Project 4

Report

2017314331

김도엽

신동군 교수님

운영체제

0. Page Swap Implementation 을 위한 전반적인 변경 및 추가 코드의 개략적인 설명 :

(1) Swap in 의 구현 :

kalloc.c 에서 kalloc() 함수, trap.c 에서 page_fault_handler() 함수에서 page swap in 상황에 대한 기능 구현을 진행하였습니다. kalloc() 함수에서는

(2) Swap out 의 구현 :

vm.c 에서의 freevm() 함수 외 필요한 User 함수, , trap.c 에서 page_fault_handler() 함수에서 page swap out 상황에 대한 기능 구현을 진행하였습니다.

(3) Page replacement algorithm: clock algorithm 의 구현:

별도의 함수를 작성하지 않고, 이상 설명한 freevm(), kalloc, page_fault_handler 함수에서 전역으로 선언되어있는 (struct page *page_lru_head)의 정보를 가져와 매 상황마다 판단하는 코드로써 구현하였습니다.

1. Swap-In 의 구현 :

(1) kalloc.c 의 kalloc() 함수 수정 :

수정된 kalloc() 함수는 다음과 같으며, 각 줄에 사용된 주석으로 코드에 대한 간략한 설명을 대체하겠습니다.

```
char* kalloc(void) {
  struct run *r;
  struct page *pg;
  pte_t *pte;

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

  r = kmem.freelist;

  if(V2P((char*)r) >= PHYSTOP) { //pages[PHYSTOP/PGSIZE]의 index 를 넘는
범위 error handling
    cprintf("kalloc: Out Of memory\n");

    if(kmem.use_lock)
        release(&kmem.lock);
    return 0;
  }

  if(!r) {
    // free page 가 없으면, 페이지를 교체
```

```
    if (num_lru_pages == 0) {
      // Out of memory error
      cprintf("kalloc : Out of memory\n");
      if(kmem.use_lock)
        release(&kmem.lock);
      return 0;
    }


    // LRU 목록의 맨 앞에서 페이지 제거
    if (!clock_hand) {
      clock_hand = page_lru_head;
    }

    // Implementation of clock algorithm
    while (1) {
      pde_t *pgdir = clock_hand->pgdir;
      pte = &((pte_t*)P2V(PTE_ADDR(pgdir[PDX(clock_hand-
>vaddr)])))[PTX(clock_hand->vaddr)];

      if (!(*pte & PTE_A)) {
        // Page 에 access 불가능할 경우 evict
        pg = clock_hand;
        break;
      }

      // Page 에 access 가능한 경우 : clear the accessed bit and move the
clock hand
      *pte &= ~PTE_A;
      clock_hand = clock_hand->next;
    }

    // Swap out
    int swap_offset = find_free_swap_space();
    if(swap_offset < 0) {
      release(&kmem.lock);
      panic("kalloc : no free swap space");
    }

    swapwrite(P2V(PTE_ADDR(*pte)), swap_offset);
    *pte = swap_offset | (*pte & 0xFFF);
    *pte &= ~PTE_P;  // Clear present bit

    //  LRU list 에서 evicted page 제거
    uint pa = PTE_ADDR(*pte);
    struct page *page = &pages[pa >> PTXSHIFT];
    page->vaddr = 0;
    if (pg->next == pg) {
```

```
      page_lru_head = clock_hand = (void*)0;
    } else {
      pg->next->prev = pg->prev;
      pg->prev->next = pg->next;
      if (pg == page_lru_head) {
        page_lru_head = pg->next;
      }
      if (pg == clock_hand) {
        clock_hand = pg->next;
      }
    }
    num_lru_pages--;

    // evicted page의 physical page 재사용
    r = (struct run*)pg->vaddr;
    memset(r, 0, PGSIZE);
  }

  if(r){
    kmem.freelist = r->next;
    uint pa = V2P((char*)r);
    int pageIndex = (V2P((char *) r)) / PGSIZE;
    // Assumes pages[] was initialized as in initialize_pages()
    pages[pageIndex].vaddr = (char *) r;
    struct page *page = &pages[pa >> PTXSHIFT];
    page->vaddr = (char*)r;
    num_free_pages--;
  }

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

  return (char*)r;
}
```

(1) 메모리(kmem.freelist)에 여유 페이지가 있는지 확인합니다. 그렇지 않은 경우 LRU Clock 알고리즘을 기반으로 교체할 페이지를 선택합니다.

(2) 선택한 페이지에 대한 페이지 테이블 항목(pte)을 가져온 다음 swapwrite()를 사용하여 디스크로 Swap 합니다.

(3) 또한 'PTE_P' 비트를 지우고 Swap offset 을 설정하여 페이지가 교체되었음을 반영하기 위해 페이지 테이블의 페이지 상태를 변경합니다.

(4) 페이지를 Swap Out 한 후 Swap Out 된 페이지의 물리적 메모리를 새 할당에 재사용합니다.

(5) Swap 공간에 여유 공간이 있는지 확인하고 그렇지 않으면 Panic 을 일으킵니다.

(6) 마지막으로 Free Page 수를 하나씩 줄이고 잠긴 경우 잠금을 해제합니다.

2. page_fault_handler() 함수의 구현 :

```
void page_fault_handler(struct trapframe *tf) {
  uint faulting_address;
  struct proc *curproc = myproc();

  // CR2 (제어 레지스터 2) 읽기. 이 레지스터는 잘못된 주소 저장
  asm volatile("movl %%cr2, %0" : "=r" (faulting_address));
  //faulting_address = tf->cr2;
  pde_t *pgdir = curproc->pgdir;
  pte_t *pte;
  struct page *pg;

  // 페이지 디렉터리 항목이 없으면 (즉, 페이지 매핑이 안 된 경우),
  // 페이지 결함은 정당하며 존재하지 않는 페이지 오류로 처리.
  if(!(pgdir[PDX(faulting_address)] & PTE_P)) {
    // Handle page not present error...
    cprintf("Page not present error at address: %x\n", faulting_address);
    curproc->killed = 1; // Terminate the process
    return;
  } else {
    // 페이지 테이블 항목 포인터 얻기
    pte = (pte_t*) P2V(PTE_ADDR(pgdir[PDX(faulting_address)]));
    // 이 페이지가 swapped out 된 것인지 확인
    if (pte[PTX(faulting_address)] & ~PTE_P) {
      char *mem = kalloc();
      if (!mem) {
        panic("out of memory");
      }
      memset(mem, 0, PGSIZE);

      // 스왑될 페이지의 블록 번호를 가져오기
      int blkno = pte[PTX(faulting_address)] & ~0xFFF  ;

      if (blkno < 0 || blkno >= SWAPMAX ) {
        cprintf("\nfaulting adress : %x",faulting_address);

        cprintf("\nblkno : %d\n", blkno);
        panic("page_fault_handler: blkno exceeded range");
      }
      swapread(mem, blkno);


      // pte 업데이트
      pte[PTX(faulting_address)] = V2P(mem) | PTE_P | PTE_W | PTE_U;

      // 이 페이지에 대한 구조체 페이지 찾기
      struct page *page = &pages[blkno >> PTXSHIFT];
```

```
      if (page->vaddr != 0) {
        panic("page_fault_handler: page was not swapped out");
      }
      pg = &pages[V2P(mem) >> PTXSHIFT];
      if(pg < pages || pg >= pages + (PHYSTOP/PGSIZE)) {
        panic("page_fault_handler: out of pages array bounds");
      }

      // 이 페이지를 LRU 목록의 끝에 추가
      if (page_lru_head == (void*)0) {
        page_lru_head = pg;
        pg->next = pg->prev = pg;
      } else {
        pg->next = page_lru_head;
        pg->prev = page_lru_head->prev;
        page_lru_head->prev->next = pg;
        page_lru_head->prev = pg;
      }
      num_lru_pages++;

      return;
    }else if (!(pte[PTX(faulting_address)] & PTE_P)) {
      cprintf("Page not present error at address: %x\n", faulting_address);
      curproc->killed = 1; // Terminate the process
      return;
      }
    // Handle page not present error...
  }
}
```

사용 가능한 메모리가 없을 때 Swap 과정을 kalloc() 함수를 통해 진행합니다. page_fault_handler() 함수는 존재하지 않는 Page 또는 Swapped Out Page 에서 생성된 오류를 적절하게 처리하는 데 중점을 둡니다.

Swapped Out Page 가 감지되면 page_fault_handler()는 kalloc() 함수 구현된 Clock Algorithm 을 활용하여 해당 Page 에 대한 처리를 시도합니다.

3. Pages[] array 를 이용한 page management :

```
struct page pages[PHYSTOP/PGSIZE];
```

선언 되어있는 pages[] array 를 통해 kalloc()과 kfree() 함수가 실행될 때마다, page 가 등록된 영역에서의 replacement 가 일어난 정보를 저장 및 수정합니다. Kinit2() 함수에서 pages array 는 초기화 됩니다. 해당 코드는 앞선 kalloc() 함수 설명에 포함되어있으며, kfree() 함수에서 사용된 코드는 다음과 같습니다.

```
void
kinit2(void *vstart, void *vend)
{
  num_free_pages = 0;
  freerange(vstart, vend);
  num_free_pages = PHYSTOP/PGSIZE;
  for (int i = 0; i < PHYSTOP/PGSIZE; ++i) {
    pages[i].vaddr = (char *) (i * PGSIZE);
  }

  kmem.use_lock = 1;
}
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;
  int pageIndex = (V2P((char *) r)) / PGSIZE;
  //pages 에 존재하는 vaddr 를 0 으로 바꾸음으로써 다시 할달되기 전에 사용하지
못하는 page 로 변경
  pages[pageIndex].vaddr = 0;
  r->next = kmem.freelist;
  kmem.freelist = r;
  num_free_pages++;

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

4. **LRU 알고리즘에서 lru_list 에 존재하는 page 를 다루는데에 사용되는 함수 설명 :**

다음 내용은 vm.c 의 코드를 변경하며 이루어집니다.

vm.c 에서 전역변수로 다음과 같은 swap_bitmap [] array 를 선언합니다.

```
char swap_bitmap[SWAP_BITMAP_SIZE];
void remove_page_from_lru(struct page* pg);
```

여기서 쓰인 SWAP_BITMAP_SIZE 는 param.h 에 선언되어 사용됩니다.

```
#define SWAP_BITMAP_SIZE (SWAPMAX / PGSIZE / 8)
```

```
void
freevm(pde_t *pgdir)
{
  uint i;

  if(pgdir == 0)
    panic("freevm: no pgdir");
  deallocuvm(pgdir, KERNBASE, 0);
  for(i = 0; i < NPDENTRIES; i++){
    if(pgdir[i] & PTE_P){
      char * v = P2V(PTE_ADDR(pgdir[i]));
      kfree(v);
    }else if(pgdir[i] & PTE_U){
      int swap_offset = PTE_ADDR(pgdir[i]);

      // 가상(임의)의 swap space 에 존재하는 bitmap 제거
      clear_swap_space(swap_offset);

      //  LRU list 에서 해당 page 제거
      remove_page_from_lru(&pages[swap_offset / PGSIZE]);
    }
  }
  kfree((char*)pgdir);
}

void
clearpteu(pde_t *pgdir, char *uva)
{
  pte_t *pte;

  pte = walkpgdir(pgdir, uva, 0);
  if(pte == 0)
    panic("clearpteu");

  if (*pte & PTE_P) {
```

```
    *pte &= ~PTE_U;
  } else if (*pte & PTE_U) {
    int swap_offset = PTE_ADDR(*pte);

    // 해당 swap_offset 을 가진 bitmap space 초기화
    clear_swap_space(swap_offset);

    // LRU list 에서 해당 offset 을 가진 page 제거
    remove_page_from_lru(pages + (swap_offset / PGSIZE));
  }
}
```

가상 메모리에 할당된 데이터를 free 하는 freevm() 함수를 수정하였습니다. 코드 설명은 주석과 같습니다. 또한, 여기와 page_fault_handler() 함수에서 쓰일 LRU list 를 관리하는 함수는 아래와 같습니다.

```
int find_free_swap_space(void) {
  int i, j;
  for (i = 0; i < SWAP_BITMAP_SIZE; i++) {
    if (swap_bitmap[i] != 0xFF) {  // If not all bits are set
      for (j = 0; j < 8; j++) {
        if (!(swap_bitmap[i] & (1 << j))) {  // If this bit is not set
          swap_bitmap[i] |= (1 << j);  // Set the bit
          return i * 8 + j;  // Return the index of the bit
        }
      }
    }
  }
  panic("Out of swap space");
}

void clear_swap_space(int swap_offset) {
  // SWAPMAX 크기의 전역 배열 swap_bitmap[]이 있다고 가정합니다.
  // 배열의 각 요소는 스왑 공간의 한 블록을 나타냅니다.
  // 블록의 크기는 페이지의 크기와 같습니다.
  // 값이 0 이면 블록이 비어 있음을 나타내고, 값이 1 이면 블록이 사용 중임을 나타냅니다.
  swap_bitmap[swap_offset] = 0;
}

void remove_page_from_lru(struct page* pg) {
  // page 가 list 에 오직 하나만 존재할 때
  if (pg->next == pg) {
    page_lru_head = (void*)0;
  } else {
    // list 에 다른 page 가 존재할 때, head 를 넘겨주고 list 에서 삭제.
    if (page_lru_head == pg) {
```

```
      page_lru_head = pg->next;
    }
    pg->next->prev = pg->prev;
    pg->prev->next = pg->next;
  }
  num_lru_pages--;
}
```

5. 실행 결과 :

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

#define M 1024  // Size of the memory chunk in kilobytes

void child_proc() {
    int *m;
    int i = 0;

    while(1) {
        m = (int*) sbrk(M * 1024);
        if((int)m == -1) {
            printf(1, "sbrk failed, i = %d\n", i);
            exit();
        }

        // Fill the memory with data (to ensure it's not optimized away)
        for(int j = 0; j < M * 1024 / sizeof(int); j++)
            m[j] = i;

        printf(1, "memory allocated, i = %d\n", i);
        i++;
    }
}

int main(void) {
    printf(1, "Starting swap test...\n");

    // Fork a child process to allocate memory
    if(fork() == 0) {
        child_proc();
    }
```

```
    // Parent process waits for the child to finish
    wait();

    printf(1, "Swap test finished.\n");
    exit();
}
```

위와 같은 test code 를 만들어 실행한 결과입니다. 해당 test code 는 sbrk()와 fork() 를 무한하게 실행하여 일정하게 커지는 process 를 계속 생성합니다.

```
memory allocated, i = 207
memory allocated, i = 208
memory allocated, i = 209
memory allocated, i = 210
memory allocated, i = 211
memory allocated, i = 212
memory allocated, i = 213
memory allocated, i = 214
memory allocated, i = 215
memory allocated, i = 216
memory allocated, i = 217
memory allocated, i = 218
memory allocated, i = 219
kalloc: Out Of memory
allocuvm out of memory
sbrk failed, i = 220
Swap test finished.
$
```

pages[] array 에 할당된 모든 page 영역이 다 쓰이고, 더 이상 swap 되지 못할 때 page 할당에서 벗어난 adress 를 통해 alloc 을 시도하고, 오류 메시지를 print 하고 종료가 되는 것을 확인할 수 있습니다.