Page Replacement

1.
Fifo: 9
LRU: 9
Optimal: 7
Random: Pretty much any answer bigger than or equal to 7

2. Here we were looking for an ARCHITECTURAL change that DID NOT INVOLVE ADDING MORE MEMORY ("assume that the total amount of physical memory is to remain fixed.")

Two good answers were:
Make the pages bigger.
Make the pages smaller.

A. Make the pages bigger

Reason: Bigger pages have the effect of prefetching the data from what would otherwise be the next page thereby potentially reducing the number of page faults.

For example, the system above is running with n=2 pages for a memory size of np, where p is the size of each page.

With np bytes though, I could also have a system with 1 page but twice as many bytes per page.

Let's look at what happens with optimal -- I'll use the same stream, but will now consider pages as being referenced in pairs
ab --> a reference to A or B, brings in the contiguous virtual range AB
cd --> same as for C or D

Then, our reference stream becomes:

ab ab cd ab ab cd ab cd ab cd

since there is only one page of memory, all of our replacement policies will work the same.

REF ab ab cd ab ab cd ab cd ab cd
ACTION F - F F - F F - F F

That is, seven faults. In contrast, only optimal got seven faults when we had smaller pages. The rest did worse.

Of course, for this approach to work, there needs to be good cross-page locality, otherwise, I'll end up doing worse.

Questions to think about:
why might I do worse without cross-page locality?

how might I realize larger pages using a pure software solution (eg, no changes to the system architecture or hardware).

how do the concepts of internal fraqmentation and external fragmentation apply to "bigger page" strategy?

B. Make the pages smaller

Reason: Smaller pages mean I can have a larger number of smaller things active at a time.

For example, let's say that a page is 1 byte. Then, instead of having n pages of p bytes each, I can have np pages, meaning that I can handle up to np unique addresses without taking a fault. With bigger pages, I can handle np addresses only if the references are constrained to fall on p pages.

So, for the stream above, let's say that not only am I referencing the named page, but each reference is to the same byte on the page.

Then, I have plenty of memory for my stream: ABCABDADBC -- and could run my whole program in only four faults.

Question to consider:

It looks like I started with exactly the wrong page size -- that is, in light of answers A and B, I picked a page size which is not as good as one that is either bigger or smaller. How could this be?

What new problem starts to become important if I make the pages too small? (think about translation). What can I do about it? Now, what problem arises?

Part 3: How might I achieve optimal by changing the OS API and application?

Optimal assumes perfect knowledge of the future without any information about the behavior of the program.

Suppose instead the program was able to INFORM the OS as to its future reference requirements. For example, suppose the program could provide the OS with its reference stream (or some approximation of it) before it generated the stream. Then, the OS can look "to the future" of the stream and replace the page that won't be used for the longest period of time. In the literature, this is sometimes called an INFORMED STRATEGY.

Question: What are the downsides of an INFORMED STRATEGY?

To Consider:

Going less to optimal and more to the goals of optimal, I would like to make the program run as fast as possible with a given memory size.

Recall that the effective access time for a memory reference is:

Ea = Ph * Th + (1-Ph) * Tm

(Prob hit * Time for a hit) + (Prob miss * Time for a miss)

To reduce Ea, I need to either increase Ph or decrease Tm.

For each of the strategies below, think about which terms change and why

	Ph	Th	1-Ph	Tm
OPTIMAL				
INFORMED STRATEGY				
SMALLER PAGES				
BIGGER PAGES				