TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사 : Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 : 황수정
sue100012@naver.com
34 일차 (2018. 04. 10)

목차

Chapter 3 : 태스크 관리

7) 런 큐와 스케줄링

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

컴퓨터 시스템의 가장 중요한 자원 중 하나인 CPU 를 어떤 태스크가 사용하도록 해줄 것인가?

스케줄러가 점유율을 관리한다. 이 때, **공정**하게 태스크에게 분배해주어야, **효율적이여야 가장 높은 처리율**을 낼 것이다. **가장 급한 태스크(=우선순위가 높은 태스크)**를 한가한 태스크보다 먼저 수행하도록 하면, 높은 반응성을 보일 것이다.

task_struct 구조체는 policy, prio, rt_priority 등의 필드(멤버 변수)가 존재한다.

policy 필드 : 태스크가 어떤 스케줄링 정책을 사용하는지 나타낸다.

rt_priority 필드 : 우선순위를 설정할 때 사용하고, 0~99 까지의 우선순위를 가진다.

실시간 태스크 : SCHED_FIFO, SCHED_RR, SCHED_DEADLINE

우선순위 설정을 위해 rt_priority 필드를 사용한다.

SCHED_FIFO

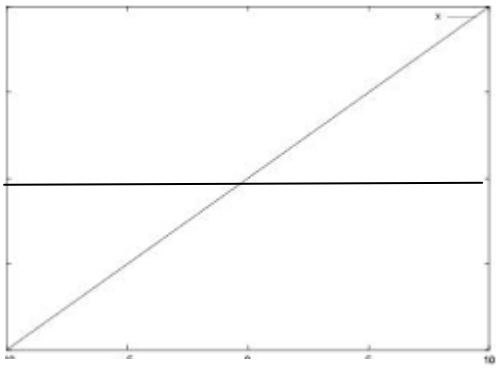
선입선출로 순서대로 들어가서 순서대로 나오는 스케줄링 방식이다. 우선순위가 없어 순서대로 처리된다. 만약 200 번에 첫번째가 들어 있다면, 그 앞의 일들이 다 실행될 때까지 기다릴 수 밖에 없다.

SCHED_RR

태스크가 수행을 종료하거나, 스스로 중지하거나, 자신의 타임 슬라이스를 다 쓸 때 까지 CPU 를 사용한다. 동일 우선순위를 가지는 태스크가 복수개인 경우 타임 슬라이스 기반으로 스케줄링 된다. 만약 동일 우선순위를 가지는 RR 태스크가 없으면 FIFO 와 동일하게 동작한다. 여기서, '우선순위를 고려한다.'라는 것은 time slice 가 다르다.

※ 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.



옆의 표는 O(n)과 O(1)을 나타낸 것이다. 직선이 O(1)이다. 세로축이 처리 개수고 가로축이 시간이다. 그래프가 차지하는 면적이 총 처리해야 할 CPU 명령어 분량이다.

O(1)은 데이터가 얼마가 있어도 걸리는 시간은 일정하다. 반면 O(n)은 데이터가 많을수록 시간도 오래 걸린다. 이러한 양상 때문에 시간을 예측할 수 없다는 것으로 처리해야 할 태스크가 많으면 O(1)이 더 유용하다.

O(1)로 바꾸는 방법은 Hash, Map 이라는 자료구조로 만들 수 있다. 태스크들이 가질 수 있는 모든 우선순위 레벨(0~99)을 표현할 수 있는 비트맵을 준비한다. 태스크가 생성되면 그 태스크의 우선순위에 해당하는 비트를 1 로 set 한 뒤, 태스크의 우선순위에 해당되는 큐에 삽입한다. 태스

크를 생성하고 1 로 세팅하는 이유는 &연산하면 1 이 있는 것만 확인이 되고 없으면 0 이 나오게 되기 때문이다. 큐(queue)를 사용하는 이유는 같은 우선순위를 가진 task 들이 존재할 수 있기 때문에 우선순위에 따른 태스크들을 큐로 넣어 리스트로 관리하는 것(비트맵으로 해결이 되지 않기 때문)이다. 이렇게 해서 비트맵에 가장 처음으로 set 되어있는 비트가 가장 우선순위가 높은 것이므로 그 우선순위 큐에 매달려 있는 태스크를 선택하여 for 문이나 while 문을 사용하지 않고도 스케줄링 작업이 고정시간 내에 완료하여 O(1)방식으로 처리할 수 있게 된다.

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 등)과 제약시간을 가지는 응용들에 효과적으로 적용 가능하다.

7-3 일반 태스크 스케줄링(CFS)

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

Vruntime

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

리눅스는 주기적으로 발생되는 타이머 인터랩트 핸들러에서 scheduler_tick()함수를 호출함으로써 현재 수행중인 태스크의 vruntime 값을 갱신한다.

vruntime += physicalruntime(실제 구동시간) * (weight 0 (0 번의 가중치) / weight curr(현재의 가중치))

로 구할 수 있다.

스케줄링 대상이 되는 태스크를 빠르게 고르는 방법은 가장 작은 vruntime 값을 가지는 태스크를 선정한다. vruntime 이 가장 작다는 것은 가장 과거에 CPU 를 사용했음을 의미하기 때문이다. 이때, 수행된 시간만큼 태스크의 vruntime 값이 증가되며, 항상 작은 vruntime 값을 가지는 태스크가 스케줄링 된다면, 너무 자주 스케줄링이 발생되는 문제가 생길 수 있다. 이처럼 너무 자주 스

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

그렇다면 스케줄러는 어떻게 호출이 될까? 이는 직접적으로 schedule()함수를 호출하는 것과 현재 수행되고 있는 태스크의 thread_info 구조체 내부에 존재하는 flags 필드 중 need_resched 라는 필드를 설정하여 호출하는 것이 있다.

또한 태스크를 만든 비율이 너무 많이 차이가 나는 경우 CPU 를 사용하는 비율도 차이가 많이 나기 때문에(예를 들어 99 개의 태스크와 1 개의 태스크가 따로 생성되었을 경우) 그룹 스케줄링 정책을 지원한다. 이는 vruntime 을 번갈아 가며 실행하여 기회를 주는 것으로 vruntime 은 같아야 하기 때문이다.

9) 태스크와 시그널

시그널은 **태스크에게 비동기적인 사건의 발생을 알리는 메커니즘**이다. 태스크가 시그널을 원활히 처리하기 위해서는 3 가지 기능을 지원해야 한다.

1. 다른 태스크에게 시그널을 보낼 수 있어야 한다. 리눅스는 이를 위해 sys_kill()이라는 시스템 호출을 제공한다. ex) kill -번호 pid
2. 자신에게 시그널이 오면 그 시그널을 수신할 수 있어야 한다. 이를 위해 task_struct 에 signal, pending 이라는 변수 존재한다.
3. 자신에게 시그널이 오면 그 시그널을 처리할 수 있는 함수를 지정할 수 있어야 한다. task_struct 내에 sighand 라는 변수 존재한다.

사용자가 쉘 프롬프트에서 $kill PID 와 같은 명령어를 사용하여 특정 PID 를 가지고 있는 태스크를 종료시키려고 한다. 이때, 사용자는 PID 를 공유하고 있는 쓰레드들이 모두 종료되는 것을 기대할 것이다. 리눅스에서의 PID 는 실제로는 tgid 를 의미(이는 앞에서 getpid()를 사용하여 확인하는 예제로 tgid 를 받아오는 것을 확인했었다!)한다. 따라서 PID 가 같은 태스크들은 의미상 같은 쓰레드 그룹임을 의미한다. 그러므로 **PID 를 공유하고 있는 모든 쓰레드들 간에 시그널을 공유하는 메커니즘이 필요**하다. 이렇게 여러 태스크들 간에 공유해야 하는 시그널이 도착하게 되면 이를 **task_struct 구조체의 signal 필드에 저장**해 둔다. 이러한 시그널을 보내는 작업은 sys_kill()과같은 시스템 호출을 통해 이뤄진다.

반대로 **특정 태스크에게만 시그널을 보내야 하는 경우** task_struct 구조체에 pending 필드에 저장해둔다. signal 이나 pending 에 저장할 때는 시그널 번호 등을 구조체로 정의하여 큐에 등록시키는 구조로, 이를 위해 sys_kill()과 같은 시스템 호출을 도입한다.

각 태스크에 **특정 시그널**이 발생했을 때, 수행할 함수 즉, 시그널 핸들러를 지정할 수 있다. 사용자가 지정한 시그널 핸들러를 설정해주는 것이 **sys_signal()**이다. 태스크가 지정한 시그널은 task_struct 에 sighand 에 저장된다. 이 때, **특정 시그널을 받지 않도록 설정**할수 있는데 이는 task_struct 구조체의 blocked 필드를 통해 이뤄진다. 그러나 **SIGKILL 과 SIGSTOP 은 시그널을 받지 않도록 설정하거나 무시할 수 없다.**

다른 태스크에 시그널을 보내는 과정을 보자. 시그널을 보내는고정은 우선 해당태스크의 task-struct 구조체를 찾아내고, 보내려는 번호를 통해 siginfo 자료 구조를 초기화하고, 시그널 성격에

따라 task_struct 의 signal 이나 pending 필드에 매달아 준다. 이때, blocked 필드를 검사하여 받지 않도록 설정한 시그널이 아닌지 검사한다.

수신한 시그널 처리는 태스크가 커널 수준에서 사용자 수준 실행 상태로 전이할 때(시스템 콜 처리 후 사용자로 넘어올 때) 이루어진다. 커널은 pending 의 비트맵이 켜져있는지, 혹은 signal 필드의 count 가 0 이 아닌지를 검사를 통해 처리 대기중인 시그널이 있는지 확인할 수 있다. 이들 변수가 0 이 아니라면 어떤 시그널 대기중인지 검사하고, 이 시그널이 블록이 되어 있지 않다면 그 번호에 해당되는 시그널 핸들러를 sighand 필드의 action 배열에서 찾아 수행시켜주게 된다.

인터럽트와 트랩 시그널 간의 차이점은 무엇일까? **인터럽트와 트랩이 사건의 발생을 커널**에게 알리는 방법(커널이 처리하기 때문)이라면, **시그널은 사건의 발생을 태스크**에게 알리는 방법(보통 프로세스가 처리하기 때문)이다.

Chapter 4 : 메모리 관리

1) 메모리 관리 기법과 가상메모리

물리 메모리의 한계를 극복하기 위해 여러 가지 기법들이 개발되었는데 그중 하나가 **가상메모리** 기법이다. 가상 메모리는 실제 시스템에 존재하는 물리 메모리의 크기와 관계없이 가상적인 주소 공간을 사용자 태스크에게 제공한다. 32bit CPU 에서는 각 태스크마다 4GB 의 가상 주소 공간을 가지고 있다. 64bit 는 16GB 의 주소 공간을 제공한다.

그러나 이는 물리적으로 4GB 의 메모리를 모두 사용자 태스크에게 제공하는 것이 아니다. 4GB 라는 공간은 프로그래머에게 개념적으로 제공되는(가상적으로 제공되는) 공간이며 실제로는 사용자가 필요한 만큼의 물리 메모리를 제공한다. 즉, 가상메모리는 사용자에게 개념적으로 4GB 를 제공함과 동시에 물리 메모리는 필요한 만큼의 메모리만(demand on paging) 사용되므로 가능한 많은 태스크가 동시에 수행될 수 있다는 장점을 제공한다. 메모리 배치 정책이 불필요하며 태스크 간 메모리 공유/보호가 쉽고, 태스크의 빠른 생성(task_struct 만 보사면 되므로)이 가능하다. 이것이 가능한 것은 컴파일러는 태스크의 배치만 신경쓰고, 커널의 exec 이 물리 메모리 배치에 신경쓰기 때문이다. 즉, 운영체제의 sys_exec 이 물리 메모리에 올려준다.