Apellidos, Nombre: _____ Nº Matrícula: _____ D.N.I.: _____

Sistemas Operativos – Gestión de Memoria – GII & GMI

19 de Mayo de 2014

Dispone de 50 minutos. Las notas saldrán el Miércoles 28 de Mayo. La revisión será el día Viernes 30 de Mayo a las 11.00 en la sala Multiusos del DATSI (D4200, Vestíbulo de la planta 2ª del Bloque IV)

Cuestiones (conteste en el espacio reservado al efecto)

1) **[1,5 pto]** Suponga que se dispone de una variable global `int p` a la que durante la ejecución del proceso se le asigna el valor constante 7 (`p=7;`) y a continuación se invoca el servicio `fork` que está implementado con la optimización Copy-On-Write (COW). En el proceso hijo se realiza una modificación en el contenido de la variable p. Detalle cómo afecta esta modificación a la tabla de páginas del proceso hijo.

Al invocar el fork, se crea un clon de la imagen de memoria del proceso hijo. Con la optimización de Copy-On-Write (COW), realmente solo se duplica la tabla de páginas del padre, quedando el hijo apuntando a las mismas páginas y marcos que la del padre (la parte de la tabla que apunta a regiones compartidas, como el texto, no se duplica). Para las regiones privadas de cada página, tanto en la tabla del padre como en la del hijo, se marca COW y se elimina el permiso de escritura. Además, se incrementa el contador COW de dicha página. En el momento que el hijo intenta escribir en la variable p, la MMU genera una excepción de violación de memoria. El gestor de memoria comprueba que se trata de un intento de escritura en una página COW por lo que asigna un nuevo marco de página al proceso hijo, copiando el contenido de la página original. A continuación modifica la tabla de páginas del hijo para reflejar el nuevo marco de página y para sacar dicha página del COW y ponerle permisos de lectura y escritura. Además, decrementa el contador COW de la página compartida y, si llega a un solo proceso, modificará la tabla de páginas del padre para poner permisos de lectura y escritura en dicha página y para eliminarla de COW. Cuando el proceso vuelva a intentar escribir en p, la MMU marcará dicho marco de página como accedido y modificado.

2) **[1,5 pto]** Suponga ahora que en lugar de invocar el servicio `fork` se crean dos threads, Thread_1 y Thread_2. El Thread_1 modifica el valor de la variable global `int p` asignándole el valor constante 7 (`p=7;`). Detalle cómo afecta esta modificación a la tabla de páginas de ambos threads.

La tabla de páginas es una estructura de datos del sistema operativo que está asociada a los procesos, no a los threads. Todos los threads de un proceso comparten la imagen de memoria del proceso y, por tanto, la tabla de páginas que determina dicha imagen de memoria.
Al modificar el Thread_1 la variable p, queda afectada la página donde esté ubicada dicha variable, lo que puede incidir en la tabla de páginas de acuerdo a los siguientes supuestos.
Si la página está en un marco de página, el sistema operativo no realiza ninguna modificación en la tabla de páginas. Sin embargo, la MMU, al realizar la escritura de la variable p marcará dicho marco de página como accedido y modificado.
Si la página no está en un marco de página, se producirá un fallo de página, que el sistema operativo resolverá actualizando la tabla de páginas del proceso para reflejar el marco de página que ha asignado a dicha página. Por su lado, la MMU, al realizar la escritura de la variable p, marcará dicho marco de página como accedido y modificado.

Problema (conteste en una hoja aparte)

Sea un sistema con memoria virtual que presenta las siguientes características: páginas de 8 KiB; palabras de 32 bits; el sistema funciona con alineamiento de los datos.
Al ejecutar el mandato *size*, sobre un fichero ejecutable y sobre una biblioteca, la salida que se obtiene es:

 # size mi_programa

text	*data*	*bss*	*dec*	*hex*	*filename*
12096	*1080+ ¿?*	*8724+ ¿?*	*21900+ ¿?*	*558C + ¿?*	*mi_programa*

 # size mi_biblio

text	*data*	*bss*	*dec*	*hex*	*filename*
645024	*16384 + ¿?*	*13424 + ¿?*	*674832 + ¿?*	*A4C10 + ¿?*	*mi_biblio*

donde ¿? se corresponde con el valor deducido de la información disponible en el código que está distribuido entre los siguientes ficheros (suponga que también están incluidos todos los ficheros de cabecera necesarios para las declaraciones de las funciones del lenguaje y los servicios del sistema pero esta información no se considera para el cálculo de ¿?):

Fichero *cabecera.h:*

```
int a=3, b=5;    // Considere que el tipo int requiere 4 bytes
float *c;        // Considere que el tipo float requiere 4 bytes
struct tipo_estruct{
      short a;   // Considere que el tipo short requiere 2 bytes
      int b;
      char c;
      char d;
      double e;  // Considere que el tipo double requiere 8 bytes
      } var_estruct;
```

Fichero *load_dyn.c:*

```
#include "cabecera.h"
void *load_dyn(char * nombre) {
      void * hd;
      hd=dlopen (nombre, RTLD_NOW));
/* Punto A */
      return (hd);}
```

Fichero *fuente.c:*

```
#include "cabecera.h"
extern int load_dyn(char * nombre);
char literal[20]="0123456789=!?$%&/()";
int main (void) {
      int fd=0, j=5;
      void *hd;
      hd=load_dyn("mi_biblio");
      c=dlsym(hd,"mi_variable"); // En la biblioteca, int mi_variable=7;
      struct tipo_estruct *vest, *aux;
      vest=malloc(sizeof(struct tipo_estruct));
      vest->a=(short) a;
      vest->d='d';
      vest->e=(double) mi_variable;
      aux=vest;
/* Punto B */
      free(vest);
      fd = open ("datos.txt", O_RDWR);
      vest=mmap (0,sizeof(struct tipo_estruct), PROT_READ|PROT_WRITE,
      MAP_SHARED,fd,0); close(fd);
      aux=vest;
      aux->c=literal[10];
      aux->d=literal[11];
      munmap(aux,sizeof(struct tipo_estruct));
      printf("c: %c, d: %c, e: %f\n",vest->c, vest->d, vest->e);
      return 0;
/* Punto C */
}
```

El fichero datos.txt es un fichero ASCII que ocupa 40 B.

Se pide:

3) **[1 pto]** Indique el espacio ocupado por el tipo de datos `tipo_estruct` **justificando la respuesta.**

El campo b de tipo int no puede ajustarse al campo a de tipo short al existir alineamiento de los datos y trabajar con direcciones de 32 bits, por lo que queda un hueco de 2 bytes sin poder utilizarse entre estos dos campos. Lo mismo ocurre entre los campos d y e. En este caso, el hueco es de 1 byte. Por lo tanto, la estructura ocupa 20 bytes. Solamente si se activara la optimización del compilador, éste podría reordenar los distintos campos de la estructura para que el orden de almacenamiento de éstos permitiera acceder a cada uno de ellos en un solo acceso a memoria. Una posible reordenación sería: `struct tipo_estruct{ short a; char c; char d; int b; double e; }` y en este caso, la estructura pasaría a ocupar 16 B.

Apellidos, Nombre: _____ Nº Matrícula: _____ D.N.I.: _____

Sistemas Operativos – Gestión de Memoria – GII & GMI

19 de Mayo de 2014

Dispone de 50 minutos. Las notas saldrán el Miércoles 28 de Mayo. La revisión será el día Viernes 30 de Mayo a las 11.00 en la sala Multiusos del DATSI (D4200, Vestíbulo de la planta 2ª del Bloque IV)

---

4) **[3 pto]** Describa el mapa de memoria del proceso en los puntos A y B. Para cada región existente, indique el espacio ocupado en B y el número de páginas que abarca la región, sus permisos, si es una región privada o compartida, el soporte físico que tiene la región y el origen de su contenido. Suponga que a la región asociada a la pila se le asignan inicialmente 32 KiB y las variables de entorno ocupan 500 B.

Punto A:

- Región de código: 12096 B; 2 páginas; R-X; compartida; fichero ejecutable; fichero ejecutable.
- Región de datos con valor inicial: 1080 B + 8 B (2 int) +20 B (20 char); 1 página; RW-; privada; swap; fichero ejecutable.
- Región de datos sin valor inicial: 8724 B + 4 B (float *) + 20 B (tipo_estruct); 2 páginas; RW-; privada; swap; rellenar con ceros.
- Región de heap: 0 B; 0 páginas; privada; swap; rellenar con ceros.
- Región de pila: a la pila se le asignan inicialmente 32 KiB, que permiten almacenar sin problemas las variables de entorno, el bloque de activación de la función main con sus variables locales (20 B; 2 int + 3 punteros), el bloque de activación de la función load_dyn, la variable local de esta función (4 B de un puntero) y el bloque de activación de la función dlopen; 4 páginas; privada; swap; rellenar con ceros.

La invocación del montaje de la biblioteca se ha realizado con la opción RTLD_NOW, por lo que en ese momento se monta explícitamente la biblioteca.

- Región de código de la biblioteca: 645024 B; 79 páginas; R-X; compartida; fichero biblioteca; fichero biblioteca.
- Región de datos con valor inicial de la biblioteca: 16384 B + 4 B (1 int mi_variable); 3 páginas; RW-; privada; swap; fichero biblioteca.
- Región de datos sin valor inicial de la biblioteca: 13424 B + 0 B; 2 páginas; RW-; privada; swap; rellenar con ceros.

Si en la solución se ha considerado el modelo clásico de UNIX con una única región de datos que aglutina las regiones de datos con y sin valor inicial y el heap, se necesitarían 2 páginas (2 páginas > [1080 B + 8 B (2 int) + 8724 B + 4 B (float) + 20 B (20 char) + 20 B (tipo_estruct)]); privada; swap; los datos con valor inicial se encontrarían en el fichero ejecutable y el resto de datos proceden del mecanismo de rellenar con ceros los marcos de página. En este caso, la región de datos de la biblioteca se describiría: 16384 B + 4 B (1 int mi_variable) + 13424 B + 0 B; 4 páginas; RW-; privada; swap; rellenar con ceros.

Punto B:

Además de las regiones existentes en el punto A, en el punto B se produce la siguiente modificación en el heap:

- Región de heap: 20 B; 1 páginas; privada; swap; rellenar con ceros.

Si en la solución se ha considerado el modelo clásico de UNIX con una única región de datos que aglutina las regiones de datos con y sin valor inicial y el heap, se necesitarían las mismas 2 páginas (2 páginas > 1080 B + 8 B (2 int) + 8724 B + 4 B (float) + 20 B + 20 B (heap)); privada; swap; los datos con valor inicial se encontrarían en el fichero ejecutable y el resto de datos proceden del mecanismo de rellenar con ceros los marcos de página.

5) **[1 pto]** Explicar cómo le afecta al heap y a la región proyectada en memoria la ejecución de free(vest).

Las funciones malloc y free actúan solamente sobre el heap, mientras que el fichero proyectado en memoria es una región completamente independiente del heap que se gestiona con los servicios mmap y munmap. Al invocar la función free pasando como argumento la variable vest de tipo tipo_estruct, se liberará una zona de memoria en el heap correspondiente al tamaño de sizeof(struct tipo_estruct).

6) **[1 pto]** Indicar a qué región pertenece la dirección a la que apunta la variable vest en los puntos A, B y C.

La variable vest es una variable local de la función main y se almacenará en la pila, pero no está inicializada. Por lo tanto, en el instante A, al no estar inicializada, se puede considerar que la variable apuntará a posiciones basura porque el programador no tiene ningún control sobre el contenido inicial de la posición donde se ha almacenado vest.
En el B, apunta al heap tras la invocación de malloc.
Tras la proyección del fichero en memoria, recoge el valor de comienzo de la región creada. Al llegar al instante C, esa región ya no existe en la imagen de memoria del proceso tras la invocación del servicio munmap, por lo que apunta a una dirección no válida en el espacio de memoria del proceso.

7) **[1 pto]** Indique los valores impresos al invocar la función printf.

Al invocar la función printf se está accediendo a posiciones de memoria que ya no son válidas en el proceso porque se ha eliminado la región de memoria en la que estaban incluidas, por lo tanto se producirá un error de acceso a memoria y se abortará la ejecución del proceso.