**소프트웨어학부 20180752 박은정 || ~ 2020.12.23. (Free day 5일 사용)**

‒ 개발환경



Makegraph.py의 경우 python3, jupyter notebook을 이용하여 작성하였습니다.

‒ 실행 결과 스크린샷 ‒ Application 실행 후 산출물 분석

**1. 기본 개념**

|  |  |
| --- | --- |
| **물리주소**  컴퓨터의 메인 메모리를 접근할 때 사용되는 주소 | **가상주소**  Process의 관점에서 사용하는 주소 |

**언제 물리 주소가 가상 주소로 바뀔 것인가**

Process가 실행 될 때 물리 주소가 바뀌는 경우, 물리 주소에 대한 Binding은 process가 실행될 때 일어난다. MMU의 도움을 받아 Paging이나 Swapping을 통해서 Process가 올려지는 Memory의 물리 주소는 바뀔 수 있다.

**[ 한정된 물리 메모리에서 용량이 큰 프로그램들을 동시에 실행시키고 싶었다. ]**

**가상메모리**

현재 실행되고 있는 Code 부분만이 실제 Memory에 있으면 Process는 실행이 가능하다. 그럴므로, 가상 메모리에는 Paging결과를 넣고, 물리메모리와 가상메모리를 Mapping한 Page Table를 이용하여, 물리메모리에는 실제 필요한 부분만 올린다. 실행 시키고자 하는 프로그램의 크기가 커도 물리 메모리에 필요한 page만 올리기 때문에 (demand paging) 실제 필요한 물리 메모리의 크기는 가상메모리를 사용하지 않았을 때보다 훨씬 적다. 즉, 가상 메모리 입장에서 메모리가 실제보다 확장되는 효과를 가진다.

**Segment (가상 메모리 관리 방식 - 가변 크기)**

프로세스를 서로 다른 크기의 논리적인 단위인 데이터, 힙, 스택 영역로 나눈 각각을 Segment라고한다. Paging과 달리 논리적 단위로 프로세스를 나누기 때문에 Segment는 각각 다른 크기를 가진다. Segment 단위로 메모리를 할당하는 방식을 Segmentation 기법이라고 한다.

Segmentation는 논리적인 단위로 나눴기 때문에 보호와 공유가 용이하다. 하지만, 각각 다른 크기를 가지기 때문에 External Fragmentation이 발생한다.

**External Fragmentation (가변 크기의 가상 메모리 관리 방식을 택했을 때의 문제점)**

External Fragmentation은 남은 Memory의 총 공간은 새로운 할당 요청에 충분하지만, 그 공간이 연속적이지 않아 Memory의 낭비가 있는 상태이다. External Fragmentation의 크기는 4KB보다 클 수도 있다. (Page의 크기를 4KB라고 했을 경우에 Page크기보다 클 수 있다.)

**Paging (가상 메모리 관리 방식 – 고정 크기)**

프로세스를 페이지라는 고정 크기 단위로 나누어 구분 관리하는 방식이다. 운영체제 커널은 각 프로세스에 대한 page table을 유지, 관리한다. 사용되지 않는 페이지의 복사본은 swap영역에 보관되고 필요할 때만 물리 메모리에 로드 된다. 만약, 현재 물리 메모리에 없는 페이지를 참조하면, page fault가 발생되고, 해당 페이지를 찾아서 메모리에 적재하는 동안 프로세스 실행이 잠시 중지된다.

- Frame: 물리 Memory를 고정된 크기로 나누었을 때, 하나의 Block

- Page: 가상 Memory를 고정된 크기로 나누었을 때, 하나의 Block

- page table : 물리 메모리에 위치함. Page와 Frame 간의 mapping 정보를 담고 있다.

**Internal Fragmentation (고정 크기의 가상 메모리 관리 방식을 택했을 때, 발생가능한 문제점)**

Internal Fragmentation은 Paging은 고정된 크기 단위로 나누어 관리하기 때문에, 만약에 Page 크기보다 작은 크기의 Process의 한 부분이 하나의 Page에 할당된다면, 나머지 부분이 잉여공간으로 남게 된다. 하지만, 잉여 공간이 최대 Page의 크기(=4KB)보다 작게 발생하여 External Fragmentation보다 적다. Page의 크기를 적절하게 조절하여 Internal Fragmentation의 발생량을 줄일 수있다.

**Demand Paging**

실행에 필요한 Page만을 메모리에 올리는 Paging 기법이다. 그래서, 실제로 필요한 물리 메모리양을 줄일 수 있다. Paging Service를 이용해서 한 Process에 필요한 Page를 Memory와 Secondary Storage 간에 이동시킨다. 참조하려는 Page가 invalid한 경우 Page Fault가 발생한다.

**Page Fault**

Page가 할당받은 Frame에 없는 경우, trap 발생 => backing storage에서 해당 page찾음 => 해당 Page를 physical memory에 가져옴. 만약, physical memory에 없다면, page replacement algorithm을 이용해서 victim page를 찾고 그 위치에 새로운 page로 대체한다.

**Thrashing**

Process의 실행 시간 중, Page fault를 처리하는 시간이 execution 시간보다 많아 지는 상황이다.

CPU schedular는 CPU Utilization이 떨어지는 것을 보고 CPU Utilization을 높이기 위해 새로운 프로세스를 추가하여 multiprogramming 정도를 높인다. 새로 시작하는 프로세스는 실행중인 프로세스들로부터 프레임을 가져오고자 하여 더 많은 page fault와 더 긴 페이징 장치 대기 시간을 발생시킨다. **CPU Utilization을 높이기 위해** 메모리에 **새로운 프로세스를 추가하지만**, **메모리가 부족해져 CPU Utilization이 낮아지는** 악순환이 반복되어 결국은 thrashing이 발생한다.

현재 고정된 페이지 프레임 수가 충분하지 않아서 page fault가 자주 발생하고, CPU 사용률이 떨어진다. 이를 예방하기 위해서 working set이 사용된다.

**Working Set**

WS(t,w)={pages P such that P was referenced in the time interval (t,t-w)}

현재 process가 실행되기위해 SWAP되지 않고 물리 메모리에 있어야하는 하는 Page들의 집합이다. Locality를 기반으로 Page Fault를 줄이기 위한 방법이다. 적당한 Working set의 크기를 결정하는 것이 중요하다.

working set size(WSS): 워킹 셋 안에 페이지 수 어떤 시간안에 참조된 page 수로 Locality에 따라 변화한다. 로컬리티가 떨어지는 경우에는 더 많은 페이지들이 reference되어야 한다.

운영체제는 프로세스의 Page Fault 발생 횟수를 확인하여 working set을 설정한다. Page Fault 발생 횟수에 따라 적절하게 Page Frame을 할당한다. (Resident Set 크기 설정.)

**Swap**

Page Out으로 Memory부족을 해결하지 못할 경우, swap out 대상인 Process 전체를 Secondary Storage로 보내는 데 이렇게 Page Out이나 Swapping에 사용되는 Secondary Storage를 Swap 영역이라고 한다. 가상 메모리 관리를 위해 사용되는 디스크 영역이다. 리눅스 커널에서는 LRU policy를 사용한다.

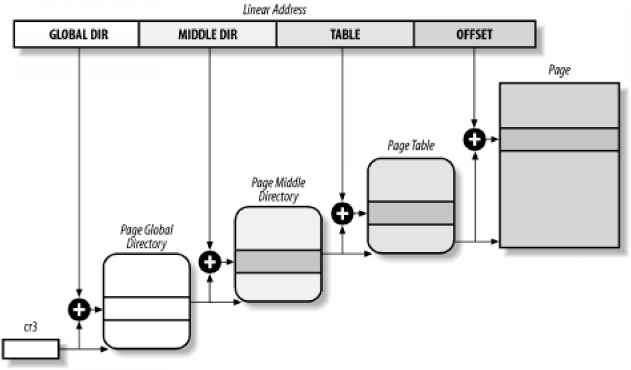
**LRU (Least Recently Used)**

가장 오랜 시간 사용되지 않은 page를 교체하는 방법이다.

**2. 리눅스의 Paging, Working Set, Thrashing, Swap Partition**

**Linux에서의 Paging**

리눅스 커널은 3단계 페이징을 사용한다. (multi-level paging)



PGD(Page Global Directory)의 page frame number을 이용하여 가상 주소 중 일부를 PMD(Page Middle Directory)의 entry로 변환하고, 이를 이용하여 인덱싱하여 PTE(Page Table Entry)를 얻게 된다. Page Table에서 인덱싱하면 실제 페이지 프레임 주소가 나온다.

**Working Set – 위와 동일, Thrashing – 위와 동일**

**Swap Partition**

메모리가 꽉 찼을 때, 추가적으로 실행되는 프로그램은 메모리가 아닌 swap에서 실행된다. Swap 영역에 옮겨 생긴 메모리의 여유공간을 이용해 더 빨리 수행할 수 있도록 한다. 최대 절전 기능을 사용할 수 있게 한다.

메모리와 swap 영역을 나누는 역할을 하는 것이 swap partition이다. Swap은 하드디스크에 저장된다. Swap의 크기는 적어도 설치된 메모리 크기와 같아야 하고, swap partition에 이미 옮겨져 있는 여러 항목을 위해 추가적으로 10%~25%의 여유 공간을 주어야 한다.

|  |
| --- |
| # swapon -s    # free -m =    Swapon , swapoff 했을 때 |

**3. 작성한 Code 설명**

**man page 참고한 내용**

|  |  |
| --- | --- |
| /proc/[pid]/statm | /proc/[pid]/status |
| Statm \* 4 = Status (Statm은 Page기준으로 측정됨. Page크기 4kB: getconf PAGESIZE 로 확인)  **Size** : 가상 메모리 (==VmSize )  **Resident** : **물리메모리** (==VmRSS )  **Shared** : 공유메모리 (= RssFile+RssShmem)  **Text**[2], **data+stack**[344293] | |

**작성한 파일 (총 5개)**

|  |  |
| --- | --- |
| getphymem.c | 물리메모리 |
| getsharedmem.c | 공유메모리 |
| getvirtmem.c | 가상메모리 |
| getdse.c | Text와 data |
| Makeobj.sh | 위 파일들 컴파일 용 |
| makegraph.py | 데이터 분석 용 (그래프 작성) |

**주요 코드 내용**

|  |
| --- |
| **getsharedmem.c**  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <time.h>  #include <unistd.h>  int is\_pidndur\_ok(int argc, char\*\*argv, int\*pid, int\*dur){  // pid값을 입력하지 않았을 때  if(argc<2){printf("%s pid duration: duration default=10",\*(argv)); return -1;}  else if(argc!=2){\*dur = atoi(\*(argv+2));}  \*pid = atoi(\*(argv+1));  // 문자를 입력했을 때  if(\*pid<=0||\*dur<=0)return -1;  return 0;  }  int get\_sharedinfo(char \*spid){  char buffer[1024];  **char dest[30];**  **sprintf(dest, "/proc/%s/statm", spid);**  int size, resident, shared ;  FILE \*fp=fopen(dest,"r");  if(fp==NULL){return -1;}  **//해당 위치의 파일을 한 줄씩 읽어들인다음에, 원하는 값만 화면에 출력한다.**  while(fgets(buffer,1024,fp)!=NULL){  **sscanf(buffer,"%d %d %d ",&size,&resident,&shared);**  **printf("%d\n",shared);//공유메모리**  memset(buffer,0,sizeof(buffer));  }  fclose(fp);  return 0;  }  int main(int argc, char\* argv[]){  int pid = -1;  int dur = 10; //default값 :10초  int pidexist=-1;  if(is\_pidndur\_ok(argc, argv, &pid, &dur)==-1){return -1;}  char spid[10];//입력받은 pid 값  sprintf(spid, "%d", pid);  for(int time=0; time<dur\*2; time++){  pidexist=get\_sharedinfo(spid);  if(pidexist == -1){printf("%s isn't exist.",spid);break;}//pid 존재 여부 판단  usleep(500000);//usleep(1000000 ) 1초  }  } |
| **getvirtmem.c**  int get\_virtinfo(char \*spid){  char buffer[1024];  **char dest[30];**  **sprintf(dest, "/proc/%s/statm", spid);**  int size, resident, shared ;  FILE \*fp=fopen(dest,"r");  if(fp==NULL){return -1;}  while(fgets(buffer,1024,fp)!=NULL){  **sscanf(buffer,"%d %d %d ",&size,&resident,&shared);**  **printf("%d\n",size);//가상메모리**  memset(buffer,0,sizeof(buffer));  }  fclose(fp);  return 0;  } |
| **getphymem.c**  int get\_phyinfo(char \*spid){  char buffer[1024];  char dest[30];  sprintf(dest, "/proc/%s/statm", spid);  int size, resident, shared ;  FILE \*fp=fopen(dest,"r");  if(fp==NULL){return -1;}  while(fgets(buffer,1024,fp)!=NULL){  **sscanf(buffer,"%d %d %d ",&size,&resident,&shared);**  **printf("%d\n",resident);//물리메모리**  memset(buffer,0,sizeof(buffer));  }  fclose(fp);  return 0;  } |
| **getdse.c**  **/\* [statm] statm의 data, text 확인-> text는 고정 data(stack+data(heap))는 변함.을 확인하기 위해서 \*/**  int get\_textNdata(char \*spid){/\*text=code, data=data+stack\*/  char buffer[1024];  char dest[30];  sprintf(dest, "/proc/%s/statm", spid);  int size, resident, shared,text,lib,data ;  FILE \*fp=fopen(dest,"r");  if(fp==NULL){return -1;}  while(fgets(buffer,1024,fp)!=NULL){  **sscanf(buffer,"%d %d %d %d %d %d",&size,&resident,&shared,&text,&lib,&data);//공백 기준으로 2번째 숫자이용.**  **printf("%d\t%d\n",text,data); //text와 data를 프린트한다.**  memset(buffer,0,sizeof(buffer));  }  fclose(fp);  return 0;  } |

\* 컴파일 편하게 하기 위해서 makeobj.sh파일을 만들었습니다.

**4. 가상/공유/물리 Memory에 대해 Graph를 그린다.**

Python을 이용하여 그래프를 그렸다. 전체적으로 physical과 virtual memory간에 비슷해보였다.

**4.1. Background 로 떠 있는 Application: Bash (2분 실행)**

|  |  |
| --- | --- |
| Bash\_physical | Bash\_shared |
| Bash\_virtual | Bash\_text |
| Bash\_data | \* 2분간 실행시켰지만, 변화가 없었다. |

**4.2. Memory 사용량이 큰 Application : FireFox (2분 실행)**

|  |  |
| --- | --- |
| Firefox\_physical | Firefox\_shared |
| Firefox \_virtual | 웹서핑을 진행하는 동안 매우 많은 메모리가 사용되었으며 변화도 역동적이었다. 중간 중간 physical memory 량이 급격하게 떨어지는 경우가 있는데, Page 교체가 일어난 것이라고 예상을 하였다. |
| Firefox \_data | Firefox \_text |

**4.3. 영상 (2분실행)**

영상을 계속 재생하였다.

|  |  |
| --- | --- |
| movie\_physical | movie \_shared |
| movie \_virtual | 너무 큰 용량의 영상을 돌리는 경우, 전체적으로 다른 작업들도 느려지는 경우가 있었다. 이를 확인해보기 위해서 영상을 재생하였다.  영상의 경우, 정해져있는 text량에 비해 data가 실행 중에 변화하는 비율이 매우 높았다. 그리고 Shared 메모리가 일정했다. 아마, 영상을 재생하는데 있어서 필요한 기본적인 내용은 shared메모리에 공유하고 있고, data영역은 영상 특징으로 인해 높낮이가 심한 그래프가 작성된 것 같다. |
| movie \_data | movie \_text |

**4.4. text파일 (2분 실행)**



|  |  |
| --- | --- |
| text\_phyiscal | text\_shared |
| text\_virtual | Text 파일을 vi 편집기로 열면, 항상 저렿게 swp 파일이 나와서 궁금했다.  파일이 일정 크기 이상이 되면, 갑자기 사용하는 메모리 량이 증가하는 것을 알 수 있었다. 특히, 이전에 할당된 메모리의 부족이 있다면, virtual memory가 먼저 증가하고, 그다음에 shared memory 그리고 physical memory가 증가함을 알 수 있었다. |
| text \_data | text \_text |

**5. 순수하게 해당 Process가 사용하는 Memory량에 대한 그래프**

**(2분 실행)**

가상 메모리(Size)에서 공유 메모리(Shared)를 제외해서 시간별 순수한 Process의 사용량에 대한 그래프를 작성하였다.

|  |  |
| --- | --- |
| **5.1. Bash** | **5.2. FireFox** |
| **5.3.영상** | **5.4 .** **text** |

virtual-shared을 각 프로세스별로 평균을 내었다.

|  |  |
| --- | --- |
|  | 순수 프로세스가 차지하는 메모리 량은  Bash>text>movie>Firefox순으로 높았다. 하지만, 이는 각 프로세스의 공유메모리 크기를 고려하지 않았기에 아래와 같은 그래프를 그렸다. |

각 프로세스에 대해 시간 별로 가상 메모리에서 공유 메모리가 차지하는 비율 그래프를 작성해보았다.

|  |  |
| --- | --- |
|  | FireFox>movie>bash>text 순으로 각 프로세스의 가상메모리에서 공유메모리가 차지하는 비율이 높았다.  왼쪽 그래프는 평균을 낸 그래프이다.  프로세스마다, 시간에 따라 공유메모리가 차지하는 비율이 다르기 때문에, 메모리 분석할때는 상황을 고려할 필요가 있음을 알 수 있었다. |

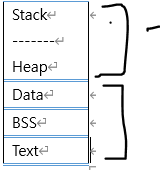
**6. 결과 분석**

**- 물리 Memory 변화량을 토대로 Working set을 어떻게 정하면 좋을지 분석할 것**

Working set이 특정 시간안에 참조된 페이지의 집합이므로, working set의 최대 크기는 물리 메모리 크기와 같다. **그러므로 working set의 크기는 최대 물리 메모리 크기보다 작거나 같고, 최소 물리 메모리 크기보다 크거나 같게 설정해야 한다고 생각한다.** 만약, working set이 너무 작다면, locality가 있는 page들도 working set에 없어서Page Fault 횟수가 많이 발생하고, working set이 너무 크다면, locality관련성이 없는 page들도 working set에 있기 때문에, 메모리 낭비가 일어나 성능에 좋지 않기 때문이다. 물리 메모리의 크기를 기준으로 삼은 이유는, 실제 참조된 page의 개수를 의미하기 때문이다. 만약, 물리 메모리 크기가 갑자기 줄어드는 경우, Page가 교체되는 경우라고 예상할 수 있기 때문에, Page Fault를 줄이는 방향으로 Working set을 설정해야 하기 때문이다.

**- 물리 Memory 변화량이 Stack,Data,Code 중 Stack에 위치할 것이다.**

이유 :

Stack (지역변수,매개변수,pc… 임시 메모리)

Heap (프로그래머가 동적으로 사용하는 영역)

Data (런타임 이전에 초기화 된, 전역변수, 정적변수, 배열, 구조체)

BSS (런타임 이후 초기화)

Text (code, read only, 프로세스 종료시까지 변하지 않는다.)

위에는 실행 중 크기가 결정되고, 아래는 컴파일 중 크기가 결정된다. 즉, 실행 중에는 Stack이나 Heap 부분에서 물리 메모리의 크기에 영향을 줄 것이다.

또한, 위 결과에서도 볼 수 있듯이, Text은 크기가 변경되지 않는다.

**- 가상 Memory량과 공유 Memory량을 토대로 순수하게 해당 Process가 사용하는 Memory 량은 얼마인지 분석하고 Graph를 그릴 것.**

가상 Memory량(Size)에서 공유 Memory량(Shared)을 제외해서 순수한 Process의 사용량에 대한 그래프를 작성하였다. 5에서 확인 가능하다.