**운영체제 2차 과제**

Operating System 2nd Assignment Report



• 학과 : 컴퓨터학과

• 학번 : 2015410056

• 이름 : 김지윤

• 제출 일자 : 2017년 5월 4일

• 담당 교수 : 유혁 교수님

1. **CFS Scheduler 분석**
2. **CFS Scheduler의 스케줄링 정책**

CFS (Completely Fair Scheduler)의 스케줄링 알고리즘은 기본적으로 multi-tasking processor의 아이디어에 기반한다. 여러 task들이 ‘동시에’ 수행되는 multi-tasking을 흉내내기 위해서는 각 task에 배정되는 time quantum을 아주 작게 해야 하고 극심하게 빈번한 switching을 필요로 하는데 이는 아주 비효율적이다. 따라서 CFS는 **vruntime**이라는 새로운 개념을 도입해 효율적으로, multi-tasking과 최대한 가깝게 스케줄링을 수행한다.

1. **Vruntime과 Real time slice**

CFS의 수행과정을 이해하기위해서는 먼저 vruntime과 real time slice를 구분 지어 생각할 필요가 있다.

CFS는 multi-tasking을 흉내 내고자 한다. 하지만 multi-tasking은 빈번한 switching을 필요로하고 이는 엄청난 overhead를 초래하기에 CFS는 **TSL** (Target Scheduling Latency)라는 상수를 이용해 프로세스의 최소 수행시간을 정한다. 이는 각 프로세스의 priority에 따른 weight를 고려해서 계산되는데 이를 **[real time slice]**라고 한다. Real time slice는 실행주기에 (해당 프로세스의 priority weight/ 모든 프로세스의 priority weight 합)을 곱해서 계산되기 때문에 각 priority에 비례한다. 하지만 TSL이 상수로서 고정되어 있는 상황에서 실행할 프로세스의 수가 늘어나면 낮은 priority의 프로세스는 0의 가까운 시간만을 소비하게 되고 이는 비효율적인 switching을 초래하게 된다. 이러한 상황을 막기위해 CFS는 **minimum task runtime granularity**라는 값을 정해 프로세스 실행시간의 하한선을 둔다. 이렇게 해서 특정 프로세스가 CPU를 사용할 차례가 되면 real time slice만큼 또는 minimum task runtime granularity만큼의 시간동안 실행되는 것이다.

그리고 CFS는 실질적으로 사용한 시간 (real time slice)에 해당 프로세스의 weight역수에 비례하도록 가중치를 매긴 가상의 실행시간을 계산해 이를 토대로 스케줄링을 수행하게 되는데, 이것이 **[vruntime]**이라는 변수이다. 이는 real time slice에서 각 프로세스의 priority weight만큼 나눠준 값이기 때문에 process dependent한 부분이 상쇄되어 모든 프로세스에 동일하게 부여되며 높은 priority를 가진 프로세스일수록 같은 실행시간에 대해 vruntime은 더 느리게 증가한다.

이를 토대로 CFS Scheduling의 개괄적인 수행과정을 보자면 다음과 같다.

특정 프로세스가 CPU를 사용할 차례가 되면 해당 프로세스는 priority를 고려해서 계산한 값인 real time slice만큼 실행되지만 다음에 수행될 프로세스를 결정하는 스케줄링에 있어서는 vruntime이 기준이 된다. CFS에선 공정한 스케줄링을 위해 vruntime이 가장 작은 프로세스를 선택해 낮은 priority를 가진 프로세스의 수행이 high priority 프로세스에 의해 일방적으로 뒤쳐지게 되는 상황을 막는다. CFS에선 vruntime값이 최소인 프로세스에 CPU를 할당하기 때문에 새로운 프로세스가 실행 큐에 삽입되었을 때 일정시간 동안은 이 프로세스가 CPU를 독점하게 되는 문제가 발생한다. 이를 해결하기 위해서 CFS는 새로운 프로세스에 대해 vruntime으로 기존 실행 큐에 존재하는 가장 작은 vruntime, min\_vruntime값을 부여한다.

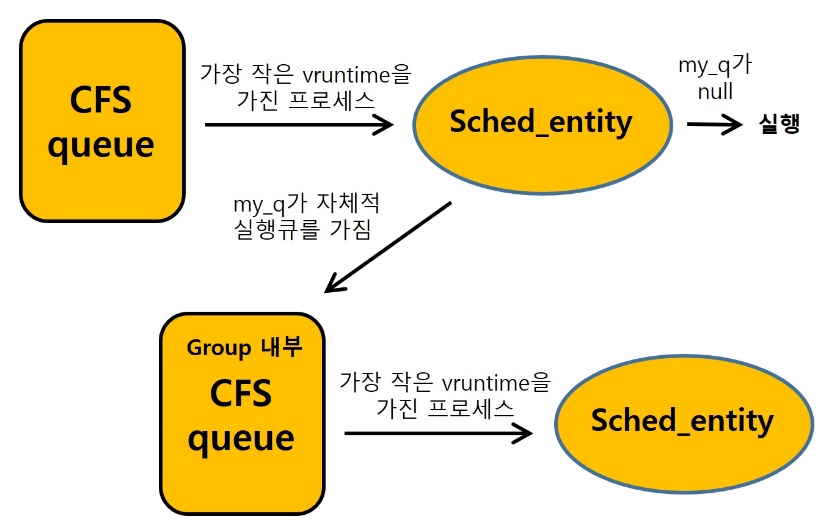
1. **Data Structure – Red Black Tree**

CFS에선 runqueue로써 Red-Black Tree를 사용한다. 각 노드는 task(프로세스)를 나타내고 vruntime에 따라 정렬된다. 따라서 트리의 leftmost 노드는 항상 가장 작은 vruntime값을 가진 프로세스가 되고 스케줄링은 단순히 leftmost노드만을 선택하면 되므로 단순해진다.

1. **Group Scheduling**

수 많은 자식 프로세스를 생산하는 프로세스를 고려해보면 스케줄링의 공평성을 위해서는 그룹 스케줄링이 필요하다는 것을 알 수 있다. CFS에선 위와 같은 프로세스를 각각 동일하게 취급하는 대신 group으로 묶어 group내에서 vruntime을 공유하게 한다.

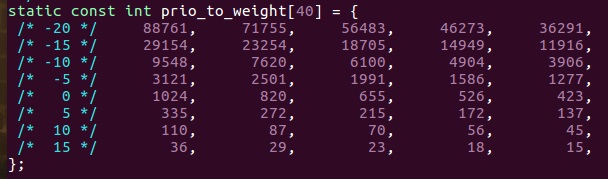
그룹에 대한 priority는 share라는 변수로 표현이 되고 프로세스를 다뤘던 방법과 똑같이 share에 따른 real time slice를 할당하고 그룹 단위의 vruntime을 갖는다. 각 그룹별로 CFS 실행 큐는 따로 가지고 있으며 이 안에서 또 다시 프로세스마다 priority에 따른 vruntime, real time slice가 주어지게 된다.

이를 위해 CFS는 sched\_entity라는 구조체를 활용하며 내부의 my\_q값이 null이면 단일 프로세스로 간주하여 일반적으로 수행하고 my\_q가 자체적인 실행 큐를 가진 sched\_entity라면 해당 큐로 다시 이동해 가장 작은 vruntime을 갖는 프로세스를 선택한다.

1. **우선순위를 결정하는 기준과 업데이트 방법**
2. **Nice**

리눅스에서 프로세스의 priority는 nice라는 값으로 표현되며 priority값을 부여할 때 배열을 사용하는데 이때 이 nice값이 index의 역할을 한다고 할 수 있다. 이는 -20에서 20까지의 범위를 가지고 아래 코드는 nice값에 따른 priority의 weight값을 미리 계산해둔 테이블로 nice값이 작을수록 더 높은 weight값과 mapping되어있다.

**weight = prio\_to\_weight[nice-20]**



Real time slice는 priority값을 고려해 계산 된다. 예를 들어 각 nice값이 -10, -5, 0인 세 프로세스 A, B, C가 있고 실행 주기는 10ms라고 한다면 A의 real time slice는 다음과 같이 계산된다. (10ms) \* { (9,548) / (9,548 + 3,121 + 1,024) }

1. **Vruntime**

위에서 언급했듯이 vruntime은 real time slice에 공통적으로 nice 0의 weight를 곱하고 각각에 대해서는 해당 프로세스의 weight 역수만큼 더 곱해준 값이기 때문에 process dependent한 부분을 상쇄시켜 모든 프로세스에 동등하게 부여되는 값이다. 위의 예시로 든 세 프로세스에 nice 값이 +5와 +10인 프로세스 D, E를 더해 총 5개의 프로세스가 실행 큐에 들어와 있다고 가정해보자. 실행주기가 10ms일 때 time slice와 vruntime은 아래 표와 같다.

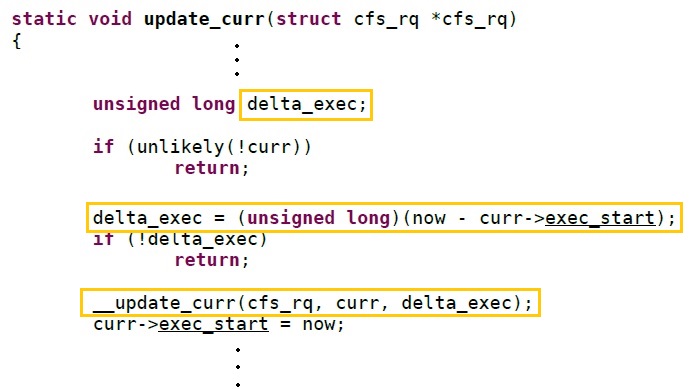
**Vruntime = 실제 실행시간 \* ( nice 0 일 때 weight / 해당 프로세스의 weight )**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Process | Nice | Wpro / Wsum | Real time slice | vruntime |
| A | -10 | 0.6753 | 6.753ms | 0.7242 |
| B | -5 | 0.2207 | 2.207ms | 0.7242 |
| C | 0 | 0.0742 | 0.724ms | 0.7242 |
| D | +5 | 0.0237 | 0.237ms | 0.7242 |
| E | +10 | 0.0078 | 0.078ms | 0.7242 |

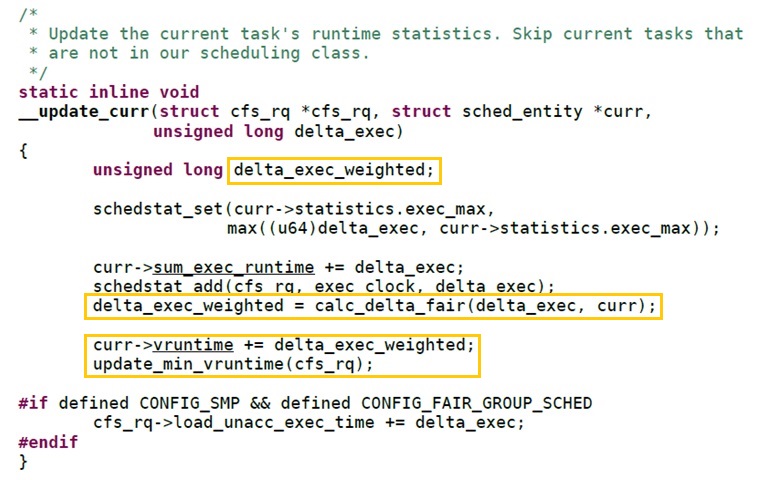
위에서 언급했듯 CFS에선 minimum task runtime granularity를 설정한다. 만약 min\_granularity값을 0.1ms로 정해 놨다면 프로세스 E의 경우 이보다 적은 time slice가 할당되었기 때문에 이 대신 0.1만큼 수행되지만 이에 따라 큰 vruntime이 반영되기 때문에 rb tree의 오른쪽 노드에 자리잡게 된다.

Vruntime은 프로세스에 상관없이 동등하게 부여되는 값이지만 그 증가속도는 weight값에 반비례하기 때문에 사실상 우선순위를 반영한다고 할 수 있다. 결국 CFS에서 vruntime의 크기가 프로세스의 우선순위가 된다.

1. **Vruntime의 업데이트**

Vruntime이 증가되는 것은 \_\_update\_curr()함수를 통해서이다. \_\_update\_curr()은 update\_curr()함수를 통해 호출되는데 아래 코드가 update\_curr()함수이다.

Update\_curr은 delta\_exec이라는 변수를 가지는데, 지난번 update\_curr()실행이후 (curr->exec\_start) 지금까지의 지나간 시간 간격 값을 담고있으며 따라서 해당 프로세스의 실행시간을 나타낸다. 이 변수값을 \_\_update\_curr()함수의 인자로서 넘겨주고 curr->exec\_start값을 현재로 설정한다. \_\_update\_curr()함수는 아래 코드와 같다.



이 함수는 delta\_exec\_weighted라는 새로운 변수를 가지는데 delta\_exec값을 vruntime으로 변환한 것이다. 우선 delta\_exec값을 sum\_exec\_runtime변수에 더해서 총 실행시간을 갱신한다. 그리고 delta\_exec\_weighted를 구해 weight가 적용된 vruntime을 curr->vruntime에 더해 해당 프로세스의 총 vruntime을 갱신한다. 이어서 min\_vruntime 또한 업데이트 시켜준다.

1. **스케줄러가 호출되는 세가지 시점**

스케줄링은 현재 프로세스가 계속 수행할 수 없는 상황을 만났을 때 발생하며 이는

1. 더 높은 우선순위를 가지는 프로세스가 수행가능한 상태가 되었을 경우
2. 현재의 프로세스가 I/O를 위해서 wait하는 경우
3. 현재 프로세스의 time slice를 다 사용한 경우

이렇게 세가지로 나누어 볼 수 있다.

비디오 인코더와 텍스트 에디터 두 프로세스가 스케줄링 되는 상황을 생각해보자.

두 프로세스는 같은 nice레벨을 갖고 있다고 가정하면 같은 time slice에 vruntime증가 속도도 같다. 이때 텍스트 에디터는 I/O bound프로세서로 사용자의 key presses를 기다리는 데에 많은 시간을 소비하지만 비디오 인코더는 processor bound로 본인 차례가 되었을 때 최대한 CPU를 활용한다. 텍스트 에디터와 비디오 인코더가 실행 큐에 들어오면 처음엔 차례대로 동일한 real time slice만큼 CPU를 사용할 수 있게 한다. 이때 텍스트 에디터가 I/O (key press)를 위해 wait 하게 되면 스케줄링이 발생 (경우 ②) 하게 되고 비디오 인코더가 CPU를 주어진 시간동안 사용하게 된다. 이 사이 텍스트 에디터가 wake up하게 되면 그의 vruntime이 더 작기 때문에 (우선순위가 높기 때문에) 비디오 인코더를 선점하게 된다. (경우 ①) 또한 실행중인 프로세스가 time slice를 모두 사용했을 경우 자연적으로 스케줄링이 발생하여 vruntime이 가장 작은 프로세스를 선택한다. (경우 ③)

1. **Task가 wake up 되었을 때의 스케줄러**

Sleeping 상태의 task가 wake up 되었을 때 이는 wake\_up()을 호출하여 다시 스케줄링 되는데 이 함수가 하는 동작은 다음과 같다.

1. Wake up될 task를 다시 runqueue상에 올려놓는다.
2. 해당 task의 상태를 TASK\_RUNNING상태로 세팅한다.
3. 만약 깨어난 task가 현재 실행중인 다른 task보다 우선순위가 높으면 (vruntime이 더 작다면) need\_resched라는 flag가 세팅되며 이는 schedule()을 호출하여 CPU를 선점하게 한다.
4. **또 다른 스케줄링 아이디어**

실행 큐에 들어간 프로세스들 중 특정 프로세스의 속도를 조절하는 방법으로는 그룹 스케줄링의 활용을 생각 해볼 수 있다.

속도를 높이고 싶은 프로세스를 제외한 나머지 프로세스들에 대해 nice값이 같은 것끼리 두 개 단위로 묶어 그룹을 생성하는 것이다. 그룹에 대한 priority(share)는 해당 nice값에 대한 weight를 그대로 준다. 이때 동일한 nice값을 가진 프로세스가 없고 자기 자신 뿐이라면 그대로 수행하되 그룹에 속해 있는 것처럼 real time slice에 따른 vruntime값을 두배로 해서 그룹을 생성한 프로세스들과 동등하게 해준다. 이렇게 하면 CPU를 사용할 기회가 두배로 늘어나는 셈이기 때문에 해당 프로세스를 빠르게 수행 할 수 있다.

이 방법을 활용하면 기존 CFS 스케줄러가 가지고 있는 아래 상황과 같은 문제점도 해결 가능하다

Nice값이 낮은 (priority weight가 높은) 프로세스 여러 개와 이들과 weight값이 차이가 많이 나는 프로세스들이 실행 큐에 존재한다고 하자. 이때 low priority 프로세스에 대해 같은 vruntime당 배정되는 real time slice는 극히 작을 것이다. minimum task runtime granularity를 정해 차례가 되면 일정한 시간 동안 실행할 수 있도록 한다고 해도 실제 실행 시간에 비해 큰 vruntime이 누적될 것이고 low weight 프로세스들은 weight 값이 높은 프로세스들이 상대적으로 많은 시간을 소비하며 vruntime을 천천히 채워 본인의 vruntime을 따라잡을 때까지 기다려야만 한다. 이에 대해서 위의 방법을 활용 할 수 있는데 프로세스들의 값들 중 min\_nice와 일정 값 (예를 들면 15) 이상 차이나는 프로세스들은 제외하고 그룹으로 묶는 것이다. 예를 들어 프로세스 A, B, C, D, E가 존재하고 각 nice 값은 -18, -18, -15, 0, 5라고 했을 때 프로세스 A와 B를 하나의 그룹으로 묶고 share로 nice -18에 해당되는 weight을 부여한다. C도 위에 언급한 것과 같이 그룹에 속해 있는 것처럼 한다. 프로세스 D와 E에 대해서는 min\_nice와 nice값이 15이상 차이 나기 때문에 단일 프로세스로 취급해서 실행하는데 이렇게 되면 weight값이 작은 프로세스들의 대기시간은 꽤 줄어든다.

1. **Load Balancing on SMP System**
2. **SMP (Symmetric Multiprocessing) System & Load Balancing**

SMP 시스템이란 두 개 또는 그 이상의 CPU가 한 개의 공유 메모리를 사용하는 다중 프로세서 컴퓨터 구조이다. 하나의 운영체제가 모든 프로세서를 관리하며 메모리와 I/O Bus 및 데이터 경로 등 또한 공유한다.

Load Balancing은 busy CPU의 task를 idle 상태의 CPU로 옮겨 CPU 활용도를 최대로 높이기 위해 도입되었다. 리눅스 스케줄러는 주기적으로 여러 CPU들 상에 task들이 어떻게 분산되어 있는지 체크하고 필요하다면 load balancing을 수행한다.

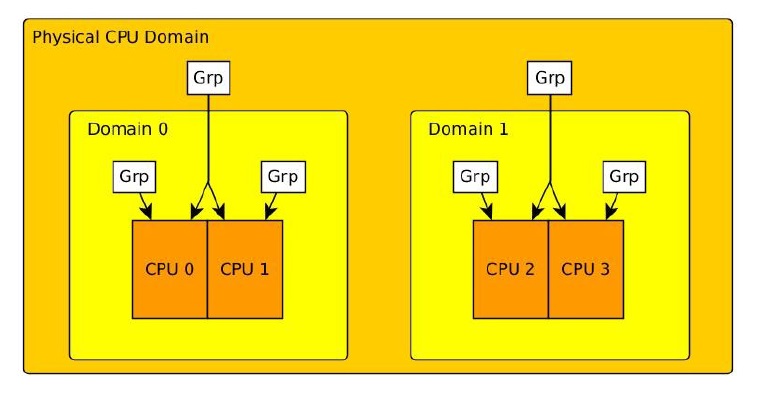
그러나 SMP system의 topology는 다양하게 존재하여 busy CPU의 task를 idle한 CPU로 옮기는 것이 캐시 무효화 등에 따른 비용이 더 커서 오히려 비효율을 초래하는 경우도 있고 캐시를 공유하여 이러한 task 이동을 더 유연하게 해내는 경우도 있다. 또한 NUMA 구조는 공유 메모리에 접근하는 시간이 모든 프로세스에 동일했던 기존의 방식과 달리 메모리와 프로세스 간의 상대적인 위치에 따라 달라지게 되므로 이야기가 또 달라진다. 이러한 다양한 구조의 시스템을 다루기 위해 scheduling domain이라는 개념이 도입된다.

1. **Scheduling Domain & Scheduling Group**

**Scheduling Domain** 이란 스케줄링 정책과 특성을 공유하고 task들이 그 사이에서 서로 분산 (Balancing)될 수 있는 CPU set을 말한다. 각 domain은 하나 이상의 **Scheduling group**을 포함하는데 이는 스케줄링 도메인 위에서 한 단위로 취급된다. 따라서 domain상에서 group간 load balancing이 일어날 땐 그룹 내부에 대한 처리는 고려하지 않는다. domain위엔 더 높은 계층의 physical CPU Domain이 존재한다.

Scheduling Group은 CPU별로 할당되면서 동시에 domain 내 여러 CPU를 묶어서 구성되기도 한다.

NUMA 시스템의 경우 아래 그림에서 보는 물리적 CPU domain이 여러 개 존재한다고 보면 된다. 각 도메인은 NUMA node로써 표현되며 이 경우 제 3의 Physical CPU Domain이 모든 NUMA node를 포함한다.



스케줄링 도메인과 스케줄링 그룹은 코드에서 각 sched\_domain, sched\_group이라는 자료구조로서 존재하며 스케줄링 도메인이 그룹을 포함하는 상위 자료구조이다. 스케줄링 그룹은 group weight와 다른 그룹의 주소를 포함하고 스케줄링 도메인은 스케줄링 그룹의 주소와 balancing 간격의 최대 최소를 나타내는 변수 등을 포함한다.

1. **Balancing Policy**

각 스케줄링 도메인은 balancing policy를 가지는데 해당 계층의 domain내에서만 유효하다. Policy parameter은 load balancing을 언제, 얼마나 자주 시도 할 것인지, load balancing 시도 이전에 특정 프로세서에 대해 어느 정도의 load를 허용할 것인지, 특정 프로세스의 cache affinity(사용할/사용하지 않을 CPU를 명시함)가 고려되지 않을 정도의 idle 시간은 어느 정도인지 등의 내용을 포함한다.

또한 특정 상황 (예를 들면 한 CPU가 idle상태일 때 balancing이 가져올 task를 찾아야 하는지, task가 wake up되었을 땐 어떤 CPU에 스케쥴 되어야 하는지)에 대한 처리를 명시해주는 다양한 Policy flags를 활용해 Balancing을 수행한다.

1. **Active Balancing**

Active load balancing은 각 CPU위에서 규칙적으로 일어난다. Active balancing 동안 커널은 현재의 CPU가 속해 있는 domain에서부터 출발해 각 도메인 계층을 돌아다니면서 balance될 예정인지 살피고 만약 그렇다면 balancing operation을 초기화 시킨다.

스케줄러가 초기화 될 때 soft irq 핸들러가 정기적인 로드 분산을 수행하기 위해 등록된다. 이러한 동작은 *scheduler\_tick()*에서 *trigger\_load\_balance()*를 통해 호출된다. *Trigger\_load\_balance()*는 타이머를 체크해 balancing이 예정되어 있는지 확인하고 해당 flag인 SCHED\_SOFTIRQ로 soft irq를 발생시킨다.

그리고 irq 핸들러로서 등록되어 있는 함수는 *run\_rebalance\_domains()*이며 이는 실제 수행을 위해 *rebalance\_domains()*을 호출한다.

*Rebalance\_domains()*는 도메인 계층을 돌아다니며 만약 도메인이 SD\_LOAD\_BALANCE flag가 set되어 있고 balancing 간격이 만료되었다면 *load\_balance()*를 호출한다. 이때 active balancing은 실행중인 CPU에 대한 pull 동작이다. 즉, 과부하된 다른 CPU의 일부 task를 현재 실행중인 CPU에 적재하는 것이다. 이러한 동작을 하는 것이 *load\_balance()*함수이고 불균형한 그룹을 발견하면 하나 이상의 task를 현재의 CPU로 이동시키고 0이상의 값을 리턴한다.

*Load\_balance()*는 주어진 도메인 상에서 imbalance한 그룹을 찾아 busiest group으로 리턴하는 함수인 *find\_busiest\_group()*을 호출한다. 조건에 맞는 그룹을 찾게 되면 리턴 값은 *find\_busiest\_queue()*로 넘겨지게 되는데 이는 해당 그룹의 busiest logical CPU를 리턴한다. 그리고 *load\_balance()*는 결과로 나온 CPU의 실행 큐를 검색해 *move\_task()*를 호출하여 현재 CPU의 큐로 넘겨줄 작업을 찾는다. *find\_busiest\_group()*에 설정된 Imbalance 매개변수가 이동해야 하는 task의 수를 지정하는데 이때 cache affinity로 인해 모든 task가 대상이 될 수 있다. 이러한 경우 *load\_balance()*는 이전에 대상이 되었던 CPU를 제외하고 검색을 다시 시작한다.

\*\**find\_busiest\_group()*함수에는 에너지 절약에 대한 기능이 숨겨져 있는데, 해당 domain policy에 SD\_POWERSAVINGS\_BALANCE flag가 설정되어 있고 busiest group이 없는 경우 이 함수는 sched\_domain에서 load가 가장 적게 적재되어 있는 그룹을 찾아 해당 CPU를 idle 상태로 돌릴 수 있다. 하지만 이러한 기능은 안드로이드 커널에선 지원되지 않으며 우분투에서도 제거되었다.

1. **Idle Balancing**

Idle balancing은 CPU가 idle 상태로 돌입하는 즉시 발생한다. 그러므로 이는 특정 CPU의 스케줄링 실행 큐가 빈 상태가 될 때 *schedule()*함수를 통해 동작된다.

먼저, idle한 CPU 실행 큐의 평균 idle 상태 지속 시간이 다른 CPU의 task를 해당 CPU로 옮기는 비용보다 큰지 확인한다. 즉, task를 옮길 만한 가치가 있는지 아니면 기존의 CPU에 그대로 두고 다른 task의 수행을 기다리는 것이 더 나은지를 확인한다. Task를 옮기는 작업이 더 효율적인 경우 *idle\_balance()*함수가 동작되는데 이는 active balance의 *rebalance\_domain()*과 거의 흡사하게 진행된다. 하나 혹은 그 이상의 task가 idle한 CPU로 넘겨지면 검색을 멈추고 해당 함수를 종료 시킨다.

1. **새로운 task에 대한 runqueue 배정**

Balancing이 고려해야할 세 번째 사항은 task가 wake up되거나 새로 생성되어 실행 큐에 배치되어야 할 때이다. Runqueue는 전체 task balancing을 고려하여 선택되어야 한다. 각 스케줄링 클래스는 자신의 task를 처리하기 위한 자체 전략을 구현하고 스케줄러에 의해 호출되는 select\_task\_rq()를 제공하는데 이는 SD\_BALANCE\_EXEC, SD\_BALANCE\_FORK, 그리고 SD\_BALANCE\_WAKE의 세 가지 flag중 하나가 setting되었을 경우 발생한다.