CES304 Operating System Project1

202211167 장지원

Contents Table

Description of implementation

1. **MLPQ/MQMS description**
2. **MLPQ/MQMS scheduling implementation (with C code)**

Analysis of benchmark programs

Result Analysis

Description of implementation

1. MLPQ/MQMS description

MLPQ와 MQMS 방식을 사용해서 scheduler를 구현해 보았다.

MLPQ(Multi-level Feedback Queue)란 figure 1과 같이 Multi-level로 이루어져 있는 queue이다. 프로세스들은 생성시에 Top-level Queue에 할당되고, time slice를 다 쓸 때 마다 Queue-level이 내려간다. 이때 timeslice는 하위 Queue 일수록 더 길다.

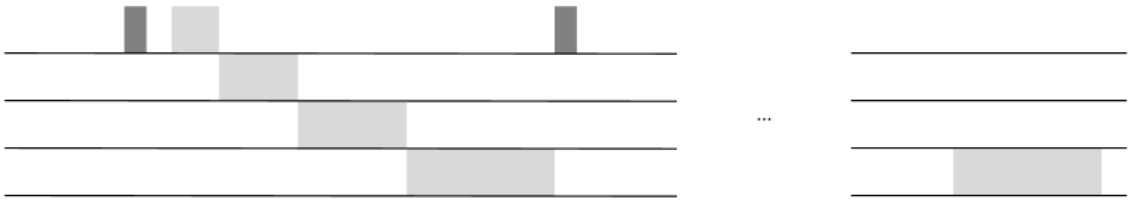

figure 1. Multi-level Feedback Queue

Priority boost 는 일정 주기마다 프로세스들의 Queue-level 을 올려주는 metric 이다. 프로세스의 behavior 가 바뀔 수 있고, 하위 Queue 에 있는 프로세스들에 대해 CPU 할당이 unfair 해질 수 있기 때문에 사용한다.

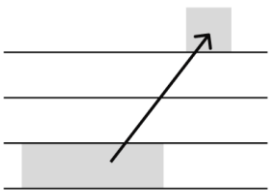

figure 2. Priority boost

MQMS(Multi-Queue Multi-processor Scheduling)는 multi-proc scheduling 방식이다. Queue를 두 개 만듦으로써 cache affinity와 locking에 대한 overload를 해결한다.

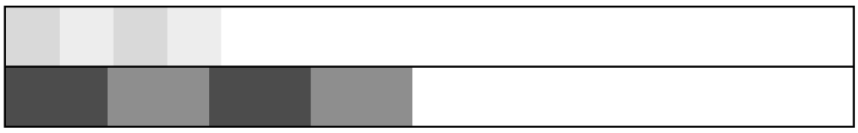

figure 3. Multi-Queue Multi-processor Scheduling

2. MLPQ/MQMS scheduling implementation (with C code)

이번 implementation에서 설정한 hyper parameter들은 다음과 같다. 이 parameter들은 schedbench를 통해 실험적으로(heavy 프로세스들이 CPU1에 할당되는지에 대한 테스트) 결정해본 값들이다.

	CPU0	CPU1
timeslice	10	500
priority boost cycle	50	50

	Queue 4	Queue 3	Queue 2	Queue 1
timeslice	4	8	300	400

figure 4. Scheduler's hyperparameter setting

CPU0(little core)는 Queue 4, 3을 CPU1(big core)는 Queue 2, 1을 담당한다. 그리고 이는 MQMS 특성을 따르며, cache affinity를 보장한다. 각 CPU는 자신의 Queue만 담당하기 때문이다. 또한 timeslice가 긴 프로세스들이 하위 Queue에 배치되기 때문에 자연스럽게 CPU1이 더 heavy한 프로세스들을 담당할 수 있게 된다.

```
if (c->apicid == 0) {
    for (int i = TOP_LEVEL_QUEUE; i > 2 ; i--) {
...
if (c->apicid == 1) {
    for (int i = 2; i > 0 ; i--) {
```

각 프로세스는 가장 상위 Queue에 배정받게 된다. 따라서 프로세스가 initializing되면 CPU0에서 처음 동작하게 된다.

```
p->queue_level = TOP_LEVEL_QUEUE;
```

이후 각 프로세스는 자신이 위치해 있는 Queue의 timeslice 동안 실행된다.

```
while (ticks - start_time <= timeslice) {
       if (p->state == RUNNABLE) {
       } else {
           break;
       }
       p->state = RUNNING;
       swtch(&(c->scheduler), p->context);
}
```

만약 해당 프로세스가 자신의 timeslice를 모두 사용한다면 해당 프로세스의 Queue level을 한 칸 내린다.

```
if (p->state == RUNNABLE) {
    p->queue_level--;
    if (p->queue_level <= 0) {
        p->queue_level = 1;
   }
}
```

각 CPU의 timeslice를 다 쓰면 다시 상위 Queue로 돌아가 작업을 진행한다.

```
CPU0_timeslice_counter += ticks - start_time;
if (CPU0_timeslice_counter >= CPU0_timeslice) { // timeslice 넘기면 다시 실행
    i = TOP_LEVEL_QUEUE;
    CPU0_timeslice_counter = 0;
}
```

일정 주기가 지나면 boosting(모든 proc의 Queue를 상위 Queue로 옮겨주는 과정)을 해준다.

```
CPU0_boosting++;
if (CPU0_boosting >= 50) { // 일정 시간 지나면 boosting
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
         if (p->queue_level > 2)  p->queue_level = TOP_LEVEL_QUEUE;
    }
    CPU0_boosting = 0;
}
```

즉 밑 그림처럼 동작한다. 이 때 boosting은 각 CPU 내의 Queue에서만 진행하는데(ex. CPU1에서 boosting하면 CPU1이 담당하고 있는 프로세스들의 Queue를 2로 올려준다), 이는 cache affinity를 보장하기 위해서이다. 단 이 경우에는 load imbalanced 문제가 생길 수 있는데, 주어진 schedbench에서는 load imbalanced한 상황은 발생하지 않기에 이 설계를 유지하였다.

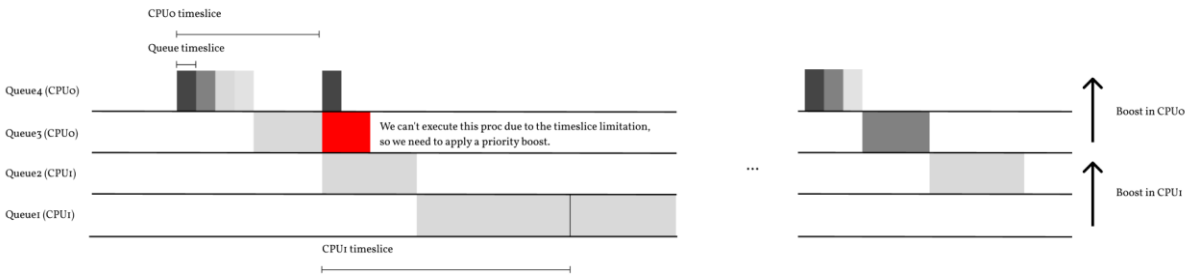

figure 5. MLFQ & MQMS scheduling

이 스케쥴러가 잘 동작하는지에 대한 분석은 schedbench를 분석해본 뒤에 진행해 볼 것이다.

Analysis of benchmark programs

schedbench에서는 여러 프로세스를 생성하고 실행한다. 밑 표는 생성되는 프로세스들이 하는 작업과 그 작업을

진행하는 프로세스들의 type(i.e. light or heavy)을 나타낸 표이다.

case	task	type
1	do_compute(LONG_TIME)	heavy
2	do_compute(SHORT_TIME)	light
3	do_sleep(LONG_TIME*3)	light
4	do_sleep(SHORT_TIME)	light
5	do_sleep_and_compute(1, LONG_TIME, LONG_TIME/2)	heavy
6	do_sleep_and_compute(30, 2, 8)	heavy
7	do_sleep_and_compute(30, 8, 2)	light
8	do_fileread()	light

figure6. task and type

밑 표는 각 작업들이 어떤 역할을 하는지에 관한 설명이 담긴 표이다.

task	description
do_compute(int time)	time 동안 CPU 작업을 진행한다
do_sleep(int time)	time 동안 sleep한다.
do_sleep_and_compute(int num_iter, int sleep_time, int compute_time)	주어진 횟수 동안 sleep 후 CPU 작업을 진행한다.
do_fileread()	파일 읽기, 쓰기 작업을 한다.

figure6. task and description

Result Analysis

결과는 다음과 같았다.

Option	Result
1 (For project submission): Create 3 processes for each task 0-7	schedbench start Time from fork() to wait(): 2019 (0) [cpu 0->1] [tick 480->808] response time: 3, turaround time: 331 (1) [cpu 0->0] [tick 484->525] response time: 7, turaround time: 48 (2) [cpu 0->0] [tick 488->1388] response time: 11, turaround time: 911 (3) [cpu 0->0] [tick 488->500] response time: 10, turaround time: 22 (4) [cpu 0->1] [tick 488->947] response time: 10, turaround time: 469 (5) [cpu 0->1] [tick 500->2350] response time: 22, turaround time: 1872 (6) [cpu 0->0] [tick 500->911] response time: 21, turaround time: 432 (7) [cpu 0->0] [tick 500->962] response time: 21, turaround time: 483 (8) [cpu 0->1] [tick 500->1237] response time: 21, turaround time: 758 (9) [cpu 0->0] [tick 504->554] response time: 16, turaround time: 66 (10) [cpu 0->0] [tick 508->1408] response time: 20, turaround time: 920 (11) [cpu 0->0] [tick 508->532] response time: 20, turaround time: 44 (12) [cpu 0->1] [tick 508->1376] response time: 20, turaround time: 888 (13) [cpu 0->1] [tick 508->2358] response time: 19, turaround time: 1869 (14) [cpu 0->0] [tick 508->903] response time: 19, turaround time: 414 (15) [cpu 0->0] [tick 508->967] response time: 18, turaround time: 477 (16) [cpu 0->1] [tick 508->1666] response time: 18, turaround time: 1176 (17) [cpu 0->0] [tick 512->577] response time: 22, turaround time: 87 (18) [cpu 0->0] [tick 516->1416] response time: 26, turaround time: 926 (19) [cpu 0->0] [tick 516->541] response time: 25, turaround time: 50 (20) [cpu 0->1] [tick 516->1805] response time: 25, turaround time: 1314 (21) [cpu 0->1] [tick 516->2496] response time: 25, turaround time: 2005 (22) [cpu 0->0] [tick 548->895] response time: 57, turaround time: 404

```
(23) [cpu 0->0] [tick 548->972] response time: 32, turaround time: 456
(Child) Total response time: 488 (avg: 20), Total turnaround time: 16422 (avg: 684)
schedbench end
```

figure7. Result

light한 프로세스들은 CPU0에 heavy한 프로세스들은 CPU1에 할당되어 돌아가는 것을 확인할 수 있다.