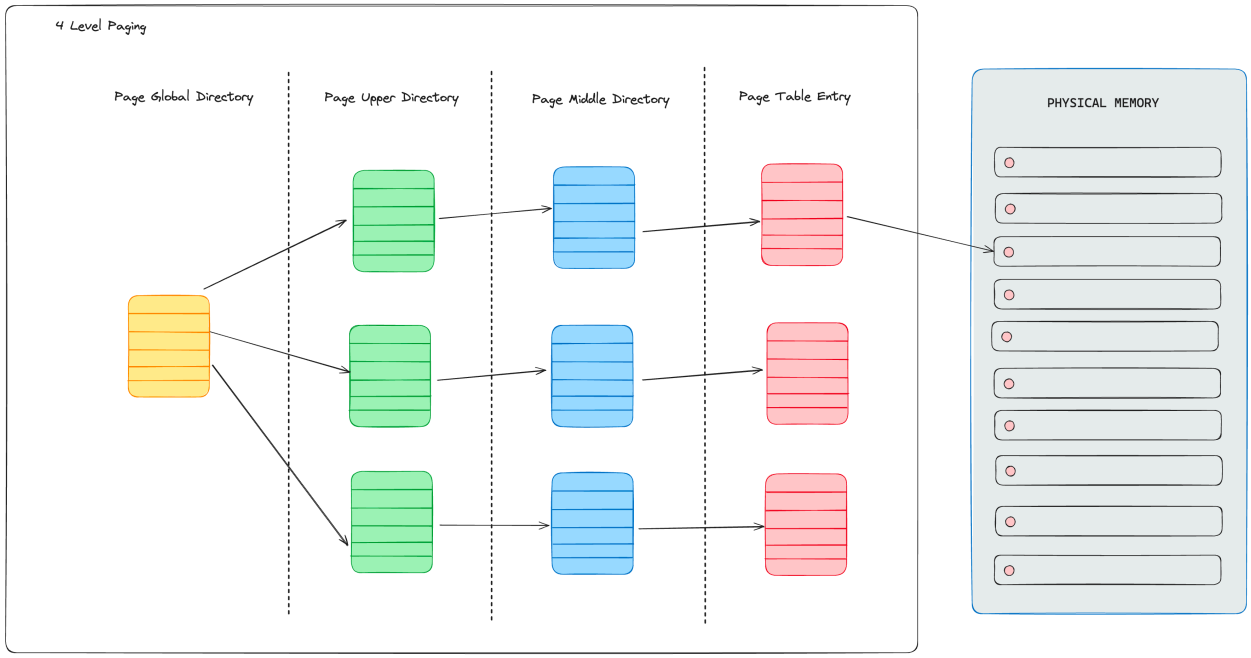
시스템프로그래밍 2 차 과제 사전 보고서

2017147581 서혁준

1. Linux 2.6.29 버전

linux 2.6.29 버전은 4-level paging 을 지원한다.



4-level paging 은 `x86_64`의 4-level paging 을 OS-level 에서 지원하기 위하여 `2.6.11` 버전 패치에서 `x86` 아키텍쳐에 대해 기존 3-level paging 에 `PUD`를 추가하는 방식으로 구현되었다.

따라서 해당 버전에서는 전체 아키텍쳐가 아닌 `x86`을 포함한 일부 아키텍쳐만 4-level paging 을 지원하고 있다.

4-level paging 에서는 Virtual Address 를 아래와 같은 과정을 통해 Physical Address 로 변환한다.
Virtual Address → `PGD` → `PUD` → `PMD` → `PTE` → Physical Address

해당 과정에서는 각각의 테이블 및 디렉토리의 엔트리를 나타나기 위해
`pgd_t`, `pud_t`, `pmd_t`, `pte_t` 와 같은 자료구조를 정의하여 사용하고 있다.

이제 Virtual Address 를 Physical Address 로 변환하는 과정을 살펴보자.
모든 코드는 `x86_64` 아키텍쳐 기준으로 조사하였다.

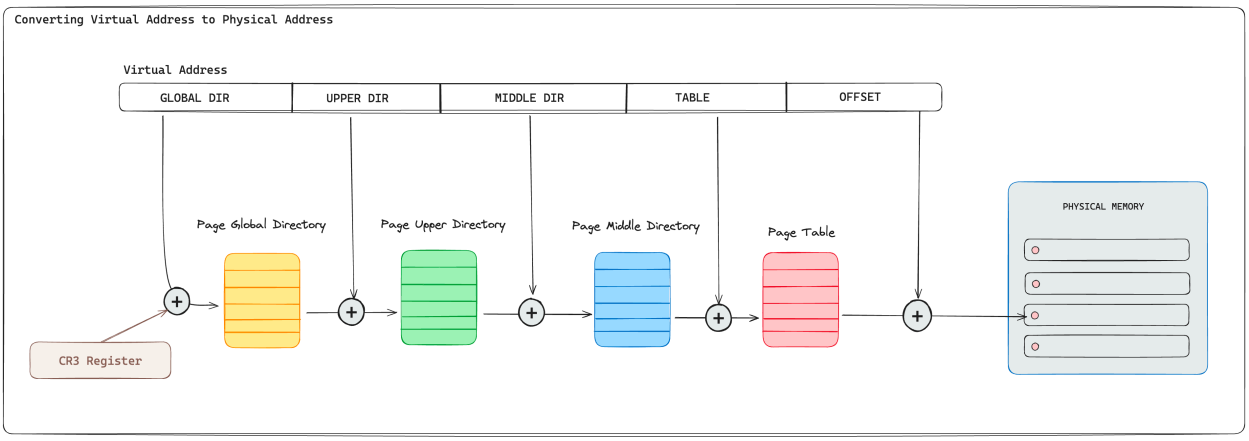
1.1 init_64.h

```
void set_pte_vaddr(unsigned long vaddr, pte_t pteval)
{
  pgd_t *pgd;
  pud_t *pud_page;

  pr_debug("set_pte_vaddr %lx to %lx\n", vaddr, native_pte_val(pteval));

  pgd = pgd_offset_k(vaddr);
  if (pgd_none(*pgd)) {
    printk(KERN_ERR
      "PGD FIXMAP MISSING, it should be setup in head.S!\n");
    return;
  }
  pud_page = (pud_t*)pgd_page_vaddr(*pgd);
  set_pte_vaddr_pud(pud_page, vaddr, pteval);
}
```

`set_pte_vaddr()` 함수는 virtual address 로부터 PTE 를 매핑하는 함수이다.

함수는 변환할 virtual address 를 나타내는 `vaddr` 와 해당 virtual address 에 따라 값을 세팅할 `pteval` 을 인자로 받는다.

내부 동작을 보면 `pgd_offset_k()` 함수를 통해 virtual address 에 해당하는 PGD 를 `pgd` 에 담는다.
다음으로 `pgd_page_vaddr()` 함수를 통해 해당 vaddress 에 대응되는 PUD 의 위치를 `pud_page`에 저장한다.
이후 `set_pte_vaddr_pud()` 함수를 통해 조금 더 낮은 레벨에서의 PTE 정보 파싱을 진행한다.



```
void set_pte_vaddr_pud(pud_t *pud_page, unsigned long vaddr, pte_t new_pte) {
  pud_t *pud;
  pmd_t *pmd;
  pte_t *pte;

  pud = pud_page + pud_index(vaddr);
  if (pud_none(*pud)) {
    pmd = (pmd_t *) spp_getpage();
    pud_populate(&init_mm, pud, pmd);
    if (pmd != pmd_offset(pud, 0)) {
      printk(KERN_ERR "PAGETABLE BUG #01! %p <-> %p\n",
        pmd, pmd_offset(pud, 0));
      return;
    }
  }
  pmd = pmd_offset(pud, vaddr);
  if (pmd_none(*pmd)) {
    pte = (pte_t *) spp_getpage();
    pmd_populate_kernel(&init_mm, pmd, pte);
    if (pte != pte_offset_kernel(pmd, 0)) {
      printk(KERN_ERR "PAGETABLE BUG #02!\n");
      return;
    }
  }

  pte = pte_offset_kernel(pmd, vaddr);
  set_pte(pte, new_pte);

  /*
   * It's enough to flush this one mapping.
   * (PGE mappings get flushed as well)
   */
  __flush_tlb_one(vaddr);
}
```

`set_pte_vaddr_pud()` 함수는 주어진 virtual address 와 PUD 를 가지고 PTE 값을 저장하는 함수이다.

여기에서는 PUD 에 대한 포인터인 `pud_t *pud_page` 를 추가적인 인자로 받는다.

내부 동작을 살펴보면,

- `pud = pud_page + pud_index(vaddr)`
  우선 주어진 virtual address 에 대해서 PUD index 를 구한다.
- `if(pud_none(*pud)) { ... }`
  PUD 인덱스에 대해서 pmd 가 존재하지 않으면 새로운 PMD 를 할당한다.

- `pmd = pmd_offset(pud, vaddr)`
  virtual address 와 PUD 를 이용하여 PMD index 를 구한다.
- `if(pmd_none(*pmd)) { ... }`
  해당 pmd 에 대한 PTE 가 존재하지 않으면 새로운 PTE 를 할당한다.
- `pte = pte_offset_kernel(pmd, vaddr); set_pte(pte, new_pte)`
  주어진 `vaddr` 에 대응되는 PTE 를 `new_pte`에 저장해준다.

정리해보자면, `init_64.c` 에 정의된 `set_pte_vaddr()` 과 `set_pte_vaddr_pud()` 는 4-level paging 을 이용해 virtual address 와 physical address 를 매핑하는 중추적인 역할을 하는 함수이다.

이제 이 과정에서 사용된 자료구조와 매크로 및 기타 함수들을 알아보자.

1.2 pgtable.h

```
/*
 * the pgd page can be thought of an array like this: pgd_t[PTRS_PER_PGD]
 *
 * this macro returns the index of the entry in the pgd page which would
 * control the given virtual address
 */
#define pgd_index(address) (((address) >> PGDIR_SHIFT) & (PTRS_PER_PGD - 1))
/*
 * pgd_offset() returns a (pgd_t *)
 * pgd_index() is used get the offset into the pgd page's array of pgd_t's;
 */
#define pgd_offset(mm, address) ((mm)->pgd + pgd_index((address)))
/*
 * a shortcut which implies the use of the kernel's pgd, instead
 * of a process's
 */
#define pgd_offset_k(address) pgd_offset(&init_mm, (address))
```

`init_64.c` 에서 사용되었던 `pgd_offset()`, `pud_offset()` 등 다양한 함수들 중에서 `pgd_offset()`만 `pgtable.h`에 정의되어 있는 것을 볼 수 있는데, 이는 2.6.29 버전이 배포되던 당시 4-level paging 을 사용하는 아키텍쳐가 많지 않았기 때문에 pgd 관련 함수만 `pgtable`에 정의한 것이라고 예상해 볼 수 있다.

각각의 구현을 확인해 보면 주어진 virtual address 에 정해진 `PGDIR_SHIFT` 값을 이용하여 SHIFT 연산을 수행함으로서 Page Global Directory 내에서 해당하는 인덱스를 Virtual Address 로부터 추출해 내고 있다.
그리고 `pgd_offset()` 함수에서 mm_struct 내에 저장되어 있는 pgd 에 virtual address 가 지정하는 index 를 추가하여 최종적으로 원하는 page directory 의 entry 를 가져올 수 있다.

1.3 pgtable_64.h

```
/*
 * Conversion functions: convert a page and protection to a page entry,
 * and a page entry and page directory to the page they refer to.
 */

/*
 * Level 4 access.
 */
#define pgd_page_vaddr(pgd)           \
  ((unsigned long)__va((unsigned long)pgd_val((pgd)) & PTE_PFN_MASK))
#define pgd_page(pgd)   (pfn_to_page(pgd_val((pgd)) >> PAGE_SHIFT))
#define pgd_present(pgd) (pgd_val(pgd) & _PAGE_PRESENT)
static inline int pgd_large(pgd_t pgd) { return 0; }
#define mk_kernel_pgd(address) __pgd((address) | _KERNPG_TABLE)

/* PUD - Level3 access */
/* to find an entry in a page-table-directory. */
#define pud_page_vaddr(pud)           \
  ((unsigned long)__va(pud_val((pud)) & PHYSICAL_PAGE_MASK))
#define pud_page(pud) (pfn_to_page(pud_val((pud)) >> PAGE_SHIFT))
#define pud_index(address) (((address) >> PUD_SHIFT) & (PTRS_PER_PUD - 1))
#define pud_offset(pgd, address)          \
  ((pud_t *)pgd_page_vaddr(*(pgd)) + pud_index((address)))
#define pud_present(pud) (pud_val((pud)) & _PAGE_PRESENT)

static inline int pud_large(pud_t pte)
{
  return (pud_val(pte) & (_PAGE_PSE | _PAGE_PRESENT)) ==
    (_PAGE_PSE | _PAGE_PRESENT);
}
```

`pgtable_64.h` 에는 4-level paging 에서 사용하는 각 자료구조에 대해서 offset 을 구하거나, index 를 구하거나 하는 데에 필요한 함수들이 정의되어 있다.

세부적인 구현 혹은 함수들의 조합은 `pgd` 관련 함수와 거의 유사함을 확인할 수 있다.

```
/* PMD  - Level 2 access */
#define pmd_page_vaddr(pmd) ((unsigned long) __va(pmd_val((pmd)) & PTE_PFN_MASK))
#define pmd_page(pmd)   (pfn_to_page(pmd_val((pmd)) >> PAGE_SHIFT))

#define pmd_index(address) (((address) >> PMD_SHIFT) & (PTRS_PER_PMD - 1))
#define pmd_offset(dir, address) ((pmd_t *)pud_page_vaddr(*(dir)) + \
          pmd_index(address))
#define pmd_none(x) (!pmd_val((x)))
#define pmd_present(x)  (pmd_val((x)) & _PAGE_PRESENT)
#define pfn_pmd(nr, prot) (__pmd(((nr) << PAGE_SHIFT) | pgprot_val((prot))))
#define pmd_pfn(x)  ((pmd_val((x)) & __PHYSICAL_MASK) >> PAGE_SHIFT)

#define pte_to_pgoff(pte) ((pte_val((pte)) & PHYSICAL_PAGE_MASK) >> PAGE_SHIFT)
#define pgoff_to_pte(off) ((pte_t) { .pte = ((off) << PAGE_SHIFT) | \
              _PAGE_FILE })
#define PTE_FILE_MAX_BITS __PHYSICAL_MASK_SHIFT

/* PTE - Level 1 access. */

/* page, protection -> pte */
#define mk_pte(page, pgprot)  pfn_pte(page_to_pfn((page)), (pgprot))

#define pte_index(address) (((address) >> PAGE_SHIFT) & (PTRS_PER_PTE - 1))
#define pte_offset_kernel(dir, address) ((pte_t *) pmd_page_vaddr(*(dir)) + \
           pte_index((address)))
```

해당 함수들의 정의를 보면 여러 MACRO 들이 정의되고 사용되고 있는 것을 확인할 수 있다.

- `PAGE_SHIFT`
- `PUD_SHIFT`
- `PMD_SHIFT`

...

1.4 page.h

마지막으로, PGD, PUD, PMD, PTE 의 위치를 가져오는 데에 사용되는 `pgd_val()`, `pud_val()` 등의 함수가 정의되어 있는 `page.h` 파일을 살펴보자.

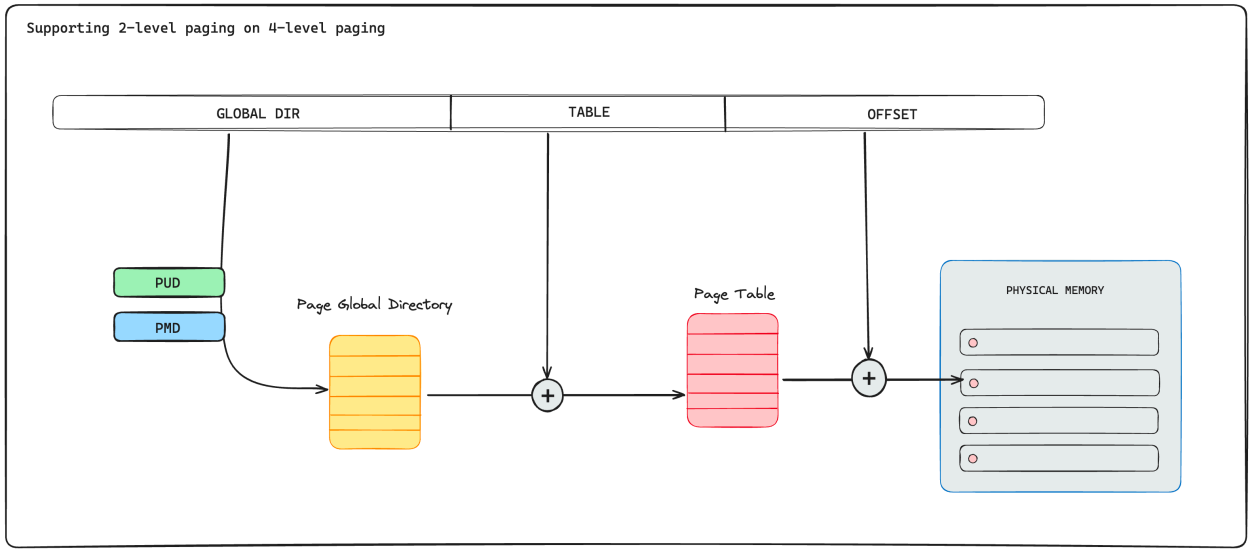
```
#define pgd_val(x)  native_pgd_val(x)
#define __pgd(x)  native_make_pgd(x)

#ifndef __PAGETABLE_PUD_FOLDED
#define pud_val(x)  native_pud_val(x)
#define __pud(x)  native_make_pud(x)
#endif

#ifndef __PAGETABLE_PMD_FOLDED
#define pmd_val(x)  native_pmd_val(x)
#define __pmd(x)  native_make_pmd(x)
#endif

#define pte_val(x)  native_pte_val(x)
#define pte_flags(x)  native_pte_flags(x)
#define __pte(x)  native_make_pte(x)
```

각각의 함수들이 native~ 접두사가 붙은 함수를 가리키는 것을 보아 아키텍쳐에 따라 정의를 다르게 할 수 있도록 모듈화 해 둔 것을 확인할 수 있다.

```
static inline pgd_t native_make_pgd(pgdval_t val)
{
  return (pgd_t) { val };
}

static inline pgdval_t native_pgd_val(pgd_t pgd)
{
  return pgd.pgd;
}

#if PAGETABLE_LEVELS >= 3
#if PAGETABLE_LEVELS == 4
typedef struct { pudval_t pud; } pud_t;

static inline pud_t native_make_pud(pmdval_t val)
{
  return (pud_t) { val };
}
```

```
static inline pudval_t native_pud_val(pud_t pud)
{
  return pud.pud;
}
#else /* PAGETABLE_LEVELS == 3 */
#include <asm-generic/pgtable-nopud.h>
static inline pudval_t native_pud_val(pud_t pud)
{
  return native_pgd_val(pud.pgd);
}
#endif  /* PAGETABLE_LEVELS == 4 */
typedef struct { pmdval_t pmd; } pmd_t;
static inline pmd_t native_make_pmd(pmdval_t val)
{
  return (pmd_t) { val };
}
static inline pmdval_t native_pmd_val(pmd_t pmd)
{
  return pmd.pmd;
}
#else  /* PAGETABLE_LEVELS == 2 */
#include <asm-generic/pgtable-nopmd.h>
static inline pmdval_t native_pmd_val(pmd_t pmd)
{
  return native_pgd_val(pmd.pud.pgd);
}
#endif  /* PAGETABLE_LEVELS >= 3 */
static inline pte_t native_make_pte(pteval_t val)
{
  return (pte_t) { .pte = val };
}
static inline pteval_t native_pte_val(pte_t pte)
{
  return pte.pte;
}
static inline pteval_t native_pte_flags(pte_t pte)
{
  return native_pte_val(pte) & PTE_FLAGS_MASK;
}
```

여기에서 우리가 주목할 점은 `PAGETABLE_LEVELS` 에 따라서 다른 함수가 정의에 매핑되도록 하여, 4-level paging 을 지원하면서도 하위 레벨 페이징이 가능하도록 해 둔 점이다.



좋은 예시로, `PAGETABLE_LEVELS == 2` 인 경우를 보자.

```
#else  /* PAGETABLE_LEVELS == 2 */
#include <asm-generic/pgtable-nopmd.h>

static inline pmdval_t native_pmd_val(pmd_t pmd)
{
  return native_pgd_val(pmd.pud.pgd);
}
```

이 경우 `pmd_val()` 함수에서 PGD 의 위치를 반환함으로서 `pgd`와 `pud`를 건너뛰고 바로 PMD 가 PGD 의 실제 위치를 가리키도록 하고 있다.

이를 통해 우리가 기존에 알고 있었던 동작과 동일하게 코드가 구현되어 있다는 것을 확인할 수 있다.

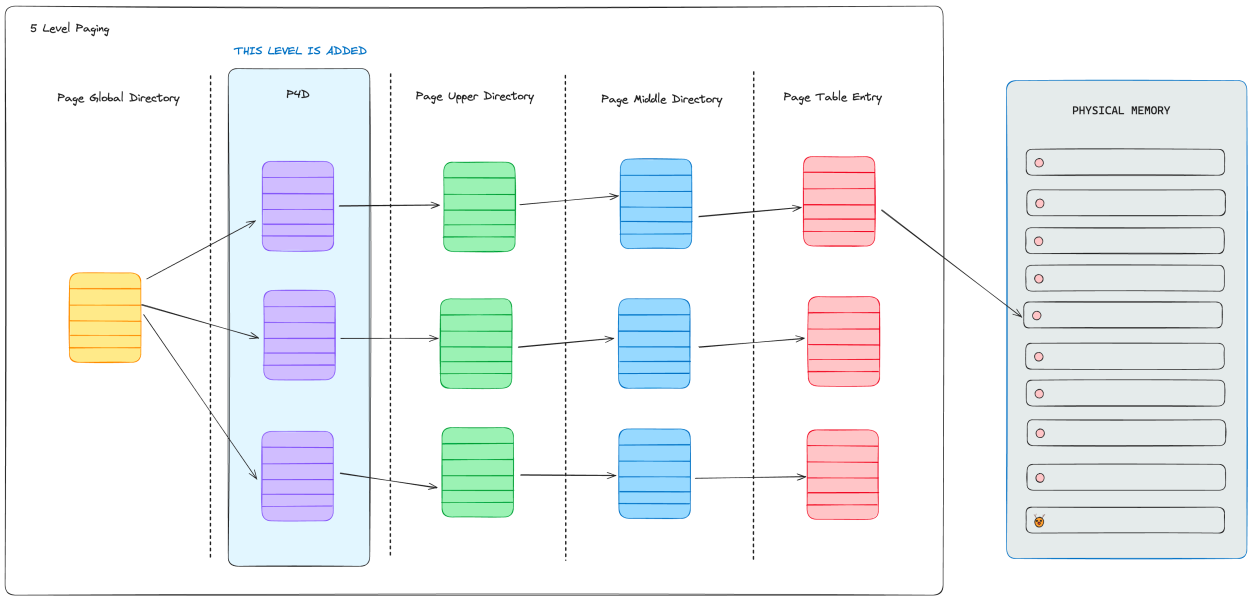
2. Linux 6.2

`2.6.29` 버전과 `6.2` 버전의 가장 큰 차이는 `6.2` 버전에서는 5-level paging 을 지원한다는 점이다.

이전 4-level paging 관련 리눅스 커뮤니티의 논의를 살펴보면

> After various deductions, that is sufficient to implement a 128TB address space,
> which should last for a little while.

라는 말을 볼 수 있는데, 이로부터 약 12 년이 지난 후, 128TB 이상의 주소 공간을 지원할 필요가 생기면서 5-level paging 기능이 도입되게 되었다.



5-level paging 은 `PGD` 와 `PUD` 사이에 `P4D` 라고 하는 새로운 레벨을 추가하는 방식으로 구현되었다. 이는 4-level paging 을 구현하는 과정에서 `PGD` 위에 레벨을 추가할 것인지, 아니면 `PGD` 아래에 새로운 레벨을 추가할지에 대해 논의하여 최종적으로 `PGD` 아래에 새로운 레벨을 추가하는 방식으로 구현된 것을 참고한 것으로 보인다.

당시의 논의를 살펴보면, 테이블을 중간에 추가함으로서 실제 페이지 테이블을 탐색하는 코드에 대한 수정만 하면 되도록 하여 페이지 테이블 위계를 확장하는 데에 있어 최소한의 변경만 가능하도록 하였다.

> Others are in less of a hurry, however, and see merit in Nick's patches. In particular, Linus prefers placing the new level below the PGD as the least intrusive way of extending the page table hierarchy.
>
> Basically, by doing the new folded table in the middle, it _only_ affects code that actually walks the page tables. Basically, what I wanted in the original 2->3 level expansion was that people who don't use the new level should be able to conceptually totally ignore it. I think that is even more true in the 3->4 level expansion.

2.1 init_64.c

```
static void __set_pte_vaddr(pud_t *pud, unsigned long vaddr, pte_t new_pte) {
  pmd_t *pmd = fill_pmd(pud, vaddr);
  pte_t *pte = fill_pte(pmd, vaddr);

  set_pte(pte, new_pte);

  /*
   * It's enough to flush this one mapping.
   * (PGE mappings get flushed as well)
   */
  flush_tlb_one_kernel(vaddr);
}
void set_pte_vaddr_p4d(p4d_t *p4d_page, unsigned long vaddr, pte_t new_pte)
{
  p4d_t *p4d = p4d_page + p4d_index(vaddr);
  pud_t *pud = fill_pud(p4d, vaddr);

  __set_pte_vaddr(pud, vaddr, new_pte);
}
void set_pte_vaddr_pud(pud_t *pud_page, unsigned long vaddr, pte_t new_pte)
{
  pud_t *pud = pud_page + pud_index(vaddr);

  __set_pte_vaddr(pud, vaddr, new_pte);
}
void set_pte_vaddr(unsigned long vaddr, pte_t pteval)
{
  pgd_t *pgd;
  p4d_t *p4d_page;

  pr_debug("set_pte_vaddr %lx to %lx\n", vaddr, native_pte_val(pteval));

  pgd = pgd_offset_k(vaddr);
  if (pgd_none(*pgd)) {
    printk(KERN_ERR
      "PGD FIXMAP MISSING, it should be setup in head.S!\n");
    return;
  }

  p4d_page = p4d_offset(pgd, 0);
  set_pte_vaddr_p4d(p4d_page, vaddr, pteval);
}
```

```
pmd_t * __init populate_extra_pmd(unsigned long vaddr) {
  pgd_t *pgd;
  p4d_t *p4d;
  pud_t *pud;

  pgd = pgd_offset_k(vaddr);
  p4d = fill_p4d(pgd, vaddr);
  pud = fill_pud(p4d, vaddr);
  return fill_pmd(pud, vaddr);
}

pte_t * __init populate_extra_pte(unsigned long vaddr) {
  pmd_t *pmd;

  pmd = populate_extra_pmd(vaddr);
  return fill_pte(pmd, vaddr);
}
```

기본적으로 `vaddr` 를 이용하여 `pte` 의 값을 세팅하는 함수의 구조는 동일하나,
5-level paging 을 지우너하면서 `p4d`를 이용하는 함수를 추가적으로 구현한 것을 확인할 수 있다.

또한 각 `set_pte_vaddr_()` 함수 내에 바로 테이블 엔트리를 채우는 비즈니스 로직이 구현되어 있던 2.6.29 버전과는 다르게 그러한 로직이 `fill_` 접두사를 붙인 함수로 이동하여 구현된 것을 확인할 수 있다.

```
static p4d_t *fill_p4d(pgd_t *pgd, unsigned long vaddr) {
  if (pgd_none(*pgd)) {
    p4d_t *p4d = (p4d_t *)spp_getpage();
    pgd_populate(&init_mm, pgd, p4d);
    if (p4d != p4d_offset(pgd, 0))
      printk(KERN_ERR "PAGETABLE BUG #00! %p <-> %p\n",
             p4d, p4d_offset(pgd, 0));
  }
  return p4d_offset(pgd, vaddr);
}

static pud_t *fill_pud(p4d_t *p4d, unsigned long vaddr) {
  if (p4d_none(*p4d)) {
    pud_t *pud = (pud_t *)spp_getpage();
    p4d_populate(&init_mm, p4d, pud);
    if (pud != pud_offset(p4d, 0))
      printk(KERN_ERR "PAGETABLE BUG #01! %p <-> %p\n",
             pud, pud_offset(p4d, 0));
  }
  return pud_offset(p4d, vaddr);
}
```

```
static pmd_t *fill_pmd(pud_t *pud, unsigned long vaddr)
{
  if (pud_none(*pud)) {
    pmd_t *pmd = (pmd_t *) spp_getpage();
    pud_populate(&init_mm, pud, pmd);
    if (pmd != pmd_offset(pud, 0))
      printk(KERN_ERR "PAGETABLE BUG #02! %p <-> %p\n",
             pmd, pmd_offset(pud, 0));
  }
  return pmd_offset(pud, vaddr);
}

static pte_t *fill_pte(pmd_t *pmd, unsigned long vaddr)
{
  if (pmd_none(*pmd)) {
    pte_t *pte = (pte_t *) spp_getpage();
    pmd_populate_kernel(&init_mm, pmd, pte);
    if (pte != pte_offset_kernel(pmd, 0))
      printk(KERN_ERR "PAGETABLE BUG #03!\n");
  }
  return pte_offset_kernel(pmd, vaddr);
}
```

테이블 혹은 디렉토리 엔트리를 채우는 함수들이 대부분 동일한 구조를 하고 있으므로 `fill_pmd()` 를 예시로 구현을 살펴보면, `pud`가 존재하는 지 먼저 확인하고 주어진 Page Upper Directory 가 가리키는 값에 virtual address 를 이용한 Page Middle Directory 의 오프셋을 활용하여 pmd 를 찾아서 반환하고 있다.

2.2 pgtable.h

```
#define pgd_val(x)  native_pgd_val(x)
#define __pgd(x)  native_make_pgd(x)

#ifndef __PAGETABLE_P4D_FOLDED
#define p4d_val(x)  native_p4d_val(x)
#define __p4d(x)  native_make_p4d(x)
#endif

#ifndef __PAGETABLE_PUD_FOLDED
#define pud_val(x)  native_pud_val(x)
#define __pud(x)  native_make_pud(x)
#endif

#ifndef __PAGETABLE_PMD_FOLDED
#define pmd_val(x)  native_pmd_val(x)
#define __pmd(x)  native_make_pmd(x)
#endif

#define pte_val(x)  native_pte_val(x)
#define __pte(x)  native_make_pte(x)
```

`2.6`과 동일한 함수들이지만, p4d 에 대한 함수들이 추가된 것을 확인할 수 있다.

```
/* to find an entry in a page-table-directory. */
static inline p4d_t *p4d_offset(pgd_t *pgd, unsigned long address) {
  if (!pgtable_l5_enabled())
    return (p4d_t *)pgd;
  return (p4d_t *)pgd_page_vaddr(*pgd) + p4d_index(address);
}

/* Find an entry in the second-level page table.. */
static inline pmd_t *pmd_offset(pud_t *pud, unsigned long address) {
  return pud_pgtable(*pud) + pmd_index(address);
}

static inline pud_t *pud_offset(p4d_t *p4d, unsigned long address) {
  return p4d_pgtable(*p4d) + pud_index(address);
}

static inline pgd_t *pgd_offset_pgd(pgd_t *pgd, unsigned long address) {
  return (pgd + pgd_index(address));
};

/*
 * a shortcut to get a pgd_t in a given mm
 */
#define pgd_offset(mm, address)   pgd_offset_pgd((mm)->pgd, (address))
```

기본적으로 `pgd_offset()` 등의 오프셋을 구하는 함수의 구현 형태는 리눅스 `2.6.29` 와 크게 달라지지 않았다. 다만 한 가지 주목해볼만한 점은, `2.6.29` 버전이 배포될 당시에는 4-level 아키텍쳐를 지원하는 프로세서가 많지 않아 해당 코드들이 `arch/x86/include/asm/pgtable_64.h` 에 구현되어 있었다면 리눅스 `6.2` 에서는 `/include/linux/pgtable.h` 에 구현되어 있다는 점이다.

2.3 pgtable_types.h

```
static inline pgd_t native_make_pgd(pgdval_t val)
{
  return (pgd_t) { val & PGD_ALLOWED_BITS };
}

static inline pgdval_t native_pgd_val(pgd_t pgd)
{
  return pgd.pgd & PGD_ALLOWED_BITS;
}

static inline pgdval_t pgd_flags(pgd_t pgd)
{
  return native_pgd_val(pgd) & PTE_FLAGS_MASK;
}

#if CONFIG_PGTABLE_LEVELS > 4
typedef struct { p4dval_t p4d; } p4d_t;

static inline p4d_t native_make_p4d(pudval_t val)
{
  return (p4d_t) { val };
}

static inline p4dval_t native_p4d_val(p4d_t p4d)
{
  return p4d.p4d;
}
#else
#include <asm-generic/pgtable-nop4d.h>

static inline p4d_t native_make_p4d(pudval_t val)
{
  return (p4d_t) { .pgd = native_make_pgd((pgdval_t)val) };
}

static inline p4dval_t native_p4d_val(p4d_t p4d)
{
  return native_pgd_val(p4d.pgd);
}
#endif

#if CONFIG_PGTABLE_LEVELS > 3
typedef struct { pudval_t pud; } pud_t;
```

```
static inline pud_t native_make_pud(pmdval_t val)
{
  return (pud_t) { val };
}

static inline pudval_t native_pud_val(pud_t pud)
{
  return pud.pud;
}
#else
#include <asm-generic/pgtable-nopud.h>

static inline pud_t native_make_pud(pudval_t val)
{
  return (pud_t) { .p4d.pgd = native_make_pgd(val) };
}

static inline pudval_t native_pud_val(pud_t pud)
{
  return native_pgd_val(pud.p4d.pgd);
}
#endif

#if CONFIG_PGTABLE_LEVELS > 2
static inline pmd_t native_make_pmd(pmdval_t val)
{
  return (pmd_t) { .pmd = val };
}

static inline pmdval_t native_pmd_val(pmd_t pmd)
{
  return pmd.pmd;
}
#else
#include <asm-generic/pgtable-nopmd.h>

static inline pmd_t native_make_pmd(pmdval_t val)
{
  return (pmd_t) { .pud.p4d.pgd = native_make_pgd(val) };
}

static inline pmdval_t native_pmd_val(pmd_t pmd)
{
  return native_pgd_val(pmd.pud.p4d.pgd);
}
```

`2.6` 때와 동일하게 `CONFIG_PGTABLE_LEVELS` 값에 따라서 `p4d_val`, `pud_val`, `pmd_val` 이 가리키는 값이 달라지도록 구현되어 있는 것을 확인할 수 있다.

3. References

- Four-level page tables, https://lwn.net/Articles/106177/
- Four-level page table merged, https://lwn.net/Articles/117749/
- Five-level page tables, https://lwn.net/Articles/717293/
- Bootlin, https://elixir.bootlin.com/linux/latest/source

시스템프로그래밍 2 차 과제 실습 보고서

2017147581 서혁준

1. 과제 수행 환경

1.1 OS 정보

```
antares@ubuntu:~/Documents/SystemProgramming2023/Assignment2/hw2_2017147581$ uname -a
Linux ubuntu 6.2.0-2017147581+ #6 SMP Sat Oct 14 09:17:16 UTC 2023 aarch64 aarch64 aarch64 GNU/Linux
```

1 차 과제를 진행하였던 커널을 그대로 사용하였다.

1.2 메모리 환경

```
antares@ubuntu:~/Documents/SystemProgramming2023/Assignment2/hw2_2017147581$ cat /proc/meminfo
MemTotal:        8125320 kB
MemFree:         5639940 kB
MemAvailable:    7018740 kB
Buffers:           79848 kB
Cached:          1456940 kB
SwapCached:            0 kB
Active:           550088 kB
Inactive:        1599664 kB
Active(anon):       1948 kB
Inactive(anon):   660088 kB
Active(file):     548140 kB
Inactive(file):   939576 kB
Unevictable:       26192 kB
Mlocked:           26192 kB
SwapTotal:       4004860 kB
SwapFree:        4004860 kB
```

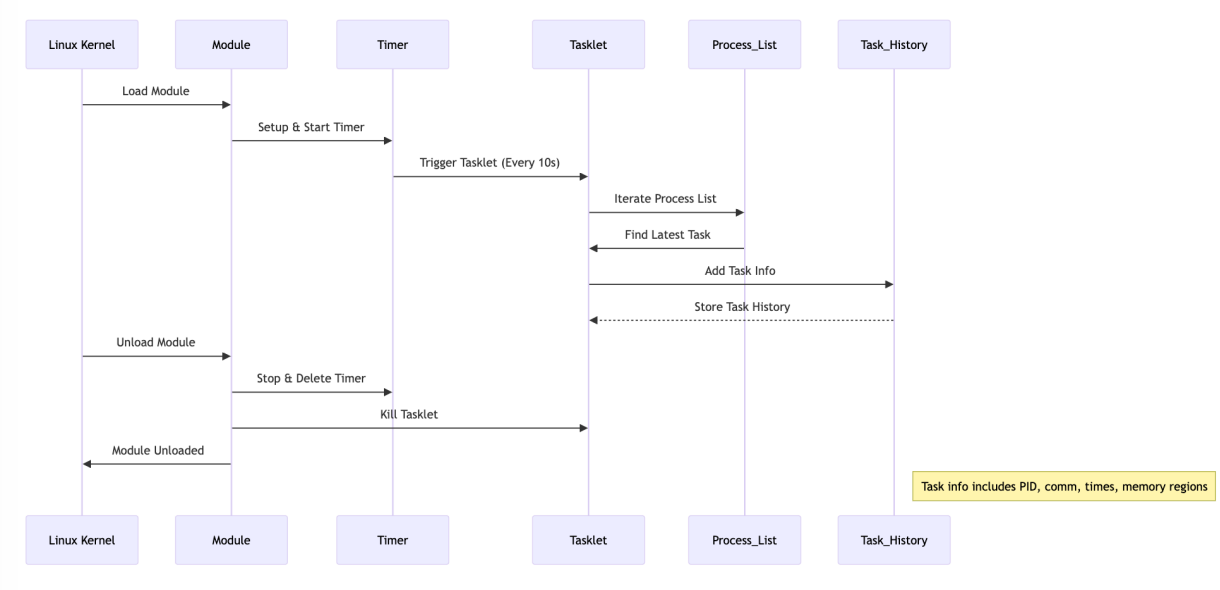
1.2 CPU 환경

```
antares@ubuntu:~/Documents/SystemProgramming2023/Assignment2/hw2_2017147581$ lscpu
Architecture:            aarch64
  CPU op-mode(s):        64-bit
  Byte Order:            Little Endian
CPU(s):                  4
  On-line CPU(s) list:   0-3
Vendor ID:               Apple
  Model:                 0
  Thread(s) per core:    1
  Core(s) per socket:    4
  Socket(s):             1
  Stepping:              0x0
  BogoMIPS:              48.00
```

```
antares@ubuntu:~/Documents/SystemProgramming2023/Assignment2/hw2_2017147581$ cat /proc/cpuinfo
processor       : 0
BogoMIPS        : 48.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm j
ca pacg dcpodp flagm2 frint
CPU implementer : 0x61
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0x000
CPU revision    : 0

processor       : 1
BogoMIPS        : 48.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm j
ca pacg dcpodp flagm2 frint
CPU implementer : 0x61
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0x000
CPU revision    : 0

processor       : 2
BogoMIPS        : 48.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm j
ca pacg dcpodp flagm2 frint
CPU implementer : 0x61
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0x000
CPU revision    : 0

processor       : 3
BogoMIPS        : 48.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm j
ca pacg dcpodp flagm2 frint
CPU implementer : 0x61
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0x000
CPU revision    : 0
```

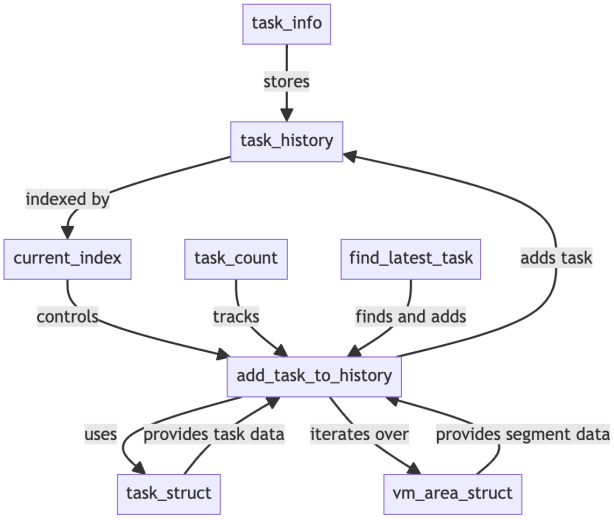
2. 커널 모듈 작성 보고서

2.1 프로그램 동작 구조



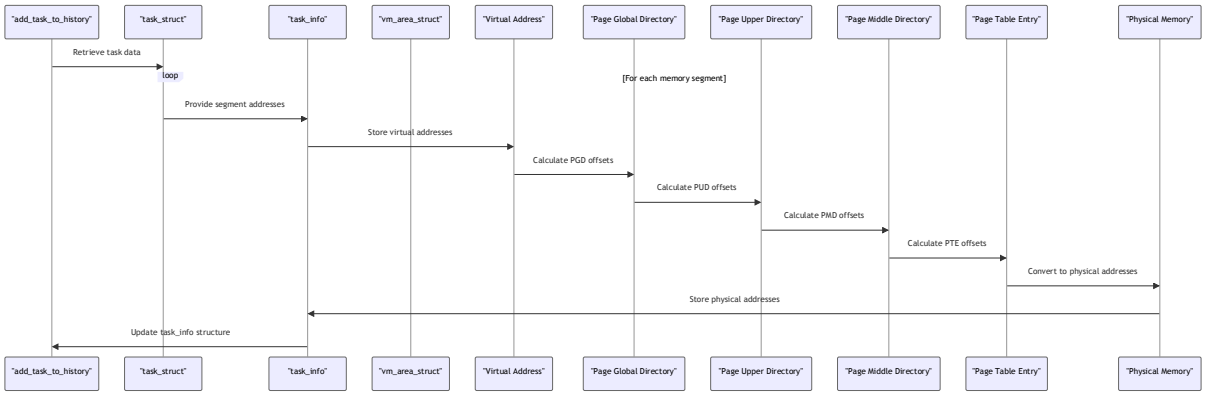
이번 모듈은 Timer 와 Tasklet 을 사용한다. 모듈을 등록하게 되면 10 초마다 Tasklet 을 실행시키는 콜백을 타이머에 등록한다.
이 Tasklet 은 `find_latest_task()` 함수를 실행시키며, `find_latest_task()`는 가장 최근 실행된 태스크를 찾아 이에 대한 정보를 task_history 에 저장시키는 `add_task_history()` 함수를 호출한다.

2.2 주요 로직



이러한 구조의 관계는 위 다이어그램을 통해서 좀 더 명확하게 확인할 수 있다.
Task 의 virtual/physical address 를 저장할 수 있도록 정의된 task_info 구조체는 Task_history 라는 배열으로 정의되어 있다.
find_latest_task 를 통해서 가장 최근 실행된 task 를 add_task_history 함수에 전달하면 add_task_history 는 task_struct 와 vm_area_struct 로부터 해당 태스크의 정보 및 세그먼트 정보를 받아와 이를 task_history 에 추가한다.

2.3 add_task_to_history() 동작 과정



`add_task_to_history()` 함수는 VMA 로부터 받아온 Segment 의 virtual address 정보를 PGD, PUD, PMD, PTE 가 제공하는 매크로들을 이용하여 물리 주소로 변환하여 task_info 에 저장한다.

3. 실습 과제 결과 검증 및 분석

3.1 모듈 등록 및 해제

```
antares@ubuntu:~/Documents/SystemProgramming2023/Assignment2/hw2_2017147581$ sudo insmod hw2.ko
antares@ubuntu:~/Documents/SystemProgramming2023/Assignment2/hw2_2017147581$ cat /proc/hw2 > output.log
antares@ubuntu:~/Documents/SystemProgramming2023/Assignment2/hw2_2017147581$ sudo rmmod hw2
antares@ubuntu:~/Documents/SystemProgramming2023/Assignment2/hw2_2017147581$
```

정상적으로 모듈이 등록되고 해제되는 것을 확인할 수 있다.

3.2 커널 정보 출력

```
antares@ubuntu:~/Documents/SystemProgramming2023/Assignment2/hw2_2017147581$ cat /proc/hw2
[System Programming Assignment #2]
ID : 2017147581
Name: Seo, Hyeokjun
Uptime (s): 1370
------------------------------------------------
[Trace #0]
Uptime (s): 4294966996
Command: sudo
PID: 3449
Start time (s): 1667
PGD base address: 00000000dca9e457
Code Area
- start (virtual): 0xaaaacf500000
- start (PGD): 0xffff000093cf6aa8, 0xffff000093cf6aa8
- start (PUD): 0xffff000095554558, 0xffff000095554558
- start (PMD): 0xffff00008cfe33d0, 0xffff00008cfe33d0
- start (PTE): 0xffff0000904ef800, 0xffff0000904ef800
- start (physical): 0x12aad34d00000
- end (virtual): 0xaaaacf5319d4
- end (PGD): 0xffff000093cf6aa8, 0xffff000093cf6aa8
- end (PUD): 0xffff000095554558, 0xffff000095554558
- end (PMD): 0xffff00008cfe33d0, 0xffff00008cfe33d0
- end (PTE): 0xffff0000904ef988, 0xffff0000904ef988
- end (physical): 0x12aad34d319d4
Data Area
- start (virtual): 0xaaaacf541b30
- start (PGD): 0xffff000093cf6aa8, 0xffff000093cf6aa8
- start (PUD): 0xffff000095554558, 0xffff000095554558
- start (PMD): 0xffff00008cfe33d0, 0xffff00008cfe33d0
- start (PTE): 0xffff0000904efa08, 0xffff0000904efa08
- start (physical): 0x12aad34d41b30
- end (virtual): 0xaaaacf5442f8
- end (PGD): 0xffff000093cf6aa8, 0xffff000093cf6aa8
- end (PUD): 0xffff000095554558, 0xffff000095554558
- end (PMD): 0xffff00008cfe33d0, 0xffff00008cfe33d0
- end (PTE): 0xffff0000904efa20, 0xffff0000904efa20
- end (physical): 0x12aad34d442f8
Heap Area
- start (virtual): 0xaaaae05ca000
- start (PGD): 0xffff000093cf6aa8, 0xffff000093cf6aa8
- start (PUD): 0xffff000095554558, 0xffff000095554558
- start (PMD): 0xffff00008cfe3810, 0xffff00008cfe3810
- start (PTE): 0xffff000089280e50, 0xffff000089280e50
- start (physical): 0x12aad45dca000
- end (virtual): 0xaaaae0650000
```

현재 커널에서 실행중인 프로세스에 대해서 Virtual Address, Physical Address, 그리고 중간의 PGD, PUD, PMD, PTE 등의 값을 요구사항대로 출력하고 있다.

3.3 프로그램 동작 검증

```
antares@ubuntu:~/Documents/SystemProgramming2023/Assignment2/hw2_2017147581/module$ cat /proc/hw2 | grep 'time\|Trace'
Uptime (s): 4530
[Trace #0]
Uptime (s): 49
Start time (s): 4437
[Trace #1]
Uptime (s): 59
Start time (s): 4437
[Trace #2]
Uptime (s): 69
Start time (s): 4437
[Trace #3]
Uptime (s): 79
Start time (s): 4437
[Trace #4]
Uptime (s): 90
Start time (s): 4437
```

출력 결과에서 Uptime 및 Start Time 정보만 추출한 결과는 아래와 같다.
실행중인 프로세스의 Uptime 및 Start time 을 약 10 초 간격으로, 오름차순으로 잘 출력하고 있다.

```
antares@ubuntu:~/Documents/SystemProgramming2023/Assignment2/hw2_2017147581$ cat /proc/hw2 | grep 'Trace\|Area\|virtual\|physical'
[Trace #0]
Code Area
- start (virtual): 0xaaaadf570000
- start (physical): 0x12aad44d70000
- end (virtual): 0xaaaadf6bc480
- end (physical): 0x12aad44ebc480
Data Area
- start (virtual): 0xaaaadf6cc818
- start (physical): 0x12aad44ecc818
- end (virtual): 0xaaaadf6d9280
- end (physical): 0x12aad44ed9280
Heap Area
- start (virtual): 0xaaaaeb526000
- start (physical): 0x12aad50d26000
- end (virtual): 0xaaaaeb6ce000
- end (physical): 0x12aad50ece000
Stack Area
- start (virtual): 0xffffe805b50
- start (physical): 0x1800264005b50
- end (virtual): 0xffffe807000
- end (physical): 0x1800264007000
[Trace #1]
Code Area
- start (virtual): 0xaaaadf570000
- start (physical): 0x12aad44d70000
- end (virtual): 0xaaaadf6bc480
- end (physical): 0x12aad44ebc480
Data Area
- start (virtual): 0xaaaadf6cc818
- start (physical): 0x12aad44ecc818
- end (virtual): 0xaaaadf6d9280
- end (physical): 0x12aad44ed9280
Heap Area
- start (virtual): 0xaaaaeb526000
- start (physical): 0x12aad50d26000
- end (virtual): 0xaaaaeb6ce000
- end (physical): 0x12aad50ece000
Stack Area
- start (virtual): 0xffffe805b50
- start (physical): 0x1800264005b50
- end (virtual): 0xffffe807000
- end (physical): 0x1800264007000
```

위 결과는 각 영역의 virtual / physical address 정보만 추출한 결과이다.

4. 애로사항

4.1 VMA 접근

리눅스 6.1 버전에서 VMA 관리 방식이 기존의 자료구조를 사용하는 것에서 maple_tree 라는, 리눅스에서 자체적으로 새로 구현한 자료구조를 사용하는 것으로 바뀌면서 관련 프로세스에 접근하는 데에 있어 어려움이 있었다.

구글링을 통해 maple_tree 의 사용법을 찾아 보았으나 잘 나오지 않아, `/lib/test_maple_tree.c` 를 통해 maple_tree 자료구조에 대한 사용 예시를 확인하고 이를 사용하였다.

5. References

- https://hyeyoo.com/175
- https://elixir.bootlin.com/linux/v6.2/source/lib/test_maple_tree.c#L446