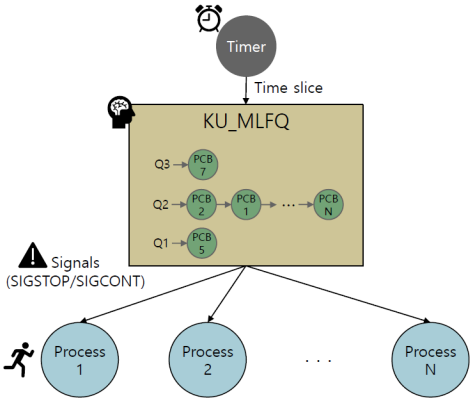
#Assignment 1: KU_MLFQ

개요



MLFQ를 유저 레벨에서 구현해 봅시다.

조건:

Scheduling Metrics:

- Time slice: 1s
- Gaming Tolerance: 2s
- S for Priority Boost: 10s
- A number of cpu cores: 1

System call이나 I/O Interrupt등은 전혀 없으며, OS는 오직 timer interrupt로 주도권을 갖습니다.

각 Job들은 cpu-intensive 하기 때문에, 자기가 가진 자원을 양보하지 않고

시간이 다할 때 까지 쭉 점유하고 있을 것입니다.

Priority Boost로 Job들의 time allotment를 0으로 초기화해 줄 때,

최상위 큐의 job들은 0으로 초기화 하지 않고 **그대로** 유지합니다.

또한, Priority Boost 당시 실행 했던 프로세스의 time allotment가 2이고,

그 job이 최상위 큐에 있다면 우선 순위를 낮춰줍니다.

구현 환경

Windows 10에서 CLion과 WSL 2 (Ubuntu 20.04 LTS)로 구현을 했습니다.

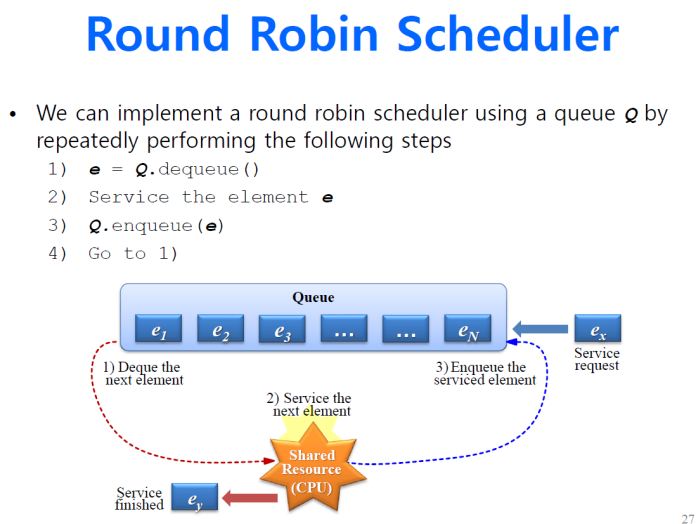
gcc 버전은 9.3.0입니다.

구현 해야할 것들

1. Ready queue & Round Robin

   리스트로 구현을 하되,

   프로세스가 몇 개가 들어올지 모르기 때문에, 동적 할당을 이용해서 구현을 합니다.



2019년도 2 - 1 자료구조: 하영국 교수님

   RR은 2학년 1학기 자료구조 시간 때 배웠으니까

   무리 없이 구현할 수 있을 것 같습니다. 아마도요.

   구현한 큐에서 enqueue와 dequeue연산을 반복하여 RR 알고리즘 역시 구현하겠습니다.

2. Priority 조정하기 (Gaming Tolerance, Priority Boost)

   프로세스의 cpu 독점 방지 및 자원을 아예 받지 못하는 상황을 방지하기 위해,

   cpu를 일정 시간 (time allotment) 이상 점유하면 우선 순위가 한 단계 내려갑니다.

   그리고 일정 시간이 지나면 (S) 모든 프로세스들의 우선 순위가 가장 높게 배정됩니다.

3. Interrupt handler for SIGALRM

   대망의 Interrupt handler.

   이 안에 5가지 rule들을 구현하면 되겠습니다.

4. Process Control Block 설정하기

struct pcb 내에 어떤 정보들이 들어갈 것인지 잘 생각해 봐야 합니다.

설계

스케줄링을 담당하게 되는 것은 바로 SIGALRM의 핸들러입니다.

itimerval로 Interval Timer를 설정해서, 일정한 간격 X마다 timer interrupt를 발생 시켜서

OS (과제에서는 핸들러) 에게 간섭하고 정리할 기회를 줍니다.

프로세스들은 주어진 ku_app을 execl하며, 실행 즉시 waiting 상태입니다.

프로세스의 실행은 여기서는 kill(pid, SIGCONT)로 대체하며,

다른 프로세스로의 전환은 실행 중인 프로세스에게 SIGSTOP을 보내고,

실행 할 프로세스에게 SIGCONT를 보내는 것입니다.

주어진 running time 동안의 실행이 끝나면,

모든 자식 프로세스에게 SIGINT를 보내고 끝이 나게 됩니다.

Timer Interrupt Frequency (HZ)

강의에서는 보통 1ms, 4ms를 주기로 OS가 timer interrupt를 통해서

OS가 참견할 기회를 얻는다고 했습니다.

이번 과제에서는 이 주기를 몇 초로 놔야 할까요?

저는 1초로 놔도 충분하다고 생각합니다.

물론 실제로는 상황은 더 다양하지만, 이번 과제에서는 time slice가 1인 만큼

Timer Interrupt를 1초로 두고 구현하겠습니다.

Scheduling Algorithm

기본적으로 rule 5개와 교수님께서 제시해주신 방향을 토대로 구현했습니다.

중간 중간 구현을 하면서 정의가 필요한 부분은 직접 정의를 했습니다.

Rule들이 겹칠 때:

항상 모든 것들은 순탄하게만 흘러가지 않습니다.

아마 저뿐만 아니라 다른 분들도 부딪혔을 문제 같습니다.

Priority Boost와 Game Tolerance가 겹칠 때.

Q3	A	B	C	A	B	C					
Q2							A	B	C	A	
Q1											
Time	1	2	3	4	5	6	7	8	9	10	11

3개의 프로세스 A, B, C가 동시에 시스템에 arrive했다고 가정하겠습니다.

Time allotment = 2s, 그리고 Priority Boost를 위한 S = 10s로 주어져 있습니다.

A를 보시면, A는 9 ~ 10초에 실행되고 나서 Q1로 우선 순위가 떨어져야 합니다. (Game tolerance)

공교롭게도 우선 순위가 떨어져야 하는데, 이때는 S = 10이므로 priority boost할 시간입니다.

즉, 두 rule이 중첩됐을 때는 Gaming Tolerance를 **먼저** 처리해주고

그 후에 Priority Boost를 적용합니다. 이때 Q2와 Q1의 원소들을 Q3에 넣어주어야 할 텐데요.

이때 어느 큐 먼저 최상위 큐로 넣어주어야 할까요?

어느 큐를 먼저 최상위 큐로 넣어주어야 할까요?

프로세스가 3개인 경우

그래서 다시, 프로세스 A가 cpu 점유 시간 2초를 다 채워서 Q1으로 내려간 상황입니다.

이때 큐 상황은 이렇습니다.

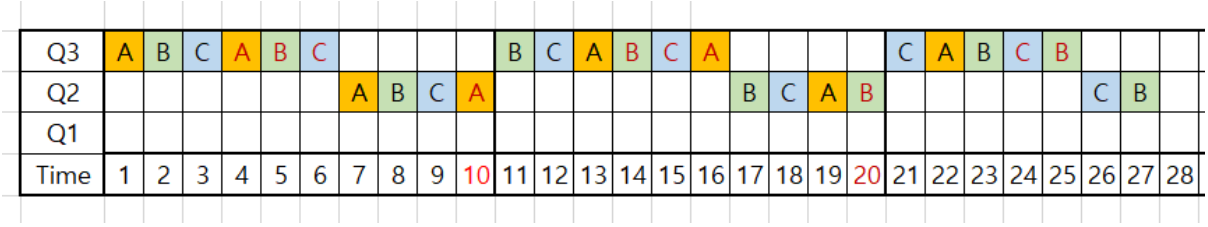
Q3:

Q2: B → C

Q1: A

이때 Priority Boost를 해준다면, Q2부터 차례로 enqueue하고 Q1도 넣어줍니다.

Q3: B → C → A인 상황이 되겠습니다.



앞으로 T = 40까지의 시간을 그려본다면 이렇게 그려질 것 같습니다.

글씨가 빨간 프로세스들은 time allotment를 다 채웠다는 뜻입니다.

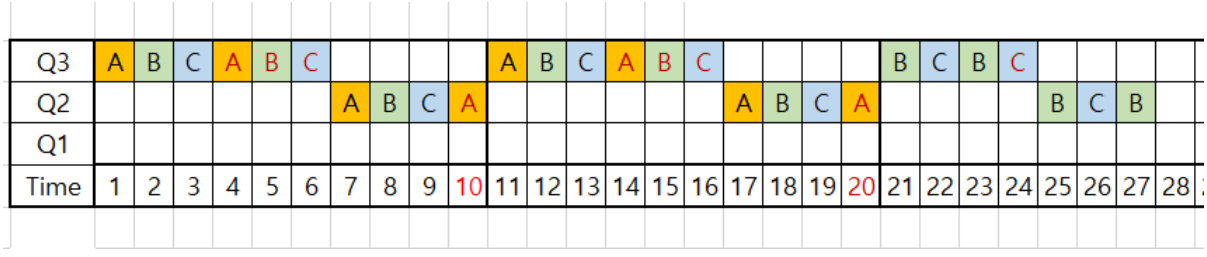
가정대로 A, B, C가 모두 0에 동시에 arrive하고,

각각의 CPU Time이 8초, 10초, 9초라고 가정하겠습니다.

이때의 Turnaround Time은 이렇게 됩니다:

$$T_{turnaround} = \frac{22 + 27 + 26}{3} = 25$$

만약 Q1을 Q2보다 먼저 올려줬다면 어떨까요?



Q1에 있던 A가 Q3으로 Q2에 있는 B, C보다 먼저 올라가면서, A가 B와 C보다 더 많이

CPU 자원을 점유하고 있는 것을 볼 수 있습니다.

$$T_{turnaround} = \frac{20 + 27 + 26}{3} = 24.3$$

솔직히 제가 예상했던 결과와는 다르게 나와서 좀 슬프네요.

하지만 프로세스들이 더 많아진다면 어떨까요?

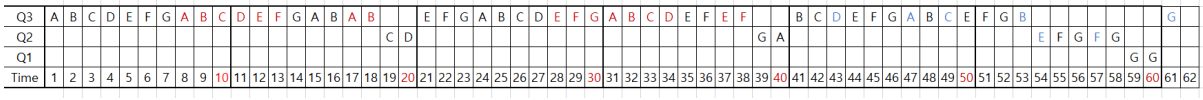
프로세스가 7개인 경우

프로세스가 이번엔 A, B, C, D, E, F, G까지 무려 7개가 있습니다.

그리고 역시 동시에 시스템에 arrive했다고 가정하겠습니다.

A, B, C, D, E, F, G의 실행 시간을 각각 8, 9, 7, 6, 9, 10, 12라고 하겠습니다.

1. Q2를 Q1보다 먼저 Q3으로 넣어줬을 때:

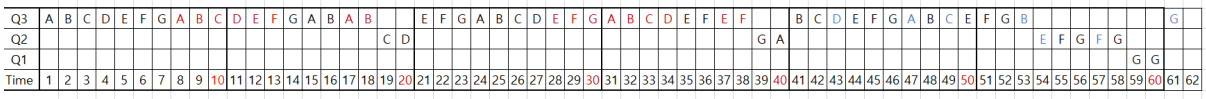


사진이 너무 작네요. 여기서 파란색으로 표기한 프로세스들은

실행을 마치고 종료되었다는 뜻입니다.

$$T_{turnaround} = \frac{47 + 53 + 49 + 43 + 54 + 57 + 61}{7} = 52$$

2. Q1을 Q2보다 먼저 Q3으로 넣어줬을 때:



네. 놀랍게도 1.의 경우와 같습니다. 이 예제에서는 실행 도중에 프로세스의 우선 순위가 1까지

내려간 적이 없기 때문입니다.

프로세스의 개수 × Time Slice의 길이 > S (Priority Boost) 이기 때문인 것 같습니다.

실제로 Solaris같은 운영체제의 경우, 60개의 큐가 default로 있으며,

제일 높은 우선 순위의 time slice는 20ms부터

제일 낮은 우선 순위의 time slice는 몇 백ms라고 합니다. Boost도 대략 1초마다 한다고 합니다.

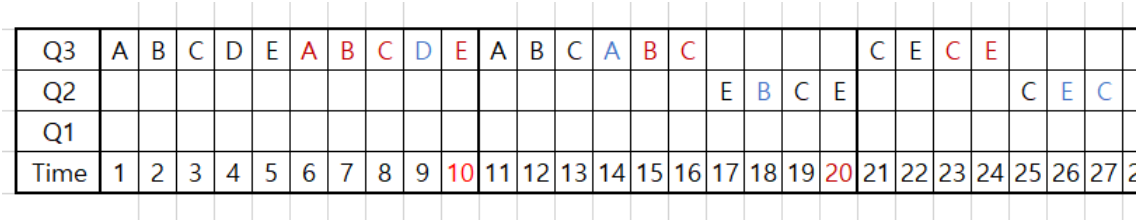
마지막으로 프로세스가 5개인 경우를 해보겠습니다.

프로세스가 5개인 경우

마찬가지로 프로세스 A, B, C, D, E가 동시에 시스템에 arrive했다고 하겠습니다.

이번엔 A, B, C, D, E의 실행 시간은 각각 4, 5, 9, 2, 7 입니다.

1. Q2를 Q1보다 먼저 Q3으로 넣어줬을 때:



$$T_{turnaround} = \frac{14 + 18 + 27 + 9 + 26}{5} = 18.8$$

2. Q1을 Q2보다 먼저 Q3으로 넣어줬을 때:

Q3	A	B	C	D	E	A	B	C	D	E	A	B	C	A	B	C					E	C	E	C				
Q2																	E	B	C	E					E	C	C	
Q1																												
Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	2

$$T_{turnaround} = \frac{14 + 18 + 27 + 9 + 25}{5} = 18.6$$

또 근소한 차이로 Q1을 먼저 넣어 주는 것이 더 빨랐습니다.

프로세스가 더 많은 경우를 보겠습니다.

프로세스가 26개인 경우

A, B, C, ..Z = 가 각각

10, 2, 5, 9, 6, 7, 8, 1, 2, 4, 7, 9, 7, 15, 24, 12, 17, 16, 5, 1, 15, 16, 8, 13, 12, 23의 값을 가진다고

가정하겠습니다. Gantt Chart를 그리면 무려 200이 넘어가는 긴 실행 시간 때문에,

직접 엑셀에 그려가기엔 너무 무리인 탓에, turnaround time을 구하는 것을 자동화했습니다.

그래서 scheduler 내에서 각각 job의 turnaround time을 구하게 했습니다.

1. Q2를 Q1보다 먼저 Q3으로 넣어줬을 때:

```
254: [0] P: 2 T: 1
174+ 28+ 95+ 164+ 113+ 136+ 150+ 8+ 34+ 78+ 138+ 170+ 141+ 226+ 254+ 202+ 240+ 236+ 109+ 20+ 231+ 237+ 166+ 218+ 210+ 253+
avgTime: 806.200000
```

대략 806.2초가 걸립니다.

2. Q1을 Q2보다 먼저 Q3으로 넣어줬을 때:

```
254: [0] P: 2 T: 1
174, 28, 95, 164, 113, 136, 150, 8, 34, 78, 138, 170, 141, 226, 254, 202, 240, 236, 109, 20, 231, 237, 166, 218, 210, 253,
avgTime: 806.200000
```

사실 어떤 순서든 이 과제에서는 크게 중요하지 않은 것 같습니다.

제대로 알아보려면 더 다양한 데이터 셋들을 두고 해야겠지만,

그래도 저는 Q2부터 먼저 넣어주는게 맞다고 생각합니다.

Q1의 프로세스들은 이미 cpu자원을 많이 잡아 먹은 전적이 있기에,

CPU-bound하다고 볼 수 있습니다. 따라서 Q2의 프로세스들을 Q1보다 먼저,

Q3에 넣어주는 게 독점을 방지하는데 더 효과적인 것 같습니다.

구현

Process Control Block (PCB)

```
typedef struct pcb {
    pid_t pid;
    double arrivalTime;
    int pname, priority, timeAllot;
}PCB;
```

```
double getTime() {
    struct timeval t;
    gettimeofday(&t, (void*)0);
    return (double)t.tv_sec + (double)t.tv_usec / 1e6;
}
```

프로세스의 PCB에는 process id뿐만 아니라, 스케줄링에 필요한 다양한 정보들이 들어가 있습니다.
cpu 누적 점유 시간 (time allotment), 현재 우선순위 (priority), 프로세스의 이름 (pname),
그리고 도착 시간 (arrivalTime)이 있습니다.

프로세스의 이름은 ku_app을 실행할 때 주는 인자 (알파벳)입니다.
도착 시간은 fork() 해주고 나서, 부모 프로세스에서 바로 시간을 재줍니다.

Scheduler

`void scheduler(int sig)`

main()에서 itimerval의 간격을 1초로 했습니다.
그리고 전역 변수로 `volatile int alarmCount`를 주어서,
timer가 몇 번 expire되었는지 세고 이를 바탕으로 현재의 elasped time을 잽니다.

Scheduler에서 가장 마지막으로 실행한 프로세스의 pid를 저장하고,
timer interrupt가 발생했을 때 현재 실행한 프로세스를 정지시킵니다. (SIGSTOP)
그리고, 현재 비어있지 않은 큐중 가장 높은 큐에서 가장 앞에 있는 프로세스를 실행시킵니다.
이 때, 실행 시키고 time allotment가 2가 되었다면 바로 우선 순위를 내려줍니다.

이렇게 한 프로세스를 실행 시키고 나서, 만약 현재 시간이 S의 배수이면,
Priority Boost를 실행합니다. 최상위 큐를 제외한 나머지 큐들의 time allotment를 0으로 초기화하고
우선 순위를 Q2 → Q1 순으로 Q3에 넣어줍니다. 이때 방금 실행한 프로세스가
time allotment가 2가 되어서 최상위 큐에서 우선 순위가 떨어졌던 프로세스라면
그 프로세스에 대한 boost는 시행하지 않습니다.

이 스케줄링은 ku_mlfq가 실행될 때 주어진 time slice만큼만 실행됩니다.

번외: Tuning MLFQs

- Time slice
  - 1 second
- Gaming tolerance
  - Time allotment: 2 seconds
- *S*
  - 10 seconds

Scheduling Metrics가 주어졌었습니다.

Turnaround time을 구하는 과정도 자동화 했는데,

이 상수들을 조금만 다르게 줘서 어떻게 Average turnaround time이 변하는지

조금만 알아보겠습니다.

이전의 데이터 셋을 활용하겠습니다.

26개의 프로세스와 각각 실행 시간을 전과 같이 두겠습니다.

1. Gaming tolerance: 3s, S: 10s

```
253: [Z] P: 2 T: 1
254: [Z] P: 2 T: 2
181, 28, 93, 164, 115, 136, 150, 8, 34, 79, 138, 165, 141, 225, 252, 203, 238, 235, 109, 20, 230, 240, 167, 218, 209, 254,
avgTime: 806.400000
```

2. Gaming tolerance: 4s, S: 10s

```
254: [0] P: 2 T: 1
174, 28, 96, 164, 113, 132, 150, 8, 34, 79, 139, 170, 141, 225, 254, 204, 240, 235, 109, 20, 230, 236, 166, 218, 210, 253,
avgTime: 805.600000
```

3. Gaming tolerance: 2s, S: 15s

```
253: [0] P: 0 T: 0
254: [Z] P: 0 T: 0
170, 28, 90, 164, 112, 133, 145, 8, 34, 73, 135, 171, 137, 226, 253, 205, 239, 235, 110, 20, 230, 236, 166, 224, 211, 254,
avgTime: 801.800000
```

4. Gaming tolerance: 2s, S: 7s

```
251: [0] P: 1 T: 0
252: [Z] P: 1 T: 1
253: [Z] P: 2 T: 0
254: [0] P: 2 T: 0
175, 28, 95, 167, 114, 136, 153, 8, 34, 79, 139, 170, 141, 224, 254, 201, 240, 233, 109, 20, 234, 237, 164, 222, 213, 253,
avgTime: 808.600000
```

5. Gaming tolerance: 3s, S: 15s

```
252: [0] P: 1 T: 2
253: [Z] P: 0 T: 0
254: [0] P: 0 T: 0
178, 28, 83, 154, 118, 137, 155, 8, 34, 79, 141, 168, 143, 225, 254, 200, 240, 236, 111, 20, 229, 237, 166, 218, 210, 253,
avgTime: 805.000000
```

제가 시도해본 결과 중에서는 3번인, S를 15초로 늘리는 것이 average turnaround time측면에서는

제일 좋게 나왔습니다.

단순히 어느 한 쪽을 줄이고, 한 쪽을 늘리고 함으로써

average turnaround time을 고려하는 것은 어려운 일인 것 같습니다.

실제로 무슨 프로세스들이 실행될지도 모르고, 너무나도 다양한 상황들이 존재하기 때문입니다.

Solaris 처럼 큐가 그저 3개가 아닌, 60개가 존재할 수도 있고
time slice를 큐마다 다르게 줄 수도 있습니다.
혹은 아예 동적으로 time slice를 시간이 지남에 따라 서서히 바뀌게도 할 수 있고,
FreeBSD처럼 우선 순위와 time allotment로 관계식을 만들어서 그에 맞게 조정할 수 있습니다.
CFS는 nice와 vruntime을 사용하기도 했었습니다. 이렇게 성능 개선을 위해서
OS마다 다양한 policy들을 채택해서 scheduling을 구현했습니다.

직접 Gantt chart를 그려가면서 이 다음에 누굴 실행시키고,
누구에게서 자원을 뺏어와야 하나와 같은 고민들은 머리가 아프지만
그래도 정말 유익하고 재밌는 경험이었습니다.
현재 OS의 스케줄러를 만드신 분들이 얼마나 고생을 하셨는지 감히 짐작도 못하겠습니다.
이상입니다.

201811244 컴퓨터공학과 김창엽