OS project 1 report

b06902111 資工三 林慶珠

設計

我使用兩顆cpu，一顆scheduler，一顆專門跑process(job)。

以下的變數意義：`running` 表示現在跑的process的index，如果`running == -1` 表示沒人在跑。一個process做完或還沒ready `pid == -1` (沒工作要做)。

我用process的ready time當成start time，也就是行程的產生時間當成start time。

Scheduler

如果是FIFO, SJF, PSJF都是有新的人ready才跑algorithm決定誰要跑：

```
if( policy != RR && new_come == 1 ){
        //select one from algorithm, wake up next
        int next = next_process( proc, N, policy);
        //next == -1 : no job ready / all done
        if( next != -1 ){
                if( running != next ){
                        proc_setscheduler(proc[next].pid, SCHED_OTHER); //wakeup
                        proc_setscheduler(proc[running].pid, SCHED_IDLE); //block

                        if( running != -1 ) last_run = running;
                        running = next;
                        rr_time_cumulate = 0;
                }
        }
}
```

如果是RR就是如果現在沒人跑，或是某個人的time quatum用完了，才跑algorithm：

```
else if( running == -1 || rr_time_cumulate%RR_TIMEQUANTUM == 0 ){
        //select one from algorithm, wake up next
        int next = next_process( proc, N, policy);
        //next == -1 : no job ready / all done
        if( next != -1 ){
                if( running != next ){
                        proc_setscheduler(proc[next].pid, SCHED_OTHER); //wakeup
                        proc_setscheduler(proc[running].pid, SCHED_IDLE); //block

                        if( running != -1 ) last_run = running;
                        running = next;
                        rr_time_cumulate = 0;
                }
        }

}
```

FIFO

如果有人在跑就讓他繼續跑(non-preemptive)，跑完會在scheduler那邊設成`pid=-1` 。因為一開始的時候proc是用ready time排序，所以直接iterate proc直接拿到下一個。

```
if( policy == FIFO ){
    if( running != -1 ) return running; //non-preemptive

    //proc is sorted by ready time
    for( int i = 0 ; i < N ; i++){
        if( proc[i].pid != -1 ) return i; //ready and has job
    }
    return -1; //no next, all job done
}
```

RR

如果有人在跑而且他在一個time quantum裡面就讓他繼續跑。

如果要換人有分兩種：上一個跑完了，此時`running == -1` 會從`last_run` 知道剛剛跑完的是誰，那麼就從proc裡面的下一個process開始找是否有工作要做。如果一個跑到一半時間到了，會從`running` index往後找誰有工作要做。

```
else if( policy == RR ){
    //run complete time quantum
    if( running != -1 && (rr_time_cumulate % RR_TIMEQUANTUM) != 0 ){
        if( rr_time_cumulate >= RR_TIMEQUANTUM ) rr_time_cumulate %= RR_TIMEQUANTUM;
        return running;
    }

    if( running == -1 ){ // pick new one from last
        for( int i = last_run+1; i < N+last_run+1 ; i++)
            if( proc[(i%N)].pid != -1 ) return (i%N);
    }
    else{ //time up pause
        for( int i = running+1 ; i < N+running+1 ; i++){
            if( (i%N) == running ) continue;
            if( proc[(i%N)].pid != -1 ) return (i%N);
        }
    }
    return -1; //no next, all job done
}
```

SJF 和 PSJF

SJF 和 PSJF 只有差在是不是preemptive而已。如果有剩下最少的就執行它，如果剩下最少的人跟`running` 剩下的時間一樣就繼續執行running process（減少context switch）。

```
else{ //SJF and PSJF

        if( policy == SJF ){
                if( running != -1 ) return running; //non-preemptive
        }

        int min_idx = -1;
        for( int i = 0 ; i < N ; i++){
                if( proc[i].pid == -1 ) continue;

                if( min_idx == -1 ) min_idx = i;
                else if( proc[i].t_exec < proc[min_idx].t_exec ) min_idx = i;
        }

        if( min_idx == -1 ) return -1; //no job

        if( running != -1 && proc[min_idx].t_exec == proc[running].t_exec ) return running;
        else return min_idx; //less context switch
}
```

核心版本

Linux 4.14.25

比較實際結果與理論結果，並解釋造成差異的原因

一開始我把應該要印在kernel log的message印出來的時候，我發現跑process(job)的cpu會比scheduler的cpu早完成工作（順序對，會早一點點），這可能是因為scheduler不只有計算時間（需要計算時間來看誰ready），也有判斷現在現在該輪到誰來做，所以花的時間會比較久。

在TIME_MEASUREMENT中把job的elapsed time除以經過的number of time unit，得到500 unit time的時間約為1.4秒：

in `TIME_MEASUREMENT.txt`

```
FIFO
10
P0 0 500
P1 1000 500
P2 2000 500
P3 3000 500
P4 4000 500
P5 5000 500
P6 6000 500
P7 7000 500
P8 8000 500
P9 9000 500
```

in `TIME_MEASUREMENT_dmesg.txt`

```
[ 2462.827268] [Project1] 3889 1588130588.354542404 1588130589.761368884
[ 2465.710026] [Project1] 3890 1588130591.064452931 1588130592.644430680
[ 2468.455731] [Project1] 3891 1588130593.979743087 1588130595.390366477
[ 2471.207885] [Project1] 3892 1588130596.679003235 1588130598.142754664
[ 2473.927891] [Project1] 3893 1588130599.460023386 1588130600.862996446
[ 2476.648006] [Project1] 3894 1588130602.149134238 1588130603.583356075
[ 2479.392677] [Project1] 3895 1588130604.868779569 1588130606.328260342
[ 2482.224868] [Project1] 3896 1588130607.681796185 1588130609.160697319
[ 2484.897541] [Project1] 3899 1588130610.438555568 1588130611.833604624
[ 2487.705526] [Project1] 3900 1588130613.155581789 1588130614.641826581
```

可以從`FIFO_1.txt` / `FIFO_1_dmesg.txt` 看到同時ready多個job會有誤差（0.1秒），也可以從`FIFO_3.txt` / `FIFO_3_dmesg.txt` 看到如果job的execution time很長，他的誤差會較多。

`FIFO_1.txt`：

```
FIFO
5
P1 0 500
P2 0 500
P3 0 500
P4 0 500
P5 0 500
```

`FIFO_1_dmesg.txt`：

```
[ 1351.475720] [Project1] 3518 1588129476.836389991 1588129478.353549028
[ 1352.973394] [Project1] 3519 1588129476.844718817 1588129479.851466092
[ 1354.456556] [Project1] 3520 1588129476.848961126 1588129481.334839433
[ 1355.872072] [Project1] 3521 1588129476.836701899 1588129482.750617922
[ 1357.326653] [Project1] 3522 1588129476.840755525 1588129484.205449790
```

`FIFO_3.txt`：

```
FIFO
7
P1 0 8000
P2 200 5000
P3 300 3000
P4 400 1000
P5 500 1000
P6 500 1000
P7 600 4000
```

`FIFO_3_dmesg.txt`：

```
[ 1641.107408] [Project1] 3578 1588129744.812222497 1588129767.997289297
[ 1656.106115] [Project1] 3579 1588129747.493188560 1588129782.994456463
[ 1664.866060] [Project1] 3580 1588129747.745032920 1588129791.753646861
[ 1668.040825] [Project1] 3581 1588129748.033063979 1588129794.928151158
[ 1670.743094] [Project1] 3582 1588129748.332810256 1588129797.630238692
[ 1673.100711] [Project1] 3583 1588129748.316821327 1588129799.987668696
[ 1682.721070] [Project1] 3584 1588129748.604762707 1588129809.607451039
```

使用說明

執行之前要先照著hw1的投影片把`kernel_files`的檔案放到該放的位置並編譯kernel。

```
$ make
$ sudo ./main.out < [data in] > [data out]
```

＊我的demo影片太大放不上git，放到雲端。

參考資料

Linux man page

https://www.gnu.org/software/libc/manual/html_node/index.html#SEC_Contents

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/sect-using_library_calls_to_set_priority

https://www.princeton.edu/~cad/emsim/files/example.c