<운영체제 1차 과제 – 시스템 콜 추가 및 이해>

2017320123 김장영

4/24일 제출 (Free day: 0일)

1. 개발환경

- Oracle VM VirtualBox (Host: Window)
- VM: 4core process, 4096MB memory, KVM Para-Virtualization
- 리눅스 Ubuntu 18.04 LTS – 64bit
- Kernel Version 4.20.11

2. 리눅스의 시스템 콜과 호출 루틴

2.1. 시스템 콜이란

System Call 은 커널의 서비스를 제공받기 위한 유저 스페이스의 요청이며, 운영 체제 커널은 많은 서비스를 제공합니다. 프로그램이 파일에 뭔가를 쓰거나 읽으려 할 때, 소켓 연결을 위해 listen 을 시작할 때, 디렉토리를 생성하거나 삭제할 때, 심지어는 해당 작업을 끝낼 때 프로그램이 시스템 호출(System call)을 사용합니다. 다시 말하면, System Call 은 유저 스페이스의 프로그램이 일부 요청들을 처리하기 위해 호출하는 커널 스페이스에 작성되어 있는 c 언어 함수일 뿐입니다. 커널에서 제공하는 protected 서비스를 user application이 이용하기 위해 user mode에서 kernel mode로 진입하는 통로라고도 볼 수 있습니다. 이러한 함수들의 헤더 파일이 syscalls.h 이며 원한다면 임의의 System Call 을 추가해볼 수도 있습니다. 리눅스 커널은 이러한 함수들의 세트를 제공하며 각 아키텍처에서 자체적인 세트를 제공합니다 System Call 은 고유한 번호가 지정되어 있으며 이는 System Call table 에 저장되어 있습니다. 리눅스 커널이 제공하는 System Call 의 개수는 아키텍처마다 다를 수 있습니다.

2.2 호출 루틴

대부분의 System Call은 libc를 통한 Wrapper Function의 형태로 제공받을 수 있습니다. 유저가 작성한 프로그램에서 system call이 발생하면 다음과 같은 과정에 따라 호출이 일어나게 됩니다. 예를 들어 응용 프로그램에서 open(...) 함수를 실행하면 libc에서 제공한 open 함수 속에서 인수 데이터(argument)를 레지스터에 넣고 Trap(소프트웨어 인터럽트)을 실행합니다. 이때 전달해야 할 인수 데이터가 많을 경우 메모리에 테이블 혹은 블럭을 선언하여 해당 주소값과 크기를 레지스터에 넣습니다.

트랩에 의해 TSR(Trap Service Routine)이 있는 커널의 트랩 처리 위치를 찾아 해당 주소로 실행을 옮깁니다. 이때 CPU는 권한수준이 최고수준인 실행모드가 됩니다. 이 과정은 CPU의 트랩 처리 메커니즘에 의해 자동 변환됩니다. mydrv_open()함수의 **return**에 따라 커널의 함수 호출이 완료되고, 커널은 다시 해당 응용 프로그램을 스케줄링에 의해 활성화하고 해당 프로세서로 전환합다. 이 때 CPU의 권한수준은 다시 사용자 모드로 전환됩니다.

만약 write 함수나 read 함수의 호출에 의해, 커널의 함수가 호출된 후 return에 의해 종료되지 않으면 응용 프로그램은 스케줄링에서 빠져 커널의 상태에서 머물면서 블럭킹 현상이 발생하는데, 이런 경우 해당 드라이버의 인터럽트 등으로 블럭킹을 해제할 수 있습니다.

3. 수정 및 작성한 부분과 설명

3. 1. linux-4.20.11/arch/x86/entry/syscalls/syscall_64.tbl

리눅스에서 제공하는 모든 시스템 콜의 고유 번호가 저장되어 있는 테이블로서 시스템 콜의 symbol 정보의 집합입니다. 커스텀 시스템 콜을 추가하기 위해서는 제일 먼저 이 테이블에 system call 번호를 추가해주어야 합니다. Linux kernel source tree에 흩어져 있는 시스템 콜의 함수의 주소들을 저장하는 테이블로서 시스템 콜 주소는 링커가 자동으로 관리합니다. 아래와 같이 335에 oslab_enqueue를, 336에 oslab_dequeue를 지정해 주었습니다.

```
# oslab
# Add custom syscall symbol to Sys_call_table
335     common  oslab_enqueue           __x64_sys_oslab_enqueue
336     common  oslab_dequeue           __x64_sys_oslab_dequeue
```

<syscall_64.tbl>

3. 2. Linux-4.20.11/include/linux/syscalls.h

다음으로는 시스템 콜 함수들의 prototype을 정의해야 합니다. Syscalls.h 파일에 asmlinkage를 사용하여 등록합니다. Asmlinkage를 사용하는 이유는 시스템 콜 호출은 int 80 인터럽트 핸들러에서 호출되는데 인터럽트 핸들러는 assembly 코드로 작성되기 때문입니다. Amlinkage를 함수 앞에 선언하면, assembly code에서도 c함수 호출이 가능해집니다. 아래와 같이 sys_oslab_enqueue(int)와 sys_oslab_enqueue(void)를 추가해 주었습니다.

```
/*
 * oslab
 * syscall prototype define
 * --> can call from assembly code
 */

asmlinkage void sys_oslab_enqueue(int);
asmlinkage int sys_oslab_dequeue(void);
```

<syscalls.h>

3. 3. Linux-4.20.11/kernel/my_queue_syscall.c

이제 실제 시스템 콜 함수을 작성해 주어야 합니다. 구현할 함수는 각각 전역으로 선언된 queue에 enqueue하는 함수, dequeue하는 함수입니다.

```
/*
 *  linux/kernel/my_queue_syscall.c
 *
 *  2021-04-09  Jangyoung Kim - Korea Univ.
 *      Add custom syscall for queueing
 *      oslab_enqueue(335); enqueue
 *      oslab_dequeue(336); dequeue
 */

#include <linux/syscalls.h>
#include <linux/kernel.h>
#include <linux/linkage.h>

// Declare helper functions
int IsEmpty(Queue *queue);
int IsAlreadyExist(Queue *queue, int new_input);
void PrintQueue(Queue *queue);
```

```
// Define Node
typedef struct Node {
    int data;
    struct Node *next;
}Node;

// Define Queue
typedef struct Queue
{
    Node *front;
    Node *rear;
    int count;
    int point_num;
}Queue;
```

<my_queue_syscall.c>

우선 Helper Function(isEmpty, isAlreadyExist, printQueue)들과 구현에 사용할 Queue 구조체를 선언해. 주었습니다. Helper function들은 각각 큐가 비었는지 확인, 큐에 이미 있는 값인지 확인, 큐의 값을 출력 하는 역할을 하고, 별도로 설명하지 않겠습니다.

```
/** Main Routine **/
/* Declare Global Queue */
struct Queue queue = {NULL,NULL,0, 0};
struct Node n[MAX_LENGTH] = {0, };
```

<Declare Global Queue>

다음으로 Enqueue, Dequeue 함수를 작성하기 전에 Global Queue를 선언, 초기화 해주었습니다. 선언된 큐에 enqueue, dequeue 함수가 접근하여 작업을 실행합니다.

```
/* Enqueue Syscall */
SYSCALL_DEFINE1(oslab_enqueue, int, a) {
    Node *now = n;
    now += sizeof(Node)*queue.point_num;// Create Node & Node initialize to enqueue
    now->data = a;
    now->next = NULL;

    if (IsEmpty(&queue)) {              // if queue is empty
        queue.front = now;                  // set front to new node
    }
    else {
        if (IsAlreadyExist(&queue, a)) {      // if input value is already exits -> break
        printk(KERN_INFO "[Error] - Already existing value \n");
        return;
        }
        queue.rear->next = now;           // else rear->next to new node
    }
    queue.rear = now;
    queue.count++;
    queue.point_num++;

    printk(KERN_INFO "[System call] oslab_enqueue(); -----\n");
    PrintQueue(&queue);
}
```

<oslab_enequeue>

전역으로 선언된 Queue에 인자로 받은 정수 a를 enqueue하는 함수입니다. System call을 작성할 때는 malloc을 사용할 수 없어서 빈 Node를 전역으로 선언해 둔 뒤 Queue에 들어있는 entry의 개수만큼 주소를 이동하여 새롭게 할당하도록 했습니다.

만약 큐가 비어있다면 전역 Queue 구조체의 front에 새로운 노드의 주소를 할당하며, rear또한 새로운 노드의 주소로 설정한 뒤, count를 1 증가시킵니다.

만약 큐가 비어있지 않다면, 정수 a가 이미 queue에 존재하는 값인지를 조사한 뒤, 존재하는 값이 아니라면 queue의 rear가 새로운 노드를 가리키도록 합니다. 이를 통해 연결리스트로 구현된 큐를 구현할 수 있습니다. 이후 queue의 rear를 새로운 노드로 설정한 뒤 count를 1 증가 시킵니다. 연결리스트를 통해 구현하게 되면 크기의 제한없이 queue를 사용할 수 있습니다. Enqueue가 완료되면 PrintQueue function을 이용하여 Queue 전체를 출력합니다.

```
/* Dequeue Syscall */
SYSCALL_DEFINE0(oslab_dequeue) {
    int re = 0;
    Node *now;
    if (IsEmpty(&queue)) {      // if queue is empty -> print "EMPTY QUEUE"
    printk(KERN_INFO "[Error] - EMPTY QUEUE--------------\n");
        return -2;
    }
    now = queue.front;
    re = now->data;             // Return value setting
    queue.front = now->next;
    queue.count--;          // discounting

    printk(KERN_INFO "[System call] oslab_dequeue(); ------\n");
    PrintQueue(&queue);
    return re;
}
```

<oslab_dequeue>

전역으로 선언된 Queue에서 1개의 node를 dequeue하는 함수입니다. 인자는 따로 없으며, 만약 Queue가 비어 있는데 dequeue를 수행하려 한다면 printk를 통해 error를 출력합니다.

Queue가 비어있지 않다면 선입선출에 따라 front 노드의 data를 return 값으로 설정하고, 해당노드를 Queue에서 제거해줍니다. Dequeue가 완료되면 PrintQueue를 이용하여 Queue 전체를 출력한 뒤, dequeue된 값을 리턴합니다.

3. 4. Linux-4.20.11/kernel/Makefile

```
# Makefile for the linux kernel.
#

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 my_queue_syscall.o
```

Kerel make를 통해 커널을 컴파일 할 때 작성한 syscall이 추가되도록 makefile을 수정합니다. 오브젝트 파일명은 앞서 작성한 my_queue_syscall.c 와 같은 my_queue_syscall.o로 작성해 줍니다.

3. 5. Call_my_queue

```c
/*
 *  call_my_queue.c
 *  2021-04-10  Jangyoung Kim - Korea Univ.
 *      Application using my_queue_syscall()
 */

#include <stdio.h>
#include <unistd.h>
#define my_queue_enqueue 335
#define my_queue_dequeue 336

int main(){
    int a;
    for(int i=1; i<=3; i++){
    a = syscall(my_queue_enqueue, i);
    printf("Enqueue : %d\n", i);

    }
    a = syscall(my_queue_enqueue, 3);
    printf("Enqueue : %d\n", 3);

    for(int i=1; i<=3; i++){
    a = syscall(my_queue_dequeue);
    printf("Dequeue : %d\n", a);
    }
}
```

중복을 피하기 위해 반복문을 사용하고, Syscall()이라는 매크로 함수를 이용하여 시스템 콜을 호출했습니다.

4. 실행결과 스냅샷

4. 1. call_my_queue

```
jay@jay-VirtualBox:~/hw1$ gcc call_my_queue.c -o call_my_queue
jay@jay-VirtualBox:~/hw1$ ./call_my_queue
Enqueue : 1
Enqueue : 2
Enqueue : 3
Enqueue : 3
Dequeue : 1
Dequeue : 2
Dequeue : 3
```

4. 2. dmesg

```
[  114.862196] [System call] oslab_enqueue(); -----
[  114.862198] Queue Front-----------------------
[  114.862199] 1
[  114.862199] Queue Rear--------------------
[  114.862339] [System call] oslab_enqueue(); -----
[  114.862340] Queue Front-----------------------
[  114.862341] 1
[  114.862342] 2
[  114.862342] Queue Rear--------------------
[  114.862347] [System call] oslab_enqueue(); -----
[  114.862348] Queue Front-----------------------
[  114.862349] 1
[  114.862349] 2
[  114.862350] 3
[  114.862350] Queue Rear--------------------
[  114.862354] [Error] - Already existing value
[  114.862356] [System call] oslab_dequeue(); ------
[  114.862357] Queue Front-----------------------
[  114.862358] 2
[  114.862358] 3
[  114.862359] Queue Rear--------------------
[  114.862362] [System call] oslab_dequeue(); ------
[  114.862363] Queue Front-----------------------
[  114.862363] 3
[  114.862364] Queue Rear--------------------
[  114.862372] [System call] oslab_dequeue(); ------
[  114.862373] Queue Front-----------------------
[  114.862373] Queue Rear--------------------
jay@jay-VirtualBox:~/hw1$
```

5. 숙제 수행 과정 중 발생한 문제점과 해결방법

5. 1. Queue의 길이 제한 및 malloc 사용불가

문제 조건에 int 배열 Queue를 선언하여 size를 제한해도 된다는 부분을 보지 못하여, 길이 제한을 해결하기 위해 연결리스트의 구조로 Queue를 구현하려 했습니다. 보통 C를 이용하여 연결리스트를 구현할 때는 Node 구조체 인스턴스에 malloc을 통해 동적으로 메모리를 할당하여 사용하곤 했었는데, system call을 구현할 때는 stdlib.h을 사용할 수 없다는 사실을 깨달았습니다. 검색을 통해 커널 프로그래밍시에 사용할 수 있도록 임시로 구현한 라이브러리가 있다라는 사실을 알게 되었으나, 적절한 라이브러리를 찾지 못하였고, 이미 연결리스트에 대한 로직을 구현한 상태여서, 다른 해결방법을 찾게 되었습니다.

Queue의 경우 전역 변수로 선언하여 사용하였는데 이와 마찬가지로 한 개의 노드를 임시로 전역 변수로 선언한 뒤, 실제 노드 사이즈 만큼의 주소를 더하거나 빼서 연결리스트를 구현해 보면 어떨까라고 생각했습니다. Queue에 point_num이라는 새로운 변수를 도입해 enqueue가 될 때 1씩 증가하도록 설정한 뒤, 전역변수로 선언된 Node 주소 값에 Node의 크기와 point_num을 곱하여 더해 주었습니다. 이후 dequeue가 될 때는 Null 값으로 초기화 해주게 됩니다.

하지만 한개의 노드를 전역변수로 선언해두고 주소를 증가시키면서 새롭게 할당하는 부분에서 결함을 발견했고, 이 부분은 전역변수로 설정된 메모리 영역을 넘어 힙 영역을 임의로 침범함에 따라 생기는 오류로 판단했습니다. 때문에 Node 리스트를 전역변수로 선언하여 접근하는 방법을 사용하게 되었습니다.

5. 2. 0을 enqueue 할 수 없는 문제

배열을 이용해 Queue를 구현하게 되면 생기는 또다른 문제점은 처음에 0을 이용하여 초기화를 하기 때문에 정수인 0을 Queue에 삽입할 수 없다는 것입니다. 배열 Queue에서 이를 해결하려면 배열을 탐색하여 count보다 적은 개수의 정수가 존재할 때, already exist를 안 띄우는 방법이 있는데, 연결리스트로 구현을 하게 되면 이러한 문제가 해결됩니다.