운영체제 과제2 : 스케줄링

201619460 이성규

과제 2-1, 2-2, 2-3

먼저 완성된 2-3 코드를 통해 설명을 하겠습니다.

파란 박스 속 코드 및 아무것도 없는 코드는 공통된 코드 부분이며, 빨간 박스 속 코드는 각 과제 당 다르게 구현된 부분을 뜻합니다.

코드 :

```
122 typedef struct code_t{
123     unsigned char op;  // 동작
124     unsigned char len; // 길이(동작 수행 시간)
125 } code;
126
127 typedef struct {
128     int pid;            //ID
129     int arrival_time;   //도착시간
130     int code_bytes;     //코드길이(바이트)
131     int PC;
132     int Waiting;
133     int Response;
134     bool isFirst;
135     int Timeslice;
136     code *operations;   //code tuples 가 저장된 위치
137     struct list_head job, Real_Time, Normal, Idle;
138 } process;
139
```

-> 코드, 프로세스 정의 선언

과제1과 비교하자면,
Waiting, Response, isFirst, Timeslice 변수가 추가로 생겼으며 큐의 종류가 변동됨.

먼저 main 이전에, 즉 프로그램을 실행하기 전 코드와 프로세스 정의를 선언합니다.
추가로 선언된 변수가 일부 있는데, 이름에서 대충 유추할 수 있듯이 각 변수의 의미는 다음과 같습니다.

waiting = 이 프로세스가 로드가 된 이후 ready 상태에서 스케줄링을 대기한 시간.
Response = 이 프로세스가 Arrival 하고, 처음 스케줄링 되기까지 시간.
isFirst = 이 프로세스가 처음 실행되는지. (한번이라도 CPU에 올려졌다면 False가 됨.)
Timeslice = 과제 2-3에만 있는 변수로, Round-Robin scheduling을 위한 타임 슬라이스 변수.

```
141 int main(int argc, char* argv[]) {
142
143     process *cur, *next; // cur : 파일 읽은거 담을 구조체(프로세스) 포인터, next : 리스트속에 넣을 실제 구조체로 파일담은 cur에서 옮겨 담을 포인터
144
145     LIST_HEAD(job_q); //잡 큐 리스트
146
147     cur = (process *) malloc(sizeof(process));
148
149
150     while(fread(cur, sizeof(int)*3, 1, stdin) == 1) { //프로세스 읽기, 실패시 while문 종료
151
152         next = (process *) malloc(sizeof(process));
153
154         next->pid = cur->pid;
155         next->arrival_time = cur->arrival_time;
156         next->code_bytes = cur->code_bytes;      //옮기기
157         next->PC = 0;
158         next->Waiting =0;
159         next->Response =0;
160         next->isFirst =true;
161         next->Timeslice =50;
162
163         next -> operations = (code *) malloc(next->code_bytes); // 할당
164         fread(next->operations,next->code_bytes,1,stdin); //코드 넣기
165
166         INIT_LIST_HEAD(&next->job);
167         INIT_LIST_HEAD(&next->Real_Time);
168         INIT_LIST_HEAD(&next->Normal);
169         INIT_LIST_HEAD(&next->Idle);   // 초기화
170
171         list_add_tail(&next->job,&job_q); //잡큐에 넣기
172
173     }
```

-> test2.bin을 읽어 프로세스 저장

이는 변수 일부 추가가 된 것을 빼면 과제 1번과 동일함.

과제 1과 동일하게 fread를 통해 test2.bin으로부터 프로세스 정보를 받아오고, 받은 프로세스들을 모두 Job 큐에 넣습니다.

```
175     cur->pid = 100;
176     cur->arrival_time = 0;
177     cur->code_bytes = 2;
178     cur->PC = 0;
179     cur->Waiting=0;
180     cur->Response=0;
181     cur->isFirst =true;
182     cur->Timeslice = 50;
183     cur->operations = (code *) malloc(cur->code_bytes);
184     cur->operations[0].op = 0xFF;
185     cur->operations[0].len = 0;
186     INIT_LIST_HEAD(&cur->job);
187     INIT_LIST_HEAD(&cur->Real_Time);
188     INIT_LIST_HEAD(&cur->Normal);
189     INIT_LIST_HEAD(&cur->Idle);
190     list_add_tail(&cur->job,&job_q);    // Idle 프로세스 초기화 후 잡큐에 추가
191
192
193     int clock = 0;
194     int idle_time = 0;
195     int jobQnum = 0;      //잡큐에 들어온 프로세스 개수
196     int FinishProcess = 0;    //완전히 종료한 프로세스 개수
197     int endoptime = 0;       //명령어 끝나는 clock시간 저장
198     bool isworkCPU = true;   // CPU가 일을 했는지 (Idle 타임 파악에 쓰임)
199     process *last;            //running PID 대신 쓰이는 포인터
200     LIST_HEAD(Real_Time_q);   //리얼타임 큐 리스트
201     LIST_HEAD(Normal_q);      //노말 큐 리스트
202     LIST_HEAD(Idle_q);        //아이들 큐 리스트
203
205     list_for_each_entry(cur, &job_q, job){
206         jobQnum++;
207     }
208
```

-> Idle 프로세스 제작

수동적으로 과제에서 주어진 대로 수치를 지정하여 Idle 프로세스 제작, 그 후 job 큐에 넣음.

-> 변수 설명

job큐 속 프로세스의 수와 끝낸 프로세스 수를 비교하며 while문을 clock을 늘리며 돌리게 되는데, 그때 쓰이는 변수들 선언.

그 이후 Idle 프로세스를 직접 제작하여 Job 큐에 넣고 스케줄링을 할 때 사용할 변수들을 선언합니다.
이제 뒤 코드에선 idle을 제외한 job 큐 속 모든 프로세스를 완료할 때 까지 while 문을 돌며 스케줄링 하게 되는데 (라인 209) 이때, while문 안에서 cur 포인터는 현재 CPU작업을 수행하는 running 프로세스를 의미하며, last 포인터는 이전 clock에서 running 하였던 프로세스를 의미합니다.
그 외 다른 변수들은 어떤 용도로 쓰이는지 주석으로 설명을 해 놓았습니다.

```
209 while( jobQnum-1 != FinishProcess ) // 잡큐속 프로세스수와 명령어 다 끝난 프로세스 들어온 프로세스 개수가 동일할 때까지
210 {
211
212     // 도착하면 PID에 따라서 각자 맞는 큐에 넣어짐.
213     list_for_each_entry(next, &job_q, job){
214
215         if( next->arrival_time == clock) //도착시간이 clock과 같다면
216         {
217             if(next->pid == 100) // Pid가 100이라면
218             {
219                 list_add_tail(&next->Idle,&Idle_q); //Idle큐에 프로세스 추가
220             }
221             else if(next->pid >= 80) // Pid가 100이 아니고 80 이상이라면
222             {
223                 list_add_tail(&next->Real_Time,&Real_Time_q); //Real Time 큐에 프로세스 추가
224             }
225             else // 나머지
226             {
227                 list_add_tail(&next->Normal,&Normal_q); //Normal Time 큐에 프로세스 추가
228             }
229             fprintf(stdout, "%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n", clock, next->pid, next->arrival_time, next->code_bytes, next->PC);
230         }                                                                    //로드된 프로세스 정보 출력
231
232     }
233
235
```

-> 프로세스 로드

프로세스가 arrive 하면, PID에 따라 알맞은 큐 속에 들어감.

앞서 언급 하였듯, while문을 통해 idle 프로세스를 제외한 모든 프로세스를 다 끝낼 때 까지 매 루프 당 clock을 증가시키며 스케줄링을 하게 됩니다. (라인 209) 매 while 루프 처음에는 잡 큐를 순회하며 프로세스의 arrival_time 가 clock과 동일 할 때, pid에 따라 Real_Time 큐, Normal 큐, idle 큐에 나뉘어 들어가게 됩니다.

```
236     last = cur; // 큐에서 새로운 프로세스 꺼내기전, 미리 실행하던 프로세스 저장하기 위한 포인터
237
240     //스케줄링
241     if(list_empty(&Real_Time_q)) // Real_Time 큐가 비어있다면.
242     {
243         if(list_empty(&Normal_q)) // Real_Time 큐가 비어있고 Normal 큐도 비어있다면.
244         {
245             cur = list_entry(Idle_q.next, process, Idle); // Idle 프로세스 가져오기
246         }
247         else  // Real_Time 큐가 비어있으나 Normal 큐 프로세스 있다면.
248
249
                 Normal 큐의 구현 스케줄링에 따라 다른 코드
285
286     }
287     else
288     {
289         cur = list_entry(Real_Time_q.next, process, Real_Time); // Real_Time 큐에서 맨 앞 프로세스 가져오기
290     }
```

-> 공통 스케줄링

우선순위 :
Real > Normal > Idle

-> 개별 스케줄링

공통스케줄링(Multi-level queue 구조)

먼저 last 포인터에 이전 클락 에서 했던 프로세스를 저장하고 (라인 236), cur 포인터에 현재 running할 프로세스를 스케줄링 합니다. 먼저 공통적으로 우선순위를 고려하여 Real 큐에 프로세스가 있다면 우선적으로 Real 큐에서 프로세스를 뽑고, 그 다음은 Normal큐, 마지막으로 두 큐가 비어있을 때 Idle 큐에서 프로세스를 뽑습니다.

과제 2-1 개별 스케줄링:

```
else  // Real_Time 큐가 비어있으나 Normal 큐 프로세스 있다면.
{
    cur = list_entry(Normal_q.next, process, Normal); // Normal 큐에서 맨 앞 프로세스 가져오기
}
```

과제 2-1 의 경우 큐에 따른 우선순위를 부여하는 공통 스케줄링의 구현(Multi-level queue 구조)이 목적이므로 Normal 큐에서 FIFO스케줄링을 채용합니다. 그러므로 순서대로 그냥 Normal큐의 맨 앞 프로세스를 가져옵니다.

과제 2-2 개별 스케줄링:

```
else  // Real_Time 큐가 비어있으나 Normal 큐 프로세스 있다면.
{
    if( clock >= endoptime ) // 수행중인 명령어가 있다면 Normal 큐에서 스케줄링 하지 않음.
    {
        next = list_entry(Normal_q.next, process, Normal);
        list_for_each_entry(cur, &Normal_q, Normal)
        {
            if(cur->operations[0].len < next->operations[0].len)
            {
                next = cur;
            }
        }
        cur = next; // Normal 큐에서 명령어길이 가장 작은 프로세스 선택
    }
}
```

과제 2-2는 SRJF 스케줄링을 채용합니다.

현재 수행중인 명령어가 없다면 Normal 큐를 순회하며 가장 명령어 길이가 작은 프로세스를 선택하여 cur포인터에 갱신합니다. 만약 수행중인 명령어를 신경 쓰지 않고 Normal 큐에서 길이가 작은 프로세스를 선택하게 된다면 Normal 큐 사이에서도 프로세스 수행 중에 명령어 길이에 따라 preemption이 일어나게 됩니다. 교수님께서 preemption 은 realtime process 에 대해서만 일어난다고 제시해 주었기 때문에 if문 조건을 추가하였습니다.

과제 2-3 개별 스케줄링:

```
else  // Real_Time 큐가 비어있으나 Normal 큐 프로세스 있다면.
{

    bool allTimesliceZero = true;   // 큐속 모든 프로세스의 타임슬라이스가 0인지

    list_for_each_entry(next, &Normal_q, Normal)
    {
        if(next->Timeslice != 0)
        {
            allTimesliceZero = false; // 0이 아닌게 1개라도 있으면 변수 수정
        }

    }

    if(allTimesliceZero) // Normal 큐속 모든 프로세스 타임 슬라이스가 0 = 라운드 종료
    {
        fprintf(stdout, "%04d CPU: ROUND ENDS. Recharge the Timeslices\n", clock);
        list_for_each_entry(next, &Normal_q, Normal)
        {
            next->Timeslice +=50;
        }
    }

    next = list_entry(Normal_q.next, process, Normal);
    list_for_each_entry_reverse(cur, &Normal_q, Normal)
    {
        if(cur->Timeslice != 0)
        {
            next = cur;
        }
    }
    cur = next; // Timeslice가 0이 아닌 Normal 큐의 맨 앞 프로세스(FIFO 기반)

    if(allTimesliceZero && last == cur ) // 라운드가 재시작 되었는데 프로세스가 동일함. : CPU 작업 전환 X
    {
        fprintf(stdout, "%04d CPU: Not Switched\tPID: %03d\n", clock, cur->pid);
    }
}
```

과제 2-3은 Round-Robin Scheduling을 구현합니다.

← 먼저 Normal 큐를 순회하며 Time slice를 모두 소비했는지 확인하여,

← 라운드가 종료되었는지 판단합니다. 라운드가 종료되었다면 라운드가 종료되었음을 출력하고 Time slice를 각 프로세스에 채워줍니다.

←
각 라운드 안에서는 FIFO 스케줄링을 하므로 Normal 큐의 맨 앞 프로세스를 cur 프로세스로 갱신합니다.

↑ 그리고 라운드가 재시작 되었으나 이전 클락에서 실행한 프로세스와 이번 클락에서 실행하는 프로세스가 동일하다면 CPU 작업이 전환되지 않음을 출력합니다.

```
291         if(clock == 0)  // 처음에는 컨텍트 스위치 없음.
292         {
293             last = cur;
294         }
295 
296 
297         //Context Switching
298         if( cur->pid != last->pid ) // 이전 실행했던 pid랑 다를경우 컨텍스트 스위치
299         {
300             if(clock < endoptime) // 명령어가 안끝났는데, last 와 cur 프로세스가 다르다면 우선 순위로 인한 프로세스 변경이므로 Preemption.
301             {
302                 last->PC -= 1;
303                 last->operations[cur->PC].len = (unsigned char)(endoptime - clock); // 수행한 clock 만큼 명령어 길이 감소
304                 endoptime = clock;
305                 last->isFirst = false;
306             }
307 
308             list_for_each_entry(next, &Normal_q, Normal) // 프로세스가 처음이 아닌경우 context swiching은 waiting 시간이므로
309             {
310                 if(!next->isFirst) //큐속 프로세스가 isFirst가 false인 것은 대기중인것임.
311                 {
312                     next->Waiting += 5;
313                 }
314             }
315             list_for_each_entry(next, &Real_Time_q, Real_Time)
316             {
317                 if(!next->isFirst) //context switching 시간 동안 waiting 시간 더하기
318                 {
319                     next->Waiting += 5;
320                 }
321             }
322 
323             clock += 5;
324             idle_time += 5;
325             fprintf(stdout, "%04d CPU: Switched\tfrom: %03d\tto: %03d\n" , clock, last->pid, cur->pid); //CPU 작업 전환 정보 출력
326             int i;
327             for(i=1 ; i<5 ; i++)   // 컨텍스트 스위치 도중에 Arrival하는 프로세스 추가.
328             {
329                 list_for_each_entry(next, &job_q, job)
330                 {
331                     if( next->arrival_time == (clock-5+i))// 컨텍트 스위치 도중 Arrival한 프로세스는 끝나고 도달한걸로 표시됨.
332                     {
333                         if(next->pid == 100) // Pid가 100이라면
334                         {
335                             list_add_tail(&next->Idle,&Idle_q); //Idle큐에 프로세스 추가
336                         }
337                         else if(next->pid >= 80) // Pid가 100이 아니고 80 이상이라면
338                         {
339                             list_add_tail(&next->Real_Time,&Real_Time_q); //Real Time 큐에 프로세스 추가
340                             if(cur->pid != 0) // 컨텍스트 스위치 도중 Real Time 들어오면 Response 가 갱신이 안됨. 단 프로세스0번은 예외
341                             {
342                                 cur->Response = clock - cur->arrival_time;
343                             }
344                         }
345                         else // 나머지
346                         {
347                             list_add_tail(&next->Normal,&Normal_q); //Normal Time 큐에 프로세스 추가
348                         }
349                         fprintf(stdout, "%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n",
350                                         clock, next->pid, next->arrival_time, next->code_bytes, next->PC); //로드된 프로세스 정보 출력
351 
352                     }
353 
354                 }
355                 continue; // 컨텍스트 스위치하고 cur을 다시 갱신 해야함.
356             }
357 
```

-> Context Switching

(298 라인) : 만약 while문을 돌 때 last와 cur가 다르다면 (이전 동작한 프로세스와 현재 동작하는 프로세스 가 다르다면) 컨텍스트 스위치가 일어납니다. 조건을 cur != last로 바꾸어도 동일하게 동작하였습니다.

(300 - 306 라인) : preemption을 구현해주었습니다. 이 코드에선 명령어를 끝날 때 PC를 하나 올리는 게 아닌 실행시작 할 때 PC를 1을 더하게 되는데, 명령어를 다 못했으므로 다시 PC를 1 줄이고, 실행한 만큼 명령어의 수행 시간을 감소시킵니다. 그 후 프로세스가 한번 CPU에 올라왔다는 것을 isFirst 변수를 변경하여 표시해줍니다.

(308 - 325 라인) : Context Switching을 하며 clock이 소모됩니다.

(308 - 321 라인) : 큐 속에 있는데 실행 된 적이 있는 프로세스는 대기 중인 프로세스입니다. 큐 속에서 대기 중인 프로세스는 컨텍스트 스위치 동안 waiting을 5 더하도록 구현하였습니다.

(326 - 354 라인) : Context Switching 도중 Arrival한 프로세스를 For문을 돌며 PID에 따라 큐 속에 넣습니다. 단, 이 부분에서 주의 깊게 봐야할 부분이 있는데, 바로 라인 340 - 343입니다. 제 코드 구현 상, Context Switching 도중 Real Time 프로세스가 들어오게 된다면 Response를 갱신해야합니다.
지금 설명하고 있는 Context Switching 코드 다음으로는 실질적으로 명령어를 수행하며 Response를 지정하는 코드가 옵니다. 즉 Response는 보통 명령어를 처음 수행할 때 지정을 하도록 제가 구현을 한 것입니다.
그러나 355라인에서 스케줄링을 다시하게 되는데, 이때 스케줄링 순위에 밀리게 될 경우 (Real Time 이 있을 경우) 명령어를 수행하는 코드에 진입 하지 못하기 때문에 Response가 갱신이 되지 않게 됩니다.
즉 실질적으로는 스케줄링에 밀리게 되면 명령어는 수행하지 못하나, Response는 갱신이 되어야하므로 수동적으로 Response를 지정해 주도록 구현한 것입니다.

(355 라인) : Context Switching을 하는 동안 추가된 프로세스가 있으니 cur을 다시 갱신해야합니다. 우선순위가 더 높은 Real Time 프로세스가 들어왔을 수 도 있기 때문입니다.
그냥 Context Switching IF문안에 앞의 프로세스 스케줄링 코드를 복사하여 넣어서 구현할 수 있었으나, 코드가 길어지고 보기 안 좋다 생각하여 continue를 사용하여 스케줄링을 다시 실행하도록 구현하였습니다.
만약 continue를 사용하지 않고 그냥 다음 clock으로 넘어가게 구현 한다면 일부 Context Switching에서는 Context Switching 하는 것 자체에 1 clock을 소모하게 되어, 총 6 clock이 소모되는 현상이 일어났습니다.

```
361     // 실질적 명령어 실행
362     if( cur->pid == 100 )
363     {
364         isworkCPU = false;
365     }
366     else if ( clock >= endoptime ) // idle 프로세스(PID = 100) 아니며 현재 진행 중인 명령어가 끝난 경우
367     {
368         isworkCPU = true;
369         if( cur->PC < cur->code_bytes/2 )// 프로세스의 모든 명령어가 끝났는 지 확인 (안끝남)
370         {
371             if(cur->isFirst) // 이 프로세스 명령어 처음 실행
372             {
373                 if(cur->Response == 0) // Response가 갱신 안된 경우만
374                 {
375                     cur->Response = clock - cur->arrival_time; //클락이 도착한 후 몇 클락 지난지 Response 에 저장
376                 }
377                 cur->Waiting  = clock - cur->arrival_time;  //클락이 도착한 후 몇 클락 지난지 Waiting 에 저장
378             }
379             endoptime = cur->operations[cur->PC].len + clock; //이 명령어가 끝나는 시간 저장
380             cur->PC += 1;                                     //PC레지스터 1더해서 프로세스 끝난것을 표시
381         }
382         else // 프로세스의 모든 명령어가 끝났음
383         {
384             if( cur->pid >= 80 ) // realtime 프로세스
385             {
386                 list_del_init(&cur->Real_Time); // 프로세스 명령어 다 수행 끝났으니 큐에서 삭제
387             }
388             else // 노말 프로세스
389             {
390                 list_del_init(&cur->Normal); // 프로세스 명령어 다 수행 끝났으니 큐에서 삭제
391             }
392             fprintf(stdout, "%04d CPU: Process is terminated\tPID:%03d\tPC:%03d\n" , clock, cur->pid , cur->PC ); // 종료 출력
393             FinishProcess++; // 종료 개수 1개증가.
394             continue; // 프로세스의 명령어가 끝났으니 아무행동도 하지않음 = clock이 소모되지 않음.
395         }
396
397     }
```

-> 프로세스 수행

pid = 100 인 경우

pid != 100 인 경우

이제 스케줄링 된 프로세스(cur)를 수행합니다. 먼저 pid가 100인지 아닌지에 따라 구분을 합니다. (362 라인)

만약 pid가 100이라면 CPU가 일하지 않는다고 체크하여 idle 시간을 증가시키게 됩니다. (362 - 365 라인)

만약 pid가 100이 아니라면 CPU가 지금 수행하고 있는 명령어가 있는지 확인합니다. (366 라인)
이는 명령어가 끝나는 시간(endoptime 변수 : End of Operation Time)과 clock을 비교하여 clock이 더 크다면 수행하고 있는 명령어가 끝났다고 판단합니다.

pid가 100이 아니며 CPU가 명령어 수행을 하고 있는 게 없다면 cur 프로세스의 명령어가 끝났는지 PC를 통해 확인합니다. (369 라인, 382 라인)

cur 프로세스의 명령어가 끝나지 않았다면 (369 - 381 라인)
먼저 이 프로세스가 처음 명령어를 실행하는 것인지 판단합니다. 만약 cur가 처음 명령어를 실행하는 프로세스라면 로드되고 처음 실행되기 까지 시간을 waiting 과 response 시간에 계산하여 지정해줍니다.(371 - 378 라인)
단 여기서 일부 프로세스는 명령어를 수행하지 않고 response만 한 프로세스도 존재 할 수 있으므로 response가 지정되지 않은 프로세스만 갱신하도록 합니다. (373 - 376 라인)
waiting 과 response 시간을 지정하였다면, 그 후 명령어 끝나는 시간 저장과 PC 증가를 통해 명령어를 수행해 줍니다.
만약 처음 실행되는 프로세스가 아니라면 그냥 명령어만 수행해 줍니다.

만약 cur 프로세스의 pid가 100이 아니며 명령어도 모두 끝나있다면 (382 - 395 라인) pid에 따라 알맞은 큐에서 cur 프로세스를 제거해주고, 다시 스케줄링을 실행합니다. 이때, 명령어를 수행한 게 아니라 그저 스케줄링을 다시 하여 cur를 갱신할 뿐이므로 clock은 소모되지 않습니다. (continue 사용)

여기까지 스케줄링을 통해 프로세스를 선택하고, 이전 clock에서 실행한 프로세스와 pid가 다를 시 컨텍스트 스위치를 하도록 구현해주었으며 실질적인 프로세스 명령어 수행도 구현하였습니다. 이후 while문 안의 코드들은 클락 관련된 코드가 작성되어있습니다.

```
400     //큐에 존재중인 프로세스 웨이트 타임 증가
401     list_for_each_entry(next, &Normal_q, Normal)
402     {
403         //실행한적 있는데 현재 진행한 프로세스가 아님. 단, PC값이 줄어 있어야만함. -> waiting 타임 증가
404         if(!next->isFirst && cur != next && next->PC < next->code_bytes/2)
405         {
406             next->Waiting += 1;
407         }
408     }
409     list_for_each_entry(next, &Real_Time_q, Real_Time)
410     {
411         // isFirst가 false 이며 현재 진행한 프로세스가 아닌경우
412         if(!next->isFirst && cur != next && next->PC < next->code_bytes/2)
413         {
414             next->Waiting += 1;
415         }
416     }
```

-> 프로세스의 Waiting Time 증가

프로세스가 한번 실행되었다가 다른 프로세스로 인해 큐에서 대기하게 된다면 (preemption 등으로 인해) 매 클락 waiting 변수가 증가하도록 구현한 코드입니다.

특정 프로세스가 한번 실행이 된 적이 있는데 (isFirst = false) 명령어가 안 끝난 채로 큐에 들어있으며 (PC < code_bytes/2), 지금 실행 중인 것이 아니라면 (cur != next) 이 프로세스는 대기 중인 프로세스입니다.
(404, 412 라인)

```
419         // 클락 관련
420         if( cur->pid < 80 ) // 현재 진행 프로세스가 노말 클래스의 프로세스인 경우
421         {
422             cur->Timeslice -= 1;
423         }
424         if(!isworkCPU) // cpu 일 안했으면 idle시간 증가
425         {
426             idle_time++;
427         }
428         clock++;
429 
430     }
```

-> Round-Robin 스케줄링

cpu 사용한다면 매 클락 마다 Time slice 감소

-> Idle 시간, clock 증가

그리고 이제 while문의 마지막 부분으로 CPU가 일하지 않았다면 idle_time을 증가시켜주고 (424 라인)
매 while 루프마다 clock을 증가시켜줍니다. (428 라인)

그리고 과제 2-3만 해당되는 부분으로, 만약 이번 clock에 실행한 프로세스가 노말 클래스의 프로세스인 경우 Time slice를 1 감소해줍니다. (420 - 423 라인)

Time slice 관련 코드의 경우 아래 코드처럼 CPU가 일한 경우 Time slice를 줄여주는 방식으로 구현해도 딱히 작동하는 데에는 상관이 없었습니다.

```
if(!isworkCPU) // cpu 일 안했으면 idle시간 증가
{
    idle_time++;
}else   // cpu 일 했을때 timeslice 감소
{
    cur->Timeslice -= 1;
}
```

그러나 코드의 작동 원리 상 Real-Time 클래스의 프로세스는 Time slice를 줄여주는 것이 무의미하고,
Normal 클래스의 프로세스의 경우에만 줄이도록 한정하는 것이 보다 더 논리에 맞는 것 같아 이렇게 구현을 하게 되었습니다.

```
432     int All_Waiting =0;
433     int All_Response =0;
434 
435     list_for_each_entry(cur, &job_q, job) // PID 큰거부터 순서대로 출력하기 위해 비어있는 노말큐를 사용.
436     {
437         list_add_tail(&cur->Normal,&Normal_q); //Job의 프로세스를 전부 노말큐에 넣음.
438     }
439 
440     // 각 프로세스별 정보 출력 : 이미 선언한 cur과 last 사용
441     while(!list_empty(&Normal_q)) //노말큐 빌때까지
442     {
443         last = list_entry(Normal_q.next, process, Normal);
444         list_for_each_entry(cur, &Normal_q, Normal)
445         {
446             if(cur->pid > last->pid)
447             {
448                 last = cur;
449             }
450         }                                   last->pid, last->arrival_time, last->code_bytes, last->Waiting, last->Response);
451         fprintf(stdout, "PID: %03d\tARRIVAL: %03d\tCODESIZE: %03d\tWAITING: %03d\tRESPONSE: %03d\n",
452         All_Waiting += last->Waiting;
453         All_Response += last->Response;
454         list_del_init(&last->Normal);
455     }
```

이전까지 while문은 코드가 다 끝났습니다. 즉 위의 코드는 모든 프로세스 작업을 종료한 이후의 코드입니다.
(450라인 의 코드는 451라인 코드의 뒷부분을 적어놓은 것입니다.)

JOTA 채점에서는 모든 작업 종료 후, 각 프로세스별 정보 출력을 해야 하는데, pid가 큰 거부터 순서대로 출력되는 것을 파악했습니다.

test 케이스로 주는 bin 파일은 프로세스의 pid가 낮은 거부터 저장되어 있는 것을 알고 있고, 그렇기에 과제 1-1 처럼 list_for_each_entry_reverse을 이용해 Job 큐를 역순회하며 프로세스 정보를 출력하면 PID가 큰 거부터 정보가 출력되는 것을 이미 알고 있었습니다.
그러나 이는 bin 파일에 의존적인 코드 구현이고 무엇보다 과제 1-1에서 해보았던 구현이니 다른 방법으로 코드를 짜보고 싶어 이런 형태로 코드를 작성하게 되었습니다.

일단 Job 큐의 프로세스를 모두 비어있는 Normal 큐에 옮깁니다. (435 - 438 라인)
그냥 Job 큐를 사용하지 않고 옮긴 이유는 Job 큐는 동적할당을 해제하는데 사용해야하기 때문입니다.
그 이후 Normal 큐를 순회하며 pid가 높은 프로세스순서대로 정보를 출력하였고 이때 총 waiting과 response 시간을 계산하였습니다. (441 - 455 라인)

```
457     //최종 리포트
458     fprintf(stdout, "*** TOTAL CLOCKS: %04d IDLE: %04d UTIL: %2.2f%% WAIT: %2.2f RESPONSE: %2.2f\n", clock, idle_time, (double)(clock-idle_time)*100/clock ,
459                                                                                                                 (double)All_Waiting/FinishProcess, (double)All_Response/FinishProcess);
460     // 리스트 제거, 할당 해제 : 이미 선언한 cur과 next 사용
461     list_for_each_entry_safe(cur, next, &job_q, job){
462         list_del(&cur->job);
463         free(cur->operations);
464         free(cur);
465     }
466
467
468     return 0;
469 }
```

그 이후 마지막으로 최종 리포트를 출력하고 동적할당해제를 Job 큐를 돌며 해주었습니다.
(459라인 의 코드는 458라인 코드의 뒷부분을 적어놓은 것입니다.)

실행 결과 :

```
ubuntu@201619460:~/hw2$ gcc -o os2 os2-1.c
ubuntu@201619460:~/hw2$ cat test2.bin | ./os2
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0002 CPU: Process is terminated PID:000 PC:001
0007 CPU: Switched      from: 000       to: 100
0007 CPU: Loaded PID: 001       Arrival: 004    Codesize: 002   PC: 000
0007 CPU: Loaded PID: 002       Arrival: 005    Codesize: 002   PC: 000
0007 CPU: Loaded PID: 080       Arrival: 006    Codesize: 002   PC: 000
0012 CPU: Switched      from: 100       to: 080
0022 CPU: Process is terminated PID:080 PC:001
0027 CPU: Switched      from: 080       to: 001
0040 CPU: Loaded PID: 081       Arrival: 040    Codesize: 002   PC: 000
0045 CPU: Switched      from: 001       to: 081
0055 CPU: Process is terminated PID:081 PC:001
0060 CPU: Switched      from: 081       to: 001
0217 CPU: Process is terminated PID:001 PC:001
0222 CPU: Switched      from: 001       to: 002
0302 CPU: Process is terminated PID:002 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000
PID: 081        ARRIVAL: 040    CODESIZE: 002   WAITING: 005    RESPONSE: 005
PID: 080        ARRIVAL: 006    CODESIZE: 002   WAITING: 006    RESPONSE: 006
PID: 002        ARRIVAL: 005    CODESIZE: 002   WAITING: 217    RESPONSE: 217
PID: 001        ARRIVAL: 004    CODESIZE: 002   WAITING: 043    RESPONSE: 023
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000
*** TOTAL CLOCKS: 0302 IDLE: 0030 UTIL: 90.07% WAIT: 54.20 RESPONSE: 50.20
```

과제 2-1

Submission of 2021운영체제 과제 2-1 by os201619460

View source
Resubmit

Compilation Warnings

```
oshw21c.c: In function 'main' :
oshw21c.c:162:3: warning: ignoring return value of 'fread' , declared with attribute warn_
   fread(next->operations,next->code_bytes,1,stdin); //코드 넣기
   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Execution Results

✔ ✔ ✔ ✔ ✔

Test case #1: AC [0.016s,784.00 KB] (2/2)
Test case #2: AC [0.019s,784.00 KB] (2/2)
Test case #3: AC [0.019s,784.00 KB] (2/2)
Test case #4: AC [0.019s,784.00 KB] (2/2)
Test case #5: AC [0.016s,784.00 KB] (2/2)

Resources: 0.090s, 784.00 KB
Final score: 10/10 (10.0/10 points)

과제 2-2

```
ubuntu@201619460:~/hw2$ gcc -o os2 os2-2.c
ubuntu@201619460:~/hw2$ cat test2.bin | ./os2
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0002 CPU: Process is terminated PID:000 PC:001
0007 CPU: Switched      from: 000       to: 100
0007 CPU: Loaded PID: 001       Arrival: 004    Codesize: 002   PC: 000
0007 CPU: Loaded PID: 002       Arrival: 005    Codesize: 002   PC: 000
0007 CPU: Loaded PID: 080       Arrival: 006    Codesize: 002   PC: 000
0012 CPU: Switched      from: 100       to: 080
0022 CPU: Process is terminated PID:080 PC:001
0027 CPU: Switched      from: 080       to: 002
0040 CPU: Loaded PID: 081       Arrival: 040    Codesize: 002   PC: 000
0045 CPU: Switched      from: 002       to: 081
0055 CPU: Process is terminated PID:081 PC:001
0060 CPU: Switched      from: 081       to: 002
0127 CPU: Process is terminated PID:002 PC:001
0132 CPU: Switched      from: 002       to: 001
0302 CPU: Process is terminated PID:001 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000
PID: 081        ARRIVAL: 040    CODESIZE: 002   WAITING: 005    RESPONSE: 005
PID: 080        ARRIVAL: 006    CODESIZE: 002   WAITING: 006    RESPONSE: 006
PID: 002        ARRIVAL: 005    CODESIZE: 002   WAITING: 042    RESPONSE: 022
PID: 001        ARRIVAL: 004    CODESIZE: 002   WAITING: 128    RESPONSE: 128
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000
*** TOTAL CLOCKS: 0302 IDLE: 0030 UTIL: 90.07% WAIT: 36.20 RESPONSE: 32.20
```

Submission of 2021운영체제 과제 2-2 by os201619460

View source
Resubmit

Compilation Warnings

```
oshw22c.c: In function 'main' :
oshw22c.c:162:3: warning: ignoring return value of 'fread' , declared with attribute warn_
   fread(next->operations,next->code_bytes,1,stdin); //코드 넣기
   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Execution Results

✔ ✔ ✔ ✔ ✔

Test case #1: AC [0.021s,776.00 KB] (2/2)
Test case #2: AC [0.020s,776.00 KB] (2/2)
Test case #3: AC [0.020s,776.00 KB] (2/2)
Test case #4: AC [0.018s,776.00 KB] (2/2)
Test case #5: AC [0.017s,776.00 KB] (2/2)

Resources: 0.096s, 776.00 KB
Final score: 10/10 (10.0/10 points)

과제 2-3

```
ubuntu@201619460:~/hw2$ gcc -o os2 os2-3.c
ubuntu@201619460:~/hw2$ cat test2.bin | ./os2
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0002 CPU: Process is terminated PID:000 PC:001
0007 CPU: Switched      from: 000       to: 100
0007 CPU: Loaded PID: 001       Arrival: 004    Codesize: 002   PC: 000
0007 CPU: Loaded PID: 002       Arrival: 005    Codesize: 002   PC: 000
0007 CPU: Loaded PID: 080       Arrival: 006    Codesize: 002   PC: 000
0012 CPU: Switched      from: 100       to: 080
0022 CPU: Process is terminated PID:080 PC:001
0027 CPU: Switched      from: 080       to: 001
0040 CPU: Loaded PID: 081       Arrival: 040    Codesize: 002   PC: 000
0045 CPU: Switched      from: 001       to: 081
0055 CPU: Process is terminated PID:081 PC:001
0060 CPU: Switched      from: 081       to: 001
0102 CPU: Switched      from: 001       to: 002
0152 CPU: ROUND ENDS. Recharge the Timeslices
0157 CPU: Switched      from: 002       to: 001
0212 CPU: Switched      from: 001       to: 002
0242 CPU: Process is terminated PID:002 PC:001
0242 CPU: ROUND ENDS. Recharge the Timeslices
0247 CPU: Switched      from: 002       to: 001
0297 CPU: ROUND ENDS. Recharge the Timeslices
0297 CPU: Not Switched  PID: 001
0317 CPU: Process is terminated PID:001 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000
PID: 081        ARRIVAL: 040    CODESIZE: 002   WAITING: 005    RESPONSE: 005
PID: 080        ARRIVAL: 006    CODESIZE: 002   WAITING: 006    RESPONSE: 006
PID: 002        ARRIVAL: 005    CODESIZE: 002   WAITING: 157    RESPONSE: 097
PID: 001        ARRIVAL: 004    CODESIZE: 002   WAITING: 143    RESPONSE: 023
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000
*** TOTAL CLOCKS: 0317 IDLE: 0045 UTIL: 85.80% WAIT: 62.20 RESPONSE: 26.20
```

Submission of 2021운영체제 과제 2-3 by os201619460

View source
Resubmit

Compilation Warnings

```
oshw23c.c: In function 'main' :
oshw23c.c:164:3: warning: ignoring return value of 'fread' , declared with attribute warn_u
  fread(next->operations,next->code_bytes,1,stdin); //코드 넣기
  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Execution Results

✔✔✔✔✔

❯ Test case #1: AC [0.017s,812.00 KB] (2/2)
❯ Test case #2: AC [0.015s,812.00 KB] (2/2)
❯ Test case #3: AC [0.017s,812.00 KB] (2/2)
❯ Test case #4: AC [0.018s,812.00 KB] (2/2)
❯ Test case #5: AC [0.011s,812.00 KB] (2/2)

과제 2-4

과제 2-4를 저는 보다 쉽게 하고자 스케줄링 테스트 프로그램을 제작하였습니다.
os2-4 코드가 생각보다 많이 길어서 실행하는 출력 값을 보며 이 코드의 작동 기전을 설명 드리고, 중요하다 생각되는 코드 부분만 보고서에 기재하도록 하겠습니다.

먼저 코드를 실행하면 다음과 같이 실험할 프로세스 개수를 입력받습니다.

```
ubuntu@201619460:~/hw2$ gcc -o os2 os2-4.c
ubuntu@201619460:~/hw2$ ./os2
실험할 프로세스의 개수 입력 :
```

프로세스 개수는 1~79까지만 입력이 가능한데, PID는 1부터 1씩 더하며 순서대로 생성되기 때문입니다.

```
ubuntu@201619460:~/hw2$ gcc -o os2 os2-4.c
ubuntu@201619460:~/hw2$ ./os2
실험할 프로세스의 개수 입력 : -3
0 ~ 80 사이의 양수를 입력하세요! : 99
0 ~ 80 사이의 양수를 입력하세요! : 0
0 ~ 80 사이의 양수를 입력하세요! :
```

교수님께서 Real-time class 프로세스는 없다고 가정하라고 과제에서 언급해 주셨기 때문에, 80이상의 PID를 생성하지 않도록 이렇게 구현하였습니다.
이에 대한 구현 코드는 다음과 같습니다.

```
fprintf(stdout, "실험할 프로세스의 개수 입력 : "); // 실험 프로세스 몇개할건지
scanf("%d",&answerNum);
while(getchar() != '\n');

while(answerNum < 1 || answerNum >= 80)
{
    fprintf(stdout, "0 ~ 80 사이의 양수를 입력하세요! : ");
    scanf("%d",&answerNum);
    while(getchar() != '\n');
}
```

특정 변수에 scanf를 통해 키보드 값을 입력받고, 그 변수가 예외의 값일 경우 다시 입력받도록 while문을 돌려줍니다. 대부분의 대답을 요하는 코드는 위와 거의 동일한 구조의 코드로 되어있으므로 앞으로 대답에 관련된 코드는 기재하지 않겠습니다.
프로세스의 개수를 입력하면 그 다음으로 다음과 같이 프로세스에 대한 정보를 직접 설정할 것인지 물어봅니다.
다음 페이지의 이미지 속 출력에도 나와 있듯이 Y를 누르면 내가 직접 프로세스를 설정하고, N을 누르면 랜덤으로 프로세스의 정보가 설정되게 됩니다.

```
ubuntu@201619460:~/hw2$ ./os2
실험할 프로세스의 개수 입력 : 3

프로세스를 직접 설정하시겠습니까? ( 아닐 시 랜덤 생성 )
Y/N :
```

```
프로세스를 직접 설정하시겠습니
Y/N : aaadasd
Y 또는 N으로 대답하세요! :
```

이 대답 또한 Y/N이 아닌 대답에 대해 예외처리가 되어있습니다.
여기서 만약 Y를 입력하게 된다면 다음과 같이 arrival_time과 명령어의 길이를 입력받습니다.

```
프로세스를 직접 설정하시겠습니까? ( 아닐 시 랜덤 생성 )
Y/N : Y

PDI: 0 프로세스의 arrival_time, 명령어 길이 입력 ( 음수 입력시 랜덤 ):
```

설명에도 나와 있듯이, 음수를 지정해주면 랜덤으로 설정할 수 있습니다.

```
PDI: 0 프로세스의 arrival_time, 명령어 길이 입력 ( 음수 입력시 랜덤 ): -1 -1
PDI: 1 프로세스의 arrival_time, 명령어 길이 입력 ( 음수 입력시 랜덤 ): 5 -3
PDI: 2 프로세스의 arrival_time, 명령어 길이 입력 ( 음수 입력시 랜덤 ): 9 6
```

만약 위와 같이 입력을 했다면, pid 0 은 도착시간과 명령어 길이가 랜덤,
pid 1 은 5 clock에 도착하고 명령어의 길이는 랜덤,
pid 2 는 9 clock에 도착하고 6의 길이를 가진 명령어가 됩니다.

여기에서 N을 선택하거나 음수를 지정하여 랜덤으로 프로세스 정보를 지정하는 건 오른쪽 코드와 같이 교수님께서 주신 os-gen-cpu.c 파일을 참고하여 작성하였습니다.

```
if(answer == 'N') //프로세스 랜덤 생성
{
    int i, prev_arrival=0;

    for(i=0; i<jobQnum; i++) {
        next = (process *) malloc(sizeof(process));
        next->pid = i;
        next->arrival_time = prev_arrival;
        prev_arrival += rand()%10;

        //next->code_bytes = 2+rand()%5*2;
        next->code_bytes = 2;

        next->operations = (code *) malloc(next->code_bytes);

        next->operations->op = 0;
        next->operations->len = 35+rand()%200;

        next->PC = 0;
        next->Waiting =0;
        next->Response =0;
        next->isFirst =true;
        next->Timeslice =0;
        next->ProvideTime = 0;
        next->TurnaroundTime = 0;

        INIT_LIST_HEAD(&next->job);
        INIT_LIST_HEAD(&next->Real_Time);
        INIT_LIST_HEAD(&next->Normal);
        INIT_LIST_HEAD(&next->Idle);   // 초기화

        list_add_tail(&next->job,&job_q); //잡큐에 넣기
    }
}
```

그 이후, 이 프로세스들을 갖고 실험하고 싶은 스케줄링을 고를 수 있는 선택지가 나옵니다.

```
테스트를 할 스케줄링 기법을 다음중에서 선택하세요.
 1. FIFO    2. SRJF(과제)    3.SRJF(수업)    4.Round-Robin
선택 :
```

여기에서 1,2,4 번은 과제에서 구현한 것과 동일한 스케줄링입니다. 4번은 뒤에서 설명하겠습니다만, 라운드 내에서의 스케줄링을 고를 수 있도록 구현되어 있습니다.

여기에서 3번 SRJF(수업)이 2번 SRJF(과제)와 무엇이 다른지 설명하겠습니다.
먼저 SRJF(과제)는 Normal큐 사이에서 preemption이 일어나지 않지만 SRJF(수업)은 일어나는 스케줄링입니다.
무슨 말이냐면,
예를 들어 다음과 같은 프로세스가 있다고 생각해 보겠습니다.
pid = 0 , arrival = 0 , operations Length = 50
pid = 1 , arrival = 3 , operations Length = 10

만약 2번, 즉 과제 2-2와 같은 SRJF(과제) 스케줄링이라면 아마 프로세스는 0 clock에서 pid=0 이 50 clock 까지 명령어를 완료 한 이후 Context switching 하여 55 clock에서 pid=1이 실행될 것입니다.

그러나 3번, 즉 SRJF(수업) 스케줄링은 매 클락 마다 최소 명령어의 프로세스를 탐색합니다.
즉, 0 clock에서 pid = 0 프로세스가 명령어를 시작하지만, pid=1 이 도착한 3 clock 시점에 다시 명령어가 작은 프로세스를 탐색하여 스케줄링 하는 것입니다.
3 clock을 기준으로 pid = 0 프로세스는 남은 명령어가 길이가 47 정도일 것이고, pid = 1 은 남은 명령어 길이가 10 이므로 Context switching이 일어나 pid = 1 프로세스가 실행이 됩니다.
3번 이름을 SRJF(수업) 으로 한 이유는 이 스케줄링이 수업시간에 교수님께서 말씀해주신 SRJF 스케줄링이기 때문입니다. 마찬가지로 2번 이름을 SRJF(과제)로 한 이유도 과제에서 구현한 SRJF 스케줄링이기 때문입니다.

이전 페이지에서 예시로 들은 프로세스 스케줄링을 제 프로그램을 통해 비교해 보겠습니다.

```
테스트 정보 출력
사용 스케줄링 : SRJF(과제)
프로세스 목록 ( arrival_time 기준 정렬 ):
PID: 000        ARRIVAL: 000    operations.len: 050     Timeslice: 000
PID: 100        ARRIVAL: 000    operations.len: 000     Timeslice: 000
PID: 001        ARRIVAL: 003    operations.len: 010     Timeslice: 000

 스케줄링 테스트 시작
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0003 CPU: Loaded PID: 001       Arrival: 003    Codesize: 002   PC: 000
0050 CPU: Process is terminated PID:000 PC:001
0055 CPU: Switched      from: 000       to: 001
0065 CPU: Process is terminated PID:001 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 001        ARRIVAL: 003    CODESIZE: 002   WAITING: 052    RESPONSE: 052   TURNAROUND: 062
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 050
*** TOTAL CLOCKS: 0065 IDLE: 0005 UTIL: 92.31% WAIT: 26.00 RESPONSE: 26.00
*** THROUGHPUT(100 CLOCKS): 3.08 TURNAROUND: 56.00
```

위의 출력은 2번 SRJF(과제) 스케줄링입니다. SRJF(과제)는 0 clock에 pid = 0의 명령어수행을 시작합니다. 왜냐면 큐에서 pid = 0 밖에 없기 때문에 제일 짧은 명령어이기 때문입니다. 그리고선 50 clock까지 명령어를 끝낼 때 까지 preemption이 없습니다. 명령어가 다 끝난 이후 55 clock에 pid = 1 로 Context switching이 일어나게 됩니다.

```
테스트 정보 출력
사용 스케줄링 : SRJF(수업)
프로세스 목록 ( arrival_time 기준 정렬 ):
PID: 000        ARRIVAL: 000    operations.len: 050     Timeslice: 000
PID: 100        ARRIVAL: 000    operations.len: 000     Timeslice: 000
PID: 001        ARRIVAL: 003    operations.len: 010     Timeslice: 000

 스케줄링 테스트 시작
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0003 CPU: Loaded PID: 001       Arrival: 003    Codesize: 002   PC: 000
0008 CPU: Switched      from: 000       to: 001
0018 CPU: Process is terminated PID:001 PC:001
0023 CPU: Switched      from: 001       to: 000
0070 CPU: Process is terminated PID:000 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 001        ARRIVAL: 003    CODESIZE: 002   WAITING: 005    RESPONSE: 005   TURNAROUND: 015
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 020    RESPONSE: 000   TURNAROUND: 070
*** TOTAL CLOCKS: 0070 IDLE: 0010 UTIL: 85.71% WAIT: 12.50 RESPONSE: 2.50
*** THROUGHPUT(100 CLOCKS): 2.86 TURNAROUND: 42.50
```

그에 반해 3번 SRJF(수업) 스케줄링은 동일하게 0 clock에 pid = 0의 명령어를 수행하지만, 매 클락 마다 명령어가 가장 작은 프로세스를 갱신합니다. 3 clock 에 pid = 1 이 도착하게 되고, 3 clock에서 비교할 때 가장 짧은 프로세스는 pid = 1 이므로 바로 Context switching이 일어나게 됩니다. 이 점이 SRJF(과제) 와의 차이점입니다. 즉, 명령어가 수행중일 때 preemption이 되는지 유무가 2번 3번의 큰 차이점이라 볼 수 있겠습니다.

그럼 다시 프로그램의 작동 방법으로 돌아오자면, 스케줄링 4번 : Round-Robin 을 선택 하게 되면 다음과 같이 추가 질문이 나오게 됩니다.

```
테스트를 할 스케줄링 기법을 다음중에서 선택하세요.
 1. FIFO    2. SRJF(과제)    3.SRJF(수업)    4.Round-Robin
선택 : 4

Round-Robin 스케줄링을 선택하셨습니다. Round 내에서의 스케줄링을 선택하세요.
 1. FIFO    2. SRJF
선택 :
```

추가적으로 말씀 드리면, Y/N 또는 숫자를 입력 받는 모든 코드 부분은 올바르지 않게 입력되는 경우에 따라 예외처리가 모두 구현되어 있습니다. 이후의 설명에서는 이에 대해서는 생략하도록 하겠습니다.

출력에도 나와 있듯이 4번 Round-Robin 스케줄링을 선택했으면 Round 내에서의 스케줄링을 추가적으로 선택할 수 있습니다. 1번 FIFO 는 2-3 과제와 동일하게 FIFO로 동작하고, 2번 SRJF는 교수님께서 과제에서 언급해주신 대로 명령어의 길이가 아닌, Time slice가 남아있는 게 가장 적은 프로세스를 고르도록 구현하였습니다.

```
Round-Robin 스케줄링을 선택하셨습니다. Round 내에서의 스케줄링을 선택하세요.
 1. FIFO    2. SRJF
선택 : 1

Round-Robin 스케줄링에 쓰일 공통 time slice을 입력해주세요. : 50

혹시 부여하는 time slice를 다르게 할 프로세스가 있나요? Y/N :
```

Round 속 스케줄링을 골랐다면 그 이후에는 time slice를 설정하고, 특정 pid에 time slice를 지정해줄 수 있습니다. 위 질문에 N을 답한다면 모든 프로세스가 동일한 time slice를 갖게 되고,
Y를 답하게 된다면, 다음과 같이 특정 프로세스에 대해 매 라운드당 주어지는 Time slice를 변경할 수 있습니다.

```
Round-Robin 스케줄링에 쓰일 공통 time slice을 입력해주세요. : 50

혹시 부여하는 time slice를 다르게 할 프로세스가 있나요? Y/N : Y

새로 설정할 프로세스의 개수를 입력해주세요. : 1
새로 설정할 프로세스 PID 와 새로 설정할 Timeslice 입력 : 99 -3
0 ~ 2 의 숫자로 PID를 설정해주세요! 1 이상의 Timeslice를 제공해주세요!
새로 설정할 프로세스 PID 와 새로 설정할 Timeslice 입력 : 0 40

테스트 정보 출력
사용 스케줄링 : Round-Robin : FIFO
프로세스 목록 ( arrival_time 기준 정렬 ):
PID: 000        ARRIVAL: 000    operations.len: 138     Timeslice: 040
PID: 100        ARRIVAL: 000    operations.len: 000     Timeslice: 050
PID: 001        ARRIVAL: 001    operations.len: 133     Timeslice: 050
PID: 002        ARRIVAL: 009    operations.len: 139     Timeslice: 050
```

위 출력의 테스트 정보 출력 란을 보면 0번 프로세스의 Time slice가 변경 된 것을 알 수 있습니다.
이 처럼 특정 프로세스의 Time slice를 바꾸는 것도 가능하게 구현하였습니다.
이렇게 스케줄링과 프로세스를 모두 설정하게 되면 스케줄링을 테스트 하게 됩니다.
테스트가 다 끝나고서, 다음과 같이 출력됩니다.

```
Test 완료. 같은 프로세스로 다시 실험하시겠습니까? :
```

```
ubuntu@201619460:~/hw2$ ./os2
실험할 프로세스의 개수 입력 : 2

프로세스를 직접 설정하시겠습니까? ( 아닐 시 랜덤 생성 )
Y/N : N

테스트를 할 스케줄링 기법을 다음중에서 선택하세요.
 1. FIFO    2. SRJF(과제)    3.SRJF(수업)    4.Round-Robin
선택 : 4

Round-Robin 스케줄링을 선택하셨습니다. Round 내에서의 스케줄링을 선택하세요.
 1. FIFO    2. SRJF
선택 : 1

Round-Robin 스케줄링에 쓰일 공통 time slice을 입력해주세요. : 50

혹시 부여하는 time slice를 다르게 할 프로세스가 있나요? Y/N : Y

새로 설정할 프로세스의 개수를 입력해주세요. : 1
새로 설정할 프로세스 PID 와 새로 설정할 Timeslice 입력 : 0 30

테스트 정보 출력
사용 스케줄링 : Round-Robin : FIFO
프로세스 목록 ( arrival_time 기준 정렬 ):
PID: 000        ARRIVAL: 000    operations.len: 042     Timeslice: 030
PID: 100        ARRIVAL: 000    operations.len: 000     Timeslice: 050
PID: 001        ARRIVAL: 007    operations.len: 082     Timeslice: 050

 스케줄링 테스트 시작
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0007 CPU: Loaded PID: 001       Arrival: 007    Codesize: 002   PC: 000
0035 CPU: Switched      from: 000       to: 001
0085 CPU: ROUND ENDS. Recharge the Timeslices
0090 CPU: Switched      from: 001       to: 000
0102 CPU: Process is terminated PID:000 PC:001
0107 CPU: Switched      from: 000       to: 001
0139 CPU: Process is terminated PID:001 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 001        ARRIVAL: 007    CODESIZE: 002   WAITING: 050    RESPONSE: 028   TURNAROUND: 132
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 060    RESPONSE: 000   TURNAROUND: 102
*** TOTAL CLOCKS: 0139 IDLE: 0015 UTIL: 89.21% WAIT: 55.00 RESPONSE: 14.00
*** THROUGHPUT(100 CLOCKS): 1.44 TURNAROUND: 117.00

Test 완료. 같은 프로세스로 다시 실험하시겠습니까? : Y

테스트를 할 스케줄링 기법을 다음중에서 선택하세요.
 1. FIFO    2. SRJF(과제)    3.SRJF(수업)    4.Round-Robin
선택 : 1
                •
                •
                •

Test 완료. 같은 프로세스로 다시 실험하시겠습니까? : N

테스트 프로그램 종료
```

여기에서 Y를 입력하면 프로세스를 동일하게 스케줄링만 다시 설정하여 테스트를 할 수 있게 되고 N을 입력하면 프로그램이 종료하게 됩니다.

왼쪽 사진은 대략적인 전체 테스트 출력입니다.

이처럼 이 프로그램은 원하는 동일한 프로세스를 다른 스케줄링으로 돌려볼 수 있도록 구현하였습니다.
이 프로그램의 스케줄링 코드는 다음과 같이 기존 과제 2-3에서 Normal 큐를 스케줄링 할 때, 스케줄링을 어떤 기법으로 하는지 변수에 저장하여 이에 따라 Switch문을 사용해 분기 하도록 구현하였습니다.

```
//스케줄링
if(list_empty(&Real_Time_q)) // Real_Time 큐가 비어있다면.
{
    if(list_empty(&Normal_q)) // Real_Time 큐가 비어있고 Normal 큐도 비어있다면.
    {
        cur = list_entry(Idle_q.next, process, Idle); // Idle 프로세스 가져오기
    }
    else  // Real_Time 큐가 비어있으나 Normal 큐 프로세스 있다면.
    {
        switch(scheduling)
        {
            case 1 :  //FIFO
                cur = list_entry(Normal_q.next, process, Normal); // Normal 큐에서 맨 앞 프로세스 가져오기
                break;
            case 2 :  //SRJF(과제)
                if( clock >= endoptime ) // 수행중인 명령어가 있다면 Normal 큐에서 스케줄링 하지 않음.
                {
                    if(cur->PC >= cur->code_bytes/2 && clock != 0)
                    {
                        list_del_init(&cur->Normal); // 프로세스 명령어 다 수행 끝났으니 큐에서 삭제

                        fprintf(stdout, "%04d CPU: Process is terminated\tPID:%03d\tPC:%03d\n" , clock, cur->pid , cur->PC ); //
                        FinishProcess++; // 종료 개수 1개증가.
                        cur->TurnaroundTime = clock - cur->arrival_time;
                    }
                    next = list_entry(Normal_q.next, process, Normal);
                    list_for_each_entry(cur, &Normal_q, Normal)
                    {
```

마지막으로, 기존 2-3 과제와 다른 점은 스케줄링 성능 분석 및 평가를 좀 더 용이하게 위해 결과 값 출력에 Throughput 과 turnaround time을 추가로 구현해 주었습니다.

Throughput은 100 clock당 몇 개의 프로세스를 완료하는지 마지막 종합 출력 값에 나타냈습니다. 이는 총 완료한 프로세스를 총 걸린 clock으로 나누고, 그걸 100을 곱하여 계산하였습니다.
turnaround time 은 각 프로세스가 스케줄링이 종료 된 후 정보를 출력할 때 추가로 출력하고, 마지막 종합 출력 값에도 평균을 계산하여 출력하도록 구현하였습니다.
개별적으로 프로세스가 arrive 한 시간과 terminated 될 때의 clock 차를 turnaround time에 저장하고, 모든 프로세스의 turnaround time 합을 프로세스로 나누어 평균값을 계산했습니다.

```
PID: 001        ARRIVAL: 007    CODESIZE: 002   WAITING: 040    RESPONSE: 040   TURNAROUND: 122
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 042
*** TOTAL CLOCKS: 0129 IDLE: 0005 UTIL: 96.12% WAIT: 20.00 RESPONSE: 20.00
*** THROUGHPUT(100 CLOCKS): 1.55 TURNAROUND: 82.00
```

이 외에도 이 보고서에서는 간략하게 하고자 코드는 올리지 않지만, test2.bin과 동일한 프로세스를 이용하여 스케줄링을 바꾸어 해보는 프로그램도 제작하였습니다. (Real_time 구현) 압축파일에 들어있는 2-4test2.c를 실행하여 JOTA와 동일한 조건을 선택하게 된다면, 결과 값이 동일하게 나오는 것을 확인 할 수 있습니다.

스케줄링 성능 분석 및 평가

1. FIFO vs SRJF(과제) vs SRIF(수업)

일단 먼저 랜덤으로 프로세스를 만들어서 이 세 스케줄링을 비교해 보겠습니다.

```
프로세스 목록 ( arrival_time 기준 정렬 ):
PID: 000        ARRIVAL: 000    operations.len: 095     Timeslice: 000
PID: 100        ARRIVAL: 000    operations.len: 000     Timeslice: 000
PID: 001        ARRIVAL: 008    operations.len: 086     Timeslice: 000
PID: 002        ARRIVAL: 016    operations.len: 070     Timeslice: 000
```

랜덤으로 만들어진 프로세스 목록입니다.

```
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0008 CPU: Loaded PID: 001       Arrival: 008    Codesize: 002   PC: 000
0016 CPU: Loaded PID: 002       Arrival: 016    Codesize: 002   PC: 000
0095 CPU: Process is terminated PID:000 PC:001
0100 CPU: Switched      from: 000       to: 001
0186 CPU: Process is terminated PID:001 PC:001
0191 CPU: Switched      from: 001       to: 002
0261 CPU: Process is terminated PID:002 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 002        ARRIVAL: 016    CODESIZE: 002   WAITING: 175    RESPONSE: 175   TURNAROUND: 245
PID: 001        ARRIVAL: 008    CODESIZE: 002   WAITING: 092    RESPONSE: 092   TURNAROUND: 178
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 095
*** TOTAL CLOCKS: 0261 IDLE: 0010 UTIL: 96.17% WAIT: 89.00 RESPONSE: 89.00
*** THROUGHPUT(100 CLOCKS): 1.15 TURNAROUND: 172.67
```

FIFO

FIFO

CLOCKS: 0261 IDLE: 0010 UTIL: 96.17%

WAIT: 89.00 RESPONSE: 89.00

THROUGHPUT: 1.15 TURNAROUND: 172.67

```
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0008 CPU: Loaded PID: 001       Arrival: 008    Codesize: 002   PC: 000
0016 CPU: Loaded PID: 002       Arrival: 016    Codesize: 002   PC: 000
0095 CPU: Process is terminated PID:000 PC:001
0100 CPU: Switched      from: 000       to: 002
0170 CPU: Process is terminated PID:002 PC:001
0175 CPU: Switched      from: 002       to: 001
0261 CPU: Process is terminated PID:001 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 002        ARRIVAL: 016    CODESIZE: 002   WAITING: 084    RESPONSE: 084   TURNAROUND: 154
PID: 001        ARRIVAL: 008    CODESIZE: 002   WAITING: 167    RESPONSE: 167   TURNAROUND: 253
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 095
*** TOTAL CLOCKS: 0261 IDLE: 0010 UTIL: 96.17% WAIT: 83.67 RESPONSE: 83.67
*** THROUGHPUT(100 CLOCKS): 1.15 TURNAROUND: 167.33
```

SRJF(과제)

SRJF(과제)

CLOCKS: 0261 IDLE: 0010 UTIL: 96.17%

WAIT: 83.67 RESPONSE: 83.67

THROUGHPUT: 1.15 TURNAROUND: 167.33

```
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0008 CPU: Loaded PID: 001       Arrival: 008    Codesize: 002   PC: 000
0013 CPU: Switched      from: 000       to: 001
0016 CPU: Loaded PID: 002       Arrival: 016    Codesize: 002   PC: 000
0021 CPU: Switched      from: 001       to: 002
0091 CPU: Process is terminated PID:002 PC:001
0096 CPU: Switched      from: 002       to: 001
0179 CPU: Process is terminated PID:001 PC:001
0184 CPU: Switched      from: 001       to: 000
0271 CPU: Process is terminated PID:000 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 002        ARRIVAL: 016    CODESIZE: 002   WAITING: 005    RESPONSE: 005   TURNAROUND: 075
PID: 001        ARRIVAL: 008    CODESIZE: 002   WAITING: 085    RESPONSE: 005   TURNAROUND: 171
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 176    RESPONSE: 000   TURNAROUND: 271
*** TOTAL CLOCKS: 0271 IDLE: 0020 UTIL: 92.62% WAIT: 88.67 RESPONSE: 3.33
*** THROUGHPUT(100 CLOCKS): 1.11 TURNAROUND: 172.33
```

SRJF(수업)

SRJF(수업)

CLOCKS: 0271 IDLE: 0020 UTIL: 92.62%

WAIT: 88.67 RESPONSE: 3.33

THROUGHPUT: 1.11 TURNAROUND: 172.33

일단 공통적으로 FIFO에 비해 SRJF가 RESPONSE와 WAIT, TURNAROUND 가 낮아진 것을 확인할 수 있습니다. 이것의 의미는 프로세스의 민감성(responsiveness)이 증가했다는 뜻입니다. 이건 시뮬레이터이므로 약간 오차도 있겠으나 민감성 뿐 아니라 SRIF 스케줄링은 FIFO에 비해 적어도 이 결과에선 모두 효능적으로 개선된 결과가 나왔습니다. SRIF는 이상적인 스케줄링이지만, 실제 구현이 어려운 스케줄링입니다.

이번에는 특정 상황을 직접 설정하여 스케줄링을 비교해보겠습니다.


```
프로세스 목록 ( arrival_time 기준 정렬 ):
PID: 000        ARRIVAL: 000    operations.len: 024     Timeslice: 000
PID: 100        ARRIVAL: 000    operations.len: 000     Timeslice: 000
PID: 001        ARRIVAL: 001    operations.len: 003     Timeslice: 000
PID: 002        ARRIVAL: 002    operations.len: 003     Timeslice: 000
```

(9주차 강의에 나온 프로세스)

각 프로세스가 명령어 실행 중간에 도착하도록 시간을 지정해 주었습니다. (강의영상에선 0초에 모두 동시 도착)

```
 스케줄링 테스트 시작
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0001 CPU: Loaded PID: 001       Arrival: 001    Codesize: 002   PC: 000
0002 CPU: Loaded PID: 002       Arrival: 002    Codesize: 002   PC: 000
0024 CPU: Process is terminated PID:000 PC:001
0029 CPU: Switched      from: 000       to: 001
0032 CPU: Process is terminated PID:001 PC:001
0037 CPU: Switched      from: 001       to: 002
0040 CPU: Process is terminated PID:002 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 002        ARRIVAL: 002    CODESIZE: 002   WAITING: 035    RESPONSE: 035   TURNAROUND: 038
PID: 001        ARRIVAL: 001    CODESIZE: 002   WAITING: 028    RESPONSE: 028   TURNAROUND: 031
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 024
*** TOTAL CLOCKS: 0040 IDLE: 0010 UTIL: 75.00% WAIT: 21.00 RESPONSE: 21.00
*** THROUGHPUT(100 CLOCKS): 7.50 TURNAROUND: 31.00
```

FIFO

FIFO

CLOCKS: 0040 IDLE: 0010 UTIL: 75.00%

WAIT: 21.00 RESPONSE: 21.00 THROUGHPUT: 7.50

TURNAROUND: 31.00

```
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0001 CPU: Loaded PID: 001       Arrival: 001    Codesize: 002   PC: 000
0002 CPU: Loaded PID: 002       Arrival: 002    Codesize: 002   PC: 000
0024 CPU: Process is terminated PID:000 PC:001
0029 CPU: Switched      from: 000       to: 001
0032 CPU: Process is terminated PID:001 PC:001
0037 CPU: Switched      from: 001       to: 002
0040 CPU: Process is terminated PID:002 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 002        ARRIVAL: 002    CODESIZE: 002   WAITING: 035    RESPONSE: 035   TURNAROUND: 038
PID: 001        ARRIVAL: 001    CODESIZE: 002   WAITING: 028    RESPONSE: 028   TURNAROUND: 031
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 024
*** TOTAL CLOCKS: 0040 IDLE: 0010 UTIL: 75.00% WAIT: 21.00 RESPONSE: 21.00
*** THROUGHPUT(100 CLOCKS): 7.50 TURNAROUND: 31.00
```

SRJF(과제)

SRJF(과제)

CLOCKS: 0040 IDLE: 0010 UTIL: 75.00%

WAIT: 21.00 RESPONSE: 21.00

THROUGHPUT: 7.50 TURNAROUND: 31.00

SRJF(수업)

CLOCKS: 0045 IDLE: 0015 UTIL: 66.67%

WAIT: 12.67 RESPONSE: 5.67

THROUGHPUT: 6.67 TURNAROUND: 22.67

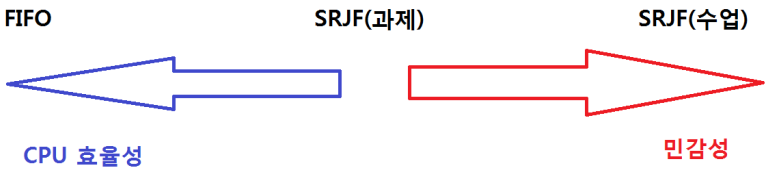
이번에는 결과가 이전과는 조금 다르게 나왔습니다.

```
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0001 CPU: Loaded PID: 001       Arrival: 001    Codesize: 002   PC: 000
0006 CPU: Switched      from: 000       to: 001
0006 CPU: Loaded PID: 002       Arrival: 002    Codesize: 002   PC: 000
0009 CPU: Process is terminated PID:001 PC:001
0014 CPU: Switched      from: 001       to: 002
0017 CPU: Process is terminated PID:002 PC:001
0022 CPU: Switched      from: 002       to: 000
0045 CPU: Process is terminated PID:000 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 002        ARRIVAL: 002    CODESIZE: 002   WAITING: 012    RESPONSE: 012   TURNAROUND: 015
PID: 001        ARRIVAL: 001    CODESIZE: 002   WAITING: 005    RESPONSE: 005   TURNAROUND: 008
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 021    RESPONSE: 000   TURNAROUND: 045
*** TOTAL CLOCKS: 0045 IDLE: 0015 UTIL: 66.67% WAIT: 12.67 RESPONSE: 5.67
*** THROUGHPUT(100 CLOCKS): 6.67 TURNAROUND: 22.67
```

SRJF(수업)

FIFO와 SRJF(과제)가 완전히 동일하고, SRJF(수업)은 오히려 CLOCK이 늘었습니다. 이는 SRJF(과제)와 SRJF(수업)의 갱신 조건 때문에 발생한 일입니다. SRJF(과제)는 어느 특정 프로세스가 작동하고 있으면 preemption이 불가능 하므로 그 명령어가 끝날 때까지 실행합니다. 만약 우연히 프로세스가 동작을 하고 있을 때만 프로세스가 순서대로 1개씩 들어오게 된다면, 이는 FIFO와 다를 바 없이 동작할 것입니다. 그에 반해 SRJF(수업)은 명령어가 실행되는 것과 상관없이 매 클락 마다 명령어가 적은 프로세스를 탐색하므로 Response 시간을 효율적으로 줄 일 수 있습니다. 그러나 이번 실험에서 Clock이 FIFO에 비해 늘어 난 것처럼, 더 많은 Context Switching로 인해 Response(waiting)은 감소할지라도 CPU 효율이 감소될 수 있습니다.
즉 FIFO는 CPU의 효율이 극대화 되는 스케줄링이고, 상대적으로 SRIF는 FIFO에 비해 CPU 효율은 Context Switching등으로 인해 조금 감소할 수 있을지라도, 프로세스의 민감성을 높여주는 스케줄링임을 알 수 있습니다.

FIFO는 CPU효율성을 중시하고,
SRJF(수업)은 민감성을 중시하고,
SRJF(과제)는 민감성을 FIFO보다 중시하되,
SRJF(수업) 보다는 약간 덜 중시한다.
라고 이해했습니다.



2. Round-Robin

2-1. Time slice.
- 공통 크기.

먼저 Round-robin 스케줄링의 time slice 크기가 어떤 영향을 미치는지 실험해보겠습니다.
무작위로 프로세스를 생성하고, Round 안에서의 스케줄링은 FIFO로 한 채 Time slice의 크기만 바꿔보겠습니다.

사용 스케줄링과 프로세스는
다음과 같습니다.

```
사용 스케줄링 : Round-Robin : FIFO
프로세스 목록 ( arrival_time 기준 정렬 ):
PID: 000        ARRIVAL: 000    operations.len: 051     Timeslice: 050
PID: 001        ARRIVAL: 000    operations.len: 111     Timeslice: 050
PID: 100        ARRIVAL: 000    operations.len: 000     Timeslice: 050
PID: 002        ARRIVAL: 001    operations.len: 085     Timeslice: 050
PID: 003        ARRIVAL: 004    operations.len: 169     Timeslice: 050
PID: 004        ARRIVAL: 006    operations.len: 099     Timeslice: 050
```

```
 스케줄링 테스트 시작
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 001       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0001 CPU: Loaded PID: 002       Arrival: 001    Codesize: 002   PC: 000
0004 CPU: Loaded PID: 003       Arrival: 004    Codesize: 002   PC: 000
0006 CPU: Loaded PID: 004       Arrival: 006    Codesize: 002   PC: 000
0051 CPU: Process is terminated PID:000 PC:001
0056 CPU: Switched      from: 000       to: 001
0161 CPU: Switched      from: 001       to: 002
0246 CPU: Process is terminated PID:002 PC:001
0251 CPU: Switched      from: 002       to: 003
0356 CPU: Switched      from: 003       to: 004
0455 CPU: Process is terminated PID:004 PC:001
0455 CPU: ROUND ENDS. Recharge the Timeslices
0460 CPU: Switched      from: 004       to: 001
0471 CPU: Process is terminated PID:001 PC:001
0476 CPU: Switched      from: 001       to: 003
0545 CPU: Process is terminated PID:003 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 004        ARRIVAL: 006    CODESIZE: 002   WAITING: 350    RESPONSE: 350   TURNAROUND: 449
PID: 003        ARRIVAL: 004    CODESIZE: 002   WAITING: 372    RESPONSE: 247   TURNAROUND: 541
PID: 002        ARRIVAL: 001    CODESIZE: 002   WAITING: 160    RESPONSE: 160   TURNAROUND: 245
PID: 001        ARRIVAL: 000    CODESIZE: 002   WAITING: 360    RESPONSE: 056   TURNAROUND: 471
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 051
*** TOTAL CLOCKS: 0545 IDLE: 0030 UTIL: 94.50% WAIT: 248.40 RESPONSE: 162.60
*** THROUGHPUT(100 CLOCKS): 0.92 TURNAROUND: 351.40
```

```
 스케줄링 테스트 시작
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002
0000 CPU: Loaded PID: 001       Arrival: 000    Codesize: 002
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002
0001 CPU: Loaded PID: 002       Arrival: 001    Codesize: 002
0004 CPU: Loaded PID: 003       Arrival: 004    Codesize: 002
0006 CPU: Loaded PID: 004       Arrival: 006    Codesize: 002
0051 CPU: Process is terminated PID:000 PC:001
0056 CPU: Switched      from: 000       to: 001
0167 CPU: Process is terminated PID:001 PC:001
0172 CPU: Switched      from: 001       to: 002
0257 CPU: Process is terminated PID:002 PC:001
0262 CPU: Switched      from: 002       to: 003
0417 CPU: Switched      from: 003       to: 004
0516 CPU: Process is terminated PID:004 PC:001
0516 CPU: ROUND ENDS. Recharge the Timeslices
0521 CPU: Switched      from: 004       to: 003
0540 CPU: Process is terminated PID:003 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 004        ARRIVAL: 006    CODESIZE: 002   WAITING: 411    RESPONSE: 411   TURNAROUND: 510
PID: 003        ARRIVAL: 004    CODESIZE: 002   WAITING: 367    RESPONSE: 258   TURNAROUND: 536
PID: 002        ARRIVAL: 001    CODESIZE: 002   WAITING: 171    RESPONSE: 171   TURNAROUND: 256
PID: 001        ARRIVAL: 000    CODESIZE: 002   WAITING: 056    RESPONSE: 056   TURNAROUND: 167
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 051
*** TOTAL CLOCKS: 0540 IDLE: 0025 UTIL: 95.37% WAIT: 201.00 RESPONSE: 179.20
*** THROUGHPUT(100 CLOCKS): 0.93 TURNAROUND: 304.00
```

각 타임 슬라이스에 따른 결과는 다음과 같습니다.

Time Slice	Clocks	Idle	Util	Wait	Response	Throughput	turnaround
10	755	240	68.21%	506.40	27.80	0.66	609.40
50	570	55	90.35%	333.40	107.80	0.88	436.40
100	545	30	94.50%	248.40	162.60	0.92	351.40
150	540	25	95.37%	201.00	179.20	0.93	304.00
200	535	20	96.26%	183.00	183.00	0.93	286.00
<control> FIFO	535	20	96.26%	183.00	183.00	0.93	286.00

파란색 박스는 Time Slice가 작아짐에 따라 수치가 커지는 것, 빨간 박스는 그 반대를 의미합니다.
동일한 프로세스들 간 스케줄링에서, 타임 슬라이스가 커짐에 따라, Clock수와 idle 수치는 서서히 감소하고, 반대로 Wait 시간과 Response 시간은 증가했습니다. 그리고 프로세스의 최대 명령어 길이(pid = 3 인 프로세스. 명령어 길이 169)보다 긴 200 만큼 Time slice를 제공하자, FIFO와 완전히 동일한 성능을 보였습니다. 이를 통해 Round Robin 스케줄링을 할 때 Time slice가 커질수록 높은 CPU활용도를 주지만, 그 만큼 Context Swiching이 일어나지 않아 프로세스들의 Waiting 및 Response가 감소하게 되므로 민감성, 공정성이 떨어진다고 볼 수 있습니다.

- 개별 크기

이번에는 다른 공통된 time slice가아닌, 개별의 time slice 크기가 어떤 영향을 미치는지 해보도록 하겠습니다.
동일하게, 무작위 프로세스를 생성하고, FIFO 로 Round를 스케줄링 하여 Round-robin 스케줄링을 테스트 해보겠습니다. 이번에는 다른 프로세스는 Time slice를 고정 수치로 두고, 특정 프로세스만 time slice를 다르게 부여해 보겠습니다.

사용 프로세스는 오른쪽과 같습니다.

```
사용 스케줄링 : Round-Robin : FIFO
프로세스 목록 ( arrival_time 기준 정렬 ):
PID: 000        ARRIVAL: 000    operations.len: 171     Timeslice: 020
PID: 001        ARRIVAL: 000    operations.len: 086     Timeslice: 020
PID: 100        ARRIVAL: 000    operations.len: 000     Timeslice: 020
PID: 002        ARRIVAL: 005    operations.len: 227     Timeslice: 020
PID: 003        ARRIVAL: 005    operations.len: 051     Timeslice: 020
```

다른 프로세스들은 time slice를 20으로 고정하고, 1번 프로세스의 Time slice를 10, 20, 50으로 변경해보고 결과를 관찰해 보겠습니다.

pid = 1 의 time slice 가 10일 경우

```
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 003        ARRIVAL: 005    CODESIZE: 002   WAITING: 200    RESPONSE: 060   TURNAROUND: 251
PID: 002        ARRIVAL: 005    CODESIZE: 002   WAITING: 448    RESPONSE: 035   TURNAROUND: 675
PID: 001        ARRIVAL: 000    CODESIZE: 002   WAITING: 522    RESPONSE: 025   TURNAROUND: 608
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 426    RESPONSE: 000   TURNAROUND: 597
*** TOTAL CLOCKS: 0680 IDLE: 0145 UTIL: 78.68% WAIT: 399.00 RESPONSE: 30.00
*** THROUGHPUT(100 CLOCKS): 0.59 TURNAROUND: 532.75
```

pid = 1 의 time slice 가 20일 경우

```
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 003        ARRIVAL: 005    CODESIZE: 002   WAITING: 230    RESPONSE: 070   TURNAROUND: 281
PID: 002        ARRIVAL: 005    CODESIZE: 002   WAITING: 428    RESPONSE: 045   TURNAROUND: 655
PID: 001        ARRIVAL: 000    CODESIZE: 002   WAITING: 311    RESPONSE: 025   TURNAROUND: 397
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 417    RESPONSE: 000   TURNAROUND: 588
*** TOTAL CLOCKS: 0660 IDLE: 0125 UTIL: 81.06% WAIT: 346.50 RESPONSE: 35.00
*** THROUGHPUT(100 CLOCKS): 0.61 TURNAROUND: 480.25
```

pid = 1 의 time slice 가 50일 경우

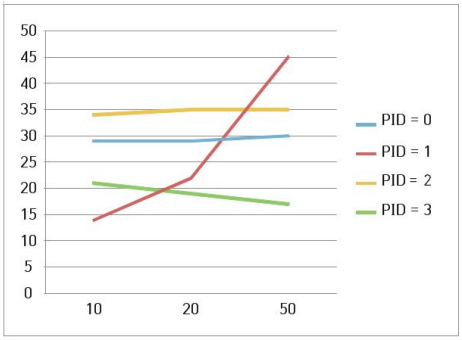
```
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 003        ARRIVAL: 005    CODESIZE: 002   WAITING: 251    RESPONSE: 100   TURNAROUND: 302
PID: 002        ARRIVAL: 005    CODESIZE: 002   WAITING: 413    RESPONSE: 075   TURNAROUND: 640
PID: 001        ARRIVAL: 000    CODESIZE: 002   WAITING: 105    RESPONSE: 025   TURNAROUND: 191
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 402    RESPONSE: 000   TURNAROUND: 573
*** TOTAL CLOCKS: 0645 IDLE: 0110 UTIL: 82.95% WAIT: 292.75 RESPONSE: 50.00
*** THROUGHPUT(100 CLOCKS): 0.62 TURNAROUND: 426.50
```

일단 먼저 전체 성능부터 먼저 보겠습니다. pid=1의 time slice가 늘어남에 따라, 전체 CPU의 효율성은 올라가고 Response 효능은 조금씩 떨어집니다. 이는 이전에 보았던 공통 Time slice의 크기 증가와 비슷하게 볼 수 있습니다. pid =1 프로세스의 Time slice를 늘려준다는 것은 결론적으로 전체 프로세스의 평균 Time slice 또한 늘려주는 것이므로 같은 효과가 나타나는 것입니다.
이젠 pid =1 프로세스입장으로 봐보겠습니다. Turnaround에서 waiting을 빼면 실질적으로 프로세스가 동작한 시간이니, 이를 통해 프로세스가 arrive 하고 terminated 되기까지의 시간 중 cpu를 점유한 시간이 어느 정도인지 파악할 수 있습니다.

오른쪽 표는 Turnaround에서 waiting을 뺀 값을 Turnaround에 비했을 때 몇 퍼센트인지 나타낸 것입니다.

pid	1번의 TS = 10	1번의 TS = 20	1번의 TS = 50
0	약 29%	약 29%	약 30%
1	약 14%	약 22%	약 45%
2	약 34%	약 35%	약 35%
3	약 21%	약 19%	약 17%

pid = 1 프로세스는 Time slice가 늘어날 때마다 눈에 띄게 %가 상승합니다. (가로 축)



이것 뿐 아니라 그저 단순히 Time slice에 따른 Turnaround의 변화만 관찰 하더라도
같은 길이의 명령어를 Time slice가 많이 부여 되었을수록
빠르게 처리하게 되는걸 알 수 있습니다.

2-1. Time slice.

Round-robin 스케줄링을 채용할 때, Round 안에서의 스케줄링 기법은 효능에 어떤 영향을 미치는지 , 그리고 그 스케줄링 기법에 비해 Round-robin은 그 스케줄링에게 어떤 영향을 미치는지 알아보도록 하겠습니다.

Time slice는 50으로 고정하고 랜덤으로 프로세스를 생성해 보겠습니다.

오른쪽 출력 값이 생성된 프로세스 입니다.

```
프로세스 목록 ( arrival_time 기준 정렬 ):
PID: 000        ARRIVAL: 000    operations.len: 163     Timeslice: 050
PID: 001        ARRIVAL: 000    operations.len: 224     Timeslice: 050
PID: 100        ARRIVAL: 000    operations.len: 000     Timeslice: 050
PID: 002        ARRIVAL: 009    operations.len: 118     Timeslice: 050
```

Round-Robin : FIFO

```
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 001       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0009 CPU: Loaded PID: 002       Arrival: 009    Codesize: 002   PC: 000
0055 CPU: Switched      from: 000       to: 001
0110 CPU: Switched      from: 001       to: 002
0160 CPU: ROUND ENDS. Recharge the Timeslices
0165 CPU: Switched      from: 002       to: 000
0220 CPU: Switched      from: 000       to: 001
0275 CPU: Switched      from: 001       to: 002
0325 CPU: ROUND ENDS. Recharge the Timeslices
0330 CPU: Switched      from: 002       to: 000
0385 CPU: Switched      from: 000       to: 001
0440 CPU: Switched      from: 001       to: 002
0458 CPU: Process is terminated PID:002 PC:001
0458 CPU: ROUND ENDS. Recharge the Timeslices
0463 CPU: Switched      from: 002       to: 000
0476 CPU: Process is terminated PID:000 PC:001
0481 CPU: Switched      from: 000       to: 001
0531 CPU: ROUND ENDS. Recharge the Timeslices
0531 CPU: Not Switched  PID: 001
0555 CPU: Process is terminated PID:001 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 002        ARRIVAL: 009    CODESIZE: 002   WAITING: 331    RESPONSE: 101   TURNAROUND: 449
PID: 001        ARRIVAL: 000    CODESIZE: 002   WAITING: 331    RESPONSE: 055   TURNAROUND: 555
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 313    RESPONSE: 000   TURNAROUND: 476
*** TOTAL CLOCKS: 0555 IDLE: 0050 UTIL: 90.99% WAIT: 325.00 RESPONSE: 52.00
*** THROUGHPUT(100 CLOCKS): 0.54 TURNAROUND: 493.33
```

Round-Robin : FIFO

CLOCK : 0555
IDLE : 50
UTIL : 90.99%
WAIT: 325.00
RESPONSE : 52.00
THROUGHPUT : 0.54
TURNAROUND : 493.33

Round-Robin : SRJF

CLOCK : 0560
IDLE : 55
UTIL : 90.18%
WAIT: 308.33
RESPONSE : 52.00
THROUGHPUT : 0.54
TURNAROUND : 476.67

Round-Robin : SRJF

```
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 001       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0009 CPU: Loaded PID: 002       Arrival: 009    Codesize: 002   PC: 000
0055 CPU: Switched      from: 001       to: 002
0110 CPU: Switched      from: 002       to: 000
0160 CPU: ROUND ENDS. Recharge the Timeslices
0165 CPU: Switched      from: 000       to: 002
0220 CPU: Switched      from: 002       to: 001
0275 CPU: Switched      from: 001       to: 000
0325 CPU: ROUND ENDS. Recharge the Timeslices
0330 CPU: Switched      from: 000       to: 002
0348 CPU: Process is terminated PID:002 PC:001
0353 CPU: Switched      from: 002       to: 001
0408 CPU: Switched      from: 001       to: 000
0458 CPU: ROUND ENDS. Recharge the Timeslices
0463 CPU: Switched      from: 000       to: 001
0518 CPU: Switched      from: 001       to: 000
0531 CPU: Process is terminated PID:000 PC:001
0531 CPU: ROUND ENDS. Recharge the Timeslices
0536 CPU: Switched      from: 000       to: 001
0560 CPU: Process is terminated PID:001 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 002        ARRIVAL: 009    CODESIZE: 002   WAITING: 221    RESPONSE: 046   TURNAROUND: 339
PID: 001        ARRIVAL: 000    CODESIZE: 002   WAITING: 336    RESPONSE: 000   TURNAROUND: 560
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 368    RESPONSE: 110   TURNAROUND: 531
*** TOTAL CLOCKS: 0560 IDLE: 0055 UTIL: 90.18% WAIT: 308.33 RESPONSE: 52.00
*** THROUGHPUT(100 CLOCKS): 0.54 TURNAROUND: 476.67
```

FIFO

```
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 001       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0009 CPU: Loaded PID: 002       Arrival: 009    Codesize: 002   PC: 000
0163 CPU: Process is terminated PID:000 PC:001
0168 CPU: Switched      from: 000       to: 001
0392 CPU: Process is terminated PID:001 PC:001
0397 CPU: Switched      from: 001       to: 002
0515 CPU: Process is terminated PID:002 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 002        ARRIVAL: 009    CODESIZE: 002   WAITING: 388    RESPONSE: 388   TURNAROUND: 506
PID: 001        ARRIVAL: 000    CODESIZE: 002   WAITING: 168    RESPONSE: 168   TURNAROUND: 392
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 163
*** TOTAL CLOCKS: 0515 IDLE: 0010 UTIL: 98.06% WAIT: 185.33 RESPONSE: 185.33
*** THROUGHPUT(100 CLOCKS): 0.58 TURNAROUND: 353.67
```

FIFO
CLOCK : 515 IDLE : 10
UTIL : 98.06%
WAIT : 185.33
RESPONSE : 185.33
THROUGHPUT : 0.58
TURNAROUND : 353.67

SRJF(과제)
CLOCK : 515
IDLE : 10
UTIL : 98.06%
WAIT : 150.00
RESPONSE : 150.00
THROUGHPUT : 0.58
TURNAROUND : 318.33

SRJF(과제)

```
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 001       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0009 CPU: Loaded PID: 002       Arrival: 009    Codesize: 002   PC: 000
0163 CPU: Process is terminated PID:000 PC:001
0168 CPU: Switched      from: 000       to: 002
0286 CPU: Process is terminated PID:002 PC:001
0291 CPU: Switched      from: 002       to: 001
0515 CPU: Process is terminated PID:001 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 002        ARRIVAL: 009    CODESIZE: 002   WAITING: 159    RESPONSE: 159   TURNAROUND: 277
PID: 001        ARRIVAL: 000    CODESIZE: 002   WAITING: 291    RESPONSE: 291   TURNAROUND: 515
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 163
*** TOTAL CLOCKS: 0515 IDLE: 0010 UTIL: 98.06% WAIT: 150.00 RESPONSE: 150.00
*** THROUGHPUT(100 CLOCKS): 0.58 TURNAROUND: 318.33
```

스케줄링	Clocks	Idle	Util	Wait	Response	Throughput	turnaround
RR:FIFO	555	50	90.99	325.00	52.00	0.54	493.33
RR:SRJF	560	55	90.18	308.33	52.00	0.54	476.67
FIFO	515	10	98.06	185.33	185.33	0.58	353.67
SRJF(과제)	515	10	98.06	150.00	050.00	0.58	318.33
SRJF(수업)	520	15	97.12%	143.00	100.33	0.58	311.33

각 스케줄링의 테스트에 따른 결과 값입니다.
먼저 Round-Robin부터 비교해 보면, 다른 조건이 동일할 시 Round Robin 스케줄링은 내부의 round 스케줄의 특성에 영향을 받는 것 같습니다. 과제 2-4의 1번에서 이야기 했듯이, 상대적으로 FIFO는 CPU효율에 중점을 (clock 대비 CPU 효율 정도), SRJP는 좀 더 공정한 작업처리에 중점을(waiting, response 등 감소) 두고 있는 특성을 관찰할 수 있었는데, 이들의 모습을 각각 채택한 R-R 스케줄링으로부터 미약하지만 살펴 볼 수 있었습니다. FIFO를 채택한 R-R 스케줄링은 Util이, SRJF를 채택한 R-R 스케줄링은 turnaround와 waiting time이 조금씩 효능이 더 좋은 것을 보면 알 수 있습니다.
그러면 반대로, FIFO와 SRJF의 관점에서는 Round-Robin 스케줄링에 채용된 다는 것이 어떤 의미인지 보겠습니다. 표를 보면 직관적이게도, Round-Robin 스케줄링과 그냥 스케줄링을 비교해 보았을 때 확연히 Round-Robin 스케줄링이 훨씬 CPU 효율이 떨어지고, Wait 타임이 높은 반면 Response는 개선된 것을 알 수 있습니다.
이와 관련 되서는 2-1 에서, Context Switching과 관련하여 Time slice를 늘릴수록 심해진 다는 것을 test해 보았습니다.

즉 Round-Robin 스케줄링은 Context Switching의 오버헤드가 커서 CPU사용율과 Wait타임(Idle)의 측면에선 효율이 떨어지는 반면, 그만큼 응답시간이 짧아지기 때문에 실시간 시스템에 유리한 스케줄링임을 알 수 있습니다.

이를 반대로 이야기하면 Context Switching에 의한 오버헤드를 적게 하여 CPU 활용도를 높이는 것이 FIFO스케줄링 이라 할 수 있겠고, 여기에 추가로 평균 대기시간을 최소화하기 위해 CPU 점유시간이 가장 짧은 프로세스를 탐색하는 것이 SRJF 스케줄링이라 할 수 있겠습니다.

SRJF(수업)

```
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 001       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0009 CPU: Loaded PID: 002       Arrival: 009    Codesize: 002   PC: 000
0014 CPU: Switched      from: 000       to: 002
0132 CPU: Process is terminated PID:002 PC:001
0137 CPU: Switched      from: 002       to: 000
0291 CPU: Process is terminated PID:000 PC:001
0296 CPU: Switched      from: 000       to: 001
0520 CPU: Process is terminated PID:001 PC:001
PID: 100        ARRIVAL: 000    CODESIZE: 002   WAITING: 000    RESPONSE: 000   TURNAROUND: 000
PID: 002        ARRIVAL: 009    CODESIZE: 002   WAITING: 005    RESPONSE: 005   TURNAROUND: 123
PID: 001        ARRIVAL: 000    CODESIZE: 002   WAITING: 296    RESPONSE: 296   TURNAROUND: 520
PID: 000        ARRIVAL: 000    CODESIZE: 002   WAITING: 128    RESPONSE: 000   TURNAROUND: 291
*** TOTAL CLOCKS: 0520 IDLE: 0015 UTIL: 97.12% WAIT: 143.00 RESPONSE: 100.33
*** THROUGHPUT(100 CLOCKS): 0.58 TURNAROUND: 311.33
```

os1 에서 수정사항 파악	O
PID에 따라 도착할 때 넣어지는 큐 분기 구현	O
실행할 프로세스 판단 분기 구현 - IDLE일 때, 아닐 때.	O
context switching 구현 (과제1 알고리즘이 약간 변화했음)	O
Preemption 구현 - process 포인터를 하나 더 선언.	O
코드 점검을 위한 PID 높은 순서 출력 구현	O
Response time 구현 - 프로세스에 추가 변수 첫 프로세스 실행 여부 판단하는 bool변수를 프로세스 내에 선언	O
Waiting time 구현 - 큐에는 있되 동작 안할 때	O
과제 2-1	O
FIFO 스케줄링을 SRJF 스케줄링으로 변경 -리스트 돌며 가장 적은 명령어 길이 탐색 단 명령어가 종료 되었을 때만	O
과제 2-2	O
Round-Robin scheduling 구현 - time slice 변수 프로세스 당 추가 - 라운드 구현 : time slice 변수가 모두 0일 때	O
Time slice를 clock 마다 1씩 감소 구현 - 예외처리 철저.	O
과제 2-3	O
프로세스 성능 비교를 쉽게 할 방법 구상	O
추가적으로 성능의 지표 더 구현할 요소 생각 - throughput - turnaround time - turnaround time 과 wait로 CPU효율 : 철회 (보고서에는 작성 완료)	O
프로그램 동작 방식 구상	O
추가적인 스케줄링 구상 (강의 9주차) - 강의버전 SRJF 탄생	O
실험 중간 중간 오류 수정	O
과제 2-4	O