Operating System Project 1 Wiki

2019092306 구본준

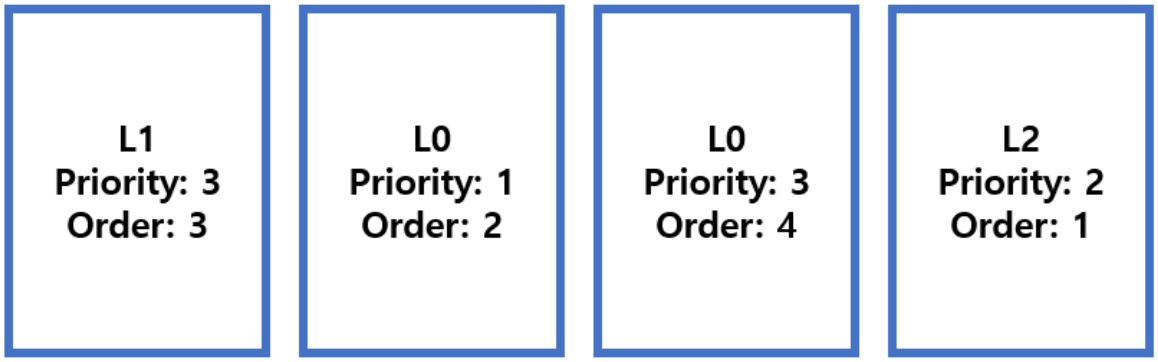
Design

1. MLFQ

L0, L1, L2 queue를 실제 자료구조 queue로 구현하는 것이 아닌 기존에 있던 배열에서 프로세스를 뽑는 순서만 조절하여 abstract하게 구현할 계획이다. 먼저 기존 ptable 자료구조는 유지하되 각 프로세스 proc 구조체가 time quantum, queue level, priority, queue에 들어온 순서 정보를 갖도록 변경해야 한다. Scheduler 함수로 들어올 때마다 ptable 전체를 순회하게 되는데 이때 각 프로세스의 정보를 가지고 가장 우선되는 프로세스를 선정한다. 이때 기준은 다음과 같다.

Queue level이 낮은가 → priority가 더 높은가 (L2 queue 인 경우만) → queue에 들어온 순서가 빠른가

Queue에 들어온 순서는 ptable의 변수로부터 할당을 받아야 한다. 매 tick 마다 1씩 증가하는 변수의 값을 프로세스가 받아 저장하여 그 프로세스가 몇 번째 tick에 queue에 들어왔는지 알 수 있도록 한다.



위와 같은 예시로 ptable의 정보가 저장되어 있는 경우 먼저 queue level이 가장 낮은 두 번째 세 번째 프로세스가 선정되어야 하며, L0 queue이므로 priority는 신경쓰지 않고 order만 비교하여 더 값이 작은 두 번째 프로세스를 최종 스케줄링해야 한다.

스케줄링은 매 tick마다 발생하도록 하고 각 queue에서 돌아가며 time quantum을 다 소진한 프로세스는 다음 queue로 이동하게 된다. 앞에서 계획한 내용처럼 실제 이동은

하지 않고 프로세스 구조체의 queue level 값만 1 증가시켜준다.

2. Boosting

Priority boosting은 global ticks가 100이 될 때마다 실행되며 global ticks는 100이 될 때 다시 0으로 초기화되도록 구현할 것이다. Boosting이 실행되면 ptable에 있는 모든 프로세스의 time quantum, queue level, priority 값을 초기화해준다. 이 때 boosting 후 프로세스들의 순서를 유지하기 위하여 sorting 알고리즘이 필요하며 이를 quick sort로 구현할 계획이다

3. Scheduler Lock

Scheduler lock 함수는 ptable에 새로운 변수를 만들어 구현할 계획이다. Scheduler lock을 잡은 프로세스가 있는지 여부를 ptable 내부 변수에 그 프로세스의 주소를 저장함으로써 알도록 한다. Scheduling이 될 때 MLFQ가 실행되기 전 scheduler lock을 잡은 프로세스가 있는지 확인한 후 만약 있다면 그 프로세스를 다음 프로세스로 스케줄링 해주도록 만든다. Scheduler unlock 함수의 경우는 단순히 만들어 놓았던 ptable 내부의 변수를 초기화 해주고 프로세스의 값을 L0의 가장 높은 순서에 오도록 설정해준다.

Implement

1. ptable

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
  int ordernum;
  struct proc *lockproc;
} ptable;
```

ptable 구조체이다. 기존 구조에 *ordernum, lockproc* 변수를 추가하였다. *ordernum* 변수의 경우 매 tick마다 1씩 증가하며 boosting이 일어나는 경우 *NUMLOC* 값으로 초기화된다. 프로세스의 순서 정보를 부여하여 각 queue 내부에서 어떤 프로세스가 먼저 들어왔는지 구분하기 위해 사용된다. *lockproc* 변수는 scheduler lock을 위해 존재하며 *lockproc* 값이 세팅 되어있는 경우 MLFQ가 대신 *lockproc* 변수에 연결되어 있는 프로세스를 scheduling 해준다.

2. proc

```
  int priority;                // Process priority (0~3)
  int ticks;                   // Current used time quantum
  enum queuelevel qlevel;      // Current queue level
  int order;                   // Process order given by ptable
```

Proc 구조체이다. 기존 구조에 네 가지 변수를 추가하였다. *priority* 변수는 이름 그대로 priority 값을, *ticks* 변수는 사용한 time quantum 값을, *qlevel* 변수는 현재 프로세스가 위치하고 있는 queue level을 나타낸다. *order* 변수는 ptable의 *ordernum* 변수에서 받은 값을 저장하고 있다.

3. allocproc()

```
 // Process init.
  p->qlevel = L0;
  p->priority = 3;
  p->ticks = 0;
  p->order = ++ptable.ordernum;
```

allocproc() 함수 내부에 추가한 코드이다. 프로세스가 새롭게 할당되었을 때 proc 구조체에 추가하였던 변수들도 초기화해준다.

4. scheduler()

```
// Scheduller Lock, if scheduler locking process exists and it is runnable.
    if(ptable.lockproc != 0 && ptable.lockproc->state == RUNNABLE) {
      np = ptable.lockproc;
    }
```

lockproc 변수가 세팅 되어 있는지 확인하여 만약 그렇다면 scheduler lock을 잡고 있는 프로세스를 스케줄링 한다. *np* 변수에 프로세스의 주소 값을 넣어준다.

```
// MLFQ
    else {
      // Reset the scheduler locking process.
      ptable.lockproc = 0;

      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
          continue;

        // First find runnable process.
        if(np == 0)
```

```
        np = p;
      // Compare current process and newly selected process.
      // Update np to preferred process.
      else
        np = compProc(np, p);
    }

    // Scheduler could not find runnable process.
    if(np == 0) {
      release(&ptable.lock);
      continue;
    }
  }
```

Lockproc 변수가 세팅 되어 있지 않거나 *lockproc* 변수가 가리키는 프로세스가 *RUNNABLE* 하지 않으면 MLFQ로 스케줄링 된다. 이때 *RUNNABLE* 하지 않은 이유는 프로세스가 *exit()*로 종료되었거나 *sleep()* 상태인 경우이므로 lock을 해제하기 위하여 *lockproc*을 초기화해준다. 이후 ptable을 전부 돌면서 다음 순서로 스케줄링 할 프로세스를 결정한다. *RUNNABLE* 하지 않은 프로세스를 제외하고 *compProc()* 함수를 이용하여 현재 프로세스와 다음 프로세스를 비교하며 *np* 변수를 업데이트한다. 최종적으로 *np*에 들어있는 값이 0이 아니라면 나머지 스케줄링 과정을 진행한다. 이후 과정은 기존 xv6의 스케줄러와 동일하다.

5. yield()

```
// For MLFQ scheduler, scheduler lock is not excuted
  if(ptable.lockproc == 0) {
    // If the process has exhausted its time quantum.
    if(++myproc()->ticks >= 2*myproc()->qlevel+4) {
      myproc()->ticks = 0;
      // If the process has been in L0 or L1, moves to the next queue.
      if(myproc()->qlevel < L2)
        myproc()->qlevel++;
      // If the process has been in L2, increases priority (decreases number).
      else if(myproc()->priority > 0)
        myproc()->priority--;
    }
    // Update order in the last place (to the biggest order number)
    myproc()->order = ++ptable.ordernum;
  }
```

1tick마다 스케줄링이 일어났을 때 MLFQ를 위한 프로세스 정보를 업데이트 해주기 위해 추가한 코드이다. Scheduler lock이 잡혀 있을 경우 업데이트 하지 않도록 구현하여 만약

scheduler lock이 실행된다면 그 시점의 정보가 계속 유지된다. 기본적으로 *ticks* 변수를 1씩 증가시켜주며 만약 그 queue level에서의 time quantum을 모두 사용하였으면 queue level을 증가시키거나(L0, L1인 경우) priority 값을 감소시킨다(L2인 경우).

6. getLevel()

```
// Return the queue level of the current process.
int
getLevel(void)
{
  return myproc()->qlevel;
}
```

현재 실행되고 있는 프로세스의 queue level을 return해준다.

7. setPriority()

```
// Set priority of the given-pid process.
void
setPriority(int pid, int priority)
{
  struct proc *p;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid)
      p->priority = priority;
  }

  release(&ptable.lock);
}
```

Parameter로 입력 받은 pid의 priority를 세팅해준다. ptable을 돌면서 일치하는 pid를 찾은 후 업데이트하는 방식으로 구현하였다. 만약 일치하는 pid가 없다면 아무 일도 일어나지 않도록 하였다.

```
if(priority < 0 || priority > 3)
    return -1;
```

setPrioirity() 함수의 wrapper function인 *sys_setPriority()* 함수에서 들어온 *priority* 인자 값이 0~3 범위가 아닐 경우 -1을 return해주는 예외처리를 해주었다.

8. schedulerLock()

```
// If the password is wrong. Yield the process immediately.
  if(password != PASSWORD) {
    cprintf("wrong password: pid %d, time quantum %d, queue level %d\n",
            myproc()->pid, myproc()->ticks, myproc()->qlevel);
    yield();
    return;
  }
```

비밀번호가 틀렸을 경우의 예외처리이다. 현재 프로세스의 pid, time quantum, queue level을 출력한 후 프로세스를 종료하고 다시 스케줄링 한다. 이때 출력하는 프로세스의 정보는 프로세스가 scheduler lock을 잡기 직전의 갖고 있던 정보이다. 비밀번호가 틀렸기 때문에 당연히 다음 함수의 과정은 진행되지 않는다.

```
// Set the sceduler locking process to the current process.
  acquire(&ptable.lock);
  ptable.lockproc = myproc();
  release(&ptable.lock);

  resetticks();
```

비밀번호가 맞게 들어왔을 경우 실행되는 부분이다. ptable의 *lockproc* 변수를 현재 프로세스의 주소로 업데이트 해준다. 이후 100ticks 동안의 사용을 보장하기 위하여 global ticks의 값을 0으로 초기화해준다.

9. schedulerUnlock()

```
if(password != PASSWORD) {
    cprintf("wrong password: pid %d, time quantum %d, queue level %d\n",
            myproc()->pid, myproc()->ticks, myproc()->qlevel);

    // Reset the scheduler locking process.
    acquire(&ptable.lock);
    ptable.lockproc = 0;
    release(&ptable.lock);

    yield();
    return;
  };
```

비밀번호가 틀렸을 경우 나머지는 scheduler lock과 같으며 ptable의 *lockproc* 변수를 0으로 초기화하여 lock이 해제되도록 하였다.

```
acquire(&ptable.lock);

  // If it is called before schedulerLock function
  if(ptable.lockproc == 0) {
    release(&ptable.lock);
    return;
  }

  // Move the current process to the front of the L0 queue.
  myproc()->qlevel = L0;
  myproc()->priority = 3;
  myproc()->order = 0;

  // Reset the scheduler locking process.
  myproc()->state = RUNNABLE;
  myproc()->ticks = 0;
  ptable.lockproc = 0;

  sched();

  release(&ptable.lock);
```

비밀번호가 맞게 들어온 경우 먼저 예외처리로 현재 scheduler lock이 잡혀 있는 상황이 맞는지 확인한다. 맞게 확인이 되면 프로세스가 L0 queue의 가장 앞에 올 수 있도록 프로세스 정보를 업데이트 해준 후 *yield()* 함수가 아닌 *sched()* 함수를 호출하여 추가적인 업데이트 없이 스케줄러로 바로 넘어갈 수 있도록 구현하였다.

10. compProc()

```
// Compare which process has preference.
struct proc*
compProc(struct proc* procA, struct proc* procB)
{
  // Compare the queue level.
  if(procA->qlevel < procB->qlevel)
    return procA;
  else if(procA->qlevel > procB->qlevel)
    return procB;
  else if(procA->qlevel == L2) {
    // Compare the priority if both A and B are in the L2 queue.
    if(procA->priority < procB->priority)
      return procA;
    else if(procA->priority > procB->priority)
      return procB;
    // Compare the order if the priority is same.
```

```
    else if(procA->order < procB->order)
      return procA;
    else
      return procB;
  }
  // Compare the order if both A and B are in the same queue (except L2).
  else if(procA->order < procB->order)
    return procA;
  return procB;
}
```

두 프로세스의 포인터 값을 받아 어떤 프로세스가 앞서서 스케줄링 되어야 하는지 결정해주는 함수이다. 먼저 queue level을 비교하여 더 작은 프로세스를 return해준다. 만약 같다면 L2 queue 내부인지 확인하여 그런 경우 *priority*를 비교하여 더 높은 프로세스를 return해준다. 이후 queue에 들어올 때 부여받은 *order*값을 비교하여 누가 더 먼저 queue에 들어왔는지를 확인한 후 더 작은 프로세스를 return해준다.

11. boosting()

```
// Reset processes.
  for(int i = 0; i < NPROC; i++){
    p = &ptable.proc[i];
    p->qlevel = L0;
    p->priority = 3;
    p->ticks = 0;
    temp[i] = p;
  }
```

ptable을 돌며 프로세스의 정보를 초기화해준다. *temp*라는 임시 배열을 만들어 ptable값을 복사해주는데 이는 boosting이 된 후에 이전 순서를 유지하기 위한 과정에 사용하기 위해서이다. 직접 *ptable.proc*의 값을 정렬하는 것 보다 *temp* 배열을 정렬한 후 정렬한 순서대로 프로세스의 *order* 값들을 설정해주는 것이 포인터가 꼬일 위험이 없어 안전하다고 판단하여 이러한 방식으로 구현하였다.

```
// For preserving processes order.
  quicksort(temp, 0, NPROC - 1);
  for(int i = 0; i < NPROC; i++)
    temp[i]->order = i + 1;

  // If the scheduler locking process exists.
  // Set order to the front of the L0 queue.
  if(ptable.lockproc != 0) {
    ptable.lockproc->order = 0;
    ptable.lockproc = 0;
```

```
  }

  // Reset the ordernum
  ptable.ordernum = NPROC;
```

temp 배열을 quicksort를 통해 정렬한 후 정렬한 순서대로 프로세스의 order 값을 업데이트 해준다. 이후 만약 scheduler lock이 잡혀 있었다면 그 프로세스를 가장 앞 순서로 설정해야 하므로 *order* 값으로 0을 넣어준다. 다음 스케줄링부터는 *NPROC+1* 숫자부터 *order* 값을 부여해줘야 하므로 *ptable.ordernum*을 *NPROC* 값으로 초기화해 준다.

12. partition(), quicksort()

```
// Partition for quicksort
int
partition(struct proc **arr, int low, int high)
{
    struct proc *pivot = arr[high], *temp;
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        // Using compProc function!
        if (compProc(arr[j], pivot) == arr[j]) {
            i++;

            // Swap arr[j] and arr[i]
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // Swap arr[i + 1] and arr[high]
    temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = arr[i + 1];

    return (i + 1);
}

// quicksort
void quicksort(struct proc **arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quicksort(arr, low, pi - 1);
```

```
        quicksort(arr, pi + 1, high);
    }
}
```

Quick sort를 위한 *partition()* 함수와 *quicksort()* 함수이다. 알고리즘의 기본적인 코드와 거의 동일하며 다른 점은 proc의 포인터 값을 인자로 받으며 비교를 하는 부분이 *compProc()* 함수로 대체되어 있다는 점이다.

13. resetticks()

```
// Reset the global ticks.
void
resetticks(void)
{
  acquire(&tickslock);
  ticks = 0;
  release(&tickslock);
}
```

Global ticks 값을 0으로 초기화해 준다.

14. trap()

```
// Priority boosting on 100 ticks.
  if(tf->trapno == T_IRQ0+IRQ_TIMER && ticks == 100)
    boosting();
```

trap() 함수 내부에서 타임 인터럽트를 받았을 때 만약 *ticks*값이 100이면 boosting을 실행하도록 구현하였다.

Result

1. 컴파일 및 실행과정

```
make CPUS=1
make fs.img
```

컴파일 시 CPUS=1 인자를 통해 CPU 개수를 하나로 설정해주었다.

```
qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m
512
```

부팅을 위한 명령어이다.

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 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
```

부팅이 정상적으로 완료된 모습이다. 이후 mlfq_test, schedlock_test를 입력하면 각각 MLFQ, scheduler lock 테스트 코드가 실행된다.

2. MLFQ 테스트 코드

```
MLFQ test start
[Test 1] default
Process 5
L0: 15583
L1: 19880
L2: 64537
L3: 0
L4: 0
Process 6
L0: 15593
L1: 20609
L2: 63798
L3: 0
L4: 0
Process 7
L0: 19377
L1: 28694
L2: 51929
L3: 0
L4: 0
Process 8
L0: 19271
L1: 27969
L2: 52760
L3: 0
L4: 0
[Test 1] finished
done
```

MLFQ가 의도한대로 동작하는지 테스트하는 코드이다. 프로세스 네 개를 fork()하여 각각 loop를 100,000번씩 돌도록 한다. 이 때 프로세스가 L0, L1, L2 queue 중 어디에 위치해 있는지를 출력한다. 기본적으로 L2에 50~60%, L1에 20%, L0에 15%가량 위치해 있다는 결과가 나왔으며 의도한 비율대로 적절히 나온 것을 알 수 있다. 또한 프로세스의 순서도 fork()된 순서대로 유지가 되었는데 이 경우는 모든 프로세스가 boosting이 일어났을 시 같은 queue에 있었기 때문이다. 만약 5번 프로세스만 L2 queue에 있고 6, 7,

8번 프로세스가 L1 queue에 있는 상태로 boosting이 일어났다면 5번 프로세스가 맨 뒤에 가도록 정렬되기 때문에 순서가 바뀔 수 있다.

3. Scheduler Lock 테스트 코드

```
Scheduler Lock test start
[Test 1] default
Process 12
L0: 100000
L1: 0
L2: 0
L3: 0
L4: 0
Process 9
L0: 10331
L1: 12935
L2: 76734
L3: 0
L4: 0
Process 10
L0: 13374
L1: 19456
L2: 67170
L3: 0
L4: 0
Process 11
L0: 13830
L1: 19096
L2: 67074
L3: 0
L4: 0
[Test 1] finished
done
```

MLFQ 테스트 코드에서 가장 늦게 fork()되는 프로세스에 scheduler lock을 걸어주도록 변경한 코드이다. Scheduler lock을 걸어준 프로세스가 가장 먼저 실행이 끝난 것을 볼 수 있다. Scheduler lock을 실행한 시점의 queue level 정보가 그대로 유지되도록 구현하였기 때문에 L0 queue 정보를 유지하고 있어 L0에만 100,000이 찍히게 되었다.

Trouble Shooting

1. ptable lock 문제

```
    // Scheduler could not find runnable process.
    if(np == 0) {
      release(&ptable.lock);
      continue;
```

scheduler() 함수 내 MLFQ 코드에서 *RUNNABLE*한 프로세스를 찾지 못했을 시 반복문을

continue 해주는 부분이다. 처음 코드를 짰을 때에는 ptable lock을 *release* 해주지 않고 단순히 *continue*만 불렀다. 그렇게 하니 *release*가 되지 않은 상태로 다시 *acquire*을 하는 문제가 발생하였고, 시스템이 부팅되지 않았다. 부팅조차 되지 않아 디버깅에 어려움을 겪었지만 ptable lock의 흐름을 전부 파악하고 난 후 해결을 할 수 있었다. 이후 scheduler lock 코드에서도 예외처리를 하는 과정에서 함수를 즉시 종료해야 하는 경우가 있는데 이때에도 ptable lock을 *release* 해 준 다음에 종료하도록 수정을 하였다.

2. Boosting시 생긴 문제

```
// For preserving processes order.
  quicksort(temp, 0, NPROC - 1);
  for(int i = 0; i < NPROC; i++)
    temp[i]->order = i + 1;
```

같은 queue 내부에서의 순서를 *order*이라는 프로세스 내부 변수를 통해 결정을 하도록 구현하였는데 이 때 boosting 이후에 순서를 보장해주기 어려운 문제가 발생하였다. 모든 프로세스가 L0로 들어가게 되는데 그렇게 되면 queue level의 차이로 순서가 정해졌던 프로세스들을 *order* 값으로 순서가 정해지도록 바꿔야 하기 때문이었다. 따라서 quicksort를 이용하여 queue, priority로 정해진 순서대로 프로세스를 일렬로 세운 후 순서에 맞는 *order* 값을 새로 부여하였다.

3. Scheduler lock 문제

```
// Scheduller Lock, if scheduler locking process exists and it is runnable.
    if(ptable.lockproc != 0 && ptable.lockproc->state == RUNNABLE) {
      np = ptable.lockproc;
    }
```

scheduler() 함수 내부에서 scheduler lock을 잡고 있는 프로세스가 존재하는지 확인한 후 그렇다면 그 프로세스가 스케줄링 되도록 하는 코드이다. 처음에는 프로세스의 상태가 *RUNNABLE* 인지 확인하지 않고 바로 스케줄링을 하였는데 이 때 scheduler lock을 잡은 프로세스가 그대로 *exit()* 하거나 *sleep()* 하면 *RUNNABLE* 하지 않은 프로세스를 스케줄링 하는 문제가 발생하였다. 이를 해결하기 위해 *RUNNABLE* 조건을 추가하였다.

```
// Reset the scheduler locking process.
      ptable.lockproc = 0;
```

또한 *exit()* 또는 *sleep()*을 하였다면 그 프로세스의 scheduler lock을 자동으로 해제하도록 구현하기 위하여 MLFQ에 들어가게 되면 바로 ptable의 *lock* 변수를 0으로

초기화해주도록 설계하였다. 이 코드를 통해 만약 scheduler lock을 잡은 상태로 *sleep()*을 하는 프로세스는 바로 scheduler lock을 잃게 되며 이후 *wakeup*을 한다고 다시 scheduler lock이 주어지지 않는다.