- Describe how you implement page allocation, deallocation, fork, copy-on-write, and TLB.

1. allocation

주어진 vpn과 writable 정보를 가지고 current process에 page allocate하기 위해서 먼저 page를 allocate할 page table의 위치를 찾아주었다. 이번 과제는 2-level page table로 구현하였으므로, vpn의 상위 16bit, 하위 16bit로 나누어 page table에 접근하여 pte의 위치를 찾도록 하였다. 그리고 접근하고자 하는 pagetable의 pte directory가 비어있을 경우(NULL)를 고려하여 비어있는 경우에는 동적할당을 이용해 공간을 할당해주고, 위의 방법을 이용해 page table에 접근하여 찾은 pte의 요소들을 채워 넣어주었으며, pagetable의 pte directory가 비어있지 않으면 바로 pte의 요소들을 채워 넣었다. pte의 요소를 채워넣을 때에는 valid 값을 1로 setting하고, read/write모드로 할당된 경우에는 writable도 1로 setting해주었다. 또한 mapcounts를 순회하며 아무 page도 참조되지 않은 frame(mapcounts[i]=0) 중 가장 작은 pfn을 가진 것을 찾아 해당 인덱스 i를 pte의 pfn값으로 setting해주었다.

2. dellocation

allocation과 동일한 방법으로 할당을 해제할 pte를 찾기 위해 vpn의 상위 16bit와 하위 16bit로 나누어 page table에 접근하여 할당을 해제할 pte를 찾아주었다. 할당을 해제하기 위해서는 pte의 모든 요소(valid, writable, pfn)을 모두 0으로 reset해주었다. 또한 해당 페이지가 free되면서, page가 참조하고 있던 frame의 pfn을 인덱스로 하는 mapcounts[pfn]의 값을 하나 줄여주었다. 마지막으로 만약 해당 page가 TLB에 존재했을 경우를 고려하여, TLB를 순회하며 free될 page가 존재하면 해당 TLB entry를 모두 0으로 reset해주었다.

3. fork

process가 switch될 때, switch하고자 하는 process의 pid가 readyqueue안에 존재하지 않는 process라면 fork가 실행되고, fork된 child process로 context switching되도록 구현하였다. 먼저 새로운 struct process* 타입의 변수 new를 하나 만들어, current process의 정보를 모두 복사하고, new의 pid만 switch하고자 했던 process의 pid로 설정한다. 이때, current에 할당된 모든 page들을 동일하게 갖고 있으므로, page들이 참조한 frame을 공유하게끔 current에 할당된 모든 page의 mapcounts들을 1 증가시켰으며, 두 개의 프로세스가 동일한 page frame을 참조하기 때문에 parent process와 child의 process의 writable을 모두 0으로 setting함으로써 write를 일시적으로 꺼주고, 대신 private을 1로 setting해주어, 프로세스가 원래 writable이었음을 표시해주었다. 이렇게 fork가 끝나면 current에 있던 process는 readyqueue에 넣고, 새로 생성된 process new를 current로 설정하며 context swtich해주었고, current page에 대한 TLB entry가 존재한다면 모두 reset해주어 TLB flush처리도 해주었다.

4. copy-on-write

page fault가 발생했을 때, current process의 writable이 0으로 꺼져있고, 동시에 private이 1로 setting되어 있으면, fork로 인해 일시적으로 write를 꺼놓은 것이므로 copy-on-write를 처리할 수 있도록 구현하였다. 먼저 write를 수행하고자 하는 page의 writable을 다시 1로 setting함으로써 write기능을 켜주었다. 만약 write하고자 하는 page가 mapping된 frame에 해당 process만 mapping되어 있다면, 참조하고 있는 frame을 바꾸어주지 않고, 만약 write하고자 하는 page가 mapping된 frame에 여러개의 process의 page가 참조하고 있다면 current의 page가 새로운 frame을 참조할 수 있도록 비어있는 frame을 찾아 mapping해준다. 마찬가지로 원래 참조하고 있던 frame의 mapcount를 1 줄이고, 새로 참조하게 된 frame의 mapcount를 1 증가시켜주었다.

5. TLB

lookup_tlb()에서는 TLB 배열을 순회하며 input으로 주어진 vpn과 동일한 vpn을 가진 tlb entry가 발견되면 pfn을 현재 process의 동일한 vpn을 가진 page의 pfn으로 설정해주고 hit을 알려주는 true를 반환하도록 하였고, 만약 주어진 vpn과 동일한 vpn을 가진 tlb entry가 tlb안에 존재하지 않으면 miss를 뜻하는 false를 반환하도록 구현하였다.

insert_tlb()에서는 TLB에 자리가 있으면 앞에서부터 차곡차곡 값을 삽입하도록 하였고, TLB에 자리가 없으면 가장 먼저 들어온(첫 번째 인덱스)값의 정보를 빼내고, 배열의 인덱스를 하나씩 앞으로 당겨 맨 뒤(마지막 인덱스)에 새로운 정보가 들어오도록 FIFO방식으로 구현하였다. 물론 TLB entry에 정보를 할당해줄 때, tlb entry의 valid를 1로 setting해주고, vpn과 pfn을 주어진 값으로 setting해주었다. (free된 페이지에 대한 tlb의 entry 처리는 deallocation에 명시)

- Describe your entire journey to complete the assignment. You must clarify why you choose the particular design you use over other alternatives, what is your key idea in implementing those features, explains your failed tries, the analysis on the causes of those failures, and how you resolved them.
Must include at least one case of your failure, what the problem was, and how you fixed it.

처음 과제를 시작할 때에는 demand paging의 개념이 완벽하게 이해되지 않아서 전체적인 설계없이 handout에서 주어진 힌트에 주어진대로 allocation, deallocation, fork..의 순서로 차근차근 구현하였다. 따라서 특별한 전략보다는 문제에서 요구하는 기능들을 충실히 수행할 수 있도록 하나씩 순차적으로 구현해나갔다.

demand paging을 구현하면서 전체적으로 변하는 것들은 pte와 mapcounts, TLB이었기 때문에, 각 기능을 수행할 때, 이 세가지의 요소들이 어떻게 변하는지를 따져주어 반영해주는 것을 중심으로 구현하였고, 이 외에 각 기능마다 추가적으로 필요한 것들(linked list, current 등)을 반영해주는 형태로 과제를 수행하였다.

각 기능별 핵심 아이디어를 나열하자면 다음과 같다. allocation은 할당할 page의 pte 요소들을 적절하게 설정해주는 것, 특히 사용가능한 frame을 잘 찾아 알맞은 pfn을 설정해주는 것, deallocation은 할당을 해제하고자 하는 page를 찾아 해당 page와 TLB entry를 초기화해주는 것, switch는 readyqueue에서 switch하고자 하는 process를 찾아, current로 설정해주고, 실행하던 process는 readyqueue에 넣는 것, 만약 switch하고자 하는 process가 없다면 fork를 이용해 current와 동일한 정보를 갖는 process를 만들어, current와 context switch하고, parent/child process의 write을 임시로 꺼주고 private을 1로 setting해주어 이를 표시하는 것, 마지막으로 TLB는 TLB안에서 정보를 찾을 때에는 TLB를 순회하며 vpn을 비교해 찾는 것과 hit이면 pfn을 올바르게 setting해줘야하는 것, 그리고 TLB에 정보를 삽입할 때에는 빈 entry를 찾아 값을 setting해 주되, TLB가 꽉 차있으면 FIFO방식으로 TLB정보를 빼내고 정보를 다시 삽입하는 것이다.

사실 문제에서 요구한 기능과 핵심아이디어는 거의 유사하기 때문에 위의 핵심아이디어만 잘 구현해주면 과제를 해결할 수 있었다. 다만, 여러 가지 문법적인 이유로 인하여서 어려움을 겪기도 하였다. 바로 process switch를 수행할 때, 각 process의 page table이 따로 업데이트되어야 하는데, 모든 process에 대해 page table이 공유되는 문제였다. process0에서 vpn0번의 page를 할당하고 fork를 통해 process1이 생성시킨 후에 switch 1을 하여 process1에서 vpn 0번 page를 free하면 process1에서만 page가 free되어야하는데, process0에서도 vpn 0번 page가 사라지는 것을 알 수 있었다. 처음에는 fork된 process 자체가 context switch되지 않아서 생기는 문제인줄 알고, process가 readyqueue자체에 안 들어갔을 것이라고 생각하여, dump_status() 함수를 만들어 readyqueue의 변화를 관찰하였지만, process switch자체는 잘 수행하고 있음을 알 수 있었다. 따라서 fork된 새로운 process의 pagetable에 current process의 pagetable을 통째로 복사해주어, 포인터를 공유하여 생기는 문제라고 파악했고, 새로운 process의 pagetable의 모든 page directory를 각각 동적할당해주어, 각각의 pte의 값을 일일이 모두 복사해주어 pagetable이 공유되지 않고, 독립적으로 사용될 수 있도록 해결해주었다. 이 외에는 NULL처리를 해주지 않아

생기는 오류와 같은 자잘한 실수밖에 없었다.

- Explain what happens to the page tables while running testcases/cow-2. Show the final state of the page tables for process 0, 1, and 2, and explain why they are like that.

< final state of the page tables for process 0, 1, 2 >

PID 0		
vpn/16 vpn%16	valid:v writable:w	pfn
00:00	v	0
00:01	v	1
00:02	vw	2
00:03	vw	6
00:04		
00:05		
00:06		
00:07		
00:08		
00:09		
00:10		
00:11		
00:12		
00:13		
00:14		
00:15		

PID 1		
vpn/16 vpn%16	valid:v writable:w	pfn
00:00	v	0
00:01	v	1
00:02	v	4
00:03	v	3
00:04		
00:05		
00:06		
00:07		
00:08		
00:09		
00:10		
00:11		
00:12		
00:13		
00:14		
00:15		

PID 2		
vpn/16 vpn%16	valid:v writable:w	pfn
00:00	v	0
00:01	v	1
00:02	v	4
00:03	vw	5
00:04		
00:05		
00:06		
00:07		
00:08		
00:09		
00:10		
00:11		
00:12		
00:13		
00:14		
00:15		

cow-2 명령어	page table 변화
alloc 0 r	process0의 vpn0을 통해 접근한 00:00위치에 read only mode로 page할당, 모든 frame비어있으므로, pfn 0을 참조
alloc 1 r	process0의 vpn1을 통해 접근한 00:01위치에 read only mode로 page할당, pfn0에는 다른 page가 참조하므로, 비어있는 pfn 1참조
alloc 2 rw	process0의 vpn2을 통해 접근한 00:02위치에 read/write mode로 page할당, pfn0,1에는 다른 page가 참조하므로, 비어있는 pfn 2참조
alloc 3 rw	process0의 vpn3을 통해 접근한 00:03위치에 read/write mode로 page할당, pfn0,1,2에는 다른 page가 참조하므로, 비어있는 pfn 3참조
switch 1	process0을 fork하여 process1을 만든다. process1으로 context switch된다. process1의 pagetable은 process0의 pagetable과 동일하며, page들이 참조하는 frame을 공유하기 때문에 process0의 page들을 할당한 frame들의

	mapcount가 1씩 증가되고. process0과 process1의 writable한 page들의 write가 일시적으로 꺼지고, private은 1로 설정된다.
show	table변화 x
read 1	table변화 x
write 2	vpn2를 갖는 page의 write가 일시적으로 꺼져있으므로 다시 켜고, vpn2가 참조하는 frame에 다른 process의 page도 참조하고 있으므로, 비어있는 frame중 가장 작은 pfn을 갖는 pfn4의 frame을 참조하여, vpn2의 page의 writable은 1로, private은 0으로, pfn이 4로 setting된다.
show	table변화 x
switch 2	process1을 fork하여 process2을 만든다. process0으로 context switch된다. process2의 pagetable은 process1의 pagetable과 동일하며, page들이 참조하는 frame을 공유하기 때문에 process1의 page들을 할당한 frame들의 mapcount가 1씩 증가되고. process1과 process2의 writable한 page들의 write가 일시적으로 꺼지고, private은 1로 설정된다.
show	table변화 x
read 1	table변화 x
write 3	vpn3을 갖는 page의 write가 일시적으로 꺼져있으므로 다시 켜고, vpn3이 참조하는 frame에 다른 process의 page도 참조하고 있으므로, 비어있는 frame중 가장 작은 pfn을 갖는 pfn5의 frame을 참조하여, vpn3의 page의 writable은 1로, private은 0으로, pfn이 5로 setting된다.
show	table변화 x
switch 0	table변화 x (process0으로 context switch)
read 2	table변화 x
write 2	vpn2을 갖는 page의 write가 일시적으로 꺼져있으므로 다시 켜고, vpn2이 참조하는 frame에 다른 process의 page가 참조하지 않으므로, 변하지 않고 동일한 pfn을 갖는다. vpn2의 page의 writable은 1로, private은 0으로 setting된다.
write 3	vpn3을 갖는 page의 write가 일시적으로 꺼져있으므로 다시 켜고, vpn3이 참조하는 frame에 다른 process의 page도 참조하고 있으므로, 비어있는 frame중 가장 작은 pfn을 갖는 pfn6의 frame을 참조하여, vpn3의 page의 writable은 1로, private은 0으로, pfn이 6으로 setting된다.
show	table변화 x
pages	table변화 x

- Lesson learned

paging의 개념에 대해 어느정도 알고 있었고, 수업을 통해 전체적으로 어떤 흐름으로 동작하는 것을 이해하고 있었지만, 내부적인 디테일한 작동법이라던가 발생가능한 문제가 무엇이 있는지 명확하게 파악하지 못했었다. 하지만 실제로 fork와copy-on-write를 구현해보며 fork를 통해 shared memory를 구현하는 것, 이에 대한 예외(copy-on-write에 관한)처리를 해주지 않았을 때, 문제가 발생하는 것을 직접 확인하면서 왜 copy-on-write처리를 해주어야하는지, 어떻게 구현해주어야 하는지 완벽하게 파악할 수 있었다.

또한 hierarchical page table을 통해 memory에 어떻게 접근해야하는지 잘 감이 오지 않았었는데, 2-level page table을 구현하면서 page table에 접근하는 방법을 쉽게 이해할 수 있었다. 마찬가지로 hierarchical page table을 통해 메모리에 접근하면, 매우 큰 memory에 더 쉽게 접근하여 page를 mapping할 수 있음을 깨달을 수 있었다.

마지막으로 abstract된 변수자체가 포인터 변수가 아니더라도, 구조체 안에 포인터 변수가 포함되어 abstract되어 있다면, 새로운 객체를 만들어 copy할 때, 포인터 변수를 공유할 수 있음을 깨달았고, 주의할 수 있도록 하였다.