OS Term Project #1

손봉진 20154445
이남섭 20154485
유충모 20154473

[환경 설정]

- OS : ubuntu 64bit
- cpu frequency값 고정시킴
- Kernel version : 3.18.21(longterm)

[user level program]

> run.sh를 통해 수행되는 프로그램은 각 시스템콜을 불러오는 프로그램을 인자값을 주어 실행시켜 해당 수행 시간을 받아오는 프로그램으로써, 프린트 되는 값은 아래의 결과를 나타낸다.

```
short-input write mean=18930
short-input write var=8251142
long-input write mean=41524
lont-input write var=13165340
open mean=20851
open var=1756700
open with write mean=23898
open with write var=1877997
read mean=5395
read var=241529
read with write mean=7234
read with write var=15716867
bigread mean=13429
bigread var=31715
mkdir mean=12437
mkdir var=258652354
```

-쓰기 값이 작을 때 write 시스템 콜 수행 평균 시간, 분산
-쓰기 값이 클 때 write 시스템 콜 수행 평균 시간, 분산
-기본 열기 시 시스템 콜 수행 평균 시간, 분산
-쓰기 수행중인 파일 열기 시 시스템 콜 수행 평균 시간, 분산
-기본 read 시스템 콜 수행 평균 시간, 분산
-쓰기 수행중인 파일 read 시 시스템 콜 수행 평균 시간, 분산
-read 버퍼 값이 클 때 시스템 콜 수행 평균 시간, 분산
-mkdir 명령어 수행시 평균 시간, 분산

위 값 순으로 표시된다.

[커널 값 선정 과정]

각 시스템 콜 파일에서 불러오는 함수들에 ctag를 이용하여 접근하여 on/off 가능한 시간 값 측정 함수를 넣어 전체 수행 시간을 측정하였다. 전체 함수들을 위해 변경된 파일들은 함께 압축하여 제출하였다. 해당 값들은 /kernel 파일에 기존 /usr/src/linux/ 아래에 있던 경로 대로 옮겨 기존 위치 파악이 쉽도록 하였다.

1. write system call

- 정의

> 파일에 데이터를 쓰는 함수. write() 시스템 콜은 자신을 호출한 프로세스의 사용자 모드 주소 공간에 있는 데이터를 커널 자료 구조로 옮긴 다음, 다시 디스크로 옮긴다. 파일 객체의 write 메소드는 파일 시스템들이 자신만의 write 연산을 generic_file_write() 함수에 정의하도록 한다.

- 함수 원형

size_t write (int fd, const void *buf, size_t n)

> write는 read 시스템 콜과 유사하게 인수로 파일 디스크립터 fd, 메모리 영역 (전송할 자료를 포함한 버퍼) 주소 buf, 그리고 얼마나 많은 버퍼를 전송해야 하는지를 나타내는 숫자 count 를 갖는다. 최종적으로 그 정보를 파일의 *offset 위치에 쓰고 *offset 값(일반적으로 파일 포인터에 대응)이 증가시킨다.

- 시간 측정

1)기본 수행시

> 처음엔 fget_light()을 호출하여 fd로부터 대응하는 파일 객체의 주소 file을 얻고 access_ok() 함수를 호출하여 buf 및 count 매개 변수에 대해 간단한 검사를 수행한다. 그 후 rw_verify_area() 함수를 호출하여

접근할 파일 일부분을 위한 충돌하는 강제적인 락이 있는지를 그 후에 write 함수를 호출하고, 실제로 전송된 바이트의 수를 반환하고 동시에 파일 포인터를 갱신한다. 이후에 fput_lighter() 함수를 호출하여 파일 객체를 해지한다. 마지막으로 실제로 전송된 바이트 수를 반환한다.

[3240.198885] rw_verify_area start
[3240.198886] rw_verify_area time: 0 sec, 115 ns
[3240.198890] write_iter time: 0 sec, 2917 ns
[3240.198891] fsnotify, add_wchar time: 0 sec, 138 ns
[3240.198892] inc_syscw, file_end_write time: 0 sec, 75 ns

2)쓰기 내용에 따른 변화 추이
> 쓰기 내용을 string 크기가 xx 두 자리 수의 길이일때와 xxxxx 다섯 자리 수의 길이일때를 각각 유저단에서 측정하였을시에 약 2배에 가까운 큰 차이가 발생함을 알 수 있었다.

*** INPUT = 약 10 개의 문자로 이뤄진 string**
write() 평균 = 22987 (콘솔 실 표기 : short-input write mean=22987)
write() 표준편차 = 4607.43 (콘솔 실 표기 : short-input write var=21228473)

*** INPUT = 약 31100 개의 문자로 이뤄진 string**
write() 평균 = 42863 (콘솔 실 표기 : long-input write mean=42863)
write() 표준편차 = 4335.09 (콘솔 실 표기 : long-input write var=18792967)

커널로 들어가 write에 관여하는 것 같은 시스템 콜 들에 각각 printk를 넣어 커널 컴파일 후 dmesg를 통해 두 개의 시간에서 어떠한 부분이 큰 영향을 받는지에 대해 알아보았다. 그 결과 다른 함수들에서는 큰 시간적 차이가 없었지만 write_iter time 함수에서는 정말 큰 차이가 발생하는것을 알 수 있었다. 다음 과제에 이 부분에 대하여 더 정확히 어떤 세부적인 콜이 있는지에 대해 콜 스택에 대해 알아보려한다.

*** INPUT = 약 10 개의 문자로 이뤄진 string**	*** INPUT = 약 31100 개의 문자로 이뤄진 string**
[3892.019203] rw_verify_area start	[3892.027262] rw_verify_area start
[3892.019207] rw_verify_area time: 0 sec, 187 ns	[3892.027266] rw_verify_area time: 0 sec, 147 ns
[3892.019217] write_iter time: 0 sec, **7302 ns**	[3892.027299] write_iter time: 0 sec, **31792 ns**
[3892.019219] fsnotify, add_wchar time: 0 sec, 328 ns	[3892.027301] fsnotify, add_wchar time: 0 sec, 283 ns
[3892.019220] inc_syscw, file_end_write time: 0 sec, 163 n	[3892.027303] inc_syscw, file_end_write time: 0 sec, 163 n

2.Open system call

- 선정 이유
파일 관련해서 가장 많이 수행되는 시스템 콜이며, 쓰기 수행 시 읽기가 제한된다는 점에서 해당 변화를 측정해보고자 선정하였다.

- 시간 측정
기본적인 전체 시스템콜 성능 측정은 다음과 같이 진행하였다.
1)기본 수행시
기본 수행시에는 20번 수행하여 평균 5395ns가 소요되었으며, 분산은 241529, 표준 편차는 491이다.
2)해당 파일에 대하여 쓰기 수행중 읽기 수행 시간
열기를 수행하는 파일에 대하여 쓰기가 진행중일 때는 평균 23898ns가 소요되었으며, 분산은 1877997, 표준 편차는 1370이다.

3)기본 수행 시 상세 함수 수행시간
open 시스템 콜을 SYSCALL_DEFINE3을 통해 전체 시스템 콜을 시작한다. 처음으로는 getname() 함수를 호출하여 프로세스 주소 공간에서 파일 경로명을 읽어온다. 이후 get_unused_fd() 함수를 호출하여 파일 디스크립터에서 빈 공간을 확인하고, 해당 인덱스를 저장하였다. build_open_flags를

실행시켜서 파일 플래그 수행을 설정하였으며, 해당 시간에는 255ns가 소요되었다. 이후 do_sys_open을 통해 전체 open 수행이 시작하였으며, 이는 vfs_open 함수를 통해 실제 파일 디스트립터를 얻어오는 작업을 시작하고, 이때에 do_dentry_open을 통해 inode를 얻어오는 작업을 2001 ns 동안 수행하였다. 이후 dentry_open을 수행하고 여기서 open_check_o_direct 함수를 통해 115ns 동안 직접 입출력 작업이 이 파일에 대해 수행될 수 있는지 검사하였다. 이후 전체를 파일 객체의 주소를 반환하고 전체 수행은 22826ns가 수행되었다.

4)쓰기 진행중인 파일을 읽기 수행 시 상세 수행 시간

쓰기 진행중인 파일을 접근할 경우에는, SYSCALL_DEFINE3은 140ns, build_open_flags는 235 ns로 이전과 비슷하게 수행되고 do_sys_open으로 실제 open 시스템 콜이 시작하였다. 여기서 vfs_open을 235ns동안 수행하고 바로 멈추게 되는데, 이는 파일이 쓰기 수행중이라 해당 부분에서 읽기 작업을 잠시 멈추게 된다. 이후 do_dentry_open을 통해 3077ns 동안 수행하였는데 이는 쓰기 작업 이후 inode를 가져오는 과정에서 더 사용된 시간인 것으로 생각된다. 이후 마찬가지로 open_check_o_direct 함수가 146ns 동안 비슷하게 수행되며 전체 수행시간은 28567ns로 늘어난 결과로 종료되었다.

open의 경우 해당 결과 값에서 직접적인 시스템 콜 내부 요소를 살펴보기 위하여 fs/open.c파일에 로그를 찍어 확인해 보았다. 기본적인 흐름은 vfs_open() 함수와 do_sys_open()함수를 통해 불려진 open 시스템콜은 특정 부분에서 시간이 더 소요되어 증가하는 것을 확인할 수 있었다.

3.Read system call

- 선정이유

읽기 이후에 기본적으로 수행되는 것이 읽기 작업이므로 해당 작업에서 수행되는 시간을 측정하고자 하였다.

- 시간 측정

기본적인 전체 시스템콜 성능 측정은 다음과 같이 진행하였다.

1)기본 수행시

기본 수행시에는 20번 수행하여 평균 20851ns가 소요되었으며, 분산은 1756700, 표준 편차는 1325이다.

2)해당 파일에 대하여 쓰기 수행중 읽기 수행 시간

읽기를 수행하는 파일에 대하여 쓰기가 진행중일 때는 평균 7234ns가 소요되었으며, 분산은 15716867, 표준 편차는 3964이다.

1)기본 수행시 상세 수행시간

기본 수행시 fget_light를 호출하여 fd로부터 대응하는 파일을 얻는다. 이후 file->f_mode의 플래그가 요청한 접근을 허락하지 않으면 에러코드를 반환한다. 이 후 access_ok()함수를 통해 접근할 buf 및 count 매개 변수에 대해 간단한 검사를 수해한다. 이후 rw_verify_area함수를 호출하여 접근할 파일 일부분을 위한 충돌하는 강제적 락이 있는지를 검사한다. 이후 file->f_op->read가 정의되면 호출하여 자료를 전송하고, 실제로 전송된 바이트 수를 반환한다. 마지막으로 fput_light함수를 호출하여 파일 객체를 해지한다. 여기서 vfs_read함수가 230ns동안 수행되며, 전체는 5495ns동안 수행되었다.

2)쓰기 진행중인 파일을 읽기 수행 시 상세 수행시간

쓰기 진행 중인 파일을 읽기 수행할 때는 함수 수행이 늘어난다. 먼저 rw_verify_area를 통해 해당 파일이 읽기 가능한 상태인지 확인하는데, 해당 과정이 235ns가 소요되며, 이후 vfs_read 함수가 직접적으로 읽기를 시작한다. 여기서 write_iter가 1637ns 동안, fsnotify, add_wchar가 350ns 동안 수행되며, inc_syscw, file_end_write 함수가 235ns 동안 수행하고, 다시 rw_verify_area 함수를 호출한다. 여기서 135ns동안 재 검사를 진행하며, write_iter를 통해 1551ns동안 대기한다. 이후 fsnotify, add_wchar를 실행하여 290ns진행하고, inc_syscw, file_end_write함수를 통해 210ns동안 파일이 끝난지 확인한다. 이후 다시 rw_verify_area를 85ns 동안 수행하여 락을 재검사하며, 가능할 경우 write_iter를 1606ns동안 수행하며 값을 확인하여 vfs_read를 마치는 것으로 전체 수행이 이루어지며, 전체 소요시간은 7234ns이었다.

3)버퍼의 크기에 따른 상세 수행시간

버퍼의 크기는 xx일때, xxxxxx일때에 따라 수행하였으며, 각각의 수행 시간은 5271ns와 7694ns로 버퍼 크기에 따라 큰 폭으로 변화함을 확인할 수 있었다.

***버퍼 크기가 10일때**	***버퍼 크기가 100000일때**
[2799.010686] rw_verify_area start [2799.010687] rw_verify_area time: 0 sec, 284 ns [2799.010688] new_sync_read start [2799.010690] new_sync_read time: 0 sec, **1559 ns** [2799.010691] vfs_read time: 0 sec, 5271 ns	[613.094501] vfs_read start [613.094505] rw_verify_area start [613.094506] rw_verify_area time: 0 sec, 330 ns [613.094507] new_sync_read start [613.094512] new_sync_read time: 0 sec, **3639 ns** [613.094513] vfs_read time: 0 sec, 7694 ns

read의 경우 해당 시스템 콜의 내부 함수 값 측정을 위해 fs/read_write.c와 fs/locks.c의 함수들에 측정 위치를 넣어 측정하였다. 기본적으로 파일이 쓰기 중일때 함수에서 시간이 가장 오래 걸렸으며, 해당 위치에서 조건에 따라 변화함을 확인할 수 있는 것으로 볼 때, 쓰기 시 해당 함수가 wait 상태에서 대기함을 확인 할 수 있었다. 또한, 버퍼 값이 증가함에 따라 가장 오래 걸린 함수는 new_sync_read() 이었다.

4. mkdir system call

- 정의
MaKe DIRectory의 약어로서 새로운 디렉토리를 만들때 사용하는 시스템 콜이다.

- 함수 원형
int mkdir(const char *pathname, mode_t mode);
> pathname : 생성할 디렉토리 경로와 이름
> mode : mode는 사용할 수 있는 권한에 대한 허가권을 지정하며, 이것은 일반적으로 umask에 의해 수정된다.

- 리턴값
성공 시 : 0 보다 큰 값, 실패 시 : -1

- 시간 측정

1) 기본 수행시(현재 존재하지 않는 디렉토리 이름을 넣었을 경우)
평균 43719ns가 소요되었으며, 분산은 49808215ns, 표준 편차는 7057ns이다.

user_path_create(7219ns) - {kern_path_create(5179ns)}
security_path_mkdir(243ns)
vfs_mkdir(2645ns) - {may_create(1501ns), security_inode_mkdir(58ns), fsnotify_mkdir(348ns)}
done_path_create(5829ns) - {dput(403ns), mutex_unlock(36ns), mnt_drop_write(74ns)}

2) 현재 존재하고 있는 디렉토리 이름을 넣어서 수행 시
평균 13076ns가 소요되었으며, 분산은 132873298ns 표준 편차는 11527ns이다.

user_path_create(2745ns) - {kern_path_create(901ns)}
security_path_mkdir(140ns)
vfs_mkdir(2029ns) - {may_create(1451ns), security_inode_mkdir(55ns), fsnotify_mkdir(85ns)}
done_path_create(5077ns) - {dput(50ns), mutex_unlock(36ns), mnt_drop_write(93ns)}

3) 아무런 인자값 없이 수행 시
평균 13492ns가 소요되었으며, 분산은 171984151ns, 표준 편차는 13114ns이다.

user_path_create(4374ns) - {kern_path_create(2460ns)}
security_path_mkdir(142ns)
vfs_mkdir(1981ns) - {may_create(1378ns), security_inode_mkdir(54ns), fsnotify_mkdir(95ns)}
done_path_create(5314ns) - {dput(158ns), mutex_unlock(36ns), mnt_drop_write(94ns)}