Material permitido: **Calculadora NO programable.** Tiempo: **2 horas.** N1	**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. (*2 p*) Explique **razonadamente** si las siguientes afirmaciones son verdaderas o falsas:

a) (*1 p*) El conmutador `struct bdevsw` es un conmutador para dispositivos modo bloque.

b) (*1 p*) La creación de un proceso ligero requiere el uso de llamadas al sistema.

RESPUESTA:

a) VERDADERO. Un *conmutador de dispositivo* es una estructura de datos que define puntos de entrada para cada dispositivo que debe soportar. Existen dos tipos de conmutadores: `struct bdevsw` para dispositivos modo bloque y `struct cdevsw` para dispositivos modo carácter. (*Ver 9.4.3 Los conmutadores de dispositivos)*

b) VERDADERO. Los procesos ligeros tienen algunas limitaciones. La mayoría de las operaciones de los procesos ligeros, tales como creación, destrucción y sincronización, requieren del uso de llamadas al sistema (*Ver. 4.9.3 Procesos ligeros*)

2. (*1.5 p*) Explique **razonadamente** la realización de la llamada al sistema `fork` en un sistema con una política de gestión de memoria por demanda de página.

RESPUESTA:

(Ver Apartado 7.4 Fork() en un sistema con paginación)

Al atender la llamada al sistema `fork`, el núcleo duplica cada región del proceso padre y se la asigna al proceso hijo. Tradicionalmente, el núcleo en un sistema con intercambio hace una copia física del espacio de direcciones del padre para asignársela al proceso hijo.

En el sistema de paginación del System V, el núcleo **evita realizar la copia** del espacio de direcciones del padre mediante la adecuada manipulación de la tabla de regiones, las tablas de páginas y la tabla *dmp*. El núcleo simplemente incrementa el contador de referencias en la tabla de regiones de las regiones compartidas (como por ejemplo la región de código) por el proceso padre y el proceso hijo.

Para regiones privadas tales como la región de datos o la de pila, sin embargo, el núcleo asigna una nueva entrada de la tabla de regiones y una nueva tabla de páginas y después examina cada entrada de la tabla de páginas del padre.

- Si una página es válida, incrementa el contador de referencias ubicado en la entrada de la tabla *dmp*, que indica el número de procesos que comparten la página a través de diferentes regiones (en oposición al número de procesos que comparten la página por compartir la región).

- Además, si la página existe en un dispositivo de intercambio, incrementa el contador de entradas de la tabla de intercambio.

La página ahora puede ser referenciada a través de ambas regiones, que comparten la página hasta que un proceso la escriba. En dicho caso el núcleo entonces copia la página para que cada región tenga una copia privada. Para poder proceder de este modo, el núcleo activa el bit *copiar al escribir* en cada entrada de la tabla de páginas asignada a una región privada del padre y del hijo durante fork.

Si un proceso escribe una página, provocará un fallo de protección. Cuando se trate el fallo, el núcleo hará una nueva copia de la página para el proceso que provocó el fallo. La copia física de la página es así aplazada hasta que un proceso realmente la necesita.

3. (*1.5 p*) Responda razonadamente a las siguientes cuestiones:
a) (*0.5 p*) ¿Qué forma el contexto a nivel de usuario?
b) (*1 p*) ¿Qué contenido tiene el contexto a nivel de registros?

RESPUESTA
(Ver Apartado 3.6 Contexto de un proceso)

a) El contexto a nivel de usuario de un proceso está formado por su *código*, *datos*, *pila de usuario* y *memoria compartida* que ocupan el espacio de direcciones virtuales del proceso.

b) El contexto de registros de un proceso está formado por el contenido de los siguientes registros de la máquina:

- El **contador del programa** que indica la dirección de la siguiente instrucción que debe ejecutar la CPU. Esta dirección es una dirección virtual del espacio de memoria del núcleo o del usuario.

- El **registro de estado** del procesador que indica el estado del hardware de la máquina en relación al proceso en ejecución. Contiene diferentes campos para almacenar la siguiente información: el modo de ejecución, el nivel de prioridad de interrupción *npi*, el indicador de rebose, el indicador de arrastre, etc.

- El **puntero de la pila** donde se almacena la dirección virtual, dependiendo de la arquitectura de la máquina, de la próxima entrada libre o de la última utilizada en la pila de usuario (ejecución en modo usuario) o en la pila del núcleo (ejecución en modo núcleo). Análogamente, la máquina indica la dirección de crecimiento de la pila, hacia las direcciones altas o bajas.

- Los **registros de propósito general**, que contienen datos generados por el proceso durante su ejecución.

4 *(2p)* Conteste razonadamente a los siguientes apartados:
a) *(1.5 p)* ¿Qué concepto se implementa a partir de las siguientes líneas de código? Explique para qué sirve, sus posibles usos y las principales desventajas.

```
/*implementación de init_sem, wait_sem  y signal_sem */
    …
    semid=semget(IPC_PRIVATE,1,IPC_CREAT|0600);
    init_sem(semid,0);
    if ((pid=fork())==-1) exit(1);
    if (pid==0)
         {
         wait_sem(semid) ;
         printf("Hijo acaba\n");
         }
    else
         {
         printf("Padre empieza\n");
         sleep(5);
         signal_sem(semid);
         wait(&status);
         printf("Padre acaba\n");
         }
    …
```

b) *(0.5 p)* Describe el comportamiento de las funciones que son invocadas en el código anterior.

RESPUESTA:

(Ver Apartado 6.7.1 Semáforos; 6.4.2, Ej. 6.10)

A) El concepto que se implementa a través del código es el de los semáforos, uno de los mecanismos de sincronización.

Un semáforo *semáforo* genérico es un objeto que puede tomar valores enteros que soporta dos operaciones atómicas: `P()` y `V()`. La operación `P()`, que en adelante denominaremos `wait_sem()` decrementa en una unidad el valor del semáforo y bloquea al proceso que solicita la operación si su nuevo valor es menor que cero. La operación `V()`, que llamaremos `signal_sem()` incrementa en una unidad el valor del semáforo; si el valor resultante es mayor o igual a cero, `V()` despierta a los procesos que estuvieran esperando por este evento. El núcleo garantiza que las operaciones sobre el semáforo serán atómicas, incluso en sistemas multiprocesador.

Tiene los siguientes usos:

1. ***Exclusión mutua*** sobre un determinado recurso. Para ello se debe asociar un semáforo a un recurso compartido e inicializarlo a uno. Cada proceso realiza una operación `wait_sem` para adquirir el uso del recurso en exclusiva y una operación `signal_sem` para liberarlo.

2. También es posible utilizar un semáforo para implementar la ***espera por un determinado evento***. En este caso el semáforo se debe inicializar a cero. El proceso que haga una operación `wait_sem` se bloqueará. Cuando se produzca el evento se hará una operación `signal_sem`. **Esta es la operación implementada en el código de este ejercicio**.

3. Los semáforos también resultan útiles **administrar un cierto recurso limitado**, es decir, un recurso con un número fijo de instancias inicializando el semáforo al número de instancias. Esta es una solución natural al clásico problema de los ***consumidores-productores***.

Las principales desventajas son:

Aunque los semáforos suministran una abstracción simple suficientemente flexible para tratar diferentes tipos de problemas de sincronización, poseen algunos inconvenientes que hacen que su uso no sea adecuado en ciertas situaciones.

En primer lugar, un semáforo es una abstracción de alto nivel basada en primitivas de bajo nivel que suministran la atomicidad y los mecanismos de bloqueo. Para que las operaciones `signal_sem` y `wait_sem` sean atómicas en un sistema multiprocesador, **debe haber una operación atómica** para garantizar el acceso exclusivo a la propia variable semáforo.

En segundo lugar el bloqueo y el desbloqueo de procesos implican la realización **de cambios de contexto** y la manipulación de las colas del planificador y de las colas de procesos dormidos, lo cual hace que sean operaciones **lentas**. Esto puede ser aceptable para algunos recursos que necesitar ser retenidos durante un periodo de tiempo largo, pero resulta inaceptable si los recursos sólo van a estar retenidos brevemente.

Finalmente, la abstracción de semáforo también **oculta información** sobre si un proceso ha tenido que bloquearse definitivamente en una operación `wait_sem`.

B) El comportamiento de las funciones invocadas en las líneas de código es el siguiente:

En la función principal se crea un semáforo privado y se crea un proceso hijo. A continuación el hijo espera (wait_sem) a que el semáforo esté abierto y después imprime un mensaje y termina.

El padre por su parte duerme durante 5 segundos, abre el semáforo lo que permite que la ejecución de su hijo continúe. Hecho esto espera a que el hijo termine, imprime un mensaje y cierra el semáforo. La ejecución del programa es la siguiente:

```
Padre empieza      (pasan 5 segundos)
Hijo acaba
Padre acaba
```

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

a) (2 p) Explicar el significado de las sentencias enumeradas (**[ ]**) del programa que se muestra en la página siguiente.

b) *(1 p)* Explicar el funcionamiento del programa si se compila en el ejecutable `programa`, se invoca con la orden `./programa OSAyD` y no se produce ningún error.

NOTA: `fflush(stdout)` vacía el buffer de salida estándar, forzando una impresión inmediata de su contenido. Supóngase asemás que el `pid` del proceso creado es 1000 y que el resto de `pids` se asignan secuencialmente.

La pregunta continua en la página siguiente

```
        #include <stdio.h>
        #include <stdlib.h>
        #include <sys/wait.h>
        #include <unistd.h>
        #include <string.h>

        void manejador(int sig) { }

        int main(int argc, char *argv[]){

        int i, pid, tuberia[2];
        int tam=strlen(argv[1]);
        char texto[]=" ";

[1]     if (argc!=2){printf("Argumentos incorrectos\n"); exit(1);}
[2]     if(pipe(tuberia)==-1)
             {
             perror("Error pipe:");
             exit(2);
             }

[3]     signal(SIGALRM,manejador);

        for(i=0;i<tam;i++) //bucle que crea más hijos
             {
[4]          if ((pid=fork())==-1) {perror("Error en fork:"); exit(3);}
             if(pid==0)
                  {
[5]               printf("Hijo %i, PID=%d\n",i,getpid()); fflush(stdout);
[6]               close(tuberia[0]);
[7]               alarm(i+1);
[8]               pause();
[9]               write(tuberia[1],argv[1]+tam-1-i,1);
                  close(tuberia[1]);
                  exit(0);
                  printf("incorrrecto ");
                  }
             }

[10]    sleep(1);
        printf("¡Ejercicio %d ",tam);fflush(stdout);
        close(tuberia[1]);
[11]    while (read(tuberia[0],&texto,1)>0)
             {
             printf("%s",texto);fflush(stdout);
             }
        close(tuberia[0]);
        printf(" corrrecto! \n");
        }
```

RESPUESTA

APARTADO A):

[1] Dicha sentencia comprueba si el número de argumentos pasado a la función es 2, en caso contrario termina el proceso con la llamada al sistema `exit` y el código de retorno 1.

[2] La llamada al sistema `pipe()` crea una tubería sin nombre. Si se produce un error devuelve -1, ejecutándose el bloque de código a continuación invoca `perror` que muestra mensaje de error "Error en pipe:" seguida de la descripción textual del código de error almacenado en la variable `erno`. Seguidamente termina el programa con el código de salida 2. Si no hay ningún error, coloca en el array tubería los manejadores que permiten leer y escribir en la tubería recién creada.

[3] la llamada al sistema `signal` asocia la función `manejador` a la recepción de la señal `SIGALARM`.

[4] La llamada al Sistema `fork()` crea un proceso hijo que es una copia idéntica del proceso padre. Al proceso padre le devuelve el `pid` del hijo mientras que al proceso hijo le devuelve 0. En caso de error, `fork` devuelve -1, de tal modo que se ejecuta la cláusula `if`, se imprime "`Error fork`" y se termina el proceso con la llamada al sistema `exit` y el código de retorno 3.

[5] `printf` Imprime por pantalla "`Hijo`" seguido del número de hijo `i` así como "`PID=`" seguido del su pid que obtiene mediante la llamada al sistema `getpid`.

[6] La llamada al sistema `close` cierra el extremo de lectura de la tubería (`tuberia[0]`).

[7] La llamada al sistema `alarm` establece una alarma de tiempo real, su argumento es el tiempo de espera en segundos. En este caso se planifica una alarma transcurridos `i+1` segundos.

[8] La llamada al sistema `pause` bloquea la ejecución del proceso que la invoca hasta que reciba cualquier señal que no ignore o que no tenga bloqueada.

[9] `white` es una llamada al sistema que escribe un byte en el extremo de entrada de la tubería (`tuberia[1]`) desde el buffer `argv[1]+tam-i`. Puesto que `argv[1]` es un puntero al segundo argumento de entrada de la función y además `tam` es el tamaño de la cadena `argv[1]` calculado previamente, entonces `argv[1]+tam-i` apuntará al byte i-ésimo de la cadena `argv[1]` **empezando por el final**.

[10] `sleep` es una llamada al Sistema que duerme al proceso que la invoca. En este caso el proceso duerme durante un segundo.

[11] `read` es una llamada al sistema que lee 1 byte (tercer argumento) del extremo de lectura de la tubería (`tuberia[0]`) y lo almacena en la cadena `texto`. Si la lectura es correcta devuelve el número de bytes leídos (que ha de ser 1). Dicha lectura se utiliza como condición de un bucle `while` comprobando que se ha conseguido leer de la tubería (el retorno es >0). Esto significa que el bucle `while` se ejecuta mientras sea posible leer de la tubería.

APARTADO B:

El funcionamiento del programa es el siguiente:

En primer lugar, se define una función `manejador` que se asociará posteriormente con la señal `SIGALRM`. Dicho manejador se encuentra vacío y por lo tanto no se realiza ninguna acción, pero su presencia evita que al recibir la señal `SIGALRM` se termine el proceso en curso.

Hecho esto, la función principal `main` define las variables necesarias para el programa y calcula el tamaño `tam` de la segunda entrada (`strlen`). Puesto que en UNIX el primer argumento de entrada es el nombre del programa, la segunda entrada será `OSAyD`, luego `tam=5`.

Hecho esto se crea una tubería sin nombre `tuberia`, y se establece el manejador de la señal `SIGALRM` sin que se produzca ningún error.

A continuación, se entra en un bucle que va creando `tam` procesos hijos. En cada iteración del bucle se realiza una llamada al sistema `fork`. El hijo recién creado H_i ejecuta el bloque `if` que hay a continuación, mientras que el proceso padre pasa directamente a la siguiente iteración del bucle. El resultado es que se crean 5 procesos hijos H_0… H_4.

Cada uno de los procesos hijos, al crearse, imprime inmediatamente por pantalla (`fflush`) su número de hijo `i` así como su PID que según el enunciado es `1000+i+1` y cierra el extremo de lectura de la tubería. Dado que el orden de planificación de los procesos es arbitrario, la salida por lo general será **desordenada**, por ejemplo:

```
Hijo 4, PID=1005
Hijo 2, PID=1003
Hijo 1, PID=1002
Hijo 0, PID=1001
Hijo 3, PID=1004
```

Hecho esto, cada hijo H_i programa una alarma de tiempo real para `i+1` segundos después y queda a la espera de recibir dicha alarma con `pause`.

Transcurridos los `i+1` segundos despertará el hijo i-esimo que escribirá el byte i-esimo empezando por la cola del argumento de entrada "`OSAyD`". Esto es, H_0 se despertará al cabo de un segundo y escribirá "`D`", H_1 despertará al cabo de dos segundos y escribirá "`y`", y así sucesivamente hasta que H_0 escriba "`O`". Por tanto los procesos hijos irán escribiendo en secuencia la palabra "`DyASO`" en la tubería a un ritmo de un carácter por segundo.

Una vez que cada proceso concluye la escritura, cierra el extremo abierto de la tubería y termina normalmente con `exit(0)`. De este modo la sentencia `printf` siguiente no se ejecuta y por lo tanto **no** se imprime "`incorrecto `".

El proceso padre, una vez terminado el bucle de creación de procesos hijos espera un segundo y escribe por pantalla:

```
¡Ejercicio 5
```

Seguidamente cierra el extremo de escritura de la tubería y entra en un bucle de lectura que va leyendo carácter a carácter el contenido de la tubería e imprimiéndolo inmediatamente por pantalla. De este modo imprimirá la palabra `DyASO` a un ritmo de un carácter por segundo.

Hecho esto cerrará la tubería e imprimirá el texto “ `correcto!`”. De este modo la salida completa del programa será similar a la siguiente:

```
Hijo 4, PID=1005
Hijo 2, PID=1003
Hijo 1, PID=1002
Hijo 0, PID=1001
Hijo 3, PID=1004
¡Ejercicio 5 DyASO corrrecto!
```

NOTA: Dependiendo del orden de planificación de los procesos, el orden de los mensajes iniciales impresos por los hijos no puede determinarse. No obstante, la salida “¡Ejercicio 5 DyASO corrrecto!” **siempre es posterior** a la escritura de todos los hijos, ya que el padre espera un segundo, dando tiempo a que todos los hijos se planifiquen. Además, los caracteres siempre se leen en el mismo orden en el que se escriben debido a la sincronización proporcionada por la tubería.

Finalmente, es importante destacar que el proceso padre no tiene que esperar explícitamente con `wait` a la terminación de sus hijos. Esta espera es implícita: mientras que quede algún hijo sin cerrar el extremo de lectura de la tubería, el proceso padre continuará esperando a leer datos con `read`. Cuando el último hijo cierre el extremo de lectura de la tubería ya no habrá más procesos escritores y el padre recibe un EOF, de modo que la lectura termina. Por este motivo es fundamental que el padre cierre el extremo de escritura en la tubería antes de entrar en el bucle (para no quedar esperando indefinidamente).