과제 #2 : xv6 SSU Scheduler

○ 과제 목표

- xv6의 프로세스 관리 및 스케줄링 기법 이해

- 동적 우선순위 조정이 가능한 다단계 피드백 큐 스케줄링(Multi-level Feedback Queue with Dynamic Priority Adjustment)
  - ✓ 다단계 큐 스케줄링 (Multi-level Queue Scheduling)을 변형하여, 프로세스의 실행 패턴에 따라 동적 우선순위 조정이 가능한 다단계 피드백 큐(Multi-level Feedback Queue, MLFQ) 스케줄링을 xv6에 구현.
  - ✓ 동적 우선순위 : 조정 프로세스가 오랜 시간 동안 실행되거나, 자주 I/O를 대기할 경우 우선순위를 동적으로 조정
  - ✓ 이를 통해 CPU 집중형 작업과 입출력(I/O) 집중형 작업 간의 공정성을 개선하고, 전체 시스템 성능을 최적화

- 시간 제한 기반 큐 이동 : 각 프로세스는 정해진 시간 제한(타임 슬라이스) 내에 완료되지 않으면, 낮은 우선순위 큐로 이동

- 보상 메커니즘: 오랫동안 낮은 우선순위에 머무른 프로세스의 우선순위를 높여주는 보상 메커니즘을 추가

- 성능 분석: 새로운 스케줄링 알고리즘이 시스템 성능에 미치는 영향을 분석

○ 기본 지식

- 스케줄링
  - ✓ 스케줄링은 다중 프로그래밍을 가능하게 하는 운영체제 커널의 기본 기능임
  - ✓ 기존 xv6의 스케줄링 기법은 다음 실행할 프로세스를 process table을 순회하며 RUNNABLE 상태인 프로세스를 순차적으로 선택함

○ 과제 내용

1. 기존 xv6 스케쥴러 분석 - 함수 단위로 상세하게 분석과 함수간 콜 그래프 등은 필수로 포함되어야 함.
   - ✓ 어떻게 분석해야 하는지, 무슨 내용이 들어가야 할지에 대한 질문 받지 않음. 학생들이 스스로 생각해서 소스코드 분석(주석 포함), 함수 콜 그래프(순서도), 기능 등을 명시해야 함.

2. 새로운 SSU Scheduler 구현

- (1) 큐와 프로세스 구조 변경
  - ✓ 각 프로세스는 (1)총 4개의 큐를 중 프로세스가 어떤 큐에 있는지 나타내는 q_level, (2) 프로세스당 Time Quantum(slice) 내에서 cpu 사용시간인 cpu_burst, (3) 프로세스 당 runnable 상태된 후 해당 큐 내에서 대기한 시간인 cpu_wait, (4) 프로세스 당 해당 큐에서 sleeping 상태 시간인 io_wait_time 및 (5) 응용 프로그램의 cpu 총 사용 할당량인 end_time 멤버 변수를 기본적으로 proc struct에 추가. 이 멤버 변수들은 프로세스의 실행 패턴을 추적하는데 사용
  - ✓ 예) 특정 프로세스가 end_time이 300 tick인 경우, 0 번째 큐에서 10 tick + 1 번째 큐에서 20 tick, + 2 번째 큐에서 40 tick + 3번째 큐에서 80 tick 실행 + 계속 3번째 큐에서 80 tick + 계속 3번째 큐에서 70 tick = 300 tick 수행.

- (2) 다단계 피드백 큐 구성
  - ✓ 우선순위 큐의 개수는 4개로, 각 큐는 서로 다른 Time Quantum(slice)을 가짐.
  - ✓ 큐의 레벨은 0~3으로 레벨이 낮을수록 우선순위가 높은 큐임. 각 큐의 Time Quantum(slice) 10 tick(레벨 0,가장 높은 우선순위, 최상위 큐), 20 tick(레벨 1), 40 tick(레벨 2), 80 tick(레벨 3, 가장 낮은 우선 순위, 최하위 큐) 으로 지정
  - ✓ 높은 우선순위 큐는 짧은 Time Quantum을, 낮은 우선순위 큐는 더 긴 Time Quantum을 설정하여, I/O 바운드 프로세스가 높은 우선순위를 유지.

- (3) 프로세스 큐 이동 및 우선순위 조정
  - ✓ 프로세스가 특정 큐에서 Time Quantum(slice) 내에 실행을 완료하지 못하면, 한 단계 낮은 우선순위 큐로 이동
  - ✓ Aging 메커니즘을 통해, 오랫동안(250 tick으로 고정) CPU를 사용하지 않은 프로세스의 우선순위를 한 레벨씩 높여줌

- (4) 스케줄러 변경
  - ✓ 기존 scheduler() 함수를 수정하여, 레벨이 가장 낮은(레벨 0 즉, 우선 순위가 가장 높은) 큐부터 검사하여 다음에 실행할 프로세스를 선정하고 실행
  - ✓ 동적으로 우선순위를 조정하고, 프로세스를 적절한 큐로 이동시킴 (aging 시 위로, 주어진 Time Quantum을 다 사용하면 아래 큐로)
  - ✓ I/O bound 프로세스의 우선순위를 높여 시스템의 성능과 응답 시간을 향상

- (5) 프로세스 생성 시 초기화
  - ✓ idle 프로세스, init 프로세스, 쉘 프로세스는 항상 최하위 레벨(레벨 3, 80 tick)의 우선순위 큐에 존재하고 다른 큐로 이동하지 않는다.
  - ✓ 쉘 프로세스 이후에 생성된 새로운 프로세스는 최상위 레벨(레벨 0, 10 tick)의 우선순위 큐에 추가

3. 새로운 SSU Scheduler 테스트 프로그램

- (1) 스케줄링 테스트를 위해 프로세스의 초기 q_level, cpu_burst, cpu_wait_time, io_wait_time, end_time 값을 설정할 수 있는 set_proc_info() 시스템 콜 추가

✓ set_proc_info(int q_level, int cpu_burst, int cpu_wait_time, int io_wait_time, int end_time);

✓ set_proc_info()는 프로세스 생성시 q_level = 0, cpu_burst = 0, cpu_wait_time = 0, io_wait_time =0 값과 CPU를 할당량(end_time = -1, tick)로 설정. 스케쥴러 테스트 프로그램 시작 시 이 값들은 인자로 받아 처리. 테스트 프로그램 시작 시각 프로세스 마다 set_proc_info(0, 0, 0, 0, 500) 형식으로 지정. 테스트 프로그램의 자식 프로세스는 3개까지만 처리. 쉘 제외 스케쥴 테스트 프로세스 1개 + fork()로 최대 3개 자식 생성. 스케쥴테스트 프로그램은 "start scheduler_test"와 "end of schedueIr_test"만 표준 출력. 다음 〈예시 1-1〉과 〈예시 1-2〉는 1개 프로세스 생성했을 때 예시이며, 예시 1-3〉은 3개의 프로세스를 생성했을 때 예시임.

✓ set_proc_info()을 통해 프로세스의 종료 시간을 설정하지 않으면 프로세스는 프로그램이 정상적으로 완료되면 종료

✓ 해당 테스트 프로그램은 생성한 프로세스가 모두 종료된 후 종료되도록 구현

✓ 디버깅 정보와 함께 테스트 프로그램 실행 예시 첨부

(예시 1-1). scheduler_test.c 실행 결과 set_proc_info(0, 0, 0, 0, 500)//0번째 큐에서부터 시작

```
$ test1-1
start scheduler_test
PID: 4 created                                  // 1개 프로세스 생성, 아래 (2)의 스케쥴링 출력
Set process 4's info complete                   // set_proc_info 시스템 콜 수행
PID: 4 uses 10 ticks in mlfq[0], total(10/500) //0번째 큐에서 10 tick 수행
PID: 4 uses 20 ticks in mlfq[1], total(30/500) //1번째 큐에서 20 tick 수행
PID: 4 uses 40 ticks in mlfq[2], total(70/500) //2번째 큐에서 40 tick 수행
PID: 4 uses 80 ticks in mlfq[3], total(150/500)
PID: 4 uses 80 ticks in mlfq[3], total(230/500)
PID: 4 uses 80 ticks in mlfq[3], total(310/500)
PID: 4 uses 80 ticks in mlfq[3], total(390/500)
PID: 4 uses 80 ticks in mlfq[3], total(470/500)
PID: 4 uses 30 ticks in mlfq[3], total(500/500)
PID: 4, used 500 ticks. terminated               // 프로세스 종료
end of schedueIr_test
```

(예시 1-2). scheduler_test.c 실행 결과 set_proc_info(1, 0, 0, 0, 500) //1번째 큐에서부터 시작

```
$ test1-2
start scheduler_test
PID: 4 created                                  // 1개 프로세스 생성, 아래 (2)의 스케쥴링 출력
Set process 4's info complete                   // set_proc_info 시스템 콜 수행
PID: 4 uses 20 ticks in mlfq[1], total(20/500) //1번째 큐에서 20 tick 수행
PID: 4 uses 40 ticks in mlfq[2], total(60/500) //2번째 큐에서 40 tick 수행
PID: 4 uses 80 ticks in mlfq[3], total(140/500)
PID: 4 uses 80 ticks in mlfq[3], total(220/500)
PID: 4 uses 80 ticks in mlfq[3], total(300/500)
PID: 4 uses 80 ticks in mlfq[3], total(380/500)
PID: 4 uses 80 ticks in mlfq[3], total(460/500)
PID: 4 uses 40 ticks in mlfq[3], total(500/500)
PID: 4, used 500 ticks. terminated
end of schedueIr_test
```

(예시 1-3). scheduler_test.c 실행 결과 첫 번째 set_proc_info(2, 0, 0, 0, 300), 두 번째 set_proc_info(2, 0, 0, 0, 300), 세 번째 set_proc_info(2, 0, 0, 0, 300) //모든 프로세는 2번째 큐에서부터 시작, 모든 프로세스는 300 tick 수행 후 종료

```
$ test1-3
start scheduler_test
PID: 4 created                                  // 1개 프로세스 생성, 아래 (2)의 스케쥴링 출력
PID: 5 created                                  // 1개 프로세스 생성, 아래 (2)의 스케쥴링 출력
PID: 6 created                                  // 1개 프로세스 생성, 아래 (2)의 스케쥴링 출력
Set process 6's info complete // set_proc_info 시스템 콜 수행
PID: 6 uses 40 ticks in mlfq[2], total(40/300) //2번째 큐에서 40 tick 수행, 3번째 큐로 이동
Set process 5's info complete // set_proc_info 시스템 콜 수행
PID: 5 uses 40 ticks in mlfq[2], total(40/300) //2번째 큐에서 40 tick 수행, 3번째 큐로 이동
Set process 4's info complete // set_proc_info 시스템 콜 수행
PID: 4 uses 40 ticks in mlfq[2], total(40/300) //2번째 큐에서 40 tick 수행, 3번째 큐로 이동
PID: 4 uses 80 ticks in mlfq[3], total(120/300) //3번째 큐에서 80 tick 수행
PID: 4 uses 80 ticks in mlfq[3], total(200/300) //3번째 큐에서 80 tick 수행
PID: 6 Aging // 250 tick 대기 후 에이징. 3번째 큐에서 2번째 큐로 이동
PID: 5 Aging // 250 tick 대기 후 에이징. 3번째 큐에서 2번째 큐로 이동
PID: 4 uses 80 ticks in mlfq[3], total(280/300) //3번째 큐에서 80 tick 수행
PID: 5 uses 40 ticks in mlfq[2], total(80/300) //2번째 큐에서 40 tick 수행, 3번째 큐로 이동
PID: 6 uses 40 ticks in mlfq[2], total(80/300) //2번째 큐에서 40 tick 수행, 3번째 큐로 이동
PID: 6 uses 80 ticks in mlfq[3], total(160/300) //3번째 큐에서 80 tick 수행
PID: 6 uses 80 ticks in mlfq[3], total(240/300) //3번째 큐에서 80 tick 수행
PID: 4 Aging // 250 tick 대기 후 에이징. 3번째 큐에서 2번째 큐로 이동
PID: 5 Aging // 250 tick 대기 후 에이징. 3번째 큐에서 2번째 큐로 이동
PID: 6 uses 60 ticks in mlfq[3], total(300/300) //3번째 큐에서 60 tick 수행
PID: 6, used 300 ticks. terminated
PID: 5 uses 40 ticks in mlfq[2], total(120/300) // 2번째 큐에서 40, 3번째 큐로 이동
PID: 4 uses 20 ticks in mlfq[2], total(300/300) //2번째 큐에서 20 tick 수행
PID: 4, used 300 ticks. terminated
PID: 5 uses 80 ticks in mlfq[3], total(200/300) //3번째 큐에서 80 tick 수행
PID: 5 uses 80 ticks in mlfq[3], total(280/300) //3번째 큐에서 80 tick 수행
PID: 5 uses 20 ticks in mlfq[3], total(300/300) //3번째 큐에서 20 tick 수행
PID: 5, used 300 ticks. terminated
end of schedulelr_test
```

- (2) 다음에 실행될 프로세스 선정 과정 스케쥴링 과정 출력 구현

✓ 〈예시 2〉에서 보여주듯이, xv6 빌드 시 “debug=1” 매개변수 전달을 통해 프로세스가 생성된 시점, set_proc_info() 시스템 콜이 호출된 시점, CPU를 사용중인 프로세스가 자신의 타임 슬라이스 만큼 CPU를 사용한 이후 시점, 프로세스가 CPU를 전부 사용하고 종료하는 시점에 프로세스의 정보를 출력함.

☞ 출력 형식은 〈예시 1-1〉, 〈예시 1-2〉, 〈예시 1-3〉 빨간색 부분 참고

☞ 다음 프로세스를 선택할 수 없는 경우에는 출력하지 않음

☞ 〈예시 3〉 및 〈예시 4〉는 debug 매개변수 및 디버그 콜에 따라 현재 프로세스의 pid와 이름을 출력하는 주요 코드로 참고.

(예시 2). xv6 빌드 시 “debug=1”매개변수 전달

```
$ make debug=1 qemu-nox
```

(예시 3). Makefile(아래 내용 추가 필요)

```
ifeq ($(debug), 1)
CFLAGS += -DDEBUG
endif
```

(예시 4). void scheduler(void)(예시 4와 연관)

```
#ifdef DEBUG
        if (p)
                cprintf("PID: %d, NAME: %s,₩n", p->pid, p->name);
#endif
```

4. 위 1, 2, 3을 기반으로 기존 xv6 스케쥴러와 SSU 스케쥴러의 기능 및 성능 비교 분석

- ✓ https://www.usenix.org/system/files/conference/atc18/atc18-bouron.pdf 참고하되 참고 논문과 같이 성능 분석을 제대로 할 필요는 없음.
- ✓ 기능의 차이점과 성능의 차이점을 비교 분석하되, 기능의 차이는 도표를 포함해야 하고 성능의 차이는 반드시 그래프 포함해야 함
- ✓ 관련한 질문 받지 않음. 학생들이 창의적으로 생각해서 비교 분석하기 바람.

○ 과제 주의 사항

- 스케줄링에 사용되는 자료구조 및 변수
  - ✓ struct proc 구조체: proc.h 내에 선언되어 있으며 프로세스의 실행 패턴을 추적하기 위해 변수 추가
- 우선순위 큐 구현
  - ✓ 큐 내에서 우선순위는 io_wait_time 값이 가장 큰 프로세스가 우선순위가 높으며 동일한 io_wait_time 값을 가진 프로세스가 2개 이상 존재할 경우, 나중에 들어온 프로세스의 우선순위가 높음
  - ✓ 나머지 우선순위 큐의 구현 방식은 알아서 구현
- 프로세스의 필드 재계산
  - ✓ 매 클럭 tick 마다 cpu를 사용하고 있는 프로세스는 cpu_burst 값 1 증가
  - ✓ 매 클럭 tick 마다 cpu 사용을 기다리는 프로세스는 cpu_wait 값 1 증가
  - ✓ 매 클럭 tick 마다 I/O 작업이나 어떤 이벤트를 기다리는 프로세스는 io_wait_time 값 1 증가
- 프로세스의 큐 이동
  - ✓ 한 단계 낮은 우선순위 큐로 이동하면 프로세스의 cpu_burst, cpu_wait_time, io_wait_time 값을 0으로 설정
    - 만약 이미 프로세스가 우선순위가 가장 낮은 큐에 있었다면, 프로세스의 cpu_burst 값을 0으로 초기화하고 다른 큐로 이동하지 않음
  - ✓ 프로세스가 자신이 속한 큐에서 CPU를 250 tick 이상 기다렸다면 상위 레벨의 우선순위 큐로 이동
- 필요한 변수 및 자료구조가 있으면 자유롭게 추가 가능.
- 해당 과제는 간단한 스케줄링 함수 구현과 테스트를 위해 CPU 코어 개수를 1개로 제한함
  - ✓ Makefile 수정

○ 과제 제출 마감

- 2024년 10월 29일 (일) 23시 59분까지 구글클래스룸으로 제출
- 보고서 (hwp, doc, docx 등으로 작성)
- xv6에서 변경한 소스코드, 테스트 쉘 프로그램 소스코드, Makefile 등
- 마감시간 이후 24시간까지 지연 제출 가능. 그 이후 제출은 0점 처리. 설계과제 마감시간 이후 지연 제출은 30% 감점.

○ 필수 구현(설치 및 설명 등)

- 1, 2, 3

○ 배점 기준

- 1. 기존 xv6 스케쥴러 분석 : 15점
- 2. 새로운 SSU Scheduler 구현 : 40점
- 3. 새로운 SSU Scheduler 테스트 프로그램 : 30점
- 4. 기존 xv6 스케쥴러와 SSU 스케쥴러의 기능 및 성능 비교 분석 : 15점