PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

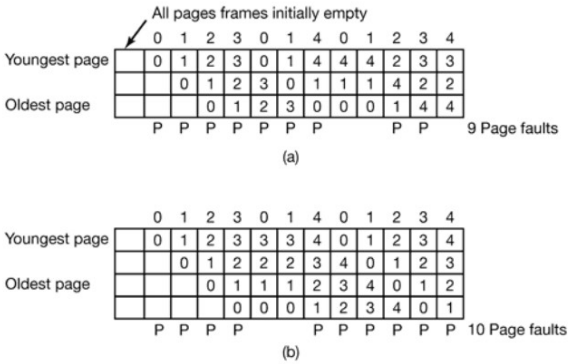
SISTEMAS OPERATIVOS

4ta práctica (tipo a)
(Primer semestre de 2018)

Horario 0781: prof. V. Khlebnikov
Horario 0782: prof. A. Bello R.

Duración: 1 h. 50 min.
Nota: No se puede usar ningún material de consulta.
La presentación, la ortografía y la gramática influirán en la calificación.
La práctica debe ser desarrollada en el cuadernillo usando lapicero.
Lo escrito con lápiz NO será evaluado.
Puntaje total: 20 puntos

Pregunta 1 (10 puntos – 50 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.



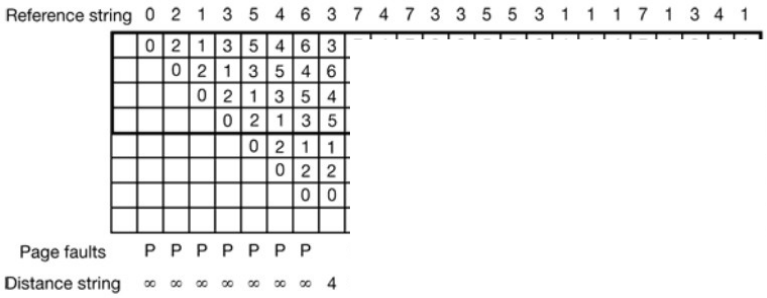
Many researchers in computer science were dumbfounded by Belady's anomaly and began investigating it. This work led to the development of a whole theory of paging algorithms and their properties. All of this work begins with the observation that every process generates a sequence of memory references 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) (2 puntos – 10 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) (3 puntos – 15 min.) Complete the Fig.2 using the LRU algorithm.

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.

(c) (1 punto – 5 min.) ¿Cuáles son los conjuntos de páginas $M(4, 17)$ y $M(5, 17)$ (después de la referencia a la página 1)?

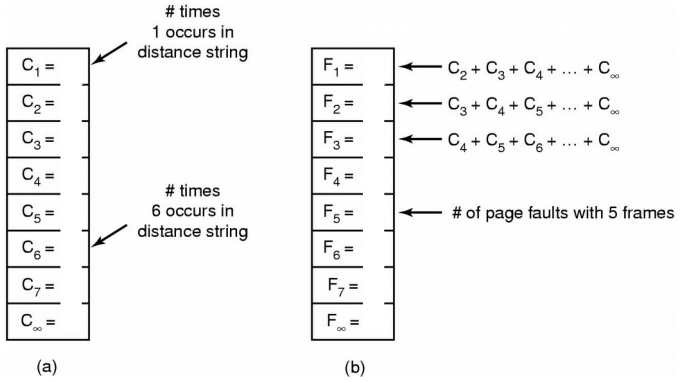
(d) (1 punto – 5 min.) En la Fig. 1 (el algoritmo FIFO, al inicio de la pregunta 1), ¿después de qué referencia *r* no se cumple la propiedad $M(3, r) \subseteq M(4, r)$?

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 3 in the 9th column of Fig.2 is a reference to a page at a distance 4 from the top of the stack (because page 3 was in 4th 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 punto – 5 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 – 10 min.) Complete the values of Fig.3(a) and (b) (**Fe de erratas**: en la figura faltan C_8 y F_8, usted las debe añadir. Tampoco puede existir F_∞.)

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.

<u>Pregunta 2</u> (5 puntos – 25 min.) Local versus Global allocation Policies

In the preceding sections we have discussed several algorithms for choosing a page to replace when a fault occurs. A major issue associated with this choice (which we have carefully swept under the rug until now) is how memory should be allocated among the competing runnable processes.

Take a look at Fig. 3-22(a). In this figure, three processes, A, B, and C, make up the set of runnable processes. Suppose A gets a page fault. Should the page replacement algorithm try to find the least recently used page considering only the six pages currently allocated to A, or should it consider all the pages in memory? If it looks only at A's pages, the page with the lowest age value is A5, so we get the situation of Fig. 3-22(b).

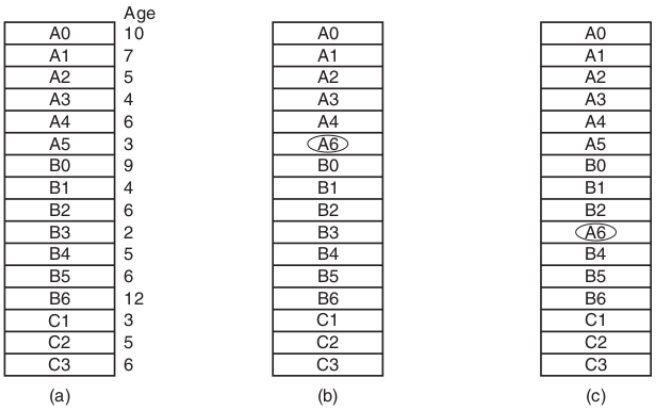


Figure 3-22. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement.

On the other hand, if the page with the lowest age value is removed without regard to whose page it is, page B3 will be chosen and we will get the situation of Fig. 3-22(c). The algorithm of Fig. 3-22(b) is said to be a **local** page replacement algorithm, whereas that of Fig. 3-22(c) is said to be a **global** algorithm.

Sea un sistema de gestión de memoria virtual con paginación por demanda, con un tamaño de memoria principal de 5000 bytes y un tamaño de página de 1000 bytes. En un momento determinado se tienen 3 procesos P1, P2 y P3 en el sistema que generan la siguiente secuencia de direcciones lógicas (se han representado pares compuestos por proceso y la dirección lógica): (P1,1023) (P2,224) (P1,783) (P3,3848) (P3,1089) (P3,98) (P2,2345) (P1,787) (P1,1654) (P3,2899) (P3,3008) (P3,1111). Elabore un diagrama (cuadro de doble entrada) de la situación de cada página en memoria física utilizando el algoritmo de reemplazo óptimo, en los siguientes casos:

a) (2 puntos – 10 min.) Con política de ubicación global
b) (3 puntos – 15 min.) Con política de ubicación local

<u>Pregunta 3</u> (5 puntos – 25 min.) En un determinado sistema con memoria virtual con paginación por demanda, una dirección lógica consta de 16 bits, 10 de offset (desplazamiento) y 6 para el número de página. Se dispone de 4 marcos. Dada la siguiente secuencia de direcciones lógicas:

512 1102 2147 3245 5115 5200 4090 4207 1070 6200
7168 8200 7200 8300 9300 7410 8525 9700 5300 4387 1007

Se pide:

a) (1 punto – 5 min.) Escriba la secuencia de referencias a páginas.
b) (4 puntos – 20 min.) Indicar y contar el número de fallos de página suponiendo los algoritmos de reemplazo FIFO y LRU.

La práctica ha sido preparada por AB (2,3) y VK (1)
en Linux Mint 18.3 Sylvia con LibreOffice Writer.

Profesores del curso: (0781) V. Khlebnikov
(0782) A. Bello R.

Pando, 8 de junio de 2018