**Informe del Segundo Proyecto**

**Sistemas Operativos**

Manuel Garat / Bárbara Mattioli

**Secuencias de Letras**

Una característica común a todas las implementaciones de secuencias es que cada proceso es una letra que, dependiendo de la cadena requerida, espera por la señal de ciertos procesos (letras) para luego escribir la letra que le corresponde y, finalmente, señalizar a uno o más procesos. Para modelar que una letra está precedida por distintas letras en distintos lugares de la cadena, se utilizan variables locales que indican cuándo esperar por una o por la otra. La misma idea es aplicada cuando distintas letras suceden a la misma.

Para la implementación con pipes, se declara uno por cada transición de letras distinta. Así, la espera y señal, se realiza con read y write en los mismos.

Para la implementación con colas, cada transición se simula con un tipo en particular de mensaje. De esta forma, si, por ejemplo, la letra A espera por E, va a leer un mensaje de tipo AE.

En la segunda secuencia, para las letras A, B y C, es necesario declarar las lecturas de pipes/colas como no bloqueantes, ya que si uno no puede, por ejemplo, recibir el mensaje de E, debe recibir el de la letra que sí lo recibió. Y, si tampoco pudo hacerlo, que vuelva a recibir a E.

**Problemas/Limitaciones:**

* Es necesario, para lograr un correcto funcionamiento, que los *printf()* incluyan un ‘\n’. Este detalle complica la

visualización del resultado, pero no pudimos encontrar una respuesta de por qué este caracter es necesario para que no se genere una cadena incorrecta y, por ello, tuvimos que dejarlo.

**Oso y Abejas**

Utilizando memoria compartida, la implementación no difiere mucho de una con hilos utilizando semáforos globales. La gran diferencia está en la configuración del espacio compartido entre los procesos y en la forma de cómo se acceden a estos semáforos: se utiliza un struct almacenando todos los semáforos necesarios y una variable que muestra la cantidad de espacios ocupados en el tarro. Cabe aclarar que esta variable **NO** es utilizada para la sincronización, solamente para una mejor visualización por pantalla de lo que está ocurriendo. Este struct se inserta en la memoria compartida, y los procesos oso y abeja, vinculados a este segmento, acceden a los semáforos a través de él. Cuando una abeja inserta su porción de miel, luego de obtener el tarro, verifica si fue la última. Si esto es así, despierta al oso, sino restaura la porción (ya que la verificación consiste en realizar un *wait(vacio)*) y libera el tarro para la próxima abeja. El oso restaura los *vacio*, y libera el tarro.

Con cola de mensajes, se utiliza un mensaje que contiene la cantidad de porciones libres del tarro. Esta variable es utilizada por cada abeja para verificar si fue la última en insertar una porción de miel. Se utilizan los tipos de mensaje para despertar al oso, apropiar/liberar al tarro.

**Problemas/Limitaciones:**

* Cada vez que se ejecuta la implementación con memoria compartida, se crea el segmento con clave 1234. Por lo tanto, si se ejecuta más de una vez, es muy probable que se genere un error al tratar de volver a crear el mismo (ya que este existe). Esto se soluciona con el comando *ipcrm -M 1234*, previo a la ejecución del programa.

**Archivos y Privilegios**

Este programa necesita cuatro parámetros: un nombre de archivo (o ruta), y tres cadenas de permisos/modos de la forma ABC, cada uno, donde A, B y C pueden ser:

* ‘r’ o ‘R’ para señalar un permiso de lectura;
* ‘w’ o ‘W’ para señalar un permiso de escritura;
* ‘x’ o ‘X’ para señalar un permiso de ejecución;

Estos permisos se pueden combinar de cualquier forma y no necesariamente debe haber tres (xrw y rw son ejemplos de permisos válidos). El programa permite, por ejemplo, el permiso “rrr”. Sin embargo, el resultado es igual al de aplicar ‘r’. Algo similar sucede si se ingresa una cadena de permisos con más de tres caracteres. Por ejemplo, aplicar “rwrwrwrw” da el mismo resultado que “rw”. Si ningún permiso se quiere especificar para usuarios/grupo/otros, se puede escribir ‘-’. Cabe destacar que especificar un permiso como “-r”, resulta en error de permiso no válido. ‘-’ se debe especificar solamente para no asignar permisos.

El archivo especificado es al que se le van a asociar los permisos especificados. La primera serie de permisos corresponde al usuario, la segunda al grupo y la tercera a otros. Las tres cadenas de permisos deben especificarse (). Para aclarar, se da la siguiente línea de comando, que muestra una correcta llamada al programa: **./ap archivo.txt xr wrw -**Esto significa asociar al **archivo.txt** el permiso de *lectura* y *ejecución* al **usuario**, de *escritura* y *lectura* al **grupo** y *ninguno* en particular a **otros**.

El programa consiste, básicamente, en decodificar estos permisos y “unificarlos” con el operando ‘|’, para luego llamar a la función *chmod()* y asignar esta disyunción de permisos al archivo especificado.

**Manejador de Memoria Virtual**

Se utiliza un arreglo para guardar cada dirección lógica del archivo de direcciones. Luego, a partir de este, se crean dos más: uno para el número de página y otro para el offset de la dirección. Para buscar el número de página en el TLB, primero se lee el bit de válido: si es 1 la página del TLB es válida y entonces se la compara con la página que proviene del archivo. Si no se encuentra la página, se usa su número para indexar la tabla de páginas. Si el bit de válido es 1 la página es valida. Si no se encuentra en la tabla de páginas, buscamos un frame libre en el cual se pueda almacenar la nueva página. Para ello, se mantiene una lista de marcos donde se indica cuáles están libres y cuáles ocupados. Al encontrar uno, se agrega el frame y la página a la tabla de páginas y al TLB (en este caso sólo el número de página). Para agregar una entrada al TLB se tiene en cuenta, además del bit de válido, el bit de referenciada que da la información de si la entrada fue referenciada luego de ser agregada al TLB. Si el bit de referenciada está en 0 (no se accedió a la misma), se reemplaza porque se considera que probablemente no vaya a ser utilizada en el futuro cercano. Si está en 1, se considera que probablemente también se la vuelva a utilizar; por lo tanto no se reemplaza. Desde ya, si el bit de válido está en 0 (es decir que la entrada no es válida), se puede reemplazar. Si todas son válidas y fueron referenciadas, entonces se podría reemplazar cualquiera. Sin embargo, se mantiene una variable que va rotando en el TLB para no reemplazar siempre la misma entrada. La tabla de páginas solo dispone de bit de válido y al reemplazar una entrada se lo setea a 1.

**Problemas/Limitaciones:**

* La solución planteada se aplica para solo un proceso. De haber más procesos, se debería tener en cuenta las páginas asignadas a cada proceso (cuya información está en los PCB) para poder modificar el bit de válido en el TLB y la tabla de páginas cada vez que un proceso comienza a ejecutar.
* Los marcos libres se marcan con un 0, mientras que los ocupados se marcan con un 1.
* El bit de válido en la tabla de páginas es el último bit de cada fila de la tabla, es decir el bit de la posición 8.
* El bit de referenciada del TLB se encuentra en la posición 17, mientras que el de válido en la 18. Se utiliza dos constantes, “ref” y “val” que almacenan los valores 16 y 17, respectivamente, para facilitar su visualización a la hora de indexar en el arreglo que representa al TLB.

**Contenido Incluido**

* *secuencia1\_pipe.c y s1p*: implementación y ejecutable de la secuencia i. con pipes;
* *secuencia2\_pipe.c y s2p*:implementación y ejecutable de la secuencia ii. con pipes;
* *secuencia3\_pipe.c y s3p*: implementación y ejecutable de la secuencia iii. con pipes;
* *secuencia2\_colas.c y s2c*:implementación y ejecutable de la secuencia ii. con colas;
* *secuencia3\_colas.c y s3c*: implementación y ejecutable de la secuencia iii. con colas;
* *oso\_abejas\_shm.c y oa\_shm*: implementación y ejecutable del problema del oso y las abejas con memoria compartida;
* *oso\_abejas\_colas.c y oa\_c*: implementación y ejecutable del problema del oso y las abejas con colas de mensajes;
* *archivo\_permisos.c y ap*: implementación y ejecutable del problema de archivos y permisos;
* *manejador\_memoria.c y mm*: implementación y ejecutable del manejador de memoria virtual;
* *constantes.h*: un header con todas las constantes, códigos de error y algunas funciones auxiliares que se utilizan a lo largo del proyecto.

**Sistema de Archivos**

Como para cada archivo se necesita un file descriptor, la cantidad de ambos, n, es la misma. Así, según la distribución de archivos y sus tamaños, se puede obtener la siguiente ecuación:

1)

Reemplazando el tamaño del bloque en 1):

2)

Aplicando factor común con 512B en 2):

3)

Resolviendo las sumas y productos entre paréntesis y dividiendo a ambos términos por 1B en 3):

4)

Resolviendo la suma y productos del término de la izquierda en 4):

5)

Dividiendo a ambos términos por 908.8 y luego resolviendo el término de la derecha en 5):

6)

De esta forma se obtiene, calculando el piso del resultado, que se puede almacenar un total de 5633 archivos.

Para calcular la cantidad de espacio requerido para los file descriptor de estos archivos, como hay uno por archivo, se tienen 5633 file descriptor. Como cada uno ocupa 64B se obtiene un total de o, aproximadamente, 352KB, es decir, 704 bloques de disco.