OS ASSIGNMENT-1 Wiki

2019082279 김영현

1. DESIGN

Xv6 - 명세는 MLFQ가 L0와 L1은 Round Robin방식을 채용하고, L2는 Priority 스케줄링 방식을 채용하기를 원한다. 이를 구현하기 위해서, L0, L1은 기존 xv6 방식을 참고했다. Xv6의 경우, scheduler는 무한 루프를 하면서 runnable 한 process를 찾는다. 이후 runnable process를 찾으면 cpu의 proc을 그 runnable process의 포인터로 지정하며 swtch 과정을 거쳐 running state로 지정하며 이후 다시 swtch를 해서 cpu의 점유 proc의 running을 종료한다. Xv6의 경우 10ms마다 timer interrupt를 발생시키고 trap이 호출되면 timer interrupt가 발생했을 때마다 ticks를 증가시킨다. 이때, 현재 프로세스가 running state이면 yield를 호출해 cpu의 권한이 다시 넘어간다. 이후 ptable의 runnable process를 탐색하여 다시 cpu에 할당하는 방식을 반복한다.

Global tick - 이 방법을 L0와 L1 queue에도 동일하게 적용하였다. 다만, **global tick을 xv6의 자체 ticks가 아닌 MLFQ의 uint gTicks**로 따로 설정하였으며, 기존 xv6의 방식과 마찬가지로 timer interrupt가 발생하고 현재 cpu에 할당된 process가 running이면 yield를 호출한다. 또한 그러한 과정이 cpu상에서 process가 running을 한 번 했다고 판단하여 process의 자체 tick (runTime)을 증가시켰다.

xv6의 기본 스케줄링 방식의 ticks를 global tick으로 이용하지 않은 이유는 다음과 같다. 현재 cpu에 할당된 process가 timer interrupt보다 더 빨리 종료되어 cpu의 현재 process가 null인 경우가 발생하여 현재 process의 상태와 상관없이 xv6의 자체 ticks는 증가하는 상황이 빈번하게 발생하며 이는 timer interrupt를 적절하게 활용하지 않은 것이고 timer interrupt 발생시 cpu를 점유하지 않은 프로세스가 있는 상황에서 cpu에서 프로세스가 running했다고 보는 것은 맞지 않다고 판단하였기 때문이다. 또한, xv6의 자체 ticks를 global tick으로 활용할 경우 timer interrupt발생시 running한 프로세스의 자체 tick (runTime)을 증가시키면 global tick이 100이 되었을 때 priority boosting을 호출하는 상황에서 프로세스들이 각 큐의 time quantum을 충족시키지 못해서 L2나 L1까지 위치하기 전에 L0로 다시 리셋 되는 상황이 빈번하게 발생하기 때문이다. 따라서, timer interrupt가 발생했을 때 현재 cpu 점유 프로세스가 running이면 실제로 cpu에서 running했다고 판단하여 yield를 호출하고 프로세스의 자체 tick과 global tick을 증가시키는 방식을 채택했다.

MLFQ - L0와 L1은 round-robin 방식으로 스케줄링을 하며, L2는 priority 방식을 이용하며, 같은 priority의 경우 FCFS를 이용한다. Round-robin방식을 위해서 **L0와 L1은 circular queue**를 채택했다. Circular queue는 일반 queue와 다르게 scheduling 과정에서 enqueue와 dequeue를 반복해도 queue size를 front나 rear 가 넘어설 걱정을 하지 않아도 된다는 점에서 좋다고 판단하였다. 또한 **L2**의 경우는 priority 방식을 이용하기 때문에 priority 정렬이 수월하게 **minHeap을 바탕으로 구현한 priority queue**를 채택했다.

Scheduler L0 L1 L2 MLFQ

L0 – 스케줄링은 L0부터 runnable process가 있는지 탐색한다. 이후 단순히 L0 queue에서 dequeue를 하여 runnable이면 cpu에 할당하고 아니면 enqueue한다. Queue를 이용함에도 불구하고 runnable process를 탐색하는 것은 비효율적이라 볼 수도 있지만 이는 process가 L2에 도달하기 전에 자식 process를 가져서 sleep 상태가 될 경우 runnable이 아니면 dequeue와 enqueue를 무한 반복하는 방식에서 해당 queue에서 무한 loop를 도는 것을 방지하기 위한 것이다. 이후, enqueue를 하기 전에 process의 queue level이 해당 queue에 적합한지, zombie process인지 필터링하는 과정을 거친다. Queue level의 경우, yield를 통해 process의 runtime tick을 증가시키고 이후 해당 queue의 time quantum을 다 채웠으면 다음 queue에 enqueue하는 방식을 사용하였기 때문이다. 쉽게 표현하면

(scheduler) dequeue L0-> p running && time interrupt -> (trap)yield -> p runtime++ -> if satisfy time quantum -> enqueue L1 -> (sched) p runnable -> swch -> (scheduler) if enqueue L0?

과정을 거치기 때문이다.

L1 – L1도 L0와 방식이 같다. 다만 L1의 time quantum을 만족시켰을 경우 priority queue로 구현한 L2에 enqueue를 하기 때문에 L1 자체의 enqueue 와 dequeue 방식은 L0와 동일하지만 L2로의 enqueue는 priority queue의 enqueue이다.

L2 – L1에서 L2로 처음 enqueue되는 process들은 priority가 3으로 초기화 되어있다. L1에서 L2로의 enqueue 과정에서 L2 내부의 정렬 과정에서 priority가 동일할 경우 FCFS 방식에 사용하기 위한 L2time을 MLFQ의 global tick으로 초기화 시킨다. 기본 enqueue와 dequeue는 priority queue의 방식을 사용한다. 단, priority가 같을 경우 L2time을 바탕으로 더 적은 global tick이 기록된 process가 처리된다. 이후 dequeue를 하고 runnable process일 경우 cpu 처리를 위해 running state로 변환시키고 다시 runnable로 돌아오면 enqueue를 한다. 하지만 이러한 과정에서 priority가 가장 낮은, 우선순위가 가장 높은 process가 runnable이 아닐 경우가 발생하여 무한 루프가 발생할 수도 있는데, 이를 enqueue할 때 runnable이 아니면 정렬을 시키지 않고 뒤에 추가시키는 방식으로 극복한다.

Yield - time interrupt가 발생하고 현재 cpu 점유 프로세스가 running이면 프로세스가 running했다고 판단하며 이후, 다른 프로세스로 넘어가기 위해 yield를 호출한다. yield에서는 yield가 호출되는 상황이 cpu에서 프로세스가 1 tick을 running 한 것이라고 규정했다. 그래서 yield에서 global tick인 MLFQ 구조체의 gTicks를 증가시키며, priority boosting이 일어나지 않았고 lock된 프로세스가 아니라면 프로세스 자체의 runTime tick또한 증가시켰다. 그 후, 각 프로세스가 dequeue하기 전에 위치했던 queue에 맞는 time quantum과 비교하여 전부 채웠을 경우 다음 queue로 넘어가거나 L2의 경우 priority를 감소시킨다. 또한 global tick이 100이 되었을 경우, priority boosting을 호출하여 dequeue한 상태의 현재 process를 L0 맨 처음에 enqueue하며, 이후 L0와

L1, L2에 있는 프로세스들을 모두 L0에 enqueue시킨다.

Scheduler Lock, UnLock - MLFQ에서 프로세스를 스케줄링하기 전에 스케줄링이 되어야 하는 프로세스를 설정할 수 있어야 한다. 하지만 schedulerLock과 unlock은 부여하는 권한을 체크하기 위한 용도인 학번만을 인자로 받으며 이는 cpu에서 running 중이며 MLFQ방식으로 처리되고 있는 프로세스에 lock을 부여하거나 unlock하기 위한 것이라 해석했다. scheduler lock 시스템 콜이 호출되면 현재 실행중인 프로세스가 다시 swtch되어서 scheduler의 queue로 enqueue하기 전에 lock 권한을 표시하기 위한 flag로 enqueue할지 말지 정하는 과정이 필요하다고 판단해서 isScheduleLocked flag를 첨가했다. Schedule lock된 프로세스가 스케줄링이 되어 cpu에 할당하는 과정은 MLFQ 스케줄러가 작업을 수행하기 전에 해당 프로세스를 스케줄하는 것을 반복하는 방식으로 구현하였다. 스케줄 과정에서도 기존의 timer interrupt와 cpu running process일 경우 1 tick씩 yield하는 과정은 동일하기 때문에 동일하게 global tick이 증가하고, global tick이 100일 경우 priority boosting이 호출되며 이때, scheduler unlock을 수행하여 다시 L0에 처음으로 해당 프로세스를 enqueue해서 MLFQ 스케줄링 방식으로 작동하게 된다. 만약, 인자를 잘못 입력했을 경우, 해당 프로세스를 kill한다. Lock이 되지 않은 프로세스에서 unlock을 시도했을 경우, 경고 메시지를 출력하고 별다른 행동을 취하지 않아 기존의 MLFQ 스케줄링 방식에서 정상 작동되도록 한다. schedulerLock의 경우, 다양한 state에서의 프로세스를 schedulerLock 할 수는 있지만 결국 locking이 되는 프로세스는 schedulerLock의 구조상 cpu에서 running하는 프로세스여야 한다고 판단했기 때문에 running 프로세스가 아닐 경우 에러 메시지를 출력하고 return한다. schedulerUnlock의 경우, 해당 시스템콜이 호출되는 상황은 두 가지이다. cpu에서 running하는 프로세스를 unlock하거나, 이미 lock이 되었지만 cpu에서 처리하지 못하는 프로세스를 unlock 시도할 때이다. 특히 lock되어 단독으로 스케줄링 되던 프로세스가 zombie state로 변하는 경우 runnable state가 아니기 때문에 unlock을 해주는 방식으로 구현했다.

2. Code summary

```
struct {
  struct spinlock lock;
  struct proc *lockedProc;
  struct proc proc[NPROC];
} ptable;
```

(proc.c) Ptable에 lock된 프로세스를 나타내기 위한 lockedProc을 추가해 주었다.

```
//circular queue for RR
typedef struct _Queue{
    int front;
    int rear;
    struct proc *p[QUEUE_MAX_SIZE];
}Queue;
//prioiry queue using min heap
typedef struct _pQueue{
    int count;
```

```
    struct proc *p[QUEUE_MAX_SIZE];
}pQueue;

typedef struct _MLFQ{
    uint gTicks;
    Queue *L0;
    Queue *L1;
    pQueue *L2;
}Mlfq;
//L0 queue
Queue q0 = {0, 0, {0, }};
//L1 queue
Queue q1 = {0, 0, {0, }};
//L2 queue
pQueue q2 = {0, {0, }};
//mlfq
Mlfq mlfq = {0, &q0, &q1, &q2};
```

(mlfq.h) Mlfq 구조체와 (proc.c) 초기화되는 mlfq

```
static struct proc*
allocproc(void)
```

(proc.c) allocproc시 프로세스를 초기화해주고 L0에 enqueue한다.

```
    /**************
     SCHEUDLER LOCK
    ***************/
    if(ptable.lockedProc){
```

(proc.c)의 scheduler에서 처리하는 scheduler lock된 프로세스

```
    /**************
    MLFQ SCHEDULING
    ***************/
    struct proc *p;
    //Is there RUNNABLE pro in L0
    if(queue_isThereRunnable(mlfq.L0))
```

(proc.c)의 scheduler에서 scheduler lock된 프로세스가 없을시 처리하는 MLFQ scheduler

```
  int pbflag = 0;
  if(mlfq.gTicks == 100){
    pbflag = 1;
    priorityBoosting();
  }
  mlfq.gTicks++;
```

(proc.c)의 yield에서 처리하는 global tick 100일시 priority boosting을 호출한다

```
  if(!pbflag){
    struct proc *p = myproc();
    switch(p->qLevel){
```

```
      case 0:
        if(p->runTime < 4){
          p->runTime++;
          break;
        }
        if(p->runTime >= 4){
          queue_timeQuantumOverflow_toQ(myproc());
```

(proc.c)에서 global tick이 100이 아니면 running process의 runtime을 1 증가시키며 timeQuantum이 충족되었을 경우 다음 queue로 enqueue한다.

```
void
priorityBoosting(void)
```

(proc.c)에서 priority boosting을 호출한 running process가 schedulerLock된 process면 schedulerUnlock을 호출하고 이후 priority boosting 과정을 수행한다.

```
void
schedulerLock(int password)
void
schedulerUnlock(int password)
```

(proc.c)에서 schedulerLock 조건들을 바탕으로 출력을 다르게 하거나 프로세스의 isScheduleLocked flag를 설정해주거나 kill한다

```
  else if(tf->trapno == T_SCHEDLOCK){
  else if(tf->trapno == T_SCHEDUNLOCK){
```

(trap.c) 시스템 콜을 interrupt로 호출하는 경우를 구현했다.

```
  if(myproc()&&myproc()->state == RUNNING && tf->trapno == T_IRQ0 +
IRQ_TIMER){
    yield();
  }
```

(trap.c) 기존 xv6의 yield 방식과 같다

```
int
sys_schedulerLock(void)
```

(sysschedulerLock.c)

```
int
sys_schedulerUnlock(void)
```

(sysschedulerUnlock.c)

```
int
sys_setPriority(void)
```

(syssetPriority.c)

```
int
sys_yield(void)
```

(sysyield.c)

```
int
sys_getLevel(void)
```

(sysgetLevel.c)

3. Result

Testcode는 mlfq_test.c 와 다른 테스트 코드를 수정한 a_test.c이다.

Xv6 폴더 안에서 **make clean -> make -> make fs.img -> ./bootxv6.sh** 를 통해 실행했다.

bootxv6.sh는 실습 때와 동일하다.

(mlfq_test)

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ mlfq_test
MLFQ test start
[Test 1] default
Process 4
L0: 8700
L1: 10166
L2: 81134
Process 5
L0: 9223
L1: 22611
L2: 68166
Process 6
L0: 30311
L1: 28378
L2: 41311
Process 7
L0: 21772
L1: 32676
L2: 45552
[Test 1] finished
done
$
```

(a_test-1~3)

```
init: starting sh
$ a_test
MLFQ test start

Focused priority
process 4: L0=1906, L1=3007 L2=45087
process 5: L0=3773, L1=43413 L2=2814
process 6: L0=23507, L1=16595 L2=9898
process 7: L0=6570, L1=9750 L2=33680
process 8: L0=5841, L1=35772 L2=8387

Without priority manipulation
process 9: L0=8248, L1=105962, L2=185790
process 13: L0=16456, L1=150242, L2=133302
process 12: L0=37572, L1=47435, L2=214993
process 10: L0=46664, L1=62838, L2=190498
process 11: L0=50414, L1=204924, L2=44662

With yield
process 14: L0=19239, L1=26926, L2=153835
process 18: L0=31251, L1=43736, L2=125013
process 15: L0=39486, L1=55280, L2=105234
process 17: L0=45464, L1=63657, L2=90879
process 16: L0=50001, L1=69999, L2=80000
```

(a_test-4)

```
schedulerLock
setting scheduler Lock
ticks: 2, pid: 21, isLock: 1
ticks: 3, pid: 21, isLock: 1
ticks: 4, pid: 21, isLock: 1
ticks: 5, pid: 21, isLock: 1
ticks: 6, pid: 21, isLock: 1
ticks: 7, pid: 21, isLock: 1
ticks: 8, pid: 21, isLock: 1
ticks: 9, pid: 21, isLock: 1
ticks: 10, pid: 21, isLock: 1
ticks: 11, pid: 21, isLock: 1
ticks: 12, pid: 21, isLock: 1
ticks: 13, pid: 21, isLock: 1
ticks: 14, pid: 21, isLock: 1
ticks: 15, pid: 21, isLock: 1
ticks: 16, pid: 21, isLock: 1
ticks: 17, pid: 21, isLock: 1
ticks: 18, pid: 21, isLock: 1
ticks: 19, pid: 21, isLock: 1
ticks: 20, pid: 21, isLock: 1
ticks: 21, pid: 21, isLock: 1
ticks: 22, pid: 21, isLock: 1
ticks: 23, pid: 21, isLock: 1
ticks: 24, pid: 21, isLock: 1
ticks: 25, pid: 21, isLock: 1
ticks: 26, pid: 21, isLock: 1
ticks: 27, pid: 21, isLock: 1
ticks: 28, pid: 21, isLock: 1
ticks: 29, pid: 21, isLock: 1
ticks: 30, pid: 21, isLock: 1
ticks: 31, pid: 21, isLock: 1
ticks: 32, pid: 21, isLock: 1
```

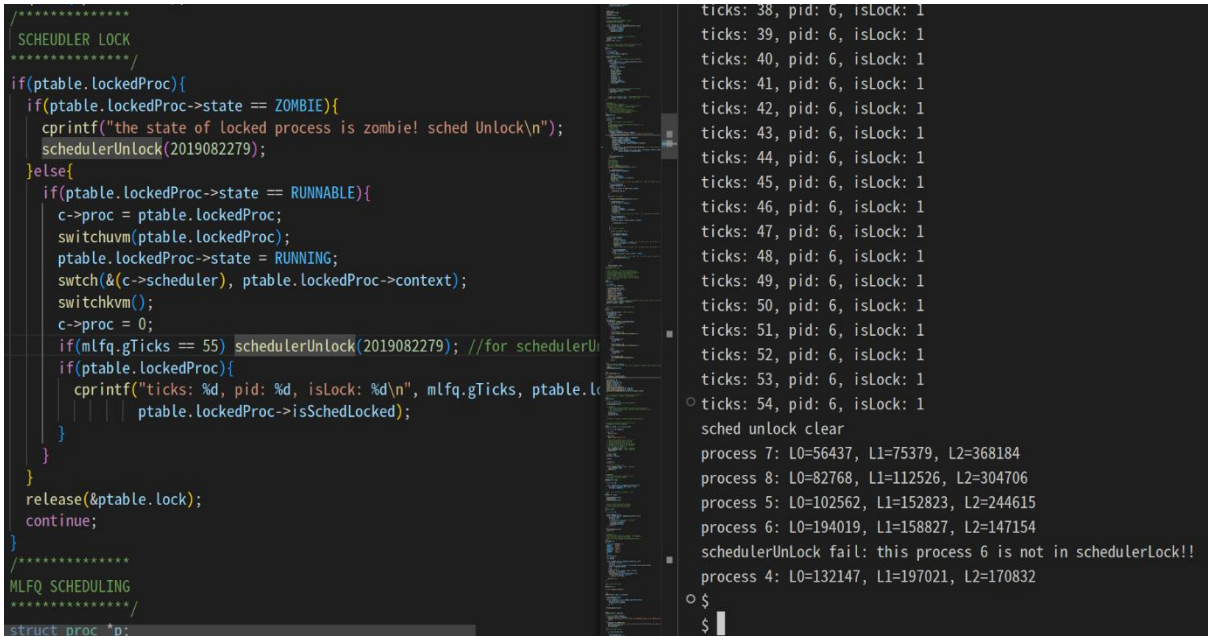
중간 생략

```
ticks: 86, pid: 21, isLock: 1
ticks: 87, pid: 21, isLock: 1
ticks: 88, pid: 21, isLock: 1
ticks: 89, pid: 21, isLock: 1
ticks: 90, pid: 21, isLock: 1
ticks: 91, pid: 21, isLock: 1
ticks: 92, pid: 21, isLock: 1
ticks: 93, pid: 21, isLock: 1
ticks: 94, pid: 21, isLock: 1
ticks: 95, pid: 21, isLock: 1
ticks: 96, pid: 21, isLock: 1
ticks: 97, pid: 21, isLock: 1
ticks: 98, pid: 21, isLock: 1
ticks: 99, pid: 21, isLock: 1
ticks: 100, pid: 21, isLock: 1
sched unlock clear
process 21: L0=180266, L1=52754, L2=266980
schedulerUnLock fail: this process 21 is not in schedulerLock!!
process 22: L0=72819, L1=99813, L2=327368
process 19: L0=100423, L1=132992, L2=266585
process 23: L0=118218, L1=151734, L2=230048
process 20: L0=121501, L1=180799, L2=197700
$
```

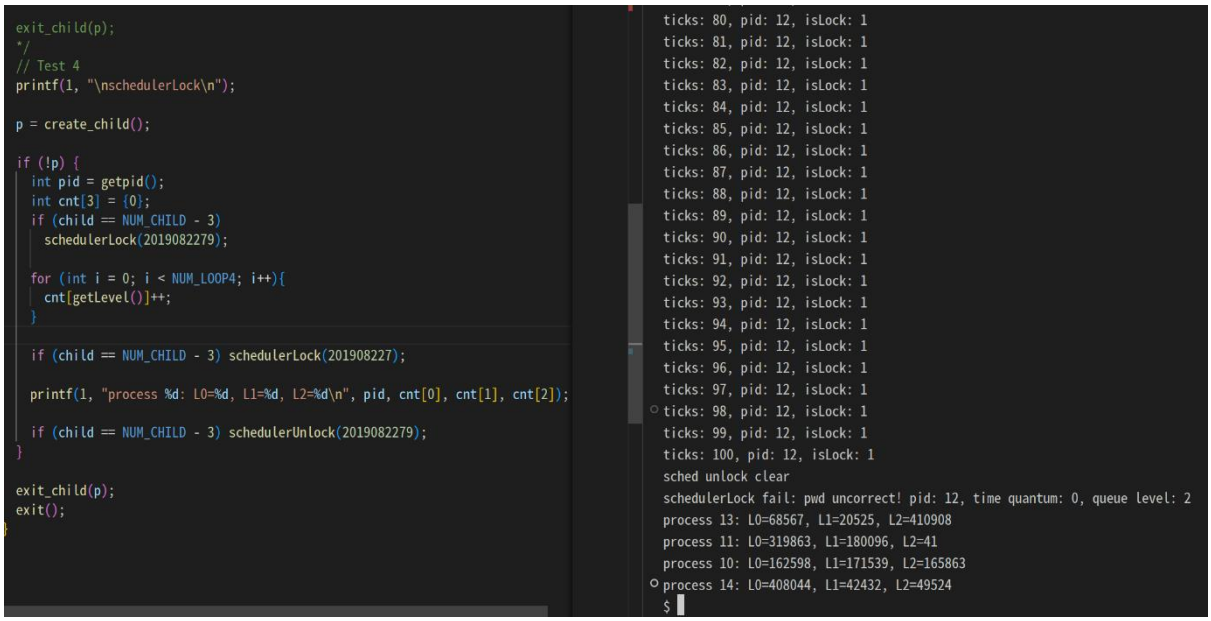
(interrupt to systemcall)

```
$ schedLock
129 pid 24, try schedulerLock
setting scheduler Lock
the state of locked process is zombie! sched Unlock
sched unlock clear
$ schedUnlock
130 pid 25, try schedulerUnLock
schedulerUnLock fail: this process 25 is not in schedulerLock!!
$
```

(**a_test** – proc.c에서 lock된 프로세스 처리하는 부분에서 55틱일 때 schedUnlock을 시도하는 주석을 수정하여 코드로 변환한다음 a_test는 test 1~3까지 주석 처리하여 수행한 상황, schedulerLock된 프로세스를 스케줄링하는 과정 중에 schedulerUnlock을 시도하는 상황 테스트)



(**a_test** – a_test.c에서 test 1~3까지 주석 처리 한 후, test 4의 주석 처리된 부분을 해제하여 schedulerLock에서 인자를 잘못 입력했을 상황 구현)



4. Trouble shooting

schedulerLock과 schedulerUnlock을 테스트하는 방법을 올바르게 했는지에 대한 의문이 든다. schedulerLock 자체가 학번을 인자로 받아서 호출 되는데, 다른 인자를 받지 않기 때문에 running process만을 처리한다. 이 부분을 초점에 맞춰서 구현하려고 했다. schedulerLock과 unlock의 상황

을 테스트하기 위해 a_test.c와 proc.c의 해당 부분을 적절하게 변경하여 시도했다.

queue자료형을 이용해서 구현했을 경우 확실한 시간 상의 이점이 있어야 한다고 생각하는데 결국 스케줄러에서 runnable이 존재하는지 탐색하는 과정을 거쳐서 과연 이득을 얻었는지에 대한 의문이 든다. 하지만 priorityboosting에서 해당 프로세스를 L0에 먼저 enqueue하여 처리하는 방식을 구현하기 수월했으며 L2에서 priority 정렬 과정을 수월하게 할 수 있었으며 이 부분에서 배열방식보다 다소 빠를 것이라 예측한다.

scheduler에서 mycpu에 프로세스를 할당하고 running state로 변경했음에도 trap에서 timer interrupt가 발생할 때 xv6 자체 ticks은 증가함에도 불구하고 cpu에 할당된 프로세스가 없거나 running인 상황이 아닌 경우가 발생했다. Timer interrupt를 이용해야 하는 것이 의도된 바라 생각했고 xv6의 자체 ticks로 global ticks를 하고자 했으나 이러한 상황 때문에 global ticks가 증가하는 것보다 scheduler에서의 running process가 더 빨리 종료되는 경우가 발생했다. 이 점을 아예 다르게 생각하여 실제로 cpu가 처리하는 상황은 timer interrupt가 발생하고 cpu에 running process가 존재할 때라 정했으며 xv6의 ticks가 아닌 따로 지정한 mlfq.gTicks를 global tick이라 정의했다.