SOA (10/04/2025)

Primer control de laboratorio

Crea un fichero que se llame "respuestas.txt" donde escribirás las respuestas para los apartados de los ejercicios del control. Indica para cada respuesta, el **número de ejercicio y el número de apartado** (por ejemplo, 1.a, 1.b, ...).
Importante: para cada uno de los ejercicios tienes que partir del código suministrado de Zeos.

1. *(3 puntos)* Comprensión de Zeos

a) (1 punto) ¿En la pila de qué proceso nos encontramos al recibir la 1ª interrupción de reloj? ¿Cómo lo has hallado?
b) (1 punto) Indica los cambios en el Makefile necesarios para desactivar la optimización del código de usuario.
c) (1 punto) Crea un fichero en assembler con una función en ensamblador que devuelva el contenido del registro MSR pasado como parámetro. Para ello puedes usar la instrucción assembler `rdmsr` que según el manual de Intel "*Reads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX*". Indica **la línea (o líneas) de comandos** necesarias para compilar y linkar este objeto en ZeOS.

2. *(3 puntos) Where are my args?*

Queremos cambiar el paso de parámetros en las llamadas a sistema y, para evitar la copia en los registros, dejar los parámetros en la pila de usuario. Dado que ahora los parámetros se encontrarán en la pila de usuario, habrá que apilarlos explícitamente en la pila de sistema justo antes de hacer la llamada a la rutina de servicio que corresponda. Como el número de parámetros a apilar depende de cada llamada usaremos una rutina auxiliar en ensamblador para ayudarnos, que se llamará *sys_XXX_wrap*, donde XXX corresponde a la llamada en cuestión. Por ejemplo, la rutina *sys_write_wrap* debe, usando el registro EBP, apilar los parámetros necesarios(3), hacer la llamada a sistema real (a *sys_write*) y retornar. Además, dado que según la convención de llamadas a función en C, no es necesario mantener el valor de los registros ECX y EDX al volver de una función, queremos simplificar el código existente. Implementa esta simplificación y el nuevo mecanismo de paso de parámetros.
En particular, tienes que:

a) Simplificar la gestión de los registros ECX i EDX en los *wrappers*.
b) Modificar el código para realizar el paso de parámetros mediante la pila de usuario.
c) Implementar el código de la rutina *sys_write_wrap.*
d) Usa las rutinas auxiliares donde corresponda en la tabla *sys_call_table*. Indica si seria necesario crear más rutinas auxiliares.

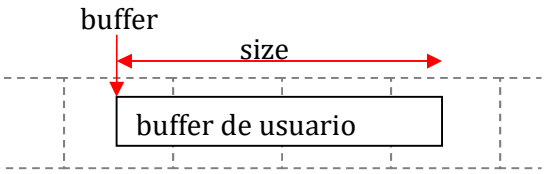
NOTA: La corrección de este ejercicio será binaria, de forma que todos los apartados tienen que estar correctos para conseguir la puntuación máxima. Caso contrario, el ejercicio tendrá una puntuación de 0.

3. (4 puntos) Peeping...

Queremos crear una nueva funcionalidad de sistema que nos permita consultar la memoria de otro proceso. En particular queremos que esta funcionalidad tenga la siguiente interficie:

```
int peep(int pid, char* src, int size, char* dst)
```

Que copiará ´size´ bytes a partir de la dirección ´src´ del espacio de direcciones del proceso ´pid´ al buffer de usuario ´dst´ en el proceso actual (este buffer debe tener espacio suficiente para guardar el contenido a copiar). Recuerda que la dirección del buffer puede ser cualquiera, y en particular no tiene por que estar alineada a página:



Con esta funcionalidad puedo implementar una sincronización básica como esta que muestra "AFTER" sí, y solo sí, se ha ejecutado el código del hijo [siempre y cuando la optimización del código de usuario esté deshabilitada]:

```
int var = 0;
int p = fork();
if (p == 0) {
    var = 42;
    while(1);
}
int fillvar = 0;
while(fillvar == 0) {
    peep(p, &var, sizeof(int), &fillvar);
}
write(1,"AFTER", 5);
```

a) (0.5 puntos) Programa la comprobación de parámetros de la rutina *sys_peep*.
b) (1 punto) Añade el código para buscar el proceso con identificador ´*pid*´.
c) (2 puntos) Añade el código para hacer la copia.
d) (0.5 puntos) ¿Cambiaría el comportamiento del programa mostrado si cambiamos el bucle infinito por una llamada ´*exit*´?

4. (1 punto) *Generic Competences Third Language (Development Level: mid)*

The following paragraph belongs to Wikipedia:

"A paged MMU also mitigates the problem of external fragmentation of memory. After blocks of memory have been allocated and freed, the free memory may become fragmented (discontinuous) so that the largest contiguous block of free memory may be much smaller than the total amount. With virtual memory, a contiguous range of virtual addresses can be mapped to several non-contiguous blocks of physical memory; this non-contiguous allocation is one of the benefits of paging.

However, paged mapping causes another problem, internal fragmentation. This occurs when a program requests a block of memory that does not cleanly map into a page, for instance, if a program requests a 1 KB buffer to perform file work. In this case, the request results in an entire page being set aside even though only 1 KB of the page will ever be used; if pages are larger than 1 KB, the remainder of the page is wasted. If many small allocations of this sort are made, memory can be used up even though much of it remains empty."

Create a text file named "generic.txt" and answer the following questions (since this competence is about text comprehension, you can answer in whatever language you like):

1.- Does ZeOS suffer from external fragmentation?
2.- Does ZeOS suffer from internal fragmentation?
3.- What is the ideal page size for the code section of a process in ZeOS to prevent internal fragmentation if present?

5. Entrega

Sube al Racó los ficheros "respuestas.txt" y "generic.txt" junto con el código que hayas creado en cada ejercicio. Para entregar el código utiliza:

```
> tar zcfv examen.tar.gz zeos
```