Lab 1: Scheduling

Ankit Goyal
ankit@cs.utexas.edu
CS380L

September 23, 2014

1 Setup

1.1 Hardware

Host Processor: 64 bit 4 core Intel(R) Xeon(R) CPU E3-1270 V2 @ 3.50GHz
Host Memory: 16GB
HyperThreading: Yes
Logical CPUs after Hyperthreading: 8
CPU frequency scaling: Disabled in BIOS (turned off Intel SpeedStep and C-states)

1.2 Software

Host Operating System: Ubuntu with 3.13.0-34-generic 64 bit kernel.
libcgroup.h used for managing cgroups programatically.
cityhash.h c++ version from Google, used to calculate 128bit Hash values.

2 Single Process Hashing

Single process computed about 722000 Hashes in 5 seconds.

3 Multi Process Hashing

Hyperthreading The processor has 4 physical cores and 8 logical cores (due to Hyperthreading). Table 1 shows that the rate of increase in total time is less when threads are increased from 4 to 8 than when increasing from 8 to 16. *Hyperthreading is better than no Hyperthreading but not as better as extra physical cores.* Note there are no explicit background processes running.

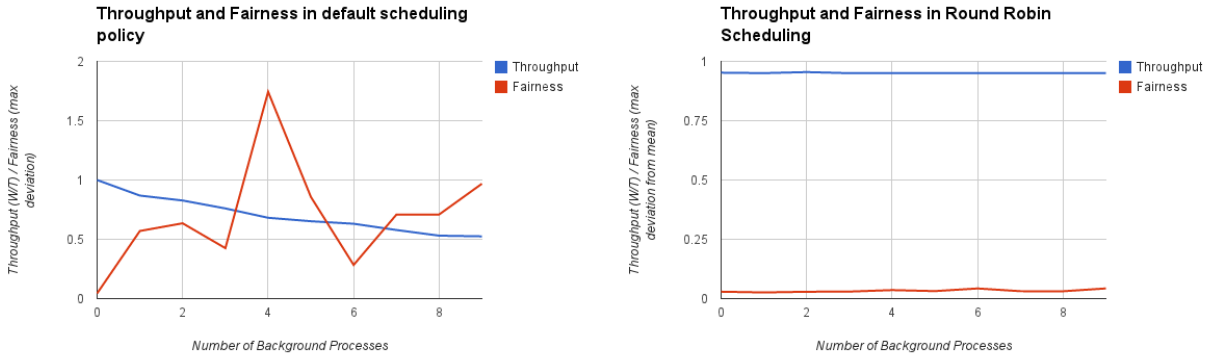
Number of Hashing Processes	Total Time
4	5.515
8	8.056
16	16.113
32	32.229

Table 1: Showing increase in total time with number of hashing processes.

Hashing Processes #	Background Processes #	Total Time	Throughput	max deviation from mean (fairness)
8	8	15.2418	0.5249	0.7092
8	9	15.5910	0.5131	0.9686

Table 2: Throughput for different number of background and hashing processes.

N vs N+1 Processes Table 2 shows the total time taken, throughput and fairness for (8 hashing, 8 background) and (8 hashing, 9 background) processes. Total time is the average value taken over 5 different runs. Throughput in (8 hashing, 8 background) is slightly higher than the throughput in (8 hashing, 9 background) since there are more number of processes to compete with.



(a) (b)

Figure 1: Throughput (higher is better) and Fairness (lower is better) with different number of background processes. (a) Default Scheduling Policy (`SCHED_OTHER`) (b) Round Robin Scheduling Policy (`SCHED_RR`)

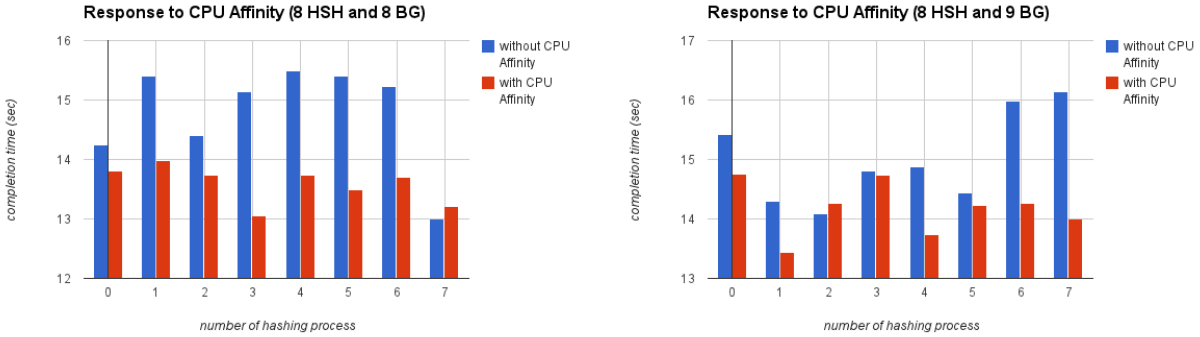
Figure 1a shows that throughput decreases (in case of fixed hashing processes) as number of background processes increase which is expected since the processor is shared among more processes and less work (by hashing process) is done per unit time. However fairness is not consistent and depends on each run. The Completely Fair Scheduler(CFS) algorithm tries to run the task with the smallest `p->se.vruntime` (i.e., the task which executed least so far). Since there are number of background processes and in different runs the order in which they are executed effects the fairness among hashing threads; some of the threads can finish before the background processes and some will finish after them. In essence, if we take system as a whole CFS is quite fair. In section 5, we see how fairness and (throughput) can be increased among a group of tasks.

4 Throughput

4.1 `sched_setaffinity`

Using `sched_setaffinity`, we attach each hashing process to a specific CPU. CPU Affinity improves cache performance since whenever a processor adds a line to its cache, all other processors needs to invalidate the data in their cache. In certain cases there could be huge improvements in performance due to affinity. The scheduler in 2.5 exhibit excellent natural affinity and prevents bouncing of a process from one CPU to another. CPU affinity could be very effective in case data is shared among different threads (in our case there's no sharing).

Figure 2 shows the effect of `sched_setaffinity` in both (8 Hashing, 8 Background) and (8 Hashing, 9 Background processes). In both cases the throughput increased by 10.9% and 9.5% respectively.



(a) (b)

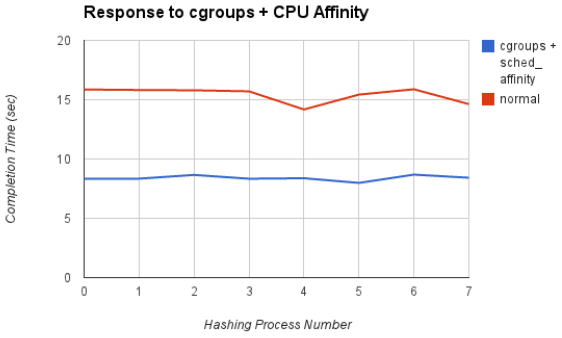
Figure 2: Response to CPU affinity: `completion time` vs `hashing process number`

4.2 `cgroups`

Control Groups allow you to allocate resources - such as CPU time, system memory, network bandwidth - among user defined group of tasks (processes) running on the system. Using cgroups we can increase the amount of CPU available for hashing processes, which can significantly improve the throughput.

To increase `cpu.share` for hashing processes, a new cgroup for each process was created under the subsystem CPU. The `cpu.share` for all the tasks in these cgroups was increased to 2048 (default value is 1024). As a result all the hashing processes get twice the amount of cpu than other processes with default cgroups (background processes belong to default cgroups).

Figure 3 (8 Hashing, 9 Background Processes) shows relative time taken by each process with and without cgroups (and sched_affinity). **Throughput was increased by 79.5%**, since the CPU was given twice to hashing processes than the background processes. It doesn't seem very useful to talk about fairness here, since we deliberately increased the CPU share and tied each hashing process to a different logical CPU.



(a)

Type	Total Time (sec)	Throughput
normal	15.5910	0.5131
cgroup+affinity	8.684	0.9212

(b)

Figure 3: **Response to CPU affinity + cgroups**. (a) Relative time taken by each process with and without cgroup + affinity, (b) Throughput increase of 79.5% for cgroup + affinity

```
void computeHash(int ITERATIONS){
  for (int i = 0; i < ITERATIONS; i++) {
    if (BE_FAIR && i != 0 && i % (ITERATIONS/numProc) == 0){
      sched_yield();
    }
    /* calculate hash here */
     CityHash128(buf, 4096);
}

/* change priority and scheduling for each process to Round Robin */
inline void makeFair(int pid) {
  struct sched_param param;
  param.sched_priority = 99;

  if (sched_setscheduler(pid, SCHED_RR, &param) != 0) {
    perror("sched_setscheduler");
    exit(EXIT_FAILURE);
  }
}
```

Codeblock 1: Implementation of scheduling policy change and voluntarily yielding.

Process Number #	Total Time Taken	Deviation from mean (fairness)
0	8.37	0.0305
1	8.308	0.0315
2	8.37	0.0305
3	8.374	0.0345
4	8.308	0.0315
5	8.308	0.0315
6	8.37	0.0305
7	8.308	0.0315

Table 3: Deviation from mean with `SCHED_RR` scheduling policy. (8 Hashing, 9 Background Processes)

5 Fairness

The default scheduler in linux - CFS (Completely Fair Scheduler) does quite a good job in scheduling fairly most of the times. However it's not always fair, in some cases the deviation from mean was a bit higher than normal as can be seen in Figure 1a.

One way to ensure fairness is by using Round Robin Scheduling (`SCHED_RR`) with high priority and yielding each process after a fix number of iterations. This ensures fairness among the hashing processes in all cases. However this doesn't provide fairness among other processes (for e.g., background processes) and *only root can do that.*

`BE_FAIR` flag can be used to enable `SCHED_RR` scheduling with priority of 99 (max) in the submitted code. Codeblock 1 shows the basic implementation, where `makeFair` (which changes the scheduling policy and priority) is called before initiating the compute loop. Inside the compute loop (in `computeHash` function) each hashing process voluntarily yields the processor to other hashing processes with same priority after a fixed number of iterations, hence giving fair share to all hashing processes belonging to same priority.

Table 3 shows the total time taken by each process under Round Robin scheduling (`SCHED_RR`). **The maximum**

deviation from mean in this case is 0.03 compared to 0.96 with normal scheduling (`SCHED_OTHER`).

Moreover, Figure 1b shows that both the throughput and Fairness are highly consistent since we have hashing processes at much higher priority than background processes. It's interesting to see that throughput is better than normal scheduling and time is fairly distributed. This is true, since our hashing processing are running at higher priority than background processes.

Minor Implementation Comment: The branch in `computeHash`, even though will most likely be optimized by compiler in case `BE_FAIR` is false, is still put under pre-processor so that it doesn't get generated if `BE_FAIR` is false. This is done to keep the work constant as the compiler may or may not optimize or branch predict.

6 Cryptographic vs Non-Cryptographic Hash

A cryptographic hash function provides security guarantees that the non-cryptographic hash functions may or may not provide. Most importantly, in cryptographic hash functions, it should be hard to find collisions or pre-images. **Hard** usually means it takes very long time or is practically impossible to find collisions or pre-images as the length increases. Due to these guarantees, cryptographic functions are usually slower than non-cryptographic functions. Example usage of non-cryptographic functions are CRC checks and cryptographic are for storing passwords.

Time Spent on the lab ≈16 hours

7 References:

1. `http://www.linuxjournal.com/article/6799`
2. `http://security.stackexchange.com/questions/11839`
3. `http://linux.die.net/man/2/clone`
4. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`