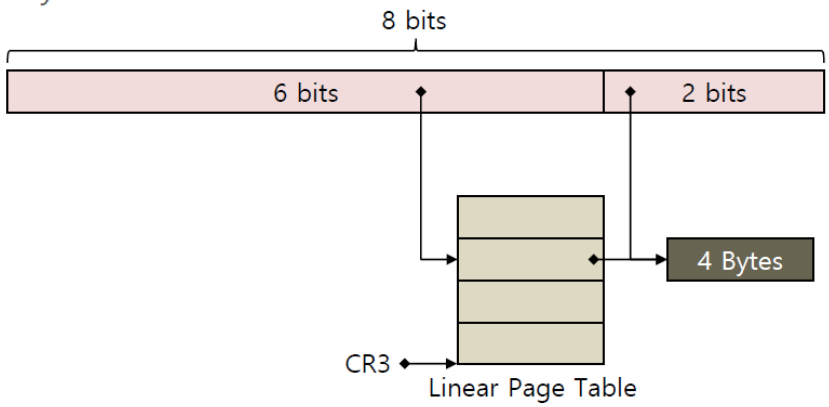
#Assignment 2: KU_MMU

Definition: Address Translation

- 8-bit addressing
  - Address space: 256 Bytes
  - Page size: 4 Bytes
  - PTE: 1 Byte



va가 8비트로 주어지면, 앞의 6비트는 VPN을 뜻하고, 뒤의 2비트는 page offset을 뜻합니다.

`ku_traverse()` 에서는 va와 현재 프로세스의 cr3 (PDBR)를 받아서

address translation을 진행합니다.

(현재 과제에서는 ku_cr3는 프로세스의 page table의 base address를 가리킵니다)

*(ku_cr3 + VPN) 로 PTE를 찾아서, 현재 접근하려고 하는 페이지가 mapping 되어 있는지.

현재 페이지가 physical memory에 present 한지 검사합니다.

만약 mapping이 되어있고 present pa = PFN + page_offset를 리턴합니다.

이러면 Address Translation이 잘 진행되었다는 뜻이고 터미널에도 결과를 잘 출력할 것입니다.

Address Translation이 실패하는 경우

1) va = 0일 경우.

과제에서 이 경우는 null pointer라고 정의가 되어있습니다.

그래서 이때는 page fault handler가 불려도 딱히 해줄 수 있는게 없습니다.

2) PTE = 0인 경우.

주어진 va로 현재 프로세스의 page table에서 해당 page table[VPN]의 PTE가 0일 때.

즉 mapping된 적이 한 번도 없을 때의 경우입니다.

이때는 page fault handler가 mapping을 해줘야 합니다. free list를 통해서

physical memory에서 빈 공간을 찾아서 mapping해줍니다.

만약 빈 공간이 없다면, 가장 마지막으로 mapped된 page를 swap out 시키고

그 자리에 mapping시켜줍니다. **만약** swap space도 다 차서 swap out 시켜줄 수 없다면.

즉 물리 메모리와 swap 공간이 다 찼다면 address translation은 실패입니다.

-1를 리턴해줍니다.

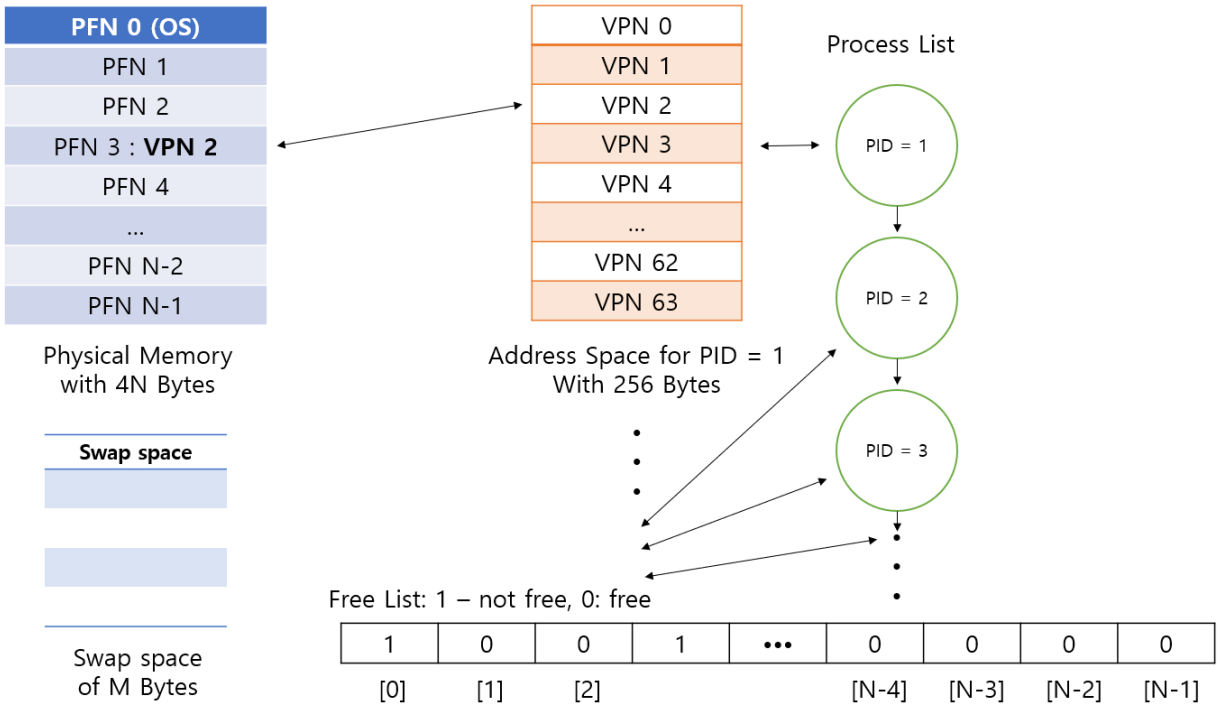
3) mapping은 되어 있으나 present하지 않은 경우.

2)와 비슷하게 현재 물리 메모리에서 빈 공간을 찾아서 mapping해줍니다.

만약 빈 공간이 없다면, 물리 메모리에서 가장 마지막으로 할당된 페이지를 swap space로 옮기고

이렇게 생긴 빈 공간에 현재 프로세스의 페이지를 할당시킵니다.

Basic Structure and Design



해당 사진에서는 pid = 1의 VPN 2와 PFN 3이 mapped 되었다고 가정하고 있습니다.

```
struct ku_pcb {
    char pid;
    struct ku_pte *ptba; /* page table base address */
    struct ku_pcb *next;
};

struct ku_allocNode {
    unsigned char pid, PFN, VPN;
    struct ku_pte PTE;
    struct ku_allocNode *next;
};

unsigned int ku_mmu_mem_size;
unsigned int ku_mmu_swap_size;

struct ku_pte *ku_mmu_pmem; /* Physical Memory */
unsigned char *ku_mmu_sspace; /* Swap space */
struct ku_pcbList ku_mmu_pcbList; /* Process List */
struct ku_allocList ku_mmu_allocList; /* Allocation List */
unsigned char *ku_mmu_freeList; /* Free List */
```

Global variable들입니다.

physical memory, swap space는 `ku_mmu_init`시 인자로 주어진 만큼 할당됩니다. 그리고 그 크기는

각각 `ku_mmu_mem_size`와 `ku_mmu_swap_size`로 저장됩니다.

각각의 프로세스가 가지는 page table들은 프로세스가 새로 실행되었을 때,

ku_run_proc에서 PCB를 생성하면서 그와 동시에 해당 프로세스를 위한 page table도 할당해 줍니다.

PCB는 프로세스의 pid와 프로세스에 배정된 page table의 주소를 가지고 있습니다.

그래서 나중에 context switch를 할 때는 process list에서 실행하려는 프로세스의 pid와 같은 녀석을

process list에서 찾으면, 해당 프로세스의 PCB의 PTBA (Page Table Base Address)로 ku_cr3의 값을

갱신할 수 있습니다.

`ku_mmu_freeList`와 `ku_mmu_allocList` 는 물리 메모리의 free / alloc 여부를 알려줍니다.

freeList는 해당 page frame이 현재 allocated 되어 있으면 1을,

free하면 0이라는 값을 가지고 있습니다. allocList는 mapping history를 나타냅니다.

해당 프로세스와 서로 mapping된 PFN, VPN, 그리고 그 때 PTE값을 가지고 있습니다.

나중에 물리 메모리가 가득 차있어서 page를 evict해야 할 때, FIFO 순서로 page를

swap space로 잠시 swap out 시켜줄 때 사용됩니다.

프로세스가 swap in되거나 swap out될 때, process list에서 enqueue, dequeue 되면서

동시에 해당 PFN의 free 여부도 바뀜에 따라 freeList에서도 그 내용을 반영해줍니다.

Descriptions

Function Name	Function Description	Parameters	Return Value

Function Name	Function Description	Parameters	Return Value
ku_mmu_init	pmem, swap space, process list, free list, alloc list를 위한 공간을 할당받습니다.	pmem_size, swap_size	물리 메모리의 주소. 실패시 0;
ku_run_proc	context switch를 담당합니다. 프로세스가 process list에 없으면, 새로 pcb를 만들고 이를 위한 page table도 할당해줍니다. 그리고 process list에 방금 만든 pcb를 추가합니다.	pid, ku_cr3	성공시 0. 실패시 -1 (page table 할당에 실패한 경우)
ku_page_fault	어떤 이유로 address translation이 되지 않았을 때, 그 원인을 분석해서 문제를 최대한 해결해줍니다. mapping이 되어 있지 않거나 present하지 않으면, swapping도 해주고 mapping도 새로 해줍니다. 다만 va = 0이거나 물리 메모리 / 스왑 공간이 다 차면 -1를 리턴합니다.	pid, va	성공시 0. 실패시 -1.
ku_freeList_firstFit	freeList를 앞에서 부터 traverse하면서 빈 page frame을 찾습니다. 만약 찾지 못한다면, 0을 return합니다.		성공시 해당 PFN. 실패시 0.

이 외에는 pcbList에서의 init, insert, search 함수와 allocList에서의 enqueue, dequeue 와 같이

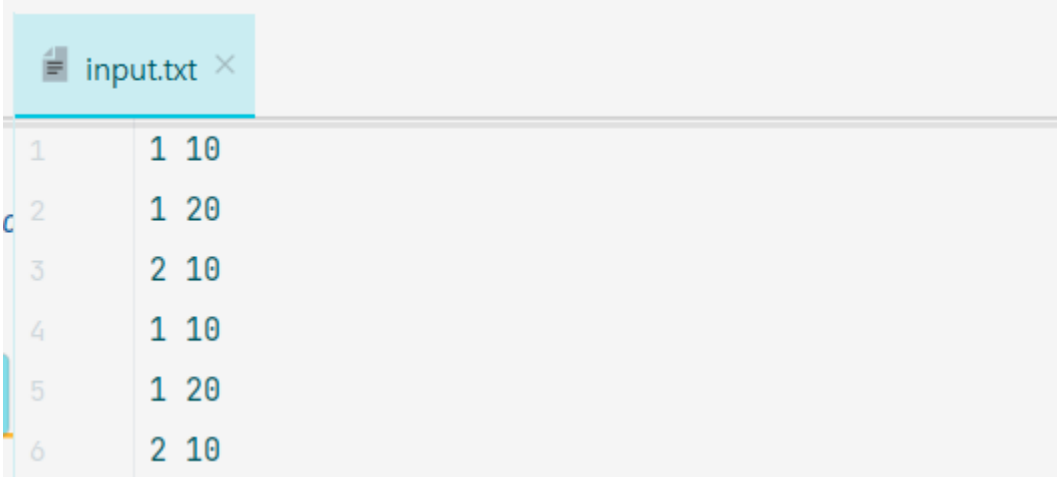
자료구조들을 조작하는 함수들이 있습니다. allocList에서 enqueue하거나 dequeue할 때,

즉 page를 swap in하거나 swap out할 때 allocList에서도 그것을 반영시키고,

이렇게 enqueue와 dequeue가 일어났었을 때 freeList에서도 해당 페이지의 free 여부를 갱신해줍니다.

Conclusion

1) 물리 메모리 공간이 필요한 page의 수 보다 작을 때.



현재 필요한 page의 수는 OS포함 4개입니다.

하지만 물리 메모리의 크기를 8 Byte로 주면 어떻게 될까요?

```
pridom1128@DESKTOP-I73HUGL:/mnt/c/Ocean/OS$ ./ku_cpu input.txt 8 256
[1] VA: 10 -> Page Fault
[1] VA: 10 -> PA: 6
[1] VA: 20 -> Page Fault
[1] VA: 20 -> PA: 4
[2] VA: 10 -> Page Fault
[2] VA: 10 -> PA: 6
[1] VA: 10 -> Page Fault
[1] VA: 10 -> PA: 6
[1] VA: 20 -> Page Fault
[1] VA: 20 -> PA: 4
```

OS를 위한 4 Byte를 제외하면, 실질적으로 사용가능한 공간은 딱 PFN 1개 뿐입니다.

따라서 프로세스를 실행할 때 마다 swapping이 발생되는 것을 볼 수 있습니다.

2) 물리 메모리가 넉넉할 때.

```
pridom1128@DESKTOP-I73HUGL:/mnt/c/Ocean/OS$ ./ku_cpu input.txt 16 8
[1] VA: 10 -> Page Fault
[1] VA: 10 -> PA: 6
[1] VA: 20 -> Page Fault
[1] VA: 20 -> PA: 8
[2] VA: 10 -> Page Fault
[2] VA: 10 -> PA: 14
[1] VA: 10 -> PA: 6
[1] VA: 20 -> PA: 8
[2] VA: 10 -> PA: 14
```

이때는 swap space 없이 충분히 물리 메모리 내에서 mapping을 끝낼 수 있기 때문에,
프로세스가 처음 실행될 때에만 page fault가 발생하고
그 뒤에는 발생하지 않는 것을 볼 수 있습니다.

3) 공간이 부족할 때.

```
pridom1128@DESKTOP-I73HUGL:/mnt/c/Ocean/OS$ ./ku_cpu input.txt 12 4
[1] VA: 10 -> Page Fault
[1] VA: 10 -> PA: 6
[1] VA: 20 -> Page Fault
[1] VA: 20 -> PA: 8
ku_cpu: Fault handler is failed
```

물리 메모리에는 총 PFN 3개가 할당될 수 있고 이미 하나는 OS의 것입니다.
input.txt처럼 프로세스들이 실행되려면 OS 포함 4개의 PFN이 필요합니다.
적어도 16 Byte가 필요한데, swap space의 0번째 공간은 사용하지 않기 때문에
실질적으로 12 Byte밖에 없는 상황입니다. 이때는 page 2개를 할당하고 (OS 포함 3개 할당)
나머지 하나를 실행했을 때 공간이 없어서 Fault handler가 문제를 해결할 수 없어서

-1를 리턴하게 됩니다.

3 - 1) swap space에 추가적으로 4 Byte를 더 주었을 때.

```
pridom1128@DESKTOP-I73HUGL:/mnt/c/Ocean/OS$ ./ku_cpu input.txt 12 8
[1] VA: 10 -> Page Fault
[1] VA: 10 -> PA: 6
[1] VA: 20 -> Page Fault
[1] VA: 20 -> PA: 8
[2] VA: 10 -> Page Fault
[2] VA: 10 -> PA: 6
[1] VA: 10 -> Page Fault
[1] VA: 10 -> PA: 10
[1] VA: 20 -> Page Fault
[1] VA: 20 -> PA: 4
```

이때는 swap space 포함 16 Byte가 있어서 성공적으로 할당해줄 수 있습니다.

다만 page fault가 공간의 부족으로 자주 발생할 것입니다.

모듈화를 제대로 해주지 못해서 코드가 조금 지저분합니다.

죄송합니다.

부족한 보고서를 읽어주셔서 감사합니다.

201811244

컴퓨터공학과 김창엽