PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2333 — Sistemas Operativos y Redes — 1/2017

Tarea 2

12 de abril de 2017

Fecha de Entrega: Miércoles 26-Abril de 2017, 23:59
Composición: grupos de n personas, donde $n \leq 2$
Esta tarea se compone de tres partes, independientes entre si.

Parte I. `msh` (myShell)

El objetivo de esta parte es construir un intérprete de comandos básico que llamaremos `msh`. Deberá recibir por `stdin` la ubicación de un ejecutable, junto con una cantidad **arbitraria** de argumentos. El ejecutable debe ser invocado con los parámetros dados, y cuando éste termine se deberá poder especificar otro ejecutable y otros parámetros para repetir el proceso. Abajo se puede ver un ejemplo de funcionamiento.

```
> Ingrese un comando: /usr/bin/echo Hello World
Hello World
> Ingrese un comando: /usr/bin/date
Fri Apr  7 17:28:13 -03 2017
> Ingrese un comando: /usr/bin/time /usr/bin/echo Hello World
Hello World
0.00user 0.00system 0:00.00elapsed 0\%CPU (0avgtext+0avgdata 1516maxresident)k
0inputs+0outputs (0major+68minor)pagefaults 0swaps
> Ingrese un comando:
```

`msh` debe reconocer algunos comandos propios (*built-in*):

- `exit` provoca que `msh` termine inmediatamente.
- `setPrompt string` provoca que el texto que precede al comando que ingresará el usuario sea `string`. Puede definir un *prompt* por defecto. Si `string` contiene un `*`, este se reemplaza por el *exitCode* del último proceso ejecutado.
- Si el último argumento es `&`, el proceso debe ser ejecutado en el *background*, esto es,
- **Bonus**: Permitir el uso de `|` entre dos comandos. Por ejemplo, `/bin/ls | /usr/bin/more` provoca que la salida de `/bin/ls` sea entregada como entrada a `/usr/bin/more`.

Al presionar Ctrl C el comportamiento varía según si se está ejecutando un comando o no. Si se está ejecutando un comando, debe terminarse el comando (y no la shell básica). En caso contrario, debe terminar `msh`.

Tenga en cuenta

1. Deberá usar `fork`, `exec` y `wait` (no *threads*).
2. Deberá construir `argc` y `argv` para pasarlos al comando que se va a ejecutar. Investigue cómo puede pasar estos valores.

Parte II. Juego de la vida

El juego de la vida es una simulación del tipo autómata celular, cuya evolución sólo depende del estado anterior. Consiste en una matriz de $n \times m$ celdas, donde cada celda tiene uno de dos estados posibles: viva o muerta. Para pasar de un estado a otro existen las siguientes reglas:

1. Una celda en (i, j) es vecina de otra celda en $(i', j') \Leftrightarrow$ son distintas y $|i - i'| \leq 1 \wedge |j - j'| \leq 1$.
2. Toda celda viva con menos de dos vecinos vivos, muere.
3. Toda celda viva con más de tres vecinos vivos, muere.
4. Toda celda muerta con **exactamente** tres vecinos vivos, se convierte en una celda viva.

El objetivo de esta parte es construir un simulador de este autómata utilizando varios procesos y memoria compartida, en un modelo de *master* con *workers*. El proceso *master* crea la matriz y setea el estado inicial. Luego, genera tantos procesos de tipo *worker* como **cores** tenga la CPU del sistema en que se ejecuta el programa. El *master* divide entre los *workers*, de forma equitativa, el trabajo de actualizar la matriz.

Su programa deberá aceptar un input con el siguiente formato mediante `stdin`:

1. La primera línea contendrá cuatro números enteros t, n, m, k separados por un espacio entre ellos, donde t es es el número de iteraciones a simular, n el número de filas de la matriz, m el número de columnas, y k el número de celdas vivas en el estado inicial. Siempre se tendrá que $t, m, n > 0$ y $0 < k \leq n \times m$.
2. Cada una de las siguientes k líneas contendrá dos números enteros i, j que representan la posición de una celda viva en el estado inicial, donde i corresponde al número de la fila, y j al número de la columna. Siempre se tendrá que $0 \leq i < n$ y $0 \leq j < m$, y también puede asumir que cada celda no aparece más de una vez.

El programa debe generar como output n líneas, donde cada línea representa una fila de la matriz después de t iteraciones. Cada línea tendrá m números enteros, que es 0 si en esa posición hay una celda muerta, o 1 si hay una celda viva. Abajo se muestra un ejemplo de input y output esperado.

Input de ejemplo

```
1 3 3 3
0 1
1 1
2 1
```

Output de ejemplo

```
0 0 0
1 1 1
0 0 0
```

Tenga en cuenta

1. Los procesos (deben ser procesos, no *threads*) deberán trabajar sobre un espacio de memoria en común.
2. Sólo debe crear los *workers* una vez.
3. Es muy probable que requiera coordinar los distintos procesos. Para ello puede utilizar semáforos que guardará en el espacio compartido de memoria. **Si no utiliza ninguna forma de coordinar los procesos, estará obligado a especificar por qué no fue necesario en su `README`.**

Parte III. Memoria Virtual

Esta parte consiste en simular las etapas involucradas en paginación y manejo de memoria virtual con *swapping*, para operaciones de lectura. Se trabajará sobre una memoria virtual con direcciones de 16 bits, donde los primeros 8 representan el número de página, y los últimos representan el offset. Además, se trabajará con una tabla de páginas de un sólo nivel y con una TLB de 32 entradas. En relación a la memoria física, habrán 128 *frames* de 2^8 bytes cada uno.

Funcionamiento

Para traducir una dirección lógica en una física, se intenta consultar el número de página en la TLB. Si hubo un TLB-*hit*, el número de *frame* es obtenido de la TLB. En caso de un TLB-*miss*, se debe consultar la tabla de páginas. En el último caso, es posible obtener el número de *frame* de la tabla de páginas u obtener un *page-fault*.

En el caso de un *page-fault*, usted deberá implementar **paginación por demanda**. Para esto, el archivo `data.bin` actuará como "disco", por lo que ante un *page-fault* deberá leer el *frame* correspondiente en el archivo, y cargarlo en la memoria física. Una vez que haya cargado el *frame*, hay que actualizar la TLB y la tabla de páginas.

Tome en cuenta que el espacio de direcciones físicas es menor al de direcciones virtuales, por lo que habrá *swapping*. Es por ello que deberá implementar una política de reemplazo de páginas `LRU`. Implemente esta misma política para actualizar la TLB.

Input

Su programa recibirá mediante `stdin` una secuencia de enteros que representan direcciones de memoria lógicas, desde el 0 hasta el 65535, además de un archivo llamado `data.bin` de 65536 bytes que se encontrará en el mismo directorio del ejecutable.

Output

Para cada dirección lógica leída, deberá traducirla a su respectiva dirección física e imprimir en pantalla el valor del *byte* (sin signo) correspondiente. Además, al finalizar el programa deberá reportar el porcentaje de *page-faults* y de TLB-*hits*.

Tome en cuenta

1. Por cada *page-fault* va a tener que leer el archivo `data.bin` y buscar una cierta posición. Se recomienda usar las funciones `fopen`, `fread`, `fseek` y `fclose`. **No mantenga este archivo en memoria, no es la idea de la tarea.**
2. En esta simulación la TLB no ayuda a hacer más rápido encontrar una página (de hecho lo contrario).
3. La tarea no es complicada. Debe modelar una TLB, una tabla de páginas y un espacio para la memoria física.

README

Deberá incluir un archivo `README` que indique quiénes son los autores de la tarea, cómo compilarla y correr sus programas, a grandes rasgos el funcionamiento de cada programa y cuáles fueron las decisiones de diseño involucradas, qué supuestos adicionales ocuparon, entre otras cosas que considere necesarias para una mejor corrección de su tarea. Se sugiere utilizar formato **markdown**.

Formalidades

La tarea será entregada mediante `git` en un repositorio **privado** que ustedes deberán crear. Se revisará el contenido de la rama `master` el día Miércoles 26-Abril de 2017, 23:59.

- Puede ser realizada en forma individual, o en grupos de 2 personas.
- Su tarea deberá compilar de acuerdo a las instrucciones de su `README`.
- Su repositorio **DEBE ser privado**, de lo contrario calificará como copia (alguien les podría haber copiado). Esta restricción es fundamental. Si no la cumple **no** se revisará su tarea.
- La entrega será automatizada, basta que **registren** su grupo y repositorio mediante un formulario que habilitaremos para ello (no por email).
- Como el repositorio debe ser privado, tendrá que permitir acceso especial al curso, autorizando al servidor de tareas acceder al contenido. Para esto basta registrar la llave pública del curso en las **Deployment Keys** de su repositorio.
- Lo mejor es que use Bitbucket.

Evaluación

- 10 %. Formalidades. Esto incluye cumplir las normas de la sección formalidades.
- 20 %. Shell básica
  - 8 % Lectura de `stdin`. Paso de argumentos. Construcción de `argc` y `argv`.
  - 12 % Funcionamiento correcto.
- 35 %. Juego de la vida.
  - 10 % Correctitud del algoritmo, input y output.
  - 25 % Uso de memoria compartida, varios procesos y sincronización entre ellos.
- 35 %. Memoria virtual.
  - 12 % Modelación de TLB (con LRU) y tabla de páginas.
  - 12 % Paginación por demanda. Reemplazo de páginas LRU.
  - 3 % Modelación de frames.
  - 8 % Correctitud y estadísticas.

El no respeto de las formalidades o un código extremadamente desordenado podría originar descuentos adicionales.

Preguntas

A través del foro de EdX.