CS612 Algorithm Design and Analysis

Lecture 16. Paging problem [1]

Dongbo Bu

Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China

[1] The slides are made based on Algorithm Design, Randomized algorithm by R. Motwani and P. Raghavan, and a lecture by T. Chan.

Outline

- Introduction
- Greedy algorithm: Furthest-Future principle;
- Label on-line algorithm framework;
- The performance of LRU principle;
- A randomized on-line algorithm (Fiat et al '91);

PAGING problem I

INPUT:
Given a sequence of requests $r_1, r_2, ..., r_n$, and a cache of size k;
OUTPUT:
schedule the eviction decisions to reduce cache-missing as much as possible.

An example I

An eviction sequence: see a fig.

A dynamic-programming method I

- Subproblem: finding the optimal evictions for requests $r_i ... r_n$ when cache contents are $c_1, c_2, ..., c_k$, $c_i \in \Sigma$, $|\Sigma| = m$.
- Let $OPT(i, c_1, c_2, .., c_k)$ be the optimal solution value to the subproblem. We have the following recursion:

  $OPT(i, c_1, c_2, ..., c_k) = \min \begin{cases} OPT(i+1, c'_1, c'_2, ..., c'_k) + 1 \\ OPT(i+1, c_1, c_2, ..., c_k) \end{cases}$ and

  $OPT(n, c_1, c_2, ..., c_k) = 0$.

  Here, $c'_1, c'_2, ..., c'_k$ differs from $c_1, c_2, ..., c_k$ at only one page.

Time-complexity: DP table size: $O(nC_m^k)$. Filling each entry takes $k(m-k)+1$ time.

A greedy solution: Furthest-Future principle (L. Belady, '66) I

FF rule: evicts the farthest-future element;
Furthest-Future eviction sequence S_{FF}: see a fig.

Theorem

S_{FF} incurs no more missing than any other schedule S^* and hence is optimal.

Proof:

- Exchange argument again!.
- Basic idea: From an optimal schedule S^*, we generate a series of schedule $S_1, S_2, ..., S_n$, such that:
  1. The first i evictions of S_i are the same to that of S_{FF}. Thus, $S_n = S_{FF}$.
  2. S_{i+1} incurs no more missing than S_i.

A greedy solution: Furthest-Future principle (L. Belady, '66) II

S0=S*	—	—	—	7	5	3	0	5	7
S1	—	—	—	$\boxed{3}$	5	7	0	5	7
S2	—	—	—	$\boxed{3}$	$\boxed{0}$	7	0	5	7
S_FF	—	—	—	$\boxed{3}$	$\boxed{0}$	9	—	—	7

Basic idea: interpolating a sequence of Si to change S* to S_FF.
Two requirements:
1. Si: the first i evitions of Si are same to that of S_FF.
2. Si+1 incurs no more evictions than Si.

- Difficulty: how to construct S_{i+1} based on S_i?
  1. Consider the $j+1$-th request d. S_i and S_{FF} have the same cache content till now.

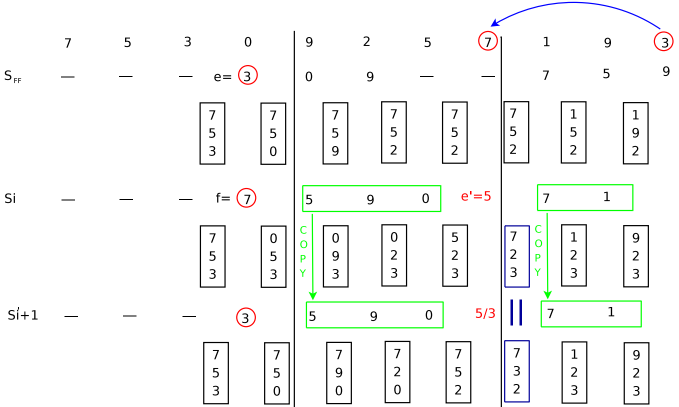
A greedy solution: Furthest-Future principle (L. Belady, '66) III

2. Suppose S_i evicts f but S_{FF} evicts $e \neq f$. We design S_{i+1} as follows:
3. S_{i+1} evicts e at the $j+1$-th step. Now, S_{i+1} and S_i has different cache content.
4. S_{i+1} simulates the actions of S_i from $j+2$ step until the following two events occurs:
5. Case 1: request $g \neq e, f$, and S_i evict e.
   We let S_{i+1} evict f. The cache are same now. Thus, we can copy the remaining of S_i to S_{i+1}.
6. Case 2: request f and S_i evicts e'.
   We let S_{i+1} evict e', too. And fill e if needed.

Key: the Furthest-Future principle ensures that before an request of e, there should be a request of f (Case 1).

Case 1:

FF evicts e rather than f since e is a request farther than f.

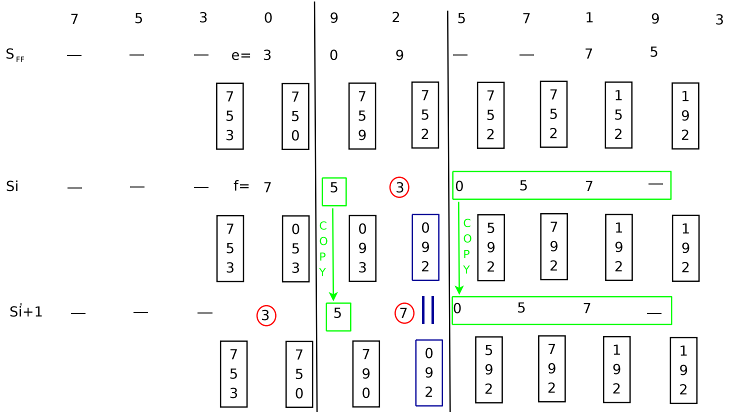


1. Si+1 simulate the action of S_FF first. Thus, Si+1 differs from Si only at 'e' and 'f'.

2. Then copy Si's actions.
3. When a request 'f' arrives, it is a good chance for Si+1 to change to Si by simply evicting e'.

4. Now Si+1 and Si have the same cache content; thus Si+1 can simply copy Si's actions.

Case 2:



1. Si+1 simulate the action of S_FF first. Thus, Si+1 differs from Si only at 'e' and 'f'.
2. Then copy Si's actions.
3. When Si evicts 'e', Si+1 evicts 'f' to reduce the difference.
4. Now Si+1 and Si have the same cache content; thus Si+1 can simply copy Si's actions.

From FF to LRU

- LRU: least recently used.
- Intuition: "longest in the past rather than the farthest in the future" since we have no idea of the future requests.
- Reason: locality of reference, i.e., a program will generally keep accessing the things it has just been accessing.

Theoretical analysis of LRU principle (Sleator and Tarjan) I

Key idea: divide the requests into "phases". Each phase consists of a set of evictions.

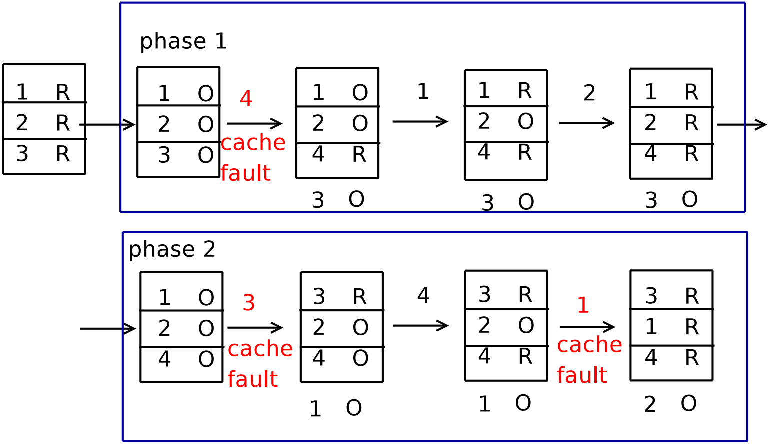
A Label-based algorithm framework:

1. initially make all pages in cache as "old";
2. when a request of page P arrives,
3. mark P "recent";
4. if P is not in cache
5. if all cache pages are "recent",
6. remark all pages "old", and begin a new phase;
7. choose an old page q to evict;

Old: the pages have already been loaded before this phase.

Recent: the pages are loaded in this phase.

Theoretical analysis of LRU principle (Sleator and Tarjan) II



Theoretical analysis of LRU principle (Sleator and Tarjan) III

Analysis:
Suppose there are r phases.

- Fact 1: Every phase contains k distinct requests. (Reason: when a page changes from "Old" to "Recent", it will stay in cache till the phase ends.)
- Fact 2: At each phase, there are at most k evictions. Thus, there are at most rk evictions. (Intuition: evicting a page cause a page remarking from "Old" to "Recent".)
- Fact 3: An optimal solution incurrs at least $r - 1$ missing. (Reason: the first request of a phase i cause a remarking of a page from "Old" to "Recent".)
- Therefore, the ratio of any Label-based algorithm is k.

Worst-case: repeating a cycle of requests $1, 2, ..., k + 1$ when cache size is k.
Note: LRU is a Label-based method.

Algo: choose a “random” old page to evict.

Theorem

Let denote the minimal eviction number as F^*. The expected number of evictions of RandomEvition algorithm is at most $2 \ln k F^*$.

Proof:
Consider phase i.

- Let A be the cache content at beginning. Sort A according to the request order in this phase, say $A = \{a_1, a_2, ...a_k\}$
- Let b_i be the requests that are not in A.
- When a_j is requested, and a_j is marked “Old”; (Reason: the case that a_j is “Recent” is omited since it causes no cache fault.)
- $\#OldPages = k - (j-1)$; (Reason: $a_1, a_2, ..., a_{j-1}$ are marked as “Recent”.)

- $\#OldPagesInCache = k-(j-1)-|b_i|$; (Reason: a request in b_i evicts an "Old" page out of cache.)

Pages in cache are in random. Thus, we have:

- $\Pr(a_j$ is in cache $) \geq \frac{k-(j-1)-|b_i|}{k-(j-1)}$;
- $\Pr(a_j$ is NOT in cache $) \leq 1 - \frac{k-(j-1)-|b_i|}{k-(j-1)}$ (cache missing);
-

$$\begin{aligned} E(\#missing) &\leq b_i + \sum_{j=1}^{k} \frac{|b_i|}{k-(j-1)} && (1) \\ &= |b_i| \log(k-(j-1)) && (2) \\ &= O(\log(k)) && (3) \end{aligned}$$

In phase i and $i+1$, there are $k+b_i$ distinct pages requested. Thus we can bound the number of faults as follows:

- $\#missing \geq b_i$

A randomize algo III

- $\#total - missing \geq \frac{1}{2}\sum_i |b_i|$
- ratio: $\leq 2\log(k)$.