OS HW 1 - Round Robin Scheduler

블랙보드 제출일 : 2019. 11. 18

2016320272 권영진

목차

* 개요

message(IPC) 와 signal 을 이용하여 라운드 로빈 스케쥴링 기법을 구현한다. 부모 프로세스는 10개의 자식 프로세스를 생성하는데, 생성된 프로세서들끼리 메세지와 신호시스템을 통하여 대기큐와 실행큐의 상태를 확인해 볼 것이다.

* 개발환경

가상머신 : Oracle Virtual Box
클라이언트 : Ubuntu 16.04.6 LTS (64bit) with linux 4.4
하드웨어 스펙 : Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz (CPU 4 core), 8GB(RAM)
가상머신 할당스펙 : CPU 4 core, 4GB RAM
사용언어 : C++/C

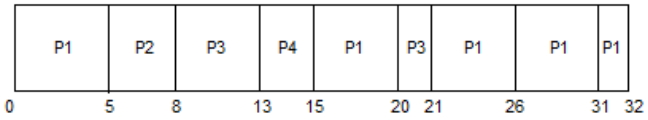
* 코드 이해를 위한 배경 지식

Round Robin Scheduling

Round Robin은 프로세스들 사이에 우선순위를 두지 않고, 순서대로 시간단위(Time Quantum)로 CPU를 할당하는 방식의 CPU 스케줄링 알고리즘이다. 이름의 유래는 Robin 이라는 새는 자식에게 모이를 나눠줄 때, 한명의 자식마다 조금씩 나눠주며 한 무리를 주고나면 다시 첫 자식부터 주는것을 반복하는 것에서 따왔다고 한다. 아무래도 모든 자식들이 공평하게 식사를 할 수 있으므로 starvation(기아상태)를 예방할 수 있을 것이다. 프로세스 에서는 주로 시간단위를 10 ms ~ 100 ms로 지정하여 지정한 Quantum을 마친 프로세스는 다시 준비 큐의 끝으로 돌아가거나 수행을 마치게 된다.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,

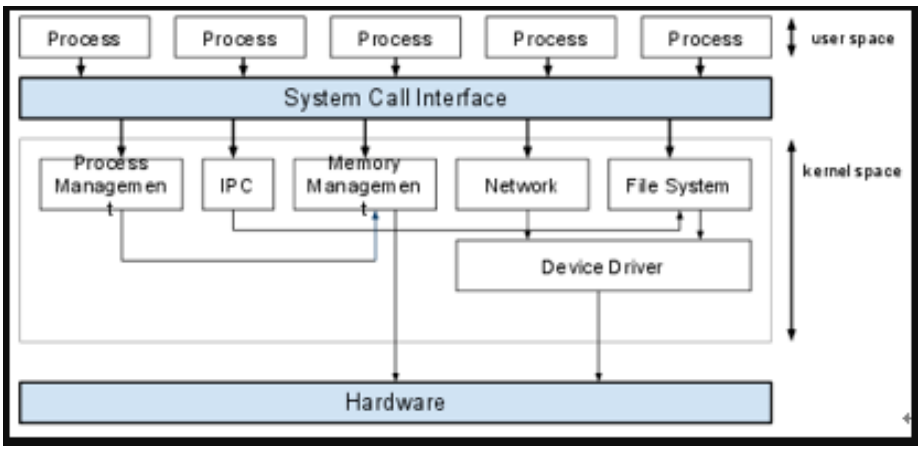


The average waiting time will be, 11 ms.

[Round Robin Scheduling] from 'studytonight.com'

IPC(Inter Process Comunication)

수많은 객체로 이루어진 Linux 시스템이 원할히 작동 하기 위해서는 각 객체간의 원할한 "통신"이 필수적이다. 프로세스가 파일을 열기 위해서는 일련의 작업을 위해서 커널과 통신을 해야 할것이다. 특정 프로세스가 가지는 데이타를 다른 프로세스에게 넘겨서 이를 처리하도록 하는 작업도 필요하다. 이러한 데이타 통신을 위해서 Linux에서는 IPC 라는 내부 프로세스간 통신을 위한 도구를 사용한다.



[리눅스 커널 구조] : IPC example

프로세스간 데이터를 통신할때는 pipe, semaphore 등 다양한 방법이 존재한다. 이번 보고서에서는 System V message queue 방식을 사용할 것인데, 이는 다른 방식에 비하여 구현이 직관적이며 메시지 큐에 사용하고 싶은 데이터에 번호를 붙임으로써 여러 개의 프로세스가 동시에 데이터를 쉽게 다룰 수 있기 때문이다. 실제 코드는 다음과 같다.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key_t key, int msgflg);
// key_t key : 다른 큐와 구별하기 위한 번호.
// int msgflg : 큐생성 옵션, 이번 보고서에서는 IPC_CREAT 사용.
// IPC_CREAT :  key에 해당하는 큐가 있다면 큐의 식별자를 반환하며, 없으면 생성.
int msgsnd (int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
// int msqid : 메시지 큐 식별자로서 msgget 함수의 리턴값이다.
// void *msgp : 전송할 자료, 원하는 자료형을 사용할 수 있다.
// size_t msgsz : 전송할 자료의 크기, sizeof(자료형) 사용한다.
// int msgflg : 동작 옵션, 큐에 공간이 있을때까지 기다리면 0,
// 여유공간 없으면 복귀 시킬경우, IPC_NOWAIT.
ssize_t msgrcv (int msqid, struct  msgbuf  *msgp,  size_t msgsz,
                long msgtyp, int msgflg);
// logn msgtyp : 수신할 메시지 타입, 0 보다 크면 원하는 타입 지정가능
// int msgflg : 동작 옵션
```

메시지큐의 생성과 기존에 있던 메시지큐의 참조는 msgget(2) 를 이용해서 이루어진다. 메시지를 보내기 위해서는 msgsnd(2) 를 사용한다. 보낼때 원하는 자료형을 정할 수 있다. 데이타는 msgrcv(2) 함수를 이용해서 가져올수 있다. 이때 4번째 인자는 가져올 메시지 타입인데 0번을 지정하면 큐의 첫 데이터를 가져오게되고, 0 보다 큰 숫자를 입력하면 원하는 타입을 지정할 수 있다.

SIGNAL

프로세스는 어떠한 일을 처리할 때 여러 예외 상황이 발생할 수 있다. 이에 대한 처리를 위한것이 signal이다. 즉 Ctrl+C(SIGINT) 와같은 비동기적인 상황에 대하여 예외적인 처리를 해주기 위한것이다. SIGNAL은 다음와 같은 종류가 있다.

```
[root@localhost root]# kill -l
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      32) SIGRTMIN    33) SIGRTMIN+1
34) SIGRTMIN+2  35) SIGRTMIN+3  36) SIGRTMIN+4  37) SIGRTMIN+5
38) SIGRTMIN+6  39) SIGRTMIN+7  40) SIGRTMIN+8  41) SIGRTMIN+9
42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13
46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMAX-15 49) SIGRTMAX-14
50) SIGRTMAX-13 51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10
54) SIGRTMAX-9  55) SIGRTMAX-8  56) SIGRTMAX-7  57) SIGRTMAX-6
58) SIGRTMAX-5  59) SIGRTMAX-4  60) SIGRTMAX-3  61) SIGRTMAX-2
62) SIGRTMAX-1  63) SIGRTMAX
```

이번 보고서에서 사용할 신호는 SIGSUSR1, SIGSUSR2 (사용자 정의 신호)와 SIGALRM(지정한 시간마다 신호) 이다. 일반적으로 정수형의 SIGNAL에 사용자가 원하는 이벤트핸들러를 지정하여 사용한다. 이번 보고서에 사용된 예시를 통해서 확인하자.

```
void timer_start() {
  struct sigaction alrm_sigact;
  struct itimerval timer;

  sigemptyset(&alrm_sigact.sa_mask);
  memset(&alrm_sigact, 0, sizeof(alrm_sigact));
  alrm_sigact.sa_handler = &per_tik_handler;
  sigaction(SIGALRM, &alrm_sigact, NULL);

  timer.it_value.tv_sec = 0;
  timer.it_value.tv_usec = 30000;
  timer.it_interval.tv_sec = 0;
  timer.it_interval.tv_usec = 30000;

  setitimer(ITIMER_REAL, &timer, NULL);
}

void per_tik_handler(int sig) {
  // 전체시간(total tik)을 관리하고
  // 매 tik 마다 큐의 상태와 큐의 이동을 담당하는 함수
}
```

'sigaction' 함수는 일정 시간마다 발생하는 SIGALRM 마다 미리 지정한 이벤트핸들러인 per_tik_handler를 실행시킨다.

'setitimer' 함수는 일정 주기마다 신호를 발생시킨다. timer.it_value 는 첫 신호까지의 시간이며 timer.interval 은 그 이후 반복되는 시간 주기를 나타낸다. 여기서는 3000 마이크로초를 기준으로 지정하였다.

* 실제 코드 분석(rr_scheduler.c)

- Headerfile & Define Constant

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h> // signal 사용
#include <sys/ipc.h> // ipc 사용
#include <sys/msg.h> // message queue 사용
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/types.h> // pid_t defined
#include <time.h>
#include <unistd.h>
#include <queue> // deque 자료구조 사용
using namespace std;
```

```
#define CHILDNUMBER 10 // 생성될 자식 프로세스의 수
#define QUANTUM 5 // 프로세스 별 부여받을 퀀텀
#define READY 1
#define WAIT 0
#define MAX_TIK 10000 // 타이머가 동작하는 최대 횟수
```

IPC(message) 및 signal 등을 사용하기 위한 헤더파일을 선언하고 큐를 사용하기위하여 queue 헤더파일을 선언다. c++에 정의된 큐를 이용합니다. 편의를 위하여 자식 프로세스의 수 및 퀀텀은 미리 지정한다.

별다른 설명이 필요하지 않다. 주석에 따라서 사용할 함수와 상수들을 정의한다.

- Data Structure & Functions

```
// Data Structure & Fucntions
struct process {
  long pid;
  int status;
  int remain_quantum;
  int remain_io;

  process(long p, int s, int r_q, int r_io)
      : pid(p), status(s), remain_quantum(r_q), remain_io(r_io){};
} typedef process;

struct parent_process {
  long pid;
  int remaining_io_time;

  parent_process() :pid(0), remaining_io_time(0){};
};

struct child_process {
  long pid;
  int cpu_burst;
  int io_burst;
  child_process(int p, int c, int i) : pid(p), cpu_burst(c), io_burst(i){};
};

struct msg_buffer {
  long m_type;
  pid_t pid;
  int io_msg;
}; //메세지를 통해 io time 을 관리한다.

FILE *fp; // 파일 입출력 위한 포인터
pid_t pid;
int total_tik; // 전체 시간 측정하기 위한 변수
struct parent_process processes[CHILDNUMBER];
// 부모프로세스가 관리할 자식 프로세스들
struct child_process *pchild;
struct sigaction usr_sigact1;
// 사용자 정의신호에 따른 동작 1
struct sigaction usr_sigact2;
// 사용자 정의신호에 따른 동작 2
deque<process> run_queue;
```

```
// 실행큐 선언
deque<process> wait_queue;
// 대기큐 선언
deque<process> done_queue;
// 프로세스 종료후 새로운 burst를 설정하기 전에 저장을 위해 사용.
void burst_handler(int sig);
// io_burst가 종료되면 임의의 수로 새로 할당하는 함수.
void remain_cpu_handler(int sig);
// cpu_burst를 관리하고 이에 따라서 io meg 를 보냄.
void per_tik_handler(int sig);
// 매 tik 마다 큐의 상태에 따라서 신호 발생시킴.
void parent_work();
// 메세지를 기다리며 io 발생시 대기 큐로 이동시킴
void child_work();
// 신호에 따라 burst_handler와 reamin_cpu_handler가 실행되어 프로세스가 관리됨.
void timer_start();
// 타이머를 동작시킴.
int process_num(long pid);
```

사용할 프로세스와 메세지에 사용될 자료구조를 구조체의 형태로 정의하였다. 여기서 주목해야할것은 done_queue이다. done_queue는 종료된 프로세스들을 저장하는데 모든 프로세스가 종료되면 새로운 io & cpu burst를 부여하여 다시 스케쥴링을 하기 위함이다.

- Main

```
int main() {
  total_tik = 0; // set start time.
  fp = fopen("Round_Robin_Dump.txt", "w"); //output file

  for (int i = 0; i < CHILDNUMBER; i++) {
    pid = fork();
    if (pid == -1) {
      exit(1);
    } else if (pid == 0) {  // child proces
      child_work();
      return 0;
    } else {  // parent process
      processes[i].pid = pid;
      run_queue.push_back(process(pid, READY, QUANTUM, 0));
    }
  }
  timer_start();
  parent_work();
  kill(pid, SIGKILL);
  delete &wait_queue;
  delete &run_queue;
  return 0;
}
```

초기 시작시간과 출력할 파일을 지정하고 10개의 자식프로세스를 생성한다. 생성된 부모/자식 프로세스에 따라서 설정된 함수가 실행되지만 이것만으로는 어떤 동작을 하는지 알 수 없다. 앞

서 말하였듯이 생성된 프로세서들끼리 메세지와 신호시스템을 통하여 대기큐와 실행큐를 통제하는데 중점을 맞추어 다음 함수들을 분석하자.

void child_work()

```
void child_work() {
  struct sigaction old1; // 저장을 위한 임시 신호
  struct sigaction old2; // 저장을 위한 임시 신호
  sigemptyset(&old1.sa_mask);
  sigemptyset(&old2.sa_mask);
  srand(time(NULL));
  pchild = new child_process(getpid(), (rand() % 20 + getpid() % 9 + 6),
                  (rand() % 20 + getpid() % 5 + 4)); // 난수 생성

  memset(&usr_sigact1, 0, sizeof(usr_sigact1));
  usr_sigact1.sa_handler = &remain_cpu_handler;
  sigemptyset(&usr_sigact1.sa_mask);
  sigaction(SIGUSR1, &usr_sigact1, 0); //SIGUSR 1에따른 remain_cpu_handler 발생.

  memset(&usr_sigact2, 0, sizeof(usr_sigact2));
  usr_sigact2.sa_handler = (void (*)(int))burst_handler;

  sigemptyset(&usr_sigact2.sa_mask);
  sigaction(SIGUSR2, &usr_sigact2, 0); //SIGUSR 2에 따른 burst_handler 발생.
  while(1);
  sigaction(SIGUSR1, &old1, NULL);
  sigaction(SIGUSR2, &old2, NULL);
}
```

자식 프로세스는 생성시에 임의의 cpu,io burst 시간을 부여받는다. SIGUSR 1 신호에 따라서 현재 프로세스의 cpu burst time을 감소시키는reamin_cpu_handler'가 동작하고 SIGUSR 2 신호가 발생하면 새로운 burst time을 생성하여 지정하는 'burst_handler '가 동작할 것이다.

그러면 우리는 remain_cpu_handler 와 busrt_handler가 어떻게 구현되어있는지 확인하고 SIGUSR1, 2가 언제 생성되는지 알아야 할것이다. 차례대로 확인해 보자.

void remain_cpu_handler(int sig)

```
void remain_cpu_handler(int sig) {
  int msqid;
  struct msg_buffer msg;
  key_t key;
  key = ftok(".", 'A');

  if (-1 == (msqid = msgget(key, IPC_CREAT | 0644))) {
    perror("msgget() failed");
    exit(1);
  } else {
    if (pchild->cpu_burst != 0) {
      pchild->cpu_burst--; // 현재 실행중인 프로세스 cpu burst time 감소
      printf("REMAINING CPU BURST : %d\n", pchild->cpu_burst);
```

```
      fprintf(fp, "REMAINING CPU BURST : %d\n", pchild->cpu_burst);
      if (pchild->cpu_burst == 0) {
        // cpu burst time을 다 사용하면 자식프로세스는
        // 부모 프로세스에게 next I/O-burst time을 보낸다.
        msg.m_type = 4;
        msg.pid = getpid();
        msg.io_msg = pchild->io_burst;
        msgsnd(msqid, &msg, sizeof(struct msg_buffer), 0);
      }
    }
  }
}
```

SIGUSR1 신호(실행 큐에 프로세스가 존재하면 발생)가 발생하면 현재 실행중인 자식프로세스들의 cpu burst time을 소모시킨다. 모든 cpu burst time을 사용하면 부모프로세스에게 io burst time을 보낸다.

void burst_handler(int sig)

```
void burst_handler(int sig) {
  srand(time(NULL));
  pchild->cpu_burst = (rand() % 20 * getpid() % 9 + 1);
  pchild->io_burst = 6;(rand() % 20 * getpid() % 7 + 1);
}
```

SIGUSR2(busrt time 종료 되면 발생) 신호가 발생하면 현재 프로세스의 cpu, io burst time을 초기화 한다.

void timer_start()

```
void timer_start() {
  struct sigaction alrm_sigact;
  struct itimerval timer;

  sigemptyset(&alrm_sigact.sa_mask);
  memset(&alrm_sigact, 0, sizeof(alrm_sigact));
  alrm_sigact.sa_handler = &per_tik_handler; // 매 시간마다 발생되는 신호
  sigaction(SIGALRM, &alrm_sigact, NULL);

  timer.it_value.tv_sec = 0;
  timer.it_value.tv_usec = 30000; // 처음 신호까지의 시간
  timer.it_interval.tv_sec = 0;
  timer.it_interval.tv_usec = 30000; // 다음 부터 주기적인 신호 시간

  setitimer(ITIMER_REAL, &timer, NULL); // 타이머 시작
}
```

프로세스들은 정해진 일정 시간(tik) 마다 burst time이 감소되면 동작하는데 이때 이 타이밍을 결정하는 역할을 하는 함수이다. 앞서 예시로 한번 설명하였지만 30000 마이크로초 단위로 tik을 발생 시킬 것이다. 이때 tik 마다 발생하는 per_tik_handler 함수에서 SIGUSR1, SIGUSR2 신호를 발생시켜서 자식 프로세스들을 관리할 것이다.

void per_tik_handler(int sig)

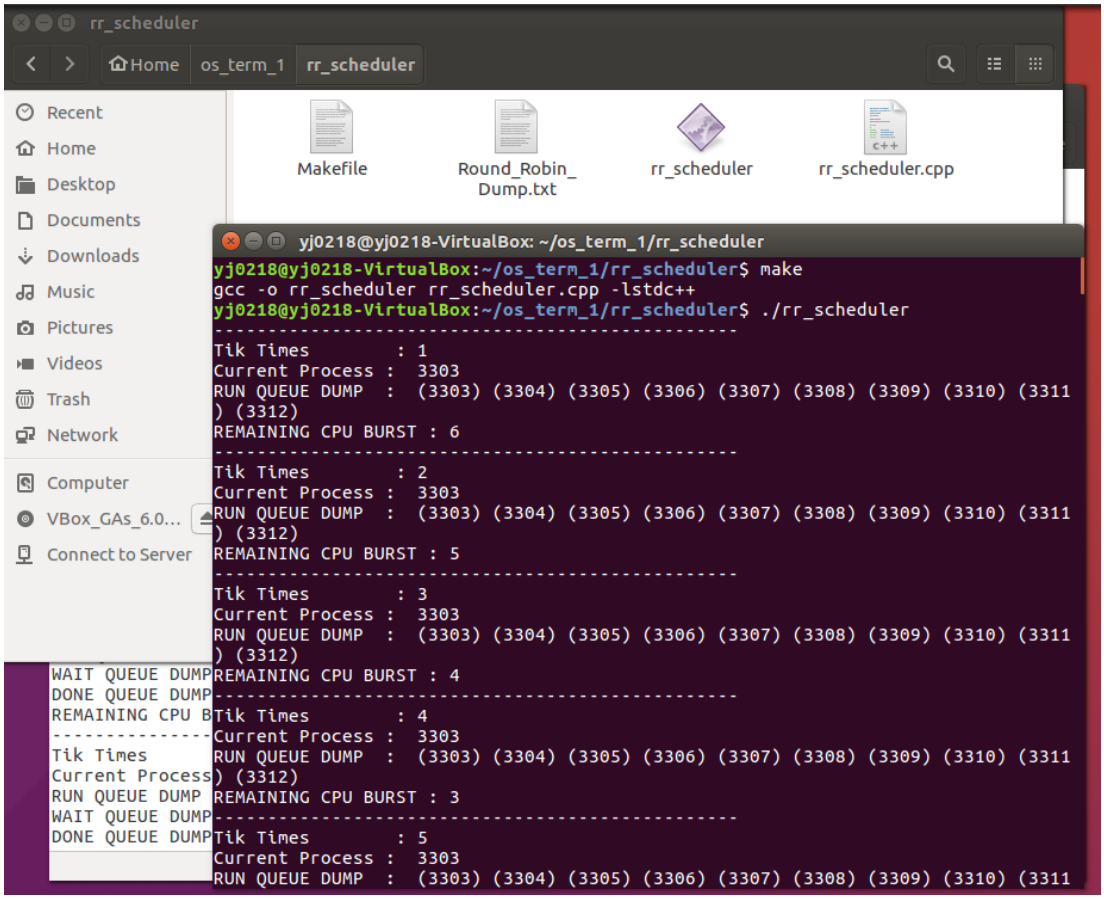
```
void per_tik_handler(int sig) {
  total_tik++;
  printf("--------------------------------------------------\n");
  printf("Tik Times        : %d\n", total_tik); // 누적 시간 출력
  fprintf(fp,
          "--------------------------------------------------\nTime : %d\n",
          total_tik);
  if (!run_queue.empty()) {
    // RUN QUEUE 의 상태 출력
    printf("Current Process :  %ld\n", run_queue.front().pid);
    fprintf(fp, "Current Process :  %ld\n", run_queue.front().pid);
    printf("RUN QUEUE DUMP  : ");
    fprintf(fp, "RUN QUEUE DUMP  : ");
    for (int i = 0; i < run_queue.size(); i++) {
      printf(" (%ld)", run_queue[i].pid);
      fprintf(fp, " (%ld)", run_queue[i].pid);
    }
    printf("\n");
    fprintf(fp, "\n");
  }
  if (!wait_queue.empty()) {
    // WAIT QUEUE 의 상태 출력
    printf("WAIT QUEUE DUMP : ");
    fprintf(fp, "WAIT QUEUE DUMP : ");
    for (int i = 0; i < wait_queue.size(); i++) {
      printf(" (%ld)", wait_queue[i].pid);
      fprintf(fp, " (%ld)", wait_queue[i].pid);
    }
    printf("\n");
    fprintf(fp, "\n");
  }
  if (!done_queue.empty()) {
    // DONE QUEUE 의 상태 출력
    printf("DONE QUEUE DUMP : ");
    fprintf(fp, "DONE QUEUE DUMP : ");
    for (int i = 0; i < done_queue.size(); i++) {
      printf(" (%ld)", done_queue[i].pid);
      fprintf(fp, " (%ld)", done_queue[i].pid);
    }
    printf("\n");
    fprintf(fp, "\n");
  }

  if (!run_queue.empty()) {
    // 런큐의 프로세스에 burst time을 줄이고 SIGUSR1 신호를 발생시킨다.
    process &cur = run_queue.front();
    cur.remain_quantum--;
    if (total_tik >= MAX_TIK) {
```

```
      fprintf(fp, "TIME OUT : TIK EXCEEDED(10000)\n");
      for (int i = 0; i < CHILDNUMBER; i++) {
        kill(processes[i].pid, SIGKILL);
      }
      kill(getpid(), SIGKILL);
    }
    if (cur.remain_quantum == 0 ) {
      // 현재 수행중인 프로세스가 역할을 다하면 DONEQUEUE에 삽입.
      process tmp = run_queue.front();
      run_queue.pop_front();
      tmp.remain_quantum = QUANTUM;
      done_queue.push_back(tmp);

      if (run_queue.empty()) {
        // RUNQUEUE가 비어있으면 DONEQUEUE의 저장했던 종료된 프로세스와 교환.
        deque<process> tmp_q;
        tmp_q = run_queue;
        run_queue = done_queue;
        done_queue = tmp_q;
      }
    }
    kill(cur.pid, SIGUSR1);
  }

  if (!wait_queue.empty()) {
  // 대기큐의 io 시간을 감소시키며 io가 완료되면 실행큐로 이동.
  deque<process>::iterator cur = wait_queue.begin();
    for (int i=0; i<wait_queue.size(); i++) {
      cur->remain_io--;
      deque<process>::iterator tmp = cur;
      if (cur->remain_io <= 0) {
        tmp->status = READY;
        tmp->remain_quantum = QUANTUM;
        run_queue.push_back(*tmp);
        wait_queue.pop_front();
        kill(tmp->pid, SIGUSR2);
      }
      cur++;
    }
  }
}ㅠ
```

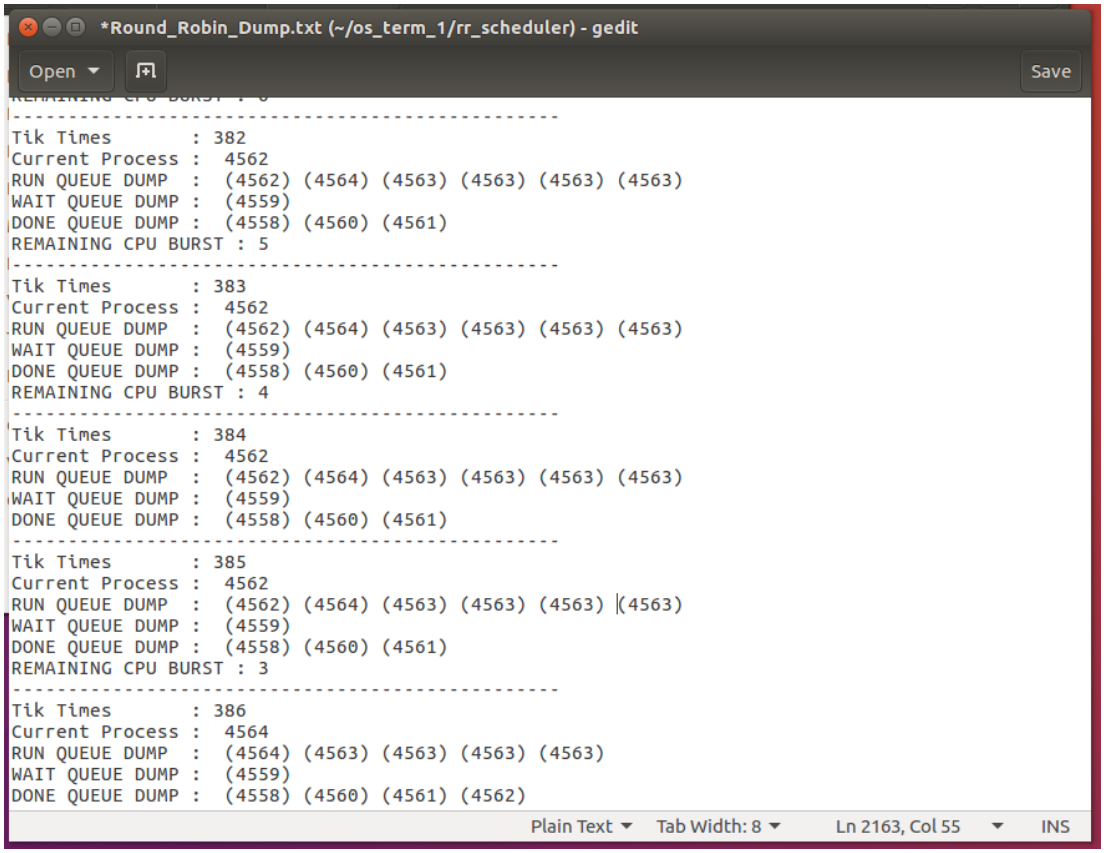
가장 핵심적인 함수로 매 tik 마다 현재 실행중인 프로세스, 실행큐, 대기큐, 완료큐의 상태를 출력한다. 실행큐와 대기큐의 상태에 따라서 cpu_burst와 io_busrt를 소모시키며 사용자 정의신호(USRSIG1, USRSIG2)를 발생시킨다.

* 결과화면

make를 통하여 rr_scheduler 를 생성하고 실행시켜서 그에따른 쉘 화면이 출력되고 텍스트 파일이 생성되었다.


rr_scheduler
Home
os_term_1
rr_scheduler
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
Network
Computer
VBox_GAs_6.0...
Connect to Server
Makefile
Round_Robin_Dump.txt
rr_scheduler
rr_scheduler.cpp
yj0218@yj0218-VirtualBox: ~/os_term_1/rr_scheduler
yj0218@yj0218-VirtualBox:~/os_term_1/rr_scheduler$ make
gcc -o rr_scheduler rr_scheduler.cpp -lstdc++
yj0218@yj0218-VirtualBox:~/os_term_1/rr_scheduler$ ./rr_scheduler
Tik Times : 1
Current Process : 3303
RUN QUEUE DUMP : (3303) (3304) (3305) (3306) (3307) (3308) (3309) (3310) (3311) (3312)
REMAINING CPU BURST : 6
Tik Times : 2
Current Process : 3303
RUN QUEUE DUMP : (3303) (3304) (3305) (3306) (3307) (3308) (3309) (3310) (3311) (3312)
REMAINING CPU BURST : 5
Tik Times : 3
Current Process : 3303
RUN QUEUE DUMP : (3303) (3304) (3305) (3306) (3307) (3308) (3309) (3310) (3311) (3312)
REMAINING CPU BURST : 4
Tik Times : 4
Current Process : 3303
RUN QUEUE DUMP : (3303) (3304) (3305) (3306) (3307) (3308) (3309) (3310) (3311) (3312)
REMAINING CPU BURST : 3
Tik Times : 5
Current Process : 3303
RUN QUEUE DUMP : (3303) (3304) (3305) (3306) (3307) (3308) (3309) (3310) (3311

[스케쥴러 실행]



[생성된 txt파일]

전체 소스코드

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#include <queue>
using namespace std;
#define CHILDNUMBER 10
#define QUANTUM 5
#define READY 1
#define WAIT 0
#define MAX_TIK 10000

// Data Structure & Fucntions
struct process {
  long pid;
  int status;
```

```
  int remain_quantum;
  int remain_io;

  process(long p, int s, int r_q, int r_io)
      : pid(p), status(s), remain_quantum(r_q), remain_io(r_io){};
} typedef process;

struct parent_process {
  long pid;
  int remaining_io_time;

  parent_process() :pid(0), remaining_io_time(0){};
};

struct child_process {
  long pid;
  int cpu_burst;
  int io_burst;
  child_process(int p, int c, int i) : pid(p), cpu_burst(c), io_burst(i){};
};

struct msg_buffer {
  long m_type;
  pid_t pid;
  int io_msg;
};

FILE *fp;
pid_t pid;
int total_tik;
struct parent_process processes[CHILDNUMBER]; // 부모프로세스가 관리할 자식 프로세스들
struct child_process *pchild;
struct sigaction usr_sigact1; // 사용자 정의신호에 따른 동작 1
struct sigaction usr_sigact2; // 사용자 정의신호에 따른 동작 2
deque<process> run_queue;
deque<process> wait_queue;
deque<process> done_queue;

void burst_handler(int sig); // io_burst가 종료되면 임의의 수로 새로 할당하는 함수.
void remain_cpu_handler(int sig); // cpu_burst를 관리하고 이에 따라서 io meg 를 보냄.
void per_tik_handler(int sig); // 매 tik 마다 큐의 상태에 따라서 신호 발생시킴.
void parent_work(); // 메세지를 기다리며 io 발생시 대기 큐로 이동시킴
void child_work(); // 신호에 따라 burst_handler와 reamin_cpu_handler가 실행되어 프로세스가 관리됨.
void timer_start(); // 타이머를 동작시킴.
int process_num(long pid);

int main() {
  total_tik = 0; //set start time.
  fp = fopen("Round_Robin_Dump.txt", "w"); // output file

  for (int i = 0; i < CHILDNUMBER; i++) {
    pid = fork();
    if (pid == -1) {
      exit(1);
    } else if (pid == 0) {  // child proces
      child_work();
      return 0;
    } else {  // parent process
      processes[i].pid = pid;
      run_queue.push_back(process(pid, READY, QUANTUM, 0));
    }
```

```
  }
  timer_start();
  parent_work();
  kill(pid, SIGKILL);
  delete &wait_queue;
  delete &run_queue;
  return 0;
}

void parent_work() {
  struct msg_buffer msg;
  int msqid;
  key_t key;
  key = ftok(".", 'A');
  if (-1 == (msqid = msgget(key, IPC_CREAT | 0644))) {
    perror("msgget() failed");
    exit(1);
  }

  while (1) {
    if (msgrcv(msqid, &msg, sizeof(struct msg_buffer), 4, 0) == -1) {
      // perror("msgrcv() failed");
    } else {
      if (-1 == process_num(msg.pid)) {
        break;
      }
      if (!run_queue.empty()) {
        process tmp_node = run_queue.front();
        run_queue.pop_front();
        tmp_node.remain_io = msg.io_msg;
        tmp_node.status = WAIT;
        wait_queue.push_back(tmp_node);
      }
    }
  }
}

void child_work() {
  struct sigaction old1; // 저장을 위한 임시 신호
  struct sigaction old2; // 저장을 위한 임시 신호
  sigemptyset(&old1.sa_mask);
  sigemptyset(&old2.sa_mask);
  srand(time(NULL));
  pchild = new child_process(getpid(), (rand() % 20 + getpid() % 9 + 6),
                  (rand() % 20 + getpid() % 5 + 4)); // 난수 생성

  memset(&usr_sigact1, 0, sizeof(usr_sigact1));
  usr_sigact1.sa_handler = &remain_cpu_handler;
  sigemptyset(&usr_sigact1.sa_mask);
  sigaction(SIGUSR1, &usr_sigact1, 0); //SIGUSR 1에따른 remain_cpu_handler 발생.

  memset(&usr_sigact2, 0, sizeof(usr_sigact2));
  usr_sigact2.sa_handler = (void (*)(int))burst_handler;

  sigemptyset(&usr_sigact2.sa_mask);
  sigaction(SIGUSR2, &usr_sigact2, 0); //SIGUSR 2에 따른 burst_handler 발생.
  while(1);
  sigaction(SIGUSR1, &old1, NULL);
  sigaction(SIGUSR2, &old2, NULL);
}
```

```
void remain_cpu_handler(int sig) {
  int msqid;
  struct msg_buffer msg;
  key_t key;
  key = ftok(".", 'A');

  if (-1 == (msqid = msgget(key, IPC_CREAT | 0644))) {
    perror("msgget() failed");
    exit(1);
  } else {
    if (pchild->cpu_burst != 0) {
      pchild->cpu_burst--; // 현재 실행중인 프로세스 cpu burst time 감소
      printf("REMAINING CPU BURST : %d\n", pchild->cpu_burst);
      fprintf(fp, "REMAINING CPU BURST : %d\n", pchild->cpu_burst);
      if (pchild->cpu_burst == 0) {
        // cpu burst time을 다 사용하면 자식프로세스는
        // 부모 프로세스에게 next I/O-burst time을 보낸다.
        msg.m_type = 4;
        msg.pid = getpid();
        msg.io_msg = pchild->io_burst;
        msgsnd(msqid, &msg, sizeof(struct msg_buffer), 0);
      }
    }
  }
}

void burst_handler(int sig) {
  srand(time(NULL));
  pchild->cpu_burst = (rand() % 20 * getpid() % 9 + 1);
  pchild->io_burst = 6;(rand() % 20 * getpid() % 7 + 1);
}

void timer_start() {
  struct sigaction alrm_sigact;
  struct itimerval timer;

  sigemptyset(&alrm_sigact.sa_mask);
  memset(&alrm_sigact, 0, sizeof(alrm_sigact));
  alrm_sigact.sa_handler = &per_tik_handler; // 매 시간마다 발생되는 신호
  sigaction(SIGALRM, &alrm_sigact, NULL);

  timer.it_value.tv_sec = 0;
  timer.it_value.tv_usec = 30000; // 처음 신호까지의 시간
  timer.it_interval.tv_sec = 0;
  timer.it_interval.tv_usec = 30000; // 다음 부터 주기적인 신호 시간

  setitimer(ITIMER_REAL, &timer, NULL); // 타이머 시작
}

void per_tik_handler(int sig) {
  total_tik++;
  printf("-------------------------------------------------\n");
  printf("Tik Times        : %d\n", total_tik);
  fprintf(fp,
          "-------------------------------------------------\nTime : %d\n",
          total_tik);
  if (!run_queue.empty()) {
    printf("Current Process :  %ld\n", run_queue.front().pid);
    fprintf(fp, "Current Process :  %ld\n", run_queue.front().pid);
    printf("RUN QUEUE DUMP  : ");
    fprintf(fp, "RUN QUEUE DUMP  : ");
```

```
    for (int i = 0; i < run_queue.size(); i++) {
      printf(" (%ld)", run_queue[i].pid);
      fprintf(fp, " (%ld)", run_queue[i].pid);
    }
    printf("\n");
    fprintf(fp, "\n");
  }

  if (!wait_queue.empty()) {
    printf("WAIT QUEUE DUMP : ");
    fprintf(fp, "WAIT QUEUE DUMP : ");
    for (int i = 0; i < wait_queue.size(); i++) {
      printf(" (%ld)", wait_queue[i].pid);
      fprintf(fp, " (%ld)", wait_queue[i].pid);
    }
    printf("\n");
    fprintf(fp, "\n");
  }
  if (!done_queue.empty()) {
    printf("DONE QUEUE DUMP : ");
    fprintf(fp, "DONE QUEUE DUMP : ");
    for (int i = 0; i < done_queue.size(); i++) {
      printf(" (%ld)", done_queue[i].pid);
      fprintf(fp, " (%ld)", done_queue[i].pid);
    }
    printf("\n");
    fprintf(fp, "\n");
  }

  if (!run_queue.empty()) {
    process &cur = run_queue.front();
    cur.remain_quantum--;
    if (total_tik >= MAX_TIK) {
      fprintf(fp, "TIME OUT : TIK EXCEEDED(10000)\n");
      for (int i = 0; i < CHILDNUMBER; i++) {
        kill(processes[i].pid, SIGKILL);
      }
      kill(getpid(), SIGKILL);
    }
    if (cur.remain_quantum == 0 ) {
      process tmp = run_queue.front();
      run_queue.pop_front();
      tmp.remain_quantum = QUANTUM;
      done_queue.push_back(tmp);

      if (run_queue.empty()) {
        deque<process> tmp_q;
        tmp_q = run_queue;
        run_queue = done_queue;
        done_queue = tmp_q;
      }
    }
    kill(cur.pid, SIGUSR1);
  }

  if (!wait_queue.empty()) {
  deque<process>::iterator cur = wait_queue.begin();
    for (int i=0; i<wait_queue.size(); i++) {
      cur->remain_io--;
      deque<process>::iterator tmp = cur;
      if (cur->remain_io <= 0) {
```

```
        tmp->status = READY;
        tmp->remain_quantum = QUANTUM;
        run_queue.push_back(*tmp);
        wait_queue.pop_front();
        kill(tmp->pid, SIGUSR2);
      }
      cur++;
    }
  }
}

int process_num(long pid) {
  for (int i = 0; i < CHILDNUMBER; i++) {
    if (pid == processes[i].pid) {
      return i;
    }
  }
  return -1;
}
```