

# Assignment 4

과목명	운영체제
학과	컴퓨터정보공학부
학번	2020202096
성명	우성원
교수님	김태석교수님
제출일	2023.11.23

## ● Introduction

4\_1과제는 PID를 바탕으로 프로세스 정보를 출력하는 과제이다.task\_struct와 vm\_area\_struct를 이용하여 구현하였으며 task\_struct의 mm구조체변수와 vma구조체변수이다.vma변수는 가상메모리 구조체변수로 가상메모리에 대한 여러 정보를 담고있다.mm구조체도 task\_struct의 코드영역,힙영역,데이터영역등의 다양한 변수들을 담고있다.4\_2과제는 동적재컴파일링을 수행하는 과제로 원본 어셈블리코드를 가상 메모리에 담은후 이를 최적화하는 과제이다.최적화 방법은 연속된 명령어 군집을 대표명령어하나로 대체하는 방식으로 구현하였으며 실험결과 평균 60%의 수행시간이 절감되는 효과를 얻었다.

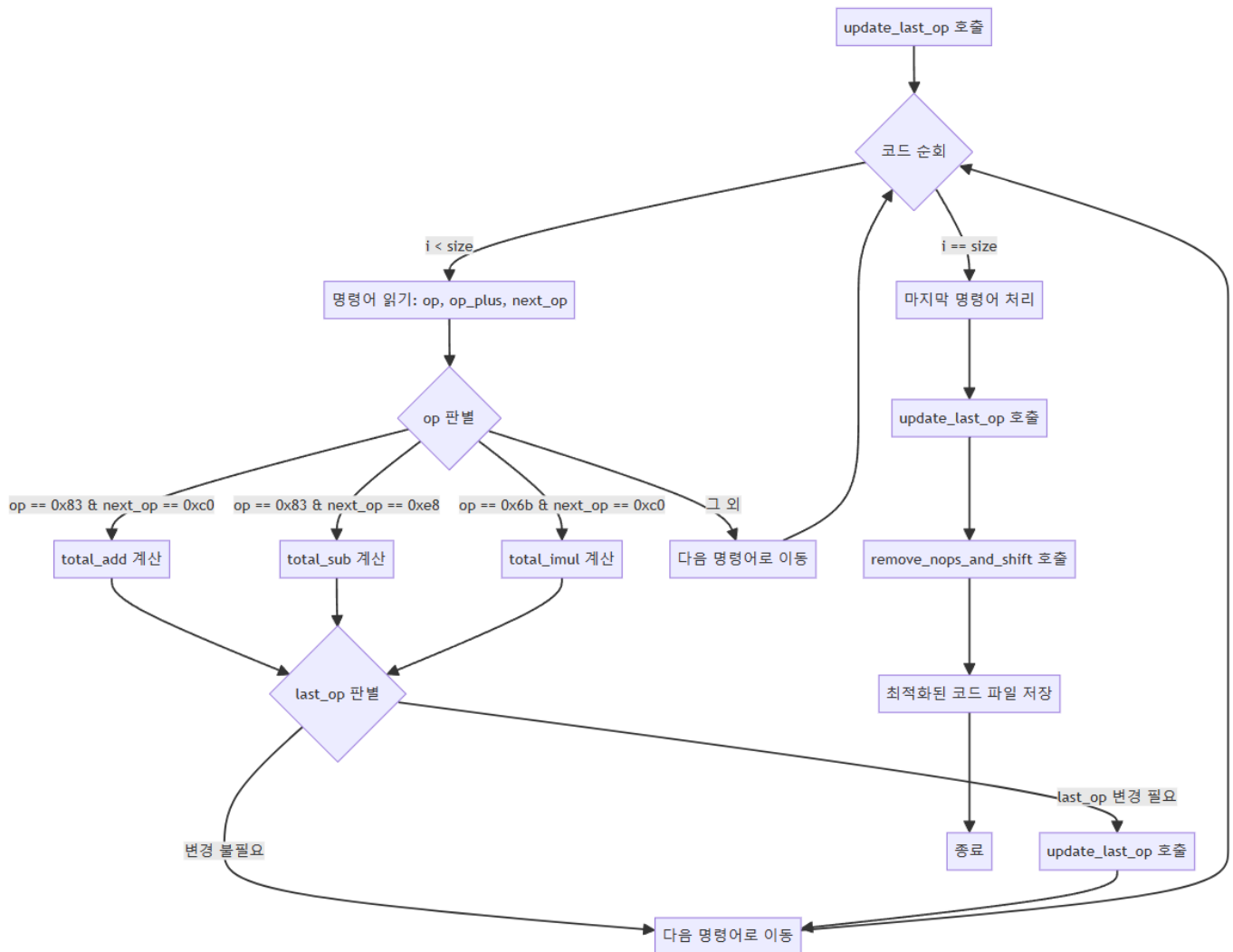
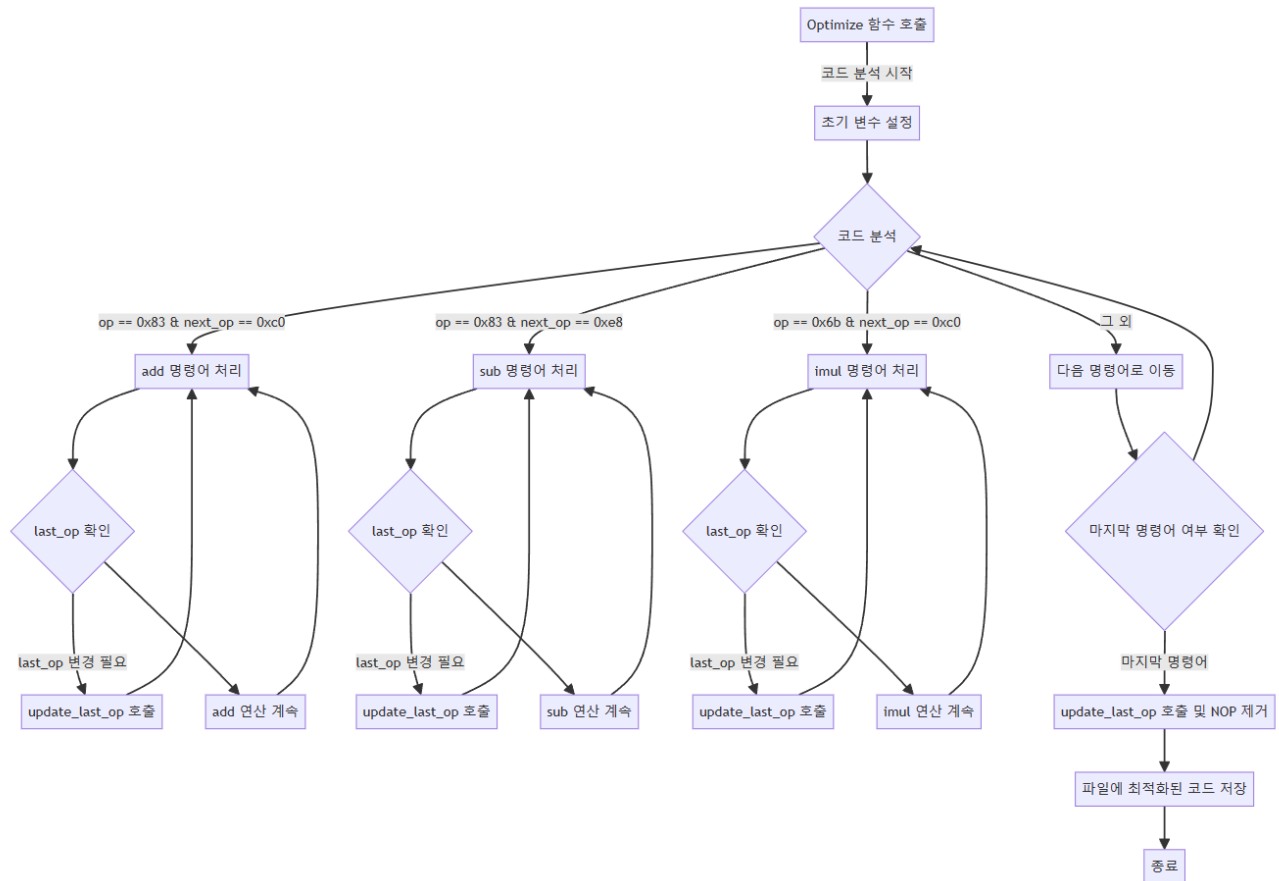
## ● Assignment 4-1

4-1 과제에서는 프로세스 이름, PID, 그리고 해당 프로세스의 task\_struct에 포함된 가상 메모리, 데이터, 코드, 힙 영역과 이들의 전체 주소에 접근해야 한다. 이를 위해 task\_struct와 vm\_area\_struct 구조체를 사용했다. task\_struct에서는 메모리 관리 구조체 변수인 mm에 접근해 start\_code, end\_code, start\_data, end\_data, start\_brk, brk를 이용해 해당 task\_struct의 데이터, 코드, 힙 주소 정보를 얻었다. 또한 mm 구조체의 mmap을 사용해 VMA 정보를 탐색했다. VMA 가상 메모리 구조체 정보를 활용해 해당 task\_struct의 가상 메모리 세그먼트 정보를 조사했다. VMA 구조체의 vm\_start와 vm\_end 변수를 사용해 각 세그먼트의 가상 주소를 확인할 수 있었다. 이어서, VMA의 vm\_file 변수에 접근해 vm\_file이 null이 아닌 경우에만 해당 VMA 정보를 출력했다. 이는 세그먼트의 경로를 출력할 수 있기 때문이다. 마지막으로, vm\_file의 f\_path를 통해 세그먼트의 파일 경로를 출력했다.

```
[117741.439947] ##### Loaded files of a process 'test(35905)' in VM #####
[117741.439949] men[400000~401000] code[400000~4007b4] data[600e10~601048] heap[225a000~225a000] /home/os2020202096/assignment4_1/test
[117741.439952] men[600000~601000] code[400000~4007b4] data[600e10~601048] heap[225a000~225a000] /home/os2020202096/assignment4_1/test
[117741.439953] men[601000~602000] code[400000~4007b4] data[600e10~601048] heap[225a000~225a000] /home/os2020202096/assignment4_1/test
[117741.439954] men[7f513933b000~7f51394fb000] code[400000~4007b4] data[600e10~601048] heap[225a000~225a000] /lib/x86_64-linux-gnu/libc-2.23.so
[117741.439956] men[7f51394fb000~7f51396fb000] code[400000~4007b4] data[600e10~601048] heap[225a000~225a000] /lib/x86_64-linux-gnu/libc-2.23.so
[117741.439957] men[7f51396fb000~7f51396ff000] code[400000~4007b4] data[600e10~601048] heap[225a000~225a000] /lib/x86_64-linux-gnu/libc-2.23.so
[117741.439958] men[7f51396ff000~7f5139701000] code[400000~4007b4] data[600e10~601048] heap[225a000~225a000] /lib/x86_64-linux-gnu/libc-2.23.so
[117741.439960] men[7f5139705000~7f513972b000] code[400000~4007b4] data[600e10~601048] heap[225a000~225a000] /lib/x86_64-linux-gnu/ld-2.23.so
[117741.439961] men[7f513972a000~7f513992b000] code[400000~4007b4] data[600e10~601048] heap[225a000~225a000] /lib/x86_64-linux-gnu/ld-2.23.so
[117741.439962] men[7f513992b000~7f513992c000] code[400000~4007b4] data[600e10~601048] heap[225a000~225a000] /lib/x86_64-linux-gnu/ld-2.23.so
[117741.439964] #####
os2020202096@ubuntu:~/assignment4_1$
```

결과를 보면 프로그램 시작시 test라는 테스트프로그램의 이름과 해당 pid가 잘 나오는 것을 확인할수있다. 또한 원본프로그램 세그먼트의 주소와 코드영역,데이터영역,힙영역의 주소가 나오는 것을 볼수있고 해당 경로또한 나오는 것을 확인할수있다.이어서 해당 프로그램의 세그먼트들의 가상메모리 주소,코드영역,데이터영역,힙영역의 주소와 파일경로도한 잘 나오는 것을 확인할수있다.

## ● Assignment 4-2

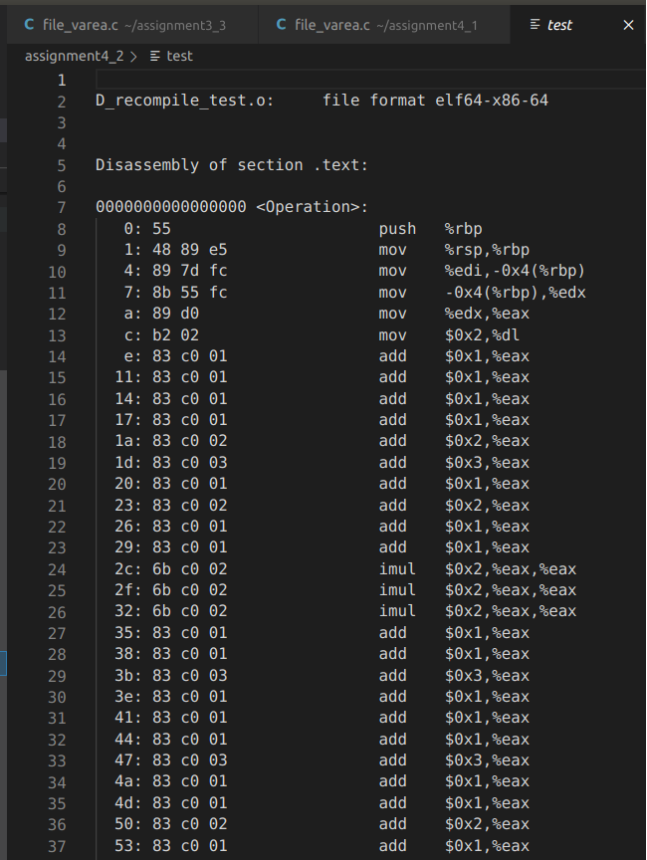


4-2 과제에서 구현한 어셈블리 코드 최적화 부분을 설명하겠다. 먼저 각 명령어가 add인지 sub인지 imul 인지 확인후 해당 명령어의 연속적인 군집이 발견시 마지막 명령어만 대표 명령어로 남겨놓고 그 외 연속적인 해당 명령어 군집을 모두 nop로 바꾼다. 여기서 add,sub,imul의 산술값을 total이라는 임의의 변수에 저장해놓고 대표명령어에 할당한다. 이를 통해 연속적인 여러 명령을 하나의 명령으로 바꿀수있다. 함수에 대해 설명하자면 optimize함수는 메모리에서 어셈블리어를 읽고 하나의 군집을 판별하는 역할을 한다. 군집이 바뀔 때 update\_last\_op함수가 호출되는데 이 함수는 total이라는 연속된명령어의 산술값의 총합을 대표명령어에 할당한다. 또한 대표명령어의 인덱스를 기반으로 그 전의 대표명령어와 같은 군집의 명령어는 모두 nop로 바꾸고 대표명령어에 산술값을 할당한다.

최종적으로 중복된 명령어는 모두 하나의 명령어로 바뀌고 nop로 처리된 명령어는 remove\_nops\_and\_shift함수를 통해서 모두 제거되고 중복이 없는 어셈블리 코드로 바뀐다. 다만,div는 이러한 방식으로 처리하기가 어려워 구현하지 못하였고 예외처리하였다.

## -objdump

```
os2020202096@ubuntu:~/assignment4_2$ ls
decompile D_recompile D_recompile.c D_recompile_test.o Makefile optimized_code.txt test test2
os2020202096@ubuntu:~/assignment4_2$ gcc -c D_recompile_test.c
os2020202096@ubuntu:~/assignment4_2$ objdump -d D_recompile_test.o > test
os2020202096@ubuntu:~/assignment4_2$ ls
decompile D_recompile D_recompile.c D_recompile_test.c D_recompile_test.o D_recompile_test.o Makefile optimized_code.txt test test2
os2020202096@ubuntu:~/assignment4_2$
```



objdump 명령어는 컴파일된 바이너리 파일들을 분석하는데 사용되는 도구다. -D 옵션은 파일 내의 모든 섹션들을 디스어셈블한다. 이 과정은 기계어 코드를 어셈블리 코드로 변환해 출력하는 역할을 한다. 그리고 이렇게 변환된 어셈블리 코드는 > 리다이렉션을 통해 test 파일에 저장된다. 이 방식은 프로그램의 실행 과정을 이해하거나 코드가 어떻게 구성되어 있는지 분석할 때 유용하다.

왼쪽 사진을 보면 어셈블리 코드를 볼수있는데 .o파일은 컴파일된 오브젝트파일로 이를 objdump를 이용하여 컴파일된 D\_recompile\_test.o를 test라는 이름으로 디스어셈블한 모습이다.

여기서 add랑 sub등 명령어가 중복으로 있는 것을 보고 연속적인 명령어군집을 합쳐서 하나의 명령어로 만들고 나머지를 명령어를 nop로 바꾼후 마지막에 nop를 전부 없애면 어떨까라는 아이디어가 생각이났다. 이를 구현하였고 결과는 평균수행시간이 60% 줄어든 효과를 볼수있었다.

## -Result

```
os2020202096@ubuntu:~/assignment4_2$ sync
os2020202096@ubuntu:~/assignment4_2$ echo 3| sudo tee /proc/sys/vm/drop_caches
3
os2020202096@ubuntu:~/assignment4_2$ gcc -o test2 D_recompile_test.c
os2020202096@ubuntu:~/assignment4_2$ ./test2
result: 15
Data was filled to shared memory.
os2020202096@ubuntu:~/assignment4_2$ make
gcc -o drecompile D_recompile.c
os2020202096@ubuntu:~/assignment4_2$ ./drecompile
result: 15
Total execution time: 0.102000 sec
os2020202096@ubuntu:~/assignment4_2$ sync
os2020202096@ubuntu:~/assignment4_2$ echo 3| sudo tee /proc/sys/vm/drop_caches
3
os2020202096@ubuntu:~/assignment4_2$ ./test2
result: 15
Data was filled to shared memory.
os2020202096@ubuntu:~/assignment4_2$ make dynamic
gcc -Ddynamic -o drecompile D_recompile.c
os2020202096@ubuntu:~/assignment4_2$ ./drecompile
result: 15
Total execution time: 0.032000 sec
os2020202096@ubuntu:~/assignment4_2$ █
```

결과를 보면 동적컴파일링한것과 그렇지 않은것의 차이가 두드러지는 것을 확인할수있다.실제로 50set를 수행하였을 때 60% 가까이 수행시간이 절감되는 것을 확인할수있었다.

## -50 Set Test

일반 컴파일링		동적 컴파일링	
1	0.101	1	0.03
2	0.073	2	0.032
3	0.064	3	0.05
4	0.087	4	0.035
5	0.07	5	0.031
6	0.1	6	0.029
7	0.089	7	0.043
8	0.066	8	0.029
9	0.069	9	0.031
10	0.087	10	0.037
11	0.066	11	0.034
12	0.066	12	0.033
13	0.072	13	0.032
14	0.074	14	0.031
15	0.068	15	0.03
16	0.065	16	0.029
17	0.071	17	0.028
18	0.069	18	0.027
19	0.067	19	0.026

20	0.068	20	0.025
21	0.07	21	0.026
22	0.072	22	0.027
23	0.071	23	0.028
24	0.069	24	0.029
25	0.07	25	0.03
26	0.101	26	0.031
27	0.078	27	0.032
28	0.087	28	0.027
29	0.065	29	0.026
30	0.101	30	0.025
31	0.063	31	0.026
32	0.062	32	0.027
33	0.061	33	0.028
34	0.06	34	0.029
35	0.059	35	0.03
36	0.058	36	0.031
37	0.065	37	0.032
38	0.072	38	0.033
39	0.064	39	0.034
40	0.065	40	0.035
41	0.088	41	0.036
42	0.083	42	0.037
43	0.051	43	0.038
44	0.062	44	0.039
45	0.075	45	0.04
46	0.061	46	0.03
47	0.09	47	0.032
48	0.07	48	0.031
49	0.089	49	0.029
50	0.066	50	0.037
평균	0.0728	평균	0.03154

최종적으로 50set를 보면 평균적으로 60% 수행시간이 빨라진 것을 확인할수있다.  
최적화의 중요성을 시간적으로 확인할수있는것이다.

## ● 고찰

4.2에서 생각한 최적화 기법을 구현하는 부분에서 상당히 어려웠다. 구체적으로는 index를 이용해서 연속된 명령어 군집 중 대표명령어를 저장하고 그 총합을 갱신하는 과정에 부수적인 여러 index를 이용해야하고 메모리에서 index를 관리하는데에 하나라도 예외가 발생하면 모든 index에 영향을 주기 때문에 더욱 복잡했던것같다.특히,처음과 마지막에 있는 최적화 하면 안될 예외적인 명령어들을 생각하면서 index를 관리하는 것이 힘들었다. 하지만 실제 최적화를 하고 시각적인 성능향상을 경험하니 이래서 최적화가 중요하다는 것을 깨달았다. 이번 과제에서 div를 구현하지 못하였지만 아이디어는 유지하고 코드를 조금 더 구조적으로 잘 짚으면 가능할것같다는 생각이 들었다.그리고 좀 더 예외사항에 robust한 프로그래밍에 중요성을 깨달았다.또한 구조적으로 코드를 잘 짜는 법에 대해서 깊게 생각하는 계기가 되었다.굳이 함수를 2개를 나눌필요도 없었을것같고 더 짧고 더 적은 변수로 가시성있게 짤수있는 방법에 대해서 고민하게 되었다.추후 과제에는 이러한 부분에 초점을 맞추어 과제를 수행할것이다.

## ● 참조

[\[Linux Kernel\] 메모리 관리 : 가상 메모리 \(oopy.io\)](https://oopy.io/)