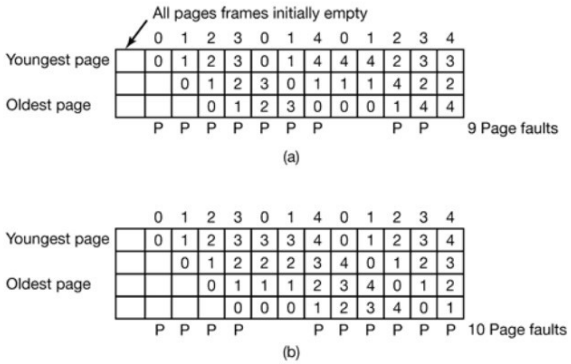
PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

SISTEMAS OPERATIVOS
Examen 2
(Segundo semestre de 2022)

Horarios 0781, 0782: prof. V. Khlebnikov

Duración: 3 horas
Notas: No está permitido el uso de ningún material o equipo electrónico.
La presentación, la ortografía y la gramática influirán en la calificación.
Puntaje total: 20 puntos

Pregunta 1 (10 puntos – 1 hora 30 min.) Intuitively, it might seem that the more page frames the memory has, the fewer page faults a program will get. Surprisingly enough, this is not always the case. Belady et al. (1969) discovered a counterexample, in which FIFO caused more page faults with four page frames than with three. This strange situation has become known as Belady's anomaly. It is illustrated in Fig.1 for a program with five virtual pages, numbered from 0 to 4. The pages are referenced in the order 0 1 2 3 0 1 4 0 1 2 3 4. In Fig.1(a) we see how with three page frames a total of nine page faults are caused. In Fig.1(b) we get ten page faults with four page frames.



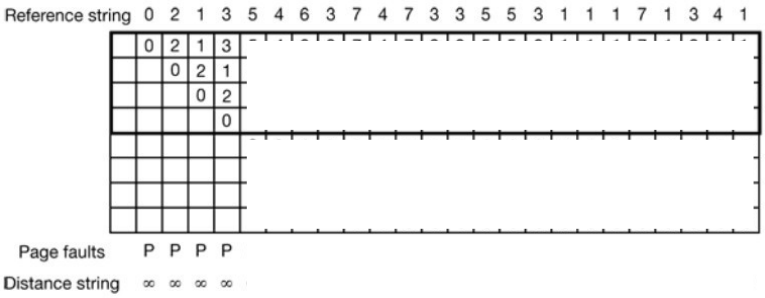
Many researchers in computer science were dumbfounded by Belady's anomaly and began investigating it. This work led to the development of a whole theory of paging algorithms and their properties. All of this work begins with the observation that every process generates a sequence of memory references as it runs. Each memory reference corresponds to a specific virtual page. Thus conceptually, a process' memory access can be characterized by an (ordered) list of page numbers. This list is called the **reference string**, and plays a central role in the theory. For simplicity, in the rest of this section we will consider only the case of a machine with one process, so each machine has a single, deterministic reference string (with multiple processes, we would have to take into account the interleaving of their reference strings due to the multiprogramming).

A paging system can be characterized by three items: (1) the reference string of the executing process; (2) the page replacement algorithm; (3) the number of page frames available in memory, *m*. Conceptually, we can imagine an abstract interpreter that works as follows. It maintains an internal array, *M*, that keeps track of the state of memory. It has as many elements as the process has virtual pages, which we will call *n*. The array *M* is divided into two parts. The top part, with *m* entries, contains all the pages that are currently in memory. The bottom part, with *n* - *m* pages, contains all the pages that have been referenced once but have been paged out and are not currently in memory. Initially, *M* is the empty set, since no pages have been referenced and no pages are in memory. As execution begins, the process begins emitting the pages in the reference string, one at a time. As each one comes out, the interpreter checks to see if the page is in memory (i.e., in the top part of *M*). If it is not, a page fault occurs. If there is an empty slot in memory (i.e., the top part of *M* contains fewer than *m* entries), the page is loaded and entered in the top part of *M*. This situation arises only at the start of execution. If memory is full (i.e., the top part of *M* contains *m* entries), the page replacement algorithm is invoked to remove a page from memory. In the model, what happens is that one page is moved from the top part of *M* to the bottom part, and the needed page entered into the top part. In addition, the top part and the bottom part may be separately rearranged.

(a) (1 punto – 9 min.) Assume that you have a page-reference string for a process with *m* frames (initially all empty). The page-reference string has length *p*; *n* distinct page numbers occur in it. Answer this question for any page-replacement algorithms: What are a lower and an upper bounds on the number of page faults?

To make the operation of the interpreter clearer let us look at a concrete example using LRU page replacement. The virtual address space has eight pages and the physical memory has four page frames. At the top of Fig.2 we have a reference string consisting of the 24 pages: 0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 4 1. Under the reference string, we have 25 columns of 8 items each. The first column, which is empty, reflects the state of *M* before execution begins. Each successive column shows *M* after one page has been emitted by the reference and processed by the paging algorithm. The heavy outline denotes the top of *M*, that is, the first four slots, which correspond to page frames in memory. Pages inside the heavy box are in memory, and pages below it have been paged out to disk.

The first page in the reference string is 0, so it is entered in the top of memory (because this page is the least recently used), as shown in the second column. The second page is 2, so it is entered at the top of the third column (because now the page 2 is the least recently used). This action causes 0 to move down. The contents of *M* exactly represent the contents of the LRU algorithm.



(b) (2 puntos – 18 min.) Complete the Fig.2 using the LRU algorithm.

(c) (1 punto – 9 min.) ¿Cuál estructura de datos es la más apropiada para implementar el procedimiento descrito?

Although this example uses LRU, the model works equally well with other algorithms. In particular, there is one class of algorithms that is especially interesting: algorithms that have the property

$$M(m, r) \subseteq M(m + 1, r)$$

where *m* varies over the page frames and *r* is an index into the reference string. What this says is that the set of pages included in the top part of *M* for a memory with *m* page frames after *r* memory references are also included in *M* for a memory with *m* + 1 page frames. In other words, if we increase memory size by one page frame and re-execute the process, at every point during the execution, all the pages that were present in the first run are also present in the second run, along with one additional page.

From examination of Fig.2 and a little thought about how it works, it should be clear that LRU has this property. Algorithms that have this property are called **stack algorithms**. These algorithms do not suffer from Belady's anomaly and are thus much loved by virtual memory theorists.

(d) (1 punto – 9 min.) Why stack based algorithms do not suffer Belady's anomaly? Because these type of algorithms assigns a priority to a page (for replacement) that is independent of … (**complete la sentencia)**

For stack algorithms, it is often convenient to represent the reference string in a more abstract way than the actual page numbers. A page reference will be henceforth denoted by the distance from the top of the stack where the referenced page was located. For example, the reference to page 1 in the last column of Fig.2 is a reference to a page at a distance 3 from the top of the stack (because page 1 was in third place *before* the reference). Pages that have not yet been referenced and thus are not yet on the stack (i.e., not yet in *M*) are said to be at a distance ∞. The distance string for Fig.2 is given at the bottom of the figure.

(e) (1 puntos – 9 min.) Complete the distance string of Fig.2.

Note that the distance string depends not only on the reference string, but also on the paging algorithm. With the same original reference string, a different paging algorithm would make different choices about which pages to evict. As a result, a different sequence of stacks arises.

One of the nice properties of the distance string is that it can be used to predict the number of page faults that will occur with memories of different sizes. We will demonstrate how this computation can be made based on the example of Fig.2. The goal is to make one pass over the distance string and, from the information collected, to be able to predict how many page faults the process would have in memories with 1, 2, 3, ..., *n* page frames, where *n* is the number of virtual pages in the process' address space.

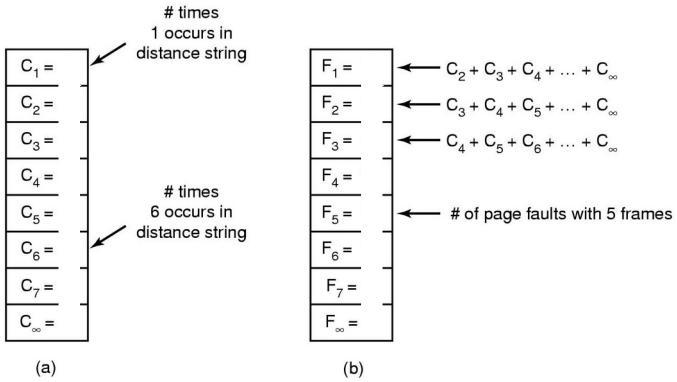
The algorithm starts by scanning the distance string, page by page. It keeps track of the number of times 1 occurs, the number of times 2 occurs, and so on. Let C_i be the number of occurrences of *i*. For the distance string of the figure, the *C* vector is illustrated in Fig.3(a). Let C_∞ be the number of times ∞ occurs in the distance string.

Now compute the *F* vector according to the formula $$F_m = \sum_{k=m+1}^{n} C_k + C_\infty$$

The value of F_m is the number of page faults that will occur with the given distance string and *m* page frames.

For the distance string of Fig.2, Fig.3(b) gives the *F* vector.

(f) (2 puntos – 18 min.) Complete the values of Fig.3(a) and (b).



To see why this formula works, go back to the heavy box in Fig.2. Let *m* be the number of page frames in the top part of *M*. A page fault occurs any time an element of the distance string is *m* + 1 or more. The summation in the formula above adds up the number of times such elements occur.

(g) (2 puntos – 18 minutos) Consider the following page reference string: 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6. How many page faults would occur for the optimal replacement algorithm, assuming one, two, three, four, five, six, or seven frames? Remember all frames are initially empty, so your unique pages will all cost one fault each.

<u>Pregunta 2</u> (10 puntos – 1 hora 30 min.) (*http://digital-forensics.sans.org/blog/2011/03/28/digital-forensics-understanding-ext4-part-3-extent-trees*) Nos interesa cómo se colocan los bloques de un archivo grande (1.5GB) en el sistema de archivos *ext4* (de la partición `/dev/sda7`):

```
$ ls -li linuxmint-17.3-cinnamon-64bit.iso
19964250 -rw-r--r-- 1 vk vk 1581383680 dic  5 12:14 linuxmint-17.3-cinnamon-64bit.iso
```

Este archivo es la imagen de DVD de instalación de la versión de Linux Mint 17.3 “Rosa” con el entorno de escritorio Cinnamon que contiene 386080 (= 1581383680 / 4096) bloques de 4KB.

(a) (1 punto) ¿Cuántos bloques ocuparían todas las entradas de FAT32 solo para este archivo si estarían juntas?

(b) (2 puntos) ¿Cuántos bloques ocuparían todos los punteros a bloques de *inode* para este archivo en el *ext2*?

Pero colocamos este archivo en el sistema de archivos *ext4* creado en el dispositivo `/dev/sda7`. Lo verificamos:

The utility `file` determines file type (option `-s` is for special files):

```
# file -sL /dev/sda7
/dev/sda7: Linux rev 1.0 ext4 filesystem data, UUID=4225d72e-9cce-4a6e-aabd-c12f7778cb1b (needs journal recovery) (extents) (large files) (huge files)
```

The utility `blkid` prints block device attributes:

```
$ blkid /dev/sda7
/dev/sda7: UUID="4225d72e-9cce-4a6e-aabd-c12f7778cb1b" TYPE="ext4"
```

El utilitario `tune2fs` proporciona la información más detallada:

```
# tune2fs -l /dev/sda7
tune2fs 1.42.9 (4-Feb-2014)
Filesystem volume name:   <none>
Last mounted on:          /
Filesystem UUID:          4225d72e-9cce-4a6e-aabd-c12f7778cb1b
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir_index filetype needs_recovery extent flex_bg
sparse_super large_file huge_file uninit_bg dir_nlink extra_isize
Filesystem flags:         signed_directory_hash
Default mount options:    user_xattr acl
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              27303936
Block count:              109207040
Reserved block count:     5460352
Free blocks:              83528046
Free inodes:              26886776
```

```
First block:              0
Block size:               4096
Fragment size:            4096
Reserved GDT blocks:      997
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         8192
Inode blocks per group:   512
Flex block group size:    16
Filesystem created:       Sat Aug 15 14:00:43 2015
Last mount time:          Tue Nov 17 12:38:20 2015
Last write time:          Tue Nov 17 12:38:19 2015
Mount count:              18
Maximum mount count:      -1
Last checked:             Sat Aug 15 14:00:43 2015
Check interval:           0 (<none>)
Lifetime writes:          2190 GB
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
Inode size:               256
Required extra isize:     28
Desired extra isize:      28
Journal inode:            8
First orphan inode:       4849717
Default directory hash:   half_md4
Directory Hash Seed:      e682682a-db28-4b55-a091-8964c91e064f
Journal backup:           inode blocks
```

The utilily `fsstat` displays the details associated with a file system:

```
# fsstat /dev/sda7 | less

FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: Ext4
Volume Name:
Volume ID: 1bcb78772fc1bdaa6e4ace9c2ed72542

Last Written at: Tue Nov 17 12:38:19 2015
Last Checked at: Sat Aug 15 14:00:43 2015

Last Mounted at: Tue Nov 17 12:38:20 2015
Unmounted properly
Last mounted on: /

Source OS: Linux
Dynamic Structure
Compat Features: Journal, Ext Attributes, Resize Inode, Dir Index
InCompat Features: Filetype, Needs Recovery,
Read Only Compat Features: Sparse Super, Has Large Files,

Journal ID: 00
Journal Inode: 8

METADATA INFORMATION
--------------------------------------------
Inode Range: 1 - 27303937
Root Directory: 2
Free Inodes: 26886776
Orphan Inodes: 6422937, 6422936, 6422934, 4849717, 5771291, 26484315, 25435015, 6422923, 6422921, ...,

CONTENT INFORMATION
--------------------------------------------
Block Range: 0 - 109207039
Block Size: 4096
Free Blocks: 83528046

BLOCK GROUP INFORMATION
--------------------------------------------
Number of Block Groups: 3333
Inodes per group: 8192
Blocks per group: 32768

Group: 0:
  Inode Range: 1 - 8192
  Block Range: 0 - 32767
  Layout:
    Super Block: 0 - 0
    Group Descriptor Table: 1 - 27
    Data bitmap: 1025 - 1025
    Inode bitmap: 1041 - 1041
```

```
    Inode Table: 1057 - 1568
    Data Blocks: 1569 - 32767
  Free Inodes: 8177 (99%)
  Free Blocks: 23513 (71%)
  Total Directories: 2
...
```

El *inode* que nos interesa (`19964250`) está en el `Group 2437` (= 19964250 / 8192):

```
Group: 2437:
  Inode Range: 19963905 - 19972096
  Block Range: 79855616 - 79888383
  Layout:
    Super Block: 79855616 - 79855616
    Group Descriptor Table: 79855617 - 79855643
    Data bitmap: 79691781 - 79691781
    Inode bitmap: 79691797 - 79691797
    Inode Table: 79694368 - 79694879
    Data Blocks: 79694880 - 79888383
  Free Inodes: 7282 (88%)
  Free Blocks: 0 (0%)
  Total Directories: 0
```

Antes de nuestro *inode* en este grupo hay 19964250 – 19963905 = 345 *inodes*. Cada *inode* ocupa 256 (2^8) bytes, cada bloque es de 4K (2^{12}) bytes, entonces, en cada bloque hay 16 (2^4) *inodes*. Los 345 *inodes* ocupan 345 / 16 = 21 bloques completos (con 336 *inodes*), entonces nuestro *inode* está (después de 9 *inodes*) en el bloque 79694368 + 21 = 79694389.

Ahora podemos extraer el bloque que contiene el *inode* que nos interesa a un archivo:

```
# blkcat /dev/sda7 79694389 >blk-with-our-inode
```

El contenido del bloque se puede analizar con un editor hexadecimal, por ejemplo, `ghex` (*a GNOME Hex Editor*). Cada *inode* es de 256 (0x100) bytes, nos interesa el *inode* numero 9 (= 345 – 336) del bloque, o sea, él con el offset 0x900:

```
00000900   A4 81 E8 03 00 00 42 5E F8 6C 64 56 78 1B 63 56            inode #19964250
00000910   78 1B 63 56 00 00 00 00 E8 03 01 00 50 21 2F 00
00000920   00 00 08 00 01 00 00 00 0A F3 01 00 04 00 02 00
00000930   00 00 00 00 00 00 00 00 9A 51 C4 04 00 00 0E 00
00000940   64 10 00 00 3B 80 0E 00 00 00 0E 00 00 3E 00 00
00000950   69 82 0F 00 00 00 0F 00 00 78 02 00 F2 93 A8 00
00000960   00 00 0F 00 00 EF 60 C9 00 00 00 00 00 00 00 00
00000970   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000980   1C 00 00 00 D4 8E FC 3C D8 CE 35 0F 08 C7 93 D4
00000990   0F 16 63 56 8C 8C 06 13 00 00 00 00 00 00 00 00
000009A0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000009B0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000009C0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000009D0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000009E0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000009F0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Según la estructura de `inode` en `ext4`, con el *offset* 4, en 4 bytes, está el tamaño del archivo: `00 00 42 5E`, el valor que en *little-endian* será 0x5E420000, o 1581383680, lo que es exactamente el tamaño de nuestro archivo según los programas `ls` y `stat`.

Si `ext2` y `ext3` tienen `inodes` de tamaño 128 bytes y cada `inode` contiene 15 punteros a bloques (12 directos y 3 indirectos), `ext4` tiene `inodes` de tamaño 256 bytes. La bandera 0x80000 (EXT4_EXTENTS_FL en el *offset* 0x20) indica que nuestro `inode` usa `extents`. Por eso, en el offset 0x28 no están 15 punteros sino el `extent tree`. O sea, los `extents` se organizan en un árbol. Cada nodo de este árbol comienza con su *header* que tiene tamaño de 12 bytes:

bytes
- `0x0-1`: Magic number (`0xF30A`)
- `0x2-3`: Number of valid entries following the header (`0x0001 = 1`)
- `0x4-5`: Maximum number of entries that could follow the header (`0x0004 = 4`)
- `0x6-7`: Depth of this extent node in the extent tree (`0x0002 = 2`). 0 = this extent node points to data block; otherwise, this extent node points to other extent nodes.
- `0x8-b`: Generation of the tree (`0x00000000 = 0`)

When EXT4 needs to use more than four extents, it creates a tree structure on disk for holding the necessary extent fields -- that's what the "depth of tree" field in the extent header is trying to help us with. The leaf nodes at the very bottom of the tree are regular extent structures like we saw in Part 1. But the "interior" nodes in the rest of the tree are a different kind of structure called an *extent index*. We know we're dealing with an extent index structure here because "depth of tree" is non-zero, so we're not at a leaf node.

Internal nodes of the extent tree, also known as index nodes, are 12 bytes long with the following structure:

bytes `0x0-3`: Logical block number (del archivo), this index node covers file blocks from 'block' onward.
`0x4-7`: Lower 32 bits of physical block address of the extent node that is the next level lower in the tree. The tree node pointed to can be either another internal node or a leaf node.
`0x8-9`: Upper 16 bits of the previous field.
`0xa-b`: Not used.

Essentially the extent header contains two values. The first is the logical block where the extents found beneath this node in the tree begin. The other value in the extent index is the physical block number of a data block that holds the information about the next level in our tree.

(c) (2 puntos) ¿Cuáles son estos dos valores en hexadecimal?

Ahora sabemos en qué bloque está el nodo de siguiente nivel del árbol y lo podemos desplegar:

```
# blkcat -h /dev/sda7 0x... | less
0x000   0af30900 54010100 00000000 00000000
0x010   b278c404 00000e00 64100000 3b800e00
0x020   00000e00 003e0000 69820f00 00000f00
0x030   bc9a0100 b702a300 00000f00 00780200
0x040   f293a800 00000f00 f8940200 3f94a800
0x050   00000000 00ce0200 b3812a01 00000000
0x060   00f00300 01843801 00000000 005b0400
0x070   43813901 0000f9ff 00000000 0080f9ff
0x080   00000000 0080f9ff 00000000 0080f9ff
0x090   000e9681 2f7ffcff 806f9f16 2f7f0000
0x0a0   c03a4552 2f7f0000 40d2b25d 2f7f0000
0x0b0   00000000 00000000 30219ebc 2f7f0000
```

Efectivamente, aquí encontramos el número mágico del *extent tree header* y sus otros campos.

(d) (3 puntos) Interpretando la información de los campos del *extent tree header*. ¿a cuántos *extent nodes* apunta este nodo interno de árbol? ¿En que direcciones (hexadecimales) comienza y termina cada nodo interno apuntado? ¿Qué número hexadecimal del bloque lógico del archivo y qué número hexadecimal del bloque físico contiene el ***último*** nodo interno apuntado?

Ahora desplegamos el bloque físico que está apuntado por el último nodo interno del árbol (de nivel 1):

```
# blkcat -h /dev/sda7 0x... | less
0x000   0af31600 54010000 e5e5e5e5 005b0400
0x010   00050000 00bb3901 00600400 00600000
0x020   00e03901 00c00400 00280000 00583a01
0x030   00e80400 00080000 00f83a01 00f00400
0x040   00080000 00183b01 00f80400 00100000
0x050   00503b01 00080500 00100000 00983b01
0x060   00180500 00100000 00b03b01 00280500
0x070   00080000 00403c01 00300500 00080000
0x080   00d83f01 00380500 00200000 00e04001
```

Según el encabezado, se puede observar que aquí están los nodos-hojas que apuntan a los bloques con datos de nuestro archivo. La estructura de estos nodos (también de 12 bytes) es la siguiente:

bytes `0x0-3`: First file block number that this extent covers.
`0x4-5`: Number of blocks covered by extent.
`0x6-7`: Upper 16 bits of the block number to which this extent points.
`0x8-b`: Lower 32 bits of the block number to which this extent points.

(e) (2 puntos) ¿Cuántos nodos-hojas están presentes en este bloque? El primer nodo-hoja, ¿cuáles bloques lógicos del archivo cubre y en qué bloques físicos están éstos?

Preparado por VK
con LibreOffice Writer en Linux Mint 21 “Vanessa”

Profesor del curso: V. Khlebnikov

Pando, 7 de diciembre de 2022