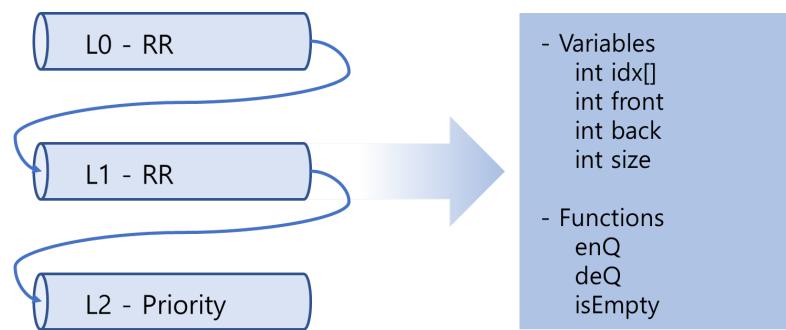


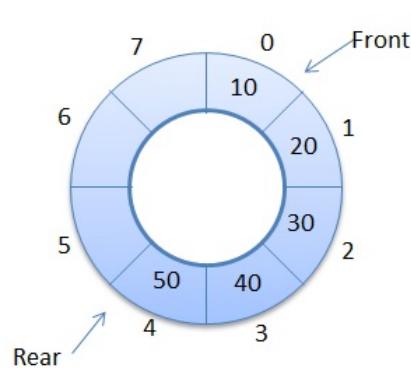
## 1. Design

### 1-1. MLFQ



- 각각의 Q를 Process table에 저장되어 있는 index를 저장하는 Circular Q로 구현
- Process 탐색 시 Front부터 Back 까지 Q 탐색
- Q 자료구조를 위한 기본 변수 및 함수 필요

#### 1-1-1. Circular Q인 이유



xv6에서 최대 생성 가능한 process의 수가 `NPROC 64`로 정해져 있기 때문에, 효율적인 구현을 위해서 Circular Q로 구현

Circular Q는 Front와 Rear(Back)이 내부적으로 순환하는 구조

이미지 출처 : <https://swengineer7.tistory.com/33>

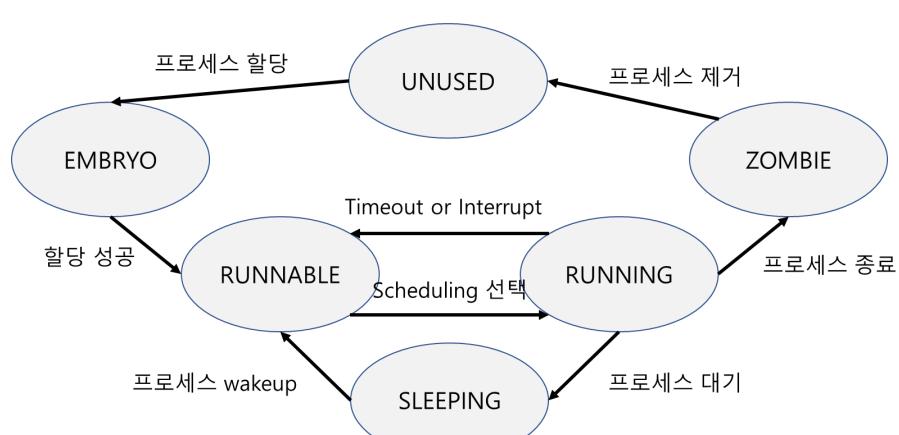
[Circular Q의 구조]

#### 1-1-2. $L_0, L_1, L_2$ 의 scheduling

- $L_0, L_1$  : RR방식의 scheduling
    - for - loop이용하여 RUNNABLE한 process탐색
      - 발견 시 scheduling
      - 발견하지 못하면 다음 Level의 Q로 이동하여 탐색
  - $L_2$  : Priority를 고려한 FCFS scheduling
    - Q 전체를 1회 탐색하면서 RUNNABLE한 process 중에서 제일 우선순위가 높은(priority의 숫자가 제일 작은) process 탐색
      - 발견한 process의 위치를 저장
      - 발견하지 못하면 scheduling 재시작
    - 기억했던 process의 위치까지 이동하여, process scheduling
- ★  $L_2$ 의 다른 구조 디자인
- 숫자가 작은 priority를 우선적으로 scheduling하는 구조  
⇒ Min-Heap의 구조로  $L_2$  설계 가능
  - Min - Heap을 선택하지 않은 이유
    - $L_2$ 를 위해서 새로운 자료구조를 만들어 주는 것이 비효율적
    - Priority Boosting 할 때에,  $L_2$ 에 들어온 순서를 보존하여  $L_0$ 에 올려야 하는데, Heap구조는 그렇게 하기 힘듦

### 1-1-3. Q 내부의 Process

xv6의 process 자료구조는 다음과 같은 총 6개의 상태를 가질 수 있다.



- UNUSED - 사용하지 않는 상태, 비어있는 process를 나타낸다.
- EMBRYO - allocproc을 통해 UNUSED 상태였던 공간을 확보한 상태이나, 아직 완벽히 process로 기능하지는 않은 상태이다.
- RUNNABLE - scheduling의 대상이 될 수 있는 process의 상태
- RUNNING - scheduling의 대상이 되어 CPU를 점유하고 있는 process의 상태
- SLEEPING - I/O 발생 등의 interrupt를 기다리는 등의 process 대기 상태
- ZOMBIE - exit 호출로 process가 종료된 상태 (그러나 아직 완전히 제거되지 않은 상태)

이 중에서 MLFQ에는 일반적인 process에 대해서 오직 **RUNNABLE**과 **SLEEPING**만 존재한다.

즉, RUNNABLE인 process가 Q에서 선택되어 RUNNING이 되면 Q에서 나오게 된다.

이렇게 구현하는 것이 Q에서 쓸데없는 공간을 차지하지 않으면서, scheduler의 의미를 살릴 수 있는 구현이라고 생각했다. (자세한 내용은 Trouble Shooting에서 설명)

## 1-2. Process

xv6의 Process 자료구조에 추가적인 변수 필요하다.

```
struct proc {  
    ...  
    ...  
    int qlv;  
    int priority;  
    int consumed_tq;  
    ...  
};
```

- `qlv` : 어떤 Level의 MLFQ에 속해있는지 나타내는 변수
- `priority` : Process의 우선순위를 나타내는 변수,  $[0, 3] \in \mathbb{N}$
- `consumed_tq` : Process가 소모한 time quantum을 나타내는 변수

### 1-3. Priority Boosting

- Priority Boosting이란?

Process의 Starvation을 막기 위해서 실행되는 mechanism이다.

- Priority Boosting을 위해서 고려할 점

1. 100ticks의 계산
2. Q 내부 Process사이의 stability 유지

#### 1-3-1. 100ticks의 계산

xv6에는 global ticks를 저장하는 `ticks` 변수가 존재한다.

다음과 같은 구현들을 고려할 수 있다.

1. `ticks` 를 직접 이용하여 Boosting을 위한 계산하기

- `sys_uptime()` 을 통해서 호출 가능
- xv6에 전체에 대한 변수

⇒ ticks를 직접적으로 이용하는 것은 안정성에 문제가 발생할 수 있다.

2. `ticks` 를 본딴 다른 변수를 이용하여 Boosting을 위한 계산하기

- System 시작 시에, `ticks` 와 같은 값을 갖도록 변수 생성

- ticks 와 같이 올라가도록 변수 조절

⇒ ticks 와는 별도로 control 가능하고, Boosting을 위해서만 사용하는 변수로 이용

결론적으로 ticks 를 본딴 다른 변수를 이용하여 Boosting하는 타이밍을 계산했다.

### 1-3-2. Stability 유지

Q에 들어온 순서가 유지되기 위해서는,  $L_1$ 의 front의 원소가  $L_0$ 의 back에 삽입되어야하고, 이는  $L_2$  역시 마찬가지이다.

## 1-4. Scheduler Lock

우선적으로 실행되기 원하는 process를 위한 system call이자, interrupt 129번을 통해서 호출될 수 있는 함수

### 1-4-1. 우선적으로 실행

- 다른 process보다 우선적으로 실행된다는 의미는, scheduler가 이 process만을 선택한다는 의미
- MLFQ에서 process를 찾기 이전에, scheduler Lock을 호출한 process가 있는지 확인
  - 만약 존재한다면, process를 scheduling
  - 만약 존재하지 않는다면, MLFQ에서 scheduling 할 process 탐색

### 1-4-2. System call vs Interrupt

명세의 '두 시스템 콜은 Interrupt를 통해 실행될 수 있어야 합니다.'라는 부분에 대한 해석을 어떻게 해야할 지에 대해서 고민했다.

1. System call이 Interrupt를 통해서 호출되어야 한다.
  - System call은 64번 Interrupt를 통해서 호출이 되고 있다.
  - 다른 Interrupt를 통해서 system call을 호출하는 과정은 이상하다.
2. system call로도 호출될 수 있고, 또는 Interrupt를 통해서도 호출 될 수 있다.
  - 기본적으로 프로그램에서 system call을 통해서 함수를 호출했다.
  - 추가적인 경우로 명시적인 Interrupt 호출(`asm("int $129");`)을 통해서도 호출할 수 있다.

⇒ 2번의 해석이 더 알맞은 해석이라고 생각해서 2번과 같이 구현했다.

### 1-4-3. VALID vs NOT VALID

NOT VALID한 상황을 제외하고는 모두 VALID한 상황으로 가정한다.

- |   |   |
|---|---|
| <ul style="list-style-type: none"> <li>• <b>NOT VALID</b> : 함수의 악의적인 실행, 예외적인 상황<br/>pid, time quantum, 현재의 Q level을 출력하고, process를 강제로 종료한다.           <ul style="list-style-type: none"> <li>◦ Scheduler Lock이 된 상황에서 또 다시 호출 되는 경우               <ul style="list-style-type: none"> <li>▪ Process가 악의적으로 CPU를 점유하여 다른 process 가 처리되지 못하게 막는 경우로 간주</li> </ul> </li> <li>◦ 인자로 전달되는 PASSWORD가 틀린 경우               <ul style="list-style-type: none"> <li>▪ Process가 우선적으로 처리되어야 할 자격을 증명하지 못한 경우로 간주</li> </ul> </li> </ul> </li> </ul> | <ul style="list-style-type: none"> <li>• <b>VALID</b> : 예외적이지 않은 일반적인 상황, 악의적이지 않은 상황<br/>함수가 실행되면 다음과 같은 procedure 진행           <ul style="list-style-type: none"> <li>◦ Priority Boosting을 위한 변수 0으로 초기화</li> <li>◦ Scheduler Lock인 상태와 Process를 나타내는 변수 설정</li> <li>◦ Process의 time quantum 0으로 초기화</li> <li>◦ Process의 Q level을 -1로 간주</li> </ul> </li> </ul> |
|---|---|
- ⇒ 우선순위가 높은  $L_0$ 보다 더 상위의 Q인  $L_{-1}$ 에서 process가 Scheduling 되는 것처럼 생각

### 1-4-4. 종료조건

종료조건은 아래의 세 가지 조건 중 하나를 만족한 경우이다.

#### 1. global ticks가 100이 된 경우

- Scheduler Lock을 호출한 process를 MLFQ의  $L_0$ 의 맨 앞으로 이동
- Priority Boosting을 실행

#### 2. Scheduler Unlock이 호출된 경우

- Scheduler Lock을 호출한 process를 MLFQ의  $L_0$ 의 맨 앞으로 이동
- 해당 process의 time quantum과 priority를 기본값으로 초기화

### 3. Scheduler Lock을 호출한 process가 종료( `exit()` )된 경우

- Priority Boosting 혹은 Scheduler Unlock을 실행하지 않고, MLFQ scheduling 방식으로 돌아간다.

## 1-4-5. 고려사항

### • Sleep process에 대한 처리

Scheduler Lock을 호출한 process가 SLEEPING 상태가 된 경우에는 종료조건으로 처리하지 않는다.

왜냐하면, process가 Scheduler Lock을 걸고 할당된 100ticks를 활용하는 의도적인 동작이라고 판단했기 때문이다.

물론 CPU가 `sleep()` 도중에 사용되지 않는 비효율성 문제가 부각될 수는 있으나, I/O 등의 처리를 위해서 `sleep()` 을 호출할 때에 Lock이 해제되는 것은 바람직하지 않다고 생각했다.

그래서 SLEEPING일 때는 단지 시간을 보내다가 global ticks가 100이 되기 전에 깨어난다면, scheduling을 통해서 작업을 한다.

만약 SLEEPING 상태에서 global ticks가 100이 되면, 종료조건 1에 의해서 Priority Boosting이 발생한다.

## 1-5. Scheduler Unlock

우선적으로 실행되기 원하는 process를 해제하기 위한 system call이자, interrupt 130번을 통해서 호출될 수 있는 함수

### 1-5-1. ‘우선적으로 실행’해제

- 다른 process보다 ‘우선적으로 실행’을 해제한다는 의미는, 더 이상 scheduler가 이 process만을 선택하지 않고 MLFQ를 통해서 process를 선택한다는 의미

### 1-5-2. System call vs Interrupt

명세의 ‘두 시스템 콜은 Interrupt를 통해 실행될 수 있어야 합니다.’라는 부분에 대한 해석을 어떻게 해야할 지에 대해서 고민했다.

#### 1. System call이 Interrupt를 통해서 호출되어야 한다.

- System call은 64번 Interrupt를 통해서 호출이 되고 있다.
- 다른 Interrupt를 통해서 System call을 호출하는 과정은 이 상하다.

#### 2. System call로도 호출될 수 있고, 또는 Interrupt를 통해서도 호출될 수 있다.

- 기본적으로 프로그램에서 System call을 통해서 함수를 호출했다.
- 추가적인 경우로 명시적인 Interrupt 호출(`_asm__("int $130");`)을 통해서도 호출할 수 있다.

⇒ 2번의 해석이 더 알맞은 해석이라고 생각해서 2번과 같이 구현했다.

### 1-5-3. VALID vs NOT VALID

크게 2가지, 세부적으로 3가지의 상황이 존재한다.

- NOT VALID : 함수의 악의적인 실행, 예외적인 상황

- NOT VALID

- pid, time quantum, 현재의 Q level을 출력하고, process를 강제로 종료한다.

- 인자로 전달되는 PASSWORD가 틀린 경우

- Process가 우선적으로 처리되어야 할 자격을 증명하지 못한 경우로 간주

- VOID

- Scheduler Unlock이 된 상황(Lock이 되지 않은 상황)에서 또 다시 호출 되는 경우

- Unlock은 Lock인 상황에서만 호출이 되어야한다고 생각했기 때문에 정상적이지 않은 경우로 간주

- 그러나 언제 Priority Boosting이 발생해서 Lock이 해제되는지 사용자가 알기 어렵기 때문에 아무것도 하지 않는다.

- Scheduler Unlock을 호출한 Process가 Scheduler Lock을 호출한 Process가 아닌 경우

- Scheduler Unlock을 할 권한이 있는 Process는 오직 Scheduler Lock을 호출한 Process이기 때문에, 올바르지 않은 경우로 간주

⇒ Scheduler Lock 상황에서는 Lock을 건 Process만 호출되기 때문에 실제로는 발생하지 않는 상황이이다.

- VALID : 예외적이지 않은 일반적인 상황, 악의적이지 않은 상황

함수가 실행되면 다음과 같은 procedure 진행

- Priority Boosting을 위한 변수 0으로 초기화
  - Scheduler Lock인 상태와 Process를 나타내는 변수 설정
  - Process의 time quantum 0으로 초기화
  - Process의 Q level을 -1로 간주
- ⇒ 우선순위가 높은  $L_0$ 보다 더 상위의 Q인  $L_{-1}$ 에서 process가 Scheduling 되는 것처럼 생각

## 1-6. Set Priority

Process의 priority를 임의로 설정할 수 있는 system call이다.

### 1-6-1. VALID vs NOT VALID

NOT VALID한 상황을 제외하고는 모두 VALID한 상황으로 가정한다.

- **NOT VALID** : 예외적인 상황
  - 경고문구를 화면에 출력한다.
  - Priority를 설정하고자 하는 process가 존재하지 않는 경우
  - 범위  $[0, 3] \in \mathbb{N}$ 에 포함되지 않는 경우
- **VALID** : 일반적인 상황
  - 함수가 실행되면 다음과 같은 procedure 진행
  - pid에 맞는 process의 우선순위를 변경한다.

### 1-6-2. 고려사항

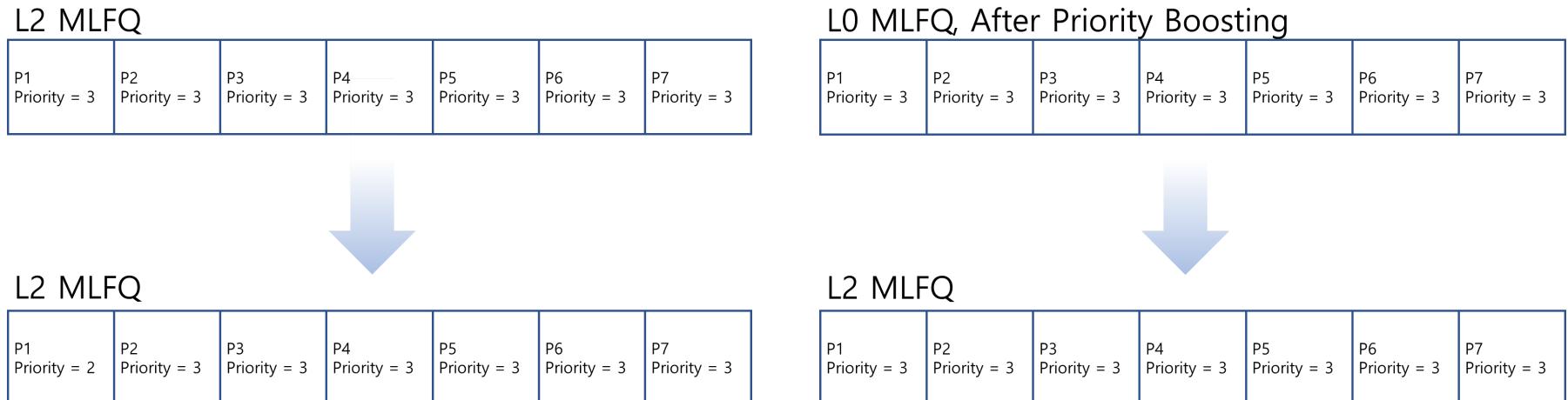
#### • Scheduling Fairness

일반적으로 프로세스 사이에는 다른 프로세스끼리 어떠한 조작을 하는 것이 바람직하지 않다.

그러나, 과제의  $L_2$ 가 제일 먼저 도달한 process를 먼저 scheduling하는 구조를 갖기 때문에 다른 process는 starvation이 필연적으로 발생한다. 이를 막기 위해서 Priority Boosting이 실행되기는 하지만, 그것만으로는 부족하다고 생각하여, 다른 프로세스 사이에서도 priority를 수정할 수 있도록 구현했다.

아래의 예시를 보도록 하자.

아래의 예시의 가정은  $P_1, \dots, P_7$ 의 process만 존재하고 다른 Q에는 RUNNABLE한 process가 존재하지 않고, 추가적인 process가 생성되지 않는다는 가정이다.



Process가 처음  $L_2$ 에 도착했을 때, 왼쪽 위의 그림과 같았다면, 이 때는 모두 priority가 동등하기 때문에 RR과 같은 방식으로 scheduling이 된다.

그러다가 시간이 지나면 왼쪽 하단과 같은 그림이 되는데,  $P_1$ 이 time quantum 8을 먼저 소비하게 되고,  $L_2$ 의 policy에 따라서 priority의 숫자가 1감소하게 되어, 이 때부터는  $P_1$ 이 우선하게 계속 scheduling이 된다.

이 상태에서 Priority Boosting이 되면 오른쪽 위의 그림과 같은 상태가 된다.

$L_0$ 는 RR 방식으로 scheduling이 될 것이고, 시간이 지나면 오른쪽 아래와 같은 그림이 된다.

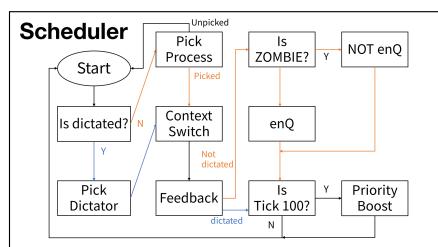
이는 왼쪽 위의 그림과 같기 때문에 결과적으로  $P_1$ 의 CPU 점유율이 다른 process에 비해서 비정상적으로 높아진다.

이러한 process 사이의 비대칭적인 CPU 점유율은 공평한 scheduling이 아니라고 생각했고, 어떤 process에서 다른 process의 priority를 조절하여 위와 같은 상황을 방지하거나 최소한 여러 process가 돌아가면서 CPU를 점유할 수 있도록 구현했다.

## 1-7. Scheduler

위의 모든 디자인은 제대로 된 **Scheduler**를 구현하기 위한 밑작업이었다.

### 1-7-1. Scheduler의 Sequence



1. Dictated인 경우, Dictator에게 CPU에 할당
2. Dictated가 아닌 경우, MLFQ에서  $L_0$ 부터 process를 탐색하며, 각 Q의 policy에 맞는 RUNNABLE한 process를 골라서 CPU를 할당
3. Scheduling을 한 뒤의 process에 대한 feedback 작업 → Dictator는 과정 4 생략
4. Process의 상태에 따라서 MLFQ로 다시 enQ
5. Global ticks가 100인지 확인하여, Priority Boosting을 실행
6. 1의 과정 반복

### 1-7-2. 고려사항

#### 1. Dictator

- Dictator를 Scheduler에 우선적으로 할당하는 방법에 대해서는 [1-4-1](#)에서 설명했다.
- Dictator는 **MLFQ보다 우선적으로 Scheduling의 대상이 되기 때문에 MLFQ에 존재하지 않는다.** 논리적으로  $L_{-1}$ 인 Q에 존재한다고 생각하지만, 실질적으로 그러한 Q는 존재하지도 않을 뿐더러, dictator는 어떤 Q에도 들어있지 않다. 즉, RUNNABLE 혹은 SLEEPING인 process임에도 Q에 있지 않은 매우 특별한 존재이다.
- Dicated는 dictator에 의해서 scheduler가 지배당하는 상태를 의미한다.

#### 2. Pick Process

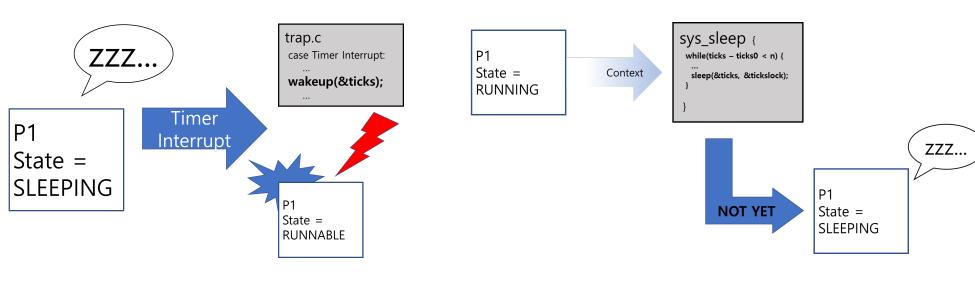
- 각 Q의 policy에 맞는 정책을 적용하여  $L_0, L_1, L_2$  순서대로 process를 탐색하는 함수이다.
- Scheduling의 대상이 되는 process를 찾으면 deQ한 뒤 return해준다.

#### 3. Time quantum

- Scheduling 이후 process가 사용한 time quantum을 1 올려준다.  
⇒ Scheduling의 대상이 되어서 단 한 번이라도 Context Switch가 발생했다면, time quantum을 1 올려준다.
- 이 때에 SLEEPING이었던 process도 `sys_sleep()` 함수의 구조 때문에 time quantum이 올라간다.

이러한 디자인을 채택한 이유는 다음과 같다.

##### 1. `sleep()`의 원리



`sleep()`이 작동하는 구조는 다음과 같다.

1. Timer Interrupt가 발생한다.
2. `trap.c`에서 `wakeup()`을 통해서 RUNNABLE process로 변한다.
3. Scheduling의 대상으로 선택되어 RUNNING으로 변한다.
4. Context에서 아직 지정한 sleep 시간을 채우지 못했으면 다시 SLEEPING이 된다.

⇒ 즉, SLEEPING인 process도 RUNNING이 잠깐씩 되기 때문에 scheduling에 선택되고, 이는 CPU를 할당받은 것이기 때문에 사용한 time quantum이 올라간다.

##### 2. SLEEPING 이유를 알 수 있음

Scheduler에서 어떤 process가 Context Switch 이후에 SLEEPING이 된 이유가 `sys_sleep()`에서 지정된 시간을 채우지 못해 서인지, 어떤 일을 한 뒤에 process가 명시적으로 `sleep()`을 호출한 것인지 알 수 없다.

##### 3. `sys_sleep()`의 안전성 보장

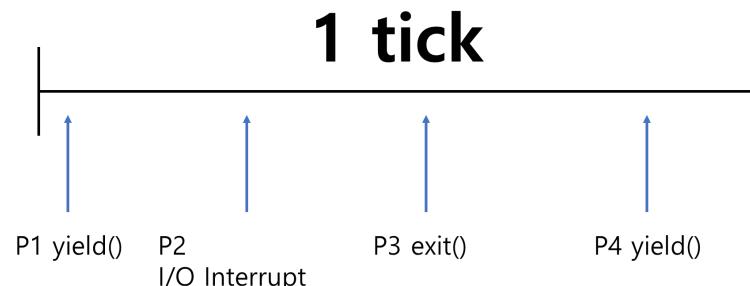
System call 내부에서 process의 변수를 직접적으로 설정하는 것은 위험하다고 생각했다. system call은 사용자가 호출할 수 있고, 이를 이용해서 process에 대한 악의적인 조작이 가능할 수도 있기 때문이다.

#### 4. ZOMBIE

- Context Switch 이후에 다시 Q에 넣기 위해서 process의 state를 확인해야 한다.
- ZOMBIE state는 이미 `exit()` 을 호출하여 종료된 process이기 때문에 enQ할 필요가 없다.
- Dictator는 Q에 넣을 필요가 없기 때문에 이 과정을 생략한다.

#### 5. Ticks

나의 Scheduler 구현에는 총 3가지의 ticks와 관련된 변수가 존재한다.



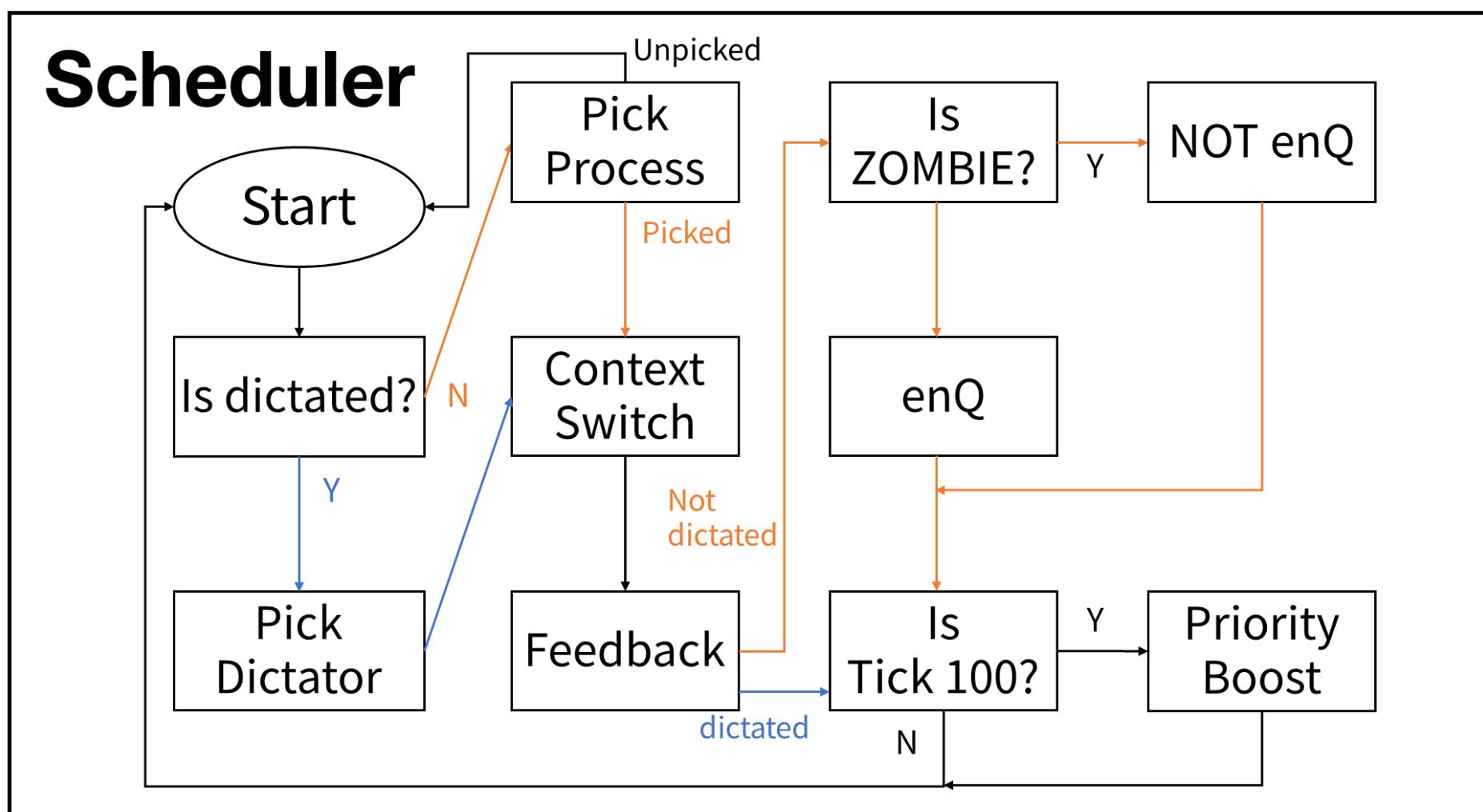
- (Real) `ticks` : 실제로 xv6에서 ticks 단위를 세는 변수
  - `global_ticks` : Priority Boosting을 위해서 ticks와 같이 증가하는 변수
  - (process) `consumed_tq` : process가 사용한 ticks를 나타내는 변수
- 그러나 `consumed_tq` 와 `ticks` 는 올라가는 속도가 다르다.

왜냐하면, `ticks` 는 Timer Interrupt를 기준으로하고, `consumed_tq` 는 하나의 process가 scheduler에 잡힌 횟수를 기준으로 하기 때문이다.

이것은 위와 같이 1 tick 안에서 여러 개의 process가 scheduling 될 수 있는 상황을 반영한 것이다.

## 2. Implement

### 2-1. Scheduler()



위의 Scheduler 디자인과 실제 작성 코드를 보면서 진행한다. (코드의 line number가 실제 line number가 아님에 유의)

#### 2-1-1. Is dictated?

```
1 // Is dictated?
2 if (is_dictated)
```

Scheduler가 Lock인 상태인지 확인한다.

- Yes인 경우, Pick Dictator로 이동
- No인 경우, Pick Process로 이동

## 2-1-2. Pick Dictator

```
1 // Pick Dictator
2 // Scheduler is dictated
3 // by process that called schedulerLock().
4 proc_idx = scheduler_dictator;
```

Scheduling process로 dictator를 고른다.

- `proc_idx` 는 Scheduling process를 나타내는 변수
- `scheduler_dictator` 는 dictator를 나타내는 변수
- Context Switch로 이동한다.

```
1 p = &ptable.proc[proc_idx];
```

## 2-1-3. Pick Process

```
1 // Loop over mlfq looking for process
2 // to run with respect to lv from 0 to 2.
3 for (lv = 0;lv < 3;lv++) {
4     // PickProc
5     proc_idx = pickProc(lv);
6     if (proc_idx != -1) {
7         found = 1;
8         break;
9     }
10 }
```

```
1 // There is no RUNNABLE process.
2 if (!found) {
3     release(&ptable.lock);
4     continue;
5 }
```

```
1 p = &ptable.proc[proc_idx];
```

MLFQ를  $L_0$ 부터  $L_2$ 를 보면서 process를 고른다.

- `pickProc()`을 통해서 process를 찾은 경우 Context Switch로 이동한다.
- 그렇지 않은 경우 다시 처음으로 돌아간다.

### pickProc() 구현

```
1 int pickProc(int lv) {
2     if (isEmpty(lv)) return -1;
3
4     int sz = mlfq[lv].size;
5
6     if (lv == 2) {
7         int proc_priority = USELESS;
8         int mlfq_idx = 0;
9         // Look through mlfq[2] to find process
10        // who has highest priority.
11        for (int i = 0;i < sz;i++) {
12            int proc_idx = deQ(lv);
13            enQ(lv, proc_idx);
14            if (ptable.proc[proc_idx].state != RUNNABLE) {
15                continue;
16            }
17            if (proc_priority > ptable.proc[proc_idx].priority) {
18                proc_priority = ptable.proc[proc_idx].priority;
19                mlfq_idx = i; // This will be picked.
20            }
21        }
22        if (proc_priority == USELESS)
23            return -1;
24        int proc_idx = deQ(lv);
25        for (int i = 0;i < mlfq_idx;i++) {
26            enQ(lv, proc_idx);
27            proc_idx = deQ(lv);
28        }
29        return proc_idx;
30    }
31
32    else {
33        for (int i = 0;i < sz;i++) {
34            int proc_idx = deQ(lv);
35            if (ptable.proc[proc_idx].state != RUNNABLE) {
36                enQ(lv, proc_idx);
37                continue;
38            }
39            return proc_idx;
40        }
41    }
42    return -1;
43 }
```

Line을 기준으로 설명

- Line 24, Q가 비어있는지 체크
- Line 28:52,  $L_2$ 에서 process를 선택
- Line 33:43,  $L_2$ 를 돌면서 제일 높은 우선순위인 process 위치 탐색
- Line 44:45, process를 찾았는지 확인
- Line 45:51, 찾았던 process 위치까지 `deQ` 와 `enQ` 반복
  - $L_2$ 내의 process의 순서 보존
- Line 54:64,  $L_0, L_1$ 에서 process 선택
- Line 55:61, `deQ` 와 `enQ`를 반복하면서 RUNNABLE인 process 탐색
- return 값은 `ptable.proc` 의 index

## 2-1-4. Context Switch

```
1 // Context Switch
2 // Switch to chosen process. It is the process's job
3 // to release ptable.lock and then reacquire it
4 // before jumping back to us.
5 c->proc = p;
6 switchuvvm(p);
7 p->state = RUNNING;
8 swtch(&(c->scheduler), p->context);
```

- process의 State를 RUNNING으로 바꾸어준다.
- `swtch()`를 통해서 Context Switch 실행.

## 2-1-5. Feedback

```
1 switch kvm();
2
3 // Feedback
4 p->consumed_tq++;
5 if (!is_dictated) {
6
7     // Adjust Qlv + 1 or priority - 1.
8     if (p->consumed_tq >= lv * 2 + 4) {
9         if (lv == 2) {
10             p->priority =
11                 p->priority - 1 >= 0 ? p->priority - 1 : 0;
12         }
13         lv = lv + 1 < 2 ? lv + 1 : 2;
14         p->qlv = lv;
15         p->consumed_tq = 0;
16     }
}
```

- process의 `consumed_tq` 를 1 증가시켜준다.
- dictator가 아닌 경우에 line 5:16을 실행한다.
  - `consumed_tq` 가 해당 Q에서 있을 수 있는 시간을 넘은 경우
    - Process가  $L_2$ 에 있는 경우 priority를 1 감소시킨다.
    - 그렇지 않은 경우 다음 level의 Q로 이동시킨다.
    - 그에 따라 process의 `qlv` 를 변경한다.
    - `consumed_tq` 를 0으로 초기화한다.

## 2-1-6. Is ZOMBIE? & enQ / NOT enQ

```
1 // Is ZOMBIE?
2 if (p->state != ZOMBIE) {
3     enQ(lv, proc_idx);
4 }
5 }
```

- ZOMBIE가 아닌 경우, `enQ` 한다.  
⇒ Q에 넣는 process는 RUNNABLE과 SLEEPING 뿐이다.

## 2-1-7. Is Tick 100? & Priority Boost

```
1 // Is tick 100?
2 if (global_ticks >= 100) {
3     // Do priority Boosting
4     global_ticks = 0;
5     priorityBoost();
6 }
```

- Priority Boosting을 위한 변수인 `global_ticks` 가 100을 넘었는지 확인한다.
- 만약 넘었다면, `global_ticks` 를 0으로 초기화하고, `priorityBoost()` 를 실행한다.  
`global_ticks`  
`global_ticks` 는 `ticks` 와 같이 올라가지만 scheduler의 작동 순서때문에 RUNNABLE한 process가 존재하지 않으면, 즉 MLFQ 내부에 있는 모든 process가 SLEEPING 상태라면 `global_ticks` 가 100이 넘어가도 Priority Boosting이 일어나지 않는다.

이렇게 구현한 이유는 SLEEPING process만 있을 경우에 Priority Boosting을 하는 것은 의미가 없다고 생각했기 때문이다.

### priorityBoost() 구현

```
1 void priorityBoost(void) {
2     // If scheduler has been dictated,
3     // dictator should be enQed in front of mlfq[0].
4     if (is_dictated) {
5         int sz = mlfq[0].size;
6         enQ(0, scheduler_dictator);
7         for (int i = 0; i < sz; i++) {
8             enQ(0, deQ(0));
9         }
10        is_dictated = 0;
11        scheduler_dictator = -1;
12    }
13    // priorityBoost
14    for (int lv = 0; lv <= 2; lv++) {
15        int sz = mlfq[lv].size;
16        for (int i = 0; i < sz; i++) {
17            int boost_idx = deQ(lv);
18            ptable.proc[boost_idx].qlv = 0;
19            ptable.proc[boost_idx].priority = 3;
20            ptable.proc[boost_idx].consumed_tq = 0;
21            enQ(0, boost_idx);
22        }
23    }
24 }
25
26 // It is impossible that enQ() blocked because of full Q.
27 // int -> void?
28 int enQ(int lv, int idx) {
29     mlfq[lv].proc_idx[mlfq[lv].back] = idx;
30     mlfq[lv].back++;
31     mlfq[lv].back %= NPROC;
32     mlfq[lv].size++;
33     return idx;
34 }
```

### Line을 기준으로 설명

- Line 4:12, dictator가 있는 경우 작업
  - Dictator를  $L_0$ 의 맨 앞으로 이동
  - Dictator 해제
- Line 14:23, 각 Q를 돌면서 변수 초기화
- Line 17, Q에서 하나의 process `deQ`
- Line 18, process의 `qlv` 초기화
- Line 19, process의 `priority` 초기화
- Line 20, process의 `consumed_tq` 초기화
- Line 21, process를  $L_0$ 에 `enQ`

## 2-2. userinit() & fork()

### Real Line Number 기준

```
177 // Set process default property
178 p->state = RUNNABLE;
179 p->priority = 3;
180 p->qlv = 0;
181 p->consumed_tq = 0;
182 int uproc_idx = p - (ptable.proc);
183 // EnQ process to mlfq[0]
184 enQ(0, uproc_idx);
```

각각 process가 처음으로 생성되어 RUNNABLE이 되는 부분이다.

생성된 Process의 추가적인 변수들에 대해서 초기화를 진행해주고, `enQ` 한다.

[`userinit()`]

```
254 // Set process default property
255 np->state = RUNNABLE;
256 np->priority = 3;
257 np->qlv = 0;
258 np->consumed_tq = 0;
259 int uproc_idx = np - (ptable.proc);
260 // EnQ process to mlfq[0]
261 enQ(0, uproc_idx);
```

[`fork()`]

### 2-2-1. allocproc()

모든 process는 `allocproc()`을 통해서 생성된다. 그럼에도 `allocproc()`에서 변수를 초기화하거나 Q에 넣지 않은 이유는 다음과 같다.

#### Real Line Number 기준

1. `allocproc()`에서는 아직 process가 EMBRYO 상태이기 때문이다.

⇒ Q에는 오직 RUNNABLE과 SLEEPING인 process만 존재

```
114 found:
115     p->state = EMBRYO;
116     p->pid = nextpid++;
```

[`allocproc()`에서 state]

2. `allocproc()`을 성공했다하더라도, `userinit()`과 `fork()` 안에서 추가적으로 통과해야하는 작업이 있기 때문에 불필요한 작업이라고 생각했기 때문이다.

```
155 if ((p->pgdir = setupkvm()) == 0)
156     panic("userinit: out of memory?");
```

```
229 // Copy process state from proc.
230 if ((np->pgdir = copyuvvm(curproc->pgdir, curproc->sz)) == 0) {
231     kfree(np->kstack);
232     np->kstack = 0;
233     np->state = UNUSED;
234     return -1;
235 }
```

## 2-3. exit()

### Real Line Number 기준

```
307 // Jump into the scheduler, never to return.
308 curproc->state = ZOMBIE;
309 if (scheduler_dictator == curproc - ptable.proc) {
310     is_dictated = 0;
311     scheduler_dictator = -1;
312 }
```

- `exit()`은 process를 종료하는 함수이다.
- 만약, dictator가 `schedulerUnlock()`을 명시적으로 호출하지 않고 process를 종료하는 경우 문제가 발생할 수 있기 때문에, dictator를 해제한다.

## 2-4. getLevel()

```
1 int getLevel(void) {
2     struct proc* p = myproc();
3     return p->qlv;
4 }
```

- 현재 실행 중인 process가 있는 Q의 Level을 return하는 함수
- 나의 구현에서는 “**실행 중인 process가 있는 Q**”가 존재하지 않는다.  
⇒ 실행 중인 process는 곧 RUNNING 상태의 process인데, Q에는 오직 RUNNABLE과 SLEEPING만 존재하기 때문  
개념적으로 **process가 RUNNABLE 상태 일 때 있었던 Q**로 해석했다.
- Dictator는 예외적이다.  
⇒ Dictator 역시 Q에 존재하지 않지만, 그 특성상 MLFQ보다 우선하여 scheduling된다.  
MLFQ의 개념을 확장하여  $L_0$ 보다 우선하는  $L_{-1}$  Q가 존재하고, 그 Q에서 dictator가 선택된다고 해석하여 **dictator는 -1을 return**한다.

## 2-5. setPriority()

```
1 void setPriority(int pid, int priority) {
2     acquire(&ptable.lock);
3     int found = 0;
4     int proc_idx = -1;
5     for (int lv = 0; lv < 3; lv++) {
6         int sz = mlfq[lv].size;
7         for (int i = 0; i < sz; i++) {
8             proc_idx = deq(lv);
9             enQ(lv, proc_idx);
10            if (ptable.proc[proc_idx].pid != pid)
11                continue;
12            found = 1;
13            break;
14        }
15        if (found)
16            break;
17    }
18    // NOT VALID condition
19    if (priority < 0 || priority > 4) {
20        fprintf("Priority should be in range from 0 to 3\n");
21        release(&ptable.lock);
22        //exit();
23    }
24    // NOT VALID condition
25    else if (!found) {
26        fprintf("Not Running Process pid on Memory\n");
27        release(&ptable.lock);
28        //exit();
29    }
30    else {
31        // VALID condition
32        ptable.proc[proc_idx].priority = priority;
33        release(&ptable.lock);
34    }
35 }
```

- 인자로 주어진 **pid**의 **priority**를 설정하는 함수
- Line 5:17, 인자의 **pid**와 일치하는 **pid**를 갖는 process를 탐색
- Line 19:23, 인자의 **priority**가 정해진 범위가 아닌 경우에 오류문구를 출력한다.
- Line 25:29, 인자의 **pid**가 정해진 범위가 아닌 경우에 오류문구를 출력한다.
- Line 30:34, 찾은 process의 **priority**를 설정한다.
- NOT VALID한 실행에 대한 처리
  - 오류문구만을 출력하고, process에 대해서 종료하지 않는다.
  - 종료하지 않는 이유는 다음과 같다.
    1. NOT VALID한 경우가 특정 process에 CPU를 과하게 할당하는 등의 OS에 대한 안정성을 침해하지 않는다.
    2. 만약 임의의 pid를 brute-force로 찾아서 공격하는 행위라고 해도, 그것이 VALID한지 아닌지에 대한 판단을 할 수 없다.
    3. MLFQ가  $L_0$ ,  $L_1$ 에서 RR로 scheduling되기 때문에 process가 전혀 실행되지 않는 경우는 발생하지 않는다.

## 2-6. schedulerLock()

```
1 void schedulerLock(int password) {
2     struct proc* p = myproc();
3     acquire(&ptable.lock);
4
5     // NOT VALID condition
6     if (is_dictated) { // Block trying to dictate again.
7         fprintf("pid = %d, time quantum = %d, current queue level = %d\n",
8             p->pid, p->consumed_to, p->qlv);
9         scheduler_dictator = -1; // dictator out.
10        is_dictated = 0;
11        release(&ptable.lock);
12        exit();
13    }
14    else if (password != PASSWORD) { // It is Wrong Password.
15        fprintf("pid = %d, time quantum = %d, current queue level = %d\n",
16            p->pid, p->consumed_to, p->qlv);
17        release(&ptable.lock);
18        exit();
19    }
20    else {
21        // VALID condition
22        global_ticks = 0; // global ticks set to 0.
23        scheduler_dictator = p - ptable.proc; // variable to schedule one process.
24        is_dictated = 1; // variable to represent scheduler dictated.
25        p->consumed_tq = 0;
26        p->qlv = -1;
27        // Dictating scheduler can be considered
28        // that process is in the most highest MLFQ which level is - 1.
29        release(&ptable.lock);
30    }
31 }
```

- MLFQ보다 우선하여 처리할 process를 위한 함수
- Line 6:13, 이미 dictator가 존재한다면 그것은 자신일 수 밖에 없고 그러한 호출은 NOT VALID 하다고 판단하여 pid, time quantum, 현재 Q level을 출력하고 process를 종료한다. 이 때 dictator이므로, dictator를 해제한다.  
⇒ Dictator의 악의적인 CPU의 강제점유라고 간주하여 process를 종료한다.
- Line 14:19, 인자로 전달되는 **password**가 틀린경우, NOT VALID 하다고 판단하여 pid, time quantum, 현재 Q level을 출력하고 process를 종료한다.
- Line 20:30, VALID한 호출로 다음과 같은 작업을 진행한다.
  - Priority Boosting을 위한 **global\_ticks** 변수를 0으로 초기화한다.
  - Dictator를 **schedulerLock()** 을 호출한 process로 설정한다.
  - Dictator의 **consumed\_tq** 를 0으로 초기화한다.
  - Dictator의 **qlv** 를 -1로 설정한다. ( $L_{-1}$ 로 생각)

## 2-7. schedulerUnlock()

```
1 void schedulerUnlock(int password) {
2     struct proc* p = myproc();
3     acquire(&ptable.lock);
4     // NOT VALID condition
5     if (password != PASSWORD) {
6         cprintf("pid = %d, time quantum = %d, current queue level = %d\n",
7             p->pid, p->consumed_tq, p->qlv);
8         release(&ptable.lock);
9         exit();
10    }
11    // There is no dictator OR He is not dictator.
12    // VOID condition
13    else if (!is_dictated || p - ptable.proc != scheduler_dictator) {
14        // Do NOTHING.
15        release(&ptable.lock);
16    }
17    else {
18        // VALID condition
19        // posed to in front of L0.
20        p->priority = 3;
21        p->qlv = 0;
22        p->consumed_tq = 0;
23
24        int lv = 0;
25        int sz = mlfq[lv].size;
26        enQ(lv, p - ptable.proc);
27        for (int i = 0; i < sz; i++) {
28            enQ(lv, deQ(lv));
29        }
30
31        scheduler_dictator = -1; // variable to schedule one process .
32        is_dictated = 0; // variable to represent scheduler dictated.
33        release(&ptable.lock);
34    }
35 }
```

- MLFQ보다 우선하여 처리하는 process를 해제하기 위한 함수
- Line 5:10, NOT VALID한 호출에 대해서 pid, time quantum, 현재 Q level을 출력하고 process를 종료한다.  
NOT VALID한 경우는 인자로 전달되는 password가 틀린경우이다.
- Line 13:16, VOIDgks 한 호출이며 dictator가 없는 경우로 아무것도 하지 않는 호출이다.
- Line 11:28, VALID한 호출로 다음과 같은 작업을 진행한다.
  - Dictator의 priority, qlv, consumed\_tq를 초기화한다.
  - Dictator를  $L_0$ 의 맨 앞으로 옮긴다.
  - Dictator를 해제한다.

### Dictator가 없는 경우에 대한 호출

`schedulerUnlock()`은 dictator가 없을 때에도 호출이 가능하다.

그럼에도 이러한 호출에 대해서 NOT VALID로 간주하고 종료하지 않는 이유는 사용자의 입장에서 process가 얼마나 실행되어야 `global_ticks` 가 100이 되는지 알기 어렵기 때문이다.

각 instruction마다 실행되는 속도가 다르기 때문에 사용자가 `schedulerUnlock()` 을 명시적으로 호출했더라도, 그 전에 `global_ticks` 가 100이 넘어 Priority Boosting이 발생할 수 있다.

이런 경우 사용자가 의도하지 않았음에도 process가 종료되는 경우가 발생하기 때문에, 아무것도하지 않는 호출이 필요하다.

## 2-8. MLFQ

### 2-8-1. Structure

```
1 struct fq {
2     int proc_idx[NPROC]; // Save index of ptable.proc[]
3     int front;          // Front of Q
4     int back;           // Back of Q
5     int size;           // Size of Q
6 };
7 struct fq mlfq[3];
```

- MLFQ는 `fq`라는 자료구조의 배열 형태이다.
- `fq`는 내부에 `proc_idx[]`, `front`, `back`, `size`의 속성을 가진다.
  - `proc_idx[]` - `ptable.proc[]`에 저장되어 있는 index를 나타낸다.
  - `front` - Q의 맨 앞을 가리킨다.
  - `back` - Q의 맨 뒤를 가리킨다.
  - `size` - Q 전체의 크기를 나타낸다.

### 2-8-1. Functions

```
1 int enQ(int lv, int idx) {
2     mlfq[lv].proc_idx[mlfq[lv].back] = idx;
3     mlfq[lv].back++;
4     mlfq[lv].back %= NPROC;
5     mlfq[lv].size++;
6     return idx;
7 }
```

```
1 int deQ(int lv) {
2     if (isEmpty(lv))
3         return -1;
4     int ret = mlfq[lv].proc_idx[mlfq[lv].front];
5     mlfq[lv].front++;
6     mlfq[lv].front %= NPROC;
7     mlfq[lv].size--;
8     return ret;
9 }
```

- `enQ()`
  - Q에 원소를 enqueue하는 함수이다.
  - 인자로 `lv` 와 `idx`를 받는다.
  - `mlfq[lv]`의 `back`에 `idx`를 집어넣는다.
  - `back`과 `size`를 1 증가시킨다.
- `deQ()`
  - Q에 원소를 dequeue하는 함수이다.
  - 비어있으면 -1을 return 한다.
  - 인자로 `lv`를 받는다.
  - `mlfq[lv]`의 `front`에 저장된 값 `ret`을 꺼낸다.
  - `front`를 1 증가시킨다.

```

1 int isEmpty(int lv) {
2     return mlfq[lv].size == 0;
3 }

```

- `size` 를 1 감소시킨다.
- `ret` 을 return 한다.
- `isEmpty()`
  - Q에 원소가 존재하지 않는지 확인하는 함수이다.
  - Q가 비어있다면 1을, 그렇지 않다면 0을 return한다.
- `(isFull)`
  - 함수가 가득 찬지 확인하는 함수이다.
  - 일반적인 Q에서는 사용하지만, 여기에서는 사용하지 않는다.
- `allocproc()` 에서 `NPROC` 의 수만큼 process를 생성하고, 그 이상의 경우에 대해서는 자체적으로 막아주기 때문에, Q가 가득차더라도 추가로 process를 넣어야하는 경우가 발생하지 않는다.

## 2-9. Process structure

```

struct proc {

...
...
int qlv;
int priority;
int consumed_tq;
...
};

```

```

1 struct proc {
2     uint sz;
3     pde_t* pgdir;
4     char* kstack;
5     enum procstate state;
6     int pid;
7     struct proc* parent;
8     struct trapframe* tf;
9     struct context* context;
10    void* chan;
11    int killed;
12    struct file* ofile[NOFILE];
13    struct inode* cwd;
14    int priority;
15    int qlv;
16    uint consumed_tq;
17    char name[16];
18 }

```

- xv6에서의 process를 나타내는 proc 자료구조이다.
- 추가한 부분은 line14:16이다.
- `priority`
  - process의 우선순위를 나타낸다.
  - 초기값 3
- `qlv`
  - process가 속한 Q의 level을 나타낸다.
  - 초기값 0
- `consumed_tq`
  - process가 사용한 time quantum을 나타낸다.
  - 초기값 0

## 2-10. System calls

[`syscall.c`]

```

1 extern int sys_yield(void);
2 extern int sys_getLevel(void);
3 extern int sys_setPriority(void);
4 extern int sys_schedulerLock(void);
5 extern int sys_schedulerUnlock(void);

```

```

1 [SYS_yield] sys_yield,
2 [SYS_getLevel] sys_getLevel,
3 [SYS_setPriority] sys_setPriority,
4 [SYS_schedulerLock] sys_schedulerLock,
5 [SYS_schedulerUnlock] sys_schedulerUnlock,

```

[`syscall.h`]

```

1 #define SYS_yield      23
2 #define SYS_getLevel   24
3 #define SYS_setPriority 25
4 #define SYS_schedulerLock 26
5 #define SYS_schedulerUnlock 27

```

```

1 int sys_yield(void) {
2     yield();
3     return 1;
4 }
5 int sys_getLevel(void) {
6     return getLevel();
7 }
8 int sys_setPriority(void) {
9     int pid;
10    int priority;
11
12    if (argint(0, &pid) < 0 || argint(1, &priority) < 0)
13        return -1;
14    setPriority(pid, priority);
15    return 1;
16 }
17 int sys_schedulerLock(void) {
18     int password;
19
20     if (argint(0, &password) < 0)
21         return -1;
22     schedulerLock(password);
23     return 1;
24 }
25 int sys_schedulerUnlock(void) {
26     int password;
27
28     if (argint(0, &password) < 0)
29         return -1;
30     schedulerUnlock(password);
31     return 1;
32 }

```

```

1 void yield(void);
2 int getLevel(void);
3 void setPriority(int, int);
4 void schedulerLock(int);
5 void schedulerUnlock(int);

```

[`usys.S`]

```

1 SYSCALL(yield)
2 SYSCALL(getLevel)
3 SYSCALL(setPriority)
4 SYSCALL(schedulerLock)
5 SYSCALL(schedulerUnlock)

```

- `yield()`, `getLevel()`, `setPriority()`, `schedulerLock()`, `schedulerUnlock()` 을 system call로 제공하기 위해서 추가한 내용이다.
- `user.h`에 정의된 system call 함수가 호출되면, `usys.S`에서 해당 함수를 system call로 바꾸어 전달한다.
- 각각의 system call의 번호는 `syscall.h`에 정의되어 있고, 이에 따라서 `syscall.c`에 선언되어 있는 wrapper function을 실행하게 된다.
- `sysproc.c`에 세부적인 wrapper function의 정의가 되어 있다.
- 다른 system call과 같이 인자를 받아서 실제 함수를 호출하는 형식으로 되어 있다.
- 특이사항으로 일반적인 system call 함수는 실패의 의미로 -1을 return 하는데, `sys_getLevel()`의 경우 dictator의 `qlv` 가 -1이므로 주의해야 한다.

## 2-11. Interrupts

[traps.h]

```
1 #define T_LOCK 129
2 #define T_UNLOCK 130
```

[trap.c]

```
1 SETGATE(idt[T_LOCK], 1, SEG_KCODE << 3, vectors[T_LOCK], DPL_USER); // int 129 can be called from user level
2 SETGATE(idt[T_UNLOCK], 1, SEG_KCODE << 3, vectors[T_UNLOCK], DPL_USER); // int 130 can be called from user level
```

[trap.c]

```
1 uint global_ticks;
```

```
1 case T_LOCK:
2     schedulerLock(PASSWORD);
3     break;
4 case T_UNLOCK:
5     schedulerUnlock(PASSWORD);
6     break;
```

```
1 ticks++;
2 global_ticks++;
```

- `traps.h`에 `schedulerLock()` 과 `schedulerUnlock()` interrupt를 받기 위한 Interrupt Number가 정의되어 있다.
- `trap.c`에 `ticks` 를 본딴, `global_ticks` 가 선언되어 있다.
- `trap.c`에 `schedulerLock()`, `schedulerUnlock()` 를 user가 호출할 수 있는 interrupt로 바꾸었다.
- `trap.c`에 `global_ticks` 는 `ticks` 와 같이 증가하도록 만들어 두었다.
- `trap.c`에 interrupt가 발생했을 때, `schedulerLock()` 과 `schedulerUnlock()` 을 실행하는 부분을 추가했다.

## 2-12. Definitions

[defs.h]

```
1 // newly defined variables or functions
2 #define GLOBAL_TICKS_LIMIT 100
3 #define NUMQ 3
4 #define PASSWORD 2018008240
```

[defs.h]

```
1 /*****Added properties in proc.c*****
2 int getLevel(void);
3 void setPriority(int pid, int priority);
4 void schedulerLock(int password);
5 void schedulerUnlock(int password);
6 void priorityBoost(void);
7 int enQ(int lv, int idx);
8 int deQ(int lv);
9 int isEmpty(int lv);
10 int pickProc(int lv);
11 extern int scheduler_locked;
12 extern uint global_ticks;
13 #define USELESS 200
```

- `proc.c` 등에 사용하는 변수들을 선언했다.

- `proc.c` 등에 새로 정의된 함수들을 선언했다.

# 3. Result

## 3-1. Test 1 - Default

### [Test 1의 소스 코드]

```
1 printf(1, "[Test 1] default\n");
2 pid = fork_children();
3
4 if (pid != parent)
5 {
6     for (i = 0; i < NUM_LOOP; i++)
7     {
8         int x = getLevel();
9         if (x < 0 || x > 2)
10        {
11            printf(1, "Wrong level: %d\n", x);
12            exit();
13        }
14        count[x]++;
15    }
16    printf(1, "Process %d\n", pid);
17    for (i = 0; i < MAX_LEVEL; i++)
18        printf(1, "Process %d , L%d: %d\n", pid, i, count[i]);
19 }
20 exit_children();
21 printf(1, "[Test 1] finished\n");
```

### [Test 1의 실행 결과]

```
$ mlfq_test_fixed 1
MLFQ test start
[Test 1] default
Process 4
Process 4 , L0: 8907
Process 4 , L1: 10086
Process 4 , L2: 81007
Process 6
Process 6 , L0: 13361
Process 6 , L1: 18081
Process 6 , L2: 68558
Process 7
Process 7 , L0: 13339
Process 7 , L1: 20718
Process 7 , L2: 65943
Process 5
Process 5 , L0: 13478
Process 5 , L1: 19833
Process 5 , L2: 66689
[Test 1] finished
done
$ |
```

### [분석]

- 가장 기본적인 process의 scheduling의 결과이다.
- 부모 process가 자식 process 4 ~ 7을 생성하고, 자식 process 들은 scheduling될 때마다 자신의 level을 확인해서 `count[]` 를 1씩 증가시킨다.
- 모든 process가  $L_0 < L_1 < L_2$ 의 빈도로 scheduling되는 것은 MLFQ의 policy에 잘 맞는다.
- process의 번호가 낮을수록  $L_2$ 의 비율이 높아지는 경향이 있는데, 이는 위에서 설명했듯이,  $L_2$ 에 먼저 도달하여,  $L_2$ 에서 가장 많이 scheduling되기 때문이다.
- process 종료순서가 생성순서와 다른 것은 Priority Boosting의 영향으로 보인다.

## 3-2. Test 2 - Priority

### [Test 2의 소스 코드]

```
1 printf(1, "[Test 2] priorities\n");
2 pid = fork_children2();
3
4 if (pid != parent)
5 {
6     for (i = 0; i < NUM_LOOP; i++)
7     {
8         int x = getLevel();
9         if (x < 0 || x > 2)
10        {
11            printf(1, "Wrong level: %d\n", x);
12            exit();
13        }
14        count[x]++;
15    }
16    printf(1, "Process %d\n", pid);
17    for (i = 0; i < MAX_LEVEL; i++)
18        printf(1, "Process %d , L%d: %d\n", pid, i, count[i]);
19 }
20 exit_children();
21 printf(1, "[Test 2] finished\n");
```

### [Test 2의 실행 결과]

```
$ mlfq_test_fixed 2
MLFQ test start
[Test 2] priorities
before sleep pid: 4, lv: 0
before sleep pid: 5, lv: 0
before sleep pid: 7, lv: 0
before sleep pid: 6, lv: 0
after sleep pid: 4, lv: 1
after sleep pid: 5, lv: 1
after sleep pid: 7, lv: 1
after sleep pid: 6, lv: 1
Process 4
Process 4 , L0: 5812
Process 4 , L1: 14566
Process 4 , L2: 79622
Process 7
Process 7 , L0: 9729
Process 7 , L1: 22247
Process 7 , L2: 68024
Process 6
Process 6 , L0: 10407
Process 6 , L1: 22235
Process 6 , L2: 67358
Process 5
Process 5 , L0: 10612
Process 5 , L1: 21442
Process 5 , L2: 67946
[Test 2] finished
done
$ |
```

### [분석]

- `fork_children2()` 는 자식 process를 생성하고 생성한 순서대로  $[0, 3] \in \mathbb{N}$ 의 priority를 부여한다. 즉, MLFQ가 우선순위에 대한 scheduling을 잘 하는지 확인하는 테스트이다.
- `fork_children2()` 안에 `sleep()` 함수가 있기 때문에 자식 process는 `fork_children2()` 가 끝난 시점에서  $L_1$ 에 있다.
- Process 4를 제외하고는 위의 Test 1과 결과가 비슷한데, 이는 Process 4가  $L_2$ 에서 우선적으로 scheduling이 되다가 Priority Boosting이 발생하면, 모든 process의 priority가 3으로 초기화 되기 때문이다.
- process 종료순서가 생성순서와 다른 것은 Priority Boosting의 영향으로 보인다.

### 3-3. Test 3 - yield

#### [Test 3의 소스 코드]

```

1 printf(1, "[Test 3] yield\n");
2 pid = fork_children2();
3
4 if (pid != parent)
5 {
6     for (i = 0; i < NUM_YIELD; i++)
7     {
8         int x = getLevel();
9         if (x < 0 || x > 2)
10        {
11            printf(1, "Wrong level: %d\n", x);
12            exit();
13        }
14        count[x]++;
15        yield();
16    }
17    printf(1, "Process %d\n", pid);
18    for (i = 0; i < MAX_LEVEL; i++)
19    {
20        printf(1, "Process %d , L%d: %d\n", pid, i, count[i]);
21    }
22    exit_children();
23    printf(1, "[Test 3] finished\n");

```

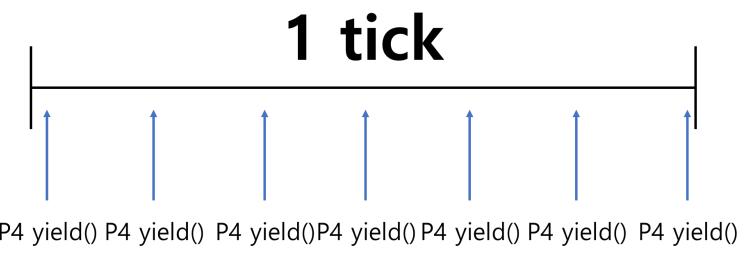
#### [Test 3의 실행 결과]

```

mlfq_test_fixed 3
MLFQ test start
[Test 3] yield
before sleep pid: 4, lv: 0
before sleep pid: 5, lv: 0
before sleep pid: 6, lv: 0
before sleep pid: 7, lv: 0
after sleep pid: 4, lv: 1
after sleep pid: 5, lv: 1
after sleep pid: 6, lv: 1
after sleep pid: 7, lv: 1
Process 4
Process 4 , L0: 15
Process 4 , L1: 28
Process 4 , L2: 19957
Process 7
Process 7 , L0: 15
Process 7 , L1: 28
Process 7 , L2: 19957
Process 5
Process 5 , L0: 20
Process 5 , L1: 35
Process 5 , L2: 19945
Process 6
Process 6 , L0: 20
Process 6 , L1: 35
Process 6 , L2: 19945
[Test 3] finished
done
$
```

#### [분석]

- Process가 `yield()` 를 호출 했을 때, time quantum을 어떻게 처리하는지를 확인하기 위한 테스트이다.
- 나의 구현에서는 `yield()` 와는 상관없이 process가 scheduling의 대상이 되면 time quantum을 증가시키기 때문에  $L_2$ 로 빠르게 내려가게 되고, 그에 따라서  $L_2$ 에서 scheduling되는 횟수가 압도적으로 많게 된다.
- 이 때에, Priority Boosting이 발생하는 `global_ticks` 와 process가 사용한 time quantum이 다르기 때문에, 1 tick에 대해서 여러 번 프로세스가 실행될 수 있기 때문에 이러한 결과가 발생한다.
  - 아래의 그림과 같은 상황이 발생한다.



### 3-4. Test 4 - sleep

#### [Test 4의 소스 코드]

```

1 printf(1, "[Test 4] sleep\n");
2 pid = fork_children2();
3 if (pid != parent)
4 {
5     int my_pid = getpid();
6     printf(1, "pid: %d fork: %d\n", my_pid, getpid());
7
8     for (i = 0; i < NUM_SLEEP; i++)
9     {
10        int x = getLevel();
11        if (x < 0 || x > 2)
12        {
13            printf(1, "Wrong level: %d\n", x);
14            exit();
15        }
16        count[x]++;
17        sleep(1);
18    }
19    sleep(my_pid * 3);
20    printf(1, "Process %d\n", pid);
21    for (i = 0; i < MAX_LEVEL; i++)
22    {
23        printf(1, "Process %d , L%d: %d\n", pid, i, count[i]);
24    }
25    exit_children();
26    printf(1, "[Test 4] finished\n");

```

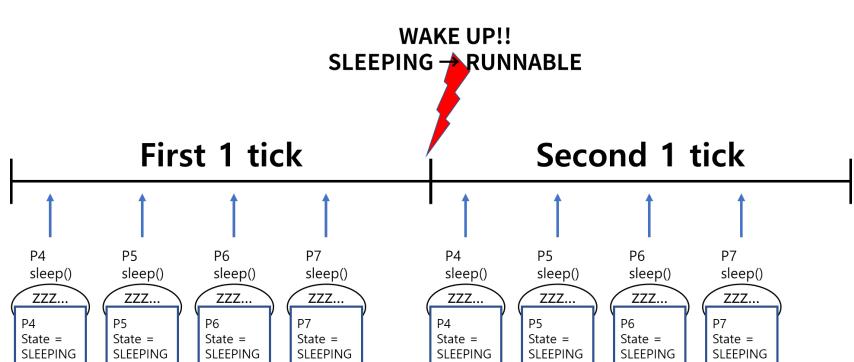
#### [Test 4의 실행 결과]

```

$ mlfq_test_fixed 4
MLFQ test start
[Test 4] sleep
before sleep pid: 4, lv: 0
before sleep pid: 5, lv: 0
before sleep pid: 6, lv: 0
before sleep pid: 7, lv: 0
after sleep pid: 4, lv: 1
pid: 4 fork: 1
after sleep pid: 5, lv: 1
pid: 5 fork: 1
after sleep pid: 6, lv: 1
pid: 6 fork: 1
after sleep pid: 7, lv: 1
pid: 7 fork: 1
Process 4
Process 4 , L0: 20
Process 4 , L1: 35
Process 4 , L2: 445
Process 5
Process 5 , L0: 20
Process 5 , L1: 34
Process 5 , L2: 446
Process 6
Process 6 , L0: 20
Process 6 , L1: 34
Process 6 , L2: 446
Process 7
Process 7 , L0: 20
Process 7 , L1: 34
Process 7 , L2: 446
[Test 4] finished
done
$
```

#### [분석]

- Process가 `sleep()` 를 호출 했을 때, time quantum을 어떻게 처리하는지를 확인하기 위한 테스트이다.
- 나의 구현에서는 `sleep()` 과는 상관없이 process가 scheduling의 대상이 되면 time quantum을 증가시키기 때문에  $L_2$ 로 빠르게 내려가게 되고, 그에 따라서  $L_2$ 에서 scheduling되는 횟수가 압도적으로 많게 된다.
- Process 사이에서 level에 머무는 정도가 거의 비슷한 이유는 `yield()` 와는 다르게, SLEEPING인 process는 scheduling의 대상이 아니기 때문에,  $L_2$ 에 먼저 내려와서 priority가 더 높아진 process라도 scheduling의 대상이 되지 않고, 다른 process가 선택되기 때문이다.
  - 아래의 그림과 같은 상황이 발생한다.



### 3-5. Test 5 - Max level

[Test 5의 소스 코드]

```

1 printf(1, "[Test 5] max_level\n");
2 pid = fork();
3 printf(1, "Process %d's max level is %d\n", getpid(), max_level);
4 if (pid != parent) // child
5 {
6     for (i = 0; i < NUM_LOOP; i++)
7     {
8         int x = getlevel();
9         if (x < 0 || x > 2)
10        {
11            printf(1, "Wrong level: %d\n", x);
12            exit();
13        }
14        count[x]++;
15        if (x > max_level) {
16            if (x == 3) {
17                // printf("%d: Process %d's max level is %d and Q level is %d\n", getpid(), max_level, x);
18                yield();
19            }
20        }
21    }
22    printf(1, "Process %d\n", pid);
23    for (i = 0; i < MAX_LEVEL; i++)
24    printf(1, "Process %d , %d\n", pid, i, count[i]);
25
26 }
27 exit_children();
28 printf(1, "[Test 5] finished\n");

```

[Test 5의 실행 결과]

```

$ mlfq_test_fixed 5
MLFQ test start
[Test 5] max level
Process 3's max level is 0
Process 4's max level is 0
Process 7's max level is 3
Process 5's max level is 1
Process 6's max level is 2
Process 7
Process 7 , L0: 10349
Process 7 , L1: 16522
Process 7 , L2: 73129
Process 6
Process 6 , L0: 10109
Process 6 , L1: 17151
Process 6 , L2: 72740
Process 5
Process 5 , L0: 23952
Process 5 , L1: 33211
Process 5 , L2: 42837
Process 4
Process 4 , L0: 30754
Process 4 , L1: 48
Process 4 , L2: 69198
[Test 5] finished
done
$ |

```

[분석]

- Process가 자신에게 설정된 `max_level` 변수보다 높은 Q에 있으 면 `yield()`를 호출하여, process의 scheduling을 종료하는 함 수이다.
- **Process 4**
  - `max_level` : 0
  - $L_1, L_2$ 에서 매번 `yield()`를 호출하기 때문에  $L_1$ 이 극단적 으로 낮다. 다만  $L_2$ 는 다른 프로세스와 비슷한 수준이다.  
⇒ 이는 마지막에 종료되기 때문에,  $L_2$ 에 process 4만 남 아서 `yield()`를 해도 계속 scheduling되기 때문에  $L_2$ 가 70%의 비율이 나온 것으로 보인다.
- **Process 5**
  - `max_level` : 1
  - $L_2$ 에서 매번 `yield()`를 호출하기 때문에  $L_2$ 가 다른 프로 세스에 비해서 매우 낮다.

### 3-6. Test 6 - setPriority Exception

[Test 6의 소스 코드]

[Test 6의 실행 결과]

```

1 printf(1, "[Test 6] setPriority return value\n");
2 child = fork();
3
4 if (child == 0)
5 {
6     // int r;
7     int grandson;
8     sleep(10);
9     grandson = fork();
10    if (grandson == 0)
11    {
12        int my_pid = getpid();
13        printf(1, "Process %d set process %d's priority to %d\n",
14               my_pid, my_pid - 2, 0);
15        setPriority(my_pid - 2, 0);
16
17        printf(1, "Process %d set process %d's priority to %d\n",
18               my_pid, my_pid - 3, 0);
19        setPriority(my_pid - 3, 0);
20    }
21    else
22    {
23        int my_pid = getpid();
24        printf(1, "Process %d set process %d's priority to %d\n",
25               my_pid, grandson, 0);
26        setPriority(grandson, 0);
27
28        printf(1, "Process %d set process %d's priority to %d\n",
29               my_pid, grandson, 0);
30        setPriority(my_pid + 1, 0);
31    }
32    sleep(20);
33    wait();
34 }
35 else
36 {
37     int child2 = fork();
38     sleep(20);
39     if (child2 == 0)
40         sleep(10);
41     else
42     {
43         int my_pid = getpid();
44         printf(1, "Process %d set process %d's priority to %d\n",
45               my_pid, child, -1);
46         setPriority(child, -1);
47
48         printf(1, "Process %d set process %d's priority to %d\n",
49               my_pid, child, 11);
50         setPriority(child, 11);
51
52         printf(1, "Process %d set process %d's priority to %d\n",
53               my_pid, child, 10);
54         setPriority(child, 10);
55
56         printf(1, "Process %d set process %d's priority to %d\n",
57               my_pid, child + 1, 10);
58         setPriority(child + 1, 10);
59
60         printf(1, "Process %d set process %d's priority to %d\n",
61               my_pid, child + 2, 10);
62         setPriority(child + 2, 10);
63
64         printf(1, "Process %d set process %d's priority to %d\n",
65               my_pid, -1, 2);
66         setPriority(-1, 2);
67
68         printf(1, "Process %d set process %d's priority to %d\n",
69               my_pid, child2 + 10, 2);
70         setPriority(child2 + 10, 2);
71    }
72 }
73 exit_children();
74 printf(1, "done\n");
75 printf(1, "[Test 6] finished\n");

```

```

$ mlfq_test_fixed 6
MLFQ test start
[Test 6] setPriority return value
Process 4 set prProcess 6 set process 4's priority to 0
Process 6 set process 3's priority to 0
ocess 6's priority to 0
Process 4 set process 6's priority to 0
Process 3 set process 4's priority to -1
Priority should be in range from 0 to 3
Process 3 set process 4's priority to 11
Priority should be in range from 0 to 3
Process 3 set process 4's priority to 10
Priority should be in range from 0 to 3
Process 3 set process 5's priority to 10
Priority should be in range from 0 to 3
Process 3 set process 6's priority to 10
Priority should be in range from 0 to 3
Process 3 set process -1's priority to 2
Priority should be in range from 0 to 3
Process 3 set process 15's priority to 2
Priority should be in range from 0 to 3
done
[Test 6] finished
done
$ |

```

[분석]

- `setPriority()` 함수의 인자에 따라서 어떻게 처리하는지를 확인하기 위한 테스트이다.
- 나의 구현은 `setPriority()`의 인자인 `pid`나 `priority`가 잘못 들어오게 되면, 오류문구를 출력하고 함수를 종료한다.
- 그렇지 않은 경우, `pid`를 갖는 process의 의 `priority`를 주어진 인자로 바꾼다.
- Line 10:31, `setPriority()`에 VALID한 인자가 전달되어 잘 실행되었다.

- Line 44:62, `setPriority()` 에 인자로 전달된 `priority` 가  $[0, 3] \in \mathbb{N}$ 을 만족하지 않아 오류문구가 출력되었다.
  - Line 64:70, `setPriority()` 에 인자로 전달된 `pid` 가 존재하지 않는 process이므로 오류문구가 출력되었다.

## 3-7. Test 7 - schedulerLock / Unlock Normal Case

## [Test 7의 소스 코드]

```

1 printf(1, "[Test 7] schedulerLock / Unlock Normal Case\n");
2 child = fork();
3 if (child == 0) { //child
4     int my_pid = getpid();
5     printf(1, "Process %d scheduled Lock\n", my_pid);
6     schedulerLock(PASSWORD);
7     printf(1, "Process %d in Q level %d\n", my_pid, getLevel());
8     for (int i = 0; i < NUM_LOOP;i++) {
9         if (i % (NUM_LOOP / 10) == 0)
10             printf(1, "Process %d in Q level %d\n", my_pid, getLevel());
11     }
12     schedulerUnlock(PASSWORD);
13     printf(1, "Process %d scheduled Lock\n", my_pid);
14 }
15 else { // parent
16     int my_pid = getpid();
17     printf(1, "Process %d scheduled Lock\n", my_pid);
18     schedulerLock(PASSWORD);
19     printf(1, "Process %d in Q level %d\n", my_pid, getLevel());
20     for (int i = 0; i < NUM_LOOP;i++) {
21         if (i % (NUM_LOOP / 10) == 0)
22             printf(1, "Process %d in Q level %d\n", my_pid, getLevel());
23     }
24     schedulerUnlock(PASSWORD);
25     printf(1, "Process %d scheduled Lock\n", my_pid);
26 }
27 exit_children();
28 printf(1, "[Test 7] finished\n");

```

## [Test 7의 실행 결과]

```
$ mlfq_test_fixed 7
MLFQ test start
[Test 7] schedulerLock / Unlock Normal Case
Process 3 scheduler Lock
Process 3 in Q level -1
Process 3 scheduler Unlock
Process 4 scheduler Lock
Process 4 in Q level -1
Process 4 scheduler Unlock
[Test 7] finished
done
$
```

## [분석]

- `schedulerLock()`, `schedulerUnlock()` 이 잘 작동하지는 지에 대한 테스트이다.
  - `NUM_LOOP` 는 100,000으로 정의되어 있다.
  - 만약 process 3,4가 MLFQ로 scheduling이 되었다면  $L_0, L_1$  에서는 RR의 scheduling 방식을 따르기 때문에, 서로 번갈아가면서 for-loop 안의 `printf` 구문이 출력되었을 것이다.
  - 그러나 `schedulerLock()` 이 실행되어 process 3이 먼저 실행되고 `schedulerUnlock()` 을 호출한 뒤, process 4가 `schedulerLock()` 과 `schedulerUnlock()` 을 호출하였다.
  - `schedulerLock()` 을 호출 한 process인 Dictator의 Q level은 -1로 정의했기 때문에 정상적인 출력이다.

## 3-8. Test 8 - schedulerLock / Unlock PASSWORD ERROR

## [Test 8의 소스 코드]

```
1 printf(1, "[Test 8] schedulerLock / Unlock Wrong Case1 : PASSWORD);
2 child = fork();
3 if (child == 0) { //child
4     int my_pid = getpid();
5     printf(1, "Process %d schedluer Lock\n", my_pid);
6     printf(1, "This procedure intends to scheduler Lock Error for
7
8     schedulerLock(PASSWORD + 1);
9     // It will not be executed under below.
10    printf(1, "FROM THIS EXECUTION WILL NOT BE EXECUTED!!\n");
11    printf(1, "Process %d in Q level %d\n", my_pid, getLevel());
12    schedulerUnlock(PASSWORD + 1);
13    printf(1, "Process %d schedluer Unlock\n", my_pid);
14 }
15 else { // parent
16     int child2 = fork();
17     if (child2 == 0) {
18         int my_pid = getpid();
19         printf(1, "Process %d schedluer Lock\n", my_pid);
20         schedulerLock(PASSWORD);
21         printf(1, "Process %d in Q level %d\n", my_pid, getLevel());
22         printf(1, "This procedure intends to scheduler Unlock Error
23         schedulerUnlock(PASSWORD + 1);
24         // It will not be executed under below.
25         printf(1, "FROM THIS EXECUTION WILL NOT BE EXECUTED!!\n");
26         printf(1, "Process %d schedluer Unlock\n", my_pid);
27     }
28 }
29 exit_children();
30 printf(1, "[Test 8] finished\n");
```

## [분석]

- `schedulerLock()`, `schedulerUnlock()`에 잘못된 인자가 들어왔을 때 제대로 처리하는지 확인하기 위한 테스트이다.
  - 두 함수 모두 NOT VALID 호출에 대해서 process의 pid, time quantum, Q level을 출력하고 process를 종료한다.
  - Line 8, `schedulerLock()` 호출 시에 인자를 잘못 넘겨주었을 때, 아래의 `printf`가 실행되지 않으므로 process가 종료되었음을 알 수 있다.
  - Line 23, `schedulerUnlock()` 호출 시에 인자를 잘못 넘겨주었을 때, 아래의 `printf`가 실행되지 않으므로 process가 종료되었음을 알 수 있다.

## [Test 8의 실행 결과]

```
$ mlfq_test_fixed 8
MLFQ test start
[Test 8] schedulerLock / Unlock Wrong Case1 : PASSWORD ERROR
Process 4 schedluer Lock
This procedure intends to scheduler Lock Error for PASSWORD
pid = 4, time quantum = 1, current queue level = 0
Process 5 schedluer Lock
Process 5 in Q level -1
This procedure intends to scheduler Unlock Error for PASSWORD
pid = 5, time quantum = 2, current queue level = -1
[Test 8] finished
done
$ |
```

### 3-9. Test 9 - schedulerLock / Unlock duplication ERROR

#### [Test 9의 소스 코드]

```
1 printf(1, "[Test 9] schedulerLock / Unlock Wrong Case2 : duplication ERROR\n");
2 child = fork();
3 if (child == 0) { //child
4     int my_pid = getpid();
5     printf(1, "Process %d schedluer Lock\n", my_pid);
6
7     schedulerLock(PASSWORD);
8     printf(1, "Process %d in Q level %d\n", my_pid, getLevel());
9     printf(1, "This procedure intends to scheduler Lock Error for duplication\n");
10    schedulerLock(PASSWORD);
11
12    // It will not be executed under below.
13    printf(1, "FROM THIS EXECUTION WILL NOT BE EXECUTED!!\n");
14    printf(1, "Process %d schedluer Unlock\n", my_pid);
15 }
16 else { // parent
17     int child2 = fork();
18     if (child2 == 0) {
19         int my_pid = getpid();
20         printf(1, "Process %d schedluer Lock\n", my_pid);
21         schedulerLock(PASSWORD);
22         printf(1, "Process %d in Q level %d\n", my_pid, getLevel());
23         printf(1, "This procedure intends to scheduler Unlock Error for duplication\n");
24         schedulerUnlock(PASSWORD);
25         printf(1, "Process %d schedluer Unlock\n", my_pid);
26         schedulerUnlock(PASSWORD);
27
28        // It will be executed under below.
29        printf(1, "FROM THIS EXECUTION SHOULD BE EXECUTED!!\n");
30        printf(1, "Process %d schedluer Unlock Again\n", my_pid);
31
32    }
33    exit_children();
34    printf(1, "[Test 9] finished\n");
35 }
```

#### [Test 9의 실행 결과]

```
$ mlfq_test_fixed 9
MLFQ test start
[Test 9] schedulerLock / Unlock Wrong Case2 : duplication ERROR
Process 5 schedluer Lock
Process 5 in Q level -1
This procedure intends to scheduler Unlock Error for duplication
Process 5 schedluer Unlock
FROM THIS EXECUTION SHOULD BE EXECUTED!!
Process 5 schedluer Unlock Again
4 schedluer Lock
Process 4 in Q level -1
This procedure intends to scheduler Lock Error for duplication
pid = 4, time quantum = 2, current queue level = -1
[Test 9] finished
done
$
```

### 3-10. Test 10 - schedulerLock / Unlock Interrupt Call

#### [Test 10의 소스 코드]

```
1 int my_pid = getpid();
2 printf(1, "[Test 10] schedulerLock / Unlock Interrupt Call Test\n");
3 printf(1, "Process %d schedluer Lock\n", my_pid);
4 __asm__("int $129");
5 printf(1, "Process %d in Q level %d\n", my_pid, getLevel());
6
7 printf(1, "Process %d schedluer Unlock\n", my_pid);
8 __asm__("int $130");
9 printf(1, "Process %d in Q level %d\n", my_pid, getLevel());
10 printf(1, "[Test 10] finished\n");
```

#### [Test 10의 실행 결과]

```
$ mlfq_test_fixed A
MLFQ test start
[Test 10] schedulerLock / Unlock Interrupt Call Test
Process 3 schedluer Lock
Process 3 in Q level -1
Process 3 schedluer Unlock
Process 3 in Q level 0
[Test 10] finished
done
$ |
```

#### [분석]

- `schedulerLock()`, `schedulerUnlock()` 이 중복되어 호출되는 경우에 대한 테스트이다.
  - `schedulerLock()` 이 중복되어 호출되는 경우 Dictator가 다시 `schedulerLock()` 을 호출하는 경우를 말한다. Process가 악의적으로 CPU를 점유하려는 것으로 간주해 process의 pid, time quantum, Q level을 출력하고 process를 종료한다.
  - `schedulerUnlock()` 이 중복되어 호출되는 경우 Dictator가 존재하지 않을 때, `schedulerUnlock()` 을 호출하는 경우를 말한다. 잘못된 호출이지만, 사용자가 의도하지 않은 결과가 나올 수 있기 때문에 종료하지 않는다.
- Line 3:15, `schedulerLock()` 을 중복해서 호출하는 경우이다.
  - Line 10에서 `schedulerLock()` 을 중복해서 호출하기 때문에 line 13이후로 실행되지 않는다.
- Line 16:33, `schedulerUnlock()` 을 중복해서 호출하는 경우이다.
  - Line 26에서 `schedulerUnlock()` 을 중복해서 호출하지만, 아무것도 하지 않기 때문에 line 29이후가 실행된다.

## 4. Trouble shooting

### 4-1. Design for Scheduler

Scheduler의 Design을 어떻게 할 것인가는 이 과제의 대주제이자, 세부적인 구현을 하기 위해서는 필수적으로 고민해야 하는 것이었다.

두 가지의 Design을 놓고 고민을 했는데, 하나는 MLFQ를 Q로 구현하는 방법이고, 다른하나는 특별한 자료구조를 만들지 않고, for문을 이용해서 논리적으로 MLFQ처럼 보이는 방법이었다.

#### 4-1-1. MLFQ with for-loop

##### • 장점

- 다른 자료구조를 만들지 않아도 된다.
- `scheduler()` 함수가 단순해진다.

##### • 단점

- 코드를 함수화하기 어려워서 명확하게 보이지 않는다.
- 논리적으로 MLFQ를 보이는 것이 어렵다.
- Process의 생애를 관리하기 어렵다.

#### 4-1-2. MLFQ with Q

##### • 장점

- 코드를 함수화하기 쉬워서 전체적인 구조를 파악하기 쉽다.
- Q에 대한 연산을 이용해서 process를 관리하기 쉽다.

##### • 단점

- 다른 자료구조를 만들어야 한다.
- `scheduler()` 함수가 복잡해진다.

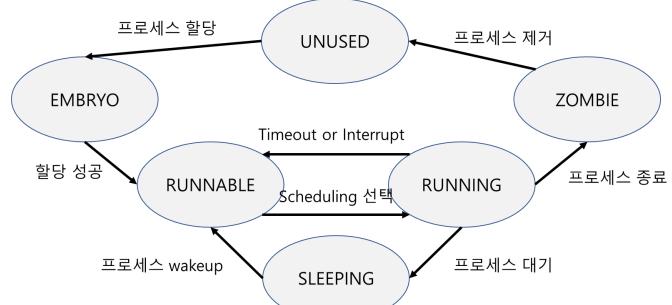
### 4-1-3. 결론

Q를 직접 구현하여 MLFQ scheduling을 하는 것을 구현했다.

처음에는 for-loop을 이용하는 방법으로 진행했었지만 process를 골라서 scheduling하는 방법을 구현하기 어려웠고, Priority Boosting을 할 때도 순서관계를 보존하면서 옮기는 것이 어려웠다. 그래서 Q를 직접 만드는 방법으로 진행했다.

Q를 직접 만들기만 하면, 코드가 전체적으로 많아질 뿐이고 구조화가 잘 되기 때문에 코딩하기에는 더 쉬웠다.

## 4-2. Process in Q



MLFQ에 어떤 상태의 process를 넣어서 유지할 것인지는 굉장히 중요한 문제이다.

왜냐하면, process의 상태는 다양하고, 수시로 바뀌기 때문에 잘못된 process가 Q에 있게되면 Q의 성능 저하뿐만 아니라, 의도하지 않는 문제가 발생해서, OS 자체에 문제가 생길 수 있기 때문이다.

우선 UNUSED는 process가 아닌 상태이므로 MLFQ에 넣을 대상이 아니다.

다음으로 ZOMBIE는 종료된 process이므로 MLFQ에 넣을 대상이 아니다.

EMBRYO는 수업시간에 scheduling 대상이 되기 전의 대기 상태라고 하였기 때문에, MLFQ에 넣을 대상이 아니다.

RUNNABLE은 scheduling의 대상이므로 반드시 MLFQ에 넣어야 한다.

### 문제가 되는건은 SLEEPING과 RUNNING이다.

SLEEPING은 scheduling의 대상이 아니지만, `sleep()`이 종료되면 RUNNABLE의 상태로 바뀐다.

RUNNING은 RUNNABLE, SLEEPING, ZOMBIE의 3가지 상태로 바뀔 수 있는 가능성을 가지고 있다.

만약 SLEEPING을 MLFQ에 넣을 대상에 빼게 되면, SLEEPING → RUNNABLE로 바뀌는 모든 경우에 대해서 process를 MLFQ에 enqueue해주어야 한다. 대신에, MLFQ에는 오직 RUNNABLE한 process들만 있기 때문에, scheduling에 소모되는 overhead는 매우 감소한다.

반면에, SLEEPING을 MLFQ에 넣을 대상으로 생각하면, 구현이 매우 간단해 진다. 그 이유는 SLEEPING 상태는 모두 `sleep()`에서만 이루어지는데, `sleep()`에서는 `sched()`을 호출해서 다시 `scheduler()`로 돌아오기 때문이다. 즉, SLEEPING을 찾아다니면서 Q에 넣을 필요없이 `scheduler()`에서 process의 상태를 확인해서 ZOMBIE가 아닌 경우에만 Q에 넣어주면 된다. 이러한 구현에서는 SLEEPING인 process가 많으면 많아질수록 overhead가 커지지만, xv6에서 허용하는 정도의 크기라면 감수할 만한 정도라고 생각했다.

결론적으로 나의 MLFQ에는 RUNNABLE과 SLEEPING의 상태를 갖는 process만 들어있다.

## 4-3. Ticks

Ticks를 어떻게 사용할 것인가, 1 tick을 어떤 개념으로 생각할 것인가는 굉장히 조심스러운 문제였다. 왜냐하면 Priority Boosting과 time quantum 때문이다.

### 4-3-1. Global ticks

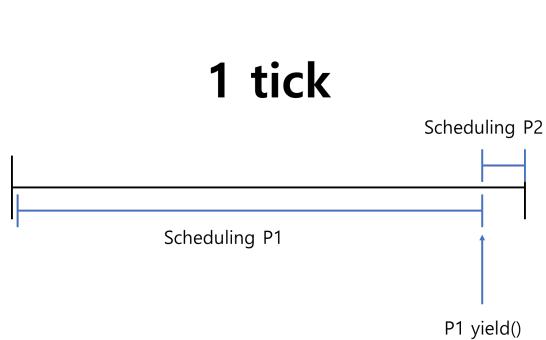
Global ticks는 Priority Boosting과 관련된 문제이기 때문에 중요하다. 또한 Global ticks는 `sys_uptime()`을 통해서 호출될 수도 있기 때문에 조심해야 한다.

그래서 실제로 xv6 내에서 global ticks를 나타내는 변수인 `ticks`와 같이 증가하는 `global_ticks` 변수를 선언하여, 이 변수를 통해서 Priority Boosting을 실행한 뒤에, `global_ticks`를 0으로 초기화하는 방식으로 진행했다.

### 4-3-2. Ticks vs Time quantum

Time quantum은 process가 CPU를 사용한 시간을 나타내고, Ticks는 매 Timer interrupt 사이의 간격이다.

처음에는 이 두 개의 차이를 잘 이해하지 못해서, Ticks와 Time quantum이 동일하게 올라가도록 구현했다. 그러나 이 방법은 다음과 같은 경우에 이상하게 작동한다.



왼쪽의 그림을 보면 전체 1 tick 중에서 대부분의 경우  $P_1$ 이 사용하고 마지막에 `yield()`를 호출했다.

그리고 남은 시간에 대해서  $P_2$ 가 scheduling되다가 Timer Interrupt가 발생했다.

그러면 실제로 CPU를 많이 사용한 것은  $P_1$ 임에도 불구하고  $P_2$ 의 time quantum이 올라가는 일이 발생한다.

이런 상황을 방지하기 위해서는 두 가지의 구현이 가능하다.

1. `yield()`를 호출할 때마다 time quantum을 증가시킨다.
2. scheduling이 될 때마다 time quantum을 증가시킨다.

위의 두 가지를 비교하고 왜 2번의 구현을 선택했는지에 대해서 설명한다.

- `yield()`를 호출할 때마다 time quantum을 증가시킨다.
- scheduling이 될 때마다 time quantum을 증가시킨다.

#### ◦ 장점

- `yield()`를 호출하는 함수의 time quantum만을 증가시키기 때문에 Ticks의 의미와 Time quantum의 의미가 잘 맞는다.
- 악의적으로 Timer Interrupt가 발생하기 전에 `yield()`를 호출하는 process를  $L_0$ 에서  $L_2$ 로 내릴 수 있다.
- `sleep()`을 호출하는 process는 time quantum이 올라가지 않기 때문에 실질적인 의미의 SLEEPING 상태를 구현할 수 있다.

#### ◦ 단점

- `sleep()`을 호출하는 process는 time quantum이 올라가지 않기 때문에 계속해서 같은 Q level에 존재한다.  
⇒ `sleep()`의 구조상 매 tick마다 process가 RUNNABLE로 바뀌기 때문에 Q에서 scheduling의 무의미한 시간을 보내게 된다.

#### ◦ 종합

- Ticks와 Time quantum, SLEEPING의 의미가 잘 맞는다.
- 악의적인 `yield()`를 제외한다면, Time quantum은 process가 소모한 시간을 나타낸다.
- SLEEPING process에 대한 overhead가 커질 우려가 있다.

#### ◦ 장점

- 악의적으로 Timer Interrupt가 발생하기 전에 `yield()`를 호출하는 process를  $L_0$ 에서  $L_2$ 로 내릴 수 있다.
- `sleep()`을 호출하는 process의 time quantum도 올라가기 때문에 process를  $L_2$ 로 내릴 수 있다.

#### ◦ 단점

- Ticks와 Time quantum의 의미가 달라진다.
  - Ticks : Timer Interrupt의 간격
  - Time quantum : scheduling된 횟수
- SLEEPING의 실질적인 의미를 구현할 수 없다.
- SLEEPING 상태에서 완전히 벗어났을 때에,  $L_2$ 에 있을 가능성이 높기 때문에 빠르게 scheduling을 할 수 없다.

#### ◦ 종합

- `yield()`와 `sleep()`에 대해서 모두 time quantum을 올려준다.
- Ticks와 Time quantum의 의미가 달라진다.
- SLEEPING의 실질적인 의미가 달라진다.

두 방법에 대한 우위비교는 SLEEPING 상태를 어떻게 처리하느냐에 대한 문제였다. 아래의 이유 때문에 2번의 방법을 선택했다.

- SLEEPING process가 많을 경우에 모두  $L_0$ 에 있어서 scheduling의 방해를 하는 것보다는,  $L_2$ 에 있는 것이 전체적으로 더 overhead가 작아진다고 생각했다.
- $L_2$  내에서도 priority가 작아지게 되면, SLEEPING에서 깨어났을 때 빠르게 scheduling할 수 있다고 생각했다.
- 최악의 경우를 제외하면, 2번째 방법은 Priority Boosting이 발생하는 경우에도 적절한 우선순위를 유지할 수 있지만, 1번의 방법은 Priority Boosting이 여러 번 발생한 뒤에는 반드시 RUNNABLE process가 가장 나중에 scheduling된다.

## 4-4. Scheduling 철학

Scheduling을 어떻게 하는 것이 가장 최선의 방법일지에 대해서 고민했다.

1. CPU를 최적으로 사용하는 것
2. 사용자의 의도를 최대한 반영하는 방법
3. 최대한 공평한 scheduling을 하는 것

이 중에서 공평한 scheduling에 초점을 맞추어 구현했다. 가치의 우선순위를 3>2>1로 두고 가치가 서로 충돌하는 경우에는 우선순위가 더 높은 방법으로 구현을 하려고 노력했다.

각각의 가치에 대해서 우선순위를 3>2>1로 설정한 이유는 다음과 같다.

### 4-4-1. 공평한 scheduling

공평한 scheduling이 가장 중요한 가치인 이유는 OS는 어떤 process가 중요한 process인지 알 수 없기 때문이다.

비록 process에 priority를 나타내는 변수가 있지만, 이는 사용자가 임의로 설정할 수 있는 값이기 때문에 중요하지 않다고 생각했다.

process가 CPU를 사용하는 정보를 분석하면서 CPU를 할당할 수 있다면 그것이 가장 좋겠지만, 지금은 불가능한 상황이기 때문에 모든 process에게 동등한 가치를 주고 공평한 scheduling을 하고자 노력했다.

### 4-4-2. 사용자의 의도

CPU를 최적으로 사용하는 것보다도 사용자의 의도를 우선한 것은, OS는 사용자가 시스템을 쉽게 사용하도록 도와주는 도구이기 때문이다. 사용자가 효율적이지 않은 실행을 했더라도, 사용자의 의도에 맞는 scheduling을 하는 것이 사용자 입장에서 편리하게 시스템을 사용할 수 있도록 도와주는 방법이라고 생각했기 때문이다.

또한 사용자가 의도하지 않게 잘못된 실행을 하는 경우에는 오류 문구를 통해서 안내하는 방식을 채택했다.

단, 악의적인 조작에 대해서는 의도하든 하지 않았든 허용하지 않고 process를 종료한다. 이는 OS의 안정성을 위해서이다.

### 4-4-3. CPU 최적 사용

CPU를 최적으로 사용하는 것을 최하위에 둔 것은 다른 가치에 밀려서 그런것도 있지만, 이 가치 자체가 multi-Tasking에서 보존하기 어려운 가치라고 생각했기 때문이다.

만약 CPU를 최적으로 사용하는 것이 최고의 가치라면, scheduling에 CPU를 소모하지 말고, 하나의 process만을 계속해서 처리하는 것이 더 효율적일 것이다.

그러나 그렇게 하지 않는 것이 multi-Tasking이고, 여러 개의 process를 동시에 작업하는 것처럼 보이게 하는 것이 사용자의 만족도가 더 높기 때문에 제일 낮은 우선순위의 가치로 두었다.