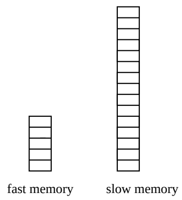
Demand Paging



We consider the **page fault model**, for measuring the performance of paging algorithms. In this model, we charge 1 for bringing a page into the fast memory, and we charge 0 for accessing a page which is already in the fast memory.

This is a very simplistic model, which does not address a number of factors, notably it does not address the fact that access to a fast memory page is in practice not free of charge.

Also, our algorithms are **demand paging**, meaning that unless there is a page fault, they never evict a page from the fast memory.

Claim: Any page replacement algorithm (online or offline) can be modified to be demand paging without increasing the overall cost on any request sequence.

We are going to prove this claim. Keep in mind that in a demand paging algorithm:

1. Each page fault causes exactly one eviction,
2. there are no evictions between misses.

So let A be any paging algorithm. We show how to modify it to make it a demand paging algorithm A', in such a way that on any input sequence A' incurs at most the cost (makes at most as many page moves from slow to fast memory) as A.

Suppose that A has a cache of size k. Define A' as follows: A' also has a cache of size k, and also k registers. A' runs a simulation of A, keeping in its k registers the page numbers of the pages that A would have in its cache. Based on the behaviour of A, A' makes decisions to evict pages.

Suppose page p is requested. If p is in the cache of A', do nothing. Otherwise, if a page fault occurs, A' behaves as follows:

- If A also has a “page fault” (that is, the number of p is *not* in the registers), and A “evicts” a page from register i to make room for p, then A' evicts a page from slot i in its cache, to make room for p.
- If A does not have a “page fault”, then the number of p must be in, say, register i. In that case, A' evicts the contents of slot i in its cache, and moves p in there.

Thus A' is a demand paging algorithm.

We now show that A' incurs at most the cost of A on any imput sequence; that is, A' has at most as many page faults as A.

To do this, we pair each page move of A' with a page move of A in a unique manner as follows: If A' and A both incur a page fault, then match the corresponding page moves.

Otherwise, if A already had the page in its cache, it must have moved it there before, so match that move with the current move of A'.

It is never the case that two different moves of A' are matched with a single move of A.

To see this, suppose that on some input sequence, we encounter for the first time the situation where two moves of A' are matched with the same move of A.

This can only happen in the following situation: page p is requested, A' incurs a page fault, it moves p into its cache, and we match this move with a past move of A, which has been matched already!

But this means that page p was already requested, and after it has been requested, it has been evicted from the cache of A' (otherwise, A' wouldn't have had a page fault).

Thus, A' evicted page p while A did not, so they were not in the same slot. But A' put (the first time) p in the same slot as A! Contradiction.

$\therefore$ We couldn't have matched a move twice.

Thus, we can match each move of A' with a move of A, in a one-to-one manner, and hence A' makes at most as many moves as A.

The idea is that A does not gain anything by moving a page into its cache pre-emptively (before the page is actually needed). A' waits for the request before taking the same action.

In the meantime (between the time that A moves in the page and the time that it is requested and A' brings it in), A' can only gain, because there are no requests for that page during that time, but there might be a request for the page that A evicted pre-emptively.

Note that in the simulation, A' only needs k extra registers, to keep track of the page numbers of the pages in the cache of A, so it is an *efficient* simulation.