Material permitido: **Calculadora NO programable.** Tiempo: **2 horas.** N	**Aviso 1:** Todas las respuestas deben estar razonadas. **Aviso 2:** Escriba sus respuestas con una letra **lo más clara posible.** **Aviso 3: No use *Tipp-ex*** o similares (atasca el escáner).

ESTE EXAMEN CONSTA DE 5 PREGUNTAS

1. (*1.5 p*). Responda a las siguientes cuestiones:

a) (*0.75 p*). Explique el resultado de ejecutar la orden:

```
$ cat < /etc/host.conf
```

b) (*0.75 p*). Explique el funcionamiento de la siguiente llamada al sistema y el significado de sus parámetros.

```
resultado=nice(incremento);
```

RESPUESTA:

Apartado a)

(Apartado 2.4) La ejecución de la orden muestra por pantalla el contenido de `host.conf`, ya que la orden `cat` muestra por pantalla el contenido de los ficheros que se le pasan como argumentos. Si no se introducen ficheros cómo argumento, realiza la lectura de la entrada estándar. En este caso el fichero `/etc/host.conf` es redirigido a la entrada estándar con `<` de modo que la ejecución del programa es similar a haber invocado directamente `cat /etc/host.conf`, esto es, se muestra por la salida estándar el contenido del fichero `host.conf` situado en el directorio `etc`.

Apartado b)

(Apartado 5.4.1) La llamada al sistema `nice` permite aumentar o disminuir el factor de amabilidad actual del proceso que la invoca (un proceso no puede modificar el factor de amabilidad de otro proceso usando esta orden). Su sintaxis es:

```
resultado=nice(incremento);
```

Donde `incremento` es una variable entera que puede tomar valores entre -20 y 19. El valor de incremento será sumado al valor del factor de amabilidad actual `p_nice`. Sólo el superusuario puede invocar a `nice` con valores de `incremento` negativos. Si se produce un error durante la ejecución de `nice`, entonces `resultado` contendrá el valor -1.

2. (*1.5 p*). Explique razonadamente si las siguientes afirmaciones son verdaderas o falsas:

a) (*0.75 p*). Un proceso hijo puede acceder a una tubería sin nombre creada por su proceso padre.
b) (*0.75 p*). El núcleo puede acceder directamente a los campos del *área U* del proceso que se está ejecutando pero no al *área U* de otros procesos.

RESPUESTA:

Apartado a) Verdadera

(Apartado 6.3.2 a) Las *tuberías sin nombre* se crean invocando a la llamada al sistema `pipe` y solamente pueden ser utilizadas por el proceso que hace la llamada y sus descendientes.

La sintaxis de esta llamada es:

```
resultado=pipe(tubería);
```

Donde `tubería` es un array entero de dos elementos y `resultado` es una variable entera. Si la llamada al sistema se ejecuta con éxito en `resultado` se almacenará el valor 0 y en `tubería` se habrán almacenado dos descriptores de ficheros.

Para leer de la tubería hay que usar el descriptor almacenado en `tubería[0]`, mientras que para escribir en la tubería hay que usar el descriptor almacenado en `tubería[1]`. En caso de error durante la ejecución de `pipe` en `resultado` se almacenará el valor -1.

Cuando un proceso invoca a la llamada al sistema `fork` para crear un proceso hijo éste hereda todos los descriptores de ficheros de su progenitor. Ésta es la razón por la que un proceso hijo puede también acceder a una tubería creada por su progenitor. Este mismo razonamiento se aplica para los descendientes de este proceso hijo. De esta forma, en cada tubería pueden escribir y leer varios procesos relacionados genealógicamente. Cada uno de estos procesos puede escribir o/y leer en la tubería.

Apartado b) Verdadera

(Apartado 7.9) Cada proceso tiene su propia área U. Sin embargo el núcleo accede a ella como si solamente existiera una única área U en todo el sistema, la del proceso actual. El núcleo cambia su mapa de traducción de direcciones virtuales de acuerdo con el proceso que se está ejecutando para acceder al área U correcta.

Cuando se compila el sistema operativo, el cargador asigna a la variable *u* una dirección virtual fija asociada siempre al área U del proceso actual. Luego el núcleo únicamente puede acceder simultáneamente al área U de un cierto proceso, el proceso actual.

3. (*2 p*). Conteste a las siguientes cuestiones:

a) (0.5 p). Defina qué son las hebras de usuario.
b) (1.5 p). Describa sus ventajas e inconvenientes.

RESPUESTA:

Apartado a)

(Apartados 4.9 y 4.9.4).

De forma general un proceso se puede considerar como una entidad compuesta que puede ser dividida en dos componentes: un conjunto de hebras y una colección de recursos. La hebra es un objeto dinámico que representa un punto de control en el proceso y que ejecuta una secuencia de instrucciones. Los recursos (espacio de direcciones, ficheros abiertos, credenciales de usuario, cuotas,...) son compartidos por todas las hebras de un proceso. Además cada hebra tiene sus objetos privados, tales como un contador de programa, una pila y un contador de registro.

Un proceso UNIX tradicional tiene una única hebra de control. Las *hebras de usuario* son objetos de alto nivel no visibles para el núcleo. Pueden utilizar procesos ligeros, si éstos son soportados por el núcleo, o pueden ser implementadas en un proceso UNIX tradicional sin un apoyo especial por parte del núcleo.

Es posible transferir la abstracción de hebra enteramente al nivel de usuario, sin que el núcleo intervenga para nada. Esto se consigue a través de paquetes de librería tales como *C-threats* de Mach y *pthreads* de POSIX. Estas librerías suministran todas las funciones para crear, sincronizar, planificar y gestionar hebras sin ninguna asistencia especial por parte del núcleo. Las interacciones entre las hebras no involucran al núcleo y por lo tanto son extremadamente rápidas.

Apartado b)

(Apartado 4.9.4). Las hebras de usuario poseen varias **ventajas**.

- Suministran una forma natural de programar muchas aplicaciones, como por ejemplo todas aquellas gestionadas mediante ventanas. Además, suministran un **paradigma de programación síncrona**, al ocultar las complejidades de las operaciones asíncronas en la librería de hebras, lo cual las hace muy útiles, incluso aunque el sistema carezca de soporte para ellas. Un sistema puede suministrar varias librerías de hebras, cada una de ellas optimizada para un determinado tipo de aplicaciones.

- La mayor ventaja de las hebras de usuario es su comportamiento. Son computacionalmente **ligeras** y no consumen recursos del núcleo excepto cuando se acotan a único proceso ligero. Son capaces de implementar la funcionalidad a nivel de usuario sin utilizar llamadas al sistema. Esto **evita la sobrecarga** de los cambios de modo. Una noción útil es la de tamaño crítico de una hebra, que indica la cantidad de trabajo que una hebra debe realizar para ser considerada útil como entidad separada. Este tamaño depende de la sobrecarga asociada con la creación y uso de una hebra. Para las hebras de usuario, el tamaño crítico es del orden de unos pocos cientos de instrucciones y puede ser reducido a menos de un centenar con el apoyo de un buen compilador. Las hebras de usuario requieren de mucho menos tiempo para su creación, destrucción y sincronización en comparación con los procesos ligeros y los procesos.

Pero también presentan ciertas **desventajas**:

- las hebras de usuario poseen varias limitaciones, principalmente debidas a la separación total de información entre el núcleo y las librerías de hebras. Puesto que **el núcleo no sabe de la existencia de las hebras de usuario**, no puede usar sus mecanismos de protección para proteger una hebras de otras. Cada proceso tiene su propio espacio de dirección, que el núcleo protege de accesos no autorizados de

otros procesos. Las hebras de usuario **no disfrutan de esta protección**, operan en el espacio de direcciones que es propiedad del proceso. La librería de hebras debe suministrar mecanismos de sincronización, los cuales requieren de la cooperación entre las hebras.

- El modelo de planificación dividida produce problemas adicionales. La librería de hebras planifica las hebras de usuario y el núcleo planifica a los procesos subyacentes o procesos. No existe ningún intercambio de información sobre **planificación** entre ambos. Asimismo, como el **núcleo no conoce las prioridades relativas de las hebras** de usuario, quizás expropie a un proceso ligero que ejecuta una hebra de usuario de alta prioridad para planificar a un proceso ligero que ejecuta una hebra de usuario de baja prioridad.

- Finalmente, **sin el apoyo explícito del núcleo**, las hebras de usuario pueden mejorar la concurrencia, pero **no incrementar el paralelismo**. Incluso en un sistema multiprocesador, las hebras de usuario compartiendo un único proceso ligero no pueden ejecutarse en paralelo.

4. (*2 p*). Escriba un programa en C que cree una zona de memoria compartida privada de tamaño 1024 bytes, donde solo el usuario va a tener permisos de lectura y escritura.

RESPUESTA:

Ejemplo 6.15:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
...
int shmid;
...
shmid=shmget(IPC_PRIVATE,1024,IPC_CREAT | 0600);
if (shmid==-1)
{
/* Error en la creación de la memoria compartida.
Tratamiento del error.*/
```

5. (*3 p*) Conteste razonadamente a los siguientes apartados:

a) (*1.5 p*) Explicar el significado de las sentencias enumeradas (**[ ]**) del siguiente programa.

b) (*1.5 p*) El programa es compilado enlazando la librería `lpthread` y creándose el ejecutable `Examen`. Describir el **funcionamiento** así como la **salida** obtenida salida cuando se invoca la orden "`$ ./Examen`". Supóngase que el sistema asigna PIDs a los procesos consecutivamente comenzando con pid=1000 y que no se produce ningún error.

```
        #include <pthread.h>
        #include <stdlib.h>
        #include <stdio.h>
        #include <string.h>

        char cadena1[]="Problema 5", cadena2[]="incorrecto.";

[1]     void *fun1(void *arg)
        {
[2]     wait();
        printf("PidB=%d\n", getpid());
[3]     strcpy(cadena2,"correcto.");
        }

        void main(void)
        {
        int pid;
[4]     if ((pid=fork())==-1){perror("Fallo en fork:"); exit(1);}
        if (pid==0)
             {
[5]          sleep(1);
             strcpy(cadena1,"Ejercicio imposible");
             printf("PidA=%d\n", getpid());
             }
        else
             {
             pthread_t hilo;
[6]          if (pthread_create(&hilo,NULL,fun1,NULL))
                  {
                  printf("Error creando pthread");
                  exit(1);
                  }
[7]          pthread_join(hilo,NULL);
[8]          printf("PidC=%d\n", getpid());
             printf("%s del examen DyASO %s\n",cadena1,cadena2);
             }
        }
```

RESPUESTA

Apartado a):

[1] Se declara un puntero a función (`fun1`) que admite como entada un puntero nulo `arg`. Dicha función será usada en **[6]** para crear una hebra.

[2] La llamada al sistema `wait()` hace que el proceso padre espere a la finalización del proceso hijo (o a la recepción de otra señal) antes de continuar.

[3] La función `strcpy` sirve para copiar cadenas. En este caso la instrucción `strcpy(cadena2,"correcto.")` se encarga de copiar en la cadena de caracteres `cadena2` (que inicialmente contenía el texto "`incorrecto.`") el nuevo texto texto "`correcto.`".

[4] La llamada al sistema `fork()` crea un proceso hijo al que devuelve 0, mientras que al padre le devuelve el pid del proceso hijo. Este resultado se almacena en la variable `pid`.

En caso de producirse un error devolverá `-1`, cumpliéndose entonces la condición de la cláusula `if`. Esto invocaría a `perror` que imprimirá "`Fallo en fork:`" seguido del descriptor del error almacenado en la variable `errno`. A continuación terminara el proceso con la llamada al sistema `exit()`, devolviendo la condición de retorno 1.

[5] La llamada al sistema `sleep(1)` duerme al proceso que la invoca durante un segundo.

[6] `pthread_create` es una llamada a la biblioteca de hebras posix cuya misión es crear una nueva hebra de ejecución dentro del contexto del proceso que la invoca.

Dicha función recibe como argumentos un puntero al manejador hilo, parámetros de creación de la hebra (en este caso `NULL`), el nombre de la función que iniciará la hebra `fun1` y un vector de parámetros a dicha función (también nulos). Si la llamada es exitosa se creará un nuevo hilo almacenando el manejador en la variable `hilo`.

[7] La orden `pthread_join(hilo,NULL)` espera a la terminación del hilo creado en **[6].**

[8] `printf("PidC=%d\n", getpid());` imprime por salida estándar el texto "`PidC=`" seguido del pid del proceso que la invoca que es obtenido mediante la llamada al sistema `getpid()`. A continuación realiza un salto de línea.

Apartado b)

El programa posee dos variables globales `cadena1` y `cadena2` que son arrays de caracteres inicializados inicialmente a los valores "`Problema 5`" e "`incorrecto.`" Respectivamente.

Cuando el usuario invoca la orden "`$ ./Examen`" comienza la ejecución del programa a partir de la función principal `main()`. De acuerdo al enunciado, el pid asignado por el SO al proceso será 1000.

A continuación, se crea una variable `pid` de tipo entero y seguidamente se realiza una llamada al sistema `fork()` para crear un proceso hijo. `Fork` devolverá el pid del hijo al proceso padre y 0 al proceso hijo. Terminada dicha llamada al sistema, dado que no se produce ningún error, existen dos procesos Padre e Hijo con pids 1000 y 1001 respectivamente.

El proceso hijo ejecutará el código contenido en el bloque de código de la cláusula `if` siguiente mientras que el padre continuará su ejecución en la cláusula `else`.

El proceso hijo es una copia idéntica al proceso padre de modo que hereda **una copia** de las variables globales `cadena1` y `cadena2`. En la primera instrucción el proceso hijo se duerme durante un segundo.

Una vez que despierte modificará **su copia** de la variable `cadena1`, lo que **no tiene ningún e**fecto sobre la variable de nombre similar que existe en el contexto de ejecución **del proceso padre**. A continuación imprime `PidA=1001` termina.

Por su parte, el proceso padre crea una nueva hebra de ejecución cuyo código es el de la función `fun1` definida al comienzo del programa. La hebra se ejecuta concurrentemente con el proceso que la ha creado pero, a diferencia del proceso hijo, la hebra hija se ejecuta **en el mismo contexto** que la hebra padre. De este modo el proceso padre consistirá en dos hebras, la hebra padre original H1 y la nueva hebra H2 que denominaremos hebra hija.

Esto significa que cuando se ejecute dicha hebra hija H2 imprimirá por salida estándar `PidB=1000` (Nótese que la hebra padre y la hebra hija **son el mismo proceso** y por lo tanto **tienen el mismo pid)**.

A continuación la hebra hija modificará el valor de la variable global `cadena2` asignándole el valor "`correcto.`". Este cambio, al contrario que en el caso, anterior **será visible para la hebra padre ya que las hebras** comparten las variables globales.

Finalmente la hebra padre H1 **esperará a que termine** la hebra hija H2 (mediante `pthread_join`) así como el proceso hijo (mediante `wait`). Seguidamente imprimirá por la salida estándar el texto "`PidC=1000`", seguido de un salto de línea y el texto "`Problema 5 del examen DyASO correcto.`" Imprimiendo finalmente un último salto de línea.

Es importante destacar que **sea cual sea el orden** en el que el SO planifique los procesos y sea cual sea el orden **de planificación** de los hilos dentro del proceso padre, **la salida** del programa **siempre es la misa. El programa espera un segundo y a continuación imprime la siguiente salida**:

```
PidA=1001
PidB=1000
PidC=1000
Problema 5 del examen DyASO correcto.
```

Esto es así puesto que la hebra padre H1 no podrá imprimir nada, ni terminar su ejecución, hasta que termine la hebra hija H2 (ya que está esperando por ella debido a `pthread_join`). Del mismo modo, la hebra H1 no imprime nada ni termina hasta que lo haga el proceso hijo, ya que se encuentra esperando la terminación del mismo (`wait`).