운영체제 과제3 : 가상메모리

201619460 이성규

먼저 os3-1의 코드에 대해 설명을 한 이후 os3-2의 코드는 차이점 위주로 설명을 진행하겠습니다.

코드 : 과제 3-1 (os3-1)

```
8   #define PAGESIZE (32)
9
10  #define PAS_FRAMES (256)  // fit for unsigned char frame in PTE
11
12  #define PAS_SIZE (PAGESIZE*PAS_FRAMES) //32*256 = 8192 B
13
14  #define VAS_PAGES (64)
15
16  #define VAS_SIZE (PAGESIZE*VAS_PAGES)  // 32*62 = 2048 B
17
18  #define PTE_SIZE (4)  // sizeof(pte)
19
20  #define PAGETABLE_FRAMES (VAS_PAGES*PTE_SIZE/PAGESIZE) //64*4/32 = 8 consecutive frames
21
22  #define PAGE_INVALID (0)
23
24  #define PAGE_VALID (1)
25
26  #define MAX_REFERENCES (256)
```

←
먼저 교수님께서 영상에서 올려주신 Define입니다.
저도 강의영상 속 그대로 매크로(전처리기)들을 채용하여 사용하였습니다.

그 외의 다른 구조체들의 정의도 똑같이 채용하였습니다. →
pte는 Page Table Entry로써,
process_raw는 bin파일로부터 정보를 받을 프로세스로,
그리고 frame은 물리메모리를 나누는 프레임으로
사용하였습니다.

```
29  typedef struct{
30
31      unsigned char frame; // allocated frame
32
33      unsigned char vflag; // valid-invalid bit
34
35      unsigned char ref;   // reference bit
36
37      unsigned char pad;   // padding
38  } pte; // Page Table Entry (total 4 bytes, always)
39
42  typedef struct {
43
44      int pid;
45
46      int ref_len;  // Less than 255
47
48      unsigned char *references;
49  } process_raw;

53  typedef struct{
54
55      unsigned char b[PAGESIZE];
56
57  } frame;
```

여기에 강의영상에서는 없던 제가 임의로 추가한 구조체가 하나 있는데, ↓ 아래에 나와 있는 process 구조체입니다.

```
60  typedef struct{
61
62      process_raw pcess;
63
64      int AllocatedFrames;
65
66      int PageFault;
67
68      int ReferenceCount;
69
70  }process;
```

JOTA 출력 양식을 보면 각 프로세스별 PageFault가 일어난 횟수와 ReferenceCount, Allocated Frames를 저장하고 출력해야했습니다. 이에, 각 프로세스가 저장해야할 변수를 Process_raw와 같이 묶어서 구조체를 새로 선언하여 다루었습니다.

여기서부터 메인 함수입니다.

```
78      process *cur; // cur : 파일 읽은거 담을 구조체(프로세스) 포인터
79
80      cur = (process *) malloc(sizeof(process));
81
82
83      process* processArr[10];          // bin 파일에서 받는 프로세스 저장할 배열
84      int processnum = 0;             // bin 파일에서 몇개의 프로세스 받았는지 저장
```

먼저 process 포인터 cur을 선언하고, 동적 할당을 해주었습니다.
이를 통해 bin파일로부터 정보를 받을 생각입니다.
그리고 process를 가리킬 수 있는 배열 processArr 과 bin으로부터 들어온 프로세스 개수를 카운트하는 pocessnum 선언하였습니다.

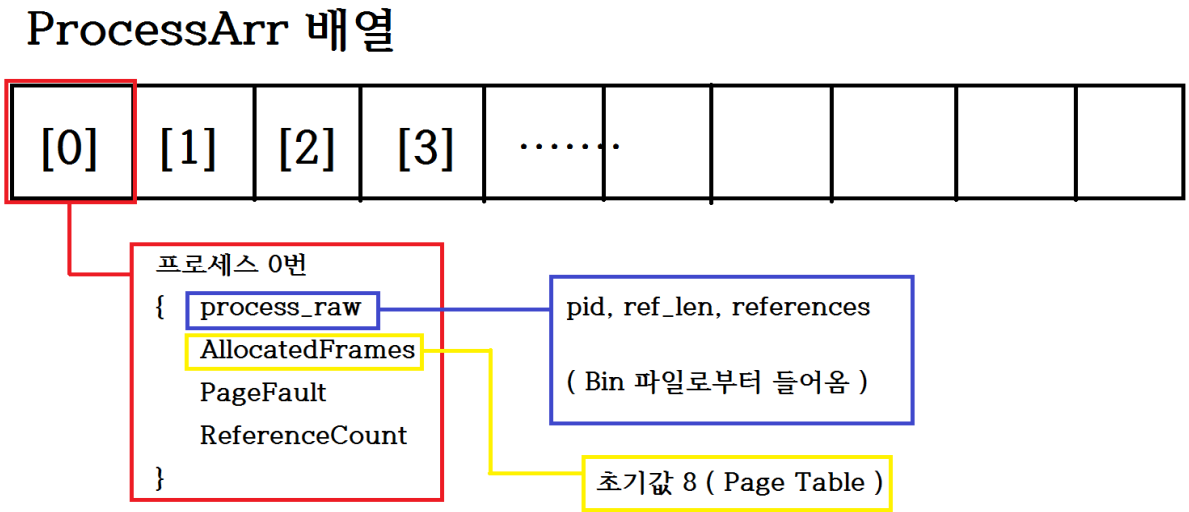
```
86      while(fread(cur, sizeof(int)*2, 1, stdin) == 1) { //프로세스 읽기, 실패시 while문 종료
87
88          processArr[processnum] = (process *) malloc(sizeof(process));
89
90          processArr[processnum]->pcess.pid = cur->pcess.pid;
91          processArr[processnum]->pcess.ref_len = cur->pcess.ref_len;
92
93          processArr[processnum]->pcess.references = (unsigned char *) malloc(processArr[processnum]->pcess.ref_len); // 할당
94          fread(processArr[processnum]->pcess.references,processArr[processnum]->pcess.ref_len,1,stdin); //코드 넣기
95
96
97          processArr[processnum]->AllocatedFrames = 8; // 기본으로 Table 할당되므로 기본값 8으로 설정
98          processArr[processnum]->PageFault =0;
99          processArr[processnum]->ReferenceCount =0;
100
101
102         processnum++;
103
104     }
```

그 이후, while문을 돌며 bin파일로부터 프로세스 정보를 받아왔습니다.
process는 0번부터 순서대로 들어오므로, process 0번은 processArr[0]이 가리키도록, process 1번은 processArr[1]이 가리키도록 배열의 인덱스와 프로세스의 번호가 동일하게 저장되도록 구현하였습니다.
또한 프로세스 개수가 추가됨에 따라 processnum을 증가시켜 줌으로써 processArr 배열에서 어디까지 실질적으로 프로세스가 할당되어있는지 파악하기 쉽게 하였습니다.

단 여기에서 97 라인을 보면 모든 프로세스에서 AllocatedFrames를 8로 기본값 설정하는데, 이는 os3-1에서는 기본적으로 할당되는 Page Table이 8 Frame을 차지하기 때문입니다.

```
processArr[processnum]->AllocatedFrames = 8; // 기본으로 Table 할당되므로 기본값 8으로 설정
```

교수님께서 수업 강의 영상에서 불필요하게 반복 복사하는 코드에 대해 언급을 하셨던 게 기억났습니다. 그래서 이번에는 이전 과제들과는 다르게 cur을 선언하지 않고 직접 fread를 통해 processArr에 정보를 받으며 파일의 끝을 검사하는 코드를 구현해 보도록 노력해 보았으나, 생각보다 한 개의 변수로 검사와 정보 받는 것을 둘 다 구현하는 것이 어려워서 결국 구현하지는 못했습니다. 이 부분이 조금 어려웠던 것 같습니다.



처음에는 이전 과제처럼 배열이 아닌, Linux List를 사용하여 프로세스를 관리할까 생각하였습니다. 그러나 교수님께서 과제 영상에서 보여주신 process_raw 구조체에서는 List Head가 포함되어 있지 않았었고 (Process 구조체로 한번 더 감싼 것은 구현 도중에 추가 된 것이기에 이때 당시에는 생각하지 못했습니다.) 과제 조건과 os3-gen.c을 보면 프로세스의 개수는 10이하로 들어온다고 제시되어 있었기 때문에 좀 더 저에게 익숙하여 다루기 쉬운 배열을 사용하여 프로세스를 관리하였습니다.

- PID (4B), Length of reference sequence (4B), Page reference sequence (variable)
  - PID < 10, Ref_len < 256, page number < 64
  - Locality 를 가진 레퍼런스 패턴이 나오도록 설정되어 있음 (os3-gen.c 참조)

```
110     frame* PAS = (frame*)malloc( PAS_SIZE ); // 물리 메모리
113
114     int UsedPAS = 0;              // 물리 메모리에서 사용된 frame수를 저장. PAS[UsedPAS] 는 메모리가 다음에 써져야 할 공간을 의미함
115     UsedPAS += processnum * 8;  // 프로세스 갯수만큼 Table 할당됨.
```

모든 프로세스를 로드하였으니 그 이후 물리메모리를 구현해줍니다. (110 라인)
PAS_SIZE는 Page Size * 256 Frame 이므로 이는 PAS[0]부터 PAS[255]까지 존재하게 됩니다.
그 후 PAS가 얼마만큼 사용되었는지 저장하는 변수 UsedPAS 를 선언하였습니다.
UsedPAS는 논리적으로 PAS의 다음에 사용될 Frame 번호(인덱스)를 의미합니다.
PAS가 모두 사용되었을 때 UsedPAS는 256일 것입니다.
과제 os3-1에서는 프로세스마다 8 Frame의 테이블을 갖고 있으므로 프로세스의 개수만큼 8 Frame이 테이블로써 사용되었음을 표시해주었습니다. (라인 115)

그 이후 페이지 테이블에 Entry로서 접근하기 위한 변수로 cur_pte를 선언해주고 할당된 프로세스의 page table을 초기화 해주었습니다.
각 프로세스의 page table 은 "프로세스번호 * 8" 의 PAS 프레임에 64개의 pte로 할당되어있습니다. (라인 125)

```
119     pte* cur_pte;  // 테이블 접근 위한 포인터
120
121     int i;
122     int j;
123     for( i = 0 ; i < processnum ; i++) // page table 초기화
124     {
125         cur_pte = (pte *) &PAS[i*8];
126
127         for( j = 0 ; j < 64 ; j++)
128         {
129             cur_pte[j].vflag = 0; // 0이면 유효하지 않음. 1이면 유효함
130             cur_pte[j].ref = 0;   // 초기 ref = 0.
131         }
132
133     }
```

즉 0번 프로세스는 0 * 8 = 0번. PAS[0]에서 PAS[7]까지, 1번 프로세스는 1 * 8 = 8번. PAS[8]에서 PAS[15]까지 Page Table로 할당되었습니다. (각 페이지 테이블은 cur_pte[0] ~ cur_pte[63]으로 할당)

각 프로세스를 bin으로부터 다 받았고, 프로세스가 할당된 Page Table도 모두 초기화 하였으니 이제 while문을 돌며 Demand paging 동작을 진행합니다. while문은 Ref_len의 최대치인 255 까지 진행되거나, 실행 도중 PAS가 부족할 때까지 진행되도록 구현하였습니다. (라인 142)

- PID (4B), Length of reference sequence (4B), Page reference sequence (variable)
  - PID < 10, Ref_len < 256, page number < 64
  - Locality 를 가진 레퍼런스 패턴이 나오도록 설정되어 있음 (os3-gen.c 참조)

```
142 while( REF < 256 && UsedPAS < 256) // 레퍼런스 다 보거나 PAS가 사용 가능할때
143 {
144     for( i = 0 ; i < processnum; i++ )                                  // 0번 프로세스부터 순서대로 진행
145     {
146         // 헷갈리지 않게 현재 진행중인 process 변수 갱신.
147         cur = processArr[i];                //현재 ref하는 프로세스
148         cur_pte = (pte *) &PAS[i*8];    //현제 ref하는 프로세스의 page table
149
150         if( cur->pcess.ref_len > REF) // Refer할게 있다면
151         {
152             if( cur_pte[cur->pcess.references[REF]].vflag == 1 ) // 물리 프레임 할당 되있음. ( 유효 )
153             {
154                 // 별 작동 없이 ref만 더하기
155                 cur_pte[cur->pcess.references[REF]].ref += 1;
156                 cur->ReferenceCount += 1;                               // 프레임 할당 되있음.
157             }
158             else // 물리 프레임 할당 X -> Page Fault
159             {                                                           // 프레임 할당 X : Page Fault
160
161                 if( UsedPAS >= 256 ) // 메모리 부족시 중지                  .
162                 {
163                     break;
164                 }
165
166                 cur->PageFault += 1;
167                 cur->ReferenceCount += 1;  // 프로세스의 ref 카운트 더하기
168                 cur_pte[cur->pcess.references[REF]].frame = UsedPAS;   // 새로운 물리 프레임 할당 저장
169                 cur_pte[cur->pcess.references[REF]].ref += 1;          // ref 더하기
170                 cur_pte[cur->pcess.references[REF]].vflag = 1;         // 유효하다고 표시
171                 cur->AllocatedFrames += 1;
172                 UsedPAS++;
173
174
175             }
176
177         }
178
179     }
180
181     REF++;
182 }
```

위에 나온 while문 코드가 어떻게 동작하는지 차분히 살펴보겠습니다.
먼저 for 문을 통해 배열 processArr의 0번 인덱스 프로세스부터 마지막 프로세스까지 순회하는데,
각 순회할 때마다 현재 진행 중인 process를 cur로, Page Table을 cur_pte 로 알맞게 갱신합니다.
페이지 테이블은 프로세스별 위치한 PAS 프레임이 다르므로 갱신을 해줘야하지만 프로세스는 cur로 굳이 갱신을 하지 않아도 됩니다만, 통일성과 가독성을 위해 cur로 갱신을 해주었습니다.

```
144         for( i = 0 ; i < processnum; i++ )
145         {
146             // 헷갈리지 않게 현재 진행중인 process 변수 갱신.
147             cur = processArr[i];                //현재 ref하는 프로세스
148             cur_pte = (pte *) &PAS[i*8];    //현제 ref하는 프로세스의 page table
```

processArr은 0번부터 프로세스 pid 순서대로 들어있게 하였으므로, 프로세스 순서대로 Reference하는 과제 조건에 일치하게 됩니다.

```
150             if( cur->pcess.ref_len > REF) // Refer할게 있다면
```

cur을 알맞게 갱신했다면, cur프로세스에서 reference할 항목이 있는지 확인하고 그 항목이 Page Table에 할당이 되어있는지 검사하게 됩니다. (150 라인)

```
152             if( cur_pte[cur->pcess.references[REF]].vflag == 1 ) // 물리 프레임 할당 되있음. ( 유효 )
153             {
154                 // 별 작동 업이 ref만 더하기
155                 cur_pte[cur->pcess.references[REF]].ref += 1;
156                 cur->ReferenceCount += 1;
157             }
```

만약 cur프로세스가 요구하는 reference가 이미 프레임에 할당이 되어있다면 그냥 각 테이블과 프로세스의 reference를 증가시킵니다. (152 ~ 157 라인)

이때 여기에서, 152 라인을 보면 cur_pte[cur->pcess.references[REF]].vflag를 통해 페이지 폴트를 검사하게 되는데, 이것이 의미하는 것은 다음과 같습니다.
cur_pte는 cur 프로세스의 Page table입니다. 총 cur_pte[0]부터 cur_pte[63] 까지 64개가 있습니다.
cur->pcess.references[REF] 는 현재 진행하는 프로세스(cur)의 REF번째 references를 의미합니다.
즉 152라인은 "페이지테이블[접근reference].vflag" 를 의미합니다.

```
158             else // 물리 프레임 할당 X -> Page Fault
159             {
160 
161                 if( UsedPAS >= 256 )  // 메모리 부족시 중지
162                 {
163                     break;
164                 }
165 
166                 cur->PageFault += 1;
167                 cur->ReferenceCount += 1;  // 프로세스의 ref 카운트 더하기
168                 cur_pte[cur->pcess.references[REF]].frame = UsedPAS;   // 새로운 물리 프레임 할당 저장
169                 cur_pte[cur->pcess.references[REF]].ref += 1;          // ref 더하기
170                 cur_pte[cur->pcess.references[REF]].vflag = 1;         // 유효하다고 표시
171                 cur->AllocatedFrames += 1;
172                 UsedPAS++;
173 
174 
175             }
```

만약 물리 메모리에 프레임이 할당이 되어 있지 않다면 페이지 테이블의 vflag가 0일 테고, 페이지 폴트가 발생합니다. (라인 158 ~ 175) 이에 cur프로세스의 Page Fault값과 Reference값을 올리고, Page Table의 정보를 업데이트 합니다. (라인 166 ~ 171)
그리고 PAS를 현재 접근하려는 페이지로써 메모리 할당했다는 것을 의미하고자 UsedPAS를 증가시켜줍니다.
(라인 172)

이때 Page Fault가 일어났을 때 사용 중인 메모리를 검사하여 더 이상 PAS가 없다면 시뮬레이터를 종료합니다.

```
if( UsedPAS >= 256 ) // 메모리 부족시 중지
{
    break;
}
```

라인 161 ~ 164를 통해 UsedPAS. 즉 다음에 사용해야할 메모리가 256이상이라면 (PAS[256]은 존재하지 않으므로) break를 통해 프로세스를 순회하고 있는 for문을 나오게 되고,
140 라인의 While문의 조건식인 “UsedPAS < 256” 에 걸려서 바로 reference가 남아있더라도 다음 while 루프를 돌지 않고 시뮬레이터를 종료하게 됩니다.

이처럼 for문을 통해 0번 프로세스부터 순차적으로 페이지에 접근을(reference) 합니다.
for문이 다 끝나게 되면 REF를 증가시키고 다시 반복을 하는데, 이는 PAS를 다 쓰거나 255레퍼런스 모두 다 할 때까지 반복합니다.

```
180
181         REF++;
182     }
```

While문이 다 종료되어 시뮬레이터가 끝났다면 시뮬레이터가 종료된 이유가
물리적 메모리를 다 소모해서인지 확인합니다.

```
186     if( UsedPAS >= 256 && REF < 256 ) // 메모리가 부족해서 종료된것인지 확인
187     {
188         fprintf(stdout, "Out of memory!!\n");
189     }
```

시뮬레이터가 끝났을 때, UsedPAS가 256 이상이며 (일반적으로는 256임), REF가 256 이하인 경우에 Out of memory 메시지를 출력해줍니다. (186 라인)
만약 확인할 때 REF는 확인하지 않고 UsedPAS만 확인하여 OOM을 출력하게 된다면, 딱 맞게 PAS[255] 까지만 메모리 할당을 하고 Page Fault가 일어나지 않은 채 정상적으로 종료된 경우에도 OOM 메시지가 출력되므로 if문에는 REF < 256 조건도 함께 있어야 합니다.

이후, 최종 리포트 출력을 위해 각 프로세스를 돌며 페이지 폴트 값과 Ref 값을 저장할 변수를 선언해주고,
(라인 191, 192)
for문을 돌며 각 프로세스별 정보 와 페이지 테이블의 유효 값인 부분만 출력해줍니다.

```
191     int AllFaults = 0;
192     int AllRef = 0;
193
194     // 각 프로세스별 정보 및 페이지 테이블 출력
195     for( i = 0 ; i < processnum ; i++)
196     {
197         fprintf(stdout, "** Process %03d: Allocated Frames=%03d PageFaults/References=%03d/%03d\n" ,
198         AllFaults += processArr[i]->PageFault;          processArr[i]->pcess.pid, processArr[i]->AllocatedFrames,  processArr[i]->PageFault, processArr[i]->ReferenceCount);
199         AllRef += processArr[i]->ReferenceCount;
200
201         cur_pte = (pte *) &PAS[i*8];
202
203         for( j = 0 ; j < 64 ; j++)
204         {
205             if(cur_pte[j].vflag == 1)
206             {
207                 fprintf(stdout, "%03d -> %03d REF=%03d\n" , j, cur_pte[j].frame, cur_pte[j].ref);
208             }
209
210         }
211
212     }
213
214     // 모든 작업 종료 후 최종리포트
215     fprintf(stdout, "Total: Allocated Frames=%03d Page Faults/References=%03d/%03d\n" , UsedPAS,AllFaults, AllRef);
```

라인 198 뒤쪽의 “processArr[i]....” 코드는 197라인 코드의 뒷부분입니다.

이때 프로세스의 페이지 테이블 출력 코드를 자세히 보면 (201 라인 ~ 210 라인)

```
cur_pte = (pte *) &PAS[i*8];

for( j = 0 ; j < 64 ; j++)
{
    if(cur_pte[j].vflag == 1)
    {
        fprintf(stdout, "%03d -> %03d REF=%03d\n" , j, cur_pte[j].frame, cur_pte[j].ref);
    }

}
```

이렇게 되어있습니다.
여기에서 i는 프로세스의 pid를 의미하고, j는 페이지 테이블의 index를 의미합니다. os3-1에서는 페이지 테이블의 인덱스가 프로세스의 page number와 동일하므로 이렇게 출력하도록 구현하였습니다.
os3-2에서는 조금 다르게 출력하도록 구현되어 있습니다. 이는 페이지 테이블이 레벨이 나뉘어 있어 Lv2 페이지 테이블의 인덱스가 프로세스의 page number 과 다르기 때문입니다. 이에 관해서는 뒤에서 os3-2 설명할 때 다시 이야기 하도록 하겠습니다.

코드 : 과제 3-2 (os3-2)
기본적으로 os3-1과 작동 방식과 선언된 변수 및 구조체들은 크게 다르지 않습니다. 그렇기에 os3-1에서 바뀐 부분만 부분적으로 설명하겠습니다.

먼저 os3-1과 동일하게 bin 파일로부터 프로세스정보를 받습니다.

```
86      while(fread(cur, sizeof(int)*2, 1, stdin) == 1) { //프로세스 읽기, 실패시 while문 종료
87
88          processArr[processnum] = (process *) malloc(sizeof(process));
89
90          processArr[processnum]->pcess.pid = cur->pcess.pid;
91          processArr[processnum]->pcess.ref_len = cur->pcess.ref_len;
92
93          processArr[processnum]->pcess.references = (unsigned char *) malloc(processArr[processnum]->pcess.ref_len); // 할당
94          fread(processArr[processnum]->pcess.references,processArr[processnum]->pcess.ref_len,1,stdin); //코드 넣기
95
96
97          processArr[processnum]->AllocatedFrames = 1; // 기본으로 Level 1 Page Table 할당되므로 기본값 1으로 설정
98          processArr[processnum]->PageFault =0;
99          processArr[processnum]->ReferenceCount =0;
100
101
102         processnum++;
103
104     }
```

단 이때, processArr 배열에 들어있는 각 프로세스의 AllocatedFrames를 8이 아닌 1로 설정해 줍니다.
(라인 97)
이는 os3-2에서는 기본으로 프로세스가 로드될 때 주어지는 페이지 테이블(Lv 1 Page Table)이 1프레임을 차지하기 때문입니다. (os3-1 에서는 8 프레임을 차지했었습니다.)

```
114     int UsedPAS = 0;              // 물리 메모리에서 사용된 frame수를 저장. PAS[UsedPAS] 는 메모리가 다음에 써져야 할 공간을 의미함
115     UsedPAS += processnum * 1;  // 프로세스 갯수만큼 Table 할당됨.
116
117
118
119     pte* cur_pte_L1;  // 테이블 접근 위한 포인터 (level 1)
120     pte* cur_pte_L2;  // 테이블 접근 위한 포인터 (level 2)
```

그 후 동일하게 UsedPAS변수와 테이블 접근을 위한 포인터를 선언하고 초기화 해주는데,
페이지 테이블의 기본 할당된 Frame수가 줄어들었음에 따라 사용된 PAS용량, 즉 UsedPAS 변수도 적게 증가하도록 바꾸었습니다. (라인 115)
그리고 이번에는 페이지 테이블 접근을 위한 포인터를 레벨별로 두 개 선언하였습니다.
원래는 한 개의 포인터로 가리키는 페이지 테이블을 LV1때와 LV2때 마다 바꿔가며 코드를 작성하였으나, 마지막 while문이 끝나고 최종 출력 부분에서 한 개의 변수로 결과를 출력하기가 약간 구현하기 까다로웠기에, 가독성도 높일 겸 두 개의 페이지 테이블 접근 포인터를 구분하여 선언하였습니다.

```
124     for( i = 0 ; i < processnum ; i++) // page table 초기화
125     {
126         cur_pte_L1 = (pte *) &PAS[i*1];
127 
128         for( j = 0 ; j < 8 ; j++)
129         {
130             cur_pte_L1[j].vflag = 0; // 0이면 유효하지 않음. 1이면 유효함
131             cur_pte_L1[j].ref = 0;   // 초기 ref = 0.
132         }
133 
134     }
```

페이지 테이블 초기화 코드도 64개(8 frame)에서 8개(1 frame)로 줄었습니다.

이제 프로세스 정보를 받았으니 while문을 돌며 Lv2 Demand paging 동작을 진행합니다.

```
143     while( REF < 256 && UsedPAS < 256) // 레퍼런스 다 보거나 PAS가 사용 가능할떄
144     {
145         for( i = 0 ; i < processnum; i++ )
146         {
147             // 헷갈리지 않게 현재 진행중인 process 변수 갱신.
148             cur = processArr[i];              //현재 ref하는 프로세스
149 
150             if( cur->pcess.ref_len > REF) // Refer할게 있다면
151             {
152 
153 
154                 cur_pte_L1 = (pte *) &PAS[i*1];       // ref하는 프로세스의 Level 1 page table 갱신
155 
156 
157                 if( cur_pte_L1[cur->pcess.references[REF]/8].vflag == 0 ) // L1 할당이 되어있지 않다면. 할당. ( L1 Page Fault )
158                 {
159                     if( UsedPAS >= 256 )  // 메모리 부족시 중지
160                     {
161                         break;
162                     }
163 
164                     cur->PageFault += 1;
165                     cur_pte_L1[cur->pcess.references[REF]/8].frame = UsedPAS;   // 새로운 물리 프레임 할당 저장
166                     cur_pte_L1[cur->pcess.references[REF]/8].vflag = 1;         // 유효하다고 표시
167                     cur->AllocatedFrames += 1;
168                     UsedPAS++;
169                 }
170 
```

os3-1과 동일하게 Ref_len의 최대치인 255 까지 진행되거나, 실행 도중 PAS가 부족할 때까지 진행합니다.
(라인 143)
각 while 루프 마다 for 문을 통해 배열 processArr의 0번 프로세스부터 마지막 프로세스까지 순회하며 각 프로세스가 레퍼런스를 합니다. (라인 145 ~)

일단 cur에 현재 프로세스를 갱신합니다. (148 라인)
그 이후 프로세스의 Lv1 페이지 테이블을 가리키도록 cur_pte_L1을 갱신합니다. (154 라인)

그리고 나선, 갱신한 그 Lv1 페이지 테이블에서 해당 접근하려는 페이지가 할당되어있는지 확인합니다.
(157 라인)
L1 페이지 테이블을 확인할 때에는 과제 영상에서 교수님께서 설명 해주셨듯이 접근하려는 페이지를 8로 나누기 연산을 한 값을 이용하였습니다.
(157 라인 : cur_pte_L1의 “cur->pcess.reference[REF] / 8” 인덱스에 접근)

만약 해당 Lv1 테이블 속 PTE의 vflag을 확인하였을 때, Page Fault인 경우 (라인 157 ~ 169)
프로세스의 PageFault 카운트를 증가시키고, 새로운 메모리를 새로 할당해 줍니다.
이때에도, UsedPAS 변수를 이용해 남아있는 메모리가 있는지 파악한 후 할당해 줍니다. (라인 159 ~ 162)
(메모리가 없다면 break를 통해 시뮬레이션 종료)

```
174             cur_pte_L2 = (pte *) &PAS[cur_pte_L1[cur->pcess.references[REF]/8].frame];      //Level 2 page table로 cur_pte 갱신.
175
176
177             if( cur_pte_L2[cur->pcess.references[REF]%8].vflag == 1 ) // L2 에서 할당 되어 있음.
178             {
179                 // 접근
180                 cur_pte_L2[cur->pcess.references[REF]%8].ref += 1;
181                 cur->ReferenceCount += 1;
182             }
183             else // L2 Page Fault
184             {
185                 if( UsedPAS >= 256 )
186                 {
187                     break;
188                 }
189
190                 cur->PageFault += 1;
191                 cur->ReferenceCount += 1;  // 프로세스의 ref 카운트 더하기
192                 cur_pte_L2[cur->pcess.references[REF]%8].frame = UsedPAS;   // 새로운 물리 프레임 할당 저장
193                 cur_pte_L2[cur->pcess.references[REF]%8].ref += 1;          // ref 더하기
194                 cur_pte_L2[cur->pcess.references[REF]%8].vflag = 1;         // 유효하다고 표시
195                 cur->AllocatedFrames += 1;
196                 UsedPAS++;
197             }
198
199
200         }
```

Lv1 페이지 테이블에 저장된 정보를 통해 Lv2 페이지 테이블이 어느 PAS 프레임에 저장 되어있는지 확인을 하고, 이 정보를 통해 cur_pte_L2 포인터를 이 프로세스의 Lv2 페이지 포인터로 갱신을 합니다. (라인 174)

```
cur_pte_L2 = (pte *) &PAS[cur_pte_L1[cur->pcess.references[REF]/8].frame];      //Level 2 page table로 cur_pte 갱신.
```

PAS[Lv1.Frame]

그 이후 갱신한 Lv2 페이지 테이블에서 페이지 접근을 하게 됩니다.
이때 또한 과제 영상에서 교수님께서 이야기해주신 대로 접근하고자 하는 페이지에 % 8 나머지 연산을 사용하였습니다. (라인 177 : cur_pte_L2의 “cur->pcess.reference[REF] % 8” 인덱스에 접근)

Lv2 페이지 테이블에서의 접근은 과제 os3-1와 거의 유사합니다.
PTE를 확인하여 page fault가 일어나지 않았다면 프로세스와 PTE의 reference count를 올려주고,
(라인 177 ~ 182)
만약 PTE를 확인하였을 때 page fault가 일어났다면 os3-1과 동일히 PAS에 새로 메모리를 할당해주고 Lv2 페이지 테이블의 PTE를 갱신해 주었습니다. 물론 이때 Page Fault 카운트와 Ref 카운트도 증가시킵니다.
(라인 183 ~ 197)
단, 여기에서도 메모리를 할당하기 전에, UsedPAS 변수를 통해 남아있는 메모리가 있는지 확인을 해주었습니다.
(라인 185 ~ 188 : 메모리가 없다면 break를 통해 시뮬레이션 종료)

이러한 작동을 REF를 다 살피거나 PAS가 다 소모될 때까지 while문을 통해 진행합니다.
모두 진행이 끝나 시뮬레이션이 종료되면 이 시뮬레이션이 Out of Memoey로 인해 종료한 것인지
확인하여 출력한 다음,
최종적으로 프로세스 정보와 그리고 Lv1과 Lv2의 페이지 테이블 정보, 최종 레포트를 출력합니다.

Out of Memory나 프로세스 정보, 최종 레포트 출력은 os3-1과 동일합니다.
그에 반해 페이지 테이블의 출력의 경우는 약간 다른 점이 있었습니다.

```
225         cur_pte_L1 = (pte *) &PAS[i*1];
226
227         for( j = 0 ; j < 8 ; j++)
228         {
229             if(cur_pte_L1[j].vflag == 1)
230             {
231                 fprintf(stdout, "(L1PT) %03d -> %03d\n" , j, cur_pte_L1[j].frame);  // L1 출력
232
233                 cur_pte_L2 = (pte *) &PAS[cur_pte_L1[j].frame];
234
235                 for( k = 0 ; k < 8 ; k++)
236                 {
237                     if(cur_pte_L2[k].vflag == 1)
238                     {
239                         fprintf(stdout, "(L2PT) %03d -> %03d REF=%03d\n" , j*8+k, cur_pte_L2[k].frame, cur_pte_L2[k].ref); // L2 출력
240
241                     }
242                 }
243             }
244
245         }
```

위 코드는 출력 부분에서 Page Table의 출력 부분만 가져온 코드입니다.
여기에서 i는 프로세스의 pid를 의미하고, j는 Lv1 페이지 테이블의 index를 의미합니다.
그리고 k는 Lv2 페이지 테이블의 index를 의미합니다.
라인 239를 보면 page number 출력 인자로 j*8 + k 가 쓰여 있습니다.
이는 Lv2 Demand paging 동작에서 (위에서 설명한 코드 내에서) 어떻게 페이지테이블에 접근했었는지 생각해 보면 이해하기 쉽습니다. 저희는 먼저 접근하고자 하는 Page를 8로 나누어 Lv1 페이지 테이블의 인덱스로 접근하였습니다. 그 이후 Lv2 페이지 테이블에 접근할때는 % 나머지 연산을 통해 접근할 테이블 인덱스를 정했습니다.
이는 어찌 보면 8진수 표기법과 이론이 동일합니다.
즉 Lv1 index에는 8을 곱하고 Lv2 index를 더하면 원래 Page number가 나오는 것입니다.

예를 들어, 10은 Lv1 -> 10/8 = 1, Lv2 -> 10%8 = 2 이므로 1*8 + 2 = 10이 나옵니다.

이 부분이 출력에서 과제 os3-1과 다른 점입니다.

실행 결과 :
과제 3-1 (os3-1) 결과

```
ubuntu@201619460:~/hw3$ gcc -Wall os3-1.c && cat test3.bin | ./a.out
** Process 000: Allocated Frames=013 PageFaults/References=005/008
017 -> 024 REF=001
050 -> 022 REF=001
051 -> 019 REF=002
052 -> 016 REF=002
053 -> 021 REF=002
** Process 001: Allocated Frames=013 PageFaults/References=005/007
004 -> 018 REF=002
005 -> 023 REF=001
006 -> 020 REF=001
007 -> 017 REF=002
021 -> 025 REF=001
Total: Allocated Frames=026 Page Faults/References=010/015
```

과제 3-1 Jota 통과

Submission of 2021운영체제 과제 3-1 by os201619460

View source
Resubmit

Compilation Warnings

```
oshw31c.c: In function 'main':
oshw31c.c:94:9: warning: ignoring return value of 'fread', declared with attribute warn_unu
        fread(processArr[processnum]->pcess.references,processArr[processnum]->pcess.r
        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Execution Results

✔ ✔ ✔ ✔ ✔

- **Test case #1:** AC [0.049s,816.00 KB] (2/2)
- **Test case #2:** AC [0.041s,816.00 KB] (2/2)
- **Test case #3:** AC [0.048s,816.00 KB] (2/2)
- **Test case #4:** AC [0.041s,816.00 KB] (2/2)
- **Test case #5:** AC [0.041s,816.00 KB] (2/2)

Resources: 0.219s, 816.00 KB
Final score: 10/10 (10.0/10 points)

과제 3-2 (os3-2) 결과

```
ubuntu@201619460:~/hw3$ gcc -Wall os3-2.c && cat test3.bin | ./a.out
** Process 000: Allocated Frames=008 PageFaults/References=007/008
(L1PT) 002 -> 012
(L2PT) 017 -> 013 REF=001
(L1PT) 006 -> 002
(L2PT) 050 -> 010 REF=001
(L2PT) 051 -> 007 REF=002
(L2PT) 052 -> 003 REF=002
(L2PT) 053 -> 009 REF=002
** Process 001: Allocated Frames=008 PageFaults/References=007/007
(L1PT) 000 -> 004
(L2PT) 004 -> 006 REF=002
(L2PT) 005 -> 011 REF=001
(L2PT) 006 -> 008 REF=001
(L2PT) 007 -> 005 REF=002
(L1PT) 002 -> 014
(L2PT) 021 -> 015 REF=001
Total: Allocated Frames=016 Page Faults/References=014/015
```

과제 3-2 Jota 통과

Submission of 2021운영체제 과제 3-2 by os201619460

View source
Resubmit

Compilation Warnings

```
oshw32c.c: In function 'main':
oshw32c.c:94:9: warning: ignoring return value of 'fread', declared with attribute warn_unu
        fread(processArr[processnum]->pcess.references,processArr[processnum]->pcess.r
        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Execution Results

✔ ✔ ✔ ✔ ✔

- **Test case #1:** AC [0.049s,816.00 KB] (2/2)
- **Test case #2:** AC [0.037s,816.00 KB] (2/2)
- **Test case #3:** AC [0.045s,816.00 KB] (2/2)
- **Test case #4:** AC [0.049s,816.00 KB] (2/2)
- **Test case #5:** AC [0.037s,816.00 KB] (2/2)

Resources: 0.217s, 816.00 KB
Final score: 10/10 (10.0/10 points)

자율 진도표 :

1. 프로세스 저장 형식 관련 고민	Linux List, 임시변수 (출력만 하고 저장 없이), 배열	O (배열)
2. 프로세스 정보(pid, ref 등) 로드받기	변수 선언 개수 최소화 고려.	O
3. PAS 구현	동적 할당 사용.	O
4. PAS에 각 프로세스별 페이지 테이블 할당 구현	UsedPAS 변수 선언.	O
5. 할당 된 table 초기화	cur_pte 포인터 선언	O
6. 어떻게 진행할지? - whlie 조건 어떻게?	최대 Reference인 255까지 진행, 또는 실행 도중 PAS가 남아있지 않으면 종료.	O
7. 어떻게 진행할지? - 작동 방법?	각 REF마다 프로세스를 갖고 있는 배열을 통해 모든 프로세스 page table 탐색.	O
8. Page fault 구현	pte의 vflag 이용.	O
9. 메모리 아웃 경우 구현	page fault일 때와 메모리 없을 때 고려.	O
10. 동적 할당 해제	빠지지 않게 조심.	O
11. 정보 결과 값 출력하기	프로세스 출력 시 필요 변수 선언, 페이지 테이블 정보 값 출력, vflag 유효한 것만 출력	O
12. 과제 3-1		O
13. 과제 3-1에서 과제 3-2 달라지는 시작 조건 파악	8 Fraame 할당하던 테이블을 1 Frame 할당. 테이블 초기화 64 -> 8로 단축	O
14. 레벨 1 테이블 구현		O
15. 레벨 2 테이블 구현		O
16. 정보 결과 출력 수정	변수 cur_pte_L2의 필요성 인지, 테이블 출력 시 인덱스 변화 고려.	O
17. 과제 3-2		O

진행하며 :
기본적으로 과제를 진행하며 어려워서 고민했던 주요한 부분은 대부분 코드 설명 란에 함께 기재하였습니다.
그렇기 때문에 이 페이지에서는 간략히 자율 진도표를 진행할 때 어떠한 생각과 고민을 했는지 살짝 정리만 하도록 하겠습니다.

1. 프로세스 저장 형식 관련
위 os3-1 코드에서도 언급을 하지만, 처음에 프로세스를 어떻게 저장하고 관리해야하나 고민하였습니다. 처음엔 과제2처럼 Linux List로 관리해야하는가 고민을 하였으나, 되도록 교수님이 쓰신 구조체를 동일하게 모방하여 사용하고 싶었고, 이에 main 함수 자체에 임시 변수들을 선언하거나 배열을 선언하여서 관리하는거에 대해 생각을 하였습니다. 개인적으로 main에 변수를 선언한다면 생각보다 많이 코드가 복잡하고 지저분해질것이라 생각하였기에, 결국 배열을 선택하게 되었습니다.

2. 프로세스 정보(pid, ref 등) 로드받기
교수님께서 수업시간에 fread할 때 필요 이상으로 반복해서 변수에 복사를 하는 행위를 하는 일부 학생이 있다고 지적해준 적이 있습니다. 물론 저를 칭한 것은 아닐 수 있지만, 저도 지난 과제를 생각해보니 조금 불필요하게 변수가 많이 선언 된 것 같다고 생각을 하게 되었고, 이에 변수를 효율적으로 사용할 방법을 생각해 보았습니다.

4. PAS에 각 프로세스별 페이지 테이블 할당 구현
전북대 Old LMS에 가면 교수님께서 적어놓은 글이 있습니다. 바로 이거입니다.

> 실제로 PAS 내에 있는 메모리 공간에 pagetable 이 적재 되어야 합니다.
> 이 부분은 JOTA 에서 체크가 어려운 부분이라, 제가 직접 코드를 보며 판단할 예정입니다.

이에 PAS에 페이지 테이블을 할당하고 접근하는 방법에 대해 생각해보았습니다. 그러던 중 과제설명 영상 마지막 페이지(과제 주안점 페이지)에서 pte를 이용하여 포인터 선언한 예시를 보았고, 이에 많은 힌트를 얻었습니다. 사실상 ppt속 코드 그대로 채용한 것이라 봐도 무방합니다.
(pte* cur_pte = (pte*) &pas[frame_number]; <- 이 코드

5. 할당 된 table 초기화
4번 내용과 겹치는 부분입니다.

6. 어떻게 진행할지? - whlie 조건 어떻게?
1번 배열을 선택하게된 것도 그렇고 이번 while 조건도 그렇고 과제 설명영상에 주어진 조건이 많은 힌트가 되었습니다. bin 파일에 들어오는 자료들이 모두 범위가 지정되어있었기에, 이런 코드 작동기전을 짤 때, 많은 도움이 되었습니다. 그중 하나가 이 부분입니다.

9. 메모리 아웃 경우 구현
의외로 이 부분에서 무지 헤맸습니다. 코드를 돌려 보았을 때 메모리가 다 소모되지 않는 경우라면 무조건 정답이 되었는데, 메모리가 다 차서 시뮬레이션이 종료되는 경우에는 꼭 Ref라던가 Page Fault라던가 꼭 둘 중 하나는 1씩, 2씩 출력 결과가 다르게 나와 오답이 되었기 때문입니다. 이에 관련 되서는 LMS에 질문도 남기고 여러 가지로 많이 고민하였습니다. 아마 코드를 짜며 가장 오랜 시간 고민한 부분이 아닐까 싶습니다.

16. 정보 결과 출력 수정 - Lv2 테이블 정보 출력
처음에 이 부분을 구현할 땐 강의영상에서 “과제3-1과 동일하게 출력하면 된다.”(34분 5초경) 라고만 간단히 말씀하시고 그냥 빠르게 넘어가셔서, Lv2 Page Table의 인덱스를 그대로 출력하게 구현했었습니다. 말 그대로 과제3-1과 동일하게 출력하도록 코드 자체를 복사하여 그대로 채용하였던 것입니다. 이때 이 코드를 Jota에 제출을 했었는데, “어 분명 reference와 page Fault는 맞는데? 왜 틀리다고 나오지?” 라며 Page number가 잘못 된 걸 깨닫지 못하고 reference와 page fault 오류만 찾아보느라 시간을 소비하였던 기억이 있습니다.