과제 #2 : Stride Scheduling in XV6

○ 과제 목표
- 티켓 개수에 비례하여 CPU 시간을 할당하는 결정론적 Stride 스케줄러를 XV6 커널에 구현
- 기존의 우선순위나 가중치 기반 스케줄링과는 다른 방식으로 CPU 분배를 달성하고, 스케줄러를 변경하는 과정 학습
- 커널 수준에서 스케줄링 알고리즘 구현 방법, 프로세스 구조체 확장, 시스템 콜 추가 방법을 복습하고, 스케줄링 기법의 차이점을 이해

○ 기본 지식
- 라운드 로빈 스케줄링
  - ✓ xv6 운영체제의 기본 CPU 스케줄링 방식
  - ✓ 모든 프로세스에 동일한 시간 할당을 부여함
- Lottery 스케줄링
  - ✓ 라운드 로빈 스케줄링 기법을 개선한 스케줄러
  - ✓ 복권 추첨하듯 무작위로 티켓을 뽑아 해당 티켓을 가진 프로세스에 CPU를 할당하는 방식
  - ✓ 프로세스마다 가중치를 주어 CPU 시간을 비례적으로 배분
  - ✓ 프로세스들의 장기적인 공정성을 확보
- Stride 스케줄링
  - ✓ Lottery 스케줄링은 추첨에 의존하기 때문에 단기적인 변동성이 존재하고 실시간성 요구가 있는 상황에서는 예측이 어려움
  - ✓ 위의 단점을 해결하기 위해 Stride 방식이 고안됨
  - ✓ 난수 없이 결정론적 방식으로 티켓 기반 비례 분배를 수행하여 보다 예측 가능하고 안정적인 스케줄링을 제공
  - ✓ 각 프로세스는 티켓 수에 따라 Stride 값을 부여받아 실행 간격을 조절하며, 난수를 사용하지 않고 티켓 비율에 따른 공정한 CPU 분배를 달성
  - ✓ 결과적으로 모든 프로세스는 자신이 보유한 티켓 비율에 근접한 CPU 실행 시간을 얻음

구분	Lottery 스케줄링 (추첨 기반)	Stride 스케줄링 (Stride 기반)
선택 방식	무작위 추첨으로 티켓 당첨 프로세스 선택	pass 최소값으로 결정론적 프로세스 선택 (최소 pass 프로세스)
단기 공정성	단기 편차 존재 – 확률적으로 일부 프로세스가 연속 선택되는 등 변동 가능	단기 편차 미미 – 엄격히 교대 실행되어 티켓 비율에서 크게 벗어나지 않음 (편차가 bounded)
난수 사용	필요 (난수 생성기로 추첨 진행)	불필요 (추첨 없음, 고정된 알고리즘으로 동작)
예측 가능성	낮음 – 실행마다 결과가 달라질 수 있음 (통계적으로만 비례 보장)	높음 – 매 실행 결과가 반복 가능하고 일정한 비례 분배를 보임
구현 복잡도	비교적 단순 – 티켓 합계 및 난수 추첨 구현	약간 복잡 – stride, pass 관리 및 overflow 처리 필요

○ 구현해야 할 내용
- 1. 기본 설정 및 구현
  - ✓ (1) Makefile 설정
    - 첨부한 테스트 파일을 실행시키기 위해 Makefile 수정
    - 파일은 이미 컴파일이 완료된 이진 파일이기 때문에, EXTRA에는 추가시키지 않고, UPROGS에만 추가

(예 1). 첨부된 테스트 파일이 추가된 Makefile 코드 일부

```
… (생략) …

UPROGS=\
        _cat\
        _echo\

… (중략) …

        _wc\
        _zombie\
        _debug_test\  #첨부된 파일은 UPROGS에만 추가
        _syscall_test\
        _scheduler_test\

fs.img: mkfs README $(UPROGS)
        ./mkfs fs.img README $(UPROGS)

… (생략) …
```

✓ (2) proc 구조체 확장 및 stride 스케줄러에게 필요한 상수 정의

(예 2). 확장해야할 proc.h 구조체 예시 및 상수 정의 (상수 값을 제외한 나머지 변수는 자유롭게 구현)

```
… (생략) …

struct proc {

… (중략) …

  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int tickets;
  uint stride;                // 기본값 : 0 (setticket으로 설정)
  uint pass;                 // 기본값 : 0 (setticket으로 설정)
  int ticks;                  // 기본값 : 0
  int end_ticks;             // 기본값 : -1 (양수인 경우 ticks 변수가 end_ticks값이 되면 프로세스 종료)
};

… (중략) …

#define STRIDE_MAX 100000
#define PASS_MAX 15000
#define DISTANCE_MAX 7500
```

✓ (3) 디버깅 코드 구현

- 채점에 필요한 debug 출력 구현
- debug 출력은 본인과 부모의 pid가 모두 2보다 클 때만 출력
- debug 출력 상황은 아래 1, 2, 3과 같음

☞ 1. 커널 모드의 프로세스가 스케줄러에게 CPU 제어권을 넘기기 직전

(예 3-1). 커널 모드의 프로세스가 스케줄러에게 CPU 제어권을 넘기는 시점 (trap.c)

```
… (생략) …

//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{

… (중략) …

  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();
  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
  // Check if the process has been killed since we yielded
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();
}

… (생략) …
```

☞ 2. 프로세스 생성이 완료된 직후

(예 3-2). 프로세스 생성이 완료되는 시점 (proc.c)

```
… (생략) …
int
fork(void)
{
… (중략) …
  np->state = RUNNABLE;
  release(&ptable.lock);

  return pid;
}

… (생략) …
```

☞ 3. 프로세스가 종료되는 시점

(예 3-3). 프로세스가 종료되는 시점 (proc.c)

```
… (생략) …

void
exit(void)
{

… (중략) …

  curproc->state = ZOMBIE;

  sched();
  panic("zombie exit");
}

… (생략) …
```

✓ **출력**

(예 3-4). debug_test 실행 결과 (출력 형식 준수 필수, 틱은 달라질 수 있음)

```
$ debug_test
Process 4 start
Process 4 selected, stride : 0, ticket : 1, pass : 0 -> 0  (1/-1)
Process 4 selected, stride : 0, ticket : 1, pass : 0 -> 0  (2/-1)
Process 4 selected, stride : 0, ticket : 1, pass : 0 -> 0  (3/-1)
Process 4 selected, stride : 0, ticket : 1, pass : 0 -> 0  (4/-1)

… (중략) …

Process 4 selected, stride : 0, ticket : 1, pass : 0 -> 0  (62/-1)
Process 4 selected, stride : 0, ticket : 1, pass : 0 -> 0  (63/-1)
Process 4 selected, stride : 0, ticket : 1, pass : 0 -> 0  (64/-1)
result : 1540746712                                    (debug_test 프로세스 출력)
Process 4 exit
```

- 커널 모드의 프로세스가 스케줄러에게 CPU 제어권을 넘기기 직전 시 debug 출력 내용
- ☞ "Process 4 selected, stride : 0, ticket : 1, pass : 0 -〉 0 (1/-1)"
- ☞ "Process 4 selected, ": Process 뒤에 선택된 프로세스의 pid 를 출력
- ☞ "stride : 0, ticket : 1, " : 선택된 프로세스의 stride 값 및 ticket 값 출력
- ☞ "pass : 0 -〉 0 " : 선택된 프로세스의 pass 값 변화를 출력
- ☞ "(62/-1)" : 선택된 프로세스가 cpu를 점유한 총 틱 수와 종료 틱 수를 출력, 종료 틱이 없다면 -1 출력

- 2. settickets 시스템 콜 구현

✓ (1) int settickets(int tickets, int end_ticks)

- 첫 번째 인자로 입력 받은 티켓을 통해 stride 값을 구하고, 두 번째 인자로 입력 받은 틱 수를 통해 프로세스의 수명을 결정하는 시스템 콜 구현
  - ☞ 첫 번째 인자 int tickets : tickets 값은 1이상 이고 STRIDE_MAX 값 이하여야 함. 아니라면 -1 리턴
  - ☞ 두 번째 인자 int end_ticks : end_ticks 값이 1이상이 아니라면 무시
- 시스템 콜 번호는 22번으로 구현할 것. (주의) 기존 설계과제 #1에서 진행한 내용은 삭제 후 진행. 모든 설계과제는 독립적임. 즉 다른 설계과제에서 수행한 내용은 없어야 함.

✓ (2) 프로세스의 stride 값을 갱신

- stride = STRIDE_MAX / tickets
- C언어 나누기 연산을 사용 (나머지 버림)

(예 4). syscall_test 실행 결과 (출력 형식 준수 필수, 알맞은 debug 구현 시, 실행하면 동일한 결과 출력)

```
$ syscall_test
Process 4 start
Process 4 selected, stride : 1000, ticket : 100, pass : 0 -> 1000  (1/6)
Process 4 selected, stride : 1000, ticket : 100, pass : 1000 -> 2000  (2/6)
Process 4 selected, stride : 1000, ticket : 100, pass : 2000 -> 3000  (3/6)
Process 4 selected, stride : 1000, ticket : 100, pass : 3000 -> 4000  (4/6)
Process 4 selected, stride : 1000, ticket : 100, pass : 4000 -> 5000  (5/6)
Process 4 selected, stride : 1000, ticket : 100, pass : 5000 -> 6000  (6/6)
Process 4 exit
```

- 3. Stride 기반 scheduler() 함수

✓ XV6 커널의 스케줄러 함수를 Stride 스케줄러로 수정. 스케줄러는 다음과 같은 기능을 구현해야 함

- 스케줄러 구현을 위해 scheduler 함수 이외의 다른 함수 및 파일에 접근해야 할 수 있음
- 프로세스 선택 알고리즘
  - ☞ RUNNABLE인 프로세스만 고려함
  - ☞ 프로세스 탐색시 프로세스 테이블 락을 얻은 상태로 탐색
    - 만약 실행 가능한 프로세스가 하나도 없다면 락을 해제하고 다시 프로세스 탐색 알고리즘 반복
    - RUNNABLE인 프로세스 중에서 pass 값이 가장 작은 프로세스를 선택해야 함
    - pass 값이 가장 작은 프로세스가 여러 개라면 그 중에서 pid가 가장 작은 프로세스를 선택
    - 선택된 프로세스에게 1틱만큼 cpu 제어권을 부여하고, 해당 프로세스가 cpu 점유를 마치면 다시 프로세스를 선택
  - ☞ pass 갱신
    - 각 프로세스가 1틱만큼 cpu 점유를 마치고 제어권을 넘길 때, 본인의 stride 값만큼 pass 값을 올림
    - stride 값은 setticket() 시스템 콜을 통해 설정되며, 티켓이 높으면 stride 값이 낮아짐
    - stride 값이 낮다는 말은 매 틱마다 pass 값이 stride 값이 더 많은 프로세스에 비해 조금만 증가하여, 실행될 확률이 더 높음
  - ☞ 오버플로우 방지
    - pass가 계속해서 증가한다면 오버플로우가 발생할 수 있음
    - 본 과제의 스케줄러는 오버플로우가 발생하는 상황을 대처하는 기능을 구현해야 함
    - 어떤 프로세스가 PASS_MAX 값을 넘길 경우 모든 RUNNABLE인 프로세스 중 가장 pass 값이 작은 프로세스를 0으로 만들고, pass를 감소시킨 만큼 (즉, 가장 pass 값이 작은 프로세스의 pass 값만큼) 모든 RUNNABLE 프로세스의 pass 값을 동일하게 감소시킴
    - 이 때, 어떤 프로세스의 감소 이후의 pass 값이 너무 크다면 오버플로우 방지 알고리즘이 너무 많이 실행 되어 오버헤드가 생기기 때문에, 모든 프로세스 간 pass 값의 차이가 DISTANCE_MAX 값을 넘기지 않아야 함
    - 즉, 가장 pass 값이 작은 프로세스의 pass 값보다 더 많이 pass 값이 감소 되는 프로세스가 존재할 수 있음
- 인자가 있는 실행 예 (학생 테스트용, 1~5 범위의 값, 예시는 1, 4 실행)

(예 5-1). scheduler_test 실행 결과 (출력 형식 준수 필수, 파란색은 이해를 위한 디버깅용 출력임)

```
$ scheduler_test 1
test1 start

Process 4 start
Process 5 start
… (중략) …
Process 6 selected, stride : 100, ticket : 1000, pass : 1900 -> 2000  (20/20)
Process 6 exit

test1 end

$ scheduler_test 4
test4 start

Process 8 start
Process 9 start
Process 10 start
Process 8 selected, stride : 9090, ticket : 11, pass : 0 -> 9090  (1/20)
Process 9 selected, stride : 4347, ticket : 23, pass : 0 -> 4347  (1/20)
Process 10 selected, stride : 3571, ticket : 28, pass : 0 -> 3571  (1/20)
… (중략) …
Rebase Process Start
Process 8's pass is standardize from 18180, to 0
Rebase Process End

Process 8 selected, stride : 9090, ticket : 11, pass : 0 -> 9090  (20/20)
Process 8 exit

test4 end
```

- 인자가 없는 실행 예, 테스트 1부터 5까지 모두 실행함 (전체 내용은 첨부 파일 참고)

(예 5-2). scheduler_test 실행 결과 (출력 형식 준수 필수, 파란색은 이해를 위한 디버깅용 출력임)

```
$ scheduler_test
test1 start

Process 4 start
Process 5 start
Process 6 start
Process 4 selected, stride : 100, ticket : 1000, pass : 0 -> 100  (1/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 0 -> 100  (1/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 0 -> 100  (1/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 100 -> 200  (2/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 100 -> 200  (2/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 100 -> 200  (2/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 200 -> 300  (3/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 200 -> 300  (3/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 200 -> 300  (3/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 300 -> 400  (4/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 300 -> 400  (4/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 300 -> 400  (4/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 400 -> 500  (5/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 400 -> 500  (5/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 400 -> 500  (5/20)
… (중략) …
Rebase Process Start
Process 19's pass is standardize from 15438, with distance cutting, to 7500
Process 22's pass is standardize from 7142, to 0
Rebase Process End

Process 22 selected, stride : 7142, ticket : 14, pass : 0 -> 7142  (9/10)
Process 22 selected, stride : 7142, ticket : 14, pass : 7142 -> 14284  (10/10)
Process 22 exit
Process 19 selected, stride : 11111, ticket : 9, pass : 7500 -> 18611  (10/10)

Rebase Process Start
Process 19's pass is standardize from 18611, to 0
Rebase Process End

Process 19 exit

test5 end
```

○ 과제 제출 마감
- 2025년 10월 19일(일) 23시 59분까지 구글클래스로 제출 (강의계획서 확인). 1초라도 늦으면 0점 처리
- 가산점 2 부여 마감 기한 : 2025년 9월 28(일)/10월 05일(일)/10월 12일(일) 각각 23시 59분까지 구글클래스로 제출 1초라도 늦으면 가산점2 없음
- 가산점 2 부여 점수. 9월 29일까지 해당 과제 부여 점수 * 2 추가 부여, 10월 05일까지 해당 과제 부여 점수 * 1.5 추가 부여, 10월 12일까지 해당 과제 부여 점수 * 1 추가 부여

○ 주의사항
- 보고서 (hwp, doc, docx 등으로 작성 - 수행된 결과 (캡쳐 등) 반드시 포함시킬 것
- 수정한 파일은 반드시 제출해야 함
- **설계과제 #2 소스코드 내 설계과제#1의 소스코드가 포함되지 않도록 해야 함. 설계과제 #1에서 수정하고 구현한 내용은 설계과제 #2와 전혀 상관이 없음. 예를 들어 시스템콜 번호도 xv6의 원래 소스코드에서 설계과제 #1에서 추가한 시스템콜 관련 내용 등이 전혀 없어야 함. 설계과제 #1의 내용을 포함시킬 경우 감점 -20점.**
- **제출 파일명 : #2_학번_분반.zip, zip 파일 내 #2_학번_분반.hwp (doc 가능) 과 소스코드 디렉토리가 있어야 함. 소스코드 디렉토리 내에는** xv6에서 변경한 소스코드, 추가한 소스코드및 테스트 쉘 프로그램 소스코드 (helloxv6.c, psinfo.c) 등

○ 필수 구현
- 1, 2, 3 (각 테스트별 점수에 따라 점수 부여, 테스트 내용은 변경될 수 있음)

○ 배점 기준
- 1. 기본 설정 및 구현 : 20점
- 2. settickets() 시스템콜 추가 및 구현 : 30점
- 3. stride 기반 scheduler() 함수 구현 : 50점