리눅스 프로세스와 스케줄링의 이해

컴퓨터학과 2017320122 김정규

제출일: 2020.06.14. Freeday Used: 5

환경: Microsoft Azure Ubuntu 18.04.02 (64bit), Linux kernel 4.20.11 (가상 환경)

CONTENTS

과제 개요 및 개념

소스 코드

CPU Burst 측정 및 결과

발생한 문제점과 해결방법

**과제 개요**

운영체제는 멀티프로그래밍 환경에서 스케줄링을 제공한다. 이는 메모리에 존재하는 프로세스들을 CPU가 효율적으로 처리하도록 프로세스들을 하나씩 선택하여 실행하는 것이다. 이 스케줄링은 특히 사용자 중심으로 사용되는 개인용 컴퓨터에서 중요하게 다루어진다. 개인용 컴퓨터에서는 프로세스들이 공평하게 동작하도록 함으로써 Starvation이 일어나지 않게 해야한다. 그래야 사용자에게 여러 프로세스들이 동시에 돌아가는 것 같은 느낌을 줄 수 있기 때문이다. 따라서 본 과제에서는 리눅스의 스케줄러 코드를 분석하고 수정하여 스케줄러에 대한 이해를 구체화 할 것이다. 또한 리눅스에서 실행되는 프로세스들의 CPU Burst를 측정하여 CPU Bound와 I/O Bound의 차이가 무엇인지 확인할 것이다.

1. **프로세스란**

프로세스란 간단히 말해 HDD에 저장된 프로그램이 CPU를 통해 동작하기 위하여 메모리에 적재되고 실행되는 것이라 볼 수 있다. 이 과정을 더 자세히 말하자면, HDD의 프로그램 소스코드가 컴파일러를 통해 object file이 되고 linker를 통해 실행 가능한 상태가 되며, 마지막으로 메모리에 로드되어 프로세스로서 존재하게 된다. 이는 곧 컴퓨터에 있어서 하나의 실행 단위라 부를 수 있다.

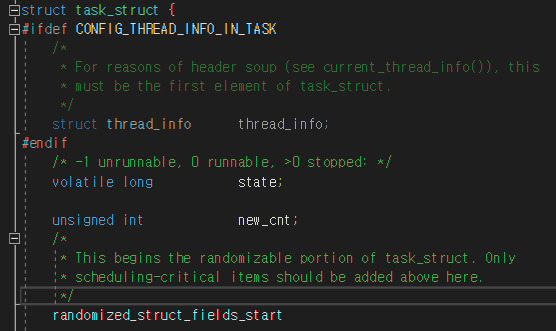
1. **스케줄러란**

단일프로세서 시스템(Single Processor System)에서는 한 번에 한 프로세스만 실행될 수 있다. 하지만 프로세스가 항상 CPU를 사용하는 것은 아니다. 키보드나 마우스 등의 입력 장치에서 사용자의 입력을 기다리거나, 프린터 등의 느린 출력장치에서 데이터를 출력할 때 CPU는 일을 하지 않고 가만히 있는다. 일반적으로 프로세스는 CPU를 한차례 사용(CPU burst)하고 I/O를 한차례 사용(I/O burst)하는 주기를 반복한다. I/O를 사용하는 주기에서는 CPU를 사용하지 않는다. 그렇다면 여러 프로세스를 처리할 때, 한 프로세스가 모든 작업이 끝날 때까지 기다렸다가 다음 프로세스를 실행하는 방법보다 한 프로세스를 실행 가능한 시점까지 실행하고, I/O 등 CPU를 사용하지 않는 작업을 할 때는 다른 프로세스를 실행한다면 CPU 사용 효율을 높일 수 있다.  
 오늘날의 운영체제는 위 아이디어를 프로세스 스케줄링(Process Scheduling)이라는 기술로 구현한다. 프로세스 스케줄링은 운영체제가 하는 가장 중요한 일 중 하나이다. 멀티프로그래밍 환경에서 메모리 내에 실행 준비된 여러 프로세스 중 어떻게 하나의 프로세스를 선택하여 CPU를 할당할 것인지에 대한 방법을 만드는 것이다. 이런 스케줄러를 디자인하는 과정에서는 다양한 요소를 고려해야 한다. 먼저 여러 프로세스들 간의 공평한 CPU 점유를 따지는 ‘Fairness’와, 이와 다르게 중요한 프로세스들을 우선권을 주는 ‘Priority’가 존재하며 우선순위가 너무 낮아 영원히 실행되지 못하는 프로세스 또한 실행시켜줘야 하는 ‘Starvation’, 그리고 마지막으로 문맥 전환에 의한 Cost를 줄이는 ‘오버헤드’ 관점의 요소가 존재한다.

**소스 코드**

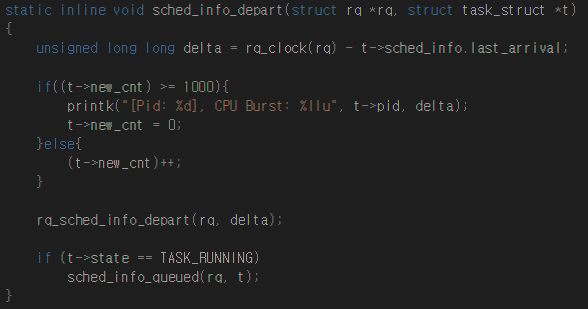
총 3개의 파일을 수정하였다. 수정한 내용은 각각 PCB 역할의 프로세스 구조체인 task\_struct, CPU 점유 종료 시 문맥 전환 이전에 호출되는 sched\_info\_deprt(), 마지막으로 init\_task.c에서의 변수 초기화이다.

1. **/include/linux/sched.h**



CPU Burst 값이 1000회당 1회씩 그 값이 출력되어야 하므로 PCB 역할의 프로세스 구조체인 task\_struct 안에 변수 new\_cnt를 선언한다.

1. **/kernel/sched/stats.h**



프로세스의 CPU 점유가 끝나면 호출되는 sched\_info\_depart() 함수에는 ns단위로 현재 시간을 나타내는 rq\_clock 변수와 해당 프로세스가 CPU 점유를 시작했을 때의 시간을 나타내는 last\_arrival 변수가 존재한다. 그리고 이들의 차를 통해 CPU burst를 ns단위로 나타내 주는 delta라는 변수가 존재하는데, 여기서 task\_struct에 정의한 new\_cnt를 해당 함수가 호출될 때마다 1씩 증가 시켜 값가 1000보다 크거나 같으면 CPU burst를 출력하도록 코드를 수정하였다.

1. **init/init\_task.c**

task\_struct 파트에 ‘.new\_cnt = 0’을 추가하여 변수를 초기화한다.

**CPU Burst 측정 및 결과**

1. **개요**

이 측정은 상기의 파일들을 수정하여 각 CPU Burst를 메시지로 출력해 터미널로 확인할 수 있도록 설정한 뒤에 다음의 두 가지 측정을 약 한시간 가량 진행하였다.

1. CPU Bound Process : 연산을 중심으로 실행되는 프로세스로 CPU Burst가 긴 편에 속한다. 이 프로세스가 사용되는 측정을 진행하기 위해 간단한 계산을 지속적으로 반복하는 프로그램을 실행하여 측정하였다.
2. I/O Bound Process : 입출력 처리를 중심으로 실행되는 프로세스로 네트워크 상호작용이나 키보드/마우스 입출력을 포함한다. CPU Burst가 짧은 편에 속하며 이 프로세스가 사용되는 측정을 진행하기 위해 측정 시간 동안 youtube 영상을 틀어두었다.
3. **결과**
4. CPU Burst 출력 빈도 : CPU Burst 값은 문맥 전환이 일어나 프로세스가 CPU를 점유할 때마다 갱신된다. 따라서 CPU Burst의 출력 빈도는 문맥 전환이 일어난 횟수에 비례한다고 볼 수 있다. 여기서 출력 빈도가 더 높은 것은 I/O Bound Process이다. 이는 I/O Bound Process가 I/O Burst를 중심으로 돌아가기 때문에 입출력이 발생하는 시점마다 자발적으로 CPU 점유를 양보하게 된다. 이는 문맥 전환 횟수가 CPU Bound Process보다 더 많다는 것을 의미한다. 이에 반해 CPU Burst에 초점이 맞춰져 있는 CPU Bound Process의 경우 문맥 전환이 일어나는 수가 I/O Bound Process보다 적어 출력 빈도가 상대적으로 낮다.
5. CPU Burst 길이 : 위 실험의 그래프를 분석해보면 CPU Bound Process의 CPU Burst가 I/O Bound Process보다 더 길다는 것을 확인 할 수 있다. 이는 곧 CPU Bound Process가 CPU를 점유하는 시간이 더 길다는 뜻이다. CPU Bound Process는 CPU 연산 횟수가 많기 때문에 더 긴 시간 동안 CPU를 사용할 수 있게 된다. 이와 달리 I/O Bound Process는 주기적으로 일어나는 문맥 전환으로 인해 CPU 점유 시간이 상대적으로 낮다.
6. 적합한 Time Quantum : 0.1ms로 추정한다. 왜냐하면 0.1ms를 Time Quantum으로 사용한다면 CPU Bound Process의 경우 약 65%, I/O Bound Process의 경우 약 90%의 CPU Burst가 단 한 번의 Time Quantum으로 종료될 수 있기 때문이다.

**발생한 문제점과 해결방법**

과제 수행 중 파일을 수정하고 컴파일하는 과정에서 우분투에 커널 패닉이 발생했다. 어떻게든 고쳐서 진행하려 했으나 해결 방법에 대한 정보와 시간이 부족하여 기존 가상 머신을 버리고 새 가상 머신에 우분투를 설치하여 다시 진행하였다. 커널 단계의 코드를 수정할 때에는 매우 조심해야 한다는 사실을 다시금 느낄 수 있었다.