프로세스 및 리눅스 스케줄링의 이해

(Practical understanding of Linux process scheduling)

컴퓨터학과 2013210063 문기준

제출일:2019.06.06. Freeday Used: 2

Korea University

Contents

과제 개요 / 주요 개념 / CPU burst 그래프 / 소스 코드 / Problem & Solution

과제 개요

운영체제는 Multiprogramming 환경에서 항시 하드디스크를 통해 메모리에 적재된 프로세스들을 보다 효율적으로 CPU가 처리할 수 있도록 여러 프로세스 중 하나의 프로세스를 선택하여 실행하는 스케줄링을 제공한다. 이는 사용자 위주로 돌아가는 개인용 컴퓨터에서 더욱이 그 중요성이 대두된다. 개인용 컴퓨터의 사용자에게는 여러 개의 프로세스가 concurrent하게 돌아간다는 느낌을 들게 해주어야 하기에 Starvation이 발생하지 않게 여러 프로세스 간의 공평성을 보장해주는 것이 중요하다. 이렇듯 이번 과제에서는 공평성과 성능 간의 균형이 효율적으로 이루어진 Linux의 CFS(Completely fair scheduler) 코드 분석 및 수정을 통해 보다 Practical한 리눅스의 프로세스 및 스케줄러에 대한 이해를 얻고자 한다. 또한 수정한 코드를 통해 각 프로세스가 점유하는 CPU Burst를 계산하여 CPU Bound 프로세스와 I/O Bound 프로세스 간의 차이를 알아보려 한다.

주요 개념: 프로세스와 스케줄러

1. **프로세스 란?**

프로세스란 프로그래머에 의해 구현되어 하드디스크에 저장된 파일 형태의 정적인 프로그램이 CPU에 의해 동작되기 위해 메모리에 적재되어 실행되고 있는 상태를 의미한다. 이러한 변화과정은 다음과 같다. 정적으로 하드디스크에 존재하는 프로그램의 소스코드가 Compiler를 통해 object file이 되고 이어 object file은 linker를 통해 executable 이 되며 이는 loader를 통해 최종적으로 메모리 위에 적재되어 process가 된다. 이에 프로세스는 운영체제의 관점에서 전체적인 컴퓨터 시스템의 실행 단위(Execution Unit)에 해당한다. 태초의 컴퓨터는 그 용도가 다양하지 않아 한번에 실행되는 프로세스가 하나에 불과하여 일괄처리(Batch) 형식으로 프로세스를 수행했지만 CPU가 빈번히 idle 상태로 전환된다는 큰 단점이 있었다. 이에 2개 이상의 프로세스를 동시에 실행되는Multiprogramming 환경에 들어서면서 매 순간 실행될 프로세스를 효율적으로 선택하는 것이 대두되어 스케줄러가 탄생하였다. 일괄처리 형식과는 다르게 프로세스의 실행이 끝나기 전에 다른 프로세스로 실행이 전환되는 경우가 생기게 되었고 이에 실행되고 있었던 프로세스에 대한 정보를 저장하기 위한 PCB가 생기게 되었으며 이러한 전환을 문맥전환이라 한다.

1. **스케줄러 란?**

Multiprogramming 환경에서 메모리 내에 실행 준비된 여러 프로세스 중 어떻게 하나의 프로세스를 선택하여 CPU를 할당할 것인지에 대한 정책을 새우는 것이 바로 스케줄러이다. 스케줄러는 해당 시스템의 용도에 따라 스케줄링 정책 특성이 달라지지만 통상적으로 최대의 CPU사용룰, 최대의 처리량, 최소의 응답시간과 최소의 대시시간을 가진 스케줄러가 가장 이상적이다. 하지만 이 모든 조건을 만족시키는 스케줄러를 만드는 것은 현실적으로 불가능하다. 이에 스케줄러를 디자인하는 관점에서 고려해야 하는 요소는 다양하다. 먼저 여러 프로세스가 어떻게 하면 공평하게 CPU를 점유할 수 있는지에 대한 Fairness관점의 요소와 이와 반대로 중요한 프로세스들을 먼저 처리하는 Priority 측면의 요소가 존재하며 우선순위가 낮아 계속 실행되지 못하고 있는 프로세스 또한 수행시켜줘야 하는 Starvation측면의 요소 그리고 마지막으로 축적되는 문맥전환에 의한 Cost를 줄이는 오버헤드 관점의 요소가 존재한다. 이러한 다양한 요소들을 복합적으로 고려해야 한다.

1. **리눅스에서의 프로세스와 스케줄러**

리눅스에서는 thread단위로 스케줄링이 이루어지지만 리눅스 체계 안에서 thread는 많은 공통분모를 공유하고 있는 하나의process로 간주하기에 process와 thread를 같은 선상에서 보고 있다. 간단히 말해 프로세스, thread와 task는 리눅스에서 개념적으로 같은 것이라 간주한다. 이러한 실행 단위를 기반으로 리눅스는 각 프로세스의 가상 CPU 사용 시간인 vruntime을 통해 스케줄링 순서를 결정하는 CFS(Completely fair scheduler)를 사용한다. 해당 스케줄링 방식은 성능과 공평성을 두루 갖추고 있다.

CPU Burst측정 실험 및 결과 분석

1. **실험 개요**

해당 실험은 수정한 스케줄러 코드로 각 프로세스의 CPU burst 를 printk를 통해 출력하여 dmesg로 데이터를 받아 볼 수 있는 환경을 구축한 후에 3가지 실험을 독립적으로30분간 진행하였다. 실험 진행에 앞서 프로세스 종류에 따른 CPU burst의 차이를 확인하기 위하여 확연히 특성이 다른 CPU bound process와 I/O bound process 를 각각 독립적인 실험에서 실행시켰다. 결과적으로 첫번째, 기본 커널 데몬 및 쓰레드 만을 실행했을 때의 실험. 두번째, CPU bound process 만을 수행한 실험. 세번째, I/O bound process만을 수행한 실험이다.

CPU bound process: 수행시간 대부분을 CPU를 점유하는데 사용하는 프로세스로 CPU연산을 위주로 실행되어 CPU burst가 상대적으로 길다. 해당 프로세스를 구현하기 위하여 python을 이용하여 파이(Pi)값을 지속적으로 연산하는 프로그램을 이용하였다. 해당 코드 파일은 과제 파일 안에 첨부하였다.

I/O bound process: I/O처리를 위해 기다리는 시간이 개부분을 차지하는 프로세스로 상대적으로 짧은 CPU burst를 갖는다. 전형적으로 disk나 network communication이 주가 되는 프로세스들이 I/O bound process에 해당한다. 이에 해당 실험에서는 웹 브라우저를 통한 youtube 영상을 재생하는 프로세스를 수행하였다. 해당 유튜브 영상에 대한 URL또한 과제 파일 안에 첨부하였다.

1. **실험 결과**

아래의 모든 실험 결과는 dmesg를 txt파일로 저장하고 linux의 awk command

(예: cat IObound\_process.txt | awk -F': ' '{split($2, pid, "],"); if(pid[2]==" CPUburst"){print $3;}}' > IObound\_result.txt )

|  |  |
| --- | --- |
| * 1. I/O bound process 만을 수행 | * 1. CPU bound process 만을 수행 |
|  |  |
| C. 기본 커널 데몬 및 쓰레드 | |
|  | |

를 사용하여 CPU Burst와 Process ID만을 필터링하여 excel의 frequency 함수를 통해 histogram을 그려냈다. 결과에 대한 데이터 또한 과제 파일 안에 첨부하였다.

1. **실험 결과 분석**
   1. CPU Burst의 출력 빈도

I/O bound process: 5849 CPU bound process: 3924 기본 커널 데몬 및 쓰레드: 714

수정된 코드를 기반으로 생각해 보면 해당 프로세스의 CPU burst의 값이 1000회 바뀔 때 마다 printk의 값이 출력이 된다. CPU burst의 값은 context switching이 일어나 CPU를 점유할 때 마다 바뀌게 되기 때문에 CPU burst의 출력 빈도 값은 문맥 전환이 일어난 횟수와 비례적인 관계를 가지고 있다고 할 수 있다. 이렇듯 출력 빈도가 제일 높은 것은 I/O bound process이다. 이는 I/O bound process가 I/O burst중심으로 돌아가기 때문에 I/O가 발생하는 시점 마다 자발적으로 CPU점유를 양보하게 된다. 이는 상대적으로context switching의 수가 다른 두 실험보다 더 많다는 것을 의미한다. 이에 반해 CPU burst에 초점이 맞춰져 있는 CPU bound process의 경우 문맥전환이 일어나는 수가 I/O bound process보다 적어 출력빈도가 상대적으로 낮다. 이어 아무런 기타 프로세스가 실행되지 않고 있는 기본 커널 데몬 및 쓰레드를 실행하는 실험의 경우 실행되고 있는 프로세스가 다른 두 실험에 비해 현저히 없 때문에 출력 빈도의 최저치를 기록한다.

* 1. CPU Burst의 길이

평균- I/O bound process: 0.15 μs CPU bound process: 0.6 μs 기본 커널 데몬 및 쓰레드: 0.25 μs

위 실험의 그래프를 전제적으로 분석해보면 CPU bound process의 CPU burst가 다른 두 실험에 비해 더 긴 것을 할 수 있다. 이는 CPU bound process가 보다 더 길게 CPU를 점유하고 있다는 것이다. CPU bound process는 CPU연산에 대한 횟수가 상대적으로 더 많기 때문에 더 긴 시간 동안 CPU를 사용할 수 있게 된다. 이에 반해 I/O bound process는 주기적인 자발적 문맥전환으로 인해 CPU 점유 시간이 상대적으로 낮다. 이어 아무런 기타 프로세스가 실행되지 않고 있는 기본 커널 데몬 및 쓰레드를 실행하는 실험의 경우 실행되고 있는 다른 프로세스가 없지만 I/O bound process보다 문맥전화율이 낮아 CPU Burst의 길이는 중간을 기록한다.

* 1. Histogram의 경향을 통한 적합한 time quantum

대체적으로 최근의 모든 운영체제는 상수형태의 정적인 time quantum을 사용하지 않지만 아직 실제 CPU burst time을 정확히 예측할 수 없기에 이번 실험을 통해 적합한 time quantum을 찾는다면 그것은 0.1ms이 적합할 것이라 추정된다. 0.1ms 의 time quantum을 채용한다면 A실험의 91%, B실험의 66%, C실험의 92%가 한번의 time quantum으로 CPU burst가 끝나기 때문이다.

수정한 소스코드 분석

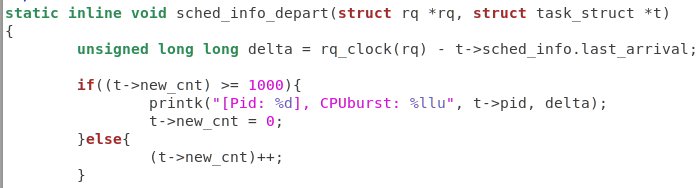
CPU burst를 출력하는 동작을 구성하기 위해 리눅스 4.20.11버전의 커널 코드의 분석은 필수적이었다. 원하는 해당 동작을 구현하기 위하여 수정해야 하는 파일은 총 3개이다. 1) PCB역할을 하는 프로세스의 구조체인 task\_struct 2) CPU점유가 끝났을 때 문맥전환 이전에 호출되는 함수인 sched\_info\_depart() 3)init/init\_task.c에서의 변수 초기화이다.

1. /include/linux/sched.h안의 task\_struct 구조체



CPU burst 값이 1,000회 바뀔 때 마다 printk를 통해 CPU burst 값을 출력을 해야 함으로 CPU burst 값이 바뀔 때 마다 1씩 increment 해줄 변수를 PCB역학을 하는 프로세스 구조체인 task\_struct안에 선언한다. 변수의 이름은 new\_cnt설정하였다.

1. /kernel/sched/stats.h안의 sched\_info\_depart()



프로세스가 CPU점유가 끝났을 때 호출하는 sched\_info\_depart() 함수는 ns단위로 현재 시간을 나타내는 rq\_clock변수와 해당 프로세스가 CPU점유를 시작했을 때의 시간을 알려주는 변수(last\_arrival)의 차이를 통해 CPU burst를 ns단위로 나타내 주는 delta라는 변수가 존재한다. 여기서 task\_struct에 정의한 편수를 해당 함수가 호출 될 때 마다 1씩 증가 시켜 해당 변수가 1000보다 크거나 같으면 CPU burst를 출력하는 형식으로 코드를 수정하였다. 또한 출력과 동시에 변수를 0으로 초기화하여 다음 1000번째를 대비하였다.

3) init/init\_task.c에서의 변수 초기화: “.new\_cnt = 0;”를 통해 초기화 진행

숙제 수행 과정 중 발생한 문제점과 해결방법

1. 코드 수정 부분에 있어서의 검토과정

이번 과제를 진행하면서 코드를 크게 한번 정정하게 되었다. 이는 kernel코드를 분석하는 과정에서 task\_struct 구조체 안에 문맥전환의 수를 카운팅하는 nvcsw와 nivcsw라는 변수가 CPU burst 횟수와 같을 것이라고 예상하고 진행하였다. nvcsw와 nivcsw변수는 /kernel/sched/core.c파일에서 switch\_count라는 포인터 변수를 통해 자발적인 문맥전환 시 nvcsw를 비자발적인 문맥전환 시에는 nivcsw를 증가시킨다. 하지만 보다 확실한 답을 위해서 현재 코드에 쓰인 new\_cnt와 prink를 통해 실시간으로 값을 비교해본 결과 nvcsw와 nivcsw를 더한 값이 더 작은 값이 나오게 되어 sched\_info\_depart()함수를 통해 증가되는 new\_cntd를 기반으로 최종 코드를 수정하였다.

1. pid와 tpid

본 과제는 프로세스 별 CPU burst를 계산하여 분석하는 내용이 포함되어 있지 않았기 때문에 pid와 tpid의 정의가 크게 중요하지는 않지만 kernel 코드를 이해하는 과정에 있어 이 두 요소의 의미 파악이 Linux에서의 process와 thread 개념을 보다 확실하게 알려주는 중요한 단서라는 것을 알게 되었다. 결론적으로 tpid는 thread 단위의 아이디이며 이thread가 같은 process에 종속되어 있다면 같은 pid를 갖는 구조로 되어 있다는 것을 구글링을 통해 알게 되었다.

1. PPT상의 histogram과의 큰 차이

생각보다 실험결과과 PPT의 histogram의 경향을 띄지 않아 여러 번 실험을 진행하였다. 몇 번을 진행을 해봐도 비슷한 모양이 나오지 않아. 4시간으로 시간을 늘려서 측정한 결과 PPT의 histogram과 비슷한 양상을 띄는 것을 확인하였다.