Xilinx Zynq FPGA, TI DSP, MCU 기반의 프로그래밍 전문가 과정

<리눅스 커널>

2018.04.10 - 34 일차

강사 - 이상훈

gcccompil3r@gmail.com

학생 - 안상재

sangjae2015@naver.com

<Chapter 3 태스크 관리>

7-2 실시간 태스크 스케줄링
- task_struct 구조체의 rt_priority 를 사용해서 우선순위를 설정함. (0~99 까지의 우선순위)
- 우선순위에 따라 할당되는 시간이 다름.
- tasklist 라는 이중 연결 리스트를 이용해 태스크에 접근해서 우선순위가 가장 높은 태스크를 골라낸다.

* HASH 자료구조 (복잡도 O(N)을 O(1)로 바꾸어 줌)
→ 우선순위 레벨(0~99)을 표현할 수 있는 비트맵에서 태스크의 우선순위에 해당하는 비트를 1 로 set 함.
→ 스케줄링 하는 시점이 되면 커널은 비트맵에서 가장 처음으로 set(우선 순위 가장 높은 태스크) 되어 있는 비트를 찾아 낸 뒤, 그 우선순위 큐에 있는 태스크를 선택함.

* DEADLINE 정책 (정적 우선순위, 실시간 방식의 스케줄링)
- deadline 이 가장 가까운 태스크를 스케줄링 대상으로 선정함.
- 각 태스크들은 deadline 을 이용하여 RBTRee 에 정렬되어 있음.
- 스케줄러가 호출되면 가장 가까운 deadline 을 가지는 태스크를 스케줄링 대상으로 선정
→ DEADLINE 정책을 사용하면 태스크의 실행 순서가 deadline 시간에 의해 결정되므로 우선순위는 의미가 없다.
* DEADLINE 스케줄링 기법의 자료구조
- struct rt_rq : 실시간 전용 런큐
- struct dl_rq : deadline 에 해당하는 런큐

7-3 일반 태스크 스케줄링 (CFS)
- 모든 태스크들에게 CPU 사용시간의 공정한 분배
1) 정해진 시간단위를 기준으로 모든 태스크들에게 동일한 CPU 시간을 할당함.
- 시간 단위가 너무 길면 태스크의 반응성이 떨어짐. ex) 키보드 반응속도
- 시간 단위가 너무 짧다면 잦은 문맥교환으로 인해 오버헤드가 높아짐.

2) 우선순위가 높은 태스크에게 가중치를 두어 더 긴 시간의 CPU 를 사용할 수 있게 해줌.
- 일반 태스크의 사용자 수준의 우선순위 : -20~0~19
- 커널 내부적으로 120 을 더해서 실제 우선순위는 100~139 에 해당됨. (실시간 태스크의 우선순위보다 낮은 것이 보장됨)
- vruntime += physicalruntime * (0 순위의 가중치 / 현재 태스크 순위의 가중치)
→ 우선순위가 높은 태스크의 경우, 시간이 느리게 흘러가는 것처럼 관리하고, 우선순위가 낮은 태스크의 경우 시간이 빠르게 흘러가는 것처럼 관리함.

* 언제 스케줄러가 호출되는가?
- nice_sys 의 시스템 콜에 의해 스케줄 함수 호출
- thread_info 구조체 내부에 존재하는 flags 의 상태가 1 로 셋팅 될때

9. 태스크와 시그널

2) signal 공유
- 모든 태스크들에게 시그널을 보낼 때 : task_struct 구조체의 signal 필드에 시그널을 저장하고 sys_kill() 과 같은 시스템 콜을 호출함.
- 특정 태스크에게만 시그널을 보내야 할 때 : 공유하지 않는 시그널은 task_struct 구조체의 pending 필드에 저장해둠.

3) 시그널 핸들러
- 태스크에서 지정한 핸들러는 task_struct 구조체의 sighand 필드에 저장됨.
- 시그널을 받지 않기 위해서는 task_struct 구조체의 blocked 필드를 통해 이루어짐.
- SIGKILL 과 SIGSTOP 시그널은 무시 불가능함.

4) 수신한 시그널의 처리는 태스크가 커널 수준에서 (시스템 콜을 처리한 후) 사용자 수준 실행 상태로 전이할 때 이루어짐.

* 인터럽트, 트랙과 시그널 간의 차이점
1) 인터럽트, 트랩 : 어떤 사건의 발생을 커널에게 알림.
2) 시그널 : 사건의 발생을 태스크에게 알림.

<Chapter 4 메모리 관리>

1. 메모리 관리 기법과 가상 메모리

1) 가상메모리
- 물리 메모리의 한계를 극복하기 위해 개발
- 32 비트 CPU → 4GB 가상 주소 공간 , 64 비트 CPU → 16EB 의 주소 공간
- 4GB 의 공간을 프로그래머에게 제공하고, 물리 메모리는 필요한 만큼의 메모리만 사용됨.
- 컴파일러는 무조건 가상메모리에 모두 저장함 → 커널영역의 sys_exec() 에 의해 물리 메모리에 적재됨.

2. 물리 메모리 관리 자료 구조
1) 리눅스는 부팅 시 시스템에 존재하는 전체 물리 메모리에 대한 정보를 저장함.
2) CPU 와 메모리 공유 구조
- UMA : cpu 의 메모리에 대한 접근 속도가 모두 동일함.
- NUMA : cpu 의 메모리에 대한 접근 속도가 cpu 에 따라 다름.

2-1 Node
1) 뱅크 : 접근 속도가 같은 메모리의 집합
- UMA : 한 개의 뱅크 존재, 한 개의 노드 존재
- NUMA : 복수 개의 뱅크 존재, 복수 개의 노드 존재
- 노드 : 리눅스에서 뱅크를 표현하는 구조체, 리눅스의 전역 변수인 contig_page_data 를 통해 접근 가능
- 복수 개의 노드는 pg_data_t 라는 배열의 리스트를 통해 관리됨.

2) 캐시 메모리 활용
- 리눅스가 물리 메모리의 할당 요청을 받게 되면, 되도록 할당을 요청한 태스크가 수행되고 있는 CPU 와 가까운 노드에서 메모리를 할당하려 함 → 높은 성능을 얻게 됨.

2-2 ZONE
1) zone 의 의미 : 메모리의 영역
- DMA 용도
- DMA 32 용도
- 일반적인 메모리 공간

2) 노드의 물리 메모리 영역
- 노드에 존재하는 물리 메모리 중 16MB 이하 부분은 zone(ZONE_DMA, ZONE_DMA32)
- ZONE_DMA 에 속하지 않는 16MB 이상의 메모리 : ZONE_NORMAL

3) ZONE_HIGHMEM
- 리눅스의 가상 주소 공간을 물리 메모리 공간과 1:1 크기로 맵핑을 시켜버리면 물리 메모리의 남는 공간은 사용할 수 없게 된다. 물리 메모리가 가상 메모리보다 월등히 클 때 이것은 치명적인 약점이 된다.
→ 32bit cpu(리눅스 가상 메모리 1GB)에서 물리 메모리가 1GB 이상이라면 896MB 까지를 커널의 가상 주소 공간과 1:1 로 연결해주고, 나머지 부분은 필요할 때 동적으로 연결한다.

* zone 구조체의 멤버
- watermark : 32bit cpu 에서 896MB 넘어가면 메모리 해제하기 위한 알림 역할을 함.
- vm_stat : 통계적으로 메모리의 할당/해제가 되는 빈도를 보고 watermark 값을 정함.