

# 과제 #1

---

(1-1) Linux List Library 기반 연결 리스트 구현

(1-2, 1-3) 멀티프로그래밍 기반 배치 시스템 시뮬레이터

# 과제 내용 및 제출 방법

- 프로그램 1개 작성: /home/ubuntu/hw1/os1.c (반드시 지정된 이름으로 작성)
  - stdin 으로부터 이진파일 형태의 프로세스 정보를 입력받아, (과제 0)
  - List Library를 이용한 연결 리스트 형태로 관리하며, (1-1)
  - 멀티프로그래밍 기반 배치 시스템 형태로 처리하는 프로그램 (1-2, 1-3)
- 제출 방법, 내용 및 기한
  - LMS “과제1” 제출
    - 보고서와 소스코드 3개를 함께 압축 (zip) 해서 하나의 파일로 제출
    - 소스코드는 os1-1.c os1-2.c os1-3.c 3개 모두 각각 제출
  - 보고서 내용
    - 표지 포함 총 A4 12장 이하, PDF 형식으로 제출 (이외 형식은 10%p 감점)
    - 간략한 자율 진도표 포함 (for Iterative and incremental development)
    - 주석이 포함된 전체 코드와 각 주요한 부분에 대한 서술, test1.bin 을 이용한 수행 결과
    - JOTA “2021 운영체제 과제 1-1, 1-2, 1-3” 제출 결과 캡처 (분량에 포함되지 않음)
    - 과제 수행 시 어려웠던 점 및 해결 방안
  - 기한: 4/14 (수) 23:59 (지각 감점: 5%p / 12H, 1주 이후 제출 불가)

# Iterative and incremental development

- 반복적, 점진적 프로그래밍 습관을 기르기 위함
- 도달하고자 하는 최종 목표까지 단계를 잘게 나누어 하나씩 검증, 달성하며 진행
  - 단계별로 백업 파일을 남기는 것도 좋은 습관 (Git 사용 시, Commit 의 단위가 됨)
- 아래 예시를 참고하여 각자 **자율적으로 진도표**를 간략하게 작성해서, **보고서에 반드시 첨부할 것**

단계	완료 여부
List 자료구조 파악: 예제 수행	○
list_for_each_entry()를 이용한 순회	○
list_for_each_entry_safe_reverse()...	○
과제 1-1 완료 (JOTA 확인)	○
Idle process 구현	
...	
과제 1-2 완료 (JOTA 확인)	
...	
과제 1-3 완료 (JOTA 확인)	

---

# 1-1. Linux List Library 기반

## 연결 리스트 구현



# 입력되는 이진 프로세스 정보의 형태

- Process tuple: 프로세스 정보와 코드로 구성
  - 입력되는 이진 데이터는 N개의 process tuple 로 구성
  - 각 tuple의 크기는 코드의 크기에 따라 가변적
- 프로세스 정보 (고정 크기)
  - PID: 프로세스 ID
  - 도착시간: 프로세스가 실행을 시작한 시간
  - 코드길이: 프로그램 코드의 길이 (바이트 단위, 짝수)
- 코드 (가변 크기)
  - Code tuple 의 집합: 1 tuple = 2 Bytes
    - 예) 코드 길이가 10 Bytes인 경우, 5개의 tuple 로 구성
  - Code tuple: 각 1 바이트 크기의 동작과 길이로 구성
    - 동작: 시스템에 요청하는 작업의 종류 (예. 00 -> CPU 작업, 01 -> IO 작업)
    - 길이: 해당 동작을 수행하는데 걸리는 시간
    - 예) 00 05 = CPU 작업을 5 만큼의 시간 동안 수행

```
typedef struct {  
    int pid;           //ID  
    int arrival_time;  //도착시간  
    int code_bytes;    //코드길이(바이트)  
} process;
```

# 과제 내용: os1-1.c, JOTA 1-1

- Linux List Library 기반 연결 리스트 구현
  - 과제 설명에 첨부된 list.c 파일의 내용을 복사하여 사용 (JOTA 채점을 위해 파일은 1개로 유지)
- 과제 1에서 사용할 리스트들
  - Job queue: 이진파일로부터 로드된 process tuple들을 관리 (1-1 구현 내용)
    - Linux List Library 를 활용하여 프로세스 정보들을 이중 연결 리스트로 저장
    - 각 process tuple 에서 code tuples 는 연속된 메모리 공간을 할당해 저장
    - 참고: 시뮬레이터에서만 존재. 실제 OS에는 없음
  - ready queue: CPU 작업을 대기 중인 프로세스들을 관리
  - wait queue: IO 작업의 완료를 대기 중인 프로세스들을 관리
- 과제 1-1 결과물
  - 과제 0번 결과 코드를 기반으로 Job queue를 구현하고,
  - 로드된 프로세스 정보를 각각 job queue에 저장한 후,
  - 프로세스 정보를 로드된 순서의 역순으로 출력
    - 각 프로세스 정보의 code tuples은 프로세스 정보 출력 이후, 순서대로 출력



# test1.bin

- 3개의 프로세스 정보가 저장됨
  - 0번 프로세스: 도착시간=0 코드길이=4 (0x04)
  - 1번 프로세스: 도착시간=1 코드길이=6 (0x06)
  - 2번 프로세스: 도착시간=16 (0x10) 코드길이=2 (0x02)
  - (Little endian 방식으로 저장된 이진 파일)
- 짧은 operations 로 debugging 이 용이하도록 수정하였음
  - 유의할 점: 0번 프로세스의 긴 IO 처리 시간 (0xFF): 모든 프로세스가 종료된 이후에도 IO 작업은 끝나지 않을 수 있음

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	00	00	00	00	00	00	00	04	00	00	00	00	04	01	FF
00000010	01	00	00	00	03	00	00	00	06	00	00	00	00	02	01	48
00000020	00	05	02	00	00	00	10	00	00	00	02	00	00	00	00	A7

operation length

# JOTA: 1-1

## 2021운영체제 과제 1-1

- LMS(구버전) 과제 1 참조

stdin 으로부터 Binary 형태의 프로세스 정보와 코드를 읽어들이 아래와 같이 역순으로 출력하시오.

### 1-1.

프로세스 정보를 연결 리스트로 로드하여 역순으로 출력

출력 형식

위: 각 프로세스별 정보 아래: code tuples 출력

```
"PID: %03d\tARRIVAL: %03d\tCODESIZE: %03d\n"
"%d %d\n"
```

출력 예 (과제 포함 **test1.bin** 이용)

```
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
```





# JOTA: 1-1 수행 결과 w/ test1.bin

---

```
ubuntu@41983:~/hw1$ cat test.bin | ./a.out
PID: 002          ARRIVAL: 016          CODESIZE: 002
000 167
PID: 001          ARRIVAL: 003          CODESIZE: 006
000 002
001 072
000 005
PID: 000          ARRIVAL: 000          CODESIZE: 004
000 004
001 255
```



---

1-2, 1-3

# Multiprogramming-based Batch System Simulator



# 과제 내용: 시뮬레이터 기본 구조

- 모든 프로세스 정보는 미리 job queue 에 로드되어 있음 (과제 1-1)
    - Idle 프로세스 추가. 다른 프로세스가 모두 wait 상태일때만 Idle 프로세스가 수행됨 (pid=100, idle operation code = 0xFF)
  - Simulator 는 무한루프를 돌며, clock 단위로 operation processing
    - clock: 임의의 시간 단위. 동작 길이의 단위. 0 부터 시작.
      - 예. Code tuple 이 “00 05” 인 경우: CPU 작업을 5 clocks 진행
    - 각 프로세스는 Clock == Arrival time 이면 ready queue에 순서대로 삽입
      - Job queue 에는 프로그램이 종료될 때까지 그대로 유지
      - 즉, 각 프로세스는 arrival time 이후, job queue 와 ready or wait queue 양쪽에 동시에 존재
    - Operation processing: 각 프로세스의 Code tuple 을 순서대로 하나씩 처리
      - 각 프로세스는 자신의 operations 가 모두 끝난 다음 (terminated)에는 job queue 에만 존재
    - 모든 프로세스 (idle 제외) 의 작업이 종료되면, 시뮬레이터를 종료. Final report 출력
  - Final report
    - 전체 수행된 clocks 와 IO 처리, context switching 등으로 인한 Idle clocks, CPU 활용률 출력
- \*\*\* TOTAL CLOCKS: 0525 IDLE: 0347 UTIL: 33.90%

# 과제 내용: Operation Processing

- Program Counter (PC)로 각 프로세스의 수행 진척도 (수행 중인 코드 위치) 관리
  - a register in a computer processor that contains the address (location) of the instruction being executed at the current time
  - 각 프로세스의 코드 수행 위치를 저장. 0부터 시작하고, 각 operation 이 끝나면 1씩 증가
- 1. CPU 작업인 경우, 동작 길이 만큼 계속해서 clock 증가
  - 시뮬레이터이므로 실제로 어떤 작업을 처리하지는 않음
  - IO 작업의 종료 확인 등 각 clock 마다 필요한 작업 처리
- 2. IO 작업인 경우, 아래 두 가지 버전으로 작성
  - IO 작업 중에 다른 프로세스를 동작시키지 못하는 경우 (w/o multiprogramming, os1-2.c)
    - 다른 프로세스로 스위칭 하지 않고, CPU는 Idle 상태로 IO 작업의 종료를 대기
  - IO 작업 중에 다른 프로세스를 동작시키는 경우 (w/ multiprogramming, os1-3.c)
    - 프로세스를 대기 상태로 전환
    - 현재 프로세스를 wait queue 의 맨 뒤에 삽입
    - Ready queue 의 맨 앞에 있는 프로세스를 선택 (단순 FIFO 스케줄링 정책. Idle 은 가장 낮은 우선순위)
    - 해당 프로세스에서 마지막으로 처리한 operation 다음부터 processing 진행 (PC 이용)

# 과제 내용: Context Switching

- 실행 중인 프로세스가 작업을 모두 종료하였거나, IO 작업을 수행하기 위해 wait 상태로 전환된 경우, ready queue 의 가장 앞쪽에 저장된 프로세스를 선택하여 작업을 재개함 (Idle 제외)
- 만약 더 이상 작업할 프로세스가 남아있지 않은 경우, 시뮬레이터 종료 (Idle 제외)
- Context switching 은 10 clocks 를 소요함: idle clocks 로 계상
- Context switching 이루어지고 있는 동안에는 다른 처리가 불가함.
  - IO 작업 종료 처리, 새로운 process arrival 등의 처리는 모두 context switching 종료 이후에 이루어짐
  - 예를 들어 100 에 끝나는 IO 작업이 있는 상황에서 95 부터 context switching 이 시작되어 105에 끝난다면, 해당 IO작업의 종료는 105가 되어야 OS가 처리함)
- (참고) 최대한 점진적으로, 결과를 확실하게 확인하며 단계적으로 구현할 것
  - 어떻게 동작하는게 맞는지 먼저 손으로 써보고, 수행 결과와 비교하면서 수정해나갈 것



# test1.bin

- 3개의 프로세스 정보가 저장됨
  - 0번 프로세스: 도착시간=0 코드길이=4 (0x04)
  - 1번 프로세스: 도착시간=1 코드길이=6 (0x06)
  - 2번 프로세스: 도착시간=16 (0x10) 코드길이=2 (0x02)
  - (Little endian 방식으로 저장된 이진 파일)
- 짧은 operations 로 debugging 이 용이하도록 수정
  - 유의할 점: 0번 프로세스의 긴 IO 처리 시간 (0xFF): 모든 프로세스가 종료된 이후에도 IO 작업은 끝나지 않을 수 있음

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	00	00	00	00	00	00	00	04	00	00	00	00	04	01	FF
00000010	01	00	00	00	03	00	00	00	06	00	00	00	00	02	01	48
00000020	00	05	02	00	00	00	10	00	00	00	02	00	00	00	00	A7

# Example: full result for 1-3 w/ test1.bin

```
1  0 0 4
2  0 4
3  1 255
4  1 3 6
5  0 2
6  1 72
7  0 5
8  2 16 2
9  0 167
10 100 0 2
11 255 0
12 Start Processing. loaded_procs = 3
13 0000 CPU: Loaded PID: 000  Arrival: 000  Codesize: 004  PC: 000
14 0000 CPU: Loaded PID: 100  Arrival: 000  Codesize: 002  PC: 000
15 0000 CPU: OP_CPU START len: 004 ends at: 0004
16 0003 CPU: Loaded PID: 001  Arrival: 003  Codesize: 006  PC: 000
17 0004 CPU: Increase PC  PID:000 PC:000
18 0004 CPU: OP_IO START len: 255 ends at: 0259
19 0005 CPU: Reschedule  PID: 000  Status: 03
20 0015 CPU: Switched from: 000  to: 001
21 0015 CPU: OP_CPU START len: 002 ends at: 0017
22 0016 CPU: Loaded PID: 002  Arrival: 016  Codesize: 002  PC: 000
23 0017 CPU: Increase PC  PID:001 PC:000
24 0017 CPU: OP_IO START len: 072 ends at: 0089
25 0018 CPU: Reschedule  PID: 001  Status: 03
```



# 수행 결과 예시

```
26 0028 CPU: Switched from: 001 to: 002
27 0028 CPU: OP_CPU START len: 167 ends at: 0195
28 0089 IO : COMPLETED! PID: 001 IOTIME: 089 PC: 001
29 0195 CPU: Increase PC PID:002 PC:000
30 0195 CPU: Process is terminated PID:002 PC:001
31 0195 CPU: Reschedule PID: 002 Status: 04
32 0205 CPU: Switched from: 002 to: 001
33 0205 CPU: Increase PC PID:001 PC:001
34 0205 CPU: OP_CPU START len: 005 ends at: 0210
35 0210 CPU: Increase PC PID:001 PC:002
36 0210 CPU: Process is terminated PID:001 PC:003
37 0210 CPU: Reschedule PID: 001 Status: 04
38 0220 CPU: Switched from: 001 to: 100
39 0220 CPU: OP_IDLE START
40 0259 IO : COMPLETED! PID: 000 IOTIME: 259 PC: 001
41 0260 CPU: Reschedule PID: 100 Status: 01
42 0270 CPU: OP_IDLE ENDS
43 0270 CPU: Switched from: 100 to: 000
44 0270 CPU: Increase PC PID:000 PC:001
45 0270 CPU: Process is terminated PID:000 PC:002
46 *** TOTAL CLOCKS: 0270 IDLE: 0090 UTIL: 66.67%
47 DONE. Freeing the processes in job queue
48 PID: 100 ARRIVAL: 000 CODESIZE: 002 PC: 000
49 PID: 002 ARRIVAL: 016 CODESIZE: 002 PC: 001
50 PID: 001 ARRIVAL: 003 CODESIZE: 006 PC: 003
51 PID: 000 ARRIVAL: 000 CODESIZE: 004 PC: 002
```





# JOTA: 1-2

---

## 2021운영체제 과제 1-2

---

- LMS(구버전) 과제 1 참조

stdin 으로부터 Binary 형태의 프로세스 정보와 코드를 읽어들이고, 아래와 같이 출력하시오.

### 1-2.

멀티프로그래밍 배치 시스템으로 선입선출 (FIFO) 스케줄링을 수행하되, IO 작업 시 다른 프로세스로 스위칭 하지 않고, CPU는 Idle 상태로 IO 작업의 종료를 대기한다.

출력 형식

CPU clock 진행에 따라,

```
로드된 프로세스 정보: "%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n"
```

```
IO 작업 시작: "%04d CPU: OP_IO START len: %03d ends at: %04d\n"
```

```
모든 작업 종료 후, 최종 리포트: "*** TOTAL CLOCKS: %04d IDLE: %04d UTIL: %2.2f%\n"
```



# JOTA: 1-2 수행 결과 w/ test1.bin

---

```
ubuntu@41983:~/hw1$ cat test.bin | ./a.out
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%
```



# JOTA: 1-3

## 2021운영체제 과제 1-3

- LMS(구버전) 과제 1 참조

stdin 으로부터 Binary 형태의 프로세스 정보와 코드를 읽어들이어, 아래와 같이 출력하시오.

### 1-3.

멀티프로그래밍 배치 시스템으로 선입선출 (FIFO) 스케줄링을 수행하되, IO 작업 시 다른 프로세스로 스위칭하여 작업을 수행한다. IO 작업의 종료가 끝나면, 해당 프로세스를 다시 ready queue 로 삽입한다.

### 출력 형식

CPU clock 진행에 따라,

```
로딩된 프로세스 정보: "%04d CPU: Loaded PID: %03d\tArrival: %03d\tCodesize: %03d\tPC: %03d\n"
IO 작업 종료: "%04d IO : COMPLETED! PID: %03d\tIOTIME: %03d\tPC: %03d\n"
CPU 작업 전환: "%04d CPU: Switched\tfrom: %03d\tto: %03d\n"
모든 작업 종료 후, 최종 리포트: "*** TOTAL CLOCKS: %04d IDLE: %04d UTIL: %2.2f%%\n"
```

Copy



# JOTA: 1-3 수행 결과 w/ test1.bin

---

```
ubuntu@41983:~/hw1$ cat test.bin | ./a.out
```

```
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%
```

