==========

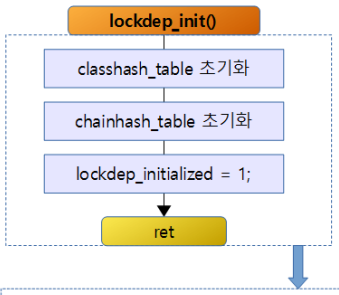
32주차 (2015.11.28)

==========

start_kernel()

```
asmlinkage __visible void __init start_kernel(void)
{
        lockdep_init();
        set_task_stack_end_magic(&init_task);
        smp_setup_processor_id();
        debug_objects_early_init();
        boot_init_stack_canary();
        cgroup_init_early();
        local_irq_disable();
        boot_cpu_init();
        page_address_init();
        setup_arch(&command_line);
        mm_init_cpumask(&init_mm);
        setup_command_line(command_line);
        setup_nr_cpu_ids();
        setup_per_cpu_areas();
        smp_prepare_boot_cpu();
        build_all_zonelists(NULL, NULL);
        page_alloc_init();
        parse_early_param();
        jump_label_init();
        setup_log_buf(0);
        pidhash_init();
        vfs_caches_init_early();
        sort_main_extable();
        trap_init();
        mm_init();
        sched_init();
        preempt_disable();
        local_irq_disable();
        idr_init_cache();
        rcu_init();
        trace_init();
        context_tracking_init();
        radix_tree_init();
        early_irq_init();
        init_IRQ();
        tick_init();
        rcu_init_nohz();
        init_timers();
        hrtimers_init();
        softirq_init();
        timekeeping_init();
        time_init();
        sched_clock_postinit();
        perf_event_init();
        profile_init();
        call_function_init();
        early_boot_irqs_disabled = false;
        local_irq_enable();
        kmem_cache_init_late();
        console_init();
        lockdep_info();
        locking_selftest();
        page_ext_init();
        debug_objects_mem_init();
        kmemleak_init();
        setup_per_cpu_pageset();
        numa_policy_init();
        late_time_init();
        sched_clock_init();
        calibrate_delay();
        pidmap_init();
        anon_vma_init();
        acpi_early_init();
        thread_info_cache_init();
        cred_init();
        fork_init(totalram_pages);
        proc_caches_init();
        buffer_init();
        key_init();
        security_init();
        dbg_late_init();
        vfs_caches_init(totalram_pages);
        signals_init();
        page_writeback_init();
        proc_root_init();
        nsfs_init();
        cgroup_init();
        cpuset_init();
        taskstats_init_early();
        delayacct_init();
        check_bugs();
        sfi_init_late();
        efi_late_init();
        efi_free_boot_services();
        ftrace_init();
        rest_init();
}
```

lockdep_init() - (1)



lock 디버깅을 위해 관련 테이블을 초기화 한다.

lockdep

lockdep는 lock(spinlock, mutex, semaphore 등)을 디버깅하기 위해 사용되며, 커널 빌드 시 커널 설정 메뉴에서 디버깅 옵션을 켜서 사용하며 /proc/lock* 화일을 통해 lock의 상태 확인 등을 할 수 있고 dead-lock 발생 시 printk 출력된다.

```
* 커널 빌드 설정 메뉴
Kernel hacking  --->
    Lock Debugging (spinlocks, mutexes, etc...)  --->
        [*] RT Mutex debugging, deadlock detection
        -*- Spinlock and rw-lock debugging: basic checks
        -*- Mutex debugging: basic checks
        [*] Wait/wound mutex debugging: Slowpath testing
        -*- Lock debugging: detect incorrect freeing of live locks
        [*] Lock debugging: prove locking correctness
        [*] Lock usage statistics
        [*] Lock dependency engine debugging
        [*] Sleep inside atomic section checking
        [*] Locking API boot-time self-tests
        <M> torture tests for locking
```

```
* 락 상태 확인용 화일
        /proc/lockdep
        /proc/lockdep_chains
        /proc/lockdep_stat
        /proc/locks
        /proc/lock_stats
```

```
#cat /proc/lockdep
all lock classes:
80a1c55c OPS:       19 FD:   50 BD:    2 +.+...: cgroup_mutex
 -> [80a1c6b0] cgroup_idr_lock
 -> [80a1c5b0] css_set_rwsem
 -> [80a30a74] devcgroup_mutex
 -> [80a1cf3c] freezer_mutex
 -> [80a274ac] kernfs_mutex
(...)
```

```
#cat /proc/lockdep_chains
all lock chains:
irq_context: 0
[80a1c55c] cgroup_mutex

irq_context: 0
[80a16a3c] resource_lock
(...)
```

```
#cat /proc/locks
1: FLOCK  ADVISORY  WRITE 2057 00:0e:7200 0 EOF
2: FLOCK  ADVISORY  WRITE 2246 00:0e:7097 0 EOF
```

lockdep_init() - (2)

```
#cat /proc/lock_stat
lock_stat version 0.4
-----------------------------------------------------------------------------------------------------------------------------------------------
                              class name    con-bounces    contentions   waittime-min   waittime-max waittime-total   waittime-avg    acq-bounces   acquisitions   holdtime-min   holdtime-max holdtime-total   holdtime-avg
-----------------------------------------------------------------------------------------------------------------------------------------------

 &mapping->i_mmap_rwsem-W:           1630           2186           0.99       64477.45      988876.13         452.37           37732         174127           2.19       66922.50     2247589.89          12.91
 &mapping->i_mmap_rwsem-R:              0              0           0.00           0.00           0.00           0.00              12            727          10.57        3269.64       36358.60          50.01
 -----------------------
   &mapping->i_mmap_rwsem           1017          [<801343b0>] unlink_file_vma+0x34/0x50
   &mapping->i_mmap_rwsem            319          [<8013448c>] vma_link+0x44/0xbc
   &mapping->i_mmap_rwsem            327          [<80024244>] copy_process.part.44+0x1440/0x17e4
   &mapping->i_mmap_rwsem            523          [<801347cc>] vma_adjust+0x2c8/0x604
 -----------------------
   &mapping->i_mmap_rwsem            274          [<8013448c>] vma_link+0x44/0xbc
   &mapping->i_mmap_rwsem            320          [<80024244>] copy_process.part.44+0x1440/0x17e4
   &mapping->i_mmap_rwsem            603          [<801347cc>] vma_adjust+0x2c8/0x604
   &mapping->i_mmap_rwsem            989          [<801343b0>] unlink_file_vma+0x34/0x50

.....................................................................................................................................................
```

```
#cat /proc/lockdep_stat
 lock-classes:                          1212 [max: 8191]
 direct dependencies:                   4230 [max: 32768]
 indirect dependencies:                13035
 all direct dependencies:              63105
 dependency chains:                     6166 [max: 65536]
 dependency chain hlocks:              18187 [max: 327680]
 in-hardirq chains:                       29
 in-softirq chains:                      261
 in-process chains:                     4580
 stack-trace entries:                  60133 [max: 524288]
 combined max dependencies:        36006660
 hardirq-safe locks:                      28
 hardirq-unsafe locks:                   471
 softirq-safe locks:                      91
 softirq-unsafe locks:                   415
 irq-safe locks:                          97
 irq-unsafe locks:                       471
 hardirq-read-safe locks:                 3
 hardirq-read-unsafe locks:              80
 softirq-read-safe locks:                 9
 softirq-read-unsafe locks:              77
 irq-read-safe locks:                    10
 irq-read-unsafe locks:                  80
 uncategorized locks:                   130
 unused locks:                            1
 max locking depth:                      15
 max bfs queue depth:                   159
 chain lookup misses:                  6175
 chain lookup hits:                10136672
 cyclic checks:                         4705
 find-mask forwards checks:             1593
 find-mask backwards checks:           31551
 hardirq on events:                  8784803
 hardirq off events:                 8784808
 redundant hardirq ons:               356471
 redundant hardirq offs:             6574108
 softirq on events:                   116537
 softirq off events:                  116565
 redundant softirq ons:                    0
 redundant softirq offs:                   0
 debug_locks:                              1
```

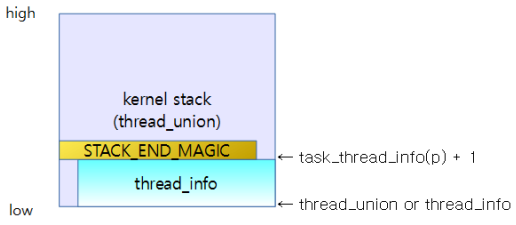
set_task_stack_end_magic()

set_task_stack_end_magic()

overflow 감지를 위해 스택의 마지막에 STACK_END_MAGIC 코드를 기록

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

```
#define task_thread_info(task)  ((struct thread_info *)(task)->stack)
```




```
struct thread_info {
    unsigned long       flags;              /* low level flags */
    int         preempt_count;              /* 0 => preemptable, <0 => bug */
    mm_segment_t        addr_limit;         /* address limit */
    struct task_struct  *task;              /* main task structure */
    struct exec_domain  *exec_domain;       /* execution domain */
    __u32           cpu;                    /* cpu */
    __u32           cpu_domain;             /* cpu domain */
    struct cpu_context_save cpu_context;    /* cpu context */
    __u32           syscall;                /* syscall number */
    __u8            used_cp[16];            /* thread used copro */
    unsigned long       tp_value[2];        /* TLS registers */
    union fp_state      fpstate __attribute__((aligned(8)));
    union vfp_state     vfpstate;
};
```

==========
33주차 (2015.12.05)
==========

http://www.iamroot.org/xe/FreeBoard/291113

==========
35주차 (2015.12.19) - start_kernel()
==========

1. cgroup

[Linux] cgroup - (task) control group (1)

Linux: 2.6.39-rc1

cgroup은 시스템 상에서 동작 중인 태스크들을 임의로 그룹지어 제어할 수 있도록 도와주는 기능이다.
cgroup은 구현된 subsystem의 종류에 따라 임의의 용도로 사용될 수 있지만
일반적으로는 시스템의 자원을 일정한 기준에 따라 분배하여 사용하도록 제어하는 용도로 사용된다.

cgroup을 이용하려면 커널 설정 시 CONFIG_CGROUPS 옵션을 선택해야 하며
이와 관련하여 여러 subsystem 중에서 필요한 것들을 추가적으로 선택하면 된다.

cgroup 자체로는 아무런 부가 기능 없이 순수하게 태스크들을 그룹지을 수 있는 기능 만을 제공하므로
실제로 cpu, memory, disk, device 등의 자원을 분배하려면 해당 기능을 구현하고 있는 subsystem이 필요하다.
이 글에서는 cgroup 자체에만 초점을 맞추어 살펴보도록 하겠다.

일단 사용자 측면에서 본다면 cgroup은 별도의 시스템 콜이나 장치 파일 등의 도움 없이
일반적인 파일 시스템 기능을 이용하여 각 그룹을 생성/제거/조작하도록 구현되어 있다.
커널은 cgroup 이라는 특별한 파일 시스템을 제공하는데 이를 마운트하면 cgroup을 바로 이용할 수 있고
해당 디렉터리 아래에 새로운 하위 디렉터리를 만들게 되면 새로운 그룹이 생성되는 방식이다.

예를 들어 다음과 같이 실행하여 cgroup을 이용할 수 있다.
(# 프롬프트는 이를 실행하기 위해 root 권한이 필요하다는 것을 의미한다!
sudo 명령을 이용해도 동작하지 않는다면 이 링크를 참조하기 바란다)

```
# mkdir -p /opt/cgroup
# mount -t cgroup nodev /opt/cgroup
```

```
# ls /opt/cgroup
cgroup.clone_children   cgroup.event_control   cgroup.proc
notify_on_release       release_agent          tasks
```

위에서 ls 명령을 수행했을 때 나타난 파일들은 커널 내의 cgroup이 자체적으로 제공하는 파일들이다.
만일 앞서 설명한 대로 커널 설정 시에 여러 cgroup subsystem들을 선택했다면
이보다 더 많은 파일들을 볼 수 있을 것이다.

위와 같이 실행한 경우라면 최상위의 cgroup 하나만 만들어진 상태이며
모든 태스크들은 자동으로 최상위 cgroup 내에 속하게 된다.
(그룹이 하나 밖에 없으니 그럴 수 밖에 없다.. ;;)

새로운 그룹을 생성하려면 단순히 하위 디렉터리를 생성하기만 하면 된다.

```
# mkdir /opt/cgroup/grp-a
# ls /opt/cgroup/grp-a
cgroup.clone_children   cgroup.event_control   cgroup.proc
notify_on_release       tasks
```

위에서 보듯이 새로 만들어진 디렉터리 (즉, cgroup)에는 상위 디렉터리와 마찬가지로
사용할 파일들을 커널이 자동으로 생성해준다. (단 release_agent 파일은 최상위 cgroup에만 존재한다)

이제 해야할 일은 태스크를 해당 그룹 (grp-a)에 포함시키게 하는 일이다.
이를 위해서는 각 태스크를 고유하게 식별할 수 있도록 태스크의 pid를 알아야 한다.
사실 더욱 정확하게 말하면 pid가 아니라 tid (thread id - POSIX의 pthread_t와는 다르다!)가 필요한데
그렇다는 것은 cgroup을 스레드 별로 다르게 지정하는 것이 가능하다는 의미이다.
(참고로 cgroup.proc 파일을 읽어보면 해당 그룹에 속한 프로세스들의 pid를 (중복을 제거한 형태로) 출력해준다)

태스크의 tid를 알아냈다면 이를 해당 cgroup 내의 task 파일에 쓰면 된다.

단순히 shell에서 다음과 같은 명령을 이용하여 이를 수행할 수 있다.
(태스크의 tid는 임의로 1234라고 가정하도록 하겠다)

```
# echo 1234 > /opt/cgroup/grp-a/tasks
```

이제 다시 tasks 파일을 읽어보면 우리가 입력한 tid가 보일 것이다.

```
# cat /opt/cgroup/grp-a/tasks
1234
```

주의할 점은 만일 여러 태스크를 동시에 cgroup에 넣고 싶은 경우라도
반드시 한 번에 하나의 tid 만을 써주어야 한다는 것이다.

새로 생성된 태스크는 자동적으로 부모 태스크가 속한 cgroup에 속하게 되므로
일반적으로 (테스트를 위해?) 사용하는 방법은 shell 프로세스를 특정 cgroup에 속하게 하는 방식이다.
현재 터미널에서 이용 중인 shell의 pid (이 경우 tid와 동일하다)는 $$ 변수를 이용하여 쉽게 알 수 있다.

```
# echo $$
1690
#
# echo $$ > /opt/cgroup/grp-a/tasks
# cat /opt/cgroup/grp-a/tasks
1234
1960
22349
```

위의 경우 1690이 shell 프로세스의 pid이고, 1234는 위에서 입력한 태스크 정보가 남아있는 것이고
22349의 경우 내용을 출력하기 위해 실행한 cat 프로세스의 pid이다.
cat 프로세스는 shell이 생성하여 실행한 것이므로 shell과 같은 cgroup에 속한다는 것을 볼 수 있다.

새로운 cgroup을 생성하는 것은 일반적인 디렉터리 구조와 마찬가지로 tree 구조를 이루게 되며

따라서 몇 번이고 중첩되도록 하위 디렉터리를 생성할 수 있다.

cgroup을 제거하는 것은 반대로 디렉터리를 제거하면 되는데
이 때 해당 cgroup에 속한 태스크가 없어야만 가능하다.
즉 cgroup 내의 모든 태스크가 종료되거나 다른 cgroup으로 옮겨져야 그룹을 없앨 수 있다.

이 때 notify_on_release 파일의 값이 1인 경우 (물론 우리가 직접 1로 써줘야 한다)
cgroup이 제거된다는 것을 release_agent로 지정된 프로그램에게 알려준다.

다음과 같이 간단히 release_agent로 사용할 프로그램을 작성해 보자.
(C 언어로 작성할 수도 있지만 여기선 간단히 shell script를 사용할 것이다.
만약 C 언어를 이용한다면 제거되는 cgroup의 경로가 argv[1]을 통해 넘어오게 된다.)

myagent.sh:

```
#/bin/sh

echo cgroup $1 released! > /tmp/cgroup-release-msg
```

주의할 점은 release_agent는 커널이 background에서 실행시키기 때문에
아무리 echo 혹은 printf 문을 이용하여 메시지를 출력해도 터미널에서 볼 수 없다는 것이다.
여기서는 특정 파일에 기록하도록 하였지만 좀 더 시각적인 효과를 원한다면
(GNOME의 경우) notify-send 등의 프로그램을 이용할 수 있을 것이다.

이제 이를 myagent.sh 파일에 저장한 뒤 실행 권한을 주고 release_agent로 등록한다.

```
# chmod +x /some/where/myagent.sh
# echo /some/where/myagent.sh > /opt/cgroup/release_agent
```

이제 새로운 그룹을 생성한 뒤 notify_on_release를 1로 기록하자.

```
# mkdir /opt/cgroup/mygrp
# echo 1 > /opt/cgroup/mygrp/notify_on_release
```

하지만 바로 그룹을 삭제한다고 release_agent가 호출되지는 않는다.
해당 그룹에서 최소한 하나 이상의 동작이 이루어져야지만 notify 기능이 활성화된다.
간단히 태스크를 그룹에 추가한 후에 (삭제를 위해서) 다시 제거하도록 하자.

```
# echo $$ > /opt/cgroup/mygrp/tasks
# echo $$ > /opt/cgroup/tasks
```

이제 mygrp라는 그룹을 삭제해보면 release_agent가 실행되었음을 간접적으로 알 수 있다.

```
# rmdir /opt/cgroup/mygrp
# cat /tmp/cgroup-release-msg
cgroup /mygrp released!
```

[Linux] cgroup - (task) control group (2)

앞서 cgroup에 대한 기본적인 사용법을 살펴보았으나 subsystem에 대한 설명이 빠져있었다.
말했듯이 cgroup 자체로는 태스크를 그룹지을 수 있는 방법 만을 제공하는 것이고
해당 그룹에 의미/역할을 부여하는 것은 각 subsystem이 담당한다.

이 글을 쓰고있는 현재 최신 버전인 2.6.39-rc1의 경우 다음과 같은 subsystem을 포함하고 있으며
실제 적용 여부는 커널 설정 시 적절한 옵션을 선택하여 활성화할 수 있다.

cpuset, debug, ns, cpu, cpuacct, memory, devices, freezer, net_cls, blkio, perf

현재 시스템 내에 실제로 포함된 subsystem의 목록은 /proc/cgroups 파일을 통해 볼 수 있다.

```
# cat /proc/cgroups
#subsys_name    hierarchy    num_cgroups    enabled
cpuset    0    1    1
debug    0    1    1
cpu    0    1    1
memory    0    1    1
freezer    0    1    1
...
```

cgroup 파일 시스템 마운트 시에 -o 옵션으로 각 subsystem의 이름을 쓰게 되면
해당 cgroup 파일 시스템을 통해 주어진 subsystem에 대한 제어를 할 수 있게 된다.
즉, 특정 cgroup과 subsystem이 연결되는 것인데
이 때 한 cgroup 파일 시스템에 임의의 여러 subsystem이 동시에 연결될 수 있다.

예를 들면 다음과 같다.

```
# mount -t cgroup -o cpuset,cpu nodev /opt/cpu-group

# mount -t cgroup -o memory nodev /opt/mem-group

# mount -t cgroup nodev /opt/oth-group
```

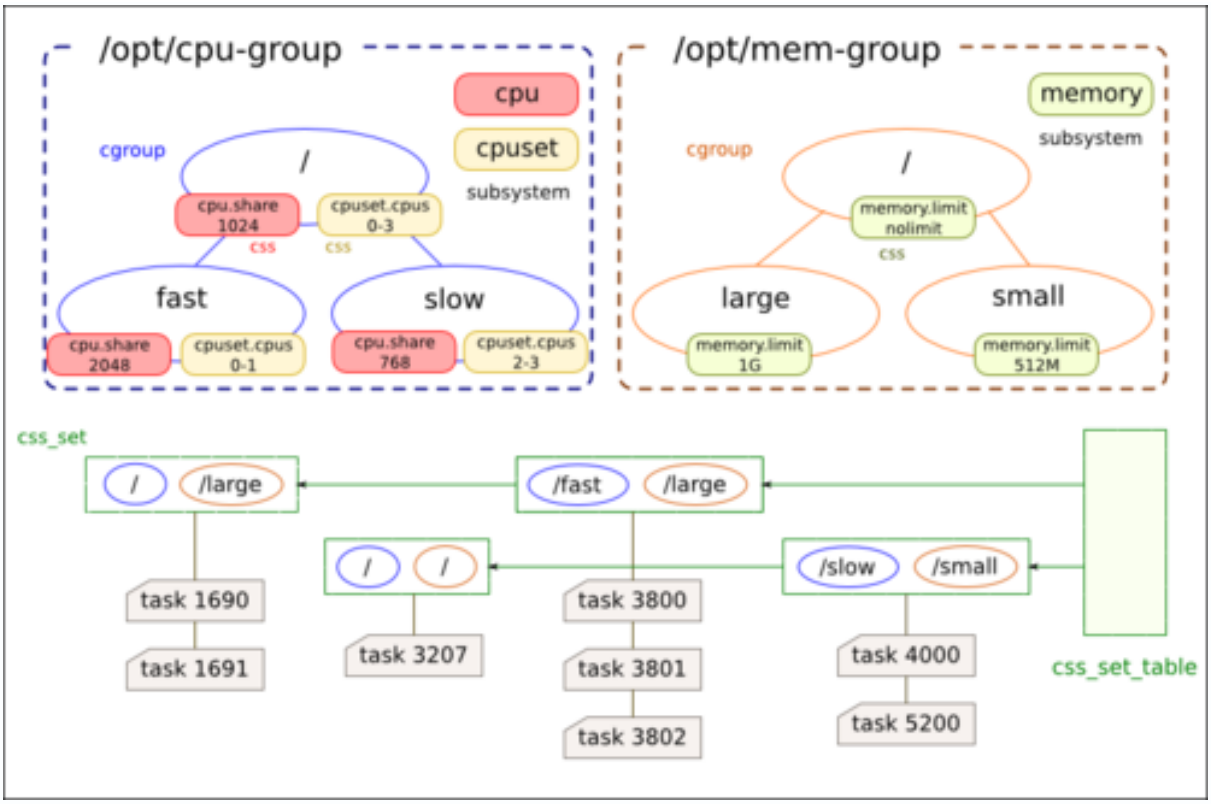
/opt/cpu-group 파일 시스템에는 cpuset과 cpu라는 2개의 subsystem이 연결되고
/opt/mem-group 파일 시스템에는 memory subsystem이 연결되고
/opt/oth-group 파일 시스템에는 나머지 모든 subsystem이 연결되게 된다.

이 때 주의할 점은 하나의 subsystem은 오직 하나의 cgroup 파일 시스템에만 연결될 수 있다는 점이다.
즉, 만약 cpuset이라는 subsystem을 /opt/cpu-group에 연결한 상태라면
다시 다른 cgroup을 마운트할 때 중복해서 cpuset을 지정할 수 없다는 의미이다.

subsystem이 cgroup 파일 시스템과 연결되고 나면 파일 시스템 내의 각 cgroup (즉, 디렉터리)마다
subsystem과 관련된 설정을 별도로 적용할 수 있게 된다.
커널은 이렇게 각 cgroup 마다 지정된 subsystem의 상태를 별도로 유지하기 위해
cgroup_subsys_state (css)라는 구조체를 사용한다.

하지만 위에서 보듯이 시스템 전체적으로는 하나 이상의 cgroup 파일 시스템이 존재할 수 있으므로
태스크 입장에서보면 하나의 태스크는 여러 cgroup에 동시에 속할 수가 있게 된다.
이러한 복잡한(?) 관계를 잘 추적하기 위해 태스크가 속한 모든 cgroup의 css를 쉽게 찾아낼 수 있도록
css_set이라는 구조체를 도입하여 해시 테이블을 통해 간단히 접근하도록 하였다.

이와 같은 구조를 그림으로 나타내면 대략 다음과 같다.
(귀차니즘으로 인해 2개의 cgroup 파일 시스템에 대해서만 나타내었다..;;)



위의 그림에서는 /opt/cpu-group 아래에 fast와 slow라는 디렉터리를 생성하였고
/opt/mem-group 아래에 large와 small이라는 디렉터리를 생성하였다.

쿼드코어 시스템을 가정하면 0부터 3까지의 총 4개 CPU가 존재할 것이고
cpuset subsystem을 통해 이를 각 그룹에 분배하였다. (cpuset.cpus)
또한 cpu.share 파일은 스케줄링 시 각 cgroup이 상대적으로 차지하는 비중을 나타낸 것으로
fast cgroup에 속한 태스크들은 root (/) cgroup에 비해 2배, slow group의 거의 3배에 가까운
cpu 시간을 할당받아 실행될 것이다.
메모리도 마찬가지로 루트 cgroup에 속한 태스크들은 아무런 제한이 없지만
large cgroup과 small cgroup에 속한 태스크들은 각각 총 1GB, 512MB의 메모리 만을 사용할 수 있게 된다.

위의 예에서 1690번 태스크와 1691번 태스크는 모든 CPU (0-3)에서 실행될 수 있으며
정상적인 (1024) 양의 cpu 시간을 할당받을 것이며 사용할 수 있는 메모리에는 제한이 없다 (nolimit).
하지만 4000번 태스크와 5200번 태스크의 경우 2번과 3번 CPU에서만 실행될 수 있으며
다른 태스크의 75% (768)에 해당하는 cpu 시간 만을 할당받고,
두 태스크의 메모리 사용량을 합산하여 512MB 이상을 사용할 수 없게 된다.

-> http://egloos.zum.com/studyfoss/v/5505982
-> http://studyfoss.egloos.com/5506102

2. IDR & IDA
   -> http://egloos.zum.com/studyfoss/v/5187192

3. RCU (Read Copy Update)
   -> 커널 락을 없애려는 시도로 도입된 동기화 기법?
   -> **동기화 기법: Read-Copy-Update**

컴퓨터 과학 2010.12.18 02:55

2.6 시대에 리눅스 커널을 뜯어본 사람이라면 RCU라는 것을 알지도 모르겠다. RCU란 리눅스 커널 내에서 주로 읽기 연산만 일어나고 쓰기 연산의 비중은 매우 작은 객체에 주로 쓰이는 동기화 기법이다. Reader-Writer lock과 비슷한 동기화 기법인데, RW lock에 대해 RCU가 가지는 상대적 강점으로는 읽기 연산이 Wait-free이며 (다시 말해 블럭이 일어나지 않으며) 그 오버헤드가 극도로 작다는 점 등이 있다. (리눅스에서는 kernel preemption을 끄고 컴파일하면 RCU로 보호되는 메모리를 읽을 때 동기화 오버헤드가 제로다!) 그 대신 쓰기 연산의 오버헤드가 꽤 큰데다 무슨 일을 하건 쓰기에 필요한 동기화의 시간 복잡도가 최소한 RCU로 보호되는 객체의 크기에 비례한다.

RCU의 아이디어는 꽤 심플하다. 그 기저에 깔린 전제들만 알면 쉽게 이해할 수 있는데, 내 생각엔 그 목록은 대충 아래와 같다.

- 불변 *Immutable* 객체는 쓰레드 안전하다. 좀 더 구체적으로, 어떤 코드 영역을 수행하는 동안 공유되는 메모리 구간이 불변이라면 해당 코드로 인한 상태의 변화는 항상 예측가능하다. 다시 말해 코드 수행의 결과가 결정적 *Deterministic* 이다.
- 메모리 배리어 등을 통해 가시성이 확보한 경우 단일 워드에 대한 쓰기 연산은 대개 원자적이다. 적어도 다른 쓰레드 입장에선 그렇게 보인다.
- 어떤 객체의 상태를 변경하는 코드를 수행할 때, 그 변경이 다른 쓰레드의 입장에서 한 순간에 원자적으로 이루어지는 것처럼 보인다면 연관된 코드들의 수행 내역은 단일 쓰레드로도 재현 가능하다. 쓰레드 간 코드 수행 순서에 대한 구체적인 합의가 필요한 경우는 드믈기 때문에 대부분은 이 정도만 보장되어도 동기화로써 충분한 역할을 할 수 있다.

이러한 전제들을 사용하면 흥미로운 결과를 도출해 낼 수 있다. 우선 가비지 컬렉션을 지원하는 요즘 언어들에서는 객체를 직접 변경하는 대신 불변 객체를 새로 만들어서 쓰는 것은 드믈지 않은 일이다. 이를테면 문자열 객체를 불변으로 다루는 Java 같은 언어에서는 문자열을 변경하려 하는 경우 전역적으로 관리되는 문자열 객체 풀에서 변경된 결과의 불변 문자열을 가져와서/새로 만들어서 사용한다. (이를 보통 String Internalization이라 한다.) 굳이 객체 풀을 만들 필요까지도 없고 이미 노출된 객체를 변경하는 대신 새로운 불변 객체를 만드는 방법을 이용하면 쓰레드 안전성을 쉽게 확보할 수 있다.

그렇다면 이를 안전하게 노출시키기만 하면 된다. 세 번째 전제에 따르면 객체의 갱신 과정이 원자적으로 보이기만 하면 실용적인 동기화 방식으로 활용할 수 있는데, 두 번째 전제에 따라 레퍼런스

역할을 하는 포인터의 값으로 새로운 객체의 주소를 집어 넣기만 하면 된다. 다만 그냥 대입하면 새로운 객체가 완성되기 전에 CPU나 컴파일러의 비순차 실행 최적화에 의해 포인터의 대입 연산이 먼저 수행될 수 있다. 이 경우 객체의 변경이 완료되기 전에쓰레드 외부로노출될 우려가 있으므로 메모리 배리어로 포인터의 대입이 객체의 생성 이후에 이루어지도록 보장할 필요가 있다.

여기까지 설명한 것만으로도 이미 RCU의 기본적인 동작 방식을 절반 이상 설명했다. RCU로 보호되는 객체에 대한 읽기를 하려는 경우 평소처럼 포인터를 가져와서 읽기 전용으로 해당 객체를 다루면 된다. RCU로 보호되는 객체를 변경하려는 경우 객체를 직접 변경하는 대신 새로 할당한 메모리 영역에 객체를 복사한 뒤 새로 만든 객체를 변경한다. 기존의 객체는 불변이므로 이 과정은 안전하다. 변경이 완료되면 RCU로 보호되는 포인터에 새로운 객체의 포인터를 대입한다. 이러면 대입 이전에 이 객체를 읽어서 사용하던 쓰레드는 이전의 객체를 계속 사용하게 될 것이고, 대입 이후에 사용하는 쓰레드는 새로운 버젼의 객체를 읽어 사용하게 될 것이다. 각각의 쓰레드에서 여러 버젼의 객체가 동시에 사용된다는 점에서 데이터베이스에서 사용되는 동시성 관리 기법인 MVCC[1]와도 비슷하다. 아래는 RCU를 사용하는 방식을 보여주는 가짜 코드이다.

```
SomeObject object;

 // RCU로 보호되는 객체
RCU<SomeObject> rcu(object);

...

// reader thread
rcu.beginRead();

// Wait-free로 구현됨.
SomeObject* ptr = rcu.getPointer();
ReadOnlyOperation(ptr);
rcu.endRead();

...

// writer thread
rcu.beginWrite();

// 일반적인 락과 유사. 어차피 객체의 갱신은 직렬적으로 처리되어야 한다.
SomeObject* ptr = rcu.copyObject();
```

```
// 객체를 복사함.
WriteOperation(ptr);
rcu.endWrite();
rcu.setPointer(ptr);
```

이 쯤에서 자원 관리가 어떻게 될까 의문을 가질 것이다. 사실 RCU 구현의 핵심도 바로 이 부분이다. 자원 관리에 신경을 쓰지 않아도 되는 언어라면 굳이 이런 개념을 도입할 필요도 없었을 것이다. 하지만 수동으로 자원을 관리해줘야 하는 언어를 사용하는 경우라면 RCU 차원에서 더 이상 사용되지 않는 객체를 적절하게 해제할 수 있도록 도와줘야 할 필요가 있다. reader 입장에서 더 이상 접근할 방도가 없는 객체인 경우만 해제해주면 되므로 위의 가짜 코드에서 setPointer를 한 이후 예전 객체를 읽는 reader들의 코드 수행이 전부 종료되는 시점에 해제를 해주면 된다. 물론 이를 구현하는 방법은 여러 가지가 있겠지만, 가장 직관적인 방법으로는 레퍼런스 카운팅이 떠오른다.

이 정도면 누구라도 쉽고 재미있게 구현할 수 있을 것 같다. ... 정말 그럴까?

아쉽게도 그렇지 않다. -_- 만약 레퍼런스 카운팅을 한다 치면 읽기 영역에 진입/이탈 시 원자적인 정수 연산이 적어도 한 번씩은 일어날 것이다. 그런데 일반적인 RW lock도 경쟁이 없다는 전제 하에선 읽기에 대해 원자적인 연산 두어개의 오버 헤드만을 가지도록 구현하는 것이 가능... **아마** 가능할 것이다 -_-; 즉 RW lock과 차별화되는 강점 하나가 사라진다. 물론 읽기가 Non-blocking이라는 매력적인 조건이 있지만 이 것만 가지고 쓰기 시의 추가적인 오버헤드를 용납하기는 어렵다. RCU 자체가 읽기에 수반되는 동기화의 Critical path를 최소화하려는 시도이므로 읽기의 오버헤드는 최소화되어야 한다.

리눅스 커널에서는 이걸 아주 간단하게 해결하고 있다. RCU는 커널 코드 내에서만 쓰이며, 커널 모드에서의 수행이 끝나면 RCU의 읽기 구간 수행 역시 끝날 것이다. 따라서 위 가짜 코드의 setPointer를 수행하는 시점에 돌아가고 있는 모든 CPU들에 대해 해당 CPU로의 문맥 전환을 한번씩 요청한다. 커널 모드에서 문맥 전환이 일어나지 않는 비선점형 커널을 사용하고 있는 경우 이 과정이 끝나면 해당 시점에 커널 모드에서 돌아가던 모든 동작들이 종료된 것이므로 (= 기존 객체에 대한 읽기가 끝난 것이므로) 메모리를 안전하게 해제할 수 있다. 선점형 커널인 경우는 읽기 구간에 들어간 동안에만 커널 선점을 꺼버리는 방법을 사용하고 있다고 한다. (요즘의 구현은 다를지도 모르겠다. 논문들을 보면 Hazard pointer 등을 이용한 구현도 있는 모양이다.)

헌데 유저 모드에서 적은 오버헤드로 RCU를 구현하려고 생각해보면 암담하다. -_-; OS의 지원을 받지 못하니 선점 모드를 끄는 등의 작업은 명백하게 월권이다. 특정 쓰레드가 읽기 구간에 있다는 것을

알리는 변수를 세팅한다고 해도 해제하는 쓰레드가 이에 대한 가시성을 얻고 코드 수행의 순서 일관성을 보장하기 위해서는 메모리 배리어가 필요하다. 허나 메모리 배리어의 사용 역시 만만치 않은 오버헤드를 가지고 있기 때문에 이를 사용하려 들면 문제는 원점으로 돌아가 버린다. 결론은 **답이 없다** 되겠다. -_-;

그런데 사실 오버헤드가 좀 있다손 치더라도 읽기의 동기화 과정이 Wait-free라는 것만으로도 RCU는 상당히 매력적인 모델이다. 이 말은 곧 읽기에 대해서는 최대한의 병행적인 확장성을 제공한다는 이야기이기 때문이다. 또한 읽기를 위한 동기화는 넌블럭이고 쓰기 역시 하나의 락만 사용하므로 데드락의 가능성이 거의 없다는 것 역시 강점이다. 또한 쓰기에 걸리는 락 역시 쓰기 동기화를 위해서는 당연히 필요한 오버헤드이기 때문에 결국 RCU로 인해 생기는 오버헤드는 객체 복사 및 추가적인 할당/해제 정도가 전부이다.

개인적으로는 이렇게 시점 별로 여러 버젼의 불변객체를 유지하는 MVCC 류의 동시성 제어 기법은 가비지 컬렉터가 지원되는 언어에서 더욱 빛을 발할 것이라 생각한다. 자원 해제 관련된 부분이 완전히 투명해지므로 (심지어는 성능적인 부분에 있어서도) 사용하기가 훨씬 간편해지며, 메모리 압축을 지원하는 언어에서는 할당 오버 헤드 역시 매우 작으니 큰 문제가 되지 않는다. 복사로 인한 오버헤드는 읽기에서 락을 사용하지 않음으로 얻는 퍼포먼스 이득 및 데드락으로부터의 해방을 생각해보면 충분히 감수할만한 가치가 있다.

물론 이 방식이 은탄환인 건 절대 아니다. 일단 원하는 데이터가 연속된 메모리 공간에 위치하지 않은 경우에는 이 기법을 적용할 수 없다. 즉, 이 기법만을 사용해서는 임의의 메모리 공간들에 대해 원자적인 갱신을 할 수 없다.졸면서 쓴 부분을 수정한다. -_-; 정확히는 다루는 객체가 **의미적으로 불변**인 객체여야 한다는 제약이다. 이러면 포인터를 바꿔칠 수 없으므로 RCU로 동기화되는 객체들을 조합해서 마찬가지로 RCU로 동기화되는 더 큰 객체를 만드는 등의 작업은 불가능하고, 그냥 불변 객체를 생으로 만들고 복사하는 방식으로 써야 한다. 다르게 말하자면 객체 간의 합성이 까다로운 동기화 기법인 셈이다. 방법이 있는데 내가 모르는 것일지도 모르겠지만... 하지만 이 방법은 높은 병행성을 가지는 자료구조나 알고리즘을 구현하기 위한 빌딩 블럭으로써 꽤 괜찮은 기법이다. 실제로 리눅스에서는 RCU를 이용해서 리스트를 관리하기도 한다. 개인적으로는 RCU를 잘 정제해서 동시성을 제어하는 기본 요소로 언어에 도입해보는 것도 좋은 시도가 아닐까 생각한다. (이미 있을지도 모르겠다)

http://summerlight.tistory.com/entry/%EB%8F%99%EA%B8%B0%ED%99%94-%EA%B8%B0%EB%B2%95-Read-Copy-Update

4. kprobe

(끝)

asm 코드 내에 volatile 사용 이유
https://gcc.gnu.org/ml/gcc/1997-11/msg00413.html

코드의 이동을 막기 위함...

cpu active 상태와 cpu online 상태의 구분
http://vh21.github.io/linux/2015/04/28/linux-cpu-mask.html

==========
44주차 (2016.02.20)
==========

gcc 확장기능 이용하기 - __builtin_return_address
http://forum.falinux.com/zbxe/index.php?document_srl=550242&mid=lecture_tip

==========
45주차 (2016.02.27)
==========
#define에 do { } while(0)를 사용하는 이유
http://kernelnewbies.org/FAQ/DoWhile0