운영체제 Project #1

2018009125 조성우

1 MLFQ scheduler의 이해

MLFQ scheduler는 3개의 Queue로 구성이 되어있고, 각각의 level을 0,1,2,라 하였을 때, level이 낮은 Queue부터 scheduling을 하는 방식입니다. 기본적으로 level0,1, queue는 round-robin 방식으로 scheduling이 진행되고 level2는 priority scheduling을 진행합니다. Level0 queue에 process가 존재하지 않으면 level1 queue에 있는 process를 실행하고 level1마저 없다면 level2에 있는 process를 실행하게 됩니다. 각각의 process는 정해진 time quantum을 모두 소비하고 나면 현재 level+1의queue로 이동하게 됩니다. 만약 level2 queue에 있는 process가 time quantum을 모두 소비하고 나면 더 이상 level이 늘어나지 않고 자신의 priority가 1줄어들게 되고 time quantum은 0으로 초기화 됩니다. 이러한 방식으로 scheduling이 진행된다면 실행시간이 긴 process들은 결국 마지막 level2 queue에 쌓이게 되고 그들의 priority는 모두 0이 됩니다. 따라서 level2는 priority scheduling 방식이지만 마치 round-robin 방식처럼 돌아가게 됩니다. 또한 level0 queue에 지속적으로 새로운 process가 들어오게 된다면 level1, level2 queue에 들어가 있는 process들은 실행될 수 없는 상태에 놓이게 됩니다. 이 두가지 문제점을 해결하고자 priority boosting이 실행됩니다. 임의의 process가 실행될 때 global tick도 같이 늘어나게 설정을 합니다. 이 global tick이 100tick이 된다면 실행가능상태에 놓인 process들을 level0 queue에 초기화를 시킵니다. 이때 time quantum과 priority도 같이 초기화가 됩니다.

2 과제 설계

먼저, 크게 두가지의 방식을 생각했습니다. 첫번째는, 물리적인 queue을 3개 생성하여 각각을 level0, 1, 2로 설정하는 것입니다. 두번째는, 논리적인 queue였습니다. 물리적으로 3개의 queue를 만드는 것이 아니라, 각각의 process들에게 level을 붙이고 이 level을 통해 이 process가 어느 queue에 논리적으로 속해 있는지를 확인하는 것입니다. 초기에는 물리적인 queue를 3개 만드는 방식으로 방향을 잡았지만 queue생성과 queue를 관리하는 데에 쓰이는 자료구조를 만드는 것의 비용이 생각보다 높았습니다. 따라서 proc.h에 있는 struct구조체에 level이라는 변수를 설정하였고 proc.c에 있는 scheduler 함수내에서 이것을 관리하는 것이 효율적이라고 판단되어 결국 논리적인 queue를 만들기로 결정하였습니다. Struct proc 구조체에 level이라는 변수뿐만 아니라 process의 time quantum, priortiy에 관한 변수까지 설정해 놓았습니다. 이 변수들은 어떠한 process가 처음 실행될 때, 부모로부터 fork되었을 때 바로 초기화를 진행해주었습니다. 그리고 이렇게 설정된 변수들은 scheduler함수내에서 MLFQ scheduling에 사용됩니다.

3. 코드 작성

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int priority;                // Process priority
  int level;                   // Process Queue Level
  int time;                    // process timequantum
};
```

Proc.h내에 있는 proc의 구조체입니다. Int priority, int level, int time을 새로 포함시켰습니다.

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

static struct proc *initproc;

int nextpid = 1;
int global = 0;
extern void forkret(void);
extern void trapret(void);

static void wakeup1(void *chan);
```

Proc.c내에 있는 변수 선언 부분입니다. Int global이라는 새로운 변수를 여기서 설정하였는데 이 변수는 global tick을 나타냅니다. scheduler함수 내에서 임의의 process가 실행될 때마다 이 값이 1씩증가하게 됩니다.

```
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->level = 0;                         //set level, this process get into Level0 Queue
  p->time = 0;                          //set timequantum. initially this process get 0
  p->priority = 3;                      //set priority, initially this process get 3
  release(&ptable.lock);
```

Proc.c내에있는 allocproc함수입니다. 새로운 process가 실행되었을 때 그 process를 초기화하는 함수입니다. P->state를 통해 ptable내에서 UNUSED상태인 곳을 찾아내고 그곳에 process를 배정합니다. found: 에서 추가로 p->level =0, p->time = 0, p->priority=3를 넣었습니다. 이를 통해 새로 실행된 process를 level0 queue에 넣었고, time quantum은 0, priority는 3으로 설정하였습니다.

```
pid = np->pid;

acquire(&ptable.lock);
np->priority = 3;                       //set priority = 3
np->time = 0;                           //set time quantum = 0
np->level = 0;                          //set queue level = 0
np->state = RUNNABLE;

release(&ptable.lock);

return pid;
```

Proc.c내에 있는 fork함수입니다. 새로운 process가 실행될 때 부모 process의 정보를 복사하여 실행되는 경우입니다. 이 부분에도 저 3개의 정보를 초기화 했습니다.

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }
```

사실 fork의 윗부분에서 np=allocproc()에서 이미 np는 저 정보를 담고 있는 것으로 확인되지만 혹시 몰라 더블체크의 느낌으로 한 번 더 초기화를 진행했습니다.

```
scheduler(void)
{
  struct proc *p;
  struct proc *p1;                              //used in globaltime check and priortiy scheduler
  struct proc *highp;                           //used in priority scheduler
  struct cpu *c = mycpu();
  int high = 0;                                 //high is the lowest queue level, in our scheduler.
  int count[3] = {0};                           //count array is used to determine high. count[0] is the number of runnable process in level 0 queue and count[1], count[2] is same as.
  c->proc = 0;
```

```
void
scheduler(void)
{
  struct proc *p;
  struct proc *p1;
  struct proc *highp;
  struct cpu *c = mycpu();
  int high = 0;
  int count[3] = {0};
  c->proc = 0;
```

드디어 scheduler함수입니다. 변수선언 부분인데 여기서 새로이 proc *p1과 proc *highP, int high, int count[3]을 설정하였습니다. 우선 *p1은 p를 보조하는 역할로 주로 사용되며 ptable을 for 문 문에 돌릴 때 사용됩니다. *highP는 오직 level2 queue를 실행 할 때에만 사용되며 가장 낮은 priority를 가진 process를 가리키게 됩니다. High는 실행되어야 할 queue의 level을 나타내게 됩니다. 즉 만약 level0 queue에 process가 있다면 high = 0이 될 것이고 level0 queue에 process가 없고 level1 queue에만 있다면 high = 1 , level 0,1 queue에 process가 없거나 실행가능한 process가 모든 queue에서 존재하지 않는다면 high =2 를 가리키게 됩니다. High에 담긴 정보를 통해 scheduler는 high보다 높은 level을 가지고 있는 process는 실행하지 않게 됩니다. Count 배열은 high를 결정하기 위해 사용됩니다. 매 scheduling을 진행하기 전에 전체 ptable을 조사하게 되는데 이때 level0, 1,2에 속해 있는 process가 각각 몇 개인지가 이 count배열의 담기게 됩니다. 즉 level0, 1, 2 queue에 각각 5,4,1 개의 process가 담겨 있다면 count[0] = 5, count[1] = 4, count[2] = 1이 됩니다.

이 과제를 구현할 때 scheduler함수의 뼈대는 그대로 유지한 채 진행했습니다.

```
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;
  swtch(&(c->scheduler), p->context);
  switchkvm();
```

이 swtch함수를 통해 scheduler에서 실행할 process로 cpu가 넘어가게 됩니다. 그리고 그 process가 yield되어 sched가 실행되면 다시 이 부분으로 cpu가 넘어오게 되여 switchkvm()을 실행하고 그 밑의 코드를 실행하게 됩니다. 그 밑의 코드에는

```
    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0;
    count[0] = 0;                                   //reset queue's information that has the number of process in each queue.
    count[1] = 0;
    count[2] = 0;
```

```
    for ( p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++ )        //check which levels' queue is running before scheduling.
    {
        if ( p1->state == RUNNABLE )
        {
            if ( p1->level == 0 )
              count[0]++;
            else if ( p1->level == 1 )
              count[1]++;
            else
              count[2]++;
        }
    }
    if(count[0] > 0) high = 0;                                      //first check if level0 exists.
    else if(count[1] > 0 ) high = 1;                                //no level0 process,  check if level1 exists.
    else high = 2;                                                  //no level0 and level1 its high = 2. this state means scheduler has level2 process or nothing.
}
```

가 있습니다. (두 사진 사이에 코드가 하나 더 있습니다.)우선 c->proc = 0으로 설정하는 것으로 cpu가 process를 실행하고 있지 않음을 나타냅니다. 그리고 count배열을 모두 0으로 초기화를 진행합니다. 그 이유는 바로 밑에서 runnable한 모든 process의 level을 조사하기 때문입니다. 밑의 이미지를 보면 for문은 원래 scheduler가 하는 것과 비슷합니다. 다른 점이 있다면 이 for문은 swtch를 통해 cpu가 다시 scheduler로 넘어올 때마다 실행되기 때문에 매번 ptable의 처음부터 끝 까지를 조사하게 된다는 점입니다 ptable에 있는 process가 RUNAABLE하다면 그 process의 level을 조사해 해당 count[level]을 +1을 하게 됩니다. for문이 종료되고 나면 count[0], 즉 level0인 process들부터 조사하게 됩니다. 만약 이 값이 0보다 크다면 level0인 proecss가 존재한다는 뜻이므로 high를 0으로 설정하고 다음 실행할 process를 찾는 for으로 돌아가게 됩니다. 이 for문은 밑의 이미지에서 for(;;)가 아니라(앞으로 이 for문을 무한for문이라 하겠습니다.) 두번째 for(p=ptable,,,,)(앞으로 이 for문을 scheduler for문이라고 하겠습니다.)를 뜻합니다.

```
for(;;){
  // Enable interrupts on this processor.
  sti();

  // Loop over process table looking for process to ru
  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE || p->level > high)
      continue;
    if ( high < 2 )
    {
        p->time++;
        global++;
        if ( p->time == (2 * p->level + 4) )
        {
            p->level++;
            p->time = 0;
        }
    }
```

다시 high로 돌아와서 count[0] =0이고 count[1]>0 이라면 level0은 없고 level1은 존재한다는 것이므로 high =1로 설정합니다. 둘다 해당되지 않는다면, 즉 level0, 1 둘 다 0이라면 high를 2로 설정합니다. 이 상태는 실행가능한 process가 없는 경우에도 해당됩니다.

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE || p->level > high)       //check RUNNALBE and p->level < high, p->level <high means this process is in the lower level queue than others, so this process can run
    continue;
  if ( high < 2 )                                   //this is level 1 or 0 queue, so we run process by round-robin
  {
      p->time++;                                    //process's timequantum +1
      global++;                                     //global ticks + 1
      if ( p->time == (2 * p->level + 4) )          //if allocated timequantum is done, process's queue level +1 and timequantum = 0
      {
          p->level++;
          p->time = 0;
      }
  }
```

방금 그 scheduler for문입니다. 우선 RUNNABLE하지 않거나 level이 설정된 high보다 높은 process들은 컷합니다. 만약 설정된 high보다 작거나 같은 level을 가지고 있고 RUNNABLE하다면 continue밑에 있는 if문이 실행됩니다. 여기서 미리 설정된 high를 통해 level0,1 queue를 실행할 것인지 level2queue를 실행할 것인지 결정합니다. 여기선 high가 0이었다고 가정하겠습니다. 그러면 이 process의 time quantum이 time변수를 통해 1증가하게 되고 globaltick도 1증가합니다. 그리고 만약 이 time이 그 level queue에 미리 설정되어 있던 time quantum에 다다르면 이 process의 level 즉 queue level 1올리고 time을 0으로 reset합니다. 그리고 이 process는 swtch를 통해 실행되게 됩니다. 그리고 다시 scheduler로 돌아왔을 때 scheduler for문은 이 process 다음 process부터 조사하게 되고 해당 scheduler for문에서 다시는 저 process가 실행되지 않습니다 scheduler for문이 끝나고 무한for문에 의해 다시 새로운 scheduler for문이 시작되는데 이때 ptable의 처음부터 조사하게 됩니다. 즉 해당 process의 뒤와 앞을 모두 조사하고 나서야 이 process에게 다음 차례가 오게 됩니다. 따라서 process는 자연스럽게 해당 level queue의 가장 마지막으로 들어가게 된 것처럼 됩니다. 다음 process이 과정은 level0,1 queue를 실행할 때 진행됩니다.

```
else                                                //this's level 2 queue, so we run process by priority
{
    highp = p;                                      //highP = p =, initiallization
    for ( p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++ )      //check whose priority level is lower than highP
    {
        if ( p1->state != RUNNABLE )
            continue;
        if ( highp->priority > p1->priority )                     //Find lower priority then, change highP
            highp = p1;
    }
    p = highp;                                                    //determine which process is running. and its timequantum +1 and globalticks+1
    p->time++;
    global++;
    if ( p->time == 8 )                                           //if this process's timequantum is over, set timequantum = 0 and its priorty -1
    {
        p->time = 0;
        p->priority--;
        if ( p->priority < 0 )                                    //if its priority is -1
            p->priority = 0;
    }
}
```

다음은 if(high<2)에 대응하는 else문입니다. 즉 이 else문은 level2 queue를 담당합니다. 그리고 이 queue는 priority queue입니다. 우선 highP를 p로 설정합니다 이 p는 scheduler for문을 통해 이미 level 2를 가지고 있는 process입니다. 그러고는 scheduler for문과 비슷한 for문을 한번 실행합니다 이를 priority for문이라고 하겠습니다. 이 priority for문에서는 ptable의 처음부터 끝까지 p보다 같지도 않고, 무조건 낮은 priority를 가진 level인 2인 process를 찾습니다. 만약 그런 process가 있다면 highP를 그 process로 바꿔줍니다. 그러고는 level0,1 queue와 마찬가지로 time을 늘려주고 globaltick도 늘려줍니다. Time이 level2 queue의 time quantum과 같은지 확인하고 같으면 time을 0으로 리셋하고 priority를 낮춥니다. 이때 -1이 된다면 다시 0으로 복구해줍니다. Scheduler for문에서는 같은 priortiy라도 queue앞쪽에 있는 process를 선별해주고 이 priority for문에서 이 process보다 낮은 priority를 가진 process를 검색하기 때문에 같은 priority를 가진

process여도 queue앞쪽의 process를 먼저 선택해줍니다.

```
    if(global==100){                                          //if globaltick = 100, all process's information is reset.
      for( p1 = ptable.proc; p1< &ptable.proc[NPROC]; p1++){
       if(p1->state != RUNNABLE ) continue;
        p1->level = 0;
        p1->priority = 3;
        p1->time = 0;
      }

      global = 0;
    }
```

아까 그 두 사진 사이에 껴 있던 코드입니다. (swtch밑의 코드입니다.) 만약 globaltick이 100이 되었다면 ptable안에 들어있는 모든 RUNNABLE한 process를 level0으로 옮기고 prioritiy, time도 reset합니다. 그리고 다시 globaltick을 0으로 초기화합니다.

다음은 system call 구현입니다.

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

우선 setPriorty입니다. Pid와 priority를 인자로 받고, for문을 통해 해당 pid를 가지고있는 process를 찾아냅니다. 그리고 그 process의 priority를 바꿔줍니다. 이때 ptable안에있는 prcess에 접근하는 것이기 때문에 acquire lock을 해주어야 합니다.

```
int
getLevel(void)
{
    struct proc *curproc = myproc();
    int lev;
    lev = curproc->level;
    return lev;
}
```

다음은 getLevel입니다. 현재 실행되고 있는 process의 정보를 myproc()을 통해 받은 다음. lev이라는 변수에다가 그 process의 level을 받습니다. 그리고 이를 return합니다.

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

다음은 yield입니다. 이는 원래 proc.c에 있던 그 yield와 동일합니다.

```
int
sys_setPriority(void)
{
  int pid, pr;
  if(argint(0, &pid) < 0)
          return -1;
  if(argint(0, &pr) < 0)
          return -1;

  setPriority(pid, pr);
  return 0;
}

int
sys_getLevel(void)
{
 return getLevel();
}

int
sys_yield(void)
{
  yield();
  return 0;
}
```

그 다음 sysproc.c에 이렇게 wrapper function을 만들어 주었고, defs.h syscall.c,h , user.h, usys.S에도 syscall에 필요한 작업들을 했습니다.

4. 실행결과

다음은 piazza resource를 통해 다운받은 mlfq_test.c를 통한 test 결과입니다.

```
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 5
L0: 9960
L1: 19147
L2: 70893
L3: 0
L4: 0
Process 4
L0: 10489
L1: 29498
L2: 60013
L3: 0
L4: 0
Process 6
L0: 16288
L1: 28579
L2: 55133
L3: 0
L4: 0
Process 7
L0: 14927
L1: 26844
L2: 58229
L3: 0
L4: 0
[Test 1] finished
done
$
```

동작과정은 3. 코드 작성에서 설명했으므로 스킵하도록 하겠습니다.

5. 문제 발생 및 해결 과정

처음 작성하였던 코드는 ptable안에 count배열과 queue를 넣어서 짰었습니다. ptable을 acquire lock하고 release lock 하는 과정이 많아졌고 이것을 반복하다 보니 acquire오류가 발생하여 코드 전체를 갈아엎어야 했었습니다. 이에 ptable을 건드는 것은 무리라는 판단을 하였고 최대한 xv6의 기본 동작방식을 따라가되 필요한 부분만을 바꿔보자는 생각을 가지게 되어 기본 뼈대는 유지한 채 scheduler함수의 내용을 바꾸게 되었습니다. 각 process의 tick을 늘릴 때에도 어차피 모든 process는 한번에 끝나지 않으면 scheduler함수를 거쳐가기 때문에 그곳에서 tick을 관리하자는 생각도 하게 되었습니다. 그리고 코드를 보시면 아시겠지만. scheduler lock과 unlock을 구현하지 못했습니다. 최대한 ptable lock을 잡지 않고 이 두 함수를 구현하고자 하여 scheduler 쪽에서 코드를 짜보았지만 적절한 해결책이 나오지 않게 되었습니다.

이상 마치겠습니다. 감사합니다.