Paraskevi Dimoragka SID 862121820
CS202 lab
Fall 2019

LOTTERY SCHEDULER

To benchmark the lottery scheduler, we set the number of CPUs to 1 in the Makefile.

```
Booting from Hard Disk..xv6...
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
From  sh-4: 1 sleep  init tickets=10 ticks=22
From  sh-4: 2 sleep  sh tickets=10 ticks=14
From  sh-4: 3 sleep  sh tickets=10 ticks=2
From  sh-4: 4 run    sh tickets=10 ticks=1
From  sh-4: 5 runble sh tickets=10 ticks=0
From  sh-6: 1 sleep  init tickets=10 ticks=22
From  sh-6: 2 sleep  sh tickets=10 ticks=14
From  sh-6: 3 sleep  sh tickets=10 ticks=3
From  sh-6: 6 run    sh tickets=10 ticks=1
From  sh-6: 5 runble sh tickets=10 ticks=0
From  sh-6: 7 runble sh tickets=10 ticks=0
From  prog1-5: 1 sleep  init tickets=10 ticks=22
From  prog1-5: 2 sleep  sh tickets=10 ticks=14
From  prog1-5: 3 runble prog3 tickets=10 ticks=189
From  prog1-5: 5 run    prog1 tickets=30 ticks=495
From  prog1-5: 7 runble prog2 tickets=20 ticks=336
zombie!
From  prog2-7: 1 sleep  init tickets=10 ticks=23
From  prog2-7: 2 sleep  sh tickets=10 ticks=14
From  prog2-7: 3 runble prog3 tickets=10 ticks=245
From  prog2-7: 7 run    prog2 tickets=20 ticks=493
zombie!
From  prog3-3: 1 sleep  init tickets=10 ticks=24
From  prog3-3: 2 sleep  sh tickets=10 ticks=14
From  prog3-3: 3 run    prog3 tickets=10 ticks=499
```

During the exit of prog1 when all 3 processes are in the **lottery scheduler** we get:

prog1: 495 ticks
prog2: 336 ticks
prog3: 189 ticks
Total : 1020 ticks

Allocated ratio per process:

prog1: 495/1020 = 0.485 (Theoretical 0.50)
prog2: 336/1020 = 0.3294 (Theoretical 0.33..)
prog3: 189/1020 = 0.1852 (Theoretical 0.166..)

File: syscall.h

We define the system call number for the function that sets the number of tickets to a processes:

```
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_info   22
#define SYS_settickets 23
```

File: syscall.c

We add the function **sys_settickets(void)** as external:

```
extern int sys_uptime(void);
extern int sys_info(void);
extern int sys_settickets(void);
```

```
[SYS_close]   sys_close,
[SYS_info]    sys_info,
[SYS_settickets] sys_settickets
};
```

File: sysproc.c

We add the implementation of the method **sys_settickets(void)** that sets the number of tickets to a process:

```
int
sys_settickets(void)
{
  int n;

  if(argint(0, &n) < 0)
    return -1;

  struct proc *proc = myproc();
  proc->tickets = n;

  return 23;
}
```

File: proc.h

We add two new integers members in the proc structure:

```
// 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 syscallcount;            // Number of system calls by process
  int tickets;                 // Number of tickets
  int ticks;                   // Number of ticks
};
```

tickets: number of tickets assigned to the process
ticks: number of times the process was scheduled to run

File: proc.c

We add the header that contains the random generator function to use in the scheduler:

```
#include "proc.h"
#include "spinlock.h"
#include "rand.h"
```

In the function **allocproc**, we initialize the tickets for every process to 10 and also initialize the tickets to 0:

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

We patch the **exit** function with the code that was provided (with the necessary changes):

```
/*--------The following code is added to format the output--------*/
/* NOTE that you need to replace sched_times in the cprintf with whatever you use to record the execution time */
static char *states[] = {
  [UNUSED]    "unused",
  [EMBRYO]    "embryo",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
};
char *state;
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("From  %s-%d: %d %s %s tickets=%d ticks=%d \n", myproc()->name, myproc()->pid, p->pid, state, p->name, p->tickets, p->ticks);
}
/*-------------------patch end------------------------ */
```

to have formatted output. In the scheduler function, we calculate the total number of tickets for the runnable processes, we generate a winning ticket between [0,**totaltickets**] and run the process if the **totaltickets>=winningticket** where we then increase the number of ticks of the process by one:

```
int ticketspassed = 0;
int totaltickets = 0;

// total number of tickets of runnable processes
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;
  totaltickets = totaltickets + p->tickets;
}

//
long winningticket = random_less_than(totaltickets);

// Loop over process table looking for process to run.
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;

  //
  ticketspassed += p->tickets;
  if(ticketspassed < winningticket){
    continue;
  }
  p->ticks +=1;

  // 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();

  // Process is done running for now.
  // It should have changed its p->state before coming back.
  c->proc = 0;
  break;
}
release(&ptable.lock);
```

File: user.h

We add the function that sets the number of tickets, **settickets**, and the three functions (**prog1**, **prog2**, **prog3**) that contain our programs with the 3 different ticket numbers (30,20,10):

```
int info(int);
int settickets(int);
int prog1(void);
int prog2(void);
int prog3(void);
```

File: usys.S

We add to the system calls the function **settickets**:

```
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(info)
SYSCALL(settickets)
```

Files: prog1.c, prog2.c, prog3.c

Provided program to benchmark the lottery scheduler where we call our method, **settickets**, for setting the number of tickets to the process for three different values, 30,20 and 10:

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

int main(int argc, char *argv[])
{
  settickets(30);
  int i,k;
  const int loop=43000;
  for(i=0;i<loop;i++)
  {
     asm("nop"); //in order to prevent the compiler from optimizing the for loop
     for(k=0;k<loop;k++)
     {
        asm("nop");
     }
  }
  exit();
}
```

C code to help for random number generation are given in the files rand.h and rand.c

STRIDE SCHEDULER

```
Booting from Hard Disk..xv6...
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
From  sh-4: 1 sleep  init tickets=10 ticks=23
From  sh-4: 2 sleep  sh tickets=10 ticks=15
From  sh-4: 3 sleep  sh tickets=10 ticks=1
From  sh-4: 4 run    sh tickets=10 ticks=1
From  sh-4: 5 runble sh tickets=10 ticks=0
From  sh-6: 1 sleep  init tickets=10 ticks=23
From  sh-6: 2 sleep  sh tickets=10 ticks=15
From  sh-6: 3 sleep  sh tickets=10 ticks=3
From  sh-6: 6 run    sh tickets=10 ticks=2
From  sh-6: 5 runble sh tickets=10 ticks=2
From  sh-6: 7 runble sh tickets=10 ticks=0
From  prog1-5: 1 sleep  init tickets=10 ticks=23
From  prog1-5: 2 sleep  sh tickets=10 ticks=15
From  prog1-5: 3 runble prog3 tickets=10 ticks=181
From  prog1-5: 5 run    prog1 tickets=30 ticks=530
From  prog1-5: 7 runble prog2 tickets=20 ticks=355
zombie!
From  prog2-7: 1 sleep  init tickets=10 ticks=24
From  prog2-7: 2 sleep  sh tickets=10 ticks=15
From  prog2-7: 3 runble prog3 tickets=10 ticks=264
From  prog2-7: 7 run    prog2 tickets=20 ticks=521
zombie!
From  prog3-3: 1 sleep  init tickets=10 ticks=25
From  prog3-3: 2 sleep  sh tickets=10 ticks=15
From  prog3-3: 3 run    prog3 tickets=10 ticks=531
```

During the exit of prog1 when all 3 processes are in the **stride scheduler** we get:

prog1: 530 ticks
prog2: 355 ticks
prog3: 181 ticks
Total : 1066 ticks

Allocated ratio per process:

prog1: 530/1066 = 0.497 (Theoretical 0.50)
prog2: 336/1066 = 0.333 (Theoretical 0.33..)
prog3: 189/1066 = 0.170 (Theoretical 0.166..)

We note that even on a single run the results of the stride scheduler are closer to the theoretical values compared to the lottery scheduler.

File: syscall.h
We define the system call number for the function that sets the number of tickets to a processes:

```
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_info   22
#define SYS_settickets 23
```

File: syscall.c

We add the function **sys_settickets(void)** as external:

```
extern int sys_uptime(void);
extern int sys_info(void);
extern int sys_settickets(void);
[SYS_close]   sys_close,
[SYS_info]    sys_info,
[SYS_settickets] sys_settickets
};
```

File: proc.c

Initialize tickets, ticks and stride:

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->tickets = 10;
  p->stride = LCM / p->tickets;
  p->ticks = 0;
  release(&ptable.lock);
```

File: sysproc.c

Function **sys_settickets** sets the number of tickets and the stride based on the number of tickets.

```
int
sys_settickets(void)
{
  int n;

  if(argint(0, &n) < 0)
    return -1;
  if(n % 10 != 0 || n < 10 || n > 60)
  {
    return -1;
  }
  struct proc *proc = myproc();
  proc->tickets = n;
  proc->stride  = LCM / n;

  return 23;
}
```

Changes in the scheduler to accommodate the stride logic:

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

  // Loop over process table looking for process to run.
  acquire(&ptable.lock);

  p = getminproc();
  int tentativepass = getmaxpass();
  if (tentativepass > TENTATIVE_CEIL)
  {
    lowerpassval();
  }

  if (p != 0) {

    p->pass += p->stride;
    p->ticks++;

    // 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();

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0;
  }
  release(&ptable.lock);
 }
}
```

File: types.h

Auxiliary constants added:

```
// Least Common Multiple of 10,20,30,40,50,60
#define LCM 600
#define TENTATIVE_CEIL 5000000000u
typedef unsigned int   uint;
typedef unsigned short ushort;
typedef unsigned char  uchar;
typedef uint pde_t;
```

File: user.h

We add the function that sets the number of tickets, **settickets**, and the three functions (**prog1**, **prog2**, **prog3**) that contain our programs with the 3 different ticket numbers (30,20,10):

```
int info(int);
int settickets(int);
int prog1(void);
int prog2(void);
int prog3(void);
```

File: usys.S

We add to the system calls the function **settickets**:

```
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(info)
SYSCALL(settickets)
```

Files: prog1.c, prog2.c, prog3.c

Provided program to benchmark the lottery scheduler where we call our method, **settickets**, for setting the number of tickets to the process for three different values, 30,20 and 10:

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

int main(int argc, char *argv[])
{
  settickets(30);
  int i,k;
  const int loop=43000;
  for(i=0;i<loop;i++)
  {
     asm("nop"); //in order to prevent the compiler from optimizing the for loop
     for(k=0;k<loop;k++)
     {
        asm("nop");
     }
  }
  exit();
```