be sent to?
jcameron@gmail.com)
received:
```

```
     ---
      arch/x86/entry/syscalls/syscall_64.tbl | 2 ++
      include/linux/syscalls.h               | 2 ++
      kernel/sys.c                           | 6 ++++++
      3 files changed, 10 insertions(+)

     diff --git a/arch/x86/entry/syscalls/syscall_64.tbl
b/arch/x86/entry/syscalls/syscall_64.tbl
     index 78847b32e..03ef99a85 100644
     --- a/arch/x86/entry/syscalls/syscall_64.tbl
     +++ b/arch/x86/entry/syscalls/syscall_64.tbl
     @@ -343,6 +343,8 @@
      332    common  statx                   sys_statx
      333    common  io_pgetevents           sys_io_pgetevents
      334    common  rseq                    sys_rseq
     +335    common  add2int                 sys_add2int
     +
       # don't use numbers 387 through 423, add new calls after the
last
       # 'common' entry
       424    common  pidfd_send_signal       sys_pidfd_send_signal
      diff --git a/include/linux/syscalls.h
b/include/linux/syscalls.h
      index 7c354c295..5399fcf6b 100644
      --- a/include/linux/syscalls.h
      +++ b/include/linux/syscalls.h
      @@ -1424,4 +1424,6 @@ long compat_ksys_semtimedop(int semid,
struct sembuf __user *tsems,
                                  unsigned int nsops,
                                  const struct old_timespec32 __user
*timeout);

      +asmlinkage long sys_add2int(int a, int b);
       #endif
      +
      diff --git a/kernel/sys.c b/kernel/sys.c
      index 00a96746e..6567d65c0 100644
      --- a/kernel/sys.c
      +++ b/kernel/sys.c
      @@ -2691,4 +2691,10 @@ COMPAT_SYSCALL_DEFINE1(sysinfo, struct
compat_sysinfo __user *, info)
                      return -EFAULT;
              return 0;
       }
      +SYSCALL_DEFINE2(add2int, int, a, int, b)
      +{
      +       long result = a + b;
      +       printk(KERN_INFO "syscall: add2int: a=%i, b=%i,
result=%li\n", a, b, result);
      +       return result;
      +}
       #endif /* CONFIG_COMPAT */
```

****

To whom should the emails be sent (if anyone)? **<another email you have>**

Message-ID to be used as In-Reply-To for the first email (if any)? **<leave empty>**

...wait for a few seconds

```
Password for 'smtp://<your-bu-username>@bu.edu@smtp.bu.edu:465':
<enter your password for bu email>
```

If email is successfully sent then you will get an “OK Log says:” message on the stdout of your shell as in the example output above.

****

Defining a new System Call and patching the kernel to include it.

Adding a new system call to Linux is simple:

1.) Add your new syscall to the end of the syscall table in arch/x86/entry/syscalls/syscall_64.tbl, taking the next available system call number.

2.) Add the appropriate asmlinkage declaration in include/linux/syscalls.h

3.) Add your new syscall code in kernel/sys.c using the SYSCALL_DEFINE2(...) macro.
(FYI the macros do most of the work for you automagically)

Find the code in the appendix under add2int.

Rebuilding the kernel to add the new system call.

```
% time make O=build -j8 bzImage modules

% sudo make O=build modules_install install

% sudo init 6
```

Write a test program

```
% cd
% nano test.c
```

Code is in the appendix

```
% gcc test.c -o test
% ./test 5 3
```

Adding page-cache hit/miss counters

This exercise is to see the page-cache hit and miss, by adding two new files to the /proc filesystem and then exercising the page-cache.

1) **Create a kernel patch to:**
   a) Add page-cache hit and miss counters to the kernel function pagecache_get_page().
      i) In the file linux/mm/filemap.c define two global integers, pagecache_miss and pagecache_hit. They must be defined before and outside of the pagecache_get_page() function.
      ii) In the same file, the function pagecache_get_page() calls find_get_entry() to determine if the requested page is in the page-cache. If find_get_entry() returns a non-zero value, the page is in the page-cache (a hit), but if find_get_entry returns a zero value, the page is not in the page-cache (a miss). You need to add the code that increments the hits and misses accordingly.
   b) Export these counters to /proc/sys/vm/pagecache_hits and /proc/sys/vm/pagecache_misses respectively. This is *relatively* easy since the macros do most of the work. In fact you only need to add an entern and a table entry for each.
      i) In the file linux/kernel/sysctl.c, define two externs for the counters you defined in mm/filemap.c. For examples, look for the comment "External variables not in a header file". (A bit of googling for this might help, depending if said comment exists in your source code.)
      ii) In the same file, locate the "static struct ctl_table vm_table[]" array and add two new entries after the first one, which is for "overcommit_memory".

```
static struct ctl_table vm_table[] = {
{
        .procname       = "overcommit_memory",
        .data           = &sysctl_overcommit_memory,
        .maxlen         = sizeof(sysctl_overcommit_memory),
        .mode           = 0644,
        .proc_handler   = proc_dointvec_minmax,
        .extra1         = &zero,
        .extra2         = &two,
},
{
        Your first entry goes here…
},
{
        Your second entry goes here…
},
{
        procname        = "panic_on_oom",
        ……..
```

All you need to do is copy and paste the entire overcommit_memory entry twice, right after the overcommit_memory entry, one for the pagecache hits and one for the pagecache misses. Then change the procname, data, maxlen fields to match whatever pagecache hit or miss variable names you chose. The mode can remain the same since you want the files and associated variables read-only by everyone except root. The proc_handler can also remain the same generic integer handler as long as your hit and miss counters are integers. Finally you can delete the extra1 and extra2 fields since you will not use them. For both entries make sure you include the beginning & ending braces and the comma separator:

```
{
                .procname     = ...
                .data         = …
                .maxlen       = …
                .mode         = …
                .proc_handler = …
},
```

After making these changes, you should rebuild, install and reboot your kernel without any problems. Once the new kernel is up and running you should see your new files in /proc/sys/vm and should see their values increasing as the page-cache hits and misses respectively. Since the mode is 0644 they are read-only to the group and others but read-write to root, so you can reset the values if you wish.

2) **Write a simple filesystem exerciser that writes/reads a large file (4GB or 1 million 4KB pages)**
   a) One mechanism
      i) Create a large file using O_CREAT
      ii) malloc() a 4KB buffer and fill it with junk
      iii) Write the buffer to the file 1 million times
      iv) Close the file
      v) Flush the page-cache in another window, using the hint below
      vi) Open the file using O_READ
      vii) Read the entire file, you should see lots of page-cache misses
      viii) Close the file
      ix) Open the file using O_READ
      x) Read the entire file, you should see lots of page-cache hits
      xi) Close the file
   b) Explore using mmap() to read the file and also try reading the file front-to-back Vs. back-to-front.

3) **Write a utility that counts the page-cache hits and misses via /proc and prints out the page-cache hit and miss ratios every second.**
   a) Open the page-cache hit and miss files in /proc
      i) In a loop:
         (1) Read hits and misses
         (2) Sleep for 1 second
         (3) Read hits and misses
         (4) Calculate and print differences and ratios

      Hint: The contents of the files in /proc are NULL terminated character strings. Read them into a character array and then use atoi() to convert.
      Hint: Think about where the file pointer is after reading the file. lseek() may help you out…

4) **Demonstrate what the hit/miss ratio is when data is cached against when it is not cached, and then obverse what the hit/miss ratio is when the filesystem read-ahead is turned off altogether.**
   a) To drop all of the cache, use "**sudo su**" to run "**echo 1 > /proc/sys/vm/drop_caches**" or run the command "**sudo sh -c "/usr/bin/echo 1 > /proc/sys/vm/drop_caches"**"

b) To turn off read-ahead, use **"sudo su"** to type:
"for f in */sys/block/; do echo 0 > $f/bdi/read_ahead_kb; done"**

This is what you'll see with a page-cache that has been dropped, lots of misses, not many hits:

```
hit_ratio:  0.52 (69) miss_ratio:  0.48 (64)
hit_ratio:  0.00 (0) miss_ratio:  0.00 (0)
hit_ratio:  0.00 (0) miss_ratio:  0.00 (0)
hit_ratio:  0.00 (17) miss_ratio:  1.00 (421504)
hit_ratio:  0.00 (0) miss_ratio:  1.00 (269873)
hit_ratio:  0.00 (0) miss_ratio:  1.00 (99865)
hit_ratio:  0.00 (0) miss_ratio:  1.00 (85261)
```

And if you have disabled look ahead, you'll see times increase when reading the large file you made:

```
[user@localhost documents]$ time ./read
bigFILE was opened
Closed bigFILE

real    0m5.452s
user    0m0.026s
sys    0m1.306s
```

Vs.

```
[user@localhost documents]$ time ./read
bigFILE was opened
Closed bigFILE

real    2m12.648s
user    0m0.160s
sys    0m8.274s
```

A New System Call

The first system call we defined above (add2int) was fairly primitive, BUT it did show you the basic steps needed to implement a new syscall. The code we need to write to read from the /proc filesystem is also quite a lot, so what if we wrote a syscall to read those two numbers for us? All we would need to do was pass it a buffer for each value and it would do a lot of the work for us in kernel-space.

For this next exercise, implement a syscall that will return the number of pagecache_hits and pagecache_misses that can be used in a program. Remember, you will need to

1. Add the new system call to the syscall table
2. Add the appropriate asmlinkage to syscalls.h
3. Add your new syscall to sys.c

More detail of the steps can be found above where we defined add2int.

HINT: A function you will need to use is copy_to_user() because of how virtual memory is assigned between user-space and kernel-space. You may struggle getting data out of kernel-space without it.

HINT: You'll want to think about how to read the files from kernel-space. You cannot use system calls in system calls- this is because the OS thinks that you are in user-space when you call a system call.

An example implementation can be found in the appendix and on the github repo. (It worked on my machine ;))

Appendix

Dummy-kmod

The Makefile:

```
> [student@localhost dummy-kmod]$ cat Makefile
> obj-m += hello.o
>  KDIR:=/lib/modules/$(shell uname -r)/build
> all:
```

```
>         $(MAKE) -C $(KDIR) M=$(PWD) modules
>
> clean:
>         $(MAKE) -C $(KDIR) M=$(PWD) clean
>
```

The module code:

```
> [student@localhost dummy-kmod]$ cat hello.c
> /* hello.c -- a very simple kernel module */
> #include <linux/kernel.h>
> #include <linux/init.h>
> #include <linux/module.h>
>
> MODULE_LICENSE("GPL");
> MODULE_AUTHOR("Your Name Here <you@mail.srv>");
> MODULE_DESCRIPTION("A simple hello world kernel module.");
> MODULE_VERSION("1.0");
>
> static int __init hello_init(void)
> {
>         printk(KERN_INFO "Hello, kernel world!\n");
>         return 0;
> }
>
> static void __exit hello_exit(void)
> {
>         printk(KERN_INFO "Goodbye, kernel world!\n");
> }
>
> module_init(hello_init);
> module_exit(hello_exit);
```

add2int

Editing the syscall table.

```
% cd linux
% nano arch/x86/entry/syscalls/syscall_64.tbl
```

Add a line after entry #334

```
% 335      common  add2int       sys_add2int
```

Save and quit

Editing the syscall header file

```
% cd linux
% nano include/linux/syscalls.h
```

Add a line

```
% asmlinkage long sys_add2int(int a, int b);
```

Save and quit

Editing the syscall file

```
% cd linux
% nano kernel/sys.c
```

Add the block

```
SYSCALL_DEFINE2(add2int, int, a, int, b)
{
     long result = a + b;
     printk(KERN_INFO "syscall: add2int: a=%i, b=%i, result=%li\n", a,
     b, result);
     return result;
}
```

Save and exit.

test.c

```
/*
  * Test the add2int syscall (#335)
  */
 #define _GNU_SOURCE
 #include <unistd.h>
 #include <stdlib.h>
 #include <sys/syscall.h>
 #include <stdio.h>
```

```
#define SYS_add2int 335

int main(int argc, char **argv)
{
     int a, b;

     if (argc <= 2) {
          printf("Must provide 2 integers to pass to the system
          call.\n");
          return -1;
     }
     a = atoi(argv[1]);
     b = atoi(argv[2]);

     printf("Making system call with a=%d, b=%d\n", a, b);
     long res = syscall(SYS_add2int, a, b);
     printf("System call returned %ld.\n", res);

     return res;
}
```

A less primitive syscall

The code added to the kernel.

```
+++ b/arch/x86/entry/syscalls/syscall_64.tbl
@@ -344,6 +344,7 @@
 333    common  io_pgetevents        sys_io_pgetevents
 334    common  rseq                 sys_rseq
 335    common  add2int              sys_add2int
+336    common  pagecache_counter          sys_pagecache_counter

 # don't use numbers 387 through 423, add new calls after the last
 # 'common' entry
diff --git a/include/linux/syscalls.h b/include/linux/syscalls.h
index 5399fcf6b..e7816d2bc 100644
--- a/include/linux/syscalls.h
+++ b/include/linux/syscalls.h
```

```
@@ -1425,5 +1425,6 @@ long compat_ksys_semtimedop(int semid, struct sembuf __user
*tsems,
                                 const struct old_timespec32 __user *timeout);

 asmlinkage long sys_add2int(int a, int b);
+asmlinkage int sys_pagecache_counter(char* hits, char* miss);
 #endif

diff --git a/kernel/sys.c b/kernel/sys.c
index 6567d65c0..b11ac91cc 100644
--- a/kernel/sys.c
+++ b/kernel/sys.c
@@ -2697,4 +2697,29 @@ SYSCALL_DEFINE2(add2int, int, a, int, b)
        printk(KERN_INFO "syscall: add2int: a=%i, b=%i, result=%li\n", a, b, result);
        return result;
 }
+
+SYSCALL_DEFINE2(pagecache_counter, char*, hits, char*, miss)
+{
+       int fd0, fd1;
+       int succ0, succ1;
+       char hitsbuf[8];
+       char missbuf[8];
+
+       mm_segment_t old_fs = get_fs();
+       set_fs(KERNEL_DS);
+
+       fd0 = ksys_open("/proc/sys/vm/pagecache_hits", O_RDONLY, 0);
+       ksys_read(fd0, hitsbuf, 8);
+       ksys_close(fd0);
+       succ0 = copy_to_user(hits, hitsbuf, 8);
+
+       fd1 = ksys_open("/proc/sys/vm/pagecache_misses", O_RDONLY, 0);
+       ksys_read(fd1, missbuf, 8);
+       ksys_close(fd1);
+       succ1 = copy_to_user(miss, missbuf, 8);
+
+       set_fs(old_fs);
+
+       return 0;
+}
 #endif /* CONFIG_COMPAT */
```

The code used for testing

```
#include <stdio.h>
#include <sys/syscall.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define SYS_pagecache_counter 336

int main(){
    char hits[8];
    char miss[8];

    int numhits, nummiss;

    long ret = syscall(SYS_pagecache_counter, hits, miss);

    printf("syscall returned: %ld\n", ret);

    numhits = strtol(hits, (char**)NULL, 10);
    nummiss = strtol(miss, (char**)NULL, 10);

    printf("numhits: %d. nummiss: %d\n", numhits, nummiss);

    return 0;
}
```