REPORT

보고서 작성 서약서

1. 나는 타학생의 보고서를 복사(Copy)하지 않았습니다.
2. 나는 타학생의 보고서를 인터넷에서 다운로드 하여 대체하지 않았습니다.
3. 나는 타인에게 보고서 제출 전에 보고서를 보여주지 않았습니다.
4. 보고서 제출 기한을 준수하였습니다.

나는 보고서 작성시 위법 행위를 하지 않고,
성.균.인으로서 나의 명예를 지킬 것을 약속합니다.

과 목 : 운영체제(SWE3004_41)
과 제 명 : Virtual memory management 기법 구현
담당교수 : 엄영익 교수님
학 과 : 시스템경영공학과
학 년 : 3
학 번 : 2016314726
이 름 : 정영준
제 출 일 : 2020.12.03

1) 개발환경

OS: Ubuntu 20.04.1 LTS

Python: 3.8.2

Python 사용 사유: input 으로 주어지는 reference string 의 길이와 다른 여러 정보를 저장하기위해 list 를 만들어 사용하려고 계획하였습니다. 동적할당이 빈번하고 더 편리한 리스트의 built-in 함수가 있는 python 이 더 적합하다고 생각하였고 시스템 경영공학과에서 복수전공을 하여 더 익숙한 언어였기 때문에 개발언어로 python 을 사용하였습니다.

2) 컴파일 및 실행방법

```
C:₩Users₩JUNE₩Desktop₩os>python OS2020-2_2016314726_정영준_P3.py FIFO
N:  7
M:  4
W:  3
K:  14
ref:  [0, 1, 2, 6, 1, 4, 5, 1, 2, 1, 4, 5, 6, 4, 5]
```

Window 와 Linux ubuntu 환경에서 모두 성공적으로 실행하였습니다.

위와 같이 python OS2020-2_2016314726_정영준_P3.py (algorithm name)을 입력하여 실행이 가능합니다.

가능한 algorithm name 은 MIN, FIFO, LRU, LFU, Clock, WS 가 있습니다.

input.txt 파일에 저장된 정보에 따라 주어진 virtual memory replacement algorithm 으로 계산을 마치고 결과와 과정을 log.txt 파일에 작성하는 프로그램입니다.

3) 설계/구현

```python
def main():
    vmm = VirtualMemoryManagement('input.txt','log.txt')
    vmm.check_argv()
    vmm.operate(sys.argv[1])



if __name__ == "__main__":
    main()
```

위와 같이 main 함수는 프로그램의 실행과 함께 실행되며 VirtualMemoryManagement 라는

Class 를 사용하여 vmm 이라는 변수에 객체를 할당합니다. Input.txt 와 log.txt 는 각각 주어진 입력을 나타내는 파일과 출력 로그를 기록하는 파일명입니다. 다음 check_argv 함수와 operate 함수는 VirtualMemoryManagement 클래스 내부의 함수로 아래에서 설명하겠습니다.

```
class VirtualMemoryManagement():
    def __init__(self,memory_info_file,log_file):
        self.file = open(memory_info_file,'r')
        self.N, self.M, self.W, self.K = list(map(int,self.file.readline().split()))
        self.ref = [0]+list(map(int,self.file.readline().split()))
        self.log_path = log_file
        self.log = open(log_file,'w')
        self.page_load_time= dict()
        self.pointer = 0


    def check_argv(self):
        print('N: ',self.N)
        print('M: ',self.M)
        print('W: ',self.W)
        print('K: ',self.K)
        print('ref: ',self.ref)
```

VirtualMemoryManagement 클래스가 생성자로 객체생성을 호출받으면 먼저 __init__ 함수를 통해 내부 변수를 초기화 시킵니다. self.file 에는 input.txt 파일을 열고 이를 할당합니다. self.N, self.M, self.W, self.K 는 각각 input.txt 파일에서 읽어낸 process 의 page 개수, 할당 page frame 개수, window size, page reference string 의 길이를 나타냅니다. self.ref 는 input.txt 파일의 2 번째 줄에 주어진 page reference string 을 나타냅니다. 앞에 list 의 원소로 0 이 추가되는데 이는 시간과 index 를 일치시키기 위해 index 0 을 더미값으로 채운 것 입니다. self.log 는 작성할 로그 파일을 말하며 self.page_load_time 은 FIFO 알고리즘에서 로드된 시각을 저장하기 위한 list 입니다. Self.pointer 는 Clock 알고리즘에서 pointer 의 위치를 저장한 변수입니다.

Check_argv 함수는 input.txt 파일을 올바르게 읽어왔는지 인식하기 위해 각 변수를 출력해주는 함수입니다.

```
def operate(self,algorithm):
    time = 1
    total_pf = 0
    self.memory_state = [-1]*self.M
    if algorithm == 'Clock':
        self.log.write('|{:^8}|{:^5}|{:^16}| {}\n'.format('Time','P.f','Pclock  Qclock','Memory state  [page, reference bit]'))
        self.memory_state = [[p,1] for p in range(self.M)]
    elif algorithm == 'WS':
        self.log.write('|{:^8}|{:^5}|{:^6}|{:^6}|{:^17}|{}\n'.format('Time','P.f','P.ws','Q.ws','Num.alloc.frame','Memory state'))
        self.memory_state = [[p,-1] for p in range(self.N)]
    else:
        self.log.write('|{:^8}|{:^5}|{:^8}| {}\n'.format('Time','P.f','Victim','Memory state'))
    while time <= self.K:
        victim, page_fault = self.replace_process(algorithm,time)
        if page_fault:
            total_pf += 1
        self.write_log(time,page_fault,victim,algorithm)
        time += 1
    self.log.close()
    with open(self.log_path,'r') as f:
        entire_log = f.read()
    with open(self.log_path,'w') as f:
        f.write('Total Page Fault: {:<6}\n\n{}'.format(total_pf,entire_log))
```

다음 함수는 operate 함수입니다. Algorithm 을 추가로 인자로 받아 실행되며 먼저 시간을 나타내는 time 과 page fault 횟수를 저장하는 total_pf 변수를 초기화시킵니다. Memory state 는 self.M 개의 -1 로 이루어진 list 로 -1 은 현재 할당된 page 가 없다는 것을 의미합니다.

다음으로 알고리즘에 따라 log 의 형태가 달라지므로 if 문을 사용하여 이를 처리하였습니다. 그리고 time 이 self.K 이하일 경우 반복해서 self.replace_process 함수를 실행하며 log 를 작성하고 time 을 1 씩 증가시킵니다. 시간이 지날 때 마다 로그를 작성하고 time 을 증가시켜 다음 시간으로 이동하여 이를 반복하는 부분입니다.

마지막으로 page fault 횟수를 log 최상단에 기록하고 함수를 끝마칩니다.

```
def replace_process(self,algorithm,time):
    page_fault = False
    victim = ''
    if algorithm == 'MIN':
        if self.ref[time] not in self.memory_state:
            try:
                self.memory_state[self.memory_state.index(-1)] = self.ref[time]
                page_fault = True
            except:
                ref_period = [0]*self.M
                for i,page in enumerate(self.memory_state):
                    if page in self.ref[time:]:
                        ref_period[i] = self.ref[time:].index(page)
                    else:
                        ref_period[i] = 1000000
                mx = -1
                mx_page = -1
                for j,period in enumerate(ref_period):
                    if period > mx:
                        mx = period
                        mx_page = self.memory_state[j]
                    elif period == mx:
                        mx_page = self.memory_state[j] if self.memory_state[j] < mx_page else mx_page
                victim = mx_page
                self.memory_state[self.memory_state.index(mx_page)] = self.ref[time]
                page_fault = True
```

가상 메모리 교체의 적용이 실질적으로 매 시각 이루어지게 하는 replace-process 함수입니다. Page fault 변수는 Bool 자료형으로 True 또는 False 로 page fault 가 일어났는지 기록합니다. Victim 은 이번 시간에 교체 당한 page 의 number 를 기록하기 위한 변수로 공백으로 초기설정이 되어있습니다. MIN 알고리즘의 구현에 대해 설명드리겠습니다. 먼저 가장 처음 if 문으로 현 시점에 reference 되는 page 가 memory state 에 존재하는지 검사합니다. 만약 존재한다면 page fault 는 그대로 False, victim 은 그대로 공백으로 return 이 이루어지고 이는 log 에 기록됩니다. 그렇지 않다면 새로운 페이지의 삽입이 필요하다는 것으로 다음 과정으로 넘어갑니다.

self.memory_state[self.memory_state.index(-1)]=self.ref[time]를 실행하는데 이는 memory state 에 아직 할당되지 않은 자리가 남아있을 경우 이를 현재 시각 reference page 에 할당해준다는 것을 말합니다. Error 가 발생하지 않으면 page fault 는 True 가 되고 victim 은 그대로 공백으로 return 이

되고 함수가 끝나고 log 가 기록됩니다. 하지만 남은 memory 가 없다면 page replacement 가 필요하다는 것으로 다음 줄로 넘어갑니다.

여기부터 MIN 알고리즘의 적용이 시작됩니다. self.ref[:time]은 미래의 page reference 를 나타내는 list 로 python list 의 index method 를 사용하여 memory state 의 각 page 의 다음 reference 까지 걸리는 시간을 ref_period 에 저장합니다. 만약 다음 reference 가 없는 page 는 ref_period 를 1000000 으로 설정하여 self.K 의 최대값인 100000 보다 크게하여 victim 이 되도록 설정합니다. ref_period 가 가장 긴 page 는 victim 으로 설정되어 교체됩니다. 2 개 이상의 page 의 ref_period 가 같다면 page number 가 작은 page 가 victim 으로 설정됩니다.

```
elif algorithm == 'LRU':
    if self.ref[time] not in self.memory_state:
        try:
            self.memory_state[self.memory_state.index(-1)] = self.ref[time]
            page_fault = True
        except:
            ref_period = [0]*self.M
            reversed_past_ref = self.ref[:time].copy()
            reversed_past_ref.reverse()
            for i,page in enumerate(self.memory_state):
                ref_period[i] = reversed_past_ref.index(page)
            mx = -1
            mx_page = -1
            for j,period in enumerate(ref_period):
                if period > mx:
                    mx = period
                    mx_page = self.memory_state[j]
            victim = mx_page
            self.memory_state[self.memory_state.index(mx_page)] = self.ref[time]
            page_fault = True
```

다음으로 LRU 알고리즘입니다. try 부분은 위 MIN 알고리즘에서 설명한 부분과 동일합니다. ref_period 는 현 시점에서 가장 최근 reference 된 시점까지의 기간을 기록하기 위한 list 입니다.

reversed_past_ref 는 시작부터 현 시점까지 page reference string 을 reverse 시킨 list 입니다. 이를 python list 의 index method 를 사용하여 ref_period 에 그 값을 저장합니다. 그리고 마지막 reference time 이 가장 오래된 page 가 victim 으로 선정됩니다.

```
elif algorithm == 'FIFO':
    if self.ref[time] not in self.memory_state:
        self.page_load_time[self.ref[time]] = time
        try:
            self.memory_state[self.memory_state.index(-1)] = self.ref[time]
            page_fault = True
        except:
            loaded_page_dict = {k:v for k,v in self.page_load_time.items() if k in self.memory_state}
            replace_page = min(loaded_page_dict.items(),key= lambda x: x[1])[0]
            victim = replace_page
            self.memory_state[self.memory_state.index(replace_page)] = self.ref[time]
            page_fault = True
```

FIFO 알고리즘은 코드가 짧습니다. 먼저 try 부분에 page 가 load 된 time 을 기록하는 page_load_time dictionary 에 값을 할당하는 부분이 추가됩니다. 처음으로 memory state 에 존재하지 않는 page 가 reference 를 받을 경우 그 시각을 {page number : time} 형식으로 저장합니다. loaded_page_dict 는 page_load_time 에서 현재 memory state 에 존재하는 page 의 정보만을 추출한 subset 입니다. 그리고 가장 load 된 시각이 이른 page 를 victim 으로 선정합니다.

```
elif algorithm == 'LFU':
    if self.ref[time] not in self.memory_state:
        try:
            self.memory_state[self.memory_state.index(-1)] = self.ref[time]
            page_fault = True
        except:
            min_ref = 1000000
            min_ref_page = -1
            for page in self.memory_state:
                if min_ref > self.ref[:time].count(page):
                    min_ref = self.ref[:time].count(page)
                    min_ref_page = page
                elif min_ref == self.ref[:time].count(page):
                    if self.ref[:time][::-1].index(page) > self.ref[:time][::-1].index(min_ref_page):
                        min_ref_page = page


            victim = min_ref_page
            self.memory_state[self.memory_state.index(min_ref_page)] = self.ref[time]
            page_fault = True
```

LFU 는 지금까지 memory state 에 존재하는 page 들의 reference 횟수를 비교하여 victim 을 선정합니다. self.ref[:time].count(page)는 time 시점에 page 가 지금까지 reference 된 횟수를 말합니다. 이를 비교하여 가장 적게 reference 가 된 page 를 victim 으로 선정합니다. 횟수가 같을경우 elif 문에 해당하는 부분으로 LRU 알고리즘을 통해 victim 을 선정합니다.

```
elif algorithm == 'Clock':
    if self.ref[time] not in [x[0] for x in self.memory_state]:
        while True:
            if self.pointer == self.M:
                self.pointer = 0
            if self.memory_state[self.pointer][1] == 0:
                victim = [self.ref[time],self.memory_state[self.pointer][0]]
                self.memory_state[self.pointer] = [self.ref[time],1]
                page_fault = True
                self.pointer += 1
                break
            else:
                self.memory_state[self.pointer][1] = 0
                self.pointer += 1
    else:
        self.memory_state[[x[0] for x in self.memory_state].index(self.ref[time])][1] = 1
```

다음은 Clock 알고리즘입니다. Memory state 가 지금까지 알고리즘과 다르게 page number 와 reference bit 2 개의 정보를 포함하고 있습니다. 먼저 self.ref[time] not in [x[0] for x in self.memory_state 를 조건문으로 사용하여 현재 참조하는 page 가 memory state 에 존재하지 않는지 확인합니다. 존재한다면 else 문이 실행되어 현재 참조하는 page 의 reference bit 가 1 이 됩니다. 그렇지 않다면 while 문이 시작됩니다.

먼저 pointer 가 주어진 범위를 벗어났다면 처음으로 돌아갈 수 있도록 합니다. 다음으로 만약 현재 pointer 가 가리키는 page 의 reference bit 가 0 이면 그 page 가 victim 으로 선정되고 page 의 교체가 이루어지고 pointer 가 한 칸 이동합니다. Reference bit 가 1 이라면 reference bit 를 0 으로 바꾸고 pointer 를 다음 page 로 넘깁니다.

```
elif algorithm == 'WS':
    victim = ['']*2
    for i,m in enumerate(self.memory_state):
        if m[1] >= 0:
            self.memory_state[i][1] += 1

    for i,mem in enumerate(self.memory_state):
        if (mem[1] == self.W+1) and (mem[0] != self.ref[time]):
            mem[1] = -1
            victim[1] = mem[0]


    if self.memory_state[[x[0] for x in self.memory_state].index(self.ref[time])][1] == -1:
        page_fault = True
        self.memory_state[[x[0] for x in self.memory_state].index(self.ref[time])][1] = 0
        victim[0] = self.ref[time]
    else:
        self.memory_state[[x[0] for x in self.memory_state].index(self.ref[time])][1] = 0
```

마지막으로 Working set 을 사용하는 알고리즘입니다. self.M 은 사용되지 않고 self.W 가 대신 사용됩니다. 여기에서 memory state 는 [page number, ws_state]으로 초기화됩니다. ws_state 가 -1 일 경우 현재 working set 에 포함되어 있지 않다는 것을 의미하고 0 이상일 경우 이는

reference 되고나서 지난 시간을 의미합니다. 먼저 한 시점의 교체 알고리즘이 시작될 때 ws_state 가 -1 이 아닌 (현재 working set 에 포함된) page 의 ws_state 는 1 씩 증가합니다. 이는 reference 되고 나서 시간이 1 만큼 더 지났기 때문입니다. 다음으로 모든 page 의 ws_state 를 조사합니다. 만약 self.W+1 과 같을 경우 이는 Qws 로 선정이 됩니다. 하지만 현재 reference 하는 page 가 만약 이번에 Qws 로 선정이 된 page 일 경우 Qws 는 공백으로 유지됩니다.

```
if self.memory_state[[x[0] for x in self.memory_state].index(self.ref[time])][
1] == -1
```

위 코드의 조건문은 현재 참조되는 page 가 현재 working set 의 page 가 아닐경우를 의미하며 이때 page fault 는 True 가 되고 현재 참조하는 page 는 Pws 로 정해지고 ws_state 을 0 으로 바꿔줍니다.

위 코드의 조건문을 만족하지 못할 경우 Pws 와 page fault 는 공백과 False 로 변하지 않고 해당 page 의 ws_state 만 0 으로 변하게 됩니다.

```
def write_log(self,time,page_fault,victim,algorithm):
    memory = self.memory_state.copy()
    while -1 in memory:
        memory.remove(-1)
    if page_fault:
        pf = 'F'
    else:
        pf = ' '
    if algorithm == 'Clock':
        self.log.write('|{:^8}|{:^5}|{:^16}| {}\n'.format(time,pf,(' '*7).join(list(map(str,victim)))+'',' '.join(list(map(str,memory)))))
    elif algorithm == 'WS':
        num_alloc = self.N
        memory = [x[:] for x in self.memory_state]
        for i,page in enumerate(memory):
            if page[1] < 0:
                memory[i][0] = ' '
                num_alloc -= 1

        self.log.write('|{:^8}|{:^5}|{:^6}|{:^6}|{:^17}| {}\n'.format(time,pf,victim[0],victim[1],num_alloc,' '.join(list(map(str,[m[0] for m in memory])))))
    else:
        self.log.write('|{:^8}|{:^5}|{:^8}| {}\n'.format(time,pf,victim,' '.join(list(map(str,memory)))))
```

마지막으로 매 시점 log 를 작성하는 write_log 함수입니다.

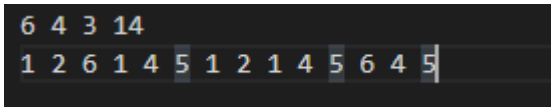
Clock, WS 알고리즘의 경우 추가로 Pclock Qclock / Pws Qws 현재 할당되고있는 페이지의 수가 log 에 기록되기 때문에 따로 처리를 하였습니다. 그 외 알고리즘은 기본적인 출력 형태를 같게 유지합니다.

4) 설정가설

먼저 MIN 알고리즘에서 현재 memory state 에 존재하는 2 개 이상의 page 가 앞으로 reference 가 되지 않아 Tie 가 형성되면 page number 가 작은 page 를 victim 이 됩니다.

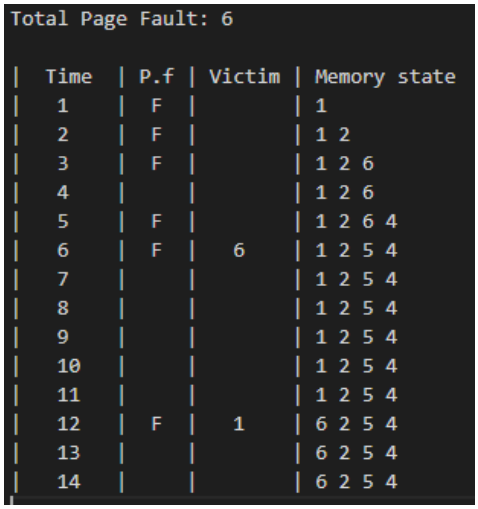
입력파일의 경우 과제 공지에 명시된 내용을 그대로 따른다는 가정하에 코드를 작성하였습니다. WS 알고리즘에서 window size 를 벗어남과 동시에 그 시각의 reference 가 될 경우 이 page 를 memory state 에서 제거하고 그대로 다시 할당하는 과정은 비효율적이므로 page fault 가 일어나지 않았다고 가정하고 현재 memory state 를 그대로 유지하였습니다.

5) 실행 결과 출력물

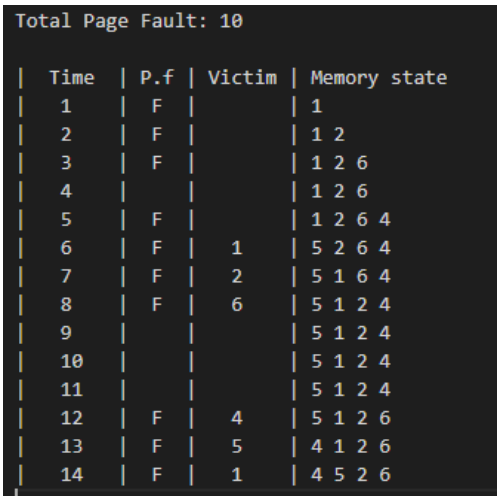


Input txt 는 위와 같이 강의 ppt 에 MIN, FIFO, LRU, LFU 알고리즘에 사용되는 input 을 그대로 가져왔습니다.

MIN 알고리즘 실행 결과



FIFO 알고리즘 실행 결과



LRU 알고리즘 실행 결과

```
Total Page Fault: 7

|  Time  | P.f | Victim | Memory state
|   1    |  F  |        | 1
|   2    |  F  |        | 1 2
|   3    |  F  |        | 1 2 6
|   4    |     |        | 1 2 6
|   5    |  F  |        | 1 2 6 4
|   6    |  F  |   2    | 1 5 6 4
|   7    |     |        | 1 5 6 4
|   8    |  F  |   6    | 1 5 2 4
|   9    |     |        | 1 5 2 4
|   10   |     |        | 1 5 2 4
|   11   |     |        | 1 5 2 4
|   12   |  F  |   2    | 1 5 6 4
|   13   |     |        | 1 5 6 4
|   14   |     |        | 1 5 6 4
```

LFU 알고리즘 실행 결과

```
Total Page Fault: 7

|  Time  | P.f | Victim | Memory state
|   1    |  F  |        | 1
|   2    |  F  |        | 1 2
|   3    |  F  |        | 1 2 6
|   4    |     |        | 1 2 6
|   5    |  F  |        | 1 2 6 4
|   6    |  F  |   2    | 1 5 6 4
|   7    |     |        | 1 5 6 4
|   8    |  F  |   6    | 1 5 2 4
|   9    |     |        | 1 5 2 4
|   10   |     |        | 1 5 2 4
|   11   |     |        | 1 5 2 4
|   12   |  F  |   2    | 1 5 6 4
|   13   |     |        | 1 5 6 4
|   14   |     |        | 1 5 6 4
```

```
6 3 3 15
0 1 2 3 2 3 4 5 4 1 3 4 3 4 5
```

강의 PPT 의 예시를 input 으로 하면 두 알고리즘이 동일한 log 가 나와 차이를 보기위해 다른 input.txt 를 작성하고 이를 input 으로 다시 log 를 작성하였습니다.

```
Total Page Fault: 9

|  Time  | P.f | Victim | Memory state
|   1    |  F  |        | 0
|   2    |  F  |        | 0 1
|   3    |  F  |        | 0 1 2
|   4    |  F  |   0    | 3 1 2
|   5    |     |        | 3 1 2
|   6    |     |        | 3 1 2
|   7    |  F  |   1    | 3 4 2
|   8    |  F  |   2    | 3 4 5
|   9    |     |        | 3 4 5
|   10   |  F  |   3    | 1 4 5
|   11   |  F  |   5    | 1 4 3
|   12   |     |        | 1 4 3
|   13   |     |        | 1 4 3
|   14   |     |        | 1 4 3
|   15   |  F  |   1    | 5 4 3
```

```
Total Page Fault: 9

|  Time  | P.f | Victim | Memory state
|   1    |  F  |        | 0
|   2    |  F  |        | 0 1
|   3    |  F  |        | 0 1 2
|   4    |  F  |   1    | 0 3 2
|   5    |     |        | 0 3 2
|   6    |     |        | 0 3 2
|   7    |  F  |   0    | 4 3 2
|   8    |  F  |   4    | 5 3 2
|   9    |  F  |   5    | 4 3 2
|   10   |  F  |   2    | 4 3 1
|   11   |     |        | 4 3 1
|   12   |     |        | 4 3 1
|   13   |     |        | 4 3 1
|   14   |     |        | 4 3 1
|   15   |  F  |   1    | 4 3 5
```

```
5 4 3 10
2 0 3 1 4 1 0 1 2 3
```

Clock 알고리즘의 경우 강의자료의 예시를 a 부터 e 까지의 page number 를 0 부터 4 까지로 수정하여 그대로 input.txt 를 만들었습니다.

```
5 0 3 13
4 3 0 2 2 3 1 2 4 2 4 0 3
```

Working set 알고리즘은 강의자료의 예시를 이번 과제의 입력 포맷의 가정에 따라 시작 시간을 -2 에서 0 으로 수정하고 초기 memory state 를 비어있도록 수정하고 이에 따라 input 을 수정하여 새로 만들었습니다.

Clock 알고리즘 실행 결과

```
Total Page Fault: 4

|  Time  | P.f | Pclock  Qclock | Memory state  [page, reference bit]
|   1    |     |                | [0, 1] [1, 1] [2, 1] [3, 1]
|   2    |     |                | [0, 1] [1, 1] [2, 1] [3, 1]
|   3    |     |                | [0, 1] [1, 1] [2, 1] [3, 1]
|   4    |     |                | [0, 1] [1, 1] [2, 1] [3, 1]
|   5    |  F  |   4       0    | [4, 1] [1, 0] [2, 0] [3, 0]
|   6    |     |                | [4, 1] [1, 1] [2, 0] [3, 0]
|   7    |  F  |   0       2    | [4, 1] [1, 0] [0, 1] [3, 0]
|   8    |     |                | [4, 1] [1, 1] [0, 1] [3, 0]
|   9    |  F  |   2       3    | [4, 1] [1, 1] [0, 1] [2, 1]
|   10   |  F  |   3       4    | [3, 1] [1, 0] [0, 0] [2, 0]
```

WS 알고리즘 실행 결과

```
Total Page Fault: 8

|  Time  | P.f | P.ws | Q.ws | Num.alloc.frame |Memory state
|   1    |  F  |  4   |      |        1        |          4
|   2    |  F  |  3   |      |        2        |        3 4
|   3    |  F  |  0   |      |        3        | 0      3 4
|   4    |  F  |  2   |      |        4        | 0    2 3 4
|   5    |     |      |  4   |        3        | 0    2 3
|   6    |     |      |      |        3        | 0    2 3
|   7    |  F  |  1   |  0   |        3        |    1 2 3
|   8    |     |      |      |        3        |    1 2 3
|   9    |  F  |  4   |      |        4        |    1 2 3 4
|   10   |     |      |  3   |        3        |    1 2   4
|   11   |     |      |  1   |        2        |      2   4
|   12   |  F  |  0   |      |        3        | 0    2   4
|   13   |  F  |  3   |      |        4        | 0    2 3 4
```

6) 출력물 설명

MIN, FIFO, LRU, LFU 알고리즘은 log 파일에 각 time 에 page fault 발생여부와 victim page 그리고 그 time 의 memory state 가 작성되었습니다. 같은 예시로 4 개의 알고리즘을 실행한 결과 이론대로 MIN 알고리즘이 가장 적은 total page fault 를 보였습니다. 강의자료에 있는 예시를 그대로 활용하여 강의자료의 결과와 비교를 해보았는데 모두 동일한 결과를 얻었습니다. 이론을 코드로 성공적으로 적용하였다고 할 수 있습니다.

다음으로 Clock 알고리즘은 log 파일에 victim 이 아닌 Pclock 과 Qclock 이 기록됩니다. Memory state 의 경우도 reference bit 의 상태를 함께 기록하기 위해 list 형태로 이를 구현하였습니다.

WS 알고리즘의 log 는 P.ws 와 Q.ws 를 기록하였고 추가로 현재 allocated frame 의 개수를 나타내는 Num.alloc.frame 을 추가로 기록하였습니다. Window size 를 바꿔가며 여러 예시를 input 으로 log 를 비교하였는데 window size 가 커지면 page fault 가 일어나는 횟수가 적어졌으나 Num.alloc.frame 이 평균적으로 증가하였습니다.