Operating System Concepts
COP4610.02
Mini Project 2

Noah Baldwin
Cody Carroll
Paul Teleweck

Work Breakdown:

Names	Code	Report	Documentation	Presentation
Noah Baldwin	16.67%	33.34%	33.34%	50%
Cody Carroll	66.67%	33.34%	33.34%	0%
Paul Teleweck	16.67%	33.34%	33.34%	50%

Abstract:

In this project, you'll be putting a new scheduler into xv6. It is called a lottery scheduler, and the full version is described in this chapter of the online book; you'll be building a simpler one. The basic idea is simple: assign each running process a slice of the processor based in proportion to the number of tickets it has; the more tickets a process has, the more it runs. Each time slice, a randomized lottery determines the winner of the lottery; that winning process is the one that runs for that time slice.

Changes Made:

RandomNumGen.h

```
#ifndef _RANDOM_H
#define _RANDOM_H
#include "types.h"
#define PHI 0x9e3779b9

static uint Q[4096], c = 362436;

static void srand(uint x)
{
        int i;

        Q[0] = x;
        Q[1] = x + PHI;
        Q[2] = x + PHI + PHI;

        for (i = 3; i < 4096; i++)
                        Q[i] = Q[i - 3] ^ Q[i - 2] ^ PHI ^ i;
}

static uint rand(void)
{
        if(sizeof(unsigned long long) != 8){
                return 0;
        }
        unsigned long long t, a = 18782LL;
        static uint i = 4095;
        uint x, r = 0xfffffffe;
        i = (i + 1) & 4095;
        t = a * Q[i] + c;
        c = (t >> 32);
        x = t + c;
        if (x < c) {
            x++;
            c++;
        }
        return (Q[i] = r - x);
}
#endif
```

- Created class to generate random number between 0 and the total number of tickets

Pstat.h

```
#ifndef _PSTAT_H
#define _PSTAT_H
#include "param.h"

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

struct pstat {

        _Bool inuse[NPROC];

        int pid[NPROC];

        int ticks[NPROC];

        int tickets[NPROC];

        enum procstate state[NPROC];

        int total_tickets;

};

#endif
```

- Added ticks, tickets, and total_tickets. Variables to hold the ticket values.

Syscall.h

syscall.h
~/Documents/xv6-mini-project-2/include

```c
#ifndef _SYSCALL_H_
#define _SYSCALL_H_

// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_write   5
#define SYS_read    6
#define SYS_close   7
#define SYS_kill    8
#define SYS_exec    9
#define SYS_open   10
#define SYS_mknod  11
#define SYS_unlink 12
#define SYS_fstat  13
#define SYS_link   14
#define SYS_mkdir  15
#define SYS_chdir  16
#define SYS_dup    17
#define SYS_getpid 18
#define SYS_sbrk   19
#define SYS_sleep  20
#define SYS_uptime 21
#define SYS_getpinfo 22
#define SYS_settickets 23

#endif // _SYSCALL_H_
```

- Added SYS_settickets and SYS_getpinfo to syscall.h and assigned system call numbers

Proc.c

proc.c
~/Desktop/xv6-mini-project-2/XV6/kernel

```c
int total_tickets;
void setproctickets(struct proc* pp, int n)
{
        total_tickets -= pp->tickets;
        pp->tickets = n;
        total_tickets += pp->tickets;
}

void storetickets(struct proc* pp)
{
        if(pp->state != SLEEPING)
                panic("Not sleeping at storetickets");
#ifdef STORE_TICKETS_ON_SLEEP
        total_tickets -= pp->tickets;
#endif
}


void restoretickets(struct proc* pp)
{
        if(pp->state != SLEEPING)
                panic("Not sleeping at waketickets");
#ifdef STORE_TICKETS_ON_SLEEP
        total_tickets += pp->tickets;
#endif
}

        static struct proc*
allocproc(void)
{
        struct proc *p;
        char *sp;

        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
                if(p->state == UNUSED)
                        goto found;
        release(&ptable.lock);
        return 0;

found:
        p->state = EMBRYO;
        p->pid = nextpid++;
        release(&ptable.lock);
```

C Tab Width: 8 Ln 68, Col 15 INS

- Functions for ticket process

```c
        struct proc *p;

        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                if(p->pid == pid){
                        p->killed = 1;
                        if(p->state == SLEEPING)
                        {
                                restoretickets(p);
                                p->state = RUNNABLE;
                        }
                        release(&ptable.lock);
                        return 0;
                }
        }
        release(&ptable.lock);
        return -1;
}

        void
procdump(void)
{
        cprintf("Total Tickets: %d\n\n", total_tickets);
        static char *states[] = {
                [UNUSED]    "unused",
                [EMBRYO]    "embryo",
                [SLEEPING]  "sleep ",
                [RUNNABLE]  "runble",
                [RUNNING]   "run   ",
                [ZOMBIE]    "zombie"
        };
        int i;
        struct proc *p;
        char *state;
        uint pc[10];

        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                if(p->state == UNUSED)
                        continue;
                if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
                        state = states[p->state];
                else
                        state = "???";
                cprintf("%d %s %s", p->pid, state, p->name);
```

- Define struct in order to call ptable from sysproc.c

Proc.h

```c
extern int ncpu;

extern struct cpu *cpu asm("%gs:0");
extern struct proc *proc asm("%gs:4");

struct context {
        uint edi;
        uint esi;
        uint ebx;
        uint ebp;
        uint eip;
};

struct proc {
        uint sz;
        pde_t* pgdir;
        char *kstack;
        enum procstate state;
        volatile int pid;
        struct proc *parent;
        struct trapframe *tf;
        struct context *context;
        void *chan;
        int killed;
        struct file *ofile[NOFILE];
        struct inode *cwd;
        char name[16];

        _Bool inuse;
        int ticks;
        int tickets;

};

struct ptable_type {
        struct spinlock lock;
        struct proc proc[NPROC];
};
extern struct ptable_type ptable;


void setproctickets(struct proc* pp, int n);

#endif // _PROC_H
```

- Created settickets method and defined ptable's struct to call it from other files

- Sysfunc.h

```c
#ifndef _SYSFUNC_H_
#define _SYSFUNC_H_

// System call handlers
int sys_chdir(void);
int sys_close(void);
int sys_dup(void);
int sys_exec(void);
int sys_exit(void);
int sys_fork(void);
int sys_fstat(void);
int sys_getpid(void);
int sys_kill(void);
int sys_link(void);
int sys_mkdir(void);
int sys_mknod(void);
int sys_open(void);
int sys_pipe(void);
int sys_read(void);
int sys_sbrk(void);
int sys_sleep(void);
int sys_unlink(void);
int sys_wait(void);
int sys_write(void);
int sys_uptime(void);
int sys_getpinfo(void);
int sys_settickets(void);

#endif // _SYSFUNC_H_
```

- Declared variables sys_getpinfo and sys_settickets

Syscall.c

```c
int
argstr(int n, char **pp)
{
  int addr;
  if(argint(n, &addr) < 0)
    return -1;
  return fetchstr(proc, addr, pp);
}

static int (*syscalls[])(void) = {
[SYS_chdir]   sys_chdir,
[SYS_close]   sys_close,
[SYS_dup]     sys_dup,
[SYS_exec]    sys_exec,
[SYS_exit]    sys_exit,
[SYS_fork]    sys_fork,
[SYS_fstat]   sys_fstat,
[SYS_getpid]  sys_getpid,
[SYS_kill]    sys_kill,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_mknod]   sys_mknod,
[SYS_open]    sys_open,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_unlink]  sys_unlink,
[SYS_wait]    sys_wait,
[SYS_write]   sys_write,
[SYS_uptime]  sys_uptime,
[SYS_getpinfo]  sys_getpinfo,
[SYS_settickets]  sys_settickets,
};


void
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num] != NULL) {
    proc->tf->eax = syscalls[num]();
```

- Added system call to set tickets and getpinfo.

Sysproc.c

```c
int sys_settickets(void)
{
        int number_of_tickets;
        // Error
        if(argint(0, &number_of_tickets) < 0)
                return -1;

        acquire(&ptable.lock);
        setproctickets(proc, number_of_tickets);
        release(&ptable.lock);

        return 0;
}

int sys_getpinfo(void)
{
        acquire(&ptable.lock);
        struct pstat* target;
        if(argint(0, (int*)(&target)) < 0)
                return -1;

        for(struct proc* p=ptable.proc;p != &(ptable.proc[NPROC]); p++)
        {
                const int index = p - ptable.proc;
                if(p->state != UNUSED)
                {
                        target->pid[index] = p->pid;
                        target->ticks[index] = p->ticks;
                        target->tickets[index] = p->tickets;
                        target->inuse[index] = p->inuse;
                        target->state[index] = p->state;
                }
        }
        target->total_tickets = total_tickets;
        release(&ptable.lock);
        return 0;

}
int sys_fork(void)
{
        return fork();
}

int sys_exit(void)
{
        exit();
        return 0;  // not reached
}

int sys_wait(void)
{
        return wait();
}

int sys_kill(void)
{
        int pid;
```

- Created sys_settickets and sys_getpinfo to change number of tickets in process and fill table with data.

State.c

```c
        }

        // Header
        fprintf(stdout,"This process: %d\n",
                        getpid());
        fprintf(stdout,"Total tickets: %d\n\n",
                        pinfo.total_tickets);
        fprintf(stdout,
                        "PID\tTicks\tTickets\tState\n");

        // Body
        for(int i=0;i<NPROC;i++){

                if(pinfo.state[i] == 2 || pinfo.pid[i] == current_pid)
                {
                        skip_yield = 1;
                }
                int ey_left =
                        (int)expected_yield;
                int ey_right =
                        (int)((expected_yield-ey_left)*10);
                int ay_left =
                        (int)actual_yield;
                int ay_right =
                        (int)((actual_yield-ay_left)*10);


                // Indicate which process is in use
                if(pinfo.inuse[i])
                        putchar('>');
                else
                        putchar('|');

                // Write the row
                fprintf(stdout,
                                skip_yield?
                                "%d\t%d\t%d\t%d\n":
                                "%d\t%d\t%d\t%d.%d\t%d.%d\n",
                                pinfo.pid[i],
                                pinfo.ticks[i],
                                pinfo.tickets[i],
                                pinfo.state[i],
                                ey_left,
                                ey_right,
                                ay_left,
                                ay_right
                                );
        }
        return 1;
}

int main(int argc, char *argv[])
{
        ps();

        exit();
}
```

- The user program state.c displays the PID, tickets, ticks, state, and percentages of each process. State.c also displaying whether or not the process is in use.

BabyMaker.c

```c
#include "types.h"
#include "user.h"

int main(int argc, char** argv)
{
        if(argc<2){
                fprintf(stdout, "Usage: make child_1_tickets [child_2_tickets]...\n");
                exit();
        }
        fprintf(stdout, "Parent creation (pid %d)\n", getpid());

        struct pstat pinfo = {0};
        if (-1 == getpinfo(&pinfo)) {
                return 0;
                fprintf(1, "\n\t GET PINFO FAILURE\n");
        }

        int total = 0;
        for (int i = 1; i < argc; i++)
                total += atoi(argv[i]);
        fprintf(stdout, "Tickets made: %d\nTotal tickets: %d\n", total, total+pinfo.total_tickets);

        for (int i=1;i<argc;i++) {
                const int pid = fork();

                if (pid<0) {
                        fprintf(stderr, "Failed to create child.");
                        exit();
                }
                if (!pid) {
                        const int t = atoi(argv[i]);//number of tickets
                        settickets(t);
                        fprintf(stdout, "Child %d created with %d tickets\n", getpid(), t);
                        fprintf(stdout, "Child %d exiting\n", getpid());
                        exit();
                }
        }
        for (int i=1; i<argc; i++) {
                wait();
        }
        fprintf(stdout, "Parent exiting\n");
        exit();
}
```

- Creates child processes

Makefile.mk

```
# user programs
USER_PROGS := \
        cat\
        BabyMaker\
        echo\
        forktest\
        grep\
        init\
        kill\
        ln\
        ls\
        mkdir\
        state\
        rm\
        sh\
        stressfs\
        usertests\
        wc\
        zombie\
```

- Add BabyMaker and state programs to user file

Ulib.c

```c
#include "types.h"
#include "stat.h"
#include "fcntl.h"
#include "user.h"
#include "x86.h"

char
getchar()
{
        char c;
    int cc = read(stdin, &c, 1);
        return cc==-1?-1:c;
}
char*
strcpy(char *s, char *t)
{
  char *os;

  os = s;
  while((*s++ = *t++) != 0)
    ;
  return os;
}

int
strcmp(const char *p, const char *q)
{
  while(*p && *p == *q)
    p++, q++;
  return (uchar)*p - (uchar)*q;
}

uint
strlen(const char *s)
{
  int n;

  for(n = 0; s[n]; n++)
    ;
  return n;
}

void*
memset(void *dst, int c, uint n)
{
  stosb(dst, c, n);
  return dst;
}

char*
strchr(const char *s, char c)
{
  for(; *s; s++)
    if(*s == c)
      return (char*)s;
  return 0;
}
```

- Created functions to get characters

User.h

```c
#ifndef _USER_H_
#define _USER_H_
#include "pstat.h"

struct stat;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(char*, char*);
int mkdir(char*);
int chdir(char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int getpinfo(struct pstat*);
int settickets(int);

// user library functions (ulib.c)
int stat(char*, struct stat*);
char* strcpy(char*, char*);
void *memmove(void*, void*, int);
char* strchr(const char*, char c);
int strcmp(const char*, const char*);
void fprintf(int, const char*, ...);
void putchar(char c);
char getchar(void);
char* gets(char*, int max);
uint strlen(const char*);
void* memset(void*, int, uint);
void* malloc(uint);
void* calloc(uint, uint);
void free(void*);
int atoi(const char*);

// Constants
extern const int stdin;
extern const int stdout;
extern const int stderr;

#endif // _USER_H_
```

- Added system calls to user header file

Results:

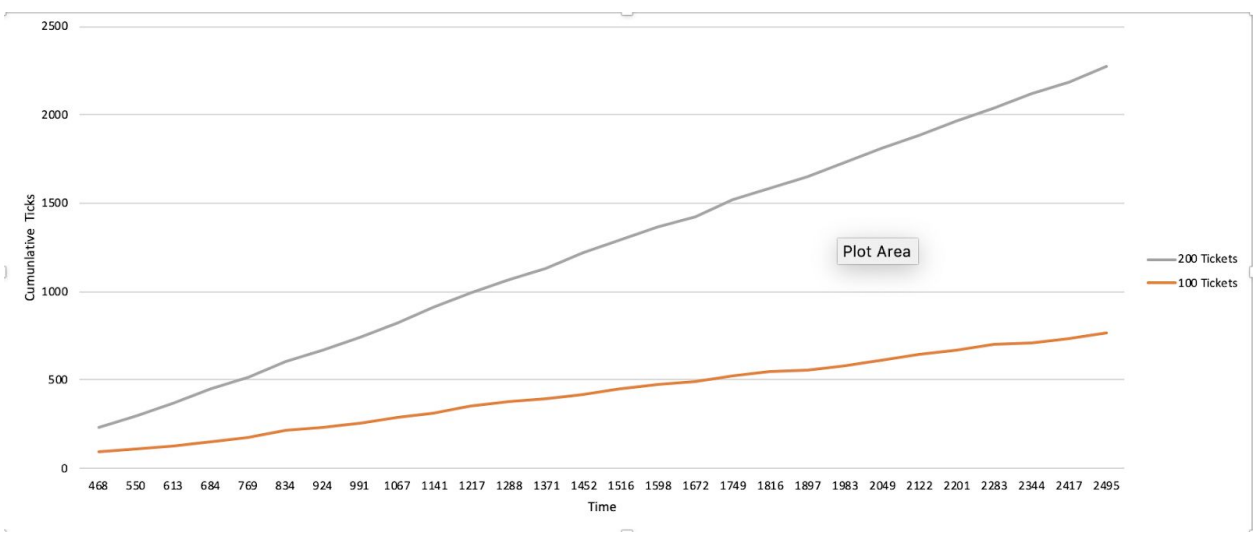
Output

```
QEMU
lapicinit: 0 0xfee00000

cpu0: starting xv6

cpu0: starting
init: starting sh
$ BabyMaker 24 54 30 64 12 6
Parent creation (pid 3)
Tickets made: 190
Total tickets: 391
Child 5 created with 54 tickets
Child 5 exiting
Child 4 created with 24 tickets
Child 4 exiting
Child 6 created with 30 tickets
Child 6 exiting
Child 7 created with 64 tickets
Child 7 exiting
Child 8 created with 12 tickets
Child 8 exiting
Child 9 created with 6 tickets
Child 9 exiting
Parent exiting
$
```

Graph



- The above graph shows the total accumulated ticks of the processes.