TI DSP, MCU, Xilinx Zynq FPGA Based Programming Expert Program

Instructor – Innova Lee (Sanghoon Lee)
gcccompil3r@gmail.com
Student – Howard Kim (Hyungju Kim)
mihaelkel@naver.com

리눅스 커널 내부 구조

1.실시간 태스크 스케줄링

일반 태스크 : SCHED_NORMAL, SCHED_BATCH 등.

실시간 태스크 : FIFO, RR, DEADLINE

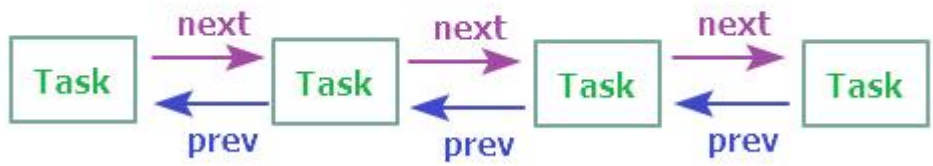
이전에 살펴본 바와 같이, 모든 태스크는 이중 연결 리스트 로 이루어져 있다.

```
task_struct{
        :
  struct list_head tasks;
        :
}
```

에서 struct list_head의 정의를 살펴보면

```
struct list_head {
    struct list_head *next, *prev;
};
```

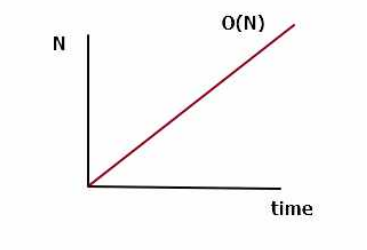
next 노드를 가리키는 포인터와 prev 노드를 가리키는 포인터가 있다는 것을 확인할 수 있으므로, 모든 태스크들이 이중 연결 리스트로 이루어져 있음을 알 수 있다.



FIFO : Firts In First Out 구조로, 큐 형태의 자료구조 안에 태스크가 저장되어 우선순위를 검색하여 가장 높은 우선순위의 태스크를 골라낸다.

RR : 동일 우선순위를 가지는 태스크가 여러 개일 경우, 타임 슬라이스를 기준으로 스케줄링된다. 즉, 동일 우선순위를 가지는 태스크가 없다면, FIFO 정책이랑 같은 방식이다.

위의 FIFO 방식과 RR 방식은, 이중 연결 리스트로 이루어진 태스크들 사이에서, 어떤 태스크가 우선순위가 가장 높은지에 대한 검색이 필요하다. 연결 리스트의 검색 시간 복잡도는 O(n)이다. 즉, 태스크의 개수가 늘어나면, 스케줄링에 걸리는 시간도 선형적으로 증가하여 스케줄링에 소모되는 시간을 예측할 수가 없다.



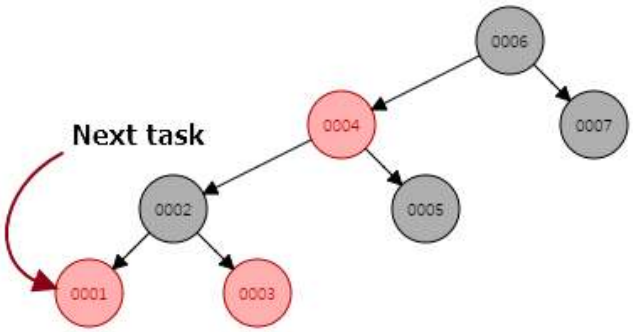
리눅스 커널 내부 구조

단순히 연결 리스트를 통해 우선순위를 알아내는 방식은 과거 리눅스 커널 버전 2.4의 스케줄러였다. 시간 복잡도를 줄이기 위해, 지금은 어떤 방식을 도입했을까?

검색에 걸리는 시간을 줄이면 된다. 즉, 모든 태스크들을 검색하여 우선순위를 알아내는 것이 아닌, 우선순위만을 저장해놓은 다른 무언가를 통하면 된다. 고정된 크기의 비트맵을 만든 후, 태스크 생성시 해당 태스크의 우선순위에 해당하는 비트를 1로 세팅한다. 스케줄링 시 비트맵에서 가장 처음으로 set된 비트가 최우선순위에 해당하므로, 그 우선순위에 해당하는 큐에 있는 태스크를 꺼내면 된다.

이러한 보완에도 불구하고, FIFO와 RR 정책은 사용하기 좋지는 않다. 모든 태스크가 공평하게 스케줄링이 되지만, 공평 = 공정은 아니기 때문이다. 시간이 많이 할당되어야 하는 태스크한테는 더 많이 CPU를 점유하게 할 필요가 있는데, 이러한 정책을 DEADLINE 정책이라고 한다.

DEADLINE : 간단히 말하면, 모든 태스크들이 정해진 시간(Dead line) 안에 수행되도록 스케줄링 하는 방법이다. deadline에 가장 가까운 태스크(deadline - 현재시간 - runtime이 가장 적은 태스크)부터 꺼낸다. 어떤 값이 가장 작은 것을 검색하는 데에는, tree 자료구조가 가장 효율적이다. 제일 왼쪽 노드를 검색하기만 하면 된다. 태스크가 스케줄링을 할 때에는, 삽입과 삭제가 매우 빈번하므로 삽입 및 삭제의 시간복잡도가 밸런스가 가장 좋은 Red-Black Tree를 사용한다.




```
477 struct dl_rq {
478     /* runqueue is an rbtree, ordered by deadline */
479     struct rb_root rb_root;
480     struct rb_node *rb_leftmost;
481
482     unsigned long dl_nr_running;
483
484 #ifdef CONFIG_SMP
485     /*
486      * Deadline values of the currently executing and the
487      * earliest ready task on this rq. Caching these facilitates
488      * the decision wether or not a ready but not running task
489      * should migrate somewhere else.
490      */
491     struct {
492         u64 curr;
493         u64 next;
494     } earliest_dl;
495
496     unsigned long dl_nr_migratory;
497     int overloaded;
498
499     /*
500      * Tasks on this rq that can be pushed away. They are kept in
501      * an rb-tree, ordered by tasks' deadlines, with caching
502      * of the leftmost (earliest deadline) element.
503      */
504     struct rb_root pushable_dl_tasks_root;
505     struct rb_node *pushable_dl_tasks_leftmost;
506 #else
507     struct dl_bw dl_bw;
508 #endif
509 };
```

이는 kernel/sched/sched.h 에서도 확인할 수 있다.

찾는 과정 : 스케줄링 방식이니, task와 관계가 있을 것이다. 그렇다면 task_struct에 있지 않을까? vi -t task_struct를 찾는다. 없다. 하지만 분명 존재하는 개념이므로, 다른 어딘가에 있을 것이다. task_struct 이외의 어디에 있는지 감이 안온다면, grep을 해보자. grep -rn dl_rq ./ 를 해보니, kernel/sched/sched.h 에 해당하는 구조체가 선언되있음을 알 수 있다.

struct rb_node *rb_leftmost;

가장 왼쪽 노드를 의미하고, 값이 가장 작은 노드이므로 deadline까지 가장 얼마 안남은, 제일 먼저 처리해야 할 태스크가 저장되어 있음.

리눅스 커널 내부 구조

2.일반 태스크 스케줄링

CFS 스케줄링 : 말 그대로, 가장 공정한 스케줄링(Completely Fair Scheduling) 이다.
공정이란 무엇일까? 여러 개의 태스크가 있다고 할 때, 각 태스크에게 같은 시간을 할당하는 것은 공평하다고는 할 수 있을지 몰라도, 공정하다고는 할 수 없다. 우선 순위가 높은 태스크에게는 더 많은 시간을 할당해야 한다. 그래서 생긴 개념이 vruntime이다. 모든 태스크는 같은 양의 vruntime을 할당받는다. 다만, 우선순위가 높은 태스크는 그 vruntime을 모두 소모하는 데 더 오랜 시간이 걸려, 더 많이 CPU를 점유할 수 있다.

vruntime += 가중치*실제시간;

가중치 = $\frac{Weight_0}{Weight_{curr}}$

위의 계산식에서 가중치가 작을수록, 같은 vruntime 동안 더 많은 실제 시간을 할당받을 수 있다. 어떤 개념인지 보다 자세히 설펴보기 위해, 커널을 찾아보자.

찾는 과정 : 각각의 태스크마다 고유의 vruntime 및 가중치를 가져야 한다. 따라서, 관련된 코드가 task_struct 아래에 있을 확률이 매우 높다. 안에 보니, struct sched_entity 라는 구조체가 눈에 띈다. 뭔가 스케줄링 관련 요소들을 모아놓은 구조체라는 뜻이니, 관련이 있을 것 같다.

```
1244 struct sched_entity {
1245     struct load_weight  load;       /* for load-balancing */
1246     struct rb_node      run_node;
1247     struct list_head    group_node;
1248     unsigned int        on_rq;
1249
1250     u64         exec_start;
1251     u64         sum_exec_runtime;
1252     u64         vruntime;
1253     u64         prev_sum_exec_runtime;
1254
1255     u64         nr_migrations;
1256
1257 #ifdef CONFIG_SCHEDSTATS
1258     struct sched_statistics statistics;
1259 #endif
1260
1261 #ifdef CONFIG_FAIR_GROUP_SCHED
1262     int         depth;
1263     struct sched_entity *parent;
1264     /* rq on which this entity is (to be) queued: */
1265     struct cfs_rq       *cfs_rq;
1266     /* rq "owned" by this entity/group: */
1267     struct cfs_rq       *my_q;
1268 #endif
1269
1270 #ifdef CONFIG_SMP
1271     /* Per entity load average tracking */
1272     struct sched_avg    avg;
1273 #endif
1274 };
```

대놓고 vruntime이라는 변수가 있고, 밑에는 struct cfs_rq라는 구조체의 포인터가 선언되어 있다. 가중치 적용을 리눅스는 어떤 식으로 구현했는지 알아보자. struct load_weight이라는 구조체에 왠지 있을 것 같다.

리눅스 커널 내부 구조

```
1183 struct load_weight {
1184     unsigned long weight;
1185     u32 inv_weight;
1186 };
1187
1188 /*
1189  * The load_avg/util_avg accumulates an infinite geometric series.
1190  * 1) load_avg factors frequency scaling into the amount of time that a
1191  * sched_entity is runnable on a rq into its weight. For cfs_rq, it is the
1192  * aggregated such weights of all runnable and blocked sched_entities.
1193  * 2) util_avg factors frequency and cpu scaling into the amount of time
1194  * that a sched_entity is running on a CPU, in the range [0..SCHED_LOAD_SCALE].
1195  * For cfs_rq, it is the aggregated such times of all runnable and
1196  * blocked sched_entities.
1197  * The 64 bit load_sum can:
1198  * 1) for cfs_rq, afford 4353082796 (=2^64/47742/88761) entities with
1199  * the highest weight (=88761) always runnable, we should not overflow
1200  * 2) for entity, support any load.weight always runnable
1201  */
```

주석을 보니, 가장 큰 가중치가 88761이라는 것이 눈에 띈다. 이를 토대로 검색해보니 가중치 배열이 나왔다(안나와서 멘붕오다가 막 하다보니깐 됐다..)

```
1128 static const int prio_to_weight[40] = {
1129  /* -20 */     88761,     71755,     56483,     46273,     36291,
1130  /* -15 */     29154,     23254,     18705,     14949,     11916,
1131  /* -10 */      9548,      7620,      6100,      4904,      3906,
1132  /*  -5 */      3121,      2501,      1991,      1586,      1277,
1133  /*   0 */      1024,       820,       655,       526,       423,
1134  /*   5 */       335,       272,       215,       172,       137,
1135  /*  10 */       110,        87,        70,        56,        45,
1136  /*  15 */        36,        29,        23,        18,        15,
1137 };
```

3.태스크와 시그널

앞서 시스템 프로그래밍에서 시그널 사용법에 대해 배웠다. 기본적으로 프로세스는 시그널을 맞으면 프로세스가 종료되었다. 시스템 콜인 signal() 함수를 통해, 특정 시그널에 대한 처리 루틴, 즉, 핸들러를 지정할 수 있었다.

task_struct를 보면, signal 관련 변수들이 있는 것을 볼 수 있다.

```
1566 /* signal handlers */
1567     struct signal_struct *signal;
1568     struct sighand_struct *sighand;
1569
1570     sigset_t blocked, real_blocked;
1571     sigset_t saved_sigmask; /* restored if set_restore_sigmask() was used */
1572     struct sigpending pending;
1573
1574     unsigned long sas_ss_sp;
1575     size_t sas_ss_size;
1576
1577     struct callback_head *task_works;
1578
1579     struct audit_context *audit_context;
```

struct signal_struct* signal은 시그널을 수신하기 위한 변수

struct sigpending pending 역시 시그널을 수신하기 위한 변수,

차이점은 해당 process에게 signal을 보낼 것인가 process에 속한 특정 thread에게만 보낼 것인가 등.

struct sighand_struct *sighand;는 시그널에 대한 처리 핸들러 관련 변수일 듯 하다.

리눅스 커널 내부 구조

```
1183 struct load_weight {
1184     unsigned long weight;
1185     u32 inv_weight;
1186 };
1187
1188 /*
1189  * The load_avg/util_avg accumulates an infinite geometric series.
1190  * 1) load_avg factors frequency scaling into the amount of time that a
1191  * sched_entity is runnable on a rq into its weight. For cfs_rq, it is the
1192  * aggregated such weights of all runnable and blocked sched_entities.
1193  * 2) util_avg factors frequency and cpu scaling into the amount of time
1194  * that a sched_entity is running on a CPU, in the range [0..SCHED_LOAD_SCALE].
1195  * For cfs_rq, it is the aggregated such times of all runnable and
1196  * blocked sched_entities.
1197  * The 64 bit load_sum can:
1198  * 1) for cfs_rq, afford 4353082796 (=2^64/47742/88761) entities with
1199  * the highest weight (=88761) always runnable, we should not overflow
1200  * 2) for entity, support any load.weight always runnable
1201  */
```

주석을 보니, 가장 큰 가중치가 88761이라는 것이 눈에 띈다. 이를 토대로 검색해보니 가중치 배열이 나왔다(안나와서 멘붕오다가 막 하다보니깐 됐다..)

```
1128 static const int prio_to_weight[40] = {
1129  /* -20 */     88761,     71755,     56483,     46273,     36291,
1130  /* -15 */     29154,     23254,     18705,     14949,     11916,
1131  /* -10 */      9548,      7620,      6100,      4904,      3906,
1132  /*  -5 */      3121,      2501,      1991,      1586,      1277,
1133  /*   0 */      1024,       820,       655,       526,       423,
1134  /*   5 */       335,       272,       215,       172,       137,
1135  /*  10 */       110,        87,        70,        56,        45,
1136  /*  15 */        36,        29,        23,        18,        15,
1137 };
```

3.태스크와 시그널

앞서 시스템 프로그래밍에서 시그널 사용법에 대해 배웠다. 기본적으로 프로세스는 시그널을 맞으면 프로세스가 종료되었다. 시스템 콜인 signal() 함수를 통해, 특정 시그널에 대한 처리 루틴, 즉, 핸들러를 지정할 수 있었다.

task_struct를 보면, signal 관련 변수들이 있는 것을 볼 수 있다.

```
1566 /* signal handlers */
1567     struct signal_struct *signal;
1568     struct sighand_struct *sighand;
1569
1570     sigset_t blocked, real_blocked;
1571     sigset_t saved_sigmask; /* restored if set_restore_sigmask() was used */
1572     struct sigpending pending;
1573
1574     unsigned long sas_ss_sp;
1575     size_t sas_ss_size;
1576
1577     struct callback_head *task_works;
1578
1579     struct audit_context *audit_context;
```

struct signal_struct* signal은 시그널을 수신하기 위한 변수

struct sigpending pending 역시 시그널을 수신하기 위한 변수,

차이점은 해당 process에게 signal을 보낼 것인가 process에 속한 특정 thread에게만 보낼 것인가 등.

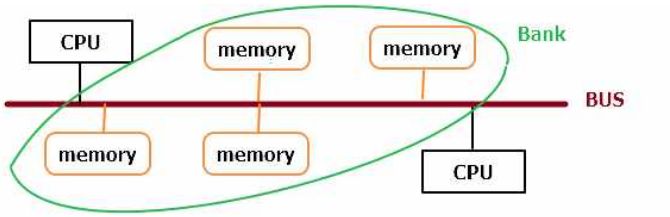
struct sighand_struct *sighand;는 시그널에 대한 처리 핸들러 관련 변수일 듯 하다.

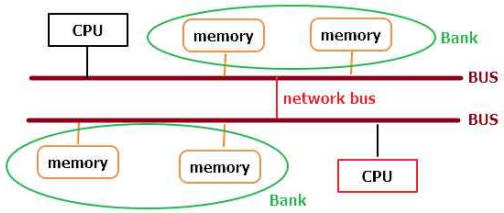
리눅스 커널 내부 구조

Chapter 4. 메모리 관리

1.UMA와 NUMA

UMA와 NUMA는 CPU가 물리 메모리에 접근하는 방식을 나타낸다. 접근 속도가 같은 메모리들의 집합을 Bank라고 한다.





UMA NUMA

Bank를 표현하는 자료 구조가 Node 이다.

UMA 구조든 NUMA 구조든 하나의 Node는 pg_data_t 구조체를 통해 표현된다.

unsigned long node_start_pfn : 물리 메모리가 메모리 맵의 몇 번지에 있는지 나타냄

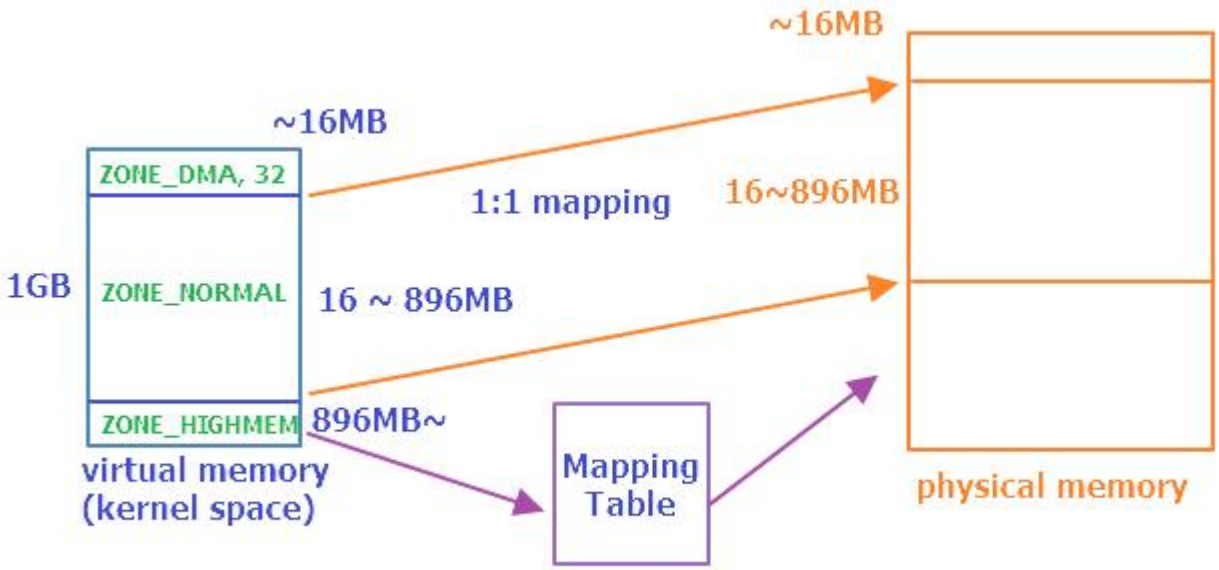
struct zone node_zones[MAX_NR_ZONES] : zone 구조체 저장 변수

int nr_zones : zone의 개수

```
637 typedef struct pglist_data {
638     struct zone node_zones[MAX_NR_ZONES];
639     struct zonelist node_zonelists[MAX_ZONELISTS];
640     int nr_zones;
641 #ifdef CONFIG_FLAT_NODE_MEM_MAP /* means !SPARSEMEM */
642     struct page *node_mem_map;
643 #ifdef CONFIG_PAGE_EXTENSION
644     struct page_ext *node_page_ext;
645 #endif
646 #endif
647 #ifndef CONFIG_NO_BOOTMEM
648     struct bootmem_data *bdata;
649 #endif
650 #ifdef CONFIG_MEMORY_HOTPLUG
651     /*
652      * Must be held any time you expect node_start_pfn, node_present_pages
653      * or node_spanned_pages stay constant.  Holding this will also
654      * guarantee that any pfn_valid() stays that way.
655      *
656      * pgdat_resize_lock() and pgdat_resize_unlock() are provided to
657      * manipulate node_size_lock without checking for CONFIG_MEMORY_HOTPLUG
658      *
659      * Nests above zone->lock and zone->span_seqlock
660      */
661     spinlock_t node_size_lock;
662 #endif
663     unsigned long node_start_pfn;
664     unsigned long node_present_pages; /* total number of physical pages */
665     unsigned long node_spanned_pages; /* total size of physical page
666                          range, including holes */
667     int node_id;
668     wait_queue_head_t kswapd_wait;
669     wait_queue_head_t pfmemalloc_wait;
670     struct task_struct *kswapd; /* Protected by
671                        mem_hotplug_begin/end() */
672     int kswapd_max_order;
673     enum zone_type classzone_idx;
674 #ifdef CONFIG_NUMA_BALANCING
```

리눅스 커널 내부 구조

zone의 메모리 접근



2.external fragmentation & internal fragmentation

물리 메모리가 아래와 같이 할당중이라고 가정하자.

위의 물리 메모리가 4GB라고 할 때, 남는 공간은 약 2GB가 된다. 공간은 2GB가 있지만, kmalloc을 통해 2GB만큼 메모리를 할당하면, 할당이 되지 않는다.(빈 공간중 최대로 연속된 공간이 400MB이기 때문에) 이런 현상을 **외부 단편화(external fragmentation)**이라고 한다. 외부 단편화를 피하기 위해 vmalloc을 써야 하고, 이는 cache를 충분히 활용하지 못하는 결과가 된다.

이번엔 12byte 정도의 매우 작은 메모리를 할당한다고 하자. 물리 메모리의 최소 단위는 페이지 단위로서, 보통 4KB가 된다. 한 페이지 내의 12byte를 제외한, 4084byte가 낭비가 된다. 이런 현상을 **내부 단편화(internal fragmentation)**이라고 한다.

리눅스 커널 내부 구조

3.버디 할당자 & 슬랩 할당자

버디 할당자 : 외부 단편화를 최소화 하기 위해, 메모리를 페이지x2^n 단위로 할당하는 것.
2^n 단위로 할당 및 해제를 순서대로 읽어서 한다. 따라서 연속된 큰 메모리 덩어리를 최대한 많이 유지하게 되어 캐시를 최대한 활용할 수 있다는 장점이 있다.
단점도 있다. 어느 한 페이지 프레임을 할당 및 해제의 반복을 한다고 하자. 상위 order에서 페이지 프레임을 쪼개어 2개의 하위 order로 나누고, 다시 2개의 하위 order의 페이지 프레임 2개를 상위 order 단위 1개로 만드는 작업을 반복할 수도 있다. order를 변경하는 것 자체가 연산이 들어가고, 이것이 많아진다는 것은 결국 overhead가 증가하게 된다는 것이다.

Lazy 버디 할당자 : Lazy Buddy는 위의 버디 할당자의 단점을 커버한 방법이다.
즉, 할당과 해제가 반복적일 경우가 문제이기 때문에, 메모리 해제를 바로 하지 않고 뒤로 미루는 방법이다.

슬랩 할당자 : 페이지의 단위는 4KB이다. 4KB는 생각보다 상당히 큰 크기이다. 자료구조를 만든다고 생각해보자. 큐 자료구조를 동적 할당시 12byte를 할당한다.(데이터가 int형이라고 할 때)
이런 내부 단편화를 방지하기 위한 방법이 슬랩 할당자이다. 미리 페이지 프레임 1개, 4KB를 할당 받은 다음 그 내부 공간을 내주는 개념이다. 즉, 일종의 캐시처럼 사용할 수 있다.