TEC | Tecnológico de Costa Rica

Sede Central de Cartago

Escuela de Ingeniería en Computación

Curso de Principios de Sistemas Operativos

Profesora: Erika Marín Shumann

Proyecto II

Memoria compartida y sincronización de procesos

Estudiantes:

Brandon Dinarte Chavarría 2015088894

Julian Salinas Rojas 2015114132

Fecha de Entrega:

Miércoles 25 de octubre del 2017

Índice

Introducción

El presente documento detalla información relevante acerca del segundo proyecto del curso Principios de Sistema Operativos del segundo semestre del 2017. Este tiene como objetivo la simulación de cómo los procesos se sincronizan para compartir recursos, en este caso la memoria.

Se incluyen distintos programas que realizan tareas específicas. Uno de ellos se encarga de inicializar la memoria compartida. El segundo está a cargo de emular procesos que deberían ubicarse en memoria, ya sea usando el esquema de segmentación o el de paginación. El tercero es un programa que se encarga de devolver los recursos al sistema cuando ya se han terminado de usar. Por último, se tiene un programa espía capaz de mostrar el estado tanto de la memoria compartida como el de los procesos.

Estos programas fueron implementados usando el lenguaje de programación C sobre el sistema operativo Ubuntu 17.04.

Semáforos: ¿Qué tipo se utilizaron?

Cuando se usa memoria compartida los procesos mapean esa memoria a su propio espacio de direcciones. Esto permite una comunicación más rápida debido a que la información queda inmediatamente disponible para cualquier proceso. Esto implica que, cuando existen múltiples procesos, estos se deben de sincronizar para evitar estados inconsistentes en la memoria que comparten.

Para sincronizar estos procesos se hace uso de semáforos binarios ya que solo es necesario compartir un recurso, en este caso, la memoria. Estos representan un medio por el cual se puede regular el acceso a memoria, determinando si está disponible comprobando que la variable de control no está en cero. Antes de una

operación de escritura o lectura se debe reservar el semáforo, tan pronto se termina de usar, este debe ser desbloqueado.

Específicamente los semáforos utilizados son los implementados por GNU Linux que usan el estándar *POSIX*. Esto se justifica ya que usar un semáforo POSIX resulta más simple que usar uno de System V. Por contraparte, una ventaja de los semáforos System V es que están más ampliamente disponibles, particularmente en sistemas más antiguos basados en Unix, sin embargo, los semáforos POSIX han estado disponibles en sistemas Linux después de la versión 2.6 (aproximadamente desde el 2003), lo cual no representa una limitación actualmente.

Entre estos existen semáforos nombrados y no nombrados, los que se utilizan los semáforos nombrados pues estos pueden ser utilizados por el proceso que los crea y por cualquier otro aunque no tenga relación con el creador.

Para utilizarlos se debe incluir el encabezado *semaphore.h* dentro de archivos que necesiten acceder a la memoria compartida. Estos semáforos son representados por medio de un número entero positivo, de forma similar a como son representados los archivos por medio de un *file descriptor*. La forma en que los semáforos *POSIX* permiten la sincronización se explica en la siguiente sección, sin embargo, cabe destacar que existen dos operaciones que pueden ser realizadas para tal fin; incrementar el valor del semáforo en una unidad por medio de la función *sem_post* o disminuirlo usando *sem_wait*. Si el valor del semáforo es cero, la función *sem_wait* bloquea el proceso hasta que este valor se incremente a uno.

Sincronización: ¿Cómo se logró?

En este caso específico, varios procesos necesitan compartir un área de memoria. Entre estos se encuentran los procesos emulados por medio de hilos y el programa espía quien solo debe mostrar el estado de dichos procesos y de la memoria. Además, se comparten otros recursos como los semáforos y el archivo de bitácora.

Para lograr la sincronización es necesario solicitar de forma explícita al sistema operativo la cantidad de memoria que se requiere compartir, una vez solicitada la memoria, esta puede ser accedida desde diferentes procesos por medio de una clave única que la identifica. Para que todos los procesos tengan acceso a la misma clave, y por tanto a la memoria compartida, se escribe un archivo en disco que todos puedan acceder y que contenga dicha clave. El programa inicializador es el encargado de realizar este procedimiento, tomando como argumento el número de espacios de memoria que solicite el usuario para inicializar la memoria compartida. El programa productor necesita saber la cantidad de memoria que se puede usar, es decir, la cantidad de celdas de memoria, esto para que los procesos emulados puedan verificar donde es el límite de la memoria o celda final y así posicionarse o de lo contrario reportar el error en la bitácora. De igual manera, el programa espía necesita saber dónde termina para poder mostrar el estado completo de la memoria y de los procesos. Para lograr esto, el programa inicializador crea otro archivo que contiene la cantidad de espacios de memoria que había especificado el usuario. No es necesario asociar un semáforo con este archivo puesto que solo es de lectura después que el inicializador lo crea.

Un proceso puede estar ubicándose en la memoria compartida, mientras esto sucede los demás procesos deben esperar para poder buscar dónde ubicarse en memoria, ya que si no se hace, dos o más procesos podrían intentar tomar el mismo lugar. Para solucionar esto se coloca un semáforo antes de entrar en la región del código que cumple con dicha función. Luego de que el proceso es ubicado en la memoria, el siguiente proceso en la cola puede acceder a la región crítica. Cada uno de los semáforos (uno para cada recurso) debe tener un nombre diferente para identificar al recurso al que hacen referencia, por tanto, para asegurar que todos los procesos utilizan el mismo semáforo para acceder al mismo recurso, este debe de estar declarado en cada uno de los procesos con el mismo nombre, de forma análoga a como ocurre con la clave para acceder a la memoria compartida.

Además, cada vez que ocurre un evento, por ejemplo una asignación, una desasignación o un error, este debe ser registrado en una bitácora, esto podría ocasionar que varios procesos traten de escribir en la bitácora al mismo tiempo, generando inconsistencias, por tanto se ha implementado otro semáforo para evitar esta probabilidad. Adicionalmente, el programa espía también es capaz de mostrar la bitácora al usuario, es decir, es necesario para dicho programa solicitar el semáforo para asegurar que se lee una versión consistente del archivo.

MMAP vs SHMGET

Una de las diferencias es que la función *mmap* usa el estándar *POSIX*, mientras que *shmget* está implementada usando el modelo de memoria compartida *System V*. Estas funciones no se usan de forma conjunta, pues *mmap* requiere de un *file descriptor* (un identificador entero), similar a un archivo usando *POSIX*, mientras que *shmget* retorna un identificador del segmento de memoria compartida a partir de una clave.

Por una parte, la función *mmap,* similar a *shmmat* (*System V*) se encarga de mapear la memoria compartida al espacio de direcciones del proceso que la necesita, mientras que la función *shmget* solicita la referencia a un segmento de memoria compartida de un tamaño en especifico. Si esta memoria todavía no ha sido reservada, la función *shmget* debe ser utilizada con la bandera *IPC_CREAT*. Al contrario de su función homóloga *shm_open* (*POSIX*), el tamaño de la memoria debe ser especificada, mientras que con *shm_open*, la memoria debe ser truncada posteriormente usando la función *ftruncate* que recibe el *file descriptor* resultado de *shm_open* y el tamaño deseado.

De forma general, el flujo básico al usar *shmget* es el siguiente: declarar una clave mediante un archivo para que todos los procesos puedan acceder a la misma memoria compartida, obtener el identificador de la memoria compartida con *shmget* usando la clave creada anteriormente, obtener el puntero a la memoria compartida

con *shmmat* con base al identificador obtenido con *shmget*, hacer uso de la memoria, desvincular el puntero usando *shmdt* y por último, liberar la memoria compartida con *shmctl*.

El flujo para usar la función *mmap* es muy similar, primeramente se declara una variable que contenga un nombre único, este nombre debe ser conocido por todos los procesos que deben utilizar la memoria compartida, luego se usa usa *shm_open* para obtener un *file descriptor* asociado a la memoria compartida con base al contenido de la variable declarada previamente, después se trunca la memoria con *ftruncate*, es decir, se le da forma para que posteriormente se pueda mapear hacia el espacio de direcciones del proceso que la va a utilizar mediante la función *mmap*. Esta función necesita como argumentos el tamaño de la memoria, el file descriptor obtenido con anterioridad, así como banderas que permitan la lectura o escritura, según sea necesario. A partir de este punto se puede usar la memoria compartida como si fuese un archivo. Al terminar de usarla memoria se debe remover del espacio de direcciones del proceso mediante *munmap* y se debe cerrar el *file descriptor* mediante *close* (Nótese que se mantiene la similaridad al usar un archivo).

Análisis de resultados

En la versión final del proyecto, todas las funcionalidades se encuentran implementadas de forma correcta en relación a la especificación proporcionada.

Cada ejecutable es capaz de ejecutar las funciones que le corresponden. Por una parte el inicializador es capaz de reservar un espacio de memoria compartida, así como de crear el semáforo que se utiliza para accederlo. De forma inversa el programa finalizador puede liberar de forma correcta estos recursos.

El programa espía puede mostrar correctamente el estado de la memoria en cualquier momento. De igual forma sucede al mostrar el estado de cada uno de los procesos y la bitácora.

Algunas dificultades surgieron en el programa productor, sin embargo, se logró implementar sin mayores inconvenientes. El programa es capaz de crear varios procesos (hilos) que buscan la forma de sincronizarse para solicitar acceso a la memoria, luego que obtienen el acceso buscan un espacio donde puedan acomodar cada una de sus páginas o segmentos según se haya definido.

Casos de pruebas

Prueba 1: Programa inicializador
Descripción: Comprobar que el programa inicializador reserva la memoria compartida y crea el semáforo correspondiente.
Resultados esperados: Al ejecutar el inicializador, este debe haber reservado la memoria compartida, está debe de existir aún después de que el programa finaliza. Para verificar lo anterior, se ejecuta el comando *ipcs* esperando que exista el identificador mostrado por el programa inicializador. Para comprobar la existencia del semáforo se utiliza el comando *ls -al /dev/shm/sem.*|more*.
Resultado obtenidos: Al ejecutar el programa *Init*, este muestra el identificador de la memoria compartida. Al ejecutar el comando *ipcs* se muestra que el identificador existe, por tanto, la memoria ha sido exitosamente reservada. De igual manera con el semáforo, al ejecutar el comando *ls -al /dev/shm/sem.*|more* se muestran todos los semáforos creados, es decir, solo debe mostrarse un semáforo, lo cual es correcto.

Prueba 2: Programa finalizador
Descripción: Se pretende comprobar que el programa finalizador libera correctamente la memoria compartida reservada por el inicializador.
Resultados esperados: Después de que se ejecute el programa finalizador, al ejecutar el comando *ipcs* el identificador de la memoria compartida ya no debe de existir. Si nunca

se ejecutó el inicializador antes del finalizador, entonces el finalizador debería de mostrar un error de que no se encontró el segmento de memoria.
Resultado obtenidos: Después de que se ejecutó el finalizador el identificador de la memoria compartida ya no existe, de igual manera, al ejecutar el comando *ls -al /dev/shm/sem.*|more* se puede notar que el semáforo asociado tampoco existe.

Prueba 3.1: Programa productor
Descripción: Comprobar que los procesos están siendo creados de forma correcta, con características aleatorias para su ejecución.
Resultados esperados: Cada cierto período de tiempo, que se establece de forma aleatoria, se crea un hilo nuevo que simula un proceso. Dicho proceso tiene un tiempo de ejecución generado aleatoriamente.
Resultado obtenidos: Los procesos son creados de forma exitosa cada cierto período de tiempo generado de forma aleatoria, y comienzan la función que representa su ejecución como proceso con los parámetros especificados.

Prueba 3.2: Programa productor
Descripción: Verificar que el proceso de asignación esté siendo realizado exitosamente para cada uno de los hilos (que representan procesos). Esto se lleva a cabo mediante la revisión repetida del contenido de la memoria cada cierto intervalo de tiempo.
Resultados esperados: La memoria muestra espacios ocupados por las páginas o segmentos del proceso que ha obtenido pase del semáforo de forma más reciente. La memoria muestra espacios vacíos, luego de que un proceso retire su sección ocupada de dicha memoria. En caso de estar la memoria llena, un proceso con acceso a la región crítica, no logra modificar el contenido de la memoria.
Resultado obtenidos: La ejecución de un proceso, habiendo espacio en la memoria, altera los contenidos de la misma al agregarse sus páginas o segmentos. Los procesos retiran exitosamente sus páginas o segmentos, una vez que ya ha finalizado su ejecución. En las ocasiones que la memoria se encuentra llena, los procesos mueren sin poder ejecutarse por falta de memoria.

Prueba 3.3: Programa productor
Descripción: Verificar la correctitud del manejo de la información que requiere el programa espía, lo que incluye el número de proceso actual en memoria, la lista de procesos que se encuentran bloqueados actualmente, la lista de procesos que murieron por falta de memoria y los ejecutados correctamente.
Resultados esperados: El programa espía muestra el número de proceso que utiliza la región crítica actualmente, así como las listas de procesos bloqueados, terminados exitosamente y muertos por falta de memoria.
Resultado obtenidos: Por la velocidad con la ejecutan las operaciones los procesos, es difícil ver los procesos bloqueados y el proceso que utiliza la región crítica actualmente, pero si se realizan pausas en la ejecución, pueden verse dichos resultados de forma correcta. Los procesos terminados y muertos por falta de memoria pueden verse correctamente.

Prueba 4: Programa espía
Descripción: Permite ver el estado actual de todos los espacios de memoria definidos por el usuario. Y su valor correspondiente.
Resultados esperados: Se muestra correctamente el estado de los espacios de memoria con su respectiva información de proceso y partes de este. Se muestran los estados de procesos mencionados en la prueba 3.3.
Resultado obtenidos: Cada uno de los espacios de memoria muestra si contiene un proceso o si se encuentra vacío. En caso de contener un proceso muestra que parte o sección del proceso está conteniendo actualmente. El estado de procesos es mostrado correctamente como se menciona en la prueba 3.3.

Compilación

Para compilar el proyecto es necesario contar con make y cmake 3.7 como mínimo, estos están disponibles por defecto en Ubuntu 17.04, de lo contrario, si no se cuentan con estos programas, será necesario abrir una terminal con *Ctrl + Alt + T* y ejecutar los siguientes comandos para realizar su instalación:

```
> sudo apt-get install make && sudo apt-get install cmake
```

Una vez obtenidas estas dependencias, es necesario abrir una terminal ubicada dentro del proyecto. En esta colocamos el siguiente comando para crear el ejecutable en un carpeta llamada Bin:

```
> sudo mkdir Bin && cd Bin && cmake .. && make
```

El proyecto ya contiene la carpeta Bin, por tanto, para ejecutar el programa solamente es necesario accederla.

Ejecución

Un archivo readme.md con la información acerca de cómo ejecutar cada uno de los programas ha sido definido dentro de la carpeta donde se encuentra el código fuente. A continuación se describe lo mencionado en dicho archivo.

Existen 4 ejecutables, todos están presentes dentro del directorio *Bin* del proyecto, por tanto es necesario abrir una terminal ahí.

El primer ejecutable se llama *Init* y es el encargado de inicializar la memoria compartida, el archivo con la clave de dicha memoria y el semáforo a ser usado por los procesos que deben accederla. Para su ejecución se debe especificar una cantidad entera positiva que represente la cantidad de espacios de memoria:

```
> ./Init [cant_espacios_memoria]
```

El segundo ejecutable es *Prod*. Este crea varios procesos de forma aleatoria y con identificadores diferentes, lo que permite emular como los procesos son cargados en memoria según un algoritmo especificado que puede ser segmentación (seg) o paginación (pag):

```
> ./Prod [seg | pag]
```

El tercer ejecutable, nombrado *Spy* sirve como una interfaz de usuario que permite consultar el estado de la memoria y de todos los procesos. Se ejecuta de la siguiente manera:

```
> ./Spy
```

Al hacerlo nos mostrará los siguiente:

```
Opciones:
    A. Estado de la memoria
    B. Estado de los procesos
    C. Mostrar bitácora
    D. Algoritmo utilizado
    X. Salir del programa
Ingrese una opción:
```

Se debe ingresar la letra que contenga la opción que se desea ejecutar, puede ser en minúscula o en mayúscula. Si la opción ingresada no coincide con las opciones del menú, el programa no muestra ningún error, simplemente se omite hasta que la opción ingresada coincida.

Finalmente, el cuarto ejecutable es llamado *Fin* y es usado para liberar todos los recursos que fueron solicitados. Se ejecuta de la siguiente manera:

```
> ./Fin
```

Bitácora de trabajo

Fecha	Evento
7/10/2017	+ Se creó el repositorio en github. + Se lee el enunciado del proyecto. + Los integrantes definen las semanas de cuándo podrían empezar a trabajar.
10/10/2017	+ Creación del proyecto usando CLion, el cual permite trabajar de una forma más eficiente, al evitar que los programas tengan que ser compilados manualmente. + Se realizan pruebas de memoria compartida (POSIX). No parece ser tan difícil, es muy similar a manejar un archivo.
14/10/2017	+ Se logra escribir y leer números enteros en la memoria compartida. Pareciera que todo va por buen camino. + Las pruebas con POSIX son descartadas y se comienza a utilizar System V. No hay razón alguna, solo se hizo. + La pendiente no se torna a favor de nosotros, el objetivo del proyecto no es solo sincronizar procesos, aparentemente también a las personas quienes los programan. + Una versión preliminar tanto del programa inicializador como del finalizador es implementada. + Las pruebas para crear el programa productor son iniciadas, sin embargo, se posee poco entendimiento todavía. Las pruebas que se realizan son bastante rudimentarias.
16/10/2017	+ Se agrega una función para escribir en un archivo las cosas que los demás ejecutables necesitan tal como la clave para compartir la memoria. La clave no hubiese sido necesario de haberla guardado si se usará POSIX, pero el avance ya está hecho, no hay vuelta atrás. + Se agregan mejoras tanto al inicializador como al finalizador para que no tengan tantas variables ‘estáticas’ que fueron usadas para pruebas, aceptando ahora parámetros del usuario.

	+ Una carpeta de configuración es agregada para guardar todos los archivos que generen los programas y así lograr un mayor orden.
17/10/2017	+ El inicializador tenía muchos errores, y cosas que se podrían mejorar, por tanto, se invirtió bastante tiempo en esto, ya que si este programa no era correcto, era más difícil seguir con los otros.
18/10/2017	+ Se inicia con el programa productor ideando una forma en que la memoria pueda ser vista como un colección de celdas, creando funciones para escribir y leer en una celda en especifico. Se pensó que fueran celdas porque de está forma podrían servir, realizando pocos cambios, para paginación y segmentación.
19/10/2017	+ No hubo ningún avance, había que estudiar para el examen de sistemas operativos. + Haber estudiando para el examen mejoró notablemente el entendimiento hacía el proyecto.
20/10/2017	+ Los números aleatorios eran necesarios en el programa productor, por tanto, se implementa una función para tal fin. En realidad se estaba pensando en cómo se iba a implementar el productor, pues le falta mucho todavía, pero para no perder tiempo se hizo la funcionalidad. + Son creadas las estructuras que se usarán para paginación y segmentación aunque no se tienen bien definidas por el momento. + Se logra una versión del programa productor un poco más detallada, sin embargo, quedan muchas cosas por hacer. + Se está agotando el tiempo, y todavía se tienen muchos inconvenientes, sobretodo en cómo los procesos deben buscar el lugar donde acomodarse en memoria. + Se crea una función para facilitar la escritura de los mensajes que deben ir en la bitácora.
21/10/2017	+ Una versión preliminar del programa espía es implementada. Por el momento contiene las funciones de mostrar el estado de la memoria y la bitácora. Los mensajes en la bitácora son ficticios. + Se agregan funciones al programa productor para obtener la

	cantidad de celdas de memoria libres. + Se llega al acuerdo de que una persona continúe programando y que otra inicie la documentación. No es bueno dejarla para lo último que ya que si esta no sale bien se perderán puntos importantes y en cierta parte faciles. + La documentación está casi lista, solo es cuestión de terminar el proyecto para agregar las cosas que hacen falta. + La creación y borrado de los semáforos se mueven al inicializador y finalizador respectivamente. + El camino hacia la implementación definitiva del productor incierto. Seguiremos intentando “hasta que se aclaren los nublados del día”.
22/10/2017	+ Se han comenzado a realizar los casos de pruebas a pesar de que no se ha terminado la implementación, por lo menos estamos seguro que el inicializador y el finalizador sirven correctamente. + Estamos un poco atrasados, sin embargo, no se puede terminar el programa espía sin antes terminar el productor. + En un principio se documenta todo el código, pero actualmente se espera a que haya versiones estables de las funciones para realizar los comentarios pertinentes. + Se ha implementado el productor, solo resta optimizar el código. + Falta implementar por completo el programa espía. Se está pensando en la manera de cómo tomar el id del proceso que está acomodándose en la memoria así como los procesos en espera.
23/10/2017	+ Sin avances.
24/10/2017	+ Se resolvió el problema de almacenar los procesos que se encuentran bloqueados o en espera de acceso a la región crítica. Mediante la adición de una segunda sección de memoria compartida que almacena dichos procesos y que tiene su propio semáforo. + Se realizaron gran cantidad de ajustes de código, en términos de legibilidad y optimización.
25/10/2017	+ Se hicieron varias pruebas. + Se terminó la documentación y se envió el proyecto.

Referencias

[1] Bill Gallmeister (1995), POSIX.4 Programmers Guide: Programming for the Real World, O'Reilly Media. Recuperado de: http://elk.informatik.hs-augsburg.de/tmp/elinux/Buecher/POSIX4.pdf

[2] Universität Hamburg, Shared Memory, secciones 1.1, 1.2, 1.3. Recuperado de: https://www3.physnet.uni-hamburg.de/physnet/Tru64-Unix/HTML/APS33DTE/DOCU_004.HTM

[3] Universität Hamburg, Semaphores, secciones 9.1, 9.2, 9.3. Recuperado de: https://www3.physnet.uni-hamburg.de/physnet/Tru64-Unix/HTML/APS33DTE/DOCU_010.HTM

[4] Systutorials, Semaphores Overview, Recuperado de: https://www.systutorials.com/docs/linux/man/7-sem_overview/

[5] Softprayog, POSIX Semaphores, Recuperado de: https://www.softprayog.in/programming/posix-semaphores

[6] Ingeniería de Sistemas y Automática UMH, SITR: Funciones POSIX III Semáforos, Recuperado de: http://isa.umh.es/asignaturas/sitr/TraspSITR_POSIX3_Semaforos.pdf