OS_project04_13065_주원웅_2019060546

Design

1. Freepages 변수와 pagecount 배열 사용

Cow를 구현하기 위해 현재 남아있는 freepage의 개수를 나타내는 freepages 변수를 사용할 것이며, pagecount 배열을 사용하여 페이지 주소 값을 통해 각 페이지의 참조 횟수를 부여할 것 입니다. 배열의 인덱스는 페이지 주소 중 하위 12비트를 제거하여 인덱스로 사용할 것 입니다. Freepages 변수의 초기값은 0이고 pagecount 배열의 모든 인덱스 역시 0으로 초기화 할 것입니다.

2. Kfree 함수 수정

Freepages 변수와 pagecount 배열요소의 값을 kfree 함수에서 수정할 것입니다. Kfree 함수는 원래 매개변수로 받아온 커널영역 주소의 값들을 1로 초기화하고 freepage 들의 리스트인 freelist 에 추가하는 작업을 합니다. 하지만 이를 변경하여 받아온 커널영역 주소를 물리페이지로 변환 후 물리 페이지가 참조중인 경우 참조를 해제하기 위해 pagecount 배열요소 값을 Decrease하여 참조 횟수를 줄여줄 것이고, 값을 줄인 후 참조 횟수가 0이 된 경우 해당 물리페이지를 freelist에 추가하고 freepage 한 페이지 늘어났으므로 freepages 변수 값을 increase 하도록 수정할 것 입니다.

3. Kalloc 함수 수정

Kalloc 함수는 메모리를 할당하기 위해 freelist 에서 freepage 를 하나 받아와 return 하는 역할을 합니다. 하지만 이를 수정하여 freepage를 받아오는 과정에서 메모리를 할당하였으므로 freepage가 하나 줄어들고, 해당 페이지의 참조가 이루어지므로 pagecount 배열요소의 값을 increase 할 것 입니다. 이때 물리페이지의 참조가 이루어지는 것 이므로 freelist에 있는 커널(가상)영역을 물리페이지로 전환 후 사용할 것 입니다.

4. Copyuvm 함수 수정

프로세스를 fork() 할 시 자식 프로세스의 메모리 할당과 복사는 copyuvm 함수에서 이루어집니다. 하지만 Copy on Write 를 구현하기 위해 이 복사하는 과정을 수정할 것 입니다. 원래는 페이지 디렉토리와 페이지 테이블을 새로 만든 뒤 부모 프로세스 사이즈만큼 메모리를 모두 새로 할당하여 부모의 모든 페이지를 자식에게 복사합니다. 하지만 페이지 디렉토리와 페이지 테이블만 새로 할당하고 부모의 페이지를 모두 복사하지 않고 부모 페이지 테이블이 가리키는 모든 물리 페이지 영역을 동일하게 가리키도록 mapping할 것 입니다. 즉 복사는 하지 않고 동일한 곳을 가리키도록 만드는 것 입니다. 이후 페이지 테이블이 가리키는 페이지의 상태를(write상태 등) 나타내는 페이지 테이블의 하위 12비트를 수정하여 write 작업을 수행하지 못하도록 수정할 것 입니다. (공유하는 영역 수정 불

가)

5. CoW_Handler 함수 추가

현재 xv6에서는 페이지 폴트를 처리하지 않기 때문에 write가 불가한 상황에서 write 시도 시 나타나는 페이지 폴트를 처리하지 못합니다. 그렇기에 해당 처리를 할 함수 CoW_handler를 작성할 것입니다. 우선 공유 영역에 대한 write 작업이 발생하였으므로 페이지 테이블의 페이지 참조를 해제한 후 물리 페이지를 새로 할당하여 해당 페이지 복사 후 페이지 테이블이 새롭게 할당한 페이지를 가리키도록 할 것 입니다.

하지만 해당 페이지의 참조 횟수가 현재 1일 시 페이지 공유가 일어났지만 나머지 프로세스는 모두 새롭게 페이지를 할당하여 마지막 프로세스만 해당 페이지를 가리키는 상황이므로 새롭게 메모리를 할당하지 않고 write 플래그만 활성 하는 작업을 할 것 입니다. 마지막엔 tlb 플러시를 하여 모든 페이지에 대한 참조를 update할 것 입니다.

Implement

```
18 uint pagecount[maxpages] = {0};
19 uint freepages = 0;
```

우선 pagecount와 freepages를 kalloc.c 에 추가해주었습니다.

```
131 int
132 get_refc(uint pa)
133 {
134   if (pa >= PHYSTOP || pa < (uint)V2P(end))
135     panic("get_refc");
136
137   if (kmem.use_lock)
138     acquire(&kmem.lock);
139
140   int counts = pagecount[pa >> PTXSHIFT];
141
142   if (kmem.use_lock)
143     release(&kmem.lock);
144
145   return counts;
146 }
147
113 void
114 incr_refc(uint pa)
115 {
116   if (pa >= PHYSTOP || pa < (uint)V2P(end))
117     panic("incr_refc");
118
119   if(kmem.use_lock)
120     acquire(&kmem.lock);
121
122   pagecount[pa >> PTXSHIFT]++;
123
124   if(kmem.use_lock)
125     release(&kmem.lock);
126 }
127
128 void
129 decr_refc(uint pa)
130 {
131   if (pa >= PHYSTOP || pa < (uint)V2P(end))
132     panic("decr_refc");
133
134   pagecount[pa >> PTXSHIFT]--;
135 }
```

다음은 참조 횟수를 늘려주고 줄여주는 incr_ refc, dcre_ refc 함수와 해당 페이지의 참조횟수를 가

져올 get_refc 입니다. decr 함수는 kfree 함수 안에서 쓸 것이기에 따로 lock 설정을 하지 않았습니다. 이미 kfree 함수에서 lock을 체크 후 사용하는 상태에서 해당 함수 호출이 일어나기 때문입니다. Incr_refc 함수 같은 경우는 fork 시 일어나는 페이지 공유현상을 구현할 때 사용할 예정이기에 lock을 처리했습니다. Get_refc 함수 역시 Lock 처리를 해주었습니다.

```
64 void
65 kfree(char *v)
66 {
67   struct run *r;
68
69   if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
70     panic("kfree");
71
72   if(kmem.use_lock)
73     acquire(&kmem.lock);
74
75   // Fill with junk to catch dangling refs.
76   if(pagecount[V2P(v) >> PTXSHIFT] > 0)
77     decr_refc(V2P(v));
78
79   // increase free page number
80   if(pagecount[V2P(v) >> PTXSHIFT] == 0){
81     r = (struct run*)v;
82     r->next = kmem.freelist;
83     kmem.freelist = r;
84     freepages++;
85   }
86
87   if(kmem.use_lock)
88     release(&kmem.lock);
89 }
```

위는 kfree 함수입니다. 앞에서 말한 것 처럼 우선 페이지의 참조 횟수를 줄여주었고 이후 참조 횟수가 0인 경우 해당 페이지를 회수하는 작업과 동시에 freepages 변수 값을 증가시켜주었습니다. 참조횟수를 확인할때나 Decr함수에 매개변수로 전달할 때 매개변수로 받아온 v의 주소는 커널(가상)영역이기에 해당 영역은 건들지 않고 물리 페이지로 변환 해주는 매크로를 사용하여 물리 페이지 영역을 수정하였습니다.

```
 94 char*
 95 kalloc(void)
 96 {
 97   struct run *r;
 98
 99   if(kmem.use_lock)
100     acquire(&kmem.lock);
101   r = kmem.freelist;
102   if(r){
103     kmem.freelist = r->next;
104     // decrease free page number and increase page reference number
105     freepages--;
106     pagecount[V2P(r) >> PTXSHIFT] = 1;
107   }
108   if(kmem.use_lock)
109     release(&kmem.lock);
110   return (char*)r;
111 }
```

위 사진은 kalloc 함수입니다. Kfree 와 동일하게 r의 주소를 물리페이지로 변환 후 참조 횟수를 1로 set 하였고 해당 페이지는 할당되어 사용될 예정이므로 freepages 변수값을 감소하였습니다. 참조 횟수를 incr_refc 함수로 사용하지 않은 이유는 incr 함수에서 lock을 사용하므로 이중 락 현상이 발생하여 직접 접근해서 참조를 수정하였습니다.

```
315 pde_t*
316 copyuvm(pde_t *pgdir, uint sz)
317 {
318   pde_t *d;
319   pte_t *pte;
320   uint pa, i, flags;
321
322   if((d = setupkvm()) == 0)
323     return 0;
324   for(i = 0; i < sz; i += PGSIZE){
325     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
326       panic("copyuvm: pte should exist");
327     if(!(*pte & PTE_P))
328       panic("copyuvm: page not present");
329     pa = PTE_ADDR(*pte);
330     flags = (PTE_FLAGS(*pte) & ~PTE_W);
331
332     if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0) {
333       goto bad;
334     }
335     incr_refc(pa);
336     *pte &= ~PTE_W;
337   }
338   lcr3(V2P(pgdir));
339   return d;
340
341 bad:
342   freevm(d);
343   lcr3(V2P(pgdir));
344   return 0;
345 }
```

위 사진은 fork() 호출시 자식 프로세스에 메모리를 할당하는 copyuvm 함수 구현입니다. Kalloc을 통해 메모리를 할당받아오는 부분을 삭제하였고 매개변수로 받아온 2단계 테이블로 구성되어 있는 부모 프로세스의 페이지 다이렉트(1단계)가 가리키는 페이지(2단계, 페이지 테이블)가 활성 상태인 지 walkpgdir 함수를 통해 주소값을 받아오며 체크하였습니다.

해당 페이지 테이블의 상위 20비트(페이지 테이블이 가리키는 피지컬 메모리 페이지의 시작 주소)를 가져오는 매크로를 사용해 pa(페이지 테이블의 주소)에 할당하였습니다. 그 pa를 자식 프로세스의 페이지 다이렉트 d에 매핑함수를 통해 매핑합니다. (페이지 다이렉트에 페이지 테이블 주소 할당) 이제 해당 페이지는 공유 페이지가 되었으므로 해당 페이지 테이블의 상태를 나타내는 나머지 하위 12비트의 플래그 중 write flag를 비활성화 하여 flag를 업데이트 합니다. (12비트는 가상주소의 12비트를 사용하여 실제 물리메모리를 계산할 수 있기에 페이지 테이블의 하위 12비트는 페이지 테이블에 매핑된 페이지의 상태를 나타냄)

이후 lcr3함수를 통해 변경된 pgdir에 대해서 TLB를 flush합니다.

```
388 void CoW_handler(void)
389 {
390     uint va = rcr2(); // Faulting virtual address
391 
392     pte_t *pte;
393     uint pa;
394     char *mem;
395     struct proc *curproc = myproc();
396     int ref;
397 
398     pte = walkpgdir(curproc->pgdir, (void*)va, 0);
399 
400     if(pte == 0 || !(*pte & PTE_P))
401         panic("CoW_handler: page not present or not mapping");
402 
403     pa = PTE_ADDR(*pte);
404     ref = get_refc(pa);
405 
406     if(ref > 1) {
407       decr_refc(pa);
408 
409       if((mem = kalloc()) == 0)
410         panic("CoW_handler: out of memory");
411 
412       memmove(mem, (char*)P2V(pa), PGSIZE);
413 
414       *pte = V2P(mem) | PTE_P | PTE_W | PTE_U;
415     } else if (ref == 1) {
416       *pte = *pte | PTE_W;
417     } else {
418       panic("pagefault reference count wrong\n");
419     }
420     lcr3(V2P(curproc->pgdir));
421 }
```

위 사진은 페이지 폴트 트랩이 발생했을 때 호출할 함수 CoW_handler 구현입니다. 페이지 폴트가 발생한 위치를 rcr2() 함수로 받아온 뒤 해당 위치에 매핑된 페이지 테이블의 값을 가져옵니다. 유효한 값을 체크한 후 위에 있던 과정처럼 20비트로 물리페이지의 주소를 가져옵니다. 그 후 해당 물리 페이지의 참조횟수를 확인하고 1보다 큰 값을 가진 경우 새롭게 메모리를 할당하여 분리 후 해당 페이지의 복사를 진행합니다(페이지 분리). 이후 해당 페이지 테이블의 값을 새롭게 할당한 메모리의 주소로 변경하여 페이지 테이블의 값을 변경합니다.

만약 참조횟수가 1일 경우 해당 프로세스 외 페이지를 공유중인 프로세스가 없는 것이기에 write 비트만 활성으로 변경합니다.

두 작업 이후 lcr3 함수를 호출하여 해당 프로세스의 TLB를 flush하는 작업을 수행합니다.

```
423 int
424 countpp(void)
425 {
426   struct proc* curproc = myproc();
427   pte_t *pte;
428   uint va;
429   int count = 0;
430
431   for(va = 0; va < curproc->sz; va += PGSIZE) {
432     if((pte = walkpgdir(curproc->pgdir, (void*)va, 0)) == 0)
433       panic("copyuvm: pte should exist");
434
435     if (*pte & PTE_P)
436       count++;
437   }
438   return count;
439 }
```

다음은 프로젝트의 countpp 함수입니다. 현재 프로세스의 페이지 테이블을 탐색하고 유효한 물리 주소가 할당된 page table entry (물리 주소에 존재하는 페이지들) 의 수를 세는 시스템 콜 함수입니다. 우선 for문을 통해 프로세스의 총 크기인 sz 만큼 PGSIZE인 4096을 계속 더하여서 페이지의 시작주소를 계산한 뒤, 그 주소가 유효한 엔트리를 가졌는 지 확인하는 함수 walkpgdir을 통해 가져옵니다. 그리고 현재 그 엔트리의 flag 값 중 현재 물리 페이지의 존재하는 지 확인하는 flag PTE_P를 통해 존재한다면 count 해주는 방식으로 구현하였습니다.

```
441 int
442 countptp(void)
443 {
444   struct proc *curproc = myproc();
445   pde_t *pde = curproc->pgdir;
446   int i;
447   int count = 0;
448   for(i = 0; i < NPDENTRIES; i++) {
449     if (pde[i] & PTE_P) {
450       count++;
451     }
452   }
453
454   return count + 1;
455 }
```

다음은 countptp 함수입니다. 이 함수는 현재 프로세스의 페이지 테이블을 표현하기 위해 사용된 테이블(pgdir)의 활성화 된 페이지들을 세는 시스템 콜 함수입니다. 그리고 마지막으로 pgdir 자체의 페이지 역시 추가 해주기 위해 return 에서 +1을 해주었습니다. NPDENTIRES는 페이지 테이블의 최대 개수 1024 입니다. Pgdir을 순회하며 현재 활성화 되어있는지 PTE_P 플래그로 확인하고 count 합니다.

```
457 int
458 countvp(void)
459 {
460   struct proc* curproc = myproc();
461   int count;
462   uint va;
463
464   count = 0;
465   for(va = 0; va < curproc->sz; va += PGSIZE)
466   {
467     count++;
468   }
469   return count;
470 }
471
```

다음은 현재 프로세스의 가상메모리의 페이지 수를 세는 시스템 콜 함수입니다. For문을 통해 sz 만큼 4096 페이지 사이즈만큼 건너뛰며 페이지의 수를 count 합니다.

```
148 int
149 countfp(void)
150 {
151   int fp;
152
153   if (kmem.use_lock)
154     acquire(&kmem.lock);
155
156   fp = freepages;
157
158   if (kmem.use_lock)
159     release(&kmem.lock);
160
161   return fp;
162 }
```

다음은 현재 시스템에서 존재하는 freepages 수를 반환하는 시스템 콜 함수입니다. Lock을 사용하여 중간에 값이 수정될 일이 없도록 하였고 현재 freepage의 총 개수인 freepages 변수를 return 합니다.

```
472 int
473 sys_countptp(void)
474 {
475   return countptp();
476 }
```

```
478 int
479 sys_countpp(void)
480 {
481   return countpp();
482 }
```

```
483 int
484 sys_countvp(void)
485 {
486   return countvp();
487 }
```

```
164 int
165 sys_countfp(void)
166 {
167   return countfp();
168 }
```

위의 함수들을 syscall로 사용하기 위한 함수입니다. 인자가 필요 없기에 단순 return 하였습니다.

Result

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
init: starting sh
$ test0
[Test 0] default
ptp: 66 66
[Test 0] pass

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

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

$ 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

$
```

실행 결과 모두 잘 진행된 것을 확인하였습니다.

Trouble shooting

1. 부팅 시 Harddisk.. 이후 멈춤 현상

처음 코드를 작성하고 xv6를 부팅하였는데 init도 되지 않고 Harddisk.. 하고 멈추는 현상이 발생했습니다. 어느 부분이 잘못되었는지 도무지 모르겠어서 xv6 첫 상황으로 돌아가 처음부터 함수를 조금씩 추가하며 확인했는데, vm.c 소스파일의 mapping 함수 수정이 잘못이었습니다. mapping 함수에서 물리페이지 참조 횟수를 늘려주는 것이 맞다고 생각했는데, 그것이 멈춤 현상을 일으켰습니다. 아마도 운영체제 초기화 진행중에 mapping 함수를 사용하는데 거기서 이상하게 된 것이 아닐까 예상됩니다.

```
60 static int
61 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
62 {
63   char *a, *last;
64   pte_t *pte;
65
66   a = (char*)PGROUNDDOWN((uint)va);
67   last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
68
69   for(;;){
70     incr_refc(pa);
71     if((pte = walkpgdir(pgdir, a, 1)) == 0)
72       return -1;
73     if(*pte & PTE_P)
74       panic("remap");
75     *pte = pa | perm | PTE_P;
76     if(a == last)
77       break;
78     a += PGSIZE;
79     pa += PGSIZE;
80   }
81   return 0;
82 }
```

위 처럼 수정한 것이 잘못이었고 이후 kalloc 에서 처음 참조횟수 초기화와 copyuvm함수에서 incr_ref작업을 수행하여 해결하였습니다.

2. 이중 락 현상 수정

처음 kalloc함수에서 incr 함수 호출시 이중 락이 걸리는 현상이 발생하였습니다. 그렇기에 kalloc 함수에서는 incr 함수를 호출하지 않고 바로 참조횟수를 1로 설정하여 해결하였습니다.

3. Kfree 함수의 freepages 증가 위치 수정

Test0, test2의 경우 pass 가 잘 이루어지는데 test1, test3 에서 pass가 이루어지지 않는

경우가 발생하여 디버깅해보니 참조횟수가 0인 경우에서 freepages 를 증가하여야 하는데 항상 freepages 수를 증가하고있었습니다. 그 부분을 수정하니 모든 case의 통과가 이루어졌습니다.

4. 커널(가상) 메모리 와 물리 메모리

V2P 매크로가 왜 이루어지는지 몰랐고, 첫 kalloc 함수로 받아온 영역이 커널의 가상영역인 것을 몰랐습니다. 그렇기에 참조횟수를 수정할 때 V2P 매크로를 사용하지 않고 바로 참조횟수를 수정하였더니 이상하게 동작하였습니다. 찾아보니 kalloc은 처음 커널 영역의 공간을 할당하는 것이고 이것을 물리페이지로 매크로를 통해 변경후 사용해야 하는 것을 알았습니다.

5. Countptp 함수 이해

처음에 활성화 된 모든 페이지 테이블의 수 + 페이지 테이블을 가리키는 페이지의 수를 세는 건 줄 알았고 코드를 작성하였는데 test 케이스와 다른 큰 65605 같은 큰 값이 출력되어 이상했는데, 페이지 테이블을 가리키는 페이지 디렉토리의 개수를 세고 페이지 디렉토리 자체의 값만 count 하였더니 testcase와 같은 값이 출력되었습니다. 하지만 명세를 읽어봐도 모든 활성화된 테이블을 세라는 것 같은데 일단 test와 같이 작성하였습니다.