Design Doc 2

John Carlyle, Andrew Edwards,
Morgan McDermott, and Kaya Zekioglu

May 3, 2014

1 How to install/test

Untar, enter the directory `src`, and then run `install.sh`. This script will copy the relevant files to their proper locations under `/usr/src` and then will run `make clean install` in the `/usr/src/tools` directory. After this finishes restart minix and select option 2: the custom version of minix. Once you have logged back in return to the uncompressed directory (where `install.sh` was located) and enter `tests`.

```
src
├── README
├── install.sh
├── include
│   └── minix
│       └── config.h
├── servers
│   ├── pm
│   │   ├── schedule.c
│   │   └── utility.c
│   └── sched
│       ├── sched.h
│       ├── schedproc.h
│       └── schedule.c
└── tests
    ├── Makefile
    ├── IO_bound.sh
    ├── cpu_bound.c
    ├── runner.pl
    ├── test1.sh
    ├── test2.sh
    ├── test3.sh
    ├── test4.sh
    └── test5.sh
```

First run `make` to compile the CPU bound and IO bound executables and then proceed to execute `testN.sh` for $N \in \{1, \dots, 5\}$. See the table below for what each of these scripts demonstrates. Note that if you wish to install the dynamic version of the scheduler then before running `install.sh` edit the file `servers/sched/sched.h` and change the macro `DYNAMIC` to the value 1 (setting this to zero yields a static lottery scheduler). We show that "our dynamic scheduler actually does dynamically adjust tickets of processes" with a print statement in `do_noquantum`. As long as the dynamic lotter scheduler is in place and there are CPU bound processes, you will see messages indicating when a ticket have been taken and which endpoint it has been taken from.

Script	Property of Scheduler
`test1.sh`	Two equal CPU bound tasks with equal tickets lets them run at about the same speed
`test2.sh`	Two CPU tasks where 1 has twice the tickets, the task with more tickets finishes in 3/4 the time
`test3.sh`	Three CPU tasks with 25, 50, and 100 tickets runs the tasks in the right ratio
`test4.sh`	Several CPU tasks with 100 tickets doesn't completely starve another task with just 1 ticket
`test5.sh`	The dynamic scheduler improves performance when you mix CPU and IO bound tasks compared to keeping a fixed number of tickets for each process

2 Implementation

2.1 Kernel Space

The first step in our design was to decide what to change in the kernel. We knew that at least 2 queues would be necessary, one for lottery winners, and the other for the losers. Initially we desired to append two new queues to the 16 that minix already has, however after reading kernel source we opted to instead repurpose the last 2 queues. The goal of our kernel changes is to provide a means to differentiate user processes from those initialized by the system. If we can separate out user processes in our scheduler this will allow us to leave the scheduling of system processes unchanged, and hence not break anything. These changes can be seen in `config.h`.

2.2 User Space

Now to address what we change in user space. To store the ticket count of each process, we add an unsigned integer field the struct `schedproc` within `schedproc.h`. Also, we add a preprocessor constant `DYNAMIC` to `sched.h`, which can be used before installation to toggle using the dynamic lottery. The main file we edit is `schedule.c`. There are 6 different entry points for the scheduling server: `do_noquantum`, `do_start_scheduling`, `do_stop_scheduling`, `do_nice`, `init_scheduling`, and `balance_queues`.

2.2.1 `init_scheduling`

In `init_scheduling` we preform initialization; specifically this is where we seed our PRNG.

2.2.2 `do_start_scheduling`

In the `do_start_scheduling` function we will check if the process being started is a user or system process, if it is the latter we allow the original minix scheduling algorithm to run. If it is a user process then we initialize its ticket count to 20 and also increment the global ticket count by the same.

2.2.3 `do_stop_scheduling`

In the `do_stop_scheduling` function we preform the reverse operation. If it is a user process we decrement the global ticket count by the number of tickets it had and then we unset its `IN_USE` flag.

2.2.4 `do_nice`

To implement our desired changes to the `do_nice` function we need to be able to use the raw value `x` in the expression `nice -n x ./any`. Since this information is not normally passed with a nice pm, we will edit the pm format for the scheduler (`/servers/pm/schedule.c`) to contain it. Originally this pm passed a value for `SCHEDULING_MAXPRIO`, but since system processes do not call nice, we are not changing the default minix scheduler by modifying this field. Once we make this change the `do_nice` function receives as input a process and the desired ticket delta. If the desired increase or decrease in tickets would take that process' number of tickets below 1 or above `MAX_TICKETS` then we floor or ceiling the new number with said bounds respectively. Again the global ticket count is updated at this point to reflect the change.

2.2.5 `do_noquantum`

The function `do_noquantum` is where we preform our lottery. If the process that just ran out of quantum was a user process (i.e. its max priority is that of a user process) then we demote its priority to the losers queue and call a new lottery. Additionally when the `DYNAMIC` flag is on this is where our scheduler will remove a ticket from the offending CPU bound process' pool.

3 Test Results

Each test was performed with 5 identical trials using the Perl testing script described above.

3.1 Test 1

Show that running two equal CPU bound tasks with equal tickets lets them run at about the same speed

Command	Tickets	Average Time	Standard Dev.	Ratio
./cpu_bound 10	100	21.195s	0.669	1
./cpu_bound 10	100	22.046s	0.213	1.04

3.2 Test 2

Show that running two CPU tasks where 1 has twice the tickets, the tasks' finishing times are in ratio 3 : 4.

Command	Tickets	Average Time	Standard Dev.	Ratio
./cpu_bound 10	50	22.375s	0.232	1
./cpu_bound 10	100	16.826s	0.883	0.752

3.3 Test 3

Show that running three processes CPU tasks with 25, 50, and 100 tickets runs the tasks in the right ratio. (The proper ratio should be 7 : 10 : 12)

Command	Tickets	Average Time	Standard Dev.	Ratio
./cpu_bound 10	25	33.21s	0.236	1
./cpu_bound 10	50	28.628s	0.663	0.862
./cpu_bound 10	100	20.6s	0.62	0.62

3.4 Test 4

Show that running several CPU tasks with 100 tickets does not completely starve another task with just 1 ticket.

Command	Tickets	Average Time	Standard Dev.	Ratio
./cpu_bound 10	100	30.689s	1.765	1
./cpu_bound 10	100	32.302s	1.552	1.052
./cpu_bound 10	100	31.753s	0.653	1.034
./cpu_bound 10	1	43.544	0.486	1.418

3.5 Test 5

Show that your dynamic scheduler improves performance when you mix CPU and IO bound tasks compared to keeping a fixed number of tickets for each process

When we run test 5 with the static compared to the dynamic, we do not see an improvement in performance. We included a print statement (commented out by default) to show that tickets are indeed removed when users run out of quantum. However, we were unable to award processes tickets when the *do* block. It is also very apparent that only one process (the CPU bound process) is losing tickets. We theorize that, if we could add tickets to the IO process when it blocks, then the performance would improve.