과제 #1

- 리눅스 CPU 스케줄러 분석

제출 기한: 5/1 (금) 23:59

제출 방법: IEILMS “과제 #1”

질의 응답 프로토콜

1. IEILMS Q/A 게시판 확인
    1. PLEASE!! (과제 0 종료 이후 조회 수가 거의 50 이하였음)

2. 인터넷 (구글) 검색
    1. 문제가 생겼을 때의 error message 를 문장 혹은 키워드로 구글 검색
    2. 한글 자료는 많이 없으므로, 영어 사용

3. IEILMS Q/A 질문답변 게시판에 질문 올리기
    1. JCLOUD 인스턴스 포트 번호 쓰기 (19xxx)
    2. 가능한 자세한 정보를 제공할 것 (명령 수행 및 결과 화면 등)

* 코드 등 과제 결과물이 노출되어야 하는 경우, 반드시 메일을 사용할 것
(문제가 있는 질문 글은 임의로 삭제할 수 있음)

추가 팁:

부트 메뉴 확인 설정 및

J-Cloud 스냅샷 설명

- 이 부분 먼저 하고 과제 #1 을 수행할 것

Tip. 부트메뉴 보기 (updated)

- $ sudo –s (시스템 관리를 위해 root 유저로 변경)
- # vi /etc/default/grub.d/50-cloudimg-settings.cfg
  - 부팅 관련 설정 변경. 첫 번째 파일
  - GRUB_RECORDFAIL_TIMEOUT=5
  - GRUB_TIMEOUT=5
    - 5초간 메뉴를 표시하도록 변경. 원래 값은 0. 위는 부팅 실패 상황에서의 설정
- # vi /etc/default/grub
  - 부팅 관련 설정 변경. 두 번째 파일
  - 메뉴가 보이도록 하고, 마지막 부팅 때 사용한 커널이 default 가 되도록 변경
    - GRUB_DEFAULT=saved
    - GRUB_SAVEDEFAULT=true (<- 새로 추가해야 함)
    - GRUB_TIMEOUT_STYLE=menu
    - GRUB_TIMEOUT=5
- # update-grub
  - 변경 내용을 적용하여 부트로더 재설정

Tip. 부트메뉴 보기: 수정할 파일 내용

```
# Cloud Image specific Grub settings for Generic Cloud Images
# CLOUD_IMG: This file was created/modified by the Cloud Image build process

# Set the recordfail timeout
GRUB_RECORDFAIL_TIMEOUT=5

# Do not wait on grub prompt
GRUB_TIMEOUT=5

# Set the default commandline
GRUB_CMDLINE_LINUX_DEFAULT="console=tty1 console=ttyS0"

# Set the grub console type
GRUB_TERMINAL=console
```

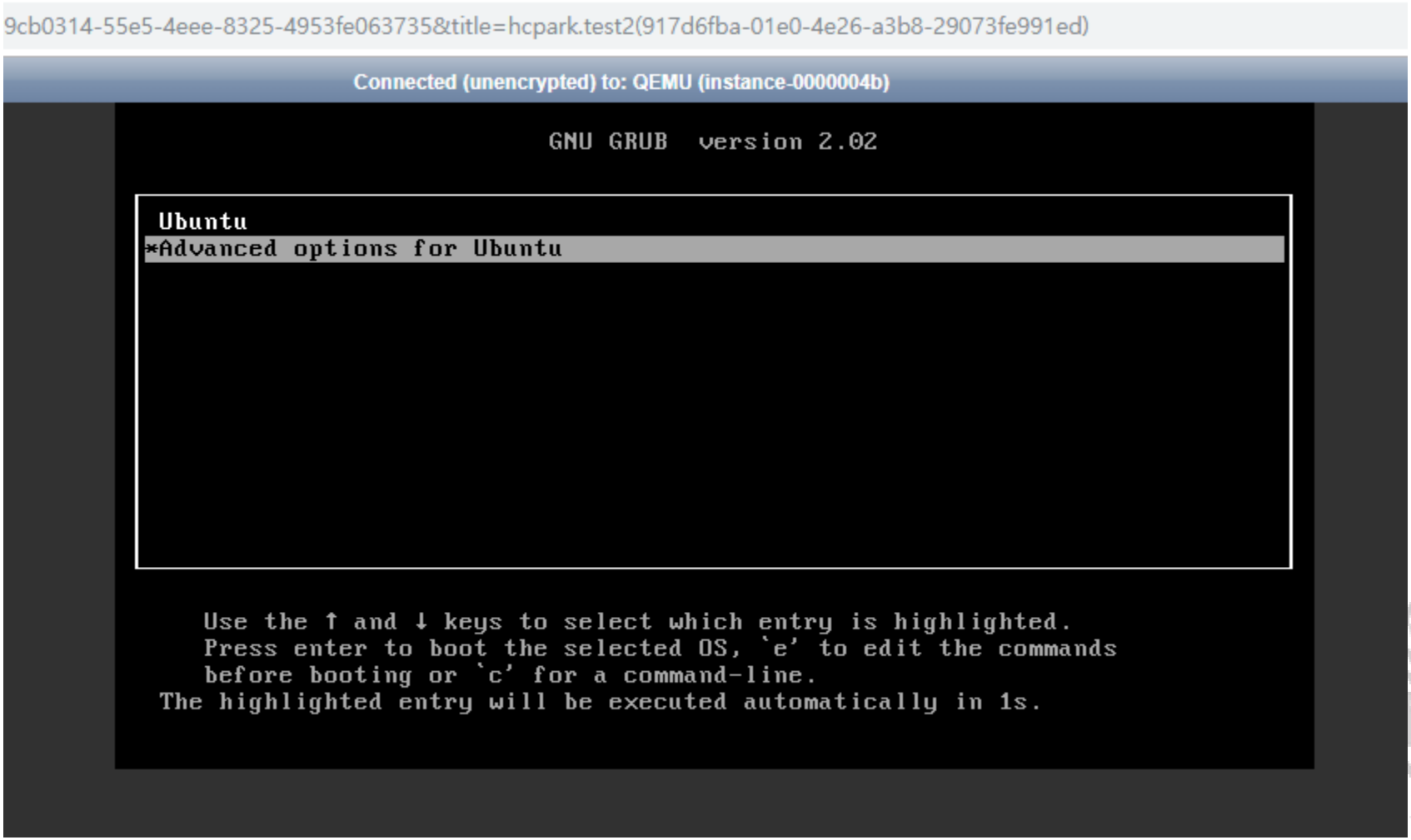
```
# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
#   info -f grub -n 'Simple configuration'

GRUB_DEFAULT=saved
GRUB_SAVEDEFAULT=true
GRUB_TIMEOUT_STYLE=menu
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="console=tty1 console=ttyS0"
GRUB_CMDLINE_LINUX=""

# Uncomment to enable BadRAM filtering, modify to suit your needs
```

```
root@hcpark:~/proj2# update-grub
Sourcing file `/etc/default/grub'
Sourcing file `/etc/default/grub.d/50-cloudimg-settin
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-4.18.20-hcpark-1000H
Found linux image: /boot/vmlinuz-4.18.20-hcpark-250HZ
Found linux image: /boot/vmlinuz-4.18.20-hcpark-100HZ
Found linux image: /boot/vmlinuz-4.18.20-hcpark
Found initrd image: /boot/initrd.img-4.18.20-hcpark
Found linux image: /boot/vmlinuz-4.18.20-hcpark.old
Found initrd image: /boot/initrd.img-4.18.20-hcpark
Found linux image: /boot/vmlinuz-4.15.0-96-generic
Found initrd image: /boot/initrd.img-4.15.0-96-generic
Found linux image: /boot/vmlinuz-4.15.0-91-generic
Found initrd image: /boot/initrd.img-4.15.0-91-generic
done
root@hcpark:~/proj2#
```

Tip. 부트메뉴 보기 (updated)

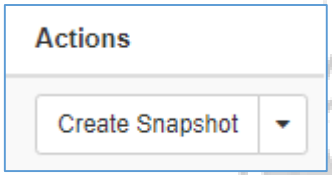
- J-Cloud 홈페이지에서 Console 화면을 켜놓고,
  SSH 클라이언트에서 reboot 수행
  - Instances 메뉴에서 본인 Instance 이름 선택
  - Console 메뉴 선택
  - “Click here to show only console” 선택하여 전체 화면 수행
    - 콘솔 화면 내를 클릭하여 조작 가능
    - 본인이 원하는 커널로 부팅
    - 5초 만에 선택하기 어려우면 숫자를 늘리면 됨
- J-Cloud console에서 바로 로그인하려면? (이것도 꼭 해둘 것)
  - SSH 접속한 후, $ sudo adduser user 입력 (“user”는 ID 예시)
  - 패스워드만 설정하고 나머지는 enter (분실하지 않도록 간단한 암호: 1234)
  - J-Cloud console 화면에서 “Click here to show only console” 선택하고 로그인
  - (키페어 분실 등으로 로그인 못하는 상황에서 긴급 복구 용으로 활용)
  - 해당 ID에 root 권한 부여: /etc/sudoers 수정 (참고)

Tip. 부트메뉴 보기: J-Cloud Console 화면



스냅샷? 백업!

- 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
  - 어플리케이션에 관한 수행, 환경 설정 내용도 똑같이 적용됨
  - 복원하기가 단순함
    - 일일이 파일 복사할 필요도 없고, 기존 파일이 있는 경우 대조를 하지 않아도 됨
- 모든 클라우드 시스템이 스냅샷 기능은 제공함
  - 저장 공간에 따라 요금이 부과될 수 있음에 유의 (J-Cloud는 무료)
- 수행 내용: 과제 #0을 완료한 인스턴스에 대해 스냅샷 작성

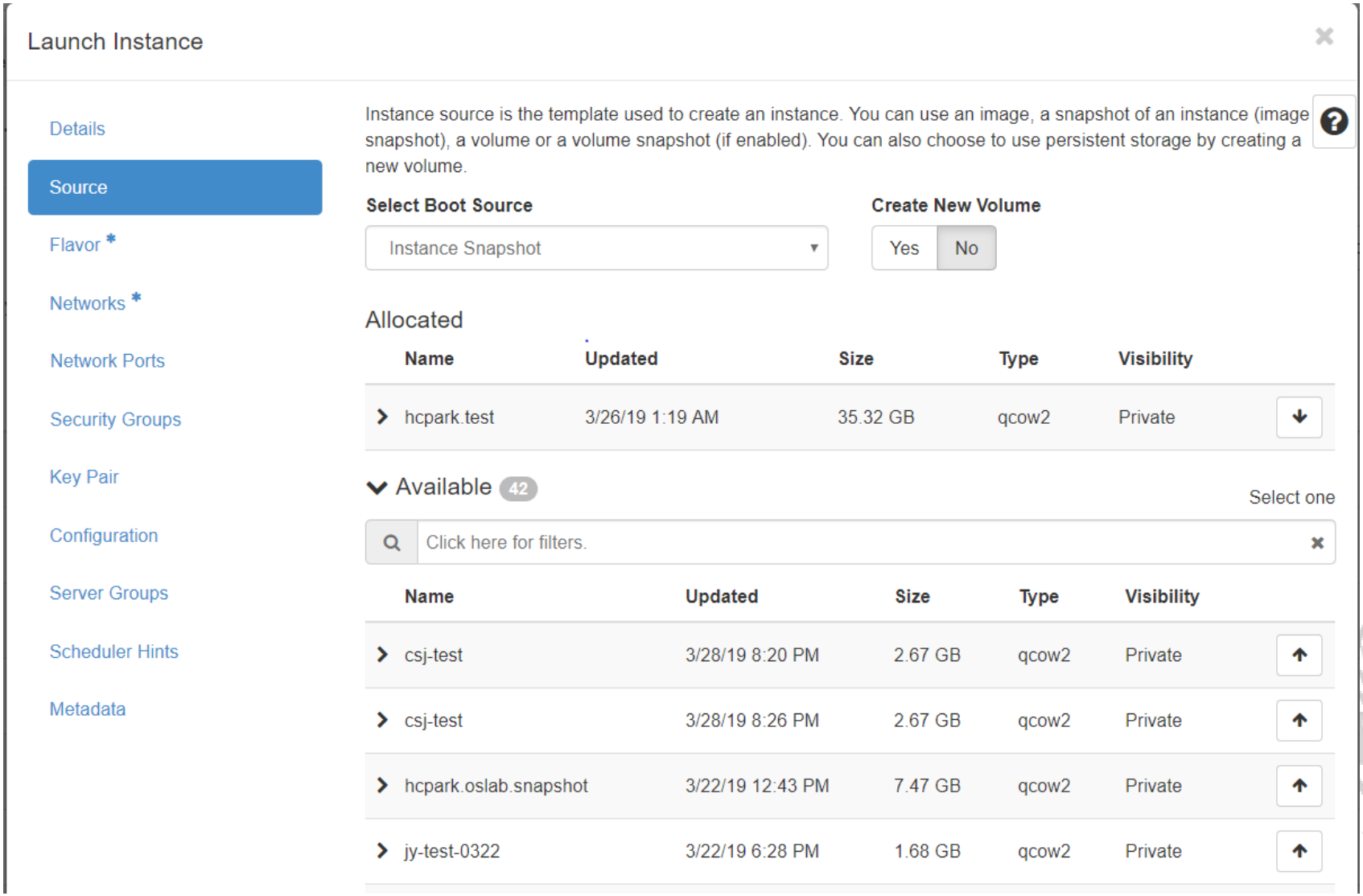


1. 인스턴스 정상 종료 (# shutdown -h now)
2. J-Cloud 접속, 내 인스턴스에 대해 오른쪽 끝 Actions 메뉴 중 Create Snapshot 선택
3. Image upload 등 진행 중 메시지가 표시됨. 약 10분 이상 걸리고, 용량은 30GB 이상
4. 작성이 완료된 후, 다음 슬라이드의 “스냅샷 기반 인스턴스 생성”으로 테스트

스냅샷 기반 인스턴스 생성

- 차후 스냅샷을 이용해 인스턴스를 복원하고자 할 때 사용함
  - 지금은 스냅샷만 생성해두고, 향후 진행 중에 문제가 생겼을 때 복원하여 사용
- 핵심
  - 인스턴스를 생성하던 기존 방법과 거의 동일함
  - 다만, image source를 image가 아닌 instance snapshot으로 설정
  - 그리고 본인의 snapshot을 지정
- 인스턴스 생성 후, SSH 접속하여 확인하고 진행
  - 기존과 동일하게 SSH 접속
    - 인스턴스 생성 시마다 ip는 달라질 수 있음
    - 그에 따라 SSH 의 접속 포트를 변경해야 함
  - 본인이 과제 #0에서 완료한 커널 버전으로 잘 부팅되었는지 확인
  - 기존 인스턴스 혹은 새로 생성한 인스턴스 둘 중 하나는 삭제할 것
    - 항상 1인당 1개의 인스턴스만 사용할 것

스냅샷 기반 인스턴스 생성



과제 #1

- 리눅스 CPU 스케줄러 분석

과제 학습 목표 및 내용

- 학습 목표
  - CPU 스케줄러 구조를 학습한다.
  - 커널 내에서의 프로그래밍을 수행해본다.
- 내용
  - 과제 1: 리눅스 CPU 스케줄러 분석
    - 1-1. 성능 평가에 사용할 유저 어플리케이션 구현
    - 1-2. Context switching 성능 분석
    - 1-3. 리눅스 CPU 스케줄러 분석
  - 과제 2: 리눅스에 새로운 CPU 스케줄러를 구현하고 성능 분석

- 과제 총 점수 50점 중 20점
  - 추가 점수를 받는 경우, 50점을 초과한 점수를 받을 수 있음
  - 비대면 강의가 지속됨에 따라 중간, 기말의 비중을 낮추고 과제 비중 높임

과제 제출 내용

- 보고서와 소스코드를 압축해서 1개 파일로 제출
- 보고서: 학번.pdf (* PDF 아니면 채점 안 함!!!! (NO PDF, NO SCORE!))
  - 표지: 제목, 학번, 이름 (1장)
  - 유저 프로그램 소스 코드 설명 (3장 이내)
  - 캡처 화면 3개 (분량 제한 없음)
    1. 5개 프로세스를 TIMESLICE 1ms 로 5초간 수행시켰을 때의 결과
    2. 5개 프로세스를 TIMESLICE 10ms 로 5초간 수행시켰을 때의 결과
    3. 5개 프로세스를 TIMESLICE 100ms 로 5초간 수행시켰을 때의 결과
  - 리눅스 CPU 스케줄러 소스 코드 분석 보고서 (분량 제한 없음)
  - 특히 어려웠던 점 및 해결 방법 (1장 이내)
- 소스코드
  - 유저 어플리케이션: 1개 파일 (cpu.c)

순서

1-1. 성능 평가에 사용할 유저 어플리케이션 구현

1-2. Context switching 성능 분석

1-3. 리눅스 CPU 스케줄러 소스 코드 분석

1-1. 성능 평가에 사용할 유저 어플리케이션 구현

- 리눅스 환경에서의 c 프로그래밍 작성
- 다양한 시스템콜의 활용

유저 어플리케이션 내용

- 내용: CPU 사용을 위해 단순 매트릭스 연산을 수행하는 C 프로그램
- Input: 각 프로세스의 연산 수행 시간, 수행 프로세스 개수
  - Command line arguments로 입력받을 것
- Output
  - 각 프로세스 별로 100ms 마다 메시지 출력
  - 종료 후, 수행된 매트릭스 연산의 총 횟수를 각 프로세스 별로 출력
- Usage: ./cpu <num of processes> <time to execute>
  - E.g. 5개 프로세스를 3초간 수행 -> $ **./cpu 5 3**
  - 이 경우, 전체 수행 시간은 3초이고, 5개 프로세스가 병렬적으로 각각 3초씩 수행됨
- 추가 수행 내용: Signal handling (추가 점수 3점)
  - CTRL+C 를 눌러서 강제 종료하였을 때, 전체 프로세스들을 종료하고, 각각의 수행 시간 출력 (Signal handling 사용: signal.h)
  - 수행한 경우, 보고서에 명시하고 추가 1장 이내로 설명할 것

매트릭스 연산

- 아래 소스 코드의 연산 부분은 그대로 사용하면 됨
- 필요에 따라 다른 부분들은 수정하여 사용

```
#define ROW (50)
#define COL ROW

int calc(int time, int cpu) {
        int matrixA[ROW][COL];
        int matrixB[ROW][COL];
        int matrixC[ROW][COL];
        int i, j, k;

        cpuid = cpu;

        while(1) {
                for(i = 0 ; i < ROW ; i++) {
                        for(j = 0 ; j < COL ; j++) {
                                for(k = 0 ; k < COL ; k++) {
                                        matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
                                }
                        }
                }
                count++;
                if(count%100 == 0) printf("PROCESS #%02d count = %02ld\n", cpuid, count);

        }

        return 0;
}
```

(Text version)

```
#define ROW (50)

#define COL ROW

int calc(int time, int cpu) {

    int matrixA[ROW][COL];

    int matrixB[ROW][COL];

    int matrixC[ROW][COL];

    int i, j, k;

    cpuid = cpu;

    while(1) {

        for(i = 0 ; i < ROW ; i++) {

            for(j = 0 ; j < COL ; j++) {

                for(k = 0 ; k < COL ; k++) matrixC[i][j] += matrixA[i][k] * matrixB[k][j];

            }

        }

        count++;

    }

}
```

결과 화면 예시

./cpu 1 1

```
root@hcpark:~/proj2# ./cpu 1 1
** START: Processes = 01 Time = 01 s
PROCESS #00 count = 0131 100 ms
PROCESS #00 count = 0270 100 ms
PROCESS #00 count = 0410 100 ms
PROCESS #00 count = 0557 100 ms
PROCESS #00 count = 0726 100 ms
PROCESS #00 count = 0893 100 ms
PROCESS #00 count = 1061 100 ms
PROCESS #00 count = 1227 100 ms
PROCESS #00 count = 1394 100 ms
PROCESS #00 count = 1560 100 ms
DONE!! PROCESS #00 : 001560 1003 ms
root@hcpark:~/proj2#
```

- 기존 부모 프로세스가 존재하고, fork() 를 통해 추가로 자식 프로세스들을 생성하여 전체 프로세스 개수를 달성
  - 따라서 화면에서 creating process 메시지는 요청한 프로세스 개수보다 1개 적게 나옴
- 최종 수행 시간은 다소 차이가 있을 수 있음
  - 1003ms, 3009ms, 5016ms 등

./cpu 3 3

```
root@hcpark:~/proj2# ./cpu 3 3
** START: Processes = 03 Time = 03 s
        Creating Process: #0
        Creating Process: #1
PROCESS #00 count = 0090 100 ms
PROCESS #02 count = 0132 100 ms
PROCESS #01 count = 0126 100 ms
PROCESS #00 count = 0222 100 ms
PROCESS #02 count = 0272 100 ms
PROCESS #01 count = 0261 100 ms
```

```
PROCESS #02 count = 4171 100 ms
PROCESS #01 count = 4346 100 ms
PROCESS #00 count = 4245 100 ms
PROCESS #02 count = 4338 100 ms
PROCESS #01 count = 4520 100 ms
PROCESS #00 count = 4419 100 ms
PROCESS #02 count = 4510 100 ms
PROCESS #01 count = 4694 100 ms
PROCESS #00 count = 4593 100 ms
PROCESS #02 count = 4684 100 ms
PROCESS #01 count = 4868 100 ms
PROCESS #00 count = 4767 100 ms
PROCESS #02 count = 4858 100 ms
PROCESS #01 count = 5042 100 ms
DONE!! PROCESS #01 : 005042 3007 ms
PROCESS #00 count = 4941 100 ms
DONE!! PROCESS #00 : 004941 3007 ms
PROCESS #02 count = 5032 100 ms
DONE!! PROCESS #02 : 005032 3009 ms
root@hcpark:~/proj2#
```

./cpu 5 5

```
root@hcpark:~/proj2# ./cpu 5 5
** START: Processes = 05 Time = 05 s
        Creating Process: #0
        Creating Process: #1
        Creating Process: #2
        Creating Process: #3
PROCESS #01 count = 0130 100 ms
PROCESS #00 count = 0131 100 ms
PROCESS #02 count = 0132 100 ms
PROCESS #04 count = 0077 100 ms
PROCESS #03 count = 0071 101 ms
PROCESS #00 count = 0269 100 ms
```

```
PROCESS #01 count = 8334 100 ms
PROCESS #00 count = 8336 100 ms
PROCESS #03 count = 8244 100 ms
PROCESS #04 count = 8247 100 ms
PROCESS #02 count = 8560 100 ms
DONE!! PROCESS #02 : 008560 5010 ms
PROCESS #01 count = 8507 100 ms
DONE!! PROCESS #01 : 008507 5012 ms
PROCESS #00 count = 8505 100 ms
DONE!! PROCESS #00 : 008505 5014 ms
PROCESS #03 count = 8418 100 ms
DONE!! PROCESS #03 : 008418 5015 ms
PROCESS #04 count = 8417 100 ms
DONE!! PROCESS #04 : 008417 5016 ms
root@hcpark:~/proj2#
```

사용할 시스템콜들

- fork()
  - 자식 프로세스 생성
  - fork()가 성공적으로 수행된 이후부터 두 개의 프로세스가 동일한 코드를 병렬로 수행함
  - http://man7.org/linux/man-pages/man2/fork.2.html
- clock_gettime()
  - 현재 및 상대 시간 가져오기
    - 과제에서는 실제 시간으로 사용 (CLOCK_MONOTONIC or CLOCK_REALTIME)
    - *_CPUTIME_ID 는 각 프로세스/스레드가 보는 가상 시간
  - 초 단위와 나노초 단위가 분리되어 있음에 유의
  - https://linux.die.net/man/3/clock_gettime
- 사용 방법을 숙지하고, 예제 프로그램을 돌려서 확인해보며 사용할 것

1-2. Context switching 성능 분석

- 리눅스 RT 스케줄링 클래스 Round robin 알고리즘 활용
- cgroup 기능을 활용한 CPU 자원 할당 설정
- Timeslice 에 따른 성능 분석

성능 분석 내용

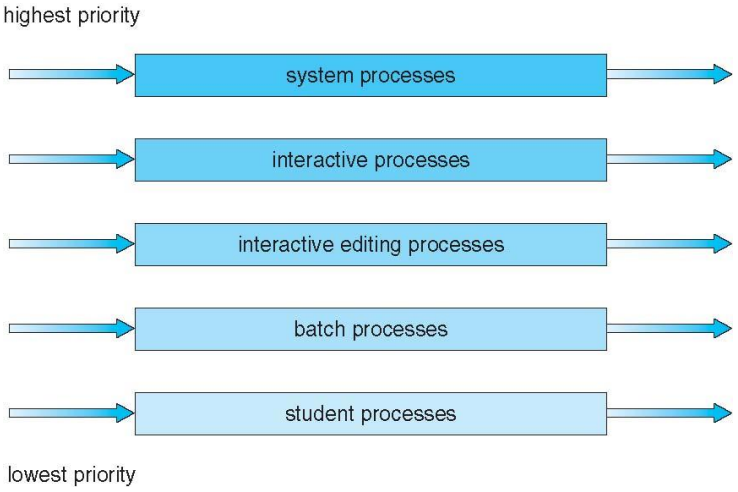
- 1-1에서 작성한 유저 어플리케이션 “CPU” 를 이용하여,
  time slice에 따른 작업 성능을 분석하고,
  이를 통해 context switching 으로 인한 overhead를 파악함

- 작업 내용
  - RT class의 Round Robin (RT-RR) 스케줄링을 이용해 Time slice를 1ms, 10ms, 100ms 로 변경
  - 각각의 경우에 대해 CPU 프로그램을 수행하여 성능을 분석함

- 작업 순서
  1. CPU 소스 코드를 수정하여 RT-RR 을 사용하도록 변경
  2. 1개의 코어만 사용하도록 cgroup 을 통해 설정 변경
     1. 여러 개의 코어를 쓰면, context switching 횟수 파악이 안되기 때문에,
        그로 인한 영향을 판단할 수 없음
  3. CPU 작업 수행

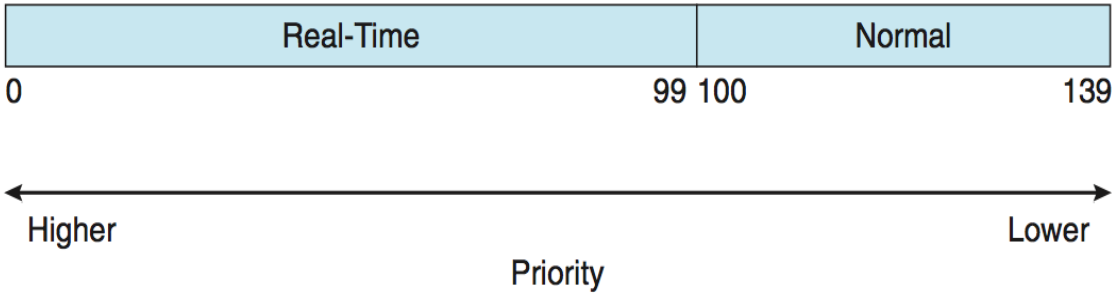
RT-RR: 리눅스 멀티레벨 스케줄링 구조

- Priority-based multilevel queue



linux/kernel/sched/sched.h

```
1302  #define sched_class_highest (&stop_sched_class)
1303  #define for_each_class(class) \
1304     for (class = sched_class_highest; class; class = class->next)
1305
1306  extern const struct sched_class stop_sched_class;
1307  extern const struct sched_class dl_sched_class;
1308  extern const struct sched_class rt_sched_class;
1309  extern const struct sched_class fair_sched_class;
1310  extern const struct sched_class idle_sched_class;
1311
1312
```



RT-RR: 스케줄링 정책 변경 방법

- sched_setattr() 시스템 콜 사용
  - SCHED_FIFO, SCHED_DEADLINE 등 class 및 정책을 지정 가능
  - 반드시 관련된 자료들 찾아보고 사용 (참고 링크)

Include/uapi/linux/sched.h

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

NAME
       sched_setattr, sched_getattr - set and get scheduling policy and attributes

SYNOPSIS
       #include <sched.h>

       int sched_setattr(pid_t pid, struct sched_attr *attr,
                         unsigned int flags);

       int sched_getattr(pid_t pid, struct sched_attr *attr,
                         unsigned int size, unsigned int flags);
```

```
00032: /*
00033:  * Scheduling policies
00034:  */
00035: #define SCHED_NORMAL        0
00036: #define SCHED_FIFO      1
00037: #define SCHED_RR        2
00038: #define SCHED_BATCH         3
00039: /* SCHED_ISO: reserved but not implemented yet */
00040: #define SCHED_IDLE      5
00041: #define SCHED_DEADLINE      6
```

- 기존 CPU App에서 fork() 수행 이전에 수행하여 모든 child process가 RT-RR로 스케줄링되도록 함
- RR-RR Timeslice 변경 방법
  - /proc/sys/kernel/sched_rr_timeslice_ms 파일에 원하는 timeslice 숫자를 기록 (ms 단위)
  - 예) 10ms 의 경우: # echo 10 > /proc/sys/kernel/sched_rr_timeslice_ms

Cgroup 을 이용한 CPU 코어 설정 및 작업 수행

1. Cgroup 을 이용해 1개의 CPU 코어만 사용하도록 설정

* 아래 설정은 ssh 로 접속한 해당 세션에서만 유효하고, 새로운 SSH 세션에는 적용 안됨.
  현재의 SSH 세션이 사용하는 Bash shell과 그 children 만 해당 cgroup 에 포함되기 때문.
  따라서 재부팅, 재접속 시에 항상 다시 수행하여야 함. Bash Script로 작성해두는 것도 좋음.

```
$ sudo -s
# cd /sys/fs/cgroup/cpuset        (cpuset에 대한 최상위 cgroup. 모든 프로세스 가 포함되어 있음)
# mkdir mycpu; cd mycpu           (cpuset에 대한 하위 cgroup 생성)
# echo 0 > cpuset.cpus            (이 cgroup은 0번 코어만 사용하도록 설정)
# echo 0 > cpuset.mems            (이 cgroup은 0번 메모리 노드만 사용하도록 설정. For NUMA)
# echo $$ > tasks                 (현재 수행 중인 bash shell 프로세스를 등록)
# cat tasks                       (잘 등록되었는지 확인)
```

2. 성능 테스트를 위한 작업 수행: ./cpu 5 10

* 위 설정을 수행한 SSH 세션에서 바로 이어서 진행하여야 함

```
# cd /home/ubuntu/                (user application 을 작성한 디렉토리로 이동)
# ./cpu 5 10                      (5개의 프로세스를 10초간 수행)
```

(참고) Cgroup: Control Group

- 리눅스에서 프로세스들에게 CPU, 메모리, 네트워크, 스토리지 장치의 사용량을 제어할 수 있도록 제공하는 기능
- 대상: 프로세스들로 구성된 작업 그룹
  - 사용자가 임의로 구성하며, 계층적으로 구성됨.
  - 즉, 부모 프로세스가 어떤 작업 그룹에 속해있으면, 자식 프로세스도 자동으로 포함됨
- 사용 방식
  - 작업 그룹 생성: Cgroup 으로 mount 된 디렉토리를 생성함
  - 작업 그룹에 프로세스 포함시키기: 해당 디렉토리의 tasks 파일에 프로세스 ID를 기록
  - 해당 디렉토리에서 필요한 설정 변경
- 사용 예
  - 특정 프로세스의 메모리 사용량을 1GB로 제한
  - 특정 프로세스 그룹의 CPU 사용량을 20% 로 제한
- 참고자료

수행 결과 예시: 2개 프로세스, 5초간 수행

RR Time slice	1ms		10ms		100ms	
	# of calc.	Time (ms)	# of calc.	Time (ms)	# of calc.	Time (ms)
Process #0	3797	5029	3873	5032	3880	5001
Process #1	3673	5032	3884	5032	4029	5002
Total calc. and Max time	**7470**	**5032**	**7757**	**5032**	**7909**	**5002**

RR Time slice	1ms	10ms	100ms
Calculations per second (total calc / max time)	1,484.50 (=7470/5.032)	1,541.53 (=7757/5.032)	1,581.17 (=7909/5.002)
Baseline=1ms	100.00%	103.84%	106.51%
Baseline=10ms	96.30%	100.00%	102.57%

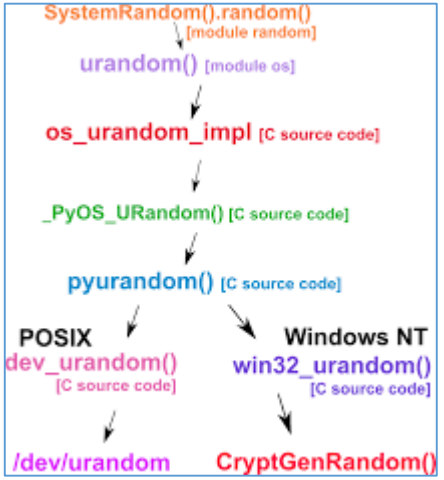
- 1ms 의 경우, 10ms와 비교 시 약 3.7%의 성능 하락이 있었음
- 100ms 의 경우, 10ms와 비교 시 약 2.57%의 성능 상승이 있었음
- 즉, Context switching이 10배 많아지는 경우, 약 3-4% 의 성능 하락이 있을 수 있음

1-3. 리눅스 CPU 스케줄러 분석

- Sched_setattr()을 통한 스케줄링 클래스 변경 Call chain 분석
- Scheduling class 및 Round Robin 코드 분석

수행 내용

- Sched_setattr()을 통한 스케줄링 클래스 변경 Call chain 분석
  - Sched_setattr() 의 호출로 CFS 클래스에서 RT-RR로 변경되는 작업 수행이
  - 어떤 함수들을 통해 어떤 방식으로 이루어지는 분석함
- Scheduling class 및 Round Robin 코드 분석
  - 리눅스에서 스케줄링 클래스가 어떻게 구성되고 관리되는지,
  - RT-RR 클래스의 소스 코드를 예시로 분석함
- 결과물: 보고서 1부 (캡처 화면은 보고서 내에 필요한 만큼 포함)
  - 보고서: 위 두 항목에 대해 관련된 소스 코드 내용을 포함
  - 캡처 화면: 관련된 함수들의 수행을 printk() 를 이용해 확인함

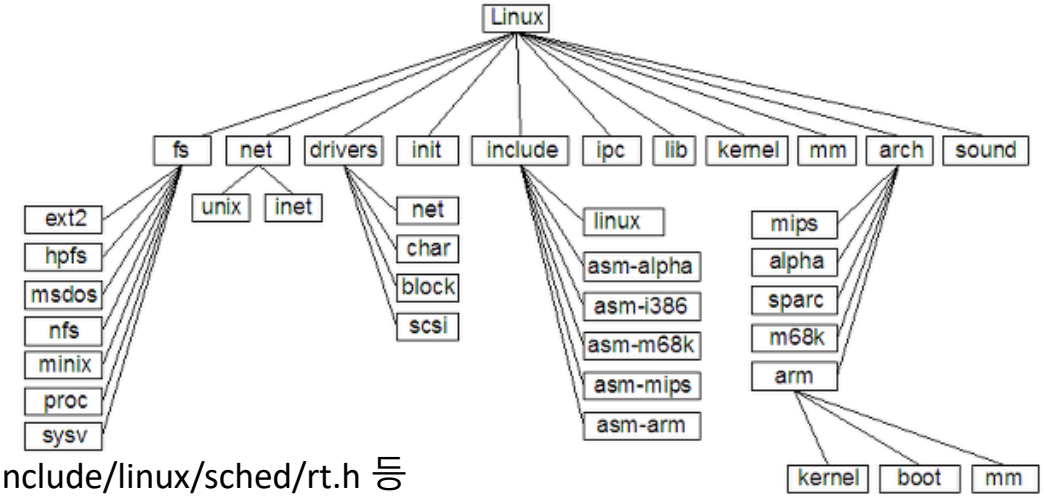


<함수 콜체인 예제>

- <u>과제 2 는 이 분석 내용을 토대로 새로운 스케줄링 클래스를 작성하는 내용</u>
  - 새로운 클래스를 만들고, 해당 클래스 내에서 자신만의 스케줄러 코드를 구현함
  - 따라서 스케줄링 클래스를 새롭게 만들어 넣는다는 목표를 갖고 소스 코드를 분석해야 함

리눅스 소스 코드의 분석

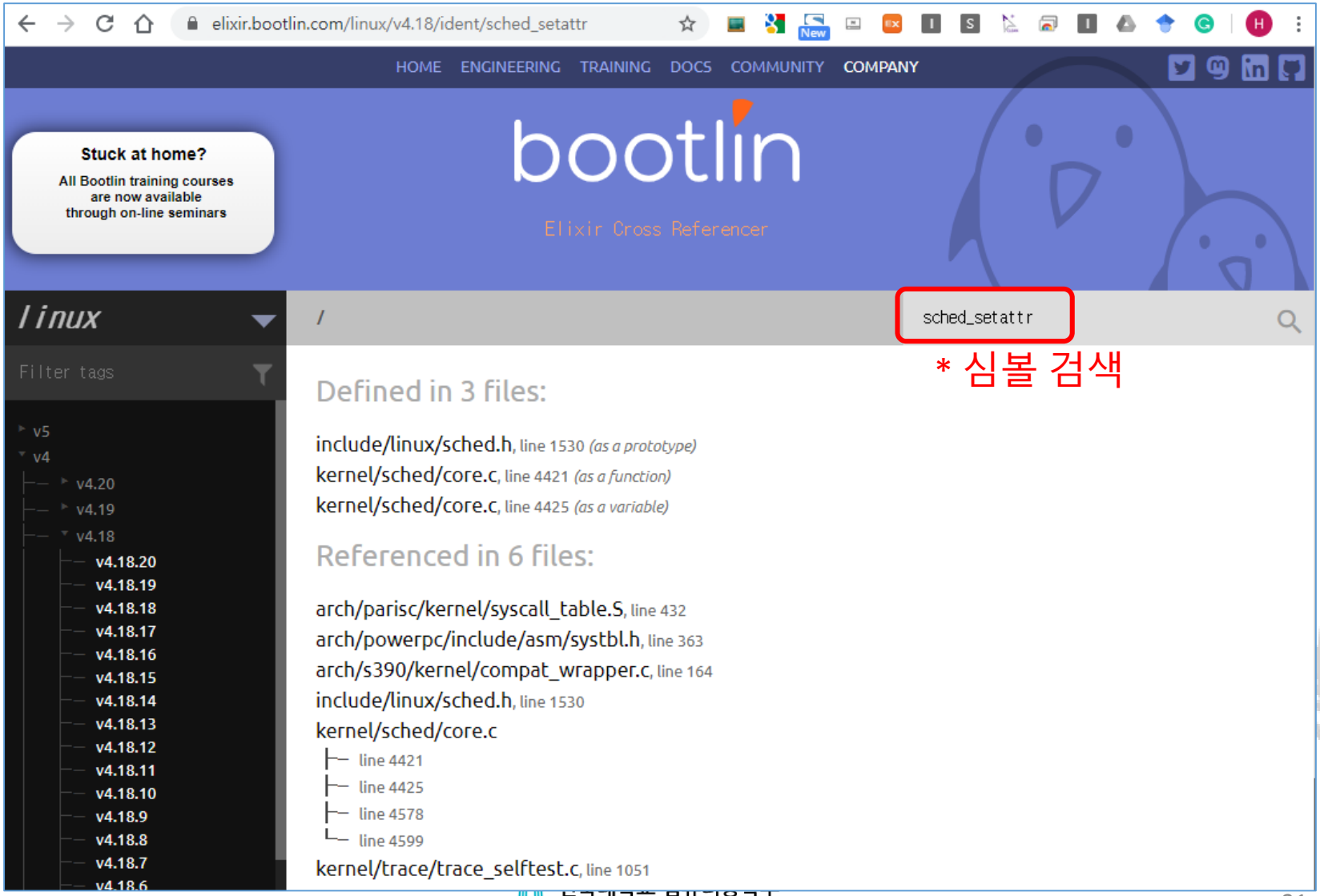
- 소스 디렉토리 구조
  - 과제에서 주로 참고할 디렉토리
    - Kernel 내의 sched 디렉토리
  - 참고 자료



- Sched 디렉토리 내 관련 파일
  - Core.c, rt.c 등
  - 관련 헤더 파일: include/linux/sched.h, include/linux/sched/rt.h 등
- 코드 분석 사전 작업
  - 분석 대상 시스템의 Architecture 및 동작을 파악해야 함
  - 리눅스 스케줄링 클래스의 구조를 공부하고, sched_setattr() 의 호출을 통해 변경되는 부분을 파악
- Call chain 분석의 entry point
  - 어디부터 분석을 시작할 것인가? (코드 분석 시, 가장 먼저 고려해야 할 사항)
  - Kernel 코드의 시작점은 시스템콜, 혹은 interrupt
  - Sched_setattr()을 실제로 처리하는 시스템콜 핸들러를 찾고,
    핸들러 내에서 호출되는 주요 함수를 따라가며 콜 체인을 파악함

코드 분석: Elixir Cross Referencer 활용

http://elixir.free-electrons.com/linux/v4.18/source



* 심볼 검색

커널 내 함수의 수행 확인

- printk() 함수
  - 커널 내에서 printf() 와 비슷한 역할을 하는 함수
    - Debug level 지정 가능: 과제에서는 KERN_DEBUG 사용
    - 출력 결과는 커널 메시지에 저장되며, dmesg 로 확인할 수 있음
- 수정 후에는 커널 재컴파일, 재인스톨, 재부팅
  - # time make -j 24 (24 or 48 무관함)
  - # time make -j 24 install
  - # reboot (재부팅 후, 본인 커널로 잘 부팅되었는지 확인)
- dmesg 명령어로 확인
  - 커널 메시지를 확인하는 명령
  - /var/log/kern.log 에서도 확인 가능
    (tail -f 명령을 이용하면 업데이트 되는 대로 바로 확인됨)
  - -c 옵션을 주면 현재까지의 메시지 삭제 가능 (루트 권한 필요)

확인 예시

- Sched_setattr() 시스템콜에 printk()를 넣고, CPU 수행 후 dmesg 확인

```
d=735 comm="apparmor_parser"
[   11.789702] audit: type=1400 audit(1586408488.270:9): apparmor="STATU
s" pid=735 comm="apparmor_parser"
[   11.789704] audit: type=1400 audit(1586408488.270:10): apparmor="STA
th-mounting" pid=735 comm="apparmor_parser"
[   11.789707] audit: type=1400 audit(1586408488.270:11): apparmor="STA
th-nesting" pid=735 comm="apparmor_parser"
[   14.798738] new mount options do not match the existing superblock, 
[   33.417195] random: crng init done
[   33.417199] random: 7 urandom warning(s) missed due to ratelimiting
[   48.603325] HCPARK: sched_setattr called
root@hcpark:~#
```

- 위와 유사하게 콜체인 분석 결과대로 함수들이 순서대로 잘 출력되는지 확인하고,
- 본인이 필요하다고 생각하는 주요 함수들의 호출 내용 및 결과 (파라미터, 결과값 등)를 자유롭게 추가함
- 반드시 관련된 설명을 보고서에 함께 기재할 것

참고자료

- Google search: Linux scheduler, linux scheduling architecture, linux scheduling class
- Documentation: In your kernel source tree
  - /v4.18/Documentation/scheduler
- Web-based Cross Reference
  - http://elixir.free-electrons.com/linux/v4.18/source
- About Linux Scheduler Class
  - http://www.cs.Columbia.edu/~krj/os/lectures/L12-LinuxSched.pdf
- Running Real-Time Tasks in Linux (x86 and ARM)
  - http://www.cse.wustl.edu/~lu/cse520s/slides/tutorial.pdf
- Others
  - https://www.ibm.com/developerworks/library/l-completely-fair-scheduler/