PRÁCTICAS DE SISTEMAS OPERATIVOS II

PRIMERA PRÁCTICA EVALUABLE

Encrucijada

---

1. Enunciado.

Antes de estudiar esta asignatura hubimos de sufrir la falta de medios para comunicar adecuadamente a los procesos entre sí. Ahora pondremos remedio a la situación permitiendo usar los nuevos mecanismos IPC recientemente aprendidos. Se tratará de regular la circulación en un cruce controlado por semáforos de tráfico.

En esta práctica usaréis una biblioteca de enlazado estático que se os proporcionará. El objetivo es doble: por un lado aprender a usar una de tales bibliotecas y por otro descargar parte de la rutina de programación de la práctica para que os podáis centrar en los problemas que de verdad importan en esta asignatura.

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

Para simplificar la realización de la práctica, se os proporciona una biblioteca estática de funciones (`libcruce.a`) que debéis enlazar con vuestro módulo objeto para generar el ejecutable. Gracias a ella, algunas de las funciones necesarias para realizar la práctica no las tendréis que programar sino que bastará nada más con incluir la biblioteca cuando compiléis el programa. La línea de compilación del programa podría ser:

```
gcc cruce.c libcruce.a -o cruce
```

Disponéis, además, de un fichero de cabeceras, `cruce.h`, donde se encuentran definidas, entre otras cosas, las macros que usa la biblioteca y las cabeceras de las funciones que ofrece.

El proceso inicial se encargará de preparar todas las variables y recursos IPC de la aplicación y registrar manejadoras para las señales que necesite. Este proceso, además, debe tomar e interpretar los argumentos

de la línea de órdenes y llamar a la función `CRUCE_inicio` con los parámetros adecuados. El proceso será responsable de crear los procesos adicionales necesarios. Cada peatón o cada coche simulado será representado mediante un proceso, hijo del proceso principal. También es responsabilidad del primer proceso el controlar que, si se pulsa CTRL+C la práctica acaba, no dejando procesos en ejecución ni recursos IPCs sin borrar. La práctica devolverá 0 en caso de ejecución satisfactoria o un número mayor que cero, en caso de detectarse un error.

La práctica se invocará especificando dos parámetros obligatorios desde la línea de órdenes. El primer parámetro será el número máximo de procesos que puede haber en ejecución simultánea. El segundo consistirá en un valor entero mayor o igual que cero. Si es 1 o mayor, la práctica funcionará tanto más lenta cuanto mayor sea el parámetro y no deberá consumir CPU apreciablemente. El modo de lograr esto lo realiza la propia biblioteca. Vosotros no tenéis más que pasar dicho argumento a la función de inicio. Si es 0, irá a la máxima velocidad, aunque el consumo de CPU sí será mayor. Por esta razón y para no penalizar en exceso la máquina compartida, no debéis dejar excesivo tiempo ejecutando en el servidor la práctica a máxima velocidad.

El programa debe estar preparado para que, si el usuario pulsa las teclas CTRL+C desde el terminal, la ejecución del programa termine en ese momento y adecuadamente. Ni en una terminación como esta, ni en una normal, deben quedar procesos en ejecución ni mecanismos IPC sin haber sido borrados del sistema. Este es un aspecto muy importante y se penalizará bastante si la práctica no lo cumple.

Es probable que necesitéis semáforos o buzones para sincronizar adecuadamente la práctica. En ningún caso podréis usar en vuestras prácticas más de un array de semáforos, un buzón de paso de mensajes y una zona de memoria compartida. Se declarará un array de semáforos de tamaño adecuado a vuestros requerimientos, el primero de los cuales se reservará para el funcionamiento interno de la biblioteca. El resto, podéis usarlos libremente.

La biblioteca requiere memoria compartida. Debéis declarar una única zona de memoria compartida en vuestro programa. Los 256 bytes primeros de dicha zona estarán reservados para la biblioteca. Si necesitáis memoria compartida, reservad más cantidad y usadla a partir del byte bicentésimo quincuagésimo séptimo.

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

- `int CRUCE_inicio(int ret, int maxProcs, int semAforos, char *zona)`
  El primer proceso, después de haber creado los mecanismos IPC que se necesiten y antes de haber tenido ningún hijo, debe llamar a esta función, indicando en `ret` la velocidad de presentación y en `maxProcs` el número máximo de procesos permitidos en esta ejecución (parámetros ambos de la línea de órdenes) y pasando además el identificador del conjunto de semáforos que se usará y el puntero a la zona de memoria compartida declarada para que la biblioteca pueda usarlos.
- `int CRUCE_pon_semAforo(int sem, int color)`
  Pone el semáforo `sem` al color `color`. El primer parámetro puede ser: `SEM_P1`, `SEM_P2`, `SEM_C1`, `SEM_C2`, para los semáforos de peatones y coches, respectivamente. El segundo parámetro puede valer: `ROJO`, `AMARILLO`(solamente para los semáforos de coches) o `VERDE`. Estas son todas macros definidas en `cruce.h`.
- `int CRUCE_nuevo_proceso(void)`
  El padre, después de haber creado todo lo necesario, se encuentra en un bucle infinito en el que va generando los nuevos procesos, coches o peatones. Esta función le devuelve `COCHE` o `PEAToN` para que sepa de qué tipo es el proceso que tiene que crear a continuación.
- `struct posiciOn CRUCE_inicio_coche(void)` y `struct posiciOn CRUCE_inicio_peatOn(void)`
  El nuevo proceso hijo, dependiendo de si es coche o peatón, llamará a una de estas dos funciones. La función correspondiente devolverá las coordenadas de la posición siguiente del objeto recién creado. La función `CRUCE_inicio_peatOn` es desaconsejada. Úsese mejor la siguiente función.
- `struct posiciOn CRUCE_inicio_peatOn_ext(struct posiciOn *posNacimiento)`
  El nuevo proceso hijo, si es un peatón, llamará a esta función. La función devolverá las coordenadas de la posición siguiente del peatón recién creado y la posición de nacimiento en el parámetro pasado por referencia. Esta función sustituye a la correspondiente del apartado anterior.

- `struct posiciOn CRUCE_avanzar_coche(struct posiciOn sgte)` y `struct posiciOn CRUCE_avanzar_peatOn(struct posiciOn sgte)`
  El proceso, después de haber llamado a la función anterior, se mete en un bucle de avance. A esta función se le pasa la posición a la que se quiere ir y devuelve la nueva posición siguiente. Del bucle se saldrá cuando en la coordenada *y* de la posición devuelta haya un valor menor que cero.
- `int pausa_coche(void)` e `int pausa(void)`
  Entre dos avances consecutivos, los coches llaman a la primera función y los peatones a la segunda. La segunda función también sirve para medir las pausas del ciclo semafórico
- `int CRUCE_fin_coche(void)` e `int CRUCE_fin_peatOn(void)`
  El proceso hijo que haya salido del bucle de avance, invoca esta función.
- `int CRUCE_fin(void)`
  El padre, una vez sabe que ha acabado la práctica y antes de realizar limpieza de procesos y mecanismos IPC debe llamar a esta función.
- `void pon_error(char *mensaje)`
  Pone un mensaje de error en el recuadro azul de la parte inferior de la pantalla y espera a que el usuario pulse "Intro". La podéis usar para depurar.

Estad atentos pues pueden ir saliendo versiones nuevas de la biblioteca para corregir errores o dotarla de nuevas funciones.

El guión que seguirá el proceso padre será el siguiente:

1. Tomará los datos de la línea de órdenes y los verificará.
2. Iniciará las variables, mecanismos IPC, manejadoras de señales y demás.
3. Llamará a la función `CRUCE_inicio`.
4. Creará el proceso gestor semafórico.
5. Entrará en un bucle infinito del que solamente saldrá si se pulsa CTRL+C. Dentro del bucle:
   a. Si hay tantos procesos como el máximo declarado, se queda esperando, sin consumo de CPU, hasta que muera alguno.
   b. Llamará a la función `CRUCE_nuevo_proceso`, que responderá indicando de qué tipo será: coche o peatón.
   c. Creará un proceso hijo, que ejecutará las funciones correspondientes a su tipo.
6. Cuando se pulse CTRL+C, se engargará de finalizar todo ordenadamente.

Por su parte, el proceso gestor semafórico realizará lo que sigue en un bucle infinito:

1. Establecerá el estado de los semáforos de tráfico según en la fase en que nos encontremos.
2. Dormirá, sin consumo de CPU, el tiempo correspondiente a esa fase.

Finalmente, los procesos que circulan (peatones o coches), entrarán en un bucle después de llamar a su función de inicio. Dentro del bucle, llamarán a la función de pausa y a la de avance hasta que esta última devuelva un valor menor que cero en la coordenada *y*. El proceso acaba no sin antes llamar a la función de fin.

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 y 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.

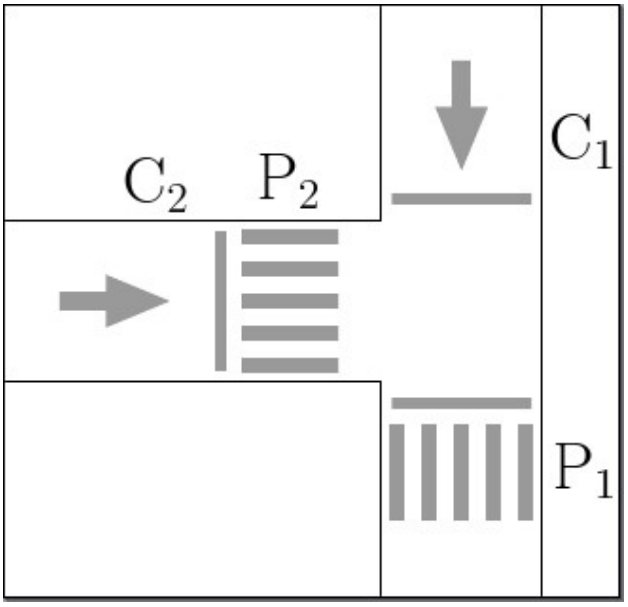
En esta práctica no se podrán usar ficheros para nada, salvo que se indique expresamente. Las comunicaciones de PIDs o similares entre procesos, si hicieran falta, se harán mediante *mecanismos IPC*.

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, un buzón de paso de mensajes y una zona de memoria compartida 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.

Ciclo semafórico

En la práctica hay cuatro semáforos de tráfico, dos de peatones (P1 y P2) y dos de automóviles (C1 y C2):



El ciclo semafórico pasa por tres fases principales:

- *Primera fase*: C1 y P2 en verde. El resto, en rojo. Duración: 6 pausas(*).
- *Segunda fase*: C2 en verde. El resto, en rojo. Duración: 9 pausas.
- *Tercera fase*: P1 en verde. El resto, en rojo. Duración: 12 pausas.

Las fases se completan con el estado de AMARILLO por el que deben pasar los dos semáforos de coches antes de pasar a ROJO. Este estado es especial, pues su duración será, como mínimo, de dos pausas, pero que podrá prolongar su duración hasta que se libere el cruce y así evitar un posible choque.

(*) Una *pausa* equivale a una llamada a la función `pausa()`.

Biblioteca de funciones `libcruce.a`

Con esta práctica se trata de que aprendáis a sincronizar y comunicar procesos en UNIX. Su objetivo no es la programación, aunque es inevitable que tengáis que programar. Es por ello que se os suministra una biblioteca estática de funciones ya programadas para tratar de que no debáis 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 `libcruce.a` y el fichero de cabeceras `cruce.h`. La biblioteca funciona con los códigos de VT100/xterm, por lo que debéis adecuar vuestros simuladores a este terminal. También se usa la codificación UTF-8, por lo que necesitáis un programa de terminal que sepa interpretarlos. Los terminales de Linux lo hacen por defecto, pero si usáis Windows, debéis

aseguraros de que el programa tiene capacidad para interpretarlos y que esta capacidad está activada. Si no es así notaréis caracteres basura en la salida de modo que no se verá nada.

Ficheros necesarios:

- `libcruce.a`: para Solaris (ver 2.0), para el LINUX de clase (ver 2.0),
- `cruce.h`: Para todos (ver 2.0).

Registro de versiones:

- 1.0: primera versión
- 1.1: un error hace que no se reconozca bien al padre en `CRUCE_nuevo_proceso`
- 1.2: la biblioteca tenía preferencia por sacar coches en la dirección horizontal
- 2.0: para controlar el nacimiento de los peatones es necesario conocer no solamente su posición siguiente, sino también, su posición de nacimiento. Para ello, se crea una nueva función: `CRUCE_nuevo_peatOn_ext`

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 semáforos y la memoria compartida, 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 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 en esos casos al proceso padre).
3. Llamar a la función `CRUCE_inicio` en `main`. Debe aparecer la pantalla de bienvenida y, pasados dos segundos, dibujarse la pantalla. Añadid también la función `CRUCE_fin`
4. Probad a que el padre encienda uno o varios semáforos
5. Cread ahora un hijo para que se encargue del ciclo semafórico. Ajustadlo para que se comporte como dice el enunciado.
6. Comentad temporalmente el hijo de control de semáforos mediante `#ifdef`s, para que no molesten
7. Haced que el padre cree un coche y que el coche se mueva.
8. Ahora, que el padre cree varios coches, uno detrás del otro (esto es, cuando uno desaparece, crea el siguiente)
9. Ahora es cuando se complica. El padre debe crear hijos separados para que cada uno se haga cargo de un coche y puedan aparecer a la vez varios en la pantalla. Incorporat también la función `CRUCE_nuevo_proceso`. Si os dice que creéis un peatón, simplemente ignoradlo y volvedla a llamar. Si lo hacéis bien, los coches, debido a que tienen distinta velocidad, puede que se choquen. También puede ocurrir que se choquen en el cruce.
10. El objetivo ahora es diseñar un sistema que haga que un coche que tenga delante otro no avance hasta que el otro haya avanzado. Un esquema con un buzón con mensajes de distintos tipos, puede ser una buena opción
11. Hay que añadir ahora el código necesario para que el padre no cree más hijos de los declarados como máximos. Un semáforo será aquí la mejor opción
12. Encended ahora los semáforos, bien en la forma final o, al principio, más simplificadamente. Los coches intentarán pasárselos en rojo o amarillo.
13. Añadir el código necesario para que respeten, sin consumo de CPU, los semáforos.
14. Probad la práctica a velocidad 0 y en Solaris, en Linux, etc. Si aparecen errores, serán probablemente debidos a una mala sincronización
15. Si aún os quedan fuerzas, podéis repetir un esquema parecido, anulando temporalmente los coches, con los peatones.
16. Activad todo finalmente.
17. Pulid los últimos detalles.

18. Si la cosa marcha, celebradlo como vuestra edad os lo permita. Os lo merecéis. Después de un largo y duro recorrido, sabe como ninguna otra cosa el llegar a la meta.

3. Plazo de presentación.

Consultad la página de entrada de la asignatura.

4. Normas de presentación.

Acá están. Además de estas normas, en esta práctica se debe entregar un esquema donde aparezcan los semáforos usados, sus valores iniciales, sus buzones, y mesajes pasados y un seudocódigo 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 de que lo que importa en el pseudocódigo 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.

Debido a la complejidad de la práctica, se permitirá presentarla sin peatones. En ese caso, no obstante, la nota máxima que se puede obtener será de seis puntos.

6. LPEs.

I. ¿Se puede usar la biblioteca en un Linux de 64 bits? Aquí se os indican las claves.
II. ¿Se puede proporcionar la biblioteca para el Sistema Operativo X, procesador Y? Por problemas de eficiencia en la gestión y mantenimiento del código no se proporcionará la biblioteca más que para Solaris-SPARC y Linux-Intel de 32 bits. A veces podéis lograr encontrar una solución mediante el uso de máquinas virtuales.
III. ¿Dónde poner un semáforo? Dondequiera que uséis la frase, "el proceso puede llegar a 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, dónde lo hará y cuál será el valor inicial.

IV. 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.

V. 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.

VI. 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.

VII. A muchos os da el error "Interrupted System Call". Mirad la sesión dedicada a las señales, apartado quinto. Allí se explica lo que pasa con `wait`. A vosotros os pasa con `semop`, pero es lo mismo. De las dos soluciones que propone el apartado, debéis usar la segunda.

VIII. A muchos, la práctica os funciona exasperantemente lenta en Solaris. Debéis considerar que la máquina cuando la probáis está cargada, por lo que debe ir más lento que en casa o en el linux de clase.

IX. A aquellos que os dé "Bus error (Core dumped)" al dar valor inicial al semáforo, considerad que hay que usar la versión de `semctl` de Solaris (con `union semun`), como se explica en la sesión de semáforos y no la de HPUX.

X. 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.

XI. El compilador de encina tiene un bug. El error típicamente os va a ocurrir cuando definái s una variable entera en memoria compartida. Os va a dar `Bus Error. Core dumped` si no definís el puntero a esa variable apuntando a una dirección que sea múltiplo de cuatro. El puntero que os devuelve `shmat`, no obstante, siempre será una dirección múltiplo de cuatro, por lo que solo os tenéis que preocupar con que la dirección sea múltiplo de cuatro respecto al origen de la memoria compartida. La razón se escapa un poco al nivel de este curso y tiene que ver con el alineamiento de direcciones de memoria en las instrucciones de acceso de palabras en el procesador RISC de encina.

XII. Se os recuerda que, si ponéis señales para sincronizar esta práctica, la nota bajará. Usad semáforos, que son mejores para este cometido.

XIII. 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`", la línea os la dará si aplicáis lo que aparece en la sección Manejo del depurador. En cualquier otro caso, 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`.

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
```

XIV. Un peatón recién nacido puede ser arrollado por otro peatón, si no tenéis cuidado. Es necesario que

el nuevo proceso lea el mensaje correspondiente a su posición de nacimiento. Para ello, debéis usar la nueva función `CRUCE_inicio_peatOn_ext` de la versión 2.0 de la biblioteca, en lugar de `CRUCE_inicio_peatOn`. Además, si al nuevo proceso le quitan la CPU justo entre que nace y es capaz de reservar la posición, puede haber problemas. Para evitarlos, lo mejor es hacer una sección crítica que impida avanzar al resto de peatones hasta que se haya reservado la posición de nacimiento.

XV. Los coches pasan el semáforo de uno en uno. Una solución con Wait0 para ellos puede volverse muy complicada. Es mucho más fácil usar un único semáforo para regular el paso. Cuando el semáforo está a 1, se puede pasar. Si no, no. El seudocódigo del paso de verde a amarillo de uno de los semáforos podía ser (usando un semáforo, valor inicial=0):

```
Coche                                          Gestor
=====                                          ======
if (posSgte==justo en la raya)                 Poner en "VERDE"
 {WAIT(S);                                     SIGNAL(S);
  posSgte=CRUCE_avanzar_coche(posSgte);        esperar_tiempo_de_esa_fase
  SIGNAL(S);}                                  WAIT(S);
                                               Poner en "AMARILLO"
```

XVI. El paso de amarillo a rojo exige que el cruce esté libre de coches. Un modo muy fácil de hacerlo es iniciar un semáforo a un valor igual o superior al número de coches que caben en la parte problemática del cruce. Cuando un coche quiere entrar a esa parte del cruce, hace un WAIT. Después de salir, hace un SIGNAL. El valor en un momento dado del semáforo distará del valor máximo que le dimos en tantas unidades como coches bloqueen en ese momento el cruce. Para pasar a rojo con seguridad basta con que el gestor vaya "robando" una unidad a una unidad valor al semáforo hasta lograr hacerlo cero. Es decir, hará tantos WAITs como el valor inicial que le dimos al semáforo. Sabrá entonces que no hay nadie en el cruce.

XVII. Una de las partes más complejas de la práctica es gestionar bien el nacimiento de los peatones. Los peatones nacen, al azar, en una posición libre del lado izquierdo o inferior de la acera inferior izquierda. Siempre se moverán en esa acera hacia la derecha o hacia arriba. Llamemos a la zona donde nacen los peatones "zona de peligro". Debido, en parte, al modo en que está construida la biblioteca, es complicado gestionar bien los nacimientos debido al siguiente problema: Imaginemos un peatón que nace en la fila de abajo de la acera y que se empeña en ir siempre a la derecha y no salir nunca de la zona de peligro. En cualquier momento puede nacerle otro peatón encima o echarse encima de un recién nacido. Parece necesario usar un semáforo para que esto no pueda ocurrir (valor inicial de semáforo=1):

```
PEAToN NACIENDO                                PEAToN MOVIeNDOSE
===============                                =================
WAIT(S);                                       WAIT(S);
posSgte=CRUCE_inicio_peatOn(&posNac);          posAnterior=posActual;
reservar(posNac);                              posActual=posSgte;
SIGNAL(S);                                     reservar(posActual);
                                               posSgte=CRUCE_avanzar_peatOn(posActual);
                                               liberar(posAnterior);
                                               SIGNAL(S);
```

De este modo, si un peatón está naciendo, ningún otro peatón puede moverse hasta que le haya dado tiempo a reservar y asegurar su posición. Y, lo recíproco también es cierto (recordad que la biblioteca nunca creará un nuevo peatón en una posición ocupada). El problema de este esquema es que un proceso se puede bloquear en la función reservar y en el caso de que esto ocurriera, habría interbloqueo (él no puede avanzar por estar ocupada su posición y el resto no pueden avanzar por estar él dentro de la sección crítica). Hay que modificar el esquema para dar con una solución aceptable. Algunas ideas:

- *Solo los peatones en la "zona peligrosa" hacen WAIT en el semáforo para moverse*. Esto alivia en parte el problema de interbloqueo, pero no lo soluciona. Por lo que no es aceptable.
- *Espera semiocupada*. Con la versión `IPC_NOWAIT` de `msgrcv`, si estoy en la zona peligrosa, compruebo si hay mensaje esperándome. Si no es así, salgo de la sección crítica un rato y lo reintento más tarde. Ya sabéis que la espera semiocupada no es una buena solución y, aunque vale, os baja la nota.
- *Regla de la zona de peligro*: mientras haya un proceso en la zona de peligro, no permitimos que nazca ninguno (mediante un semáforo). Esta solución, aunque presenta inconvenientes

evidentes, se aceptará como válida.
- *La mejor solución*. Algún protocolo que ideéis que solucione los problemas anteriores. Por ejemplo, "los lugares de la zona peligrosa tendrán asociados tres tipos de mensajes, el normal, el bis y el tris. Cuando un peatón nazca, trata de de reservar su posición leyendo el mensaje normal. Si lo hace, estupendo. Si no es capaz, manda a un mensaje bis al buzón, sale de la sección crítica y se queda recibiendo un mensaje tris. Los procesos que estén avanzando en la zona peligrosa, después de moverse, antes de liberar la posición antigua, miran a ver si hay algún mensaje bis correspondiente a esa posición. Si no lo hay, liberan normalmente. Si lo hay, liberan enviando un mensaje tris, para que uno de los que estén esperando lo lea y nazca.

XVIII. La biblioteca almacena en un `char` el número de procesos creados, por lo que si ponéis más de 127 procesos se produce desbordamiento y os dará "Número de procesos negativo" sin razón

XIX. En el caso de que no hagáis la parte de los peatones, la forma correcta de hacerlo es llamar siempre a la función `CRUCE_nuevo_proceso` y descartar las veces que sale peatón, es decir, no hacer nada en esos casos y reintentar.

XX. 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.

XXI. No se debe dormir (es decir, ejecutar `sleep`s o `pausa`s) 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.

XXII. La zona peligrosa del cruce que debe estar libre antes de pasar de fase en el ciclo semáforico puede variar. Por ejemplo, en el caso del paso de la fase segunda a la tercera, no pueden quedar coches en todo el cruce, pues se llevarían por delante a los peatones de P1.

XXIII. La biblioteca no reintenta los WAITs de su semáforo, por lo que, de recibirse una señal, podría fallar. En principio, esto no ha de ocurrir, puesto que la única señal en juego durante la ejecución es el SIGCLD que se envía al padre cuando van muriendo los hijos y la única función que el padre ha de llamar es `CRUCE_nuevo_proceso` y esta función no hace uso del semáforo. En cualquier caso, si os da problema, simplemente ignorad la señal SIGCLD en el padre como se explica más arriba.
