컴퓨터학부 20172642 신형담

1-1. 구현개요

리눅스 커널의 스케줄링 정책을 알아볼 수 있는 CFS.c와 CFSnice.c 프로그램을 구현했다. 또한 usr/src/linux1/linux-5.11.22/kernel/sched에 존재하는 core.c파일에 코드를 추가하여 리눅스 커널의 default 스케줄러를 SCHED_FIFO로 변경하였다. 마지막으로 cpu burst time을 출력하기 위해 usr/src/linux1/linux-5.11.22/kernel/sched에 존재하는 core.c, stats.h 파일과 usr/src/linux/linux-5.11.22/include/linux에 존재하는 sched.h파일에 코드를 추가하였다.

제출물 코드는 CFS.c, CFSnice.c, core_for_fifo.c, core.c, stats.h, sched.h로 구성되어 있으며 core_for_fifo.c는 스케줄러를 SCHED_FIFO로 변경할 때 수정한 core.c코드이고 core.c, stats.h, sched.h는 모두 cpu burst time출력을 위해 수정한 코드이다.

1-2. 구현과정

- chrt 명령어

```
shin@shin-virtual-machine:~$ chrt -v -p 1
pid 1's current scheduling policy: SCHED_OTHER
pid 1's current scheduling priority: 0
```

chrt명령어를 통해 현재 스케줄링 정책을 확인했다. SCHED_OTHER, 즉 CFS(Completely Fair Scheduler)로 설정되어 있는 것을 확인할 수 있었다.

다음으로 CFS를 확인하기 위한 프로그램을 구현 하였다.

- CFS.c

```c
void matrix_cal(int);

int main() {
    pid_t pid[21];
    int status;

    for(int i = 0; i < 21; i++) {
        if(pid[i] = fork() > 0) {
        }
        else if(pid[i] == 0) {
            printf(" %d process begins\n", getpid());

            if(i < 7) {
                matrix_cal(10000);
            }
            else if(i < 14) {
                matrix_cal(50000);
            }
            else if(i < 21) {
                matrix_cal(100000);
            }
            exit(0);
        }
    }

    for(int i = 0; i < 21; i++) {
        pid_t end_pid = wait(&status);
        printf(" %d process ends\n", end_pid);
    }
    printf("----------- All processes end -----------\n");
    return 0;
}
void matrix_cal(int size) {
    int** arr;
    arr = (int**)malloc(sizeof(int*) * size);
    for(int i = 0; i < size; i++) {
        arr[i] = (int*)malloc(sizeof(int) * size);
    }

    for(int i = 0; i < size; i++) {
        for(int j = 0; j < size; j++)
            arr[i][j] = i + j;
    }

    for(int i = 0; i < size; i++) {
        for(int j = 0; j < size; j++)
            arr[i][j] += 10;
    }
}
```

매우 간단한 프로그램으로 main에서 fork를 21번 수행하여 자식 프로세스를 생성하고 자식 프로세스들은 printf(" %d process begins₩n", getpid());를 통해 자신이 실행되었음을 표준 출력에 출력하고 1~7번째로 생성된 그룹은 matrix_cal(10000); ,8~14번째로 생성된 그룹은 matrix_cal(50000); , 15~21번째로 생성된 그룹은 matrix_cal(100000); 을 수행하게 된다. matrix_cal() 함수는 int형 인자 size를 받아 size * size 사이즈의 int형 2차원 배열 arr을 동적할당하여 값을 arr[i][j]를 i + j로 초기화 한 후 다시 각각의 원소에 10을 더하는 작업을 수행한다. 프로세스의 시작과 종료를 확인하기 위한 함수로 특별한 의미는 없다. 부모 프로세스는 wait()함수를 통해 자식 프로세스들의 종료를 기다리며 하나의 자식 프로세스가 종료될 때 마다 print(" %d process ends₩n", end_pid);를 통해 표준 출력에 자식 프로세스가 종료되었음을 출력한다.

- CFSnice

```c
void matrix_cal(int);

int main() {
    pid_t pid[21];
    int status;

    for(int i = 0; i < 21; i++) {
        if(pid[i] = fork() > 0) {
        }
        else if(pid[i] == 0) {
            printf(" %d process begins\n", getpid());

            if(i < 7) {
                nice(19);
                matrix_cal(20000);
            }
            else if(i < 14) {
                nice(0);
                matrix_cal(40000);
            }
            else if(i < 21) {
                nice(-20);
                matrix_cal(100000);
            }
            exit(0);
        }
    }

    for(int i = 0; i < 21; i++) {
        pid_t end_pid = wait(&status);
        printf(" %d process ends\n", end_pid);
    }
    printf("----------- All processes end -----------\n");
    return 0;
}
void matrix_cal(int size) {
    int** arr;
    arr = (int**)malloc(sizeof(int*) * size);
    for(int i = 0; i < size; i++) {
        arr[i] = (int*)malloc(sizeof(int) * size);
    }

    for(int i = 0; i < size; i++) {
        for(int j = 0; j < size; j++)
            arr[i][j] = i + j;
    }

    for(int i = 0; i < size; i++) {
        for(int j = 0; j < size; j++)
            arr[i][j] += 10;
    }
}
```

역시 매우 간단한 프로그램이며 앞서 설명한 CFS.c와 거의 동일하나 생성되는 자식 프로세스의 그룹에 따라 nice() 함수를 통해 nice값을 지정하여 준다. i < 7인 경우 는 nice 값을 19로, i < 14인 경우 0, i < 21인 경우 -20으로 지정한 뒤 우선순위가 낮은 그룹일수록 수행할 연산의 크기를 작게하여 nice값에 의한 스케줄링의 변화가 최대한 반영되도록 구현했다.

다음으로 Core.c 의 코드를 수정하여 리눅스 커널의 default 스케줄러를 SCHED_FIFO로 수정하였다. 구글링을 통해 해당 코드를 발견할 수 있었다.

- 1단계: usr/src/linux1/linux-5.11.22/kernel/sched/core.c

```
if(p->policy == SCHED_NORMAL) {
        p->prio = current->normal_prio - NICE_WIDTH - PRIO_TO_NICE(current->static_prio);
        p->normal_prio = p->prio;
        p->rt_priority = p->prio;
        p->policy = SCHED_FIFO;
        p->static_prio = NICE_TO_PRIO(0);
}
```

core.c에 정의되어있는 sched_fork()함수에 해당 코드를 추가했다. 또한 해당 라인쯤에 원래 있던 p->prio = current->normal_prio; 코드를 주석처리하였다.

- 2단계: usr/src/linux1/linux-5.11.22/kernel/sched/core.c

```
static int _sched_setscheduler(struct task_struct *p, int policy,
                               const struct sched_param *param, bool check)
{
        struct sched_attr attr = {
                .sched_policy   = policy,
                .sched_priority = param->sched_priority,
                .sched_nice     = PRIO_TO_NICE(p->static_prio),
        };

        if(attr.sched_policy == SCHED_NORMAL) {
               attr.sched_priority = param->shced_priority - NICE_WIDTH - attr.sched_nice;
               attr.sched_policy = SCHED_FIFO;
        }
```

core.c에 정의되어있는 _sched_setcheduler()함수에 해당 if문을 추가 했다.
이후 해당 커널 이미지를 다시 컴파일 하여 그 결과를 확인했다.

마지막으로 수정하기 전 core.c파일로 다시 커널 컴파일을 진행하여 리눅스 커널을 원상태로 복구한 뒤 cpu burst time 출력을 위한 작업을 진행하였다.

- 1단계: usr/src/linux1/linux-5.11.22/include/sched.h

```
#endif

        unsigned long long cpu_burst_time;
        /*
         * New fields for task_struct should be added above here, so that
         * they are included in the randomized portion of task_struct.
         */
        randomized_struct_fields_end

        /* CPU-specific state of this task: */
        struct thread_struct            thread;
```

task_struct 구조체의 멤버 선언부에 unsigned long long 타입의 cpu_burst_time 변수를 선언했다.

- 2단계: usr/src/linux1/linux-5.11.22/kernel/sched/core.c

```
int sched_fork(unsigned long clone_flags, struct task_struct *p)
{
        unsigned long flags;

        __sched_fork(clone_flags, p);
        /*
         * We mark the process as NEW here. This guarantees that
         * nobody will actually run it, and a signal or other external
         * event cannot wake it up and insert it on the runqueue either.
         */
        p->state = TASK_NEW;
        p->cpu_burst_time = 0;
```

sched_fork() 함수에서 cpu_burst_time을 0으로 초기화하는 코드를 추가했다.

- 3단계: usr/src/linux1/linux-5.11.22/kernel/sched/stats.h

```
static inline void sched_info_depart(struct rq *rq, struct task_struct *t)
{
        unsigned long long delta = rq_clock(rq) - t->sched_info.last_arrival;

        t->cpu_burst_time += delta;
        rq_sched_info_depart(rq, delta);

        if (t->state == TASK_RUNNING)
                sched_info_queued(rq, t);
}
```

cpu 스케줄러는 스케줄러 정책에 따라 여러 개의 프로세스들을 context switching 하면서 작업을 수행한다. 이때 각각의 프로세스가 cpu를 할당 받을 때 마다 시간을 측정하여 누적하는 것이 필요하다. sched_info_depart() 함수에서 delta 값을 cpu_burst_time에 더해주는 코드를 추가했다.

- 4단계: usr/src/linux1/linux-5.11.22/kernel/sched/core.c

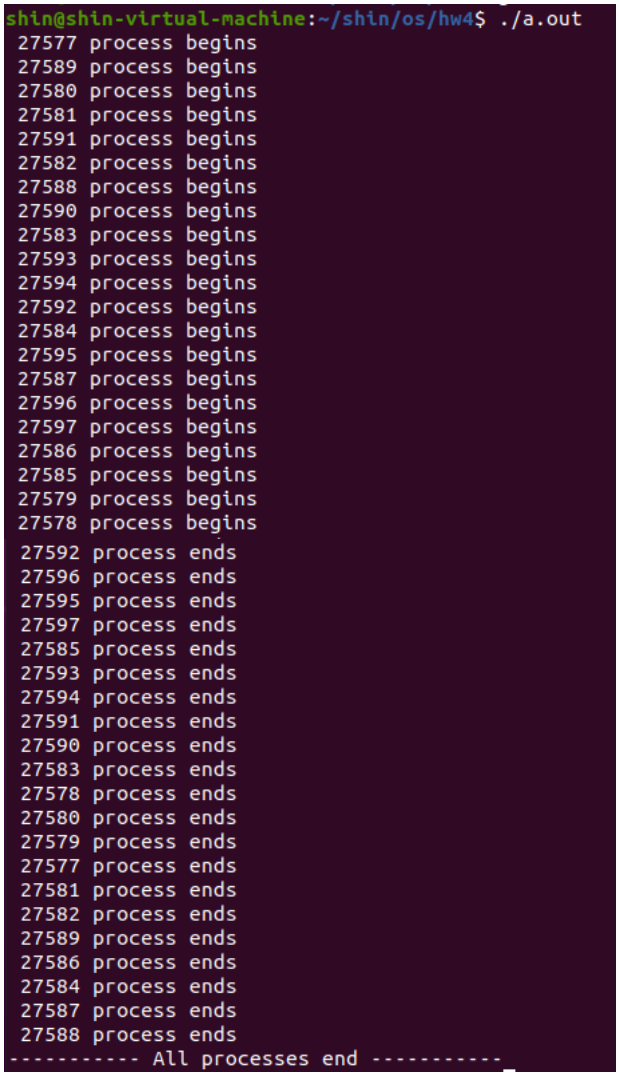
```
if (unlikely(prev_state == TASK_DEAD)) {
        printk("pid: %d's cpu_burst_time = %llu\n", prev->pid, prev->cpu_burst_time);
        if (prev->sched_class->task_dead)
                prev->sched_class->task_dead(prev);

        /*
         * Remove function-return probe instances associated with this
         * task and put them back on the free list.
         */
        kprobe_flush_task(prev);
```

계산된 cpu_burst_time을 출력하기 위해 프로세스가 작업을 끝낼 때 호출되는 finish_task_switch 함수에서 printk 함수를 통해 커널 로그에 cpu_burst_time을 출력하도록 했다.

1-3. 실행결과

- CFS.c



리눅스 커널이 CFS를 통해 21개의 프로세스를 생성한 모습이다. nice값을 지정해 주지 않았기 때문에 특별한 규칙을 보이진 않는다.

- CFSnice.c

```
shin@shin-virtual-machine:~/shin/os/hw4$ sudo ./a.out
 27948 process begins
 27949 process begins
 27967 process begins
 27950 process begins
 27951 process begins
 27952 process begins
 27953 process begins
 27954 process begins
 27955 process begins
 27956 process begins
 27957 process begins
 27958 process begins
 27966 process begins
 27959 process begins
 27960 process begins
 27961 process begins
 27962 process begins
 27963 process begins
 27965 process begins
 27964 process begins
 27947 process begins
 27965 process ends
 27967 process ends
 27963 process ends
 27961 process ends
 27964 process ends
 27966 process ends
 27962 process ends
 27957 process ends
 27954 process ends
 27959 process ends
 27956 process ends
 27958 process ends
 27960 process ends
 27955 process ends
 27952 process ends
 27951 process ends
 27953 process ends
 27949 process ends
 27948 process ends
 27947 process ends
 27950 process ends
---------- All processes end ----------
```

nice값을 지정해준 CFS의 결과이다. 프로세스의 시작이 nice값에의한 그룹별로 나뉘지는 않았지만 프로세스의 종료부분을 보면 알 수 있듯이 27961~27967(가장 많은 연산을 수행하고 nice값이 작은 그룹)이 먼저 종료되고, 27954~27960(중간의 연산과 nice값이 중간인 그룹)이 뒤이어 종료되고, 27947~27953(가장 작은 연산과 nice값이 큰 그룹)이 마지막으로 종료됨을 확인할 수 있었다.

- 커널 컴파일 이후 chrt 명령어 수행

```
shin@shin-virtual-machine:~$ chrt -v -p 1
pid 1's current scheduling policy: SCHED_FIFO
pid 1's current scheduling priority: 0
```

정상적으로 리눅스 커널의 default 스케줄러가 SCHED_FIFO로 변경되었다.

- cpu burst time 출력

```
[  316.361185] pid: 812's cpu_burst_time = 257171
[  316.361195] pid: 7's cpu_burst_time = 6261599
[  322.900124] pid: 1083's cpu_burst_time = 7376728
[  322.900214] pid: 1079's cpu_burst_time = 77856357
[  322.907753] pid: 1080's cpu_burst_time = 1067776
[  349.124534] pid: 5's cpu_burst_time = 261570
[  349.430756] pid: 2030's cpu_burst_time = 52696623
[  363.917057] pid: 2039's cpu_burst_time = 24799706
[  363.923484] pid: 2040's cpu_burst_time = 4498687
[  363.940369] pid: 2042's cpu_burst_time = 14338767
[  363.940964] pid: 2041's cpu_burst_time = 1488002
[  363.941454] pid: 2038's cpu_burst_time = 2173341
[  365.112608] pid: 2036's cpu_burst_time = 765913
[  367.370145] pid: 2048's cpu_burst_time = 7789631
[  367.379557] pid: 2044's cpu_burst_time = 7199964
[  367.391630] pid: 2050's cpu_burst_time = 7299169
[  367.410690] pid: 2049's cpu_burst_time = 7539152
[  367.412463] pid: 2046's cpu_burst_time = 7052156
[  367.412598] pid: 2047's cpu_burst_time = 8017793
[  367.448251] pid: 2045's cpu_burst_time = 8527699
[  368.544751] pid: 2054's cpu_burst_time = 171425125
[  368.560232] pid: 2053's cpu_burst_time = 172528378
[  368.562088] pid: 2052's cpu_burst_time = 169341239
[  368.591145] pid: 2051's cpu_burst_time = 172453053
[  368.602285] pid: 2057's cpu_burst_time = 172576479
[  368.605093] pid: 2055's cpu_burst_time = 170934752
[  368.621104] pid: 2056's cpu_burst_time = 175497259
[  370.317214] pid: 2058's cpu_burst_time = 683216983
[  370.338931] pid: 2060's cpu_burst_time = 681545133
[  370.368216] pid: 2061's cpu_burst_time = 678128843
[  370.375228] pid: 2059's cpu_burst_time = 680291960
[  370.382883] pid: 2063's cpu_burst_time = 682247731
[  370.388140] pid: 2064's cpu_burst_time = 682159823
[  370.406495] pid: 2062's cpu_burst_time = 680965820
[  370.406657] pid: 2043's cpu_burst_time = 2972887
```

다른 프로세스들의 cpu_burst_time도 출력되지만 일단 커널 로그에 cpu_burst_time은 정상적으로 출력되었다.