< CFS scheduler & Load Balancing on SMP 분석>

학과: 컴퓨터학과
학번: 2013210056
이름: 남관우
제출일자: 2017.05.04 (목)

[1] CFS scheduler 분석

A. CFS 는 어떠한 정책을 가지고 스케줄링 하는가?

우선 Linux Scheduler 는 5 가지 policy 를 가지고 CFS 는 이 중 3 가지 policy 를 반영한다. 스케줄러의 역할은 특정 CPU 코어에서 스레드를 어느정도 실행할 지를 결정하므로 스케줄러를 분석하기에 앞서 정책을 분석해야한다.

1) SCHED_NORMAL (or SCHED_OTHER)

기본 적인 스케줄링 정책이다. 대다수 process 들이 사용하는 정책으로, time-sharing 스케줄링을 사용하는 것이 특징이다. CFS 를 사용하여 모든 Task 들에게 fair 한 CPU 점유 기간을 제공한다. 각 process 의 thread 의 nice 값에 따라 동적으로 우선순위를 결정한다

2) SCHED_BATCH

Task 중에서 CPU-bound 를 요구하는 task 들에게 사용되는 정책이다. 상호 보완적이지 않고 CPU bound 인 process 들을 스케줄링 한다. 그러므로 cache-affinity 가 높지만 우선순위가 낮은 process 들의 정책이다.

3) SCHED_IDLE

Task 중에서 낮은 점유율을 가지는 task 들에 대한 정책이다. CPU 가 idle 상태이고 다음 task 로 정해진 것이 없을 때, idle task 를 CPU 에 할당하는 것이다.

B. 우선순위를 결정하는 기준은 무엇이고 어떻게 우선순위를 Update 하는가?

a) Nice 란 무엇인가?

➔ 어느 Task가 cpu에 선점할 수 있는 우선순위권을 표현한 value

[특징]

1) Priority를 표현하기 위한 것
2) Linux에서 process가 어떻게 scheduling 되는지
3) -20 ~ 19 사이의 value를 가진다
(-20 ← High priority || Least priority → +19)
4) default value 0
5) **nice value는 weight에 Mapping 된다**
6) nice value가 Linear하게 증가하면
weight는 exponent하게 증가함
7) nice value는 2가지 방법으로 변경 가능
Command Line: $nice –n and $renice –p
System call: int nice(int inc);

[Command line에서 process 당 nice 값]

```
ubuntu@ip-172-31-27-81:~$ ps ax -o pid,ni,cmd
  PID  NI CMD
    1   0 /sbin/init
    2   0 [kthreadd]
    3   0 [ksoftirqd/0]
    5 -20 [kworker/0:0H]
    6   0 [kworker/u30:0]
    7   0 [rcu_sched]
    8   0 [rcu_bh]
    9   - [migration/0]
   10   - [watchdog/0]
   11   0 [kdevtmpfs]
   12 -20 [netns]
   13 -20 [perf]
   14   0 [xenwatch]
   15   0 [xenbus]
   17   0 [khungtaskd]
   18 -20 [writeback]
```

➔ 총 40개의 nice value mapping weight

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,     /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,     /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,     /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,     /*  15 */        36,        29,        23,        18,        15,
};
```

prio_to_weight[nice] = 1024 / (1.25^nice)

➔ NICE value 와 mapping 되는 weight 에 대해 CPU 할당 시간의 변화 양상

Running Queue 에서 실행되고 있는 task 는 A 와 B 만 있다고 가정

Task A ← (nice, weight) = (1, 820) Task B ← (nice, weight) = (0, 1024)

전체 weight = 1844 (820 + 1024)

Task A 의 CPU 할당 = ~45% Task B 의 CPU 할당 = ~55%

<nice value 1 차이가 weight 에 mapping 되면 25%의 차이를 내고, CPU 선점은 10%의 차이 난다>

b) vruntime 은 무엇인가?

➔ CFS 의 스케줄링 기준이며 지표이고, process 의 virtual runtime 을 저장하는 정규화된 값이다

```
struct sched_entity {
  struct load_weight  load;   /* for
  struct rb_node      run_node;
  struct list_head    group_node;
  unsigned int        on_rq;

  u64     exec_start;
  u64     sum_exec_runtime;
  u64     vruntime;
  u64     prev_sum_exec_runtime;

  u64     nr_migrations;
```

1) include/linux/sched.h File 의 **struct sched_entity** 구조체 member 변수

2) ns 단위 이고 Timer tick 과 분리

3) 주어진 process 와 runtime 과 다음에 어떤 process 가 실행되어야 하는지에 대한 정확한 지표

4) 과거의 고정된 timeslice 를 사용하는 것이 아닌 유동적 CPU 사용을 위해 도입된 개념

```
curr->vruntime += calc_delta_fair(delta_exec, curr);
update_min_vruntime(cfs_rq);
```

5) sched_fair.c File 의 static void **update_curr** (struct cfs_rq *cfs_rq) 함수 내에서 update 실행

6) *vruntime += pTime * (N0 / taskW)* (NORMARLIZED)

[pTime = Task 가 실제 수행된 물리적인 Time slice]

[taskW = 현재 Task 의 weight] [N0 = NICE_0_LOAD 로서 nice 0 값 ➔ 1024 default]

7) Priority 가 높은 Task 의 vruntime 은 priority 가 낮은 vruntime 보다 느리가 증가

8) 빈번한 scheduling ➔ Context Switching ➔ Overhead growth

= sched_latency 와 sched_min_granularity 개념을 도입하여 빈번한 스케줄링 방지

➔ RB-Tree에 Task들은 vruntime을 key 값으로 저장되고 스케줄링 시 Left most node가 선택

Real Physical Time Slice ➔ 시간이 증가하면	vruntime HIGH ➔ **Right Node** in RB Tree ➔ Scheduling 우선권이 낮아진다
Task Weight ➔ Weight 가 증가하면	vruntime LOW ➔ **Left Node** in RB Tree ➔ Scheduling 우선권이 높아진다

(1) 수행시간이 많을 경우 = task가 terminate까지 얼마 남지 않음

(2) Weight ➔ Priority 가 높을 경우 = task가 running queue에서 우선권을 가진다는 의미

Task priority는 running queue에 있는 다른 task들과 상대적이며, 할당 받는 time slice는 우선순위에 따라 다르다. 먼저 예시로 weight가 running queue에서 상대적으로 높은 Task의 경우를 분석해보자. vruntime += (수행시간) * inversed_Weight 이므로 weight가 커서 vruntime이 작아 RB-Tree의 왼쪽 편에 위치한다. 그래서 스케줄링 시 left_most_Node를 select하므로 CPU를 자주 할당받는다. {BUT} vruntime += (수행시간) * inversed_Weight 에 의해 vruntime이 커져 RB-Tree의 오른편으로 위치하게 되고, 우선순위가 낮은 task들도 공평하게 CPU를 수행할 수 있도록 된다. (증가하는 속도에 가중치 부여)

즉, 하나의 running queue에서 하나의 실행 period를 진행할 때, weight가 높은 task는 time slice를 높게 받지만 weight가 크게 되므로 절대적으로 높지 않은 수치를 가지게 되고, weight가 높은 task는 물리적인 time slice를 낮게 받지만 weight가 작으므로 역수 처리 된 term때문에 vruntime이 절대적으로 낮지 않은 수치를 가지게 된다. 따라서 Running Queue에서 모든 Task들이 공평하게 CPU를 할당 받는다.

c) vruntime은 어떻게 업데이트 되는가?

➔ vruntime 의 Update는 해당 Task의 실제 수행 시간 하고 자신의 weight의 역수를 비례하게 계산한다. 실제 수행된 시간이 많을 수록, weight 가 낮을 수록 vruntime은 커지게 되므로 RB-Tree에서 오른쪽 Node로 밀려나고, 실제 수행된 시간이 적을 수록, weight 가 클 수록 왼쪽 Node로 update 된다.

➔ vruntime 을 조정하기 위한 함수들의 연결고리

```
update_curr (struct cfs_rq *cfs_rq) {
        scehd_entity *curr = cfs_rq → curr;
        u64 now = rq_clock_Task(rq_of(cfs_rq));
        ...
        delta_exec = now - curr → exec_start;
        curr → vruntime += calc_delta_fair(delta_exec, curr);
}

        calc_delta_fair (delta, *se) {
                delta = __calc_delta_mine(delta, NICE0, se → load);
        }

                __calc_delta_mine(delta, NICE_0,  ) {
                        delta = delta_exec * NICE_0 / (curr→load.weight);
                }
```

➔ 각 Function 들의 특징

(1) (sched_tick → reschedule → schedule 함수에서 진행) update_curr ()은 시스템 타이머에 의해 주기적으로 호출되며 프로세스가 실행 가능하게 되거나 블록화되어 실행 취소 할 수 없을 때마다 호출

(2) calc_delta_fair () 함수 __update_curr ()은 calc_delta_fair ()를 호출

calc_delta_fair ()는 가중치를 계산하기 위해 calc_delta_mine ()을 호출

(se-> load.weight 가 NICE_0_LOAD 와 같지 않은 경우).

➔ Process의 vruntime과 Timeslice 공식

CFS_Timeslice ➔ sched_slice = __sched_period * (now task weight / total task weight)

VRUNTIME ➔ vruntime += delta_exec * NICE_0 (curr → load.weight)

➔ vruntime update function call chain 정리

scheduler_tick ()	task_tick () 훅을 CFS 에 호출, **task_tick_fair() 호출** curr→scehd_class→task_tick(rq, curr)
task_tick_fair ()	작업 스케줄링 엔티티와 해당 실행 큐에 대해 **entity_tick ()을 호출**

entity_tick ()	첫째, 현재 예약 된 작업의 런타임 통계를 업데이트 두 번째로 현재 작업을 선점해야 하는지 확인 **update_curr () 호출**
update_curr ()	작업의 런타임 통계를 업데이트하는 책임있는 함수 현재 작업이 마지막으로 예약 된 이후의 경과 시간을 계산하고 delta_exec 결과를**__update_curr ()에 전달**
__update_curr ()	min_vruntime 을 업데이트 정기 업데이트 외에도 update_current ()는 해당 작업이 실행 가능하게 되거나 절전 모드로 전환 될 때 호출
entity_tick () 으로 return.	작업이 업데이트 된 후 **check_preempt_tick () 호출** 프로세스의 vruntime 이 red-black-tree 의 가장 왼쪽 작업의 vruntime 과 비교하여 **프로세스 전환이 필요한지 결정**

C. 스케줄러 (do_schedule())가 호출되는 3 가지 시점은 언제인가?

1) Regular runtime update of currently scheduled task

Timer interrupt → scheduler_tick() 호출

(=runqueue 의 clock 을 update, SMP config 되면 load balancing 도 진행)

→ scheduler_tick() → task_tick() 호출 (task_tick_fair())

→ entity_tick() 호출

1) update_curr() 함수 호출

2) check_preempt_tick() → resched_task

2) Running Task 가 sleep 상태로 변화될 때

Task 가 sleep 하기 위해선 특정 event 가 있을 때까지 wait 해야한다. 함수 내에서, condition 이 만족할 때 까지 prepare_wait 하고 signal handling 에서 TASK_Interruptable, TASK_uninterruptable flag 그리고 condition 을 만족하지 못하면 loop 에서 나와 wait() 종료 → Task 는 wait queue 에서 제거 됨

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define __TASK_STOPPED          4
```

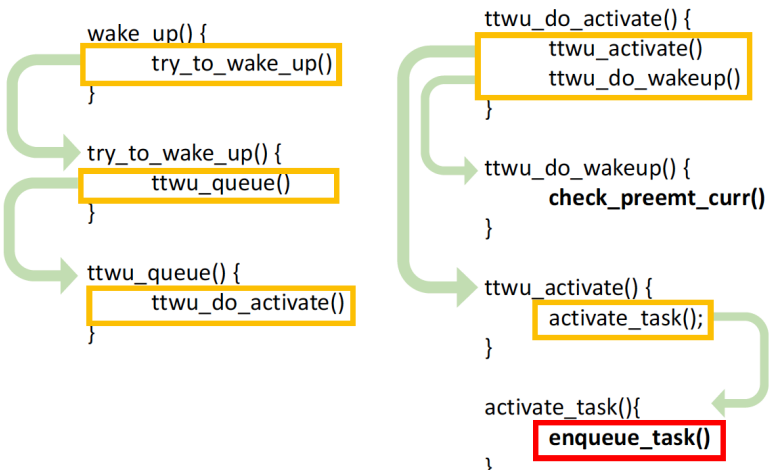
TASK_INTERRUPTIBLE → signal 받으면 깨어날 조건이 아니어도 깨어날 수 있음

TASK_UNINTERRUPTIBLE → signal 무시

1) DEFINE_WAIT macro 를 이용해서 wait queue 에 추가할 항목을 생성
2) add_wait_queue 함수를 이용해 작업을 waiting queue 에 추가
3) prepare_wo_wait 으로 process 상태를 TASK_(UN)INTERRUPTIBLE 로 변경
4) TASK_INTERRUPTIBLE 이면 signal 에 의해 깨어날 수 있음 (위의 조건)
5) 깨어난 작업의 조건 만족 확인 → 만족 시 loop exit, 아니면 **schedule() 호출**
6) TASK_RUNNING 으로 finish_wait() 함수를 호출

3) Sleeping Task 가 wakes up 할 때

Task 가 wake up 되면 balance decision 도 결정해야 함



ttwu_do_wakeup()

= 실행 task 의 선점 필요 여부 확인

check_preemt_curr() → need_resched flag → TASK_RUNNING flag 있으면 wake up process 는 종료

enqueue_task()

= schedule() 호출

D. Task 가 wake up 되었을 때, 스케줄러는 어떠한 동작을 수행하는가?

1) sleep or blocked 상태

sleep 중 혹은 blocked 상태의 작업은 실행 불가능하다.

스스로 자신을 waiting 상태임을 표시하고, wait queue에 들어가고 runnable task 의 RB-tree에서 자신을 제거하고 schedule을 호출해 새 process 를 선택하여 실행

2) wake up 과정 (스케줄러 작동 과정)

→ 과정

wake_up 함수로 처리 ("C-3" 의 function call chain 활용)

주어진 wait queue 내 모든 작업을 깨움

try_to_wake_up 함수를 호출해 TASK_RUNNING 상태 변경 → enqueue_task()를 호출하여 RB-tree에 insert

→ Function call chain

wake_up() → try_to_wake_up() → ttwu_queue() → ttwu_do_activate() → ttwu_activate() & ttwu_do_wake_up() → check_preempt_curr() → activate_task() → enqueue_task() → schedule() → try_to_wake_up()

(1) Task를 wake up 시키고 running queue에 복제

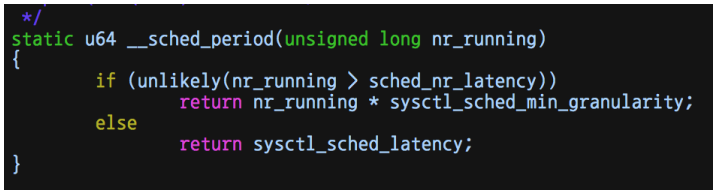
(2) state 를 TASK_RUNNING으로 하여 task woken up

(3) 새로 깨어난 Task가 현재 실행되는 Task의 priority가 높다면, 새로 wake up된 task에 need_resched flag가 붙어지고 schedule()함수가 실행된다

E. vruntime 의 증가속도를 조절하는 것 이외에, 특정 프로세스를 빠르게 스케줄링 할 수 있는 아이디어

(문제 해석) 1. 특정 process의 CPU 선점 시간을 줄인다는 의미
2. 우선순위가 낮더라도 공평성 원칙에 의해 스케줄링이 빨리 된다는 의미

1) Scheduling Latency 구할 때 constraints



Running Queue에서 스케줄링을 위해 작업을 시작하기 전 모든 process의 bound를 검사한다. Process에 새로운 PRC_BOUND_FLAG를 도입하여CPU bound process일 때 0이고I/O bound process이면 1인 조건을 도입한다. 그러면 running queue에 있는 process들의 PRC_BOUND_FLAG를 모두 곱하여 0이 되면 기존 방식을 고수하고, 1이 되면 모든 process가 I/O bound 임을 알 수 있다.

스케줄링 기간을 산출할 때 ~~*if(nr→running <= sched_nr_latency) then*~~ **periods = sched_latency**, ~~*else periods = sched_min_granularity * nr_running*~~ 으로 조건을 변형한다. 단순하게 periods를 sched_latency로 하여 최소한의 실행 보장 시간 개념을 제외시키는 것이다. I/O bound process는 wait 시간이 빈번하므로 cpu의 선점시간이 높지 않아도 되기 때문이다.

이렇게 periods 산출 조건을 변형시켜 vruntime의 증가속도를 조절하지 않고 프로세스를 빠르게 스케줄링 할 수 있다.

2) nice 값의 조정

nice value는 User space에서 system call인int nice(int inc)을 호출하여 재조정 하는 방법과 command line에서 renice –p 로 수정 하는 것이 가능하다. 프로세스의 select는 physical execution time에 반비례하고 weight (before nice mapped)에 비례하기 때문에 nice값의 재조정으로 프로세스를 빠르게 스케줄링 할 수 있다.

3) Multicore의 수를 증가 (물리적인 방법)

Physical core의 수가 증가하면 1 core의 running queue의 task수가 감소 하므로 동일 period 조건에서 스케줄링 시기가 조금 더 빨리 올 수 있다.

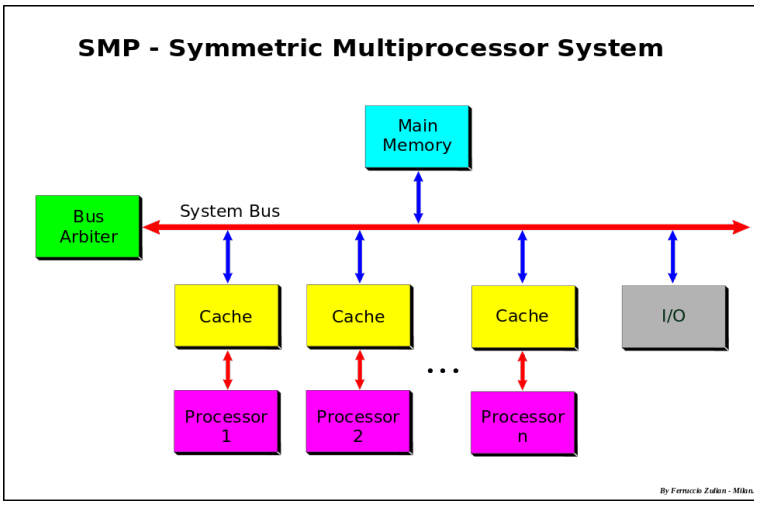
4) 스케줄링 되는 3가지 시점에 특정 process 를 등록함

‘질문 C’에서 기술한 3가지 시점은 timer interrupt, sleep, 그리고 wake 가 되는 상황을 제시하였다. 특히 CPU bound process이고 weight가 높아도 vruntime에 의해 공평성 이 보장 되기 때문에 스케줄링 시기가 빠르지 않을 수 있다. 정말 중요한 process라 하여도 nice값이 최대 40 차이를 보장하므로 1.25^20과 1.25^-19 보다 큰 차이를 필요에 의해 지정 해줄 수도 있다. 이를 위해 생각한 방법은 do_schedule() 되는 시점에 struct cfs_VIP *cfs_vip 포인터를 지정하여 스케줄 시 very important process를 검색 하고 반영 할 수 있게 조정하면 될 것이다.

[2] Load Balancing on SMP system 요약

[wikipedia]

a. SMP System Topology



(1) Multiple processor 에서 하나의 공유된 memory 사용한다

(2) Task 를 위한 Data 가 Memory 의 어느 위치에 저장 되어 있는지 상관없이 processor 가 진행 가능하다

(3) Kernel 의 지원이 있으면, SMP system 은 Load balancing 이 가능하다

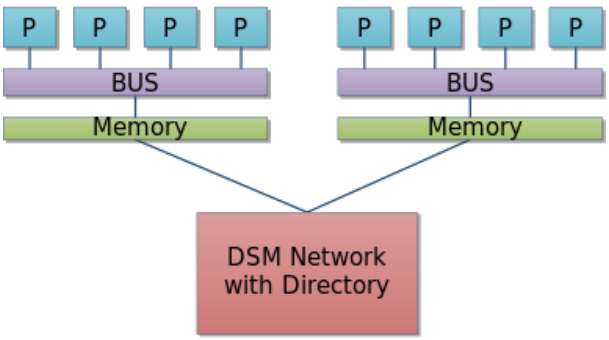
(4) 그러나, 오른쪽 그림에서 보듯이 여러 processor 는 하나의 main memory 를 share 하므로 system bus 를 통해 load/store architecture 를 진행한다

(5) SMP system 은 (4)의 원인 때문에 (memory <-> processor) 병목현상 발생 (단점)

(6) memory access 는 하나의 processor 만 허용(serialized access) 시키기 때문에 대기시간 발생

[wikipedia]

b. NUMA System



(1) SMP system에서 memory access 지연 시간을 해결하기 위해 나온 system

(2) Domain 마다 memory space 를 할당

(3) Local access의 speed는 빠르지만 Remote access의 speed는 느린 제약 조건이 있다

c. 분석

1) 목적: SMP system 의 성능을 향상 시키기 위해 Load Balancing 을 도입

2) 방법: Busy CPU 의 Task 들을 less busy or Idle CPU 에 offloading 시킴

3) 주체: Scheduler 가 System 에 task load 가 얼마나 진행되는지 확인 후 필요하다고 판단 시 load balancing 을 진행

4) Domain: SMP system, Hyper Threading 을 지원하는 system, 그리고 NUMA system 처럼 다양한 system topology 들이 존재하기 때문에 Linux 2.6 부터 Scheduling Domain 개념을 도입함
Hierarchy 가 Physical CPU → Logical Core (by hyperthread) 로 구성됨
Scheduling Domain 은 set of CPU 인데 이는 특징과 balancing 정책 등을 공유한다.

d. Sched_domain

(정책) (1) 전체 domain 에서 Load balancing 을 사용한 횟수
(2) Balance 하기 전에, 프로세서의 Load 가 얼마나 멀리 동기화 되어있는지 확인
(3) Process 가 cache 와 친화력을 얼마나 가지고 있는지 고려하기 전에, Idle 상태 기간 고려

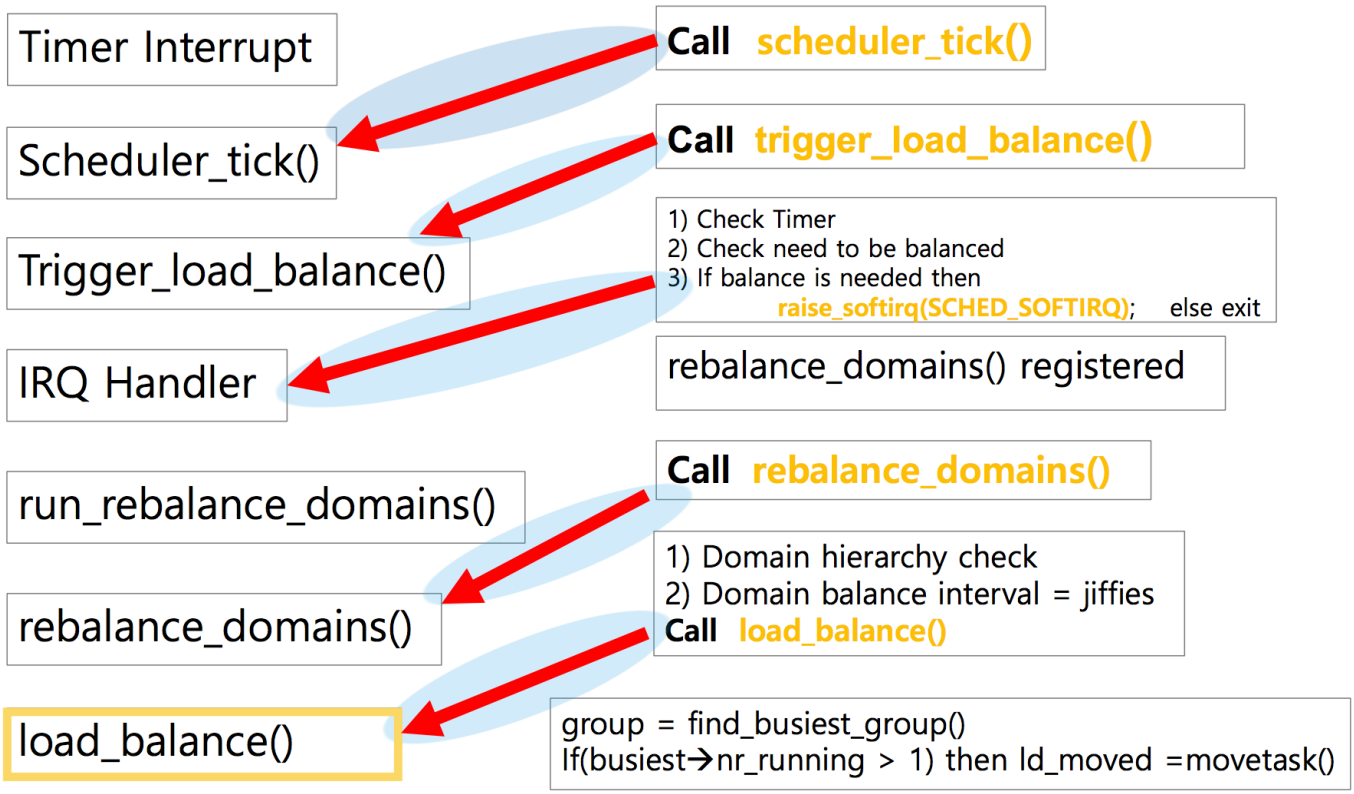
(포함) sched_group 을 포함하고 balance interval, busy_factor, 그리고 span_weight 를 포함한다

e. Active balancing

➔ **Mechanism**

각 CPU 마다 정기적으로 작동하며, kernel 은 CPU domain 부터 Scheduling domain 까지 훑어보며 balance 여부와 필요 도를 체크함

➔ **Function Chain Call**



여기서 마지막 function 인 load_balance() 가 가장 중요하다. 오른쪽 설명에서 보이듯이, 가장 load 가 심한 group 을 찾아서 offloading 을 시켜주는 역할을 하기 때문이다. 만약 SD_POWERSAVING_BALANCE flag 가 set 되어 있으면 busiest group 이 없을 경우 sched_domain 내의 load 량이 가장 작은 group 을 찾아서 CPU 에서 Idle 시켜버린다.

f. Idle Balancing

1) 목적: CPU 가 Idle 상태가 되면 schedule() function → call → Idle balancing 이 일어남

2) To do:

2.1) avg.CPU Idle time 과 load balance 하는데 걸리는 시간을 비교한다.

2.2) 만약 Migrating task 가 일리있다면 idle_balance()는 rebalance_domain()과 비슷한 동작

3) rebalance_domain() 함수는 Domain Hierarchy 를 검사하고
SD_LOAD_BALANCE Flag 가 set 되어진 Domain 에서 idle_balance() 함수를 호출하고
Idle_balance() 함수는 해당 Domain 의 Flag 를 SD_BALANCE_NEWIDLE 로 변경

4) Task 들이 1 개 이상 pulled over 되면 kernel 에서 hierarchy walking 은 종료, idle_balance() return.

g. Balancing Decision

1) Task 가 Idle 상태에서 wake up 되었을 때

2) Task 가 새로 created 되었을 때

3) Task 가 runqueue 로 진입할 때

➔ 따라서 Runqueue 는 전체적인 Task 의 balance 를 고려해야한다.

h. Selecting a Runqueue for a new task

Task 는 새로 만들어 졌을 때 자신이 실행 될 runqueue 를 선택해야 한다. 각각 runqueue 에는 scheduling class 들을 자신들 만의 task 관리 전략을 가지고 select_task_rq() (즉, hook)을 제공하며 Domain 에 해당하는 3 가지 flag 를 바탕으로 결정 된다.

```
#ifdef CONFIG_SMP
#define SD_LOAD_BALANCE          0x0001
#define SD_BALANCE_NEWIDLE       0x0002
#define SD_BALANCE_EXEC          0x0004
#define SD_BALANCE_FORK          0x0008
#define SD_BALANCE_WAKE          0x0010
#define SD_WAKE_AFFINE           0x0020
#define SD_SHARE_CPUCAPACITY     0x0080
#define SD_SHARE_POWERDOMAIN     0x0100
#define SD_SHARE_PKG_RESOURCES   0x0200
#define SD_SERIALIZE             0x0400
#define SD_ASYM_PACKING          0x0800
#define SD_PREFER_SIBLING        0x1000
#define SD_OVERLAP               0x2000
#define SD_NUMA                  0x4000
```

1) System call 인 exec() funtion 을 사용하면
sched_exec() 함수가 호출 되고
이 함수 내에서 SD_BALANCE_EXEC flag 가 사용된다.
→ 이 때, 새로운 task 의 memory 와 cache affinity 가 낮기때문에 good balancing 하기 좋은 기회가

2) 새로 만들어진 Task 가 처음 wake up 할 때
wake_up_new_task() 함수가 호출 되고
이 함수 내에서 SD_BALACNE_FORK flag 가 사용된다.

3) Task 가 running queue 에서 running 중일 때, cache affinity (친화력)을 가지게 된다
이 cache affinity 는 스케줄러가 스케줄 시 적합한 queue 에 진입할 수 있도록 도움을 준다
SD_BALANCE_WAKE flag 는 try_to_wake_up() (task 가 wake up 되었을 때) 함수 내에서 사용된다.