1. 개요

본 과제의 목표는 xv6 운영체제에 Stride Scheduling을 직접 구현하여 커널 수준에서 CPU 자원 분배 방식과 스케줄러의 동작 원리를 학습하는 데 있다. xv6의 기본 스케줄러는 라운드 로빈 방식으로 모든 프로세스에 동일한 시간을 부여하지만, 본 과제에서는 프로세스마다 서로 다른 티켓 수를 설정하고 이를 기반으로 stride 값을 계산하여 실행 순서를 결정하도록 변경하였다. 이러한 구현을 통해 단기적 변동성이 큰 라운드 로빈 스케줄링의 한계를 극복하고, 보다 예측 가능하며 안정적인 프로세스 실행을 달성할 수 있었다. 나아가 단순한 알고리즘 구현을 넘어, xv6의 전체 실행 과정과 운영체제 내부의 자원 관리 원리를 구체적으로 학습하였으며, 이를 통해 스케줄러 설계가 시스템의 공정성과 안정성을 확보하는 핵심 요소임을 체감할 수 있었다.

2. 상세설계

2-1. 기본설정

2-1-1. 개요

이 단계에서는 stride 스케줄링을 구현하기 위한 기반 환경을 마련한다. 구체적으로는 xv6 위에서 제공된 테스트 프로그램이 바로 실행될 수 있도록 빌드 구성을 준비하고, 스케줄러가 사용할 프로세스 확장 필드와 관련 상수를 정의하며, 프로세스 생성 시 적용될 초기값을 설정한다. 이러한 준비가 올바르게 이루어져야 이후 단계에서 추가되는 디버깅 코드, settickets 시스템콜, 최종 스케줄러 구현이 명세에서 제시한 출력 형식과 동작 의미대로 재현될 수 있다.

2-1-2. Makefile 수정

```
UPROGS=\
        _cat\
        _echo\
        _forktest\
        _grep\
        _init\
        _kill\
        _ln\
        _ls\
        _mkdir\
        _rm\
        _sh\
        _stressfs\
        _usertests\
        _wc\
        _zombie\
        _debug_test\
        _syscall_test\
        _scheduler_test\
```

명세에서 제공된 테스트 프로그램은 이미 컴파일된 이진 파일이므로, xv6 빌드 과정에서 별도의 컴파일 없이 실행 이미지에 포함되도록 Makefile을 수정하였다. 이때 프로그램들은 UPROGS 항목에 추가해야 하며, _debug_test, _syscall_test, _scheduler_test를 등록하여 루트 파일 시스템에 포함되도록 하였다.

2-1-3. 자료구조 및 상수 정의

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int tickets;                 // Number of tickets
  uint stride;                 // Stride value for stride scheduling
  uint pass;                   // Pass value for stride scheduling
  int ticks;                   // Number of ticks the process has run
  int end_ticks;               // Number of ticks the process has run in the end
};

// Stride scheduling constants which are fixed by spec
#define STRIDE_MAX 100000
#define PASS_MAX 15000
#define DISTANCE_MAX 7500
```

stride 스케줄링을 지원하기 위해 프로세스 구조체에 필드를 확장하고, pass 값 관리와 overflow 방지를 위해 필요한 상수를 정의한다.

추가한 변수

i) tickets : 프로세스가 가진 티켓 수를 저장한다. CPU 점유 비율을 결정하는 기준으로 사용한다.

ii) stride : tickets에 반비례하여 계산된다. 프로세스가 선택되는 간격으로 사용한다.

iii) pass : 프로세스가 CPU를 점유할 때마다 누적되는 값이다. stride 스케줄러는 가장 작은 pass 값 가진 프로세스를 선택한다.

iv) ticks : 프로세스가 실제로 점유한 틱 수이다.

v) end_ticks : 프로세스의 실행 수명이다. 설정된 틱 수에 도달하면 자동 종료된다.

추가한 상수

i) STRIDE_MAX : stride 계산 시 분모로 사용된다. 티켓 수에 따른 비례 분배를 위해 큰 값으로 설정한다.

ii) PASS_MAX : pass 값의 overflow를 방지한다. rebase 수행 기준으로 사용된다.

iii) DISTANCE_MAX : rebase 시 프로세스간 pass 값 차이가 과도하게 커지는 것을 방지하여 안정성을 보장한다.

이와 같은 구조체 확장과 상수 정의를 통해 stride 스케줄러의 핵심 동작을 가능하게 하며, settickets()와 scheduler() 구현의 기반이 된다.

2-1-4. 초기화 정책 수립

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  // Initialize for stride scheduling
  p->tickets = 1;
  p->stride = 0;
  p->pass = 0;
  p->ticks = 0;
  p->end_ticks = -1;

  release(&ptable.lock);
```

프로세스 생성 시 일관된 초기 상태를 보장하기 위해 allocproc()에서 stride 관련 필드를 초기화하였다. 이는 모든 프로세스 생성 경로가 내부적으로 allocproc()을 호출하여 새로운 proc 구조체를 할당하기 때문이다. 구체적으로 tickets = 1로 설정하여 모든 프로세스가 기본적으로 최소한의 실행 비율을 가지도록 하였고, stride = 0과 pass = 0으로 두어 settickets()가 호출되기 전에는 스케줄러가 기본값 상태에서 동작하도록 하였다. 또한 ticks = 0으로 초기화하여 생성 직후부터 누적 실행 시간을 정확히 기록할 수 있도록 하였으며, end_ticks = -1로 두어 기본적으로는 수명 제한이 없는 상태에서 시작하게 하였다. 이러한 초기화 정책은 이후 단계에서 디버깅 코드, settickets 시스템콜, stride 기반 스케줄러가 정상적으로 동작하고 명세와 동일한 형식의 결과를 출력하는 기반이 된다.

2-1-5. 타이머 주기 조정

```
  // The timer repeatedly counts down at bus frequency
  // from lapic[TICR] and then issues an interrupt.
  // If xv6 cared more about precise timekeeping,
  // TICR would be calibrated using an external time source.
  lapicw(TDCR, 0x2);          // divide by 8
  lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
  lapicw(TICR, 20000000);   // initial count
```

간헐적으로 프로세스의 stride와 ticket 값이 초기화되기 전에 selected 로그가 먼저 출력되는 현상이 관찰될 수 있다. 이는 lapic 타이머의 인터럽트 주기가 상대적으로 짧아, 프로세스 초기화 직후 첫 타이머 인터럽트가 예상보다 빠르게 발생하면서 발생하는 동기화 문제로 판단된다.

이 문제를 완화하기 위해 타이머 설정을 조정하였다. TDCR 값을 X1에서 0x2으로 변경하여 분주비를 늘리고, TICR 초기 카운트를 20000000으로 확대하여 인터럽트 발생 간격을 길게 하였다. 이로 인해 하드웨어 타이머 인터럽트가 벽시계 시간 기준으로 더 느리게 발생하게 되었으며, 결과적으로 프로세스 생성 직후 settickets() 설정이 안정적으로 반영된 뒤 첫 selected 로그가 출력되도록 개선되었다.

2-2. 디버깅 코드 구현

2-2-1. 개요

_debug_test의 출력 형식이 정확히 재현되도록 프로세스의 생성,실행,종료 지점마다 로그를 남기는 기능을 구현한다. 이는 출력 코드가 올바른 위치에 삽입되었는지 확인하는 사전 단계로서, 이후 단계의 시스템콜 및 스케줄러 검증이 가능해지도록 기반을 마련한다.

2-2-2. 함수 설계

i) fork() 디버그 로그

```
acquire(&ptable.lock);

np->state = RUNNABLE;

// Debug log for scheduler test at the start of process
if(np->pid > 2 && curproc && curproc->pid > 2)
  cprintf("Process %d start\n", np->pid);

release(&ptable.lock);

return pid;
```

자식 프로세스가 처음으로 생성되어 PID가 부여되고 RUNNABLE 상태로 전환되는 순간을 명확히 기록하기 위하여, fork() 말단에 디버그 코드를 삽입하였다. 모든 사용자 프로세스의 생성경로가 내부적으로 allocproc()을 거쳐 fork()를 통과하므로, 이 지점에서 출력하면 생성 시점을 일관되게 기록할 수 있다. pid 조건을 통해 init과 sh같은 시스템 프로세스는 제외하였으며, 출력 값에 자식의 PID를 통해 어떤 프로세스가 시작했는지 명확하게 보여준다. 또한, trap()과 exit()로그와 이어보면 생성 -> 실행 -> 종료의 연속성이 맞는지 검증할 수 있다.

ii) trap() 디버그 로그

```
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER){
  struct proc *p = myproc();

  // update ticks for the process
  p->ticks++;

  // Debug log for scheduler test at each scheduling
  if(p->pid > 2 && p->parent && p->parent->pid > 2)
    cprintf("Process %d selected, stride : %d, ticket : %d, pass : %d -> %d  (%d/%d)\n",
            p->pid, p->stride, p->tickets, p->pass, p->pass+p->stride, p->ticks, p->end_ticks);

  // update pass for the process
  p->pass += p->stride;

  // If end_ticks is set, check it and exit if the process has run enough
  if(p->end_ticks > 0 && p->ticks >= p->end_ticks) exit();

  // yield the CPU
  yield();
}
```

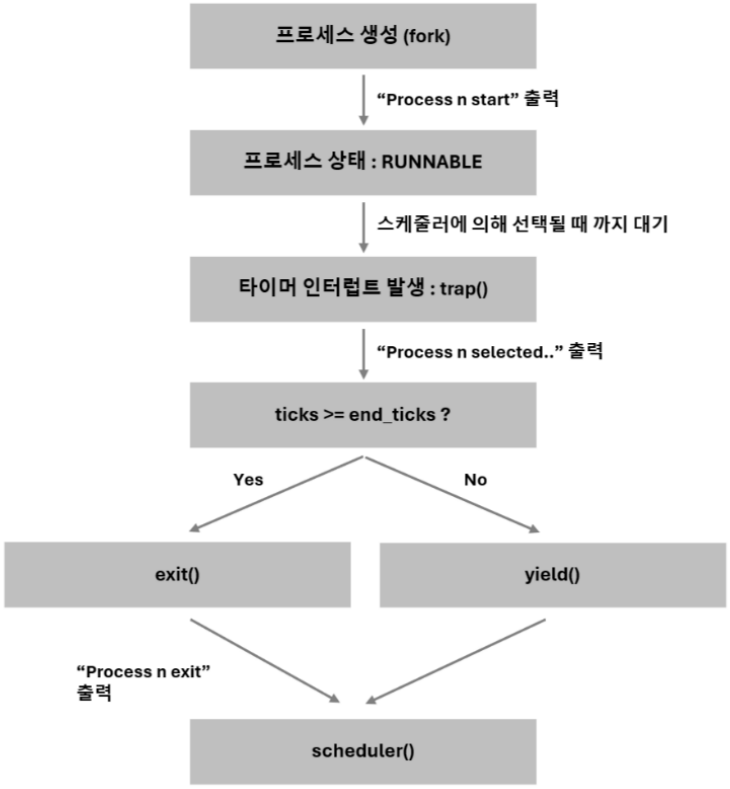
프로세스 실행 상황을 틱 단위로 추적하기 위해, 타이머 인터럽트 처리 구간(trap()의 T_IRQ0+IRQ_TIMER)에 디버그 코드를 삽입하였다. 출력 시, PID, stride, ticket, 갱신 전 pass, 갱신 후 pass, 현재 누적 틱, 종료 예정 틱을 출력한다. 이 출력은 매 tick마다 p->ticks++, p->pass += p->stride, end_ticks 도달 검사, yield() 호출이 연속적으로 수행되는 맥락 속에서 찍히므로, pass 값의 변화, end_ticks에 따른 수명 제한 여부, 틱 단위 선점 발생을 한눈에 확인할 수 있다. 또한 ticket이 클수록 stride가 작아져 자주 선택되는 비례 분배 현상, PASS_MAX 초과 시 발생하는 rebase, 특정 시점에서의 종료 판정까지 모두 이 로그로 관찰 가능하다. 출력은 역시 조건문을 통해 디버깅 대상 프로세스에 대해서만 수행된다.

iii) exit 디버그 로그

```
// Debug log for scheduler test at the end of process
if(curproc->pid > 2 && curproc->parent && curproc->parent->pid > 2)
  cprintf("Process %d exit\n", curproc->pid);
```

프로세스가 사용자 관점에서 실제로 종료되는 순간을 기록하기 위해, exit()에서 상태가 ZOMBIE로 전환되기 직전에 로그를 출력하였다. 이를 통해 end_ticks 기반의 수명 제한이 올바르게 적용되었는지, 또는 일반적인 코드 종료가 정상적으로 마무리되었는지를 확인할 수 있다. 출력 조건은 동일하게 적용되어 필요하지 않은 시스템 프로세스 로그는 남지 않는다. 이 로그는 종료 시점의 정확성을 보장하며, 시작과 실행 로그와 함께 전체 프로세스 흐름을 검증하는 역할을 한다.

2-2-3. 동작 흐름



프로세스의 생애 주기에서 디버깅 로그는 생성, 실행, 종료라는 세 단계에 맞추어 출력된다. 먼저 fork()가 호출되면 내부적으로 allocproc()을 통해 새로운 proc 구조체가 생성되고 PID가 부여된다. 이 시점에서 삽입한 Process %d start 로그가 출력되며, 프로세스가 준비 상태(RUNNABLE)로 진입했음을 확인할 수 있다. 이는 곧 생성 상태에서 준비(ready) 상태로 전환되는 과정을 기록하는 것이다.

프로세스가 실제로 CPU를 할당받아 실행되는 동안에는 하드웨어 타이머가 주기적으로 인터럽트를 발생시킨다. 인터럽트가 발생하면 trap()이 호출되고, 현재 실행 중인 프로세스의 실행 시간(ticks)이 갱신된다. 이어서 “Process %d selected, ...” 로그가 출력되며, 이 시점에서 프로세스의 pass 값이 stride에 따라 증가한다. 이후 trap() 내부에서는 프로세스의 누적 실행 시간이 종료 조건(end_ticks)에 도달했는지를 검사하고, 그 결과에 따라 두 가지 동작으로 분기한다. 만약 실행 시간이 제한값에 도달하면 exit()가 호출되어 프로세스의 상태가 ZOMBIE로 전환되기 직전에 Process %d exit 로그가 출력된다. 반대로 조건이 만족되지 않는 경우에는 yield()가 호출되어 현재 프로세스가 CPU를 양보하고 스케줄러가 다시 실행된다.

이와 같은 과정을 통해 프로세스는 매 틱마다 선점형 스케줄링의 영향을 받으며, 스케줄러는 주기적으로 개입하여 실행 대상을 교체한다. 결국 프로세스가 정상적으로 종료될 때에는 exit()가 호출되면서 자원이 회수되기 전 Process %d exit 로그가 출력되고, 이후 부모 프로세스가 wait()을 호출해야 자원이 정리된다.2-3. settickets 시스템 콜 구현

2-3-1. 개요

이 단계에서는 stride 스케줄러가 프로세스별로 서로 다른 실행 비율을 가질 수 있도록 settickets() 시스템 콜을 구현하였다. xv6의 기본 구조에서는 모든 프로세스가 동일한 실행 기회를 가지지만, 본 과제에서는 각 프로세스에 할당되는 티켓 수를 조정하여 CPU 점유율을 비례적으로 분배할 수 있도록 하였다. settickets()는 사용자 수준에서 프로세스가 원하는 티켓 수와 실행 수명(end_ticks)을 지정할 수 있게 하며, 커널 내부에서는 이를 기반으로 stride와 pass를 재계산하여 스케줄러가 올바른 정책을 적용할 수 있도록 한다.

2-3-2. 함수 설계

다음 과정들을 통해 사용자 수준에서 호출한 settickets()가 커널 내부로 전달되어 현재 프로세스의 티켓 수와 stride, end_ticks 등이 일관되게 반영되도록 시스템콜 경로 전체를 연결한다. 유저공간 - 시스템콜 진입 - 디스패치 - 커널함수 구현 - 프로세스 필드 갱신 순으로 진행되며, 아래 파일들을 수정하여 완성한다.

i) user.h : 사용자 프로토타입 선언

```
// New system calls for project 2
int settickets(int tickets, int end_ticks); // set number of tickets and end_ticks for the process
```

사용자 프로그램이 호출할 수 있도록 settickets() 프로토타입을 선언한다. 인자로 tickets와 end_ticks를 받는 것을 알 수 있다. 이 선언으로 유저 코드에서 컴파일 시점 인터페이스가 보장된다.

ii) usys.S : 시스템 콜 스텁 생성

```
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(settickets)
```

usys.S에는 SYSCALL(settickets)를 추가하여 사용자 수준 함수 호출을 커널의 시스템 콜로 연결하였다. 이 스텁은 호출 시 시스템 콜 번호 22를 EAX 레지스터에 적재하고 소프트웨어 인터럽트 명령(int $T_SYSCALL)을 실행하여 커널 모드로 진입한다. 이 과정을 통해 사용자 프로그램에서 호출한 settickets()는 커널 내부의 sys_settickets()와 직접 연결되며, xv6의 다른 시스템 콜들과 동일한 경로를 따라 일관되게 처리된다.

iii) syscall.h : 시스템 콜 번호 할당

```
#define SYS_close  21
#define SYS_settickets 22 // New system call number for settickets
```

syscall.h에는 settickets에 고유 번호 22를 부여하였다. xv6에서는 모든 시스템 콜을 정수 번호로 구분하고, 이 번호가 커널 진입 시 EAX 레지스터에 실려 전달된다. 따라서 고유 번호를 지정해야만 syscall.c의 디스패치 테이블이 올바른 함수를 찾아갈 수 있으며, 기존 시스템 콜과 충돌 없이 일관된 경로로 처리된다. 번호 22는 앞선 시스템 콜들의 순서를 이어가면서도 중복되지 않도록 배정하였으며, 이로써 settickets()가 xv6 내부에서 정식 시스템 콜로 인식될 수 있도록 보장한다.

iv) syscall.c : 외부 선언 및 디스패치 테이블 매핑

```
extern int sys_uptime(void);
extern int sys_settickets(void); // New system call for settickets

[SYS_close]   sys_close,
[SYS_settickets] sys_settickets, // New system call for settickets
```

syscall.c에서는 커널 진입 후 올바른 핸들러로 분기되도록 extern 선언과 시스템 콜 테이블 매핑을 추가하였다. 먼저 extern 선언을 통해 sysproc.c에 구현된 커널 함수가 이곳에서 참조 가능하게 하였으며, 이후 syscalls[] 배열의 인덱스 22번에 sys_settickets를 등록하였다. xv6의 syscall() 함수는 트랩프레임의 EAX 레지스터에서 시스템 콜 번호를 읽어 해당 인덱스의 함수 포인터를 호출하는 방식으로 동작한다. 따라서 사용자 공간에서 settickets()가 호출되면, usys.S 스텁과 syscall.h 번호 정의를 거쳐 결국 이 테이블을 통해 sys_settickets()로 이어지게 된다.

v) proc.h : stride 관련 상수,필드

```
  char name[16];               // Process name (debugging)
  int tickets;                 // Number of tickets
  uint stride;                 // Stride value for stride scheduling
  uint pass;                   // Pass value for stride scheduling
  int ticks;                   // Number of ticks the process has run
  int end_ticks;               // Number of ticks the process has run in the end
};

// Stride scheduling constants which are fixed by spec
#define STRIDE_MAX 100000
#define PASS_MAX 15000
#define DISTANCE_MAX 7500
```

2-1 단계에서 확장해둔 필드와 상수들이 그대로 활용되었다. settickets()는 이 정의들을 기반으로 프로세스의 티켓 수와 stride 값을 갱신하고, 이후 trap()과 scheduler()가 해당 값을 참조하여 스케줄링을 수행한다.

vi) proc.c : 초기화 정책

```
// Initialize for stride scheduling
p->tickets = 1;
p->stride = 0;
p->pass = 0;
p->ticks = 0;
p->end_ticks = -1;
```

allocproc()에서는 2-1 단계에서 stride 스케줄링을 위한 초기화 정책을 반영하였다. 프로세스가 생성될 때 일관된 기본값을 갖도록 설정해 두었기 때문에, 이후 settickets()가 호출되면 이 값들이 사용자 지정 값으로 정확히 갱신된다. 즉, 초기화 정책을 통해 settickets()가 의도한 대로 동작할 수 있게 된다.

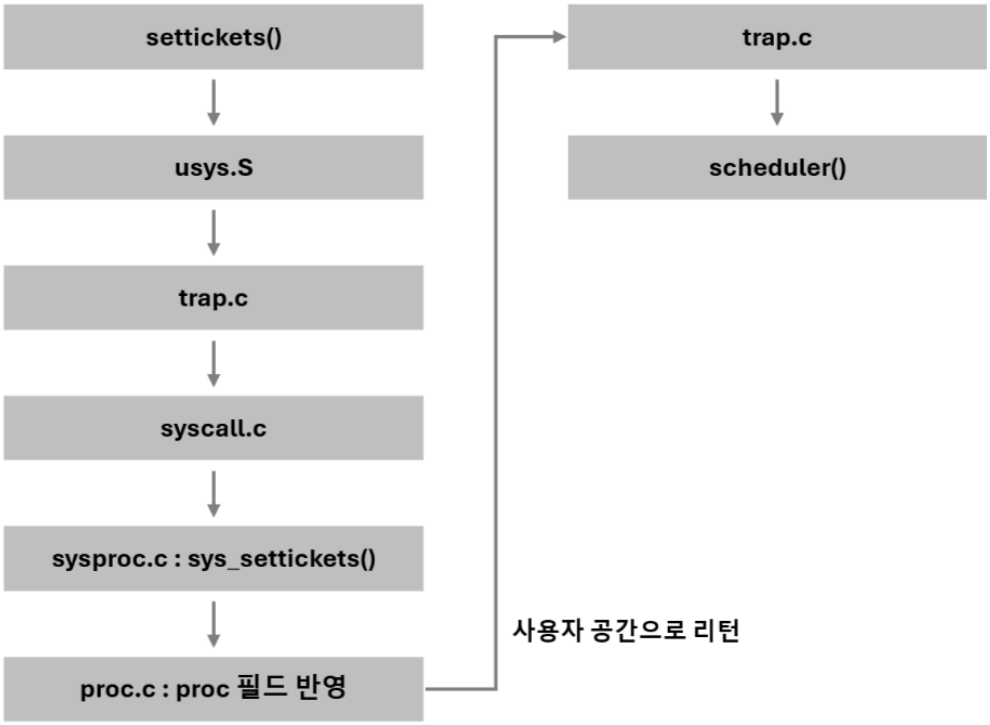
vii) sysproc.c : sys_settickets() 구현

```
// New system call for project 2
int
sys_settickets(void)
{
  int tickets, end_ticks;
  struct proc *p = myproc();

  // Invalid arguments
  if(argint(0, &tickets) < 0 || argint(1, &end_ticks) < 0) return -1;
  // Invalid num of tickets
  if(tickets < 1 || tickets > (STRIDE_MAX -1)) return -1;

  // Set the number of tickets and end_ticks for the process
  p->tickets = tickets;
  p->stride = STRIDE_MAX / tickets;

  // Set end_ticks only if it is valid
  if(end_ticks >= 1) p->end_ticks = end_ticks;

  return 0;
}
```

sysproc.c에서는 시스템 콜의 실제 동작을 수행하는 sys_settickets()를 구현하였다. 먼저 argint()를 통해 사용자 공간에서 전달된 인자를 안전하게 읽어오며, 인자가 잘못 전달된 경우 -1을 반환한다. 이어서 tickets 값이 1보다 작거나 STRIDE_MAX-1보다 큰 경우를 오류로 처리하여, 잘못된 티켓 수로 인한 계산 오류와 오버플로우를 방지하였다. 유효한 경우에는 현재 프로세스의 tickets를 갱신하고, stride를 STRIDE_MAX / tickets로 재계산하여 CPU 점유 비율이 티켓 수에 비례하도록 하였다. 또한 end_ticks가 1 이상일 때만 프로세스의 실행 수명을 설정하고, 그렇지 않은 경우 기본값(-1)을 유지하여 무제한 실행으로 처리하였다.

이 함수는 성공 시 0을, 실패 시 -1을 반환하며, 반환값은 다시 사용자 프로그램으로 전달된다. 결과적으로 sys_settickets()는 사용자 요청을 커널 내부의 프로세스 구조체에 반영하여, 이후 trap()과 scheduler()에서 실행 빈도와 종료 시점을 결정할 수 있도록 한다.

2-3-3. 동작 흐름



사용자 프로그램이 settickets(tickets, end_ticks)를 호출하면, usys.S에 정의된 시스템 콜 스텁이 실행되어 시스템 콜 번호 22가 EAX 레지스터에 적재되고 int $T_SYSCALL을 통해 커널 모드로 진입한다. 이후 커널의 trap()에서 syscall() 함수가 호출되며, syscalls[] 테이블의 22번 엔트리를 확인하여 sysproc.c의 sys_settickets()가 실행된다.

sys_settickets() 내부에서는 argint()를 통해 사용자로부터 전달된 인자를 검증하고, 유효하지 않은 경우에는 -1을 반환한다. 인자가 올바르다면 현재 프로세스의 proc 구조체에 정의된 tickets, stride, end_ticks 값이 갱신된다. 이때 stride는 STRIDE_MAX / tickets로 계산되며, 이는 해당 프로세스가 스케줄러에서 선택되는 빈도에 직접적으로 작용한다. 함수 실행이 완료되면 시스템 콜은 사용자 공간으로 반환되어, 호출한 프로그램은 0 또는 -1을 결과로 받는다.

이후의 동작은 2-2-3에서 설명했던 것과 동일하게, 매 타이머 틱마다 trap()에서 pass가 stride만큼 증가하고 end_ticks 도달 여부에 따라 exit() 또는 yield()가 호출되며, 최종적으로 sched()를 거쳐 scheduler()로 복귀한다. 따라서 settickets()는 주기적인 타이머 인터럽트와 스케줄러 선택 과정 전체에 영향을 미친다. 사용자가 지정한 실행 비율과 실행 수명이 스케줄링 정책에 반영되며, 다수의 프로세스가 동시에 실행될 경우 CPU 점유율을 비례적으로 조정하는 핵심 수단으로 기능한다. 결과적으로 stride 스케줄러는 공정성과 예측 가능성을 보장하면서 프로세스 실행을 제어하게 된다.

2-4. stride 기반 scheduler() 함수

2-4-1. 개요

xv6의 scheduler()가 stride 기반 선택 규칙을 따르도록 수정하였다. 스케줄러는 매 라운드마다 RUNNABLE 상태의 프로세스 중 가장 작은 pass 값을 가진 프로세스를 선택하고, 동률일 경우 PID가 작은 프로세스를 우선한다. pass 증가는 스케줄러가 아니라 타이머 인터럽트 경로에서 'pass += stride'로 누적되며, scheduler()는 이 값을 읽어 선택만 수행한다. 또한 PASS_MAX 초과 시 최소 pass를 기준으로 모든 프로세스의 pass를 재정렬하는 rebase를 적용하여 오버플로우와 왜곡을 방지하고, DISTANCE_MAX로 rebase 후 격차를 제한한다. 이러한 구성은 tickets에 반비례하는 stride를 통해 CPU 점유율이 티켓 수에 비례하도록 보장한다.

2-4-2. 함수 설계

i) rebase_pass() : rebase 루틴

```
static void
rebase_pass(void){
  struct proc *p;
  int flag = 0;     // Flag to check if any process has to rebase

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == RUNNABLE && p->pass > PASS_MAX){
      flag = 1;     // set flag to rebase
      break;
    }
  }
  if(!flag) return;   // No process has to rebase

  uint mn = (uint)-1;   // Minimum pass value among
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    // Find minimum pass value
    if(p->state == RUNNABLE && p->pass < mn) mn = p->pass;
  }
  if(mn == (uint)-1) return;   // No RUNNABLE process

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    // Only consider RUNNABLE processes
    if(p->state != RUNNABLE) continue;

    // Rebase pass values
    if(p->pass <= mn) p->pass = 0;
    else if(p->pass > mn){
      p->pass -= mn;
      if(p->pass > DISTANCE_MAX) p->pass = DISTANCE_MAX;
    }
  }
}
```

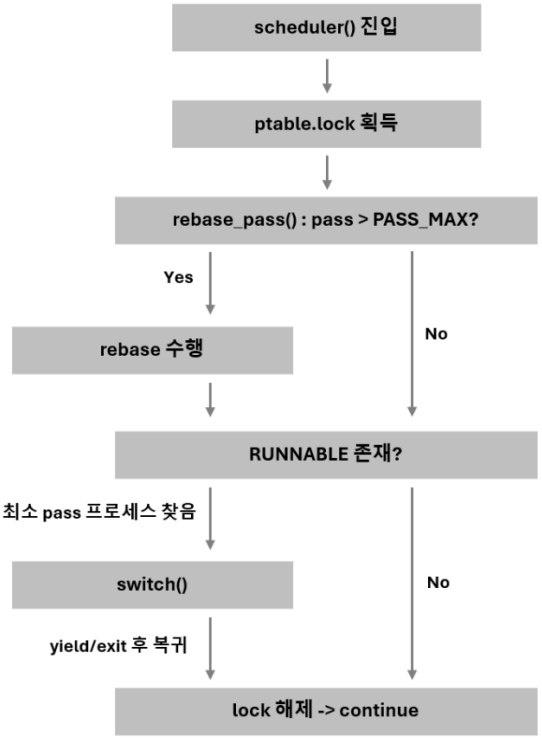
rebase_pass()는 PASS_MAX를 초과한 pass가 하나라도 존재할 때 모든 RUNNABLE 프로세스의 pass를 재정렬하도록 설계한다. 먼저 프로세스 테이블을 순회하여 RUNNABLE && pass > PASS_MAX가 있는지를 확인하고, 없으면 즉시 반환한다. rebase가 필요하면RUNANNABLE 중 최소 pass 값인 mn을 구한 뒤, 각 RUNNABLE 프로세스에 대해 pass = max(0, pass - mn)으로 rebase 연산을 수행한다. 이때 재정렬 후의 pass가 과도하게 벌어지는 것을 방지하기 위해 DISTANCE_MAX를 상한으로 적용하여 안정성을 확보한다. 최소값 기준 rebase를 통해, pass 오버플로우와 왜곡을 방지하고, 이후 scheduler()에서 "작은 pass 우선, 동률 시 PID 우선"이라는 stride 선택 규칙이 일관되게 유지되도록 한다.

ii) scheduler() : 기존 라운드로빈 방식을 stride 방식으로 교체

```
void
scheduler(void)
{
  struct proc *p, *temp;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);

    // Rebase pass values if necessary
    rebase_pass();

    // Find process with minimum pass value
    temp = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      // Only consider RUNNABLE processes
      if(p->state != RUNNABLE) continue;

      // Select process with minimum pass value
      if(temp == 0 || p->pass < temp->pass || (p->pass == temp->pass && p->pid < temp->pid))
        temp = p;
    }

    // No RUNNABLE process
    if(temp == 0){
      release(&ptable.lock);
      continue;
    }

    // Switch to chosen process.
    c->proc = temp;
    switchuvm(temp);
    temp->state = RUNNING;
    swtch(&c->scheduler, temp->context);
    switchkvm();

    // Process is done running
    c->proc = 0;

    // drop ptable.lock before next round
    release(&ptable.lock);
  }
}
```

기존에는 모든 RUNNABLE 프로세스를 라운드 로빈 방식으로 동일하게 순환 실행했지만, stride 기반으로 수정하여 매 라운드마다 RUNNABLE 집합 중 가장 작은 pass 값을 가진 프로세스를 선택하도록 한다. 동률이 발생하면 PID가 작은 프로세스를 우선한다는 조건도 만족한다. 선택된 프로세스는 c->proc으로 등록된 뒤 switchuvm()과 swtch()를 거쳐 실행되며, 실행이 끝나 복귀하면 switchkvm()과 c->proc = 0으로 정리된다. 이러한 변경을 통해 tickets와 stride 값이 실제 CPU 점유율에 반영되고, 프로세스 간 실행 빈도가 티켓 수에 비례하여 유지된다.

2-4-3. 동작 흐름



scheduler()는 xv6에서 각 CPU가 부팅된 직후 호출되며, 이후 절대 반환하지 않고 무한히 실행되는 루틴이다. 이 함수는 항상 RUNNABLE 상태의 프로세스를 탐색하여 실행 대상을 선택하고, 프로세스가 종료되거나 타이머 인터럽트에 의해 선점될 때마다 다시 제어가 돌아와 새로운 대상을 선택한다.

루프가 시작되면 먼저 ptable.lock을 획득하여 프로세스 테이블에 대한 동시 접근을 제어한다. 이어서 rebase_pass()를 호출하여 RUNNABLE 집합의 pass 값이 PASS_MAX를 초과한 경우 모든 프로세스의 pass를 최소값 기준으로 보정한다. 만약 실행 가능한 프로세스가 하나도 없다면, lock을 해제하고 다음 반복으로 넘어가 유휴 상태를 유지한다. 반대로 RUNNABLE 상태의 프로세스가 존재한다면, 그 중 가장 작은 pass 값을 가진 프로세스를 선택하고, 값이 동일한 경우에는 PID가 가장 작은 프로세스를 고른다. 선택된 프로세스는 컨텍스트 전환을 통해 실행되며, 이 과정에서 주소 공간 전환, 레지스터 교체, 상태 변경이 차례대로 이루어진다. 실행된 프로세스는 타이머 인터럽트, yield(), exit() 등을 통해 CPU를 반납하면 sched()를 거쳐 다시 스케줄러로 복귀하고, 스케줄러는 lock을 해제한 뒤 루프를 반복한다. 이러한 과정을 통해 tickets와 stride에 의해 결정된 실행 비율이 실제 CPU 점유율로 반영되며, 여러 프로세스가 동시에 존재하더라도 공정성과 비례성이 유지된다.

3. 결과

3-1. debug_test

```
$ 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)
Process 4 selected, stride : 0, ticket : 1, pass : 0 -> 0  (5/-1)
Process 4 selected, stride : 0, ticket : 1, pass : 0 -> 0  (6/-1)
Process 4 selected, stride : 0, ticket : 1, pass : 0 -> 0  (7/-1)
Process 4 selected, stride : 0, ticket : 1, pass : 0 -> 0  (8/-1)
Process 4 selected, stride : 0, ticket : 1, pass : 0 -> 0  (9/-1)
Process 4 selected, stride : 0, ticket : 1, pass : 0 -> 0  (10/-1)
result : 1540746712
Process 4 exit
```

위 결과는 디버그 출력이 올바르게 삽입되었는지를 확인하기 위한 테스트의 중간 생략 결과이다. 프로세스의 시작 시점, 매 타이머 틱, 그리고 종료 직전에 추가한 디버그 로그가 의도한 대로 출력되었음을 볼 수 있었다. 특히 "Process n selected" 로그는 매 틱마다 동일한 형식으로 반복되었는데, 값이 모두 같은 이유는 단일 프로세스 환경에서 settickets() 함수 구현 전이기 때문에 기본값인 'stride : 0, ticket : 1, pass : 0' 상태가 유지되었다. 따라서 매번 'pass : 0 -> 0' 으로 출력되며, 종료 tick도 설정되지 않아 -1로 고정되었다. 마지막으로 exit 직전에 debug_test에서 계산된 반복 연산 결과값으로 result가 출력되었는데, 이는 프로세스가 정상적으로 연산을 수행한 뒤 종료되었음을 의미한다.

3-2. syscall_test

```
$ 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
```

위 결과는 settickets() 시스템 콜이 정상적으로 동작하는지 확인하기 위한 테스트의 결과이다. 프로세스가 시작하며 "Process 4 start" 로그가 출력되었고, 종료 직전에는 "Process 4 exit" 로그가 나타났다. 실행 중 매 타이머 틱마다 'Process 4 selected ..'가 기록되었는데, stride : 1000, ticket=100 이 유지되며 pass 값이 매번 1000씩 증가하였다. 이는 시스템콜이 stride = STRIDE_MAX / tickets => 100000 / 100 => 1000을 정확히 계산하여 프로세스에 반영했다는 것을 보여준다. 또한 로그의 (n/6)을 보면, 누적 tick이 6에 도달하자 프로세스가 즉시 종료된 것을 통해 settickets()의 두번째 인자였던 end_ticks = 6이 올바르게 동작했음을 알 수 있었다.

3-3. scheduler_test

3-3-1. test1

```
$ 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)
Process 4 selected, stride : 100, ticket : 1000, pass : 500 -> 600  (6/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 500 -> 600  (6/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 500 -> 600  (6/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 600 -> 700  (7/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 600 -> 700  (7/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 600 -> 700  (7/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 700 -> 800  (8/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 700 -> 800  (8/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 700 -> 800  (8/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 800 -> 900  (9/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 800 -> 900  (9/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 800 -> 900  (9/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 900 -> 1000  (10/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 900 -> 1000  (10/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 900 -> 1000  (10/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 1000 -> 1100  (11/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 1000 -> 1100  (11/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 1000 -> 1100  (11/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 1100 -> 1200  (12/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 1100 -> 1200  (12/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 1100 -> 1200  (12/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 1200 -> 1300  (13/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 1200 -> 1300  (13/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 1200 -> 1300  (13/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 1300 -> 1400  (14/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 1300 -> 1400  (14/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 1300 -> 1400  (14/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 1400 -> 1500  (15/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 1400 -> 1500  (15/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 1400 -> 1500  (15/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 1500 -> 1600  (16/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 1500 -> 1600  (16/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 1500 -> 1600  (16/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 1600 -> 1700  (17/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 1600 -> 1700  (17/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 1600 -> 1700  (17/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 1700 -> 1800  (18/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 1700 -> 1800  (18/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 1700 -> 1800  (18/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 1800 -> 1900  (19/20)
Process 5 selected, stride : 100, ticket : 1000, pass : 1800 -> 1900  (19/20)
Process 6 selected, stride : 100, ticket : 1000, pass : 1800 -> 1900  (19/20)
Process 4 selected, stride : 100, ticket : 1000, pass : 1900 -> 2000  (20/20)
Process 4 exit
Process 5 selected, stride : 100, ticket : 1000, pass : 1900 -> 2000  (20/20)
Process 5 exit
Process 6 selected, stride : 100, ticket : 1000, pass : 1900 -> 2000  (20/20)
Process 6 exit

test1 end
```

세 개의 프로세스(4,5,6)이 모두 ticket=1000으로 동일하게 시작하였다. 실행결과를 보면 stride=100, pass 값이 각 프로세스마다 동일한 간격으로 증가하면서 라운드로빈처럼 번갈아 선택되었다. 이 결과로 티켓이 동일한 때 CPU가 균등하게 분배됨을 확인할 수 있었다.

3-3-2. test2

```
test2 start

Process 7 start
Process 8 start
Process 9 start
Process 7 selected, stride : 1000, ticket : 100, pass : 0 -> 1000  (1/20)
Process 8 selected, stride : 500, ticket : 200, pass : 0 -> 500  (1/20)
Process 9 selected, stride : 333, ticket : 300, pass : 0 -> 333  (1/20)
Process 9 selected, stride : 333, ticket : 300, pass : 333 -> 666  (2/20)
Process 8 selected, stride : 500, ticket : 200, pass : 500 -> 1000  (2/20)
Process 9 selected, stride : 333, ticket : 300, pass : 666 -> 999  (3/20)
Process 9 selected, stride : 333, ticket : 300, pass : 999 -> 1332  (4/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 1000 -> 2000  (2/20)
Process 8 selected, stride : 500, ticket : 200, pass : 1000 -> 1500  (3/20)
Process 9 selected, stride : 333, ticket : 300, pass : 1332 -> 1665  (5/20)
Process 8 selected, stride : 500, ticket : 200, pass : 1500 -> 2000  (4/20)
Process 9 selected, stride : 333, ticket : 300, pass : 1665 -> 1998  (6/20)
Process 9 selected, stride : 333, ticket : 300, pass : 1998 -> 2331  (7/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 2000 -> 3000  (3/20)
Process 8 selected, stride : 500, ticket : 200, pass : 2000 -> 2500  (5/20)
Process 9 selected, stride : 333, ticket : 300, pass : 2331 -> 2664  (8/20)
Process 8 selected, stride : 500, ticket : 200, pass : 2500 -> 3000  (6/20)
Process 9 selected, stride : 333, ticket : 300, pass : 2664 -> 2997  (9/20)
Process 9 selected, stride : 333, ticket : 300, pass : 2997 -> 3330  (10/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 3000 -> 4000  (4/20)
Process 8 selected, stride : 500, ticket : 200, pass : 3000 -> 3500  (7/20)
Process 9 selected, stride : 333, ticket : 300, pass : 3330 -> 3663  (11/20)
Process 8 selected, stride : 500, ticket : 200, pass : 3500 -> 4000  (8/20)
Process 9 selected, stride : 333, ticket : 300, pass : 3663 -> 3996  (12/20)
Process 9 selected, stride : 333, ticket : 300, pass : 3996 -> 4329  (13/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 4000 -> 5000  (5/20)
Process 8 selected, stride : 500, ticket : 200, pass : 4000 -> 4500  (9/20)
Process 9 selected, stride : 333, ticket : 300, pass : 4329 -> 4662  (14/20)
Process 8 selected, stride : 500, ticket : 200, pass : 4500 -> 5000  (10/20)
Process 9 selected, stride : 333, ticket : 300, pass : 4662 -> 4995  (15/20)
Process 9 selected, stride : 333, ticket : 300, pass : 4995 -> 5328  (16/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 5000 -> 6000  (6/20)
Process 8 selected, stride : 500, ticket : 200, pass : 5000 -> 5500  (11/20)
Process 9 selected, stride : 333, ticket : 300, pass : 5328 -> 5661  (17/20)
Process 8 selected, stride : 500, ticket : 200, pass : 5500 -> 6000  (12/20)
Process 9 selected, stride : 333, ticket : 300, pass : 5661 -> 5994  (18/20)
Process 9 selected, stride : 333, ticket : 300, pass : 5994 -> 6327  (19/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 6000 -> 7000  (7/20)
Process 8 selected, stride : 500, ticket : 200, pass : 6000 -> 6500  (13/20)
Process 9 selected, stride : 333, ticket : 300, pass : 6327 -> 6660  (20/20)
Process 9 exit
Process 8 selected, stride : 500, ticket : 200, pass : 6500 -> 7000  (14/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 7000 -> 8000  (8/20)
Process 8 selected, stride : 500, ticket : 200, pass : 7000 -> 7500  (15/20)
Process 8 selected, stride : 500, ticket : 200, pass : 7500 -> 8000  (16/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 8000 -> 9000  (9/20)
Process 8 selected, stride : 500, ticket : 200, pass : 8000 -> 8500  (17/20)
Process 8 selected, stride : 500, ticket : 200, pass : 8500 -> 9000  (18/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 9000 -> 10000  (10/20)
Process 8 selected, stride : 500, ticket : 200, pass : 9000 -> 9500  (19/20)
Process 8 selected, stride : 500, ticket : 200, pass : 9500 -> 10000  (20/20)
Process 8 exit
Process 7 selected, stride : 1000, ticket : 100, pass : 10000 -> 11000  (11/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 11000 -> 12000  (12/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 12000 -> 13000  (13/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 13000 -> 14000  (14/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 14000 -> 15000  (15/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 15000 -> 16000  (16/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 0 -> 1000  (17/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 1000 -> 2000  (18/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 2000 -> 3000  (19/20)
Process 7 selected, stride : 1000, ticket : 100, pass : 3000 -> 4000  (20/20)
Process 7 exit

test2 end
```

프로세스 7,8,9가 각각 ticket=100,200,300으로 실행되었다. stride 값은 1000,500,333으로 각각 계산되어, 티켓 수가 많은 프로세스일수록 stride가 작아 pass 증가가 느리게 나타났다. 그 결과 선택 빈도는 ticket 비율(1:2:3)에 근접하게 분배되었으며, stride 스케줄러가 티켓수에 비례한 공정성을 보장한다는 것을 알 수 있다.

3-3-3. test3

```
test3 start

Process 10 start
Process 11 start
Process 12 start
Process 13 start
Process 14 start
Process 15 start
Process 10 selected, stride : 285, ticket : 350, pass : 0 -> 285  (1/10)
Process 11 selected, stride : 800, ticket : 125, pass : 0 -> 800  (1/10)
Process 12 selected, stride : 111, ticket : 900, pass : 0 -> 111  (1/10)
Process 13 selected, stride : 137, ticket : 725, pass : 0 -> 137  (1/10)
Process 14 selected, stride : 190, ticket : 525, pass : 0 -> 190  (1/10)
Process 15 selected, stride : 125, ticket : 800, pass : 0 -> 125  (1/10)
Process 12 selected, stride : 111, ticket : 900, pass : 111 -> 222  (2/10)
Process 15 selected, stride : 125, ticket : 800, pass : 125 -> 250  (2/10)
Process 13 selected, stride : 137, ticket : 725, pass : 137 -> 274  (2/10)
Process 14 selected, stride : 190, ticket : 525, pass : 190 -> 380  (2/10)
Process 12 selected, stride : 111, ticket : 900, pass : 222 -> 333  (3/10)
Process 15 selected, stride : 125, ticket : 800, pass : 250 -> 375  (3/10)
Process 13 selected, stride : 137, ticket : 725, pass : 274 -> 411  (3/10)
Process 10 selected, stride : 285, ticket : 350, pass : 285 -> 570  (2/10)
Process 12 selected, stride : 111, ticket : 900, pass : 333 -> 444  (4/10)
Process 15 selected, stride : 125, ticket : 800, pass : 375 -> 500  (4/10)
Process 14 selected, stride : 190, ticket : 525, pass : 380 -> 570  (3/10)
Process 13 selected, stride : 137, ticket : 725, pass : 411 -> 548  (4/10)
Process 12 selected, stride : 111, ticket : 900, pass : 444 -> 555  (5/10)
Process 15 selected, stride : 125, ticket : 800, pass : 500 -> 625  (5/10)
Process 13 selected, stride : 137, ticket : 725, pass : 548 -> 685  (5/10)
Process 12 selected, stride : 111, ticket : 900, pass : 555 -> 666  (6/10)
Process 10 selected, stride : 285, ticket : 350, pass : 570 -> 855  (3/10)
Process 14 selected, stride : 190, ticket : 525, pass : 570 -> 760  (4/10)
Process 15 selected, stride : 125, ticket : 800, pass : 625 -> 750  (6/10)
Process 12 selected, stride : 111, ticket : 900, pass : 666 -> 777  (7/10)
Process 13 selected, stride : 137, ticket : 725, pass : 685 -> 822  (6/10)
Process 15 selected, stride : 125, ticket : 800, pass : 750 -> 875  (7/10)
Process 14 selected, stride : 190, ticket : 525, pass : 760 -> 950  (5/10)
Process 12 selected, stride : 111, ticket : 900, pass : 777 -> 888  (8/10)
Process 11 selected, stride : 800, ticket : 125, pass : 800 -> 1600  (2/10)
Process 13 selected, stride : 137, ticket : 725, pass : 822 -> 959  (7/10)
Process 10 selected, stride : 285, ticket : 350, pass : 855 -> 1140  (4/10)
Process 15 selected, stride : 125, ticket : 800, pass : 875 -> 1000  (8/10)
Process 12 selected, stride : 111, ticket : 900, pass : 888 -> 999  (9/10)
Process 14 selected, stride : 190, ticket : 525, pass : 950 -> 1140  (6/10)
Process 13 selected, stride : 137, ticket : 725, pass : 959 -> 1096  (8/10)
Process 12 selected, stride : 111, ticket : 900, pass : 999 -> 1110  (10/10)
Process 12 exit
Process 15 selected, stride : 125, ticket : 800, pass : 1000 -> 1125  (9/10)
Process 13 selected, stride : 137, ticket : 725, pass : 1096 -> 1233  (9/10)
Process 15 selected, stride : 125, ticket : 800, pass : 1125 -> 1250  (10/10)
Process 15 exit
Process 10 selected, stride : 285, ticket : 350, pass : 1140 -> 1425  (5/10)
Process 14 selected, stride : 190, ticket : 525, pass : 1140 -> 1330  (7/10)
Process 13 selected, stride : 137, ticket : 725, pass : 1233 -> 1370  (10/10)
Process 13 exit
Process 14 selected, stride : 190, ticket : 525, pass : 1330 -> 1520  (8/10)
Process 10 selected, stride : 285, ticket : 350, pass : 1425 -> 1710  (6/10)
Process 14 selected, stride : 190, ticket : 525, pass : 1520 -> 1710  (9/10)
Process 11 selected, stride : 800, ticket : 125, pass : 1600 -> 2400  (3/10)
Process 10 selected, stride : 285, ticket : 350, pass : 1710 -> 1995  (7/10)
Process 14 selected, stride : 190, ticket : 525, pass : 1710 -> 1900  (10/10)
Process 14 exit
Process 10 selected, stride : 285, ticket : 350, pass : 1995 -> 2280  (8/10)
Process 10 selected, stride : 285, ticket : 350, pass : 2280 -> 2565  (9/10)
Process 11 selected, stride : 800, ticket : 125, pass : 2400 -> 3200  (4/10)
Process 10 selected, stride : 285, ticket : 350, pass : 2565 -> 2850  (10/10)
Process 10 exit
Process 11 selected, stride : 800, ticket : 125, pass : 3200 -> 4000  (5/10)
Process 11 selected, stride : 800, ticket : 125, pass : 4000 -> 4800  (6/10)
Process 11 selected, stride : 800, ticket : 125, pass : 4800 -> 5600  (7/10)
Process 11 selected, stride : 800, ticket : 125, pass : 5600 -> 6400  (8/10)
Process 11 selected, stride : 800, ticket : 125, pass : 6400 -> 7200  (9/10)
Process 11 selected, stride : 800, ticket : 125, pass : 7200 -> 8000  (10/10)
Process 11 exit

test3 end
```

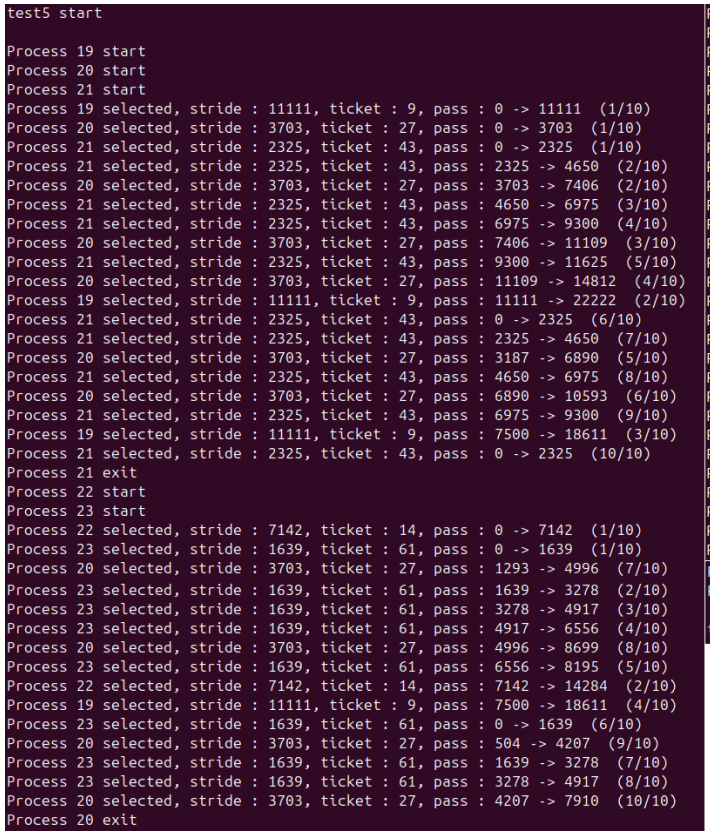
6개의 프로세스(10,11,12,13,14,15)가 서로 다른 ticket 값을 가지고 동시에 실행되었다. 출력 결과를 보면, 각 프로세스의 stride 값 차이에 따라 pass 증가 폭이 달라졌으며, CPU 선택 빈도도 이에 비례하였다. 예를 들어, 프로세스 12는 ticket=900으로 stride=111이라 pass가 111씩 증가하였고, 프로세스 11은 ticket=125로 stride=800이라 pass가 800씩 증가하였다. 결과로, 더 자주 선택되어 먼저 exit 된 것을 볼 수 있다. 또한, 프로세스 12와 프로세스 13의 pass가 333으로 동일할 때 PID가 더 작은 프로세스 12가 먼저 선택됨도 확인할 수 있었다.

3-3-4. test4

```
test4 start

Process 16 start
Process 17 start
Process 18 start
Process 16 selected, stride : 9090, ticket : 11, pass : 0 -> 9090  (1/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 0 -> 4347  (1/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 0 -> 3571  (1/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 3571 -> 7142  (2/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 4347 -> 8694  (2/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 7142 -> 10713  (3/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 8694 -> 13041  (3/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 9090 -> 18180  (2/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 0 -> 3571  (4/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 2328 -> 6675  (4/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 3571 -> 7142  (5/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 6675 -> 11022  (5/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 7142 -> 10713  (6/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 7467 -> 16557  (3/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 0 -> 3571  (7/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 309 -> 4656  (6/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 3571 -> 7142  (8/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 4656 -> 9003  (7/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 5844 -> 14934  (4/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 7142 -> 10713  (9/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 9003 -> 13350  (8/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 10713 -> 14284  (10/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 13350 -> 17697  (9/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 0 -> 3571  (11/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 650 -> 9740  (5/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 3413 -> 7760  (10/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 3571 -> 7142  (12/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 7142 -> 10713  (13/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 7760 -> 12107  (11/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 9740 -> 18830  (6/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 0 -> 3571  (14/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 1394 -> 5741  (12/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 3571 -> 7142  (15/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 5741 -> 10088  (13/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 7142 -> 10713  (16/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 7500 -> 16590  (7/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 0 -> 4347  (14/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 625 -> 4196  (17/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 4196 -> 7767  (18/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 4347 -> 8694  (15/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 6502 -> 15592  (8/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 0 -> 3571  (19/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 927 -> 5274  (16/20)
Process 18 selected, stride : 3571, ticket : 28, pass : 3571 -> 7142  (20/20)
Process 18 exit
Process 17 selected, stride : 4347, ticket : 23, pass : 5274 -> 9621  (17/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 7500 -> 16590  (9/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 0 -> 4347  (18/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 4347 -> 8694  (19/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 6969 -> 16059  (10/20)
Process 17 selected, stride : 4347, ticket : 23, pass : 0 -> 4347  (20/20)
Process 17 exit
Process 16 selected, stride : 9090, ticket : 11, pass : 7365 -> 16455  (11/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 0 -> 9090  (12/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 9090 -> 18180  (13/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 0 -> 9090  (14/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 9090 -> 18180  (15/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 0 -> 9090  (16/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 9090 -> 18180  (17/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 0 -> 9090  (18/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 9090 -> 18180  (19/20)
Process 16 selected, stride : 9090, ticket : 11, pass : 0 -> 9090  (20/20)
Process 16 exit

test4 end
```

세 개의 프로세스(16,17,18)이 각각 ticket=11,23,28을 가지고 실행되었다. 이 경우 작은 ticket 값을 가진 프로세스 16의 stride가 9090으로 매우 커서 pass 값이 급격하게 증가했고, 결국 PASS_MAX를 넘어서는 시점이 발생했다. 그 직후 pass 값이 조정되고, 결과로 가장 작은 pass 값인 0부터 시작되는 것을 통해 rebase 된 것을 볼 수 있다.

3-3-5. test5



여러 프로세스가 혼잡된 상황에서 실행되었다. 처음에는 프로세스 19,20,21이 함께 실행되었는데, ticket 값이 각각 9,27,43으로 stride가 11111,3703,2325로 계산되었다. 그 결과 프로세스 19의 pass는 한 번 선택될 때마다 11111씩 크게 증가하여 선택 간격이 길게 벌어졌고, 프로세스 21은 pass가 2325씩만 증가하여 더 자주 선택되었다. 결과로 프로세스 21이 먼저 exit 되었다. 이어서 프로세스 22,23이 추가되며 새로운 티켓 비율에 따라 선택 빈도도 다시 조정되었다. 중간에 프로세스 23이 종료된 이후에는 19,22,24가 남아서 서로다른 stride 값에 따라 선택되었으며, 이 과정에서 PASS_MAX를 초과한 프로세스의 pass 값이 rebass 되었다. 그 결과로 가장 작은 pass 값인 0이 선택되는 모습도 확인되었다. 마지막까지 비례 분배 패턴이 유지되었고, 동적인 프로세스 변화 속에서도 stride 스케줄러가 공정성을 보장하는 것을 볼 수 있었다.

4. 소스코드

4-1. proc.c

```
... (생략) ...
static struct proc*
allocproc(void)
{
  struct proc *p;
... (중략) ...
found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  // Initialize for stride scheduling
  p->tickets = 1;
  p->stride = 0;
  p->pass = 0;
  p->ticks = 0;
  p->end_ticks = -1;

  release(&ptable.lock);
... (중략) ...
int
fork(void)
{
... (중략) ...
  // Debug log for scheduler test at the start of process
  if(np->pid > 2 && curproc && curproc->pid > 2)
    cprintf("Process %d start₩n", np->pid);

  release(&ptable.lock);

  return pid;
}
// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
void
exit(void)
{
... (중략) ...
  // Debug log for scheduler test at the end of process
  if(curproc->pid > 2 && curproc->parent && curproc->parent->pid > 2)
    cprintf("Process %d exit₩n", curproc->pid);

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
... (중략) ...
static void
rebase_pass(void){
  struct proc *p;
  int flag = 0;       // Flag to check if any process has to rebase

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == RUNNABLE && p->pass > PASS_MAX){
      flag = 1;       // set flag to rebase
      break;
    }
  }
  if(!flag) return;    // No process has to rebase
```

```
  uint mn = (uint)-1;     // Minimum pass value among
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    // Find minimum pass value
    if(p->state == RUNNABLE && p->pass < mn) mn = p->pass;
  }
  if(mn == (uint)-1) return;     // No RUNNABLE process

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    // Only consider RUNNABLE processes
    if(p->state != RUNNABLE) continue;

    // Rebase pass values
    if(p->pass <= mn) p->pass = 0;
    else if(p->pass > mn){
      p->pass -= mn;
      if(p->pass > DISTANCE_MAX) p->pass = DISTANCE_MAX;
    }
  }
}

//PAGEBREAK: 42
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.   It loops, doing:
//   - choose a process to run
//   - swtch to start running that process
//   - eventually that process transfers control
//         via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p, *temp;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);

    // Rebase pass values if necessary
    rebase_pass();

    // Find process with minimum pass value
    temp = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      // Only consider RUNNABLE processes
      if(p->state != RUNNABLE) continue;

      // Select process with minimum pass value
      if(temp == 0 || p->pass < temp->pass || (p->pass == temp->pass && p->pid < temp->pid))
        temp = p;
    }

    // No RUNNABLE process
    if(temp == 0){
      release(&ptable.lock);
      continue;
    }
```

```
      // Switch to chosen process.
      c->proc = temp;
      switchuvm(temp);
      temp->state = RUNNING;
      swtch(&c->scheduler, temp->context);
      switchkvm();

      // Process is done running
      c->proc = 0;

      // drop ptable.lock before next round
      release(&ptable.lock);
    }
  }
  ... (생략) ...
```

4-2. proc.h

```
  ... (생략) ...
  // Per-process state
  struct proc {
    uint sz;                     // Size of process memory (bytes)
  ... (중략) ...
    char name[16];               // Process name (debugging)
    int tickets;                 // Number of tickets
    uint stride;                 // Stride value for stride scheduling
    uint pass;                   // Pass value for stride scheduling
    int ticks;                   // Number of ticks the process has run
    int end_ticks;               // Number of ticks the process has run in the end
  };

  // Stride scheduling constants which are fixed by spec
  #define STRIDE_MAX 100000
  #define PASS_MAX 15000
  #define DISTANCE_MAX 7500
  ... (생략) ...
```

4-3. syscall.c

```
... (생략) ...
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_settickets(void); // New system call for settickets

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
... (중략) ...
[SYS_close]   sys_close,
[SYS_settickets] sys_settickets, // New system call for settickets
};
... (생략) ...
```

4-4. syscall.h

```
  ... (생략) ...
  #define SYS_mkdir   20
  #define SYS_close   21
  #define SYS_settickets 22 // New system call number for settickets
```

4-5. sysproc.c

```
  ... (생략) ...
  // New system call for project 2
  int
  sys_settickets(void)
  {
    int tickets, end_ticks;
    struct proc *p = myproc();
```

```
    // Invalid arguments
    if(argint(0, &tickets) < 0 || argint(1, &end_ticks) < 0) return -1;
    // Invalid num of tickets
    if(tickets < 1 || tickets > (STRIDE_MAX -1)) return -1;

    // Set the number of tickets and end_ticks for the process
    p->tickets = tickets;
    p->stride = STRIDE_MAX / tickets;

    // Set end_ticks only if it is valid
    if(end_ticks >= 1) p->end_ticks = end_ticks;

    return 0;
  }
```

4-6. trap.c

```
... (생략) ...
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER){
    struct proc *p = myproc();

    // update ticks for the process
    p->ticks++;

    // Debug log for scheduler test at each scheduling
    if(p->pid > 2 && p->parent && p->parent->pid > 2)
      cprintf("Process %d selected, stride : %d, ticket : %d, pass : %d -> %d   (%d/%d)₩n",
              p->pid, p->stride, p->tickets, p->pass, p->pass+p->stride, p->ticks, p->end_ticks);

    // update pass for the process
    p->pass += p->stride;

    // If end_ticks is set, check it and exit if the process has run enough
    if(p->end_ticks > 0 && p->ticks >= p->end_ticks) exit();

    // yield the CPU
    yield();
  }
```

4-7. user.h

```
... (생략) ...
int atoi(const char*);
// New system calls for project 2
int settickets(int tickets, int end_ticks); // set number of tickets and end_ticks for the process
```

4-8. usys.S

```
... (생략) ...
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(settickets)
```

4-9. lapic.c

```
... (생략) ...
lapicw(TDCR, 0x2);              // divide by 8
lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
lapicw(TICR, 20000000);     // initial count
... (생략) ...
```

4-10. Makefile

```
... (생략) ...
UPROGS=₩
    _cat₩
    _echo₩
    ... (중략) ...
    _zombie₩
    _debug_test₩
    _syscall_test₩
    _scheduler_test₩
... (생략) ...
```