

운영체제 과제1 : 멀티프로그래밍

201619460 이성규

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; //PC레지스터
132     int ioOpEnd; //wait_q에서 io명령어 종료시간 저장
133     code *operations; //code tuples 가 저장된 위치
134     struct list_head job, ready, wait;
135 } process;
136
137
138 int main(int argc, char* argv[]) {
139
140     process *cur, *next; // cur : 파일 읽은거 담을 구조체(프로세스) 포인터, next : 리스트속에 넣을
141                               실제 구조체로 파일담은 cur에서 옮겨 담을 포인터
142     LIST_HEAD(job_q); //잡 큐 리스트
143
144     cur = (process *) malloc(sizeof(process));
145
146     while(fread(cur, sizeof(int)*3, 1, stdin) == 1) { //프로세스 읽기, 실패시 while문 종료
147
148         next = (process *) malloc(sizeof(process));
149
150         next->pid = cur->pid;
151         next->arrival_time = cur->arrival_time;
152         next->code_bytes = cur->code_bytes; //옮기기
153         next->PC = 0;
154         next->ioOpEnd = 0;
```

먼저 코드입니다.

구조체 code와 process를 선언하여 자료를 관리하였고, 과제를 거치며 조금씩 구성 변수가 늘어났습니다.

프로세스 구조체는 PC레지스터의 역할을 하는 PC변수와 wait_q에서 io 명령어가 종료 되었을 때를 알리기 위한 ioOpEnd 변수가 있습니다.

위의 초반 코드 부분은 과제 0의 연장선인 코드로, 파일 담을 구조체를 동적으로 선언한 후 리눅스 List를 이용하여 관리하고 있습니다. 여기서, C언어의 구조체 메모리 구조 적재 법에 따라, while문에서 sizeof(int)*3 만큼의 크기만 받아 구조체의 정보를 받습니다.. 이에 관련된 고민했던 부분은 간략히 코드를 본 후 나중에 이야기 하겠습니

```
다. 157     next->operations = (code *) malloc(next->code_bytes); // 할당
158     fread(next->operations,next->code_bytes,1,stdin); //코드 넣기
159
160     INIT_LIST_HEAD(&next->job);
161     INIT_LIST_HEAD(&next->ready);
162     INIT_LIST_HEAD(&next->wait); // 초기화
163
164     list_add_tail(&next->job,&job_q); //잡큐에 넣기
165
166 }
167
168 cur->pid = 100;
169 cur->arrival_time = 0;
170 cur->code_bytes = 2;
171 cur->PC = 0;
172 cur->ioOpEnd = 0;
173 cur->operations = (code *) malloc(cur->code_bytes);
174 cur->operations[0].op = 0xFF;
175 cur->operations[0].len = 0;
176 INIT_LIST_HEAD(&cur->job);
177 INIT_LIST_HEAD(&cur->ready);
178 INIT_LIST_HEAD(&cur->wait);
179 list_add_tail(&cur->job,&job_q); // Idle 프로세스 초기화 후 잡큐에 추가
180
181
182 int clock = 0;
183 int idle_time = 0;
184 int jobQnum = 0; //잡큐에 들어온 프로세스 개수
185 int finishNum = 0; //레디큐에 들어왔다가 종료한 프로세스 개수
186 int cpuOpEnd = 0; //CPU명령어 끝나는 clock시간 저장
187 bool isworkCPU = true; //CPU가 일을 했는지
188 int runningPID = 0;
189 LIST_HEAD(ready_q); //레디큐 리스트
190 LIST_HEAD(wait_q); //웨이트큐 리스트
```

일단 먼저 과제 1-2의 부분이 었 던 Idle 프로세스를 job_q에 추가 하였습니다. (168 ~ 179)

그리고 또한 C언어에서 배열은 연속된 메모리 공간을 이용하고, 사실상 배열명은 포인터와 매우 유사하고 비슷한 특징인 것을 이용하여, 구조체의 code 포인터는 인덱스를 이용하여 배열처럼 사용하였습니다. (174~175)

아래 182~ 190라인 에서는 추가적으로 변수를 선언했습니다. 간략한 주석이 있으며, 주석이 없는 clock은 시간, idle_time은 프로세스의 효율성 판단을 위한 IDLE 상태의 변수, 그리고 현재 작동되는 프로세스의 PID를 의미하는 runningPID입니다.

```

192 list_for_each_entry(cur, &job_q, job){
193     jobQnum++;
194 }
195
196 while( jobQnum != finishNum || cpuOpEnd >= clock ) // 잡류와 레디큐 들어온 프로세스 개수가 동일, 현재 프로세스가 다 끝날때까지, clock수 늘림
197 {
198
199     // 도착하면 레디큐에 넣음.
200     list_for_each_entry(cur, &job_q, job){
201
202         if( cur->arrival_time == clock){ //도착시간이 clock과 같다면
203
204             list_add_tail(&cur->ready,&ready_q); //레디큐에 프로세스 추가
205             fprintf(stdout, "%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n", clock, cur->pid, cur->arrival_time,
206                                     cur->arrival_time, cur->code_bytes, cur->PC); //로드된 프로세스 정보 출력
207         }
208     }
209
210     // wait_q에서 io명령어 다하면 ready_q로 이동
211     list_for_each_entry_safe(cur, next, &wait_q, wait){
212
213         if( cur->ioOpEnd <= clock){ //io 명령어 수행 완료하면
214             //wait_q에서 ready_q로 옮김
215             list_del_init(&cur->wait);
216             list_add_tail(&cur->ready,&ready_q);
217             fprintf(stdout, "%04d IO : COMPLETED! PID: %03d\tIOTIME: %03d\tPC: %03d\n", clock, cur->pid, clock, cur->PC); //IO 작업 종료 정보
218             cur->PC += 1;
219             출력
220         }
221     }
222
223 }
224

```

일단 job_q를 순회하며 (192~194) job_q속에 있는 프로세스의 양을 jobQnum 에 저장하였고, 이를 이용하여 while문안에서 혹여나 도착하지 않은 프로세스들을 검사하였습니다.

아까 주석에도 나와있었듯이, finishNum은 레디큐에서 들어왔다가 종료한 프로세스 수를 의미하는 변수입니다.

199~208 은 각 프로세스의 arrival_time 변수를 리눅스 리스트의 list_for_each_entry 함수를 이용하여 둘러보며 clock과 동일할 때 ready 큐에 뒤쪽부터 넣어줍니다. 비슷하게, 210라인에서 221 라인은 wait_q속 리스트를 list_for_each_entry 함수를 이용해 돌며 io작업이 종료되었는지 검사합니다. 이 코드에 의하여 clock이 돌 동안 도착하거나, io가 끝난 프로세스는 자동으로 ready_q에 들어가게 됩니다.

```

225 // 레디큐 맨 앞의 프로세스 작업
226 cur = list_entry(ready_q.next, process, ready); // 레디큐 맨 앞꺼.
227
228 if( cur->pid != 100 && clock >= cpuOpEnd ) //ready큐 맨 앞 프로세스가 idle 프로세스 아닌것. 단, 현재 실행중인 명령어가 끝난경우만
229 {
230     if( cur->PC < cur->code_bytes/2) // 프로세스 모든 명령어가 끝났는 지 확인
231     {
232         if(runningPID != cur->pid) // 이전까지 수행하던 프로세스 PID가 지금 PID와 다르면 컨텍스트 스위치
233         {
234             clock += 10;
235             idle_time += 10;
236             fprintf(stdout, "%04d CPU: Switched\tfrom: %03d\tto: %03d\n", clock, runningPID, cur->pid); //CPU 작업 전환 정보 출력
237             runningPID = cur->pid;
238             int i;
239             for(i=1;i<10;i++) // 컨텍스트 스위치 도중에 Arrival하는 프로세스 추가.
240             {
241                 list_for_each_entry(next, &job_q, job)
242                 {
243                     if( next->arrival_time == (clock+10+i)){ // 컨텍스트 스위치 도중 Arrival한 프로세스는 끝나고 도달한걸로 표시됨.
244
245                         list_add_tail(&next->ready,&ready_q); //레디큐에 프로세스 추가
246                         fprintf(stdout, "%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n", clock, next->pid,
247                                     next->arrival_time, next->code_bytes, next->PC); //로드된 프로세스 정보 출력
248                     }
249                 }
250             }
251         }
252     }
253 }
254
255

```

이후 코드는 조금 복잡해집니다. 226라인에서 ready_q의 가장 앞 쪽의 프로세스를 list_entry를 이용하여 cur에 받고, 이후 중첩 if문을 이용해 이 받은 cur가 Idle프로세스인지, 만약 맞다면 각 q들의 상태는 어떤지, 아니라면 어떤 명령어가 있는지 등 여러 분기를 통해 다음에 cur를 이용하여 어떻게 행동을 계산합니다.

간단히 살펴보면, 주석에도 나와 있듯이 228 : cur가 Idle이 아니고 실행 중인 명령어가 끝난 경우.

230 : Idle이 아니면 명령어가 끝나있는지. 232 : 컨텍스트 스위치 분기. 그리고 빨간색으로 표시 한 부분이 context switching 코드입니다. 239 라인부터의 for문은 문맥전환 도중에 해야 할 행동들을 구현해 준 것입니다.

```

256     if(cur->operations[cur->PC].op == 0) // cpu작업인경우
257     {
258         cpuOpEnd = cur->operations[cur->PC].len + clock; //이 명령어가 끝나는 시간 저장
259         cur->PC += 1; //PC레지스터 1더해서 명령어 다음꺼 생각해둠
260         isworkCPU = true;
261     }else if(cur->operations[cur->PC].op == 1) //io 작업인경우
262     {
263         cur->ioOpEnd = cur->operations[cur->PC].len + clock;
264         //ready 큐에서 wait 큐로 옮김
265         list_del_init(&cur->ready);
266         list_add_tail(&cur->wait,&wait_q);
267     }
268
269     }else // 이 프로세스 명령어가 다 끝났음
270     {
271         list_del_init(&cur->ready); // 프로세스 명령어 다 수행 끝났으니 레디큐에서 삭제
272         finishNum++; // 종료 개수 1개증가.
273         continue; // 프로세스의 명령어가 끝났으니 아무행동도 하지않음 = clock이 소모되지 않음.
274     }

```

그 이후로도 분기를 통해 이 cur라는 프로세스가 어떤 행동을 취할지 정해줍니다.

256, 261 : idle프로세스가 아니고, 종료된 프로세스가 아닌 경우 cpu작업인지 io작업인지 파악합니다.

그 이후 256 if문에서는 isworkCPU bool 변수를 true로 만들어 idle_time이 증가하지 않게 합니다. 이 bool변수는 while문 끝에서 false일 경우 idle_time 변수를 증가시킵니다.

261 else if 문에서는 io 작업인 경우 wait_q로 프로세스를 옮기게 되고, 269라인처럼 명령어가 끝나있다면 그에 따른 행동을 취하게 됩니다.

앞 코드까지가 idle이 아닌 프로세스일 경우의 분기이고, 아래의 코드들은 idle일 때 어떻게 작동할 것인지에 관한 코드입니다.

```

276     }else if( cur->pid == 100 ) // idle 프로세스
277     {
278         list_del_init(&cur->ready);
279         if(!list_empty(&ready_q)) // idle프로세스 이외의 다른 프로세스 존재
280         {
281             list_add_tail(&cur->ready,&ready_q); // idle 프로세스 ready큐 마지막으로 보냄.
282             continue; // idle 프로세스는 우선순위가 낮음. 즉 스케줄링으로 리스트 뒤로 보낸것을 코드로써 표현될 뿐 실제로는 아무 행동도 하지않음
283         }
284         else if(list_empty(&ready_q)) //ready큐에 idle프로세스만 있음
285         {
286             if( !list_empty(&wait_q) ) // wait_q에 프로세스 존재
287             {
288                 list_add_tail(&cur->ready,&ready_q);
289
290                 if(runningPID != cur->pid) // 컨텍스트 스위치
291                 {
292                     clock += 10;
293                     idle_time += 10;
294                     fprintf(stdout, "%04d CPU: Switched\tfrom: %03d\tto: %03d\n", clock, runningPID, cur->pid); //CPU 작업 전환 정보 출력
295                     runningPID = cur->pid; //100이다.
296                     int i;
297                     for(i=1;i<10;i++) // 컨텍스트 스위치 도중에 Arrival하는 프로세스 추가.
298                     {
299                         list_for_each_entry(next, &job_q, job)
300                         {
301                             if( next->arrival_time == (clock+10+i) ) // 컨텍스트 스위치 도중 Arrival한 프로세스는 끝나고 도달한것으로 표시됨.
302                             {
303                                 list_add_tail(&next->ready,&ready_q); //레디큐에 프로세스 추가
304                                 fprintf(stdout, "%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n",
305                                     clock, next->pid, next->arrival_time, next->code_bytes, next->PC); //로딩된 프로세스 정보 출력
306                             }
307                         }
308                     }
309                 }
310             }
311         }

```

cur 프로세스가 idle 프로세스일 경우 일차적으로 ready_q를 기준으로 봅니다.

ready_q에 다른 프로세스가 있는가를 일차적으로 나눈 후 ,
그 이후 wait_q를 기준으로 또 프로세스가 있는지 나눕니다.

이를 간략히 나타내면 다음과 같습니다. →



```

312     else if( list_empty(&wait_q) ) // wait_q에 프로세스가 없다면
313     {
314         if(jobQnum-1 == finishNum) // job_q속 idle 제외 프로세스 개수와 끝낸 프로세스 개수 동일
315         {
316             finishNum++;
317             break; //모든 프로세스를 다했음. clock소모 없이 그냥 끝내면됨.
318         }
319         else if( jobQnum-1 != finishNum) // 아직 도착 안한 프로세스 존재
320         {
321             list_add_tail(&cur->ready,&ready_q);
322
323             if(runningPID != cur->pid) // 컨텍스트 스위치
324             {
325                 clock += 10;
326                 idle_time += 10;
327                 fprintf(stdout, "%04d CPU: Switched\from: %03d\tto: %03d\n", clock, runningPID, cur->pid); //CPU 작업 전환 정보 출력
328                 runningPID = cur->pid; //100이다.
329                 int i;
330                 for(i=1;i<10;i++) // 컨텍스트 스위치 도중에 Arrival하는 프로세스 추가.
331                 {
332                     list_for_each_entry(next, &job_q, job)
333                     {
334                         if( next->arrival_time == (clock-10+i)) // 컨텍스트 스위치 도중 Arrival한 프로세스는 끝나고 도달한걸로 표시됨.
335                         {
336                             list_add_tail(&next->ready,&ready_q); //레디큐에 프로세스 추가
337                             fprintf(stdout, "%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n",
338                                 clock, next->pid, next->arrival_time, next->code_bytes, next->PC); //로드된 프로세스 정보 출력
339                         }
340                     }
341                 }
342             }
343         }
344     }
345 }

```

이제 마지막으로 wait_q에 프로세스가 있는지 구분합니다.

wait_q에 프로세스가 없다 (312) > 1. job_q 속 프로세스개수 만큼 ready_q에서 프로세스 종료 >> 프로그램 종료
 2. 아직 도착 못한 프로세스가 있을시 >>>context switching

이후 이 여러 분기를 통해 cur 프로세스는 매번 각 clock때마다 자신이 속하는 부분의 코드에서 특정 행동을 취하게 됩니다.

```

348
349     if(runningPID == 100)
350     {
351         isworkCPU = false;
352     }
353
354     if(!isworkCPU) // cpu 일 안했으면 idle시간 증가
355     {
356         idle_time++;
357     }
358     clock++;
359 }
360 } // While 문 경계
361
362 //최종 리포트
363 fprintf(stdout, "**** TOTAL CLOCKS: %04d IDLE: %04d UTIL: %.2f%%\n", clock, idle_time, (double)(clock-idle_time)*100/clock);
364
365 //fprintf(stdout, "finish = %d \n", finishNum);
366
367
368 // 리스트 제거, 할당 해제 : 이미 선언한 cur과 next 사용
369 list_for_each_entry_safe(cur, next, &job_q, job){
370     list_del(&cur->job);
371     free(cur->operations);
372     free(cur);
373 }
374
375 return 0;
376 }

```

이후 코드에선, clock을 증가시킨후 bool 변수에 따라 idle_time을 기록하여 최종 레포트를 출력하게됩니다.

그리고 (368) 마지막으로 job속 리스트를 모두 동적할당시키며 프로그램은 종료하게됩니다.

위 코드를 간단히 표현하면 다음과 같습니다.

- Idle 아닌 프로세스 → 명령어가 끝이남. → 이전 프로세스의 PID 다르면 Context Switching,
ready_q에서 삭제
↓
명령어 남아 있음
↓
CPU명령어 / IO명령어 → wait_q 올라감
- Idle 프로세스
↓
ready_q에 → 다른 프로세스 있음 → 뒤로 보냄(우선순위)
→ 혼자임 → wait_q 속 프로세스 있음 → Context Switching
→ wait_q 속 프로세스 없음 → job개수만큼 했는가? → 종료
↓ 도착 안 한게 있다. → Context S

실행 결과 :

과제 1-1 결과

```
ubuntu@201619460:~/hw1$ gcc -o os1 os1.c
ubuntu@201619460:~/hw1$ cat test1.bin | ./os1
PID: 002      ARRIVAL: 016      CODESIZE: 002
0 167
PID: 001      ARRIVAL: 003      CODESIZE: 006
0 2
1 72
0 5
PID: 000      ARRIVAL: 000      CODESIZE: 004
0 4
1 255
```

과제 1-2 결과

```
ubuntu@201619460:~/hw1$ gcc -o os1 os1-2.c
ubuntu@201619460:~/hw1$ cat test1.bin | ./os1
0000 CPU: Loaded PID: 000      Arrival: 000
0000 CPU: Loaded PID: 100      Arrival: 000
0003 CPU: Loaded PID: 001      Arrival: 003
0004 CPU: OP_IO START len: 255 ends at: 0259
0016 CPU: Loaded PID: 002      Arrival: 016
0271 CPU: OP_IO START len: 072 ends at: 0343
*** TOTAL CLOCKS: 0525 IDLE: 0347 UTIL: 33.90%
```

Submission of 2021운영체제 과제 1-1

[View source](#)
[Resubmit](#)

Compilation Warnings

```
oshw11c.c: In function 'main':
oshw11c.c:152:3: warning: ignoring return value of 'fread',
fread(next->operations,next->code_bytes,1,stdin); //코드
```

Execution Results

✓✓✓✓✓

> Test case #1: AC [0.003s,772.00 KB] (2/2)
 > Test case #2: AC [0.005s,772.00 KB] (2/2)
 > Test case #3: AC [0.002s,772.00 KB] (2/2)
 > Test case #4: AC [0.003s,772.00 KB] (2/2)
 > Test case #5: AC [0.004s,772.00 KB] (2/2)

Resources: 0.016s, 772.00 KB
Final score: 10/10 (10.0/10 points)

Submission of 2021운영체제 과제 1-2

[View source](#)
[Resubmit](#)

Compilation Warnings

```
oshw12c.c: In function 'main':
oshw12c.c:156:3: warning: ignoring return value of 'fread',
fread(next->operations,next->code_bytes,1,stdin); //코드
```

Execution Results

✓✓✓✓✓

> Test case #1: AC [0.007s,772.00 KB] (2/2)
 > Test case #2: AC [0.004s,772.00 KB] (2/2)
 > Test case #3: AC [0.004s,772.00 KB] (2/2)
 > Test case #4: AC [0.005s,772.00 KB] (2/2)
 > Test case #5: AC [0.007s,772.00 KB] (2/2)

과제 1-3 결과

Submission of 2021운영체제 과제 1-3

[View source](#)
[Resubmit](#)

Compilation Warnings

```
oshw13c.c: In function 'main':  
oshw13c.c:158:3: warning: ignoring return value of 'fread',  
fread(next->operations,next->code_bytes,1,stdin); //코드
```

Execution Results

✓ ✗ ✗ ✗ ✗

➤ Test case #1: AC [0.007s, 776.00 KB] (2/2)
➤ Test case #2: WA [0.006s, 776.00 KB] (0/2)
➤ Test case #3: WA [0.008s, 776.00 KB] (0/2)
➤ Test case #4: WA [0.009s, 776.00 KB] (0/2)
➤ Test case #5: WA [0.008s, 776.00 KB] (0/2)

Resources: 0.038s, 776.00 KB
Final score: 2/10 (2.0/10 points)

```
ubuntu@201619460:~/hw1$ cat test1.bin | ./os1  
0000 CPU: Loaded PID: 000 Arrival: 000 Codesize: 004 PC: 000  
0000 CPU: Loaded PID: 100 Arrival: 000 Codesize: 002 PC: 000  
0003 CPU: Loaded PID: 001 Arrival: 003 Codesize: 006 PC: 000  
0015 CPU: Switched from: 000 to: 001  
0016 CPU: Loaded PID: 002 Arrival: 016 Codesize: 002 PC: 000  
0028 CPU: Switched from: 001 to: 002  
0089 IO : COMPLETED! PID: 001 IOTIME: 089 PC: 001  
0206 CPU: Switched from: 002 to: 001  
0222 CPU: Switched from: 001 to: 100  
0259 IO : COMPLETED! PID: 000 IOTIME: 259 PC: 001  
*** TOTAL CLOCKS: 0260 IDLE: 0080 UTIL: 69.23%
```

네. 그렇습니다. 위의 코드가 많이 읽기 어렵고 하드 코딩되어 있는 이유도, 제가 많이 미숙해서 그렇습니다.

저는 오답인 이유를 제 코드에서 크게 3가지로 보았습니다.

0000 CPU: Loaded PID: 000 Arrival: 000 Codesize: 004 PC: 000	ubuntu@201619460:~/hw1\$ cat test1.bin ./os1
0000 CPU: Loaded PID: 100 Arrival: 000 Codesize: 002 PC: 000	0000 CPU: Loaded PID: 000 Arrival: 000 Codesize: 004 PC: 000
0003 CPU: Loaded PID: 001 Arrival: 003 Codesize: 006 PC: 000	0000 CPU: Loaded PID: 100 Arrival: 000 Codesize: 002 PC: 000
0015 CPU: Switched from: 000 to: 001	0003 CPU: Loaded PID: 001 Arrival: 003 Codesize: 006 PC: 000
0016 CPU: Loaded PID: 002 Arrival: 016 Codesize: 002 PC: 000	0015 CPU: Switched from: 000 to: 001
0028 CPU: Switched from: 001 to: 002	0016 CPU: Loaded PID: 002 Arrival: 016 Codesize: 002 PC: 000
0089 IO : COMPLETED! PID: 001 IOTIME: 089 PC: 001	0028 CPU: Switched from: 001 to: 002
0205 CPU: Switched from: 002 to: 001	0089 IO : COMPLETED! PID: 001 IOTIME: 089 PC: 001
0220 CPU: Switched from: 001 to: 100	0206 CPU: Switched from: 002 to: 001
0259 IO : COMPLETED! PID: 000 IOTIME: 259 PC: 001	0222 CPU: Switched from: 001 to: 100
0270 CPU: Switched from: 100 to: 000	0259 IO : COMPLETED! PID: 000 IOTIME: 259 PC: 001
*** TOTAL CLOCKS: 0270 IDLE: 0090 UTIL: 66.67%	*** TOTAL CLOCKS: 0260 IDLE: 0080 UTIL: 69.23%

1. 제 코드는 마지막에 모든 프로세스가 마쳐지는 순간 ready_q에서 제거되어 딱히 문맥 전환 없이 종료되어 모범 답안에 비해 딱 10 clock 과 10 idle_time이 부족하였습니다. 그 외 JOTA 정답코드에서도 10 차이로 오답이 난 것이 일부 보였습니다. 반대로, 마지막에 문맥전환이 없으면 정답으로 처리되는 케이스도 존재했습니다.

2. IO 명령어 complet 할때 출력되는 PC값이 모범 답안에 비해 1씩 더 높았습니다. 그러나 이는 io 작업의 경우 wait_q에 넣기 전에 PC값을 미리 올려주는데, wait_q에서 ready_q로 꺼낼 때 PC값을 올려줌으로 알고리즘을 수정하니 수정이 되었습니다.

```
ubuntu@201619460:~/hw1$ cat test1.bin | ./os1
0000 CPU: Loaded PID: 000 Arrival: 000 Codesize: 004 PC: 000
0000 CPU: Loaded PID: 100 Arrival: 000 Codesize: 002 PC: 000
0003 CPU: Loaded PID: 001 Arrival: 003 Codesize: 006 PC: 000
0015 CPU: Switched from: 000 to: 001
0016 CPU: Loaded PID: 002 Arrival: 016 Codesize: 002 PC: 000
0028 CPU: Switched from: 001 to: 002
0089 IO : COMPLETED! PID: 001 IOTIME: 089 PC: 002
0205 CPU: Switched from: 002 to: 001
0220 CPU: Switched from: 001 to: 100
0259 IO : COMPLETED! PID: 000 IOTIME: 259 PC: 002
*** TOTAL CLOCKS: 0259 IDLE: 0079 UTIL: 69.50%
```

```
}else if(cur->operations[cur->PC].op == 1) //io 작업인경우
{
    cur->ioOpEnd = cur->operations[cur->PC].len + clock;
    cur->PC += 1;
    //ready 큐에서 wait 큐로 옮김
    list_del_init(&cur->ready);
    list_add_tail(&cur->wait,&wait_q);
}

// wait_q에서 io명령어 다하면 ready_q로 이동
list_for_each_entry_safe(cur, next, &wait_q, wait){
    if( cur->ioOpEnd <= clock){ //io 명령어 수행 완료하면
        //wait_q에서 ready_q로 옮김
        list_del_init(&cur->wait);
        list_add_tail(&cur->ready,&ready_q);
        fprintf(stdout, "%04d IO : COMPLETED! PID: %03d\t", cur->PC, cur->PC);
        cur->PC += 1;
    }
}
```

옮김.
(위 코드 참고)

3. 제 불찰로, 코드 제작 할 때 io변환작업에 1 clock이 소모됨 크게 신경 쓰지 못해서 오차가 있을 거라 예상합니다.

처음에는 제가 만든 코드를 기반으로 답안 양식에 맞추어 코드를 수정해보고자 노력하였습니다.

위의 1번 3번 문제를 고쳐 보고자 노력하였으나, 너무나 코드가 가독성이 떨어져 오류 부분을 파악하기가 너무 어려웠습니다. 교수님께서 숙제로 내주신 자율 진도표를 작성하고 한 코드임에도 불구하고 제 기량이 너무 부족해서 너무나도 해제하기 어려운 모습의 알고리즘 이였습니다.

그렇기에 저는 ready_q의 첫 리스트가 idle이나 같은 조건적 판단보다 객체적으로 흐름을 조절하는 방법이 없을 까, 보다 더 간략하고 깔끔한 구조는 없을까 생각을 해보았습니다.

저는 크게 4가지를 기준으로 하여 다시 코드를 짜보기 시작하였습니다.

1. 교수님처럼 모든 행동을 출력하도록 코딩. -> 수정 보완 및 디버깅 용이
2. 코드 고칠 부분 찾기 : 버릴 부분과 가져갈 부분 구분 -> 코드의 재활용성 및 익숙함에 따른 효율적인 코드 작성가능.
3. PID 비교로 context switching 파악 -> 위 코드처럼 모든 조건에 Context switching 이 아닌 공통 이벤트 트리거 파악
4. 프로세스 실행을 그 프로세스의 pid같은 특성이 아닌 Q로만 (q가 비었는지 등에 따라) 구분.

안타깝게도 결과는 그리 좋지 못했습니다. 그래도 나름 교수님의 조언을 따라 기준을 잡고 점진적 프로그래밍을 시작했다고 생각했는데, 뜻처럼 일이 쉽게 풀리지는 않았습니다. 코드를 제작하면서도 어느 부분이 잘못 된지 파악하기 어려웠고, 아예 파악하지 못해 코드 자체를 처음부터 쓴 적도 3번 정도 있었던 것 같습니다. 코드를 다시 새로 짜더라도 결국엔 비슷한 알고리즘으로 작동하게 저의 좁은 시야에서 벗어나지 못했던 것 같습니다. 물론 변명임을 알고 있습니다. 제가 부족하여 결국 제대로 된 결과 케이스를 제출하지 못하였습니다. 제가 좀 더 일찍 시작하고, 좀 더 열심히 했어야함을 알고 있기에 더욱 아쉽습니다. 저 스스로에게 갈 길이 멀다고 새삼 느낀 일이었으며, 교수님께는 과제를 못한 점.. 진심으로 죄송하다고 생각합니다.

자율적 진도표

1. 과제0을 교수님이 나누어주신 정답수도코드를 보고 수정	자료형 이름 통일, 동적할당	O
2. List Library 파악 : 강의 속 링크 정독, 예제 수행	kmalloc 공부 : 1-1 과제 PPT 마지막 부분의 코드에서 free인 것을 보아 커널코드 기반으로 쓰지 않는다는 것을 깨달음	O
3.수정된 과제0을 보완하여 출력이 아닌 job_q list 속에 넣기로 수정	fread관련 sizeof 관계 파악	O
4. 순회하며 정보출력 : list_for_each_entry_reverse 사용하여 역으로 출력	출력할때 code tuples	O
5. 리스트 제거 및 할당 해제 : list_for_each_entry_safe 사용		O
6. 과제 1-1		O
7. Idle process 구현	조타를 참고하여 pid 100, arrival 000 codesize 2 로 설정. 0xFF로 설정에 대해 살짝 고민	O
8. 프로세스에 PC레지스터 추가		O
9. 대략적인 수행 알고리즘 생각	clock을 어떻게 돌릴건 지 고민, 변수 무엇이 필요한지 파악	O
10. 레디큐에 도착할때 넣기		O
11. 레디큐에 들어진 프로세스 실행 조건 생각		O
12. 레디큐 맨앞에 있는 프로세스 명령어 실행		O
13. CPU가 일을 안했다면 idle 시간 더하기	bool변수 사용 고민	O
14. context switching구현	bool 변수를 쓸까 하였으나, 10초간 모든 기능이 정지하므로 context switching동안 clock을 기능을 할 필요가 없다고 판단.	O
15. 계산방식 검토 및 결과값 비교		O
16. 과제 1-2		O
17. 과제1-2에서 변경할 것 고려	PID가 다른지 확인, ready큐 idle밖에 없어서 idle 오퍼레이션 구현	O
18. 고려한 변수 추가	프로세스구조체에 int ioOpEnd 추가 wait_q에서 io명령어 종료시간 저장	O
19. 멀티프로그래밍기법에 맞게 io실행시 리스트 변경	프로세스의 실행할 명령어가 io일때 waitq로 옮기기, idle bool 변수 삭제 고려.	O
20.context switching 파악하기 위한 로직 작성	방금까지 했던 PID를 runningPID에, 지금 ready_q의 프로세스 PID를 비교 기법 고려	O
22. wait_q에서 io다되면 ready_q로 옮기기		O
23. 계산방식 검토 및 결과값 비교		O
24. 변경		O
24. 코드 고칠부분 찾기 : 버릴부분과 가져갈 부분 구분		O
25. PID 비교로 context switching 파악		O
26. 과제 1-3	(죄송합니다.. πππ)	

진행 하며 : 자율적 진도표를 작성하고 과제를 해나가며, 간단히 고민했거나 생각했던 것을 정리하였습니다. 너무 간단한 것은 제외하고 진행하며 느꼈던 어려움이나, 생각을 옮겨 놓았습니다.

3. 수정된 과제0을 보완하여 출력이 아닌 job_q list 속에 넣기로 수정.

과제를 할 때 정말 바보 같은 일이 하나 있었습니다. 과제 1-1 fread함수를 사용할 때 while(fread(cur, sizeof(int)*3, 1, stdin) == 1)을 while(fread(cur, sizeof(int), 3, stdin) == 1) 으로 바꾸었더니 작동이 되지 않는 것 이였습니다. 저는 리턴값이 당연히 성공하면 1, 실패하면 0이라 생각하고 괜히 애꿎은 파일 지시자 쪽만 엉청 찾아보았었습니다. 그러나 fread는 읽기 성공한 항목수를 리턴하였기에 while(fread(cur, sizeof(int), 3, stdin) == 3) 으로 하니 잘 작동하였습니다. 덕분에, fread 및 파일함수의 파일 관리자는 확실하게 알게 된 것 같습니다. <http://www.cplusplus.com/reference/cstdio/fread/> 의 "The position indicator of the stream is advanced by the total amount of bytes read." "파일 지시자는, read를 한 byte 총 수만큼 이동한다."를 참고하였습니다.

s1.c



```
while(fread(cur, sizeof(int), 3, stdin) == 1) {  
    INIT_LIST_HEAD(&cur->job);
```

s1.c



```
while(fread(cur, sizeof(int), 3, stdin))
```

4. 순회하며 정보출력 : list_for_each_entry_reverse 사용하여 역으로 출력

과제에서 출력할 때 code tuples에 대해 어떻게 출력해야하나 생각하였는데, 교수님께서 말씀하신 "연속된 메모리 공간을 할당해 저장"에서 배열이 생각났습니다. C언어에서는 배열을 연속된 메모리 공간을 이용하고, 사실상 배열명은 포인터인 것을 기억하여 배열을 이용하였습니다.

9. 과제 1-2를 할 때 대략적인 수행 알고리즘 생각

clock을 어떻게 돌릴 건지 고민되었습니다. 변수는 일단 clock과 idle time을 선언해서 while을 돌며 매 루프마다 clock을 증가시키고 idle프로세스가 돌아가고 있을 땐 clock과 idle 둘 다 증가시키고자 생각했습니다.

과제 1-2에서 idle프로세스 동작 파악을 위해 job_q 의 프로세스 개수와 ready_q에 여태 들어온 프로세스 개수를 비교하며 종료할지 판단.

실제로 모두 구현하였습니다.

14. context switching구현

문맥 전환 알고리즘을 구현할 때 bool 변수를 사용할까 하였으나, 10초간 모든 기능이 정지하므로 context switching동안 clock의 기능을 일일이 다 구현할 필요가 없다고 판단하였습니다. 그렇기에 문맥 전환 때 clock을 하나하나 올리는 게 아닌 그냥 clock += 10을 함으로써 구현하였습니다.

단, job_q에서 ready_q에 프로세스가 들어갈 때, context switching 도중에 도달하는 프로세스가 있을 수 있다고 생각하여, ready_q에 프로세스를 넣는

if문 조건식을 (오른쪽 코드)

cur->arrival_time == clock에서
cur->arrival_time <= clock 으로
변경해보았으나,

```
// 도착하면 레디큐에 넣음.
list_for_each_entry(cur, &job_q, job){

    if( cur->arrival_time == clock){ //도착시간이 clock과 같다면

        list_add_tail(&cur->ready,&ready_q); //레디큐에 프로세스 추가
        fprintf(stdout, "%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\n", clock, cur->pid, cur->arrival_time, cur->codesize);
    }
}
```

의도치 않게 프로세스를 무한히 ready_q에 넣어서 context switching에 따로 구현을 해놓았습니다.

```
if(cur->PC == 0 && readyQnum !=0) // 컨텍스트 스위치 ( 단일 프로그래밍이므로 0번 명령어 실행시 발생 , 처음 프로세스 접근시 소모 안됨)
{
    clock += 10;
    idle_time += 10;
    int i;
    for(i=1;i<10;i++) // 컨텍스트 스위치 도중에 Arrival하는 프로세스 추가.
    {
        list_for_each_entry(next, &job_q, job)
        {
            if( next->arrival_time == (clock-10+i)){ // 컨텍스트 스위치 도중 Arrival한 프로세스는 끝나고 도달한걸로 표시됨.

                list_add_tail(&next->ready,&ready_q); //레디큐에 프로세스 추가
                fprintf(stdout, "%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n", clock, next->pid, next->arrival_time, next->codesize, next->PC);
            }
        }
    }
}
```

17. 과제1-2에서 과제 1-3로 변경할것 고려

PID가 다른지 확인 : CPU가 실행하고 있는 지금 프로세스의 PID 저장변수 선언 해야겠다고 생각.

>> context 스위칭과 연결되었습니다.

처음엔 io명령어로 인해 ready큐에서 wait큐로 옮겨간 프로세스가 언제 다시 ready큐로 돌아올지 저장할 변수로 프로세스 속 Arrival 변수를 생각했습니다. 이 arrival 변수는 처음 레디 큐 들어올 때를 제외하고는 출력값에 쓰이지 않으니 덮어씌워서 사용해 버릴까 생각하였으나, 교수님께서 강의 마지막에 정보 출력하는걸 보고 사용하지 않고 새로 개별적으로 변수를 추가하기로 생각했습니다.