Compiling Linux

Last Revised: June 7, 2024

Contents

1 Introduction **1**

2 Compiling the Kernel **1**

2.1 Downloading and Unpacking 1

2.2 The C Library 3

2.3 Configuring 4

2.4 Building 7

2.5 Installing 7

2.5.1 Making `initrd` 9

2.5.2 Configuring GRUB 9

2.5.3 Installing on Floppy 10

3 User Mode Linux **11**

3.1 Compiling UML 12

3.2 UML Root File System 13

3.3 Running UML 14

4 Code Browsing Tools **15**

4.1 Cscope 15

1 Introduction

This document describes how to compile a Linux kernel for study and development. It contains specific instructions for compiling an experimental kernel and for configuring various kernel debugging and code browsing tools. This document was originally prepared to support the operating systems class at Vermont State University. Some of the information it contains targets that audience. However, much of the contents of this document would be useful to anyone interested in studying the Linux kernel.

It is possible to run an experimental kernel on the same system as is being used for development. However, this arrangement is not ideal since errors in the experimental kernel may cause corruption of the development environment. Ideally, then, one should configure two computers: a development system running a pre-built kernel and where all the programming tools execute (compilers, editors, etc.), and a system for hacking that runs the experimental kernel but is otherwise expendable.

This document describes both single machine and dual machine arrangements. In the text below the *target system* is the system where you will ultimately run the experimental kernel. The *development system* is the system where you will do your development work. The *experimental system* is the system where you will do testing. In a single machine configuration the target system and the development system are the same, and there is no experimental system. In a dual machine configuration the target system and the experimental system are the same.

Additional details about how to set up and use the dual machine configuration as two connected virtual machines are in the companion document *DevBox and HackBox* in the same location as where you found this document.

2 Compiling the Kernel

This section describes how to compile the Linux kernel. It is intended to support individuals compiling the kernel for the first time. Note that this section assumes you are using a recent (6.x) version of the kernel.

2.1 Downloading and Unpacking

Before compiling the kernel you will need to obtain a copy of its source code. Your Linux distribution should come with the source code of the kernel (it's required by the GPL). However, because the kernel source is large and because most users do not require it, the kernel source is not normally installed by

default. Furthermore, each distribution tends to specialize the kernel in various ways. To build a new kernel from a particular distribution's kernel source package, you should consult the documentation for that distribution.

If you are interested in kernel development, or if you want to always use the latest kernel, I recommend that you download the stock kernel source from `http://www.kernel.org/`. This site is the official repository for Linux kernels. The kernels there are generic in the sense that they haven't been customized for any particular Linux distribution. As a result you may need to configure the kernel (as described below) in a non-default way before it will boot your system cleanly.

If you are interested in keeping up with the absolute latest version of the kernel you can check out its source code from GitHub here `https://github.com/torvalds/linux`. This would allow you to do your development in your own fork and periodically merge upstream changes into your work. It would also allow you to share your work with a team of collaborators. However, using Git to do these things is outside the scope of this document.

This document describes the build process specifically for kernel version 6.9.3 on 64 bit Ubuntu Linux 24.04. The commands below reflect this version number and platform. If you are using a different version or building on a different platform, modify the commands as appropriate. Note that you may need to install some extra packages on your system in order to do kernel development. This document does not describe which packages are necessary nor how to install them. Consult the documentation for your distribution for more information. If you are missing required packages, some commands below may not work. That is a sign that additional packages may be necessary.

It is useful to check which version of the kernel is running on your intended target system. You can do this with a command such as:

```
$ cat /proc/version
```

At the time of this writing, the version reported on Ubuntu Linux Server 24.04 is "Linux version 6.8.0-35-generic." Note that the experimental kernel I am proposing to install (6.9.3) is somewhat, but not extremely, newer than the one running on the system already. This is a good situation. The newer kernel will likely support all the features the current system expects, while not being so new as to introduce incompatibilities. This is particularly important if you are planning to run the experimental kernel on your development system (i.e., a single-machine configuration).

The kernel source is kept in a compressed archive called a *tarball*. For version 6.9.3, the name of this archive is `linux-6.9.3.tar.xz`. The last component of the version number is a release level. As 6.9 matures, it advances through several releases.

Once you download the tarball you will need to unpack it somewhere on your development system.

```
$ unxz < linux-6.9.3.tar.xz | tar xf -
```

Unpacking the tarball will create a `linux-6.9.3` directory beneath the current directory. While doing kernel development it is probably best to unpack the kernel somewhere in your home directory. That will make it easy for you to do development under your normal user account. You will need to be root to install the kernel on your system (e.g., in a single-machine configuration), but you can configure and build it as an ordinary user.

2.2 The C Library

The application interface to the kernel is by way of the C library. In theory, when the kernel is updated, the C library needs to be rebuilt so that it can take into account any changes in kernel-specific data structures provided by the new kernel. Applications that link to the C library statically (that is *not* using the dynamic shared library) would also need to be recompiled.

In addition, some header files from the kernel source are usually also in `/usr/include` where applications that need them (for example, applications making direct system calls) can access them. Again, in theory, when the kernel is updated those header files also need to be updated as well.

A Linux distribution installs a version of the C library and kernel headers that correspond to the installed kernel. When you update your kernel using the distribution's normal update system, these things are also updated if necessary. When you manually update your kernel, however, you might have to update these things yourself.

That said, changes in the kernel that necessitate rebuilding the C library are relatively rare. If the kernel you are installing is not too different from the one you are already using, you can probably get away without these additional complications.

Some distributions use symbolic links from `/usr/include` into the kernel source tree. In that case, they will install the kernel headers under `/usr/src` for the installed kernel even if they do not install the full kernel source. In a case like this you can change the symbolic links in `/usr/include` to point at the new headers; however you should in theory also rebuild the C library and statically linked applications if you do this.

If this sounds complicated and unreliable you aren't the only person who thinks that. There have been discussions among the kernel developers about how to "fix" this situation, but at the time of this writing I'm not sure how those discussions have concluded.

2.3 Configuring

Before you compile the kernel you should *configure* it. This involves selecting which features you want enabled in your kernel and which features should be compiled as modules that can be loaded later. The configuration process creates two files: `.config` in the root of the kernel source tree and `autoconf.h` in `include/generated`.[1] The file `.config` gives the make utility access to your desired configuration. Make uses this information to control which files are compiled and how. The file `autoconf.h` is included into the various kernel source files (and also in external modules) and gives the C compiler access to your desired configuration. Programmers can use #ifdef/#endif directives in the C source to selectively compile different code depending on the configuration options chosen.

Your Linux distribution has most likely already created a kernel configuration that is compatible with the software environment of that distribution. For example the start-up scripts may depend on certain features being enabled in the kernel. Using that configuration as a starting point allows you to configure your new kernel to match the existing configuration as closely as possible resulting in a minimum of problems.

To create a configuration based on some currently running kernel, you will need to first obtain a copy of the configuration file for the that kernel. Depending on how the running kernel was configured a compressed version of that file might be in `/proc/config.gz`. My Ubuntu 24.04 distribution stores a copy of the current configuration in `/boot/config-6.8.0-35-generic` where 35 is the current (at the time of this writing) release number of the Ubuntu-flavored kernel. You can read the file `/proc/version` to find the exact version of the running kernel. That information might be useful for determining which configuration file is the most appropriate if there is more than one.

Copy the existing configuration file to the root of your new kernel source tree under the name `.config`. Then run the command `make oldconfig` to update that configuration for use with the new kernel. For example, if you are building a new kernel for your development system:

```
$ cp /boot/config-6.8.0-35-generic .config
$ make oldconfig
```

If you are building a kernel to run on an experimental system you should create a `.config` that is suitable for that system. Thus, instead of copying the existing configuration of the development system, you should, in theory, instead take it from the experimental system. In practice, if you are using the same Linux distribution on both the development and experimental system, it is reasonable

[1] The `autoconf.h` file is created as part of the build process. It does not exist until you actually compile the kernel for the first time.

to use the configuration file from the development system as the basis for your configuration.

There are two kinds of issues that will be reported by the `make oldconfig` command. First, configuration options in the existing kernel that are not in the new kernel (because they have been removed or renamed, or because the existing kernel has been extended by the distribution) produce warnings when they can't be mapped into the new configuration.

Second, configuration options in the new kernel that are not in the existing kernel (because they are new) will prompt you to choose between 'Y' (meaning compile the feature into the kernel), 'N' (meaning don't compile the feature), or in some cases 'M' (meaning compile the feature as a module). You can just hit 'enter' to accept the default in most cases.[2]

Next, you may need to make a manual adjustment to the `.config` file with respect to the list of trusted keys known to the kernel. Linux has a facility that allows modules to be digitally signed with signatures that are checked by the kernel when each module is loaded. During the kernel build process, a public/private key pair is automatically generated by the kernel for this purpose (although you can provide your own if desired). Internally, the kernel keeps a list of public keys known to it on a "key ring."

In addition to the generated (or provided) public key mentioned above, it is also possible to provide a list of other public keys as X.509 certificates in a PEM-formatted file. These keys are built into the kernel image during the build process. For our purposes, we don't need to provide such a file, but you'll need to adjust the configuration to say so. Search the `.config` file for the following configuration parameters:

```
CONFIG_SYSTEM_TRUSTED_KEYS CONFIG_SYSTEM_REVOCATION_KEYS
```

Set their values to the empty string (there may be non-existent files mentioned by default). The values for these parameters are leftovers from the kernel configuration you copied and thus reflect values being used by your existing kernel's distributor.

After the initial configuration process is complete you can use:

```
$ make menuconfig
```

OR

```
$ make gconfig
```

[2] You can also select '?' to get help information about an option.

to further refine the configuration. You can also use these commands to change the configuration later if desired. Note that `make menuconfig` requires that you have the curses terminal handling library available and `make gconfig` requires that you be running a graphical desktop with the appropriate GTK+ graphical libraries available. *TODO: Is `make gconfig` still even supported? It seems to require an old version of the GTK+ libraries.*

Warning! If you modify the configuration with either of the commands above you will need to do a full kernel rebuild. This takes a long time so if you aren't prepared to do that be careful not to accidentally change anything when just reading the configuration.

If you are building a kernel for experimentation purposes you may want to enable some debugging features in the kernel configuration. I invite you to explore the options under the "Kernel hacking" heading. Even if you decide to not activate any debugging features at this time it would be good for you to be aware of the possibilities in case you want to try them later.

Note that debugging features, and the checks they imply, will impact the performance of your kernel in terms of both space and time. This is why many of them are off by default. In fact, some checks are so expensive that I do not recommend using them in a kernel built for general use (although they may be acceptable on the experimental system). Consult the help information on each option for more information.

If you wish to debug your kernel using a source level debugger you will want a kernel debugging option turned on. To find this option, look under "Kernel hacking", then under "Compile-time checks and compiler options", and finally under "Debug information". Full debugging information is the default in the Ubuntu 24.04 configuration. However, *be aware that this option greatly increases the amount of disk space required to build the kernel* since every object file produced by the compiler contains debugging symbols.[3]

This option is appropriate if you are creating a User Mode Linux kernel (see Section 3) or if you plan to use a kernel debugger (such as KGDB) or a tool to analyze kernel crash dumps. If you do not plan to use a source level debugger you can save a lot of disk space by setting the "Debug information" option to "Disable".

If you are building a kernel for use on an experimental system, and you wish to debug it remotely using KGDB, you will want to enable KGDB support in the kernel configuration in the "Generic Kernel Debugging Instruments" submenu beneath "Kernel hacking" (this is also the default in the Ubuntu 24.04 configuration). I say more about setting up and using KGDB in the companion document *DevBox and HackBox*.

[3] Several gigabytes of disk space are required for a full build using debugging.

2.4 Building

To actually build the kernel and all of its supporting modules do:

```
$ make
```

A kernel build takes a long time. There is a lot of code. Note that this will compile most drivers (as modules) even though your system will likely never use them. Nevertheless, you may find some drivers useful, especially in an experimental context, so it doesn't hurt to build them all.

The build process may require various libraries that you do not have installed on your system initially. If the build fails, look at the reason and then try to install the necessary package(s) to satisfy any missing requirements. Run the `make` command again to restart the build. It will pick up where it left off. You may need to do this several times. However, once you have all the necessary libraries installed, future builds should go more smoothly.

2.5 Installing

You do not need to be root to configure and compile the kernel. If you unpack the kernel source in an area where you have read/write permission, you should be able to build it as an ordinary user. However, you do need to be root to copy the new kernel to a place where it can be used to boot a machine.

Once the kernel has been built you should first copy the various compiled modules to the proper directory under `/lib/modules` so that the running kernel can find them. This is accomplished by doing:

```
# make INSTALL_MOD_STRIP=1 modules_install
```

The INSTALL_MOD_STRIP option removes debugging information from the modules as they are installed. This greatly reduces the amount of disk space used by the module library and by the initial RAM disk (described below). In low memory systems saving memory can be essential since the full module library, with debugging information included, is very large. You may be motivated to configure your experimental system with a minimal amount of memory since it won't be used for any "real" work. As a result, without stripping the modules, it is likely that the initial RAM disk will be too large for a minimally configured system to use.

One disadvantage of stripping debugging information from the modules is, obviously, you won't be able to step into those modules or set break points in them when debugging the kernel. This might be an issue if you are trying to use the

debugger to study the operation of the kernel. However, if you are mostly interested in debugging your own kernel modifications or kernel modules, removing debugging information from the distributed modules will probably not cause you any issues.

Another aspect of stripping debugging information is the effect it has on module signatures. Normally, the debugging information is covered by the signature. Thus, the signature is invalidated or removed (*TODO: Which is it?*) if the debugging information is stripped afterward. However, the `make` command above strips debugging information first, before making the signatures, so you end up with signed, stripped modules as desired. *TODO: The `file` command uses the phrase "with debug_info" as opposed to "stripped." In fact, it seems to use "stripped" for something else. The terminology here should be made consistent.*

You should install modules even on your development system because the module library of a kernel is used during the compilation of external modules for that kernel. However, it is safe to install modules for a kernel even if you usually run a different kernel. Each kernel has its own private directory under `/lib/modules`.

If plan to run your new kernel on an experimental system (called "hackbox" in the commands below) you should also copy the modules to that system where they can be used. On the development system, after installing modules, do the following:

```
# cd /lib/modules
# tar cf - 6.9.3 | gzip > modules-6.9.3.tar.gz
# scp modules-6.9.3.tar.gz hackbox:/lib/modules
```

Unpack it on the experimental system using:

```
# cd /lib/modules
# gunzip < modules-6.9.3.tar.gz | tar xf -
```

You should next copy and rename three files from your freshly built kernel to the `/boot` directory on the experimental system. For example, you might do the following:

```
# cd /home/student/linux-6.9.3
# scp arch/x86/boot/bzImage hackbox:/boot/vmlinuz-6.9.3
# scp .config hackbox:/boot/config-6.9.3
# scp System.map hackbox:/boot/System.map-6.9.3
```

The `System.map` file contains a list of all symbols in the kernel and their corresponding addresses. This can be useful for debugging and for interpreting stack

traces in kernel oops messages. *TODO: Why is it important for this file to be in* `/boot`*?*

Finally, I recommend using `ls -l` in the `/boot` directory of the experimental system to check file ownership and permissions. Use the `chown` and `chmod` commands as appropriate to match the owners and permissions on the new files to those of the existing files. While this is not an essential step, it gives the installation a clean look, and it may have importance from a security and system maintenance point of view.

2.5.1 Making `initrd`

Because modern Linux systems are so highly modularized it is possible that the kernel will need to load a module in order to read the file system. This creates a problem: how can it load a file system support module from the file system? To get around this, Linux boots in two phases. During the first phase, the bootloader loads a pre-defined RAM disk image into memory along with the kernel. The kernel uses this RAM disk image as it's initial root file system. Certain programs and kernel modules can be loaded out of this RAM disk image. Once that is done, the root file system is changed to be the normal hard disk and the usual start-up scripts are executed.

Manually creating this initial RAM disk is a somewhat involved procedure. Fortunately there is utility program named `mkinitramfs` that does most of the work. On a Ubuntu system the command is:

```
# mkinitramfs -o /boot/initrd.img-6.9.3 6.9.3
```

This creates a RAM disk using the same modules as in the existing configuration, except that it will use the modules for the right kernel version. If you attempt to use the old RAM disk, it will contain modules for the old kernel which won't load into the new kernel.

You should run this command on the experimental system where you plan to run the new kernel. Be sure you have the module library installed on that system, and be sure the new kernel configuration file is also installed in `/boot`. The `mkinitramfs` command makes use of both of those resources.

2.5.2 Configuring GRUB

Once you have `vmlinuz-6.9.3` and `initrd.img-6.9.3` in the `/boot` directory of your target system you only need to update your bootloader to provide an option to boot the new kernel. This can be done by cloning the information for the existing kernel to a new menu entry and modify the names of the kernel

image file and RAM disk file. The precise steps for doing this will depend on the bootloader you use.

On a Ubuntu system this is easily accomplished by using the `update-grub` command. This command searches `/boot` for kernels installed there and composes a suitable GRUB menu for them.

The next time you boot your system if you press the left-hand shift key early in the boot process you will see the GRUB boot menu. From there you can select your new kernel. *TODO: Say more about setting up GRUB options.* However, if the experimental kernel is the newest kernel on the system, it will be booted by default.

2.5.3 Installing on Floppy

This section is very old and needs to be rewritten (or removed?). For one thing it needs to explain how to handle the initial RAM disk. For a second thing it should probably really talk about setting up a flash drive instead of a floppy (who has floppy drives?).

The following instructions pertain to users who are booting Linux from a floppy disk. Note that this is not the normal configuration (although it is sometimes useful in lab situations).

1. Make a copy of your boot floppy. Never overwrite a known working kernel with one that you just compiled! First make a backup of the working kernel and be sure that you can boot the working kernel if necessary.

   On Windows you can back up your boot floppy with the `diskcopy` command. Open a Windows command prompt and do:

   ```
   C:\> diskcopy a:  a:
   ```

   You will be prompted for the source disk and then for the target disk. Note that you can't just copy the files from one disk to another! A boot floppy contains special information in the boot sector that will not be copied by the usual file copying operations.

   On Linux you can use the `dd` command to copy disks raw. Insert the source floppy and do:

   ```
   # dd if=/dev/fd0 of=/tmp/floppy.img bs=1024
   ```

   Then insert the target floppy and do:

   ```
   # dd if=/tmp/floppy.img of=/dev/fd0 bs=1024
   ```

   You can remove the temporary file afterward if you wish. See the manual page for the `dd` command for more information.

2. Next insert the boot floppy where you want the new kernel to go and mount it. This can be done with a command such as:

   ```
   # mount /dev/fd0 /mnt/floppy
   ```

   Use whatever directory is most appropriate if `/mnt/floppy` is not available (there should be a `/mnt` directory at least).

3. Copy `arch/x86/boot/bzImage` to `/mnt/floppy`, renaming it to `vmlinuz` in the process. This will overwrite the `vmlinuz` on the floppy with the new kernel.

   There are some control files on the floppy as well that you could edit. However, if you use the same name (and you might as well since the floppy isn't big enough to hold both kernel images) the control files should already be configured properly.

4. *Very Important!* Unmount the floppy before physically removing it. This is necessary because Linux keeps disk blocks in its cache even for floppy disks. This means that the entire file isn't actually put on the floppy until you unmount it.

   ```
   # umount /mnt/floppy
   ```

Now you can reboot the machine from your new boot floppy to check your new kernel.

3 User Mode Linux

This section is old and needs to be reviewed and updated. It has been a while since I have built a User Mode Linux kernel.

Setting up a completely independent experimental system is a nice way to do kernel development. However, there are times when it may not be desirable. For example, you might want to do kernel development on a machine that you depend on for your normal work and yet not want to risk running an experimental kernel directly on that system. One option is to use virtualization software to create a virtual machine for a separate experimental system (as described in *DevBox and HackBox*). However, if your primary machine is also running Linux, another approach is using User Mode Linux (UML).

The Linux kernel is cross-platform and with suitable cross-compilers can be compiled on one platform for execution on another. User Mode Linux is treated as a special "platform." However, the UML kernel runs on top of a host Linux system as an ordinary process. All access to hardware is translated into system calls made against the host system. UML thus allows you to run a custom Linux

kernel alongside your regular applications. You don't even need to be root on the host system.

Another advantage to UML from a kernel development point of view is that it allows you to debug the kernel using an ordinary source level debugger such as `gdb` without the complexities of setting up a separate machine and remote debugging. I will discuss how to do this in more detail later in this document.

3.1 Compiling UML

The procedure for compiling the User Mode Linux kernel itself is simple. In what follows I will assume you are using a 4.x kernel. The 4.x kernel comes with UML as one of the officially supported architectures. First, unpack the kernel source code to a suitable working directory. Next run the command:

```
$ make defconfig ARCH=um
```

It is important to use the default configuration generated by `defconfig` as the starting point for your kernel configuration. *Do not try to use the configuration of the running kernel.* It is also important to add the `ARCH=um` option to the command line. This tells the build system that you are cross compiling to a different architecture.

Next run either:

```
$ make menuconfig ARCH=um
```

OR

```
$ make gconfig ARCH=um
```

It is important to consistently use the architecture specifier.

Under "UML Specific Options" be sure that "Host filesystem" is selected. This will allow the UML kernel to access the file system of the host; an easy way to share files between the host and a running UML system. Under the "Kernel hacking" option (on the top level menu) be sure the "Compile the kernel with debug info" and the "Compile the kernel with frame pointers" options are both selected. These options make it possible to properly debug the UML kernel with `gdb`. You may or may not want to set some other debugging related options. Save these changes.

You are now ready to build the kernel using the command:

```
$ make ARCH=um
```

When the build is complete you will have an ordinary executable file named `linux` in the root directory of the source tree. Before you can use it, you will need a root file system.

3.2 UML Root File System

User Mode Linux reads its root file system out of a file in the host's file system. This file must be set up so that it contains all the normal programs and tools Linux needs to boot. You may also want development tools or other programs inside your UML's root file system. Although you may be able to download a root file system, matching the root file system with the precise kernel version and options you want to use can be tricky. The reason for this is that during the boot process, typical Linux configurations read various modules out of the root file system to enable support for features needed by the startup scripts. These modules must be compatible with the running kernel. If you are using a kernel version that is not compatible, the boot process is likely to fail, or at the very least report many errors.

Sometimes you can download a UML kernel along with a matching root file system. However, as a kernel developer, the kernel you want to use is not arbitrary; it is a specially configured kernel of your choosing. As a consequence of this, the ideal path is to build your own custom root file system for kernel development. The procedure is as follows.

TODO: FINISH ME! The description below is very incomplete.

1. Create a file to hold the root file system. This file may need to be rather large, depending on how much material you plan to install into the UML environment. Use a command such as:

   ```
   $ dd if=/dev/zero of=root_fs bs=1M count=512
   ```

   This command creates a file named `root_fs` with a size of 512 megabytes. This file is initially all zeros. It will be treated as raw disk image.

2. The image created above must then be formatted with a suitable file system.

   ```
   $ mk2efs -j root_fs
   ```

   This command creates an ext3 file system (which is the same as an ext2 file system with a journal created by the `-j` option) inside the `root_fs` file. You may need to specify the path to the `mk2efs` program; it is typically not in the path of ordinary users. The use of ext3, or some other journaled file system, is recommended. Since this environment will be used for kernel development, kernel crashes are likely, and it is nice to have the extra safety inherent in using a journaled file system.

3. Mount the root file system so that you can access its contents. To do this you will need to be the root user. First create a suitable mount point. I suggest an empty directory named `root`. Then issue:

   ```
   $ mount -o loop root_fs root
   ```

This command uses the loop back driver (which must be supported in your host kernel) to mount the file system contained in `root_fs` onto the mount point `root`.

4. You will now find an empty partition beneath the mount point, ready for you to set up your root file system. Configuring a root file system for use with Linux is a non-trivial exercise. Lack of space in this document prevents me from going into the details here. Please refer to other documentation for more information.

5. Once the root file system is ready you should (as the root user) unmount it before you try to use it with UML.

   ```
   $ umount root
   ```

   You might want to make a backup copy, perhaps in compressed form, of your fresh root file system in case you destroy your working copy while setting up UML or doing kernel development.

3.3 Running UML

Make sure the root file system is named `root_fs`. Execute `linux` to boot User Mode Linux.[4] Log in as the user root. Typically, depending on the root file system you are using, the password will either be blank or also root.

Once you have logged in you can use the command:

```
# mount none /mnt -t hostfs
```

to mount the root of the host file system onto the `/mnt` directory inside the UML environment. This allows you to copy files to and from the host file system. You may now use User Mode Linux in a manner very much the same as any other Linux system.

TODO: Talk about setting up disk partitions under UML.

To debug the running UML process, open another window on the host machine. Use a command such as:

```
$ ps aux | grep linux
```

to search for information about the running UML process. You will find several entries because UML is a multithreaded application. Note the process ID of the first entry. Then launch the `gdb` debugger, attaching `gdb` to the process of interest. For example:

[4] You can name the root file system something else, in which case you need to add the `ubd0=` boot option to the command line to specify it.

```
$ gdb linux 1234
```

where 1234 is the process ID of the running UML system.

Once `gdb` has started you will want to execute the command `handle SIGUSR1 nostop noprint`. I'm not entirely sure what this command does exactly, but it has something to do with the way `gdb` deals with multiple threads. In particular, without this command `gdb` will stop the UML system frequently because of SIGUSR1 signals.

Notice that when `gdb` attaches to a process, that process is stopped. Your UML session will appear dead. However, by issuing the `continue` command to `gdb` you can cause your UML session to resume normally. Use $\hat{\text{C}}$ in the `gdb` window to interrupt the UML session at any time. You can now set break points and single step the Linux kernel as you might any other process.

TODO: Talk about how to set a break point inside a module.

4 Code Browsing Tools

The Linux kernel is very large and finding one's way around in it can be a major chore. To simplify the navigation of large programs there exists a number of code browsing tools. I recommend using one or more of these tools when working with the Linux kernel source. In this section I talk about how to configure a few of these tools for use with the Linux kernel.

Note that you should only set up code browsing tools *after* you've applied any patches to the source code. Patches will, of course, modify files and change the line numbers where functions are defined, etc. If you index the source and then apply patches or make other changes you will want to reindex the source afterward. However, it shouldn't matter if you've compiled the kernel first or not. Code browsing tools are normally smart enough to ignore object files.

4.1 Cscope

The `cscope` tool is a simple but effective code browser with a long history. It reads a collection of C files and builds an indexed database that can be used to quickly look up declarations and points-of-use for any symbol.

The script below launches `cscope` on the Linux kernel. You should edit the three variable definitions at the top of the script to suite your system. The script does not index the entire kernel code base. In particular, it skips the driver hierarchy and only indexes the x86 architecturally specific code. This makes the database size more manageable and reduces the number of duplicate declarations the tool returns.

```
#!/bin/bash

# Set a few variables. Change here for your system.
CSCOPE_DIR=/home/student/cscope
CSCOPE_FILE=$CSCOPE_DIR/cscope.files
LNX=/home/student/linux-6.9.3

# If the database hasn't yet been created, then create it.
if [ ! -f $CSCOPE_FILE ]; then

  # Build file list. Exclude uninteresting regions.
  cd ~
  find $LNX \
    -path "$LNX/arch/*" ! -path "$LNX/arch/x86*" -prune -o \
    -path "$LNX/Documentation*" -prune -o \
    -path "$LNX/scripts*" -prune -o \
    -path "$LNX/tools*" -prune -o \
    -path "$LNX/drivers*" -prune -o \
    -name "*.[chxsS]" -print > $CSCOPE_FILE

  echo Creating database...
  cd $CSCOPE_DIR
  cscope -b -q -k

  echo Creating tags...
  ctags -L $CSCOPE_FILE
  if [ -f tags ]; then
    mv tags "$LNX"
  fi

fi

# Run cscope
cd $CSCOPE_DIR
cscope -d
```

In addition to using a dedicated code browsing tool, many programmer's text editors have a feature that allows them to read a "tags" file containing cross-reference information about entities declared and defined in a program. The script above uses the `ctags` command to create such a file for the Vim editor.

It is natural to create a tags file for your editor at the same time as you create the `cscope` database. This is because `cscope` will launch your editor (Vim is the default) whenever you ask it to display a file. Once in the editor it is convenient to continue your browsing experience using editor tags commands.

If you are an `emacs` user instead, you can create a tags file for that editor

with the `etags` command. Set the EDITOR environment variable to `emacs` to override `cscope`'s default editor setting.

TODO: FINISH ME! Need to talk about how to use `cscope` *and the tags file.*