Práctica 2.1: Introducción a la programación de sistemas

Objetivos

En esta práctica el alumno estudiará el uso básico y convenciones del API de un sistema UNIX y su entorno de desarrollo. En particular se usará las funciones disponibles para la gestión de errores y para obtener información.

Contenidos

Preparación del entorno para la práctica
Gestión de errores
Información del sistema
Información del usuario
Información horaria del sistema

Preparación del entorno para la práctica

La realización de esta práctica únicamente requiere del entorno de desarrollo (compilador, editores y utilidades de depuración). Estas herramientas están disponibles en la máquina física de los puestos del laboratorio.

En la realización de las prácticas se puede usar cualquier editor gráfico o de terminal y el lenguaje C++ (compilador g++). Si fuera necesario compilar varios archivos, se recomienda el uso de alguna herramienta para la compilación de proyectos como `make`. Finalmente, el depurador recomendado en las prácticas es `gdb`.

Como preparar el entorno:

- Abrir el "visual" - donde programas- con nano: **nano redes1.cpp**
- Esto se genera en la carpeta raiz, creo. What ever.
- **Compilar:** Necesitas un objeto de codigo. Se crea usando esto: **g++ -Wall -o Redes1 redes1.cpp**

    - **Y ahora cuando haces ./Redes1** Se debería de compilar el codigo y sacarte los couts que sea.

INCLUDES:

```
#include <stdio.h> //Why amarillo
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
#include <iostream>
#include <string.h>
#include <iomanip>
#include <time.h>
#include <math.h>
```

Gestión de errores

//-----

Usar las funciones disponibles en el API del sistema para gestionar los errores en los siguientes casos. En cada ejercicio añadir las librerías necesarias (`#include`). Se recomienda emplear las llamadas a `perror(3)` y

Ejercicio 1. Añadir el código necesario para gestionar correctamente los errores generados por la llamada a `setuid(2)`. Usando la página de manual, comprobar el propósito de la función y su prototipo.

Hace un return de un -1 con setuid(2) -> Siempre falla.

El perror("errno") te devuelve un errno: Operation not permited.

```
int main() {
   /* Comprobar la ocurrencia de error y notificarlo con la llamada perror(3) */
    setuid(0);
    return 1;
}
```

Ejercicio 2. En el código anterior imprimir el código de error generado por la llamada, tanto en su versión numérica como la cadena asociada usando la función `strerror(3)`.

Lo que sale: Operation not permited.

Sale lo mismo que si pones perror asignando errno a un caracter e imprimiendo por pantalla.

Char a = strerror (errno); y si lo sacas es igual.

Información Horaria del Sistema

Ejercicio 3. La función principal para obtener la hora del sistema es `time(2)`. Escribir un programa que obtenga la hora usando esta función y la muestre en el terminal.

Número: 1.524.066.652

Codigo empleado:

```
int main() {

  /* Comprobar la ocurrencia de error y notificarlo con la llamada perror(3) */

  time_t tiempo = 2;

  std::cout << time(&tiempo)   << std::endl;
```

```
   return 1;
}
```

Ejercicio 4. Modificar el programa anterior para que muestre además la hora en formato *legible,* usando la función `ctime(3)`. ¿Dónde se reserva espacio para el valor de la cadena que devuelve la función? ¿Es necesario liberar el puntero? ¿Se puede usar esa función en un programa multi-thread?

1524070961 Thu Jan 1 01:00:03 1970

Si se pone el número en segundos en vez del 3, sale el dia de hoy

1524071183 Wed Apr 18 19:02:41 2018

Modificación

```
int main( ) {
  /* Comprobar la ocurrencia de error y notificarlo con la llamada perror(3) */
  time_t tiempo = 2;
  time_t tiempoLegible = 3;
   std::cout << time(&tiempo) << "  " << ctime(&tiempoLegible)   << std::endl;
   return 1;
}
```

¿Donde se reserva espacio? En el struct “tm que está en la librería Time.h

¿Es necesario liberar el puntero? No, porque hace referencia a datos estáticos

¿Se puede usar en multithread? No. se usaría ctime_r() para que sea seguro usarlo en un multihilo.

Ejercicio 5. Escribir un programa que mida cuánto tarda un bucle de 10.000 repeticiones en incrementar una variable en una unidad en cada iteración. Utilitzar la llamada `clock_gettime(2)` para medir el tiempo. Usar la función pow(3) para operar con las variables de nanosegundos. **Nota:** Para compilar el programa es necesario enlazar con la librería `librt`, añadir `-lrt` al compilar el programa.

COMPILAR: g++ -lrt -Wall -o Redes1 redes1.cpp

Nos ha quedado esto:

54407760838

Codigo usado:

```
int main() {
            timespec tiempoStruct;
          long  incr = 0;
          clockid_t reloj = 2;
            for(int i = 0; i < 10000; i++){
                    clock_gettime(reloj, &tiempoStruct);
                     incr += tiempoStruct.tv_nsec;
                     incr +=(long)pow(tiempoStruct.tv_sec,3);
          }
   std::cout << incr<< std::endl;
  return 1;
}
```

Ejercicio 6. Escribir un programa que, usando la función `strftime(3)`, imprima el año (p.ej. "`Estamos en el año 1982`") y la hora en la forma: "`Hoy es Lunes, 10:34`".

Nota: Para establecer la configuración regional o *locale* (como idioma o formato de hora) en el programa según la configuración actual, usar `setlocale(LC_ALL, "")`.

```
#include <time.h>
#include <math.h>
#include <locale.h>
#include <string>
```

```
#include <iostream>
int main ()
{
  time_t tiempo;
  struct tm * strTiempo;
  char buffer [80];

  time (&tiempo);
  strTiempo = localtime (&tiempo);

  strftime (buffer,80,"Estamos en el año %Y.",strTiempo);
  std::cout << buffer << std::endl;

  setLocale(LC_ALL, "es_ES.utf8");
  strftime (buffer,80,"Hoy es %A, son las %H:%M%p",strTiempo);
  std::cout << buffer << std::endl;

  return 0;

}
```