Relatório escalonamento MINIX3

Lucas Daher Santos (RA:114830) e Daniel Barbosa Silva Costa (RA:112185)

Maio 2019

Para a realização dos experimentos, foram escolhidos dois tipos de escalonadores:

- *First in first out* e
- Loteria

Os testes foram realizados em um computador com as seguintes configurações:

- Processador Intel Core I5
- Sistema Operacional Linux Mint versão X
- 8GB de memória ram

O MINIX3(minix_R3.4.0rc6-d5e4fc0.iso.bz2) foi emulado em uma máquina virtual através do software *VirtualBox*, utilizando 1GB de memória ram e 5GB de armazenamento.

Escalonador do tipo *First in first out*

O primeiro escalonador escolhido para a realização dos testes foi o do tipo *First in first out*, utilizado em sistemas em lote, que consiste a medida que os processos vão entrando eles já são executados, em outras palavras, os processos são executados na ordem em que são adicionados na fila.

Para isso, foi modificado o arquivo **"proc.c"** do *kernel* do MINIX3, de tal forma que o processo a ser adicionado na fila seria sempre adicionado a uma mesma fila. Entretanto, foi necessário separar os processos do sistema e do usuário, para isso, os processos do sistema foram mantidos em suas respectivas filas de prioridade (0 a 6), e quando era um processo que não fosse do sistema, era adicionado sempre a uma mesma fila. Dessa forma, temos uma adaptação do escalonamento de múltiplas filas de prioridade do MINIX3 para o escalonamento do tipo *First in first out*.

```
1
2
3
4  /*==========**enqueue **==========*/
5  void enqueue(
6    register struct proc *rp  /* this process is now runnable */
7  )
8  {
9  /* Add 'rp' to one of the queues of runnable processes.  This function is
10  * responsible for inserting a process into one of the scheduling queues.
11  * The mechanism is implemented here.   The actual scheduling policy is
12  * defined in sched() and pick_proc().
13  *
14  * This function can be used x-cpu as it always uses the queues of the cpu
      the
15  * process is assigned to.
16  */
17
18
19 /*----------Trecho alterado----------*/
20
21
22   int q = 0;
23
24
25   if(rp->p_priority < 7){
26     q = rp->p_priority;
27   }
28   else
29     q = 7;
30
31
32
33   struct proc **rdy_head, **rdy_tail;
34
35   assert(proc_is_runnable(rp));
36
37   assert(q >= 0);
38
39   rdy_head = get_cpu_var(rp->p_cpu, run_q_head);
40   rdy_tail = get_cpu_var(rp->p_cpu, run_q_tail);
41
42   /* Now add the process to the queue. */
43   if (!rdy_head[q]) {    /* add to empty queue */
44       rdy_head[q] = rdy_tail[q] = rp;      /* create a new queue */
45       rp->p_nextready = NULL;    /* mark new end */
46
```

```
47    }
48    else {          /* add to tail of queue */
49        rdy_tail[q]->p_nextready = rp;     /* chain tail of queue */
50        rdy_tail[q] = rp;        /* set new queue tail */
51        rp->p_nextready = NULL;    /* mark new end */
52    }
53
54    if (cpuid == rp->p_cpu) {
55      /*
56       * enqueueing a process with a higher priority than the current one,
57       * it gets preempted. The current process must be preemptible. Testing
58       * the priority also makes sure that a process does not preempt itself
59       */
60      struct proc * p;
61      p = get_cpulocal_var(proc_ptr);
62      assert(p);
63      if((p->p_priority > rp->p_priority) &&
64          (priv(p)->s_flags & PREEMPTIBLE))
65        RTS_SET(p, RTS_PREEMPTED); /* calls dequeue() */
66    }
67 #ifdef CONFIG_SMP
68    /*
69     * if the process was enqueued on a different cpu and the cpu is idle, i.
       e.
70     * the time is off, we need to wake up that cpu and let it schedule this
       new
71     * process
72     */
73    else if (get_cpu_var(rp->p_cpu, cpu_is_idle)) {
74      smp_schedule(rp->p_cpu);
75    }
76 #endif
77
78    /* Make note of when this process was added to queue */
79    read_tsc_64(&(get_cpulocal_var(proc_ptr)->p_accounting.enter_queue));
80
81
82 #if DEBUG_SANITYCHECKS
83    assert(runqueues_ok_local());
84 #endif
85 }
```

Escalonador do tipo Loteria

No escalonamento por loteria, primeiramente foi feita um alteração no arquivo *sched-proc.h*, a fim de alterar a estrutura do processo a ser escalonado. Tal alteração teve como

base o fato de, no arquivo *schedule.c*, a função *do_start_schedulind (message *m_ptr)* utilizar como parâmetro para escalonar uma estrutura do tipo *schedproc*, no caso *register struct schedproc *rmp*. A alteração consistiu na inserção de um vetor de inteiros, os quais conterão os valores dos *tickets par ao sorteio*.

Alteração na estrutura do processo.

```
1
2  /* This table has one slot per process.  It contains scheduling information
3   * for each process.
4   */
5  #include <limits.h>
6
7  #include <minix/bitmap.h>
8
9  /* EXTERN should be extern except in main.c, where we want to keep the
       struct */
10 #ifdef _MAIN
11 #undef EXTERN
12 #define EXTERN
13 #endif
14
15 #ifndef CONFIG_SMP
16 #define CONFIG_MAX_CPUS 1
17 #endif
18
19 /**
20  * We might later want to add more information to this table, such as the
21  * process owner, process group or cpumask.
22  */
23
24 EXTERN struct schedproc {
25   endpoint_t endpoint;  /* process endpoint id */
26   endpoint_t parent;  /* parent endpoint id */
27   unsigned flags;   /* flag bits */
28
29   /* User space scheduling */
30   unsigned max_priority;  /* this process' highest allowed priority */
31   unsigned priority;    /* the process' current priority */
32   unsigned time_slice;    /* this process's time slice */
33   unsigned cpu;   /* what CPU is the process running on */
34   bitchunk_t cpu_mask[BITMAP_CHUNKS(CONFIG_MAX_CPUS)]; /* what CPUs is the
35                 process allowed
36 to run on */
37   int tickets[16];     //item adicionado
38 } schedproc[NR_PROCS];
39
40 /* Flag values */
```

```
41 #define IN_USE      0x00001 /* set when 'schedproc' slot in use */
```

A função de sorteio foi criada no arquivo *schedule.c*. Beaseando-se na última função do arquivo, *void balance_queue(void)*, a qual busca e recoloca um processo em uma fila de prioridade caso esse tenha sido colocado abaixo da prioridade da fila de menor prioridade, considerou-se então que a função passa por todos os processos. Com isso foi utilizado o mesmo *for* da função *void balance_queue(void)* retirando-se apenas a condicional interna.

Na função criada, *struct schedproc * lottery(void)*, há o retorno de uma estrutura do tipo *schedproc*. Há também criação de um valor para conter o valor do ultimo *ticket* sorteado. Primeiramente a função inicia o *tickets* dos processos com valor -1, em seguida, distribui uma quantidade de *tickets* igual à 16 menos sua prioridade, ou seja, processos de maior prioridade receberão 16 e processos com a menor prioridade receberão apenas 1. Após isso, um valor de aleatório de *ticket* é sorteado com base na quantidade de *tickets* distribuídos, assim, o processo que contiver o sorteado, será repassado como retorno.

```
1
2  /* --------- Funcao de sorteio ---------------*/
3
4
5  struct schedproc * lottery(void){
6    int long ticket_count=0;
7    int i=0;
8    struct schedproc * rmp;
9    int proc_nr=0;
10
11     for (proc_nr=0, rmp=schedproc; proc_nr < NR_PROCS; proc_nr++,
12 rmp++) {
13     if (rmp->flags & IN_USE) {
14       for (i=0;i<16;i++) {
15         rmp->tickets[i] = -1;
16       }
17     }
18   }
19
20
21     for (proc_nr=0, rmp=schedproc; proc_nr < NR_PROCS; proc_nr++,
22 rmp++) {
23     if (rmp->flags & IN_USE) {
24       for (i=0;i<16-rmp->priority;i++) {
25         rmp->tickets[i] = ticket_count;
26         ticket_count++;
27       }
28     }
29   }
30   int winner=0;
31   if(ticket_count>0){
```

```
32      winner=(random()%ticket_count)-1;
33      if (winner<0) winner=0;
34      for (proc_nr=0, rmp=schedproc; proc_nr < NR_PROCS; proc_nr++,
35  rmp++) {
36      if (rmp->flags & IN_USE) {
37        for(i=0;i<16;i++){
38          if(rmp->tickets[i]==winner){
39            return rmp;
40          }
41        }
42      }
43    }
44
45    }
46    return NULL;
47  }
48
49  static void pick_cpu(struct schedproc * proc)
50  {
51  #ifdef CONFIG_SMP
52    unsigned cpu, c;
53    unsigned cpu_load = (unsigned) -1;
54
55    if (machine.processors_count == 1) {
56      proc->cpu = machine.bsp_id;
57      return;
58    }
59
60    /* schedule sysytem processes only on the boot cpu */
61    if (is_system_proc(proc)) {
62      proc->cpu = machine.bsp_id;
63      return;
64    }
65
66    /* if no other cpu available, try BSP */
67    cpu = machine.bsp_id;
68    for (c = 0; c < machine.processors_count; c++) {
69      /* skip dead cpus */
70      if (!cpu_is_available(c))
71        continue;
72      if (c != machine.bsp_id && cpu_load > cpu_proc[c]) {
73        cpu_load = cpu_proc[c];
74        cpu = c;
75      }
76    }
77    proc->cpu = cpu;
78    cpu_proc[cpu]++;
```

```
#else
  proc->cpu = 0;
#endif
}
```

Em *int do_start_scheduling(message *m_ptr)* houve apenas uma alteração em qual será o processo selecionado, sendo esse o retorno de *lottery()* e reinicialização dos *tickets* do processo escolhido.

```


/*===========*do_start_scheduling*===========*/
int do_start_scheduling(message *m_ptr)
{
  register struct schedproc *rmp;
  int rv, proc_nr_n, parent_nr_n;

  /* we can handle two kinds of messages here */
  assert(m_ptr->m_type == SCHEDULING_START ||
    m_ptr->m_type == SCHEDULING_INHERIT);

  /* check who can send you requests */
  if (!accept_message(m_ptr))
    return EPERM;

  /* Resolve endpoint to proc slot. */
  if ((rv = sched_isemtyendpt(m_ptr->m_lsys_sched_scheduling_start.endpoint
    ,
      &proc_nr_n)) != OK) {
    return rv;
  }


  /*---------------Trecho alterado---------*/


  rmp = lottery();

  int ix=0;

  for(ix=0;ix<16;ix++){
    rmp->tickets[ix]=-1;
  }

  /* Populate process slot */
  rmp->endpoint     = m_ptr->m_lsys_sched_scheduling_start.endpoint;
  rmp->parent       = m_ptr->m_lsys_sched_scheduling_start.parent;
```

```
39   rmp->max_priority = m_ptr->m_lsys_sched_scheduling_start.maxprio;
40   if (rmp->max_priority >= NR_SCHED_QUEUES) {
41     return EINVAL;
42   }
43
44   /* Inherit current priority and time slice from parent. Since there
45    * is currently only one scheduler scheduling the whole system, this
46    * value is local and we assert that the parent endpoint is valid */
47   if (rmp->endpoint == rmp->parent) {
48     /* We have a special case here for init, which is the first
49        process scheduled, and the parent of itself. */
50     rmp->priority   = USER_Q;
51     rmp->time_slice = DEFAULT_USER_TIME_SLICE;
52
53     /*
54      * Since kernel never changes the cpu of a process, all are
55      * started on the BSP and the userspace scheduling hasn't
56      * changed that yet either, we can be sure that BSP is the
57      * processor where the processes run now.
58      */
59 #ifdef CONFIG_SMP
60     rmp->cpu = machine.bsp_id;
61     /* FIXME set the cpu mask */
62 #endif
63   }
64
65   switch (m_ptr->m_type) {
66
67   case SCHEDULING_START:
68     /* We have a special case here for system processes, for which
69      * quanum and priority are set explicitly rather than inherited
70      * from the parent */
71     rmp->priority   = rmp->max_priority;
72     rmp->time_slice = m_ptr->m_lsys_sched_scheduling_start.quantum;
73     break;
74
75   case SCHEDULING_INHERIT:
76     /* Inherit current priority and time slice from parent. Since there
77      * is currently only one scheduler scheduling the whole system, this
78      * value is local and we assert that the parent endpoint is valid */
79     if ((rv = sched_isokendpt(m_ptr->m_lsys_sched_scheduling_start.parent,
80         &parent_nr_n)) != OK)
81       return rv;
82
83     rmp->priority = schedproc[parent_nr_n].priority;
84     rmp->time_slice = schedproc[parent_nr_n].time_slice;
85     break;
```

```
86
87    default:
88      /* not reachable */
89      assert(0);
90    }
91
92    /* Take over scheduling the process. The kernel reply message populates
93     * the processes current priority and its time slice */
94    if ((rv = sys_schedctl(0, rmp->endpoint, 0, 0, 0)) != OK) {
95      printf("Sched: Error taking over scheduling for %d, kernel said %d\n",
96        rmp->endpoint, rv);
97      return rv;
98    }
99    rmp->flags = IN_USE;
100
101   /* Schedule the process, giving it some quantum */
102   pick_cpu(rmp);
103   while ((rv = schedule_process(rmp, SCHEDULE_CHANGE_ALL)) == EBADCPU) {
104     /* don't try this CPU ever again */
105     cpu_proc[rmp->cpu] = CPU_DEAD;
106     pick_cpu(rmp);
107   }
108
109   if (rv != OK) {
110     printf("Sched: Error while scheduling process, kernel replied %d\n",
111       rv);
112     return rv;
113   }
114
115   /* Mark ourselves as the new scheduler.
116    * By default, processes are scheduled by the parents scheduler. In case
117    * this scheduler would want to delegate scheduling to another
118    * scheduler, it could do so and then write the endpoint of that
119    * scheduler into the "scheduler" field.
120    */
121
122   m_ptr->m_sched_lsys_scheduling_start.scheduler = SCHED_PROC_NR;
123
124   return OK;
125 }
```

Resultados

Utilizando o arquivo disponibilizado pelo professor, para todos os escalonamentos testados, foram feitos testes em 10, 20 e 50 processos, ambos com metade dos processos de *IO bound* executarão 50000 de operações de saída. A outra metade dos processos, de

CPU bound, executarão 1000000000(1 bilhão) de operações aritméticas na *CPU*. Também foi testado o escalonador padrão do sistema operacional para fins comparativos, como veremos a seguir.

A tabela da figura a seguir mostra a média do tempo de execução dos processos para cada tipo de escalonador.

Figura 1 – Dados obtidos

Média dos tempos						
	Quantidade de processos					
Tipo de escalonamento	5 I/O	5 CPU	10 I/O	10 CPU	25 I/O	25 CPU
Padrão	127,133	417,7636	253,3035	422,5367	637,84468	419,75
First in first out	125,0056667	471,5	249,9783	237,6235	628,56796	317,345252
Loteria	126,6366	502,9996	253,1417	240,6716	646,60528	406,97468

Observa-se que o escalonamento do tipo *First in first out* a medida que o número de processos foi aumentando, o tempo obviamente também aumentou, mas aumentou a uma taxa menor comparado com o escalonador padrão do MINIX3. O de loteria, obteve uma variação diferenciada, mas que obteve resultados melhores do que o escalonador padrão para valores mais alto de processos.

Considerando os resultados obtidos, não deve-se julgar qual é o melhor tipo de escalonador no geral, mas sim qual o melhor escalonador para a sua aplicação. Nesse exercício, utilizamos processos simples para testar os escalonadores, e com isso, o escalonador do tipo *First in first out* provavelmente sempre terá o melhor desempenho. Em outros tipos de aplicações, com diferentes tipos de processos, talvez a melhor opção seja o de Loteria. Assim, observa-se que o escalonador padrão do MINIX3 não é específico para uma determinada aplicação, mas sim tenta obter um melhor resultado baseado nos diferentes tipos de uso que se pode fazer do sistema operacional. Também a quantidade de processos testados foi muito pouca, devido ao tempo muito longo para execução dos mesmo, optamos por testar poucos processos mas de forma a extrair algumas informações curiosas.