Implementação de escalonamento por loteria no *xv6* com análise de desempenho

João Pedro Winckler Bernardi[1]

[1]Universidade Federal da Fronteira Sull (UFFS)
89802-260 – Chapecó – SC – Brasil

jpwbernardi@hotmail.com

Abstract. *This paper presents lottery and stride scheduling implementations on xv6 operating system and analyzes if works as required. In the schedulers' implementations were used the data structs Binary Indexed Tree – BIT – and Segment Tree, both will be explained in this paper and from them we obtain a complexity analyzis.*

Resumo. *Este trabalho descreverá a implementação de dois processos de escalonamento distintos no sistema operacional xv6, o escalonamento por loteria (lottery scheduling) e o escalonamento em passos largos (stride scheduling), com as respectivas análises de funcionamento. Nas implementações são apresentadas as estruturas de dados BIT – Binary Indexed Tree – e árvore de segmentos e, apartir dela, obtemos a complexidade de cada uma das soluções.*

1. Introdução

Os sistemas operacionais atuais executam vários processos ao mesmo tempo, ou, pelo menos, é essa a impressão que temos. Um processador não consegue executar mais de um processo simultaneamente. Então, todo processo pronto para ser executado está competindo para ser executado pelo processador. O escalonador é o responsável pela escolha do processo a ser executado. O método que o escalonador vai usar para escolher o próximo processo é chamado de algoritmo de escalonamento [Tanenbaum 2010].

Seja n a quantidade máxima de processos que um sistema operacional pode executar, esse relatório apresenta a implementação e o funcionamento do escalonamento por loteria com complexidade de decisão de $O(\log^2 n)$ e do escalonamento em passos largos com complexidade de decisão de $O(\log n)$ no *xv6*, um sistema operacional didático de código aberto.

2. Planejamento do escalonamento por loteria

Nesse algoritmo de escalonamento, cada processo recebe uma quantidade de bilhetes, então sorteia-se um bilhete e o processo dono daquele bilhete é executado. Os processos com mais bilhetes tem mais chance de serem executados e, dado o tempo necessário, todo processo será executado.

Controlaremos a quantidade de bilhetes de cada processo através de um vetor de acúmulos. Esse vetor contém na posição i a soma da quantidade de bilhetes do processo i com a quantidade de bilhetes de todos os processos anteriores a i.

Por exemplo, seja A, B, C, D, E processos com 10, 20, 5, 1 e 13 bilhetes, respectivamente, em estado *RUNNABLE*, ou seja, todos prontos para serem escolhidos pelo escalonador. Então, temos 49 bilhetes que podem ser sorteados e o vetor estaria conforme a Tabela 1.

Tabela 1. Vetor de acúmulo de bilhetes 1

A	B	C	D	E
10	30	35	36	49

Isso significa que os bilhetes de A estão entre 1 e 10, os de B entre 11 e 30, os de C entre 31 e 35, o de D é 36 e os de E entre 37 e 49. Então, suponhamos que o bilhete sorteado foi o 15, ou seja, o processo B será executado. Se, durante sua execução, ele foi bloqueado, B não pode mais ser executado enquanto não estiver no estado pronto. Então, retiramos os bilhetes de B. O vetor de acúmulos estaria conforme a Tabela 2.

Tabela 2. Vetor de acúmulo de bilhetes 2

A	B	C	D	E
10	10	15	16	29

Agora, os bilhetes de A estão entre 1 e 10, B não tem bilhete, os bilhetes de C estão entre 11 e 15, o bilhete de D é 16 e os bilhetes de E estão entre 17e 29. Quando B estiver novamente no estado pronto (*RUNNABLE* no *xv6*), seus bilhetes são devolvidos e o vetor voltaria a ficar conforme a Tabela 1.

Com o vetor dessa forma, temos sempre um intervalo bem definido sobre o qual podemos sortear um bilhete. Se tivéssemos atribuído números fixos aos bilhetes, por exemplo, se um processo x sempre possuísse os bilhetes de 15 a 30, quando esse processo fosse bloqueado, teríamos uma falha no intervalo de bilhetes a serem sorteados e teríamos que tratar para o sorteio não considerar essa falha.

O próximo problema consiste em fazer o acúmulo de bilhetes de forma eficiente. Seja n a quantidade máxima de processos permitidos no sistema operacional, a forma ingênua para cálcular os valores das posições do vetor de acúmulo é percorrer as n posições do vetor e atualizar cada posição i com a quantidade de bilhetes do processo correspondente àquela posição somado com o acúmulo da posição $i-1$, uma complexidade de $O(n)$ para cada vez que fosse necessário atualizar a quantidade de bilhetes um processo. Porém, existe uma estrutura de dados chamada BIT – Binary Indexed Tree – que faz essa operação com complexidade $O(\log n)$ [Halim and Halim 2013]. Essa estrutura será explicada futuramente.

Por fim, temos que buscar o processo com o bilhete sorteado. A estratégia ingênua é percorrer linermente o vetor de acúmulo e a primeira posição que tiver acúmulo maior ou igual ao bilhete sorteado é a posição que corresponde processo dono do bilhete. Porém, a complexidade novamente é $O(n)$. Como sabemos que os acúmulos estão em ordem não decrescente, podemos usar uma busca binária para encontrar o processo com o bilhete sorteado.

3. Implementação e análise do escalonamento por loteria

3.1. Busca Binária

A primeira implementação foi da busca binária, que utiliza a estratégia de divisão e conquista. Começa-se considerando o intervalo de 1 a NPROC. Analisa-se a posição m do meio do intervalo que estou considerando, se o acúmulo até a posição m for maior ou igual ao bilhete sorteado, repito o processo considerando m o final do meu intervalo, caso contrário repito o processo considerando m o inicio do meu intervalo. Isso resulta numa complexidade de $O(\log n)$. Segue o código da busca, encontrado no arquivo `proc.c`.

```
int bsearch(int ticket){
  int l = 1, h = NPROC, m;
  while (l < h) {
      m = l + (h - l) / 2;
      if (ticount(m) >= ticket) h = m;
      else l = m + 1;
  }
  return l - 1;
}
```

Para facilitar a busca, modificamos o código do *xv6* para que o pid do processo correspondesse a sua posição no vetor de acúmulos e $pid - 1$ fosse sua posição no vetor de processos. Por isso a busca retorna $l - 1$. Essas mudanças vão ser melhor comentadas futuramente.

3.2. Binary Indexed Tree(BIT)

Inventada por Peter M. Fenwick em 1994, a BIT é uma estrutura de dados simples para implementar tabelas de frequências cumulativas. A implementação foi feita através de um vetor, onde cada posição guarda um acumulo parcial. Na implementação, a BIT é representada pelo vetor `stickets`. O vetor `idstack` é uma pilha estática que guarda as posições da BIT não associadas a processos, ou seja, os pids disponíveis que agora variam de 1 ao número de processos, e tp é a quantidade de pids disponíveis. Segue o novo código da estrutura `ptable`.

```
struct {
  int stickets[NPROC + 1];
  int idstack[NPROC], tp;
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

De uma forma mais genérica, o elemento da posição i da BIT `stickets` é responsável pelos elementos no intervalo $[i - (i \texttt{ AND } -i) + 1,\ i]$ e, portanto, $stickets[i]$ guarda o acumulo dos bilhetes $i - (i \texttt{ AND } -i) + 1$, $i - (i \texttt{ AND } -i) + 2$, $i - (i \texttt{ AND } -i) + 3$, ... , i.

Para obtermos o acúmulo de bilhetes de uma posição, usamos a função `ticount`. Para atualizarmos uma posição da BIT com algum valor, usamos a função `uptick`.

```
int ticount(int i){
  int count = 0;
  for (; i > 0; i -= i & -i)
    count += ptable.stickets[i];
  return count;
}

void uptick(int i, int value){
  for (; i <= NPROC; i += i & -i)
    ptable.stickets[i] += value;
}
```

3.3. Mudanças no código do *xv6*

Inicialmente, o pid de um processo era definido por uma variável global. Toda vez que um processo recebia um pid, a variável global era incrementada. Porém, como explicado anteriormente, agora o pid representa o índice da BIT associado ao processo. Toda vez que um processo termina, ele devolve seu pid para a pilha `idstack`. Como cada processo tem um pid único que varia de 1 ao número máximo de processos, então podemos usar o $pid - 1$ para a indexação do processo no vetor de processos. Antes, quando um processo era criado, o vetor de processos era percorrido linearmente até encontrar uma posição disponível, uma complexidade de $O(n)$. Agora, só retiramos da pilha um pid disponível, o que é realizado em tempo $O(1)$. Abaixo há o código dessa alteração que se encontra no arquivo `proc.c`.

```
static struct proc* allocproc(void){
 ...
  acquire(&ptable.lock);
  if (ptable.tp > 0) {
    i = ptable.idstack[--ptable.tp];
    p = &ptable.proc[i - 1];
    goto found;
  }
  release(&ptable.lock);
  return 0;
found:
  p->pid = i;
 ...
  return p;
}
```

Toda vez que o *xv6* é inicializado, a função `clean` é chamada. Ela é responsável por inicializar a pilha, ou seja, colocar todas os *pid* na pilha, e zerar a BIT.

```
void clean() {
  acquire(&ptable.lock);
  for (ptable.tp = 0; ptable.tp < NPROC; ptable.tp++)
    ptable.idstack[ptable.tp] = NPROC - ptable.tp;
  memset(ptable.stickets, 0, sizeof (ptable.stickets));
```

```
  release(&ptable.lock);
}
```

A função `scheduler` é a responsável pela mudança do processo em execução. Primeiro verificamos a quantidade de bilhetes disponíveis para serem soreados. Se essa quantidade for diferente de 0, ou seja, houver algum bilhete para ser sorteado, sorteamos um bilhete entre 1 e a quantidade de bilhetes. A busca binária retorna a posição no vetor `ptable.proc` do processo que tinha o bilhete sorteado. Então, retiramos os bilhetes desse processo e mudamos seu estado para *RUNNING*.

```
void scheduler(void){
  ...
  for(;;){
    sti();
    acquire(&ptable.lock);
    if ((qttytickets = ticount(NPROC)) != 0) {
      p = &ptable.proc[bsearch(rand() % qttytickets + 1)];
      if (p->state == RUNNABLE) {
        uptick(p->pid, -p->tickets);
        proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&cpu->scheduler, proc->context);
        switchkvm();
        proc = 0;
      }}
    release(&ptable.lock);
  }}
```

Os bilhetes do processo são devolvidos quando o estado do processo volta a estar pronto (*RUNNABLE*). Isso acontece em três situações: quando o tempo do processo no processador termina, quando o processo deixa de estar bloqueado e quando um processo que estava bloqueado é morto. Isso é feito nas funções `yield`, `wakeup1` e `kill`, todas encontradas no arquivo `proc.c`.

```
void yield(void){
  acquire(&ptable.lock);
  proc->state = RUNNABLE;
  uptick(proc->pid, proc->tickets);
  sched();
  release(&ptable.lock);
}
...
static void wakeup1(void *chan){
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if (p->state == SLEEPING && p->chan == chan) {
      p->state = RUNNABLE;
      uptick(p->pid, p->tickets);
```

```
    }
}
...
int kill(int pid){
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      if(p->state == SLEEPING){
        uptick(p->pid, p->tickets);
        p->state = RUNNABLE;
      }
      release(&ptable.lock);
      return 0;
    }}
  release(&ptable.lock);
  return -1;
}
```

3.4. Análise de desempenho

Toda vez que o escalonador busca um processo para ser executado, é chamado a busca binária, que tem complexidade $O(\log n)$, porém, para acessar o valor do acúmulo numa posição i, a complexidade é também $O(\log n)$ [Halim and Halim 2013]. Ou seja, a complexidade total para escolher um processo seria $O(\log n \cdot \log n) = O(\log^2 n)$. Nas situações já citadas em que um processo ganha ou perde tickets há o custo de atualização da BIT, $O(\log n)$.

3.5. Análise de testes

Para testar o funcionamento do que foi implementado, criamos um arquivo $schedtest$ no $xv6$. Quando executado, cria o máximo de processos possíveis. Cada processo decrementa uma variável que começa em aproximadamente 10^8 e, quando essa variável chega a 0, o processo acaba. Para obter os resultados do teste, foi modificado a função $exit()$ para que quando ela fosse chamada, exibisse a quantidade de tickets do processo que acabou. Cada processo tem uma quantidade de bilhetes diferentes. Esse teste foi realizado 10 vezes com processos de mesma quantidade de bilhete. A quantidade de bilhetes de cada processo é $n^o doprocesso \cdot 64 + 1$.

O resultado obtido está apresentado na Tabela 3. Cada coluna representa um teste e cada linha representa a ordem que o processo acabou. Por exemplo, no teste 1, o processo 38 foi o 6º processo a terminar e no teste 7 foi o 27º.

Como podemos observar, um processo ter mais bilhetes não é garantia de que ele vai ganhar mais tempo no processador. Possívelmente pela pouca diferença de bilhetes entre cada processo, pelo $rand$ que foi implementado para os testes e pelos processos serem consideravelmente rápidos, isso se tornou mais evidente. Um exemplo disso é o processo 60 que foi o 26º a terminar no teste 4, embora possuísse mais bilhetes que qualquer outro processo, e nunca terminou por primeiro.

Teste 1	Teste 2	Teste 3	Teste 4	Teste 5	Teste 6	Teste 7	Teste 8	Teste 9	Teste 10
54	49	36	32	44	42	50	38	36	39
40	51	58	46	41	55	53	36	52	42
59	52	59	42	45	59	58	40	45	48
47	41	44	44	52	37	44	33	42	38
44	55	51	57	50	57	48	44	37	50
38	38	53	29	46	53	52	42	47	45
49	46	35	45	42	33	49	43	41	51
48	44	55	59	48	49	42	28	59	55
35	57	48	36	38	44	37	45	38	47
50	43	39	38	43	58	56	37	32	52
60	60	54	34	49	47	60	48	40	53
52	58	38	40	54	46	45	55	51	58
32	39	47	35	47	40	41	30	55	43
42	50	60	27	58	51	59	47	58	41
55	48	43	52	53	45	34	60	39	49
57	37	45	41	37	48	46	58	60	37
41	45	41	50	59	43	54	50	49	31
58	59	50	51	40	41	51	39	31	27
51	35	46	31	31	32	40	49	50	46
37	47	29	47	56	36	55	27	44	40
45	54	42	53	55	50	38	53	56	30
53	33	26	37	57	34	30	32	53	44
56	56	56	54	60	30	36	54	54	54
36	42	24	39	34	56	57	35	29	33
33	40	57	30	36	60	39	52	57	57
30	32	34	60	39	54	43	56	43	35
43	53	33	58	23	39	47	29	27	60
39	34	49	43	28	52	35	41	48	32
31	26	52	26	51	25	25	57	46	36
29	30	23	55	29	35	26	34	33	56
46	29	40	23	35	26	29	46	34	25
28	28	30	48	27	31	31	51	25	34
21	27	37	49	19	38	32	59	28	59
34	36	32	25	26	23	33	31	24	26
22	24	28	56	24	29	27	21	35	21
27	25	19	33	33	24	21	25	22	28
26	31	31	21	32	18	28	26	23	20
23	23	22	28	22	28	23	23	26	23
25	22	25	17	30	27	24	22	30	24
20	21	27	22	25	20	22	24	21	29
24	19	14	24	18	21	19	19	19	17
19	14	16	19	21	13	20	17	18	19
18	20	15	20	15	17	18	20	20	22
15	18	20	18	20	22	15	18	17	14
17	16	18	16	16	19	17	16	15	18
16	15	17	15	13	16	16	15	16	15
13	13	21	11	17	15	14	12	14	16
14	12	13	13	14	14	12	11	13	12
11	17	11	10	12	12	13	14	12	13
10	8	12	12	11	11	10	13	10	10
12	11	9	14	9	10	11	10	11	9
9	9	10	8	8	9	8	8	9	11
8	10	8	7	7	8	7	9	8	8
6	5	7	9	10	6	9	7	6	7
7	7	6	6	6	7	6	5	7	6
5	6	5	5	5	5	5	6	5	5
4	4	4	4	4	4	4	4	4	4
3	3	3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	1	1

Tabela 3. Resultados

4. Planejamento do escalonamento em passos largos

Da mesma forma que o escalonamento por loteria, cada processo recebe um número de bilhetes. Então, o passo desse processo é uma constante dividida pelo número de bilhetes do processo. Todo processo inicia com distância 0. O escalonador escolhe o processo com menor distancia para ser executado. Após ser selecionado, a distância do processo aumenta no seu valor do passo. A constante escolhida para essa implementação foi 10^4.

A primeira dificuldade encontrada foi como descobrir que processo tem a menor distância de forma eficiente. A solução ingênua seria percorrer as distâncias linearmente e verificar qual processos tem a menor, com uma complexidade de $O(N)$. Porém, podemos enxergar essas distâncias como a prioridade de cada processo e, assim, montarmos uma fila com prioridade. Existem várias formas de implementação, foi escolhido implementar a fila de prioridade com uma árvore de segmentos onde podemos descobrir o processo com menor prioridade em $O(1)$ e o custo para atualizar as prioridades é $O(logn)$.

Uma árvore de segmentos, neste trabalho, é uma árvore binária balanceada em que cada nodo guarda o processo de menor distância entre seus filhos. Cada folha da árvore corresponde a uma posição no vetor de processos.

Um problema dessa solução é que todo processo, independentemente de poder ser executado ou não, está na árvore. A solução elaborada foi adicionar uma posição sentinela ao vetor de processos com a distância igual a `INF`$(2^{64}-1)$. Quando um processo não pode ser executado, atualizamos sua passada para `INF` e, no critério de desempate, o processo com menor *pid* é escolhido. Ou seja, sabemos que não há processos a serem executados quando o processo de menor prioridade é o sentinela com *pid* 0.

5. Implementação e análise do escalonamento em passos largos

5.1. Árvore de segmentos

Para podermos trabalhar com o sentinela e ainda mantermos NPROC processos, foi adicionada uma posição ao vetor de processos. Uma consequência direta foi que o *pid* de um processo agora poderia representar a posição dele no vetor de processos. Deve-se manter em mente que as mudanças realizadas no *xv6* citadas anteriormente foram mantidas. Além disso, adicionamos dois novos vetores a estrutura `ptable`, `st` para guardar a árvore e `stride` para guardar as distâncias de cada processo. E, cada processo além da quantidade de tickets agora tem uma variavel auxiliar para guardar a distância deles. A estrutura `ptable` está no arquivo `proc.c` e a estrutura do processo no arquivo `proc.h`. `ull` é um *define* encontrado no arquivo `stride.h` para abreviação de *unsigned long long*.

```
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  ...
  int tickets;
  ull prevstride;
};

struct {
```

```
  ull stride[NPROC + 1];
  ull st[4 * (NPROC + 1)];
  int idstack[NPROC], tp;
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

Antes de inicializarmos o escalonador ou alocarmos qualquer processo, devemos ter a arvore pronta. Para isso, na função `clean`, já mostrada anteriormente, adicionamos a chamada da função `build` que se encarrega de montar a árvore pela primeira vez. Ela recebe de parâmetros a raiz da árvore e os limites do vetor que estamos considerando, no caso é o vetor de distâncias. Todas os próximos trechos de código podem ser encontrados no arquivo `proc.c`.

```
void clean() {
  acquire(&ptable.lock);
  int i;
  for (ptable.tp = 0; ptable.tp < NPROC; ptable.tp++)
    ptable.idstack[ptable.tp] = NPROC - ptable.tp;
  for (i = 0; i <= NPROC; i++) ptable.stride[i] = INF;
  build(1, 0, NPROC);
  release(&ptable.lock);
}
void build(int p, int l, int r) {
  int m = (l + r) / 2, pidl, pidr;
  if (l == r) { ptable.st[p] = l; return; }
  build(left(p), l, m); build(right(p), m + 1, r);
  pidl = ptable.st[left(p)]; pidr = ptable.st[right(p)];
  ptable.st[p] =
    ptable.stride[pidl] <= ptable.stride[pidr] ? pidl : pidr;
}
```

As outras funções relacionadas à árvore são a `update`, que recebe de parâmetros a raiz da árvore, o intervalo que estamos considerando no nosso vetor de distâncias e *pid* do processo precisa ser atualizado, responsável por atualizar os nodos da árvore, e a `query`, que recebe de parâmetros a raiz da árvore, o intervalo que estamos considerando no nosso vetor de distâncias e o intervalo no qual queremos encontrar o processo de menor distância, responsável por retornar o processo de menor distância no intervalo dado. Neste caso, sempre queremos saber quem tem a menor distância dentre todos os processos, por isso a função `query` poderia simplesmente retornar o processo na raiz da árvore, mas preferimos deixar a função mais completa possível visando modificações futuras no código.

```
int query(int p, int l, int r, int i, int j) {
  int m = (l + r) / 2, pidl, pidr;
  if (i > r || j < l) return 0;
  if (i <= l && j >= r) return ptable.st[p];
  pidl = query(left(p), l, m, i, j);
  pidr = query(right(p), m + 1, r, i, j);
```

```
  return ptable.stride[pidl]
           <= ptable.stride[pidr] ? pidl : pidr;
}

void update(int p, int l, int r, int i) {
  int m = (l + r) / 2, pidl, pidr;
  if (i > r || i < l || (i == l && i == r)) return;
  update(left(p), l, m, i); update(right(p), m + 1, r, i);
  pidl = ptable.st[left(p)]; pidr = ptable.st[right(p)];
  ptable.st[p] =
    ptable.stride[pidl] <= ptable.stride[pidr] ? pidl : pidr;
}
```

A função `scheduler` funciona da mesma forma que no escalonamento por loteria. Se existe processo para ser executado, ou seja, se o *pid* retornado pela busca for diferente do sentila, marcamos esse processo como *RUNNING* e, para ele não ser escolhido novamente por outro CPU, guardamos a distância atual do processo na variável `prevstride` dele, colocamos sua distância como `INF` e atualizamos a sua prioridade na árvore.

```
void scheduler(void) {
  ...
  for(;;){
    sti();
    acquire(&ptable.lock);
    if ((pid = query(1, 0, NPROC, 0, NPROC)) != 0) {
      p = &ptable.proc[pid - 1];
      p->prevstride = ptable.stride[pid];
      ptable.stride[pid] = INF;
      update(1, 0, NPROC, pid);
      proc = p; switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();
      proc = 0;
    }
    release(&ptable.lock);
  }}
```

Por fim, ainda confrome o escalonador por loteria, é necessário devolver a distância a um processo quando ele volta a ser pronto (*RUNNABLE*) nas funções `yield`, `wakeup1` e `kill`, onde devolvemos a distância anterior somado ao passo do processo. A operação é feito mod `INF` para que nenhum processo que possa ser executado tenha valor igual a `INF`.

```
void yield(void) {
  acquire(&ptable.lock);
  proc->state = RUNNABLE;
  ptable.stride[proc->pid] =
```

```
    (proc->prevstride + CONS / proc->tickets) % INF;
  update(1, 0, NPROC, proc->pid);
  sched();
  release(&ptable.lock);
}
static void wakeup1(void *chan) {
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if (p->state == SLEEPING && p->chan == chan) {
      p->state = RUNNABLE;
      ptable.stride[p->pid]
         = (p->prevstride + CONS / p->tickets) % INF;
      update(1, 0, NPROC, p->pid);
    }
}
```

5.2. Análise de desempenho

Toda vez que o escalonador busca um processo para ser executado, é chamado a função `query`, que normalmente tem complexidade $O(\log n)$, porém, como sempre queremos saber apenas sobre a informação da raiz, ela é executada em $O(1)$. Construir a árvore tem complexidade $O(N)$, mas esse custo é pago somente uma vez antes de inicializar o sistema. Para atualizar a árvore, existe um custo de $O(\log n)$. Isso ocorre quando devemos atualizar a distância de um processo [Halim and Halim 2013].

5.3. Análise de testes

Os testes foram realizados também com o arquivo *schedtest*, com pequenas alterações. O resultado obtido está apresentado na Tabela 4. Cada coluna representa um teste e cada linha representa a ordem na qual o processo com determina quantidade de bilhetes terminou. Por exemplo, o processo com 57 bilhetes foi o quarto a terminar no teste 1.

Foram alocados vários processos, cada processo inicia com uma variável na ordem de grandeza de 10^8 e fica a decrementando. Quando essa variável chega a 0, o processo morre. Com os testes podemos observar que as prioridades em geral são respeitadas e o escalonador funciona conforme o esperado. Para realizar o teste, cada vez que um processo morria, o número de tickets desse processo era printado na função `exit`.

Teste 1	Teste 2	Teste 3	Teste 4	Teste 5	Teste 6	Teste 7	Teste 8	Teste 9	Teste 10
60	60	59	60	60	60	60	59	60	60
59	58	60	58	59	59	59	60	59	58
58	59	58	59	58	58	58	57	58	59
57	56	56	57	56	57	57	55	57	57
56	57	57	56	57	56	55	58	56	56
55	55	55	55	54	55	56	56	54	55
54	54	54	53	55	54	53	51	55	52
53	53	53	54	53	53	54	50	52	54
51	51	52	52	52	52	52	49	51	53
52	50	51	51	49	51	51	54	53	51
49	52	50	50	51	49	49	48	49	49
50	48	49	48	50	50	50	53	50	48
48	49	48	49	47	48	48	46	48	50
46	46	47	47	48	47	47	52	47	45
47	47	46	45	46	46	46	45	46	44
45	45	45	46	45	45	43	44	45	46
44	44	44	43	43	43	44	47	44	47
43	43	43	44	44	44	45	41	43	41
42	41	42	42	42	42	42	42	42	40
41	42	40	41	41	41	41	40	41	38
40	39	41	39	40	40	40	39	40	43
39	40	39	40	39	39	38	43	39	42
38	38	38	38	38	38	39	37	38	39
36	37	36	36	37	37	37	38	37	36
37	36	37	37	36	36	36	36	36	37
35	35	34	35	35	35	35	35	35	32
34	34	35	34	34	34	34	34	34	31
33	33	33	33	33	33	33	33	33	33
32	32	32	32	32	32	32	32	31	35
31	31	31	30	31	31	31	31	32	34
30	30	30	31	29	30	30	30	30	30
29	29	29	29	28	29	29	29	29	29
28	28	28	28	30	28	28	28	28	28
27	27	27	27	27	27	27	26	27	27
26	26	26	26	26	26	25	25	26	26
25	25	25	25	25	25	26	27	25	25
24	24	24	24	23	24	24	24	23	24
23	23	23	23	24	22	23	23	24	23
21	22	22	22	22	23	22	22	22	22
22	21	21	21	21	21	20	20	21	21
20	20	20	19	20	20	21	21	20	20
19	19	19	20	19	19	19	19	18	19
18	18	18	18	18	18	18	18	19	18
17	17	17	17	17	17	17	17	17	17
16	16	16	16	16	16	16	16	16	16
15	15	15	15	15	15	15	15	15	14
14	14	14	14	14	14	14	14	14	15
13	13	13	13	13	13	13	13	13	13
12	12	12	12	12	12	12	12	12	12
11	11	11	11	11	11	11	11	11	11
10	10	10	10	10	10	10	10	10	10
9	9	9	9	9	9	9	9	9	9
8	8	8	8	8	8	8	8	8	8
7	7	7	7	7	7	7	7	7	7
6	6	6	6	6	6	6	6	6	6
5	5	5	5	5	5	5	5	5	5
4	4	4	4	4	4	4	4	4	4
3	3	3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	1	1

Tabela 4. Resultados

Referências

Halim, S. and Halim, F. (2013). *Competitive Programming 3*.

Tanenbaum, A. S. (2010). *Sistemas Operacionais Modernos*. Pearson Prentice Hall, 3rd

edition.