Trabalho 2 de Laboratório de Sistemas Operacionais

Lucas Antunes, Lorenzo Windmoller, Alison Carnetti

18 de Outubro de 2023

1 Introdução

Nestre trabalho foi implementado um módulo escalonador de disco que implementa a política SSTF (Shortest Seek Time First). Para tal, foi utilizado como base o código disponibilizado em https://github.com/miguelxvr/sstf-iosched-skeleton.git.

O código fonte do trabalho está disponível em https://github.com/ItsaMeTuni/labsisop-buildroot/tree/t2.

2 Funcionamento do módulo

O módulo SSTF é idêntico ao código base disponibilizado, exceto pela função `sst_dispatch`, que escolhe qual request despachar ao kernel; pelo `struct sstf_data`, que armazena a lista de requests pendentes e o setor do último request; e pelo `sstf_init_queue`, que foi alterado para inicializar o `sstf_data.last_sector`.

```
struct sstf_data {
  struct list_head queue;
  unsigned long long last_sector;
};

// ...

static int sstf_dispatch(struct request_queue *q, int force){
  struct sstf_data *nd = q->elevator->elevator_data;
  char direction = 'R';
  struct request *rq;

  struct request *closest = NULL;
  long long closest_dist = 0;

  struct request *curr;
  list_for_each_entry(curr, &nd->queue, queuelist) {
    long long curr_dist = blk_rq_pos(curr) - nd->last_sector;
    if (curr_dist < 0) {
      curr_dist = curr_dist * -1;
    }
```

```
22
23      if (closest == NULL || curr_dist < closest_dist) {
24        closest = curr;
25        closest_dist = curr_dist;
26      }
27    }
28
29    if (closest) {
30      nd->last_sector = blk_rq_pos(closest);
31
32      list_del(&closest->queuelist);
33      elv_dispatch_sort(q, closest);
34
35      printk(KERN_EMERG "[SSTF] dsp %c %llu\n", direction, blk_rq_pos
        (closest));
36
37      return 1;
38    }
39
40    return 0;
41  }
42
43  // ...
44
45  static int sstf_init_queue(struct request_queue *q, struct
        elevator_type *e){
46    struct sstf_data *nd;
47    struct elevator_queue *eq;
48
49    eq = elevator_alloc(q, e);
50    if (!eq)
51      return -ENOMEM;
52
53    nd = kmalloc_node(sizeof(*nd), GFP_KERNEL, q->node);
54    if (!nd) {
55      kobject_put(&eq->kobj);
56      return -ENOMEM;
57    }
58    eq->elevator_data = nd;
59
60    INIT_LIST_HEAD(&nd->queue);
61    nd->last_sector = 0;
62
63    spin_lock_irq(q->queue_lock);
64    q->elevator = eq;
65    spin_unlock_irq(q->queue_lock);
66
67    return 0;
68  }
```

Em nossa implementação, os requests são adicionados à fila em ordem de chegada por meio do `sstf_add_request`. Quando o `sstf_dispatch` é chamado pelo kernel para obter o próximo request na fila para processamento, ele lê a fila de requests e procura o que tiver o setor com menor distância relativo ao `last_sector`. Assim que ele escolher o request a despachar, ele armazena o setor do request no `last_sector` e despacha o request.

3 Resultados

Escrevemos um breve script em Python para ler a saída do escalonador, calcular quantos setores foram percorridos e gerar gráficos mostrando a ordem que os requests chegaram e a ordem em que foram processados, facilitando a visualização dos resultados obtidos. Sob as mesmas condições (120 processos, cada um lendo 500 blocos de 100 bytes), foram obtidos os seguintes resultados:

Escalonador	Setores percorridos	Tempo de execução
sstf	2.101.472	7159.78ms
noop	19.742.850	7370.44ms

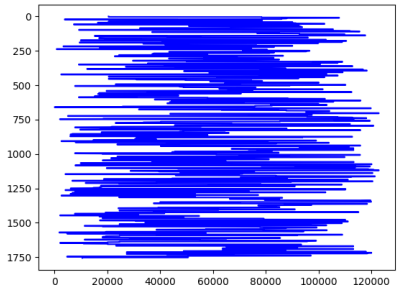

Figure 1: Ordem de chegada

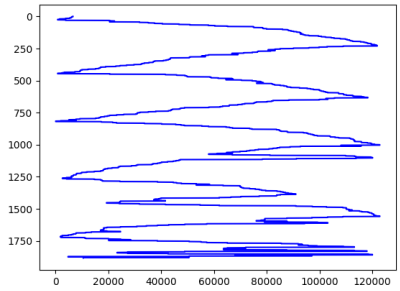

Figure 2: Ordem de processamento

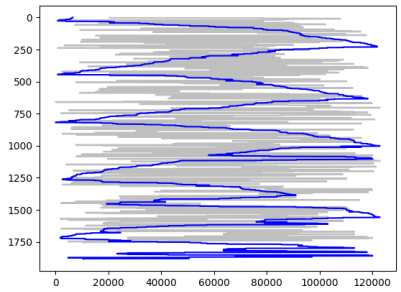

Figure 3: Ordem de chegada (cinza) e ordem de processamento (azul)