OS PA4

2020-15607 김연재

1. New data structures

fixed_page_allocated, kmem_fixed:
ZONE_FIXED에 존재하는 page를 관리하기 위해서 사용하였다. kmem_fixed는 linked list를 이용했는데 이를 통해서 ZONE_FIXED에 있는 page를 O(1)만에 free하고 allocate할 수 있게 하였다. 그리고 fixed_page_allocated는 char array를 이용해서 kfree를 할 때, 기존에 할당된 페이지인지 O(1)만에 확인할 수 있도록 하였다. 모든 ZONE_FIXED page에 대해 array로 만들어야 하는 단점이 있지만 전체 memory에서 이 부분은 수 kb정도로 작다고 판단을 했기 때문에 이런 자료구조를 사용하였다.

page_info, kmem_fifo, kmem_normal:
ZONE_NORMAL에 존재하는 page들을 관리하기 위해서 사용하였다. struct page_info array를 가지고 있는 kmem_fifo를 이용해서 할당되어 있는 page들을 fifo로 관리한다. array를 이용하기 때문에 kalloc이나 kfree가 되는 주소를 알면 O(1)만에 page에 대한 정보를 알 수 있다. 이를 이용해서 나중에 구현할 swapout을 빠르게 구현할 수 있다.

cp_length:
part3에서 compress를 했을 때의 결과를 저장하기 위한 자료구조이다. 이 역시 array로 구현했으며 압축된 zmem의 주소를 알고 있으면 decompress를 할 때 압축된 길이를 바로 알 수 있다.

zmem_page_allocated:
zfree를 할 때, 기존에 zalloc을 통해 할당되어있는지를 판단하기 위한 array 자료구조이다. 앞선 fixed_page_allocated와 똑같이 구현된다. 하지만 zmem은 Half Page로 할당될 수도 있기 때문에 넣어주는 값이 0, ZHALF, ZFULL을 이용한다.

inverted_pte, ipt:
swap out이 될 때, 해당 page가 어떤 process에서 사용되는 page인지를 array를 이용해서 빠르게 판단할 수 있도록 한다. 다만, 구현에서는 pid가 아닌 pagetable과 va를 이용해서 inverted_pte(inverted page table entry)를 구현했는데 이는 exec등의 부분에서 좀 더 자연스럽게 하기 위해 이런 구조를 사용하였다.

zmem:
free인 2kb와 4kb를 관리하는 linked list 자료구조이다. 각각을 나누어서 관리를 한다. 기본적으로는 4kb단위를 관리를 하고 2kb에 대한 요청이 올 때 4kb를 잘라서 준다. 그리고 2kb가 free되는 경우에는 buddy 주소가 free list에 있는지를 찾고 있으면 병합해준다. 이를 이용해 allocate을 할

때, O(1)만에 위치를 찾도록 하고 free를 할 때에는 4kb일 때는 O(1), 2kb일때는 상황에 따라서 O(1) ~ O(n)의 시간이 걸린다.

2. Algorithm design

swapin:

먼저, kalloc(ZONE_NORMAL)를 통해서 어디로 swap in을 할 것인지를 정한다. 메모리 주소가 정해지면 swapin을 하는 process의 pagetable의 page table entry를 수정해준다. 그 후, compress가 되어있던 page를 decompress 하여 해당 page로 옮겨주고 inverted page table을 수정해준다. swapin을 할 때, ZONE_NORMAL에 자리가 없더라도 kalloc내부에서 swapout을 해주기 때문에 ZONE_NORMAL로 swapin을 할 수 있다.

Swapout:

앞선 자료구조 중 하나인 page_fifo에 대한 함수 dequeue()를 이용해서 어떤 page를 swap out을 할 것인지를 정한다. 그리고 이에 대한 inverted page table을 수정한다. 그리고 swap out을 할 page를 compress를 해야하기 때문에 이에 필요한 memory들을 kalloc(ZONE_FIXED)를 통해서 얻는다.

Optimization

앞선 자료구조들을 이용하는 과정에서 시간 측면에서 효율적으로 짜려고 하였다. 예를 들면, zmem에서 buddy가 free를 확인하는 과정에서 free list를 도는 것이 아니라 zmem_page_allocated를 이용해서 빠르게 free인지를 판단하고 free인 경우에만 합병을 해주는 과정으로 구현을 하였다. 나머지 최적화들은 모두 1번에서 설명하였다.

3. Testing and validation(2번의 corner case도 같이 고려)

usertests- q를 하던 도중 uvmcopy에서 pte를 old page table에서 walk를 통해서 구한다. 그리고 kalloc을 통해서 new page table을 위한 page를 allocate을 한다. 하지만 이때 kalloc으로 인해 이 page가 evict되는 경우가 있었다. 그래서 이를 해결하기 위해서 buffer를 두고 old page table에 있던 page를 옮겨준 이후에 이 buffer를 새로 할당된 page로 옮겨주도록 하였다.
다른 corner case에 대해서도 상황이 있는데 이는 4번 part에서 설명한다.

4. bonus

test 중에 가장 critical하다고 생각되는 부분은 page table entry를 이용해서 해당 page에 대한 정보를 얻었는데 timer interrupt등에 의해서 다른 process가 실행되고 나중에 해당 process가 실행될때 잘못된 메모리에 접근을 할 수 있다는 것이다. 그래서 multi core(bonus part)부분은 물론이고 single core에서도 적절히 lock을 사용해주어 이러한 문제가 발생하지 않도록 구현하였다.

이때 사용한 lock은 multi core에서도 비슷하게 적용했다. critical section으로는 대표적으로 앞선

상황이 있고, inverted page table을 update하는 등의 과정이 있다. 이런 critical section이 발생하는 코드들을 분석하는 결과 두 분류로 나눌 수 있었다. kalloc, kfree, walk부분에서 발생한다. 이때, 3가지 함수에서는 해당 cpu가 lock을 획득하지 않았다면 lock을 획득하도록 하였다. critical section이 종료되는 지점에서 lock을 해제하도록 해주었다. 하지만 단순히 이렇게 구현했을때, interrupt가 발생하고 sched함수에서 lock을 검사하는 부분에서 문제가 생긴다. 그래서 lock을 2개 가지고 있고 내가 이용한 lock을 가지고 있는 경우에는 panic이 발생하지 않도록 구현하였다. 그러고 해당 process에 돌리고 있던 cpu에 대한 정보를 넣어주고 나중에 scheduler함수에서 돌리고 있던 cpu에서만 다시 schedule이 될 수 있도록 구현을 하였다. 이를 통해, swapin, swapout등이 발생하면서 page table entry가 바뀌거나, inverted page table이 바뀌는 부분인 critical section의 시작부터 마지막 부분까지 lock을 이용해서 문제가 발생하지 않도록 구현을 하였다.