En esta práctica se repasarán conceptos prácticos sobre procesos, señales y threads. Antes de empezar tendrás que bajarte los códigos del repositorio creado en Bitbucket. Para ello ejecuta la siguiente orden en un terminal, dentro del directorio donde quieras que se bajen los códigos (dentro de ese directorio se creará otro llamado "soprocess" dentro del cual aparecerá un directorio por cada uno de los apartados de esta práctica):

```
git clone https://UMADACTC@bitbucket.org/UMADACTC/soprocess.git
```

Entrega en esta tarea aquellos códigos que se te indiquen en los distintos apartados.

Process and signal I

En esta práctica vamos a demostrar el uso de algunas syscall relacionadas con la gestión de procesos y señales, con el uso de los siguientes programas de ejemplo:

1. Inspecciona el código del fichero **fork1.c**, compílalo y comprueba su funcionamiento. Ejecuta fork1. ¿Cuántos procesos hijos crea el proceso padre? Dibuja el árbol de procesos.

2. Inspecciona el código del fichero **fork2.c**, compílalo y comprueba su funcionamiento. Ejecuta fork2. ¿Cuántos procesos hijos crea el proceso padre? Dibuja el árbol de procesos.

3. Inspecciona el código del fichero **forkls.c**, compílalo y comprueba su funcionamiento. Ejecuta forkls. ¿Quién es el que devuelve su status al proceso que ejecuta forkls?

4. Inspecciona el código del fichero **signal.c**, compílalo y comprueba su funcionamiento. Para explicar su funcionamiento ejecuta los siguientes comandos:
5. ./signal ls
   ./signal ./fork1
   ./signal ./fork2
   ./signal sleep 200 &
   (busca con ps el PID del proceso sleep 200 y haz "kill -9 PID")

6. Inspecciona el código del fichero **signal_ign.c**, compílalo y comprueba su funcionamiento. Para explicar su funcionamiento prueba a pulsar varias veces CTRL-C y CTRL-Z después del primer mensaje.

7. Inspecciona el código del fichero **signal_blq.c**, compílalo y comprueba su funcionamiento. Para explicar su funcionamiento prueba a pulsar varias veces CTRL-C y CTRL-Z después del primer mensaje. ¿Cuál es la diferencia con el programa anterior?

Process and signal II

1. En el fichero **captura_senal.c** hay un pequeño programa ejemplo de la captura de la señal de interrupción (Ctrl+C) que nos permite evitar que nuestro proceso muera debido a la pulsación de esta combinación de teclas. Usando la función signal() se le asigna a la señal que provoca la pulsación de Ctrl+C (SIGINT) una rutina de tratamiento diferente a su comportamiento por defecto (terminar el proceso).
   Inspeccionar el código, compilarlo y comprobar su funcionamiento en el laboratorio.

2. En el fichero **mata_hijo.c** hay un programa que usando la llamada al sistema fork() crea un proceso hijo que queda ejecutándose indefinidamente. Desde el proceso padre se le envía la señal de terminación SIGINT mediante la llamada al sistema para envio de señales kill(). Entonces el padre espera la finalización del proceso hijo (wait) e interpreta el estado de terminación del hijo informando del mismo.
   Inspeccionar el código, compilarlo y comprobar su funcionamiento en el laboratorio.

3. Después de resolver las dudas que os surjan de los códigos anteriores usando el manual en linux (man) y preguntado al profesor, modificar el segundo programa mata_hijo.c para que el proceso hijo creado intercepte la señal SIGINT y en vez de realizar la acción por defecto, deberá:

- imprimir un mensaje informando de las malas intenciones de su padre
- esperar durante 2 segundos (sleep)
- por último, terminar el proceso con código de terminación 10 (exit)
  entrega esta modificación en la tarea correspondiente del campus virtual con el nombre **mata_hijo1.c**

4. En una segunda versión volver a modificar el código anterior para que al recibir la señal SIGINT el proceso hijo realice las siguientes acciones:

- imprimir un mensaje informando de las malas intenciones de su padre
- esperar durante 2 segundos (sleep)
- enviar al proceso padre la señal de terminación SIGINT (para matarlo)
- continuar con la ejecución normal del proceso hijo
  ¿cómo matamos al proceso hijo ahora desde el shell?
  entrega esta modificación en la tarea correspondiente del campus virtual con el nombre **mata_hijo2.c**

5. En **hijo_zombie.c** encontrarás otra version del código anterior que ilustra el estado de zombie. El hijo termina con exit() pero el padre no le hace el wait() con lo que mientras sigue existiendo el proceso padre, el estado del hijo es de zombie, hasta que el proceso init() lo adopta al terminar el padre. Esto puede monitorizarse con top.

Race

En el fichero **race.c** hay un pequeño programa ejemplo en el que dos threads compiten salvajemente por la actualización de una variable compartida. El interfaz pthreads (POSIX threads) es el empleado en este caso. Uno de los threads incrementa la variable, el otro la decrementa. El thread que incrementa muere al actualizar la variable al valor (+N); el thread que decrementa muere cuando la variable alcanza el valor (-N). ¿Quién llegará antes al final?

1. Analiza el comportamiento del código, compílalo, y ponlo a funcionar: gcc -g race.c -lpthread
   Consulta detenidamente el manual de las funciones utilizadas

2. Para valores pequeños de N, ¿quién llega antes al final? Y, ¿para valores muy grandes?

3. ¿Qué diferencias observas en la creación de los threads con respecto a la creación de procesos con fork()?

4. Los threads creados ¿son independientes (detached)?

5. Investiga cuáles son las opciones de ps y top que muestran información sobre cada uno de los threads. Comprueba su comportamiento para valores grandes de N. ¿Cuántos threads existen?

6. Observas algún problema en la manera en que se accede a la variable compartida. ¿Sería ésta la manera de hacerlo con seguridad en la práctica?

7. ¿Tienen algún significado los valores de los contadores que se imprimen al final?

Threads vs Process

Para tener una primera visión de por qué un thread es un 'proceso ligero' puedes medir con time las diferencias en los tiempos de ejecución de estos programas de juguete que lo único que hacen es crear un número configurable de threads o de procesos...

Respawn

En el fichero **respawn.c** hay un ejemplo de código que crea un proceso hijo y ejecuta en él el programa que se le pasa como parámetro, y espera la finalización de dicho programa. Modificar respawn.c para que si el programa que se ha ejecutado (en el proceso hijo) termina por la recepción de una señal (no debe volver a ejecutar el programa si este termina normalmente). Entrega el fichero en esta tarea.