운영체제 - 과제 1

(1-1) Linux List Library 기반 연결 리스트 구현

(1-2, 1-3) 멀티프로그래밍 기반 배치 시스템 시뮬레이터

컴퓨터공학부
202246109
김기현

자율 진도표

단계	완료 여부○▲
List 자료구조 파악 : 코드 예제 분석	○
test1.bin의 데이터로 job_q 생성	○
list_for_each_entry()를 이용한 순회 후 출력	○
list_for_each_entry()_reverse를 이용한 순회 후 출력	○
list_for_each_entry_safe()를 이용한 제거 및 할당 해제	○
과제 1-1 완료 (JOTA 제출)	○
idle 프로세스 구현	○
읽어온 데이터 ready_q에 탑재 (Load) 구현	○
레디큐의 프로세스 명령처리 구현 (cpu, IO 작업)	○
Context Switching 구현	○
동작 구조 분석 및 파악 : result1.txt, result6.txt 집중분석	○
case 1 : 000 -> ~~100~~ -> 001 idle 건너 뛰는 케이스 구현	○
case 2 : 000 -> 100 idle로 전환되는 케이스 구현	○
idle 프로세스 처리 구현	○
results.txt 파일 결과값 비교 및 분석	○
과제 1-2 완료 (JOTA 제출)	○
1-2 구조 재구성 : 주요 기능 함수화	○
~~isIdle() 구현~~	○
switchProcess() 구현	○
executeProcess() 구현	○
1-3 구조 파악 : results.txt의 1-3 실행 결과 집중분석	○
프로세스 대기 상태 전환(ready_q -> wait_q) 구현	○
IO 일 때 ready_q -> wait_q 구현	○
checkIoEnd() (wait_q -> ready_q) 구현	○
구조 분석 : 실행 결과와 Example(full result for 1-3) 집중분석	○
구조 재구성 : Example 에 기반한 구조로 재구성	○
JOTA 제출 결과 비교 및 분석	○
구조 재구성 : checkIoEnd() 시간 오차 수정	○
과제 1-3 완료 (JOTA 제출)	○

코드설명

```c
typedef struct{
    unsigned char op; //cpu인지 io인지
    unsigned char len; //수행시간
} code_tuple;

typedef struct{
    int pid;
    int arrival_time;
    int code_bytes;
    int PC;                 //Program Counter   operations[PC]
    int ioEnd;              //IO 끝나는 시간
    code_tuple *operations; //code_tuple이 저장된 위치를 가리킴
    struct list_head job, ready, wait;
} process;
```

code_tuple 과 process 는 과제 0에 기반한 구조체 입니다.
추가된 변수는 PC, ioEnd, operations가 있습니다.
PC는 Program Counter로 수행 중인 코드 위치를 관리합니다.
Operations 는 code_tuple이 저장된 위치를 가리키는 포인터 입니다.
ioEnd는 io작업의 종료 시점을 저장합니다.

```c
int jobQnum = 0;        //job_q에 있는 프로세스 수
int clock = 0;
int opEndTime = 0;     //operation이 끝나는 시간
int terminateNum = 0;   //종료된 프로세스 수
int idleTime = 0;       //idle시 카운트
int prevPid = 0;        //이전 프로세스 pid
bool doIdle = false;    //idleTime++ 용 플래그
bool reschedule = false;    //리스케줄 플래그
```

전역변수들 입니다.
주요 기능 함수화 중 함수의 매개변수 전달이 비효율적이어서
main()의 까다로운 지역변수들을 전역변수화했습니다.

```c
int main(int argc, char* argv[]){
    LIST_HEAD(job_q);       //시작점 생성(head)
    LIST_HEAD(ready_q);     //시작점 생성(head)
    LIST_HEAD(wait_q);     //시작점 생성(head)
    process *cur, *next;

    while((cur = (process*)malloc(sizeof(process))) && fread(cur, sizeof(int) * 3, 1, stdin)){
        cur->PC = 0;
        cur->ioEnd = 0;
        cur->operations = (code_tuple*)malloc(cur->code_bytes);
        fread(cur->operations, cur->code_bytes, 1, stdin);
        INIT_LIST_HEAD(&cur->job);  //초기화
        INIT_LIST_HEAD(&cur->ready);  //초기화
        INIT_LIST_HEAD(&cur->wait);  //초기화
        list_add_tail(&cur->job, &job_q); //끝에 넣고
        jobQnum++;
    }

    {   //idle프로세스 추가
        cur = (process*)malloc(sizeof(process));
        cur->pid = 100;
        cur->arrival_time = 0;
        cur->code_bytes = 2;
        cur->operations = (code_tuple*)malloc(cur->code_bytes);
        cur->operations[0].op = 0xff;
        cur->operations[0].len = 0;
        cur->PC = 0;
        cur->ioEnd = 0;
        INIT_LIST_HEAD(&cur->job);  //초기화
        INIT_LIST_HEAD(&cur->ready);  //초기화
        INIT_LIST_HEAD(&cur->wait);  //초기화
        list_add_tail(&cur->job, &job_q); //끝에 넣고
    }
```

메인함수 입니다.
LIST_HEAD를 사용해 head를 생성합니다
연결리스트에 관여하는 변수들은 local 선언하였습니다.

과제 0을 기반으로 .bin의 데이터를 읽어옵니다.
읽을 데이터는 int형으로 구성된 pid, arrival_time, code_bytes 셋이므로
fread()의 size 인자를 sizeof(int) * 3 으로 설정하였습니다.

동시에, job_q에 프로세스를 추가하는 과정에서,
job_q에 탑재된 프로세스의 수를 알리는 jobQnum을 카운트 합니다.

Idle 프로세스를 생성 및 추가합니다.
(이 때 idle은 jobQnum에 미포함)

코드 설명

```
bool isIdle(process *cur, struct list_head *ready_q, int *idleTime, int *clock){…
}

void switchProcess(int *idleTime, int *clock, int *prevpid, process *cur, process *next, struct list_head *job_q , struct list_head *ready_q){…
}

bool excuteProcess(int *clock, int *processEnd, process *cur, int *idleTime, int *prevpid, int *terminateNum){…
}

while(jobQnum != terminateNum){ //jobQ에 들어있는거(idle때문에 -1) 랑 종료된 프로세스 수가 같아질 때 까지

    //ready_q에 넣는 코드
    list_for_each_entry(cur, &job_q, job){  //순회 해서
        if(clock == cur->arrival_time){     //도착시간이랑 시간이 같으면
            list_add_tail(&cur->ready, &ready_q);   //ready_q에 추가
            printf("%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n", clock, cur->pid, cur->arrival_time, cur->code_bytes, cur->PC);
        }
    }

    cur = list_entry(ready_q.next, process, ready); //cur의 위치 조정 ready_q의 맨 앞으로

    if(isIdle(cur, &ready_q, &idleTime, &clock)){    //현재 프로세스가 idle인지 확인하고 처리
        continue;
    }else{
        if(cur->pid != prevpid && cur->pid != 0){
            switchProcess(&idleTime, &clock, &prevpid, cur, next, &job_q, &ready_q);
        }
        if(excuteProcess(&clock, &processEnd, cur, &idleTime, &prevpid, &terminateNum)) continue;
    }
    clock++;
}
```

1-2 의 함수와 주요 동작 구조입니다. 1-3에서 1-2의 내용이 중복되기에 상세한 내용은 1-3에서 설명하도록 하겠습니다.

isIdle 함수는 현재 프로세스가 idle일 때 발생하는 상황들을 처리하는 함수입니다.
idle의 우선순위를 따지거나 idleTime을 증가시킵니다.

switchProcess 함수는 context switching을 수행합니다.

executeProcess는 cpu, io 명령을 수행합니다.

1-3 에서 사용하는 함수입니다. 1-2의 함수를 분리하고 checkIoEnd 함수를 추가했습니다.

```
bool opCpu(process *cur){…
}
```

CPU 명령을 처리하는 함수입니다.

```
bool opIO(process *cur, struct list_head *wait_q){…
}
```

IO 명령을 처리하는 함수입니다.

```
bool terminateProcess(process *cur){…
}
```

프로세스의 종료를 처리하는 함수입니다.

```
bool opIdle(process *cur, struct list_head *ready_q){…
}
```

IDLE 명령을 처리하는 함수입니다.

```
bool executeProcess(process *cur, struct list_head *ready_q, struct list_head *wait_q){…
}
```

CPU, IO, IDLE 프로세스를 처리하는 함수입니다.

```
void checkIoEnd(process *cur, process *next,  struct list_head *wait_q, struct list_head *ready_q){…
}
```

IO 처리의 종료를 확인하는 함수입니다.

```
void switchProcess(process *cur, process *next, struct list_head *job_q, struct list_head *ready_q, struct list_head *wait_q){…
}
```

Context Switching 함수입니다.

```
while(jobQnum != terminateNum){
    //Load
    if(!reschedule){
        list_for_each_entry(cur, &job_q, job){          //job_q 순회 해서
            if(clock == cur->arrival_time && reschedule == false){          //도착시간이랑 시간이 같으면 && 리스케줄 요청시 스위칭으로 인한 로딩 중복 방지
                list_add_tail(&cur->ready, &ready_q);   //ready_q에 추가
                printf("%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n", clock, cur->pid, cur->arrival_time, cur->code_bytes, cur->PC);
            }
        }
    ① checkIoEnd(cur, next, &wait_q, &ready_q);  //Io작업이 끝났는지 확인하는 함수;
    }
```

본격적인 로직이 시작되는 부분입니다.
jonQnum 에는 job_q의 프로세스수 terminateNum은 모든 작업 후 종료된 프로세스의 수를 저장하고
while문의 조건에서 두 수가 같아질 때 까지 반복합니다.

job_q를 순회해서 도착시간(arrival_time)이 현재시간(clock)과 같을 때
프로세스를 ready_q에 추가합니다.

checkIoEnd()는 IO작업이 종료된지 확인하는 함수 입니다.

리스케줄 요청으로 인한 continue문이 실행되어
로딩과 종료확인이 중복, Context switching시 체크가 되기 때문에
reschedule == false 일 때 되도록 해봤습니다.

①

```
void checkIoEnd(process *cur, process *next,  struct list_head *wait_q, struct list_head *ready_q){
    //Io작업이 끝났는지 확인하는 함수;
    list_for_each_entry_safe(cur, next, wait_q, wait){  //wait_q를 순회해서
        if(clock >= cur->ioEnd){                  //IO 명령이 끝나면
            list_del_init(&cur->wait);              //wait_q -> ready_q
            list_add_tail(&cur->ready, ready_q);
            printf("%04d IO : COMPLETED! PID: %03d\tIOTIME: %03d\tPC: %03d\n", clock, cur->pid, cur->ioEnd, cur->PC);
            cur->PC++;                              //IO 작업이 끝났으니 PC
        }
    }
}
```

checkIoEnd의 코드입니다.
연결리스트에 관여하는 변수들을 매개변수로 받습니다.

IO명령의 종료시점을 저장하는 ioEnd와 현재시간이 같을 때 프로세스 상태를 전환합니다.
(wait -> ready)
IO명령이 종료되면 wait_q에서 **제거** 후 ready_q에 삽입해야 하니 **list_for_entry_safe**로 순회 합니다.

IO명령이 종료되었으니 PC를 증가시켜 수행코드의 위치를 관리합니다.

```
① checkIoEnd(cur, next, &wait_q, &ready_q);  //Io작업이 끝났는지 확인하는 함수;

  cur = list_entry(ready_q.next, process, ready);     //cur의 위치 조정 ready_q의 맨 앞으로

② if(reschedule){                                     //reschedule을 요청할 때(reschedule == true) 리스케쥴링
      reschedule = false;                             //flag 초기화;

   ① if(cur->pid == 100){                             //pid 000 -> 100 -> 001 으로 가는 케이스 방지
                                                      //    000 --------> 001 으로 가야 함
        list_del_init(&cur->ready);                   //지웠을 때
     ⑴ if(!list_empty(&ready_q)){                    //ready_q가 안비어있으면 idle 외에 다른 작업이 남아있음
            list_add_tail(&cur->ready, &ready_q);     //idle을 ready_q 맨 뒤로 보냄 idle은 가장 낮은 우선순위
            reschedule = true;
            continue;
     ⑵ }else{                                         //ready_q 비었을 때 (idle밖에 없을 때)
            list_add_tail(&cur->ready, &ready_q);
            doIdle = true;
        }
    }

    //printf("%04d Reschedule\tPID:%03d\n", clock, cur->pid);
   ② switchProcess(cur, next, &job_q, &ready_q);   //Context Switching

    checkIoEnd(cur, next, &wait_q, &ready_q);    //clock + 10에 대한 체킹
}

③ if(executeProcess(cur, &ready_q, &wait_q)) continue;
  // cpu, io, idle 처리하는 함수
  // bool 형으로 return으로 true가 넘어올 때(ex. terminated, idle)
  // clock 소모 방지로 continue;


  if(doIdle){          //doIdle 이 true일 때 idleTime 증가
      idleTime++;
  }

  clock++;
}
```

코드의 구조는 크게 네 가지 부분으로 나뉘어 있습니다.

⓪. 프로세스 로딩: 프로세스를 ready_q에 삽입.

①. IO 종료 처리: checkIoEnd() 함수를 호출하여 IO 작업이 완료되었는지 확인.

②. 리스케줄링: reschedule 플래그가 true일 때 실행되며, Context Switching 수행.

③. 프로세스 명령 처리: executeProcess() 함수를 호출하여 프로세스의 명령 처리.

앞서 0번과 1번에 대한 내용을 다루었습니다. 이제 2번에 대해 자세히 살펴보겠습니다.

2번은 reschedule 플래그가 true일 때 리스케줄링을 수행합니다.
ready_q에 변화가 있을 때 플래그를 true로 변경해 스케줄링을 요청합니다.

따라서 코드에서 reschedule 플래그가 true인 경우에는 리스케줄링을 수행하고,
그 후에는 해당 플래그를 다시 false로 설정하여 다음 스케줄링을 대기하게 됩니다.

2번은 다시 두 부분으로 나뉩니다.

2-1. IDLE 상태의 처리

선택 할 프로세스가 IDLE인 경우, 다음과 같은 두 가지 경우를 고려해야 합니다.

(1) ready_q에 IDLE 외의 다른 프로세스가 존재하는 경우

이 경우 IDLE은 우선순위가 낮으므로 IDLE이 아닌 프로세스를 선택해야 합니다.

이 때 IDLE을 ready_q의 뒤로 보낸 후 clock을 소요하면 안되므로 continue문을 사용해 방지합니다.

(2) ready_q에 IDLE 외의 다른 프로세스가 존재하지 않는 경우

이 경우 CPU는 IDLE상태가 되며 idleTime을 카운트 합니다.

2-2. switchProcess() : Context Switching

앞서 IDLE 케이스를 고려하였으므로 switchProcess()에선 온전히 Context Switching만 하면 됩니다.

②
```
void switchProcess(process *cur, process *next, struct list_head *job_q, struct list_head *ready_q){
    // Context Switching
    idleTime += 10;
    clock += 10;
    printf("%04d CPU: Switched\tfrom: %03d\tto: %03d\n", clock, prevPid, cur->pid);
    prevPid = cur->pid;

    //스위칭 중에는 다른 일을 하지 못함
    //스위치 도중 프로세스가 도착 했을 때
    for(int i = 0; i < 10; i++){
        list_for_each_entry(next, job_q, job){
            if(next->arrival_time == clock - 10 + i){    //clock 10 증가시켜놔서
                list_add_tail(&next->ready, ready_q);
                printf("%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n", (clock), next->pid, next->arrival_time, next->code_bytes, next->PC);
            }
        }
    }
}
```

switchProcess()의 코드 입니다. 이 함수는 Context Switching을 수행합니다.

Context Switching은 10clock을 소요하고 idle clock으로 계산하기 때문에 clock과 idleTime을 증가 시킵니다.

Context Switching 이루어지고 있는 동안에는 다른 처리가 불가하기 때문에 10clock 동안 발생한 프로세스 로딩과 IO 종료 처리를 for문과 checkIoEnd() 함수를 호출해 보완하였습니다.

③
```
if(executeProcess(cur, &ready_q, &wait_q)) continue;
```

```
bool executeProcess(process *cur, struct list_head *ready_q, struct list_head *wait_q){
    // cpu, io, idle 처리하는 함수
    // 프로세스가 종료되거나 idle이 종료될 때 true를 리턴 함으로써 continue가 됨

    if(clock >= opEndTime && cur->pid != 100){  //명령 처리 중(opEndTime > clock) 에는 작동 안함
     (1)if(clock != 0 && clock == opEndTime){   //명령 처리가 끝났을 때 PC 증가
            //printf("%04d CPU: Increase PC \tPID:%03d PC:%03d\n", *clock, cur->pid, cur->PC);
            cur->PC++;
        }

     (2)if(cur->PC < (cur->code_bytes / 2)){         //프로세스가 실행할 코드가 남아 있는지
            if(cur->operations[cur->PC].op == 0){   //op가 0 이면 cpu
                return opCpu(cur); ①
            }else if(cur->operations[cur->PC].op == 1){  //op가 1 이면 io
                return opIO(cur, wait_q); ②
            }
     (3)}else{ //모든 명령이 끝났을 때 프로세스는 종료
            return terminateProcess(cur); ③
        }
    }else if(cur->pid == 100){                  //프로세스가 idle 일 때
        return opIdle(cur, ready_q);④
    }
    return false;
}
```

3번 프로세스 명령처리 단계는 Bool 형의 executeProcess() 함수를 활용합니다.

리턴 값이 true인 경우 continue문이 실행되고, false인 경우 while루프가 계속되며 clock을 소요합니다.

함수는 주로 두 부분으로 나뉩니다.

1. 일반 프로세스

   명령 종료 시간인 opEndTime보다 현재 시간이 작은 경우는 명령 처리가 아직 진행 중임을 나타내므로 false를 반환합니다.

   일반 프로세스의 경우 세 부분으로 나뉩니다.

   (1) 현재 시간과 명령 종료 시간이 같은 경우

   명령 처리가 종료됨을 나타내므로 현재 프로세스의 PC를 증가시켜 수행 코드 위치를 관리합니다.

   (2) 처리할 명령이 있는 경우

   opCpu() 와 oplo() 함수로 cpu 명령과 io 명령을 수행합니다.

   (3) 모든 명령이 처리되어 프로세스가 종료되는 경우

   terminateProcess() 함수를 호출합니다.

①
```
bool opCpu(process *cur){
    opEndTime = cur->operations[cur->PC].len + clock;    //명령 종료시간
    //printf("%04d CPU: OP_CPU START \tPID: %03d len: %03d ends at: %04d PC: %3d\n", *clock, cur->pid, cur->operations[cur->PC].len, *opEndTime, cur->PC);
    return false;
}
```

CPU 명령을 처리하는 opCpu() 함수입니다.
코드의 길이와 현재 시간을 더하여 opEndTime에 명령 종료 시간을 설정합니다..
clock을 소요하는 작업이므로 false를 반환합니다.

②
```
bool opIO(process *cur, struct list_head *wait_q){
    cur->ioEnd = cur->operations[cur->PC].len + clock;
    list_del_init(&cur->ready);             //ready_q -> wait_q
    list_add_tail(&cur->wait, wait_q);
    reschedule = true;                      //ready_q의 상태가 변했으니 리스케쥴 요청
    //printf("%04d CPU: OP_IO START len: %03d ends at: %04d\n", *clock, cur->operations[cur->PC].len, cur->ioEnd);
    return false;  //cpu는 일을 했으니 clock 소모
}
```

IO 명령을 처리하는 oplo() 함수입니다.
checkIoEnd()가 IO작업의 종료를 감지 할 수 있도록 ioEnd에 IO 작업이 완료된 시점을 설정합니다.

IO 작업을 수행하는 프로세스는 대기상태로 전환됩니다.
레디큐에서 프로세스가 제거 되었기 때문에 레디큐의 상태가 변하고
이에 따라 reschedule 플래그를 true로 설정해 리스케쥴링을 요청합니다.

이 작업은 clock을 소요하는 작업이기 때문에 false를 반환합니다.

③
```
bool terminateProcess(process *cur){
    //printf("%04d CPU: Process is terminated PID: %03d PC: %03d\n", *clock, cur->pid, cur->PC);
    prevPid = cur->pid;             //pid 기록
    list_del_init(&cur->ready);     //readyQ에 프로세스 삭제
    terminateNum++;                 //종료된 프로세스 수 카운트
    reschedule = true;              //list_del로 인해 ready_q 의 상태가 변했으니 리스케쥴 요청
    free(cur->operations);
    return true;
}
```

모든 명령이 끝나고 프로세스의 종료를 처리하는 terminateProcess() 함수 입니다.
프로세스가 종료되므로 terminateNum 을 카운트 합니다.

ready_q에서 종료된 프로세스를 제거하기 때문에 ready_q의 상태가 변하고
이에 따라 reschedule 플래그를 true로 설정해 리스케쥴링을 요청합니다.

이 작업은 clock을 소요하지 않으므로 true를 반환해 continue문을 수행합니다.

```
bool opIdle(process *cur, struct list_head *ready_q){
    list_del_init(&cur->ready);

    if(list_empty(ready_q)){       //ready_q가 비었으면
                                    //1. 모든 프로세스가 종료            //(main의 while에서 처리 됨)
                                    //2. 프로세스가 아직 도착하지 않음   // 0
        list_add_tail(&cur->ready, ready_q);
        if(doIdle == false){       //본격적인 idle 타임
            doIdle = true;
            //printf("%04d CPU: OP_IDLE START idleTIME : %d\n", *clock, *idleTime);
        }
        return false;               //clock 소모 해도 됨
    }else if(!list_empty(ready_q)){       //ready_q가 안비어있으면 작업이 남아있다는 소리

        list_add_tail(&cur->ready, ready_q);    //idle을 ready_q 맨 뒤로 보냄
        reschedule = true;              //리스케쥴 요청;
        if(doIdle){             //idle 처리하다가 끝나는 상황
            doIdle = false;
            idleTime++;       //true를 보내 continue를 할 예정이기 때문에
            clock++;          //여기서 처리
            //printf("%04d CPU: OP_IDLE END  idleTime : %d\n", *clock, *idleTime);
        }
        return true;
    }
}
```

Idle 프로세스를 처리하는 opIdle() 함수입니다. 함수는 두 부분으로 나뉩니다.

(1) ready_q에 IDLE 외의 다른 프로세스가 존재하지 않는 경우

CPU는 IDLE 프로세스를 처리하며 doIdle 플래그를 true로 변경해 main()에서idleTime을 카운트하게 됩니다.

(2) ready_q에 IDLE 외의 다른 프로세스가 존재하는 경우

IDLE 프로세스는 우선순위가 낮으므로 다른 프로세스로 전환해야 합니다.

ready_q의 상태가 변화했으므로 reschedule 플래그를 true로 변경합니다.

CPU가 IDLE 프로세스를 처리 중 이었다면 doIdle 플래그를 false로 변경합니다.

함수가 true를 반환함에 따라 main 함수에서 추가적인 clock 소모 없이 continue 문이 실행됩니다.

따라서 true를 반환하기 전에, clock과 idleTime을 보정합니다.

```
        if(executeProcess(cur, &ready_q, &wait_q)) continue;
        // cpu, io, idle 처리하는 함수~
        // bool 형으로 return으로 true가 넘어올 때(ex. terminated, idle)
        // clock 소모 방지로 continue;


        if(doIdle){         //doIdle 이 true일 때 idleTime 증가
            idleTime++;
    }

        clock++;
    }

    printf("*** TOTAL CLOCKS: %04d IDLE: %04d UTIL: %2.2f%%\n", clock, idleTime, (float)(clock - idleTime) * 100 / clock );

        //printf("DONE. Freeing to processes in job queue\n");
    list_for_each_entry_safe_reverse(cur, next, &job_q, job){
        //printf("PID: %03d\tARRIVAL: %03d\tCODESIZE: %03d\tPC: %3d\n", cur->pid, cur->arrival_time, cur->code_bytes, cur->PC);
        list_del_init(&cur->job);
        free(cur);
    }

    return 0;
}
```

executeProcess 함수가 종료된 후

doIdle의 플래그가 true인 경우에만 idleTime이 카운트됩니다.

루프를 돌며 모든 프로세스가 처리 되며

jobQnum 과 terminateNum이 같은 경우 while루프는 종료됩니다.

전제 수행 clocks, idle clocks 그리고 CPU 활용률을 포함한 Final report를 출력하고

할당된 메모리를 해제하며 프로그램은 종료됩니다.

test.bin 수행 결과 및 JOTA 제출 결과

Os1-1

```
ubuntu@jcode-client1-181:~/hw1/os1,2,3$ cat ../tests/test1.bin | ./os1-1
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
```

os202246109의 운영체제 과제 1-1 제출

View source
다시 제출

컴파일 경고

```
oshw11c.c: In function 'main':
oshw11c.c:141:17: warning: unused variable 'codes' [-Wunused-variable]
  141 |     code_tuple *codes;
      |                 ^~~~~
oshw11c.c:140:20: warning: unused variable 'next' [-Wunused-variable]
  140 |     process *cur, *next;
      |                    ^~~~
oshw11c.c:149:13: warning: ignoring return value of 'fread' declared with attribute 'warn_unused_result' [-Wunused-result]
  149 |             fread(cur->operations, cur->code_bytes, 1, stdin); //////////욤 codetuple 사이즈 때문에 안 읽힘;;;~> 주소전달
      |             ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Execution Results

✔ ✔ ✔ ✔ ✔

Test case #1: AC [0.004s,1.02 MB] (2/2)
Test case #2: AC [0.003s,1.02 MB] (2/2)
Test case #3: AC [0.003s,1.02 MB] (2/2)
Test case #4: AC [0.003s,1.02 MB] (2/2)
Test case #5: AC [0.003s,1.02 MB] (2/2)

Resources: 0.017s, 1.02 MB
Maximum single-case runtime: 0.004s
Final score: 10/10 (10.0/10 points)

Os1-2

```
ubuntu@jcode-client1-181:~/hw1/os1,2,3$ cat ../tests/test1.bin | ./os1-2
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
0004 CPU: OP_IO START len: 255 ends at: 0259
0016 CPU: Loaded PID: 002       Arrival: 016    Codesize: 002   PC: 000
0271 CPU: OP_IO START len: 072 ends at: 0343
*** TOTAL CLOCKS: 0525 IDLE: 0347 UTIL: 33.90%
```

os202246109의 운영체제 과제 1-2 제출

View source
다시 제출

컴파일 경고

```
oshw12c.c: In function 'main':
oshw12c.c:239:9: warning: ignoring return value of 'fread' declared with attribute 'warn_unused_result' [-Wunused-result]
  239 |         fread(cur->operations, cur->code_bytes, 1, stdin);
      |         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
oshw12c.c:285:17: warning: 'next' may be used uninitialized [-Wmaybe-uninitialized]
  285 |                 switchProcess(&idleTime, &clock, &prevpid, cur, next, &job_q, &ready_q);
      |                 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Execution Results

✔ ✔ ✔ ✔ ✔

Test case #1: AC [0.003s,1.02 MB] (2/2)
Test case #2: AC [0.003s,1.02 MB] (2/2)
Test case #3: AC [0.003s,1.02 MB] (2/2)
Test case #4: AC [0.003s,1.02 MB] (2/2)
Test case #5: AC [0.003s,1.02 MB] (2/2)

Resources: 0.015s, 1.02 MB
Maximum single-case runtime: 0.003s
Final score: 10/10 (10.0/10 points)

Os1-3

```
ubuntu@jcode-client1-181:~/hw1/os1,2,3$ cat ../tests/test1.bin | ./os1-3
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
0205 CPU: Switched      from: 002       to: 001
0220 CPU: Switched      from: 001       to: 100
0259 IO : COMPLETED! PID: 000   IOTIME: 259     PC: 001
0270 CPU: Switched      from: 100       to: 000
*** TOTAL CLOCKS: 0270 IDLE: 0090 UTIL: 66.67%
```

os202246109의 운영체제 과제 1-3 제출

View source
다시 제출

컴파일 경고

```
oshw13c.c: In function 'main':
oshw13c.c:301:9: warning: ignoring return value of 'fread' declared with attribute 'warn_unused_result' [-Wunused-resu
  301 |         fread(cur->operations, cur->code_bytes, 1, stdin);
      |         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
oshw13c.c:345:9: warning: 'next' may be used uninitialized [-Wmaybe-uninitialized]
  345 |         checkIoEnd(&clock, cur, next, &wait_q, &ready_q);
      |         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Execution Results

✔ ✔ ✔ ✔ ✔

Test case #1: AC [0.003s,1.02 MB] (2/2)
Test case #2: AC [0.003s,1.02 MB] (2/2)
Test case #3: AC [0.003s,1.02 MB] (2/2)
Test case #4: AC [0.003s,1.02 MB] (2/2)
Test case #5: AC [0.003s,1.02 MB] (2/2)

Resources: 0.014s, 1.02 MB
Maximum single-case runtime: 0.003s
Final score: 10/10 (10.0/10 points)

Os1-3 (full result)

```
ubuntu@jcode-client1-181:~/hw1/os1,2,3$ cat ../tests/test1.bin | ./os1-3
PID: 000        ARRIVAL: 000    CODESIZE: 004
0 4
1 255
PID: 001        ARRIVAL: 003    CODESIZE: 006
0 2
1 72
0 5
PID: 002        ARRIVAL: 016    CODESIZE: 002
0 167
PID: 100        ARRIVAL: 000    CODESIZE: 002
255 0
Start Processing. Loaded_procs = 3
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 004   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: OP_CPU START  PID: 000 len: 004 ends at: 0004
0003 CPU: Loaded PID: 001       Arrival: 003    Codesize: 006   PC: 000
0004 CPU: Increase PC   PID:000 PC:000
0004 CPU: OP_IO START len: 255 ends at: 0259
0005 CPU: Reschedule    PID:000
0015 CPU: Switched      from: 000       to: 001
0015 CPU: OP_CPU START  PID: 001 len: 002 ends at: 0017
0016 CPU: Loaded PID: 002       Arrival: 016    Codesize: 002   PC: 000
0017 CPU: Increase PC   PID:001 PC:000
0017 CPU: OP_IO START len: 072 ends at: 0089
0018 CPU: Reschedule    PID:001
0028 CPU: Switched      from: 001       to: 002
0028 CPU: OP_CPU START  PID: 002 len: 167 ends at: 0195
0089 IO : COMPLETED! PID: 001   IOTIME: 089 PC: 001
0195 CPU: Increase PC   PID:002 PC:000
0195 CPU: Process is terminated PID: 002 PC: 001
0195 CPU: Reschedule    PID:002
0205 CPU: Switched      from: 002       to: 001
0205 CPU: OP_CPU START  PID: 001 len: 005 ends at: 0210
0210 CPU: Increase PC   PID:001 PC:002
0210 CPU: Process is terminated PID: 001 PC: 003
0210 CPU: Reschedule    PID:001
0220 CPU: Switched      from: 001       to: 100
0259 IO : COMPLETED! PID: 000   IOTIME: 259 PC: 001
0260 CPU: OP_IDLE END  idleTime : 80
0260 CPU: Reschedule    PID:100
0270 CPU: Switched      from: 100       to: 000
0270 CPU: Process is terminated PID: 000 PC: 002
*** TOTAL CLOCKS: 0270 IDLE: 0090 UTIL: 66.67%
DONE. Freeing to processes in job queue
PID: 100        ARRIVAL: 000    CODESIZE: 002   PC:   0
PID: 002        ARRIVAL: 016    CODESIZE: 002   PC:   1
PID: 001        ARRIVAL: 003    CODESIZE: 006   PC:   3
PID: 000        ARRIVAL: 000    CODESIZE: 004   PC:   2
```

어려웠던 점 및 해결 방안

1. List.h 이해
엄청난 define과 포인터로 범벅인 코드를 보니 정신이 아득해졌습니다…
교수님이 제공해주신 영상과 사이트들을 보며 차근차근 이해해갔습니다. 매크로의 존재도 처음 알았고
Linux list library의 강력한 성능에 감탄하기도 했습니다.

2. coredump
프로그램 실행 시 coredump가 일어나서 번거로웠습니다.
cur에 정보를 입력할 때 list_head 초기화를 해주지 않아 발생했습니다.

3. 무에서 유
본격적인 시뮬레이터를 구성하려는데 정말 막막했습니다.
result.txt 로는 구조가 파악이 안되어서 과제 설명의 full result를 분석하여 구조를 파악했습니다.
파악한 구조를 기반으로 모든 기능을 구현해야겠다고 생각했고 구현했습니다.

4. 작업종료 감지
문제점이 readyQ에 idle만 있을 때 도착하지 않은 프로세스가 있음에도 불구하고 프로그램이 종료됐습니다.
jobQnum, terminateNum을 선언해 두 변수가 같을 때 종료되도록 루프 종료조건을 설정했습니다.

5. idle 건너 뛰기
context switching을 할 때 문제점이
test1 에선 readyQ에 000 100 001 이 있는데 000 -> 001로 바로 스위칭 되고
test6 에선 readyQ에 000 100 이 있는데 000 -> 100 으로 스위칭 되야 합니다.
test1에서 000 -> 100 -> 001 로 안 되고 000 -> 001 로 idle을 건너뛰는 부분에서 막혔습니다.

Switch -> execute -> idle 이렇게 판단하는 기존의 구조에서
idle -> switch -> execute 로 idle프로세스를 먼저 처리하도록 변경하니 해결됐습니다.

6. 구조 재구성
5번을 해결하는 과정에서 main에 코드를 통으로 짰기 때문에 구조를 변경하는데 있어서 까다로웠습니다.
주요 기능들을 함수로 구현하여 구조변경에 용이하게 했습니다.
함수로 구현 중 전달해야 할 인자들이 많아 전역변수로 처리해서 단순화했습니다.

7. idleTime 오차
Final report에서 idletime이 맞지 않는 불상사가 일어났습니다.
프로그램의 구조를 fullresult로 작성했기 때문에 프로그램의 결과와 result.txt의 내용을 pdf의 full result 처럼 변환하여 idletime을 일일히 비교하며 계산했습니다.
허점을 파악한 결과, reschedule플래그를 설정하는 구조로 변경하여 문제를 해결했습니다.

Test case #2: WA [0.003s, 1.01 MB] (0/2)

```
Your output
0000 CPU: Loaded PID: 000      Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100      Arrival: 000    Codesize: 002   PC: 000
0002 CPU: Loaded PID: 001      Arrival: 002    Codesize: 010   PC: 000
0009 CPU: Loaded PID: 002      Arrival: 009    Codesize: 002   PC: 000
0063 CPU: Switched      from: 000       to: 001
0090 CPU: Switched      from: 001       to: 002
0146 IO : COMPLETED! PID: 001   IOTIME: 146     PC: 001
0156 CPU: Switched      from: 002       to: 001
0265 CPU: Switched      from: 001       to: 100
0378 IO : COMPLETED! PID: 001   IOTIME: 378     PC: 003
0389 CPU: Switched      from: 100       to: 001
*** TOTAL CLOCKS: 0440 IDLE: 0164 UTIL: 62.73%
```

```
Correct Answer

0000 CPU: Loaded PID: 000      Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100      Arrival: 000    Codesize: 002   PC: 000
0002 CPU: Loaded PID: 001      Arrival: 002    Codesize: 010   PC: 000
0009 CPU: Loaded PID: 002      Arrival: 009    Codesize: 002   PC: 000
0063 CPU: Switched      from: 000       to: 001
0090 CPU: Switched      from: 001       to: 002
0156 CPU: Switched      from: 002       to: 100
0156 IO : COMPLETED! PID: 001   IOTIME: 146     PC: 001
0167 CPU: Switched      from: 100       to: 001
0276 CPU: Switched      from: 001       to: 100
0389 IO : COMPLETED! PID: 001   IOTIME: 389     PC: 003
0400 CPU: Switched      from: 100       to: 001
*** TOTAL CLOCKS: 0451 IDLE: 0175 UTIL: 61.20%
```

8 / 10 WA | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전
10 / 10 AC | C	운영체제 과제 1-3 os202246109 14시간 전

8. 동시 발생 케이스

레포트에 결과를 첨부하려고 jota에 다시 제출하다 발견했습니다.
jota에선 스위칭이 먼저 이루어지고 complete이 출력되는데
제 코드는 complete이 먼저 출력되고 switching이 되는 문제입니다.

jota에서 프로세스가 랜덤으로 생성되기 때문에 디버깅하기 어려웠습니다.
디버깅에 용이하도록 원하는 프로세스를 txt파일로 받아서 바이너리 파일로 변환해주는 gen.c 를 만들었습니다.

hw1 > in.txt

```
0 0 2
0 53
1 2 10
0 16
1 67
0 98
1 124
0 51
2 9 2
0 56
```

```
ubuntu@jcode-client1-181:~/hw1$ ./gen
ubuntu@jcode-client1-181:~/hw1$ hexdump out.bin
0000000 0000 0000 0000 0000 0002 0000 3500 0001
0000010 0000 0002 0000 000a 0000 1000 4301 6200
0000020 7c01 3300 0002 0000 0009 0000 0002 0000
0000030 3800
0000032
```

1) io종료가 io시작 or 프로세스 도착과 겹칠 때
Reschedule 플래그를 true로 설정해서 continue 문이 실행되어 리스케줄링을 시작하기 전
프로세스 도착과 io종료 체크가 실행되어 문제가 발생합니다.
리스케줄 전 구문을 if문으로 묶어서 리스케줄 플래그가 true면 스위칭이 먼저 일어나도록 처리했습니다.

```c
    //jobQ에 들어있는거랑 종료된 프로세스 수가 같아질 때 까지
while(jobQnum != terminateNum){

    //Load
    list_for_each_entry(cur, &job_q, job){          //job_q 순회 해서
        if(clock == cur->arrival_time){             //도착시간이랑 시간이 같으면
            list_add_tail(&cur->ready, &ready_q);   //ready_q에 추가
            printf("%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n", clock, cur->pid, cur->arrival_time, cur->code_bytes, cur->PC);
        }
    }

    checkIoEnd(cur, next, &wait_q, &ready_q);   //Io작업이 끝났는지 확인하는 함수;
```

↓

```c
while(jobQnum != terminateNum){
    //Load
    if(!reschedule){
        list_for_each_entry(cur, &job_q, job){          //job_q 순회 해서
            if(clock == cur->arrival_time && reschedule == false){          //도착시간이랑 시간이 같으면 && 리스케줄 요청시 스위칭으로 인한 로딩 중복 방지
                list_add_tail(&cur->ready, &ready_q);   //ready_q에 추가
                printf("%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n", clock, cur->pid, cur->arrival_time, cur->code_bytes, cur->PC);
            }
        }
    checkIoEnd(cur, next, &wait_q, &ready_q);   //Io작업이 끝났는지 확인하는 함수;
}
```

2) io종료가 프로세스 종료와 겹칠 때

명령 종료 -> **io감지** -> 종료 감지 -> continue -> 리스케줄

모든 명령이 끝나고 프로세스의 종료를 감지하기 전 io종료 감지가 먼저 되는 문제입니다.

명령 종료 -> 종료 감지 -> continue -> 리스케줄

while문이 실행되고 종료감지를 어떻게 해야 하는지 고민을 했습니다.

```
if(!reschedule){

    list_for_each_entry(cur, &job_q, job){          //job_q 순회 해서
        if(clock == cur->arrival_time && reschedule == false){              //도착시간이랑 시간이 같으면 && 리스케줄 요청시 스위칭으로 인한 로딩 중복 방지
            list_add_tail(&cur->ready, &ready_q);   //ready_q에 추가
            printf("%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n", clock, cur->pid, cur->arrival_time, cur->code_bytes, cur->PC);
        }
    }

    //checkIoEnd(cur, next, &wait_q, &ready_q);   //Io작업이 끝났는지 확인하는 함수;

    list_for_each_entry_safe(cur, next, &wait_q, wait){  //wait_q를 순회해서
        if(clock >= cur->ioEnd && opEndTime != cur->ioEnd){                 //IO 명령이 끝나면
            list_del_init(&cur->wait);              //wait_q -> ready_q
            list_add_tail(&cur->ready, &ready_q);
            printf("%04d IO : COMPLETED! PID: %03d\tIOTIME: %03d\tPC: %03d\n", clock, cur->pid, cur->ioEnd, (cur->PC) - 1);
        }
    }

}
```

io종료 체크가 opEndTime 과 ioEnd가 같으면 건너 뛰도록 수정했습니다.

checkIoEnd함수를 수정하니 전체에 영향이 가서 while문 초반의 checkIoEnd대신
새로운 조건을 추가한 checkIoEnd의 동작 코드를 작성했습니다.

```
ubuntu@jcode-client1-181:~/hw1$ cat out.bin | ./os1
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0002 CPU: Loaded PID: 001       Arrival: 002    Codesize: 010   PC: 000
0009 CPU: Loaded PID: 002       Arrival: 009    Codesize: 002   PC: 000
0063 CPU: Switched      from: 000       to: 001
0090 CPU: Switched      from: 001       to: 002
0146 IO : COMPLETED! PID: 001   IOTIME: 146     PC: 001
0156 CPU: Switched      from: 002       to: 001
0265 CPU: Switched      from: 001       to: 100
0378 IO : COMPLETED! PID: 001   IOTIME: 378     PC: 003
0389 CPU: Switched      from: 100       to: 001
*** TOTAL CLOCKS: 0440 IDLE: 0164 UTIL: 62.73%
```

(수정 전)

```
ubuntu@jcode-client1-181:~/hw1$ cat out.bin | ./os1
0000 CPU: Loaded PID: 000       Arrival: 000    Codesize: 002   PC: 000
0000 CPU: Loaded PID: 100       Arrival: 000    Codesize: 002   PC: 000
0002 CPU: Loaded PID: 001       Arrival: 002    Codesize: 010   PC: 000
0009 CPU: Loaded PID: 002       Arrival: 009    Codesize: 002   PC: 000
0063 CPU: Switched      from: 000       to: 001
0090 CPU: Switched      from: 001       to: 002
0156 CPU: Switched      from: 002       to: 100
0156 IO : COMPLETED! PID: 001   IOTIME: 146     PC: 001
0167 CPU: Switched      from: 100       to: 001
0276 CPU: Switched      from: 001       to: 100
0389 IO : COMPLETED! PID: 001   IOTIME: 389     PC: 003
0400 CPU: Switched      from: 100       to: 001
*** TOTAL CLOCKS: 0451 IDLE: 0175 UTIL: 61.20%
```

(수정 후)

수정 전 결과는

Io 종료

002 -> 001로 전환되고

수정 후 결과는

002 -> 100

Io 종료

100 -> 001 로 전환됩니다.

수정 후의 결과의 효율이 수정 전의 효율보다 더 낮은걸 확인했습니다.