Assignment 3

- Prachika Agarwal (pa2191)
- Shreya Singh (ss14937) **(To be used for Grading)**

NICE SYSTEM CALL

To implement nice system call we create two system calls - ps (to list the process id, state of processes and priority) and nice system call where we pass the process id and priority which we want to set

Steps to create ps and nice system calls and nicetest to test nice systemcall:

1) Usys.S

   Add the following code:

```
SYSCALL(ps)
SYSCALL(nice)
```

2) defs.h

   Add the following code under the section proc.c

```
int             nice(int pid, int priority);
int             ps(void);
```

3) syscall.h

   Add the following code:

```
#define SYS_ps     22
#define SYS_nice   23
```

4) user.h

   Add the following code:

```
int nice(int pid, int priority);
int ps(void);
```

5) syscall.c

   Add the following code:

```
extern int sys_nice(void);
extern int sys_ps(void);
```

In static int (*syscalls[])(void) add the following code:

```
[SYS_nice] sys_nice,   [SYS_ps] sys_ps,
```

6) Makefile

In UPROGS add the path of the file ps.c, nice.c for the system call you created, and nicetest.c to test the nice function:

```
$U/_ps\
$U/_nice\
$U/_nicetest\
```

7) Proc.h

Here in struct proc declare the variable priority.

```
int priority;                // Nice system call priority
```

8) Sysproc.c

Add system calls for ps and nice:

```
int sys_ps(void)
{
 return ps();
}

int sys_nice(void)
{
 int pid, pr;
 if(argint(0, &pid) < 0)
   return -1;
 if(argint(1, &pr) < 0)
   return -1;

 return nice(pid, pr);
}
```

9) proc.c

Add system calls for ps and nice:

```
//List Processid and state

int ps()
{
struct proc *p;
```

```
sti();
acquire(&ptable.lock);
cprintf("name \t pid \t state \t priority \n");
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
   if(p->state == SLEEPING)
     cprintf("%s \t %d \t SLEEPING \t %d \n ", p->name,p->pid,p->priority);
   else if(p->state == RUNNING)
     cprintf("%s \t %d \t RUNNING \t %d \n ", p->name,p->pid,p->priority);
   else if(p->state == RUNNABLE)
     cprintf("%s \t %d \t RUNNABLE \t %d \n ", p->name,p->pid,p->priority);
}
release(&ptable.lock);
return 22;
}

// Nice system call

int nice(int pid, int priority)
{
   struct proc *p;
   acquire(&ptable.lock);
   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
     if(p->pid == pid){
           p->priority = priority;
           break;
       }
   }
   release(&ptable.lock);
   return priority;
}
```

10) Create a file ps.c

Add the following code:

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user.h"
#include "kernel/fcntl.h"

int main(void){
  ps();
  exit();
}
```

11) Create a file nice.c

Add the following code:

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user.h"
#include "kernel/fcntl.h"

int main(int argc, char *argv[])
{
 int priority, pid;
 if(argc < 3){
   printf(2,"Usage: nice pid priority\n");
   exit();
 }
 pid = atoi(argv[1]);
 priority = atoi(argv[2]);
 if (priority < 0 || priority > 20){
   printf(2,"Invalid priority (0-20)!\n");
   exit();
 }
 nice(pid, priority);
 exit();
}
```

12) Create a file nicetest.c

Add the following code:

```
#include "kernel/types.h"
#include "user.h"

int
main(int argc, char *argv[])
{
 printf(1, "nice test\n");
 int testval = nice(1,4); // get nice val
 int ival;
 if (testval < 10) {
   ival = nice(1,5); // inc by 5
   if (ival == testval + 1) {
     printf(1, "successfully called nice(5)\n");
   } else {
```

```
      printf(1, "nice(5) failed. oval: %d, nval: %d\n", testval, ival);
    }
  } else {
    ival = nice(1,-5); // dec by 5
    if (ival == testval - 5) {
      printf(1, "successfully called nice(-5)\n");
    } else {
      printf(1, "nice(-5) failed. oval: %d, nval: %d\n", testval, ival);
    }
  }

  testval = nice(2,10);
  ival = nice(2,90);
  if (ival == testval + 80) {
    printf(1, "successfully called nice(90)\n");
  } else {
    printf(1, "nice(90) failed\n");
  }

  ival = nice(2,0);
  if (ival == testval - 10) {
    printf(1, "successfully reset nice value to 0\n");
  } else {
    printf(1, "reset nice value failed\n");
  }

  exit();
}
```

OUTPUT of nice system call:

Call ps function get the process id of the process whose priority you want to update, call the nice function as nice <process_id> <priority>. Call the ps function again to see if priority is updated.

```
$ ps
name     pid     state   priority
init     1       SLEEPING        0
 sh      2       SLEEPING        0
 ps      3       RUNNING         0
 $ nice 1 3
$ ps
name     pid     state   priority
init     1       SLEEPING        3
 sh      2       SLEEPING        0
 ps      5       RUNNING         0
 $
```

OUTPUT of nicetest:

Call the file nicetest where you have written your test cases. In this file we attempt to update the priority of process with processid 1 to 5 and for the process with process id 2 we increment the priority to 90 and then reset it to 0.

```
$ ps
name     pid     state   priority
init     1       SLEEPING        0
 sh      2       SLEEPING        0
 ps      3       RUNNING         0
 $ nicetest
nice test
successfully called nice(5)
successfully called nice(90)
successfully reset nice value to 0
$ ps
name     pid     state   priority
init     1       SLEEPING        5
 sh      2       SLEEPING        0
 ps      5       RUNNING         0
 $
```

RANDOM NUMBER SYSTEM CALL

To implement the random number system call we create a system call called randomno and a test case file randomtest1.c

Steps are:

1) Usys.S

   Add the following code:

```
SYSCALL(randomno)
```

2) defs.h

   Add the following code under the section proc.c

```
unsigned int    randomno(void);
```

3) syscall.h

   Add the following code:

```
#define SYS_randomno 24
```

4) user.h

   Add the following code:

```
unsigned int randomno(void);
```

5) syscall.c

Add the following code:

```
extern int sys_randomno(void);
```

In static int (*syscalls[])(void) add the following code:

```
[SYS_randomno]  sys_randomno,
```

6) Makefile

In UPROGS add the path of the file randomno.c for the system call you created, and randomtest1.c to test the randomno function:

```
$U/_randomno\
$U/_randomtest1\
```

7) Sysproc.c

Add system calls for randomno:

```
int sys_randomno(void)
{
 return randomno();
}
```

8) proc.c

Add system calls for randomno:

```
unsigned int randomno(void)
{
 static unsigned int z1 = 12345, z2 = 12345, z3 = 12345, z4 = 12345;
 unsigned int b;
 b  = ((z1 << 6) ^ z1) >> 13;
 z1 = ((z1 & 4294967294U) << 18) ^ b;
 b  = ((z2 << 2) ^ z2) >> 27;
 z2 = ((z2 & 4294967288U) << 2) ^ b;
 b  = ((z3 << 13) ^ z3) >> 21;
 z3 = ((z3 & 4294967280U) << 7) ^ b;
 b  = ((z4 << 3) ^ z4) >> 12;
 z4 = ((z4 & 4294967168U) << 13) ^ b;
 int ans = (z1 ^ z2 ^ z3 ^ z4) / 2;
 cprintf("Random no is: %d", (z1 ^ z2 ^ z3 ^ z4) / 2);
 exit();
return ans;
}
```

9) randomno.c

Add the following code:

```
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/defs.h"



//Random no system call

unsigned int getrandomno(void)
{
 return randomno();
 exit();
}
```

10) randomtest1.c

Add the following code to test randomno function:

```
#include "kernel/types.h"
#include "user.h"

#define NUM_ITEMS 400

int main(int argc, char *argv[])
{
 printf(1, "random test\n");

 // random numbers between 0 and ((2^32 - 1) / 2), which is 2147483647.
 int i;

 printf(1, "random numbers between 0 and 2147483647:\n");
 for (i = 0; i < NUM_ITEMS; i++) {
   printf(1, "%d ", randomno());
 }
 exit();
}
```

OUTPUT:

Randomno system call:

```
$ randomno
Random no is: 1669098581$
$
```

Randomtest1.c :

```
$ randomtest1
random test
random numbers between 0 and 2147483647:
Random no is: 113630796$
```

LOTTERY SCHEDULER

For lottery scheduling we have designed a random number generator that helps us pick out the winning ticket. We have also kept the the default Round Robin algorithm which can be found in the scheduler function in proc.c file and can be run using “make qemu DLOTTERY”.
For lottery scheduling,

- We first run over all the process to run using acquire(&ptable lock)
- We then fetch the total number of tickets and check if it is zero, if that’s the case then that would mean all the processes have run. In this case we release the lock.
- We then go on to hold lottery, we decide the winner using our random function in the range 0-total tickets.
- Once the winner is decided we loop through all the processes which are in RUNNABLE state to look for the winner.
- Once we get our winner, we decrease it’s nice value and so increase its priority.

In our three test cases we have tested our scheduler prioritizes lower nice values but at the same time some lower nice valued processes have been executed before others. This shows that, that process is our random winner and so have been executed. This solves the issue of starvation.

Steps for implementing Lottery Scheduler and test cases:

1) Usys.S

   Add the following code:

```
SYSCALL(settickets)
SYSCALL(getpinfo)
SYSCALL(yield)
```

2) defs.h

   Add the following lines of code under the section proc.c

```
int             settickets(int);
void            getpinfo(struct pstat* pt);
void            yield(void);
```

3) syscall.h

   Add the following code:

```
#define SYS_settickets 25
#define SYS_getpinfo   26
#define SYS_yield  27
```

4) User.h

   Add the following code:

```
int settickets (int);
int getpinfo(struct pstat*);
int yield(void);
```

5) syscall.c

   Add the following code:

```
extern int sys_settickets(void);
extern int sys_yield(void);
extern int sys_getpinfo(void);
```

   In static int (*syscalls[])(void) add the following code:

```
[SYS_getpinfo]   sys_getpinfo,  [SYS_yield]   sys_yield, [SYS_randomno]
sys_randomno,
```

6) Makefile

Add the following code under UPROGS, settickets is our lottery scheduler and lottery1, lottery2, lottery3, lotteryTest are our test cases for lottery scheduler

```
$U/_settickets\
$U/_lottery1\
$U/_lottery2\
$U/_lottery3\
```

7) proc.h

Here in struct proc declare the variables tickets and nice.

```
int tickets;              // Number of tickets assigned for scheduling
char nice;
```

8) sysproc.c

Add system calls for settickets, yield and getpinfo:

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

 if(argint(0, &n) < 0)

   return -1;

 proc->tickets = n;

 return n;

}

int sys_yield(void)
{
   yield();
   return 0;
}

int sys_getpinfo(void) {
   struct pstat* pt;

   if(argptr(0, (void*)&pt, sizeof(struct pstat*)) < 0)
       return -1;
```

```
  void pinfo(struct pstat* pt);
  return 0; // use function in proc.c for access to ptable
}
```

9) Proc.c

Modify the code for scheduler as follows:

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

#ifndef LOTTERY
 // Round Robin
 for(;;){
   // Enable interrupts on this processor.
   sti();

   // 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;

     // cprintf("RUN %s, [pid %d]\n", p->name, p->pid);

     // Switch to chosen process.  It is the process's job
     // to release ptable.lock and then reacquire it
     // before jumping back to us.
     proc = p;
     switchuvm(p);
     p->state = RUNNING;
     swtch(&cpu->scheduler, proc->context);
     switchkvm();

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

```
#else
 // Lottery Ticket Scheduling
 // compile with -DLOTTERY
 for(;;){
   // Enable interrupts on this processor.
   sti();

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

   int x = 200;
   luckyincrease(x); // approximately every x times this is called, increase
all RUNNABLE priorities to 20 (max).

   // get total number of tickets by iterating through every process
   uint total = totaltickets();
   if (total == 0) {
     release(&ptable.lock);
     continue;
   }
   // cprintf("total: %d\n", total);

   // hold lottery
   uint counter = 0; // used to track if we've found the winner yet
   uint winner = randomrange(1, (int) total);
   // cprintf("winner: %d\n", winner);

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

     int nice = p->nice;
     counter += numtickets(nice);
     if (counter < winner)
       continue;

     // cprintf("Winner: %s, [pid %d]\n", p->name, p->pid);

     // Switch to chosen process.  It is the process's job
     // to release ptable.lock and then reacquire it
     // before jumping back to us.
     proc = p;
```

```
      switchuvm(p);
      p->state = RUNNING;

      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      proc->nice = proc->nice - 1;

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

#endif

}
```

Add a function pinfo as follows:

```
void pinfo(struct pstat* pt) {
   struct proc* p;

   acquire(&ptable.lock);
   int i = 0;
   for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) { // go through procs
       if (p->state == UNUSED) continue;
       pt->pid[i] = p->pid;
       pt->tickets[i] = p->tickets;
      // pt->ticks[i] = p->ticks; // TODO: no clue how track ticks
       i++;
   }
   pt->num_processes = i + 1;
   release(&ptable.lock);
}
```

10) Rand.c

Create a new file rand.c in the kernel as follows:

```
#define N 624
```

```
#define M 397
#define MATRIX_A 0x9908b0df
#define UPPER_MASK 0x80000000
#define LOWER_MASK 0x7fffffff
#define TEMPERING_MASK_B 0x9d2c5680
#define TEMPERING_MASK_C 0xefc60000
#define TEMPERING_SHIFT_U(y)  (y >> 11)
#define TEMPERING_SHIFT_S(y)  (y << 7)
#define TEMPERING_SHIFT_T(y)  (y << 15)
#define TEMPERING_SHIFT_L(y)  (y >> 18)

#define RAND_MAX 0x7fffffff

static unsigned long mm[N];
static int mmi=N+1;

void sgenrand(unsigned long seedval)
{
   mm[0]= seedval & 0xffffffff;
   for (mmi=1; mmi<N; mmi++)
       mm[mmi] = (69069 * mm[mmi-1]) & 0xffffffff;
}

long genrand(void)
{
   unsigned long y;
   static unsigned long mag[2]={0x0, MATRIX_A};

   if (mmi >= N) {
       int kk;

       if (mmi == N+1)
           sgenrand(4357);

       for (kk=0;kk<N-M;kk++) {
           y = (mm[kk]&UPPER_MASK)|(mm[kk+1]&LOWER_MASK);
           mm[kk] = mm[kk+M] ^ (y >> 1) ^ mag[y & 0x1];
       }
       for (;kk<N-1;kk++) {
           y = (mm[kk]&UPPER_MASK)|(mm[kk+1]&LOWER_MASK);
           mm[kk] = mm[kk+(M-N)] ^ (y >> 1) ^ mag[y & 0x1];
       }
```

```
        y = (mm[N-1]&UPPER_MASK)|(mm[0]&LOWER_MASK);
        mm[N-1] = mm[M-1] ^ (y >> 1) ^ mag[y & 0x1];

        mmi = 0;
    }
     y = mm[mmi++];
    y ^= TEMPERING_SHIFT_U(y);
    y ^= TEMPERING_SHIFT_S(y) & TEMPERING_MASK_B;
    y ^= TEMPERING_SHIFT_T(y) & TEMPERING_MASK_C;
    y ^= TEMPERING_SHIFT_L(y);

    return y & RAND_MAX;
}

long random_at_most(long max) {
 unsigned long
    num_bins = (unsigned long) max + 1,
    num_rand = (unsigned long) RAND_MAX + 1,
    bin_size = num_rand / num_bins,
    defect   = num_rand % num_bins;

 long x;
 do {
  x = genrand();
 }
 while (num_rand - defect <= (unsigned long)x);

 return x/bin_size;
}
```

11) Rand.h

Create a new file rand.h in kernel as follows:

```
void sgenrand(unsigned long);
long genrand(void);
long random_at_most(long);
```

12) settickets.c

Create a file settickets.c in users as follows:

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

```
#include "kernel/fcntl.h"

int main(void){
 settickets(0);
 exit();}
```

13) lottery1.c

Create a new test case lottery1.c in user as follows:
Here we test the lottery ticket scheduler, this test case makes sure that the process with the highest priority gets to run the most.
It also assures that processes with a priority of 0 (lowest possible) still gets to run
Here, we fork into 4 processes with different nice values

```
#include "user.h"
#include "kernel/types.h"
#define X 5

int main(int argc, char *argv[])
{
 printf(1, "lottery test 1\n");
 int pid;

 pid = fork();
 if (pid == 0) {
   int count = 0;
   int laps = 0;
   while(count<5) {
     // only print after X iterations as prints are expensive
     count++;
     if (laps < X) {
       laps++;
       count = 0;
       printf(1, "BEST \n");
     }

     // keep priority between 18 and 20
     if (nice(pid,0) <= 18)
       nice(pid,5); // increment nice by 5
   }
 }

 pid = fork();
```

```
if (pid == 0) {
  int count = 0;
  int laps = 0;
  while(count<5) {
    // only print after X iterations as prints are expensive
    count++;
    if (laps < X) {
      laps++;
      count = 0;
      printf(1, "HIGH \n");
    }

    // keep priority between 12 and 15
    if (nice(pid,0) <= 12)
      nice(pid,5); // increment nice by 5
  }
}

pid = fork();
if (pid == 0) {
  int count = 0;
  int laps = 0;
  while(count<5) {
    // only print after X iterations as prints are expensive
    count++;
    if (laps < X) {
      laps++;
      count = 0;
      printf(1, "LOW \n");
    }

    // keep priority between 8 and 10
    if (nice(pid,0) <= 8)
      nice(pid,5); // increment nice by 5
  }
}

pid = fork();
if (pid == 0) {
  int count = 0;
  int laps = 0;
  while(count<5) {
```

```
    // only print after X iterations as prints are expensive
    count++;
    if (laps < X) {
      laps++;
      count = 0;
      printf(1, "WORST \n");

    }

    // keep priority between 1 and 5
    if (nice(pid,0) <= 1)
      nice(pid,5); // increment nice by 5
  }
 }

 while(1) {}

 exit();
}
```

OUTPUT of Lottery1.c

```
$ lottery1
lottery test 1
BEST
BEST
BEST
BEST
BEST
HIGH
HIGH
HIGH
HIGH
HIGH
HIGH
HIGH
HIGH
HIGH
HIGH
LOW
LOW
LOW
LOW
LOW
LOW
LOW
LOW
LOW
LOW
LOW
LOW
LOW
LOW
LOW
LOW
LOW
LOW
WORST
```

14) lottery2.c

Create a new test case lottery2.c in user as follows:
Here we make sure that the process with the highest priority gets to run the most also we ensure that the processes which have a priority of 0 still gets to run

```
#include "kernel/types.h"
#include "user.h"
#define X 5

int
main(int argc, char *argv[])
{
 printf(1, "lottery test 2\n");

 int pid;

 int i;
 for (i = 5; i >= 0; i--) {
   pid = fork();
   if (pid == 0) {
     int count = 0;
     int laps = 0;
     while(count < X) {
       count++;
       laps++;
       if (laps < count) {
         printf(1, "%d\n", i);
       }

       if (nice(pid,0) < i)
         nice(pid,1);
     }
   }
 }

 exit();
}
```

Output of Lottery2.c

```
$ lottery2
lottery test 2
5
5
5
5
5
4
4
4
4
4
4
4
4
4
3
3
3
3
3
3
3
3
3
3
3
3
3
2
2
2
2
2
1
1
1
1
0
0
0
0
0
0
0
0
0
0
1
0
0
0
0
0
zombie!
zombie!
zombie!
zombie!
0
0
0
0
zombie!
zombie!
zo0
0
0
0
0
0
mbie!
zombie!
```

15) Lottery3.c

Create a new test case lottery2.c in user as follows:

```
#include "kernel/types.h"
#include "user.h"
#define X 5

int
main(int argc, char *argv[])
{
 printf(1, "lottery test 3\n");

 // 5 processes with equal nice values
 int pid;

 int i;
 for (i = 5; i >= 0; i--) {
   pid = fork();
   if (pid == 0) {
     int count = 0;
     int laps = 0;
     while(count<3) {
       // only print after X iterations as prints are expensive,
       count++;
       if (laps < X) {
         laps++;
         //count = 0;
         printf(1, "%d \n", i);
       }
       // keep priority at i
       if (nice(pid,0) < 5) // keep around 4-6
         nice(pid,2);
       printf(1, "%d\n", i); // process number
     }
   }
 }
  exit();
}
```

Output of lottery3.c

```
$ lottery3
lottery test 3
5 0
5
5 0
5
5 0
5
4 0
4
4 0
4
4 0
4
4 0
4
4 0
4
4 0
4
3 0
3
3 0
3
3 0
3
3 0
3
3 0
3
3 0
3
3 0
3
3 0
3
2 0
2
2 0
2
2 0
2
1 0
1
1 0
1
1 0
1
1 0
1
1 0
1
1 0
1
1 0
1
1 0
1
1 0
1
1 0
1
11 0
1
1 0
1
1 0
1
1 0
1
0 0
0
0 0
0
1 0
zombie!
zombie!
zombie!
1
1 0
1
1 0
zombie!
zombie!
zombie!
0 0
0
0 0
0
0 0
0
1
zombie!
zombie!
zombie!
zombie!
zombie!
zombi0 0
0
0 0
0
0 0
0
e!
zombie!
zombie!
zombie!
zombie!
```