

과제 #1

- 커널 레벨 클립보드 서비스

제출 기한: 5/21 (화) 23:59

제출 방법: KEDILMS “과제 #1”

질의응답 프로토콜

1. KEDILMS Q/A 게시판 확인
2. 인터넷 (구글) 검색
 1. 문제가 났을 때의 문장 혹은 키워드로 구글 검색
 2. 한글 자료는 많이 없으므로, 영어 사용 권장
3. KEDILMS Q/A 질문답변 게시판에 질문 올리기
 1. 최대한 빠르게 답변하려고 노력 중이며,
 2. 질문 답변은 분반 간에 공유됨

과제 학습 목표 및 내용

- 학습 목표

- 시스템 콜의 역할, 동작 원리를 이해한다.
- 커널 내에서의 프로그래밍을 수행해본다.
- (+) 동기화 문제를 확인하고 이해한다.

- 내용

- 커널 레벨 클립보드 서비스 구현: K-board
 - 커널 내에 ring buffer 기반으로 클립을 관리하는 K-board 구현
 - K-board에 접근하기 위한 시스템 콜 구현: enqueue, dequeue
 - K-board를 사용하는 예제 프로그램 구현: copy, paste

- 과제 총점수 30점 중 15점

- 기본 수행 12점
- (+) 동기화 추가 실험 3점

과제 제출 내용

- 보고서와 소스코드를 압축해서 제출
- 보고서 (학번.pdf)
 - 제목, 학번, 이름, 캡처 화면 1개, 어려웠던 점 및 해결 방안, 소스코드
 - 유저 프로그램 결과 화면 캡처 1개 (분리 가능)
 - Copy, paste 를 연속해 실행한 결과 화면 캡처
 - Ring buffer 동작 확인을 위해, 여러 차례 동작시킨 결과를 캡처
 - Full, empty 상태 확인을 포함하는, copy and paste 한 결과
 - 소스 코드는 수정한 부분, 핵심적인 부분만 포함할 것. 캡처 무방.
 - 캡처 화면, 소스 코드 외의 내용은 A4 3장을 절대 넘기지 말 것
 - (+) 동기화 관련 내용을 추가 수행한 경우, 해당 내용은 추가 3장 내에 작성
- 소스코드 (학번.zip)
 - 커널: 4개 파일 (수정 및 추가한 파일)
 - 유저: 3개 파일 (라이브러리, 소스 2개)

순서

0. J-Cloud: snapshot을 이용한 instance 생성

1. K-board skeleton 구현
2. 시스템콜 호출을 위한 유저 프로그램 작성
3. K-board 서비스 구현
4. 유저 프로그램 확장: library, copy, paste
5. 테스트 및 버그 수정
6. (+) 동기화 문제 확인 및 해결



J-Cloud: snapshot을 이용한 instance 생성

- * 5/2 이후, 공지사항을 확인하고 과제를 시작할 것
- * 5/1(수) 전체 점검 예정



스냅샷?

- Snapshot
 - 내 가상머신의 저장장치를 그대로 복사해두는 것 = 가장 단순한 “백업”
- 왜 스냅샷을 이용해야 하나?
 - OS, DB, Webserver 등, 설치 및 설정을 마무리한 내용이 모두 그대로 복원됨
 - 어플리케이션의 수행 내용도 똑같이 적용됨
 - 복원하기가 단순함
 - 일일이 파일 복사할 필요도 없고, 기존 파일이 있는 경우 대조를 하지 않아도 됨
- 모든 클라우드 시스템이 스냅샷 기능은 제공함
 - 다만 저장 공간에 따라 요금이 부과될 수 있음
- 현재 과제 #0을 완료한 인스턴스에 대해 모두 스냅샷을 작성해 둬

스냅샷 기반 인스턴스 생성

- 핵심

- 인스턴스를 생성하던 기존 방법과 거의 동일함
- 다만, image source를 image가 아닌 instance snapshot으로 설정
- 그리고 본인의 snapshot을 지정

- Note

- 인스턴스 생성 시마다 ip는 달라질 수 있음
- os2019.flavor 의 머신 사양이 달라졌음
 - vCPU 12개, 메모리 3GB, SSD는 그대로 40GB
 - 너무 많은 인스턴스의 압박으로 인해...
 - make 할 때 48개 쓰레드를 만들면 메모리 부족 문제 발생
 - -j 24 로 수행할 것

스냅샷 기반 인스턴스 생성

Launch Instance

Details

Source

Flavor *

Networks *

Network Ports

Security Groups

Key Pair

Configuration

Server Groups

Scheduler Hints

Metadata

Instance source is the template used to create an instance. You can use an image, a snapshot of an instance (image snapshot), a volume or a volume snapshot (if enabled). You can also choose to use persistent storage by creating a new volume.

Select Boot Source

Instance Snapshot

Create New Volume

Yes

No

Allocated

Name	Updated	Size	Type	Visibility	
> hcpark.test	3/26/19 1:19 AM	35.32 GB	qcow2	Private	↓

▼ Available 42

Select one

Q Click here for filters.

Name	Updated	Size	Type	Visibility	
> csj-test	3/28/19 8:20 PM	2.67 GB	qcow2	Private	↑
> csj-test	3/28/19 8:26 PM	2.67 GB	qcow2	Private	↑
> hcpark.oslab.snapshot	3/22/19 12:43 PM	7.47 GB	qcow2	Private	↑
> jy-test-0322	3/22/19 6:28 PM	1.68 GB	qcow2	Private	↑



SSH 접속하여 확인 후, 진행

- 기존과 동일하게 SSH 접속
- 본인이 과제 #0에서 완료한 커널 버전으로 잘 부팅되었는지 확인
 - 문제가 있을 경우, 이메일로 연락할 것
 - hyunchan.park@jbnu.ac.kr

순서

0. J-Cloud snapshot을 이용한 instance 생성
1. K-board skeleton 구현
2. 시스템콜 호출을 위한 유저 프로그램 작성
3. K-board 서비스 구현
4. 유저 프로그램 확장: library, copy, paste
5. 테스트 및 버그 수정
6. (+) 동기화 문제 확인 및 해결

K-board skeleton 구현

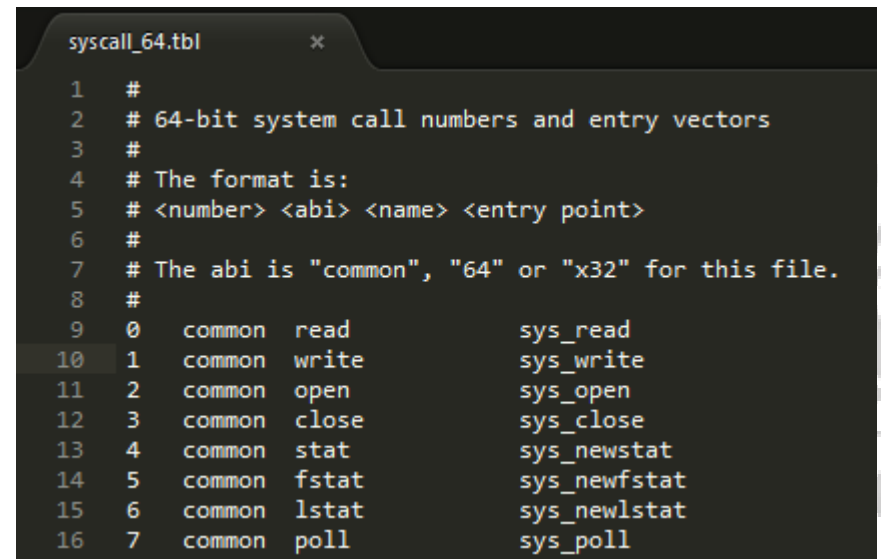


커널 작업 내용

- 작업 디렉토리 (과제 0에서 사용한 소스 디렉토리)
 - /usr/src/linux-source-4.18.0/linux-source-4.18.0/
- 수정할 파일
 - 시스템콜 번호 등록
 - arch/x86/entry/syscalls/syscall_64.tbl
 - ~~시스템콜 처리 함수 원형 등록 (수행하지 않아도 됨)~~
 - ~~include/linux/syscalls.h~~
 - 시스템콜 처리 함수 구현
 - kernel/os_kboard.c (새로 만들 파일. 주로 여기서 작업이 이루어짐)
 - 시스템콜 처리 함수 컴파일 설정
 - kernel/Makefile

시스템콜 번호 등록

- syscall_64.tbl 파일 확인
- 기존 내용과 포맷 맞춰서 새로운 시스템콜 정의
 - 334 이후에 아래 내용 입력해서 등록
 - 335 common kb_enqueue __x64_sys_kb_enqueue
 - 336 common kb_dequeue __x64_sys_kb_dequeue

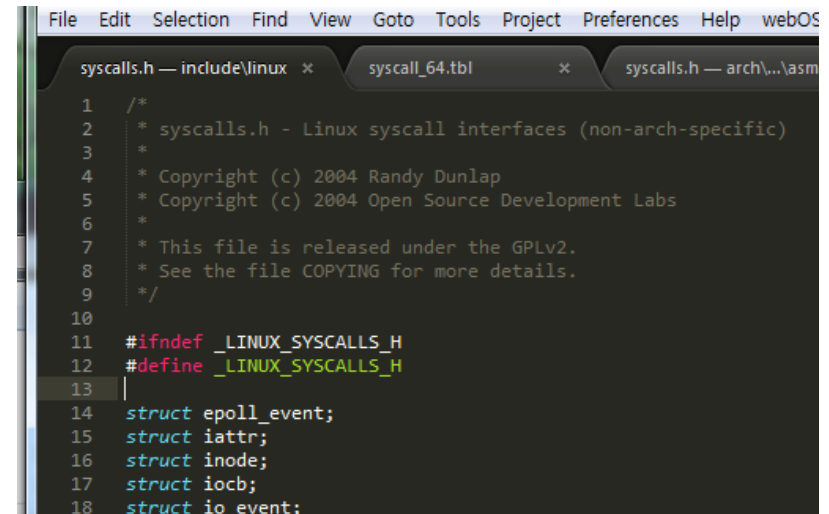


The image shows a terminal window with a file named `syscall_64.tbl` open. The file contains a list of system call definitions. The first few lines are comments explaining the format: `#`, `# 64-bit system call numbers and entry vectors`, `#`, `# The format is:`, `# <number> <abi> <name> <entry point>`, `#`, and `# The abi is "common", "64" or "x32" for this file.` The following lines are the actual definitions, numbered 0 to 7. Line 10 is highlighted. The definitions are:

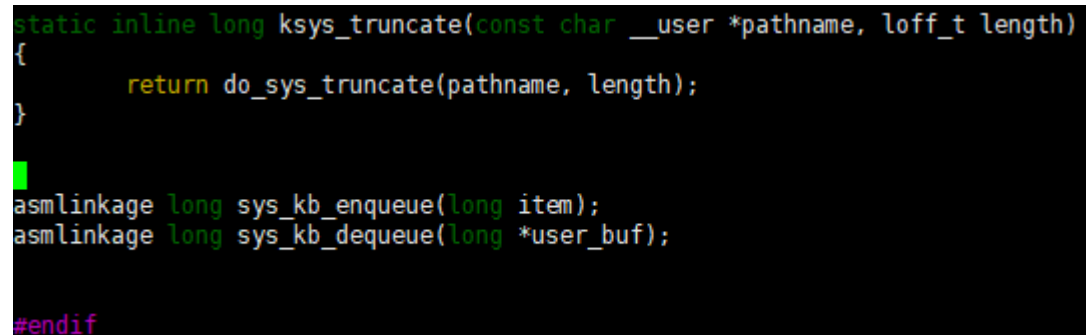
Number	ABI	Name	Entry Point
0	common	read	sys_read
1	common	write	sys_write
2	common	open	sys_open
3	common	close	sys_close
4	common	stat	sys_newstat
5	common	fstat	sys_newfstat
6	common	lstat	sys_newlstat
7	common	poll	sys_poll

시스템콜-처리-함수-원형-등록

- syscalls.h 에 Proto type 정의
 - 파일 맨 아래, #endif 앞
 - 앞 슬라이드에서 정의한 함수 이름과 동일하게
- Parameters
 - enqueue: int item
 - dequeue: int *user_buf
 - Return 값은 둘다 long 형태로 정의



```
File Edit Selection Find View Goto Tools Project Preferences Help webOS
syscalls.h — include/linux x syscall_64.tbl x syscalls.h — arch/.../asm
1  /*
2  * syscalls.h - Linux syscall interfaces (non-arch-specific)
3  *
4  * Copyright (c) 2004 Randy Dunlap
5  * Copyright (c) 2004 Open Source Development Labs
6  *
7  * This file is released under the GPLv2.
8  * See the file COPYING for more details.
9  */
10
11 #ifndef _LINUX_SYSCALLS_H
12 #define _LINUX_SYSCALLS_H
13
14 struct epoll_event;
15 struct iattr;
16 struct inode;
17 struct iocb;
18 struct io event;
```



```
static inline long ksys_truncate(const char __user *pathname, loff_t length)
{
    return do_sys_truncate(pathname, length);
}

asmlinkage long sys_kb_enqueue(long item);
asmlinkage long sys_kb_dequeue(long *user_buf);

#endif
```

시스템콜 처리 함수 구현: os_kboard.c

- printk() 함수
 - 커널 내에서 printf() 와 비슷한 역할을 하는 함수
 - Debug level 지정 가능: 과제에서는 KERN_DEBUG 사용
 - 결과물은 커널 메시지에 저장 (유저는 dmesg 로 확인)

- os_kboard.c 파일에 skeleton code 작성

- 인터페이스만 맞춰 호출을 테스트하는 용도의 뼈대 코드
- 차후 살을 붙여서 기능을 수행하도록 함
- 새로운 파일을 생성하여야 함

```
#include <linux/syscalls.h>
#include <linux/printk.h>
#include <linux/uaccess.h>

long do_sys_kb_enqueue(int item) {
    printk(KERN_DEBUG "HCPARK: do_sys_kb_enqueue() CALLED!! item=%d\n", item);

    return 0;
}

long do_sys_kb_dequeue(int *user_buf) {
    printk(KERN_DEBUG "HCPARK: do_sys_kb_dequeue() CALLED!! addr=0x%p\n", user_buf);

    return 0;
}

SYSCALL_DEFINE1(kb_enqueue, int, item) {
    return do_sys_kb_enqueue(item);
}

SYSCALL_DEFINE1(kb_dequeue, int __user *, user_buf) {
    return do_sys_kb_dequeue(user_buf);
}
```



시스템콜 처리 함수 구현: os_kboard.c

- 시스템콜 정의

- os_kboard.c 맨 아래에 다음 코드 삽입

```
SYSCALL_DEFINE1(kb_enqueue, int, item) {  
    return do_sys_kb_enqueue(item);  
}  
  
SYSCALL_DEFINE1(kb_dequeue, int __user *, user_buf) {  
    return do_sys_kb_dequeue(user_buf);  
}
```

- SYSCALL_DEFINEx 는 syscalls.h 에 정의된 매크로
 - 처리할 인자 개수에 따라 시스템콜 핸들러 함수를 정의해 줌
 - do_sys_* 로 전달하기만 하는 wrapping function
 - __user 는 해당 인자가 유저 공간의 주소를 가리키고 있음을 명시



시스템콜 처리 함수 컴파일 설정

- kernel/Makefile 수정
- 새로 추가한 파일이 컴파일 과정에 포함되도록 수정
 - 커널 컴파일 과정이 Make 시스템을 이용하기 때문

```
obj-y = fork.o exec_domain.o panic.o \  
      cpu.o exit.o softirq.o resource.o \  
      sysctl.o sysctl_binary.o capability.o ptrace.o user.o \  
      signal.o sys.o umh.o workqueue.o pid.o task_work.o \  
      extable.o params.o \  
      kthread.o sys_ni.o nsproxy.o \  
      notifier.o ksysfs.o cred.o reboot.o \  
      async.o range.o smpboot.o ucount.o os_kboard.o
```



커널 컴파일 및 설치

- `sudo make -j 24`
- `sudo make -j 24 modules_install install`
- (성공 후 재부팅)
- 참고
 - 처음 .h .tbl 파일들을 수정했을 때는 첫 컴파일과 유사하게 시간이 오래 걸림
 - 영향을 받는 파일이 많기 때문
 - `modules_install` 수행해야 함
 - 이후 `os_keyboard.c` 파일만 수정하면 시간이 짧음 (2-3분)
 - 영향을 받는 다른 파일이 없음
 - `modules_install` 제외하고 `install` 만 수행

시스템콜 호출을 위한 유저 프로그램 작성



syscall()을 이용한 시스템콜 호출

- 유저 프로그램 작성 (C 언어)
 - 시스템콜 호출 함수: syscall()
 - 시스템콜 호출을 위한 라이브러리 서비스
 - man syscall 을 통해 매뉴얼 확인 후 사용
 - 사용하기 어려운 경우, 구글에서 example code 검색
 - 335번, 336번 사용할 것
 - copy.c 작성 및 컴파일
 - gcc -o copy copy.c
 - 작업 경로는 자유

```
SYSCALL(2) Linux Programmer's Manual

NAME
    syscall - indirect system call

SYNOPSIS
    #define _GNU_SOURCE          /* See feature_test_macros(7) */
    #include <unistd.h>
    #include <sys/syscall.h>    /* For SYS_XXX definitions */

    long syscall(long number, ...);

DESCRIPTION
```

수행 결과: 시스템콜 만들기 전

- 결과

- 아무런 메시지도 나오지 않음
 - 현재 해당 번호로 구현된 시스템 콜이 없기 때문
- Perror() 로 확인
 - 시스템 에러 메시지 확인
- 2번(open)를 호출하면?
 - syscall(2);
 - syscall(2, 1);
 - syscall(2, 1, 2);

```
error - print a system error message

SYNOPSIS
#include <stdio.h>

void perror(const char *_s);

#include <errno.h>

const char * const sys_errlist[];
int sys_nerr;
int errno; /* Not really declared this way; see errno(3) */

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

sys_errlist, sys_nerr: BSD SOURCE
```

```
hccpark@hccpark-VirtualBox:~/user$ gcc -o copy copy.c && ./copy
syscall: Function not implemented
syscall: Function not implemented
```

```
i = syscall(2, 1, 2);

perror("syscall");
```

```
hccpark@hccpark-VirtualBox:~/user$ gcc -o copy copy.c
hccpark@hccpark-VirtualBox:~/user$ ./copy
syscall: Bad address
hccpark@hccpark-VirtualBox:~/user$
```

수행 결과: 시스템콜 만들고, 재부팅 후

- dmesg
 - 커널 메시지 확인
 - /var/log/kern.log 에서도 확인 가능
(tail -f 명령을 이용하면 업데이트 되는 대로 바로 확인됨)
- dmesg -c
 - 커널 메시지 확인 및 현재까지의 메시지 삭제
 - 관리 권한 필요: sudo 사용
- 결과
 - 유저: perror()는 success
 - 커널: dmesg 에서 호출됨을 확인
- K-Board 서비스를 구현할 준비완료

```
hcpark@hcpark-VirtualBox:~/user$ gcc -o copy copy.c && ./copy
syscall: Success
syscall: Success
hcpark@hcpark-VirtualBox:~/user$ dmesg
```

수행 결과

- 아래 결과는 테스트를 위해 내부 버퍼를 dump 한 결과를 포함하고 있음
- 현 단계에서는 호출이 정상적으로 이루어지는지, 인자가 정확히 전달되는지만 확인

```
root@os41983:/usr/src/linux-source-4.18.0/linux-source-4.18.0# tail -f /var/log/kern.log
Apr 30 17:39:38 os41983 kernel: [ 10.106170] audit: type=1400 audit(1556613575.652:5): apparmor="STATUS" operation="profile_load"
profile="unconfined" name="man_groff" pid=731 comm="apparmor_parser"
Apr 30 17:39:38 os41983 kernel: [ 10.107803] audit: type=1400 audit(1556613575.652:6): apparmor="STATUS" operation="profile_load"
profile="unconfined" name="/usr/sbin/tcpdump" pid=734 comm="apparmor_parser"
Apr 30 17:39:38 os41983 kernel: [ 10.108997] audit: type=1400 audit(1556613575.656:7): apparmor="STATUS" operation="profile_load"
profile="unconfined" name="/usr/lib/napd/snap-confine" pid=732 comm="apparmor_parser"
Apr 30 17:39:38 os41983 kernel: [ 10.109002] audit: type=1400 audit(1556613575.656:8): apparmor="STATUS" operation="profile_load"
profile="unconfined" name="/usr/lib/napd/snap-confine//mount-namespace-capture-helper" pid=732 comm="apparmor_parser"
Apr 30 17:39:38 os41983 kernel: [ 10.110566] audit: type=1400 audit(1556613575.656:9): apparmor="STATUS" operation="profile_load"
profile="unconfined" name="/sbin/dhclient" pid=729 comm="apparmor_parser"
Apr 30 17:39:38 os41983 kernel: [ 10.110571] audit: type=1400 audit(1556613575.656:10): apparmor="STATUS" operation="profile_load"
profile="unconfined" name="/usr/lib/NetworkManager/nm-dhcp-client.action" pid=729 comm="apparmor_parser"
Apr 30 17:39:38 os41983 kernel: [ 10.110575] audit: type=1400 audit(1556613575.656:11): apparmor="STATUS" operation="profile_load"
profile="unconfined" name="/usr/lib/NetworkManager/nm-dhcp-helper" pid=729 comm="apparmor_parser"
Apr 30 17:39:38 os41983 kernel: [ 12.551597] new mount options do not match the existing superblock, will be ignored
Apr 30 17:39:42 os41983 kernel: [ 17.303146] random: crng init done
Apr 30 17:39:42 os41983 kernel: [ 17.303148] random: 7 urandom warning(s) missed due to ratelimiting
Apr 30 17:40:07 os41983 kernel: [ 42.501975] HCPARK: sys_kb_enqueue() CALLED!! item=123
Apr 30 17:40:07 os41983 kernel: [ 42.501977] DUMP! count=1
Apr 30 17:40:07 os41983 kernel: [ 42.501978] 0 123
Apr 30 17:40:07 os41983 kernel: [ 42.501979] 1 0
Apr 30 17:40:07 os41983 kernel: [ 42.501980] 2 0
Apr 30 17:40:07 os41983 kernel: [ 42.501981] 3 0
Apr 30 17:40:07 os41983 kernel: [ 42.501982] 4 0
Apr 30 17:40:07 os41983 kernel: [ 42.501997] HCPARK: sys_kb_dequeue() CALLED!! addr=0x00000000c0d31a4f
Apr 30 17:40:07 os41983 kernel: [ 42.501999] DUMP! count=0
Apr 30 17:40:07 os41983 kernel: [ 42.502000] 0 -7777
Apr 30 17:40:07 os41983 kernel: [ 42.502001] 1 0
Apr 30 17:40:07 os41983 kernel: [ 42.502002] 2 0
Apr 30 17:40:07 os41983 kernel: [ 42.502003] 3 0
Apr 30 17:40:07 os41983 kernel: [ 42.502004] 4 0
```

K-board 서비스 구현



명세

- `long sys_kb_enqueue(int clip)`
 - 기능: Clip을 받아 내부 링버퍼에 저장
 - 문제없이 성공한 경우 0 리턴
 - 에러 처리
 - 링버퍼가 가득 찬 상태이면 에러 상태 출력하고, -1 리턴
 - 전달받은 clip이 음수이면 에러 상태 출력하고, -2 리턴
- `long sys_kb_dequeue(int *user_buf)`
 - 기능: 링버퍼의 clip을 하나 꺼내어 유저에게 전달
 - `user_buf`가 가리키는 유저 영역으로 clip 값을 복사해서 전달
 - 문제없이 성공한 경우, 0을 리턴
 - 에러 처리: -1 리턴
 - 링버퍼가 비어있는 상태이면 에러 상태 출력하고, -1 리턴

링 버퍼

- int 형 배열을 선언해서 사용할 것
- 최대 저장 개수는 5개. 변경하지 말 것.
- 코드 예)
 - #define MAX_CLIP (5)
 - int ring[MAX_CLIP];



유저 영역으로 데이터 복사

- Why?

- Kernel area 에 저장된 값은 유저 어플리케이션이 접근 불가
- 따라서 커널이 user area 의 어떤 위치에 값을 복사해주어야 함
- User는 값을 저장할 메모리 주소값을 커널에 인자로 넘겨주어야 함 (*user_buf)

- How?

- #include <linux/uaccess.h>
- copy_to_user() 함수 이용

- 참고

- 인자로 전달받은 값은 실제로 레지스터를 통해 전달
- 따라서 다시 유저-커널 간 복사할 필요가 없음
- 주소를 받은 경우, 유저-커널 간 복사를 해야 실제 데이터를 얻어올 수 있음

```
static __always_inline unsigned long __must_check
copy_to_user(void __user *to, const void *from, unsigned long n)
{
    int sz = __compiletime_object_size(from);

    kasan_check_read(from, n);
    might_fault();

    if (likely(sz < 0 || sz >= n)) {
        check_object_size(from, n, true);
        n = _copy_to_user(to, from, n);
    } else if (!__builtin_constant_p(n))
        copy_user_overflow(sz, n);
    else
        __bad_copy_user();

    return n;
}
```



수행 팁

- Ring buffer 알고리즘 구현은 유저 프로그램으로 먼저 수행
 - 커널 내에서는 수정-테스트 시간이 너무 오래 걸림
 - 수정 후 커널 컴파일: 3분
 - 수정 후 커널 재설치: 2분
 - 재부팅: 1분 (정상적으로 잘 되면)
 - 유저 프로그램으로 먼저 테스트를 충분히 한 다음에, 커널 코드로 이식하는 것이 보다 빠른 작업 방법
- Divide and conquer!
 - 단계를 분할해서 확실하게 동작을 하나씩 확인하며 진행할 것
- make clean
 - 커널 코드의 동작이 의심스러울 때는 컴파일 결과를 모두 제거하고, 완전히 새로 컴파일해볼 필요도 있음
 - Header 등의 수정이 이루어진 경우, make 만 반복하면 적용이 안될 수 있음

수행 팁

- 참고할 만한 시스템콜 구현
 - asm linkage long sys_open(const char __user *filename, int flags, umode_t mode);
 - 구현: fs/open.c

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
    if (force_o_largefile())
        flags |= O_LARGEFILE;

    return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

- 실제 처리: do_sys_open()



유저 프로그램 확장:

library, copy, paste



유저 프로그램 구성

- kboard.h
 - Kboard 커널 서비스에 대한 high level API (사용자 인터페이스) 제공
 - long kboard_copy(int clip);
 - int kboard_paste(int *clip);
 - Syscall() 함수를 이용해 커널 내의 enqueue, dequeue 호출
 - 마치 C 라이브러리 처럼 동작
- copy.c
 - kboard.h를 include 해서 kboard_copy() 서비스 이용
 - 사용자에게 copy 할 내용 (int 정수형)을 인자로 받음
- paste.c
 - kboard.h를 include 해서 kboard_paste() 서비스 이용

컴파일 및 실행

- gcc -o copy copy.c
- gcc -o paste paste.c
- ./copy 111
- ./paste

```
hcpark@hcpark-VirtualBox:~/user$ gcc -o copy copy.c
hcpark@hcpark-VirtualBox:~/user$
hcpark@hcpark-VirtualBox:~/user$ gcc -o paste paste.c
hcpark@hcpark-VirtualBox:~/user$ ./copy 111
Copy succes: 111
hcpark@hcpark-VirtualBox:~/user$ ./paste
Paste success: 111
hcpark@hcpark-VirtualBox:~/user$ █
```



테스팅 및 버그 수정



수행 내용

- `sudo dmesg -c`
- `sudo dmesg -c`
- Paste 1 번
 - Empty 상태이므로 에러 및 `sudo dmesg -c` 확인
- Copy 3 번
- Paste 1 번
- Copy 4 번
 - 마지막 copy 때 full 상태이므로 에러 및 `sudo dmesg -c` 확인
- Paste 5 번
 - Copy 된 순서대로 출력되는 것을 확인
- Copy 1 번 (음수 삽입)
 - 에러 및 `sudo dmesg -c` 확인

결과 예시 1/2

```
hcpark@hcpark-VirtualBox:~/user$ sudo dmesg -c
hcpark@hcpark-VirtualBox:~/user$ sudo dmesg -c
hcpark@hcpark-VirtualBox:~/user$ ./paste
Fail: KBoard is empty
hcpark@hcpark-VirtualBox:~/user$ sudo dmesg -c
[ 2675.183717] HCPARK: sys_my_dequeue() CALLED!!
[ 2675.183718] HCPARK: sys_kb_dequeue() ring is empty!!
hcpark@hcpark-VirtualBox:~/user$ ./copy 111; ./copy 222; ./copy 333;
Copy succes: 111
Copy succes: 222
Copy succes: 333
hcpark@hcpark-VirtualBox:~/user$ ./paste
Paste success: 111
hcpark@hcpark-VirtualBox:~/user$ ./copy 444; ./copy 555; ./copy 666; ./copy 777
Copy succes: 444
Copy succes: 555
Copy succes: 666
Fail: KBoard is full or invalid clip
hcpark@hcpark-VirtualBox:~/user$ sudo dmesg -c
[ 2730.037827] HCPARK: sys_my_enqueue() CALLED!! clip = 111
[ 2730.038581] HCPARK: sys_my_enqueue() CALLED!! clip = 222
[ 2730.039723] HCPARK: sys_my_enqueue() CALLED!! clip = 333
[ 2734.401185] HCPARK: sys_my_dequeue() CALLED!!
[ 2741.188791] HCPARK: sys_my_enqueue() CALLED!! clip = 444
[ 2741.189599] HCPARK: sys_my_enqueue() CALLED!! clip = 555
[ 2741.190261] HCPARK: sys_my_enqueue() CALLED!! clip = 666
[ 2741.190787] HCPARK: sys_my_enqueue() CALLED!! clip = 777
[ 2741.190788] HCPARK: sys_my_enqueue() ring is full!!
```



결과 예시 2/2

```
hcpark@hcpark-VirtualBox:~/user$ ./paste; ./paste; ./paste; ./paste; ./paste
Paste success: 222
Paste success: 333
Paste success: 444
Paste success: 555
Paste success: 666
hcpark@hcpark-VirtualBox:~/user$ ./copy -1
Fail: KBoard is full or invalid clip
hcpark@hcpark-VirtualBox:~/user$ sudo dmesg -c
[ 2756.626203] HCPARK: sys_my_dequeue() CALLED!!
[ 2756.627025] HCPARK: sys_my_dequeue() CALLED!!
[ 2756.628021] HCPARK: sys_my_dequeue() CALLED!!
[ 2756.628698] HCPARK: sys_my_dequeue() CALLED!!
[ 2756.629357] HCPARK: sys_my_dequeue() CALLED!!
[ 2762.772296] HCPARK: sys_my_enqueue() CALLED!! clip = -1
[ 2762.772297] HCPARK: sys_my_enqueue() invalid integer clip = -1
hcpark@hcpark-VirtualBox:~/user$
```



순서

0. J-Cloud snapshot을 이용한 instance 생성
1. K-board skeleton 구현
2. 시스템콜 호출을 위한 유저 프로그램 작성
3. K-board 서비스 구현
4. 유저 프로그램 확장: library, copy, paste
5. 테스트 및 버그 수정
6. (+) 동기화 문제 확인 및 해결

동기화 문제 확인 및 해결

* 추가 점수 3점



동기화 문제 확인

- 현재 코드는 Synchronization 슬라이드 첫 머리에 나온 예제와 똑같음
 - count 변수에 의해 동기화 문제가 발생할 수 있음
- 문제 확인
 - Make a race condition
 - enqueue, dequeue 동작을 병렬적으로 계속 수행하여 race condition 발생시킴
 - 동기화 문제가 발생하기 전, 수행한 횟수를 표시할 것
 - Data validation
 - 동기화 문제가 발생하였음을 알 수 있도록, 미리 약속된 형태의 데이터를 enqueue 하고, dequeue 했을 때 그 값을 확인할 것
 - Buffer initialization
 - 버퍼 내에서 실제로 값이 저장되지 않은 entry에서 값을 가져오는 상황이 생길 수도 있음. 따라서 초기에, 그리고 dequeue 했을 때 빈 entry 에는 임의의 초기값을 설정하여 이를 파악할 수 있도록 함

동기화 문제 확인: 결과 예시

- Data validation: 기대값과 dequeue 로 꺼내온 값을 비교
- 두 값이 틀린 경우, 현재의 iteration 횟수를 출력하고 종료
- 다양한 timing 에 결과가 나올 수 있음
 - 거의 대부분, 1 초 이상 수행되지 않을 것
- 한 번 동기화 문제가 발생한 다음에는?
 - Count 값이 이미 잘못되었기 때문에, 계속해서 문제가 즉각 발생할 수 있음
 - 그럼 다시 확인할 때마다 재부팅을?
 - Buffer 와 Count 값을 초기화시켜주는 시스템콜을 추가할 수 있음

```
root@os41983:~/proj1# ./paste
Validation fault! current=15744 mustbe=15739 iteration=15743
```

```
root@os41983:~/proj1# ./paste
Validation fault! current=-7777 mustbe=0 iteration=3446430
```

```
root@os41983:~/proj1# ./paste
Validation fault! current=-7777 mustbe=0 iteration=0
```

동기화 문제 해결

- (spinlock 을 배운 다음에 다시 살펴볼 것)
- Spinlock
 - 가장 단순한 형태의 동기화 기법
 - 한 번에 하나의 프로세스만 획득할 수 있는 lock 을 정의
 - Lock을 획득해야만 공유 자원을 접근 및 수정할 수 있도록 프로그래밍
 - Lock을 획득하기 위해서는 spin (polling)을 하며 lock의 상태를 계속 점검함
- include/linux/spinlock.h
 - `spinlock_t lock; //spinlock 에서 사용할 lock 선언`
 - `spin_lock_init(&lock); //lock 변수 초기화`
 - `spin_lock(&lock); //lock 획득을 위해 spin 하며 대기`
 - `spin_unlock(&lock); //획득한 lock 의 반환`

그래서 해결되었나?

- 아무 문제 없이 validation이 10초 이상 성공하면 해결된 것으로 볼 수 있음

```
Last login: Tue Apr 30 23:39:38 2019 from 59.2.5.105
hcpark@os41983:~$ sudo -s
root@os41983:~# cd proj1/
root@os41983:~/proj1# ./copy
root@os41983:~/proj1# ./copy
root@os41983:~/proj1# █

^C
root@os41983:~/proj1# time ./paste
^C

real    0m17.408s
user    0m6.741s
sys     0m10.668s
root@os41983:~/proj1# █
```

- 더 파보고 싶으면?
 - 성능을 테스트해보자. 1초동안 몇 번이나 수행되는가?
 - 동기화 기법은 필연적으로 성능을 떨어뜨림

수행 팁

- 대단히 많은 횟수의 시스템콜 호출이 이루어지므로, debugging message 는 끄거나, 최소한으로 출력하여야 함
 - 1000번에 한번, 혹은 error 만 출력
 - 그렇지 않으면, 시스템 로그 파일로 인해 용량이 가득차는 문제가 발생할 것
 - 로그 용량 확인: `cd /var/log; du -alhd 1`
 - 로그 비우기
 - `Cat /dev/null > /var/log/syslog`
 - `Cat /dev/null > /var/log/kern.log`

보고서 작성

- 동기화 관련해서 추가로 작성된 내용은 A4 3장 이하로 작성
- 동기화 문제를 어떤 방식으로 확인하였는지,
- 동기화 문제를 어떻게 해결하였는지 설명할 것
- 추가로 더 고려한 부분이나 진행한 부분이 있다면 꼭 명시할 것