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

Sistemas Operativos – GII

Tercer Parcial. Gestión de Memoria. 18 de diciembre de 2017

Dispone de 50 minutos. Las notas saldrán el 12 de enero. La revisión será el 15 de enero a las 12:00 en la sala de cristal S4200.

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)

Responder brevemente pero de forma razonada a las siguientes preguntas.

1) [1 punto] Sea un programa del usuario root que hace uso de una biblioteca dinámica con montaje automático en tiempo de carga. Especifique qué regiones de memoria tendría al iniciar su ejecución así como los permisos, la fuente, si son privadas o compartidas y si son de tamaño fijo o variable.

Las regiones que tendría serían (1) código, (2) datos con valor inicial, (3) datos sin valor inicial, (4) heap, (5) código de biblioteca dinámica, (6) datos con valor inicial de biblioteca dinámica, (7) datos sin valor inicial biblioteca dinámica y (8) pila

Y sus características serían:

(1): compartida, tamaño fijo, R-X, fichero ejecutable
(2): privada, tamaño fijo, RW-, fichero ejecutable
(3) y (7): privada, tamaño fijo, RW-, rellenar a 0s
(4) y (8): privada, tamaño variable, RW-, rellenar a 0s
(5): compartida, R-X, fichero biblioteca
(6): privada RW-, fichero biblioteca

2) [1 punto] Explicar en qué consiste la técnica de optimización “Buffering de páginas”.

Consiste en mantener una reserva mínima de marcos libres, realizando las operaciones de reemplazo de forma anticipada. De esta forma, el tratamiento del fallo de página es más rápido puesto que basta con coger un marco de página libre. Cuando el sistema operativo detecta que el número de marcos de página disminuye por debajo de un cierto umbral, aplica repetidamente el algoritmo de reemplazo hasta que el número de marcos libres llegue a otro umbral que corresponda a una situación de estabilidad. Las páginas liberadas limpias pasan a una lista de marcos libres, mientras que las páginas sucias pasan a una lista de marcos modificados que deberán limpiarse (escribirse en memoria secundaria) antes de poder volver a utilizarse. Esta limpieza de marcos se puede intentar hacer cuando el sistema esté menos cargado y en lotes para obtener un mejor rendimiento del dispositivo de swap.

3) [1 punto] Un programador quiere hacer uso de la biblioteca plug-in.so solamente en el caso de que exista dicho fichero de biblioteca y en caso contrario escribir un mensaje de aviso por la salida estándar. Razonar qué tipo de montaje de biblioteca dinámica necesitaría usar y escribir el extracto del código necesario. Se podrá hacer uso de la función int existe_fichero (char* nombre_fichero); que devuelve 0 si no existe y 1 si existe el fichero pasado como argumento.

Debería usar montaje explícito para poder realizar o no la llamada a dlopen, con el nombre del fichero, dependiendo de si se cumple la condición.

```
void* handler;
if (existe_fichero("plug-in.so" ))
      handler = dlopen("plug-in.so", RTLD_LAZY);
else
      fprintf(stderr, "Aviso: no se ha encontrado la biblioteca plug-in.so\n");
```

Problema 1 (7 puntos)

Suponga un sistema de memoria virtual con páginas de 4 KiB que dispone de la optimización Copy-On-Write. El programa descrito a continuación tiene un fichero ejecutable con una cabecera de 1 KiB, una sección de código de 5 KiB y el espacio correspondiente a los datos descritos en el código. La pila y el heap ocupan inicialmente 52 KiB. Suponga además que se realiza montaje dinámico al invocar el procedimiento y que las regiones de código y datos de la biblioteca del lenguaje ocupan respectivamente 313 KiB y 85 KiB.

programa.c:

```
pid_t pid[10];
int array_aux[2000];

int main(void)
{
  int status;
  int a=5, i;
  printf("%d\n",a);
  for (i=0;i<10;i++){
    switch (fork()){
       case -1: break;
     case 0: break;
     default: break;
     }
  }
  for (i=0;i<10;i++){
    wait(&status);
  }
  return 0;
}
```

a) (1 punto) A partir del código descrito anteriormente, indique el número de páginas total que necesitarán las imágenes de memoria de todos los procesos una vez que se han creado todos. Explique la respuesta.

El proceso cuenta inicialmente con regiones de código (2 páginas), DVI+DSVI+Heap (0+2+13=15 páginas), pila (13 páginas) y monta dinámicamente la biblioteca libc que tiene una región de código (79 páginas) y una región de datos (22 páginas). Los imágenes de memoria de los procesos hijo son clones del padre. Los procesos hijos no realizan ninguna modificación en ninguna variable, así que con la optimización COW compartirán las páginas asignadas al padre hasta que algún proceso realice alguna escritura en las páginas compartidas. Si en el código del hijo se hubiera invocado exit(0), como hubiera sido deseable, solo existiría un nivel de recursión en la creación de los hijos, pero al no incorporarse, se crea una genealogía de procesos con 10 niveles de profundidad. Si nos fijamos solamente en la primera generación, solo el proceso padre va modificando la variable i en cada iteración del bucle y que sirve de índice del bucle, por lo que la página de la pila que contiene esa variable y que era compartida entre el proceso padre y el proceso hijo creado en cada iteración se desdobla. Por lo tanto, el número de páginas será la suma de las regiones enumeradas, 131 páginas, más las páginas de la pila desdobladas, 10. En total, 141 páginas.

b) (1 punto) Si en lugar de utilizar el servicio fork se utilizara la función de biblioteca pthread_create para crear threads, ¿cuántas regiones tendría el proceso? Explique el resultado.

Cada thread dispondría de su propia pila, más las regiones mencionadas anteriormente para el proceso padre: región de código, datos junto con heap, pila, y las dos regiones correspondientes a la biblioteca. En total, 15.

c) (2 puntos) Incorpore las modificaciones que crea pertinentes en el código anterior para que cada proceso en ejecución disponga de una imagen de memoria distinta durante su ejecución. Explique la respuesta.

Basta con incorporar la invocación del servicio exec en el código de los hijos en donde el argumento del programa a ejecutar sea distinto en la ejecución de cada hijo. Con el servicio exec se cambia la imagen de memoria del proceso en ejecución, por lo que la imagen del padre se modificará en los hijos con la mera invocación del exec. En el caso de los procesos hijos, basta con incorporar un argumento diferente en cada caso, p. ej. el índice del bucle o el pid del proceso. De esta forma, la pila de cada proceso hijo tendrá un contenido distinto.

```
      ...
 case 0:    char literal[100];
            sprintf(literal,"%d",i);
            execl("./programaA","programaA",literal,(char *)NULL);
            perror("Error en el exec"); exit(1);
            break;
```

d) (3 puntos) Incorpore las mínimas modificaciones necesarias para que todos los procesos puedan acceder a una región cuyo contenido sea el del fichero "./datos.txt".

Basta con realizar la proyección del fichero en memoria con el flag MAP_SHARED antes del bucle de creación de los procesos hijo. De esta forma, las imágenes de memoria de los procesos hijos ya tendrán el contenido del fichero con los permisos adecuados para realizar las modificaciones:

```
char * orig;
int fd;
struct stat fich;

fd=open("datos.txt",O_RDWR);
if (fd == -1){
      perror("Error en la apertura del fichero datos.txt");
      exit(1);
      }
if (fstat(fd,&fich)==-1){
      perror("Error al obtener el tamaño del fichero");
      exit(2);
      };
orig=mmap(NULL,fich.st_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
if (orig == MAP_FAILED){
      perror("Error en la creación de la región");
      exit(3);
      }
close(fd);
for (i=0;i<10;i++){
      switch(fork()){
      ...
```