[1]

# Simulador de paginación y segmentación (Mayo 2018)

Luis Cordero, José Murillo y Roger Villalobos

***Abstract*—La sincronización de procesos es uno de los problemas que deben solucionar los sistemas operativos modernos. De esta manera se busca que sea un solo proceso que tiene el acceso a un recurso compartido a la vez para esto garantizar la atomicidad y evitar la incoherencia de datos.**

**el sistema propuesto para solucionar este problema usa un semáforo binario que se le asigna al proceso cuando este tiene turno de usar la memoria. La primera vez que se le asigna buscará si puede poner sus páginas o segmentos dentro de la memoria, luego lo libera. La segunda vez que lo asigna ya es para eliminar sus datos de la memoria y así dejar espacio disponible para que nuevos procesos puedan guardar sus datos.**

**El sistema tiene un funcionamiento correcto en cuanto a la sincronización de procesos y el acceso a la memoria. sin embargo a la hora de asignar, se utiliza una técnica de first-fit. La técnica de first-fit busca el primer espacio disponible de memoria lo suficientemente grande para albergar los datos solicitados por el proceso. Esta técnica resulta ser eficaz pero no necesariamente es la más eficiente. Se recomienda usar otras técnicas para el uso del sistema y así poder comparar el funcionamiento de este.**

[1]

## I. Introduction

Este documento busca presentar los resultados de una propuesta de sincronización de procesos realizado por el equipo de desarrolladores en el lenguaje de programación C. La solución hace el uso de un semáforo binario y la creación de hilos para simular el comportamiento de un sistema operativo hasta cierto punto. De acuerdo a las pruebas realizadas por el equipo, el programa sirve en su totalidad en cuanto a simular el manejo de memoria con sincronización de procesos a través de los esquemas de paginación y segmentación.

Los procesos que se van a simular se consideran como procesos cooperativos, los cuales pueden afectar o verse afectados por los otros procesos que estén ejecutándose en el sistema. El acceso concurrente de los procesos puede causar una incoherencia de datos. Se debe buscar una manera de sincronizar los procesos y así garantizar que solo un proceso a la vez pueda manipular los datos de una zona compartida.

Para poder incrementar la productividad, se debe permitir que muchos procesos puedan almacenar sus datos, pero solo en aquellos espacios que no se encuentran con datos. Para poder asignar espacio se pueden utilizar los esquemas de paginación o segmentación. El primero trabaja con una cantidad finita de páginas de tamaño fijo, el segundo trabaja con una cantidad variable de segmentos y espacios que toma en la memoria.

## II. mmap y shmget

La memoria compartida es uno de los recursos compartidos que ofrece los sistemas UNIX a los programadores para que los procesos puedan intercambiarse información. En el lenguaje de programación C en UNIX, es posible crear dos procesos distintos y que puedan compartir una región compartida de memoria y de esta manera compartir datos.

Cuando un proceso hace la operación mmap, este está haciendo que los contenidos de una parte de un archivo sean parte de su espacio de direcciones. El uso de esta operación permite tener un respaldo de datos ya que estos están siendo guardados en un archivo. El uso de un archivo permite la persistencia de datos entre reinicios del sistema.

Cuando un proceso hace la operación shmget, este está creando un segmento de memoria que va a ser compartido, o lo usa para localizar el segmento ya creado. Shmget se usa para establecer la clave de acceso al segmento, los permisos de acceso y el tamaño del segmento.

A diferencia de mmap, un segmento compartido creado usando shmget no puede crecer en tamaño.

[1]

## III. Tipo de semáforo utilizado para el sistema

El semáforo que que se decidió utilizar para el desarrollo del sistema es un semáforo binario. La memoria se puede ver como un recurso de una sola instancia. Tener un semáforo binario garantiza que solo un proceso pueda tener acceso al recurso y así poder insertar o eliminar datos de la memoria sin que haya otros procesos accediendo concurrentemente.

## IV. Sincronización en el sistema

El sistema tiene un hilo que se encarga de crear otros hilos, que se pueden ver como procesos. Estos procesos primero esperan el semáforo binario para tener acceso a la memoria. Una vez que el proceso adquiere el semáforo, los otros procesos creados deben esperar a conseguirlo, ya que es un semáforo binario. El proceso que tiene el semáforo por primera vez, tiene oportunidad de ingresar sus datos en memoria. En caso de que la cantidad de espacios disponibles no cumpla con la demanda del proceso, este muere, de lo contrario este guarda sus datos en memoria. Independiente de la acción, el proceso libera el semáforo.

Cuando el proceso libera el semáforo y ha guardado datos en memoria, este hace un sleep() para así darle oportunidad a otros procesos de poder asignar sus datos en memoria y así incrementar la productividad del sistema.

Cuando el proceso se levanta del sleep(), el proceso ahora compite con los otros creados por tener acceso a la memoria y así eliminar sus datos.

## V. Resultados

El sistema se compone de una función inicializador que se encarga de pedir la cantidad de espacios de memoria que se utilizaran en la simulación, donde cada índice es un espacio en memoria. El inicializador es capaz de validar los datos de entrada. Se valida que la cantidad de espacios de memoria sea potencia de dos. Es importante mencionar que el inicializador solo va a trabajar con números enteros, por lo que un dato de entrada como “2.0” no va a ser contado como entero. El inicializador en las pruebas muestra que efectivamente logra conseguir un espacio de memoria compartida.

Una vez que la función de inicializador termina, el hilo principal crea un nuevo hilo que se va a encargar de crear hilos o procesos con datos generados en tiempo de ejecución de manera aleatoria. Una vez que se crea el proceso, el hilo creador de procesos hace un sleep aleatorio entre treinta y sesenta segundos. Durante las pruebas, el creador muestra generar procesos con los valores que pertenecen al rango establecido para un proceso (Paginas: 1 a 10, tiempo: 20 a 60 segundos, cantidad de segmentos: 1-5, cantidad de espacios que toma un segmento: 1-3).

Con respecto a los procesos creados por el hilo creador. La sincronización si se pudo lograr gracias a la implementación del semáforo binario. Esto ayudó la atomicidad de las transacciones cuando se intentaba cargar y eliminar datos del proceso en la memoria compartida. La escritura en la bitácora también se incluyó dentro de las instrucciones que debe realizar el proceso solo cuando tiene acceso a la memoria compartida. Lo que no se puede apreciar de manera fácil es ver el proceso que tiene actualmente el semáforo si se intenta usar el comando “E”. Esto la adquisición y liberación del semáforo de hace de manera rápida. Para solventar eso se muestra un mensaje en pantalla pero si se desea, se puede agregar un sleep() de un tiempo adecuado para que se permita ver de mejor manera la cola de procesos en espera del semáforo.

El programa espía fue implementado como parte del hilo principal. Sirve para verificar el estado de la memoria y los procesos en un determinado tiempo. En cuanto al comando de mostrar el contenido de memoria, se muestra en todos los índices el identificador del proceso lo ocupa, y su segmento o página correspondiente, sino muestra que está libre. El contenido de todos los espacios de memoria en pantalla, un índice por línea. El número más alto con el que se probó el sistema fue con 8192, y esto hacía difícil poder ver todos los índices en una sola pantalla.

El programa finalizador se implementó como el cambio de una variable dentro del hilo principal. Esta variable es consultada por el hilo creador de procesos, cuando ambos hilos ven que se debe terminar ambos terminan su ejecución debidamente. Se realizaron pruebas con el comando “ps -a -t” antes y después de ejecutar el sistema. La cantidad de hilos que tenía corriendo terminó siendo el mismo, lo que garantiza que todos los hilos creados durante la simulación fueron terminados de manera apropiada. También se hizo una liberación del semáforo y memoria compartida y el archivo bitácora para así poder volverlos a solicitar en su siguiente ejecución.

El algoritmo que se utilizó para poder meter datos dentro de la memoria refleja el uso de first-fit, donde se busca el primer espacio lo suficientemente grande para guardar la página o el segmento. Esto puede disminuir la productividad del sistema al tener que hacer esperar a los otros procesos pero mejora el uso de CPU pues no tiene que recorrer todos los espacios de la memoria para así encontrar el espacio óptimo. Se puede intentar hace un algoritmo de compactación junto con otros métodos de asignación para futuras investigaciones y así comparar ese modelo con el propuesto en este sistema.

## VI. Casos de Prueba

### A. Inicializador

A continuación se muestra una tabla con los datos ingresados para el inicializador, el resultado esperado y el resultado obtenido de correr el programa.

| Entradas (espacios de memoria, espacios que toma una página) | Resultado esperado | Resultado actual |
|---|---|---|
| (16, 2) | El sistema acepta los datos y reserva la memoria solicitada | El sistema acepta los datos y reserva la memoria solicitada |
| (32, 2.0) | El sistema no acepta el dato de espacios que toma una página | El sistema no acepta el dato de espacios que toma una página |
| (8192, 64) | El sistema acepta los datos y reserva la memoria solicitada | El sistema acepta los datos y reserva la memoria solicitada |

### B. Creador de procesos

La siguiente tabla muestra algunos de los procesos que fueron creados durante las corridas tanto para trabajar con paginación como segmentación. Todos los procesos creados durante las pruebas fueron creados cumpliendo con los límites establecidos, ningún proceso fue perdido. Las pruebas fueron realizadas usando memoria de 16 espacios con cada página tomando 2 espacios por página.

| Propiedades del proceso | Cumple con los rangos establecidos |
|---|---|
| Id: 1, Páginas: 4, Tiempo: 56 | Sí |
| Id: 2, Páginas: 6, Tiempo: 27 | Sí |
| Id: 1, Segmentos: 2, Espacios por segmento: 2, Tiempo: 20 | |
| Id: 2, Segmentos: 4, Espacios por segmento: 2, Tiempo: 26 | Sí |
| | |

### C. Corridas del simulador

La siguiente prueba muestra el flujo de pasos que ocurrieron durante la simulación. Se utilizó un espacio de memoria de 16 donde cada página media 4 espacios

20:25:39 El proceso 1 fue creado
20:25:39 El proceso 1 adquiere el semáforo
20:25:39 El proceso 1 pone sus 4 páginas en memoria.
20:25:39 El proceso 1 libera el semáforo y entra en modo de ejecución (sleep)
20:26:12 El proceso 2 fue creado
20:26:12 El proceso 2 adquiere el semáforo
20:26:12 El proceso 2 no puede poner sus 6 páginas en memoria.

Estado de procesos:
→ Procesos en memoria: 1
→ Proceso buscando espacio: 0
→ Procesos que murieron: 2
→ Procesos que terminaron: 0

## VII. Experiencias

Para cada uno de los miembros del grupo de desarrollo, esta fue la primera experiencia de programación que involucra crear una solución para la sincronización de procesos. Hasta el momento en que se planteó el problema, nunca se les había presentado el desafío de buscar una manera de asegurar la consistencia de datos cuando hay más de un proceso tratando de modificar dichos datos. Se optó usar shmget para conseguir la memoria compartida ya que se quería trabajar con un segmento de memoria y no se veía necesario querer trabajar con archivos adicionales a la bitácora.

Se presentó un problema a la hora de construir la funcionalidad de la cola de espera. La estructura que se usaba para guardar el PID de los procesos que no pudieron guardar datos en la memoria presentaba inconsistencias contra lo que se mostraba en pantalla. Por medio de criterio ingenieril se logró solucionar este problema y así evitar resultados erróneos.

El desarrollo del sistema se realizó de manera estructurada puesto que los integrantes ya conocían la manera de trabajar de los otros. Unos se encargaban de diseñar las funciones necesarias dado los datos de entrada y salida especificados por el líder del equipo.

Al principio se intentó desarrollar el sistema separando la lógica de componentes en diferentes archivos, pero esto no resultó posible porque habían unos datos globales que no se actualizaban durante las pruebas. Se optó por manejar hilos que invocaban funciones para modificar los datos globales dentro de un solo archivo. No se tuvo suficiente tiempo para volver a intentar separar los componentes.

La separación del problema en subsistemas ayudó a que se agilizara el desarrollo y pruebas ya que solo se iban probando los componentes para asegurar su funcionamiento correcto. La integración de componentes no resultó difícil de realizar ya que en la etapas anteriores se había definido el componente como una serie de funciones para ser invocadas por el programa principal.

## VIII. Referencias

### A. Compilación del programa

El programa incluye un makefile, para recompilar el sistema solo se debe usar el comando “make”.

### B. Comandos a usar

- S: realiza la terminación del programa
- E: muestra donde se encuentran los procesos creados (cola de espera, en memoria, terminados, cola de muertos)
- M: muestra en pantalla el contenido de cada espacio en memoria

### C. Bibliografía


[1] Chapter 3. Sharing Memory Between Processes Prev Part II. Interprocess Communication. (n.d.). Accesado Mayo 15, 2018 de http://menehune.opt.wfu.edu/Kokua/More_SGI/007-2478-008/sgi_html/ch03.html

[2] Roxas, C. (2007, Febrero 4). Memoria compartida en C para Linux. Accesado Mayo 16, 2018, de http://www.chuidiang.org/clinux/ipcs/mem_comp.php

[3] Roxas, C. (2007, Febrero 4). Semáforos en C para Linux. Accesado Mayo 16, 2018, de http://www.chuidiang.org/clinux/ipcs/semaforo.php

[4] (2015, febrero 03). Accesado Mayo 16, 2018, de https://www.youtube.com/watch?v=TYnNKdf7cZM

[5] Sem_wait(3) - Linux manual page. (2018, Abril 30). Accesado Mayo 16, 2018, de http://man7.org/linux/man-pages/man3/sem_wait.3.html

[6] (n.d.). <semaphore.h> Accesado Mayo 16, 2018, de http://pubs.opengroup.org/onlinepubs/7908799/xsh/semaphore.h.html

[7] King, S. (N.d.). *Pthread Examples* [PowerPoint presentation]. Accesado de https://courses.engr.illinois.edu/cs241/fa2010/ppt/10-pthread-examples.pdf.

[8] (n. d.). Procesos e Hilos en C (2012, Febrero, 6). Accesado de http://www.um.es/earlyadopters/actividades/a3/PCD_Activity3_Session1.pdf

[9] Mohomed, I (2004). Systems Programming V (Shared Memory, Semaphores, Concurrency Issues) [PowerPoint presentation]. Accesado de http://www.cs.toronto.edu/~iq/csc209s/smalllectures/csc209_w10_4.pdf