

Name: Chenyang Yu **SUID:** 862052273
Source: <https://github.com/mit-pdos/xv6-public.git>
Environment: Ubuntu 16.04.3 LTS
Command:

```
$make qemu-nox-gdb // $make qemu-nox for not waiting gdb connecting
( change CPU to 1 in makefile)
In another terminal
$gdb -q (c to continue)
$killall qemu-system-i386
```

How to add a system call:

In defs.h:

Add the declaration of system call

```
void            yield(void);
// cs202
void            info(int param);
void            hello(void);
void            setticket(int ticket);
```

In sysproc.c

The implement of system call method (use argint() to pass argument)

```
// cs202
int sys_info()
{
    int param;
    if(argint(0, &param) < 0)
        return -1;
    else
        info(param);
    return 0; //shouldn't be here
}

int sys_hello(void)
{
    hello();
    return 0;
}

int sys_setticket()
{
    int ticket;
    if(argint(0, &ticket) < 0)
        return -1;
    else
        setticket(ticket);
    return 0; //shouldn't be here
}
```

In syscall.c/ syscall.h

Extern define the function that connect the shell and the kernel (the position is defined in the syscall.h). We use the position 22,23,24 to add the function call in the system call vector

```
//cs202
extern int sys_info(void);
extern int sys_hello(void);
extern int sys_setticket(void);

// cs202
#define SYS_info    22
#define SYS_hello   23
#define SYS_setticket 24
```

In user.h

Define the function that can be called through the shell

```
//cs202
int info(int param);
void hello(void);
int setticket(int ticket);
```

In usys.S

Define the connect the call of user to the system call function

```
SYSCALL(info)
SYSCALL(hello)
SYSCALL(setticket)
```

info(int param);

In proc.h: The declaration of num_syscall for recording number of system call, initial it in allocproc(). The function is in proc.c

```
// cs202
int numsyscall; // the number of syscall
```

found:

```
p->state = EMBRYO;
p->pid = nextpid++;
// cs202
// initial number of syscall
p->numsyscall = 0;
```

setticket(int ticket);

In proc.h

The declaration of ticket (given by system call), stride (by evaluate), accumulate stride (increase when running), sche_cnt (increase when running). All initial in allocproc()

```
int ticket; // the number of ticket
int stride; // the number of stride = ticket/ all_ticket
int include_all; // has this process included into pool? initial:0
int acc_stride; // accumulate stride, used for comparison
int sche_cnt; // how many times does this process execute?

#if STRIDE
p->ticket = 0; // the number of ticket
p->stride = 0; // the number of stride = ticket/ all_ticket
p->include_all = 0; // has this process included into pool? initial:0
p->acc_stride = 0; // accumulate stride, used for comparison
p->sche_cnt = 0; // how many times does this process execute?
#endif
```

The function in proc.c. Set the number of ticket for process and evaluate the stride.

```
void setticket(int ticket){
    struct proc *p = myproc();
    p->ticket = ticket;
    //all_ticket += ticket;
    p->acc_stride = ticket/all_ticket;
    cprintf("ticket: %d\n",ticket);
    Lflag = 1;
    return;
}
```

Add testing file:

Test.c for part one and prog1.c/prog2.c/prog3.c for part two. Modify Makefile to include these file.

Test: Different system call according to parameter. We call 5 hello() system call to see if the number of system call increases or not.

```

param = atoi(argv[1]);
if(param ==1){
    info(1);
}else if(param ==2){
    info(2);
    for(int i =0;i<5 ;i++)
        hello();
    info(2);
}else if(param ==3){
    info(3);
}else{
    printf(1, "unexpect input!! usage: test [1|2|3]\n");
}

```

Prog: Add setticket() before main loop, different ticket number for different program

```

//FUNCTION_SETS_NUMBER_OF_TICKETS(30);
setticket(100);
int i,k;
const int loop=43000;
for(i=0;i<loop;i++){
    asm("nop"); //in order to preven
    for(k=0;k<loop;k++){
        asm("nop");
    }
}
exit();

```

Part1:

(1) Print out the number of process by calling procdump_cnt(). Go through the process table and record the process whose state is not "UNUSED"

```

void procdump_count(void){
    static char *states[] = {
        [UNUSED]    "unused",
        [EMBRYO]    "embryo",
        [SLEEPING]  "sleep ",
        [RUNNABLE]  "runble",
        [RUNNING]   "run   ",
        [ZOMBIE]    "zombie"
    };
    struct proc *p;
    char *state;
    int count = 0;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != UNUSED){
            count++;
            if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
                state = states[p->state];
            else
                state = "???";
            cprintf("%d %s %s\n", p->pid, state, p->name);
        }
    }
    cprintf("====num of process(not unused): %d====\n",count);
}

```

(2) The number of system call is initialize as zero. Add a counter in syscall.c

```

num = curproc->tf->eax;
if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
    // cs202
    curproc->numsyscall++; // only increase when syscall is found and done
} else {
    cprintf("%d %s: unknown sys call %d\n",
        curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
}

```

(3) Parameter two return memory size (store in process structure, divide it with page size: 4KB)

```

} else if (param == 3) {
    struct proc *p = myproc();
    cprintf("====the process size is %d====\n", p->sz); // Size of process memory (bytes)
    // xv6 uses a page size of 4KB
    // if there is remain, means p use another page
    if ((p->sz) % PGSIZE == 0)
        cprintf("page size: %d\n", (p->sz) / PGSIZE);
    else
        cprintf("page size: %d\n", (p->sz) / PGSIZE + 1);
}

```

Result:

Param one return the number of process. Param two output system call number, which different after we call 5 hello() system call. Param three return the page size this process used.

```

$ test 1
1 sleep init
2 sleep sh
10 run test
====num of process(not unused): 3====
$ test 2
====number of system calls: 2====
hello 0
hello 0
hello 0
hello 0
hello 0
====number of system calls: 8====
$ test 3
====the process size is 12288====
page size: 3
$

```

Part2:

We rewrite the scheduler to implement stride, using a compile flag to separate from original code.

```

// cs202
// #define STRIDE 0 to use default scheduler
#define STRIDE 1
int all_ticket = 10000;

```

In our scheduler, we use one loop to find the lowest acc_stride, this is the process we choose to run next

```

int min_stride = 1000000; // used for finding minimal stride
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->state != RUNNABLE)
        continue;
    if (p->acc_stride < min_stride) {
        min_stride = p->acc_stride;
    }
}

```

The second loop used for finding the process with the lowest acc_stride and output the detail of this process (the ticket it hold, the stride it hold, the current stride it is, and the running time)


```

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE || p->acc_stride != min_stride)
        continue;
    // Switch to chosen process. It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    c->proc = p;
    switchvm(p);
    p->state = RUNNING;
    // cs202 p with the lowest acc_stride and choose to run
    if(p->ticket != 0){
        p->acc_stride += all_ticket/(p->ticket);
        cprintf("ticket[%d]=stride[%d], currentStride[%d], runtime[%d]\n", p->ticket, p->acc_stride, p->currentStride, p->runtime);
        p->sche_cnt++;
    }

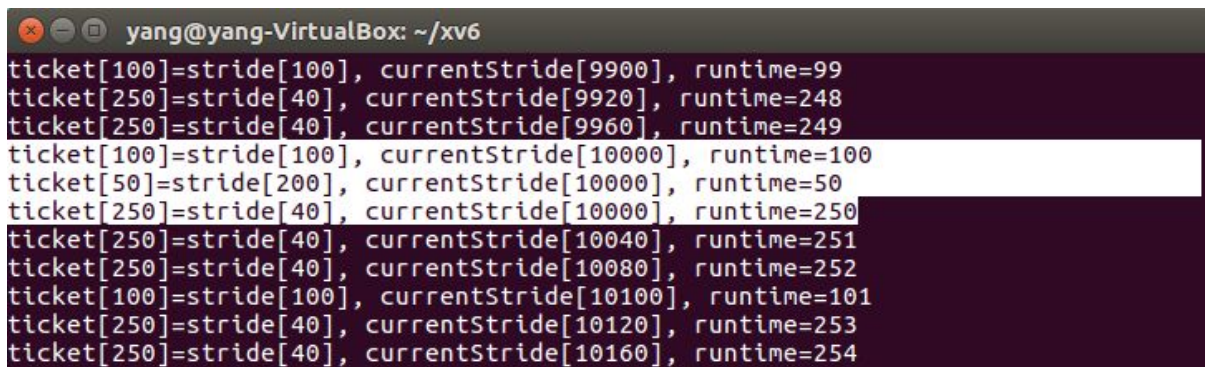
    swtch(&(c->scheduler), p->context);
    switchkvm();

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0;
}

```

Result:

The running time is proportion to the number of tickets. The running time ratio between ticket 100:50:250 is 100:50:250.



```

yang@yang-VirtualBox: ~/xv6
ticket[100]=stride[100], currentStride[9900], runtime=99
ticket[250]=stride[40], currentStride[9920], runtime=248
ticket[250]=stride[40], currentStride[9960], runtime=249
ticket[100]=stride[100], currentStride[10000], runtime=100
ticket[50]=stride[200], currentStride[10000], runtime=50
ticket[250]=stride[40], currentStride[10000], runtime=250
ticket[250]=stride[40], currentStride[10040], runtime=251
ticket[250]=stride[40], currentStride[10080], runtime=252
ticket[100]=stride[100], currentStride[10100], runtime=101
ticket[250]=stride[40], currentStride[10120], runtime=253
ticket[250]=stride[40], currentStride[10160], runtime=254

```

Further comparison:

We add start time and end time to measure turnaround time. Add running time (only increase when the process is chosen in scheduler). Add response time (Calculate the time between start and first time in scheduler). The declaration is in proc.h and initialization is in proc.c.

```

int acc_stride;           // accumulate stride, used for comparison
int sche_cnt;             // how many times does this process execute?
int start_time;
int end_time;
int turnaround_time;
int running_time;         // waiting time = turnaround time - running time
int response_time;

```

Output the result in exit()

```

if(curproc->ticket > 0){
    curproc->end_time = ticks;
    curproc->turnaround_time = curproc->end_time - curproc->start_time;
    cprintf("ticket[%d]:start_time:%d,end_time:%d,turnaround_time:%d,run_time:%d,
response_time:%d\n",curproc->ticket,curproc->start_time,curproc->end_time,curproc-
>turnaround_time,curproc->running_time,curproc->response_time);
}
sched();

```

Result:

Three process starts at approximately same time. Since the context of program is the same, all three process have approximately same running time. The response time is little shorter when holding more ticket, show that the stride scheduling also affect on response time.

```
$ ticket[250]out, turnover:542
ticket[250]:start_time:373,end_time:1248,turnaround_time:875,run_time:544, response_time:6
zombie!

$ ticket[100]out, turnover:543
ticket[100]:start_time:362,end_time:1736,turnaround_time:1374,run_time:543, response_time:10
zombie!

$
$ ticket[50]out, turnover:558
ticket[50]:start_time:365,end_time:2024,turnaround_time:1659,run_time:559, response_time:12
zombie!
```

Reference:

[1] Environment setup from UCR CS153

<http://www.cs.ucr.edu/~nael/cs153/labs.html>