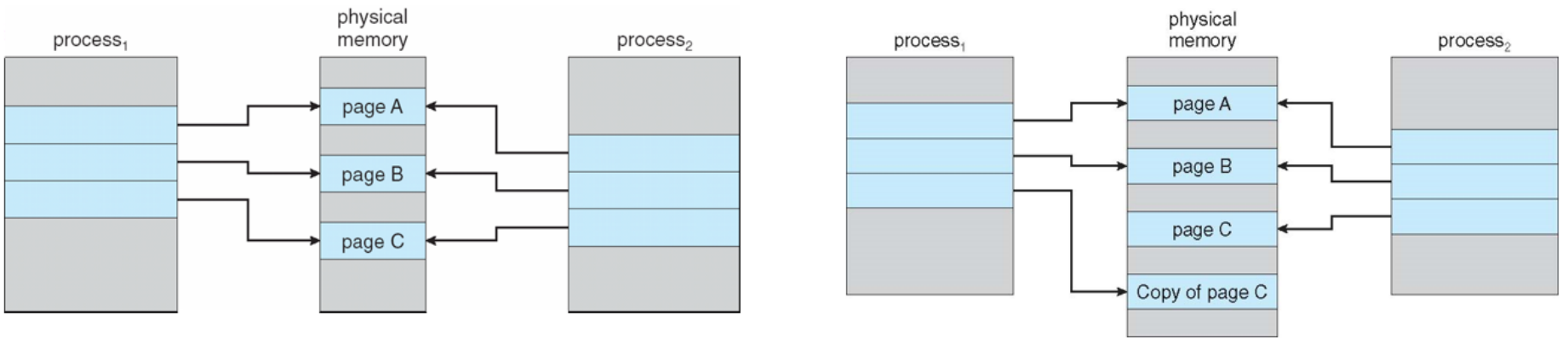
project04 wiki - 2022028522

1. Design

> 📌 명세에서 요구하는 조건에 대해서 어떻게 구현할 계획인지, 어떤 자료구조와 알고리즘이 필요한지, 자신만의 디자인을 서술합니다.



과제를 진행하기에 앞서, copy-on-write의 개념에 대해 정리해보았습니다. 여러 프로세스가 동일한 메모리 페이지를 읽기 전용으로 공유하고 있다가, 어느 한 프로세스가 해당 페이지에 write를 시도할 때 그제서야 페이지를 복사하여 두 프로세스를 분화시킵니다. 이를 통해 `fork()`를 호출했을 때 부모 프로세스와 자식 프로세스의 페이지를 공유하게 함으로써 메모리 낭비를 줄이고, 자식 프로세스가 부모 프로세스 전체를 복사하여 발생하는 시간 낭비를 줄일 수 있습니다.

xv6에서 copy-on-write를 구현하기 위해 해야 할 일들은 아래와 같습니다.

1. **참조 카운팅:**
   - 페이지마다 해당 페이지를 가상 주소 공간에 매핑한 프로세스의 수를 기록해야 합니다.
   - 페이지가 처음 할당될 때 이 값은 1로 설정됩니다.
   - 추가 프로세스가 이미 존재하는 페이지를 가리킬 때, 또는 프로세스가 더 이상 페이지를 가리키지 않을 때 참조 카운팅이 증가하거나 감수하는 함수가 필요합니다.
   - 페이지의 참조 횟수가 0일 때에만 페이지를 free하고 freelist로 반환할 수 있습니다.
2. **fork 시스템 콜 수정:**
   - 부모 프로세스의 메모리 페이지를 자식 프로세스와 공유하도록 수정합니다.
   - 이때, 페이지는 읽기 전용으로 설정되고, 참조 횟수를 증가시킵니다.
3. **page fault handler 작성:**
   - 부모 또는 자식 프로세스가 읽기 전용으로 표시된 페이지에 쓰기를 시도할 때, page fault가 발생하고, 이때 page fault handler인 `CoW_handler`가 호출됩니다.
   - `CoW_handler`는 page fault가 발생한 주소를 확인하고, 해당 페이지가 잘못된 범위에 속해있는지 확인합니다.
   - 잘못된 범위에 속해있다면 에러 메시지 출력과 함께 프로세스를 종료하고, 잘못된 범위에 속해있지 않다면 새로운 페이지를 할당하고 기존 페이지의 내용을 복사합니다.
4. **TLB flush:**
   - initial sharing 단계에서 부모와 자식 프로세스가 페이지를 공유하게 되면, 페이지 테이블을 재설치하고 TLB를 플러시하여 페이지 테이블 변경 사항을 반영합니다.
   - 읽기 전용으로 설정된 페이지에 write를 시도하여 page fault가 발생하면 handler가 페이지를 복사하는데, 이때 페이지 테이블 엔트리를 업데이트하면서 TLB flush로 변경사항을 반영해야 합니다.

2. Implement

📌 실제 구현 과정에서 변경하게 되는 코드영역이나 작성한 자료구조 등에 대한 설명을 구체적으로 서술합니다

kalloc.c

kmem 자료구조 수정

`kmem`은 xv6 운영체제에서 물리 메모리 할당과 해제를 관리하는 자료구조입니다. `kmem` 구조체에 필드를 추가함으로써 `countfp()` 함수와 참조 카운팅 관련 함수들을 구현할 수 있습니다.

```
struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
  uint num_free_pages;                          // 추가된 부분
  uint page_ref_count[PHYSTOP >> PGSHIFT];      // 추가된 부분
} kmem;
```

- `uint num_free_pages` : 현재 시스템에서 사용 가능한 자유 메모리 페이지의 수를 나타냅니다.
- `uint page_ref_count[PHYSTOP >> PGSHIFT]` : 각 물리 페이지의 참조 횟수를 저장하는 배열입니다. 페이지가 다른 프로세스에서 공유될 때 참조 횟수를 증가시키고, 더 이상 사용되지 않을 때 참조 횟수를 감소시킵니다. `PHYSTOP`은 물리 메모리의 끝 주소를 나타내며, `PGSHIFT`는 페이지 크기를 나타내는 비트 시프트 연산입니다. 그러므로 `PHYSTOP >> PGSHIFT`는 사용할 수 있는 전체 물리 메모의 양인 `PHYSTOP`에서 페이지 크기인 `PGSHIFT`를 나눈 값, 즉 시스템이 관리할 수 있는 페이지의 양과 같습니다.

kinit1 함수 변경

`kinit` 함수는 물리 메모리 페이지를 초기화하는 함수입니다.

```
void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  kmem.num_free_pages = 0;       // 추가된 부분
  freerange(vstart, vend);
}
```

- 이 함수에서 `kmem.num_free_pages`를 `0`으로 설정합니다.

freerange 함수 변경

```
void
freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE){
    kmem.page_ref_count[V2P(p) >> PGSHIFT] = 0;        // 추가된 부분
    kfree(p);
  }
}
```

- `V2P(p)`는 가상 주소 `p`를 물리 주소로 변환하고, `V2P(p) >> PGSHIFT`는 물리 주소를 페이지 번호로 변환합니다. 따라서 `kmem.page_ref_count[V2P(p) >> PGSHIFT]`는 현재 페이지의 참조 횟수를 나타내며, 이를 0으로 설정하여 초기화합니다.

kfree 함수 변경

`kfree` 함수는 주어진 물리 페이지를 해제하는 함수이며, 자유 메모리 리스트에 추가하는 역할을 합니다.

```
void
kfree(char *v)
{

...

  if(kmem.page_ref_count[V2P(v) >> PGSHIFT] > 0){
    kmem.page_ref_count[V2P(v) >> PGSHIFT] -= 1;
  }

  if(kmem.page_ref_count[V2P(v) >> PGSHIFT] == 0){
    memset(v, 1, PGSIZE);
    r->next = kmem.freelist;
    kmem.num_free_pages += 1;
    kmem.freelist = r;
  }

...

}
```

- copy-on-write를 구현하기 위해 이 함수에서 페이지의 참조 횟수를 확인하여, 0보다 크면 참조 횟수를 1 감소시킨 뒤, 참조 횟수가 0이 되었을 때만 페이지를 실제로 해제합니다.

kalloc 함수 변경

`kalloc` 함수는 자유 메모리 페이지를 할당하여 반환하는 역할을 하며, copy-on-write 기능을 지원하기 위해 이 함수에서 페이지의 참조 횟수를 관리해주어야 합니다.

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

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;

  if(r){
    kmem.freelist = r->next;
    kmem.page_ref_count[V2P((char*)r) >> PGSHIFT] = 1;
    kmem.num_free_pages -= 1;
  }

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

  return (char*)r;
}
```

- 이 함수에서 변경된 부분은 새로 할당된 페이지의 참조 횟수를 1로 설정하고, 자유 페이지 수인 `kmem.num_free_pages`를 1 감소시킵니다.

incr_refc 함수

```
void
incr_refc(uint pa)
{
  if(pa >= PHYSTOP || pa < (uint)V2P(end))
    panic("incrememt referece count error");

  acquire(&kmem.lock);

  kmem.page_ref_count[pa >> PGSHIFT] += 1;

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

- 우선 인자로 들어온 물리 주소 `pa`가 유효한지 검사합니다. `pa`가 시스템에서 사용할 수 있는 물리 메모리의 상한선인 `PHYSTOP`보다 크거나 커널의 끝 주소인 `V2P(end)`보다 작으면 잘못된 주소로 간주하여 `panic` 함수를 호출하여 오류를 발생시킵니다.
- 그 다음으로 `kmem.lock`을 획득한 뒤, `kmem.page_ref_count` 배열의 해당 페이지 인덱스에 접근하여 참조 횟수를 1 증가시킵니다. 이 때 `pa >> PGSHIFT`를 통해 물리 주소 `pa`를 페이지 번호로 변환할 수 있고, 이 페이지 번호를 통해 배열에 접근할 수 있습니다.
- 마지막으로 `kmem.lock`을 해제합니다.

decr_refc 함수

```
void
decr_refc(uint pa)
{
  if(pa >= PHYSTOP || pa < (uint)V2P(end))
    panic("decrement reference count error");

  acquire(&kmem.lock);

  kmem.page_ref_count[pa >> PGSHIFT] -= 1;

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

- `kmem.page_ref_count[pa >> PGSHIFT]`를 1 감소시키는 점만 빼면 `incr_refc`와 거의 유사합니다.

get_refc 함수

```
int
get_refc(uint pa)
{
  if(pa >= PHYSTOP || pa < (uint)V2P(end))
    panic("get reference count error");

  acquire(&kmem.lock);

  int count = kmem.page_ref_count[pa >> PGSHIFT];

  release(&kmem.lock);

  return count;
}
```

- `kmem.page_ref_count[pa >> PGSHIFT]` 자체를 반환하며, `incr_refc`와 `decr_refc`와 거의 유사합니다.

countfp 함수

```
int
countfp(void)
{
  acquire(&kmem.lock);

  int num_free_pages = kmem.num_free_pages;

  release(&kmem.lock);

  return num_free_pages;
}
```

- 현재 시스템에서 사용 가능한 자유 페이지의 수를 추적하기 위해 `kmem`에 추가해뒀던 필드인 `kmem.num_free_pages`를 `num_free_pages` 변수에 저장한 뒤 이를 반환합니다.

trap.c

```
void
trap(struct trapframe *tf)
{

...

  switch(tf->trapno){
  case T_PGFLT:
    CoW_handler();
    break;

...

}
```

- page fault의 예외 번호 `T_PGFLT`가 발생했을 때 `CoW_handler` 함수를 호출하여 page fault를 처리합니다. `CoW_handler` 함수는 후술하도록 하겠습니다.

vm.c

copyuvm 함수 변경

기존 `copyuvm` 함수는 페이지 테이블 엔트리를 단순히 복사하고, 자식 프로세스가 부모 프로세스의 메모리를 독립적으로 사용할 수 있도록 설정했습니다. 그러나 copy-on-write 기능을 추가하면 자식과 부모가 페이지를 공유할 수 있으며, 처음에는 페이지를 읽기 전용으로 설정하고, 필요할 때 페이지를 복사합니다. 이를 위해 참조 횟수 관리와 페이지 테이블 엔트리의 읽기 전용 설정이 필요합니다. 기존 `copyuvm` 함수와 달라진 점은 아래와 같습니다.

```
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){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");

    *pte = (~PTE_W) & *pte;
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);

    if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0) {
      freevm(d);
      lcr3(V2P(pgdir));

      return 0;
    }
    incr_refc(pa);
  }
  lcr3(V2P(pgdir));

  return d;
}
```

- `*pte = (~PTE_W) & *pte` : 페이지 테이블 엔트리의 쓰기 권한 비트(`PTE_W`)를 제거하여 페이지를 읽기 전용으로 설정합니다.
- `if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0)` : `mappages` 함수를 호출하여 새로운 페이지 디렉토리 `d`에 페이지를 매핑합니다. 매핑에 실패하면 자식 페이지 디렉토리를 해제하고 0을 반환합니다.
- `incr_refc(pa)` : 페이지의 참조 횟수를 증가시킵니다.
- `lcr3(V2P(pgdir))` : 페이지 디렉토리를 업데이트하고 TLB를 flush하여 변경 사항을 반영합니다.

countvp 함수

`countvp` 함수는 현재 프로세스의 사용자 메모리에 할당된 가상 페이지(logical page)의 수를 계산하고 반환하는 함수입니다.

```
int
countvp(void)
{
  return PGROUNDUP(myproc()->sz)/PGSIZE;
}
```

- 이 함수는 현재 프로세스의 가상 주소 공간 크기인 `myproc()->sz`를 기반으로 페이지 수를 계산하는데, `PGROUNDUP(myproc()->sz)/PGSIZE`는 가상 주소 공간 크기를 페이지 크기로 나눈 값으로, 이는 현재 프로세스의 가상 페이지 수를 나타냅니다.

countpp 함수

`countpp` 함수는 현재 프로세스의 페이지 테이블을 탐색하고, 유효한 물리 주소가 할당된 page table entry의 수를 반환하는 함수입니다.

```
int
countpp(void)
{
  pde_t *pgdir = myproc()->pgdir;
  int count = 0;
  uint i;

  for(i = 0; i < myproc()->sz; i += PGSIZE) {
    pte_t *pte = walkpgdir(pgdir, (void *)i, 0);
    if(pte && (*pte & PTE_P)) {
      count++;
```

```
    }
  }
  return count;
}
```

- 우선 `myproc()` 함수를 호출하여 현재 실행 중인 프로세스를 가져오고, 실행 중인 프로세스의 페이지 디렉토리를 `pgdir` 변수에 저장합니다.
- 그 다음으로 for문을 통해 현재 프로세스의 가상 주소 공간 크기인 `myproc()->sz`만큼 페이지 단위인 `PGSIZE`로 순회하는데, `walkpgdir` 함수를 사용하여 현재 페이지 디렉토리인 `pgdir`에서 가상 주소 `i`에 해당하는 페이지 테이블 엔트리를 찾습니다.
- 해당 엔트리가 존재하면 `pte`에 그 포인터를 저장하고, 엔트리가 존재하지 않으면 `NULL`을 반환합니다.
- 페이지 테이블 엔트리가 존재하고(`pte`), 해당 페이지가 실제로 존재하는 경우(`*pte & PTE_P`), `count`를 증가시킵니다.

countptp 함수

`countptp` 함수는 현재 프로세스의 페이지 디렉토리에서 페이지 디렉토리 엔트리(PDE)를 검사하여 유효한 페이지 테이블 페이지(PTP)의 수를 반환하는 역할을 합니다.

```
int
countptp(void)
{
  int count = 0;

  for(int i = 0; i < NPDENTRIES; i++){
    if(myproc()->pgdir[i] & PTE_P){
      count++;
    }
  }
  return count+1;
}
```

- `for` 루프를 통해 페이지 디렉토리 엔트리의 수인 `NPDENTRIES`만큼 순회하며, 해당 엔트리가 유효한 페이지 테이블 페이지를 검사하며 (`if(myproc()->pgdir[i] & PTE_P)`) 유효하면 `count`의 값을 증가시킵니다.
- 이때 페이지 디렉토리 자체도 메모리에서 한 페이지를 차지하므로, 전체 페이지 테이블 페이지 수를 정확하게 계산하기 위해 페이지 디렉토리 자체를 추가로 카운트한 값인 `count+1`을 `return`합니다.

CoW_handler 함수

`CoW_handler` 함수는 copy-on-write를 구현하기 위해 page fault를 처리하는 핸들러로, page fault가 발생했을 때 페이지를 복사하여 쓰기 가능 상태로 변경하는 작업을 수행합니다.

```
void
CoW_handler(void)
{
  uint va = rcr2();
  pte_t *pte = walkpgdir(myproc()->pgdir, (void*)va, 0);

  if(PTE_W & *pte)
    panic("Already writable");

  if(pte == 0 || !(*pte) || va >= KERNBASE || !PTE_U || !PTE_P){
    myproc()->killed = 1;
    cprintf("Illegal virtual addrress, killing process\n");

    return;
  }

  uint pa = PTE_ADDR(*pte);
  uint refc = get_refc(pa);
```

```
if(refc == 1){
  *pte = PTE_W | *pte;

  lcr3(V2P(myproc()->pgdir));
  return;
}
else{
  char* mem = kalloc();

  if(mem != 0){
    memmove(mem, (char*)P2V(pa), PGSIZE);

    *pte = PTE_U | PTE_W | PTE_P | V2P(mem);

    decr_refc(pa);

    lcr3(V2P(myproc()->pgdir));
      return;
    }
    else{
      myproc()->killed = 1;
      cprintf("Out of memory, killing process\n");
      return;
    }
  }
  lcr3(V2P(myproc()->pgdir));
}
```

- `rcr2()` 함수를 통해 page fault가 발생한 가상 주소를 CR2 레지스터에서 읽어옵니다.
- `walkpgdir` 함수를 사용하여 현재 프로세스의 페이지 디렉토리에서 가상 주소 `va`에 대한 페이지 테이블 엔트리를 찾습니다.
- 페이지 테이블 엔트리가 이미 쓰기 가능 상태인지, 페이지 테이블 엔트리가 유효하지 않은지, 가상 주소 `va`가 커널 주소 공간에 속하지는 않은지, 사용자 접근 권한 또는 페이지 존재 비트가 설정되지 않았는지를 확인하여 `panic`을 발생시키거나 프로세스를 종료시킵니다.
- 페이지 테이블 엔트리에서 물리 주소 `pa`를 가져오고, `get_refc` 함수를 사용하여 물리 주소 `pa`에 대한 참조 횟수를 가져옵니다.
- 참조 횟수가 1이면 페이지를 쓰기 가능 상태로 설정하고 TLB를 플러시합니다.
- 참조 횟수가 1보다 크면 새로운 페이지를 할당하고 기존 페이지의 내용을 복사한 뒤 새로운 페이지를 테이블 엔트리에 설정하고, 기존 페이지의 참조 횟수를 감소시킵니다. 여기서도 TLB를 플러시하여 변경 사항을 반영합니다.
- 새로운 페이지 할당에 실패하면 프로세스를 종료합니다.

---

3. Result

📌 컴파일 및 실행 과정과, 해당 명세에서 요구한 부분이 정상적으로 동작하는 실행 결과를 첨부하고, 동작 과정에 대해 설명합니다.

xv6-public 디렉토리 안에서 다음과 같은 명령어를 차례대로 입력합니다.

```
$ make clean
$ make
$ make fs.img
```

```
…
$ ./bootxv6.sh
```

```
$ test0
$ test1
$ test2
$ test3
```

테스트 결과는 아래와 같습니다.

test0

```
$ test0
[Test 0] default
ptp: 66 66
[Test 0] pass
```

4가지 시스템 콜인 countfp, countvp, countpp, countptp가 제대로 작동하며 추가적인 페이지가 알맞게 할당되었음을 확인할 수 있습니다.

test1

```
$ test1
[Test 1] initial sharing
[Test 1] pass
```

자식 프로세스와 부모 프로세스가 같은 물리 페이지를 가리키고 있음을 확인할 수 있습니다.

test2

```
$ test2
[Test 2] Make a Copy
[Test 2] pass
```

자식 프로세스가 부모 프로세스와 공유하고 있는 변수를 수정하였을 때 새로운 물리 페이지를 할당 받았음을 확인할 수 있습니다.

test3

```
$ test3
[Test 3] Make Copies
child [0]'s result: 1
child [1]'s result: 1
child [2]'s result: 1
child [3]'s result: 1
child [4]'s result: 1
child [5]'s result: 1
child [6]'s result: 1
child [7]'s result: 1
child [8]'s result: 1
child [9]'s result: 1
[Test 3] pass
```

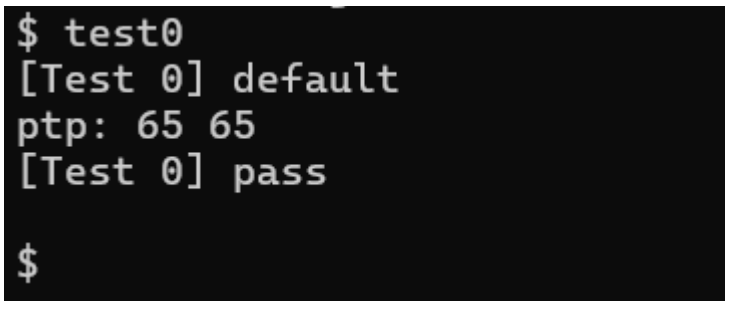
10개의 자식 프로세스가 부모 프로세스와 공유하는 변수를 수정한 뒤 종료되고 나서 free page의 회수가 적절하게 이루어졌음을 확인할 수 있습니다.

---

4. Trouble Shooting

> 📌 과제 수행 과정에서 겪은 문제가 있다면, 해당 문제와 해결 과정을 서술합니다. 해결하지 못했다면 어떤 문제였고 어떻게 해결하려 했는지 서술합니다.

countptp 함수 문제



처음에 과제 명세를 제대로 이해하지 못해 `countptp`의 값이 잘못 나왔으나 이 함수를 적절히 잘 수정하여 과제 명세와 같이 `66 66`이 나오도록 수정하였습니다.