Project 1 Report

추성재 (201911185)

1. CFS + Extra Credit

CFS와 Extra Credit을 만족하기 위해 xv6에서 다음 사항들을 구현했다.

a. Red black tree

Red black tree는 리눅스 커널 v2.6.23 rbtree.c, rbtree.h에 구현되어 있는 것을 사용했다. 이때, xv6에서 구동할 수 있도록 일부를 변형했다. 해당 사항은 rbtree.c와 rbtree.h에 구현되어 있다.

b. proc.c, proc.h

proc.h에 정의된 변수들	proc->key : vruntime. proc_tree의 key가 된다. proc->time_slice : process가 할당받은 time_slice proc->node : proc_tree의 node proc->last_sched_time : 최근에 scheduling 되었을 때 시간 proc->nice_value : process의 priority level. priority[], nice_value_array[]의 index이다.
Global variables	sched_latency, min_granularity : 슬라이드에서 정의된 것과 동일 sched_tick : 마지막으로 time slice를 계산한 시간 curproc_num : 현재 몇 개의 process가 tree 안에 있는지를 나타냄 runnable_proc_num : process의 state가 RUNNING 또는 RUNNABLE한 process의 개수 cfs_prior priority[] : 각 priority 별 유지해야 하는 값들을 저장하는 array. Index가 priority level이다. nice_value_array[] : priority level별 nice_value를 저장 ptable.proc_tree : vruntime을 key로 하는 red black tree의 root
struct cfs_prior	각 priority level 별로 유지해야 하는 값들을 저장 cfs_prior->curproc_num : 각 priority level 별 총 process 개수 cfs_prior->runnable_proc_num : 각 priority level 별 runnable_proc_num
red-black tree 연산	proc_tree_insert : process의 priority를 이용해 process의 새로운 time_slice, vruntime을 할당하고, proc_tree에 넣는다. Vruntime은 기존 vruntime에, nice_value_array에서 들고온 process의 priority에 따른 nice value에다가 priority 별 time slice를 곱해서 구한다. proc_tree_erase : process를 tree에서 지운다.
pinit()	새로 추가된 global variable들을 초기화하는 코드를 추가
allocproc()	proc 속 새로 정의한 변수들을 초기화한 후 proc_tree에 넣는 코드를 추가
exit()	종료되는 process를 proc_tree에서 제거하는 코드를 추가
sleep(), wakeup1()	process가 잠들 때는 global curproc_num, runnable_proc_num과 process의 priority level의 curproc_num, runnable_proc_num을 줄이고, process가 깨어날 때는 반대로 늘리는 코드를 추가
yield()	process가 새로이 scheduling 되기 이전에 기존 request를 proc_tree에서 제거하고, 새로운 request를 넣는 코드를 추가

scheduler()	scheduler()에서는 proc_tree의 첫 번째 노드부터 시작해 RUNNABLE state에 있는 process 중 vruntime이 가장 작은 process를 선택한다. 그 후 proc->last_sched_time을 update한 뒤 context switch한다.
nice value 연산	process의 priority를 늘리고 줄이는 연산을 하는 __nice_value_up, __nice_value_down을 추가함. Process의 priority를 늘리고 줄임에 따라 priority[] 속 값들을 조정하고, priority 별 time_slice를 다시 계산한다.

c. trap.c

trap.c 안에는 매 sched_latency마다 priority 별 time_slice를 계산하는 코드를 추가했다. 이때, priority 별 time slice는 sched_latency를 priority 별 runnable_proc_num을 나눈 값이다.

또한, timer interrupt가 일어나도, process가 할당받은 time_slice만큼 실행된 이후(proc->last_sched_time + proc->time_slice < ticks)에 yield 될 수 있도록 yield()를 부르는 곳을 수정했다.

d. System-call implementation

nicevalueup, nicevaluedown 두 user function을 만들고, sys_nicevalueup, sys_nicevaluedown를 추가해 user level에서 __nice_value_up, __nice_value_down을 이용해 process의 priority를 조절할 수 있도록 만들었다.

System-call implementation은 mini-project 01에서 hostname을 조절하는 system-call을 만든 방법과 동일하게 구현했다.

다음은 구현된 CFS와 Round-Robin(RR)의 bench1부터 4까지 average response time, average turnaround time을 나타낸 표이다. 단위는 tick이다.

	bench1	bench2	bench3	bench4
avg response time(RR)	2	4	6	3
avg response time(CFS)	2	18	72	34
avg turnaround time(RR)	2	39	159	59
avg turnaround time(CFS)	2	22	88	49

RR의 경우, bench program 속 모든 process들이 bench program의 시작부에서 1 tick씩 돌아가며 실행되므로 min_granularity가 2인 CFS보다 average response time이 짧다. 하지만, RR의 경우에는 bench program 속 모든 process들의 종료되는 시점이 bench program이 끝나기 바로 직전에 몰려있다. 따라서 average turnaround time이 CFS보다 크다. Turnaround time 측면에서 보면 CFS가 RR보다 더 좋은 scheduling policy이다.

다음으로 extra credit인 priority에 따른 bench5의 결과 차이를 분석해보았다. 구현한 CFS에는 세 단계의 priority가 있다. 각각의 priority 별로 time_slice가 다르고, nice value 또한 다르다. 최상단 priority의 nice value는 1이고, priority가 낮아질수록 nice value의 값을 키워 vruntime이 다른 상위 priority를 가진 process보다 크도록 만들어, 결과적으로 priority가 낮은 process의 경우에는 상위 process들보다 tree 속에서 오른쪽에 위치하게 된다. 모든 process들은 생성되었을 때 가장 높은 priority를 가진다. 현재로서는 별도의 priority를 조정하는 policy는 구현하지 않았고, system call만 구현한 상태이다.

Priority에 따른 bench5의 결과 분석에 앞서서 현재 구현에서 time_slice의 특징을 살펴보고자 한다. 구현된 CFS에서는 timer interrupt로 인한 trap이 발생할 때마다, 마지막으로 time_slice를 계산한 뒤로 sched_latency 만큼의 시간이 지날 때마다 time slice를 계산한다. Context switching이 일어날 때마다 time slice를 계산하고 있지 않다. 이 구현 방식으로 인해 다음과 같이 동작한다.

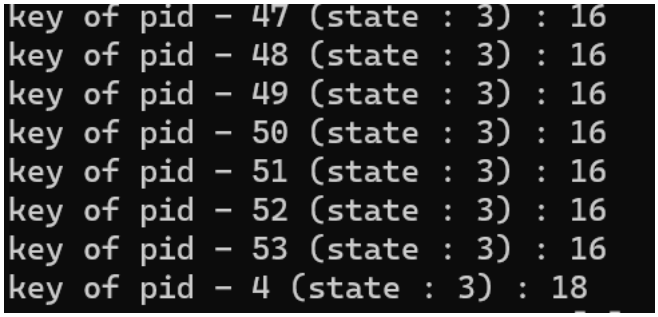
a. bench5를 실행하고, child process들을 fork하는 시점에서 time slice는, 이전에 shell만이 runnable process일 때 계산된 time slice인 16이다. 모든 fork된 child process들은 16의 time slice를 할당 받고, vruntime 또한 16을 가지고 proc_tree 안에 들어간다.

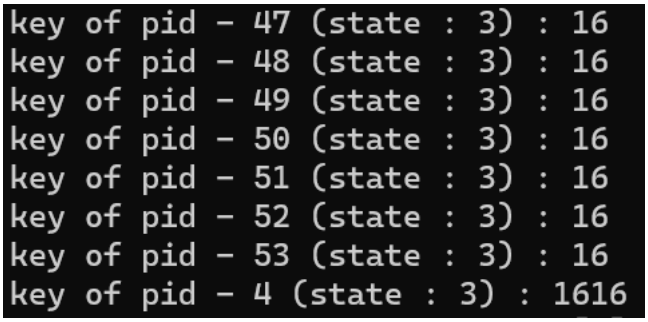
b. 이후 timer interrupt에서 sched_latency가 지난 후 time slice를 다시 계산하더라도, child process들은 새로 계산된 time slice와는 상관없이, 할당 받은 time slice 16만큼 실행된다. 후에 child process들이 timer interrupt에 의해 yield 된다면, 이때 새로 계산된 time slice를 할당 받는다.

c. 하지만 bench5의 경우, short task들의 실행시간이 최초 할당된 time slice 16보다 모두 작으므로, 새로 계산된 time slice를 한 번도 할당 받지 못하고 종료된다.

d. Long task의 경우에는 timer interrupt에 의해 yield 된 후 vruntime은 18이 된다. 이때, long task는 자연스럽게 다른 process보다 큰 vruntime을 가지게 되므로 별도의 extra credit 없이도 proc_tree의 제일 오른쪽에 위치하게 된다.

이로 인해, bench5에 nice value를 조정하는 system call을 추가하는 것 이외의 변형 없이는 extra credit의 여부와 상관없이 동일한 결과가 나온다. Debugging 과정에서 proc_tree의 값들을 보면 다음과 같다.





왼쪽 그림은 bench5를 extra credit이 없는 CFS로, 오른쪽 그림은 bench5를 extra credit이 포함된 CFS로 실행했을 때, long task(pid : 4)가 최초 실행된 직후의 proc_tree의 내용을 나타낸 것이다. 두 경우 모두 long task가 proc_tree의 최하단에 위치해 있다. $vruntime += nicevalue * timeslice$ 로 계산할 때, extra credit이 없는 경우는 nice value = 1, time slice = 2이므로 vruntime이 18이지만, extra credit이 있는 경우에는 priority가 한 단계 내려갔으므로 time slice = 16, nice value = 100이므로 vruntime이 1616이 되었다. 즉, system call로 인해 long task의 priority는 정상적으로 내려갔지만, 이전과 차이가 없다.

그래서 성능 비교를 위해 bench5의 short task들의 실행 시간을 sched_latency보다 크게 만들도록 MULTIPLE이라는 상수를 bench5에 추가했다. MULTIPLE은 bench5 속 cmp_time에 곱해지던 3이라는 상수를 대체하는 상수다.

다음 표는 MULTIPLE의 차이에 따른 마지막 short task의 turnaround time을 extra credit이 없는 CFS와, extra credit이 반영된 CFS별로 기록한 표이다. 단위는 tick이다.

Multiple	3	5	7	10
CFS(no extra)	318	528	729	1042
CFS(with extra)	319	522	717	1017

표의 결과를 보면, system call을 이용해 long task의 priority를 낮추면, short task들이 최대한 우선으로 할당되고, short task들이 모두 종료된 이후에 long task가 실행되기 때문에, 마지막 short task의 turnaround time이 priority를 모두 동일하게 유지한 경우보다 priority를 낮췄을 때 감소한 모습을 볼 수 있다. 이 차이는 multiple이 증가하면 더 커진다.

2. EEVDF

EEVDF의 구현을 위해 xv6에 다음 사항들을 구현했다.

a. Red-black tree

CFS에서 사용한 동일한 red-black tree를 사용했다

b. proc.c, proc.h

proc.h에 정의된 변수	proc->key : process의 virtual deadline proc->lag : process의 lag proc->vt_eligible : process의 virtual eligible time proc->weight : process의 weight proc->quantum : process의 quantum proc->used_time : process가 이때까지 할당 받은 quantum의 총합 proc->vt_init : process가 활성화(생성되거나 깨어날 때) 되었을 때 virtual time VT_MULTIPLE : VirtualTime의 소수점을 없애기 위해 곱하는 값. 10으로 설정되어 있다.
global variable	VirtualTime : virtual time. 소수점이 생기는 것을 방지하기 위해, VirtualTime과, Virtual Time을 이용하는 변수들에는 VT_MULTIPLE(10)이 곱해져있다. TotalWeight : 현재 RUNNABLE한 process들의 weight 합
red-black tree 연산	proc_tree_insert : 별도의 proc->key 연산 없이 virtual deadline을 key 값으로 proc_tree에 process를 넣는다. proc_tree_erase : 별도의 proc->key 연산 없이 process를 proc_tree에서 제거한다.
pinit()	새로 추가된 global variable을 초기화한다.
allocproc()	proc 속 새로 추가된 변수들을 초기화하고, eevdf_join을 부른 다음 생성된 process를 proc_tree에 넣는다.
exit()	eevdf_leave를 부른 다음, 종료되는 process를 proc_tree에서 제거한다.
sleep(), wakeup1()	process가 sleep에 들어가면 eevdf_leave를 부르고, process가 깨어나면 eevdf_join을 부른다.
yield()	process가 새로이 scheduling 되기 이전에 기존 request를 proc_tree에서 제거하고, 새로운 request를 proc_tree에 넣는다
scheduler()	scheduler()에서는 proc_tree의 첫 번째 process 부터 시작해 다음을 수행한다. 1. 만약 process의 virtual eligible time이 현재 virtual time보다 작다면, 이 process는 eligible하므로 p->lag를 0으로 초기화한다. 2. 만약, 이 process가 RUNNABLE하지 않거나, lag가 0보다 작다면 (eligible 하지 않은 process), proc_tree 속 다음 process를 선택해 1로 돌아간다. 3. 선택된 process의 p->used_time, p->last_sched_time을 update한다. 4. p->lag를 update한다. 이때, update에는 다음 수식을 사용한다. $lag = weight_p * (VirtualTime - VirtualTime_{\text{init}}) - VTMULTIPLE * used_p$ 이때, VirtualTime과 p->vt_init이 같다면, lag를 update하지 않는다.

	이는 해당 process가 처음으로 scheduling 되자마자 lag가 음수가 되어 scheduler가 불필요한 spin을 하는 것을 막기 위해 해당 조건을 넣었다. 5. 이후 p->vt_eligible에 ((p->used_time * VT_MULTIPLE) / p->weight)를 더하고, p->key를 p->vt_eligible + (p->quantum/p->weight)*VT_MULTIPLE로 update한다. 6. 이후 process로 context switch한다. proc_tree 속 옛날 request를 없애고 계산된 값들을 이용한 새로운 request를 proc_tree에 집어넣는 과정은 모두 yield()에서 수행된다.
eevdf_join	eevdf_join의 과정은 논문의 appendix A에 있는 pseudo code를 참고해 작성했다. 다음 과정을 수행한다. 1. TotalWeight에 join하고자 하는 process의 weight을 더한다. 2. Process의 used time을 0으로 초기화한다 3. VirtualTime을 논문에서 나온 식 VirtualTime = VirtualTime – (p->lag / TotalWeight)으로 update한다. 4. P->vt_init을 VirtualTime으로 초기화한다. 5. p->vt_eligieble = VirtualTime으로(process의 used time이 0이기에), p->key를 p->vt_eligible + (p->quantum / p->weight)*VT_MULTIPLE로 초기화한다.
eevdf_leave	eevdf_leave의 과정은 논문의 appendix A에 있는 pseudo code를 참고해 작성했다. 다음 과정을 수행한다 1. TotalWeight에서 leave하고자 하는 process의 weight을 뺀다. 2. VirtualTime을 논문에서 나온 식 VirtualTime + (p->lag / TotalWeight)을 이용해 update한다.

c. trap.c

trap.c에서는 timer interrupt가 일어날 때마다 TotalWeight이 0보다 큰 경우에 VirtualTime을 VirtualTime = VirtualTime + (VT_MULTIPLE / TotalWeight)로 update한다.

또한, process가 p->quantum만큼 실행되는 것을 보장해주기 위해, timer interrupt가 일어났을 때, p->last_sched_time에 p->quantum을 더한 값이 현재 ticks보다 작다면, 이때 yield 되도록 만들었다.

다음은 구현된 EEVDF와 Round-Robin(RR)의 bench1부터 4까지 average response time, average turnaround time을 나타낸 표이다. 단위는 tick이다.

	bench1	bench2	bench3	bench4
avg response time(RR)	2	4	6	3
avg response time(EEVDF)	1	4	4	1
avg turnarround time(RR)	2	39	159	59
avg turnarround time(EEVDF)	2	52	172	107

Average response time에서는 RR과 EEVDF가 큰 차이를 보이지 않는다. 하지만 average turnaround time에서는 RR보다 EEVDF의 성능이 크게 떨어진다. 이 현상은 다음 차이 때문에 발생한 것으로 보인다.

a. Process의 quantum도 1로 고정되어 있고, process의 weight도 1로 고정되어 있어서, 매 timer interrupt마다 yield()가 불리게 된다. 이는 xv6의 RR과 동일한 환경이다. 이때, RR과 달리, 구현된 EEVDF에서는 yield할 때 이전의 request를 proc_tree에서 빼내고, 새로운 request를 proc_tree에 넣는 과정이 존재한다. 이 과정 때문에 RR과 quantum이 동일하더라도 turnaround time이 느릴 수 있다.

b. 현재 구현에서는 process의 lag를 process가 schedule 된 이후에 update를 한다. 이때, 한 번 lag가 음수가 된 process는 다시는 lag의 update가 이루어지지 않는다. 이 process가 다시 eligible process가 될 때는 자신의 virtual eligible time이 현재 virtual time보다 작을 때이다. 현재 구현에서는 tree 속에 lag의 부호와는 상관없이 process들이 진입하고, tree에서 완전히 나올 때는 process가 완전히 종료될 때이다. 즉, 때로는 proc_tree의 모든 process들의 lag가 음수가 되는 경우가 있다. 이때는 어느 한 process의 virtual eligible time이 현재 virtual time보다 작아질 때까지 scheduler는 spin한다. 이 현상이 발생하면 process들의 실행 시간 자체가 늘어나게 되므로 EEVDF의 average turnaround time이 RR에 비해 커질 수 있다.