PRÁCTICAS DE SISTEMAS OPERATIVOS II

PRÁCTICA LINUX DE GRUPO

Un día en las carreras (de coches)

1. Enunciado.

En esta práctica vamos a simular, mediante procesos de UNIX, una carrera automovilística.

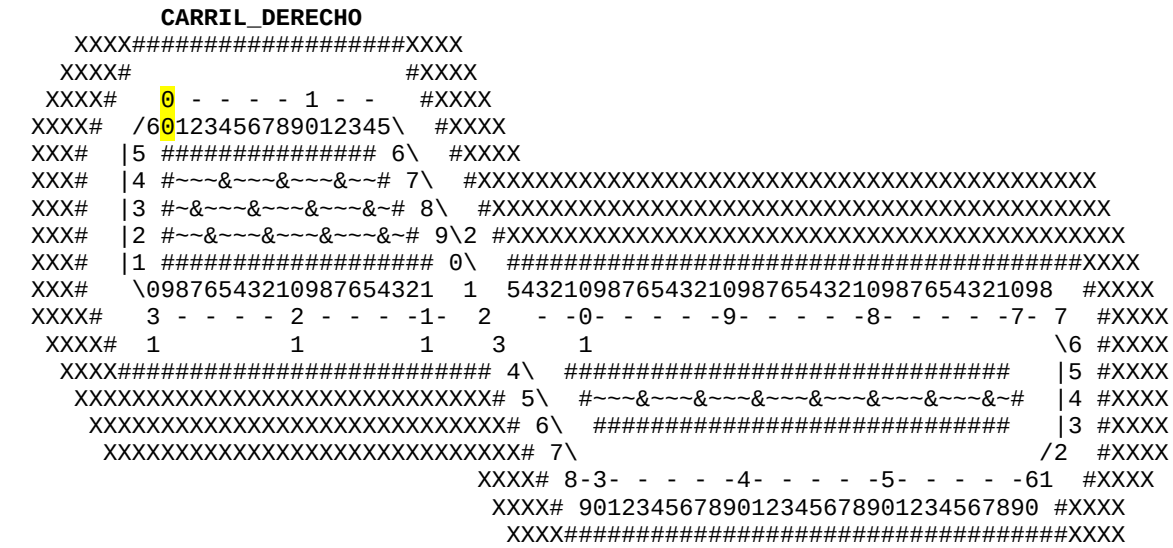
El programa constará de un único fichero fuente, **falonso.c**, cuya adecuada compilación producirá el ejecutable **falonso**. Respetad las mayúsculas/minúsculas de los nombres, si las hubiere.

Para simplificar la realización de la práctica, se os proporciona una biblioteca estática de funciones **libfalonso.a** que debéis enlazar con vuestro módulo objeto para generar el ejecutable. Gracias a ella, muchas de las funciones no las tendréis que programar sino que os bastará con incluir la biblioteca cuando compiléis el programa. La línea de compilación del programa podría ser:

```
gcc falonso.c libfalonso.a -o falonso
```

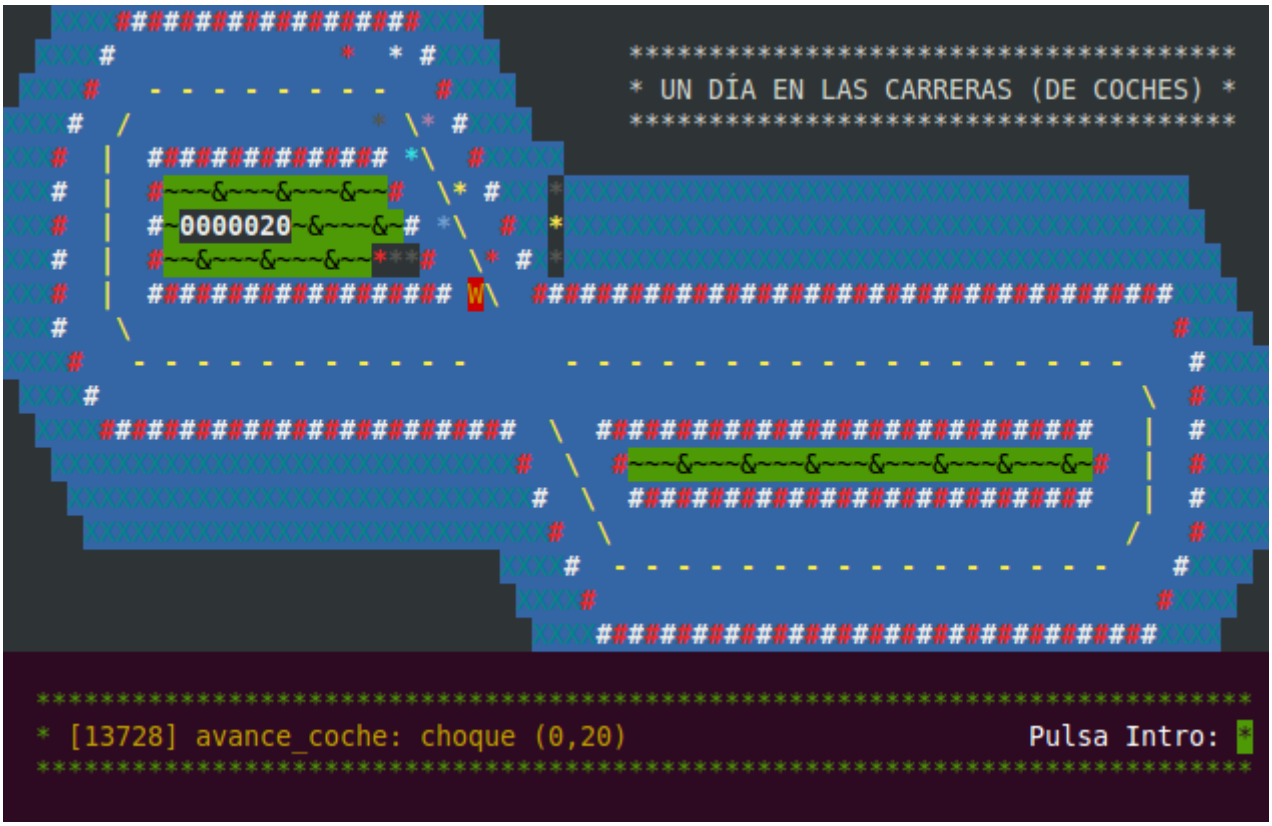
Disponéis, además, de un fichero de cabeceras, **falonso.h**, donde se encuentran definidas las macros que usa la biblioteca.

El proceso inicial se encargará de preparar todas las variables y recursos IPC de la aplicación. También se encargará de crear todos los procesos que gobernarán los coches. Manejará así mismo, los semáforos del circuito. El circuito tiene forma de lemniscata y posee dos carriles por los que se circula en el mismo sentido. Los denominaremos carril derecho (CD) y carril izquierdo (CI). La posición de un coche en el circuito viene completamente determinada dando su carril y el desplazamiento dentro de él. El desplazamiento es un número entero comprendido entre 0 y 136, ambos incluidos. Sendos planos del circuito, uno para cada carril, con los desplazamientos indicados se pueden observar a continuación:



```
        CARRIL_IZQUIERDO
   XXXX####################XXXX
  XXXX# 60123456789012345 #XXXX
 XXXX# 5 0 - - - - 1 - - 6 #XXXX
XXXX# 4/                  \7 #XXXX
XXX# 3|  ###############  \8 #XXXX
XXX# 2|  #~~~&~~~&~~~&~~#  \9 #XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXX# 1|  #~&~~~&~~~&~~~&~#  \0 #XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXX# 031 #~~&~~~&~~~&~~~&~# 2\1 #XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXX# 9|  ###################  \2 ##########################################XXXX
XXX# 8 \     1         1    1   3                                          #XXXX
XXXX# 7 - - -2- - - - -1- - 0    4 - - - - 9 - - - - 8 - - - - 7 - - -     #XXXX
 XXXX# 654321098765432109876543210  5  654321098765432109876543210987654 \  #XXXX
  XXXX############################  \6 ################################  3|  #XXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXX#  \7 #~~~&~~~&~~~&~~~&~~~&~~~&~~~&~# 2|  #XXXX
    XXXXXXXXXXXXXXXXXXXXXXXXXXXXX#  \8 ############################  1|  #XXXX
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXX#  \90123456789012345678901234567890/   #XXXX
                                XXXX#  -3- - - - -4- - - - -5- - - - -6   #XXXX
                                 XXXX#                                   #XXXX
                                  XXXX########################################XXXX
```

Según se va ejecutando el programa, se ha de ver una imagen parecida a la siguiente:



La práctica se invocará especificando **dos parámetros** desde la línea de órdenes. El primero es el **número de coches** que van a participar en la carrera, que podrá ser un mínimo de 2 y un máximo de 135, y si no viene, usad 10 por defecto. El segundo es el **retardo** o velocidad de la simulación, que puede ser 'R' ó 'N' (por defecto si no viene). Si es 'N', la variable del retardo será 1, y la práctica funcionará a velocidad normal. Si es 'R', el retardo será 0, e irá a la máxima velocidad.

Los coches, al principio, se pueden colocar como se desee, siempre que dos coches no compartan la misma posición. Una vez en movimiento, ningún coche podrá chocar con otro de la pista.

Cada coche corre por un carril del circuito, y una velocidad, pero se pueden variar, es decir, se puede acelerar o frenar, y se puede cambiar de carril, pero eso se deja a vuestra discreción.

Existe una función de cambio de carril hace que un coche cambie su carril y el desplazamiento según la siguiente tabla:

Cambio de Carril	
Derecho → Izquierdo	
0..13	0..13
14..28	15..29
29..60	29..60
61..62	60..61
63..65	61..63
66..67	63..64
68	64
69..129	64..124
130	127
131..134	129..132
135..136	134..135

Cambio de Carril	
Izquierdo → Derecho	
0..15	0..15
16..28	15..27
29..58	29..58
59..60	60..61
61..62	63..64
63..64	67..68
65..125	70..130
126	130
127..128	130..131
129..133	131..135
134..136	136

Así, por ejemplo, un coche situado en la posición 80 del Carril Derecho (CD, 80) pasa a (CI, 75) al cambiar de carril. Uno que esté en (CI, 126) pasa a (CD, 130), por su parte.

Existe **un cruce** cerca del centro del circuito que está regulado mediante un par de semáforos (luminosos), uno vertical y otro horizontal. El funcionamiento de los semáforos lo controla el proceso principal y deberá ser razonable (no los puede mantener apagados o siempre en verde, o siempre en rojo, por ejemplo). Si un semáforo está en rojo, ningún coche deberá sobrepasarlo.

El circuito también cuenta con un **contador de vueltas** automático, situado en las posiciones (CD, 133) y (CI, 131). La aplicación desarrollada llevará cuenta, por su parte, en una variable compartida del número de pasos por el contador. Esto es, cada coche, al pasar por el contador, incrementará la variable compartida. Al finalizar el programa, se deberá pasar la dirección de memoria a la biblioteca para que esta compruebe que vuestra cuenta y la de ella coinciden.

El programa estará en continua ejecución hasta que el usuario pulse las teclas CTRL+C desde el terminal. En ese momento, todos los coches deben parar (en un estado consistente) y morirse. El proceso primero esperará por su muerte. Informará a la biblioteca de que ha acabado, proporcionándole vuestra cuenta de vueltas y destruirá los mecanismos IPC utilizados.

Para que cualquier proceso pueda conocer en todo momento el estado del sistema, se va a usar una zona de memoria compartida. Los procesos no escribirán nunca en las partes controladas por la biblioteca. Son sólo informaciones que pueden consultarse. El mapa de la zona, expresado en bytes, es:

- 0-136: estado del carril derecho (un espacio ' ' <=> libre, otro valor <=> ocupado por un coche)
- 137-273: ídem carril izquierdo (lo mismo que 137+[0..136]).
- 274: estado del semáforo para la dirección horizontal (ROJO, AMARILLO, VERDE ó NEGRO, macros de **`falonso.h`**)
- 275: ídem para la dirección vertical.
- 276-300: reservado biblioteca.
- 301-...: libre para vuestras necesidades, en particular para que contéis las vueltas.

Siguiendo la misma filosofía, deberéis crear un conjunto de semáforos, el primero de los cuales (el índice 0) se reservará para el funcionamiento interno de la biblioteca. El resto, podéis usarlos libremente.

Para simplificar la realización de la práctica, se os proporciona una biblioteca estática de funciones **libfalonso.a** que debéis enlazar con vuestro módulo objeto para generar el ejecutable. Disponéis, además, de un fichero de cabeceras, **falonso.h**, donde se encuentran definidas las macros que usa la biblioteca.

Las funciones proporcionadas por la biblioteca son las que a continuación aparecen. De no indicarse nada, las funciones devuelven -1 en caso de error:

- **int inicio_falonso(int ret, int semAforos, char *zona)**
  El primer proceso, después de haber creado los mecanismos IPC que se necesiten, y antes de haber tenido ningún hijo (coches), debe llamar a esta función, indicando en **ret** si desea velocidad normal (1) o no (0) y pasando el identificador del conjunto de semáforos y el puntero a la zona de memoria compartida recién creados.

- **int inicio_coche(int *carril, int *desp, int color)**
  Esta función se debe invocar cuando se cree un nuevo coche. Se pasarán por referencia las variables para indicar el carril y desplazamiento del coche y por valor, su color. La variable **carril** podrá tener los valores CARRIL_DERECHO o CARRIL_IZQUIERDO, macros definidas en **falonso.h**. Los colores disponibles se encuentran definidos en el fichero de cabeceras **falonso.h**. No está permitido un coche azul porque no se ve. Podéis añadir 8 al código de color, si deseáis un tono más tenue o 16, si lo queréis más vivo. Si la posición donde queremos colocar el coche ya está ocupada, también se producirá un error.

- **int avance_coche(int *carril, int *desp, int color)**
  Dado un coche situado en la posición del circuito indicada por las variables enteras **carril** y **desp**, esta función le hará avanzar una posición en el circuito, modificando dichas variables de un modo acorde.

- **int cambio_carril(int *carril, int *desp, int color)**
  Igual que la función anterior, pero el coche sólo cambia de carril, manteniéndose en la misma posición (según la función de cambio de carril).

- **int luz_semAforo(int direcciOn, int color)**
  Pone el semáforo indicado en **direcciOn** (HORIZONTAL, VERTICAL) al color señalado en **color** (ROJO, AMARILLO, VERDE o NEGRO).

- **int pausa(void)**
  Hace una pausa de aproximadamente una décima de segundo, sin consumir CPU.

- **int velocidad(int v, int carril, int desp)**
  Dado un coche situado en (**carril**, **desp**), y que marcha a una velocidad **v**, esta función ejecuta una pausa, sin consumo de CPU, del tamaño justo que hay entre dos avances del coche. La velocidad **v** estará comprendida entre 1 y 99, reservándose al valor 100 para la máxima velocidad que permite el ordenador (pausa efectiva nula). A esta función se llamará cada vez que se vaya a intentar un avance de coche.

- **int fin_falonso(int *cuenta)**
  Se llama a esta función después de muertos los hijos y tras haber esperado por ellos, justo antes de destruir los recursos IPC. El parámetro es la dirección de memoria compartida donde se ha llevado la cuenta de las vueltas de los coches (pasos por línea de meta).

Notas acerca de las funciones:

*** Observad que existe mucha sincronización que no se ha declarado explícitamente y debéis descubrir dónde y cómo realizarla. Os desaconsejamos el uso de señales para sincronizar. Una pista para saber dónde puede ser necesaria una sincronización son frases del estilo: "después de ocurrido esto, ha de pasar aquello" o "una vez todos los procesos han hecho tal cosa, se procede a tal otra".

Respecto a la sincronización interna de la biblioteca, se usa el semáforo reservado para conseguir atomicidad en la actualización de la pantalla, el manejo de la memoria compartida, y en las verificaciones. Para que las sincronizaciones que de seguro deberéis hacer en vuestro código estén en sintonía con las de la biblioteca, debéis saber que sólo las funciones que actualizan valores sobre la pantalla están sincronizadas mediante el semáforo de la biblioteca, de todas formas, para que esas sincronizaciones estén en sintonía con la biblioteca, a continuación tenéis un pseudo código de las funciones que realiza la biblioteca. S es el semáforo interno que utiliza.

```
* inicio_falonso:
     - limpia la pantalla
     - S=1
     - mensaje de bienvenida
     - dibuja el circuito

* inicio_coche:
     - comprobación de parámetros
     - W(S)
     - si posición ocupada, S(S), poner error, volver.
     - pintar el coche y actualizar la memoria compartida
     - S(S)

* avance_coche:
     - comprobación de parámetros
     - W(S)
     - borrar el coche y actualizar la memoria compartida
     - avanzar una posición
     - si hay choque, S(S), poner error, volver.
     - pintar el coche y actualizar la memoria compartida
     - si pasamos por el contador, incrementarlo y pintarlo
     - S(S)
     - si nos hemos saltado un semáforo en rojo, poner error, volver.

* cambio_carril:
     - comprobación de parámetros
     - W(S)
     - borrar el coche y actualizar la memoria compartida
     - cambiar de carril
     - si hay choque, S(S), poner error, volver.
     - pintar el coche y actualizar la memoria compartida
     - S(S)
     - si nos hemos saltado un semáforo en rojo, poner error, volver.

* luz_semAforo:
     - comprobación de parámetros
     - W(S)
     - dibujar semáforo y actualizar memoria compartida
     - S(S)

* fin_falonso:
     - si no coinciden las vueltas, poner error, volver.

Cada vez que se pone un error, se pone W(S) y S(S) rodeando la impresión
en pantalla. Las funciones pausa() y velocidad() no usan el semáforo.
```

En cuanto al consumo de CPU, no debe consumirse CPU cuando un coche espere a que el semáforo de su carril se ponga en verde. Tampoco en los instantes iniciales cuando, después de haber colocado todos, deis el pistoletazo de salida. Podéis consumir CPU para evitar choques o alcances, o los coches que estén esperando a la cola de un semáforo en rojo, aunque evidentemente se premiará al que logre hacerlo bien. En estos casos, y para ser respetuosos con el ordenador, se realizará en todo caso "semiespera ocupada", intercalando en el sondeo una pausa de una décima de segundo (llamar a la función "`pausa()`").

En esta práctica no se podrán usar ficheros para nada, salvo que se indique expresamente. Las comunicaciones de PIDs o similares entre procesos, si hicieran falta, se harán mediante ***mecanismos IPC***.

Siempre que en el enunciado o LPEs se diga que se puede usar `sleep()`, se refiere a la *llamada al sistema*, no a la orden de la línea de órdenes.

Los mecanismos IPC (semáforos, memoria compartida y paso de mensajes) son recursos muy limitados. Es por ello, que vuestra práctica **sólo podrá usar** un conjunto de semáforos, una zona de

memoria compartida y un buzón de paso de mensajes (si lo usáis) como máximo. Además, si se produce cualquier error o se finaliza normalmente, los recursos creados han de ser eliminados. Una manera fácil de lograrlo es registrar la señal SIGINT para que lo haga y mandársela uno mismo si se produce un error.

Biblioteca de funciones `libfalonso`

Con esta práctica se trata de que aprendáis a sincronizar y comunicar procesos en UNIX. Su objetivo no es la programación. Es por ello que se os suministra una biblioteca estática de funciones ya programadas para tratar de que no tengáis que preocuparos por la presentación por pantalla, la gestión de estructuras de datos (colas, pilas, ...) , etc. También servirá para que se detecten de un modo automático errores que se produzcan en vuestro código. Para que vuestro programa funcione, necesitáis la propia biblioteca **`libfalonso.a`** y el fichero de cabecera **`falonso.h`**. La biblioteca funciona con los códigos de VT100/xterm, por lo que debéis adecuar vuestros simuladores a este terminal.

2. Pasos recomendados para la realización de la práctica.

Aunque ya deberíais ser capaces de abordar la práctica sin ayuda, aquí van unas guías generales:

1. Crear los recursos IPC necesarios (semáforos, memoria compartida, buzón de paso de mensajes), y comprobad que se crean bien, con `ipcs`. Es preferible, para que no haya interferencias, que los definái privados.
2. Registrad SIGINT para que cuando se pulse CTRL+C se ordene la terminación, se limpien los procesos hijo y se eliminen los recursos IPC. Lograr que si el programa acaba normalmente o se produce cualquier error, también se eliminen los recursos (mandad una señal SIGINT al propio proceso en esos casos).
3. Tratad los parámetros de la línea de órdenes, aunque no los utilicéis por ahora.
4. Llamar a la función `inicio_falonso` en `main`. Debe aparecer la pantalla de bienvenida y, pasados dos segundos, dibujarse el circuito. Dormir un tiempo y añadid al final la llamada a la función `fin_falonso`.
5. Probad a poner los semáforos a distintos colores.
6. Llega el momento de probar el circuito. Cread un hijo que maneje un coche que no cambie de carril y corra a velocidad constante.
7. Añadid otro coche y plantead una rutina que evite los choques por alcance.
8. Introducid los cambios de carril, si no lo habéis hecho ya para evitar los choques, y cuidad de que no se produzcan choques al cambiar de carril.
9. Haced que el primer proceso maneje de modo razonable los semáforos.
10. Que los coches respeten el semáforo en rojo en cualquiera de las direcciones.
11. Tened en cuenta el argumento que indica el número de coches y cread tantos como indique dicho número.
12. Programar la cuenta, corregir la pulsación de CTRL+C si en alguna ocasión no funciona y completar la llamada a `fin_falonso` con la cuenta obtenida.
13. Pulid los últimos detalles.

3. Presentación y Plazo.

Esta práctica de grupo para el Sistema Operativo LINUX, se entregará a través de una tarea creada al efecto en la plataforma moodle de la asignatura (http://studium.usal.es), con lo que el plazo límite de presentación viene marcado por dicha tarea. Debéis consultar los plazos de la misma para conocer específicamente las fechas de entrega. No obstante, se estima que, desde su planteamiento, deberá ser entregada en **cuatro semanas** aproximadamente.

4. Normas de presentación.

1. Tenéis que entregar el código fuente y la documentación en un solo archivo (comprimido en .zip o .rar), con el nombre indicado. Además, sólo podéis subir uno a Moodle.
2. En la cabecera del código fuente, se especificará los años del curso (2015-2016 por ejemplo), el número y tipo de práctica en grupo (primera, segunda, etc. para Linux o Windows), el título, el código de grupo de prácticas asignado, los nombres y apellidos de los integrantes del grupo, y la fecha.
3. Se incluirá junto con la práctica la documentación indicada más adelante. Dentro del código fuente se incluirán los comentarios relativos a novedades introducidas sobre lo visto en clase y sólo eso. Esto se hace para facilitar el proceso de corrección. En la mayor parte de los casos, no añadiréis nada por ser obvio, pero a veces es bueno comentar los algoritmos utilizados.
4. Si la práctica incluye salida por pantalla, hay que ajustarse exacta y literalmente a lo especificado, carácter a carácter, línea a línea, por si se automatiza parte del proceso de corrección. Es evidente que debéis borrar, comentar o dejar inactivos todos los comentarios de depuración que aparezcan por la pantalla, si es que los habéis usado.
5. Es necesario que los integrantes del grupo de prácticas informen previamente al profesor de quienes son para que se les asigne un código de grupo.
6. El no cumplimiento de los plazos y modo de entrega dará lugar a una reducción de la nota o a que la práctica esté suspensa, dependiendo de la gravedad de la falta.

Documentación Adicional:

Además de estas normas, en esta práctica se debe entregar un esquema donde aparezcan los recursos IPC usados (semáforos con sus valores iniciales, memoria compartida con sus variables, usos y significados, buzones y mensajes pasados, en caso que se usen), y un pseudo código sencillo para cada proceso con las operaciones *wait* y *signal*, *send* y *receive* realizadas sobre ellos. Por ejemplo, si se tratara de sincronizar dos procesos C y V para que produjeran alternativamente consonantes y vocales, comenzando por una consonante, deberíais entregar algo parecido a esto:

```
SEMÁFOROS Y VALOR INICIAL: SC=1, SV=0.

SEUDOCÓDIGO:

       C                              V
      ===                            ===
 Por_siempre_jamás              Por _siempre_jamás
 {                              {
     W(SC)                          W(SV)
     escribir_consonante            escribir_vocal
     S(SV)                          S(SC)
 }                              }
```

Daos cuenta que lo que importa en el pseudo código es la sincronización. El resto puede ir muy esquemático. Un buen esquema os facilitará muchísimo la defensa.

5. Evaluación de la práctica.

Dada la dificultad para la corrección de programación en paralelo, el criterio que se seguirá para la evaluación de la práctica será: si...

a) la práctica cumple las especificaciones de este enunciado y,
b) la práctica no falla en ninguna de las ejecuciones a las que se somete y,
c) no se descubre en la práctica ningún fallo de construcción que pudiera hacerla fallar, por muy remota que sea esa posibilidad...

...se aplicará el principio de "presunción de inocencia" y la práctica estará aprobada. La nota, a partir de ahí, dependerá de la simplicidad de las técnicas de sincronización usadas, la corrección en el tratamiento de errores, la cantidad y calidad del trabajo realizado, etc.

6. Notas sobre el desarrollo de la práctica LPEs.

I. ¿Dónde poner un semáforo? Dondequiera que uséis la frase, "el proceso debe esperar hasta que..." es un buen candidato a que aparezca una operación wait sobre un semáforo. Tenéis que plantearos a continuación qué proceso hará signal sobre ese presunto semáforo y cuál será su valor inicial.

II. Si ejecutáis la práctica en *segundo plano* (con ampersand (&)) es normal que al pulsar CTRL+C el programa no reaccione. El terminal sólo manda `SIGINT` a los procesos que estén en primer plano. Para probarlo, mandad el proceso a primer plano con `fg %` y pulsad entonces CTRL+C.

III. Un "truco" para que sea menos penoso el tratamiento de errores consiste en dar valor inicial a los identificadores de los recursos IPC igual a -1. Por ejemplo, `int semAforo=-1`. En la manejadora de `SIGINT`, sólo si `semAforo` vale distinto de -1, eliminái s el recurso con `semctl`. Esto es lógico: si vale -1 es porque no se ha creado todavía o porque al intentar crearlo la llamada al sistema devolvió error. En ambos casos, no hay que eliminar el recurso.

IV. Para evitar que todos los identificadores de recursos tengan que ser variables globales para que los vea la manejadora de `SIGINT`, podéis declarar una estructura que los contenga a todos y así sólo gastáis un identificador del espacio de nombres globales.

V. A muchos os da el error "*Interrupted System Call*". Mirad la sesión de "Sucesos Asíncronos", en el apartado "Llamadas al sistema interrumpidas (bloqueantes y no bloqueantes)". Allí se explica lo que pasa con `wait`. A vosotros os pasa con `semop`, pero es lo mismo. Aprovechad la solución que se propone al final del apartado.

VI. Al acabar la práctica, con CTRL+C, al ir a borrar los recursos IPC, puede ser que os ponga "*Invalid argument*", pero, sin embargo, se borren bien. La razón de esto es que habéis registrado la manejadora de `SIGINT` para todos los procesos. Al pulsar CTRL+C, la señal la reciben todos, el padre y los otros procesos. El primero que obtiene la CPU salta a su manejadora y borra los recursos. Cuando saltan los demás, intentan borrarlos, pero como ya están borrados, os da el error. Esto debería evitarse, configurando la señal tras la creación de los hijos o similar.

VII. Se os recuerda que, si ponéis señales para sincronizar esta práctica, la nota bajará. Usad semáforos, que son mejores para este cometido, y de eso se trata la práctica también.

VIII. Todos vosotros, tarde o temprano, os encontráis con un error que no tiene explicación: un proceso que desaparece, un semáforo que parece no funcionar, etc. La actitud en este caso no es tratar de justificar la imposibilidad del error. Así no lo encontraréis. Tenéis que ser muy sistemáticos. Hay un árbol entero de posibilidades de error y no tenéis que descartar ninguna de antemano, sino ir podando ese árbol. Tenéis que encontrar a los procesos responsables y tratar de localizar la línea donde se produce el error. Si el error es "*Segmentation fault. Core dumped*", es que estáis accediendo a zonas de memoria no permitidas para el proceso, y para localizar el problema no os quedará más remedio que depurar mediante órdenes de impresión dentro del código.

Para ello, insertad líneas del tipo:

```
fprintf(stderr,"...",...);
```

donde sospechéis que hay problemas. En esas líneas identificad siempre al proceso que imprime el mensaje. Comprobad todas las hipótesis, hasta las más evidentes. Cuando ejecutéis la práctica, redirigid el canal de errores a un fichero con `2>salida.txt`, por ejemplo.

Si cada proceso pone un identificador de tipo "P1", "P2", etc. en sus mensajes, podéis quedaros con las líneas que contienen esos caracteres con:

```
grep "P1" salida > salida2
```

IX. Os puede dar un error que diga "*Resource temporarily unavailable*" en el fork del padre. Esto ocurre cuando no exorcizáis adecuadamente a los procesos hijos zombies del padre. Hay dos posibilidades para solucionarlo:
   1. La más sencilla es hacer que el padre ignore la señal `SIGCLD` con un `sigaction` y `SIG_IGN`. El S.O. tradicionalmente interpreta esto como que no queréis que los hijos se queden zombies, por lo que no tenéis que hacer `waits` sobre ninguno de ellos para que acaben de morir.
   2. Interceptar `SIGCLD` con una manejadora en el padre y, dentro de ella, hacer los `waits` que sean necesarios para que los hijos mueran. Pero esto trae un segundo problema algo sutil: al recibirse la señal, todos los procesos bloqueados en cualquier llamada al sistema bloqueante (en particular, los WAITs de los semáforos) van a fallar. Si no habéis puesto comprobación de errores, los semáforos os fallarán sin motivo aparente. Si la habéis puesto, os pondrá *Interrupted system call* en el `perror`. Como podéis suponer, eso no es un error y debéis interceptarlo para que no ponga el `perror` y reintente el WAIT. La clave está en la variable `errno` que valdrá `EINTR` en esos casos.

X. No se debe dormir (es decir, ejecutar sleeps o pausas) dentro de una sección crítica. El efecto que se nota es que, aunque la práctica no falla, parece como si solamente un proceso se moviera o apareciera en la pantalla a la vez. Siendo más precisos, si dormís dentro de la sección crítica, y soltáis el semáforo para, acto seguido, volverlo a coger, dais muy pocas posibilidades al resto de procesos de que puedan acceder.

XI. La biblioteca no reintenta los WAITs de su semáforo, por lo que, de recibirse una señal, podría fallar. Si os da problemas, simplemente ignorad la señal `SIGCLD` en el padre como se explica más arriba.

XII. El número de coches máximo no está fijado, aunque como podéis imaginar será el máximo de posiciones diferentes del circuito menos una al menos para que se puedan mover. Sin embargo, ninguna de las pruebas que se hagan para evaluar la práctica pasará de **veinte coches**.

XIII. Os preguntaréis acerca de dónde se hacen las comprobaciones de semáforos y el incremento de vueltas. Aquí lo tenéis:
   - Para el semáforo vertical, los puntos de comprobación son (CD, 21) y (CI, 23). Es decir, los coches se deben parar en el (CD, 20) y (CI, 22).
   - Para el semáforo horizontal, los puntos de comprobación son (CD, 106) y (CI, 99).
   - Los incrementos de vueltas se efectúan cuando un coche avanza hasta (CD, 133) ó (CI, 131). Cuidado con los cambios de carril, no contéis vueltas de más.

XIV. Respecto al "pistoletazo" de salida, considerad que se trata de una sincronización de *rendezvous* o reunión. Ningún coche se puede adelantar al pistoletazo y todos los coches tienen que estar iniciados cuando el pistoletazo se produzca. ¡A discurrir!