CS162 Discussion #9: Demand Paging

Pages on Demand

Previously, we found out how to use virtual addresses as a way to abstract physical memory management away from user programs. However, while these schemes have allowed us to let programs act as if they have a chunk of contiguous memory when that might not be true in physical memory, they still limit the size of virtual address spaces. This means there must be enough physical memory to store all of the program's required pages at once for it to be able to run. It's easy to see that this can become extremely limiting, especially when we want to run a lot of programs at once.

Instead, we want to allow each program to act as if it has as much memory as it could ever want, regardless of whether or not we have enough room to store it all in physical memory at the moment – programs don't need all of their code at the same time, so we might as well not keep it all in memory if we're not going to be using it. In order to achieve this effect, we implement a mechanism called *demand paging*: we only put pages into memory when we need them, and the rest of the pages stay out on disk until we want to use them. When a page is requested that isn't currently in memory, this generates a *page fault* which causes the desired page to be loaded in from disk, possibly swapping out a page from physical memory back onto disk to make room for the new one. (Remember that we know a page isn't currently in memory when its page table entry isn't valid.) In this sense, physical memory becomes a cache for pages held on disk.

On a page fault, there are a number of steps the OS has to take:

1. Choose a free page in physical memory to load the page into if one exists, otherwise choose a page to replace.
2. If a page is getting replaced, check if it's dirty. If it is dirty, write it back to disk so the changes are saved.
3. Modify the replaced page's page table entry and TLB entry (if it has one) to be invalid.
4. Load the new page from disk into the chosen location in physical memory.
5. Update the page table entry for the new page.

When this is finished, the thread will continue from the faulting location as usual. This allows the program to act as if the page was there all along – the page fault causes the access to take a little longer, but the program otherwise doesn't have to know that any of this is happening.

As always, with step 1 above where we choose a page to replace, we have different options for replacement policies we could use in deciding which page to choose. Here are some policies we could use:

- *First In First Out (FIFO)*. Replace the page that has been in memory the longest.

- *Minimum (MIN)*. Replace the page that won't be used for the longest amount of time in the future.
- *RANDOM*. Randomly choose a page to replace.
- *Least Recently Used (LRU)*. Replace the page that hasn't been used for the longest amount of time.

Out of these policies, MIN would be the optimal choice, but in reality we can't really predict when each page will be used, so it's not really feasible. Because of locality within programs, however, a page that hasn't been used recently is less likely to be used in the near future. This makes LRU a decent (and more doable) approximation for MIN.

The Clock Algorithm

Implementing a completely correct LRU policy is possible, but not really necessary. We can get close enough to LRU and save resources by just choosing some page that hasn't been used recently, instead of the absolute oldest page. We do this by using the *clock algorithm*.

The clock algorithm views pages in memory as being arranged in a circle around a clock. The "hand" of the clock always points at the last page that was chosen for replacement. Then, on a page fault, the hand will tick, advancing until it finds a page that hasn't been used recently. How do we know if a page has been used recently? Remember that any access to a page causes the corresponding page table entry's used bit to be set. Therefore, each tick will check the used bit of the page the clock hand is on. If the bit is on, that means the page has been used recently so we should continue looking for a page. Before inspecting the next page, we set the used bit to 0 again to show that we've just checked this page. If the used bit is 0, that means that between the last time we looked at this page and now, the page hasn't been used, so we can replace it. We repeat this process until a page is chosen. (This means the hand can conceivably tick up to the number of physical pages on any page fault…but not any more than this, because when it reaches the place it started from, the used bit on that page must be 0.)

This basically gives us two chances to look at each page. We can generalize this to an *n-th chance algorithm*, where each page can be looked at n times before being chosen for eviction. In this scenario, each page has a counter associated with it. Each time the clock hand reaches a page with used bit 0, it increments the counter for that page. If the counter reaches n, the page is chosen to be replaced. Otherwise, the clock hand continues ticking. A page with used bit 1 gets its counter reset along with the used bit reset as in the clock algorithm above.

The result of using an n-th chance algorithm is that a higher n means we get a better LRU approximation, because it requires a page to have not been used for a larger number of clock iterations, with respect to other pages, in order to be replaced. However, the tradeoff for better LRU behavior is higher overhead. Therefore, we have to be careful what n we choose for our algorithm.