

## Introduction to the Simple Kernel

### 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 to do

1. You need a PC running Linux, ideally a machine that is OK to reboot (i.e. which does not have one of the DO NOT REBOOT stickers).
2. Copy the file `~ob3/SimpleKernelExercise1.tar.gz`  
(You can do that by typing `cp ~ob3/SimpleKernelExercise1.tar.gz .` – the dot at the end being crucial.)  
*A .tar.gz file is a compressed archive of files – usually of a directory structure that contains a software distribution. This is one of the most common formats in which open-source software is distributed – learn to love it ☺.*
3. Unpack the archive file by typing  
`tar xvzf SimpleKernelExercise1.tar.gz`  
*It's worth having a look at the manual pages for the tar command. If you just want to see what files are contained in the archive, substitute the x (for "extract") in the first argument for the above command for a t (for "type"). The other letters in the first argument mean: v for verbose, z indicates that this is a compressed archive and f means that the next argument is the filename of the archive. Alternatively to the above command, you can use gunzip to uncompress the archive first, resulting in a file with just the .tar extension. You can then use the tar command without the z in the first argument.*
4. Unpacking the archive file creates a directory called `SimpleKernelExercise1`. Explore that directory.  
You should find two immediate subdirectories: `BochsSimulatorFiles` and `ICOS`. The Bochs simulator (see also <http://bochs.sourceforge.net>) is a PC simulator that will save you the trouble of always 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. In the `ICOS` subdirectory you should see
  - A number of text files (`README`, `TOUSE`, `USERCALLS`) which you can read with `less <filename>` (type "q" to get out of less, you can use arrows etc to navigate the file while in less).
  - Some directories (`boot`, `system`, `tools`). `boot` and `tools` contain code adapted from Linux; `system` contains the Simple Kernel source code.

- 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. Explore the `system` directory. Inspect some of the source code files, e.g. have a look at `proc.c`, `time.c`, `sem.c`, `switch.S` and `main.c`). Hopefully some of the contents will seem familiar!  
Look at the file `user.c` and see if you can predict what will happen when the kernel starts running.
  7. (If you have an *EMPTY* floppy disk)  
From the `ICOS` directory, type `make floppy`, with your floppy in the drive. Then reboot the computer (floppy still in drive) and observe what happens. To exit, press Ctrl-Alt-Delete and take out the floppy as soon as the computer shuts down to reboot.  
Did you see what you expected to see by inspecting `user.c`?  
*This part is to show you that Simple Kernel is a real OS, capable of running on a real PC. For actually working with Simple Kernel, it is easier to use a PC simulator than to keep making a boot floppy and rebooting.*
  8. From the `ICOS` directory, type `./simulate`. Hopefully the PC simulator will start running. Compare with what you saw in the previous step. To exit, press the “on/off” button at the top of the simulator window.  
Did you see what you expected to see by inspecting `user.c`?
  9. Go back into the `system` directory and edit `user.c`. Can you use Simple Kernel semaphores to make communicating processes as follows:
    - Create a globally shared integer variable, initially set to ‘x’.
    - Create two processes that regularly set the value of this variable – one sets it to ‘y’, the other to ‘n’.
    - Create a process that regularly prints out the value of the shared variable.
    - Use a single semaphore to make sure that only the value ‘y’ can get printed. *You will have to make sure that the printing function and the function setting the variable to the wrong value exclude each other.*
  10. Play around with the Simple Kernel code and watching the effect it has. Some suggestions:
    - What happens when you take out the line that starts the `kbd2screen` process in `user.c`?
    - Add a `con_real_outc('c')` statement 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).