Presentación

Esta práctica plantea una serie de actividades con el objetivo que el estudiante pueda aplicar sobre un sistema UNIX algunos de los conceptos introducidos en los primeros módulos de la asignatura.

El estudiante deberá realizar una serie de experimentos y responder a las preguntas planteadas. También deberá escribir un programa en lenguaje C.

Competencias

Transversales:

- Capacidad para adaptarse a las tecnologías y a los futuros entornos actualizando las competencias profesionales

Específicas:

- Capacidad de analizar un problema en el nivel de abstracción adecuado a cada situación y aplicar las habilidades y conocimientos adquiridos para abordarlo y resolverlo
- Capacidad de diseñar y construir aplicaciones informáticas mediante técnicas de desarrollo, integración y reutilización

Enunciado

Para realizar esta práctica os facilitamos el fichero `pr2so.zip` que contiene ficheros fuente. Descompactadlo con la orden `unzip pr2so.zip`. Para compilar un programa, por ejemplo `prog.c`, debéis ejecutar la orden `gcc -o prog prog.c`

1. (5 puntos, 1 punto por apartado) Dados los siguientes códigos, indicad justificadamente cuál es el resultado de ejecutarlos. Indicad cuál es la jerarquía de procesos creada, qué mensajes (y en qué orden) se muestran por los dispositivos de entrada/salida, bloqueos,...

   Se aconseja que en primer término intentéis determinar su comportamiento sin ejecutarlos en terminal. Ejecutadlos para verificar vuestra respuesta. Podeu assumir que ninguna llamada devolverá error.

   (a) Código 1: Contestad para N=3 y después generalizad la respuesta para cualquier valor de N mayor que 0.

```
#include <unistd.h>

#define N 3

int main() {
  int i=0;

  write(1, "Hello\n", 6);
```

```
  while (i<N) {
    fork();
    i++;
  }
  write(1, "Bye\n", 4);
}
```

El proceso inicial escribe `Hello`.

El proceso inicial entra en un bucle en el que la primera iteración crea un proceso hijo que también vuelve a iterar. En la segunda iteración tenemos dos procesos que crean dos nuevos procesos y los cuatro vuelven a iterar En la tercera iteración tenemos cuatro procesos que crean cuatro nuevos procesos y los ocho abandonan el bucle.

Los ocho procesos esriben `Bye`, no podemos saber en qué orden escribirán los mensajes.

La generalització es que el programa muestra `Hello`, crea 2^{N-1} procesos y muestra `Bye` 2^N veces.

(b) Código 2:

```
#include <unistd.h>

#define N 3

int main() {
  int i=0;

  write(1, "Hello\n", 6);
  while (i<N) {
    fork();
    i++;
    execlp("date", "date", NULL);
  }
  write(1, "Bye\n", 4);
}
```

El proceso inicial escribe `Hello`.

El proceso inicial entra en un bucle en el que la primera iteración crea un proceso hijo Los dos procesos hacen un `execlp()` de `date` con lo que se carga un nuevo ejecutable y se inicia su ejecución, con lo que se muestra dos veces la fecha actual. Cuando cada proceso ejecute la llamada `exit()` presente en `date`, el proceso morirá sin volver al código anterior.

Por tanto, no se volverá a iterar ni se mostrará ningún mensaje `Bye`.

(c) Códigos 3a i 3b:

```
#include <unistd.h>
#include <fcntl.h>

#define FILE "out.txt"
```

```c
int main() {
  // Creates an empty file
  close(open(FILE, O_WRONLY|O_TRUNC|O_CREAT, 0600));

  close(1);
  open(FILE, O_WRONLY);

  fork();

  execlp("date", "date", NULL);
}
```

```c
#include <unistd.h>
#include <fcntl.h>

#define FILE "out.txt"

int main() {
  // Creates an empty file
  close(open(FILE, O_WRONLY|O_TRUNC|O_CREAT, 0600));

  fork();

  close(1);
  open(FILE, O_WRONLY);

  execlp("date", "date", NULL);
}
```

La diferencia entre los códigos es que en un caso se abre el fichero de salida antes de invocar la llamada `fork()` y en el otro se abre después. Esto implica que en el primer caso padre e hijo comparten el puntero de lectura/escritura sobre el fichero mientras que en el segundo tienen punteros independientes. Por tanto, en el primer caso, las escrituras que haga el padre en el fichero de salida afectarán al puntero de lectura/escritura que se utilizará en las escrituras que haga el proceso hijo (y viceversa).

Consecuentemente, en el primer caso el fichero de salida tendrá la fecha actual dos veces mientras que en el segundo caso sólo aparecerá una vez.

(d) Código 4:

```
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define N 5

int global1=0, global2=0;

int main() {
  int i, p, st;
  char s[80];

  for(i=0; i<N; i++) {
    p = fork();
    if (p==0) {
      global1 = global1 + 1;
      exit(global1);
    }
    wait(&st);
    global2 = global2 + WEXITSTATUS(st);
  }
  sprintf(s, "%d␣%d\n", global1, global2);
  write(1, s, strlen(s));
}
```

Los procesos creados con `fork()` no comparten memoria. Por lo tanto, las modificaciones que hacen los hijos a `global1` no tendrán efecto en la variable `global1` del proceso padre ni en las de los otros hijos. Por tanto, la variable `global1` de todos los hijos acabarán valiendo 1 y la utilizan como código de finalización.

El proceso padre recoge el código de finalización de todos los hijos (1) y los suma, con lo que la variable `global2` del padre acabará valiendo 5.

El proceso padre acaba escribiendo 0 y 5.

(e) Código 5:

```
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>

int main() {
  int p[2];
  char c;

  pipe(p);
  if (fork() == 0) {
    write(p[1], "abcde", 5);
    exit(0);
  }
  else {
    while (read(p[0], &c, 1) > 0) {
      write(1, &c, 1);
    }
```

```
    wait(NULL);
  }
}
```

El proceso inicial crea una pipe y un proceso hijo. El hijo escribe `abcde` en la pipe y muere. El padre lee el contenido de la pipe y lo escribe en la salida estándar pero no finaliza debido a que se bloquea leyendo de la pipe puesto que, aunque está vacía, todavía existe un proceso que tiene abierta la pipe en modo de escritura (el propio proceso padre).

2. (5 puntos) Escribid un programa en C que, cada `N` segundos, muestre la hora actual. Para mostrar la hora actual debéis ejecutar la orden `date`. Para esperar `N` segundos, **no podéis utilizar ni signals ni rutinas como `sleep`**, debéis crear un proceso hijo que ejecute la orden `sleep N`. Tampoco podéis utilizar rutinas como `system` y es preciso hacer el tratamiento de errores en las llamadas al sistema.

   (a) En primer término, fijad `N` igual a 5 segundos. Se adjunta ejemplo:

```
[enricm@willy dev]$  ./prob
Thu Nov  8 12:39:33 CET 2018
Thu Nov  8 12:39:38 CET 2018
Thu Nov  8 12:39:43 CET 2018
Thu Nov  8 12:39:48 CET 2018
^C
[enricm@willy dev]$
```

   (b) Ahora haced que `N` sea un parámetro que deba especificarse en la línea de órdenes. Se adjunta ejemplo de ejecución:

```
[enricm@willy dev]$  ./prob_par 3
Thu Nov  8 12:40:00 CET 2018
Thu Nov  8 12:40:03 CET 2018
Thu Nov  8 12:40:06 CET 2018
Thu Nov  8 12:40:09 CET 2018
^C
[enricm@willy dev]$
```

- Apartado a)

```c
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>

void
error (char *m)
{
  char s[128];

  sprintf (s, "Error %d %s\n%s\n", errno, m, strerror (errno));
  write (2, s, strlen (s));

  exit (0);
}

void
wait5seconds ()
{
  int pid;

  switch (pid = fork ())
    {
    case -1:
      error ("fork");

    case 0:
      execl ("/bin/sleep", "sleep", "5", NULL);
```

```
      error ("execl");

    default:
      if (wait (NULL) != pid)
        error ("wait");
    }
}

void
printTime ()
{
  int pid;

  switch (pid = fork ())
    {
    case -1:
      error ("fork");

    case 0:
      execl ("/bin/date", "date", NULL);
      error ("execl");

    default:
      if (wait (NULL) != pid)
        error ("wait");
    }
}

int
main (int argc, char *argv[])
{
  while (1)
    {
      printTime ();
      wait5seconds ();
    }
  return 0;
}
```

- Apartado b)

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>

void
error (char *m)
{
  char s[128];

  sprintf (s, "Error %d %s\n%s\n", errno, m, strerror (errno));
  write (2, s, strlen (s));

  exit (0);
}

void
```

```
waitNseconds (char *strN)
{
  int pid;

  switch (pid = fork ())
    {
    case -1:
      error ("fork");

    case 0:
      execl ("/bin/sleep", "sleep", strN, NULL);
      error ("execl");

    default:
      if (wait (NULL) != pid)
        error ("wait");
    }
}

void
printTime ()
{
  int pid;

  switch (pid = fork ())
    {
    case -1:
      error ("fork");

    case 0:
      execl ("/bin/date", "date", NULL);
      error ("execl");

    default:
      if (wait (NULL) != pid)
        error ("wait");
    }
}

int
main (int argc, char *argv[])
{
  if (argc != 2) error("Missing arguments");

  while (1)
    {
      printTime ();
      waitNseconds (argv[1]);
    }
  return 0;
}
```

Recursos

- Módulos 1, 2, 3, 4, 5 y 6 de la asignatura.
- Documento "Introducción a la programación de UNIX" (disponible en el aula) o cualquier otro manual similar.

- Documento "Intérprete de comandos UNIX" (disponible en el aula) o cualquier otro manual similar.
- El aula "Laboratorio de Sistemas Operativos" (podéis plantear vuestras dudas relativas al entorno UNIX, programación,...).

Criterios de evaluación

El peso de cada pregunta está indicado en el enunciado.

Se valorará la justificación de las respuestas presentadas.

Formato y fecha de entrega

Se creará un fichero `zip` que contendrá todos los ficheros de la entrega (un `pdf` con la respuesta a las preguntas y ficheros fuente).

Fecha límite de entrega: 24:00 del 10 de diciembre de 2018.