**Operating System**

**- 3차과제 -**

**가상 메모리의 개념 이해 및 procfs를 활용한**

**메모리정보 출력 프로그램 구현**

**컴퓨터학과**

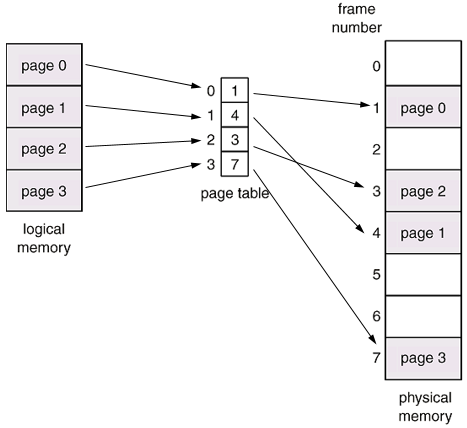
**2013210107**

**김정수**

**2016.06.11. 제출**

1. **개발환경**

Oracle VM VirtualBox에서 배포판 리눅스 Ubuntu 14.04 LTS를 사용  
새로 컴파일한 Kernel Version 3.14.63

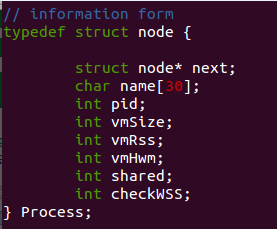
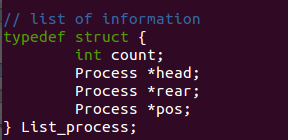
1. **이론 설명**
2. **리눅스에서 Paging**  
   메모리를 관리할 때 중요한 개념은 물리 메모리와 가상 메모리 이 두 가지다.  
   물리 메모리(Physical Memory)는 실제로 정보를 가지고 있고, 프로그램이 실행될 때 이 곳에 저장된 데이터를 가져와 일을 수행하게 된다. 보통 RAM이라고 부르는 주기억장치가 바로 이것이다.  
   가상 메모리(Virtual Memory)는 실제 존재하지는 않지만 사용자에게 메모리로서의 역할을 하는 메모리이다. 프로세스가 수행되기 위해서 프로그램의 모든 부분이 물리 메모리에 있을 필요는 없으며, 현재 실행되고 있는 code부분만 실제 메모리에 올려져 있으면 된다. 이를 효율적으로 관리하기 위하여 가상 메모리를 사용하게 되는 것이다.  
   Paging이란, 주소 공간을 동일한 크기의 Page로 나누어 관리하는 것을 의미한다.  
   보통 1page는 4kB로 나누어 사용한다. 물리 메모리에서는 하나의 페이지(=block)을 프레임(Frame)이라고 부르며, 가상 메모리에서는 그대로 페이지(Page)라고 부른다. 명칭만 다를 뿐 같은 개념이다. 페이지는 페이지 번호(Page number)와 각 페이지의 내부 주소를 가리키는 페이지 주소(Page offset)로 구성되어있다.  
   이제 주소공간을 할당하고 불러오고 관리할 때는 페이지 단위로 관리하게 된다.  
   사용자 쪽에서는 가상 메모리를 다루게 되며, 내부적으로 이 가상 메모리 주소는 실제 해당하는 물리 메모리 주소와 mapping 되게 된다. 이 mapping과정은 Page Table에 의하여 관리된다.  
     
   왼쪽 그림과 같이 Logical Memory는 사용자 입장에서 보이는 page  
   number이며, 실제로는 해당되는 실제 메모리에서의 frame number가 페이지 테이블 안에 기록되어있다.  
   page offset의 경우에는 같은 값을 사용하므로, page number만 페이지 테이블을 통하여 맵핑하게 된다.
3. **Working Set**  
   Paging 방법에서는 실제 데이터로의 접근이 페이지 테이블에 한 번, 물리 메모리 내의 데이터에 한 번으로 총 두 번의 메모리 접근을 거치게 된다. 이는 메모리 접근 속도를 크게 떨어뜨리게 된다(2배 더 느리게). 이를 해결하기 위해 미리 페이지 테이블을 이용해 변환된 주소를 TLB에 저장해둔다. TLB는 레지스터이기 때문에 빠른 수행이 가능하도록 만들어주지만 크기가 작아 모든 변환 주소를 저장해 둘 수 없는 단점이 있다. 따라서 Demand Paging 기법을 사용하게 된다.   
   Demand Paging이란 프로세스 실행을 위한 모든 page를 메모리에 올리지 않고, 필요한 page의 요청이 발생할 때 메모리에 올리는 기법이다. 프로세스 전체의 이미지를 메모리에 올리지 않기 때문에 실제 필요한 물리 메모리 양을 줄일 수 있는 장점이 있다.  
   단, 참조하고자 하는 page가 실제 물리 메모리에 있을 때(vaild)는 상관이 없지만, 실제 물리 메모리에 없는 경우(invalid) Page fault가 발생하게 된다.   
   그런데 일반적인 경우, 한 프로그램에서는 사용했던 주소를 계속해서 여러 번 접근하게 되는 경우가 빈번하다. 이렇게 일정하게 분산되어있지 않고 한 군데에 집중되어 있는 것을 Locality라고 하는데, 이를 활용하기 위해 Working Set이 등장하게 된다.  
   Working Set은 어떤 시간에 임의로 정해진 window동안에 접근한 page들의 집합이다.  
     
     
   위 그림은 연속적으로 접근한 page 번호를 모아 둔 것이고, 빨간 네모상자는 window를 의미한다. 0에 위치한 화살표는 현재 시간을 의미하며, window동안의 working set은 {4,5,6,7,8,0} 이다. 이 working set이 이 전에 말한 locality를 의미하는 것을 알 수 있다.  
   working set은 시간에 따라 계속해서 바뀌며, …0120123... 구간에서는 {0,1,2,3}으로 전체 크기 또한 바뀔 수 있다.  
   working set의 window 크기를 너무 크게 잡으면 지역성의 장점이 사라질 우려가 있고,  
   window의 크기를 너무 작게 잡으면 thrashing(아래 설명 참조)에 위험이 생긴다.  
   따라서 window의 크기를 적절히 상황에 따라 잘 결정해야 할 필요가 있다.
4. **Thrashing**  
   Thrashing은 Page fault가 너무 많이 발생하게 되어, 심각한 성능 저하를 초래하게 되는 일은 말한다. 다시 말해, 실제 프로그램이 수행되는 시간보다 page fault를 처리하는(backing store에서 페이지의 내용을 frame에 불러들이고, 페이지 테이블을 재구성하는 페이지 교환) 시간이 더 큰 경우를 일컫는다.  
   주로 프로그램이 활발하게 사용하는 페이지 집합보다 실제 할당 받는 프레임이 더 작을 때 계속해서 페이지 교체를 필요로 하게 되어 Thrashing이 발생하게 된다.  
   이러한 Thrashing 현상을 방지하기 위해서는 각 프로세스가 필요로 하는 최소한의 frame 수를 보장해 줄 필요가 있다. Working set은 시간에 따라 working set이 작아지면 그만큼 frame을 회수하고 커지게 되면 frame을 더 할당하여 가변적으로 메모리를 관리하게끔 만드는 frame 할당의 지표가 된다.
5. **Swap 파티션**  
   모든 유저 프로세스가 사용하는 페이지 수보다 물리 메모리의 프레임 수가 적어지는 경우를 메모리 과다 할당 상태(Over-allocating of memory)라 부른다. 다시 말해, 디스크에서 요구된 페이지를 물리 메모리의 free frame에 할당해야 하는데 사용할 수 있는 free frame이 더 이상 존재하지 않는 경우를 말한다. 이 때, 다양한 페이지 교체 알고리즘을 사용하여 교체할 프레임(victim frame)을 찾아 free frame처럼 취급하는 것이다.(Page replacement)  
   이런 page out으로도 메모리 부족을 해결하지 못할 경우, Swapping 기법을 사용하게 된다. Swapping은 swap out 대상이 된 프로세스 전체를 디스크(secondary storage)로 보내버리는 기법이다. 원래는 메모리 위에 있어야 하지만 메모리의 용량이 부족하여, 임시로 디스크 공간에 대신 저장해 두는 것이다. 디스크 공간의 일부분은 오직 이러한 페이지 swapping을 하기 위해서만 사용되는데 이 공간을 바로 swap partition이라 부른다. 이 공간을 통해 우리는 더 많은 물리적 메모리 공간을 확보할 수 있다.

**구현**

Procf 엔트리를 활용하여 필요 정보를 가져온다.  
가져온 정보는 다음과 같다.

**/proc/sys/kernel/pid\_max** : 가장 큰 pid값. 이 값을 기준으로 전체 Loop 한번의 횟수를 결정하며, index i를 0부터(실질적으로는 1) 시작하여 pid\_max까지 돌아가며 한번씩 /proc/index안에 있는 파일에 접근한다. 만약 index에 해당하는 pid가 존재하지 않으면 그냥 pass하고, 존재할 경우 일을 처리한다.  
  
C:\Users\김정수\Desktop\4-1\os\assignment3\Pictures\status.PNG  
**/proc/<pid>/status :** man page를 통해 본 각 pid에 대한 status는 stat과 statm에 대한 정보를 가지고 있다. 설명에서 볼 수 있듯, 각 주요 정보들을 사용자가 파싱하기 쉽게 모아두었다.  
  
  
**/proc/<pid>/statm** : status에는 기록되어 있지 않은, shared memory 부분을 알기 위해 이 부분을 사용하였다. Status와 다르게 각 값에 대한 정보는 출력되지 않고 단순히 그 값 자체만 출력된다. 1번째 인자는 size로 status의 vmSize와 같고, 2번째 인자는 resident set size로 status의 vmRSS와 같다. 그리고 3번째 인자가 바로 우리가 찾기 위한, shared page에 대한 정보이다. 하지만 실제 status의 값과 비교해 보면 그 값이 다름을 알 수 있는데, status는 실제 그 페이지가 차지하고 있는 전체 크기를 의미하고, statm은 그 페이지의 개수를 의미하기 때문이다. 일반적으로 1page의 크기를 4kB로 정해두는 것을 고려해 보았을 때, status의 vmSize, vmRSS와 statm의 그 값을 비교해보면 정확히 4를 곱한 값임을 알 수 있다. (status에는 역시 kB 단위 또한 정확히 명시되어 있다.)

우리가 구하고자 하는 정보는 PID, Name, Virtual memory, Physical memory, shared memory, 그리고 Working set size이다. Man page의 다음 그림을 통해 각각의 정보를 갖고 있는 변수를 순서대로 알 수 있을 것이다. (shared를 제외한 모두는 status에서 가져온다.)  
  
  
  
  
 : /proc/[PID]/statm  
  
  
Resident set size는 process에 의하여 Physical memory(RAM)에 할당된 실질적인 양이다.  
VmHWM는 특정한 시점에서의 물리메모리의 최대의 값이다. 이것이 위에서 설명한 working set size의 의미를 갖게 된다. Working set은 필요한 자원을 미리 부분적으로 가져와 기록해 두는 것인데(locality를 가지고) 프로세스가 사용하고 있는 물리메모리의 최대값 자체의 의미가 실제로 가져온 그 필요한 자원을 의미한다. 따라서 working set 이론에 의하면 필요에 따라 효율적으로 이 size가 변동될 것이고, 만일 변동된다면 추후 실제 프로그램에서 WSS에 표시하도록 한다.(0.5초간 업데이트, 1초간 표시)

**소스코드**우선 자료구조로써 임의로 Process구조와 List\_Process를 만들었다.  
 

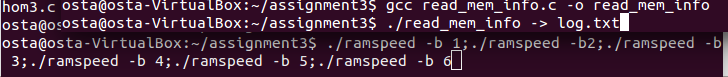
**Typedef struct Process :** 가장 기본이 되는 하나의 Process 그 자체를 의미한다.  
우선 프로세스에 대한 정보인 name, pid, vmSize, vmRss, vmHwm, shared을 가진다..  
또한 next를 통하여 Process들의 집합 안에서 다음 프로세스들을 가르키는 포인터를 넣어준다. CheckWSS의 경우는 나중에 출력 시 Working set size의 변화를 체크하여 \*를 표시하기 위해 체크하는 변수이다.  
  
**Typedef sturct List\_process :** 프로세스는 하나만 기록하는 것이 아니라, 여러 개를 가지고 있어야 할텐데, 전체 프로세스를 List형태로 가지고 있는 집합이다. 전체 프로세스의 개수 count와 처음과 끝을 의미하는 head, rear, 그리고 실질적으로 현재의 프로세스를 가리킬 pos를 가진다.

C:\Users\김정수\Downloads\AAAA (1).PNG  
**add\_process 함수** : 만일 해당하는 pid가 기존의 list에 없을 시(주로 중간에 실행된 경우), 해당 Process를 List에 추가해 준다. 추가방식은 Queue와 비슷하게 맨 뒤로 추가한다.

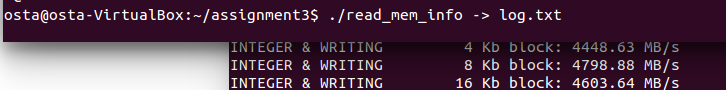
1. 우선 프로그램 실행 시, 우선적으로 가장 큰 PID를 찾아 가져온다. /proc/sys/kernel/pid\_max 파일을 열어 바로 얻어지는 정보가 바로 그 것이며 이 값을 변수 max\_PID에 저장한다.
2. 바로 전 프로그램들의 History를 기록할 프로그램들의 집합, List\_Process를 선언 및 초기화 한다. (WSS의 변화를 알아보기 위하여 History가 필요하다.)
3. 이제 실행하고자 하는 작업을 while(1)문을 통하여 계속해서 반복작업을 하여준다. 단, 미리 k를 두어 k값이 특정 값에 도달하면 break로 while문을 빠져나오게 한다. 이후 0.5초씩 프로그램을 잠재울 것을 감안해보면, 이 k값은 1당 0.5초 동안의 실행시간을 의미하게 된다.
4. 0부터 시작하여 미리 구한 max\_PID값이 될 때까지 반복해서 /proc/[PID]/에 접근한다. Fileopen에 에러가 생기면 이는 해당 pid가 존재하지 않는 것으로 간주하여 pass하고 다음 pid에 계속해서 접근하여본다. 계속해서 접근하는 파일은 status이다.
5. 하나의 pid에 대한 파일이 열렸을 때, 우선 해당 pid에 대한 history가 있는지를 알아보기 위해 pre\_process를 선언하고 초기화 시킨다. 그리고 history List에서 접근한 pid가 있는지를 검색하여 만약에 있다면 list의 pos를 해당 pid를 가리키게 한다. 또한 update 되기 전에, 미리 선언해둔 pre\_process에 기존 history의 정보를 기록한다.(0.5초전 정보)
6. 현재 프로세스의 정보를 저장해둘 노드, process를 선언하여 fget을 이용하여 한 줄 씩 가져와 token을 이용하여 name, pid, vmsize, vmRSS, vmHWM의 정보를 process에 저장한다. 모든 작업이 끝나면, 이미 해당 pid가 존재하는 것을 알고 있기 때문에 따로 확인 절차없이 /proc/[PID]/statm 파일을 열어 3번째 인자인 shared 값을 가져온다.(다른 중복된 정보는 이미 status에서 얻었다.) 이 때, page이기때문에 \*4를 하여 byte값의 표현으로 가져온다.
7. 기존 List에 해당 pid가 없었다면 add\_process를 통하여 추가하고, 있었다면 현재의 정보로 update하여 준다. 이 때, 변화가 있었는지를 표시하는 checkWSS가 0보다 크다면 1만큼 감소시킨다.
8. Pre\_process와 현재 process의 vmHWM값을 비교하여, working set size의 변화가 있었는지를 check하고 만약 변화가 있었다면 checkWSS에 2를 대입한다. (Loop 한번에 0.5초이기 때문에 이는 1초를 의미하게된다.)
9. 현재 프로세스에 대한 정보를 출력한다. 단, checkWSS의 값을 체크하여, 만약 0이 아니라면, WSS항목에 \* 를 표시하여준다. 출력 정보는 byte가 아닌 page의 수로 출력하기 위해 전체 값을 4로 나누어주었다. (shared 역시 \*4 상태의 값이므로 다시 4로 나눈다.)
10. 한 번의 for Loop가 끝나면(전체 프로세스를 한번씩 훑음), 현재 시간 Time과 현재 실행된 전체 프로세스의 수를 출력하고 0.5초 동안 쉰 후, 다시 동일한 과정을 반복한다.

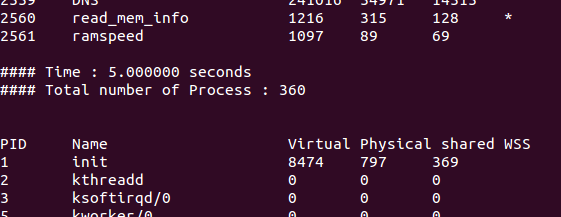
**실행결과**

read\_mem\_info를 실행 하여, log.txt에 저장한다. 동시에 다른 화면의 창에서는 ramspeed를 옵션을 이용하여 1부터 6까지 순차적으로 실행.



실행화면

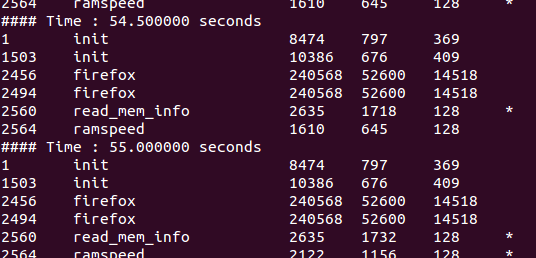


log파일 일부분 결과.  


read\_mem\_info의 경우 working set size가 바뀌어, \*이 표시됨을 알 수 있다.

이후, log.txt에서 필요한 부분만을 가져오기 위해, grep을 사용하여 다른 log파일에 저장한다.  
C:\Users\김정수\Desktop\4-1\os\assignment3\nono\log_2txt.PNG  
해당 파일에 initctl이라는 프로그램도 포함되어 이를 제거하기 위해,  
C:\Users\김정수\Desktop\4-1\os\assignment3\nono\logfinaltxt.PNG

이렇게 실행하여 마지막 log\_final.txt를 만들어준다.

**결과화면**  


* **Init, firefox, read\_mem\_info 산출물 분석**

Init과 firefox의 경우에는 일정한 virtual memory와 일정한 Physical memory를 보여준다. Shared의 경우는 모든 경우가 일정하며 아주 작은 값으로 유지된다. 이는 실제로 할당된 공유 메모리가 아니라 가상으로 할당된 공유메모리이며, 물리적인 변화가 없기 때문에 가상으로 할당된 공유 메모리 역시 아무런 변화가 없는 것으로 추측된다. Firefox의 경우는 메모리 사용량이 아주 크지만 변변화는 별로 없다. 그 이유는 프로그램 실행동안 미리 켜둔 firefox를 아무런 동작도 하지 않았기 때문에 계속해서 같은 상태를 유지하기 때문인 것으로 추측된다. 중간에 프로그램 실행도중 여러 번 firefox를 이용하여 다른 페이지를 들어가거나 하는 수행을 했을 때, network와 관련된 다른 프로그램들이 생성되고 소멸되는 것을 관찰할 수 있었다.

Read\_mem\_info는 보고서 안내서에서의 내용과 다르게 계속해서 virtual memory와 physical memory가 증가하는 경향을 보인다. 이는, 아마도 read\_mem\_info를 코딩할 때, malloc을 이용한 동적할당 때문에 일어난 것으로 추측된다. 계속해서 노드를 만들 때, malloc에 의한 동적할당을 하였지만, 이후 컴파일 및 실행에서 segmentation fault (core dumped) 문제로 인해 free시키는 것을 빼버렸다. 그 때문에 해제되지 못한 쓰레기 값으로 돌아다니는 것들이 메모리 상에 계속해서 축적되어버려 지속적으로 증가하는 것으로 추측된다.

Ramspeed와 read\_mem\_info는 둘다 지속적으로 메모리 할당의 변화를 요구하고, 특징으로 보자면 항상 virtual memory가 physical memory가 증가하기 전에 먼저 증가함을 볼 수 있다. 사용자 입장에서 더욱더 효율적으로 메모리를 관리하기 위해 실제 물리메모리 보다 더 큰 메모리를 가지고 있다고 가상으로 착각하게 만들어 쓰는 것을 확인할 수 있었다.

* **물리 메모리의 변화량을 토대로 워킹 셋(working set)을 어떻게 정하면 좋을지**

주로 물리 메모리가 쓰이는 경향은 한번 증가하기 시작할 때, 다시 말해 무언가 작업을 수행하기 시작할 때, 지속해서 그 증가하는 경향이 유지되는 경우가 많다. Read\_mem\_info에서는 보지 못했지만, ramspeed의 경우에는 한번의 수행이 끝날 때마다 모든 관련 메모리를 해제시켜주는 것을 볼 수 있다. 따라서 일정 시간 동안 물리 메모리가 증가한다면, 미리 그 history를 감지하여 working set을 크게 늘려줄 필요가 있다. 그러면 이후에 발생할 thrashing을 방지할 가능성이 높을 것이다.

* **물리 메모리의 변화량이 메모리 영역 중 어느 영역에 위치할 지**

물리 메모리의 변화량은 매우 가변적이다. 따라서 stack에 위치하여 일정 조건이 만족하면 쌓였다가 다시 삭제되었다를 반복하며, 가장 효율적인 workingset을 만들어 줄 수 있도록 할 것이다.

**수행과정 중 발생한 문제점 및 해결방법**  
  
a) **segmentation fault (core dumped)**  
동적 할당인 malloc을 사용하다 보니 적절한 때에 free를 시켜주는 것이 힘들었다. 이론적으로 맞다고 생각하는데 실제로 동작을 하지 않아 어디서 문제인지도 찾기 힘들었다. 굳이 free일 때 뿐만 아니라 정확한 포인터의 지정 또한 필요했기 때문에 여기저기 많이 헤맸던 것 같다.  
결과적으로는 free는 주석처리 하여 사용하지 않았다. ( 결국 쓰레기 값으로 남음 )

b) **Linux에는 undo가 없다**

모든 코드를 만들고 숙제를 제출하기 위해 이제 tar로 압축을 하는데, 모르고 첫번째 인자로 압축할 파일의 이름이 아닌 read\_mem\_info.c 를 써버렸다. 그래서 압축파일이 덮어씌어져서 코딩한 파일이 전부 날아가 버렸는데 이를 되돌릴 undo가 윈도우와 달리 linux에는 존재하지 않았다. 그래서 숙제를 내지 못하고 다시 한번 하게되었다.