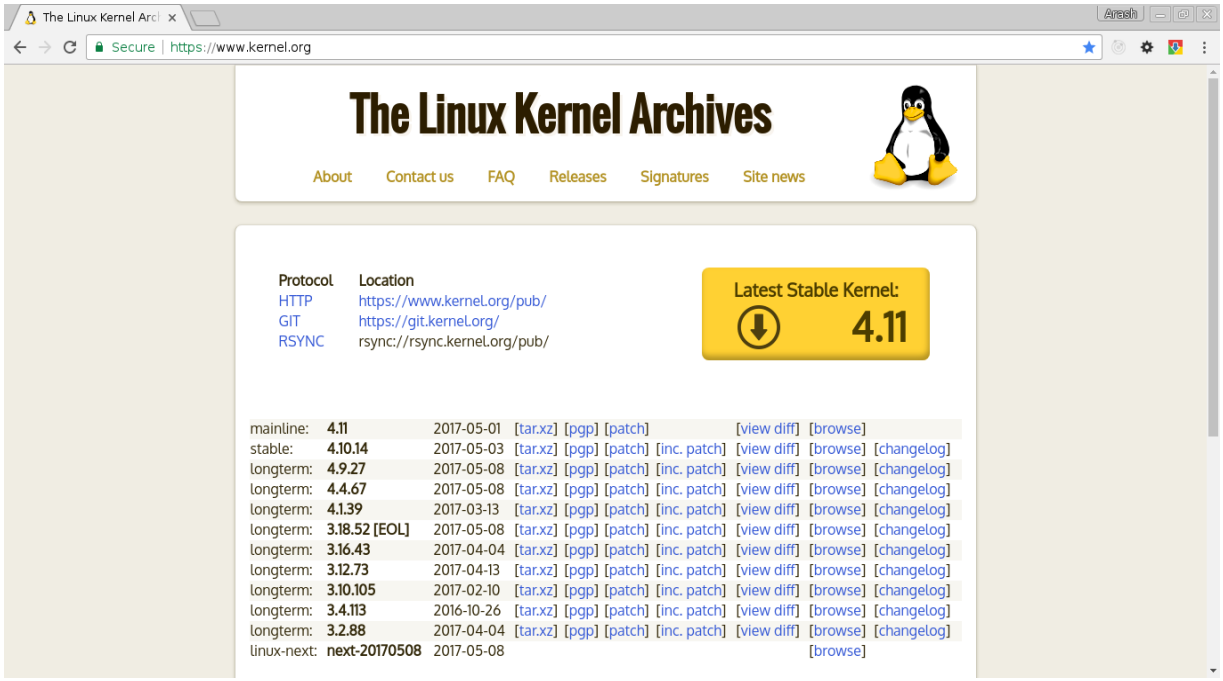
Compiling the Linux Kernel

1. download latest stable Linux kernel from https://www.kernel.org



2. extract it somewhere in the file system (ex. /usr/src/ directory)

with the following command:

```
arash@arash:~/Downloads/SOURCE$ tar -xf linux-4.10.8.tar.xz -C /usr/src/
```

(note that your version could be different!)

3. open the terminal and change directory to where you had extracted it

```
arash@arash:~$ cd /usr/src/linux-4.10.8/
```

4. create a folder for your future system calls (ex. “custom” directory)

```
arash@arash:/usr/src/linux-4.10.8$ mkdir custom
```

5. change directory (cd) to it and create a source file for your custom system call

```
arash@arash:/usr/src/linux-4.10.8$ cd custom/
arash@arash:/usr/src/linux-4.10.8/custom$ nano hello.c
```

(note that I use “nano” for my text editor and choose “hello” for my source code name, you can use any text editor you like and choose any name that best describe functionality of your system call)

6. write the following program in the “hello.c” file.

```
 GNU nano 2.2.6

#include <linux/kernel.h>

asmlinkage long sys_hello(void)
{
        printk("Hi! This is my first system call!\n");
        return 0;
}
```

press “Ctrl+x” and “y” then ”Enter” to save and close.

(this simple program, write a string of text to the kernel log message buffer, that we will see after compiling and booting the new kernel)

7. create a “Makefile” for the program.

```
arash@arash:/usr/src/linux-4.10.8/custom$ nano Makefile
```

(Makefiles are configuration files for “Make”, the build system that Linux kernel use. For more information about “Build Systems” and “Make” search the web.)

8. write the following in the Makefile that you just created.

```
 GNU nano 2.2.6

obj-y := hello.o
```

9. now we have to address our custom system call folder in the kernel main Makefile.

```
arash@arash:/usr/src/linux-4.10.8/custom$ cd ..
arash@arash:/usr/src/linux-4.10.8$ nano Makefile
```

10. search for “core-y” in the Makefile and add the custom folder at the end of it, then save and close the file.

```
ifeq ($(KBUILD_EXTMOD),)
core-y          += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ custom/
```

(if you are using nano, press “Ctrl+W” to search and use “Alt+W” to see the next result)

11. add the system call in your preferred architecture System Call Table.

I have a x86_64 Linux distribution installed on my computer so I'm going to add it here:

```
arash@arash:/usr/src/linux-4.10.8$ cd arch/x86/entry/syscalls/
arash@arash:/usr/src/linux-4.10.8/arch/x86/entry/syscalls$ nano syscall_64.tbl
```

at the end of 64-bit system calls and before x32-specifics

```
322     64      execveat                sys_execveat/ptregs
323     common  userfaultfd             sys_userfaultfd
324     common  membarrier              sys_membarrier
325     common  mlock2                  sys_mlock2
326     common  copy_file_range         sys_copy_file_range
327     64      preadv2                 sys_preadv2
328     64      pwritev2                sys_pwritev2
329     common  pkey_mprotect           sys_pkey_mprotect
330     common  pkey_alloc              sys_pkey_alloc
331     common  pkey_free               sys_pkey_free
332     common  hello                   sys_hello

#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation.
#
512     x32     rt_sigaction            compat_sys_rt_sigaction
513     x32     rt_sigreturn            sys32_x32_rt_sigreturn
514     x32     ioctl                   compat_sys_ioctl
```

(note that the number 332 happened to be my next system call number, yours my be different)

12. now we should add it to the Linux System Calls Header file.

```
arash@arash:/usr/src/linux-4.10.8/arch/x86/entry/syscalls$ cd ../../../../
arash@arash:/usr/src/linux-4.10.8$ cd include/linux/
arash@arash:/usr/src/linux-4.10.8/include/linux$ nano syscalls.h
```

at the end right before the “#endif”

```
asmlinkage long sys_mlock2(unsigned long start, size_t len, int flags);

asmlinkage long sys_pkey_mprotect(unsigned long start, size_t len,
                                  unsigned long prot, int pkey);
asmlinkage long sys_pkey_alloc(unsigned long flags, unsigned long init_val);
asmlinkage long sys_pkey_free(int pkey);

asmlinkage long sys_hello(void);

#endif
```

13. install the prerequisites for building the kernel.

```
arash@arash:/usr/src/linux-4.10.8/include/linux$ cd ../../
arash@arash:/usr/src/linux-4.10.8$ sudo apt-get install linux-headers-$(uname -r) build-essential
```

(note that in the process of compiling, you may encounter different error messages and warnings, you have to solve them by your self, searching the web and asking for help could be valuable resources in these situations)

14. copy your distribution kernel config with the following command.

```
arash@arash:/usr/src/linux-4.10.8$ cp /boot/config-$(uname -r) .config
```

15. change it to meet your needs with “menuconfig” (or “oldconfig” or “xconfig” or “qconfig” [search the web to find out how they differ from each other!])

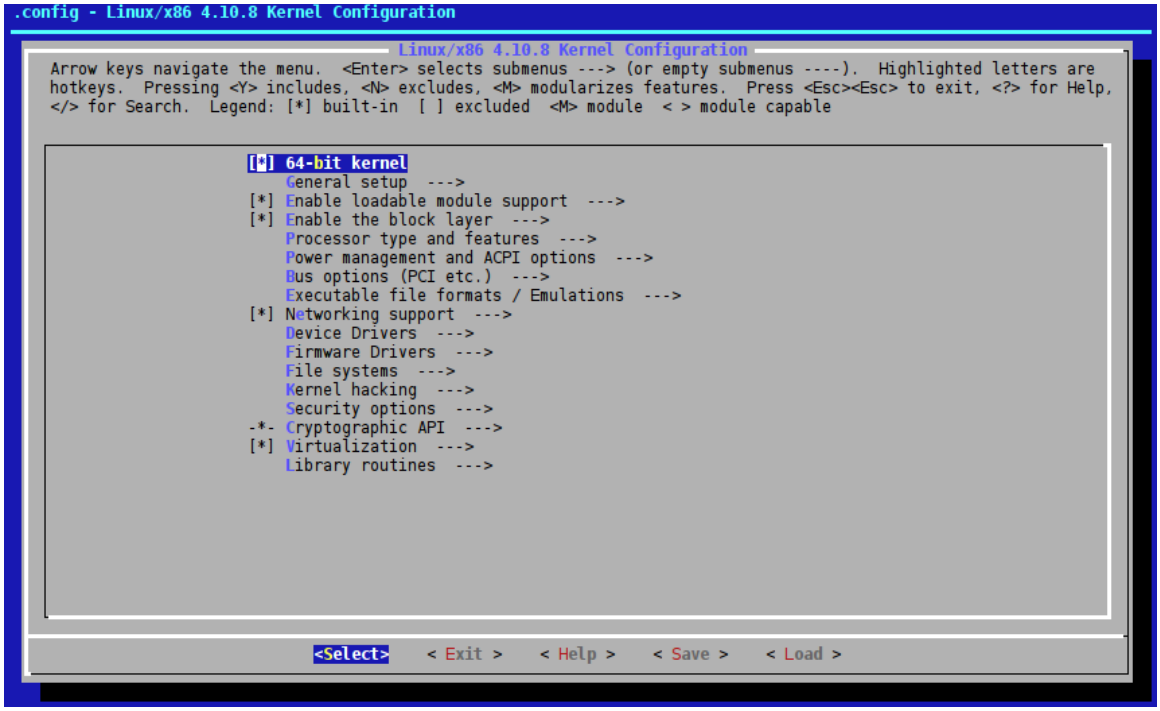
first install “menuconfig” dependencies:

```
arash@arash:/usr/src/linux-4.10.8$ sudo apt-get install libncurses5-dev libncursesw5-dev
```

and start it with the following command:

```
arash@arash:/usr/src/linux-4.10.8$ sudo make menuconfig
```

you should see something like this:



if you don't want to change anything just “Save” and “Exit”

16. compile and install the kernel with the following command:

```
arash@arash:/usr/src/linux-4.10.8$ sudo make -j 4 && sudo make modules_install -j 4 && sudo make install -j 4
```

“-j 4” tell Make to use 4 processor cores concurrently (to speed up the process) change it to match your processor specification. If you don't know how many cores you have available use the following command:

```
arash@arash:/usr/src/linux-4.10.8$ nproc
4
```

17. now you can reboot and you should see new kernel added to the grub boot-loader. boot with the new kernel and test it with the following command:

```
arash@arash:/usr/src/linux-4.10.8$ uname -r
```

you should see version of the new kernel.

18. test the system call with the following program:

```
arash@arash:~$ nano custom_system_call.c
```

```
 GNU nano 2.2.6                                          File: custom_system_call.c

#include <stdio.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>
int main()
{
        long int syscall_return_value = syscall(332);
        printf("System call sys_hello returned %ld\n", syscall_return_value);
        return 0;
}
```

(note that you should use your system call number instead of “332”)

19. build and run it.

```
arash@arash:~$ gcc custom_system_call.c -o custom_system_call
arash@arash:~$ ./custom_system_call
System call sys_hello returned 0
```

you should see it returns 0.

20. now you can see what your system call logged in the kernel log message buffer with the following command:

```
arash@arash:~$ dmesg
```

and you should see something like this at the end of it

```
[  544.295072] Hi! This is my first system call!
```

21. enjoy and understand what you did :)