Attack to erase welcome’s code using memset (sets it all to 0’s)

- The 0’s erased the permissions and made it inaccessible, so we get pagefaults
- How do we fix it?
  - We can’t protect one process against another process without making it inaccessible to every other process except the kernel
  - Since otherwise we might prevent it from reaching its own code as well, we should fix it by setting a second level pagetable
  - Allocate space for hello’s new pagetable
  - Memset first level pagetable to 0’s (we’ll only ever need the first entry, and we set it to point to the top of the new second level table)
  - Take away user permissions then add them back for the first entry (which is the program’s own code)

Give each process its own physical page to make it seem like each process has control of the entire kernel

- You can map x in process 1 and x in process 2 and not have a conflict
- We do this with virtual addresses (VA’s)
  - Mapping from one virtual address to a physical address
    - %cr3 → control register
      - Holds the address of the current process’s L1 page table
    - The offset is the location in the physical page that the process wants to access
      - It’s the least significant 12 bits of the VA
      - You can’t change the offset when you translate
    - Identity mapping — mapping the virtual address to the same physical address
    - The Pagetable Level 1 Index (PTL1) is the most significant (top) 10 bits of the VA
    - The Pagetable Level 2 Index (PTL2) is the next most significant 10 bits of the VA
  - Example 1
    - Given:
      - Virtual Address: 0x00408050
      - Physical Address (PA): 0x1111050
      - %cr3: 0x10000000
      - [0, 0x03000000 | P | U | W | 0]
    - We get:
- **PTL1** = first 10 bits of 0x004 = 4
  - 0x004 = 0000 0000 0100 in decimal
- **Pagetable Level 2 Index (PTL2)** = last 10 bits of 0x408 = 8
  - 0x408 = 0100 0000 1000 in decimal
- **Offset** = 0x050
  - 0x80
    - 32 00000|P|W|U
  - If you wanted to map VA: 0x00408050 to PA: 0x00408052, you CAN'T
    - page table never messes with the offset → 050 has to go to 052
  - Kernel keeps a process descriptor → holds all the info of a given process
  - Takes all the state of process 1 and saves them all
  - When it goes back to P1, reloads all the registers and restarting the process
- **OS02: Three Processes**
  - Allocator
  - Recursive process
  - Simple fork
  - Systems calls vs. function calls
    - System call: put your arguments in registers, put return in eax, and then **call interrupt** which notifies kernel
  - Allocator
    - Allocates a page
  - **First Attack**
    - Can't call sys_page_alloc because it's not properly aligned
    - How should the kernel fix this?
      - Return an error when argument is not multiple of PAGESIZE
      - Check that the address that was passed into the system call is aligned, otherwise return -1
  - **Another attack**: Pass addresses in the Kernel itself
    - sys_page_alloc(char*) [addr]
    - Now we're messing with Kernel memory
    - Will be worse than a panic → OS continuously duplicates
    - Double fault handler → computer gets reset. This is a triple fault since the kernel trap causes a single fault, the handler check causes a double fault, and the double fault handler also fails, which makes it a triple fault
    - git checkout origin/v05
      - Not only check that it's aligned, but check that address is in user space and not kernel memory
  - **Recursive Program**
    - Calculate recursively from 0 to 999 n*(n+1)/2
- Helper function prevents compiler from inlining everything
- Problem: `sys_page_alloc` returned -1
  - going to run out of memory
  - The recursive functions create a huge chain of stack frames that can overflow
  - We pagefault because we try to step past the allocated memory
  - We can fix this by allocating a new page every time we pagefault
    - Trapping and handling allows OS to fix problems as they happen
    - The general pattern is having the process fail, fixing the problem, and restarting the process
    - This solution could fail because many different things can trigger a pagefault

- **Adventures in Systems Research**
  - A project that Eddie and his grad students have been working on — Web servers/data storage at a data center
  - If too many people are trying to access a data center through a web server, it can cause a malfunction in the data storage (even a fire, apparently)
  - Need to get data from somewhere else → data storage is a bottleneck
  - How do we fix this?
    - We could just increase the number of data centers, and be able to handle all the requests, but they’d still be slow
    - We can implement caching, buffered I/O
    - This reduces the load on the datastore (and prevents fires)
    - It also improves the read performance
      - There’s higher throughput and lower latency
        - Important because people are impatient (if Amazon took 10 seconds to load a page, customers would go elsewhere)
        - Latency = response time for a user to access a web page
    - When/What to cache?
      - When reads are repeated
Whole pages (Wikipedia) or fragments (Reddit)

- When reads are predictable
  - Computed results or aggregates & summaries

Twitter

- How can we use caching to help construct twitter timelines faster?
- You take the posts from the people you follow, sort them, and return them
- There’s more than 100 timeline checks per every new tweet (so reading dominates)
- 2M users, fixed workload

Caching Idea

- Cache as many recent posts and follows as possible
- The client collects the posts and assembles the timeline
- This comes with problems
  - It’s too slow to do on every timeline check, and it’d be useless if there are no updates to read
  - There is a linear relationship between runtime and percent of active users

Rethinking the Cache

- Is anything repeated? No. You don’t get the same tweet twice
- Is anything predictable? Yes! Tweets are always getting refreshed
  - You’re going to want to look at most recent tweets
- Is anything computed? Yes again!
- Is it expensive to compute? Yup.

A Good Idea

- Optimize for reads by doing more work on writes
- Keep an updates, cached timeline per user
  - On a write, append the tweet to the followers’ timelines
- For a timeline check, just read from the cache
  - In case of a cache miss, just construct the timeline as before
- Twitter already does this, and the linear relation is less sloped
  - It performs better for ~10% or greater active users, but worse for less

Research Idea

- Allow the cache to generate new data from the existing data
  - Want to be able to keep it up to date after it's been generated
  - On the first access, generate it anew
  - Maintain the derived data incrementally
- Optimize for predictable reads
  - Social media streams and aggregates of high-rate data

Pequod

- A key-value cache with additional features
  - Range scan primitive
- Cache joins
  - Relate input data (sources) with generated data (sinks)
  - Contact Eddie, Margo Seltzer, Stratos Idreos, or Steve Chong if you're interested!