**Operating System**

**- 2차과제 -**

**Process 특성을 고려한 CPU 스케줄러 구현**

**sched\_faster, sched\_slower**

**컴퓨터학과**

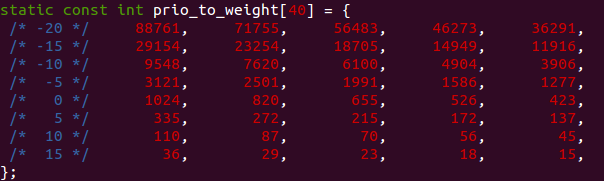
**2013210107**

**김정수**

**2016.05.05. 제출**

1. **개발환경**

Oracle VM VirtualBox에서 배포판 리눅스 Ubuntu 14.04 LTS를 사용  
새로 커파일한 Kernel Version 3.14.63

1. **CFS 스케줄러**
2. **CFS는 어떠한 정책을 가지고 스케줄링 하는가?**  
   기본적으로 Process Scheduling이란 현재 실행되고 있는 process들에게 CPU를 할당(분배) 시켜주는 것을 말한다. 매 순간, 다음에 실행될 적당한 next process는 스케줄링 class나 정책, 그리고 우선순위(process priority)에 따라 정해지게 된다. 이렇게 여러 개의 process가 동시에 빠르게 시간단위로 나누어 CPU를 사용함으로써 마치 모든 process가 동시에 실행되고 있는 듯한 모습이 보여진다.  
   CFS(Completely Fair Scheduler)는 이상적인 multi-tasking processor를 전제하고 있으며, 말 그대로 모든 process를 공평하게 실행하는 것이다. 만약 두 개의 process가 실행된다면 같은 시간에 각각 50%의 processing power를 가져야 한다. 다시 말해, 두 task가 같은 시간에 시작했다면, 모든 순간에 두 task의 실행시간은 같아야 한다. 그러나 물리적으로 이러한 방법은 불가능하고 비효율적이다. 따라서 virtual runtime의 개념을 이용하여 균형을 맞춘다.  
   결론적으로 모든 실행중인 process는 현재 실행되고 있는 전체 process 수에 의하여 CPU처리시간을 할당 받게 된다. 또한, 필요에 따라 task의 priority에 의해 변화하기도 한다.
3. **우선순위를 결정하는 기준은 무엇이고 어떻게 우선순위를 업데이트하는가?  
   - Nice 란 무엇인가?**  
   process의 우선순위(priority)를 반영하기 위해서는 결국 각 프로세스마다 가중치를 설정해 주어야 한다. 이러한 가중치, 즉 weight는 CPU를 사용하는 시간을 정해주는데 필요한 Virtual runtime을 계산하는데 쓰인다. 이러한 weight 값들은 Nice를 통해 지정된다.  
   Nice는 -20~19 의 값을 가지며, 작을수록 큰 priority를 의미한다. 각 Nice 값에 대한 실제 weight값은 sched.h의 prio\_to\_weight[] 배열에 저장되어 있다.  
     
   Nice는 단순히 priority라 보면 되겠고, weight는 그에 상응하는 실제 계산되는 값이다.  
     
     
   - **Vruntime은 무엇인가?**  
   CFS에서 각 프로세서가 동일하게 실행되었다고 표현은 하지만, 실질적으로 모두가 똑같은 실행시간을 갖는다는 것은 불가능하다. 따라서 프로그램의 실제 실행시간이 아닌 가상 실행시간 Vruntime을 공평하게 나누었다는 점에서 CFS가 수행되는 것이다.  
   Vruntime은 프로그램의 실제 한 번의 실행시간에(delta\_exec) 우선순위(NICE)를 적용하여 계산한 다음 누적되는 식으로 계산된다. (calc\_delta\_fair)  
   정리하면, 기존의 스케줄러 구현에서 time slice라 부르던 time quantum이 바로 Vruntime이다. Vruntime의 time slice가 10으로 주어진다 가정하면, 각 프로세스의 Vruntime 실행시간이 10에 도달하면 task\_switching이 발생하게 된다. 그 다음, 스케줄러는 현재 실행 큐에서 가장 vruntime이 작은 프로세스를 선택하여 수행하게 된다.

*time interrupt 🡪 vruntime 증가 🡪 vruntime 실행시간이 time quantum도달 🡪 스케줄링*

이를 효율적으로 하기 위해 CFS는 Red-Black tree를 사용해 프로세스를 저장하고 불러온다.

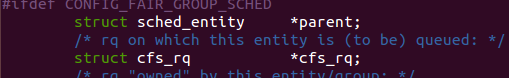
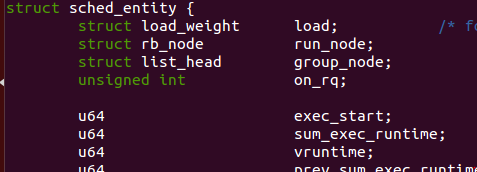
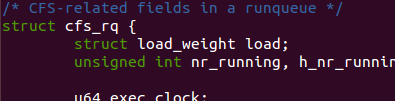
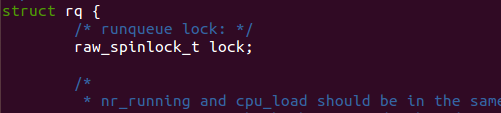
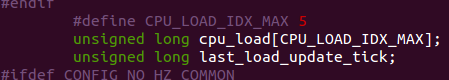
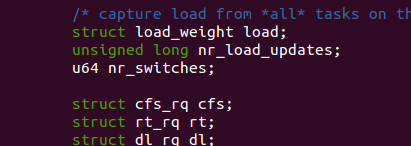
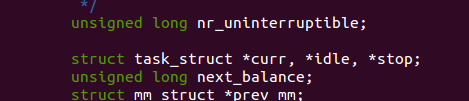
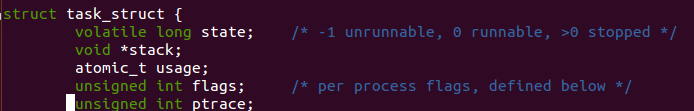
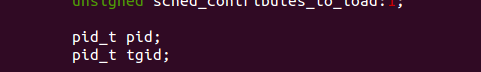
- **Vruntime은 어떻게 업데이트 되는가?**  
우선, 스케줄링 주기(priod)의 개념에 대해 알아야 하는데, 이 주기 동안 스케줄러상의 실행 큐에 대기중인 프로세스가 모두 실행된다. 단순히 한번의 실행시간 단위라고 생각하면 되겠다. 간단히 10ms라 가정하자.

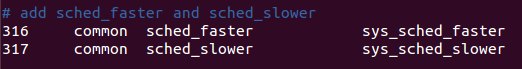
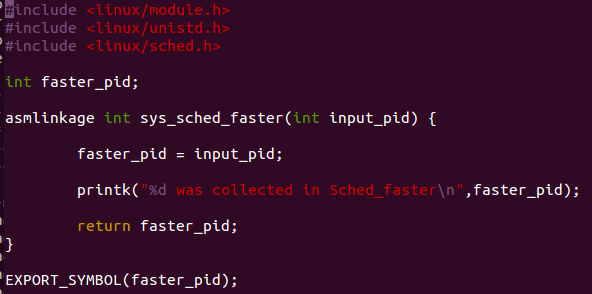
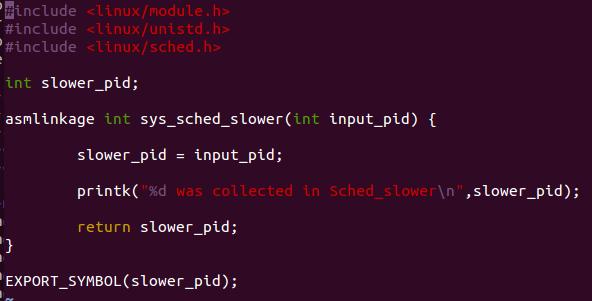
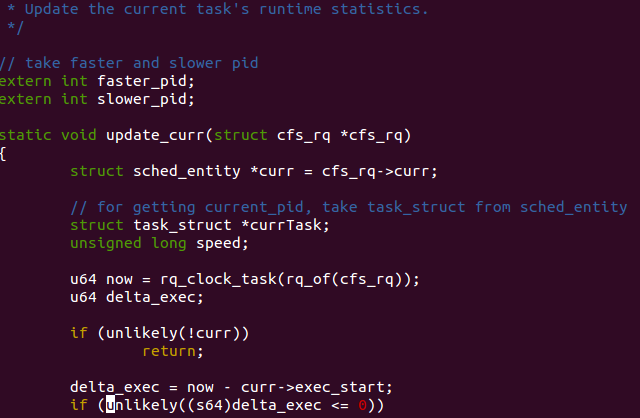
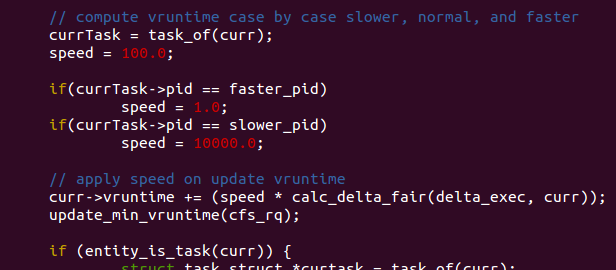
이제는 Vruntime과 실제 실행시간을 나누어서 생각해 보겠다.  
실제 각 프로세스의 실행시간은 다음과 같이 계산된다.  
프로세스 A = ( 10ms ) \* (A priority) / (A priority + B priority)  
priority는 위에서 설명하였듯, 해당 Nice값에 의하여 실제 weight가 load된다.  
다시 다른 예로 설명하면,  
프로세스 B = ( 10ms ) \* (B weight) / (A weight + B weight)  
결국, 한 주기에서 전체 weight중에 자신의 weight가 차지하는 비율만큼 실행 시간을 가져가는 것이다. 이는 A, B 두 개의 프로세스 실행일 때고, 그 수가 더 많아지면 각 프로세스마다 그냥 추가만 하여주면 된다.   
Ex) (A weight) / (A weight + B weight + C weight + D weight)

하지만, Vruntime은 실제 실행시간과 달리 각 process마다의 dependent부분을 없애준다.  
프로세스 A = ( 10ms ) / ( A weight + B weight )  
다시 말해, 모든 프로세스가 동일한 Vruntime을 갖게 된다는 말이다.  
이러한 과정은 실제 sched/fair.c 에서 update\_curr 함수를 통해 실현된다.  
curr->vruntime 에 계산된 값을 쌓아가면서 scheduling 시점을 불러온다.  
상대적으로 다른 프로세서들은 vruntime이 작아져(작게 보여) 다음 스케줄링에 선택될 확률이 높다.

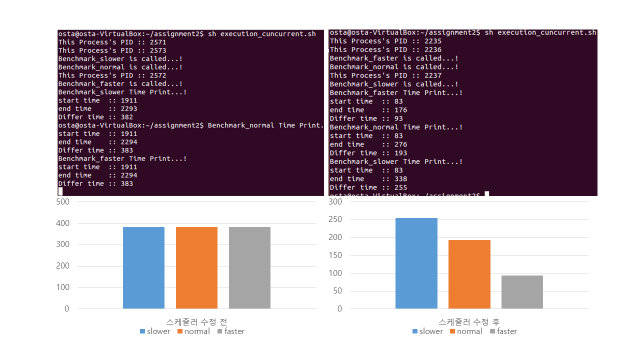
1. **스케줄러(do\_schedule())가 호출되는 3가지 시점은 언제인가?**  
   우선적으로 외부에서 지속적인 time interrupt가 들어와야 한다.  
   처음 실행할 task가 준비될 때 스케줄러가 호출되어 그 task를 실행하게 한다.  
   해당 task의 vruntime이 time quantum을 넘어서면 다시 스케줄러가 호출된다.  
   해당 task가 끝난 시점에서도 나머지 task들이 다시 CPU할당량을 재배분 받아야 하기 때문에 스케줄러가 호출된다.  
   스케줄러의 주 entry point는 바로 kernel/sched.c에 있는 schedule()이다. 주 목적은 실행될 next task를 찾고, local variable next에 이를 넣는 일이다. 그리고 그 next task로 문맥전환(context\_switching)을 한다. 만약 previous task밖에 없거나 여전히 runnable한다면, 아무 일도 수행하지 않는다.
2. **Task가 wake up되었을 때, 스케줄러는 어떠한 동작을 수행하는가?**  
   task가 다시 혹은 처음으로 wake up될 때, select\_task\_rq()가 수행된다. 다시 말해 waking task가 들어갈 runqueue를 선택하게 된다. 이는 RT스케줄러에 의해 시행되는데, 만약 CPU runqueue에 있는 현재 실행중인 task가 RT task고, 그 우선순위도 높다면 다른 runqueue를 찾게 된다. 그렇지 않다면 같은 CPU에 두고 RT스케줄러를 쉬게 둔다.
3. **Vruntime의 증가속도를 조절하는 것 이외에 특정 프로세스를 빠르게 스케줄링 할 수 있는 아이디어**  
   vruntime이 아니라 결과적으로 계산되는 Nice, 즉 weight를 직접적으로 건드리면 이 또한 CPU를 선점하는데 도움이 되지 않을까 싶다. calc\_delta\_fair에서 vruntime을 증가시켜줄 값을 계산하는데, 이 때 weight값이 이용된다. 이 부분을 건드리는 것 또한 하나의 방법이 될 듯 싶다.  
   tgid라는 변수를 보니 실행되는 프로세스가 그룹화 되어 있는데, 우선순위 별로 그룹을 묶어 원하는 그룹마다 선점율을 높여 프로세스마다 그룹별로 처리하는 것도 좋을 듯 싶다.
4. **구현**

구현을 위해서는 필요한 자료구조를 따라가다 보니  
sched\_entity, cfs\_rq, rq 및 task\_struct에 대한 사전지식이 필요함을 알게 되었다.

1. **sched\_entity**  
     
   앞서 말한 각 프로세스에 대한 weight값을 저장하는 load\_weight와,   
   가상 실행시간 vruntime 값을 가지고 있다. 또한 해당 entity가 등록된 runqueue를 지칭하는 포인터 \*cfs\_rq도 있음을 볼 수 있다.
2. **cfs\_rq**  
     
   \\SEC-PC\ShareFolder\move5.PNG  
   \\SEC-PC\ShareFolder\move6.PNG  
   sched\_entity와 같이 weight에 대한 정보도 갖고 있고,  
   runqueue 내에서의 current, next, last sched entity를 의미하는 포인터도 정의되어 있다.  
   이는 스케줄링 시 다음 entity를 결정할 때 next 포인터가 사용됨을 유추해 볼 수 있다.  
   다음으로 이 cfs\_rq가 실제 접촉되어 있는 cpu runqueue를 지정하는 \*rq가 있다.
3. **rq** – main CPU runqueue 자료구조이다.   
     
     
     
     
   직접적인 cpu runqueue이기 때문에 cpu에 대한 정보와 다른 자료구조와   
   마찬가지로 weight에 대한 정보가 있음을 알 수 있다.  
   이 runqueue안에는 현재 진행중인 task인 \*curr이 있음을 알 수 있다.  
   task를 표현하는 task\_struct의 자료구조를 마지막으로 찾아보자.
4. **task\_struct**  
     
     
   실제 각각의 정보인 task의 자료구조는 상당히 내용이 많다.   
   현재 state를 나타내는 state (-1 : unrunnable, 0 runnable, >0 stopped)가 눈에 띈다. 또한 중간쯤에 위치한 pid와 tgid가 바로 우리가 사용하기 위한 현재 실행중인 process id를 알아내는데 사용됨을 알 수 있다. pid는 프로세스 id이고 tgid는 프로세스가 그룹형태로 있을 경우 그 그룹을 의미하는 id이다.

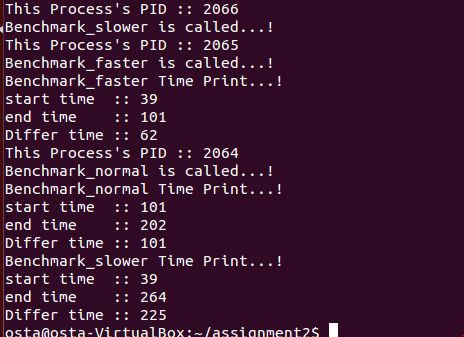
* **sched\_faster와 sched\_slower을 각각 번호 316, 317로 system call에 추가.**  
  syscall\_64.tbl  
    
  syscalls.h  
  \\SEC-PC\ShareFolder\move14.PNG  
  Makefile  
  \\SEC-PC\ShareFolder\move15.PNG
* **sched\_faster.c**  
    
  application에서는 sys\_sched\_faster 호출과 함께 자신의 pid를 인자로 넘겨준다.  
  이를 받은 sys\_sched\_faster은 자신의 변수 faster\_pid에 그 pid를 저장하고,  
  커널의 다른 함수에서도 이 변수 faster\_pid를 쓸 수 있도록 EXPORT시켜준다.  
  EXPORT\_SYMBOL 사용.
* **sched\_slower.c**  
    
  sched\_faster.c와 동일. faster\_pid만 slower\_pid로 바꾸어 주었다.
* **fair.c**  
    
  sys\_sched\_faster와 sys\_sched\_slower를 통해 전달받은 faster\_pid와 slower\_pid를  
  extern을 이용해 저장시켜둔다.  
  update\_curr은 설명에도 나와있듯 task update의 진입점이다. 이곳에서 vruntime을 계산하고 업데이트한다.  
  현재 실행중인 pid정보를 알아내기 위한 task\_struct를 저장할 변수로 \*currTask로 선언한다. 또한 각각 vruntime update 속도를 조절할 변수로 unsigned long speed를 선언한다.  
    
  cfs\_rq에는 현재의 정보를 담고 있는 자료구조로 sched\_entity를 택한다. 따라서 현재 실행되는 프로세스는 sched\_entity로 표현된다. 하지만, 이 안에는 pid에 대한 정보가 없기 때문에 미리 만들어둔 task\_struct \*currTask에 sched\_entity의 task를 뽑아낸다.  
  이제 task\_struct안에는 pid정보가 있기 때문에 현재 수행중인 pid를 currTask->pid를 통해 알아낼 수 있다.  
  현재 pid가 각각 faster\_pid, normal\_pid, slower\_pid냐에 따라 vruntime에 더해주는 변수를 100배 느리게, 같게, 100배 빠르게 설정하여준다. 미리 선언된 speed에 값을 지정하여 해당 값에 곱하여 설정해준다. ( speed가 각각 0.01, 1, 100이 아니라 100씩 곱해진 값인데, 그 이유는 뒤에 문제점에서 설명하도록 하겠다. )  
  사실 현재의 pid를 알아내기 위해서는 task\_of(sched\_entity)가 아니라 직접적으로 미리 알아낸 자료구조 사이의 관계를 이용하여 cfs\_rq->rq->curr->pid를 따라가면 구할 수 있는데, 그렇게 하지 못한 이유 역시 뒤에 문제점을 통하여 설명하도록 하겠다.

1. **실행결과**  
   미리 주어진 execution\_cuncurrent.sh파일에는 ./benchmark\_normal & ./benchmark\_faster & ./benchmark\_slower가 실행되도록 설정되어 있기 때문에, 각각의 이름대로 미리 실행파일을 만들어 두고 실행한다. (추가적으로 benchmark에 시스템 콜 주석부분은 해제)

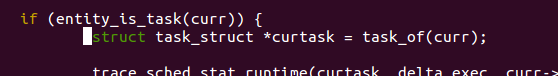


위와 같이 수정 후 faster, normal, slower 순으로 빠르게 task가 처리됨을 알 수 있다. 이는 현재 pid가 faster\_pid인 경우 vruntime이 느리게 증가하여 해당 프로세스가 더 많은 시간동안 cpu를 붙잡고 있기 때문이다. 정해진 time quantum까지, 곧 다시 스케줄링을 하기까지 faster\_pid는 긴 시간 동안, slower\_pid는 비교적 짧은 시간 동안 처리를 하기 때문에 faster\_pid가 더 빨리 끝날 수 밖에 없다. 따라서 실행결과 역시 같은 시간 동안 더 많이 cpu를 붙잡은 프로세스가 빨리 끝나게 된다.

1. **과제를 수행하면서 발생한 문제점 및 해결 방법**

a) **start time 불일치 문제**  
  
생각보다 과제를 빨리 끝낼 수 있었음에도, (이미 완성했었음에도) 생각보다 과제를 오래 끌게 되었다. 그 이유는 내 논리대로 만든 함수가 제대로 된 결과를 보여주지 않았기 때문이다. 그림에서 알 수 있듯, benchmark\_faster는 나머지와 다른 start time을 갖고 있다. 위 그림은 benchmark\_slower도 빠르게 시작했는데, 사실 계속적으로 benchmark\_faster만 먼저 처리가 되고 이후에 normal와 slower가 실행되었었다. (아쉽게도 너무 여러 번 커널 컴파일을 할 탓에 해당 스크린샷은 없다.)  
위 그림은 speed를 0.01, 1, 100으로 맞추지 않고 0.1, 1, 100으로 맞췄을 때 우연적으로 benchmark\_slower가 먼저 실행되어 저런 결과를 갖게 되었다.  
benchmark\_faster가 다른 것에 비해 너무 오랜 시간 cpu를 붙들어서 끝날 때 까지 다른 프로세스가 실행조차 되지 않아 끝나고 난 이후에 실행이 된 것이 아닌가 싶다.  
그래서 start time이 benchmark\_faster의 end time이 되지 않았나 싶다.  
결과적으로 faster\_pid의 speed가 너무 빠르지 않게 각각 1, 100, 10000으로 설정하였더니 원하는 결과가 나오긴 하였다.

b) **Loading initial ramdisk …. 부팅 문제**  
처음 benchmark\_faster가 먼저 시작하는 문제에 대해 함수가 잘못되었다 생각하여,  
기존 cfs\_rq->rq->curr->pid를 통하여 현재 수행중인 프로세스의 pid를 구하는 방법을  
update\_curr함수에 미리 코딩된 부분을 참고하여 수정하였다.



따라서 내가 사용한 변수 선언시 struct task\_struct \*currTask = task\_of(curr);을 사용하였는데, 계속 해서 부팅 시에 Loading initial ramdisk에 멈추는 현상이 발생하였다.  
부트로더에 들어가는 법을 몰라 다시 우분투를 설치 및 컴파일 하는 과정을 여러 번 하였다. 결과적으로 부팅시 shift키를 누르면 다른 커널에 들어갈 수 있음을 알게 되었고,  
앞으로의 커널 에러시 많은 도움이 될 것 같다.  
해당 문제는 \*currTask만 선언 후 입력은 나중에 함으로써 해결되었다. 뭐가 다른지는 잘 모르겠다.

1. **느낀점**  
   생각보다 함수들의 변수가 너무 많긴 하지만, 부분적으로 이해는 가능했다.  
   코딩 시에 에러가 나면 컴파일 시간이 너무 오래 걸려 힘들었다. 또한, 리눅스를 잘 다뤄보지 않아 어느 정도 익숙하긴 하지만 시간을 너무 허비한게 많다.