PA2 REPORT

전자공학도의 윤리 강령 (IEEE Code of Ethics)

(출처: http://www.ieee.org)

나는 전자공학도로서, 전자공학이 전 세계 인류의 삶에 끼치는 심대한 영향을 인식하여 우리의 직업, 동료와 사회에 대한 나의 의무를 짐에 있어 최고의 윤리적, 전문적 행위를 수행할 것을 다짐하면서, 다음에 동의한다.

1. **공중의 안전, 건강 복리에 대한 책임**: 공중의 안전, 건강, 복리에 부합하는 결정을 할 책임을 질 것이며, 공중 또는 환경을 위협할 수 있는 요인을 신속히 공개한다.
2. **지위 남용 배제**: 실존하거나 예기되는 이해 상충을 가능한 한 피하며, 실제로 이해가 상충할 때에는 이를 이해 관련 당사자에게 알린다. (이해 상충: conflicts of interest, 공적인 지위를 사적 이익에 남용할 가능성)
3. **정직성**: 청구 또는 견적을 함에 있어 입수 가능한 자료에 근거하여 정직하고 현실적으로 한다.
4. **뇌물 수수 금지**: 어떠한 형태의 뇌물도 거절한다.
5. **기술의 영향력 이해**: 기술과 기술의 적절한 응용 및 잠재적 영향에 대한 이해를 높인다.
6. **자기계발 및 책무성**: 기술적 능력을 유지, 증진하며, 훈련 또는 경험을 통하여 자격이 있는 경우이거나 관련 한계를 전부 밝힌 뒤에만 타인을 위한 기술 업무를 수행한다.
7. **엔지니어로서의 자세**: 기술상의 업무에 대한 솔직한 비평을 구하고, 수용하고, 제공하며, 오류를 인정하고 수정하며, 타인의 기여를 적절히 인정한다.
8. **차별 안하기**: 인종, 종교, 성별, 장애, 연령, 출신국 등의 요인에 관계없이 모든 사람을 공평하게 대한다.
9. **도덕성**: 허위 또는 악의적인 행위로 타인, 타인의 재산, 명예, 또는 취업에 해를 끼치지 않는다.
10. **동료애**: 동료와 협력자가 전문분야에서 발전하도록 도우며, 이 윤리 헌장을 준수하도록 지원한다.

위 IEEE 윤리헌장 정신에 입각하여 report를 작성하였음을 서약합니다.

학 부: 전자공학과

제출일: 2025/11/21

과목명: 전자공학운영체제

교수명: 오영환 교수님

분 반: C064-1

학 번: 202021025, 202126989

성 명: 안준영, 유현종

1. 목표

xv6에서 getreadcount(), settickets(), getpinfo() 시스템 콜을 구현한다. 이를 활용해 기존의 라운드 로빈 스케줄러를 Lottery Scheduler 및 2-level Lottery Scheduler로 교체하고 테스트한다. 이를 통해 실제 커널 xv6에 대한 이해를 증진한다.

2. 과제 지시 사항

2.1 Phase1: getreadcount()

1) 새로운 시스템 콜, getreadcount() 구현한다.

2) getreadcount()는 커널 부팅 이후read() 시스템 콜이 사용자 프로세스에 의해 호출된 총 횟수를 반환한다.

3) ./test - getreadcount.sh 스크립트로 테스트한다.

2.2 Phase2: Lottery Scheduler

2.2.1 새로운 시스템 콜

1) int settickets(int number)

호출한 프로세스의 티켓 수 설정. 성공 시 0을, 실패 시-1을 리턴한다.

2) int getpinfo(struct pstat*)

모든 실행 중인 프로세스의 상태 정보를 반환. 성공 시0, 실패 시-1을 리턴한다.

2.2.2 fork() 시스템 콜 수정

자식 프로세스는 부모와 동일한 티켓 수를 상속받음. 자식의 ticks(실행 횟수)는0으로 초기화되어야 한다.

2.2.3 scheduler() 함수 수정

proc.c의 round-robin scheduling을 구현하고 있는 scheduler() 함수를 Lottery로 교체한다. 티켓 수 내의 난수를 기반으로 실행할 프로세스를 선택한다. proc.c 파일의 rand(), srand()를 사용한다.

2.2.4 Test Lottery Scheduler

xv-6public/test_lottery.c로 테스트한다.

2.3 Phase3: 2-Level Lottery Scheduler

2.3.1. 2-Level Lottery Scheduler

1) 레벨1 & 레벨2

레벨1: 커널이 유효 티켓 기반으로 당첨될 UID 선택한다.

레벨2: UID에 속한 실행 가능한 프로세스 내에서 당첨될 프로세스를 선택한다.

2) 상한

상한은 1000개이다. 사용자의 총 티켓(모든 프로세스의 합)이1000개를 넘어가면 레벨1 추첨에서 해당 사용자의 유호 티켓은 1000개로 간주한다.

2.3.2 주요 기능 구현

1) int setuid(int uid)

새로운 시스템 콜로, 호출 프로세스의 UID를 설정한다.

2) int fork()

기존의 것 수정해서 구현한다. 자식 프로세스가 부모의 uid와 티켓 수를 상속하게 한다.

3) int getpinfo(struct pstat *st)

pstat 구조체 st에 uid, tickets, tick 정보가 포함되도록 업데이트한다.

4) scheduler()

로터리 스케줄러를 2-레벨로 구현한다.

2.3.3 테스트

./test-currency.sh로 테스트한다.

3. 수행 과정

3.1 Phase 1 과제 수행 과정

3.1.1 데이터 구조

1) proc.h / pstat.h

proc.h

```
// For lottery scheduler
int inuse;
int ticks;
int tickets;
int getreadcount; // ph1
```

pstat.h

```
struct pstat {
  int inuse[NPROC];
  int tickets[NPROC];
  int pid[NPROC];
  int ticks[NPROC];
};
```

proc.h의 proc 구조체 내에 getreadcount를 추가했다. 이를 통해 read 시스템 콜이 호출된 횟수가 프로세스마다 따로 저장될 수 있다. phase1에서는 read 시스템 콜을 몇 번 호출 했는지만 알면 되기 때문에 uid는 전혀 사용되지 않기 때문에 pstat 구조체에서 지웠다.

2) defs.h

```
int             getreadcount(void); //ph1
```

시스템 콜 연결을 위해서 커널 내의 다른 곳에서도 함수가 인식될 수 있게 선언했다.

3.1.2 시스템 콜 등록

1) syscall.h

```
#define SYS_getreadcount 22
```

syscall.h에서는 맨 밑에 #define SYS_getreadcount 22를 추가했다. 이는 getreadcount라는 시스템 콜에 고유한 번호를 붙여주기 위함이다.

2) syscall.c

```
extern int sys_getreadcount(void);
```

```
[SYS_getreadcount]    sys_getreadcount,
```

왼쪽과 같이 선언해주고, 오른쪽과 같이 syscalls 배열에 추가했다. 이를 통해서 유저가 SYS_getreadcount의 번호(22)를 호출했을 때 커널의 sys_getreadcount 함수를 연결할 수 있다.

3) user.h

```
int getreadcount(void);
```

user.h에 함수 선언을 추가했다. 이는 유저 프로그램이 getreadcount 함수를 C언어 코드로 쉽게 호출하는데 필요하다.

4) usys.S

```
SYSCALL(getreadcount)
```

user.h의 C 코드가 SYS_getreadcount 번호를 레지스터에 넣고, 커널을 호출하는 어셈블리 코드로 자동 변환되도록 하기 위해서 위와 같이 추가했다.

3.1.3 함수 구현

1) sysfile.c

```
int read_cnt = 0;
```

```
int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;
  read_cnt ++;
  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}

int
sys_getreadcount(void){ // Ph1
  return read_cnt;
}
```

전역 변수로 read_cnt = 0을 선언한다. 시스템 콜이 호출된 횟수를 세고 이 변수에 저장한다. 횟수를 세는 방법은 sys_read 함수 내부에 read_cnt ++;을 추가해서 프로세스가 이 함수를 호출하면 이 변수의 값을 1 늘리는 식으로 구현한다. 아래 sys_getreadcount 함수를 따로 추가해 지금까지 누적된 read_cnt 값을 유저에게 반환한다. 이를 통해 지금까지 read 함수가 호출된 횟수를 알 수 있다.

2) proc.c / sysproc.c

두 코드에서 phase1에서는 불필요한 uid에 관한 코드를 제거했다. 먼저proc.c의 allocproc에서 uid에 관련된 코드를 제거했다. 그 다음으로 sysproc.c에서는 sys_setuid를 제거했다.

3.1.4 로직 분석

크게 유저가 read를 요청할 때, 카운터 값을 요청할 때로 나눌 수 있다. 먼저 read를 요청할 때 흐름은 다음과 같다. 유저 프로그램이 read 함수를 호출한다. 그러면usys.S를 통해서 커널 모드로 들어간다. 그러면 syscall.c가 SYS_read의 번호를 확인하고, sys_read 함수를 호출한다. 호출된 read 함수는 읽기 동작을 수행하면서 추가한 코드 read_cnt++에 의해서 카운터 값이 증가한다.

다음은 유저가 카운터 값을 요청했을 때의 흐름이다. 마찬가지로 유저 프로그램이 getreadcount 함수를 호출한다. user.h에 선언된 함수 원형을 통해서 usys.S의 어셈블리 코드가 실행된다. 레지스터에 SYS_getreadcount 번호를 넣으면 커널의 syscall() 함수가 그 번호를 통해서 sys_getreadcount 함수를 호출한다. 함수가 호출되면 별다른 과정 없이 read_cnt 값을 그대로 반환한다.

3.1.5 테스트 실행 결과

initial-xv6 디렉터리에서 ./test-getreadcount.sh를 실행했다. 테스트 프로그램이 루프를 돌며 read 함수를 여러 번 실행하고 getreadcount를 실행했을 때 반환된 값이 read 함수를 실행한 횟수와 같은지 확인한다. 두 개의 테스트 코드를 모두 통과해 getreadcount가 read의 호출 횟수를 정확히 집계하고 있음을 확인했다.

[Phase 1 테스트 결과]

```
test 1: passed
test 2: passed
```

3.1.6 결론

새로 만든 시스템 콜 getreadcount가 성공적으로xv6 커널에 등록되었다. 또한 유저 공간과 커널 공간 사이의 트랩 매커니즘이 정상적으로 동작하는 것도 테스트 통과를 통해 확인했다.

3.2 Phase 2 과제 수행 과정

3.2.1 데이터 구조

```
int inuse;
int ticks;
int tickets;
int getreadcount; // PH1
```

```
struct pstat {
  int inuse[NPROC];
  int tickets[NPROC];
  int pid[NPROC];
  int ticks[NPROC];
};
```

3.2.2 시스템 콜 등록

1) syscall.h에 새로 추가할 시스템 콜 settickets, getpinfo를 위해 새 번호를 추가(할당)한다.

```
#define SYS_getreadcount 22
#define SYS_settickets 23
#define SYS_getpinfo 24
```

2) syscall.c에 시스템 콜 테이블을 연결하기 위해 sys_settickets와 sys_getpinfo를 extern으로 선언한다. 그리고syscalls[] 배열 맨 아래에 두 함수를 추가한다.

```
extern int sys_getreadcount(void);
extern int sys_settickets(void);
extern int sys_getpinfo(void);
```

```
[SYS_getreadcount]    sys_getreadcount,
[SYS_settickets] sys_settickets,
[SYS_getpinfo] sys_getpinfo,
```

3) user.h에 int settickets(int); int getpinfo(struct pstat*);를 선언해 호출할 수 있도록 원형을 추가한다. 그리고 struct pstat을 선언한다.

```
struct pstat;
```

```
int getreadcount(void); //ph1

int settickets(int);
int getpinfo(struct pstat*);
```

4) usys.S에다가 스텁코드를 추가해 실제 커널이 번호를 이용해 호출 하도록한다.

```
SYSCALL(getreadcount)
SYSCALL(settickets)
SYSCALL(getpinfo)
```

3.2.3 함수 구현

1) defs.h에 아래의 두 함수를 선언한다. proc.c에 만들 함수들을 sysproc.c가 알 수 있도록 defs.h에 선언한다.

```
int             getreadcount(void); // ph1
int             settickets(int);
int             getpinfo(struct pstat*);
```

2) sysproc.c에, user program에서 넘어온 값인 티켓 수, 포인터를 커널이 쓸 수 있게 변환하고, settickets와 getpinfo를 호출하는 래퍼 함수를 만든다.

```
#include "pstat.h"
int
sys_settickets(void)
{
        int n; // 티켓 숫자
        if(argint(0, &n) < 0) // (int)0 -> n에 저장
                return -1;

        return settickets(n);
}

int
sys_getpinfo(void)
{
        struct pstat *p; // pstat

        if(argptr(0, (void*)&p, sizeof(*p)) < 0) // 첫 인자를 포인터 변환해 저장
                return -1;

        return getpinfo(p);
}
```

3) proc.c의 settickets와 getpinfo 함수를 구현한다.

```
#include "pstat.h"
int
sys_settickets(void)
{
        int n; // 티켓 숫자
        if(argint(0, &n) < 0) // (int)0 -> n에 저장
                return -1;

        return settickets(n);
}

int
sys_getpinfo(void)
{
        struct pstat *p; // pstat

        if(argptr(0, (void*)&p, sizeof(*p)) < 0) // 첫 인자를 포인터 변환해 저장
                return -1;

        return getpinfo(p);
}
```

4) proc.c의 allocproc 함수 중, 프로세스 생성 시 proc.h에 추가한 변수들을 초기화한다.

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->inuse = 1; // 사
  p->tickets = 10; //
  p->ticks = 0;
```

5) proc.c의 wait() 함수 중, 프로세스 정리 시 inuse 리셋을 추가한다.

```
// Found one.
pid = p->pid;
kfree(p->kstack);
p->kstack = 0;
freevm(p->pgdir);
p->pid = 0;
p->parent = 0;
p->name[0] = 0;
p->killed = 0;
p->inuse = 0; // 사용안함
p->state = UNUSED;
release(&ptable.lock);
return pid;
```

6) proc.c의 fork() 중, 자식 프로세스가 부모 프로세스의 티켓 수를 상속받게 한다.

```
np->sz = curproc->sz;
np->parent = curproc;
*np->tf = *curproc->tf;
np->tickets = curproc->tickets; // 티켓 수 상속받기
// Clear %eax so that fork returns 0 in the child.
np->tf->eax = 0;
```

7) proc.c의 pinit() 중, 난수 발생을 위해 srand(1);을 추가한다.

```
static void srand(uint);
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
  srand(1);
}
```

8) proc.c의 scheduler를 Lottery 스케줄러로 교체한다.

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

  for(;;){
    sti();

    int total_tickets = 0;

    //실행 가능(RUNNABLE) 프로세스 애들의 모든 티켓수 계산하기
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state == RUNNABLE) {
        total_tickets += p->tickets;
      }
    }
    // 실행할수있는거있음
    if(total_tickets > 0) {
      // 당첨 번호 만들기(0~total_tickets-1)
      int winner_ticket = rand() % total_tickets;
      int counter = 0;

      // 당첨자 찾기
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
          continue;
        // 티켓을 누적하다가 당첨 번호를 넘어설때의  프로세스가 당첨자
        counter += p->tickets;
        if (counter > winner_ticket) {

          // 당첨자 실행
          c->proc = p;
          switchuvm(p);
          p->state = RUNNING;

          p->ticks++; // 이 프로세스의 실행 횟수(ticks) 1 증가

          swtch(&(c->scheduler), p->context);
          switchkvm();

          // Process is done scheduling.
          c->proc = 0;

          break; // 당첨자를 찾았으니 내부 루프 탈출하기
        }
      }
    }
    release(&ptable.lock);
  }
}
```

9) Makefile을 수정해 test_lottery 파일을 만들게 한다. phase2에 필요 없는 uid 관련 코드들을 지웠기 때문에 test_currency는 주석 처리한다.

```
UPROGS=\
        _cat\
        _echo\
        _forktest\
        _grep\
        _init\
        _kill\
        _ln\
        _ls\
        _mkdir\
        _rm\
        _sh\
        _stressfs\
        _usertests\
        _wc\
        _zombie\
        _test\
        _test_lottery\
        #_test_currency\
```

```
EXTRA=\
        mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
        ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
        printf.c umalloc.c\
        test_lottery.c\
        test_currency.c\
        README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
        .gdbinit.tmpl gdbutil\
```

3.2.4 로직 설명

1) Scheduler

- 초기화

스케줄러는 무한 루프 속에서 운영체제가 꺼질 때까지 동작한다. 시작하기 전에 총 티켓 수를 저장할 변수 total_ticket을 선언하고, 다른 cpu가 동시에 프로세스를 컨트롤하지 않도록 acquire(&ptable.lock)을 통해 잠근다.

- 전체 티켓 수 집계

준비를 마치면 for 문을 통해 프로세스 목록을 처음부터 끝까지 훑으면서 전체 티켓의 개수를 구한다. 프로세스가 실행 가능한 상태인 경우에만 그 프로세스의 티켓의 수를 total_ticket 변수에 더해준다. for 문이 끝나면 실행 가능한 프로세스의 전체 티켓의 수가 total_ticket에 저장된다.

-당첨 번호 추첨

전체 티켓의 개수를 구한 다음은 당첨 번호를 뽑는다. 0부터 뽑기 때문에 번호는 0부터 total_ticket-1 만큼 있고, 당첨 번호는 그 중 랜덤하게 뽑는다. 이는 랜덤한 수를 total_ticket으로 나눈 나머지를 통해 구현한다. 당첨 번호를 정하고 나서는 당첨자를 찾기 위한 카운터인 counter 변수를 선언한다.

- 당첨 프로세스 찾기

당첨 번호도 정해졌고, 당첨자를 찾기 위한 변수도 선언했으니, 이제 당첨자를 찾는다. 전체 티켓의 수를 구할 때처럼 프로세스 목록을 처음부터 끝까지 훑으면서 당첨자를 찾는다. 마찬가지로 실행 불가능한 경우에는 건너뛴다. 프로세스가 실행 가능한 경우에만 counter에 티켓의 수를 더한다. 그렇게 더하다가 counter가 당첨 번호인 winner_ticket을 넘어서면 그 프로세스가 당첨자가 된다.

- 당첨 프로세스 실행

당첨자가 된 프로세스는 실행된다. 이 프로세스의 ticks 변수를 1 증가시켜서 실행됐음을 확인한다. switchuvm(p)를 통해 유저 가상 메모리로 전환하고, 이 프로세스의 상태를 RUNNABLE에서 RUNNING으로 바꾼다. 그리고 swtch 코드를 통해 현재 스케줄러의 실행 상태를 저장하고 당첨된 프로세스의 상태를 불러와 그 프로세스를 실행한다. 프로세스의 실행이 종료되면 이전에 실행 상태를 저장해뒀기 때문에 스케줄러의 그 다음 줄인 switchkvm()을 통해 커널 가상 메모리로 전환한다.

-정리 및 재시작

당첨자를 찾고 실행하기까지 끝났으니 c->proc=0를 통해서 CPU가 실행 중인 프로세스가 없음을 나타낸다. 그리고 break를 통해 당첨자를 찾는 for문을 탈출한다. 그리고 다시 처음으로 돌아가 지금까지의 과정을 반복한다.

2) settickets : 티켓 수 설정하는 시스템콜

티켓의 수(n)가 1보다 작은 경우 추첨을 할 수 없기 때문에 실패이므로 -1을 반환한다. 그렇지 않은 경우에는 프로세스의 티켓의 수를 인자인 n으로 하고 성공적으로 변경됐으므로 1을 반환한다. 예를 들어 어떤 프로세스A에서 settickets(30)을 호출하면A->tickets에다가 30을 저장한다.

3) getpinfo : 시스템에서 올라간 모든 프로세스의 상태 정보를pstat에 담기

데이터를 읽는 도중에 다른 쪽에서 프로세스를 건드리면 안 되므로 lock을 건다. 그 후 프로세스 테이블을 처음부터 끝까지 돌며 구조체 pstat p의 배열에 커널 정보를 복사해 넣는다. 다 복사했다면 lock을 풀고 1을 반환한다. 유저 프로그램은 이 함수를 호출해서 값을 확인한다.

3.2.5 테스트 실행 결과

6번의 실행 결과, 19057 : 12844 : 6340으로 평균 3.0058 : 2.026 : 1의 CPU 할당 횟수 비를 갖게 되는 것을 확인했다. 개별 테스트 결과는 아래 표 1, 표 2에서 확인할 수 있다.

[표 1] 테스트 결과

[테스트 1]	[테스트 2]	[테스트 3]
$ test_lottery pid:3, pid:4, pid:5 tickets:30, tickets:20, tickets:10 3178, 2104, 1086,	$ test_lottery pid:6, pid:7, pid:8 tickets:30, tickets:20, tickets:10 3177, 2098, 1042,	$ test_lottery pid:9, pid:10, pid:11 tickets:30, tickets:20, tickets:10 3171, 2030, 1008,
[테스트 4]	**[테스트 5]**	**[테스트 6]**
$ test_lottery pid:12, pid:13, pid:14 tickets:30, tickets:20, tickets:10 3179, 2306, 1070,	$ test_lottery pid:15, pid:16, pid:17 tickets:30, tickets:20, tickets:10 3181, 2292, 1078,	$ test_lottery pid:18, pid:19, pid:20 tickets:30, tickets:20, tickets:10 3171, 2104, 1056,

[표 2] 테스트 결과2

[테스트 1]	[테스트 2]	[테스트 3]	[테스트 4]	[테스트 5]	[테스트 6]
2.926:1.937:1	3.049:2.008:1	3.146:2.014:1	2.971:2.155:1	2.951:2.126:1	3.003:1.992:1

3.2.6 결론

테스트 결과, 구현한 Lottery Scheduler의 스케줄링에 따라 티켓 수의 비인 3:2:1에 매우 근접하는 비율로 스케줄링되는 것을 확인할 수 있었다. Lottery Scheduler의 특성 상, 총 실행 횟수가 많을수록 CPU 할당 비가 티켓 비에 근접해지는 것 역시 6번의 테스트 평균 비율을 확인하며 확인할 수 있었다. 또한, 확률에 의해 실행되기에 티켓 수가 10개로 적은 프로세스 역시 기아 상태에 빠지지 않고 실행됨을 확인할 수 있었다.

3.3 Phase 3 과제 수행 과정

3.3.1 데이터 구조 및 헤더

1) proc.h의 proc 구조체와 **pstat.h**의 pstat 구조체 내에 uid를 추가했다. 이는 이 프로세스의 소유자 id를 나타낸다.

proc.h

```
// For lottery scheduler
int getreadcount; // Ph1
int inuse;
int ticks; // Ph3
int tickets; // Ph2
int uid; // Ph3
```

pstat.h

```
struct pstat {
  int inuse[NPROC];
  int tickets[NPROC];
  int pid[NPROC];
  int ticks[NPROC];
  int uid[NPROC];
};
```

2) defs.h

```
int setuid(int); // Ph3
int getreadcount(void); // Ph1
int settickets(int); // Ph2
int getpinfo(struct pstat*); // PH2
```

defs.h에 setuid를 선언해서proc.c에서 만들어진 이 함수가 다른 파일에서도 호출될 수 있도록 한다.

3.3.2 시스템 콜 등록

1) syscall.h

```
#define SYS_setuid 22 // Ph3
#define SYS_getreadcount 23 // Ph1
#define SYS_settickets 24 // Ph2
#define SYS_getpinfo 25 // Ph2
```

#define SYS_setuid 22를 추가했다. 이를 통해 SYS_setuid 시스템 콜에 고유한 번호(22)를 부여했다.

2) syscall.c

```
extern int sys_setuid(void); // Ph3
extern int sys_getreadcount(void); // Ph1
extern int sys_settickets(void); // Ph2
extern int sys_getpinfo(void); // Ph2
```

```
[SYS_setuid]     sys_setuid, // Ph3
[SYS_getreadcount] sys_getreadcount, // Ph1
[SYS_settickets] sys_settickets, // Ph2
[SYS_getpinfo] sys_getpinfo, // Ph2
```

왼쪽과 같이 선언하고, syscalls 배열에 오른쪽과 같이 추가했다. 이를 통해 시스템 콜 번호와 sysproc.c의 함수를 연결한다.

3) user.h

```
int setuid(int); // Ph3
int getreadcount(void); // Ph1
int settickets(int); // Ph2
int getpinfo(struct pstat*); // PH2
```

유저 프로그램에서 사용할 함수의 원형(int setuid(int))을 선언한다.

4) usys.S

```
SYSCALL(setuid)# Ph3
SYSCALL(getreadcount)# Ph1
SYSCALL(settickets)# Ph2
SYSCALL(getpinfo)# Ph2
```

SYSCALL(setuid)를 추가해서 실제 커널이 번호를 이용해서 호출할 수 있도록 한다.

3.3.3 함수 구현

1) sysproc.c

```
int
sys_setuid(void) // Ph 3
{
  int uid;
  if(argint(0, &uid) < 0)
    return -1;

  if(uid < 0 || uid >= NUID)
    return -1;

  myproc()->uid = uid;
  return 0;
}
```

현재 실행 중인 프로세스의 uid를 설정하는 함수를 만든다. 유저 프로그램에서 받은 인자가 범위 내에 있는지 확인하는 과정을 거친 후 그 값을uid로 설정한다.

2) proc.c

2-1) int getpinfo(struct pstat *p)

```
int
getpinfo(struct pstat *p) // Ph2 & Ph3
{
  struct proc *proc;
  int i = 0;

  // Null일때 거름. 실패 처리
  if (p == 0)
    return -1;

  // 프로세스 테이블 접근 보호
  acquire(&ptable.lock);

  // 모든 프로세스 순회
  for(proc = ptable.proc; proc < &ptable.proc[NPROC]; proc++, i++){
    // proc.h의 필드들을, pstat에다가 복사하기
    p->inuse[i] = proc->inuse;
    p->tickets[i] = proc->tickets;
    p->pid[i] = proc->pid;
    p->ticks[i] = proc->ticks; // Ph2
    p->uid[i] = proc->uid; // Ph3
  }

  release(&ptable.lock);

  return 0; // 성공처리
}
```

pstat에 복사해 넣을 정보에 프로세스의 uid 정보도 추가한다.

2-2) fork()

```
np->sz = curproc->sz;
np->parent = curproc;
*np->tf = *curproc->tf;
np->uid = curproc->uid; // Ph3
np->tickets = curproc->tickets; // Ph2 티켓 수 상속받기
// Clear %eax so that fork returns 0 in the child.
np->tf->eax = 0;
```

phase2에서 자식 프로세스가 부모 프로세스의 티켓 수를 상속 받는 것과 같은 방식으로 자식 프로세스가 부모 프로세스의 uid도 상속 받게한다.

2-3) allocproc(void)

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->uid = myproc() ? myproc()->uid : 0; // Phase 3
  p->inuse = 1; // 사용중으로 표시
  p->ticks = 0; // 0으로 초기화해야지 // PH2
  p->tickets = 10;
```

p->uid = myproc() ? myproc()->uid : 0;을 추가해서 부모 프로세스가 존재한다면 부모 프로세스의 uid를 자식 프로세스에 상속시킨다. 만약 없다면 0으로 초기화시킨다.

2-4) scheduler(void)

proc.c의 scheduler()를2-level Lottery Scheduler로 교체한다.

[Level 1]

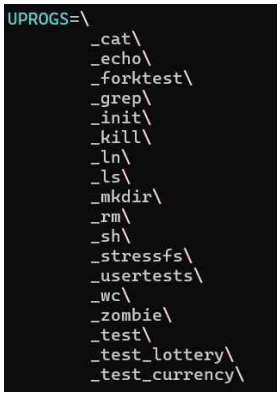
```
void
scheduler(void) // Ph3 level 2 Scheduler
{
struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    sti();

    acquire(&ptable.lock);

    // Level 1
    int uid_list[NPROC];
    int uid_tickets[NPROC];
    int n_users = 0;

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

      int u = p->uid;
      int idx;

      // uid가 이미 uid_list에 있는지 찾기
      for(idx = 0; idx < n_users; idx++){
        if(uid_list[idx] == u)
          break;
      }

      // 처음 보는 uid면 새로 추가
      if(idx == n_users){
        uid_list[n_users] = u;
        uid_tickets[n_users] = 0;
        n_users++;
      }

      // 해당 uid의 티켓 합산 (프로세스 tickets)
      uid_tickets[idx] += p->tickets;
    }
```

```
    // RUNNABLE 없을때
    if(n_users == 0){
      release(&ptable.lock);
      continue;
    }

    // 유저의 총 티켓
    int total_user_tickets = 0;
    for(int i = 0; i < n_users; i++){
      int eff = uid_tickets[i];
      if(eff > USER_CURRENCY) // 상한선
        eff = USER_CURRENCY;

      if(eff < 0)
        eff = 0;

      uid_tickets[i] = eff;
      total_user_tickets += eff;
    }

    if(total_user_tickets == 0){ // 유저없으면 안함
      release(&ptable.lock);
      continue;
    }

    int user_winner = rand() % total_user_tickets; // 유저당첨
    int sum = 0;
    int winner_uid = -1;

    for(int i = 0; i < n_users; i++){
      sum += uid_tickets[i];
      if(sum > user_winner){
        winner_uid = uid_list[i];
        break;
      }
    }
```

[Level 2]

```
    // Level 2 Start
    int total_proc_tickets = 0;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state == RUNNABLE && p->uid == winner_uid)
        total_proc_tickets += p->tickets;
    }

    if(total_proc_tickets == 0){ // 방어
      release(&ptable.lock);
      continue;
    }

    int proc_winner = rand() % total_proc_tickets; // 프로세스당첨
    int counter = 0;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE || p->uid != winner_uid)
        continue;

      counter += p->tickets;
      if(counter > proc_winner){
        // 최종 당첨 프로세스 실행
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;


        swtch(&(c->scheduler), p->context);
        switchkvm();
        c->proc = 0;

        break;  // 당첨자 찾았으니 루프 탈출
      }
    }

    release(&ptable.lock);
  }
}
```

3) Makefile 수정



qemu-nox에서의 테스트를 위해 _teset_currency를 추가한다.

3.3.4 로직 설명

phase 2 때와 마찬가지로 무한 루프를 돌면서 운영 체제가 꺼질 때까지 동작한다. 그리고 다른 코어와 충돌하지 않도록 프로세스 테이블에 lock을 건다. 준비를 마친 후 2-level scheduling을 구현한다. 먼저 유저를 고르는 level 1을 구현한다.

[Level 1]

level 1을 구현하기 위해 유저에 대한 정보를 저장할 변수 및 배열을 선언한다. uid_list에 발견된 유저의 목록을 저장하고, uid_tickets에 각 유저 별 티켓의 합계를 저장하고, n_users에는 실행 가능한 유저의 수를 저장한다.

- 활성 사용자 탐색 및 티켓 세기

선언을 마친 후에 프로세스 테이블을 스캔하면서 실행 가능한(RUNNABLE) 프로세스를 찾는다. uid_list에 없는 새로운 UID가 나오면 등록하고, 이미 있는 uid면 해당 인덱스 idx를 찾는다. 그리고 그 인덱스의 uid_tickets에 현재 프로세스의 티켓을 더해서 해당 uid의 티켓을 합산한다. 이 결과 uid_list에는 활성 유저 목록이 저장되고, uid_tickets에는 각 사용자가 실제로 가진 티켓의 총합이 저장된다. 만약 실행 가능한 프로세스가 없다면 lock을 푼다.

- currency limit 적용

앞서 구한 유저의 총 티켓 개수(eff)에는 1000장이라는 한도가 존재하기 때문에 이를 구현해준다. for 문을 통해 유저 한 명씩 확인하면서 유저의 총 티켓이 1000을 넘는다면 강제로 1000으로 줄인다. 이 한도 값을 적용한 총 티켓 수를 다시 uid_tickets 배열에 저장한다. 그리고 유저 전체의 티켓의 합도 한도가 적용된 값을 더해서 계산한다. 만약 유저 전체의 티켓의 합이 0이면 lock을 푼다.

- 유저 추첨

당첨 유저(user_winner)는0부터 전체 유효 티켓의 수 -1까지의 수 중의 랜덤한 수 하나가 된다.

유저의 티켓의 수를 계속해서 더해가다가 그 합이 user_winner보다 커지게 되면 그 유저가 당첨자가 된다.

[Level 2]

- 최종 당첨 프로세스 추첨

당첨된 유저의 내부에서 phase2에서와 같이 lottery 추첨을 통해서 최종 당첨 프로세스를 뽑는다. 프로세스 테이블을 돌면서 실행 가능하고, 당첨 유저의 프로세스일 때만 카운터에 티켓 값을 더해준다. 그러다 카운터 값이 랜덤하게 정한 당첨 숫자를 넘어서면 해당 프로세스가 당첨 프로세스가 된다. 당첨된 프로세스를 context switch해서 실행하고 끝나면 다시 스케줄러로 돌아온다.

-정리 및 재시작

2-level로 당첨 유저와 프로세스까지 뽑았고, 그 프로세스를 실행까지 끝냈기 때문에 c->proc=0를 통해서 CPU가 실행 중인 프로세스가 없음을 나타낸다. 그리고 lock을 풀고 다시 처음으로 돌아가 당첨 유저와 프로세스를 뽑는 과정을 반복한다.

3.3.5 테스트 실행 결과

1) test-currency.sh

테스트 스크립트의 두 테스트를 통해서 프로세스의 UID 변경이 정상적으로 수행됐고, getpinfo를 통해서 변경된 UID가 올바르게 조회됨을 확인했다. 추가로 프로세스의 티켓 수 변경이 정상적으로 적용 됐음을 확인했다.

[test-currency.sh 테스트 결과]

```
enteryhj@csapp:~/test3/initial-xv6$ ./test-currency.sh
=====  Initializing Test Environment =====
Cleaning xv6-public...
Generating temporary Makefile (xv6-public/Makefile.test)...
Copying object files to xv6-public...
Building xv6 (linking files using Makefile.test)...
Build complete.

===== Running Test: cur2 =====
Executing tests/cur2.run...
Comparing results...
test cur2: passed

===== Running Test: cur3 =====
Executing tests/cur3.run...
Comparing results...
test cur3: passed
```

2) test_currency

[test_currency 테스트 결과]

```
$ test_currency
Starting currency scheduler test (NUID=10, CAP=1000)...
Processes created. Measuring ticks for 10 seconds...
---------------------------------
--- Test Results ---
User 1 (Demand: 200)  ran for: 121 ticks
User 2 (Demand: 500) ran for: 282 ticks
User 3 (Demand: 1500) ran for: 597 ticks
---------------------------------

Total ticks (User 1,2,3): 1000
Expected Ratio (200:500:1000) => 11% : 29% : 58%
Actual Ratio   (Actual Ticks)  => 12% : 28% : 59%
```

```
$ test_currency
Starting currency scheduler test (NUID=10, CAP=1000)...
Processes created. Measuring ticks for 10 seconds...
---------------------------------
--- Test Results ---
User 1 (Demand: 200)  ran for: 122 ticks
User 2 (Demand: 500) ran for: 281 ticks
User 3 (Demand: 1500) ran for: 599 ticks
---------------------------------

Total ticks (User 1,2,3): 1002
Expected Ratio (200:500:1000) => 11% : 29% : 58%
Actual Ratio   (Actual Ticks)  => 12% : 28% : 59%
```

```
$ test_currency
Starting currency scheduler test (NUID=10, CAP=1000)...
Processes created. Measuring ticks for 10 seconds...
---------------------------------
--- Test Results ---
User 1 (Demand: 200)  ran for: 121 ticks
User 2 (Demand: 500) ran for: 279 ticks
User 3 (Demand: 1500) ran for: 601 ticks
---------------------------------

Total ticks (User 1,2,3): 1001
Expected Ratio (200:500:1000) => 11% : 29% : 58%
Actual Ratio   (Actual Ticks)  => 12% : 27% : 60%
```

```
$ test_currency
Starting currency scheduler test (NUID=10, CAP=1000)...
Processes created. Measuring ticks for 10 seconds...
---------------------------------
--- Test Results ---
User 1 (Demand: 200)  ran for: 121 ticks
User 2 (Demand: 500) ran for: 281 ticks
User 3 (Demand: 1500) ran for: 601 ticks
---------------------------------

Total ticks (User 1,2,3): 1003
Expected Ratio (200:500:1000) => 11% : 29% : 58%
Actual Ratio   (Actual Ticks)  => 12% : 28% : 59%
```

```
$ test_currency
Starting currency scheduler test (NUID=10, CAP=1000)...
Processes created. Measuring ticks for 10 seconds...
---------------------------------
--- Test Results ---
User 1 (Demand: 200)  ran for: 121 ticks
User 2 (Demand: 500) ran for: 278 ticks
User 3 (Demand: 1500) ran for: 601 ticks
---------------------------------

Total ticks (User 1,2,3): 1000
Expected Ratio (200:500:1000) => 11% : 29% : 58%
Actual Ratio   (Actual Ticks)  => 12% : 27% : 60%
```

```
$ test_currency
Starting currency scheduler test (NUID=10, CAP=1000)...
Processes created. Measuring ticks for 10 seconds...
---------------------------------
--- Test Results ---
User 1 (Demand: 200)  ran for: 122 ticks
User 2 (Demand: 500) ran for: 280 ticks
User 3 (Demand: 1500) ran for: 599 ticks
---------------------------------

Total ticks (User 1,2,3): 1001
Expected Ratio (200:500:1000) => 11% : 29% : 58%
Actual Ratio   (Actual Ticks)  => 12% : 27% : 59%
```

위의 6번 반복 실행한 결과를 표로 정리한 결과는 아래와 같다.

Total ticks	User1	User2	User3
1000	12% (121)	28% (282)	59%(597)
1002	12% (122)	28% (281)	59% (599)
1001	12% (121)	27% (279)	60% (601)
1003	12% (121)	28% (281)	59% (601)
1000	12% (121)	27% (278)	60% (601)
1001	12% (122)	27% (280)	59% (599)

user1은 12%, user2는 27%~28%, user3은 59%~60%로 나온다

3.3.6 결론

실행할 때마다 ticks의 수치는 미세하게 바뀐다. 하지만 정말 미세한 변화이기 때문에 비율은 거의 일정하게 유지된다. 그리고 그 비율은 이론적인 기대치인 11%, 29%, 58%에 매우 근접한다.

특히 주목해서 봐야할 부분은 user3이다. 1500장을 요구했지만 이 값은 1000장 제한을 넘어선 값이기 때문에 티켓이 1000장으로 설정된다. 만약 1000장 제한이 적용되지 않았을 경우에는 1500/2200, 약 68%를 차지했을 것이다. 하지만 1000장 제한이 반복 실험 동안 잘 적용되었기 때문에 68%에 가까운 수치는 나오지 않았고, 1000/1700, 약 59%에 근접한 수치가 6번의 반복적인 실행 동안 안정적으로 나오게 된다.

4. 역할 분담

공통 : 개인별 학습을 위해 각자 실습 진행했다. 그 과정에서 학습한 내용과 진행 상황을 공유해가며, 시스템 콜 연결 및 구현과 스케줄링 알고리즘 구현을 함께 했다.

유현종 : Phase 3: 2-Level Lottery Scheduler 알고리즘 분석에 도움을 주었다.

안준영 : Phase 2: Lottery Scheduler 알고리즘 분석에 도움을 주었다.