PRÁCTICAS DE SISTEMAS OPERATIVOS II

PRIMERA PRÁCTICA EVALUABLE

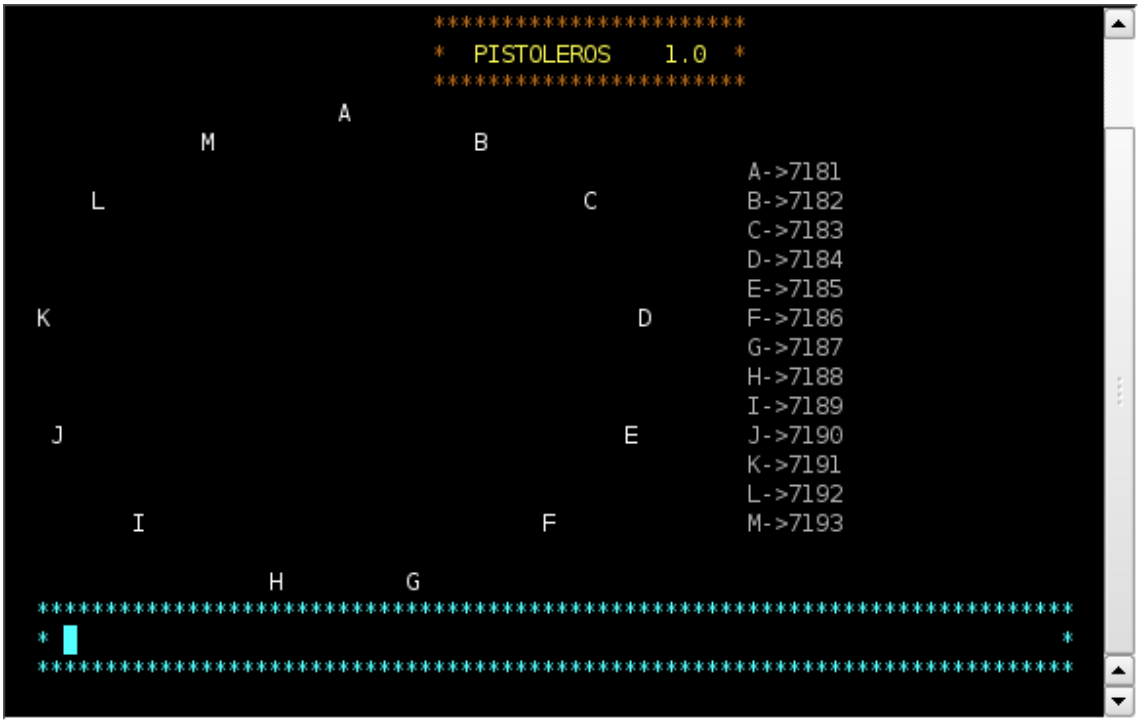
Pistoleros de Wisconsin

---

1. Enunciado.

Antes de estudiar esta asignatura hubimos de sufrir la falta de medios para comunicar adecuadamente a los procesos entre sí. En esta práctica pondremos remedio a la situación permitiendo usar los nuevos mecanismos IPC recientemente aprendidos. Se tratará de simular el problema del pistolero visto en la segunda práctica de Sistemas Operativos I, con ligeras modificaciones.

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, `pist2.c`, cuya adecuada compilación producirá el ejecutable `pist2`. Respetad las mayúsculas/minúsculas de los nombres.

Para simplificar la realización de la práctica, se os proporciona una biblioteca estática de funciones (`libpist.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 pist2.c libpist.a -o pist2 -lm
```

Disponéis, además, de un fichero de cabeceras, `pist2.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 `PIST_inicio` con los parámetros adecuados. El proceso será responsable de crear los procesos pistoleros (un proceso hijo por cada pistolero que participe en la primera ronda) y de controlar que, si se pulsa CTRL+C, la práctica acaba, no dejando procesos en ejecución ni recursos IPCs sin borrar. La práctica devolverá el número del pistolero que ha sobrevivido ó 0 si no quedó ninguno. En el caso de que se produzca un error, el código que se devolverá será 100.

La práctica se invocará especificando dos parámetros obligatoriamente y uno tercero opcional desde la línea de órdenes. El primer parámetro será el número de pistoleros (entre 2 y 26). 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 tercer parámetro, que es opcional, será el valor de la semilla que la biblioteca usará para generar números aleatorios. Si no se especifica, se usará una

semilla basada en el tiempo del sistema. La razón de este último parámetro es el poder verificar si habéis hecho bien la práctica. Poniendo un valor concreto en la semilla, el resultado será siempre el mismo y, por tanto, verificable.

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. Se declarará un array de semáforos de tamaño adecuado a vuestros requerimientos, el primero de los cuales (el semáforo 0) 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 `libpist.a` son las que a continuación aparecen. De no indicarse nada, las funciones devuelven -1 en caso de error:

- `int PIST_inicio(unsigned int nPistoleros, int ret, int semAforos, char *zona, int semilla)`
  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 `nPistoleros` el número de pistoleros, en `ret` la velocidad de presentación (parámetro de la línea de órdenes) y pasando el identificador del conjunto de semáforos que se usará (que incluye como primer elemento el semáforo reservado para la biblioteca y el resto, los que usáis vosotros) y el puntero a la zona de memoria compartida declarada para que la biblioteca pueda usarlos. El último parámetro es la semilla del generador de números pseudoaleatorios. De no haber sido especificada en la línea de órdenes, pasad un cero.
- `int PIST_nuevoPistolero(char pist)`
  Cada pistolero, una vez ha nacido, debe llamar a esta función

para indicar a la biblioteca la letra identificadora que le ha correspondido ('A' para el primero, 'B' para el segundo, y así).

- `char PIST_vIctima(void)`
  Con esta función, cada pistolero sabe la letra identificadora del pistolero que va a matar. La función devuelve arroba('@') en caso de error.
- `int PIST_disparar(char pist)`
  La función se usa para disparar al pistolero cuya letra identificadora se pasa como argumento.
- `int PIST_morirme(void)`

  Después de que todos los pistoleros han disparado en esa ronda, los pistoleros que han sido alcanzados deben invocar esta función **una única vez** para morirse. La función no mata al proceso. Simplemente le indica a la biblioteca que el proceso ha sido alcanzado por uno o más disparos. Es posible que, después de haber llamado a la función, el proceso tenga que hacer algunas labores antes de morirse como proceso.

- `int PIST_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:

8. Tomará los datos de la línea de órdenes y los verificará.
9. Iniciará las variables, mecanismos IPC, manejadoras de señales y demás.
10. Llamará a la función `PIST_inicio`.
11. Tendrá tantos hijos como pistoleros haya.
12. Se quedará esperando a que los hijos mueran.
13. Llamará a la función `PIST_fin`.
14. Limpiará los procesos, mecanismos IPC, etc. y devolverá el valor adecuado al sistema operativo (cero, el identificador del que quedó vivo ó 100).

El padre, por lo tanto, no participará en la sincronización de los procesos pistoleros.

Por su parte, los hijos pistoleros realizarán lo que sigue:

15. Llamarán a la función `PIST_nuevoPistolero`.
16. Entrarán en un bucle de rondas, donde:
    a. El pistolero de letra menor (menos avanzada en el orden alfabético) de los vivos será el coordinador de la ronda. Si sólo hay uno, debe morir (sin llamar a la función `PIST_morirme`).
    b. Los pistoleros eligen víctima.
    c. Una vez el coordinador avise, dispararán todos a la vez. Para ello, además de llamar a la función `PIST_disparar`, usaréis un buzón de paso de mensajes. Se enviará un mensaje de "MUERTE" al pistolero elegido. No se usarán señales para matar a los procesos, sino que "se suicidarán" una vez hayan recibido uno o más mensajes de tipo "MUERTE".
    d. Cuando todos hayan disparado, los pistoleros que hayan recibido uno o más mensajes de "MUERTE", llamarán a la función `PIST_morirme` y se suicidarán.
    e. Comienza una nueva ronda.

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. 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.

Biblioteca de funciones `libpist.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 `libpist.a` y el fichero de cabecera `pist2.h`. La biblioteca funciona con los códigos de VT100/xterm, por lo que debéis adecuar vuestros simuladores a este terminal.

Ficheros necesarios:

- `libpist.a`: para Solaris (ver 1.0), para el LINUX de clase (ver 1.0),
- `pist2.h`: Para todos (ver 1.0).

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:

0. Crear los semáforos y el buzón, y comprobad que se crean bien, con `ipcs`. Es preferible, para que no haya interferencias, que los definái privados.
1. Registrar SIGINT para que cuando se pulse ^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).

2. Llamar a la función `PIST_inicio` en `main`. Debe aparecer la pantalla de bienvenida y, pasados dos segundos, dibujarse la pantalla.
3. Probad a crear los hijos y que llamen a `PIST_nuevoPistolero` y ved que los PIDs que aparecen en la pantalla coinciden con los suyos.
4. Probad la función `PIST_disparar`
5. Completad el código del padre y el modo en que los hijos le van a decir si han muerto disparados o son el único que ha quedado vivo.
6. La parte fundamental de la práctica consiste en pararse ahora y diseñar los mecanismos de sincronización sobre el papel. Estos mecanismos son fundamentales para el proceso que se realiza en cada ronda.
7. Haced todo el proceso de una ronda, con su sincronización. Con un poco de suerte, la práctica fallará y podéis corregir vuestros errores de sincronización. Si no falla, nunca estaréis seguro de si está bien o simplemente es que no han salido los errores a la luz.
8. Solucionad el problema de que comience una nueva ronda y el caso de que nada más quede un proceso.
9. Acabad la práctica y probadla a velocidad normal y a velocidad cero. Es seguro que, a velocidad cero, pueden salir a relucir problemas ocultos a velocidades menores.
10. Pulid los últimos detalles.

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

. la práctica cumple las especificaciones de este enunciado y,
a. la práctica no falla en ninguna de las ejecuciones a las que se somete y,
b. 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. LPEs.

. ¿Se puede usar la biblioteca en un Linux de 64 bits? Aquí se os indican las claves.
I. ¿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 de 32 bits. A veces podéis lograr encontrar una solución mediante el uso de máquinas virtuales.
II. ¿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 su valor inicial.
III. 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.

IV. 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.

V. 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.

VI. 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.

VII. A muchos, la práctica os funciona exasperantemente lenta en el servidor. 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.

VIII. 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.

IX. 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.

X. 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.

XI. Os recuerdo que, si ponéis señales para sincronizar esta práctica, la nota bajará. Usad semáforos, que son mejores para este cometido.

XII. 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
    ```

7.
