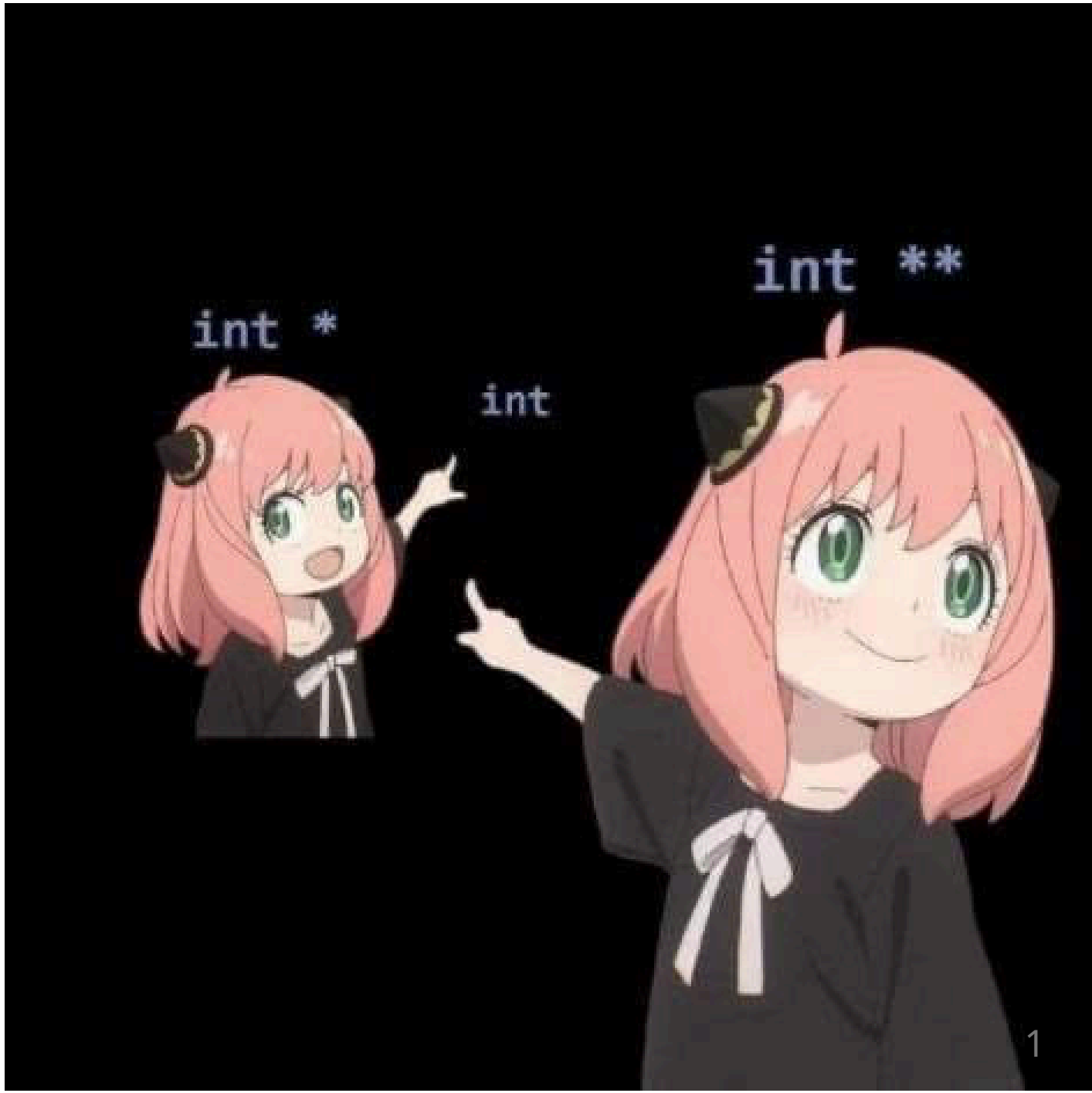
Lab 13: virtual memory notes

Stanford Winter 2026



1st rule of Virtual Memory Fight Club = it's trivial.

All we are doing is making a `fn:int->int` function.

- domain: virtual addr (for us: 32-bit number)
- range: physical addr (for us: 32-bit number) or exception
- use table (page) to build `fn:int->int` by recording each mapping
- roughly: hardware rewrites load/store/i-fetch to use translate:

```
ldr r0, [r1]     ===> ldr r0, [xlate(r1)]
str r2, [r4,#64] ===> str r2, [xlate(r4+64)]
```

Each time you get confused, always come back to this:

- just an integer function.
- 107 kids make these w/ linked lists, arrays, hash tables. not that hard.

Example: Simplistic translation

```
// xlate: any virtual byte to any physical byte
// input: 32-bit virtual addr
// output 32-bit physical addr or exception.
uint32_t xlate_store32(uint32_t va) {
  if(va&3)
      throw "alignment exception";
  let e = map[va];
  if(!e)
      throw "section fault"
  if((e->perm & WRITE_PERM) == 0)
      throw "protection fault"
  return e->pa;
}
```

Why do even this?

Think about how to run two blink progs.

1. They both jump to `0x8000` and run — doesn't work.
    - Could link both all at different addresses
        - Annoying; also doesn't work if others running.
    - Can copy in and out (we will do this next week :)
        - Simple but slow + clumsy.
    - VM: let them both use `0x8000`
        - Map virtual `0x8000` to different physical addresses.
        - Switch page table pointer when switching b/n them.
2. They can smash each other's memory.
    - Use VM to stop blink-1 from read/write blink-2.

So why 100+ pages of crazy neologisms/jargon?

Speed. Speed. Speed.

- That is it. Translation never fast enough.
- Every load. Every store. Every instruction fetch.

ARMv6 uses fairly common methods:

- Use hardware to speed up translation. Downside: complex b/c hardware defines page table layout.
- Reduce table lookups by caching translations (TLB)
- ASID so don't have to flush TLB entries when switch address space
- Quantize memory into fixed-sized chunks (4k, 64k, 1MB, 16MB) so one translation goes farther.

Universal: there has to be a table.

Whenever OS busies itself mapping X to Y, then:

- You know there is an O(n) table of some kind.
- Conceptually probably simple (yea), but has a bunch of details/jargon (sorry).

Quick OS proof: N arbitrary mappings require a O(n) table:

1. If want to support purely arbitrary `X => Y` mappings
2. Then we must be able to express purely random mappings;
3. Definitionally: random = not compressible;
4. Therefore: must record everything;
5. QED: table is O(n) w.r.t. number of translations.

Map location X to Y: old problem, old tensions

- Full flexibility? Need a table (map)
  - You are at Elm and need to get to Pine.
  - If no partial order (e.g., sort), then need a table (map) to look up.
  - Pro: flexible. Can stick anything, anywhere
  - Con: Map takes space. Map lookup slow.
- No flexibility? throw out map, use arithmetic.
  - You are at 1st street and want to get to 10th.
  - Cartesian grid? Walk north.
  - Pro: fastest lookup = no lookup, smallest map = no map.
  - Con: Inflexible. If don't allow 1/2 (or 1/10th) can't insert b/n.

Virtual memory is a mixture of map+arithmetic

- Map every byte to every byte? Would need huge table.
  - Usual hack: quantize into blocks (pages, super-pages).
  - Any page can map to any page (need table), but within page is identical (no table).
- We'll use 1MB sections today:

- Usual page size tradeoff = larger the page then:
  - pro: fewer translations needed, so TLB goes farther, table smaller.
  - but: internal fragmentation worse. (Need 1 byte? Get 1 page).

Today: 1MB section translation

- Section: 1MB-aligned 1MB chunk. (recall `1024*1024=2^10*2^10=2^20`).
- Section offset = low 20-bits of 32-bit address.
- Section number = upper 12-bits of 32-bit address.

```
// map virtual section to physical section
uint32_t xlate_write32(uint32_t va) {
  if(va&3)
      throw "alignment exception";
  uint32_t offset = va & ((1u<<20)-1u); // get lo 20 bits
  uint32_t sec    = va >> 20;           // get hi 12 bits
  let e = map[sec];
  if(!e)
      throw "section fault"
  if((e->perm & WRITE_PERM) == 0)
      throw "protection fault"
  // offset is the same
  return e->sec << 20 | offset;
}
```

- VM bugs = the hardest you'll ever hit.
  - Much worse than memory corruption.
  - Wrong translation = map correct code/its data to garbage.
  - Result: correct code does "impossible" things.
- Absolutely must do epsilon induction:
  i. Start with working system.
  ii. Take smallest step possible. (Even smaller than that.)
  iii. Check.
  iv. Step small again.

- "It's correct b/c it worked when I ran it"
  - LOL. OMFG: *NO*, *NO*, *NO*.
- Testing = Low-recall.
  - VM bugs often from coherency/reuse problems
  - "It worked" != correct.
  - Did you run all combinations of:
    - processes?
    - caching?
    - memory reuse?
    - ASIDs reuse?
- Absolutely have to obey docs.

VM error detection

- Once we are translating, we can do array bounds checking "for free"
  - Does the input virtual address exist?
  - Does it have sufficient permissions?
- Nice compared to software bounds checking:
  - can't jump over it.
  - happens transparently
  - much faster (can happen in parallel).
- Bad:
  - very coarse: e.g., page level, not object level.
  - doesn't track "referent" (what you were supposed to point to).

```
p = malloc(sizeof *p);
p += 100000;  // if now points to another legal object
*p = 1;       // then error not detected.
```