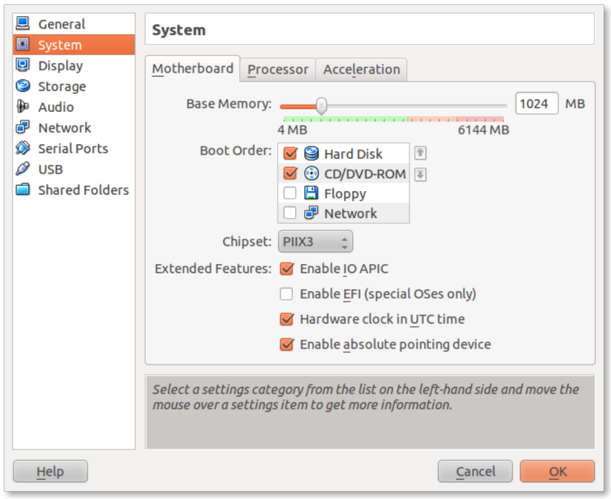
CS 360 Fall 2013
Adding a System Call to the Linux Kernel

In this project you will study the system-call interface provided by the Linux operating system and learn how user programs communicate with the operating system kernel via this interface. Your task is to incorporate a new system call into the kernel, thereby expanding the functionality of the operating system.

Part 0: Preliminaries

Since this assignment may require multiple kernel builds (a computationally intensive task), you will want to enable additional processors in your virtual machine to make better use of your dual+-core machines. Under System and then Processor, enable the number of CPU's that you have. Hyperthreaded cores will make it look like you have twice as many physical cores than you actually have – if you have a dual core, choose 2 (not 4), and if you have a quad core, choose 4 (not 8). This will require you to enable IO APIC (basically, IO APIC enables communication between the cores). This shouldn't affect your install. Do this before you boot your machine. You can tell if it worked or not by running the `top` command once you boot and then pressing '1.' Do you see multiple CPUs show up at the top??



System

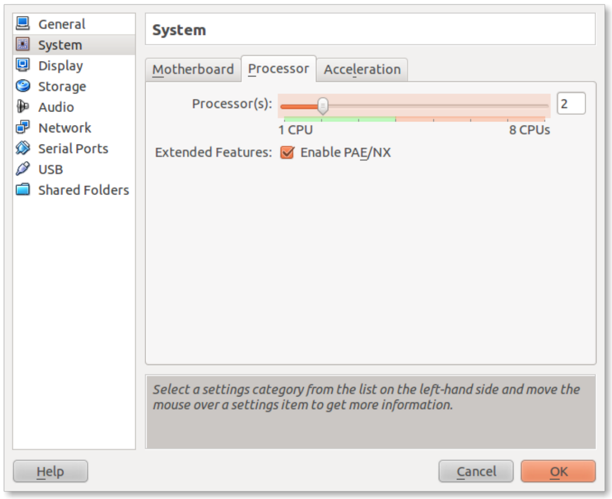
Base Memory: 1024 MB

Boot Order:
- ✓ Hard Disk
- ✓ CD/DVD-ROM
- Floppy
- Network

Chipset: PIIX3

Extended Features:
- ✓ Enable IO APIC
- Enable EFI (special OSes only)
- ✓ Hardware clock in UTC time
- ✓ Enable absolute pointing device

Select a settings category from the list on the left-hand side and move the mouse over a settings item to get more information.



System

Processor(s): 2

Extended Features: ✓ Enable PAE/NX

Select a settings category from the list on the left-hand side and move the mouse over a settings item to get more information.

Text in bold is information specific to our Ubuntu server virtualbox system. The non-bold text is from our textbook. Commands to type are in orange.

We will also need to ensure our system is up-to-date and that the tools and software we will need are installed. Log in to your virtual Ubuntu server and make sure you've got all the updates you need.

```
sudo apt-get update
sudo apt-get upgrade
```

Now, install the software needed to compile the kernel, including the kernel source.

```
sudo apt-get build-dep linux
sudo apt-get install kernel-package fakeroot
sudo apt-get source linux-image-$(uname –r)
```

You may also want to install a different text editor. Vi and nano are already installed. To install emacs:

```
sudo apt-get install emacs
```

Since we may need to select between the originally installed kernel and our custom kernel at boot time, we need to modify the grub (bootloader) settings in /etc/default/grub to display a menu of kernel choices at boot.

You will likely need to start the editor with the ‘sudo’ command because of file permissions on the file. Update the following two settings at the top of the file as shown:

```
sudo nano /etc/default/grub
GRUB_HIDDEN_TIMEOUT=
GRUB_HIDDEN_TIMEOUT_QUIET=false
GRUB_TIMEOUT=5
```

Change directory into the source

```
cd linux-lts-<version>
```

The .config file is used to store information on the options to be included into the kernel. We'll copy the one from the default kernel you booted with, since we know it is configured for the devices we need.

```
sudo cp -vi /boot/config-`uname -r` .config
```

Set CONCURRENCY_LEVEL to the number of cpus/cores + 1 to take use of dual+ core

```
export CONCURRENCY_LEVEL=3
```

Part 1: Getting Started

A user-mode procedure call is performed by passing arguments to the called procedure either on the stack or through registers, saving the current state and the value of the program counter, and jumping to the beginning of the code corresponding to the called procedure. The process continues to have the same privileges as before.

System calls appear as procedure calls to user programs but result in a change in execution context and privileges. In Linux on the Intel 386 architecture, a system call is accomplished by storing the system-call number into the EAX register, storing arguments to the system call in other hardware registers, and executing a trap instruction (which is the INT 0x80 assembly instruction). After the trap is executed, the system-call number is used to index into a table of code pointers to obtain the starting address for the handler code implementing the system call. The process then jumps to this address and the privileges of the process are switched from user to kernel mode. With the expanded privileges, the process can now execute kernel code, which may include privileged instructions that cannot be executed in user mode. The kernel code can then carry out the requested services, such as interacting with I/O devices, and can perform process management and other activities that cannot be performed in user mode.

The system call numbers for recent versions of the Linux kernel are listed in `/usr/src/linux-2.x/include/asm-i386/unistd.h`. (For instance, `__NR_close` corresponds to the system call `close()`, which is invoked for closing a file descriptor and is defined as value 6.) The list of pointers to system-call handlers is typically stored in the file `/usr/src/linux-2.x/arch/i386/kernel/entry.S` under the heading `ENTRY(sys_call_table)`. Notice that `sys_close` is stored at entry number 6 in the table to be consistent with the system-call number defined in the `unistd.h` file. (The keyword `.long` denotes that the entry will occupy the same number of bytes as data value of type long.)

The author’s references are to the old-style kernel source install in /usr/src. For security reasons, the current thinking is that it’s better to generate install packages in user accounts. For us, you should usually be able to roughly replace /usr/src/linux-2.x with *linux-lts-<version>* in your home directory.

Part 2: Building a New Kernel

Before adding a system call to the kernel, you must familiarize yourself with the task of building the binary for a kernel from its source code and booting the machine with the newly built kernel. This activity comprises the following tasks, some of which depend on the particular installation of the Linux operating system in use.

- Obtain the kernel source code for the Linux distribution. If the source code package has already been installed on your machine, the corresponding files may be available under /usr/src/linux or `/usr/src/linux-2.x` (where the suffix corresponds to the kernel version number). If the package has not yet been installed, it can be downloaded from the provider of your Linux distribution or from http://www.kernel.org.

- Learn how to configure, compile, and install the kernel binary. This will vary among the different kernel distributions, but some typical commands for building the kernel (after entering the directory where the kernel source code is stored) include:

  ```
  make xconfig
  make dep
  make bzImage
  ```

- Add a new entry to the set of bootable kernels supported by the system. The Linux operating system typically uses utilities such as `lilo` and `grub` to maintain a list of bootable kernels from which the user can choose during machine boot-up. If your system supports `lilo`, add an entry to `lilo.conf` such as:

  ```
  image=/boot/bzImage.mykernel
  label=mykernel
  root=/dev/hda5
  read-only
  ```
  where `/boot/bzImage.mykernel` is the kernel image and `mykernel` is the label associated with the new kernel during the boot-up process. You will then have the option of either booting the new kernel or booting the unmodified kernel if the newly built kernel does not function properly.

`make localmodconfig` will only compile modules that are actually being used in the system! If you get prompts, accept the default values. Using `localmodconfig` will make things considerably faster to compile. If we were making a kernel to distribute that needed to work on many different platforms, we'd use `make oldconfig`. (Interestingly, this will lower the time to maybe around 2 hours from 4+ hours!!) Lastly, prepare to compile by cleaning out possible old stuff from previous attempts.

```
sudo fakeroot make localmodconfig
sudo fakeroot make clean
sudo fakeroot make-kpkg clean
```

On our system, we can compile the kernel replacing '*-some-string-here*' with your initials... Pay close attention to places where there are and are not spaces. The shell will let you continue lines with the '\' character.

```
sudo fakeroot make-kpkg --initrd \
--append-to-version=-some-string-here \
kernel-image kernel-headers
```

If this worked, you should have files in your directory called something like:

```
linux-image-3.8.13.6-tl_3.8.13.6-tl-10.00_Custom_i386.deb
linux-headers-3.8.13.6-tl_3.8.13.6-tl-10.00.Custom_i386.deb
```

You can now install the new kernel and headers. You can install them with:

```
sudo dpkg -i *.deb
```

Now reboot and try to load your kernel!

```
sudo shutdown -r now
```

Part 3: Extending the Kernel Source

You can now experiment with adding a new file to the set of source files used for compiling the kernel. Typically, the source code is stored in the `/usr/src/linux-2.x/kernel` directory, although that location may differ in your Linux distribution. There are two options for adding the system call. The first is to add the system call to an existing source file in this directory. The second is to create a new file in the source directory and modify `/usr/src/kernel/linux-2.x/kernel/Makefile` to include the newly created file in the compilation process. The advantage of the first approach is that when you modify an existing file that is already part of the compilation process, the Makefile need not be modified.

Your kernel source files will be in the `linux-lts-<version>` subdirectory off your home directory.

Part 4: Adding a System Call to the Kernel

Now that you are familiar with the various background tasks corresponding to building and booting Linux kernels, you can begin the process of adding a new system call to the Linux kernel. In this project, the system call will have limited functionality; it will simply transition from user mode to kernel mode, print a message that is logged with the kernel messages, and transition back to user mode. We will call this the *helloworld* system call. While it has only limited functionality, it illustrates the system-call mechanism and sheds light on the interaction between user programs and the kernel.

- Create a new file called `helloworld.c` to define your system call. Include the header files `linux/linkage.h` and `linux/kernel.h`. Add the following code to this file:

```
#include <linux/linkage.h>
#include <linux/kernel.h>
asmlinkage int sys_helloworld() {
   printk(KERN_EMERG “hello world!”);

   return 1;
}
```

- Add your file `helloworld.c` to the Makefile (if you created a new file for your system call.) Save a copy of your old kernel binary image (in case there are problems with your newly created kernel). You can now build the new kernel, rename it to distinguish it from the unmodified kernel and add an entry to the loader configuration files (such as `lilo.conf`). After completing these steps, you can boot either the old kernel or the new kernel that contains your system call.
- This creates a system call with the name `sys_helloworld()`. If you choose to add this system call to an existing file in the source directory, all that is necessary is to add the `sys_helloworld()` function to the file you choose. In the code, `asmlinkage` is a remnant from the days when Linux used both C++ and C code and is used to indicate that the code is written in C. The `printk()` function is used to print messages to a kernel log file and therefore may be called only from the kernel. The kernel messages specified in the parameter to `printk()` are logged in the file `/var/log/kernel/warnings`. The function prototype for the `printk()` call is defined in `/usr/include/linux/kernel.h`.
- Define a new system call number for `__NR_helloworld` in `/usr/src/linux-2.x/include/asm-i386/unistd.h`. A user program can use this number to identify the newly added system call. Also be sure to increment the value for `__NR_syscalls`, which is stored in the same file. This constant tracks the number of system calls currently defined in the kernel.
- Add an entry `.long sys_helloworld` to the `sys_call_table` defined in the `/usr/src/linux-2.x/arch/i386/kernel/entry.S` file. As discussed earlier, the system-call number is used to index into this table to find the position of the handler code for the invoked system call.

`helloworld.c` can go into `linux-lts-<version>/arch/x86/kernel`. To update the `Makefile` in the kernel subdirectory, add `helloworld.o` to one of the lines with `obj-y`. Update `linux-lts-<version>/arch/x86/syscalls/syscall)32.tbl` by adding a new line to the end of the file:

```
351    i386   helloworld    sys_helloworld
```

Uodate `linux-lts-<version>/include/linux/syscalls.h` by adding the line

```
asmlinkage long sys_helloworld(void);
```

before the `#endif`.

Now rebuild the kernel with
```
sudo fakeroot make-kpkg --initrd --append-to-version=-some-string-here \
kernel-image kernel-headers
```
and reboot.

Part 5: Using the System Call from a User Program

When you boot with the new kernel, it will support the newly defined system call; you now simply need to invoke this system call from a user program. Ordinarily, the standard C library supports an interface for system calls defined for the Linux operating system. As your new system call is not linked into the standard C library, however, invoking your system call will require manual intervention.

As noted earlier, a system call is invoked by storing the appropriate value in a hardware register and performing a trap instruction. Unfortunately, these low-level operations cannot be performed using C language statements and instead require assembly instructions. Fortunately, Linux provides macros for instantiating wrapper functions that contain the appropriate assembly instructions. For instance, the following C program uses the `_syscall0()` macro to invoke the newly defined system call:

```
#include <linux/errno.h>
#include <sys/syscall.h>
#include <linux/unistd.h>

_syscall0(int, helloworld);

main() {
   helloworld();
}
```

- The `_syscall0` macro takes two arguments. The first specifies the type of the value returned by the system call; the second is the name of the system call. The name is used to identify the system-call number that is stored in the hardware register before the trap instruction is executed. If your system call requires arguments, then a different macro (such as `_syscall0`, where the suffix indicates the number of arguments) could be used to instantiate the assembly code required for performing the system call.
- Compile and execute the program with the newly built kernel. There should be a message “`hello world!`” in the kernel log file `/var/log/kernel/warnings` to indicate that the system call has executed. To do this, type

The _syscall0 macro has been deprecated. Before trying this program, make sure you have installed the new kernel headers and installed the new kernel image *and rebooted* into the new kernel. To invoke the system call, create a program called `try_helloworld.c` with:

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

#define __NR_helloworld    351  /* or whatever value you put in unistd_32.h above */

int helloworld() {
   return (int) syscall(__NR_helloworld);
};

main() {
   printf(“The return code from the helloworld system call is %d\n”, helloworld());
}
```

To compile the program:
gcc –o try_helloworld try_helloworld.c

To test the call, `./try_helloworld` and then `tail /var/log/kern.log` to see if “Hello World!” appears in the log. It will likely not appear immediately. As Dr. Figura says, “Wait for it!”

Part 6: What To Do If Something Goes Horribly Wrong

To start over, you should reboot back to the original kernel that doesn't have modifications, so that we can remove our custom kernel. Select the original kernel from the menu. It may be in the Previous Kernels section of the menu.

```
sudo shutdown –r now
```

To remove the custom kernel from the system,

```
sudo dpkg –r linux-image-<version>
sudo dpkg –r linux-headers-<version>
```

In our particular case right now,

```
sudo dpkg –r linux-image-3.8.13.6-tl_3.8.13.6-tl-10.00_Custom_i386.deb
```

Now try going back to Part 4.

If this doesn't work, you can delete the source directory and pick up in Part 0 with the tar command.