Nombre:
DNI:

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 numero de apartado (por ejemplo, 1.a).
Justifica brevemente todas tus respuestas. Una respuesta sin justificar se considerará como no contestada.

Importante: para cada uno de los ejercicios tienes que partir de la versión de Zeos original que te hemos suministrado.

Queremos añadir nuevas funcionalidades para bloquear y desbloquear procesos. Para ello tienes que crear dos llamadas a sistema nuevas:

void block(void);

Esta función bloquea al proceso actual hasta que algún otro proceso lo desbloquee. Es decir, añade el proceso en una cola de procesos bloqueados y llama a la función del interfaz de planificación para poner otro proceso a ejecutar.

int unblock(void);

Esta función desbloquea al primer proceso bloqueado, devolviendo 0 si lo ha conseguido o -1 en caso de que no hubiera ningún proceso bloqueado. Es decir, coge el primer proceso bloqueado y lo añade a la cola de procesos pendientes de ejecución.

1. *(5 puntos + 0,5 puntos) Mecanismos Entrada al Sistema*

Para añadir estas funcionalidades, **NO debes modificar nada de la implementación actual del mecanismo de llamadas a sistema**, sino replicar esta implementación para gestionar estas nuevas funcionalidades. Tienes que usar la interrupción 0x81 para generar estas nuevas llamadas a sistema. Los identificadores serán 1 y 2 para las llamadas *block* y *unblock* respectivamente.

a) (*0.25 puntos*) Indica qué estructura de ZeOS tienes que modificar para implementar esta nueva funcionalidad.
b) (*0.25 puntos*) Indica qué nuevas estructuras de ZeOS tienes que añadir.
c) (*0.25 puntos*) Escribe el código de ZeOS para activar esta nueva funcionalidad.
d) (*0.75 puntos*) Escribe el código del wrapper para la función *block*.
e) (*0.75 puntos*) Escribe el código del handler para esta funcionalidad.

f) (*0.5 puntos, Opcional*) Implementa en ZeOS los apartados anteriores.

g) (*1 punto*) El siguiente fragmento de código de usuario intenta ejecutar el código del hijo antes que el del padre:

```
p = fork();
if (p==0) {
        ... [codigoHijo]...
        unblock();
        exit();
}
block();
...[codigoPadre]...
```

Describe la secuencia de ejecución de los procesos suponiendo una política de planificación FIFO.

h) (*0.5 puntos*) ¿Pueden servir estas nuevas llamadas para sincronizar más de dos procesos? ¿Por qué?

i) (*1 punto*) Dado el fragmento de código del apartado anterior, con una política Round Robin. El padre ejecuta el *fork* y justo a continuación se le termina el quantum, pasando la ejecución al proceso hijo ¿Qué problema se puede encontrar el proceso padre? Describe la secuencia de ejecución de los procesos.

j) (*0.25 puntos*) ¿Cómo puedes arreglar el problema anterior modificando la implementación de block y/o unblock propuesta?

2. *(4 puntos) Gestión de procesos y memoria*

Modifica el código del fork para que el proceso padre use solamente la región de memoria 0x200000-0x210000 para acceder a la zona de datos del hijo y realizar la herencia de datos de usuario, en lugar de usar la región consecutiva a la zona de datos del padre.

a) (*0.5 puntos*)¿Cuantas páginas tiene esta región temporal?(SIZE)

b) (0.5 puntos)¿Cual es la primera página de datos de un proceso de usuario? (START_DATA)

Suponiendo que el bucle de copia fuera tal que así:

```
for(i=0; i < NUM_PAG_DATA; i++) {
        set_ss_page(...[1]...
        copy_data(...[2]...
        del_ss_page(...[3]...
        ...[4]...
}
```

Muestra como seria el código para hacer esta copia de forma eficiente. Puedes usar las variables SIZE y START_DATA de los apartados anteriores, así como usar otras

variables/funciones de ZEOS o nuevas que creas convenientes siempre y cuando lo indiques y muestres su inicialización.

c) (*0.75 puntos*) Indica el código de [1]
d) (*0.75 puntos*) Indica el código de [2]
e) (*0.75 puntos*) Indica el código de [3]
f) (*0.75 puntos*) Indica el código de [4]

3. *(1 punto) Cambio de contexto*

La rutina *inner_task_switch* es la encargada de realizar el cambio de contexto, ahora bien, nos damos cuenta que si modificamos el código ensamblador para que que quede así (usando un único bloque de ensamblador inline en lugar de 4):

```
...
__asm__ __volatile__(
        "movl %%ebp, %0\n\t"
        "movl %1, %%esp\n\t"
        "popl ebp\n\t"
        "ret\n\t"
        : "=g" (current()->register_esp)
        : "g" (new->task.register_esp) );
```

El cambio de contexto deja de funcionar.

a) ¿Por qué?
b) ¿Cómo lo has averiguado?