Objetivos de la sesión

En esta sesión se introduce al alumno el entorno de trabajo que se utilizará a lo largo de gran parte de las prácticas de laboratorio. Este entorno se basa en el sistema operativo GNU/Linux. Para esta sesión introductoria los principales objetivos son:

- Preparar el entorno e introducir el flujo de trabajo.
- Introducir el sistema operativo GNU/Linux y sus comandos básicos.
- Aprender a conectarse a máquinas GNU/Linux remotas y transferir ficheros.

Para conseguir estos objetivos se arrancará un sistema operativo Linux.

Conocimientos y materiales necesarios

Para poder realizar esta sesión, el alumno debe:

- Disponer de una máquina, física o virtual, con un sistema operativo Ubuntu Server 16.04.6 LTS 32 bits sobre el que se trabajará a lo largo del curso. En la parte inicial de la práctica se explica cómo crear la máquina virtual.

1. Preparando el entorno

Inicialmente debemos realizar unos preparativos para poder realizar las prácticas de la asignatura. Estos comprenden la instalación de la máquina GNU/Linux que vamos a utilizar y la definición del flujo de trabajo a seguir.

1.1. Creación de la máquina virtual

Para el desarrollo de las prácticas utilizaremos un sistema GNU/Linux. Puede servir tanto una máquina física como virtual, por lo que utilizaremos esta última opción por simplicidad.

Los profesores de la asignatura hemos preparado una máquina virtual con todas las herramientas necesarias que utilizaremos a lo largo de las prácticas. Concretamente, contiene un *driver* desarrollado específicamente que se carga automáticamente al arrancar el sistema operativo.

- Abre VirtualBox y elije la opción **Archivo › Importar servicio virtualizado** y selecciona el fichero que con la imagen de la máquina virtual que te indique el profesor. Opcionalmente, puedes descargar el fichero con la máquina virtual a tu ordenador.

  > Es posible que tengas que modificar antes la configuración de VirtualBox para crear la máquina en un directorio en el que tengas permiso de escritura. Asegúrate que las las máquinas se guardan en tu perfil de usuario usando la opción **Archivo › Preferencias**.

- Establece el nombre de la máquina virtual a `2AC` y deja el resto de opciones por defecto. Activa la opción ***Reinicializar la dirección MAC de todas las tarjetas de red***.
- Pulsa **Importar**.

1.2. Arrancando el sistema

GNU/Linux es un sistema operativo multitarea y multiusuario, lo que indica que puede ejecutar varias tareas de distintos usuarios al mismo tiempo. Este operativo pertenece a la familia de operativos UNIX, por lo que presenta grandes similitudes con otros sistemas operativos como BSD o Mac OS X.

> - Inicia la máquina con el sistema GNU/Linux instalado.

GNU/Linux ofrece dos modos de trabajo al usuario: modo gráfico y modo texto. En el modo gráfico se puede trabajar sobre el equipo a través de un escritorio similar al que puede encontrarse en otros sistemas operativos de ventanas. El modo texto ofrece la misma funcionalidad que el modo gráfico, con la desventaja de que todas las operaciones sobre el equipo deben realizarse a través de una interfaz de comandos. Este último modo es el utilizado habitualmente cuando nos conectamos de forma remota al equipo o bien cuando el sistema no tiene configurado el modo gráfico, típicamente en servidores, para ahorrar recursos en la máquina.

Dada la gran flexibilidad que ofrece la interfaz de comandos del operativo, y que no siempre estará disponible el modo gráfico cuando trabajemos con máquinas UNIX, este será nuestro entorno de trabajo. A continuación se introducirán los comandos básicos de la interfaz de comandos.

2. Interfaz de comandos

Cuando accedemos al sistema utilizando el modo texto de UNIX directamente aparece la interfaz de comandos del sistema operativo.

Puedes acceder a través de la consola de la máquina si estás ante ella (ya sea física o virtual), o bien a través de la red utilizando SSH. Necesitarás conocer la IP de la máquina para conectarte usando, por ejemplo, el programa PuTTY. En estas prácticas se asumirá este último caso, pues siempre podrá seguirse. En la consola de PuTTY puedes copiar texto seleccionando con el botón izquierdo del ratón y pegar pulsando el botón derecho.

> - La máquina virtual se configura en modo NAT e incluye una regla que redirige el tráfico SSH, por lo que debes utilizar la dirección `localhost` y el puerto `60022` para conectarte a la máquina virtual con PuTTY.
>
>> La primera vez que te conectes se mostrará un aviso.
>
> - Entra en sesión con el nombre de usuario `student` y la contraseña `student`.
> - Cambia la contraseña siguiendo las instrucciones. Apunta la contraseña utilizada (idealmente, en tu gestor de contraseñas).
>
>> Ten cuidado cuando entres en sesión por primera vez para cambiar la contraseña. Inicialmente, se te pedirá la contraseña actual (`student`) para después introducir la nueva contraseña dos veces. Además, como medida de seguridad, la consola de Linux no muestra el eco local cuando introduces contraseñas, esto es, no verás los caracteres que escribes (no te aparecerán asteriscos). Después de cambiar la contraseña se cerrará la conexión. Podrás conectarte de nuevo, pero ya utilizando la nueva contraseña.

UNIX permite utilizar diferentes interfaces de comandos, aunque la más común es la interfaz `bash`. Esto tiene sus implicaciones a la hora de desarrollar tareas *batch*, es decir, cuando programamos para la propia interfaz de comandos a través de *scripts*.

> Un *script* es un fichero de texto que contiene comandos.

Dentro de la interfaz de comandos puedes utilizar la tecla `Tab` para autocompletar comandos (y nombres de ficheros) y las flechas de arriba y abajo para acceder al histórico de comandos introducidos en la consola.

2.1. Directorio de trabajo

Una vez abierta la interfaz de comandos se muestra el *prompt*, que es el texto que aparece al principio de la línea y que, por defecto, suele indicar el directorio de trabajo actual. Todos los usuarios tienen asignado un directorio de trabajo por defecto denominado `home`, que aparece representado en el *prompt* a través del carácter ~. El comando `pwd`[1] sirve para obtener el directorio actual de trabajo:

```
$> pwd
/home/student
```

El concepto de directorio de trabajo es de vital importancia, pues sobre él se realizarán por defecto todas las operaciones de acceso a ficheros, por ejemplo, crear un fichero de texto, eliminar un fichero, etc., cuando no se especifiquen rutas absolutas en los nombres de ficheros.

Al igual que otros sistemas de ficheros, en UNIX los ficheros se agrupan en directorios organizados de forma jerárquica[2]. A diferencia de los sistemas Windows, en UNIX no existen letras de unidad como `C:` o `D:`. En su lugar, todos los directorios cuelgan del directorio raíz, referenciado por `/`. Para cambiar el directorio de trabajo se utiliza el comando `cd`.

> - Cambia el directorio de trabajo al directorio raíz.
>
> ```
> $> cd /
> ```
>
> - Baja un nivel en la jerarquía de directorios al directorio `/etc`.
>
> ```
> $> cd etc
> ```
>
> - Vete al directorio `/usr/include` especificando una ruta absoluta.
>
> ```
> $> cd /usr/include
> ```
>
> - Retrocede al directorio anterior.
>
> ```
> $> cd -
> ```
>
> - Sube un nivel en la jerarquía de directorios. Ten cuidado, hay un espacio entre el nombre del comando y los dos puntos.
>
> ```
> $> cd ..
> $> pwd
> /
> ```
>
> - Vete al directorio `home`, es decir, al directorio de trabajo por defecto cuando arrancas la interfaz de comandos. Esto puede hacerse de varias formas.
>
> ```
> $> cd
> $> cd ~
> ```

```
$> cd $HOME
```

Hay dos formas de especificar una ruta a través de la jerarquía de directorios: de forma absoluta o de forma relativa. Se puede especificar una ruta absoluta indicando el conjunto de directorios a través del que hay que bajar desde el raíz hasta llegar al fichero deseado. Por ejemplo, el siguiente comando utiliza la orden `cat`, que muestra el contenido de un fichero, especificando su ruta absoluta y, por lo tanto, funcionará independientemente del directorio donde esté situado el usuario.

```
$> cat /etc/hostname
2ac
```

También pueden utilizarse los directorios especiales `.` y `..` que referencian al directorio actual y al padre respectivamente para especificar una ruta relativa, es decir, a partir del directorio de trabajo actual. Observa que una ruta relativa nunca comienza por el directorio raíz `/` al contrario de lo que ocurría con una ruta absoluta.

```
$> cd
$> pwd
/home/student
$> cat ../../etc/hostname
2ac
$> cd /etc
$> cat hostname
2ac
$> cd ..
$> cat ./etc/hostname
2ac
$> cat etc/hostname
2ac
```

El contenido de un directorio puede listarse con el comando `ls`.

```
$> ls
bin   cdrom  etc   initrd.img  media  opt   rofs  sbin     srv  tmp  var
boot  dev    home  lib         mnt    proc  root  selinux  sys  usr
vmlinuz
```

La gran mayoría de sistemas UNIX organizan sus sistemas de ficheros de forma parecida, de tal forma que presentan directorios colgando del raíz con misiones específicas:

- `/home`. Es el directorio donde se ubican los directorios de los usuarios. Típicamente existe un directorio por usuario.
- `/bin`. Contiene comandos que pueden invocarse, tales como el comando `pwd`.
- `/dev`. En él se ubican los ficheros especiales que permiten acceder a los dispositivos del computador y periféricos.
- `/etc`. Contiene la mayor parte de los ficheros para la configuración del sistema y de los programas instalados.
- `/proc`. Contiene ficheros con información del sistema.

2.2. Editando ficheros

Para editar ficheros de texto a través de la consola podemos utilizar editores como `nano`.

Vamos a editar un fichero de texto utilizando el editor en modo consola. Ten cuidado con el directorio de trabajo, pues es donde se crea el fichero resultante.

- Abre el editor indicándole que vamos a editar un nuevo fichero de nombre `1-1texto.txt`. La opción `-c` hace que el editor muestre los números de línea en la parte inferior.

```
$> cd
$> nano -c 1-1texto.txt
```

- Escribe algo en el fichero.
- Guarda el fichero. Para ello utiliza la combinación `Ctrl`+`o` y pulsa `ENTER` para confirmar el nombre del fichero.
- Sal del editor utilizando la combinación `Ctrl`+`x`. Como se ve en la figura, este editor muestra también en la parte inferior una ayuda con el significado de diferentes combinaciones de teclas, donde `^` indica la tecla `Ctrl`.

Como cualquier otro comando, se le puede indicar al editor un nombre de fichero incluyendo una ruta relativa o absoluta. En el ejemplo anterior se especificó una ruta relativa, es decir, se le indicó al editor que creara el fichero en el directorio de trabajo actual, pero también puede especificarse una ruta absoluta con el fichero.

- Abre el fichero recién creado especificando su ruta absoluta.

```
$> nano -c /home/student/1-1texto.txt
```

- Sal del editor.

2.3. Permisos

Como ocurre en otros sistemas operativos multiusuario, UNIX etiqueta cada entrada en el sistema de ficheros (fichero, directorio o enlace simbólico) con una serie de permisos. En concreto, existen tres tipos de permisos, representados por distintos caracteres cuando se hace un listado de un directorio: permiso de lectura (`r`), de escritura (`w`) y de ejecución (`x`). Además, estos permisos se asignan en tres niveles distintos: al usuario propietario del fichero (directorio o enlace simbólico), al grupo propietario y al resto de usuarios del sistema.

Veámoslo con un ejemplo que utiliza la opción `-l` del comando `ls` para mostrar, además del nombre de los ficheros, sus atributos (permisos, propietario, etc.):

```
$> ls -l
total 12
drwxr-xr-x 2 student student 40 2011-06-10 10:42 directorio
-rw-rw-r-- 1 student student 43 2011-06-10 10:07 1-1texto.txt
-rw-r--r-- 1 student root   16 2011-06-10 10:40 otro.txt
-rwxr-xr-x 1 student student 18 2011-06-10 10:39 script
```

La información que se muestra en el listado está organizada en columnas con el siguiente significado:

Figura 1. Significado de los permisos en UNIX

La primera entrada del listado se corresponde con un directorio, de ahí el carácter `d`, el resto de entradas son ficheros regulares, de ahí que muestren el carácter `-` en el tipo de fichero. Los enlaces simbólicos muestran el carácter `l`.

El usuario `student` es el propietario de las cuatro entradas que aparecen en el listado anterior. Además, este usuario tendrá permisos de lectura y escritura sobre todos los ficheros, mientras que tendrá además permisos de ejecución sobre el directorio y el fichero `script`.

Los permisos de un fichero pueden cambiarse utilizando el comando `chmod`. Por ejemplo, se puede eliminar el permiso de escritura sobre el fichero `1-1texto.txt` de la siguiente forma:

```
$> chmod -w 1-1texto.txt
$> ls -l 1-1texto.txt
-r--r--r-- 1 student student 43 2011-06-10 10:07 1-1texto.txt
```

También pueden modificarse todos los permisos considerando cada tripleta como un número binario. De esta forma, se especifica un número formado por 3 bits, cada uno de los cuales equivale a la representación en binario de los permisos `rwx`, donde habría un `1` para cada permiso activo. Así, si queremos asignar los siguientes permisos:

- Usuario propietario: todos los permisos ⇒ `rwx` ⇒ `111b` ⇒ `7d`.
- Grupo propietario: lectura y escritura ⇒ `rw-` ⇒ `110b` ⇒ `6d`.
- Resto de usuarios: solo lectura ⇒ `r--` ⇒ `100b` ⇒ `4d`.

```
$> chmod 764 1-1texto.txt
$> ls -l 1-1texto.txt
-rwxrw-r-- 1 student student 43 2011-06-10 10:07 1-1texto.txt
```

En ocasiones, igual que ocurre en otros sistemas operativos, es necesario tener los privilegios de administrador para realizar una tarea o ejecutar un comando. El usuario administrador en sistemas UNIX es habitualmente el usuario `root` (o superusuario). Sin embargo, cuando entramos en sesión rara vez lo hacemos como usuario administrador.

> - Comprueba con qué usuario estás dentro del sistema. Utiliza el comando `whoami`.
>
> ```
> $> whoami
> student
> ```

Los sistemas UNIX ofrecen la posibilidad de ejecutar cualquier comando o programa con los privilegios del usuario *root* utilizando el comando `sudo`. Para ello el usuario o alguno de los grupos a los que pertenece debe aparecer en el fichero `/etc/sudoers`. Por ejemplo, para actualizar el sistema o instalar nuevos paquetes puede utilizarse el comando `apt-get`. Aunque este comando puede ser iniciado por cualquier usuario, durante su ejecución se requieren permisos de *root*. Vamos a instalar en nuestra máquina las últimas actualizaciones disponibles.

> Se utiliza el término paquete en UNIX para denominar el software que puede instalarse (programas, bibliotecas, etc.).
>
> - Prueba a actualizar la lista de paquetes instalables invocando el comando `apt-get`. Verás que ocurre un error, pues no tienes los permisos adecuados.

```
$> apt-get update
```

- Invoca ahora el comando con permisos de superusuario.

```
$> sudo apt-get update
```

- Actualiza el sistema. El siguiente comando actualiza todos los paquetes para los que exista una versión más moderna.

```
$> sudo apt-get upgrade
```

Recuerda el uso del comando `sudo`, ya que se utilizará a menudo a lo largo del curso.

2.4. Operaciones básicas con ficheros

Hasta ahora hemos visto cómo crear y editar ficheros de texto, listar el contenido de los directorios del sistema o cambiar el directorio actual de trabajo. Ahora veremos operaciones básicas como copiar, mover, renombrar o eliminar ficheros.

- Crea un directorio de nombre `dir` que cuelgue de tu directorio `home`. Para ello debes utilizar el comando `mkdir`, que como el resto de comandos puede recibir una ruta relativa o absoluta.

```
$> mkdir ~/dir
```

- Haz una copia del fichero `1-1texto.txt` dentro del directorio recién creado que tenga el nombre `1-1copia1.txt`. Para ello debes utilizar el comando `cp`. Observa el ejemplo de invocación del comando e intenta entender qué significan los parámetros que se le pasan. Si tienes alguna duda pregúntale al profesor.

```
$> cd ~/dir
$> cp ../1-1texto.txt ./1-1copia1.txt
```

- Renombra el fichero `1-1copia1.txt` a `1-1copia2.txt` al tiempo que lo mueves a tu directorio `home` utilizando el comando `mv`. Este comando sirve tanto para mover como para renombrar ficheros.

```
$> mv 1-1copia1.txt ../1-1copia2.txt
```

- Elimina el fichero `1-1copia2.txt` utilizando el comando `rm`. Mucho cuidado con este comando, pues no solicita confirmación cuando se borran los ficheros.

```
$> cd ..
$> rm 1-1copia2.txt
```

- Elimina el directorio `dir` que creamos al principio con el comando `rmdir`. Este comando requiere que el directorio esté vacío.

```
$> rmdir dir
```

2.5. Ayuda en línea

GNU/Linux es un sistema operativo eminentemente orientado a usuarios desarrolladores de aplicaciones en C. De hecho, las distribuciones suelen incluir el compilador y el entorno de desarrollo, entre otras ayudas a la programación. Una de ellas es la ayuda en línea, que cubre aspectos de programación en C, así como comandos básicos de la interfaz de comandos.

La ayuda en línea de GNU/Linux y, en general, de los sistemas operativos UNIX se ofrece a través de manuales organizados en categorías. Cada una de estas categorías se numera desde el 1 hasta el 8 y comprenden comandos generales de línea de comandos (categoría 1) o funciones de biblioteca invocables desde un programa escrito en el lenguaje C (categoría 3).

Puedes acceder a la ayuda de cualquier comando utilizando la orden `man` seguida del nombre del comando.

- Consulta el manual acerca del comando `ls`. Puedes salir del manual pulsando `q`.

```
$> man ls
```

- También puedes especificar directamente la categoría donde consultar el manual. Consulta ahora el manual de la función `printf` de C. Esta función sirve para mostrar información en pantalla tal como se verá más adelante.

```
$> man 3 printf
```

En el manual se muestra información útil sobre cómo invocar la función y qué ficheros de cabecera hay que incluir para poder invocarla.

- Sal del manual.

La indicación explícita de la categoría a consultar resulta útil en el caso anterior, pues también existe un comando de la línea de comandos de nombre `printf`. Si no se hubiese especificado la categoría, se mostraría la ayuda del comando de la línea de comandos en lugar de la función de C. Puedes probar a invocar el comando anterior sin indicar la categoría y verás el resultado.

> Existe también un comando de la línea de comandos llamado `printf`.

La ayuda se puede consultar también a través de buscadores de internet, que habitualmente indexan los manuales. Generalmente, el resultado de consultar la ayuda desde la línea de comandos o desde un buscador lleva a un resultado equivalente.

2.6. Deteniendo el sistema

Como con cualquier otro sistema operativo, al acabar nuestro trabajo debemos detener el sistema.

- Apaga la máquina virtual usando el siguiente comando.

  ```
  $> sudo poweroff
  ```

  > Siempre debes utilizar esta comando para apagar la máquina virtual. Nunca lo hagas a través de la opción *Apagar máquina virtual* de VirtualBox, pues equivaldría a desconectar la máquina de la alimentación eléctrica. Sí podrías hacerlo con la opción *Enviar señal de apagado*.

- Vuelve a arrancar la máquina virtual.

  > Puede utilizarse el comando `reboot` para reiniciar la máquina, esto es, apagarla y volver a arrancarla.

-

3. Control de versiones

Cuando se trabaja con código fuente (y en general con cualquier tipo de documento que sufre una cierta evolución) se antoja muy recomendable el uso de los sistemas de control de versiones. En los casos de proyectos software de envergadura, resulta entonces imprescindible.

Estos sistemas permiten llevar registro de todos los cambios que se producen en el código (quién, qué y cuándo) de forma automática. A lo largo de esta asignatura utilizaremos *git* como sistema de control de versiones, si bien un flujo de trabajo mínimo. Concretamente, utilizaremos un repositorio[3] *git* en la nube como sistema de *backup* de los ficheros que vayamos generando.

- Configura *git* con tu nombre de usuario y tu correo electrónico. Estos serán los datos que aparecerán en el historial de cambios.

```
$> git config --global user.name "<usuario>"
$> git config --global user.email "<correo>"
```

> No pongas los símbolos "<" y ">": se utilizan para indicar que no debes escribir esa pala las comillas.

- Regístrate en Bitbucket, que es un servicio de repositorios *git* gratuito. Puedes utilizar tu *UO* como nombre de usuario.
- Crea un nuevo repositorio pinchando en el símbolo `+` dentro del menú y eligiendo la opción **Repositorios › Crear repositorio**. Dale como nombre `2ac-plX-uo<xxxxxx>`, dejando el resto de opciones por defecto. Sustituye `plX` por el identificador de tu grupo de laboratorio.
- Dale a tu profesor de prácticas acceso al repositorio de escritura en la sección de **Settings › User and group access**. El profesor te indicará su nombre de usuario.
- Pincha en el símbolo `+` dentro del menú y elige la opción **Clonar**. Copia el texto que aparece en la ventana emergente (es un comando).
- Desde la interfaz de comandos de Linux colócate en tu directorio `home` y ejecuta el comando que acabas de copiar para clonar el repositorio de Bitbucket en tu máquina. Debería ser parecido al siguiente, donde `<usuario>` representa tu nombre de usuario en Bitbucket.

```
$> git clone https://<usuario>@bitbucket.org/<usuario>/2ac-plX-
uo<xxxxxx>.git
```

- Cambia al directorio `2ac-uo<xxxxxx>` donde se ha copiado el repositorio.

> De aquí en adelante, crearás siempre los ficheros de las prácticas dentro de este dire

-

El flujo de trabajo que seguiremos siempre será:

1. Al comienzo de cada sesión de prácticas traer del repositorio de Bitbucket los cambios que se hubiesen realizado con anterioridad.

> No ejecutes estas órdenes todavía. Esto es sólo una explicación.

```
2. $> git pull
```

3. Crear o modificar los ficheros que desarrollemos.
4. Añadir los ficheros o los cambios que hayamos realizado que queremos que estén bajo el control de versiones (de los que queremos *backup*) con la siguiente orden, donde `<fichero>` es el nombre del fichero añadido o modificado.

```
$> git add <fichero>
```

5. Confirmar los cambios realizados, de forma que se registren en el sistema de control de versiones. Se solicitará un mensaje en el que indicarás una breve descripción de los cambios realizados. Normalmente realizaremos varios *commits* en cada sesión de prácticas para crear sucesivas versiones en el repositorio.

```
$> git commit
```

> Se abrirá un editor para que puedas introducir la descripción de los cambios realizados. Guarda el fichero para confirmar el *commit*.

6. Cuando acabemos de trabajar, enviar los cambios hacia el repositorio de Bitbucket.

```
$> git push
```

Resultaría conveniente crear un subdirectorio para cada sesión de prácticas dentro del directorio con el respositorio *git*.

A modo de ejemplo, vamos a añadir un fichero con la descripción del contenido del repositorio que aparecerá en la interfaz gráfica de Bitbucket.

> - Crea un fichero de texto de nombre `README.md` y añade el siguiente texto.

```
# General #

Repositorio para la asignatura Arquitectura de Computadores.
```

> - Una vez guardado el fichero, puedes comprobar cómo git detecta que existe un nuevo fichero que no está bajo el control de versiones.

```
$> git status
```

> - Indícale a git los cambios que se han producido (se dice añadir los cambios al índice). Esto le indica a git qué cambios debe incluir en la siguiente versión.

```
$> git add README.md
```

> - Comprueba de nuevo el estado del repositorio. Ahora debería aparecer que el nuevo fichero está listo para incluirse en la siguiente versión.

```
$> git status
```

> - Confirma los cambios indicando una descripción de los mismos.

```
$> git commit
```

> > Debes proporcionar una descripción de los cambios de forma obligatoria. Recuerda gua reflejar esta descripción.

> - Envía los cambios al respositorio remoto (en Bitbucket).

```
$> git push --set-upstream origin master
```

> Una vez que ejecutes este comando por primera vez, ya no es necesario indicar la opción `--set-upstream origin master` en posteriores llamadas; basta con llamar al comando `git push`.

- Tras esto, puedes comprobar cómo la interfaz de Bitbucket te muestra el cambio que se ha producido y te permite inspeccionar el contenido del respositorio.
- También puedes comprobar el histórico de cambios en tu repositorio local utilizando el siguiente comando.

```
$> git log
```

Puedes encontrar más información del funcionamiento de git en este libro online.

4. Transferencia de ficheros

Un problema que se plantea cuando se utilizan máquinas virtuales es cómo transferir ficheros entre la máquina cliente en la que iniciamos sesión local (anfitrión) y la máquina virtual (invitado).

Vamos a utilizar un programa que hace uso de una conexión SSH para transferir ficheros entre las dos máquinas. Se trata de WinSCP, que es un programa portable que no requiere instalación.

> - Si no está ya instalado en tu equipo anfitrión, descarga la versión portable del WinSCP desde su página de descarga.
> - Ejecuta el programa y establece una conexión SFTP con la máquina remota (misma dirección y puerto que con PuTTY). Fíjate que este procedimiento es el mismo independientemente de que la máquina remota sea virtual o física.
> - Verás que la interfaz de WinSCP te muestra dos árboles de directorios: el local (a la izquierda) y el remoto (a la derecha). Prueba a copiar algún fichero hacia la máquina remota y a la local sin más que arrastrar los ficheros entre ambas ventanas.

---

1. Ten cuidado al introducir comandos porque la interfaz de comandos distingue entre mayúsculas y minúsculas.

2. En Windows es común denominar carpetas a los directorios.

3. Un repositorio es un contenedor con el historial de cambios de los ficheros a él asociados.

Objetivos de la sesión

En esta sesión se introduce a la programación en C. Los principales objetivos de esta sesión son:

- Introducir los conceptos básicos del lenguaje C.
- Conocer el funcionamiento de los punteros en C.

Para conseguir estos objetivos se realizarán pequeños programas escritos en lenguaje C.

Conocimientos y materiales necesarios

Para poder realizar esta sesión, el alumno debe:

- Disponer de una máquina, física o virtual, con un sistema operativo Ubuntu Server 16.04.6 LTS.
- Durante la sesión se plantearán una serie de preguntas que puedes responder en el correspondiente cuestionario en el campus virtual. Puedes abrir el cuestionario en otra pestaña del navegador pinchando en el enlace mientras mantienes pulsada la tecla `Ctrl`.

1. Programación en C

El lenguaje de programación que se utilizará a lo largo de la asignatura será el lenguaje C. El lenguaje C es el lenguaje de programación de sistemas por excelencia y se encuentra íntimamente relacionado con los sistemas operativos UNIX. En esta sección repasaremos algunos conceptos sobre C y veremos algunos nuevos como los punteros.

> - Arranca la máquina virtual y entra en sesión.
> - Cambia al directorio de tu repositorio y sincronízalo con el repositorio de Bitbucket para actualizar los cambios que se hubiesen realizado con anterioridad.
>
>   ```
>   $> git pull
>   ```
>
> - Crea un subdirectorio de nombre `sesion1-1` en el subdirectorio que contiene el repositorio de *git* `2ac`.
>
>   ```
>   $> mkdir sesion1-1
>   ```
>
> - Entra en el nuevo directorio.

Deberás crear todos los programas que desarrolles a partir de ahora en el subdirectorio que acabas de crear.

1.1. Primer programa

Vamos a editar nuestro primer programa en C y ejecutarlo para ilustrar cuál es el procedimiento que debe seguirse en el desarrollo de programas.

> - Edita el fichero `1-1hello.c` para que tenga el contenido que se muestra a continuación:

```c
#include <stdio.h>

int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

En esta asignatura se adopta el convenio de programar en inglés. Cualquier graduado en informática debe ser capaz de leer y escribir código escrito en inglés, lo que le permitirá integrarse en proyectos internacionales.

Para generar el ejecutable es necesario realizar dos pasos: compilar el fuente y enlazar el ejecutable.

- Compilación: consiste en convertir los ficheros fuentes en lenguaje C a su equivalente código objeto (código máquina). La compilación se realiza invocando al compilador pasándole la opción `-c` junto con los ficheros fuente a compilar. En este caso:

```
$> gcc -c 1-1hello.c
$> ls *.o
1-1hello.o
```

- Enlazado: consiste en la generación del ejecutable a partir de los códigos objeto y las bibliotecas necesarias. Para realizar el enlazado se invoca al mismo compilador, que este caso actúa de enlazador, pasándole los ficheros con el código objeto. Opcionalmente, se le puede indicar el nombre del ejecutable deseado utilizando la opción `-o`. En otro caso, el compilador siempre generará un ejecutable de nombre `a.out`. Habría que indicarle también al compilador las bibliotecas con las que enlazar, pero en este caso no es necesario, pues no se necesitan bibliotecas adicionales. Por defecto el compilador enlaza el ejecutable con la biblioteca estándar de C, que contiene las funciones definidas por el estándar tales como `printf`.

```
$> gcc 1-1hello.o -o 1-1hello
$> ls -l 1-1hello
-rwxr-xr-x 1 student student 7155 2011-06-10 16:26 1-1hello
```

Como ves, se genera un fichero ejecutable de nombre `1-1hello`. Puedes invocar el ejecutable indicando la ruta en la que se encuentra, ya que por defecto el directorio de trabajo no está dentro de los directorios que consulta la interfaz de comandos para la ejecución de comandos y programas.

- Invoca el programa especificando únicamente su nombre. Verás que te aparece un mensaje indicativo de que no se encontró el comando.

```
$> 1-1hello
1-1hello: orden no encontrada
```

- Invoca el programa especificando ahora la ruta donde se encuentra, en este caso en el directorio actual. De esta forma, se le indica a la interfaz de comandos dónde debe buscar el comando.

```
$> ./1-1hello
Hello, World!
```

- Borra los ficheros `1-1hello.o` y `1-1hello`. Usa el comando `rm`.

Recuerda especificar la ruta cuando ejecutes programas creados durante las prácticas.

Por último, vamos a incorporar el fichero recién creado al repositorio *git*.

- Añade el fichero que acabas de crear al control de versiones.

```
$> git add 1-1hello.c
```

- Confirma los cambios.

```
$> git commit
```

1.2. Invocando funciones de biblioteca

Cuando un programador desarrolla programas en C, y en general en cualquier lenguaje de programación, utiliza funciones que se repiten entre programa y programa. Tal es el caso de las funciones que muestran información en pantalla, solicitan memoria de forma dinámica, trabajan con cadenas, etc. No tiene sentido que el programador tenga que escribir estas funciones una y otra vez, por lo que el lenguaje define un mecanismo para reutilizar estas funciones.

Estas funciones comunes pueden implementarse en ficheros separados que serán enlazados con el programa a desarrollar en la fase de enlazado. Estos ficheros pueden ser ficheros objetos como hemos visto o bibliotecas de funciones. No obstante, el compilador necesita conocer estas funciones que el programa utiliza pero que no define, para lo cual se utilizan ficheros de cabecera. Estos ficheros contienen declaraciones de funciones, es decir sus prototipos, que únicamente le indican al compilador cómo se llaman las funciones, cuántos parámetros reciben y de qué tipos.

En la fase de enlazado hay que especificarle al enlazador cuáles son los ficheros objetos o bibliotecas donde se encuentran estas funciones. La figura siguiente ilustra todo este proceso.

Figura 1. Ciclo de desarrollo de programas en C

Vamos a verlo con un ejemplo. Crearemos una función `circleArea` que devuelva el área de un círculo a partir de su radio. Esta función sería reutilizable por más programas, por lo que se definirá en un fichero distinto a donde definimos el programa principal.

- Edita el fichero `1-1circle.c` para que tenga el siguiente contenido:

```
1  #include "1-1circle.h"
2
3  double circleArea(double radius)
4  {
5      const double PI = 3.1415;
6      return PI * radius * radius;
7  }
```

- Edita el fichero de cabecera `1-1circle.h` donde se definirá el prototipo de la función `circleArea`.

```
1  // Always use this statement in header files
2  #pragma once
3
4  // Function prototype
5  double circleArea(double radius);
```

La línea `#pragma once` hace que el compilador añada a la compilación el código del fichero de cabecera una sola vez. Debe hacerse siempre para evitar errores de compilación por duplicidad de identificadores al hacer referencia al fichero de cabecera desde dos o más ficheros fuente.

- Edita el fichero `1-1program.c` con el programa principal que hace uso de la función círculo:

```
 1  #include <stdio.h>
 2
 3  // Include the header file
 4  #include "1-1circle.h"
 5
 6  int main()
 7  {
 8      double radius = 3.0;
 9      double area = circleArea(radius);
10      printf("Circle area (radius=%f) is %f\n", radius, area);
11      return 0;
12  }
```

- Compila ambos ficheros para obtener los ficheros objeto.

```
$> gcc -c 1-1circle.c 1-1program.c
```

- Enlaza el programa indicando el fichero objeto donde está la función `circleArea`.

```
$> gcc 1-1circle.o 1-1program.o -o 1-1program
```

- Ejecuta el programa resultante e incorpora los tres ficheros fuente a tu repositorio (`git add` y `git commit`).

El estándar que describe el lenguaje C define adicionalmente una biblioteca de funciones que deben incluir todos los compiladores denominada la biblioteca estándar de C. Esta biblioteca incluye el código de las funciones definidas en el estándar de C.

Un ejemplo de función incluida en la biblioteca estándar de C es la función `printf` antes citada. Puedes consultar el manual de la función para ver cómo se invoca y qué fichero de cabecera es necesario incluir.

- Abre el manual de la función `printf`.

```
$> man 3 printf
```

- ¿Qué fichero de cabecera es necesario incluir? Responde en el cuestionario: pregunta 1.

El enlazador por defecto enlaza todos los programas con la biblioteca estándar de C, por lo que solo debes preocuparte de incluir el fichero de cabecera correspondiente.

Sin embargo, hay algunas funciones que se definen en otras bibliotecas. Tal es el caso de la función `sqrt`, que calcula la raíz cuadrada de un número real.

- Consulta el manual de la función `sqrt`.

```
$> man 3 sqrt
```

- ¿Qué fichero de cabecera es necesario incluir? Responde en el cuestionario: pregunta 2.
- ¿Es necesario indicarle alguna opción al enlazador? Responde en el cuestionario: pregunta 3.
- Edita el fichero `1-1root.c`.

```
1   #include <stdio.h>
2   // TODO: Add the required #include directive
3
4   int main()
5   {
6       double num = 9.0;
7       double root = sqrt(num);
8       printf("The square root of %f is %f\n", num, root);
9       return 0;
10  }
```

- Compila el programa.
- Enlaza el programa, verás que te aparece un error pues la función `sqrt` no está definida. Recuerda que solo has incluido el fichero de cabecera, que contiene únicamente el prototipo de la función, no su código.

```
$> gcc 1-1root.o -o 1-1root
1-1root.o: In function 'main':
1-1root.c:(.text+0x2d): undefined reference to 'sqrt'
collect2: ld returned 1 exit status
```

- Enlaza ahora el programa incluyendo en la línea de comandos la opción que te indicaba la página del manual de `sqrt`.
- Ejecuta el programa e incorpora el fuente al repositorio *git*.

Es de vital importancia que sepas diferenciar los errores de compilación de los errores de enlazado.

1.3. Salida por pantalla

Para mostrar información por pantalla utilizaremos la función `printf`, que se encuentra declarada en el fichero de cabecera `stdio.h`. Para imprimir por pantalla una cadena de texto se podría utilizar la siguiente sentencia:

```
printf("This string is shown on the screen");
```

Esta función recibe un número variable de parámetros, de los cuales el primero es una cadena de formato que puede contener:

- Texto. Este texto se mostrará en la pantalla tal cual.
- Secuencias de escape. Comienzan por el carácter `\` y van seguidas por otro carácter. Las secuencias `\n` y `\t` representan un salto de línea y una tabulación respectivamente.
- Especificadores de conversión. Se utilizan para mostrar el valor de variables. Comienzan por el carácter `%`:
  - `%c`: carácter (tipo `char`).

- `%d`: entero decimal con signo (tipo `int` o `short`).
- `%f`: real (tipo `float` o `double`).
- `%s`: cadena de caracteres (tipo `char *`).
- `%x`: entero en formato hexadecimal.
- `%p`: puntero.

El resto de parámetros de la función son las variables a mostrar que deben aparecer en el mismo orden que aparecen los especificadores de conversión.

- Edita el fichero `1-1output.c` con el contenido que se muestra a continuación.

```
 1   #include <stdio.h>
 2
 3   int main()
 4   {
 5       int i = 3;
 6       float f = 2.4;
 7       char * s = "Content";
 8
 9       // TODO: Print the content of i
10
11       // TODO: Print the content of f
12
13       // TODO: Print the content of the string
14
15
16       return 0;
17   }
```

- Completa las líneas comentadas que faltan de tal forma que al ejecutar el programa lo que se muestre por pantalla sea lo siguiente.

```
i content: 3
f content: 2.400000
s content: Content
```

- Compila y enlaza el programa.
- Ejecuta el programa.

```
$> ./1-1output
```

- Incorpora el fichero fuente `1-1output.c` al repositorio *git*.

1.4. Punteros

Los punteros y su aritmética es con mucho la parte más complicada del lenguaje C, pero también la que le da la capacidad de manejar la memoria a un nivel muy bajo, básica para programación de sistemas. Un puntero es una variable que guarda una dirección de memoria, o lo que es lo mismo, que apunta a un dato en memoria, de ahí su nombre. Para declarar un puntero es necesario indicar el tipo de datos al que apunta junto con el carácter `*`. Algunas definiciones de punteros son las siguientes.

```
char * pChar;  // Refers to a piece of data in memory 1-byte wide
int * pInt;    // Refers to an int in memory (usually 4-bytes wide)
double * pDouble1, * pDouble2;  // Refer to float numbers with double
                                // precision (8-bytes wide)
```

Un puntero definido de esta forma no puede ser directamente utilizado, ya que apunta a una zona de memoria desconocida. En lugar de ello, hay que hacer que el puntero apunte a una zona de memoria válida para poder leer y escribir en las posiciones de memoria a las que apunta. Esto puede hacerse de dos formas: haciendo que el puntero apunte a la zona de memoria de una variable del programa (utilizando el operador `&` para obtener la dirección de una variable), o haciendo que el puntero apunte a una zona de memoria reservada de forma dinámica, habitualmente en el montículo o *heap*.

> El montículo es una zona de memoria del programa que se reserva de forma dinámica.

```
1   int main()
2   {
3       int x;
4       int * pInt;
5       char * pChar;
6
7       x = 3;
8       pInt = &x;
9       pChar = (char *)malloc(20 * sizeof(char));
10      free(pChar);
11      return 0;
12  }
```

Observa en el ejemplo anterior cómo se reserva memoria de forma dinámica en el montículo con la función `malloc`. El operador `sizeof` devuelve el tamaño en bytes de un tipo. Además, es necesario hacer una conversión (*casting*) al asignar la dirección al puntero, ya que la función retorna `void *`, que quiere decir «puntero a un tipo indeterminado».

A diferencia de lenguajes que incorporan recolector de basura como Java, en C es necesario liberar la memoria reservada de forma explícita con la función `free` cuando ya no se use.

Para utilizar los punteros se usa el operador indirección `*`. Cuando este operador precede al nombre de una variable puntero, significa que se hace referencia al valor de la memoria a la que el puntero apunta. Si a continuación del código anterior se añade la siguiente sentencia justo antes del `return`, la salida por pantalla sería 3 3.

```
printf("%d %d", x, *pInt);
```

Un uso muy común de los punteros es en el paso de parámetros a una función. En C los parámetros se pasan por valor[1]. Esto quiere decir que si se utiliza una variable como parámetro de una función, se copia su valor a una nueva variable dentro de la función. Si se cambia este valor dentro de la función, el valor de la variable que está fuera de la función no cambiará.

Aunque una función siempre recibe los parámetros por valor (recibe una copia del valor de la variable), en la práctica se puede conseguir el paso por referencia si se le pasa la dirección de la variable, es decir, un puntero. En este último caso la función recibe una copia del puntero, con lo que es posible modificar los datos a los que apunta el puntero usando el operador de indirección pero no la dirección (el puntero en sí).

El siguiente código muestra un ejemplo de paso de una variable a un procedimiento por valor.

```
1   #include <stdio.h>
2
3   void f(double d)
4   {
5       d = 4;
6   }
7
8   int main(int argc, char* argv[])
```

```
 9   {
10       double d;
11       d = 3;
12
13       f(d);
14       printf("%f\n", d);
15       return 0;
16   }
```

- Genera el programa `1-1pointer1.c` a partir del listado anterior y comprueba cuál es el valor mostrado por pantalla.
- Ahora modifica el programa, llamándolo `1-1pointer2.c`, para que la función `f` en lugar de recibir un `double` reciba un puntero a un `double`. Además, tendrás que modificar la asignación para modificar el contenido de la dirección apuntada por el puntero (utiliza el operador de indirección). Por último, deberás modificar la llamada a la función para que reciba la dirección de la variable `d`. Comprueba que el programa imprime ahora el valor `4`. Si no es así, consulta a tu profesor.
- Incorpora ambos ficheros fuente al repositorio *git*.

1.5. Estructuras

Las estructuras son agrupaciones de tipos básicos en memoria, de tal forma que se ubican en memoria varios elementos de tipos básicos uno a continuación de otro. El siguiente programa muestra cómo definir y utilizar una estructura denominada `Person` que incluye una cadena de texto, un entero y un real de precisión doble.

- Edita el fichero `1-1person.c` con el contenido que se muestra a continuación.

```
 1   #include <stdio.h>
 2   #include <string.h>
 3
 4   struct _Person
 5   {
 6       char name[30];
 7       int heightcm;
 8       double weightkg;
 9   };
10   // This abbreviates the type name
11   typedef struct _Person Person;
12
13   int main(int argc, char* argv[])
14   {
15       Person Peter;
16       strcpy(Peter.name, "Peter");
17       Peter.heightcm = 175;
18
19       // TODO: Assign the weight
20
21       // TODO: Show the information of the Peter data
22   structure on the screen
23
24       return 0;
     }
```

- Completa las líneas comentadas que faltan de tal forma que al ejecutar el programa se muestre por pantalla lo siguiente.

```
Peter's height: 175 cm; Peter's weight: 78.7 kg
```

- Genera el ejecutable y ejecútalo.
- Incorpora el fichero al repositorio *git*.

También es posible crear punteros a estructuras. Volviendo a la estructura `Person` que creamos anteriormente, podemos definir un nuevo tipo puntero a dicha estructura.

```
1   // Type name abbreviations
2   typedef struct _Person Person;
3   typedef Person* PPerson;
4
5   int main(int argc, char* argv[])
6   {
7       Person Javier;
8       PPerson pJavier;
9
10      // Memory location of variable Javier is assigned to the pointer
11      pJavier = &Javier;
12      Javier.heightcm = 180;
13      Javier.weightkg = 84.0;
14      pJavier->weightkg = 83.2;
15      return 0;
16  }
```

Fíjate cómo para acceder a los campos de una estructura a través del puntero se debe hacer uso del operador `->`. Por ejemplo, `pJavier->weightkg` es equivalente a `(*pJavier).weightkg`, es decir, acceder a donde apunta `pJavier` (que es una estructura de tipo `Person`) y luego a su campo `weightkg`.

- Modifica el fichero `1-1person.c` para añadir el tipo puntero a persona `PPerson` tal como se indicó anteriormente.
- Incorpora los cambios sobre el fichero al repositorio *git*.

1.6. Conversiones de tipos

El lenguaje C permite conversiones entre diferentes tipos básicos. Las conversiones pueden realizarse de forma implícita o explícita.

En el primer caso, no es necesaria ninguna instrucción por parte del programador, sino que es el propio lenguaje el que define las conversiones más adecuadas a la hora de evaluar expresiones. Estas conversiones generan avisos en algunos compiladores en el momento de la compilación si implican pérdida de información.

En cambio, las conversiones explícitas se producen a instancia del programador. Se utilizan para convertir entre tipos que no tienen conversión (simplemente se le insta al compilador a interpretar el valor como el nuevo tipo) y para evitar los avisos del compilador cuando se producen conversiones con pérdida de información.

Vamos a verlo con un ejemplo.

- Edita el fichero `1-1implicit.c` con el contenido que se muestra a continuación.

```c
#include <stdio.h>

int main() {
    double div = 3.75;
    int num = 6;
    int result = num / div;
    printf("Variable result is %d, but the division was %f\n", result, num
/ div);
    return 0;
}
```

- Compila y ejecuta el programa. Verás que se compila sin mostrar ningún aviso, pero el resultado de la división no coincide con el mostrado.

> En las últimas versiones de gcc no se muestran avisos cuando se producen conversiones información. Para mostrar este tipo de avisos puede utilizarse la opción `-Wconversic`

- Compila de nuevo el programa pero indicando la orden `-Wconversion`.
- Verás que ahora aparece un aviso indicando que el valor puede variar al hacer la conversión.
- Intenta comprender qué conversiones se producen al evaluar la expresión del ejemplo. Si tienes dudas pregúntale a tu profesor.
- Para evitar el aviso, escribe justo delante de la expresión entre paréntesis el tipo al que se va a convertir para realizar un *casting* (conversión explícita). De esta forma se le indica al compilador que sabemos que se produce esa conversión a pesar de perder información.
- Compila de nuevo con la opción `-Wconversion` para comprobar que el aviso ya no se produce.
- Incorpora los ficheros `1-1implicit.c` y `1-1explicit.c` al repositorio *git*.

La mayor potencia de la conversión de tipos en C viene cuando se combina con punteros. Esto permite interpretar posiciones de la memoria como si se tratasen de cualquier tipo de datos.

Vamos a verlo con un ejemplo que nos va a permitir comprobar la *endianness* de la máquina (orden en el que se almancenan los bytes en memoria para un dato de más de un byte).

- Edita el fichero `1-1explicit.c` con el contenido que se muestra a continuación.

```c
#include <stdio.h>

int main() {
    int x = 1;  // 32 bits

    const char *p = (const char*)&x;

    if (*p) {
        printf("Little-endian\n");
    }
    else {
        printf("Big-endian\n");
    }

    return 0;
}
```

- Fíjate cómo el puntero se hace apuntar a la dirección más baja del entero, lo que permite interpretar este byte como un carácter. Recuerda que un entero se representa con 32 bits (4 bytes).
- Si tienes dudas pregúntale a tu profesor.

Archivos de la práctica

Debes guardar todos los ficheros sobre los que has trabajado durante la práctica. Si has incorporado los cambios realizados a tu repositorio, basta con sincronizar tu repositorio con el almacenado en Bitbucket.

- Sincroniza tu repositorio en Bitbucket con el repositorio local. El siguiente comando sube a Bitbucket los cambios que has incorporado a tu repositorio local.

```
$> git push
```

- Comprueba en Bitbucket que están todos los ficheros.

Ejercicios

Se proponen los siguientes ejercicios adicionales:

- Haz un programa de nombre `1-1add.c` que defina una función `add` que reciba por parámetro un vector de enteros y su tamaño. Dicha función debe devolver como resultado la suma de todos los elementos del vector. La función `main` de dicho programa será la siguiente.

```
 1  #include <stdio.h>
 2
 3  #define NUM_ELEMENTS 7
 4
 5  int main()
 6  {
 7      int vector[NUM_ELEMENTS] = { 2, 5, -2, 9, 12, -4, 3 };
 8      int result;
 9
10      result = add(vector, NUM_ELEMENTS);
11      printf("The addition is: %d\n", result);
12      return 0;
13  }
```

- Incorpora `1-1add.c` al repositorio *git*.
- Haz un programa de nombre `1-1copy.c` que copie cadenas de texto a través de la función `copy` que tendrá el siguiente prototipo.

```
int copy(char * source, char * destination, unsigned int lengthDestination);
```

  En el primer parámetro recibe la cadena a copiar. Recuerda que las cadenas en C terminan con `0`. El segundo parámetro es un puntero a la zona de memoria donde copiar la cadena, mientras que el tercero indica el tamaño de esta zona, para evitar que la función escriba fuera de esta zona.

  Incorpora `1-1copy.c` al repositorio *git*.

Apéndice A: Automatizando la generación del ejecutable

Los sistemas UNIX suelen incluir una herramienta, denominada `make`, que permite automatizar la generación de ejecutables. Para ello, utiliza un fichero que habitualmente se llama `Makefile` y que contiene una serie de reglas que indican cómo debe generarse el ejecutable teniendo en cuenta qué ficheros deben regenerarse cada vez que cambien otros. Cuando un fichero A debe regenerarse cuando cambie un fichero B, se dice que B es una dependencia de A.

La gran ventaja de la herramienta `make` es que automáticamente comprueba si los ficheros que se especifican como dependencias han sido modificados, y genera solo los ficheros que dependan de dichos ficheros modificados, con lo que se ahorra tiempo al generar los programas.

Las reglas de un fichero `Makefile` asocian un objetivo con las dependencias y los comandos que son necesarios ejecutar para, a partir de las dependencias, obtener el objetivo. Son de la forma:

```
objetivo : dependencias
<Tabulador> comandos
```

Antes de indicar el comando que crea el objetivo a partir de la lista de dependencias hay que utilizar un carácter de tabulación. A continuación se muestra un ejemplo de `Makefile` que puede utilizarse para compilar y enlazar el programa `1-1program` visto anteriormente.

```
1   # Rule for generating the main program
2   1-1program : 1-1program.o 1-1circle.o
3       gcc 1-1program.o 1-1circle.o -o 1-1program
4
5   # Compile source files
6   1-1program.o : 1-1program.c 1-1circle.h
7       gcc -c 1-1program.c
8
9   1-1circle.o : 1-1circle.c 1-1circle.h
1       gcc -c 1-1circle.c
0
1   # Clean object files
1   clean :
1       rm -f 1-1program.o 1-1circle.o
2
1
3
1
4
```

- Edita el fichero `Makefile` en el mismo directorio que los ficheros `1-1program.c`, `1-1circle.c` y `1-1circle.h` de forma que contega el listado mostrado.

  > Ten cuidado al copiar y pegar en la consola el listado del fichero, pues deberás su tabulaciones.

- Para que veas cómo compila make, borra todos los ficheros objeto con `rm *.o` y borra también `1-1program`.

- Genera el ejecutable invocando al comando `make`. Puedes hacerlo indicando el nombre del ejecutable a generar o bien sin parámetros, ya que la regla que genera el ejecutable es la primera del fichero `Makefile`.

```
$> make
gcc -c 1-1program.c
gcc -c 1-1circle.c
gcc 1-1program.o 1-1circle.o -o 1-1program
```

- Intenta volver a generar el ejecutable. El comando `make` comprobará que las dependencias del objetivo `1-1program` no han cambiado, por lo que inferirá que el ejecutable está actualizado.

```
$> make
make: '1-1program' está actualizado.
```

- Elimina los ficheros temporales generados invocando la regla `clean` del `Makefile`.

```
$> make clean
rm -f 1-1program.o 1-1circle.o
```

- Incorpora el fichero `Makefile` al repositorio *git*.

Puedes utilizar este fichero `Makefile` como patrón para generar el resto correspondientes a los programas que desarrollarás a lo largo de las prácticas. Debes tener cuidado de modificar apropiadamente los nombres de fichero que aparecen en el `Makefile` para adecuarlos al programa a generar.

---

1. En C++ las funciones pueden recibir referencias a variables (por ejemplo, el tipo de un parámetro puede ser `int&`, es decir, una referencia a entero). Sin embargo, C no tiene referencias.

Objetivos de la sesión

En esta sesión se introduce al alumno en los conceptos sobre medición del tiempo de ejecución de programas en GNU/Linux con el objeto de analizar el rendimiento del sistema. Para ello se utilizan las características que proporciona el sistema operativo para acceder a relojes de altas prestaciones del sistema. Además de medir el tiempo de ejecución de varios programas utilizados como benchmarks sintéticos, se comparará el rendimiento de varios computadores basándose en un benchmark de aplicación.

Conocimientos y materiales necesarios

Para obtener el máximo aprovechamiento de esta sesión, el alumno debe:

- Tener instalada y correctamente configurada la máquina virtual con el sistema operativo GNU/Linux modificado sobre el que se trabajará a lo largo del curso.
- Ser capaz de compilar y ejecutar programas escritos en C a través de una terminal del sistema operativo Linux, como se estudió en la sesión anterior.
- Durante la sesión se plantearán una serie de preguntas que puedes responder en el correspondiente cuestionario en el campus virtual. Puedes abrir el cuestionario en otra pestaña del navegador pinchando en el enlace mientras mantienes pulsada la tecla `Ctrl`.

1. Instrumentación del código fuente en GNU/Linux

Antes de iniciar el análisis de prestaciones de un computador es necesario configurar la máquina virtual para que se puedan tener resultados significativos en algunos apartados de la sesión. También, para que los resultados sean comparables, se debería mantener esa configuración a lo largo de toda la sesión.

> - Antes de iniciar la máquina virtual, selecciónala en VirtualBox y pulsa sobre la opción **Configuración** del menú.
> - Selecciona **Sistema** y pulsa sobre la pestaña *Procesador*.
> - Incrementa el número de procesadores hasta 4.
> - Acepta y arranca la máquina virtual.
> - Cambia al directorio de tu repositorio y sincronízalo con el repositorio de Bitbucket para actualizar los cambios que se hubiesen realizado con anterioridad.
>
> ```
> $> git pull
> ```
>
> - Descarga los ficheros necesarios para la práctica y descomprímelos con estas órdenes:
>
> ```
> $> wget
> http://rigel.atc.uniovi.es/grado/2ac/files/sesion1-
> 2.tar.gz
> $> tar xvfz sesion1-2.tar.gz
> ```
>
> - Añade los ficheros al índice git y confirma los cambios con estas órdenes:

- `$> git add sesion1-2`
  `$> git commit`
- Cámbiate al directorio `sesion1-2`.
- Copia el fichero de la hoja de cálculo `tema1.xls` a Windows para poder modificarlo.

> Puedes utilizar la versión portable del programa WinSCP para intercambiar ficheros entre tus máquinas Windows y Linux.

En ocasiones es necesario medir el tiempo de ejecución de un programa o de un fragmento de programa. Estas mediciones pueden utilizarse con múltiples fines, como comparar el rendimiento de varios sistemas o determinar cuál es la parte de un programa que más tiempo consume para tratar de optimizarla, por ejemplo.

Los sistemas operativos suelen proporcionar funciones de alto nivel que permiten el acceso a contadores de marcas de tiempo del procesador, o TSC (*Time Stamp Counter*). Estos contadores cuentan el número de pulsos de la señal de reloj del procesador. A partir del número de pulsos transcurridos entre dos eventos en el sistema, y conociendo la frecuencia de dicha señal de reloj, se puede determinar con una precisión elevada el tiempo transcurrido entre ambos eventos. Diferentes sistemas operativos proporcionan diferentes métodos para acceder a estos contadores.

La API POSIX que implementa Linux proporciona para los lenguajes C y C++ la siguiente función que permite obtener una marca de tiempo de alguno de los relojes utilizados por el sistema, como el reloj de tiempo real (`CLOCK_REALTIME`).

```
int clock_gettime(clockid_t clk_id, struct timespec * tp);
```

Obtiene el tiempo del reloj especificado por `clk id` y lo almacena en la variable apuntada por `tp`. Retorna 0 en caso de que la función se ejecute correctamente y -1 en caso contrario (en este caso la variable `errno` toma el valor apropiado para identificar el error).

La variable de tipo `timespec` sobre la que se almacena la marca de tiempo es una estructura que contiene la siguiente información, donde `time_t` es un dato de tipo entero que depende de la configuración del sistema.

```
struct timespec {
    time_t   tv_sec;        /* seconds */
    long     tv_nsec;       /* nanoseconds */
};
```

Esta función se puede utilizar para medir con precisión el tiempo transcurrido entre dos eventos del sistema, como muestra el siguiente esqueleto de programa.

```
 1   #include <time.h>
 2
 3   int main()
 4   {
 5       struct timespec tStart, tEnd;
 6       double dElapsedTimeS;
 7
 8       // Start time measurement
 9       clock_gettime(CLOCK_REALTIME, &tStart);
10
11       //  Code to be measured
12       ......
13       ......
14
15       // Finish time measurement
```

```
16      clock_gettime(CLOCK_REALTIME, &tEnd);
17
18      //  Compute the elapsed time, in seconds
19      dElapsedTimeS = (tEnd.tv_sec - tStart.tv_sec);
20      dElapsedTimeS += (tEnd.tv_nsec - tStart.tv_nsec) / 1e+9;
21
22      return 0;
23  }
```

> Fíjate que no se comprueba el valor retornado por `clock_gettime`. Además, al calcular el tiempo transcurrido en segundos se hace una conversión de entero a real de precisión doble que podría ocasionar pérdidas de precisión. Para los valores que se van a medir en esta práctica, ni muy pequeños ni muy grandes, esto no es importante, pero podría serlo en otro contexto.

2. Características del sistema a medir

Antes de comparar el rendimiento entre varios computadores vamos a identificar las características de cada uno de ellos. Esta información se puede obtener bien a través de la información que proporciona el sistema operativo o cualquier otro programa que se ejecute sobre el mismo. A continuación se te propondrán una serie de comandos y utilidades con los que podrás obtener la información necesaria. Alguno de ellos requerirá la instalación previa en el sistema operativo. También será necesario, para obtener más información, ejecutarlos como *root* usando el comando `sudo`.

Debes apuntar la información que vas a obtener en la primera hoja de la hoja de cálculo de resultados `tema1.xls`. Esta hoja está cubierta con la información del equipo que se utiliza como base, S_{ref}, o sistema de referencia, para comparar el rendimiento de otro equipo, *SUT* (*System Under Test*). El equipo *SUT* será la máquina virtual de prácticas sobre la que estás desarrollando esta sesión.

- Empieza obteniendo información del procesador del sistema con los comandos `lscpu` y `lshw`. También puedes consultar el fichero de información del sistema.

```
$> cat /proc/cpuinfo
```

> Recuerda que la ayuda se obtiene pasándole al comando la opción `-h` y que la salida de puede redirigir a un archivo con el operador `>`. También puedes buscar por Internet a pa procesador que te muestre el sistema.

- El modelo de la placa base puedes obtenerlo con el siguiente comando. El resto de información tendrás que buscarla en Internet a partir del modelo.

```
$> sudo dmidecode -t 2
```

- Obtén ahora información de la memoria que se te solicita utilizando alguna de estos comandos: `free`, `top` o `lshw`. También puedes consultar el fichero de información del sistema.

```
$> cat /proc/meminfo
```

- El tipo de memoria puede consultarse con el siguiente comando.

```
$> sudo dmidecode -t 17
```

> Es muy problable que al ejecutar Linux en máquina virtual este comando no te devuelva inform
> caso, deja en blanco la casilla correspondiente en la hoja excel.

- Otros comandos que podrías utilizar son `hwinfo` y `inxi`, si bien tendrías que instalarlos previamente.

> Puedes instalar ambos paquetes con el siguiente comando `sudo apt-get install`

- Intenta obtener, mediante la información que te ofrece el sistema operativo, el nombre del disco en el que está instalado y apúntalo en la hoja de cálculo.
- Completa la información de la hoja de cálculo con la versión del sistema operativo sobre el que estás desarrollando esta sesión de prácticas. Puedes obtener esta información a través del comando del sistema.

```
$> cat /etc/lsb-release
```

Ten en cuenta que vamos a realizar las pruebas de rendimiento sobre la máquina virtual y no sobre una máquina física, por lo que la capa de virtualización añadirá cierta sobrecarga.

3. Análisis del rendimiento mediante carga sintética monolítica

Los benchmarks sintéticos, o programas para medir el rendimiento de un computador basados en carga sintética, son programas que intentan reproducir las operaciones más habituales que se desarrollan en los programas reales. Dependiendo de la carga que se quiera reproducir, estos programas pueden ejecutar compiladores o traductores de lenguajes de programación, cálculos matemáticos complejos, procesamiento de gráficos, etc.

En este apartado vamos a trabajar con benchmarks sintéticos que realizan operaciones sobre números reales con estructuras de datos de diferentes tamaños, lo que, además del procesador del sistema, involucra al sistema de memoria y al sistema de entrada/salida del computador. Estos benchmarks podrían ser útiles para comparar el rendimiento de computadores orientados a realizar cálculos científicos, por ejemplo.

El fichero `bench-fp` contiene el código incompleto de un programa que accede al reloj de altas prestaciones del sistema para medir el tiempo de ejecución de la función `Task`. Esta función se encarga de invocar a la función `DoFloatingPoint` con diferentes parámetros. La función `DoFloatingPoint` reserva espacio en memoria para tres vectores de números reales (mediante la función `malloc`), inicializa dos de ellos con valores aleatorios (mediante la función `rand`) y realiza varias operaciones aritméticas sobre números reales para inicializar los elementos del tercer vector.

A continuación se muestra una breve descripción de las funciones que aparecen en el programa:

```
void * malloc(size_t size);
```

Reserva tantos bytes en memoria como los indicados por `size` y devuelve un puntero a la memoria reservada. La zona de memoria reservada no se inicializa con ningún valor. En caso de que la función no se ejecute correctamente el valor retornado es `NULL`.

```
int rand(void);
```

Retorna un valor entero pseudo-aleatorio en el rango [0,`RAND_MAX`].

```
void srand(unsigned int seed);
```

Coloca el valor `seed` como semilla para la generación de números pseudo-aleatorios que devolverá la función `rand`. Las secuencias de números retornadas por `rand` serán idénticas cuando se invoque a `srand` con el mismo valor de la semilla.

- A la vista del código, ¿cuántas veces se ejecuta la instrucción `pdDest[i] = sqrt(pdDest[i])` cada vez que se ejecuta el programa? Responde en el cuestionario: pregunta 1.
- Completa el código del programa en los puntos en los que aparece `#error *** Complete ***` de acuerdo al esquema mostrado en la primera sección del enunciado de esta sesión.
- Accede, a través de una terminal del sistema operativo, a la carpeta `bench_fp`. Compila y enlaza el programa utilizando el `Makefile` que se acompaña.
- Ejecuta el programa bench_fp. Recuerda que debes indicar la ruta relativa.

  ```
  $> ./bench_fp
  ```

  Si has completado correctamente el fichero fuente, se debería mostrar en la consola el tiempo transcurrido durante la ejecución de la función `Task`.
- Dado que una única medición puede contener un error que no somos capaces de estimar, ejecuta el programa otras cuatro veces. Anota los cinco tiempos medidos en la hoja de cálculo proporcionada para anotar los resultados de la práctica (dentro de la hoja `1-2`, en las columnas `t_1` a `t_5` de la fila `bench_fp`), calcula el valor medio de estos tiempos de ejecución y su desviación típica mediante sendas fórmulas en las columnas `Media t_1 a t_5` y `Desv. Típ. t_1 a t_5`, y el intervalo de confianza para la media con un nivel de confianza del 95% también con una fórmula.
- Si se considera a la función `Task` como la *tarea* a ejecutar por el sistema, ¿cuál es la productividad media del sistema expresada en tareas por minuto? Anota el resultado en la hoja de cálculo en la columna `Productividad (tareas/min)`.

  ¿Cuál es la fórmula que has utilizado en esa celda para realizar el cálculo? Responde en el cuestionario: pregunta 2.
- Calcula la aceleración del *SUT* con respecto al sistema de referencia utilizando el programa `bench_fp` como carga. Anota el resultado en la hoja de cálculo.

  ¿Cuál es la fórmula que has utilizado en esa celda para realizar el cálculo? Responde en el cuestionario: pregunta 3.
- A la vista del programa, y de las características hardware del procesador que has obtenido anteriormente, ¿crees que este benchmark es realmente útil para medir las capacidades del sistema en cuanto a cálculo científico? Para comprobar tu respuesta, ejecuta en otra ventana el comando `top` del sistema. El programa `top` muestra la información del estado del sistema cada `n` segundos. En la parte de arriba, muestra un resumen del sistema y, en las filas inferiores, muestra el detalle proceso a proceso. Si se desea información más frecuentemente, se puede modificar el intervalo de captura de `top` con la siguiente opción:

  ```
  $> top -d s
  ```

donde *s* son los segundos entre actualizaciones. Por defecto, `top` refresca cada 3 segundos.

¿Cuál es el porcentaje de uso del procesador que muestra `top` en la cabecera? Responde en el cuestionario: pregunta 4.

¿Cuál es el porcentaje de uso del procesador que muestra `top` en la línea específica de `bench-fp`? Responde en el cuestionario: pregunta 5.

¿Sabrías explicar la razón de las discrepancias? Si tienes dudas pregúntale a tu profesor.

4. Análisis del rendimiento mediante carga sintética multihilo

Independientemente de otros factores que pueden afectar, como el sistema de memoria, el rendimiento del programa `bench_fp` está limitado por el rendimiento del sistema en la ejecución de un único hilo al ser un programa monohilo. En los sistemas actuales, en los que los procesadores suelen tener varios núcleos (cada encapsulado contiene realmente varios procesadores físicos ensamblados de forma conjunta) y en los que cada núcleo puede dar soporte a la ejecución de varios hilos de forma simultánea (como la tecnología HyperThreading de Intel), para obtener el máximo aprovechamiento del sistema es necesario recurrir a la programación paralela (o ejecutar varias instancias del mismo programa monohilo).

A continuación vamos a estudiar conceptos básicos sobre hilos para poder mejorar nuestro benchmark. Un hilo es la unidad mínima planificable por el sistema operativo. Por defecto, un programa tiene un único hilo de ejecución, por lo que la creación y gestión de hilos dentro de un programa es responsabilidad del programador, o de la biblioteca de hilos que utilice. En este último caso, el estándar POSIX define la API para la creación y gestión de hilos dentro de una aplicación.

Para crear un hilo se debe utilizar la función siguiente, declarada en el fichero de cabecera `pthread.h`.

```
int pthread_create( pthread_t * thread, const pthread_attr_t * attr,
     void * (*start_routine) (void *), void * arg);
```

Crea un hilo y lo pone listo para ejecutar. En el parámetro `thread` se almacena el identificador del hilo creado, mientras que a través del parámetro `attr` se pueden indicar atributos de creación del hilo, aunque normalmente se le pasará `NULL`. El parámetro `start_routine` es un puntero a la función que ejecutará el hilo, que recibirá como paramétro el indicado en `arg`. Si la función tiene éxito retorna 0 y un código de error en caso contrario.

El hilo principal, o cualquier otro hilo, puede esperar a que finalice un hilo para continuar. Para ello, debe utilizar la siguiente función.

```
int pthread_join(pthread_t thread, void ** retval);
```

Espera por la finalización de un hilo. El hilo por el que se espera se indica a través del parámetro `thread`. Se puede recoger el puntero retornado por el hilo al finalizar en el parámetro `retval` o indicar `NULL` para descartarlo. De cualquier forma, ten en cuenta que el puntero retornado por el hilo debe apuntar a una zona de memoria válida aun después de finalizado el hilo, por lo que habitualmente se retornarán punteros apuntando a una zona del montículo (memoria reservada con `malloc`). Si la función tiene éxito retorna 0 y un código de error en caso contrario.

A continuación se muestra un ejemplo muy simple de programa multihilo.

Fichero 1-2thread.c

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void* ThreadProc(void* arg)
{
    const int TIMES = 6;
    int i;

    // Cast
    int n = *((int*)arg);
    for (i = 0; i < TIMES; i++)
    {
        printf("Thread %d, message %d\n", n, i);
        sleep(1);  // Sleep 1 second
    }
    printf("Thread %d finished.\n", n);
    return NULL;
}

int main(int argc, char* argv[])
{
    const int N = 5;
    const int TIMES = 3;
    pthread_t thread[N];
    int i;
    int a[N];

    // Thread creation
    for (i = 0; i < N; i++)
    {
        a[i]=i;
        if (pthread_create(&thread[i], NULL, ThreadProc, &a[i]) != 0)
        {
            fprintf(stderr, "ERROR: Creating thread %d\n", i);
            return EXIT_FAILURE;
        }
    }

    for (i = 0; i < TIMES; i++)
    {
        printf("Main thread, message %d\n", i);
        sleep(1);  // Sleep 1 second
    }

    // Wait till the completion of all threads
    printf("Main thread waiting...\n");
    for (i = 0; i < N; i++)
    {
        pthread_join(thread[i], NULL);
    }
    printf("Main thread finished.\n");
    return EXIT_SUCCESS;
}
```

- Crea un `Makefile` para que el programa `1-2thread.c` se compile y enlace mediante la siguiente orden.

```
$> gcc 1-2thread.c -o 1-2thread -lpthread
```

- Ejecuta el programa varias veces e intenta comprender qué pasa con la salida por pantalla. ¿Puedes explicarlo?

Una vez que hemos estudiado cómo escribir un programa multihilo en C, en la carpeta `bench-fp-mt` se encuentra una versión multihilo creada a partir del programa `bench_fp`. Puedes observar que el programa declara una constante denominada `NUM_THREADS` que se inicializa con el número de hilos que se pretenden ejecutar en el programa (más un hilo que ejecuta la función `main`). Cada uno de estos hilos se crea mediante la función `pthread_create`, que recibe, entre otros parámetros, la dirección de la función que ejecutará el hilo creado. En este caso todos los hilos ejecutarán la función `Task`. A continuación, lo que hace el hilo principal, el que ejecuta la función `main`, es esperar a que los `NUM_THREADS` hilos creados anteriormente finalicen su ejecución antes de finalizar la ejecución de la función `main`. Esta sincronización es necesaria dado que todo programa en C o C++ finaliza cuando se acaba de ejecutar la función `main`. Por lo tanto, sin esta sincronización el programa finalizaría sin que hubieran finalizado los hilos que se crean para ejecutar la función `Task`. Dicha sincronización se realiza mediante la función `pthread_join`.

- A la vista del código, ¿cuántas veces se ejecuta la instrucción `pdDest[i] = sqrt(pdDest[i])` cada vez que se ejecuta el programa? Responde en el cuestionario: pregunta 6.
- Completa el código del programa en los puntos en los que aparece `<<Complete>>` de acuerdo al esquema mostrado en la primera sección del enunciado de esta sesión.
- Compila, enlaza y ejecuta el programa.
- Al igual que con el programa monohilo, ejecuta otras cuatro veces este programa y anota las cinco mediciones de tiempo en la hoja de cálculo. Calcula además el tiempo promedio de la ejecución del programa y su desviación típica mediante sendas fórmulas en las columnas `Media t_1 a t_5` y `Desv. Típ. t_1 a t_5` respectivamente, y el intervalo de confianza para la media con un nivel de confianza del 95% también con una fórmula.
- Si se considera a la función `Task` como la *tarea* a ejecutar por el sistema, ¿cuál es la productividad media del sistema expresada en tareas por minuto? Anota el resultado en la hoja de cálculo en la columna `Productividad (tareas/min)`.

  ¿Cuál es la fórmula que has utilizado en esa celda para realizar el cálculo? Responde en el cuestionario: pregunta 7.

  ¿Es el número de veces que se incrementa la productividad superior, inferior o igual al incremento en el número de núcleos del procesador empleado? Responde en el cuestionario: pregunta 8.
- Calcula la aceleración del *SUT* con respecto al sistema de referencia utilizando el programa `bench_fp_mt` como carga. Anota el resultado en la hoja de cálculo.

  ¿Cuál es la fórmula que has utilizado en esa celda para realizar el cálculo? Responde en el cuestionario: pregunta 9.
- Ejecuta una vez más el programa observando a la vez los resultados que muestra `top`.

  ¿Cuál es el porcentaje de uso del procesador que muestra `top` en la cabecera? Responde en el cuestionario: pregunta 10.

  ¿Cuál es el porcentaje de uso del procesador que muestra `top` en la línea específica de `bench-fp_mt`? Responde en el cuestionario: pregunta 11.

¿Sabrías explicar la razón de las discrepancias?

5. Análisis del rendimiento mediante carga real

Los benchmarks basados en carga real o benchmarks de aplicación consisten en la ejecución de aplicaciones reales para comparar el rendimiento entre diferentes sistemas. Estas aplicaciones deberían representar la carga final que va soportar el sistema para que las comparaciones sean útiles.

En este apartado vamos a trabajar con una aplicación real que se utilizará para comparar el rendimiento entre dos sistemas. La aplicación se llama *Mencoder* y se utiliza para codificar archivos de vídeo.

- Inicia el programa `top`.
- Accede a la carpeta `mencoder` a través de otra terminal del sistema operativo. Ejecuta el script `x264` indicándole al sistema operativo que muestre el tiempo de ejecución del mismo de la siguiente forma.

  ```
  $> time ./x264
  ```

  Ten en cuenta que, para poder ejecutarlo, debes proporcionarle al *script* permisos de ejecución. Tal como se ha visto en una sesión anterior, puedes hacerlo así:

  ```
  $> chmod +x x264
  ```

  Este *script* invoca el programa Mencoder para que reescale un vídeo utilizando el codec x264. El *script* indica en las opciones de invocación de Mencoder que el codec utilice un solo hilo, por lo que `top` debería mostar que solo una CPU tiene un rendimiento del 100%.

  > Por defecto, `top` muestra el uso de CPU para una única CPU, por lo que podrían ap
  > encima del 100%. Debes pulsar `1` mientras `top` está corriendo para mostrar el uso

  Puedes ver el contenido del *script* si ejecutas el siguiente comando.

  ```
  $> more x264
  ```

- Ejecuta el *script* cinco veces y obtén las estadísticas del tiempo de respuesta de la aplicación. Calcula además la productividad, es decir, cuántos vídeos por minuto es capaz de codificar el sistema, y la aceleración del *SUT* con respecto al sistema de referencia.

  ¿Cuál es la fórmula que has utilizado para calcular la productividad? Responde en el cuestionario: pregunta 12.

La implementación del codec x264 es multihilo, por lo que vamos a realizar la misma tarea de reescalar el vídeo, pero en este caso utilizando tantos hilos como procesadores tenga el sistema, con lo que se reparte la tarea entre varios hilos.

- Ejecuta desde la terminal del sistema operativo el siguiente comando en el directorio `mencoder`:

  ```
  $> time ./x264_mt
  ```

Observa que todos los procesadores del sistema deben estar en uso. Ejecuta el *script* cinco veces y obtén las estadísticas del tiempo de respuesta medio de la aplicación. Es posible que el tiempo de usuario sea mayor que el tiempo real. Esto es normal, pues el tiempo real es el que realmente tarda en ejecutarse el programa, mientras que el de usuario es la suma de los tiempos de usuario en todas las CPU en las que se ejecuta. En cualqyuer caso, recuerda que debes usar el tiempo real.

Calcula la productividad del *SUT* y su aceleración con respecto al de referencia.

¿Cuál es la fórmula que has utilizado en este caso para calcular la productividad? Responde en el cuestionario: pregunta 13.

- En la página web de ayuda de Mencoder se afirma, en relación con la implementación multihilo del codec x264, que

  > *[...] increase encoding speed linearly with the number of CPU cores (about 94% per CPU core), with very little quality reduction [...]*
  >
  > — *Mencoder Docs*

  ¿Se cumple esta afirmación en las mediciones que has tomado?

Una vez que tienes en la hoja de cálculo los valores de aceleración del *SUT* en comparación con el sistema de referencia para los programas `bench_fp`, `bench_fp_mt` y `mencoder` en Linux, podemos calcular un ratio de aceleración del *SUT* con respecto al S_{ref} teniendo en cuenta todos estos resultados. Aunque podríamos pensar en la media aritmética del valor de aceleración de todas las aplicaciones, en casos en los que se comparan números normalizados, como ratios de rendimiento, suele ser más apropiado el uso de la media geométrica, que, dados n valores x_1, x_2, …, x_n, se calcula como se muestra a continuación:

$$G=x1\cdot x2\cdot\dots\cdot xn\sqrt[n]{}=\left(\prod i=1nxi\right)1nG=x1\cdot x2\cdot\dots\cdot xnn=\left(\prod i=1nxi\right)1n$$

- Calcula el ratio de aceleración del *SUT* en relación al sistema de referencia utilizando la media geométrica y anota el resultado en la celda `C17` de la hoja de cálculo.

Archivos de la práctica

Añade los archivos `bench_fp.c` y `bench_fp_mt.c` completados y la hoja de cálculo `tema1.xls` a tu repositorio *git* local y realiza una copia en tu repositorio de Bitbucket. Recuerda los pasos a realizar.

- Añadir los ficheros o los cambios que hayamos realizado que queremos que estén bajo el control de versiones (de los que queremos *backup*) con la siguiente orden, donde `<fichero>` es el nombre del fichero añadido o modificado.

  ```
  $> git add <fichero>
  ```

- Confirmar los cambios realizados, de forma que se registren en el sistema de control de versiones. Se solicitará un mensaje en el que indicarás una breve descripción de los cambios realizados.

  ```
  $> git commit
  ```

- Por último, enviar los cambios hacia el repositorio de Bitbucket.

```
$> git push
```

Ejercicios adicionales

Se propone el siguiente ejercicio adicional:

- Modifica la configuración de la máquina virtual y reduce el número de procesadores de 4 a 1. Repite las medidas realizadas con los programas `bench-fp` y `bench-fp-mt`. ¿Se obtienen los mismos resultados? ¿Cuál de los dos bechmarks produce más discrepancias con lo ya calculado? ¿Cuál crees que es la razón?

Objetivos de la sesión

En esta práctica se introduce al alumno en los conceptos de análisis del rendimiento de programas utilizando *profilers*. Se aplicará la ley de Amdahl para guiar el proceso de optimización y para determinar el grado de mejora máximo que se puede conseguir mediante este proceso. Además, se verá cómo influyen las características de la arquitectura en el rendimiento de los programas.

Conocimientos y materiales necesarios

Para poder realizar esta sesión, el alumno debe:

- Disponer de una máquina, física o virtual, con un sistema operativo Ubuntu.
- Disponer en Windows de la hoja de cálculo que comenzaste a rellenar en la sesión anterior. Debería estar en tu repositorio.
- Repasar la ley de Amdahl.
- Durante la sesión se plantearán una serie de preguntas que puedes responder en el correspondiente cuestionario en el campus virtual. Puedes abrir el cuestionario en otra pestaña del navegador pinchando en el enlace mientras mantienes pulsada la tecla `Ctrl`.

1. Introducción al análisis de rendimiento de programas

- Arranca Linux.
- Cambia al directorio de tu repositorio y sincronízalo con el repositorio de Bitbucket para actualizar los cambios que se hubiesen realizado con anterioridad.

```
$> git pull
```

- Descarga los ficheros necesarios para la práctica y descomprímelos con estas órdenes:

```
$> wget
http://rigel.atc.uniovi.es/grado/2ac/files/sesion1-
3.tar.gz
$> tar xvfz sesion1-3.tar.gz
```

- Añade los ficheros al índice git, confirma los cambios y súbelos al servidor con estas órdenes:

```
$> git add sesion1-3
$> git commit
$> git push
```

- Cámbiate al directorio `sesion1-3`.
- Copia el fichero de la hoja de cálculo a Windows para seguir modificándolo.

El rendimiento de los programas es un aspecto muy importante de su funcionamiento. El proceso de optimización intenta mejorar este rendimiento, pero debe ser llevado a cabo de manera racional. Sin embargo, muchas veces se hacen de manera intuitiva «optimizaciones» (modificaciones del código que el programador

cree que harán al programa más eficiente) que en realidad no mejoran el rendimiento y, además, pueden ir en contra de otras características muy importantes del código, como son la claridad, la legibilidad y la mantenibilidad. Una famosa cita de Donald Knuth, uno de los más reconocidos expertos en ciencias de la computación, recalca este problema: «La optimización prematura es la raíz de todos los males».

La optimización prematura es aquella que intenta optimizar antes de conocer qué partes del código tienen importancia real en el tiempo de ejecución del programa. El análisis de la complejidad de los algoritmos es una herramienta muy útil para comparar el rendimiento teórico de distintos algoritmos. Sin embargo, los programas reales tienen una gran cantidad de líneas de código, se ejecutan sobre arquitecturas de computadores muy complejas y pasan por una fase de compilación que puede llevar a cabo optimizaciones que no son obvias en el lenguaje de alto nivel; todo esto hace que el análisis de la complejidad no sea suficiente. Se requiere, además, medir la ejecución del programa para conocer en realidad qué partes interesa optimizar.

La medición se puede llevar a cabo de varias formas. Como se ha hecho en la práctica anterior, se puede instrumentar el código manualmente, es decir, introducir en el código sondas que tomen tiempos y, mediante la diferencia entre el tiempo en dos puntos del código, se puede saber cuánto se ha tardado en ejecutar esa parte del código. Si la marca de tiempo que toman esas sondas está basada en la hora del sistema, se obtiene lo que se conoce como *wall-clock time*. En un sistema multitarea, este tiempo no dependerá sólo de la propia ejecución del código del programa, sino que puede verse afectado por la ejecución de otros programas. Lógicamente, a la hora de optimizar, lo que importa es el tiempo realmente consumido por el programa, por lo que se intentará realizar la medición cuando haya menos elementos que puedan perturbarlo y se repetirá la prueba varias veces para que los errores aleatorios tengan menos importancia.

Este método de medición es muy útil. Una buena práctica que se sigue en muchos programas, en especial en entornos servidores, es incluir de manera habitual sondas en el código para llevar un registro (*log*) que sirve tanto para depurar el programa como para conocer qué partes del código están consumiendo la mayor parte del tiempo. Este registro suele estar activo «en producción», es decir, cuando el programa está siendo utilizado normalmente, lo que permite obtener datos muy valiosos sobre el funcionamiento del sistema en condiciones reales. Sin embargo, estas mediciones suelen ser de grandes fragmentos de código, ya que si se introducen demasiadas sondas, el código de medición ralentiza la aplicación.

Para conocer con precisión cuánto tiempo consume cada parte del código, se han desarrollado herramientas específicas que se utilizan en la fase final del desarrollo y no en producción. Las más importantes son los denominados *profilers*, que obtienen un perfil (*profile*) de la ejecución del programa. Este perfil está compuesto por una serie de datos sobre el programa como, por ejemplo, cuántas veces se ejecuta un fragmento de código (típicamente una función, aunque hay *profilers* que pueden llegar al nivel de línea de código), cuánto tarda de media en cada ejecución, etc. Con estos datos ya se pueden tomar decisiones de optimización adecuadas.

Los *profilers* habitualmente pueden obtener dos tipos de perfiles:

- Perfil plano: indica cuánto tiempo se consume de media en cada función y cuántas veces se llama en total a cada función.
- Grafo de llamadas: indica cuántas veces se llamó cada función desde otra función, lo que permite identificar relaciones entre funciones.

El objetivo de los *profilers* es encontrar «puntos calientes» (*hotspots*), es decir, zonas del programa que consumen mucho tiempo. Siguiendo la Ley de Amdahl, estos son los puntos donde más interesa realizar optimizaciones porque serán los que proporcionen una mayor aceleración.

En esta práctica se va a mostrar el proceso de análisis de rendimiento de programas a través de pequeños programas. Lógicamente, cuanto más complejo es el programa más difícil es este proceso y más útiles resultan las optimizaciones.

2. Análisis del rendimiento de programas en C

2.1. Estudio del programa a analizar

Antes de utilizar un *profiler* se va a realizar una toma de contacto con el código a analizar.

- Abre con un editor el programa `prog1.c`, que forma parte de los ficheros que descargaste al principio de la prática.
- Observa la función `main` situada al final del código. Como verás, esta función declara dos vectores y a continuación llama a una función que los rellena con números aleatorios. Después llama a la función `contarPrimosComunes`, que cuenta los números primos que están en ambos vectores. Finalmente, `main` llama a una función que ordena el `vector`.
- Estudia el código de la función `contarPrimosComunes`. Como verás, realiza un bucle que recorre el primer vector y, para cada uno de sus elementos, comprueba si está en el otro vector y si es un número primo. La función `estaEnVector` simplemente recorre un vector hasta que encuentre el número por el que se le pregunta y, si no lo encuentra, devuelve `FALSE`. La función `esPrimo` recorre todos los números entre 2 y el número que se está analizando comprobando si hay alguna división con resto cero, en cuyo caso el número no será primo. Si todas las divisiones tienen resto distinto de cero, el número será primo.
- Estudia el código de la función `ordenaVector`. Esta función realiza una ordenación siguiendo el método de *selection sort*.
- Compila el programa con esta orden:

```
$> gcc prog1.c -o prog1
```

Esta orden utiliza GCC, el compilador de C desarrollado por GNU, para compilar y enlazar en un solo paso el archivo `prog1.c`. El parámetro `-o prog1` indica que el ejecutable de salida (*output*) se llame `prog1`.

- Ejecuta el programa midiendo cuánto tarda con esta orden:

```
$> time ./prog1
```

- Haz la medición cinco veces y apunta los valores `real` obtenidos en la fila `Prog1 sin optimizar` de la hoja `1-3` de la hoja de cálculo, calcula la media en la celda correspondiente, así como la desviación típica y el intevalo de confianza para la media con un nivel de confianza del 95%.

  ¿Qué debes introducir en la celda `H3` de la hoja `1-3` para calcular la desviación estándar? Responde en el cuestionario: pregunta 1.

  ¿Qué debes introducir en la celda `I3` de la hoja `1-3` para calcular el límite inferior del intervalo con un 95% de confianza? Responde en el cuestionario: pregunta 2.

A continuación, para comprobar lo difícil que es optimizar sin realizar mediciones, vamos a realizar algunas suposiciones y algunos cálculos y luego comprobaremos en qué grado acertamos.

¿Cuál crees que es la función que más tiempo consume del código? Apunta el valor en la celda `B14`. ¿Qué porcentaje del tiempo de ejecución crees que se corresponde con esa función? Apunta el valor en la celda `B15`.

Imagina que consigues optimizar el código de esa función haciendo que tarde la mitad de lo que tarda ahora. ¿Cuál sería la aceleración lograda en esa función? Escribe el valor en la celda `B16`. Responde en el cuestionario: pregunta 3. ¿Cuál sería la aceleración lograda en el programa completo? Escribe la fórmula correspondiente en la celda `B17`, utilizando referencias a otras celdas, no sus valores. Responde en el cuestionario: pregunta 4. ¿Cuál sería la fórmula para obtener el tiempo de ejecución del programa? Escríbela en la celda `B18`. Responde en el cuestionario: pregunta 5.

Para comprobar cuál es la realidad vamos a utilizar un *profiler*.

2.2. *Profiling* con GProf

En primer lugar se va a mostrar el proceso de *profiling* utilizando `gprof`, que es una herramienta libre con licencia GNU. A modo de resumen, el proceso es el siguiente:

- Recompilar el programa a analizar con `-pg`, opción que le indica al compilador que incluya el código del *profiler*.
- Ejecutar el programa. La ejecución generará el perfil, que es un archivo binario de nombre `gmon.out`. Si ya existiese un archivo con ese nombre, será sobreescrito.
- Utilizar el programa `gprof` para mostrar de manera textual la información de `gmon.out`.

En el caso concreto de tu programa, debes hacer lo siguiente:

> - Recompila el programa añadiendo el *profiler* con esta orden:
>
> ```
> $> gcc prog1.c -pg -o prog1
> ```
>
> > Fíjate que se ha añadido `-pg`.
>
> - Ejecuta el programa con esta orden:
>
>   ```
>   $> time ./prog1
>   ```
>
>   No deberías observar un cambio significativo en el tiempo de ejecución con respecto a las ejecuciones anteriores. Si fuese así, eso significaría que el código de instrumentación está cambiando significativamente la ejecución del programa.
> - Muestra los ficheros del directorio para comprobar que se ha generado el fichero `gmon.out`:
>
>   ```
>   $> ls
>   ```
> - Genera el informe de *profiling* con esta orden:
>
>   ```
>   $> gprof ./prog1 > informe.txt
>   ```
>
>   Fíjate que a `gprof` no hay que pasarle el fichero `gmon.out` sino el ejecutable con el que se generó el `gmon.out`. Como `gprof` genera la salida por pantalla, se utiliza la última parte de la orden, `> informe.txt`, para redireccionar la salida de `gprof` al fichero `informe.txt`.
> - Edita el fichero `informe.txt`.

La salida de `gprof` tiene dos partes: el perfil plano y el grafo de llamadas. En el primero verás una línea por cada función. En cada línea, las columnas significan lo siguiente:

- `% time`: tanto por ciento del tiempo que el programa pasa en esa función.
- `cumulative seconds`: número de segundos en los que el programa está en esa función y en todas las que están sobre ella dentro del archivo generado.
- `self seconds`: número de segundos que el programa pasa en esa función. Es la columna que se utiliza para ordenar las funciones.
- `calls`: número de llamadas a la función.
- `self ms/call`: media del número de milisegundos que se emplean en esa función en cada llamada.
- `total ms/call`: media del número de milisegundos que se emplean en cada llamada en esta función y las funciones a las que llama.
- `name`: nombre de la función.

Tras la explicación de cada columna, se encuentra el grafo de llamadas. Este grafo está divido por líneas de guiones en secciones. En cada sección se muestra información de la función que tiene un número entre corchetes en la primera columna. Las funciones de la misma sección que aparecen antes de la función son las que la llaman y las que aparecen después son a las que esa función llama. Por ejemplo:

```
index % time    self  children    called     name

[...]

                0.00    0.32       1/1           main [1]
[3]      2.1    0.00    0.32       1         contarPrimosComunes[...]
                0.27    0.00  100000/100000      estaEnVector[...]
                0.05    0.00  100000/100000      esPrimo(int) [5]
-----------------------------------------------
```

da información de la función `contarPrimosComunes`, que es llamada por `main` y llama a `estaEnVector` y `esPrimo`.

Para la función principal de la sección, las columnas significan lo siguiente:

- `% time` indica el tanto por ciento que consume la función y las funciones a las que llama en el total del programa.
- `self` indica el tiempo total en segundos consumido en la función (sin contar el tiempo consumido en las funciones a las que llama).
- `children` indica el tiempo que consumen las funciones a las que llama la función.
- `called` indica el número total de veces que ha sido llamada la función.

Ahora que has medido con el *profiler* la aplicación, puedes comprobar si tus suposiciones fueron adecuadas:

> - ¿Cuál es la función que consume más tiempo? Escribe el valor en la celda `C14`. Responde en el cuestionario: pregunta 6.
> - ¿Qué porcentaje del tiempo consume? Escribe el valor en la celda `C15`. Responde en el cuestionario: pregunta 7.

El *profiler* nos puede indicar dónde consume tiempo el programa, pero no la forma de solucionarlo. Hay que analizar por qué ocurre eso: ¿se está utilizando un algoritmo muy complejo en esa función que se podría

solucionar utilizando un algoritmo más eficiente? ¿Se está llamando demasiadas veces a esa función y hay que cambiar el algoritmo en la función que provoca esas llamadas? ¿Las instrucciones que ejecuta la función no son las más eficientes? ¿Se podrían utilizar tipos de datos más eficientes?

En este sencillo caso, sabiendo que *selection sort* es uno de los peores algoritmos de ordenación, la primera solución podría ser cambiarlo por *quicksort*, lo que se hace en `prog2.c`. Vamos a comprobar que funciona:

> - Compila y mide cinco veces el tiempo de ejecución de esta nueva versión del programa. Escribe estos valores en la fila `Prog2` optimizado de la hoja de cálculo. Calcula la media en la celda `G4`. ¿Cuál ha sido la aceleración del programa? Escribe la fórmula en la celda `K4`. Responde en el cuestionario: pregunta 8.
> - Para hacer más rápido el programa ahora habría que hacer otro análisis de rendimiento. Compila con `-pg` el programa `prog2`, ejecútalo y obtén con `gprof` el perfil. ¿Cuál es ahora la función que consume la mayor parte del tiempo? Responde en el cuestionario: pregunta 9. ¿Qué porcentaje de tiempo consume? Responde en el cuestionario: pregunta 10.

GProf es una herramienta con una interfaz rudimentaria. Una alternativa en Linux es utilizar KCachegrind, un visualizador que utiliza los datos obtenidos por Callgrind, que es una herramienta que utiliza el sistema de instrumentación de Valgrind. Esta última herramienta tiene como función principal analizar el uso de memoria de los programas pero con Callgrind se convierte en un *profiler*. Para utilizar KCachegrind habría que usar Linux con interfaz gráfica, así que no vamos a verlo aquí.

2.3. Influencia de la optimización del compilador en el rendimiento

Una instrucción de un lenguaje de alto nivel se puede traducir a instrucciones máquina de muchas formas, algunas con un rendimiento mejor que otras. Los compiladores modernos incluyen una fase de optimización que se puede controlar mediante opciones. Activar las optimizaciones tiene algunas desventajas:

- Al tener que hacer más trabajo, el compilador tarda más. En proyectos con una gran cantidad de código esto puede suponer una pérdida de productividad de los programadores si deben esperar demasiado tiempo en cada compilación.
- El compilador, al igual que los seres humanos, puede equivocarse creyendo que ciertas modificaciones harán el código más eficiente.
- Las optimizaciones pueden llegar a realizar cambios tan profundos en el código generado que sea difícil relacionar el código máquina con el código fuente. Esto dificulta la depuración línea a línea del código fuente y hace menos inteligible el resultado de los *profilers*.

Estos motivos pueden aconsejar la no utilización de las optimizaciones en algunos casos, pero en general lo conveniente es activarlas, por lo menos las más básicas para el código final si se encuentran problemas de rendimiento.

Desde el punto de vista del proceso de optimización mediante *profilers* que se está trabajando en esta práctica, hay que tener en cuenta que no merece la pena optimizar cambiando el código fuente cuando el compilador puede lograr la misma optimización de manera transparente para el programador por dos razones: 1) se está perdiendo tiempo del programador en realizar una tarea que pueden hacer automáticamente los compiladores y 2) como ya se ha comentado, las optimizaciones en muchas ocasiones hacen el código fuente menos inteligible. En cualquier caso, hay optimizaciones, sobre todo de alto nivel (por ejemplo, cambiar el algoritmo

usado como se hizo en el ejemplo anterior) que no puede hacer el compilador. La forma correcta de trabajar es medir el código con las optimizaciones del compilador y sólo entonces, si no se logra el rendimiento requerido, se procede al proceso de optimización del código fuente por parte del programador.

Vamos a comprobar a continuación cómo afectan las optimizaciones del compilador. En los ejemplos que se han hecho hasta ahora no se había activado ninguna optimización. Vamos a volver a repetir algunos experimentos activando opciones de optimización para ver las diferencias.

> - Para activar el nivel más básico de optimización, se debe utilizar el parámetro `-O1` (la `O` viene de «Optimización»)[1]. Compila el programa `prog1.c` con esta orden:
>
> ```
> $> gcc -O1 prog1.c -o prog1
> ```
>
> - Ejecuta el programa cinco veces tomando tiempos con esta orden:
>
> ```
> $> time ./prog1
> ```
>
> - Apunta los valores obtenidos en la fila `Prog1 -O1` de la hoja de cálculo y calcula el resto de celdas de esa fila extendiendo la fórmula de la fila inmediatamente superior, pero teniendo cuidado de que la aceleración se calcule sobre el programa 1 sin optimizar. ¿Mejora el tiempo de respuesta con respecto a la versión sin optimizar? Responde en el cuestionario: pregunta 11.
> - Compila ahora con el segundo nivel de optimización mediante esta orden:
>
> ```
> $> gcc -O2 prog1.c -o prog1
> ```
>
> - Ejecuta el programa cinco veces tomando tiempos, apúntalos en la hoja de datos en la fila `Prog1 -O2` y calcula el resto de celdas de esa fila. ¿Mejora el tiempo de respuesta con respecto a `-O1`? Responde en el cuestionario: pregunta 12.
> - Finalmente, repite el proceso con el máximo nivel de optimización:
>
> ```
> $> gcc -O3 prog1.c -o prog1
> ```
>
> - Ejecuta el programa cinco veces tomando tiempos, apúntalos en la hoja de datos en la fila `Prog1 -O3` y calcula el resto de celdas de esa fila. ¿Mejora el tiempo de respuesta respecto a `-O2`? Responde en el cuestionario: pregunta 13.

Como ves, el proceso de optimización no es nada sencillo. Hasta el momento hemos probado estas opciones:

1. Compilar el programa original con `-O1`.
2. Compilar el programa original con `-O2`.
3. Compilar el programa original con `-O3`.
4. Utilizar `prog2`, el programa optimizado a mano.

A la vista de todos los resultados que has obtenido, ¿cuál es la opción más rápida? Responde en el cuestionario: pregunta 14. Pero todavía queda saber qué ocurre si se aplican las optimizaciones a `prog2`. Pruébalo con los tres niveles de optimización. ¿Cuál es el más rápido? Responde en el cuestionario: pregunta 15.

Se podría dar el caso de que el nivel de optimización `-O3` no fuese más rápido que el `-O1`. Estudiar con detalle las razones por las que podría ocurrir esa situación requeriría un análisis detallado del código generado por GCC que escapa del alcance de esta práctica. Sin embargo, y solo para ver los complejos modos en los que influye la arquitectura en el rendimiento, se va exponer una posible explicación: el nivel `-O3` aplica

optimizaciones que, en principio, suponen una mejora en el tiempo de ejecución pero incrementan el tamaño del ejecutable. Una consecuencia de esto es que es posible que código (por ejemplo, un bucle) que antes cabía dentro de la memoria caché del procesador ahora no quepa y haya que ir a buscarlo a un nivel inferior, lo que, como se estudiará en el tema de memoria, supone una penalización muy alta. En los ejercicios adicionales de esta práctica se estudia más este problema.

Ten en cuenta que los resultados que has obtenido en estas prácticas no reflejan necesariamente lo que vas a observar en los programas reales que, en general, serán mucho más complejos. Lo más importante que debes haber aprendido es que el proceso de optimización no es sencillo porque, debido a las diversas capas de abstracción que hay entre el alto nivel y la arquitectura que ejecuta los programas, se producen a veces resultados no intuitivos y, por lo tanto, es fundamental realizar mediciones y emplear herramientas como los *profilers* y las optimizaciones del compilador, pero no confiando ciegamente en ellas.

2.4. Influencia de la optimización del compilador en el *profiler*

Como se ha señalado anteriormente, las optimizaciones del compilador pueden afectar al proceso de *profiling*. Vamos a comprobarlo:

- Compila `prog1` con el máximo nivel de optimizaciones y el código del *profiler*:

```
$> gcc prog1.c -O3 -pg -o prog1
```

- Ejecuta el programa para que se genere `gmon.out`:

```
$> ./prog1
```

- Genera el perfil:

```
$> gprof ./prog1 > informe.txt
```

- Edita el archivo `informe.txt`. ¿Cuántas funciones aparecen? Responde en el cuestionario: pregunta 16. Apunta el valor en la celda `B23`. Esto es porque una de las optimizaciones que ha realizado el compilador consiste en transformar el código de una función en código que se pone directamente donde se llama la función, evitando así realizar un `call`.
- Repite el mismo proceso utilizando el nivel `-O1`. ¿Cuántas funciones aparecen? Responde en el cuestionario: pregunta 17. Apunta el valor en la celda `B24`. Analiza si puedes ver en el perfil todas las funciones importantes.

Archivos de la práctica

- Copia la hoja de cálculo a Linux, añádela al repositorio con `git add Tema1.xls`, confirma los cambios con y `git commit` y súbelos al servidor con `git push`.

Ejercicios

- GCC ofrece un nivel de optimización indicado por el parámetro `-Os` que prioriza generar código optimizado sin incrementar el tamaño del código objeto generado. Para ello, aplica las mismas optimizaciones que en `-O2` excepto aquellas que incrementen el tamaño del código objeto. Prueba `prog1` con esta opción. ¿Obtienes mejor rendimiento que con `-O1`? ¿Y que con `-O2`? Compara el

tamaño de los ejecutables obtenidos con las distintas opciones de optimización. ¿Cuál es el mayor? ¿Y el más pequeño?

- Como se ha comentado, el nivel de optimizaciones elegido hace variar, además del tiempo de ejecución del programa compilado, el tiempo de compilación, es decir, el tiempo que necesita el compilador para hacer su tarea. Para comprobarlo, mide el tiempo de compilación de `prog1` sin optimizaciones y con los tres niveles de optimización. ¿Cuál es la ganancia de compilar sin optimizaciones con respecto a cada nivel de optimización?

- El operador `&&` de C realiza una evaluación perezosa, lo que significa que evalúa las sub-expresiones que forman parte de la expresión en orden secuencial y, en cuanto encuentra una que es falsa, ya no evalúa el resto porque el resultado será falso independientemente del valor de las sub-expresiones que queden por evaluar. Esto es, en principio, una optimización, aunque teniendo en cuenta las complejidades de las arquitecturas actuales, que intentan ejecutar instrucciones por adelantado y si tienen que cancelar su ejecución sufren una penalización, puede no resultar en un rendimiento mejor.

  En el programa `prog2`, el tiempo de ejecución está dominado por la función `contarPrimosComunes`, que tiene una sentencia condicional que llama a dos funciones: `estaEnVector` y `esPrimo`. Usando un *profiler* determina si cambiar el orden de las llamadas dentro de la expresión cambia el número de llamadas a cada función y si cambia el rendimiento del programa.

- A veces los *profilers* dan información incorrecta. Lee el artículo *Evaluating the accuracy of Java profilers*, donde explican cómo utilizaron cuatro *profilers* distintos sobre un mismo código y sólo dos de ellos señalaron la misma función como la que más tiempo consumía, lo que demuestra que al menos dos estaban dando datos incorrectos (y es posible que los cuatro estuvieran equivocados).

---

1. Puedes conocer con detalle las opciones de optimización que ofrece GCC ejecutando la orden `man gcc`.

Comenzado el	lunes, 13 de septiembre de 2021, 19:26
Estado	Finalizado
Finalizado en	lunes, 13 de septiembre de 2021, 19:33
Tiempo empleado	7 minutos 23 segundos

Pregunta **1**
Finalizado

Sin calificar

Marcar pregunta

Enunciado de la pregunta

¿Qué fichero de cabecera es necesario incluir para poder utilizar la función ***printf***?

Respuesta: stdio.h

Retroalimentación

En la parte inicial del manual de ayuda se especifica como:

#include <stdio.h>

La respuesta correcta es: stdio.h

Pregunta **2**
Finalizado

Sin calificar

Marcar pregunta

Enunciado de la pregunta

¿Qué fichero de cabecera es necesario incluir para poder utilizar la función ***sqrt***?

Respuesta: math.h

Retroalimentación

En la parte inicial del manual de ayuda se especifica como:

#include <math.h>

La respuesta correcta es: math.h

Pregunta **3**
Finalizado

Sin calificar

Marcar pregunta

Enunciado de la pregunta

¿Es necesario indicarle alguna opción al enlazador para utilizar la función ***sqrt***?

Respuesta: -lm

Retroalimentación

Aparece tras el texto Link with *-lm*.

La respuesta correcta es: -lm

Comenzado el	lunes, 20 de septiembre de 2021, 18:31
Estado	Finalizado
Finalizado en	sábado, 25 de septiembre de 2021, 10:35
Tiempo empleado	4 días 16 horas

Pregunta **1**
Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

Análisis del rendimiento del computador basado en benchmarks

- ¿cuántas veces se ejecuta la instrucción `pdDest[i] = sqrt(pdDest[i])` cada vez que se ejecuta el programa?

Respuesta: 40000000

Retroalimentación
La respuesta correcta es: 40000000

Pregunta **2**
Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta
¿Qué fórmula has utilizado en la hoja Excel para realizar el cálculo de la productividad?

*Copia todo lo que has escrito en la celda **a partir del signo =***

Respuesta: 60/ ((SUMA(C13:G13)/ 5)

Retroalimentación
Se solicitan las tareas por minuto, luego hay que multiplicar por 60.

La respuesta correcta es: 60/H13

Pregunta **3**
Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Qué fórmula has utilizado en la hoja Excel para realizar el cálculo de la aceleración?

*Copia todo lo que has escrito en la celda **a partir del signo =***

Respuesta: L13/ D4

Retroalimentación

La aceleración se puede calcular a partir del tiempo de respuesta o de las productividades.

La respuesta correcta es: C4/H13

Pregunta **4**
Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Cuál es el procentaje de uso del procesador que muestra `top` en la cabecera en la ejecución de `bench_fp`?

Respuesta: 25%

Retroalimentación

La respuesta correcta es: 25 %

Pregunta **5**
Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Cuál es el procentaje de uso del procesador que muestra `top` en la línea específica de la ejecución de `bench_fp`?

Respuesta: 100%

Retroalimentación

La respuesta correcta es: 100 %

Pregunta **6**
Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta
Análisis del rendimiento del computador basado en benchmarks

- ¿cuántas veces se ejecuta la instrucción `pdDest[i] = sqrt(pdDest[i])` cada vez que se ejecuta el programa?

Respuesta: 400000000

Retroalimentación
La respuesta correcta es: 400000000

Pregunta **7**
Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta
¿Qué fórmula has utilizado en la hoja Excel para realizar el cálculo de la productividad?

*Copia todo lo que has escrito en la celda **a partir del signo =***

Respuesta: 60/ ((SUMA(C14:G14)/ 5)

Retroalimentación
Hay que multiplicar por el número de tareas, que ahora es 10.

La respuesta correcta es: 10 (60/H14)

Pregunta **8**
Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

Análisis del rendimiento del computador basado en benchmarks

- ¿Es el número de veces que se incrementa la productividad superior, inferior o igual al incremento en el número de núcleos del procesador empleado?

Respuesta: Inferior

Retroalimentación

La respuesta correcta es: Inferior

Pregunta **9**

Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Qué fórmula has utilizado en la hoja Excel para realizar el cálculo de la aceleración?

*Copia todo lo que has escrito en la celda **a partir del signo =***

Respuesta: L14/ D5

Retroalimentación

Al usar las productividades, no es necesario tener en cuenta el número de tareas, puesto que va implícito.

La respuesta correcta es: C5/H14

Pregunta **10**

Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Cuál es el procentaje de uso del procesador que muestra `top` en la cabecera en la ejecución de `bench_fp_mt`?

Respuesta: 100%

Retroalimentación

La respuesta correcta es: 100 %

Pregunta **11**

Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Cuál es el procentaje de uso del procesador que muestra `top` en la línea específica de la ejecución de `bench_fp_mt`?

Respuesta: 400%

Retroalimentación

La respuesta correcta es: 400 %

Pregunta **12**

Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Qué fórmula has utilizado en la hoja Excel para realizar el cálculo de la productividad en la ejecución del programa mencoder?

*Copia todo lo que has escrito en la celda **a partir del signo =***

Respuesta: 60/ (SUMA(C15:G15)/ 5)

Retroalimentación

Se solicitan las tareas por minuto, luego hay que multiplicar por 60. En este caso se considera la ejecución del script como la *tarea*.

La respuesta correcta es: 60/H15

Pregunta **13**

Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Qué fórmula has utilizado en la hoja Excel para realizar el cálculo de la productividad en la ejecución del programa mencoder?

*Copia todo lo que has escrito en la celda **a partir del signo =***

Respuesta: 60/ (SUMA(C16:G16)/ 5)

Retroalimentación

Como en el caso anterior, se considera la ejecución del script como la *tarea*. Una única tarea.

La respuesta correcta es: 60/H16

Comenzado el	lunes, 27 de septiembre de 2021, 18:22
Estado	Finalizado
Finalizado en	lunes, 27 de septiembre de 2021, 19:02
Tiempo empleado	40 minutos 41 segundos

Pregunta **1**
Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Qué debes introducir en la celda H3 de la hoja 1-3 para calcular la desviación estándar?

Respuesta: =DESVEST.M(B3:F3)

Retroalimentación

Se puede calcular de varias formas, pero la más sencilla y robusta es utilizar la función de Excel DESVEST.M(). Si has utilizado otro método, comprueba que sale lo mismo que con esta función.

La respuesta correcta es: =DESVEST.M(B3:F3)

Pregunta **2**
Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Qué debes introducir en la celda I3 de la hoja 1-3 para calcular el límite inferior del intervalo con un 95% de confianza?

Respuesta: =G3-2*H3

Retroalimentación

La respuesta correcta es: =G3-INTERVALO.CONFIANZA.T(0.05;H3;5)

Pregunta **3**
Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Cuál sería la aceleración lograda en una función si se reduce su tiempo de ejecución a la mitad?

Respuesta: 2

Retroalimentación

La respuesta correcta es: 2

Pregunta **4**
Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Cuál sería la fórmula que tendrías que introducir en B17 para calcular la aceleración del programa?

Respuesta: =1/ (1-B15+(B15/ B16))

Retroalimentación

La respuesta correcta es: =1/((1-B15)+(B15/B16))

Pregunta **5**
Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Cuál sería la fórmula que tendrías que introducir en B18 para obtener el tiempo de ejecución del programa?

Respuesta: =(1-B15+B15/ B16)*G3

Retroalimentación

La respuesta correcta es: =G3/B17

Pregunta **6**
Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Qué función es la que consume más tiempo según gprof y, por lo tanto, la que debes escribir en la celda C14?

Respuesta: ordenaVector()

Retroalimentación

La respuesta correcta es: ordenarVector

Pregunta **7**
Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Qué porcentaje debes apuntar en la celda C15?

Respuesta: 97.8%

Retroalimentación

La respuesta correcta es: Debería ser más del 90%

Pregunta **8**
Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Qué formula debes escribir en la celda K4 para calcular la acelaración obtenida?

Respuesta: =G3/ G4

Retroalimentación

Tiempo viejo / tiempo nuevo

La respuesta correcta es: =G3/G4

Pregunta **9**
Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Qué función consume la mayor parte del tiempo en prog2?

Respuesta: esPrimo()

Retroalimentación

La respuesta correcta es: estaEnVector

Pregunta **10**

Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Qué porcentaje del tiempo consume la función que consume más en prog2?

Respuesta: 74

Retroalimentación

La respuesta correcta es: Sobre un 75%

Pregunta **11**

Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Mejora la versión compilada con optimizaciones de prog1 a la versión sin optimizaciones?

Seleccione una:

a.

No

b.

Sí

Retroalimentación

Respuesta incorrecta.

La respuesta correcta es: Sí

Pregunta **12**

Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Mejora el tiempo de respuesta de prog1 compilando con -O2 con respecto a la versión compilada con -O1?

Seleccione una:

a.

No

b.

Sí

Retroalimentación

Respuesta incorrecta.

La respuesta correcta es: Sí

Pregunta **13**

Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Mejora el tiempo de respuesta de prog1 compilando con -O3 con respecto a la versión compilada con -O2?

Seleccione una:

a.

No

b.

Sí

Puede que te salga que no

Retroalimentación

Respuesta correcta

La respuesta correcta es: Sí

Pregunta **14**
Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Cuál es la versión más rápida?

Seleccione una:

a.

Utilizar `prog2`, el programa optimizado a mano

b.

Compilar el programa original con `-O2`

c.

Compilar el programa original con `-O1`

d.

Compilar el programa original con `-O3`

Retroalimentación

Respuesta correcta

La respuesta correcta es: Utilizar `prog2`, el programa optimizado a mano

Pregunta **15**
Incorrecta

Puntúa 0,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Cuál es la versión más rápida de prog2?

Seleccione una:

a.

Con -O1

b.

Con -O3

c.

Con -O2

d.

Sin optimizaciones

Retroalimentación

Respuesta incorrecta.

Puede que te salga -O2

La respuesta correcta es: Con -O3

Pregunta **16**

Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Cuántas funciones aparecen en el profile de prog1 cuando se utiliza -O3?

Respuesta: 1

Retroalimentación

La respuesta correcta es: 1

Pregunta **17**

Correcta

Puntúa 1,00 sobre 1,00

Marcar pregunta

Enunciado de la pregunta

¿Cuántas funciones aparecen en el profile de prog1 cuando se utiliza -O1?

Respuesta: 2

Retroalimentación

La respuesta correcta es: 2

Notas Extra:

Clock:
Clock_gettime(CLOCK_REALTIME, &tStart) == -1
Clock_gettime(CLOCK_REALTIME, &tEnd)
dElapsedTimeS = (tEnd.tv_sec - tStart.tv_sec);
dElapsedTimeS += (tEnd.tv_nsec - tStart.tv_nsec) / 1e+9;== -1

Linkers:
#include <math.h> ➔ gcc <programa.c> –o <programa> –lm
#include <pthread.h> ➔ gcc <programa.c> –o <programa> –lpthread

Excel:
Accel(A/B) = Prod(A) / Prod(B) = T(B) / T(A)
Valor Agregado = Media.Geom(Accel1, Accel2, Accel3, … , AccelN)
Amdahl = 1/((1-Fracc.M)+(Fracc.M/Accel))
T' = T0*Fracc.M/Accel

MultiThread
Productividad = 60/<media> * <NumThreads>

Optimizaciones auto:
gcc –O<n> <programa.c> -o <programa>

Gprof:
gcc <programa.c> –pg –o <programa>
./<programa>
gprof ./<programa> > informe.txt

Top:
1 → Media Procesadores / Procesadores individuales
S → Mod tiempo actualización

Chmod:
+x = Permiso ejecución
+r = Permiso lectura
+w = Permiso escritura

Chmod +<letra> <programa>