Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Sistemas Operacionais 2018.1

Relatório Final

Sérgio Vale Campos
Professor

Antônio Côrtes Rodrigues
Deiziane Natani da Silva
Paula Jeniffer dos Santos Viriato
Semar Augusto da Cunha Mello Martins

Belo Horizonte, 3 de julho de 2018

1. Introdução

Desde o Linux 2.6.23, o escalonador padrão é o CFS - *Completely Fair Scheduler*. Cada *thread* possui uma política de escalonamento associada e uma prioridade de escalonamento estática, ***sched_priority***. O escalonador toma suas decisões com base no conhecimento dessa política e da prioridade estática de todos as *threads* no sistema. Para *threads* em uma das políticas de escalonamento normais (SCHED_OTHER, SCHED_IDLE, SCHED_BATCH), *sched_priority* não é usado (deve ser especificado como 0). Os processos programados em uma das políticas de tempo real (SCHED_FIFO, SCHED_RR) têm um valor de *sched_priority* no intervalo de 1 (baixo) a 99 (alto). Desde a versão 3.14, o Linux fornece uma nova política de escalonamento (SCHED_DEADLINE). Esta política é implementada atualmente usando o GEDF (Global Earliest Deadline First) em conjunto com o CBS (Constant Bandwidth Server).

O intuito deste trabalho é a implementação de um algoritmo para ser usado como uma nova política de escalonamento, sendo esse um híbrido de *Lottery Scheduler* e *Shortest Job First*, chamado neste trabalho de **LSJF**. O escalonador também é um híbrido com Round-Robin, pois faz o processo de um slot de tempo que falta para um dado processo.

2. Abordagem

- ➔ Usaremos a versão **4.15.0-20** do kernel do Linux no **Ubuntu 18.04 LTS** por ser uma das mais recentes versões estáveis.
- ➔ Utilizados os códigos disponibilizados pela plataforma GitHub de desenvolvimento compartilhado para a versão dita acima (**Ubuntu Bionic**)[2].
- ➔ Os códigos relacionados à parte de escalonamento do kernel linux são agrupados no diretório **sched**, e portanto nosso trabalho será restringido à este diretório, composto por 25 arquivos em C, sendo o principal o **core.c**.
- ➔ Os códigos criados se encontram no repositório: https://github.com/PaulaViriato/Lottery-versus-LSJF

3. Desenvolvimento

Embora o Linux implemente várias formas diferentes de escalonamento, ele não possui nativamente as formas de escalonamento que são de interesse para este trabalho. O primeiro passo então foi a criação dos módulos que fazem *Lottery Scheduler* e *Shortest Job First*.

A ideia para a criação do LSJF é atribuir mais tickets para os trabalhos mais curtos que fossem identificados. Os trabalhos mais curtos são identificados pela quantidade de slots de tempo de processamento que faltam para terminar.

O sistema operacional também atribui prioridades às tarefas, e quanto maior prioridade, maior o número de tickets dados àquela tarefa. Assim, tarefas de menor tamanho tem mais prioridade e são privilegiados pelo escalonador LSFJ, apesar de que processos maiores também possuirem probabilidade de serem escolhidos.

A principal característica do *Lottery Scheduler* é a aleatoriedade parcial nas escolhas das tarefas à serem executadas pelo escalonador. Tal característica é herdada pelo LSJF, que basicamente implementa a mais uma delegação de tickets inversamente proporcional ao tamanho do processo. Ou seja, quão menor for o processo, maior será a quantidade de tickets dados a ele, e consequentemente, maior será a probabilidade de tal processo ser escolhido.

Ambos possuem a mesma estratégia de escolha da tarefa a ser escalonada. É escolhido randomicamente um valor (chamado de *winner*) que é, no máximo, a soma das prioridades de todas as tarefas ativas, sendo que no LSJF são somados também os tickets de tamanho. A lista encadeada de tarefas é caminhada, e à cada tarefa um contador (iniciado com zero) é incrementado com suas prioridades (prioridade no sistema e ticket de tamanho). No momento em que o contador ultrapassa o valor de *winner* a tarefa atual da lista encadeada é escolhida, escalonada, e executada pelo sistema.

O grande diferencial do LSJF é o delegador de tickets *lsjf_setticket*. Caso uma nova tarefa seja menor que as restantes, mais tickets são dados para ela. E no caso de ela ser maior, são dados mais tickets para todas as outras tarefas, como é mostrado nos códigos e comentários abaixo.

Código 1: funcionamento do *Lottery Schedule*

```
1  void __lottery (){
2      int counter, winner;
3      struct task *current;
4
5      // enquanto existem tarefas ativas...
6      while (lottery_priorit_total > 0) {
7          counter = 0;
8
```

```
        /* winner representa o valor que marca
         * a tarefa que sera executada */
        winner  = rand()%lottery_priorit_total;

        // tarefa cabeca (primeira tarefa)
        current = lottery_task;

        // enquanto houver tarefas...
        while (current) {
            // contador marca as prioridades
            counter += current->prio;

            /* a tarefa que estiver no momento
             * que o contador ultrapassar o
             * winner e escalonada e escolhida */
            if (counter > winner){ break; }
            current = current->next;
        }

        // tarefa e executada
        execute_task (current);
    }
}
```

Código 2: funcionamento do *LSJF Schedule*

```
void __lsjf (){
    int counter, winner;
    struct task *current;

    // enquanto existem tarefas ativas...
    while (lsjf_ticket_total > 0){
        counter = 0;

        /* winner representa o valor que marca a
         * tarefa que sera executada */
        winner  = rand()%lsjf_priorit_total;

        // tarefa cabeca (primeira tarefa)
        current = lsjf_task;

        // enquanto houver tarefas...
        while (current) {
            // contador marca as prioridades e quao menor e
            counter += (current->prio*imp_prio)+current->ticket;

            /* a tarefa que estiver no momento
             * que o contador ultrapassar o
             * winner e escalonada e escolhida */
            if (counter > winner){ break; }
            current = current->next;
        }

        // tarefa e executada
        execute_task(current);
    }
}
```

Código 3: funcionamento do *lsjf_setticket*

```
/* esta funcao faz com que as tarefas recebam mais
 * tickets o quanto forem menores que as outras,
 * e assim tenham mais chance de serem escalonadas */

void lsjf_setticket (struct task *t){
    int check = 0;
    struct task *aux = lsjf_task;

    // enquanto houver tarefas...
    while (aux){
        /* verifica se a nova tarefa e menor ou
         * igual. Se igual fica com a mesma quanti-
         * dade de tickets, se menor recebe a mes-
         * ma quantidade +1. */

        if (t->prev_time <= aux->prev_time){
            check = 1;
            if (t->prev_time < aux->prev_time){
                t->ticket = aux->ticket +1;
            } else { t->ticket = aux->ticket; }
        }
        aux = aux->next;
    }

    /* caso seja maior que todas, recebe um
     * ticket e todas as tarefas aumentam em
     * um ticket. */

    if (check == 0){
        t->ticket = 1;
        if (lsjf_task != NULL){
            aux = lsjf_task;

            while (aux){
                aux->ticket ++;
                lsjf_ticket_total ++;
                lsjf_priorit_total ++;
                aux = aux->next;
            }
        }
    }
}
```

4. Testes

Testamos os escalonadores Lottery e LSJF por meio de simulações de diferentes tarefas, para observar o sorteio e escolha de cada uma delas. Os escalonadores criados não foram instalados em sistemas operacionais, mas somente simulados.

Os testes têm como base observar o comportamento individual dos escalonadores, além de realizar uma comparação entre os dois. Para cada tipo de teste, temos o gráfico do desempenho do Lottery e do LSFJ e a análise.

As tarefas (tasks) testes em cada um dos casos foram feitas com um número arbitrário de slots de tempo restantes, e isto foi feito para observar as principais características que o escalonador deveria ter.

O eixo y dos testes indicam quantos slots de tempo precisam ser feitos para uma tarefa ser concluída. O eixo x representa o decorrer do tempo que passou. Cada tarefa é representada por uma linha de diferentes cores. Os gráficos representam uma situação teste, onde todas as tarefas estão no escalonador e uma deve ser escolhida a um dado momento.

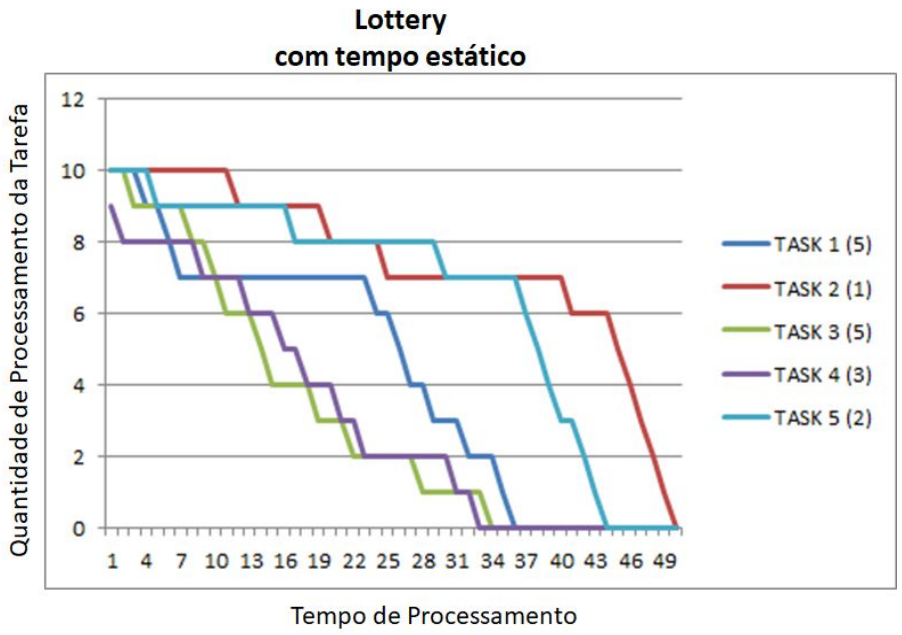
Assim, todas as linhas permanecem constantes ou caem, pois ou não foram escolhidas e não estão trabalhando (e ficam constantes), ou foram escolhidas e realizam algum processamento (e assim caem o número de slots de tempo necessários para sua conclusão).

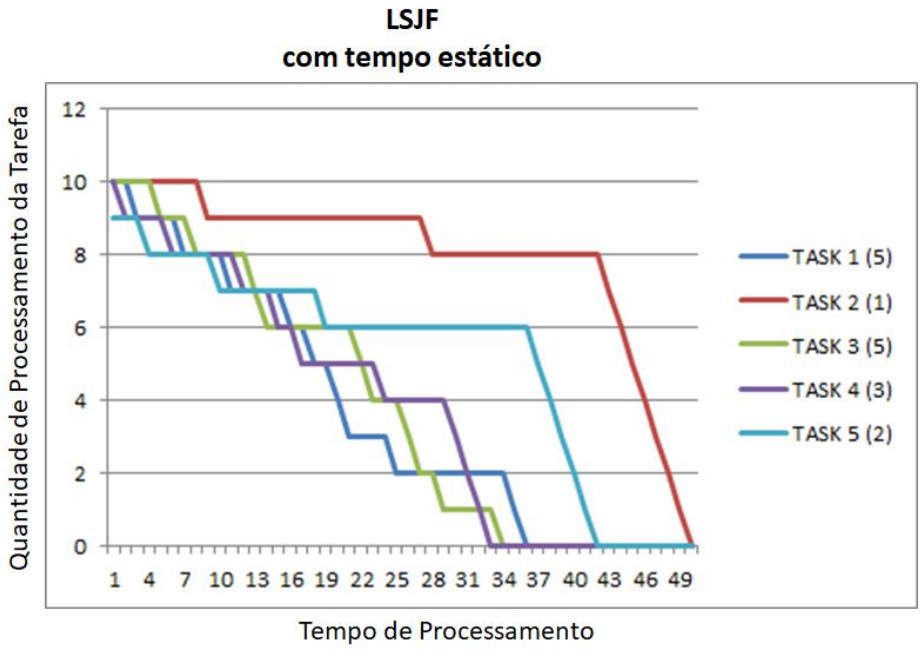
Tarefas também possuem prioridades que o sistema as atribui. As escolhas também são feitas considerando essa prioridade. Cada tarefa tem sua prioridade dada entre parênteses em sua respectiva legenda.

4.1 Teste para tarefas de prioridade variada e tamanhos iguais

Para este teste, o fator que começa mais importante é a prioridade que o sistema operacional atribui para cada tarefa. No Lottery vemos que as Tasks 1, 3 e 4 caem rapidamente no ínicio, pois são as que possuem mais prioridade, e tem maior chance de serem sorteadas. Tasks 2 e 5 esperam o maior tempo e são processadas quase totalmente ao fim, 5 acabando antes por ter uma vantagem sobre 2.

No LSJF temos um fenômeno similar, mas a medida que o tamanho das tarefas cai, a chance delas serem sorteadas também aumenta, pois o escalonador prioriza tarefas menores. Assim, temos um efeito acentuado nas tarefas 1, 3 e 4, que acabam mais rápido que o Lottery, e 2 e 5 têm mais processamento ainda no fim.

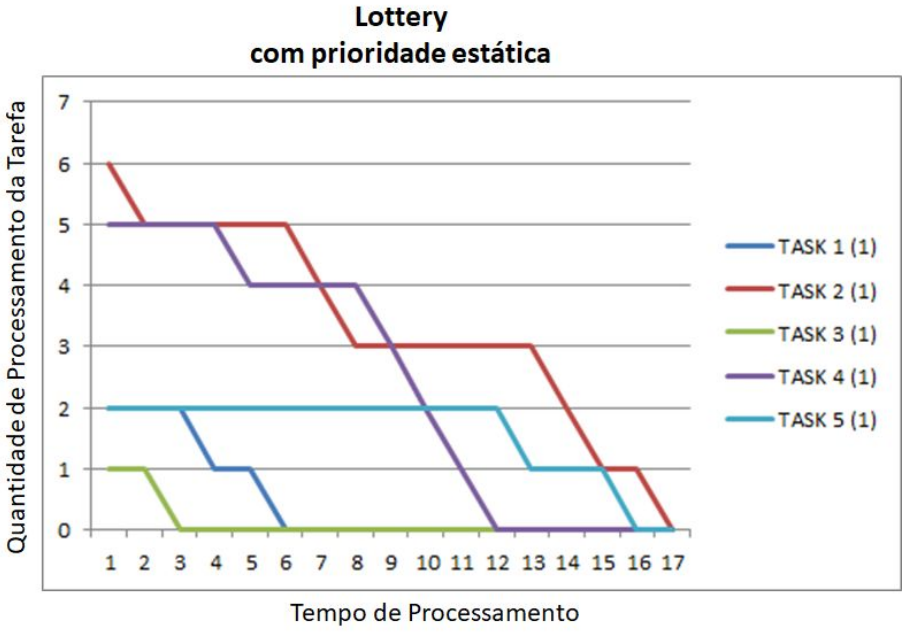


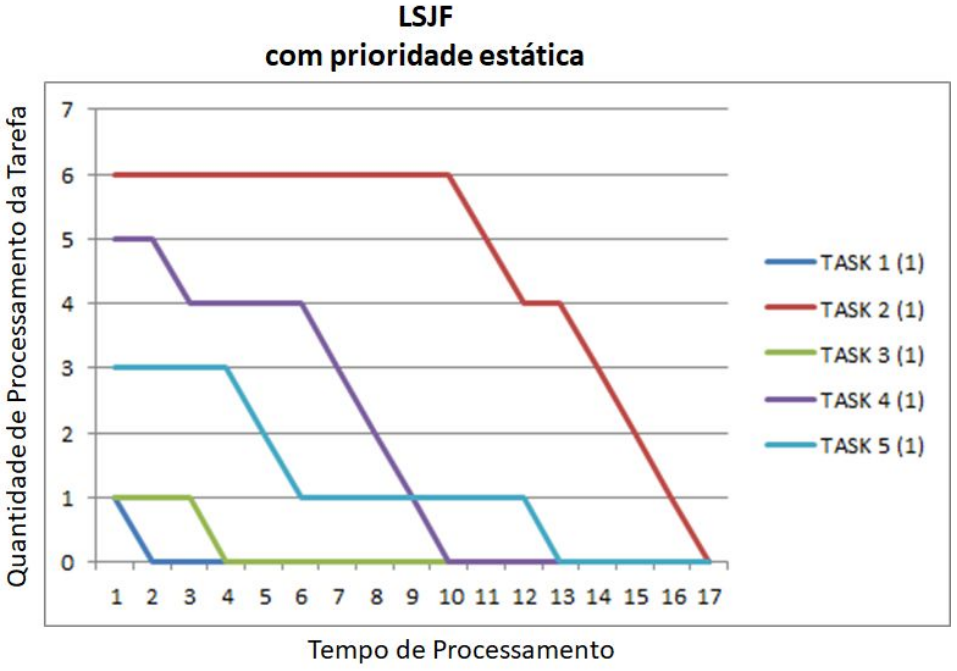


4.2 Teste para tarefas com prioridades iguais e tamanhos variados

Nestes testes podemos verificar a preferência que o LSJF tem por tarefas que estão mais perto de acabar, e existe a tendência de tarefas indo acabando por ordem crescente. Isto não sempre ocorre porque, afinal, o escalonador tem um sorteio da próxima tarefa, e uma tarefa maior pode ser escolhida. O algoritmo funciona exatamente como esperado.

O Lottery também acaba em ordem crescente, mas vemos que existe uma tempo maior para as tarefas acabarem. Os sorteios são aleatórios, e os pequenos acabam antes por precisar de menos sorteios.



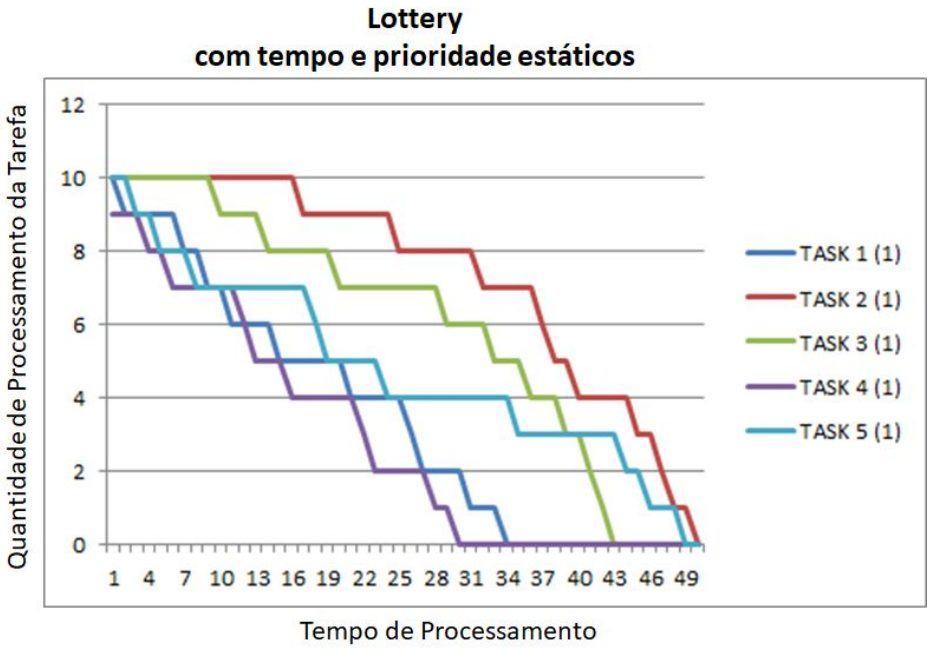


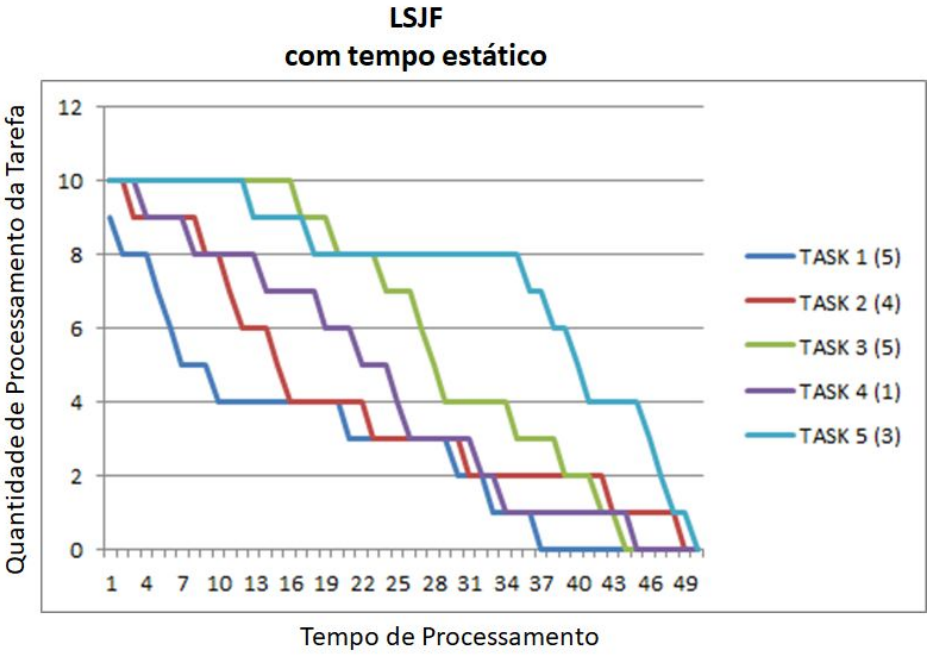
4.3 Testes de prioridades e tamanhos constantes

Neste teste vemos que todos as tarefas começam com propriedades iguais. No Lottery, as escolhas se tornam aleatórias pois as prioridades são iguais, e vemos que os primeiros escolhidos tendem a acabar primeiro, mas esta tendência não acontece de maneira muito rápida.

No LSJF as tarefas que são escolhidas primeiro tendem mais rápido a acabar primeiro, porque este escalonador favorece as tarefas que ficam menores, e este efeito só aumenta com mais escolhas. No Lottery, as tarefas menores acabam

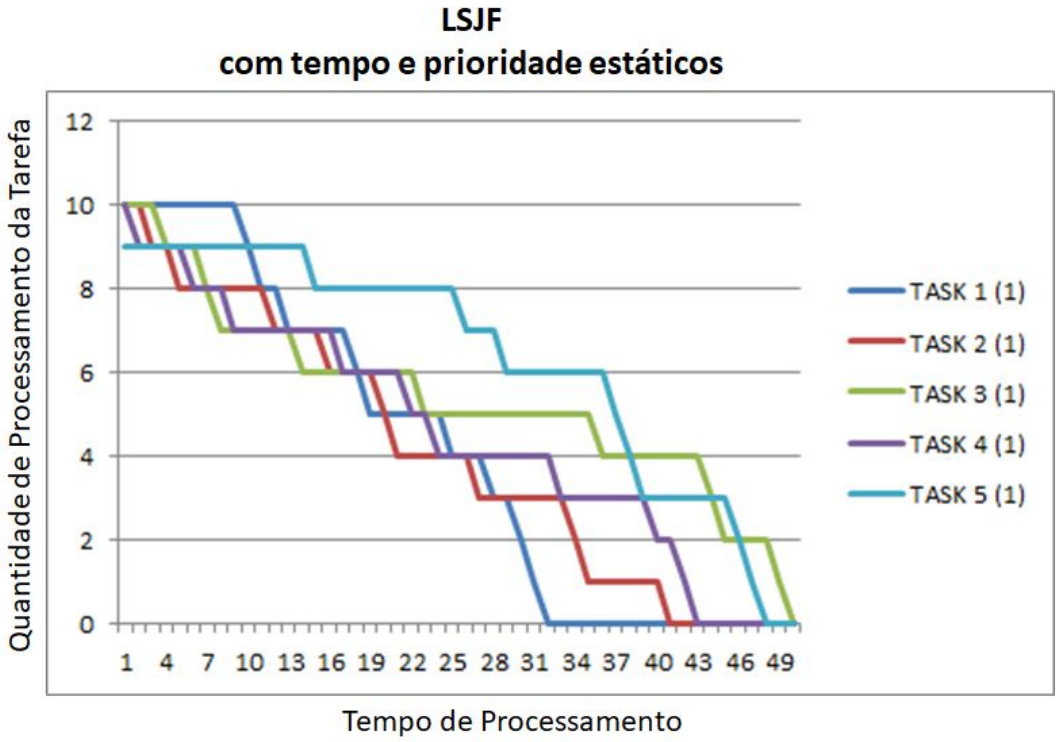
primeiro porque precisam de menos sorteios, mas aqui as menores são de fato privilegiadas.

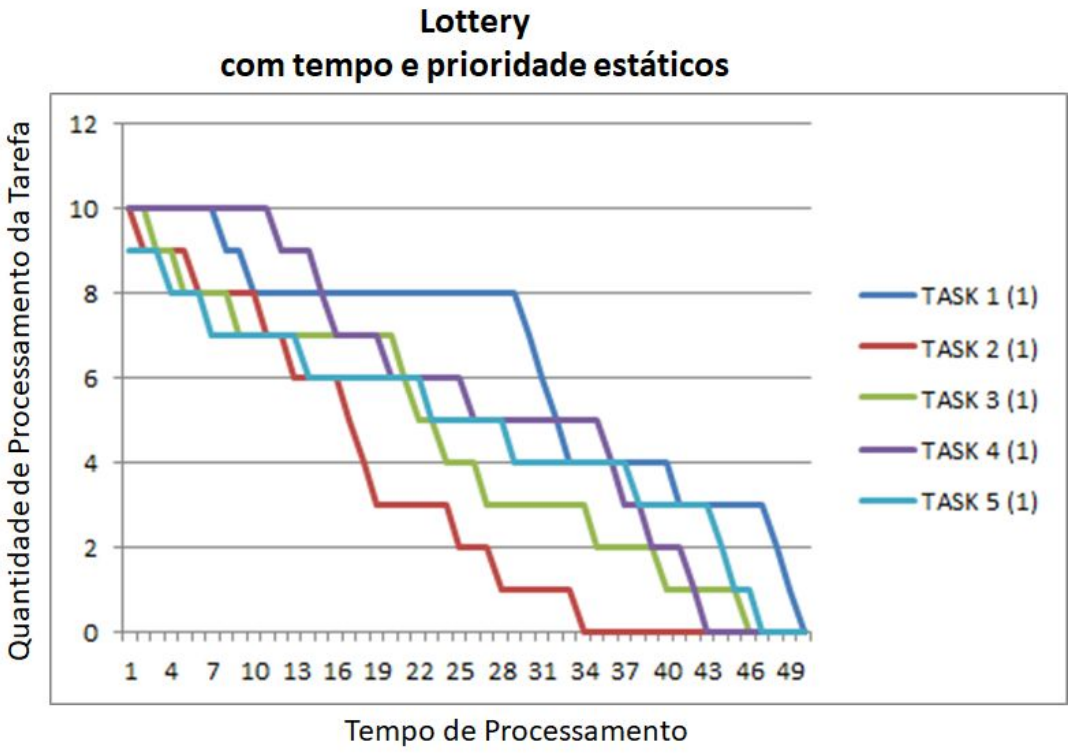




Aqui temos a mesma situação mas um outro teste. Observamos que existe uma tendência de beneficiar tarefas que já tiveram uma maior quantidade processada, mas que em geral ocorre um equilíbrio maior, e o LSFJ teve sorteios mais variados.

O Lottery neste novo teste também ficou mais balanceado, o que é mais esperado para prioridades fixas.



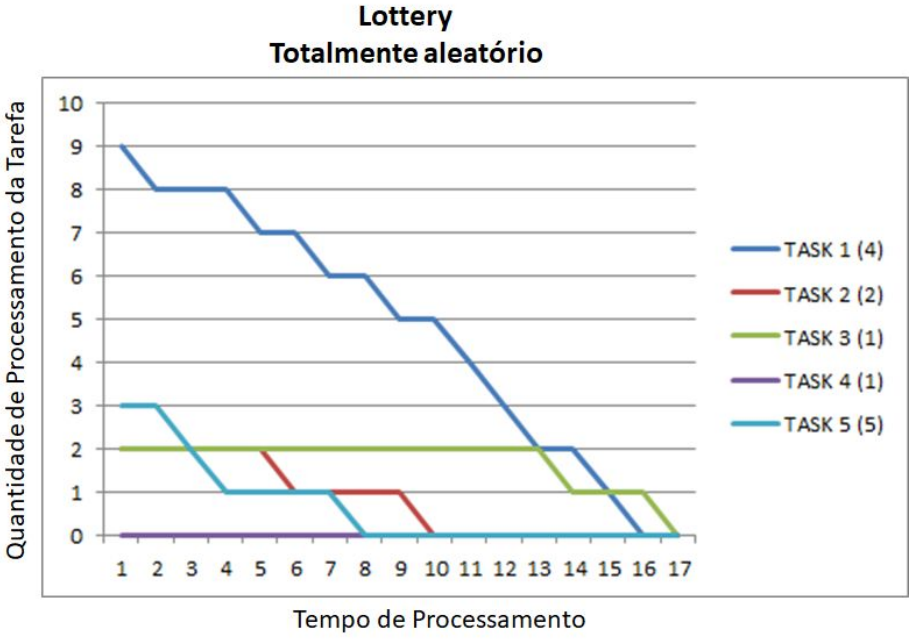


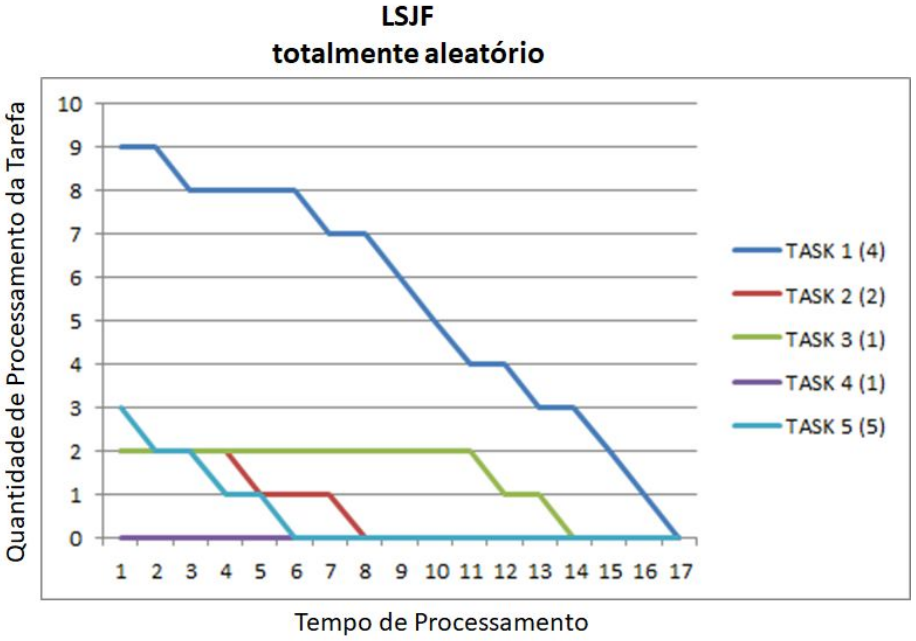
4.4 Teste para tarefas de prioridade e tamanhos variados

O Lottery normal considera somente as prioridades que o sistema atribui, e não observa o tamanho das tarefas. Como a escolha é aleatória, existe uma tendência de tarefas pequenas acabarem antes, pois requerem menos sorteios para acabarem.

Mesmo assim, podemos observar que a Task 1 que era bem maior que as demais e foi completada antes de Task 3, que possuía prioridade bem menor.

O LSFJ força esta vantagem dos menores serem mais escolhidos devido a sua política de escolheres mais os trabalhos menores. As Tasks 1 e 3 se mostram interessantes de novo, pois no fim vemos que 1 foi priorizada (pela importância e pelo sorteio) por um tempo considerável, até que 3 foi sorteada e concluída.





5. Análise

O escalonador funcionou exatamente como esperado para as diferentes situações. Em todos os testes ele tendeu a priorizar os parâmetros que nos eram interessantes: a prioridade atribuída pelo sistema operacional e tamanhos menores das tarefas; mas por fazer sorteios essas prioridades não foram sempre seguidas. Isto permitiu que tarefas de menor preferência (maior tamanho) também executassem, e isto previne *starvation*.

Quando as prioridades são iguais, privilegiar tarefas menores ajuda para terminar mais processos, e ter alguma política que não aleatoriedade para eles.

Algo que pudemos notar é que o Lottery acaba naturalmente concluindo tarefas menores primeiro, já que elas precisam de ser escolhidas menos vezes. Acrescentar Shortest Job First concretiza este efeito e faz com que processos menores acabem bem mais rápido.

A aleatoriedade observada nos testes foi algo muito interessante que de fato mostrou que diferentes processos podem ser beneficiados e serem escolhidos. Talvez funcionando em um sistema verdadeiro o escalonador sofresse por deixar muitas tarefas grandes acumularem, já que a aleatoriedade não seria tão favorável com vários processos chegando; mas a introdução de um *aging* nos processos levaria o LSJF a ser um sistema consideravelmente robusto.

6. Conclusão

O trabalho foi desenvolvido com muita dificuldade, não houve nada simples sobre o desenvolvimento. O parte prática é distante do curso teórico, gerando complicações para alunos que não têm experiência na área de sistemas operacionais.

Mesmo com todas as adversidades, a experiência mostrou as complicações de modificar um sistema operacional, que é extremamente complexo. O trabalho pôde ser feito e os resultados alcançados foram extremamente bons, considerando que o grupo tinha nenhuma experiência na área.

Foi interessante juntar caraterísticas de políticas diferentes em um novo escalonador, que possui diferentes prioridades. A característica do LSJF de juntar aleatoriedade com algumas prioridades levou a um escalonador com boas propriedades e rendimento.

Referências

[1] *Linux Programmer's Manual* . Disponível em: <https://manpages.debian.org/jessie/manpages/sched.7.en.html>

[2] *Repositório da Versão Linux Ubuntu Bionic (18.04).* Disponível em: <http://kernel.ubuntu.com/git/ubuntu/ubuntu-bionic.git/>

[3] Operating Systems: Three Easy Pieces. Disponível em <http://pages.cs.wisc.edu/~remzi/OSTEP/>