Linux Kernel Scheduling Policy

확인 보고서

20172609

김시온

Makefile 사용 안하였음.

First.c : 기본 CFS 정책 적용한 c파일.

Second.c : 조정된 NICE값을 적용한, CFS정책을 적용한 c파일.

gcc First.c -> ./a.out으로 컴파일진행하였음.

gcc Second.c -> sudo ./a.out으로 컴파일 진행하였음.

core.c : /usr/src/linux/linux-5.11.22/kernel/sched 폴더 안에 위치.

- 서론

리눅스 커널은 기본적으로 CFS 정책을 사용하고있다.

CFS는 실행 대기 상태인 프로세스들을 우선 순위에 따라 최대한 공정하게 실행하는 스케줄러이다. 이를 확인하기 위해 리눅스 커널의 스케줄링 정책을 확인할 수 있는 프로그램 작성하고, (기본 CFS, 조정된 NICE 값을 적용한 CFS)

커널 수정을 통한 Real Time FIFO 스케줄러 구현하였다.

- 본론

첫번째로, 기본 CFS 정책을 적용한 상태로 프로그램을 생성하였다.

생성되는 프로세스들을 3개의 그룹으로 나누었다.

1) 첫번째 그룹은 수행량이 적게 (사칙연산 1억번)

2) 두번째 그룹은 수행량이 보통으로 (사칙연산 25억번)

3) 세번째 그룹은 수행량이 많게 (사칙연산 100억번)

이는 두번째 프로그램 (NICE 값 조정하여 우선순위 변경) 과

비교하기 위함이다.

프로그램 소스 코드 :

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <pthread.h>
#define SMALL 10000;
#define MIDDLE 50000;
#define BIG 100000;
pid_t pid[21];
void Calc(int n){
        int N;
        if(0<=n && n<7) { N=SMALL;}
        else if (7<=n && n<14){ N=MIDDLE;}
        else if (14<=n && n<21){ N=BIG;}
        else
        { printf("\nplease put 1~3 number.\n"); return;}
        for(int i=0;i<N;i++){
                for(int j=0;j<N;j++){
                        int A = 10 + 10;
                }       }
}
int main(){
        int status;
        for(int i=0;i<21;i++)
        {
                pid[i]=fork();
                if(pid[i]<0){
                        printf("error happnened.\n");
                        exit(1);
                }
                else if( pid[i]>0){
                        printf("%d program begins \n",pid[i]);
                }
                if(pid[i]==0){
                        Calc(i);
                        exit(0);
                }

        }
        for(int i=0;i<21;i++){
                pid_t pid = wait(&status);
                printf("%d process ends \n",pid);
        }
        printf("---------- All processes end ----------\n");
        return 0;
}
```

main 함수에서, fork() 함수를 통해 21개의 자식 프로세스를 생성한다.

그리고, pid가 0보다 클 때, 즉 부모프로세스에서, for문에서 각각 생성되는 프로세스의 PID를 출력한다. 이는 , 자식프로세스에서 printf(PID " program begins ") 를 수행하였을 시 연산속도 및 여러 환경에 따라 생성되는 시간과 다르게 출력되는 경우가 있기 때문이다.

자식프로세스를 21번 생성한 후 , 그 뒤에 for문에서 wait() 함수를 호출하여, 자식프로세스가 종료될 때까지 아무일도 하지 않고 기다리도록 하였다.

수행 결과.

→

프로세스가 생성된 순서대로 종료되진 않지만, 수행량을 다르게 한 그룹별로 순서대로 출력이 됨을 확인할 수 있다.

2. NICE 값을 적용한 프로그램

위에서 작성한 프로그램에서, 각 그룹별로 우선순위를 바꾸었다.

첫번째 그룹은 수행량이 적게 (사칙연산 1억번) , 우선순위 낮게.

두번째 그룹은 수행량이 보통으로 (사칙연산 25억번), 우선순위 보통

세번째 그룹은 수행량이 많게 (사칙연산 100억번) , 우선순위 높음

우선순위는 nice값과 반비례로 설정된다.

nice값은 nice() 함수를 통해 조정할 수 있다.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <pthread.h>

#define SMALL 10000;
#define MIDDLE 50000;
#define BIG 100000;

pid_t pid[21];

void Calc(int n){

    int N;
    if(0<=n && n<7) { N=SMALL;}
    else if (7<=n && n<14){ N=MIDDLE;}
    else if (14<=n && n<21){ N=BIG;}
    else
    { printf("\nplease put 1~3 number.\n"); return;}
```

```
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++){
            int A = 10 + 10;
        }
    }
}


int main(){
    int status;
    for(int i=0;i<21;i++)
    {
        pid[i]=fork();

        if(pid[i]<0){
            printf("error happnened.\n");
            exit(1);
        }
        else if(pid[i]>0){
            printf("%d program begins \n",pid[i]);
        }


        if(pid[i]==0){

            if(i>=0 && i<7 )nice(15);
                        else if (i>=7 && i<14) nice (0);
                        else if (i>=14 && i<21) nice (-19);

            Calc(i);
            exit(0);
        }

    }

    for(int i=0;i<21;i++){
        pid_t pid = wait(&status);
        printf("%d process ends \n",pid);
    }
    printf("---------- All processes end ----------\n");
    return 0;

}
```

First.c 함수와 동일하나, 자식프로세스에서 그룹에 따라 nice()값을 다르게 주었다.

수행 결과

기본 CFS 정책결과와 다르게, nice값에 따라 순서가 바뀌었음을 알 수 있다.

기본이라면, 수행량이 많은 3그룹이 마지막에 종료 되어야한다.

우선순위가 높아져서, 3번 그룹이 먼저 수행되고,

그 후 2번그룹, 1번그룹이 마지막에 수행 되었음을 알 수 있다.

3. 커널 수정을 통한 Real Time FIFO 스케줄러 구현

/usr/src/linux/linux-5.11.12/kernel/sched 폴더 속 core.c 파일에서,

스케쥴러 정책을 변경 할 수 있다.

나는 그 중 두 함수에서 코드를 삽입하고 변경하였다.

3709번째 줄에 위치한

1) int sched_fork(unsigned long clone_flags,strcut task_struct *p)

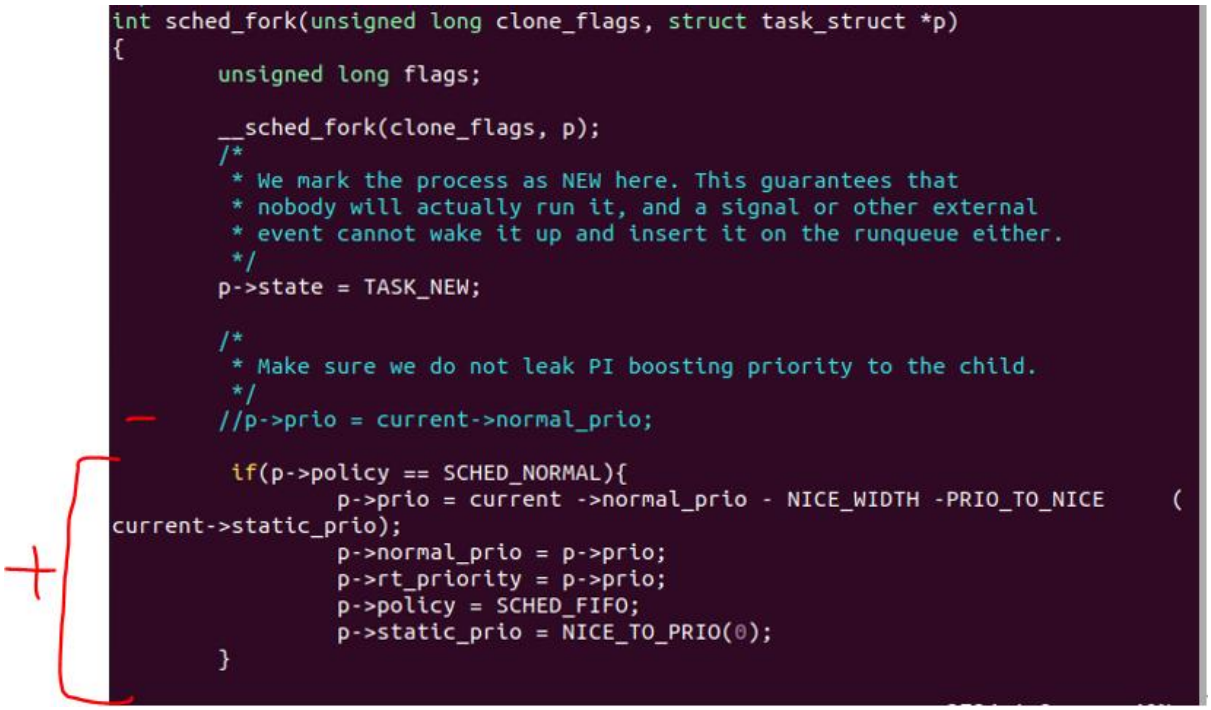
를 우선 수정하였다.

task_struct 에서는 세개의 priority 가 관리되는데,
dynamic priority (task_struct->prio)
static priority(task_struct->static_prio)
normal priority(task_struct->normal_prio)

이 셋 중 시작점은 static priority 가 되는데 처음 process 가 fork 되는 타이밍에
static_prio 는 parent 의 static_prio 를 상속받게 되고, prio 는 parent 의 normal_prio 를 상속받게 된다.

이 보고서에서 세가지의 prio 는 같은 의미라고 생각하고 작성하였다.

3709번째 줄



```
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;

        /*
         * Make sure we do not leak PI boosting priority to the child.
         */
        //p->prio = current->normal_prio;

         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);
        }
```

```
(-)//p->prio = current->normal_prio;
```

p->prio = current -> normal_prio 값을 없앰으로써,

현재 task_struct 의 우선순위를 재조정하였다.

(+)

```
 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);
        }
```

task struct의 정책이 NORMAL, 즉 CFS라면,

그 우선순위를 모두 높이고, p->policy 를 SCHED_FIFO로 설정함으로써 CFS를 FIFO로 바꾸었다.

6040번째줄

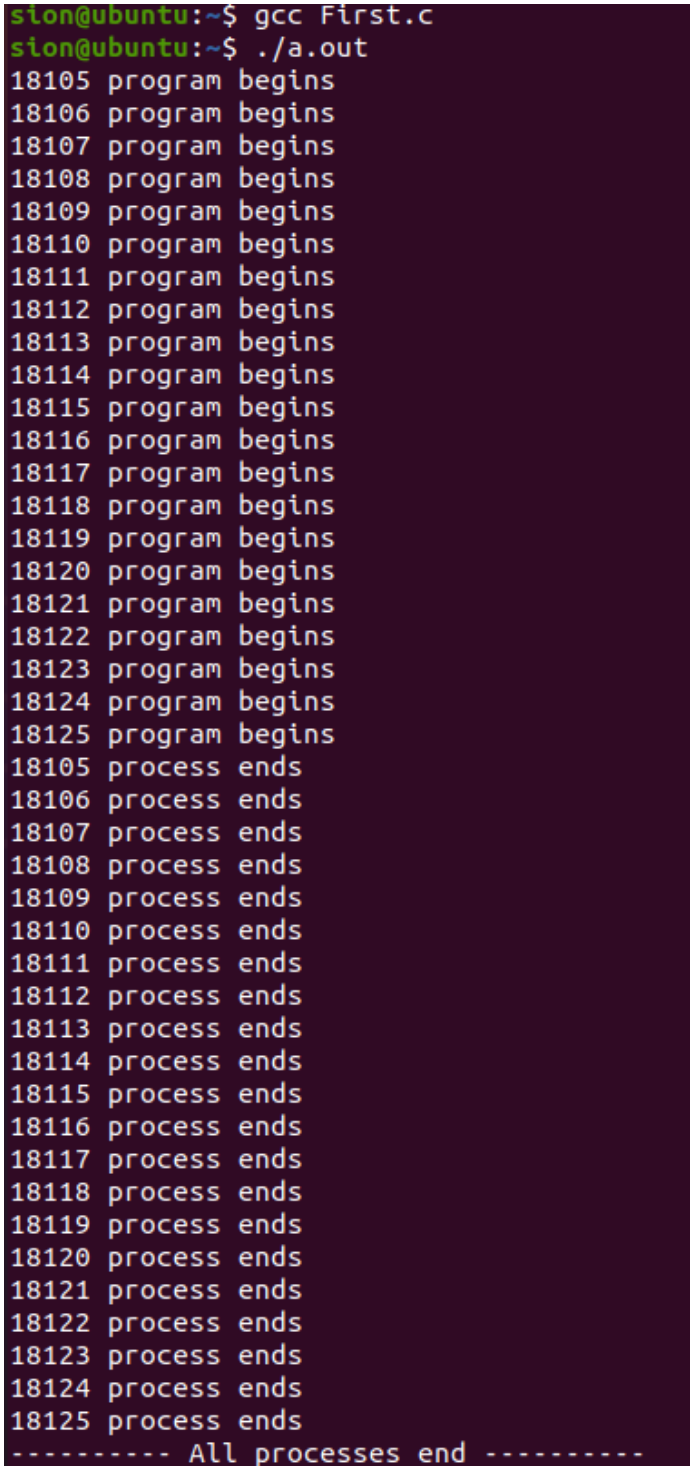
```
static int _sched_setscheduler(struct task_struct *p, int policy,
                               const struct sched_param *param, bool check)
{
        struct sched_attr attr = {
                .sched_policy   = policy,
                .sched_priority = param->sched_priority,
                .sched_nice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

sched_setscheduler함수는, 지정하는 프로세스의 스케쥴 방침을 return 하는 함수이다.

이 함수에서도, CFS정책일 시 , FIFO로 정책을 변환하여 수행하였다.

수행 결과 :

1)First.c 수행.



수행순서대로 프로세스가 종료됨을 알 수 있다.

2) Nice값을 조정한 Second.c 수행

```
sion@ubuntu:~$ gcc Second.c
sion@ubuntu:~$ sudo ./a.out
[sudo] password for sion:
2456 program begins
2457 program begins
2458 program begins
2459 program begins
2460 program begins
2461 program begins
2462 program begins
2463 program begins
2464 program begins
2465 program begins
2466 program begins
2467 program begins
2468 program begins
2469 program begins
2470 program begins
2471 program begins
2472 program begins
2473 program begins
2474 program begins
2475 program begins
2476 program begins
2456 process ends
2457 process ends
2458 process ends
2459 process ends
2460 process ends
2461 process ends
2462 process ends
2463 process ends
2464 process ends
2465 process ends
2466 process ends
2467 process ends
2468 process ends
2469 process ends
2470 process ends
2471 process ends
2472 process ends
2473 process ends
2474 process ends
2475 process ends
2476 process ends
---------- All processes end ----------
sion@ubuntu:~$
```

NICE값을 정한 프로세스도 마찬가지로 순서대로 종료됨을 알 수있다.

- 결론 및 느낀점

기존 CFS는 공평하게 프로세스를 동작하게 하고,

NICE 값을 적용하면 그 우선순위를 우선으로 공평하게 프로세스를 동작한다.

또한 FIFO 스케쥴러는 프로세서를 실행한 순서대로 종료된다.

그저 편하게만 썼던 , 있는 존재의 이유를 몰랐던 OS와 스케쥴러의 작동원리와 의미를 이해하는 재미가 있었다.

그리고 정말 많은 구조체와 잘 짜여진 코드로 리눅스 환경이 구성되는구나 다시한번 새삼 느꼈고,

CFS와 RR의 차이에 관하여 공부하고,

리눅스에 사용되는 함수들을 정리해놓은 홈페이지를 찾아보는 경험을 하였다.