Tecnológico De Costa Rica

Escuela De Computación

Principios de Sistemas Operativos

Realizado Por:

José Miguel Mora Rodríguez

Dylan Rodríguez Barboza

Karina Zeledón Pinell

Profesor:

Ing. Erika Marín Schumann

Proyecto 2: Simulación Paginación / Segmentación

Octubre, 2017

Tabla de Contenido

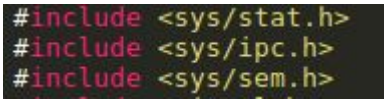
Tipo de semáforo utilizado

Para la implementación de esta tarea programada, se optó por la utilización de semáforos binarios. Estos son conocidos por tomar sólo dos valores: 0 y 1, son inicializados en 1 y son usados cuando solo un proceso puede acceder a un recurso a la vez. Esto se relaciona con el contexto del enunciado de la tarea programada, en la cual se requiere que solo un proceso tuviese la capacidad de pedir memoria a la vez, seguidamente, se debía escribir en bitácora lo que estuviese sucediendo en un momento dado de la ejecución, ya fuese asignación o desasignación de recursos para un proceso.

¿Por qué se eligió este tipo de semáforo?

Se seleccionaron los semáforos binarios, debido a que estos se adaptan en gran medida a los requerimientos del proyecto programado. Facilita el acceso de un proceso a la vez a los recursos (asignación de memoria), en la que una vez uno estuviese dentro, los demás procesos quedaban en espera de pedir sus recursos.
Finalmente, estos semáforos son fáciles de implementar en código en el lenguaje de programación C.

Implementación de los semáforos



En este imagen se pueden apreciar las inclusiones necesarias para la implementación de los semáforos binarios.

```c
/*Función que libera el semáforo*/
void doSignal(int semid, int numSem) {
    struct sembuf sops; //Signal
    sops.sem_num = numSem;
    sops.sem_op = 1;
    sops.sem_flg = 0;

    if (semop(semid, &sops, 1) == -1) {
        perror(NULL);
        error("Error al hacer Signal");
    }
}

/*Función que utiliza el semáforo*/
void doWait(int semid, int numSem) {
    struct sembuf sops;
    sops.sem_num = numSem; /* Sobre el primero, ... */
    sops.sem_op = -1; /* ... un wait (resto 1) */
    sops.sem_flg = 0;

    if (semop(semid, &sops, 1) == -1) {
        perror(NULL);
        error("Error al hacer el Wait");
    }
}

/*Función que inicializa el semáforo*/
void initSem(int semid, int numSem, int valor) { //iniciar un semaforo

    if (semctl(semid, numSem, SETVAL, valor) < 0) {
    perror(NULL);
        error("Error iniciando semaforo");
    }
}
```

En este imagen se aprecia como se implementan las funciones correspondientes: **doSignal** es el encargado de liberar el semáforo, **doWait** lo pide y deja a los demás procesos en espera, finalmente, **initSem** inicializa el semáforo, este estará listo para ser solicitado después.

¿Cómo se logró la sincronización?

La sincronización fue lograda gracias a la implementación de los semáforos binarios, los cuáles: se implementaron en las funciones que manejan los hilos (procesos), en el código del equipo de trabajo existe una función encargada de pedir la cantidad de páginas que requiere cada proceso, antes de hacer el llamado a la función, se llamaba a la función **doWait**, para que solo el primer proceso en el momento pudiese acceder a la memoria, luego de pedir el acceso, en caso de que hubiesen suficientes páginas, se llamaba a la función **doSignal**, para proceder al sleep del hilo, con esto ya se le permite a otro proceso acceder a la memoria para hacer sus peticiones.

Además de esto, debía liberarse el proceso una vez que finalizara el sleep, por lo que, una vez finalizado, se volvía a llamar a **doWait**, (dejando a los demás procesos en espera), se devuelven los recursos (en este caso, páginas), se escribía en bitácora y, finalmente, se llamaba a **doSignal**, para que los demás procesos también pudiesen liberar sus recursos.

Implementación

```
doWait(semaforo,0); /*Solicita el semáforo, si está siendo utilizado, el pr

/*-------------------------------------------------------------------------

sprintf(mensaje +strlen(mensaje), "----- Termina el sleep del proceso: ");
sprintf(mensaje +strlen(mensaje), "%d", idProceso);
sprintf(mensaje +strlen(mensaje), "-----\n");
sprintf(mensaje +strlen(mensaje), "-----------------------------------------

/*-------------------------------------------------------------------------

escribirEnBitacora(mensaje);
mensaje = (char*)malloc(sizeof(char*));


liberarMemoriaSegmentacion(idProceso);

//liberarProcesoEnMemoria(idProceso); /*Espía*/

//ESCRIBIR EN BITÁCORA

escribirProcesoTerminado(idProceso, "Segmentacion", idThread); /*Espía*/


/*-------------------------------------------------------------------------

sprintf(mensaje +strlen(mensaje), "--- Terminó la ejecución del proceso: ")
sprintf(mensaje +strlen(mensaje), "%d", idProceso);
sprintf(mensaje +strlen(mensaje), "---\n");
sprintf(mensaje +strlen(mensaje), "-----------------------------------------

escribirEnBitacora(mensaje);

/*-------------------------------------------------------------------------


mensaje = (char*)malloc(sizeof(char*));


doSignal(semaforo,0); /*Libera el semáforo*/
```

Como se explicó anteriormente, **doWait** hace que los demás procesos queden en espera para que el proceso actual pueda hacer sus operaciones, en este caso, liberar la memoria y escribir en la bitácora. Al final se puede apreciar la llamada al **doSignal**, para darle la oportunidad a los demás procesos de hacer las mismas operaciones.

Diferencia entre las funciones shmget y mmap

Memoria mapeada

La memoria mapeada permite que los procesos se comuniquen a través de un archivo compartido. Esto se diferencia de los segmentos compartidos de las siguientes formas: la memoria mapeada puede ser usada para IPC (comunicación entre procesos) o para acceder los contenidos de un archivo, además, la memoria mapeada forma una asociación entre la memoria del proceso del archivo.
Linux divide el archivo en fragmentos de tamaño de página y luego los copia en páginas de memoria virtuales, esto para hacerlos disponibles en un espacio de direccionamiento en un proceso, esto permite un acceso rápido a los archivos.

Implementación

El system call **mmap** se puede usar para asignar un archivo a espacios en direcciones de la memoria virtual del proceso. De cierta manera, **mmap** es más flexible que **shmget**.
Una vez que se establece el mapeo, se pueden usar llamadas al sistema estándar en lugar de llamadas especializadas para manipular la memoria compartida.
A diferencia de la memoria, el contenido de un archivo es no volátil y permanecerá disponible incluso cuando el sistema sea apagado o reiniciado.

Parámetros de este system call

Start: es la dirección para acoplarse.
Lenght: el número de bytes que serán acoplados.
Prot: para indicar el tipo de acceso (protección) para el segmento.
Fd: es un descriptor de archivo abierto válido.
Offset: para indicar la posición inicial para el mapeo.

Si el system call **mmap** sale bien, este retornará una referencia al objeto de memoria mapeado.
Si falla, retorna la constante MAP_FAILED (la cual es el valor -1).

Memoria compartida

La memoria compartida le permite a múltiples procesos compartir espacio de memoria virtual, esta es la forma más rápida para la comunicación entre procesos. En general, un proceso crea o asigna el segmento de memoria compartida, dicho esto, el tamaño y los permisos de acceso al segmento deben ser definidos cuando son creados. Un proceso, seguidamente, conecta el segmento compartido, de manera que se puedan mapear sus datos actuales.
Una vez creado, y si los permisos que se le asignaron lo permiten, otros procesos pueden tener acceso al segmento de memoria compartida y mapear en sus datos. Cada proceso accede a la memoria compartida, de forma relativa a su dirección.

Creación de un segmento de memoria compartida

Para este fin, se utiliza el system call llamado **shmget**, el cuál permite la creación de un segmento de memoria compartida y la generación de una estructura de datos del sistema asociada. El segmento de memoria compartida y la estructura de datos del sistema poseen un identificador único, este es retornado de **shmget**.

Parámetros de este system call

El primer parámetro es una llave que especifica cuál segmento crear.
El segundo parámetro especifica el número de bytes en el segmento.
El tercer parámetro es una bandera, la cual puede tomar diferentes valores, para efectos del proyecto programado, se utilizó la bandera IPC_CREAT, el cuál se encarga de solamente crear el nuevo segmento.

Análisis de resultados del programa

Funcionalidad	Estado
Pedido de un segmento en memoria compartida mediante un programa inicializador.	Completa. Se hizo un archivo llamado “Configuración”, en el cual se tienen los tamaños que se pretenden apartar de la memoria compartida.
Conexión a la memoria compartida.	Completa
Programa productor de procesos con un esquema de paginación.	Completa
Programa productor de procesos con un esquema de paginación.	Completa
Uso de semáforos.	Completa. Se utilizaron semáforos binarios.
Sincronización de procesos.	Completa. Se utilizaron los semáforos para este fin.
Interfaz que indique qué sucede en cada momento durante la ejecución del programa.	Completa
Escritura en un bitácora.	Completa
Sincronización al escribir en la bitácora.	Completa

Generación aleatoria de páginas, segmentos y tiempos de ejecución para los procesos.	Completa
Acceso en todo momento al estado de la memoria (espía).	Completa
Acceso en todo momento a los procesos que están en memoria (espía).	Completa
Acceso en todo momento al proceso actual que esté pidiendo memoria (espía). Estado: .	Completa. Para hacer efectiva esta prueba. Se hizo un sleep a un proceso al pedir el semáforo, ya que, como el proceso de pedido de memoria es muy rápido, es difícil tener este dato sin esa intervención.
Acceso en todo momento a los procesos que estén bloqueados (espía). Estado:	Completa. Se hizo un sleep a un proceso al pedir el semáforo, ya que, como el proceso de pedido de memoria es muy rápido, es difícil tener este dato sin esa intervención.
Acceso en todo momento a los procesos que estén muertos (espía).	Completa
Acceso en todo momento a los procesos que hayan terminado su ejecución (espía).	Completa
Matar un proceso mediante un programa finalizador.	Completa
Liberar los segmentos en memoria compartida mediante un programa finalizador.	Completa
Manejo de errores. Validación que le indique al usuario que no se puede acceder a la memoria compartida si no está apartada, tanto en el productor de procesos, espía y finalizador.	Completa

Casos de pruebas

Para la implementación de esta tarea programada, se hicieron las siguientes pruebas:

Inicializador

- Ejecución del programa, el cuál incluye el archivo de configuraciones con valores debidamente configurados. Se esperaba un mensaje de confirmación que indicara que se han creado los segmentos de memoria compartidos. Se obtuvo dicho mensaje de confirmación al ejecutarlo.
- Ejecución del programa, pero esta vez, con memoria compartida ya apartada, dicha ejecución se hizo con tamaños de segmentos de memoria compartida diferentes a los de la prueba pasada. Se esperaba un mensaje que indique que no se puede asignar la memoria compartida, debido a que ya había una apartada con un tamaño diferente. Se obtuvo dicho mensaje de confirmación al ejecutarlo.
- Ejecución del programa, en el que se inicializaran valores para los segmentos de memoria compartida, para ser accedidos y verificar si se inicializaron correctamente. Se esperaba la impresión de los datos digitados. Se obtuvieron dichos datos al ejecutarlo.

Productor de procesos

- Ejecución del programa sin haber ejecutado el inicializador primero. Se esperaba un mensaje que indicara que no puede accederse a la memoria compartida. Se obtuvo dicho mensaje al ejecutarlo.
- Ejecución del programa en la modalidad de paginación y segmentación. Se esperaba que se hicieran efectivos los requerimientos que pedía el enunciado, tales como asignación, desasignación, sincronización, matar y terminar procesos. Se obtuvieron dichos resultados al ejecutarlo.
- Ejecución del programa en la modalidad paginación y segmentación. Se esperaban impresiones de los datos de los procesos que se estuviesen produciendo, su estado, el estado de la memoria antes de intentar asignar páginas al proceso. Se obtuvieron dichos datos al ejecutarlo.
- Ejecución del programa en la modalidad paginación y segmentación. Se esperaban impresiones que identificaran al proceso cuando no hubiesen suficientes páginas para asignarle a un proceso, además de indicar que se iba a matar al proceso. Se obtuvieron dichos datos al ejecutarlo.
- Ejecución del programa en la modalidad paginación y segmentación. Se esperaban impresiones que indicaran cuando un proceso empezara y terminara su ejecución. Se obtuvieron dichos datos al ejecutarlo.

• Ejecución del programa en la modalidad paginación y segmentación. Se esperaba que se estuviese escribiendo en memoria compartida, los datos que se querían implementar para la bitácora. Se obtuvieron dichos resultados al ejecutarlo, esto haciendo una impresión del contenido del segmento de memoria compartida de la bitácora.
• Ejecución del programa en la modalidad segmentación. Se esperaban impresiones que identificaran a un segmento cuando se pudiese o no asignar a páginas contiguas, y matar al proceso en caso de que no hubiesen suficientes. Se obtuvieron dichos datos al ejecutarlo.
• Ejecución del programa en la modalidad segmentación, en la que se asignaron ciertos espacios ocupados y desocupados a la memoria compartida desde la incialización. Se esperaba que, con el suficiente número de páginas para el proceso y suficientes páginas contiguas para cada segmento, se pudiese asignar cada uno a las páginas contiguas que más se adaptaran a las páginas requeridas, de manera que no afectara la asignación de los demás segmentos. Se obtuvieron dichos resultados al ejecutarlo.
• Ejecución del programa en la modalidad segmentación, en la que se asignaron ciertos espacios ocupados y desocupados a la memoria compartida desde la inicialización. Se esperaba que, con el suficiente número de páginas para el proceso, pero no el número suficiente de páginas contiguas para cada segmento, se imprimieran mensajes indicando que no se puede asignar un segmento y, posteriormente, se matara al proceso. Se obtuvieron dichos resultados al ejecutarlo.

Espía

• Ejecución del programa espía sin haber ejecutado el incializador primero. Se esperaba que se imprimiera un mensaje que indicara que no se podía acceder a la memoria compartida. Se obtuvieron dichos resultados al ejecutarlo.
• Ejecución del programa para pedir el estado actual de la memoria. Se esperaba que se imprimieran las páginas, su estado (ocupada o desocupada) y los procesos que las estuviesen ocupando, en caso de que hubiesen. Se obtuvieron dichos resultados al ejecutarlo.
• Ejecución del programa para pedir los procesos que estuviesen en la memoria. Se esperaba que se imprimieran los procesos en memoria (pId y mecanismo), en caso de que hubiesen, y un mensaje que indicara que no habían procesos en memoria en caso de que no. Se obtuvieron dichos resultados al ejecutarlo.
• Ejecución del programa para pedir al proceso que estuviese pidiendo memoria en el momento. Se esperaba que se imprimieran el proceso pidiendo memoria (pId y mecanismo), en caso de que hubiese uno, y un mensaje que indicara que no había ningún proceso pidiendo memoria en caso de que no. Se obtuvieron dichos resultados.

- Ejecución del programa para pedir los procesos que estuviesen bloqueados. Se esperaba que se imprimieran los procesos bloqueados (pId y mecanismo), en caso de que hubiesen, y un mensaje que indicara que no habían procesos bloqueados en caso de que no. Se obtuvieron dichos resultados al ejecutarlo.
- Ejecución del programa para pedir los procesos que estuviesen muertos. Se esperaba que se imprimieran los procesos muertos (pId y mecanismo), en caso de que hubiesen, y un mensaje que indicara que no habían procesos muertos en caso de que no. Se obtuvieron dichos resultados al ejecutarlo.
- Ejecución del programa para pedir los procesos que hayan terminado su ejecución. Se esperaba que se imprimieran los procesos que terminaron su ejecución (pId y mecanismo), en caso de que hubiesen, y un mensaje que indicara que no habían procesos que hayan terminado su ejecución en caso de que no. Se obtuvieron dichos resultados al ejecutarlo.
- Ejecución del programa para pedir todos los datos mencionados en los puntos anteriores. Se esperaban los resultados dados en los puntos anteriores, con la diferencia de que estos van a ir juntos en una misma salida de datos. Se obtuvieron dichos resultados al ejecutarlo.

Finalizador

- Ejecución del programa finalizador sin haber ejecutado el incializador primero. Se esperaba que se imprimiera un mensaje que indicara que no se podía acceder a la memoria compartida. Se obtuvieron dichos resultados al ejecutarlo.
- Ejecución del programa con la ejecución previa del inicializador. Se esperaba que se liberaran los segmentos de memoria compartida y se indicara por medio de impresiones. Se obtuvieron dichos resultados al ejecutarlo.
- Ejecución del programa. Se esperaba que se generara un txt con el contenido de la bitácora después de la ejecución de este programa. Se obtuvieron dichos resultados al ejecutarlo.

Bitácora de trabajo

9 de octubre del 2017:

- **9:15pm**: se realizó un análisis del enunciado del proyecto, se apuntó en una hoja las funcionalidades que se deben implementar.

11 de octubre del 2017:

- **11:00pm**: investigación previa sobre cómo apartar memoria compartida por medio del comando shmget.
- **12:00am**: programación del inicializador, se programaron dos funciones para lograrlo.

12 de octubre del 2017:

- **9:00am**: acceso a la memoria compartida desde otro programa.
- **10:00am**: inicio de programación del productor de procesos.

16 de octubre del 2017:

- **11:00pm**: se hizo un menú principal para manejar el productor de procesos.
- **12:00pm**: inicio de programación del mecanismo de paginación del productor de procesos, se incluyeron subrutinas como solicitud de memoria, tomo de memoria y liberación de la misma.

17 de octubre del 2017:

- **11:00pm**: implementación casi total del mecanismo de paginación, con solo una funcionalidad restante.

20 de octubre del 2017:

- **11:00pm**: implementación total del mecanismo de paginación.

21 de octubre del 2017:

- **3:00pm**: implementación de un espacio de memoria compartida para la bitácora y otro para el programa espía. Se mejoraron los print de los datos actuales.
- **10:00pm**: se aparta memoria compartida para las diferentes secciones que pide el espía.
- **11:00pm**: se comienza a programar al espía, 1:00am: programación total del espía.
- **2:00am**: comienzo de programación del programa finalizador.

22 de octubre del 2017:

- **11:00pm**: seguimiento de programación del finalizador.

23 de octubre del 2017:

- **1:00pm**: comienzos de implementación de escritura en la bitácora del programa

24 de octubre del 2017:

- **1:00pm**: implementación completa de la parte de la bitácora.
- **6:00pm**: implementación completa del finalizador.

25 de octubre del 2017:

- **12:00pm**: cambio en los menús principales de todos los programas.

Compilación y ejecución de cada programa

Note que en la compilación de los cuatro programas se utiliza el parámetro **-w**, el cual permite la compilación sin warnings.
Note que en la compilación del productor de procesos se utiliza el parámetro **-pthread**, el cual permite la compilación de esta librería.

Inicializador

Compilación: gcc Inicializador.c -o runInicializador -w
Ejecución: ./Inicializador

Productor de procesos

Compilación: gcc ProductorProcesos.c -o runProductorProcesos -pthread -w
Ejecución: ./runProductorProcesos

Espía

Compilación: gcc Espia.c -o runEspia -w
Ejecución: ./runEspia

Finalizador

Compilación: gcc Finalizador.c -o runFinalizador -w
Ejecución: ./runFinalizador

Conclusiones

- En la implementación de los hilos, se devuelve un id cada vez que se crea uno, los id pueden reutilizarse una vez que muere un hilo.

- La implementación del mecanismo de paginación puede entenderse como más simple que el de segmentación, debido a que este no necesita que las páginas disponibles sean contiguas, también puede verse como una reutilización de algunas subrutinas que pueden implementarse en el mecanismo de segmentación.

- Es importante el uso de semáforos para este tipo de implementaciones de sistemas, por ejemplo, para el pedido de memoria, de esta forma, dos procesos no solicitarán un mismo espacio, evitando futuros conflictos entre los mismos.

- La implementación de semáforos binarios ayudan a la implementación de una sincronización de procesos, debido a que solo uno podrá acceder a la región crítica por vez.

Bibliografía

1. Marshall, D. (5 de 1 de 1999). Obtenido de Cardiff School of Computer Science & Informatics: https://users.cs.cf.ac.uk/Dave.Marshall/C/node27.html