과제 #4 : Scheduling

○ 과제 목표

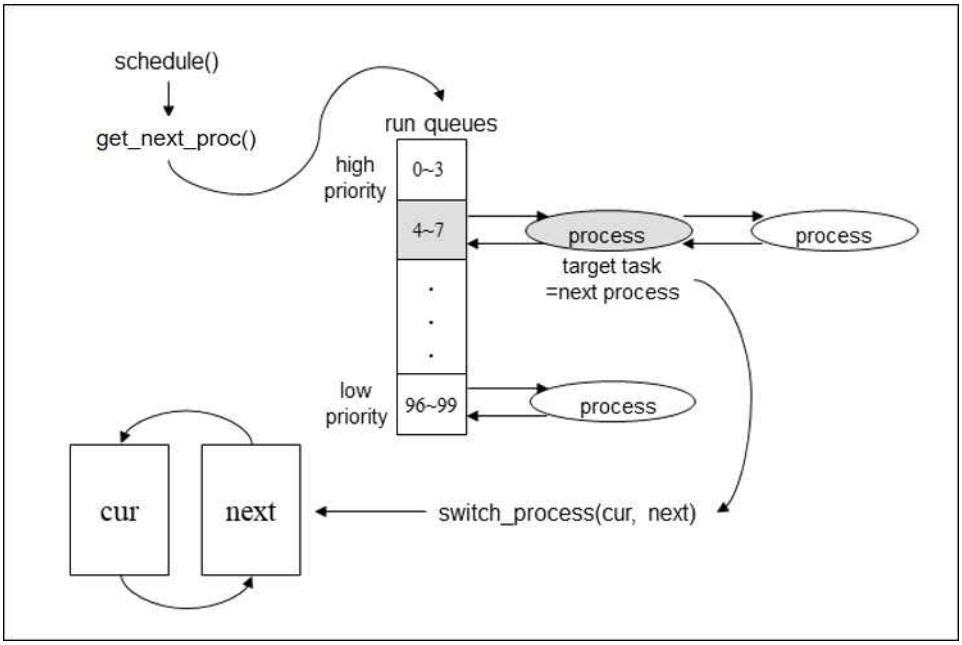
- 실제 스케줄링 기법 구현
- FreeBSD 5.4 이후 구현된 ULE(non-interactive)를 SSUOS에 구현
  - ✓ 프로세스의 우선순위 + 프로세스가 사용한 CPU 시간을 스케줄링에 반영

○ 기본 배경 지식

- 스케줄링
  - ✓ 스케줄링은 다중 프로그래밍을 가능하게 하는 운영체제 커널의 기본 기능
  - Non-preemptive Scheduling
  - Preemptive Scheduling

○ 과제 내용

- 다운로드 받은 SSUOS는 현재 preemptive 기반 FIFO 스케줄러로 구현되어 있음
- 내부 자료구조를 이해하고 응용하여 ULE(non-interactive) SSUOS 스케줄러 구현
- 기존 SSUOS 스케줄러 동작 방식
  - ✓ 기본적으로 60 tick 마다 스케줄링 됨
  - ✓ schedule() 함수가 호출되는 상황은 아래와 같음
  - 주어진 time_slice를 모두 소비한 경우
  - I/O 작업을 요청하는 경우
  - 특정 프로세스가 종료하는 경우
- 새로운 SSUOS 스케줄러 동작 방식(과제)
  - ✓ 새로운 SSUOS에서는 priority값으로 0~99 사이의 값을 취하도록 구현
  - ✓ 0번 프로세스(idle)는 schedule()함수만 계속해서 호출하도록 구현
  - ✓ 문맥교환 중 모든 프로세스의 tick이 증가하지 않도록 구현



[그림 1] 새로운 SSUOS 스케줄러의 동작 과정

- ✓ 0번 프로세스가 schedule() 함수 호출을 통하여 스케줄링 할 프로세스 선택하고 context switching
- ✓ 0번 프로세스를 제외한 프로세스는 schedule() 함수 호출을 통해 0번 프로세스로 context switching
- ✓ 우선순위(priority)의 값이 낮을수록 우선순위가 높은 프로세스를 뜻함
- ✓ 우선순위가 높은 프로세스부터 수행되기 때문에 기아 현상이 나타날 수 있음
- ✓ 각 run queue는 우선순위 4개를 포함하고 있음
- ✓ 현재 프로세스와 context switch 할 프로세스 탐색 과정
  - run queues에서 우선순위가 높으며 비어있지 않은 리스트 탐색
  - 위에서 찾은 리스트에서 실행가능하면서 가장 앞에 있는 프로세스 선택
- ✓ 만약, 실행 중이던 프로세스가 I/O 요청 시, 프로세스의 상태를 바꾸고 schedule() 함수 호출
- ✓ I/O 대기 중인 프로세스는 스케줄링 되지 않음
- ✓ Context Switch 도중 0번 프로세스를 제외한 나머지 프로세스들의 tick이 증가하지 않도록 구현

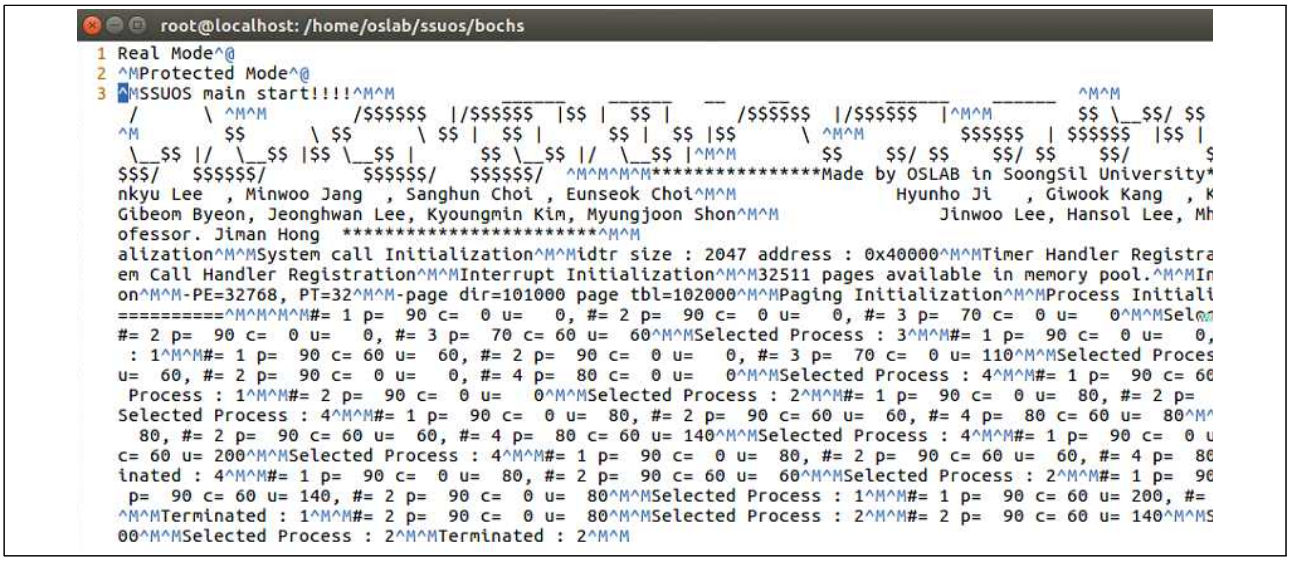
○ 디버깅 팁

- 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 경

우 깔끔하게 보이지 않으므로 od와 같은 명령어를 사용해서 내용을 확인할 것



○ 과제 수행 내용

- (0) 스케줄링에 사용되는 자료구조 및 변수

✓ struct process 구조체 : 멤버 변수에 스케줄링을 위한 적절한 값 할당

- priority: 우선순위 값으로 범위는 0~99이며, idle 프로세스는 99로 고정.
- time slice: 프로세스가 스케줄링 된 이후 다시 스케줄링 되기 전까지 CPU를 사용한 시간 (tick)으로 time quantum을 의미함
- time used: 프로세스가 생성된 이후 CPU를 사용한 총합 시간 (tick)
- time sleep: I/O 처리(proc_sleep() 호출) 시, 프로세스의 다시 일어날 시간(tick)을 명시

✓ runq 구현

- 각 큐는 리스트 구조체로 구현 (list.h, list.c 참고)
- run queues에서 프로세스가 삽입될 큐의 인덱스는 (프로세스의 priority / 4) 값임

- (1) priority 값 재계산

✓ 매 클럭 tick 마다 time_slice, time_used 값을 증가시켜 CPU 사용량을 계산함

✓ 스케줄링 된 이후 매 30tick 마다 CPU 사용량에 따라 priority값을 재계산함.

$$priority \mathrel{+}= \frac{time\,slice}{10}$$

✓ priority 값을 변경한 후 run queues의 인덱스를 재계산하여 해당 인덱스 리스트 맨 마지막에 삽입 (인덱스가 변경되지 않는 경우에도 리스트의 맨 마지막에 삽입)

✓ 스케줄 된 이후 CPU 사용시간을 나타내는 time_slice 값이 클수록 프로세스 우선순위 값이 커져 우선순위가 낮아짐

✓ 0번 프로세스는 idle 프로세스로 priority값이 변하지 않음 (priority값 99)

- (2) schedule() 수정

✓ schedule()에서 0번 프로세스만이 다른 프로세스들을 스케줄링 해야 하며 ,0번이 아닌 프로

세스는 0번 프로세스로 스케줄링 되어야 함

✓ Context Switching 도중에는 다른 인터럽트가 발생하더라도 인터럽트 핸들러가 수행되지 않도록 해야 함

✓ 출력 형식은 아래의 Case1, Case2의 실행결과와 비슷하게만 맞출 것

- (2) get_next_proc() 함수 구현

✓ 주어진 runq 자료구조 및 priority 값에 알맞게 탐색하는 루틴 구현

- (3) 실행 될 프로세스 내용 구현

✓ kernel1_proc(), kernel2_proc(), kernel3_proc() 을 아래의 Case를 보고 구현

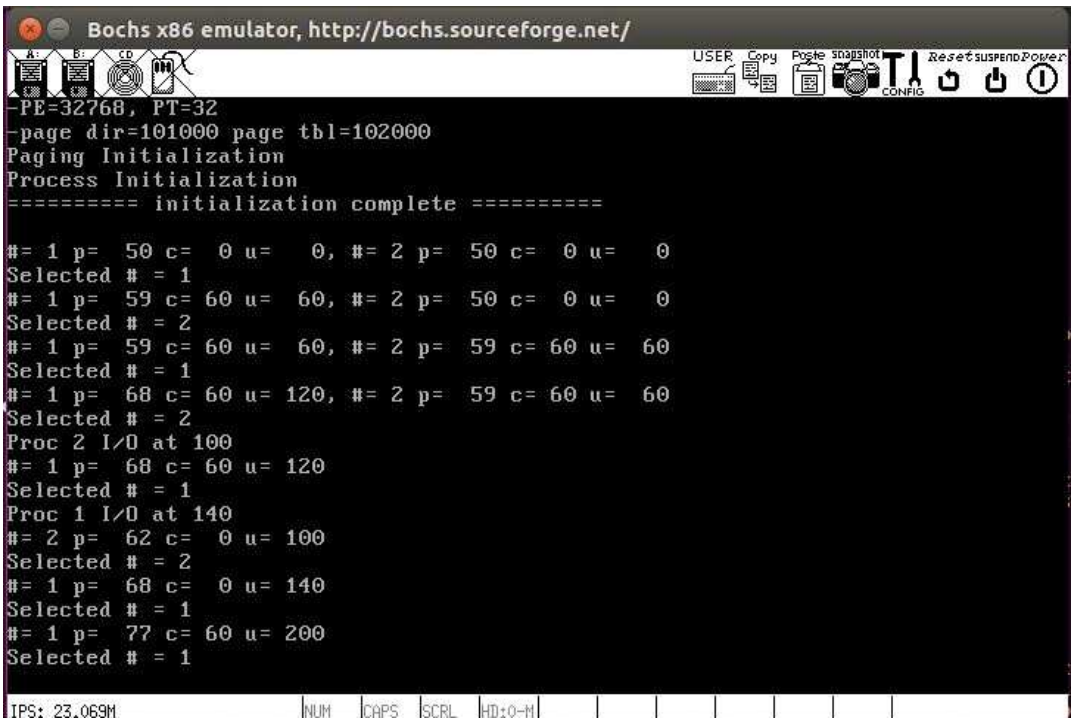
✓ I/O 처리는 proc_sleep()으로 대체

- (4) 과제 수행 결과

- [Case 1], [Case2]에 알맞게 구현해야 함 (제출 시 [Case 1]이 실행되도록 구현)
- 총 수행시간에 I/O 수행 시간은 포함되지 않음

PID (순서)	초기 priority	I/O 제외 총 수행시간 (tick)	I/O 수행 시점 (tick)	참고
1	50	200	140	kernel1_proc() 호출
2	50	120	100	kernel2_proc() 호출

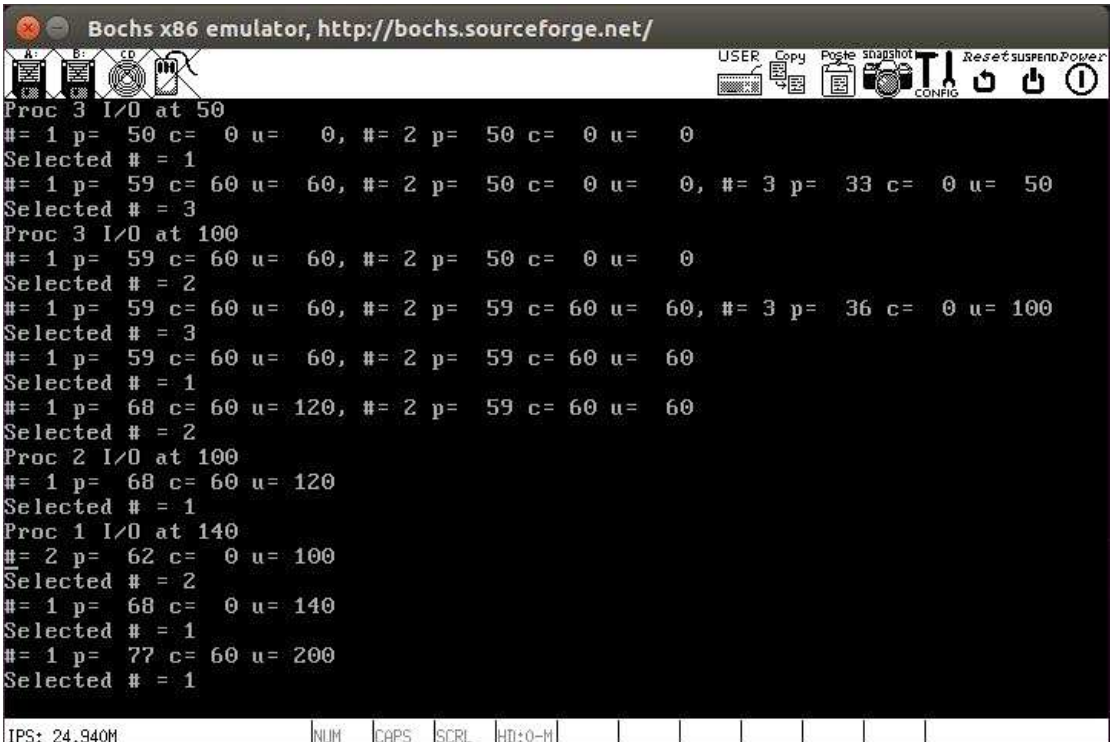
[Case 1] 수행될 프로세스 2개의 실행 정보



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

PID (순서)	초기 priority	I/O 제외 총 수행시간 (tick)	I/O 수행 시점 (tick)	참고
1	50	200	140	kernel1_proc() 호출
2	50	120	100	kernel2_proc() 호출
3	30	150	50, 100	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=  50 c=  0 u=   0, #= 2 p=  50 c=  0 u=   0, #= 3 p=  30 c=  0 u=   0
Selected # = 3
Proc 3 I/O at 50
#= 1 p=  50 c=  0 u=   0, #= 2 p=  50 c=  0 u=   0
Selected # = 1
#= 1 p=  59 c= 60 u=  60, #= 2 p=  50 c=  0 u=   0, #= 3 p=  33 c=  0 u=  50
Selected # = 3
Proc 3 I/O at 100
#= 1 p=  59 c= 60 u=  60, #= 2 p=  50 c=  0 u=   0
Selected # = 2
#= 1 p=  59 c= 60 u=  60, #= 2 p=  59 c= 60 u=  60, #= 3 p=  36 c=  0 u= 100
Selected # = 3
#= 1 p=  59 c= 60 u=  60, #= 2 p=  59 c= 60 u=  60
Selected # = 1
#= 1 p=  68 c= 60 u= 120, #= 2 p=  59 c= 60 u=  60
Selected # = 2
Proc 2 I/O at 100
#= 1 p=  68 c= 60 u= 120
Selected # = 1
Proc 1 I/O at 140
#= 2 p=  62 c=  0 u= 100
Selected # = 2
#= 1 p=  68 c=  0 u= 140
Selected # = 1
#= 1 p=  77 c= 60 u= 200
Selected # = 1
```

○ 구현 시 주의할 점

- printk()의 인자가 4개 이상일 경우 간혹 에러 발생 가능하기 때문에 여러 번의 printk()를 사용할 것

○ 과제제출

- 2019년 10월 13일 (일) 23시 59분까지 제출
- 배점 기준

✓ 보고서 10%

- 개요 2%
- 상세 설계 명세(기능 명세 포함) 5%
- 실행 결과 3%

✓ 소스코드 90%

- 컴파일 여부 5%(설계 요구에 따르지 않고 설계된 경우 0점 부여)
- 실행 여부 85% (0번 프로세스가 스케줄링 함 10점, priority 값 재계산 15점, I/O로 대기 중인 프로세스의 우선순위를 제외한 프로세스들의 우선순위 수정여부 15점, 이론상의 스케줄링과 동일하게 동작 (tick 오차에 따른 1~2 정도의 값 차이는 인정) 40점)

- 최소 구현사항

✓ 과제 수행 내용 (1), (2)