Part 1: System Call Implementation:

Implementation:

1) proc.c :- The system call is implemented in this file.

   As shown in the figure when the param is 1 then the ptable is iterated to count the number of running processes and to only count the running processes state of the process is checked.
   For param 2 the syscall_count is fetched from the TCB of the calling process as this variable in the TCB keeps the number of system call made by the process.
   For param 3 the size of the program is divided by the 4KB which is the page size the sz member of the TCB contains the size of the program.

```
int info (int param)
{

struct proc *c_proc = myproc();
if (param == 1)
 {
        struct proc *p;
        int count_p = 0;
          acquire(&ptable.lock);
          for (p = ptable.proc; p<&ptable.proc[NPROC]; p++)
                {
                        if (p->state != UNUSED)
                                count_p = count_p + 1;
                }
           release(&ptable.lock);

                        cprintf("Total Number of running processes: %d\n", count_p);
 }
else if (param== 2)
cprintf("Total Number of system calls made by process: %d\n",c_proc->syscall_count);
else if (param== 3){
        if((c_proc->sz)%PGSIZE == 0){
          int number = (c_proc->sz)/PGSIZE;
          cprintf("Number of pages used :%d\n",number);
        }
        else{
          int number1 = ((c_proc->sz)/PGSIZE) + 1 ;
          cprintf("Number of pages used:%d\n",number1);
        }
}
return 0;
}
```

2) sysproc.c :- The system call input arguments are checked here and function prototype is also added

```
int
sys_info(void)
{
int param;
argint(0, &param);

if(param < 1 && param > 3)
 return -1;

info(param);
return 0;

}
```

3) proc.h :- To count the number of system call made by the program a variable (member in the struct proc) named syscall_call has been added here in the structure proc which is the task control block for each process.

```
// 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 syscall_count;
```

4) syscall.c :- When a system call is made by any process a function is called in syscall.c the name of the function is syscall(void), so the syscall_count is increased every time the program calls a system call.

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {

    curproc->syscall_count++;

    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

5) usys.S : An interface for the system call is defined here so that the user can use it.
6) syscall.h : System call number is added here i.e. 22.
7) syscall.c : Function is defined here and function pointer to the systcall function is also added here.
8) defs.h : The figure shows the changes made in the defs.h file.
9) user.h : The function called by the user is defined here.

The output of the system call is shown in the figure below:
(1) A count of the processes in the system.
(2) A count of the total number of system calls that a process has done so far.
(3) The number of memory pages the current process is using.

```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test
Total Number of running processes: 3
Total Number of system calls made by process: 4
Number of pages used :3
```

Part 2: Modifying xv6 scheduler:

In order to implement stride schedular two system call named **assigntickets and countticks** were implemented.
Assigntickets system call is used to assign tickets to the process
The files changed to implement assign tickets

1) proc.c :- The system call implementation is done here. The tickets to be assigned are assigned to the appropriate member of the task control block struct of the calling process. Stride is also calculated here and assigned to the task control block member. The pass value will be same as the stride.

```
int assigntickets(int tickets)
{
        struct proc *proc = myproc();

        proc->tickets = proc->tickets+tickets;
        proc->stride = (max_tkt/tickets);
        proc->pass = proc->stride;


cprintf("Number of tickets and stride of process: %s = %d , %d\n", proc->name, proc->tickets,proc->stride);
return 0;

}
```

2) sysproc.c :- The system call input arguments are checked here and function prototype is also added

```
int
sys_assigntickets(void)
{
        int tickets;
        argint(0, &tickets);

        if(tickets<0)
        return -1;

        assigntickets(tickets);

        return 0;
 }
```

3) proc.h :- In the TCB struct member such as stride, pass and tickets are added.

```
// 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 syscall_count;
  int stride;
  int pass;
  int tickets;
  int counter;
```

4) usys.S : An interface for the system call is defined here so that the user can use it.
5) syscall.h : System call number is added here i.e. 23.
6) syscall.c : Function is defined here and function pointer to the systcall function is also added here.
7) defs.h : The figure shows the changes made in the defs.h file.
8) user.h : The function called by the user is defined here.

Countticks system call is used to count the number of the ticks for a certain process.

1) sysproc.c:- The calling process counter value from the TCB is fetched which is the variable which keep track of the ticks for a process.

```
int
sys_countticks(void)
{
  return myproc()->counter;
}
```

2) proc.h :- In the TCB struct member such as counter is added to keep the track of the ticks are added.
3) usys.S : An interface for the system call is defined here so that the user can use it.
4) syscall.h : System call number is added here i.e. 23.
5) syscall.c : Function is defined here and function pointer to the systcall function is also added here.
6) defs.h : The figure shows the changes made in the defs.h file.
7) user.h : The function called by the user is defined here

Implementation of Stride schedular

1) The implementation of the schedular is shown in the figure below.

```
void
scheduler(void)
{
  struct proc *p;
  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);

    int min_pass = 10000000;  // used to find process with minimum pass

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  // Loop to find the process with the minimum pass
      if(p->state != RUNNABLE){
          continue;
      }
      if( p->pass < min_pass){
        min_pass = p->pass;
      }
    }


    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

      if(p->state != RUNNABLE || p->pass != min_pass){  // only allow the process which have the minimum number of pass
          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;
      switchuvm(p);
      p->state = RUNNING;

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

    if(p->tickets != 0){                     // allow the process which have been assigned tickets to increase their passes by the value of the stride
          p->pass = p->pass + p->stride;
          p->counter++;     // increase the count to indicate that the process ran
      }
      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;

      }
    release(&ptable.lock);

   }
}
```

Arrow 1

Arrow 2

Arrow 3

The schedular algorithm first searches for the process with the minimum number of the passes (**Arrow** 1) and then only allow that process two run not allowing any other process to run (**Arrow** 2).

At the end the count value of the process which ran by 1 and the pass value is increased by stride of that process.(**Arrow** 3).

2) In proc.c file the value of tickets, stride, pass, count and syscall_count are initialized to 0 in the allocproc() function.

When the prog1 prog2 and prog3 are allowed to run with the ticket values of 30, 20, 10 the result is shown below.

```
$ prog1&;prog2&;prog3&
Number of tickets and stride of process: prog1 = 30 , 333
Number of tickets and stride of process: prog2 = 20 , 500
Number of tickets and stride of process: prog3 = 10 , 1000
$ Prog1 : 715
zombie!
Prog3 : 457
zombie!
Prog2 : 1198
```

The resource allocation of prog1, prog2 and prog3 are approximately in the ratio of ½ , 1/3, and 1/6.

When the prog1 prog2 and prog3 are allowed to run with the ticket values of 100, 50, 250 the result is shown below. The strides are also as follow 100, 200, 40. There was a message “zombie!” it was unknown as there were no child process.

```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ prog1&;prog2&;prog3&
Number of tickets and stride of process: prog1 = 100 , 100
$ Number of tickets and stride of process: prog2 = 50 , 200
Number of tickets and stride of process: prog3 = 250 , 40
Prog1 : 381
zombie!
Prog3 : 931
zombie!
Prog2 : 1011
```