<swap entry>

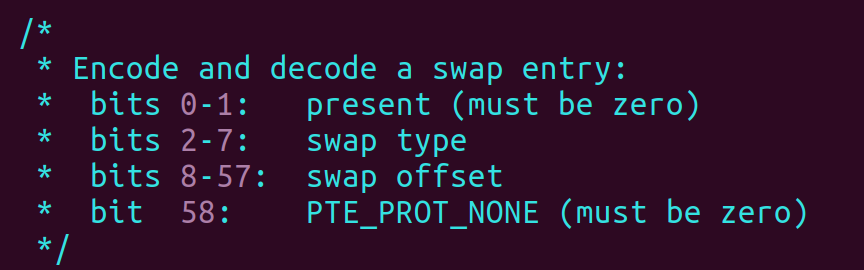
1. XArray를 사용하는 방식

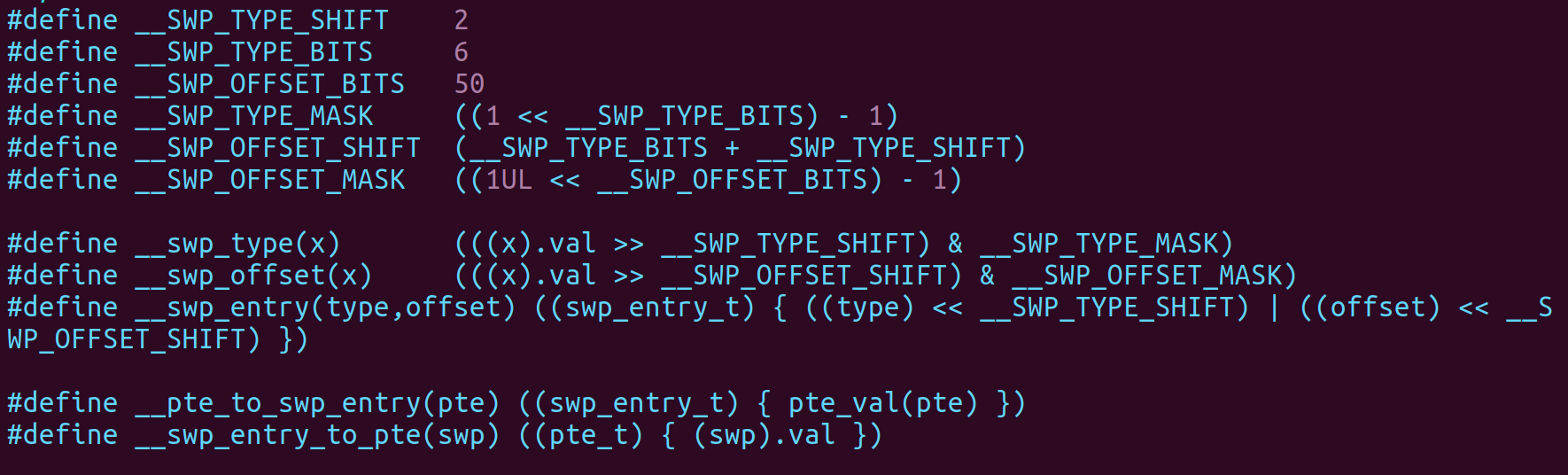
2. swap\_entry\_t 구조 ([Q1]t란?)

[Q2]우리는 ARM64 아키텍쳐? ㅇㅇㅇ

[Q3]구조에서 0, type이 의미하는 것은? -> type은 swap 영역 구분, 0은 unmapping상태로 이 페이지에 접근하면 fault에러가 발생한다?

[Q4] not used는 내 맘대로 써도 되는 공간? ㅇㅇ





(arch/arm64/include/asm/pgtable.h)

mask는 왜 쓰는건지? 아마 비트수 넘어가는 값이 이상하게 나오면 없애려고?

pte <-> swp\_entry간 구성은 똑같은 거?

pte\_none(pte): (!pte\_val(pte)) 로 pte가 0이면 1이 되는 값 -> pte가 있는지 초기값인지

pte\_present(pte): (!!(pte\_val(pte) & (PTE\_VALID | PTE\_PROT\_NONE))) 으로 pte 값의 valid bit와 prot\_none bit가 설정되어있는지 확인 (VALID: PTE의 bit1, PROT\_NONE: PTE의 bit 58)

이 둘 다 없다면 swap out 된 pte라는 것

entry관련 함수들은 define된 ARM64전용으로 써도되고, swapops.h에 선언된 걸 써도 되는거?

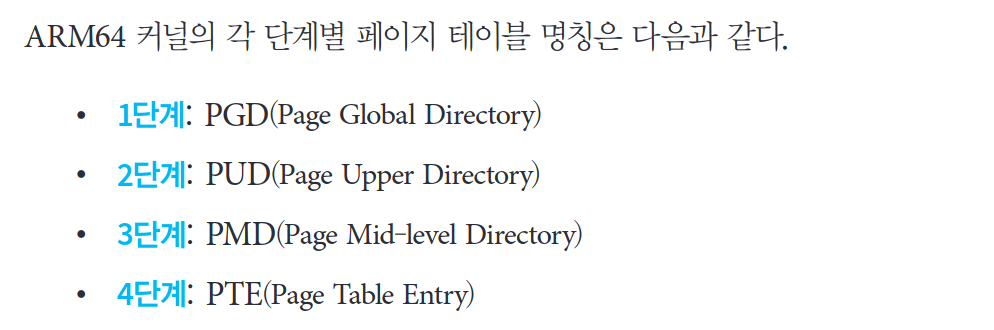
include/linux/mm\_types.h에 구조체 존재

어떤 application의 page인지는 주소+offset으로 아는? 어떻게 아는거지 entry에는 어떤 process의 swap\_page라는게 없는데

- Swap PTE 엔트리 식별

!pte\_none(pte) && !pte\_present(pte)면 swap된 pte (none 설정도 아니고 present 설정도 없는 경우)

none 설정이란..? pte bit에는 present, protect, modified 아닌지?



## \* Anonymous Page

익명 페이지는 커널로부터 프로세스에게 할당된 일반적인 메모리 페이지이다.

즉, **익명 페이지는 힙을 거치지 않고 할당받은 메모리 공간**이다.

(힙도 익명 페이지이다. malloc, new 같은 메모리 할당자는 익명 페이지에서 일부 메모리를 잘라 할당 받는것이다.)

먼저 ‘익명’ 이라는 뜻은 파일에 기반하고 있지 않은(파일로부터 매핑되지 않은) 페이지라는 뜻이다.

페이지가 파일에 매핑되어 있다면, 그 메모리는 파일 내용을 담고 있을 것이다.

하지만 익명 페이지는 파일에 매핑되어 있지 않았기 때문에 0으로 초기화된 값을 담고 있다.

프로세스가 mmap()으로 커널에게 익명 페이지를 할당 요청하게 되면, 커널은 프로세스에게 가상 메모리 주소 공간을 부여하게 된다.

부여된 가상 메모리 공간은 아직까지는 실제 물리 메모리 페이지로 할당되지 않은 공간이다.

부여된 가상 메모리는 메모리 읽기 쓰기시, 다음과 같은 커널 도움을 받아 zero 페이지로 에뮬레이션 되거나, 실제 물리 페이지로 매핑된다.

1. 프로세스가 그 메모리 공간에 읽기 작업 시, 커널은 zero로 초기화된 메모리 페이지 (file-backed page with /dev/zero)을 제공한다.
2. 프로세스가 그 메모리 공간에 쓰기 작업 시, 커널은 실제 물리 페이지를 할당하고 write된 데이터를 보관한다.

익명 페이지는 private 또는 shared로 할당받을 수 있다.

프로세스의 힙과 스택이 private로 할당된 anonymous page이다.

shared는 프로세스간 통신을 위해 사용되는 anonymous page이다.

익명 페이지를 할당 받으려면 다음과 같은 코드가 이용된다.

void\* ptr = mmap(NULL, size, PROT\_READ | PROT\_WRITE, MAP\_ANONYMOUS|MAP\_SHARED, -1, 0);  
// anonymous page는 fd가 -1이여야 하며, offset은 0이여야 한다.  
// 이후 익명 페이지를 읽으면 0으로 초기화된 값이 읽히며,  
// 익명 페이지를 쓰면 실제 물리 메모리 페이지가 할당된다.

<swap 1>

- swap 매커니즘

1. swap cache

2. Frontswap - zswap (memory, tmem) (front-end, 보통 synchronous) [Q1] tmem?

3. Swap space - hdd, ssd, ram (back-end, 보통 asynchronous)

화

\*clean anon page

1. swap 영역이 없어 swap을 할 수 없어, swap X

-Frontswap

1. front-end에 만든 swap 시스템

2. swap할 페이지에 대해 synchronous 저장이 가능할 정도로 고속 swap 장치 사용

3. tmem을 사용함. (transcendent memory) => DRAM 메모리 또는 persistent 메모리

\*Transcendent memory: 부족한 RAM 개선을 위한 새로운 메모리 관리 기술.

리눅스 커널에서 synchrnous 접근이 가능한 빠른 메모리, 간접 주소 지정 방식, 가변사이즈

--- Swap 영역 (Swap Backing Store) ---

- swap 파일 및 swap 디스크(block device)에 지정하여 사용

- network를 통해 마운트된 파일 시스템에 위치한 swap 파일 => NBD?

- ZRAM에 압축하여 저장도 가능

여기서 ZRAM과 NBD를 swap 장치라고 설명. 따라서 swap 영역이 본 작품에서 행하는 swap space가 될 것 같음.

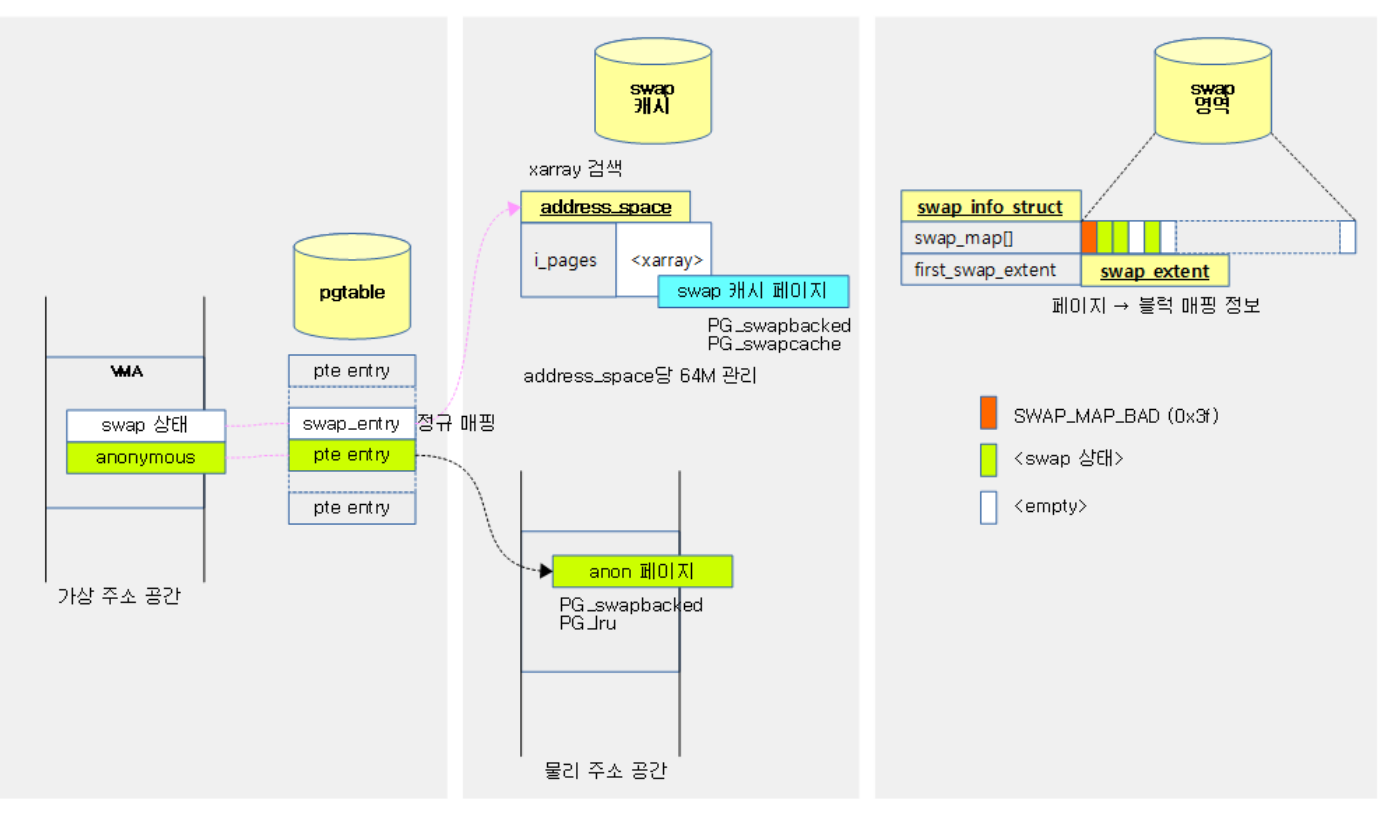
-swap영역

> swap disk 사용 (block device)

> swap file 사용

disk와 file의 차이점..?

영역 지정하는 방법? 그림만 봐서는 이해가 잘 안됨.



1개의 swap 영역(type=0)의 첫 번째 헤더 페이지를 제외한 그 다음 페이지(offset=1)를 swap 엔트리로 할당한 경우

* ARM64용 swap 엔트리 값 0x100이 페이지 테이블에 매핑되고, swap 캐시의 xarray에는 엔트리 값을 키로 페이지가 저장된다.

-swap 맵 초기화

inc\_cluster\_info\_page(swap\_info\_struct \*p, swap\_cluster\_info \*cluster\_info, pagenumber)

pagenumber: swap 영역에서의 몇번째 페이지인지 (= swap map에서 몇번째 페이지인지)

cluster: swap map을 cluster #0, cluster #1 등의 단위로 잘라서 한번에 관리?? page마다 관리하는 것이 아니고

일정 개수의 page를 묶어서 하나의 cluster로 관리 (e.g. 한 cluster에 256 pages) -> 실제 물리 주소에서 cluster단위로 나뉘는것?

swap 엔트리용 address\_space 관리

swapper\_spaces[type][offset]으로 관리.

swap\_entry에서 type과 offset 뽑아내서 2차원 배열로 관리

각각은 address\_space 배열이 지정됨.

swap\_info\_struct 구조체에서 swap 영역의 info가 들어있음. extern struct s\_i\_s \*swap\_info[]가 전역적으로 선언되어 모든 swap 영역에 대한 info를 이 swap\_info[] 배열에서 manage하는듯?

swapon을 하면 이 swap 영역이 사용되면서 SWP\_USED flag가 1로 set.

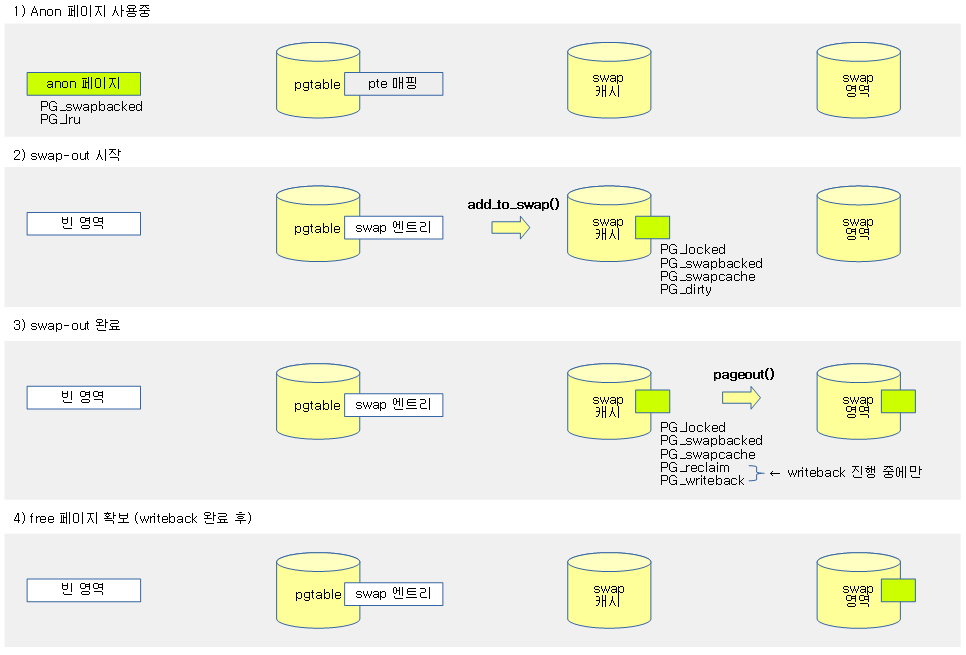
\*\* 이 SIS의 구조체에도 ZRAM인지 NBD인지 알려주는 구조 필요? (해당 영역이 ZRAM인지 NBD인지)

<swap -2- (Swapin & Swapout)>

- Swapout 과정: normal anon 페이지가 swap 영역에 기록된 후 페이지가 free되는 순서

=> normal anon page -> swapcache -> unmap -> write out -> free page

(anon 페이지란? Anonymous 페이지라고 파일과 관련없는 페이지, kernel에서 할당하는 page)



1. 사용중인 page를 swap-out 시작

2. add\_to\_swap() 함수를 통해 해당 page를 swap 캐시에 추가 (add\_to\_swao\_cache() 함수 호출) + 해당 page를 dirty 처리 => reclaim 과정에서 pageout()함수가 호출되어 swap 영역에 저장

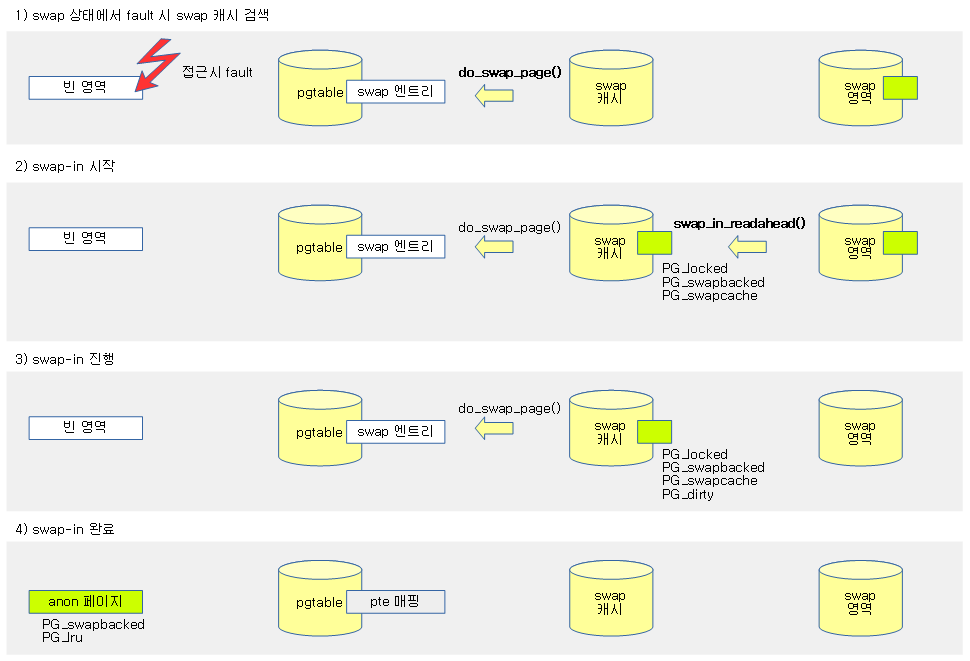
1)

\*reclaim의 정확한 동작?

\*shrink\_page\_list 에서 pageout()호출. shrink\_page\_list()가 pagelist 빌 때 까지 swap-out하는? shrink\_page\_list 함수의 정확한 동작 목적은? -> 페이지 회수 진행

add\_to\_swap\_cache() 에서 nr = 1UL << compound\_order(page); 로 설정하는데 이 뜻은?

1UL: 1 unsigned long (nr의 뜻과 compound\_order의 뜻은?)

- **Swapin 과정**: 새 페이지 할당 -> swapcache로 변경 -> swap 영역 (파티션) 에서 읽기 -> 매핑  
1. Swap된 상태에서 fault 시 swap cache 검색

2. swapcache에 없다면 swap-in시작 (swapin\_readahead()로 swap 영역에서 읽어옴)

1) swapin\_readahead(): swap\_use\_vma\_readahead가 true면 vma 기반 readahead 사용, 그렇지 않으면 cluster 기반의 readahead 사용.

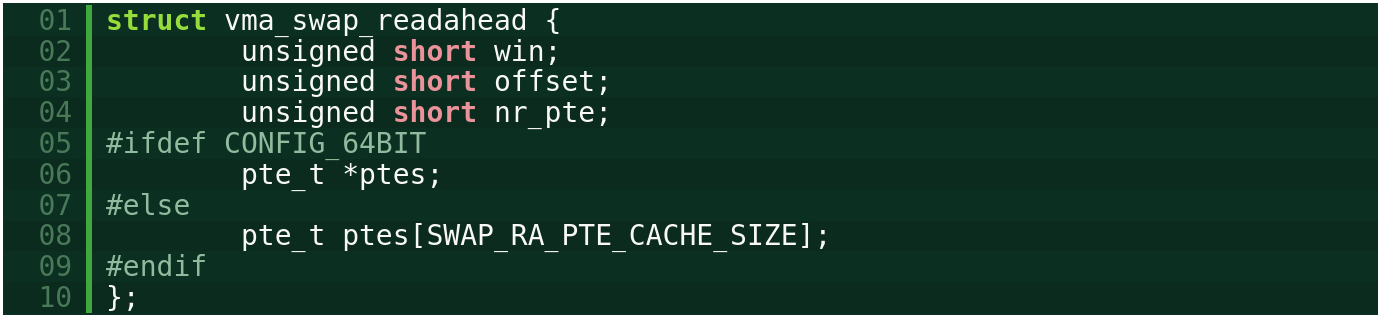
[함수]

**<SWAPOUT>**

1. swap 영역에 추가
   1. **add\_to\_swap()**
      1. swapout전에 PG\_lock과 PG\_uptodate가 설정되어있어야함.
      2. swap할 page의 swap entry를 얻어와서 그것을 key로 add\_to\_swap\_cahce를 통해 swapcache에 추가
      3. set\_page\_dirty(page)를 통해 dirty 설정.
      4. dirty된 page는 reclaim 과정에서 pageout()함수가 호출되어 swap 영역에 저장됨.
2. swap cache에 추가
   1. add\_to\_swap\_cache()
      1. swap entry 정보를 키로 anon page를 swapcache에 추가

**<SWAPIN>**

1. VMA 기반 readahead 방식 swap-in
   1. **swap\_vma\_readahead()**
      1. swap용 readahead 정보를 구성
      2. blk\_start\_plug(&plug)를 통해 blk\_plug 초기화하고 blk\_finish\_plug(&plug)가 끝나기 전까지 블록 디바이스에 submit 유보시킴
      3. pte 수만큼 pte들을 순회하며 swap entry 정보 기록된 pte에 대해 swapcache에서 page 읽어옴 (\_\_read\_swap\_cache\_async())
      4. 새로 할당된 page라면 (swap cache에서 찾지 못한 page) swap\_readpage(page, false)를 통해 swap영역에서 async하게 bio 요청하여 page 읽어옴. 그리고 할당한 page가 요청한 offset이 아니면 PG\_reclaim flag를 설정.



vma\_swap\_readahead 구조체

win: swapin 시 readahead할 산출된 page수, offset: fault pfn을 기준으로 readahead 시작 pfn offset, nr\_pte: swapin 시 readahead할 pte 수, \*ptes: fault 페이지 pte+offset에 해당하는 pte 주소

1. swapcache에서 페이지 읽어오기
   1. **read\_swap\_cache\_async()**
      1. swapcache swap entry에 해당하는 페이지를 읽어오는데, cache에서 찾을 수 없으면 swap 영역에서 읽어오고 swap 영역에서 읽어오는 page를 위한 새로운 swapcache를 할당해서 준비해둠. (page\_was\_allocated에 true값)
      2. 이 과정에서 \_\_read\_swap\_cache\_async() 함수 호출
      3. 만약 page\_was\_allocated가 true면 (swapcache에서 못찾아서 새로운 swapcache를 만듦) swap\_readpage(page, do\_poll)를 통해 swap 영역에서 읽어옴. do\_poll 인자가 false면 비동기로 읽어옴.
   2. **\_\_read\_swap\_cache\_async()**
      1. 실제로 swapcache에서 find\_get\_page() 함수를 통해서 page를 찾아보고, 그 page가 없다면 alloc\_page\_vma를 통해 새로운 page를 할당.
      2. 할당할 page를 cache에 추가할 준비를 하고, add\_to\_swap\_cache를 통해서 cache에 새로운 page를 추가. 추가가 완료되면 새 swapcache page를 return
      3. cache에 추가가 실패하면 메모리가 부족하지 않는 한 다시 반복.

* bio 요청: block I/O 요청인듯?

1. swap영역에서 읽어 swapcache에 저장하기
   1. **swap\_readpage()**
      1. swapcache page의 private 멤버를 통해 swap\_info\_struct (sis)를 읽어옴. (\*sis = page\_swap\_info(page))
      2. 블록 디바이스를 통해 사용되는 swap 영역: ret = bedv\_read\_page()함수를 통해 읽어옴. 이 때 ret가 0이 아니라면 제대로 읽은 것으로 PSWPIN 카운터를 증가시킨 후 return 0 (return 0가 swap\_readpage() 성공적)
      3. bio 요청: bio = get\_swap\_bio()를 통해 page에 읽어올 준비 후 PSWPIN 카운터 증가시키고, bio\_get(bio)를 통해 실제 요청하여 page 읽어옴. 그 뒤 synchronous 설정된 경우 싱크 맞을 때까지 대기.
2. swapcache page 찾기: fault된 pte 값에 swap entry가 기록되어 있으면 do\_swap\_page()를 호출하여 swapcache 영여에서 찾아 anon 페이지로 매핑.
   1. **lookup\_swap\_cache()**
      1. find\_get\_page() 함수에서 인자로 swap\_adrdress\_space(entry)와 swp\_offset(entry)를 통해 page를 cache에서 찾아냄.
      2. page를 찾아냈다면 readahead 여부를 조사하고 readahead를 한다면 관련된 처리를 하고 page 반환
      3. swapcache stat 관리
         1. INC\_CACHE\_INFO(find\_total): swapcache의 find\_total stat 증가 (hit + miss인듯?)
         2. INC\_CACHE\_INFO(find\_success): swapcache의 find\_success stat 증가, page를 찾았을 때 하는것으로 보아 hit stat인듯?
3. swap 영역에 기록: anon page를 swapcache에 저장한 후 dirty를 설정하고 빠져나와 mapping 해제. 그 뒤 pageout()으로 swapcache를 swap 영역에 기록.
   1. **swap\_writepage()**
      1. dirty상태의 swapcache를 swap영역에 기록 (실제 writeback 작업이라고 보면 될듯?)
      2. ret = \_\_swap\_writepage() 를 통해 실제 page를 swap 영역에 기록요청. sync/async 기록이 완료되면 end\_swap\_bio\_write()함수를 호출하여 writeback이 완료되었음을 알림. (page flag를 통해. 아마 PG\_writeback을 clear하는듯?)
   2. **swap\_set\_page\_dirty()**
      1. page의 private 영역을 통해 sis 정보를 얻어옴.
      2. SWP\_FS가 설정되어 있으면 드라이버 제공의 함수를 통해 dirty 표식
      3. 그 외의 일반 swap영역 (아마 여기일듯?): \_\_set\_page\_dirty\_no\_writeback() 함수를 호출하여 dirty 설정.
   3. **\_\_set\_page\_dirty\_no\_writeback()**
      1. page를 인자로 받아와 해당 page의 dirty flag가 설정되어 잇지 않으면 TestSetPageDirty(page)를 통해 dirty 설정 후 1 반환, 이미 dirty 표식 되어있다면 0 반환

내가 해야되는 일 (ARCH = ARM64), 붉게 표시한건 현 정책에서 알아야 할 것

1. **NBD에 있는 page들을 bulk prefetch 해오는 것. (swapin)**
   1. nbd에 있는 page들을 순회하면서 내가 실행한 어플리케이션 (process라고 보면 되나?)에 해당하는 page들을 한꺼번에 swapin해옴.
      1. 해당 process의 page라는 것은 어떻게 아는지? struct page 구조체를 봄
         1. 이건 page가 들어있는 주소를 보고 알아야하는건가..? swapentry의 type + offset? 아닐거같은데..
         2. swap 영역에 어떤 process를 위한 swap 영역이라는것도 없을듯. 그게 되면 locality 고민을 할 필요없지.
         3. page마다 struct page 구조체가 존재. mm\_types.h에 참조. map count 보면 몇개의 process가 참조하는지 알 수 잇음 (공유 페이지인지 private인지) 그럼 이 struct page는 swp\_entry로 바뀌어도 존재하는 건가? swap 영역에는 page가 어떤식으로 적혀있는거지?
      2. 그냥 무작위로 빈 공간에 swapout한다면 locality가 없어서 bulk prefetch할 때 page들을 검색하는데 시간이 오래걸린다고 했으니, swapout 코드도 고쳐야하는지?
      3. 현재 swapin은 readahead 방법으로만 하는 함수들만 있는 듯 한데, (swapin이 page fault가 났을 때에 그 때 readahead를 하는상황밖에 없는 것? 즉, 현 정책에서는 swapin == readahead라고 볼 수 있는지?ㅇㅇㅇ)
         1. 이 readahead는 page fault라는 조건하에서만 실행? 그러면 process가 동작했을 때 무조건적으로 prefetch하고 실행하도록 코드 수정?
         2. 그럼 readahead 함수는 그대로 두고 bulk prefetch를 위한 함수를 새로 만들어야 하는? 이 함수를 만들면 이 함수가 bulk prefetcher machine이라고 볼 수 있는지?
   2. a-ii 관련 내용. swapout 코드를 고친다면 물리적으로 주소를 고정시켜서 보내야하나? 그건 너무 하드 코딩인데..
      1. 그러면 현재 nbd에 남아있는 공간들을 확인하고 process당 몇개의 page까지 swapout 할 수 있게 하는 그런식? process마다 다른 값?
      2. process마다 고정된 base addr를 갖도록? 그러면 커널에서 새로운 process를 swapout할 때마다 base addr를 계산하도록..?
         1. e.g. OS 시작되고 첫 process를 실행시킬때 base addr를 1000이라고 하면, 해당 process는 물리주소 1000 까지 swapout할 수 있고, 다음 process는 2000 을 base addr로 갖도록.
         2. kernel에서 base\_addr\_std 라는 변수로 1000을 초기값으로 가지고있고, proc1에게 메모리 할당하면서 해당 base\_addr\_std를 1000+해당 process에게 할당된 크기 => base\_addr\_std를 2000으로 update. proc2의 base\_addr\_std 를 2000으로 시작하고, proc2가 시작되면 base\_addr\_std 3000으로 update하는식.
         3. 이렇게 하려면 process간 할당하는 swapout page 크기를 동일하게 하는것이 구현하기 편하긴 할것. 하지만 단편화 생길듯 process마다 swapout하는 정도가 다르니까 swapout 많이 안하는 process는 공간낭비. swapout 많이 하는 process는 결국 locality도 떨어짐 (e.g. proc2가 2000~2999를 쓰는데 더 swapout을 하려고하면 proc3이 실행된 전제하에 base\_addr\_std는 4000이고, proc2에게 4000~4999를 할당하면 locality 감소.) - 내생각이긴함 ㅠ
         4. 근데 이렇게만 해도 swapout을 너무 많이 고치게 되는 것 아닌지..? 할 수 있을까..
      3. 이렇게 되면 해당 swap space가 지워질 때는 base\_addr\_std가 어떻게 될지..
      4. 현재는 어떤식으로 되는지? 현재는 그냥 swap 영역을 할당하고 거기에 아무 process나 swapout하는 형식?
   3. 실제 swapin 해올 때, 그러니까 bulk하게 prefetch를 할 때, 원래는 한 페이지마다 읽어서 swapcache에 넣는 것 같던데, 이렇게 안하고 어떤 buffer에 페이지를 많이 담아서 swapin을 해와야하나..? 그럼 read\_swap\_page하는 방식도 많이 바뀌어야 하는건지?
      1. 아니면 swapcache 밑에 또다른 cache같은 걸 만들어서 계층을 하나 더 만드는 느낌..? 아 이건 아닌거같은데
      2. 뭔가 buffer? bus?를 통해서 page를 한번에 읽어오는 걸 어떻게 할지.. swap영역에서(nbd에서) 해당 process에 맞는 page들을 buffer에 한 번에 쭉 담고 그 buffer를 local로 전송?
         1. 이렇게 전송한다 쳐도 그 buffer를 process의 메모리에 어떻게 넣나?
2. **ZRAM에 있는 page들을 background로 NBD로 swapout 하는 것.**
   1. 현재 정책에서는 page가 ZRAM에 있는지 NBD에 있는지 식별 불가?
      1. swap\_entry의 빈 bit에 현재 page가 ZRAM / NBD에 있는지 확인하는 metadata 생성?
         1. 58 bit가 PTE\_PROT\_NONE 인지 아닌지?
         2. 그냥 59 bit에 박기? (59bit가 0이면 ZRAM에, 1이면 NBD에 있다는)
         3. swap type(bits 2 ~ 7)은 swap영역을 알려준다고 했는데, 이는 swap영역의 종류가 아니라 단순 이때까지 할당한 swap영역들 중 몇번 swap 영역에 있는지를 알려주는?
      2. 만약 metadata 생성하면 swapout할 때 ZRAM으로 보내면 59bit 0 NBD로 보내면 1? 이런식으로 바꾸게끔 해야하나?
         1. swapout할 때 보내는 영역이 ZRAM인지 NBD인지 어떻게 아나? swap\_type에 그 정보가 잇나? swap\_type은 2-a-i-3의 내용만 가지고 있는거 아닌지?
         2. sis (swap\_info\_struct)에 새로운 metadata 생성해서 ZRAM / NBD / neither 를 알 수 있도록 해야하나? 그래서 swapout할 때 해당 영역의 sis를 참조해서 어디로 swapout하는지 알 수 있도록?
         3. sis 새로운 metadata를 만들게 되면 mkswap이나 swapon처럼 파티션 할당할 때 해당 metadata도 작성하도록 만들어야하는?
         4. SWAP-1-에 SWAPON 내용에 swap\_info\_struct를 할당 및 초기화하는게 있음, alloc\_swap\_info() 여기서 metadata를 고쳐주는 식?
         5. 그리고 swapout 할 때 그 59 bit을 ZRAM으로 보내면 그대로 두고 NBD로 보내면 1로 하는? 아니 근데 봐바 여기서 ZRAM을 0으로 두면 안되지않나 ZRAM을 0으로 두면 ZRAM에 있는것만 NBD로 보내려고 했는데 ZRAM이 아닌 다른 SWAP 영역에 있는 애들까지 NBD로 보낼거같은데..
            1. 그러면 ZRAM을 1로? 아니면 bit를 2개 쓰든가
            2. ZRAM이 1이면 1인 애들을 계속 모아서 NBD로 보내기를 하면 되긴 될거같음..
      3. swap 영역에 ZRAM과 NBD만 있는게 아님. 그러면 bits 60:59를 써서
         1. 00: ZRAM도, NBD도 아닌 영역 (기존의 영역들?)
         2. 01: ZRAM
         3. 10: NBD
         4. 11: reserved

이런식으로 되도록? 한비트만 써도 될듯? 사실 ZRAM이랑 NBD만 구별하면 되고 나머지는 안 건드릴거니까..

* + 1. 만약 이 메타데이터가 성공적으로 적용되면, page table을 돌면서 bits 59를 확인해서 0이면 1으로 swapout 하도록? 근데 이러면 pte를 확인해야하는 것 아닌지? 내가 이해하기로는 현재 사용중인 anon page면 pte로 관리하고 swap out 되어있는 page면 swap entry로 관리하는것 아닌가...
       1. 아니지 ZRAM에 있다면 이미 swapout 된 거라는건데, pte가 아니라 swap entry로 봐야지. 그럼 swapout된 page를 다시 nbd로 swapout한다는 느낌? ZRAM은 local이라 메모리가 적고, nbd는 아무래도 메모리용량이 더 크니까. 그럼 이건.. page\_list에 swapout된 page들이 쭉 있나? 이런 swapout 된 page들은 어디서 접근 할 수 있는지?
       2. 그 process의 page table에 가면 해당 process에서 사용하는 page들을 참조하고 있는 entry들이 쭉 있을거임. 그 중에서 is\_swp\_entry로 일단 swp\_entry인지 보고 거기서 swp\_entry라면 ZRAM에 있는지 보면 되지않으려나? is\_swp\_entry() && !swp\_entry\_ZRAM\_NBD() 이런식으로 조건 걸면 될듯?
  1. scheduler에서 주기적으로 background로 보내도록 해야하는지? 애초에 background로 실행시키게 하는 방법은? 그냥 scheduler에서 시키면 이건 background가 아니지않나?
  2. 이것도 ZRAM\_background\_swapout을 위한 함수를 새로 만들어야 하는지?
     1. 구현해야하는 범위가 1)bg\_swapout하는 함수인지 2)그 함수를 background에서 정기적으로 쓰도록 하는 것까지 해야하는지?
  3. mapcount가 1인 (공유 page가 아닌) page들만 (private page) b.g에서 ZRAM -> NBD로 갈 수 있도록 (이유: )
  4. process마다 pagetable을 가지고 있고, 해당 page table을 pgtable walk를 하면서 그 process의 page들을 쭉 검색. 메인메모리에 있는 애들은 pte, swap영역에 있는 애들은 swap\_entry로 되어있을텐데 그 중 swp\_entry인 애들을 봄. 거기서 ZRAM인지 NBD인지 보고, ZRAM에 있다면 nbd에 쓰도록. 그리고 해당 pg를 참조하는 포인터들을 ZRAM에서 NBD로 화살표 바꿔주기 해야함. 근데 이 연결 끊고 다시 연결하는 과정에서 해당 page가 공유페이지라면, 다른 process를 실행할 때 영향을 줌. 그러니까 mapcount가 1인 page들만 (private page) ZRAM->NBD로가도록 함. 이 과정을 b.g로 실행?
     1. 근데 공유페이지라 다른 process에서 해당 page를 참조하려고 하는데, 아 해당 process의 page table에서는 그 page가 ZRAM에 있다고 mapping 되어있는데 실제로는 nbd로 가니까 문제가 생긴다 이말인듯!
  5. 실제 ZRAM에서 NBD로 page 옮기는 과정은?
     1. ZRAM에는 page가 압축되어있음. 그러면 이 page를 압축해제해서 읽어서 그 내용을 어떤 temp page에 저장한 다음에 그 page를 NBD에 write하는?
        1. 그럼 압축해제하는 함수는 이미 있을 듯? 기존의 ZRAM에서 swapin해오려면 그런 과정이 필요했을 거니까.
        2. 해당 함수 찾아서 내가 만들 daemon에서 함수 호출하고, page 저장하고 그걸 NBD에 적는?
     2. ZRAM에서 NBD로 옮겼다면 그 page를 참조하고 있는 swp entry의 정보도 바꿔줘야할듯.
        1. swp\_entry 59 bit를 0->1로?
        2. swp\_entry의 offset과 type도 다 바뀔걸 그러면 NBD의 swp\_entry 자체를 page table의 swp\_entry에다가 쓰는 방식은 어떤지?

\* ZRAM 파티션 위치: /dev/zram0, NBD 파티션 위치: /dev/nbd0 -> nbd는 휴대폰 부팅될 때마다 mknod해줘야할걸?

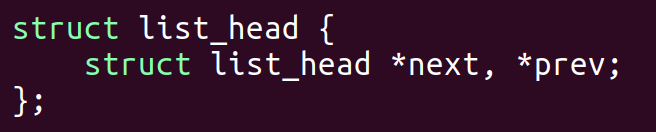
<관련 함수들>

* swapin & swapout의 실제 page가 읽히고 쓰이는건 page\_io.c에 함수들이 있는듯?

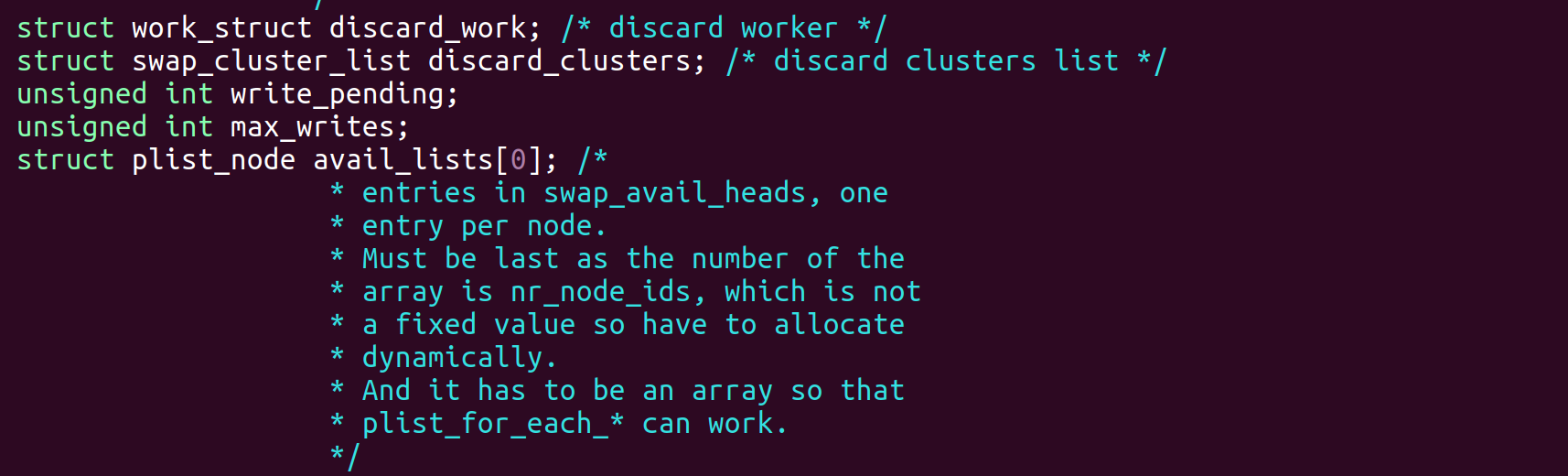
1. SWAPOUT
   1. swap\_writepage, \_\_swap\_writepage (swap\_writepage에서 호출)
      1. dirty 상태의 swap page를 swap영역에 기록
         1. swapout시 dirty가 아닌 page는 swapout안하나..? 어차피 swapin을 해도 해당 page의 원본정보는 swap영역에 있으니.. dirty가 아니라면 해당 원본정보를 바꿀 필요가 없었나? 기억이 잘 안나네.. (즉, swapin을 할 때 해당 page를 swapin해도 그 page는 swap영역에 남아있는지?)
2. SWAPIN
   1. swap\_readpage
      1. 실제 swap 영역에서 읽어온 페이지를 swap cache에 저장하도록 bio요청
      2. \_\_swap\_writepage와 함수 구조가 유사한 듯?
3. SWAPENTRY
   1. ARM64에서 swapentry관련 함수들(define으로 정의된 함수들 e.g. \_\_swp\_type, offset, entry)과 아키텍쳐 독립 (swapops.h 속 함수들) swapentry 함수들 둘 다 쓸 수 있는건지?

**<잡다한 define이나 구조체>**



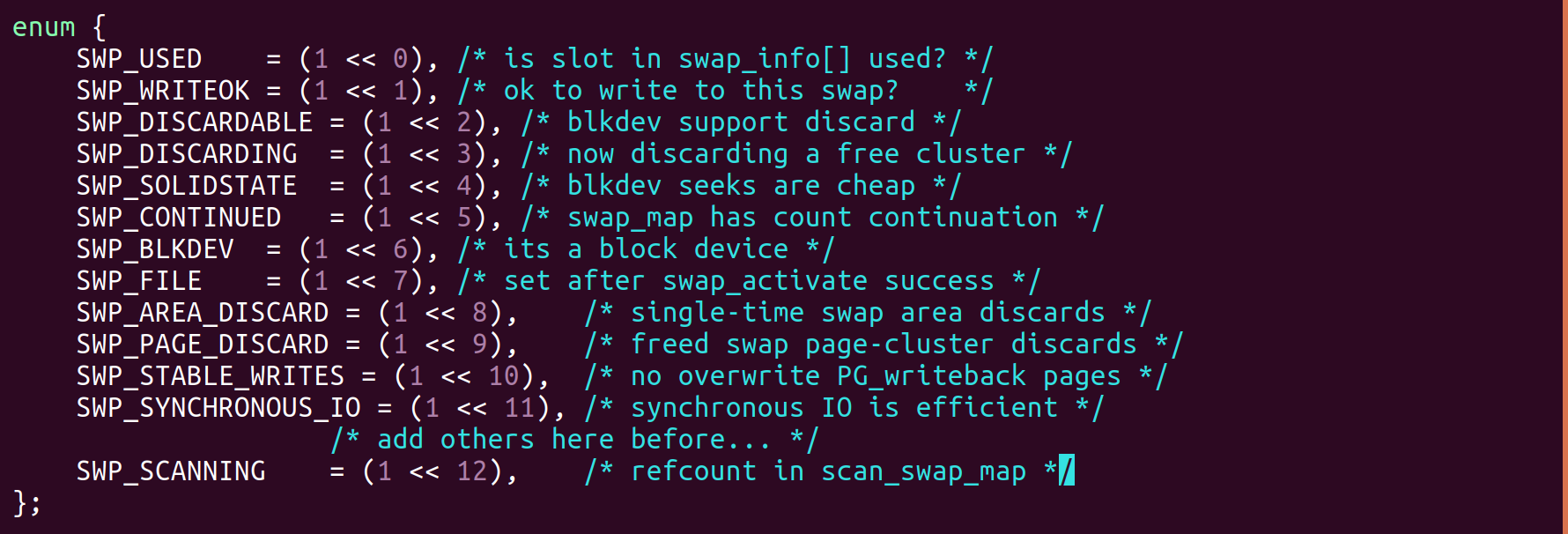






swap 영역마다 하나씩 사용된다.

* flags
  + swap 영역에 사용되는 플래그 값들이다. (아래 참조)



* SWP\_USED: swapon되어 사용 중인 swap 영역이다.
* SWP\_WRITEOK: swap 영역에 write 가능한 상태이다.
* SWP\_DISCARDABLE: swap 영역이 discard를 지원한다. (SSD)
* SWP\_DISCARDING: swap 영역에 discard 작업을 수행 중이다.
* SWP\_SOLIDSTATE: swap 영역이 SSD이다.
* SWP\_CONTINUED: swap 영역에서 usage 카운터가 0x3e를 초과하여 사용 중이다.
* SWP\_BLKDEV: swap 영역이 블럭 디바이스이다.
* SWP\_ACTIVATED: swap 영역이 swap\_activate 후크를 성공한 경우에 설정된다.
* SWP\_FS: 파일 시스템을 통한 swap 파일이다.
* SWP\_AREA\_DISCARD: 한 번만 discard를 지원한다.
* SWP\_PAGE\_DISCARD: 클러스터 단위로 자유롭게 discard를 사용할 수 있다.
* SWP\_STABLE\_WRITES: 체크섬등을 사용한 무결성 저장 장치에 부여한다.
* SCSI Data Integrity Block Device
* SWP\_SYNCHRONOUS\_IO: 동기 저장이 가능한 swap 영역이다. (RAM 등)
* SWP\_SCANNING: swap 영역을 검색중일 때 설정된다.
* prio
  + 영역의 사용에 대한 순서를 정하기 위한 우선 순위 값이다.
  + default 값은 -2부터 생성되는 순서대로 감소된다.
  + 사용자가 지정하는 경우 양수를 사용할 수 있다.
* list
  + swap\_active\_head 리스트에 사용되는 노드이다.
* type
  + swap 영역에 대한 인덱스 번호이다. (0~)
* max
  + swap 영역의 전체 페이지 수 (bad 페이지 포함)
* \*swap\_map
  + swap 영역에 대한 swap 맵으로 1페이지당 1바이트의 값을 사용된다.
  + 예) 0=free 상태, 1~0x3e: 사용 상태(usage counter), 0x3f: bad 상태
* \*cluster\_info
  + swap 영역에서 사용하는 모든 클러스터들을 가리킨다.
* free\_clusters
  + 사용 가능한 free 클러스터들을 담는 리스트이다.
* lowest\_bit
  + swap 영역에서 가장 낮은 free 페이지의 offset 값이다.
* highest\_bit
  + swap 영역에서 가장 높은 free 페이지의 offset 값이다.
* pages
  + swap 영역의 전체 사용가능한 페이지 수 (bad 페이지 제외)
* inuse\_pages
  + swap 영역에서 할당되어 사용중인 페이지 수
* cluster\_next
  + 높은 확률로 다음 할당 시 사용할 페이지 offset를 가리킨다.
* cluster\_nr
  + 다음 클러스터를 검색하기 위한 countdown 값
* \*percpu\_cluster
  + cpu별 현재 지정된 클러스터를 가리킨다.
  + 해당 cpu에 지정된 클러스터가 없는 경우 그 cpu의 값은 null을 가진다.
* \*curr\_swap\_extent
  + 현재 사용중인 swap extent를 가리킨다.
* first\_swap\_extent
  + swap 영역에 내장된 첫 swap extent이다.
* \*bdev
  + swap 영역에 지정된 블럭 디바이스 또는 swap 파일의 블럭 디바이스
* \*swap\_file
  + swap 파일을 가리킨다.
* old\_block\_size
  + swap 블럭 디바이스의 사이즈
* \*frontswap\_map
  + frontswap의 in-use 상태를 비트로 표기하고, 1 비트는 1 페이지에 해당한다.
* frontswap\_pages
  + fronswap의 in-use 카운터
* lock
  + swap 영역에 대한 lock이다.
* cont\_lock
  + swap\_map[]의 usage 카운터가 0x3e를 초과하는 경우에 swap 영역의 맵 관리는 swap count continuation 모드로 관리되는데 이 때 사용되는 swap count continuation 페이지 리스트에 접근할 때 사용하는 lock 이다.
* discard\_work
  + discard를 지원하는 SSD에서 사용할 워크이다.
* discard\_clusters
  + discard할 클러스터들이 담긴 리스트이다.
  + discard 처리 후 free 클러스터 리스트로 옮긴다.
* avail\_lists[0]
  + 노드별로 관리되는 swap\_avail\_heads[] 리스트의 노드로 사용된다.

**<진용이형 kernelstudy>**

1. struct page
   1. *unsigned long* flags: page flags
   2. *struct list\_head* lru: LRU list의 next & prev link
   3. *struct adress\_space* \*mapping: pointer to
      1. address\_space: file backed page
      2. anon\_vma: anonymous page
   4. *pgoff\_t* index: mapping 안에 offset
   5. *unsigned long* private: private data로 PageSwapCache라면 swp\_entry\_t에 사용됨.
   6. *atomic\_t* \_mapcount: userspace에 page가 mapping되었을 때, 해당 page가 page table에 의해서 몇번 참조되었는지.
      1. 근데 하나의 process의 page table에서 한 페이지를 여러 번 참조할 수는 없는가? \_mapcount가 1인 녀석이 무조건 하나의 process가 참조하는건 알겠지만 \_mapcount가 2 이상인 녀석이 무조건 여러 개의 process가 참조한다고 볼 수 있는건가?
      2. 물론 이걸 구별하기 힘들다면 \_mapcount == 1 인 page만 background로 보내는게 맞을듯
   7. *unsigned int* page\_type: kernel page type으로 pageslab도 userspace에 mapping 할 페이지도 아닌 page는 이 변수로 어디에 사용하는지 알 수 있음. page-flags.h에서 어떤 type들이 있는지 확인가능.
2. Swap out
   1. page 회수 path
      1. shrink\_page\_list(): 페이지 회수 진행
      2. add\_to\_swap(): 위에서 정리한 것 참고
      3. try\_to\_unmap(): page에 mapping되어있는 모든 page table과의 mapping을 제거
         1. 현재 page를 mapping 하고있는 pte를 모두 제거. 그 뒤 page\_mapcount(page)를 통해 unmap한 page의 \_mapcount를 보고 그 값이 0 즉, 아무 pte도 그 page를 보고 있지 않으면 성공. true 반환
         2. pte를 제거하면서 해당 pte가 swap됐다는 걸 알려주기위해서 rmap\_walk 함수를 통해 pte에 swp\_entry를 써줌.
      4. pageout(): dirty page에 대해서 실제로 swap 영역에 writeback하는 함수.
         1. dirty flag가 set되어있다면 swap slot에 write
      5. \_\_remove\_mapping: write 끝난 후에 swap cache를 삭제.
3. Swap in: page fault가 났을 때, pte에 swp entry가 적혀있다면 do\_swap\_page() 함수 호출 (fault가 나면 handle\_pte\_fault 함수가 호출되는듯 그리고 그 함수에서 do\_swap\_page 호출)
   1. do\_swap\_page()
      1. pte에서 swp entry를 읽음
      2. lookup\_swap\_cache() 호출해서 swap cache에서 한 번 찾아봄. 없으면 swapin\_readahead() 호출
   2. swapin\_readahead()
      1. vma 기반인지 cluster기반인지 식별하여 해당 swap\_vma(cluster)\_readahead 함수 호출
      2. swap\_vma\_readahead 는 위 참고.
   3. do\_swap\_page(): 읽어온 page로 page table에 pte 써줌

pte안에 pfn + protect bit들? pfn이 physical frame number니까 실제 물리적 주소일거고

[할일 구체화]

인자로 pid를 받는 syscall를 일단 먼저 만들자

1. page table을 돌면서 ZRAM에 swapout되어있는애 찾기
   1. page table walk하는 코드 참고
2. ZRAM에서 해당 swapout되어있던 page 내용 읽어오기 (swapin 코드 참고, swap\_readpage 참고)
   1. 이게 swapin하면서 자동으로 압축해제를 해주는지?? 그건 모르겠네
3. 읽어온 page를 nbd에 적음 (swapout 코드 참고)
   1. 원래 swapout할 때 get\_swap\_entry()를 통해 swap entry를 할당받아옴. 근데 이 get\_swap\_entry()는 priority에 따라서 자동으로 swap entry를 할당해줌.
   2. 이번 작품에서 prio 1이 ZRAM, prio 2가 nbd일건데 prio 1 ZRAM이 가득 차야 nbd로 swapout을 할 수 있도록 entry를 할당해줄것임.
   3. 따라서 기존의 swapout 코드로는 nbd의 swap entry를 받아올 수 없음.
   4. get\_swap\_entry()를 참고해서 prio에서 받아오는 것이 아닌 무조건 nbd에서 받아오도록 새로운 함수를 만드는?
   5. swap\_info\_struct 구조체(swap 영역의 정보를 담고 있는 구조체)에서 현 swap영역이 ZRAM인지 NBD인지 알게하는 새로운 멤버를 추가?

<syscall 선언하기>

sysctl 변수를 선언해서 해당 변수 + handler를 만듦. adb shell안에서 echo ‘pid’> /cat/proc/.. 해서 동작 확인, sysctl 변수로 선언하면 kernel에서 하드코딩을 해둬도 실시간으로 통신하면서 매개변수 값을 바꿀 수 있음

1. msm-google/include/linux 에 헤더파일 만듦
   1. 헤더파일 처음에 #ifndef \_LINUX\_FILENAME\_H, #define \_LINUX\_FILENAME\_H
      1. FILENAME: ksg\_declare.h
   2. 헤더파일 끝에 #endif
   3. 헤더파일에 extern int SysctlName 과 extern int SysctlName\_handler(struct ctl\_table \*table, int write, void \_\_user\*buffer, size\_t \*length, loff\_t \*ppos) 로 선언해둠
      1. sysctl name: zram\_to\_nbd
      2. sysctl handelr name: zram\_to\_nbd\_handler
2. msm-google/kernel/sysctl.c 에서
   1. #include <linux/HEADERFILE.h>
   2. static struct ctl\_table kern\_table[] = {} 내부에
      1. {

.procname = “sysctl\_name”,

.data = &sysctl\_name,

.maxlen = sizeof(unsigned int),

.mode = 0644,

.proc\_handler = sysctl\_name\_handler,

} 의 형식으로 선언

1. msm-google/mm/vmscan.c
   1. #include <linux/HEADERFILE.h>
   2. int sysctl\_name;
      1. 이 변수에 실제로 echo를 통해 pid를 넣을것임.
   3. int sysctl\_name\_handler(strcut ctl\_table \*table, int write, void \_\_user \*buffer, size\_t \*length, loff\_t \*ppos)두개 선언
   4. 해당 sysctl을 호출하면 자동으로 sysctl\_handler가 호출되어 일을 처리하는 형식이기 때문에 실제 syscall에서 어떤 일을 할지 이 handler에 정의해둠
   5. handler에서 필수적으로 지켜야 할 구조
      1. int rc = proc\_dointvec\_minmax(table, write, buffer, length, ppos);

if(rc)

return rc;

if(write) {

실제로 해야하는 일들 (처음에는 printk를 통해 해당 sysctl이 제대로 선언되어 있 는지 확인하는게 좋을듯?

}

return 0;

* 1. sysctl -> sysctl\_handler -> 이후에 EXPORT\_SYMBOL(sysctl\_handler); 해주면 sysctl 선언 끝!

1. adb shell 에서 sysctl



kernel빌드할 때 ./build\_redbull.sh SKIP\_MRPROPER=1

<zram\_to\_nbd\_handler 구현>

1. pid 를 통해 task\_struct를 가져와 target\_proc에 저장. (page table walk를 위해) (task: 자원 관리의 단위, 프로그램 실행의 단위인데 linux에서는 process와 thread의 구분이 없는듯, 하지만 process로 볼 것. process의 속성이 저 task\_strcut 구조체 안에 들어있음)
2. target\_proc에서 page table walk 하면서 pte가 존재하는 곳 찾음
3. pte가 존재한다면 swapout된 pte인지 (swap entry인지) 확인하면서 swap된 pte 찾음
   1. is\_swap\_pte(\*pte)를 통해서 swap된 entry인지 확인
4. swap된 entry 중에서도 ZRAM에 있는 entry를 찾음
   1. swap entry의 type이 0인지 확인f
   2. swp\_type(target\_entry) == ZRAM\_TYPE
5. ZRAM에 있는 entry를 읽어옴? 이건 읽어와야 mapcount를 알 수 있나
   1. mapcount는 page를 어디에 연결하는게 아닌 이상 바뀌지 않음 (-1이더라) 그래서 swp\_swapcount를 통해서 이게 1인 애들만 읽어올 것임.
      1. swp\_swapcount(entry)를 했는데 모든 swp\_swapcount가 1이었음 그래도 다른 어플도 켜고 하면 모르니 1인 애들만 읽어오도록 함
   2. ZRAM에 있는 거 읽어오려면 아마 swap\_readpage()하는데 두번째 인자 싱크는 true로
      1. swap\_readpage(struct \*page, sync)인데 이 때 struct\* page는 어떻게 알 수 있음? 이걸 entry를 기준으로 struct page를 읽어와야 할 것 같은데
   3. entry가 아니라 page를 읽어오는 것.
      1. pte / swp\_entry: main memory안에서 mapping을 위해 존재 (64bit)
      2. page: 4kB짜리 우리가 생각하는 메모리 관리 단위
      3. struct page 구조체: 아마 3~4B짜리인데 main memory의 한 page당 할당되는 구조체. 해당 page의 정보를 담고 있음. 처음 시스템 시작할 때 main memory에 대해서 딱 할당되고 그 안에 멤버 정보들만 쭉 바뀜
   4. ZRAM에 있는 page를 읽어와야함. 그러면 내 함수내에서 temp buffer같은게 필요. zRAM에서 읽어온 page를 잠시 저장해둘 공간.
      1. 이건 정적으로 할당하거나 동적으로 할당하면됨
         1. 정적으로 할거면 char tmp\_page[4096];
         2. 동적으로할거면 kmalloc같은거 쓰면 될듯. => kmalloc은 page size보다 작은 size의 memory를 할당할 때 사용되는 함수. 우리는 page를 할당할거기때문에 그거보다 더 큰 사이즈를 할당할 때 쓰는 alloc\_pages로 할당.
         3. 동적으로 하는것으로 결정. alloc\_pages(gfp\_mask, order = 0)을 사용할건데 gfp\_mask 어떻게 할지 찾아보기
            1. 해당 page는 할당받은 뒤 따로 free해주기 전까지 LRU에 의해서 evict 되면 안됨.
            2. gfp\_mask는 kernel memory를 할당받을거니까 GFP\_KERNEL으로
            3. unevictable은 page를 할당 받은 뒤에 SetPageUnevictable(page)으로 해결. PageUnevictable(page)로 확인 완료.
            4. 바뀌면 안되니까 HIGHMEM이 아닌 LOWMEM으로? HIGHMEM 비트 안넣기
         4. free\_pages(addr, order)로 dealloc. 이때 그냥 struct page\*는 못 넣으므로 page\_addresss(page)로 page의 가상주소를 넣어줌.
      2. 그 공간에 계속 ZRAM에서 읽어와서 해당 내용 NBD로 보내고 이런식으로 하면 될듯
      3. struct \*page는 메모리 공간내의 한 page당 그 물리 페이지를 관리하기 위한 하나의 struct임. 1:1 대응. 우리가 alloc\_pages로 1page 할당하면 그 page를 관리하기 위한 sturct \*page가 return되고 그걸 요리조리 함수의 parameter로 쓰면될듯.
   5. 원래 swap 영역에서 page를 읽는 함수 swap\_readpage를 참고
      1. 이 함수는 항상 \_\_read\_swap\_cache\_async 뒤에 나옴. 그럼으로써 함수 parameter로 들어가는 page에 뭔가 설정되는듯?
      2. swap\_readpage에서 block device일 때 하는 걸로 1차 코드 (ZRAM은 block device이니까?)
         1. bdev\_read\_page의 2번째 인자 swap\_page\_sector는 page의 정보가 없기 때문에 바로 못함
         2. swap\_page\_sector(page)는 \_\_page\_file\_index(page) << (PAGE\_SHIFT - 9) 를 반환.
            1. \_\_page\_file\_index(page)는 page의 private으로 swap\_entry를 구성한뒤에 그 swap\_entry의 offset을 반환
            2. 우리는 이미 swap\_entry를 알고있으니 그 offset을 여기에 넣으면 될듯
         3. 했는데 adb shell에서 process 스왑시키고 sysctl 활성화 시키니 디바이스 꺼짐 (그냥 swap\_readpage를 했을때와 유사)
            1. 디버깅하다보니 그냥 swap\_readpage나 swap\_page\_sector등 page를 parameter로 사용하는 내부함수들이 있는데 현재 page는 할당만 되어있는 NULL상태라 이걸 참조하여 값을 반환하려고 하니 null 참조가 뜨면서 system down.
            2. page가 아닌 entry에서 찾도록 함. entry를 인자로 갖게하는 새로운 함수 만듦.
   6. swapout 해야함. 이 때 zram이 아닌 nbd에 해야함.
      1. nbd에 해당하는 entry는 어떻게 가져올지?
         1. 새로운 함수 ksg\_get\_swap\_page를 만들어 count두고 count == 1일때 si에서 가져오도록?
         2. 함수 짜는 도중 get\_swap\_page\_of\_type이라는걸 찾음.
            1. 이게 type을 int로 넣으면 해당 type에 맞는 si를 swap\_type\_to\_swap\_info로 가져온 뒤에 그 영역에서 swap entry를 가져오는 함수인듯... 개고생만했다ㅠㅠ
            2. swap\_info\_get(entry)하면 entry의 swapinfo 가져올 수 있음! 이걸로 type한 번 알아보자 => 이게 결국 swap\_type\_to\_swap\_info라서 그냥 안해도됨
            3. entry와 nbd\_entry의 sis를 가져온다음에 type비교해서 0 1 이면 성공적으로 가져온것임!
            4. printk 해서 sis의 type 해본 결과 성공적으로 nbd의 entry를 가져옴!
         3. 실제 swap 영역에 writeback하는 함수인 \_\_swap\_writepage를 참고하여 ksg\_swap\_writepage(page, wbc, end\_swap\_bio\_write, nbd\_entry)를 구현
            1. wbc: writeback\_control하는 struct, \_\_swap\_writepage를 호출하는 swap\_writepage를 호출하는 함수에서 살펴봤을 때 sync\_mode에 대해서만 채운채로 선언한 뒤, 그 포인터를 넣는걸 보아 함수 내에서 wbc 채우는듯?

해당 함수에서는 sync\_mode를 WB\_SYNC\_NONE으로 했으나 우리는 모든 wirteback을 sync하게 해야하기에 WB\_SYNC\_ALL으로 설정

* + - * 1. wirteback이 끝나면 end\_swap\_bio\_write 라는 함수를 호출하여 writeback이 끝났음을 알리고 내부적으로 flag 변경해줌.

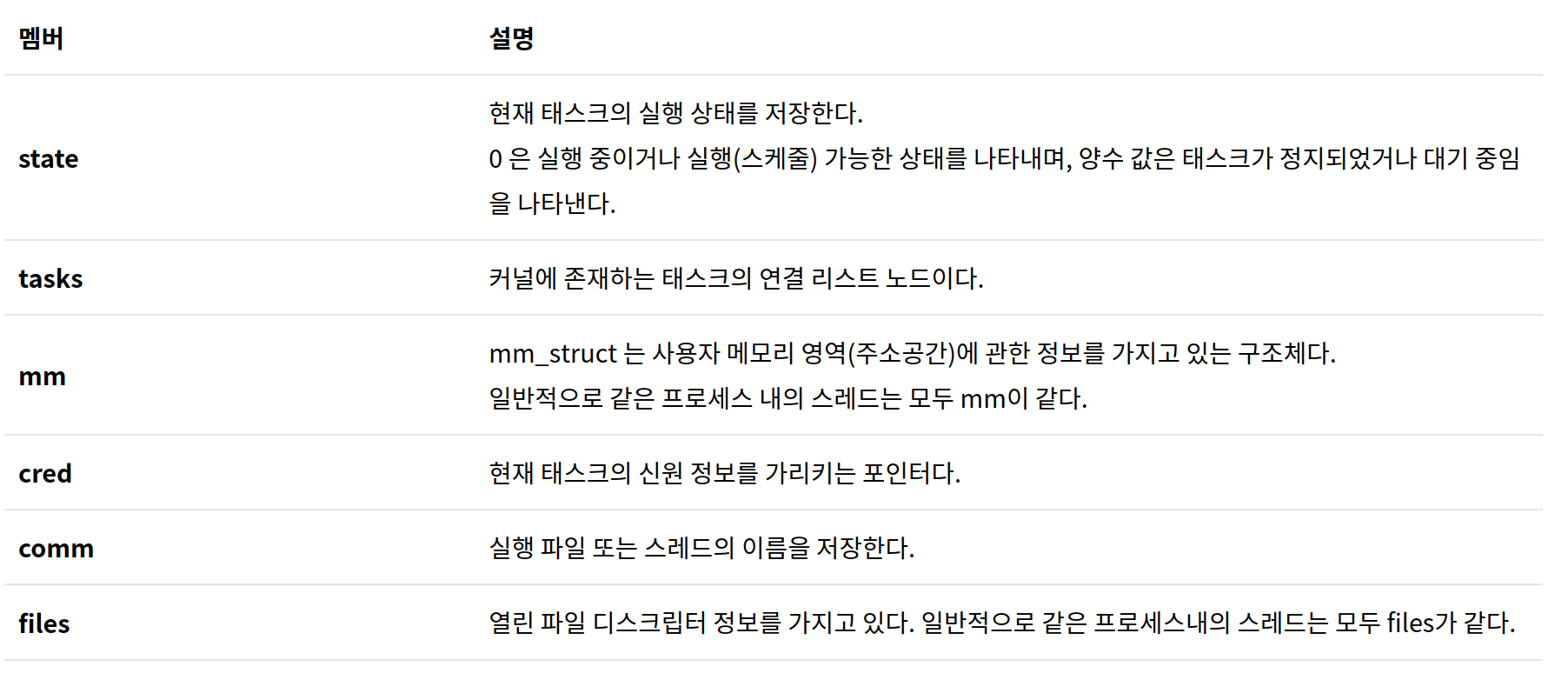
하지만 nbd는 bdev라서 상관없을듯.

실제로 해보니 nbd는 bdev가 아니라 bio로 처리

아마 /dev/block안에 들어있지않고 그냥 /dev/nbd0로 mknod해서 그런듯? 근데 /dev안에 nbd0가 있던데.. 이건 뭘까ㅠ

* + 1. nbd에 넣은 뒤 zram의 entry free 시켜야함 + pagetable에서 바라보고 있는 entry를 바꿔줘야함 (그냥 entry -> nbd\_entry로)
       1. entry를 바꿔줘야 나중에 pagefault가 났을 때, zram으로 가는것이 아닌 우리가 page를 옮긴 nbd로 갈 것임.
          1. entry를 바꾸는건 try\_to\_unmap과 try\_to\_unmap\_one을 참고
          2. 여기서 swp\_entry를 pte로 바꾸고 그걸 set\_pte\_at 함수로 pgtable에 넣는듯?
          3. 근데 이러면 swap\_entry가 아닌 pte로 들어가게 되는것 아닌지..? 정확히 이 함수에서 어떤 일을 하는건지 모르겠네ㅠ
       2. 그러면 zram의 entry는 쓸모없기 때문에 write가 끝나면 또다른 swapout을 기다리는 것이 아닌 해당 swap\_entry free
          1. 근데 나중에 다시 swapout 시키게 되면 이때는 다시 zram으로 하겠지?

1. process 관련 구조체: task\_struct (include/linux/sched.h)



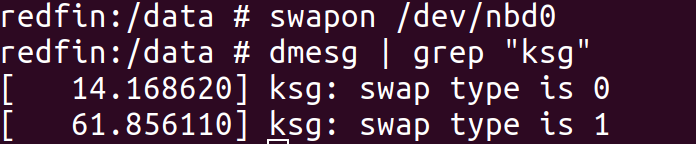
현재 zram에 있는지 없는지를 확인하는 방법이 없음. type으로 혹시 확인 가능할까?

alloc\_swap\_info() 를 swapon에서 호출하여 하나의 swap\_info\_struct에 하나의 type을 할당하긴함. 그러면 하나의 swap 영역에 하나의 sis를 할당할까? 즉 하나의 swap 영역에 하나의 type을 할당하는지? 하나의 영역에 하나의 type이라면 type으로도 가능할 듯.

mm/swapfile.c 에 linux/kernel.h 를 include하여 printk로 확인해봄. alloc\_swap\_info() 맨 마지막에 sis 할당하면서 sis return하기전에 그 sis의 type을 printk하도록

확인결과 처음 부팅할 때 zram0의 하나의 영역만 할당하며 그 type은 0 아마 nbd0 영역 mkswap후 swapon하면 그 type은 1이 될듯? 그러면 type0: zram, type1: nbd로 보고 type0인 애들 쭉 swapout 시키면 되지않을까

zram은 type0, nbd는 type1임.



4/14

1. vma가 정확히 어떤거인지? 이건 프로세스당 하나가 있는건가 아니면 그냥 시스템 전체에 하나?
   1. process 마다 메모리 매핑되어있는걸 관리하는 mm struct 이 하나 잇고 여기에 vma가 여러 개 있음. 이 vma들은 모두 하나의 mm을 가리키고있음
2. 여기서 불러오는 mm은?
   1. process마다 mm struct가 하나 있고, 이 mm struct가 vma에 다 연결되어있음.
3. try\_to\_unmap\_one을 cscope c로 호출하는 함수를 찾았는데 안나옴. try\_to\_unmap같은 곳에서 struct내에서 쓰이는 건 봤는데 따로 인자를 넣지는 않음. 이게 호출될 후크함수를 지정하는거라던데?
   1. 후크함수가 도대체 무엇인가.. 이 말 많이는 들었는데 검색해봐도 감이 안잡힘
   2. rmap walk가 실제 매핑을 해제하는 거인지? (문C블로그 + 진용이형 커널스터디)
   3. try\_to\_unmap -> rmap\_walk -> rmap\_walk\_anon을 가니 rwc->rmap\_one(page, vma, address, rwc->arg)가 있음 이게 try\_to\_unmap\_one 호출하는거라고 보면 될듯!
   4. 실제 행동은 try\_to\_unmap\_one을 하는거니까 이 함수를 보는게 맞는 것같음.
4. try\_to\_unmap에서 rwc 정의
   1. rmap\_one = try\_to\_unmap\_one (후크함수 지정)
   2. arg = (void \*)flags: 이건 나중에 채우는 듯
   3. done = page\_mapcount\_is\_zero
   4. anon\_lock = page\_lock\_anon\_vma\_read
   5. a,c,d는 후크함수 지정해두고 나중에 하위 함수에서 rwc->rmap\_one(page,vma,address,rwc->arg)처럼 파라미터 지정해주는 듯
5. try\_to\_unmap\_one의 파라미터: page, vma, address, arg. arg = ttu\_flags 이건 문C블로그 참고. 근데 arg를 void\* flags로 하는거 보아 별 상관은 없을듯?
   1. 이 파라미터들에서 vma = avc->vma 근데 우리는 vma를 이미 구해놈
   2. addresss = vma\_address(page, vma)
      1. 이 인자만 분석하면 set\_pte\_at 쓸 수 있을듯 + ptep가 왜 포인터변수인지도 봐야지
      2. vma\_address 함수 분석: unsigned long 반환, 해당 vma에서의 page virtual address인듯!
   3. avc 는 struct anon\_vma\_chain으로 anon\_vma\_interval\_tree\_foreach(avc, &anon\_vma->rb\_root, pgoff\_start, pgoff\_end)를 통해 구하는건가..?
   4. try\_to\_unmap\_one에서 set\_pte\_at의 파라미터로 들어가는거
6. try\_to\_unmap 부터해서 그 밑 모든 하위 함수들이 page를 기반으로 돌아감.. entry 기반으로 돌아가는게 없음.. 이정도면 page를 어떻게 연결해서 그걸 기반으로 돌리는게 낫지않을까..
   1. 근데 얘네는 이미 mapping 되어있는 page를 unmap하고, 그걸 swp\_entry로 채우는 거기때문에 이미 swp\_entry가 되어있는 entry를 다른 entry로 끼우는 작업만 하면 되는 우리 입장으로는 하위 함수들 몇개만 잘 쓰면 될것같기두?
   2. set\_pte\_at이 쓰이는 곳의 설명이 pte자리에 swp\_entry를 끼우는거 -> 진짜루 이거 쓰면 될거같은데?
      1. set\_pte\_at(mm, addr, \*ptep, pte): mm의 addr에 있는 옛날 pte ptep를 pte로 끼우는? 여기서 addr를 뭘로 줘야할지 모르겠네
      2. 여기서의 addr는 pvmw.address 또는 address로 들어오는데 둘은 같은 값임.
      3. 여기의 address는 rmap\_walk\_anon에서 vma\_address(page,vma) 값이 들어옴
      4. vma\_address(page,vma) 에서는 결국 \_\_vma\_address(page,vma)와 vma->vm\_start중 큰 값을 반환함.
         1. \_\_vma\_address(page,vma)에서는 vma->vm\_start + ((pgoff - vma->vm\_pgoff) << PAGE\_SHIFT 반환 : vma에서 pgoff만큼 전진한 값
         2. 결국엔 vma\_address란 vm\_start (page offset이 0) 또는 pgoffset만큼 전진한 vma의 어떤 가상주소 값 == 결국 이게 VPAGE
         3. set\_pte\_at 의 addr parameter에는 VPAGE 를 넣으면 될것같음!

[DEBUGING]

readpage -> writepage -> set\_pte\_at 을 했음에도 어플리케이션 팅김

[ 915.007477] adm\_callback: cmd = 0x1035d returned error = 0x3

[ 915.007524] adm\_set\_pp\_params: DSP returned error[ADSP\_EUNSUPPORTED]

[ 915.007535] adm\_pack\_and\_set\_one\_pp\_param: Failed to set parameter data, error -95

[ 915.007542] adm\_swap\_speaker\_channels: Failed to set swap speaker channels on port[0x9040] failed -95

[ 915.007550] msm\_routing\_stereo\_channel\_reverse\_control\_put:Swap\_channel failed, err=-95

이런 패턴의 에러가 마지막에 나옴. 근데 이건 일반적으로, 그러니까 echo anon을 통해 swap영역으로 강제로 보낸뒤에 내 핸들러를 작동시키지 않고 일반적인 swapin을 했을 때에도 보여지는 현상..

zap\_pte\_range: 함수 꺼질 때 마지막으로 들어가는 함수. 여기서 dump\_stack() 해서 콜트레이스 보기. do\_swap\_page에서 뭔가 오류가 날 거같음..

[ 153.987461] Call trace:

[ 153.987464] dump\_backtrace+0x0/0x168

[ 153.987468] show\_stack+0x18/0x24

[ 153.987470] dump\_stack+0xbc/0xf8

[ 153.987474] zap\_pte\_range+0x6a0/0x6e4

[ 153.987477] unmap\_page\_range+0xd4/0x1e8

[ 153.987481] unmap\_vmas+0xa0/0xbc

[ 153.987485] unmap\_region+0xd0/0x140

[ 153.987488] do\_munmap+0x2a4/0x434

[ 153.987491] \_\_arm64\_sys\_munmap+0x74/0xb4

[ 153.987494] el0\_svc\_common+0x9c/0x168

[ 153.987497] el0\_svc\_handler+0x60/0x6c

[ 153.987500] el0\_svc+0x8/0xc

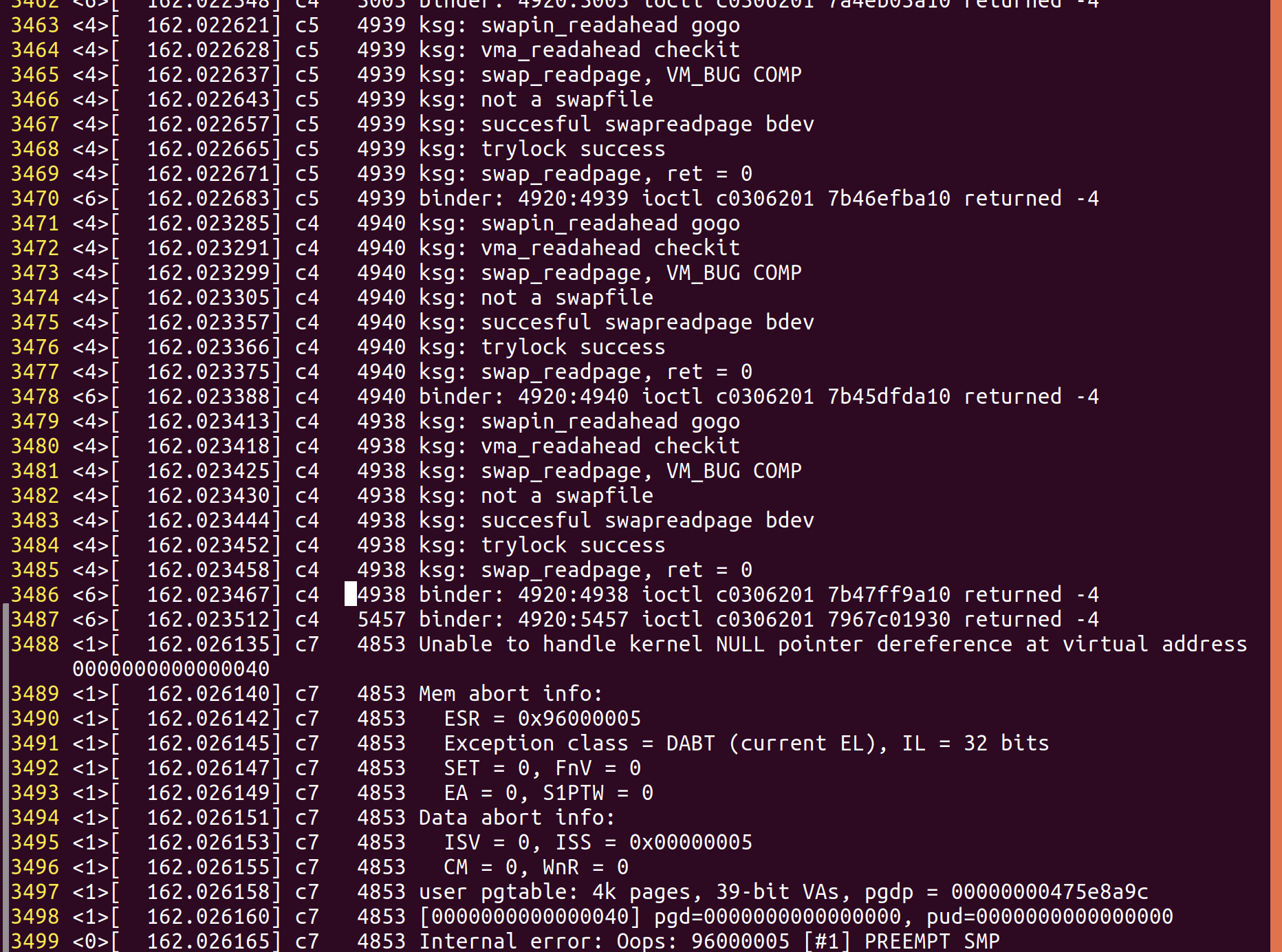
unmap page range에서 들어오는..? 뭐지 swapin하면서 unmap을 못하나? 그럼 쓰는 페이지가 제대로 연결이 안돼있어서 그런건가

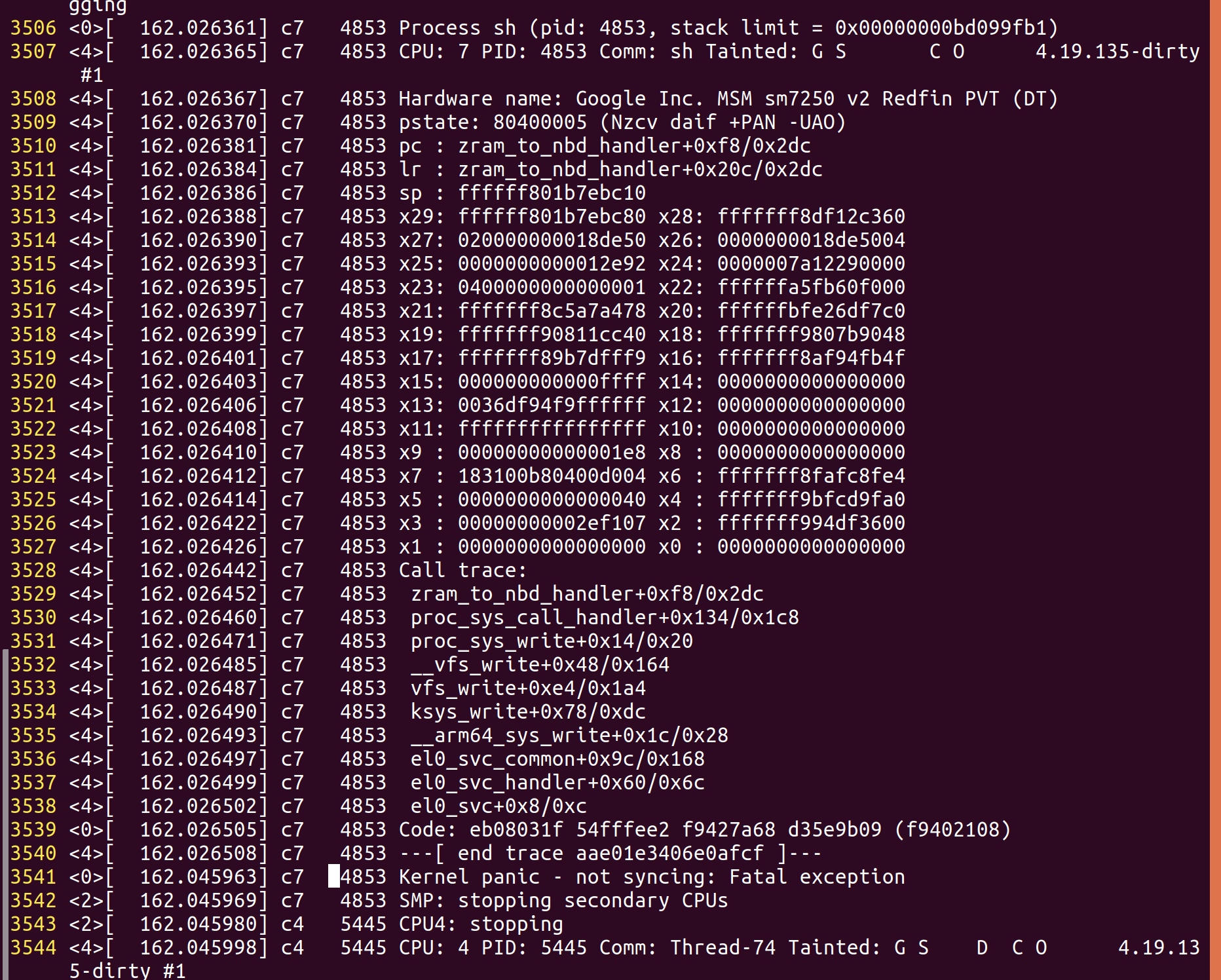
우리는 한 페이지씩 쓰는데 커널에서 vma\_readahead를 통해 한번에 여러개의 page씩 읽어오기 때문에 아무것도 없는 page를 읽어오려다 보니 어플리케이션이 팅기는게 아닐까 하는 생각

1. vma\_readahead에서 무조건 goto skip으로 가게하여 win == 1 즉 1페이지씩만 읽게 하였음.

2. 이렇게 하니 내 sysctl handler를 작동시키니까 모바일 기기가 다운됨. /sys/fs/pstore에 있는 다잉메시지 읽기

3. 근데 또 어쩔때는 프로세스만 죽고 어쩔때는 디바이스가 다운됨;;





흠.. 혹시 내가 page에 lock을 제대로 걸지 않고 read -> write -> set\_pte\_at 을 해서 read나 write를 하는 도중에 background swapin을 해서?? zram에서 읽어오나? 근데 이건 어플이나 디바이스가 다운될 이유는 아닌 것같은데.. 어쨌든 읽어올 수 있다는 것 아닌가?

만약 entry가 swap되어있다고 알고, 읽기 직전에 해당 엔트리를 swapin해온다면?

dmesg 했을 때 더 많이 보기

config\_log\_buf\_shift 를 늘리기. 이건 arch/arm64/configs/ranchu64\_defconfig에 있었음.

원래 값 14였는데 20으로 늘림.

이걸론 안되고 그냥 log\_buf\_shift 대신 25를 넣음

근데 dmesg 만 바꾸고 했는데 디바이스가 다운되지 않고 그냥 프로세스만 죽음

[ 236.315778] binder: undelivered transaction 171825, process died.

[ 236.315797] binder: undelivered transaction 171826, process died.

[ 236.315804] binder: undelivered transaction 171827, process died.

[ 236.315811] binder: undelivered transaction 171828, process died.

[ 236.326807] binder: 5101:5119 ioctl c0306201 dad1e068 returned -4

[ 236.326880] binder: 5101:5118 ioctl c0306201 dae1f068 returned -4

[ 236.327231] binder: 5101:5120 ioctl c0306201 dac1d068 returned -4

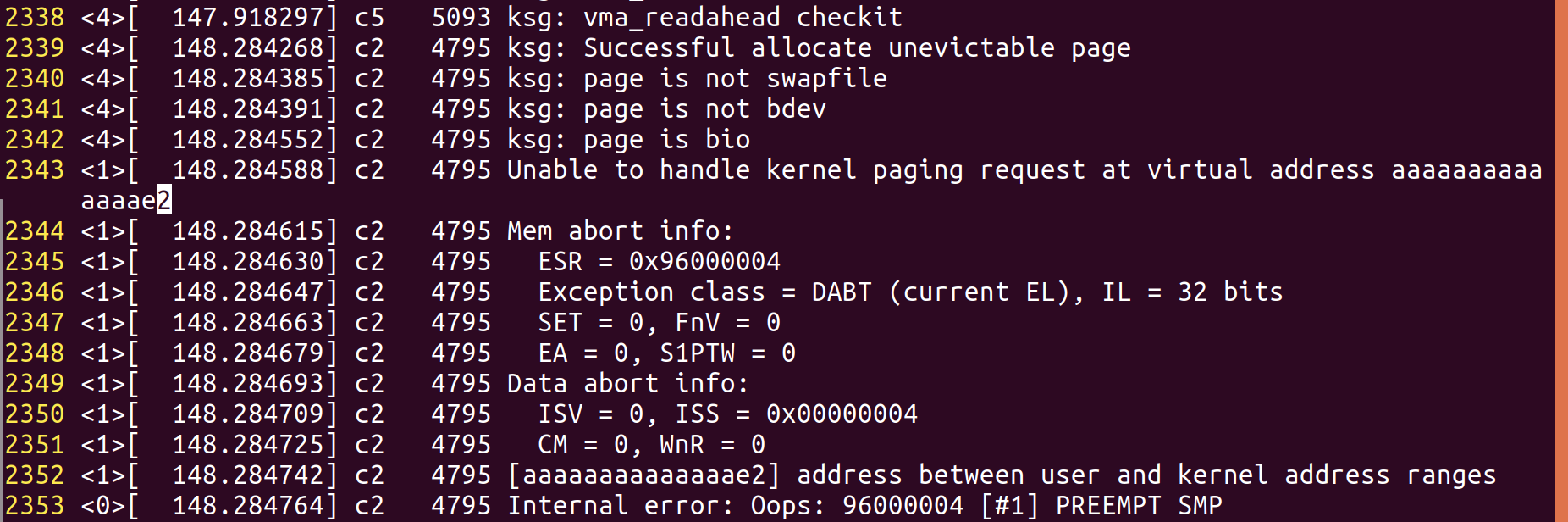
[ 236.347855] binder: 4325:4344 ioctl c0306201 763709da10 returned -4

[ 236.348753] binder: 4325:4343 ioctl c0306201 763919ba10 returned -4

[ 236.375148] binder: undelivered transaction 171891, process died.

그리고 다시 돌려보니까 디바이스가 죽음;

아 이번에 디바이스 죽은건 writepage에서 page에 대해 뭔가 printk하도록 뽑았는데 여기서 죽음. 왜냐하면 page는 아무것도 아니니까! 근데 이러면...어떻게 write한거지? 확실히 bio인데



이유를 모르겠네.. 그냥 ksg\_swap\_readpage랑 ksg\_swap\_writepage를 원래 있던 함수로 바꿀까..

근데 이건 page에서 정보 뽑아내는 걸 entry에서 뽑아내는걸루 바꾸는 정도였는데ㅜ do\_swap\_page 에서 보다가 page만 할당하고 지금처럼 하는게 있음!! entry를 기반으로 sis를 얻은 다음에 page를 할당받고, page를 할당 받았으면 \_\_SetPageLocked, \_\_SetPageSwapBacked, set\_page\_private!! 이걸로 entry정보를 page의 private에 넣는듯 이렇게 한 번 해볼까

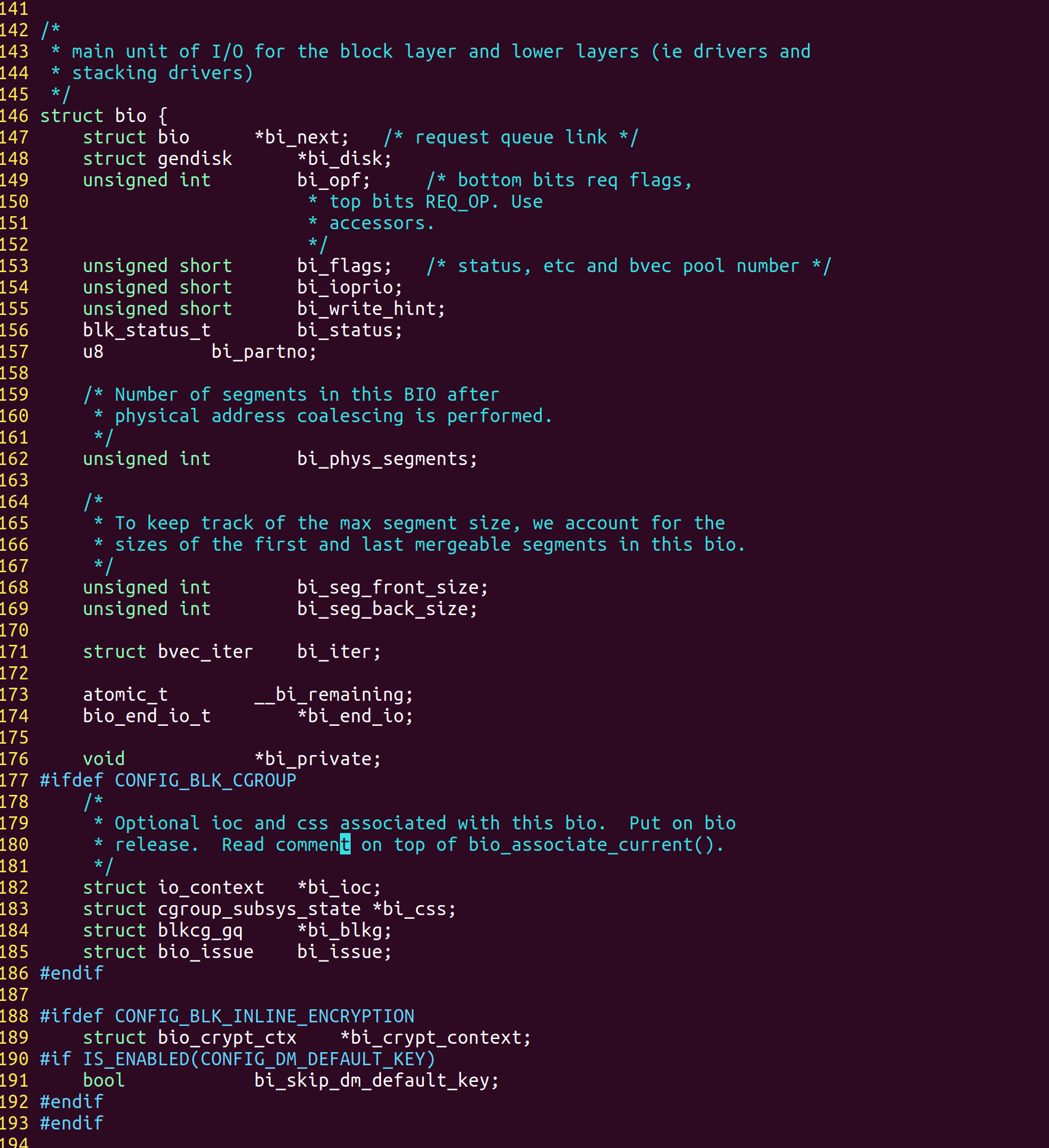
writepage할 때 아마 bio일텐데 bdev에만 바꾸고 bio는 안바꿈 -> 이러면 read한 page에 대해 그냥 하는게 아닌가? 어떻게 nbd에 썼지..? 일단 지금 문제는 writepage할 때 bio를 통해 하는데 제대로 안쓰는게 문제인듯!!! set\_page\_private을 통해 page의 private을 nbd entry로 바꾸고 해볼까?

writepage시 bio를 get\_swap\_bio 를 통해 가지고 오는데 이때 뭔가 제대로 가지고오는지? NULL로 안빠지니까 제대로 갖고오긴할텐데.. 뭐지

get\_swap\_bio내부를 보니 bio를 할당한 뒤에, map\_swap\_page를 통해 page를 bdev에 연결함. 그리고 bdev 내용으로 bio를 채움.

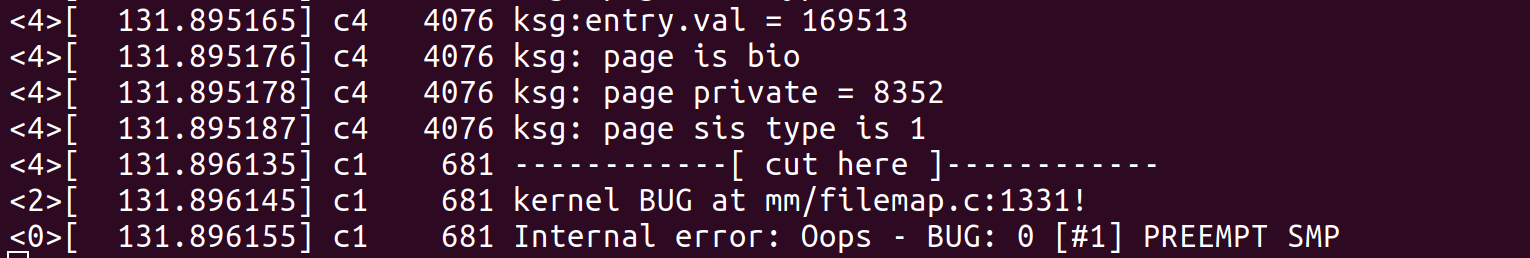
map\_swap\_page 내부를 보니 page의 private으로 entry.val을 채워 map\_swap\_entry를(entry,bdev)를 반환.

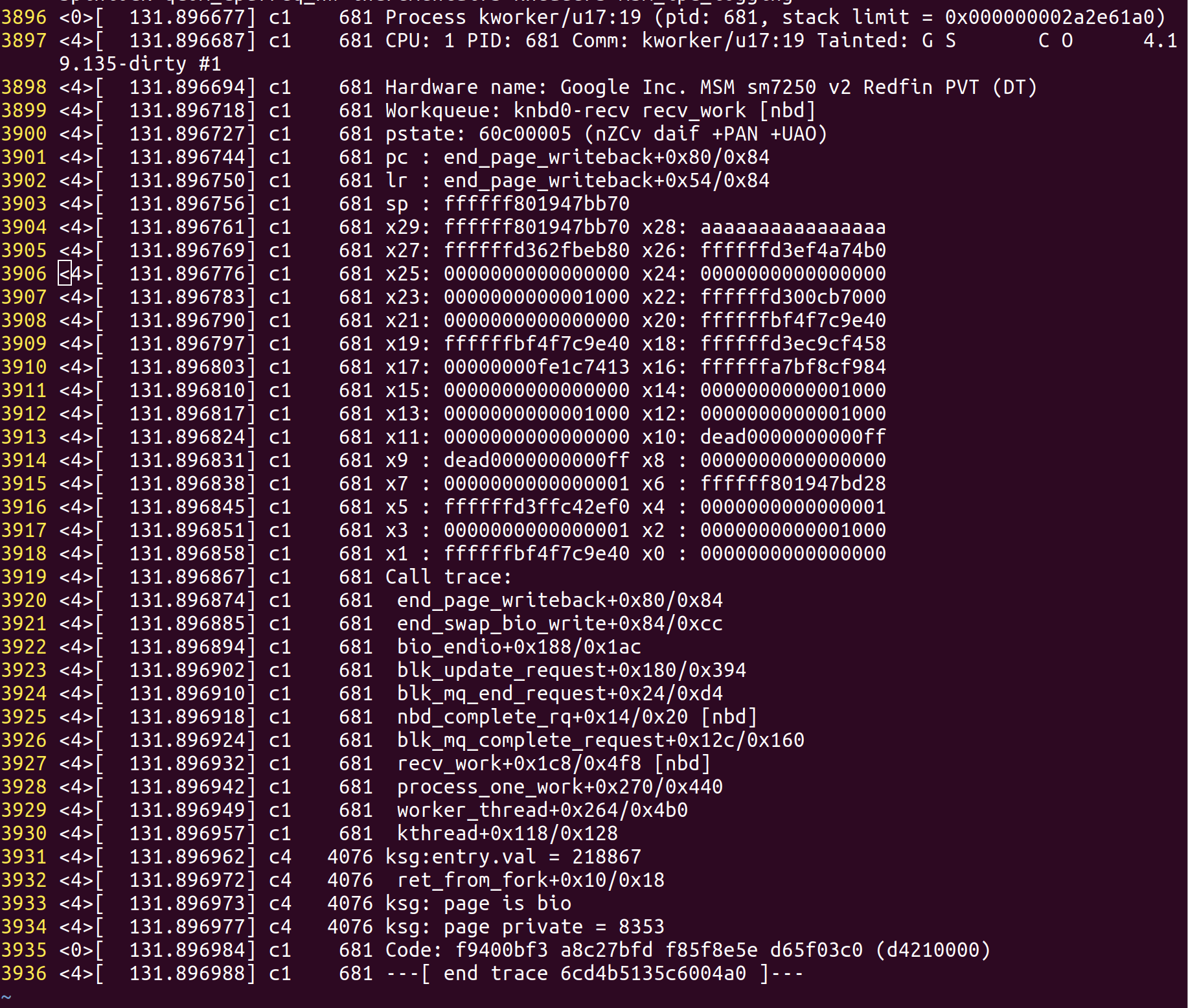
map\_swap\_entry를 보니 entry를 통해 sis를 받아와 그 bdev를 bdev에 넣음. 그리고 실제로 어디에 쓸지 pageoffset과 swap entry를 통해 가져오는듯. 이러면 page의 private을 채운뒤에 하면 되는거 아닐까..



/include/linux/blk\_types.h 에 bio struct

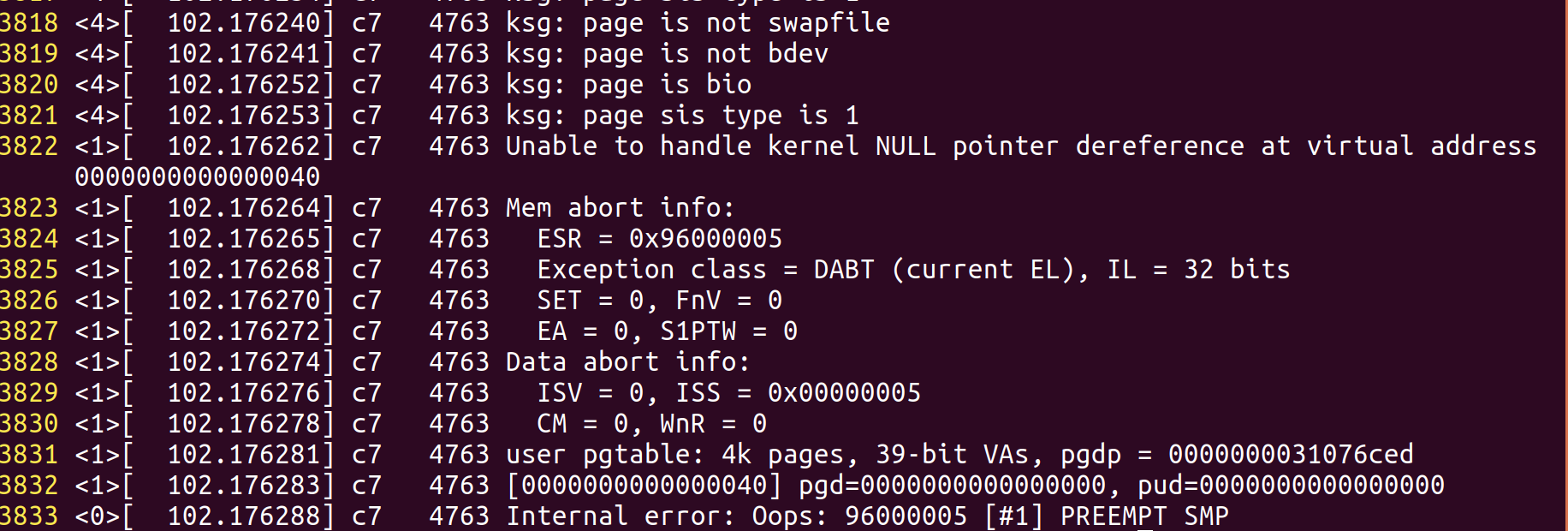
set\_page\_private을 통해서 swap\_readpage전에는 entry.val을 넣고, \_\_swap\_writepage전에는 nbd\_entry.val을 넣어서 햇는데 디바이스 다운됨.





그래서 그냥 page에 setting하고, ksg R/W 함수를 쓰고, write도 bdev가 되도록 해봄. -> 디바이스 다운

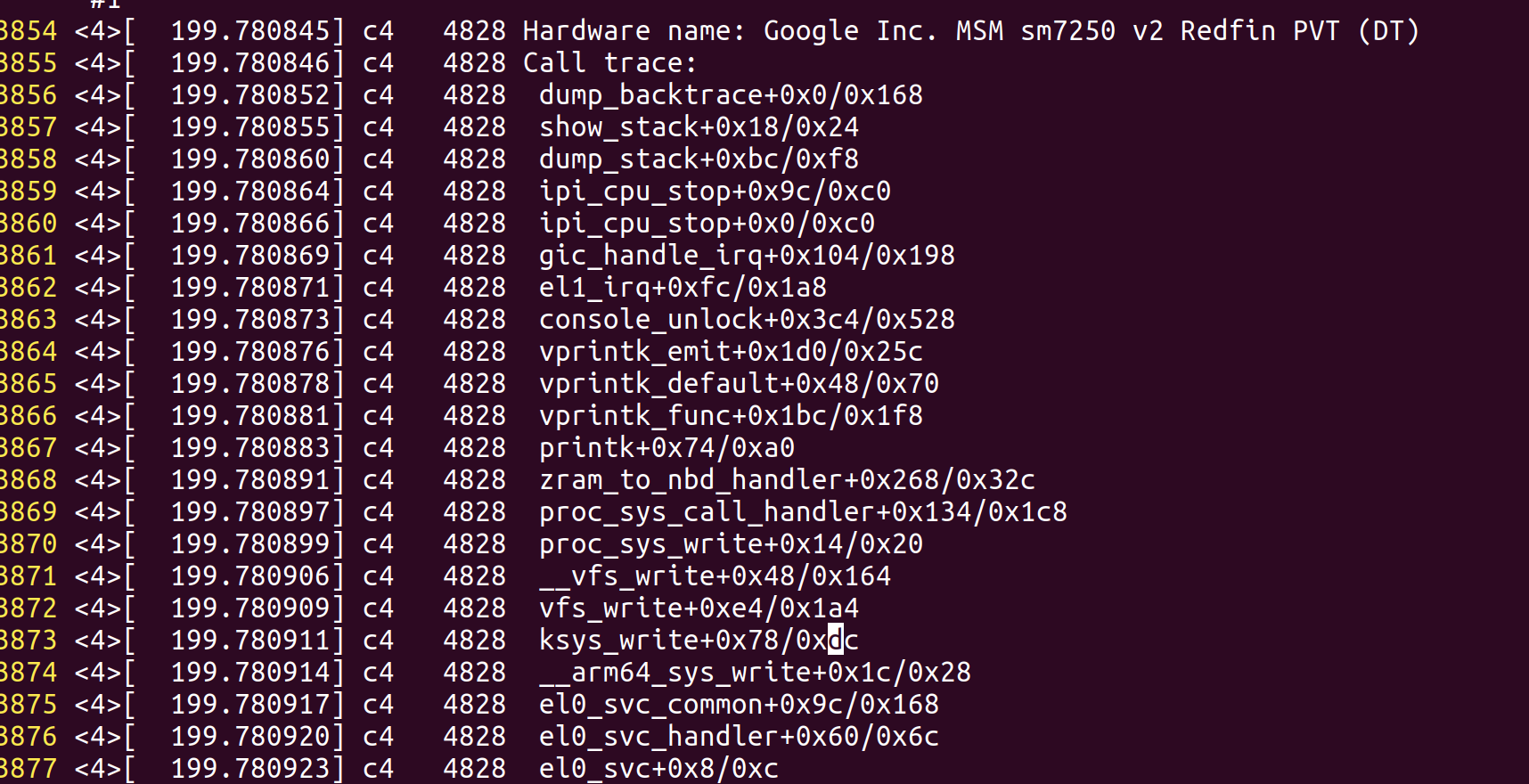
다잉메시지를 보니 bdev가 안되어있음 nbd는 어디 porting하더라도 bio로 가는듯?..



다시 nbd porting을 /dev/nbd0를 만들고 porting. page private을 entry -> nbd\_entry로 두 번 set. 근데 이렇게 하니 pid 672 ( Comm: kworker 에서 ) kernel BUG at mm/filemap.c:1331! 이 뜸 근데 이게 두 번 뜸.. 이거뭐지? filemap.c:1331로 가니 !test\_clear\_page\_writeback(page) 면 BUG(); 하도록.. 모르겠다..

근데 4828 sh, 4884 nbd-client, 4925 rovio.baba였는데 왜 4925가 아닌 4828에서 오류가나지? 아닌가 664에서 난건가? 664는 뭐지 process kworker인데

CPU: 4 PID: 4828 Comm: sh => 일단 sh의 pid에서 Successful allocate unevictable page가 출력되는거봐서 sh가 zram\_to\_nbd\_handler를 호출하는 process인듯 (sysctl 변수 핸들러를 sh process가 다루는?)



printk는 왜.. printk 다 지우니까 sh에서는 안남 근데 kworker에서 계속남남

문제점은..

1. 잘못 읽고있다

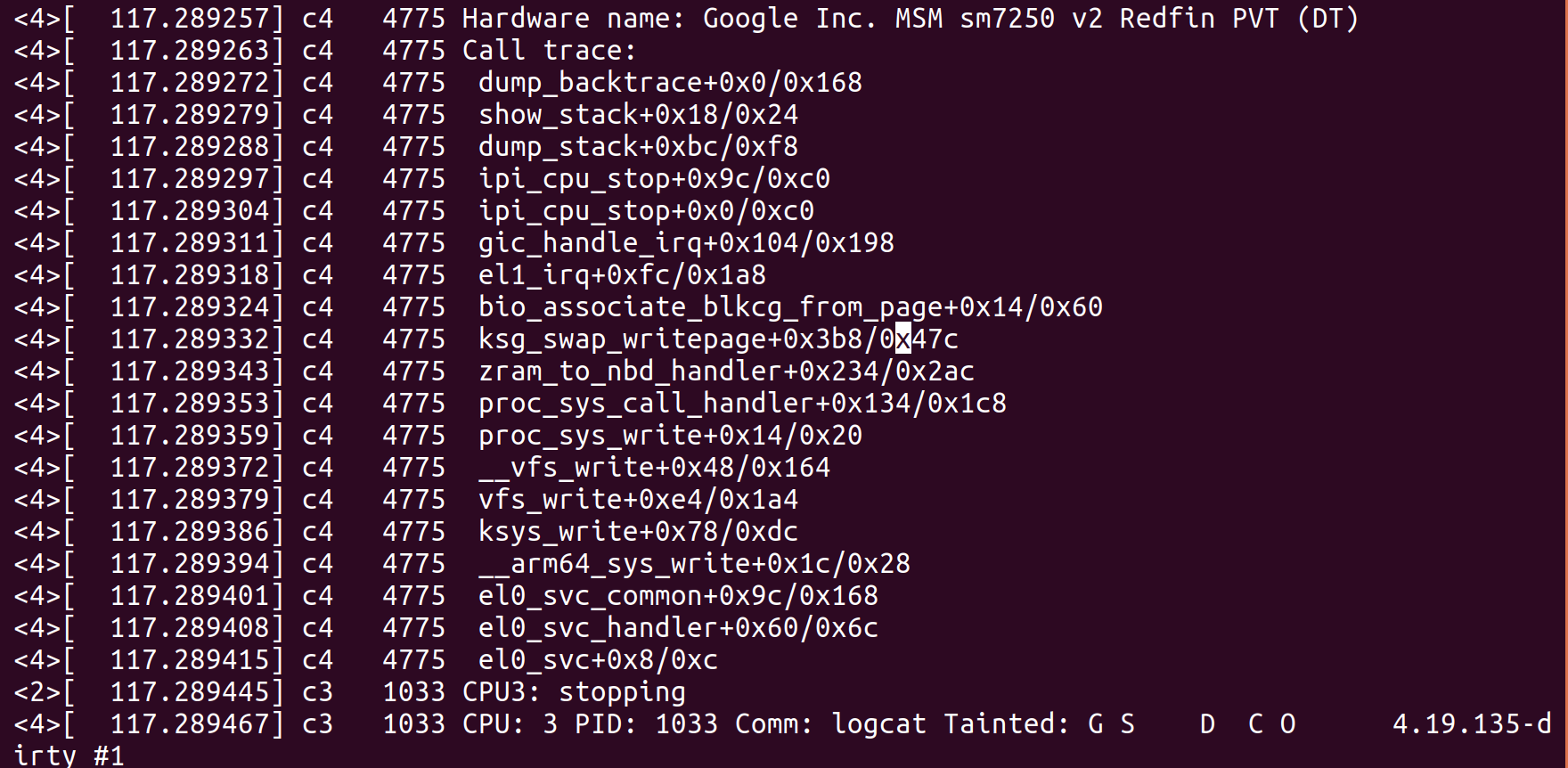
2. 잘못 쓰고있다

3. pgtable에서 entry를 잘못바꾸었다 (set\_pte\_at)

일단 디바이스 다운되는건 3을 주석처리하고 해도됨. 즉 set\_page\_private으로 page의 private을 nbd\_entry의 value값으로 채운 뒤에 swap\_writepage를 했을 때 나타남! 쓰는데 문제가 있긴있는듯? 아 이거 bdev로 쓸 수 있다면 참 좋을텐데..

일단 set\_page\_private을 read전에는 안쓰고 write할때만 써보자! nbd\_entry의 value값으로 채워서!

read 후 set\_page\_private을 통해 private 값 바꾸는게 문제인건가?? 이렇게만 해도 디바이스 다운됨.



bio\_associate\_blkcg\_frm\_page 라는 함수? 에서 el1\_irq가 호출됨 -> 이건 exception interrupt하는 거인듯

root 640 2 0 03:21 ? 00:00:00 [kworker/u17:2-pil\_workqueue]

root 642 2 0 03:21 ? 00:00:00 [kworker/u17:3-pil\_workqueue]

root 643 2 0 03:21 ? 00:00:00 [kworker/u17:4-dwc\_wq]

root 649 2 0 03:21 ? 00:00:00 [kworker/u17:5-pil\_workqueue]

root 650 2 0 03:21 ? 00:00:00 [kworker/u17:6-pil\_workqueue]

root 651 2 0 03:21 ? 00:00:00 [kworker/u17:7-pil\_workqueue]

root 652 2 0 03:21 ? 00:00:00 [kworker/u17:8-pil\_workqueue]

root 653 2 0 03:21 ? 00:00:00 [kworker/u17:9-pil\_workqueue]

root 654 2 0 03:21 ? 00:00:00 [kworker/u17:10-pil\_workqueue]

root 656 2 0 03:21 ? 00:00:00 [ipa\_interrupt\_w]

root 657 2 0 03:21 ? 00:00:00 [kworker/u17:11-pil\_workqueue]

root 658 2 0 03:21 ? 00:00:00 [ipawq36]

root 659 2 0 03:21 ? 00:00:00 [kworker/u17:12-pil\_workqueue]

root 660 2 0 03:21 ? 00:00:00 [kworker/u17:13-pil\_workqueue]

root 661 2 0 03:21 ? 00:00:00 [kworker/u17:14-pil\_workqueue]

root 662 2 0 03:21 ? 00:00:00 [kworker/u17:15-pil\_workqueue]

root 663 2 0 03:21 ? 00:00:00 [kworker/u17:16-dwc\_wq]

root 664 2 0 03:21 ? 00:00:00 [kworker/u17:17-pil\_workqueue]

root 665 2 0 03:21 ? 00:00:00 [iparepwq36]

root 667 2 0 03:21 ? 00:00:00 [kworker/u17:18-pil\_workqueue]

root 668 2 0 03:21 ? 00:00:00 [ipataskletwq36]

root 669 2 0 03:21 ? 00:00:00 [kworker/u17:19-pil\_workqueue]

root 670 2 0 03:21 ? 00:00:00 [kworker/u17:20-dwc\_wq]

root 671 2 0 03:21 ? 00:00:00 [ipawq33]

root 672 2 0 03:21 ? 00:00:00 [iparepwq33]

root 673 2 0 03:21 ? 00:00:00 [ipataskletwq33]

root 674 2 0 03:21 ? 00:00:00 [kworker/u17:21-pil\_workqueue]

root 676 2 0 03:21 ? 00:00:00 [kworker/u17:22-pil\_workqueue]

root 677 2 0 03:21 ? 00:00:00 [ipawq32]

root 678 2 0 03:21 ? 00:00:00 [kworker/u17:23-pil\_workqueue]

root 679 2 0 03:21 ? 00:00:00 [kworker/u17:24+knbd0-recv]

root 680 2 0 03:21 ? 00:00:00 [kworker/u17:25-pil\_workqueue]

root 681 2 0 03:21 ? 00:00:00 [kworker/u17:26-pil\_workqueue]

root 682 2 0 03:21 ? 00:00:00 [iparepwq32]

root 683 2 0 03:21 ? 00:00:00 [kworker/u17:27-pil\_workqueue]

root 684 2 0 03:21 ? 00:00:00 [kworker/u17:28-pil\_workqueue]

root 685 2 0 03:21 ? 00:00:00 [kworker/u17:29-pil\_workqueue]

root 686 2 0 03:21 ? 00:00:00 [kworker/u17:30-pil\_workqueue]

root 687 2 0 03:21 ? 00:00:00 [ipataskletwq32]

root 688 2 0 03:21 ? 00:00:00 [kworker/u17:31-pil\_workqueue]

root 690 2 0 03:21 ? 00:00:00 [kworker/u17:32-pil\_workqueue]

root 691 2 0 03:21 ? 00:00:00 [clnt\_req]

root 692 2 0 03:21 ? 00:00:00 [kworker/u17:33-pil\_workqueue]

root 694 2 0 03:21 ? 00:00:00 [kworker/u17:34-pil\_workqueue]

root 696 2 0 03:21 ? 00:00:00 [kworker/u17:35-pil\_workqueue]

root 698 2 0 03:21 ? 00:00:00 [kworker/u17:36-pil\_workqueue]

root 699 2 0 03:21 ? 00:00:00 [kworker/u17:37]

지속적으로 kernel BUG가 뜨는곳은 [kworker/u17:11+knbd0-recv] 라는 process.. 뭔가 nbd관련 kworker인듯? 여기서 end\_page\_writeback을 하는데 !test\_~ 가 안돼서 BUG 나는듯 이게 writeback flag가 clear돼야하는데 안돼서 그런건가??

\_\_swap\_writepage 대신 더 윗단의 함수인 pageout을 쓰기로함

pageout 중에서도 if(clear~) 안의 부분만 사용. 이를 위해서는 mapping과 sc 인자가 필요없음. 새로운 ksg\_pageout을 만듦. 호출전에 dirty set해주고 mapping->a\_ops->writepage가 실제 swap\_writepage 함수임을 앎. 이거 대신 swap\_writepage로 사용

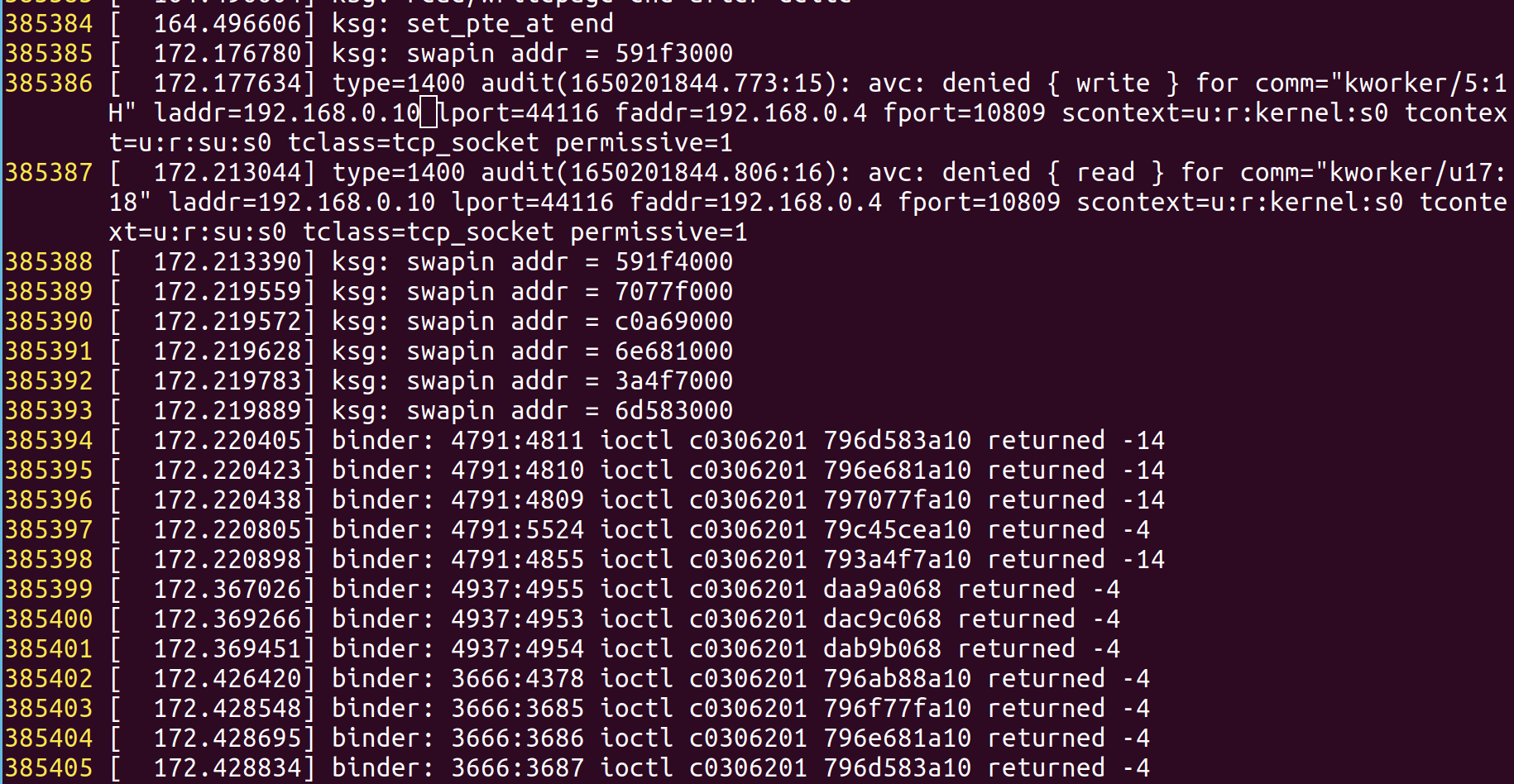
돌리니까 clear~에서 BUG. 보니까 page에 lock이 안걸려있으면 bug나는 것. swap\_readpage에서 lock을 한 번 풀어줘서 이렇게 되는듯 새로 lock을 걸고 돌려봄.

이전의 kworker BUG뜸. 여기가 mm/filemap.c 1331 인데 이 함수 아넹서 mapping을 써서 그런거일수도??일단 page의 mapping 추가해서 해보기로함. 그리고 이거 안되면 그냥 page를 cache에 잠시 넣었다가 뺐다가 하는식으로 해보자

일단 page의 mapping을 추가해서 하는건 안됨... 다잉메시지 확인 -> 똑같은거에서 죽음..

page를 add\_to\_swap\_cache / \_\_delete\_from\_swap\_cache를 통해서 swapcache에 넣었다 뺐다하기. 이때 entry는 nbd\_entry를 줌.

write는 되긴 되는듯 근데 예전처럼 다시 앱에서 읽으려고 하면 앱이 크러쉬남. type = 1400 aduit~ 에 대해서 더 알아볼것.



한 vpage (vaddr)에 여러 번 접근하는 현상 발생..? page table walk가 이상한가?

9ed23000 에서

62894000 로

가는게 포문상 말이안되지않나 vpage는 vma->start부터 vma->end까지 PAGE\_SIZE만큼 커지면서 가는데?/ vma 끼리는 낮은 addr -> 높은 addr 순으로 연속적으로 있는게 아닌가?

fdc55000 -> d0d2000 -> ddee000 -> 10000000 -> 2005a000 그 뒤부턴 다시 연속적으로 5~까진 그럴듯한 숫자 60000000 -> 70000000 -> 80000000 -> 90000000으로 vma->vm\_start 가는게 이상하긴함

f0000000까지갔다가 다시 vma->st = 0으로 감...이게 뭐누.. vma = vma->vm\_next로 가는건데 이건 linked list로 돼있어서 vma->vm\_start가 꼭 연속적이지는 않다는건가? (작은 값 -> 큰 값으로)

0 -> 10000000 으로 가서 겹쳐지게됨

type=1400 avc:denied가 문제일까 해서 일단 zram->zram으로 보내고 다시 읽어오게함

이래도 프로세스 다운됨.

binder: undelivered transaction 141695 ~97, process died.

ksg\_pageout 의 return 값 뽑아내보니 PAGE\_SUCCESS를 return 하지못함!!! 그래서 어디서 막히나 쭉 보려고 printk 넣어봄.

clear\_page\_dirty\_for\_io의 if문 자체에 못들어감. add\_to\_swap에서 add\_to\_swap\_cache 호출후에 마지막에 set\_dirty\_page를 해주는데 나는 이걸 안해줘서 dirty big이 없어서 안된듯 dirty bit 세팅해주니 됨!!!!! 어플 크러쉬 안남 ㅠㅠㅠ 근데 zram->zram이라 이제 nbd로 다시 해봐야할듯.

nbd로 보내니 예전과 같은 문제 발생. kernel BUG at mm/filemap.c:1331....

결국 이걸 해결하기위해 pageout을쓰고, cache에 넣었다뺐다가도 한건데 이게 해결이안됨. 이러면 \_\_swap\_writepage를 그대로 쓰고 zram -> zram도 되긴 됐을듯?

결국 이 kernel BUG는 page의 writeback bit를 꺼야하는데 못꺼서 그게 return 0로 return되어 BUG()가 활성화되는것. 이 writeback bit가 잘 가다가 중간 한 페이지에서 return 0를 함. 이게 말이되나?

-> 혹시 멀티코어때문에 다른 코어에서 이 page를 건드려서 flag를 껐는데 그걸 어떻게 하나?

-> cpu pinning으로 cpu4만 켜고 했는데 같ㅇ느 문제 발생

flag만 lock을 잡는것이 아닌 실제로 page의 lock을 잡는 lock\_page -> unlock\_page를 통해 이 문제는 해결

근데 set\_pte\_at을 하니 app crash..

zramtozram은 되는데 왜 zramtonbd는 안될까? 일단 zramtozram은 무조건 다 읽어서 다 쓰고 난 다음에 다시 swapin이 일어나는것같음. 그전에는 swapin이 일어나지 않는것같음 아니네 일어나긴하네 근데 잘 되는듯 무조건 set\_pte\_at이 끝나고 그걸 swapin 해오는데? 이게 type이 바뀌는게 뭔가 문제가 잇는건가..

zram to zram에서는 handler 내에서 vpage 역행도 없음 아니다 이것도 있긴한데 그리 빈번하지않음

그리고 swapin 해오는거도 handler가 도는 와중에는 거의 없고 handler 끝난 뒤에 swapin 해오는게 많음

ZRAM에서 page read -> NBD에 write -> set pte at

의 과정사이에 page가 접근됐나? 그러니까 set pte at 을 하려고하는데 이미 기존 pte가 바뀌어있는? 근데 이게 문제가 되나.. 이게 문제가 되진 않을것같은데..

확인방법: pgtable walk 처음 시작할 때 pte 저장해두고 pte 바꿔주기전에 바뀌었는지 한번 더 보기 + 기존에는 cache에 없는 환경인데 nbd에 쓰기전에 cache에 그게 들어와있는지도 확인.

아니지 만약 접근 돼서 swapin을 해온다면 pte 바뀐지가 아니라 swap entry인지 보면 되는거 아닌가? 그러면 is\_swap\_pte인지랑 find\_get\_page해서 cache에 있는지 두 개!

결국 이런 경우는 없는것같음.. 내가 조건으 잘못줫나?

pageout한 후에 unlock page가 있음 그래서 새로 lock 걸어주기 -> unlock page에서 lock 안걸려잇는건 해결 근데 deletefrom cache에서는 !pglock !pgswapcache pgwriteback 걸려잇음 근데 이건 그냥 일반 page를 swapin할때도 이러는거같은데?

계속 앱이 아예 멈춰버리는 현상 발생 dmesg 를 제외한 ps -ef나 logcatg이 안먹힘

nbd 포팅후 zram swapoff시키고 그냥 swapout시켜봄. 근데 똑같이 앱이 아예 멈춰버리는 현상 발생! 이건 swapin하기도전에? (swapin dmesg가 뜨지않음) nbd device와 IO하는 것 자체가 문제가 있는?

결국 avc denied돼있는 문제? 이걸 해결을 해야하나 SELinux 정책이라는데.. 근데 getenforce해보면 permissive임 하지만 tclass=tcp\_socket permissive=1이 돼있음

근데 gmail은 잘 되는듯? 뭐지

근데 nbd포팅할때부터 type=1400 audit, avc:denied {read}랑 {write}가 있음 이건 해결할 문제아닐까 nbd device에 읽고 쓰는게 denied라는거같은데

11aosp/device/google/redfin-sepolicy/vendor/google

SElinux 는 setenforce 0 으로 해서 permissive로 해두면 logging은 뜨지만 정책 자체는 꺼져있어서 문제 X

=> 결국 뭐가 문제일까.. 일단 table 두번 도는게 또 발생

pte lock 과 page table lock을 걸어봄 -> 이것도 안돼..

drivers/android/binder.c 에 5235 pr\_info("%d:%d ioctl %x %lx returned %d\n")

errno-base.h에서 보면 -4는 interrupted system call을 받는 경우, -14는 bad address인 경우

:cs find e process died 로 찾으면 나머지 log 도 찾을 수 있음 근데 이거는 process가 꺼져서 undelivered transaction이 일어났다는것으로 process꺼지는 것ㅇㅔ 대한 log 아님

pageout 과 \_\_swap\_writepage대신 한 page만 쓰는 write\_one\_page api를 쓰니까 됨!!! 미쳤따;

write\_one\_page는 무조건 sync로 1page씩. 속도는 확실히 좀 느리긴 했음. 무조건 sync로 하니까 readahead켜도 되지않을까? 켜보자 => 잘됨!

근데 이러면 cache에 넣을필요 없는것 아닌가 해서 add\_to\_cache를 빼봤는데 address\_space mapping 이 있어야 write\_one\_page 사용가능 아니면 NULL pointer handling뜸

swap\_free(entry)를 통해 free도 완료.

adb reboot 안 되는게 swapoff /dev/nbd0하는데 오래걸려서 그런느듯 일단 swapoff부터 하고 adb reboot하니 잘됨!

background daemon화

실제 daemon을 만들기는 힘들어, 같은 swap 관련인 kswapd daemon 안에 넣기

walk\_process\_tree in include/linux/sched/signal.h kernel/fork.c 도

이건 아닌거같음 같은 파일안에서 for\_each\_process(task)

tasklist\_lock 걸어야하나? 근데 이게 걸리면 background에서 하는 의미가 없는것 아닌지?

이게 안되면 page 두개 할당받아서 하나는 read용으로 해서 zram cache, 하나는 write용으로 해서 nbd cache에 넣고 page끼리 memcpy