Operating Systems Concepts **Tutorial exercises**

Introduction to the Simple Kernel

This exercise sheet is available on-line at
http://www.doc.ic.ac.uk/~wjk/OperatingSystemsConcepts

This exercise is intended to

1. Introduce you to the Simple Kernel.
2. Give you practical experience of what an operating system kernel is, what services it provides, how these services are invoked and what the underlying code looks like.
3. Reinforce concepts of process synchronisation in the context of the Simple Kernel.
4. Give you more experience of a Linux development environment.

What you should do:

1. Find a PC running Linux, or reboot a lab machine that is running Windows NT and select Linux from the boot menu.

2. Log in and click on the "shell" icon at the bottom of your screen to bring up a Linux command line console.

3. At the Linux command line type:

   ```
   cd

   cp –rd ~wjk/SimpleKernelExercise .
   ```

   (please note the <space><dot> at the end of the above command – the dot indicates the current directory; also do not forget the “d” option on the copy) This should give you a copy of all the files you need for this exercise in a folder called `SimpleKernelExercise` (you can check everything went according to plan by doing an `ls –l` to check that the folder `SimpleKernelExercise` now exists).

4. At the Linux command line type:

   ```
   cd SimpleKernelExercise

   ls –l
   ```

   You should see two directories (amongst other files and directories):

`BochsSimulatorFiles`

`ICOS`

The Bochs simulator is a PC simulator that will save you the trouble of creating a boot floppy and rebooting your PC each time you want to run the Simple Kernel.

ICOS (Imperial College OS) is a mini-operating system consisting of the Simple Kernel and modules scavenged from Linux.

5. At the Linux command line type:

   `cd ICOS`

   `ls –l`

   You should see:
   (a) a number of text files (`README, TOUSE, USERCALLS`) which you can read with `cat <filename>` (or `more <filename>).`
   (b) some directories (`boot, system, tools`). `boot` and `tools` contain code adapted from Linux; `system` contains the Simple Kernel source code.
   (c) a script `simulate` which will compile everything together and run the PC simulator (as an alternative you can type `make floppy` to make a boot disk which you can then use to boot the PC. WARNING: `make floppy` will erase everything on the disk, without asking you if this is what you intended!)

6. Change into the `system` directory and get a directory listing. Inspect some of the source code files (use `emacs, vi` or `nedit` according to your taste and experience), e.g. have a look at `proc.c, time.c, sem.c, switch.S` and `main.c).` Hopefully the contents will seem familiar!

7. Look at the file `user.cpp` and see if you can predict what will happen when the kernel starts running.

8. Go back to the previous directory (`cd ..`) and type `./simulate`. Hopefully the PC simulator will start running, and you will be able to see if your prediction was correct.

9. Go back into the `system` directory and edit `user.cpp.` Can you use Simple Kernel semaphores to make two communicating processes such that one performs a V operation on the semaphore every 5 clock ticks, while the other repeatedly performs a P operation on the semaphore and then outputs a ‘*’?
   (you can see if your changes compile by typing `make` in the `system` directory; when everything compiles go back to the previous directory and run

`./simulate`).

10. Have some fun playing around with the Simple Kernel code and watching the effect it has. Some recommendations:

    - add a `con_real_outc('c'); sta`tement to the `tick()` function in `time.c` so you can see when clock ticks occur. Can you figure out how the function `tick()` gets called?
    - add output statements to the interrupt handler `int_interrupt(int inum)` in `int.c` so you can see when interrupts occur.
    - create your own processes in `user.cpp` with different priorities and try to predict how they will be scheduled (next week's tutorial will cover this topic in detail).