Operating System Project04 Wiki

2019073309 윤무원

Design

페이지 참조 횟수 관리

CoW를 구현하기 위해 가장 먼저 생각한 것은 페이지 참조 횟수를 어떻게 관리하냐는 것이었다. 각 페이지 별로 참조 횟수를 추적해야하기에 단순하게 정수형 배열을 선언하여 인덱스를 페이지 번호로 활용하여 참조 횟수를 추적하였다. 또한 별도의 lock을 선언하여 참조 횟수를 증감할때 locking을 하여 동기화를 보장해주었다.

이렇게 선언한 자료 구조를 가지고 페이지 참조 증감을 관리하였다. 새로운 페이지가 생성될 때마다 해당 페이지의 참조 횟수를 1로 설정하고 페이지가 free될때도 프로세스가 특정 페이지를 free 할때 해당 페이지의 참조 횟수를 확인하여 0일때만 실제로 free하여 freelist에 반환하였다. 0이 아닌 경우엔 그저 참조 횟수를 1 감소하였다.

fork의 동작 수정

CoW는 기본적으로 프로세스가 새로 만들어질때는 새로운 페이지를 할당 받는 것이 아닌, 부모 프로세스의 페이지를 공유해야하므로 기존의 fork 방식을 수정할 필요가 있었다.

기존의 fork

1. 자식 프로세스가 새로운 페이지 테이블을 할당 받는다.
2. 부모 프로세스의 크기와 동일하게 페이지를 새로 생성한다.
3. 부모 프로세스의 페이지 내용을 그대로 복사하여 새로운 페이지에 채워놓고 자식 프로세스에게 할당한다.

나는 이를 수정하여 아래와 같은 과정을 수행하는 새로운 fork 함수를 구현하였다.

새로운 fork

1. 자식 프로세스가 새로운 페이지 테이블을 할당 받는다.

2. 부모 프로세스의 페이지 테이블을 순회하여 페이지의 물리 주소를 자식 프로세스의 페이지 테이블에 매핑한다.
3. 해당 페이지의 권한을 읽기 전용으로 변경하고 참조 횟수를 증가시킨다.

Make a copy 구현

여러 프로세스가 동일한 페이지를 공유하고 있기 때문에 특정 프로세스가 공유된 페이지를 수정하려고 하면 이를 감지해서 페이지의 내용을 새로운 페이지에 복사하여 페이지 테이블에 매핑시켜준 후 새로운 페이지에 수정을 하도록 해야한다.

앞서 fork를 구현할때 공유되는 페이지를 읽기 전용으로 변경하였기 때문에 특정 프로세스가 해당 페이지에 수정을 시도하면 페이지 폴트가 발생하게 된다. 그렇기에 페이지 폴트 핸들러에서 페이지를 복사해서 새로운 페이지를 수정을 시도한 프로세스의 페이지 테이블에 매핑하고 수정을 하도록 구현하였다.

여기서 새로운 페이지가 생성될때 참조 횟수를 1로 설정하고 페이지 수정을 시도한 프로세스는 더이상 기존의 페이지를 참조하지 않으므로 기존 페이지의 참조 횟수를 1 감소하였다.

만약 페이지 폴트가 발생한 페이지의 참조 횟수가 1이라면 다른 프로세스는 해당 페이지를 참조하지 않는 것이므로 새로운 페이지를 만들 필요 없이 해당 페이지에 수정을 할 수 있도록 구현하였다.

Implement

ref_table

```
// kalloc.c
struct {
  struct spinlock lock;
  int count[PHYSTOP >> PTXSHIFT];
} ref_table;
```

`PHYSTOP` : 피지컬 메모리 주소의 TOP을 나타내는 변수이다. (= 0xE000000)

`PTXSHIFT` : 피지컬 메모리 주소로 페이지 테이블 엔트리의 오프셋을 얻을 수 있는 변수이다. (=12)

`lock` : 참조 횟수를 증감할때 동기화를 위한 lock 변수이다.

물리 페이지를 쉽게 관리하기 위해 인덱스로 피지컬 주소를 PTXSHIFT로 나눈 값을 활용하였다. 이렇게 하면 특정 피지컬 주소에 해당하는 페이지 번호로 변환되어 배열에 접근할 수

있기 때문이다.

kalloc()

```
// kalloc.c
char*
kalloc(void)
{
  struct run *r;
  uint pa;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);

  pa = V2P((char*)r);

  if(kmem.use_lock){
    incr_refc(pa);
  }

  return (char*)r;
}
```

페이지가 새로 생성되면 해당 페이지의 참조 횟수를 1로 설정해주었다.

kfree()

```
// kalloc.c
void
kfree(char *v)
{
  struct run *r;
```

```
  if(kmem.use_lock){
    decr_refc(V2P(v));
    if(get_refc(V2P(v)) != 0)
      return;
  }

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

실제로 페이지를 free하기 전에 먼저 해당 페이지의 참조 횟수를 1 감소시킨 후 만약 0이 아니라면 아직 다른 프로세스가 해당 페이지를 참조하고 있는 것이므로 free를 하지 않고 함수를 종료하였다. 참조 횟수가 0이 된다면 더이상 해당 페이지를 참조하는 프로세스가 없는 것이므로 그때 free를 진행하여 freelist에 페이지를 반환하였다.

incr_refc() & decr_refc() & get_refc()

```
// kalloc.c
void incr_refc(uint pa) {
  acquire(&ref_table.lock);
  ref_table.count[pa >> PTXSHIFT]++;
  release(&ref_table.lock);
}

void decr_refc(uint pa) {
  acquire(&ref_table.lock);
  ref_table.count[pa >> PTXSHIFT]--;
```

```
  release(&ref_table.lock);
}

int get_refc(uint pa) {
  int count;
  acquire(&ref_table.lock);
  count = ref_table.count[pa >> PTXSHIFT];
  release(&ref_table.lock);
  return count;
}
```

`incr_refc()` : 인자로 받은 피지컬 주소에 해당하는 페이지의 참조 횟수를 1 늘리는 함수이다.

`decr_refc()` : 인자로 받은 피지컬 주소에 해당하는 페이지의 참조 횟수를 1 감소하는 함수이다.

`get_refc()` : 인자로 받은 피지컬 주소에 해당하는 페이지의 참조 횟수를 반환하는 함수이다.

세 함수 모두 인자로 받은 피지컬 주소와 PTXSHIFT를 활용하여 배열의 인덱스로 접근하였다. 또한 `ref_table.lock` 을 활용하여 locking을 사용해 동기화를 보장해주었다.

CoW_copyuvm()

```
// vm.c
int CoW_copyuvm(pde_t *pgdir, pde_t *newpgdir, uint sz){
  pte_t *pte;
  uint pa, i, flags;

  for(i = 0; i < sz; i += PGSIZE){
    if ((pte = walkpgdir(pgdir, (void *)i, 0)) == 0)
      return -1;
    if (!(*pte & PTE_P))
      return -1;
    pa = PTE_ADDR(*pte);
    *pte &= ~PTE_W; // change parent's physical pages with re
    flags = PTE_FLAGS(*pte);

    if (mappages(newpgdir, (void*)i, PGSIZE, pa, flags) < 0)
```

```
      goto bad;

    incr_refc(pa);
  }

  lcr3(V2P(pgdir)); // TLB flush

  return 0;

bad:
  freevm(newpgdir);
  return -1;
}
```

fork를 통해 새로운 프로세스가 생성될때 부모 프로세스의 물리 페이지를 똑같이 가리키게 하는 함수이다. `walkpgdir()` 를 활용하여 부모 프로세스의 페이지 테이블을 순회하여 페이지의 권한을 읽기 전용으로 바꾸고 자식 프로세스의 페이지 테이블에 매핑시켰다. 이후 해당 페이지의 참조 횟수를 1 증가시키고 TLB를 flush 해주었다.

fork()

```
// proc.c
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // allocate new pgdir
  if((np->pgdir = setupkvm()) == 0){
    kfree(np->kstack);
    np->kstack = 0;
```

```
    np->state = UNUSED;
    return -1;
  }

  ....

  if (CoW_copyuvm(myproc()->pgdir, np->pgdir, myproc()->sz) <
    freevm(np->pgdir);
    np->state = UNUSED;
    return -1;
  }

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

allocproc을 통해 프로세스가 하나 생성되면 `setupkvm()` 을 활용하여 새로운 페이지 테이블을 할당하였다. 이후 `CoW_copyuvm()` 을 호출해 부모 프로세스의 물리 페이지를 자식 프로세스의 페이지 테이블에 똑같이 매핑시켜주었다.

CoW_handler()

```
// vm.c
void CoW_handler(void){
  uint va = rcr2(); // virtual address where page fault
  pte_t *pte = walkpgdir(myproc()->pgdir, (void *)va, 0);

  if (va >= KERNBASE || va >= myproc()->sz) {
    cprintf("invalid address range\n");
    myproc()->killed = 1;
    return;
  }
```

```
  if (pte && !(*pte & PTE_W)) {
    uint pa = PTE_ADDR(*pte);

    if(get_refc(pa) == 1){ // if refc is 1, just modify permi
      *pte |= PTE_W;
    } else {
      char *mem = kalloc();

      memmove(mem, (char*)P2V(pa), PGSIZE);

      *pte = V2P(mem) | PTE_P | PTE_W | PTE_U; // point to ne

      decr_refc(pa); // decrease refc old page
    }

    lcr3(V2P(myproc()->pgdir)); // TLB flush
  }
}
```

페이지 폴트가 발생했을때 해당 페이지의 가상주소를 이용하여 물리 페이지에 접근, 해당 페이지를 복사하여 새로운 페이지를 만들고 이를 페이지 테이블에 매핑한다. 가상주소가 잘못된 범위에 속해 있을 경우 에러 메시지를 출력하고 프로세스를 종료한다. 만약 페이지의 참조 횟수가 1이라면 자신 외에는 아무도 참조를 하고 있지 않는 것이므로 새로운 페이지를 만들 필요 없이 쓰기 권한을 부여하였다. 새로운 페이지를 만들어 이를 새롭게 페이지 테이블에 매핑하였으면 기존 페이지는 참조 횟수를 1 감소하였다. 새로운 페이지에 대한 참조 횟수는 `kalloc()` 에서 이미 1로 설정하였으므로 여기선 따로 처리해주지 않았다. 모든 동작이 끝난 후에는 TLB를 flush 하였다.

trap()

```
// trap.c
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
```

```
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  // page fault handler
  if(tf->trapno == T_PGFLT){
    CoW_handler();
    return;
  }

  ...

}
```

페이지 폴트가 일어났을때 `CoW_handler()` 를 호출하여 Copy on Write를 수행할 수 있게 구현하였다.

countfp() & countvp() & countpp() & countptp()

```
// kalloc.c
int countfp(void){
  int count = 0;
  struct run *r;
  acquire(&kmem.lock);
  for(r = kmem.freelist; r != 0; r = r->next)
    count++;
  release(&kmem.lock);
  return count;
}

// vm.c
int countvp(void){
  struct proc *p = myproc();
  return PGROUNDUP(p->sz) / PGSIZE;
}
```

```
int countpp(void){
  int count = 0;
  pte_t *pte;
  struct proc *p = myproc();
  for(int i = 0; i < p->sz; i += PGSIZE){
    pte = walkpgdir(p->pgdir, (void *)i, 0);
    if (pte && (*pte & PTE_P))
      count++;
  }
  return count;
}

int countptp(void){
  int count = 0;
  pde_t *pde;
  struct proc *p = myproc();
  pde = p->pgdir;

  count++; // count page for page directory

  for(int i = 0; i < NPDENTRIES; i++){
    if(&pde[i] && (pde[i] & PTE_P)){
      count++;  // count page for page table
    }
  }

  return count;
}
```

`countfp()` : 모든 free page는 `kmem` 구조체 내의 `freelist` 에 링크드 리스트로 관리가 되어 있으므로 `kmem.freelist` 를 순회하여 free page의 갯수를 반환한다.

`countvp()` : 프로세스에게 할당된 메모리 공간의 사이즈가 `proc` 구조체 내의 `sz` 에 저장되므로 `p->sz` 값을 `PGROUNDUP()` 을 통해 페이지 크기의 배수로 올림을 해주고 `PGSIZE` 로 나누면 프로세스에게 할당된 로지컬 페이지의 수가 계산된다.

`countpp()` : 현재 프로세스의 페이지 테이블을 `walkpgdir()` 를 이용하여 탐색하고 유효한 페이지 테이블 엔트리인지 확인한다. `pte` 가 NULL이 아니고, 페이지 테이블 엔트리에 `PTE_P` 가 설정되어 있는지 확인하여 만족하면 해당 페이지 테이블 엔트리는 유효한 것이므로 이런 방식으로 모든 엔트리를 검사하여 갯수를 반환한다.

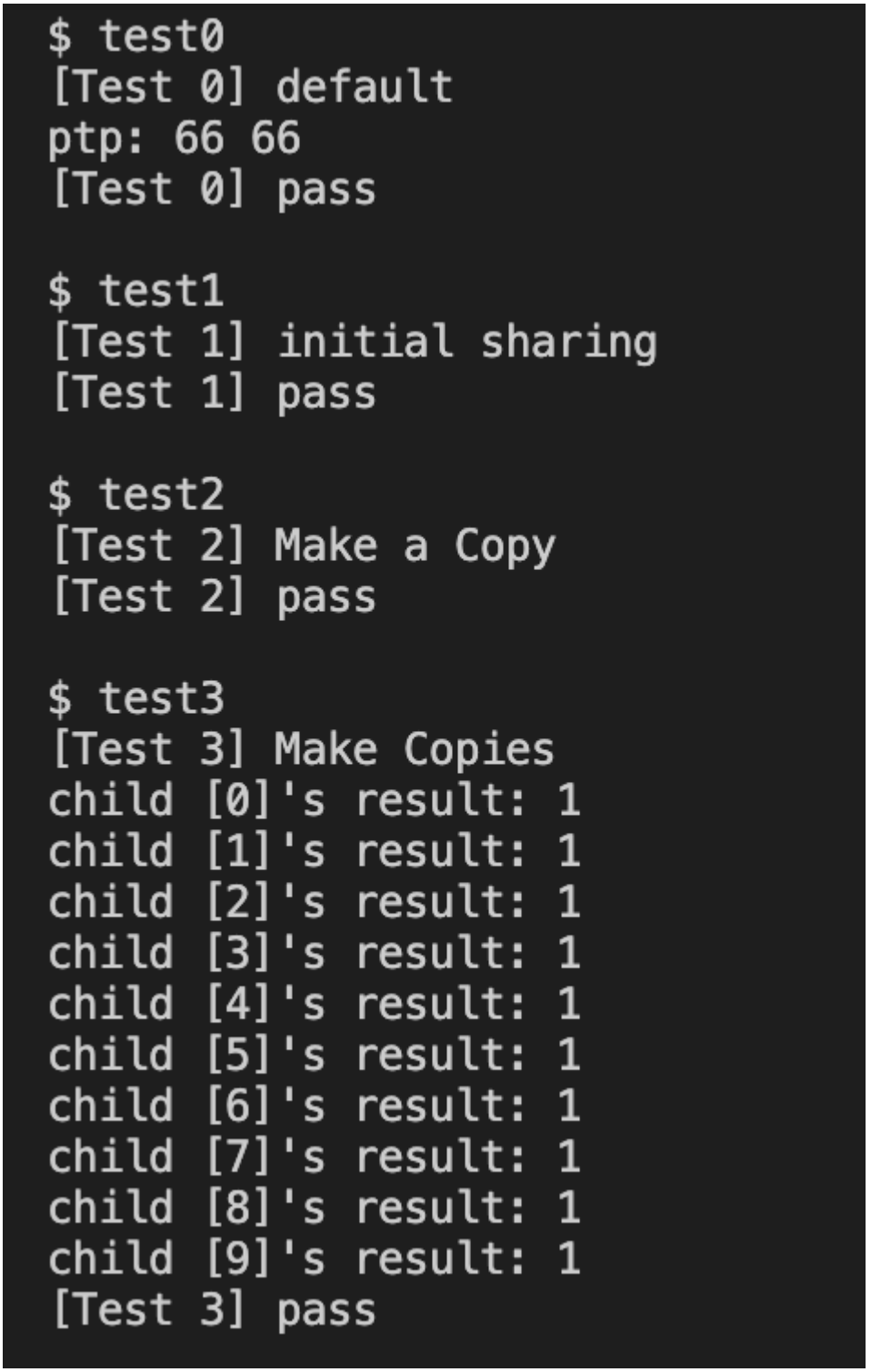
`countptp()` : 페이지 테이블을 저장하는데 사용된 페이지의 수를 반환한다. 페이지 디렉토리에 페이지가 1개 사용되기에 이를 포함시켜주고 페이지 디렉토리를 순회하여 유효한 디렉토리 엔트리(페이지 테이블)의 갯수를 합산하여 반환한다.

Result

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00



Booting from Hard Disk...
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
```

xv6가 정상적으로 부팅된다.

test0 & test1 & test2 & test3 실행결과



test0, test1, test2, test3 모두 정상적으로 실행되고 결과 또한 테스트 예시와 동일한 결과를 나타낸다.

Trouble shooting

kalloc(), kfree() 내에서 잘못된 lock 사용

```
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00


Booting from Hard Disk..
```

`kalloc()`과 `kfree()` 내에서 `incr_refs()` 와 `decr_refs()` 를 호출하니 위와 같이 xv6가 부팅되다가 멈춰버리는 오류가 발생하였다. 이런저런 방법으로 디버깅해보고 코드를 뜯어보니 `ref_table.lock`을 사용하는 부분이 문제라는 것을 알게 되었다. 결과적으로는 `incr_refs()` 와 `decr_refs()` 를 호출하기 전에, 즉 `ref_table.lock`을 사용하기 전에 `if(kmem.use_lock)` 로 조건을 걸었더니 정상으로 작동하였다. `kalloc()` 과 `kfree()` 내부에서 `kmem.lock` 을 사용할때도 똑같은 조건을 설정하는 것으로 보아 xv6의 동작 방식과 관련이 있는 것으로 추정되지만 정확한 이유는 알아내지 못하였다.