ICE4131 - High Performance Computing (HPC)

Lab 1: Editing files

Tutor: Franck Vidal

Objectives

The aim of this tutorial is to introduce you to the GNU/Linux environment used by SuperComputing Wales. You will learn how to

1. log in,
2. write and
3. compile code, and
4. run your first parallel tasks using POSIX threads (also known as Pthreads).

At the beginning of the session next week, each student will demonstrate that they are capable to complete every task. **You will demonstrate your work next week. There will be a mark assigned to each task. It will count toward the final lab mark.**

Logging in to the SuperComputer

To access SuperComputing Wales, you can use a terminal emulator, which connects your localkeyboard and screen to the remote system. It uses a protocol called "Secure Shell", or "SSH"; this is available as standard on a Linux workstation, but requires a terminal emulator to be downloadedand installed on a Windows workstation. In order to access SuperComputing Wales, you need tolog in to the main login server (see Figure 1). From there you will have access to the compute nodes:

- 201 nodes,
- totalling 8,040 cores,
- 46.080 TBytes total memory

Each node has

- CPU: 2x Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz with 20 cores each
- RAM: 192 GB by default, or 384GB on high memory and GPU nodes
- GPU: 26x nVidia P100 GPUs with 16GB of RAM on 13 nodes
- Storage:
  - 692TB (usable) scratch space on a Lustre filesystem
  - 420TB of home directory space over NFS

Logging in from Windows

Check if your machine has "PuTTY" installed; if not, you can download a copy from https://www.chiark.greenend.org.uk/sgtatham/putty/. The software is open source and freely available. If you cannot install the software locally (e.g. as you do not have administrator rights/privileges), then you may be able to install a "portable application" packaged version, available from http://portableapps.com/apps/internet/putty_portable. When you have PuTTY installed, run it and you will see a screen similar to that presented in Figure 2.

In order to log in to SuperComputing Wales, enter "hawklogin.cf.ac.uk" into the "Host Name (or IP address)" box; you may wish to save this for reuse, in which case also enter a name in the "Saved Sessions" box, and click the "Save" button.

After entering the address of the machine you wish to log into the Hawk system, click on the "Open" button to connect to SuperComputing Wales. You may be asked to accept a securitycertificate. This will happen the first time you log into the cluster from your machine. Accept thecertificate by clicking on "yes", and you will be connected to the login node of SuperComputingWales, your first point of entry as shown in Figure 3.

You are now ready to log in, please enter your username, in the form "b.eese10", such as presented in Figure 4 and press return.

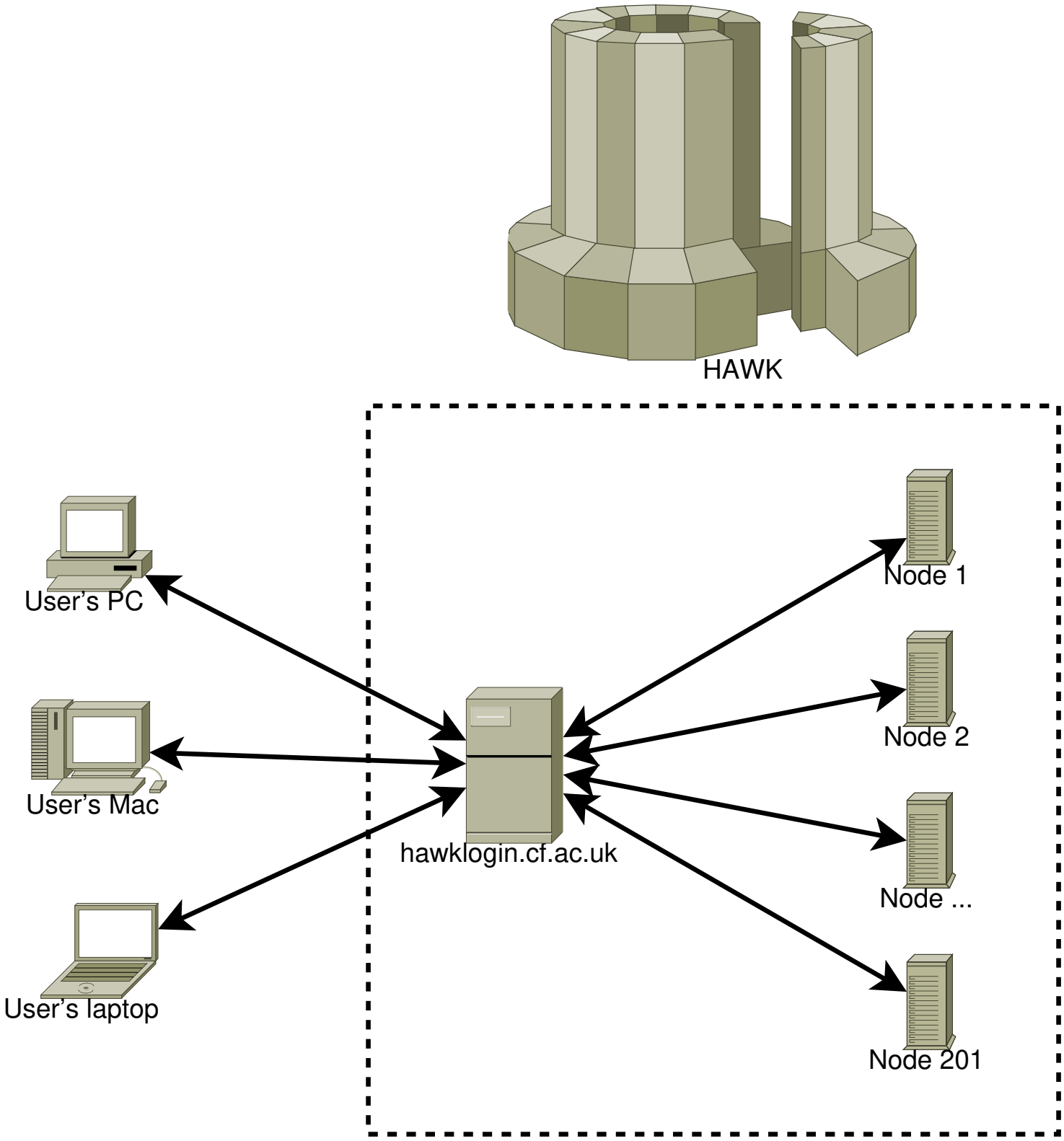

Figure 1: Overall structure

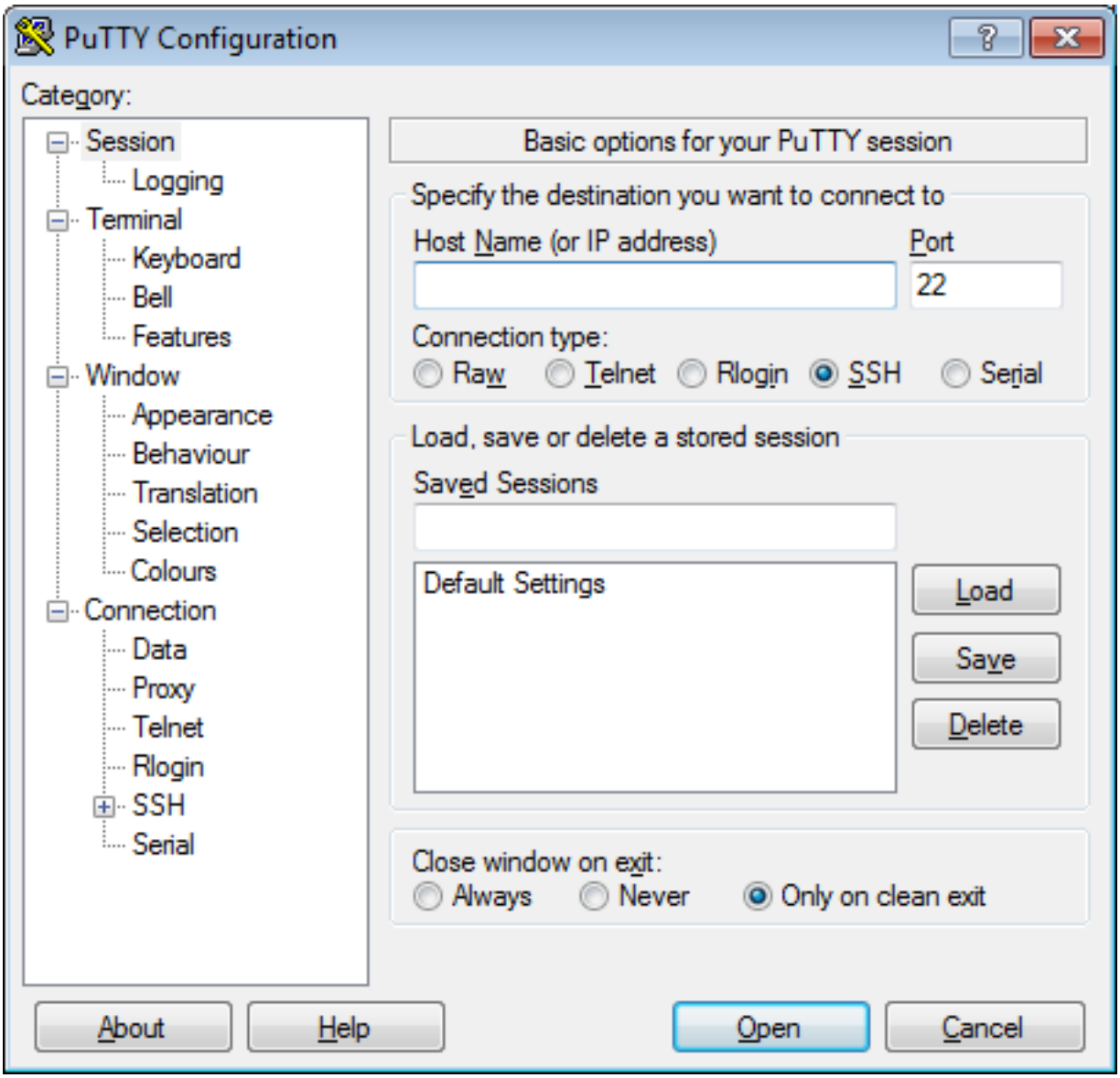

Figure 2: PuTTY Configuration Screen

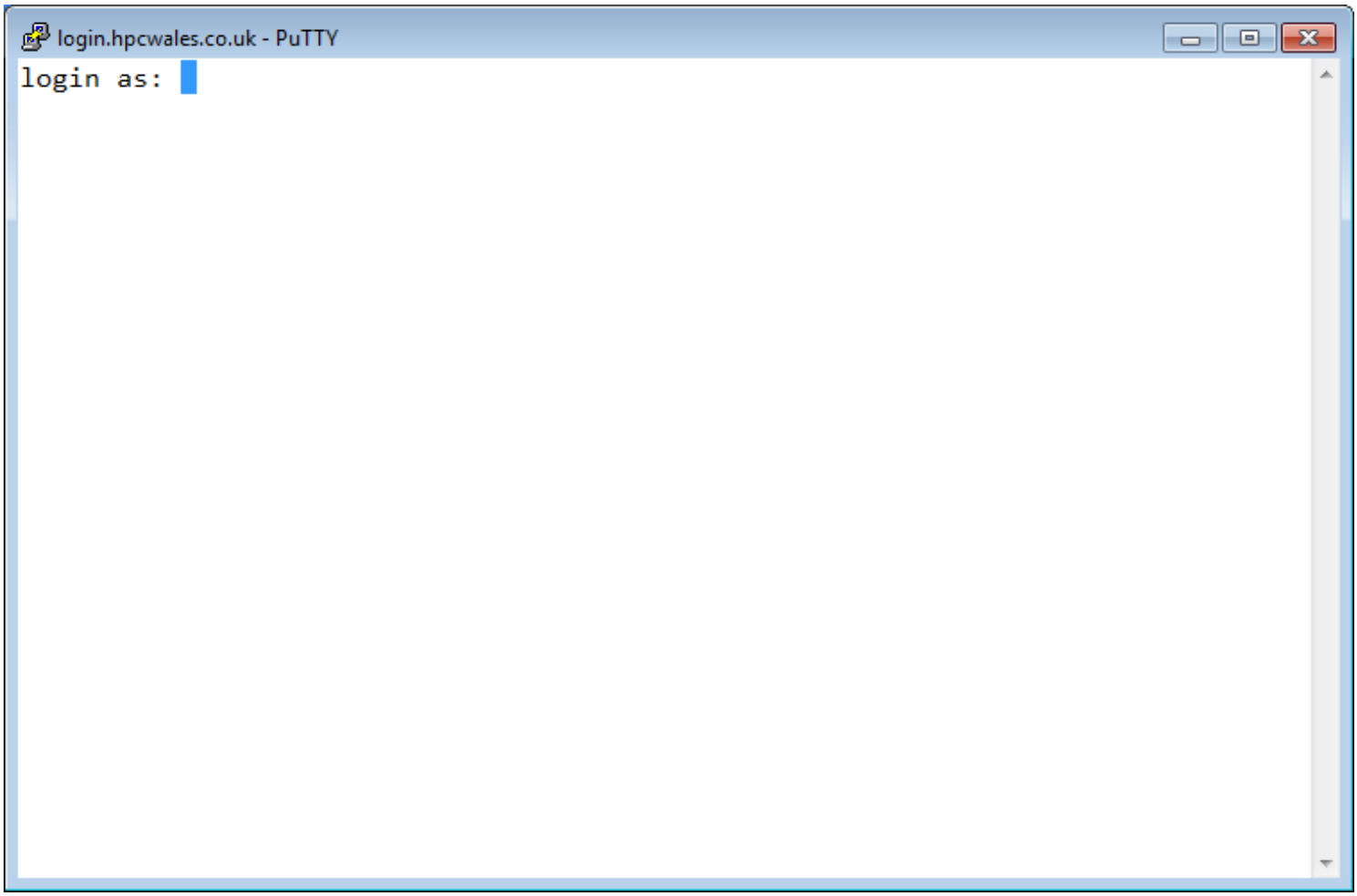

Figure 3: Login prompt in PuTTY

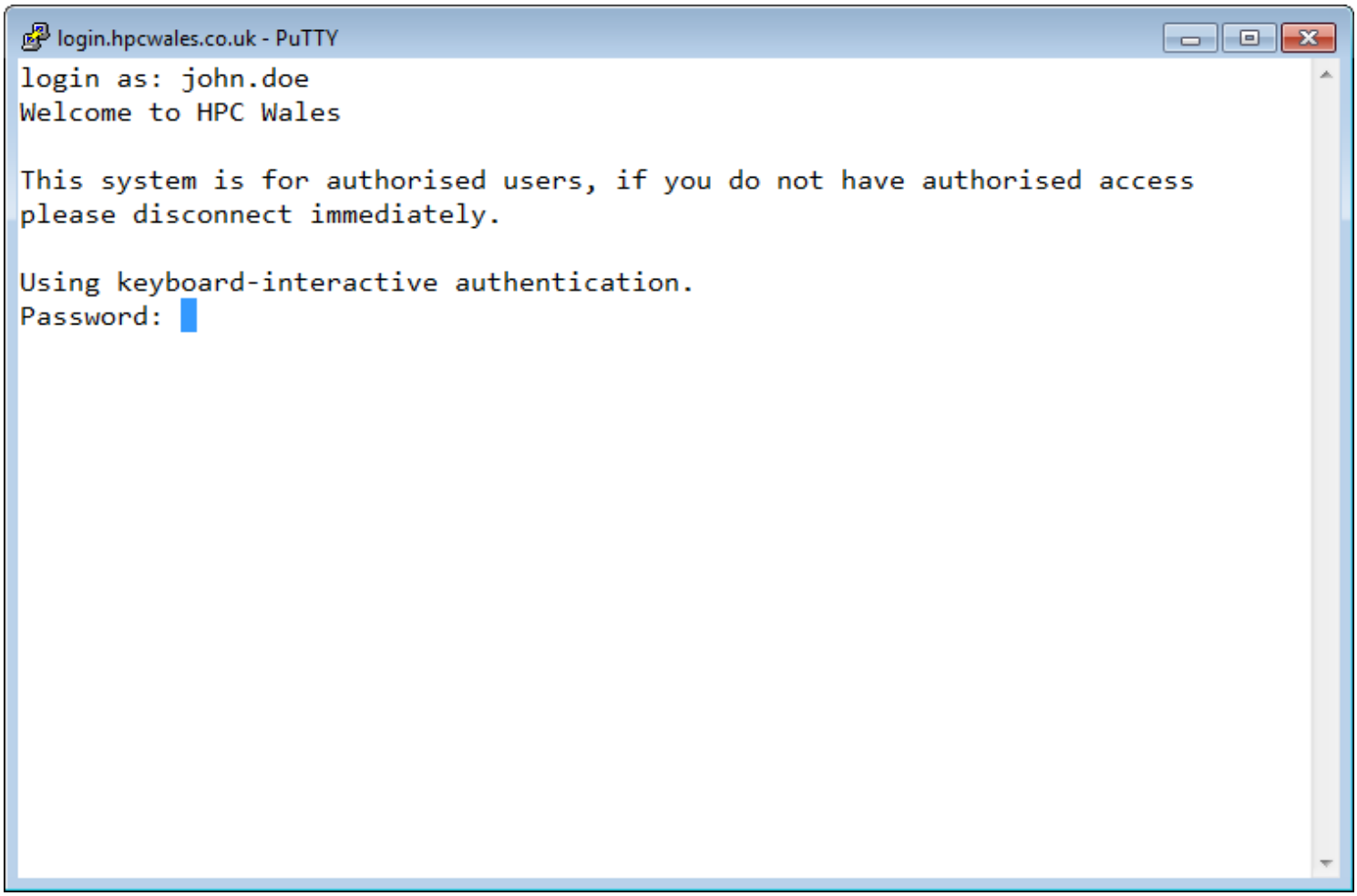

Figure 4: Entering Your Username in PuTTY

Logging in from GNU/Linux or MacOSX

Linux and OSX have secure shell built-in, so you should not need to install it manually. To log into SuperComputing Wales, open a Terminal Window and type:

```
$ ssh b.eese10@hawklogin.cf.ac.uk
```

Where you should replace "b.eese10" with your username. DON'T FORGET THE "b." as you are using a Bangor University account. You may be asked to accept a security certificate. This will happen the first time you log into the cluster from your machine. Accept the certificate by typing "yes" and press return. Proceed to the next section

Entering your Password

Your password will now be requested; please enter your password, only you knows it. You are now at the command prompt on the login node of SuperComputing Wales. This is the machine that links SuperComputing Wales to the internet, and is heavily secured (think of it as the gate keeper to the SuperComputing Wales network).

Basic Linux commands

Command Prompt Basics

- `man man`

  Displays manual information on the manual command

- `man [command]`

  Displays manual information on command

- `clear`

  Clears the screen

- `exit`

  Exits the command interpreter

Manipulating Directories

- `cd ..`

  Change to the parent directory

- `cd [directory]`

  Change to directory `[directory]`

- `mkdir [directory]`

  Create directory `[directory]`

- `rmdir [directory]`

  Remove directory `[directory]`

Listing Files

- `ls`

  Display list of files and sub directories in standard format `<name>` excluding hidden files

- `ls -a`

  Display list of files and sub directories in standard format `<name>` including hidden files

- `ls -l`

  Display list of files and sub directories in long format `<permissions owner group size datetime name>`

- `ls -lh`

  Display list of files and sub directories in long format `<permissions owner group size datetime name>` with human readable size

- `ls -lt`

  Display list of files and sub directories in long format `<permissions owner group size datetime name>` sorted by time

- `ls -lr`

  Display list of files and sub directories in long format `<permissions owner group size datetime name>` in reverse order

- `ls -ltrh`

  Display list of files and sub directories in long format `<permissions owner group size datetime name>` sorted by time, in reverse order, with human readable size

Moving Files

- `mv [source] [dest]`

  Move file `[source]` to file `[dest]`

- `mv -i [source] [dest]`
  - Move file `[source]` to file `[dest]`
  - Prompt before overwriting `[dest]` if it exists

- `mv -f [source] [dest]`
  - Move file `[source]` to file `[dest]`
  - Overwrite `[dest]` if it exists

Removing Files

- `rm [file]`

  Remove file `[file]`

- `rm -i [file]`
  - Remove file `[file]`
  - Prompt before removing

- `rm -R [directory]`
  - Remove directory `[directory]`
  - Remove all sub directories and files

Copying Files

- `cp [source][dest]`

  Copy file `[source]` to file `[dest]`

- `cp -i [source][dest]`

  - Copy file `[source]` to file `[dest]`
  - Prompt before overwriting `[dest]` if it exists
- `cp -R [source][dest]`

  Copy directory [source] to directory [dest]Copy all sub directories and files

Displaying Files

- `more [file]`
  - Display `[file]` on the screen
  - Will scroll through one screen at a time
  - Press space to scroll one screen at a time
  - Press enter to scroll one line at a time

Comparing Files

- `diff [file1] [file2]`

  Display differences between `[file1]` and `[file2]`
- `fgrep "string" [file]`

  Find "`string`" in `[file]`
- `sort [file]`

  Sort `[file]`

Command Modifiers

- Wildcards allow you to specify multiple items to operate on
  - `ls *.txt`
  - `rm *.txt`
- Redirection allows you to direct the output of one command to a file
  - `sort unsorted.txt > sorted.txt`
- Filters are external commands that change data in some manner
  - `fgrep "string" [file]`
- Pipes let you direct the output of one command as input to another
  - `ls | find "txt"`

Other Commands

- `who`

  Show who is logged on
- `top`

  Show which tasks are running
- `watch`

  Run a task repeatedly
- `history`

  Show which tasks you ran
- `date`

  Display or set the date and time

- `cat`

  Concatenate files and print on screen

- `head`

  Print top of file on screen

- `tail`

  Print bottom of file on screen

- `uniq`

  Report or omit repeated lines

- `pwd`

  Display name of current/working directory

- `hostname`

  Display the system's host name

Editing Files

There are three main console-based text editor:

- Nano
- Emacs
- Vi

If you are not familiar with Vi nor Emacs, then I recommend you to use Nano. It is easier to handle.

Compiling code

In your home directory, create a new sub-directory called "LAB1". In "LAB1", create "HELLOWORLD". Go to LAB2/HELLOWORLD, create a new file, "HelloWorld.cxx", and type the source code as follows using your favourite text editor:

```
#include <iostream>

int main(int argc, char** argv)
{
    for (int i(0); i < 10; ++i)
    {
        std::cout << "Hello World." << std::endl;
    }

return (0);
}
```

To create an executable file, you need to compile your source code. This is what we are going to do next.

Module Environment

With a UNIX system, such as SuperComputing Wales, it can get very complicated for the end user to maintain many different versions of available programs. To simplify matters, we use the "module environment" - which enables the user to pull in software as and when it is needed - and also enables the user to specify which version they want. To find out what software is available, type:

```
$ module avail
```

It lists all the available software for you to use.

To look at compilers only, type:

```
$ module avail compiler
```

It will return:

```
------------------------------------------- /apps/modules/compilers ------------------------------------------
compiler/gnu/4/8.5    compiler/gnu/7/3.0    compiler/intel/2017/4 compiler/intel/2018/3
compiler/gnu/5/5.0    compiler/gnu/8/1.0    compiler/intel/2017/7 compiler/intel/2018/4
compiler/gnu/6/4.0    compiler/intel/2016/4 compiler/intel/2018/2 compiler/pgi/18/4
```

For us, we will require one tool to compile some source code, and to run the resulting executable code. At present, compilation tools are not available - for example if you type

```
$ icc
```

the command prompt will respond with

```
-bash: icc:
command not found
```

If you type `g++ --version`, you will get an old version of g++, e.g. g++ (GCC) 4.8.5 from 2015. We will be using the latest GNU compiler suite; to install it, type:

```
$ module load compiler/gnu/8/1.0
```

This will load in the new compiler we wish to use. Now, if you type in `g++ --version`, you will see "g++ (GCC) 8.1.0".

To save having to type this in each time we log into the machine, we have put these commands in a batch script for you - this is a short text file that contains the above commands. Let's now start to look at compiling the example code, and using the batch script file to save typing. This is covered in the next section.

Compiling Code: C++

To compile your source code, just type:

```
$ g++ HelloWorld.cxx -o HelloWorld
```

If you did not make any mistake, it should compile without any error. A new file, "HelloWorld", was created. To run it, type:

```
$ ./HelloWorld+
```

Parallelising Code using Pthreads

General concepts

A very common strategy to parallelise code is to identify for/while loops and replace them by parallel code. In the listing above, there is a for loop at Line 5. We are going to parallelise it using POSIX Threads (usually referred to as Pthreads). It is a POSIX standard for threads. Pthreads are a simple and effective way of creating a multi-threaded application.

There are 5 main steps to convert serial code to parallel code with Pthreads:

1. All C/C++ programs using Pthreads need to include the pthread.h header file.
2. `#include <pthread.h>` at the top of your file.
3. Create an entry point for the thread
   - When creating a thread using Pthreads, you need to point it to a function for it to start execution.
   - It is the thread's callback function.
   - It returns `void*` and take a single `void*` argument.
   - For example, if you want the function to take an integer argument, you will need to pass the address of the integer and dereference it later. This may sound complicated but, as is shown below, it's pretty simple. An example function signature would be:

```
void* callback_function(void* param);
```

4. Define thread reference variables
   - The variable type `pthread_t` is a means of referencing threads.
   - There must be a `pthread_t` variable for every thread being created!
   - For example, it can be: `pthread_t thread0;`
5. Create the thread
   - Once the `pthread_t` variable has been defined and the entry point function created, we can create the thread using `pthread_create`.
   - This method takes four arguments:
     1. A pointer to the `pthread_t` variable,
     2. Any extra attributes (don't worry about this for now - just set it to `NULL`),
     3. A pointer to the function to call (i.e. the name of the callback) and
     4. The pointer being passed as the argument to the function.
   - Now there's a lot of pointers in that call, but don't stress - it's not as tricky as it sounds. This call will look something like:

```
pthread_create(&thread0, NULL, callback_function, &parameter);
```

6. Join everything back up
   - When the newly-created thread has finished doing it's bits, we need to join everything back up.
   - This is done by the `pthread_join` function which takes two parameters: the `pthread_t` variable used when `pthread_create` was called (not a pointer this time) and a pointer to the return value pointer (don't worry about this for now - just set it to `NULL`).
   - This call will look something like:

```
pthread_join(thread0, NULL);
```

And that's all there is to it. The function used as the thread entry point can call other functions, create variables or do anything any other function can do. It can also use the variables set by the other thread.

First example: parallel HelloWorld

Below is the HelloWorld program with parallelisation using Pthread. The for loop has been replaced by concurrent threads. Create a new file "HelloWorld-pthread1.cxx" with the source code below.

```
#include <iostream>
#include <vector>
#include <pthread.h> // 1. Add the Pthread header file

// 2. Declare the callback
void* callback_function(void* param);

//------------------------------
int main(int argc, char** argv)
//------------------------------
{
    // The total number of threads
    unsigned int number_of_threads(10);

    // 3. Declare an array of 10 pthreads
    std::vector<pthread_t> p_pthread_set(number_of_threads);

    // 4. Create the threads
    for (int i(0); i < number_of_threads; ++i)
    {
        pthread_create(&p_pthread_set[i], 0, callback_function, 0);
    }

    // 5. Join the threads
    for (int i(0); i < number_of_threads; ++i)
    {
```

```
        pthread_join(p_pthread_set[i], 0);
    }

    return (0);
}


//----------------------------------
void* callback_function(void* param)
//----------------------------------
{
    std::cout << "Hello World." << std::endl;

    return (0);
}
```

When compiling the program, you will also need to add `-lpthread` to the compile command:

```
g++ HelloWorld-pthread1.cxx -lpthread -o HelloWorld-pthread1
```

Now run HelloWorld-pthread1. **Can you explain what happens?** To make it a little bit easier to understand what is happening, complete the next section.

Second example: passing parameters

In this example, we show how to pass parameters to a thread. We add an extra `vector` to store a unique numerical ID (as `unsigned int`) for each thread (see Lines 20 and 25). Remember that the address of the variable is passed to the thread (see `&` at Line 26). In the callback, the address as to be converted from `void*` to `unsigned int*` (see Line 44). Then it is possible to retrieve the numerical value (see Line 47).

```
#include <iostream>
#include <vector>
#include <pthread.h> // 1. Add the Pthread header file

// 2. Declare the callback
void* callback_function(void* param);

//------------------------------
int main(int argc, char** argv)
//------------------------------
{
    // The total number of threads
    unsigned int number_of_threads(10);

    // 3. Declare an array of 10 pthreads
    std::vector<pthread_t> p_pthread_set(number_of_threads);

    // Declare an array so that an unsigned int value
    // is associated with a thread
    std::vector<unsigned int> p_data_set(number_of_threads);

    // 4. Create the threads
    for (int i(0); i < number_of_threads; ++i)
    {
        p_data_set[i] = i; // Save the data
        pthread_create(&p_pthread_set[i], 0, callback_function, &(p_data_set[i]));
    }

    // 5. Join the threads
```

```
    for (int i(0); i < number_of_threads; ++i)
    {
        pthread_join(p_pthread_set[i], 0);
    }

    return (0);
}


//---------------------------------
void* callback_function(void* param)
//---------------------------------
{
    // Convert from void* to unsigned int*
    unsigned int* p_i(static_cast<unsigned int*>(param));

    // Get the numerical value from the pointer
    unsigned int i(*p_i);

    std::cout << "Hello World: " << i << std::endl;

    return (0);
}
```

Create a new file "HellowWorld-pthread2.cxx" with this source code. You can copy HellowWorld-pthread1.cxx into HellowWorld-pthread2.cxx to save some time:

```
$ cp HellowWorld-pthread1.cxx HellowWorld-pthread2.cxx
```

Now modify the content of "HellowWorld-pthread2.cxx". Compile your new program and run it several times. **Can you explain what is happening?**

Third example: Mutual exclusion

A critical section is a piece of code that accesses a shared resource. In the previous example, threads shared the standard output and tried to write into it at the same time. This is why the output is so messy. To tidy it up, we want to make sure that only one thread writes in the console at a time. In other words, we will ensure that no two concurrent threads are in their critical section at the same time. This mechanism is called "Mutual exclusion" (commonly known as "mutex").

We will add three lines of code to the previous example. On Line 10, we declare and initialise a mutex. Before the critical section (i.e. before writing in the console), we lock the mutex (see Line 55). After the critical section, we release the mutex (see Line 60).

This is what we do in a new file "HellowWorld-pthread3.cxx":

```
#include <iostream>
#include <vector>
#include <pthread.h> // 1. Add the Pthread header file

// 2. Declare the callback
void* callback_function(void* param);


// Global variable
pthread_mutex_t g_console_mutex = PTHREAD_MUTEX_INITIALIZER;


//----------------------------
int main(int argc, char** argv)
//----------------------------
{
```

```
    // The total number of threads
    unsigned int number_of_threads(10);

    // 3. Declare an array of 10 pthreads
    std::vector<pthread_t> p_pthread_set(number_of_threads);

    // Declare an array so that an unsigned int value
    // is associated with a thread
    std::vector<unsigned int> p_data_set(number_of_threads);

    // 4. Create the threads
    for (int i(0); i < number_of_threads; ++i)
    {
        p_data_set[i] = i; // Save the data
        pthread_create(&p_pthread_set[i], 0, callback_function, &(p_data_set[i]));
    }

    // 5. Join the threads
    for (int i(0); i < number_of_threads; ++i)
    {
        pthread_join(p_pthread_set[i], 0);
    }

    return (0);
}


//---------------------------------
void* callback_function(void* param)
//---------------------------------
{
    // Convert from void* to unsigned int*
    unsigned int* p_i(static_cast<unsigned int*>(param));

    // Get the numerical value from the pointer
    unsigned int i(*p_i);

    // Lock the mutex and then wait for signal to relase mutex
    pthread_mutex_lock(&g_console_mutex);

    std::cout << "Hello World: " << i << std::endl;

    // Unlock the mutex
    pthread_mutex_unlock(&g_console_mutex);

    return (0);
}
```

Compile and run “HellowWorld-pthread3.cxx”, voilà! The output is presented as expected. Only one thread writes in the standard output at a time.