TI DSP,MCU 및 Xilinux Zynq FPGA

프로그래밍 전문가 과정

이름	문지희
학생 이메일	mjh8127@naver.com
날짜	2018/4/10
수업일수	34 일차
담당강사	Innova Lee(이상훈)
강사 이메일	gcccompil3r@gmail.com

목차

NPTL(Native POSIX Thread Library)

: NGPT 처럼 n:1 방식으로 매칭되는 것이 아닌 task_struct 를 했을 때 커널에 존재하는 커널 전용 프로그램과 유저 전용 프로그램이 1:1매칭이 된다. n:1 방식은 프로세스 중앙에서 관리하는 사이즈가 너무 커 메모리를 올리기 힘들고 그러다 보니 캐시가 깨져 캐시를 활용하지 못하는 문제가 있게 되는데 NPTL 은 1:1 매칭을 함으로서 NGPT 에서 나타난 문제들을 해결했다. 또 프로그램이 너무 무거워지지 않게 프로그램 사이즈를 커널 스택을 별도로 만들어 크기를 제한하고 유저랑 통신하며 사용할 정보를 커널 스택에서 활용한다. 커널 스택은 thread_union 에 위치해 있고 NPTL 을 사용함으로써 성능이 빨라지게 되었다.

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

Chapter 3 태스크 관리

(7) 런 큐와 스케줄링

스케줄링 : 여러 개의 태스크들 중에서 다음번 수행시킬 태스크를 선택하여 CPU 라는 자원을 할당하는 과정.
리눅스는 140개의 우선순위 중 실시간 태스크는 0~99단계로 real time 방식으로 실시간 태스크 제어를 한다. 나머지 일반태스크는 100~139까지 동적 우선순위를 사용하게 되고 실시간 태스크는 항상 일반태스크보다 우선순위가 높다. (숫자가 낮을수록 높은 우선순위를 나타낸다.)

7-1 런큐와 태스크

운영체제는 스케줄링 작업 수행을 위해, 수행 가능한 상태의 태스크를 자료구조(이중연결 리스트 next, prev)를 통해 관리한다. 리눅스에서는 이 자료구조를 런 큐라 한다. 운영체제의 구현에 따라 런큐는 한 개 혹은 여러 개 존재할 수 있다. 런 큐는 task_struct 내에 cfs_rq 와 rt_rq 로 구현되어 있는데 일반적인 스케줄링 방식과 real time 스케줄링 방식 두 가지가 있다. 리눅스는 하이브리드 방식이어서 둘 다 사용한다.

```
        struct sched_entity     *parent;
        /* rq on which this entity is (to be) queued: */
        struct cfs_rq           *cfs_rq;
        /* rq "owned" by this entity/group: */
        struct cfs_rq           *my_q;
#endif

#ifdef CONFIG_SMP
        /* Per entity load average tracking */
        struct sched_avg        avg;
#endif
};

struct sched_rt_entity {
        struct list_head run_list;
        unsigned long timeout;
        unsigned long watchdog_stamp;
        unsigned int time_slice;

        struct sched_rt_entity *back;
#ifdef CONFIG_RT_GROUP_SCHED
        struct sched_rt_entity  *parent;
        /* rq on which this entity is (to be) queued: */
        struct rt_rq            *rt_rq;
        /* rq "owned" by this entity/group: */
        struct rt_rq            *my_q;
```

-복수 개의 CPU 에서의 런 큐 자료구조와 태스크의 관계

태스크가 처음 생성되면 init_task 를 헤드로 하는 이중연결 리스트에 삽입된다. 모든 태스크들은 연결리스트에 연결되어 있는데 이 중 TASK_RUNNING 상태인 태스크는 시스템에 존재하는 런 큐 중 하나에 소속된다.

CPU	A ○○○
CPU	B ○○
CPU	C ○

1) A 에 프로세스가 많이 쌓이지 않았을 때

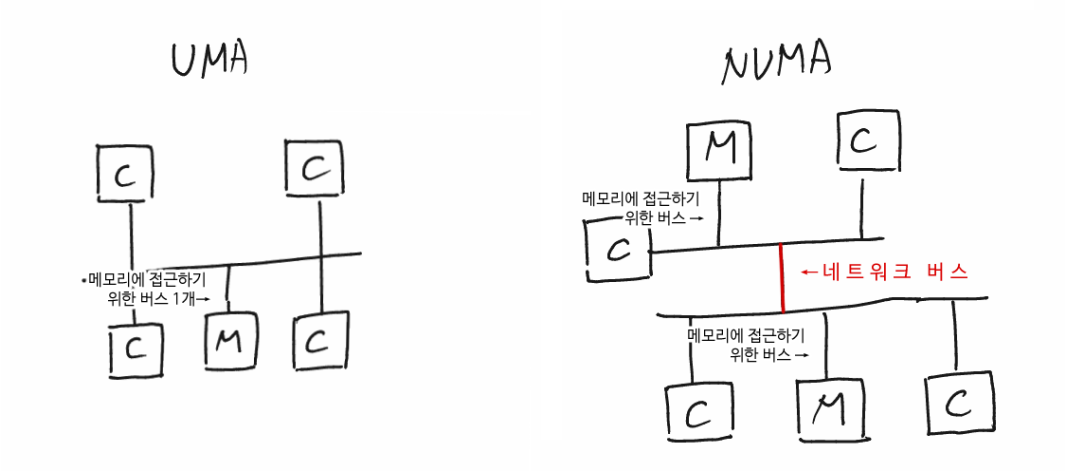
CPU	A ○○○○○○…○○○○○
CPU	B ○○
CPU	C ○

2) A 에 프로세스가 많이 쌓였을 때

1)에서 A 프로세스가 fork()를 하게 되면 A 가 있는 CPU 에서 기존의 캐시를 참조해서 프로세스가 만들어진다. 하지만 2)에서 A 가 fork()를 하게 되면 C 프로세스가 있는 CPU 에 태스크가 만들어지게 된다. 캐시의 용량이 있기 때문에 내부적으로 로드 밸런싱(부하분산)을 하여 조정한다. A 에서 fork 를 하게 된다면 CPU 를 얻기 위해 경쟁을 많이 하게 될 것이고 C 에서 캐시를 참조하지 않고 새로 만드는 것 이 더 효율적이기 때문에 C 에서 생성되게 된다.

*하이퍼 쓰레딩(hyper-threading)

실제 CPU 는 4개이지만 회로로 fork(task_struct)를 구현하여 CPU 가 8개처럼 동작하도록 만든 기술을 하이퍼 쓰레딩이라고 한다. 발열을 줄이며, 공통된 것을 같이 활용하여 로직을 만들지 않아 공간을 더 절약할 수 있다.



- **UMA** - **NUMA**

-

UMA(Uniform Memory Access) = SMP(Symmetric Multi Processor)

: 메모리 접근하는 속도가 모두 같다. 모든 CPU 들이 메모리를 공유한다.

NUMA(Non-Uniform Memory Access)

: 메모리 접근 속도가 다르다. NUMA 에서 메모리 접근 타이밍이 다르다. 필요한 메모리가 반대쪽 메모리에 존재할 때 네트워크 버스를 타고 접근함. 이기종 아키텍처에 사용되는 방식이다.

7-2 실시간 태스크 스케줄링(FIFO, RR and DEADLINE)

스케줄러가 CPU 점유를 관리하는데, 컴퓨터 시스템의 가장 중요한 자원 중의 하나인 CPU 를 공정하게 태스크에게 분배해야 효율적이고 높은 처리율을 낼 수 있다.
task_struct 구조체에 policy, prio, rt_priority 등의 필드가 존재한다.

policy 필드 - 태스크가 어떤 스케줄링 정책을 사용하는지 나타낸다.(SCHED_FIFO, SCHED_RR, SCHED_DEADLINE…)
rt_priority 필드 - 우선순위 설정할 때 사용하고, 0~99까지의 우선순위를 가진다.

```
        int prio, static_prio, normal_prio;
        unsigned int rt_priority;
        const struct sched_class *sched_class;
        struct sched_entity se;
        struct sched_rt_entity rt;
#ifdef CONFIG_CGROUP_SCHED
        struct task_group *sched_task_group;
#endif
        struct sched_dl_entity dl;

#ifdef CONFIG_PREEMPT_NOTIFIERS
        /* list of struct preempt_notifier: */
        struct hlist_head preempt_notifiers;
#endif

#ifdef CONFIG_BLK_DEV_IO_TRACE
        unsigned int btrace_seq;
#endif

        unsigned int policy;
        int nr_cpus_allowed;
        cpumask_t cpus_allowed;
```

실시간 태스크의 스케줄링 정책

1. SCHED_FIFO
 : 순서대로 들어가서 순서대로 나오는 스케줄링 방식이다. 우선순위가 없어 순서대로 처리된다.
2. SCHED_RR

: 태스크가 수행을 종료하거나, 스스로 중지하거나, 자신의 타임 슬라이스를 다 쓸 때 까지 CPU 를 사용하고, 동일 우선순위를 가지는 태스크가 복수개인 경우 타임 슬라이스 기반으로 스케줄링 된다. 만약 동일 우선순위를 가지는 RR 태스크가 없으면 FIFO 와 동일하게 동작한다. → 우선순위를 고려한다 라는 것은 우선시간에 따라 주어지는 시간이 다르다.

3. SCHED_DEADLINE

: EDF(Earliest Deadline First) 알고리즘을 구현 한 것으로 가장 가까운(가장 급한) 태스크를 스케줄링 대상으로 선정한다. 정책을 사용하는 태스크들은 deadline 을 이용하여 RBTree 에 정렬되어 있다. DEADLINE 을 사용하는 태스크의 경우 우선순위는 의미가 없고 주기성을 가지는 프로그램(영상,음성,스트리밍)과 제약시간을 가지는 응용들에 효과적으로 적용 가능하다.

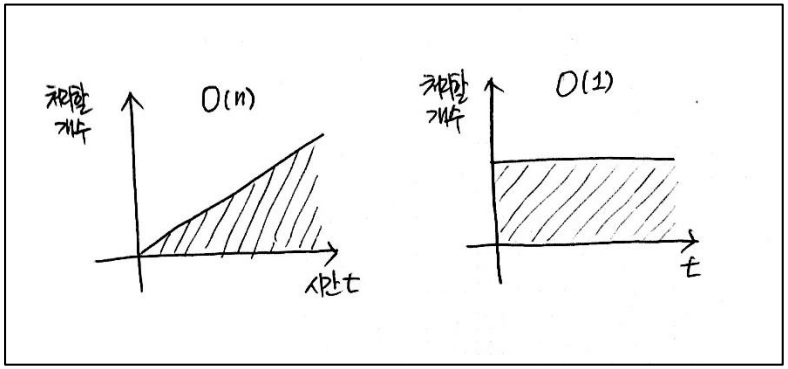
DEADLINE 스케줄링 기법에서 사용하는 자료구조가 담겨있는 struct rt_rq, struct dl_rq

```
/* Real-Time classes' related field in a runqueue: */
struct rt_rq {
        struct rt_prio_array active;
        unsigned int rt_nr_running;
#if defined CONFIG_SMP || defined CONFIG_RT_GROUP_SCHED
        struct {
                int curr; /* highest queued rt task prio */
#ifdef CONFIG_SMP
                int next; /* next highest */
#endif
        } highest_prio;
#endif
#ifdef CONFIG_SMP
        unsigned long rt_nr_migratory;
        unsigned long rt_nr_total;
        int overloaded;
        struct plist_head pushable_tasks;
#ifdef HAVE_RT_PUSH_IPI
        int push_flags;
        int push_cpu;
        struct irq_work push_work;
        raw_spinlock_t push_lock;
#endif
#endif /* CONFIG_SMP */
        int rt_queued;

        int rt_throttled;
        u64 rt_time;
        u64 rt_runtime;
        /* Nests inside the rq lock: */
        raw_spinlock_t rt_runtime_lock;

#ifdef CONFIG_RT_GROUP_SCHED
        unsigned long rt_nr_boosted;

        struct rq *rq;
        struct task_group *tg;
#endif
};
```

```
/* Deadline class' related fields in a runqueue */
struct dl_rq {
        /* runqueue is an rbtree, ordered by deadline */
        struct rb_root rb_root;
        struct rb_node *rb_leftmost;

        unsigned long dl_nr_running;

#ifdef CONFIG_SMP
        /*
         * Deadline values of the currently executing and the
         * earliest ready task on this rq. Caching these facilitates
         * the decision wether or not a ready but not running task
         * should migrate somewhere else.
         */
        struct {
                u64 curr;
                u64 next;
        } earliest_dl;

        unsigned long dl_nr_migratory;
        int overloaded;

        /*
         * Tasks on this rq that can be pushed away. They are kept in
         * an rb-tree, ordered by tasks' deadlines, with caching
         * of the leftmost (earliest deadline) element.
         */
        struct rb_root pushable_dl_tasks_root;
        struct rb_node *pushable_dl_tasks_leftmost;
#else
        struct dl_bw dl_bw;
#endif
};
```

SCHED_FIFO, SCHED_RR 사용할 때의 문제점

:태스크의 개수가 늘어나면 그만큼 스케줄링에 걸리는 시간도 선형적으로 증가하게 되며 (O(n)의 시간복잡도) 스케줄링에 소모되는 시간을 예측할 수 없다.



색칠한 면적은 총 처리해야할 CPU 명령어 갯수이다. 처리해야 할 task 가 적으면 O(n)방식이 좋고 처리해야 할 task 가 많으면 O(1)방식이 좋음.

O(1)로 바꾸는 방법은 Hash, Map 이라는 자료구조를 만들면 O(1)으로 만들 수 있다.

태스크들이 가질 수 있는 모든 우선순위 레벨(0~99)을 표현할 수 있는 비트맵과, 태스크가 생성되면 그 태스크의 우선순위에 해당하는 비트를 1로 set 한 뒤, 태스크의 우선순위에 해당되는 큐에 삽입된다.

태스크를 생성하고 1로 세팅하는 이유는 &연산하면 1이 있는 것만 확인이 되고 없으면 0이 나오게 되고, queue 를 사용하는 이유는 같은 우선순위를 가진 task 들이 존재할 수 있기 때문에 우선순위에 따른 task 들을 queue 로 넣어 리스트로 관리한다.

이렇게 해서 bitmap 에 가장 처음으로 set 되어있는 비트가 가장 우선순위가 높은 것 이므로 그 우선순위 큐에 매달려 있는 태스크를 선택하여 for 문이나 while 문을 사용하지 않고도 스케줄링 작업이 고정시간 내에 완료하여 O(1)방식으로 처리할 수 있다.

위의 설명에서 bitmap 에 매핑하는 것 까지가 hash 자료구조이다.

7-3 일반 태스크 스케줄링(CFS)

리눅스가 일반 태스크를 위해 사용하고 있는 스케줄링 기법은 CFS(Completely Fair Schedluer) 태스크 A 와 B 가 있을 때 '가상' 사용시간이 항상 1:1로 같아야 한다는 것이다. (실제시간이 아님) 1초를 시간단위로 한다면 A 에 가상시간을 0.5초, B 에게 0.5초동안 CPU 사용시간을 주는 것이다. 시간단위가 너무 길면 태스크의 반응성이 떨어지고 시간단위가 너무 짧다면 context switching 을 많이 해야 하므로 저장 해야 할 정보가 너무 많아져 오버헤드가 높아지므로 적절한 값을 설정해야 한다.
CFS 는 태스크마다 우선순위가 높은 태스크에 가중치를 두어 좀 더 긴 시간동안 CPU 를 사용할 수 있도록 하는데 가상 사용시간은 1:1로 같아야 하기 때문에 vruntime 이라는 개념을 도입한다.

vruntime

: 각 태스크는 자신만의 vruntime 값을 가지며 이 값은 스케줄링되어 CPU 를 사용하는 경우 사용시간과 우선순위를 고려하여 증가된다. 일반 태스크는 사용자 수준에서 볼 때 -20~0~19사이의 우선순위를 가지며, 커널 내부적으로는 priority+120으로 변환된다. 일반 태스크의 실제 우선순위는 100~139에 해당한다.
우선순위가 높으면 시간이 천천히 흘러야 하므로 가중치가 낮아야 하고 우선순위가 낮으면 시간이 빠르게 흘러야 하므로 가중치가 높아야한다.

$$\text{vruntime} \mathrel{+}= \text{physicalruntime(실제 구동시간)} * \frac{\text{weight}_0\ (0\text{번의 가중치})}{\text{weight}_{curr}(\text{현재의 가중치})}$$ 여기에 수식을 입력하세요.

스케줄링 대상이 되는 태스크를 빠르게 고르기 위해 가장 작은 vruntime 값을 가지는 태스크를 선정하는 방식으로 한다. vruntime 이 가장 작다는 것은 가장 과거에 CPU 를 사용했음을 의미하기 때문이다.

너무 자주 스케줄링이 발생되지 않게 리눅스는 각 태스크별로 선점되지 않고 CPU 를 사용할 수 있는 타임 슬라이스가 상수로 미리 지정되어 있다.

스케줄러는 직접적으로 schedule()함수를 호출할 때, 현재 수행되고 있는 태스크의 thread_info 구조체 내부에 존재하는 flags 필드 중 need_resched 라는 필드를 설정할 때 호출된다.

태스크를 만든 비율이 너무 많이 차이나는 경우 CPU 를 사용하는 비율도 차이가 많이 나기 때문에 그룹 스케줄링 정책을 지원한다. vruntime 을 번갈아가며 실행하고 vruntime 은 같아야 하는 정책이다.

(9) 태스크와 시그널

시그널은 태스크에게 비동기적인 사건의 발생을 알리는 메커니즘이다.

태스크가 시그널을 원활히 처리하기 위한 3가지 기능

1.다른 태스크에게 시그널을 보낼 수 있어야함.
리눅스는 이를 위해 sys_kill()이라는 시스템 호출을 제공.
ex) kill -번호 pid

2.자신에게 시그널이 오면 그 시그널을 수신할 수 있어야한다.
이를 위해 task_struct 에 signal, pending 이라는 변수 존재

3.자신에게 시그널이 오면 그 시그널을 처리할 수 있는 함수를 지정할 수 있어야한다.
task_struct 내에 sighand 라는 변수 존재

-특정 PID 를 가진 태스크를 종료할 때
:시그널 체인을 만들어 같은 tgid 를 가진 태스크들을 한번에 정리한다.

-특정 태스크에게만 시그널을 보내야 할 때
:pending 필드에 저장하여 특정 태스크에게만 시그널을 보낸다.

-특정 시그널이 발생했을 때
: sighand 필드에 시그널 핸들러를 설정
: blocked 필드에 특정 시그널을 받지 않도록 설정, SIGKILL 과 SIGSTOP 은 무시할 수 없다.

다른 태스크에 시그널을 보내는 과정
해당 태스크의 task_struct 찾기 > 시그널 번호를 통해 siginfo 자료구조 초기화 > signal(같은 tgid)이나 pending(특정 task)필드에 매달아주기. 이때 blocked 필드 검사하여 시그널 받지 않도록 지정했는지 검사
수신한 시그널의 처리는 system call 을 다 처리하고 사용자로 넘어올 때 처리한다. pending 필드의 비트맵이 켜져있는지
signal 필드의 count 가 0이아닌지 검사를 통해 처리를 대기중인 시그널이 있는지 확인하고 블록 되어있지 않다면 시그널 번호에 해당되는 시그널 핸들러를 sighand 필드의 action 배열에 찾아서 수행시킨다. 핸들러를 등록하지 않았을 땐 디폴드 액션을 취함.

인터럽트 트랩 VS 시그널
인터럽트와 트랩이 사건의 발생을 커널에 알리는 방법이면 시그널은 사건의 발생을 태스크에 알리는 방법이다.

Chapter 4 메모리 관리

(1) 메모리 관리 기법과 가상 메모리

가상 메모리 : 물리 메모리의 한계를 극복하기 위해 가상 메모리(virtual memory) 사용. 실제 시스템에 존재하는 물리 메모리의 크기와 관계없이 가상적인 주소 공간을 사용자 태스크에게 제공한다. 32Bit CPU 에서는 각 태스크마다 4GB 의 공간을 가지는데 이는 가상의 공간이고 물리메모리는 필요한 만큼의 메모리 만 사용된다.

장점

-개념적으로 4GB 의 공간을 제공하는 것이고 물리메모리는 필요한 만큼만 사용되어서 많은 태스크가 동시에 수행되는 장점을 가진다.
-가상메모리를 사용하여 메모리 배치 정책이 불필요하다. 물리메모리는 배치정책이 필요.
-태스크간 메모리 공유와 보호가 쉽고 태스크의 빠른 생성이 가능하다. task_struct 복사하기만 하면 끝나기 때문에 빠른 생성이 가능하다.

(2) 물리 메모리 관리 자료 구조

리눅스는 시스템에 존재하는 전체 물리 메모리에 대한 정보를 가지고 있어한다.

SMP(Symmetric Multiprocessing) : 모든 CPU 가 메모리와 입 출력 버스 등을 공유하는 구조. 이러한 구조를 UMA 라 함.

UMA : 복수개의 CPU 가 메모리 자원을 공유하는 구조. 성능상 병목 현상이 발생한다.
NUMA : CPU 들을 몇 개의 그룹으로 나누고 각각의 그룹에게 별도의 지역 메모리를 주는 구조.

2-1 Node

뱅크(bank)

: 리눅스에선 접근 속도가 같은 메모리의 집합을 뱅크라 한다.

NUMA 구조에서는 2개 이상의 복수개의 뱅크가 존재하고, 뱅크를 표현하는 구조가 노드이다.(contig_page_data 로 node 에 접근 가능)

```
struct zone {
        /* Read-mostly fields */

        /* zone watermarks, access with *_wmark_pages(zone) macros */
        unsigned long watermark[NR_WMARK];

        unsigned long nr_reserved_highatomic;

        /*
         * We don't know if the memory that we're going to allocate will be
         * freeable or/and it will be released eventually, so to avoid totally
         * wasting several GB of ram we must reserve some of the lower zone
         * memory (otherwise we risk to run OOM on the lower zones despite
         * there being tons of freeable ram on the higher zones).  This array is
         * recalculated at runtime if the sysctl_lowmem_reserve_ratio sysctl
         * changes.
         */
        long lowmem_reserve[MAX_NR_ZONES];

#ifdef CONFIG_NUMA
        int node;
#endif

        /*
         * The target ratio of ACTIVE_ANON to INACTIVE_ANON pages on
         * this zone's LRU.  Maintained by the pageout code.
         */
        unsigned int inactive_ratio;

        struct pglist_data      *zone_pgdat;
        struct per_cpu_pageset __percpu *pageset;

        /*
         * This is a per-zone reserve of pages that should not be
         * considered dirtyable memory.
         */
        unsigned long           dirty_balance_reserve;
```

```
#ifndef CONFIG_NEED_MULTIPLE_NODES
struct pglist_data __refdata contig_page_data = {
        .bdata = &bootmem_node_data[0]
};
```

->UMA

하나의 노드는 pg_data_t 구조체를 통해 표현. pg_data_t 는 해당 노드에 속해있는 물리 메모리의 실제 양(node_present_pages)이나, 해당 물리메모리가 메모리 맵의 몇번지에 위치하고 있는지를 나타내는 변수(node_start_pfn)등이 정의 되어있다.

만약 리눅스가 물리 메모리의 할당 요청을 받게 되면 되도록 할당을 요청한 태스크가 수행되고 있는 CPU 와 가까운 노드에서 메모리를 할당하려 한다. → 캐시를 활용한다.