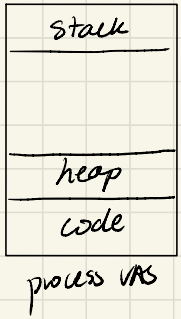
4/29/24 Eviction

```
struct vspace {
  struct vregion regions[NREGIONS]; // the regions for a process' virtual space
  pml4e_t* pgtbl;                   // process' page table
};
```



process VAS

```
struct vregion {
  enum vr_direction dir;  // direction of growth
  uint64_t va_base;       // base of the region
  uint64_t size;          // size of region in bytes
  struct vpi_page *pages;  // pointer to array of page_infos
};
```

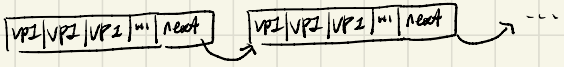
```
struct vpi_page {
  struct vpage_info infos[VPIPPAGE];  // info struct for the given page
  struct vpi_page *next;              // the next page
};
```

```
struct vpage_info {
  short used;      // whether the page is in use
  uint64_t ppn;    // physical page number
  short present;   // whether the page is in physical memory
  short writable; // does the page have write permissions
  // user defined fields

};
```

per page metadata

linked list of vpage info array



Software page table walk

```
pte_t *
walkpml4(pml4e_t *pml4, const void *va, int alloc)
{
  pml4e_t *pml4e;
  pdpte_t *pdpt, *pdpte;
  pde_t *pgdir, *pde;
  pte_t *pgtab;

  pml4e = &pml4[PML4_INDEX(va)];

  if (*pml4e & PTE_P) {
    pdpt = (pdpte_t*)P2V(PDPT_ADDR(*pml4e));
  } else {
    if(!alloc || (pdpt = (pdpte_t*)kalloc()) == 0)
      return 0;
    memset(pdpt, 0, PGSIZE);
    *pml4e = V2P(pdpt) | PTE_P | PTE_W | PTE_U;
  }

  pdpte = &pdpt[PDPT_INDEX(va)];

  pdpte = &pdpt[PDPT_INDEX(va)];

  if (*pdpte & PTE_P) {
    pgdir = (pde_t*)P2V(PDE_ADDR(*pdpte));
  } else {
    if(!alloc || (pgdir = (pde_t*)kalloc()) == 0)
      return 0;
    memset(pgdir, 0, PGSIZE);
    *pdpte = V2P(pgdir) | PTE_P | PTE_W | PTE_U;
  }

  pde = &pgdir[PD_INDEX(va)];

  if (*pde & PTE_P) {
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    memset(pgtab, 0, PGSIZE);
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }

  return &pgtab[PT_INDEX(va)];
}
```

indexing into the top level PT

translate to kernel virtual addr

stores physical address

pointer to pte (last level entry)
→ contains frame # (if mapped)

x86_64 vm.c

Page Faults

→ valid page faults (demand paging, mmap, cow) → handle & retries instr.

→ invalid page faults (NULL, seg fault, permission violation) → terminates.

To handle a valid page fault ⇒ allocate a new frame

→ what if there's no frame available?

→ blocks until a frame is freed? might block for arbitrary amount of time, could cause deadlock

→ make a frame available by writing it to storage (swap)

large capacity, slower but the write will finish

→ select a process to kill

→ general purpose OS tries to avoid it

→ will reclaim page cache, max out swap before killing a process

look for one that uses large amt of memory & is young.

→ happens more on mobile OSes

device: (crash amount/total amount/percentage of total)

- iPhone X: 1392/2785/50% (iOS 11.2.1)
- iPhone XS: 2040/3754/54% (iOS 12.1)
- iPhone XS Max: 2039/3735/55% (iOS 12.1)
- iPhone XR: 1792/2813/63% (iOS 12.1)
- iPhone 11: 2068/3844/54% (iOS 13.1.3)
- iPhone 11 Pro Max: 2067/3740/55% (iOS 13.2.3)
- iPhone 12 Pro: 3054/5703/54% (iOS 16.1)

iOS developers trying to figure out memory budget for their apps to avoid being killed

3 iPhone 5 crashes at ±645 MB. – asp_net Dec 15, 2013 at 21:03

iPhone 5S crashes at ±646 MB pretty reliably here. – eAi Oct 3, 2014 at 13:30

iPhone 4S crashes at ±286MB (286MB/512MB/56%). – Xaree Lee May 29, 2015 at 21:50

iPhone 4S doesn't crash until it reaches ±363 MB for me. (iOS 5.1.1) – Soeholm Jun 3, 2015 at 16:53

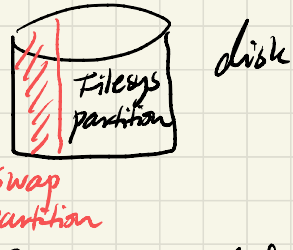
2 Awesome that this list has been created and maintained. In my experience, I've had to keep memory much lower to be safe, maybe 20% of what's shown here. Device-to-device differences are also highly variable. – user1021430 Aug 10, 2015 at 19:24

1 Just ran this on a 12.9 iPad Pro. Memory warning at 2451MB, crash at 3064MB, total 3981MB. – lock Jul 15, 2016 at 13:22

iPhone 6s+: 1392MB/2048MB/ 68% (iOS 10.2.1); iPhone 7+: 2040MB/3072MB/66% (iOS 10.2.1) – Slyv Feb 13, 2017 at 15:39

Eviction Mechanism

→ write a frame to somewhere on disk

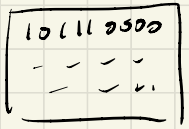


any free space?

→ swap has different life-time from a normal filesys partition
(no longer needed after a crash)

→ OS manages the swap partition (allocation & deallocation)

→ uses a bit map to track usage info



1 → in use each bit represents a block's usage info.

0 → free

Eviction Steps (after selecting a frame to evict)

1). allocate swap space from the bitmap

2). remove old page to frame mapping, TLB shootdown

→ ensures no modification can be done on the evicted page.

→ possibly mapped to multiple pages, must remove all mappings!

→ how to find the pages? per frame metadata (ex: coremap entry)

3). write the frame to allocated swap block

4). track the swap loc of the evicted page somewhere.

→ reuse pte w/ the present bit as 0.

What happens if the evicted page is accessed here?

Page fault! must synchronize it properly!

5). reallocate the frame to the faulting page.

→ zero out or overwrite old data

Eviction Policy

→ How to choose an eviction candidate?

→ kernel frames? pinned in physical memory

→ shared libraries?

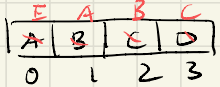
→ only frames within the faulting process? → good for performance isolation, but what should be the limit? (local page replacement)

→ any frame in the system? → flexible allocation (global page replacement)

→ FIFO: a queue of frames based on allocation order, evict frame at the front of the queue, easy to implement

→ bad access pattern?

access page A B C D E on repeat over 4 frames



→ Belady's Anomaly: increasing # of frames can lead to more page faults using FIFO