TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사 - 이상훈
gcccompil3r@gmail.com

학생 – 이우석
colre99@naver.com

[4/11(수) – 35일차]

[chaper 3] (지난주)

[런큐(Run queue) & 스케줄링(Scheduling)]

-**스케줄링**: 여러개의 태스크들 중에서 다음번 수행시킬 태스크를 선택하여 **CPU** 라는 자원을 할당하는 과정
(다중 프로그래밍을 가능하게 하는 운영체제의 동작기법. 자원배정을 적절히 함으로써 시스템 성능 개선 가능)

- **런큐**: 일반적인 운영체제는 스케줄링 작업 수행을 위해, 수행가능한 상태의 태스크를 자료구조를 통해 관리.
리눅스 에서는 이 자료구조를 런큐(Run queue)라고 한다.

* 리눅스의 태스크는 실시간 태스크 & 일반 태스크 로 나뉜다. (각각을 위해 별도의 스케줄링 알고리즘이 구현되어 있음)

[리눅스가 제공하는 140 단계의 우선순위]

실시간 태스크 : 0 부터 99 단계 사용. (숫자가 낮을수록 우선순위 나타냄)
일반 태스크: 100 부터 139 단계 사용.
→ !! 실시간 태스크는 **항상** 일반 태스크 보다 **우선**하여 실행된다. !!

-실시간 태스크 스케줄링 (FIFO, RR, & DEADLINE)-

:

task_struct 구조체는 policy, prio, rt, priority 등의 필드가 존재.

Policy 필드는 이 태스크가 어떤 스케줄링 정책을 사용하는지를 나타낸다.

리눅스의 태스크는 총 6 개의 정책이 존재. (**실시간** 태스크를 위해 **3** 개 + **일반** 태스크를 위해 **3** 개)

*** 실시간 태스크**를 위해서는 3 개의 정책: SCHED_FIFO, SCHED_RR, SCHED_DEADLINE 정책이 사용된다

*** 일반 태스크를** 위해서도 3 개의 정책: SCHED_NORMAL, SCHED_IDLE, SCHED_BATCH 정책이 사용된다.

* **부가설명:**

<**실시간**>

SCHED_FIFO: first in first out(선입선출). Priority 값을 1 부터 99 까지 가질수 있고, sched_yield()호출 하거나, runnable 한 상태가 되면 priority 에 해당하는 대기 리스트의 맨뒤, 자신보다 더 높은 priority 를 가진 스레드에 의해서만 선점되고, 선점한 스레드가 CPU 를 놓아주면 계속 동작하는 정책.

SCHED_RR: 우선, 라운드 로빈(round robin)은 동등한 우선순위를 가진 스레드에게 동등한 간격으로 시간을 주는것을 말하고, 이 스케줄링 정책은 우선순위에 관계없이 일정한 시간을 할당해서 하는 정책을 말함.

SCHED_DEADLINE: 마감시간이 있는 스케줄링 정책으로 우선순위를 먼저 배정해서 실행하고 후속작업은 나중에 최소 시간으로 실행하게 하여 시간내에 동작시키도록 하는 정책.

<**일반**>

SCHED_NORMAL: 스레드는 priority 값을 0 만 가질수 있고, time share 하면서 동등하게 스케줄링 하는 정책.

SCHED_IDLE: 중요하지 않은 일을 수행하는 태스크가 CPU 를 점유 하는것을 막기위해 가장낮은 우선순위로 스케줄링 되게한다.

SCHED_BATCH: 사용자와의 상호 작용이 없는 CPU 중심의 일괄 작업(batch job) 태스크를 위한 정책.

*sched_setscheduler()등의 함수를 통해 태스크의 스케줄링 정책을 위 세가지중 하나로 바꾸게 되면 실시간 태스크.
*실시간 태스크는 우선순위 설정을 위해 task_struct 구조체의 rt_priority 필드를 사용. (rt_priority 는 0 부터 99 까지의 우선수위를 가짐)
  태스크가 수행을 종료하거나, 스스로 중지하거나, 혹은 자신의 타임 슬라이스를 다 쓸때까지(이 경우는 sched_RR 정책만 해당된다)
  CPU 를 사용한다.

즉, RR 인 경우 동일 우선순위를 가지는 태스크가 복수개인 경우 타임 슬라이스 기반으로 스케줄링 된다. 만약, 동일 우선순위를 가지는 RR 태스크가 없는 경우라면 FIFO 와 동일하게 동작된다. 또한 실시간 정책을 사용하는 태스크는 고정 우선순위를 가지게 된다.
따라서, 우선순위각 높은 태스크가 낮은 태스크 보다 먼저 수행된다는 것을 보장한다.

시스템에 존재하는 모든 태스크는 tasklist 라는 이중연결 리스트에 연결되어 있으므로, 이 리스트를 이용하면 시스템 내의 모든 태스크를 접근하는 것이 가능하다.

*스케줄링 방식에는 O(n) 와 O(1)의 시간복잡도 가 있다. (장기적으로는 O(1)이 더 낫다) – 그림설명 첨부필요.
결론만 말하자면, 분량이 적을때 O(n)이 빠르겠지만, 분량이 많아진다면 시간이 지남에 따라 O(1)이 처리가 더 빠르다.
* 리눅스의 DEADLINE 정책에서는 완료시간 = deadline, 작업량 = runtime, 초당 30 회라는 주기성 = period.
정상적인 동작을 위해서는 태스크의 runtime(작업량) 과 deadline(완료시간)은 (현재시간 + runtime < deadline)의 조건을 만족해야 한다.
단, DEADLINE 정책을 사용하는 태스크들의 runtime 합은 CPU 의 처리량을 넘어서는 안된다. 이는 결국, 새로운 태스크가 DEADLINE

정책을 사용하는 기존 태스크들의 runtime(작업량) 과 period(주기성)를 이용해 해당 태스크의 성공적인 완료 여부를 확정적(deterministic)으로 결정 할 수 있다는 의미. 이러한 확정성은 실시간 스케줄링 기법의 가장 중요한 요소 중 하나.

CFS(일반 태스크 스케줄링)

:리눅스가 일반 태스크를 위해 사용하고 있는 스케줄링 기법.
이 기법은 공정한 스케줄링을 추구하기에 CPU 사용시간의 공정한 분배를 의미한다.
예를들어, A ,B 두개의 태스크가 수행중이라면, A 와 B 의 CPU 사용시간이 항상 1 대 1 로 같아야 한다는 것이다.
하지만 실제로는 시간상 차이가 난다. 가상의 시간으로는 1 대 1 이지만, 실제 부여받은 시간은 우선순위에 따라 다르다.
ex) A 가 B 보다 우선순위가 높다면 A 에게 할당되는 시간은 B 보다 높다.

이를 위해 리눅스는 vruntime 개념을 도입했다. 각 태스크는 자신만의 vruntime 값을 가지며, 이 값은 스케줄링 되어 CPU 를 사용하는 경우 사용시간 과 우선순위 를 고려하여 증가된다. 일반 태스크의 우선순위는 별도의 지정(nice() 같은 시스템 호출) 이 없다면 부모의 우선순위와 같다. 일반 태스크는 사용자 수준에서 볼때 -20 ~ 0 ~ 19 사이의 우선수위를 갖게 되며, 이 값은 커널 내부적으로 (priority + 120) 으로 변환된다. 따라서 일반 태스크의 실제 우선순위는 100 ~ 139 에 해당되며, 항상 실시간 태스크의 우선순위 보다 낮은것이 보장된다.

리눅스는 주기적으로 발생되는 타이머 인터럽트 핸들러 에서 scheduler_tick() 함수를 호출함 으로써 현재 수행중인 태스크의 vruntime 값을 갱신한다. 이때 vruntime 값의 증가분 식은 vruntime += physicalruntime * weight0/weight(curr).

리눅스는 가장 낮은 vruntime 값을 가지는 태스크를 빠르게 찾아내기 위해 Rbtree 자료구조를 사용한다.(RED BLACK)
각 태스크는 vruntime 값을 키로 하여 RBtree 에 정렬되어 있으며, 이 트리에서 가장 좌측에 존재하는 태스크(vruntime 값이 가장 작은 태스크)가 다음번 스케줄링의 대상이 된다. 스케줄링된 태스크는 수행 될수록 값이 증가되며 따라서 트리의 가장 좌측에서 점차 우측으로 이동된다. 반면, 스케줄링 되지않은 태스크는 대기하는 동안 점점 자신의 키 값이 (상대적으로) 감소되며 점차 트리의 좌측으로 이동하게 된다.

리눅스는 시간단위 를 태스크의 우선순위, 즉, 가중치에 기반하여 각 태스크에게 분배한다.
계산하는 과정은 커널의 sched_slice() 함수에 구현되어 있다. 이때, 시간단위는 너무잦은 스케줄링으로 인한 오버헤드를 최소화 하기위해 시스템에 존재하는 태스크의 개수를 고려하여 정해지며, 커널의 __sched_period()함수에서 계산된다.

*스케줄러의 호출방법엔 2 가지가 있다.

직접적으로는 schedule()함수를 호출하는 방법이 있고, 현재 수행되고 있는 태스크의 thread_info 구조체 내부에 존재하는 flags 필드 중 need_resched 라는 필드를 설정하는 방법이다.

*CFS 는 불공정한 상황을 해결하기 위해 그룹 스케줄링 정책을 지원한다.

[태스크(task) & 시그널(signal)]

시그널은 태스크에게 비동기적인 사건의 발생을 알리는 메커니즘이다.
태스크가 시그널을 원활히 처리하려면 **3 가지 기능**을 지원해야한다

첫째, 다른 태스크에게 시그널을 보낼수 있어야한다. 이를 위해 리눅스커널은 sys_kill() 이라는 시스템 호출을 제공한다.
둘째, 자신에게 시그널이 오면 그 시그널을 수신할 수 있어야 한다. 이를 위해 task_struct 에는 signal, pending 이라는 변수가 존재.
셋째, 자신에게 시그널이 오면 그 시그널을 처리할 수 있는 함수를 지정할 수 있어야 한다. 이를 위해 sys_signal()이라는 시스템 호출이 존재.
task_struct 내에 sighand 라는 변수가 존재.

< 리눅스는 시그널을 2 가지로 구분 >

- 기본적으로 지원하는 일반 시그널 32 개
- POSIX 표준을 위해 도입한 실시간 시그널 32 개
(실제 리눅스 커널에서는 실시간 시그널이 사용 되지는 않는다.)

* 리눅스에서 PID 는 tgid 를 의미. 따라서 PID 가 같은 태스크들은 의미상 같은 스레드 그룹을 의미.

*여러태스크들 간에 공유해야 하는 시그널이 도착하게 되면 이를 task_struct 구조체의 signal 필드에 저장하고, 이러한 시그널을 보내는 작업은 sys_kill()과 같은 시스템 호출을 통해 이뤄진다.

반대로, 특정 태스크에게만 시그널을 보내야 하는경우, 공유하지 않는 시그널은 task_struct 구조체의 pending 필드에 저장해 둔다.
시그널을 signal 필드나 pending 필드에 저장할 때는 시그널 번호 등을 구조체로 정의하여 큐(queue)에 등록시키는 구조를 택하고 있으며,
이를 위해 sys_tkill() 과 같은 시스템 호출을 도입.

각 태스크는 특정 시그널이 발생했을때 수행될, 시그널 함수를 지정할 수 있다. 이때 사용자 지정 시그널 핸들러를 설정하게 해주는 함수가 sys_signal()이다. 태스크가 지정한 시그널 핸들러는 task_struct 구조체의 sighand 필드에 저장된다. 또한, 태스크는 특정 시그널을 받지 않도록 설정할 수 있는데 이는 task_struct 구조체의 blocked 필드를 통해 이뤄진다. 태스크에서 sigprocmask() 와 같은 함수를 사용하면 인자로 넘긴 시그널 번호에 해당되는 비트를 blocked 필드에서 설정함으로써 특정 시그널을 받지 않도록 할 수 있다. 그런데, 시그널 중에 SIGKILL 과 SIGSTOP 이라는 시그널은 받지 않도록 설정 하거나, 무시할 수 없다.

수신한 시그널의 처리는 태스크가 커널 수준에서 사용자 수준 실행 상태로 전이 할 때 이루어진다.
즉, 커널은 pending 필드의 비트맵이 켜져 있는지, 혹은 signal 필드의 count 가 0 이 아닌지 검사를 통해 처리를 대기 중인 시그널이 있는지 확인할 수 있다. 만약, 태스크가 명시적으로 핸들러를 등록하지 않은 경우 커널은 시그널 무시, 태스크 종료, 태스크 중지 등과 같은 디폴드 액션을 취하게 된다.

끝으로, 사건의 발생을 커널에게 알리는 방법은 인터럽트 와 트랩. 태스크에게 알리는 방법은 시그널 이다.

[Chapter 4] – 메모리 관리 (배운거 정리)

: 시스템의 모든 물리 메모리를 효율적으로 관리하기 위한 구조와 물리 메모리를 할당/해제 하는 기법을 알아본다.

물리 메모리의 한계를 **극복**하기 위해 가상 메모리 사용(virtual memory).

가상 메모리는 물리 메모리의 크기와 관계없이 가상적인 주소 공간을 사용자 태스크에게 제공한다.
(페이징을 통해서 뒤에서 관리한다.)

따라서 32bit CPU 의 경우 주소 지정할 수 있는 최대 크기인 2^32 크기(4GB)의 가상 주소 공간을 사용자에게 제공.
64bit CPU 의 경우 2^64 크기(16EB) 의 주소 공간을 사용자에게 제공.
(32bit CPU 에서는 각 태스크 마다 4GB 의 공간을 가지고 있다)

주의할 점은, 물리적으로 4GB 의 메모리를 모두 사용자 태스크에게 제공하는 것은 아니다. 4GB 라는 공간은 프로그래머에게 개념적으로 제공되는 공간(=가상 주소 공간)이며, 실제로는 사용자가 필요한 만큼의 물리 메모리를 제공한다.

결국, 가상 메모리는 사용자에게 개념적으로 4GB 의 큰 공간을 제공함과 동시에 물리 메모리는 필요한 만큼의 메모리만 사용되므로 가능한 많은 태스크가 동시에 수행될 수 있다는 장점을 제공한다. 이외에도 메모리 배치 정책이 불필요하며 태스크 간 메모로 공유/보호가 손쉽고, 태스크의 빠른 생성이 가능하다는 장점을 가진다.

[물리 메모리 관리 자료구조]

리눅스는 시스템에 존재하는 전체 물리 메모리에 대한 정보를 가지고 있어야 한다. (= 부팅시 항상 잡아준다)
한정된 용량의 메모리를 효율적으로 사용하기 위해 몇몇 정책(ex: Buddy, Slab, paging, UMA, NUMA 등)들이 있다.

SMP(Symmetric Multi processor)
:모든 CPU 가 메모리와 입출력 버스 등을 공유하는 구조
(복수 개의 CPU 가 메모리 등의 자원을 공유하기 때문에 성능상 병목현상이 발생할 수 있다.)

NUMA(Non-Uniform Memory Access)
:CPU 들을 몇개의 그룹으로 나누고 각각의 그룹에게 별도의 지역 메모리를 주는 구조 (병목현상 해결위해 사용(SMP))

*기존 시스템은 UMA(Uniform Memory Access)

* NUMA 구조에서는 CPU 에서 어떤 메모리에 접근하는냐에 따라 성능의 차이가 생길수 있다.
예를들어, CPU 는 자신에게 가까운 곳에 위치하고 있는 메모리를 주로 사용해야 다른 버스에 있는 메모리를 접근할 때,
보다 빠르게 데이터를 읽어올 수 있다. 즉, 각각의 구조에 적합한 메모리 정책을 사용해야 한다는 것.

[2-1] Node

뱅크(bank)
: 리눅스에서 접근속도가 같은 메모리의 집합

노드(node)
: 리눅스에서 뱅크를 표현하는 구조

* 리눅스는 하드웨어 시스템에 관계없이 노드라는 일관된 자료구조를 통해서 전체 물리 메모리를 접근할 수 있게 되는 것. 복수 개의 노드는 리스트를 통해 관리된다. 이는 pgdat_list 라는 이름의 배열을 통해 접근가능 하다.

하나의 노드는 pg_data_t 구조체를 통해 표현된다. 이 구조체는 해당노드에 속해있는 물리 메모리의 실제양 (node_present_pages)이나, 해당 물리메모리가 메모리 맵의 몇번지에 위치하고 있는지를 나타내기 위한 변수 (node_start_pfn) 등이 정의되어 있다.

Cache 를 활용하면 높은 성능을 얻을 수 있다.
→ 가까운 곳의 위치한 메모리를 사용

[2-2] Zone

node 를 관리하는 zone.(노드에 존재하는 물리메모리 중에) (node 의 일부분을 따로 관리할 수 있도록 자료구조를 만듦)

ex) ZONE_DMA, ZONE_DMA32
(DMA: Direct Memory Access) 직접 메모리 접근.
:주변장치들이 메모리에 직접 접근하여 읽거나 쓸수 있도록 하는 기능으로, 컴퓨터 내부의 버스가 지원하는 기능.

-ZONE_HIGHMEM-
:커널의 가상주소공간 과 1:1 로 연결해주고, 나머지 부분은 필요할때 동적으로 연결하여 사용하는 구조.

(물리메모리가 1GB 이상이라면, 896MB 까지만)

여기에서 커널의 가상주소공간과 1:1 로 연결 에서 물리메모리에 접근하려면 먼저 페이징을 해줘야 한다.

그리고, 나머지 부분은 필요할때 동적으로 연결. 이부분은 간접참조. (주소를 참조해서 매칭)

[2-3] Page frame

3. Buddy 와 Slab

[3-1] 버디 할당자 (Buddy Allocator)

nptl 핵심:
커널전용 프로그램 , 사용자의 프로그램 이 1 대 1. 프로그램 사이즈크면 캐시가 깨짐 = 성능저하, 메모리 올리기힘듬.
캐시활용 저하. 기존의 1 대 n 모델을 1 대 1 모델로 바꾸고, 프로그램 사이즈에 제한을 둔다. 커널스택이란걸 만들어서
유저랑 통신할수 있도록 커널스택에서 활용. 커널스택은 스레드_유니온 이 커널스택. 즉 nptl 의 성능이 빨라졌다.

Deadline 은 정적 / 사용자 스케줄링은 동적. 실제시간은 달라도, 가상시간은 같아야 한다.

[3-2] Lazy Buddy

[3-3] 슬랩 할당자 (Slab Allocator)

[가상 메모리 관리 기법]