운영체제 실습

[Assignment #4]

Class : 목 3

Professor : 최상호

Student ID : 2020202031

Name : 김재현

Introduction

4-1 은 PID 를 기반으로 특정 프로세스의 상세 정보를 출력하는 Linux 커널 모듈을 작성해 봄으로써, Linux 시스템의 메모리 구조와 프로세스 관리에 대한 이해를 심화하는 것을 목표로 합니다.

4-2 는 OPTIMAL, FIFO, LRU, CLOCK 등의 다양한 페이지 교체 알고리즘을 구현하고 분석하는 것을 목표로 합니다. 이 알고리즘들을 시뮬레이션함으로써, 각 정책의 동작 방식, 성능 지표, 그리고 메모리 관리 효율성을 보다 깊이 이해할 수 있습니다.

결과화면

4-1.

struct task_struct 에는 다양한 멤버변수가 존재합니다.

1. 프로세스 식별 및 상태 관련 멤버변수입니다.

   pid: 프로세스의 id 를 나타냅니다.

   tgid: 프로세스 그룹아이디입니다.

   state: 프로세스의 상태를 나타냅니다. (TASK_RUNNING, TASK_INTERRUPTIBLE 등)

   flags: 프로세스의 플래그 정보를 나타냅니다.

2. 스케쥴링 관련

   prio: 현재 프로세스의 유효 우선순위를 나타냅니다. 0~99 는 실시간 프로세스의 우선순위고, 100~139 는 일반 프로세스의 우선순위입니다.

3. 메모리 관리 관련

   mm: 유저 모드 주소 공간입니다.

   active_mm: 실행 중인 주소 공간입니다.

   stack: 커널 스택의 시작 주소입니다.

   이외에도 여러 정보를 나타내는 멤버변수가 많지만, 이번 실습에서는 메모리 관리 관련 멤버변수인 struct mm_struct 를 다룹니다.

mm_struct 에도 다양한 멤버 변수가 존재합니다. 대표적인 것만 알아보자면,

1. mmap: vm_area_struct list 의 시작 주소입니다.
2. start_code, end_code: 코드 세그먼트 영역입니다.

3. start_data, end_data: 데이터 세그먼트 영역입니다.
4. start_brk, brk: 힙 세그먼트 영역입니다.
5. start_stack 스택의 세그먼트 영역의 시작 위치입니다.

마지막으로 vm_area_struct 입니다.

struct vm_area_struct 는 가상메모리를 분할하여 관리하기 위한 데이터 구조입니다.
vm_area_struct 의 대표적인 멤버 변수를 알아보자면,

1. vm_mm: 해당 가상 메모리 영역을 사용하고 있는 mm_struct 구조체를 가리키는 포인터 변수입니다.
2. vm_start: 해당 가상메모리 영역 vm_area_struct 의 시작 주소입니다.
3. vm_end: 해당 가상메모리 영역 vm_area_struct 의 끝 주소입니다.
4. vm_next: vm_area_struct list 에서 다음 vm_area_struct 를 가리키는 포인터 변수입니다. 마지막 vm_area_struct 의 vm_next 는 NULL 값을 가집니다.
5. 결론은, vma = mm_struct->mmap 부터 vma->vm_next 로 반복해서 이동하다 보면 가상메모리의 전체 영역을 방문할 수 있습니다.

```
33      // find process
34      task = pid_task(find_vpid(pid), PIDTYPE_PID);
35      if (task == NULL) {
36          printk(KERN_INFO "Process with PID %d not found.\n", pid);
37          return -1;
38      }
```

pid_task() 함수를 통해, 특정 pid 를 가진 프로세스의 task_struct 를 가져옵니다.

```
40      mm = get_task_mm(task);
41      if (mm == NULL) {
42          printk(KERN_INFO "Process %d has no memory structure.\n", pid);
43          return -1;
44      }
```

get_task_mm 함수를 통해 특정 pid 를 가진 프로세스의 user address 를 가져옵니다.

```
46      printk(KERN_INFO "######## Loaded files of a process '%s(%d)' in VM ########\n", task->comm, pid);
```

해당 프로세스의 프로세스명과, pid 를 출력합니다.

```
48      // go around every vma
49      for (vma = mm->mmap; vma; vma = vma->vm_next) {
50          // get file direction
51          file = vma->vm_file;
52          if (file == NULL)
53              continue;
54
55          path = d_path(&file->f_path, buf, sizeof(buf));
56          printk(KERN_INFO "mem(%lx-%lx) code(%lx-%lx) data(%lx-%lx) heap(%lx-%lx) %s\n",
57              vma->vm_start, vma->vm_end,
58              vma->vm_mm->start_code, vma->vm_mm->end_code,
59              vma->vm_mm->start_data, vma->vm_mm->end_data,
60              vma->vm_mm->start_brk, vma->vm_mm->brk,
61              path);
62      }
63      printk(KERN_INFO "##############################################\n");
```

첫 번째 네모는, 모든 vm_area_struct 를 방문하는 것입니다. 마지막 vm_area_struct 의 vm_next 는 null 일 것이므로, vma 가 null 이 되는 순간 반복문을 탈출합니다.

두 번째 네모는, vma 의 vm_file 을 참조합니다. 해당 vma 가 vm_file 이 존재하지 않을 수도 있으므로, 조건문으로 존재유무를 체크해줍니다. vm_file 을 참조하는 이유는 해당 vma 의 파일경로를 알기 위해 d_path 함수를 쓰기 위함입니다.

세 번째 네모는, 각 vma 의 메모리 범위와, vm_area_struct 로 구성된 vma 부모 mm_struct 의 code, data, heap 영역주소를 확인하는 코드입니다.

```
[ 3892.800859] ######## Loaded files of a process 'a.out(19048)' in VM ########
[ 3892.800869] mem(5567a394b000-5567a394c000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /home/os2020202031/Assignment4-1/a.out
[ 3892.800872] mem(5567a394c000-5567a394d000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /home/os2020202031/Assignment4-1/a.out
[ 3892.800874] mem(5567a394d000-5567a394e000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /home/os2020202031/Assignment4-1/a.out
[ 3892.800876] mem(5567a394e000-5567a394f000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /home/os2020202031/Assignment4-1/a.out
[ 3892.800878] mem(5567a394f000-5567a3950000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /home/os2020202031/Assignment4-1/a.out
[ 3892.800880] mem(7f74e93d0000-7f74e93f2000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
[ 3892.800882] mem(7f74e93f2000-7f74e956a000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
[ 3892.800884] mem(7f74e956a000-7f74e95b8000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
[ 3892.800886] mem(7f74e95b8000-7f74e95bc000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
[ 3892.800888] mem(7f74e95bc000-7f74e95be000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /usr/lib/x86_64-linux-gnu/libc-2.31.so
[ 3892.800890] mem(7f74e95d4000-7f74e95d5000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /usr/lib/x86_64-linux-gnu/ld-2.31.so
[ 3892.800892] mem(7f74e95d5000-7f74e95f8000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /usr/lib/x86_64-linux-gnu/ld-2.31.so
[ 3892.800894] mem(7f74e95f8000-7f74e9600000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /usr/lib/x86_64-linux-gnu/ld-2.31.so
[ 3892.800896] mem(7f74e9601000-7f74e9602000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /usr/lib/x86_64-linux-gnu/ld-2.31.so
[ 3892.800898] mem(7f74e9602000-7f74e9603000) code(5567a394b000-5567a394c215) data(5567a394edb0-5567a394f010) heap(5567a4447000-5567a4447000) /usr/lib/x86_64-linux-gnu/ld-2.31.so
[ 3892.800899] ##############################################
os2020202031@ubuntu:~/Assignment4-1$
```

출력결과를 보면, 모든 vma 의 code, data, heap 영역의 주소가 동일하게 출력되는 것을 확인할 수 있습니다. 이를 통해 vma 가 mm_struct 를 구성하는 수많은 vm_area_struct 중 하나이기에, 각각의 메모리영역은 다를지라도, 부모인 mm_struct 는 동일하다는 것을 알 수 있습니다.

4-2

```
281     // read the number of frame count
282     fgets(buf, sizeof(buf), rfile);
283     frameCount = atoi(buf);
284
285     // read page information
286     fgets(buf, sizeof(buf), rfile);
287     pageCount = strlen(buf) / 2 + 1;
288     ptr = strtok(buf, " ");
289     for(int i = 0; ptr != NULL; i++)
290     {
291         pages[i] = atoi(ptr);
292         ptr = strtok(NULL, " ");
293     }
294
295     OPT(pages, pageCount, frameCount);
296     FIFO(pages, pageCount, frameCount);
297     LRU(pages, pageCount, frameCount);
298     CLOCK(pages, pageCount, frameCount);
299
300     return 0;
301 }
```

먼저 main 함수입니다.

첫 번째 네모는 입력파일 첫째줄에 입력된 page frame size 를 정수로 저장하는 부분입니다.(frameCount)

두 번째 네모는 입력파일 둘째줄에 입력된 page reference string 의 page 개수 pageCount 를 구하고, string 을 공백으로 parsing 하여 정수배열 pages 로 만드는 부분입니다.

세 번째 네모는 구현한 OPT, FIFO, LRU, CLOCK 알고리즘을 호출하는 부분입니다.

OPT replacement policy 는 앞으로 사용될 페이지 중에서 가장 늦게 사용되는 페이지를 frame 에서 evict 하고 지금 당장 사용될 페이지를 frame 에 넣는 정책입니다.

finOptimalFrameToReplace 는 evict 될 페이지를 고르는 함수입니다.

```
7  int findOptimalFrameToReplace(int *pages, int *frames, int frameCount, int pageCount, int currentIndex) {
8      int farthest = currentIndex;
9      int frameToReplace = -1;
10
11     for (int i = 0; i < frameCount; i++) {
12         int j;
13         for (j = currentIndex; j < pageCount; j++) {
14             if (frames[i] == pages[j]) {
15                 // if the page appears later comparing previous pages
16                 if (j > farthest) {
17                     farthest = j;
18                     frameToReplace = i;
19                 }
20                 break;
21             }
22         }
23         // it never appears in pages
24         if (j == pageCount) {
25             frameToReplace = i;
26             break;
27         }
28     }
29
30     // all future pages don't exist in frames
31     if (frameToReplace == -1)
32         return 0; // just return 0
33     else
34         return frameToReplace;
35 }
```

pages 는 pages 정수배열, frames 는 frames 정수배열, frameCount 는 frames 개수, pageCount 는 pages 개수, currentIndex 는 pages 에서 OPT 정책을 위해 확인해야할 첫번째 페이지의 index 입니다.

farthest 는 frames 를 방문하면서 확인된 페이지 중, currentIndex 에서 가장 멀리 떨어진 index 입니다. frameToReplace 는 교체되어야 할 frame 의 index 입니다.

세 번째 네모는 frames 를 처음부터 끝까지 방문합니다.

네 번째 네모는 pages 를 currentIndex 부터 끝까지 방문합니다. 현재 프레임과 일치하는 페이지가 존재하고, 그 페이지가 현재까지 확인된 페이지 중에서 가장 멀리 있다면, farthest 와 frameToReplace 를 업데이트합니다.

다섯 번째 네모는 예외를 처리합니다. 만약 현재 프레임과 일치하는 페이지가 존재하지 않아서 반복문을 끝까지 돌아버린다면, j==pageCount 가 될 것입니다. 이는 해당 프레임은 앞으로 다시는 쓰일 일이 없다는 것을 의미합니다. 이 경우엔 frameToReplace 를 해당 프레임으로 업데이트합니다.

여섯 번째 네모는 frameToReplace 를 return 합니다. 단, 만약 frameToReplace 가 그대로 초기값 -1 과 동일하다면, frame 에는 page 와겹치는 부분이 단 하나도 존재하지 않다는 뜻이므로, 단지 0 을 반환합니다.

다음은 OPT 알고리즘 구현입니다.

```
43      // frames is empty
44      for (int i = 0; i < frameCount; i++)
45          frames[i] = -1;
```

처음엔 frames 엔 어떤 pages 도 존재하지 않으므로 -1 로 초기화합니다.

```
47      for(int i = 0; i < pageCount; i++)
48      {
49          // check if page exists in frames
50          int valid = 0;
51          for(int j = 0; j < frameCount; j++)
52          {
53              if(pages[i] == frames[j])
54              {
55                  valid = 1;
56                  break;
57              }
58          }
59
60          // page already exists in frames
61          if(valid)
62              continue;
63
64          // there are empty frame
65          if(frameSize < frameCount)
66              frames[frameSize++] = pages[i];
67          else
68          {
69              int frameToReplace = findOptimalFrameToReplace(pages, frames, frameCount, pageCount, i + 1);
70              frames[frameToReplace] = pages[i];
71          }
72          numPageFaults++;
73      }
```

첫 번째 네모는 모든 페이지를 방문함을 의미합니다.

두 번째 네모는 모든 프레임을 방문하며, 해당 페이지와 동일한 프레임이 존재하는지를 체크합니다. 존재한다면 더 이상의 방문을 그만두고, valid 를 1 로 업데이트합니다.

세 번째 네모는 페이지가 이미 프레임에 존재한다면(valid 가 1 이라면) 다음 페이지로 넘어가는 부분입니다. 페이지가 프레임에 없다면 아래로 계속 진행합니다.

네 번째 네모는 프레임에 들어가있는 페이지가 프레임크기보다 작다면(프레임에 빈 공간이 있다면) 가장 뒷자리에 페이지를 삽입해주는 동시에 frameSize 를 증가시킵니다.(frameSize 는 프레임에 존재하는 페이지 개수입니다.) 만약 프레임이 이미 꽉 차있다면, findOptimalFrameToReplace 함수를 통해 교체할 프레임을 찾고 그 위치에 페이지를 삽입합니다.

다섯 번째 네모는 페이지폴트의 횟수를 카운트하는 numPageFaults 를 +1 합니다. 세번째 네모에서 다음 페이지로 넘어가지 못했다는 것은, 페이지폴트가 발생했다는 뜻이기 때문입니다.

다음은 FIFO page replacement policy 구현입니다.

```
87      // frames is empty
88      for (int i = 0; i < frameCount; i++)
89          frames[i] = -1;
```

처음엔 frames 가 비어있으므로 -1 로 초기화해줍니다.

```
91      for(int i = 0; i < pageCount; i++)
92      {
93          // check if page exists in frames
94          int valid = 0;
95          for(int j = 0; j < frameCount; j++)
96          {
97              if(pages[i] == frames[j])
98              {
99                  valid = 1;
100                 break;
101             }
102         }
103
104         // page already exists in frames
105         if(valid)
106             continue;
107
108         // there are empty frame
109         if(frameSize < frameCount)
110         {
111             frames[frameEnd] = pages[i];
112             frameEnd = (frameEnd + 1) % frameCount;
113             frameSize++;
114         }
115         else
116         {
117             frames[frameEnd] = pages[i];
118             frameStart = (frameStart + 1) %frameCount;
119             frameEnd = (frameEnd + 1) % frameCount;
120         }
121         numPageFaults++;
122     }
```

페이지가 프레임에 존재하는지 확인하는 부분은 OPT 와 동일합니다.

그러나 FIFO 는 queue 를 사용하여 구현해야 하기 때문에 페이지가 프레임에 존재하지 않는 경우는 구현이 다릅니다. frameStart, frameEnd 라는 변수를 사용하여 frames 배열을 마치 원형큐처럼 동작하도록 구현했습니다.

두번째 네모는 프레임에 빈 공간이 존재하는 경우와 프레임이 꽉 찬 경우를 나누어 구현합니다. 프레임에 빈 공간이 존재하는 경우, frameEnd index 에 페이지를 삽입하고,

frameEnd 와 frameSize 를 +1 해줍니다. 프레임이 꽉 차 있는 경우는, frames 에 가장 먼저 들어온 frameStart index 에 위치한 프레임을 evict 하고, frameEnd index 에 페이지를 삽입합니다. frameStart, frameEnd 를 +1 해줍니다.

%frameCount 연산을 진행하는 이유는 index 가 프레임의 크기보다 커질 경우 0 으로 돌아와야 하기 때문입니다.

세 번째 네모는 역시 마찬가지고 페이지폴트 횟수를 +1 합니다.

다음은 LRU page replacement policy 입니다.

```
155         // page already exists in frames
156         if(valid)
157         {
158             // move frame that corresponds to this page to top
159             int tmp = frames[j];
160             int k, l;
161 
162             k = j;
163             l = (j + 1) % frameCount;
164             while(l != frameEnd)
165             {
166                 frames[k] = frames[l];
167                 k = l;
168                 l = (l + 1) % frameCount;
169             }
170             if(frameEnd == 0)
171                 frames[frameCount - 1] = tmp;
172             else
173                 frames[frameEnd - 1] = tmp;
174             continue;
175         }
176 
177         // there are empty frames
178         if(frameSize < frameCount)
179         {
180             frames[frameEnd] = pages[i];
181             frameEnd = (frameEnd + 1) % frameCount;
182             frameSize++;
183         }
184         else
185         {
186             frames[frameEnd] = pages[i];
187             frameStart = (frameStart + 1) % frameCount;
188             frameEnd = (frameEnd + 1) % frameCount;
189         }
190         numPageFaults++;
191     }
192 
```

전체적인 구조는 동일하나, valid 가 1 일 때 변화가 있습니다. LRU policy 역시 FIFO 와 동일하게 queue 를 사용하여 구현했습니다. queue 는 선입선출이라는 특성이 참조된지 가장 오래된 페이지를 evict 하는 LRU policy 와 유사하기 때문입니다.

첫 번째 네모는 해당 페이지가 프레임에 존재할 때, 페이지를 queue 의 맨 뒤로 보내는 동작을 추가했습니다. 페이지가 프레임의 중간에 존재한다면, 프레임의 중간에 빈 공간이 생기므로, 해당 index 뒤에 존재하는 프레임들을 모두 한 칸씩 앞으로 당기는 작업을

통해 빈 공간을 제거했습니다. 이로써 참조된지 가장 오래된 페이지를 evict 하는 queue 가 구현됐습니다.

두 번째 네모는 FIFO 와 동일합니다. 페이지가 프레임에 존재하지 않는다면 큐의 제일 앞 부분이 참조된지 가장 오래된 프레임이라는 뜻과 같기 때문입니다.

세 번째 네모는 페이지폴트 횟수를 +1 해줍니다.

마지막으로 CLOCK page replacement policy 입니다.

```
203         // frames is empty
204         for (int i = 0; i < frameCount; i++)
205             frames[i] = -1;
206 
207         // use bit is empty
208         for (int i = 0; i < frameCount; i++)
209             useBit[i] = 0;
```

프레임은 처음에 비어있으므로 -1 로 초기화합니다.

use bit 또한 처음엔 0 으로 초기화합니다.

```
211     for(int i = 0; i < pageCount; i++)
212     {
213         // check if page exists in frames
214         int valid = 0;
215         int j;
216         for(j = 0; j < frameCount; j++)
217         {
218             if(pages[i] == frames[j])
219             {
220                 valid = 1;
221                 break;
222             }
223         }
224 
225         // page already exists in frames
226         if(valid)
227         {
228             useBit[j] = 1;
229             continue;
230         }
231 
232         // there are empty frame
233         if(frameSize < frameCount)
234         {
235             frames[frameEnd] = pages[i];
236             frameEnd = (frameEnd + 1) % frameCount;
237             frameSize++;
238         }
239         else
240         {
241             while(useBit[useArrow] == 1)
242             {
243                 useBit[useArrow] = 0;
244                 useArrow = (useArrow + 1) % frameCount;
245             }
246             frames[useArrow] = pages[i];
247             useArrow = (useArrow + 1) % frameCount;
248         }
249         numPageFaults++;
250     }
```

FIFO policy 와 동일한데 네모친 부분만 다릅니다.

첫 번째 네모는 페이지가 프레임에 존재하는데 참조된 경우, use bit 를 1 로 업데이트 하는 부분을 추가했습니다.

두 번째 네모는 프레임이 full 일 때, use bit 가 1 이라면 use bit 를 0 으로 초기화하고, arrow 를 +1 합니다(arrow 는 evict 할 프레임을 가리키는 역할). use bit 가 0 이라면 arrow 가 가리키는 프레임을 evict 하고 해당 index 에 페이지를 삽입합니다.

```
printf("Clock Algorithm:\n");
printf("Number Of Page Faults : %d\n", numPageFaults);
printf("Page Fault Rate: %.2lf%%\n\n", (double)numPageFaults / pageCount * 100);
```

모든 policy 의 성능은 다음과 같이 출력합니다.

```
os2020202031@ubuntu:~/Assignment4-2$ ./page_replacement_simulator input.1
Optimal Algorithm:
Number Of Page Faults : 7
Page Fault Rate: 53.85%

FIFO Algorithm:
Number Of Page Faults : 10
Page Fault Rate: 76.92%

LRU Algorithm:
Number Of Page Faults : 9
Page Fault Rate: 69.23%

Clock Algorithm:
Number Of Page Faults : 8
Page Fault Rate: 61.54%
```

input.1:

3

7 0 1 2 0 3 0 4 2 3 0 3 2

```
os2020202031@ubuntu:~/Assignment4-2$ ./page_replacement_simulator input.2
Optimal Algorithm:
Number Of Page Faults : 5
Page Fault Rate: 50.00%

FIFO Algorithm:
Number Of Page Faults : 7
Page Fault Rate: 70.00%

LRU Algorithm:
Number Of Page Faults : 6
Page Fault Rate: 60.00%

Clock Algorithm:
Number Of Page Faults : 6
Page Fault Rate: 60.00%
```

input.2:

3

1 2 3 4 2 1 2 1 3 1

```
os2020202031@ubuntu:~/Assignment4-2$ ./page_replacement_simulator input.3
Optimal Algorithm:
Number Of Page Faults : 8
Page Fault Rate: 57.14%

FIFO Algorithm:
Number Of Page Faults : 10
Page Fault Rate: 71.43%

LRU Algorithm:
Number Of Page Faults : 10
Page Fault Rate: 71.43%

Clock Algorithm:
Number Of Page Faults : 10
Page Fault Rate: 71.43%
```

input.3:

4

1 3 0 3 5 6 3 2 5 2 4 1 0 5

분석:

Optimal Algorithm:

- 이론적으로 최적의 결과를 보이며, 가장 적은 Page Fault 수를 기록했습니다.

FIFO Algorithm:

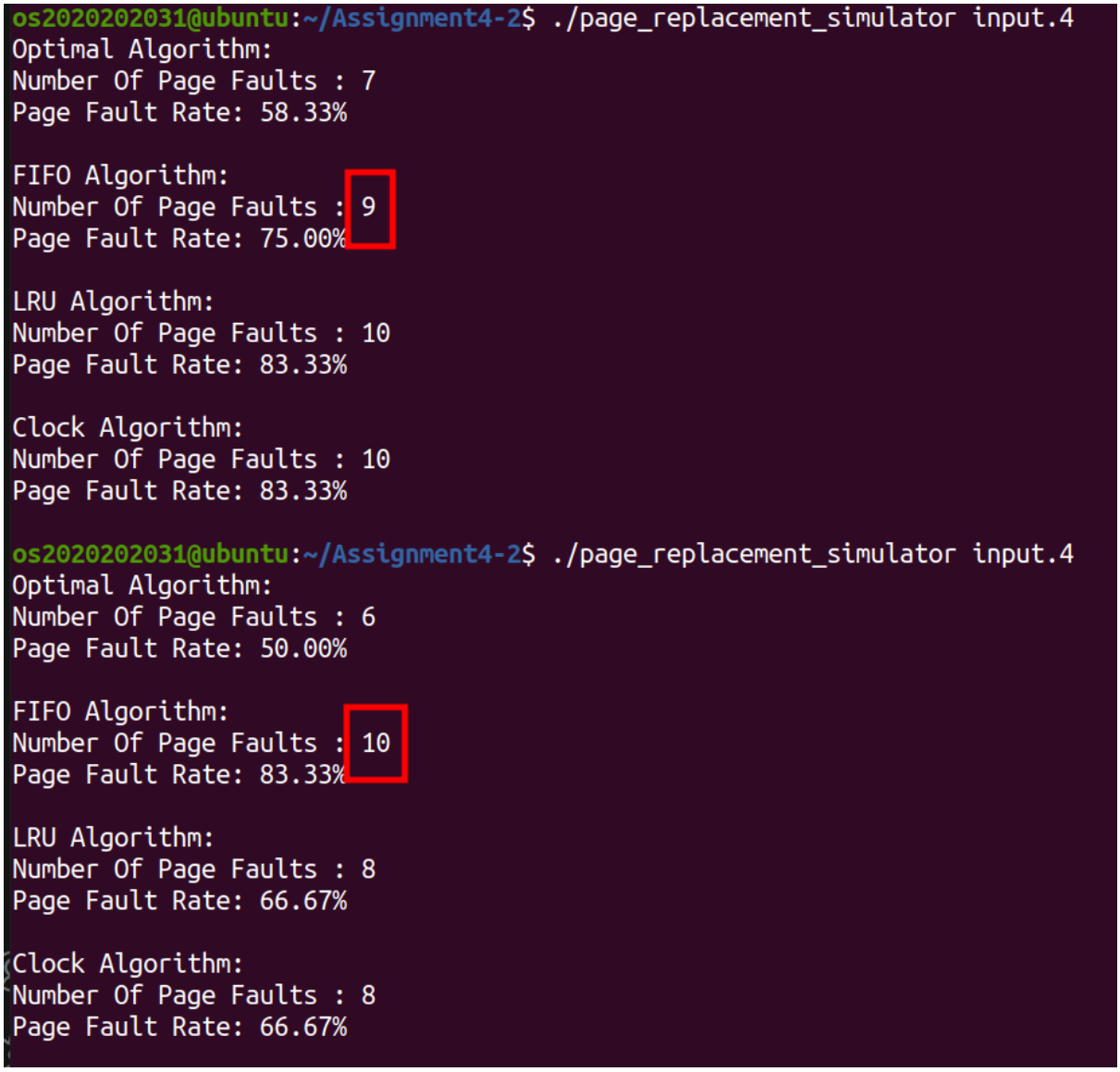
- 참조 패턴이 FIFO 의 동작 방식에 적합하지 않아 성능이 가장 낮았습니다.
- 교체할 페이지 선택에 참조 정보가 전혀 고려되지 않기 때문에 비효율적입니다.

LRU Algorithm:

- 실제로 구현 가능한 알고리즘 중 Optimal 에 가까운 성능을 보여줍니다.
- 참조 패턴을 잘 반영하여 FIFO 보다 나은 결과를 보였습니다.

Clock Algorithm:

- LRU 를 근사화한 알고리즘으로, 페이지 교체 과정에서 오버헤드가 적으면서도 효율적인 성능을 보여줍니다.
- LRU 와 Page Fault 수가 같거나 적은 결과를 보였으며, 특정 참조 패턴에서 더 나은 성능을 발휘할 수 있음을 보여줍니다.



input.4:

3 -> 4

1 2 3 4 1 2 5 1 2 3 4 5

다음은 BELADY's ANOMALY를 보이는 FIFO 예제입니다. 실제로 frame size를 4로 늘리니 Page fault 가 1 증가하는 것을 알 수 있습니다.

고찰

vm_area_struct 의 멤버변수 vm_file 이 존재하지 않는 경우가 존재했습니다. 이런 경우 d_path 를 통해 파일의 경로를 얻으려고 하면, 오류가 발생함을 알 수 있었습니다. 따라서 vm_file 이 존재하는 경우만 출력하도록 예외를 처리했습니다.

OPT, FIFO, LRU, CLOCK 순서로 알고리즘을 구했습니다.

진짜 알고리즘을 구현하는 것이 아닌 알고리즘 성능평가를 구현하는 것이다 보니, 대체로 기본적인 구조는 비슷하고, 세부적인 구현만 달라지는 것을 알 수 있습니다. 비록 성능평가 구현이었지만 그 과정에서 page replacement policy 들에 대해 보다 더 자세히 알게됐습니다.

Reference

d_path 에 대하여

https://archive.kernel.org/oldlinux/htmldocs/filesystems/API-d-path.html