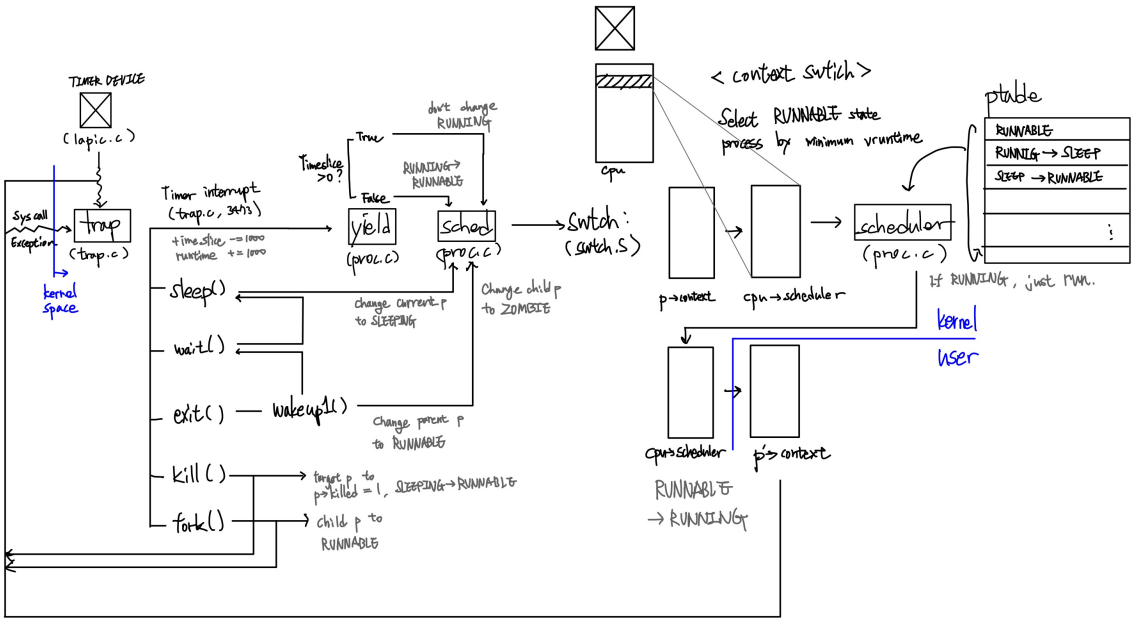
Pa2 Report

2019311779 최재경

1. Scheduling Diagram

xv6 book과 osoperating system, 소스코드를 보고 다음과 같이 Schdueling과정을 확인 할 수 있게 구성하였다.



Xv6에서, RUNNING인 프로세스는 interrupt, system call, 또는 exception에 의해 user스페이스에서 kernel 스페이스로 이동하며, trap에서 이를 처리한다.Timer interrupt는 `lapic.c` 가 담당하고, `trap.c` 에서 handling한다. trap.c에서는 먼저 syscall인지 확인하고 그 후에 timer interrupt인 경우를 확인한다. 둘다 아닐경우, exception이 발생한다.

Timer interrupt는 `yield()` 를 호출하고 이는 다시 `sched()` 를 호출하고, 여기에서 `swtch(struct proc **old, struct proc *new)` 를 호출하여 `swtch.S` 의 어셈블리코드로 context switch가 진행된다. `swtch` 는 `p->context` 에서 `cpu→scheduler` 로 context swtich를 하고, 따라서, `sched()` 로 리턴하지 않고 `scheduler()` 의 `swtch()` 호출 부분에서 시작된다. for(;;) loop안에서 다시 ptable에서 RUNNABLE이고 vruntime이 가장 작은 프로세스를 찾

아 `swtch()` 를 호출해 `p->context` 로 context switch한다. 이 과정은 프로세스가 trap으로 다시 들어오면서 반복된다.

Breidf code review of

- sleep

sleep()은 현재 프로세스 `struct proc *p = myproc();` 의 채널을 설정하고, state를 `SLEEPING` 으로 바꾼후 `sched()` 를 호출하여 context switch를 준비한다.

```
p->chan = chan;
p->state = SLEEPING;
sched();
```

- fork

fork()는 allocproc()으로 새로운 프로세스를 생성 및 초기화 후 , 부모와 같은 page table을 복사해서 가지는 과정이 구현되어 있고, fork()한 부모 프로세스는 RUNNING상태로 timeslice를 점유하며, 자식프로세스는 ptable에 RUNNABLE 상태로 새로 들어게가 된다.

- wait

wait를 호출한 프로세스가 `exit()` 하지 않은 자식프로세스가 하나라도 있다면, `sleep()` 을 호출한다. 만약 woken 되었을때, for(;;) 안으로 들어오게 되는데, `ZOMBIE` 인 자식프로세스를 찾고, 있다면 free해준다.

- kill

target process를 ptable에서 pid로 찾고, 그 프로세스의 p->killed =1로 해준다. 필요하다면 SLEEPING이라면 state도 RUNNABLE로 wakeup을 거치지 않고 바꿔주는데, 이는 나중에 결국 RUNNABLE에서 scheduling되거나 timer interrupet 를 통해 trap에서 p->killed =1 가 체크된뒤, exit()될 것이다.

- exit (wakeup)

`wakeup1()` 함수를 호출하여 channel(`curproc->parent`)에서 sleep하고 있는 프로세스, 즉 부모 프로세스의 state를 `RUNNABLE` 로 바꾼다. 또한 `scheduler()` 에서 woken process 인지를 판단하기 위해서 `p->woken = 1` 로 플래그를 설정해 준다.

```
if(p->state == SLEEPING && p->chan == chan) {
    p->state = RUNNABLE;
    p->woken = 1;
}
```

이후 현재 프로세스의 state를 `ZOMBIE` 로 바꾸고,(부모프로세스에서 free될 것이다.) `sched()` 를 호출한다. (for(;;)안에 들어있지 않으며, 다시 return하지 않는다.)

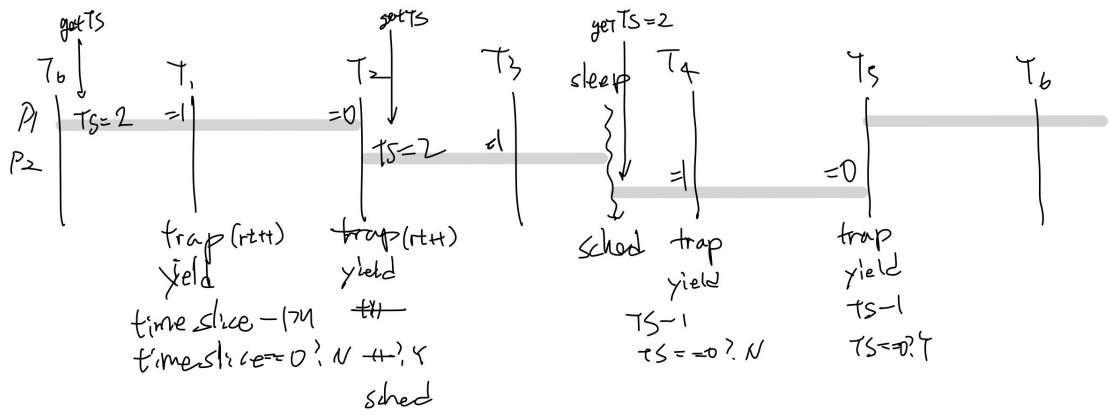
- mycpu()

`apicid` 를 확인하고, `cpus[]` 에서 현재 프로세서의 정보를 담은 `struct cpu*`를 리턴한다. (multiprocessor를 사용하지 않으므로 한개이다.)

- myproc()

`mycpu()`를 호출하여 현재 프로세서 `c` 에서 현재 working process인 `c->proc` 을 받는다.

2. Time Slicing synario



3. Code modifications

Makefile

Multiprocessor setting off.

```
ifndef CPUS
CPUS := 1
```

proc.h

Adding several member variables.

```
int niceval;
uint runtime;
uint vruntime;
uint prev_runtime;
int weight;
int timeslice;
char vrunchar[21]; //큰수에서의 vruntime을 비교하기 위해서 사용한다. 20자리수까지 나타낼 수 있다.
int woken;
```

trap.c

```
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER) {
    myproc()->runtime += 1000;
    myproc()->timeslice -= 1000;
    yield();
  }
```

Timeslice를 감소시키고, runtime을 증가시킨다. (1000militick 단위)

proc.c

```
int weight_of_niceval[40] =
{
  /*  0 */  88761,  71755,  56483,  46273,  36291,
  /*  5 */  29154,  23254,  18705,  14949,  11916,
  /*  10*/   9548,   7620,   6100,   4904,   3906,
  /*  15*/   3121,   2501,   1991,   1586,   1277,
  /*  20*/   1024,    820,    655,    526,    423,
  /*  25*/    335,    272,    215,    172,    137,
  /*  30*/    110,     87,     70,     56,     45,
  /*  35*/     36,     29,     23,     18,     15,
};
```

priority에서 weight의 계산을 하드코딩해주었다.

allocproc()

found 했을때 process를 초기화 하는 부분을 추가해주었다.

```
dd
```

yield()

```
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  if(myproc()->timeslice > 0) { //don't change current RUNNING state
    sched();
  }else{
    myproc()->state = RUNNABLE;
    sched();
  }
  release(&ptable.lock);
}
```

원래는 `RUNNING`을 `RUNNABLE`로 바꾼뒤 `sched()`를 호출하지만, 할당된 timeslice가 모두 소진되지 않았을 때, `scheduler()` 진입 후 바로 다시 scheduling 되기 위해서 `p->RUNNING` 의 state를 바꾸지 않는다.

sched()

```
//if(p->state == RUNNING)
//  panic("sched running");
```

`yield`()에서 호출됐을때, `p->RUNNING` 인지 더블체크하는 부분을 주석처리하여 timeslice가 남은 프로세스가 바로 scheduling되도록 한다.

scheduler()

runtime, vruntime, weight, timeslice가 업데이트 되는 곳이다.

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  //my varialbes
  int num_of_proc;
  int loop_cnt;
  int table_index;
  int total_weight;
  int mid_value;
  uint d_runtime;
  uint min_vruntime;

  for(;;){
    // Enable interrupts on this processor.
    num_of_proc = 0;
```

```
    loop_cnt = 0;
    table_index = 0;
    total_weight = 0;
    d_runtime = 0;
    min_vruntime = 4294967295;
    mid_value = 0;
    //min_vruntimechar[20] = "00000000004294967295";
    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 || p->state == RUNNING) {
//RUNNING 조건도 통과시켜서 timeslice가 남았을때, 바로 다시 scheduling될 수 있도록 한다.
        num_of_proc++; // 아래 루프에서 사용할 변수이다.
        d_runtime = p->runtime - p->prev_runtime;
        p->prev_runtime = p->runtime;
//trap.c 에서 prev_runtime을 업데이트 하지 않고, 차를 구한후, 업데이트 해 준다.

        p->weight = weight_of_niceval[p->niceval];
        total_weight += p->weight;
        mid_value = d_runtime * 1024 / p->weight;
        bigsum(p, mid_value); //p->vruntime += mid_value;

        if(p->state == RUNNING) {
          table_index = loop_cnt;
          goto do_unfinished_job; // Go to run RUNNING (which remains timeslice)
        }
//최소값과, ptable에서 인덱스를 찾는다.
        if(bigcomp(p, min_vrunchar)) {
          safestrcpy(min_vrunchar, p->vrunchar, sizeof(p->vrunchar));
          table_index = loop_cnt;
        }
      }
      loop_cnt++;
    }
    loop_cnt = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
//woken된 프로세스는 조금 다른 공식이 적용되므로, loop를 다시 순회하여 값을 바꿔준다.
      if(p->woken && p->state == RUNNABLE){
//if there is no RUNNABLE? no woken process.
        p->woken = 0; //woken의 값을 다시 정리한다.
        if(num_of_proc == 1) {
          p->vruntime = 0;
//RUNNABLE이 한개인 경우 =0 설정해준다. (RUNNING은 goto로 분기된다.)
        }else{
          mid_value = (1000 * 1024 / p->weight);
          //p->vruntime = min_vruntime - midvalue;
          bigdiff(p, mid_value);
//woken
        }

//최소값과, ptable에서 인덱스를 찾는다.
        if(bigcomp(p, min_vrunchar)) {
          safestrcpy(min_vrunchar, p->vrunchar, sizeof(p->vrunchar));
          table_index = loop_cnt;
        }
      }
```

```
      loop_cnt++;
    }

    if(num_of_proc == 0){
      release(&ptable.lock);
//RUNNABLE이 없을경우 continue로 무한 루프를 돌면서 interrupt를 기다린다.
      continue;
    }
    p = ptable.proc;
    p += table_index; //찾은 인덱스로 RUNNING할 프로세스를 가져온다.
    p->timeslice = 10000 * p->weight / total_weight;
//timeslice를 결정해준다. 10tick = 10 * 1000militick이다.

    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
do_unfinished_job:
  p = ptable.proc;
  p += table_index;

//context switching을 진행한다.
    c->proc = p;
    switchuvm(p);
    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);

  }
}
```

주석에 자세한 설명을 달았다.

bigsum()

int와 char * 의 덧셈 연산을 수행한다.

```
void
bigsum(struct proc *p, int val){
  char x[21] = "00000000000000000000";
  char y[21] = "00000000000000000000";
  int sum = 0;
  int carry = 0;

  for(int i = 19; val > 0; i--) {
    x[i] = (val % 10) + '0';
    val /= 10;
  }
```

```
  for(int i = 19; i > 0 ;i--) {
    sum = 0;
    sum = carry + (p->vrunchar[i] - '0') + (x[i] - '0') ;
    carry = sum / 10;
    sum %= 10;
    y[i] = sum +'0';
  }
  safestrcpy(p->vrunchar, y, sizeof(p->vrunchar));
}
```

bigcomp()

vrunchar가 작을경우 true를 리턴해 min값을 찾는다.

```
int
bigcomp(struct proc *p, char* min){ //is vruntime is small than min value?
  for(int i = 0; i < 20; i++){
    if(p->vrunchar[i] - '0' < min[i] -'0') {
      return 1; //true
    }else if(p->vrunchar[i] - '0' > min[i] - '0'){
      return 0; //false
    }//same digit
  }
  return 0; //exactly
}
// if use '<=', the latter process on ptable will be selected
```

bigdiff()

*vruntime = min_vruntime - (1000 * 1024 / current weight);* 와 같은 연산을 수행한다.

```
void
bigdiff(struct proc *p, int val){
  char x[21] = "00000000000000000000";
  char y[21] = "00000000000000000000";
  int borrow = 0;


  for(int i = 19; val > 0; i--) {
    x[i] = (val % 10) + '0';
    val /= 10;
  }
  if(bigcomp(p, x)){
    safestrcpy(p->vrunchar, "00000000000000000000", sizeof(p->vrunchar));
    return;
  }
  for (int i = 19; i >= 0; i--) {
    int diff = (p->vrunchar[i] - '0') - (x[i] - '0') - borrow;
    if (diff < 0) {
        diff += 10;
        borrow = 1;
    } else {
```

```
        borrow = 0;
    }
    y[i] = diff + '0';
  }
  safestrcpy(p->vrunchar, y, sizeof(p->vrunchar));
}
```

4. Handling Overflow

Xv6는 내부적으로 `unit tick` 을 이용하여 타이머의 tick을 측정한다. 하지만 32-bit 변수이므로, 0~2^32-1(0 to 4,294,967,295)범위만 표현이 가능하며, vruntime은 militick 단위이므로 이 범위를 금방 넘어서게 된다. 따라서 `vruntime_char[20]` 을 이용하여, 20자리수 까지 나타낼 수 있는 적은 범위에서의 정수 overflow를 handling 할 수 있다. vruntime은 덧셈과 최솟값 비교의 연산을 수행하므로, 이에 맞추어 핸들링 함수를 proc.c에 따로 구현하면된다. (`bigsum(struct *proc, int val)`, `bigcomp(struct *proc, int val)` , bigdiff(sturct proc *p, int))

5. Test by mytest1, 2, 3

mytest.c mytest2.c mytest3.c

`mytest`을 먼저 시작하고, `mytest2`, `mytest`도 순차적으로 백그라운드에서 실행한다. (또는 동시에 실행한다.) `fork()` 후에 자식프로세스는 무한 루프를 도는데, 이는 자원을 놓고 1, 2 3의 mytest가 경쟁하게 한다.

```
setnice(getpid(), 5); //mytset는 5, mytest2는 35, mytest3은 27
int pid;
pid = fork();
if(pid < 0) {
  printf(2, "mytest fork problem\n");
}else if(pid == 0) {
  for(;;){}
}else {
  wait();
```

- priority가 같을 경우

먼저 실행되는 프로세스가 먼저 많은 runtime을 점유하고, 따라서 vruntime도 커질 것이므로, 나중에 실행되는 프로세스의 vruntime 값이 적어 scheduling에 이점을 갖고, 많은 context switch후에는 결국 vruntime의 값이 거의 같아지게 된다. 부모 프로세스는 wait()으로 자식프로세스가 무한 루프를 실행하므로, `SLEEPING`에서 나오지 않는다.

- priority가 다르고 동시에 실행할 경우(0에 가까울수록 큰 priority를 가진다.)

우선순위에 따라 실행시간이 달라지지만, vruntime은 유사해진다.

6. Result

```
Name: JaeKyung Choi
=========Hello=========
$ mytest1&
$ mytest2&
$ mytest3&
$ ps 0
name    pid     state           priority        runtime/weight  runtime                 vruntime                tick 1159000
init    1       SLEEPING        20              1               2000                    1000
sh      2       SLEEPING        20              1               2000                    0
mytest1 5       RUNNABLE        5               23              690000                  24185
mytest1 4       SLEEPING        5               0               2000                    35
mytest2 8       RUNNABLE        35              27              1000                    28444
mytest2 7       SLEEPING        35              0               1000                    0
mytest3 11      RUNNABLE        27              27              6000                    28572
mytest3 10      SLEEPING        27              0               1000                    0
ps      12      RUNNING         20              0               0                       0
$ ps 0
name    pid     state           priority        runtime/weight  runtime                 vruntime                tick 2048000
init    1       SLEEPING        20              1               2000                    1000
sh      2       SLEEPING        20              1               2000                    0
mytest1 5       RUNNABLE        5               53              1570000                 54985
mytest1 4       SLEEPING        5               0               2000                    35
mytest2 8       RUNNABLE        35              55              2000                    56888
mytest2 7       SLEEPING        35              0               1000                    0
mytest3 11      RUNNABLE        27              55              12000                   57144
mytest3 10      SLEEPING        27              0               1000                    0
ps      13      RUNNING         20              0               1000                    1000
$ ps 0
name    pid     state           priority        runtime/weight  runtime                 vruntime                tick 2846000
init    1       SLEEPING        20              1               2000                    1000
sh      2       SLEEPING        20              1               2000                    0
mytest1 5       RUNNABLE        5               80              2360000                 82635
mytest1 4       SLEEPING        5               0               2000                    35
mytest2 8       RUNNABLE        35              83              3000                    85332
mytest2 7       SLEEPING        35              0               1000                    0
mytest3 11      RUNNABLE        27              83              18000                   85716
mytest3 10      SLEEPING        27              0               1000                    0
ps      14      RUNNING         20              0               0                       0
$ ps 0
name    pid     state           priority        runtime/weight  runtime                 vruntime                tick 3171000
init    1       SLEEPING        20              1               2000                    1000
sh      2       SLEEPING        20              1               2000                    0
mytest1 5       RUNNABLE        5               91              2680000                 93835
mytest1 4       SLEEPING        5               0               2000                    35
mytest2 8       RUNNABLE        35              111             4000                    113776
mytest2 7       SLEEPING        35              0               1000                    0
mytest3 11      RUNNABLE        27              93              20000                   95240
mytest3 10      SLEEPING        27              0               1000                    0
ps      15      RUNNING         20              0               1000                    1000
```