

OS Project01_WIKI_2022058740 양경혁

Design :

이번 프로젝트의 목적은 xv6 운영체제에 원래 기존의 RR방식의 스케줄러를 **FCFS(First-Come-First-Serve)** 스케줄러와 **MLFQ(Multi-Level Feedback Queue)** 스케줄러로 변경하고, 사용자가 시스템 콜을 통해 동적으로 스케줄링 정책을 전환할 수 있도록 만드는 것이다. 또한, MLFQ의 우선순위 상승(Priority Boosting), 큐 레벨 별 time quantum, trap을 통한 tick 관리를 정확히 수행해야 한다.

Implement :

```
454 void
455 scheduler(void)
456 {
457     struct proc *p;
458     struct cpu *c = mycpu();
459
460     c->proc = 0;
461     for(;;){
462         intr_on();
463
464         if (scheduler_mode == FCFS_MODE) {
465             struct proc *earliest = 0;
466             for(p = proc; p < &proc[NPROC]; p++) {
467                 acquire(&p->lock);
468                 if (p->state == RUNNABLE) {
469                     if (earliest == 0 || p->ctime < earliest->ctime) {
470                         if (earliest)
471                             release(&earliest->lock);
472                         earliest = p;
473                     } else {
474                         release(&p->lock);
475                     }
476                 } else {
477                     release(&p->lock);
478                 }
479             }
480
481             if (earliest) {
482                 earliest->state = RUNNING;
483                 c->proc = earliest;
484                 swtch(&c->context, &earliest->context);
485                 c->proc = 0;
486                 release(&earliest->lock);
487             } else {
488                 intr_on();
489                 asm volatile("wfi");
490             }
491         }
492     }
493 }
```

위 코드는 kernel 내의 proc.c 파일에서 scheduler() 함수의 기존 RR 방식을 수정하여 FCFS 스케줄링을 구현한 것이다. FCFS 스케줄링을 구현하기 위해, scheduler() 함수 내에서 scheduler_mode가 FCFS_MODE일 경우, 프로세스 생성 시각(ctime)을 기준으로 가장 먼저 생성된 RUNNABLE 상태의 프로세스를 선택하여 실행한다.

함수의 핵심 로직은 이렇다.

struct proc *earliest 변수를 통해 가장 먼저 생성된 RUNNABLE 프로세스를 탐색

모든 프로세스를 순회하며 p->ctime이 가장 작은 프로세스를 찾음

이전에 선택한 프로세스가 있다면 해당 lock을 해제

최종적으로 선택된 프로세스를 RUNNING 상태로 전환하고, swtch()로 컨텍스트 전환 수행

```

492     else if (scheduler_mode == MLFQ_MODE) {
493         if (tick_count >= 50) {
494             acquire(&ptable_lock);
495             for (p = proc; p < &proc[NPROC]; p++) {
496                 acquire(&p->lock);
497                 if (p->state != UNUSED) {
498                     p->level = 0;
499                     p->priority = 3;
500                     p->time_quantum = 0;
501                 }
502                 release(&p->lock);
503             }
504             tick_count = 0;
505             release(&ptable_lock);
506         }
507         struct proc *selected = 0;
508         for (int level = 0; level <= 2; level++) {
509             for (p = proc; p < &proc[NPROC]; p++) {
510                 acquire(&p->lock);
511                 if (p->state == RUNNABLE && p->level == level) {
512                     selected = p;
513                     break;
514                 }
515                 release(&p->lock);
516             }
517             if (selected)
518                 break;
519         }
520         if (selected) {
521             selected->state = RUNNING;
522             c->proc = selected;
523             swtch(&c->context, &selected->context);
524             c->proc = 0;
525             if (selected->state == RUNNABLE) {
526                 selected->time_quantum++;
527                 if (selected->level == 0 && selected->time_quantum >= 1) {
528                     selected->level = 1;
529                     selected->time_quantum = 0;
530                 }
531                 else if (selected->level == 1 && selected->time_quantum >= 3) {
532                     selected->level = 2;
533                     selected->time_quantum = 0;
534                 }
535                 else if (selected->level == 2 && selected->time_quantum >= 5) {
536                     selected->time_quantum = 0;
537                 }
538             }
539             release(&selected->lock);
540         } else {
541             intr_on();
542             asm volatile("wfi");
543         }
544     }
545 }
546 }
547 }
548

```

MLFQ 스케줄링은 프로세스의 실행 빈도 및 양에 따라 우선순위를 동적으로 조정하는 방식으로, 3단계 큐(L0, L1, L2)를 사용하며, 각 큐마다 time quantum이 다르게 설정되어 있다. 본 코드는 scheduler_mode == MLFQ_MODE일 때 동작한다. 아래는 해당 코드의 주요 기능이다.

Priority Boosting:

글로벌 변수인 tick_count가 50 이상일 때 모든 프로세스를 L0으로 복귀시키고, time quantum과 priority를 초기화한다.

Queue Traversal:

L0 → L1 → L2 순서로 각 큐를 순회하며, 가장 먼저 발견된 RUNNABLE 상태의 프로세스를 선택

Time Quantum 기반 레벨 변화:

L0: 1 tick 초과 시 L1으로

L1: 3 tick 초과 시 L2로

L2: 5 tick 초과 시 quantum만 초기화 (최하위 큐)

```

84 extern struct spinlock ptable_lock;
85
86 // Per-process state
87 struct proc {
88     struct spinlock lock;
89
90     // p->lock must be held when using these:
91     enum procstate state;          // Process state
92     void *chan;                  // If non-zero, sleeping on chan
93     int killed;                  // If non-zero, have been killed
94     int xstate;                  // Exit status to be returned to parent's wait
95     int pid;                      // Process ID
96     uint64 ctime;                // FCFS 스케줄링을 위한 생성 시작 (creation time)
97     int priority;                // 프로세스 우선순위 (0~3)
98     int level;                   // MLFQ 큐 레벨 (0, 1, 2)
99     int time_quantum;            // 현재 사용한 시간 퀀텀

```

필드 이름 설명

state 프로세스의 현재 상태 (RUNNABLE, RUNNING, SLEEPING, 등) → 스케줄링 대상인지 확인할 때 사용됨

ctime 프로세스가 생성된 시간 (tick 기준) → FCFS 모드에서 가장 먼저 생성된 프로세스를 고르기 위해 사용

level 현재 프로세스가 속한 MLFQ 큐의 레벨 (0, 1, 2) → MLFQ 모드에서 레벨별 우선순위 확인에 사용

필드 이름	설명
time_quantum	해당 레벨에서 소비한 시간 양 → 정해진 quantum을 초과했는지 판단하여 레벨을 바꾸는데 사용
priority	MLFQ 동작 중 tick_count >= 50일 때 모든 프로세스를 초기화하는 우선순위 기준으로 사용
lock	해당 프로세스의 동기화를 위한 spinlock → 스케줄링 중 다른 CPU와의 경합 방지를 위해 acquire() / release()로 감싸 사용

```

109 uint64
110 sys_yield(void)
111 {
112     yield();
113     return 0;
}

116 uint64
117 sys_getlev(void)
118 {
119     struct proc *p = myproc();
120     if (scheduler_mode == FCFS_MODE)
121         return 99;
122     return p->level;
123 }

125 uint64
126 sys_setpriority(void)
127 {
128     int pid, priority;
129     struct proc *p;
130
131     argint(0, &pid);
132     argint(1, &priority);
133
134     if (priority < 0 || priority > 3)
135         return -2;
136
137     acquire(&ptable_lock);
138     for (p = proc; p < &proc[NPROC]; p++) {
139         if (p->pid == pid) {
140             p->priority = priority;
141             release(&ptable_lock);
142             return 0;
143         }
144     }
145     release(&ptable_lock);
146
147     return -1;
148 }

106 extern uint64 sys_yield(void);
107 extern uint64 sys_getlev(void);
108 extern uint64 sys_setpriority(void);
109 extern uint64 sys_mlfqmode(void);
110 extern uint64 sys_fcfsmode(void);

```

25 #define SYS_yield 24 138 [SYS_yield] sys_yield,
 26 #define SYS_getlev 25 139 [SYS_getlev] sys_getlev,
 27 #define SYS_setpriority 26 140 [SYS_setpriority] sys_setpriority,
 28 #define SYS_mlfqmode 27 141 [SYS_mlfqmode] sys_mlfqmode,
 29 #define SYS_fcfsmode 28 142 [SYS_fcfsmode] sys_fcfsmode,

kernel 내의 syscall.c, syscall.h파일의 변경사항 (정상적인 시스템콜을 위함)

```

176 uint64
177 sys_fcfsmode(void)
178 {
179     if (scheduler_mode == FCFS_MODE) {
180         return -1;
181     }
182
183     acquire(&ptable_lock);
184     scheduler_mode = FCFS_MODE;
185
186     for (struct proc *p = proc; p < &proc[NPROC]; p++) {
187         if (p->state != UNUSED) {
188             p->priority = -1;
189             p->level = -1;
190             p->time_quantum = -1;
191         }
192     }
193     tick_count = 0;
194     release(&ptable_lock);
195     return 0;
196 }


```

이 코드는 현재 스케줄링을
 FCFS로 전환하는 시스템
 콜이다.
 test.c나 사용자
 영역프로그램에서
 fcfsmode() 호출 시
 내부적으로 커널이 이
 함수로 진입한다. FCFS
 모드이므로, MLFQ모드에서
 사용되는 변수들은 -1로
 초기화한다.

```

150 uint64
151 sys_mlfqmode(void)
152 {
153     if (scheduler_mode == MLFQ_MODE) {
154         return -1;
155     }
156
157     acquire(&ptable_lock);
158
159     printf("CPU %d: Switching to MLFQ mode\n", cpuid());
160
161     scheduler_mode = MLFQ_MODE;
162
163     for (struct proc *p = proc; p < &proc[NPROC]; p++) {
164         if (p->state != UNUSED) {
165             p->priority = 3;
166             p->level = 0;
167             p->time_quantum = 0;
168         }
169     }
170     tick_count = 0;
171     release(&ptable_lock);
172
173     return 0;
174 }

```

이 코드는 현재 스케줄링을 MLFQ로 전환하는 시스템 콜이다.

사용자 프로그램에서 mlfqmode() 호출 시 커널 내부적으로 커널이 이 함수로 진입한다.

또한 모드전환 시점에 처음 들어온 프로세스에 대하여 priority는 3, level은 0으로 설정한다.

```

136 void
137 kerneltrap()
138 {
139     int which_dev = 0;
140     uint64 sepc = r_sepcc();
141     uint64 sstatus = r_sstatus();
142     uint64 scause = r_scause();
143
144     if((sstatus & SSTATUS_SPP) == 0)
145         panic("kerneltrap: not from supervisor mode");
146     if(intr_get() != 0)
147         panic("kerneltrap: interrupts enabled");
148
149     if((which_dev = devintr()) == 0){
150         // interrupt or trap from an unknown source
151         printf("scause=0x%lx sepc=0x%lx stval=0x%lx\n", scause, r_sepcc(), r_stval());
152         panic("kerneltrap");
153     }
154
155     // give up the CPU if this is a timer interrupt.
156     if(which_dev == 2 && myproc() != 0)
157         yield();
158
159     if (which_dev == 2)
160         clockintr();
161     // the yield() may have caused some traps to occur,
162     // so restore trap registers for use by kernelvec.S's sepcc instruction.
163     w_sepcc(sepc);
164     w_sstatus(sstatus);
165 }

```

kerneltrap()은 타이머 인터럽트를 포함한 다양한 커널 인터럽트를 처리하는 함수이다.

기존 xv6에서는 타이머 인터럽트 발생 시 단순히 yield()를 호출하여 CPU 양보만 했음.

```

167 void
168 clockintr()
169 {
170     if(cpuid() == 0){
171         acquire(&tickslock);
172         ticks++;
173         tick_count++;
174         wakeup(&ticks);
175         release(&tickslock);
176     }
177
178     // ask for the next timer interrupt. this also clears
179     // the interrupt request. 1000000 is about a tenth
180     // of a second.
181     w_stimecmp(r_time() + 1000000);
182 }

```

하지만 MLFQ 구현이나 tick 기반 스케줄링 기능을 추가하려면, 전역 tick 처리와 priority boosting 로직이 매 tick마다 실행되어야 하므로, 전역변수 tick_count를 시간에 따라 증가도록 할 수 있는 clockintr()에 tick_count++; 을 추가하였음.

```
27 int yield(void);
28 int getlev(void);
29 int setpriority(int pid, int priority);
30 int mlfqmode(void);
31 int fcfsmode(void);
```

```
41 entry("yield");
42 entry("getlev");
43 entry("setpriority");
44 entry("mlfqmode");
45 entry("fcfsmode");
```

유저 프로그램에서 이번 프로젝트의 기능을 가능케하는 함수 호출이 가능하도록, 또 그 유저 함수가 시스템 콜 트랩으로 연결되게 하는 user.h 와 usys.pl에 추가한 코드이다.

Result :

```
xv6 kernel is booting
init: starting sh
[$ test
FCFS & MLFQ test start

[Test 1] FCFS Queue Execution Order
Process 4 executed 100000 times
Process 5 executed 100000 times
Process 6 executed 100000 times
Process 7 executed 100000 times
[Test 1] FCFS Test Finished

nothing has been changed
CPU 0: Switching to MLFQ mode
successfully changed to MLFQ mode!

[Test 2] MLFQ Scheduling
Process 8 (MLFQ L0-L2 hit count):
L0: 3779
L1: 21918
L2: 74303
Process 9 (MLFQ L0-L2 hit count):
L0: 11415
L1: 32229
L2: 56356
Process 10 (MLFQ L0-L2 hit count):
L0: 14530
L1: 44504
L2: 40966
Process 11 (MLFQ L0-L2 hit count):
L0: 14496
L1: 45462
L2: 40042
[Test 2] MLFQ Test Finished

FCFS & MLFQ test completed!
$
```

[Test 1] FCFS 테스트 : 각 프로세스가 100,000번 루프를 정확히 수행하며, FCFS 스케줄링의 순차적 실행이 확인되었음

[Test 2] MLFQ 테스트 : 각 프로세스가 다양한 큐(L0~L2)를 통해 실행되며, priority boosting, time quantum에 따른 정상적인 level 이동이 반영된 스케줄링 결과를 출력함

Compile 흐름

test 명령어 실행 시, FCFS와 MLFQ 스케줄러 동작을 테스트하는 test.c 실행

fork_children() 함수를 통해 자식 프로세스들을 생성하고, 각각이 일정량의 연산을 수행하며 실행 순서를 기록

FCFS 모드에서는 프로세스 생성 시점인 ctime을 기준으로 가장 먼저 생성된 프로세스부터 실행

모든 자식 프로세스가 동일한 횟수로 작업을 완료한 뒤 종료되며, 실행 순서는 고정

mlfqmode()를 호출하면 CPU 0: Switching to MLFQ mode 메시지와 함께 스케줄링 방식이 MLFQ로 변경

이 과정에서 모든 프로세스의 priority, level, time_quantum이 초기화
다시 자식 프로세스들을 fork_children()을 통해 생성하고, 각 프로세스가 자신의 실행 큐(L0~L2)에서 어느 정도의 tick을 소비했는지 출력

마지막으로 결과를 보면, 프로세스가 처음에는 L0에서 시작하고, time quantum 초과 시 L1 또는 L2로 이동하며 글로벌 tick이 50이 되면 모든 프로세스가 priority boosting에 따라 L0으로 복귀하는 흐름이 구현되었음.

Troubleshooting :

프로젝트 중간에 전역변수를 선언하는 것에 있어 난항을 겪었지만(ex ptable_lock), 여러번 코드를 복기하고 컴파일 흐름을 분석하며 해결했음.