UG3 Operating Systems 2014-15, Practical Excercise

1 Introduction

This practical exercise starts in week 3, and runs until week 10, with a final submission deadline of **16:00 Friday 21 November 2014**. The submission mechanism will be electronic, by the "submit" command. No other form of submission will be accepted. The practical instructions will be issued in three phases.

1.1 Phase 1: Background

In the first phase, you will be asked to do some background reading. You will be given instructions on using the VirtualBox emulator, and expected to become familiar with it.

You will take a first look at the Linux kernel source, and at the kernel build procedure.

During this phase, you should work on revising and extending your knowledge of C. The key concepts that differ from Java are structs and pointers. You should also aim to get a general understanding of Makefiles and the kernel build procedure, although no detailed knowledge of this will be needed.

The main objectives and intended outcomes for phase 1 are:

- [course objective] understanding of the basic structure of the Linux kernel, and how that structure is reflected in the structure of the source code
- [course objective] understanding of the virtual machine concept as provided by VirtualBox
- [transferable skills] basic C skills
- [transferable skills] use of online resources for system programming

Time: about 2 weeks, continuing into phase 2.

1.2 Phase 2: Simple exercise and more reading

In this phase, you will be be given instructions on how to build a kernel module that can be loaded into a running kernel. You should study appropriate documentation (key references will be given) to ensure that you understand the supplied module template, and you will need to ensure that you can successfully compile a module, transfer it to your VirtualBox system, and load it into the running kernel. You will continue your acquisition of basic C knowledge, and you will be asked to look at one component of the Linux kernel source in more detail.

Main objectives and intended outcomes:

- [course objective] understanding of the principle of modular kernel design, as implemented in Linux
- [course objective] deeper understanding of one component of the Linux kernel and its relation to the general presentation in lectures.
- [course objective] understanding of a shell and its basic implementation structure
- [transferable skills] continuing basic C skills

Time: about 1 week.

1.3 Phase 3: Implementation and experimentation

In the final phase, which will produce the assessed result of the exercise, you will be asked to write a module to perform a specific function (possibly purely information gathering) connected to the kernel component studied in phase 2. You will submit your final module code; it will be automatically tested, and manually reviewed for style.

You should strictly follow the guidelines below:

- **Your code must compile in the specified environment.**
- **If your code contains any hidden dependencies on another system, and therefore fails to compile, it will receive a fail mark.**

Main objectives:

- to demonstrate your successful attainment of the Phase 1 and 2 objectives
- to reinforce your understanding of operating system implementation

Time: about 1.5 - 2 weeks.

1.4 Permissible collaboration

The primary purpose of this practical is educational. However, it also contributes to your mark for this course. Therefore the following policy on collaboration applies:

During Phases 1 and 2, you are permitted, and actively encouraged, to discuss any aspect of the exercise, whether in person with each other, or via the course newsgroup.

During Phase 3, you should be working mainly on your own. That means that the module code you submit must be your own code, produced without significant assistance from others. The following forms of collaboration and discussion are permitted:

- asking for assistance (from each other or the newsgroup) with any problems you have with the C language or the compilation process;
- asking a colleague to help you debug your code by, for example, listening to you talk through it and trying to spot mistakes;
- asking for assistance using any debugging facilities provided by the system.

If you have any question or difficulty that does not obviously fall into the above categories, you should ask it publicly on the newsgroup, and I will determine a suitable public answer.

1.5 Using other machines

The practical material will be provided to you in the DICE file system, and the instructions will assume that you are working on a DICE workstation.

You are welcome to do the practical on a home machine. However, if you do this, you must be aware that:

- I will not provide any support or assistance in transferring the practical to another system
- you will need to copy several hundred megabytes of files to your home machine
- your final submitted code must compile and run, unmodified, in the specified environment.

2 Phase 1: Background

2.1 A short introduction to the Linux kernel

First, you should find some general descriptions of the Linux kernel. Many such descriptions may go into more detail than we have so far covered in the course; however, we have reviewed the high-level structure of OSes, which should suffice. At this stage, you can ignore more detailed descriptions of virtual memory structures, etc., although you are of course free to read up on this in advance of lectures.

Warning: Linux is a rapidly evolving system. In this exercise, you will be using kernel 2.6.21.1. This is older than the kernel now running on DICE; we're using an older kernel because some of the things we rely on have been locked down in new kernels. (3.16 is the current production series; kernel 2.6.21.1 is slightly outdated, but can still meet the main purpose of this practical)

Unfortunately, there is a shortage of good documentation about the 2.6 kernels. However, the basics have not undergone any really dramatic changes, although many of the details have. Most high-level information and quite a lot of detailed information about 2.4 or even 2.2 kernels can be (cautiously) applied to the 2.4 series. Information about earlier releases should be carefully checked before you assume it applies to modern kernels.

Suitable descriptions can be found OS textbooks that take Linux as a case study. Silberschatz has a chapter on Linux, of about the right level for this phase. Stallings does not have a single chapter on Linux, but several chapters contain sections on aspects of Linux: specifically sections 2.7, 4.6, 8.4, 10.3. On the Stallings book website, there is a PDF file collecting together all the Linux information.

There are also many online resources, which are themselves sufficient. The centre for Linux documentation is **The Linux Documentation Project** (http://www.tldp.org/).

You should look round this site, and consider particularly the following items in the "Guides" section:

- **The Linux Users' Guide.** This is a basic introduction to end-user use of Linux, covering such things as the use of the shell, etc. If you are not confident in traditional (i.e. command-line, not GUI) use of Linux, this guide may be useful. (Real system programmers don't use GUIs, of course.)
- **The Linux Kernel.** Although this is very out-dated, it is probably the single most useful online resource (apart from the kernel source itself!). It provides a good description of the 2.0 kernel, and does not assume a great deal of pre-existing technical OS knowledge. You should read this document.
- **The Linux Kernel Hackers' Guide.** This is more outdated, referring to the 1.0 and 1.1 releases. However, it may be useful in conjunction with other documents.
- **Linux Kernel 2.4 Internals.** This is the other main online resource. It provides quite detailed information on the implementation of the current kernel. It is written for a fairly expert audience, in a concise and unforgiving style. Note that the scheduler (which is the part we are most likely to look at) has been completely re-written between 2.4 and 2.6.
- **The Linux Kernel Module Programming Guide.** Since you're going to be writing a kernel module, it's well worth looking through this.

And from outside:

- **Understanding the 2.6.8.1 Linux Scheduler** (http://joshaas.net/linux/)**.** This is of course a bit out of date, but I don't think there are really serious scheduler changes between 2.6.8 and 2.6.21.

There are also several relevant HOWTOs. In particular:

- **The KernelAnalysis-HOWTO.** This is a summary of the 2.4 kernel structure. Read it after reading the more leisurely "The Linux Kernel".

- **The Kernel-HOWTO.** This describes the process of building the kernel. You will not be required to build a kernel, but it will do no harm to get an idea of what is involved.

2.2 (Re-)Learning (rudimentary) C

There are many course notes available on the Web designed to introduce Java programmers to C.

One such page is http://www.comp.lancs.ac.uk/computing/users/ss/java2c/ which gives a fairly detailed view of the differences between C and Java.

There are also many other such pages, and indeed textbooks – use Google, or look at those linked from the Stallings book page.

Most of the basic control constructs are very similar to Java (Java was designed that way). In understanding kernel source, there are some crucial features of C that do not occur in Java, and that have to be understood.

2.2.1 The C pre-processor

When a C program module (usually stored in a *.c file, e.g. sched.c) is compiled, there is a "pre-processing" phase before the actual compilation. This serves several purposes: it allows other physical files to be included in the current file before it is compiled. Such files are usually "header files" (with a .h suffix), which contain declarations of external functions and variables that this program will use. Thus, a header file performs something of the function of a Java interface, although it is cruder. For example, almost all C user programs (but not kernel source!) contain the line

```
#include <stdlib.h>
```

which includes the declarations for the functions in the standard C runtime library.

It allows so-called "macros" to be defined. For example, if your program contains the line

```
#define FOOBAR "derived from a vulgar acronym"
```

then before the program is compiled, any occurrence of the word FOOBAR (except inside strings and comments) will be replaced by "derived from a vulgar acronym". Macros are sometimes used to define constants that are considered somehow fundamental; or constants that may vary from one architecture to another! The latter use relies on the third point:

It allows "conditional compilation": that is, the compiler can be given different code, depending on the setting of macro-defined constants. For example, if your program contains the lines:

```
#ifdef LINUX
  ... some code for linux
#else
  ... some code for something else
#endif
```

then the compiler will see the linux code only if earlier in the file the macro symbol LINUX is defined.

2.2.2 Structs and pointers

C does not have objects. The way to build complex data structures in C is via structs and pointers. A struct is a bit like an object: it is a bunch of data gathered together into a single record. The data can be normal data, like Java instance variables, or functions.

Example: the declaration

```
struct some_stuff_struct {
    int a;
    int b;
    char s[24];
};
```

declares a type called "struct some_stuff_struct", which contains two integers and a 24-character string (in C, a string and an array of characters are (almost) the same – not like Java!). The components can be accessed using dot notation, rather as in Java:

```
struct some_stuff_struct mystuff; // declare variable mystuff to have
                                  // type  struct some_stuff_struct
mystuff.a = 42;                   // set "a" component to 42
mystuff.s[10] = 'T' ;             // set character 11 of the string
                                  // to 'T'
```

It gets boring typing "struct" all the time, so it is common to define a short name like this:

```
typedef struct some_stuff_struct some_stuff_t;
```

This makes "some_stuff" mean the same as "struct some_stuff_struct". In fact, this is often done at the time of the original declaration of the struct type, so you will see, instead of the first declaration above:

```
typedef struct some_stuff_struct {
    int a;
    int b;
    char s[24];
} some_stuff_t; // simultaneously declare the struct and its short name
```

Pointers are like object references in Java, except that a C pointer can refer to anything, not just a struct. Pointers are often used with structs to achieve the same sort of effect as Java objects.

If you have a piece of data, you can get a pointer to it by prefixing an ampersand &. If you have a pointer, you can get the thing it points to by prefixing a star *. Owing to the highly confusing way in which C types are formed, in order to declare a pointer variable, you have to use a star...

Here is an example:

```
some_stuff_t mystuff;       // declare mystuff as above
some_stuff_t *stuffptr;     // stuffptr is a variable whose type is
                            // "pointer to some_stuff_t".
mystuff.a = 42;             // set "a" component of mystuff to 42
stuffptr = &mystuff;        // stuffptr now points to mystuff
if ( (*stuffptr).a == 42 ) {
    /* this is true */
} else {
    /* this is false */
}
```

Now we can pass stuffptr around in function calls, and the functions will get a pointer to mystuff – so they can change it.

Because one is usually passing around pointers to structs, and then using them with things like (*stuffptr).a, C provides a short form for that: we can write stuffptr−>a (mnemonic: the a field of the thing that stuffptr points to).

So really, stuffptr−>a is the equivalent of the Java notation obj.a , and the C notation mystuff.a doesn't have an equivalent in Java, because in Java you can only ever get a reference to an object, not the object itself.

The other use of pointers is to allow a function to modify an argument passed to it. Of course, that can never really happen, so what one does is pass a pointer to the argument; then the function dereferences the pointer, and gets the original argument. For example, consider the following code fragment:

```
void inc(int x) {
    x++;
}

int a = 42;       // integer
inc(a);           // call the inc routine, passing a

/* a  still has the value 42 */
```

This works exactly as in Java, and so the value of a is unchanged by passing it to the function inc, which just increments its parameter variable x.

However, in C we can write:

```
void incp(int *xptr) {
    (*xptr)++;   // increment the integer pointed to by  xptr
}

int a = 42;
incp(&a);        // call incp, passing it a pointer to  a.

/* a  now has the value 43 */
```

This is rather like defining a class containing a single integer field, but without the trouble of actually doing so.

Putting it together, here is how one defines a simple linked list of integers in C:

```
// the following makes list_pointer mean ``pointer to struct list_record''
typedef struct list_record *list_pointer; // pre-declare the pointer type

// the following actually declares the struct itself
struct list_record {
    int value;  // the integer in this element of the list
    list_pointer next;  // pointer to the next element in the list
};
```

In C, the null pointer is just (the integer) 0. However, there is often a predefined macro constant NULL, so that you write NULL whenever you mean a null pointer.

So far, I haven't talked about you actually create a new structure (the equivalent of the Java new operation). This is, unfortunately, messy. With luck, you won't have to do it.

2.3 Running a virtual Linux system

For rather obvious reasons, we can't let you loose in the kernel of one of our machines. However, nowadays there are available excellent emulators which allow you to run an entire modern system inside an emulated PC - and modern PCs are so fast that this emulation is itself capable of running applications (never mind OS exercises) at a perfectly reasonable speed. Therefore you will do your kernel programming inside such a virtual machine.

The emulator we'll use is Oracle's VirtualBox. You **could** run it at home, but on the whole I recommend doing the practical on DICE.

Running VirtualBox on DICE machines

NOTE: these instructions work ONLY on DICE machines, and moreover you must have an X display set. That is, you must either be on a DICE workstation, or you must be logged from via ssh with X display forwarding enabled. (Or you must have explicitly set an appropriate DISPLAY.) Do NOT run VirtualBox on any of the School's server machines.

A former Teaching Assistant Zheng Wang has made it very easy for you to set up VirtualBox for this practical. Here is what to do:

Bring up an xterm (or whatever other shell window you use – doing this inside an Emacs shell buffer is possible, but advisable only for hardcore Emacsers). Make a directory for this practical, and cd into it.
Then issue the command:

```
/group/teaching/cs3/os/VBox/setup.sh
```

and follow the instructions it gives you.

After this, you will be able to start your virtual machine just by starting VirtualBox.

When you start VirtualBox, the application GUI will appear; when you start the virtual machine just created, a console window (which is the console tty of the VM) will appear. You will see a normal Linux boot sequence, and then you should see a login prompt.

Now log in (as "root" if you're feeling confident, or "user" if you're feeling nervous) and have a look round the virtual machine.

Shutting down: to shutdown your virtual machine, log in to the console as root, and do

```
shutdown -h now
```

Note that an unclean shutdown will have the same mangling effect on your virtual disk that an unclean shutdown of a real machine has on its real disks, so you should always follow this procedure.

The VM has two virtual disks attached. One is mounted on /, and contains the system - it is a stripped down Fedora system. NOTE that you cannot make any permanent changes to files - any changes you make will be lost when you shut down the VM. To provide VM-internal storage, a second disk is mounted on /work, and changes in here are persistent. Note that this second virtual disk is in reality a file in your filespace, so consumes part of your quota - so don't copy any BIG files into /work.

When the VM was set up for you, a directory called "shared" was created in your practical directory. You can mount this in the VM by (as root):

```
mount -t vboxsf shared /mnt/shared
```

Finally, you may (indeed, will!) get to a state where your console is wedged with a command not responding. It is not possible (currently) to ssh in to your VM from outside; however, like all standard Linux systems, the system has six virtual (i.e. virtually virtual!) consoles available on the (virtually real) system console. You can switch to virtual console N by pressing Host-FnN, where Host is the right Control key (unless you've changed it). So to switch to virtual console 2, press Host-Fn2. You can then log in there.

2.4 The Linux kernel directory structure

Having done your background reading, you will have an idea of the general structure of the kernel. If you haven't already started, now is the time to look at the source code and see how it fits together.

In the directory `/group/teaching/cs3/os/VBox/linux-2.6.21.1/`, you will find the kernel source from which your virtual machine kernel was built.

The main subdirectories are as follows:

Documentation/
: What it says. There's a lot there, but some of it is not up to date. However, there is documentation on the scheduler: sched-design.txt and sched-coding.txt are the two worth looking at; sched-domains.txt describes a facility which I hope we won't need to understand.

arch/
: This is the directory where the architecture-specific parts of the Linux kernel live. There is only one directory relevant to us; all the others are for other types of machine. It is

  i386/
  : This is the code for the Intel PC architecture.

  Each of these directories has a structure like that of the top-level directory.

drivers/
: This contains the various device drivers included in the kernel code. You should not need to look at code in here.

fs/
: This contains code relating to filesystems. Again, you shouldn't need it.

include/
: This is a critical subdirectory. It contains most of the *.h files where the Linux kernel data structures are defined. The main subdirectory is

linux/
: which contains most of the *.h files, both those used by the kernel and those used by other programs needing access to Linux interfaces and data structures. There are very many files in here, most of which are irrelevant to you.

The other subdirectory is

asm/ (a symlink to asm-i386/)
: which contains architecture-specific data structure definitions.

init/
: This contains the kernel initialization code. You don't need it.

ipc/
: Deals with System V style IPC – not relevant.

kernel/
: This is the core of the Linux kernel, containing the code you should be looking at. Described in more detail later.

lib/
: contains a few useful routines for the kernel. (Unlike normal user programs, the kernel cannot use standard libraries. Exercise: why not?)

mm/
: contains the memory management code, which has been separated out from kernel/ because it's big and complex.

net/
: various code to do with networking.

scripts/
: stuff used in the kernel build process.

The kernel/ directory contains quite a few files, many of which should not concern us. The following are those that I believe may have to be looked at.

exit.c
: Deals with process termination and clean-up.

fork.c

```
" *  'fork.c' contains the help-routines for the 'fork' system call
  * (see also entry.S and others).
  * Fork is rather simple, once you get the hang of it, but the memory
  * management can be a bitch. See 'mm/memory.c': 'copy_page_range()' "
```

futex.c
: Hopefully not needed for the practical, but might be interesting as an example of mutex implementation.

resource.c
: General routines for doing resource management.

sched.c
: The main scheduler routines.

signal.c
: Implements the Unix process signal mechanism.

softirq.c

Various code to do with tasklets and related concepts (i.e. assorted kernel actions to be performed on behalf of processes).

sys.c

Implements many of the system calls by which the kernel is accessed by user programs.

sysctl.c

Implements the /proc filesystem, which allows many system parameters to be read from pseudo-files, or updated by writing to pseudo-files.

3 Phase 2: Simple exercise and more reading

3.1 Part 1: Compiling and loading kernel modules

In your background reading, you should by now have found out about the principles of Linux kernel modules.

First, it would be a good idea to skim through the Modules-HOWTO (available at `www.tldp.org`, of course), and/or The Linux Kernel Module Programming Guide.

The directory `/group/teaching/cs3/os/Modules` contains a "Hello World" module, together with a makefile for compiling it. Under 2.6, the kernel build procedure should be used for compiling standalone modules, and the kernel build procedure is a very complex beast. In this practical, you should be able to get away with trivial changes to the Makefile I've provided, and not need to understand the build procedure beyond this.

Copy this directory to a directory in your "shared" VM directory. In the virtual machine, change to that directory, and type

```
make
```

and you should get a compiled module hello.ko

Now (as root, if you weren't already), do

```
insmod hello.ko
```

You should see a message (two, in fact: you will see a warning that the licence "unspecified" "taints the kernel". This is because our hello module has not been declared to be a GPL module.)

If you do

```
lsmod
```

then you will see that indeed the module is loaded.

To unload the module, you do

```
rmmod hello
```

at which you should see another message.

If you now do

```
dmesg
```

you should see the messages at the end, so showing that they were indeed kernel messages, rather than just things printed to the terminal.

That's it: easy.

3.2 Part 2: Background on the shell

This part only involves just reading the following brief description on the UNIX shell. The OS command interpreter is the program that people interact with in order to launch and control programs. On UNIX systems, the command interpreter is usually called the shell: it is a user-level program that gives people a command-line interface to launching, suspending, and killing other programs. `sh`, `ksh`, `csh`, `tcsh`, and `bash` are all examples of UNIX shells.

Every shell is structured as a loop that includes the following:

1. Print a prompt (e.g., MyShell>).
2. Read a line of input from the user.
3. Parse the line into the program name, and an array of parameters.
4. Use the `fork()` system call to spawn a new child process.
   - The child process then uses the `exec()` system call to launch the specified program.
   - The parent process (the shell) uses the `wait()` system call to wait for the child to terminate.
5. Once the child (i.e. the launched program) finishes, the shell repeats the loop by jumping to 1.

Although most of the commands people type on the prompt are the name of other UNIX programs (such as `ls` or `cat`), shells recognize some special commands (called internal commands) that are not program names. For example, the `exit` command terminates the shell, and the `cd` command changes the current working directory. Shells directly make system calls to execute these commands, instead of forking child processes to handle them.

The structure of the main routine of a shell program may be similar to the following simple skeleton:

```
/* register a signal handler for Ctrl-C */
signal (...)

while (...)
{
    /* Wait for input */
    printf ("MyShell> ");
    fgets (...);
    /* Parse input */
    while (( ... = strsep (...)) != NULL)
    {
        ...
    }

    /* Check if the given command is internal one */
    if (/* command is cd */)
    {
        ...
        chdir (...)
        ...
    }
    else if (/* command is exit */)
    {
        ...
        exit (...)
    }

    /* Check if executable exists and is executable */


    /* Launch executable */
    if (fork () == 0)
    {
        ...
        execvp (...);
        ...
    }
    else
    {
        wait (...);
    }
}
```

3.3 Part 3: The scheduler

The major focus of phase 3 in Section 4 will be on the scheduler. Thus, you should now look in more detail at the scheduler, which is in some sense the core of the kernel. The paper **Understanding the 2.6.8.1 Linux Scheduler** will be very useful to you. The original copy can be downloaded from here. Print, if necessary, this 2-up on A4 paper copy. If you have not already looked at this, you should start reading it. It's 38 pages long. While there have of course been some significant changes between 2.6.8 and 2.6.21, this paper still gives an adequate overview for the purposes of this practical.

4 Phase 3: Implementation and experimentation

The assessed submission for this practical comprises four installments.

4.1 Part 1: Build a new shell (myshell.c)

Write a shell program (in C, not C++ or any other languages) **myshell.c** that has the following features:

- It should recognize two internal commands:
  - `exit [n]` terminates the shell, either by calling the `exit()` standard library routine or causing a return from the shell's `main()`. **If an argument (n) is given, it should be the exit value of the shell's execution.** Otherwise, the exit value should be the value returned by the last executed command (or 0 if no commands were executed.)
  - `cd [dir]` uses the `chdir()` standard library routine to change the shell's working directory to the argument directory. If no argument is given, the value of the `HOME` environment variable is used.
- If `Ctrl-C` (`^c`) is pressed (Hint: what signal does `^c` generate?), your shell prints a shell prompt on a new line.
- If the command line doesn't invoke an internal command, the shell assumes it is of the form `<executable name> <arg0> <arg1> .... <argN>`.
- Your shell uses the `fork()` standard library call, and some flavor of `exec()`, to invoke the executable, passing it any command line arguments.

Assume that executable names are specified as they are using "a real shell," i.e., name resolution involves the `PATH` environment variable (hint: is there a version of the `exec()` function that involves the `PATH` variable?). When your shell fails to locate the command to be executed, it prints out a warning message such as "No such file or directory". Try to use the same prompt (i.e., MyShell>) as in the following:

```
MyShell> date
Tue Sep 16 21:39:28 BST 2014
MyShell> /bin/cat /etc/motd /etc/shells
This is staff.compute.inf.ed.ac.uk running Scientific Linux 6 (sl6) DICE.
Please 'nice' all processes to preserve a quick response
on the command line.  'man nice' gives details.  Thanks.
/bin/sh
/bin/bash
/sbin/nologin
/bin/tcsh
/bin/csh
/usr/local/bin/bash
/usr/bin/tmux
/bin/dash
MyShell> ./date
./date: No such file or directory
```

Notes:

- The prompt (i.e., MyShell>) in the sample session above is output by the shell, and the words to the right of the prompt (e.g., "date" or "/bin/cat /etc/motd /etc/shells") are typed by the user.
- Please take a look at the manual pages for `execvp`, `fork`, `wait`, and `getenv`.
- To allow users to pass arguments to executables, you will have to parse the input line into words separated by whitespace (spaces and '\t' (the tab character), and then create an array of strings pointing at the words. You might try using `strtok()` for this (`man strtok` for a very good example of how to solve exactly this problem using it). See the readline library for a good method of taking input from the user.

- You'll need to pass the name of the command as well as the entire list of tokenized strings to one of the other variants of `exec`, such as `execvp()`. These tokenized strings will then end up as the `argv[]` argument to the `main()` function of the new program executed by the child process. Try `man execv` or `man execvp` for more details.
- Users tend to have a lot of bugs, and consequently dealing with them in a robust way can require a lot of code, especially if you're feeling like being helpful about reporting just what the mistake might be. Except as explicitly noted in the "specs" above, or when trivial to implement and of significant value, it's fine not to be very helpful toward the user in writing this shell.
- You are allowed to use the skeleton code presented in Section 3.2.

The total mark of this part is **30% of the practical marks**, which is divided as follows: The implementation and successful execution of function `exit [n]` is worth **5%**, those of `cd` function accounts for **5%**, handling `Ctrl-C` (`^c`) properly takes **10%** and the final **10%** is allocated to the implementation and execution of running external commands (e.g., `date`, `ls`, `mkdir`, etc.).

You should ensure that your source code is compiled with a simple command in the following:

```
gcc -o myshell myshell.c
```

You have to name the file as **myshell.c** and submit it as follows:

```
submit os 1 myshell.c
```

4.2 Part 2: Playing with kernel module

In the directory `/group/teaching/cs3/os/Modules` which you should already have copied into your shared directory, you will find a module `worker.c`. This module starts a new kernel thread (i.e. a thread running in kernel space), which just prints a message every ten seconds. When it is rmmod'ed, it asks the kernel thread to terminate.

You should note the use of a wait queue (basically a semaphore) to achieve synchronization between rmmod and the kernel thread. Since the kernel thread is running code from the module, the thread should terminate before the module is unloaded! Hence the `cleanup_module()` routine (which is called before the module is unloaded) sets a shared flag to ask the kernel thread to terminate, and then waits until the thread signals to the wait queue (semaphore) that it has finished.

Thus the expected behaviour is:

```
insmod worker.ko
```

and then get a greeting, followed by a message every ten seconds. When you do

```
rmmod worker
```

there will be a delay of up to ten seconds before the thread terminates and the module is removed.

The first part of the assessment simply asks you to verify this.

First, verify that you can compile and run the module, with the behaviour described above.

Now copy `worker.c` to `worker1.c`, and adjust the `Makefile` accordingly.

Edit the messages in `worker1.c` to something different, preferably including something unique and witty (but not your name, please). Compile the module, and then

```
insmod worker1.ko
```

wait for 30 seconds or so, and

```
rmmod worker1
```

Then capture the output of the dmesg command into a file in your shared directory, e.g. by

```
dmesg >/mnt/shared/part2-output
```

(For safety, unmount or shut down the VM before assuming that any files written to your home directory are actually there! You can also run 'sync' command.)

You should now submit this file using the command

```
submit os 1 part2-output
```

(You may re-submit at any time up to the practical deadline.)

That is the end of part 1; it is trivial, but it assures you that you have successfully understood how to use UML and modules.

Successful completion of this part instantly gets you **10% of the practical marks!**

4.3 Part 3: SMP-safe kernel module

This part contributes **20% of the practical.**

In this part, you need to prepare three files for submission. The first is a text file containing your description and analysis of a problem; the second is a corrected version of a program; the third is dmesg output as in Part 2. The text file should be called **part3-analysis**; the program should be called **fixedworker.c**; the output should be called **part3-output**.

Consider the worker module from Part 2.

Recall that when the module is rmmod'ed, it sets a flag to ask the kernel thread to stop, and then waits to be told by the thread that it is done. On a uniprocessing, unmodified, Linux system, this technique is safe, because kernel routines cannot be pre-empted. On an SMP system, however, the kernel thread might be running at the same time as the rmmod command.

In the directory `/group/teaching/cs3/os/Modules` there is a slightly modified worker, called `stuckworker.c`, which contains a delay to simulate the effect of possible multiprocessing.

Copy this file (and the `Makefile` - or update your own `Makefile` appropriately).

Compile and install this module, and then do exactly what you did for worker, namely, insmod, wait thirty seconds, and rmmod .

In the file part3-analysis:

- Describe what happens. [You should not exceed 10 lines]
- Analyse the cause of the problem. [You should not exceed 10 lines]
- Propose and explain a solution for an SMP-safe version of worker. [You should not need more than 20 lines]

Implement your solution in `fixedworker.c`. Run your `fixedworker` in the usual way, and capture the dmesg output in your `part3-output` file.

Hints: there are several ways to do this. You may find the `wait_event` macro (in `<linux/wait.h>`) of interest. You are free to use Google to assist you in solving this problem, but you should not consult humans. If you use a solution based on something on the net, you should acknowledge your source. Of course, you must understand and explain your solution.

When you have finished this part, and no later than the final practical deadline, submit your files with the command

```
submit os 1 part3-analysis fixedworker.c part3-output
```

DO NOT submit any other files, or use any other name for the files. Such files will not be marked.

If you have done the suggested reading and followed lectures, it should take at most a couple of hours, and possibly much less.

4.4 Part 4: Addressing CPU busy-waiting problem

This is the third and final installment, counting for **40% of the practical marks**.

In this part, you are asked to modify the worker module to do something (possibly) useful. As in previous parts, you should need to write little actual code. It should not take you more than a few hours, even allowing for making silly mistakes and crashing your kernel a few times.

NOTE: this has been updated for the 2.6 kernel we're using. However, it's always possible that I've made some silly mistakes in my understanding or explanations of the 2.6 code - so if you have any suspicions about anything, mail me or the class list.

First, some background.

There are certain I/O devices which, even these days, do not have a proper interrupt-driven interface. Rather, the CPU has to busy-wait while the data is transferred to the device. For example, PC-card SCSI adapters have this problem.

The consequence is that if you are transferring data to such a device, the CPU is busy-waiting almost all the time; worse, a single transfer may take significant (even by human standards) time. Since I/O is done in the kernel, which cannot be preempted, nothing else happens during that time. This brings system response down to a crawl: especially under X windows, where every keystroke results in many context switches. For example, if you back up a laptop to a SCSI tape drive attached via a PC-card, the machine will be effectively unusable for anything else during the backup, which may take several hours. It doesn't even help to "nice" the backup process, because on a normal system, even a nice'd process will get enough time to saturate the I/O interface.

You can see this problem in action on a kernel patched to fake the effect. We have modified the kernel you are using to simulate the problem on the virtual disk attached that appears as `/dev/sdc1`, and is mounted on `/mnt/sdc1`: the kernel busy-waits for 1 second on every access to this device.

To see this, issue a command that will transfer lots of data from the sdc1 disk, for example

```
ls -lR /mnt/sdc1
```

This will slowly produce a listing. Leave it running, and switch to another console (right-ctrl-F2) and log in: observe the lousy interactive response you get. You can switch back to the first console (right-ctrl-F1) and terminate the listing with ctrl-C, though it may take a little while to notice.

Your task

Your task is to produce a module that starts a kernel thread that takes action to alleviate this problem, by preventing busy-waiting processes from executing while there appears to be user activity going on. On the other hand, the busy-waiting process should not be completely starved: it should get some run time, even if the user is typing continuously. (The user will necessarily be delayed while the busy process is running, but that's too bad.)

To keep things simple, you should stick with the worker scheduling policy of just doing its work every ten seconds.

The worker's work should be as follows:

if a key has been pressed in the last ten seconds (i.e. since the last time round), any excessively busy process should be prevented from executing during the next minute (i.e. six times round).

I will now flesh this out a bit, so that you do not have to do excessive amounts of research in the kernel...

To detect key presses, it suffices to check whether the number of "keyboard" interrupts has changed since last time. On the standard PC architecture, the keyboard is on interrupt line 1.

You can get the total number of interrupts on line 1 since boot-up by calling `kstat_irqs(1)`. To get the `kstat_irqs` function, you need to

```
#include <linux/kernel_stat.h>
```

in the top of your program.

The next problem is, what does "excessively busy" mean? The following definition is not particularly principled, but is a plausible first guess, which has the advantage of automatically ensuring that the excessively busy process will run 10% of the time, even if there is a user working. We will say a process is "excessively busy" if all of the following are true:

1. It has used more than 1 second of CPU time in kernel routines. (This prevents your module from stopping short commands that just do a bit of I/O.) The figure of 1 second is a guess; you should make this a variable or `#define`, so that it can easily be changed.

2. It has used CPU time in kernel routines that amounts to more than 10% of the elapsed time since the process started.

3. It is a user process, not a kernel thread. (You may ignore this requirement, since in practice there are no kernel threads that will satisfy the first two requirements, but in principle it should be observed.)

For (1), you need to find the amount of CPU time a process has spent inside kernel routines. Fortunately, the kernel keeps track of this. It is stored in the stime field in the `task_struct`. It is kept in units of "jiffies"; there is a macro `cputime_to_sec` to convert jiffies to seconds, so if p is a pointer to a task struct, `cputime_to_secs(p->stime)` is the number of seconds of kernel time the task has consumed.

For (2), you need to know how roughly long the process has been running. The `start_time` field of the `task_struct` records the time the process started, measured in seconds since boot. This field is a struct timespec; the `tv_sec` field is seconds, so `p->start_time.tv_sec` is the time in seconds at which the process started. The current time in the same format may be obtained from `do_posix_clock_monotonic_gettime()` which requires that your module is declared to be

`MODULE_LICENSE("GPL")`.

Thus, your routine should traverse the task list (NOT the run queue!), check each task to see whether it is currently excessively busy, and if so, stop it from being executed in the next minute. Note that the task list can be traversed by the `next_task` macro, starting at `&init_task`.

There are (at least) two issues remaining that we may reasonably discuss.

Firstly, this problem seems to require you to keep some information about the excessively busy processes, so you can keep them stopped for a minute. In general, this would require you to allocate some kernel memory. I don't want to get into memory allocation in C, let alone kernel memory allocation. Therefore you may make the following simplifying assumption:

You may assume that there are at most N, for some fixed value of N, excessively busy processes.

If you don't even want to use arrays, you may even assume that there is at most one excessively busy process.

However, in either case, you should have your routine print a warning if it finds more than N busy processes. In that case, it should stop only the first N busy processes on the task list.

Secondly, how do you stop a process from being scheduled? There are several possibilities, which you will find by looking through the scheduler code.

As a further simplifying assumption, you may, **for 30 of the 40 marks**, assume a uniprocessor system. If you wish to obtain **the final 10 marks**, you should add the necessary code to make your routines SMP-safe.

Please add a comment on the FIRST LINE of your module code that is either

```
/* This module is *not* SMP-safe */
```

or

```
/* This module *is* SMP-safe */
```

Your module should use `printk` to print some diagnostics: at the least, when it disables or enables a busy process, it should log this, preferably with the PID and command name of the process.

You should name your module `relaxer.c`.

You should also capture the output of dmesg in a file `part4-output` immediately after doing a successful test of your module.

Finally, a warning: even in the relatively friendly environment of a virtual machine, mistakes in kernel routines are annoying. I strongly suggest that you develop incrementally, testing every small change. For example, check your task list scanning code with some printk's, before you think about doing anything with the task.

By the practical deadline, you should submit your answers via

```
submit os 1 relaxer.c part4-output
```