**운영체제 2차 과제**

정보대학 컴퓨터학과

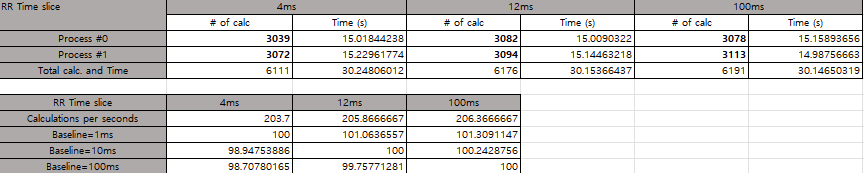
2019320034 김민규

Freeday 0일 사용

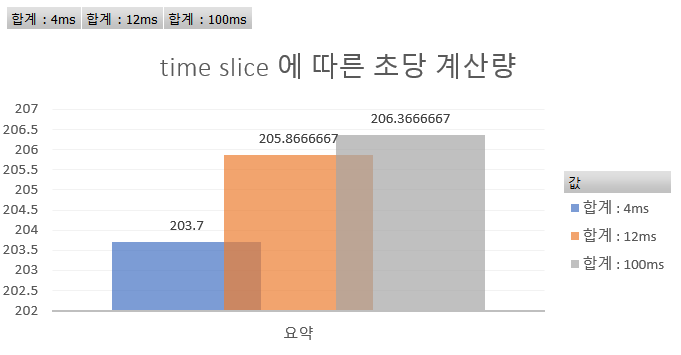
**과제 개요 및 Round Robin 스케줄링에서 time-slice의 영향 설명**

* RR 스케줄링 알고리즘 상에서, time slice 와 행렬 연산량의 상관관계를 분석하고, 이에 따른 context-switching 오버헤드를 측정 및 분석 한다.
* RR 스케줄링은 선점형 스케줄링 알고리즘으로, time slice 에 영향을 많이 받는다.
* Time slice 의 시간이 매우 짧다면 너무 많은 context-switching 이 일어나 overhead가 커지고, time slice 값이 크다면 FCFS 에 가까워지며, 프로세스들의 대기시간이 길어진다.
* 항상 같은 동작(행렬 연산)을 하는 user app 을 구현하여, 고정된 시간 내에서 수행하였을 때, 각기 다른 time slice 값이 실제적으로 얼마나 많은 overhead 를 발생시키는지, overhead 의 크기를 측정한다.

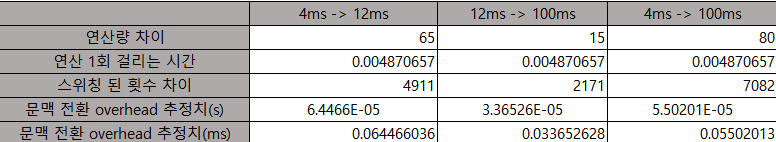
**./cpu 2 30 시, time-slice의 변화에 따른 성능을 관찰한 표 및 결과 분석**

****

**그림 1. Time-slice 의 변화에 따른 성능 분석 표**



**그림 2. Time-slice 의 변화에 따른 초당 계산 량 그래프**

* 각 time slice 를 4ms, 12ms, 100ms 으로 설정하였음.
  + 1ms 으로 설정하였으나, 실제 cpu burst time 이 평균 4ms 으로 측정됨.
* **time slice 대비 성능 분석**
  + 4ms의 경우 12ms 대비 약 1.06%의 성능 하락 발생
  + 12ms의 경우 100ms 대비 약 0.24%의 성능 하락 발생
  + 4ms의 경우 100ms 대비 약 1.3%의 성능 하락 발생
* **time slice 별 연산량, CPU Burst 횟수 측정을 통해 context switching overhead 추정**
* ****

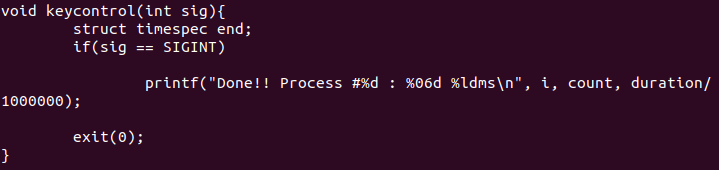
**그림 3. 연산량 차이에 기반한 context switching overhead 추정표**

* + 4ms, 12ms, 100ms 에서 각 발생한 context switching 의 횟수를 측정하고, time slice 별 연산량의 차이를 바탕으로 overhead 추정
    - Time slice 4ms, 12ms 에서의 context switching overhead 추정치 구하기
      * 행렬 연산이 1회 걸리는 시간은 (연산 횟수의 합) / 90 으로 구함.
      * (4ms 에서의 연산량 – 12ms 에서의 연산량) \* (연산 1회 걸리는 시간) / ((4ms 에서의 context switching 횟수) - (12ms 에서의 context switching 횟수))
    - **평균 0.051ms 의 context switching overhead 추정.**

**작성한 모든 소스코드 및 작업 내용에 대한 설명**

* **User Application (cpu.c)**
  + Argument input 받기
  + sched\_attr 구조체, sched\_setattr syscall 로 SCHED\_RR 스케줄러 설정
  + **argument 값 만큼 child process 생성 (fork)**
    - 프로세스가 fork 된 시각을 clock\_gettime 으로 측정 후, 전역변수로 저장
    - Ctrl + C 시 동작할 signal syscall 지정





**그림 4. Ctrl + C kill signal 수행 전, 현재까지의 작업 과정 출력 코드**

모든 child process 에 customized signal handler 를 적용하기 위해, fork 후에 signal syscall 을 사용해 등록함.

* + - 행렬 연산
      * 100\*100\*100 한 번이 끝날 때
        + count 1 증가.
        + 100ms 지났는지 체크, 100ms 지났으면 현재의 count 출력
        + 목표 시간까지 도달했는지 체크(process 생성 시각 – 현재 시각과 argument 에서의 시간을 비교). 도달하지 않았으면 루프
  + Parent process 는 마지막 child process 가 끝날때까지 대기
* **dmesg 출력을 위한 커널 코드 수정 (stats.h)**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**그림 4. Kernel 내부 코드를 수정하여 CPU burst time 측정**

커널 코드 수정 후, 커널 재 컴파일

* **menuconfig 에서 kernel buffer size 를 증가시켜 정확한 dmesg 를 볼 수 있게 수정.**
* **CPU Burst 측정**
  + Single core 로 변경
  + Proc 활용하여 time slice 를 변경해가며, ./cpu 2 30 실행
  + 각 time slice 마다 dmesg 를 추출하여 엑셀로 분석.

**문제점과 해결 과정**

* 낮은 time slice 로 수행 시, buffer 가 작아 처음부터 끝까지의 dmesg 를 얻을 수 없었다.
  + Kernel 내의 buffer size 를 담당하는 constant 를 수정하여 해결함
* Ctrl + C signal 발생 시, 현재까지의 수행 시간을 측정하는 문제
  + 자식 프로세스의 생성 시각을 측정하여 전역 변수로 설정하여 해결함.
* Context switching 의 overhead 를 산출해내는 방식
  + 각 time slice 마다 발생한 switching 의 횟수를 알아내어 이를 바탕으로 수식을 세워 1회의 switching 당 발생하는 overhead 를 추측함.

**의문점**

* **time slice 를 어떤 값으로 설정해도, 가장 가깝고 큰 4의 배수로 CPU Burst time 이 측정되었다.**
  + 1ms 로 설정하였을 때는, 4ms 의 CPU Burst time 을 측정했고, 5ms 로 설정하였을 때는, 8ms 의 CPU Burst time 이 측정되었다.
  + Time slice 의 설정 방식에 대한 의문이 들었음.

**참고자료**

* [clock time](https://mkblog.co.kr/c-measure-time-clock-gettime-function/)
* [fork](https://prod.velog.io/@jungbumwoo/fork-%EB%A5%BC-%EC%95%8C%EC%95%84%EB%B3%B4%EC%9E%90)
* [signal handler](https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=sdug12051205&logNo=221057274601)
* [dmesg buffer size증가]
  + (<https://www.linuxtopia.org/online_books/linux_kernel/kernel_configuration/re07.html>])
* [헤더의 경로 알아낸 사이](https://elixir.bootlin.com/linux/latest/source/kernel/sched/stats.h#L268)트
* [rt-rr 스케줄링 변경](https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=alice_k106&logNo=221170316769)