

CS50300: Operating Systems

LAB3 ANSWERS

Zichen Wang
wang4113@purdue.edu

October 14, 2018

1 Performance evaluation of Linux CFS scheduling

In R3 mode, the LOOP1 and LOOP2 are 10 and 10000000 for CPU-bound processes, and they are 150 and 600000 for IO-bound processes. The IOSLEEP is 100ms.

1. Scenario 1: create 8 app processes that are all CPU-bound. Both R3 and CFS scheduling have the very similar performance. Their measured CPU usage and waiting time is approximately the same. The order of termination is the same as the order of creation. Table 1 and 2 show the performance of CFS scheduling and R3 scheduling respectively.
2. Scenario 2: create 8 app processes that are all I/O-bound. The performance of both R3 and CFS scheduling is approximately the same. There is only minor difference in the average waiting time. Table 3 and 4 show the performance of CFS scheduling and R3 scheduling respectively.

PID	proctype	clktimemilli (ms)	gross CPU usage (ms)	average waiting time (ms)
4	0	17865	2258	171.153846
5	0	18051	2259	172.912087
6	0	18061	2258	173.32967
7	0	18072	2258	173.153846
8	0	18083	2259	173.263736
9	0	18093	2258	173.384615
10	0	18104	2258	173.505494
11	0	18115	2259	173.615384

Table 1: The performance of Linux CFS scheduling in scenario 1

PID	proctype	clktimemilli (ms)	gross CPU usage (ms)	average waiting time (ms)
4	0	18040	2258	173.76923
5	0	18051	2259	173.186813
6	0	18061	2258	173.307692
7	0	18072	2258	173.428571
8	0	18083	2259	173.538461
9	0	18093	2258	173.659340
10	0	18104	2258	173.780219
11	0	18115	2259	173.890109

Table 2: The performance of R3 scheduling in scenario 1

PID	proctype	clktimemilli (ms)	gross CPU usage (ms)	average waiting time (ms)
10	1	17238	1956	1.42944
9	1	17507	1970	2.398907
11	1	17704	1976	3.92307
6	1	17831	1983	3.798076
5	1	17928	1991	4.115207
7	1	18153	1995	4.767543
4	1	18211	2002	4.991525
8	1	18564	2015	5.696498

Table 3: The performance of Linux CFS scheduling in scenario 2

PID	proctype	clktimemilli (ms)	gross CPU usage (ms)	average waiting time (ms)
11	1	17453	1965	2.547486
7	1	17710	1977	3.522613
5	1	17792	1982	3.740384
6	1	18033	1988	4.583710
8	1	18065	1989	4.745454
9	1	18124	1992	4.845814
10	1	18200	1993	5.131004
4	1	18216	1997	5.94420

Table 4: The performance of Linux R3 scheduling in scenario 2

PID	proctype	clktimemilli (ms)	gross CPU usage (ms)	average waiting time (ms)
4	0	15293	2271	87.167785
6	0	15333	2269	89.689655
7	0	15425	2272	87.879194
5	0	15431	2275	86.735099
8	1	19077	1958	13.121019
10	1	19091	2043	12.431250
9	1	19253	1956	13.900621
11	1	19289	2049	13.462962

Table 5: The performance of Linux CFS scheduling in scenario 3

PID	proctype	clktimemilli (ms)	gross CPU usage (ms)	average waiting time (ms)
8	1	17085	1950	0.662251
10	1	17112	1950	0.841059
4	0	17123	2341	78.26455
11	1	17126	1951	0.927152
9	1	17140	1953	0.993464
5	0	17172	2340	77.878947
6	0	17194	2341	77.989473
7	0	17216	2341	78.105263

Table 6: The performance of Linux R3 scheduling in scenario 3

- Scenario 3: create 4 app processes that are CPU-bound and 4 app processes that are I/O-bound. The performance of 4 CPU-bound processes is approximately the same, and the same goes for the 4 I/O-bound processes. The average waiting time in CFS is slightly larger than that in R3. Table 5 and 6 show the performance of CFS scheduling and R3 scheduling respectively.

2 Dynamic workload of Linux CFS scheduling

Table 7 shows the dynamic workload of Linux CFS scheduling for 4 CPU-bound and 4 I/O-bound processes. Compared with Table 5, the tendency is approximately the same. Among the 4 CPU-bound processes their gross CPU usage and average waiting time is approximately the same, and the same goes for the 4 I/O-bound processes. However, the finish time of each process in each group is not the same since a 500ms delay is injected between successive process creation.

PID	proctype	clktimemilli (ms)	gross CPU usage (ms)	average waiting time (ms)
4	0	9256	2266	73.210526
5	0	11891	2271	71.904761
6	0	12982	2275	74.209302
7	0	13318	2282	69.977611
8	1	20800	1989	10.73170
9	1	21238	1973	9.619631
10	1	21515	1971	8.273291
11	1	22292	1981	9.497005

Table 7: The performance of Linux CFS scheduling in dynamic workload

3 Performance evaluation of real-time RMS scheduling

3.1 Test 4 real-time periodic processes

The computation time and periods of real-time periodic processes are (10, 50), (20, 200), (3, 30) and (7, 100). The admission control is less than 0.5, so they can be created by `rms_create()`. Before creating these processes, we give the `main()` process a sufficient large priority so that all real-time processes are ready before running. The `XINUSCHED` is in `R3` mode. Each real-time process has 100 period by default. Table 8 shows the part of results.

1. The real-time process with highest priority will finish its periods first, since it will run first and preempt other real-time processes after waking up. The process with second highest priority will finish second and so on.
2. Since we have 4 real-time processes, there will be 4 patterns in the results. Table 8 shows the 4 patterns respectively. First, 4 processes will scramble for the CPU. The second part contains 3 processes, and so on. In the end, only real-time process (20, 200) runs on the CPU.

Issue: In my opinion, this method for testing of real-time processes is not good enough. First, we expect that each real-time process start to record its `period_start` at the same time, but we cannot give them the same `clktimemilli` in this way, since one process will record `period_start` only after other processes with higher priority sleep. Second, after one process wakes up, it should immediately be given a new `period_start`. However, if there is another process with higher priority running now, it will wait but this waiting time will not count in the next new period. In one word, `period_start` would not be the same as the true start time of a period.

PID	x	y	period number	clktimemilli	y - (clktimemilli - period_start)
6	3	30	1	23	27
4	10	50	1	34	40
7	7	100	1	42	93
6	3	30	2	54	27
5	20	200	1	67	176
6	3	30	3	85	27
4	10	50	2	89	36
6	3	30	4	116	27
4	10	50	3	136	40
6	3	30	5	147	27
7	7	100	2	148	89
6	3	30	6	178	27
...
5	20	200	18	3569	180
4	10	50	70	3591	40
4	10	50	71	3642	40
7	7	100	36	3650	93
4	10	50	72	3693	40
4	10	50	73	3744	40
7	7	100	37	3752	93
5	20	200	19	3773	180
4	10	50	74	3795	40
4	10	50	75	3846	40
7	7	100	38	3854	93
...
5	20	200	41	8230	180
7	7	100	82	8310	93
7	7	100	83	8411	93
5	20	200	42	8432	180
7	7	100	84	8512	93
7	7	100	85	8613	93
...
5	20	200	96	19294	180
5	20	200	97	19495	180
5	20	200	98	19696	180
5	20	200	99	19897	180
5	20	200	100	20098	180

Table 8: The performance of 4 real-time periodic processes.

PID	proctype	clktimemilli (ms)	gross CPU usage (ms)	average waiting time (ms)
9	0	12517	2286	87.273504
8	0	12573	2288	86.991525
11	0	12614	2298	87.254237
10	0	12821	2291	87.583333

Table 9: The performance of 4 CPU-bound processes

3.2 Test 4 real-time periodic processes with 4 additional CPU-bound processes

1. The CPU-bound processes can only be running when all real-time processes are sleeping. Hence, the CPU-bound processes are very likely to terminate after most of real-time processes have finished.
2. Table 9 shows the performance of 4 CPU-bound processes in this situation. They all terminate after 3 real-time processes have finished.
3. The overall running time of all these 8 processes is the same as that of 4 real-time processes, since CPU-bound processes can be running when all real-time processes are sleeping.

4 Bonus problem

For EDF, the kernel has to dynamically track which real-time task has earliest deadline over time. Moreover, since the admission control of EDF is 1, the kernel must use interrupt disabling sparingly so that interrupts for real-time processes are not delayed.

First, we need to change the admission control in `rms_create()`. Second, we need to maintain a binary search tree to dynamically track which process has the earliest deadline more quickly. Third, since XINU system calls will disable interrupts at start and restore them only at the end, which may delay interrupt processing of real-time processes in case of slow system calls. Hence, we must modify each system call and make sure the interrupt disabling time as short as possible.