Operation System
Project #2: Implementing CFS Scheduler on xv6

2014311577 Global Economics
Dongmin Kim

1. Basic Data Structure
   In proc.h, struct proc is defined as follows.

```
// 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)
  // for cfs scheduling(proj 2)
  int priority, weight;
  uint vruntime[2];            // vruntime
  uint runtime, druntime;      //runtime, delta runtime
  int time_slice;
};
```

- Variables priority, weight are defined for checking priority and weight, and time_slice, according to the priority.
- Variables related to runtime(vruntime[2], runtime, druntime) are defined for capturing various aspects of runtime.
- runtime corresponds to actual runtime measured in mili-tick unit.
- druntime corresponds to delta runtime measured in mili-tick unit. It is updated to 0 whenever time_slice expires.
- vruntime corresponds to virtual runtime measured in mili-tick unit. According to CFS scheduling, Time goes slower or faster based on priority. Array size of 2 is used to handle integer overflow, which will be discussed later.

2. Pre-defined helper variables

   In proc.c,
   two variables are defined outside individual functions.

```
static int weights[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 char *states[] =
{
  [UNUSED]    "UNUSED  ",
  [EMBRYO]    "EMBRYO  ",
  [SLEEPING]  "SLEEPING",
  [RUNNABLE]  "RUNNABLE",
  [RUNNING]   "RUNNING ",
  [ZOMBIE]    "ZOMBIE  "
};
```

3. Integer Overflow Handling
    1) Assumption

    runtime and total tick is assumed to be in 4 byte unsigned integer range(0 to 4,294,967,295). However, vruntime may overflow due to its property: time may go slower, or faster. vruntime is calculated as:

    $$vruntime \mathrel{+}= \Delta runtime * 1024 / weight\ of\ current\ process$$

    According to formula above, if current process's priority is low, weight of current process could be decreased to 15(when priority is 39), which makes the range to cover for variable vruntime be

    $$0 < vruntime < 293,203,100,672(4,294,967,295 * 1024 / 15) < 2^{39}$$

    In other words, we need at least 39 bits to cover up.

    2) Strategy

    To handle integer overflow, following strategy has been utilized.

    - unsigned int array of size 2 has been utilized.
    - 1st index of array is lower part of decimal digits, and 2nd index of array is upper part of decimal digits.
    - If lower part reaches 1,000,000,000 or more, overflow occurs, which makes carry to upper part of digit.

    For example, number in range 0~999,999,999 can be:

0	0	0	0	0	0	0	0	0	0	9	9	9	9	9	9	9	9	9	9
<				vuntime[1]					><				vruntime[0]						>

    Another example, number over that range can be

0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	2	3	4
<				vuntime[1]					><				vruntime[0]						>

3) Implementation
In proc.c, utilities for vruntime is implemented as follows:

```
/*
vruntime utility.
*/
// uint covers up to 4,294,967,295
// newly made overflow condition is 1,000,000,000
// (can express up to 999,999,999)

#define VINT_MAX 1000000000
void vadd(uint vr[], uint x){
  vr[0] += x;
  if(vr[0] >= VINT_MAX){
    vr[1]++;
    vr[0] = vr[0]- VINT_MAX;
  }
}

void vsub(uint vr[], uint x){
  if(vr[0] > x){
    vr[0] -= x;
  } else {
    vr[1]--; // borrow from upper digit.
    vr[0] = VINT_MAX + vr[0] - x;
  }
}
```

vadd(uint[], uint x) and vsub(uint[], uint x) are defined here.
User made overflow condition is given as VINT_MAX. For addition, if the lower part of vruntime overweighs VINT_MAX, upper part is increased by 1, whereas lower part is decreased by VINT_MAX.
Similarly, subtraction borrows from upper digit, if lower digit is not enough to handle integer.

```
int vcompare(uint vra[], uint vrb[]){
  /*
    value is same, return 0
    a is bigger, return 1
    b is bigger, return 2
  */
  // compare upper digit first
  if(vra[1] > vrb[1])
    return 1;
  else if(vra[1] < vrb[1])
    return 2;
  else // if no overflow occured
  {
    if(vra[0] > vrb[0])
      return 1;
    else if(vra[0] < vrb[0])
      return 2;
    else
      return 0;
  }
}
```

For comparison of two different vruntime, another function vcompare(uint a[], uint b[]) is defined. This function compares upper part first, and then lower part. if a is bigger, this returns 1, b is bigger, returns 2, and if same, returns 0.

```
void printvr(uint vr[])
{
  if(vr[1])
    cprintf("%d", vr[1]);

  cprintf("%d", vr[0]);
}
```

Helper function printvr(uint vr[]) print out in decimal form.
if upper part is 0, it will only print out lower part.
[0][123456789] => 123456789
else, upper part is printed first, and then sequentially lower part.
[147][123456789] => 147123456789

4. Initialization: From UNUSED to EMBRYO state.

Initialization of process is done in function allocproc() proc.c.

```
static struct proc*
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->priority = 20;
  p->weight   = weights[p->priority];

  release(&ptable.lock);

 ...
```

In here, weight is set according to priority.

Additionally, for forked process, it inherits parent's priority, weight, and vruntime as follows.

```
int
fork(void)
{
  (...)
  // inherits parent process's priority, weight, vruntime
  np->priority = curproc->priority;
  np->weight   = curproc->weight;
  np->vruntime[0] = curproc->vruntime[0];
  np->vruntime[1] = curproc->vruntime[1];

  // set to RUNNABLE.
  np->state = RUNNABLE;
(...)
```

5. Scheduling: select process with minimum vruntime
In proc.c, scheduler is defined as follows:

```
void
scheduler(void)
{
  struct proc *p, *p_minvrun, *_p1, *_p2;
  struct cpu *c = mycpu();
  int sum_weights;
  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;

      p_minvrun = p;
      for(_p1 = ptable.proc; _p1 < &ptable.proc[NPROC]; _p1++){
        if(_p1->state != RUNNABLE)
          continue;

        // Select minimum vruntime RUNNABLE process
        if(vcompare(_p1->vruntime, p_minvrun->vruntime) == 2)
          p_minvrun = _p1;

      }

      p = p_minvrun;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p); // HW setups.

      // Sum of weights: Helper loop to get sum of weights
      sum_weights = 0;
      for(_p2 = ptable.proc; _p2 < &ptable.proc[NPROC]; _p2++){
        if(_p2->state != RUNNABLE && _p2->state != RUNNING)
          continue;
```

```
        sum_weights += _p2->weight;
      }

      p->time_slice = (10000 * p->weight)/sum_weights;
      p->state      = RUNNING;

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

In bigger loop, ptable is reference and being iterated.

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

If there is no RUNNABLE process, loop ends. Else, it search for minimum vruntime process.

```
p_minvrun = p;
for(_p1 = ptable.proc; _p1 < &ptable.proc[NPROC]; _p1++){
   if(_p1->state != RUNNABLE)
      continue;

    // Select minimum vruntime RUNNABLE process
   if(vcompare(_p1->vruntime, p_minvrun->vruntime) == 2)
      p_minvrun = _p1;
   }
p = p_minvrun;
// Switch to chosen process.  It is the process's job
// to release ptable.lock and then reacquire it
// before jumping back to us.
c->proc = p;
switchuvm(p); // HW setups.
```

6. Process Yield

Timer interrupt is on every tick. This is defined at trap.c, so It is possible to modify trap.c to implement yield for the time slice overload.

Firstly, in trap.c, timer is on and tick is calculated. Meanwhile, runtime and vruntime of current process is updated.

```
void
trap(struct trapframe *tf)
{
  (...)
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;

      struct proc *curproc = myproc();
      int vtick;

      // record elapsed runtimes: runtime, druntime, vruntime
      if(curproc){
        if(curproc->state == RUNNING) {
          vtick              = (1024000)/curproc->weight;
          curproc->runtime  += 1000;
          curproc->druntime += 1000;

          vadd(curproc->vruntime, vtick);
        }
      }
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
   (...)
```

tick is updated by 1, and along that, If the process is in RUNNING state, runtime and druntime is updated by unit of militick. for vruntime, it is updated according to the formula

$$vruntime \mathrel{+}= \Delta runtime * 1024 / weight\ of\ current\ process$$

in militick unit.

Secondly, yield() is called according to the comparison between druntime and time_slice.

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER) {
    // If task runs more than time slice, enforce a yield of CPU.
    if(myproc()->druntime > myproc()->time_slice){
      myproc()->druntime = 0; // reset delta runtime.
      yield();
    }
  }
```

Here, if druntime(delta runtime) is higher than time slice, or if task run more than time slice, it enforces a yield of the CPU, and also resetting druntime to 0.

7. Wakeup

During the wake-up procedures of process, the process need to be scheduled first, so the vruntime is set to be less than minimum vruntime.

Implementation code is as follows.

```
static void
wakeup1(void *chan)
{
  (...)
  // get minimum vruntime here
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
      continue;

    flag         = 1; // some RUNNABLE exists.
    p_minvrun    = p;
    min_vruntime[1] = p_minvrun->vruntime[1];
    min_vruntime[0] = p_minvrun->vruntime[0];

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

      if(vcompare(_p->vruntime, p_minvrun->vruntime) ==2 ){
        p_minvrun = _p;
        min_vruntime[1] = p_minvrun->vruntime[1];
        min_vruntime[0] = p_minvrun->vruntime[0];
      }
    }
  }
  (...)
```

Firstly, minimum vruntime is being searched. Values of minimum vruntime is stored in min_vruntime, and reference is stored in p_minvrun.

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state == SLEEPING && p->chan == chan){
    p->state = RUNNABLE;
    wtick    = ((1000 * 1024)/p->weight);
    if(flag){
      uint wtick_vector[2] = {wtick, 0};
      int underflow = 0;
      underflow = vcompare(min_vruntime, wtick_vector);
      vsub(min_vruntime, wtick);
      p->vruntime[0] = min_vruntime[0];
      p->vruntime[1] = min_vruntime[1];

      if(underflow == 2){
        p->vruntime[0] = 0;
        p->vruntime[1] = 0;
      }
    }
  }
}
```

And then, by iterating through the ptable, all sleeping process is being woken up, with vruntime of minvruntime less than 10tick * 1024/p->weight.

8. Test results

Test result using testcfs.c:

```
init: starting sh
Student ID: 2014311577
Name: Dongmin Kim
Message from developer: Welcome to xv6 OS!
$ testcfs
=== TEST START ===
name            pid   state     priority        runtime/weight  runtime             vruntime            tick 4614
init            1     SLEEPING  20              1               2000                2000
sh              2     SLEEPING  20              0               1000                3000
testcfs         3     RUNNABLE  5               29              848000              43295
testcfs         4     RUNNING   0               40              3575000             43325
name            pid   state     priority        runtime/weight  runtime             vruntime            tick 7161
init            1     SLEEPING  20              1               2000                2000
sh              2     SLEEPING  20              0               1000                3000
testcfs         3     RUNNING   5               116             3394000             132405
testcfs         4     ZOMBIE    0               40              3576000             43336
=== TEST   END ===
```