과제 #4 : Scheduling

○ 과제 목표

- 스케줄링 이해
- SSU OS에서 µcLinux 및 Linux Kernel Version 2.6 초창기에 사용된 O(1) 스케줄링 구현
  - ✓ 프로세스의 우선 순위에 기반하여 프로세스를 스케줄함

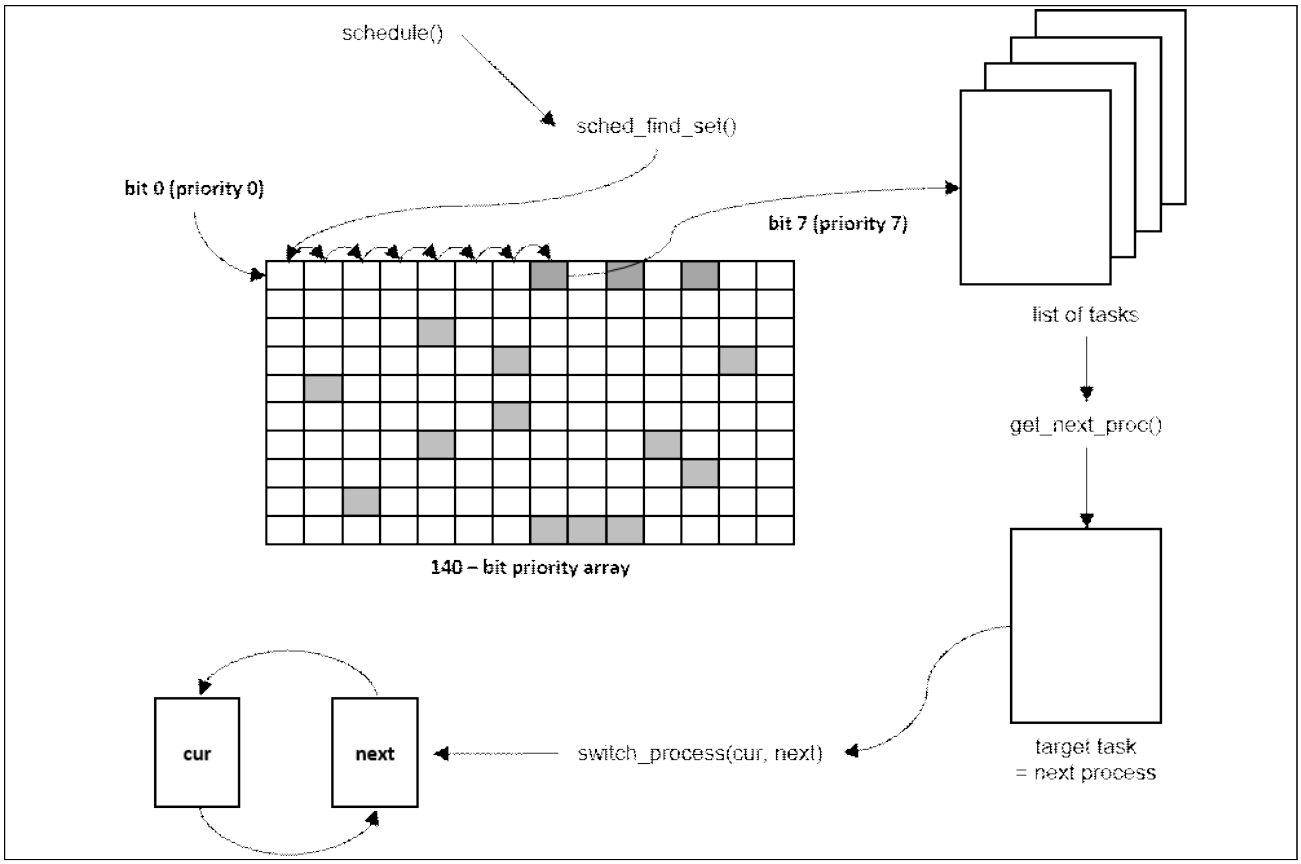
○ 기본 배경 지식

- 스케줄링
  - ✓ 스케줄링은 다중 프로그래밍을 가능하게 하는 운영 체제의 동작 기법이다. 운영 체제는 프로세스들에게 CPU 등의 자원 배정을 적절히 함으로써 시스템의 성능을 개선할 수 있다.
  - ✓ 비선점형 스케줄링(Non-preemptive Scheduling) : 어떤 프로세스가 CPU를 할당 받으면 그 프로세스가 종료되거나 입출력 요구가 발생하여 자발적으로 중지될 때까지 계속 실행되도록 보장한다. 순서대로 처리되는 공정성이 있고 다음에 처리해야 할 프로세스와 관계없이 응답 시간을 예상할 수 있으며 선점 방식보다 스케줄러 호출 빈도가 낮고 문맥 교환에 의한 오버헤드가 적다. 일괄 처리 시스템에 적합하며, CPU 사용 시간이 긴 하나의 프로세스가 CPU 사용 시간이 짧은 여러 프로세스를 오랫동안 대기시킬 수 있으므로, 처리율이 떨어질 수 있다는 단점이 있다.
  - ✓ 선점형 스케줄링(Preemptive Scheduling) : 어떤 프로세스가 CPU를 할당받아 실행 중에 있어도 다른 프로세스가 실행 중인 프로세스를 중지하고 CPU를 강제로 점유할 수 있다. 모든 프로세스에게 CPU 사용 시간을 동일하게 부여할 수 있다. 빠른 응답시간을 요하는 대화식 시분할 시스템에 적합하며 긴급한 프로세서를 제어할 수 있다. '운영 체제가 프로세서 자원을 선점'하고 있다가 각 프로세스의 요청이 있을 때 특정 요건들을 기준으로 자원을 배분하는 방식이다.
- O(1) Scheduler
  - ✓ µcLinux에서 최초로 구현된 스케줄러이며, 후에 Linux Kernel Version 2.6 초창기에 사용되었다.
  - ✓ 완벽한 SMP 확장성, 좋은 인터액티브 성능을 보장하며, 공평성을 유지, 프로세서가 많은 멀티프로세서 환경으로 확장이 용이하도록 하는 선점형 스케줄러이다.
  - ✓ 프로세스 우선순위를 두 가지로 나눠서 구현, 하나는 -20 ~ +19사이의 값을 가지는 nice값과 0~99사이의 값을 가지는 real-time priority값이 있다.
  - ✓ 기존의 Linux나 Unix 스케줄러와는 다르게 Time Slice를 재계산하는 과정이 필요치 않다.
  - ✓ 우선순위만을 기반으로 Time Slice를 할당하기 때문에 스위칭 발생 비율이 0이 될 수 있으므로 기아(Starvation)문제와 동시에 불공평성의 문제가 생긴다.

○ 과제 내용

- 다운로드 받은 SSU OS는 현재 단순 FIFO스케줄러로 구현되어 있음
  - ✓ 내부 자료구조를 이해하고 응용하여 SSU OS O(1) 스케줄러를 구현 할 것
- SSU OS O(1) 스케줄러
  - ✓ µcLinux와 Linux Kernel Version 2.6에서는 nice값으로 -20 ~ 19사이의 값을 취했으나, SSU OS에서는 0~40사이의 값을 취하도록 구현
  - ✓ 0번 프로세스(idle)는 schedule() 함수만 계속해서 호출하도록 구현

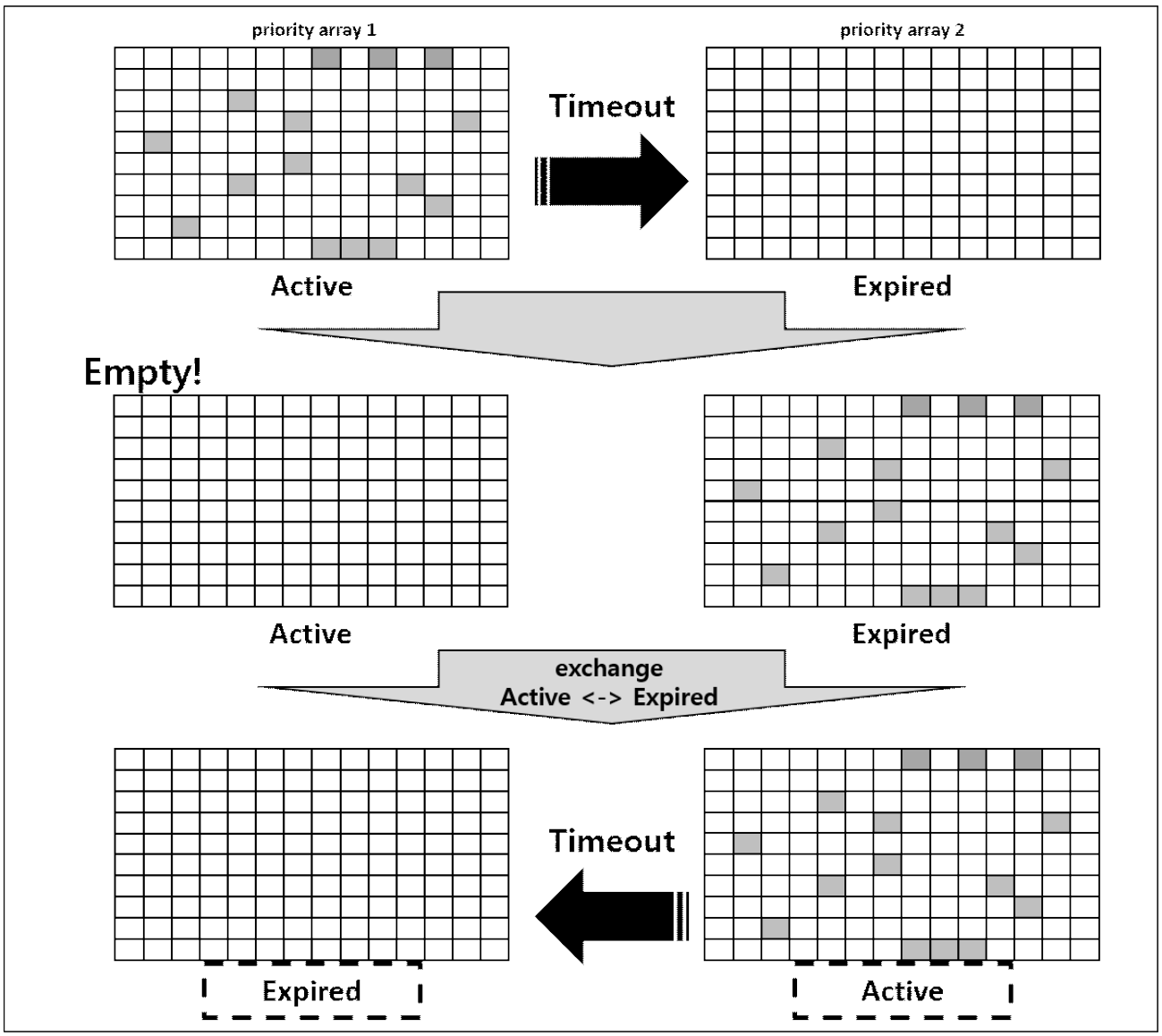
- SSU OS O(1) 스케줄러 동작 루틴



[그림 1] SSU OS O(1) 스케줄러의 대략적인 동작 과정

- ✓ 0번 프로세스가 schedule() 함수 호출을 통하여 스케줄링할 프로세스 선택 및 스케줄링
- ✓ 0번 프로세스를 제외한 프로세스는 schedule() 함수 호출을 통해 0번 프로세스로 Context Switch함
- ✓ sched_find_set() 함수 호출을 통해 priority array에서 가장 우선순위가 높은 리스트를 탐색
- ✓ 기본 수준 사용자 우선 순위(priority)는 nice와 rt_priority의 합으로 결정
- ✓ 우선 순위의 값이 낮을수록 우선순위가 높은 프로세스를 뜻함
- ✓ 우선순위가 높은 프로세스부터 수행되기 때문에, 기아 현상이 나타날 수 있음
- ✓ 프로세스마다 주어지는 기본 time slice는 60 tick
- ✓ schedule() 함수가 호출되는 상황은 주어진 time slice를 모두 소비하거나, I/O 작업을 요청하는 경우
- ✓ priority array는 active, expired 두 종류가 존재
  - active : 스케줄링의 대상이 되는 프로세스들이 존재하는 배열. I/O와 같은 인터럽트가 발생하는 상황으로 인해 time_slice를 모두 소비하지 못한 프로세스는 이 배열에 존재함
  - expired : active에 있던 프로세스는 할당받은 time_slice를 모두 소비하면 이 배열로 들어오게됨
  - active 배열에 프로세스가 하나도 존재하지 않게 되는 시점에 expired 배열과 교체를 수행
- ✓ 가로 방향을 우선적으로 탐색하며, 탐색 중 비어 있지 않은 리스트가 있다면 그 리스트를 get_next_proc() 함수의 인자로 전달
- ✓ get_next_proc() 함수는 인자로 받은 리스트에서 실행가능 하면서 가장 앞에 있는 프로세스를 리턴
- ✓ schedule() 함수에서는 리턴 받은 프로세스와 현재 프로세스를 Context Switch
- ✓ 만약, 실행 중이던 프로세스가 I/O 요청시, 프로세스의 상태를 바꾸고 스케줄링을 수행

✓ I/O 대기 중인 프로세스는 스케줄링 되지 않음
✓ I/O 처리는 proc_sleep() 함수 호출로 대체함
✓ Context Switch 도중 0번 프로세스를 제외한 나머지 프로세스들의 tick이 증가하지 않도록 구현



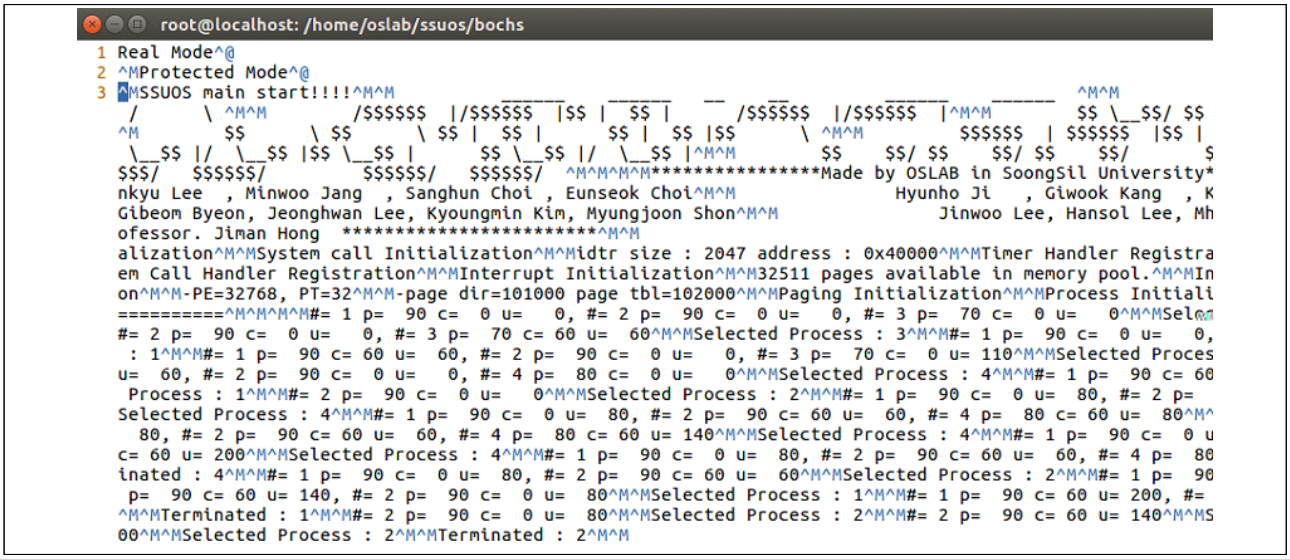
[그림 2] Active와 Expired가 교체되는 과정

○ 디버깅 팁

- bochsrc 수정(ssuos/bochs/bochsrc)
- bochs 터미널 크기가 한정적이므로, 터미널에 출력되는 모든 내용을 확인할 수 있도록 설정을 추가

```
com1:enabled=1, mode=file, dev=test.out
```

- 위의 문장을 bochsrc 파일에 추가할 경우, ssuos/bochs/ 디렉토리에 test.out이라는 파일이 생성됨. test.out 파일에는 터미널에 출력된 내용이 전부 들어있음
- 파일의 내용을 확인할 때 아래와 같이 vi 에디터를 사용하거나, cat 명령어를 사용할 경우 깔끔하게 보이지 않으므로 od와 같은 명령어를 사용해서 내용을 확인할 것



○ 과제 수행 방법

- (1) schedule() 함수 수정
  - ✓ schedule() 함수 내부에서 0번 프로세스만이 다른 프로세스들을 스케줄링할 수 있고, 그 외의 프로세스는 0번 프로세스와 Context Switch 하도록 함수를 수정
  - ✓ Context Switching 도중에는 다른 인터럽트가 발생하더라도 인터럽트 핸들러가 수행되지 않도록 해야 함
  - ✓ 출력 형식은 아래의 Case1, Case2의 실행결과와 비슷하게만 맞출것

- (2) sched_find_set() 함수 구현
  - ✓ 주어진 priority array 자료구조 및 priority 값에 알맞게 탐색하는 루틴 구현
  - ✓ 비어있지 않은 리스트 발견 시, 그 리스트를 get_next_proc() 함수의 인자로 호출

- (3) 스케줄링에 사용되는 자료구조
  - ✓ process 구조체에 priority 구현
    - nice값과 rt_priority값을 합친 값을 사용 (priority = nice + rt_priority)
  - ✓ process 구조체 내부의 각 변수에 적절한 값이 들어가도록 구현
    - time_slice : 프로세스가 스케줄링 된 이후 다시 스케줄링 되기 전까지 CPU를 사용한 시간(tick)
    - time_used : 프로세스가 생성된 이후 CPU를 사용한 총합 시간(tick)
    - time_sleep : 프로세스의 I/O 처리 시, 프로세스의 다시 일어날 시간(tick)을 명시
  - ✓ priority array 구현
    - priority array로 사용될 배열을 struct list 자료구조를 이해하고 2차원 배열로 2개 구현. 배열의 행과 열의 길이는 커널 소스를 참조하여 설정
    - list.h, list.c를 참조하여 필요한 함수를 적절하게 사용

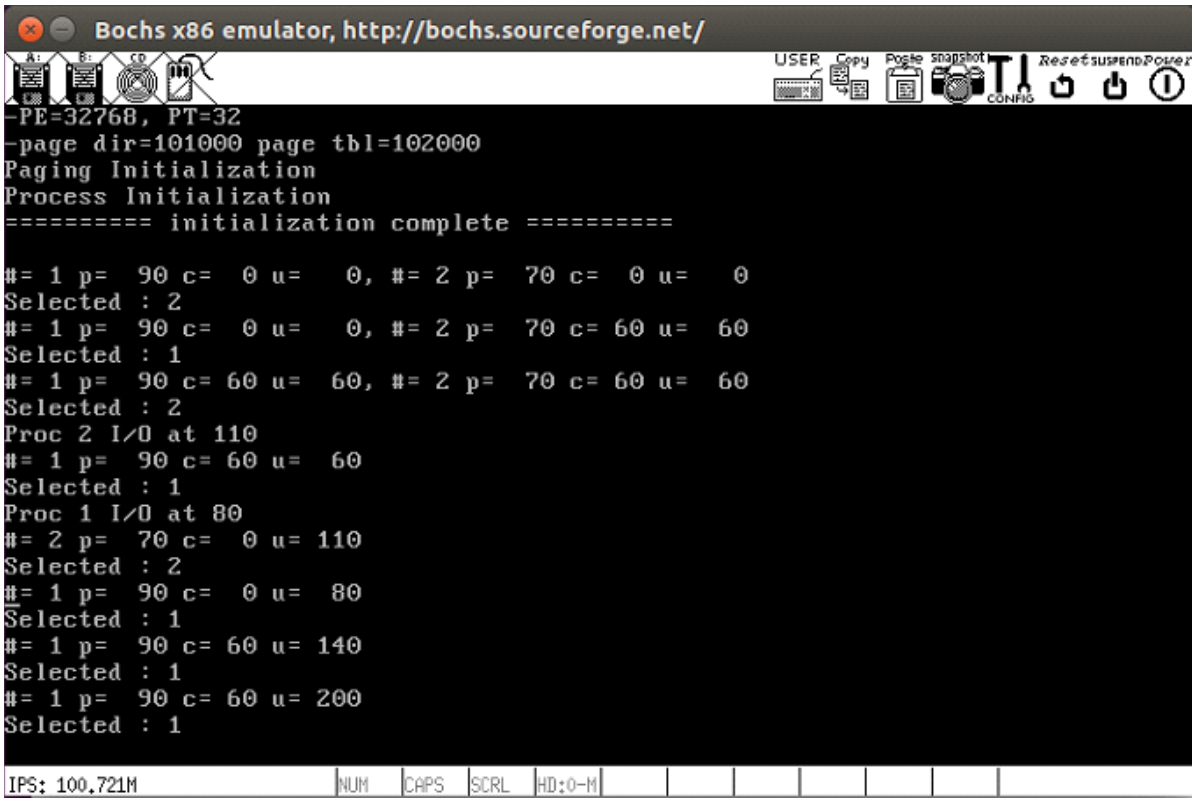
- (4) 실행 될 프로세스 구현
  - ✓ kernel1_proc(), kernel2_proc(), kernel3_proc()을 아래의 Case를 보고 구현

○ 과제 수행 결과

- [Case 1], [Case 2]에 알맞게 구현해야 함 **(제출 시 [Case 1]이 실행되도록 구현)**
- 총 수행시간에 I/O 수행 시간은 포함되지 않음

PID	nice	rt_priority	I/O 제외 총 수행시간 (tick)	I/O 수행 시점 (tick)	참고
1	20	70	200	80	kernel1_proc() 호출
2	20	50	120	110	kernel2_proc() 호출

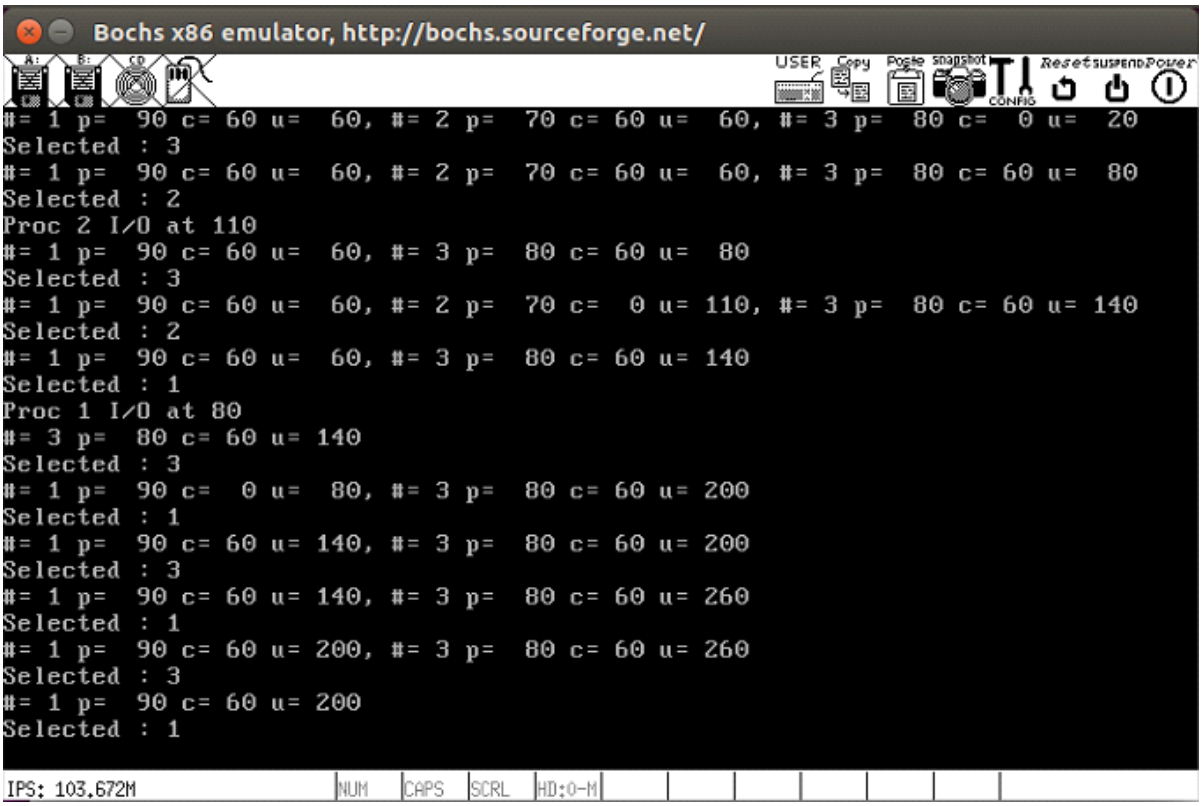
[Case 1] 수행될 프로세스 2개의 실행 정보



[Case 1 결과] 프로그램 수행 시 결과화면
(# = pid, p = 우선순위(priority),
c = 스케줄 된 이후 CPU사용시간, u = 프로세스의 CPU 총 사용시간)

PID	nice	rt_priority	I/O 제외 총 수행시간 (tick)	I/O 수행 시점 (tick)	참고
1	20	70	200	80	kernel1_proc() 호출
2	20	50	120	110	kernel2_proc() 호출
3	15	65	300	20	kernel3_proc() 호출

[Case 2] 수행될 프로세스 3개의 실행 정보



[Case 2 결과] 프로그램 수행 시 결과화면의 일부
(# = pid, p = 우선순위(priority),
c = 스케줄 된 이후 CPU사용시간, u = 프로세스의 CPU 총 사용시간)

[Case 2 실행 결과]

```
Keyboard Handler Registration
System Call Handler Registration
Interrupt Initialization
32511 pages available in memory pool
Interrupt Initialization
Palloc Initialization
PE=32768, PT=32
page dir=101000 page tbl=102000
Paging Initialization
Process Initialization
========== initialization complete ==========
#= 1 p= 90 c= 0 u= 0, #= 2 p= 70 c= 0 u= 0, #= 3 p= 80 c= 0 u= 0
Selected : 2
#= 1 p= 90 c= 0 u= 0, #= 2 p= 70 c= 60 u= 60, #= 3 p= 80 c= 0 u= 0
Selected : 3
Proc 3 IO at 20
#= 1 p= 90 c= 0 u= 0, #= 2 p= 70 c= 60 u= 60
```

```
Selected : 1
#= 1 p=  90 c= 60 u=  60, #= 2 p=  70 c= 60 u=  60, #= 3 p=  80 c=  0 u=  20
Selected : 3
#= 1 p=  90 c= 60 u=  60, #= 2 p=  70 c= 60 u=  60, #= 3 p=  80 c= 60 u=  80
Selected : 2
Proc 2 IO at 110
#= 1 p=  90 c= 60 u=  60, #= 3 p=  80 c= 60 u=  80
Selected : 3
#= 1 p=  90 c= 60 u=  60, #= 2 p=  70 c=  0 u= 110, #= 3 p=  80 c= 60 u= 140
Selected : 2
#= 1 p=  90 c= 60 u=  60, #= 3 p=  80 c= 60 u= 140
Selected : 1
Proc 1 IO at 80
#= 3 p=  80 c= 60 u= 140
Selected : 3
#= 1 p=  90 c=  0 u=  80, #= 3 p=  80 c= 60 u= 200
Selected : 1
#= 1 p=  90 c= 60 u= 140, #= 3 p=  80 c= 60 u= 200
Selected : 3
#= 1 p=  90 c= 60 u= 140, #= 3 p=  80 c= 60 u= 260
Selected : 1
#= 1 p=  90 c= 60 u= 200, #= 3 p=  80 c= 60 u= 260
Selected : 3
#= 1 p=  90 c= 60 u= 200
Selected : 1
```

○ 과제 제출

- 2017년 10월 10일 (화) 23시 59분 59초까지 과제 게시판으로 제출

○ 최소 기능 구현

- (1) 스케줄링 과정 수정
- (2) schedule() 함수 수정
- (3) sched_find_set() 함수 구현
- (4) 스케줄링에 사용되는 자료구조 구현
- (5) Case 1에 알맞게 프로세스 실행 주기 구현

○ 배점 기준

- 보고서 20점
  - ✓ 개요 3점
  - ✓ 상세 설계 명세(기능 명세 포함) 7점

✓ 구현 방법 설명(코드 주석 포함) 8점
✓ 실행 결과 2점

- 소스코드 80점
✓ 컴파일 여부 10점(설계 요구에 따르지 않고 설계된 경우 0점 부여)
✓ 실행 여부 70점(0번 프로세스가 스케줄링함 10점, CPU 사용량에 따른 우선순위 부여 15점, I/O 로 대기 중인 프로세스의 우선순위를 제외한 프로세스들의 우선순위 수정여부 15점, 이론상의 스케줄링과 동일하게 동작(tick 오차에 따른 1~2 정도의 값 차이는 인정) 20점)