운영체제 2차 과제 보고서

소속 학과	고려대학교 컴퓨터학과
학번	2018380603
이름	정이든
제출 날짜	6/13
Free day 사용 일수	4 Day

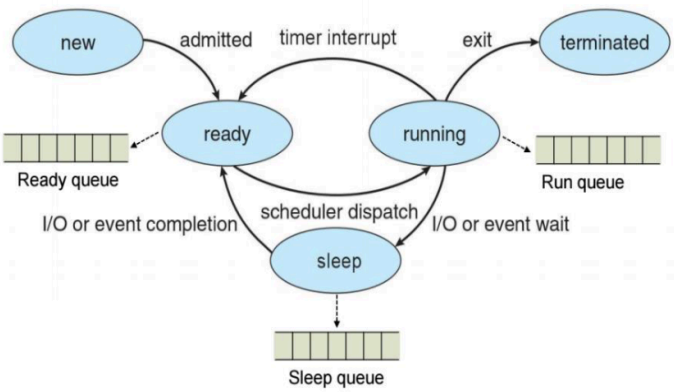
1. 과제 개요

본 과제의 목적은 프로세스의 CPU 점유 시간을 결정하는 리눅스의 프로세스 스케줄러에 대한 이해를 바탕으로 프로세스의 종류에 따라 스케줄러가 어떻게 프로세스에 CPU 점유 시간(burst)을 할당하는지를 살펴보는 것이다. 우리는 이를 CPU burst duration을 측정함으로써 조사해본다.
또한 이 CPU Burst Time도 프로세스가 실행되면서 누적된 Time 값이 아닌 특정 스케줄링 횟수에 도달했을 때의 burst time을 측정하는 것이다. 일반적인 운영체제 커널은 사용자가 이용하는 프로세스들에 대해서 CPU Burst Time을 알 수는 없다. 따라서 이번 과제를 수행하기 위해서 리눅스라는 운영체제에서 스케줄러가 프로세스에게 할당하는 Time Quantum, 스케줄링 방식에 대한 abstraction을 코드를 통해 이해하고, 이를 기반으로 커널을 분석 및 수정하여 프로세스들의 전반적인 CPU Burst time 분포를 측정하는 것이다.

2. 프로세스와 스케줄러의 개념

프로세스를 이해하기 앞서, 프로세스는 프로그램과 밀접한 관계에 있기 때문에 프로그램과 연관 지어서 이해할 필요가 있다. 일반 사용자가 작성하는 여러 파일에 걸친 소스 코드들은 컴파일러를 통해서 컴퓨터가 이해할 수 있는 여러 목적 파일들로 컴파일 되며, 이런 목적 파일들은 또 다시 라이브러리들과 링킹되어 하나의 실행가능한 파일로 바뀌게 된다. 디스크에 저장된 실행가능한 파일은 프로세스로 변환할 수 있도록 header, text, 여러 data 들이 포함되어 있기에, 이 파일을 실행하게 되면 메모리에 이 프로그램이 적재되면서 프로세스가 된다. 이런 프로세들은 스케줄링의 단위(abstraction)로서 의미를 갖는데, 이 말은 곧 프로세스들은 스케줄러에 의해서 처리되는 것을 알 수 있다. 또한 프로세스들이 갖는 특성이 있는데 각 프로세스들은 메모리에서 Protection Domain 이 존재한다는 것이다. 즉, 한 프로세스가 다른 프로세스의 메모리 영역을 침범, 간섭하지 못한다는 뜻이다.

여러 프로세스들은 CPU 에서 수행될 때 context switch 가 일어나면서 수행되는데, context switch 는 Time Quantum 이 소진되거나 사용자가 I/O 를 호출하게 되었을 때 발생한다. 이런 상황들을 조절하고 처리하는 것이 바로 스케줄러다. context switch 가 발생하면 프로세스 old 의 state 를 저장하고 동작을 멈춤과 동시에, 프로세스 new 의 state 를 복원하고 수행한다. 즉, 프로세스는 여러 state 들을 비롯한 여러 정보들을 특정 구조체를 통해서 갖고 있으며, 스케줄러는 프로세스의 상태에 따라서 스케줄링 할 특정 자료구조들이 존재한다.



그렇다면 스케줄링이란 무엇인가? 스케줄링은 여러 프로세스들에 대해서 CPU 의 사용을 어떻게 분배하는가에 대한 Policy 라고 할 수 있다. CPU-bound 프로세스, I/O-bound 프로세스 중에서 어떤 프로세스가 많은가에 따라서 여러 스케줄링 기법의 효율성이 달라지지만, 공통적인 목표는 CPU 의 Utilization 을 최대화하는 것이다. 스케줄링을 수행하는 것이 바로 위에서 언급한 것과 같이 스케줄러인데, CPU 사용률과 처리량을 최대화, 응답시간과 대기시간을 최소화하는 스케줄러가 가장 이상적이다. 하지만 현실적으로 이는 불가능하기에 사용하는 시스템의 용도에 따라서 요구사항이 달라진다.

3. 리눅스 커널 소스 코드 분석 및 작성한 소스 코드의 설명

```
static inline void sched_info_depart(struct rq *rq, struct task_struct *t)
{
        unsigned long long delta = rq_clock(rq) - t->sched_info.last_arrival;

        rq_sched_info_depart(rq, delta);

        if (t->state == TASK_RUNNING)
                sched_info_queued(rq, t);

        if(t->sched_info.pcount % 1000 == 0) {
                printk("[Pid: %d], CPUburst: %lld\n", t->tgid, delta);
        }
}
```

해당 과제의 목표는 매 1000번째 스케줄링 때마다 샘플링된 프로세스의 CPU burst를 기록하는 것이기 때문에, 프로세스가 CPU 점유를 마치고 나온 직후에 수행되는 함수인 sched_info_depart 함수를 과제의 목적에 맞게 수정해야 함을 느꼈다. 코드를 짜면서 샘플링 횟수를 세는 변수가 필요함을 인지하고, 프로세스가 state변화를 할 때마다 횟수를 counting 해주는 변수를 직접 선언하여 코딩해봤는데, 이렇게 해서는 안됨을 느끼고 난 후, 인자로서 받는 task_struct를 자세히 뜯어보기 시작했다. 그리고 sched_info 구조체의 pcount 변수를 찾을 수 있었다.

위의 코드는 /usr/src/linux/kernel/sched/stats.h 파일에 있는 sched_info_depart 함수를 앞에서 말한 Abstraction에 맞게 수정한 코드이다. 먼저 delta는 CPU로부터 할당이 끝난 프로세스의 CPU burst이다. rq_clock(rq)은 현재 시각이므로 depart() 함수이 호출된 시점과 같게 된다. 그리고 t->sched_info.last_arrival은 CPU burst가 끝난 프로세스가 CPU를 마지막으로 할당 받은 시각이기 때문에, 이 두 값을 빼면 task_struct 포인터 t가 가리키는 프로세스의 CPU burst 시간이 된다. 그리고 task_struct 구조체 중에서 sched_info라는 구조체가 있는데, 그 구조체에 pcount라는 프로세스의 호출 횟수를 기록하는 integer 변수가 있다.

우리는 샘플링된 프로세스가 1000회 호출될 때 마다 CPU burst를 출력해야 하므로 t->sched_info.pcount가 1000으로 나누어 떨어질 때 task_struct가 가리키는 프로세스의 ID(tgid)와, CPU burst(delta)를 출력하도록 코드를 작성하였다

.

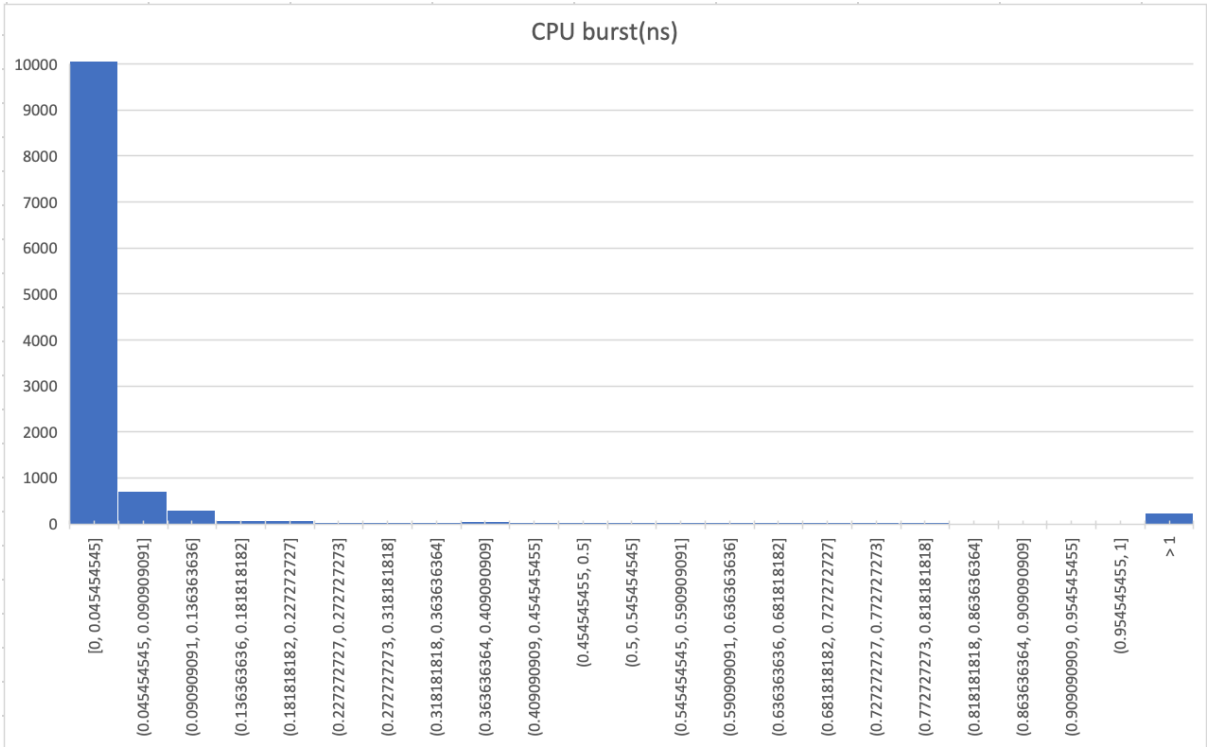
위의 sched_info_depart함수에서 t->sched_info.pcount가 1000으로 나눠 떨어질 때 수행하도록 한 이유는 task_struct 구조인 t는 프로세스가 스케줄링 될 때 memset을 통해 t가 가리키는 구조체의 sched_info의 모든 멤버 변수들을 0으로 초기화 해주기 때문이다. 따라서 sched_info의 멤버 변수 pcount가 0으로 초기화 되어 스케줄을 시작하면서 sched_info_arrive에서 1씩 증가해가므로, 1000번째마다 스케줄링을 샘플링하기 위해서 (t->sched_info.pcount % 1000) == 0 조건문을 만들어 주었다. 스케줄링을 시작할 때 수행되는 함수들은 core.c에 존재하기 때문에 memset을 통해 멤버 변수를 초기화하는 해당 코드는 /kernel/sched/core.c에서 밑과 같이 확인할 수 있었다

.

```
#ifdef CONFIG_SCHED_INFO
        if (likely(sched_info_on()))
                memset(&p->sched_info, 0, sizeof(p->sched_info));
#endif
```

또한 프로세스의 CPU Burst time뿐만 아니라 프로세스를 식별할 수 있어야하므로, task_struct t 아래에 존재하는 프로세스 ID인 tgid를 사용하여 프로세스의 ID를 출력해주었다.

4. CPU burst에 대한 그래프 및 결과분석



위 그래프는 커널에서 작동하는 기본 데몬과 쓰레드 이외에 추가적으로 Wireshark, Calculator, Klondlike(게임), Youtube, Naver 등의 웹 브라우저를 포함하여 총 5개의 프로세스를 30분동안 실행시킨 후 각 프로세스들의 CPU burst를 분석한 히스토그램이다. 실행한 프로세스들은 대부분 CPU-bound 프로세스이건 I/O-bound 프로세스이건 대체적으로 0.1ms 이하의 CPU Burst Time 을 갖는 프로세스들이 대다수였다. 그 0.1ms 이하의 Time Quantum 을 가진 프로세스들 중에서도 대다수는 ns 중에서도 1000ns 처럼 어마어마하게 작은 ns 단위를 이용하고 있었다. 하지만 몇몇 프로세스들의 CPU Burst Time 이 다른 프로세스들에 비해서 현저히 높은 프로세스들이 있었다. 이 프로세스는 대부분 게임에서 생성된 프로세스였다. 물론 게임을 제외한 몇 안 되는 프로세스가 높은 CPU Burst Time 을 갖기는 했지만 확실히 교수님께서 게임을 실행하여 측정하는 것이 좋다고 말씀하신 이유를 알 수 있었다. 게임을 실행하지 않고 실험을 했다면 다양한 많은 프로세스들을 실행했음에도 불구하고 높은 CPU Burst Time 을 측정하기는 어려웠을 것이라고 생각되고, 곧 높은 CPU Burst Time 을 갖는 프로세스가 최근 운영체제에는 없는 것인지 아니면 그런 Time Quantum 을 갖는 프로세스를 실행하지 못한 것인지 알 수 없었을 것이다. 비록 많은 프로세스들을 실행한 것은 아니었지만 기본 커널 쓰레드와 데몬에서 나오는 프로세스들을 포함하여 Burst Time 의 관찰한 결과 수업시간에 배운 CPU Burst Time 의 편향대로 나타나는 것을 확인할 수 있었다. 결과적으로 스케줄러가 프로세스들을 스케줄링 할 때 동일한 Time Quantum 을 할당하지 않고 프로세스 별로 다른 Time Quantum 을 할당할 뿐만 아니라, 대부분 프로세스에서 사용하는 Time Quantum 은 수업시간에 배운 ms 단위보다는 ns 중에서도 어마어마하게 작은 ns 시간 단위를 이용하는 것을 알 수 있었다.

5. 과제 수행하면서 마주친 문제 및 해결과정

먼저 과제를 수행하면서 샘플링의 호출 횟수를 세어주는 pcount 변수를 찾기 위해 가장 많은 시간을 쏟았던 것 같다. 처음에는 task_struct가 가리키는 프로세스가 준비 큐에서 CPU로 올라갈 때마다 0부터 1씩 더해주면서 카운트를 해주는 변수를 sched_info_depart 함수에서 직접 선언해보았는데, 이러면 프로세스가 CPU 할당이 끝날 때마다 0으로 초기화가 되기 때문에 내가 직접 변수를 만들어서 카운트를 해주는 것은 솔루션이 아님을 인지하고 sched_info_depart 함수의 인자로서 받는 task_struct의 멤버 변수들을 자세히 분석하기 시작하였다. 워낙 task_struct 구조체의 필드 변수가 많기도 했고, 그것의 역할들을 하나씩 찾아보는 과정에서 커널을 많이 분석했던 것 같다.

이것 이외에도 1000번까지의 누적 CPU burst를 출력하는 것인지, 아니면 1000번 째의 CPU burst를 출력하는 것인지 아니면 1000번 째 호출 마다 CPU burst를 출력하는 것인지를 파악하는게 헷갈렸고, 그래프를 그리면서 편향을 파악할 때 직접 실행시키는 프로그램만 측정하는 것인지, 아니면 커널이 기본적으로 제공하는 시스템 프로세스도 포함을 해야하는지도 조금 헷갈렸다. 과제 수행하면서 궁금했던 점은 pid와 tgid의 차이였다. 처음에는 pid가 프로세스 아이디, tgid는 그룹 아이디를 뜻하는 것 밖에 알지 못했다. 그래서 과제 튜토리얼에서 tgid를 잘못 체크한 것이 아닌가 싶었는데, 자세히 찾아보니 t->pid는 고유한 태스크의 ID를 의미하고 t->tgid는 고유한 프로세스 ID를 의미하여 프로세스 생성 시 마다 생성되는 것임을 알 수 있었다. 따라서 튜토리얼에서 나온 tgid가 잘못된 것이 아니였던 것을 깨달았다.

내가 가장 먼저 찾아 보았던 것은 stats.h 에 sched_info_depart 와 sched_info_arrive 가 있었기 때문에 stats.h 부터 찾아봤고, 바로 해당 함수들을 이해하기는 어려워서 함수에서 쓰이는 구조체들에 대해서 분석을 했다. sched.h 에 있는 task_struct 와 rq 에 대해서 살펴보니 쉽게 이해하기가 어려워서 시간 날 때마다 꾸준하게 봤던 기억이 난다. 그 결과 task_struct, rq의 전반적인 구조를 이해할 수 있었고, 그렇게 다시 stats.h를 보니 sched_info_arrive 내에서 실행되는 rq_sched_info_arrive 및 조정되는 인자들과 sched_info_depart 내에서 실행되는 rq_sched_info_depart 및 조정되는 인자들을 수월하게 이해할 수 있었다. 또한 각 프로세스들이 스케줄러에 의해서 rq에 들어갔다가 나오면서 context switch가 일어나는 부분도 찾을 수 있었다. 더불어 context switch 가 어디에서 호출되는지도 찾을 수 있었는데, 이는 core.c 에서 전반적인 스케줄링이 처리될 때 호출되는 것을 찾을 수 있었다. 이내 다시 한 번 더 task_struct 로 들어가서 해당 구조체를 분석해보니 pcount라는 멤버 변수를 찾을 수 있었고, 이 변수는 프로세스가 스케줄 되어 Run Queue에 들어가 CPU에서 수행될 때마다 값을 누적해가는 변수였다. 디폴트 커널이 스케줄 결과를 일일이 로그로 찍어내지는 않더라도 그 값을 언제든 쓸 수 있게 변수가 다 존재한다는 것을 보고 정말 '잘 만들어진' 프로그램임을 다시 한 번 더 느낄 수 있었다. 결과적으로 비록 커널 소스의 전체를 이해하지는 못했지만, 필요한 부분 별로 시간을 두고 차근차근 보고 나니 과제에 필요한 과정들은 대체적으로 이해할 수 있었다. 사실 이번 과제는 CPU Burst Time 을 측정하면서 시스템 전반적으로 이용하는 프로세스들과 사용자가 이용하는 프로세스들의 Time Quantum 할당 편향을 파악하는 것이었음에도 불구하고, 프로세스 별 CPU Burst 편향을 파악하는 것보다 커널 소스를 분석해보는 것이 정말 좋은 경험이 되었다. 언제 어떻게 또 프로그래밍을 하게 될지는 모르겠으나 이제까지 내 코딩 습관을 반성할 수 있는 좋은 기회이기도 했다. 이번 과제 때문만이 아니라, 종강 후 방학 때 다른 부분도 조금씩 읽어갈 생각이다