운영체제 2차 과제 Tutorial

컨텍스트 스위칭에 따른 CPU 스케줄링 오버헤드 분석

고려대학교 운영체제연구실

2023년 5월 2일

과제 내용

학습목표

1. 성능 분석을 위한 프로그램을 구현하여, 프로그래밍 역량을 기른다.
2. 스케줄링 Time Quantum(= Time Slice)에 따른 스케줄링 동작을 살펴보고, 이로 인해 발생하는 오버헤드를 분석하는 방법을 살펴본다.
3. 리눅스 커널 프로그래밍을 통해 성능을 분석하는 방법을 살펴본다.

과제 내용 (파트)

1. 성능 분석을 위한 유저 프로그램 구현 (cpu.c)
2. Time Slice에 따른 성능변화 분석
3. CPU Burst Time 측정

1. 성능 분석을 위한 유저 프로그램 구현

프로세스 스케줄링

프로세스 스케줄링

다양한 프로세스가 컴퓨터 자원(CPU 등)을 요구할 때, 자원의 사용을 시스템의 요구사항에 따른 최선의 방법으로 분배하는 방법.

스케줄링 성능지표

스케줄링 기법의 성능을 평가하는 지표는 CPU 활용율, 처리량, 처리시간, 응답시간, 대기시간 등이 있다.

스케줄링 알고리즘 선택

모든 지표에서 완벽을 보이는 스케줄러를 만드는 것은 현실적으로 불가능.

시스템 요구사항에 따라서 적절한 기법을 선택해야 한다.

참고: CPU Scheduling Criteria - GeeksforGeeks

Round Robin 스케줄링

개요

Time sharing (시분할) 시스템을 기반으로 설계된 선점형 스케줄링 기법.

프로세스들 사이에 우선순위를 두지 않고, 순서대로 Time Slice 만큼 자원을 할당하는 방식의 알고리즘. (Time Slice는 보통 10ms ~ 100ms)

특징

각각의 프로세스는 CPU 자원을 Time Slice (q) 시간 만큼 쪼개어서 사용한다.

존재하는 프로세스의 수가 n일 때, 각 프로세스는 다음 CPU 할당이 돌아오기 까지의 최대 (n – 1) x q의 대기시간을 지닌다.

Time Slice가 충분히 클 경우, FCFS와 거의 동일한 알고리즘이 된다.

장점 및 단점

Response Time이 짧고, 모든 프로세스가 공평하게 반드시 실행된다.

Preemption이 발생할 때, 발생하는 컨텍스트 스위칭 오버헤드가 크다.

성능 분석을 위한 프로그램 구현 (cpu.c)

개요

CPU 사용을 위해 단순 행렬 연산을 수행하는 C 프로그램을 구현한다.
(행렬 연산: 11p에서 설명)

입력 (input)

입력은 CLI에서 프로그램 실행과 함께 제공되어야 한다.

Parameter	Description
Number of Processes	생성할 프로세스의 개수
Time to Execute	각 프로세스의 수행 시간 (ms)

실행 명령어

프로그램은 컴파일 이후, CLI 환경에서 다음과 같은 명령어로 실행되어야 한다.

Command
`./cpu {Number of Processes} {Time to Execute}`

성능 분석을 위한 프로그램 구현 (cpu.c)

참고

프로세스의 개수는 cpu.c 프로그램이 실행 후 생성하는 자식(child) 프로세스의 수를 의미.
즉, 몇번의 fork 시스템 콜이 호출되는지를 의미한다.

각 자식 프로세스는 각자 행렬연산을 수행함.

출력 (실행 중에)

각 프로세스 별로 행렬 연산을 시작한 시점부터 100ms 마다 메시지를 출력한다.

Message Format
`PROCESS #{ProcessNumber} count = {CalculateCount} time = {EpochDuration}`

Parameter	Description
`ProcessNumber`	메시지를 출력하는 프로세스의 식별자
`CalculateCount`	이번 epoch 동안 실행한 행렬 연산 횟수
`EpochDuration`	이번 epoch의 실제 실행시간 (ms)

성능 분석을 위한 프로그램 구현 (cpu.c)

출력 (실행 완료 시)

프로세스 실행이 종료될 때, 프로세스가 연산한 총 횟수와 시간을 출력한다.

Message Format
`PROCESS #{ProcessNumber} totalCount = {TotalCount} time = {ExcutedTIme}`

Parameter	Description
ProcessNumber	메시지를 출력하는 프로세스의 식별자
TotalCount	프로세스가 종료되기까지 수행한 행렬 연산의 총 횟수
ExcutedTime	프로세스가 종료되기까지 수행된 총 시간 (ms)

성능 분석을 위한 프로그램 출력 예시

./cpu 1 1

```
root@osta-VirtualBox:~/ku_os# ./cpu 1 1
Creating Process: #0
PROCESS #00 count = 301 100
PROCESS #00 count = 599 100
PROCESS #00 count = 903 100
PROCESS #00 count = 1208 100
PROCESS #00 count = 1512 100
PROCESS #00 count = 1815 100
PROCESS #00 count = 2120 100
PROCESS #00 count = 2424 100
PROCESS #00 count = 2721 100
PROCESS #00 count = 3022 100
DONE!! PROCESS #00 : 003022 1000
```

./cpu 3 3

```
root@osta-VirtualBox:~/ku_os# ./cpu 3 3
Creating Process: #0
Creating Process: #1
Creating Process: #2
PROCESS #01 count = 310 100
PROCESS #01 count = 311 100
PROCESS #01 count = 619 100
PROCESS #00 count = 294 100
```

```
PROCESS #01 count = 3034 100
PROCESS #01 count = 3334 100
PROCESS #00 count = 2713 100
PROCESS #00 count = 2714 100
DONE!! PROCESS #00 : 002714 3097
PROCESS #01 count = 3640 100
DONE!! PROCESS #01 : 003640 3000
PROCESS #02 count = 3037 100
PROCESS #02 count = 3038 100
PROCESS #02 count = 3039 100
PROCESS #02 count = 3316 100
DONE!! PROCESS #02 : 003316 3000
root@osta-VirtualBox:~/ku_os#
```

./cpu 5 5

```
root@osta-VirtualBox:~/ku_os# ./cpu 5 5
Creating Process: #0
Creating Process: #1
Creating Process: #2
Creating Process: #3
Creating Process: #4
PROCESS #01 count = 310 100
PROCESS #03 count = 304 100
PROCESS #00 count = 299 100
PROCESS #00 count = 300 100
PROCESS #00 count = 301 100
PROCESS #00 count = 302 100
```

```
DONE!! PROCESS #00 : 002367 5003
PROCESS #01 count = 2988 100
PROCESS #01 count = 2989 100
PROCESS #01 count = 2990 100
PROCESS #01 count = 3274 100
DONE!! PROCESS #01 : 003274 5000
PROCESS #03 count = 3590 100
PROCESS #03 count = 3591 100
PROCESS #03 count = 3592 100
PROCESS #03 count = 3885 100
PROCESS #04 count = 2990 100
PROCESS #04 count = 2991 100
PROCESS #04 count = 2992 100
PROCESS #04 count = 3290 100
PROCESS #02 count = 3009 100
DONE!! PROCESS #02 : 003009 5099
PROCESS #03 count = 3886 100
DONE!! PROCESS #03 : 003886 5000
PROCESS #04 count = 3591 100
DONE!! PROCESS #04 : 003591 5000
root@osta-VirtualBox:~/ku_os#
```

추가 점수: 성능 분석을 위한 프로그램 구현 (cpu.c)

추가 점수 과제 개요:

프로세스가 `SIGINT` 시그널에 의해서 종료될 때, 프로세스를 바로 종료하지 않고 지금까지 수행한 연산의 결과를 출력한 뒤에 종료되도록 구현한다.

출력 (`SIGINT` 시그널에 의한 종료 시)

프로세스 정상 종료시 출력하는 출력과 동일함.

주의사항

추가 과제를 수행한 경우 반드시 **보고서에 명시하고 그 구현을 설명**한다.

성능 분석을 위한 프로그램 구현 (cpu.c)

행렬 연산

행렬 연산은 다음 코드를 기반으로 구현한다.

```c
#define ROW (100)
#define COL ROW

int calc() {
    int matrixA[ROW][COL];
    int matrixB[ROW][COL];
    int matrixC[ROW][COL];
    int i, j, k;
    int count = 0;  // 연산 횟수

    for (i = 0; i < ROW; i++) {
        for (j = 0; j < COL; j++) {
            for (k = 0; k < COL; k++) {
                matrixC[i][j] += matrixA[i][j] * matrixB[i][j];
            }
        }
    }

    // 연산 횟수 증가 기준점
    count++;
}
```

성능 분석을 위한 프로그램 구현 (cpu.c)

사용하는 시스템 콜

fork, clock_gettime

- fork를 이용한 프로세스 생성 및 동작 방식의 예제를 충분히 찾아보고 구현할 것

키워드(추가 점수 과제)

시그널 핸들러, `SIGINT` 등

2. Time Slice에 따른 행렬연산 성능변화 분석

Time Slice에 따른 성능변화 분석

개요

스케줄링 방식을 Round Robin으로 변경하고, Time Slice에 따른 작업 성능을 분석한다.

1에서 구현한 cpu.c 내에 스케줄링 정책을 RR로 변경하는 루틴을 추가하여 실험한다.

이를 통해 컨텍스트 스위칭에 따른 오버헤드 분석을 수행한다.

proc

개요

프로세스와 기타 시스템 정보를 계층적 파일 구조로 제공하는 파일 시스템.

시스템이 부팅될 때 생성되고 종료될 때 제거된다.

운영체제가 실행 중에 커널의 속성을 조회하고 변경하는 데 사용할 수 있다.

proc 사용법

READ (변수 읽기) `cat /proc/{file name}`

WRITE (변수 값 변경하기) `echo {value} > /proc/{file name}`

예) file name `/proc/sys/kernel/pid_max`

- 동시에 수행 가능한 최대 프로세스 개수 지정하는 proc 파일/변수

```
root@ostavbox:/# cat /proc/sys/kernel/pid_max
32768
root@ostavbox:/# echo 65535 > /proc/sys/kernel/pid_max
root@ostavbox:/# cat /proc/sys/kernel/pid_max
65535
root@ostavbox:/#
```

RT (Real Time)

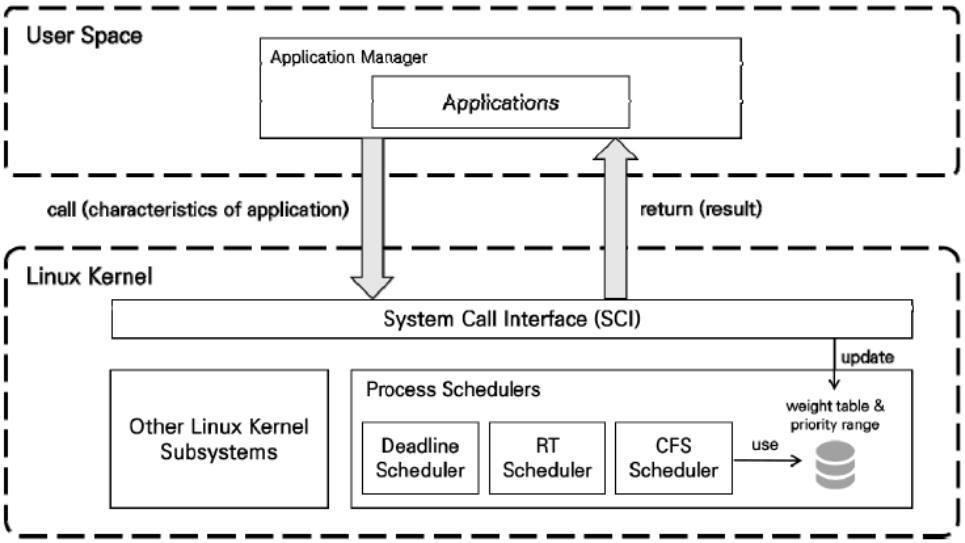
리눅스 스케줄러

리눅스 커널에는 5개의 스케줄러가 존재한다. (Stop, Deadline, RT, CFS, Idle)

유저 태스크는 우선순위에 따라 실시간 태스크와 비실시간 태스크로 나누어진다.

비실시간 태스크는 SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE 중 하나의 스케줄링 정책을 가지고 CFS 스케줄러에서 동작한다.

리눅스 커널은 0 ~ 139의 우선순위를 사용하는데, 우선순위가 0 ~ 99인 태스크는 RT 스케줄러에 의해 100 ~ 139인 태스크는 CFS 스케줄러에 의해 스케줄된다.



RT (Real Time)

RT 스케줄러 정책

RT_FIFO

처음 들어온 작업이 완료되면 다음 들어온 작업을 수행한다

RT_RR ⬅️ 본 과제에서 분석할 스케줄링 정책

Round Robin 기법으로 Time Slice에 따라 작업을 수행한다.

스케줄링 정책 변경

`sched_setattr` 시스템 콜을 사용하여 스케줄링 정책을 지정할 수 있다.

- `sched_setattr` 시스템 콜의 사용법을 확인하고, 1에서 구현한 cpu.c 에서 수정/호출함
- 반드시 fork 이전에 스케줄링 정책을 변경하여 모든 자식 프로세스가 RT-RR을 따르도록 해야함.

RT-RR Time Slice 변경 방법

`/proc/sys/kernel/sched_rr_timeslice_ms` 파일에 원하는 time-slice 기록 (ms)

CPU 코어 설정

본 연구과제에서의 설정

1개의 CPU 코어만 사용하도록 설정하여 Round Robin 스케줄링에 따른 정확한 컨텍스트 스위칭 오버헤드 결과를 얻을 수 있다.

설정 방법

```
#!/usr/bin/bash

cd /sys/fs/cgroup/cpuset
mkdir mycpu; cd mycpu
echo 0 > cpuset.cpus
echo 0 > cpuset.mems
echo $$ > tasks
cat tasks
```

주의사항

이렇게 설정한 CPU 제한은 실행하는 쉘의 pid($$)를 참조하여 지정하였으므로, 해당 세션에서만 적용된다.

적용 여부는 코어 수 감소에 따른 연산 횟수 감소를 분석하여 확인할 수 있다.

실험 방법

수정한 cpu.c를 proc을 통해 timeslice를 바꿔가면서 분석함

실험 방법

수정이 완료된 cpu.c 코드를 컴파일 하고,

proc을 통해 time slice를 1ms, 10ms, 100ms로 변경해가며 실행하여

변화하는 행렬 연산 횟수를 분석한다.

단, 이상치에 의한 실험 결과 오류를 방지하기 위해 각각의 Time Slice 별로 5회 이상 반복 수행하여 분석할 것.

실험 기준 설정 (cpu.c)

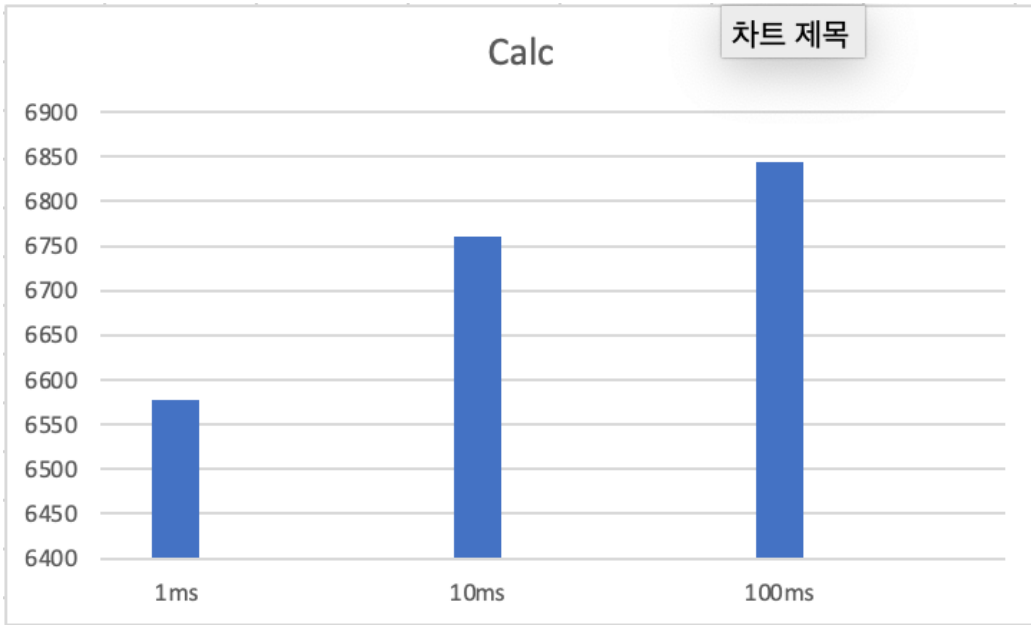
프로세스 개수 : 2, 동작 시간: 30초

실험 결과 (예시)

분석 (계산) 지표

단위 시간(30초) 동안 fork된 각 프로세스가 수행한 행렬 연산 횟수의 합

RR Time slice	1ms	10ms	100ms
	# of calc.	# of calc.	# of calc.
Process #0	3300	3369	3421
Process #1	3278	3392	3423
Total calc.	6578	6761	6844



1ms의 경우, 10ms와 비교 시 약 2.54%의 연산 횟수 하락이 발생

10ms의 경우, 100ms와 비교 시 약 1.21%의 연산 횟수 하락이 발생

3. CPU Burst Time을 활용한 성능 변화 분석

CPU Burst Time 측정의 필요성

이전 실험의 한계

이전의 실험에서는 설정된 Time Slice에 따른 단순 연산 횟수만을 바탕으로 계산을 수행하였음. 따라서 성능 하락치를 계산하는데 우리가 설정한 실행 시간(30초)이 사용됨.

> (실제로는 2.1 과제에서 우리가 어플리케이션 레벨에서 측정한 실행 시간을 사용할 수 있지만, 이는 커널 레벨에서 측정한 결과보다 부정확함)

하지만, 실제로 OS에서 프로세스들은 CPU 자원을 점유하는 시간이 제각기 다르기 때문에 각 프로세스별 연산 시간은 우리가 설정한 값과 상이함.

따라서, **단위 시간 당 프로세스가 수행한 행렬 연산량을 정확히 구해내기 위해서**는 각 프로세스별로 정확한 **CPU Burst Time**을 구하고 **이를 기준으로 성능을 분석**해야 함

커널 코드 분석

커널 코드 분석 방법

bootlin(https://elixir.bootlin.com/)을 이용하여 리눅스 커널 코드를 분석

원하는 버전을 선택한 뒤, 검색창에 검색할 키워드 입력

(예) `sched_info_depart`

sched_info_depart

정의

```
static inline void sched_info_depart(
    struct rq *rq,
    struct task_struct *t
);
```

파라미터

Parameter	Type	Description
*rq	Struct rq	Run Queue
*t	Struct task_struct	CPU 점유를 마친 프로세스의 PCB 구조체

Run Queue

스케줄링 수행을 위해 수행 가능한 상태의 태스크를 관리하는 자료구조

리눅스에서는 ‘sched.h’ 파일 내부에 구조체로 정의됨

sched_info_depart

`rq_clock(rq)`
현재 시간을 ns 단위로 반환

`t->sched_info.last_arrival`
프로세스가 CPU를 점유하기 시작했을 때 시간을 ns 단위로 반환

주의사항: 반드시 cpu.c에서 RT-RR 스케줄링의 priority가 10이 되도록 설정하고, 커널에서는 priority가 10인 프로세스에 대한 로그만 출력한다.

- 스케줄링 정책을 설정한 것처럼 priority를 설정하는 방법을 찾아볼 것

```
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);

        /* yhgo */
        if (t->rt_priority == 10) {
                printk("[Pid: %d], CPUburst: %lld, rt_priority:%u\n", t->pid, delta, t->rt_priority);
        }
}
```

로그 확인

dmesg

커널에서 작성된 로그를 얻을 수 있는 명령어

로그 확인

```
#!/usr/bin/bash

dmesg > log.txt
cat log.txt
```

```
[71725.470964] [Pid: 4129], CPUburst: 100214270, rt_priority:10
[71725.570344] [Pid: 4130], CPUburst: 99380226, rt_priority:10
[71725.570739] [Pid: 4128], CPUburst: 395893, rt_priority:10
[71725.570748] [Pid: 4125], CPUburst: 9482, rt_priority:10
[71725.571057] [Pid: 4129], CPUburst: 309147, rt_priority:10
[71725.670393] [Pid: 4130], CPUburst: 99335015, rt_priority:10
[71725.670432] [Pid: 4125], CPUburst: 39355, rt_priority:10
root@osta-VirtualBox:~/ku_os#
```

CPU Burst Time 측정

개요

더 나아가 유저 어플리케이션의 CPU Burst를 측정하여
해당 프로세스가 실제로 CPU 자원을 사용하는 시간을 바탕으로 행렬 연산의 성능 변화를 분석한다.

실험 방법

커널 내부의 `sched_info_depart()` 함수를 수정하여 CPU Burst Time을 출력하고, Time Slice를 1ms, 10ms, 100ms로 변경해가며 CPU Burst Time를 측정한다.

Outlier(이상치)에 의한 실험 결과 오류 방지를 위해 각각의 Time Slice 별로 최소 5회 실행하고 분석한다.

실험 기준 설정 (cpu.c)

프로세스 개수 : 2, 실행 시간 : 30초

sched_info_depart

프로세스가 CPU 점유를 마쳤을 때 호출되는 함수.

printk() 메소드를 이용해 로그를 출력하도록 수정하고, dmesg를 통해 값을 확인하자

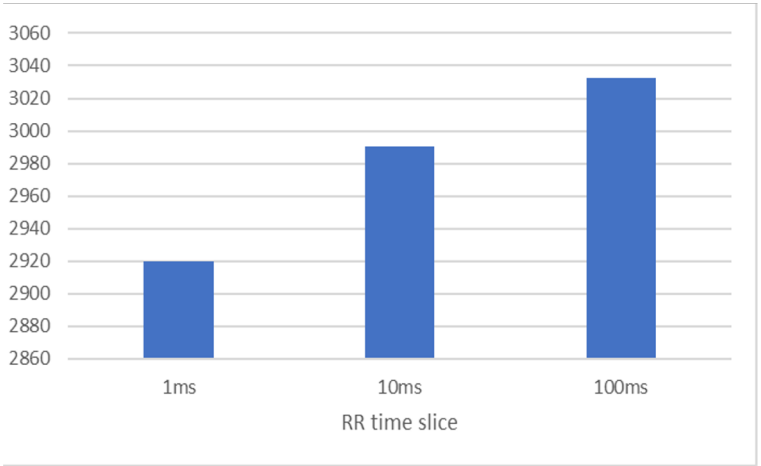
실험 결과 (예시)

분석 (계산) 지표

단위시간(CPU Burst Time) 대비 연산 횟수 (연산 횟수 / CPU Burst Time)

RR Time slice	1ms		10ms		100ms	
	# of calc.	Tims(s)	# of calc.	Time(s)	# of calc.	Time(s)
Process #0	3300	15.04498184	3369	14.9874909	3421	15.00635323
Process #1	3278	14.94775838	3392	15.0309063	3423	15.09578081
Total calc. and Time	6578	29.99274022	6761	30.0183972	6844	30.10213404

RR Time slice	1ms	10ms	100ms
Calculations per second	219.2666667	225.3666667	228.1333333
Baseline=1ms	100	102.7820006	104.0437823
Baseline=10ms	97.29329981	100	101.227629



1ms의 경우, 10ms와 비교 시

약 2.4%의 성능 하락이 발생

10ms의 경우, 100ms와 비교 시

약 1.43%의 성능 하락이 발생

과제 요약

과제 요약

2.1 성능 분석을 위한 프로그램 구현

과제에서 요구한 Spec에 맞추어 cpu.c 소스코드 작성 (4 ~12)

페이지 11에 제시된 행렬 연산 소스코드를 사용할 것

2.2 TimeSlice에 따른 행렬연산 성능변화 분석

1. cpu.c를 수정하여 프로세스 스케줄링 방식을 Round Robin으로 변경 (16 ~ 17)
2. Proc을 이용하여 RR의 TimeSlice 변경 (15 ~ 17)
3. 정확한 실험을 위해 쉘의 최대 프로세서 사용 수를 1로 설정 (18)
4. 실험 진행 및 2 ~ 3 반복 (TimeSlice 변경)
   - 단, 이상치에 의한 실험 결과 오류를 방지하기 위해 각각의 Time Slice 별로 5회 이상 반복 수행하여 분석할 것.
5. 결과 분석 (20): 단위 시간(30초) 동안 fork된 각 프로세스가 수행한 행렬 연산 횟수의 합

과제 요약

2.3 CPU Burst time 측정

1. 커널 내부의 sched_info_depart() 함수 수정 (24 ~ 26)

2. 실험 진행 및 반복 (TimeSlice 변경)
   - 단, 이상치에 의한 실험 결과 오류를 방지하기 위해 각각의 Time Slice 별로 5회 이상 반복 수행하여 분석할 것.

3. 결과 분석 (28): 단위시간(CPU Burst Time) 대비 연산 횟수 (연산 횟수 / CPU Burst Time)

과제 주의사항

제출 내용

제출 목록

1. 보고서
2. 직접 작성한 소스 파일 전체
3. 결과 표와 그래프 원본 파일 (예) 엑셀 등

주의사항

측정 결과를 엑셀 등의 프로그램으로 정리하여 표와 그래프로 정리할 것

도출한 표 및 그래프의 결과를 나름의 근거를 들어 분석

본인이 사용하는 시스템에 따라 예시와 동일한 결과가 나오지 않을 수 있으나
구현을 모두 완료하였고, 결과를 논리적으로 설명한다면 점수 부여

과제 내용에 대한 질문 이외에 구현 세부내용에 대한 질문은 받지 않음.

각 단계별로 수행한 내용까지라도 꼭 제출할 것