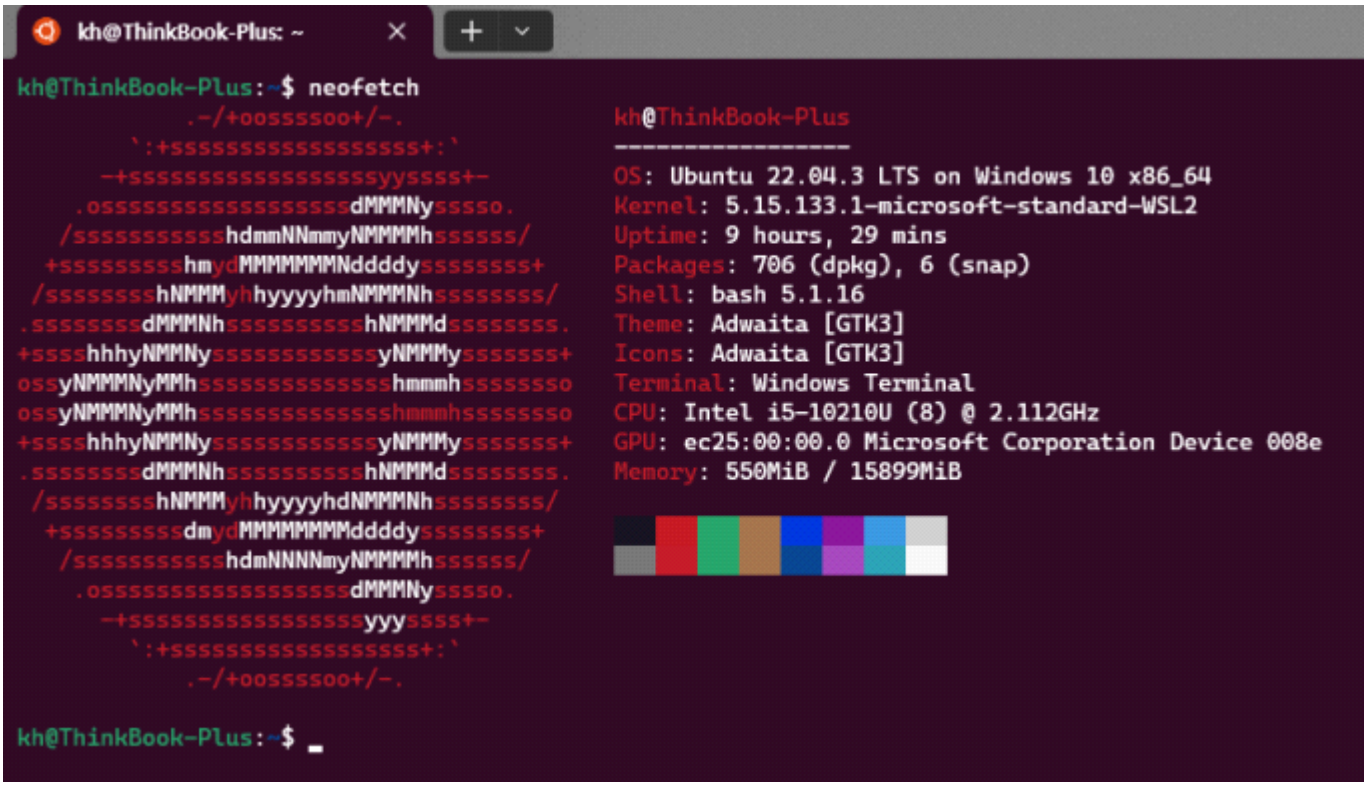
운영체제(가) 4차 과제

일어일문학과

20181755 이건희

2023년 12월

1. Development Environment



장치 사양

장치 이름	ThinkBook-Plus
프로세서	Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz
설치된 RAM	32.0GB(31.8GB 사용 가능)
장치 ID	[redacted]
제품 ID	[redacted]
시스템 종류	64비트 운영 체제, x64 기반 프로세서
펜 및 터치	이 디스플레이에 사용할 수 있는 펜 또는 터치식 입력이 없습니다.

3차 과제를 진행했을때와 동일하다. WSL2(ubuntu 22.04)에서 Vim으로 작업하였다.

2. Hints

11.20(월), 11.27(월) 수업의 마지막에 과제에 대한 힌트를 주셔서, 이를 하나하나 살펴보겠다.

2-1. mmu.h

mmu.h에서 #define ~~ 12로 되어있는 부분이 있다 하셨다.

```
// A virtual address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/

// page directory index
#define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)

// page table index
#define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)

// construct virtual address from indexes and offset
#define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))

// Page directory and page table constants.
#define NPDENTRIES      1024    // # directory entries per page directory
#define NPTENTRIES      1024    // # PTEs per page table
#define PGSIZE          4096    // bytes mapped by a page

#define PTXSHIFT        12      // offset of PTX in a linear address
#define PDXSHIFT        22      // offset of PDX in a linear address
```

이는 mmu.h에 정의되어있다. PTXSHIFT가 쓰이는곳을 보니 바로 위의 PTX(va)에 메크로가 있다. 메크로 PTX(va)는 읽어보면 알겠지만 va(가변주소)를 PTXSHIFT(=12)만큼 오른쪽으로 비트시프트한 뒤, 0x3FF(2진수로 0011 1111 1111)과 and연산을 한다. xv6의 논리적 주소는 사진의 가장 위 주석처럼 되어있다. 여기에서 xv6은 메모리를 32비트로 접근함을, page크기(frame크기)는 4KB임을 알 수 있었다. PDXSHIFT가 22인 것을 보면, PDX와 PTX는 directory index, table index를 연산하기 위한 메크로임을 알 수 있다.

2-2. trapframe

3차 과제에서 필자는 trap.c의 trap()을 자세히 살펴본 적이 있다. trap()은 argv로 trapframe *tf를 받는데, struct trapframe의 원형은 x86.h에 정의되어있다.

```
//PAGEBREAK: 36
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;
```

사진의 아래는 잘라냈다. 레지스터에 저장될 수 있는 값들이 struct member로 있다.

2-3. V2P(p->pgdir)

이것은 수업중에 나온 힌트는 아닌데, 과제의 배경지식으로 있어서 알아두고 과제를 해결하는 것이 도움이 될 것 같아 미리 알아본다.

```
kh@ThinkBook-Plus:~/os/xv6$ grep -r "V2P"
main.c:     *(int**)(code-12) = (void *) V2P(entrypgdir);
main.c:     lapicstartap(c->apicid, V2P(code));
kalloc.c:   if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
memlayout.h:#define V2P(a) (((uint) (a)) - KERNBASE)
memlayout.h:#define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but withs
entry.S: start = V2P_WO(entry)
```

V2P는 memlayout.h에 메크로로 정의되어있다. V2P(a)는 uint(address)에서 – KERNBASE를 뺀, 상대적인 주소를 구한다.

```
// Memory layout

#define EXTMEM  0x100000            // Start of extended memory
#define PHYSTOP 0xE000000           // Top physical memory
#define DEVSPACE 0xFE000000         // Other devices are at high addresses

// Key addresses for address space layout (see kmap in vm.c for layout)
#define KERNBASE 0x80000000         // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked

#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) ((void *)(((char *) (a)) + KERNBASE))

#define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
#define P2V_WO(x) ((x) + KERNBASE)    // same as P2V, but without casts
```

KERNBASE역시 memlayout.h에 정의되어있다. 이를 통해 예상컨대 V2P는 Virtual Address To Physical Address를 줄인 것 같다. 아니, 이렇게 단순하게 주소변환이 된다? 혹시 필자가 무엇을 놓치지는 않았는지 괜히 걱정이다.

처음 제출한 1차 과제에서 'xv6를 만든 mit에서 직접 만든 xv6의 메뉴얼이 있다, xv6가 x86버전에서 risc-v로 넘어가기 위해 x86버전은 유지보수를 중단한다고 README에 적혀있다'라 보고서에 적어서 제출했었다. 또한 수업의 내용 역시나 ia-32를 기준으로 배우고 있다. 따라서 x86용 xv6 메뉴얼을 찾을 필요가 있었고, 검색하니 나왔기에 본 과제의 보고서와 같이 해당 메뉴얼[1]을 제출하겠다. 다운로드한 곳[2]은 주석으로 달아두었다.

1) Russ Cox 외, xv6: a simple, Unix-like teaching operating system, 2018.
2) https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf

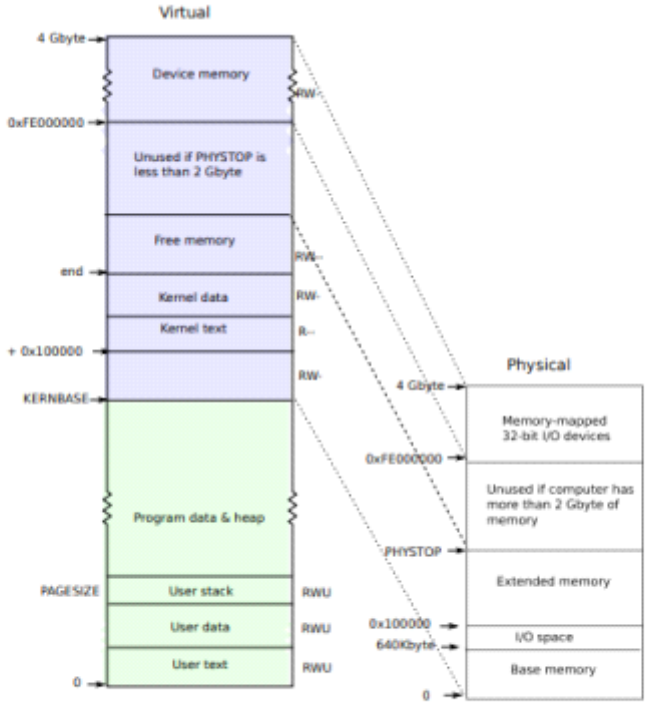


Figure 2-2. Layout of the virtual address space of a process and the layout of the physical address space. Note that if a machine has more than 2 Gbyte of physical memory, xv6 can use only the memory that fits between KERNBASE and 0xFE00000.

메뉴얼의 p.31에 나와있는 xv6의 메모리구조이다. xv6은 컴퓨터가 2기가 이상의 물리적인 메모리가 있을 경우 KERNBASE(=0x80000000)~0xFE000000의 영역만 사용한다는 그림의 설명이 있다. 그런데 그림의 PHYSTOP~0xFE0000000 에는 컴퓨터의 메모리가 2GB 이상일 경우 사용하지 않는다 적혀있다. 메뉴얼에서 PHYSTOP 관련된 부분을 찾아보면

The function `main` calls `kinit1` and `kinit2` to initialize the allocator (3131). The reason for having two calls is that for much of `main` one cannot use locks or memory above 4 megabytes. The call to `kinit1` sets up for lock-less allocation in the first 4 megabytes, and the call to `kinit2` enables locking and arranges for more memory to be allocatable. `main` ought to determine how much physical memory is available, but this turns out to be difficult on the x86. Instead it assumes that the machine has 224 megabytes (`PHYSTOP`) of physical memory, and uses all the memory between the end of the kernel and `PHYSTOP` as the initial pool of free memory. `kinit1` and `kinit2` call

main함수는 allocator를 초기화하기 위해 kinit1과 kinit2를 호출한다. 두 번의 호출을 하는 이유는 대부분의 메인은 락이나 4MB가 넘어가는 메모리를 살 수 없기 때문이다. kini1 호출은 처음 4MB에서 락 없는 할당을 설정하고, kinit2 호출은 락을 활성화하고 더 많은 메모리를 할당할 수 있도록 준비한다. main은 사용 가능한 물리메모리의 양을 결정해야하지만, x86에서는 어려운 것으로 나타났다. 대신에 머신에 224MB(PHYSTOP)의 물리적 메모리가 있다고 가정하고, 커널의 끝과 PHYSTOP 사이의 모든 메모리를 초기 여유메모리의 풀로 사용한다.

p.33에 위와같은 내용이 나온다. V2P/P2V가 저렇게 단순하게 주소변환이 되는 것을 이제야 납득할 수 있다.

3. Codes

[1 단계] : getNumFreePages 시스템 콜 추가

kalloc.c의 수정에 앞서 이 파일이 무엇을 하는지 알아봐야 하는데, 가장 처음의 주석에 나와있다. 주석에 의하면 유저프로세스, 커널 스택, 페이지테이블의 페이지들, 파이프 버퍼들을 할당하도록 의도된 물리메모리 할당자라 되어있다. 또한 4096 바이트(4KB)크기의 페이지를 할당한다 한다. mmu.h를 살펴보면서 알게된 내용, 수업시간에 배운 내용과 동일하다.

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "spinlock.h"

uint num_free_pages;

void freerange(void *vsta
extern char end[]; // fi
```

전역변수 num_free_pages를 선언했다.

```
// Initialization happens in two phases.
// 1. main() calls kinit1() while still using entrypgdir to place just
// the pages mapped by entrypgdir on free list.
// 2. main() calls kinit2() with the rest of the physical pages
// after installing a full page table that maps them on all cores.
void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  num_free_pages = 0;
  kmem.use_lock = 0;
  freerange(vstart, vend);
}
```

num_free_pages를 0으로 초기화 한 것은 한 거고, 주석의 내용을 읽어보면

> 초기화는 두가지 단계로 일어난다
> 1. main()은 Entrypgdir을 사용하는 동안 kinit1()을 호출하여 Entrypgdir에 의해 매핑된 페이지만 사용 가능한 목록에 배치한다.
> 2. main()은 모든 코어에 매핑되는 전체 페이지 테이블을 설치한 후 나머지 물리 페이지와 함께 kinit2()를 호출한다.

kinit은 이렇게 작동한다고 한다.

```
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r){
    kmem.freelist = r->next;
    num_free_pages--;
  }
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

```
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;

  ++num_free_pages;

  if(kmem.use_lock)
    release(&kmem.lock);
}
```

kalloc, kfree에 num_free_pages 조작을 추가했다.

```
int
getNumFreePages(void)
{
    return num_free_pages;
}

int
sys_getNumFreePages(void)
{
    return getNumFreePages();
}
```

syscall getNumFreePages의 추가는 2차 과제처럼 했으나, getNumFreePages의 원함수와 wrapping함수는 kalloc.c에 추가했다. 이외 모든 과정을 보고서에 남기기에는 몇 번이나 해본 syscall 추가이기도 하고, 너무 길어지므로 생략한다.

[2 단계] : 각 메모리 페이지들을 대상으로 reference counter 적용

과제 파일에서 “kalloc.c 파일에 uint pgrefcount[PHYSTOP >> PGSHIFT]”이라는 내용이 있다.

```
kh@ThinkBook-Plus:~/os/hw4$ grep -r "PHYSTOP"
main.c:  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startoo)
kalloc.c:uint pgrefcount[PHYSTOP >> PGSHIFT];
kalloc.c:  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
grep: .git/objects/pack/pack-3a9a3dd50c6f703c146b6a619ce34233f2f64572.packs
memlayout.h:#define PHYSTOP 0xE000000           // Top physical memory
kernel.asm:  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
kernel.asm:  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after st)
grep: .kalloc.c.swp: binary file matches
vm.c://   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
vm.c:// between V2P(end) and the end of physical memory (PHYSTOP)
vm.c:// (directly addressable from end..P2V(PHYSTOP)).
vm.c: { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+mey
vm.c:  if (P2V(PHYSTOP) > (void*)DEVSPACE)
vm.c:    panic("PHYSTOP too high");
kh@ThinkBook-Plus:~/os/hw4$ vi memlayout.h
kh@ThinkBook-Plus:~/os/hw4$ grep -r "PGSHIFT"
kalloc.c:uint pgrefcount[PHYSTOP >> PGSHIFT];
grep: .git/objects/pack/pack-3a9a3dd50c6f703c146b6a619ce34233f2f64572.packs
grep: .kalloc.c.swp: binary file matches
kh@ThinkBook-Plus:~/os/hw4$
```

PHYSTOP은 memlayout.h에 0xE000000로 정의되어있는데, PGSHIFT는 정의되어있지 않았다. 27(월)의 수업 마지막에 과제 설명에서 교수님이 #define PGSHIFT 12;하라 하신걸 필기한게 있는데, 이는 위에서 살펴본 mmu.h에서 이미 PTXSHIFT로 정의되어있었다. 구글링을 해보니 구버전의 xv6에는 pgshift가 있었다.

la를 10:10:12로 쪼갠 것이 수업시간에 배운 내용중 two level page table과 비슷한거같은데 정확하게 모르겠어서, 구글링을 해보니 이에대한 좋은 글들[3]이 있었다. 전공서[4]도 살펴보니, IA-32 아키텍쳐에서 사용되는 페이징기법에 대한 설명이 있었다.

```
// A virtual address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/
```

이들을 읽어보니 10:10:12로 쪼갠 것은 선형주소(linear address)라 하는데, page directory는 2^10개의 page table의 첫주소(포인터)를 가지고 있는데, 이들을 pde라 한다. 여기에서 cr3 레지스터는 프로세스의 page directory의 첫주소(포인터)를 가리킨다.(2차 과제를 했다면 struct proc은 pgdir*을 구조체로 가지고있음을 당연히 알아야한다). 이를 pde라 한다. page table은 알고있는 그 page table이다. 여기에 있는 entry들은 수업시간에 배운 pte라 한다. 이와 같은 페이징 기법은 전공서에서도 two level page table과 비슷한 기법이라 한다. 수업은 아키텍쳐 종속적으로 가르치지 않으려는 의도에서였는지, p.25에서 pte의 길이는 20비트라 뭉퉁그려서 넘어가는식으로 필기가 되어있었다. pde와 pte의 차이를 기억해두는 것이 좋아보인다. 사실 과제를 진행함에 있어서 몰라도 풀수는 있었겠으나 알아보면 좋을 것 같아서 적어봤다.

3) https://rockball.tistory.com/entry/Paging, https://bbolmin.tistory.com/155, https://oasess.tistory.com/44, https://blog.naver.com/hermet/50407725,

4) Abraham Silberschatz 외, Operating System Concepts, 박민규 옮김, 퍼스트북, p.419-420.

그래서 결론은; PGSHIFT를 12로 define해놓고 PHYSTOP >> PGSHIFT를 하는 것은 linear address에서 offset을 제외한 20비트들로만 무언가를 하겠다는 의도로 느껴져서, 필자는 이미 정의되어있는 PTXSHIFT로 PGSHIFT를 대체하겠다.

```
uint num_free_pages;
uint pgrefcount[PHYSTOP >> PTXSHIFT];
```

사진처럼 kalloc.c에 전역변수로 pgrefcount를 선언하였다.

refcount 관련한 함수들을 생각해보자. get_refcount는 물리적인 주소가 주어졌을 때, pgrefcount[해당 주소]를 return해야 한다. int_refcount는 물리적인 주소가 주어졌을 때, pgrefcount[해당 주소]를 하나 올려야한다. dec_refcount는 이의 반대이다.

```
// pa : physical address
uint
get_refcount(uint pa)
{
    return pgrefcount[pa >> PTXSHIFT];
}

void
inc_refcount(uint pa)
{
    ++pgrefcount[pa >> PTXSHIFT];
}

void
dec_refcount(uint pa)
{
    --pgrefcount[pa >> PTXSHIFT];
}
```

이들을 kalloc.c에 위와 같이 정의하고, defs.h도 추가해주었다.

이제 만들어진 함수들을 과제에 명세된 함수들에서 사용해야하는데, kalloc.c는 처음 보는 파일이라 각 함수가 어떤역할을 하는지 알아보면서 하려니 지난주처럼 시간이 너무 많이 걸릴 것 같아서, 1-3에서 언급한 메뉴얼에서 페이징 관련 부분을 참고했다.

먼저, freerange 함수에서 pgrefcount[]을 0으로 초기화 해줘야 한다. 이와 관련된 내용은 메뉴얼의 p.33에 있다.

of the kernel and PHYSTOP as the initial pool of free memory. kinit1 and kinit2 call freerange to add memory to the free list via per-page calls to kfree. A PTE can only refer to a physical address that is aligned on a 4096-byte boundary (is a multiple of 4096), so freerange uses PGROUNDUP to ensure that it frees only aligned physical addresses. The allocator starts with no memory; these calls to kfree give it some to

kinit1, kinit2는 freerange로 메모리를 페이지 단위의 kfree 호출을 하여 free list에 추가한다. PTE (page table entry)는 오직 4kb경계로 정렬된 물리적 주소만 참고할 수 있으므로, freerange는 물리적 메모리에 저장된 주소만 free하기 위해 PGROUNDUP을 사용한다.

지난번 과제를 해결할 때 모르는 함수가 나올 때 마다 혼자서 계속 생각하는게 정말 시간낭비였다. 메뉴얼을 정독하는 습관을 들어야겠다.

```
void
freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE){
    pgrefcount[V2P(p) >> PTXSHIFT] = 0;
    kfree(p);
  }
}
```

따라서 freerange 함수에서 위와 같이 pgrefcount를 0으로 초기화 해 주었다.

두 번째로, kfree 함수에서 dec_refcount를 호출해줘야 한다. 수업의 마지막에 주신 힌트가 떠올랐다. 완벽하게 기억하지는 못하나 refcount가 1이상이면 이를 하나씩 감소시키고, refcount가 0이 되었을때만 free하게 해야한다는 내용이였던 것 같다. 지금 과제를 하면서 드는 생각이, refcount는 uint고 0에서 더 감소가 되면 최대값으로 overflow가 발생하기 때문에 처리를 확실하게 해줘야 할 것 같다. 또한, kfree의 함수의 원형을 보면 v를 run으로 casting하여 r=v이 되고, r->next = kmem.freelist라 지정해주고, kmem.freelist = r이라 지정해준다. 또한 구조체 run은 자기 참조 구조체이기에, freelist는 linked list라 생각이 가능하다.

```
void
kfree(char *v)
{
    struct run *r;

    if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP){
        panic("kfree");
    }

    if(kmem.use_lock)
        acquire(&kmem.lock);

    if(get_refcount(V2P(v)) > 0){
        dec_refcount(V2P(v));
    }
    if(get_refcount(V2P(v)) == 0){
        // Fill with junk to catch dangling refs.
        memset(v, 1, PGSIZE);
        r = (struct run*)v; // casting v;
        r->next = kmem.freelist;
        kmem.freelist = r;
        ++num_free_pages;
    }

    if(kmem.use_lock)
        release(&kmem.lock);
}
```

kfree 함수의 원본은 refcount를 전혀 고려하지 않고, 단순히 패닉이 일어나지 않으면 아래의 모든 내용이 실행이 되지않는 식이다. 새로운 kfree함수는 refcount가 0을 초과하면 dec_refcount를 실행하고, refcount가 0이면 기존 함수에서 panic이 일어나지 않았을 때 실행되던 모든 내용을 실행하도록 만들었다.

세 번째로, copyuvm 함수에서 inc_refcount를 호출해줘야 한다. 그런데 다음 단계에 copyuvm 함수의 수정이 있어서 이 때 같이 하도록 하겠다.

네 번째로, kalloc 함수에서 refcount를 1로 설정해야 한다. 이는 r이 0이 아닐때의 분기에서 간단히 추가가 가능하다.

```
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r){
    kmem.freelist = r->next;
    pgrefcount[V2P((char*)r) >> PTXSHIFT] = 1;
    num_free_pages--;
  }
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

위와 같이 수정하였다.

[3 단계] : copyuvm 수정

이전 단계에서 미뤄둔 inc_refcount를 추가해야 한다. 이는 3단계에도 명세되어있다. 그리고 child process가 parent process의 page들을 복사하는 것이 아닌, 같은 주소로 접근하되 write권한을 없애야 한다. 또 수업시간에 교수님이 주신 힌트가 기억난다 - "저라면 write 비트를 없앨 때 PTE_W와 and 연산을 사용하겠습니다".

fork 함수에서 copyuvm을 호출하는 부분을 보면 아래와 같다.

```
// Copy process state from proc.
if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
  kfree(np->kstack);
  np->kstack = 0;
  np->state = UNUSED;
  return -1;
}
```

child process(np)의 pgdir은 parent의 메모리를 pgdir을 참조하여 parent의 size만큼 복사해온다.

```
// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PS          0x080   // Page Size
```

일단 flag가 뭔지 살펴봐야하는데, 이는 mmu.h에 위와 같이 정의되어있다. 따라서 copyuvm에서 모종의 방법으로 해당 비트를 만져줘야 할 것이다.

```
// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
```

주석에 의하면, copyuvm은 parent의 ptable을 복사하여, 그의 사본을 child에게 준다 한다.

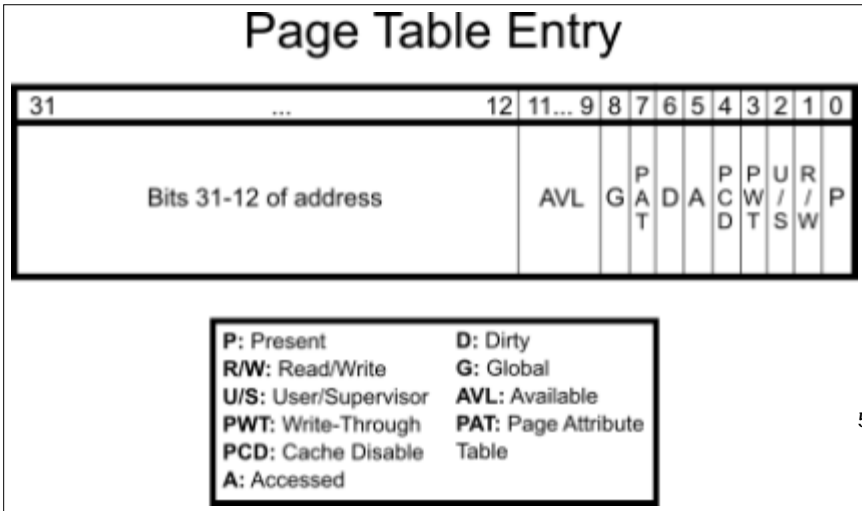
copyuvm함수를 좀 더 자세히 들여다 볼 필요성을 느껴서 이를 살펴보겠다.

```
== copyuvm ==============================================================================
pde_t *d;
pte_t *pte;
uint pa, I, flags;
char *mem;
d = setupkvm(), d == 0이면 return 0
setupkvm은 pde_t(=어떤 ptable의 첫주소)를 return한다.
for( i = 0; i < proc의 메모리 상 크기; i += PGSIZE(=4096) ){
  pte = walkpgdir(pgdir(=pde), i, 0), pte == 0이면 패닉;
```

`walkpgdir` (1735) mimics the actions of the x86 paging hardware as it looks up the PTE for a virtual address (see Figure 2-1). `walkpgdir` uses the upper 10 bits of the virtual address to find the page directory entry (1740). If the page directory entry isn't present, then the required page table page hasn't yet been allocated; if the `alloc` argument is set, `walkpgdir` allocates it and puts its physical address in the page directory. Finally it uses the next 10 bits of the virtual address to find the address of the PTE in the page table page (1753).

walkpgdir은 가상주소에 대한 PTE를 조회할 때 x86의 페이징 하드웨어를 모방한다. walkpgdir은 pde를 찾기 위해 가상주소의 상위 10비트를 사용한다. pde가 없으면, 필요한 pte가 아직 할당되지 않은 것이다. alloc argument가 설정되면 walkpgdir은 이를 할당하고 물리주소를 페이지 디렉토리에 넣는다. 마지막으로, 가상주소의 다음 10비트를 사용하여 페이지 테이블에서 pte를 찾는다.

walkpgdir은 pte의 주소를 return한다.



5)

pte는 위와 같은 구조로 되어있다. la의 상위 20비트를 가지고있고, 그 아래는 플래그비트들이다.

if(!(pte참조 & Present플래그))면 (=pte참조의 Present 플래그가 0이면) 패닉;

pa = PTE_ADDR(pte참조)

flags = PTE_FLAGS(pte참조)

```
mmu.h:#define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
kh@ThinkBook-Plus:~/os/hw4$
```

```
mmu.h:#define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF)
kh@ThinkBook-Plus:~/os/hw4$
```

PTE_ADDR, PTE_FLAGS는 mmu.h에 정의되어있다.

PTE_ADDR은 pte참조와 ~0xFFF의 and 연산 결과이다.
~0xFFF는 32비트에서 2진수로 1111 1111 1111 1111 1111 0000 0000 0000이다.
이름처럼 and연산으로 pde+pte만 유의미하게 남기려는 듯 하다.

PTE_FLAGS는 pte참조와 0xFFF의 and 연산 결과이다.
0xFFF는 0000 0000 0000 0000 0000 1111 1111 1111이다.
이 역시 이름처럼 and 연산으로 flag 12비트만 유의미하게 남기려는 듯 한다.

5) https://wiki.osdev.org/images/6/60/Page_table_entry.png

mem = kalloc()이 0이면(=kalloc 실패하면)
goto bad
memmove(mem(=kalloc으로 return받은 어떤 free했던 페이지의 주소), P2V(pa(=PTE_ADDR로 유의미하게 남긴 pde+pte), PGSIZE(=4096))

memmove는 string.c에 정의되어있다. dst <- src로 n만큼 메모리를 복사(이동)하는 함수이며, dst의 주소를 return 한다.
위의 라인은 pte의 상위 20개 비트로 받은 pte를 가상주소화 시킨 주소에서, kalloc으로 받은 주소로, n만큼 메모리를 복사한다는 뜻이다.

mappages(d(=pde), (void*) i, PGSIZE, V2P(mem(=kalloc받은 주소)), flags) 가 0 미만이면
kfree(mem), goto bad

```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
```

mappages는 vm.c에 정의되어있는데, 주석을 읽어보면 pa에서 시작하는 물리적 주소를 참조하는 va에서 시작하는 가상주소에 대한 pte를 생성한다 한다. 성공시 0, 실패시 -1을 return한다.

}
return d(=어떤 ptable의 첫 주소)

bad:
....
==========

이런 구조로 되어있다. 함수와 변수들을 정리해보자면 아래와 같다.

형식	변수명	함수명	return 값
pde_t *	d	setupkvm	child process의 새로운 pgdir의 주소(=ptable의 첫 주소)
pte_t *	pte	walkpgdir	argv로 받은 pgdir에서 찾은 pte
uint	pa	PTE_ADDR	pte에서 상위 20비트(=실질적인 주소)
uint	flags	PTE_FLAGS	pte에서 하위 12비트(=플래그들)
char *	mem	kalloc	할당받은 빈 메모리
		memmove	pa에서 에서 dst(mem)으로 복사하고, dst의 주소
		mappages	어떤 pgdir에서 pte를 등록하고, 성공시 0

fork함수로 돌아와서 생각해보자. np->pgdir은 copyuvm(curproc->pgdir, curproc->sz)이고, np->pgdir은 return한 d(=어떤 ptable의 첫 주소)가 된다. 이 때 copy가 되는 것이 아닌, parent의 pgdir을 쓰기 권한이 없이 읽기만 하려면 어떻게 해야겠는가? 위의 정리된 함수와 변수들과 같이 생각해보자.

우리는 프로세스마다 각각의 pgdir을 가지고있는 것을 알고있기에, d가 존재함을 생각해낼 수 있다. pte는 child의 pgdir에 등록해야하기에 필요하다. 또한 pa, flags는 mappages로 pte를 등록하는데 필요하기에 필요하다.

실행되지 않아야 할 부분들은 새로 할당받은 mem 관련한 부분들이다. parent를 참조해야하기에 mem을 kalloc으로 할당할 필요도, memmove로 pa에서 mem으로 복사 할 필요도 없다. 그리고 애초에 메모리를 kalloc으로 할당하지 않았기에 mappage의 오류처리 분기에서 kfree도 실행될 이유가 없다. mem이 사라졌기에, mappages에서 등록할 pte 역시 pa가 되어야한다.

새롭게 실행되어야 할 부분은, child는 parent의 pte를 참조해야하기에 flag를 조작하여 쓰기 권한을 없애야 한다. 다음으로, ptable에 변화가 생겼으므로, lcr3를 호출해줘야한다. 과제의 배경지식에서 lcr3은 pa를 받아야함을 알게되었다.

```
copyuvm(pde_t *pgdir, uint sz)
{
  /*** this annotation should not be execute ***/
  // this annotation is newly added
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  /*** char *mem; ***/

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    *pte &= (~PTE_W);
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    /***
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    ***/
    if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0) {
      /*** kfree(mem); ***/
      goto bad;
    }
    inc_refcount(pa);
  }
  lcr3(V2P(pgdir)); // newly added
  return d;
```

위의 생각대로 수정된 copyuvm이다.

[4 단계] : page fault handler 구현

trap.c의 trap 함수를 수정하여 pagefalut가 발생할시 vm.c의 pagefault 함수를 실행하도록 해야한다. 3차 과제를 진행하면서 traps.h에 trap 과 interrupt에 대한 정의들이 있는 것을 알게 되었다. 이를 다시 한번 보고 가겠다.

```
#define T_SEGNP         11      // segment not present
#define T_STACK         12      // stack exception
#define T_GPFLT         13      // general protection fault
#define T_PGFLT         14      // page fault
// #define T_RES        15      // reserved
#define T_FPERR         16      // floating point error
#define T_ALIGN         17      // aligment check
```

trapno가 14일 때 pagefault라 정의되어 있다. 그런데 trap 함수를 보면, 이에 대한 switch 분기가 없다.

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_PGFLT:
    pagefault();
    break;
  case T_IRQ0 + IRQ_TIMER:
```

그래서 추가해주었다.

이제 vm.c에서 새롭게 정의되어야 할 pagefault 함수를 생각해보자. 과제에는 다음과같은 명세들이 있다.

> rcr2() 함수를 호출하여 page fault 가 발생한 VA 읽어 들인다.
> 해당 가상 주소의 page table entry 확인후 PA 찾는다.
> PA 를 이용하여 해당 메모리 페이지의 reference counter 값 확인
> reference counter 가 1 보다 큰 경우 새로운 페이지를 할당 받아서 복사
> reference counter 가 1 인경우 현재 pte 의 write 권한만 추가

위에서 copyuvm 함수를 자세히 살펴본 보람이 있었다. 사용해야 하는 함수들을 copyuvm을 살펴보면서 이미 익숙해지니, 명세에 적힌 순서대로 코드가 술술 나왔다.

```
void pagefault(void){
    // rcr2를 통해 pf가 발생한곳의 va를 불러온다
    uint pgflt_va = rcr2();

    // va가 속한 pte를 가져온다
    struct proc *curproc = myproc();
    pte_t *pte = walkpgdir(curproc->pgdir, (void*)pgflt_va, 0);

    // pa와 rc를 얻는다
    uint pa = PTE_ADDR(*pte);
    uint rc = get_refcount(pa);

    // 해당 pa가 child에 의해 참조되는 경우
    if(rc > 1){
        // child에 새로운 page 할당
        char *mem = kalloc();
        // 메모리 parent -> child으로 복사
        memmove(mem, (char*)P2V(pa), PGSIZE);
        // pte에 or 연산으로 flag 지정하여 등록
        *pte = V2P(mem) | PTE_P | PTE_U | PTE_W;
        // 새로운 페이지를 할당받았기 때문에, refcount 감소;
        dec_refcount(pa);
    }
    // 참조되지 않는 경우
    else if(rc == 1){
        //현재 pte에 write flag만 켜준다
        *pte |= PTE_W;
    }

    lcr3(V2P(curproc->pgdir));
}
```

pagefault handler는 위와 같이 만들었다.

```
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15436
echo           2 4 14316
forktest       2 5 8760
grep           2 6 18280
init           2 7 14936
kill           2 8 14404
ln             2 9 14300
ls             2 10 16868
mkdir          2 11 14424
rm             2 12 14404
sh             2 13 28460
stressfs       2 14 15336
usertests      2 15 62836
wc             2 16 15860
zombie         2 17 13984
console        3 18 0
$ kh@ThinkBook-Plus:~/os/hw4$
```

여기까지 만들고, make하니 일단 오류는 없다.

4. Test

kh@ThinkBook-Plus: ~/os/h

```c
#include "types.h"
#include "stat.h"
#include "user.h"

int main(void) {
    int p;

    printf(1, "** parent ** pid : %d\n", getpid());

    printf(1, "1. (in parent) before fork\n");
    printf(1, "free pages : %d\n\n", getNumFreePages());

    p = fork();

    if(p < 0){ // fork error
        printf(1, "fork error\n");
    }
    else if(p > 0){ // parent process
        wait();

        printf(1, "3. (in parent) after termination of child\n");
        printf(1, "free pages : %d\n\n", getNumFreePages());
    }
    else{ // child process
        printf(1, "** child ** pid : %d\n", getpid());
        printf(1, "2. (in child) after fork\n");
        printf(1, "free pages : %d\n\n", getNumFreePages());
    }

    exit();
}
```

과제의 명세에 만들라 했던 getNumFreePages syscall을 적절하게 써서 만들었다.

```
$ test
** parent ** pid : 3
1. (in parent) before fork
free pages : 56734

** child ** pid : 4
2. (in child) after fork
free pages : 56666

3. (in parent) after termination of child
free pages : 56734

$ _
```

free pages가 줄고 느는 것을 볼 수 있었다.

5. Trouble shooting

과제의 명세도 상세하고, 두 번이나 과제에 대해 이런저런 힌트를 수업시간에 알려주셔서, 만들면서 문제될 것이 딱히 없었다.

6. etc

지난번 3차과제를 하면서, xv6가 round-robin방식 스케줄러를 가지고있다는 것에 대해 timer interrupt 관련된 코드를 도무지 찾을수가 없어서 교수님께 물어보니, 어셈블리 코드로 존재한다고 말씀하셨다. 그래서 gpt도 돌려보고 구글링도 해보고 했지만, 몇주가 지난 본 보고서를 작성하는 지금까지도 납득이 안갔었다.

이번 과제를 수행하면서 을 많이 참고했는데, 이도 메뉴얼에서 찾아보니

Real world

The xv6 scheduler implements a simple scheduling policy, which runs each process in turn. This policy is called *round robin*. Real operating systems implement more sophisticated policies that, for example, allow processes to have priorities. The idea is

이제야 받아들이게 되었다. 뭐 만든사람들이 직접 round robin이라 하는데 여기에 대해 다른 의견을 제시할 수가 있겠는가.