OS Project04 - Wiki

2019040591
컴퓨터소프트웨어학부
박진수

Implementation

1. kalloc.c

```
struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
  uint freepagecnt; // Count the number of total free pages i
  // array for counting the numbers of references for each ph
  uint pgrefcnt[PHYSTOP / PGSIZE]; // PHYSTOP is the top of t
} kmem;
```

시스템의 총 빈 physical 페이지 개수를 세는 변수인 freepagecnt와 각 physical 페이지를 참조하는 가상페이지가 몇 개 인지에 대한 정보를 담는 pgrefcnt 배열을 kmem 구조체 안에 추가했습니다. pgrefcnt[PHYSTOP / PGSIZE]의 경우, PHYSTOP은 physical memory adress의 제일 마지막 부분, 즉 물리 주소 공간의 전체 크기를 나타내고, 해당하는 주소를 하나의 물리 페이지의 크기로 나눠줌으로써 전체 physical page의 개수 만큼의 크기를 갖는 배열이 됩니다.

```
void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  kmem.freepagecnt = 0; // initialize the number of free phys
  freerange(vstart, vend);
}
```

kinit1()는 메인함수의 main()에서 제일 처음 호출되는 함수입니다. 이 함수는 부팅 이후, 커널이 사용할 physical memory 영역을 initialize 하는 역할을 하는데, 커널 텍스트 영역과 데이터 영역이 적혀있는 부분의 다음 주소인 end 부터 시작하여 4MB 까지의 메모리를 초기화해줍니다. 전체 커널이 완성되기 전에, 커널 시스템의 동작을 위해 필수적으로 필요한 부분의 physical memory를 먼저 할당(free)하는 역할을 합니다. freerange()를 호출하기 전에, freepagecnt를 0으로 먼저 초기화해줍니다. freerange(vstart, vend)는 가상 메모리 주소 vstart, vend 사이의 역역에 대응되는 physical 페이지를 free해주는 역할을 합니다. 이후, freerange()내의 kfree()가 호출되면서 해당하는 physical page가 free되고, freepagecnt를 1씩 늘려주게 됩니다. 이러한 방식으로 kinit1()이 실행되면 해당하는 부분의 physical page가 초기화되고 해당하는 만큼의 freepagecnt가 늘어나게 됩니다.

```
void
kinit2(void *vstart, void *vend)
{
  freerange(vstart, vend);
  kmem.use_lock = 1;
}
```

kinit2()는 메인함수에서 kinit1()이 호출된 이후 커널의 시스템들이 초기화 되고 나서, 나머지 physical memory 부분을 초기화해주는 부분입니다. 여기서도 마찬가지로 freerange(vstart, vend)가 호출되고, 또 해당하는 virtual address 에 해당하는 만큼의 phyiscal page가 초기화 되고, freepagecnt도 늘어나게 됩니다.

```
void
freerange(void *vstart, void *vend)
{
  char *p;
  uint pa; // physical address
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
  {
    pa = V2P(p);
    kmem.pgrefcnt[pa / PGSIZE] = 0; // initialize page ref co
    kfree(p);
```

```
  }
}
```

freerange() 함수는 free 해줄 부분의 가상 메모리 주소를 인자로 받는데, 여기서 해당하는 부분을 phyiscal page로 초기화시켜줍니다. 앞서 각각의 physical page를 virtual page 몇 개가 참조하는지 추적하기 위해 pgrefcnt 배열을 만들었었는데, 여기서 물리페이지들이 초기화 될 때 마다 해당하는 pgrefcnt 배열의 요소를 0으로 초기화해줍니다.

```
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);

  uint pa = V2P(v); // physical address of v

  if(get_refc(pa) >= 1) { // first, just decrement the number
    decr_refc(pa);
  }
  if(get_refc(pa) == 0) { // free the page if ref count is 0
    // Fill with junk to catch dangling refs.
    memset(v, 1, PGSIZE);
    r = (struct run*)v;
    r->next = kmem.freelist;
    kmem.freelist = r;
    kmem.freepagecnt++;
  }
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

kfree()의 경우, virtual address v에 대응되는 물리 페이지를 free해주는 역할을 하는데, 이번 과제의 경우 하나의 물리 페이지를 여러 virtual page가 참조할 수 있기에, 그에 대응되게 수정했습니다. get_refc(pa)의 경우, 물리 주소인 pa를 인자로 받아 해당되는 주소의 물리 페이지를 참조하는 virtual page들이 몇 개 있는지를 반환하는 함수인데, 이 숫자가 1 이상이라는 것은 해당하는 물리 페이지를 참조하고 있는 가상 페이지가 존재한다는 뜻이므로, decr_ref(pa)를 통해 1 감소시켜줍니다. 그런데 만약 공유가 되고 있지 않은 물리 페이지였다면, 해당 값이 1에서 0이 될 것이고, 만약에 공유가 되고 있었다면 1 이상의 값을 가지게 됩니다. 1 이상이라면 그냥 물리 페이지를 참조 횟수를 나타내는 값을 1만큼 빼주면 됩니다. 그리고 밑의 if 문에서 해당 값이 0인지 확인합니다. 해당 if 문은 두가지 경우에 실행되는데, 하나는 가상 페이지에 의해 참조되고 있던 물리 페이지가 free되는 경우와, 최초로 해당 물리 페이지가 초기화 될 때 free 되는 경우입니다. 이 경우, 기존 코드와 동일하게 free 로직을 진행하고, freepagecnt값을 1 증가시키는 부분만 추가했습니다.

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

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

  r = kmem.freelist;
  if(r) {
    kmem.freelist = r->next;
    incr_refc(V2P((char*)r)); // increase from 0 to 1
    kmem.freepagecnt--;
  }
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

kalloc()의 경우, kreelist에 메달려있는 free 상태의 physical page가 있을 경우, 해당하는 페이지를 freelist에서 떼어내 해당 physical page에 대응되는 virtual address를 넘겨주는 함수입니다. 결국 해당 물리 주소를 다른 가상 페이지가 참조하게 되므로, incr_refc()를 통해 값을 0에서 1로 증가시켜줍니다. 그리고 freepagecnt의 개수도 1 감소시켜줍니다.

```
void incr_refc(uint pa)
{
  kmem.pgrefcnt[pa / PGSIZE]++;
}

void decr_refc(uint pa)
{
  kmem.pgrefcnt[pa / PGSIZE]--;
}

int get_refc(uint pa)
{
  int refcnt = (int)kmem.pgrefcnt[pa / PGSIZE];

  return refcnt;
}
```

위의 함수들은 과제의 명세에 요구되었던 함수들로, 각각의 물리 주소에 해당되는 참조 횟수를 1 증가시키거나, 1 감소시키거나, 해당 횟수의 값을 반환하는 역할을 합니다. locking의 경우, 해당 함수를 호출시키는 부분에서 구현하였습니다. 왜냐하면 이들 함수는 acquire(&kmem.lock)이 이미 적용된 부분에서도 호출되고, locking이 걸려있지 않은 부분에서도 호출되는데, locking이 적용되지 않은 부분에서만 이들 함수를 호출하기 전에 locking을 위해 생성한 struct spinlock pgrefcnt_lock에 해당되는 acquire(&pgrefcnt_lock);를 호출하여 locking을 구현하였습니다.

```
int countfp(void)
{
  int freepages;
  acquire(&kmem.lock);
  freepages = (int)kmem.freepagecnt;
  release(&kmem.lock);

  return freepages;
}
```

위의 함수는 과제의 명세에 요구되었던 countfp 시스템콜을 위해 구현한 커널 함수로, freepagecnt에 접근하기 위해 kalloc.c에 구현하였습니다. 나머지 시스템콜을 위한 커널 함수들의 경우, vm.c에 구현하였습니다. countfp()가 호출되는 부분에서는 kmem의 멤버 변수인 freepagecnt에 접근하기 때문에 acquire(&kmem.lock);을 해주었고, freepagecnt의 값을 받아와 반환하게 해주었습니다.

2. vm.c

```
#include "spinlock.h"
...
// For project 4
struct spinlock pgrefcnt_lock;
```

get_refc(), incr_refc(), decr_refc()를 호출할 때 locking을 하기 위해 추가하였습니다.

```
void
seginit(void)
{
  struct cpu *c;

  // Map "logical" addresses to virtual addresses using ident
  // Cannot share a CODE descriptor for both kernel and user
  // because it would have to have DPL_USR, but the CPU forbi
  // an interrupt from CPL=0 to DPL=3.
  c = &cpus[cpuid()];
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USE
  c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
  lgdt(c->gdt, sizeof(c->gdt));

  // Project 4
  initlock(&pgrefcnt_lock, "pgrefcnt");
}
```

메인 함수에서 vm.c와 관련하여 초기화를 위해 제일 먼저 호출되는 함수여서 해당 함수의 마지막 부분에 pgrefcnt_lock을 초기화해주었습니다.

```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){ // each page in the parent
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0) // finds
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");

    *pte &= (~PTE_W); // disable writable flag
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);

    // delete codes that allocate a new page of physical memo
    // and that copies the content form the old page to the n

    if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0) {
      goto bad;
    }
    acquire(&pgrefcnt_lock);
    incr_refc(pa); // increment the number of reference of th
    release(&pgrefcnt_lock);
  }

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

bad:
```

```
  freevm(d);
  return 0;
}
```

이 함수는 새로운 프로세스를 생성할 때, 부모 프로세스의 가상 메모리 공간을 자식 프로세스에게 복사해주고, 새로운 페이지 디렉토리를 반환해주는 역할을 합니다. 해당 과제에서는 CoW를 구현해야 하므로, 기존 XV6의 경우, 자식 프로세스에게 새로운 physical memory를 free하고 할당해주지만, 여기서는 physical memory를 할당해주는 부분을 지우고, 자식 프로세스가 새로운 페이지 테이블을 할당 받은 이후, 부모와 자식의 페이지 테이블이 동일한 physical 페이지를 가리키도록 해주었습니다. 다만, writable flag를 disable 시켜주어, 추후에 쓰기가 발생하여 서로의 페이지 내용이 달라져야 할때는, page fault를 발생시켜 CoW_handler()를 호출시켜 새로운 물리 페이지를 생성하고 그때 복사를 진행하도록 하였습니다. 나머지 flag는 부모와 동일합니다. 그리고 이 과정에서 locking을 하고, ref_count를 1 증가시켜주었습니다. 그리고 lcr3(V2P(pgdir));를 통해 TLB를 flush 해주었습니다.

```
void CoW_handler(void)
{
  pte_t *pte;
  uint pa;
  uint refcnt;
  uint fault_addr = rcr2(); // read the CR2 register to get t

  if (myproc() == 0) {
    panic("no current process");
  }

  // walk the page table to find the page table entry for the
  pte = walkpgdir(myproc()->pgdir, (void *)fault_addr, 0);
  if (pte == 0 || !(*pte & PTE_P)) {
    // null check & check whether the page present bit is set
    cprintf("error: page not found or valid\n");
    myproc()->killed = 1; // prevent the propagation of error
    return;
  }

  if(!(*pte & PTE_W) && (myproc()->tf->err & 0x2)) { // if tr
    pa = PTE_ADDR(*pte); // extract the physical adress from
```

```
    acquire(&pgrefcnt_lock);
    refcnt = get_refc(pa); // return the number of references
    release(&pgrefcnt_lock);
    if(refcnt > 1) { // if more than 1 process are sharing a
      char *mem;
      if((mem = kalloc()) == 0) {
        cprintf("error: out of memory\n");
        myproc()->killed = 1;
        return;
      }
      memmove(mem, (char*)P2V(pa), PGSIZE); // copy data from
      *pte = V2P(mem) | PTE_FLAGS(*pte) | PTE_W; // preserves
      acquire(&pgrefcnt_lock);
      decr_refc(pa); // decrement the reference count by 1
      release(&pgrefcnt_lock);
    } else if(refcnt == 1) {
      *pte += PTE_W; // just enable writeable flag, no additi
    }
  }
  lcr3(V2P(myproc()->pgdir)); // Flush TLB
}
```

CoW_handler()의 경우, writable flag가 disable 되어있는 페이지에 write를 하려고 했을 때 page fault가 발생하고, trap.c에 추가한 page fault handler에서 실행되는 함수입니다. 페이지 폴트가 발생한 virtual address가 CR2 레지스터에 담기게 되는데, 해당 주소에 맵핑되는 pte에 실제 물리 페이지를 새로 생성해 가리키도록 하였습니다. 먼저 해당 가상 주소에 맵핑되는 pte가 유효한지 검사하고, 해당 fault가 writable flag가 disable 된 상황에서 쓰려고 하여 발생했는지 체크합니다. 맞다면, 먼저 에러가 난 물리 페이지를 몇개의 가상 페이지에서 참조하고 있었는지 확인합니다. 만약에 2개 이상의 페이지에서 참조하고 있었던 페이지라면, 실제로 physical memory를 새로 할당한 이후, 부모의 페이지와 동일한 내용으로 copy해줍니다. 이때 writable flag만 기존과 달리 enable 시켜줍니다. (최초로 생성되어 혼자만 해당 페이지를 사용하므로) 나머지는 부모와 동일하게 설정합니다. 만약에 1개의 페이지만 참조하고 있던 상황이라면 원래는 그냥 써도 되는 상황입니다. 이 경우에는 해당 page table entry의 writable flag만 다시 enable 시켜주어 쓸 수 있도록 해줍니다. 마찬가지로 이 경우에도 페이지 테이블 항목이 변경되었으므로, lcr3(V2P(myproc()->pgdir));를 통해 TLB flush를 진행해줍니다.

```
int countvp(void)
{
  struct proc *curproc = myproc();
  if (curproc == 0)
    return -1;

  int logical_pages = curproc->sz / PGSIZE;

  return logical_pages;
}
```

현재 프로세스의 가상 메모리 공간의 크기(virtual address space)를 페이지 하나의 크기로 나누어 총 virtual page의 개수를 구하여 반환합니다.

```
int countpp(void)
{
  struct proc *curproc = myproc();
  pde_t *pgdir;
  pde_t *pte;
  uint count = 0;
  uint va = 0;

  if (curproc == 0)
    return -1;

  pgdir = curproc->pgdir;

  while (va < KERNBASE) { // ensuring only user space is cons
    pte = walkpgdir(pgdir, (void *)va, 0); // retrieve the co
    if(pte && (*pte & PTE_P)) { // check whether pte exists a
      count++;
    }
    va += PGSIZE;
  }
  return count;
}
```

프로세스의 가상 메모리 주소 0 부터, kernel space의 주소가 시작되는 KERNBASE 사이의 user space pages 들을 순회하면서 walkpgdir를 통해 해당 pagedirectory에 유효한 물리 주소가 할당되어 있는지 확인하여(if(pte && (*pte & PTE_P))), 할당되어 있을 경우 0으로 초기화 된 count 변수를 1씩 증가시켜줍니다. 그리고 다 순회한 이후 count 변수를 리턴합니다.

```
int countptp(void)
{
  struct proc *curproc = myproc();
  pde_t *pgdir;
  pde_t *pde;
  pde_t *pgtab;
  uint pde_idx, pte_idx;
  uint count = 0;

  if (curproc == 0)
    return -1;

  pgdir = curproc->pgdir;

  count++; // count the page directory itself

  // Traverse the page directory
  for (pde_idx = 0; pde_idx < NPDENTRIES; pde_idx++) {
      pde = &pgdir[pde_idx];
      if (*pde & PTE_P) { // this page directory entry is pre
          count++; // count this pde
          pgtab = (pte_t*)P2V(PTE_ADDR(*pde));

          // Traverse the page table
          for (pte_idx = 0; pte_idx < NPTENTRIES; pte_idx++)
              if (pgtab[pte_idx] & PTE_P) {
                  count++;
              }
          }
      }
  }
```

```
  return count;
}
```

XV6는 2-level page table 구조를 가지고 있는데, 따라서 먼저 해당 프로세스의 루트 page directory를 위해 1을 추가시켜주었습니다. 그리고 해당 페이지 디렉토리의 페이지들을 순회하며 해당 페이지가 null이 아니고 PTE_P가 1, 즉 유효한 페이지인 경우에도 1을 추가시켜 주었습니다. 그리고 해당 페이지에 담겨 있는 페이지 테이블의 주소를 참조하여 마찬가지로 동일하게 유효하면 1을 추가시켜 주었습니다. 이 과정에서 자연스럽게 커널 페이지 테이블 매핑을 저장하는 페이지 테이블의 수도 구해지게 됩니다.

3. trap.c

```
void
trap(struct trapframe *tf)
{
...
switch(tf->trapno){
    // project 4
  case T_PGFLT:
    // Handler for page fault
    CoW_handler();
    break;
 ...
}
```

write_enable flage가 disable 되어있는 페이지에 write를 하려고 하면 pagefault 가 발생하게 되는데, 이 경우 해당 T_PGFLT를 이용해 handling 하여 앞서 설명한 CoW_handler를 실행시킵니다.

4. defs.h

```
// project 4
void            incr_refc(uint pa);
void            decr_refc(uint pa);
```

```
int             get_refc(uint pa);
void            CoW_handler(void);
int             countfp(void);
int             countvp(void);
int             countpp(void);
int             countptp(void);
```

아래의 커널 함수들은 vm.c와 kalloc.c 등 여러 c 파일에서 호출될 수 있기에 defs.h에 추가하였습니다.

5. 그 외 system call 구현을 위한 부분

sysproc.c

```
int
sys_countfp(void)
{
  return countfp();
}

int sys_countvp(void)
{
  return countvp();
}

int sys_countpp(void)
{
  return countpp();
}

int sys_countptp(void)
{
  return countptp();
}
```

syscall.h

```
#define SYS_countfp 22
#define SYS_countvp 23
#define SYS_countpp 24
#define SYS_countptp 25
```

syscall.c

```
extern int sys_countfp(void);
extern int sys_countvp(void);
extern int sys_countpp(void);
extern int sys_countptp(void);
...
static int (*syscalls[])(void) = {
...
[SYS_countfp] sys_countfp,
[SYS_countvp] sys_countvp,
[SYS_countpp] sys_countpp,
[SYS_countptp] sys_countptp,
};
```

user.h

```
// For project 4
int countfp(void);
int countvp(void);
int countpp(void);
int countptp(void);
```

usys.S

```
SYSCALL(countfp)
SYSCALL(countvp)
SYSCALL(countpp)
SYSCALL(countptp)
```

6. test 파일 추가를 위한 부분

test0.c, test1.c, test2.c,test3.c 파일들을 xv6-public 디렉토리 안에 추가해준 이후 Makefile에 아래와 같이 추가하였습니다.

Makefile

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _test0\
    _test1\
    _test2\
    _test3\

    ...
```

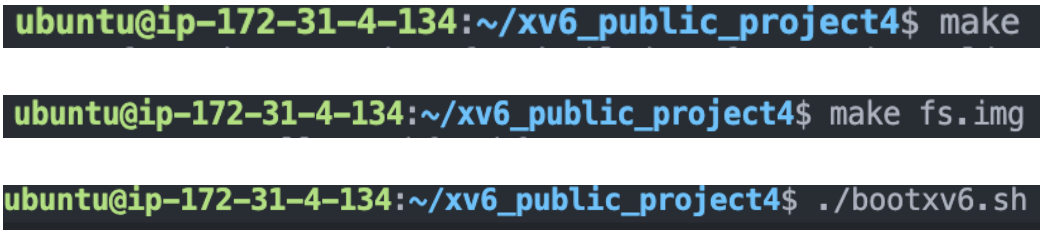
```
EXTRA=\
mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
printf.c umalloc.c test0.c test1.c test2.c test3.c\
README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
.gdbinit.tmpl gdbutil\
```

7. bootxv6.sh

```
#!/bin/bash

qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6
```

Result

해당 폴더는 xv6_public_project4로 되어 있지만, 제출시에 명세와 같이 폴더명을 xv6-public으로 제출하였습니다.

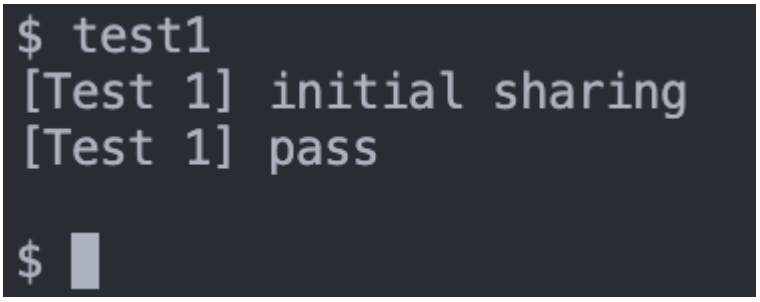


```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 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
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16388
echo           2 4 15240
forktest       2 5 9548
grep           2 6 18608
init           2 7 15828
kill           2 8 15268
ln             2 9 15124
ls             2 10 17752
mkdir          2 11 15368
rm             2 12 15344
sh             2 13 27988
stressfs       2 14 16260
usertests      2 15 67364
wc             2 16 17120
zombie         2 17 14936
test0          2 18 15956
test1          2 19 15524
test2          2 20 15644
test3          2 21 16320
console        3 22 0
$
```

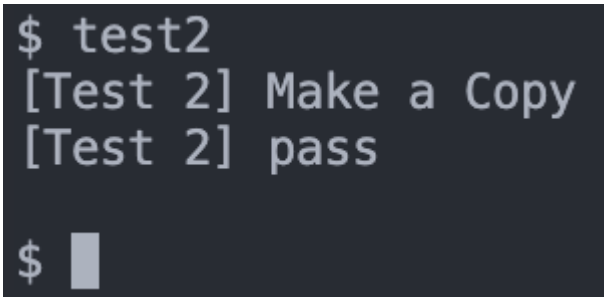
아래는 해당 xv6 shell에 test0을 입력한 결과화면입니다.

```
$ test0
[Test 0] default
ptp: 65605 65606
[Test 0] pass
```

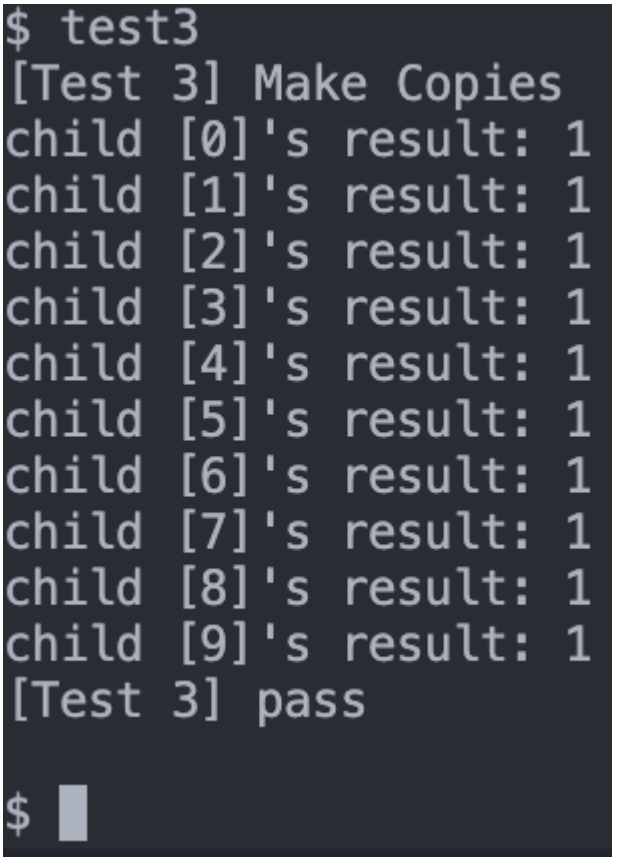
아래는 test1을 입력한 결과입니다.



아래는 test2를 입력한 결과입니다.



아래는 test3을 입력한 모습입니다.



Troubleshooting

맨 처음, incr_refc(), decr_refc(), get_refc()에서 locking을 구현하려고 할 때, 내부에서 lock을 걸면, 해당 함수를 호출하기 전 바깥쪽에서 걸린 kmem.lock으로 인해 deadlock이 발생하여, 부팅이 멈추는 문제가 있었습니다. 지난 보고서에서 gdb 사용법을 몰라 디버깅을 잘 하지 못했었는데, 이번에는 사용법을 공부하여, gdb를 실행하여 수정한 부분들에 break point를 걸고 continue, skip을 하면서 프로그램의 실행을 확인했는데, 내부에 lock을 걸었을 때, 해당 부분에서 lock 관련 에러 핸들링 코드로 빠져 나가는 부분을 활용하여 문제를 해결할 수 있었습니다. 그래서 함수 바깥쪽에 locking을 구현하는 방식으로 해결할 수 있었습니다. 지난 과제에서 gdb를 실행할 줄 몰라 애를 먹었었는데, 터미널 두개를 띄워놓고 한쪽에서 gdb 로컬 서버를 열고, 다른 쪽에서 gdb를 실행하여 target remote를 통해 디버깅을 하는 방법을 알아내고, 실제로 디버깅을 gdb를 이용해서 해본게 많이 도움이 된 것 같습니다.

긴 글 읽어주셔서 감사합니다.