OS project-2 Report

CFS Scheduling

Department of Software

2016312568

Jung Hee Yoon

CFS(Completely Fair Scheduler)은 Run queue에서 Runnable인 상태인 프로세스를 공정하게 실행되도록 가중치를 두는 scheduler이다. 이를 위하여 CFS는 time slice, priority, vruntime라는 3가지 속성을 가진다. time slice란, scheduler가 프로세스에게 부여할 실행 시간을 의미한다. 프로세스는 time slice만큼 실행되고 context switching이 되어 다음 프로세스에게 cpu를 넘겨주게 된다. 여기서 time slice의 크기는 priority와 total weight (runnable인 프로세스들의 weight 합)에 의존한다. weight는 말 그대로 가중치이며, 이는 priority에 의해 계산되어진다. vruntime (virtual runtime) 은 "프로세스가 그 동안 실행한 시간을 정규화한 시간 정보"이며 scheduler가 어떤 프로세스를 먼저 실행할지 정하는 지표이다. CFS는 맨 처음 시작할 때, 각 프로세스가 할당된 time slice를 다 소진하였을 때마다 vruntime이 최소인 process를 찾아 cpu에 할당한다.

xv6에서 이러한 CFS를 구현하기 위해서는 다음과 같은 과정이 필요하다.

1. process에 vruntime, priority, time slice 속성을 추가한다.
2. weight 계산을 없애기 위하여 각 priority에 해당하는 weight를 하드코딩한다.
3. scheduling을 실행하는 scheduler()이라는 함수에서 매 scheduling 마다 RUNNABLE상태인 프로세스들의 weight 합 (total weight)을 구하고 최소 vruntime인 process를 찾는다. 찾은 프로세스를 cpu에 배정한다.
4. 배정된 프로세스가 time slice만큼 돌아가게 한다. (process가 running일때 Timer interrupt가 발생하면 time slice = time slice -1을 해주어 남은 실행시간을 갱신시켜준다.)
5. sleep인 프로세스가 wake up했을때 우선 실행이 되도록 최소 vruntime을 배정한다.
6. vruntime의 오버플로우를 생각해준다.

위의 과정을 구현한 것을 각각 자세히 설명하자면 다음과 같다.

1. process에 vruntime, priority, time slice 속성을 추가한다.

```
// 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;
  int maked;
  int execTime;
  int vruntime;
  int timeslice;
  int over;
};

// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap

static const int weightInfo[40] = {
    88761, 71755, 56483, 46273, 36291,
    29154, 23254, 18705, 14949, 11916,
    9548, 7620, 6100, 4904, 3906,
    3121, 2501, 1991, 1586, 1277,
    1024, 820, 655, 526, 423,
    335, 272, 215, 172, 137,
    110, 87, 70, 56, 45,
    36, 29, 23, 18, 15
};
static const int INT_MAX = 2147483647;
```

프로세스 정보가 담겨 있는 proc.h에 위의 사진처럼 수정하여 준다.

process의 정보를 담는 proc구조체에 priority, execTime (Run time), vruntime, timeslice, over라는 변수를 추가하여 준다. over는 vruntime의 오버플로우를 handling하기 위하여 추가하여준 변수로 오버플로우의 설명에서 자세히 다루도록 하겠다.

2. weight 계산을 없애기 위하여 각 priority에 해당하는 weight를 하드코딩한다.

위의 사진처럼 각 priority에 해당하는 weightInfo배열을 하드코딩하여주고, 최소 vruntime 구하기와 오버플로우를 handling하기 위한 INT_MAX값도 추가하여 준다.

3. scheduling을 실행하는 scheduler()이라는 함수에서 매 scheduling 마다 RUNNABLE상태인 프로세스들의 weight 합 (total weight)을 구하고 최소 vruntime인 process를 찾는다.

```
void
scheduler(void)
{
  int totalWeight=0;
  struct proc *p;
  struct proc *choosen;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      choosen=p;
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->state==RUNNABLE) {
            totalWeight+=weightInfo[p->priority];
            if(p->over == choosen->over)
            {
                if((p->vruntime) < (choosen->vruntime)) choosen=p;
            }
            else
            {
                if(p->over < choosen->over) choosen=p;
            }
            //if(p->vruntime < choosen->vruntime) choosen=p;
        }
      }

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.

      p=choosen;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      p->timeslice = 10 * weightInfo[p->priority]/totalWeight;
      p->execTime += 1000;

      if(p->vruntime > INT_MAX - 1024000/weightInfo[p->priority])
      {
          p->over++;
          p->vruntime=(p->vruntime-INT_MAX)+1024000/weightInfo[p->priority];
      }
      else p->vruntime += 1024000/weightInfo[p->priority];
      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
      totalWeight=0;
    }
    release(&ptable.lock);

  }
}
```

scheduling 함수는 proc.c에 기본적으로 정의되어 있으며 이를 CFS로 수정한 scheduler함수는 왼쪽 사진과 같다.

choosen은 scheduler에서 cpu에 할당할 프로세스이다.

state 가 RUNNABLE이고 vruntime이 최소인 것을 choosen으로 한다.

이렇게 choosen을 찾고, timeslice를 공식에 맞추어 계산하여 넣어준다.

여기서 execTime, vruntime에 한틱만큼 더하여 주는데 이 이유는 choosen process가 1틱 안에 끝나는 경우 runtime과 vruntime은 0이 되어버리기에, 시작할때 한 tick을 더해주고 시작한다. 이는 1보다 작은 tick도 1틱으로 치기에 문제가 없다.

4. 배정된 프로세스가 time slice만큼 돌아가게 한다. (process가 running일때 Timer interrupt가 발생하면 time slice = time slice -1을 해주어 남은 실행시간을 갱신시켜준다.)

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER) {
  myproc()->timeslice--;
 // cprintf("%s %d %d \n",myproc()->name,myproc()->pid,myproc()->timeslice);
  myproc()->execTime+=1000;
  if(myproc()->vruntime > INT_MAX - 1024000/weightInfo[myproc()->priority])
   {
      myproc()->over++;
      myproc()->vruntime=(myproc()->vruntime-INT_MAX)+1024000/weightInfo[myproc()->priority];
   }
   else myproc()->vruntime += 1024000/weightInfo[myproc()->priority];

  //myproc()->vruntime+=1024000/weightInfo[myproc()->priority];
  if(myproc()->timeslice<=0) {
     // cprintf("yield!!!!! cpu : %d\n",mycpu());
      yield();
  }
}
```

위 사진은 interrupt가 발생했을 때 처리해주는 trap.c 에서 trap함수의 아랫부분이다.

trap함수는 switch문으로 인터럽트들을 처리해준다. 여기서 default에 process가 존재하며, process가 running인 상태이고 타이머 인터럽트가 발생했을경우를 처리해준다. 남은 시간(time slice)를 1빼주고, vruntime, runtime을 한틱만큼 더해준다. time slice를 모두 소진하였을 경우 yield() 시켜준다. yield()는 현재 실행중인 프로세스를 멈추고 state를 RUNNING에서 RUNNABLE로 바꾸어 주는 함수이다.

5. sleep인 프로세스가 wake up했을때 우선 실행이 되도록 최소 vruntime을 배정한다.

```
static void
wakeup1(void *chan)
{
  struct proc *p;
  int smallest = INT_MAX;
  int smallest_over = INT_MAX;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
      if(p->state == RUNNABLE || p->state == RUNNING)
      {
          if(p->over < smallest_over){
              smallest_over=p->over;
              smallest=p->vruntime;
          }
          else if (p->over == smallest_over && p->vruntime < smallest) smallest=p->vruntime;
      }
  }
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
      if(p->state == SLEEPING && p->chan == chan)
      {
          p->state = RUNNABLE;
          if(smallest_over==INT_MAX) {
              p->vruntime = 0;
              p->over=0;
          }
          else {
              if(smallest-1024000/weightInfo[p->priority]<0)
              {
                  if(smallest_over>0) {
                      smallest_over--;
                      p->vruntime=(smallest-1024000/weightInfo[p->priority])+INT_MAX;
                  }
                  else p->vruntime=0;
              }
              else p->vruntime = smallest-1024000/weightInfo[p->priority];
              p->over = smallest_over;
          }
      }
  }
}
```

위의 사진은 sleep 상태인 process중 특정한 process들을 runnable상태로 만들어주는 함수이다. sleep에서 runnable상태로 바뀐 프로세스들은 우선적으로 실행되어야 하기에 runnable, running인 프로세스들 중 최소 vruntime을 가진 프로세스의 vruntime - 1tick (가중치 생각한 값)을 해준다. 여기서 wake up 한 프로세스들이 여럿일 경우 이들 중에 priority 값이 낮은 것 이 먼저 실행되어야 하지만 과제 조건에서 vruntime 은 음수일경우 0으로 처리하라고 명시되어 있기에 0으로 처리하였다.

추가적으로 fork()가 이루어질때 자식은 부모의 priority와 vruntime을 상속받을 수 있게끔 하였다.

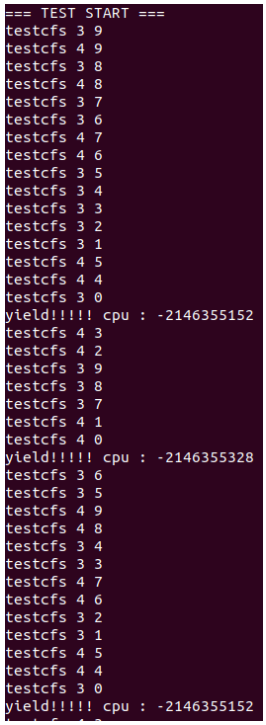
6. vruntime의 오버플로우를 생각해준다.

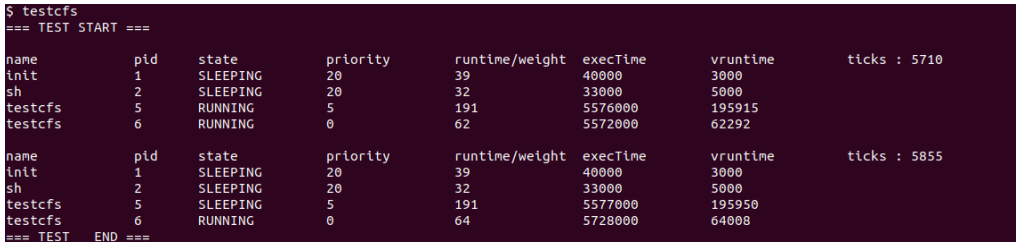
위의 사진들에서 vruntime을 계산하여 줄때 INT_MAX의 값을 넘어가면 over이라는 변수에 +1을 하여준다. 여기서 over은 INT_MAX unit이라고 생각하면 된다.
vruntime+1 tick (가중치를 생각한) > INT_MAX로 할 경우에 부등식의 좌변에서 오버플로우가 날 수 있으므로 vruntime > INT_MAX - 1tick (가중치를 생각한)으로 변환하여 overflow를 handling해준다. 이렇게 하면 총 64비트까지 나타날 수 있으며, 이러한 유닛을 여러개 만든다면 96, 128 등 더 표현 가능하며 유닛을 나타내는 변수를 포인터로 만들고 때에 따라 malloc, realloc등을 시켜주어 무제한으로 표현 가능하다. long long int, unsigned int등의 자료형 출력이 없으며, 이러한 자료형을 출력하기 위해서 형변환을 해야하는데 여기서 데이터손실이 일어날 수 있으므로 이러한 방식으로 오버플로우를 handling하였다. 이때, 시각적으로 편하게 하기위하여 vruntime이 INT_MAX보다 클경우에는 vruntime+INT_MAX*over 처럼 출력하게 해놓았다. vruntime이 INT_MAX보다 작을경우는 vruntime만 출력되도록 해놓았다.

TEST CASE & explain about each case

NCPU는 default로 8로 되어있다. 위와같은 방법으로 CFS를 구현할 경우 testcfs의 결과를 확인하기 어렵다. 따라서 두가지 경우를 출력하고 따로 설명하겠다.

1. NCPU > 1 인 경우의 testcfs.c 출력값



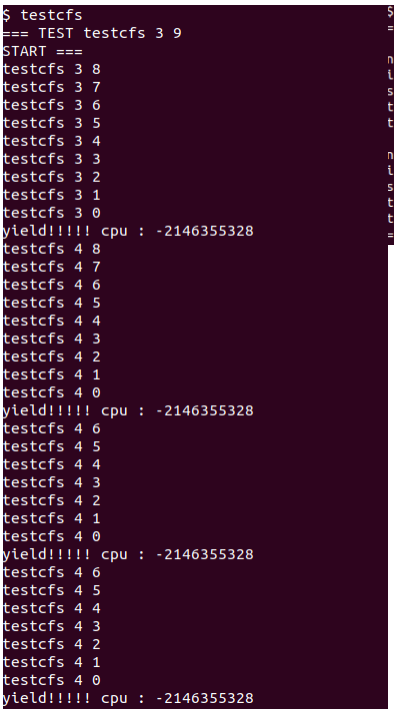


위의 사진에서 좌측의 사진은 trap.c에서 timeslice-1이되는 부분에서 process name, process pid, process timeslice을 출력하고 yield()된 경우에서 이 프로세스를 yield시킨 cpu의 주소를 나타낸 정보이다. 우측의 사진은 이와 같은 정보를 출력하지않고 testcfs의 결과값을 찍은 사진이다.

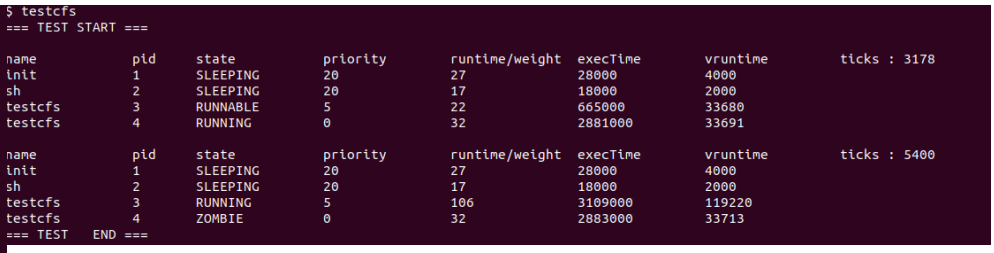
먼저, 좌측의 사진에서 pid=3, pid=4인 프로세스가 동시에 실행됨을 볼 수 있으며, 이는 cpu가 8개이고, yield() 시킨 cpu의 주소값이 다른것으로 보아 각 cpu에 프로세스가 하나씩 할당된 것임을 알 수 있다.

그렇기에 우측의 사진에서 execTime (runtime)이 부모와 자식이 비슷하고 vruntime이 많이 다르다.

2. NCPU = 1 인 경우의 testcfs.c 출력값



```
$ testcfs
=== TEST testcfs 3 9
START ===
testcfs 3 8
testcfs 3 7
testcfs 3 6
testcfs 3 5
testcfs 3 4
testcfs 3 3
testcfs 3 2
testcfs 3 1
testcfs 3 0
yield!!!!! cpu : -2146355328
testcfs 4 8
testcfs 4 7
testcfs 4 6
testcfs 4 5
testcfs 4 4
testcfs 4 3
testcfs 4 2
testcfs 4 1
testcfs 4 0
yield!!!!! cpu : -2146355328
testcfs 4 6
testcfs 4 5
testcfs 4 4
testcfs 4 3
testcfs 4 2
testcfs 4 1
testcfs 4 0
yield!!!!! cpu : -2146355328
testcfs 4 6
testcfs 4 5
testcfs 4 4
testcfs 4 3
testcfs 4 2
testcfs 4 1
testcfs 4 0
yield!!!!! cpu : -2146355328
```

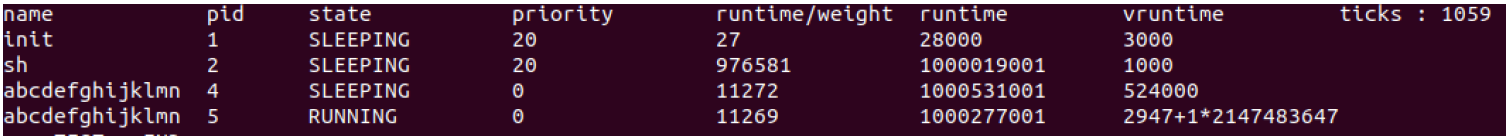


```
$ testcfs
=== TEST START ===

name            pid     state           priority        runtime/weight  execTime        vruntime        ticks : 3178
init            1       SLEEPING        20              27              28000           4000
sh              2       SLEEPING        20              17              18000           2000
testcfs         3       RUNNABLE        5               22              665000          33680
testcfs         4       RUNNING         0               32              2881000         33691

name            pid     state           priority        runtime/weight  execTime        vruntime        ticks : 5400
init            1       SLEEPING        20              27              28000           4000
sh              2       SLEEPING        20              17              18000           2000
testcfs         3       RUNNING         5               106             3109000         119220
testcfs         4       ZOMBIE          0               32              2883000         33713
=== TEST   END ===
```

위의 사진에서 좌, 우측의 사진은 1번의 경우와 같은 조건이다.

좌측의 사진을 보면 cpu가 한개이며 pid=3인것이 먼저 실행되고, fork()된 후에 pid=3인 process의 priority를 5, pid=4인 process의 priority를 0으로 바꾸어주었기에 pid=3이 먼저 실행된 후 priority가 0인 pid=4인 것이 계속하여 할당받음을 알 수 있다. 이것과 yield() 시켜준 cpu의 주소가 모두 같은것으로 보아 CFS가 옳게 작동함을 알 수 있다. 우측 사진에서 보면 pid=3, 4인 프로세스의 vruntime은 거의 같고 execTime은 priority에 의해 많이 다르기에 CFS가 옳게 작동했음을 알 수 있다.

3. ps에서 각 항목의 이름 또는 값이 길 경우 align처리와 overflow시 handling



```
name            pid     state           priority        runtime/weight  runtime         vruntime        ticks : 1059
init            1       SLEEPING        20              27              28000           3000
sh              2       SLEEPING        20              976581          1000019001      1000
abcdefghijklmn  4       SLEEPING        0               11272           1000531001      524000
abcdefghijklmn  5       RUNNING         0               11269           1000277001      2947+1*2147483647
```