

Lab. 3 프로세스 스케줄러

1) 멀티 프로세스 수행과 프로세스 스위칭 : taskset

: 멀티 프로세스가 하나의 프로세서에서 수행될 때 일정 시간이 지나거나 입출력 함수가 호출되면 CPU 프로세서가 다른 프로세스로 변경되어 수행된다. 하나 이상의 프로세스를 수행하면서 각 프로세스가 차지한 프로세서 시간과 진행도(%)를 taskset 명령을 통해 측정한다.
sched.c 테스트 프로그램을 실행하면 아래 그림과 같이 동작한다. 파라미터 n=1은 수행할 프로세스의 개수를 지정하고, total=10은 수행 시간을 지정하며 resol=2는 프로세스가 수행 후 교체할 타임아웃 값이다.

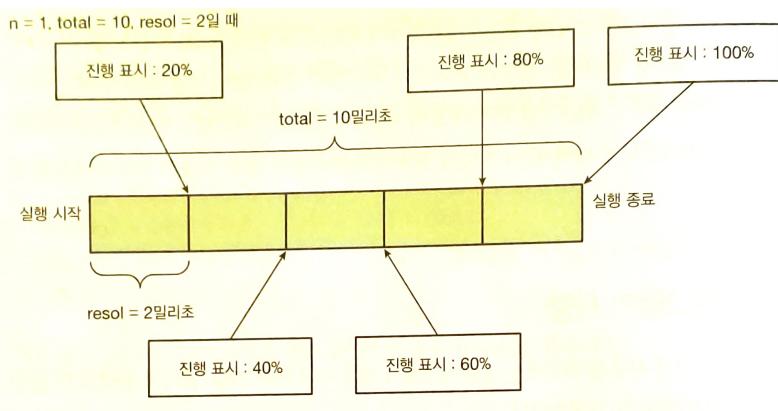


그림 1 테스트 프로그램 sched.c의 동작

```
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <err.h>

#define NLOOP_FOR_ESTIMATION 1000000000UL
#define NSECS_PER_MSEC 1000000UL
#define NSECS_PER_SEC 1000000000UL

static unsigned long nloop_per_resol;
static struct timespec start;

static inline long diff_nsec(struct timespec before, struct timespec after)
{
    return ((after.tv_sec * NSECS_PER_SEC + after.tv_nsec)
            - (before.tv_sec * NSECS_PER_SEC + before.tv_nsec));
}

static unsigned long estimate_loops_per_msec()
{
    struct timespec before, after;
    clock_gettime(CLOCK_MONOTONIC, &before);
```

```

        unsigned long i;
        for (i = 0; i < NLOOP_FOR_ESTIMATION; i++)
            ;

        clock_gettime(CLOCK_MONOTONIC, &after);

        int ret;
        return NLOOP_FOR_ESTIMATION * NSECS_PER_MSEC / diff_nsec(before, after);
    }

static inline void load(void)
{
    unsigned long i;
    for (i = 0; i < nloop_per_resol; i++)
        ;
}

static void child_fn(int id, struct timespec *buf, int nrecord)
{
    int i;
    for (i = 0; i < nrecord; i++) {
        struct timespec ts;

        load();
        clock_gettime(CLOCK_MONOTONIC, &ts);
        buf[i] = ts;
    }
    for (i = 0; i < nrecord; i++) {
        printf("%d\t%d\t%d\n", id, diff_nsec(start, buf[i]) / NSECS_PER_MSEC, (i + 1) * 100 / nrecord);
    }
    exit(EXIT_SUCCESS);
}

static pid_t *pids;

int main(int argc, char *argv[])
{
    int ret = EXIT_FAILURE;

    if (argc < 4) {
        fprintf(stderr, "usage: %s <nproc> <total[ms]> <resolution[ms]>\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }

    int nproc = atoi(argv[1]);
    int total = atoi(argv[2]);
    int resol = atoi(argv[3]);

    if (nproc < 1) {
        fprintf(stderr, "<nproc>(%d) should be >= 1\n", nproc);
        exit(EXIT_FAILURE);
    }

    if (total < 1) {
        fprintf(stderr, "<total>(%d) should be >= 1\n", total);
        exit(EXIT_FAILURE);
    }
}

```

```

    }

    if (resol < 1) {
        fprintf(stderr, "<resol>(%d) should be >= 1\n", resol);
        exit(EXIT_FAILURE);
    }

    if (total % resol) {
        fprintf(stderr, "<total>(%d) should be multiple of <resolution>(%d)\n", total,
resol);
        exit(EXIT_FAILURE);
    }
    int nrecord = total / resol;

    struct timespec *logbuf = malloc(nrecord * sizeof(struct timespec));
    if (!logbuf)
        err(EXIT_FAILURE, "failed to allocate log buffer");

    puts("estimating the workload which takes just one milli-second...");
nloop_per_resol = estimate_loops_per_msec() * resol;
    puts("end estimation");
fflush(stdout);

    pids = malloc(nproc * sizeof(pid_t));
    if (pids == NULL)
        err(EXIT_FAILURE, "failed to allocate pid table");

    clock_gettime(CLOCK_MONOTONIC, &start);

    ret = EXIT_SUCCESS;
int i, ncreated;
for (i = 0, ncreated = 0; i < nproc; i++, ncreated++) {
    pids[i] = fork();
    if (pids[i] < 0) {
        int j;
        for (j = 0; j < ncreated; j++)
            kill(pids[j], SIGKILL);
        ret = EXIT_FAILURE;
        break;
    } else if (pids[i] == 0) {
        // children
        child_fn(i, logbuf, nrecord);
        /* shouldn't reach here */
        abort();
    }
}
// parent
for (i = 0; i < ncreated; i++)
    if (wait(NULL) < 0)
        warn("wait() failed.");

    exit(ret);
}

```

세 가지 실험을 아래 표와 같이 수행한다.

실험명	n	total (ms)	resol (ms)
실험 4-1	1	100	1
실험 4-2	2	100	1
실험 4-3	4	100	1

위 표와 같이 각 실험에 사용할 taskset 명령의 파라미터 값이 제시된 대로 세 가지 실험은 각각 1, 2, 4개의 프로세스를 생성하고 각 프로세스가 프로세서 내에서 어떻게 차례대로 문맥교환되어 수행되는지 관찰해야 한다.

```
$ cc -o sched sched.c  
$ taskset -c 0 ./sched <n> <total> <resol>
```

이 명령어 수행문에서 “-c 0”은 실험의 목적을 위해 CPU 내 다수의 core에 무작위로 흩어져 수행되지 않고 오직 0번 core에서만 실행되도록 제한한다. 실험의 정확성을 위해 필요한 것이다.

이제 1개 프로세스를 수행하는 첫 번째 실험을 아래와 같은 문장으로 수행하고 나머지 파라미터도 변경해서 나머지 두 개의 실험도 수행한다.

```
$ taskset -c 0 ./sched 1 100 1
```

실행 결과는 아래와 같이 출력된다.

```
estimating workload which takes just one milisecond  
end estimation  
0      1      1  
0      2      2  
0      3      3  
0      4      4  
0      4      5  
(중략)  
0      96     96  
0      97     97
```

첫 번째 칸은 프로세스의 번호이며, 두 번째, 세 번째 칸은 진행 %와 경과 시간이다. 버전에 따라 다른 모양을 보일 수 있으나 중요하지 않다. 100밀리초를 1초 간격으로 측정했으므로 총 100줄의 결과가 나온다. 결과를 정리해서 아래의 그래프 두 개를 그리고 해석하는 것이 실험의 목적인데, 엑셀을 이용해서 그래프를 그리려면 데이터가 파일에 저장되는 것이 좋으므로 다음과 같이 출력 파일을 지정해서 다시 수행한다.

```
$ taskset -c 0 ./sched 1 100 1 > 1core-1process.txt
```

1core-1process.txt 파일을 엑셀 프로그램에 올려서 2차원 그래프를 아래 그림 3과 4처럼 두 가지를 그린다. 그림 3은 경과시간 100밀리초 동안 y축에 수행된 프로세스 번호가 나타난다. 당연히 0번만이 나타난다. 그림 4는 0번 프로세스가 100밀리초 경과시간동안 똑같은 진행도(%)로 진행되었음을 보여준다. 왜 이런 결과가 나왔을까? 여기서는 수행되는 프로세스가 오직 0번 하나이기 때문이다. 이제 나머지 두 개의 실험을 수행하고 그림 3, 4와 같은 그래프를 그리고 해석 및 분석한다.

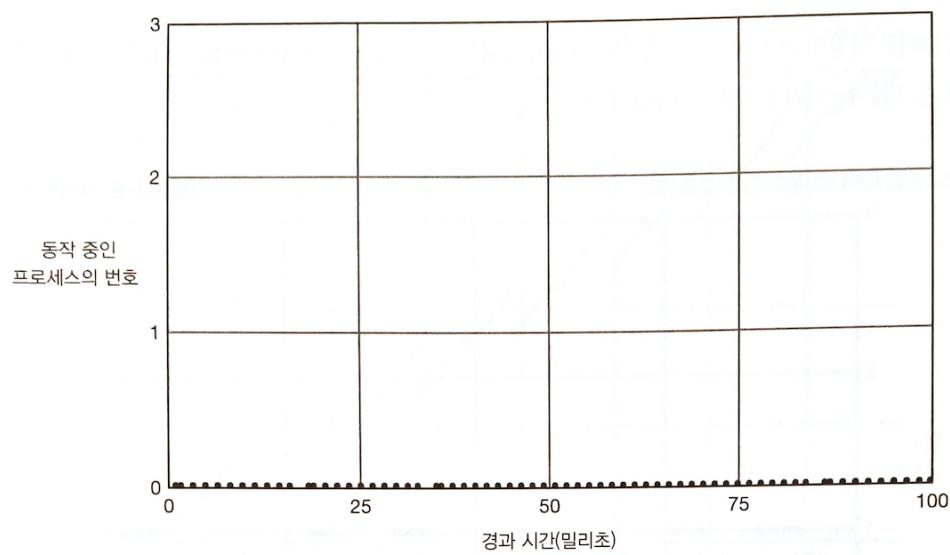


그림 3 논리 CPU로 동작 중인 프로세스 식별

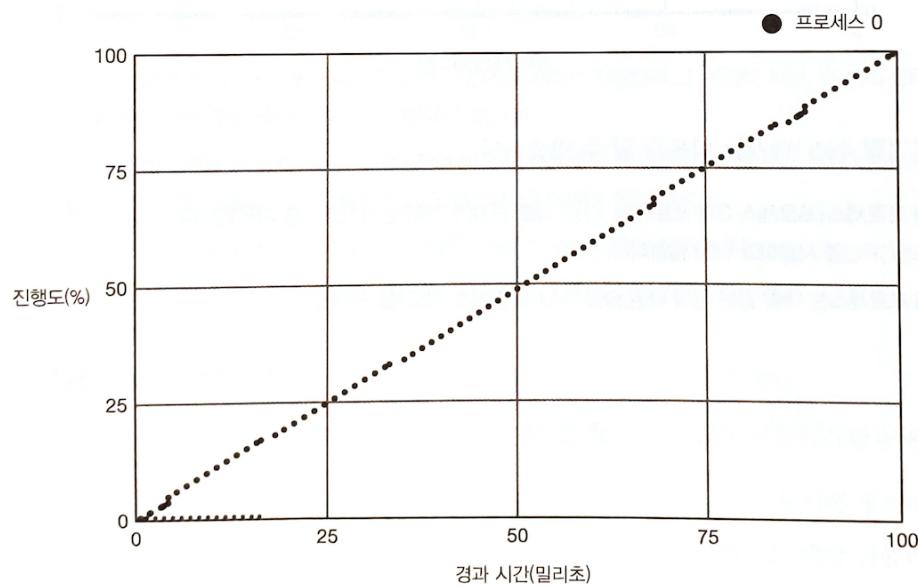


그림 4 프로세스 0의 진행도

2) Throughput과 latency 측정

- : 성능 측정의 지표인 스루풋과 레이턴시를 측정한다.
- 스루풋: 단위 시간당 처리된 일의 양 (높을수록 좋은 성능)
 - = 완료한 프로세스의 수 / 경과 시간
- 레이턴시: 각각의 프로세스가 시작부터 종료까지 경과된 시간 (짧을수록 좋음)
 - = 처리 종료 시간 - 처리 시작 시간

그림 4를 사례로 계산한 스루풋과 레이턴시 예

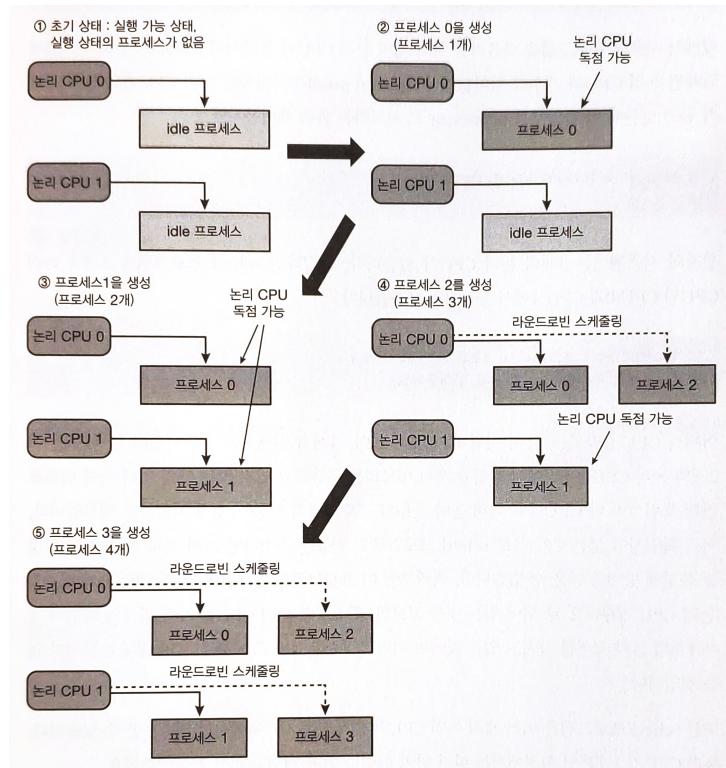
- 스루풋 = 1 프로세스 / 100밀리초

- = 1 프로세스 / 0.1초
- = 10 프로세스 / 1초
- = 10
- 평균 레이턴시 = 프로세스 0의 레이턴시
 - = 100밀리초 - 0밀리초
 - = 100밀리초

프로세스가 2개와 3개의 경우에도 1) 멀티 프로세스 수행과 프로세스 스위칭: taskset 실험의 결과를 이용해서 스루풋과 각 프로세스의 레이턴시, 평균 레이턴시를 계산하고 프로세스의 개수와 스루풋, 레이턴시의 관계를 고찰해본다. 스루풋과 레이턴시의 관계도 어떤 성질을 띠고 있는지 분석해 본다.

3) 멀티 프로세서(다수의 논리 CPU)

: 멀티 프로세서가 동작하는 시스템은 로드 밸런서load balancer 또는 글로벌 스케줄러 global scheduler 기능이 각 프로세서에 프로세스를 배정한다. 멀티 프로세서에 멀티 프로세스를 배정해서 성능을 측정한다. 아래 그림은 2개의 논리 CPU에 4개의 프로세스가 차례로 생성되어 수행될 경우를 보여준다.



먼저 자신의 시스템에 논리 CPU 개수를 아래와 같이 확인한다.

```
$ grep -c processor /proc/cpuinfo
```

일반적으로 짹수개의 값이 나왔을 것이다. 이제 앞에서 사용했던 sched.c 프로그램을 멀티

프로세서 환경에 멀티 프로세스로 수행한다.

```
$ taskset -c 0,4 ./sched <n> 100 1      /* '-c 0,4'에서 0과 4를 띄어쓰기 하지 말 것*/
# <n> = 1, 2, 4 /* 프로세스 개수; 총 3회의 실험을 수행 */
```

taskset 명령을 통해 CPU 0과 4에 프로세스를 배정한다. 0은 첫 번째 프로세서이고 4는 0과 캐시 메모리를 공유하지 않고 논리적으로 최대한 독립된 CPU를 고른 것이다. 성능 측정에서는 서로 독립된 CPU를 선택하는 것이 더 효과적이다. 주의할 점은 4는 하나의 예제이고 실제 여러분의 시스템에서는 위에서 확인한 ‘논리 CPU 개수 / 2’로 선택해야 한다.

아래 그림은 1개의 프로세스만 실행한 실험 결과이다. 여러분은 나머지 2, 4개의 프로세스가 동시에 실행된 실험 결과 그래프를 그려야 한다.

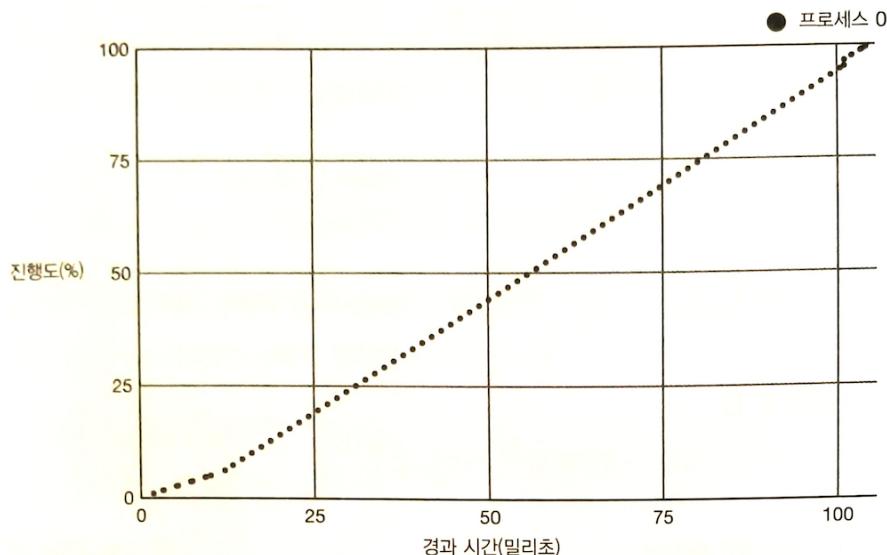


그림 6 CPU 2개, 프로세스 1개의 진행도 그래프

그림 6 기반의 스루풋, 레이턴시 계산은 아래와 같다.

- 스루풋 = 1 프로세스 / 100밀리초
= 1 프로세스 / 0.1초
= 10 프로세스 / 1초
= 10

- 레이턴시 = 100밀리초

나머지 2개의 실험에 대해서도 스루풋과 레이턴시를 계산하고 분석한다.

실제 시스템에서 프로세스의 수행에 사용된 경과 시간과 사용 시간을 time 명령어로 측정할 수 있다.

- 경과 시간: 프로세스 시작부터 종료까지 시간
- 사용 시간: 프로세스가 실제로 CPU를 사용한 시간

다음과 같이 CPU 1개 프로세스 1개로 경과 시간 등을 측정한다.

```
$ time taskset -c 0 ./sched 1 10000 10000      /* 10s 수행 */
0 9811 100

real 0m11.567s
user 0m11.560s
sys 0m0.000s
$
```

위 결과는 하나의 예제이다. 실제로 여러분의 시스템에서 실행하고 결과를 기록한다.

'real'은 경과시간이며, 'user'와 'sys'를 합한 값이다. 'user'는 프로세스가 CPU를 사용한 시간이며, sys는 프로세스 수행 중 사용자 모드에서 시스템 호출을 통해 시스템 콜이 처리된 시간이다.

아래와 같은 조건의 실험을 수행하고 결과를 정리한다.

- CPU 수 = 1, 프로세스 수 = 2
\$ time taskset -c 0 ./sched 2 10000 10000
- CPU 수 = 2, 프로세스 수 = 1
\$ time taskset -c 0,1 ./sched 1 10000 10000
- CPU 수 = 2, 프로세스 수 = 4
\$ time taskset -c 0,1 ./sched 4 10000 10000

3) 프로세스 우선 순위: nice()

: 프로세스가 우선 순위에 따라 CPU 스케줄링에서 먼저 선택이 되면 레이턴시와 같은 성능 지표가 높아진다. 실험을 통해 이러한 결과를 확인해보자.

nice() 호출은 우선 순위를 -19 ~ 20까지 설정한다. 기본 값은 0이며, -19가 가장 높은 값이다.

아래의 예제 프로그램 sched_nice.c를 통해 fork() 함수로 자식 프로세스를 만들고 nice()를 통해 우선 순위 5를 부여한다. 부모 프로세스는 기본 값 0을 갖고 자식 프로세스는 5이므로 부모보다 살짝 낮은 순위를 갖는다.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <err.h>

#define NLOOP_FOR_ESTIMATION 1000000000UL
#define NSECS_PER_MSEC 1000000UL
```

```

#define NSECS_PER_SEC 10000000000UL

static inline long diff_usec(struct timespec before, struct timespec after)
{
    return ((after.tv_sec * NSECS_PER_SEC + after.tv_nsec)
            - (before.tv_sec * NSECS_PER_SEC + before.tv_nsec));
}

static unsigned long loops_per_msec()
{
    unsigned long i;
    struct timespec before, after;

    clock_gettime(CLOCK_MONOTONIC, &before);

    for (i = 0; i < NLOOP_FOR_ESTIMATION; i++)
        ;

    clock_gettime(CLOCK_MONOTONIC, &after);

    int ret;
    return NLOOP_FOR_ESTIMATION * NSECS_PER_MSEC / diff_usec(before, after);
}

static inline void load(unsigned long nloop)
{
    unsigned long i;
    for (i = 0; i < nloop; i++)
        ;
}

static void child_fn(int id, struct timespec *buf, int nrecord, unsigned long nloop_per_resol,
struct timespec start)
{
    int i;
    for (i = 0; i < nrecord; i++) {
        struct timespec ts;

        load(nloop_per_resol);
        clock_gettime(CLOCK_MONOTONIC, &ts);
        buf[i] = ts;
    }
    for (i = 0; i < nrecord; i++) {
        printf("%d%Wt%ld%Wt%ld%Wn", id, diff_usec(start, buf[i]) / NSECS_PER_MSEC, (i
+ 1) * 100 / nrecord);
    }
    exit(EXIT_SUCCESS);
}

static void parent_fn(int nproc)
{
    int i;
    for (i = 0; i < nproc; i++)
        wait(NULL);
}

static pid_t *pids;

```

```

int main(int argc, char *argv[])
{
    int ret = EXIT_FAILURE;

    if (argc < 3) {
        fprintf(stderr, "usage: %s <total[ms]> <resolution[ms]>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    int nproc = 2;
    int total = atoi(argv[1]);
    int resol = atoi(argv[2]);

    if (total < 1) {
        fprintf(stderr, "<total>(%d) should be >= 1\n", total);
        exit(EXIT_FAILURE);
    }

    if (resol < 1) {
        fprintf(stderr, "<resol>(%d) should be >= 1\n", resol);
        exit(EXIT_FAILURE);
    }

    if (total % resol) {
        fprintf(stderr, "<total>(%d) should be multiple of <resolution>(%d)\n", total,
resol);
        exit(EXIT_FAILURE);
    }
    int nrecord = total / resol;

    struct timespec *logbuf = malloc(nrecord * sizeof(struct timespec));
    if (!logbuf)
        err(EXIT_FAILURE, "malloc(logbuf) failed");

    unsigned long nloop_per_resol = loops_per_msec() * resol;

    pids = malloc(nproc * sizeof(pid_t));
    if (pids == NULL) {
        warn("malloc(pids) failed");
        goto free_logbuf;
    }

    struct timespec start;
    clock_gettime(CLOCK_MONOTONIC, &start);

    int i, ncreated;
    for (i = 0, ncreated = 0; i < nproc; i++, ncreated++) {
        pids[i] = fork();
        if (pids[i] < 0)
            goto wait_children;
        } else if (pids[i] == 0) {
            // children

            if (i == 1)
                nice(5);
            child_fn(i, logbuf, nrecord, nloop_per_resol, start);
            /* shouldn't reach here */
        }
}

```

```

    }

    ret = EXIT_SUCCESS;

    // parent

wait_children:
    if (ret == EXIT_FAILURE)
        for (i = 0; i < ncreated; i++)
            if (kill(pids[i], SIGINT) < 0)
                warn("kill(%d) failed", pids[i]);

    for (i = 0; i < ncreated; i++)
        if (wait(NULL) < 0)
            warn("wait() failed.");

free_pids:
    free(pids);

free_logbuf:
    free(logbuf);

    exit(ret);
}

```

표 3 sched_nice.c

```

$ cc -o sched_nice sched_nice.c
$ taskset -c 0 ./sched_nice 100 1 > priority_sch.txt

```

수행 결과를 통해 동작 중인 프로세스 번호를 표시하는 그래프(그림 3)와 프로세스별 경과시간과 진행도를 나타내는 그래프(그림 4)를 그리고 분석한다.