시스템 프로그래밍 과제 보고서

2014122011 응용통계학과 신은수

* 보고서의 모든 내용은 x86-64 아키텍쳐와 리눅스 커널 소스 4.9.50 기준으로 작성됨

사전 조사 보고서

1. 버디시스템

   버디시스템은 커널이 메모리를 관리하는 한 가지 방법이다. 메모리는 크게 3개의 zone으로 나눌 수 있는데 0 – 16MB가 DMA zone, 16MB – 896MB가 normal zone, 896MB 이상의 메모리는 highmem zone이다. 각 zone 별로 메모리를 관리하는 2 가지 방법이 있는데 그 중 하나가 버디시스템이다. 버디시스템은 연속한 물리메모리를 할당 받을 수 있게 해준다.

   A. 메모리 할당 과정

      alloc_pages, alloc_page, get_zeroed_page 등의 함수를 통해서 페이지를 할당 받을 수 있는데 이 함수들은 최종적으로 __alloc_pages_nodemask를 사용하게 된다.

      i. struct page * __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order, struct zonelist *zonelist, nodemask_t *nodemask) (mm/page_alloc.c)

         - 주요 함수 파라미터
           unsigned int order → 할당 받을 페이지의 수를 의미, 2^order개의 페이지를 할당 받음
         - 주요 동작
           노드 및 gfp_mask 확인 후 get_page_from_freelist() 함수를 통해 할당을 시도한다.

      ii. static struct page * get_page_from_freelist(gfp_t gfp_mask, unsigned int order, int alloc_flags, const struct alloc_context *ac) (mm/page_alloc.c)

         - 주요 동작
           각 zone을를 순회하면서 충분한 페이지가 있는 zone을 찾는다. 그 후에 해당 zone에 페이지를 할당 받기 위해 buffered_rmqueue() 함수를 통해 할당 받을 페이지를 찾는다.

      iii. struct page *buffered_rmqueue(struct zone *preferred_zone, struct zone *zone, unsigned int order, gfp_t gfp_flags, unsigned int alloc_flags, int migratetype) (mm/page_alloc.c)

         - 주요 함수 parameter
           struct zone *zone → 페이지를 할당할 zone을 의미
         - 주요 동작
           order가 0인지 확인한다. 0이면 pcp(Per CPU Page Frame Cache, 이하 pcp)를 통해 할당 받을 페이지

를 찾으며 0보다 큰 경우에는 버디시스템을 이용한다. 할당 받을 페이지를 찾아서 free_area를 적절하게 업데이트 하는 __rmqueue 함수이다. 이 함수 전에 스핀락을 걸고 irqsave를 한다. 함수 실행이 끝나면 스핀락만 해제하고 zone에 대한 정보를 업데이트 한 후(zone_statistics() 함수, __count_zone_vm_events 등) irq_restore를 한다.

iv. static struct page *__rmqueue(struct zone *zone, unsigned int order, int migratetype) (mm/page_alloc.c)

- 주요 동작

  __rmqueue_smallest 함수를 실행하여 요청된 페이지의 order보다 같은 페이지를 할당 받거나 요청된 order보다 크면서 가장 작은 페이지를 할당 받고 그 페이지를 반환한다.

v. static inline struct page *__rmqueue_smallest(struct zone *zone, unsigned int order, int migratetype) (mm/page_alloc.c)

- 주요 동작

  이 함수는 free lists를 순회하면서 주어진 할당 받을 수 있는 가장 작은 페이지의 order를 free list에서 제거한다. 제거한 페이지를 반환한다.

  이 함수의 가장 중요한 부분은 다음과 같다.

  for (current_order = order; current_order < MAX_ORDER; ++current_order) {

  → free list중 요청한 페이지의 order보다 같거나 큰 인덱스만 확인한다.

  area = &(zone->free_area[current_order]);

  → 해당 인덱스의 struct free_area를 가져온다.

  page = list_first_entry_or_null(&area->free_list[migratetype], struct page, lru);

  if (!page)

  continue;

  → 해당 인덱스에서 할당 받을 수 있는 페이지가 없으면 다음 order를 확인한다.

  list_del(&page->lru);

  →할당 받을 수 있는 struct page를 lru 리스트에서 제거한다.

  rmv_page_order(page);

  area->nr_free--;

  → free 페이지의 수를 줄인다.

  expand(zone, page, order, current_order, area, migratetype);

  → 요청 받은 order 보다 더 큰 order로 페이지를 할당 받은 경우 할당 받은 페이지 order보다 작은 free 페이지를 업데이트 한다. 예를 들어 2^4의 페이지를 요청했는데 2^6의 페이지를 할당 받은 경우 free_area[3] 4만큼, free_area[5]의 free 페이지는 2개가 줄어든다.

  set_freepage_migratetype(page, migratetype);

  return page;

  }

order를 MAX_ORDER(11)까지 확인해도 여전히 페이지를 할당 받지 못하면 NULL을 할당한다.

vi. 위의 함수 콜 스택을 거슬러 올라가면서 할당 받은 페이지에 대한 몇 가지 확인을 한다.
예를 들면 get_page_from_freelist 함수에서 prep_new_page라는 함수를 실행하여 할당 받은 페이지에서 poison된 부분이 없는지 확인하고 결과에 따라 페이지를 적절하게 처리한다.

B. 메모리 해제 과정

i. free_page, free_pages를 통해 메모리 해제를 커널에 요청할 수 있다. 최종적으로 __free_pages라는 함수를 실행하게 된다.

ii. void __free_pages(struct page *page, unsigned int order)

- 주요 parameter
  struct page *page → 해제할 페이지
  unsigned int order → 해제할 페이지의 수 2^order

- 주요 동작
  put_page_testzero 함수를 통해 페이지 참조 카운터를 확인해 0인 경우 메모리 해제를 시도한다. order가 0이면 Per CPU Page Frame Cache를 통해 페이지 해제를 시도하며 0 이상인 경우 버디 시스템을 이용해 메모리를 해제한다. 버디 시스템을 통해 메모리 해제를 시도하는 경우 __free_pages_ok 함수를 호출하게 된다.

iii. static void __free_pages_ok(struct page *page, unsigned int order)
해제할 페이지에 대한 flags 확인 등을 한 후 local_irq_save를 한 후 free_one_page를 통해 메모리 해제를 시도한다. 메모리 해제 후 local_irq_restore하여 free_one_page를 호출하기 전의 상태로 복원한다.

iv. static void free_one_page(struct zone *zone, struct page *page, unsigned long pfn, unsigned int order, int migratetype)
몇 가지 확인을 한 후 __free_one_page를 호출하여 버디시스템을 이용한 메모리 해제를 시작한다.

- static inline void __free_one_page(struct page *page, unsigned long pfn, struct zone *zone, unsigned int order, int migratetype)
  이 함수를 통해 버디를 병합하고 free area을 업데이트한다. 함수에서 가장 중요한 부분은 다음과 같다.
  continue_merging:
  while (order < max_order - 1)
  → free하는 페이지의 order부터 max_order-1까지 free_page 병합을 시작한다. order가 max_order이면 바로 free_list에 추가하면 되므로 max_order-1까지만 루프를 돈다.
  buddy_idx = __find_buddy_index(page_idx, order);
  → 버디의 인덱스를 찾는다.(page_idx ^ (1 << order))

buddy = page + (buddy_idx - page_idx);

→ 버디 페이지를 찾는다.

if (!page_is_buddy(page, buddy, order))

goto done_merging;

→ page와 버디가 서로 버디인지 확인하고 버디가 아니라면 병합을 중지하고 done_merging으로 내려간다.

/* ... */

combined_idx = buddy_idx & page_idx;

→ order + 1 에서 사용할 페이지 인덱스

page = page + (combined_idx - page_idx);

→ order + 1에서의 페이지

page_idx = combined_idx;

→ 다음 iteration에서는 page_idx가 combined_idx가 된다.

order++;

}

/* ... */

goto continue_merging

done_merging:

set_page_order(page, order);

if ((order < MAX_ORDER-2) && pfn_valid_within(page_to_pfn(buddy))) {

→ order<11-2 이고 버디가 valid한 영역에 있는 경우

struct page *higher_page, *higher_buddy;

combined_idx = buddy_idx & page_idx;

→ 더 높은 order에서 사용될 페이지 인덱스

higher_page = page + (combined_idx - page_idx);

→ combined_idx에 대한 페이지

buddy_idx = __find_buddy_index(combined_idx, order + 1);

→ combined_idx의 버디 인덱스

higher_buddy = higher_page + (buddy_idx - combined_idx);

→ buddy_idx에 대한 페이지

if (page_is_buddy(higher_page, higher_buddy, order + 1)) {

→ higher_page와 higher_buddy가 order+1에서도 버디일 때

list_add_tail(&page->lru,

&zone->free_area[order].free_list[migratetype]);

→ page의 migration type에 따라 리스트의 종류가 다름. lru 리스트

마지막에 page를 추가. 같은 order의 page끼리 연결하기 위함

goto out;

}

}

list_add(&page->lru, &zone->free_area[order].free_list[migratetype]);

→ 리스트가 비어있는 경우 리스트 맨 처음에 추가

out:

zone->free_area[order].nr_free++;

→ 해당 free_area의 free 페이지의 개수 증가

v. 최종적으로 병합된 order의 free_area의 nr_free를 업데이트 하고 페이지들을 적절하게 병합함으로써 메모리 해제가 종료된다.

2. 64 bit 4-level paging

64bit의 메모리를 지원하기 위해 4 level 페이징 기법이 지원되기 시작하였다. 기zone의 3 level 에서는 Page Global Directory와 Page Middle Directory, Pate Table을 통해 물리 주소를 참조했다면 4 level 페이징에서는 3 level 페이징에 Page Upper Directory가 추가된다는 점이 큰 차이점이다. 우선 4 level 페이징에 대해 알아본다.

A. Page Global Directory(PGD), Page Upper Directory(PUD),Page Middle Directory(PMD), Pate Table(PTE)의 관계와 선형 주소

x86-64아키텍쳐에서의 4 level 페이징이 이루어지는 리눅스에서는 선형 주소를 48비트만 사용하여 실질적으로는 48비트까지의 virtual address가 있는 것이다. 유저 공간은 0x0000000000000000~0x00007fffffffffff이며 커널 공간은 0xffff800000000000~0xffffffffffffffff로 둘다 256TB의 공간을 사용할 수 있다[1]. 48비트의 허선형 주소의 비트를 그림으로 나타내면 다음과 같다.

[1] Memory Layout on AArch64 Linux, https://www.kernel.org/doc/Documentation/arm64/memory.txt

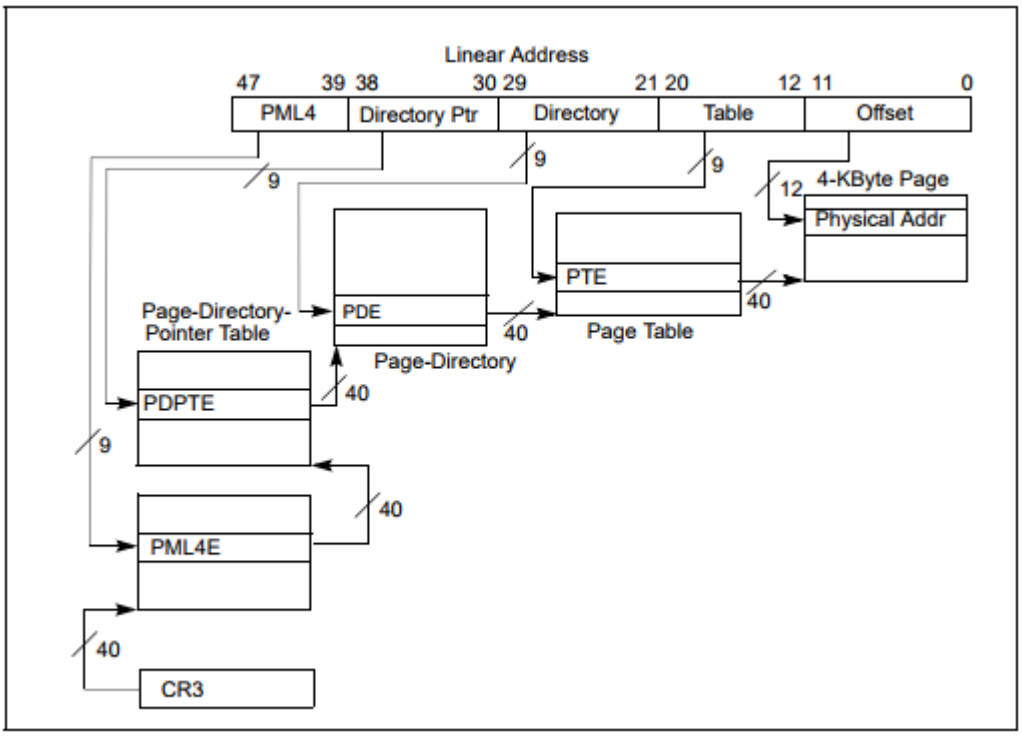


Figure 1-1. Linear-Address Translation Using IA-32e Paging

source: [2]

선형 주소의 상위 9비트를 PML4E 즉 PGD의 인덱스로 사용하고, PGD의 인덱스에 해당하는 값을 통해 PUD의 physical base address를 구한다. 선형 주소의 다음 상위 9비트가 PUD(그림에서 PDPTE)의 인덱스가 되어 PGD를 통해 구한 PUD의 base address와 더하여 PUD의 엔트리 위치를 알아내고 그 엔트리에 해당하는 값을 통해 PMD(그림에서 PDE)의 physical base address를 구한다. 선형 주소의 그 다음 9비트는 PMD의 인덱스에 해당하며 앞서 구한 PMD의 base address에 더하여 Page Table의 base address를 구하며 이를 선형 주소의 그 다음 9비트와 더하여 최종적인 페이지의 위치 또는 base address를 구한다. 선형 주소의 마지막 12비트는 페이지 내부에서의 인덱스를 의미하여 앞서 구한 페이지의 base address와 더하여 최종적인 물리 주소를 알 수 있게 된다.

B. 물리 메모리 참조 프로세스의 흐름

물리 메모리를 참조의 과정이 실제로 어떻게 구현되었는지 확인하기 위해 관련 자료구조, 매크로, 함수 등을 확인하면서 물리 메모리 참조 과정을 확인한다.

i. 관련 자료 구조 및 함수

- pgtable_type.h.
  typedef struct { pgdval_t pgd; } pgd_t;
  typedef struct { pudval_t pud; } pud_t;

[2] 5-Level Paging and 5-Level EPT, intel, MAY 2017, p.12

typedef struct { pmdval_t pmd; } pmd_t;
typedef struct { pteval_t pte; } pte_t;
typedef unsigned longpteval_t;
typedef unsigned longpmdval_t;
typedef unsigned longpudval_t;
typedef unsigned longpgdval_t;
이 자료 구조들이 PGD, PUD, PMD, PTE를 나타내는 자료구조이다. 실질적인 엔트리값은 단순 unsgiend long 타입이다. 64비트 머신이므로 64비트의 정수형 자료다.

- PGD 메모리 할당
  관련 함수는 arch/x86/mm/pgtable.c에 정의되어 있다.
  pgd_t *pgd_alloc(struct mm_struct *mm)
  → static inline pgd_t *_pgd_alloc(void)
  → (pgd_t *)__get_free_page(PGALLOC_GFP)
  순으로 최종적으로는 버디시스템이나 Per CPU Page Frame Cache를 통해 PGD 테이블을 할당 받게 된다. 메모리를 할당 받은 후에는 mm_struct의 pgd 필드의 값을 __get_free_page에서 할당 받은 주소로 바꿔준다.

- PGD 메모리 해제
  관련 함수는 arch/x86/mm/pgtable.c에 정의되어 있다.
  PGD가 해제되기 전에 하위 테이블들이 (e.g. PUD, PMD)를 해제했는지 확인하고 해제한다. 그 후에 __pgd_free →free_page를 통해 버디시스템이나 Per CPU Page Frame Cache를 통해 최종적으로 테이블의 메모리를 해제한다.

- PUD 메모리 할당
  pud_alloc 함수는 mm/mm.h에 정의되어 있다.
  static inline pud_t *pud_alloc(struct mm_struct *mm, pgd_t *pgd, unsigned long address)
  → int __pud_alloc(struct mm_struct *mm, pgd_t *pgd, unsigned long address) (mm/memory.c)
  → static inline pud_t *pud_alloc_one(struct mm_struct *mm, unsigned long addr)(asm/pgalloc.h)
  → get_zeroed_page → __get_zeroed_page
  를 통해 최종적으로 버디시스템이나 Per CPU Page Frame Cache를 이용하여 PUD를 메모리에 할당 받게 된다.

- PUD 메모리 해제
  pud_free 함수는 asm/pgalloc.h에 있다.
  static inline void pud_free(struct mm_struct *mm, pud_t *pud)
  → free_page((unsigned long)pud)
  를 통해 메모리를 해제하게 된다.

- PMD 메모리 할당
  pmd_alloc 함수는 linux/mm.h에 정의되어 있다.
  static inline pmd_t *pmd_alloc(struct mm_struct *mm, pud_t *pud, unsigned long address)
  → int __pmd_alloc(struct mm_struct *mm, pud_t *pud, unsigned long address) (mm/memory.c
  → static inline pmd_t *pmd_alloc_one(struct mm_struct *mm, unsigned long addr) (asm/pgalloc.h)
  → page = alloc_pages(gfp, 0);
  를 통해 메모리를 할당 받게 된다.

- PMD 메모리 해제
  pmd_free 함수는 asm/pgalloc.h에 있다.
  static inline void pud_free(struct mm_struct *mm, pud_t *pud)
  → free_page((unsigned long)pmd)
  를 통해 메모리를 해제하게 된다.

- PTE 메모리 할당
  pte_alloc 함수는 /include/linux/mm.h에 정의되어 있다.
  → int __pte_alloc(struct mm_struct *mm, pmd_t *pmd, unsigned long address) (mm/memory.c)
  → pgtable_t pte_alloc_one(struct mm_struct *mm, unsigned long address) (arch/x86/mmpgtable.c)
  → alloc_pages
  를 통해 메모리를 할당 받게 된다.

- PTE 메모리 해제
  __free_page는 arch/x86/include/asm/pgalloc.h에 정의되어 있다.
  → __free_page(pte);
  를 통해 메모리를 해제한다.

- virtual base address 구하는 매크로/함수
  테이블 엔트리로 있는 base address는 물리 주소인데 커널 자체는 virtual address만 다루므로 다시 virtual address로 바꿔줘야 인덱스를 더해서 다음 테이블 엔트리의 위치를 알 수 있다.

  - PGD → static inline unsigned long pgd_page_vaddr(pgd_t pgd)
  - PUD → static inline unsigned long pud_page_vaddr(pud_t pud)
  - PMD → static inline unsigned long pmd_page_vaddr(pmd_t pmd)

- 테이블의 인덱스를 구하는 매크로/함수 (/arch/x86/include/asm/pgtable.h)

  - PGD → #define pgd_index(address) (((address) >> PGDIR_SHIFT) & (PTRS_PER_PGD - 1))
    PGDIR_SHIFT = 39, PTRS_PER_PGD = 512

- PUD → static inline unsigned long pud_index(unsigned long address)
- PMD → static inline unsigned long pmd_index(unsigned long address)
- PTE → static inline unsigned long pte_index(unsigned long address)

- offset구하는 매크로/함수(arch/x86/include/asm/pgtable.h)
  - PGD → #define pgd_offset(mm, address) ((mm)->pgd + pgd_index((address)))
  - PUD → static inline pud_t *pud_offset(pgd_t *pgd, unsigned long address)
  - PMD → static inline pmd_t *pmd_offset(pud_t *pud, unsigned long address)

ii. 물리 메모리 참조 프로세스

PGD의 base address는 CR3 레지스터에서 참조할 수 있다. 프로세스 별로 사용하는 PGD가 다르기 때문에 context-switch를 하는 중 CR3가 다음 프로세스에 해당하는 PGD를 가리키도록 업데이트 된다. 해당 프로세스의 PGD physical base address는 task_struct의 mm_struct 안에 pgd_t * pgd를 통해 확인할 수 있다.

Table 4-12. Use of CR3 with 4-Level Paging and CR4.PCIDE = 0

Bit Position(s)	Contents
2:0	Ignored
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the PML4 table during linear-address translation (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the PML4 table during linear-address translation (see Section 4.9.2)
11:5	Ignored
M-1:12	Physical address of the 4-KByte aligned PML4 table used for linear-address translation[1]
63:M	Reserved (must be 0)

source: [3]

64bit 아키텍쳐의 CR3 레지스터에서 12비트 부터 M-1비트까지(M은 최대 물리 주소를 나타냄 보통은 52)가 PGD의 base address를 나타낸다. 메모리를 참조할 때마다 MMU[4]는 CR3의 PGD에 해당하는 비트들을 참조하여 PGD의 base address를 구한다.

---

[3] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide(이하 Intel architecture manual vol3), October 2017, Vol. 3A 4-19

[4] Memory Management Unit: virtual address에서 physical address로 변환 및 메모리 보호 등의 일을 하는 하드웨어

PGD의 base address를 구하고 선형 주소의 상위 9비트에 해당하는 수를 더하여 PGD의 엔트리 값을

63	62	61	60	59	58	57	56	55	54	53	52	51		M[1]	M-1			32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved[2]															Address of PML4 table																								Ignored							PCD	PWT	Ign.			CR3
XD	Ignored													Rsvd.	Address of page-directory-pointer table																							Ign.			Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	PML4E: present
Ignored																																																0	PML4E: not present
XD	Prot. Key[4]		Ignored											Rsvd.	Address of 1GB page frame																							Reserved				PAT	Ign.		G	1	D	A	PCD	PWT	U/S	R/W	1	PDPTE: 1GB page
XD	Ignored													Rsvd.	Address of page directory																							Ign.			0	Ign	A	PCD	PWT	U/S	R/W	1	PDPTE: page directory
Ignored																																																0	PDTPE: not present
XD	Prot. Key[4]		Ignored											Rsvd.	Address of 2MB page frame																							Reserved				PAT	Ign.		G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 2MB page
XD	Ignored													Rsvd.	Address of page table																							Ign.			0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																																																0	PDE: not present
XD	Prot. Key[4]		Ignored											Rsvd.	Address of 4KB page frame																							Ign.			PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																																																0	PTE: not present

확인한다. 엔트리는 다음과 같이 구성되어 있다.

source: [5]

위의 표에서 PML4E present가 PGD의 엔트리를 의미한다. 리눅스에서 M=52이므로 총 40비트(12-52)를 다음 directory인 PUD의 base address를 구하는데 사용된다. 0-11비트는 0으로 만든 후에 PUD의 base address로 사용하게 된다.

PUD의 base address를 찾게 되면 선형 주소의 30-38비트를 인덱스로 하여 PMD의 base address를 찾아낸다. PUD의 base address는 PGD의 엔트리 값에서 하위 12비트는 0으로 그 다음 40비트(12-52)는 그대로 둔 것이다. PUD의 base address에 선형 주소의 30-38비트를 더하면 PMD의 physical base address를 알 수 있게 된다.

[5] Intel architecture manual vol3, Figure 4-11

PMD의 base address를 찾게 되면 PUD와 마찬가지로 PUD에서 base address를 구하고 선형 주소의 29-21비트를 인덱스로 하여 PMD의 엔트리 값에 접근하며 접근한 값의 12-52비트 총 40비트는 PTE의 physical base address가 된다.

PTE의 base address를 찾았기 때문에 참조 요청 받은 페이지의 물리 주소를 알 수 있다. 선형 주소의 20-12비트를 통해 PTE의 인덱스를 알 수 있다. base address에 인덱스를 더하여 (실제로는 인덱스*엔트리의 크기를 더한다.) PTE의 엔트리 값에 접근한다. 엔트리 값을 통해 4KB 페이지의 물리 주소를 알 수 있다.

페이지의 크기를 알았으므로 선형 주소의 12LSB를 통해 실제로 참조해야 하는 위치의 physical memory를 알 수 있다. 물리 메모리를 참조하기 위한 일련의 과정들은 위의 그림 Figure 1-1에서 확인할 수 있다.

C. 3 level paging vs. 4 level paging

3 level paging은 인텔 아키텍쳐의 PAE(Page Address Extension)을 기준으로 한다. PAE는 32비트의 주소공간으로 4GB이상의 물리 메모리의 접근하기 위해 만들어진 것이다. 선형 주소는 32비트이기 때문에 유저프로세스 입장에서는 4GB만 보이지만 커널은 32비트의 가상 주소를 36비트의 물리주소로 맵핑 시키기 때문에 64GB의 물리 메모리로 접근 가능하다. 가상 주소와 물리 주소의 범위의 차이가 있기 때문에 유저 프로세스는 64GB가 아닌 4GB만 사용할 수 있게 되어 유저 프로세스가 접근할 수 없는 물리 주소가 생긴다.

/arch/x86/include/asm/pgtable-3level_types.h에서는 선형 주소의 비트를 어떻게 나누는지 확인할 수 있다. 상위 2개 비트는 PGD, 그 다음 9비트는 PMD, 그다음 9비트는 PTE, 남은 12비트는 페이지 오프셋인 것을 확인할 수 있다. 3 level 페이징과 4 level 페이징의 차이점은 PUD의 사용 여부인 것을 확인할 수 있다. 또한 가상 주소의 비트 수가 32비트와 48비트로 차이가 나기 때문에 커널 공간의 크기와 유저 주소 공간의 크기의 차이가 발생한다. 3 level의 경우 3GB를 유저 공간으로 활용하고 커널은 1GB만 활용한다. 한편, 4 level paging의 경우 주소 공간의 크기가 매우 크기 때문에 앞서 언급한 것처럼 유저공간, 커널 공간 모두 256TB를 사용할 수 있다.

아래는 PAE를 사용할 때 선형 주소의 비트에 대한 매크로다.

```
#ifdef CONFIG_X86_PAE // pgtable-3level_types.h
# include <asm/pgtable-3level_types.h>
# define PMD_SIZE          (1UL << PMD_SHIFT)      // PMD_SHIFT=21
# define PMD_MASK          (~(PMD_SIZE - 1))
#else
# include <asm/pgtable-2level_types.h>
#endif
#define PGDIR_SIZE (1UL << PGDIR_SHIFT)            // PGDIR_SHIFT = 30
#define PGDIR_MASK         (~(PGDIR_SIZE - 1))
```

→선형 주소 비트를 30, 21, 12(page offset)으로 구분하는 것을 확인할 수 있다.

리눅스에서는 4 level 페이징을 기본적으로 지원하는 상태에서 3 level 페이징을 구현하기 위해 pgd와 pud

를 동일하게 간주하고 사용한다. 아래 코드들은 이를 뒷받침하는 코드들이다.

```
#include <asm-generic/pgtable-nopud.h> // /arch/x86/include/asm/pgtable_types.h
static inline pudval_t native_pud_val(pud_t pud)
{
   return native_pgd_val(pud.pgd); → pgd의 엔트리 값을 그대로 반환한다.
}
```

pgd의 엔트리값 및 주소를 pud 관련 연산을 할 때도 그대로 사용함으로써 코드레벨에서는 4 level로 작동되나 실제로는 3 level로 메모리 addressing이 이루어지는 것을 확인할 수 있다.

3. 유저 프로세스의 메모리 관리

유저 프로세스의 메모리 할당은 페이지 폴트로 인해 일어나게 된다. 우선 페이지 폴트와 어떻게 처리하는지 확인한다.

A. 페이지 폴트

페이지 폴트는 하드웨어(MMU)가 물리 메모리 주소로 변환하는 과정에서 발생한다. 해당 물리 메모리가 메모리 접근을 요청한 프로세스의 속한 곳이기는 하지만 할당 받은 적이 없는 경우에 발생한다. 그 때 do_page_fault()→__do_page_fault()를 실행하여 페이지 폴트를 처리한다.

i. static noinline void __do_page_fault(struct pt_regs *regs, unsigned long error_code, unsigned long address) (/mm/memory.c)

```
{
        struct vm_area_struct *vma;
        struct task_struct *tsk;
        struct mm_struct *mm;
        int fault, major = 0;
        unsigned int flags = FAULT_FLAG_ALLOW_RETRY | FAULT_FLAG_KILLABLE;

        tsk = current;
        mm = tsk->mm;

        /* ... some checks */

        /* 페이지 폴트가 커널 스페이스에서 일어났을 때 */
        if (unlikely(fault_in_kernel_space(address))) {
                // 커널 모드의 권한을 가지고 있을 때
                if (!(error_code & (PF_RSVD | PF_USER | PF_PROT))) {
                        // vmalloc_fault 일 때
```

```
                        if (vmalloc_fault(address) >= 0)
                                return;

                        if (kmemcheck_fault(regs, address, error_code))
                                return;
                }
                /* some checks .. */
                /* bad area로 감 */
                bad_area_nosemaphore(regs, error_code, address, NULL);

                return;
}

/* some checks ...   */

/* 인터럽트 컨텍스트에 있을 때 → bad_area */
if (unlikely(faulthandler_disabled() || !mm)) {
                bad_area_nosemaphore(regs, error_code, address, NULL);
                return;
}
/* ... */
/* vm_area가 mm 안에 있는 address인지 확인 */
vma = find_vma(mm, address);
if (unlikely(!vma)) {
                bad_area(regs, error_code, address); // 아니면 → bad_are
                return;
}
if (likely(vma->vm_start <= address))
                goto good_area;

/* address가 유저모드 스택에 포함될 수 있는지 확인 아니면 → bad area*/
if (unlikely(!(vma->vm_flags & VM_GROWSDOWN))) {
                bad_area(regs, error_code, address);
                return;
}
```

```
        if (error_code & PF_USER) {
                /*
                 * Accessing the stack below %sp is always a bug.
                 * The large cushion allows instructions like enter
                 * and pusha to work. ("enter $65535, $31" pushes
                 * 32 pointers and then decrements %sp by 65535.)
                 */
                if (unlikely(address + 65536 + 32 * sizeof(unsigned long) < regs->sp)) {
                        bad_area(regs, error_code, address);
                        return;
                }
        }

        /* stack 확장을 시도한다. 아니면 → bad area*/
        if (unlikely(expand_stack(vma, address))) {
                bad_area(regs, error_code, address);
                return;
        }

        /* good area: demanding page나 Copy on Write를 통해 처리할 수 있음( handle_pte_fault() )*/
good_area:
        /* write, readable...등의 access를 체크한다. */
        if (unlikely(access_error(error_code, vma))) {
                bad_area_access_error(regs, error_code, address, vma);
                return;
        }
        /* 페이지 폴트를 처리한다. */
        fault = handle_mm_fault(vma, address, flags);
        major |= fault & VM_FAULT_MAJOR;
        /* 다른 케이스의 처리*/
}
```

ii. int handle_mm_fault(struct vm_area_struct *vma, unsigned long address, unsigned int flags)
→ static int __handle_mm_fault(struct vm_area_struct *vma, unsigned long address, unsigned int flags)
(/mm/memory.c)

```
{
```

```
        struct fault_env fe = { // fault 환경 설정
                .vma = vma,
                .address = address,
                .flags = flags,
        };
        struct mm_struct *mm = vma->vm_mm;
        pgd_t *pgd;
        pud_t *pud;

        pgd = pgd_offset(mm, address);
        // PUD 먼저 할당, 이미 있으면 그 PUD를 반환
        pud = pud_alloc(mm, pgd, address);
        if (!pud)
                return VM_FAULT_OOM;
        // PMD 할당, 이미 있으면 그 PMD를 반환
        fe.pmd = pmd_alloc(mm, pud, address);
        if (!fe.pmd)
                return VM_FAULT_OOM;
        /* do something ... */
        return handle_pte_fault(&fe); // 페이지 폴트 환경(fe)에 따라 페이지 폴트 처리
}
```

iii. static int handle_pte_fault(struct fault_env *fe) (/mm/memory.c)

```
{
        /* do some checks and handle it... */
        /* pte가 전부 0인 경우, 한번도 access 되지 않은 경우*/
        if (!fe->pte) {
                /* 익명 페이지인 경우(파일에서 불러오는 것이 아닌 경우)*/
                if (vma_is_anonymous(fe->vma))
                        return do_anonymous_page(fe); // 사실상 do_no_page 와 동일
                else
                        return do_fault(fe); // memory mapped file의 페이지를 할당할 때
        }
        /* pte에서 엔트리를 찾을 수 없으면 swap 캐시에서 찾기 시도 */
        if (!pte_present(entry))
                return do_swap_page(fe, entry);
```

```
        /* ... */
        if (fe->flags & FAULT_FLAG_WRITE) {
                if (!pte_write(entry))
                        /* Copy on Write, fork 계열의 함수 */
                        return do_wp_page(fe, entry);
                entry = pte_mkdirty(entry);
        }
        /* ... */
        return 0;
}
```

iv. static int do_anonymous_page(struct fault_env *fe) (/mm/memory.c)

```
{
        struct vm_area_struct *vma = fe->vma;
        struct mem_cgroup *memcg;
        struct page *page;
        pte_t entry;

        /* do some chekcs ...*/
        /* pte 할당 시도 */
        if (pte_alloc(vma->vm_mm, fe->pmd, fe->address))
                return VM_FAULT_OOM;

        /* do some checks .. and do something*/

        /* 익명 페이지 생성 */
        if (unlikely(anon_vma_prepare(vma)))
                goto oom;
        page = alloc_zeroed_user_highpage_movable(vma, fe->address);
        if (!page)
                goto oom;
        /* 페이지에 대한 */
        entry = mk_pte(page, vma->vm_page_prot);
        if (vma->vm_flags & VM_WRITE)
                entry = pte_mkwrite(pte_mkdirty(entry));
```

```
        fe->pte = pte_offset_map_lock(vma->vm_mm, fe->pmd, fe->address,
                        &fe->ptl);
        if (!pte_none(*fe->pte))
                goto release;

        /* .... */
        /* 새로 만든 페이지를 맵핑하고 관련 counter 증가*/
        inc_mm_counter_fast(vma->vm_mm, MM_ANONPAGES);
        page_add_new_anon_rmap(page, vma, fe->address, false);
        mem_cgroup_commit_charge(page, memcg, false, false);
        lru_cache_add_active_or_unevictable(page, vma);
        /* .... */
}
```

v. static int do_fault(struct fault_env *fe) (/mm/memory.c)

```
{
        /* .... */
        /* 파일을 메모리로 불러온다. */
        if (!(fe->flags & FAULT_FLAG_WRITE))
                return do_read_fault(fe, pgoff); // 읽기 권한
        if (!(vma->vm_flags & VM_SHARED))
                return do_cow_fault(fe, pgoff); // COW 가능
        return do_shared_fault(fe, pgoff); // 여러 태스크에서 공유 가능(서로 쓰는 것이 가능함)
}
```

vi. int do_swap_page(struct fault_env *fe, pte_t orig_pte) (/mm/memory.c)

```
{
        /* ... */
        /*architecture-dependent한 PTE를 architecture-independent한 swp_entry_t 타입으로 바꿈 */
        entry = pte_to_swp_entry(orig_pte);
        if (unlikely(non_swap_entry(entry))) {
                /* entry가 swap 엔트리가 아닌 경우 처리하고 함수 반환 ..*/
        }
        delayacct_set_flag(DELAYACCT_PF_SWAPIN);
        /* swap 캐시 페이지 구하기 */
        page = lookup_swap_cache(entry);
        if (!page) {
```

```
                /* 페이지를 못찾으면 페이지를 읽어온다.*/
                page = swapin_readahead(entry,
                                        GFP_HIGHUSER_MOVABLE, vma, fe->address);
                /* do some checks and properly handle it ... */
        } else if (PageHWPoison(page)) {
                /* 페이지가 poisoned 되었을 때 처리 */
        }

        swapcache = page;
        locked = lock_page_or_retry(page, vma->vm_mm, fe->flags);

        /* do some checks and proper setting */

        /* 레퍼런스 카운터 업데이트 */
        inc_mm_counter_fast(vma->vm_mm, MM_ANONPAGES);
        dec_mm_counter_fast(vma->vm_mm, MM_SWAPENTS);
        /* PTE 만들기 */
        pte = mk_pte(page, vma->vm_page_prot);
        /* .... */
        /* 맵핑 */
        set_pte_at(vma->vm_mm, fe->address, fe->pte, pte);

        /* ... */
        /* swap 엔트리를 제거 */
        swap_free(entry);
        /* ... */
        if (fe->flags & FAULT_FLAG_WRITE) {
                /* COW가 필요한 경우 */
                ret |= do_wp_page(fe, pte);
                if (ret & VM_FAULT_ERROR)
                        ret &= VM_FAULT_ERROR;
                goto out;
        }
        /* ... */
}
```

vii. static int do_wp_page(struct fault_env *fe, pte_t orig_pte) __releases(fe->ptl) (/mm/memory.c)

```
/* Copy on Write를 해야할 때 실행 */
{
        struct vm_area_struct *vma = fe->vma;
        struct page *old_page;
        /* PTE와 연관된 struct page를 반환한다. */
        old_page = vm_normal_page(vma, fe->address, orig_pte);
        if (!old_page) {
                /* 공유 가능하면서 쓰기 권한이 있는 경우 공유 상태로 만든다.*/
                if ((vma->vm_flags & (VM_WRITE|VM_SHARED)) ==
                                        (VM_WRITE|VM_SHARED))
                        return wp_pfn_shared(fe, orig_pte);
                /* 쓰기권한이 가능한 동일한 페이지 프레임을 만든다. (COW)*/
                pte_unmap_unlock(fe->pte, fe->ptl);
                return wp_page_copy(fe, orig_pte, old_page); → 아래 참조
        }
        /* some extra logic*/
}
```

viii. static int wp_page_copy(struct fault_env *fe, pte_t orig_pte, struct page *old_page) (/mm/memory.c)

```
{
        struct vm_area_struct *vma = fe->vma;
        struct mm_struct *mm = vma->vm_mm;
        struct page *new_page = NULL;
        pte_t entry;
        int page_copied = 0;
        const unsigned long mmun_start = fe->address & PAGE_MASK;
        const unsigned long mmun_end = mmun_start + PAGE_SIZE;
        struct mem_cgroup *memcg;
        /*...*/
        /* 기zone에 있던 페이지가 zero 페이지면 zero 페이지를 할당*/
        if (is_zero_pfn(pte_pfn(orig_pte))) {
                new_page = alloc_zeroed_user_highpage_movable(vma, fe->address);
                if (!new_page)
                        goto oom;
        } else {
```

```
		/* 해당 vma에 페이지를 할당, 이때의 vma는 copy를 진행하는 vma */
		new_page = alloc_page_vma(GFP_HIGHUSER_MOVABLE, vma,
					fe->address);
		if (!new_page)
			goto oom;
		/* 할당한 페이지에 old_page의 내용을 복사한다. */
		cow_user_page(new_page, old_page, fe->address, vma);
	}

	/* .... */
	/*
	 * PTE 다시 확인
	 */
	fe->pte = pte_offset_map_lock(mm, fe->pmd, fe->address, &fe->ptl);
	if (likely(pte_same(*fe->pte, orig_pte))) {
		if (old_page) {
			// 기zone 페이지가 익명페이지가 아닐 때
			if (!PageAnon(old_page)) {
				// 기zone의 mm_counter 감소
				dec_mm_counter_fast(mm,
						mm_counter_file(old_page));
				// 새로운 mm_counter 증가
				inc_mm_counter_fast(mm, MM_ANONPAGES);
			}
		} else {
			inc_mm_counter_fast(mm, MM_ANONPAGES);
		}
		flush_cache_page(vma, fe->address, pte_pfn(orig_pte));
		// 해당 페이지에 대한 PTE 생성
		entry = mk_pte(new_page, vma->vm_page_prot);
		// 해당 페이지 writable하게 만듦
		entry = maybe_mkwrite(pte_mkdirty(entry), vma);
		/* 페이지 테이블 맵핑 */
		ptep_clear_flush_notify(vma, fe->address, fe->pte);
		page_add_new_anon_rmap(new_page, vma, fe->address, false);
```

```
                mem_cgroup_commit_charge(new_page, memcg, false, false);
                lru_cache_add_active_or_unevictable(new_page, vma);

                set_pte_at_notify(mm, fe->address, fe->pte, entry);
                update_mmu_cache(vma, fe->address, fe->pte);
                if (old_page) {
                        page_remove_rmap(old_page, false);
                }
                /* ... */
        } else {
                /* ... */
                }
                /* do some check, lock, unlock ... */
                        return page_copied ? VM_FAULT_WRITE : 0;
                /* .... */
        }
```

B. 메모리 할당

페이지 폴트를 해결하는 함수로 크게 4 가지가 있다 - do_anonymous_page(), do_fault(), do_swap_page(), do_wp_page(). 유저 프로세스가 메모리를 할당해야 하는 경우를 크게 6 가지로 분류할 수 있다. 각 경우에 어떤 함수를 써서 페이지를 할당 받는지 알아본다.

i. 새로운 프로세스 → fork()

fork로 새로운 프로세스를 만든 경우 새로 페이지를 할당하지 않고 부모 프로세스의 mm_struct를 그대로 사용한다. 이후에 자식 프로세스 또는 부모 프로세스가 mm_struct에 쓰려고 하는 경우 do_wp_page()를 통해 쓰기가 새로운 페이지를 할당해 쓰기가 필요한 페이지만 복사한 후 페이지 테이블을 복사한다.

코드 레벨에서 살펴 보면, fork() (/kernel/fork.c) → copy_mm() (/kernel/fork.c) 에서 CLONE_VM 플래그가 set된 경우 단순히 새로운 mm_struct가 부모 프로세스의 mm_struct를 가리키도록 함

```
if (clone_flags & CLONE_VM) { // copy_mm() 내부
                atomic_inc(&oldmm->mm_users); // 레퍼런스 카운터 증가
                mm = oldmm;
                goto good_mm;
        }
mm = dup_mm(tsk); // CLONE_VM unset되어 있으면 dup_mm을 실행, 완전히 새로운 mm을 할당
good_mm:
        tsk->mm = mm;
```

```
tsk->active_mm = mm;
return 0;
```

ii. 파일에 대해 메모리 맵핑을 할 때 → mmap() 시스템 콜 호출
memory mapped 파일은 익명페이지가 아니므로 do_fault를 통해 처리한다. 해당 페이지의 권한에 따라 do_read_fault(), do_cow_fault(), do_shared_fault()의 함수에 따라 페이지가 할당되며 페이지 테이블이 적절하게 업데이트 된다.

iii. 유저모드 스택에 새로운 데이터 추가
스택은 익명페이지에 해당하므로 같은 주소 공간 안에서 쓰기를 시도하고 있고 데이터를 추가한다는 것은 이전에 접근했던 적인 없는 새로운 주소에 대한 접근을 의미하므로 do_anonymous_page()를 통해 새로운 페이지가 할당된다.

iv. 다른 Cooperating process와 데이터 공유(IPC shared memory)

v. malloc()을 통해 힙 메모리에 메모리 영역을 할당 받을 때
힙은 익명페이지에 해당하므로 같은 주소 공간 안에서 쓰기를 시도하고 있고 메모리 영역을 할당 받는다는 것은 이전에 접근했던 적인 없는 새로운 주소에 대한 접근을 의미하므로(swap 된 적이 없음) do_anonymous_page()를 통해 새로운 페이지가 할당된다.

vi. 완전히 다른 프로그램 실행 → exec()를 실행했을 때
보통은 fork를 하고 자식 프로세스에서 exec를 실행하기 때문에 fork를 하므로 mm_struct를 공유한다. 이후 자식 프로세스에서 완전히 다른 프로그램으로 바꿔나가는 과정(exec)에서 페이지 테이블은 완전히 초기화 한 후(이 과정에서 COW가 발생) 새로 실행되는 프로그램에 필요한 정보를 로드하기 때문에 메모리에 접근할 때마다 페이지 폴트가 일어난다. 메모리 할당 시 익명 페이지도 있을 수 있고 메모리 맵 파일(Memory Mapped File)도 있을 수 있기 때문에 할당 받을 페이지에 따라 do_anonymous_page()과 do_fault() 호출된다.

C. 메모리 해제
프로세스가 종료될 때 exit_mm으로 주소 공간을 해제한다.
do_exit() (/kernel/exit.c) → exit_mm(tsk);

i. static void exit_mm(struct task_struct *tsk) (/kernel/exit.c)
{

```
struct mm_struct *mm = tsk->mm;
struct core_state *core_state;
/* 같은 mm을 참조하는 다른 프로세스가 있다면 레퍼런스 카운터만 감소시키고 실제로 mm을 완전히 제거하지는 않는다.*/
mm_release(tsk, mm);
```

```
        if (!mm)
                return;
        /*mm을 참조하는 다른 프로세스가 없는 경우*/
        /* ... */
        /*  __mmput()을 통해 vm_area_struct를 메모리에서 해제하고 __mmdrop을 통해
        mm_struct도 해제*/
        mmput(mm);
        /* ... */
}
```

실습 과제 수행 보고서

1. 과제 수행 시스템 환경

```
Linux ubuntu 4.9.50-2014122011 #1 SMP Sun Nov 26 03:14:29 PST 2017 x86_64 x86_64
 x86_64 GNU/Linux
eunsoo@ubuntu:/boot$ cat config-4.9.50-2014122011 | grep CONFIG_SPARSE_IRQ
CONFIG_SPARSE_IRQ=y
```

Memory	4 GB
Processors	4
Hard Disk (SCSI)	40 GB
CD/DVD (SATA)	Auto detect
Network Adapter	NAT
USB Controller	Present
Sound Card	Auto detect
Printer	Present
Display	Auto detect

2. 수행과정

A. 모듈 파라미터를 사용하는 방법을 간단하게 익히고 구현하였다.

B. 입력 받은 주기마다 정보를 업데이트 할 수 있도록 timer를 사용하여 주기마다 해당 함수가 실행되도록 설정

C. 업데이트가 되지 않을 때에는 항상 같은 정보를 출력해야 하므로 업데이트 된 순간의 정보들을 저장하기 위한 자료구조들이 필요하였다. 정보를 보관하기 위한 구조체들을 만든다. 구조체는 크게 rss가 가장 큰 5 개의 프로세스를 저장하는 rss_array_t, 버디시스템의 정보를 저장하는 node_info_list_t, 정해진 프로세스의 코드 시작하는 주소의 물리 주소를 얻기 위해 필요한 정보들을 저장하는 virt_addr_info_t 총 3 개의 구조체를 설정하고 구조체의 초기화, 업데이트, 제거에 필요한 함수들을 정의하였다.

D. proc 파일로 출력하기 위해서 proc파일 및 함수 작성, seq_file 함수 작성하였다.

E. 최종 테스트를 진행하고 /proc 폴더 내부의 시스템 정보를 출력한 것과 비교하면서 출력값 확인을 진행하였다.

3. 세부 내용

A. 코딩 스타일

i. 각 구조체 별 유지 보수의 효율성을 높이기 위해 소스코드를 구조체별로 구분하여 작성하고 각 소스파일에 대한 헤더파일은 메인 함수에서 꼭 필요한 함수들과 구조체 선언만 포함하였다. 내부적으로 사용되는 함수들은 포함하지 않았다.

ii. 컴파일 시 발생하는 경고 제거를 위해 사용되는 변수는 가장 앞에 선언하였다.

iii. 구조체에 대한 메모리 할당을 위해 kmalloc이 아닌 vmalloc을 사용하였다. 연속적인 물리 메모리의 할당이 필요 없기 때문이다.

iv. 코드 가독성을 높이기 위해 동사를 먼저 사용한 함수명들을 사용하였다.
e.g. rss_array_init (X) → init_rss_array (O)

B. 모듈 동작 상세

i. 모듈 로드시

✓ hw2 proc 파일 생성

✓ rss가 가장 큰 5개의 프로세스를 저장하는 rss_array 메모리 할당 및 초기화

✓ 버디시스템의 정보를 저장하는 buddy_info_list 메모리 할당 및 초기화

✓ virtual address와 관련된 정보를 저장하는 virt_addr_info 메모리 할당 및 초기화

✓ 메모리 할당 시 kmalloc보다는 vmalloc을 통해 메모리 할당, 물리적으로 연속적일 필요가 없음, Interrupt handling할 때 메모리 할당을 하지 않으므로 vmalloc을 사용해도 무방하다.

✓ tasklet과 timer 초기화 및 시작 → tasklet_hrtimer_init/ tasklet_hrtimer_start (include/linux/interrupt.h) timer를 이용하여 tasklet이 실행될 수 있도록 하는 함수를 사용. tasklet을 만들고 타이머 관련 플래그를 tasklet_hrtimer_init에 넘기면 타이머 설정 및 tasklet이 알아서 등록된다. tasklet_hrtimer_start를 호출하면 타이머가 작동하기 시작한다.

ii. tasklet(hw2_tasklet_function)
일반적인 tasklet의 형태와는 다르나 tasklet_hrtimer_init이나 관련 함수 내부적으로는 tasklet을 등록하고 타이머가 expire되면 등록된 tasklet을 실행하도록 함

✓ 마지막으로 업데이트 된 시간을 현재 시간으로 업데이트

- ✓ rss_array 업데이트
- ✓ buddy_info_list 업데이트
- ✓ virt_addr_info 업데이트
- ✓ tasklet이 호출된 시간 계산하기 → ktime_get() 사용
- ✓ hrtimer_forward에 현재 시간과 interval을 넘김으로써 타이머를 초기화 한다.
- ✓ cat /proc/hw2와 같은 proc파일을 read하는 경우 print()함수를 실행한다. print함수는 저장해 놓은 정보들을 그대로 출력하는 함수다.

iii. rss_array 업데이트

- ✓ for_each_process로 모든 프로세스를 순회하면서 mm_struct가 있는 프로세스만 rss를 계산하였다. rss는 task_struct->mm_struct->rss_stat.count[]를 통해 확인할 수 있다. count의 인덱스에 MM_FILEPAGES, MM_ANONPAGES, MM_SHMEMPAGES를 각각 넣은 후 다 더하여 구하였다.
- ✓ rss_array_t 내부의 task struct와 그 프로세스의 rss에 대한 array가 있다. 삽입 정렬을 이용해서 array의 0번 인덱스가 가장 작은 rss를 가지도록 하였다. array의 크기는 5인데 5개의 프로세스가 이미 들어가 있는 경우 새로 들어올 프로세스의 rss와 0번 인덱스의 프로세스와 rss를 비교하고 0번 인덱스 프로세스의 rss가 작으면 0번 인덱스를 빼고 그 자리에 새로운 프로세스를 넣는다 .이후 array 성분 간 swap을 하여 그 프로세스에 맞는 자리에 저장되게 된다. 연결 리스르를 사용하지 않고 array로 구현한 이유는 연결 리스트로 작성하더라도 결국에는 리스트를 한 바퀴 순회해야 새로운 프로세스가 들어갈 자리를 알게 되기 때문이다. 또한 array 크기가 5이기 때문에 array의 프로세스를 정렬하기 들어가는 시간도 그리 크지 않다. 마지막으로 array를 사용하면 추가적인 함수들이 필요가 없기 때문에(각 process에 대한 동적할당과 해제에 관련된 함수) 구조가 간단해진다는 장점이 있다.
- ✓ 주요 함수
  - rss_array_t* rss_array_init(rss_array_t* rss_array) → 메모리 할당 및 초기화
  - void rss_array_clear(rss_array_t* rss_array); → 메모리 해제
  - void rss_array_insert(rss_array_t* rss_array, struct task_struct* p) → 태스크 p를 rss_array에 삽입
  - void rss_array_update(rss_array_t* rss_array); → rss_array 업데이트, rss_array를 초기화 한 후 for_each_process로 rss_array_insert를 호출하는 식으로 업데이트함
- ✓ 주요 자료 구조
  - rss_elem_t
    struct task_struct* p

unsigned long rss → p에 대한 rss

- rss_array_t
  rss_elem_t array[RSS_ARRAY_MAX_SIZE]; → 사이즈 5
  bool is_full; → array가 다 찼는지 확인

iv. buddy_info_list 업데이트

✓ 자료 구조

- zone_info_t → zone에 대한 정보
  unsigned long nr_free[MAX_ORDER]; → 각 zone의 free_area[MAX_ORDER]의 free 페이지 수
  char* type; → struct zone의 char* name (e.g. "DMA", "DMA32", "Normal" ...)

- node_info_t → node에 대한 정보
  zone_info_t zone[MAX_NR_zoneS];→ zone의 정보, 각 인덱스는 DMA, DMA32, Normal 등을 의미
  int node_id; → 노드 ID pg_data_t->node_id(int) (/include/linux/mmzone.h)
  struct buddy_info_node* next; → 다음 노드에 대한 포인터

- node_info_list_t → node_info_t 리스트의 헤드와 리스트 개수를 가지고 있음
  node_info_t* head;
  unsigned int size;

✓ node_info_list_t* buddy_info_list의 초기화
for_each_online_pgdat을 통해 node의 수만큼 buddy_info_array 내부에 node_info_t를 할당 받고 포인터를 연결한다.

✓ buddy_info_list 업데이트 → void update_buddy_info(node_info_list_t* array)
for_each_online_pgdat을 통해 모든 노드를 순회한다. pg_data_t->node_id를 node_info_list_t의 node_id에 저장한 후 노드 내부의 zone을 순회한다. 각 zone의 free_area 배열의 nr_free 필드를 zone_info_t에 저장한다.

v. virt_addr_info_t 업데이트

✓ 자료 구조

- pgd_info_t(struct pgd_info)
  unsigned long long pgd_base_addr; → PGD base address (mm_struct->pgd)
  unsigned long long pgd_addr; → pgd_base_addr + 선형 주소 pgd offset
  unsigned long long pgd_val; → 실제 엔트리 값
  unsigned long long pfn_addr; → PUD의 base address에 해당하는 부분(12-51비트)
  unsigned int pgd_flags; → PGD 플래그

- pud_info_t(struct pud_info)
  unsigned long long pud_addr; → PUD 엔트리의 virtual address
  unsigned long long pud_val; → PUD 엔트리 값
  unsigned long long pfn_addr; → PMD의 base address에 해당하는 부분(12-51비트)

- pmd_info_t(struct pmd_info)
  unsigned long long pmd_addr; → PMD 엔트리의 virtual address
  unsigned long long pmd_val; → PMD 엔트리 값
  unsigned long long pfn_addr; → PTE의 base address에 해당하는 부분(12-51비트)

- pte_info_t (struct pte_info)
  unsigned long long pte_addr; → PTE 엔트리의 virtual address
  unsigned long long pte_val; →PTE 엔트리 값
  unsigned long long pg_base_addr; → 페이지의 base address에 해당하는 부분(12-51비트)
  unsigned int pte_flags; → PTE 플래그
  unsigned long long phys_addr; → physical memory의 주소

✓ 함수 및 매크로 함수

- 선형 주소에서 offset 구하는 매크로 함수 → get_*_offset(linear addr)
  e.g.) get_pgd_offset(addr) → 선형 주소에서의 상위 9비트를 추출한 후 39만큼 shift left

- flag 추출 매크로 함수 → get_*_pgd()/ get_*_pte()
  e.g.) #define get_page_size_pgd(pgd_flags) ((pgd_flags) & 0x80)

- 초기화 함수 → virt_addr_info_t* init_virt_addr_info(void);

- 업데이트 함수 → void update_virt_addr_info(virt_addr_info_t* v);

✓ virt_addr_info_t 업데이트

- mm_struct에서 직접 구할 수 있는 주소들 (e.g. mm_struct->start_code, end_code …) 들은 직접 대입하고 page수는 (end address – start address)>>12 + 1을 하여 구한다. 예를 들어 code 영역의 페이지 수를 구하려면 (end_code – start_code)>>12 + 1 로 구하는 것이다. mm_struct에서 직접 구할 수 없는 주소와 페이지 수는 직접 vm_area_struct를 순회하면서 구한다. 이미 알고 있는 부분은 지나치며 BSS부분과 share library가 있는 부분 stack부분 (stack은 시작 주소 밖에 없다.)을 확인한다.

- pgd_info_t 업데이트 → void update_pgd_info(struct task_struct* p,pgd_info_t* pgd_info)
  PGD의 base address는 mm_struct->pgd에서 확인 가능하다. PGD의 base address에 선형 주소에서 구한 PGD의 offset을 더하여 PGD의 엔트리의 주소를 구한다. 엔트리의 주소를 dereference하

여 pgd_info_t 내부의 필드값들을 업데이트 한다. 엔트리 값의 base address는 물리 주소 이므로 (이는 PUD, PMD, PTE 모두 동일하다.) 물리 주소를 phys_to_virt를 사용해 다시 virtual address로 바꿔줘야 한다.

- pud_info_t 업데이트
  → void update_pud_info(struct task_struct* p, pgd_info_t* pgd_info, pud_info_t* pud_info)
  pgd_info_t를 업데이트 한 상태로 pud_info_t를 업데이트 한다. 위와 마찬가지로 PUD의 base address에 선형 주소의 오프셋을 더해 엔트리의 주소값을 확인한 후 엔트리 값을 통해 pud_info_t를 업데이트 한다.

- pmd_info_t 업데이트 → pud_info_t와 동일하다.

- pte_info_t 업데이트
  위와 거의 동일하나 마지막 최종적인 물리 주소를 구해야 한다. 물리 주소는 선형 주소의 12 LSB와 pg_base_addr와 더하면 된다.

4. 결과 분석

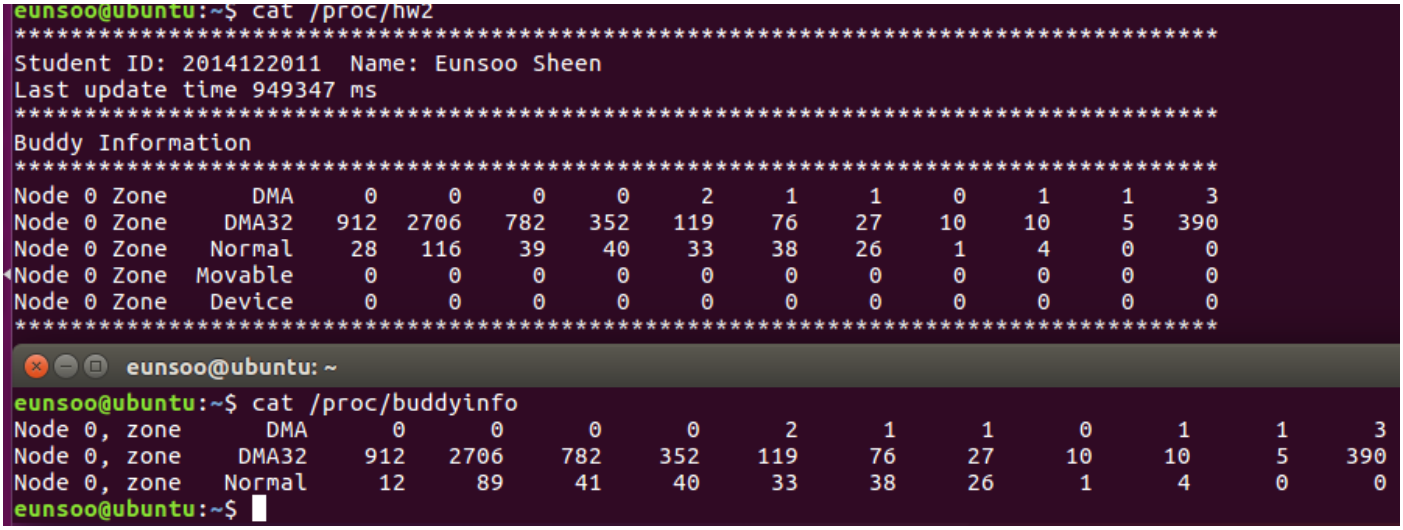
A. rss 정보

```
********************************************************************************
RSS Information
********************************************************************************
pid        rss               comm
1683     48926             compiz
973      22191               Xorg
1757     20038      gnome-software
1729     15082     evolution-calen
2991     14072          notify-osd
********************************************************************************
```

```
eunsoo@ubuntu: /proc
eunsoo@ubuntu:/proc$ ps aux --sort -rss
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
eunsoo     1683  2.8  4.8 1542576 195704 ?      Ssl  17:27   1:20 compiz
root        973  3.0  2.2 516724 88764 tty7     Ssl+ 17:26   1:28 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/run/lightdm/
eunsoo     1757  0.0  1.9 750856 80152 ?        Sl   17:27   0:01 /usr/bin/gnome-software --gapplication-service
eunsoo     1729  0.0  1.4 876864 60328 ?        Sl   17:27   0:00 /usr/lib/evolution/evolution-calendar-factory
eunsoo     2991  0.0  1.3 524056 56288 ?        Sl   17:30   0:01 /usr/lib/x86_64-linux-gnu/notify-osd
```

ps의 rss 계산 방식과는 차이 또는 정보 갱신 주기로 인해 구체적인 수치 차이는 있으나 순서는 맞는 것으로 보아 rss 정보가 제대로 수집되고 있음을 확인할 수 있다.

B. 버디시스템



자료의 갱신 주기의 granularity가 sec 단위이므로 1sec을 period로 하여 테스트한다. 자료의 갱신 주기가 system에서 제공하는 정보에 비해 매우 길기 때문에 완벽하게 동일할 수는 없으나 비슷하다는 것을 확인할 수 있다. 특히 할당의 빈도가 낮은 order가 높은 부분은 거의 정확하게 동일하다. 따라서 버디시스템의 정보가 제대로 구해지고 있는 것을 알 수 있다.

C. virtual address info

정확한 테스트를 위해 프로세스를 "evolution-calen"으로 선택하였다.

cat /proc/(pid)/maps를 출력하여 비교하였다.

****

Virtual Memory Address Information

Process (evolution-calen:1729)

****

0x55f0c2859000 - 0x55f0c285a2ac : Code Area, 2 page(s)

0x55f0c2a5acc8 - 0x55f0c2a5b0b0 : Data Area, 1 page(s)

0x00000000 - 0x00000000 : BSS Area, 0 page(s)

0x55f0c2b1e000 - 0x55f0c4d35000 : Heap Area, 8728 page(s)

0x7f12df7c0000 - 0x7f13130e6000 : Shared Libraries Area, 210448 page(s)

0x7fff5da5f000 - 0x7fff5da7da80 : Stack Area, 33 page(s)

i. 코드 영역 (0x55f0c2859000 - 0x55f0c285a2ac Code Area, 2 page(s))

55f0c2859000-55f0c285b000 r-xp 00000000 08:01 1315448 /usr/lib/evolution/evolution-calendar-factory

code 영역의 시작 주소가 같고 주소의 범위도 위의 vm_area에 포함되며 권한도 read, execution인 것으로 보아 code 영역이 제대로 구해진 것 같다.

ii. 데이터 영역 확인 (0x55f0c2a5acc8 - 0x55f0c2a5b0b0 : Data Area, 1 page(s))
55f0c2a5a000-55f0c2a5b000 r--p 00001000 08:01 1315448
/usr/lib/evolution/evolution-calendar-factory
55f0c2a5b000-55f0c2a5c000 rw-p 00002000 08:01 1315448
/usr/lib/evolution/evolution-calendar-factory
시작 주소와 끝나는 주소가 vm_area 구간 안에 들어가기 때문에 제대로 구해진 것으로 보인다. 메모리 영역의 권한이 read-only와 read-write 2개로 나누지는 이유 static global 변수인 동시에 const한 성질을 가진 변수가 있기 때문에 2 구간으로 나눠진 것으로 보인다.

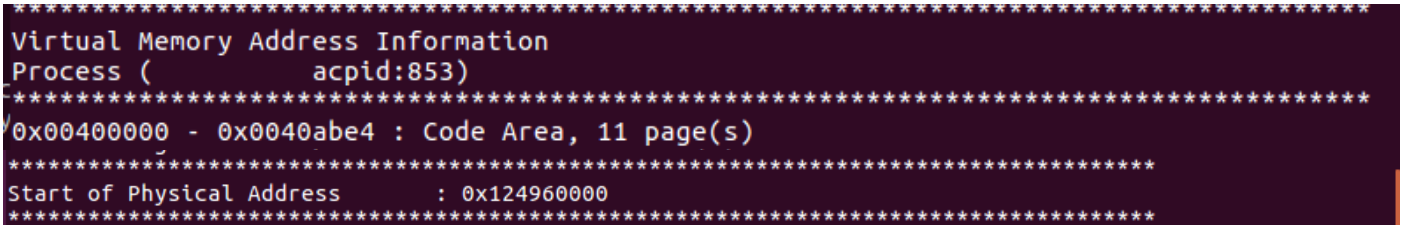
iii. BSS 영역 (0x00000000 - 0x00000000 : BSS Area, 0 page(s))
BSS 영역 초기화를 0으로 하기 때문에 mm->end_data보다 크면서 mm->start_brk보다 작은 vm_area가 없어서 위와 같이 나타난 것으로 보인다.

iv. Heap 영역 (0x55f0c2b1e000 - 0x55f0c4d35000 : Heap Area, 8728 page(s))
55f0c2b1e000-55f0c4d35000 rw-p 00000000 00:00 0 [heap]
7f12df7c0000-7f12df7cb000 r-xp 00000000 08:01 529706 /lib/x86_64-linux-gnu/libnss_files-2.23.so
55f0c2b1e000이 힙 영역의 시작 주소인데 hw2커널 모듈을 통해 구한 주소와 동일하다. 힙 영역의 끝 주소 역시 mm_struct->brk와 일치하므로 힙 영역 정보가 제대로 수집되었음을 확인할 수 있다. 또한 힙 영역 이후에 바로 shared library 영역이 나오는 것으로 보아 heap 영역의 끝 주소가 맞다.

v. shared library 영역 (0x7f12df7c0000 - 0x7f13130e6000 : Shared Libraries Area, 210448 page(s))
shared library 영역은 mm_struct->mmap_base에서 아래로 커진다. 따라서 위에서 구한 shared library의 시작 주소는 실제로는 shared library의 끝 주소를 의미한다. 커널 모듈을 통해 구한 shared library의 시작 주소가 위에서 구한 시작주소와 일치하므로 성공적으로 수집되었음을 확인할 수 있으며 끝 주소 역시 shared library 영역을 포함하고 있다.

vi. stack 영역 (0x7fff5da5f000 - 0x7fff5da7da80 : Stack Area, 33 page(s))
7fff5da5f000-7fff5da80000 rw-p 00000000 00:00 0 [stack]
모듈을 통해 구한 stack의 범위를 proc을 통해 구한 범위가 잘 포함하고 있으므로 제대로 구해졌다고 볼 수 있다.

D. Memory Mapping (virtual address → 물리 주소)
최종적으로 구해진 물리 주소가 정확한지 확인하려면 /proc/<pid>/pagemap을 통해 확인할 수 있다.



parsing을 좀더 쉽게 하기 위해 간단한 파이썬 스크립트를 참고하였다[6].

```
eunsoo@ubuntu:~/hw2/test$ sudo python test.py 853 0x00400000
PFN: 0x124960L
Is Present? : True
Is file-page: True
Page count: 1
Page flags: 0x400000868
eunsoo@ubuntu:~/hw2/test$
```

제대로 구해진 것을 확인할 수 있다.

5. 문제점 및 해결방법

A. 커널 스레드는 mm_struct가 없는 문제

mm_struct의 rss 정보를 확인하며 가장 큰 5개의 프로세스를 선택하기 위해 for_each_process를 통해 프로세스를 순회하는 중 커널 모듈이 제대로 작동하지 않거나 리눅스 자체가 멈춰버리는 현상이 발생하였다. 원인을 파악하던 중 커널 스레드는 mm_struct가 NULL아기 때문에 NULL에 대한 dereference로 인해 문제가 생긴다는 것을 알게 되었다. 이후 포인터를 다루는 함수에서는 항상 포인터가 NULL인지 확인하는 코드를 삽입하여 NULL포인터 에러가 발생하지 않도록 하였다.

B. base address 처리 문제

선형 주소에서 물리 주소로의 변환은 하드웨어가 직접하기 때문에 각 페이지 테이블 또는 디렉토리 엔트리 값들은 virtual address가 아닌 물리 주소다. 이 사실을 간과한채 물리 주소에 직접선형 주소에서 구한 offset을 더하여 다음 테이블의 엔트리 값을 찾으려 했으나 page_fault 또는 컴퓨터가 정지되는 현상이 일어나 발생하였다. OS 자체는 virtual address만 다루기 때문에 phys_to_virt 매크로 함수를 사용해 변환하여 사용하였다.

C. atomic 변수 읽기 문제

rss를 계산하기 위해서는 atomic으로 선언된 변수를 읽어야 한다. 일반 변수를 다루는 것처럼 atomic변수를 다루면 에러가 나기 때문에 atomic 변수를 읽기 위한 함수들(e.g. atomic_long_read(x))를 이용해 읽어들였다.

D. for_each_online_pgdat 컴파일 에러

for_each_online_pgdat 매크로를 확장하면 다음과 같다. (/include/linux/mmzone.h)

```
#define for_each_online_pgdat(pgdat)        \
  for (pgdat = first_online_pgdat();        \
       pgdat;                               \
       pgdat = next_online_pgdat(pgdat))
```

여기서 컴파일을 진행하면 first_online_pgdat()과 next_online_pgdat(pgdat)이 없다며 에러가 발생한다. 커널 내부에서 EXPORT_SYMBOL해서 커널 외부 모듈에서 쓸 수 있도록 해야 하는데 그렇게 하지 않아서 발생한 문제다. 커널 자체를 수정하면 안되므로 mm/mmzone.c에서 함수 정의를 그대로 소스파일로 옮겨서 컴파

[6] Parse the Pagemap Interface, https://blog.jeffli.me/blog/2014/11/08/pagemap-interface-of-linux-explained/

일을 진행하였다.

6. 애로사항

A. bss 영역을 구체적으로 찾기 어려웠으며 맞는지 확신할 수 없어서 mm_strcut->end_data와 mm_struct->start_brk 사이의 페이지 수로 BSS 영역을 찾았다. 간단한 hello world. 프로그램을 짠 후 초기화 되지 않은 전역 변수를 놓고 readelf로 확인하려고 했으나 입출력을 terminal로 하기 때문에 순수하게 BSS의 위치를 확인하기 어려웠다. 찾아보니까 컴파일러마다 같은 변수를 다른 영역에 넣을 수 있기 때문에 (예를 들면 const인 전역 변수, GCC는 코드 영역에 넣는 것으로 되어 있음) 정확한 확인이 어려웠다.

B. 모듈 테스트를 위해 /proc/<pid>/pagemap 확인하며 비교하려고 하였으나 pagemap 자체를 출력하면 5기가 이상의 출력물이 나온다. 적절하게 parsing을 하는 스크립트를 찾아서 해결하였다.