Lab 2 - Report

Karthik Harpanahalli - 862254197
Varun Sapre - 862255166

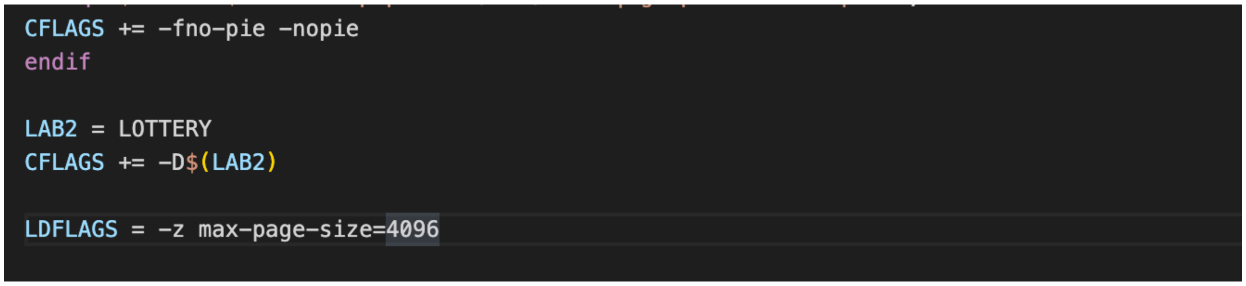
a) List of all files modified:

10 files were modified in total

1. Makefile
2. kernel/defs.h
3. kernel/proc.c
4. kernel/proc.h
5. kernel/syscall.c
6. kernel/syscall.h
7. kernel/sysproc.c
8. user/lab2.c
9. user/user.h
10. user/usys.pl

b) A detailed explanation on what changes you have made and screenshots showing your work and results

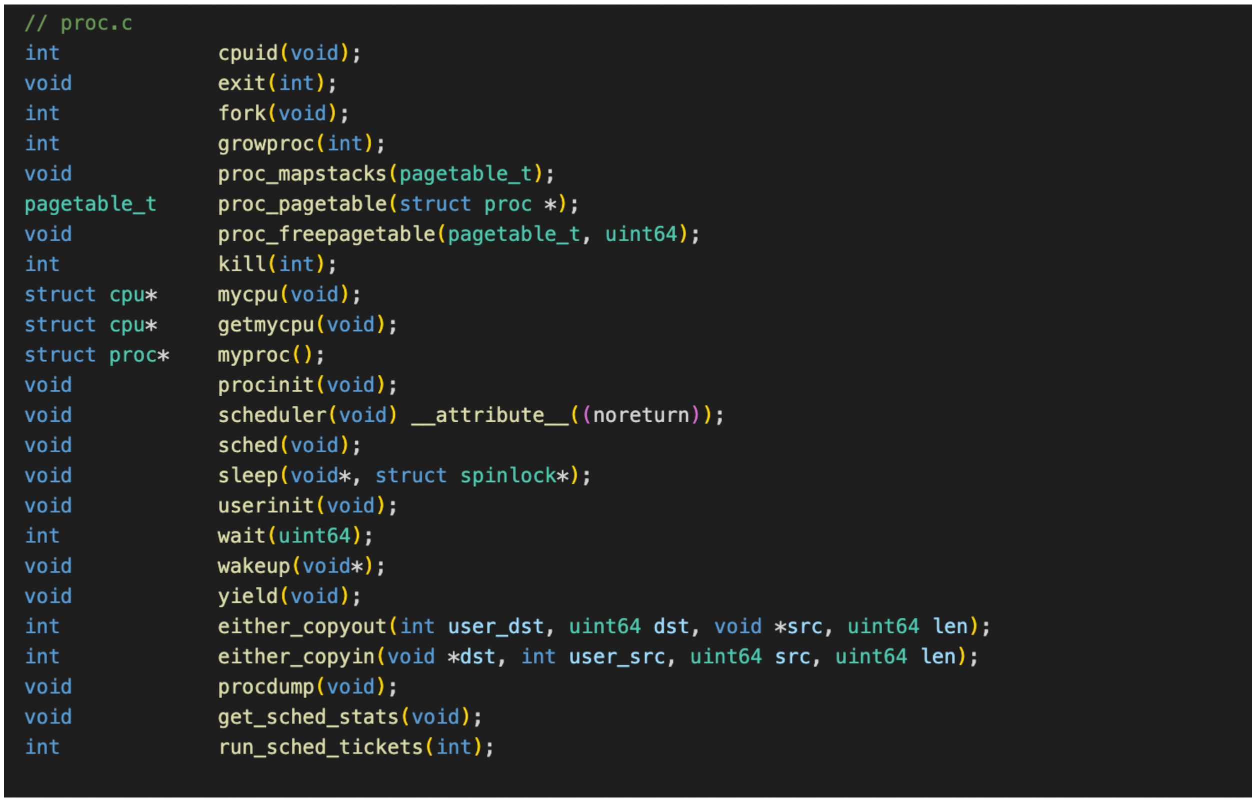
1) Makefile



```
CFLAGS += -fno-pie -nopie
endif

LAB2 = LOTTERY
CFLAGS += -D$(LAB2)

LDFLAGS = -z max-page-size=4096
```

We add the lines necessary for the toggle switch for both stride and lottery scheduling.

2) kernel/defs.h

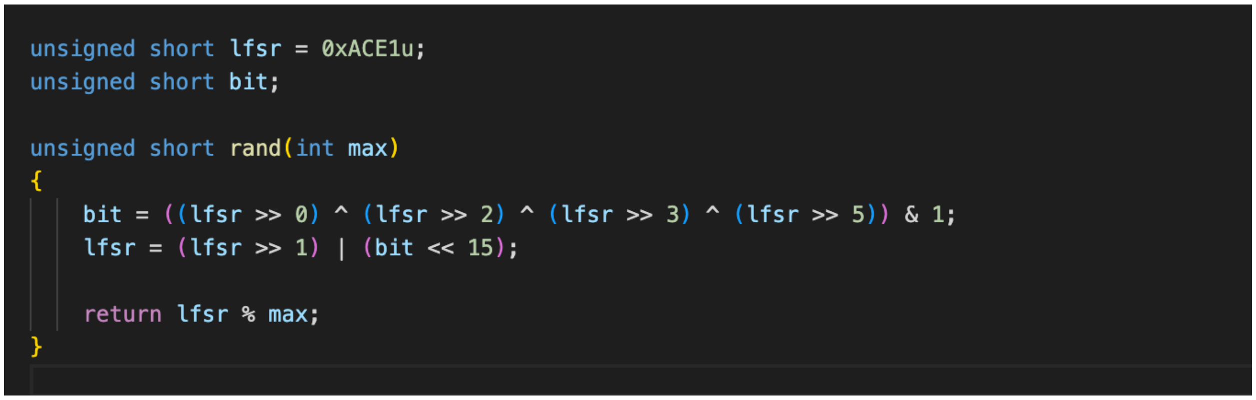


```
// proc.c
int             cpuid(void);
void            exit(int);
int             fork(void);
int             growproc(int);
void            proc_mapstacks(pagetable_t);
pagetable_t     proc_pagetable(struct proc *);
void            proc_freepagetable(pagetable_t, uint64);
int             kill(int);
struct cpu*     mycpu(void);
struct cpu*     getmycpu(void);
struct proc*    myproc();
void            procinit(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(uint64);
void            wakeup(void*);
void            yield(void);
int             either_copyout(int user_dst, uint64 dst, void *src, uint64 len);
int             either_copyin(void *dst, int user_src, uint64 src, uint64 len);
void            procdump(void);
void            get_sched_stats(void);
int             run_sched_tickets(int);
```

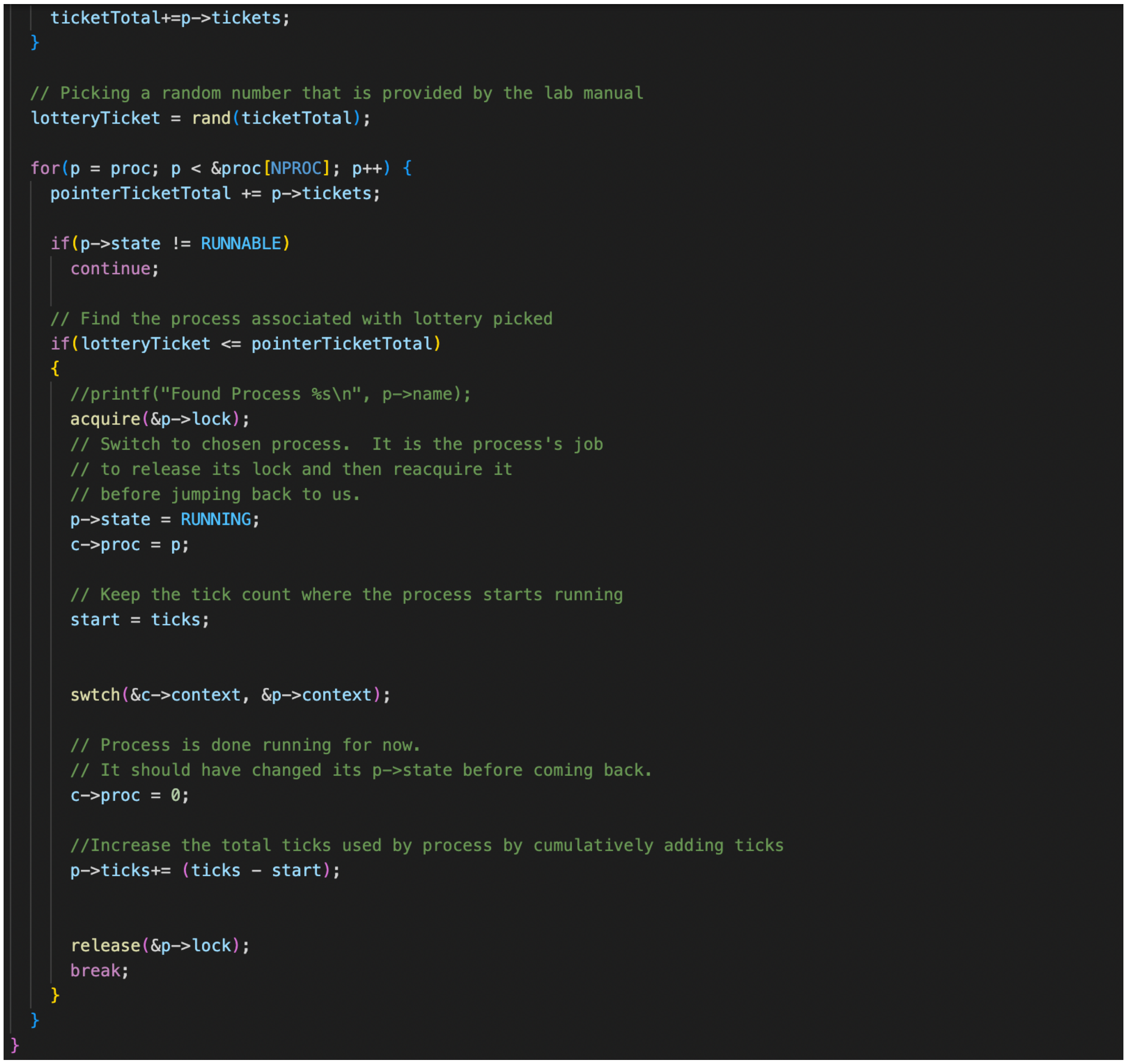
We add our procedures in the defs.h file to make them accessible. We define three methods to achieve each required functionality.

1. **get_sched_stats(void):** parses all the non-unused processes and prints the process stats in the required format.
2. **run_sched_tickets(int tickets):** checks the total number of tickets assigned in the system and if its less than 5000, it will assign the specified tickets to the current proc.

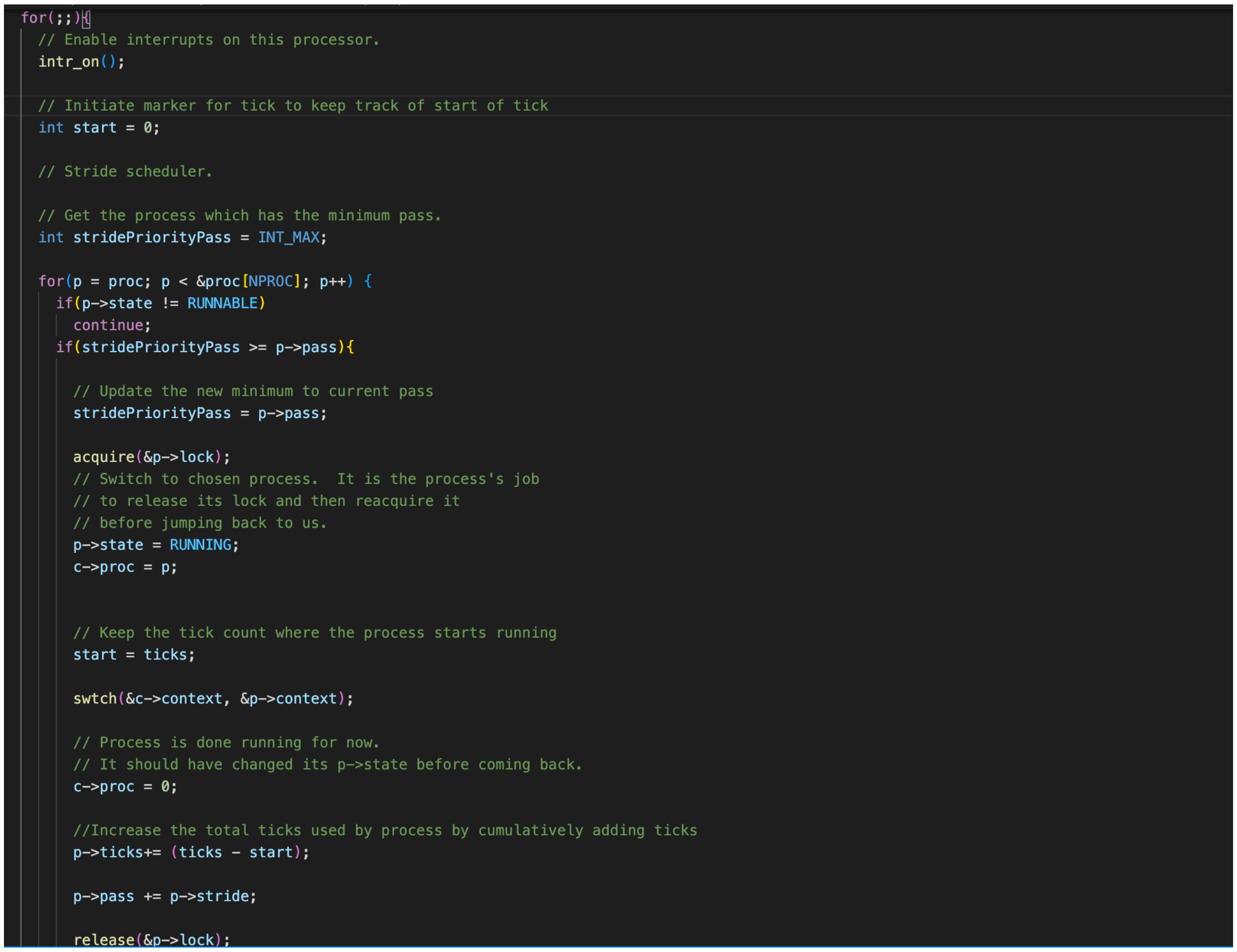
3) kernel/proc.c



```c
unsigned short lfsr = 0xACE1u;
unsigned short bit;

unsigned short rand(int max)
{
    bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5)) & 1;
    lfsr = (lfsr >> 1) | (bit << 15);

    return lfsr % max;
}
```

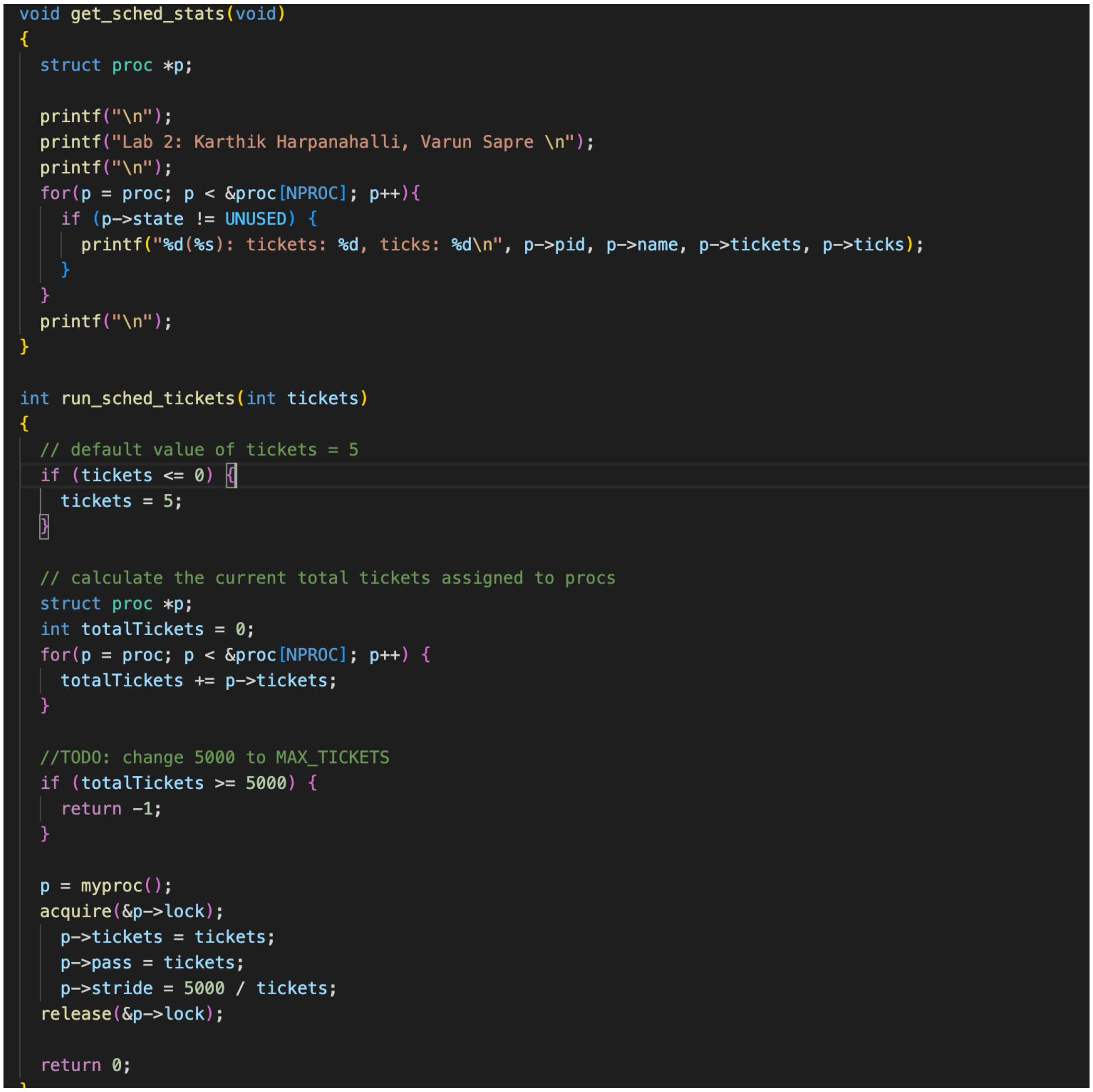


```c
    ticketTotal+=p->tickets;
  }

  // Picking a random number that is provided by the lab manual
  lotteryTicket = rand(ticketTotal);

  for(p = proc; p < &proc[NPROC]; p++) {
    pointerTicketTotal += p->tickets;

    if(p->state != RUNNABLE)
      continue;

    // Find the process associated with lottery picked
    if(lotteryTicket <= pointerTicketTotal)
    {
      //printf("Found Process %s\n", p->name);
      acquire(&p->lock);
      // Switch to chosen process.  It is the process's job
      // to release its lock and then reacquire it
      // before jumping back to us.
      p->state = RUNNING;
      c->proc = p;

      // Keep the tick count where the process starts running
      start = ticks;


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

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

      //Increase the total ticks used by process by cumulatively adding ticks
      p->ticks+= (ticks - start);


      release(&p->lock);
      break;
    }
  }
}
```



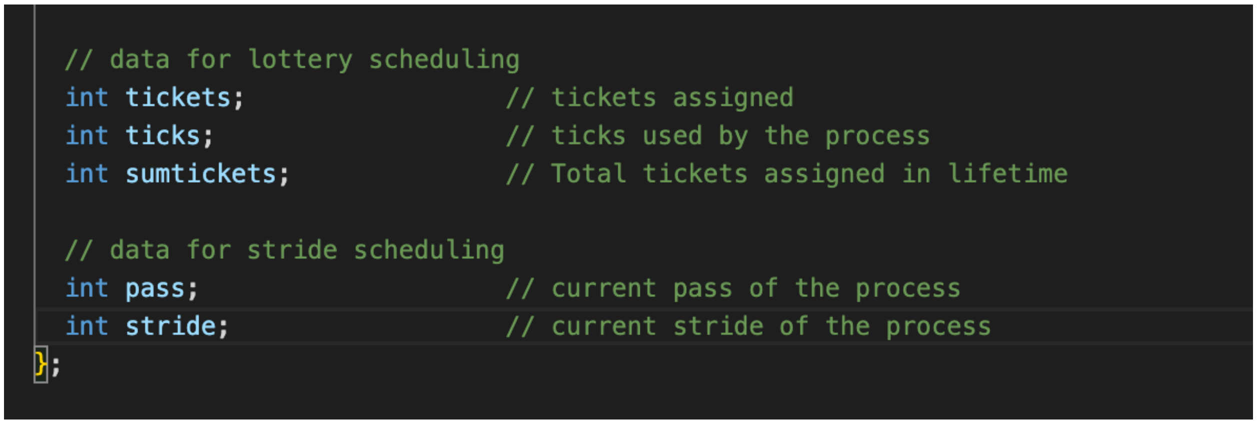
```
for(;;){
  // Enable interrupts on this processor.
  intr_on();

  // Initiate marker for tick to keep track of start of tick
  int start = 0;

  // Stride scheduler.

  // Get the process which has the minimum pass.
  int stridePriorityPass = INT_MAX;

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

      // Update the new minimum to current pass
      stridePriorityPass = p->pass;

      acquire(&p->lock);
      // Switch to chosen process.  It is the process's job
      // to release its lock and then reacquire it
      // before jumping back to us.
      p->state = RUNNING;
      c->proc = p;


      // Keep the tick count where the process starts running
      start = ticks;

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

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

      //Increase the total ticks used by process by cumulatively adding ticks
      p->ticks+= (ticks - start);

      p->pass += p->stride;

      release(&p->lock);
```

This is the implementation of the logic that achieves the above mentioned functionality. Lottery scheduling and stride scheduling have been explained in a detailed manner further below.



```c
void get_sched_stats(void)
{
  struct proc *p;

  printf("\n");
  printf("Lab 2: Karthik Harpanahalli, Varun Sapre \n");
  printf("\n");
  for(p = proc; p < &proc[NPROC]; p++){
    if (p->state != UNUSED) {
      printf("%d(%s): tickets: %d, ticks: %d\n", p->pid, p->name, p->tickets, p->ticks);
    }
  }
  printf("\n");
}

int run_sched_tickets(int tickets)
{
  // default value of tickets = 5
  if (tickets <= 0) {
    tickets = 5;
  }

  // calculate the current total tickets assigned to procs
  struct proc *p;
  int totalTickets = 0;
  for(p = proc; p < &proc[NPROC]; p++) {
    totalTickets += p->tickets;
  }

  //TODO: change 5000 to MAX_TICKETS
  if (totalTickets >= 5000) {
    return -1;
  }

  p = myproc();
  acquire(&p->lock);
    p->tickets = tickets;
    p->pass = tickets;
    p->stride = 5000 / tickets;
  release(&p->lock);

  return 0;
```

4) kernel/proc.h

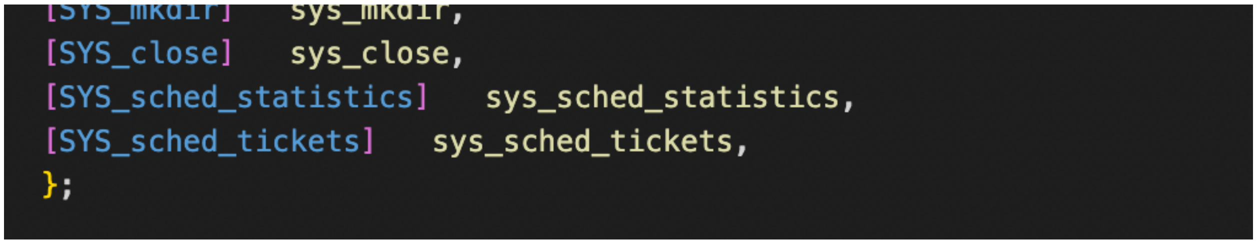


```
  // data for lottery scheduling
  int tickets;                 // tickets assigned
  int ticks;                   // ticks used by the process
  int sumtickets;              // Total tickets assigned in lifetime

  // data for stride scheduling
  int pass;                    // current pass of the process
  int stride;                  // current stride of the process
};
```

Fig: Changes in proc.h file

We are modifying the proc structure to include a variable that holds the count of the tickets, total ticks and sumtickets related to the associated process for lottery scheduling. And for stride scheduling, we have a pass that holds pass value and stride that holds the stride that is computed in the run_sched_tickets(int tickets) function.

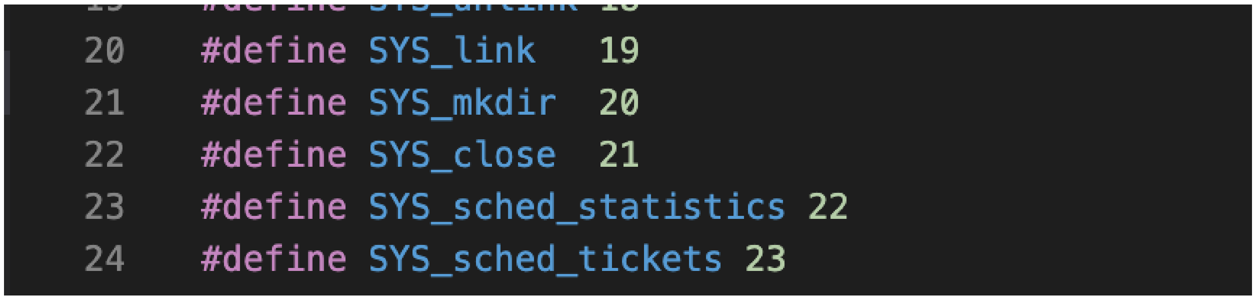
5) kernel/syscall.c



```
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_sched_statistics]   sys_sched_statistics,
[SYS_sched_tickets]   sys_sched_tickets,
};
```

Syscall.c has the implemented syscalls in the array

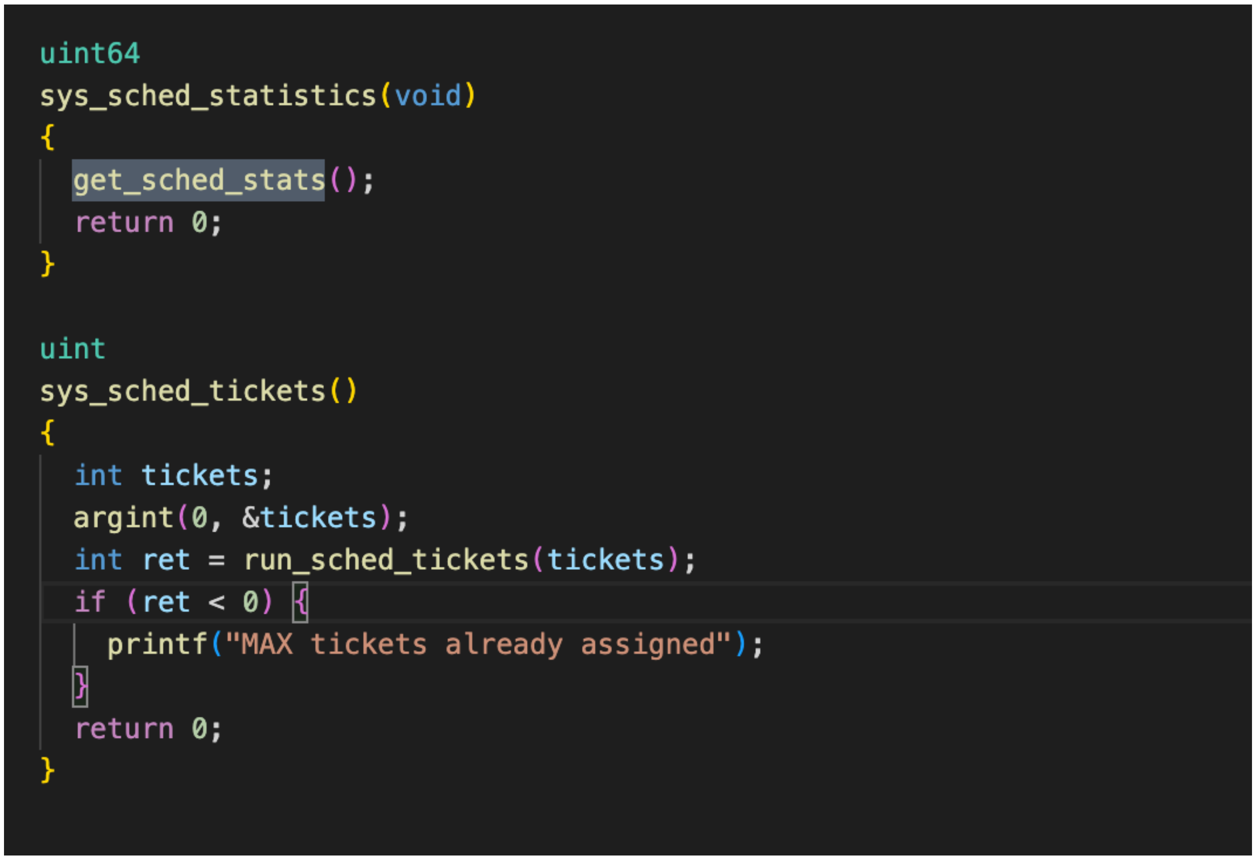
6) kernel/syscall.h



```
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_sched_statistics 22
#define SYS_sched_tickets 23
```

We are defining our syscall as 22 and 23 which is used to write in a7 register to identify the required syscall.

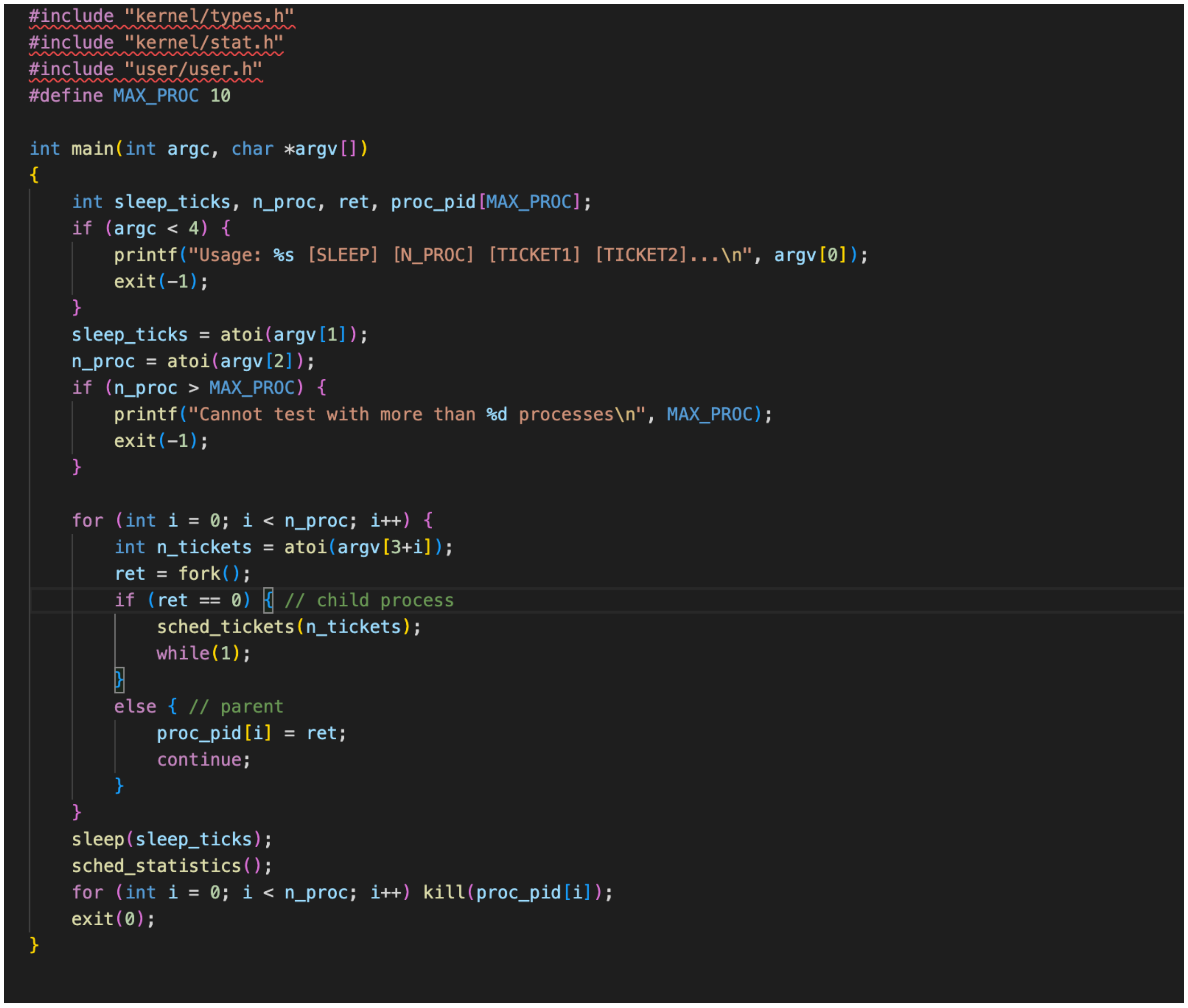
7) kernel/sysproc.c



```c
uint64
sys_sched_statistics(void)
{
  get_sched_stats();
  return 0;
}

uint
sys_sched_tickets()
{
  int tickets;
  argint(0, &tickets);
  int ret = run_sched_tickets(tickets);
  if (ret < 0) {
    printf("MAX tickets already assigned");
  }
  return 0;
}
```

This method handles the actual logic and dispatches the required procedure call written in proc.c by reading the param value. If run_sched_tickets encounters an error where max tickets have already been assigned and cannot assign more.

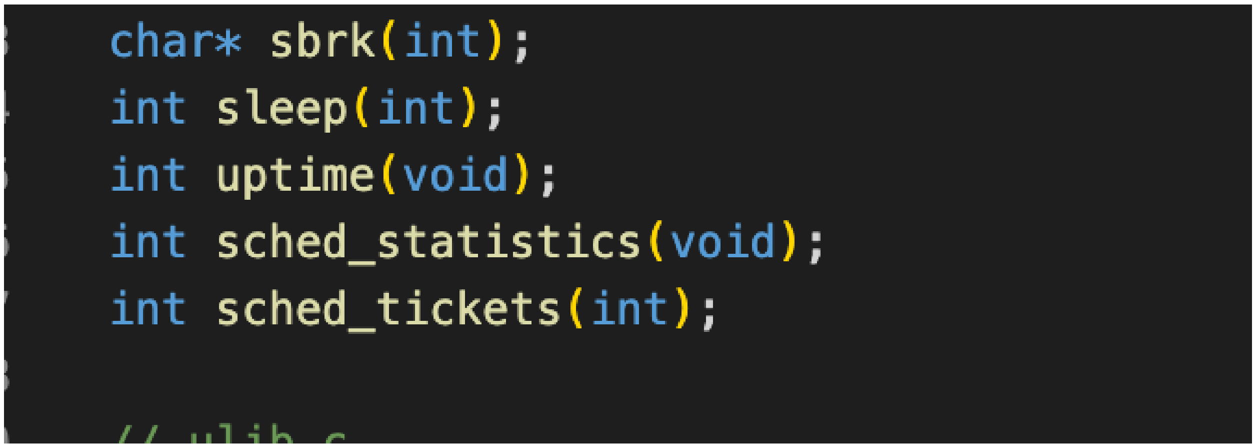
8) user/lab2.c



```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
#define MAX_PROC 10

int main(int argc, char *argv[])
{
    int sleep_ticks, n_proc, ret, proc_pid[MAX_PROC];
    if (argc < 4) {
        printf("Usage: %s [SLEEP] [N_PROC] [TICKET1] [TICKET2]...\n", argv[0]);
        exit(-1);
    }
    sleep_ticks = atoi(argv[1]);
    n_proc = atoi(argv[2]);
    if (n_proc > MAX_PROC) {
        printf("Cannot test with more than %d processes\n", MAX_PROC);
        exit(-1);
    }

    for (int i = 0; i < n_proc; i++) {
        int n_tickets = atoi(argv[3+i]);
        ret = fork();
        if (ret == 0) { // child process
            sched_tickets(n_tickets);
            while(1);
        }
        else { // parent
            proc_pid[i] = ret;
            continue;
        }
    }
    sleep(sleep_ticks);
    sched_statistics();
    for (int i = 0; i < n_proc; i++) kill(proc_pid[i]);
    exit(0);
}
```

This is the user application that is the starting point for our implementation. It calls an appropriate function which dispatches the appropriate syscall.

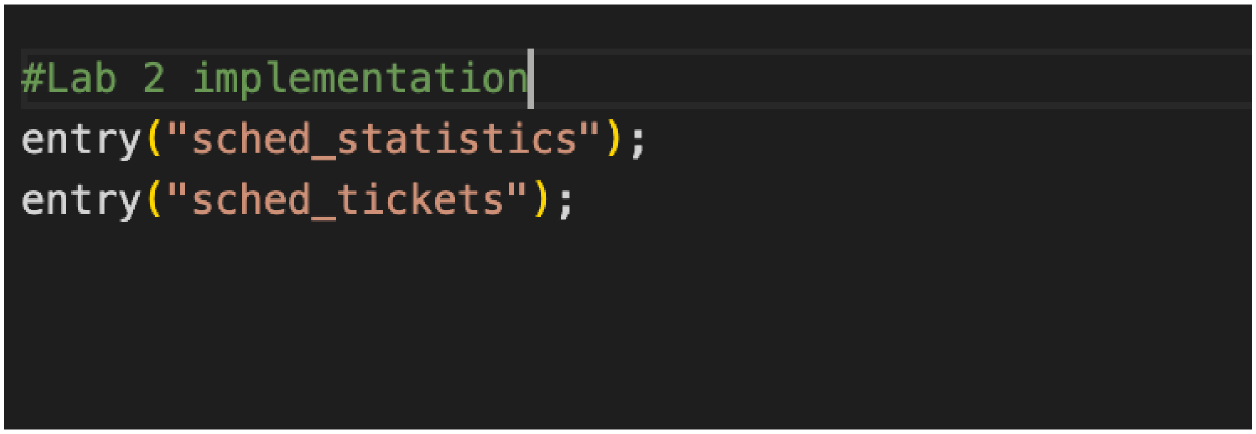
9) user/user.h



```
char* sbrk(int);
int sleep(int);
int uptime(void);
int sched_statistics(void);
int sched_tickets(int);

// ulib.c
```

We define our syscall here to provide all the available syscalls to the user that can be invoked.

10) user/usys.pl



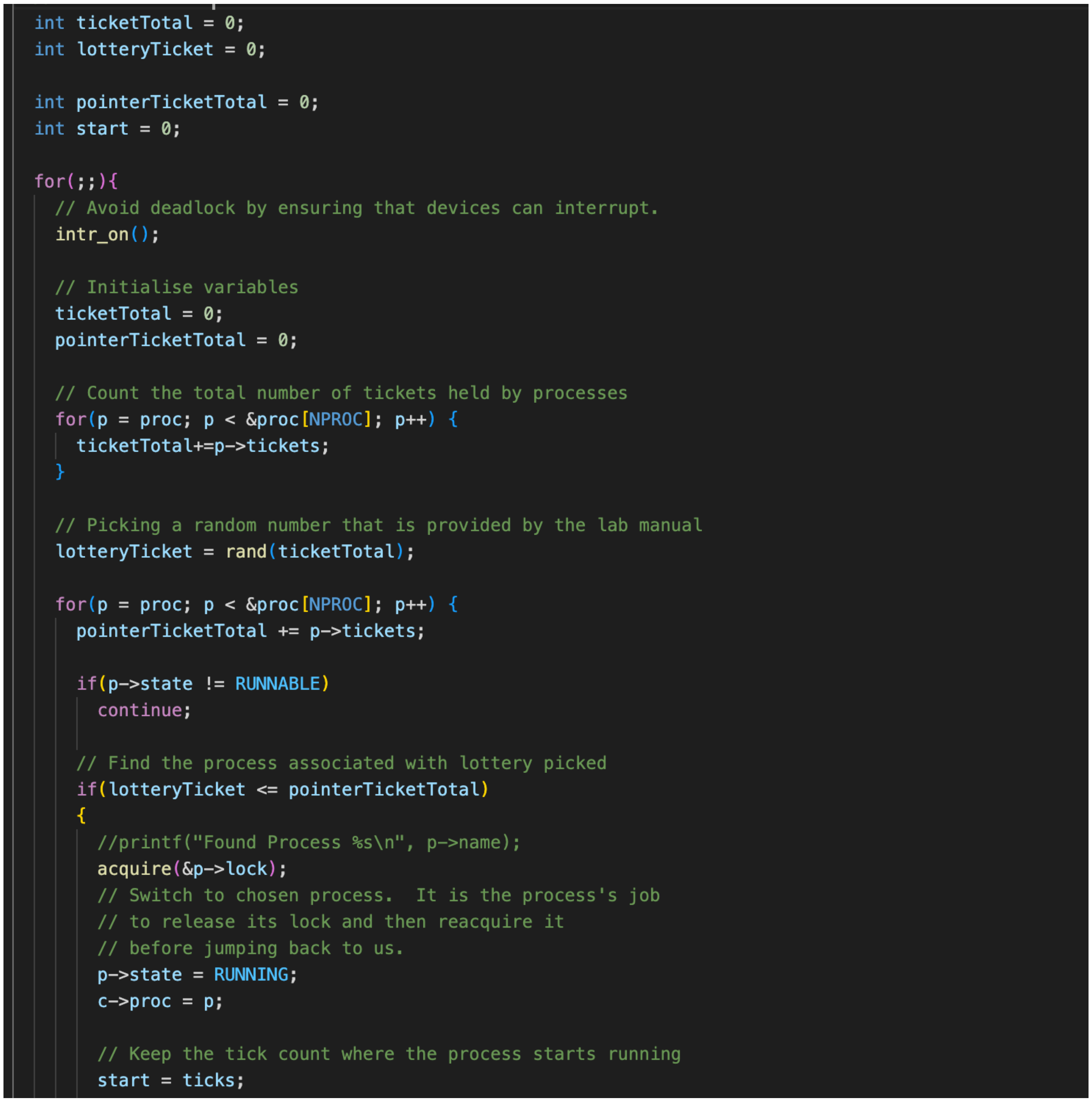
```
#Lab 2 implementation
entry("sched_statistics");
entry("sched_tickets");
```

We register our stub to identify the appropriate syscall to be dispatched from the user space.

Part 2: Lottery Scheduling

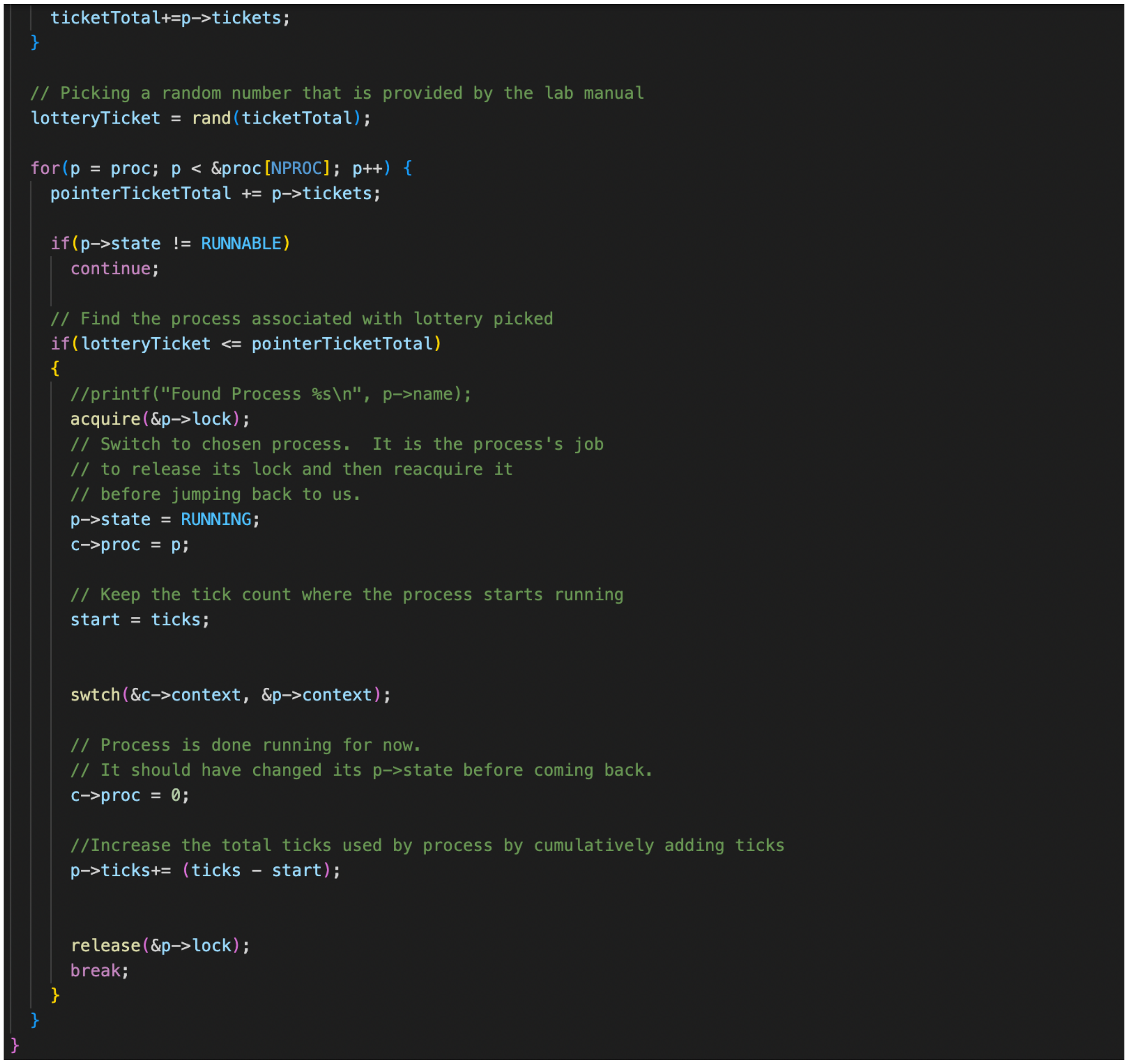
In lottery scheduling, each process is allocated a number of tickets. The number of tickets determine the relative priority of the process. A ticket is chosen randomly for each time slice or time quanta and then the ticket holding process is scheduled. Higher the number of tickets the higher the chances to be picked. Lottery scheduling is fair in the long term but in the short term it can be said it is unfair. It is probabilistic. There is a small chance that tickets of a certain process might never get picked.

We initiate counters in the structure of the process to hold tickets and total ticks that were used by the process. A random function provided by the lab instructions is used to generate a lottery ticket, we then loop through the p table to find a runnable process and check if the lottery ticket is of the same range. If yes, then the process is scheduled and the tick counter is updated.



```c
int ticketTotal = 0;
int lotteryTicket = 0;

int pointerTicketTotal = 0;
int start = 0;

for(;;){
  // Avoid deadlock by ensuring that devices can interrupt.
  intr_on();

  // Initialise variables
  ticketTotal = 0;
  pointerTicketTotal = 0;

  // Count the total number of tickets held by processes
  for(p = proc; p < &proc[NPROC]; p++) {
    ticketTotal+=p->tickets;
  }

  // Picking a random number that is provided by the lab manual
  lotteryTicket = rand(ticketTotal);

  for(p = proc; p < &proc[NPROC]; p++) {
    pointerTicketTotal += p->tickets;

    if(p->state != RUNNABLE)
      continue;

    // Find the process associated with lottery picked
    if(lotteryTicket <= pointerTicketTotal)
    {
      //printf("Found Process %s\n", p->name);
      acquire(&p->lock);
      // Switch to chosen process.  It is the process's job
      // to release its lock and then reacquire it
      // before jumping back to us.
      p->state = RUNNING;
      c->proc = p;

      // Keep the tick count where the process starts running
      start = ticks;
```

Fig: Changes in scheduler function for Lottery scheduling

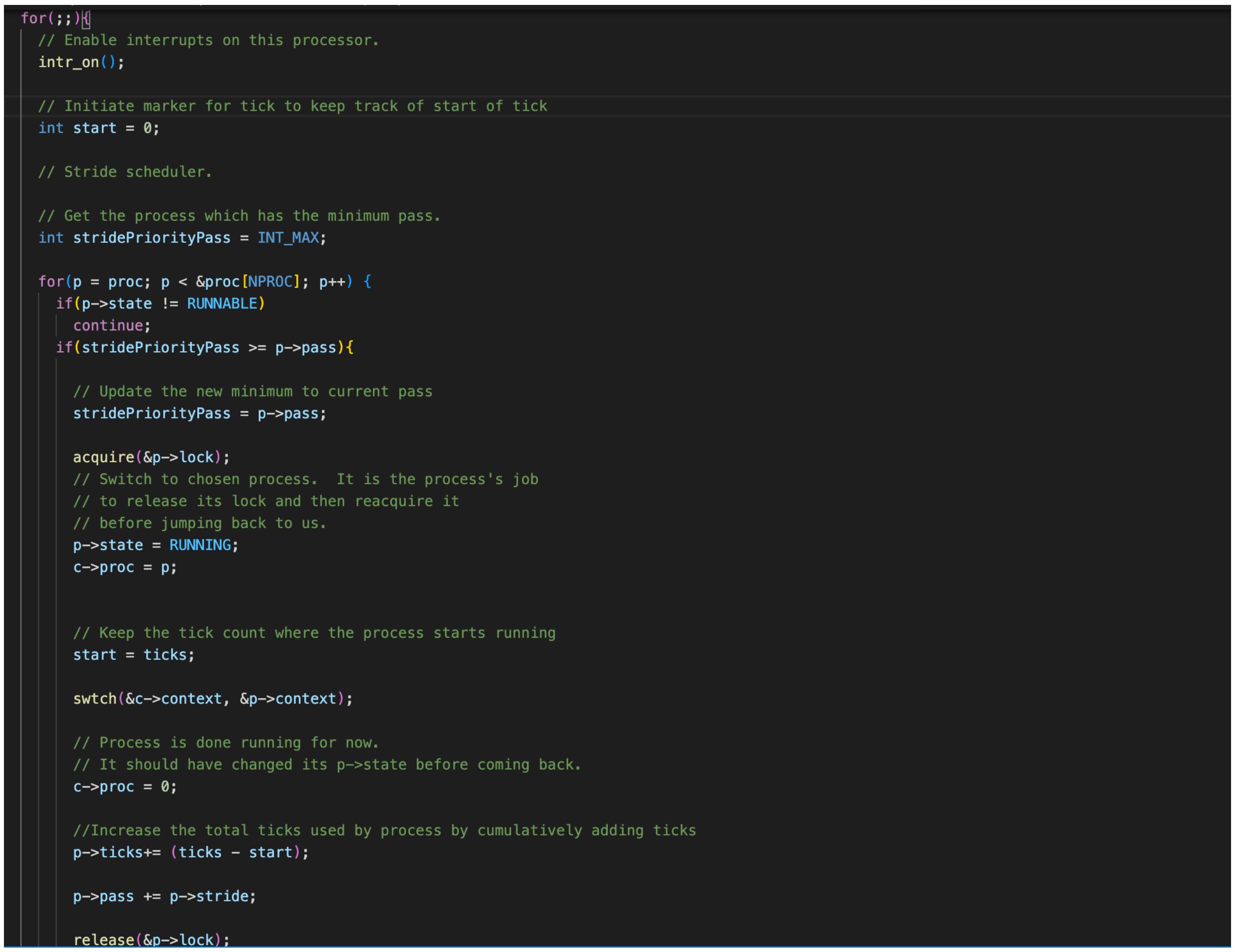


```
      ticketTotal+=p->tickets;
    }

    // Picking a random number that is provided by the lab manual
    lotteryTicket = rand(ticketTotal);

    for(p = proc; p < &proc[NPROC]; p++) {
      pointerTicketTotal += p->tickets;

      if(p->state != RUNNABLE)
        continue;

      // Find the process associated with lottery picked
      if(lotteryTicket <= pointerTicketTotal)
      {
        //printf("Found Process %s\n", p->name);
        acquire(&p->lock);
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;

        // Keep the tick count where the process starts running
        start = ticks;


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

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

        //Increase the total ticks used by process by cumulatively adding ticks
        p->ticks+= (ticks - start);


        release(&p->lock);
        break;
      }
    }
  }
```

Fig: Changes in scheduler function for Lottery scheduling (continued)

Stride Scheduling

Stride scheduling, in contrast to lottery system scheduling, is deterministic. Each process is allocated a number of tickets and the number of tickets denote the relative priority of the process. That is, the more tickets held by a process the more often it gets access to system resources. In stride scheduling, each process is assigned a pass value that is computed by dividing a large constant by the number of tickets held by the process. And the ticket with the minimum pass value is selected to be scheduled. Once scheduled it's pass value is incremented by adding the stride value to it. This way none of the process will be starved due to the probabilistic nature of scheduling.

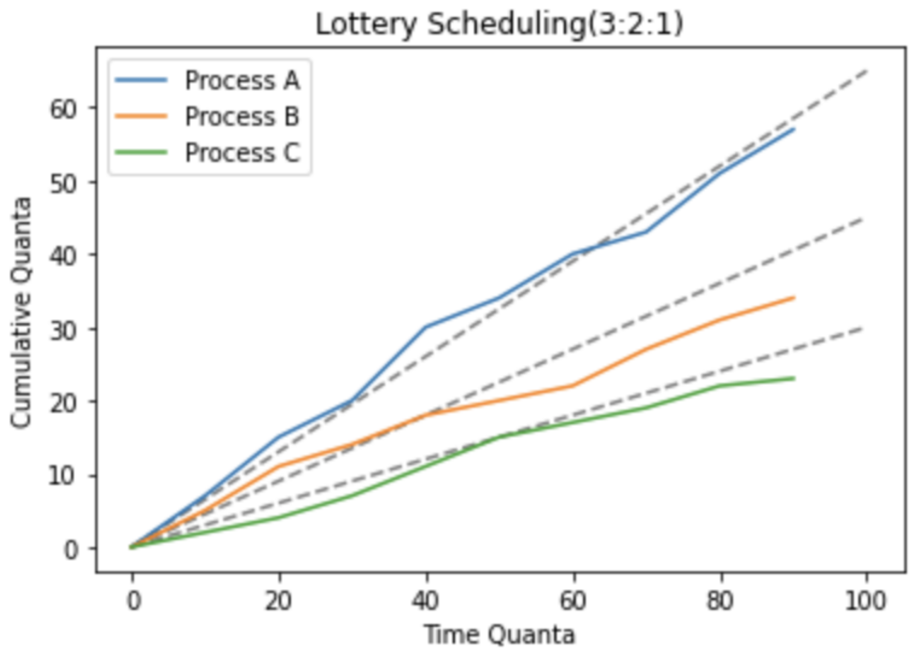


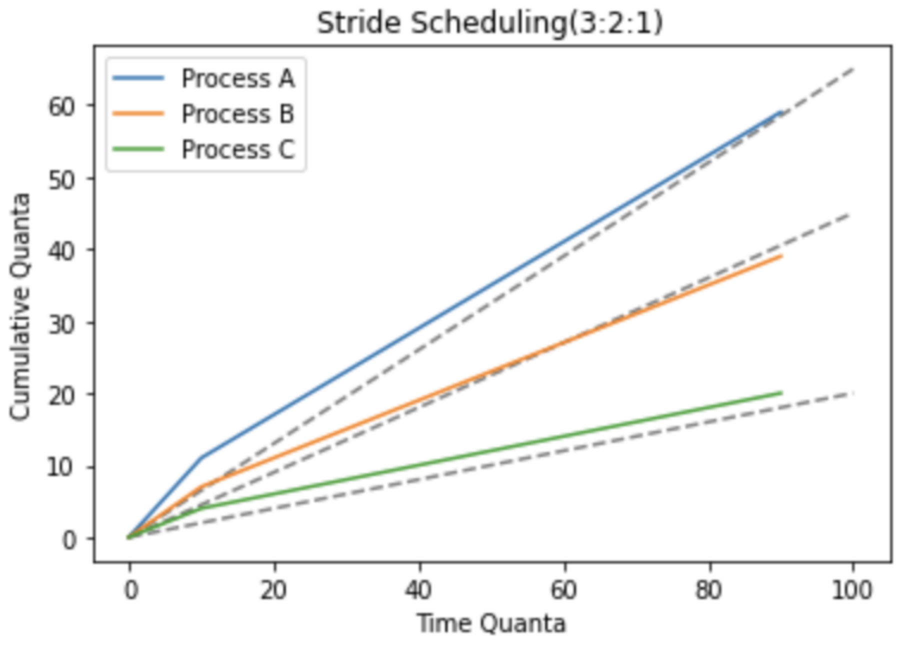
```
for(;;){
  // Enable interrupts on this processor.
  intr_on();

  // Initiate marker for tick to keep track of start of tick
  int start = 0;

  // Stride scheduler.

  // Get the process which has the minimum pass.
  int stridePriorityPass = INT_MAX;

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

      // Update the new minimum to current pass
      stridePriorityPass = p->pass;

      acquire(&p->lock);
      // Switch to chosen process.  It is the process's job
      // to release its lock and then reacquire it
      // before jumping back to us.
      p->state = RUNNING;
      c->proc = p;


      // Keep the tick count where the process starts running
      start = ticks;

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

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

      //Increase the total ticks used by process by cumulatively adding ticks
      p->ticks+= (ticks - start);

      p->pass += p->stride;

      release(&p->lock);
```

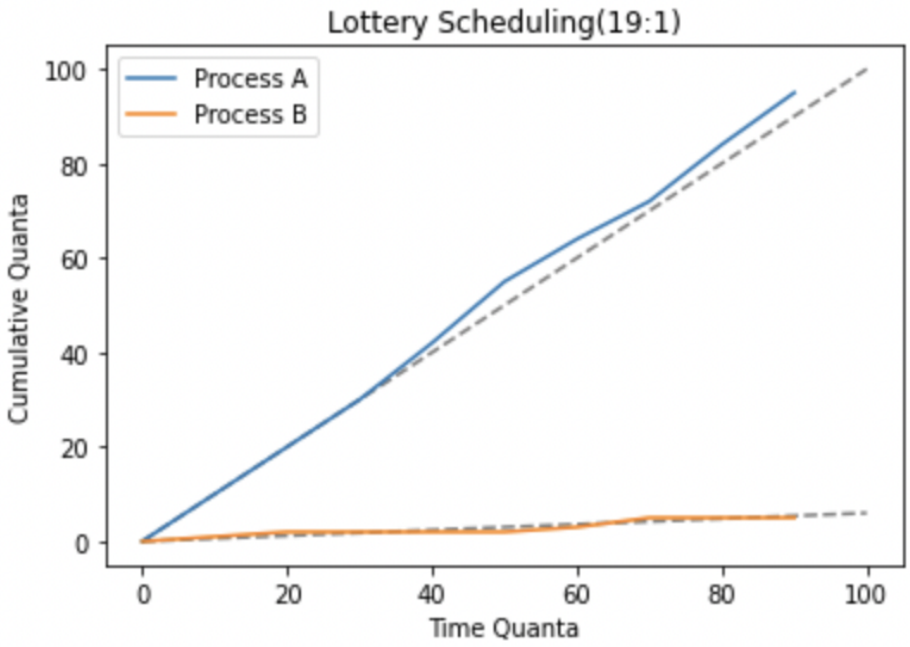
Fig: Changes in scheduler function for stride scheduling

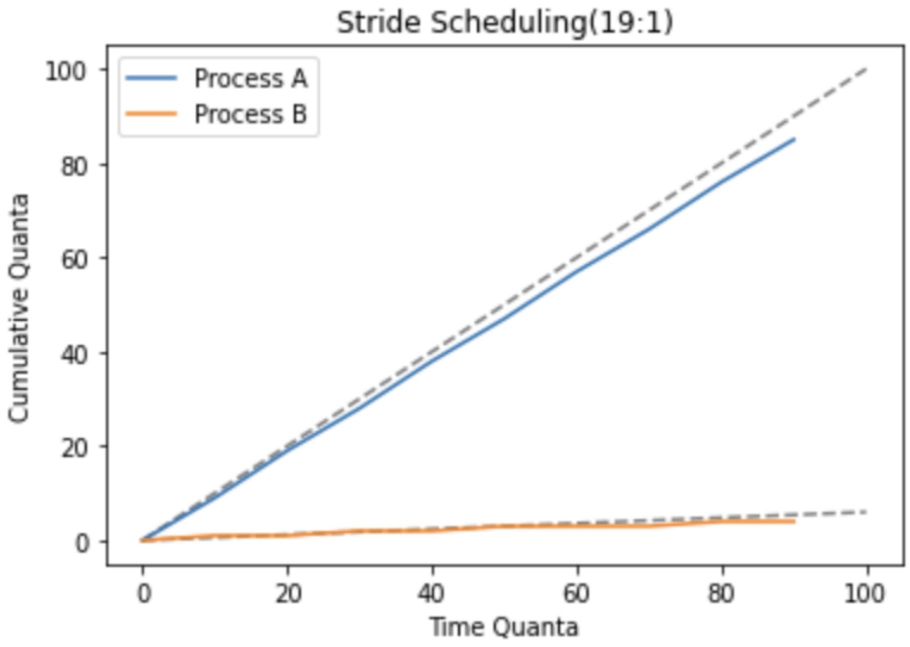
In the above snippet, we have implemented the Stride Scheduling. We make sure that interrupts are enabled. We start a counter for measuring ticks that acts as a marker and will help determine the ticks taken by a process cumulatively. We start with the highest value possible for stridePriorityPass. We loop through the p table to find the processes that are runnable. Once found we check if the process is less than the current value, if yes, then it is scheduled to be executed. Once the process is done executing, we update the pass value by incrementing it by stride value. The tick counter in the process structure is also updated to include the current tick as well. The initial pass and stride value are calculated in the int run_sched_tickets(int tickets) function.

Part 3: Experiments






Lottery Scheduling(19:1)
Process A
Process B
Cumulative Quanta
Time Quanta
0
20
40
60
80
100


Stride Scheduling(19:1)
Process A
Process B
Cumulative Quanta
Time Quanta
0
20
40
60
80
100