2020-1

REPORT

Operating System

Project-2

Virtual Memory Management 기법 구현

2020-1 Operating System

SWE3004_43

엄영익 교수님

문헌정보학과 2017310301 / 김승윤

목차

1. 개발 플랫폼과 컴파일 및 실행 방법 소개 3

1) MS Window 운영체제 3

2) 컴파일 및 실행 방법 4

2. 구현 코드 설명 5

1) Page 구조체와 main 함수 5

3) MIN algorithm 7

4) FIFO algorithm 9

5) LRU algorithm 11

6) LFU algorithm 13

7) Working set algorithm 15

3. 실행 결과 17

1. 개발 플랫폼과 컴파일 및 실행 방법 소개

1) MS Window 10 운영체제

운영체제 virtual memory management 프로젝트를 위해 사용한 개발 플랫폼은 마이크로소프트 윈도우 10 (Microsoft Windows 10, MS Window 10) 입니다. 마이크로소프트 윈도우는 마이크로소프트가 개발한 컴퓨터 운영체제로 개인용, 서버용 컴퓨터에 주로 사용되며 멀티태스킹과 GUI 환경을 제공합니다.

- MS Window 10 Virtual Memory

Virtual memory는 각 프로그램에 실제 메모리 주소가 아닌 가상 메모리 주소를 할당하는 메모리 관리 방식입니다. 이 방식은 멀티태스킹 운영체제에서 실제 주기억장치보다 큰 메모리 영역을 제공하는 방법으로 사용됩니다. 마이크로소프트 윈도우 10에서도 가상 메모리 기능을 제공하여 부족한 시스템 메모리를 보조합니다. 마이크로소프트 윈도우 10의 메모리 관리자는 실제 메모리 공간이 가득 차면 당장 사용하지 않을 페이지 일부를 하드 디스크의 페이징 파일(Pagefile.sys)로 옮겨 실제 메모리의 여유 공간을 확보합니다. 그러나 페이징 파일의 페이지가 필요할 때 다시 실제 메모리로 옮겨서 사용합니다. 즉, 프로그램의 페이지 사용 상황에 맞춰 실제 메모리와 가상 메모리 사이에서 데이터를 교환하는 방식으로 메모리 관리를 합니다. 페이징 파일은 기본적으로 윈도우에서 자동 관리되어 동적으로 사용량에 맞춰 크기를 조절할 수 있습니다. 마이크로소프트에서 권장하는 가상 메모리의 최대 크기는 시스템에 설치된 물리적 메모리의 3배이고, 디폴트 크기는 1.5배입니다. 사용자 지정으로 페이지 프레임의 크기를 늘릴 수 있지만 디폴트 크기를 유지할 것을 권장합니다.

일반적인 운영체제의 가상메모리 운영 방식은 다음과 같습니다. 운영체제는 물리 메모리의 효율적 사용을 위해 프로세스의 모든 페이지를 물리 메모리에 적재해 놓지 않습니다. 그래서 프로세스가 참조하고자 하는 페이지가 물리 메모리에 부재할 수도 있는데 이를 페이지 폴트 (Page fault)라고 합니다. 페이지 부재가 발생하면 운영체제는 가상 메모리로 가서 해당 페이지를 찾아봐야 합니다. 이때 운영체제는 각 페이지가 저장되어 있는 주소 값들이 저장된 페이지 테이블을 활용하여 가상 메모리에서 페이지를 찾습니다. 만약 가상 메모리에서도 페이지가 존재하지 않아 페이지 폴트가 발생하면 물리적 메모리의 페이지 프레임에 참조된 페이지를 적재하고 페이지 테이블을 갱신한 뒤 다시 프로세스를 실행시킵니다. 그런데 페이지 프레임에 빈 공간이 없을 경우 FIFO, LRU 등의 페이지 교체 알고리즘을 사용하여 희생될 페이지를 선택하여 참조된 해당 페이지와 교체합니다.

2) 컴파일 및 실행 방법

마이크로소프트 비주얼 스튜디오 Microsoft Visual Studio)를 활용하여 가상 메모리 관리를 위해 필요한 페이지 교체 알고리즘을 구현한 코드를 컴파일하고 실행시켰습니다. 비주얼 스튜디오는 마이크로소프트 윈도우 운영체제에서 작동하며 다양한 언어로 프로그래밍할 수 있는 마이크로소프트의 통합 개발 환경입니다. 본 프로젝트에서는 C 언어로 코드를 작성하였습니다.

2. 구현 코드 설명

1) Page 구조체와 main 함수

Page 구조체는 다음과 같이 구현하였습니다.

```
typedef struct page {
    int page;
    int flag;
    int count;
} Page;
```

- int page: page 번호를 담을 변수
- int flag: page fault 발생 여부 표시할 변수
- int count: 각 함수 마다 다르게 활용된 변수로 필요시 아래 함수 코드 부분에서 설명하겠습니다.

Main 함수는 다음과 같습니다.

```
int main(void) {
    Page* prs;
    int n, m, w, k;
    scanf("%d %d %d %d", &n, &m, &w, &k);

    prs = (Page*)malloc(sizeof(Page) * k);
    prs->count = 0;
    prs->flag = 0;

for (int i = 0; i < k; i++) {
        scanf("%1d", &prs[i].page);
    }

    MIN(prs, k, m);
    FIFO(prs, k, m);
    LRU(prs, k, m);
    LFU(prs, k, m);
    WS(prs, k, m, w);
}
```

프로세스가 갖는 페이지의 개수 n, 페이지 프레임의 개수 m, 워킹 셋 알고리즘에서 활용될 윈도우 사이즈 w, 입력 받을 페이지 참조열의 길이 k를 연속으로 입력 받습니다. 그리고 sizeof(Page)*k만큼 동적 할당을 한 Page *prs을 하고, prs[i].page에 입력 받은 page reference string의 정수를 하나씩 저장합니다. 함수 호출시 입력 받은 값들을 각 함수의 매개변수로 전달해 줍니다.

2) 공통 코드

모든 함수에 공통적으로 적용된 코드에 대한 설명입니다.

```
Page *pf = (Page*)malloc(sizeof(Page) * m);
```

참조된 페이지를 적재할 페이지 프레임을 m만큼의 페이지를 적재할 수 있도록 배열로 구현했습니다.

```
Page *prs_(알고리즘]) = (Page*)malloc(sizeof(Page) * m);
for (int i = 0; i < k; i++) {
    prs_(알고리즘)[i].count = 0;
    prs_(알고리즘)[i].flag = 0;
    prs_(알고리즘)i].page = prs[i].page;
}
int page_fault_(알고리즘) = 0;
```

*prs_(알고리즘)은 각 알고리즘마다 활용할 페이지 참조열의 페이지들을 저장할 배열입니다. prs_(알고리즘)의 모든 요소들의 count와 flag를 0으로 초기화하고 main 함수에서 입력 받은 페이지 값들을 전달 받아 prs_(알고리즘) 배열에 저장했습니다. 그리고 각 알고리즘 별 총 페이지 부재 발생 횟수를 카운트할 page_fault_(알고리즘) 변수를 선언하고 0으로 초기화했습니다.

```
if (m >= k) {
    printf("총 page fault 발생 횟수: %d₩n", page_fault_(알고리즘));
    printf("no page fault occur₩n");
    return;
}
```

페이지 프레임의 개수 m이 페이지 참조열의 페이지 개수 k보다 크면 참조되는 모든 페이지들이 페이지 프레임에 적재되어 페이지 부재가 발생하지 않는 경우를 처리한 부분입니다.

```
printf("총 page fault 발생 횟수: %d₩n", page_fault_(알고리즘));
printf("page fault가 발생하는 page reference string의 index: ");
for (int i = 0; i < k; i++) {
    if (prs_(알고리즘)[i].flag == 1) {
        printf("%d ", i);
    }
}
```

각 알고리즘별 총 페이지 부재 발생 횟수와 페이지 참조열 중 페이지 부재가 발생하는 페이지의 인덱스를 출력하기 위한 코드입니다.

3) MIN algorithm

MIN 알고리즘은 1966년 Belady가 제안한 알고리즘으로 OPT(optimal) 알고리즘, B0 알고리즘이라고도 불립니다. MIN 알고리즘은 앞으로 가장 오랜 시간 동안 사용되지 않을 페이지를 교체하는 기법으로 다른 어떤 알고리즘보다 페이지 부재가 적게 발생합니다. 현재 시점에서 페이지 프레임에 적재되어 있는 특정 페이지가 현재부터 현재 이후에 참조될 시점까지의 거리를 forward distance라고 합니다. MIN 알고리즘에서는 Forward distance가 가장 큰 페이지가 교체 대상이 됩니다. 해당 알고리즘은 구현이 불가능하지만 페이지 부재 발생이 최소인 기법으로 교체 기법의 성능을 측정하는 기준점을 구하는데 활용됩니다.

페이지 참조열의 모든 참조 페이지에 대하여 아래의 알고리즘을 적용하여 구현했습니다.
페이지가 참조될 때마다 page_fault_(알고리즘)을 1씩 증가시키고, 참조된 페이지의 flag를 1로 설정했습니다.

```
if (i < m) {
        page_fault_min++;
        pf[i].page = prs_min[i].page;
        prs_min[i].flag = 1;
 }
```

처음에는 페이지 프레임이 빈 상태이기 때문에 참조되는 페이지를 모두 적재 시킬 수 있습니다. 그래서 교체 알고리즘을 적용할 필요가 없습니다. 지금부터 페이지 프레임에 빈 공간이 없어 다음 페이지 참조부터는 페이지 알고리즘을 적용하기 위한 코드에 대한 설명을 하겠습니다.

```
if (i >= m) {
        int match = -1;
        int prsi = prs_min[i].page;
        for (int j = 0; j < m; j++) {
           if (prsi == pf[j].page) {
               match = 1;
               break;
           }
           else
               match = 0;
       }
```

match는 페이지 프레임 내에 현재 참조하는 페이지가 있는지 여부를 표시하는 변수입니다. 현재 참조하는 페이지를 나타내는 prs_(알고리즘)[i].page를 prsi 라는 변수에 담고 페이지 프레임에 적재된 모든 페이지들과 비교하는 반복문을 통해서 일치하는 페이지가 있으면 match를 1로, 없으면 0으로 설정했습니다. match가 0인 경우, 즉, 페이지 부재가 발생한 경우 MIN 알고리즘을 적용하기 위한 코드는 아래와 같습니다.

```
if (match == 0) {
    page_fault_min++;
    prs_min[i].flag = 1;
```

일단 페이지 부재 발생을 표시하기 위해 page_fault_min을 1 증가시키고, 현재 참조 페이지의 flag를 1로 바꿉니다. 그리고 교체 대상을 고르기 위해 페이지 프레임에 있는 각 페이지들의 forward distance를 구합니다.

```
                for (int j = 0; j < m; j++) {
                    int p = i + 1;
                    while ((pf[j].page != prs_min[p].page) && (p < k)) {
                        p++;
                    }
                    pf[j].count = p - i;
                }
```

페이지 참조열에서 현재 페이지 이후의 페이지들을 검사하며 해당 페이지와 동일한 페이지를 발견할 때까지 p값을 증가시키며 이동합니다. 현재 이후의 페이지들을 검사하기 위해 p는 i+1부터 검사를 시작합니다. 페이지 프레임에 있는 각 페이지의 forward distance를 나타내는 count 변수에 p값을 반영합니다.

```
                max = pf[0].count;
                int temp = 0;
                for (int j = 0; j < m; j++) {
                    if (pf[j].count > max) {
                        max = pf[j].count;
                    }
                }
                for (int j = 0; j < m; j++) {
                    if (pf[j].count == max) {
                        temp = j;
                    }
                }
```

페이지 프레임의 각 페이지들의 forward distance를 모두 구하여 count에 값을 저장한 뒤 count가 가장 큰 값을 교체 대상으로 선정합니다. pf[i].count의 값이 가장 큰 pf[i]를 구하기 위한 코드는 위와 같습니다. count 값이 max로 저장된 pf[i]의 인덱스 값을 temp에 저장합니다.

```
                Page temp2 = prs_min[i];
                prs_min[i] = pf[temp];
                pf[temp] = temp2;
                prs_min[i].flag = 1;
            }
```

위의 결과를 통해 얻은 temp를 활용하여 페이지 프레임 내에서 가장 큰 forward distance를 갖는 페이지 pf[temp]에 접근할 수 있습니다. 현재 참조 페이지와 교체 대상인 pf[temp]와 교체를 해줍니다. 그리고 이후에 페이지 부재가 발생한 페이지의 인덱스를 표시하기 위해 교체된 페이지의 flag도 1로 바꿔줍니다. 지금까지 MIN 알고리즘을 적용하여 페이지 교체를 위한 코드를 살펴보았습니다. 아래는 match가 1인 경우 즉, 페이지 프레임에 참조하는 페이지가 존재하면 페이지 부재가 발생하지 않으므로 위의 코드를 페이지 참조열이 끝날 때까지 반복할 수 있도록 continue를 합니다.

```
            else if (match == 1) {

                continue;
            }

        }
    }
```

4) FIFO algorithm

FIFO(First-In First-Out) 알고리즘은 페이지 프레임에 적재된 시간이 가장 오래된 페이지를 교체하는 알고리즘입니다. 주로 큐 자료 구조를 이용하여 간단하게 구현할 수 있습니다. 그러나 보통 페이지 프레임의 크기를 증가시키면 페이지 부재 발생률이 감소하지만 FIFO 알고리즘의 경우 페이지 프레임의 크기를 증가시켜도 페이지 부재가 증가하는 이상 현상이 발생하기도 합니다. 이러한 현상을 FIFO 모순, Belady의 모순이라고 합니다. FIFO 알고리즘을 구현한 코드는 다음과 같습니다.

```
for (int i = 0; i < k; i++) {
            int prsi = prs_fifo[i].page;
            //page frame 채우기
            if (i < m) {
                page_fault_fifo++;
                pf[i].page = prsi;
                prs_fifo[i].flag = 1;
            }
```

FIFO 알고리즘도 MIN 알고리즘과 동일하게 페이지 프레임이 가득 찰 때까지 page_fault_fifo를 1씩 증가시키고, 참조되는 페이지의 flag를 1씩 증가시켰습니다. 그리고 페이지 프레임이 가득 찬 이후 교체 대상 선정을 위해 FIFO알고리즘 적용하는 코드는 다음과 같습니다.

```
            if (i >= m) {
                int match = -1;
                for (int j = 0; j < m; j++) {
                    if (prsi == pf[j].page) {
                        match = 1;
                        break;
                    }
                    else
                        match = 0;
                }
```

페이지 부재 발생 여부를 표시하는 match를 활용하는 것은 MIN 알고리즘과 동일합니다. FIFO 알고리즘은 큐 자료구조를 이용하여 구현할 수 있지만 큐 자료구조를 위한 enqueue, dequeue 등의 함수를 따로 정의하는 것이 더 복잡할 것 같아 아래와 같은 방식으로 구현했습니다.

```
                if (match == 0) {
                    page_fault_fifo++;
                    prs_fifo[i].flag = 1;
```

일단 match가 0 이면 즉, 페이지 부재가 발생하면 page_fault_fifo를 1 증가시키고, 참조 페이지의 flag를 1로 설정합니다.

```
                    for (int j = 0; j < m - 1; j++) {
                        Page temp = pf[j];
                        pf[j] = pf[j + 1];
                    }
                    pf[m - 1] = prs_fifo[i];
```

그리고 페이지 프레임 내부에 FIFO가 적용될 수 있도록 배열을 한 칸씩 밀고 왼쪽으로 밀어 가장 먼저 들어온 페이지를 내보내고 페이지 프레임의 마지막 공간 pf[m-1]에 현재 참조 페이지를 적

재합니다.

```
                }
                else if (match == 1) {
                    continue;
                }
```

match가 1 인 경우는 MIN 알고리즘과 동일합니다.

```
            }
        }
    }
```

5) LRU algorithm

LRU(Least_Recently_Used) 알고리즘은 가장 오랫동안 참조되지 않은 페이지를 교체하는 기법으로 오늘날 대부분의 시스템에서 사용되고 있습니다. LRU 알고리즘은 가장 오랫동안 참조되지 않은 페이지를 교체합니다. 현재 이전에 참조된 시점부터 현재까지의 거리를 의미하는 backward distance가 가장 큰 페이지가 교체 대상이 됩니다. 이 알고리즘은 지역성을 잘 활용하여 MIN 알고리즘에 가장 근사하는 성능을 보입니다. LRU 알고리즘은 forward distance를 이용하는 MIN 알고리즘과 유사합니다.

```
for (int i = 0; i < k; i++) {
            if (i < m) {

                page_fault_lru++;
                pf[i].page = prs_lru[i].page;
                prs_lru[i].flag = 1;
            }
```

위의 함수들과 동일한 방식으로 페이지 프레임이 가득 찰 때까지 참조 페이지를 적재하고 page_fault_lru를 1씩 증가시키고 참조 페이지의 flag를 1로 설정합니다.

```
            if (i >= m) {
                int match = -1;
                int prsi = prs_lru[i].page;

                for (int j = 0; j < m; j++) {
                    if (prsi == pf[j].page) {
                        match = 1;
                        break;
                    }
                    else
                        match = 0;
                }
```

위의 함수들과 동일한 방식으로 match 변수를 활용하여 페이지 부재 여부를 표시합니다.

```
                if (match == 0) {
                    page_fault_lru++;
                    prs_lru[i].flag = 1;
                    int prsi = prs_lru[i].page;

                    for (int j = 0; j < m; j++) {
                        int p = i;
                        while ((pf[j].page != prs_lru[p].page) && (p >= 1)) {
                            p--;
                        }
                        pf[j].count = i - p;
                    }
```

MIN 알고리즘과 유사하지만 LRU 알고리즘에서는 페이지 참조열에서 현재 이전의 페이지들을 확인하여 forward distance를 활용합니다. 그래서 p를 감소시키며 페이지 참조열에서 현재 이전의 페이지들을 확인하며 pf[j]와 동일한 페이지가 있는지 확인합니다. 페이지 프레임마다 현재 위치 i를 기준으로 backward distance를 구해 count에 저장합니다.

```
                int max = pf[0].count;
                int temp = 0;
                for (int j = 0; j < m; j++) {
                    if (pf[j].count > max) {
                        max = pf[j].count;
                    }
                }
                for (int j = 0; j < m; j++) {
                    if (pf[j].count == max) {
                        temp = j;
                    }
                }
```

MIN 알고리즘과 유사한 방식으로 count에 저장한 backward distance 값이 가장 큰 pf[j]를 구하고 해당 페이지의 인덱스를 temp에 저장합니다.

```
                Page temp2 = prs[i];
                pf[temp] = temp2;
                prs_lru[i].flag = 1;
            }
```

그리고 pf[temp]와 현재 참조 페이지를 교체해줍니다. 그리고 교체된 참조 페이지의 flag를 1로 바꿔줍니다.

```
            else if (match == 1) {
                continue;
            }
        }
    }
}
```

위의 함수들과 마찬가지로 페이지 부재가 발생하지 않았을 경우에는 위의 반복문을 계속해서 수행하게 합니다.

6) LFU algorithm

LFU(Least-Frequently-User) 알고리즘은 현재까지 프로세스가 실행되면서 참조된 횟수가 가장 작은 페이지를 교체합니다. 따라서 각 페이지마다 현재까지 참조된 누적 횟수를 카운트 해야 하는 오버헤드가 있고, 최근에 참조되어 지역성 측면에서 참조 가능성이 높은 페이지도 교체될 수 있다는 단점이 있습니다.

```
for (int i = 0; i < k; i++) {
            int prsi = prs_lfu[i].page;
            if (i < m) {
                page_fault_lfu++;
                pf[i].page = prs_lfu[i].page;
                prs_lfu[i].flag = 1;
                prs_lfu[i].count++;
            }
```

위의 함수들과 동일한 방식으로 페이지 프레임이 가득 찰 때까지 참조 페이지를 적재하고 page_fault_lfu를 1씩 증가시키고 참조 페이지의 flag를 1로 설정합니다. LRU알고리즘은 각 페이지의 참조 횟수를 교체 기준으로 하기 때문에 페이지의 count를 참조 횟수로 활용했습니다.

```
            if (i >= m) {
                int match = -1;
                for (int j = 0; j < m; j++) {
                    if (prsi == pf[j].page) {
                        match = 1;
                        break;
                    }
                    else
                        match = 0;
                }
```

위의 함수들과 동일한 방식으로 match 변수를 활용하여 페이지 부재 여부를 표시합니다.

```
                if (match == 1) {
                    prs_lfu[i].count++;
                    continue;
                }
```

LRU 알고리즘에서는 페이지 프레임에 해당하는 참조 페이지가 적재해 있으면 다음 참조 페이지로 넘어가서 반복문을 계속해서 실행합니다. 그런데 여기서 다른 알고리즘과 다른 점은 페이지 참조 횟수를 증가시켜야 하는 것입니다.

```
                else if (match == 0) {
                    page_fault_lfu++;
                    prs_lfu[i].flag = 1;
                    prs_lfu[i].count++;

                    int min = pf[0].count;
                    int temp = 0;

                    for (int j = 0; j < m; j++) {
                        if (pf[j].count < min) {
                            min = pf[j].count;
                        }
```

페이지 참조열에 따라 참조 페이지의 부재 및 적중 여부를 확인하면서 각 페이지의 count변수에 참조 횟수를 표시해 두었습니다. 위의 코드를 통해 가장 작은 count를 가진 페이지의 count값을 얻을 수 있습니다.

```
                else if (pf[j].count == min) {
                    min = pf[0].count;
                }
            }
```

그런데 페이지 프레임에 있는 페이지들의 참조 횟수가 동일한 경우 최소 count값을 구할 수 없습니다. 이런 경우 random 알고리즘을 적용하여 페이지 프레임의 0번 인덱스에 있는 페이지를 교체 대상으로 선정하기 위해 pf[0].count를 min 값으로 설정합니다.

```
            for (int j = 0; j < m; j++) {
                if (pf[j].count == min) {
                    temp = j;
                }
            }
```

위의 코드를 통해 페이지 프레임의 각 페이지 중에서 최소 참조 횟수를 가진 페이지의 참조 횟수를 min에 저장했습니다. 그래서 페이지 프레임 내 페이지의 count값이 min과 동일한 페이지가 LRU 알고리즘에서 페이지 교체 대상이 됩니다. 교체 작업에 활용하기 위해 해당 페이지의 인덱스 값을 temp에 저장합니다.

```
                Page temp2 = prs_lfu[i];
                pf[temp] = temp2;
                prs_lfu[i].flag = 1;
            }
        }
    }
```

페이지 프레임 내에서 가장 작은 참조 횟수를 가진 페이지를 현재 참조 페이지와 교체 시켜 줍니다. 그리고 해당 참조 페이지의 페이지 부재 발생 여부를 표시하기 위해 flag를 1로 설정합니다.

7) Working set algorithm

고정할당기법인 위의 알고리즘들과 달리 working set 알고리즘은 가변할당기법 기반의 알고리즘입니다. 워킹 셋은 한 프로세스가 특정 시점에 집중적으로 참조되는 페이지의 집합을 의미합니다. Working set 알고리즘은 현재 시점을 기준으로 직전 window size Δ 만큼의 시간동안 참조된 페이지를 워킹 셋에 놓고 워킹 셋은 프로세스 실행 내내 메모리에 상주합니다. 메모리 할당량을 조절할 수 있기 때문에 굳이 프로세스 자기 자신의 페이지를 교체하지 않고 메모리 전체를 대상으로 교체 기법을 적용할 수 있습니다. 따라서 global 교체 기법입니다.

```
for (int i = 0; i < k; i++) {
            int prsi = prs_ws[i].page;
            if (i < w) {
                page_fault_ws++;
                pf[i].page = prsi;
                prs_ws[i].flag = 1;
            }
```

위의 함수들과 동일한 방식으로 페이지 프레임이 가득 찰 때까지 참조 페이지를 적재하고 page_fault_ws를 1씩 증가시키고 참조 페이지의 flag를 1로 설정합니다.

```
            else if (i >= w) {
                for (int j = i - w, k = 0; k < w; j++, k++) {
                    pf[k].page = prs_ws[j].page;
                }
```

Working set 알고리즘에서는 고정된 윈도우 사이즈만큼 참조 페이지를 적재할 수 있고, 시간에 따라 변화 합니다. 여기서 time 1마다 페이지 1개를 참조한다고 가정하였습니다. 그래서 윈도우 사이즈만큼 k값을 증가시키며 페이지 프레임에 현재 참조 페이지 이전의 페이지 참조열에서 윈도우 사이즈만큼의 범위에 포함된 페이지들을 차례대로 적재하며 워킹 셋을 유지시켰습니다. 따라서 페이지 참조열에서 현재 참조하는 페이지의 인덱스를 나타내는 i값이 변화함에 따라 워킹 셋도 변화합니다.

```
                int match = -1;
                for (int j = 0; j < w; j++) {
                    if (prs_ws[i].page == pf[j].page) {
                        match = 1;
                        break;
                    }
                    else
                        match = 0;
                }
```

위의 함수들과 동일한 방식으로 match 변수를 활용하여 페이지 부재 여부를 표시합니다.

```
                if (match == 0) {
                    page_fault_ws++;
                    prs_ws[i].flag = 1;
                }
```

페이지 부재가 발생하면 page_fault_ws를 1씩 증가시키고, 해당 참조 페이지의 flag를 1로 설정합니다.

```
                else if (match == 1) {
```

```
                continue;

            }
        }
    }
```

다른 함수들과 마찬가지로 페이지 부재가 발생하지 않으면 별 다른 조치 없이 반복문을 계속해서 수행하게 합니다.

3. 실행 결과

5 3 3 15

012323454134345

프로세스가 갖는 페이지의 개수 n을 5, 페이지 프레임에 적재할 수 있는 페이지의 개수 m을 3, working set 알고리즘에서 사용할 윈도우 사이즈 w를 3, 페이지 참조열의 길이 k를 15로 입력했습니다. 그리고 012323454134345를 페이지 참조열로 입력하였습니다. 입력에 대한 출력 결과물은 다음과 같습니다.

```
========== MIN Algorithm ==========

총 page fault 발생 횟수: 7
page fault가 발생하는 page reference string의 index: 0 1 2 3 6 7 10

========== FIFO Algorithm ==========

총 page fault 횟수 : 10
page fault가 발생하는 page reference string의 index: 0 1 2 3 6 7 9 10 11 14

========== LRU Algorithm ==========

총 page fault 발생 횟수: 9
page fault가 발생하는 page reference string의 index: 0 1 2 3 6 7 9 10 14

========== LFU Algorithm ==========

총 page fault 발생 횟수: 10
page fault가 발생하는 page reference string의 index: 0 1 2 3 4 6 7 9 10 14

========== WS Algorithm ==========

총 page fault 발생 횟수 : 9
page reference string에서 page fault가 발생한 index : 0 1 2 3 6 7 9 10 14
```

MIN, FIFO, LRU, LFU, Working set 알고리즘 순서대로 결과가 출력됩니다. 각 알고리즘을 적용했을 때 총 페이지 부재 발생 횟수와 페이지 부재가 발생하는 페이지 참조열의 페이지 인덱스를 출력했습니다. 입력 예시와 같이 페이지 프레임에 적재할 수 있는 페이지의 개수를 3개로 제한하고 페이지 참조열의 길이가 15인 경우에 MIN 알고리즘을 적용하면 페이지 부재가 7번 발생합니다. 이는 입력 조건에서 페이지 부재가 최소 7번 발생한다는 것을 의미합니다. 각 실행 결과를 바탕으로 MIN의 성능에 가장 가깝게 페이지 부재가 발생하는 알고리즘은 LRU와 Working set 알고리즘이라는 것을 알 수 있습니다.

4. 참고문헌

bogus919, <윈도우의 가상 메모리란 무엇인가? - 가상 메모리(페이징 파일) 관리하기>, <<당신이 무엇을 배우고 있다는 것은 어제의 당신보다 훨씬 높은 가치가 되었다는 것이다>, 2020.05.29, https://bogus919.tistory.com/entry/%EB%8F%84%EC%9A%B0%EC%9D%98-%EA%B0%80%EC%83%81-%EB%A9%94%EB%AA%A8%EB%A6%AC%EB%9E%80-%EB%AC%B4%EC%97%87%EC%9D%B8%EA%B0%80-%EA%B0%80%EC%83%81-%EB%A9%94%EB%AA%A8%EB%A6%AC%ED%8E%98%EC%9D%B4%EC%A7%95-%ED%8C%8C%EC%9D%BC-%EA%B4%80%EB%A6%AC%ED%95%98%EA%B8%B0

MS Windows, wikipedia, 2020.05.29, https://ko.wikipedia.org/wiki/%EB%A7%88%EC%9D%B4%ED%81%AC%EB%A1%9C%EC%86%8C%ED%94%84%ED%8A%B8_%EC%9C%88%EB%8F%84%EC%9A%B0

Preantree, <[IT 기술면접 준비자료] 가상메모리의 동작과 페이지폴트(Page Fault)>, <<Preamtree의 행복로그>>, 2016.12.11, https://preamtree.tistory.com/21

마이크로소프트 윈도우 가상메모리, Microsoft, 2020.05.29, https://answers.microsoft.com/ko-kr/windows/forum/windows_xp-performance/%EA%B0%80%EC%83%81/2586a8e5-0728-42b3-a259-2f76f918e6b9

비주얼 스튜디오, wikipedia, 2020.05.29, https://ko.wikipedia.org/wiki/%EB%A7%88%EC%9D%B4%ED%81%AC%EB%A1%9C%EC%86%8C%ED%94%84%ED%8A%B8_%EB%B9%84%EC%A3%BC%EC%96%BC_%EC%8A%A4%ED%8A%9C%EB%94%94%EC%98%A4