Operating Systems, Project 5

2013-11392 김지현

개요

이번 과제는 xv6에서 프로세스가 fork() 될 때, 유저 페이지들을 즉시 복사하는 대신 공유하게 만들어, xv6의 메모리 사용량을 줄이는 과제이다. 강의시간에 배운 레퍼런스 카운팅과 Copy-on-Write(CoW) 테크닉을 적용시켜 구현할 수 있었다. 여러 코너케이스들을 빠짐없이 처리하고, 페이지들이 존재할 수 있는 여러 상태들을 잘 고려해 특정 페이지가 invalid state에 빠지거나, 스스로 정한 invariant를 깨지 않도록 조심하면 과제를 성공적으로 마칠 수 있었다.

특이사항으로, 채점 사이트 테스트케이스에 문제가 있어, 풀은 문제를 풀지 못했다고 오래 착각해, 과제 완료에 시간이 다소 오래걸렸다.

구현 설명

과제를 아래의 두 단계로 나누어 풀었다.

1. Shared read-only page 구현
2. Shared CoW page 구현

1번과 2번을 독립적으로 구현할 수 있었다.

Shared read-only page 구현

아래와 같은 전략으로 구현하였다.

i. 모든 유저용 memory page를 “unmanaged” page와 “shared” page로 나눔. 모든 “shared” page들은 각각 자신을 참조하는 프로세스의 수를 나타내는 reference count(이하 RC)를 갖는다.

ii. “exec()” 함수에서 ELF program header를 파싱할 때, ELF program header에 들어있는 권한 flag도 함께 파싱하여, uvmalloc() 함수를 호출할 때 파싱한 권한 비트를 PTE에 기록하도록 수정. 또한 새로 만들어지는 페이지가 Read-only일 경우, 그 페이지를 “shared” page로 지정하고, RC를 1로 설정하는 구현도 함께 들어가야 한다.

iii. “fork()” 함수에서 uvmalloc() 함수를 호출할 때, “shared” page들은 복사하는 대신 공유하도록 설정하고, RC 1 증가

iv. 프로세스가 꺼질 때, 프로세스가 갖고있던 페이지 테이블을 정리하는 과정에서 uvmunmap() 함수를 부르게 되는데, 이 때 “shared” page들은 RC를 1 감소시키며, RC가 0이 될 경우 페이지 할당 해제

v. (optional) 코드 영역을 제외한 유저 페이지들은 기본적으로 executable 권한을 갖지 못하도록 막음

이 때, 각 “shared” page들이 reference count를 어떻게 갖게 할것이냐에 design choice가 들어간다. 안타깝게도 kalloc() 동작원리 상, page 근처에 metadata를 저장할만한 공간은 딱히 없다. page들에 metadata를 부여하려면, page의 address나 id를 키로 갖고 그 페이지의 메타데이터를 value로 갖는 dictionary 자료구조를 하나 만들어야한다.

어떤 자료구조로 dictionary를 구현할것인가 고민하면, 아래와 같은 선택지들이 있다.

- BTree, Red-black tree, HashMap
- ArrayMap (Use sparse array as a map, using array index as a key)
- Linked list
- Associated array
- Associated array sorted by key
- …

당연히 BTree나 Red-black tree, HashMap이 시간복잡도나 메모리 오버헤드를 종합하여 판단했을 때 제일 좋은 선택이지만, 저러한 자료구조들은 xv6 안에 구현하기 여의치 않아 고려하지 않았다. Page의 id를 길이로 갖는 거대한 배열(ArrayMap)은 구현하기엔 가장 단순하겠지만, 메모리 오버헤드가 너무 커서 선택하지 않았다. Linked list를 쓰는것은 성능이 안좋은 반면 Associated array보다 구현이 딱히 더 단순한것도 아니여서 선택하지 않았다.

결과적으로 메모리 오버헤드와 성능 오버헤드가 많지 않으면서도, 구현이 적절히 단순한 Associated array를 채택하였다. Associated array는 key와 value의 tuple을 동적배열에 저장하는 형태의 자료구조이다. Associated array는 항상 key로 정렬시켜, lookup 연산 수행시 이진탐색 알고리즘으로 시간복잡도 이득을 볼 수 있도록 구현하였다. Kalloc()을 잘 사용하면 길이제한이 없는 associated array도 만들 수 있지만, 과제에서 요구하는 범위를 넘어서는 것 같아 그렇게까지 구현하지는 않았다.

테스트 케이스 디버깅

일반적인 상황에서는 여기까지 구현하면 c_share 항목에서 점수를 받을 수 있는데, 안타깝게도 테스트케이스에 문제가 있어 점수를 받지 못했다. 처음에는 테스트케이스에 문제가 있다는 것을 알지 못했고, 스스로의 코드를 디버깅하기 위해 아래와 같은 방법들을 사용했다.

1. 다양한 테스트케이스 제작
2. 모든 프로세스들의 page table을 모조리 순회하며 dump하는 함수 개발

먼저 fork() 함수와 다른 여러 시스템콜을 호출하는 다양한 테스트케이스를 스스로 제작하였고, 만들었던 모든 예제들이 커널패닉이나 메모리릭 없이 정상적으로 종료되었다. 여기서 좀 더 가시성을 확보하기 위해 procdump() 함수 내에 Figure 1와 같은 코드를 넣어, ctrl+p 키를 누르면 모든 프로세스들의 모든 PTE 정보와 플래그, 각 페이즈들의 레퍼런스 카운트 수가 출력되도록 구현하였다.

```
// TODO: Remove me
// 이 프로세스의 virtual address space 생김새 출력
printf(
  "\n"
  "virtual address    | UXWRV | physical address   | info                | RC\n"
  "-------------------|-------|--------------------|---------------------|---\n"
);
for (int i = 511; i >= 0; --i) {
  // Level 2
  pte_t pte = p->pagetable[i];
  if (pte == 0) { continue; }
  pagetable_t pagetable = (pagetable_t)PTE2PA(pte);

  for (int j = 511; j >= 0; --j) {
    // Level 1
    pte_t pte = pagetable[j];
    if (pte == 0) { continue; }
    pagetable_t pagetable = (pagetable_t)PTE2PA(pte);

    for (int k = 511; k >= 0; --k) {
      // Level 0
      pte_t pte = pagetable[k];
      if (pte == 0) { continue; }
      const void *physical_addr = (const void *)PTE2PA(pte);
      uint64 virtual_addr = ((uint64)i<<30) | ((uint64)j << 21) | ((uint64)k << 12);

      printf("%p | %d%d%d%d%d | %p | %s | ",
        virtual_addr,
        !!(pte & PTE_U), !!(pte & PTE_X), !!(pte & PTE_W), !!(pte & PTE_R), !!(pte & PTE_V),
        physical_addr,
        virtual_addr == (uint64)TRAMPOLINE                        ? "trampoline          " :
        virtual_addr == (uint64)TRAPFRAME                         ? "trapframe           " :
        (pte & (PTE_U | PTE_X | PTE_W | PTE_R | PTE_V)) == 0b10111  ? "stack+heap+data     " :
        (pte & (PTE_U | PTE_X | PTE_W | PTE_R | PTE_V)) == 0b00111  ? "stackguard          " :
        (pte & (PTE_U | PTE_X | PTE_W | PTE_R | PTE_V)) == 0b10011  ? "stack+heap+data (cow)" :
        (pte & (PTE_U | PTE_X | PTE_W | PTE_R | PTE_V)) == 0b00011  ? "stackguard      (cow)" :
        (pte & (PTE_U | PTE_X | PTE_W | PTE_R | PTE_V)) == 0b11011  ? "code+rodata         " :
        "                "
      );
      uint8 rc = __meta_rc(physical_addr);
      if (rc) {
        printf("%d\n", rc);
      } else {
        printf("\n");
      }
    }
  }
}
printf("\n");
// TODO: Remove me
```

Figure 1. Enhanced procdump()

이를 실행하여 Figure 2와 같이, 각 PID의 모든 PTE 정보를 손쉽게 출력할 수 있었고, 스스로 만들었던 모든 테스트케이스들에서, 정상적으로 code 영역이 share 되고있음을 확인할 수 있었다.

```
9 sleep  forktest2

virtual address    | UXWRV | physical address   | info            | Shared
-------------------|-------|--------------------|-----------------|--------
0x0000003ffffff000 | 01011 | 0x0000000080007000 | trampoline      |
0x0000003fffffe000 | 00111 | 0x0000000087f2a000 | trapframe       |
0x0000000000003000 | 11111 | 0x0000000087f22000 | stack, heap     |
0x0000000000002000 | 01111 | 0x0000000087f23000 | stack guard     |
0x0000000000001000 | 10111 | 0x0000000087f24000 | bss, sbss       |
0x0000000000000000 | 11011 | 0x0000000087f44000 | code, rodata    | Y

10 sleep  forktest2

virtual address    | UXWRV | physical address   | info            | Shared
-------------------|-------|--------------------|-----------------|--------
0x0000003ffffff000 | 01011 | 0x0000000080007000 | trampoline      |
0x0000003fffffe000 | 00111 | 0x0000000087f21000 | trapframe       |
0x0000000000003000 | 11111 | 0x0000000087f19000 | stack, heap     |
0x0000000000002000 | 01111 | 0x0000000087f1a000 | stack guard     |
0x0000000000001000 | 10111 | 0x0000000087f1b000 | bss, sbss       |
0x0000000000000000 | 11011 | 0x0000000087f44000 | code, rodata    | Y
```

Figure 2. Dump of PTEs of each processes

과제 마감 기한이 지난 뒤에도 끊임없이 위와 같이 디버깅을 하였으나 문제를 찾을 수 없었고, 2일째 되는 날 새벽에 테스트케이스가 틀렸다는 가정으로 선회하였다.

먼저 많은 학생들이 테스트케이스에 문제를 제기하지 않았기 때문에, 특정한 구현 아래에선 분명히 c_share가 점수가 나올것이라고 가정하였다. c_share가 점수가 나오는 상황을 찾기위해 일부러 레퍼런스 카운터도 제거하고, 복잡한 associated array 구현도 제거해 ArrayMap으로 대체한 뒤, 원본 뼈대코드와의 diff가 최소화된 가장 간단한 구현(A)을 만들었다.

그리고 단순한 구현 위에서 Fuzzing하듯이 코드를 랜덤한 방법으로 계속 수정해가며 계속 제출하다가, 마침내 점수가 나오는 조건을 우연히 찾을 수 있었다. 코드, 데이터 가리지 않고 모든 유저 페이지들을 무조건 share 하도록 구현(B)하는 것 이었다.

코드 데이터를 가리지 않고 무조건 페이지를 share하면 당연히 틀린 구현이다. 그러나 점수가 나오는 구현을 찾았기 때문에, 두 구현체 A와 B를 이진탐색하며 어떤 차이가 점수차이를 만들어내는지 실험하였다. 그 결과, 점수차를 형성하는 가장 작은 diff는 Figure 3과 같았다.

```
project-5/kernel/vm.c
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
343
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);

    if((flags & PTE_U) && (flags & PTE_X)){
    if((flags & PTE_U) && (flags & PTE_X) && (!(flags & PTE_W))){
      // The page is read-only, share the page
      mem = (char*)pa;
    } else {
```

Figure 3. Writable한 페이지를 share하지 않으려고 결정하는 순간, c_share가 0점이 된다.

Writable한 페이지를 share하지 않으려고 결정하는 순간 c_share가 0점이 된다는 것을 확인할 수 있었고, 이는 테스트케이스 입력으로 주어진 ELF 바이너리들의 코드 영역이 read-only가 아니기 때문으로 추정되었다. 이 이후부터는 더 이상 디버깅이 불가능하기에, 알아낸 사실을 조교님과 교수님께 알렸고, 교수님께서 테스트케이스에 오류가 있다는 사실을 확인해주실 수 있었다.

당장은 Executable한 section일 경우, read-only가 아니더라도 강제로 페이지를 share하도록 구현하여 이 문제를 workaround할 수 있었다.

Shared CoW page 구현

기존의 구현을 아래와 같이 수정하였다.

i. “unmanaged” type 삭제, “CoW” type 추가.

```
165 enum uvm_type { UVM_SHARED = 1, UVM_COW = 2 };
166 uint64          uvmalloc(pagetable_t, uint64, uint64, int, enum uvm_type);
```

Figure 4. enum uvm_type

이제 모든 유저 페이지들은 “shared” type이거나 “CoW” type이어야 한다. 다른 말로 하면, 모든 유저 페이지들은 레퍼런스 카운터로 항상 관리되어야한다는 뜻이다.

ii. Read-only가 아닌 모든 페이지들 CoW로 설정

Read-only 코드 영역이 아닌 유저 page 를 만드는 경우는 총 아래와 같고, 모두 “CoW” 타입으로 페이지를 관리하도록 설정해야한다. 새로 만들어지는 CoW page의 RC는 모두 1이어야 한다.

a. “exec()” 함수의 data section 페이지 (.bss .sbss)
b. “exec()” 함수의 스택 가드 페이지, 스택 페이지
c. “growproc()” 함수의 heap 페이지
d. “uvminit()” 함수의 init 프로세스용 페이지

iii. 무조건 레퍼런스 카운트 수행하기

기존 uvmalloc(), uvmcopy(), uvmunmap() 함수에선 이 페이지의 타입이 “unmanaged”인지 “shared”인지를 검사하고 “shared”일 때에만 레퍼런스 카운터를 업데이트하도록 구현되었었으나, 이제는 모든 유저 페이지들은 항상 레퍼런스 카운터로 관리되어야하므로, 검사 없이 바로 레퍼런스 카운터를 조작하는 구현으로 변경한다.

iv. 레퍼런스 카운트 늘릴 때 write flag 지우기

```
struct uvm_meta *meta = meta_of((void*)pa);
meta_incr(meta);
// Set as unwritable if CoW borrow occured
if (meta->type == UVM_COW) {
  *pte &= ~PTE_W;
  flags &= ~PTE_W;
}
```

Figure 5. Clearing write flags

레퍼런스 카운터가 2 이상인 “CoW” 페이지를 가리키는 모든 PTE에는, write flag가 없어야한다. CoW 페이지에 Write를 시도할 때 이를 trap으로 잡기 위해서이다. 이를 위해 CoW 페이지의 레퍼런스 카운터를 증가시킬 때마다 해당 페이지를 가리키는 PTE들을

찾아 write flag를 없애줘야 하고, uvmcopy() 함수가 레퍼런스 카운터를 증가시키는 유일한 장소이다.

레퍼런스 카운터가 1인 “CoW’ 페이지는 writable 해도 되고 writable 하지 않아도 된다. 아래의 Trap에서 레퍼런스 카운터가 1인 페이지는 해당 페이지를 writable로 세팅하고 곧바로 속행하도록 구현할 것이기 때문이다.

v. 유저가 CoW 페이지에 쓰려고 시도하는 경우 처리하기

```
  syscall();
} else if (scause == 15) {
  uint64 va = r_stval();
  if(handle_cow_write(va, p->pagetable)) {
    printf("usertrap(): invalid address access pid=%d\n", p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }
} else if((which_dev = devintr()) != 0){
```

Figure 6. Handling store page fault

Usertrap() 함수에서 scause 레지스터가 15인 경우, 쓰기 명령을 수행하다가 권한이 부족해 실패했다는 뜻이다. 유저가 사용하려 시도한 주소가 유효한 주소인지 검사하고, 권한이 없다면 그대로 프로세스를 종료시키면 되지만, 해당 페이지가 CoW 타입일 경우엔 의도적으로 write flag 를 제거하여 실패한것이므로 프로세스를 종료시키면 안된다.

CoW 페이지의 레퍼런스 카운터가 1이었을 경우 share하는중이 아니니 write flag를 다시 붙여준 뒤 trap을 마치면 된다. CoW 페이지의 레퍼런스 카운터가 2 이상일 경우, 쓰기를 시도한 페이지를 복사하여 새 페이지를 만든 뒤, PTE가 새 페이지를 가리키도록 설정해야한다. 이 때 새로 만든 페이지 또한 “CoW” 타입으로 관리해야하고, RC는 1로 설정하며 write flag도 활성화시켜야 한다.

vi. 커널이 CoW 페이지에 쓰려고 시도하는 경우 처리하기

일반적으로 커널이 유저 페이지에 데이터를 함부로 쓰는 일은 흔치 않지만, copyout() 함수에선 해당 동작이 발생한다. “usertrap()” 함수를 고친것과 동일한 방식으로, 유효하지 않은 주소에 쓰려고 시도한경우 panic 하되, CoW 페이지에 쓰려고 시도한 경우 레퍼런스 카운트가 1인지 2인지에 따라 알맞은 동작을 수행하면 된다.

여기서 유의해야하는 invariant는 아래와 같다.

1. 모든 페이지는 “shared” 타입이거나, “CoW” 타입이어야 한다.
2. 모든 페이지의 레퍼런스 카운터는 1 이상이어야 한다. (0이 될경우 해제되어야 한다)
3. 레러런스 카운터가 2 이상인 CoW 페이지를 가리키는 모든 PTE에는 write flag가 없어야 한다.

위 invariant를 깨지 않도록 코너 케이스를 잘 처리해줘야만 한다.

No Memory Leak

여기까지 구현을 마치면 결과적으로 모든 유저 페이지에는 레퍼런스 카운터가 달리게 되고, 레퍼런스 카운터가 메모리를 관리해주기 때문에 릭은 자연스럽게 발생하지 않는다.

"fork()" 함수를 통해 같은 페이지를 share 하는 프로세스가 늘 때마다 레퍼런스 카운터가 1씩 증가하고, 프로세스가 꺼지거나, CoW로 인해 페이지가 복사될 때마다 레퍼런스 카운터가 1씩 준다. CoW로 인해 페이지가 복사될 때엔 레퍼런스 카운터가 절대 0이 되지 않지만, 프로세스가 꺼질때엔 페이지들의 레퍼런스 카운터가 0이 될 수 있다. 좀비 프로세스가 정리되면서 uvunmap() 함수가 호출될 때, 각 페이지들의 레퍼런스 카운터를 1씩 감소시키고, 레퍼런스 카운터가 0이 되었는지 검사해 0이 된 경우 해당 페이지와 페이지 메타데이터를 삭제해줘야 한다. 이 과정이 Figure 7과 같다.

```
--*p_counter;
if (*p_counter == 0) {
  // RC reached zero, delete metadata from array
  memmove(
    &meta_list[res.index],
    &meta_list[res.index + 1],
    sizeof(struct uvm_meta)*(meta_list_length - res.index - 1)
  );
  --meta_list_length;
  // Free the page
  kfree(pa);
}
```

Figure 7. Decrementing reference counter

Figure 8. Result

실험 결과

Figure 8과 같이, 채점 사이트에서도 만점이 나왔고, xv6에서 이런 저런 프로세스들을 실행해보았을 때에도 정상적으로 작업이 마무리되는 것을 확인할 수 있었다. 다만 위에서 언급했듯이 프로세스를 많이 생성할경우 메타데이터를 저장하는 자료구조에 개수제한이 있어 실패하는 경우가 있었다. 이부분은 과제의 범위를 넘는것 같아 따로 조치하진 않았다. 실제로는 kalloc() 을 사용해 동적으로 배열의 길이가 늘도록 만들어주면 된다.

결론

각 메모리 페이지들이 원하는 invariant를 항상 만족하는 유한상태기계처럼 행동하게하여, xv6 위에서 페이지 share와 CoW를 구현할 수 있었다. 이 과제에선 싱글코어 CPU를 가정했어서 구현이 비교적 단순하였는데, 실제 상황처럼 멀티코어 CPU 를 지원해야하고, 개수제한도 없어야 한다면 구현이 훨씬 복잡했을 것 같다. 운영체제 프로그래밍의 어려움과 재미를 직접 익혀볼 수 있었던 유익한 과제였다.