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

Sistemas Operativos – GII & GMI

Tercer Parcial. Gestión de Memoria. 9 de mayo de 2016

Dispone de 50 minutos. Las notas saldrán el Martes 24 de mayo. La revisión será el Jueves 26 de mayo a las 15:15 en la sala Multiusos del DATSI (S4200, Vestíbulo de la planta 2ª del Bloque IV)

C1	C2	C3	P1a	P1b	P1c	P1d

Cuestiones (conteste las cuestiones en el espacio reservado al efecto en esta misma hoja y el problema en una hoja aparte)

1) [1 pto] Exlique en qué consiste la optimización *Copy-on-Write* (COW) y porqué supone un beneficio su utilización.

La optimización Copy-on-Write permite que, al hacer un fork, se compartan todas las páginas de las regiones de memoria, incluidas las que tienen permiso de escritura, y seguirán siendo compartidas mientras los accesos a dichas regiones sean únicamente de lectura. Al realizar el primer acceso de escritura se asignará una nueva página al proceso que realiza esta operación replicando la información contenida en la página original, y una vez hecha la copia se realizará la escritura. El mecanismo que se suele emplear es llevar un contador de los procesos que están compartiendo la región, incrementándolo cuando un nuevo proceso comparte la página y decrementándolo cuando se desdobla. Cuando el contador tenga valor "1" no será necesario realizar el desdoble y se desactiva el bit de protección contra escritura de la página que hasta ese momento estaba siendo compartida.

2) [1 pto] Suponiendo que se ha proyectado un fichero en memoria con permisos de escritura, indique cómo afectan las escrituras realizadas al contenido de dicho fichero para cada una de las opciones MAP_SHARED y MAP_PRIVATE.

Con MAP_SHARED las escrituras en la región proyectada se hacen efectivas en el disco, mientras que con MAP_PRIVATE no. En el caso de MAP_PRIVATE, la zona de intercambio de la región se ubica en el swap, mientras que en el caso de MAP_SHARED se hace swap sobre el propio fichero.

3) [1 pto] Indique los mecanismos típicos usados para el montaje de librerías dinámicas.

- Montaje explicito: el programador explicita en el código cuándo y cómo se realizará el montaje de una librería dinámica.

- Montaje al crear el mapa de memoria: al crear el mapa de memoria se montarán las librerías dinámicas que requiera el ejecutable.

- Montaje al invocar el procedimiento: cada librería dinámica se montará la primera vez que se invoque algún procedimiento o función de dicha biblioteca.

Problema 1 (Solución)

Un sistema operativo con memoria virtual sigue el modelo clásico UNIX con una región de datos que engloba los datos estáticos y los dinámicos (heap). Además, asigna por defecto un heap de 160KiB y una pila de 340KiB. El tamaño de página es de 16 KiB, el de los bloques en disco 4 KiB y el de los sectores 512 B. El montaje de la biblioteca es dinámico al invocar el procedimiento. Mediante el mandato size obtenemos los siguientes valores:

size mi_codigo

text	data	bss	dec	hex	filename
15872	512	4024	20408	4FB8	mi_codigo

size mi_biblioteca

text	data	bss	dec	hex	filename
115321	3080	5824	124225	1E541	mi_biblioteca

a) [1 pto] Deduzca el número de páginas necesarias para almacenar todas las regiones de la imagen de memoria del proceso en el momento de su creación así como el espacio en disco (en B y en bloques) que ocupa el fichero ejecutable. En el caso de que no se haya especificado el tamaño para alguna sección del fichero ejecutable, especifique un valor que considere plausible.

La región de código requiere 1 página. Como nos dicen en el enunciado, la región de datos engloba los datos con valor inicial (data), los datos sin valor inicial (bss) y el heap, por lo que se necesitan 11 páginas. La pila utiliza 22 páginas. Con el mecanismo de carga dinámica al invocar el procedimiento, la biblioteca se cargará solamente si se invoca algún objeto contenido en la biblioteca, por lo que al comienzo de la ejecución del proceso no existen más regiones en memoria. Por lo tanto, son necesarias 34 páginas.

El fichero ejecutable contiene la cabecera, p. ej. 1 KiB, la tabla de secciones, p. ej. 1 KiB, la sección de código, 15872 B, y la sección de datos con valor inicial, 512 B. También puede contener una tabla de símbolos, que suponemos de 3072 B. La sección de datos sin valor inicial (bss) no se almacena en el fichero ejecutable, y solo se recoge su descripción en la tabla de secciones. Lo mismo ocurre con el heap y la pila. Por lo tanto, el fichero ejecutable ocupa 21 KiB y son necesarios 6 bloques para su almacenamiento.

b) [2 ptos] mi_codigo incluye las siguientes líneas de código:

```
char *p;
p = mmap(NULL, 18192, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
n = fork();
p = p+18192;
for (i = 0; i < 5; i++,p++)
   *p = i*i;
munmap(p-18197, 18192);
```

Explique cuál será el contenido del fichero que ocupa 17000 B y que está asociado al descriptor fd.

No se indica en qué orden ejecutan los procesos padre e hijo, pero al ejecutar el mismo código ambos, el contenido de la región no depende de su orden de ejecución. La región debe ser múltiplo del tamaño de la página. Al reservarse 18192 B, son necesarias dos páginas para almacenar la información asociada al fichero. El puntero que almacena la posición donde se ubica la región se modifica posicionándose más allá del final del fichero, fuera del contenido original del fichero en disco, pero dentro de la región. Las posiciones p+18192 hasta p+18197 siguen siendo un direccionamiento válido para cualquiera de los dos procesos, por lo que ejecutarán el bucle con toda normalidad. El contenido del fichero original no se modifica, pues los accesos a la región son a posiciones que se ubican más allá del final del fichero.

c) [2 ptos] mi_codigo define una variable global int *p que ocupa 4 B e incluye las siguientes líneas de código:

Thread principal (TP)

```
1TP p = mmap(NULL, 18192, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
2TP *p = p;
3TP munmap(p, 18192);
....
```

Thread 1 (T1)
4T1 for (i = 0; i < 5; i++)
5T1 p[i] = i*i;

Explique el contenido de la región durante la ejecución de estas instrucciones en los dos casos siguientes: líneas 1TP,2TP,3TP,4T1,5T1 y líneas 1TP,4T1,5T1,2TP,3TP.

Al tratarse de threads del mismo proceso, comparten las variables globales, por lo que al crear la región en 1), el contenido de la variable p está accesible para los dos threads.

En el primer caso, en la instrucción 2TP) se asigna a la variable p la posición de memoria donde comienza la región. Al ejecutar munmap en la línea 3) se elimina la región sin salvar el contenido en el fichero al haberse creado con el flag MAP_PRIVATE, por lo que al acceder en la línea 5T1) a una posición de memoria referenciada dentro de la región se produce un error de ejecución por violación de memoria.

En el segundo caso, en el bucle se va escribiendo la secuencia 0, 1, 4, 9, 16 en los primeros 20 B de la región. Cuando se ejecuta la instrucción de la línea 2TP), se sustituye el 0 por el valor de la dirección de memoria donde se ubica la región. Finalmente, al ejecutar la línea 3TP), se elimina la región sin salvar su contenido en el fichero al haberse creado con el flag MAP_PRIVATE.

d) [2 ptos] mi_codigo realiza montaje al invocar el procedimiento de la biblioteca /lib/mi_biblioteca.so. mi_codigo accede a la biblioteca recuperando el valor de la variable double mi_var y utiliza la función int (*mi_func) (double *) también contenida en dicha biblioteca. Escriba un fragmento de código que utilice dicha función mi_func y el argumento de su invocación sea mi_var.

```
void *h;
int (*mi_funcion) (double *);
double *mi_variable;


h = dlopen("/lib/mi_biblioteca.so", RTLD_LAZY);
mi_funcion = dlsym(h,"mi_func");
mi_variable = dlsym(h,"mi_var");
printf("%d\n", (*mi_funcion)(mi_variable));
dlclose(h);
```