```c
/* Destroys page directory PD, freeing all the pages it
   references. */
void
pagedir_destroy (uint32_t *pd)
{
  uint32_t *pde;

  if (pd == NULL)
    return;

  ASSERT (pd != init_page_dir);
  for (pde = pd; pde < pd + pd_no (PHYS_BASE); pde++)
    if (*pde & PTE_P)
      {
        uint32_t *pt = pde_get_pt (*pde);
        uint32_t *pte;

        for (pte = pt; pte < pt + PGSIZE / sizeof *pte; pte++)
          if (*pte & PTE_P)
            palloc_free_page (pte_get_page (*pte));
        palloc_free_page (pt);
      }
  palloc_free_page (pd);
}
```

user VA에 대응되는 PA를 look up한다. ⇒ kernel VA를 리턴함.
↳ 여기에 대응되는 ↳ 대응되는게 없으면 NULL 리턴

```c
/* Looks up the "physical address" that corresponds to user virtual
   address UADDR in PD.  Returns the kernel virtual address
   corresponding to that physical address, or a null pointer if
   UADDR is unmapped. */
void *
pagedir_get_page (uint32_t *pd, const void *uaddr)
{
  uint32_t *pte;

  ASSERT (is_user_vaddr (uaddr));

  pte = lookup_page (pd, uaddr, false);
  if (pte != NULL && (*pte & PTE_P) != 0)
    return pte_get_page (*pte) + pg_ofs (uaddr);
  else
    return NULL;
}
```

P.43 every user virtual page is aligned ~
이게 어떻게 작용하는.

뭐지?

```c
/* Returns the block device fulfilling the given ROLE, or a null
   pointer if no block device has been assigned that role. */
struct block *
block_get_role (enum block_type role)
{
  ASSERT (role < BLOCK_ROLE_CNT);
  return block_by_role[role];
}
```

```c
/* The block block assigned to each Pintos role. */
static struct block *block_by_role[BLOCK_ROLE_CNT];
```

```c
/* Type of a block device. */
enum block_type
  {
    /* Block device types that play a role in Pintos. */
    BLOCK_KERNEL,               /* Pintos OS kernel. */
    BLOCK_FILESYS,              /* File system. */
    BLOCK_SCRATCH,              /* Scratch. */
    BLOCK_SWAP,                 /* Swap. */
    BLOCK_ROLE_CNT,

    /* Other kinds of block devices that Pintos may see but does
       not interact with. */
    BLOCK_RAW = BLOCK_ROLE_CNT,  /* "Raw" device with unidentified contents. */
    BLOCK_FOREIGN,              /* Owned by non-Pintos operating system. */
    BLOCK_CNT                   /* Number of Pintos block types. */
  };
```

```c
/* A block device. */
struct block
  {
    struct list_elem list_elem;         /* Element in all_blocks. */

    char name[16];                      /* Block device name. */
    enum block_type type;                /* Type of block device. */
    block_sector_t size;                 /* Size in sectors. */

    const struct block_operations *ops;  /* Driver operations. */
    void *aux;                          /* Extra data owned by driver. */

    unsigned long long read_cnt;        /* Number of sectors read. */
    unsigned long long write_cnt;       /* Number of sectors written. */
  };
```

uint_32t

```c
/* Returns the number of sectors in BLOCK. */
block_sector_t
block_size (struct block *block)
{
  return block->size;
}
```

```c
/* Write sector SECTOR to BLOCK from BUFFER, which must contain
   BLOCK_SECTOR_SIZE bytes.  Returns after the block device has
   acknowledged receiving the data.
   Internally synchronizes accesses to block devices, so external
   per-block device locking is unneeded. */
void
block_write (struct block *block, block_sector_t sector, const void *buffer)
{
  check_sector (block, sector);
  ASSERT (block->type != BLOCK_FOREIGN);
  block->ops->write (block->aux, sector, buffer);
  block->write_cnt++;
}
```

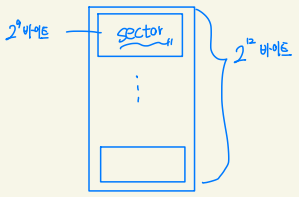
```c
/* Reads sector SECTOR from BLOCK into BUFFER, which must
   have room for BLOCK_SECTOR_SIZE bytes.
   Internally synchronizes accesses to block devices, so external
   per-block device locking is unneeded. */
void
block_read (struct block *block, block_sector_t sector, void *buffer)
{
  check_sector (block, sector);
  block->ops->read (block->aux, sector, buffer);
  block->read_cnt++;
}
```

```c
/* Creation and destruction. */

/* Creates and returns a pointer to a newly allocated bitmap with room for
   BIT_CNT (or more) bits.  Returns a null pointer if memory allocation fails.
   The caller is responsible for freeing the bitmap, with bitmap_destroy(),
   when it is no longer needed. */
struct bitmap *
bitmap_create (size_t bit_cnt)
{
  struct bitmap *b = malloc (sizeof *b);
  if (b != NULL)
    {
      b->bit_cnt = bit_cnt;
      b->bits = malloc (byte_cnt (bit_cnt));
      if (b->bits != NULL || bit_cnt == 0)
        {
          bitmap_set_all (b, false);
          return b;
        }
      free (b);
    }
  return NULL;
}
```

unsigned long

```c
/* Sets all bits in B to VALUE. */
void
bitmap_set_all (struct bitmap *b, bool value)
{
  ASSERT (b != NULL);

  bitmap_set_multiple (b, 0, bitmap_size (b), value);
}
```

page

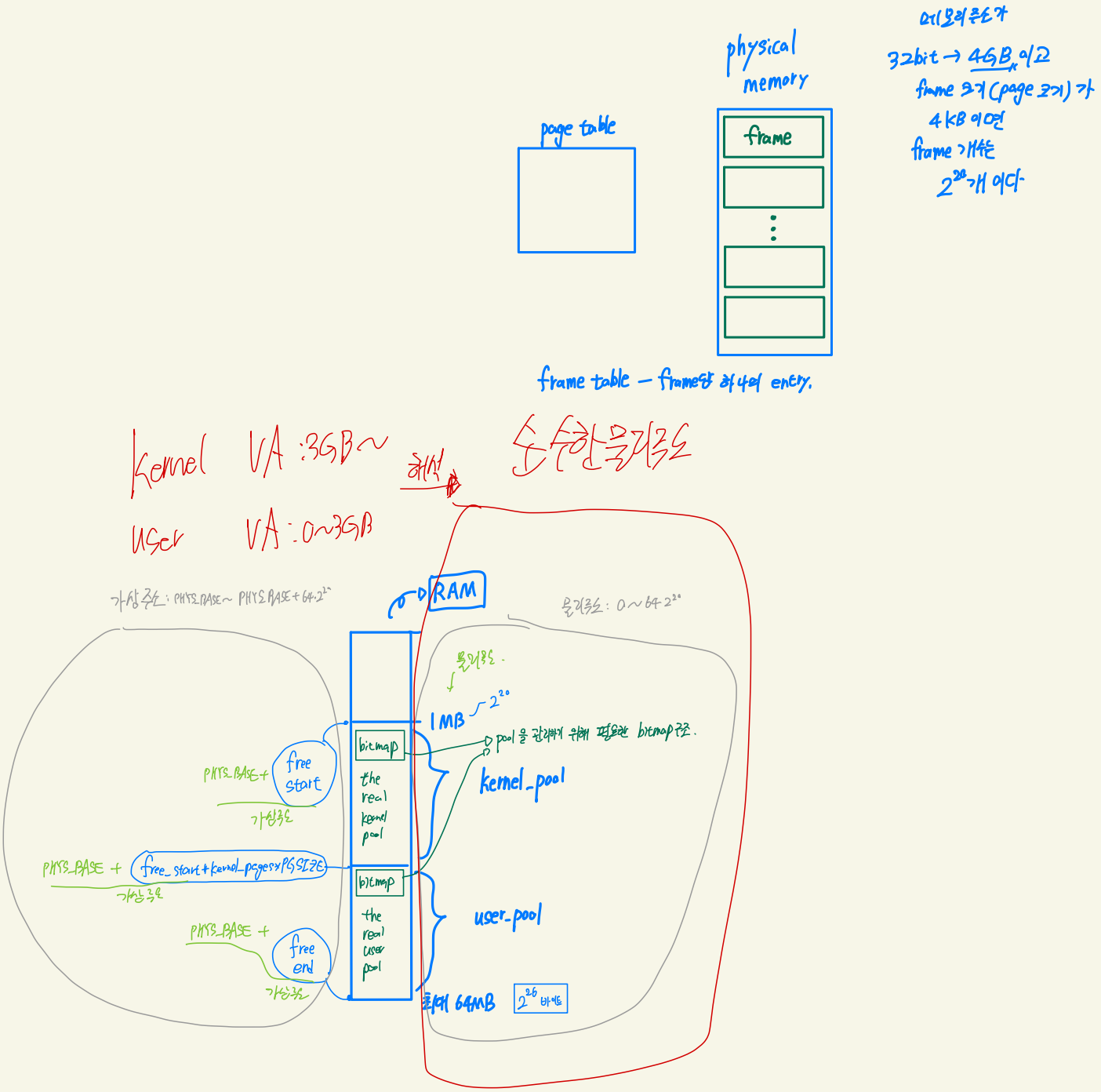


페이지 단위로 할당됨

```c
/* Finds the first group of CNT consecutive bits in B at or after
   START that are all set to VALUE, flips them all to !VALUE,
   and returns the index of the first bit in the group.
   If there is no such group, returns BITMAP_ERROR.
   If CNT is zero, returns 0.
   Bits are set atomically, but testing bits is not atomic with
   setting them. */
size_t
bitmap_scan_and_flip (struct bitmap *b, size_t start, size_t cnt, bool value)
{
  size_t idx = bitmap_scan (b, start, cnt, value);
  if (idx != BITMAP_ERROR)
    bitmap_set_multiple (b, idx, cnt, !value);
  return idx;
}
```

evict시킨것 다할당됨...

```
C init.c src/threads 2
    pd = init_page_dir = palloc_get_page (PAL_ASSERT | PAL_ZERO);
    pt = palloc_get_page (PAL_ASSERT | PAL_ZERO);
C malloc.c src/threads 1
    a = palloc_get_page (0);
C palloc.c src/threads 1
    palloc_get_page (enum palloc_flags flags)
C palloc.h src/threads 1
    void *palloc_get_page (enum palloc_flags);
C thread.c src/threads 1
    t = palloc_get_page (PAL_ZERO);   ~ thread_create
C thread.h src/threads 1
    allocation with malloc() or palloc_get_page()
C pagedir.c src/userprog 3
    uint32_t *pd = palloc_get_page (0);   → page directory만듦.
    pt = palloc_get_page (PAL_ZERO);   → page table 할때의 예외.
    with palloc_get_page().
C process.c src/userprog 7 3
    fn_copy = palloc_get_page (0); //allocate in kernel pool   → process_execute
    kpage = palloc_get_page (PAL_USER | PAL_ZERO);   → stack 페이지 frame에서?
    with palloc_get_page().
C tss.c src/userprog 1
    tss = palloc_get_page (PAL_ASSERT | PAL_ZERO);   tss할당 받을 때
C frame.c src/vm 1 2
    kernel_VA_for_new_frame = palloc_get_page(flags);
    kernel_VA_for_new_frame = palloc_get_page(flags);   ) 새로 추가할것.
```



메모리주소가
32bit → 4GB 이고
frame 크기(page 크기)가
4kB 이면
frame 개수는
2^{20}개 이다.

palloc.c ~ pool을 쓴다. ~ bitmap을 쓴다.

```c
/* Page allocator.  Hands out memory in page-size (or
   page-multiple) chunks.  See malloc.h for an allocator that
   hands out smaller chunks.

   System memory is divided into two "pools" called the kernel
   and user pools.  The user pool is for user (virtual) memory
   pages, the kernel pool for everything else.  The idea here is
   that the kernel needs to have memory for its own operations
   even if user processes are swapping like mad.

   By default, half of system RAM is given to the kernel pool and
   half to the user pool.  That should be huge overkill for the
   kernel pool, but that's just fine for demonstration purposes. */

/* A memory pool. */
struct pool
  {
    struct lock lock;                   /* Mutual exclusion. */
    struct bitmap *used_map;            /* Bitmap of free pages. */
    uint8_t *base;                      /* Base of pool. */
  };

/* Two pools: one for kernel data, one for user pages. */
static struct pool kernel_pool, user_pool;
```

malloc.h에선 page보다 작은단위로 할당.

(user ⟷ kernel) pool

ㅋㅋ

system의 RAM을 반씩 먹는다.

→ page단위 관리. ∴ bitmap!

1. 핀토스 RAM 크기가 최대 64MB이다. (물리주소가 최대 64-?? MB인걸까?)

2. PHYS_BASE 3GB ... 3GB] → 그럼 (ㅁ)에서 여유공간 있는거 아닌가?

disk가 붙으면 또 얘기가 다른가? / frame은 RAM에 물리주소로 할당되어 있는다.

```c
/* Initializes the page allocator.  At most USER_PAGE_LIMIT
   pages are put into the user pool. */
void
palloc_init (size_t user_page_limit)
{
  /* Free memory starts at 1 MB and runs to the end of RAM. */
  uint8_t *free_start = ptov (1024 * 1024);
  uint8_t *free_end = ptov (init_ram_pages * PGSIZE);
  size_t free_pages = (free_end - free_start) / PGSIZE;
  size_t user_pages = free_pages / 2;
  size_t kernel_pages;
  if (user_pages > user_page_limit)
    user_pages = user_page_limit;
  kernel_pages = free_pages - user_pages;

  /* Give half of memory to kernel, half to user. */
  init_pool (&kernel_pool, free_start, kernel_pages, "kernel pool");
  init_pool (&user_pool, free_start + kernel_pages * PGSIZE,
             user_pages, "user pool");
}
```

kernel virtual address 단위로

→ free page 개수.

½ 반씩 갈라먹기

가상 / 가상 / 2^{12}

pool* / void* / size_t

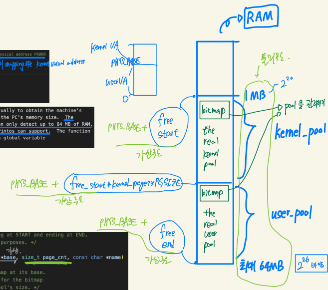
bitmap의 base로 저장된곳에 "가상주소"가 들어있다. ∴ kernel

```c
/* Returns kernel virtual address at which physical address PADDR
   is mapped. */
static inline void *
ptov (uintptr_t paddr)
{
  ASSERT ((void *) paddr < PHYS_BASE);

  return (void *) (paddr + PHYS_BASE);
}
```

이 mapping 되는 kernel virtual address

```c
/* The startup code's first task is actually to obtain the machine's
   memory size, by asking the BIOS for the PC's memory size.  The
   simplest BIOS function to do this can only detect up to 64 MB of RAM,
   so that's the practical limit that Pintos can support.  The function
   stores the memory size, in pages, in global variable
   @code{init_ram_pages}.
```

kernel VA / PHYS_BASE / user VA



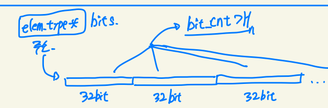

```c
/* Initializes pool P as starting at START and ending at END,
   naming it NAME for debugging purposes. */
static void
init_pool (struct pool *p, void *base, size_t page_cnt, const char *name)
{
  /* We'll put the pool's used_map at its base.
     Calculate the space needed for the bitmap
     and subtract it from the pool's size. */
  size_t bm_pages = DIV_ROUND_UP (bitmap_buf_size (page_cnt), PGSIZE);
  if (bm_pages > page_cnt)
    PANIC ("Not enough memory in %s for bitmap.", name);
  page_cnt -= bm_pages;

  printf ("%zu pages available in %s.\n", page_cnt, name);

  /* Initialize the pool. */
  lock_init (&p->lock);
  p->used_map = bitmap_create_in_buf (page_cnt, base, bm_pages * PGSIZE);
  p->base = base + bm_pages * PGSIZE;
}
```

가상.

앞에 bitmap 구조체들에 못쓰게 되었다네요.

```c
/* Creates and returns a bitmap with BIT_CNT bits in the
   BLOCK_SIZE bytes of storage preallocated at BLOCK.
   BLOCK_SIZE must be at least bitmap_needed_bytes(BIT_CNT). */
struct bitmap *
bitmap_create_in_buf (size_t bit_cnt, void *block, size_t block_size UNUSED)
{
  struct bitmap *b = block;

  ASSERT (block_size >= bitmap_buf_size (bit_cnt));

  b->bit_cnt = bit_cnt;
  b->bits = (elem_type *) (b + 1);
  bitmap_set_all (b, false);
  return b;
}
```

```c
/* From the outside, a bitmap is an array of bits.  From the
   inside, it's an array of elem_type (defined above) that
   simulates an array of bits. */
struct bitmap
  {
    size_t bit_cnt;     /* Number of bits. */
    elem_type *bits;    /* Elements that represent bits. */
  };
```




```c
/* Element type.

   This must be an unsigned integer type at least as wide as int.

   Each bit represents one bit in the bitmap.
   If bit 0 in an element represents bit K in the bitmap,
   then bit 1 in the element represents bit K+1 in the bitmap,
   and so on. */
typedef unsigned long elem_type;
```

32bit

```c
/* Creation and destruction. */

/* Creates and returns a pointer to a newly allocated bitmap with room for
   BIT_CNT (or more) bits.  Returns a null pointer if memory allocation fails.
   The caller is responsible for freeing the bitmap, with bitmap_destroy(),
   when it is no longer needed. */
struct bitmap *
bitmap_create (size_t bit_cnt)
{
  struct bitmap *b = malloc (sizeof *b);
  if (b != NULL)
    {
      b->bit_cnt = bit_cnt;
      b->bits = malloc (byte_cnt (bit_cnt));
      if (b->bits != NULL || bit_cnt == 0)
        {
          bitmap_set_all (b, false);
          return b;
        }
      free (b);
    }
  return NULL;
}
```

malloc 까진 내려가지 말자.
→ 이것도 결국 OS가 관리...

아래를 보니까... 단계가 지원되네!

```c
/* Obtains a single free page and returns its kernel virtual
   address.
   If PAL_USER is set, the page is obtained from the user pool,
   otherwise from the kernel pool.  If PAL_ZERO is set in FLAGS,
   then the page is filled with zeros.  If no pages are
   available, returns a null pointer, unless PAL_ASSERT is set in
   FLAGS, in which case the kernel panics. */
void *
palloc_get_page (enum palloc_flags flags)
{
  return palloc_get_multiple (flags, 1);
}
```

→ 인자가 PANIC.

```c
/* Obtains and returns a group of PAGE_CNT contiguous free pages.
   If PAL_USER is set, the pages are obtained from the user pool,
   otherwise from the kernel pool.  If PAL_ZERO is set in FLAGS,
   then the pages are filled with zeros.  If too few pages are
   available, returns a null pointer, unless PAL_ASSERT is set in
   FLAGS, in which case the kernel panics. */
void *
palloc_get_multiple (enum palloc_flags flags, size_t page_cnt)
{
  struct pool *pool = flags & PAL_USER ? &user_pool : &kernel_pool;
  void *pages;
  size_t page_idx;

  if (page_cnt == 0)
    return NULL;

  lock_acquire (&pool->lock);
  page_idx = bitmap_scan_and_flip (pool->used_map, 0, page_cnt, false);
  lock_release (&pool->lock);

  if (page_idx != BITMAP_ERROR)
    pages = pool->base + PGSIZE * page_idx;
  else
    pages = NULL;

  if (pages != NULL)
    {
      if (flags & PAL_ZERO)
        memset (pages, 0, PGSIZE * page_cnt);
    }
  else
    {
      if (flags & PAL_ASSERT)
        PANIC ("palloc_get: out of pages");
    }

  return pages;
}
```

(유저풀 / 커널풀) 고르고

page_cnt 개수만큼 할당 받음 "표시!"

가상 pages / 가상 / → 4KB

페이지의 가상주소가 리턴되네

"obtain"

pages(가상주소) 부터 2^{12} * page_cnt "바이트"만큼 memset 0으로 만든다.

→ 메모리 관련 함수...

"표시한 위치에다가 memset" → RAM 상의 위치

→ RAM상의 주소. (PHYS_BASE 만큼 더해진)

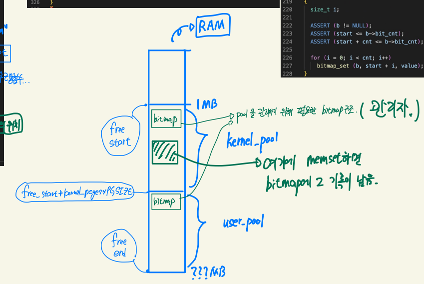
bitmap 사용.

오...연속된

```c
/* Finds the first group of CNT consecutive bits in B at or after
   START that are all set to VALUE, flips them all to !VALUE,
   and returns the index of the first bit in the group.
   If there is no such group, returns BITMAP_ERROR.
   If CNT is zero, returns 0.
   Bits are set atomically, but testing bits is not atomic with
   setting them. */
size_t
bitmap_scan_and_flip (struct bitmap *b, size_t start, size_t cnt, bool value)
{
  size_t idx = bitmap_scan (b, start, cnt, value);
  if (idx != BITMAP_ERROR)
    bitmap_set_multiple (b, idx, cnt, !value);
  return idx;
}
```

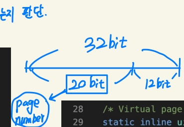
```c
/* Finds and returns the starting index of the first group of CNT
   consecutive bits in B at or after START that are all set to
   VALUE.
   If there is no such group, returns BITMAP_ERROR. */
size_t
bitmap_scan (const struct bitmap *b, size_t start, size_t cnt, bool value)
{
  ASSERT (b != NULL);
  ASSERT (start <= b->bit_cnt);

  if (cnt <= b->bit_cnt)
    {
      size_t last = b->bit_cnt - cnt;
      size_t i;
      for (i = start; i <= last; i++)
        if (!bitmap_contains (b, i, cnt, !value))
          return i;
    }
  return BITMAP_ERROR;
}
```

```c
/* Sets the CNT bits starting at START in B to VALUE. */
void
bitmap_set_multiple (struct bitmap *b, size_t start, size_t cnt, bool value)
{
  size_t i;

  ASSERT (b != NULL);
  ASSERT (start <= b->bit_cnt);
  ASSERT (start + cnt <= b->bit_cnt);

  for (i = 0; i < cnt; i++)
    bitmap_set (b, start + i, value);
}
```



user pool 이든 kernel pool 이든 각각의 pool의 범위에 있는지 판단.

```c
/* Returns true if PAGE was allocated from POOL,
   false otherwise. */
static bool
page_from_pool (const struct pool *pool, void *page)
{
  size_t page_no = pg_no (page);
  size_t start_page = pg_no (pool->base);
  size_t end_page = start_page + bitmap_size (pool->used_map);

  return page_no >= start_page && page_no < end_page;
}
```




```c
/* Virtual page number. */
static inline uintptr_t pg_no (const void *va) {
  return (uintptr_t) va >> PGBITS;
}
```

12

```c
/* Bitmap size. */

/* Returns the number of bits in B. */
size_t
bitmap_size (const struct bitmap *b)
{
  return b->bit_cnt;
}
```

1 페이지를 1비트로 다룸

```c
/* Frees the PAGE_CNT pages starting at PAGES. */
void
palloc_free_multiple (void *pages, size_t page_cnt)
{
  struct pool *pool;
  size_t page_idx;

  ASSERT (pg_ofs (pages) == 0);
  if (pages == NULL || page_cnt == 0)
    return;

  if (page_from_pool (&kernel_pool, pages))
    pool = &kernel_pool;
  else if (page_from_pool (&user_pool, pages))
    pool = &user_pool;
  else
    NOT_REACHED ();

  page_idx = pg_no (pages) - pg_no (pool->base);

#ifndef NDEBUG
  memset (pages, 0xcc, PGSIZE * page_cnt);
#endif

  ASSERT (bitmap_all (pool->used_map, page_idx, page_cnt));
  bitmap_set_multiple (pool->used_map, page_idx, page_cnt, false);
}

/* Frees the page at PAGE. */
void
palloc_free_page (void *page)
{
  palloc_free_multiple (page, 1);
}
```

kernel virtual address 겠지?

pool을 고른다.

→ page의 index 고른다.

→ "초기화" using memset.

→ 초기화했다고 "표시"

```c
/* Base address of the 1:1 physical-to-virtual mapping.  Physical
   memory is mapped starting at this virtual address.  Thus,
   physical address 0 is accessible at PHYS_BASE, physical
   address address 0x1234 at (uint8_t *) PHYS_BASE + 0x1234, and
   so on.

   This address also marks the end of user programs' address
   space.  Up to this point in memory, user programs are allowed
   to map whatever they like.  At this point and above, the
   virtual address space belongs to the kernel. */
#define PHYS_BASE ((void *) LOADER_PHYS_BASE)

/* Returns true if VADDR is a user virtual address. */
static inline bool
is_user_vaddr (const void *vaddr)
{
  return vaddr < PHYS_BASE;
}

/* Returns true if VADDR is a kernel virtual address. */
static inline bool
is_kernel_vaddr (const void *vaddr)
{
  return vaddr >= PHYS_BASE;
}

/* Returns kernel virtual address at which physical address PADDR
   is mapped. */
static inline void *
ptov (uintptr_t paddr)
{
  ASSERT ((void *) paddr < PHYS_BASE);

  return (void *) (paddr + PHYS_BASE);
}

/* Returns physical address at which kernel virtual address VADDR
   is mapped. */
static inline uintptr_t
vtop (const void *vaddr)
{
  ASSERT (is_kernel_vaddr (vaddr));

  return (uintptr_t) vaddr - (uintptr_t) PHYS_BASE;
}
```

ok.

ok.

? 물리주소는 PHYS_BASE가 최대임?

p.a. 0 는 PHYS_BASE로 접근

p.a. 0x1234는 PHYS_BASE+0x1234로 접근

PHYS_BASE는 user address의 끝점이다.

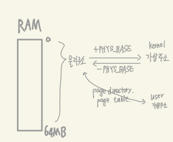
0~PHYS_BASE 이내는 user program 가상주소

PHYS_BASE~이후 V.a는 kernel관리.

```c
/* Kernel virtual address at which all physical memory is mapped.
   Must be aligned on a 4 MB boundary. */
#define LOADER_PHYS_BASE 0xc0000000     /* 3 GB. */
```

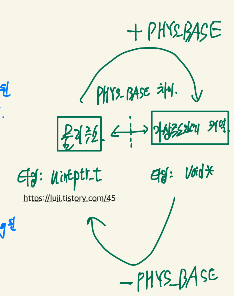
virtual address — user : 0~PHYS_BASE / kernel : PHYS_BASE ~ 4GB(?)

physical address



physical address가 mapping된 kernel virtual address를 리턴.

kernel virtual address가 mapping된 physical address를 리턴



https://luji.tistory.com/45

pagedir.c

pd는 kernel pool에 생긴다.

```c
/* Creates a new page directory that has mappings for kernel
   virtual addresses, but none for user virtual addresses.
   Returns the new page directory, or a null pointer if memory
   allocation fails. */
uint32_t *
pagedir_create (void)
{
  uint32_t *pd = palloc_get_page (0);
  if (pd != NULL)
    memcpy (pd, init_page_dir, PGSIZE);
  return pd;
}
```

→ kernel pool로

paging_init 함수에서 만든 init_page_dir를 가져와 복사 → 이 만든 p.d를 리턴.

★ 이래서 기존 pintos는 process 마다 pagedir_create 해도 똑같았던 것이다.

└ load 함수에서 t→pagedir = pagedir_create();

```c
/* Page directory with kernel mappings only. */
uint32_t *init_page_dir;
```

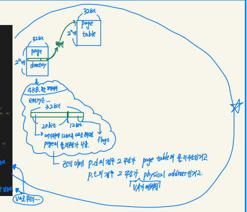
```c
/* Populates the base page directory and page table with the
   kernel virtual mapping, and then sets up the CPU to use the
   new page directory. Points init_page_dir to the page
   directory it creates. */
static void
paging_init (void)
{
  uint32_t *pd, *pt;
  size_t page;
  extern char _start, _end_kernel_text;

  pd = init_page_dir = palloc_get_page (PAL_ASSERT | PAL_ZERO);
  pt = NULL;
  for (page = 0; page < init_ram_pages; page++)
    {
      uintptr_t paddr = page * PGSIZE;
      char *vaddr = ptov (paddr);
      size_t pde_idx = pd_no (vaddr);
      size_t pte_idx = pt_no (vaddr);
      bool in_kernel_text = &_start <= vaddr && vaddr < &_end_kernel_text;

      if (pd[pde_idx] == 0)
        {
          pt = palloc_get_page (PAL_ASSERT | PAL_ZERO);
          pd[pde_idx] = pde_create (pt);
        }

      pt[pte_idx] = pte_create_kernel (vaddr, !in_kernel_text);
    }

  /* Store the physical address of the page directory into CR3
     aka PDBR (page directory base register). This activates our
     new page tables immediately. See [IA32-v2a] "MOV--Move
     to/from Control Registers" and [IA32-v3a] 3.7.5 "Base Address
     of the Page Directory". */
  asm volatile ("movl %0, %%cr3" : : "r" (vtop (init_page_dir)));
}
```

kernel pool → 0으로 초기화. 공간할당하고 그 공간의 포인터를 void*로 전달. 가상주소 → 32비트짜리배열! ram에 감당할 수 있는 page수. → "RAM"에 있는 모든 page 주소를 "물리주소"로 차곡차곡. → 가상주소로 바뀐다. → p.d / p.t 모두… 3GB + 12.8KB. 할당했던것은 본다. → 다 여기에 걸릴것이다. "page directory"를 증감. 가상주소. pt → uint32_t* — 안되면 PANIC. → uint32_t (물리주소). → uint32_t (물리주소)

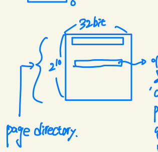
- base p.d랑 page table을 "populate"한다.
- kernel mapping은
- 새로운 pd 으로 셋팅. cpu에

page directory (init_page_dir). 32bit = 4byte. 물리 4KB 이니… 2^{10} byte. 2^{10}개. RAM의 kernel pool의 page table.

```c
/* Returns a PDE that points to page table PT. */
static inline uint32_t pde_create (uint32_t *pt) {
  ASSERT (pg_ofs (pt) == 0);
  return vtop (pt) | PTE_U | PTE_P | PTE_W;
}
```



PHYS_BASE



```c
/* Destroys page directory PD, freeing all the pages it
   references. */
void
pagedir_destroy (uint32_t *pd)
{
  uint32_t *pde;

  if (pd == NULL)
    return;

  ASSERT (pd != init_page_dir);
  for (pde = pd; pde < pd + pd_no (PHYS_BASE); pde++)
    if (*pde & PTE_P)
      {
        uint32_t *pt = pde_get_pt (*pde);
        uint32_t *pte;

        for (pte = pt; pte < pt + PGSIZE / sizeof *pte; pte++)
          if (*pte & PTE_P)
            palloc_free_page (pte_get_page (*pte));
        palloc_free_page (pt);
      }
  palloc_free_page (pd);
}
```

→ 그런 애는 없애면 안돼. 0xC0000000 : 3GB. 2^{12} / 2^{2}. page table 에 존재하면 → 제거. 경계해제. page directory 제거.

```c
/* Returns a pointer to the page table that page directory entry
   PDE, which must "present", points to. */
static inline uint32_t *pde_get_pt (uint32_t pde) {
  ASSERT (pde & PTE_P);
  return ptov (pde & PTE_ADDR);
}
```

→ 존재해야함.

pde의 상위 20bit에 뒤쪽 12bit는 0으로. (의미: page table 의 물리주소)를 → virtual address로.

32bit, 2^{10} → 이 32bit에서 PTE_P 체크, '0'이라면 page table이 없는 것임. page directory. 만약에 없다면? VA받아서 palloc_free_page로 page table 없애.

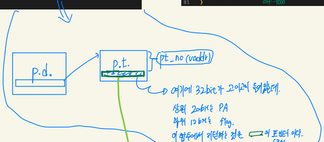
```c
/* Returns the address of the page table entry for virtual
   address VADDR in page directory PD.
   If PD does not have a page table for VADDR, behavior depends
   on CREATE. If CREATE is true, then a new page table is
   created and a pointer into it is returned. Otherwise, a null
   pointer is returned. */
static uint32_t *
lookup_page (uint32_t *pd, const void *vaddr, bool create)
{
  uint32_t *pt, *pde;

  ASSERT (pd != NULL);

  /* Shouldn't create new kernel virtual mappings. */
  ASSERT (!create || is_user_vaddr (vaddr));

  /* Check for a page table for VADDR.
     If one is missing, create one if requested. */
  pde = pd + pd_no (vaddr);
  if (*pde == 0)
    {
      if (create)
        {
          pt = palloc_get_page (PAL_ZERO);
          if (pt == NULL)
            return NULL;

          *pde = pde_create (pt);
        }
      else
        return NULL;
    }

  /* Return the page table entry. */
  pt = pde_get_pt (*pde);
  return &pt[pt_no (vaddr)];
}
```

pd에 있는 VA인 vaddr을 위한 page table entry를 반환한다. static함수다!

가상 / 가상. ⇔ if (create) then is_user_vaddr(vaddr)를 ASSERT. → page directory entry가 0 이라면… → page table을 만든다! → page table 없음. page table 없은 잘 만들었으니, 잘 받아왔나? → page table 에 주소 저장 (이걸 오로지 pd의 page table에 하나면 되는데…). 가상. 이 page table에 주소 저장하고 page directory entry를 만든다. → page table 의 주소. 물리.

```c
/* Returns a PDE that points to page table PT. */
static inline uint32_t pde_create (uint32_t *pt) {
  ASSERT (pg_ofs (pt) == 0);
  return vtop (pt) | PTE_U | PTE_P | PTE_W;
}
```

```c
/* Returns a pointer to the page table that page directory entry
   PDE, which must "present", points to. */
static inline uint32_t *pde_get_pt (uint32_t pde) {
  ASSERT (pde & PTE_P);
  return ptov (pde & PTE_ADDR);
}
```



→ 여기에 32bit가 그대로 들어있음. 상위 20bit가 PA, 하위 12bit가 flag. 이 형식에 저장된 32bit를 → 이 포인터 주소.

```c
/* Adds a mapping in page directory PD from user virtual page
   UPAGE to the physical frame identified by kernel virtual
   address KPAGE.
   UPAGE must not already be mapped.
   KPAGE should probably be a page obtained from the user pool
   with palloc_get_page().
   If WRITABLE is true, the new page is read/write;
   otherwise it is read-only.
   Returns true if successful, false if memory allocation
   failed. */
bool
pagedir_set_page (uint32_t *pd, void *upage, void *kpage, bool writable)
{
  uint32_t *pte;

  ASSERT (pg_ofs (upage) == 0);
  ASSERT (pg_ofs (kpage) == 0);
  ASSERT (is_user_vaddr (upage));
  ASSERT (vtop (kpage) >> PTSHIFT < init_ram_pages);
  ASSERT (pd != init_page_dir);

  pte = lookup_page (pd, upage, true);

  if (pte != NULL)
    {
      ASSERT ((*pte & PTE_P) == 0);
      *pte = pte_create_user (kpage, writable);
      return true;
    }
  else
    return false;
}
```

user virtual page. physical frame. → page table entry의 주소를 반환한다. → 존재하면 안됨. ⊙ 저 부분은 수정된다. 실제 32bit는 ㅁ로 해야한다.

```c
/* Returns a PTE that points to PAGE.
   The PTE's page is readable.
   If WRITABLE is true then it will be writable as well.
   The page will be usable only by ring 0 code (the kernel). */
static inline uint32_t pte_create_kernel (void *page, bool writable) {
  ASSERT (pg_ofs (page) == 0);
  return vtop (page) | PTE_P | (writable ? PTE_W : 0);
}
```

```c
/* Returns a PTE that points to PAGE.
   The PTE's page is readable.
   If WRITABLE is true then it will be writable as well.
   The page will be usable by both user and kernel code. */
static inline uint32_t pte_create_user (void *page, bool writable) {
  return pte_create_kernel (page, writable) | PTE_U;
}
```

```c
/* Looks up the physical address that corresponds to user virtual
   address UADDR in PD. Returns the kernel virtual address
   corresponding to that physical address, or a null pointer if
   UADDR is unmapped. */
void *
pagedir_get_page (uint32_t *pd, const void *uaddr)
{
  uint32_t *pte;

  ASSERT (is_user_vaddr (uaddr));

  pte = lookup_page (pd, uaddr, false);
  if (pte != NULL && (*pte & PTE_P) != 0)
    return pte_get_page (*pte) + pg_ofs (uaddr);
  else
    return NULL;
}
```

→ page table entry의 주소를 반환. → pte에 0 이면 NULL. 32bit: 20bit / 12bit.

```c
/* Returns a pointer to the page that page table entry PTE points
   to. */
static inline void *pte_get_page (uint32_t pte) {
  return ptov (pte & PTE_ADDR);
}
```

```c
/* Offset within a page. */
static inline unsigned pg_ofs (const void *va) {
  return (uintptr_t) va & PGMASK;
}
```

0…0[…] 12개

커널은 VA → PA mapping이 당연한거 아닌가?..?

커널에서 VA·PA가 있지? 유저도 다르게 있잖아. ∵ 여러 유저가 같은 VA를 쓰니 실제로도 VA → PA에 매핑이 되어야 PA값 사이에 있을 수 있다. 이걸 pd랑 pt로 할당해야 한다.

page directory는 entry로 page table의 주소를 가지고 있다.

page table '굴곡 있게도' (사실은 시작 주소의 의미) page 하나의 크기와 딱 맞음 (4KB) + RAM에 page-aligned 되어있음.

그래서 page table의 주소는 20bit로 표현 가능 (뒤에 12bit는 0으로 비었음)

p.d.e. : 20 bit | 12 bit — "page table"이 dirty·pre.access 등 정보 나타냄. ※ 에 저장된 bit는 "물리주소"임.

그럼 page table을 알아보자. 얘가… 실제로 사용하는 page를 담당.

p.t.e. : 20 bit | 12 bit — "page"가 dirty, present, accessed 등 정보를 나타냄. ※ 에 저장된 bit는 "물리주소"임.

p.d.e의 역할: p.t의 주소를 가리키기. → 어? p.t는 4KB에?
p.t.e의 역할: page의 주소를 가리키기. → 어? page는 4KB에?

"가리키는 것들도 같은 구조가 된다."

VA ↔ PA? 왜 palloc_get_page 하고 pde_create로 보내기…

커널… 가상/물리 다 써볼까?

— vtop 쓰는게 잘 이해된다. 언제, 왜 쓰는지.

왜 kernel virtual address를 반환하나 ??

pagedir.c 다보고.

process.c의 load 함수

exception.c 의 page_fault 함수

진짜 다 이어야 한다!

```c
/* Marks user virtual page UPAGE "not present" in page
   directory PD.  Later accesses to the page will fault.  Other
   bits in the page table entry are preserved.
   UPAGE need not be mapped. */
void
pagedir_clear_page (uint32_t *pd, void *upage)
{
  uint32_t *pte;

  ASSERT (pg_ofs (upage) == 0);
  ASSERT (is_user_vaddr (upage));

  pte = lookup_page (pd, upage, false);
  if (pte != NULL && (*pte & PTE_P) != 0)
    {
      *pte &= ~PTE_P;
      invalidate_pagedir (pd);
    }
}
```

user virtual page를 "not present"로 바꾼다.
해당 페이지로의 다음번 접근은 page fault.
page fault는 cpu가 발생시키겠지?
가상 / page인지 확인. / 물리
· NULL이면 아무것X
· 이미 not present면 아무것X
present bit를 '0'으로.

page table entry를 변경시키면
CPU에 있는 TLB랑 page table 이랑 말이 안맞을 수도 있다.
→ 그래서 TLB를 invalidate 하기위해 reactivate 한다.

```c
/* Seom page table changes can cause the CPU's translation
   lookaside buffer (TLB) to become out-of-sync with the page
   table.  When this happens, we have to "invalidate" the TLB by
   re-activating it.

   This function invalidates the TLB if PD is the active page
   directory.  (If PD is not active then its entries are not in
   the TLB, so there is no need to invalidate anything.) */
static void
invalidate_pagedir (uint32_t *pd)
{
  if (active_pd () == pd)
    {
      /* Re-activating PD clears the TLB.  See [IA32-v3a] 3.12
         "Translation Lookaside Buffers (TLBs)". */
      pagedir_activate (pd);
    }
}
```

가상 / 지금 activate 되어있는 p.d가 인자로온게 pd가 아니라면 아무것 하지마. ∵ TLB를 invalidate 할 필요가 없거든.

```c
/* Returns the currently active page directory. */
static uint32_t *
active_pd (void)
{
  /* Copy CR3, the page directory base register (PDBR), into
     `pd'.
     See [IA32-v2a] "MOV--Move to/from Control Registers" and
     [IA32-v3a] 3.7.5 "Base Address of the Page Directory". */
  uintptr_t pd;
  asm volatile ("movl %%cr3, %0" : "=r" (pd));
  return ptov (pd);
}
```

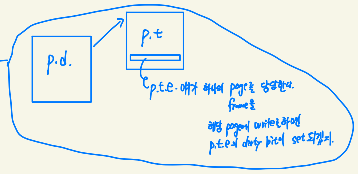
P→V
물리 / 가상 / 여기에 있는거 VA로 바꿔서 보내주네.

```c
/* Loads page directory PD into the CPU's page directory base
   register. */
void
pagedir_activate (uint32_t *pd)
{
  if (pd == NULL)
    pd = init_page_dir;

  /* Store the physical address of the page directory into CR3
     aka PDBR (page directory base register).  This activates our
     new page tables immediately.  See [IA32-v2a] "MOV--Move
     to/from Control Registers" and [IA32-v3a] 3.7.5 "Base
     Address of the Page Directory". */
  asm volatile ("movl %0, %%cr3" : : "r" (vtop (pd)) : "memory");
}
```

V→P
가상 / 물리

```c
/* Returns true if the PTE for virtual page VPAGE in PD is dirty,
   that is, if the page has been modified since the PTE was
   installed.
   Returns false if PD contains no PTE for VPAGE. */
bool
pagedir_is_dirty (uint32_t *pd, const void *vpage)
{
  uint32_t *pte = lookup_page (pd, vpage, false);
  return pte != NULL && (*pte & PTE_D) != 0;
}
```

~이후로
p.d에 있는




```c
/* Set the dirty bit to DIRTY in the PTE for virtual page VPAGE
   in PD. */
void
pagedir_set_dirty (uint32_t *pd, const void *vpage, bool dirty)
{
  uint32_t *pte = lookup_page (pd, vpage, false);
  if (pte != NULL)
    {
      if (dirty)
        *pte |= PTE_D;
      else
        {
          *pte &= ~(uint32_t) PTE_D;
          invalidate_pagedir (pd);
        }
    }
}
```

p.t.e의 포인터 받아와서 dirty 바꿔준다.
이거 바꾸면 TLB 안거든다.
dirty를 1→0으로 할때 TLB invalidate.

```c
/* Returns true if the PTE for virtual page VPAGE in PD has been
   accessed recently, that is, between the time the PTE was
   installed and the last time it was cleared.  Returns false if
   PD contains no PTE for VPAGE. */
bool
pagedir_is_accessed (uint32_t *pd, const void *vpage)
{
  uint32_t *pte = lookup_page (pd, vpage, false);
  return pte != NULL && (*pte & PTE_A) != 0;
}

/* Sets the accessed bit to ACCESSED in the PTE for virtual page
   VPAGE in PD. */
void
pagedir_set_accessed (uint32_t *pd, const void *vpage, bool accessed)
{
  uint32_t *pte = lookup_page (pd, vpage, false);
  if (pte != NULL)
    {
      if (accessed)
        *pte |= PTE_A;
      else
        {
          *pte &= ~(uint32_t) PTE_A;
          invalidate_pagedir (pd);
        }
    }
}
```

process.c

```c
/* Free the current process's resources. */
void
process_exit (void)
{
  struct thread *cur = thread_current ();
  uint32_t *pd;
  struct list_elem * e;
  struct process * p;
  struct list_elem * e_file;
  struct file_desc * p_fd;
  // lab2 added
  if (cur->exit_status == INIT_EXIT_STATUS) //case 1 in process_wait
  {
    exit(-1); //inform parent to it is abnormal process_exit. its svae to call exit because main logic on thread_exit not yet occured.
  }

  // just calling close for each fd_number that current process has. we are just customor in this situation. close will do free.
  e_file = list_begin(&(cur->fd_list));
  while(e_file != list_end(&(cur->fd_list)))
  {
    p_fd = list_entry(e_file, struct file_desc, elem_f);
    e_file = list_next(e_file);
    close(p_fd->fd_number);//read-bad-ptr test wrong due to this point.
  }
  // free child_list's element
  while (!list_empty(&(cur->child_list)))
  {
    e = list_pop_front (&(cur->child_list));
    p = list_entry(e, struct process, elem_p);
    free(p);
  }

  if (cur -> executing_file)
  {
    file_allow_write(cur->executing_file);
    file_close(cur->executing_file);
  }

  // if parent is waiting for 'cur' thread to exit, signal to parent
  if (cur->parent->waiting_child_pid == cur->tid)
  {
    sema_up (&(cur->parent->sema_child_exit));
  }
  /* Destroy the current process's page directory and switch back
     to the kernel-only page directory. */
  pd = cur->pagedir;
  if (pd != NULL)
    {
      /* Correct ordering here is crucial.  We must set
         cur->pagedir to NULL before switching page directories,
         so that a timer interrupt can't switch back to the
         process page directory.  We must activate the base page
         directory before destroying the process's page
         directory, or our active page directory will be one
         that's been freed (and cleared). */
      cur->pagedir = NULL;
      pagedir_activate (NULL);
      pagedir_destroy (pd);
    }
}
```


```c
/* Loads page directory PD into the CPU's page directory base
   register. */
void
pagedir_activate (uint32_t *pd)
{
  if (pd == NULL)
    pd = init_page_dir;

  /* Store the physical address of the page directory into CR3
     aka PDBR (page directory base register).  This activates our
     new page tables immediately.  See [IA32-v2a] "MOV--Move
     to/from Control Registers" and [IA32-v3a] 3.7.5 "Base
     Address of the Page Directory". */
  asm volatile ("movl %0, %%cr3" : : "r" (vtop (pd)) : "memory");
}
```


CPU님에게 "user code" 돌린다고 말씀드린다!

```c
/* Sets up the CPU for running user code in the current
   thread.
   This function is called on every context switch. */
void
process_activate (void)
{
  struct thread *t = thread_current ();

  /* Activate thread's page tables. */
  pagedir_activate (t->pagedir);

  /* Set thread's kernel stack for use in processing
     interrupts. */
  tss_update ();
}
```

현재 process의 thread의 stack을 다음에 interrupt 생기면 쓸 수 있게 준비.

```c
/* Sets the ring 0 stack pointer in the TSS to point to the end
   of the thread stack. */
void
tss_update (void)
{
  ASSERT (tss != NULL);
  tss->esp0 = (uint8_t *) thread_current () + PGSIZE;
}
```

8bit 의 포인터 4가 여전히 32bit.

```c
/* Returns the running thread.
   This is running_thread() plus a couple of sanity checks.
   See the big comment at the top of thread.h for details. */
struct thread *
thread_current (void)
{
  struct thread *t = running_thread ();

  /* Make sure T is really a thread.
     If either of these assertions fire, then your thread may
     have overflowed its stack.  Each thread has less than 4 kB
     of stack, so a few big automatic arrays or moderate
     recursion can cause stack overflow. */
  ASSERT (is_thread (t));
  ASSERT (t->status == THREAD_RUNNING);

  return t;
}
```

```c
/* Returns the running thread. */
struct thread *
running_thread (void)
{
  uint32_t *esp;

  /* Copy the CPU's stack pointer into `esp', and then round that
     down to the start of a page.  Because `struct thread' is
     always at the beginning of a page and the stack pointer is
     somewhere in the middle, this locates the curent thread. */
  asm ("mov %%esp, %0" : "=g" (esp));
  return pg_round_down (esp);
}
```

IA32 Intel Arch

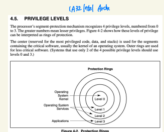
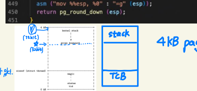


Figure 4-2. Protection Rings

```c
/* The Task-State Segment (TSS).

   Instances of the TSS, an x86-specific structure, are used to
   define "tasks", a form of support for multitasking built right
   into the processor.  However, for various reasons including
   portability, speed, and flexibility, most x86 OSes almost
   completely ignore the TSS.  We are no exception.

   Unfortunately, there is one thing that can only be done using
   a TSS: stack switching for interrupts that occur in user mode.
   When an interrupt occurs in user mode (ring 3), the processor
   consults the ss0 and esp0 members of the current TSS to
   determine the stack to use for handling the interrupt.  Thus,
   we must create a TSS and initialize at least these fields, and
   this is precisely what this file does.

   When an interrupt is handled by an interrupt or trap gate
   (which applies to all interrupts we handle), an x86 processor
   works like this:

     - If the code interrupted by the interrupt is in the same
       ring as the interrupt handler, then no stack switch takes
       place.  This is the case for interrupts that happen when
       we're running in the kernel.  The contents of the TSS are
       irrelevant for this case.

     - If the interrupted code is in a different ring from the
       handler, then the processor switches to the stack
       specified in the TSS for the new ring.  This is the case
       for interrupts that happen when we're in user space.  It's
       important that we switch to a stack that's not already in
       use, to avoid corruption.  Because we're running in user
       space, we know that the current process's kernel stack is
       not in use, so we can always use that.  Thus, when the
       scheduler switches threads, it also changes the TSS's
       stack pointer to point to the new thread's kernel stack.
       (The call is in thread_schedule_tail() in thread.c.)

   See [IA32-v3a] 6.2.1 "Task-State Segment (TSS)" for a
   description of the TSS.  See [IA32-v3a] 5.12.1 "Exception- or
   Interrupt-Handler Procedures" for a description of when and
   how stack switching occurs during an interrupt. */
struct tss
  {
    uint16_t back_link, :16;
    void *esp0;                         /* Ring 0 stack virtual address. */
    uint16_t ss0, :16;                  /* Ring 0 stack segment selector. */
    void *esp1;
    uint16_t ss1, :16;
    void *esp2;
    uint16_t ss2, :16;
    uint32_t cr3;
    void (*eip) (void);
    uint32_t eflags;
    uint32_t eax, ecx, edx, ebx;
    uint32_t esp, ebp, esi, edi;
    uint16_t es, :16;
    uint16_t cs, :16;
    uint16_t ss, :16;
    uint16_t ds, :16;
    uint16_t fs, :16;
    uint16_t gs, :16;
    uint16_t ldt, :16;
    uint16_t trace, bitmap;
  };

/* Kernel TSS. */
static struct tss *tss;
```

OS 대부분은 portability 때문에 TSS 무시했다.
근데... 때문에 TSS를 쓰게됐다...
유저모드에서 인터럽트를 위한 stack switching
interrupt를 handling 하는데 사용할 stack을 찾아야 해!
ss0이랑 esp0 에게 위치함.
이때.
이때.
바꿔줘야지 그래.

레지스터 %esp에는 ★ 가 있다.
32bit




```c
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif
    //lab2 added

    // for exit
    int exit_status;
    struct semaphore sema_child_exit;
    tid_t waiting_child_pid;

    // for exec
    struct semaphore sema_child_exec;
    bool is_child_load_success;

    // for parent-child relation
    struct list child_list; // struct process will be element of this list
    struct thread * parent;

    // for file handling
    struct list fd_list; // struct file_desc will be element of this list
    int next_fd_number;
    struct lock fd_number_lock;

    // for denying write to executables
    struct file * executing_file;

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

p.d가 thread마다 (process) 하나씩 있기도 하고.
전체에도 있네...

생각해보면... user stack은 load 함수에서 user pool에 할당된다.
(thread_create) thread는 생성될때 page가 allocate된다. 그 페이지안에서 채우고 붉고 stack이 사용.

```c
/* Initializes the kernel TSS. */
void
tss_init (void)
{
  /* Our TSS is never used in a call gate or task gate, so only a
     few fields of it are ever referenced, and those are the only
     ones we initialize. */
  tss = palloc_get_page (PAL_ASSERT | PAL_ZERO);
  tss->ss0 = SEL_KDSEG;
  tss->bitmap = 0xdfff;
  tss_update ();
}

/* Returns the kernel TSS. */
struct tss *
tss_get (void)
{
  ASSERT (tss != NULL);
  return tss;
}
```

0x10


```c
/* Sets the ring 0 stack pointer in the TSS to point to the end
   of the thread stack. */
void
tss_update (void)
{
  ASSERT (tss != NULL);
  tss->esp0 = (uint8_t *) thread_current () + PGSIZE;
}
```

load함수의 배경을 생각해보자...

```c
/* Loads an ELF executable from FILE_NAME into the current thread.
   Stores the executable's entry point into *EIP
   and its initial stack pointer into *ESP.
   Returns true if successful, false otherwise. */
bool
load (const char *file_name, void (**eip) (void), void **esp)
{
  struct thread *t = thread_current ();
  struct Elf32_Ehdr ehdr;
  struct file *file = NULL;
  off_t file_ofs;
  bool success = false;
  int i;

  /* Allocate and activate page directory. */
  t->pagedir = pagedir_create ();
  if (t->pagedir == NULL)
    goto done;
  process_activate ();

  /* Open executable file. */
  file = filesys_open (file_name);
  if (file == NULL)
    {
      printf ("load: %s: open failed\n", file_name);
      goto done;
    }

  /* Read and verify executable header. */
  if (file_read (file, &ehdr, sizeof ehdr) != sizeof ehdr
      || memcmp (ehdr.e_ident, "\177ELF\1\1\1", 7)
      || ehdr.e_type != 2
      || ehdr.e_machine != 3
      || ehdr.e_version != 1
      || ehdr.e_phentsize != sizeof (struct Elf32_Phdr)
      || ehdr.e_phnum > 1024)
    {
      printf ("load: %s: error loading executable\n", file_name);
      goto done;
    }

  /* Read program headers. */
  file_ofs = ehdr.e_phoff;
  for (i = 0; i < ehdr.e_phnum; i++)
    {
      struct Elf32_Phdr phdr;

      if (file_ofs < 0 || file_ofs > file_length (file))
        goto done;
      file_seek (file, file_ofs);

      if (file_read (file, &phdr, sizeof phdr) != sizeof phdr)
        goto done;
      file_ofs += sizeof phdr;
      switch (phdr.p_type)
        {
        case PT_NULL:
        case PT_NOTE:
        case PT_PHDR:
        case PT_STACK:
        default:
          /* Ignore this segment. */
          break;
        case PT_DYNAMIC:
        case PT_INTERP:
        case PT_SHLIB:
          goto done;
        case PT_LOAD:
          if (validate_segment (&phdr, file))
            {
              bool writable = (phdr.p_flags & PF_W) != 0;
              uint32_t file_page = phdr.p_offset & ~PGMASK;
              uint32_t mem_page = phdr.p_vaddr & ~PGMASK;
              uint32_t page_offset = phdr.p_vaddr & PGMASK;
              uint32_t read_bytes, zero_bytes;
              if (phdr.p_filesz > 0)
                {
                  /* Normal segment.
                     Read initial part from disk and zero the rest. */
                  read_bytes = page_offset + phdr.p_filesz;
                  zero_bytes = (ROUND_UP (page_offset + phdr.p_memsz, PGSIZE)
                                - read_bytes);
                }
              else
                {
                  /* Entirely zero.
                     Don't read anything from disk. */
                  read_bytes = 0;
                  zero_bytes = ROUND_UP (page_offset + phdr.p_memsz, PGSIZE);
                }
              if (!load_segment (file, file_page, (void *) mem_page,
                                 read_bytes, zero_bytes, writable))
                goto done;
            }
          else
            goto done;
          break;
        }
    } // for

  /* Set up stack. */
  if (!setup_stack (esp))
    goto done;

  /* Start address. */
  *eip = (void (*) (void)) ehdr.e_entry;

  success = true;

done:
  /* We arrive here whether the load is successful or not. */
  file_close (file);
  return success;
}
```

(?) into

그 thread 만 쓰는 p.d.

파일을 열고...

파일에 있는 걸 ehdr로 읽어낸다. 파일의 앞단에.

1 2 3 4 5 6 7

약속된 형태 아니면 컷.

여기서부터 유의미한 값이 들어있는듯. (앞에 헤더정보 거르는듯)

헤더 넘겼으니 이제 파일을 읽자.

file의 offset (아마 가리키는위치겠지)를 따로 다루네.

타입에따라 하는일이 다르다.

이외x. 다음 segment본다.

그만. 4가지.

segment 하나를 phdr에 담았다. 검증작업.

0...0 1...1 (20개, 12개)

user virtual address로 사용된다.

```c
/* Creates a new page directory that has mappings for kernel
   virtual addresses, but none for user virtual addresses.
   Returns the new page directory, or a null pointer if memory
   allocation fails. */
uint32_t *
pagedir_create (void)
{
  uint32_t *pd = palloc_get_page (0);
  if (pd != NULL)
    memcpy (pd, init_page_dir, PGSIZE);
  return pd;
}
```

```c
/* Sets up the CPU for running user code in the current
   thread.
   This function is called on every context switch. */
void
process_activate (void)
{
  struct thread *t = thread_current ();

  /* Activate thread's page tables. */
  pagedir_activate (t->pagedir);

  /* Set thread's kernel stack for use in processing
     interrupts. */
  tss_update ();
}
```

→ thread의 stack을 나중을 위해 작업해둔.

```c
/* Loads page directory PD into the CPU's page directory base
   register. */
void
pagedir_activate (uint32_t *pd)
{
  if (pd == NULL)
    pd = init_page_dir;

  /* Store the physical address of the page directory into CR3
     aka PDBR (page directory base register).  This activates our
     new page tables immediately.  See [IA32-v2a] "MOV--Move
     to/from Control Registers" and [IA32-v3a] 3.7.5 "Base
     Address of the Page Directory". */
  asm volatile ("movl %0, %%cr3" : : "r" (vtop (pd)) : "memory");
}
```

```c
/* Opens the file with the given NAME.
   Returns the new file if successful or a null pointer
   otherwise.
   Fails if no file named NAME exists,
   or if an internal memory allocation fails. */
struct file *
filesys_open (const char *name)
{
  struct dir *dir = dir_open_root ();
  struct inode *inode = NULL;

  if (dir != NULL)
    dir_lookup (dir, name, &inode);
  dir_close (dir);

  return file_open (inode);
}
```

```c
/* Reads SIZE bytes from FILE into BUFFER,
   starting at the file's current position.
   Returns the number of bytes actually read,
   which may be less than SIZE if end of file is reached.
   Advances FILE's position by the number of bytes read. */
off_t
file_read (struct file *file, void *buffer, off_t size)
{
  off_t bytes_read = inode_read_at (file->inode, buffer, size, file->pos);
  file->pos += bytes_read;
  return bytes_read;
}
```

```c
/* Find the first differing byte in the two blocks of SIZE bytes
   at A and B.  Returns a positive value if the byte in A is
   greater, a negative value if the byte in B is greater, or zero
   if blocks A and B are equal. */
int
memcmp (const void *a_, const void *b_, size_t size)
{
  const unsigned char *a = a_;
  const unsigned char *b = b_;

  ASSERT (a != NULL || size == 0);
  ASSERT (b != NULL || size == 0);

  for (; size-- > 0; a++, b++)
    if (*a != *b)
      return *a > *b ? +1 : -1;
  return 0;
}
```

```c
/* Opens a file for the given INODE, of which it takes ownership,
   and returns the new file.  Returns a null pointer if an
   allocation fails or if INODE is null. */
struct file *
file_open (struct inode *inode)
{
  struct file *file = calloc (1, sizeof *file);
  if (inode != NULL && file != NULL)
    {
      file->inode = inode;
      file->pos = 0;
      file->deny_write = false;
      return file;
    }
  else
    {
      inode_close (inode);
      free (file);
      return NULL;
    }
}
```

```c
/* Returns the size of FILE in bytes. */
off_t
file_length (struct file *file)
{
  ASSERT (file != NULL);
  return inode_length (file->inode);
}

/* Sets the current position in FILE to NEW_POS bytes from the
   start of the file. */
void
file_seek (struct file *file, off_t new_pos)
{
  ASSERT (file != NULL);
  ASSERT (new_pos >= 0);
  file->pos = new_pos;
}
```


```c
/* Reads SIZE bytes from FILE into BUFFER,
   starting at the file's current position.
   Returns the number of bytes actually read,
   which may be less than SIZE if end of file is reached.
   Advances FILE's position by the number of bytes read. */
off_t
file_read (struct file *file, void *buffer, off_t size)
{
  off_t bytes_read = inode_read_at (file->inode, buffer, size, file->pos);
  file->pos += bytes_read;
  return bytes_read;
}
```


```c
/* Executable header.  See [ELF1] 1-4 to 1-8.
   This appears at the very beginning of an ELF binary. */
struct Elf32_Ehdr
  {
    unsigned char e_ident[16];
    Elf32_Half    e_type;
    Elf32_Half    e_machine;
    Elf32_Word    e_version;
    Elf32_Addr    e_entry;
    Elf32_Off     e_phoff;
    Elf32_Off     e_shoff;
    Elf32_Word    e_flags;
    Elf32_Half    e_ehsize;
    Elf32_Half    e_phentsize;
    Elf32_Half    e_phnum;
    Elf32_Half    e_shentsize;
    Elf32_Half    e_shnum;
    Elf32_Half    e_shstrndx;
  };
```

```c
/* Program header.  See [ELF1] 2-2 to 2-4.
   There are e_phnum of these, starting at file offset e_phoff
   (see [ELF1] 1-6). */
struct Elf32_Phdr
  {
    Elf32_Word p_type;
    Elf32_Off  p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    Elf32_Word p_filesz;
    Elf32_Word p_memsz;
    Elf32_Word p_flags;
    Elf32_Word p_align;
  };

/* Values for p_type.  See [ELF1] 2-3. */
#define PT_NULL    0            /* Ignore. */
#define PT_LOAD    1            /* Loadable segment. */
#define PT_DYNAMIC 2            /* Dynamic linking info. */
#define PT_INTERP  3            /* Name of dynamic loader. */
#define PT_NOTE    4            /* Auxiliary info. */
#define PT_SHLIB   5            /* Reserved. */
#define PT_PHDR    6            /* Program header table. */
#define PT_STACK   0x6474e551   /* Stack segment. */

/* Flags for p_flags.  See [ELF3] 2-3 and 2-4. */
#define PF_X 1          /* Executable. */
#define PF_W 2          /* Writable. */
#define PF_R 4          /* Readable. */
```

```c
/* Checks whether PHDR describes a valid, loadable segment in
   FILE and returns true if so, false otherwise. */
static bool
validate_segment (const struct Elf32_Phdr *phdr, struct file *file)
{
  /* p_offset and p_vaddr must have the same page offset. */
  if ((phdr->p_offset & PGMASK) != (phdr->p_vaddr & PGMASK))
    return false;

  /* p_offset must point within FILE. */
  if (phdr->p_offset > (Elf32_Off) file_length (file))
    return false;

  /* p_memsz must be at least as big as p_filesz. */
  if (phdr->p_memsz < phdr->p_filesz)
    return false;

  /* The segment must not be empty. */
  if (phdr->p_memsz == 0)
    return false;

  /* The virtual memory region must both start and end within the
     user address space range. */
  if (!is_user_vaddr ((void *) phdr->p_vaddr))
    return false;
  if (!is_user_vaddr ((void *) (phdr->p_vaddr + phdr->p_memsz)))
    return false;

  /* The region cannot "wrap around" across the kernel virtual
     address space. */
  if (phdr->p_vaddr + phdr->p_memsz < phdr->p_vaddr)
    return false;

  /* Disallow mapping page 0.
     Not only is it a bad idea to map page 0, but if we allowed
     it then user code that passed a null pointer to system calls
     could quite likely panic the kernel by way of null pointer
     assertions in memcpy(), etc. */
  if (phdr->p_vaddr < PGSIZE)
    return false;

  /* It's okay. */
  return true;
}
```

```c
/* Base address of the 1:1 physical-to-virtual mapping.  Physical
   memory is mapped starting at this virtual address.  Thus,
   physical address 0 is accessible at PHYS_BASE, physical
   address address 0x1234 at (uint8_t *) PHYS_BASE + 0x1234, and
   so on.

   This address also marks the end of user programs' address
   space.  Up to this point in memory, user programs are allowed
   to map whatever they like.  At this point and above, the
   virtual address space belongs to the kernel. */
#define PHYS_BASE ((void *) LOADER_PHYS_BASE)

/* Returns true if VADDR is a user virtual address. */
static inline bool
is_user_vaddr (const void *vaddr)
{
  return vaddr < PHYS_BASE;
}
```

```c
/* Loads a segment starting at offset OFS in FILE at address
   UPAGE.  In total, READ_BYTES + ZERO_BYTES bytes of virtual
   memory are initialized, as follows:

      - READ_BYTES bytes at UPAGE must be read from FILE
        starting at offset OFS.

      - ZERO_BYTES bytes at UPAGE + READ_BYTES must be zeroed.

   The pages initialized by this function must be writable by the
   user process if WRITABLE is true, read-only otherwise.

   Return true if successful, false if a memory allocation error
   or disk read error occurs. */
static bool
load_segment (struct file *file, off_t ofs, uint8_t *upage,
              uint32_t read_bytes, uint32_t zero_bytes, bool writable)
{
  ASSERT ((read_bytes + zero_bytes) % PGSIZE == 0);
  ASSERT (pg_ofs (upage) == 0);
  ASSERT (ofs % PGSIZE == 0);

  file_seek (file, ofs);
  while (read_bytes > 0 || zero_bytes > 0)
    {
      /* Calculate how to fill this page.
         We will read PAGE_READ_BYTES bytes from FILE
         and zero the final PAGE_ZERO_BYTES bytes. */
      size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
      size_t page_zero_bytes = PGSIZE - page_read_bytes;

      /* Get a page of memory. */
      uint8_t *kpage = palloc_get_page (PAL_USER);
      if (kpage == NULL)
        return false;

      /* Load this page. */
      if (file_read (file, kpage, page_read_bytes) != (int) page_read_bytes)
        {
          palloc_free_page (kpage);
          return false;
        }
      memset (kpage + page_read_bytes, 0, page_zero_bytes);

      /* Add the page to the process's address space. */
      if (!install_page (upage, kpage, writable))
        {
          palloc_free_page (kpage);
          return false;
        }

      /* Advance. */
      read_bytes -= page_read_bytes;
      zero_bytes -= page_zero_bytes;
      upage += PGSIZE;
    }
  return true;
}
```

가상

user pool에 page 받는다.

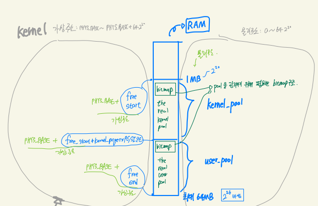
"할당받은곳으로 읽어온다." → 이건 lazy하게 할듯?

시키는대로 안하면 free후 false리턴.

남은건 0으로.

user 가상주소. 가상 (kernel)

)얘는 그대로해둬야 나중에 page_fault가 뜰 수 있을듯. 아 될수도.




```c
/* Adds a mapping from user virtual address UPAGE to kernel
   virtual address KPAGE to the page table.
   If WRITABLE is true, the user process may modify the page;
   otherwise, it is read-only.
   UPAGE must not already be mapped.
   KPAGE should probably be a page obtained from the user pool
   with palloc_get_page().
   Returns true on success, false if UPAGE is already mapped or
   if memory allocation fails. */
static bool
install_page (void *upage, void *kpage, bool writable)
{
  struct thread *t = thread_current ();

  /* Verify that there's not already a page at that virtual
     address, then map our page there. */
  return (pagedir_get_page (t->pagedir, upage) == NULL
          && pagedir_set_page (t->pagedir, upage, kpage, writable));
}
```

NULL이어야 없는것.

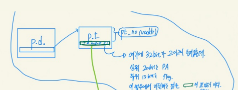
이제 upage 영역의 주소를 읽으면 '실제로' kpage가서 값읽는다.

user virtual page

cpu가 주소번역을 그렇게 하게끔 p.d.에 기록해둠.

이 부분 설명 쓰기.

코드채팅해야지.




```c
/* Adds a mapping in page directory PD from user virtual page
   UPAGE to the physical frame identified by kernel virtual
   address KPAGE.
   UPAGE must not already be mapped.
   KPAGE should probably be a page obtained from the user pool
   with palloc_get_page().
   If WRITABLE is true, the new page is read/write;
   otherwise it is read-only.
   Returns true if successful, false if memory allocation
   failed. */
bool
pagedir_set_page (uint32_t *pd, void *upage, void *kpage, bool writable)
{
  uint32_t *pte;

  ASSERT (pg_ofs (upage) == 0);
  ASSERT (pg_ofs (kpage) == 0);
  ASSERT (is_user_vaddr (upage));
  ASSERT (vtop (kpage) >> PTSHIFT < init_ram_pages);
  ASSERT (pd != init_page_dir);

  pte = lookup_page (pd, upage, true);

  if (pte != NULL)
    {
      ASSERT ((*pte & PTE_P) == 0);
      *pte = pte_create_user (kpage, writable);
      return true;
    }
  else
    return false;
}
```

user virtual page, physical frame

page table entry의 주소를 얻어와준다.

저 부분을 수행하며 실제 32bit는 이로채워진다.

```c
/* Looks up the physical address that corresponds to user virtual
   address UADDR in PD.  Returns the kernel virtual address
   corresponding to that physical address, or a null pointer if
   UADDR is unmapped. */
void *
pagedir_get_page (uint32_t *pd, const void *uaddr)
{
  uint32_t *pte;

  ASSERT (is_user_vaddr (uaddr));

  pte = lookup_page (pd, uaddr, false);
  if (pte != NULL && (*pte & PTE_P) != 0)
    return pte_get_page (*pte) + pg_ofs (uaddr);
  else
    return NULL;
}
```

32bit, 20bit, 12bit

```c
/* Page fault handler.  This is a skeleton that must be filled in
   to implement virtual memory.  Some solutions to project 2 may
   also require modifying this code.

   At entry, the address that faulted is in CR2 (Control Register
   2) and information about the fault, formatted as described in
   the PF_* macros in exception.h, is in F's error_code member.  The
   example code here shows how to parse that information.  You
   can find more information about both of these in the
   description of "Interrupt 14--Page Fault Exception (#PF)" in
   [IA32-v3a] section 5.15 "Exception and Interrupt Reference". */
static void
page_fault (struct intr_frame *f)
{
  bool not_present;  /* True: not-present page, false: writing r/o page. */
  bool write;        /* True: access was write, false: access was read. */
  bool user;         /* True: access by user, false: access by kernel. */
  void *fault_addr;  /* Fault address. */

  /* Obtain faulting address, the virtual address that was
     accessed to cause the fault.  It may point to code or to
     data.  It is not necessarily the address of the instruction
     that caused the fault (that's f->eip).
     See [IA32-v2a] "MOV--Move to/from Control Registers" and
     [IA32-v3a] 5.15 "Interrupt 14--Page Fault Exception
     (#PF)". */
  asm ("movl %%cr2, %0" : "=r" (fault_addr));

  /* Turn interrupts back on (they were only off so that we could
     be assured of reading CR2 before it changed). */
  intr_enable ();

  /* Count page faults. */
  page_fault_cnt++;

  /* Determine cause. */
  not_present = (f->error_code & PF_P) == 0;
  write = (f->error_code & PF_W) != 0;
  user = (f->error_code & PF_U) != 0;

  /* To implement virtual memory, delete the rest of the function
     body, and replace it with code that brings in the page to
     which fault_addr refers. */
  printf ("Page fault at %p: %s error %s page in %s context.\n",
          fault_addr,
          not_present ? "not present" : "rights violation",
          write ? "writing" : "reading",
          user ? "user" : "kernel");
  kill (f);
}
```

```c
/* Handler for an exception (probably) caused by a user process. */
static void
kill (struct intr_frame *f)
{
  /* This interrupt is one (probably) caused by a user process.
     For example, the process might have tried to access unmapped
     virtual memory (a page fault).  For now, we simply kill the
     user process.  Later, we'll want to handle page faults in
     the kernel.  Real Unix-like operating systems pass most
     exceptions back to the process via signals, but we don't
     implement them. */

  /* The interrupt frame's code segment value tells us where the
     exception originated. */
  switch (f->cs)
    {
    case SEL_UCSEG:
      /* User's code segment, so it's a user exception, as we
         expected.  Kill the user process.  */
      printf ("%s: dying due to interrupt %#04x (%s).\n",
              thread_name (), f->vec_no, intr_name (f->vec_no));
      intr_dump_frame (f);
      thread_exit ();

    case SEL_KCSEG:
      /* Kernel's code segment, which indicates a kernel bug.
         Kernel code shouldn't throw exceptions.  (Page faults
         may cause kernel exceptions--but they shouldn't arrive
         here.)  Panic the kernel to make the point.  */
      intr_dump_frame (f);
      PANIC ("Kernel bug - unexpected interrupt in kernel");

    default:
      /* Some other code segment?  Shouldn't happen.  Panic the
         kernel. */
      printf ("Interrupt %#04x (%s) in unknown segment %04x\n",
             f->vec_no, intr_name (f->vec_no), f->cs);
      thread_exit ();
    }
}
```

이거쓰기,

지금은 그냥 kill

나중엔 이걸 다룰거임.

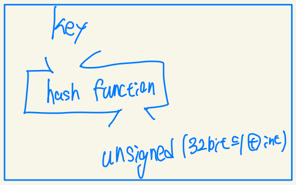
1. 지금까지 분석한걸 추가하기 (위에보면서 캡쳐넣기)

2. 조교문서읽오 ...

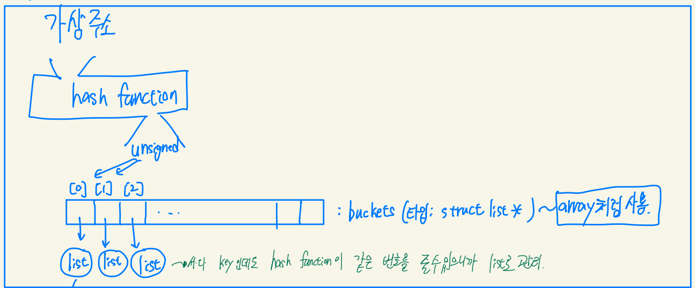
SUP p.t 가 담당하는 정보들이 각각 어디서사용되는지 연결해.

```c
/* Hash table. */
struct hash
  {
    size_t elem_cnt;            /* Number of elements in table. */
    size_t bucket_cnt;          /* Number of buckets, a power of 2. */
    struct list *buckets;       /* Array of `bucket_cnt' lists. */
    hash_hash_func *hash;       /* Hash function. */
    hash_less_func *less;       /* Comparison function. */
    void *aux;                  /* Auxiliary data for `hash' and `less'. */
  };
```

hash는 현재 이런것임.



이렇게 사용할것임.



↳ 이 list의 element에 가상주소와 관련된 정보가 담길것임.

초기화

```c
/* Initializes hash table H to compute hash values using HASH and
   compare hash elements using LESS, given auxiliary data AUX. */
bool
hash_init (struct hash *h,
           hash_hash_func *hash, hash_less_func *less, void *aux)
{
  h->elem_cnt = 0;
  h->bucket_cnt = 4;
  h->buckets = malloc (sizeof *h->buckets * h->bucket_cnt);   // list처럼 쓸것이니까.
  h->hash = hash;
  h->less = less;      // — hash의 element 비교위함.
  h->aux = aux;

  if (h->buckets != NULL)
    {
      hash_clear (h, NULL);
      return true;
    }
  else
    return false;
}
```

```c
/* Removes all the elements from H.

   If DESTRUCTOR is non-null, then it is called for each element
   in the hash.  DESTRUCTOR may, if appropriate, deallocate the
   memory used by the hash element.  However, modifying hash
   table H while hash_clear() is running, using any of the
   functions hash_clear(), hash_destroy(), hash_insert(),
   hash_replace(), or hash_delete(), yields undefined behavior,
   whether done in DESTRUCTOR or elsewhere. */
void
hash_clear (struct hash *h, hash_action_func *destructor)   // ↙ NULL
{
  size_t i;

  for (i = 0; i < h->bucket_cnt; i++)
    {
      struct list *bucket = &h->buckets[i];

      if (destructor != NULL)
        while (!list_empty (bucket))   // 리스트를 죄다 비우려고 하는구나.
          {
            struct list_elem *list_elem = list_pop_front (bucket);
            struct hash_elem *hash_elem = list_elem_to_hash_elem (list_elem);   // → hash의
            destructor (hash_elem, h->aux);   // → hash_elem의 할당해제를 다룬다. → 내책임.
          }

      list_init (bucket);   // → bucket은 'struct list'의 array이니까. array의 각 list를 초기화해야함.
    }

  h->elem_cnt = 0;
}
```

마지막 정리

```c
/* Destroys hash table H.

   If DESTRUCTOR is non-null, then it is first called for each
   element in the hash.  DESTRUCTOR may, if appropriate,
   deallocate the memory used by the hash element.  However,
   modifying hash table H while hash_clear() is running, using
   any of the functions hash_clear(), hash_destroy(),
   hash_insert(), hash_replace(), or hash_delete(), yields
   undefined behavior, whether done in DESTRUCTOR or
   elsewhere. */
void
hash_destroy (struct hash *h, hash_action_func *destructor)
{
  if (destructor != NULL)
    hash_clear (h, destructor);
  free (h->buckets);   // array 처럼쓰던걸 할당해제.
}
```

※

```c
/* Hash element. */
struct hash_elem
  {
    struct list_elem list_elem;
  };
```

hash의 element로 사용할 구조체에 struct hash_elem을 넣어야함. → ⓐ를 얻기위해 사용.

```c
/* Converts pointer to hash element HASH_ELEM into a pointer to
   the structure that HASH_ELEM is embedded inside.  Supply the
   name of the outer structure STRUCT and the member name MEMBER
   of the hash element.  See the big comment at the top of the
   file for an example. */
#define hash_entry(HASH_ELEM, STRUCT, MEMBER)                   \
        ((STRUCT *) ((uint8_t *) &(HASH_ELEM)->list_elem        \
                     - offsetof (STRUCT, MEMBER.list_elem)))
```

buckets[i]의 list임. 원소 / ⓐ 이름. / ⓐ에서 hash_elem의 변수명

※

```c
/* Computes and returns the hash value for hash element E, given
   auxiliary data AUX. */
typedef unsigned hash_hash_func (const struct hash_elem *e, void *aux);

/* Compares the value of two hash elements A and B, given
   auxiliary data AUX.  Returns true if A is less than B, or
   false if A is greater than or equal to B. */
typedef bool hash_less_func (const struct hash_elem *a,
                             const struct hash_elem *b,
                             void *aux);
```

이거 정리해야함.

※

```c
/* Returns a hash of integer I. */
unsigned
hash_int (int i)
{
  return hash_bytes (&i, sizeof i);   // 4
}
```

```c
/* Fowler-Noll-Vo hash constants, for 32-bit word sizes. */
#define FNV_32_PRIME 16777619u
#define FNV_32_BASIS 2166136261u
```

```c
/* Returns a hash of the SIZE bytes in BUF. */
unsigned
hash_bytes (const void *buf_, size_t size)
{
  /* Fowler-Noll-Vo 32-bit hash, for bytes. */
  const unsigned char *buf = buf_;
  unsigned hash;

  ASSERT (buf != NULL);

  hash = FNV_32_BASIS;
  while (size-- > 0)
    hash = (hash * FNV_32_PRIME) ^ *buf++;

  return hash;
}
```

```c
/* Inserts NEW into hash table H and returns a null pointer, if
   no equal element is already in the table.
   If an equal element is already in the table, returns it
   without inserting NEW. */
struct hash_elem *
hash_insert (struct hash *h, struct hash_elem *new)
{
  struct list *bucket = find_bucket (h, new);
  struct hash_elem *old = find_elem (h, bucket, new);

  if (old == NULL)
    insert_elem (h, bucket, new);

  rehash (h);          // 오... NULL 리턴이 좋은거네.

  return old;
}
```

```c
/* Finds and returns an element equal to E in hash table H, or a
   null pointer if no equal element exists in the table. */
struct hash_elem *
hash_find (struct hash *h, struct hash_elem *e)
{
  return find_elem (h, find_bucket (h, e), e);
}
```

```c
/* Finds, removes, and returns an element equal to E in hash
   table H.  Returns a null pointer if no equal element existed
   in the table.

   If the elements of the hash table are dynamically allocated,
   or own resources that are, then it is the caller's
   responsibility to deallocate them. */
struct hash_elem *
hash_delete (struct hash *h, struct hash_elem *e)
{
  struct hash_elem *found = find_elem (h, find_bucket (h, e), e);
  if (found != NULL)
    {
      remove_elem (h, found);
      rehash (h);
    }
  return found;
}
```

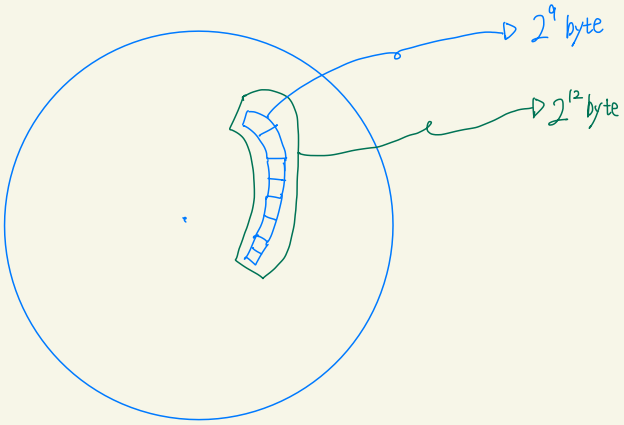
hash_elem 이 속한 bucket을 찾아줌.

```c
/* Returns the bucket in H that E belongs in. */
static struct list *
find_bucket (struct hash *h, struct hash_elem *e)
{
  size_t bucket_idx = h->hash (e, h->aux) & (h->bucket_cnt - 1);   // 이런 연산하면 나오나보다
  return &h->buckets[bucket_idx];
}
```

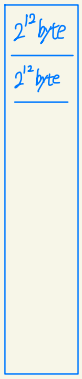
```c
/* Searches BUCKET in H for a hash element equal to E.  Returns
   it if found or a null pointer otherwise. */
static struct hash_elem *
find_elem (struct hash *h, struct list *bucket, struct hash_elem *e)
{
  struct list_elem *i;

  for (i = list_begin (bucket); i != list_end (bucket); i = list_next (i))
    {
      struct hash_elem *hi = list_elem_to_hash_elem (i);
      if (!h->less (hi, e, h->aux) && !h->less (e, hi, h->aux))
        return hi;
    }
  return NULL;   // 못찾으면 NULL
}
```

```c
/* Inserts E into BUCKET (in hash table H). */
static void
insert_elem (struct hash *h, struct list *bucket, struct hash_elem *e)
{
  h->elem_cnt++;
  list_push_front (bucket, &e->list_elem);
}

/* Removes E from hash table H. */
static void
remove_elem (struct hash *h, struct hash_elem *e)
{
  h->elem_cnt--;
  list_remove (&e->list_elem);
}
```

```c
/* Changes the number of buckets in hash table H to match the
   ideal.  This function can fail because of an out-of-memory
   condition, but that'll just make hash accesses less efficient;
   we can still continue. */
static void
rehash (struct hash *h)
{
  size_t old_bucket_cnt, new_bucket_cnt;
  struct list *new_buckets, *old_buckets;
  size_t i;

  ASSERT (h != NULL);

  /* Save old bucket info for later use. */
  old_buckets = h->buckets;
  old_bucket_cnt = h->bucket_cnt;

  /* Calculate the number of buckets to use now.
     We want one bucket for about every BEST_ELEMS_PER_BUCKET.
     We must have at least four buckets, and the number of
     buckets must be a power of 2. */
  new_bucket_cnt = h->elem_cnt / BEST_ELEMS_PER_BUCKET;
  if (new_bucket_cnt < 4)
    new_bucket_cnt = 4;
  while (!is_power_of_2 (new_bucket_cnt))
    new_bucket_cnt = turn_off_least_1bit (new_bucket_cnt);

  /* Don't do anything if the bucket count wouldn't change. */
  if (new_bucket_cnt == old_bucket_cnt)
    return;            // ↳ 아무것X.

  /* Allocate new buckets and initialize them as empty. */
  new_buckets = malloc (sizeof *new_buckets * new_bucket_cnt);
  if (new_buckets == NULL)
    {
      /* Allocation failed.  This means that use of the hash table will
         be less efficient.  However, it is still usable, so
         there's no reason for it to be an error. */
      return;
    }
  for (i = 0; i < new_bucket_cnt; i++)
    list_init (&new_buckets[i]);

  /* Install new bucket info. */
  h->buckets = new_buckets;
  h->bucket_cnt = new_bucket_cnt;

  /* Move each old element into the appropriate new bucket. */
  for (i = 0; i < old_bucket_cnt; i++)
    {
      struct list *old_bucket;
      struct list_elem *elem, *next;

      old_bucket = &old_buckets[i];
      for (elem = list_begin (old_bucket);
           elem != list_end (old_bucket); elem = next)
        {
          struct list *new_bucket
            = find_bucket (h, list_elem_to_hash_elem (elem));
          next = list_next (elem);
          list_remove (elem);
          list_push_front (new_bucket, elem);
        }
    }

  free (old_buckets);
}
```



disk



memory

```
pass tests/filesys/base/syn-write
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/sc-boundary-3
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-bound
pass tests/userprog/exec-bound-2
pass tests/userprog/exec-bound-3
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
```

```
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
FAIL tests/vm/pt-grow-stack
FAIL tests/vm/pt-grow-pusha
pass tests/vm/pt-grow-bad
FAIL tests/vm/pt-big-stk-obj
pass tests/vm/pt-bad-addr
pass tests/vm/pt-bad-read
pass tests/vm/pt-write-code
FAIL tests/vm/pt-write-code2
FAIL tests/vm/pt-grow-stk-sc
FAIL tests/vm/page-linear
FAIL tests/vm/page-parallel
FAIL tests/vm/page-merge-seq
FAIL tests/vm/page-merge-par
FAIL tests/vm/page-merge-stk
FAIL tests/vm/page-merge-mm
pass tests/vm/page-shuffle
FAIL tests/vm/mmap-read
FAIL tests/vm/mmap-close
pass tests/vm/mmap-unmap
FAIL tests/vm/mmap-overlap
FAIL tests/vm/mmap-twice
FAIL tests/vm/mmap-write
FAIL tests/vm/mmap-exit
FAIL tests/vm/mmap-shuffle
FAIL tests/vm/mmap-bad-fd
FAIL tests/vm/mmap-clean
FAIL tests/vm/mmap-inherit
FAIL tests/vm/mmap-misalign
FAIL tests/vm/mmap-null
FAIL tests/vm/mmap-over-code
FAIL tests/vm/mmap-over-data
FAIL tests/vm/mmap-over-stk
FAIL tests/vm/mmap-remove
pass tests/vm/mmap-zero
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
27 of 113 tests failed.
```