Tarea 3

Memoria Virtual (Paginación bajo demanda)

Publicada el 25 de septiembre de 2017

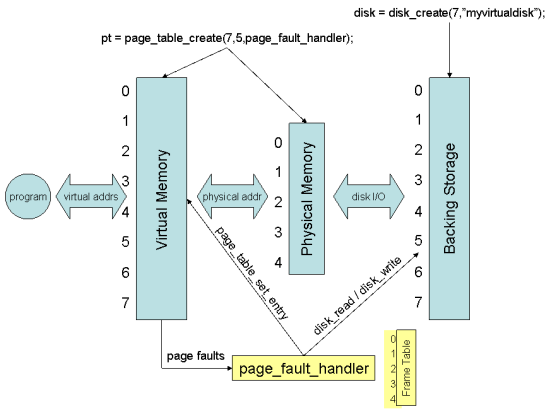
Fecha de entrega: viernes 13 de octubre de 2017

1. Objetivo

El objetivo de esta tarea es implementar paginación bajo demanda con distintos algoritmos de reemplazo de páginas, simular su funcionamiento con procesos de distinto comportamiento y comprar los resultados.

2. Descripción y Requisitos

En esta tarea deberás implementar un sistema de paginación bajo demanda. Si bien la memoria virtual es generalmente implementada en el kernel del sistema operativo, también puede ser implementada en espacio de usuario. De hecho, esta técnica es utilizada en la práctica por las máquinas virtuales modernas. En la siguiente figura, verás un resumen de los componentes del sistema de memoria virtual:



En la tarea contarás con un código base que te proveerá una tabla de páginas "virtual" y un disco "virtual" para hacer *swapping*, además el código base provee funciones para actualizar las entradas de la tabla de páginas y los bits de protección. Cuando una aplicación usa la memoria virtual, incurrirá en faltas de página que deberán ser resueltas por una función manejadora de faltas de

página. Tu trabajo consistirá en implementar dicha función, haciendo que ésta tome las decisiones de administración de memoria según sea el caso. Generalmente, la función deberá actualizar la tabla de páginas y realizar intercambio entre el disco y la memoria física.

Una vez que tu sistema funcione correctamente, podrás evaluar el desempeño de los algoritmos de reemplazo de página aleatorio (*random page replacement*), FIFO y finalmente, un algoritmo propuesto por Uds. La evaluación de desempeño deberá realizarse mediante la ejecución de programas sencillos que requieren distintos tamaños de memoria. Junto con el código, deberás entregar un informe conciso con los resultados de experimentos utilizando cada algoritmo de reemplazo de páginas y un análisis comparativo entre los algoritmos.

2.1. Para comenzar la tarea

Para empezar la tarea, descarga el código base desde Canvas y compílalo (con `make`). Si observas `main.c` verás que el programa simplemente crea la tabla de páginas y el disco virtual, y luego intenta ejecutar uno de tres "programas" utilizando la memoria virtual. Como no existe un *mapping* entre la memoria virtual y la memoria física, inmediatamente ocurre una falta de página:

```
$./virtmem 100 10 rand sort
page fault on page #0
```

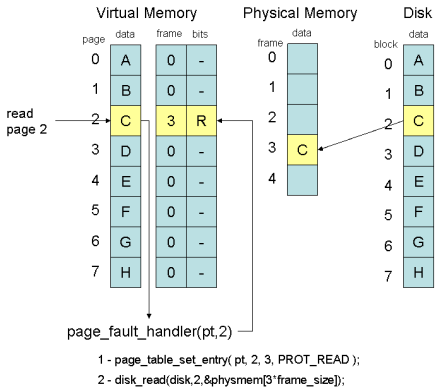
Como ves, el programa aborta inmediatamente porque no existe una función que maneje las faltas de página – ¡es tu trabajo implementarla!

Para comenzar a probar, puedes intentar mapear directamente las páginas de memoria a marcos; si ejecutas el programa con igual cantidad de marcos y páginas, no se requiere usar el disco. Simplemente puedes mapear la página N al marco N de la siguiente forma:

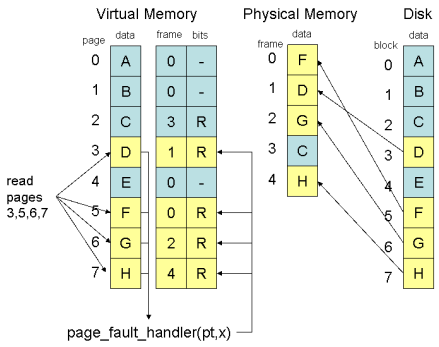
```
page_table_set_entry(pt,page,page,PROT_READ|PROT_WRITE);
```

Cualquiera de los programas funcionará bajo este esquema de paginación, pero si usas una cantidad de marcos menor a la cantidad de páginas, entonces necesitarás implementar algoritmos de reemplazo de páginas en el manejador de faltas de página.

2.2. Ejemplo de operación

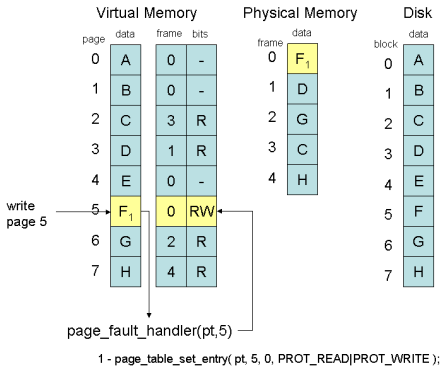


La tabla de páginas es similar a la que hemos visto en clases (más detalles en las diapositivas de clases y en el libro de Silberschatz, capítulo 9), pero no tiene bits de referencia o bit de suciedad para cada página. Solamente tiene bits de protección de lectura (`PROT_READ`), escritura (`PROT_WRITE`) y ejecución (`PROT_EXEC`), declarados en `<sys/mmap.h>`.
Mira el ejemplo de operación en la figura arriba. Imaginemos que la memoria física inicialmente está vacía. Si la aplicación comienza intentando leer la página 2, esto causará una falta de página. El manejador de falta de página escogerá un marco libre, por ejemplo, el 3. Luego ajustará la tabla de páginas para que la página 2 quede asociada al marco 3, con permisos de lectura. Finalmente, cargará la página 2 desde el disco al marco 3. Cuando el manejador de falta de página termina, la operación de lectura de la aplicación es reanudada y concluye exitosamente.

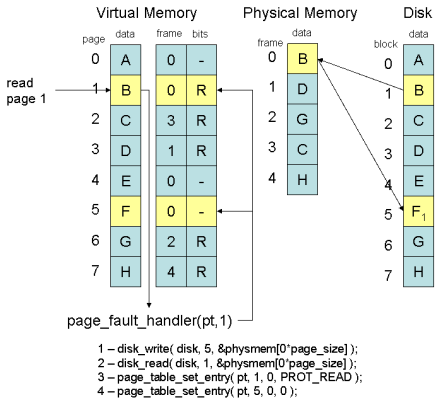


La aplicación continua ejecutando, leyendo varias páginas. Supongamos que lee las páginas 3,

5, 6 y 7, y que cada uno de estos accesos causa una falta de página. Luego de cargar las páginas necesarias, la memoria física queda totalmente utilizada, como se puede observar en la tabla de marcos ("*physical memory*" en la figura arriba).



Supongamos ahora que la aplicación intenta escribir en la página 5. Como esta página solamente tiene el bit de permiso de lectura seteado, R, ocurrirá una falta de página. El manejador de falta de página vera que la página solamente tiene seteado el bit R y ahora seteará también el bit W, para que la aplicación pueda escribir en la página. Es importante familiarizarse con las operaciones *bitwise* AND (`&`) y OR (`|`) en C para manejar estos bits[1]. Luego de setear el bit W en la página 5, el manejador de falta de página retorna, y la aplicación realiza escritura en la página 5 (marco 0) exitosamente.



[1]Operaciones bitwise de C en Wikipedia: `https://en.wikipedia.org/wiki/Bitwise_operations_in_C`

Finalmente, supongamos que la aplicación lee la página 1. La página 1 aún no está en memoria física y el manejador de falta de pagina debe decidir en este caso hacer un reemplazo de página, y escoger la página víctima. Si el manejador de falta de página escoge la página 5 (marco 0), sabe que ésta ha sido accedida para escritura anteriormente, y debe entonces llevar esta página modificada a disco, y luego carga la página 1 en el marco 0. Se actualizan dos entradas (de la página 1 y de la 5) en la tabla de páginas para reflejar esta situación.

2.3. Requisitos de la tarea

El programa debe invocarse desde la línea de comandos de la siguiente forma:

```
./virtmem -n npages -f nframes -a rand|fifo|custom -p scan|sort|focus
```

Donde cada uno de los parámetros en la línea de comandos tiene el siguiente significado:

- `n`: El número de páginas de memoria virtual que podrá tener un proceso.
- `f`: El número de marcos de memoria física en el sistema.
- `a`: El algoritmo de reemplazo de páginas a utilizar; random, FIFO o un algoritmo *custom*, es decir, creado por Uds.
- `p`: El programa a ejecutar; `scan`, `sort`, o `focus`. Cada uno de estos programas tiene patrones de acceso a memoria diferentes. Puedes revisar los programas en el archivo `program.c`.

En el código, sólo puedes modificar `main.c`, y agregar allí las funciones que requieras. La ejecución correcta del programa de la tarea debe realizar lo siguiente:

- Imprimir en la salida el resultado del programa `scan`, `sort`, o `focus` ejecutado.
- Un resumen con el número de faltas de página, lecturas y escrituras a disco (por operaciones de *swapping*). Mientras desarrollas tu solución puedes agregar impresión de otros mensajes para seguimiento de errores y depuración, pero al momento de entregar la tarea, debes desactivar estos mensajes.

Una vez que hayas implementado paginación bajo demanda con los algoritmos de reemplazo de páginas random, FIFO y custom deberás realizar una comparación entre ellos. Para cada programa `scan`, `sort`, o `focus`, ejecuta cada uno de los algoritmos, usando siempre 100 páginas y variando el número de marcos entre 2 y 100. En tu informe escribe una hipótesis para cada caso (¿cómo debería comportarse los tres algoritmos? ¿cómo se comparan?). Luego, grafica para cada programa una comparación entre los tres algoritmos, mostrando cantidad de faltas de página vs. número de marcos de memoria, cantidad de lecturas en disco vs. número de marcos de memoria, y cantidad de escrituras a disco vs. número de marcos de memoria. Para graficar usa gráficos de segmentos de línea. Si detectas que un algoritmo se comporta consistentemente mejor que otro bajo ciertas condiciones, debes destacar esto.

2.4. Consejos

Hacemos las siguientes recomendaciones:

- Para ver si una página está siendo escrita en disco o en memoria, puedes usar la función `page_table_get_entry(page, frame, bits)`. Pasando el número de la página, y punteros a variables `frame` y `bits` podrás obtener el número del marco y los permisos de acceso a la página. Si los permisos de la página son distintos de 0, entonces la página está cargada en memoria. Si los bits son 0, entonces la página no está cargada en memoria y el número del marco es irrelevante.
- Debes **crear una tabla de marcos** que mantenga el estado de cada marco de memoria física. La tabla podría construirse utilizando un simple arreglo. Esto te facilitará la búsqueda de marcos libres o marcos víctima para reemplazo.
- Para implementar el algoritmo random (en caso que tu grupo sea de 3 integrantes), usa la función `lrand48()` para generar números aleatorios, no uses `srand` o `rand`.

2.5. Forma de Trabajo

Siendo esta una tarea grupal (se permiten grupos de hasta 3 integrantes), se les recomienda encarecidamente trabajar con Git. Una buena alternativa es usar BitBucket `www.bitbucket.org` que provee repositorios privados en forma gratuita. La cuenta del profesor en BitBucket es `claudio_alvarez`. Idealmente, algún miembro del equipo puede crear un proyecto y repositorio, e invitar a sus compañeros y al profesor como participantes.

La tarea debe compilar en Ubuntu 16.04, sistema operativo oficial del curso. Si trabaja en otra plataforma asegúrese de que su tarea compila y funciona correctamente en el sistema operativo oficial, siendo que pueden existir diferencias o incompatibilidades entre distintos sistemas operativos.

Pueden agregar al código base entregado todos los archivos que requieran y además pueden modificar el `Makefile` entregado si así lo requieren.

3. Evaluación

La evaluación de la tarea contemplará los siguientes criterios:

60 % Correcta implementación de la paginación bajo demanda con cualquier patrón de acceso (i.e., todos los programas deben funcionar correctamente), variando la cantidad de memoria física y virtual disponible.

10 % Por cuidado con manejo de errores, manejo de memoria, comentarios explicativos en el código, etc.

30 % Informe con análisis de resultados de los experimentos realizados.

3.1. Modalidad de Trabajo

La tarea debe ser desarrollada en grupos de 2 (parejas). Cada integrante debe contar con una cuenta de usuario en GitHub. El desarrollo debe realizarse por ambos integrantes, y deben quedar claramente registradas sus operaciones de *commit* en el sistema, con comentarios descriptivos en cada una de estas operaciones.

Para obtener un repositorio con el código base, los integrantes de cada grupo deben acceder a: `https://classroom.github.com/g/WwjKdAJl`.

El profesor publicará en Canvas el medio a través del cual se deben inscribir los grupos e informar las cuentas de usuario en GitHub.

Se proporcionará junto con este enunciado un código base. Para compilar la tarea basta invocar `make` en el directorio en donde se encuentra el código.

4. Entrega

La tarea debe compilar; en caso contrario, la nota máxima será un 3,9 – apelable – si el error de compilación es un detalle mínimo.

El informe de la tarea debe encontrarse en el directorio raíz del repositorio Git, y debe tener nombre `informe.pdf/docx/txt`.

El lenguaje C tiene una serie de dialectos y extensiones según la versión de las herramientas que se esté usando para desarrollar. Para evitar confusiones, la evaluación será hecha compilando con `gcc` (la versión incluída en la distribución de Ubuntu de referencia), usando los parámetros `-std=c11 -Wall -Werror`.

La entrega de la tarea debe hacerse hasta las 23:59 hrs. en el repositorio Git del grupo. Para hacer efectiva esta restricción, se evaluará la última revisión del repositorio del grupo previa a la hora de cierre. Si un grupo hace su primer *commit* y/o *push* después de la hora de cierre, tendrá derecho a nota máxima 4,0.