과제2

202011376_조현서_운영체제(3199)_과제2_05.26

1. **과제의 목적**

- 8bit addressing, Linear Page Table을 사용하는 Memory Management Unit의 demo를 구현해 봄으로써 운영체제가 어떠한 방식으로 메모리를 관리하는지 이해할 수 있습니다.
- 특히, 프로세스의 virtual address(이하, va)가 어떠한 방식으로 physical address(이하, pa)로 할당이 되는지, 어떠한 방식으로 swap out/in이 발생하며 free list들은 어떠한 방식으로 관리가 되는지 이해할 수 있습니다.

2. **개요**

- ku_cpu.c, ku_trav.c 그리고 ku_mmu.h 총 3개의 파일을 필요로 합니다.
- main함수는 ku_cpu.c에 구현되어 있으며 ku_trav.c는 va를 pa로 바꿔주는 함수를 가지고 있습니다.
- ku_mmu.h에서는 ku_cpu에서 필요한 메모리 초기화, context switch를 위한 cr3 변경, page fault handler의 기능을 수행합니다.
- 주된 동작으로는 입력으로 input file과 physical memory(이하 pm), swapping space(이하 ss)의 사이즈가 주어졌을 때, free space가 있는 경우 메모리를 할당해 주며, 없는 경우에는 swapping를 이용하며 이때 스케쥴링 알고리즘으로는 fifo를 이용합니다.
- pm의 가장 처음 부분은 OS에 할당되었다고 가정하며, ss의 offset 또한 (neither mapped nor swapped out)한 pte와 구별하기 위해 1부터 시작합니다.

3. **디자인(Design)**

- 기본적인 노드들(free/alloc list의 node, PCB) 전부 linked list 형태의 구조체로 구현하여 해당 자료 구조의 삽입/삭제가 원활하게 일어날 수 있도록 합니다.
- pm, ss을 다루는 free/alloc list들은 swapping시 fifo를 적용하기 때문에 큐 구조를 사용합니다.
- PCB를 다루는 공간은 문제의 상황에서 새로운 프로세스가 추가되기만 할 뿐, 삭제되는 경우는 없기 때문에 스택 구조를 사용합니다.
- PTE를 위한 구조체에는 unsigned char만이 존재하여 8bit를 나타낼 수 있습니다.
- PCB를 위한 구조체에는 pid, page table의 시작 주소, 그리고 linked list 구조를 갖기 위한 다음 PCB 포인터가 존재합니다. page table은 생성 시 0으로 초기화 됩니다.
- free/alloc list들은 pa, ss를 위해 각각 한 쌍씩, 총 4개 존재하며 pa에서 ss로의 이동은 불가합니다.
- 이 list들의 node를 위한 구조체에는 생성 이후에는 바뀌지 않는 pa, ss를 위한 offset 정보와 이 공간을 가리키는 pte 주소, linked list 구조를 갖기 위한 다음 node 포인터가 존재하며, pte 주소는 free space에 있을 때에는 NULL을 가리키게 됩니다.

4. **구현(Implementation)** - 주요 코드(Pseudo)

ku_mmu_init

```
void* ku_mmu_init(unsigned int mem_size, unsigned int swap_size) {
        if (mem_size < 4 || swap_size < 4 || (mem_size % 4) || (swap_size % 4)) return NU;
        // space allocation of mem, swap
        ku_mmu_mem = malloc(sizeof(char) * mem_size);
        ku_mmu_swap = malloc(sizeof(char) * swap_size);

        // initialize mmu_Queues and a mmu_Stack
        큐와 스택을 NULL로 초기화

        // malloc pmem_nodes, initialize and put into pmem_free, mmu_pmem ~ mmu_pmem + 3 allocated for OS
        for (int i = 1; i < mem_size/4; i++) {
                ku_mmu_node* n_node = malloc(sizeof(ku_mmu_node));
                ku_mmu_initNode(&n_node, i);
                ku_mmu_insert(&ku_mmu_mem_free, n_node);
        }
        // malloc swap_nodes, initialize and put into swap_free, offset starts from 4
        for (int i = 1; i < swap_size/4; i++) {
                ku_mmu_node* n_node = malloc(sizeof(ku_mmu_node));
                ku_mmu_initNode(&n_node, i);
```

```
                ku_mmu_insert(&ku_mmu_swap_free, n_node);
        }

        return (void*)ku_mmu_mem;
}
```

- pa와 ss를 입력만큼 할당해 줍니다.
- 각각의 Queue들의 head, tail 값을 NULL로 초기화 해 줍니다.
- pa와 ss의 0번은 사용하지 않으며, 각각의 page에는 4개의 offset이 존재하기 때문에 1부터 mem_size/4 - 1 까지의 노드를 생성하여 각각의 free list에 순서대로 추가해 줍니다.
- pa의 시작 주소를 리턴하며 종료합니다. 반면에 입력 인자가 4보다 작거나 4의 배수가 아닌 경우 0을 리턴합니다.

ku_run_proc

```
// fail -1, success 0
int ku_run_proc(char pid, void** ku_cr3) {
        unsigned char upid = (unsigned char)pid;
        ku_mmu_PCB* block = ku_mmu_get_PCB(upid);
        // if there is no block add block
        if (!block) block = ku_mmu_push(upid);
        if(!block->page_table) return -1;

        *ku_cr3 = block->page_table;
        return 0;
}
```

- pid에 해당하는 PCB를 받아옵니다.
- pid에 해당하는 PCB가 없을 경우 pid에 해당하는 PCB를 생성한 후 PCB Stack에 넣어 줍니다.
- cr3레지스터를 해당 PCB의 page table의 시작 주소로 할당해 주며 만약에 이 값이 NULL이라면 -1을 리턴하여 실패 했음을 나타냅니다.

ku_page_fault

```
//fail -1, success 0
int ku_page_fault(char pid, char va) {
        unsigned char upid = (unsigned char)pid;
        unsigned char uva = (unsigned char)va;

        //get pte using vpn
        char vpn = uva >> 2;
        ku_pte* pte = ku_mmu_get_PTE(upid, vpn);

        if (!pte) return -1;

        // if pte is allocated, pte is in swap_alloc
        if (pte->value) {
            swap_alloc에서 해당 pte를 갖는 노드를 찾은 후 제거, swap_free에 추가
        }

        // if free space exists allocate it
        if (ku_mmu_mem_free.head) {
               free_list의 처음 노드를 할당 한 후 free_alloc에 추가
        }
        // if there is free swap space, swap out and alloc
        else if (ku_mmu_swap_free.head) {
              swap free의 처음 노드에 mem alloc의 처음 노드를 정보를 대입한 후, swap alloc에 추가
              mem alloc의 처음 노드를 할당한 후 mem alloc의 맨 뒤에 추가
        }
        // else, there is no empty space for the allocation
        else return -1;

        return 0;
}
```

- 먼저 해당하는 pid, va의 pte를 얻어 옵니다. 이 때, 해당 pte가 존재하지 않으면 error가 발생합니다.
- page fault가 발생한 상황에서 pte에 value가 있으면, 해당 pte는 swapped out된 상황이므로 해당 pte를 얻어 온 후 그 pte가 있었던 ss을 swap_alloc에서 swap_free로 이동시킵니다.
- 이 후에는 만약에 pm에 free space가 남아있으면 그 공간을 할당하며, ss에만 남아있는 경우에는 mem_alloc 큐의 맨 앞의 원자를 ss에 집어 넣으며 그 pm에 새로운 page를 할당해 주며 mem_alloc 큐의 맨 뒤에 삽입합니다.
- 만약에 free space가 그 어느 곳에도 남아있지 않다면, 더 이상 새로운 공간을 할당할 수 없어 error가 발생합니다.

5. **실행 예**
   a. **./ku_cpu input.txt 100 100**

```
hyeons@hyeons-X10SRA:~/Desktop/OS2$ ./ku_cpu input.txt 100 100
[1] VA: 100 -> Page Fault
[1] VA: 100 -> PA: 4
[1] VA: 10 -> Page Fault
[1] VA: 10 -> PA: 10
[1] VA: 20 -> Page Fault
[1] VA: 20 -> PA: 12
[2] VA: 10 -> Page Fault
[2] VA: 10 -> PA: 18
[2] VA: 20 -> Page Fault
[2] VA: 20 -> PA: 20
[2] VA: 30 -> Page Fault
[2] VA: 30 -> PA: 26
[1] VA: 10 -> PA: 10
```

- pa의 크기가 충분히 커 swapping이 발생하지 않는 경우를 나타냅니다.
- 처음 생성될 때에는 전부다 page fault가 발생하며 그 다음 ([1] VA : 10) 번 호출 시에는 page fault가 발생하지 않습니다.
- PA가 4부터 4씩 순서대로 증가하고 va의 하위 4bit가 offset를 나타내며 fifo로 free memory를 할당해 주기 때문에 PA = 4*n + (VA%4) 가 되는 것을 확인할 수 있습니다.(단, n은 생성 순서)

   b. **./ku_cpu input.txt 20 12**

```
hyeons@hyeons-X10SRA:~/Desktop/OS2$ ./ku_cpu input.txt 20 12
[1] VA: 100 -> Page Fault
[1] VA: 100 -> PA: 4
[1] VA: 10 -> Page Fault
[1] VA: 10 -> PA: 10
[1] VA: 20 -> Page Fault
[1] VA: 20 -> PA: 12
[2] VA: 10 -> Page Fault
[2] VA: 10 -> PA: 18
[2] VA: 20 -> Page Fault
[2] VA: 20 -> PA: 4
[2] VA: 30 -> Page Fault
[2] VA: 30 -> PA: 10
[1] VA: 10 -> Page Fault
[1] VA: 10 -> PA: 14
```

- pm의 크기가 작아 swapping이 일어나는 상황을 나타냅니다.
- pa와 ss의 처음 0번 offset는 사용하지 않기 때문에 ((20-4) + (12-4))/ 4 = 6. 총 6개의 page를 저장할 수 있는 상황에서 (pid,va)= [(1,100),(1,10),(1,20),(2,10),(2,20),(2,30)] 6개의 page를 할당해야 하는 상황입니다.
- pm에는 (20-4)/4 = 4 개의 page를 저장할 수 있기 때문에, 처음 4개까지(PA 시작 주소= 4, 8, 12, 16)는 순서대로 할당하였으며 그 다음 3개의 process가 다시 pa를 요청할 때 fifo로 (PA 시작 주소 = 4, 8, 12) pa를 할당 받는 것을 확인할 수 있으며 이 과정에서 swapped out되었던 (1,10)이 swap out되어 page fault를 일으키며 이전과는 다른 pa(PA 시작 주소 8⇒12)를 할당 받은 것을 확인할 수 있습니다.

   c. **./ku_cpu input.txt 20 8**

```
hyeons@hyeons-X10SRA:~/Desktop/OS2$ ./ku_cpu input.txt 20 8
[1] VA: 100 -> Page Fault
[1] VA: 100 -> PA: 4
[1] VA: 10 -> Page Fault
[1] VA: 10 -> PA: 10
[1] VA: 20 -> Page Fault
[1] VA: 20 -> PA: 12
[2] VA: 10 -> Page Fault
[2] VA: 10 -> PA: 18
[2] VA: 20 -> Page Fault
[2] VA: 20 -> PA: 4
ku_cpu: Fault handler is failed
```

- pm=20, ss=8, ((20-4) + (8-4))/ 4 = 5. 총 5개의 page를 할당할 수 있는 상황에서, 6개의 page를 요청할 시에 5개의 page까지는 할당이 가능하나, 6번째 page인 (2,20)을 추가하려는 순간 handler가 fail한 것을 확인할 수 있습니다.

함수

함수 이름	인자	설명
ku_mmu_initNode	-	노드 생성 시 노드 정보(포인터)를 초기화 해 주는 함수.
	ku_mmu_node** n_node	새롭게 생성하려는 노드 포인터의 주소.
	unsigned char offset	새로운 노드 포인터가 가리키는 메모리 offset정보로 생성 이후에는 변경되지 않음.
	Return Value	(void)
ku_mmu_insert	-	queue의 맨 뒤에 노드를 추가하는 함수.
	ku_mmu_Queue* queue	노드를 추가하고자 하는 목적 큐.
	ku_mmu_node* n_node	추가하려는 노드.
	Return Value	(void)
ku_mmu_delete	-	queue의 맨 앞에서 노드를 삭제하는 함수.
	ku_mmu_Queue*	노드를 삭제하고자 하는 목적 큐.
	Return Value	(ku_mmu_node*)삭제한 노드 주소를 반환.
ku_mmu_push	-	새로운 PCB를 위한 메모리 공간을 만들어 초기화(page table, pid) 한 후 PCB stack의 맨 위에 넣는 함수.
	unsigned char pid	새로 만들 PCB의 pid.
	Return Value	(ku_mmu_PCB*) 새로 만든 PCB 포인터 반환.
ku_mmu_get_PCB	-	원하는 PCB블록을 PCB Stack에서 찾고자 할 때 사용하는 함수.
	unsigned char pid	찾고자 하는 PCB의 pid.
	Return Value	(ku_mmu_PCB*) 찾고자 하는 PCB의 주소를 반환.
ku_mmu_get_PTE	-	원하는 pte를 pid와 vpn을 통해 찾고자 할 때 사용하는 함수.
	unsigned char pid	찾고자 하는 pte의 pid.
	unsigned char vpn	찾고자 하는 pte의 vpn.
	Return Value	(ku_pte*) 찾고자 하는 pte의 주소를 반환.

(인자가 '-' 인 경우 함수 자체의 설명을 의미하며, 문제에서 정의된 함수들은 생략하였습니다.)

붙임. ku_mmu.h.끝.