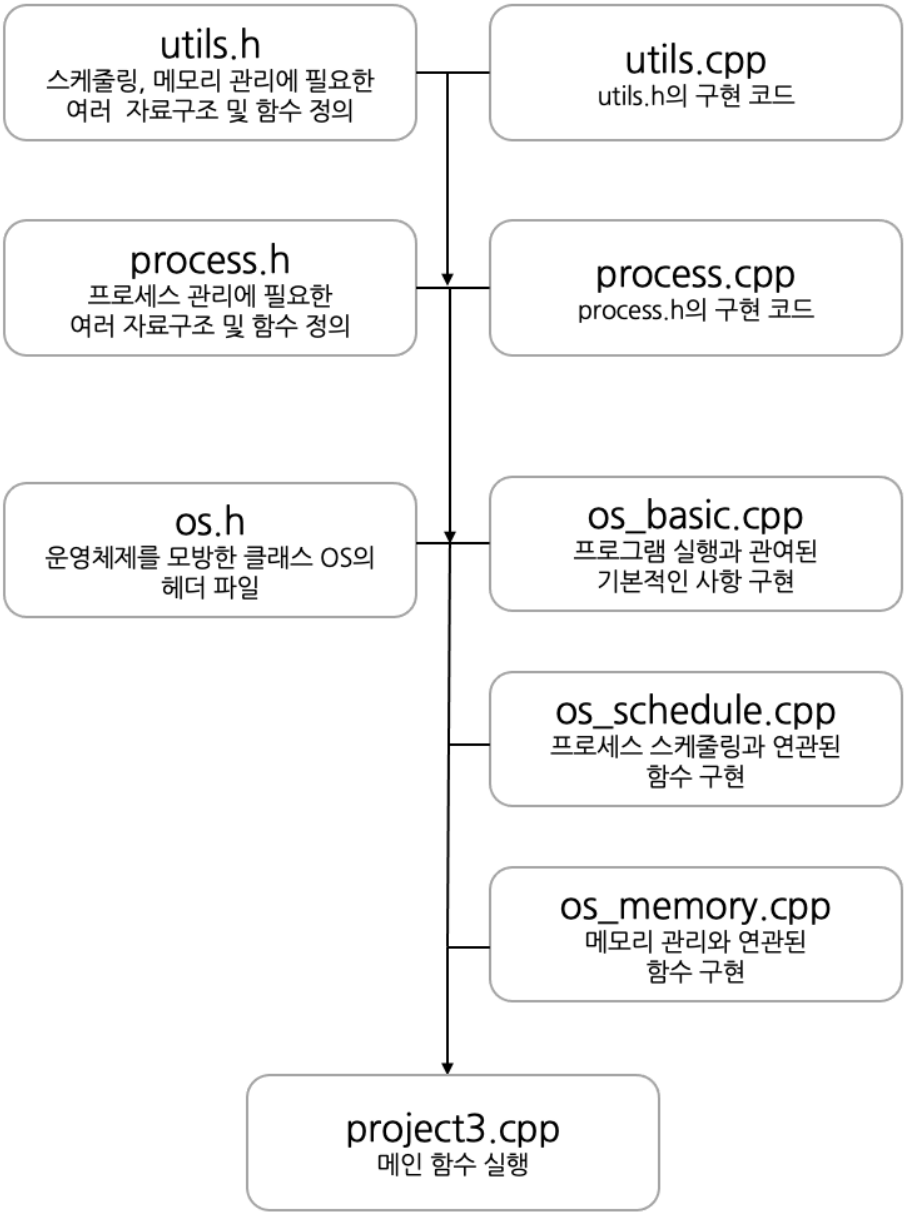
2021-1 운영체제 과제 #3

응용통계학과 2017122063 정재은

1. 작성한 프로그램의 동작 과정 및 구현 방법

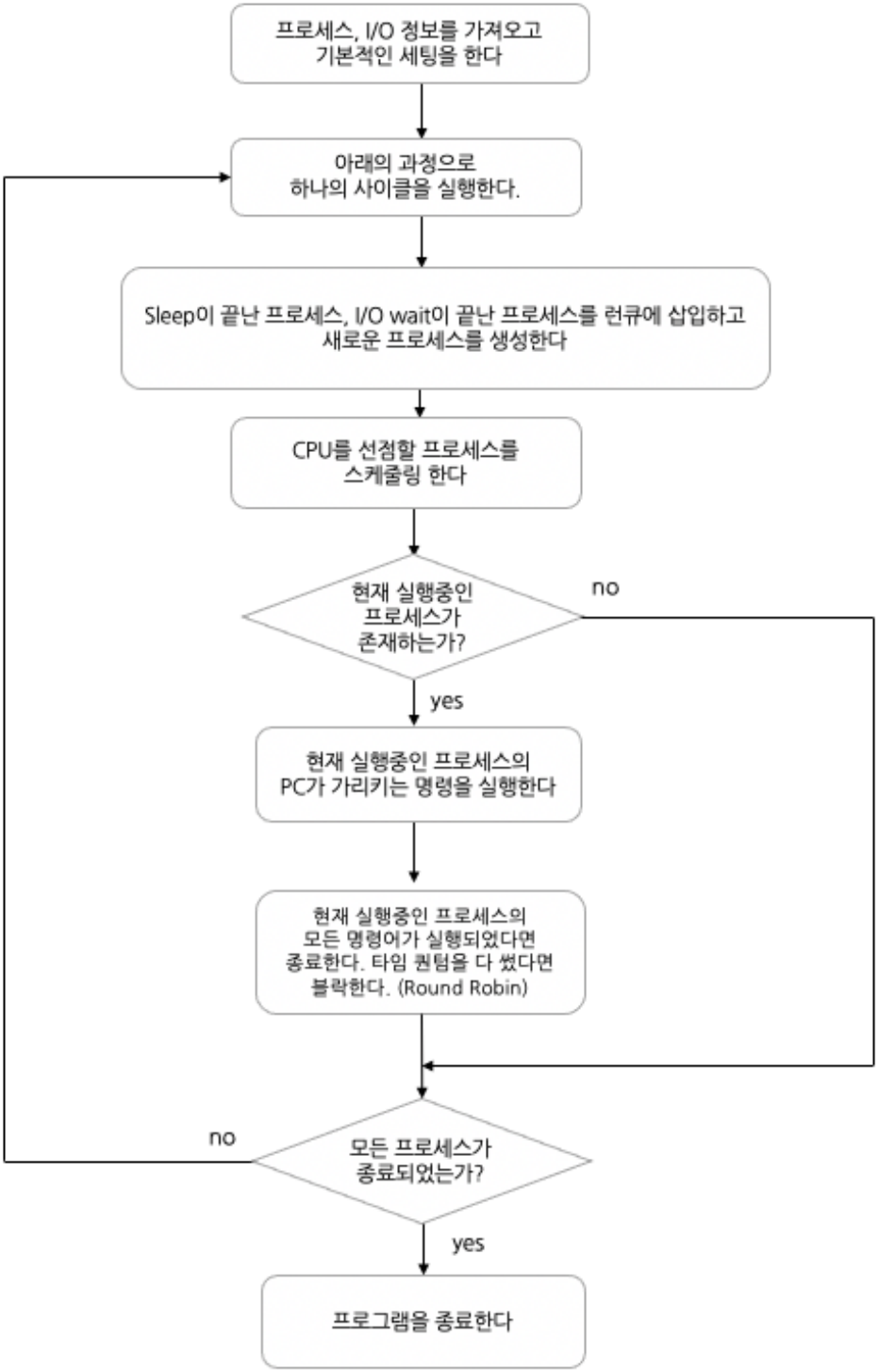
   A. 전체 구성

   다음은 이번 과제의 전체적인 구성도이다. 여러 헤더 파일과 cpp 파일을 작성하여 전체 소스 코드를 모듈화 했고 보다 편리하게 코드를 관리하고 수정할 수 있었다.



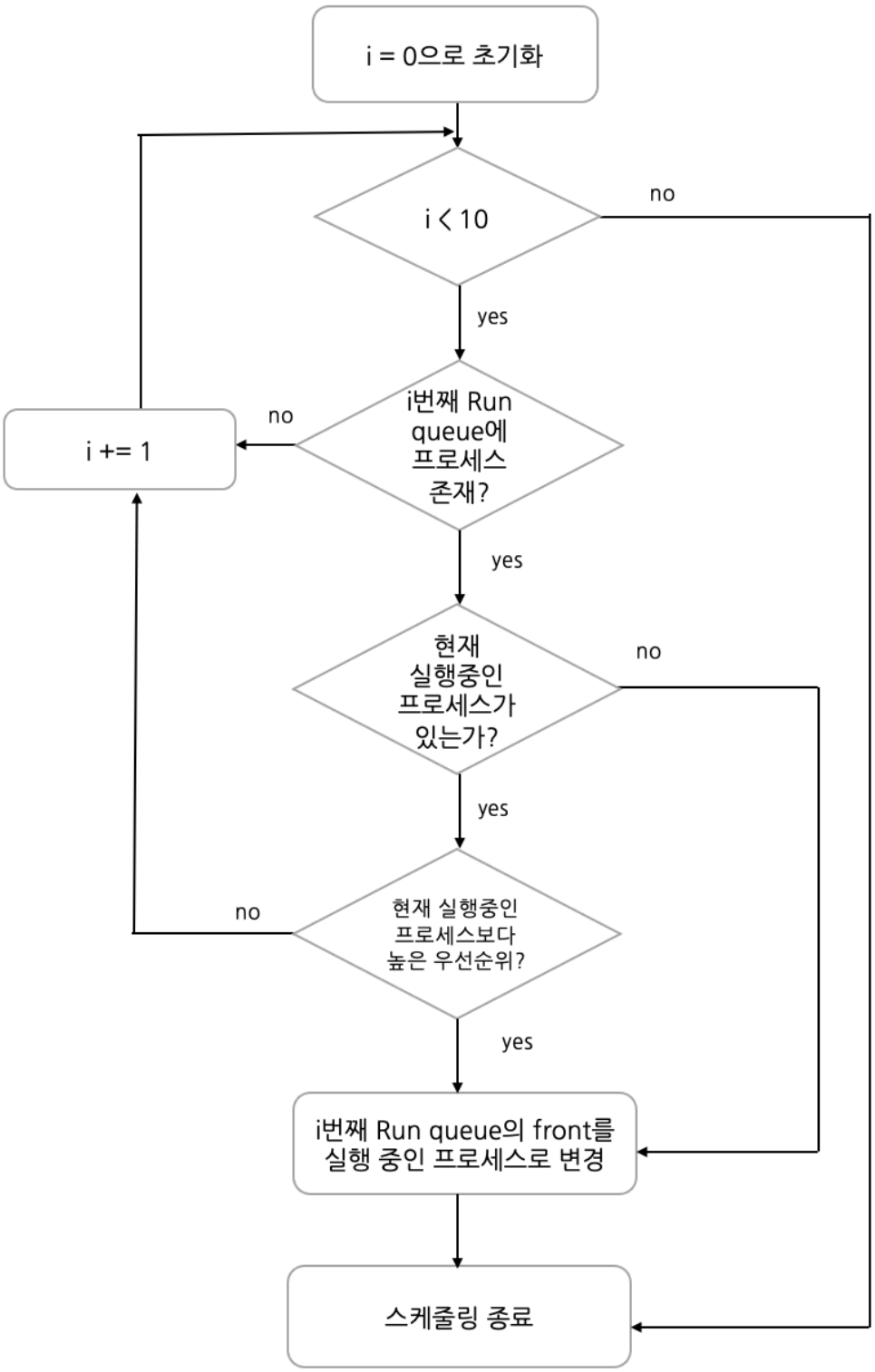
B. 전체 흐름도

다음은 이번 과제의 전체 흐름을 나타내는 순서도이다.



C. 스케줄링

아래는 스케줄링 알고리즘을 도식화하여 나타낸 것이다.



D. 명령어 실행

a. Memory allocation

가상 메모리에 필요한 페이지 수만큼 연속적으로 메모리를 할당하는 작업을 수행한다. 가상 메모리를 선형적으로 탐색하며 연속적으로 비어있는 공간을 찾고 가상 메모리를 할당한다. 현재 실행중인 페이지(nowPage)를 해당 page table entry 로 변경하고 페이지 테이블에도 <페이지 아이디, page table entry> 쌍을 추가한다.

b. Memory access

가상 메모리에 할당된 페이지가 물리 메모리에도 할당되도록 하는 작업이다. 예전에 memory access 가 일어났다면 allocation id 를 부여하지 않지만 처음 access 하는 것이라면 allocation id 를 부여한다.

물리 메모리에 할당되어 있지 않다면 버디 시스템을 이용하여 물리 메모리에 할당한다. 물리 메모리에 공간이 충분하지 않다면 페이지 교체 알고리즘을 통해 공간이 충분할 때까지 페이지를 교체한다. 페이지 교체 알고리즘에 대한 자세한 설명은 E 에서 기술하였다.

c. Release memory

가상 메모리와 물리 메모리에 할당된 페이지를 해제한다. b 와 마찬가지로 버디 시스템을 이용하여 해제하였다. 해제한 노드는 항상 말단 노드(leaf node)이므로 해제한 노드의 형제 노드도 비어 있다면 버디 시스템 알고리즘에 따라 두 노드를 합쳐주었다. 재귀적으로 이러한 과정을 수행한다.

d. Non-memory instruction

메모리와 관련된 명령이 아니므로 프로세스의 코드에서 명령을 하나 삭제하는 것 이외에는 별도의 작업을 하지 않았다.

e. Sleep

마지막 명령이 아니라면 실행하고 있는 프로세스를 sleep list 에 삽입한다.

f. I/O Wait

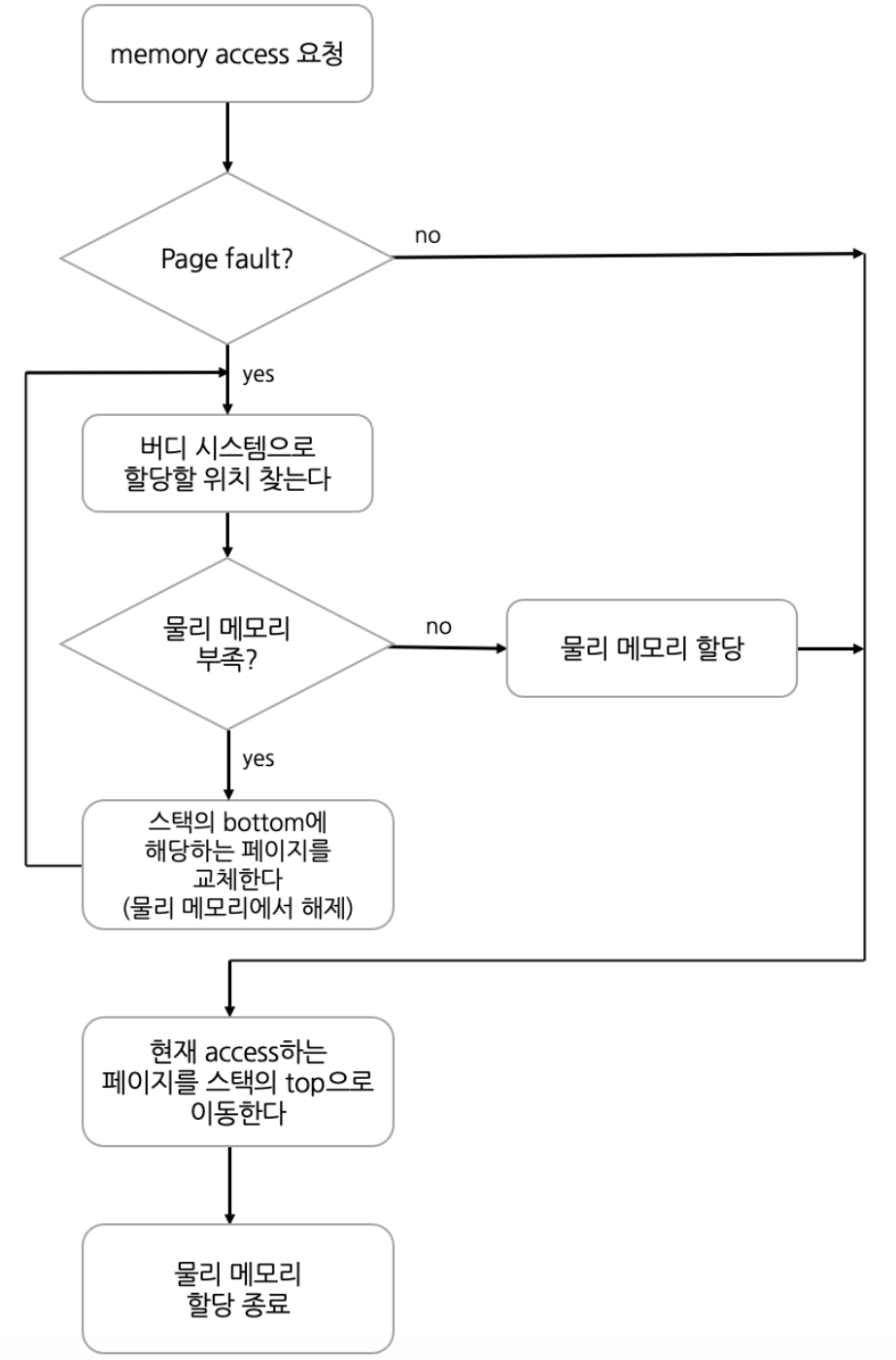
마지막 명령이 아니라면 실행하고 있는 프로세스를 I/O wait list 에 삽입한다.

E. 페이지 교체 알고리즘

다음은 버디 시스템에 의해 물리 메모리를 할당할 때, 할당할 공간이 부족할 때 물리 메모리에 존재하는 페이지 중 어떤 것을 교체할지 선택하는 알고리즘이다.

a. LRU

스택 기반으로 LRU 페이지 교체 알고리즘을 구현하였고, 알고리즘은 아래와 같다. 물리 메모리에 존재하는 페이지들(valid bit == 1 인 페이지들)을 저장하는 스택을 따로 만들어주었다.

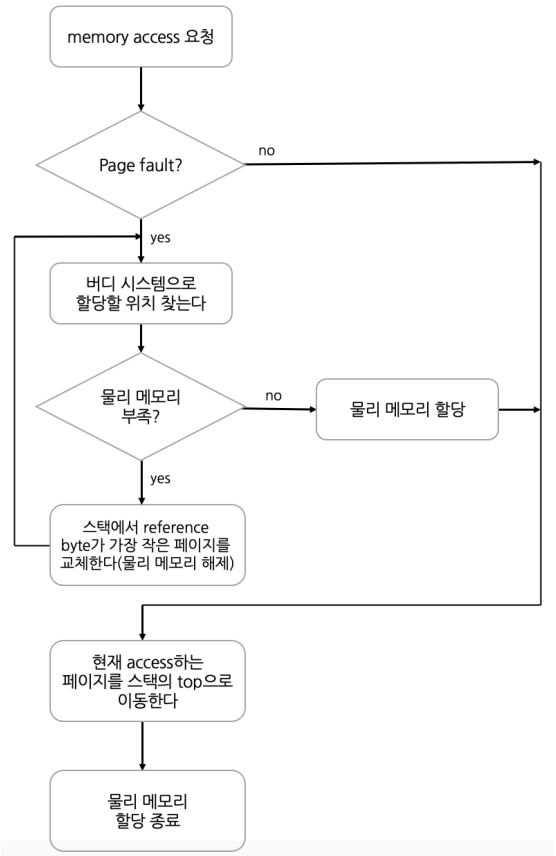


b. Sampled LRU

B 에서는 나타내지 않았지만 refreshRefByte() 함수를 호출하여 페이지 교체 알고리즘이 sampled LRU 이고 사이클이 8 의 배수일때 실행 중이거나 block 되어 있는 모든 프로세스의 페이지에 대해서 reference byte 를 갱신했다.

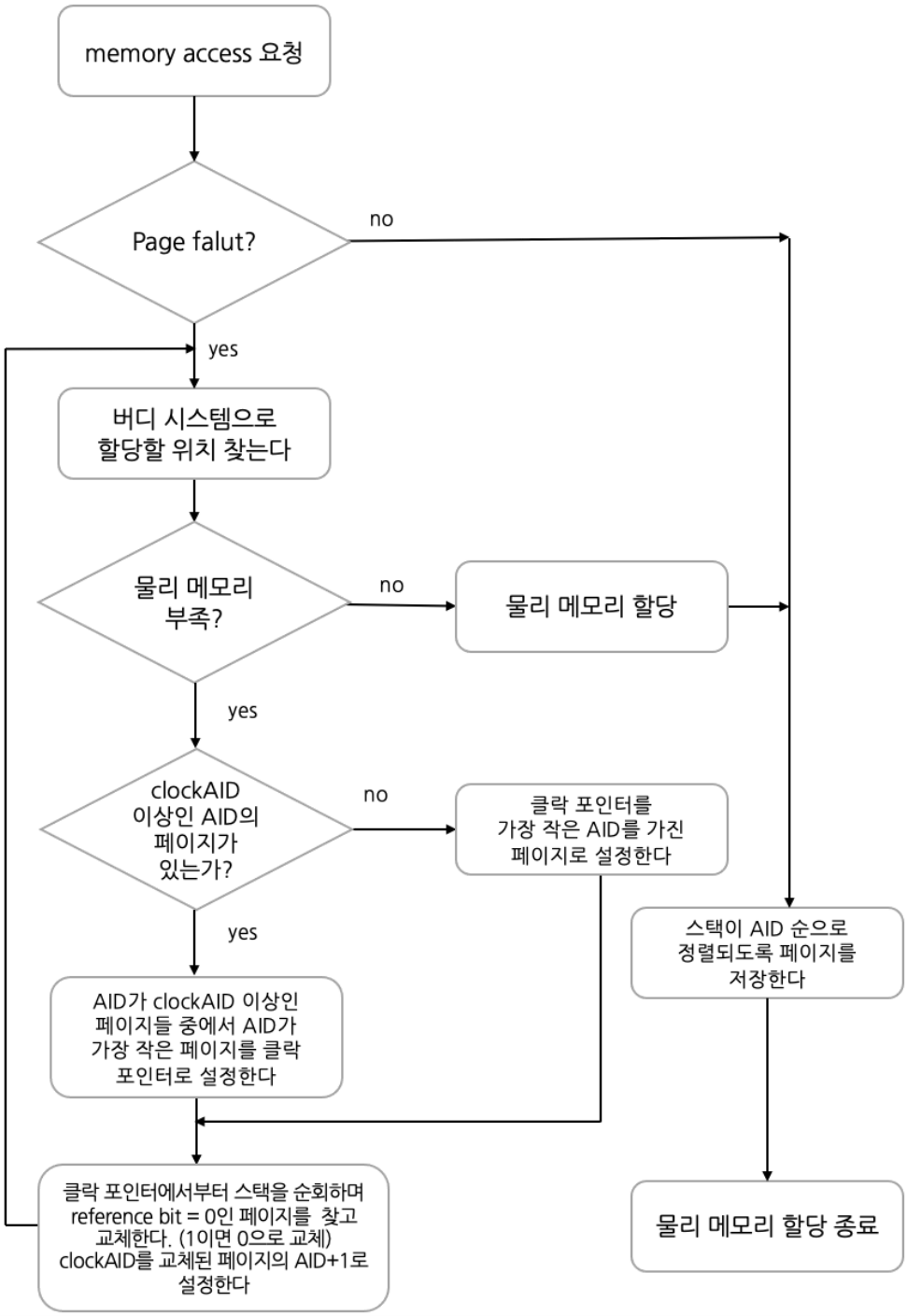
```
void Process::refreshRefByte() {
    /**
     * Refresh reference byte
     */
    unordered_map<int, PTE *>::iterator iter;
    for(iter=pageTable.begin();iter!=pageTable.end();iter++) {
        PTE *pte = iter->second;
        pte->refByte >>= 1;
        pte->refByte[TIME_INTERVAL-1] = pte->refBit;
        pte->refBit = 0;
    }
}
```

프로세스의 페이지 테이블을 순회하며 reference byte 를 1 bit 만큼 shift right 해주고, reference byte 의 MSB 를 페이지의 reference bit 으로 교체해주었다. a 에서와 마찬가지로 물리 메모리에 존재하는 페이지들을 담고 있는 스택을 활용하였다. 다만 이 알고리즘에서는 스택의 용도로 쓰기 보다는 리스트의 용도로 사용하였다. 아래는 순서도이다.



c. Clock(Second-chance)

Clock 알고리즘을 사용하기 위해서는 가장 최근에 교체되어 나간 페이지의 allocation id 를 저장하는 변수 int clockAID 를 사용하였다. 또한 스택에 새로운 페이지를 저장할 때 allocation id 순서대로 정렬되도록 하였다. 아래는 도식도이다.



2. 개발 환경 명시

A. uname -a 실행결과

```
jaeunjung@ubuntu:~$ uname -a
Linux ubuntu 5.10.19-2017122063 #1 SMP Thu Apr 8 20:35:05 PDT 2021 x86_64 x86_64
 x86_64 GNU/Linux
```

과제 1 에서 컴파일한 커널을 사용하였다.

B. 컴파일러 정보

```
jaeunjung@ubuntu:~$ g++ --version
g++ (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

jaeunjung@ubuntu:~$ gcc --version
gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

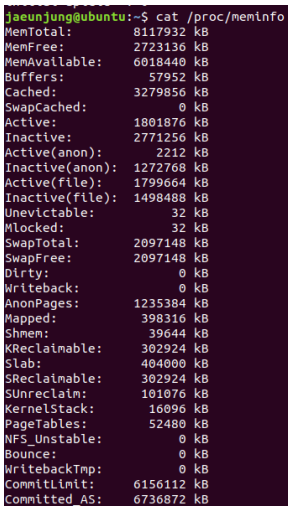
컴파일러는 g++ 7.5.0 을 사용하였다.

C. CPU 정보

```
jaeunjung@ubuntu:~$ grep -c processor /proc/cpuinfo
2
jaeunjung@ubuntu:~$ head -n 15 /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 126
model name      : Intel(R) Core(TM) i5-1038NG7 CPU @ 2.00GHz
stepping        : 5
microcode       : 0x96
cpu MHz         : 1996.800
cache size      : 6144 KB
physical id     : 0
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
```

CPU 는 2 개를 할당했다.

D. 메모리 정보



```
jaeunjung@ubuntu:~$ cat /proc/meminfo
MemTotal:        8117932 kB
MemFree:         2723136 kB
MemAvailable:    6018440 kB
Buffers:           57952 kB
Cached:          3279856 kB
SwapCached:            0 kB
Active:          1801876 kB
Inactive:        2771256 kB
Active(anon):       2212 kB
Inactive(anon):  1272768 kB
Active(file):    1799664 kB
Inactive(file):  1498488 kB
Unevictable:          32 kB
Mlocked:              32 kB
SwapTotal:       2097148 kB
SwapFree:        2097148 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:       1235384 kB
Mapped:           398316 kB
Shmem:             39644 kB
KReclaimable:     302924 kB
Slab:             404000 kB
SReclaimable:     302924 kB
SUnreclaim:       101076 kB
KernelStack:       16096 kB
PageTables:        52480 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     6156112 kB
Committed_AS:    6736872 kB
VmallocTotal:   34359738367 kB
VmallocUsed:       42488 kB
VmallocChunk:          0 kB
Percpu:            47104 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
FileHugePages:         0 kB
FilePmdMapped:         0 kB
CmaTotal:              0 kB
CmaFree:               0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:               0 kB
DirectMap4k:      345920 kB
DirectMap2M:     6993920 kB
DirectMap1G:     3145728 kB
```

위는 사용한 메모리 정보이다.

3. 결과 토의

A. 스케줄링 알고리즘

Multilevel scheduling 알고리즘의 한계를 분석하기 위해 두 가지 인풋을 만들어주었다. 각각의 인풋은 다음과 같다.

input

report > sched_input1 > input

```
6 2048 1024 32
1 program1 6
5 program2 5
10 INPUT 0
10 program3 2
15 INPUT 1
20 INPUT 2
```

input

report > sched_input2 > input

```
6 2048 1024 32
1 program1 0
5 program2 1
10 INPUT 0
10 program3 5
15 INPUT 1
20 INPUT 2
```

```
[30 Cycle] Scheduled Process: 2 program3 (priority 2)
Running Process: Process#2(2) running code program3 line 9(op 1, arg 0)
RunQueue 0: Empty
RunQueue 1: Empty
RunQueue 2: Empty
RunQueue 3: Empty
RunQueue 4: Empty
RunQueue 5: Empty
RunQueue 6: Empty
RunQueue 7: Empty
RunQueue 8: Empty
RunQueue 9: Empty
SleepList: Empty
IOWait List: Empty
```

```
[33 Cycle] Scheduled Process: 2 program3 (priority 5)
Running Process: Process#2(5) running code program3 line 9(op 1, arg 0)
RunQueue 0: Empty
RunQueue 1: Empty
RunQueue 2: Empty
RunQueue 3: Empty
RunQueue 4: Empty
RunQueue 5: Empty
RunQueue 6: Empty
RunQueue 7: Empty
RunQueue 8: Empty
RunQueue 9: Empty
SleepList: Empty
IOWait List: Empty
```

Program1, program2, program3 는 모두 8 개의 CPU instruction 과 1 개의 I/O instruction 으로 구성되어 있다. 실행결과, 첫 번째 인풋을 실행하는데 총 30 사이클이 소모되었다. **두 번째 인풋은 33 사이클이 소모되었고 전자에 비해 CPU utilization 이 낮아졌음을 확인할 수 있었다.**

첫 번째 인풋에서는 우선순위가 낮은 프로세스에게 높은 우선순위를 가진 프로세스가 들어오기 전까지 명령을 실행할 시간이 주어지는데 비해 두 번째 인풋에서는 계속 기다려야 한다. 구체적으로, program1 과 program2 가 FCFS(FIFO) 알고리즘에 의해 스케줄링 되므로 우선순위가 낮은 program3 는 두 프로세스가 스스로 block 되거나 종료되기를 기다리는 수 밖에 없다. 즉, **이 스케줄링 알고리즘은 CPU utilization 이 프로세스의 들어오는 순서와 우선순위에 의존적이라는 한계점이 존재한다.** 아래는 프로세스별로 waiting 시간을 나타낸 표이다. (Sleep list, I/O wait 은 고려하지 않았다.)

	Input1	Input2
program1	3 cycles	0 cycles
program2	6 cycles	2 cycles
program3	0 cycles	6 cycles
Average waiting time	3 cycles	2.67 cycles

B. 페이지 교체 알고리즘 비교

페이지 교체 알고리즘을 비교하기 위해 Python 언어를 사용하여 랜덤하게 4 가지의 인풋을 만들어주었다. 각 인풋은 모두 10 개의 프로그램으로 이루어져 있으며, 프로세스가 들어오는 순서와 우선순위는 모두 다음과 같이 동일하며 가상 메모리는 모두 64 개의 페이지를 가진다. 다만 물리 메모리 사이즈는 64 개의 페이지를 가질 때, 32 개의 페이지를 가질 때로 나눠서 비교했다.

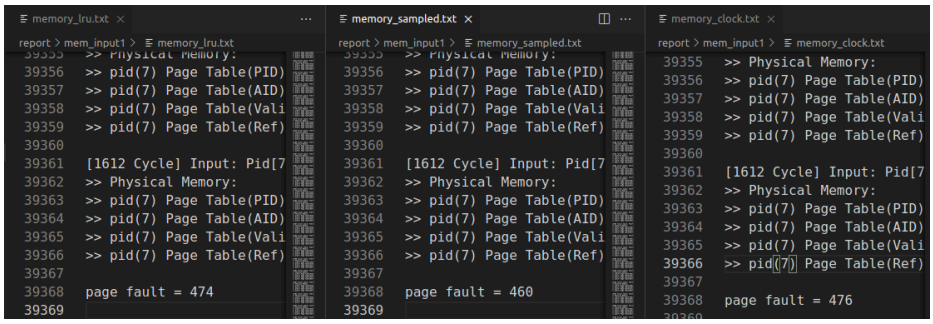
```
10 2048 1024 32
1 program1 3
5 program2 4
10 program3 5
15 program4 6
20 program5 1
25 program6 0
30 program7 2
35 program8 9
40 program9 8
40 program10 7
```

```
10 2048 512 32
1 program1 3
5 program2 4
10 program3 5
15 program4 6
20 program5 1
25 program6 0
30 program7 2
35 program8 9
40 program9 8
40 program10 7
```

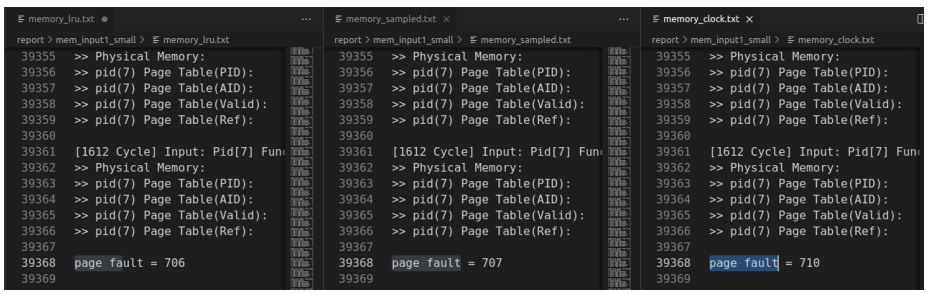
a. 코드의 temporal locality 가 없는 경우

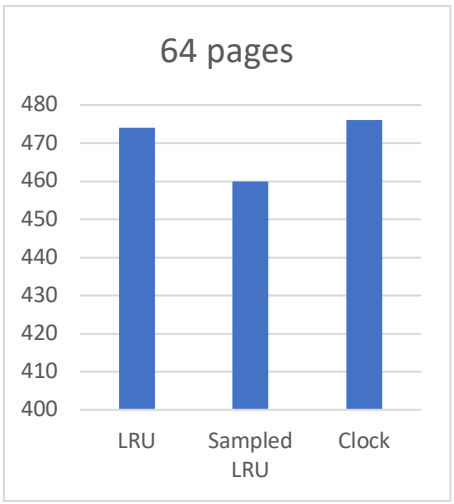
각 프로그램의 시작에서는 [16, 8, 8, 4, 4, 4, 4, 4, 4, 2, 2, 2, 1, 1]로 구성된 배열을 무작위로 섞은 다음 배열의 원소만큼 가상 메모리를 할당했다. 이후로는 메모리가 모두 해제될 때까지 80%/10%/10% 비율로 랜덤하게 페이지를 골라 memory access, memory release, sleep(5 cycles) 명령어를 수행한다. Page fault 가 발생한 결과는 다음과 같다.

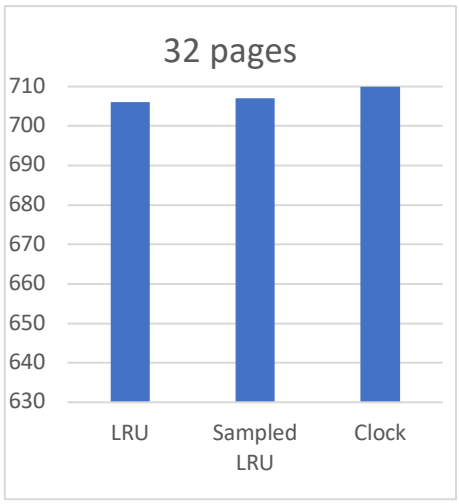
- # of pages = 64



- # of pages = 32





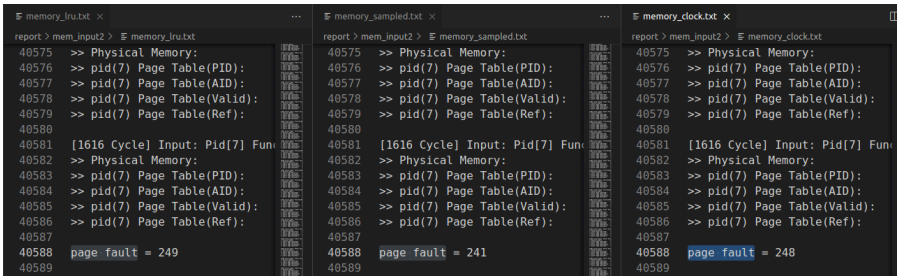


세 가지 알고리즘 모두 대략적으로 비슷한 결과가 나왔음을 알 수 있다. 이는 **sampled LRU, clock 알고리즘이 LRU 알고리즘을 근사하는 알고리즘**이기 때문이다. **물리 메모리가 부족할 때 세 가지 알고리즘의 성능이 거의 비슷했는데, 이는 page fault 가 더 자주 일어났기 때문에 sampled LRU, clock 알고리즘이 LRU 알고리즘을 더 잘 근사했으며 thrashing 이 심화되어 page fault 가 더 자주 발생했음을 볼 수 있다.**

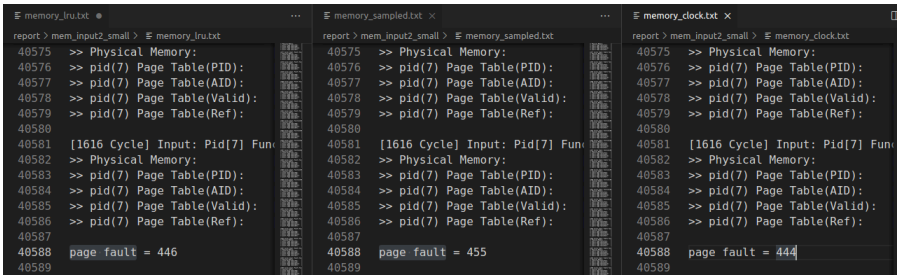
b. 코드의 temporal locality 가 있는 경우

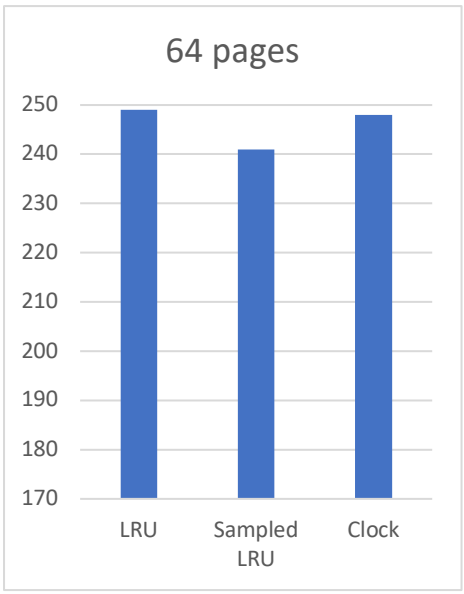
a 와 마찬가지로 각 프로그램의 시작에서는 [16, 8, 8, 4, 4, 4, 4, 4, 4, 2, 2, 2, 1, 1]로 구성된 배열을 무작위로 섞은 다음 배열의 원소만큼 가상 메모리를 할당했다. 이후로는 14 개의 전체 페이지를 **두 부분으로 나누어서** 각 부분에 대해 a 에서 한 것과 마찬가지로 메모리가 모두 해제될 때까지 80%/10%/10% 비율로 랜덤하게 페이지를 골라 memory access, memory release, sleep(5 cycles) 명령어를 수행했다. Page fault 가 발생한 결과는 다음과 같다.

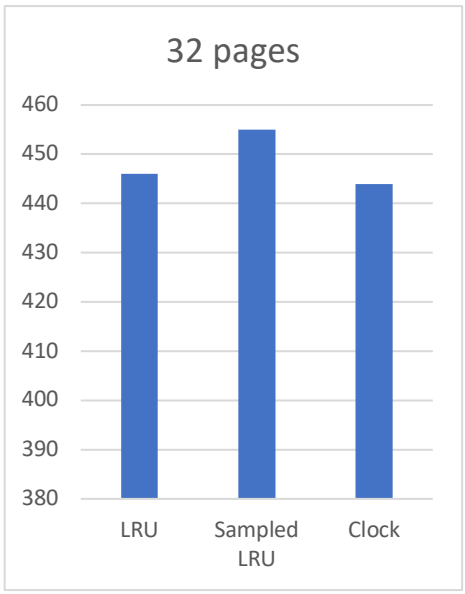
- # of pages = 64



- # of pages = 32







a 보다 page fault 수가 확연히 적어졌음을 알 수 있다. 물리 메모리가 32 개의 페이지로 구성되었지만 코드의 temporal locality 가 지켜졌을 때는 64 개의 페이지를 가진 물리 메모리가 temporal locality 가 지켜지지 않은 프로그램을 수행했을 때보다 좋은 결과를 보였다. **즉, 이 실험을 통해서 코드의 temporal locality 가 지켜지도록 프로그래밍을 하고 컴파일러를 최적화하는 것의 중요성을 알 수 있었다.**

4. 과제 수행 시 겪었던 어려움과 해결 방법

A. 초기 설계의 어려움

9 개의 헤더 파일과 cpp 파일을 작성할 만큼 코드의 양이 방대했기 때문에 초기 설계가 매우 중요했다. 초기 설계를 잘못한 까닭에 디버깅에 많은 시간을 소모했고 결국 처음부터 설계를 하는 결과를 초래했다. 프로그램을 작성할 때 초기 설계의 중요성을 알 수 있었다.

B. 여러 예외 케이스

동시에 여러 프로세스가 생성되거나, sleep 이 시작할 때 time quantum 이 끝나는 등 여러 예외 케이스를 고려하면서 프로그래밍을 하는 것에 어려움을 겪었다. 이를 해결하기 위해 다양하게 랜덤한 인풋을 생성해가며 디버깅하는 과정을 거쳤다.