Lottery scheduling

Lottery scheduling is a probabilistic scheduling algorithm for processes in an operating system. Processes are each assigned some **number of lottery tickets**, and the scheduler draws a **random** ticket to select the next process. The distribution of tickets need not be uniform; **granting a process more tickets provides it a relative higher chance of selection**. This technique can be used to approximate other scheduling algorithms, such as Shortest job next and Fair-share scheduling.

The lab consists of implementing this algorithm in minix with the follow modification:
If processes with more than 10 tickets have won the lottery 3 times, the next process to win (to execute) is going to be the process with less tickets in the rdy queue. Then you restart this count (count_wins).

Implementation

1. Create a global variable total_tckts and count_wins in the file /usr/src/kernel/glo.h

```
extern struct kmessages kmessages;        /* diagnostic messages in
kernel */
extern struct loadinfo loadinfo;          /* status of load average */
extern struct minix_kerninfo minix_kerninfo;

EXTERN struct k_randomness krandom;     /* gather kernel random
information */
EXTERN int total_tckts; /*sum of all tickets*/
EXTERN int cont_wins; /* winning process count with tickets > 10*/
vir_bytes minix_kerninfo_user;

#define kmess kmessages
#define kloadinfo loadinfo
```

2. Compile the system: go to /usr/src/releasetools and execute:
   make services
   make install
   make hdboot
   shutdown the machine and restart with default boot option (2)

3. Modify the enqueue function (/usr/src/kernel/proc.c void enqueue) to accumulate the quantity of tickets:

```
assert(proc_is_runnable(rp));
assert(q >= 0);
rdy_head = get_cpu_var(rp->p_cpu, run_q_head);
rdy_tail = get_cpu_var(rp->p_cpu, run_q_tail);
if(rp->tickets>0)
    total_tckts =total_tckts +(rp->tickets);
```

4. Add the process to the queue sorted in **Descending** order (the process with fewer tickets at the end of the queue)

```
. . . .
/* Now add the process to the queue. */
if (!rdy_head[q]) { /* add to empty queue */
    rdy_head[q] = rdy_tail[q] = rp; /* create a new queue */
    rp->p_nextready = NULL; /* mark new end */
}else { /* add to tail of queue */
    struct proc *start = rdy_head[q];
    //Case that needs to be added to the head
    if(){
    }else{//Find position to insert
        // COMPLETE THE IMPLEMENTATION
    }
}
. . . .
```

Hint:
-At this point take a **snapshot** of the machine.
-**Compile and reboot** the machine to check if there is any Error. (/usr/src/releasetools: make services, make hdboot...)

5. To complete the lottery algorithm is necessary to generate random numbers, which is going to be the winner ticket (the owner of the ticket is the process to be execute).
   Implement the following function to **generate random numbers** (at the beginning of the file /usr/src/kernel/proc.c)

```
static void idle(void);
unsigned short lfsr = 0xACE1u;
unsigned bit;

unsigned rand()
{
    bit  = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5) ) & 1;
    return lfsr =  (lfsr >> 1) | (bit << 15);
}
/**
 * Made public for use in clock.c (for user-space scheduling)
   static int mini_send(struct proc *caller_ptr, endpoint_t dst_e,
message
   *m_ptr, int flags);
*/
```

Note: Why stdlib.h can't be imported in proc.c to use its rand function?

6. Modify the **pick_proc** function to select a process based on the lottery and not just the head of the queue.

```
/*===========================================================================*
 *                                pick_proc                                  *
 *===========================================================================*/
static struct proc * pick_proc(void)
{
/* Decide who to run now.  A new process is selected an returned.
 * When a billable process is selected, record it in 'bill_ptr', so that the
 * clock task can tell who to bill for system time.
 * This function always uses the run queues of the local cpu!
 */
```

Hint:

```
idProceso Ganador(){
 int suma = 0;
 int papeletaGanadora;
 idProceso proceso = preparados.primero;            /* se usa la cola de preparados */
 papeletaGanadora = Sorteo(numPapeletasTotales);          /* se realiza el sorteo */
 while ( suma < papeletaGanadora) {                    /* ¿quién tiene la papeleta? */
   suma = suma + tablaDescriptores[proceso].numPapeletas;
   if (suma < papeletaGanadora)
    proceso = tablaDescriptores[proceso].siguiente;
}
 return idProceso                                      /* Este proceso es el ganador */
}
```

Se saca el proceso ganador de la cola de preparados, se decrementa el número de papeletas que tiene asignadas del total.

```
   numPapeletasTotales = numPapeletasTotales - tablaDescriptores[proceso].numPapeletas;
```

This is an algorithm of the lottery scheduler. You need to modify it to meet the new requirements: If processes with more than 10 tickets have won the lottery 2 times, the next process to win (to execute) is going to be the process with less tickets in the rdy queue. Then you restart the counter (count_wins)
Since this is a lottery algorithm, the output could be different for each test but this is an "ideal" Example output:

Assuming that we have in the ready queue:

rdy_queue[q][0]=32

rdy_queue[q][1]=19

rdy_queue[q][2]=18

rdy_queue[q][3]=17

rdy_queue[q][4]=16

rdy_queue[q][5]=15

(Note that the queue is already in Descending order)

As the winner proccess is selected randomly, we cant predict an exact order of execution. But for testing purposes let's assume that the first process (head) always win the lottery (like rand() is always returning 1). Then the execution order should be:

1)rdy_queue[q][0]=32 //count_wins=1

2)rdy_queue[q][1]=19 //count_wins=2

3)rdy_queue[q][2]=15 // count_wins is >= 2 we pick the process with fewer tickets to execute and we restart the counter

4)rdy_queue[q][3]=18 //count_wins=1

5)rdy_queue[q][4]=16 // Againcount_wins is >= 2 we pick the process with fewer tickets to execute

5)rdy_queue[q][5]=17 //count_wins=1